Rudolf Pecinovský

Návrhové vzory

33 vzorových postupů pro objektové programování

Zefektivněte své programování a snižte pravděpodobnost chyb

Popisy a výklad řešení prakticky pro všechny situace

Výklad nezávislý na programovacích jazycích, příklady v jazyce Java

press

CPRESS

Rudolf Pecinovský

Návrhové vzory

Computer Press
Brno
2013

Návrhové vzory

Rudolf Pecinovský

Odborná korektura: Jaroslava Pavlíčková
Obálka: Martin Sodomka
Odpovědný redaktor: Václav Kadlec
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-1582-4

Vydalo nakladatelství Computer Press v Brně roku 2013 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 16 694.

Dotisk 1. vydání

ALBATROS MEDIA a.s.

Mé ženě Jarušce a dětem Štěpánce, Pavlínce, Ivance a Michalovi

Rudolf Pecinovský, 2007

Stručný obsah

Obsah

ČÁST 4

Optimalizujeme rozhraní

ČÁST 5

Vytvořte to univerzální

ČÁST 8

Přílohy

Poděkování

Vím, že se v českých knížkách většinou neděkuje, ale tahle silná kniha je spojena s tolika pomocníky a tolika oběťmi lidí z mého okolí, že bych měl velkou újmu na duši, kdybych tak neučinil.

Chtěl bych především nesmírně poděkovat své ženě Jarušce, která byla po celou dobu mojí největší oporou a která si za dobu mého psaní vysloužila již nejednu svatozář. Nemenší poděkování patří i dětem, které se mi také snažily v rámci svých možností pomáhat: testovaly programy nebo za mne zařizovaly nejrůznější záležitosti, abych měl rozumný klid na psaní.

Na vylepšování textu knihy se ale podílela řada dalších lidí. Mezi nimi musím poděkovat především manželům Pavlíčkovým, kteří knihu velmi podrobně přečetli, upozornili mne na nejrůznější nesrovnalosti v textu a odchylky mezi popisovanou a skutečnou podobou doprovodných programů a kteří se ji na závěr uvolili ještě jednou podrobně pročíst a zlektorovat. Naše debaty o správné interpretaci některých zásad moderního programování a o některých vlastnostech popisovaných návrhových vzorů byly občas poměrně vášnivé.

Knihu četla a připomínkami doplňovala i řada dalších lidí, mezi nimiž bych jmenoval Frantu Hunku a Milana Šedého, jejichž soubory s přípomínkami byly také velmi podrobné.

Rád bych touto cestou poděkoval i Michaelu Köllingovi a jeho spolupracovníkům, jejichž myšlenky mne přivedly k nové metodice výuky a jejichž vývojový nástroj *BlueJ* realizaci takto koncipované výuky vůbec umožnil.

Musím vyjádřit svůj velký dík také veškerému osazenstvu firmy Amaio Technologies. Tito lidé mne k Javě přivedli a po celou dobu přípravy knihy mne všestranně podporovali. Upozorňovali mne na zajímavé články a oponovali některé programy. Bez jejich podpory by kniha nevznikla.

Na závěr nesmím zapomenout ani na Václava Kadlece z nakladatelství Computer Press, který dostal moji knihu na starost a který si se mnou užil, když se termín odevzdání neustále vzdaloval. Díky jeho trpělivosti a vstřícnosti se kniha po několika odkladech konečně dostala do stavu, ve kterém ji otevíráte.

Úvod

Znalost základních návrhových vzorů a schopnost je efektivně využívat ve svých programech patří ve světě k povinné výbavě zkušeného programátora. Knihy, které tuto oblast vysvětlují, patří k trvalým bestsellerům. V našich školách a programátorských kurzech se však tato problematika příliš neučí a řada programátorů (a to i těch, kteří se považují za zkušené) o existenci návrhových vzorů dokonce ani netuší.

Před časem u nás vyšel překlad [10] knihy [16], která je dlouhodobým světovým bestsellerem a základní biblí návrhových vzorů, na niž se téměř všechny ostatní učebnice návrhových vzorů odvolávají (přesněji nepotkal jsem takovou, která by tak nečinila). Přeložená publikace se však u nás setkala s podivuhodným nezájmem. Nevím, zda to bylo ne zcela vydařeným překladem[1] nebo zda byl na vině styl příručky, který pro průměrného programátora není příliš čtivý, anebo zda se na nezájmu podepsala i ignorace tohoto tématu ze strany vyučujících. Vypadá to zkrátka tak, že návrhové vzory u nás netáhnou.

Čísla v hranatých závorkách označují pořadí dané knihy v seznamu literatury uvedeném v příloze *Seznam doporučené a nedoporučené literatury* na straně 517.

Kniha [16] se v originále jmenuje *Design Patterns* s podtitulem *Elements of Reusable Object-Oriented Software*. Poprvé vyšla v roce 1995 a napsala ji čtveřice autorů, která zanedlouho na to dostala přezdívku *Gang of four* (banda čtyř) – ve zkratce GoF. Pod touto zkratkou se na jejich publikaci řada ostatních příruček odvolává, aby bylo zřejmé, že se odvolávají právě na ni, a ne na nějakou z mnoha dalších knih, které mají termín *design patterns* v titulu. Nebudu se proto odlišovat, a budu-li se někde odvolávat na tuto publikaci, také ji označím zkratkou GoF.

Vím, že je proto ode mne troufalé pokoušet se napsat další knihu, která by se věnovala této nesmírně důležité, avšak u nás stále opomíjené problematice, ale jako nenapravitelný optimista stále doufám, že návrhové vzory přece jenom získají v povědomí našich programátorů místo, které jim náleží. Přispěje-li k tomu i tato kniha, budu nesmírně potěšen.

[1] Problémem českého překladu této publikace je bohužel to, že překladatel nebyl programátor, a na překladu je to často vidět. Přiznávám, že já jsem se po prvních kapitolách doprovázených nepublikovatelnými výkřiky zbaběle uchýlil k anglickému originálu. Pokusím se proto na rozdíl od běžných zvyklostí prezentovat látku tak, abyste uvedenou příručku-bibli nepotřebovali.

Koncepce knihy

Co se mi na většině příruček návrhových vzorů nelíbí

Většina příruček zabývajících se problematikou návrhových vzorů probírané návrhové vzory pouze vyjmenuje (nejlépe podle abecedy) a u každého uvede jeho základní princip a jeden či dva příklady jeho použití. V některých jejich autoři ještě učeně pohovoří o možných důsledcích nebo o možné zastupitelnosti či spolupráci s jinými vzory.

Zkušenost ukazuje, že takovýto přístup řadě programátorů nestačí. Takovýto výklad pro ně často bývá příliš abstraktní. Potřebovali by vysvětlit řadu konkrétních otázek a rozšířit množinu příkladů, na nichž se látka demonstruje a na nichž si ji pak mohou sami vyzkoušet.

Mnozí programátoři přiznávají, že výše zmiňovanou „bibli“ GoF sice ve své knihovně mají, ale její vysvětlení příliš nechápou. Připadá jim jako kniha, kterou psali teoretici pro teoretiky. Svoji snahu o pochopení návrhových vzorů z této příručky přirovnávají k pokusu naučit se matematiku studiem sbírky vzorců.

Kniha je záznam rozhovoru

Koncipoval jsem proto tuto příručku jako rozhovor mezi zkušeným a začínajícím programátorem. Jeho inspirací byly rozhovory, které jsem nad daným tématem vedl se svými dětmi, s žáky a studenty, kteří navštěvují mé lekce programování, i s profesionálními programátory navštěvujícími mé kurzy, v nichž se přeškolují z klasického programování na programování objektově orientované.

Příklady, na nichž výklad stojí

Celý výklad jsem se snažil ilustrovat na takových příkladech, které jsou na jednu stranu dostatečně jednoduché, takže probíraná problematika se neztrácí v šumu ostatních příkazů, ale které na druhou stranu nebudou jen nějaké AHA-příklady, jež pouze demonstrují princip vzoru a chod programu simulují prostřednictvím tisků na standardní výstup (i když se k nim, pravda, občas z nedostatku fantazie také uchýlím).

Pokusím se, aby většina programů v příkladech byla maximálně praktická, abyste z nich mohli načerpat nějakou inspiraci pro své vlastní programy.

Neučím jen návrhové vzory, učím moderní programování

Tato kniha nechce být pouze výčtem základních návrhových vzorů, ale chce být komplexní učebnicí současných zásad objektově orientovaného programování (to jsem se ostatně pokusil naznačit i v jejím názvu). Zásad, o kterých se běžné učebnice většinou nezmiňují (přiznejme si, že se také většinou nejedná o učebnice programování [byť se to jejich titul snaží naznačit], ale pouze o učebnice syntaxe některého jazyka).

V řadě případů autoři učebnic v demonstračních příkladech tyto zásady dokonce porušují. Knih s různými, často do nebe volajícími prohřešky autorů bychom na trhu našli více. (Doufám, že se mezi ně časem nezařadí kniha, kterou právě čtete.)

Nebudu zabíhat do detailů

Na druhou stranu se v knize nesnažím o podrobný rozbor všech probíraných návrhových vzorů se všemi jejich vzájemnými vazbami a různými důsledky – to by musela být daleko tlustší. Chci pouze čtenáře seznámit se základními návrhovými vzory tak, aby pochopil jejich podstatu a princip a dokázal je později využít ve svých programech. Detailní rozbory ponechávám akademičtěji orientovaným učebnicím – např. GoF.

Citace definicí z GoF

Na počátku každé kapitoly je stručná charakteristika vzoru, kterému se daná kapitola věnuje. Protože jsem se již několikrát setkal s tím, že studenti ode mne chtěli vedle mých volných popisů i přesné definice uváděné v GoF, doplnil jsem u vzorů uváděných v GoF do poznámky pod čarou i jejich originální definici a její překlad (nepřebíral jsem jej z [10], ale pokusil jsem se vytvořit vlastní). Tyto citace by vám mohly pomoci v orientaci při pročítání některých článků týkajících se návrhových vzorů uvedených v GoF.

Otázky

Charakter otázek v knize

Kniha představuje záznam fiktivního rozhovoru se 625 otázkami. Oproti jiným příručkám s otázkami a odpověďmi však v této knize nenajdete klasické dotazy, které nastolí problém, jenž je pak v odpovědi vyřešen. Obsah této knihy má opravdu simulovat zaznamenaný rozhovor, takže mezi otázkami najdete i výplňové otázky a na druhou stranu otázky, které již samy obsahují řešení problému a tazatel se pouze ubezpečuje, že toto řešení je správné.

Výhody:

Koncepce knihy psané jako záznam rozhovoru má pro čtenáře několik výhod:

– výklad se lépe sleduje

- Udržuje jej daleko lépe „v obraze“ a umožňuje mu tak lépe sledovat výklad. Čtenáři mých minulých rozhovorových knih mi dokonce psali, že je potěšilo, když tazatel pokládal otázku, která je v průběhu čtení předchozího odstavce napadla také.

– nutné odbočky méně ruší

- Když někdy potřebuji vysvětlit něco, co přímo nesouvisí s probíraným tématem, mohu vás daleko snadněji navigovat, takže neztratíte nit hlavního výkladu, což bývá u klasicky koncipovaného výkladu problém.

– čtení stylem nádech – výdech

- Otázky přesně oddělují části, které je třeba přečíst jako jeden celek. Soukromě označuji způsob „konzumace“ takovéhoto textu termínem *nádech – výdech*. Při čtení odpovědi na otázku čtenář vstřebává informace (nádech), aby se před další otázkou v klidu zastavil a ujasnil si, že vše z předchozí pasáže pochopil (výdech).

Čeština

Proč používám české termíny

Jedním z častých námětů bouřlivých diskusí mezi programátory, resp. mezi učiteli programování, je používání původních a přeložených termínů. Za dlouhou dobu své učitelské praxe jsem si vyzkoušel, že používání původních termínů v začátečnických kurzech není dobré řešení. Začátečníci mívají problémy s pochopením vlastní látky a přidání termínů, kterým nerozumějí (znalost angličtiny u nás stále není na takové úrovni, jakou bychom rádi viděli), jim situaci pouze ztěžuje.

Když na začátečníka vybafnu např. název *singleton*, málokterý bude vědět, co to slovo znamená, a nezbude mu, než si je zapamatovat jako nějaký nový, cizí termín. Když se pak po pár týdnech výuky zeptám, jaké vlastnosti má návrhový vzor singleton, začnou žáci nejprve tápat, který z probraných vzorů to je, a v řadě případů jej zamění s nějakým jiným.

Když naproti tomu použiji pro daný návrhový vzor termín *jedináček*, všichni si jej ihned pevně spojí se svojí představou jedináčka a nejenom že jej i po týdnech správně vyloží, ale navíc i lépe pochopí jeho podstatu.

Prosím proto čtenáře, kteří jsou hrdí na svoji znalost angličtiny, aby se smířili s tím, že budu vycházet vstříc většině, která konstrukce označené českými termíny lépe pochopí a daleko lépe si je zapamatuje. Ti, kteří můj počeštěný výklad nepotřebují, se jistě již dávno poučili z některé z anglicky psaných učebnic (seznam některých z těch, které se staly zdrojem inspirace pro mne, najdete v příloze *Seznam doporučené a nedoporučené literatury* na straně 517).

O anglické termíny nepřijdete

Protože je však programátorský svět veskrz anglický[1], uvedu u každého termínu při jeho zavedení i příslušný anglický ekvivalent. Všem vám pak doporučuji si tento ekvivalent zapamatovat, protože řada českých i slovenských autorů z nejrůznějších důvodů trvá na používání anglických termínů doplněných českými, resp. slovenskými koncovkami.

Použité programovací jazyky

Rozhodování o použitém jazyku

Knihu jsem se snažil napsat maximálně nezávislou na konkrétním programovacím jazyku. Při jejím koncipování jsem přemýšlel nad tím, v jakých jazycích uvádět demonstrační příklady. Volil jsem mezi možností, uvádět všechny příklady v jediném jazyku anebo ukazovat řešení v několika jazycích současně.

Proč jsem zvolil Javu

Protože jsem se bál, že by při příkladech ve více jazycích kniha neúměrně narostla, rozhodl jsem se zůstat u jediného jazyka a naprogramovat všechny příklady v Javě, která je v současné době nejpoužívanějším programovacím jazykem. Protože jsem tuto knihu psal především pro ty, kteří s objektovým programováním začínají, přihrála této volbě i skutečnosti, že Java je ve světě naprosto dominantním jazykem vstupních kurzů programování na univerzitách i středních školách[2].

Text by měl být čitelný i pro programátory v jiných jazycích

Připočteme-li syntaktickou blízkost jazyka C#, který byl vlastně odvozen z Javy, je zřejmé, že se čtením demonstračních programů nebudou mít problémy ani uživatelé tohoto jazyka. Těch pár drobných syntaktických odlišností by jim nemělo ztěžovat porozumění programům.

Zásadní problém v porozumění by neměli mít ani uživatelé ostatních jazyků určených pro programování na platformě .NET, konkrétně jazyků Delphi a Visual Basic .NET. Vzhledem k blízkosti koncepce této platformy s koncepcí platformy Java by jim měla být většina termínů zřejmá. Pouze čtení programů pro ně bude trochu obtížnější, ale snažil jsem se používat pouze jednoduché dostatečně okomentované programy, takže by měly být i pro ně pochopitelné.

---

1 Když jsem po škole nastupoval v akademii, položil mi můj školitel otázku: „Umíte anglicky?“ Než jsem si zformuloval odpověď, která by charakterizovala úroveň mých znalostí, odpověděl si sám: „No ono je to jedno – buďto budete umět anglicky, nebo změníte zaměstnání.“ A totéž platí pro všechny, kteří se chtějí vážně zabývat programováním.

2 V naší republice sice na středních školách v současné době dominuje Delphi, ale učitelé tohoto jazyka většinou objektové programování neučí. Navíc pozice tohoto jazyka neustále slábne.

Byl bych rád, kdyby se z této učebnice mohli poučit i programátoři v C++. Je sice známou pravdou, že základem syntaxe Javy je syntaxe C++, ale způsob přemýšlení v Javě se od způsobu přemýšlení obvyklého u programátorů v C++ přece jenom liší a liší se i řada prvků jazyka. Obávám se proto, že programátoři v C++ budou mít se vstřebáním informací z této knihy větší problémy než programátoři jiných uvedených jazyků.

Komu je kniha určena

„Absolventům" začátečnických programátorských příruček a kurzů

Kniha je určena programátorům, kteří mají základní znalosti objektově orientovaného programování v některém z moderních programovacích jazyků. Nejvýhodnější je pro její studium znalost jazyka Java, ale jak jsem již řekl, stejně dobře ji mohou číst i ti, kteří programují v některém z jazyků určených pro platformu .NET. Jinými slovy: je určena „absolventům" začátečnických příruček programování, resp. absolventům začátečnických kurzů.

Předpokládané znalosti

Předpokládám pouze to, že čtenář zná základní strukturované konstrukce a ví, co jsou to třídy a jejich instance a jaký je rozdíl mezi třídou a rozhraním, a umí je ve svých programech rozhraní využívat. Ví, co jsou to atributy a metody, jaký je rozdíl mezi konstruktorem a běžnou metodou a jaký je rozdíl mezi atributy a metodami třídy (používá se pro ně označení *statické*) a instancí.

Neomezuje se pouze na vzory, ale vysvětluje i obecné zásady moderního programování

Nepředpokládám však žádné hluboké znalosti. Ze zkušenosti vím, že řada kurzů objektově orientovaných jazyků toho o OOP stejně více nenaučí a řada programátorů používajících objektově orientované jazyky píše v těchto jazycích i nadále staré dobré strukturované programy. Proto se v této učebnici nehodlám omezit na pouhý výklad principů jednotlivých vzorů, ale chtěl bych vás seznámit i s některými obecnějšími zásadami moderního objektově orientovaného programování.

Doprovodné příklady

Kde je najdete

Všechny příklady, které budeme v této knize probírat, a to jak ty, u nichž budu uvádět jejich kompletní výpis, tak ty, u nichž vám tu ukážu jenom jejich klíčové části nebo se o nich dokonce pouze zmíním, najdete na adrese `http://knihy.pecinovsky.cz/vzory`. Všechny doprovodné materiály si můžete stáhnout také z adresy http://knihy.cpress.cz/k1348.

Diakritika

Na této adrese vás očekávají dva soubory: první bude obsahovat programy, v jejichž definicích je použita diakritika obdobně, jako ji budu používat v programech, které najdete v textu knihy[1]. Druhý soubor bude označen zkratkou `bhc` (= bez hacku a carek) a bude obsahovat tytéž programy, ale zbavené veškeré diakritiky.

---

[1] Doufám, že mi to zapřisáhlí „nepoužívači diakritiky" odpustí, ale při psaní té záplavy různých textů mám používání diakritiky tak hluboko pod kůží, že mi činí problémy se při programování hlídat, abych ji nepoužil. Je to obdobný problém, s jakým se potýkají programátoři, kteří jsou z programů a e-mailů zvyklí diakritiku nepoužívat a pro změnu jim činí obtíže psaní běžného textu.

Struktura programů

Všechny programy mají jednotnou strukturu, jejíž jednotlivé části jsou odděleny řádkovými komentáři, podle nichž se lze v programu rychleji orientovat. V doprovodných programech najdete vždy všechny oddělující řádkové komentáře. Nechávám si je v programech pro případ, že bych se později k programu vrátil a chtěl něco doplnit. Kromě toho se podle těchto komentářů v programu mnohem lépe orientuji a rychle poznám, které sekce daný program neobsahuje – např. že nemá definován konstruktor.

Z výpisů určených pro publikaci v této knize jsem však nepotřebné řádkové oddělovače odstranil, abyste mi nevyčítali, že musíte platit za zbytečně potištěný papír. Nelekněte se proto, že programy stažené z webu budou vypadat maličko jinak než ty, které najdete v textu knihy.

Úvodní řádky dokumentačních komentářů

Dokumentační komentáře tříd a metod začínají řádkem hvězdiček. Vyzkoušel jsem, že začátečníkům takovéto výrazné oddělení jednotlivých metod napomáhá k lepší orientaci v programu. Bude-li tato moje konvence ty zkušenější z vás obtěžovat, určitě je ve svém editoru dokážou pomocí jednoduchého regulárního výrazu hromadně odstranit.

Konvence názvů

V doprovodných programech začínají názvy všech rozhraní písmenem `I` (např. `IPosuvný`) a názvy všech abstraktních tříd písmenem `A` (např. `APosuvný`). Vím, že v Javě se tato konvence standardně nepoužívá (vynechám-li programy firmy *Microsoft*), ale ze své praxe mám vyzkoušeno, že tato konvence usnadňuje začátečníkům orientaci v projektech.

Většinou používám v názvech tříd celá slova, ale někdy mi takto utvořené názvy připadají příliš dlouhé, takže zvítězí má lenora a použiji v názvech zkratky. Doufám, že i tak zůstávají programy přehledné.

Terminologie

Není jednotná

Terminologie autorů programátorských příruček a lektorů kurzů programování není jednotná. Řada autorů zcela ignoruje českou terminologii a používá jakousi czenglish terminologii, u níž se nemusejí namáhat s vyhledáváním vhodného ekvivalentu, který by začátečníkům pomohl termín pochopit.

Nicméně ani ti, kteří se snaží hovořit na své studenty česky, se v používaných termínech neshodují. Následující pasáž je proto věnována některým termínům, které byste nemuseli znát nebo které jste zvyklí používat poněkud jinak.

Rozhraní × interface

Dva významy termínu *rozhraní*

Termín *rozhraní* se v objektově orientovaném programování používá ve dvou významech:

a) Druh datového typu označovaného v hlavičce klíčovým slovem `interface`.
b) Souhrn informací, které o sobě třída zveřejňuje.

Při svém výkladu jsem dlouho narážel na problém, že jsem neuměl dostatečně jasně odlišit, kdy hovořím o rozhraní ve významu označeném jako *a)* a kdy o rozhraní ve významu *b)*.

S uvedeným problémem jsem dlouho zápasil a nakonec jsem se rozhodl, že budu standardně používat přeložený termín *rozhraní* (v řadě případů se beztak hovoří o rozhraní v obou významech současně), a v případě, když budu chtít zdůraznit, že hovořím o druhu datového typu, uchýlím se k nepřeloženému termínu `interface`, který budu navíc vysazovat „programovým", tj. neproporcionálním písmem.

Podrobněji se o dané problematice rozhovořím ještě jednou v podkapitole *Programovat proti rozhraní* na straně 40.

Idiomy a návrhové vzory

Nebudeme návrhové vzory kastovat

Někteří autoři rozlišují „pravé" návrhové vzory (např. 23 vzorů publikovaných v GoF) a vzory, které podle nich nejsou plnohodnotnými návrhovými vzory, a nezaslouží si proto být mezi ně zařazovány. Označují je jako idiomy, případně používají jiné termíny.

Přidávám se k těm, kteří tvrdí, že typického programátora nezajímá, je-li daný vzor „plnohodnotný", ale zajímá jej, jak mu může daný vzor v jeho práci pomoci. Nebudu proto nijak rozlišovat mezi „plnohodnotnými" a „ne-plnohodnotnými" návrhovými vzory a budu pro všechny používat společný termín *návrhový vzor*.

Méně známé termíny

V dalším textu budu občas používat termíny, které nejsou v počítačové literatuře příliš běžné. Kdo četl moji učebnici Javy, tak je zná. Těm ostatním je nyní pro jistotu raději představím:

Halda

Halda (anglicky heap) je název pro část paměti sloužící k alokaci (uložení) dynamicky vytvářených objektů.

Kontejner

Kontejner je objekt sloužící k uchovávání jiných objektů. Mezi kontejnery patří nejenom klasické dynamické kontejnery, jako např. množina nebo seznam, ale i klasická pole, která jsou statickými kontejnery (statickými proto, že po jejich vytvoření již není možno změnit počet prvků, které se do nich vejdou).

Literál

Literál je konstanta zapsaná v programu svojí hodnotou. Napíšete-li v programu `7` nebo `"Ahoj"`, použili jste literál. Obecně se používání literálů v programu považuje za nevhodné a doporučuje se dávat přednost používání pojmenovaných konstant.

Překrytí metody

Překrytí metody (method overriding) je konstrukce, při níž potomek definuje metodu se stejnou charakteristikou (signaturou, tj. názvem metody a typy jednotlivých parametrů), jako má metoda předka. Při prácí s instancí potomka se pak vždy použije překrývající verze metody, a to i tehdy, vydává-li se instance potomka za instanci předka.

Někteří autoři používají pro tuto konstrukci termín *přepsání* nebo *předefinování* metody. Tyto termíny odmítám používat, protože se tady nic nepřepisuje ani nepředefinovává. Toto není refaktorace. Rodičovská verze metody je pro všechny stále k dispozici.

Prázdný odkaz

Prázdný odkaz je pouze jiný název pro odkaz s hodnotou `null`.

Přetížení metody

Přetížení metody (method overloading) je konstrukce, při níž definujeme uvnitř jedné třídy několik metod se stejným názvem, ale různými sadami typů jejich parametrů.

Správce paměti

Správce paměti je modul virtuálního stroje, který spravuje haldu: alokuje na ní nové objekty a odstraňuje objekty nepoužívané. Řada autorů jej označuje anglickým termínem *garbage collector* (česky popelář nebo uklizeč).

Vektor je alternativní termín pro jednorozměrné pole (odtud také získala svůj název kontejnerová třída `java.util.Vector`), tj. pro pole s jedním indexem (např. `int[]`).

Vlastní instance třídy

Vlastní instance třídy je instance, která je instancí dané třídy a není instancí žádného z jejích předků ani potomků.

Vlastní třída instance

Vlastní třída instance je třída, pro niž je zmiňovaná instance její vlastní instancí.

Zanořený typ

Zanořený typ (`class` – třída, `enum` – výčtový typ, `interface` – rozhraní) je typ, který je definován uvnitř jiného typu. V praxi se používají v drtivé většině případů pouze zanořené třídy. V některých situacích je však výhodné definovat i zanořená rozhraní a výčtové typy (se zanořeným rozhraním se setkáte i v doprovodných příkladech).

– Výhody zanořených typů

Klíčovou vlastností všech zanořených tříd je to, že mohou pracovat i se soukromými členy své vnější třídy. Proto je také v programech zavádíme. V řadě případů představují dokonce jediný smysluplný způsob, jak celou situaci řešit (viz např. kapitolu *Moc se mi v tom nebrab (Iterátor – Iterator)* na straně 203).

Existují tři druhy zanořených typů (jejich podobu ve zdrojovém kódu si můžete prohlédnout ve výpisu Ú.1):

– Vnořené třídy a rozhraní

- Vnořená třída a rozhraní (`embedded/nested class/interface`) jsou typy, jejichž definice jsou umístěné mezi definicemi metod a jsou doplněné modifikátorem `static`. Platí pro ně totéž co pro obyčejné typy. Lze je používat nezávisle na jejich vnější třídě. Jejich zanoření ovlivňuje pouze jejich „oslovování“, tj. název, pod nímž jsou dostupné (a samozřejmě dosažitelnost soukromých členů své vnější třídy).

  Rozhraní (`interface`) a výčtové typy (`enum`) mohou být jenom zanořené. Definujete-li rozhraní a/nebo výčtový typ uvnitř definice jiného typu bez modifikátoru `static`, překladač si jej „iniciativně“ doplní sám. Neuvedení modifikátoru `static` není tedy u těchto zanořených typů považováno za syntaktickou chybu.

– Vnitřní třídy

- Vnitřní třídy (`inner classes`) je definována obdobně jako vnořená, pouze nemá uveden modifikátor `static`. Její instance má skrytý atribut, kterým je odkaz na přidruženou instanci její vnější třídy – bez ní nemůže existovat.

Jak si možná někteří z vás domysleli, výčtový typ (`enum`) ani rozhraní (`interface`) se vám nepodaří definovat jako vnitřní, protože si případný chybějící modifikátor `static` překladač sám „iniciativně“ doplní.

– Lokální třídy

- Lokální třída (`local class`) je definována uvnitř bloku kódu. Jako lokální je možné definovat pouze třídu. Výčtový typ (`enum`) ani rozhraní (`interface`) jako lokální definovat nelze – je to syntaktická chyba.
- Anonymní třída (`anonymous class`) je lokální třída pro jedno použití, tj. jejich instance jsou vytvářeny pouze na jediném místě v programu. Vzhledem k jedinečnosti použití dané třídy není třeba pro tuto třídu vymýšlet nějaký název, ale bývá považováno za výhodnější umístit její definici jakou součást výrazu vytvářejícího její instanci.

Výpis Ú.1: Stručný přehled druhů zanořených typů

```
package rup.česky.vzory._00_úvod;

public class ZanořenéTypy
{
    /** Definice vnořené třídy je uvozena modifikátorem static. */
    protected static class VnořenáTřída {}

    /** Definice vnitřní třídy není uvozena modifikátorem static. */
    public class VnitřníTřída {}

    /** Metoda instance vnější třídy používající vlastní lokální třídu. */
    void metodaLT ( final String par ) {
        class LokálníTřída {}
    }

    /** Metoda instance vnější třídy používající vlastní anonymní třídu. */
    void metodaAT ( final String par ) {
        new Object() {
            public String toString() { return null; }
        };
    }
}
```

Pro ty, kteří by si ještě chtěli připomenout některá základní pravidla práce s jednotlivými druhy zanořených tříd, jsem připravil třídu `ZanořenéTypy2`, obsahující trošku podrobnější verzi, v níž jsou deklarované i metody a která obsahuje i jednoduchý testovací program, jejž si můžete spustit. Definici třídy naleznete ve výpisu Ú.2. V této definici jsou v některých třídách připraveny i zakomentované definice nepovolených druhů atributů a metod, abyste si mohli vyzkoušet, že se při jejich odkomentování překladač opravdu vzbouří.

Výpis Ú.2: Podrobnější přehled druhů zanořených typů

```
package rup.česky.vzory._00_úvod;

import rup.česky.společně.Dbg;

/*****************************************************************************
```

```
 * Třída ZanořenéTypy2 ukazuje definici zanořených tříd a ukázku jejich
 * použití. Využívá metody <code>Dbg.kdoVolá(int,int)</code>, která vrací
 * název volající metody, případně i její třídy.
 */
public class ZanořenéTypy2
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /** Obyčejná metoda instance vnější třídy. */
    void metoda ()  {
        System.out.println("Metoda instance vnější třídy: " + Dbg.kdoVolá(2,0));
        VnitřníTřída vnitřní = new VnitřníTřída();
        vnitřní.instančníMetodaVnitřníTřídy();
    }


    /** Metoda instance vnější třídy používající vlastní lokální třídu. */
    void metodaLT ( final String par ) {
        /** Lokální třída je deklarována uvnitř bloku příkazů. Nesmí mít
         * deklarovány modifikátory přístupu a nesmí mít statické členy.
         * Nesmí používat lokální proměnné. Má-li používat lokální data,
         * musí být deklarována jako konstanty.
         */
        class LokálníTřída {
            void metodaLokálníTřídy() {
                System.out.println( par + " Metoda: " + Dbg.kdoVolá(2,0) );
            }
        }
        LokálníTřída lt = new LokálníTřída();
        lt.metodaLokálníTřídy();
        System.out.println( "Metoda: " + Dbg.kdoVolá(2,0) );
    }


    /** Metoda instance vnější třídy používající vlastní anonymní třídu. */
    void metodaAT ( final String par ) {
        /** Anonymní třída se od lokální liší pouze tím, že nemá jméno. */
        new Thread() {
            public void run() {
                System.out.println( par + "V samostatném vlákně tisknu " +
                    "\n  Metoda: " + Dbg.kdoVolá(2,0) );
            }
        }.start();
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /*****************************************************************************
     * Vnořená třída je uvozena modifikátorem static.
     * Nejsou na ni kladena žádná omezení
     */
    protected static class VnořenáTřída {
        static String sa = "Statický atribut vnořené třídy";
        String ia = "Instanční atribut vnořené třídy";
```

```
        static void statickáMetodaVnořenéTřídy() {
            System.out.println( "Metoda: " + Dbg.kdoVolá(2,0) );
        }
        void instančníMetodaVnořenéTřídy() {
            System.out.println( "Metoda: " + Dbg.kdoVolá(2,0) );
        }
    }


    /*************************************************************************
     * Vnitřní třída není uvozena modifikátorem static, takže je navázána
     * na konkrétní instanci. Nesmí obsahovat statické členy,
     * s výjimkou konstant, jejichž hodnota je známa v době překladu.
     */
    private class VnitřníTřída {
//        static String sa = "Statický atribut vnitřní třídy";
        String ia = "Instanční atribut vnitřní třídy";

//        static void statickáMetodaVnitřníTřídy() {
//            System.out.println( "Metoda: " + Dbg.kdoVolá(2,0) );
//        }
        void instančníMetodaVnitřníTřídy() {
            System.out.println( "Metoda: " + Dbg.kdoVolá(2,0) );
        }
    }


//== TESTY A METODA MAIN ======================================================

    /*************************************************************************
     * Testovací metoda.
     */
     public static void test()
     {
         ZanořenéTypy2 zt      = new ZanořenéTypy2();
         VnořenáTřída vnořená = new VnořenáTřída();
         VnitřníTřída vnitřní = zt.new VnitřníTřída();
         zt.metoda();
         zt.metodaLT( "Lokální: "  );
         zt.metodaAT( "Anonymní: " );
         VnořenáTřída.statickáMetodaVnořenéTřídy();
         vnořená.instančníMetodaVnořenéTřídy();
         vnitřní.instančníMetodaVnitřníTřídy();
     }
     /** @param ppr Parametry příkazového řádku - nepoužité */
     public static void main(String[]ppr){ test(); }/*-*/
}
```

Použité konvence

K tomu, abyste se v textu lépe vyznali a také abyste si vykládanou látku lépe zapamatovali, používám několik prostředků pro odlišení a zvýraznění textu.

Důležité	Texty, které chci zvýraznit, jsou vysazeny **tučně**.
Názvy	Názvy návrhových vzorů jsou stejně jako názvy firem a jejich produktů vysazeny *kurzivou*.
Citace	Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, jsou vysazeny **tučným bezpatkovým písmem**.
Adresy	Názvy souborů a internetové adresy jsou vysazeny **obyčejným bezpatkovým písmem**.
`Program`	Texty programů a jejich částí jsou vysazeny `neproporcionálním písmem`.

Kromě částí textu, které považuji je důležité zvýraznit nebo alespoň odlišit od okolního textu, najdete v textu ještě řadu doplňujících poznámek a vysvětlivek. Všechny budou v jednotném rámečku, který bude označen ikonou charakterizující druh informace, kterou vám chce poznámka předat.

Symbol jin-jang bude uvozovat poznámky, s nimiž se setkáte na počátku každé kapitoly a ve kterých si povíme, co se v dané kapitole naučíme.

Obrázek knihy označuje poznámku týkající se používané terminologie. Tato poznámka většinou upozorňuje na další používané termíny označující stejnou skutečnost.

Píšící ruka označuje obyčejnou poznámku, ve které informace z hlavního proudu výkladu doplňuji o nějakou zajímavost.

ČÁST 1

Zahřívací kolo

V této části se nejprve seznámíme se základními zásadami objektově orientovaného programování a poté si ukážeme jejich aplikaci na několika velice jednoduchých návrhových vzorech, na něž se budu v dalším textu odvolávat.

Tato část je určena především pro programátory, kteří se naučili pouze syntaxi svého objektového jazyka, ale jejich „oblíbený“ vyučující či autor nepovažoval za důležité seznámit je také se základními zásadami objektově orientovaného myšlení.

KAPITOLA 1

Co je a k čemu je návrhový vzor

- **Návrhové vzory a jejich katalogy**
- **Které vzory budeme probírat**
- **Shrnutí – co jsme se naučili**

Co se v kapitole naučíme

V této kapitole se dozvíte, co to jsou návrhové vzory a k čemu vám mohou být užitečné. Zamyslíme se nad obecnými zásadami objektově orientovaného programování a povíme si, jak do nich návrhové vzory zapadají. Zmíníme se také o katalozích návrhových vzorů a o jejich typickém uspořádání.

Návrhové vzory a jejich katalogy

1. Už jsem několikrát slyšel, jak se nějací programátoři baví o jakýchsi návrhových vzorech. Co to vlastně ty návrhové vzory jsou?

Co to je návrhový vzor

Návrhové vzory (anglicky *design patterns*) jsou doporučené postupy řešení často se vyskytujících úloh. Mohli bychom je přirovnat ke vzorečkům v matematice či fyzice. Když se jednou naučíš, že řešení kvadratické rovnice

$$ax^2 + bx + c = 0$$

získáš dosazením do vzorečku

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

už nad tím příště nemusíš přemýšlet. Obdobně je to i s návrhovými vzory. Na rozdíl od matematických vzorečků však do návrhových vzorů nedosazuješ čísla, ale dosazuješ do nich třídy, rozhraní a objekty. Můžeš je považovat za vzorečky, které použiješ při návrhu architektury budoucí aplikace.

2. Takže když budu znát návrhové vzory, budu s řešením dříve hotov než ti, kteří musí ten „vzoreček“ teprve objevit?

Návrhové vzory:

– snižují pravděpodobnost chyb

Přesně tak. Navíc tak výrazně snížíš pravděpodobnost chyb, které bys udělal, kdybys řešení teprve sám vymýšlel. Současně si často ulehčíš budoucí práci, protože návrhové vzory již dopředu počítají s typickými rozšířeními, takže ti budoucí úpravy a rozšíření dají mnohem méně práce a výsledné programy budou navíc spolehlivější.

– vštěpují zásady správného programování

Druhou výhodou je, že ti v průběhu osvojování návrhových vzorů přejdou do krve klíčové zásady objektově orientovaného programování.

Mnozí z těch, kteří k OOP přecházejí z neobjektového světa, tyto zásady poněkud podceňují a dál se snaží programovat tak, jak byli zvyklí doposud. To ovšem přináší u složitějších programů řadu problémů, které často vedou k tomu, že tito programátoři objektově programování zavrhnou jako špatnou metodiku. Přitom si neuvědomují, že skutečná příčina neúspěchu jejich programů spočívá v porušování klíčových zásad OOP, tj. v tom, že neprogramovali skutečně objektově.

Při studiu návrhových vzorů bude před tebou defilovat celá řada příkladů správně uplatněných zásad OOP a jako vedlejší efekt bys měl pochopit oprávněnost těchto zásad a důvody, proč je tyto zásady třeba dodržovat.

– zestručňují a zkvalitňují komunikaci

Znalost návrhových vzorů navíc výrazně usnadňuje komunikaci ve skupinách. Když se zase odvolám na matematiku, pak je jistě jednoduší prohlásit, že musíš zjistit, jestli je diskriminant kladný, než pracně vysvětlovat, co že to chceš zjišťovat a proč.

Obdobně když v geometrii prohlásíš, že patu výšky najdeš pomocí Thaletovy kružnice, je to pro kolegy, kteří vědí, co je to Thaletova kružnice, mnohem jasnější, než kdybys jim začal vysvětlovat, že chceš vytvořit kružnici, která bude mít střed ve stře-

du dané strany a bude procházet jejími kraji, protože odněkud víš, že paty výšek spuštěných z krajních bodů této strany musí na dané kružnici ležet.

– Velké SW firmy jejich znalost vyžadují

Velké softwarové firmy o důležitosti znalosti návrhových vzorů již dávno vědí a programátor, který návrhové vzory neovládá, je pro ně ještě nedostudovaný.

3. Dobře, přesvědčil jsi mne, že bych měl návrhové vzory znát. Kde bych se o nich mohl dozvědět? Existují sbírky těchto vzorců?

Katalogy návrhových vzorů

Existují. Neříká se jim ale sbírky. Spíš se o nich hovoří jako o katalozích vzorů. Nejslavnějším z nich je kniha GoF, kterou jsem zmiňoval již v úvodu (viz stranu 18). Ta popisuje 23 základních, obecně použitelných návrhových vzorů. V dalších letech byly publikovány další vzory, které většinou popisovaly efektivní řešení problémů ve specializovaných oblastech.

Katalogy slouží spíše jako reference

Dopředu bych tě chtěl ale upozornit, že samotné vlastnictví katalogu ti pomůže obdobně jako vlastnictví sbírky matematických vzorců. Dokud se s těmito vzorci nenaučíš pracovat, bude ti sbírka vzorců na nic. Oceníš ji až ve chvíli, kdy už se vzorci pracovat umíš a potřebuješ si pouze osvěžit některé detaily.

4. To je mi jasné. Zajímalo by mne ale, jak se takové programovací vzorečky zapisují.

Zápis návrhového vzoru

Tak jednoduché jako v matematice to bohužel není. Programátoři ještě nevymysleli obecně uznávaný formalismus (a obávám se, že jej hned tak ani nevymyslí), který by byl dostatečně přehledný a výstižný. V zápisu návrhových vzorů se proto používá zakreslení vztahů a postupů pomocí UML diagramů a současně také slovní popis, který dané diagramy doprovází.

5. Obávám se, žes' mi tím vysvětlením příliš nepomohl. Co to jsou ty UML diagramy?

UML diagramy

Většina učebnic návrhových vzorů vysvětluje na počátku UML diagramy. Protože se domnívám, že většina čtenářů již UML diagramy zná, a protože se mi nechce odbočovat od tématu, umístil jsem základní výklad použitých prvků jazyka UML do přílohy *Základy jazyka UML* na straně 511. Kdybys měl zájem o podrobnější studium, zkus si sehnat některou z knih uvedených v příloze *Seznam doporučené a nedoporučené literatury* v podkapitole *Jazyk UML* na straně 512.

6. Prošel jsem si přílohu a získal základní představu o UML. Předpokládám, že si vše v průběhu výkladu ještě procvičím. Takže znovu: jak je to s tou definicí návrhového vzoru?

Struktura definice návrhového vzoru

Základní podoba této definice vychází z GoF, která ve svém katalogu popisuje každý vzor v několika sekcích:

- V první části autoři čtenářům vzor stručně (jednou či dvěma větami) představí, tj. zavedou jeho název a popíší jeho hlavní účel či podstatu.
- Po tomto stručném představení následuje příklad, na němž je možno snadno demonstrovat motivaci, která zavedení vzoru vedla.

- V další části popíší obecnou strukturu vzoru spolu se vztahy mezi jeho jednotlivými složkami a vzájemnou spoluprací těchto složek.
- Pokračuje popis implementace daného vzoru následovaný nějakým konkrétním příkladem této implementace.
- V závěrečných pasážích o daném vzoru autoři rozebírají jeho známá použití a jejich důsledky, možnost náhrady daného vzoru jinými vzory a případnou spolupráci s dalšími návrhovými vzory.

Vzory budou řazeny do logických skupin

Tuto hrubou kostru se pokusím zachovat i v dalším výkladu. Pouze nebudu řadit výklad vzorců podle abecedy, abych se nemusel odvolávat na věci, které jsme ještě neprobrali. Pokusím se seřadit vzory do logických skupin podle jejich typického použití a vykládat pokud možno nejprve ty jednodušší a postupně se probojovávat k těm složitějším. Když totiž pochopíš filozofii těch jednodušších vzorů, budou se ti ty složitější učit mnohem snadněji.

Které vzory budeme probírat

7. Už jsi mi je dostatečně vychválil, takže bychom se na ně mohli vrhnout, co říkáš?

Nebudu se omezovat pouze na vzory z GoF

Než začneme, tak bych ti chtěl ještě prozradit, že se nehodlám omezit na výše zmíněných 23 vzorů z GoF. Přidám ti k nim i některé další, které se do GoF neprobojovaly – zřejmě proto, že autorům GoF připadaly příliš jednoduché. Domnívám se však, že do výkladu o návrhových vzorech a vůbec o duchu moderního programování rozhodně patří.

Protože si myslím, že rozdělení vzorů na skutečné návrhové vzory a pouhé polovzory či idiomy, jež některé učebnice zavádějí, je spíše teoretickou záležitostí, nebudu tuto klasifikaci nijak vypichovat a u vzorů, které nenajdeš v GoF, tuto skutečnost pouze okrajově zmíním.

8. Těch „skutečných“ je opravdu jenom 23?

Návrhových vzorů není mnoho

23 vzorů bylo publikováno v GoF. To jsou všeobecně použitelné vzory. Postupně byly definovány další, avšak ty již byly často specializované – byly to návrhové vzory určené pro některé specifické oblasti programování. Některé z nich je sice možno považovat za relativně obecné, takže bychom o ně mohli rozšířit základní sadu, ale nebývá to zvykem.

9. Tím chceš říct, že se učebnice omezují pouze na výklad oněch 23 vzorů z GoF?

Nejen to. Ty dvě, které se mi líbily ze všech nejvíc (v závěrečném přehledu doporučené literatury jsou uvedeny pod čísly [18] a [15]), vykládají pouze některé z nich.[1]

Nicméně i tak je návrhových vzorů poměrně málo. Taková záplava vzorečků, jakou tě zahrnuje např. fyzika, ti v programování nehrozí.

---

[1] To byl ostatně i můj původní záměr u této knihy. Nakladatel však projevil přání, aby kniha probírala všechny návrhové vzory, takže dojde i na ty exotičtější a méně používané.

Shrnutí – co jsme se naučili

- Návrhové vzory mají v programování podobný význam jako vzorce v matematice: zrychlují řešení a zestručňují a zkvalitňují komunikaci.
- Návrhové vzory jsou často publikovány v katalozích, které mívají jednotnou strukturu.
- Za základní, univerzální vzory je považováno 23 vzorů publikovaných v GoF.
- Postupně jsou publikovány další vzory určené pro specializované oblasti programování.
- Celkový počet návrhových vzorů je relativně malý.

KAPITOLA 2

Zásady objektově orientovaného programování

- **Programovat proti rozhraní**
- **Důsledné skrytí implementace**
- **Zapouzdření a odpoutání částí kódu, které by se mohly měnit**
- **Přednost skládání před dědičností**
- **Soudržnost (cohesion): jedna entita × jeden úkol**
- **Návrh řízený odpovědnostmi**
- **Minimální vzájemná provázanost (coupling)**
- **Vyhýbání se duplicitám v kódu**
- **Nepodřizovat návrh snahám o maximální efektivitu**
- **Shrnutí – co jsme se naučili**

Co se v kapitole naučíme

Tato kapitola je věnována zásadám správného objektově orientovaného programování, kterým se většina učebnic věnuje pouze okrajově, pokud vůbec. Postupně probereme zásady, kterými bychom se při návrhu našich programů měli řídit, abychom maximalizovali efektivitu vývoje, robustnost kódu a spravovatelnost (manageability) výsledných aplikací.

10. Když jsi ty návrhové vzory tak chválil, říkal jsi, že mi při jejich výuce přejdou do krve zásady programování. Mohl bys mi je tu vyjmenovat, abych věděl, čeho si mám všímat?

Proč probereme zásady OOP

To není špatný nápad. Správně by ses je měl dozvědět na kurzech či z učebnic, podle nichž ses učil programovat. Vím však, že většina učebnic neučí programování, ale pouze syntaxi vybraného jazyka, a takovým věcem, jako jsou zásady správného programování, většinou moc prostoru nevěnuje (pokud vůbec nějaký). Takže je tu s tebou pro jistotu proberu.

Ono se nám toto počáteční opakování bude hodit, protože ti pak budu moci u probíraných vzorů ukázat, jak jsou v nich tyto zásady použity. Nebudu zde proto uvádět příliš mnoho příkladů, protože příkladem aplikace těchto zásad bude téměř každý probíraný návrhový vzor.

Probereme jenom podmnožinu

Zásad správného programování je celá řada. Pokusím se tu s tebou probrat alespoň ty, které považuji za opravdu důležité, abych se pak na ně mohl v textu v případě potřeby odkazovat. Dopředu tě upozorňuji, že takhle pohromadě je asi v žádné učebnici nenajdeš (a to pravděpodobně ani v anglické). Pokud se už autoři učebnice rozhodnou o některé z těchto zásad zmínit, málokdy je nějak výrazně vypichují, aby si je čtenář mohl vštípit.

Probírané zásady

V této kapitole bych chtěl probrat následující zásady správného objektově orientovaného programování:

- Programovat proti rozhraní a ne proti implementaci.
- Neustále dbát na důsledné zapouzdření a skrývání implementace.
- Zapouzdřit a odpoutat části kódu, které by se mohly měnit.
- Upřednostňovat skládání před dědičností.
- Maximalizovat soudržnost (cohesion) entit (balíčků, tříd a metod). Každá entita by měla řešit jen jeden konkrétní úkol.
- Koncentrovat zodpovědnost za řešení úkolu na jednu entitu – hovoříme o návrhu řízeném odpovědnostmi (responsibility driven design).
- Minimalizovat vzájemnou provázanost (coupling) entit.
- Vyhýbat se duplicitám kódu.
- Nepodřizovat návrh snahám o maximální efektivitu.

Programovat proti rozhraní

11. Co si mám např. představit pod programováním proti rozhraní?

Definice rozhraní

Než si začneme povídat o programování proti rozhraní, bylo by dobré si nejprve ujasnit, co budeme rozumět pod pojmem rozhraní. Jedna z možných definic je, že rozhraní entity je množina informací, které o sobě daná entita (atribut, metoda, třída) zveřejní. Rozhraní tedy obsahuje informace, které mohou kooperující programy využívat, resp. které musí respektovat.

Informace deklarované v rozhraní můžeme rozdělit do dvou kategorií: první označujeme jako signaturu, druhou jako kontrakt.

Signatura

12. Jestli si to dobře pamatuji, tak do signatury patří to, co se zapisuje do hlavičky třídy nebo metody.

Definice

Ne tak docela. Signatura (občas se setkáš s překladem *podpis*) představuje souhrn informací zpracovatelných (a tím pádem i kontrolovatelných) překladačem. Nerespektování signatury, tj. nerespektování informací, které do ní zahrnujeme, se většinou projeví jako syntaktická chyba.

– Data

Do signatury dat (tj. konstant a proměnných) patří jejich název a typ. Obě tyto informace se dozvíš v jejich deklaraci.

– Metody

Do signatury metod řadíme nejenom jejich název a typ návratové hodnoty, ale také typy jejich jednotlivých parametrů, vyhazované výjimky, jejich synchronizovanost[1], nativnost[2] a další atributy. Všechny tyto informace vyčteš z hlavičky metody.

– Datové typy

U tříd a obecněji u datových typů (v Javě mezi ně zahrnujeme vedle tříd i výčtové typy a rozhraní) sem patří název typu doplněný o název jeho případného předka a implementovaných rozhraní a signatury všech jeho členů: atributů, metod a zanořených tříd.

Na zjištění signatur metod ti opravdu stačí jejich hlavička a zjištění signatur atributů jejich deklarace. U datového typu však musíš informace z jeho hlavičky doplnit informacemi z hlaviček (u atributů z deklarací) jeho členů.

Signaturu lze zjistit reflexí

U datových typů se sice pojem signatura příliš nepoužívá, ale předpokládám, že si lehce odvodíš, že do něj zahrnujeme vše, co na danou entitu prozradíš ve zdrojovém kódu a co přitom okolní entity „vidí“. Mohli bychom říci, že do signatury datového typu patří to, co dokážeš zjistit od jeho class-objektu s výjimkou informací o soukromých členech, protože tyto členy ti sice class-objekt prozradí, ale pro okolní objekty jsou stejně nedostupné.

13. Takže datová struktura `interface` je taková jedna velká signatura.

`interface` = zápis signatury třídy

Více méně máš pravdu. Java zavedla speciální konstrukci nazvanou `interface`, která formalizuje deklaraci rozhraní a umožňuje deklarovat všechny potřebné informace o signatuře, aniž by bylo nutno se jakkoliv vázat na konkrétní implementaci.

---

1 Synchronizovanost metod se označuje klíčovým slovem `synchronized` a používá se při programování souběžně vykonávaných činností.

2 Jako nativní označujeme metody naprogramované ve strojovém kódu použitého počítače, tj. metody, které virtuální stroj neinterpretuje, ale jenom inicializuje jejich vyvolání. Takovéto metody nemají v javových definicích svůj kód, ale pouze svoji hlavičku (signaturu) označenou klíčovým slovem `native`.

Konstrukce `interface` však neumožňuje kompletní zápis signatury. Nedovoluje totiž deklaraci atributů s výjimkou statických konstant[1]. Uvážíme-li však, že deklarace atributů do správného rozhraní vůbec nepatří, můžeme brát `interface` jako formální zápis signatury třídy.

Jak jsem již říkal v úvodu, termínem *rozhraní* bývá označována jak *množina informací, které o sobě třída zveřejní*, tak i programová konstrukce, která je formálním zápisem signatury. Abych odlišil, kdy hovořím o rozhraní jako o množině informací a kdy o něm hovořím jako o programové konstrukci, rozhodl jsem se, že budu standardně používat přeložený termín rozhraní (v řadě případů se beztak hovoří o rozhraní v obou významech současně), a v případě, když budu chtít zdůraznit, že hovořím o druhu datového typu, uchýlím se k nepřeloženému termínu `interface`, který budu navíc vysazovat „programovým“, tj. neproporcionálním písmem (viz oddíl *Rozhraní* × *interface* na straně 23).

Kontrakt

14. Signaturu jsme probrali. Co mi povíš o kontraktu?

Definice Kontrakt označuje souhrn dalších zásad, které je třeba při použití dané entity (třídy, atributu, metody) dodržet, avšak jejichž dodržování nemůže překladač dost dobře zkontrolovat.

Příklady Sem patří např. omezení kladená na přípustné hodnoty parametrů metod nebo některé dodatečné povinné vlastnosti výsledků metod, jako např. povinnost vzájemné konzistence metod `hashCode()` a `equals(Object)` (platí-li `o1.equals(o2)`, musí být `o1.hashCode() == o2.hashCode()`), nebo to, že metoda `equals(Object)` musí být reflexivní, symetrická, tranzitivní a konstantní.

Častou součástí kontraktu jsou i informace o tom, že při implementaci jedné metody je použita jiná metoda nebo nějaký speciální algoritmus, aby se podle toho mohl případný potomek zařídit.

Příklad takovéhoto kontraktu najdeš například v dokumentaci některých metod třídy `java.util.AbstractCollection`, z níž se dozvíš, že při implementaci metody `addAll(Collection)` je použit cyklus volání metody `add(Object)`. Zajímavý popis důsledků možného přehlédnutí této drobnosti najdeš v [28] v radě 14.

15. Takovéto požadavky ale do kódu zanést nejde.

Kontrakt nemůžeš deklarovat v kódu. Pro jeho deklaraci jsou určeny dokumentační komentáře, kam autor entity své požadavky zapíše. Když pak chceš danou entitu používat, měl by ses nejprve seznámit s jejím kontraktem, abys pak nemusel řešit problémy vzniklé z jeho porušení.

---

[1] Statické konstanty jsou považovány za pouhé názvy objektů. Ve starších verzích Javy zastupovaly neexistující výčtové typy. Po zavedení výčtových typů se však jejich používání omezuje – pro podrobnější rozbor důvodů viz [28] rada 17.

Kontrola dodržení kontraktu

Nicméně to, že kontrakt nelze deklarovat tak, aby jej mohl překladač zkontrolovat, ještě neznamená, že zkontrolovat nejde. Kontroluje se však až při běhu programu vložením vhodných kontrolních příkazů, které ověřují platnost požadovaných vstupních a výstupních podmínek. Java pro tento účel zavedla ve verzi 1.4 klíčové slovo `assert`.

Bertrand Meyer dokonce vyvinul celou metodologii nazvanou *Design by Contract* (DBC), což bychom mohli přeložit jako *návrh dle kontraktu*. Podle této metodiky se na počátku metod (případně i bloků kódu) testuje splnění *vstupních podmínek* (*preconditions*) a na jejich konci splnění *výstupních podmínek* (*postconditions*). Některé moderní programovací jazyky (např. jazyk Eifel a díky příkazu `assert` částečně i Java) mají podporu pro tyto testy zabudovánu přímo ve své syntaxi.

16. To věčné testování na počátku a konci každé metody ale musí strašně zdržovat.

Testování nemusí zdržovat

Nemusí. Provádění testů v příkazech `assert` je možno zapínat a vypínat v příkazovém řádku. To, zda se tyto testy budou či nebudou provádět, je tedy možno zadat při spuštění programu. V době vývoje či hledání chyby proto můžeš provádění testů zapnout a při ostrém běhu odladěného programu, kdy záleží na čase, pak provádění testů vypneš.

17. Obávám se, že takto je ale možno zkontrolovat jenom další podmnožinu kontraktu a že se vždy najdou oblasti, které ani takovéto testy nezkontrolují.

Samozřejmě. Typickým příkladem jsou např. informace o způsobu implementace některých metod, jako byla např. metoda `addAll(Collection)`, o níž jsem se zmiňoval v odpovědi na otázku 14.

Jak zásadu dodržovat

Místo konkrétní třídy používat rozhraní nebo abstraktní třídy

18. No dobře, vysvětlil jsi mi význam termínu rozhraní. Stále mi ale chybí odpověď na otázku, jak uvedenou zásadu uplatňovat.

Už k tomu směřuji. Základní myšlenkou je, že proměnné nemají být deklarovány jako instance konkrétních tříd, ale jako „instance“ nějakého datového typu, jenž není vázán na konkrétní implementaci. Takovým typem je `interface` nebo abstraktní třída[1] (abstraktní třída také nemůže mít vlastní instance).

Když budeš chtít v programu použít např. seznam, nebudeš příslušnou proměnnou deklarovat jako instanci třídy `ArrayList`, ale jako instanci rozhraní `List`, a to i v případě, kdy víš, že bude odkazovat na nějaký `ArrayList`. Tím si uvolníš ruce pro případné budoucí úpravy, při nichž zjistíš, že by bylo výhodnější pro daný účel použít místo instancí třídy `ArrayList` instance nějaké jiné třídy.

Reprezentant skupiny typů

Obdobně budeš-li v programu zřizovat nějaký typ, jenž má reprezentovat objekty, jejichž chování se může v budoucnu různit, definuj jej jako rozhraní, které budou jednotlivé třídy konkrétních objektů implementovat.

---

[1] V souvislosti s programováním proti rozhraní sice GoF hovoří pouze o abstraktní třídě, ale musíme vzít v úvahu, že Java, která datovou konstrukci **interface** zavedla, se v době vzniku GoF teprve rodila.

19. A co když budu potřebovat pro tuto skupinu typů definovat společného rodiče, který by v sobě zahrnoval již nějakou implementaci? V takovém případě by snad byla vhodnější nějaká rodičovská třída, ne?

Zavádět rozhraní i při potřebě definice implementace

NE! I v takové situaci bys měl v Javě definovat nejprve rozhraní deklarující společnou signaturu těchto typů. Vezmi si příklad z knihovny kontejnerů ve standardní knihovně. I zde je sice např. pro všechny seznamy definována společná rodičovská třída `AbstractList`, ale nad ní je ještě rozhraní `List`, a používáš-li v programu seznam, měl bys jej deklarovat jako instanci rozhraní `List`, do níž pak přiřadíš odkaz na instanci některé třídy, která toto rozhraní implementuje (např. `ArrayList`).

Návrh vlastního rozhraní

20. Tím jsi mne přivedl na myšlenku, jestli existují nějaké zásady, které bych měl dodržovat při návrhu vlastních rozhraní.

Pár by se jich našlo. První z nich je, že bys při návrhu rozhraní měl myslet především na jeho budoucí uživatele a navrhovat je tak, aby se jim s ním co nejlépe pracovalo.

21. Tak bych měl ale navrhovat celý program. Proč mi to připomínáš zrovna u rozhraní?

Co je to API

Hovoříš-li o celém programu, pak jde o uživatelské rozhraní. Já teď mám na mysli rozhraní tvého programu, pro něž se často používá zkratka API z anglického *Application Programming Interface* – aplikační programátorské rozhraní. Tím se míní rozhraní, jež budou používat programátoři, kteří budou tvůj program používat, tj. způsob použití, které bude tvůj program (aplikace, služba, modul, komponenta, třída, …) vyžadovat.

22. Dobrá – a co pro něj platí vedle požadavku na přehlednost?

Zásady návrhu správného rozhraní

Bylo by vhodné, kdybys vnější rozhraní svého programu, tj. rozhraní, s nímž se budou setkávat jeho uživatelé, postavil na konstrukcích `interface`. Vezmi si příklad třeba z knihovny kontejnerů.

Rozhraní specifikované pomocí `interface` ti umožní mnohem lépe skrýt implementaci svého programu (budeme o tom hovořit za chvíli) a uvolní ti ruce při jeho dalším vylepšování a upravování. V průběhu dalšího povídání probereme řadu návrhových vzorů, které ti s tím pomohou.

Důsledné skrytí implementace

Terminologie související se skrýváním implementace není jednotná. Někteří autoři je berou jako automatickou součást zapouzdření (*encapsulation*). Jiní omezují termín zapouzdření na samotný fakt slučování věcí, které k sobě patří, pod jednu střechu (objekt obsahuje jak data, tak metody, které s těmito daty pracují). Skrývání implementačních detailů před „nepovolanými zraky“, které je součástí dobře provedeného zapouzdření, pak označují termínem skrývání implementace (*implementation hiding*).

Většina autorů, jejichž výukové texty se mi dostaly do ruky, však výše uvedené termíny nijak důsledně nerozlišuje. Na počátku svých výukových textů sice někteří z nich vysvětlí rozdíl mezi oběma termíny, avšak vzápětí vám prozradí, že se to většinou příliš nerozlišuje, a v dalším textu už většinou hovoří o zapouzdření ve smyslu ukrývání implementace. Protože součástí dobrého zapouzdření je i důsledné skrývání implementačních detailů, přidám se k této skupině „nerozlišovačů" i já. Budu-li proto v dalším textu hovořit o zapouzdření, budu tím jedním dechem hovořit i o maximálním možném skrytí implementačních detailů.

23. Co si představuješ pod důsledným skrytím implementace? Má to být něco víc než zásada, že všechny atributy mají být deklarovány jako `private`?

Rozhraní × Implementace

Má. Programy jsou na tom obdobně jako dávný bůh Janus – i ony mají dvě tváře: rozhraní a implementaci. Co je to rozhraní, to jsme si definovali před chvílí. Zbývá definovat, co je to ta implementace. Lidově bychom ji mohli charakterizovat jako souhrn informací odhalujících, jak to program dělá, že umí to, co umí.

Nedotknutelnost rozhraní

Rozhraní je souhrn informací, které o sobě třída zveřejní, naopak implementace je souhrn informací, které se správná třída snaží maximálně utajit. Nepříjemnou vlastností rozhraní je, že jakmile je jednou zveřejníš, je nedotknutelné. Kdybys je totiž změnil, musel bys zkontrolovat (a ve značné části případů upravit – záleží na tom, zda jsme v rozhraní změnili jenom kontrakt, nebo zda jsme změnili také signaturu) všechny programy, které toto rozhraní používají. To je v řadě případů neřešitelný úkol.

Jakmile přestane být nějaká implementační informace důkladně skrytá a stane se součástí rozhraní, bude jeho součástí na věčné časy.

Interní × publikované rozhraní

24. Tak strašné to snad nebude – jsou-li všechny třídy, jež dané rozhraní používají, součástí mojí aplikace, mohu je při změně rozhraní proběhnout a případně opravit.

Definice publikovaného rozhraní

Máš pravdu. V této souvislosti se mi líbí termín publikované rozhraní, který v [26] zavedl Martin Fowler. Publikací rozhraní rozumí jeho zveřejnění pro programátory vytvářející kód, jenž je pro tvůrce rozhraní nedostupný, a to ani nepřímo (např. tak, že tvůrci kódu změnu rozhraní oznámíte).

Definice interního rozhraní

Jako protiváhu k publikovanému rozhraní si můžeme zavést termín interní rozhraní, kterým budeme rozumět rozhraní dostupné pouze třídám, jejichž kód je autorovi rozhraní přímo či nepřímo dostupný, takže po případné změně rozhraní je schopen zařídit zkontrolování a případnou úpravu všech tříd, které na něm závisí.

25. Jinými slovy: nedotknutelnost se týká pouze publikovaných rozhraní.

Neodbývat definici interního rozhraní

Ano a ne. To, že jsi schopen na změnu interního rozhraní korektně reagovat, ještě neznamená, že jeho definice můžeš „střelit od boku" a nemusíš nad ní přemýšlet. Zejména rozhraní komponent, které budeš (byť interně) používat ve více aplikacích, a rozhraní, která bude používat řada tříd, musíš definovat stejně pečlivě jako rozhraní publikovaná. Jejich definice si proto musíš před zveřejněním důkladně roz-

myslet, aby se pak jejich návrh nestal noční můrou nejenom tobě, ale i řadě dalších programátorů.

Odstrašující příklad

Odstrašujícím příkladem špatně definovaného publikovaného rozhraní budiž metody `suspend()`, `resume()`, `stop()` a `destroy()` ve třídě `Thread`. Prakticky vzápětí po uvedení Javy se vědělo, že tyto metody jsou koncepčně zcela pomýlené a nenaprogramovatelné (a byly hned také prohlášeny za *zavržené – deprecated*).

Nicméně rozhraní, jež je obsahovalo, bylo již publikováno, takže jsou jeho součástí dodnes. Každá učebnice Javy popisující práci s vlákny věnuje nejméně jednu podkapitolku vysvětlení toho, co je na těchto metodách koncepčně pomýlené a proč není vhodné je používat (řekl bych, že je to téměř sebevražedné).

Podzásady

26. Tak už přestaň strašit a prozraď, co dalšího je třeba dodržet.

Nezveřejňovat pomocné metody

Jak jsem již řekl, spolupracující programy by neměly mít šanci zjistit cokoliv o tom, jak tvůj objekt dělá, že umí to, co umí, a neměly by ani mít šanci to jakkoliv ovlivnit. Je proto vhodné zveřejnit opravdu jen ty metody, které chceš dát ostatním programům k dispozici. Ostatní metody (sem patří především ty pomocné) by měly být soukromé.

Pokud některý „šťoura“ namítne, že návrhový vzor *Strategie*, probíraný v kapitole *Vyberte si, jak to chcete (Strategie – Strategy)* na straně 415, se právě takovýmto ovlivňováním zabývá, tak musím opáčit, že každé pravidlo má své výjimky. Před chvílí jsem na příkladu knihovny kontejnerů ukazoval, že občas se popis implementace stává součástí kontraktu. V takovém případě pak mohou okolní třídy tuto informaci využít. Třída, která ale část své implementace zveřejní v kontraktu, možnost takovéhoto využití přímo předpokládá (proto její autor vložil onu informaci do kontraktu) a je na ně „duševně připravena“.

Vytvořit soukromé dvojníky metod použitých v konstruktoru

Soukromé verze bys měl vytvořit i pro ty metody, které by sice potomci mohli překrýt, ale ty potřebuješ zaručit použití původní nepřekryté verze metody. To platí např. pro všechny metody použité přímo nebo nepřímo v konstruktorech. Potřebuješ-li použít v konstruktoru metodu, kterou mohou potomci překrýt, musíš místo ní použít jejího soukromého dvojníka (podrobněji je tento problém vysvětlen např. v [32]):

```
private void soukromýDvojník() {
    //Definice požadované činnosti
}
public void překrytelnáMetoda()  {
    soukromýDvojník();
}
```

27. Opatrně se zeptám: proč to nesmí okolní programy umět zjistit nebo ovlivnit?

Narušení kódu:

Narušení kódu může mít dvě příčiny:

– úmyslné

- buď se někdo opravdu snaží tvůj kód zvenku narušit (a to není tak řídká situace, jak by ses možná domníval)

– neúmyslné

- nebo uděláš ty nebo některý z tvých kolegů v některé jiné části programu chybu.

Nezávisle na příčině narušení bývá výsledek stejný: program přestane chodit tak, jak má. Čím lépe bude tvůj program skrývat implementační detaily, tím snadněji se bude takováto chyba či pokus o jeho narušení odhalovat.

Důsledné zapouzdřování kódu a skrývání implementačních detailů má však i další důvod.

Zapouzdření a odpoutání částí kódu, které by se mohly měnit

28. Další důvod? Jaký?

Zapouzdřený kód je měnitelný

Dobře zapouzdřený kód, jehož implementační detaily jsou skryty, ti umožní jej snáze odpoutat od zbytku kódu a dosáhnout tak toho, že jej můžeš libovolně měnit, aniž bys ovlivnil funkčnost programů, které s ním spolupracují.

29. Co si představuješ pod slovem „odpoutat“?

Odpoutání kódu

Odpoutáním kódu mám na mysli odstranění přímých vazeb. Dosáhneš toho např. tak, když se v programu nebudeš obracet přímo na instance dané třídy, ale budeš se obracet na rozhraní, které daná třída implementuje.

30. K čemu mi to bude dobré?

Kdy to využijeme

Toho využiješ ve chvíli, kdy zjistíš, že bys mohl danou úlohu řešit mnohem efektivněji, ale také ve chvíli, kdy zjistíš, že potřebuješ danou úlohu řešit různými způsoby, mezi nimiž budeš volit v závislosti na situaci.

To je ale poměrně častý případ. Zákazník si totiž často objedná program, který má řešit nějaký jednoduchý problém, a po prvních zkušenostech s programem si dodatečně uvědomí, co vše by mohl po programu ještě chtít, a svoji původní objednávku výrazně rozšíří.

Budou-li tvoje programy navrženy tak, aby nutnost změny některých částí výrazně neovlivnila chod zbytku programu, budeš mít před ostatními výraznou konkurenční výhodu.

31. Opět přestávám chápat, kam mne tlačíš.

Příklad

Zkusím ti to vysvětlit konkrétněji. Dejme tomu, že v tvém programu odkazuje atribut `a` na objekt, jenž má na starosti poskytování služby `Služba`. Ty přitom cítíš, že by danou službu bylo možno poskytovat různými způsoby, a tušíš, že zákazník bude možná chtít mít v příští verzi k dispozici volbu mezi tím, který z nich použije.

Vzpomeneš si proto na doposud probrané zásady a nedefinuješ třídu `Služba`, jejíž instance budou příslušnou službu poskytovat, ale definuješ např. rozhraní `ISlužba`, v němž deklaruješ požadavky na všechny případné poskytovatele dané služby. Tím implementaci dané služby ještě více skryješ, protože ti, kteří budou tuto službu využívat, nebudou znát dokonce ani vlastní třídy instancí, které jim danou službu poskytují. Ty pak můžeš jejich třídy dosazovat podle okamžité potřeby.

Příklad: volitelný výstup

Bude-li onou poskytovanou službou např. nějaký výstup, můžeš jej jednou posílat do standardního výstupního proudu, podruhé jej zobrazovat v nějakém grafickém rozhraní, potřetí jej ukládat do souboru a v dalším případě odesílat někam po síti. To, kam daný výstup ve skutečnosti půjde, však nijak neovlivní činnost těch, kteří budou svá data na tento výstup posílat.

Většina návrhových vzorů ukazuje řešení problému, jak správně zapouzdřit ty části kódu, které jsou klíčové pro zakomponování očekávatelných změn.

Přednost skládání před dědičností

32. Jestli jsem to dobře pochopil, tak upřednostňování skládání před dědičností, které je další zásadou, má také za cíl lepší zapouzdření.

Dědičnost narušuje zapouzdření

Pochopil jsi to správně. Jednou z nepříjemných vlastností dědičnosti je právě to, že nedovoluje zcela skrýt implementační detaily a narušuje tak správné zapouzdření (s podrobným rozborem důvodů tě opět odkážu na [32]). Kromě toho velmi svazuje ruce v tom, co si mohou potomci dovolit.

Dědičnost tříd je krásná a silná vlastnost, ale není všespasitelná. Použiješ-li místo dědičnosti atribut, který bude podle potřeby odkazovat na instance různých tříd, dosáhneš v řadě případů větší flexibility než použitím dědičnosti. Navíc budeš moci jeho hodnotu daleko snáze měnit.

33. Opět trochu tápu – mohl bys uvést nějaký příklad?

Příklad výhodné aplikace skládání

Příklad jsem uváděl před chvílí – je jím aplikace, u níž není dopředu jasné, pro jaký typ výstupu se uživatel rozhodne.

Pokud bys definoval nějakou třídu `UniverzálníVýstup` a pro každý potřebný výstup definoval speciálního potomka, měl bys ruce svázané tím, že by tito potomci nemohli být v jiné dědické hierarchii – např. v hierarchii proudů `OutputStrem`, resp. `Writer`, a pokud bys je chtěl do nějaké takovéto hierarchie zapojit, musel bys k tomu nejspíš použít nějaké zanořené třídy.

Na druhou stranu definuješ-li požadovaný výstupní objekt jako instanci nějakého rozhraní, máš mnohem volnější ruce v tom, jak jeho třídu definovat.

34. Stále ale nechápu, proč by měla dědičnost narušovat zapouzdření.

Proč dědičnost narušuje zapouzdření

Protože ke správné implementaci potomka potřebuješ velice často jisté informace o způsobu implementace předka (v odpovědi na otázku 14 jsem ti uváděl příklad třídy, která musela prozradit v kontraktu část své implementace).

Existují dokonce i situace, kdy při tvorbě předka musíš dokonce předjímat způsob implementace jeho budoucích potomků. Na podrobný rozbor tu není místo – bude-li tě zajímat, zkus si prolistovat kapitolu o dědičnosti v [32].

35. Máš zřejmě pravdu, ale uznej, že v některých případech je použití dědičnosti výhodnější.

Kdy použít dědičnost

Uznávám. Ale doporučoval bych ti používat dědičnost pouze tehdy, pokud se jeví jako výrazně nejjednodušší a nejlogičtější řešení problému. S jistou nadsázkou bych mohl říci: Použij dědičnost, až když všechno ostatní selže.

Nebezpečí špatného užití dědičnosti

Koncepce dědičnosti přináší ještě nebezpečí jejího špatného použití. Řada „objektových začátečníků" je koncepcí dědičnosti natolik opojena, že dědičnost používá i v případech, kam rozhodně nepatří. Neskoč na špek těm, kteří definují třídu jako něčího potomka jenom proto, aby mohli zdědit část implementace budoucího předka. Vytvářejí pak takové nesmysly jako obdélník, který je potomkem bodu, nebo letadlo, jež je potomkem křídla.

Dceřiné třídy definují pouze speciální druhy svých rodičů

Dědičnost používej opravdu pouze tehdy, jsou-li instance potomka speciálním druhem instancí předka. Jinými slovy: dceřiné třídy mohou blíže definovat pouze speciální druhy instancí svých rodičovských tříd. Jedině tak zůstane tvoje aplikace i po několika rozšířeních konzistentní.

Substituční princip Liskové

V této souvislosti se hovoří o substitučním principu Liskové (Liskov Substitution Principle), který říká, že *instance podtypu se musí dát použít všude, kde lze použít instanci nadtypu*; jinými slovy: kdekoliv můžeš použít instanci předka, musí být možno použít i instanci potomka.

Ti, kteří tuto zásadu nedodržují a používají dědičnost převážně k dědění implementace, většinou při dalším rozšiřování své aplikace narazí na nepřekonatelné problémy. Aby nezůstala vina na nich, svedou neúspěch na objektové programování a rozšíří množinu odpadlíků, kteří tvrdí, že OOP již vyzkoušeli a že je pro rozsáhlejší aplikace nepoužitelné.

36. Řekl bych, že by to zase chtělo pár příkladů.

Špatné příklady:

Máš je mít. Nejprve ty odstrašující.

– Letadlo

Autor [24] např. uvádí, jak za ním přišel student, který navrhoval třídu **Letadlo** a ptal se jej, jak to má udělat, když definoval třídu **Letadlo** jako potomka třídy **Křídlo**, ale nedokáže zařídit, aby bylo potomkem dvou křídel.

– Cyklista

Druhý příklad je z jednoho školení objektové verze jazyka *Logo*, kde lektor předváděl užitečnost dědičnosti na příkladu cyklisty, který potřebuje přeskákat přes řeku po kamenech. Definoval proto cyklistu jako potomka žáby, která tuto dovednost ovládala. Neuvědomil si přitom, že nezískal cyklistu, který umí skákat přes kameny, ale žábu, která umí jezdit na kole. Takováto drobná odchylka ale může významně ovlivnit funkčnost celé aplikace.

– Obdélník

Třetí příklad je z dnes již zaniklého časopisu *Softwarové noviny*, v němž autor seriálu o programování v jazyku C# definoval v jednom příkladu obdélník jako potomka bodu. Zapomněl, že potomek musí být kdykoliv považovatelný za předka, a že tedy jeho obdélníky alias body by mohly označovat např. konce úsečky, střed nějaké kružnice, vrcholy trojúhelníku atd.

– Zlomek

Rozbor dalšího špatného použití dědičnosti tě čeká v podkapitole *Příklad špatně definovaného potomka* na straně 81. Zkus tam zatím neodbíhat a počkat, až k němu dorazíme. Možná si pak lépe uvědomíš, jak záludné může takové špatné použití dědičnosti být (jeho záludnost bychom mohli výstižně označit starým normalizačním termínem *plíživá kontrarevoluce*).

Správné příklady:

37. Našel bys ve svých sbírkách také příklady toho, kdy byla dědičnost použita správně?

– Tvary

Samozřejmě. Např. hned v naší knihovně se správcem plátna jsou všechny geometrické útvary potomky abstraktní třídy `APosuvný`, která definuje společné vlastnosti objektů, u nichž můžeš zjišťovat a nastavovat jejich pozici. Trojúhelníky, obdélníky, elipsy, čáry, texty i obrázky jsou pak speciálními případy obecného posuvného objektu. Kdyby nebyly potomky společného rodiče, musel bys ve všech opakovat stejný kód, čímž by ses ale protivil jiné ze zásad správného programování.

– Standardní knihovna

Dalších příkladů je plná standardní knihovna. Projdi si dokumentaci a uvidíš, že kdykoliv je někde použita dědičnost, je potomek vždy speciálním případem předka a má kdykoliv smysl považovat jeho instanci za instanci předka.

Sporné příklady:

Abys viděl, že svět není jenom černobílý, uvedu ti ještě příklady, které mohou být v některých aplikacích správné a v jiných špatné. Typickým případem takovéto nejednoznačně správné dědičnosti je např. čtverec definovaný jako potomek obdélníka či kruh definovaný jako potomek elipsy. Oba jsou sice speciálními případy svých rodičů, ale pro některé účely jsou speciální až příliš, protože nemůžeš libovolně ovlivňovat jejich rozměr.

Nebudu se tu o tom dále rozpovídávat. Kdyby sis o dané problematice chtěl počíst podrobněji, můžeš se podívat např. do [32].

Soudržnost (cohesion): jedna entita → jeden úkol

38. Mezi zásadami byla také jedna, která hovořila o jakési soudržnosti.

Definice soudržnosti

Tato zásada hovoří o tom, že žádný balíček, třída ani žádná její metoda nemají mít na starosti několik věcí najednou. Měly by se soustředit na jeden úkol a být za jeho plnění odpovědny. Čím lépe je tato zásada dodržena, tím je daná entita považována za soudržnější (v anglické literatuře se pro tuto vlastnost používá termín *cohesion*).

39. Jenže to většinou není možné. Vezmi si např. takový obyčejný automat na jízdenky. Musí ti umět sdělit, jaké jízdenky si u něj můžeš koupit, převzít od tebe peníze, přepočítat je, vytisknout jízdenku a vrátit ti nazpět a řadu dalších věcí. A co teprve složitější automaty.

Příklad jízdenkového automatu

Právě jsi nám předvedl oblíbenou začátečnickou chybu: příliš brzy se pouštíš do rozebírání detailů. Jízdenkový automat má jediný úkol: prodat ti jízdenku. Všechny ty úkony, které jsi jmenoval, jsou úlohy jeho jednotlivých podsystémů.

Budeme-li uvažovat jízdenkový automat jako objekt, který nám má prodat jízdenku, budeme se zamýšlet nad tím, co musí umět. Budeme tedy definovat metody, které budou realizovat jednotlivé činnosti. Bude-li činnost jednoduchá, zvládne ji metoda sama, bude-li složitější, může o její realizaci požádat nějaký jiný objekt, nebo dokonce takový objekt vytvořit.

Každá metoda a každý vytvořený objekt však musí mít jediný (a navíc jednoduchý) úkol, za jehož splnění budou odpovědni.

40. Tak teď jsi mne dostal. Takhle jsem o tom nepřemýšlel. Můžeš mi ještě vysvětlit, proč se mám o soudržnost snažit?

Důvody lpění na soudržnosti

Důvodů je celá řada. Mezi nejdůležitější patří náchylnost k chybám a připravenost programu na budoucí změny.

Nejsou-li entity v tvém programu dostatečně soudržné, tj. nesoustředí-li se na jeden úkol, ale rozptylují se řadou různých povinností, často se stane, že opravou jedné funkce ovlivníš i další činnosti, na nichž se daná entita podílí. Nesoudržné programy proto komplikují následný vývoj i testování, protože po každé změně musíš otestovat všechny funkce, které má daná entita na starosti.

Příklad

Zkusím ti to ukázat na příkladu, s nímž se setkal jeden kolega. Programátor udělal program pro kreslení grafických tvarů, v němž (mimo jiné) definoval třídu `Obdélník`, `Elipsa` atd. Pak měl udělat program pro počítání některých vlastností geometrických útvarů (plocha, obvod, …). Pro zjednodušení využil třídy z první aplikace (obdélník je přeci stále obdélník) a doplnil je o metody na zjištění oněch vlastností.

Tím ale narušil soudržnost obou aplikací, protože metody pro výpočet parametrů geometrických útvarů se staly závislé na změnách metod pro jejich kreslení a naopak.

41. Má-li se každá třída a každá metoda soustředit na jedinou věc, pak se kód bude hemžit kraťoučkými metodami.

Většina správně navržených tříd má metody velice krátké

Správně! Podíváš-li se do standardní knihovny, zjistíš, že drtivá většina metod má jeden až pět příkazů. Kent Beck v [25] píše, že metoda, která má více než 10 příkazů, zavání tím, že dělá několik věcí najednou, a bylo by ji proto vhodné rozdělit na několik metod jednodušších. Cay Horstmann dokonce v [38] tvrdí, že přinese-li mu student ke zkoušce program s metodou delší než 35 příkazů, vyhodí jej, aniž by se jej na cokoliv dalšího zeptal.

Citacemi těchto známých autorů chci jenom zdůraznit, že delší metody jsou obecně považovány za špatně navržené. Jednou za čas potřebuješ napsat delší metodu, ale u standardních tříd bude značná část metod nesmírně krátká.

Krátké mají být i definice tříd a počty tříd v balíčku

Obdobně je to s velkými třídami a objemnými balíčky. Metody s dlouhým kódem, třídy s mnoha metodami a balíčky s mnoha třídami jsou s velkou pravděpodobností špatně navržené. Nemusí tomu tak být, jednou za čas je to potřeba, ale většinou tomu tak bývá.

Návrh řízený odpovědnostmi

42. Platí také obrácené pravidlo, tj. že jeden úkol má řešit jedna třída nebo metoda?

Každý úkol by se měl svěřit pouze jedné entitě

Platí. Tento způsob návrhu bývá často označován jako návrh řízený odpovědnostmi (responsibility-driven design). Při něm navrhneš oblasti zodpovědnosti a pak navrhuješ entity (balíčky, třídy, metody), které si danou oblast vezmou na starost.

43. Zkus tu zásadu zase podepřít nějakými argumenty.

Opět obrátím tvoji pozornost na úpravy programů a jejich testování a ladění. Potřebuješ-li v další verzi správně navrženého programu vylepšit nějakou funkčnost, podíváš se, která entita je za ni zodpovědná, a obrátíš na ni svoji pozornost.

Kdybys na zásadu zaměřené odpovědnosti příliš nedbal a zodpovědnosti za jednotlivé funkce byly roztroušené po celé aplikaci, musel bys pokaždé bádat nad tím, kde všude bude třeba program upravit.

Minimální vzájemná provázanost (coupling)

44. Co si mám představovat pod zásadou minimální provázanosti?

Důvody minimalizace vazeb mezi entitami

Zásada minimální provázanosti říká, že každá entita si má při plnění úkolu vystačit pokud možno sama a minimálně se obracet na jiné entity. Tuto zásada samozřejmě není možno absolutně dodržet, ale je vhodné se jí maximálně přiblížit. Jinými slovy: je třeba minimalizovat počet vazeb mezi entitami.

45. Jakých vazeb?

Příklady vazeb

Např. když metody jedné třídy používají objekty a metody jiné třídy, je první třída na té druhé závislá. Začneš-li upravovat funkcionalitu třídy, na níž někdo závisí, měl by prověřit, že tato změna neovlivnila funkčnost oněch závislých tříd.

Minimalizací vzájemných závislostí minimalizuješ počet entit, které může změna v dané entitě ovlivnit, a jejichž funkčnost je proto nutno po změně dané entity prověřit.

46. To snad ale ani nejde – napsat program tak, aby byly jeho třídy a metody na sobě nezávislé?

Neříkal jsem, že mají být jednotlivé třídy či metody na sobě nezávislé, ale že máš jejich vzájemné vazby minimalizovat. Jinými slovy, že máš program navrhnout tak, aby počet vzájemných vazeb mezi třídami byl minimální.

47. Existují nějaká doporučení, jak počet těchto vzájemných vazeb snižovat?

Samozřejmě. Např. celá kniha [21] je věnována tomu, jak program, který není navržen zcela optimálně, upravit tak, aby dělal totéž, ale byl by navržen lépe, a bylo jej proto např. snazší modifikovat.

Řadu dalších způsobů se dozvíš i v průběhu následujícího výkladu. Mnohé návrhové vzory totiž řeší právě tento problém.

Vyhýbání se duplicitám v kódu

48. Předposlední zásadou jsou jakési duplicity v kódu.

Programování metodou copy-paste

Často se stává, že se dvě či více věcí řeší velice podobně, nebo dokonce stejně. Řada programátorů proto zkopíruje na všechna místa stejný kód a v případě potřeby jej pouze drobně doupraví.

Důvody nevhodnosti

Jak jsi jistě pochopil, takovýto postup správný není. Jeho hlavní nevýhodou je, že takový kód je pak velice náchylný k chybám. Přiznejme si, že jsme lidé omylní. Uděláme-li v kopírovaném kódu chybu, musíme při její opravě obejít všechna místa, kam jsme danou část kódu zkopírovali, a na všech místech ji správně opravit. Tím většinou zaneseme do kódu několik dalších chyb.

49. Z podobného důvodu se nemají v kódu používat literály, ale máme dávat přednost pojmenovaným konstantám.

Literály × pojmenované konstanty

Máš pravdu. Důvody jsou prakticky stejné. U kódu je to však přece jenom o maličko složitější, protože kód pracuje s daty, která mu musíš nějakým způsobem poskytnout. Většinou je nejvýhodnější je předat jako parametry, ale v řadě případů to není nejlepší řešení. To bych tu ale teď nechtěl rozebírat. Řekl bych, že tato problematika je již velice podrobně rozebrána v [26].

Nepodřizovat návrh snahám o maximální efektivitu

50. V poslední zásadě tvrdíš, že se nemám pokoušet o maximální efektivitu programu. To mi nepřipadá příliš moudré.

Neříkal jsem, že se nemáš pokoušet vytvořit svůj program efektivní, ale že nemáš podřizovat návrh snahám o maximální optimalizaci. Je totiž známou skutečností, že řada zkrachovaných projektů zkrachovala mimo jiné právě díky předčasným snahám o optimalizaci.

51. Jak poznám, že má snaha o optimalizaci je předčasná.

Kdy je snaha o optimalizaci předčasná

Když se pokoušíš o optimalizaci nějaké části programu, a přitom jsi program ještě nerozběhl. Nemá smysl se pokoušet optimalizovat něco, o čem nevíš s jistotou, že tvůj program zdržuje.

Program tráví 80 % života ve 20 % kódu

Je známou skutečností, že průměrný program tráví přibližně 80 % svého života ve 20 % svého kódu. Ne vždy je dopředu patrné, která část kódu bude tvořit oněch 20 %. Zejména když po prvním předvedení programu zákazník své požadavky výrazně modifikuje. Již mnohokrát se proto stalo, že programátoři strávili dlouhé dny nad optimalizací části kódu, která byla nakonec z celého projektu vypuštěna.

Program má být především modifikovatelný

Program má být navržen především tak, aby šel dobře modifikovat, protože jestli se můžeš v programování na něco spolehnout, tak na to, že za chvíli bude všechno jinak.

Optimalizace vyžaduje soustředění na detail

Při optimalizaci kódu se soustředíš především na detail, a přitom zákonitě ztrácíš ze zřetele ty rysy programu, které vyžadují globální pohled. Snadno pak vytvoříš kód, který bude kolidovat s jakýmisi globálními předpoklady, na nichž je postaven kód ve zcela jiné části programu. Na detaily by ses měl proto soustředit až ve chvíli, kdy je celek navržen a odladěn.

52. Z toho, cos' říkal, mi vyplývá, že mám program optimalizovat až po tom, co jej odevzdám. Není to příliš pozdě?

Optimalizovat až tehdy, kdy vím, co doopravdy zdržuje

To jsi mne špatně pochopil. Já jsem neříkal po tom, co jej odevzdáš, ale poté, co jej rozběhneš. Teprve pak můžeš zodpovědně prohlásit, která část programu vykonává svoji činnost neúnosně pomalu a zbytek programu tak zbytečně zdržuje.

Předčasná snaha o optimalizaci je cestou do hrobu

Pamatuj si: předčasná snaha o optimalizaci je cestou do hrobu (její vábení považuji za natolik svůdné a nebezpečné, že jsem tuto zásadu musel vysadit tučně). Pokud totiž vábení optimalizace podlehneš, začneš vypouštět ze zřetele daleko důležitější věci. Věci, které se na výsledné hodnotě programu projeví daleko citelněji než nějaká nedotažená optimalizace.

53. Jenomže když se na optimalizaci vykašlu, tak mohu vytvořit program, který bude tak neefektivní, že bude nepoužitelný.

Je řada věcí, které mají na výslednou efektivitu vliv. Zásadní vliv má správně navržená architektura celého řešení a správně zvolené algoritmy časově kritických částí kódu.

Strategická × taktická rozhodnutí

Předčasné snahy o optimalizaci kódu, o nichž jsem hovořil, se však netýkají výše uvedených „strategických" rozhodnutí. Týkají se především drobných „taktických" anebo přímo „operativních" rozhodnutí – např.:

- zda dát v daném případě přednost klasickému poli vyžadujícímu pracnější správu anebo sáhnout po nějakém dynamickém kontejneru, který sice bere potřebnou správu na svá bedra, ale nemá ji pro daný účel navrženou optimálně;
- zda zvolit výběr prostřednictvím přepínače s mnoha větvemi či dát přednost ukládání párů [volba;akce] do nějaké mapy;
- zda uložit nějakou hodnotu do pomocné proměnné anebo o ni vždy požádat zavoláním příslušné metody;
- zda dát přednost pomaleji volané, avšak snáze modifikovatelné virtuální metodě anebo rychleji volatelné metodě statické;
- zda vytvořit pokaždé nový objekt a zvýšit tak zatížení správce paměti anebo si objekt někam uložit pro další použití, aby jej nebylo nutno neustále vytvářet a mazat;
- jak upravit příkazy tak, abych mohl všechny opakující se výpočty „vytknout" před cyklus;
- a další a další a další...

Optimalizace překladače bývá efektivnější nad čistým kódem

V současné době se ukazuje, že takovéto problémy je optimalizující překladač velmi často schopen řešit efektivněji než průměrný programátor a že předčasné pokusy o drobnou optimalizaci jsou spíše kontraproduktivní, protože znepřehledňují výsledný kód nejenom pro programátora, ale i pro překladač.

Shrnutí – co jsme se naučili

V kapitole jsme stručně shrnuli nejdůležitější zásady, které bychom měli v našich programech dodržovat. Jsou to:

- Programovat proti rozhraní a ne proti implementaci.
- Neustále dbát na důsledné skrývání implementačních detailů.
- Zapouzdřit a odpoutat části kódu, které by se mohly měnit.
- Upřednostňovat skládání před dědičností.
- Maximalizovat soudržnost (cohesion) entit (balíčků, tříd a metod). Každá entita by měla řešit jen jeden konkrétní úkol.
- Koncentrovat zodpovědnost za řešení úkolu na jednu entitu – hovoříme o návrhu řízeném odpovědnostmi (responsibility driven design).
- Minimalizovat vzájemnou provázanost (coupling) entit.
- Vyhýbat se duplicitám kódu.
- Nepodřizovat návrh snahám o maximální efektivitu.

KAPITOLA 3

Co konstruktor neumí (Jednoduchá tovární metoda – Simple Factory Method)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Jednoduchá (někdo používá termín *statická*) *tovární metoda* je zjednodušenou verzí obecnějšího návrhového vzoru *Tovární metoda* (viz kapitolu *Střihni mi to na míru (Tovární metoda – Factory Method)* na straně 279). Definuje statickou metodu nahrazující konstruktor. Používá se všude tam, kde potřebujeme získat odkaz na objekt, ale přímé použití konstruktoru není z nejrůznějších příčin optimálním řešením.

54. Když říkáš, že jsou návrhové vzory tak užitečné, tak se je rád naučím. Kterým začneme?

Než se pustíme do vzorů z GoF, tak bych tě rád seznámil s několika idiomy, které sice nejsou považovány za návrhové vzory (nenajdeš je v GoF), ale jejich znalost je neméně užitečná. Až se začneme bavit o návrhových vzorech, tak bych je rád použil a nerad bych, abychom si je v tu chvíli teprve začali vysvětlovat.

Účel

55. Neomlouvej se a začni.

Potřeba hlídání počtu instancí

V řadě situací bychom potřebovali, aby třída umožnila svému okolí získávat odkazy na její instance a pracovat s nimi, avšak bez toho, že by tomuto okolí poskytla konstruktor. Jakmile má někdo k dispozici konstruktor třídy, může si vytvářet nové instance podle libosti. Často však potřebujeme vytváření nových instancí nějakým způsobem hlídat nebo při něm provést některé akce, které nám konstruktor nedovolí.

56. Co nám konstruktor nedovolí? Vždyť je to jenom trochu zvláštní metoda.

Omezení kladená na používání konstruktorů

Obávám se, že jsi při probírání konstruktorů nedával moc pozor. Při používání konstruktoru musíme mít na paměti několik omezení:

- Prvním příkazem konstruktoru musí být volání přetíženého konstruktoru anebo rodičovského konstruktoru. (Toto omezení sice neplatí ve všech jazycích, nicméně v Javě, v C++ i v jazycích pro .NET platí.)
- Nechceme-li se v budoucnu dostat do potíží, nesmíme v konstruktoru používat žádné virtuální (tj. překrytelné) metody.
- Konstruktor vždy vytvoří novou instanci, ať už se nám to hodí nebo ne.

57. No dobře, někdy je jeho použití nešikovné. Ale bez volání konstruktoru instanci nevytvořím.

Kdy se použití konstruktoru nehodí

To je sice skoro pravda, ale jak jsem uvedl v posledním bodě, při volání konstruktoru se vždy vytvoří nová instance, a to se nám někdy nehodí.

58. Když nechci novou instanci, nebudu přece volat konstruktor.

Jenže občas potřebuješ k další práci získat nejprve odkaz na instanci, s nímž budeš dále pracovat. Tebe v danou chvíli nezajímá, jestli to bude nová instance nebo nějaká existující. Tuto starost necháš rád na někom jiném. Ty prostě potřebuješ k další práci onu instanci.

59. Začínám tě chápat. Přidej ještě nějaký příklad.

Příklad: jediná instance

Ve svých kurzech paralelního programování předvádím některé konstrukce na příkladech ze světa robotů. Roboti jsou objekty, jejichž grafické reprezentace se pohybují v okně reprezentujícím dvorek. Všichni roboti se pohybují po stejném dvorku (aby si mohli překážet).

Potřebuji-li pracovat s dvorkem (např. chci nastavit jeho velikost), musím na něj získat odkaz. Problém mohu ve třídě `Dvorek` řešit dvěma způsoby:

- definuji veřejnou konstantu obsahující odkaz na dvorek,
- definuji tzv. tovární metodu, která bude vracet odkaz na instanci dvorku a bude používána místo konstruktoru.

První přístup je možná o něco rychlejší, ale na druhou stranu budu v případě, kdy se rozhodnu používat několik dvorků, muset do programu mnohem více zasahovat.

Volání tovární metody je sice teoreticky o něco pomalejší (optimalizační překladače je však umí nahradit), ale na druhou stranu mi nechává daleko volnější ruce pro případné další úpravy a rozšíření.

60. No dobře, to je tvůj program. Našel bys také příklad ze standardní knihovny?

Příklad: třída Integer

Mraky. Např. autoři třídy `Integer` věděli, že průměrný program žádá o vytvoření instancí pro malé hodnoty nepoměrně častěji než pro velké. Třída má proto předdefinované instance pro hodnoty od -128 do 127. Kdykoliv použiješ konstruktor, vytvoříš novou instanci. Použiješ-li však pro získání instance některou z továrních metod `valueOf(???)`, obdržíš pro malé hodnoty odkaz na některou z existujících instancí. Tím se program výrazně zefektivní: instanci obdržíš rychleji, ušetříš paměť a s ní i práci správce paměti.

61. Ještě bych se vrátil k odpovědi na otázku 57, kde jsem říkal, že bez volání konstruktoru novou instanci nevytvořím, a ty jsi odpovídal, že to je skoro pravda. Proč jenom skoro?

Instance nemusí vytvářet jen konstruktor

Protože instanci můžeš vytvořit i klonováním (viz kapitolu *Baťovy cvičky (Prototyp – Prototype)* na straně 285) nebo načtením z nějakého proudu. Takto však můžeš vytvořit pouze instance, které vzniknou jako kopie nějaké dříve existující instance. Úplně novou instanci lze vytvořit jedině pomocí konstruktoru.

Až budeme hovořit o klonování, tak si povíme, že někdy musí konstruktor své instance vytvářet složitě, takže je výhodnější vzít nějakou existující instanci a prostě ji zkopírovat. No a tady se opět může uplatnit tovární metoda, která rozhodne, která z cest je v dané situaci výhodnější.

Implementace

62. Říkáš tedy, že v mnoha případech je lepší použít místo konstruktoru tovární metodu. Mohl bys ji stručně charakterizovat?

Továrních metod je několik druhů; přesněji: termín *tovární metoda* se používá pro více konstrukcí. Prozatím se omezíme na tu nejjednodušší, která bývá označována jako *Jednoduchá tovární metoda*. K těm zbylým variantám se vrátíme později.

63. Dobrá – zkus tedy stručně charakterizovat jednoduchou tovární metodu.

Charakteristické vlastnosti – obyčejná metoda

Jednoduchá tovární metoda je obyčejná metoda, která jako svoji návratovou hodnotu vrací instanci zadané třídy (připomínám, že mezi instance třídy počítáme i instance jejích potomků).

– statická

Jednoduchá tovární metoda, o které hovoříme v této kapitole, bývá definována jako statická metoda třídy, jejíž instanci vrací. Pak může být zavolána ještě před tím, než bude existovat první instance dané třídy – např. právě proto, aby nechala tuto instanci vytvořit a vrátila odkaz na ni.

– nemusí instanci vytvořit

Základní rozdíl mezi konstruktorem a tovární metodou je v tom, že konstruktor musí po svém zavolání vytvořit novou instanci (musí ji zkonstruovat – proto se tak také jmenuje), kdežto tovární metoda se může svobodně rozhodnout, zda vytvoří instanci novou nebo vrátí odkaz na instanci existující.

– může vracet instance různých typů

Obrovskou (a často využívanou) výhodou tovární metody je, že se může rozhodnout, zda vrátí instanci deklarovaného typu nebo některého z jeho potomků. O takovéto možnosti si může konstruktor nechat tak leda zdát.

– rozdíly v syntaxi

Další rozdíly jsou syntaktické. Konstruktor je potřeba volat prostřednictvím operátoru `new`, kdežto tovární metodu voláš jako jakoukoliv jinou metodu.

– větší svoboda v definici

Volá-li přetížená verze konstruktoru jeho jinou verzi, musí být toto volání prvním příkazem těla konstruktoru. Tovární metody na nás žádná takováto omezení nekladou. Budeme-li mít dvě přetížené verze, můžeme v těle druhé metody v klidu chvíli něco připravovat a teprve pak zavolat první metodu (nebo konstruktor).

– nelpí na jméně => mohou být dvě verze se stejnou sadou parametrů

Další z drobných výhod továrních metod je to, že jim můžeš zvolit jméno (i když i zde je vhodné dodržovat jisté konvence). Není proto problém vytvořit dvě různé tovární metody se stejnou sadou parametrů, což u konstruktoru z principu nelze.

64. Zatím to vypadá, že tovární metody mají oproti konstruktorům jen samé výhody.

To tak opravdu pouze vypadá, protože jsem se zatím soustředil pouze na situace, kdy použití konstruktorů přináší problémy. Každopádně používání konstruktorů se vyhnout nemůžeš, protože jinak novou instanci nevytvoříš (nanejvýš můžeš vytvořit kopii nějaké již existující).

Tovární metody slouží především k tomu, abychom mohli vytváření instancí více ovlivnit.

65. Hovořil jsi o konvencích pro názvy továrních metod.

Konvence pro názvy továrních metod

Dodržování konvencí umožní ostatním se ve tvých programech lépe vyznat – např. hned odhadnout, že se pravděpodobně jedná o tovární metodu. V praxi se většinou volí mezi následujícími možnostmi:

- `getInstance`

- `getInstance` – pravděpodobně nejčastější volba (ve standardní knihovně Javy 5.0 je použita 90krát);

- `valueOf`

- `valueOf` – používá se v případech, kdy se převádí nějaká hodnota na instanci dané třídy (ve standardní knihovně je použita 51krát);

- `getXxx`

- `getXxx`, kde `Xxx` je název třídy, na jejíž instanci metoda vrací odkaz (ve standardní knihovně je použit v několika desítkách případů). Příkladem metody s tímto názvem je např. naše `Plátno.getPlátno()`.

  Takto vytvořený název je častou volbou zejména tehdy, když třída `Xxx` není dostupná a o odkaz na její instanci je třeba požádat někoho jiného.

– jiné

- V případě, kdy metoda vrací instanci s nějakými speciálními vlastnostmi, použije se v názvu metody popis těchto vlastností. Např. třída `BigInteger` definuje tovární metodu `probablePrime`, jež vrací odkaz na instanci, která je s velkou pravděpodobností prvočíslem.

66. Proč není tovární metoda v GoF?

Je speciálním případem obecnějšího vzoru z GoF

No ona tam je a není. *Jednoduchá tovární metoda*, o které jsme si nyní povídali, je speciálním případem obecnějšího vzoru *Tovární metoda* – ten probereme později. (Pokud se nemůžeš dočkat, otevři si kapitolu *Střihni mi to na míru (Tovární metoda – Factory Method)* na straně 279.)

Příklad

67. Mohl bys demonstrovat použití jednoduché tovární metody na nějakém příkladu?

Jednoduchou tovární metodu budeme používat v průběhu dalšího výkladu mnohokrát, takže si dovolím nyní uvést pouze jednoduchý AHA-příklad.

68. Teď jsi mne zaskočil – co je to AHA-příklad?

Co je to AHA-příklad

To je jednoduchý příklad, který nemá žádný jiný účel než demonstrovat vysvětlovanou látku. Příklad, po jehož prostudování si řekneš: „Aha, takhle to tedy funguje!“

Já se snažím tyto typy příkladů používat minimálně a ukazovat použití vysvětlovaných konstrukcí na něčem prakticky použitelném. Nicméně občas jsou užitečné.

69. Chápu – tak demonstruj.

Ukázku použití metody, která nevytvoří vždy novou instanci, si ukážeme v příští kapitole. Tady ti teď předvedu, jak bychom mohli řešit situaci, kdy by uživatel žádal instanci nějakého obecného typu (v našem příkladě instanci abstraktní třídy `AČlověk`), který ani nemusí být schopen vytvářet instance – může to být abstraktní třída, nebo dokonce rozhraní.

O instanci požádá *Jednoduchou tovární metodu*, která se sama rozhodne, jakého typu bude objekt, jehož instanci vrátí. V ukázce ve výpisu 3.1. vrací metoda `getČlověk()` postupně instance jednotlivých typů člověka (typ vracené instance by bylo možno vybírat i náhodně), v jiných situacích se o tomto typu rozhodne na základě informací předaných volající metodou prostřednictvím parametrů. Vše záleží na konkrétních potřebách vytvářeného programu.

Výpis 3.1: `Člověk` – demonstrace použití jednoduché tovární metody

```
package rup.česky.vzory._03_jednoduchá_tm;


/*****************************************************************************
 * Třída AČlověk slouží k demonstraci použití jednoduché tovární metody
 * vracející instanci některého z předem definovaných typů.
 */
abstract public class AČlověk
```

```
{
//== PROMĚNNÉ ATRIBUTY TŘÍDY ================================================

    private static int index = 0;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==========================================

    /**************************************************************************
     * Jednoduchá tovární metoda vracející odkaz na člověka
     * postupně měněného typu.
     */
    public static AČlověk getČlověk() {
        switch( index++ % 3 ) {
            case 0:  return new Lenoch();
            case 1:  return new Čilouš();
            case 2:  return new Pracant();
            default: throw new RuntimeException( "Špatně definované maximum" );
        }
    }


//== ABSTRAKTNÍ METODY ======================================================

    abstract public void budíček();
    abstract public void práce();
    abstract public void volno();
    abstract public void spánek();


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    private static class Lenoch extends AČlověk {
        public void budíček() { System.out.println("Pomalu vstávám" ); }
        public void práce()   { System.out.println("Líně pracuji"   ); }
        public void volno()   { System.out.println("Odcházím spát"  ); }
        public void spánek()  { System.out.println("Stále spím"     ); }
    }


    private static class Čilouš extends AČlověk {
        public void budíček() { System.out.println("Rychle vstávám"    ); }
        public void práce()   { System.out.println("Čile pracuji"      ); }
        public void volno()   { System.out.println("Aktivně odpočívám" ); }
        public void spánek()  { System.out.println("Omdlím a spím"     ); }
    }


    private static class Pracant extends AČlověk {
        public void budíček() { System.out.println("Brzy vstávám"     );  }
        public void práce()   { System.out.println("Zaníceně pracuji" ); }
        public void volno()   { System.out.println("Stále pracuji"    ); }
        public void spánek()  { System.out.println("Usínám nad prací" ); }
    }
```

```
//== TESTY ====================================================================

    /*****************************************************************************
     * Několikrát si řekne o nového člověka a nechá jej prožít pracovní den.
     */
    public static void test() {
        for( int i=1;   i <= 3;   i++ ) {
            AČlověk č = getČlověk();
            System.out.println( "\n=== Nový člověk: " +
                č.getClass().getSimpleName());
            č.budíček();
            č.práce();
            č.volno();
            č.spánek();
        }
    }
    /** @param ppr Parametry příkazového řádku - nepoužité */
    public static void main( String[] args )  {  test();  }
}
```

Shrnutí – co jsme se naučili

- Při používání konstruktorů musíme respektovat různá omezení:
  - pokaždé vytvoří instanci,
  - mohou vytvořit pouze instance své třídy,
  - je třeba respektovat další omezení v kódu.
- Jedním z možných řešení je použít k získání instance místo konstruktoru *Jednoduchou tovární metodu*.
- *Jednoduchá tovární metoda* je obyčejná, většinou statická metoda vracející instanci deklarovaného typu.
- *Jednoduchá tovární metoda* nemusí instanci vytvořit – může vrátit odkaz na nějakou existující instanci.
- *Jednoduchá tovární metoda* může vracet i odkazy na instance potomků deklarovaného typu.
- Pro tvorbu názvů továrních metod platí jisté konvence.
- *Jednoduchá tovární metoda* je speciálním případem obecnějšího vzoru *Tovární metoda* z GoF.

KAPITOLA 4

Nehemži se mi pod rukama (Neměnné objekty – Immutable Objects)

- **Účel**
- **Implementace**
- **Příklad**
- **Příklad špatně definovaného potomka**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Neměnný objekt je objekt, u nějž není možno změnit jeho hodnotu.

Účel

70. Copak tam máš na mne připraveno dalšího?

Teď bychom si mohli např. povědět, jak to zařídit, abychom mohli s objekty pracovat obdobně jako s hodnotami primitivních typů, tj. aby ses mohl spolehnout na to, že uložíš-li do proměnné odkaz na objekt, bude mít daný objekt stále stejné vlastnosti.

71. To mi budeš muset vysvětlit podrobněji.

Chování hodnot primitivních typů

Uložíš-li např. do nějaké proměnné dvojku, tak do té doby, než do této proměnné uložíš něco jiného, najdeš v ní vždy dvojku, a to nezávisle na tom, do kolika jiných proměnných jsi její obsah zkopíroval.

Chování běžných objektů

Uložíš-li však do nějaké proměnné např. modrý obdélník o velikosti 100 × 100 bodů umístěný v počátku souřadnic a zkopíruješ-li obsah této proměnné do několika dalších proměnných, může se stát, že až si jej přijdeš za chvíli z původní proměnné vyzvednout, najdeš v ní odkaz na červený obdélník velikosti 20 × 50 umístěný navíc zcela mimo zobrazovanou oblast. Nic totiž nebrání tomu, aby někdo obdélník, uložený v pro něj dostupné proměnné, nepřebarvil či neposunul.

72. To je přece normální. Jakmile někdo obdrží odkaz na objekt, může si s ním dělat, co chce.

Právě. A to se nám někdy nehodí. Dovol mi nejprve drobnou teoretickou odbočku:

V programech používám dva typy datových typů, které označujeme jako **hodnotové** a **referenční** objektové typy.

Hodnotové a referenční datové typy

73. Já jsem si doposud myslel, že termín *hodnotový typ* je synonymum k termínu *primitivní typ*.

Dvě používané interpretace

Máš pravdu. Tady je takový drobný terminologický problém, kdy potřebujeme najít termín pro dvě podobné věci.

Hodnotový datový typ

Termín *hodnotové datové typy*, o němž jsi hovořil, používají někteří autoři jako souhrnný název pro typy, jejichž „instance“ se v parametrech metod předávají hodnotou. V Javě mezi ně patří pouze primitivní typy. (V jiných jazycích to tak být nemusí – např. v C++ můžeš předat hodnotou cokoliv.)

Referenční datový typ

Protože instance objektových typů se předávají v parametrech zásadně odkazem, označuje je řada autorů jako odkazové neboli *referenční datové typy*. Jenomže v Javě nikdy nebudeš mít „v ruce“ nic jiného než odkaz na objekt. A ten se předává hodnotou.

Začátečnické problémy

Ze zkušenosti vím, že toto dělení začátečníky v některých situacích mate, protože v nich vyvolává dojem, že se objekty chovají podobně jako klasické parametry předávané odkazem. To ale není pravda, protože nikdy nemůžeš předat metodě nějaký objekt jako parametr a po skončení metody najít v příslušné proměnné odkaz na jiný objekt.

Používaná terminologie

Proto také tyto termíny nepoužívám a dávám přednost dělení na *primitivní* a *objektové* datové typy – pak vždy víš, na čem jsi a jak se instance daného typu bude chovat. Tyto termíny také budu v celé knize důsledně používat.

74. Před chvílí jsi ale použil termíny hodnotový a referenční typ!

Nepoužil. Podíváš-li se zpět na moji odpověď na otázku 72, zjistíš, že jsem hovořil o hodnotových a referenčních objektových typech. Já vím, že je to jenom nepatrný terminologický rozdíl, ale lepší termín jsem zatím nevymyslel. Budu-li proto někdy v budoucnu hovořit o hodnotových či referenčních typech, budu tím mít na mysli vždy hodnotové či referenční objektové typy.

Instance hodnotových objektových typů mají (no, spíš mohou mít) nějaké vlastnosti primitivních datových typů, kdežto instance referenčních objektových datových typů jsou jim svými vlastnostmi na hony vzdáleny.

Hodnotové objektové typy

75. Dobře – tak začni třeba těmi hodnými ☺, vlastně hodnotovými objektovými typy.

Podstata hodnotových objektových typů

Instance hodnotových objektových typů slouží k uchovávání hodnot. Máme-li dvě instance hodnotových typů, můžeme zjišťovat, obsahují-li stejnou hodnotu obdobně, jako se na to můžeme ptát u dvou proměnných primitivních datových typů. Mohli bychom říci, že hodnotové typy jsou ty, u nichž má smysl definovat jejich vlastní verzi metody `equals(Object)`.

Předchozí tvrzení lze používat i obráceně: má-li nějaký datový typ smysluplně definovánu vlastní metodu `equals(Object)`, jedná se o hodnotový datový typ.

Příklady

Ze známých tříd ze standardní knihovny bychom sem mohli zařadit obalové typy, třídy `java.lang.String`, `java.io.File`, `java.awt.Point` a řadu dalších. U jejich instancí má smysl se pídit po shodnosti hodnot, tj. obsahují-li např. dvě instance třídy `String` stejný řetězec či označují-li dvě instance třídy `Point` stejnou pozici.

Další dělení:

Množina akcí, které můžeme s instancemi hodnotových typů bezpečně provádět, ale závisí na tom, může-li se hodnota těchto instancí v průběhu jejich života měnit. Z tohoto hlediska rozdělujeme hodnotové typy do dvou skupin:

– Neměnné

- Neměnné (anglicky immutable) hodnotové typy nám nenabízejí žádnou možnost, jak změnit hodnotu uchovávanou v jejich instanci. Instance neměnných hodnotových typů se proto chovají naprosto stejně jako hodnoty primitivních datových typů. Jakmile do nich uložíme odkaz na nějakou hodnotu, můžeme se spolehnout na to, že už budou vždycky odkazovat na tuto hodnotu. Z výše jmenovaných tříd sem patří obalové typy a třídy `String` a `File`.

  Konstanty (tj. atributy s modifikátorem `final`) neměnných hodnotových objektových typů proto můžeme v případě potřeby deklarovat jako veřejné. Musíme se ovšem smířit s tím, že se tak stanou součástí rozhraní, a že proto nebudeme moci toto své rozhodnutí v budoucnu změnit. Proto je často lepší nezveřejňovat ani ty.

  Jako veřejné statické konstanty jsou ve standardní knihovně definovány např. základní barvy (instance třídy `java.awt.Color`) nebo nejčastěji používané hodnoty „číselných“ tříd `java.math.BigInteger` a `java.math.BigDecimal`.

– Proměnné

- Proměnné (anglicky mutable) typy nám neměnnost uchovávaných hodnot nezaručí. Hodnota uchovávaná v jejich instanci se naopak může kdykoliv změnit. Z výše jmenovaných tříd sem patří třída `Point`.

76. Má smysl uvažovat o takových hybridních datových typech, jejichž instance mají některé atributy konstantní a jiné proměnné?

No, někdy se to hodí. Pokud to ale jenom trochu jde, tak bys měl datový typ definovat tak, aby atributy, z jejichž hodnoty se odvozuje výsledek (tj. návratová hodnota) metod `equals(Object)` a `hashCode()`, byly neměnné. U atributů, které jsou zmíněnými metodami ignorovány, by se dalo uvažovat o tom, že neovlivňují hodnotu dané instance, a že by to proto nemusely být bezpodmínečně konstanty.

Takto definované typy bychom pak mohli zařadit mezi neměnné, protože se hodnota jejich instancí nedá změnit.

77. Říkal jsi, že hodnotu instance neměnného datového typu nelze změnit. Co když ale někdo za zády programu přejmenuje soubor, na který odkazuje instance třídy `File`?

Instance `File` nemají přímou vazbu na soubor

Podlehl jsi oblíbenému klamu, že instance třídy `File` mají přímou vazbu na soubory. Není tomu tak. Instance třídy `File` jsou pouze objektovou reprezentací cesty. Reprezentovaná cesta se dané instanci přiřadí při volání jejího konstruktoru a pak už nejde změnit.

To, jestli tato cesta odpovídá nějakému reálnému souboru či složce, to není starost dané instance. Můžeš se jí na to ale kdykoliv zeptat a ona ti to zjistí. Přímou vazbu na nějaký soubor ale nemá.

78. Řekl bych, že když si odpustím deklarování veřejných konstant, budou proměnné typy výhodnější, protože mi připadají takové univerzálnější.

Neměnné typy jsou výhodnější

Opak je pravdou. Neměnné datové typy jsou mnohem výhodnější. Převážná většina hodnotových datových typů ve standardní knihovně je definována jako neměnná a u těch zbylých autory většinou mrzí, že je tak kdysi nedefinovali, protože nyní už to není možno změnit. Proměnnost nebo neměnnost daného typu je totiž součástí jeho kontraktu.

Hodnotový typ definuj vždy jako neměnný

Budeš-li potřebovat pro svoji aplikaci definovat nějaký hodnotový datový typ, snaž se jej vždy definovat jako neměnný. Proměnné hodnotové typy přinášejí řadu problémů. Jejich instance např. nemohou vytvářet množiny (leda bys definoval vlastní verzi nepoužívající hešovou tabulku), nelze je používat jako klíče běžných map a nejde s nimi dělat řadu dalších užitečných operací.

Vypíchl bych proto jednu důležitou zásadu:

**Hodnotové datové typy definuj zásadně jako neměnné.
K tomu, abys definoval hodnotový datový typ jako proměnný,
bys musel mít opravdu pádný důvod.**

79. Říkal jsi, že instance proměnných hodnotových typů není možné používat v běžných mapách. Z toho jsem pochopil, že v jakýchsi neběžných by to šlo.

Java 1.4 zavedla třídu `IdentityHashMap`, která pro vyhledání klíče nepoužívá standardní metody `equals(Object)` a `hashCode()`, ale pomocí operátoru == porovnává přímo instance. V takové mapě bys samozřejmě mohl použít jako klíče i instance proměn-

ných datových typů, ale obávám se, že by sis tak vykopal sám pro sebe jámu, do které bys vzápětí spadl. Ber to proto pouze jako cestu posledního zoufalství pro superspeciální případy.

Referenční datové typy

80. No dobře. A jak je to s těmi referenčními typy?

Podstata referenčních typů

U referenčních datových typů se po žádné hodnotě nepídíme. Mohli bychom říci, že jejich hodnotou je instance sama. Proto nám k porovnání stačí verze metody `equals(Object)` zděděná od třídy `Object`, která prohlásí dvě instance za ekvivalentní právě tehdy, jedná-li se o jednu a touž instanci.

Z tříd, s nimiž jsme doposud pracovali, bychom do této skupiny mohli zařadit všechny grafické objekty. Např. to, že dva obdélníky mají stejnou barvu, jsou stejně veliké a leží na stejných souřadnicích, nám nepřipadá jako důvod k tomu, abychom je považovali za shodné. Jsou to pro nás stále dva různé obdélníky, které se pouze v daný okamžik překrývají.

V řadě případů (i když ne vždy) nás u instancí referenčních datových typů zajímá pouze to, o kterou instanci se jedná, a nijak nám nevadí, že její vlastnosti v průběhu času mění někdo cizí (např. že ji posouvá nebo mění její rozměr).

Neměnnost instancí v praxi

81. Teoreticky ten rozdíl chápu. Teď mi to ještě přibliž k praxi.

Výhoda neměnných typů ve výrazech

Představ si, že bys chtěl definovat objekty, které bys chtěl používat v matematických výrazech. Říkali jsme si, že ve výrazech by měly vystupovat pouze objekty neměnných hodnotových typů. Pokud bys definoval tyto typy tak, že bys umožnil změnu hodnoty objektu, mohl by ses dočkat nepěkných překvapení.

Představ si, že bys definoval třídu zlomků s operací násobení podle programu ve výpisu 4.1. Spustíš-li její metodu `test()`, vypíše ti na standardní výstup:

```
Zlomek před operací: šza=[2/1], šzb=[2/1]
Zlomek po operaci:   šza=[4/1], šzb=[4/1]
```

Jak vidíš, hodnota zlomku `šza` se změnila, i když jsme s ním zdánlivě nic nedělali.

Výpis 4.1: `ŠpatnýZlomek` **– špatně definovaná třída zlomků**

```
package rup.česky.vzory._04_neměnné;


/*******************************************************************************
 * Ukázka špatně definované třídy zlomků.
 */
public class ŠpatnýZlomek
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private int čitatel, jmenovatel;
```

```
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří zlomek se zadaným čitatelem a jmenovatelem.
     * @param čitatel    čitatel vytvářeného zlomku
     * @param jmenovatel jmenovatel vytvářeného zlomku
     */
    public ŠpatnýZlomek( int čitatel, int jmenovatel )
    {
        this.čitatel    = čitatel;
        this.jmenovatel = jmenovatel;
    }


//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /*****************************************************************************
     * Vrátí součin dané instance a instance zadané jako parametr.
     * @return Vypočtený součin
     */
    public ŠpatnýZlomek krát( ŠpatnýZlomek šz )
    {
        čitatel    *= šz.čitatel;
        jmenovatel *= šz.jmenovatel;
        return this;
    }


    /*****************************************************************************
     * Převede instanci na řetězec.
     * @return Řetězcová reprezentace dané instance.
     */
    public String toString()
    {
        return "[" + čitatel + "/" + jmenovatel + "]";
    }


//== TESTY A METODA MAIN ========================================================

    /*****************************************************************************
     * Testovací metoda.
     */
    public static void test()
    {
        ŠpatnýZlomek šza = new ŠpatnýZlomek( 2, 1 );
        ŠpatnýZlomek šzb = šza;
        System.out.println("Zlomek před operací: šza=" + šza + ", šzb=" + šzb );
        šzb = šzb.krát( šzb );
        System.out.println("Zlomek po operací:   šza=" + šza + ", šzb=" + šzb );
    }
    public static void main( String[] args )  {  test();  }
}
```

82. Mám takový dojem, že mi tvorba tříd, jejichž instance budu používat v matematických výrazech, nehrozí.

Nepoužitelnost proměnných typů při ukládání do některých kontejnerů

Nejde jenom o matematické výrazy. Do problémů se můžeš dostat i ve chvíli, kdy budeš chtít uložit instanci do kontejneru, a to je velice častá operace. Uložíš-li hodnotovou instanci do kontejneru a pak ji změníš, nemusíš ji v kontejneru již nalézt.

83. Proč bych ji tam neměl nalézt?

Důvody této nepoužitelnosti

Protože např. množina mívá své instance uloženy v hešové tabulce na pozicích definovaných heš-kódem. Když se ale změní hodnoty atributů, změní se i hodnota heš-kódu. Když se po takové změně množiny zeptáš, zda danou položku obsahuje, bude ji množina hledat ve svých útrobách jinde, než kam si ji kdysi uložila, a protože ji tam nenajde, prohlásí, že položka v množině není.

Podívej se na program ve výpisu 4.2. Tam je potomek předchozích špatně definovaných zlomků, který přidává definici metod `equals(Object)` a `hashCode()`. Testovací metoda vytvoří vedle instancí `šza` a `šzb`, známých z minulého příkladu, ještě proměnnou `šzm`, která bude samostatně vytvořenou instancí s hodnotou stejnou, jako byla počáteční hodnota `šza`.

Pak uloží do množiny proměnnou `šza` a ověří, že o ní množina ví. Protože instance `šzm` má stejnou hodnotu, bude o ní program tvrdit, že je v množině také (u hodnotových typů vlastně neukládáme instance, ale hodnoty – přesněji reprezentanty hodnot).

Pak spočte druhou mocninu proměnné `šzb` a znovu se množiny zeptá na přítomnost instance `šza`. Množina ale bude tvrdit, že v ní instance není – vypíše:

```
Přítomnost v množině: šza-true,  šzm-true
Zlomek před operací:  šza=[2/1], šzb=[2/1]
Zlomek po operaci:    šza=[4/1], šzb=[4/1]
Přítomnost v množině: šza-false, šzm-false
```

Výpis 4.2: `ŠpatnýZlomek2` **– špatný zlomek doplněný o metody** `eauals(Object)` **a** `hashCode()`

```
package rup.česky.vzory._04_neměnné;

import java.util.HashSet;
import java.util.Set;

/*******************************************************************************
 * Ukázka rozšíření špatně definované třídy zlomků.
 */
public class ŠpatnýZlomek2 extends ŠpatnýZlomek
{
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*******************************************************************************
     * Vytvoří zlomek se zadaným čitatelem a jmenovatelem.
     * @param čitatel    čitatel vytvářeného zlomku
     * @param jmenovatel jmenovatel vytvářeného zlomku
     */
    public ŠpatnýZlomek2( int čitatel, int jmenovatel )
    {
```

```
        super( čitatel, jmenovatel );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * @param  Porovnávaný objekt
     * @return Informace o shodě hodnot zlomku se zadaným objektem
     */
    public boolean equals( Object o )
    {
        if( o instanceof ŠpatnýZlomek ) {
            ŠpatnýZlomek šz = (ŠpatnýZlomek)o;
            return (čitatel == šz.čitatel)  &&  (jmenovatel == šz.jmenovatel);
        }
        return false;
    }


    /**************************************************************************
     * @return Heš-kód dané instance.
     */
    public int hashCode()
    {
        return (717 + čitatel)*37 + jmenovatel;
    }


//== TESTY A METODA MAIN ========================================================

    /**************************************************************************
     * Testovací metoda.
     */
    public static void test()
    {
        ŠpatnýZlomek2 šza = new ŠpatnýZlomek2( 2, 1 );
        ŠpatnýZlomek2 šzb = šza;
        ŠpatnýZlomek2 šzm = new ŠpatnýZlomek2( 2, 1 );

        Set<ŠpatnýZlomek2> množ = new HashSet<ŠpatnýZlomek2>();

        množ.add(šza);
        System.out.println("Přítomnost v množině: šza-" + množ.contains(šza) +
                                                ", šzm-" + množ.contains(šzm) );
        System.out.println("Zlomek před operací:  šza=" + šza + ", šzb=" + šzb);
        šzb = (ŠpatnýZlomek2)šzb.krát( šzb );
        System.out.println("Zlomek po operací:    šza=" + šza + ", šzb=" + šzb);
        System.out.println("Přítomnost v množině: šza-" + množ.contains(šza) +
                                                ", šzm-" + množ.contains(šzm) );
    }
}
```

84. Vysvětli mi, prosím, ještě jednou, proč program ani jednu instanci v množině nenašel.

Jak jsem již říkal, instanci `šza` program v množině nenašel proto, že změnila svoji hodnotu a tím i heš-kód, takže ji v tabulce hledal jinde, než kam ji uložil. Instanci `šzm` sice hledal na správném místě, jenomže tam našel jenom `šza`, avšak ta již měla jinou hodnotu. Hodnotu odpovídající hodnotě `šzm` proto opět nenašel.

85. No, zavání to trochu magií, ale snad to chápu. Co když tedy definuji metodu `hashCode()` tak, aby vracela pořád stejný heš-kód?

Proč nelze definovat konstantní heš-kód

Pokud bys definoval stejný heš-kód pro všechny instance, tak bys je sice v množině pokaždé našel, ale při větším počtu instancí by hledání trvalo neúnosně dlouho, protože by v tabulce byly všechny instance na hromadě.

Pokud bys definoval metodu `hashCode()` tak, že by kód jednou spočetla a pak vracela pokaždé stejný, tak bys zase porušil kontrakt, který vyžaduje, aby všechny instance, které považuje metoda `equals(Object)` za sobě rovné, měly také stejný heš-kód.

Abys kontrakt dodržel, musel bys podrobně upravit také metodu `equals(Object)`, jenomže pak by zase neprohlásila za shodné ty instance, které by za shodné prohlásit měla.

Jediný způsob, jak se těmto problémům vyhnout, je definovat hodnotové objekty zásadně jako neměnné.

Další výhody neměnnosti

86. Tohle jsou docela pádné důvody. Vidím ti ale na očích, že bys určitě uměl přidat další.

Samozřejmě. A hned několik.

– jednodušší práce

- S neměnnými objekty se v řadě situací jednodušeji pracuje, protože se na jejich hodnotu můžeš spolehnout, a nemusíš proto vyrábět jejich kopie. Můžeš s nimi pracovat obdobně jako s hodnotami primitivních typů.

– vláknová bezpečnost

- Neměnné objekty jsou svojí podstatou vláknově bezpečné, takže nevyžadují synchronizaci. Vlákna si mohou být jistá, že je nikdo nezmění, takže je lze mezi vlákny sdílet.

– možnost sdílet stavy objektů

- Mezi vlákny je možné sdílet nejenom celé objekty, ale i jejich stavy a části jejich stavů (např. informaci o příponě názvu nějakého souboru).

Některé další podrobnosti o neměnných objektech a zásadách jejich tvorby najdeš v [28], kde je rozboru jejich problematiky věnována celá rada 13.

Implementace

87. Dobře, dobře. Už jsi mne přesvědčil. Teď mi ještě prozraď, jak má taková definice neměnného typu vypadat?

Pravidla tvorby neměnných typů

Jak jsem již říkal: nemáš-li opravdu pádný důvod k jinému řešení, měl bys hodnotové třídy definovat zásadně jako neměnné. Při jejich definici bys měl dodržet následující pravidla:

– konstantní atributy

- Definuj všechny jejich atributy, které určují hodnotu objektu, a které proto ovlivňují výsledky metod `equals(Object)` a `hashCode()`, jako konstantní (konečné, `final`). Tím zabezpečíš, že bude překladač hlídat, aby neměnnost instancí nebyla porušena.

– zapouzdřit odkazy na měnitelné objekty

- Obsahuje-li některý z atributů odkaz na měnitelný objekt, musíš zajistit, aby k tomuto měnitelnému objektu nemohl přistupovat nikdo jiný a neočekávaně změnit jeho hodnotu. Jinými slovy musíš zařídit, aby jediným vlastníkem tohoto měnitelné objektu (přesněji jediným, kdo může ovlivnit jeho hodnotu) byl tvůj neměnitelný objekt.

  Obdržíš-li proto tento objekt např. jako parametr konstruktoru, nesmíš jej do svého atributu převzít, ale musíš do atributu umístit jeho kopii, která je pro všechny ostatní nedosažitelná.

Obrat s použitím kopie objektu místo získaného originálu je jedním z hlavních rysů tzv. *defenzivního programování*. Více se o jeho zásadách dočteš např. v [41].

– metody nesmí modifikovat hodnotu – musí vracet nový objekt

- Metody, které získávají hodnotu odvozenou ze současné hodnoty objektu (např. výše předvedená metoda násobení nebo metoda třídy `String`, která převádí řetězec na velká písmena), nesmí modifikovat daný objekt, ale musí vytvořit nový objekt s požadovanou hodnotou a ten vrátit.

– definovat třídu jako konečnou

- Je-li to možné, měla by být třída konečná (tj. bez potomků) nebo by měla mít definovány své metody jako konečné, aby potomci nemohli změnit jejich neměnné chování a požadovat svoji neměnnost také v kontraktu.

88. Když ale musím vytvářet při každé změně stavu novou instanci, tak to může docela zdržovat.

Vytváření a rušení krátkodobých instancí je rychlé

Současné virtuální stroje jsou schopny pracovat s často vytvářenými a rychle zanikajícími objekty ve zvláštním režimu, který je pro takovýto druh práce optimalizován a nijak zvlášť nezdržuje.

Přístup u „velkých" typů

V některých situacích je ale neustálé vytváření nových instancí a jejich následné rušení opravdu nevhodné. U *velkých typů*, tj. u typů, jejichž instance mohou zabírat hodně paměti, je vhodné v takových případech doplnit neměnný typ typem s měnitelnou hodnotou, který se pak použije místo původního neměnného typu.

Příklad: `String`

Typickým příkladem je neměnná třída `String` a její doprovodné třídy `StringBuilder` a `StringBuffer`, které používáme tehdy, potřebujeme-li postupně vytvářet výsledný řetězec nebo se v něm nějak „přehrabovat".

Zvýšení efektivity u „malých" typů

U *malých typů* se tento problém řešívá tak, že často používané hodnoty definují jako své konstanty, a kdykoliv to je možné, tak jejich metody vrací místo nově vytvořené instance instanci některé z těchto předdefinovaných konstant.

Příklad: `Integer`

Typickým příkladem takto navržené třídy je třída `Integer`, která při svém zavedení vytvoří instance hodnot od `-128` do `127`. Je-li pak výsledkem nějaké operace hodnota uložitelná do bajtu (Java interpretuje i bajty jako čísla se znaménkem), vrátí volaná metoda odkaz na příslušnou konstantu. Nové instance těchto malých hodnot získáš

pouze přímým voláním konstruktoru. (Tato vlastnost může být i nebezpečná – podrobnosti se dozvíš např. v [31].)

89. Říkal jsi, že mají být konstantní atributy, které ovlivňují hodnoty vracené metodami `eauals(Object)` a `hashCode()`. Mohl bys mi dát příklad nějakého hodnotového objektu, který má také proměnné atributy?

Přiznám se, že mne žádný rozumný příklad nenapadá. Mohu ti nabídnout pouze jeden, který je tak trochu přitažený za vlasy.

Příklad neměnného objektu s proměnnými atributy

Představ si třídu `Barva`, jejíž instance (barvy) si budou pamatovat svůj název. Vznikne problém, jak pojmenovat barvu označovanou anglicky jako *cyan*. Někteří ji označují jako *blankytná*, někteří jako *azurová*. Můžeme tedy definovat dvě instance se stejnými barevnými složkami, avšak různými názvy.

Bude-li se shodnost instancí této třídy odvozovat pouze ze shodnosti barevných složek a nebude-li název vystupovat ani v metodě `hashCode()`, budou dvě instance specifikující stejný barevný odstín konzistentně ekvivalentní nezávisle na svém názvu. Dokonce budeš moci název dané barvy dynamicky měnit, aniž bys ovlivnil její dohledatelnost v množině či poli.

Sám však jistě cítíš, že to není optimální řešení. Lepším řešením pro většinu aplikací je dohodnout se na jediném názvu a nevnášet do aplikace zmatek. Tak je navržená i třída `Barva` v knihovně *Tvary*.

Příklad

90. Řekl bych, že o implementaci neměnných tříd jsem již poučen. Mohl bys mi na závěr ukázat správnou implementaci neměnné třídy na nějakém příkladu – např. na těch zlomcích?

Prohlédni si program ve výpisu 4.3. Tam jsem se pokusil vše ukázat.

Výpis 4.3: Třída `Zlomek` definovaná jako neměnný typ

```
package rup.česky.vzory._04_neměnné;


/*****************************************************************************
 * Třída Zlomek definuje zlomky a potřebné operace, aby se zlomky bylo
 * možno počítat obdobně jako s čísly. Definuje proto operace pro sčítání,
 * odčítání, násobení a dělení dvou zlomků a zlomku a čísla,
 * jakož i operace pro převod celého čísla na zlomek a zlomku na číslo.
*/
public class Zlomek extends Number
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final int čitatel;
    private final int jmenovatel;
```

```
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří novou instanci třídy Zlomek s hodnotou danou čitatelem a
     * jmenovatelem dodanými jako parametry. Hodnota čitatele a jmenovatele
     * uložená v atributech však bude již zkrácená a jmenovatel bude kladný.
     * @param c  Zadávaný čitatel
     * @param j  Zadávaný jmenovatel
     */
    public Zlomek(int c, int j)
    {
        if( j == 0 )
            throw new IllegalArgumentException(
                "Jmenovatel zlomku nesmí být nulový");
        int dělitel = Funkce.nsd( c, j );
        if( j < 0 )
        {
            c = -c;
            j = -j;
        }
        čitatel    = c / dělitel;
        jmenovatel = j / dělitel;
    }


    /*****************************************************************************
     * Kopírovací konstruktor – vytvoří novou instanci
     * se stejnými hodnotami atributů, jaké má zlomek zadaný jako parametr.
     * @param z  Kopírovaný zlomek
     */
    public Zlomek(Zlomek z)
    {
        čitatel    = z.čitatel;
        jmenovatel = z.jmenovatel;
    }


    /*****************************************************************************
     * Vytvoří zlomek, který bude mít hodnotu čísla zadaného jako parametr.
     * @param číslo  Hodnota vytvářeného zlomku
     */
    public Zlomek(int číslo)
    {
        čitatel    = číslo;
        jmenovatel = 1;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Vrátí hodnotu čitatele.
     * @return hodnota čitatele
     */
    public int getČitatel()
    {
        return čitatel;
```

```
    }


    /*****************************************************************************
     * Vrátí hodnotu jmenovatele.
     * @return hodnota jmenovatele
     */
    public int getJmenovatel()
    {
        return jmenovatel;
    }


    /*****************************************************************************
     * Vrátí řetězec reprezentující zlomek ve tvaru
     * [čitatel/jmenovatel]  -  např. [7/2]
     * @return Textová reprezentace zlomku.
     */
    public String toString()
    {
        return "[" + čitatel + "/" + jmenovatel + "]";
    }


    /*****************************************************************************
     * Vrátí hodnotu zlomku oříznutou na celé číslo.
     * @return  Hodnota zlomku oříznutá na celé číslo.
     */
    public int intValue()
    {
        return čitatel / jmenovatel;
    }


    /*****************************************************************************
     * Vrátí hodnotu zlomku převedenou na reálné číslo typu double.
     * @return  Hodnota zlomku převedená na reálné číslo typu double.
     */
    public double doubleValue()
    {
        return (double)čitatel / jmenovatel;
    }


    /*****************************************************************************
     * Připočte ke zlomku zlomek zadaný jako parametr
     * a vrátí zlomek, který je jejich součtem.
     * @param  z  Přičítaný zlomek.
     * @return Zlomek, který je součtem obou zlomků.
     */
    public Zlomek plus(Zlomek z)
    {
        //int násobek = Funkce.nsn( this.jmenovatel, z.jmenovatel );
        return new Zlomek( čitatel*z.jmenovatel + z.čitatel*jmenovatel,
                           jmenovatel*z.jmenovatel );
    }
```

```
/*****************************************************************************
 * Připočte ke zlomku celé číslo zadané jako parametr
 * a vrátí zlomek, který je jejich součtem.
 * @param  číslo   Přičítané číslo.
 * @return Zlomek, který je součtem obou hodnot.
 */
public Zlomek plus(int číslo)
{
    return new Zlomek( čitatel + číslo*jmenovatel, jmenovatel );
}


/*****************************************************************************
 * Odečte od zlomku zlomek zadaný jako parametr
 * a vrátí zlomek, který je jejich rozdílem.
 * @param  z  Odečítaný zlomek.
 * @return Zlomek, který je rozdílem obou zlomků.
 */
public Zlomek minus(Zlomek z)
{
    //int násobek = Funkce.nsn( this.jmenovatel, z.jmenovatel );
    return new Zlomek( čitatel*z.jmenovatel - z.čitatel*jmenovatel,
                       jmenovatel*z.jmenovatel );
}


/*****************************************************************************
 * Odečte od zlomku celé číslo zadané jako parametr
 * a vrátí zlomek, který je jejich rozdílem.
 * @param  číslo   Odečítané číslo.
 * @return Zlomek, který je rozdílem obou hodnot.
 */
public Zlomek minus(int číslo)
{
    return new Zlomek( čitatel - číslo*jmenovatel, jmenovatel );
}


/*****************************************************************************
 * Odečte zlomek od celého čísla zadaného jako parametr
 * a vrátí zlomek, který je jejich rozdílem.
 * @param  číslo   Číslo, od kterého se zlomek odečte.
 * @return Zlomek, který je rozdílem obou hodnot.
 */
public Zlomek odečtiOd(int číslo)
{
    return new Zlomek( číslo*jmenovatel - čitatel, jmenovatel );
}


/*****************************************************************************
 * Vynásobí zlomek zlomkem zadaným jako parametr
 * a vrátí zlomek, který je jejich součinem.
 * @param  z  Zlomek, kterým se násobí.
 * @return Zlomek, který je součinem obou zlomků.
```

```
     */
    public Zlomek krát(Zlomek z)
    {
        return new Zlomek( čitatel * z.čitatel,  jmenovatel * z.jmenovatel );
    }


    /*************************************************************************
     * Vynásobí zlomek celým číslem zadaným jako parametr
     * a vrátí zlomek, který je jejich součinem.
     * @param  číslo  Číslo, kterým se násobí.
     * @return Zlomek, který je součinem obou hodnot.
     */
    public Zlomek krát(int číslo)
    {
        return new Zlomek( čitatel * číslo,  jmenovatel );
    }


    /*************************************************************************
     * Vydělí zlomek zlomkem zadaným jako parametr
     * a vrátí zlomek, který je jejich podílem.
     * @param  z  Zlomek, kterým se dělí.
     * @return Zlomek, který je podílem obou zlomků.
     */
    public Zlomek děleno(Zlomek z)
    {
        return new Zlomek( čitatel * z.jmenovatel,  jmenovatel * z.čitatel );
    }


    /*************************************************************************
     * Vydělí zlomek celým číslem zadaným jako parametr
     * a vrátí zlomek, který je jejich podílem.
     * @param  z  Číslo, kterým se dělí.
     * @return Zlomek, který je podílem obou hodnot.
     */
    public Zlomek děleno(int číslo)
    {
        return new Zlomek( čitatel,  jmenovatel * číslo );
    }


    /*************************************************************************
     * Vydělí zlomkem celé číslo zadané jako parametr
     * a vrátí zlomek, který je jejich podílem.
     * @param  z  Číslo, které se dělí.
     * @return Zlomek, který je podílem obou hodnot.
     */
    public Zlomek dělČíslo(int číslo)
    {
        return new Zlomek( jmenovatel * číslo,  čitatel );
    }

    // ...  Další metody vyžadované rodičovskou třídou java.lang.Number
}
```

Výpis 4.4: Knihovní třída `Funkce` pro výpočet nejmenšího společného násobku a největšího společného dělitele

```
package rup.česky.vzory._04_neměnné;


/*******************************************************************************
 * Třída Funkce je knihovní třídou, jejíž metody řeší výpočet
 * nejmenšího společného násobku a největšího společného dělitele.
 */
public final class Funkce
{
//== OSTATNÍ METODY TŘÍDY ======================================================

    /***************************************************************************
     * Vrátí největšího společného dělitele dvou zadaných čísel.
     * Vracené číslo je vždy kladné nezávisle na znaménku parametrů.
     *
     * @param i1   První číslo
     * @param i2   Druhé číslo
     *
     * @return Největší společný dělitel zadaných čísel
     */
    public static int nsd(int i1, int i2)
    {
        i1 = Math.abs( i1 );
        i2 = Math.abs( i2 );
        while( i2 > 0 )
        {
            int pom = i1 % i2;
            i1 = i2;
            i2 = pom;
        }
        return i1;
    }


    /***************************************************************************
     * Vrátí nejmenší společný násobek zadaných čísel.
     * Vracené číslo je vždy kladné nezávisle na znaménku parametrů.
     *
     * @param i1   První číslo
     * @param i2   Druhé číslo
     *
     * @return Nejmenší společný násobek zadaných čísel
     */
    public static int nsn(int i1, int i2)
    {
        if( (i1 == 0)  ||  (i2 == 0) )
            return 0;
        return  i2 * Math.abs(i1) / nsd(i1,i2);
    }


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================
```

```
    /*************************************************************************
     * Třída nemá veřejný konstruktor => není možné vytvářet její instance.
     */
    private Funkce() {}
}
```

Příklad špatně definovaného potomka

Na závěr kapitoly ti dám ještě hádanku, kterou jsem ti sliboval na straně 48 mezi špatnými příklady v podkapitole *Přednost skládání před dědičností*. Prohlédni si znovu výpis 4.2 na straně 71 a zkus přijít na to, co je na definici třídy `ŠpatnýZlomek2` špatně – samozřejmě mimo to, že třída je definována jako proměnný hodnotový objektový typ.

91. No, určitě bych tam našel řadu věcí, ale ty se jistě ptáš na nějakou konkrétní. Co to je?

Trochu do tebe drcnu: metoda `equals(Object)` nedodržuje kontrakt, protože není symetrická, tj. nemusí být vždy pravda, že

```
a.equals(b) == b.equals(a)
```

92. Proč by neměla být symetrická. Přece když se první čitatel rovná druhému, bude se i druhý rovnat prvnímu a stejně to bude i se jmenovatelem.

To máš sice pravdu, jenomže zapomínáš na to, že jedna instance může být dceřiného typu `ŠpatnýZlomek2` a druhá rodičovského typu `ŠpatnýZlomek`. No a rodičovský typ používá metodu `equals(Object)` zděděnou od třídy `Object`, pro niž je, jak jistě víš, instance ekvivalentní pouze sama se sebou.

93. A kruciš. To je tedy opravdu pěkná – jak žes' to tehdy říkal? – plíživá kontrarevoluce.

Přiznám se, že jsem si to hned také neuvědomil – upozornila mne na to až lektorka. Původně jsem jenom nechtěl znovu opisovat stejný kód a ušetřit tak trochu papíru. Jak vidíš, cesta do pekel je lemována dobrými úmysly. Proto znovu připomínám: před každou snahou o nějaké ne zcela čisté zefektivnění programu nejprve vždy důkladně zapřemýšlej, jestli se místo zefektivnění programu nechystáš vzbudit nějakou tu „plíživku".

Shrnutí – co jsme se naučili

- Termín *hodnotové datové typy* se často používá pro primitivní datové typy a termín *referenční datové typy* pro objektové datové typy. Tyto termíny reagují na způsob předávání parametrů daných typů.
- V této knize budeme termíny *hodnotové* a *referenční* typy používat pouze v souvislosti s objektovými typy.
- Hodnotové objektové typy mohou být proměnné a neměnné.

- Instance proměnných hodnotových typů mohou měnit svoji hodnotu. To je vylučuje z řady použití.
- Nemáme-li pádný důvod k jinému řešení, měli bychom hodnotové typy definovat vždy jako neměnné.
- Neměnný datový typ by měl mít atributy, které ovlivňují jeho hodnotu (a od nichž se proto odvozuje výsledek volání metod `equals(Object)` a `hashCode()`), definovány jako konstantní (tj. s modifikátorem `final`).
- Metody neměnného typu, které mají modifikovat hodnotu své instance, musí vytvořit novou instanci s touto modifikovanou hodnotou a tu vrátit.
- Moderní virtuální stroje dokážou velmi efektivně vyřešit problém častého vzniku objektů, které velmi brzy zaniknou.
- U některých datových typů se problém nutnosti častého vytváření nových instancí řeší občas definicí sdruženého typu, který je měnitelný.
- U jiných datových typů se tento problém občas řeší definicí nejčastěji používaných hodnot jako konstant daného typu.
- *Neměnné objekty* nepatří mezi vzory z GoF.

KAPITOLA 5

Nenos mi to po jednom (Přepravka – Crate)

- **Účel**
- **Implementace**
- **Příklady ze standardní knihovny**
- **Interní přepravka**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Vzor *Přepravka* využijeme při potřebě sloučení několika samostatných informací do jednoho objektu, prostřednictvím nějž je pak možno tyto informace jednoduše ukládat nebo přenášet mezi metodami. V anglicky psané literatuře je tento návrhový vzor někdy označován termínem *Messenger*.

Účel

94. Problematickou konstrukci objektů jsme zvládli. Co nás čeká teď?

Problém: potřeba vrátit několik hodnot

Další problém. Často se v programech dostaneš do situace, kdy bys potřeboval, aby metoda vrátila několik hodnot současně – např. vodorovnou a svislou souřadnici nějakého objektu. Java však umožňuje vrátit pouze jednu hodnotu.

95. To znám, to jsme v Pascalu řešívali pomocí parametrů předávaných odkazem.

Výstupní parametry nepatří k dobrému stylu

Jenomže takové parametry Java nemá. Autoři Javy byli toho názoru, že vracení hodnot ve výstupních parametrech nepatří k dobrým programátorským zvykům, a proto takové parametry do jazyka vůbec nezavedli.

96. Mně ale vysvětlovali, že hodnotou se předávají pouze hodnoty primitivních datových typů, kdežto objekty se předávají odkazem.

Předání odkazu na objekt není předání objektu odkazem

To je oblíbená formulace. Bohužel není přesná. Na rozdíl např. od jazyka C++ v Javě nikdy nemáš v ruce (přesněji v proměnné) nějaký objekt. Vždy máš k dispozici pouze odkaz na objekt. Hovoříš-li proto kdykoliv o tom, že něco děláš s objektem (přiřadíš, porovnáš, vrátíš, …), znamená to, že to děláš s odkazem na objekt[1]. No a ten se v Javě předává hodnotou.

Metoda nemůže v parametru vrátit jiný odkaz než ten, jejž obdržela

Jinými slovy: v jazycích, které používají předávání parametrů odkazem, bys mohl předat v parametru proměnnou s odkazem na objekt a po skončení metody bys v proměnné našel odkaz na jiný objekt. To v Javě nejde. Jediné, co může metoda změnit, je stav objektu. Nikdy ti nemůže v parametru vrátit jiný objekt. To může pouze v návratové hodnotě.

97. Mohu ale definovat pole objektů, v němž bych potřebné instance předal a po skončení metody si je zase převzal.

To je sice pravda – tak bys předávání hodnotou obešel a někteří starší autoři přišedší ze strukturovaných jazyků takovýto postup opravdu doporučovali (místo pole bys mohl použít i jakýkoliv jiný měnitelný objekt). Moderní objektové programování však takový postup nedoporučuje, protože znepřehledňuje program. Vracení hodnot v parametrech je řazeno do kategorie vedlejších efektů metod. Takovýmto metodám se správný programátor snaží vyhnout.

98. A jak mám tedy řešit potřebu vrátit několik hodnot?

Definuje se pomocná třída – přepravka

Definuj si pomocnou třídu (nebo sáhni po nějaké existující), jejíž instance budou přenášet požadované hodnoty ve svých atributech. Já tuto třídu označuji jako přepravku, protože slouží k podobnému účelu.

Proč nepoužívám doslovný překlad termínu *Messenger*

V [13] a [14], kde jsem se s tímto vzorem (idiomem) poprvé setkal, jej sice autor označoval jako *Messenger*, což bychom mohli přeložit jako posel nebo kurýr, ale mně tento název nepřipadá výstižný. Posel či kurýr totiž zásilku aktivně nese a navíc ji

[1] Přesnější by asi byla formulace: Chceš-li něco dělat s objektem, předáš virtuálnímu stroji odkaz na objekt a svůj požadavek, co po objektu chceš, a virtuální stroj zařídí, aby objekt tvůj požadavek splnil (je-li toho objekt samozřejmě schopen).

nese schovanou před okolním světem. Naproti tomu přepravku si výdejce a příjemce předávají a její obsah bývá pro okolní svět přístupný. Obdobně funguje i přepravka v programech.

Implementace

Proč jsou atributy přepravky veřejné

99. Chceš tím naznačit, že atributy přepravky jsou veřejné?

Ano. Účelem zavedení přepravky je totiž především jednoduchý přenos hodnot od zdroje k příjemci. Veřejné atributy[1] usnadní práci s těmito hodnotami.

Typické použití

Když zavolám nějakou metodu a ta mi vrátí požadované hodnoty v přepravce, tak je hned zpracuji a přepravku zahodím (tj. smažu odkaz na ni a nechám její další osud na správci paměti). Nemusím se proto bát, že se bude někde povalovat a někdo nepovolaný by se k hodnotám mohl dostat.

Kdy deklarovat atributy soukromé

Potřebuješ-li ale skupinu hodnot v přepravce někam poslat a cítíš potřebu je ochránit, nic ti nebrání je deklarovat jako soukromé a pracovat s nimi prostřednictvím přístupových metod.

100. Všichni lektoři a učitelé doposud svorně tvrdili, že používání veřejných atributů je nebezpečné.

Kdy jsou veřejné atributy nebezpečné

Je to nebezpečné pouze v případě, kdy by mohl „náhodný kolemjdoucí“ změnit hodnotu tohoto atributu. Pak je ale úplně jedno, zda tuto hodnotu změní omylem (v programu je chyba) nebo záměrně („kolemjdoucím“ je nějaký záškodnický program).

Soukromé přepravky jsou bezpečné

Definuješ-li přepravku jako soukromý typ nějaké třídy, je zcela jedno, jaká přístupová práva jejím atributům dáš, protože její atributy budou beztak přístupné pouze v rámci své vnější třídy. Tam je to naprosto bezpečné a pokoušet se přistupovat k jejím atributům přes přístupové metody znamená znepřehledňovat program.

Veřejné přepravky definuj neměnné

Definuješ-li přepravku jako veřejnou, pak bych ti vřele doporučoval ji definovat jako neměnný typ, tj. definovat její atributy jako konstantní. Budou-li navíc její atributy obsahovat hodnoty primitivních typů nebo odkazy na instance neměnných hodnotových typů, nemůže program nic ohrozit.

Budou-li v jejích atributech odkazy na instance referenčních typů, nebo dokonce na instance proměnných hodnotových typů, budeš se muset rozhodnout, dáš-li v daném okamžiku přednost bezpečnosti, anebo efektivitě a pohodlnosti práce.

101. Takže máš někde připravené přepravky pro dvojice, trojice a další n-tice hodnot a vždy sáhneš po té, kterou zrovna potřebuješ.

Přepravky bývají jednoúčelové

Teoreticky by to šlo (zejména při využití parametrizovaných typů), ale nedělá se to. Na rozdíl od přepravek z běžného života, ve kterých většinou můžeš přepravovat

[1] Často stačí, jsou-li veřejné v oblasti, kde přepravku používám. Nehodlám-li přepravku použít nikde mimo daný balíček, mohou být její atributy deklarovány s implicitním nastavením přístupu označovaným často jako `package private`. (Přiznejme si, zbytečně velké zveřejňování škodí.)

téměř cokoliv, co se do nich vejde, bývají tyto přepravky vytvářeny jako přísně jednoúčelové.

102. Mohl bys to ilustrovat na nějakém příkladu?

Příklad použití přepravek

Na počátku jsem se zmiňoval o potřebě předávat dvojice souřadnic. Pro tento účel si vytvoříš přepravku, kterou pojmenuješ např. `Pozice` a která bude mít atributy pojmenované např. `x` a `y`.

Vzápětí nato budeš potřebovat někam předat rozměry. Nepoužiješ tutéž přepravku, i když by svými vlastnostmi vyhovovala (dva celočíselné atributy jsou právě to, co potřebuješ). Místo toho definuješ novou přepravku, kterou pojmenuješ např. `Rozměr` a která bude mít atributy `šířka` a `výška`.

103. To ale vypadá jako zbytečné plýtvání.

Důvod jednoúčelovosti přepravek

Vytvořením další přepravky zvýšíš spotřebu paměti jenom nepatrně, ale velice výrazně zvýšíš přehlednost programu.

Kdybys na přenosy všech dvojic celých čísel používal nějakou univerzální přepravku, dejme tomu `IntInt`, nemohl bys využívat možnost typové kontroly a za chvíli bys někam poslal pozici místo rozměru nebo se dopustil ještě větší chyby.

Standardní knihovna obsahuje často používané přepravky

Na druhou stranu je řada typických situací, v nichž se používají přepravky, takže se vyplatilo zařadit řadu takových typických přepravek do standardní knihovny. Je proto dobré znát přepravky, které jsou k dispozici ve standardní knihovně, a odpovídá-li jejich zaměření našim požadavkům, tak je použít.

104. Když jsou přepravky takové jednoúčelové, tak by mohlo být výhodné je vybavit i nějakými užitečnými metodami.

Doplnění přepravek o metody

Ano, a opravdu se to tak někdy dělá. Měly by to být ale metody sloužící především k manipulaci se svými přepravkami. Na druhou stranu řada počítačových guru takovýto přístup nedoporučuje. Jestli si vzpomínáš, říkali jsme si, že každá třída má mít pouze jeden úkol. No a v případě přepravek doplněných obecně použitelnými metodami vlastně definuješ třídu, která bude plnit dvě funkce: poskytovat instance sloužící jako přepravka a vedle toho sloužit jako knihovní třída poskytující nějaké obecně použitelné metody.

105. Co když je jedním z atributů pole. To jde sice udělat konstantní, avšak bude-li veřejné, bude moci jeho položky kdokoliv změnit.

Potřebuješ-li zabezpečit, aby pole v přepravce bylo neměnné, musíš je definovat jako soukromé a jeho položky zpřístupňovat prostřednictvím přístupové funkce. Ukázku takové přepravky najdeš třeba ve výpisu 30.2 na straně 405.

Příklady ze standardní knihovny

106. Hovořil jsi o přepravkách ve standardní knihovně. Dej mi nějaké příklady.

Příklady: `java.awt.Point`,

Když potřebuješ předat nějakou 2D pozici, můžeš k uložení souřadnic využít instance třídy `java.awt.Point` bez ohledu na to, bude-li se jednat o souřadnici bodu na obrazovce nebo o souřadnici kamene na šachovnici.

Dimension, Rectangle

Obdobně můžeš využít instance třídy `java.awt.Dimension` k uložení informací o rozměrech (přesněji o výšce a šířce) čehokoliv a instanci třídy `java.awt.Rectangle` k současnému uložení informací o pozici a rozměrech.

Nicméně definice těchto tříd jsou mezi pravověrnými „objektisty“ pomlouvány, protože je jejich autoři nedefinovali jako neměnné.

107. A příklad nějakých užitečných metod?

Příklady nadstavbových metod

Můžeme zůstat u třídy `java.awt.Point`. Vedle přístupových metod `getX()`, `getY()`, `setX(int)`, `setY(int)`, které jsou vzhledem k veřejné povaze jejích atributů prakticky zbytečné (a jak jsme si řekli, správně by hodnoty atributů neměly jít dodatečně nastavit), definuje např. i metodu `equals(Object)`, umožňující rychlé porovnání dvou souřadnic, metodu `translate(int,int)`, která posune souřadnice o zadanou vzdálenost, a další metody, které se mohou při práci s dvojrozměrnými souřadnicemi hodit.

108. Já jsem si vždy myslel, že instance třídy Point představují body.

Skutečný účel třídy `Point`

Když se podíváš do dokumentace, zjistíš, že instance třídy představuje *A point representing a location in (x, y) coordinate space, specified in integer precision.* (Bod reprezentující pozici v souřadné soustavě (x, y) zadaný s celočíselnou přesností.) Je to opravdu jenom obálka na informace pro určení pozice. Obálka, která umožňuje udržovat tyto informace pěkně pohromadě a pracovat s nimi jako s nedělitelným celkem – no a to je právě cíl přepravky.

Interní přepravka

109. Používají se přepravky i jinde než při vracení více hodnot?

Přepravky občas používají různé třídy pro svoji soukromou potřebu. Definují je pak většinou jako soukromé vnořené či vnitřní třídy a ukládají do nich skupiny hodnot – většinou proto, aby je mohly uložit pěkně pohromadě do nějakého kontejneru.

Příklad: Denní plán

110. To bych rád někde viděl.

Stručný popis

Připravil jsem si pro tebe takový jednoduchoučký plánovač denních akcí. Jeho metoda `přidej(int,int,int,String)` slouží k zařazení nové akce. Tato akce však nesmí kolidovat s žádnou z dříve zařazených akcí, jinak ji metoda nepřidá (o úspěšnosti pokusu referuje její návratová hodnota).

Plánovač si pamatuje začátek a konec každé z naplánovaných akcí spolu s textem, který oznamuje, o jakou akci se jedná. Informace o akci uloží do přepravky a tyto přepravky pak ukládá do seznamu seřazené podle času jednotlivých akcí.

Výpis 5.1: Třída `DenníPlán`

```
package rup.česky.vzory._05_přepravka;

import java.util.ArrayList;
import java.util.List;
```

```
import java.util.ListIterator;

/*******************************************************************************
 * Třída DenníPlán demonstruje použití interní přepravky na příkladu programu
 * realizujícího jednoduchý záznamník akcí naplánovaných na daný den.
 */
public class DenníPlán
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final List<Položka> akce = new ArrayList<Položka>();

//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Pokusí se přidat do časového plánu položku zadanou počátečním časem
     * a trváním a vrátí úspěch tohoto pokusu. Má-li se přidání položky podařit,
     * nesmí položka kolidovat s žádnou z dříve zadaných položek.
     */
    public boolean přidej( int hodina, int minuta, int trvání, String předmět )
    {
        int počátek = hodina*60 + minuta;
        int konec   = počátek + trvání;
        ListIterator<Položka> lip = akce.listIterator();
        while( lip.hasNext() ) {
            Položka p = lip.next();
            if( p.konec <= počátek ) {
                continue;
            }
            if( p.počátek <= konec ) {
                return false;
            }
        }
        lip.add( new Položka( počátek, konec, předmět ) );
        return true;
    }


    /*******************************************************************************
     * Vypíše seznam akcí naplánovaných na daný den.
     */
    public void vypiš()
    {
        System.out.println("\nSeznam akcí:");
        for( Položka p : akce ) {
            System.out.printf("%02d:%02d - %02d:%02d  %s\n",
                p.počátek/60, p.počátek%60, p.konec/60, p.konec%60, p.předmět );
        }
        System.out.println("———————————————-");
    }

//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /*******************************************************************************
     * Interní pomocná přepravka sloužící k uchování skupiny údajů,
     * které je třeba si o jednotlivých položkách pamatovat.
     */
```

```
    private static class Položka
    {
    //== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================
        int    počátek;
        int    konec;
        String předmět;

    //== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

        /*****************************************************************************
         * Konstruktor - inicializuje atributy.
         */
        Položka( int počátek, int konec, String předmět )
        {
            this.počátek = počátek;
            this.konec   = konec;
            this.předmět = předmět;
        }
    }


//== TESTY ======================================================================

    /*****************************************************************************
     * Pomocná testovací metoda požadující přidání zadané položky
     * a vypisující informace o přidávané položce, výsledku přidání
     * a stavu naplánovaných akcí po tomto pokusu.
     */
    public void tstPřidej( int h, int m, int t, String txt )
    {
        boolean úspěch;
        System.out.printf("Přidávám %d minut od %02d:%02d pro %s\n",
                          t, h, m, txt);
        úspěch = přidej( h, m, t, txt);
        System.out.println("          " + (úspěch? "ANO" : "NE") );
        vypiš();
    }


    /*****************************************************************************
     * Testovací metoda.
     */
    public static void test()
    {
        DenníPlán dp = new DenníPlán();
        dp.tstPřidej( 8,  0, 30, "Probuzení");
        dp.tstPřidej(10, 30, 90, "Relaxace" );
        dp.tstPřidej( 8, 30, 30, "Snídaně"  );
        dp.tstPřidej( 9, 30, 90, "Práce"    );
    }


    public static void main( String[] args )  {  test();  }
}
```

Další příklady v doprovodných programech

Mnohotvar

V kapitole *Baťovy cvičky (Prototyp – Prototype)* je na straně 285 uveden výpis třídy `Mnohotvar`, jejíž instance jsou složeninami jednodušších tvarů. Aby bylo možno tyto instance bez problému zvětšovat a zmenšovat, je pro ně definována soukromá přepravka, která uchovává klíčové rozměry jako čísla typu `double` (primárně jsou všechny hodnoty zadávány jako celočíselné).

Multipřesouvač

V doprovodné knihovně najdeš v balíčku `rup.česky.tvary` třídu `Multipřesouvač`, jejíž instance je schopna plynule přesouvat svěřené objekty do zadané cílové pozice. `Multipřesouvač` jsem ji nazval proto, že její instance je schopna přesouvat více objektů současně.

Aby s přesouváním objektů nezdržovala zbytek programu, probíhá tento přesun v samostatném vlákně. Jeho metoda `run` vždy přesune všechny přesouvané objekty o předem spočtenou vzdálenost a pak dané vlákno na nějakou dobu uspí.

Aby mohla přesouvat každý objekt nezávisle na ostatních přesouvaných objektech, potřebuje si o něm pamatovat některé informace. K jejich zapamatování je ve třídě definována soukromá přepravka, která slouží jako schránka na všechny potřebné informace.

Shrnutí – co jsme se naučili

- Přepravku využíváme nejčastěji ve dvou případech:
  - má-li metoda vracet několik hodnot současně – v takovém případě je přepravka definována jako třída viditelná pro volanou i volající metodu;
  - chceme-li v rámci třídy, resp. instance, uložit skupinu informací do soukromého kontejneru – pak bývá přepravka definována jako soukromá vnořená nebo vnitřní třída dané třídy.
- Přepravka má pro každou informaci z dané skupiny vyhrazen jeden atribut.
- Atributy přepravky se pro zvýšení efektivity většinou definují jako veřejné. Nesmí to však být na úkor bezpečnosti programu.
- Přepravky se většinou definují jako jednoúčelové.
- Programátoři občas rozšiřují přepravky o skupiny dalších užitečných metod, po jejichž implementaci přepravka často přestává být pouhou schránkou na data a přibližuje se běžným objektům.
- Ve standardní knihovně jsou typickými zástupci veřejně dostupných přepravek `java.awt.Point`, `java.awt.Dimension` a `java.awt.Rectangle`.
- Tyto třídy jsou zároveň příklady přepravek doplněných o další užitečné metody.
- Tyto třídy nejsou definovány jako neměnné, což způsobuje jisté problémy[1].
- V anglické literatuře se setkáte také s termínem *Messenger*, ale termín *Přepravka* lépe odpovídá vlastnostem a použití této třídy.
- *Přepravka* nepatří mezi vzory z GoF.

[1] Současní tvůrci standardní knihovny také tuto skutečnost svým předchůdcům vytýkají – viz [28], rada 13.

KAPITOLA 6

Udělám to za tebe (Služebník – Servant)

- **Účel**
- **Implementace**
- **Příklad: Přesouvač**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Návrhový vzor *Služebník* použijeme v situaci, kdy chceme skupině tříd nabídnout nějakou další funkčnost, aniž bychom zabudovávali reakci na příslušnou zprávu do každé z nich. *Služebník* je třída, jejíž instance (případně i ona sama) poskytují metody, které si vezmou potřebnou činnost (službu) na starost, přičemž objekty, s nimiž (nebo pro něž) danou činnost vykonávají, přebírají jako parametry.

Účel

111. Přepravka slouží k uchovávání skupiny dat. Existuje také něco k uchovávání skupiny funkcí?

Záleží na tom, co si pod onou skupinou funkcí představuješ. Pokud se domníváš, že bychom vytvořili nějakou přepravku na funkce (metody), které by obsluhovaly instance skupiny tříd, mohu ti posloužit návrhovým vzorem *Služebník*.

112. Zatím se moc nechytám – zkus jej vysvětlit podrobněji.

Úkol: přidat skupině tříd nějakou dovednost

Představ si, že mám skupinu tříd, jejichž instance bych rád naučil nějakou novou činnost, tj. chtěl bych pro ně definovat reakci na nějakou novou zprávu. Abych byl konkrétnější: mám např. třídy `Obdélník`, `Elipsa` a `Trojúhelník`, jejichž instance jsou schopny se přemístit (skokem) na zadanou pozici, a já se je rozhodnu všechny naučit přesouvat se plynule.

113. Na tom není co řešit – definuji v každé z nich příslušnou metodu.

Proč není vhodné vložit do každé třídy novou metodu

To je sice možné, ale porušíš tak jednu důležitou programátorskou zásadu, a to že bys neměl mít stejný kód na několika místech. Jinými slovy, měl by ses vyvarovat kopírování kódu. Kopie kódu totiž přinášejí obdobné problémy jako nepojmenované konstanty – literály: jakmile objevíš v kódu chybu, musíš oběhnout všechna místa, kde je daný kód použit, a na všech místech nalezenou chybu opravit. Zkušenost však ukazuje, že většinou buď nejméně jeden výskyt přehlédneš a/nebo někde doplníš metodu špatně. Zkrátka a dobře kopírování částí kódu není ten nejlepší programátorský postup.

Implementace

114. Takže co radíš?

Doporučený postup

Jednou z možností je definovat třídu, jejíž instance budou tuto úlohu řešit místo instancí původních tříd. Tuto třídu označíme jako *služebníka* a původní třídy pak označíme jako *obsluhované*. Budeš-li pak chtít po instanci některé z obsluhovaných tříd splnit zadanou úlohu (např. plynulý přesun), požádáš o tuto úlohu služebníka a obsluhovanou instanci mu předáš jako parametr.

115. Jenomže to jsi kód jenom přenesl na jiné místo. Stále se ti opakuje, protože nemají-li obsluhované třídy nějakého společného rodiče, musíš ve služebníkovi pro každou z nich definovat speciální přetíženou verzi obsluhující metody.

Krok za krokem:

Nemusíš. Při návrhu služebníka se totiž postupuje trochu jinak:

– Analýza požadavků

1. V prvním kroku zanalyzuješ, co má mít služebník na starosti. Musíš si ujasnit, jaké metody musí služebník definovat a co budou tyto metody potřebovat od obsluhovaného parametru. Jinými slovy: co bude muset obsluhovaná instance umět, aby služebníkovy metody mohly bezpečně splnit svůj úkol.

– Definice rozhraní

2. V druhém kroku definuješ `interface`, v němž deklaruješ požadované vlastnosti obsluhovaného parametru, tj. vyjmenuješ, na jaké zprávy bude muset umět obsluhovaný parametr reagovat (jinými slovy: jaké metody musí implementovat), aby mohl být služebníkem obsloužen. Bude-li chtít nějaká instance využít služeb metod služebníka, bude muset implementovat tento `interface`. Ten vlastně zastupuje onoho společného rodiče.

– Získání služebníka

3. Ve třetím kroku definuješ (nebo jinak získáš) příslušného služebníka (tj. jeho třídu).

– Implementace rozhraní

4. Ve čtvrtém, závěrečném kroku implementuješ definované rozhraní obsluhovanými třídami, tj. třídami, s jejichž instancemi mají metody služebníka a/nebo jeho instancí pracovat.

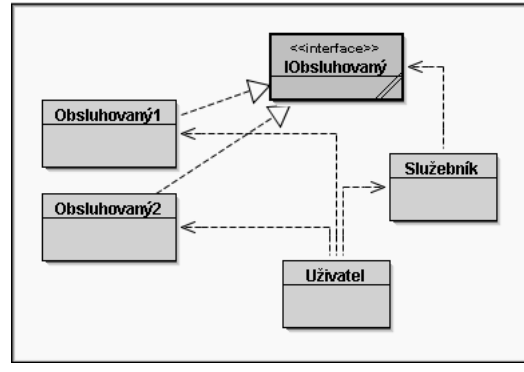
Pak už je vše jednoduché: kdykoliv potřebuješ, aby instance vykonala nějakou „obslouženou“ akci, zavoláš příslušnou metodu služebníka, předáš jí instanci jako parametr a necháš danou akci provést služebníkem.

116. Metodu služebníka volá obsluhovaná instance, anebo ten, kdo po ní něco chce?

Obě možnosti jsou použitelné. Záleží na tom, jak budeš mít vše implementováno. Pokusím se ti to ukázat na obrázcích.

Posílání zpráv služebníkovi

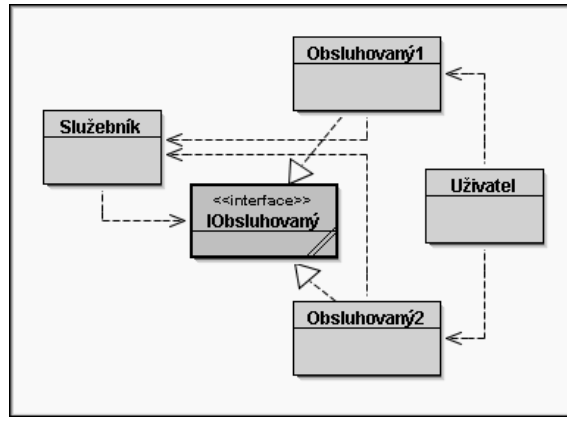
Na obrázku 6.1 implementují třídy s obsluhovanými instancemi rozhraní `IObsluhovaný`, avšak tváří se, že o třídě `Služebník` nic nevědí. Takto je možné návrhový vzor implementovat tehdy, budeš-li o příslušné služby žádat přímo instance služebníka a obsluhované instance mu budeš předávat jako parametry.



Obrázek 6.1
Uživatel požaduje operace po služebníkovi a obsluhované objekty předává jako parametr

Posílání zpráv obsluhovaným instancím

Na obrázku 6.2 naopak o třídě `Služebník` nemusí nic vědět uživatel, který bude posílat zprávy se svými požadavky přímo obsluhovaným instancím. Ty pak samy požádají instanci služebníka, aby je obsloužila a pomohla jim tak splnit příslušný požadavek.



Obrázek 6.2
Uživatel požaduje operace po obsluhovaných instancích, které pak samy požádají služebníka o svoji obsluhu

Příklad: Přesouvač

117. Nějak nechápu, proč musí obsluhovaný něco umět, aby jej mohl služebník obsluhovat. Zkus mi např. vysvětlit, jak bys to dělal s těmi elipsami a obdélníky?

Proč musí obsluhovaný něco umět

Přesně tak, jak jsem právě popsal (pro zjednodušení nebudu rozlišovat instanci a grafický objekt, který daná instance reprezentuje):

1. Mám-li posouvat instanci zdánlivě plynule po obrazovce, musím ji umět umístit na zadanou pozici a řadu mezipozic. Potřebuji proto, aby obsluhované instance definovaly např. metodu `setPozice(Pozice)` nebo nějaký její ekvivalent.

   Mám-li ale spočítat mezipozice, musím umět zjistit, kde se instance právě nachází. Potřebuji proto, aby obsluhované instance definovaly také metodu `getPozice()`, která bude vracet aktuální pozici.

   Budu-li vědět, že obsluhovaná instance umí reagovat na obě popsané zprávy, mohu na základě současné pozice a zadané cílové pozice zjistit, do jaké posloupnosti mezipozic mám přesouvanou instanci umisťovat, a postupně ji přesouvat z jedné mezipozice do druhé, až ji zdánlivě plynule přesunu ze zdrojové pozice do pozice cílové. To je standardní animace – tu jistě znáš.
2. Abych specifikoval své požadavky na obsluhované instance, definuji rozhraní `IPosuvný`, které bude od implementujících tříd požadovat implementaci metod pro zjištění a nastavení pozice – např. metod `getPozice()` a `setPozice(Pozice)`. (Místo třídy pozice mohu jako přepravku pro zadávání pozice použít třídu `java.awt.Point`.)
3. Definuji třídu `Přesouvač` (to je ten služebník), která bude nabízet dvě metody:

- metodu `přesuňDo(IPosuvný,Pozice)`, která plynule přesune zadanou instanci z jeho výchozí pozice do zadané cílové pozice, a
- metodu `přesuňO(IPosuvný, int, int)`, která plynule přesune svůj parametr z jeho výchozí pozice o zadanou vodorovnou a svislou vzdálenost.

4. Upravím definice všech tříd obrazců, tj. definice tříd `Obdélník`, `Elipsa` a `Trojúhelník` tak, aby implementovaly rozhraní `IPosuvný`. Vzhledem k tomu, že všechny tyto třídy již mají potřebné metody definované, stačí pouze deklarovat implementaci daného rozhraní v hlavičce třídy.

Výsledné definice všech tříd si můžeš prohlédnout v doprovodných programech.

Shrnutí – co jsme se naučili

- Návrhový vzor *Služebník* použijeme v případě, chceme-li skupině tříd přidat nějakou schopnost a nechceme-li definovat příslušné metody v každé z nich.
- Při implementaci návrhového vzoru *Služebník* postupujeme následovně:
  1. V analýze zjistíme, jaké schopnosti musí mít obsluhované třídy, aby je bylo možno řádně obsluhovat.
  2. Definujeme rozhraní, které bude po implementujících třídách vyžadovat implementaci deklarovaných metod, případně najdeme jeho ekvivalent.
  3. Definujeme třídu služebníka, jejímž metodám budeme předávat jako parametr instanci právě definovaného (použitého) rozhraní.
  4. Upravíme obsluhované třídy, aby implementovaly toto rozhraní.
- Existují dva způsoby implementace a z nich odvozených použití:
  - Uživatel služebníka zná (obsluhované instance jej pak znát nemusí) a zprávy se svými požadavky zasílá přímo jeho instancím, přičemž obsluhované instance jim předává jako parametry.
  - Služebníka znají obsluhované instance a uživatel posílá zprávy se svými požadavky přímo jim (uživatel pak služebníka znát nemusí). Obsluhované instance pak samy pošlou instancím služebníka zprávy, v nichž je požádají o svoji obsluhu.
- *Služebník* nepatří mezi vzory z GoF.

KAPITOLA 7

I nic může být objekt (Prázdný objekt – Null Object)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Prázdný objekt je platný, formálně plnohodnotný objekt, který použijeme v situaci, kdy by nám použití klasického prázdného ukazatele null přinášelo nějaké problémy – např. by vyžadovalo neustále testování „odkazuj na prázdnost“ nebo by vedlo k vyvolání výjimky NullPointerException.

Účel

118. Co si mám představit pod pojmem *Prázdný objekt*? To má být objekt, do nějž jsem ještě nic neuložil?

Co je to prázdný objekt

Ne. Prázdný objekt je speciální objekt, který se používá v situacích, kdy by bez použití prázdného objektu proměnná nikam neukazovala, protože by obsahovala odkaz `null`.

Kdy se hodí

V programech se musíš velice často vypořádávat se situací, kdy bys potřeboval poslat nějakému objektu zprávu, ale musíš nejprve ošetřit, zda příslušný odkaz vůbec někam ukazuje. Objektu `null` se totiž žádná zpráva poslat nedá.

119. Přiznám se, že mne to věčné testování „nullovosti" objektů občas obtěžuje. Jenom mne zatím nenapadlo, jak se z toho vylhat.

Jednou z možností je právě definice prázdného objektu, tj. objektu, na nějž bude příslušná konstanta či proměnná odkazovat v případě, kdy by jinak obsahovala prázdný odkaz – `null`.

120. No jo, ale co tím získám? Budu muset místo prázdného odkazu testovat přítomnost prázdného objektu.

Prázdný objekt je formálně plnohodnotný objekt

V tom je právě ta finta, že nebudeš. Prázdný objekt je formálně plnohodnotný objekt, takže mu můžeš poslat libovolnou zprávu, aniž by ti hrozilo vyvolání výjimky `NullPointerException` nebo nějaké od ní odvozené kolegyně.

Hlavně tím ale získáš to, že se ti na řadě míst nebude opakovat obdobný kód, protože kód, který by se měl provádět v případě, kdy narazíš na prázdný odkaz, přesuneš do definice prázdného objektu.

Implementace

121. To už je zajímavější, ale stále mi připadá, že pouze vyměním podmínky v okolním kódu za podmínky v definicích metod třídy, mezi jejímiž instancemi bude i prázdný objekt.

Prázdný objekt bývá instancí potomka

Od toho si ale můžeš odpomoci. Stačí definovat prázdný objekt jako instanci třídy, která bude potomkem té tvojí a která překryje všechny metody, před jejichž voláním je třeba otestovat prázdnost odkazu a zvolit v tomto případě místo volání metody nějaký alternativní postup. Onen alternativní postup pak definuješ ve třídě prázdného objektu jako tělo překrývající verze dané metody.

Třída prázdného objektu přitom musí mít evidentně pouze jedinou instanci, takže ji definuješ podle návrhového vzoru *Jedináček*.

122. Když zavedu prázdný objekt hned při návrhu aplikace, bude to asi jednoduché. Dodatečného zavádění prázdného objektu do nějaké rozsáhlejší aplikace bych se ale asi bál.

Co je třeba zabezpečit

Při zavádění prázdného objektu do existující aplikace se musíš postarat o několik věcí:

- prověřit, před voláním kterých metod je třeba testovat prázdnost odkazu,
- ujasnit si, jak by měl prázdný objekt v případě volání těchto metod správně reagovat (prázdné tělo nebývá vždy tím nejlepším řešením),
- zabezpečit, aby se v příslušných konstantách a proměnných objevil místo prázdného ukazatele vždy odkaz na prázdný objekt.

Pak už stačí jenom projít program a odstranit příslušné testy „nullovosti" odkazu, protože přestanou být potřeba.

Podrobný popis zavádění prázdného objektu do existující aplikace najdeš včetně dalšího výkladu např. v knize [21] na str. 245 až 251 (tam o něm hovoří jako o objektu null), resp. v jejím anglickém originále [26] na str. 260 až 266.

123. A co když bude mít aplikace reagovat na prázdný odkaz v různých situacích různě?

Viděl jsem sice v literatuře doporučení, že bys mohl mít pro takovouto situaci připraveno několik prázdných objektů, ale mně to pak již připadalo trochu násilné. Radím ti, aby ses vždy rozhodoval hlavou – někdy to může být výhodné (myslím těch několik prázdných objektů, rozhodování hlavou je výhodné vždy), jindy to bude nesmysl.

Příklad

124. Tak teď už zbývá jenom demonstrovat vše na nějakém krásném, snadno pochopitelném příkladu.

Nechtěl jsem pro prázdný objekt vymýšlet nějaký samoúčelný AHA-příklad, tak jsem se rozhodl, že počkám, až potřeba nějakého prázdného objektu přirozeně vyplyne. A stalo se, a to hned dvakrát. Oba dva prázdné objekty si budeme definovat v pasáži *Prázdný iterátor a iterovatelný objekt* na straně 218 jako příklady ke kapitole *Moc se mi v tom nebrab (Iterátor – Iterator)*.

Shrnutí – co jsme se naučili

- V řadě situací bývá výhodné nahradit přiřazení prázdného odkazu přiřazením odkazu na prázdný objekt.
- Prázdný objekt definujeme jako instanci třídy, která je potomkem třídy neprázdných objektů.
- Třída prázdného objektu bývá definována podle návrhového vzoru *Jedináček*, aby byla zabezpečena jedinečnost prázdného objektu.
- Akci, kterou je v případě prázdnosti odkazu třeba provést místo volání metody, definujeme ve třídě prázdného objektu jako překrývající verzi této metody.
- *Prázdný objekt* nepatří mezi vzory z GoF.

ČÁST 2

Ovlivňujeme počet instancí

Druhá část se soustředí na návrhové vzory, které specifikují, jak je možno ovlivnit počet vytvořených a používaných instancí zadané třídy. Postupně si probereme situace od nejjednodušší, kdy potřebujeme, aby třída neměla žádnou instanci, přes situace, kdy potřebujeme, aby měla právě jednu instanci, situace s větším, avšak předem omezeným počtem instancí až k situacím, kdy by bylo potřeba vytvořit daleko větší počet instancí, než je vzhledem k možnostem použitého počítače únosné.

KAPITOLA 8

Žádná instance (Knihovní třída – Library Class)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Knihovní třída slouží jako obálka pro soubor statických metod. Protože k tomu nepotřebuje vytvářet instance, je vhodné jejich vytváření znemožnit. Pro knihovní třídu se v anglické literatuře někdy používá termín *Utility*.

125. Tak teď už začneme skutečnými vzory?

Pořád mluvíme o skutečných vzorech. Jenom se zatím nejednalo o vzory publikované v GoF. Možná některé ještě nejsou plnohodnotnými vzory, avšak jejich znalost je stejně důležitá jako znalost vzorů z GoF.

126. Takže čemu se budeme věnovat?

Budeme si postupně ukazovat, jak lze ovlivnit počet instancí dané třídy. Začneme s něčím naprosto jednoduchým – řekl bych až triviálním: knihovní třídou. (I tento vzor je tak triviální, že jej v GoF nenajdeš.)

Účel

127. Knihovní třídou? Co to je?

Co to je knihovní třída

Ze standardní knihovny bys měl znát minimálně čtyři knihovní třídy:

- `java.lang.Math` sdružuje často používané funkce nad primitivními číselnými typy,
- `java.lang.System` sdružuje poměrně nesourodou skupinu metod pro komunikaci s virtuálním strojem a operačním systémem,
- `java.util.Arrays` sdružuje metody pro práci s poli a
- `java.util.Collections` sdružuje metody pro práci s kontejnery.

Tyto třídy mají jedno společné: jsou pouze obálkami pro skupinu statických metod. Nepotřebují vytvářet instance a vytváření jejich instancí je dokonce považováno za nežádoucí.

Protože tato úloha (tj. zamezit vytvoření instance) je opravdu triviální, předpokládám, že víš, jak ji vyřešit.

Implementace

128. Definuji pro ni jediný konstruktor, který bude soukromý, bezparametrický a s prázdným tělem.

Jak zamezit vytváření instancí

Správně. Navíc je vhodné deklarovat třídu jako konečnou (`final`). Nemožnost vytvářet její potomky je sice dána již oním jediným soukromým konstruktorem, ale takto bude deklarována průzračněji. Navíc ti při pokusu o vytvoření jejího potomka nebude překladač vysvětlovat, že třída nemá dostupný potřebný konstruktor, ale rovnou ti oznámí, že tato třída je konečná, a proto potomky mít nemůže.

129. Takto mám tedy postupovat pokaždé, když budu potřebovat definovat nějakou skupinu statických metod?

Kdy zavést knihovní třídu

Ano. Když se ti v aplikaci začnou množit statické metody, je rozumné se zamyslet nad tím, je-li výhodnější je rozpustit mezi jednotlivými třídami, anebo je naopak sdružit všechny do jedné třídy, kterou pak budeš definovat jako knihovní.

Obecně platí, že nejsou-li statické metody přímo vázány na konkrétní třídu, bývá vhodné definovat pro skupinu logicky souvisejících metod jejich společné bydliště – knihovní třídu.

Vezmi si příklad z třídy `Math`. Zde bys mohl uvažovat, jestli by metody `abs`, `min` a `max` bylo lepší umístit do tříd odpovídajících typu parametrů (přesněji jejich obalovým typům), tj. typům `Integer`, `Long`, `Float` a `Double`, anebo jestli pro ně naopak vytvořit zvláštní třídu knihovní a umístit je všechny pohromadě. Jistě cítíš, že společné umístění metod v jedné knihovní třídě je výhodnější. (Návrháři standardní knihovny měli rozhodování jednodušší, protože spolu s těmito metodami umístili do dané třídy ještě řadu dalších metod.)

Příklad

130. Obávám se, že tentokrát příklad nebude.

Proč by nebyl? Jenom ti jej tu nebudu zobrazovat. Vyjmenoval jsem ti knihovní třídy ze standardní knihovny, jejíž zdrojový kód je součástí JDK. Tam si jej můžeš vyhledat.

Dalším příkladem je třída `Funkce`, kterou jsme používali při definici tříd zlomků a jejíž zdrojový kód najdeš ve výpisu 4.4 na straně 80.

Knihovní třídou je i třída `rup.česky.společně.IO`, jejíž zdrojový kód je součástí knihovny *Tvary*.

Shrnutí – co jsme se naučili

- Potřebujeme-li definovat skupinu statických metod, bývá výhodné je společně umístit do tzv. *knihovní třídy* (library class).
- Knihovní třída by neměla umožňovat vytvoření svých instancí. Toho dosáhneme definicí soukromého konstruktoru (nejlépe bezparametrického s prázdným tělem) a deklarací třídy jako konečné (`final`).
- Knihovní třída nepatří mezi vzory z GoF.

KAPITOLA 9

Jediná instance (Jedináček – Singleton)

- **Účel**
- **Základní implementace**
- **Vícevláknové aplikace**
- **Serializovatelnost**
- **Speciální případy**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Jedináček specifikuje, jak vytvořit třídu, která bude mít nejvýše jednu instanci. Tato instance přitom nemusí být vlastní instancí dané třídy.

[1] **Definice v GoF:** Ensure a class only has one instance, and provide a global point of access to it. – Zabezpečí, že třída bude mít jedinou instanci a poskytne k ní globální přístupový bod.

Účel

131. Knihovní třída byla opravdu jednoduchá. Co tam máš dalšího?

U jednoduchých vzorů zůstaneme. Říkali jsme si, že u knihovní třídy je nežádoucí, aby měla nějakou instanci. Nyní si ukážeme, jak zabezpečit, aby třída měla právě jednu instanci – ani víc, ani míň.

Účel vzoru

Vytvoření třídy, která bude mít jedinou instanci, popisuje návrhový vzor označovaný v anglické literatuře jako *singleton*, což bychom mohli do češtiny přeložit jako *jedináček*.

132. K čemu je taková třída dobrá?

Příklady použití

Nutnost existence jediné instance se v programu objevuje poměrně často. V některé z předchozích kapitol jsem uváděl příklad světa robotů, v němž se všichni roboti pohybovali po společném dvorku. Třída definující dvorek vytvářela svoji instanci jako jedináčka.

Jako jedináčci jsou často implementovány různé fondy (pool – budeme o nich hovořit v kapitole *Konečný počet instancí (Fond – Pool)* na straně 151), zápisníkové paměti (cash memory), dialogová okna, ovladače různých zařízení a řada dalších objektů.

Známým příkladem z operačního systému je např. schránka, jejímž prostřednictvím přesouváš či kopíruješ data mezi aplikacemi. Standardní schránka je pro všechny aplikace společná. Když do ní v některé spuštěné aplikaci něco uložíš, smažeš to, co v ní bylo doposud.

133. Schránka je zajímavý příklad. Přidáš ještě nějaký ze standardní knihovny?

Standardní knihovna

Z těch známějších tříd sem žádná nepatří. Ve standardní knihovně je sice řada tříd, které produkují pouze jedináčky (dáš-li si vyhledat toto slovo ve zdrojových kódech, najdeš je v 85 souborech), ale jsou to buď interní anebo méně používané třídy pokrývající specializované oblasti.

Používají jej další návrhové vzory

Neboj se ale, že by sis jedináčka dost neužil – setkáš se s ním ještě při implementaci několika dalších návrhových vzorů.

134. Nepochopil jsem úplně přesně, proč potřebuje mít daná třída svoji instanci. Kdyby neměla žádnou instanci, byl by výsledek stejný, protože onu instanci by mohla zastupovat sama třída.

Zdánlivě máš sice pravdu, ale není to zase tak jednoduché, jak to na první pohled vypadá.

Proč není vhodné používat místo jedináčka jeho třídu

Tím, že místo třídy použiješ objekt, získáš řadu možností: tento objekt může implementovat nějaké rozhraní, odkaz na něj můžeš předat jako parametr, a dokonce si můžeš až na poslední chvíli vybrat, která třída jej „porodí“, tj. čí bude vlastní instancí. To by ti třída neumožnila.

Druhým problémem je to, že zavádění a inicializaci tříd má na starosti Java (přesněji JVM), a budeš-li mít takových tříd víc a budou všelijak vzájemně propojené, může se stát, že JVM nevybere pro inicializaci optimální pořadí. Mohou tak vzniknout různé záludné a těžko odhalitelné chyby. Při používání instancí tříd můžeš pořadí jejich inicializace mnohem lépe ovlivnit.

Obecně se proto používání tříd místo objektů příliš nedoporučuje, a to ani v případech, kdy je daný objekt jedináčkem.

Základní implementace

135. Jaká je tedy doporučená implementace jedináčka?

Doporučení je celá řada. Záleží na druhu objektu, který má být oním jedináčkem, a na řadě dalších okolností. Postupně si je probereme.

Soukromý konstruktor + statický atribut

Implementace jedináčka mají jedno společné: definují konstruktor jako soukromý, čímž komukoliv cizímu zabrání ve vytvoření další instance. Požadovanou instanci pak vytvoří uvnitř třídy a odkaz na ni uloží do statického atributu. Jednotlivé varianty implementace se pak liší tím, kdy a jak objekt konstruují a jak jej mohou ti ostatní získat.

Je vhodné označit třídu také jako finální

Když už v třídě definuješ pouze soukromý konstruktor, bývá vhodné ji současně označit jako finální, tj. jako třídu, která nemůže mít potomky. Se soukromým konstruktorem to stejně nejde a takto to alespoň bude viditelné na první pohled hned v signatuře třídy.

Časná inicializace = inicializace v deklaraci

136. Začni tou nejjednodušší verzí implementace.

Ta nejjednodušší implementace není zrovna doporučovaná, takže začnu nějakou maličko složitější, i když to bude zase pouze AHA-příklad. Typickou definici třídy, jejíž instance má být jedináček, ukazuje výpis 9.1.

Statický atribut inicializovaný v deklaraci + tovární metoda

Jak vidíš, statický atribut uchovávající odkaz na jedináčka je inicializován hned v deklaraci, a kdykoliv někdo potřebuje pracovat s jedináčkem, požádá o odkaz jednoduchou tovární metodu `geInstance()`.

V deklaraci by bylo možno inicializovat i atribut `krédo`, ale nechtěl jsem nechávat tělo konstruktoru prázdné, aby sis nemyslel, že to tak má být vždy. Tak jsem atribut `krédo` inicializoval v konstruktoru.

Výpis 9.1: `Jedináček` – příklad třídy s jedinou instancí

```
package rup.česky.vzory._09_jedináček;

/*******************************************************************************
 * Třída Jedináček demonstruje možnou implementaci jedináčka,
 * při níž je instance vytvořena již při definici statického atributu,
 * v němž budeme uchovávat odkaz na tuto jedinečnou instanci.
*/
public final class Jedináček
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final Jedináček jedináček = new Jedináček();
```

```
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private String krédo;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /**************************************************************************
     * Soukromý konstruktor zabrání vzniku nežádoucích instancí
     */
    private Jediináček()
    {
        krédo = "Do nedávna jsem byl namyšlený, ale nyní jsem již dokonalý";
    }


    /**************************************************************************
     * Jednoduchá tovární metoda vracející odkaz na jediináčka.
     */
    public static Jediináček getInstance()
    {
        return jediináček;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Vrátí text kréda.
     */
    public String toString()
    {
        return krédo;
    }

}
```

Odkaz ve veřejné konstantě

137. Jak mohu takovouto definici ještě zjednodušit?

Mohl bys deklarovat atribut `jediináček` jako veřejný a zrušit tovární metodu `getInstance()`.

Kdy se může zdát veřejná konstanta výhodnější

K takovémuto řešení můžeš inklinovat, budeš-li odkaz na jediináčka potřebovat velice často. Pak se totiž programátoři snaží vyhnout neustálému volání tovární metody, a dělávají si lokální kopie získaného odkazu. V tu chvíli ti může připadat výhodnější nahradit všechny ty lokální kopie jedinou veřejnou konstantou a uchovávat odkaz na jediináčka v ní.

Příklad: `SprávcePlátna`

Přiznám se, že v případě instance třídy `SprávcePlátna`, která se používá neustále kolem dokola, jsem se k takovému řešení také uchýlil, protože mne obtěžovalo dělat v každé třídě lokální kopie příslušného odkazu a volání tovární metody bylo pro moji línou duši vysoce obtěžující.

Nevhodné pro zřídka používané objekty

Na druhou stranu bych ti nedoporučoval uchylovat se k takovýmto zjednodušením příliš často. Globální konstanta (tj. konstanta, na kterou všichni vidí), na niž se obra-

cíš v každé druhé metodě, má své opodstatnění. Definovat však globální konstantu pro objekt, na nějž se obracíš na třech místech programu, není příliš moudré. Výmluva na zvýšení efektivity v tomto případě neobstojí.

Volání tovární metody nemusí zpomalovat program

Pamatuj si, že používání tovární metody je sice možná otravné, ale nemusí vést nutně ke zpomalení programu, protože optimalizační překladače mohou nahrazovat volání takovýchto jednoduchých metod přímým vkládáním jejich kódu, takže onen obrat se zveřejněním statického atributu provedou za tebe. Navíc jsme si říkali již na počátku, že předčasná snaha o optimalizaci je cestou do hrobu.

Námitky proti veřejné konstantě

Proč nepoužívat veřejnou konstantu

138. To, že tím program při použití optimalizačního překladače nezrychlím, asi nebude ten pravý důvod, proč se tento postup nedoporučuje. Ptám se tedy: „Proč se to tak nedělá, když je to jednodušší?“

Důvodů je celá řada:

– zbytečně odhaluje detaily implementace

- Používání veřejných atributů není obecně příliš doporučováno, protože tím zbytečně odhaluješ detaily implementace, které by ses měl naopak snažit co nejvíce skrývat. Jedině tak si totiž můžeš dovolit změnit v budoucnu implementaci, aniž by to mělo vliv na ty části programu, které upravovanou třídu používají.

  Použitím veřejného konstantního atributu všem prozrazuješ, že daná instance je jedináček. Kdyby ses proto později rozhodl, že instance dané třídy nemusí být nutně jedináček, bylo by zanášení změny do programu při použití veřejné konstanty mnohem obtížnější a především by se neobešlo bez změny rozhraní a tím i nutnosti přeprogramování programů, které tohoto jedináčka používají.

– nutí k příliš časné inicializaci

- Abys mohl atribut zveřejnit, musí se jednat o konstantu. Tu ale musíš inicializovat již v její deklaraci (nebo ve statickém konstruktoru), což někdy bývá příliš brzy na to, abys věděl vše, co k této inicializaci potřebuješ vědět. Třeba ještě vůbec nebudeš vědět, jakého typu má být vytvářená instance, anebo potřebuješ nejprve zjistit nějaké informace z konfiguračního souboru.

– nutí vytvořit instanci, i když nebude potřeba

- Použití konstanty tě nutí vytvořit instanci i v případě, kdy danou instanci nebudeš v programu potřebovat (např. budeš muset instalovat ovladač tiskárny i v případě, kdy si uživatel chce dokumenty jen prohlédnout a nehodlá je tisknout).

– nutí spoléhat na správnost pořadí, v němž JVM zavádí třídy

- Při použití inicializace v deklaraci jsi odkázán na to, že virtuální stroj zavede třídy ve správném pořadí. Při komplikovanějších závislostech se ale může stát, že se mu to nepovede. Ze špatného pořadí inicializace vzniká řada záludných a těžko odhalitelných chyb.

139. Zadrž! Skládáš to na hromadu páté přes deváté a nic nevysvětlíš. Říkal jsi, že při zavádění třídy třeba ještě nebudu vědět, jakého typu má být vytvářená instance. Jak mohu vybírat typ jedináčka, když je přece předem daný?

Jedináček s předem neznámým skutečným typem

Nemusí být. Vzpomeň si na tovární metodu `getČlověk()` ve výpisu 3.1 na straně 61. Ta vracela instance různých typů. Existují aplikace, v nichž má třída jedináčka několik potomků a tovární metoda se až na poslední chvíli rozhoduje, čí instance bude tím pravým jedináčkem. Při každém běhu programu proto může mít onen jedináček

jiný typ. Je to ale celé trochu složitější, takže bych k tomu teď neodbočoval a vrátil bych se k této otázce až za chvíli.

140. Dobře, budu si ji pamatovat. Už jsi mne skoro přesvědčil. Opatrně se ale zeptám: existují situace, kdy je používání veřejné statické konstanty užitečné?

Kdy je výhodné veřejnou statickou konstantu použít

Já bych po ní asi sáhl ve chvíli, kdy daná instance prostupuje celou aplikací a používá se velmi často. Pak totiž může její používání vést ke zpřehlednění programu. Musíš si ale být jist, že ti nehrozí žádný z výše vyjmenovaných problémů, tj. musí platit, že:

- instance je jedináček z podstaty problému, takže nehrozí, že budeš v budoucnu chtít rozšiřovat počty instancí,
- vazby mezi třídou jedináčka a ostatními třídami nejsou natolik komplikované, že by hrozila možnost nekorektní inicializace v důsledku nevhodného pořadí zavádění tříd,
- při zavádění třídy máš k dispozici všechny informace potřebné k vytvoření plnohodnotné instance,
- vytvoření inicializace není časově náročné anebo ji potřebuješ tak brzy, že nemá smysl odkládat její vytvoření na pozdější dobu.

141. Stále nechápu, proč by se měl při použití statické konstanty vytvářet jedináček příliš brzy. Jak jsi to myslel? Java se přece chlubí tím, že zavádí třídy (a při té příležitosti inicializuje jejich statické atributy) až ve chvíli, kdy třídu skutečně potřebuji.

Kdy se instance vytváří zbytečně brzy

To máš sice pravdu, ale platí to pouze v případě, kdy třída nemá statické metody, které někdo zavolá (a tím třídu zavede a inicializuje), aniž by kdokoliv potřeboval pracovat s oním jedináčkem. S takovými situacemi se však v rozumně navržených aplikacích setkáváme spíše výjimečně.

Odložená inicializace

142. No dobře, ale představ si, že se dostanu do takovéto výjimečné situace. Jaký je doporučený postup?

Jak postupovat

Je-li vytvoření jedináčka drahá operace (jsou k němu potřeba některé vnější zdroje, trvá dlouho apod.) a hrozí-li, že třída bude zavedena do paměti i v případě, kdy jedináček vůbec nebude potřeba, můžeš vytvoření a inicializaci jedináčka naprogramovat jako *odloženou* (doslovný překlad termínu *lazy initialization* je líná inicializace). Při ní nechá tovární metoda vytvořit instanci jedináčka teprve poté, co o ni bude poprvé požádána – např.:

```
public static Jedináček getInstance() {
    if( jedináček == null )
        jedináček = new Jedináček();
    return jedináček;
}
```

Implementaci odložené inicializace spolu s jejím testem ukazuje třída `Lenoch` (pojmenovali jsme ji podle anglického názvu pro odloženou inicializaci), jejíž zdrojový kód obsahuje výpis 9.2.

Výpis 9.2: `Lenoch` – jedináček s odloženou inicializací

```
package rup.česky.vzory._09_jedináček;

/*******************************************************************************
 * Třída Lenoch slouží k demonstraci jedináčka s odloženou inicializací
 * v jednovláknové aplikaci.
*/
public final class Lenoch
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Odkaz na jedináčka -
     *  jeho instanci necháme vytvořit, až ji budeme potřebovat. */
    private static Lenoch lenoch;


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /*****************************************************************************
     * Vrátí odkaz na instanci lenocha.
     * Pokud ještě neexistuje, nechá ji vytvořit.
     */
    public static Lenoch getLenoch()
    {
        System.out.print("Žádost o instanci lenocha");
        if( lenoch == null )
            lenoch = new Lenoch();
        System.out.println(" - Vracím odkaz na instanci");
        return lenoch;
    }


    /*****************************************************************************
     * Soukromý konstruktor bránící vytvoření dalších instancí.
     * V této ukázce je definován jako prázdný.
     */
    private Lenoch()
    {
       System.out.println("\n   + Vytvářím instanci lenocha");
    }


//== TESTY =====================================================================

    /*****************************************************************************
     * Metoda vrací informaci o tom, zda již existuje instance lenocha.
     */
    private static boolean existuje()
    {
        return lenoch != null;
    }


    /*****************************************************************************
     * Ověří existenci lenocha před a po žádosti o odkaz na jeho instanci.
```

```
     */
    public static void test()
    {
       System.out.println( "Lenoch existuje: " + existuje() );
       Lenoch lenoch1 = getLenoch();
       System.out.println( "Lenoch existuje: " + existuje() );
       Lenoch lenoch2 = getLenoch();
       System.out.println( "Získaní lenoši jsou " +
          ((lenoch1 == lenoch2) ? "totožní"  :  "rozdílní") );
    }

}
```

Vícevláknové aplikace

143. Máš pravdu – takto lze vytvoření jedináčka opravdu odložit až do chvíle, kdy jej budu doopravdy potřebovat. Proč jsi ale v komentáři třídy psal, že se jedná o jednovláknovou aplikaci?

Takto lze vytvoření jedináčka odložit pouze v případě, kdy o něj nebude žádat více vláken. Pak by ses mohl dostat do problému.

144. Jak?

Proč může vzniknout více instancí

Tak, jak se snaží naznačit tabulka 9.1. První vlákno zavolá metodu `getInstance()`. Ta zjistí, že jedináček ještě nebyl vytvořen, a rozhodne se jej proto vytvořit. Než tak ale učiní, bude vláknu odebrán procesor a druhé vlákno také zavolá metodu `getInstance()`. I ta zjistí, že jedináček ještě nebyl vytvořen. Vytvoří jej tedy a odkaz vrátí volající metodě, která začne instanci používat. Poté ale bude odebrán procesor druhému vláknu a vrácen vláknu prvnímu. To bude pokračovat v rozdělané práci, tj. ve vytvoření instance jedináčka. Vytvoří ji, ale instance již nebude jedináček, protože vznikne jako druhá v pořadí.

Tabulka 9.1: Možný průběh programu se dvěma vlákny žádajícími o jedináčka

První vlákno	Druhé vlákno
`Jedináček.getInstance() {` `  if( jedináček == null )`	

	`Jedináček.getInstance() {`
	`  if( jedináček == null )`
	`  jedináček = new Jedináček();`
	`  return jedináček;`
	`}`

`  jedináček = new Jedináček();`
`  return jedináček;`
`}`	

145. To je jednoduché: definuji tovární metodu jako synchronizovanou.

Problémy synchronizace celé metody

Tím se sice problému s nedodržením jedináčkovosti instance vyhneš, ale za cenu dost velké režie spojené se vstupem do synchronizované sekce. Jinými slovy: nákladům spojeným se zřízením nepotřebné instance se vyhneš za cenu zvýšení nákladů na zís-

kávání odkazů na instanci. Řadě programátorů vadí, že za to, aby při první žádosti nedošlo ke kolizi, musí při všech následných žádostech o jedináčka platit za režii spojenou se vstupem do kritické sekce.

146. No jo. To by mi při častém získávání odkazů na instanci také vadilo. Jak se tomu mohu vyhnout?

Řešení s dvojitou kontrolou

Záleží na tom, jakou verzi Javy používáš. Pracuješ-li v Javě 5.0, můžeš definovat odkaz na jedináčka jako nestálý (`volatile`) a v tovární metodě pro získání odkazu na něj využít dvojitou kontrolu, tj. definovat ji následovně (změny oproti minulému řešení jsem podbarvil):

```
private static volatile Jedináček jedináček = null;

public static Jedináček getJedináček() {
    if( jedináček == null )
        synchronized( Jedináček.class ) {
            if( jedináček == null )
                jedináček = new Jedináček();
        }
    return jedináček;
}
```

Když zjistíš, že atribut `jedináček` ještě není nastaven, otevřeš synchronizovaný blok a v něm se na nenastavenost jedináčka zeptáš ještě jednou. A protože atribut testuješ i nastavuješ v synchronizovaném bloku, máš jistotu, že jsi na to sám a s nikým se o danou informaci nepřetahuješ.

Podmínka: nestálost odkazu

Podmínkou správné funkce však je, aby atribut `jedináček` byl definován jako nestálý (`volatile`). Tím dosáhneš toho, že se optimalizační programy nebudou snažit zefektivnit jeho používání a při každé žádosti o jeho načtení jej opravdu načtou, i když jej náhodou budou mít od minula ještě načtený v registrech.

147. A proč bych kvůli tomu musel pracovat v Javě 5.0?

Problémy verzí před Javou 5.0

Předchozí verze používaly trochu jiný paměťový model a měly práci s nestálými atributy definovanou poměrně komplikovaně a ne zcela průzračně, takže s nimi virtuální stroje nepracovaly vždy tak, jak by si jejich povaha zasluhovala.

V čem byl problém

Základním problémem bylo, že překladač měl v zájmu optimalizace právo měnit pořadí příkazů (např. proto, že měl nějakou hodnotu v registru procesoru, tak aby ji tam nemusel kvůli dalšímu použití načítat za chvíli znovu). Autoři specifikace přitom zapomněli nestálé proměnné z tohoto práva vyjmout.

Řešení v Javě 5

V Javě 5 již toto opomenutí napravili, takže příkazy používající nestálé proměnné jsou nyní jako hradba, přes kterou se nesmějí ostatní příkazy přesouvat.

Tohle ale není kniha o paralelním programování, takže to tu nebudu dále rozebírat. Chceš-li se o dané problematice dozvědět více, podívej se do reference jazyka nebo na článek [43], který problematiku dvojí kontroly při konstrukci jedináčka ve starších verzích Javy podrobně rozebírá.

Doporučené řešení

Stačí-li ti pouhé konstatování skutečnosti, tak si zapamatuj, že při používání starších verzí Javy je v případě, kdy jedináčka používá několik vláken, nejlepší neriskovat a buď synchronizovat celou tovární metodu anebo jedináčka vytvořit a přiřadit odkaz

na něj hned v deklaraci příslušného atributu tak, jak jsme si ukazovali na počátku kapitoly.

Serializovatelnost

148. Budu si to pamatovat. Máš tam ještě další věci na zapamatování?

Pokud možno nedeklarovat jako serializovatelný

Pamatuj si, že jedináček by neměl být jednoduše serializovatelný. Je to totiž zbytečné, protože by se načtená instance stejně musela zahodit, když je již jedna definovaná.

149. Proč bych jej měl někam ukládat, když je to stejně jedináček?

Co může být potřeba uložit

Instance je sice jedináček, ale můžeš si potřebovat zapamatovat třeba jeho okamžitý vnitřní stav, abys mohl po restartu (nebo na jiném počítači) pokračovat přesně v tom místě, kde jsi právě přestal.

150. V tom případě mi musíš prozradit, co si představuješ pod pojmem *jednoduše serializovatelný*.

Jak vrátit místo přečteného objektu jedináčka

To znamená, že serializaci a deserializaci jedináčka nemůžeš nechat na systému, protože ten by ti při jeho načtení vyrobil druhou instanci.

Musíš využít možností, které Java pro takovéto případy nabízí – konkrétně metodu `readResolve()`, která ti umožní vrátit místo načteného objektu dosavadního jedináčka a tvářit se, že je to objekt, který jsi právě přečetl.

151. A kruciš! O tom jsem se ve své učebnici Javy nedočetl. Co to je za metodu a jaký má účel?

Funkce metody `readResolve()`

Pravda, o této možnosti se v příručkách Javy běžně nedočteš a v česky psaných už vůbec ne.[1] Metoda musí být deklarována:

```
Object readResolve() throws ObjectStreamException;
```

Najde-li systém ve třídě takto definovanou metodu, zavolá ji po načtení objektu ze vstupního proudu a návratovou hodnotu metody vrátí jako načtený objekt. Tak ti systém umožní vrátit místo odkazu na skutečně načtený objekt odkaz na úplně jiný objekt – v našem případě odkaz na dosavadního jedináčka.

152. To je dobré! Takže stačí, aby v ní byl jediný příkaz, který vrací odkaz na jedináčka?

Teoreticky ano. Pokud ale budeš chtít upravit stav jedináčka podle právě přečtených informací, můžeš přidat i další příkazy.

Dál už se tu ale o tomto problému rozpovídávat nebudu. Jestli se budeš chtít dozvědět o možnostech serializace a deserializace více, přečti si dokumentaci k rozhraní `Serializable`. Problémem je, že tato dokumentace je psána anglicky. V česky psaných příručkách (alespoň v těch, které se mi dostaly do ruky) se toho o specialitách serializace moc nedočteš (viz poslední poznámku pod čarou).

---

[1] Nepíše o ní dokonce ani [33], která se vydává za knihu pro profesionály. Jediná česky vydaná kniha, kde se o této metodě lze něco dozvědět, je [28].

Speciální případy

153. Máš tam pro mne ještě nějaké další špeky?

Něco by se tu našlo. První se týká vlastních zavaděčů tříd (*class loader*), druhý se týká prvních verzí Javy, tj. verzí předcházejících verzi 1.2. Kromě toho ti ještě dlužím vysvětlení, jak je to s jedináčky, kteří mají potomky.

Vlastní zavaděče tříd

154. Vlastní zavaděče tříd zatím nepoužívám, ale kdyby, tak co?

Možnost dvojího zavedení třídy

Kdybys je používal, musíš myslet na to, že jednotlivé zavaděče spolu moc nemluví, a mohlo by se tedy stát, že by dva různé zavaděče zavedly každý svoji vlastní verzi třídy jedináčka, takže by pak v programu opět vystupovali dva různí jedináčci.

155. Samotná představa, že bych mohl mít v systému jednu třídu dvakrát, mne děsí. Kde se něco takového používá?

Příklad použití několika zavaděčů

Např. na aplikačních serverech platformy J2EE[1]. Ty používají několik zavaděčů tříd proto, aby mohly odstranit z paměti třídy, o nichž vědí, že už je nebudou potřebovat. Odstraněním zavaděče tříd totiž odstraní i všechny třídy, které zavedl.

Tyto aplikační servery navíc používají oddělené zavaděče tříd i z bezpečnostních důvodů.

Jinými slovy: dokud nemáš zabezpečeno, že je daný jedináček sdílen prostřednictvím jediného zavaděče tříd (např. systémového), nemáš zaručeno, že je to jedináček.

Chyba v prvních verzích Javy

156. Tak mne ještě doraž s těmi prvními verzemi Javy.

Kdy nastávala

První verze Javy měly chybu ve správci paměti (*garbage collector*), která způsobovala, že správce paměti zařazoval mezi kandidáty na odstranění i takové objekty, na které se sice někdo odkazoval, avšak ten někdo byl atributem třídy daného objektu.

Pokud bys tedy podle předchozího doporučení inicializoval statický atribut s odkazem na jedináčka hned v jeho deklaraci, mohlo by se stát, že by příslušný jedináček byl odstraněn dříve, než by o něj prostřednictvím tovární metody někdo požádal.

Jedináček s dědici

157. Naštěstí mi psaní programu pro starší verze nehrozí. Takže už zbývá pouze probrání toho jedináčka, který může mít potomky.

Á pravda. Na toho jsem málem zapomněl. Jak jsem říkal, jsou situace, kdy nemůžeš dopředu odhadnout, jakého z jedináčků, kteří jsou k dispozici, budeš potřebovat.

---

[1] J2EE – *Java 2 Enterprise Edition* je platforma Javy používaná pro distribuované aplikace, tj. pro aplikace, jejichž různé části mohou běžet na různých počítačích (tento požadavek je u rozsáhlých aplikací běžný). S příchodem nové verze této platformy byla v roce 2006 platforma přejmenována na *Java EE* s následným číslem verze – v době psaní knihy *Java EE 5*.

Bývá to většinou tak, že jedináček, na nějž se obrací program, vydává za svoji instanci některého ze svých potomků.

Poskytovatelem jedináčka nemusí být jeho vlastní třída

Třídu, která bude vůči okolnímu světu vystupovat jako poskytovatel (a rodič) jedináčka, bývá optimální definovat jako abstraktní třídu, která bude tím společným rodičem budoucích jedináčků, a třídy jednotlivých jedináčků pak definovat jako její potomky.

158. Musím se přiznat, že jedináček s potomky mi připadá jako protimluv, protože třída se soukromým konstruktorem žádné potomky mít nemůže.

Jak se dělit o konstruktor

Nemáš tak docela pravdu. Pokud je potomek definován jako zanořená třída, bude vidět i na soukromý konstruktor svého rodiče.

U větších projektů jsou však potomci většinou definováni v samostatných třídách a v takovém případě musíš soukromý konstruktor oželet a nahradit jej chráněným. V důsledku toho se pak dodržení jedináčkovství prosazuje poněkud obtížněji, ale jde to.

159. Princip je mi jasný, ale pořád nějak nechápu, k čemu by to mohlo být dobré. Nemohl bys uvést nějaký příklad?

Příklad

GoF předvádí takovéhoto jedináčka na příkladu bludiště, které je základem hry. Celá hra se odehrává v jediném bludišti, ale ty si na počátku můžeš vybrat druh bludiště, kterým pak budeš procházet. Možnou podobu takovéto třídy si můžeš prohlédnout ve výpisu 9.3. V tomto příkladu jsou pro jednoduchost definována jednotlivá bludiště jako vnořené třídy, ale jistě by sis je dovedl představit definované samostatně.

Výpis 9.3: `Bludiště_GoF` – definice bludiště inspirovaného GoF

```
package rup.česky.vzory._09_jedináček;

/*******************************************************************************
 * Instance třídy Bludiště_GoF představují bludiště obdobné tomu,
 * o němž se hovoří v GoF v kapitole o návrhovém vzoru <i>Jedináček</i>.
*/

public abstract class Bludiště_GoF
{
//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================================

    /** Odkaz na instanci jedináčka. */
    private static Bludiště_GoF jedináček;

//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Tovární metoda vracející odkaz na jedináčka,
     * jehož typ je zadán prostřednictvím systémové proměnné <b>BLUDISTE</b>.
     * @return Odkaz na jedináčka
     */
    public static Bludiště_GoF getInstance()
    {
        if( jedináček == null ) {
            synchronized( Bludiště_GoF.class ) {
                if( jedináček == null ) {
                    String název = System.getenv( "BLUDISTE" );
                    if( název == null )
```

```
                    název = "";
                jedináček = vyberBludiště( název );
            }
        }
    }
    return jedináček;
}

/**************************************************************************
 * Konstruktor je tu definován pouze proto, aby jej bylo možno nastavit
 * jako soukromý.
 */
private Bludiště_GoF() {}


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ===========================================

/**************************************************************************
 * Soukromá třída vracející odkaz na instanci třídy zadané
 * prostřednictvím textu předaného jako parametr.
 * @param název Identifikace požadovaného bludiště
 * @return Odkaz na požadované bludiště
 */
private static Bludiště_GoF vyberBludiště( String název )
{
    if( název.equalsIgnoreCase( "BOMBOVE" ) )
        return new BombovéBludiště();
    else if( název.equalsIgnoreCase( "MAGICKE" ) )
        return new MagickéBludiště();
    else if( název.equalsIgnoreCase( "VESELE" ) )
        return new VeseléBludiště();
    else
        return new ImplicitníBludiště();
}


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

private static class BombovéBludiště    extends Bludiště_GoF {}
private static class MagickéBludiště    extends Bludiště_GoF {}
private static class VeseléBludiště     extends Bludiště_GoF {}
private static class ImplicitníBludiště extends Bludiště_GoF {}


//== TESTY A METODA MAIN =======================================================

/**************************************************************************
 * Testovací metoda.
 */
public static void test()
{
    Bludiště_GoF bg = getInstance();
    System.out.println( bg );
}
/** @param ppr Parametry příkazového řádku - nepoužité */
public static void main(String[]ppr){ test(); }/*-*/
}
```

160. No a teď ještě bludiště, kde potomci nebudou definováni jako zanořené třídy.

Definice bez zanořených tříd

Definici takového bludiště si snad domyslíš. Konstruktor společného abstraktního rodiče bude definován jako chráněný a třídy budou v samostatných souborech.

Zajímavé by bylo, kdybys chtěl, aby tovární metoda neznala dopředu typy svých potomků a musela si nejprve požadovaný typ zjistit např. dotazem uživateli, kterému by dala na výběr z nějakého seznamu tříd, jejichž class-soubory našla v předem známé složce a jejichž názvy vyhovují nějakým konvencím – např. začínají slovem Bludiště. Tovární metoda by pak vytvořila příslušnou instanci s využitím reflexe.

161. Chceš tím říct, že bych mohl mít hru definovanou tak, abych si mohl třídy bludiště stahovat např. po Internetu a vždy je jenom přidat na zadané místo?

Něco takového. Já ostatně provádím něco obdobného se svými studenty, kteří mají jako jednu ze svých semestrálních prací vytvořit obdobnou hru, přičemž ono abstraktní bludiště, jehož budou jejich bludiště potomky, dostanou ode mne. Já se k tomuto projektu ještě vrátím u některého z dalších vzorů.

Shrnutí – co jsme se naučili

- Návrhový vzor *Jedináček* je prvním z probíraných vzorů, který je popsán v GoF; jeho anglický název je *Singleton*.
- Návrhový vzor *Jedináček* zabezpečuje, že třída nebude mít více než jednu instanci.
- Standardní implementace využívá soukromý konstruktor, soukromý statický atribut odkazující na tuto instanci, a veřejnou jednoduchou tovární metodu, která vrací uchovávaný odkaz.
- Konstruktor lze zavolat přímo v deklaraci statického atributu, který je pak možno deklarovat jako konstantu.
- V některých případech je výhodné definovat tento konstantní atribut jako veřejný a vyhnout se tak nutnosti používání tovární metody. Obecně se však takovéto řešení příliš nedoporučuje.
- Při použití standardní implementace si necháváme otevřený prostor ke změně názoru na počet povolených instancí.
- Je-li vytvoření instance náročnější, může tovární metoda iniciovat její vytvoření až při první žádosti o odkaz na instanci. Tím zabezpečí, že se instance vytvoří pouze v případě, kdy ji bude doopravdy někdo potřebovat. Hovoříme pak o tzv. *odložené inicializaci (lazy initialization)*.
- Ve vícevláknových aplikacích je třeba u odložené inicializace ošetřit, aby inicializaci jedináčka nemohla souběžně vyvolat dvě různá vlákna. Jednou z možností je využít dvojí kontroly hodnoty odkazu.
- Popsaná dvojí kontrola funguje bezpečně pouze od verze 5.0. U starších verzí je výhodnější použít inicializaci přímo v deklaraci statické proměnné nebo synchronizovat celou tovární metodu.
- Třída jedináčka by neměla být serializovatelná. Je-li její serializovatelnost žádoucí, je třeba zabezpečit, aby metody načítající instanci ze vstupního prou-

du vracely odkaz na existujícího jedináčka. Toho lze dosáhnout použitím metody `readResolve()`.

- Při používání několika zavaděčů tříd je třeba ošetřit, aby každý zavaděč nezavedl svoji verzi jedináčkovské třídy a nevytvořil tak i vlastní instanci jedináčka.
- Při používání starších verzí Javy než 1.2 je třeba počítat s chybou ve správci paměti, který odstraní instanci i v situaci, když se na ni odkazuje pouze atribut její vlastní třídy.
- Třída jedináčka může mít řadu potomků a tovární metoda vracející odkaz na jedináčka se může při jeho inicializaci na poslední chvíli rozhodnout, která třída toho správného jedináčka dodá.
- Nejsou-li potomci definováni jako zanořené třídy svého rodiče, je třeba oželet zjednodušující požadavek na soukromost konstruktoru společného rodiče.

KAPITOLA 10

Předem známé instance (Výčtový typ – Enumerated Type)

- **Účel**
- **Implementace**
- **Funkční výčtové typy**
- **Výčtové podtypy**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Výčtové typy slouží k definici skupin předem známých hodnot a k umožnění následné typové kontroly. Vedle klasických hodnotových výčtových typů umožňuje Java definovat i *Funkční výčtové typy*, u nichž se jednotlivé hodnoty liší reakcemi na zasílané zprávy.

162. Tak co teď? Knihovní třída neměla žádnou instanci, jedináček měl jedinou instanci, tak čekám, že mi ukážeš, jak definovat třídu, která bude mít dvě instance.

Dva druhy tříd se známým počtem instancí

Nejsi tak daleko od pravdy. Neomezíme se ale pouze na dvě instance, ale ukážeme si, jak definovat třídu, která bude mít malý, předem známý počet instancí. Takovéto třídy můžeme rozdělit do dvou skupin:

– Výčtový typ

- Jsou-li předem známé i jednotlivé instance této třídy, pak můžeme danou třídu chápat jako výčtový typ. Takovým třídám bude věnována tato kapitola.

– Fond

- Je-li sice znám počet instancí, ale není-li důležité, se kterou z nich budeme pracovat, jedná se o fond (*pool*), o němž si budeme povídat v kapitole *Konečný počet instancí (Fond – Pool)* na straně 151.

Účel

163. Výčtové typy znám i z neobjektových jazyků – např. z Pascalu.

Pascal: obálky pro skupiny celočíselných hodnot

V Pascalu byly výčtové typy zavedeny jako náhrada číselných konstant. Jejich zavedení umožnilo typovou kontrolu, takže se dalo např. jednoduše zkontrolovat, kdy jednička znamená Pondělí, kdy Leden a kdy např. Sever. Obdobně mají výčtové typy definovány i některé z objektových jazyků – např. C++, C#, Delphi, Visual Basic .NET a další.

Java: objektové výčtové typy

Java však ve verzi 5.0 zavedla výčtové typy jako plnohodnotné objektové typy se všemi z toho vyplývajícími vlastnostmi. Z toho plyne řada výhod a i některé nevýhody.

164. To nejlepší si nechám na konec, takže začni těmi nevýhodami.

– nevýhoda: efektivita

Základní a asi jedinou nevýhodou této definice je efektivita. S hodnotami výčtových typů, které jsou pouze formální obálkou nad celými čísly (interně se jedná o celá čísla), pracuje program přece jenom o poznání efektivněji než s hodnotami, které jsou instancemi objektových typů.

165. A ty výhody?

– výhoda: plnohodnotné objekty

Těch je celá řada a vyplývají z objektové podstaty oněch hodnot. Hodnotám objektových typů můžeš definovat schopnost reagovat na různé zprávy a nabízet např. knihovnu metod souvisejících s daným typem.

Java 5.0, která výčtové typy zahrnula do své syntaxe, přidává další. Výčtové typy jsou v ní potomky třídy `java.lang.Enum`, od níž dědí řadu užitečných metod, a dalšími je skrytě vybaví překladač (podrobnosti jsem popsal ve [31]).

166. Tak teď ti moc nerozumím. Mohl bys mi dát nějaký konkrétní příklad?

Příklad výhod

Dejme tomu, že zavedeme výčtový typ `Směr`, který bude mít čtyři instance: `VÝCHOD`, `SEVER`, `ZÁPAD` a `JIH`. Ty instance mohou nejenom představovat svoji hodnotu, tj. příslušný směr, ale mohou umět i další věci:

- Mohou mi prozradit, do kterého směru budu natočen poté, co při natočení do jejich směru udělám vlevo v bok nebo čelem vzad;

- mohou mi prozradit, jak se bude měnit vodorovná či svislá souřadnice při pohybu daným směrem;
- když jim dodám souřadnice políčka, mohou mi prozradit souřadnice sousedního políčka ve svém směru.

167. Začínám to chápat – instance výčtových typů zastupují danou hodnotu jako u číselných výčtových typů, ale navíc jsou také objektem.

Lépe bych to neřekl. Jejich implementace v Javě 5.0 a v předchozích verzích Javy se ale liší.

Implementace

Starší verze Javy

168. Kterou implementací začneš?

Kde najít podrobný návod

Začnu implementací ve starších verzích, protože z ní implementace v Javě 5.0 vychází. Základní požadavky na vlastnosti komplexního výčtového typu a zásady jeho implementace jsou velmi dobře popsány v publikaci [41], která byla přeložena do češtiny ([28]). Tam se také obrať, budeš-li se chtít o možnostech implementace ve starších verzích dozvědět větší podrobnosti.

169. Tak ukazuj, vždyť já se v tom nějak vyznám.

Možnou implementaci datového typu `Směr`, který jsem dával před chvílí za příklad, si můžeš prohlédnout ve výpisu 10.1. Abych od sebe odlišil implementace pro starou a novou Javu, nazval jsem tuto třídu `Směr_14`[1] (je ji možno naprogramovat v Javě 1.4).

Ukázka implementace typu Směr

Výpis 10.1: `Směr_14` – ukázka implementace výčtového typu ve starších verzích Javy

```
package rup.česky.vzory._10_výčet;

import java.awt.Point;

 /*****************************************************************************
 * Třída Směr_14 demonstruje možnosti implementace výčtového typu
 * ve starších verzích Javy, tj. do verze 1.4 včetně.
 */
public class Směr_14
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Pole jednotlivých instancí. */
    private static final Směr_14[] SMĚR = new Směr_14[4];
    private static int SMĚRŮ = 0;
```

[1] Jazykoví puristi budou asi nyní namítat, že do názvů tříd znaky podtržení nepatří. Když ale tento znak nepoužiji, plete se jednička často s malým L. Dávám proto přednost vyšší přehlednosti.

```
    //Parametry jsou přírůstky souřadnic při pohybu v daném směru a název inst.
    public static final Směr_14
        VÝCHOD = new Směr_14( 1, 0, "VÝCHOD"),
        SEVER  = new Směr_14( 0,-1, "SEVER" ),
        ZÁPAD  = new Směr_14(-1, 0, "ZÁPAD" ),
        JIH    = new Směr_14( 0, 1, "JIH"   );


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final int dx, dy;     //Přírůstky souřadnic při pohybu v daném směru
    private final int pořadí;     //Pořadí dané instance mezi ostatními
    private final String název;   //Název instance


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Nemá-li mít nikdo možnost ovlivňovat počet instancí,
     * musí být konstruktor soukromý.
     */
    private Směr_14( int dx, int dy, String název )
    {
        this.dx    = dx;
        this.dy    = dy;
        this.název = název;
        pořadí = SMĚRŮ++;
        SMĚR[ pořadí ] = this;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Výčtová hodnota se představuje svým názvem.
     */
    public String toString()
    {
        return název;
    }


    /*****************************************************************************
     * Vrátí přírůstek vodorovné souřadnice při pohybu v daném směru.
     */
    public int dx()
    {
        return dx;
    }


    /*****************************************************************************
     * Vrátí přírůstek svislé souřadnice při pohybu v daném směru.
     */
    public int dy()
    {
```

```
        return dy;
    }


    /*****************************************************************************
     * Vrátí směr natočení po otočení o 90° vlevo.
     */
    public Směr_14 vlevoVbok()
    {
        return SMĚR[ (pořadí+1) % SMĚRŮ ];
    }


    /*****************************************************************************
     * Vrátí směr natočení po otočení o 180°.
     */
    public Směr_14 čelemVzad()
    {
        return SMĚR[ (pořadí+2) % SMĚRŮ ];
    }


    /*****************************************************************************
     * Vrátí souřadnice souseda zadané pozice ve svém směru.
     */
    public Point soused( Point pozice )
    {
        return new Point( pozice.x + dx, pozice.y + dy );
    }


//== TESTY ======================================================================

    /*****************************************************************************
     * Test operací s hodnotou výčtového typu.
     */
    public static void test()
    {
        Směr_14 s = VÝCHOD;
        System.out.println("Jsem otočen na " + s);
        System.out.println("Po otočení vlevo budu otočen na " +
            (s = s.vlevoVbok()) );
        System.out.println("Po následném otočení o 180° budu otočen na " +
            (s = s.čelemVzad()) );
        Point p1 = new Point( 5, 5 );
        Point p2 = s.soused( p1 );
        System.out.println("Sousedem pole " + p1 + " ve směru na " + s +
                           "\n      je pole " + p2 );
    }/** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {  test();
}
```

Hodnoty jsou statické atributy, konstruktor soukromý

170. Jestli jsem to dobře pochopil, tak vtip je v tom, že hodnoty výčtového typu jsou veřejnými statickými atributy dané třídy.

Pochopil jsi to dobře. Ještě ale musíš dodat, že konstruktor je soukromý, aby nikdo nemohl přidělávat další instance. Jinak se ale jedná o plnohodnotné instance.

171. Je to vlastně takové rozšíření jedináčka na více instancí. Používá ale veřejné statické atributy, které jsi u jedináčka pomlouval.

Rozdíly oproti jedináčkovi

Jedním z možných účelů standardní implementace jedináčka je umožnit svobodu při rozhodování o skutečné třídě oné jediné instance. Jak jsem již řekl, při každém spuštění programu může mít instance jedináčka jiný vlastní typ. U výčtového typu se naproti tomu předpokládá, že typ instancí je předem daný a pokaždé stejný. Hlavním účelem výčtového typu je totiž prezentovat definovanou množinu hodnot.

Java 5.0

172. Jak tuto definici pátá Java zjednoduší?

Nejjednodušší definice

Java zavádí pro výčtové typy zvláštní syntaktickou definici. Místo klíčového slova `class` použiješ klíčové slovo `enum` a na počátku definice uvedeš seznam jednotlivých hodnot daného výčtového typu. Pokud bys nepotřeboval, aby hodnoty výčtového typu měly nějaké další objektové schopnosti, vystačíš s velice jednoduchou definicí:

```
public enum Směr { VÝCHOD, SEVER, ZÁPAD, JIH }
```

173. To je tedy hodně jednoduchá definice. Docela se podobá té pascalské.

Definice komplexnějšího výčtového typu

Jak jsem řekl, toto je nejjednodušší možná definice, po níž sáhneš ve chvíli, kdy nepotřebuješ nic jiného než nějak inteligentně reprezentovat množinu možných hodnot jednoho typu. I takto jednoduše definovaný výčtový typ už leccos umí. Nebudu tu ale rozebírat podrobnosti – pokud je ještě neznáš, nahlédni do [31].

Pojď se raději podívat na ekvivalent onoho dokonalejšího směru, který jsme před chvílí definovali ve třídě `Směr_14`. Kdybys chtěl definovat výčtový typ s podobnými schopnostmi (tj. s instancemi, které mají další užitečné metody), můžeš do závorek za jednotlivé názvy hodnot uvést parametry konstruktoru a za výčtem hodnot definovat v těle konstruktor (samozřejmě soukromý) a další požadované metody.

Definici třídy realizující obdobný typ prostředky Javy 5.0 si můžeš prohlédnout ve výpisu 10.2. Všimni si v něm, že oproti definici třídy `Směr_14` v něm odpadla nutnost pamatovat si pořadí a název instancí (to zajišťuje překladač), ale na druhou stranu se zkomplikovala inicializace statických atributů, protože tělo třídy musí začínat výčtem hodnot daného typu.

174. Některé věci v tom programu jsou mi nejasné.

Kde sehnat informace o implementaci v Javě 5.0

Nechtěl bych se tu rozepisovat o syntaxi a vlastnostech výčtových typů. O těch jsem se již dostatečně podrobně rozepsal v [31]. Nechce-li se ti kupovat kvůli několika informacím hned celou knížku, můžeš si přečíst článek [42], v němž jsem se o vlastnostech výčtových typů rozepsal (a který se stal i základem příslušné kapitoly v knize).

Ukázka implementace typu Směr

Výpis 10.2: `Směr_50` – ukázka implementace výčtového typu v Javě 5.0

```
package rup.česky.vzory._10_výčet;

import java.awt.Point;
```

```
/*****************************************************************************
 * Třída Směr_5O demonstruje možnosti implementace výčtového typu v Javě 5.0.
 */
public enum Směr_5O
{
//== HODNOTY VÝČTOVÉHO TYPU ================================================

    // Parametry jsou přírůstky souřadnic při pohybu v daném směru
    VÝCHOD(1,0), SEVER(0,-1), ZÁPAD(-1,0), JIH(0,1);


//== KONSTANTNÍ ATRIBUTY TŘÍDY =============================================

    /** Pole jednotlivých instancí.
     *  Toto pole není nutné (lze je v pohodě nahradit polem values());
     *  uvádím jej zde pouze proto, abyste viděli, jak je třeba inicializovat
     *  případné statické členy - definovat je až za hodnotami výčtového typu
     *  a není-li je možno inicializovat v deklaraci,
     *  inicializovat je pomocí statického konstruktoru.
     */
    private static final Směr_5O[] SMĚR;
    static {
        SMĚR = new Směr_5O[] { VÝCHOD, SEVER, ZÁPAD, JIH };
    }

    private static final int SMĚRŮ = SMĚR.length;


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==========================================

    private final int dx, dy;     //Přírůstky souřadnic při pohybu v daném směru


//== KONSTRUKTORY A TOVÁRNÍ METODY =========================================

    /***************************************************************************
     * Nemá-li mít nikdo možnost ovlivňovat počet instancí,
     *musí být konstruktor soukromý.
     */
    private Směr_5O( int dx, int dy )
    {
        this.dx = dx;
        this.dy = dy;
    }


//== NESOUKROMÉ METODY INSTANCÍ ============================================

    /***************************************************************************
     * Vrátí přírůstek vodorovné souřadnice při pohybu v daném směru.
     */
    public int dx()
    {
        return dx;
    }
```

```
    /*****************************************************************************
     * Vrátí přírůstek svislé souřadnice při pohybu v daném směru.
     */
    public int dy()
    {
        return dy;
    }


    /*****************************************************************************
     * Vrátí směr natočení po otočení o 90° vlevo.
     */
    public Směr_5O vlevoVbok()
    {
        return SMĚR[ (ordinal()+1) % SMĚRŮ ];
    }


    /*****************************************************************************
     * Vrátí směr natočení po otočení o 180°.
     */
    public Směr_5O čelemVzad()
    {
        return SMĚR[ (ordinal()+2) % SMĚRŮ ];
    }


    /*****************************************************************************
     * Vrátí souřadnice souseda zadané pozice ve svém směru.
     */
    public Point soused( Point pozice )
    {
        return new Point( pozice.x + dx, pozice.y + dy );
    }


//== TESTY ====================================================================

    /*****************************************************************************
     * Test operací s hodnotou výčtového typu.
     */
    public static void test()
    {
        Směr_5O s = VÝCHOD;
        System.out.println("Jsem otočen na " + s);
        System.out.println("Po otočení vlevo budu otočen na " +
            (s = s.vlevoVbok()) );
        System.out.println("Po následném otočení o 180° budu otočen na " +
            (s = s.čelemVzad()) );
        Point p1 = new Point( 5, 5 );
        Point p2 = s.soused( p1 );
        System.out.println("Sousedem pole " + p1 + " ve směru na " + s +
                           "\n      je pole " + p2 );
    }

}
```

Funkční výčtové typy

175. Někde jsem slyšel, že různé instance výčtového typu mohou mít různé metody. To by pak ale nemohly být všechny stejného typu.

Charakteristika

Ano a ne. Chtěl bych ale hlavně upozornit, že se nejedná o různé sady metod, ale o možnost definovat pro každou instanci výčtového typu různá těla jejích metod. Využívá se k tomu anonymních vnitřních tříd.

Je pravda, že pak má každá instance jiný vlastní typ (mají pouze společného předka – svůj výčtový typ), ale typ instancí je při každém spuštění programu stejný.

176. Mohl bys mi to předvést na nějakém příkladu?

Příklad: Člověk

Samozřejmě. Ukážeme si to na alternativní variantě staršího příkladu. Když jsme probírali jednoduchou tovární metodu, definovali jsme ve výpisu 3.1 na straně 61 třídu `Člověk`, jejíž tovární metoda `getČlověk()` vracela instance různých typů. Nyní si definujeme obdobnou třídu, ale jako funkční výčtový typ. Zdrojový kód si můžeš prohlédnout ve výpisu 10.3.

Zdrojový kód ukazuje implementaci v Javě 5.0. Všimni si, že za každým názvem hodnoty výčtového typu je složená závorka otevírající tělo vnitřní třídy, jež je potomkem daného výčtového typu. V těle pak jsou definovány požadované metody.

Ekvivalentní definici ve starších verzích Javy tu předvádět nebudu, ale kdybys ji chtěl vidět, najdeš ji mezi doprovodnými programy v balíčku této kapitoly ve třídě `ČlověkE14`.

Výpis 10.3: Třída `ČlověkE50` definovaná jako výčtový typ

```
package rup.česky.vzory._10_výčet;

/*******************************************************************************
 * Třída ČlověkE50 demonstruje funkční výčtový typ implementovaný v Javě 5.0.
 */
public enum ČlověkE50
{
//== HODNOTY VÝČTOVÉHO TYPU ====================================================

    LENOCH
    {
        public void budíček() { System.out.println("Pomalu vstávám" ); }
        public void práce()   { System.out.println("Líně pracuji"   ); }
        public void volno()   { System.out.println("Odcházím spát"  ); }
        public void spánek()  { System.out.println("Stále spím"     ); }
    },

    ČILOUŠ
    {
        public void budíček() { System.out.println("Rychle vstávám"    ); }
        public void práce()   { System.out.println("Čile pracuji"      ); }
        public void volno()   { System.out.println("Aktivně odpočívám" ); }
        public void spánek()  { System.out.println("Omdlím a spím"     ); }
    },

    PRACANT
    {
```

```
        public void budíček() { System.out.println("Brzy vstávám"     );  }
        public void práce()   { System.out.println("Zaníceně pracuji" ); }
        public void volno()   { System.out.println("Stále pracuji"    ); }
        public void spánek()  { System.out.println("Usínám nad prací" ); }
    };


//== ABSTRAKTNÍ METODY ============================================================

    abstract public void budíček();
    abstract public void práce();
    abstract public void volno();
    abstract public void spánek();


//== TESTY ========================================================================

    /*********************************************************************************
     * Několikrát si řekne o nového člověka a nechá jej prožít pracovní den.
     */
    public static void test()
    {
        den( LENOCH  );
        den( ČILOUŠ  );
        den( PRACANT );
    }

    private static void den( Člověk č )
    {
        System.out.println("\nPracovní den instance " + č);
        č.budíček();
        č.práce();
        č.volno();
        č.spánek();
    }

}
```

Výčtové podtypy

177. V Pascalu jsem mohl definovat výčtový typ a potom jeho podtyp, např. `DnyVTýdnu` a `PracovníDny`. Mohu si něco podobného dovolit také v Javě?

V Javě nelze pro výčtové podtypy používat `enum`

Můžeš, ale nesmíš k tomu chtít využít výčtové typy zavedené ve verzi 5.0. Jak jsem již řekl, výčtové typy Javy totiž nemohou mít potomky. Taková definice ale vyžaduje pečlivé uspořádání jednotlivých komponent, protože jinak budeš dostávat zajímavé výsledky.

178. To by mne zajímalo, co je na takové definici tak zvláštního.

Základ všech potíží spočívá v tom, že podtyp musí být potomkem původního typu, avšak jeho instance musí být mezi výčtem instancí v rodiči. To znamená, že většina inicializací třídy (spíš všechny) musí být definována v rodičovské třídě.

179. Nechápu. Zkus to říct ještě jednou a nějak jednodušeji.

Zákonitosti hierarchie typů a podtypů

Podtyp charakterizující podmnožinu původního výčtu musí být podtypem tohoto výčtu, protože jeho instance jsou speciálními případy instancí svého rodiče – např. všední den je speciálním případem dne v týdnu.

Rodičovský typ ale musí definovat úplný výčet, tzn. že mezi jeho veřejnými statickými konstantami musí být i konstanty jeho podtypu. A teď to opatrně sleduj:

- Abych mohl vytvořit konstantu podtypu, musím zavést a inicializovat její třídu.
- Třída podtypu potřebuje nejprve zavést a inicializovat své rodiče.
- Rodičovská třída však má mezi svými statickými atributy inicializované konstanty podtypu, takže potřebuje vyvolat jejich konstruktor.
- Konstruktor podtypu se proto musí spustit ještě před tím, než je inicializována jeho třída.

Jinými slovy: má-li konstruktor podtypu používat nějaké statické atributy své třídy, je třeba tyto atributy vystěhovat do rodičovské třídy a tam je inicializovat před deklarací inicializované konstanty podtypu, tj. před voláním jejího konstruktoru.

180. Makačka na bednu. Vidím, že se o to raději pokoušet nebudu.

Kde najít ukázku

No kdyby ses o to chtěl přece jenom někdy pokusit, můžeš se inspirovat příkladem tříd definujících směry: třída `Směr` definuje 360 směrů definovaných jednotlivými úhly, třída `Směr8`, která je jejím potomkem, definuje 8 hlavních a vedlejších směrů a třída `Směr4`, která je potomkem třídy `Směr8`, definuje čtyři směry určené hlavními světovými stranami. Všechny tři i s některými pomocnými třídami najdeš v balíčku `rup.česky.vzory._10_výčet.směry`.

181. Tak mne napadá: v Pascalu jsem mohl definovat u výčtového typu libovolný počet jeho podtypů, přičemž jejich množiny se mohly překrývat. To asi v Javě nepůjde, co?

Ne, to nepůjde. Budeš-li chtít definovat výčty opravdu jako datové typy umožňující typovou kontrolu, musí být podtyp potomkem nebo sourozencem jiného typu. Nemohou proto mít některé hodnoty společné a jiné ne.

Shrnutí – co jsme se naučili

- Výčtové typy se používají od sedmdesátých let (zavedl je jazyk Pascal), ale většinou pouze jako typová obálka nad číselnými konstantami.
- V Javě se doporučuje definovat výčtové typy jako standardní objektové typy, které jsou rozšířením jedináčka, oproti němuž zavádějí předem známý počet předem známých instancí.
- Oproti jedináčkovi je u výčtových typů známý i typ jejich instancí. Tento typ je při každém běhu zaručeně stejný.
- Konstruktor výčtového typu je definován jako soukromý, avšak vlastní hodnoty jsou veřejné statické konstanty daného typu.

- Ve starších verzích Javy bylo třeba výčtové typy definovat obdobně jako normální třídy, Java 5.0 pro ně zavedla zvláštní konstrukci.
- Nepotřebujeme-li po výčtovém typu nic jiného než definici sady hodnot, můžeme v Javě 5.0 využít nejjednodušší definice, v níž pouze vyjmenujeme příslušné hodnoty.
- Požadujeme-li „chytré hodnoty“, můžeme doplnit definovaným hodnotám parametry pro jejich konstruktory a společnou sadu požadovaných metod.
- V případě potřeby můžeme definovat i funkční výčtové typy, v nichž může být každá hodnota výčtového typu definována jako instance anonymní vnitřní třídy a může tak mít definovaná vlastní těla metod.
- Potřebujeme-li definovat výčtový typ s podtypy, nemůžeme používat výčtové typy zavedené ve verzi 5.0, ale musíme se uchýlit k ručnímu naprogramování.
- Naprogramování výčtových typů a podtypů je náročné na správnou definici inicializace příslušných tříd.
- *Výčtový typ* nepatří mezi vzory z GoF.

KAPITOLA 11

Dvojníky nepotřebujeme (Originál – Original)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Návrhový vzor *Originál* použijeme v situaci, kdy budeme dopředu vědět, že se v aplikaci bude používat pouze malý počet různých instancí, avšak tyto instance budou požadovány na řadě míst kódu.

Účel

182. Na počátku předchozí kapitoly jsi říkal, že na případy, ve kterých se nejedná o předem známé instance, se aplikuje návrhový vzor *Fond*.

Jednodušší fond

To jsem sice říkal, ale jeho výklad bych si ponechal až do příští kapitoly. Teď bych se rád věnoval něčemu jednoduššímu, a to návrhovému vzoru *Originál*. To je vlastně jednodušší podoba fondu, která nevyžaduje vracení použitých instancí zpět do fondu.

183. Dobře, odložíme vracení použitého zboží a zůstaneme u toho čerstvého. O co tedy v návrhovém vzoru *Originál* jde?

Princip vzoru

V řadě situací nechceš připustit, aby vzniklo příliš mnoho instancí některé třídy, protože víš, že počet různých navzájem neslučitelných instancí tak velký nebude, a spíš očekáváš, že v řadě situací vystačíš s nějakým společným objektem.

Třída proto neustále hlídá žádosti o nové instance a kdykoliv někdo požádá o instanci, která je již vytvořena, nevytváří již její kopii, ale vrátí mu odkaz na dříve vytvořený originál. Odtud také dostal tento návrhový vzor své jméno.

184. Jako obyčejně po tobě budu chtít nějaký příklad.

Částečný příklad: třída `Integer`

V kapitole *Co konstruktor neumí (Jednoduchá tovární metoda – Simple Factory Method)* jsem se v odpovědi na otázku číslo 60 na straně 59 zmiňoval o třídě `Integer`, která má instance pro hodnoty `-128` až `127` předem připravené. Potřebuješ-li pracovat s některou z nich, nevytváří se nová instance (pokud o ni nepožádáš přímo konstruktorem), ale dostaneš odkaz na některou z předem připravených instancí.

Třída `Integer` je na půl cestě k implementaci tohoto návrhového vzoru. Má pro tebe připraveno jenom několik instancí a pro instance s větší absolutní hodnotou s žádným omezením jejich počtu nepočítá.

Úplný příklad: třída `Barva`

Lepším příkladem je třída `Barva`, kterou najdeš v doprovodných programech v balíčku `rup.česky.tvary` a kterou v této knize používám v řadě příkladů. Kdykoliv potřebuješ pracovat s nějakou barvou, požádáš tovární metodu této třídy a dostaneš instanci požadované barvy. Pokud program o tuto barvu ještě nežádal, bude pro něj vytvořena nová instance, pokud danou barvu potřeboval již někdo před ním, bude mu vrácen odkaz na dříve vytvořenou instanci.

185. Není to dohledávání dříve vytvořených instancí „dražší“ než vytvoření instance zcela nové?

Možný důvod: drahá výroba instancí

Jedná-li se navíc o nějakou třídu, jejíž instance vytvářím doopravdy draze, může být prvotní hledání existující instance výrazně kratší než vytvoření instance nové, takže ušetříš čas již při vyřizování žádosti o odkaz na instanci. Avšak ani u méně „drahých“ instancí nemusí být zdržení příliš veliké.

U všech tříd pak platí, že i když by bylo dohledávání případných dvojníků dražší, můžeme v celkovém součtu potřebný čas použitím tohoto vzoru ušetřit.

186. A na čem bychom jej ušetřili?

Šetříme na `equals(Object)`

Když vím, že v aplikaci nebudou existovat dvě instance se stejnou hodnotou, mohu nahradit volání metody `equals(Object)` obyčejným porovnáváním. Pokud je počet porovnávání výrazně větší než počet vytváření instancí (např. budeš-li v aplikaci používat mapy klíčované těmito položkami), můžeš v průběhu práce aplikace onen „vyplýtvaný čas“ získat zpět i s bohatými úroky.

Vědomí, že nebudou existovat jiné instance se stejnými hodnotami některých atributů, ti může v jistých případech usnadnit i konstrukci vlastní třídy. Tady bych už ale nerad zabíhal do detailů, protože to už opravdu záleží na konkrétní aplikaci.

Náklady na kontrolu by se měly vrátit

Abych to tedy shrnul: rozhodneš-li se pro použití návrhového vzoru *Originál*, měl by sis ujasnit, že se ti náklady (čas, paměť, systémové prostředky, …), které vyplýtváš na zjišťování existence požadované instance, v průběhu další práce aplikace vrátí.

187. Mohu mít k použití tohoto návrhového vzoru i jiné důvody než zvýšenou efektivitu programu?

Samozřejmě – ostatně většinou saháš po návrhových vzorech z jiného důvodu. Mne např. vede k použití tohoto vzoru u třídy `Barva` snaha zamezit některým kolizím.

Jiné možné důvody použití

Chtěl jsem začátečníkům umožnit zadávat barvu nejenom prostřednictvím jejího odstínu, ale také prostřednictvím jejího názvu. Protože se však názory na to, která barva je pro daný účel ta nejlepší, většinou různí, chtěl jsem jim umožnit výchozí sadu libovolně rozšiřovat o nové pojmenované barvy. No a tady jsem potřeboval vyřešit jednoznačné přiřazení názvu barvy danému odstínu a jejich možné kolize. Použití vzoru *Originál* mi připadalo jako optimální.

Implementace

188. No dobře. Budu ti věřit, že se mi v daném případě všechny mé počáteční investice vrátí. Jak mám ale tento vzor implementovat?

Možností je několik. Záleží na tom, podle čeho budeš rozhodovat o tom, které dvě instance je možné považovat za shodné a nahradit je společnou instancí zastupující obě dvě.

Nejčastěji shodné atributy

Nejčastěji si zřejmě vybereš nějaké atributy, podle jejichž hodnoty se budeš rozhodovat. Pro každou požadovanou kombinaci hodnot atributů vytvoříš novou instanci. Pak bude asi nejvýhodnější definovat mapu, jejíž klíč bude odvozen z hodnot porovnávaných atributů. Požádá-li tě někdo o odkaz na instanci s danými atributy, „jukneš“ do mapy, a najdeš-li tam takovou instanci, vrátíš odkaz na ni. Nebude-li tam, vytvoříš novou a hned ji zařadíš do mapy s klíčem odvozeným z jejích atributů.

189. Říkáš, že nejčastěji si vyberu atributy, podle jejichž hodnoty se budu rozhodovat. Z toho bych odvozoval, že by to šlo také jinak. Jak?

Barva: dvě sady atributů

Vrátím se opět ke třídě `Barva`, o níž jsem se zmiňoval již v diskusi o motivaci k sáhnutí po tomto vzoru. Instance této třídy mají dvě sady atributů, jejichž hodnoty se navzájem ovlivňují. Uživatel si může vybrat barvu podle jejího názvu anebo může

zadat její konkrétní barevný odstín (tj. velikost jejích jednotlivých barevných složek a průhlednosti) bez ohledu na její název.

Použitím návrhového vzoru *Originál* zabezpečuji, aby nemohly vzniknout dva různé, avšak stejně pojmenované odstíny nebo naopak dva názvy stejného odstínu. Zadá-li uživatel pouze název, musí se jednat o některou z dosud vytvořených barev, protože program z názvu požadovaný odstín neodvozuje. Zadá-li uživatel naopak odstín, nemusí pro něj vymýšlet název.

Zadá-li odstín i název a bude-li se jednat o některý z již vytvořených odstínů, musí se shodovat i název, jinak program vyhodí výjimku. Pokud však název nezadá, tak v případě žádosti o vytvoření zcela nového odstínu program přidělí odstínu název odvozený z podílu jeho jednotlivých barevných složek. Tento název je pak již po zbytek programu pro daný odstín závazný.

190. Jestli jsem to dobře pochopil, tak stejně jako u všech ostatních vzorů hlídajících počet instancí bude i tady konstruktor soukromý.

Soukromý konstruktor

Přesně tak. Konstruktor je soukromý a všechny požadavky na odkazy na instance musíš směřovat na některou z definovaných továrních metod.

191. Co kdybych se rozhodl vytvářet od dané třídy nějaké podtřídy? To bych pak musel definovat konstruktor jako chráněný.

Podtřídy a jejich problémy

Musel by sis to ale nejprve opravdu dobře rozmyslet. Rodičovská třídy by musela definovat základní objekt a její dceřiné třídy pak různé druhy speciálních objektů.

Kdysi jsem měl takto navrženou třídu `Směr`, která mohla mít až 360 instancí. Její podtřídou byla třída `Směr8`, která z nich „vyzobala“ pouze 8 základních a vedlejších směrů a její podtřídou pak třída `Směr4`, která měla pouze 4 instance (zmiňoval jsem se o nich již na straně 133 v odpovědi na otázku 180).

Při definici těchto tříd jsem ale narážel na řadu problémů spojených se zaváděním třídy a inicializací jejích konstantních atributů. Vzhledem k tomu, že třída `Směr8` byla podtřídou třídy `Směr`, zaváděla se do paměti nejprve třída `Směr`. Bylo proto potřeba deklarovat některé atributy, které byly ve skutečnosti atributy potomků, jako atributy rodičovské třídy.

Přibalím pro šťouraly tuto trojici tříd mezi doprovodné programy, ale tady bych se o nich nerad rozpovídával, protože tam bylo několik mírně řečeno nestandardních řešení. Jejich vysvětlování by mohlo patřit do učebnice některých pokročilejších obratů jazyka Java, ale určitě ne do učebnice návrhových vzorů a moderního programování.

192. Nu dobrá, prozraď mi tedy alespoň, proč jsi vlastně vůbec ty potomky definoval, když pak dělali takové potíže.

Příklad, proč definovat potomky

Protože tak bylo možné pro některé aplikace velice jednoduše omezit počet použitelných směrů. Když někdo definoval autíčko, které se umělo otáčet pouze do čtyř stran, tak by mu ostatní směry dělaly problémy.

Pokud byl ale autíčkem tank, tak ten mohl mít např. věž, která se dokázala otáčet daleko plynuleji. Nebyl přitom důvod, aby instance směru věže byly instancemi zcela jiné třídy než instance směru tanku.

Zkrátka a dobře: občas si aplikace vyžádá nestandardní řešení, ale to neznamená, že s tímto řešením budeš své úvahy začínat.

193. Na možnou definici odstupňovaných směrů se tedy podívám mezi doprovodné programy.

Snažil jsem se vše okomentovat, takže by ti snad studium toho programu nemělo dělat nějaké vážnější problémy.

194. Nenašel bys ve svých zásobách ještě nějaký příklad, který používá dědičnost bezproblémově?

Naši studenti na VŠE vytvářejí jako jednu ze semestrálních prací jednoduchou hru ovládanou slovními příkazy z klávesnice. Tyto příkazy jsou realizovány jako sloučení návrhových vzorů *Originál* a *Příkaz* (o něm si budeme povídat v kapitole *Řekni, až to budeš chtít (Příkaz – Command)* na straně 195).

Každý příkaz je představován objektem příslušné třídy. Měl by tedy být definován jako jedináček. V programu však nikdy nikdo nemusí o jeho vytvoření žádat, takže se o zabezpečení jeho jedináčkovství nemusíme starat do důsledků – nemusíme např. vytvářet ani statický atribut, ani tovární metodu.

Abstraktní třída, která je společným rodičem všech příkazů, totiž ve svém konstruktoru zařadí každý vytvářený příkaz do mapy vytvořených příkazů a nedovolí do ní zadat jeden příkaz dvakrát. Stačí, když třída hry zavolá konstruktory všech příkazů, na které bude umět reagovat, a pak na ně může zapomenout. Kdykoliv někdo zadá hře příkaz, předá text uživatelova příkazu společnému rodiči všech příkazů, ten podle textu příkazu najde v mapě ten správný příkaz a předá mu text zadaný uživatelem. Vrácenou hodnotu pak nechá uživateli vypsat na obrazovku.

195. U jedináčka jsi hovořil o serializovatelnosti. Jak je to se serializovatelností originálů?

Serializovatelnost

Je to zcela stejné. Potřebuješ-li mít originální instance serializovatelné, musíš definovat metodu `readResolve()`, o které jsme hovořili na straně 116 v odpovědi na otázku 151. Pokud bude načtena již existující instance, vrátíš odkaz na tuto instanci, bude-li načtena dosud neexistující instance, zařadíš ji před vrácením odkazu mezi vytvořené.

Problém může nastat v okamžiku, kdy spolu „soupeří“ několik charakteristik, jako je tomu u třídy `Barva`, kde je daná barva jednoznačně charakterizována jak svým názvem, tak hodnotami svých barevných složek. Tam se charakteristiky objektu, který načítáme, nesmí rozejít s charakteristikami případného ekvivalentního objektu, který již v aplikaci existuje. Jinými slovy, má-li načítaná barva stejné hodnoty barevných složek jako některá z již existujících barev, musí mít i stejný název.

196. To ale není nejchytřejší, mít pro jeden objekt dvě sady charakteristik, které se mohou navzájem poprat.

Není, ale občas tě k takovémuto řešení zákazník dotlačí. V případě barvy je tomu tak proto, že jsem chtěl začátečníkům umožnit zadávat barvy co nejjednodušeji, takže mohou označit barvu pouze jejím názvem, u nějž se navíc nehledí na velikost písmen a na diakritiku. Jak ale procházejí kurzem a vytvářejí stále složitější programy, je

výhodné jim umožnit definovat barvy také pomocí barvených složek. Aby kvůli tomu nemuseli přebudovávat své původní programy, nezavádím používání třídy `java.awt.Color`, ale ukážu jim, jak lze dále využít možností nabízených třídou `Barva`. Studenti pak mohou doplnit své programy o nové možnosti, aniž by museli podstatně měnit původní návrh a dříve navržené části.

Příklad

197. Řekl bych, že implementaci jsme probrali křížem krážem, takže bychom si mohli ukázat nějaký příklad.

Ukážu ti definici třídy `Barva`, o níž jsem před chvílí hovořil a která je typickým zástupcem tohoto vzoru. Obdobná řešení, s jakými se v ní setkáš, by se ti mohla někdy hodit. Jak jsem ale již řekl: musíš vědět, že náklad (čas, paměť, …) ztracený kontrolou existence instancí se ti v průběhu další práce programu vrátí.

Zobrazený zdojový kód neobsahuje testovací metodu – tu si můžeš najít v souboru mezi doprovodnými programy.

Výpis 11.1: Definice třídy `Barva`

```
package rup.česky.tvary;

//=============================================================================
import java.awt.Color;
import java.io.ObjectStreamException;
import java.io.Serializable;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import rup.česky.společně.ToASCII;

/*****************************************************************************
 *  Třída Barva definuje skupinu základních barev pro použití
 *  při kreslení tvarů před zavedením balíčků.
 *  Není definována jako výčtový typ, aby si uživatel mohl libovolně přidávat
 *  vlastní barvy.
 *
 * @author     Rudolf Pecinovský
 * @version    2.02.000, 18.2.2007
 */
public class Barva implements Serializable
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Minimální velikost složky při změnách světlosti a průhlednosti. */
    private static final int MINC = 32;

    /** Maximální velikost složky při změnách světlosti a průhlednosti. */
    private static final int MAXC = 255;
```

```
/** Koeficient změny velikosti složky při změnách světlosti a průhlednosti.*/
private static final double KC = 0.7;

/** Mapa všech doposud vytvořených barev klíčovaná jejich názvy. */
private static final Map<String,Barva> NÁZVY =
                                       new LinkedHashMap<String,Barva>();
/** Mapa všech doposud vytvořených barev klíčovaná jejich barevností. */
private static final Map<Color,Barva>  COLOR =
                                       new LinkedHashMap<Color,Barva>();
/** Seznam všech dosud vytvořených barev. */
private static final List<Barva> BARVY = new ArrayList<Barva>( 32 );


/** Černá = RGBA( 0, 0, 0, 255); */          public static final Barva
ČERNÁ   = new Barva( 0x00, 0x00, 0x00, 0xFF, "černá"   );

/** Modrá = RGBA( 0, 0, 255, 255); */        public static final Barva
MODRÁ   = new Barva( 0x00, 0x00, 0xFF, 0xFF, "modrá"   );

/** Červená = RGBA( 255, 0, 0, 255); */      public static final Barva
ČERVENÁ = new Barva( 0xFF, 0x00, 0x00, 0xFF, "červená" );

/** Fialová = RGBA( 255, 0, 255, 255); */    public static final Barva
FIALOVÁ = new Barva( 0xFF, 0x00, 0xFF, 0xFF, "fialová" );

/** Zelená = RGBA( 0, 255, 0, 255); */       public static final Barva
ZELENÁ  = new Barva( 0x00, 0xFF, 0x00, 0xFF, "zelená"  );

/** Azurová = RGBA( 0, 255, 255, 255); */    public static final Barva
AZUROVÁ = new Barva( 0x00, 0xFF, 0xFF, 0xFF, "azurová" );

/** Žlutá = RGBA( 255, 255, 0, 255); */      public static final Barva
ŽLUTÁ   = new Barva( 0xFF, 0xFF, 0x00, 0xFF, "žlutá"   );

/** Bílá = RGBA( 255, 255, 255, 255); */     public static final Barva
BÍLÁ    = new Barva( 0xFF, 0xFF, 0xFF, 0xFF, "bílá"    );

/** Krémová = RGBA( 255, 255, 204, 255); */  public static final Barva
KRÉMOVÁ = new Barva( 0xFF, 0xFF, 0xCC, 0xFF, "krémová" );

/** Šedá = RGBA( 153, 153, 153, 255); */     public static final Barva
ŠEDÁ    = new Barva( 0x99, 0x99, 0x99, 0xFF, "šedá"    );

/** Ocelová = RGBA( 0, 153, 204, 255); */    public static final Barva
OCELOVÁ = new Barva( 0x00, 0x99, 0xCC, 0xFF, "ocelová" );

/** Růžová = RGBA( 255, 153, 153, 255); */    public static final Barva
RŮŽOVÁ  = new Barva( 0xFF, 0x99, 0x99, 0xFF, "růžová"  );

/** Hnědá = RGBA( 153, 51, 0, 255); */       public static final Barva
HNĚDÁ   = new Barva( 0x99, 0x33, 0x00, 0xFF, "hnědá"   );

/** Khaki = RGBA( 153, 153, 0, 255); */      public static final Barva
KHAKI   = new Barva( 0x99, 0x99, 0x00, 0xFF, "khaki"   );
```

```
    /** Cihlová = RGBA( 255, 102, 51, 255); */   public static final Barva
    CIHLOVÁ = new Barva( 0xFF, 0x66, 0x33, 0xFF, "cihlová" );

    /** Zlatá = RGBA( 255, 153, 0, 255); */      public static final Barva
    ZLATÁ   = new Barva( 0xFF, 0x99, 0x00, 0xFF, "zlatá"   );

    /** Stříbrná = RGBA( 204, 204, 204, 255); */ public static final Barva
    STŘÍBRNÁ= new Barva( 0xCC, 0xCC, 0xCC, 0xFF, "stříbrná");

    /** Kouřová = RGBA( 0,0,0,128)-průsvitná!*/  public static final Barva
    KOUŘOVÁ = new Barva( 0x99, 0x99, 0x99, 0x80, "kouřová");

    /** Mléčná=RGBA(255,255,255,128)-průsvitná!*/public static final Barva
    MLÉČNÁ  = new Barva( 0xFF, 0xFF, 0xFF, 0x80, "mléčná");

    /** Žádná = RGBA( 0,0,0,255) - PRŮHLEDNÁ */  public static final Barva
    ŽÁDNÁ  = new Barva( 0xFF, 0xFF, 0xFF, 0x0, "žádná");


    private static final long serialVersionUID = 42L;


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final String název;     //Název barvy zadávaný konstruktoru
    private final Color  color;     //Barva ze standardní knihovny


//== NESOUKROMÉ METODY TŘÍDY ===================================================

    /*****************************************************************************
     * Vrátí pole řetězců s názvy definovaných barev.
     *
     * @return  Pole řetězců s názvy známých barev
     */
    public static String[] getZnáméNázvy()
    {
        int barev = BARVY.size();
        String[] s = new String[ barev ];
        for( int i=0;   i < barev;   i++ )
            s[i] = BARVY.get(i).název;
        return s;
    }


    /*****************************************************************************
     * Vrátí pole definovaných barev.
     *
     * @return  Pole řetězců s názvy známých barev
     */
    public static Barva[] getZnáméBarvy()
    {
        return BARVY.toArray( new Barva[0] );
    }
```

```
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /**************************************************************************
     * Vytvoří instanci barvy se zadanou velikostí barevných složek
     * a hladinou průhlednosti nastavovanou v kanále alfa
     * a se zadaným českým názvem.
     *
     * @param red       Velikost červené složky
     * @param green     Velikost zelené složky
     * @param blue      Velikost modré složky
     * @param název     Název vytvořené barvy
     */
    private Barva( int red, int green, int blue, int alpha, String název )
    {
        this( new Color(red, green, blue, alpha),  název );
    }


    /**************************************************************************
     * Vytvoří barvu ekvivalentní zadané instanci třídy <code>Color</code>
     * se zadaným českým názvem.
     *
     * @param c      Instance třídy <code>Color</code> požadované barvy
     * @param název  Název vytvářené barvy; <code>null</code> označuje,
     *               že se má pro barvu vytvořit generický název
     */
    private Barva( Color c, String název )
    {
        this.color = c;
        if( název == null ) {
            název = "barva(r=" + c.getRed()  + ",g=" + c.getGreen() +
                        ",b=" + c.getBlue() + ",a=" + c.getAlpha() + ")"
                    .toLowerCase(); //pro jistotu, kdybych na text sáhl
            while( NÁZVY.get(název) != null )
                název += '+';
        }
        this.název = název;

        BARVY.add( this );
        COLOR.put( color, this );
        NÁZVY.put( název, this );
        //Obsahuje-li název diakritiku, uloží i jeho verzi bez diakritiky
        String bhc = ToASCII.string( název );
        if( ! název.equals(bhc) )
            NÁZVY.put( bhc, this );
    }


    /**************************************************************************
     * Převede název barvy na příslušný objekt typu Barva.
     *
     * @param názevBarvy  Název požadované barvy – seznam známých názvů
     *                    je možno získat zavoláním metody getZnáméNázvy()
     *
     * @return Instance třídy Barva reprezentující zadanou barvu
     *
```

```
     * @throws IllegalArgumentException není-li barva (název) mezi známými
     */
    public static Barva getBarva( String názevBarvy )
    {
        Barva barva = NÁZVY.get( názevBarvy.toLowerCase() );
        if( barva != null )
            return barva;
        else
            throw new IllegalArgumentException(
                "\nTakto pojmenovanou barvu neznám: " + názevBarvy );
    }


    /****************************************************************************
     * Vytvoří instanci barvy se zadanou velikostí barevných složek.
     *
     * @param red   Velikost červené složky
     * @param green Velikost zelené složky
     * @param blue  Velikost modré složky
     * @return Barva se zadanými velikostmi jednotlivých složek
     */
    public static Barva getBarva( int red, int green, int blue )
    {
        return getBarva( red, green, blue, 0xFF );
    }


    /****************************************************************************
     * Existuje-li zadaná barva mezi známými, vrátí ji; v opačném případě
     * vytvoří instanci barvy se zadanou velikostí barevných složek a
     * hladinou průhlednosti nastavovanou v kanále alfa
     * a pojmenuje ji prostřednictvím jejích složek.
     *
     * @param red    Velikost červené složky
     * @param green  Velikost zelené složky
     * @param blue   Velikost modré složky
     * @param alpha  Hladina průhlednosti: 0=průhledná, 255=neprůhledná
     * @return Barva se zadanými velikostmi jednotlivých složek
     */
    public static Barva getBarva( int red, int green, int blue, int alpha )
    {
        Color color = new Color( red, green, blue, alpha );
        Barva barva = COLOR.get( color );
        if( barva != null )
            return barva;
        return new Barva( color, null );
    }


    /****************************************************************************
     * Existuje-li zadaná barva mezi známými, vrátí ji; v opačném případě
     * vytvoří novou barvu se zadanými parametry a vrátí odkaz na ni.
     *
     * @param red       Velikost červené složky
     * @param green     Velikost zelené složky
     * @param blue      Velikost modré složky
```

```
     * @param název     Název vytvořené barvy
     *
     * @return Barva se zadaným názvem a velikostmi jednotlivých složek
     * @throws IllegalArgumentException má-li některá ze známých barev některý
     *         ze zadaných názvů a přitom má jiné nastavení barevných složek
     *         nebo má jiný druhý název.
     */
    public static Barva getBarva( int red, int green, int blue, String název )
    {
        return getBarva( red, green, blue, 0xFF, název );
    }


    /***************************************************************************
     * Existuje-li zadaná barva mezi známými, vrátí ji; v opačném případě
     * vytvoří novou barvu se zadanými parametry a vrátí odkaz na ni.
     *
     * @param red       Velikost červené složky
     * @param green     Velikost zelené složky
     * @param blue      Velikost modré složky
     * @param alpha     Hladina průhlednosti: 0=průhledná, 255=neprůhledná
     * @param název     Název vytvořené barvy
     *
     * @return Instance třídy Barva reprezentující zadanou barvu.
     *
     * @throws IllegalArgumentException má-li některá ze známých barev některý
     *         ze zadaných názvů a přitom má jiné nastavení barevných složek
     *         nebo má jiný druhý název.
     */
    public static Barva getBarva( int red, int green, int blue, int alpha,
                                  String název )
    {
        if( (název == null)  ||
            ((název = název.trim().toLowerCase()) == "") )
        {
            throw new IllegalArgumentException(
                "\nPro požadovanou barvu byl zadán prázdný název" );
        }
        Color color = new Color( red, green, blue, alpha );
        Barva vNázvech = NÁZVY.get( název.toLowerCase() );
        Barva vBarvách = COLOR.get( color );

        if( vNázvech != vBarvách )
        //Alespoň jedna z barev je již definována (jinak by obě byly == null),
        //a přitom název ukazuje na jinou barvu než barevná charakteristika
        {
            Barva b = (vNázvech == null)  ? vBarvách : vNázvech;
            Color c = b.color;
            throw new IllegalArgumentException(
                "\nZadané parametry kolidují s parametry existující barvy"+
                " [existující × zadaná]:" +
                "\nNázev: "          + b.název       + "  ×  " + název  +
                "\nČervená složka: " + c.getRed()    + "  ×  " + red   +
                "\nZelená  složka: " + c.getGreen()  + "  ×  " + green +
                "\nModrá   složka: " + c.getBlue()   + "  ×  " + blue  +
                "\nPrůhlednost:    " + c.getAlpha()  + "  ×  " + alpha );
```

```
        }
        else if( vBarvách != null )
        //Oba odkazy jsou shodné a přitom nenullové => existující barva
            return vBarvách;
        else
        //Oba odkazy jsou shodně null => vytvoříme novou barvu
            return new Barva( color, název );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Převede námi používanou barvu na typ používaný kreslítkem.
     * Metoda je používaná ve třídě <code>SprávcePlátna</code>.
     *
     * @return Instance třídy Color reprezentující zadanou barvu
     */
    public Color getColor()
    {
        return color;
    }


    /**************************************************************************
     * Vrátí řetězec s charakteritikou dané barvy obsahující název a
     * hodnoty barevných složek uvedené v desítkové soustavě
     *
     * @return Řetězec s charakteristikou barvy
     */
    public String getCharakteristikaDec()
    {
        return String.format( "(d:R=%d,G=%d,B=%d,A=%d)-%s",
            color.getRed(), color.getGreen(), color.getBlue(),
            color.getAlpha(), název );
    }


    /**************************************************************************
     * Vrátí řetězec s charakteritikou dané barvy obsahující název a
     * hodnoty barevných složek uvedené v šestnáctkové soustavě
     *
     * @return Řetězec s charakteristikou barvy
     */
    public String getCharakteristikaHex()
    {
        return String.format( "(x:R=%02X,G=%02X,B=%02X,A=%02X)-%s",
            color.getRed(), color.getGreen(), color.getBlue(),
            color.getAlpha(), název );
    }


    /**************************************************************************
     * Vrátí název barvy.
     *
     * @return Řetězec definující zadanou barvu.
```

```
     */
    public String getNázev()
    {
        return název;
    }


    /*************************************************************************
     * Vrátí barvu inverzní k zadané barvě, tj. barvu s inverzními
     * hodnotami jednotlivých barevných složek, ale se stejnou průhledností.
     *
     * @return Inverzní barva
     */
    public Barva inverzní()
    {
        return getBarva( MAXC - color.getRed(),
                         MAXC - color.getGreen(),
                         MAXC - color.getBlue(), color.getAlpha() );
    }


    /*************************************************************************
     * Vrátí název barvy.
     *
     * @return  Název barvy
     */
    public String toString()
    {
        return název;
    }


    /*************************************************************************
     * Vrátí světlejší verzi dané barvy. Pozor, vzhledem k zaokrouhlovacím
     * chybám nejsou operace zesvětli a ztmav zcela reverzní.
     *
     * @return Světlejší verze barvy
     */
    public Barva zesvětli()
    {
        Color c = color.brighter();
        if( c.equals(color) )
            c = new Color( Math.max( c.getRed(),   MINC ),
                           Math.max( c.getGreen(), MINC ),
                           Math.max( c.getBlue(),  MINC ),  c.getAlpha() );
        return getBarva( c );
    }


    /*************************************************************************
     * Vrátí méně průhlednou verzi dané barvy. Pozor, vzhledem k zaokrouhlovacím
     * chybám nejsou operace zprůhledni a zneprůhledni zcela reverzní.
     *
     * @return Méně průhledná verze barvy
     */
    public Barva zneprůhledni()
```

```
    {
        int a = Math.max( Math.min((int)(color.getAlpha()/KC), MAXC), MINC);
        return getBarva(
            new Color( color.getRed(), color.getGreen(), color.getBlue(), a ));
    }


    /**************************************************************************
     * Vrátí průhlednější verzi dané barvy. Pozor, vzhledem k zaokrouhlovacím
     * chybám nejsou operace zprůhledni a zneprůhledni zcela reverzní.
     *
     * @return Průhlednější verze barvy
     */
    public Barva zprůhledni()
    {
        int a = (int)(color.getAlpha() * KC);
        return getBarva(
            new Color( color.getRed(), color.getGreen(), color.getBlue(), a ));
    }


    /**************************************************************************
     * Vrátí tmavší verzi dané barvy. Pozor, vzhledem k zaokrouhlovacím
     * chybám nejsou operace zesvětli a ztmav zcela reverzní.
     *
     * @return Tmavší verze barvy
     */
    public Barva ztmav()
    {
        return getBarva( color.darker() );
    }


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ============================================

    /**************************************************************************
     * Existuje-li zadaná barva mezi známými, vrátí ji; v opačném případě
     * vytvoří instanci barvy se zadanou velikostí barevných složek a
     * hladinou průhlednosti nastavovanou v kanále alfa
     * a pojmenuje ji prostřednictvím jejích složek.
     *
     * @param red    Velikost červené složky
     * @param green  Velikost zelené složky
     * @param blue   Velikost modré složky
     * @param alpha  Hladina průhlednosti: 0=průhledná, 255=neprůhledná
     * @return Barva se zadanými velikostmi jednotlivých složek
     */
    private static Barva getBarva( Color c )
    {
        Barva b = COLOR.get( c );
        if( b != null )
            return b;
        else
            return getBarva( c.getRed(),  c.getGreen(),
                             c.getBlue(), c.getAlpha() );
    }
```

```
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ================================

    /*****************************************************************************
     * Vrátí barvu se stejnými barevnými složkami, a je-li to možné,
     * tak i se stejným názvem. Název barvy však není považován za závazný,
     * takže pokud již barva zadaného názvu (avšak jiného odstínu) existuje,
     * přiřadí vracené barvě generický název.
     */
    private Barva readResolve() throws ObjectStreamException
    {
        Barva bc = COLOR.get( color );
        if( bc != null )
            //Barva se stejnými složkami již existuje => tu vrátíme
            return bc;
        //Pokud barva zadaného odstínu neexistuje, bude se vždy vytvářet nová,
        //aby se zabezpečilo její zařazení do map

        //Barva neexistuje - je volný i název?
        bc = NÁZVY.get( název.toLowerCase() );
        if( bc == null )
            //Neexistuje barva daného názvu - můžeme jej barvě přiřadit
            return new Barva( color, název );
        else
            //Název už existuje => barva dostane generický název
            return new Barva( color, null );
    }

// ... Vynechané testy
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Originál* popisuje jednodušší verzi fondu, která nevyžaduje návrat použitých instancí do fondu.
- Účelem je, aby se každá instance vyskytovala v programu jenom jednou, tj. aby instance neměly dvojníky se shodnou hodnotou.
- Čas ztracený při kontrole existence požadované instance můžeme získat zpět rychlejším porovnáváním instancí.
- Pro použití vzoru *Originál* mohou být i jiné důvody, např. ošetření některých možných kolizí.
- Instance třídy implementující vzor *Originál* nejčastěji ukládáme do mapy, přičemž klíčem jsou hodnoty atributů, které ji charakterizují.
- Od třídy originálů je možné vytvářet podtřídy, ale je třeba se připravit na možné problémy.
- Budeme-li chtít mít instance třídy navržené podle vzoru *Originál* serializovatelné, musíme definovat metodu `readResolve()`, která zabezpečí, aby nevznikaly kopie existujících instancí.
- Návrhový vzor *Originál* nepatří mezi vzory z GoF.

KAPITOLA 12

Konečný počet instancí (Fond – Pool)

- **Účel**
- **Implementace**
- **Příklad: Molekuly**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru

Fond využijeme ve chvíli, kdy potřebujeme z nějakého důvodu omezit počet vytvořených instancí a místo vytváření instancí nových dáme přednost „reinkarnaci“ (znovuoživení) instancí dříve použitých a v danou chvíli již nepoužívaných.

Účel

198. Na počátku předchozí kapitoly jsi říkal, že na případy, ve kterých se nejedná o předem známé instance, se aplikuje návrhový vzor *Fond*.

Kdy se hodí

Ano. V řadě situací nechceš připustit, aby vzniklo příliš mnoho instancí některé třídy. Nejčastějším důvodem pro použití fondu je to, že vytvoření a udržování instancí požadované třídy je náročné na paměť, systémové prostředky nebo strojový čas. Aplikace si proto vytvoří na ony „drahé" instance speciální fond a použité instance se pak nezahodí, nýbrž se odloží do tohoto fondu na dobu, až je bude zase někdo potřebovat. Když pak někdo požádá o instanci, nebude se vytvářet nová, ale bude mu poskytnuta některá z dříve vytvořených instancí uložených ve fondu.

Dva způsoby realizace

199. A co když ve fondu žádná volná instance není?

To lze řešit dvěma způsoby:

– pevný počet instancí s frontou požadavků

- Dotyčný požadavek se zařadí do fronty čekajících požadavků, a až se ve fondu objeví volná instance, bude mu přidělena. Tento způsob se používá v situacích, kdy je vytvoření a udržování instance opravdu „drahé" a očekávané čekací doby naopak přijatelně krátké.

– dynamické zvyšování počtu instancí dle okamžité potřeby

- Fond se vytvoří dynamický a vždy, když po příchodu požadavku není ve fondu volná instance, vytvoří se nová. Předpokládá se přitom, že počet současně používaných instancí bude rozumně malý. Proto je i v tomto případě rozumné hlídat překročení nějakého předem zadaného počtu instancí a případně vyvolat výjimku. Ale to jsou již detaily implementace.

200. Hýříš takovými nic neříkajícími „politickými" přívlastky jako opravdu drahé, přijatelně krátké, rozumně malé. Kolik instancí je např. rozumně málo?

Kolik je „rozumně málo"

To záleží na třídě a aplikaci. Někdy může být „rozumně málo" tisíc a jindy je pět už příliš mnoho.

201. Přivítal bych nějaké příklady.

Příklad konečného fondu

Příkladem instancí, které bývají ukládány do konečně velkého a navíc velmi malého fondu bývají např. připojení k databázi. Typicky jich nebývá současně otevřeno více než 5 a často se dokonce používá pouze jediné. Ono také jedno připojení k databázi může trvat i několik vteřin, takže nikdo nestojí o to, aby tento čas kvůli každému dotazu ztrácel.

Příklad dynamicky rostoucího fondu

Mechanismus rostoucího fondu se používá např. pro vlákna spojená s animovanými objekty pohybujícími se po obrazovce. Objekty vznikají a zanikají, avšak v každém okamžiku jich existuje pouze nějaký omezený počet. Ve chvíli, kdy animovaný objekt vznikne, je třeba mu přidělit vlákno, když zanikne, vrátí se vlákno zpět do fondu, kde čeká na vznik dalšího animovaného objektu, kterému bude přiděleno.

Animovaný objekt nemůže čekat, až se nějaké vlákno uvolní, protože by pak stál, zatímco ostatní by se pohybovaly. Pokud tedy ve fondu žádné volné vlákno není, musí se vytvořit nové.

202. A nějaké příklady ze standardní knihovny?

Příklady ze standardní knihovny

Opět celá řada. Když jsem hovořil o vláknech, začnu hned u nich. Java 5.0 přišla s novou knihovnou pro práci s vlákny, v níž jsou takzvané *exekutory*, které bychom mohli považovat za chytřejší vlákna. Ty jsou uloženy v zásobnících, jimiž jsou instance třídy `java.util.concurrent.ThreadPoolExecutor` a jejích potomků.

O připojeních k databázi jsem již hovořil. Když v aplikaci zavoláš metodu `getConnection()` nějaké instance rozhraní `DataSource`, obdržíš sice instanci rozhraní `java.sql.Connection`, ale tato instance bývá často pouze obálkou pro odkaz na instanci rozhraní `javax.sql.PooledConnection`, což je spojení přechovávané ve fondu.

Implementace

203. Řekl bych, že z toho, cos' mi pověděl, bych asi dokázal takový fond navrhnout. Myslíš, že je tam něco, na co bych mohl narazit?

Zásady pro implementaci

Ono to není tak prosté, jak to na první pohled vypadá. Začátečník zde může zapomenout na řadu detailů, které sníží efektivnost jeho funkce, nebo dokonce ohrozí jeho správnou činnost. Raději bych proto probral potřebné zásady jednu po druhé.

- Při zřizování fondu si vybereš, zda pro ukládání jednotlivých objektů zvolíš statické či dynamické pole. U fondů předem známé velikosti bývá výhodnější statické pole, u fondů s předem neznámou konečnou velikostí bývá rozumné použít nějaký dynamický kontejner, nejčastěji seznam (`List`).
- Fond vybavíš metodami pro získání volného objektu a pro vrácení používaného objektu poté, co jej přestaneš potřebovat.
- S využitím parametrizovaných typů je možno definovat obecný fond, kterému zadáš typ ukládaných prvků až při jeho konstrukci.
- Očekáváš-li ve fondu malý počet prvků (tak do 5), můžeš volný prvek vyhledávat procházením uložených prvků. Při větším počtu prvků je výhodnější zvolit nějaké efektivnější řešení.
- V některých situacích může být takovým řešením mapa, ve které poměrně rychle vyhledáš prvek, který z ní chceš vyjmout.
- Jinou možností je používat pro všechny prvky jeden seznam a zaměnit při každém uvolnění používaného prvku v kontejneru právě uvolněný prvek s posledním používaným. Tím dosáhneš toho, aby byly používané prvky neustále na počátku seznamu a volné prvky na jeho konci. Bude-li si každý prvek pamatovat svůj index a bude-li si fond průběžně pamatovat index prvního volného prvku, celá operace se výrazně zrychlí.

Univerzální fond

204. Mohl bys mi tu ukázat definici nějakého jednoduchého fondu?

Připravil jsem si pro tebe definici takového malého univerzálního fondu, do něhož můžeš ukládat instance libovolné třídy. Vlastní fond implementuje rozhraní `IFond`, které deklaruje sadu metod, jež by měl fond implementovat. Jeho definici najdeš ve výpisu 12.1.

Výpis 12.1: Rozhraní `IFond`

```
package rup.česky.vzory._12_fond;

/*******************************************************************************
 * Rozhraní IFond definuje základní schopnosti fondu uchovávající instance,
 * jejichž typ je zadán jako typový parametr rozhraní.
 *
 * @param <T> Typ instancí uložených ve fondu
*/
public interface IFond<T>
{
//== POŽADOVANÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Vrátí volnou instanci; není-li v rezervoáru žádná volná instance,
     * vytvoří novou.
     * @return Požadovaná instance
     */
    public T dejIhned();


    /*******************************************************************************
     * Vrátí volnou instanci; není-li v rezervoáru žádná volná instance,
     * zařadí žadatele do fronty čekajících.
     * @return Požadovaná instance
     */
    public T dejPočkám();


    /*******************************************************************************
     * Vrátí volnou instanci; není-li v rezervoáru žádná volná instance,
     * vrátí null;
     * @return Požadovaná instance, není-li žádná volná, vrátí <code>null</code>
     */
    public T dejVolnou();


    /*******************************************************************************
     * Předá do rezervoáru dále nepotřebnou instanci.
     * @param instance Instance vracená do fondu
     */
    public void vracímInstanci( T instance );

}
```

205. To jsem ještě pochopil. Teď bych ale rád viděl definici vlastního fondu.

Zřizování instancí je třeba dodat zvenku

Ještě chvíli počkej. Fond musí umět vytvářet požadované instance. Má-li však být univerzální, nemůžeš do něj tuto možnost zabudovat přímo, ale musíš mu ji dodat zvenku.

Já jsem to vyřešil tak, že jsem mezi parametry konstruktoru zařadil instanci rozhraní `ITovárna`, které deklaruje tovární metodu `newInstance()`, jež na požádání vytvoří novou instanci a vrátí odkaz na ni. Jeho definici najdeš ve výpisu 12.2.

Výpis 12.2: Rozhraní `ITovárna`

```
package rup.česky.vzory._12_fond;

/*****************************************************************************
 * Instance rozhraní ITovárna obsahují tovární metodu vracející odkaz na
 * novou instanci typu, který je typovým parametrem rozhraní.
 *
 * @param <T> Typ instancí "vyráběných" tovární metodou
*/
public interface ITovárna<T>
{
    /*************************************************************************
     * Vytvoří novou instanci a vrátí odkaz na ni.
     * @return Požadovaná instance
     */
    public T newInstance();
}
```

206. Tak teď už mi prozradíš, jak vypadá ten tvůj zázračný fond?

Fond jsem se snažil definovat jako jednoduchý univerzální fond. Není proto vázán na žádný specifický typ ukládaných instancí ani na některé způsoby žádání o novou instanci. Prohlédni si výpis 12.3, a nebude-li ti něco jasné, zeptej se.

Výpis 12.3: Třída `Fond`

```
package rup.česky.vzory._12_fond;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/*****************************************************************************
 * Třída Fond demonstruje implementaci fondu,
 * v němž jsou uloženy instance vytvářené generátorem (továrnou)
 * předaným konstruktoru jako parametr.
 *
 * @param <T> Typ instancí uložených ve fondu
 */
public class Fond<T> implements IFond<T>
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Seznam volných instancí. */
    private final List<T> volné = new ArrayList<T>();

    /** Množina instancí, které si někdo z fondu vyzvedl a stále je používá. */
    private final Set<T> užívané = new HashSet<T>();

    /** Generátor instancí ukládaných do fondu. */
    private final ITovárna<T> generátor;

    /** Povolená kapacita fondu,
```

```
     * tj. maximální přípustný počet uchovávaných instancí. */
    private final int kapacita;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Celkový počet instancí ve fondu. */
    private int celkem = 0;


//##############################################################################
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří prázdný fond s neomezenou kapacitou.
     * @param továrna Generátor instancí uchovávaných ve fondu
     */
    public Fond( ITovárna<T> továrna )
    {
        this( továrna, 0 );
    }


    /*****************************************************************************
     * Vytvoří prázdný fond se zadanou maximální kapacitou.
     * Nulová maximální kapacita označuje neomezenou maximální velikost fondu.
     *
     * @param kapacita Maximální povolený počet instancí ve fondu,
     *                 nulový počet označuje neomezenou velikost
     * @param továrna  Generátor instancí uchovávaných ve fondu
     */
    public Fond( ITovárna<T> továrna, int kapacita )
    {
        this( továrna, kapacita, 0 );
    }


    /*****************************************************************************
     * Konstruktor vytvoří fond se zadaným počtem připravených instancí
     * a zadanou maximální kapacitou. Nulový maximální počet instancí označuje
     * neomezenou maximální velikost fondu.
     *
     * @param start    Počet instancí, které budou připraveny již na počátku
     * @param kapacita Maximální povolený počet instancí ve fondu,
     *                 nulový počet označuje neomezenou velikost
     * @param továrna  Generátor instancí uchovávaných ve fondu
     * @throws RuntimeException v případě,
     *         kdy zadaná třída nebude mít požadovanou metodu
     *         nebo tato metoda nevytvoří požadovanou instanci
     */
    public Fond( ITovárna<T> továrna, int kapacita,  int start )
    {
        if( start < 0)
            throw new IllegalArgumentException (
                "\nNelze začínat se záporným počtem uložených prvků -  start="
                 + start );
```

```
        if( (kapacita != 0)  &&  (kapacita < start) )
            throw new IllegalArgumentException (
                "\nMaximální počet prvků je menší než počáteční: kapacita=" +
                kapacita + ", start=" + start );
        this.kapacita = (kapacita > 0)
                      ?  kapacita
                      :  Integer.MAX_VALUE;
        generátor = továrna;
        for( int i=0;   i < start;   i++ ) {
            T t = vytvořDalší( true );
            volné.add( t );
        }
    }



//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Vrátí volnou instanci; není-li ve fondu žádná volná instance,
     * zařadí žadatele do fronty čekajících.
     * @return Volná instance
     */
    public synchronized T dejPočkám()
    {
        if( volné.size() > 0 ) {    //Existují volné instance
            return dodejVolnou();
        } else if( celkem < kapacita ) {
            //Volné instance nejsou, ale fond má volnou kapacitu
            return vytvořDalší( false );
        } else {
            //Volné nejsou a kapacita je vyčerpána
            do {
                try {
                    wait(); //Počká, až někdo uvolní nějakou instanci
                } catch( InterruptedException ie ) {
                    //Případné přerušení není v tomto demu ošetřeno
                }
            }while( volné.size() == 0 );
            return dodejVolnou();
        }
    }


    /**************************************************************************
     * Vrátí volnou instanci; není-li ve fondu žádná volná instance,
     * vytvoří novou.
     * Metodu je nebezpečné používat u fondů s omezenou kapacitou,
     * protože při překročení povolené kapacity vyvolá výjimku.
     * @return Volná instance
     * @throws IllegalStateException Je-li při volání metody již naplněna
     *                               kapacita fondu, tj. další se tam nevejde
     */
    public synchronized T dejIhned()
    {
        if( volné.size() > 0 ) {
```

```
            return dodejVolnou();
        } else {
            return vytvořDalší( false );
        }
    }


    /*************************************************************************
     * Vrátí volnou instanci; není-li ve fondu žádná volná instance,
     * vrátí <code>null</code>.
     * @return Volná instance nebo <code>null</code>
     */
    public synchronized T dejVolnou()
    {
        if( volné.size() > 0 ) {
            return dodejVolnou();
        } else {
            return null;
        }
    }


    /*************************************************************************
     * Vrátí do fondu dále nepotřebnou instanci.
     * @param instance Vracená instance
     */
    public synchronized void vracímInstanci( T instance )
    {
        //Kontrola korektnosti vracené instance
        if( !užívané.contains( instance ) )
            throw new IllegalArgumentException (
                "\nNelze vrátit instanci, která nebyla odebraná z fondu" );
        užívané.remove( instance ); //Odebere instanci z množiny užívaných
        volné.add( instance );      //a přidá ji do seznamu volných
        notify();   //Upozorní případné čekající vlákno na to,
                    //že ve fondu se objevila volná  instance
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ================================

    /*************************************************************************
     * Vrátí některou z volných instancí.
     * @return Volná instance
     */
    private T dodejVolnou()
    {
        T ret = volné.remove( volné.size()-1 );
        užívané.add( ret );
        return ret;
    }


    /*************************************************************************
     * Nechá vytvořit novou instanci a vrátí odkaz na ni.
     * @param volná Informace o tom, má-li být vytvořená instance zařazena
```

```
     *              mezi volné (<code>true</code>) nebo mezi
     *              používané (<code>false</code>).
     * @throws IllegalStateException Je-li při volání metody již naplněna
     *                               kapacita fondu, tj. další se tam nevejde
     */
    private T vytvořDalší( boolean volná )
    {
        if( (kapacita != 0)  &&  (celkem >= kapacita) )
            throw new IllegalStateException(
                "\nByla vyčerpána povolená kapacita fondu " + kapacita);
        T t = generátor.newInstance();
        celkem++;
        if( volná )
            volné.add( t );
        else
            užívané.add( t );
        return t;
    }
```

207. Říkal jsi, že když nebudou sdílené instance k dispozici, budou na ně čekat žadatelé ve frontě. Žádné fronty jsem si tam ale nevšiml.

Fronta procesů

Ona tam je, jenomže není vidět. Tu frontu vlastně tvoří procesy čekající na to, až jim monitor přidělí klíč ke kritické sekci daného fondu. Je to sice taková zvláštní fronta s předbíháním, která funguje podobně, jako když se fanynky seběhnou kolem nějakého zpěváka a žádají jej o autogram, ale to není důležité.

208. Tak teď jsem tě moc nepochopil.

Pro pochopení je třeba znát programování vláken

Abys pochopil, musel by ses vyznat v programování vláken. O nich se tu ale nechci rozpovídávat (odložíme to do jiné knihy). Musím tě proto odkázat na některou z knih, které se touto problematikou alespoň trochu zajímají, např. [30], [33], [37].

209. Nu dobrá. Budu si myslet, že tam fronta je, i když žádnou nevidím. Nicméně abych si ujasnil, jak fond funguje, chtělo by to také nějaký funkční příklad.

Test práce fondu

Mám ho připravený – definici najdeš ve výpisu 12.4. I když se mi zdá, že testovací příklad by mohl být pro někoho hůře pochopitelný než celý fond. Aby test nebyl rozcouraný po několika souborech, tak jsem všechny potřebné třídy definoval v rámci testovací třídy. Snad tě to ale nezmate. Zkus si definici testu podrobně pročíst a zeptej se na to, co ti nebude jasné.

Hlavně si pak příklad spusť a podívej se, jak se jednotlivá vlákna přetahují o dostupné instance. Můžeš měnit počet vláken a kapacitu fondu a sledovat, jak se bude měnit činnost programu.

Výpis 12.4: Třída `FondTest`

```
package rup.česky.vzory._12_fond;

import java.util.Random;
import static rup.česky.vzory._00_společné.SynchroTisk.tiskni;


/*******************************************************************************
```

```
 * Třída FondTest testuje fungování fondu vláken.
 */
public class FondTest
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Kapacita testovaného fondu. */
    private static final int KAPACITA = 3;

    /** Počet souběžných vláken, která budou fond testovat. */
    private static final int VLÁKEN = 5;

    /** Počet žádostí o instanci, které každé vlákno realizuje. */
    private static final int ŽÁDOSTÍ = 5;

    /** Testovaný fond o zadané kapacitě. */
    private static Fond<FondTest.Instance> f =
        new Fond<FondTest.Instance>( new
            ITovárna<FondTest.Instance>() {
                public FondTest.Instance newInstance() {return new Instance();}
            },
            KAPACITA );


//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================================

    private static int počet = 0;   //Uchovává počet vytvořených instancí


//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================================

    private final int index = ++počet;  //Rodné číslo dané instance


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /* Soukromý konstruktor zabrání vytvoření instancí či potomků. */
    private FondTest() {}


//== NESOUKROMÉ METODY INSTANCÍ ===============================================

    /*************************************************************************
     * Vrátí řetězec s názvem třídy a rodným číslem instance.
     * @return Požadovaný řetězec
     */
    public String toString() { return "FondTest_" + index; }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ==================================================

    /*************************************************************************
     * Instance třídy <code>Instance</code> jsou oněmi instancemi,
     * které budou uchovávány ve fondu.
     */
    private static class Instance
    {
```

```
    static int počet = 0;
    int index = ++počet;
    /** @return Název třídy + rodné číslo instance */
    public String toString() { return "Instance_" + index; }
}


/*****************************************************************************
 * Instance třídy Vlákno slouží k testování správné činnosti fondu
 * v několika paralelních vláknech.
 */
private static class Vlákno extends Thread
{
    /** Generátor pro náhodný výběr typu žádosti o instanci. */
    private static final Random rnd = new Random();

    /** Názvy volaných metod. */
    private static final String[] METODA =
                        { "dejPočkám()", "dejIhned()", "dejVolnou()" };

    /** Rodné číslo vlákna. */
    private final int pořadí;

    /** Vytvoření pojmenovaného vlákna. */
    Vlákno( int pořadí )  {
        super( "Vlákno_" + pořadí );
        this.pořadí = pořadí;
    }

    /*************************************************************************
     * Testovací metoda, která náhodným způsobem žádá fond
     * o v něm uložené instance a vypisuje na standardní výstup
     * úspěšnost svých žádostí.
     */
    public void run()
    {
        int i = 0;
        while( ++i <= ŽÁDOSTÍ ) {
            String txt = pořadí + ". vlákno, " + i + ". pokus - vlákno ";
            Instance instance;  //Proměnná pro požadovanou instanci
            int metoda = rnd.nextInt( 3 );  //Výběr způsobu žádosti
            tiskni( txt + "se snaží získat instanci metodou " +
                    METODA[metoda] );
            switch( metoda )
            {
                case 0:
                    //Není-li instance k dispozici, počká, až bude
                    instance = f.dejPočkám();
                    break;

                case 1:
                    try {
                        //Není-li instance k dispozici, nechá vytvořit
                        //novou, při zaplněné kapacitě vyhodí výjimku
                        instance = f.dejIhned();
                    }catch( Exception e ) {
                        tiskni( txt +
```

```
                            "nekorektně požádalo o vytvoření další instance");
                        instance = null;
                    }
                    break;

                case 2:
                default:
                    for(;;) {
                        //Není-li instance k dispozici, vrátí null
                        instance = f.dejVolnou();
                        if( instance != null )
                            break;  //Mám instanci => končím
                        tiskni( txt + "neuspělo se svým požadavkem "
                                + METODA[metoda] );
                        //Instance nebyla k dispozici,
                        //za chvíli požádám znovu
                        spánek( 200, 0 );
                    }
            }
            tiskni( txt + "získalo instanci " + instance );
            spánek( 300, 700 );
            if( instance != null )  {
                f.vracímInstanci( instance );
                tiskni( txt + "vrátilo instanci " + instance );
            }
            spánek( 300, 700 );
        }
    }

    /** Uspí aktivní vlákno na zadanou dobu. */
    private static void spánek( int pevná, int volná ) {
        try{
            if( volná == 0 )
                Thread.sleep( pevná );
            else
                Thread.sleep( pevná + rnd.nextInt( volná ) );
        }catch( InterruptedException ie ){}
    }
}

//== TESTY ====================================================================

    /*****************************************************************************
     * Testovací metoda spustí zadaný počet testovacích vláken.
     */
    public static void test()
    {
        for( int i=1;   i <= VLÁKEN;   i++ ) {
            Vlákno vlákno = new Vlákno( i );
            vlákno.start();
            tiskni(i + ". vlákno spuštěno" );
        }
    }
    /** @param args Parametry příkazového řádku - nepoužívané */
    public static void main( String[] args )  {  test();  }
}
```

210. Co se stane, když někdo vrátí instanci do fondu, ale ve skutečnosti ji bude dále používat?

Co se stane při použití vrácené instance

To jsem v zájmu co nejjednoduššího řešení nezabezpečoval. Při takto definovaném fondu si takové věci musí ohlídat programátor sám. Řešitelné to ale samozřejmě je. Jednou z možností je zabalit předávanou instanci do bezpečnostní obálky a předávat obálku. Vzpomeň si na to, až budeme probírat návrhový vzor *Zástupce* (nedočkavému čtenáři prozradím, že to bude v kapitole *Pod ruce mi neuvidíš (Zástupce – Proxy)* na straně 189). Tam si také ukážeme druhou variantu správy volných a používaných objektů.

Příklad: Molekuly

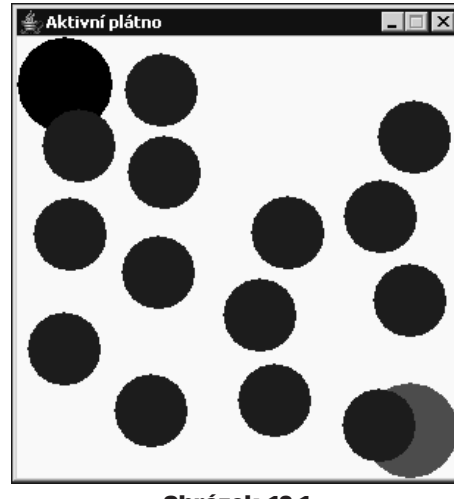
211. V testovacím příkladu jsi používal fond omezené velikosti. Zmiňoval ses ale také o fondu neomezené velikosti. Jak takový fond zabezpečí, aby se nevyčerpala paměť?

To nezabezpečuje fond. To musí zaručovat sama aplikace, která z principu nebude potřebovat současně více než nějaký rozumný, byť předem neznámý počet instancí.

212. Mohl bys mi to ukázat na příkladu?

Princip příkladu

Mám pro tebe třídu `BrownůvPohyb`, která simuluje Brownův pohyb molekul. Představ si nějaký prostor, v němž se náhodně pohybují molekuly (tímto prostorem je naše plátno). V jeho jednom rohu je vývěva, která odsává molekuly, jež se dostanou do jejího dosahu, a v jiném rohu je porodnice, která ve chvíli, kdy se v její zóně nenachází žádná molekula, „porodí“ nějakou novou.



Obrázek 12.1
Obraz průběhu simulace Brownova pohybu molekul

V příkladu neustále vnikají nové molekuly a jiné zanikají. Z podstaty problému je ale jasné, že nemůže existovat více molekul, než se do daného prostoru vejde. Kolik to

bude, to záleží na velikosti prostoru a molekul. Programátor se ale o tento počet nemusí starat. Prostě pro ně použije fond s neomezenou velikostí a nechá na něm, aby vytvořil tolik instancí, kolik bude potřeba.

Definici programu si můžeš prohlédnout ve výpisu 12.5. Je trochu delší, protože na počátku obsahuje poměrně bohatou sadu konstant a uvnitř třídy je navíc definována vnitřní třída `Molekula`, jejíž instance představují molekuly, jejichž pohyb program simuluje.

Pokaždé když program obdrží od fondu instanci molekuly nebo když naopak nějakou instanci do fondu vrátí, vypíše na standardní výstup zprávu. Spusť si tento program a chvíli sleduj standardní výstup. Uvidíš, jak na počátku vznikají nové a nové instance, ale po chvíli se nové instance přestanou objevovat, protože ve fondu bude vždy k dispozici nějaká z dříve vytvořených instancí.

Výpis 12.5: Třída `BrownůvPohyb`

```
package rup.česky.vzory._12_fond;

import java.util.LinkedHashSet;
import java.util.Random;
import java.util.Set;
import java.util.Timer;
import java.util.TimerTask;
import java.util.Iterator;

import rup.česky.tvary.Barva;
import rup.česky.tvary.Kruh;


/** Společná instance spravující zobrazování všech objektů. */
import static rup.česky.tvary.SprávcePlátna.SP;

/*******************************************************************************
 * Třída BrownůvPohyb simuluje náhodný pohyb molekul v prostoru,
 * v jehož jednom rohu je vývěva, která odsává molekuly, jež se dostanou
 * do jejího dosahu, a v jiném rohu je porodnice, která ve chvíli,
 * kdy se v její zóně nenachází žádná molekula, "porodí" nějakou novou.
 */
public class BrownůvPohyb extends TimerTask
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Společný průměr všech molekul. */
    private static final int PRŮMĚR = 50;

    /** Společný průměr všech molekul. */
    private static final int ŠÍŘKA = 6;

    /** Společný průměr všech molekul. */
    private static final int VÝŠKA = 6;

    //Statický konstruktor inicializuje plátno, na němž se vše bude dít
    static {
        SP.setKrokRozměr( PRŮMĚR, ŠÍŘKA, VÝŠKA );
        SP.setMřížka( false );
```

```
    }

    /** Frekvence překreslování plátna s molekulami. */
    public static final int FREKVENCE = 50;

    /** Maximální povolená rychlost molekuly,
     * tj. kolik bodů může molekula urazit mezi dvěma snímky. */
    private static double MAX_RYCHLOST = 10 ;

    /** Maximální povolené souřadnice molekuly. */
    private static final int X_MAX = PRŮMĚR * (ŠÍŘKA - 1),
                             Y_MAX = PRŮMĚR * (VÝŠKA - 1);

    /** Kolikrát budou průměry vývěvy a porodnice větší než průměr molekul. */
    private static final double POMĚR = 1.3;

    /** Průměr vývěvy a porodnice. */
    private static final int PRŮMĚR_VP = (int)(POMĚR * PRŮMĚR);

    /** Rozdíl průměrů molekuly a vývěvy, resp. porodnice
     *  určuje pozici odsávané i porozené molekuly */
    private static final int ROZDÍL_PRŮMĚRŮ = PRŮMĚR_VP - PRŮMĚR;

    /** Souřadnice generátoru molekul. */
    private static final int X_PORODNICE = X_MAX + PRŮMĚR - PRŮMĚR_VP;
    private static final int Y_PORODNICE = Y_MAX + PRŮMĚR - PRŮMĚR_VP;

    /** Maximální souřadnice molekuly nazasahující do generátoru. */
    private static final int X_PŘEKÁŽÍ = X_PORODNICE - PRŮMĚR;
    private static final int Y_PŘEKÁŽÍ = Y_PORODNICE - PRŮMĚR;

    /** Množina všech dosud vygenerovaných molekul.
     *  Atribut není private, aby k němu mohl přistupovat animátor. */
    private static final Set<Molekula> molekuly = new LinkedHashSet<Molekula>();

    /** Společný generátor náhodných čísel. */
    private static final Random rnd = new Random();


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final Kruh vývěva    = new Kruh( 0, 0, PRŮMĚR_VP, Barva.ČERNÁ );
    private final Kruh porodnice = new Kruh( X_PORODNICE, Y_PORODNICE,
                                             PRŮMĚR_VP, Barva.ČERVENÁ );
    private final Fond<Molekula> fond = new Fond<Molekula>( new
        ITovárna<Molekula>() {
            public Molekula newInstance() {
                return new Molekula();
            }
        }
    );


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /**************************************************************************
     * Vytvoří nový animátor molekul, který bude se zadanou frekvencí
```

```
     * žádat molekuly o aktualizaci jejich pozice
     * a následně nechávat překreslit plátno.
     * Součástí konstrukce animátoru je i nakreslení jímky v levém horním
     * a generátoru molekul v pravém dolním rohu plátna.
     */
    public BrownůvPohyb()
    {
        SP.přidejDospod( vývěva );
        SP.přidejDospod( porodnice );
        new Timer().schedule( this, 0, 1000/FREKVENCE );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Metoda požadovaná třídou TimerTask.
     * Bude spouštěna časovačem s požadovanou frekvencí.
     * Tato metoda má na starosti vlastní animaci včetně
     * "požírání molekul" jímkou a generace nových molekul generátorem.
     */
    public void run()
    {
        SP.nekresli();
            for( Molekula m : molekuly )
                m.popojeď();
            vývěva();
            porodnice();
        SP.vraťKresli();
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /**************************************************************************
     * Metoda zkontroluje, zda se nějaká molekula nedostala do dosahu vývěvy,
     * a pokud ano, tak ji "vysaje".
     */
    private void vývěva()
    {
        for( Iterator<Molekula> it = molekuly.iterator();
             it.hasNext();  ) {
           Molekula m = it.next();
           if( (m.getX() <= ROZDÍL_PRŮMĚRŮ) && (m.getY() <= ROZDÍL_PRŮMĚRŮ) ) {
               //Molekula se dostala do oblasti působnosti vývěvy
               SP.odstraň( m );          //Odstraň molekulu z plátna
               it.remove();              //a z množiny animovaných molekul
               fond.vracímInstanci( m ); //Vrať ji fondu
               System.out.println("-- " + m + " vrácena do fondu");
               break;   //Do dosahu vývěvy se může dostat jen jedna molekula
           }
       }
    }


    /**************************************************************************
     * Metoda zjistí, jestli je v oblasti porodnice volno (tj. nepřekáží zde
```

```
     * žádná molekula), a pokud ne, vyšle do prostoru novou molekulu.
     */
    private void porodnice()
    {
        boolean nepřekáží = true;
        for( Molekula m : molekuly ) {
            if( (m.getX() > X_PŘEKÁŽÍ)  &&  (m.getY() > Y_PŘEKÁŽÍ) ) {
                nepřekáží = false;
                break;
            }
        }
        if( nepřekáží ) { //Nikdo nepřekážel
            Molekula m = fond.dejIhned();   //Vyzvedne z fondu molekulu
            System.out.println("+++ " + m + " vyzvednuta z fondu");
            //Umístí ji na svoji pozici, jako by ji zde porodila.
            m.setPozice( (X_PORODNICE + ROZDÍL_PRŮMĚRŮ/2.),
                         (Y_PORODNICE + ROZDÍL_PRŮMĚRŮ/2.) );
            molekuly.add( m );  //Přidá ji mezi animované
            SP.přidej( m );     //Nechá ji zobrazit
        }
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    /**************************************************************************
     * Třída AnimátorMolekul definuje objekt, který zabezpečí
     * animaci molekul uložených ve statickém atributu molekuly
     * třídy Molekula_13e.
     * Tato verze navíc umístí do rohu oblasti s molekulami výlevku,
     * která vcucne každou molekulu, jež do ní cele vstoupí.
     */
    private static class Molekula extends Kruh
    {
    //== PROMĚNNÉ ATRIBUTY TŘÍDY ================================================

        /** Počet dosud vytvořených instancí. */
        private static int mpočet = 0;


    //== KONSTANTNÍ ATRIBUTY INSTANCÍ ===========================================

        /** Pořadí vytvoření dané instance v rámci třídy. */
        private final int mpořadí = ++mpočet;

        /** Formátovaný název molekuly. */
        private final String název = "M" + ((mpořadí <  10) ? "__" :
                                            (mpořadí < 100) ? "_"  :
                                                              ""   ) + mpořadí;

    //== PROMĚNNÉ ATRIBUTY INSTANCÍ =============================================

        /** Složky aktuální pozice molekuly -
         *  musí mít vlastní, protože je potřebuje mít double. */
        private double x=0, y=0;

        /** Složky aktuální rychlosti molekuly. */
```

```
    private double rx, ry;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /**************************************************************************
     * Vytvoří molekulu s náhodnou rychlostí.
     * Na pozici nezáleží, porodnice si ji stejně umístí.
     */
    public Molekula()
    {
        super( 0, 0, PRŮMĚR );
        this.rx = (rnd.nextDouble() - 0.5) * MAX_RYCHLOST;
        this.ry = (rnd.nextDouble() - 0.5) * MAX_RYCHLOST;
    }


//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /**************************************************************************
     * Posune molekulu do nové pozice definované její současnou pozicí
     * a snímkovou rychlostí. V nové pozici zkontroluje, jestli molekula
     * nenarazila do okraje plátna či některé jiné molekuly. Pokud ano,
     * tak posun zruší a upraví rychlost zúčastněných tak, aby simulovaly
     * dokonalý odraz.
     */
    public void popojeď()
    {
        double xn = x + rx;     //Plánovaná nová vodorovná souřadnice
        double yn = y + ry;     //Plánovaná nová svislá souřadnice
        boolean odraz = false;  //Předpokládáme, že se neodrazí

        //Test odrazu od zdi
        if( (xn <= 0) || (X_MAX <= xn) )   {  rx =-rx;  odraz = true; }
        if( (yn <= 0) || (Y_MAX <= yn) )   {  ry =-ry;  odraz = true; }

        //Testy možných odrazů od ostatních molekul
        for( Molekula m : molekuly )
        {
            if( (m != this)  &&  //Sama od sebe se neodrazí
                (Math.hypot( (xn - m.x), (yn - m.y) )  <  PRŮMĚR) )
            {
                double p;   //Molekuly si prohodí rychlosti
                p = rx;   rx = m.rx;   m.rx = p;
                p = ry;   ry = m.ry;   m.ry = p;
                odraz = true;
            }
        }
        //Pokud se molekula odrazila, její pozice se nezmění
        //(spokojíme se pouze s nastavenou změnou rychlosti)
        if( !odraz )
            setPozice( xn, yn );
    }

    /**************************************************************************
     * @return Název instance
     */
    public String toString()
```

```
        {
            return název;
        }

        /*************************************************************************
         * Nastaví souřadnice ve dvojité přesnosti a posune instanci
         * @param x  Nastavovaná vodorovná pozice
         * @param y  Nastavovaná svislá pozice
         */
        void setPozice( double x, double y )
        {
            super.setPozice( (int)x, (int)y );
            this.x = x;
            this.y = y;
        }

        /** {@inheritDoc} Sebeobrana proti opomenutí. */
        public void setPozice( int x, int y ) {
            throw new UnsupportedOperationException(
                "\nPozici je třeba nastavovat s parametry typu double" );
        }
    }


//== TESTY A METODA MAIN ======================================================

    /*************************************************************************
     * Testovací program - vygeneruje novou sadu molekul a spustí animaci
     * jejich pohybu.
     */
    public static void test()
    {
        new BrownůvPohyb();
    }
    /** @param args Parametry příkazového řádku - nepoužívané */
    public static void main( String[] args ) { test(); }
}
```

213. Tak tento program byl přece jenom trochu delší, a budu si jej proto muset pomaličku a v klidu projít.

Měl bych pro tebe ještě jednu ukázku definice fondu, i když trochu nestandardního. Je součástí závěrečného příkladu ke kapitole *Horký brambor (Řetěz odpovědnosti – Chain of Responsibility)*, začínající na straně 361, a její zdrojový kód najdeš ve výpisu 27.3 na straně 369.

Shrnutí – co jsme se naučili

- *Fond* použijeme tehdy, je-li výhodnější použitou a nyní již nepotřebnou instanci uschovat a v případě potřeby ji opět „reinkarnovat“ (oživit), než ji zahodit a posléze vytvářet novou.

- Předchozí podmínku splňují instance, jejichž vytvoření je nějakým způsobem „drahé“ – např. zabírá mnoho času.
- *Fond* použijeme také tehdy, potřebujeme-li počet vytvářených instancí omezit, tj. v situacích, kdy je výhodnější, aby vlákno na uvolnění právě používané instance počkalo, než aby se pro něj vytvářela nová.
- *Fond* můžeme definovat jak s omezenou, tak s neomezenou kapacitou.
- Při neomezené kapacitě fondu musí omezení vyplývat z povahy aplikace.
- *Fond* nepatří mezi vzory z GoF.

KAPITOLA 13

Příliš mnoho instancí (Muší váha – Flyweight)

- **Účel**
- **Implementace**
- **Příklad – hra Diamanty**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Vzor označovaný jako *Muší váha* používáme ve chvíli, kdy řešení problému vyžaduje vytvoření značného množství objektů. Ukazuje, jak je možné toto množství snížit tím, že místo skupiny objektů s podobnými charakteristikami použijeme jeden sdílený objekt tak, že z něj vyjmeme část jeho stavu, která odlišuje jednotlivé zastupované objekty, a volané metody se tyto informace v případě potřeby dozvědí z vnějšího zdroje prostřednictvím svých parametrů.

[1] **Definice v GoF:** Use sharing to support large numbers of fine-grained objects efficiently. – Díky sdílení umožňuje efektivní podporu velkého množství drobných objektů.

S výjimkou jedináčka jsme doposud hovořili o vzorech, které nebyly uvedeny v GoF, a řada puristů je proto za opravdové návrhové vzory nepovažuje. Od této chvíle se budeme zabývat téměř výlučně návrhovými vzory popsanými v GoF.

Účel

214. Probrali jsme nula instancí, jednu instanci, konečný počet instancí, takže teď zbývá nekonečný počet instancí.

Příliš mnoho instancí

Nejsi tak daleko od pravdy. Teď bych ti chtěl ukázat, jak lze postupovat, pokud by při klasickém postupu vzniklo příliš mnoho instancí, přičemž ono „příliš mnoho" může být jednou sto tisíc a podruhé jenom 100 nebo ještě méně.

215. Nadhoď zase nějaký příklad, ať se lépe orientuji.

Příklad v GoF

V GoF je použití tohoto návrhového vzoru demonstrováno na příkladu formátovaného textu. V něm by si každé písmeno mělo nést informaci nejenom o svém kódu a umístění v textu, ale také o svém písmu, jeho velikosti, řezu (obyčejné, polotučné, tučné, kurziva, …), barvě a dalších charakteristikách, které ovlivňují jeho vzhled.

Implementace

216. Kdybych si tohle měl o každém písmenu pamatovat, tak bych asi u delších dokumentů s pamětí nevystačil. Jak z toho tedy mohu vybruslit?

Zavedení vnějšího a vnitřního stavu

Rozdělíš informace do dvou skupin: na ty, které se pro každou instanci liší, a na ty, které mohou jednotlivé instance mezi sebou sdílet. Místo instancí pro každý jednotlivý výskyt daného objektu (mohli bychom je nazývat *virtuální instance*) pak definuješ pouze instance charakterizované jednotlivými hodnotami sdílených informací a každá z nich bude zastupovat několik objektů (virtuálních instancí) současně.

Informace z první skupiny, tj. informace, jimiž se jednotlivé instance liší, budeš získávat z okolního programu a předávat je metodám skutečně vytvořených instancí např. jako parametry.

217. Obávám se, že jsem opět vůbec nic nepochopil. Zkus to vysvětlit ještě jednou na nějakém příkladu.

Jak se používá sdílený objekt

Zůstal bych na chvilku ještě u těch znaků. V dokumentu bývá velké množství znaků, ale všechny znaky jsou vybírány z poměrně malé množiny znaků v kódové tabulce. Můžeme proto mít jeden objekt pro každý znak z kódové tabulky – kód znaku bude tedy představovat vnitřní stav daného objektu.

Ve chvíli, kdy je třeba některý znak vykreslit, požádám objekt odpovídající tomuto znaku, aby znak vykreslil. Přitom mu prozradíme, na které pozici jej má vykreslit – pozice zobrazovaného znaku je jeho vnější stav.

Jakmile je znak zobrazen, nebudeme daný objekt již pro tento konkrétní znak potřebovat, a můžeme proto objekt odložit a použít jej pro další znak se stejným kódem, který se však bude nejspíš zobrazovat někde úplně jinde.[1]

Popis typické implementace

218. Mám to tedy chápat tak, že taková instance je zásobárna informací o nějakém objektu, ale že uchovává méně informací, než bývá potřeba. Zbylé informace jí pak předáme až tehdy, když jí posíláme nějakou zprávu, a proto voláme její metodu.

Vnější a vnitřní stav v příkladu

Lépe bych to neřekl. Učené knihy hovoří o vnějším a vnitřním stavu dané instance. Vnitřní stav je to, co si o sobě daná instance pamatuje sama – v našem případě to mohl být např. kód znaku, jeho podoba, jeho rozměry apod. Vnější stav je pak charakterizován informacemi, které si o sobě instance nepamatuje a dozvídá se je na poslední chvíli. V našem posledním příkladu to např. mohla být pozice znaku.

Objekty zastupující celou skupinu objektů

219. Takže při použití vzoru *Muší váha* vlastně definujeme místo původních skupin objektů s nějakou společnou vlastností pouze takové jejich zástupce, z nichž každý zastupuje celou skupinu. Nemusíme pak vytvářet instance všech objektů dané skupiny.

Úprava metod

Říkáš to poměrně přesně. Pouze bych nepoužíval termín *zástupci*, aby se nám nepletli s objekty, o nichž hovoří návrhový vzor *Zástupce*. Říkejme jim reprezentanti skupiny objektů.

Zvláštnosti metod reprezentantů

Tito reprezentanti musí definovat své metody tak, aby jim bylo možno v parametrech předat informace, které navzájem odliší jednotlivé zastupované instance. Zastupující instance pak bude vědět vše, aby se mohla chovat přesně tak, jak by se měla chovat zastupovaná instance.

220. Nekomplikuje se vývoj aplikace tím, že reprezentanti musí své metody definovat jinak, než by je definovaly reprezentované objekty?

Odchylka rozhraní objektu a jeho reprezentace

Záleží na tom, kdy se rozhodneš reprezentanty zavést. Bude-li to hned v počátku návrhu, pak budeš všude hned od počátku počítat s rozhraním oněch reprezentantů a žádné komplikace by vzniknout neměly.

Pokud ale to, že objektů bude na tvé možnosti příliš mnoho, zjistíš až uprostřed prací na aplikaci a budeš vzor *Muší váha* aplikovat až dodatečně, vyjde to dráž, protože budeš muset měnit rozhraní, a to je vždy nanejvýš nepříjemné.

Příklad – hra Diamanty

221. Začíná mi to být jasné. Teď by to chtělo ještě nějaký příklad, který si mohu doopravdy vyzkoušet.

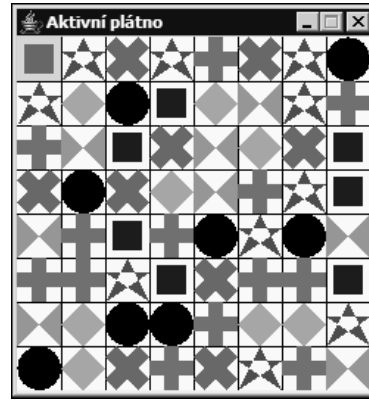
Ukážu ti použití tohoto návrhového vzoru na jednodušším příkladu, než je výše zmiňovaný textový procesor. Dopředu bych chtěl jen upozornit, že tento příklad není

[1] Setkal jsem se s několika lidmi, kteří považovali tento příklad za špatný, protože se domnívají, že daný problém ani jinak slušně řešit nejde. To vás ale napadne až ve chvíli, kdy máte OOP „pod kůží“. Programátoři, kteří si OOP ještě neosvojili, řešívali tento problém většinou tak, že jednotlivé znaky nedefinovali jako objekty.

nutno řešit pomocí vzoru *Muší váha*, ale na tomto příkladu si můžeš vyzkoušet, jak takový vzor aplikovat v situaci, kdy by jiné řešení bylo vzhledem k dostupným zdrojům příliš náročné.

Popis hry Diamanty

Kdysi jsme s dětmi v kroužku programovali hru nazvanou diamanty. Její podstata spočívala v tom, že šachovnice o rozměru 8 × 8 polí byla zaplněna náhodně rozmístěnými obrazci představujícími pomyslné diamanty sedmi druhů (viz obr. 13.1). Cílem hráče bylo prohodit diamanty na dvou vodorovně nebo svisle sousedících polích tak, aby jeden či oba prohazované diamanty utvořily po prohození se svými sousedy trojici (případně čtveřici či pětici) vodorovně či svisle sousedících diamantů. Sousedící shodné diamanty pak byly z šachovnice odstraněny (zvedly bodový zisk hráče). Na uvolněné místo se protlačili jejich horní sousedé, na jejich místo jejich sousedé a tak dále. Uvolněná místa při horním okraji šachovnice pak byla zaplněna náhodně vybranými diamanty.



Obrázek 13.1
Deska s obrazci představujícími různé druhy diamantů

222. Princip hry chápu, ale nevidím tam ty mouchy.

Možné způsoby reprezentace diamantů

Muší váhu jsme aplikovali na ty diamanty. Diamanty na šachovnici jsme totiž mohli reprezentovat dvěma způsoby:

- Každý diamant představuje samostatný objekt, který ví, jak vypadá (tj. jak se má nakreslit). Když jej máme umístit na šachovnici, vytvoříme novou instanci. Když jej z šachovnice odstraňujeme, předáme instanci správci paměti, který ji v příhodnou dobu definitivně zruší.
- Využijeme vzor *Muší váha* a budeme všechny diamanty jednoho druhu reprezentovat stejnou instancí, které budeme průběžně říkat, kde se má nakreslit. Stačí nám proto pouze tolik objektů, kolik je různých diamantů.

223. Připadá mi, že jsi v tom prvním řešení instancemi příliš hýřil – vždyť jsi mohl použít fond, o němž jsme před chvílí hovořili.

Proč není nejlepším řešením *Fond*

Mohl bych sice vytvořit fond diamantů, ale není to tak jednoduché, jak by se na první pohled zdálo. Naše šachovnice má 8 × 8 = 64 polí. Protože teoreticky může vzniknout takové uspořádání, kdy téměř 3/4 šachovnice budou pokryty diamanty stejného

druhu, musel bych mít fond, který by byl schopen uchovávat od každého druhu diamantů potřebný počet instancí. Ve srovnání s jinými aplikacemi to sice není nijak závratný počet, ale pro naši potřebu je to přece jenom zbytečně mnoho.

224. Chceš tedy říct, že jsi místo (3/4) * 64 * 7 = 336 instancí vystačil pouze se sedmi.

Zapomněl jsi započítat ještě instanci představující prázdné políčko, ale jinak máš vcelku pravdu: místo tisíců průběžně vznikajících a zanikajících instancí nebo zásobníku na přibližně 400 instancí jsme vystačili s osmi instancemi diamantů.

225. 400 instancí sice není tak moc, ale beru to jako příklad postupu, který by se mi mohl hodit v jiných aplikacích. Teď mi ukaž, jak jste to dělali.

Zásady implementace diamantů

Využili jsme toho, že kdybychom definovali každý diamant jako samostatnou instanci, tak by se jednotlivé instance diamantů stejného druhu lišily pouze svými souřadnicemi. Rozhodli jsme se proto ponechat informaci o tvaru diamantu jako jeho *vnitřní stav*, který si každý diamant pamatuje, a informaci o jeho souřadnicích zavést jako jeho *vnější stav*, který mu okolní objekty předávají při zasílání požadavku na jeho překreslení. Kdo bude chtít po diamantu, aby se nakreslil, bude mu muset předat kromě kreslítka i souřadnice, na nichž se má daný diamant nakreslit.

226. Aha. Pak jste opravdu mohli definovat pro všechny diamanty stejného druhu jejich společného reprezentanta, který na požádání nakreslil každého z nich. Můžeš mi ukázat implementaci třídy diamantů?

Spojení dvou vzorů

Při implementaci třídy `Diamant` jsme spojili dohromady dva návrhové vzory: *Muší váhu* a *Výčtový typ*. Protože jsme znali přesný počet budoucích instancí, a dokonce jsme věděli, které to budou, definovali jsme je jako instance výčtového typu. A protože se jednotlivé instance lišily vlastně pouze tím, jak se kreslily, tj. lišily se jednou použitou metodou, použili jsme funkční výčtový typ, v němž si každá instance definuje svoji vlastní verzi kreslicí metody.

Definované metody

Ve výčtovém typu diamantů jsme definovali veřejnou abstraktní metodu `nakresli(int,int,Kreslítko)`, kterou každá z jeho instancí (tj. každý druh diamantu) implementuje po svém. Jak vidíš, oproti standardním kresleným objektům, kterým žadatel o jejich překreslení dodá pouze kreslítko, jímž se nakreslí (vše ostatní si musí objekty pamatovat), musí žadatel o překreslení diamantů předat překreslující metodě nejenom kreslítko, ale také souřadnice právě kresleného objektu. V průběhu překreslování celé šachovnice bude proto kreslicí metoda každého druhu diamantů volána vícekrát, pokaždé s jinými souřadnicemi.

227. Začíná mi to být jasné. Teď už bych si takový typ skoro troufal definovat sám. Pro jistotu ale přece jenom požádám o zdrojový kód tebe.

Definici třídy `Diamant` si můžeš prohlédnout ve výpisu 13.1.

Výpis 13.1: Definice třídy `Diamant`

```
package rup.česky.vzory._13_muší_váha.diamanty;

import rup.česky.tvary.Barva;
import rup.česky.tvary.Kreslítko;
```

```
import java.util.Random;
import static rup.česky.tvary.SprávcePlátna.SP;


/*******************************************************************************
 * Instance třídy diamant představují jednotlivé typy diamantů,
 * které se mohou vyskytovat na hrací desce. Umějí jenom jediné:
 * na požádání se namalovat na zadaném políčku.
 */
public enum Diamant
{
//== HODNOTY VÝČTOVÉHO TYPU ====================================================
/** Prázdné políčko. */
    NIC {
        public void nakresli( int x, int y, Kreslítko k ) {}
    },
    /** Modrý čtverec. */
    ČTVEREC {
        public void nakresli( int x, int y, Kreslítko k ) {
            int M1 = MODUL / 6;
            int M2 = MODUL - 2*M1;
            k.vyplňRám( x+M1, y+M1, M2, M2, Barva.MODRÁ );
        }
    },
    /** Červená hvězda s průhledným středem. */
    HVĚZDA {
        public void nakresli( int x, int y, Kreslítko k ) {
            int M  = MODUL;
            int M2 = MODUL / 2;
            int M3 = MODUL / 3;
            k.vyplňPolygon(
                new int[] { x+M2, x+M, x,    x+M,  x,   x+M2 },
                new int[] { y,    y+M, y+M3, y+M3, y+M, y    },
                Barva.ČERVENÁ );
        }
    },
    /** Černý kruh. */
    KOLO {
        public void nakresli( int x, int y, Kreslítko k ) {
            k.vyplňOvál( x, y, MODUL, MODUL, Barva.ČERNÁ );
        }
    },
    /** Fialové "X". */
    X {
        public void nakresli( int x, int y, Kreslítko k ) {
            int M1 =   MODUL / 4;
            int M2 =   MODUL / 2;
            int M3 = 3*MODUL / 4;
            int M4 =   MODUL;
            k.vyplňPolygon(
                new int[] { x+M1, x+M2, x+M3, x+M4, x+M3, x+M4,
                            x+M3, x+M2, x+M1, x,    x+M1, x    },
                new int[] { y,    y+M1, y,    y+M1, y+M2, y+M3,
                            y+M4, y+M3, y+M4, y+M3, y+M2, y+M1 },
                Barva.FIALOVÁ );
        }
    },
```

```
    /** Šedomodré "+". */
    PLUS {
        public void nakresli( int x, int y, Kreslítko k ) {
            int M1 =   MODUL / 3;
            int M2 = 2*MODUL / 3;
            int M3 =   MODUL;
            k.vyplňPolygon(
                new int[] { x+M1, x+M2, x+M2, x+M3, x+M3, x+M2,
                            x+M2, x+M1, x+M1, x,    x,    x+M1 },
                new int[] { y,    y,    y+M1, y+M1, y+M2, y+M2,
                            y+M3, y+M3, y+M2, y+M2, y+M1, y+M1 },
                Barva.OCELOVÁ );
        }
    },
    /** Zlatý (okrový) kosočtverec. */
    KOSOČTVEREC {
        public void nakresli( int x, int y, Kreslítko k ) {
            int M1 =   MODUL / 2;
            int M2 =   MODUL;
            k.vyplňPolygon(
                new int[] { x+M1, x+M2, x+M1, x    },
                new int[] { y,    y+M1, y+M2, y+M1 },
                Barva.ZLATÁ );
        }
    },
    /** Dva zelené rovnoramenné trojúhelníky obrácené k sobě vrcholy. */
    MOTÝL {
        public void nakresli( int x, int y, Kreslítko k ) {
            int M1 =   MODUL / 2;
            int M2 =   MODUL;
            k.vyplňPolygon(
                new int[] { x, x+M1, x+M2, x+M2, x+M1, x  },
                new int[] { y, y+M1, y,    y+M2, y+M1, y+M2  },
                Barva.ZELENÁ );
        }
    };

//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final Diamant[] DRUH  = Diamant.values();

    /** Počet druhů diamantů včetně prázdného. */
    private static final int DRUHŮ = DRUH.length;

    private static int MODUL = SP.getKrok();

    /** Generátor náhodných čísel. */
    private static final Random RND;
    static {
        if( Hra.DEBUG > 0 )
            RND = new Random(Hra.RND_SEED); //zadané prověřené semeno
        else
            RND = new Random(); //náhodné semeno odvozené z času
    }


//== OSTATNÍ METODY TŘÍDY =====================================================
```

```
    /*****************************************************************************
     * Vrátí náhodný index dalšího diamantu, přičemž nikdy nevrací index
     * diamantu NIC.
     */
    public static int dalšíIndex() {
        return RND.nextInt(DRUHŮ-1)+1;
    }

    /*****************************************************************************
     * Vrátí náhodný diamant, přičemž nikdy nevrací diamant NIC.
     */
    public static Diamant další() {
        return DRUH[ dalšíIndex() ];
    }

    /*****************************************************************************
     * Nakreslí dodaným kreslítkem zadaný druh diamantu
     * na políčko o zadaných políčkových souřadnicích.
     */
    public static void nakresli( int index, int xp, int yp, Kreslítko k )
    {
        int x = MODUL * xp;
        int y = MODUL * yp;
        nakresliB( index, x, y, k );
    }

    /*****************************************************************************
     * Nakreslí zadaný druh diamantu dodaným kreslítkem
     * na zadané bodové souřadnice.
     */
    public static void nakresliB( int index, int xb, int yb, Kreslítko k  )
    {
        Diamant d = DRUH[ index ];
        d.nakresli( xb, yb, k );
    }

//== ABSTRAKTNÍ METODY ==========================================================

    /*****************************************************************************
     * Nakreslí diamant dodaným kreslítkem na zadané bodové souřadnice.
     *
     * @param xb    Vodorovná bodová souřadnice
     * @param yb    Svislá bodová souřadnice
     * @param k     Kreslítko, pomocí nějž se má diamant nakreslit.

     */
    abstract void nakresli( int x, int y, Kreslítko k );
}
```

228. Vidím, že jste diamanty mohli kreslit dvěma způsoby: buď jste oslovili přímo diamant nebo jste mohli použít metodu třídy.

Různé přístupy, různá řešení

My jsme nakonec úlohu řešili několika způsoby. Tento příklad ale není tak významný, abych je tu všechny rozebíral. Máš-li chuť, zkus alternativní přístupy a můžeš je porovnat.

Přiznám se, že jsem se původně chtěl k oběma možnostem vrátit v kapitole *Vyberte si, jak to chcete (Strategie – Strategy)* na straně 415, v níž budeme probírat návrhový vzor *Strategie*, jenž se zabývá právě řešením situací, kdy je možno zadanou úlohu řešit několika způsoby a nelze obecně říct, který je ten nejlepší. Vyčerpal jsem ale stránky povolené nakladatelem, takže případné experimenty zbudou opravdu na tebe.

229. No dobrá. Vrátím se tedy k diamantům. Zatím jsi je definoval. Ještě mi ukaž, jak jsi je používal.

Možnost přímého porovnání místo volání `equals(Object)`

Tím, že jsme místo vytvoření nové instance pro každý jeden diamant na desce zavedli společného reprezentanta pro všechny stejně vypadající diamanty, jsme si ušetřili práci i v jiném směru: při testování, zda dva diamanty na desce vypadají stejně, jsme nemuseli používat metodu `equals(Object)`, ale mohli jsme se omezit na prosté porovnávání odkazů, protože všechny stejně vypadající diamanty byly reprezentovány stejným reprezentantem.

Proč nebylo většinou potřeba znát vnější stav

Jinými slovy: protože tato aplikace se většinou soustředila na vzhled diamantů, umožňovala nám definovat převážnou většinu potřebných operací s diamanty pomocí operací s jejich reprezentanty. Vnější stav, tj. pozici instance na desce, jsme jejímu reprezentantu museli prozradit pouze ve chvíli, kdy jsme po něm chtěli, aby instanci nakreslil. Ve zbytku aplikace jsme vnější stav znát nepotřebovali, protože nás zajímal pouze vzhled, který byl vnitřním stavem instancí, a zástupci jej proto znali.

Proč se algoritmy zefektivnily

Jak se můžeš přesvědčit ve výpisu 13.2 ve třídě `DeskaDia`[1], která měla na starosti modelování situace na desce s diamanty, pouze jsme porovnávali, zařazovali a vyřazovali odkazy na instance reprezentantů. Metodu reprezentanta, které dodáváme v parametru vnější stav, voláme pouze jednou v metodě `nakresli(Kreslítko)`.

Tím, že jsme místo plnohodnotných instancí použili jejich reprezentanty, jsme použité algoritmy zefektivnili. To se ti při vhodné definici reprezentantů může při používání tohoto vzoru také podařit.

Výpis 13.2: Třída `DeskaDia`

```
package rup.česky.vzory._13_muší_váha.diamanty;

import rup.česky.tvary.Kreslítko;

import static rup.česky.tvary.SprávcePlátna.SP;
import static rup.česky.vzory._02_počty.diamanty.Diamant.NIC;

/*******************************************************************************
 * Instance třídy DeskaDia představuje hrací desku, na níž leží diamanty,
 * které se mají vhodným prohazováním odebírat.
 * Instance řídí veškeré operace, které na desce probíhají.
 */
public class DeskaDia implements IDeska
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================
```

[1] **DeskaDia** není deskou hlavního řeckého boha, ale deskou, v níž jsou přímo objektové reprezentace diamantů. Vedle ní se bude hovořit o třídě **DeskaInt**, v níž jsou diamanty reprezentovány pouze svými celočíselnými indexy.

```
    private final Diamanty    diamanty; //Instance řídící celou hru
    private final Diamant[][] deska;    //Deska s diamanty
    private final boolean[][] vyhodit;  //Příznaky odstraňovaných diamantů
    private final Animátor animátor  //Animátor mající na starosti
                              = new Animátor();    //plynulý pohyb


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================


    private int     ROZMĚR;         //Počet řádků, resp. sloupců desky.
    private int     MODUL;          //Velikost jednoho políčka
    private boolean hra = false;    //Indikátor toho, zda se teprve připravuje
        //výchozí podoba desky (false), nebo zda se již doopravdy hraje (true).
        //Ovlivňuje především načítání bodů po odstranění n-tic diamantů.

//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /**************************************************************************
     * Vytvoří novou desku a zaplní ji diamanty tak, aby na ní nebyly
     * trojice v řadě či sloupci sousedících diamantů stejného druhu.
     * Deska je reprezentována maticí zástupců diamantů daného vzhledu.
     *
     * @param diamanty Instance řídící celou aplikaci
     */
    public DeskaDia( Diamanty diamanty )
    {
        this.diamanty = diamanty;
        ROZMĚR = diamanty.getRozměr();
        MODUL  = diamanty.getModul();
        deska   = new Diamant[ROZMĚR][ROZMĚR];
        vyhodit = new boolean[ROZMĚR][ROZMĚR];

        //Zaplň desku náhodně vybranými diamanty
        for( int xp = 0;   xp < ROZMĚR;   xp++ ) {
            for( int yp = 0;   yp < ROZMĚR;   yp++ ) {
               deska [xp][yp] = Diamant.další();
            }
        }
        odstraňTrojice( false );

        hra = true;
        if( Diamanty.LADIM > 0 )
            tiskniDesku( "Zkonstruovaná" );
    }

//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /**************************************************************************
     * Vrátí políčkový rozměr desky, tj. počet polí v řadě, resp. sloupci.
     *
     * @return Rozměr desky
     */
    public int getRozměr()
    {
        return ROZMĚR;
    }
```

```
    /*****************************************************************************
     * Nakreslí desku dodaným kreslítkem.
     *
     * @param kreslítko  Kreslítko, kterým se má deska nakreslit.
     */
    public void nakresli( Kreslítko kreslítko )
    {
        for( int xp=0;   xp < ROZMĚR;   xp++ )
            for( int yp=0;   yp < ROZMĚR;   yp++ ) {
                deska[xp][yp].nakresli( xp*MODUL, yp*MODUL, kreslítko );
            }
        animátor.nakresli( kreslítko );
    }

    /*****************************************************************************
     * Prohodí diamanty na označených dvou polích a zkontroluje výsledek.
     * Nedošlo-li k ustavení trojice, vrátí původní stav.
     * Došlo-li k ustavení trojice, odstraní z desky všechny nově vytvořené
     * n-tice, obsah desky sesype a uvolněná místa nahradí novými diamanty.
     *
     * @param xp1  Vodorovná políčková souřadnice prvního diamantu
     * @param yp1  Svislá políčková souřadnice prvního diamantu
     * @param xp2  Vodorovná políčková souřadnice druhého diamantu
     * @param yp2  Svislá políčková souřadnice druhého diamantu
     */
    public void prohoď( int xp1, int yp1, int xp2, int yp2 )
    {
        if( Diamanty.LADIM > 0 )
            tiskniDesku( "Před prohozením" );
        Diamant první = deska[xp1][yp1];
        Diamant druhý = deska[xp2][yp2];
        deska[xp1][yp1] = druhý;
        deska[xp2][yp2] = první;
        if( Diamanty.LADIM > 0 )
            tiskniDesku( "Po prohození" );
        SP.překresli();
        if( odstraňTrojice( true ) == 0 ) {
            deska[xp1][yp1] = první;
            deska[xp2][yp2] = druhý;
        }
        SP.překresli();
    }

//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /*****************************************************************************
     * Odstraní z desky všechny trojice vodorovně, resp. svisle sousedících
     * diamantů stejného druhu (tj. se stejnými indexy).
     */
    private int odstraňTrojice( boolean pomalu )
    {
        int ret = 0;
        int body;
        while( (body = prověřŘádky() + prověřSloupce()) > 0 ) {
            ret += body;
            if( hra )
```

```
            diamanty.přidejBody( body );
        if( pomalu )
            sesypAnimovaně();
        else
            sesyp();
    }
    return ret;
}

/*****************************************************************************
 * Zkontroluje, zda v řádcích nejsou trojice či více-tice
 * stejných diamantů a vrátí počet diamantů, které se v nich vyskytují.
 * Zapamatuje si, které diamanty se vyskytují v nalezených n-ticích.
 *
 * @return Počet diamantů vyskytující se v trojicích a víceticích.
 */
private int prověřŘádky()
{
    int ret = 0;
    if( Diamanty.LADIM > 0 )
        tiskniDesku( "Před kontrolou sloupců" );
    //Cyklus přes všechny řádky
    for( int yp = 0;   yp < ROZMĚR;   yp++ ) {
        Diamant poslední = deska[1][yp];
        //Protože diamanty jsou reprezentovány zástupci,
        //mohu shodnost obrazců testovat jako shodnosti instancí zástupců
        boolean dvojice = (deska[0][yp] == poslední);
        //n-tice nemůže končit dřív než ve druhém sloupci
        for( int xp = 2;   xp < ROZMĚR;   xp++ ) {
            Diamant nový = deska[xp][yp];
            if( nový == poslední ) { //Poslední dva se shodují
               if( dvojice ) {   //Shodovaly se i předchozí dva => trojice
                    int počet = označVodorovnouNtici( xp-2, yp );
                    ret += počet;
                    xp  += počet-3;
                    dvojice = false;
                    continue;
                } else {
                    dvojice = true; //Je shodný s předchozím
                    continue;       //Poslední == nový =>
               }                    //  => Přiřazení lze přeskočit
            } else {
               dvojice = false;    //Poslední dva testované netvoří dvojici
            }
            poslední = nový;        //Nyní nový bude příště poslední
        }
    }
    return  ret;
}

/*****************************************************************************
 * Zapamatuje si, které diamanty tvoří vodorovnou n-tici stejných diamantů,
 * jež začíná vlevo na zadaných souřadnicích,
 * a vrátí počet jejích prvků.
 *
 * @param xp   Vodorovná souřadnice prvního prvku
 * @param yp   Svislá souřadnice prvního prvku
 *
```

```
     * @return Počet diamantů v n-tici
     */
    private int označVodorovnouNtici( int xp, int yp )
    {
        int ret   = 0;
        Diamant diamant = deska[xp][yp];
        //Protože diamanty jsou reprezentovány zástupci,
        //mohu shodnost obrazců testovat jako shodnosti instancí zástupců
        while( (xp < ROZMĚR) && (deska[xp][yp] == diamant) ) {
            vyhodit[xp][yp] = true;
            xp++;
            ret++;
        }
        return ret;
    }

    /*************************************************************************
     * Zkontroluje, zda v sloupcích nejsou trojice či více-tice
     * stejných diamantů a vrátí počet diamantů, které se v nich vyskytují.
     * Diamanty vyskytující se v nalezených n-ticích (i vodorovných)
     * nahradí diamantem NIC.
     *
     * @return Počet diamantů vyskytující se v trojicích a víceticích.
     */
    private int prověřSloupce()
    {
        int ret = 0;
        if( Diamanty.LADIM > 0 )
            tiskniDesku( "Před kontrolou sloupců" );
        //Cyklus přes všechny sloupce
        for( int xp = 0;   xp < ROZMĚR;   xp++ ) {
            Diamant poslední = deska[xp][1];
            //Protože diamanty jsou reprezentovány zástupci,
            //mohu shodnost obrazců testovat jako shodnosti instancí zástupců
            boolean dvojice  = (deska[xp][0] == poslední);
            //n-tice nemůže končit dřív než ve druhém sloupci
            for( int yp = 2;   yp < ROZMĚR;   yp++ ) {
                Diamant nový = deska[xp][yp];
                if( nový == poslední ) { //Poslední dva se shodují
                    if( dvojice ) { //Shodovaly se i předchozí dva => trojice
                        int počet = odstraňSvislouNtici( xp, yp-2 );
                        ret += počet;
                        yp   += počet-3;
                        dvojice = false;
                        continue;
                    } else {
                        odstraňSdružený( xp, yp-2 );
                        dvojice = true; //Je shodný s předchozím
                        continue;       //Poslední == nový =>
                    }                   //  => Přiřazení lze přeskočit
                } else {
                   dvojice = false;    //Poslední dva testované netvoří dvojici
                }
                poslední = nový;        //Nyní nový bude příště poslední
                //Prvky, které byly součástí vodorovných n-tic
                //je třeba odstranit
                odstraňSdružený( xp, yp-2 );
            }
```

```
            for( int yp = ROZMĚR-2;   yp < ROZMĚR;   yp++ )
                odstraňSdružený( xp, yp  );
        }
        return  ret;
    }

    /*************************************************************************
     * Je-li diamant na zadaném poli sdružen se svými sousedy do souvislého
     * bloku, a určen proto k odstranění, odstraní jej z desky.
     *
     * @param xp   Vodorovná souřadnice testovaného pole
     * @param yp   Svislá souřadnice testovaného pole
     */
    private void odstraňSdružený( int xp, int yp )
    {
        if( vyhodit[xp][yp] ) {
            deska[xp][yp] = NIC;
            vyhodit[xp][yp] = false;
        }
    }

    /*************************************************************************
     * Dosazením prázdných diamantů odstraní svislou n-tici stejných diamantů,
     * která nahoře začíná na zadaných souřadnicích,
     * a vrátí počet jejích prvků.
     *
     * @param xp   Vodorovná souřadnice prvního prvku
     * @param yp   Svislá souřadnice prvního prvku
     *
     * @return Počet diamantů v n-tici
     */
    private int odstraňSvislouNtici( int xp, int yp )
    {
        int ret   = 0;
        Diamant diamant = deska[xp][yp];
        //Protože diamanty jsou reprezentovány zástupci,
        //mohu shodnost obrazců testovat jako shodnosti instancí zástupců
        while( (yp < ROZMĚR)  &&  (deska[xp][yp] == diamant) ) {
            deska[xp][yp] = NIC;
            yp++;
            ret++;
        }
        return ret;
    }

    /*************************************************************************
     * Projde desku a narazí-li na volné pole, nechá sesypat všechny
     * diamanty, které se nacházejí nad ním.
     */
    private void sesyp()
    {
        if( Diamanty.LADIM > 0 )
            tiskniDesku( "Deska před sesypáním" );
        for( int yp=0;   yp < ROZMĚR;   yp++ )
            for( int xp=0;   xp < ROZMĚR;   xp++ )
                if( deska[xp][yp] == NIC )
                    sesypNad( xp, yp );
    }
```

```
/*****************************************************************************
 * Sesype diamanty nad zadaným polem, tj. posune je o jedno pole dolů.
 *
 * @param xp  Vodorovná políčková souřadnice zadaného pole
 * @param yp  Svislá políčková souřadnice zadaného pole
 */
private void sesypNad( int xp, int yp )
{
    for( int iy=yp;   iy > 0;   iy– )
        deska[xp][iy] =  deska[xp][iy-1];
    deska[xp][0] = Diamant.další();
    if( Diamanty.LADIM > 0 )
        tiskniDesku( "Sesypáno nad pozicí [" +xp+ ";" +yp+ "]:" );
    SP.překresli();
}

/*****************************************************************************
 * Projde desku, a narazí-li na volné pole, nechá animovaně sesypat všechny
 * diamanty, které se nacházejí nad ním.
 */
private void sesypAnimovaně()
{
    if( Diamanty.LADIM > 0 )
        tiskniDesku( "Deska před animovaným sesypáním" );

    for( int yp=0;   yp < ROZMĚR;   yp++ ) {
        //Nejprve předá animátoru všechny objekty, s nimiž je třeba hýbat
        for( int xp=0;   xp < ROZMĚR;   xp++ )
            if( deska[xp][yp] == NIC )
                přidejObjektyNadPolem( xp, yp );
        //a pak jej požádá o to, aby je plynule posunul o pole níž
        animátor.spadni();

        //Usadí animované diamanty do jejich cílových pozic na desku
        AnimObjekt ao;
        while( (ao = animátor.vydej()) != null )
            deska[ao.getXp()][ao.getYp()+1] = ao.getDiamant();
    }
}

/*****************************************************************************
 * Přidá objekty nad zadaným (prázdným) polem mezi ty,
 * které bude animátor záhy přesouvat do nové pozice.
 *
 * @param xp Vodorovná políčková souřadnice prázdného pole
 * @param yp Svislá políčková souřadnice prázdného pole
 */
private void přidejObjektyNadPolem( int xp, int yp )
{
    //Předá všechny objekty nad zadaným polem animátoru
    for( yp–;   yp >= 0;   yp– ) {
        animátor.přidej( deska[xp][yp], xp, yp );
        deska[xp][yp] = NIC;
    }
    //a k nim další, náhodný objekt, který přicestuje na uvolněné místo
    animátor.přidej( Diamant.další(), xp, -1 );
}
```

```
//== TESTY ====================================================================

    /*****************************************************************************
     * Vytiskne na standardní výstup indexy diamantů na desce
     *
     * @param titulek Titulek, kterým bude výpis nadepsán
     */
    public void tiskniDesku( String titulek )
    {
        System.out.println("\n" + titulek);
        System.out.println("     0  1  2  3  4  5  6  7  8  9");
        for( int yp=0;   yp < ROZMĚR;   yp++ ) {
            System.out.print(yp + ": ");
            for( int xp=0;   xp < ROZMĚR;   xp++  )
                System.out.printf( "%3d", deska[xp][yp].ordinal() );
            System.out.println();
        }
    }
}
```

230. Proč třída implementuje rozhraní `IDeska`?

To opět souvisí s návrhovým vzorem *Strategie*, jehož aplikaci jsem ti chtěl ukázat i na příkladu této hry. Jak jsem ale uvedl již v odpovědi 228, ukázka se mi sem již nevejde, tak to ber jako nápovědu pro své experimenty.

231. Dobrá, až budu mít náladu, tak si zaexperimentuju. Vidím, že třída používá vedle třídy `Diamant` ještě další třídy. Můžeš mi něco říct i o nich?

Ty už ti tu vypisovat nebudu, protože s probíraným návrhovým vzorem přímo nesouvisí. Můžeš si je najít mezi doprovodnými programy a prohlédnout si je tam.

Shrnutí – co jsme se naučili

- Vzor *Muší váha* použijeme tehdy, je-li účelné zastoupit skupinu *virtuálních instancí* jednou instancí reálnou, která bude schopna zastupovat kteroukoliv z virtuálních instancí dané skupiny – bude jejich *reprezentantem*.
- Charakteristiky stavu virtuální instance je třeba rozdělit do dvou skupin:
  - *interní charakteristiky*, které jsou pro všechny virtuální instance z dané skupiny společné, a které proto budou i charakteristikami je zastupujícího reprezentanta,
  - *externí charakteristiky*, které se u jednotlivých virtuálních instancí v dané skupině liší, a které se proto bude je zastupující reprezentant dozvídat až „na poslední chvíli“ – např. v parametrech svých metod.
- V programu pak budou místo virtuálních instancí vystupovat reprezentanti skupin, do nichž zastupované instance patří.
- Při zasílání zpráv virtuální instanci, resp. jejímu reprezentantovi, je třeba jim nějakým způsobem zprostředkovat předání informace, o kterou ze zastupovaných instancí se právě jedná.
- Nejčastějším způsobem předání těchto informací je jejich předání v parametrech volaných metod.
- Návrhový vzor *Muší váha* patří mezi vzory uvedené v GoF.

ČÁST 3

Nekoukej mi do kuchyně

V této části se podíváme na návrhové vzory, které ukazují, jak co nelépe zapouzdřit algoritmy, třídy nebo skupiny tříd tak, aby byl uživatel maximálně nezávislý na konkrétní implementaci, takže by neměl mít šanci zaznamenat její případnou změnu. Na druhou stranu uvedené návrhové vzory velice usnadňují jakékoliv změny projektu vyvolané změněnými požadavky zákazníka.

KAPITOLA 14

Pod ruce mi neuvidíš (Zástupce – Proxy)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Zavádí zástupce, který odstiňuje objekt od jeho uživatelů a sám řídí přístup uživatelů k danému objektu.

[1] **Definice v GoF:** Provide a surrogate or placeholder for another object to control access to it. – Poskytuje náhradníka či zástupce objektu, aby mohl lépe ovlivňovat přístup k tomuto objektu.

Účel

232. K čemu může být dobré, když místo s objektem komunikuji s nějakým jeho zástupcem?

Vytvoření zástupce se hodí v řadě nejrůznějších situací. Většinou se využívá k tomu, aby pomohl zapouzdřit implementaci nebo aby dodal nějakou pomocnou funkčnost potřebnou pro implementaci.

233. Tos' mi toho moc nevysvětlil – zkus to vysvětlit podrobněji.

Kategorie zástupců:

GoF uvádí několik typických příkladů situací, v nichž je rozumné použít nějakého zástupce, a podle nich zástupce i kategorizuje:

– vzdálený

- **Vzdálený zástupce** (**remote proxy**) zastupuje objekt umístěný někde jinde – většinou na jiném virtuálním stroji (a často i na jiném počítači). Zprostředkovává komunikaci mezi zastupovaným vzdáleným objektem a objekty ze svého „okolí".

  Hlavním úkolem vzdáleného zástupce je zapouzdřit detaily komunikace a umožnit tak místním objektům komunikovat se vzdáleným objektem stejně, jako kdyby byl objektem místním.

– virtuální

- **Virtuálního zástupce** (**virtual proxy**) využijeme např. při vytváření nákladných objektů, u kterých by bylo vhodné odložit jejich vytvoření na dobu, kdy budou doopravdy potřeba. Do té doby budeme využívat nějakou jejich náhražku.

  GoF uvádí příklad obrázku, který je velmi rozsáhlý, a bylo by jej proto vhodné načítat pouze v případě, kdy bude potřeba jej doopravdy zobrazit. V ostatních případech se místo něj může do dokumentu nakreslit obdélník. Dokud se objekt netiskne a lámací program se ptá pouze na rozměry obrázku, takováto zkratka stačí. Až se bude obrázek tisknout, zástupce jej nechá načíst a vykreslit.

– ochranný

- **Ochranný zástupce** (**protection proxy**) se používá tehdy, potřebujeme-li zatajit některé z dovedností objektu.

  Ochranní zástupci jsou užiteční např. tehdy, mají-li k danému objektu různé objekty různá přístupová práva. Pro každé nastavení přístupových práv je možno definovat zvláštního zástupce, který objektům nedovolí udělat se zastupovaným objektem něco, na co nemají právo.

– chytrý odkaz

- **Chytrý odkaz** (**smart reference**) umožňuje doplnit komunikaci s objektem o některé další akce vhodné např. pro zefektivnění celkového chodu aplikace, pro snazší ladění či další účely.

  K typickým použitím patří zavedení perzistentního[1] objektu do paměti při jeho prvním použití.

  GoF uvádí ještě kontrolu uzamčení objektu před jeho čtením či modifikací a počítání odkazů na objekt, aby bylo možno správně určit okamžik jeho odstranění z paměti, ale při programování v Javě se k těmto účelům používají jiné mechanismy.

---

[1] Jako perzistentní (= trvalé) označujeme objekty, které je možno uchovat mezi jednotlivými spuštěními aplikace. Většinou se tyto objekty ukládají do nějaké databáze nebo do souborů na disku. Někdy se však ukládají pouze klíčové informace o jejich stavu a objekt se v případě potřeby na základě těchto informací vytváří.

Implementace

234. Jestli jsem to dobře pochopil, tak zástupce je vlastně taková obálka na objekt, která si drží odkaz na objekt a požadavky mu buď předá, nebo je před předáním ještě trochu modifikuje.

Více méně máš pravdu. Nicméně např. u vzdáleného zástupce je to *trochu* pořádně rozsáhlé.

Vzdálený zástupce

235. Kde v Javě narazím na vzdáleného zástupce?

Kde se s nimi setkáme

Jak jsem řekl, při komunikaci s objekty běžícími na jiných virtuálních strojích. Součástí standardní knihovny je i skupina tříd realizujících podporu techniky označované RMI (Remote Method Invocation – vzdálené volání metod).

Princip zástupců u RMI

V principu jde o to, že máš dva objekty: server poskytující služby a běžící na jednom stroji a klienta využívajícího těchto služeb a běžícího na jiném virtuálním stroji. Na virtuálním stroji klienta proto existuje objekt zastupující server. Ten přebírá požadavky na server, navazuje se serverem spojení, požadavky mu předává, získává od něj výsledky a ty naopak předává volajícím objektům.

Jak víš, komunikace s druhým počítačem bývá často složitá. O to se však klient nemusí zajímat. Osloví zástupce serveru, o něco jej požádá a dostane od něj výsledek. Pro klienta se tak vzdálený server chová téměř jako lokální, jenom mu všechno trochu déle trvá.

236. Proč jenom „téměř jako lokální“? Vždyť ten zástupce je opravdu lokální a sám jsi říkal, že to, že zástupce komunikuje se vzdáleným serverem, se klient vůbec nedozví.

Co vzdálený zástupce nezakryje

To *téměř* spočívá v tom, že musíš být stále připraven na to, že komunikace se serverem může kdykoliv selhat. Metody zástupce serveru proto musí být schopny vyhazovat i výjimky oznamující, že se zástupci spojení se serverem navázat nepodařilo. Klient pak musí být na možnost vyhození takových výjimek připraven a umět na ně reagovat.

Podrobněji bych se však touto problematikou nechtěl zabývat. Kdyby ses chtěl o RMI a distribuovaných aplikacích dozvědět víc, doporučil bych ti např. [33].

237. No dobře, nechám si prozatím zajít chuť. Pochopil jsem, že vytváření vzdálených zástupců je práce pro profíky.

Neboj, není to tak strašné. Příprava knihovny vyžaduje opravdu hluboké znalosti, ale její použití již tak náročné není. Jenom je k němu potřeba vysvětlit ještě řadu souvisejících oblastí, a na to tady opravdu není místo. Pokud se ale do studia pustíš, tak tě příslušný tutoriál vytváření potřebných zástupců naučí.

Virtuální zástupce

Výhodný, stačí-li zastupovaný objekt jenom emulovat

238. Co mi tedy povíš o implementaci virtuálních zástupců?

Jak jsem již řekl, virtuální zástupci se používají např. při vykreslování grafiky. Obecně je použiješ tehdy, když máš objekt, jehož přítomnost je možno v řadě případů

jenom emulovat, tj. tvářit se, jako by objekt již existoval, i když ve skutečnosti ještě neexistuje (např. k tomu, abych oznámil, kolik místa obrázek zabere, nemusím mít obrázek načtený v paměti).

Když už to jinak nejde, tak objekt zprovozní

Ve chvíli, kdy je zástupce požádán o vykonání operace, která bez reálné existence objektu nejde vykonat (např. ono zmíněné vykreslení obrázku), tak daný objekt vytvoří, načte či jinak zprovozní a nechá jej danou operaci vykonat.

Ochranný zástupce

Implementace ochranného zástupce

239. Myslím, že význam ochranného zástupce jsem pochopil. Prostě definuje pouze takovou podmnožinu metod zastupovaného objektu, které mají příslušné objekty právo volat. Když pak někdo tuto metodu zavolá, předá volání zastupovanému objektu a volajícímu objektu pak předá obdržený výsledek.

Kdy je postup vhodný

Máš pravdu. Tebou popsaný postup je vhodný v situacích, kdy je opravdu důležité, aby volající objekt nemohl žádným způsobem zavolat metody, k jejichž volání nemá právo.

Méně přísné situace

Použití ochranného zástupce je ale vhodné i v situacích, kdy neočekáváš žádné záludné útoky a jde ti spíše o zjednodušení rozhraní, aby uživatel omylem nezavolal metodu, která není pro jeho potřeby vhodná nebo je přímo nebezpečná.

240. Opět požádám o příklad.

Příklad: závody aut

S dětmi v kroužku jsme např. programovali závody automobilů. Automobily jezdily po silnici sestávající ze segmentů. Tyto segmenty musely být schopny se při stavbě závodního okruhu začlenit do budovaného okruhu a navzájem se na sebe napojovat a opět rozpojovat.

Dvě funkce segmentů

Při závodech jsme pak ale potřebovali od segmentu pouze zjistit, kde je, kam vede a který segment se na něj napojuje. Všechny ostatní metody působily v danou chvíli spíše jako rušivý šum. Nehledě na to, že rozpojení segmentů tvořících silnici by uprostřed závodu nebylo žádoucí.

241. Stále nevidím rozdíl. Postup, který jsem uvedl, je použitelný i na tyto případy.

Zjednodušení konstrukce využitím rozhraní

Je, ale v takovýchto případech si můžeš pomoci zjednodušenou konstrukcí, kdy místo plnohodnotného zástupce předávajícího odkazy definuješ pouze `interface` s požadovanou omezenou množinou metod a místo konstruktoru použiješ tovární metodu, která vrací objekty jako instance tohoto rozhraní.

Vzrůst efektivity

Jak zajisté odhadneš, takovéto řešení je efektivnější. Za prvé dá méně práce, za druhé s ním ušetříš paměť (nemusíš vytvářet obalový objekt) a za třetí bude jeho práce i rychlejší, protože se nebudeš zdržovat předáváním požadavků a návratových hodnot. Budeš totiž vždy volat přímo metody příslušného objektu.

242. Když je tak efektivní, proč je nemohu použít vždy?

Oslabení bezpečnosti

Můžeš je použít tam, kde neočekáváš žádné nekalé úmysly. Pokud bys je však použil někde, kde záleží na bezpečnosti, mohl bys případnému útočníkovi poskytnout informace, které bys zveřejnit nechtěl.

243. Jaké?

Nebezpečí využití reflexe

Na rozdíl od plnohodnotného zástupce toto náhradní řešení nijak nebrání využít reflexi a s její pomocí vyvolat i metody, které jsou zdánlivě skryté. Kdyby některé z dětí v kroužku např. iniciativně nastudovalo používání reflexe, mohlo by se segmentu zeptat, jakého je skutečně typu, přetypovat jej na jeho skutečný typ a pak v průběhu závodu záludně měnit dráhu – sobě ji zkracovat, kdežto soupeřům prodlužovat (samozřejmě pouze pokud by byly příslušné metody z nějakého důvodu veřejné a tím i dosažitelné).

Proč to jde

Vše je důsledkem toho, že původní objekt nebyl dokonale odstíněn, ale pouze jsem omezil množinu jednoduše dostupných metod na metody deklarované daným rozhraním. Ve skutečnosti jsou však dostupné všechny metody dané instance, i když vyvolání těch ostatních dá trochu víc práce.

Aby to nešlo

Když bych místo do rozhraní „zabalil" objekt do jiného objektu, neměl by útočník šanci se na jeho skutečnou třídu doptat, protože odkaz na objekt by se mu nikdy nedostal do ruky.

Chytrý odkaz

244. Tak teď jsi mne dostal. Pověz mi raději ještě něco o implementaci chytrých odkazů.

Příklad: iterátor

Sloučením ochranných zástupců a chytrých odkazů jsou např. iterátory, o kterých si budeme vyprávět v některé z následujících kapitol. Ty ti na jednu stranu brání přistupovat přímo ke struktuře uchovávající zpracovávaná data a na druhou stranu mohou doplnit rozsáhlou škálu další funkčnosti. O nich si budeme povídat v kapitole *Moc se mi v tom nebrab (Iterátor – Iterator)* na straně 203. Tam ti také ukážu vše na příkladech.

245. Smířím se s tím, že mi iterátory ukážeš někdy v budoucnu. Neměl bys tam ale také něco, co bys mi mohl ukázat hned teď?

Některé příklady jsem uváděl již v pasáži, kdy jsem ti tento druh zástupce představoval. Abys však neřekl, že škudlím informacemi, přidám další:

Příklad: zástupce vzdáleného objektu

Chytrý zástupce může např. udržovat připojení k databázi nebo vzdálenému serveru. Vždy když dokončíš komunikaci, tak si nastaví nějaký limit, po který bude spojení udržovat. Když po tu dobu program na připojený objekt znovu neosloví, tak spojení jednoduše ukončí.

Jako chytrý odkaz se bude chovat každý virtuální zástupce. Musí se totiž umět vždy rozhodnout, kdy může zastupovat neexistující objekt a kdy už mu nezbude, než daný objekt doopravdy načíst.

Příklad

246. Bude tedy v této kapitole nějaký příklad?

Konkrétní příklad se zdrojovým kódem, který bych tu vytiskl, pro tebe a tuto kapitolu nemám. Řadu příkladů jsem již v textu zmínil, s dalšími se seznámíme v programech, které uvedeme v rámci dalších kapitol.

Shrnutí – co jsme se naučili

- Návrhový vzor *Zástupce* patří mezi vzory uvedené v GoF.
- Zástupce slouží většinou k lepšímu zapouzdření implementace.
- Někdy jej použijeme pro zlepšení funkčnosti.
- GoF rozlišuje zástupce vzdálené, virtuální, ochranné a chytré odkazy.
- Vzdálený zástupce se používá při komunikaci mezi programy na různých virtuálních strojích.
- Vzdálený zástupce nemůže zakrýt selhání spojení.
- Virtuální zástupce může šetřit zdroje emulací objektu. Zprovozní jej, až když už emulace nestačí.
- Ochranný zástupce zamezuje přímé komunikaci s objektem.
- Stačí-li omezit množinu nabízených metod a není-li třeba přísně hlídat neoprávněné pokusy, lze ochranného zástupce nahradit implementací jednoduchého rozhraní.
- Ochranný zástupce nahrazený implementací rozhraní snižuje bezpečnost, ale zvyšuje efektivitu.
- Chytrý odkaz doplňuje odkaz o dodatečnou funkčnost.

KAPITOLA 15

Řekni, až to budeš chtít (Příkaz – Command)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Zabalí metodu do objektu, takže s ní pak lze pracovat jako s běžným objektem. To umožňuje dynamickou výměnu používaných metod za běhu programu a optimalizaci přizpůsobení programu požadavkům uživatele.

[1] **Definice v GoF:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. – Zapouzdří požadavek jako objekt, čímž umožní parametrizovat klienty s různými požadavky, vytvářet fronty a záznamy požadavků a podporovat odvolatelné operace.

Účel

Převod programu na data

247. Při čtení stručné charakteristiky v úvodu této kapitoly mi připadalo, že se tento návrhový vzor snaží dělat z programu data.

To jsi docela vystihl. S podobným přístupem se u návrhových vzorů občas setkáš. Objektoví programátoři totiž velmi často řeší vhodným uspořádáním dat problémy, které by klasičtí programátoři řešili nějakým programem. Data se totiž mění mnohem snáze než program, takže takové řešení bývá daleko snáze přizpůsobitelné postupně se měnícím požadavkům zákazníka.

248. Takovéto zdůvodnění je sice krásné, ale pro moji jednoduchou hlavu příliš obecné. Zkus ty výhody popsat konkrétněji.

Příklad, kdy se hodí

Dobrá. Návrhový vzor *Příkaz* se většinou vykládá na příkladu grafického uživatelského rozhraní. V dialogových oknech se vyskytují různé komponenty. Když připravuješ např. definici tlačítka, nestaráš se o to, jakou funkci programátor tlačítku ve svých oknech přiřadí. Navíc bude tlačítku s velkou pravděpodobností přiřazena v různých oknech různá funkce.

Teoreticky bys sice mohl definovat pro každé tlačítko třídu, která by překryla tu metodu rodičovské třídy tlačítek, jež by definovala požadovanou akci. Jenomže jak jistě víš, funkce jednoho a téhož tlačítka se může v době života programu změnit. Při změně funkce tlačítka bys pak při tomto řešení musel nahradit původní tlačítko jiným, jež by bylo instancí třídy definující tu správnou reakci na stisk.

249. Jak bys tedy doporučoval takovou situaci řešit?

Doporučené řešení

Pokud bych chtěl tvárnější řešení, tak bys mohl tlačítku definovat atribut odkazující na objekt, který bude vědět, co se má při stisku udělat. Změna funkce tlačítka by spočívala pouze ve změně hodnoty daného atributu. Ostatní charakteristiky tlačítka (vzhled, umístění, popisek, …) by při ní mohly zůstat nezměněny a nemusely by se nikam kopírovat.[1]

Implementace

250. Ten objekt reagující na stisk tlačítka je vlastně služebník, který to tlačítko obsluhuje.

Rozdíl mezi služebníkem a příkazem:
– Služebník

Ano. Návrhový vzor *Služebník* a *Příkaz* jsou si velice podobné. Implementace je často prakticky stejná. Liší se vlastně pouze v tom, jak k celému problému přistupuješ:

- V případě vzoru *Služebník* máš na počátku objekty, kterým chceš zajistit obsluhu. Obstaráš proto třídu, jejíž instance umějí požadovanou činnost a která definuje rozhraní, jehož implementaci bude od obsluhovaných objektů vyžadovat. Obsluhované instance pak předáváš metodám instancí služebníka jako parametry.

[1] Zkušenější možná namítnou, že ve standardní knihovně není použit vzor *Příkaz*, ale vzor *Pozorovatel*. Tady musím namítnout, že tam jsou použity oba vzory současně. Při hlubší analýze zjistíte, že řada návrhových vzorů obsahuje ve svém návrhu jeden či více jednodušších vzorů.

– Příkaz

- V případě vzoru *Příkaz* máš na počátku objekty (nebo také jenom část kódu), které chceš doplnit o nějakou funkčnost. Definuješ proto rozhraní, jež budou muset implementovat třídy, jejichž instance by měly požadovanou funkčnost realizovat. Tyto instance pak předáš původním objektům jako parametry jejich metod (resp. zavoláš jejich metody v oné části kódu).

Pro obrat, kdy instance předává volané metodě jako parametr sebe samu, se občas používá termín zpětné volání. Instance totiž předává sama sebe proto, aby daná metoda posléze za nějaké předem dohodnuté situace zavolala zpátky domluvenou metodu této instance.

To, že jsou si vzory *Příkaz* a *Služebník* podobné, ale neznamená, že je tomu tak vždy. Existuje řada situací, kdy použití návrhového vzoru *Příkaz* s návrhovým vzorem *Služebník* nijak nesouvisí. V těchto situacích většinou opravdu potřebuješ předat volané metodě pouze odkaz na metodu, kterou bude volaná metoda při plnění svého úkolu potřebovat. Protože v Javě není možno předat odkaz na metodu, musíme předat objekt implementující rozhraní, jež deklaruje signaturu předávané metody.

Příklad

251. Budeš mít pro mne nějaký příklad?

Dokonce mám hned dva. První by ti měl být blízký, protože je zabudován ve standardní knihovně. Jde o třídění polí, resp. vyhledávání v polích s využitím komparátoru.

252. Teď mám zrovna nějaké drobné okno – mohl bys mi ten problém připomenout?

Třídění polí

Jedná se o metody `sort` a `search` třídy `java.util.Arrays`. Potřebuješ-li umět třídit pole podle různých kritérií, resp. vyhledávat v polích setříděných podle různých kritérií, bylo by poněkud nemoudré snažit se vystačit s metodou `compareTo(Object)`, jejíž vyhodnocovací algoritmus bys operativně nastavoval podle aktuálního kritéria.

Daleko výhodnější je zavolat metodu, které vedle tříděného pole, resp. pole, v němž vyhledáváš, dodáš jako další parametr komparátor, a ten bude umět rozpoznat, který ze dvou dodaných parametrů je ten menší a který ten větší.

253. Už mi začíná naskakovat. Tak mi ještě ukaž program, v němž danou metodu použiješ.

Popis zadání

Omezím se na jednoduchý AHA-příklad. Představ si, že máš pole křížovkářských výrazů, které bys potřeboval užitečně setřídit. Pro luštění křížovek není třídění podle abecedy příliš užitečné. Potřebuješ mít pole setříděné nejprve podle počtu písmen a teprve skupiny výrazů se stejným počtem písmen dotřídit podle abecedy.

V programu, jehož zdrojový kód najdeš ve výpisu 15.1, jsem abstrahoval od toho, že bys tyto výrazy měl uchovávat v přepravce, jejíž jeden atribut by obsahoval dané slovo a druhý vysvětlení jeho významu. V zájmu co největší stručnosti jsem se omezil pouze na setřídění příslušného pole řetězců.

Výpis 15.1: Třída `Křížovka` definující řazení křížovkářských výrazů

```
package rup.česky.vzory._14_příkaz;

import java.util.Arrays;
import java.util.Comparator;
import java.util.Random;
import rup.česky.společně.Slovy;


/*******************************************************************************
 * Třída Křížovka slouží jako knihovní třída.
 */
public class Křížovka
{
//== NESOUKROMÉ METODY TŘÍDY ===================================================

    /*******************************************************************************
     * Seřadí řetězce v poli nejprve podle délky
     * a pak podle abecedy bez ohledu na velikost písmen.
     * @param názvy  Tříděné pole
     */
    public static void seřaď( String[] názvy ) {
        Arrays.sort( názvy, new
            Comparator<String>() {
                public int compare( String s1, String s2 ) {
                    int d1 = s1.length();
                    int d2 = s2.length();
                    if( d1 == d2 )
                        return s1.compareToIgnoreCase( s2 );
                    else
                        return (d1 - d2);
                }
            }
        );
    }


    /*******************************************************************************
     * Seřadí řetězce v poli nejprve podle délky
     * a pak podle abecedy bez ohledu na velikost písmen -
     * verze pro ty, kdo nemají rádi anonymní třídy.
     * @param názvy  Tříděné pole
     */
    public static void seřaďB( String[] názvy ) {
        Arrays.sort( názvy, new Komparátor() );
    }


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /** Soukromý konstruktor bránící vytvoření instance. */
    private Křížovka() {}

//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    private static class Komparátor implements Comparator<String>
```

```
    {
        /** {@inheritDoc} */
        public int compare( String s1, String s2 ) {
            int d1 = s1.length();
            int d2 = s2.length();
            if( d1 == d2 )
                return s1.compareToIgnoreCase( s2 );
            else
                return (d1 - d2);
        }
    }


//== TESTY ====================================================================

    /*****************************************************************************
     * Test funkce metody řadící texty nejprve dle délky a pak dle abecedy.
     */
    public static void test() {
        final int TEXTŮ = 20;
        final int DÉLKA = 20;
        Random RND = new Random();
        String[] pole = new String[ DÉLKA ];
        for( int i=0;   i < DÉLKA;   i++ )
            pole[i] = Slovy.slovy( RND.nextInt( TEXTŮ ) );

        System.out.println("Před seřazením:");
        for( String text : pole )
            System.out.println( text );

        seřaď( pole );

        System.out.println("\nPo seřazení:");
        for( String text : pole )
            System.out.println( text );
    }/** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String... args ) { test(); }
}
```

254. Tady jsi ale předvedl pouze první polovinu, tj. definici objektů definujících příkaz a jejich předání metodě, která je bude používat.

V předchozím příkladu jsem ti chtěl hlavně ukázat příklad toho, jak je tento vzor použit ve standardní knihovně – konkrétně v metodě pro třídění, která je definována tak, aby nemusela předem znát pravidlo, podle něhož objekty třídíš. Požadovanou funkčnost jí proto předáš v objektu obalujícím onen porovnávací příkaz.

Navíc jsem ti chtěl připomenout, že při použití návrhového vzoru *Příkaz* se často používají anonymní třídy, takže pokud bys je neměl rád a nepoužíval je, měl by ses jim naučit rozumět alespoň natolik, abys mohl číst cizí programy.

Ve standardní knihovně ale bude použití vzoru vždy schováno v jejím zdrojovém kódu. Ten je sice součástí JDK, ale ze zkušeností vím, že začátečníci s jeho luštěním trochu zápasí.

255. Takže v druhém příkladu mi jej již předvedeš celý?

Budu se snažit. Říkal jsem si, že ses s jedním použitím tohoto vzoru nejspíš ještě nesetkal, protože se o něm učebnice většinou nezmiňují. Tím je použití vzoru *Příkaz* při konstrukci pole metod.

256. Pole metod? To zní zajímavě a ještě jsem o něm doopravdy neslyšel. O co jde?

Pole metod

Jsou situace, kdy máš skupinu nějakých metod řešících nějaký problém a předem nevíš, kdy budeš kterou z nich potřebovat. Definuješ si proto pole metod a podle okamžité situace se vždy rozhodneš, kterou z nich vybrat a aplikovat.

257. Jak se v Javě definuje pole metod? Já myslel, že to nejde.

Samozřejmě že to nejde. Musíš aplikovat návrhový vzor *Příkaz* a místo pole metod definovat pole objektů, které implementují rozhraní specifikující signatury příslušných metod.

258. Myslím, že bude nejlepší, když o tom nebudeme mluvit a rovnou mi to ukážeš.

Příklad využívající pole metod

Správně. Připravil jsem ti jednoduchoučký prográmek, který simuluje házení kostkou. Aby byl co nejjednodušší a použití pole metod nebylo „zašuměno“ okolním programem, nevyužívám v něm grafiky, ale posílám obrázek horní strany kostky jako text do zadaného výstupního proudu.

Jak si však jistě domyslíš, program by se choval obdobně i v případě, kdybych nahradil znakový obrázek kresleným a místo písmen posílaných na standardní výstup vykresloval na plátno kolečka či jiné obrazce.

Výpis 15.2: Program `Kostka` simulující házecí kostku

```
package rup.česky.vzory._15_příkaz;

import java.io.PrintStream;
import java.util.Random;

/*******************************************************************************
 * Instance třídy Kostka simulují házecí kostky.
 */
public class Kostka
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Generátor náhodných čísel společný pro všechny kostky. */
    private static final Random RND = new Random();

    /* Možné podoby řádků na horní straně kostky. */
    private static final String s0  = "|     |\n",
                                s1L = "|0    |\n",
                                s1C = "|  0  |\n",
                                s1R = "|    0|\n",
                                s2  = "|0   0|\n",
                                sv  = "+-----+\n";

    /** Výstupní proud, do kterého půjdou texty určené k zobrazení kostky. */
    private static final PrintStream OUT = System.out;
```

```
    /** Generátor vzhledu vrchní strany kostky při hození jednotlivých čísel. */
    private static final IHod[] hod =
    {
        null,
        new IHod() { // 1
            public void zobraz() { OUT.println( sv+s0+s1C+s0+sv ); }
        },
        new IHod() { // 2
            public void zobraz() { OUT.println( sv+s1L+s0+s1R+sv ); }
        },
        new IHod() { // 3
            public void zobraz() { OUT.println( sv+s1L+s1C+s1R+sv ); }
        },
        new IHod() { // 4
            public void zobraz() { OUT.println( sv+s2+s0+s2+sv ); }
        },
        new IHod() { // 5
            public void zobraz() { OUT.println( sv+s2+s1C+s2+sv ); }
        },
        new IHod() { // 6
            public void zobraz() { OUT.println( sv+s2+s2+s2+sv );; }
        },
    };


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Simuluje hod čísla, které následně zobrazí číslem a obrázkem kostky.
     */
    public void hoď()
    {
        int číslo = RND.nextInt( 6 ) + 1;
        System.out.println( "Padlo číslo: " + číslo );
        hod[ číslo ].zobraz();
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /** Rozhraní charakterizuje objekt
     *  vykreslující podobu horní strany kostky. */
    private static interface IHod
    {
        public void zobraz();
    }


//== TESTY A METODA MAIN =======================================================

    /*****************************************************************************
     * Vytvoří instanci kostky a vyzkouší několik hodů.
     */
    public static void test()
    {
        Kostka k = new Kostka();
        for( int i=0;   i < 10;   i++ )
```

```
            k.hoď();
    }

    public static void main( String[] args )  {  test();  }
}
```

259. Pořád se mi zdá, že použití příkazu `switch` by bylo jednodušší.

Nevýhody řešení pomocí `switch`

Jednodušší by bylo pouze do té chvíle, pokud bys měl zaručeno, že se bude neustále házet do šesti. Hrozí-li ti ale neustálé změny zadání (a to je u současných zákazníků spíše pravidlem), bude použití vzoru *Příkaz* mnohem operativnější.

Výhody při změnách algoritmu „za chodu"

V předchozím příkladu byly ještě jednotlivé příkazové objekty definovány uvnitř generátoru. Návrhový vzor *Příkaz* ti je však umožňuje definovat zcela mimo generátor, kterému předáš pole s příslušnými příkazovými objekty jako parametr. Pak budeš moci způsob vykreslování měnit dynamicky za chodu aplikace. To ti příkaz `switch` neumožní.

Další příklad

Když se ještě vrátím k minulému příkladu s tříděním, v podobném případě bys mohl tento návrhový vzor využít k tomu, aby si uživatel mohl vybrat třídicí algoritmus z nějakého seznamu a zadané pole by se seřadilo podle algoritmu vybraného zákazníkem.

Shrnutí – co jsme se naučili

- Při implementaci návrhového vzoru *Příkaz* definujeme rozhraní deklarující požadovanou metodu – příkaz. Metodám pak předáváme instance tříd implementujících toto rozhraní.
- Návrhový vzor *Příkaz* je velice blízký návrhovému vzoru *Služebník*.
- Návrhový vzor *Příkaz* využijeme v situaci, kdy potřebujeme zabezpečit možnost zpětného volání.
- Další oblastí použití jsou situace, kdy potřebujeme volané metodě předat metodu, kterou bude při své práci potřebovat.
- Málo známým použitím je i možnost definice pole metod.
- Návrhový vzor *Příkaz* patří mezi vzory uvedené v GoF.

KAPITOLA 16

Moc se mi v tom nehrab (Iterátor – Iterator)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Zprostředkuje jednoduchý a přehledný způsob sekvenčního přístupu k objektům uloženým v nějaké složité struktuře (většinou v kontejneru), přičemž implementace této struktury zůstane klientovi skryta.

[1] **Definice v GoF:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. – Zpřístupňuje prvky agregovaného objektu (kontejneru), aniž by zveřejnila jeho vnitřní reprezentaci.

Účel

260. S iterátory jsem se setkal, když jsem se učil pracovat s různými druhy kolekcí. Umím je používat, ale přiznám se, že jsem zcela nepochopil jejich podstatu.

Co je to iterátor

Iterátor je prostředek řešící problém, jak zapouzdřit způsob uložení objektů v kontejneru a přitom umožnit procházení kontejnerem a práci s uloženými prvky.

Zdůvodnění na příkladu

Máš dejme tomu množinu, kam si ukládáš zákazníky, kteří ti ještě nezaplatili. Jednou za čas potřebuješ projít všechny objekty této množiny a zákazníkům, kterým vypršela lhůta splatnosti, poslat upomínku. Chceš-li však dodržet nejdůležitější zásadu objektového programování, nemůžeš pustit žadatele k poli (případně jiné datové struktuře), v němž máš zákazníky uloženy, protože bys tím prozradil způsob implementace množiny, a jak víš, ten má zůstat zapouzdřen.

Aby se vlk nažral a koza zůstala celá, tj. aby klient mohl procházet množinou, ale její implementace přitom zůstala neprozrazená, vložíš mezi klienta a datovou strukturu, v níž máš objekty uloženy, další objekt. Ten bude klientovi na požádání poskytovat odkazy na jednotlivé uložené objekty, aniž by mu cokoliv prozradil o způsobu, jakým jsou uloženy. No a tímto objektem je iterátor.

261. Takže iterátor je vlastně takový specializovaný zástupce.

Iterátor = specializovaný zástupce

Dalo by se to tak říct. Mohli bychom jej označit jako zástupce struktury, v níž máš uloženy objekty, k nimž chceš zachovat přístup, aniž bys o této struktuře cokoliv prozradil.

Implementace

Jak zprostředkovat přístup bez narušení zapouzdření

262. Obávám se, že zavedením iterátoru se problém zapouzdření nevyřešil. Místo klientovi jsi implementaci prozradil iterátoru. Zapouzdření je tedy stejně porušené.

Ale není. Musíš iterátor definovat trochu nestandardně, aby se dostal mezi privilegované třídy, které mají přístup k soukromým složkám. V jazyku Java definuješ třídu iterátoru jako vnitřní, v jazyku C++ ji definuješ jako spřátelenou.

Požadované vlastnosti:

263. Pravda, takhle by to šlo. A jaké musí mít takový iterátor vlastnosti?

Iterátor by měl mít definované především dvě metody:

– další!

– další?

- Metodu, která vrátí další z uložených objektů.
- Metodu, která vrátí informaci o tom, zda už byly „navštíveny“ všechny uložené objekty nebo zda ještě zbývá nějaký „nenavštívený“.

Další možnosti:

Kromě toho může mít samozřejmě řadu dalších metod, které mohou sloužit nejrůznějším účelům, především pak:

– ovlivnění průběhu

– změna obsahu

– práce s položkami

- k ovlivnění průběhu iterace (změna směru iterace, přesun „kurzoru“, zadání filtru, …),
- ke změně obsahu „kontejneru“ (přidání prvku, vyjmutí prvku),
- k práci s těmito prvky.

264. Říkal jsi, že by iterátor měl mít metodu na vrácení dalšího objektu a metodu vracející informaci o ukončení iterace. Neřekl jsi ale, že je mít musí. Pochopil jsem to dobře?

Není to povinné

Pochopil. Nejsou to vlastnosti povinné, ale spíše bychom je mohli označit za typické. Iterátor např. může tyto dvě metody sloučit do jedné a v případě, že již prošel celý kontejner, tak vrátit místo dalšího prvku např. prázdný ukazatel `null`.

Skrytá iterace

Dokonce nemusí mít ani jednu z nich. Takovému iterátoru zadáš operaci, kterou chceš s prvky v kontejneru provést, a on je prostě všechny projde a danou operaci s nimi provede. Mohli bychom o něm říct, že iteruje skrytě.

265. A když ji nechci provést se všemi prvky?

Filtrace

Tak mu spolu s operací zadáš ještě filtr, který určí, zda se s daným prvkem má daná operace provádět či nikoliv.

266. Vidím, že problematika iterátorů je mnohem rozsáhlejší, než jsem se na počátku domníval.

Nekonečný iterátor

Tak abych tě dorazil, přidám ti ještě dvě informace. První z nich je, že iterátor nemusí být konečný, tj. že nemusí nikdy nastat situace, že už ti vše předal. Pokud bude iterátor vracet např. výsledky měření přicházející z vnějšího zdroje, nemusí nikdy skončit.

Zkrácený cyklus `for`

Druhou informací, která by neměla zapadnout, je, že zkrácený cyklus `for`, který doplnila pátá verze Javy, můžeš použít pro všechny iterovatelné třídy, tj. pro všechny třídy implementující rozhraní `java.lang.Iterable`.

267. Proboha, už s těmi novými informacemi zastav! Pojď se vrátit na začátek a ukaž mi, jak lze vytvořit třídu s tím nejjednodušším iterátorem.

Dobrá. Ukážu ti definici třídy `IterovatelnéPole`, která pouze zabalí pole do iterovatelného objektu – najdeš ji ve výpisu 16.1.

268. Opatrně se zeptám: je to jenom AHA-příklad, nebo může být takový objekt někdy užitečný? Vždyť pole je přirozeně iterovatelné.

Pole není běžný iterovatelný objekt

Pro pole je sice možno použít zkrácenou verzi příkazu `for`, ale není iterovatelné stejně jako jiné iterovatelné objekty. Pole totiž neimplementuje rozhraní `Iterable`, protože nedefinuje metodu `iterator()`. Chceš-li v programu postavit pole do jedné řady s ostatními iterovatelnými objekty, musíš jej takto zabalit.

269. K čemu by mi bylo takovéto „stavění do řady“ dobré?

Proč bývá vhodné zabalit pole

Abys mohl jednotně zpracovávat data získaná z různých zdrojů. Já jsem to např. použil, když jsem připravoval sady testů, z nichž některé byly zadány jako jednorozměrná pole testovacích kroků a jiné jako textové soubory zadávaných a očekávaných hodnot.

Hodné pozornosti:

270. Je na té definici něco, co bych neměl přehlédnout?

Rád bych tě upozornil na několik věcí:

– implementace `Iterable`

- Aby mohla být pro instance naší třídy používána zkrácená verze cyklu `for`, implementuje třída rozhraní `Iterable`.

– klonování

- Konstruktor neukládá do atributu předaný odkaz, ale odkaz na klon daného objektu. Jedině tak lze zabezpečit, aby se v obalovaném poli již nikdo nehrabal. I kdyby si snad někdo dodatečně vzpomněl a nějakou položku do pole přidal nebo ji z něj odstranil, klon to již nijak neovlivní.

  Pokud bys náhodou cítil jako výhodné, abys při průchodu polem měl v každém okamžiku poslední verzi dané položky, pak samozřejmě vytváření klonu opustíš a do atributu uložíš odkaz na původní pole.

  Třetí možností je uložit do atributu odkaz na původní pole a vytvářet klon až v okamžiku, kdy se konstruuje iterátor. Iterátor pak bude procházet polem v podobě, která platila v době jeho zrodu.

  Jaká verze je nejsprávnější, to záleží na konkrétní aplikaci.

Díky tomu, že se klon vytváří jak v konstruktoru iterovatelného pole, tak v konstruktoru iterátoru, je v aplikaci zbytečně „překlonováno“. Kdybys někdy potřeboval vytvořit něco podobného, měl bys nejméně jedno vytváření klonu vyhodit. Které to bude, to záleží na povaze aplikace.

– iterátor je vnitřní třídou

- Třída iterátoru je definována jako vnitřní. Potřebuje být totiž napojena na atribut instance, nad nímž iteruje.

– navíc soukromou

- Třída iterátoru je definována jako soukromá, protože nikomu není nic do toho, jak to instance dělá, že umí vrátit iterátor.

– neimplementuje `remove()`

- Iterátor neimplementuje metodu `remove()`. Přesněji: implementuje ji, ale pouze formálně, aby uspokojil překladač. Jakékoliv volání této metody však způsobí vyhození výjimky.

Výpis 16.1: Definice třídy `IterovatelnéPole`

```
package rup.česky.vzory._16_iterátor;

import java.util.Iterator;


/*******************************************************************************
 * Instance třídy IterovatelnéPole slouží jako obálka pole,
 * která poskytuje explicitní iterátor.
 *
 * @param <Položka> Typ položek uložených v poli
 */
public class IterovatelnéPole<Položka> implements Iterable<Položka>
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final Položka[] pole;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří novou instanci, která bude iterovat nad zadaným polem.
     * @param  pole  Pole, nad nímž chceme iterovat
     */
```

```
    IterovatelnéPole( Položka[] pole ) {
        this.pole = pole.clone();
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /****************************************************************************
     * Vrátí standardní iterátor procházející daným polem.
     * @return Požadovaný iterátor
     */
    public Iterator<Položka> iterator() {
        return new IteratorPole();
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /****************************************************************************
     * Iterátor nad atributem jeho instance.
     */
    private class IteratorPole implements Iterator<Položka>
    {
        int       index = 0;
        Položka[] ipole = pole.clone();

        /************************************************************************
         * @return Existuje-li ještě nějaká nevrácená položka pole
         */
        public boolean hasNext() {
            return (index < ipole.length );
        }

        /************************************************************************
         * @return Odkaz na další uloženou položku
         */
        public Položka next() {
            return ipole[ index++ ];
        }

        /************************************************************************
         * @throws UnsupportedOperationException - Operace není podporována
         */
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

//== TESTY A METODA MAIN =======================================================

    private static final Integer[] testPole = { 1, 3, 5, 7 };

    /****************************************************************************
     * Testovací metoda.
     */
    private static void test() {
        IterovatelnéPole<Integer> ip = new IterovatelnéPole<Integer>(testPole);
```

```
        for( Integer i : ip ) {
            System.out.println("Hodnota=" + i + ", mocnina=" + i*i );
        }
    }/** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {  test();  }  /*-*/
}
```

271. Protože třída `IterovatelnéPole` implementuje rozhraní `Iterable`, musí vracet iterátor implementující rozhraní `java.util.Iterator`. Má smysl vracet také jiný iterátor?

– Užitečnost vracení jiného druhu iterátoru

Záleží na tom, co si od něj slibuješ. Chceš-li využívat dobrodiní zkráceného cyklu `for`, musíš umět vracet instanci rozhraní `java.util.Iterator`. To ale neznamená, že nemůžeš vracet instanci jeho potomka, jako to ve standardní knihovně dělají seznamy, které vracejí `java.util.ListIterator`. Ten umí měnit v průběhu iterace její směr a umí dokonce vkládat nové prvky před nebo za naposledy předaný prvek.

Záleží zkrátka na tom, co chceš, aby tvůj iterátor uměl navíc. Užitečným rozšířením se meze nekladou.

272. Z toho, co jsi před tím říkal, jsem ale pochopil, že iterátor nemusí nutně implementovat rozhraní `java.util.Iterator`.

Iterátor nemusí implementovat `Iterator`

Nemusí. Nelpíš-li na možnosti využívání zkrácené verze příkazu `for`, nemusí se tvůj iterátor na rozhraní `Iterator` vůbec ohlížet a můžeš pouze implementovat ducha tohoto návrhového vzoru. Přiznejme si ale, že tuto možnost tvůrci programů v Javě příliš nevyužívají. Práci s instancemi tříd implementujících rozhraní `Iterator` a `ListIterator` mají zažitou a nechce se jim měnit zvyky. Potřebují-li proto definovat nějaký zvláštní iterátor, definují jej jako implementaci rozhraní `Iterator`.

Nicméně, jak jsem již řekl, není to povinnost a i ve standardní knihovně najdeš iterátory, které se k rozhraní `Iterator` nehlásí, protože pocházejí ze starších verzí, kdy se místo instancí rozhraní `Iterator` používaly instance rozhraní `Enumeration`.

273. Jestli jsem to dobře pochopil, tak ses snažil naznačit, že iterátory se používají i k jiným účelům než pouze k prohrabování nějakých kontejnerů. Mohl bys uvést příklad nějakého takového zvláštního iterátoru?

Příklady ze standardní knihovny

Takovými zajímavě rozšířenými iterátory jsou např. instance třídy `java.util.Scanner`, které slouží k rozdělení vstupního textu na jednotlivé položky (tokens). Odstíní tě tak od zdroje onoho textu, kterým může být obyčejný textový řetězec (`String`), soubor či cokoliv implementující rozhraní `java.lang.Readable`. Navíc jsou to zástupci oněch potenciálně nekonečných iterátorů, kterými jsem tě před chvílí tak vyděsil, protože např. vstupní proudy mohou poskytovat zdánlivě nekonečný zdroj vstupních dat.

274. Přemýšlím nad tím, jestli by se nějak nedala ošidit nutnost definovat nepoužitelnou metodu `remove()`. Řekl bych, že takovýto iterátor, který neumožňuje mazání, by mohl být potřeba docela často.

Pro tento účel jsem definoval abstraktní třídu `IteratorAdapter` (viz výpis 16.2), která definuje vzorovou verzi nepodporované metody `remove()`. Budeš-li chtít vytvořit třídu,

která má implementovat rozhraní `Iterator`, ale nebudeš v ní chtít implementovat metodu `remove()`, můžeš svoji třídu definovat jako potomka třídy `IteratorAdapter` a její verzi metody `remove()` jednoduše zdědit.

Třída `IteratorAdapter` implementuje návrhový vzor *Adaptér*, kterým se budeme podrobněji zabývat v kapitole *Je to trochu jinak (Adaptér – Adapter)* na straně 261.

Výpis 16.2: Třída `IteratorAdapter` definující adaptér s předdefinovanou „nepodporovanou" metodou `remove()`

```
package rup.česky.vzory._16_iterátor;

import java.util.Iterator;

/*******************************************************************************
 * Instance třídy IteratorAdapter slouží jako rodičovské třídy
 * pro iterátory, které nebudou povolovat odstraňování prvků
 * iterovaného kontejneru.
 *
 * @param <E> Typ prvků iterovaného kontejneru.
 */

public abstract class IteratorAdapter<E> implements Iterator<E>
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /***************************************************************************
     * {@inheritDoc}
     */
    public void remove() {
        throw new UnsupportedOperationException(
            "Tento iterátor nepovoluje odstraňování prvků" );
    }

}
```

Příklad

275. Ukážeš mi na tom příkladu, jak bych mohl rozšířit iterátor o některé ty speciální funkce, o nichž jsi před tím mluvil?

Přiznám se, že se mi to nechce demonstrovat na AHA-příkladu. Zkusíme si raději nadefinovat nějakou zajímavější iterovatelnou třídu a předvedeme si to na ní.

276. Beru. Copak sis na mne připravil.

Popis příkladu

Abys nepodlehl bludu, že iterátory pracují pouze nad lineárními daty, jakými jsou např. pole, datové proudy či běžné kontejnery, ukážu ti příklad iterátoru, který iteruje nad stromem složek. Ty mu zadáš složku a on ti bude postupně nabízet všechny soubory v této složce a všech jejích podsložkách. V tomto příkladu by se pak mohla ukázat užitečnost nejrůznějších rozšíření, o nichž jsme před tím mluvili.

277. Skvělé – sem s ním!

Možnosti třídy `Složka`

Zdrojový kód třídy `Složka`, která celou problematiku řeší, si můžeš prohlédnout ve výpisu 16.4. Třída nabízí většinu rozšiřujících možností, o nichž jsme spolu hovořili:

- metoda `setFilter(FileFilter)` umožňuje zadat filtr ovlivňující, které z iterovaných položek (tj. které ze souborů) budou iterátorem předány a které přeskočeny,
- metoda `setComparator(Comparator<File>)` umožňuje zadat komparátor ovlivňující pořadí, v němž ti bude iterátor vracet jednotlivé položky,
- metoda `aplikuj(IPříkaz)` umožňuje zadat objekt, jehož metoda bude aplikována na všechny soubory propuštěné filtrem.

Testy jsou zvlášť

Tentokrát je bez testu, protože testů je víc a jsou rozděleny do několika metod. Umístil jsem je proto do samostatné třídy, jejíž zdrojový kód je ve výpisu 16.5.

278. O rozhraní `IPříkaz`, jehož instanci dostává metoda `aplikuj(IPříkaz)` jako parametr, jsi se dosud nikde nezmiňoval.

No, vždyť je také trapně jednoduché. Je součástí aplikace návrhového vzoru *Příkaz*, o němž jsme hovořili v minulé kapitole. Jeho definici najdeš ve výpisu 16.3.

Výpis 16.3: Definice rozhraní `IPříkaz`

```
package rup.česky.vzory._16_iterátor;

/*******************************************************************************
 * Rozhraní <code>IPříkaz</code> je součástí návrhového vzoru <i>Příkaz</i>
 * a deklaruje metodu, kterou bude nabízet objekt
 * zadávaný jako příkazový parametr.
 */
public interface IPříkaz
{
//== POŽADOVANÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Metoda provádějící požadovanou činnost s předem neznámým počtem
     * parametrů blíže nespecifikovaného typu, která nic nevrací.
     *
     * @param objects  Pole s proměnným počtem parametrů
     */
    public void příkaz( Object... objects );

}
```

Příklad vylepšeného iterátoru

Výpis 16.4: Definice třídy `Složka` umožňující iterovat přes všechny soubory ve složce a jejích podsložkách

```
package rup.česky.vzory._16_iterátor;

import java.io.File;
import java.io.FileFilter;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;
```

```
import java.util.Stack;

/*******************************************************************************
 * Instance třídy Složka představují iterovatelné pseudokontejnery
 * simulující, že obsahují soubory v zadané složce a všech jejích podsložkách.
 * Jsou schopny vrátit iterátor, který postupně všechny tyto soubory projde
 * a nabídne.
 */
public class Složka implements Iterable<File>
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Složka, jejíž soubory jsou procházeny. */
    private final File složka;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Případný filtr vracených souborů. */
    private FileFilter filtr;

    /** Případný komparátor pro vracení souborů v zadaném pořadí. */
    private Comparator<File> komparátor;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*******************************************************************************
     * Vytvoří instanci napojenou na nějakou existující složku
     * @param složka Složka, na niž bude instance napojena
     */
    public Složka( String složka )
    {
        this( new File( složka) );
    }


    /*******************************************************************************
     * Vytvoří instanci napojenou na nějakou existující složku
     * @param složka Složka, na niž bude instance napojena
     */
    public Složka( File složka )
    {
        if( !(složka.isDirectory() &&  složka.exists()) )
            throw new IllegalArgumentException(
                "\nParametrem konstruktoru smí být pouze existující složka\n\""
                + složka + "\" neexistuje nebo není složkou" );
        this.složka = složka;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Nad všemi soubory své složky a jejích podsložek provede zadanou operaci.
     * @param příkaz Objekt realizující zadanou operaci
     */
```

```
    public void aplikuj( IPříkaz příkaz )
    {
        for( File soubor : this )
            příkaz.příkaz( soubor );
    }


    /*****************************************************************************
     * Iterátor, který bude procházet všechny soubory v dané složce
     * a jejích podsložkách.
     * @return Požadovaný iterátor
     */
    public Iterator<File> iterator()
    {
        return new FileIterator( filtr, komparátor );
    }


    /*****************************************************************************
     * Nastaví komparátor ovlivňující řazení vracených souborů.
     * @param comparator Nastavovaný komparátor
     */
    public void setComparator( Comparator<File> comparator )
    {
        this.komparátor = comparator;
    }


    /*****************************************************************************
     * Nastaví filtr ovlivňující, které soubory bude iterátor vracet.
     * @param filtr Nastavovaný filtr
     */
    public void setFilter( FileFilter filtr )
    {
        this.filtr = filtr;
    }


    /*****************************************************************************
     * Převede instanci na řetězec.
     * @return Řetězcová reprezentace dané instance.
     */
    public String toString()
    {
        return složka.toString();
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /*****************************************************************************
     * Instance třídy FileIterator představují iterátory procházející
     * adresářový strom do hloubky.
     */
    private class FileIterator implements Iterator<File>
    {
    //== KONSTANTNÍ ATRIBUTY INSTANCÍ ==========================================
```

```
    /** Seznam rozpracovaných složek. */
    private Stack<Přepravka> rozpracované = new Stack<Přepravka>();

    /** Případný filtr vracených souborů. */
    private FileFilter filtr;

    /** Případný comparátor pro vracení souborů v zadaném pořadí. */
    private Comparator<File> komparátor;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Další soubor na řadě. */
    private File další = null;

    /** Příznak toho, zda byl již dotázán hasNext(). */
    private boolean dotázáno = false;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /**************************************************************************
     * Vytvoří nový iterátor, který bude pracovat se zadaným filtrem
     * a komparátorem, jež budou v průběhu iterace nezměnitelné.
     * @param filtr Nastavovaný filtr
     * @param komparátor Nastavovaný komparátor
     */
    public FileIterator( FileFilter filtr, Comparator<File> komparátor )
    {
        rozpracované.push( new Přepravka( složka, komparátor ) );
        //Zapamatuje se aktuální filtr a komparátor pro případ,
        //že by je někdo v průběhu iterace změnil.
        this.filtr      = filtr;
        this.komparátor = komparátor;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * {@inheritDoc}
     * @return Informace o existenci dosud nepředaného souboru
     */
    public boolean hasNext()
    {
        if( !dotázáno ) {
            další    = dalšíSoubor();
            dotázáno = true;
        }
        return další != null;
    }


    /**************************************************************************
     * {@inheritDoc}
     * @return Další soubor
```

```
         */
        public File next()
        {
            File ret = dotázáno  ?  další  :  dalšíSoubor();
            dotázáno = false;
            if( ret == null )
                throw new NoSuchElementException();
            return ret;
        }


        /**************************************************************************
         * {@inheritDoc}
         */
        public void remove()
        {
            throw new UnsupportedOperationException();
        }

    //== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ====================================

        /**************************************************************************
         * Vrátí další soubor v rozpracované složce.
         * Je-li dalším "souborem" složka, začne procházet její soubory.
         * Po vyčerpání souborů aktuální složky se vrátí do jejího rodiče
         * a bude pokračovat v procházení jejích souborů.
         * Po projití všech souborů vrátí <code>null</code>.
         */
        private File dalšíSoubor()
        {
            if( rozpracované.empty() )  //Složka je projita
                 return null;
            //Podívej se, kde jsi posledně skončil
            Přepravka aktuální = rozpracované.peek();
            for(;;) {
                if( aktuální.index >= aktuální.soubory.length ) {
                    //Naposledy procházená složka je projita -
                    //budeme pokračovat v procházení její rodičovské složky
                    rozpracované.pop();
                    return dalšíSoubor();
                }
                //Podíváme se na další soubor v aktuální složce
                File soubor = aktuální.soubory[ aktuální.index++ ];
                if( soubor.isFile() ) {
                    //Je to soubor - ještě jestli nás zajímá
                    if( (filtr != null)  &&  !filtr.accept( soubor ) )
                        continue;   //Nezajímal nás, jdeme dál  ^^^^^^^^^
                    return soubor;  //==========>
                }
                //Soubor se ukázal býti složkou - začneme jí procházet
                aktuální = new Přepravka( soubor, komparátor );
                rozpracované.push( aktuální );
                continue;           //^^^^^^^^^
            }
        }
```

```
    //== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

        /*****************************************************************************
         * Přepravka obsahuje pole všech souborů jedné složky spolu s indexem
         * označujícím soubor, se kterým bude příště pracovat.
         */
        private class Přepravka
        {
            File[] soubory;     //Soubory nacházející se v příslušné složce
            int    index;       //Index souboru, který bude zpracováván příště

            /*************************************************************************
             * @param složka Složka, jejíž soubory si instance bude pamatovat
             * @param komparátor Je-li zadán, budou soubory v poli seřazeny
             *                   podle pravidla definovaného tímto komparátorem
             */
            Přepravka( File složka, Comparator<File> komparátor ) {
                soubory = složka.listFiles();
                if( komparátor != null )
                    Arrays.sort(soubory, komparátor );
                index   = 0;
            }
        }//Konec třídy Přepravka

    }//Konec třídy FileIterator


//== TESTY A METODA MAIN ======================================================

     /** @param ppr Parametry příkazového řádku - nepoužité */
     public static void main(String[]ppr){
         SložkaTest.test();
     }/*-*/
}
```

279. Co je na tom testu tak složitého, žes' jej rozděloval?

Co se testuje:
– vlastní iterace
– řazení výstupu
– filtrace
– skrytá iterace

Připravil jsem samostatnou testovací metodu pro test různých režimů práce:

- pro prostou iteraci,
- pro iteraci vracející soubory v zadaném pořadí,
- pro filtrovanou iteraci předávající pouze soubory vyhovující zadané podmínce,
- pro „skrytou iteraci", při níž předáš metodě objekt, jehož metoda bude aplikována na všechny položky iterovaného objektu.

Každá z testovacích metodu vypisuje seznam procházených metod na standardní výstup, kde si můžeš prohlédnout, jak se výstup postupnou aplikací modifikačních metod mění.

Testovací třída

Výpis 16.5: Definice třídy `SložkaTest`, která otestuje jednotlivé možnosti třídy `Složka`

```
package rup.česky.vzory._16_iterátor;

import java.io.File;
import java.io.FileFilter;
```

```
import java.util.Comparator;

/*******************************************************************************
 * Instance třídy Složka představují iterovatelné pseudokontejnery
 * simulující, že obsahují soubory v zadané složce a všech jejích podsložkách.
 * Jsou schopny vrátit iterátor, který postupně všechny tyto soubory projde
 * a nabídne.
 */
public class SložkaTest
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Implicitní složka, kterou bude metoda test(String) procházet,
     *  nebude-li v příkazovém řádku zadána nějaká jiná. */
    private static final String SLOŽKA = "S:/";


//== NESOUKROMÉ METODY TŘÍDY ===================================================

    /*******************************************************************************
     * Testuje, zda iterátor projde všechny soubory
     * v zadané složce a jejích podložkách.
     * @param složka Procházená složka
     */
    public static void test()
    {
        Složka ss = new Složka( "S:/" );
        testČistéIterace( ss );
        testSeřazenéIterace( ss );
        testFiltrovanéIterace( ss );
        testIterovanéAkce( ss );
    }


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /** Soukromý konstruktor bránící vytvoření instance. */
    private SložkaTest() {}


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ===========================================

    /*******************************************************************************
     * Testuje, zda iterátor projde všechny soubory
     * v zadané složce a jejích podložkách.
     */
     private static void testČistéIterace( Složka ss )
     {
         System.out.println("\n=============== testČistéIterace");
         for( File f : ss ) {
             System.out.println( f.getAbsolutePath() );
         }
     }


    /*******************************************************************************
     * Testuje, zda iterátor projde všechny soubory
```

```
     * v zadané složce a jejích podložkách.
     */
    private static void testSeřazenéIterace( Složka ss )
    {
        System.out.println("\n============== testSeřazenéIterace");
        ss.setComparator( new
            Comparator<File>() {
                public int compare( File soubor1, File soubor2 ) {
                    String ap1 = soubor1.getAbsolutePath();
                    String ap2 = soubor2.getAbsolutePath();
                    int delka = ap1.length() - ap2.length();
                    if( delka != 0 )   return delka;
                    return ap1.compareToIgnoreCase( ap2 );
                }
            }
        );
        for( File f : ss ) {
            System.out.println( f.getAbsolutePath() );
        }
    }


    /*************************************************************************
     * Testuje, zda iterátor projde všechny soubory
     * v zadané složce a jejích podložkách.
     */
    private static void testFiltrovanéIterace( Složka ss )
    {
        System.out.println("\n============== testFiltrovanéIterace");
        ss.setFilter( new
            FileFilter() {
                public boolean accept( File soubor ) {
                    return soubor.getName().endsWith(".doc");
                }
            }
        );
        for( File f : ss ) {
            System.out.println( f.getName() + " - " + f.getAbsolutePath() );
        }
    }


    /*************************************************************************
     * Testuje, zda iterátor projde všechny soubory
     * v zadané složce a jejích podložkách.
     */
    private static void testIterovanéAkce( Složka ss )
    {
        System.out.println("\n============== testIterovanéAkce");
        final long[] suma = new long[1];
        ss.aplikuj( new
            IPříkaz() {
                public void příkaz( Object... o ) {
                    File soubor = (File)o[0];
                    long délka = soubor.length();
                    System.out.printf( "%s - %,d bajtů\n",
                                       soubor.getName(), délka );
```

```
                    suma[0] += délka;
                }
            }
        );
        System.out.printf("\nCelková délka vybraných souborů: %,d bajtů\n\n",
                          suma[0] );
    }


//== TESTY A METODA MAIN ===================================================

    /** @param ppr Parametry příkazového řádku - procházená složka */
    public static void main(String[]ppr){
        String složka;
        if( ppr.length > 0)
            složka = ppr[0];
        else
            složka = SLOŽKA;
        test( složka );
    }/*-*/
}
```

Prázdný iterátor a iterovatelný objekt

280. Tak tento příklad byl opravdu trochu složitější. Bude mi chvilku trvat, než jej pochopím a vstřebám.

Mám pro tebe ještě druhou krajnost: příklad tak jednoduchý, že už jednodušší snad ani být nemůže.

Popis problému

Občas se stává, že potřebuješ vrátit iterátor, který nemá nad čím iterovat, protože příslušný objekt je prázdný, nebo dokonce ani neexistuje. Řada programátorů vrací v takovém případě prázdný odkaz `null`. Jak jsme si ale říkali v kapitole *I nic může být objekt (Prázdný objekt – Null Object)* na straně 97, tato praxe nemusí být vždy nejvýhodnější. Často bývá mnohem výhodnější definovat prázdný objekt, který je sice řádným iterátorem, ale jeho iterace skončí dřív, než doopravdy začne.

Definici takovéhoto prázdného iterátoru najdeš ve výpisu 16.6. Třída `PrázdnýIterátor` je definována jako potomek třídy `IteratorAdapter`, o které jsme již hovořili a jejíž definici najdeš ve výpisu 16.2 na stránce 209.

Aby prázdnému iterátoru nebylo smutno, přidal jsem mu hned bratříčka: prázdný iterovatelný objekt, který využiješ tehdy, nemáš-li nic a máš-li toto nic vrátit jako iterovatelný objekt. Jeho definici najdeš ve výpisu 16.7. (Tento objekt výhodně použijeme v doprovodných příkladech ke kapitole *Já to umím, upřesni jen detaily (Šablonová metoda – Template Method)*, která začíná na straně 239.)

Výpis 16.6: Definice třídy `PrázdnýIterátor`

```
package rup.česky.vzory._16_iterátor;

import java.util.NoSuchElementException;

/*****************************************************************************
 * Jediná instance třídy PrázdnýIterátor představuje iterátor
```

```
 * iterující přes prázdný kontejner.
 *
 * @param <E> Typový parametr označující typ prvků iterovaného kontejneru.
 *            Vzhledem k prázdnosti objektu je zde jenom formálně.
*/

public class PrázdnýIterátor<E> extends IteratorAdapter<E>
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final PrázdnýIterátor jediáček = new PrázdnýIterátor();


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /****************************************************************************
     * Tovární metoda vracející odkaz na svého jediáčka.
     */
    public static <T> PrázdnýIterátor<T> getInstance() {
        return jediáček;
    }


//== NESOUKROMÉ METODY INSTANCÍ ===============================================

    /****************************************************************************
     * {@inheritDoc}
     */
    public boolean hasNext() {
        return false;
    }

    /****************************************************************************
     * {@inheritDoc}
     */
    public E next() {
        throw new NoSuchElementException(
            "Prázdný iterátor iteruje přes prázdný kontejner" );
    }
}
```

Výpis 16.7: Definice třídy `PrázdnýIterable`

```
package rup.česky.vzory._16_iterátor;

import java.util.Iterator;

/*******************************************************************************
 * Instance třídy PrázdnýIterable představují ...
 *
 * @param <E> Typy položek iterovatelného kontejneru - vzhledem k prázdnosti
 *            objektu je zde tento parametr pouze formální.
*/
public class PrázdnýIterable<E> implements Iterable<E>
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================
```

```
    private static final PrázdnýIterable jedináček = new PrázdnýIterable();


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Tovární metoda vracející odkaz na svého jedináčka.
     */
    public static <T> PrázdnýIterable<T> getInstance() {
        return jedináček;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Prázdný iterovatelný objekt vrací i prázdný iterátor.
     * @return x
     */
    public Iterator<E> iterator() {
        return PrázdnýIterátor.getInstance();
    }
}
```

Shrnutí – co jsme se naučili

- Iterátor zprostředkuje jednoduchý a přehledný způsob sekvenčního přístupu k objektům uloženým v nějaké složité struktuře, přičemž tato struktura zůstane klientovi skryta.
- Iterátory typicky zprostředkovávají přístup k položkám uloženým v kontejneru, ale zdrojem položek mohou být i jiné druhy objektů.
- Iterátor pracuje jako specializovaný zástupce zprostředkující přístup, ale zakrývající implementaci.
- Iterátor bývá definován jako soukromá vnitřní třída.
- Iterátor typicky poskytuje metody ke zjištění přítomnosti dosud nezpracovaného prvku a poskytnutí odkazu na tento prvek.
- Iterátor může poskytovat další metody pro:
  - ovlivnění průběhu iterace (směr, filtr, řazení, …),
  - změnu obsahu iterované struktury,
  - práci s iterovanými prvky
  - a další.
- Iterátor může iterovat i skrytě. Stačí, když dostane objekt, jehož metodu má zavolat a předat jí každý z objektů, resp. má-li zavolat definovanou metodu všech uložených objektů.

KAPITOLA 17

Příliš mnoho rozhodování (Stav – State)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Řeší výrazný rozdíl mezi chováním objektu v různých stavech zavedením vnitřního stavu jako objektu reprezentovaného instancí některé ze stavových tříd. Změnu stavu objektu pak řeší záměnou objektu reprezentujícího stav.

[1] **Definice v GoF:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. – Umožní objektu měnit své chování při změně vnitřního stavu. Objekt pak vypadá, jako by měnil svoji třídu.

Účel

281. Co si mám představovat pod návrhovým vzorem *Stav*? Stav přece není objekt, stav je stav.

Stav může být objekt

Smiř se s tím, že v objektově orientovaném programování je všechno objekt. Tedy i stav může být reprezentován objektem.

282. No dobře, a k čemu mi to bude dobré?

K čemu reprezentovat stav třídou

Jsou situace, kdy může objekt nabývat několika různých stavů a reakce objektu v jednotlivých stavech se dost výrazně liší. Pak bývá výhodné definovat pro každý stav objektu jeho vlastní reprezentaci. Definice metod se tím výrazně zjednoduší a většinou se zjednoduší i případné další modifikace.

283. Zkus to říct trochu konkrétněji a přidej nějaký příklad.

Příklad autíčka

Dobrá. Představ si grafický objekt reprezentující autíčko, které je schopno zatáčet o 90°. Když se nad tím trochu zamyslíš, tak takové autíčko se chová výrazně jinak, když jede na východ, na sever, na západ či na jih. Podle směru, do nějž je natočeno, také bude záviset i způsob jeho otočení.

284. Proč by jeho chování mělo záviset na směru? Při popojetí se přesune vždy o jedno pole dopředu a při otočení se otočí o 90° požadovaným směrem.

Odlišné chování nasměrovaných aut

Zapomněl jsi na to, že při natočení do každého směru vypadá auto trochu jinak, a musí se tedy jinak kreslit. Popojetí dopředu závisí na natočení, protože od něj se odvozuje, jak se bude počítat cílová souřadnice popojíždějícího autíčka. Různě se chová i při zatáčení: auto jedoucí na sever bude po otočení vlevo vypadat jako auto jedoucí na západ, kdežto auto jedoucí na východ bude po otočení vlevo vypadat jako auto jedoucí na sever.

285. Přiznám se, že tohle mne v první chvíli nenapadlo. Přesto v tom nevidím žádný problém. Prostě vložím do programu sadu podmínek nebo přepínač, který podle aktuálního směru natočení auta vybere tu správnou větev, v níž bude definováno, jak se má auto chovat.

Proč nejsou vhodná klasická rozhodování

No, a právě v tom je ten problém. Program pak bude plný podmíněných příkazů či přepínačů, které jej znepřehledňují. Jednotlivé metody budou díky těmto rozhodováním dlouhé, takže se v nich budou snáze dělat a hůře hledat chyby. Navíc budou vznikat dlouhé metody, které celou myšlenku, na níž je program postaven, pouze znepřehlední.

Snazší přidávání dalších stavů

Navíc se počet stavů může v průběhu života programu měnit. U autíčka bychom se např. mohli rozhodnout, že je naučíme jezdit také úhlopříčně. Přidání dalšího stavu by znamenalo projít celý kód a zanést na řadu míst řadu oprav. A jak jistě víš, podle jednoho z programátorských axiomů se každou opravou do programu zanese nejméně jedna chyba.

Implementace

286. Uznávám, že díky přepínačům budou tyto metody maličko delší. Nevím ale, jak bys to chtěl dělat jinak?

Rozdělení objektu na více částí

Právě aplikací návrhového vzoru stav. Při ní se původní mnohastavová třída rozdělí na několik tříd:

- na třídu, jejíž instance budou představovat původní mnohastavový objekt,
- na řadu „jednostavových" tříd pro reprezentaci jednotlivých stavů.

Tím, že se každá z těchto jednostavových tříd bude starat o implementaci chování pouze jediného stavu, bude její definice výrazně jednodušší a tím i méně náchylná ke vzniku chyb.

Přidání dalšího stavu

Přidání dalšího stavu pak spočívá především v přidání další „jednostavové" třídy. Ostatní třídy vícestavového objektu většinou ovlivní jen minimálně.

287. Připomíná mi to muší váhu – tam se informace o stavu objektu rozdělovaly na vnější a vnitřní, tady na stavově závislé a stavově nezávislé.

Podobnost s muší váhou

Máš pravdu, jistá podobnost by tu byla. V návrhovém vzoru *Stav* je ale to rozdělení na vnější a vnitřní stav interní, implementační záležitostí, o které by okolí volající vícestavový objekt vůbec nemělo vědět.

Doporučený postup:

288. Dobře. Jak bych tedy měl takový vícestavový objekt naprogramovat?

Postup je jednoduchý:

– charakteristika stavů

1. Rozmyslíš si, které vlastnosti definovaných objektů závisí výrazně na stavu, v němž se objekt nachází.

– stavové rozhraní

2. Definuješ rozhraní, jehož instance budou představovat ony stavově závislé části oněch vícestavových objektů. V něm deklaruješ metody odpovídající zprávám, na které reagují instance vícestavové třídy rozdílně v závislosti na svém stavu. V dalším textu o něm budu hovořit jako o *stavovém rozhraní*.

– jednostavové třídy

3. Pro každý stav definuješ speciální „jednostavovou" třídu, která bude implementovat stavové rozhraní a definovat chování svých instancí ve stavu, jenž bude touto třídou, resp. jejími instancemi, zastupován.

– atribut stavu

4. V definici třídy vícestavového objektu deklaruješ atribut odkazující na instanci stavového rozhraní.

– inicializace atributu

5. V konstruktoru vícestavového objektu zařídíš, aby byl tento atribut inicializován odkazem na objekt, jehož typ odpovídá výchozímu stavu vytvářené instance.

289. Tak mne napadá, že ty jednostavové třídy budou mít nejspíš stejné atributy. Možná by mohly mít i společné pomocné metody. Počítá ten návrhový vzor také s tím, že bych místo stavového rozhraní použil stavovou třídu?

Nahrazení stavového rozhraní třídou

Samozřejmě. Ve funkci stavového rozhraní může vystupovat nejenom `interface`, ale také abstraktní třída. Čemu dáš přednost, to je otázka implementace a mohli bychom říci, že to vícestavovou třídu nezajímá. Ona s tvými objekty komunikuje stejně jenom

prostřednictvím deklarovaného rozhraní. Jestli je to rozhraní abstraktní třídy nebo opravdu `interface`, je pro ni nezajímavé. Pro ni to bude stavové rozhraní.

290. Říkal jsi, že jako stavové rozhraní mohu použít abstraktní třídu. Proč by to nemohla být konkrétní třída?

Proč abstraktní

Stavové rozhraní je reprezentantem jednostavových tříd, které definují reakce na zprávy, na něž je třeba v různých stavech reagovat různě. Protože jejich společný rodič nemůže vědět, jak bude která z nich reagovat, musí příslušné metody deklarovat jako abstraktní.

Společné metody všech stavů

Definuje-li jejich společný rodič nějaké metody, které jsou pro všechny tyto třídy společné, pak se jedná pouze o pomocné metody usnadňující implementaci, protože jinak bychom je mohli definovat již ve vícestavové třídě a nemuseli bychom si komplikovat život zaváděním jednostavových tříd a jejich společného rodiče.

291. A co když budou mít některé stavy nějakou metodu společnou?

Společné metody několika stavů

I tak bych považoval za čistší definovat danou metodu ve společném rodiči jako abstraktní. Budou-li mít nějakou metodu společnou, můžeš ji v rodiči definovat jako pomocnou metodu, kterou budou metody potomků volat. V opačném případě totiž zavádíš mezi třídami další vazby, na které bys mohl při pozdějších úpravách zapomenout, a to by se ti mohlo vymstít.

292. Z toho, co jsi říkal, mi prozatím není jasné, kdo má na starosti přepínání mezi stavy.

Přepínání mezi stavy

To záleží na konkrétní aplikaci. Někdy může změnu stavu dostatečně jednoduše rozpoznat majitel, tj. instance vícestavové třídy. Jindy je naopak výhodnější ponechat specifikaci nového stavu na jednostavových instancích, jejichž metody budou rovnou vracet odkaz na instanci, reprezentující nový stav.

293. A kde tu instanci vezmou?

Tvorba jednostavových instancí

To opět záleží na konkrétní aplikaci a především na ceně vytvoření instance. Je-li vytvoření instance relativně levnou záležitostí, může metoda jednostavové instance vytvořit novou instanci požadovaného stavu. Je-li vytváření instancí dražší, můžeš definovat nějaký interní fond, ze kterého se požadovaná instance vezme.

294. Ještě mne napadá, jak bych měl postupovat v případě, budou-li jednotlivé stavy potřebovat pracovat s nějakými atributy a metodami vícestavového objektu.

Vícestavový objekt jako parametr jednostavových metod

Teoreticky bys sice mohl definovat jednotlivé stavové třídy jako vnitřní, ale tím bys celou definici spíš zesložitil. Ve většině případů však nepotřebuješ přistupovat k soukromým atributům a metodám vícestavového objektu tak intenzivně, abys musel definovat jednostavové třídy jako vnitřní, a bohatě vystačíš s tím, že metodám jednostavových instancí budeš předávat příslušnou vícestavovou instanci jako parametr.

Příklad

295. Řekl bych, že už jsme vzor dostatečně rozebrali a že je pomalu čas ukázat si jeho fungování na příkladu.

Ukážu ti použití vzoru na příkladu oněch autíček, o nichž jsme hovořili na počátku kapitoly. Projekt je rozdělen do tří balíčků:

- balíček `autoi` obsahuje definice, v nichž jednostavové třídy nejsou potomky nějaké abstraktní třídy, ale pouze implementují rozhraní `IAuto1`,
- balíček `autoa` obsahuje definice, v nichž jsou jednostavové třídy potomky abstraktní třídy `AAuto1`,
- balíček `závod` definuje třídy, které umožňují testovat provozuschopnost vytvořených autíček a případně i s těmito autíčky závodit.

V následujících výpisech ukážu pouze třídy z balíčku `autoa`. Třídy v balíčku `autoi` jsou jim velice podobné, pouze jsou trochu „ukecanější". Třídy v balíčku `závod` neimplementují náš návrhový vzor, takže si povíme pouze o jejich koncepci, a pokud by ses s nimi chtěl seznámit důvěrněji, podíváš se do doprovodných programů.

296. Takže asi začneme tím závodem, abychom se včas dozvěděli, pro co budeme naše třídy připravovat.

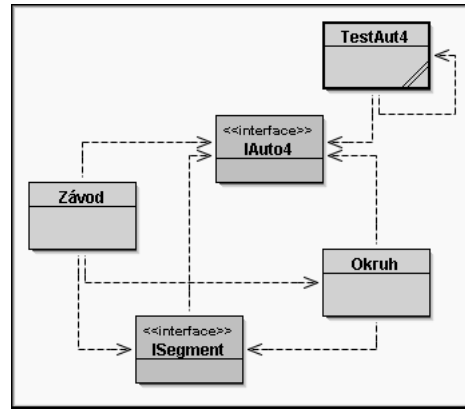
Jasně. Nejprve bych ti prozradil, že se závodí na okruzích složených ze segmentů. Segmenty mají dvě skupiny vlastností:

- ty, které využívají auta, která po nich jedou, a
- ty, které využívá třída `Okruh` při budování závodního okruhu.

Aby závodící auta nemohla využívat „budovacích" vlastností a přestavovat si v průběhu závodu trať, je třída `Segment` definována jako soukromá vnitřní třída třídy `Okruh` a autíčka dostanou k dispozici *zástupce* implementujícího rozhraní `ISegment`.

Odpoutejme se ale od budování okruhu a jeho zapouzdřování a podívejme se na balíček `závod` celkově. Tento balíček obsahuje (viz obr. 17.1):

- rozhraní `IAuto4`, které deklaruje požadované vlastnosti aut, jež jsou schopna jezdit kterýmkoliv ze čtyř hlavních směrů,
- rozhraní `ISegment`, které zde funguje jako zástupce, jenž zpřístupňuje autům pouze ty schopnosti třídy `Segment`, které musí mít auta dostupné, aby mohla úspěšně jezdit po okruhu,
- třídu `Okruh`, která má na starosti vybudování a provoz závodního okruhu,
- třídu `TestAut4`, která testuje schopnost aut implementujících rozhraní `IAuto4` jezdit správně po vybudovaném okruhu a ukazuje, jak jej jsou auta schopna objet,
- třídu `Závod`, umožňující vyzkoušet ruční řízení auta typu `IAuto4` pomocí kurzorových kláves a zkusit si projet pár okruhů na čas.

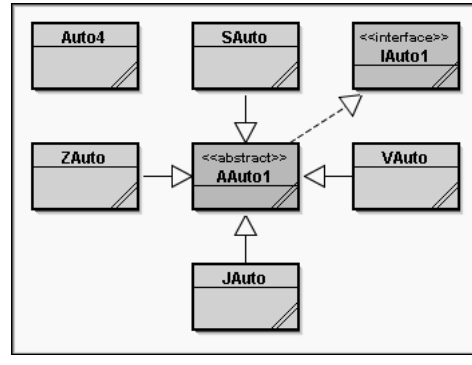


Obrázek 17.1
Diagram tříd v balíčku `závod`

297. To bych asi pochopil (a možná se ti časem na ty třídy i podívám). Teď by mne ale zajímala ta závodící auta.

Jak už jsem řekl, vzorová verze „čtyřsměrných" autíček využívajících abstraktní třídu jako společného rodiče svých „jednosměrných" stavů je v balíčku `autoa`. Najdeš v něm (viz obr. 17.2):

- třídu `Auto4`, implementující rozhraní `IAuto4` z balíčku `závod` (její instance budou ta autíčka, která budou jezdit po okruhu a případně i závodit),
- rozhraní `IAuto1`, deklarující požadavky na jednostavové třídy (to by tu sice být nemuselo, ale ponechal jsem je zde, a jak za chvíli uvidíš, atribut třídy `Auto4` se odkazuje na ně),
- abstraktní třídu `AAuto1`, která je společným rodičem všech jednostavových tříd a obsahuje společné implementace,
- třídy `VAuto`, `SAuto`, `ZAuto` a `JAuto`, jejichž instance jsou ty jednostavové podobjekty mající na starosti stavově závislé chování auta jedoucího ve směru, jehož zkratkou začíná název příslušného jednostavového typu.



Obrázek 17.2
Diagram tříd v balíčku `autoa`

298. Diagramy jsem pochopil. Teď ještě ukaž kód.

Už se nese. Ve výpisech ale nenajdeš všechny jednostavové třídy, ale pouze třídu `VAuto`, jejíž instance obstarávají stavově závislé vlastnosti autíček jedoucích na východ. Kromě toho jsem přidal definici rozhraní `IAuto4` z balíčku `závod`.

Výpis 17.1: Definice rozhraní `IAuto4`

```
package rup.česky.vzory._17_stav.závod;

import rup.česky.tvary.IPosuvný;
import rup.česky.tvary.Směr8;


/*******************************************************************************
 * Rozhraní IAuto popisuje vlastnosti aut, která se umí natočit do čtyř
 * základních světových stran.
*/
public interface IAuto4 extends IPosuvný
{
//== OSTATNÍ METODY K IMPLEMENTACI ============================================

    /***************************************************************************
     * Vrátí směr, do nějž je auto natočeno.
     * @return Směr, do nějž je auto natočeno
     */
    public Směr8 getSměr();


    /***************************************************************************
     * Popojede o jedno políčko ve směru, do nějž je natočeno.
     */
    public void popojeď();


    /***************************************************************************
     * Auto se otočí o 90° doleva.
     */
    public void vlevoVbok();


    /***************************************************************************
     * Auto se otočí o 90° doleva.
     */
    public void vpravoVbok();

}
```

Toto rozhraní je implementováno čtyřrozměrným autíčkem naprogramovaným s využitím návrhového vzoru *Stav*.

Výpis 17.2: Definice třídy `Auto4`, jejímiž instancemi jsou závodící autíčka

```
package rup.česky.vzory._17_stav.autoa;

import rup.česky.tvary.SprávcePlátna;
import rup.česky.tvary.IMultiposuvný;
```

```
import rup.česky.tvary.Kreslítko;
import rup.česky.tvary.Pozice;
import rup.česky.tvary.Směr8;

import rup.česky.vzory._16_stav.závod.IAuto4;
import rup.česky.vzory._16_stav.závod.TestAut4;

/*******************************************************************************
 * Třída Auto4 vytvoří oválné auto schopné jezdit do všech čtyř směrů.
 */
public class Auto4 implements IAuto4, IMultiposuvný
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private final static SprávcePlátna SP = SprávcePlátna.getInstance();


//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================================
    private static int počet = 0;
//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================================
    private final int pořadí = ++počet;

    /*******************************************************************************
     * {@inheritDoc}
     */
    @Override
    public String toString()
    {
        Pozice p = getPozice();
        return "Auto " + pořadí + ", x=" + p.x + ", y=" + p.y + ", směr=" +
               getSměr();
    }


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ===============================================

    /** Instance mající na starosti stavově závislou část chování. */
    private IAuto1   auto;


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /*******************************************************************************
     * Vytvoří novou instanci s implicitními rozměry a umístěním.
     * Instance bude umístěna v levém horním rohu plátna
     * a bude umístěna do čtverce o straně 8*SP.getKrok() bodů.
     */
    public Auto4()
    {
        this(Směr8.VÝCHOD, 0, 0);
    }


    /*******************************************************************************
     * Vytvoří novou instanci s implicitními rozměry a umístěním.
     * Instance bude umístěna v levém horním rohu plátna
     * a bude umístěna do čtverce o straně 8*SP.getKrok() bodů.
     *
```

```
     * @param směr  Směr, do nějž má být auto natočeno
     */
    public Auto4( Směr8 směr )
    {
        this(směr, 0, 0);
    }


    /*****************************************************************************
     * Vytvoří novou instanci s implicitními rozměry.
     * Instance bude umístěna na zadaných souřadnicích
     * a bude umístěna do čtverce o straně 8*SP.getKrok() bodů.
     *
     * @param x  x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna
     * @param y  y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna
     */
    public Auto4( int x, int y )
    {
        this( Směr8.VÝCHOD, x, y );
    }


    /*****************************************************************************
     * Vytvoří novou instanci s implicitními rozměry.
     * Instance bude umístěna na zadaných souřadnicích
     * a bude umístěna do čtverce o straně 8*SP.getKrok() bodů.
     *
     * @param x     x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna
     * @param y     y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna
     * @param směr  Směr, do nějž má být auto natočeno
     */
    public Auto4( Směr8 směr, int x, int y )
    {
        auto = AAuto1.getAuto1(směr, x, y );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * @return Směr, co nějž je auto natočeno.
     */
    public Směr8 getSměr()
    {
        return auto.getSměr();
    }


    /*****************************************************************************
     * Popojede o jedno políčko ve směru, do nějž je natočeno.
     */
    public void popojeď()
    {
        auto.vpřed();
        SP.překresli();
    }
```

```
    /*****************************************************************************
     * Auto se otočí o 90° doleva.
     */
    public void vlevoVbok()
    {
        auto = auto.doleva();
        SP.překresli();
    }


    /*****************************************************************************
     * Auto se otočí o 90° doleva.
     */
    public void vpravoVbok()
    {
        auto = auto.doprava();
        SP.překresli();
    }


    /*****************************************************************************
     * Vrátí aktuální pozici auta.
     * @return Aktuální pozice
     */
    public Pozice getPozice()
    {
        return auto.getPozice();
    }


    /*****************************************************************************
     * Nastaví novou bodovou pozici instance.
     *
     * @param x   Nová vodorovná bodová pozice instance
     * @param y   Nová svislá bodová pozice instance
     */
    public void setPozice( int x, int y )
    {
        auto.setPozice( x, y );
    }


    /*****************************************************************************
     * {@inheritDoc}
     */
    public void setPozice( Pozice pozice )
    {
        setPozice( pozice.x, pozice.y );
    }


    /*****************************************************************************
     * Metoda definuje reakci na ukončení přesunu multipřesouvačem.
     */
    public void přesunuto()
    {
        TestAut4.další( this );
    }
```

```
    /****************************************************************************
     * Vykreslí dodaným kreslítkem obraz své instance na plátno.
     * @param kr Dodané kreslítko
     */
    public void nakresli( Kreslítko kr )
    {
        auto.nakresli( kr );
    }


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ============================================
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ =========================================
//== VNOŘENÉ A VNITŘNÍ TŘÍDY ====================================================
//== TESTY A METODA MAIN ========================================================

    /****************************************************************************
     * Otestuje kostrbatý přesun mého auta po zadané dráze
     */
    public static void test()
    {
        //První parametr zadává velikost modulu testovacího okna
        //Druhý parametr zadává úplný název testované třídy včetně balíčku
        TestAut4.test( 5, "auta.autoARP.Auto4" );
    }

}
```

Požadované vlastnosti jednostavových podobjektů definuje rozhraní `IAuto1`. Všimni si, jak je vyřešena otázka přechodu mezi stavy: autíčko, které se má otočit, vrátí místo sebe jiné autíčko, které je otočené do správného směru.

Výpis 17.3: Rozhraní `IAuto1` definuje požadované vlastnosti jednostavových podobjektů

```
package rup.česky.vzory._17_stav.autoa;

import rup.česky.tvary.Směr8;


/*******************************************************************************
 * Rozhraní IAuto4 definuje povinnou sadu metod aut,
 * která vystupují jako instance jednostavových tříd.
 */
public interface IAuto1 extends rup.česky.tvary.IPosuvný
{
//== VEŘEJNÉ KONSTANTY ==========================================================
//== PŘÍSTUPOVÉ METODY ATRIBUTŮ INSTANCÍ ========================================
//== OSTATNÍ METODY K IMPLEMENTACI ==============================================

    /****************************************************************************
     * Vrátí směr, do nějž je auto natočeno.
     * @return Směr, do nějž je auto natočeno
     */
    public Směr8 getSměr();
```

```
    /*****************************************************************************
     * Popojede o jedno políčko ve směru, do nějž je natočeno.
     */
    public void   vpřed();


    /*****************************************************************************
     * Vrátí instanci umístěnou ve stejné pozici, ale otočenou o 90° vlevo.
     *
     * @return Otočená instance
     */
    public IAuto1 doleva();


    /*****************************************************************************
     * Vrátí instanci umístěnou ve stejné pozici, ale otočenou o 90° vpravo.
     *
     * @return Otočená instance
     */
    public IAuto1 doprava();

}
```

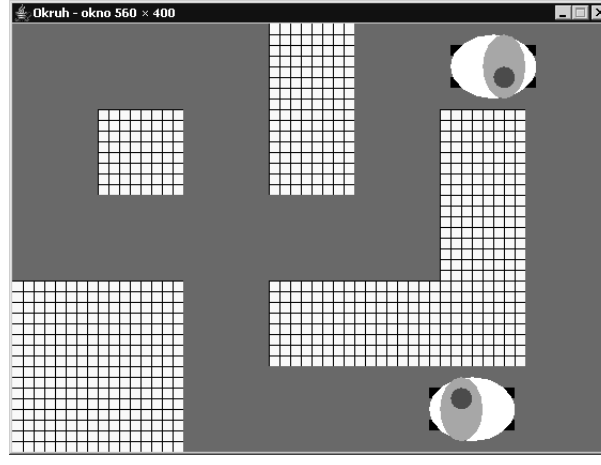
Společným rodičem všech jednostavových aut je třída `AAuto1`, která kromě toho, že sdružuje společné části implementace svých potomků, definuje jednoduchou tovární metodu, jež umožňuje získat odkaz na auto otočené do požadovaného směru.

Výpis 17.4: Třída `AAuto1` je společným rodičem všech jednostavových potomků

```
package rup.česky.vzory._17_stav.autoa;

import rup.česky.tvary.Barva;
import rup.česky.tvary.Elipsa;
import rup.česky.tvary.Kreslítko;
import rup.česky.tvary.Obdélník;
import rup.česky.tvary.Pozice;
import rup.česky.tvary.Směr8;

import static rup.česky.tvary.SprávcePlátna.SP;


/*******************************************************************************
 * Abstraktní třída AAuto1 je společným rodičem všech "jednostavových aut",
 * který mimo jiné poskytuje i společnou implementaci některých metod.
 */
public abstract class AAuto1 implements IAuto1
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    protected static final Barva B_KOL      = Barva.ČERNÁ;
    protected static final Barva B_PODVOZKU = Barva.BÍLÁ;
    protected static final Barva B_KABINY   = Barva.ZLATÁ;
    protected static final Barva B_ŘIDIČE   = Barva.HNĚDÁ;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ===============================================
```

```
    protected int      M;
    protected int      x, y;
    protected Obdélník levé;
    protected Obdélník pravé;
    protected Elipsa   podvozek;
    protected Elipsa   kabina;
    protected Elipsa   řidič;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Pouze inicializuje modul, jenž slouží k výpočtu nejrůznějších posunutí.
     */
    protected AAuto1()
    {
        M = SP.getKrok();
    }


    /*****************************************************************************
     * Vytvoří v počátku souřadnic nové auto otočené do zadaného směru.
     */
    public static AAuto1 getAuto1( Směr8 směr )
    {
        return getAuto1( směr, 0, 0 );
    }


    /*****************************************************************************
     * Vytvoří na zadané pozici nové auto otočené do zadaného směru.
     *
     * @param x  Požadovaná vodorovná souřadnice
     * @param y  Požadovaná svislá souřadnice
     */
    public static AAuto1 getAuto1( Směr8 směr, int x, int y  )
    {
        switch( směr ) {
            case VÝCHOD:   return new VAuto( x, y );
            case SEVER:    return new SAuto( x, y );
            case ZÁPAD:    return new ZAuto( x, y );
            case JIH:      return new JAuto( x, y );
        }
        throw new IllegalArgumentException(
            "Není možno vytvořit auto otočené do směru " + směr );
    }


//== ABSTRAKTNÍ METODY ==========================================================

    /*****************************************************************************
     * Nastaví novou pozici objektu.
     *
     * @param x   Nová x-ová pozice objektu
     * @param y   Nová y-ová pozice objektu
     */
    public abstract void setPozice(int x, int y);
```

```
//== NESOUKROMÉ  METODY INSTANCÍ ==============================================

    /*****************************************************************************
     * {@inheritDoc}
     */
    public Pozice getPozice()
    {
        return new Pozice( x, y );
    }


    /*****************************************************************************
     * {@inheritDoc}
     */
    public void setPozice( Pozice pozice )
    {
        setPozice( pozice.x, pozice.y );
    }


    /*****************************************************************************
     * Vykreslí dodaným kreslítkem obraz své instance na plátno.
     * @param kr Dodané kreslítko
     */
    public void nakresli( Kreslítko kr )
    {
        levé     .nakresli( kr );
        pravé    .nakresli( kr );
        podvozek .nakresli( kr );
        kabina   .nakresli( kr );
        řidič    .nakresli( kr );
    }

}
```



Obrázek 17.3
Autíčka jedoucí po osmičkovém okruhu

Z jednostavových tříd ti ukážu jenom třídu aut jedoucích na východ. Abys věděl, jak mají tato auta vypadat, můžeš si je prohlédnout na obrázku 17.3, na němž je z obrazovky sejmuta podoba jednoduchého okruhu, který otestuje schopnost zatáčení z libovolného směru do libovolného sousedního směru (sám si ale klidně můžeš definovat složitější). Po okruhu jednou dvě autíčka, ale při troše šikovnosti jich na něj můžeš nasázet víc.

Výpis 17.5: Třída `VAuto` je jednou ze čtyř velice podobných jednostavových tříd

```
package rup.česky.vzory._17_stav.autoa;

import rup.česky.tvary.Elipsa;
import rup.česky.tvary.Obdélník;
import rup.česky.tvary.Směr8;

import static rup.česky.tvary.SprávcePlátna.SP;


/*********************************************************************************
 * Instance třídy VAuto představují auta otočená na východ.
 */
public class VAuto extends AAuto1
{
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří na zadané pozici nové auto otočené na východ.
     *
     * @param x  Požadovaná vodorovná souřadnice
     * @param y  Požadovaná svislá souřadnice
     */
    public VAuto( int x, int y )
    {
        levé    = new Obdélník( 0, 0, 8*M,   M, B_KOL      );
        pravé   = new Obdélník( 0, 0, 8*M,   M, B_KOL      );
        podvozek= new Elipsa  ( 0, 0, 8*M, 6*M, B_PODVOZKU );
        kabina  = new Elipsa  ( 0, 0, 4*M, 6*M, B_KABINY   );
        řidič   = new Elipsa  ( 0, 0, 2*M, 2*M, B_ŘIDIČE   );
        setPozice( x*M, y*M );
    }


//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /*****************************************************************************
     * Vrátí směr, do nějž je auto natočeno.
     * @return Směr, do nějž je auto natočeno
     */
    public Směr8 getSměr()
    {
        return Směr8.VÝCHOD;
    }


    /*****************************************************************************
     * Nastaví novou bodovou pozici instance.
     *
     * @param x   Nová vodorovná bodová pozice instance
```

```
     * @param y   Nová svislá bodová pozice instance
     */
    public void setPozice( int x, int y )
    {
        this.x = x;
        this.y = y;
        SP.nekresli();
            levé    .setPozice( x,        y + M*2 );
            pravé   .setPozice( x,        y + M*5 );
            podvozek.setPozice( x,        y + M   );
            kabina  .setPozice( x + M,    y + M   );
            řidič   .setPozice( x + M*2,  y + M*2 );
        SP.vraťKresli();
    }


    /*************************************************************************
     * Popojede o jedno políčko ve směru, do nějž je natočeno.
     */
    public void vpřed()
    {
        setPozice( x+M, y );
    }


    /*************************************************************************
     * Vrátí instanci umístěnou ve stejné pozici, ale otočenou o 90° vlevo.
     * @return Otočená instance
     */
    public IAuto1 doleva()
    {
        return new SAuto( x/M, y/M );
    }


    /*************************************************************************
     * Vrátí instanci umístěnou ve stejné pozici, ale otočenou o 90° vpravo.
     * @return Otočená instance
     */
    public IAuto1 doprava()
    {
        return new JAuto( x/M, y/M );
    }

}
```

299. Není to zbytečné zatěžování správce paměti, když se kvůli každé otočce vytváří nové jednosměrné autíčko a to staré se ruší?

Na to jsem ti už odpovídal. Záleží na aplikaci. V té naší se to neděje tak často, aby si z toho správce paměti musel dělat těžkou hlavu. Jsou ale situace, kdy by to vadit mohlo.

Kdybychom takovou situaci chtěli řešit v naší aplikaci, mohli bychom definovat fond autíček, a pokud bychom ještě neměli vytvořené autíčko v novém směru, vytvořili bychom je. Kdyby se autíčko obrátilo do směru, v němž se již jednou pohybovalo,

použili bychom znovu dříve vytvořenou instanci, jenom bychom ji nejprve přemístili do správné pozice.

300. Ještě by mne zajímalo, jestli by se dal program upravit tak, aby se autíčka nevyměňovala sama, ale aby je vyměňovalo to čtyřstavové auto. Říkal jsi, že se v praxi používají oba přístupy.

Autíčka, která jsem ti ukazoval, jsem dělal s dětmi v době, kdy jsem ještě neměl tak chytrou třídu `Směr8`. Časem jsem do ní přidal další metody, ale auta jsem už neupravoval.

Čtyřstavové auto by mohlo zatáčet opravdu jednoduše. Metodu `vpravoVbok()` bys např. mohl definovat následovně:

```
public void vpravoVbok() {
    auto = AAuto1.getInstance( auto.getSměr().vpravoVbok(),
                               auto.getPozice() );
}
```

Jednostavového auta bych se zeptal na jeho směr, ten bych požádal, aby mi vrátil směr po otočení o 90° doprava, a ten bych předal tovární metodě abstraktní rodičovské třídy aut. Jenom bych současnou podobu té třídy rozšířil o přetíženou verzi tovární metody, která by akceptovala i celou pozici, nejenom její jednotlivé složky.

Shrnutí – co jsme se naučili

- Návrhový *Stav* použijeme tehdy, nacházejí-li se objekty nějaké třídy v různých stavech, které se liší svými reakcemi na zasílané zprávy.
- V definici takovéto vícestavové třídy popisující objekty a jejich chování deklarujeme speciální atribut, který bude zastupovat část objektu, v níž jsou soustředěny stavově závislé vlastnosti objektů.
- Atribut zastupující stavově závislou část objektu můžeme specifikovat jako instanci speciálního stavového rozhraní nebo abstraktní třídy.
- Pro každý stav definujeme speciální jednostavovou třídu implementující stavové rozhraní.
- Ve stavovém rozhraní či abstraktní třídě deklarujeme metody odpovídající zprávám, na něž reagují instance vícestavové třídy rozdílně v závislosti na svém stavu.
- Mezi deklarovanými metodami mohou být i takové, které vracejí odkaz na instanci jiné jednostavové třídy a pomáhají tak realizovat změnu stavu.
- Spolu s výše popsanými metodami musíme někdy deklarovat i další metody, které jsou potřebné pro komunikaci mezi vícestavovou třídou a jejím jednostavovým atributem.
- Je-li to potřeba, může se vícestavový objekt předávat metodám svých jednostavových „kolegů“ jako parametr.
- Návrhový vzor *Stav* patří mezi vzory uvedené v GoF.

KAPITOLA 18

Já to umím, upřesni jen detaily (Šablonová metoda – Template Method)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Definuje metodu obsahující kostru nějakého algoritmu. Ne všechny kroky tohoto algoritmu jsou však v době vzniku šablony známy – jejich konkrétní náplň definují až potomci třídy se šablonovou metodou prostřednictvím překrytí metod, které šablonová metoda volá.

[1] **Definice v GoF:** Define the skeleton of algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. – Definuje kostru algoritmu v operaci delegujíc některé kroky na podtřídy. *Šablonová metoda* umožňuje podtřídám pozměnit některé kroky algoritmu, aniž by měnila jeho strukturu.

Účel

301. Jestli jsem dobře pochopil hlášky v preambuli této kapitoly, tak se tentokrát nebudeme věnovat třídám a jejich spolupráci, ale pouze jedné metodě.

No, není to tak úplně pravda. Ta metoda sice bude klíčovým bodem našeho následujícího rozhovoru, ale bez spolupráce třídy, v níž je metoda definována, se svými potomky by nám k ničemu nebyla.

302. Jako obyčejně na začátku mlžíš. Zkus mi vysvětlit princip a účel tohoto návrhového vzoru nějak názorněji.

A ještě chvíli budu – to abych ti trochu procvičil mozkové závity a hlavně abych tě trochu napnul (ber to jako aplikaci Cimrmanových zásad, jak zařídit, aby si žáci látku lépe pamatovali).

Nejpoužívanější návrhový vzor

Nejprve ti prozradím, že šablonová metoda je jedním z daleko nejpoužívanějších návrhových vzorů. S výkladem jeho principů se dokonce setkáš prakticky ve všech učebnicích objektově orientovaných jazyků, a to dokonce i ve starších učebnicích, jejichž autoři o návrhových vzorech vůbec nic nevěděli[1], či v učebnicích, které najdeš v závěru v seznamu NEdoporučené literatury.

303. Musím se přiznat, že se ti opravdu podařilo mne napnout. Tak už to konečně vybal!

Příklad řešeného problému

Často řešíš několik věcí stejným způsobem, který se pro každou věc liší pouze v drobných detailech.

Půjčím si nápad z [15], kde vysvětluji šablonovou metodu na vaření kávy a čaje. Jistě budeš souhlasit s tím, že příprava obou nápojů má mnoho společného: u obou se musí nejprve uvařit voda, do hrnečku se vloží či nasype extrakt, který se má vyluhovat, pak se tento extrakt zalije vroucí vodou a v případě zájmu se doplní nějaká ochucovadla (mléko, cukr, citrón, rum, …).

Kdybychom definovali přípravu každého nápoje zvlášť, mohla by třída definující přípravu kávy vypadat např. tak, jak ukazuje definice ve výpisu 18.1 (odpusť si poznámky, že ty vaříš kávu trochu jinak). Protože se jedná o pouhý AHA-příklad, vynechal jsem tentokrát v zájmu ušetření místa komentáře u soukromých metod.

Výpis 18.1: Třída `KávaOrig` simuluje přípravu kávy

```
package rup.česky.vzory._18_šablona;

import rup.česky.společně.IO;


/*******************************************************************************
 * Instance třídy KávaOrig představují kávu,
```

[1] Příkladem může být můj vlastní seriál v časopise *ComputerWorld*, který jsem psal na počátku devadesátých let minulého století, pět let před tím, než vyšla GoF. Seriál ani z něj vycházející učebnice v seznamu literatury neuvádím, protože se za těch 15 let pojetí OOP výrazně změnilo a s některými svými tehdejšími tvrzeními již dnes nesouhlasím.

```
 * která se musí umět nejprve připravit.
 */
public class KávaOrig
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final String X = "\n   ";   //Oddělovač výpisů akci


//== NESOUKROMÉ  METODY INSTANCÍ ==============================================

    /*****************************************************************************
     * Připraví hypotetickou kávu, přičemž se uživatele zeptá,
     * chce-li do ní přidat mléko nebo cukr.
     */
    public void připrav()
    {
        vařitVodu();
        připravHrnek();
        kávuDoHrnku();
        zalijVodou();
        přidatMléko();
        přidatCukr();
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    private void vařitVodu() {
        System.out.println( "vařitVodu" +X+
            "Natoč vodu" +X+ "Zapni topení" );
    }


    private void připravHrnek() {
        System.out.println( "připravHrnek" +X+
            "Vezmi hrnek na kávu"  );
    }


    private void kávuDoHrnku() {
        System.out.println( "kávuDoHrnku" +X+
            "Otevři dózu" +X+ "Naber dávku kávy" +X+ "Nasyp kávu do hrnku" +X+
            "Zavři a ukliď dózu" );
    }


    private void zalijVodou() {
        System.out.println( "zalítVodou" +X+
            "Počkej, až se začne vařit" +X+ "Nalej vodu do hrnku" );
    }


    private void přidatMléko() {
        System.out.print( "přidatMléko" );
        if( IO.souhlas( "Mléko?" )  )
            System.out.println( X+"Vezmi mléko" +X+ "Nalej dávku" );
        else
```

```
            System.out.println( X+"Bez cukru" );
    }


    private void přidatCukr() {
        System.out.print( "přidatCukr" );
        if( IO.souhlas( "Osladit?" )  )
            System.out.println( X+"Vezmi cukr" +X+ "Nasyp dávku" );
        else
            System.out.println( X+"Bez mléka" );
    }


//== TESTY ==========================================================================

   public static void test() {
       KávaOrig káva = new KávaOrig();
       káva.připrav();
   }/** @param args Nepoužívané parametry příkazového řádku. */
   public static void main( String... args ) { test(); }
}
```

Řešení v různých třídách jsou velmi podobná

Podíváš-li se na definici podrobněji, bude ti hned jasné, že bys čaj vařil téměř stejně. Rozdíly by byly opravdu drobné. Vynechám-li shodné metody, mohla by definice přípravy čaje vypadat jako program ve výpisu 18.2.

Výpis 18.2: Část definice třídy `ČajOrig`, která simuluje přípravu čaje

```
// ...   Vynechané části definice

    /*****************************************************************************
     * Připraví hypotetický čaj, přičemž se uživatele zeptá,
     * chce-li do něj přidat mléko, cukr nebo citron.
     */
    public void připrav() {
        vařitVodu();
        připravHrnek();
        čajDoHrnku();
        zalijVodou();
        vyjmiČaj();
        přidatMléko();
        přidatCukr();
        přidatCitrón();
    }

// ...   Vynechané části definice
```

Jak vidíš, řada metod je společná, a ty ostatní jsou si velice podobné. A jak jsme si již několikrát řekli, není dobré mít jeden kód na několika místech programu.

Každý další nápoj vytvářený zaléváním něčeho vroucí vodou by potřeboval definovat obdobnou, a tím i zbytečně složitou definici.

Pro ono zalévané „něco“ se mi nepodařilo vymyslet žádný příhodný název, tak to budu v dalším textu označovat jako *trest*, i když to třeba (jako např. u kávy a čaje) většinou asi žádná trest nebude.

Definice společného předka

304. Máš pravdu – takže pro připravované nápoje definujeme nějakého společného předka, do nějž ty shodné metody přesuneme, abychom měli jejich definice jenom na jednom místě.

Uvažuješ správně – vidím, že už jsi leccos pochytil. Nicméně my do tohoto společného předka přesuneme nejenom ty shodné metody, ale my do něj přesuneme i celý algoritmus přípravy daného nápoje. Tato metoda bude onou *šablonovou metodou* (někdy se pro ni používá zkrácený termín *šablona*[1]), kolem níž se točí náš návrhový vzor.

305. Tak tohle nechápu. Jak můžeme do rodiče přenést celý algoritmus, když se pro každý druh nápoje liší? Sice jenom trochu, ale liší.

Jak algoritmus „vytknout" do předka

Využijeme např. abstraktních metod, tj. metod, které můžeme ve třídě používat, i když třída ještě vůbec neví, jak budou implementovány – o tom rozhodnou až její konkrétní potomci.

Obecně bychom mohli říci, že pro ty části algoritmu, v nichž se implementace jednotlivých tříd liší (nebo alespoň může lišit), definujeme zvláštní metody, které každý z potomků může nebo musí překrýt vlastní verzí. Tím daný potomek definuje svoji vlastní představu o tom, jak se má příslušná část algoritmu chovat.

306. Máš pravdu! Jenom mi ještě není zcela jasné, jak je to s tím „může nebo musí"?

Čtyři druhy použitých metod

Začnu trochu zeširoka. Metody používané v šablonových metodách bychom mohli rozdělit do čtyř skupin:

– pevné

- Pevně dané metody, u nichž se nepředpokládá, že by je potomci měnili. Tyto metody jsou definovány jako soukromé nebo alespoň konečné (tj. označené modifikátorem `final`, a proto nepřekrytelné). V našem příkladu by to mohly být metody `vařitVodu()` a `zalijVodou()`.

– překrytelné

- Metody, které mají svoji implicitní implementaci, jež některým potomkům zcela vyhovuje. Nicméně ti ostatní mají šanci překrýt tuto metodu vlastní verzí. Ty potomek definovat může, ale nemusí.

  V našem příkladu by sem mohla patřit metoda `vyjmiTresť()`, kterou bychom implicitně definovali jako prázdnou. Při vaření kávy by nám tato prázdná definice vyhovovala, při vaření čaje bychom ji překryli jeho vlastní verzí.

Pro překrytelné metody s implicitní implementací použité v šablonové metodě se v anglické literatuře používá často termín *hooks* – háčky.

– abstraktní

- Metody, které potomci definovat musí. Ty deklaruje jejich rodič jako abstraktní, takže se jeho potomci z povinnosti je překrýt nijak nevykroutí.

  V našem případě bychom sem mohli zařadit metodu, kterou bychom pojmenovali např. `tresťDoHrnku()` a kterou bychom museli pro každý druh nápoje definovat.

  Obecně se doporučuje, aby počet těchto abstraktních metod byl co nejmenší. Čím je jich více, tím náročnější je definice případného potomka.

---

[1] Nepleťte si tuto šablonu se šablonami zavedenými v jazyku C++. Řekl bych, že možná záměna byla jedním z důvodů, proč se tento návrhový vzor nenazývá *šablona*, ale *šablonová metoda*.

– šablonové

- Poslední skupinou jsou jiné šablonové metody, které jsou samy sestaveny obdobným způsobem.

  V našem případě bychom sem mohli zařadit přidávání různých dodatečných ingrediencí. Pro ně bychom mohli definovat např. metodu `dochutit()`, která by se třídy připravovaného nápoje zeptala, jaké ingredience je možno do daného nápoje přidat, a pak by se postupně ptala a v případě souhlasu dané ingredience přidávala.

Proč princip vzoru popisují i autoři, kteří o něm nevědí

Teď už možná také chápeš, proč jsem na začátku kapitoly říkal, že princip šablonové metody vysvětlují i učebnice, jejichž autoři nemají (přesněji v době psaní učebnic neměli) o existenci tohoto návrhového vzoru vůbec tušení (jak jsem řekl, mezi ně jsem při psaní své první učebnice objektového programování patřil i já). Jakmile totiž začnou vysvětlovat principy překrývání metod a následně princip abstraktních metod[1], musí zákonitě vysvětlit i princip šablonové metody, i když třeba nevědí, že to, co vysvětlují, se takto jmenuje a je to považováno za návrhový vzor.

Neříkám to proto, abych tyto autory shazoval, ale abych zdůraznil, že řada principů, které byly následně ukotveny v návrhových vzorech, byla známa a vyučována již dávno. Jenom třeba neměla jméno.

307. Vidíš, to mne ještě nenapadlo. Přiznejme si, že mnohé z toho, o čem si tu vyprávíme, jsem již před tím někde četl nebo zaslechl. Jenom to nebylo takové utříděné a systematizované, spíše to byly různé útržky informací.

Hollywoodský princip

Když už jsem u těch principů, přidám další: *Šablonová metoda* je často označována jako jeden ze vzorů, které implementují tzv. *hollywoodský princip*, charakterizovaný heslem *„Nevolejte nám, zavoláme vám“*.

Proč nemůže být šablonovou metodou konstruktor

308. Princip šablonové metody už snad chápu. Vrtá mi ale hlavou, proč je vše definováno tak, že nejprve vytváříš nějakou prázdnou instanci, kterou pak voláním další metody připravuješ. Proč to vše nedáš najednou do konstruktoru?

Protože konstruktor není možné pro šablonovou metodu použít.

309. Proč ne? Vždyť je to téměř metoda jako každá jiná.

Tady jde právě o to téměř. Já vím, že to většina učebnic tají, ale konstruktor se od řadových metod liší v jedné zásadní věci:

V konstruktoru nesmíš používat překrytelné metody!

Já osobně např. považuji použití překrytelných metod v konstruktoru za syntaktickou chybu.

---

[1] Pokud jste se učili tyto principy v obráceném pořadí, tak to váš učitel používal moderní metodiku, která doporučuje učit nejprve abstraktní metody a teprve potom překrývání. Tato metodika je ale poměrně mladá, takže se s ní setkáte jen v učebnicích vydaných v posledních několika letech, a to jen v některých.

310. Překladači to ale nevadí.

Proč nelze v konstruktoru použít šablonovou metodu

Na to nespoléhej. Použitím překrytelných metod v konstruktoru vkládáš do programu časovanou pumu. Jak jistě víš, před tím, než se začne vykonávat tělo konstruktoru potomka, musí se nejprve vytvořit podobjekt jeho předka – musí se tedy zavolat příslušný konstruktor. Narazí-li konstruktor předka při svém vykonávání na překrytelnou metodu, vyvolá zákonitě překrývající verzi, tj. verzi potomka. Ta však může používat atributy, které v době vytvoření podobjektu předka ještě vůbec neexistují, protože se začnou vytvářet až poté, co bude podobjekt předka hotov.

311. Teď jsi mne tedy zaskočil. Co když ale v konstruktoru potřebuji volat překrytelnou metodu? Jak se z toho mohu vylhat?

Kde najít podrobnosti

V „modré učebnici“ ([32]) jsem těmto problémům a jejich možnému řešení věnoval značnou část kapitoly o dědičnosti tříd. Tam si to můžeš nalistovat.

Možné řešení

Tady ti jenom prozradím, že jedním z možných řešení je definovat potřebný kód v soukromé metodě (jak jistě víš, soukromé metody není možno překrýt), kterou z oné překrytelné metody zavoláš (toto volání bude jediným příkazem v jejím těle).

Onu soukromou metodu zavoláš i v konstruktoru, takže budeš mít jistotu, že ti ji nikdo nepřekryje, a nebude v ní proto používat doposud neexistující atributy.

Implementace

312. Řekl bych, že začínám být v obraze. Tak mi ukaž, jak bychom mohli s pomocí šablonové metody uvařit ten čaj a kávu.

Aby ten příklad zase nebyl jenom z AHA kategorie, namíchal jsem ti do vzorového řešení ještě několik dalších vzorů.

Chtěl jsem ti ukázat, jak je možno např. jednoduše vyřešit to, že při vaření různých nápojů se nabízí různý počet přidatelných ingrediencí. Do kávy se dá přidat mléko a cukr, do čaje navíc ještě citrón (pokud možno bez mléka), kdybychom mezi naše „nápoje“ přidali instantní polévku, mohli bychom ochucovat solí a pepřem, pokud bychom mezi ně zařadili instantní horkou čokoládu, tak bychom do ní už asi nepřidávali nic.

313. A jak jsi to tedy vyřešil?

Jak lze řešit přidávání předem neznámého počtu ingrediencí

Využil jsem návrhový vzor *Příkaz* a pro přidání každé ingredience jsem definoval její příkaz. Jedna z metod, které může potomek překrýt, pak vrací iterovatelný objekt s těmito příkazy (pro ty, kteří nic nepotřebují, je připravena implicitní verze vracející prázdný iterovatelný objekt).

V příslušnou chvíli zavolá šablonová metoda speciální „přidávací“ metodu, která požádá o iterovatelný objekt s ingrediencemi a prochází příkaz za příkazem. Pokaždé se uživatele zeptá, jestli chce danou ingredienci přidat, a pokud ano, tak spustí příslušnou přidávací metodu.

314. Jak zjistíš, co ta přidávací metoda vlastně přidává, aby ses na to mohl zeptat.

Zabalil jsem do daného objektu dvě metody. Vedle metody realizující přidání dané ingredience tam najdeš ještě metodu, která vrací název této ingredience.

Implicitní verze vrací prázdný iterovatelný objekt

315. A když některý nápoj nemá žádné přísady (např. ta instantní polévka), tak jeho metoda vrátí prázdný odkaz, tj. `null`.

Vidím, že mne moc nesleduješ. Před chvílí jsem říkal, že implicitní verze metody vrací prázdný iterovatelný objekt.

Proč nevracet prázdný odkaz

Kromě toho jsme si už několikrát říkali, že prázdný odkaz zbytečně komplikuje další zpracování a že bývá často výhodné vrátit místo prázdného odkazu něco, co jsme označili jako prázdný objekt. No a to dělám i zde.

316. Místo tvého iterovatelného objektu by ale bylo možno použít např. pole nebo nějaký kontejner.

Proč vracet právě iterovatelný objekt

Samozřejmě. Mně jenom připadal iterovatelný objekt takový nejuniverzálnější a nejsnáze zpracovatelný.

317. Řekl bych, že už to chápu. Tak už mi ukaž ten program, ať se mohu poučit a porovnat ho s tím, jak bych to asi naprogramoval já.

Kde se co najde

Definici společného abstraktního předka najdeš ve výpisu 18.3, definici přípravy čaje ve výpisu 18.4. Jak vidíš, výrazně se zjednodušila.

Mezi doprovodnými programy najdeš ještě definici pro přípravu kávy a instantní polévky, ale protože chci šetřit místem, tak ty už zde neuvádím.

Než se zabořís do studia ukázkových programů, tak bych tě chtěl ještě upozornit, že jsem do společného předka přesunul i definice společných příkazových objektů pro přidání mléka a cukru.

Lektorka mne plísnila, že identifikátory `M` a `X` v předváděných programech nejsou dostatečně samovysvětlující, takže jí nic neříkají. Mně zase naopak vadí, když mají identifikátory pomocných, výplňových objektů dlouhé názvy a prodlužují tak (a často i znepřehledňují) vytvářené texty. (A navíc pro ně nejsem schopen vymyslet inteligentní název, který by v těchto textech nerušil.) Zkuste si zapamatovat, že `M` označuje počáteční `Mezery` a že `X` díky svému tvaru označuje něco, co texty umístěné vlevo a vpravo od něj odděluje tak, aby výsledek byl pěkně čitelný. Jak je bude oddělovat, si můžete nastavit v příslušné konstantě.

Výpis 18.3: Definice třídy `Nápoj`, která je společným rodičem tříd pro přípravu nápojů.

```
package rup.česky.vzory._18_šablona;

import rup.česky.společně.IO;

import rup.česky.vzory._16_iterátor.PrázdnýIterable;


/*******************************************************************************
 * Instance třídy Nápoj představují nápoje,
```

```
 * které se musí umět nejprve připravit.
 */
public abstract class Nápoj
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Mezery pro odsazení popisu činností. */
    protected static final String M = "   ";

    /** Oddělovač popisů činností sestavený z odřádkování a odsazení. */
    protected static final String X = "\n" + M;

    /** Příkaz pro přidání mléka. */
    protected static final IPřídavek MLÉKO =
        new IPřídavek() {
            public String název() { return "Mléko"; }
            public void přidej() {
                System.out.println( M+"Vezmi mléko" +X+ "Nalej dávku" );
            }
        };

    /** Příkaz pro přidání cukru. */
    protected static final IPřídavek CUKR =
        new IPřídavek() {
            public String název() { return "Cukr"; }
            public void přidej() {
                System.out.println( M+"Vezmi cukr" +X+ "Nasyp dávku" );
            }
        };


//== ABSTRAKTNÍ METODY =========================================================

    /*****************************************************************************
     * Dodá do hrnku "tresť", která po zalití vroucí vodou vytvoří daný nápoj.
     */
        protected abstract void tresťDoHrnku();


//== NESOUKROMÉ  METODY INSTANCÍ ===============================================

    /*****************************************************************************
     * Připraví hypotetický nápoj vzniklý přelitím nějaké "tresti" vroucí vodou.
     * Při tom se uživatele zeptá, chce-li do nápoje přidat nějaká
     * "dochucovadla", např. mléko nebo cukr.
     */
    public final void příprav()
    {
        vařitVodu();
        připravHrnek();     //Potomek může upravit
        tresťDoHrnku();     //Potomek MUSÍ definovat
        zalítVodou();
        vyjmoutTresť();
        dochutit();         //Potomek může dodat seznam ingrediencí
    }


    /*****************************************************************************
```

```
     * Připraví hrnek vhodného tvaru a velikosti.
     */
    protected void připravHrnek() {
        System.out.println( "připravHrnek" +X+
            "Vezmi hrnek 3 dl"  );
    }


    /*****************************************************************************
     * Vyjme z hrnku tresť.
     */
    protected void vyjmoutTresť() {
    }


    /*****************************************************************************
     * Vrátí iterovatelný kontejner přidávaných ingrediencí.
     * @return iterovatelný kontejner přidávaných ingrediencí
     */
    protected Iterable<IPřídavek> přídavky() {
        return PrázdnýIterable.getInstance();
    }


    /** {@inheritDoc} */
    public String toString() {
        return getClass().getSimpleName();
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ======================================

    /*****************************************************************************
     * Dá vařit vodu. Na uvedení vody do varu nečeká.
     */
    private void vařitVodu() {
        System.out.println( "vařitVodu" +X+
            "Natoč vodu" +X+ "Zapni topení" );
    }


    /*****************************************************************************
     * Počká, až se voda začne vařit, a pak ji nalije do hrnku.
     */
    private void zalítVodou() {
        System.out.println( "zalítVodou" +X+
            "Počkej, až se začne vařit" +X+ "Nalej vodu do hrnku" );
    }


    /*****************************************************************************
     * Postupně uživateli nabídne jednotlivá dochucovadla a přidá ta,
     * o jejichž přidání uživatel požádá.
     */
    private void dochutit() {
        for( IPřídavek p : přídavky() ) {
            System.out.println( "Přidat " + p.název() );
            if( IO.souhlas( p.název() + "?" )  )
```

```
                p.přidej();
            else
                System.out.println( M + "Bez " + p.název() );
        }
    }


//== TESTY ======================================================================

    /*****************************************************************************
     * Připraví zadaný nápoj.
     */
    private static void připrav( Nápoj nápoj ) {
        System.out.println( "\n===== " + nápoj + " =====");
        nápoj.příprav();
    }

   /*****************************************************************************
    * Test přípravy několika nápojů.
    */
   public static void test() {
       připrav( new Čaj() );
       připrav( new Káva() );
       připrav( new InstantníPolévka() );
   }/** @param args Nepoužívané parametry příkazového řádku. */
   public static void main( String... args ) { test(); }
}
```

Výpis 18.4: Definice třídy `Čaj`

```
package rup.česky.vzory._18_šablona;

import rup.česky.vzory._16_iterátor.IterovatelnéPole;


/*******************************************************************************
 * Instance třídy Čaj představují čaj, který musí jejich konstruktor
 * ve spolupráci s rodičovským konstruktorem nejprve uvařit.
 */
public class Čaj extends Nápoj
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /** {@inheritDoc} */
    @Override
    protected void třeťDoHrnku() {
        System.out.println( "třeťDoHrnku" +X+
            "Vyber sáček" +X+ "Otevři sáček" +X+ "vlož sáček do hrnku" );
    }


    /** {@inheritDoc} */
    @Override
    protected void vyjmoutTřeť() {
        System.out.println( "vyjmoutTřeť" +X+
            "Počkej 3 min" +X+ "Vyjmi sáček" );
    }
```

```
    /** {@inheritDoc} */
    @Override
    protected Iterable<IPřídavek> přídavky() {
        return new IterovatelnéPole<IPřídavek>( new IPřídavek[] {
            MLÉKO,
            CUKR,
            new IPřídavek() {
                public String název() { return "Citrón"; }
                public void přidej() {
                    System.out.println( M+"Vezmi citrón" +X+ "Nalej dávku" );
                }
            }
        } );
    }

}
```

Překrytí šablonové metody

318. A co když potomek překryje šablonovou metodu?

GoF označuje šablonové metody za nepřekrytelné

V této otázce není jednotný názor. GoF tvrdí, že šablonové metody by měly být nepřekrytelné. Existují-li situace, kdy má smysl překrýt šablonovou metodu, bude v návrhu nejspíše něco špatně. Šablonová metoda by měla být pro všechny společná, a proto by měla být definována jako finální, tj. nepřekrytelná.

Mohou být i soukromé

V řadě případů je šablonová metoda dokonce soukromá. Řeší nějaký složitější problém a nikdo z potomků vůbec nemusí vědět, který a jak. Potomci prostě dostanou za úkol definovat správnou implementaci předem zadaných detailů a o vše ostatní se postará společný rodič.

Standardní knihovna se jejich překrývání nebrání

Na druhou stranu se někteří z novějších autorů překrytí šablonové metody nebrání. Podíváš-li se do standardní knihovny, najdeš tam řadu šablonových metod, které jsou překrytelné.

Příklad

319. Prozraď mi, kde ve standardní knihovně najdu šablonové metody?

Šablonové metody ve standardní knihovně

Šablonové metody zde potkáš na každém kroku. Jako typický příklad bych mohl uvést např. abstraktní třídy knihovny kontejnerů v balíčku `java.util` nebo abstraktní třídy v knihovně proudů v balíčku `java.io`.

Např. třída `java.util.AbstractCollection` požaduje po svých potomcích definici metod `iterator()` a `size()` a s jejich pomocí již ostatní metody požadované rozhraním `java.util.Collection` definuje sama jako šablonové metody (i když tentokrát překrytelné). Metoda `add(Object)` je sice definována jako nepodporovaná (po svém zavolání vyhodí bez přemýšlení výjimku `UnsupportedOperationException`), ale překryješ-li ji, máš sadu požadovaných metod kompletní.

Vzhledem k tomu, že příklad, na němž jsme si princip šablonové metody vysvětlovali, byl takový AHA-příklad, tak bych ti doporučil studium zdrojových kódů těchto tříd místo příkladu, který na konci kapitol většinou uvádíme.

Shrnutí – co jsme se naučili

- *Šablonová metoda* je jeden z nejpoužívanějších vzorů.
- Implementuje v rodičovské metodě algoritmus, jehož některé detaily definují až potomci.
- *Šablonová metoda* bývá nejčastěji implementována v abstraktní třídě.
- Metody použité v *Šablonové metodě* můžeme rozdělit do čtyř skupin:
  - pevné, nepřekrytelné metody definované ve společné rodičovské třídě,
  - překrytelné metody s implicitní implementací (anglicky často označované jako *hooks* – háčky),
  - abstraktní metody, tj. metody, které musí potomek definovat,
  - jiné šablonové metody.
- Počet abstraktních metod, které potomci musí implementovat, by měl být co nejmenší.
- Šablonovou metodou nemůže být konstruktor, protože konstruktor nesmí obsahovat překrytelné metody (syntaxe to sice povoluje, ale běda tomu, kdo této možnosti využije – program se mu za to pomstí).
- Zákaz používání překrytelných metod v konstruktoru je možno obejít vyvedením těla překrytelné metody do soukromé metody, kterou bude volat jak překrytelná metoda, tak konstruktor.
- Aplikace předem neznámého počtu metod je možné řešit použitím návrhového vzoru *Příkaz* a předáním příslušných objektů-metod v nějakém iterovatelném objektu.
- GoF tvrdí, že šablonové metody by měly být nepřekrytelné.
- Standardní knihovna Javy definuje řadu šablonových metod jako překrytelné.
- Návrhový vzor *Šablonová metoda* patří mezi vzory uvedené v GoF.

ČÁST 4

Optimalizujeme rozhraní

V této části si probereme návrhové vzory, které zjednodušují rozhraní tříd, a dokonce i celých částí aplikací. Jejich účelem je maximálně usnadnit komunikaci uživatele s danou třídou či částí aplikace.

KAPITOLA 19

Je to zbytečně složité (Fasáda – Facade)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Návrhový vzor *Fasáda* použijeme ve chvíli, kdy nějaký systém začíná být pro své uživatele příliš složitý vzhledem k oblasti úloh, které chtějí s jeho pomocí řešit. Ukazuje, jak nahradit sadu rozhraní jednotlivých subsystémů sjednoceným rozhraním zastupujícím celý systém. Definuje tak rozhraní vyšší úrovně, které usnadní využívání podsystémů. Jejím cílem je zjednodušit rozhraní celého systému a snížit počet tříd, s nimiž musí uživatel přímo či nepřímo komunikovat.

[1] **Definice v GoF:** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. – Poskytuje jednotné rozhraní k množině rozhraní v subsystému. *Fasáda* definuje rozhraní vyšší úrovně, které používání subsystému usnadní.

Účel

320. Pořád tu vyprávíš, jak mi návrhové vzory zjednoduší práci. Zatím jsme hovořili pouze o situaci, kdy něco vytvářím. Napadla mne hříšná myšlenka, jestli tam nemáš také nějaký vzor na to, abych si zjednodušil nejenom vytváření systémů, ale také jejich používání.

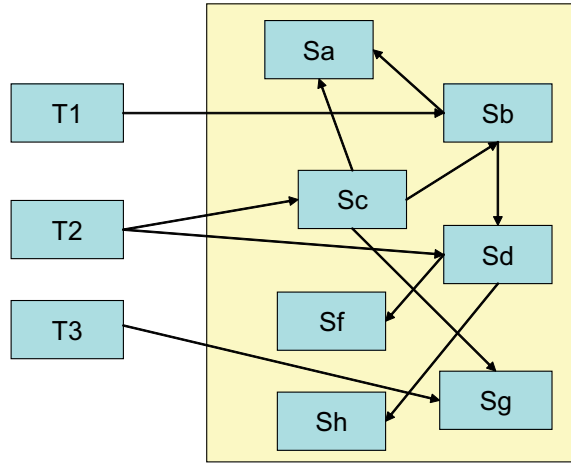
Fasáda zakrývá složitost systému

Správná myšlenka ve správnou chvíli. Něco bych tu pro tento účel měl. Příslušný návrhový vzor se jmenuje *Fasáda* a slouží k podobnému účelu jako fasáda na domě: zakrývá složitost celé stavby (dům je stavěný z jednotlivých cihel, nad okny jsou překlady, abychom usadili okno na správné místo, okolní prostor všelijak upravujeme atd.) a nabízí vnějšímu pozorovateli jednotný (a většinou i příjemný) pohled na celou stavbu.

321. Jenomže mně nestačí se na systém hezky koukat, já bych potřeboval, abych mohl systém snadno používat.

Problémy složitých systémů

To se nevylučuje. Podívej se na obr. 19.1. Tři obdélníky vlevo představují tvoje třídy `T1`, `T2` a `T3`, velký obdélník vpravo s řadou menších, navzájem pospojovaných obdélníků uvnitř představuje nějaký složitější systém, který tvoje třídy používají.



Obrázek 19.1
Spolupráce tvých tříd se složitým systémem

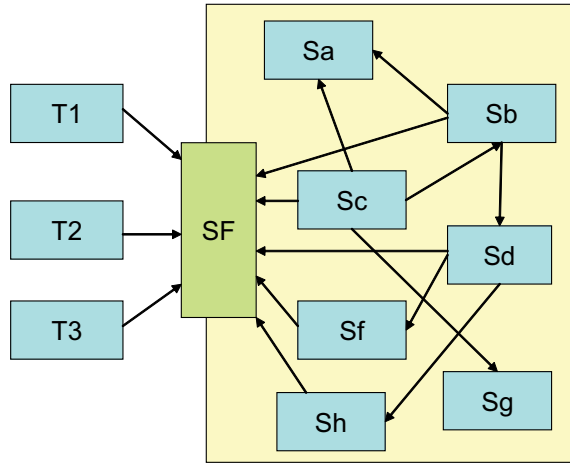
Abys mohl daný systém bezpečně používat, měl bys znát jeho třídy i hlavní vazby mezi nimi. U složitějších systémů je to však poměrně rozsáhlý soubor informací, které se ti často nechce studovat, protože víš, že budeš využívat pouze zlomek služeb, které daný systém nabízí.

Cíl: jednoduše používat složitý systém

Situace, kdy potřebuješ používat nějaký složitější systém pro relativně jednoduché úkoly, je poměrně častá a řeší ji návrhový vzor *Fasáda*. Ten doporučuje vložit mezi celý systém a jej používající třídy a objekty nějakého zprostředkovatele (třídu či skupinu tříd), který ty dva světy oddělí.

Tohoto zprostředkovatele budu v dalším textu označovat jako *fasádu*, čímž se vyhnu problémům s tím, zda oním zprostředkovatelem je jediná třída či celá skupina tříd.

Fasáda usnadňuje používání systému uživatelům – především těm s jednoduššími požadavky. Ti pak nemusí mít neustále na paměti všechny vzájemné vazby uvnitř systému, protože budou spolupracovat se zdánlivě jednodušším systémem – fasádou (viz obr. 19.2).



Obrázek 19.2
Fasáda zjednodušuje pohled na složitý systém

Účel fasády

Jinými slovy: účelem fasády je umožnit uživatelům s jednoduššími požadavky pracovat se složitým systémem, jako by byl jednoduchý.

322. Proč bychom ale měli používat složitý systém, když jej stejně nevyužijeme?

Proč používat složitý systém

Například proto, že jej máme již dávno k dispozici a nebudeme kvůli skupině jednodušších požadavků vymýšlet nějaký nový.

Autorem fasády ale většinou nebývají její budoucí uživatelé. Jejími autory jsou většinou autoři onoho složitého systému, kteří chtějí zjednodušit život uživatelům s jednoduššími požadavky, aby nemuseli chodit s kanónem na vrabce (a často také proto, aby se nezačali poohlížet po něčem jednodušším).

323. Už zase začínám tápat. Dej mi nějaký příklad, ať si ta tvoje tvrzení zasadím do života.

Příklad složitého systému: Swing

Typickým příkladem takovéhoto systému je standardní knihovna pro tvorbu grafického uživatelského rozhraní označovaná jako *knihovna Swing*. Aby mohla vyhovět všem požadavkům svých budoucích uživatelů, je poměrně složitá a vzájemně provázaná.

Když potřebuješ definovat nějaké složitější dialogové okno nebo jakékoliv jiné „rafinované“ uživatelské rozhraní, oceníš její dokonalost. Když ale potřebuješ pouze

zobrazit nějaké jednoduché okno se zprávou nebo získat nějakou jednoduchou odpověď na otázku, bude pro tebe knihovna zbytečně složitá.

324. Souhlasím. A jak to mám tedy vyřešit?

JOPtionPane – fasáda knihovny Swing

Nijak. Autoři knihovny to již vyřešili za tebe: součástí knihovny je třída `javax.swing.JOptionPane`, jejíž statické metody nabízejí velice jednoduchý způsob tvorby základních dialogových oken umožňujících zobrazení nějaké zprávy a získání jednoduchých vstupních dat.

Její instance pak nabízejí další sadu jednoduchých prostředků, které řeší úlohy maličko složitější, ale stále ještě dostatečně jednoduché na to, aby nebylo nutno zaměstnávat celou artilerii knihovny Swing.

Podrobnosti si můžeš najít v dokumentaci.

325. Říkal jsi, že místo jedné třídy se někdy používá celá skupina tříd – na to bys měl také příklad?

Fasáda skupiny tříd: knihovna `rup.česky.tvary`

Příkladem takovéto fasády může být např. moje knihovna `rup.česky.tvary`. Její třídy zpřístupňují jednoduchou práci s grafickými objekty, aniž by bylo nutno znát celou problematiku související s plnohodnotným používáním grafických objektů v Javě.

Implementace

326. Zmínkou o třídě `JOptionPane` jsi vlastně popsal i možnou implementaci: fasádu můžeme definovat jako knihovní třídu.

Fasádu lze implementovat jako knihovní třídu

Třída `JOptionPane` sice není knihovní třídou, ale postačí-li ti opravdu pouze zveřejnění základní sady metod, můžeš s obyčejnou knihovní třídou vystačit (probírali jsme ji v kapitole *Žádná instance (Knihovní třída – Library Class)* na straně 103). To ale není případ třídy `JOptionPane`. Jak jsem se zmínil, tato třída může mít i vlastní instance – ty použiješ např. tehdy, budeš-li chtít příslušné dialogové okno někam umístit, zabezpečit, aby bylo doopravdy vidět, apod.

Příklad: `java.lang.System`

Příklad: `java.lang.System`

Příkladem fasády definované jako knihovní třída je např. třída `java.lang.System`. Ta ti umožňuje „komunikovat“ s operačním systémem a chtít po něm některé služby, aniž bys musel znát komplex služeb operačního systému, a dokonce aniž bys věděl, nad jakým operačním systémem tvůj program pracuje.

327. Napadla mne další věc: třída `JOptionPane` přidává do systému další funkčnost. Metody na vytvoření jednoduchých dialogových oken v něm před tím nebyly.

Fasáda může přidat další funkčnost

Máš pravdu. Při implementaci fasády bývá často užitečné dodat do systému další funkčnost. Funkčnost, kterou přidává `JOptionPane`, je poměrně prostoduchá: přidá metody nahrazující nějaké standardní posloupnosti volání jiných metod. Často se ale v rámci fasády přidá nějaká rafinovanější funkčnost.

328. Třeba…

Příklad rafinovaného rozšíření funkčnosti

Třeba monitorování frekvence využívání jednotlivých služeb systému. Když víš, že všechny požadavky na systém půjdou přes fasádu, můžeš do fasády zabudovat nějaký monitorovací mechanismus, který bude sledovat frekvenci volání jednotlivých služeb systému (případně i s dobou jeho odezvy) a může být docela dobrým podkladem pro další vylepšování daného systému.

329. To je zajímavá myšlenka. Máš tam ještě něco podobně chytrého?

Fasáda může zapouzdřit systém

Fasádu můžeš použít k zapouzdření systému. Pokud bude fasáda jediným přístupovým bodem k celému systému, pak ti nic nebrání celý systém vyměnit za jiný. Uživatel, který komunikuje se systémem pouze prostřednictvím fasády, vůbec nemusí takovou změnu zaznamenat.

330. Takovou fasádou je např. třída `System`, o které jsi před chvílí hovořil – ta přece umožňuje komunikovat se systémem nezávisle na tom, na jakém systému program zrovna běží.

Příklad: opět `java.lang.System`

Fasádou, která tvůj program odstiňuje od platformy, na které právě běží, je vlastně celý virtuální stroj. Mohli bychom ale také říci, že pro jistou skupinu požadavků na systém nám jako takováto fasáda odstiňující nás od vlastního operačního systému slouží třída `System`. Není na to ale sama. Pomáhají jí v tom ještě další „fasádové“ třídy, jakými jsou např. třídy `Runtime` nebo `Process`.

Příklad

331. Obávám se, že tentokrát budu bez příkladu.

Probrané příklady

Vždyť už jsem ti o celé řadě příkladů pověděl:

- třída `javax.swing.JOptionPane`
- třídy `System`, `Runtime`, `Process` v balíčku `java.lang`
- balíček `rup.česky.tvary` a v něm pak zejména třídy `SprávcePlátna` a `Kreslítko`.

Pokoušet se o samostatný příklad na použití fasády mi připadá nemoudré. Problémem fasády totiž je, že k tomu, abys opravdu docenil její existenci, by bylo vhodné, aby ses přiměřeně vyznal v systému, který zakrývá. A nemá smysl budovat nějaký složitý systém jenom proto, abychom jej pak demonstrativně zakryli.

Shrnutí – co jsme se naučili

- Cílem návrhového vzoru *Fasáda* je zjednodušit rozhraní celého systému a snížit počet tříd, s nimiž musí uživatel přímo či nepřímo komunikovat.
- Má smysl o ní uvažovat v případě, kdy cena vytvoření fasády (třídy či skupiny tříd) je evidentně nižší než cena studia kompletního systému uživateli, kteří jej nebudou celý využívat, resp. cena správy celého systému.

- Využijeme jej zejména tehdy, pokud nepotřebujeme plnou funkcionalitu systému a stačí nám nějaká její poměrně útlá podmnožina.
- Využijeme ji i tehdy, chceme-li k funkcionalitě systému přidat nějakou drobnou nadstavbu, např. monitorování přístupů k zastupovanému systému.
- *Fasáda* je často implementována jako jediná třída.
- V nejjednodušších případech může být *Fasáda* implementována i jako knihovní třída (viz kapitolu *Žádná instance (Knihovní třída – Library Class)* na straně 103).
- *Fasáda* může efektivně zapouzdřit svůj podsystém a tím umožnit jeho operativní výměnu či úpravu.
- Návrhový vzor *Fasáda* patří mezi vzory uvedené v GoF.

KAPITOLA 20

Je to trochu jinak (Adaptér – Adapter)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Návrhový vzor *Adaptér* využijeme ve chvíli, kdy bychom potřebovali, aby třída měla jiné rozhraní než to, které právě má. Pak mezi ni a potenciálního uživatele vložíme třídu adaptéru, která bude mít požadované rozhraní a konvertuje tak rozhraní naší třídy na rozhraní požadované.

[1] **Definice v GoF:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. – Změní rozhraní třídy na rozhraní, které klient očekává. *Adaptér* umožní spolupráci třídám, které by vzhledem k rozlišným rozhraním jinak spolupracovat nemohly.

Účel

332. Slovo *adaptér* v názvu kapitoly mi připomíná síťový adaptér, který bývá součástí příslušenství notebooku a slouží k připojení notebooku do elektrické sítě. Má tento adaptér něco společného s tím, o němž chceš mluvit v této kapitole?

Adaptéry v běžném životě

Má. Notebooky bývají konstruovány na vstupní napětí okolo 13 V. V síti však máš napětí okolo 220 V. Adaptér ti převede síťové napětí na napětí požadované notebookem.

Tento adaptér je ale jednoúčelový. Daleko zajímavější jsou adaptéry, které převádí jednotlivé systémy zásuvek a zástrček. Vyrazíš-li někam do ciziny, často zjistíš, že se tvoje zástrčka nekamarádí se zásuvkou ve zdi a musíš použít adaptér, který je vzájemně přizpůsobí. Na tomto adaptéru je zajímavá jeho univerzalita: je na jedné straně připraven k zasunutí do libovolné zásuvky v dané oblasti a na druhé straně k přijetí libovolné zástrčky z jiné oblasti.

Účel adaptérů

Jak zmiňovaný síťový adaptér, tak adaptéry používané v objektově orientovaném programování mají za úkol přizpůsobit existující rozhraní objektu (v případě notebooku oněch vstupních 13 V, resp. tvar zástrčky) tomu rozhraní, které bys potřeboval, aby objekt měl (v naší republice síťových 220 V a tvar odpovídající našim zásuvkám).

Jiný pohled

Můžeš se na to dívat i obráceně a prohlásit adaptéry za zařízení, která mají za úkol přizpůsobit existující, požadované rozhraní (síťových 220 V) tomu rozhraní, které tvůj objekt očekává a na něž je konstruován (13 V).

333. Dejme tomu, že programátorsky popsaný význam síťového adaptéru chápu. Dej mi ale také příklad nějakého skutečného programového adaptéru.

Příklad: obalové třídy

Takovými adaptéry jsou v Javě např. všechny obalové třídy (ostatně někdy bývá tento návrhový vzor označován jako *wrapper* – *obal*). Přizpůsobují rozhraní hodnot primitivních typů rozhraní požadovanému metodami, vyžadujícími parametry objektových typů.

Typickým příkladem takovýchto metod jsou např. metody kontejnerů z balíčku `java.util`. Do těchto kontejnerů můžeš ukládat pouze hodnoty objektových typů. Chceš-li do nich uložit hodnotu primitivního typu, musíš z ní udělat objekt – např. ji zabalit pomocí adaptéru, jímž je její obalový typ.

Další příklad: adaptéry posluchačů událostí GUI

Jinými příklady adaptérů jsou např. třídy adaptérů sloužící ke zjednodušení definic obsluh událostí GUI. Řada událostí je definována ve shlucích, které pak ošetřuje společná obsluha. Nechceš-li obsluhovat všechny, můžeš využít služeb adaptéru, který ti umožní obsluhovat pouze ty vybrané (toto řešení někteří puristi za implementaci vzoru *Adaptér* nepovažují – podrobnosti za chvíli).

Implementace

334. To jsou ale docela rozdílné způsoby adaptace. Odhaduji, že jejich implementace se bude dost lišit.

Máš pravdu. Adaptér můžeš implementovat v principu třemi různými způsoby. Projděme si jeden po druhém.

Univerzální adaptér

335. Dobře, začni.

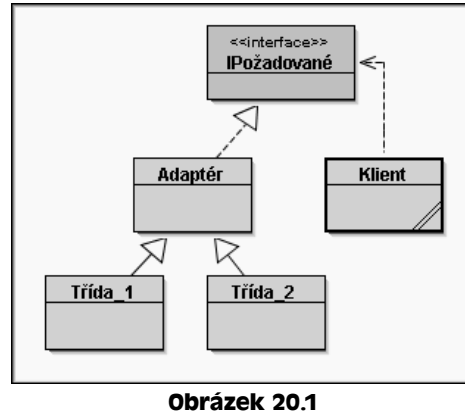
Způsob implementace

První způsob ti bude asi nejpochopitelnější. Je použit v knihovně AWT pro obsluhu událostí a jeho diagram tříd jsem ti nakreslil na obrázku 20.1. Při této implementaci definuješ třídu, která bude sloužit jako univerzální adaptér pro celou skupinu přizpůsobovaných tříd. V takto definovaném adaptéru pak nějakým jednoduchým způsobem implementuješ implicitní verze všech metod požadovaných rozhraním.

Klientem je pak třída, jejíž instance se dozvídají od systému o různých událostech a předávají tuto informaci instancím tříd majících na starosti obsluhu těchto událostí. Od těchto instancí požadují implementaci příslušného rozhraní. Adaptér slouží jako prostředník pro komunikaci mezi instancemi obslužných tříd a instancemi klienta.

Jak si lze domyslet z obrázku, tento prostředník není skutečný (tj. nebude mít své vlastní, samostatné instance), ale pouze virtuální (bude skrytým podobjektem instancí svých dceřiných tříd). On jenom zařídí, že se instance jeho dceřiných tříd budou moci vydávat za instance rozhraní `IPožadovaný`, aby s nimi byl klient ochoten komunikovat.

Jak jsem již řekl, toto řešení někteří puristi za implementaci návrhového vzoru *Adaptér* nepovažují (není uvedeno v GoF). Namítají, že adaptérem musí být instance, která zprostředkuje komunikaci dvou objektů, což v tomto případě splněno není. Já se však domnívám, že přestože toto řešení není uvedeno v GoF, splňuje základní charakteristiku adaptéru a *změní rozhraní třídy na rozhraní, které klient očekává*. Je sice pravda, že díky svému „potomkovství" dceřiná třída již klientem očekávané rozhraní má, ale bez svého předka by je neměla. Druhým důvodem pro uvedení této možnosti je, že se vás nesnažím naučit pouze odříkat návrhové vzory, ale chci vám ukázat, jak se má objektově programovat. Proto se domnívám, že tato konstrukce do výkladu rozhodně patří.



Obrázek 20.1
Adaptér jako rodič adaptované třídy

336. Mohu implicitní verze metod definovat jako prázdné?

Typické implicitní definice a příklady použití

To záleží na konkrétních požadavcích. Nemají-li nic vracet, bývá většinou optimální definovat je jako prázdné. To je příklad adaptérů z balíčku `java.awt.event`.

Často je ale volání neimplementovaných metod nepřípustné, takže jejich optimální adaptérová definice obsahuje pouze příkaz na vyhození výjimky `UnsupportedOperationException`. Příkladem adaptéru s řadou takto definovaných metod může být např. třída `java.util.AbstractList`.

337. O těchto třídách ses ale zmiňoval jako o implementacích návrhového vzoru *Tovární metoda*.

To se nevylučuje. Tyto třídy slouží jako adaptéry, jejichž implicitní verze metod jsou definovány s využitím vzoru *Tovární metoda*.

338. Jestli jsem to ale dobře pochopil, tak tento způsob implementace můžeme použít pouze tehdy, definujeme-li novou třídu, která nemusí být součástí žádné jiné dědické hierarchie.

Použitelné jen pro upravitelné třídy

Pochopil jsi to dobře. Při tomto řešení je třeba zasahovat do adaptovaných tříd. Jejich zdrojový kód proto musí být pro tebe dosažitelný a musíš mít možnost jej upravit. Nejčastěji se tohoto způsobu využívá právě tehdy, definuješ-li nové třídy a nechce se ti implementovat všechny metody požadované rozhraním.

339. Můžeš opět uvést nějaký příklad?

Příklad: `IteratorAdapter`

Již jsem o něm hovořil. Je to třída `IteratorAdapter`, o které jsem hovořil na straně 209 v souvislosti s definicí třídy IterovatelnéPole. Tento adaptér slouží při definicích iterátorů, které sice implementují rozhraní `Iterator`, avšak nechce se jim implementovat metodu `remove()`, protože její používání zakazují. Třída `IteratorAdapter` proto definuje onu nepodporovanou verzi metody (její trapně jednoduchý zdrojový kód si můžeš prohlédnout ve výpisu 16.2 na stránce 209) a ty už se můžeš v jejích potomcích starat pouze o zbylé metody, které podporovat chceš.

Adaptér jako vnořená třída implementovaného rozhraní

Obdobné adaptéry jsem si zvykl vkládat jako vnořené třídy do většiny svých knihovních rozhraní. Každé z nich má definovánu vnořenou třídu `Adaptér`, která implementuje všechny metody deklarované daným rozhraním. Většinou je přitom deklaruje tak, že předpokládá, že implementační třída nebude chtít danou metodu vůbec využívat, a metody proto vyhazují výjimku `UnsupportedOperationException`.

Příklad: `IPosuvný`

Jako příklad může sloužit např. rozhraní `rup.česky.tvary.IPosuvný`, které je potomkem rozhraní `rup.česky.tvary.IKreslený` a jeho vnitřní třída `Adaptér` (přesněji `rup.česky.tvary.IPosuvný.Adaptér`) je definována jako potomek třídy `rup.česky.tvary.IKreslený.Adaptér`, která definuje výše popsanou implicitní implementaci metody `nakresli(Kreslítko)`.

Výpis 20.1: Definice rozhraní `IPosuvný` s vnořeným adaptérem

```
package rup.česky.tvary;


/*****************************************************************************
```

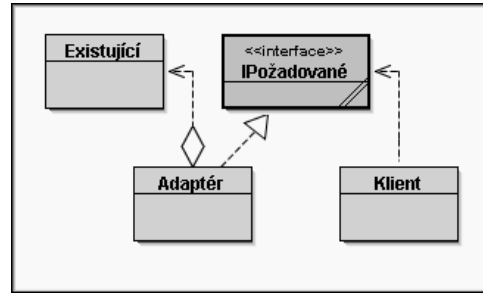
```
 * Rozhraní IPosuvný definuje povinnou sadu metod, jež musí být poskytovány
 * objekty, které mají být schopny posunu po animovaném plátně.
 */
public interface IPosuvný extends IKreslený
{
//== DEKLAROVANÉ METODY ================================================

    /**************************************************************************
     * Vrátí aktuální pozici posuvného objektu (většinou pozici levého horního
     * rohu opsaného obdélníku) jako instanci třídy {@code Pozice}.
     *
     * @return Akutální pozice objektu.
     */
     public Pozice getPozice();


    /**************************************************************************
     * Přesune posuvný objekt do nové pozice.
     *
     * @param pozice  Nová pozice objektu.
     */
    public void setPozice( Pozice pozice );


    /**************************************************************************
     * Nastaví novou pozici objektu.
     *
     * @param x   Nová x-ová pozice objektu
     * @param y   Nová y-ová pozice objektu
     */
    public void setPozice(int x, int y);


//== VNOŘENÉ TŘÍDY =====================================================

    /**************************************************************************
     * Třída <code>IPosuvný.Adaptér</code>
     * definuje adaptér pro třídy, které z nějakého důvodu musí implementovat
     * rozhraní {@link IPosuvný},
     * avšak nechtějí implementovat všechny jeho metody.
     */
    public static class Adaptér extends IKreslený.Adaptér
                             implements IPosuvný
    {
    //== NESOUKROMÉ METODY INSTANCÍ ============================================

        /**********************************************************************
         * Metoda není podporována.
         * @return Akutální pozice objektu.
         */
         public Pozice getPozice(){
             throw new UnsupportedOperationException();
         }


        /**********************************************************************
         * Metoda není podporována.
```

```
         * @param pozice  Nová pozice objektu.
         */
        public void setPozice( Pozice pozice ){
            setPozice( pozice.x, pozice.y );
        }


        /**************************************************************************
         * Metoda není podporována.
         * @param x   Nová x-ová pozice objektu
         * @param y   Nová y-ová pozice objektu
         */
        public void setPozice(int x, int y){
            throw new UnsupportedOperationException();
        }

}
```

Adaptér obsahující adaptovaný objekt

340. Jak mám vytvořit adaptér v situaci, kdy potřebuji přizpůsobit nějakou existující třídu, jejíž definici již nemohu změnit?

Při druhém způsobu již není adaptér definován jako univerzální, ale je naopak jednoúčelový. (Něco jako tvůj adaptér na notebooku – na něm je také napsáno, že jej nemáš používat s jinými zařízeními.) Diagram tříd této implementace najdeš na obrázku 20.2.



Obrázek 20.2
Adaptér obsahující adaptovaný objekt jako svůj atribut

Způsob implementace

Při něm definuješ adaptér jako třídu implementující požadované rozhraní a jako atribut jejích instancí pak definuješ odkaz na existující, adaptovanou třídu. Klient se pak bude obracet na instanci adaptéru, která všechna volání převede na posloupnost volání metod svého atributu.

Takto implementovaný adaptér je možno použít i pro potomky

341. Říkal jsi, že při této implementaci je adaptér jednoúčelový. Já si ale myslím, že tak jednoúčelový nebude, protože může posloužit nejenom pro onu existující třídu, ale i pro jakéhokoliv jejího potomka.

Máš pravdu. Vzhledem k tomu, že instance libovolné třídy musí být považovatelná za kteréhokoliv ze svých předků.

342. To bych také pochopil. Nabídneš mi pro ilustraci nějaký příklad?

Příklad: obalové typy

Příkladem takovéhoto řešení jsou např. všechny obalové typy. Jejich obalovaným atributem je hodnota příslušného primitivního typu a implementovaným rozhraním je rozhraní třídy `Object`.

Obalové typy ti tak zařídí, aby hodnoty primitivních typů mohly vystupovat jako instance objektových typů, tj. jako instance některého z potomků třídy `Object`.

343. Obalové typy nejsou dobrý příklad. Většina jejich metod je nativních, takže si jejich definice neprohlédnu. Nemáš tam něco, na co se mohu podívat?

Další příklad: `Kreslítko`

V balíčku `rup.česky.tvary` najdeš třídu `Kreslítko`, jejíž instance představují nástroj, kterým je možné kreslit na plátno. Třídu jsem se snažil definovat tak, aby se s ní začátečníkům dobře pracovalo.

Grafický systém standardní knihovny je ale koncipován tak, že pro kreslení na nejrůznější cílové objekty slouží objekt typu `java.awt.Graphics` označovaný jako grafický kontext. Tento objekt program obdrží od objektu, na nějž chce kreslit (plátno, tiskárna, obrázek, …), a prostřednictvím obdrženého grafického kontextu pak na něj kreslí.

Instance třídy `Kreslítko` jsou takovými obaly kolem obdrženého grafického kontextu. Kdykoliv kreslítko požádáte o nějakou akci, volaná metoda převede požadavek na požadavek grafickému kontextu, který jej pak realizuje.

344. Tato implementace mi připomíná tlumočníka. Ty mu něco řekneš a on to tvému protějšku sdělí tak, aby tomu rozuměl.

Analogie: tlumočník

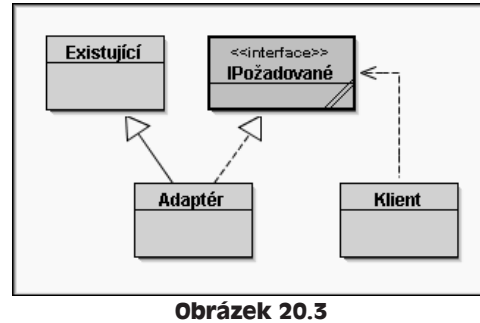
To je dobré přirovnání. Když ještě doplníš, že se tlumočník baví s tvým protějškem po telefonu, takže protějšek netuší, že se ve skutečnosti baví s tlumočníkem a ne s tebou, bude přirovnání téměř dokonalé.

Adaptér jako potomek adaptované třídy

345. Jak to vypadá s tou třetí implementací?

Způsob implementace

Diagram tříd třetího způsobu implementace si můžeš prohlédnout na obrázku 20.3. Při něm definuješ adaptér jako potomka existující, adaptované třídy a současně deklaruješ, že bude implementovat požadované rozhraní. Všechny metody požadované tímto rozhraním pak definuješ prostřednictvím volání metod adaptované třídy.



Obrázek 20.3
Adaptér jako potomek adaptované třídy

346. Není mi zcela jasné, proč tu vedle předchozí implementace s atributem ukazuješ implementaci s děděním, když jsi mi kdysi kladl na srdce, abych dával přednost skládání objektů před jejich děděním.

Přiznám se, že i mně je předchozí implementace prostřednictvím atributu sympatičtější. Na rozdíl od implementace prostřednictvím dědění totiž umožňuje, aby adaptér sloužil nejenom třídě, pro niž byl vytvořen, ale také kterémukoliv z jejích potomků.

Nevýhoda: nepodporuje dědičnost u adaptované třídy

Implementace využívající dědičnosti takovéto sdílení neumožňuje. Ta je doopravdy jednoúčelová. Všichni případní potomci třídy `Existující` z obrázku 20.3 budou „sourozenci" třídy `Adaptér`, takže jsou touto třídou neadaptovatelní.

347. Tak proč jsi ji tu uváděl, když se ti nelíbí?

Proč je uvedena

Pro uvedení této implementace jsem měl dva důvody:

– uvádí ji i GoF

- První důvod je trochu alibistický: v GoF jsou uvedeny obě implementace, takže nelze vyloučit, že se ve své praxi setkáš i s implementací využívající dědičnost, a bylo by vhodné, abys byl na tuto možnost připraven.

– umožňuje začlenění do dvou rámců

- Druhý důvod je praktičtější: Jsou situace, kdy potřebuješ začlenit instance dané třídy do dvou rámců (framework) současně, přičemž každý z těchto rámců bude vyžadovat implementaci jiného rozhraní. Přitom současně potřebuješ, aby v obou rámcích vystupovala tatáž instance, a ne jednou instance dané třídy a podruhé její adaptér.

  Implementuje-li daná třída jedno z těchto rozhraní, můžeš definovat adaptér implementující druhé rozhraní jako jejího potomka. Instance adaptéru pak budou implementovat obě rozhraní, takže mohou vystupovat v obou rámcích.

  Instance třídy `Adaptér` z obrázku 20.3 implementují jak rozhraní deklarované třídou `Existující`, tak rozhraní, jež deklaruje `interface IPožadované`. Mohou tedy vystupovat plnohodnotně ve dvou rámcích s nekompatibilními požadavky na implementovaná rozhraní.

348. A to by nešlo realizovat pomocí atributu?

Proč to nemusí jít s atributem

Šlo, ale musel bys v adaptéru definovat obě sady metod (takto jednu sadu podědíš). To by ale neměl být velký problém. Horší situace nastane, bude-li rodičovská třída `Existující` součástí nějaké knihovny, která její instance používá a do jejíhož zdrojového kódu nemůžeš zasahovat. Pak ti nezbude, než se „snížit" k využití dědičnosti a dodávat knihovním metodám místo existujících instancí instance jejího adaptovaného potomka.

349. Dobrá. Předpokládejme, že adaptovaná třída je mojí vlastní třídou. Jak bych mohl vyřešit adaptaci bez dědičnosti?

Nejjednodušší je samozřejmě implementovat požadované druhé rozhraní přímo onou existující třídou a definovat v ní obě sady metod.

Nevýhoda při setkání dvou stejných metod s různým účelem

Drobnou nevýhodou tohoto řešení je, že se může stát, že existující a požadované rozhraní deklarují metodu (nebo dokonce několik metod), která má v obou definicích stejnou signaturu (tj. stejně se jmenuje a mají stejný počet a typy parametrů), ale každá dělá něco jiného – např. jedno rozhraní bude definovat pro urychlení činnos-

ti roboty metodu `zrychli()` a pro přidání značky metodu `přidej()`, kdežto druhé bude zrychlovat po zavolání metody `přidej()` a pokládat značku po zavolání metody `polož()`.

Přiznejme si, že to je ale specialita, která by se v dobrém návrhu neměla vyskytnout, a tak se ji tu ani nebudeme snažit řešit.

Příklad

350. Na závěr by to chtělo ještě nějaký příkladeček.

Příklad k této kapitole je uveden až na konci kapitoly *Baťovy cvičky (Prototyp – Prototype)*, a to na straně 285. V příkladu si všimni, že třída sice teoreticky zdědí všechny metody od svého rodiče, avšak klíčové metody musí překrýt, aby mohla předat požadavek instanci, pro niž vytváří obálku a kterou uchovává jako atribut.

Další příklad adaptéru najdeš ve výpisu rozhraní `IPřizpůsobivý` na straně 383. Tentokrát se bude jednat o adaptér rozhraní, který je definován jako jeho vnořená třída.

Shrnutí – co jsme se naučili

- Návrhový vzor *Adaptér* využijeme ve chvíli, kdy bychom potřebovali, aby třída měla jiné rozhraní než to, které právě má.
- *Adaptér* slouží jako prostředník mezi prostředím, které požaduje nějaké rozhraní, a třídou, jejíž rozhraní neodpovídá požadovanému. Umožní tak spolupráci třídám, které by jinak spolupracovat nemohly.
- Tento návrhový vzor bývá občas označován jako *Obal* (*wrapper*).
- *Adaptér* je možno v principu implementovat třemi různými způsoby:
  - Jako společného rodiče všech adaptovaných tříd.
  - Jako třídu implementující požadované rozhraní, jejíž instance mají ve svém atributu uložen odkaz na adaptovanou instanci.
  - Jako třídu implementující požadované rozhraní, která je současně potomkem adaptované třídy.
- Implementace prostřednictvím společného rodiče se používá při definici nových tříd, které nemusí být zařazeny do žádného stromu dědičnosti.
- Implementace prostřednictvím atributu může být aplikována na původní adaptovanou třídu a na libovolného z jejích potomků.
- Implementace využívající dědičnosti lze využít při nutnosti, aby instance implementovala jak své původní, tak nově požadované rozhraní.
- Návrhový vzor *Adaptér* patří mezi vzory uvedené v GoF.

KAPITOLA 21

Bloudění strukturou (Strom – Composite)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Sjednocuje typy používaných objektů a umožňuje tak jednotné zpracování každého z nich nezávisle na tom, jedná-li se o atomický (tj. dále nedělitelný) objekt nebo o objekt složený z jiných objektů.

[1] **Definice v GoF:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. – Skládá objekty do stromových struktur reprezentujících hierarchie typu část-celek. *Strom* umožňuje klientům pracovat s jednoduchými i složenými objekty jednotně.

Účel

351. Tentokrát začnu záludným dotazem: proč překládáš termín *Composite* jako *Strom*?

Termín *Composite* bychom mohli doslovně přeložit jako *směsice*, *kombinace* – obecně znamená něco, co je z něčeho složené. Abych ti vysvětlil svůj překlad, začnu nejprve trochou teorie.

Vlastnosti struktury *Strom*

Termínem *Strom* označujeme matematickou strukturu složenou z tzv. *uzlů*, *listů* a *větví*. Z každého *uzlu* může vybíhat několik *větví*, přičemž na druhém konci *větve* může být buďto další *uzel*, v němž se může strom dále větvit, anebo *list*, což je místo obsahující konečnou hodnotu.

Ze zkušenosti znáš jistě stromovou strukturu souborů a složek (uživatelé Linuxu preferují termín *adresář*) na disku. Uzly tohoto stromu jsou jednotlivé složky, jeho listy jsou pak soubory.

352. A kde jsou větve?

Větve jsou pomyslné spojnice označující relaci, že daný soubor nebo složka patří do jiné složky.

353. Nu dobrá, ale jak to souvisí s tím překladem?

Proč překládám *Composite* jako *Strom*

Vtip je v tom, že tento návrhový vzor doporučuje definovat jednotný typ pro všechny prvky dané „směsice“ nezávisle na tom, jsou-li to atomické, tj. dále nedělitelné prvky, anebo se jedná o prvky složené z několika jiných prvků. Takovéto skládání prvků z jiných prvků lze nejlépe popsat právě stromem, kde atomické prvky vystupují jako listy a složené prvky jako stromy.

Proto jsem také pro název tohoto návrhového vzoru použil termín označující matematickou strukturu, která celé to uspořádání nejlépe popisuje.

Použitá struktura nemusí být vždy stromem

Pravda, v některých situacích nemusí být konkrétní uspořádání reprezentovatelné stromem. Strom totiž vyžaduje, aby do každého uzlu a listu vedla pouze jedna větev. Odcházet jich může, kolik chce, ale přicházet může jenom jedna. Tohle pravidlo ale bude porušeno ve chvíli, kdy bude nějaký prvek patřit do více skupin, což se někdy v praxi stát může.

Orientovaný graf

Takovou strukturu, při níž může do každého uzlu vcházet i vycházet libovolný počet větví, označujeme jako orientovaný graf. Jenomže na rozdíl od stromu tuto strukturu málokdo zná, takže v praxi bývá i orientovaný graf často označován za strom[1].

Kromě toho, přečteš-li si původní popis tohoto vzoru v GoF, tak tam autoři dokonce výslovně hovoří o stromových strukturách. Ona totiž hierarchická struktura nereprezentovatelná stromem zavání často špatným návrhem. Nemusí tomu tak být, ale často tomu tak je.

---

[1] Příkladem mohou být rozbory různých strategií procházení „stromu pozic“ v deskových či karetních hrách. Tento strom pozic je ve skutečnosti orientovaným grafem, protože v řadě případů se lze do jedné cílové pozice dostat z několika pozic výchozích.

354. A proč tedy tento název nezavedli i původní autoři vzoru?

Proč termín *Strom* nepoužívá GoF

Nevím. Asi jim použitý termín připadal méně svazující. Třeba při tom měli opravdu na mysli možnost, že obecně lze daný návrhový vzor použít i na obecnější struktury.

Já si ale naopak myslím, že mezi programátory je tento termín dostatečně známý, a že jim proto lépe přiblíží, o co vlastně v daném návrhovém vzoru jde.

355. Dejme tomu, že jsi mne přesvědčil. Teď mi ještě vysvětli, proč mají mít všechny prvky společný typ, když jsou přece tak různé.

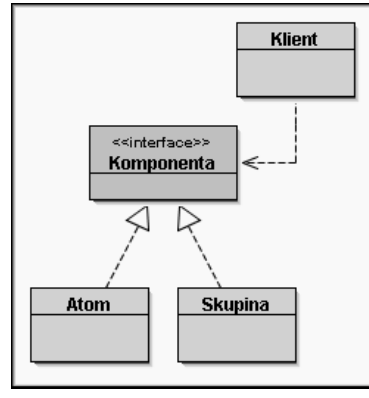
Proč mají mít všechny prvky společný typ

Aby se ti s nimi snadněji pracovalo, a aby proto byly obslužné programy jednodušší (diagram tříd tohoto návrhového vzoru si můžeš prohlédnout na obr. 21.1). Vrátím se k příkladu se stromovou strukturou souborů na disku. Jak jistě víš, ve standardní knihovně zastupují jak datové soubory, tak složky instance třídy `java.io.File` (standardní knihovna v tomto případě aplikuje návrhový vzor *Strom*). Složka je zde chápána jako soubor, jehož obsahem je seznam souborů, které do dané složky patří.

Když chceš zjistit, jak se daný objekt jmenuje, kdy byl vytvořen, jaká jsou k němu přístupová práva či některé jiné společné informace, bývá ti jedno, jedná-li se o datový soubor nebo složku. To tě začne zajímat až v okamžiku, kdyby ses zajímal např. o velikost daného objektu anebo o jeho obsah.

Častá začátečnická chyba

Základní chybou mnoha začátečníků totiž bývá to, že potřebují-li pracovat s nějakými skupinovými objekty, definují je pouze jako skupiny atomických objektů a vůbec neuvažují o tom, že by členem skupiny mohl být také skupinový objekt.



Obrázek 21.1
Diagram tříd v návrhovém vzoru Strom

Implementace

Zavést společného rodiče atomických i skupinových objektů

356. To znamená, že budu-li definovat typ objektů sestávajících ze skupiny jiných objektů, měl bych k němu ještě definovat rodičovský typ, který bude společným rodičem jak atomických, tak skupinových objektů.

Přesně tak. Prvky skupinového objektu budou instancemi onoho rodičovského typu. Dokud nebudeš potřebovat nějakou informaci či službu, kterou ti může poskytnout

pouze instance některého z dceřiných typů, můžeš se ke všem objektům chovat jednotně.

Budeš-li potřebovat něco, co ti všichni poskytnout nemohou, zjistíš si skutečný typ daného objektu a podle toho se pak zařídíš.

357. Třída `java.io.File`, kterou jsi před chvílí dával za příklad tohoto návrhového vzoru, však nemá žádného takového společného předka.

Příklad: `java.io.File`

Třída `File` je zrovna trochu zvláštní. Dokumentace o ní např. tvrdí, že nezastupuje soubory či složky, ale že je abstraktní reprezentací cesty, tj. jejich názvu. Název je vždy atomický, takže by to na první pohled vypadalo, že se v tomto případě nedá o nějakém stromu vůbec hovořit.

Odhlédneme-li však od těchto prohlášení a podíváme-li se na sadu metod v jejím portfoliu, začne se nám její význam jevit trochu jinak: instance třídy `File` můžeme považovat i za reprezentanty souborů a složek.

Obchází potřebu společného rodiče

Oproti klasické podobě návrhového vzoru však tentokrát autoři postupovali trochu jinak a definovali pouze jeden typ, který je společným typem jak pro atomické, tak pro skupinové objekty[1]. Některé metody sice mají význam jenom pro jeden druh (např. nemá smysl se ptát souboru, jaké soubory obsahuje), nicméně metody jsou definovány tak, že nezávisle na své podstatě se objekt s voláním metody korektně vypořádá (např. v předchozím případě vrátí prázdný odkaz).

Dotahuje zásadu maximalizovat společnou část rozhraní

Je to implementace, která se v učebnicích většinou nepopisuje, ale v některých situacích se může jevit velice užitečná. Každopádně tato implementace dotahuje „do vítězného konce“ zásadu uvedenou v GoF, tedy to, že je vhodné maximalizovat společné rozhraní uzlů a listů. Třída `File` tuto společnou část maximalizovala natolik, že už vedle ní nic nezbylo. Logickým důsledkem této skutečnosti je příjemné zjištění, že k takto definovanému společnému rodiči již není třeba definovat žádné potomky.

358. Vrátím se k oné „učebnicové“ implementaci se společným rodičem. Může být tímto společným rodičem i rozhraní?

Rozhraní jako společný rodič

Samozřejmě – ostatně na obrázku 21.1 je tento „společný rodič“ rozhraním. Většinou se však používá abstraktní třída, protože se nakonec ukáže, že by atomické i skupinové objekty mohly část implementace sdílet.

359. Říkal jsi, že návrhový vzor *Strom* je možno použít i pro struktury, které nejsou čistým stromem, ale pouze orientovaným grafem. Platí pro ně něco speciálního?

Implementace orientovaných grafů

Orientované grafy nebývá vhodné převádět na stromy, protože se tím příliš zvyšují paměťové nároky. Na druhou stranu práce s uzly, které mohou mít několik rodičů, také nebývá nejjednodušší. Jednou z možností, jak tento problém řešit, je využít návrhového vzoru *Muší váha* a přestěhovat informaci o rodiči daného uzlu do vnějšího stavu, který se metoda instance v případě potřeby dozví ze svého parametru.

---

[1] Ostatně před chvílí jsem říkal, že složka je vlastně pouze speciální typ souboru – soubor obsahující seznam souborů nacházejících se v dané složce.

Příklad

360. Myslím si, že už je mi všechno jasné a že bych si to teď rád ověřil na nějakém příkladu.

Jeden příklad jsem ti již uváděl – je jím třída `File`. Pravda, je to příklad trochu nestandardní, protože nepoužívá typickou implementaci, ale zároveň jej můžeš chápat jako tip, který se někdy může hodit.

Příklad pracující s touto třídou a návrhovým vzorem *Strom* jsem již předváděl v kapitole *Moc se mi v tom nehrab (Iterátor – Iterator)*. Jejich zdrojové kódy začínají na stránce 210.

Příklady v `java.awt`

Druhý příklad si také vypůjčím ze standardní knihovny, a to z oblasti tvorby GUI. V knihovně AWT je definována třída `java.awt.Component`, která je společným rodičem všech zobrazitelných prvků.

Její podtřídou je třída `java.awt.Container`. Její instance představují objekty složené z několika jiných objektů (např. v okně můžeme mít několik tlačítek a vstupních polí). Objekty vkládané do kontejneru musí být instancemi třídy `Component`. Tím je tedy automaticky dáno, že do kontejneru můžeme vložit jiný kontejner (a také to velice často děláme), protože i kontejner je komponenta.

Příklady v `javax.swing`

Návrhový vzor *Strom* je použit znovu v knihovně *Swing*. Tato knihovna definuje pro všechny součásti svých oken svého vlastního společného rodiče `javax.swing.JComponent`. Třída `JComponent` je však potomkem třídy `Container`, z čehož vyplývá, že libovolná swingová komponenta („j-komponenta“) může obsahovat několik komponent. Příkladem je např. tlačítko, které může mít na sobě vedle textu nakreslenou i nějakou ikonu.

361. Uznávám, že tvé příklady ze standardní knihovny jsou inspirující, ale přivítal bych ještě trochu kódu.

Příklad: Mnohotvar

No dobrá. Ve své učebnici [32] jsem v kapitole o kontejnerech definoval třídu `Mnohotvar`, jejíž instance představovaly složitější grafické tvary složené z několika tvarů jednodušších. Třída `Mnohotvar` byla potomkem třídy `rup.česky.tvary.AHýbací` a instancemi třídy `AHýbací` byly i všechny tvary, z nichž byl mnohotvar složen. Součástí mnohotvaru proto mohl být i jiný mnohotvar.

Protože jsem v této třídě současně ukazoval použití návrhového vzoru *Prototyp*, zařadil jsem celý příklad až na konec kapitoly *Baťovy cvičky (Prototyp – Prototype)*. Chceš-li se na něj podívat hned, najdeš jej na straně 297.

Až jej budeš procházet, tak si všimni, že v uvedené třídě je řada metod, které pracují s jednotlivými komponentami mnohotvaru, přičemž se nestarají o to, je-li daná komponenta jednoduchým tvarem nebo opět mnohotvarem. Všechny komponenty se totiž vydávají za instance rozhraní `IHýbací`.

Shrnutí – co jsme se naučili

- Návrhový vzor *Strom* sjednocuje typy objektů ve složitější (povětšinou stromové) struktuře a umožňuje tak jejich jednotné (často rekurzivní) zpracování.
- Datová struktura, na jejíž prvky je návrhový vzor aplikován, nemusí být nutně stromem.
- Začátečníci často omezují typy prvků struktury na atomické typy.
- Návrhový vzor *Strom* doporučuje definovat jak pro atomické, tak pro složené prvky společný datový typ, který by tyto rozdíly zakrýval.
- Rozhraní společné pro atomické i složené typy by mělo být co největší.
- Třída `java.io.File` tuto zásadu dotahuje do krajnosti a definuje pro obě skupiny objektů, tj. pro soubory i složky, společný datový typ `File`.
- Dalším příkladem aplikace tohoto návrhového vzoru je třída `java.awt.Component` jako společný předek všech grafických prvků použitých v GUI.
- Při aplikaci vzoru na orientovaný graf je možno na sdílené komponenty aplikovat vzor *Muší váha* a definovat jejich rodiče v rámci externího stavu.
- Návrhový vzor *Strom* patří mezi vzory uvedené v GoF.

ČÁST 5

Vytvořte to univerzální

V této části se znovu vrátíme ke vzorům souvisejícím se získáváním odkazu na instance. V části *Ovlivňujeme počet instancí,* začínající na straně 101, jsme se zabývali vzory, které řeší problémy související s ovlivňováním počtu vzniklých instancí. U většiny z nich jsme řešili danou úlohu znepřístupněním konstruktoru a jeho nahrazením jednoduchou tovární metodou.

V této části se budeme věnovat dalším aspektům tvorby instancí, především pak oddělení vlastní tvorby od požadavků na vytvoření instance, resp. na získání odkazu na ni. I zde si ukážeme, že jedním z nejvýhodnějších postupů je zapouzdření konstruktorů a jeho nahrazení jinými mechanismy. Mechanismy, které díky uvolnění vazby mezi třídou a jejím uživatelem výrazně zvýší budoucí modifikovatelnost aplikace.

KAPITOLA 22

Střihni mi to na míru (Tovární metoda – Factory Method)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Deklaruje rozhraní s metodou pro získání objektu. Rozhodnutí o konkrétním typu vráceného objektu však ponechává na svých potomcích, tj. na překrývajících verzích deklarované metody.

[1] **Definice v GoF:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. – Definuje rozhraní pro vytváření objektů, avšak rozhodnutí o tom, jakého typu bude vytvářená instance, nechává na podtřídách. *Tovární metoda* přenechává vytváření instancí podtřídám.

Účel

362. O tovární metodě jsme se již bavili někde na počátku knihy.

Obecnější verze jednoduché tovární metody

Tehdy, tj. v kapitole *Co konstruktor neumí (Jednoduchá tovární metoda – Simple Factory Method)*, začínající na straně 57, jsme hovořili o *jednoduché tovární metodě* (někdo ji označuje také jako *statická*).

Všechny tovární metody slouží k získávání odkazu na instance. *Jednoduchá tovární metoda* vrací povětšinou odkaz na instanci třídy, jejíž je statickou metodou. Využívá ji např. základní podoba jedináčka.

363. A co je na té „nejednoduché“ tovární metodě tak složitého, že jsi je nemohl probrat hned obě jedním vrzem?

V čem se liší

Především to, že standardní *Tovární metoda* bývá deklarována jako abstraktní metoda (nejčastěji v rozhraní) a neméně „abstraktní“ je i typ její návratové hodnoty – většinou to bývá také nějaké rozhraní.

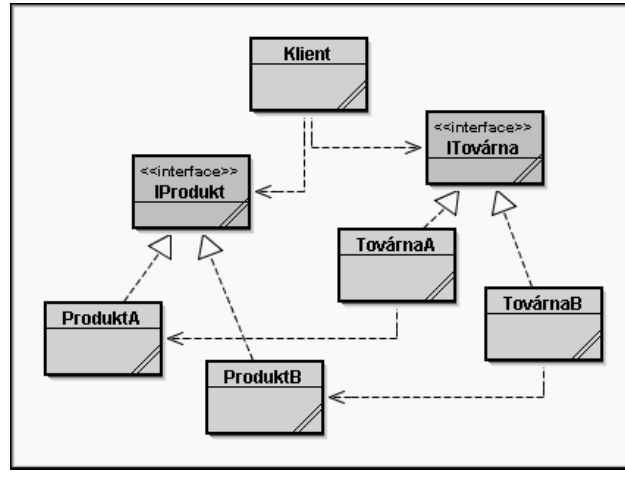
Příklady:

364. Řekl bych, že by bylo nejlepší to opět osvětlit na nějakém příkladu.

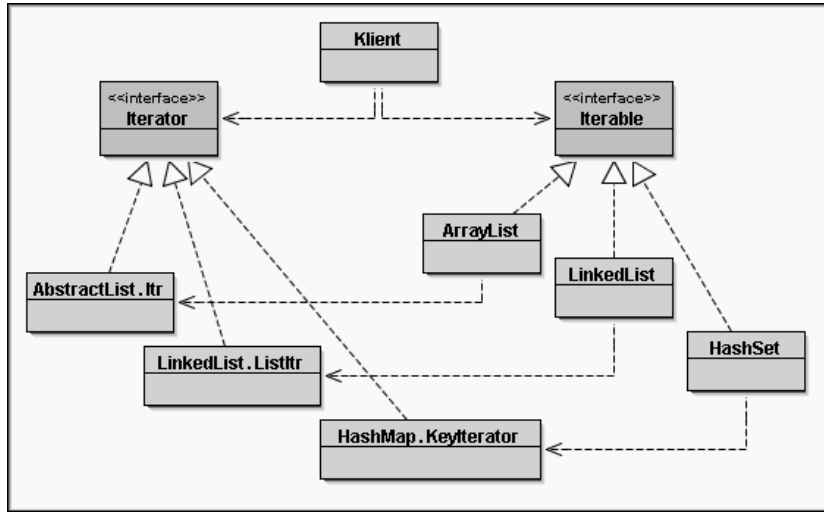
– metoda `iterator()`

Typickým příkladem tovární metody je metoda `iterator()` definovaná v rozhraní `java.lang.Iterable` a prostřednictvím svého dceřiného rozhraní `java.util.Collection` pak ve všech kontejnerech.

O iterátorech jsme se již bavili v kapitole *Moc se mi v tom nehrab (Iterátor – Iterator)*, začínající na straně 203. Podívej se na obrázek 22.1, na němž je principiální diagram tříd pro tento vzor, a pak si prohlédni obrázek 22.2, v němž jsem za jednotlivé třídy dosadil jejich ekvivalenty ze standardní knihovny.



Obrázek 22.1
Diagram tříd návrhového vzoru Tovární metoda



Obrázek 22.2
Aplikace tovární metody v knihovně kontejnerů

Rozhraní `Iterable` pouze deklaruje, že instance všech jeho implementujících tříd budou nabízet tovární metodu `iterator()`, která vrátí odkaz na instanci třídy implementující rozhraní `java.util.Iterator`. Co doopravdy vrátí, o tom se zde nemluví. Každý kontejner vrací takový objekt, který je schopen iterovat nad jeho daty co nejefektivněji.

Objednáš-li si iterátor a dostaneš jej, nemusí tě přitom zajímat, co je obdržený iterátor ve skutečnosti zač. Tobě by mělo stačit, že víš, že je to iterátor. To, že třída `LinkedList` vrátí instanci své vnitřní třídy, třída `ArrayList` instanci vnitřní třídy svého rodiče a třída `HashSet` instanci obdrženou od úplně jiné třídy, je pro daný účel nezajímavé.

365. Jak je to s iterátorem, je jasné. Máš ještě nějaký další příklad?

– metoda `getGraphics()`

Implementací tovární metody je ve standardní knihovně celá řada. Dalším typickým představitelem je objekt typu `java.awt.Graphics`, který slouží jako kreslítko, jímž se jiné objekty někam nakreslí (většina učebnic dává přednost učeněji znějícímu termínu *grafický kontext*). Chceš-li se někam nakreslit, musíš nejprve získat od cílového objektu kreslítko (= objekt `Graphics`), s jehož pomocí se nakreslíš. Abstraktní třídy `java.awt.Image` či `java.awt.Component` definují metody `getGraphics()`, které každý z jejich potomků implementuje dle svých potřeb.

– metoda `newInstance()` v definici fondu

Tovární metodu jsme použili i v případě fondu. Vzpomeň si na rozhraní `ITovárna` (viz výpis 12.2 na straně 155). Fond tehdy potřeboval generátor instancí, ale neměl dopředu tušení o tom, jakého typu dané instance budou. Deklarovali jsme proto příslušný generátor jako instanci rozhraní `ITovárna`, která deklarovala tovární metodu `newInstance()`. Pro každou třídu, jejíž instance jsme chtěli ukládat do fondu, jsme museli definovat její vlastní továrnu (tj. třídu implementující rozhraní `ITovárna`), jejíž instanci jsme předali fondu jako generátor instancí naší třídy.

366. Už jsem pochopil, že se používá opravdu často. Odhaduji proto, že vedle toho, že umožňuje optimalizovat rozhodnutí, jaký má být typ vytvářeného (nebo alespoň vraceného) objektu, určitě přináší i další výhody.

Umožňuje ignorovat konkrétní typy objektů

Použití vzoru *Tovární metoda* umožňuje ignorovat v programu konkrétní typy objektů, s nimiž se bude pracovat. Tebe nezajímá, jestli ti seznam vrátí iterátor takového nebo makového typu. Ty víš, že ti vrátí iterátor, a to je vše, co potřebuješ vědět. Jestli v příští verzi knihovny vymění typ vraceného iterátoru, to tě nevzrušuje. Pořád to bude iterátor, takže tato změna tvůj program nijak neovlivní.

Umožňuje ignorovat implementaci a soustředit se na rozhraní

Tovární metoda ti tak umožňuje ignorovat konkrétní implementaci vytvářených objektů a soustředit se na jejich rozhraní. Program, který získává instance od továrních metod, je mnohem snáze modifikovatelný než ten, který si dané instance vytváří sám. Vymění-li tovární metoda typ vraceného objektu za nějaký jiný, avšak kompatibilní (tj. implementující deklarované rozhraní), není třeba volající program nijak měnit. V případě přímého volání konstruktoru se však těmto zásahům nevyhneme.

Zapouzdří vytváření instancí

Tovární metoda zapouzdřuje vytváření instancí. Při použití tovární metody jsi ušetřen starostí o to, jaká má být přesně třída vytvářené instance, jaké je třeba jejímu konstruktoru předat parametry atd. Nemusíš se proto soustředit na to, co přesně potřebuješ vytvořit, ale k čemu chceš vytvářený objekt použít.

Zlepšuje zapouzdření zanořených tříd

Tovární metoda zlepšuje zapouzdření soukromých instancí zanořených tříd implementujících tebou požadované rozhraní. Kdybys požadoval přístup k jejich konstruktoru, nesměly by tyto třídy být soukromé a opět by se snížilo zapouzdření knihovny.

367. Zadrž. Říkáš tu různými slovy stále totéž. Pochopil jsem z toho, že existuje řada případů, kdy je výhodnější definovat pro každý bod nějaké hierarchie dvě třídy: jednu pro používané instance a jednu pro instance s jejich továrními metodami.

Pro tovární metodu není vždy nutno definovat zvláštní třídu

Nemusí být vždycky dvě. V řadě případů již druhou skupinu tříd dávno máš a stačí je opravdu jen doplnit továrními metodami. Vezmi si např. metody pro tvorbu iterátorů – ty jsou definovány ve třídách, které by beztak byly součástí tvé aplikace.

Instance s tovární metodou je zástupcem mezi konstruktorem a zbytkem aplikace

Řeknu ti to ještě jinak: ber to tak, že instance s továrními metodami jsou jacísi zástupci (viz kapitolu *Pod ruce mi neuvidíš (Zástupce – Proxy)* na straně 189) mezi tvým programem a konstruktory instancí, které chce tvůj program používat. Tak jako u jiných zástupců je i zde celá řada situací, kdy se zavedení takovéhoto zástupce velice vyplatí.

368. Zeptám se ještě z druhé strany: kdy bych měl začít přemýšlet o tom, že mám k některým třídám vytvářet jejich parťáky s továrními metodami?

Kdy sáhnout po tovární metodě

Například tehdy, budeš-li mít nějakou skupinu tříd (výrobků) a budeš chtít klienta osvobodit od toho, aby si přesně pamatoval, která třída se hodí pro tu kterou situaci.

Vezmi si náš příklad s iterátory: nemusíš si pamatovat, která ze tříd iterátorů pracuje nad daným typem kontejneru. Prostě požádáš kontejner, aby ti dodal správný iterátor.

Naprosto stejné je to i s grafickým kontextem. Opět nemusíš přemýšlet nad tím, kam se budeš kreslit jakým kreslítkem. Požádáš objekt o jeho kreslítko (grafický kontext), on ti je dodá a můžeš se začít kreslit.

Tovární metoda může zjednodušit rozhraní aplikace

Z obou předchozích příkladů je zřejmé, že vhodným zavedením tříd s továrními metodami, které ti zprostředkují získání těch správných objektů, můžeš výrazně zjednodušit rozhraní své aplikace.

Je vhodné na ni myslet i tehdy, když se zmnožení tříd teprve chystá

Tovární metodu můžeš využít i tehdy, když sice ještě žádnou větší skupinu tříd nemáš, ale očekáváš, že by na ni při dalším vylepšování programu mohlo dojít. Pak samozřejmě může být tovární metoda konkrétní (tj. nebude abstraktní) a poskytovat implicitní implementaci tovární metody, kterou její případní potomci v případě potřeby operativně překryjí.

Příklad: knihovna kontejnerů

Představ si, že vybuduješ první kontejner. Zatím je jen jeden, ale víš, že brzy jich bude celá hierarchie. Nepřipravíš si proto iterátor jako nějakou samostatnou třídu, ale definuješ tovární metodu, která jej žadateli vrátí. Jak se bude časem rozrůstat množina různých specializovaných kontejnerů, bude se rozrůstat i knihovna příslušných iterátorů. Tovární metoda ušetří uživatele této hierarchie nutnosti pamatovat si, který iterátor je třeba použít pro právě používaný kontejner.

Tovární metoda aplikuje princip programování proti rozhraní

A poslední poznámka: podíváš-li se na obrázek 22.1, pochopíš, že tovární metoda je realizací zásady, že nemáš programovat proti implementaci, ale proti rozhraní.

Implementace

369. Říkal jsi, že tovární metoda bývá deklarována v rozhraní nebo abstraktní třídě. Má někdy smysl ji deklarovat i v konkrétní třídě?

Implementace v konkrétní třídě

Má, a to tehdy, existuje-li nějaká implicitní implementace této tovární metody. Implementace, která v řadě případů stačí a pouze ve speciálních situacích bývá nahrazena nějakou překrývající metodou potomka.

370. Pomalu začínám tušit, jak to s tou „standardní“ tovární metodou je. Pro jistotu mi ale popiš typický postup.

Typický postup přípravy:

Dobře. Představ si, že máš definovánu skupinu souvisejících tříd, které vytvářejí instance nějaké třídy či rozhraní, avšak každá z nich vytváří instanci jiného konkrétního typu, který je potomkem, resp. implementací, společného rodiče.

– společný rodič výtvorů

– společný rodič tvůrců

– tvůrci definují překrývající verze továrních metod

1. Abys s nimi mohl pracovat jednotně, definuješ společného rodiče (případně společné implementované rozhraní) vytvářených instancí.
2. Současně definuješ společného rodiče (společným rodičem může být i rozhraní) tvořivých objektů (továren) a v něm pak tovární metodu zodpovědnou za vytváření požadovaných instancí. Typem návratové hodnoty bude samozřejmě před chvílí definovaný společný rodič vytvářených instancí (produktů, výrobků).
3. Každý z potomků překryje tuto metodu vlastní verzí, která vrátí odpovídající objekt (produkt, výrobek).

371. Předpokládám, že obdobný postup platí i v situacích, kdy ještě žádnou skupinu souvisejících tříd nemám.

Samozřejmě – to je třeba příklad našeho fondu. Tam jsme si ale ušetřili definici společného rodiče vytvářených instancí, protože jsme místo něj použili typový parametr.

V této úloze jsme řešili situaci, že jsme dopředu vůbec nic nevěděli o tom, jaké konkrétní typy objektů budou generátory vytvářet. Deklarovali jsme proto pouze rozhraní, které musely implementovat třídy budoucích generátorů, a rozhodnutí o konkrétním typu vytvářeného objektu jsme ponechali na jeho generátoru.

Příklad

372. Řekl bych, že jsme si vše pověděli, a je tedy čas na příklad.

Tentokrát tě zklamu, protože se domnívám, že jsem na toto téma uvedl již dva příklady: tvorbu univerzálního fondu (viz pasáž *Univerzální fond* na straně 153) a tvorbu iterátoru (výpisy začínají na straně 206). O obou jsem se v předchozím textu této kapitoly zmiňoval.

373. U fondu je ale typ vytvářených objektů zadáván v typovém parametru. Ve stručné charakteristice jsi ale o společném rodiči říkal, že „rozhodnutí o konkrétním typu vráceného objektu ponechává na svých potomcích".

To, že je v definici rozhraní `IFond<T>` použit parametrický typ, nijak neomezuje možnost potomků rozhodovat o typu skutečně vráceného objektu (ostatně rozhraní `Iterable<T>` má také typový parametr). Můžeš definovat třídu implementující toto rozhraní s konkrétní hodnotou typového parametru. Takový fond pak nebude potřebovat, aby jeho konstruktor dostával nějakou tovární instanci jako parametr (viz definici třídy `Fond` ve výpisu 12.3 na straně 155), protože fond si bude umět vyrábět potřebné instance sám.

Shrnutí – co jsme se naučili

- *Tovární metoda* je obecnější verze jednoduché tovární metody. Není statická, ale je to metoda instance zodpovědné (mimo jiné) za vytvoření požadovaného objektu.
- *Tovární metoda* uvolňuje vazbu mezi třídou používající nějaký objekt a třídou tohoto objektu.
- *Tovární metoda* umožňuje se odpoutat od implementace používaného objektu a soustředit se na jeho rozhraní.
- Instance s tovární metodou funguje jako zástupce vložený mezi konstruktor vytvářených instancí a zbytek aplikace.
- V řadě případů není třeba definovat pro tovární metodu zvláštní třídu, ale stačí ji pouze přidat mezi metody používaných tříd (viz iterátor).
- Třídy, jejichž instance tovární metody vracejí, jsou často definovány jako zanořené třídy „továrních tříd".
- Návrhový vzor *Tovární metoda* patří mezi vzory uvedené v GoF.

KAPITOLA 23

Baťovy cvičky (Prototyp – Prototype)

- **Klonování a jeho vlastnosti**
- **Účel vzoru Prototyp**
- **Implementace**
- **Příklad: Mnohotvar**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Objekt definuje tovární metodu, která bude vytvářet jeho kopie. V programu je pak možno místo přímého volání konstruktoru využívat volání kopírovací tovární metody připraveného prototypu.

[1] **Definice v GoF:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. – Specifikuje druh objektů, které budou vytvářeny za použití prototypové instance, a současně vytváření těchto objektů kopírováním zadaného prototypu.

Klonování a jeho vlastnosti

374. Není mi zcela jasné, proč je třeba kolem metody, která je podle mne pouze takovou obálkou kolem konstruktoru, vytvářet návrhový vzor.

Ona to ve většině případů nebývá obálka kolem konstruktoru, protože metodu `clone()`, která je k tomuto účelu využívána, dědíš od třídy `Object`. Problematika klonování ale není tak jednoduchá, jak se někteří domnívají.

375. Tím chceš určitě naznačit, že patřím mezi ty některé. Musím přiznat, že v učebnicích, které jsem četl, se o něm moc nepsalo, takže přivítám, když je nejprve trochu probereme.

Volání konstruktoru není jedinou možností vytvoření instance

Pravdou je, že začátečnické učebnice a kurzy často úspěšně vyvolají ve svých čtenářích či frekventantech falešnou představu, že jediným slušným způsobem, jak je možno vytvořit instanci, je zavolat konstruktor. O dalších způsobech se nezmiňují nebo se o nich zmíní jen okrajově, takže jejich absolventi pak umí málokdy zabezpečit, aby zbylé dva způsoby konstrukce objektů nenabouraly důležité invarianty.

376. Proboha, co to jsou ty invarianty?

Co to jsou invarianty

To jsou vlastnosti, které by se neměly měnit, a tím nemyslím konstanty, ale i obecnější tvrzení. Počítáš-li si např. vytvořené instance a přiřazuješ-li jim rodná čísla odvozená od pořadí jejich vzniku, pak by mělo v každém okamžiku platit, že žádné dvě různé instance nebudou mít stejné rodné číslo.

Obdobně by např. mělo platit, že jedináček zůstane vždy jedináčkem – o tom jsme si ale povídali v podkapitole *Serializovatelnost* na straně 116.

Alternativní tvorba instancí:
– serializace + deserializace

377. To bych asi pochopil. Teď mi ještě prozraď, které jsou ty další způsoby vytváření instancí.

O jednom jsme se již bavili v souvislosti se serializovatelností. Je jím uložení objektu do nějakého výstupního proudu a jeho následné načtení. Při načítání objektu z proudu se primárně vytváří nové instance těchto objektů, Java nabízí i prostředky, jak tomu v případě potřeby zabránit. Částečně jsem se o tom zmiňoval v pasáži o jedináčkovi.

– klonování

Druhým způsobem náhradní výroby objektů je jejich klonování, o němž jsem hovořil před chvílí. Při něm vytváříš kopii objektu bez použití konstruktoru. Jak jsem ale již naznačil, ani s ním to není zcela průzračné a jednoduché.

378. Přiznám se, že mně to připadá jednoduché: zavolám metodu `clone()` a dostanu kopii objektu. Co je na tom složitého?

Metoda `clone()`

Jak jistě víš, metodu `clone()` definuje již třída `Object`, takže ji mají všechny instance k dispozici, protože ji přinejmenším zdědí. Ve třídě `Object` je ale metoda `clone()` deklarovaná jako chráněná (`protected`), takže zděděnou verzi mohou použít pouze objekty samy. Nemůžeš proto libovolný objekt jen tak naklonovat.

379. Ve stejném balíčku snad mohu – chráněné metody jsou přece v rámci balíčku viditelné.

Proč na ni není vidět

Viditelné jsou, jenomže zapomínáš, že chceš-li použít nějakou zděděnou metodu, obracíš se na objekt jako na instanci rodiče, který danou metodu definoval. Chceš-li

tedy vyvolat nepřekrytou metodu `clone()`, obracíš se na objekt jako na instanci třídy `java.lang.Object` a na její chráněné metody vidí jenom její potomci.

380. Vždyť všechny třídy jsou potomky třídy `Object`, takže by na její chráněné metody měli vidět všichni.

Instance vidí pouze chráněné metody svého podobjektu

Nerad bych ti tu dával lekce z programování v jazyku Java. Není to tak, jak říkáš. Instance má přístup pouze k chráněným členům svého rodičovského podobjektu. Nikdo jiný (samozřejmě kromě instancí tříd ze stejného balíčku) jí své chráněné metody nezpřístupní. Neučiní tak ani její atributy, které jsou instancemi stejné třídy jako ona sama. Jenom její rodičovský podobjekt.

Jak to definuje specifikace jazyka

Specifikace jazyka [34] říká: *K chráněnému (protected) členu či konstruktoru může z oblasti mimo balíček, v němž je tento člen deklarován, přistupovat pouze kód, který je zodpovědný za vytvoření daného objektu.* Já vím, že to z této formulace není přímo patrné, ale specifikace považuje objekt zodpovědný za vytvoření svého podobjektu, avšak nikoliv za vytvoření objektů, na něž budou ukazovat jeho atributy. Za jejich vytvoření je totiž zodpovědný konstruktor, který bude o jejich vytvoření požádán.

381. Přiznám se, že to moc nechápu.

Tak si prostě zapamatuj, že instance vidí pouze na své chráněné atributy, ale už ne na chráněné atributy svých atributů.

Důvod povinného překrytí

382. Nebudu se snažit pochopit, proč to tak je, ale tohle tvrzení si snad zapamatovat dokážu. Takže chci-li umožnit, aby mne mohl klonovat i někdo jiný než moji potomci, musím překrýt metodu `clone()`.

Někdy stačí i `protected`

Ano. Chceš-li své klonování zpřístupnit pouze třídám ze stejného balíčku a případným potomkům, stačí ji opět deklarovat jako `protected`, nicméně v drtivé většině případů je tato překrývající verze deklarována jako `public`.

Překrytí metody `clone()`

Většinou se však rozšiřují přístupová práva k metodě `clone()` na `public`. Vyhovuje-li ti přitom rodičovská verze metody, tj. vyhovuje-li ti její způsob klonování, stačí metodu definovat:

```
public Object clone() {
    super.clone();
}
```

Celé překrytí tak vlastně definuješ pouze kvůli zviditelnění a případně kvůli tomu, abys rozšířil přístupová práva z `protected` na `public`.

383. Lze klonovat instanci bez použití `super.clone()`?

Nepoužití `super.clone()`

Samozřejmě. Rozhodneš-li se, že ti kopírování instance nevyhovuje, můžeš je nahradit přímým vytvořením nové instance prostřednictvím volání konstruktoru.

384. Já jsem myslel, že klonování používáme k tomu, abychom se používání konstruktoru vyhnuli.

Proč zde volání konstruktoru nevadí

Voláš-li konstruktor z jiné třídy, obě třídy tím svazuješ. Proto se často dává přednost továrním metodám a klonům. Tohle ale není onen případ. Tady používá třída sama svůj konstruktor. Rozhodneš-li se později definici konstruktoru jakkoliv upravit, můžeš současně upravit i její klonovací metodu (doporučuji ponechat v konstrukto-

ru pro jistotu komentář, aby na to úpravce nezapomněl). Všechny změny tak zůstávají v jedné třídě.

385. Jak vypadá klonovací metoda zděděná od třídy `Object`?

Nástin definice ve třídě `Object`

Metoda je sice definována jako nativní (tj. je definována přímo ve strojovém kódu příslušného procesoru), ale kdybychom se rozhodli převést ji do Javy, mohla by vypadat např. následovně:

```
public Object clone() {
    if( !(this instanceof Cloneable) )
        throw new CloneNotSupportedException();
    Object clone = allocateNew( this );  //Vyhradí místo v paměti
    byteCopy( this, clone );    //Zkopíruje do něj obsah instance
    return clone;
}
```

V této definici jsem použil dvě zcela smyšlené metody. Metoda

```
public Object allocate( Object );
```

vyhradila na haldě prostor pro další instanci stejného typu jako předaný parametr a vrátila na ni odkaz. Metoda

```
public void byteCopy( Object, Object );
```

pak předpokládá, že oba parametry jsou stejného typu, a zkopíruje obsah jednoho bajt za bajtem do druhého. Ještě jednou bych pro jistotu dodal, že obě metody jsem si vymyslel, abych co nejnázorněji ukázal, jak asi základní verze metody `clone()` pracuje.

386. Z té tvé definice jsem pochopil, že dokud nebudu mít své třídy deklarované jako implementující rozhraní `Cloneable`, tak mi ona zděděná verze stejně nic nenaklonuje.

Klonovat je možno pouze klonovatelné třídy

Správně. Jedinou výjimkou jsou pole, která jsou automaticky klonovatelná, i když to nikde explicitně nedeklaruješ (ono to také ani nejde – k tomu bys musel pole zabalit do nějakého vlastního typu).

Rozhraní `Cloneable`

Rozhraní `Cloneable` je ale pouze značkovací rozhraní. Nedeklaruje žádnou metodu, protože jedinou metodu, kterou ke klonování doopravdy potřebuješ, už jsi dávno zdědil od třídy `Object`.

387. Kdyby deklarovalo metodu `clone()`, tak by tím donutilo klonovatelné třídy, aby tuto metodu definovaly jako veřejnou.

Proč `Cloneable` nedefinuje `clone()`

K tomu, aby byl objekt klonovatelný, nemusí být jeho metoda `clone()` nutně veřejná. Bude-li ti stačit, aby byl klonovatelný pouze v rámci svého balíčku, bude ti chráněný přístup k této metodě docela postačovat.

Implementací rozhraní `Cloneable` třída pouze povoluje klonování svých instancí a současně slibuje, že udělala vše proto, aby se její instance správně klonovaly.

388. Zní to pěkně, ale nedá se z toho pochopit, co znamená to *„udělala vše proto, aby se její instance správně klonovaly“*.

Proč je někdy třeba definici `clone()` doplnit

Odpovím ti nepřímo. Podíváš-li se na moji fiktivní definici, uvidíš, že se obsah původní instance kopíruje do vyhrazené paměti bajt za bajtem. To znamená, že se kopírují hodnoty primitivních proměnných a současně i odkazy na objektové proměnné.

Jinými slovy: bude-li originál odkazovat na nějakou objektovou proměnnou, bude jeho klon vytvořený metodou zděděnou od třídy `Object` odkazovat na tentýž objekt.

To se nám sice často hodí, ale neméně často (no, spíš častěji) se nám to nehodí. Nehodí se to tehdy, když potřebujeme, aby klon odkazoval na své vlastní objekty a ne na objekty, které musí sdílet s někým jiným.

Druhy klonování:

V této souvislosti se hovoří o mělké a hluboké kopii.

– Mělká kopie

- Mělkou kopií (shallow copy) je klon vzniklý aplikací původní verze metody `clone()`, tj. klon, který má zkopírované hodnoty všech atributů svého vzoru, avšak nemá již zkopírované objekty, na něž tyto atributy odkazují. Odkazuje-li proto atribut vzoru na nějaký objekt, odkazuje odpovídající atribut jeho mělké kopie na týž objekt.

– Hluboká kopie

- Hlubokou kopií (deep copy) je pak klon, jenž má jednoduše zkopírovány pouze hodnoty atributů primitivních datových typů. Atributy objektových typů mají nové hodnoty, protože odkazují na nové objekty. Ty jsou většinou vytvořeny jako klony objektů, na něž odkazují odpovídající atributy vzoru. U těchto klonovaných atributů pak opět můžeme přemýšlet o tom, budeme-li potřebovat jejich mělké či hluboké kopie.

389. Mohl bys uvést nějaké příklady mělkých a hlubokých kopií?

Příklady mělké kopie

Mělkou kopii vytvářejí klonovací metody polí a kontejnerů ze systémové knihovny (tj. z balíčku `java.util`). Využiješ ji např. tehdy, vytváříš-li kopii daného kontejneru proto, aby zabezpečil, že v něm budeš mít uloženy stále stejné odkazy nezávisle na tom, co se bude dít s původním kontejnerem.

Kdy použít hlubokou kopii

Po hluboké kopii naproti tomu sáhneš v situaci, kdy potřebuješ, aby i uložený objekt byl v původním stavu a nikdo jiný nemohl tento stav změnit. Příkladem třídy, která vyžaduje používání hluboké kopie, je `Mnohotvar`, o němž jsme hovořili v kapitole *Bloudění strukturou (Strom – Composite)* a jehož zdrojový kód si můžeš prohlédnout ve výpisu 23.4 na straně 297.

390. Jestli jsem to dobře pochopil, tak u mělkých kopií vystačím se zděděnou verzí metody `clone()`, u hlubokých kopií musím definovat verzi vlastní.

Nepříjemné vlastnosti metody `clone()`

Ono to není tak jednoduché. Např. jsi zapomněl na to, že zděděná metoda je chráněná, takže není vždy jednoduše přístupná. Navíc vyhazuje kontrolovanou výjimku `CloneNotSupportedException`, jejíž povinné ošetřování obtěžuje.

Vynucení implementace veřejné metody nevyhazující kontrolovanou výjimku

Můžeš sice definovat rozhraní, které bude mít mezi vyžadovanými metodami i metodu `clone()` nevyhazující žádnou kontrolovanou výjimku a bude pracovat pouze s instancemi, jež toto rozhraní implementují, ale pak se zase dostaneš do problémů s používáním tříd, které přebíráš z jiných zdrojů (např. ze standardní knihovny) a které to tvoje rozhraní neimplementují.

Zpřístupnění existujících tříd použitím adaptéru

Opět to můžeš vyřešit např. použitím návrhového vzoru *Adaptér* (hovořili jsme o něm v kapitole *Je to trochu jinak (Adaptér – Adapter)* na straně 261), ale tím si zase přidělá práci s definicí jednotlivých adaptérů, takže co na jedné straně ušetříš, můžeš jinde ztratit.

Na druhou stranu, používáš-li metodu `clone()` u hodnotových objektových typů, můžeš využít toho, že z jistého hlediska jsou si všechny instance se stejnou hodnotou navzájem rovny, takže pokud vysloveně nepotřebuješ další instanci se stejnou

hodnotou, tak jejich metoda `clone()` vůbec nemusí vracet novou instanci, ale může být definována naprosto primitivně:

```
public Xxx clone() {
    return this;
}
```

391. Tak teď jsi mne zaskočil. Já myslel, že `clone()` musí vždy vytvořit kopii daného objektu.

Bývá to zvykem, ale povinné to není. Podíváš-li se do dokumentace, dozvíš se, že typicky sice platí

```
x.clone() != x
x.clone().getClass() == x.getClass()
x.clone().equals(x)
```

ale dodržení žádného z těchto pravidel není striktně vyžadováno.

Je zcela na tobě, o čem se rozhodneš, že to budeš vydávat za kopii objektu. Důležité je pouze to, aby toto tvé rozhodnutí nekolidovalo s žádným z tvých dalších „rozhodnutí" přijatých při návrhu aplikace.

Vezmeš-li si např. zásady návrhového vzoru *Originál* (viz kapitolu *Dvojníky nepotřebujeme (Originál – Original)* na straně 135), tam bys dokonce tuto metodu ani jinak definovat nemohl, protože bys porušil invariant třídy, který vyžaduje, aby pro každou hodnotu existovala pouze jediná instance.

392. Vidím, že nic není jednoduché.

Doporučená literatura

Bezpečné používání metody `clone()` vyžaduje citelně více znalostí, než kolik ti jich zprostředkují běžné začátečnické učebnice a kurzy. Potenciálních problémů s metodou `clone()` je celá řada. Některé jsme tu rozebrali, na řadu dalších upozorňuje kniha [41], resp. její český překlad [28].

393. Určitě se na ni někdy podívám. Nyní bych ale přivítal, kdybys to naše povídání nějak shrnul.

Shrnutí debaty o `clone()`

Řekl bych, že s používáním metody `clone()` je to obdobné jako s používáním dědičnosti a dalších dvojsečných konstrukcí. V řadě situací ti pomůže, ale na druhé straně na tebe připraví skrytá úskalí, s nimiž musíš dopředu počítat a na něž se musíš připravit.

Některá úskalí jsme tu rozebrali, další si můžeš prostudovat z literatury. Obecně však o ní platí to, co o většině konstrukcí: nepoužívat bez přemýšlení!

A ještě jedna rada: přistupovat ke standardní knihovně způsobem *„do dokumentace se dívám, až když vše ostatní selže"* je poměrně bezpečná cesta do programátorských pekel.

Účel vzoru *Prototyp*

394. Řekl bych, že o klonování jsme si toho již pověděli dost a že bychom se mohli vrátit k návrhovému vzoru *Prototyp*.

Chvála vzoru *Prototyp*

Máš pravdu. Návrhový vzor *Prototyp* využívá alternativní způsob vytváření objektů prostřednictvím továrních metod vytvářejících kopie existujících objektů. V řadě situ-

ací poskytuje skvělé možnosti zabezpečení snadné modifikovatelnosti aplikací, a to dokonce i za jejich chodu.

Opakování:

395. Odpusť si ty své plané marketingové řeči a raději mi vysvětli, co konkrétního tento vzor nabízí.

– Nevýhody konstruktoru

Začnu trochou opakování. Prozatím jsme se seznámili s konstruktorem a tovární metodou. Řekli jsme si, že konstruktor je sice jediným prostředkem, jak získat zcela novou instanci, avšak jeho přímé použití není vždy výhodné, protože příliš svazuje třídu, která o vytvoření instance žádá, s třídou, která tuto instanci vytváří. Žádající třída musí přesně vědět, kterou třídu žádá o vytvoření instance, a musí se na ni ve svých definicích metod přímo obrátit.

– Řešení tovární metodou

Těsnost tohoto spojení je ale v řadě aplikací nepřijatelná. Jednou z možností uvolnění této vazby je nahrazení volání konstruktoru voláním tovární metody, která je takovým zprostředkovatelem mezi metodou, která o objekt žádá, a volaným konstruktorem. Jakmile vznikne potřeba modifikace volání konstruktoru anebo se rozšíří spektrum tříd, jejichž instance lze k danému účelu použít, stačí změnit definice příslušných továrních metod. Není již třeba „obíhat“ všechny třídy, které o danou instanci žádají, a upravovat kód každé z nich.

Nevýhody předvedených továrních metod

Doposud předvedené tovární metody měly jednu společnou nevýhodu: volaly konstruktor tříd, jejichž instance vytvářely, a proto potřebovaly znát tyto třídy již v okamžiku svého vzniku.

V mnoha případech to nevadí, protože třída s tovární metodou i třída, jejíž instanci tato metoda vytváří, jsou beztak spojené (viz příklad iterátorů). V řadě situací bychom ale potřebovali, abychom mohli v průběhu aplikace operativně měnit druhy objektů, které budeme používat.

Hlavní výhodou návrhového vzor *Prototyp* je to, že nám umožňuje rozšiřovat spektrum typů generovaných objektů „za chodu“. Stačí někde získat instanci, která umí generovat své kopie, a máme vystaráno.

396. Počkej, počkej. To chceš naznačit, že budou za chodu vznikat nové třídy?

Klonování umožňuje dynamicky přidávat nové druhy objektů

To by sice teoreticky šlo, ale o tom jsem nemluvil. Spíše jsem měl na mysli to, že kód využívající kopírovacích schopností používaných instancí může být zásobován instancemi zcela neznámých typů. Metodě může být např. předán nějaký zcela nový druh objektu jako parametr a metoda si může bez problému vytvořit kopii daného objektu pro vlastní potřebu.

Těmito „novými druhy“ objektů nemusí být vždy nové třídy, ale i nové druhy objektů patřících do některé již existující třídy.

397. Tak to mi budeš muset vysvětlit na příkladu.

Příklad: `Mnohotvar`

Když ses před chvílí ptal na příklad použití hluboké kopie, tak jsem tě odkazoval na třídu `Mnohotvar` (její zdrojový kód najdeš ve výpisu na straně 297). Nyní se k ní znovu vrátím.

Dvě skupiny metod:
– sestavení objektu

Metody definované v definici třídy `Mnohotvar` bychom mohli rozdělit do dvou skupin:

- Nově definované metody, které slouží k sestavení objektu z jednodušších tvarů (konstruktor vytváří pouze prázdný objekt, který je třeba následně sestavit). Použitelnost těchto metod lze omezit na počáteční sestavování objektu, které ukončuje zavolání metody `sestaveno()`.

– „zprovoznění" objektu

- Metody překrývající původní verze metod zděděných z tříd `APosuvný` a `AHýbací` a zabezpečujících, že instance třídy `Mnohotvar` budou plnohodnotné *hýbací* objekty – mohli bychom o nich hovořit jako o metodách zabezpečujících plnohodnotné zprovoznění objektu.

Teď si představ, že bychom chtěli vytvořit nějaký virtuální svět. Vytvářeli bychom domy, auta, stromy, lidi a další objekty. Každý z těchto objektů by mohl být instancí třídy `Mnohotvar`.

Nevýhody samostatných tříd

Pro každý typ objektů bychom sice mohli vytvořit vlastní třídu, která by byla potomkem třídy `Mnohotvar`, ale pak bychom museli umět dopředu určit, které druhy objektů se budou v našem světě vyskytovat, a při vytváření nového světa bychom museli nově zaváděné objekty naprogramovat a celou aplikaci přeložit.

Způsob a výhody použití klonů u mnohotvaru

Kdybychom naproti tomu vytvořili pro každý druh objektu pouze jeden vzorový objekt, který bychom si někde zapamatovali a od nějž bychom v případě potřeby vytvořili pouze jeho klon, mohli bychom si zavádět nové typy objektů za chodu programu. Stačilo by vždy pouze vytvořit novou instanci *mnohotvaru*, specifikovat, jak bude složena z jednodušších tvarů, pojmenovat ji a pod zadaným jménem ji uložit do katalogu. Kdykoliv bys později chtěl daný druh objektu použít, požádal bys katalog o jeho klon a ten bys pak umístil do vytvářeného světa.

Klonování generující instance pseudotříd

398. Jestli jsem to dobře pochopil, tak návrhový vzor *Prototyp* ukazuje, jak i v překládaných jazycích vytvářet za chodu nové pseudotřídy objektů.

I tak by se to dalo říci. Použití návrhového vzoru *Prototyp* osvobozuje aplikaci od volání konstruktorů, které je natvrdo zakódováno v programu. Stačí, umíš-li nějakým způsobem získat výchozí instanci. Tu však můžeš získat i načtením ze souboru či ze sítě.

Vrátím se k předchozímu příkladu s virtuálním světem. Vytvoří-li kolega nějaký skvělý objekt, může ti jej poslat po síti či v souboru a ty jej načteš do své aplikace.

399. Existují ještě další situace, kdy je výhodné použít návrhový vzor *Prototyp*?

Další důvody pro použití *Prototypu*

GoF hovoří ještě o dvou důvodech k sáhnutí po prototypu:

- chceš-li se vyhnout budování třídní hierarchie továren odpovídající třídní hierarchii vytvářených instancí a
- mohou-li mít instance jen několik málo kombinací stavů, a je proto výhodnější si instance v jednotlivých stavech připravit a pak je klonovat, než je pokaždé znovu vytvářet za pomoci konstruktoru.

Do druhé kategorie patří částečně i náš příklad s *mnohotvarem*, protože je pro programátora mnohem jednodušší hotovou instanci naklonovat, než ji za pomoci konstruktoru a sestavovacích metod znovu vytvářet.

400. Jestli jsem to dobře pochopil, tak návrhový vzor *Prototyp* není v Javě pevně svázán s použitím metody `clone()`.

Vzor *Prototyp* není fixován na metodu `clone()`

Samozřejmě, že ne. Metoda `clone()` je nabídka systému, jak jednoduše vytvořit kopii instance bez použití konstruktoru. Klidně si ale můžeš definovat vlastní kopírovací metody, které budou konstruktor používat, nebo naopak metody, které sice budou vnitřně používat metodu `clone()`, avšak nebudou se s tím chlubit. Můžeš pak pro svůj kontejner definovat dvě kopírovací metody, které nazveš třeba `mělká()` a `hluboká()`, a necháš na uživatelích, kterou z nich v danou chvíli použijí.

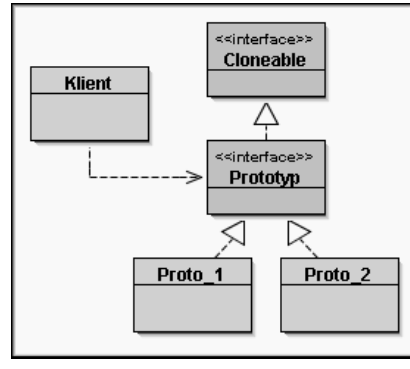
V Javě se k danému účelu nejčastěji využívá metoda `clone()`, ale např. právě výše zmiňovaná [41] doporučuje, aby sis před použitím této metody dobře rozmyslel, jestli by v tvém případě nebylo vhodnější použít pro kopírování objektů nějaké jiné řešení.

Implementace

401. Řekl bych, že jsem účel vzoru pochopil, takže můžeš začít vysvětlovat, jak jej mám implementovat.

Diagram tříd

Obecný princip návrhového vzoru *Prototyp* je znázorněn na obr. 23.1. Místo rozhraní `IPrototyp` v něm může vystupovat i abstraktní, nebo dokonce konkrétní třída. Důležité je, že tato třída definuje společné rozhraní tříd, jejichž instance chceme získávat klonováním nějakého prototypu.



Obrázek 23.1
Diagram tříd návrhového vzoru Prototyp

Třída označená jako `Klient` může představovat např. třídu, jejíž instance poskytují tovární metody na „výrobu“ instancí prototypu. Je to třída, která nějakým způsobem získá prototypové instance a vytváří jejich kopie.

Chceš-li v Javě využívat zabudovaného mechanismu klonování, měl bys navíc implementovat, resp. dědit, rozhraní `java.lang.Cloneable`. Při použití metody `clone()` je navíc vhodné myslet na to, abys uživatele klonovatelných objektů rozumným způsobem osvobodil od nutnosti zachytávání kontrolované výjimky `CloneNotSupportedException`.

402. Jak to myslíš? Když přece pro objekt definuji metodu, která tuto výjimku nevyhazuje, tak ji nikdo hlídat nemusí.

Správně. Jenomže o tom, že metoda nevyhazuje výjimku, se musí klient dozvědět. Jednou z možností je např. deklarovat ji v rozhraní `IPrototyp`. Pak už s ní bude překladač počítat a nebude tě nutit tancovat kolem jejího zachytávání.

403. Říkáš, že bych měl implementovat `Cloneable`. Znamená to, že když nechci, tak nemusím?

Záleží na tom, jak máš klonovací metodu definovanou. Budeš-li kdekoliv v hierarchii využívat metody `clone()` zděděné od třídy `Object`, tak by v opačném případě vyhodila výjimku. Obejdeš-li se bez ní a uchýlíš-li se k využití konstruktoru, rozhraní samozřejmě implementovat nemusíš.

404. Je nějaké pravidlo, podle nějž se dá odhadnout, kdy je v klonovací metodě výhodnější zkonstruovat nový objekt a kdy je výhodnější využít verzi zděděnou od třídy `Object` a volat `super.clone()`?

Kdy použít `clone()` a kdy konstruktor

Pravidlo je jednoduché: „Využij to, co dá méně práce.“ Obecně bychom mohli říci, že zděděná verze bývá výhodnější tehdy, vystačíš-li s mělkou kopií, a „konstruktorová“ verze tehdy, vyžaduješ-li hlubokou kopii, kterou je schopen dostatečně jednoduše vytvořit již konstruktor.

Zrovna v příkladu s mnohotvarem, o němž jsem se již několikrát zmiňoval, ale potřebujeme hlubokou kopii, a přitom použitím konstruktoru nic nezískáme. Konstruktor mnohotvaru totiž připraví pouze prázdný seznam, kdežto my potřebujeme vytvořit objekt s naplněným seznamem. Takže nám stejně nezbývá než explicitně kopírovat jednu položku po druhé. Pak už je ale použití metody `clone()` poměrně výhodné.

Příklad: Mnohotvar

405. O tom „mnohotvárném příkladu“ jsi již tolikrát mluvil, že si myslím, že je nejvyšší čas jej ukázat.

Dobrá, i když tento příklad má pro naše současné téma jednu nepříjemnou vlastnost: velký šum.

406. Šum?

Šum v příkladech

Šumem demonstračních programů označuji tu část kódu, která nedemonstruje přímo vysvětlovanou vlastnost (v našem případě použití návrhového vzoru *Prototyp*), ale bez níž program není funkční.

Vzory použité v příkladu

Na druhou stranu si říkám, že toho šumu zase nebude tolik, protože jsme o tomto příkladu hovořili již v kapitole o návrhovém vzoru *Strom*. Program tedy bude sloužit jako ukázka pro oba návrhové vzory a navíc také pro návrhový vzor *Adaptér*.

Než ti tu ale ukážu třídy související s mnohotvarem, chtěl bych ti prozradit, že v mnohotvaru je pro vytváření kopií instancí použita systémová metoda `clone()`. V závěrečném příkladu pro návrhový vzor *Interpret*, který je uveden v podkapitole *Příklad:*

Aritmetické výrazy na straně 488, najdeš program, který pro klonování využívá naopak volání konstruktorů.

Výpis 23.1: Definice rozhraní `IKlonovatelný`

```
package rup.česky.vzory._23_prototyp;

/*******************************************************************************
 * Rozhraní IKlonovatelný slouží k tomu,
 * aby třídy mohly hromadně deklarovat implementaci metody <code>clone</code>,
 * která je veřejná a nevyhazuje žádnou kontrolovanou výjimku,
 * a lze s ní proto v programu pracovat mnohem jednodušeji.
 */
public interface IKlonovatelný extends Cloneable
{
//== ZDĚDĚNÉ METODY ============================================================

    /***************************************************************************
     * Metoda vytvářející kopii objektu.
     *
     * @return Kopie objektu
     */
    public Object clone();
}
```

Výpis 23.2: Definice rozhraní `IKHýbací`

```
package rup.česky.vzory._23_prototyp;

import rup.česky.tvary.IHýbací;


/*******************************************************************************
 * Rozhraní IKHýbací slouží k deklaraci "hýbacích" instancí,
 * které můžeme jednoduše klonovat bez hlídání kontrolovaných výjimek.
 */
public interface IKHýbací extends IKlonovatelný, IHýbací
{
}
```

Výpis 23.3: Definice třídy `KAHýbací` sloužící jako adaptér pro instance potomků třídy `AHýbací`, které neimplementují rozhraní `IKHýbací`

```
package rup.česky.vzory._23_prototyp;

import rup.česky.tvary.AHýbací;
import rup.česky.tvary.IHýbací;
import rup.česky.tvary.Kreslítko;
import rup.česky.tvary.Oblast;


/*******************************************************************************
 * Třída <code>KAHýbací</code> slouží jako adaptér pro převod instancí třídy
 * <code>AHýbací</code>, které neimplementují rozhraní <code>KAHýbací</code>
 * na instance tohoto rozhraní.
```

```
*/
public class KAHýbací extends AHýbací implements IKHýbací
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Adaptovaná instance.
     *  Kvůli vytváření hluboké kopie při klonování nemůže být konstantní. */
    private AHýbací ah;

//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří KAHýbací obálku kolem svého parametru.
     */
    public KAHýbací( AHýbací ah )
    {
        super( ah.getX(), ah.getY(), ah.getŠířka(), ah.getVýška() );
        this.ah = ah;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Za pomoci dodaného kreslítka vykreslí obraz své instance
     * na animační plátno.
     *
     * @param kreslítko   Kreslítko, kterým se instance nakreslí na plátno
     */
    public void nakresli( Kreslítko k ) {
        ah.nakresli( k );
    }


    /*****************************************************************************
     * Nastaví novou pozici instance.
     *
     * @param x   Nová x-ová pozice instance
     * @param y   Nová y-ová pozice instance
     */
    public void setPozice( int x, int y )
    {
        super.setPozice( x, y );
        ah   .setPozice( x, y );
    }


    /*****************************************************************************
     * Nastaví nové rozměry instance.
     *
     * @param šířka    Nově nastavovaná šířka; šířka>0
     * @param výška    Nově nastavovaná výška; výška>0
     */
    public void setRozměr(int šířka, int výška)
    {
        super.setRozměr( šířka, výška );
        ah   .setRozměr( šířka, výška );
```

```
    }


    /** {@inheritdoc} */
    public KAHýbací clone() {
        //Prapředek APosuvný je sice Cloneable, ale teprve toto rozhraní
        //definuje metodu clone jako veřejnou
        KAHýbací ret = (KAHýbací) super.clone();
        ret.ah = (AHýbací) ah.clone();
        return ret;
    }
}
```

Výpis 23.4: Definice třídy `Mnohotvar`

```
package rup.česky.vzory._23_prototyp;

import java.util.ArrayList;
import java.util.List;

import rup.česky.tvary.AHýbací;
import rup.česky.tvary.IHýbací;
import rup.česky.tvary.Oblast;
import rup.česky.tvary.Pozice;

import static rup.česky.tvary.SprávcePlátna.SP;


/*******************************************************************************
 * Třída slouží k demonstraci práce se seznamy a návrhových vzorů
 * <i>Strom</i> a <i>Prototyp</i>.
 * Mnohotvar je postupně skládán z řady jednodušších tvarů,
 * které musí být instancemi rozhraní <code>IKHýbací</code>,
 * čímž deklarují svoji schopnost měnit pozici i velikost a klonovat se.
 * Při sestavování mnohotvar automaticky mění svoji pozici a rozměr tak,
 * aby pozice byla neustále v levém horním rohu opsaného obdélníka
 * a rozměr mnohotvaru odpovídal rozměru tohoto obdélníka.
 * Místo vkládaných objektů jsou používány jejich klony, aby dodatečnou
 * změnou velikosti či tvaru vloženého objektu nebyl mnohotvar narušen.
 *
 * @author    Rudolf Pecinovský
 * @version   0.00.000,  0.0.2003
 */
public class Mnohotvar extends AHýbací implements IKHýbací
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Seznam prvků, z nichž se mnohotvar skládá.
     *  Seznam nesmí být konstantní, aby mohl být u klonu vyměněn. */
    private List<Část> seznam = new ArrayList<Část>();

    /** Příznak ukončenosti tvorby mnohotvaru. */
    private boolean hotovo = false;

    /** Staré a nové souřadnice a rozměry mnohotvaru
     *  používané při přepočtech rozměrů a umístění jeho součástí
```

```
     *  při změně velikosti mnohotvaru. */
    private double smx, smy, smš, smv,
                   nmx, nmy, nmš, nmv;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /**************************************************************************
     * Vytvoří prázdný mnohotvar,
     * který převezme název generovaný rodičovskou třídou.
     */
    public Mnohotvar() {
        this( "" );
    }


    /**************************************************************************
     * Definuje prázdný mnohotvar se zadaným názvem.
     *
     * @param název  Název vytvářeného mnohotvaru.
     */
    public Mnohotvar(String název) {
        super( 0, 0, 0, 0 );
        if( název != "" )
            this.název = název;
    }


    /**************************************************************************
     * Vytvoří mnohotvar se zadaným názvem a prvky.
     *
     * @param název  Název vytvářeného mnohotvaru.
     * @param části  Jednotlivé části, z nichž bude mnohotvar sestaven,
     *               v pořadí odspodu nahoru
     */
    public Mnohotvar( String název, IKHýbací... části ) {
        this( název );
        for( IKHýbací ikh : části ) {
            zařaď( seznam.size(), ikh );
        }
    }


    /**************************************************************************
     * Vytvoří mnohotvar se zadaným názvem a prvky.
     *
     * @param název  Název vytvářeného mnohotvaru.
     * @param části  Jednotlivé části, z nichž bude mnohotvar sestaven,
     *               v pořadí odspodu nahoru
     */
    public Mnohotvar( String název, AHýbací... části ) {
        this( název );
        for( AHýbací část : části ) {
            zařaď( seznam.size(), new KAHýbací( část ) );
        }
    }
```

```
//== ABSTRAKTNÍ METODY ==========================================================
//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /****************************************************************************
     * Vytvoří kopii dané instance mnohotvaru,
     * kterou sestaví z kopií tvarů tvořících původní instanci.
     * Kopie bude mít schopnost se dále rozšiřovat (tj. přibírat další prvky)
     * bez ohledu na stav této schopnosti u originálu.
     */
    public Mnohotvar clone() {
        //Zděděná verze metody clone() slouží pouze k vytvoření objektu
        //s nastavenými hodnotami atributů zděděných od rodiče;
        //hodnoty ostatních atributů se vytvářené kopii nastaví až zde
        Mnohotvar mt = (Mnohotvar) super.clone();
        mt.hotovo = false;
        mt.seznam = new ArrayList<Část>();
        for( Část část : seznam )
            mt.seznam.add( část.clone() );
        return mt;
    }


    /****************************************************************************
     * Vrátí index ekvivalentu zadaného prvku v mnohotvaru,
     * tj. umístění tohoto ekvivalentu při kreslení mnohotvaru.
     * Pro prvek, jehož ekvivalent je zcela vespod, vrátí index 0,
     * pro prvek, který je zcela navrchu, vrátí index o jedničku menší,
     * než je počet prvků mnohotvaru.
     *
     * @param prvek  Prvek, jehož index hledáme
     * @return    Index zadaného prvku v mnohotvaru.
     *            Není-li prvek součástí mnohotvaru, vrátí -1.
     */
    public int index( IHýbací prvek ) {
        int kde = 0;
        for( Část c : seznam ) {
            if( c.původní == prvek )
                return kde;     //==========>
            kde++;
        }
        return -1;
    }


    /****************************************************************************
     * Nakreslí mnohotvar pomocí dodaného kreslítka.
     *
     * @param kreslítko   Kreslítko dodané správcem plátna
     */
    public void nakresli(rup.česky.tvary.Kreslítko kreslítko)  {
        for( Část část : seznam ) {
            část.ikh.nakresli( kreslítko );
        }
    }


    /****************************************************************************
```

```
     * Přidá do mnohotvaru zadaný prvek a příslušně upraví novou
     * pozici a velikost mnohotvaru. Touto metodou nelze přidat prvek,
     * který byl již dříve do mnohotvaru přidán.
     *
     * @param ikh  Přidávaný hýbací tvar
     */
    public void přidej( IKHýbací ikh ) {
        if( index( ikh ) >= 0 )
            throw new IllegalArgumentException( "V obrazci " + this.název +
                " je přidávaný prvek již zahrnut.\n   Přidáváno: " + ikh);
        zařaď( seznam.size(), ikh );
    }


    /*************************************************************************
     * Přidá do mnohotvaru zadanou skupinu prvků.
     * Jednotlivé prvky skupiny jsou do mnohotvaru přidávány v cyklu
     * pomocí opakovaného volání metody {@link #přidej(IKHýbací)}.
     *
     * @param ihh  Přidávané prvky
     */
    public void přidej( IKHýbací... ihh ) {
        for( IKHýbací ikh : ihh )
           přidej( ikh );
    }


    /*************************************************************************
     * Přidá do mnohotvaru zadaný prvek těsně pod označený prvek
     * a příslušně upraví novou pozici a velikost mnohotvaru.
     *
     * @param starý  Tvar, který je součástí mnohotvaru
     *               a pod nějž se nový prvek přidává
     * @param nový   Přidávaný hýbací tvar
     * @return <code>true</code> pokud prvek již byl součástí mnohotvaru
     */
    public boolean přidejPod( IKHýbací starý, IKHýbací nový ) {
        ZdaKde zk = připrav( starý, nový );
        zařaď( zk.kde, nový );
        return zk.zda;
    }


    /*************************************************************************
     * Přidá do mnohotvaru zadaný prvek těsně nad označený prvek
     * a příslušně upraví novou pozici a velikost mnohotvaru.
     *
     * @param starý  Tvar, který je součástí mnohotvaru
     *               a nad nějž se nový prvek přidává
     * @param nový   Přidávaný hýbací tvar
     * @return <code>true</code> pokud prvek již byl součástí mnohotvaru
     */
    public boolean přidejNad( IKHýbací starý, IKHýbací nový ) {
        ZdaKde zk = připrav( starý, nový );
        zařaď( zk.kde+1, nový );
        return zk.zda;
    }
```

```
/*****************************************************************************
 * Přidá do mnohotvaru zadaný prvek na jeho "dno"
 * a příslušně upraví novou pozici a velikost mnohotvaru.
 *
 * @param nový   Přidávaný hýbací tvar
 * @return <code>true</code> pokud prvek již byl součástí mnohotvaru
 */
public boolean přidejDolů( IKHýbací nový ) {
    ZdaKde zk = připrav( null, nový );
    zařaď( 0, nový );
    return zk.zda;
}


/*****************************************************************************
 * Přidá do mnohotvaru zadaný prvek na jeho vrchol
 * a příslušně upraví novou pozici a velikost mnohotvaru.
 *
 * @param nový   Přidávaný hýbací tvar
 * @return <code>true</code> pokud prvek již byl součástí mnohotvaru
 */
public boolean přidejNahoru( IKHýbací nový ) {
    ZdaKde zk = připrav( null, nový );
    zařaď( seznam.size(), nový );
    return zk.zda;
}


/*****************************************************************************
 * Oznamuje dokončení prací na sestavování instance.
 * Od tohoto okamžiku již nebude možno přidat žádný další tvar.
 */
public void sestaveno() {
    hotovo = true;
}


/*****************************************************************************
 * Přemístí celý mnohotvar na zadanou pozici.
 * Všechny součásti instance se přemisťují jako celek
 *
 * @param x  Nastavovaná vodorovná souřadnice
 * @param y  Nastavovaná svislá souřadnice
 */
public void setPozice( int x, int y )  {
    int dx = x - xPos;
    int dy = y - yPos;
    SP.nekresli();
        for( Část č : seznam ) {
            IKHýbací ikh = č.ikh;
            Pozice  pos = ikh.getPozice();
            ikh.setPozice( pos.x + dx, pos.y + dy );
        }
        super.setPozice( x, y );
    SP.vraťKresli();
}
```

```
    /*************************************************************************
     * Nastaví nový rozměr mnohotvaru. Upraví rozměry a pozice všech jeho
     * součástí tak, aby výsledný mnohotvar měl i při novém rozměru stejný vzhled.
     *
     * @param šířka  Nastavovaná šířka
     * @param výška  Nastavovaná výška
     */
    public void setRozměr( int šířka, int výška ) {
        připravKonstanty( xPos, yPos, this.šířka, this.výška,
                          xPos, yPos, šířka,      výška );
        //Uprav velikosti a pozice jednotlivých částí
        SP.nekresli();
            for( Část č : seznam )
                č.poNafouknutí();
            super.setRozměr( šířka, výška );
        SP.vraťKresli();
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /*************************************************************************
     * Přidá do seznamu prvků v mnohotvaru zadaný prvek těsně za zadaný prvek.
     *
     * @param starý  Tvar, který je součástí mnohotvaru
     *               a nad nějž se nový prvek přidává
     * @param nový   Přidávaný hýbací tvar
     */
    private ZdaKde připrav( IKHýbací starý, IKHýbací nový ) {
        ZdaKde zk = new ZdaKde();

        int in = index( nový );
        if( in >= 0 ) {     //Ukládaný tvar je již součástí mnohotvaru
            seznam.remove( in );
            zk.zda = false;
        } else              //Není ještě součástí
            zk.zda = true;

        if( starý == null ) {   //Pozice nového bude zadána absolutně
            return zk;          //==========>
        }

        //Pozice nového bude zadána relativně vůči starému
        int is = index( starý );
        if( is < 0 ) {
            throw new IllegalArgumentException(
                "\nReferenční objekt " + starý + " není součástí mnohotvaru!" );
        }
        zk.kde = is;    //Budeme vracet i pozici referenčního tvaru
        return zk;
    }


    /*************************************************************************
     * Připravuje podklady pro následnou aktualizaci uchovávaných poměrů
     * klíčových rozměrů jednotlivých součástí mnohotvaru.
     * Tyto konstanty jsou pro všechny součásti mnohotvaru společné.
```

```
     *
     * @param sx  Původní (staré) x.
     * @param sy  Původní (staré) y.
     * @param sš  Původní (stará) šířka.
     * @param sv  Původní (stará) výška.
     * @param nx  Nové x.
     * @param ny  Nové y.
     * @param nš  Nová šířka.
     * @param nv  Nová výška.
     */
    private void připravKonstanty( int sx, int sy, int sš, int sv,
                                   int nx, int ny, int nš, int nv )
    {
        smx = sx;   smy = sy;   smš = sš;   smv = sv;
        nmx = nx;   nmy = ny;   nmš = nš;   nmv = nv;
    }


    /*************************************************************************
     * Přidá do mnohotvaru zadaný prvek na zadané místo v seznamu
     * a příslušně upraví novou pozici a velikost mnohotvaru.
     *
     * @param index  Pozice v seznamu, kam bude nový prvek zařazen.
     *               Původní prvek na této pozici a následující prvky
     *               budou posunuty o jednu pozici dále.
     * @param ikh    Přidávaný hýbací tvar
     */
    private void zařaď( int index, IKHýbací ikh ) {
        if( hotovo )
            throw new IllegalStateException(
                "\nPokus o přidání nové části do objektu, " +
                "jehož tvorba byla již ukončena" );
        Oblast o = ikh.getOblast();
        if( seznam.isEmpty() )  {       //Přidáváme první tvar
            xPos  = o.x;
            yPos  = o.y;
            šířka = o.šířka;
            výška = o.výška;
            připravKonstanty( 0, 0, 0, 0, xPos, yPos, šířka, výška );
        } else {                        //Přidáváme další tvar
            int mx = xPos;  //Zapamatujeme si staré parametry mnohotvaru
            int my = yPos;
            int mš = šířka;
            int mv = výška;

            int ihx = o.x;  //Parametry přidávaného tvaru
            int ihy = o.y;
            int ihš = o.šířka;
            int ihv = o.výška;

            //Upravujeme parametry podle pozice a velikosti přidávaného tvaru
            if( ihx < xPos ) {
                šířka += xPos - ihx;
                xPos = ihx;
            }
            if( ihy < yPos ) {
                výška += yPos - ihy;
                yPos = ihy;
```

```
            }
            if( (xPos + šířka) < (ihx + ihš) )
                šířka = ihx + ihš - xPos;
            if( (yPos + výška) < (ihy + ihv) )
                výška = ihy + ihv - yPos;

            připravKonstanty( mx, my, mš, mv, xPos, yPos, šířka, výška );
            for( Část č : seznam )
                č.poPřidání();
        }
        SP.nekresli();
            SP.odstraň( ikh );  //Pro případ, že by byl před tím kreslen
            seznam.add( index, new Část( ikh ) );
        SP.vraťKresli();
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ==================================================

    /**************************************************************************
     * Třída <code>Část</code> slouží jako přepravka pro uchovávání
     * pomocných informací pro správnou změnu velikosti mnohotvaru.
     */
    private final class Část implements IKlonovatelný
    {
    //== PROMĚNNÉ ATRIBUTY INSTANCÍ ============================================

        /** Odkaz na část mnohotvaru. */
        IKHýbací ikh;

        /** Odkaz na prvek, který byl předlohou dané části mnohotvaru. */
        IKHýbací původní;

        /** Podíl odstupu od levého kraje mnohotvaru na jeho celkové šířce. */
        double dx;

        /** Podíl odstupu od horního kraje mnohotvaru na jeho celkové výšce. */
        double dy;

        /** Podíl šířky části k celkové šířce mnohotvaru. */
        double dš;

        /** Podíl výšky části k celkové výšce mnohotvaru. */
        double dv;


    //== KONSTRUKTORY A TOVÁRNÍ METODY ==========================================

        /**********************************************************************
         * Vytvoří přepravku a zapamatuje si aktuální stav některých poměrů
         * vůči současné porobě mnohotvaru.
         */
        Část( IKHýbací ikh )
        {
            //Uložíme-li místo obdrženého prvku jeho klon,
            //nejsme závislí na budoucích manipulacích s prvkem
            this.ikh      = (IKHýbací)ikh.clone();
            this.původní  = ikh;
```

```
        Oblast o = ikh.getOblast();
        dx = (o.x - nmx) / nmš;
        dy = (o.y - nmy) / nmv;
        dš = o.šířka / nmš;
        dv = o.výška / nmv;
    }


//== NESOUKROMÉ METODY INSTANCÍ ============================================

    /**************************************************************************
     * Aktualizuje uchovávané poměry klíčových rozměrů a vykreslí daný tvar
     * v aktuální pozici a rozměru.
     */
    public Část clone()
    {
        Část klon;
        try {
            klon = (Část) super.clone();
        }catch( CloneNotSupportedException e ) {
            //Rodičem je Object => musíme zachytávat výjimku, i když by k ní
            //v praxi nemělo dojít. EJejí explicitní ošetření je zde
            //definováno pro případ, že by časem někdo program upravoval
            throw new RuntimeException( "\nRodičovská třída " +
                getClass().getSuperclass().getName() +
                " nepodporuje klonování." );
        }
        //Děláme hlubokou kopii, a proto potřebujeme klonovat
        //i odkazovaný hýbací prvek
        klon.ikh = (IKHýbací)ikh.clone();
        return klon;
    }


    /**************************************************************************
     * Aktualizuje uchovávané poměry rozměrů tvaru vůči mnohotvaru.
     * Očekává, že metoda připrav nastavila hodnoty potřebných konstant.
     */
    void poPřidání() {
        //Protože se mnohotvar může jen zvětšit (nelze odebrat jeho část),
        //mohou se relativní souřadnice jeho částí pouze zmenšovat,
        //tj. vždy platí smx >= nmx
        dx = (smx - nmx + dx*smš) / nmš;
        dy = (smy - nmy + dy*smv) / nmv;

        dš = dš * smš / nmš;
        dv = dv * smv / nmv;
    }


    /**************************************************************************
     * Aktualizuje uchovávané poměry klíčových rozměrů a vykreslí daný tvar
     * v aktuální pozici a rozměru.
     */
    void poNafouknutí() {
        ikh.setPozice( (int)Math.round( nmx + dx*nmš ),
                       (int)Math.round( nmy + dy*nmv )  );
        ikh.setRozměr( (int)Math.round( nmš*dš ),
```

```
                            (int)Math.round( nmv*dv ) );
        }
    }


    /*****************************************************************************
     * Třída <code>ZdaKde</code> slouží jako přepravka pro předávání informací
     * o tom, kam se má daný prvek zařadit a zda ještě v mnohotvaru není.
     */
    private static final class ZdaKde {
        boolean zda;
        int kde;
    }

//== TESTY A METODA MAIN ======================================================
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Prototyp* ukazuje vedle volání konstruktoru či cizí tovární metody další alternativu získávání instancí prostřednictvím továrních metod vracejících kopii instance, které byla zpráva zaslána.
- Pro uvedený účel je ve standardní knihovně připravena metoda `clone()`, která je definována ve třídě `Object`.
- Pro instanci jsou přístupné pouze chráněné atributy jejích rodičů, avšak již ne chráněné atributy těchto atributů, a to nezávisle na jejich typu.
- Metoda `clone()` je definována jako chráněná, takže ji mohou volat pouze „nadobjekty" daného objektu.
- Zděděná verze metody `clone()` pracuje pouze pro instance tříd implementujících rozhraní `java.lang.Cloneable`. Pro ostatní instance vyhazuje výjimku `CloneNotSupportedException`.
- Metoda `clone()` třídy `Object` vytvoří v paměti kopii celého objektu, jehož je její instance kořenovým podobjektem.
- Potřebujeme-li metodu překrýt, bývá její propojenost s virtuálním strojem využívána voláním `super.clone()`.
- Metodu `clone()` můžeme definovat i bez využití rodičovské verze použitím konstruktoru.
- Při klonování rozlišujeme mělké a hluboké kopie.
  - Při mělkém kopírování se kopíruje pouze vlastní objekt, avšak ponechávají se odkazy na objekty, na něž se ve svých atributech odkazuje původní objekt. Originál i kopie se proto odkazují na stejné objekty.
  - Při hlubokém kopírování se kopíruje nejenom vlastní objekt, ale hluboce se kopírují také všechny objekty, na které se tento objekt odkazuje. Vzniká tak nový, naprosto nezávislý objekt.
- Aby nebylo v programech třeba ošetřovat možný výskyt kontrolované výjimky `CloneNotSupportedException`, lze po objektech požadovat, aby implemento-

valy rozhraní, v němž je metoda `clone()` deklarována jako metoda nevyhazující žádnou kontrolovanou výjimku.

- Používáme-li instance knihovních tříd, které takovéto rozhraní neimplementují, můžeme je přizpůsobit našim požadavkům prostřednictvím adaptéru.
- O problémech s klonováním se podrobněji rozepisuje např. kniha [41], resp. její český překlad [28].
- Klonování umožňuje, aby aplikace byla schopna vytvářet instance i od tříd, které ještě nezná.
- Vedle tříd umožňuje klonování definovat i pseudotřídy tvořené skupinami instancí stejné třídy.
- Návrhový vzor *Prototyp* není fixován na metodu `clone()`. Místo ní můžeme pro vytváření kopií definovat i vlastní metodu.
- Obecně bývá využití metody `clone()` výhodnější při používání mělkých kopií a využití speciální tovární metody volající konstruktor při používání hlubokých kopií. Nemusí tomu tak ale být vždy.
- Návrhový vzor *Prototyp* patří mezi vzory uvedené v GoF.

KAPITOLA 24

Dosazujeme do vzorečku (Stavitel – Builder)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Odděluje konstrukci složitého objektu (tj. postup jeho tvorby) od jeho vnitřní reprezentace. Tím umožňuje využít stejného konstrukčního postupu pro různé vnitřní reprezentace konstruovaných objektů.

[1] **Definice v GoF:** Separate the construction of a complex object from its representation so that the same construction process can create different representations. – Odděluje konstrukce složitých objektů od jejich reprezentace, takže pak může stejný konstrukční proces vytvářet různé reprezentace.

Účel

407. Musím přiznat, že ze stručné charakteristiky vzoru opět nejsem příliš moudrý.

Příklad: Stavba domu

Zkusím ti to přiblížit na příkladu ze života. Stavíš-li dům, musíš vždy nejprve vybudovat základy, pak vytvořit sklepy, poté přízemí, pak jednotlivá patra a nakonec střechu. Tento postup je stejný nezávisle na tom, stavíš-li dům z cihel, z panelů anebo ze dřeva. Svým způsobem by se dal aplikovat i na domy budované z nějaké dětské stavebnice – třeba z Lega.

408. U domků z Lega přece žádné základy ani sklepy nestavím.

No a? Tak bude příslušná metoda prázdná. S tím jsme se již přece setkali mnohokrát.

409. No dobře. To mne hned nenapadlo. Takže chceš říct, že v tomto návrhovém vzoru existuje nějaký stavitel, který ví, jak se staví obecný dům, a podle toho, jaký materiál tomuto staviteli dodáš, takový dům postaví?

Princip: S trochou představivosti bychom to tak mohli popsat. Jenom bych maličko opravil tvoji terminologii.

– Stavitel = výkonný objekt

Stavitelem bych nazval objekt, který umí postavit jednotlivé části domu ze svého oblíbeného materiálu. Jeden je specialista na domy z cihel, druhý na domy ze dřeva, další na domy z Lega. Všichni stavitelé budou implementovat rozhraní specifikující, co má umět každý stavitel postavit.

– Stavbyvedoucí = řídicí objekt

Stavitel je však pouze takový dělník – říkejme mu třeba *výkonný objekt*. Při vytváření objektu má ale hlavní slovo *řídicí objekt*, který bychom v našem příkladu mohli označit za stavbyvedoucího (GoF mu říká *director*), který ví, jaký dům se má postavit. Dostane přiděleného stavitele a bude mu postupně přidělovat úkoly, co má vybudovat: nejprve základy, pak sklep, přízemí, jednotlivá patra a nakonec celý objekt zastřešit.

Dostane-li stavbyvedoucí za úkol postavit pětipatrový dům se zadaným stavitelem, vyvolá postupně jednotlivé akce a výsledkem by měl být požadovaný dům. Příště mu dodáš jiného stavitele a on postaví jiný dům.

410. Princip vzoru již asi chápu, ale přivítal bych ještě nějaký programátorštější příklad.

Příklad z GoF: převodník textu

Líbil se mi příklad z GoF, kde ukazují použití tohoto vzoru na konvertování textových souborů z formátu RTF do některého jiného formátu. Oproti příkladům z jiných učebnic a příruček u něj není dopředu jasné, jak bude vytvářený dokument vypadat – to se pozná až při analýze převáděného dokumentu.

411. To vypadá zajímavě. Zkus mi jej popsat trochu podrobněji.

Upravený příklad: Jestli dovolíš, popsal bych trochu upravenou úlohu. V GoF je jako příklad zmíněný nějaký jinde použitý program. Vlastní program ale není uveden, protože by byl pro demonstrační účely zbytečně složitý. My si proto příklad zjednodušíme, abychom jej pak mohli naprogramovat a vše si na něm vyzkoušet.

Představ si, že náš program generuje nějaké texty, které jsou podle situace požadovány v různých formátech. K některým účelům postačuje jednoduchý textový dokument, pro jiné účely potřebuješ připravit dokument ve formátu HTML, XML, RTF či PDF.

Protože termín formát by byl v dalším textu používán ve dvou významech, jednou jako formát souboru (např. HTML) a podruhé jako formát textu (např. tučné písmo), budu po zbytek této kapitoly používat místo termínu *formát souboru* termín *tvar souboru*. Budeme proto mít za úkol vytvořit formátovaný soubor v požadovaném tvaru.

– Úkoly výkonného objektu

V každém z výstupních tvarů se znaky s kódy většími než 127 zapisují jinak, každý z nich uchovává jinak odstavce a jinak označuje zvýraznění částí textu. (Víc možností zatím uvažovat nebudeme, aby se nám program příliš nerozrostl.)

– Klient: autor

V programu můžeš mít generátor textů (to bude náš klient), který bude mít na starosti obsah výsledného dokumentu a kterého budu v našem programu označovat termínem *autor*.

– Řídicí objekt: sazeč

Autor sice texty vytvoří, ale neumí je převést do nějaké publikovatelné podoby. To bude mít na starosti *sazeč*, který bude naším řídícím objektem. Sazeč zařídí, aby ve výsledném dokumentu byly odstavce skutečně odstavci, tučný text byl vysazen tučně atd.

– Výkonný objekt: sázecí stroj

Podle toho, v jakém výstupním tvaru bude objekt požadován (HTML, RTF, PDF, …), bude mít sazeč k dispozici příslušný *sázecí stroj* (výkonný objekt), který bude převádět zadané texty a formátovací požadavky do požadovaného výsledného tvaru.

Autor se nebude starat o konkrétní podobu výstupu, ale pouze bude vědět, jak je třeba rozčlenit text do odstavců a které části textu je třeba zvýraznit. Tuto informaci nějak předá sazeči a ten pak ve spolupráci s příslušným sázecím strojem vysadí zadaný text v požadovaném tvaru.

Až bude autor potřebovat vysadit nějaký text, požádá sazeče, kterému dá k dispozici příslušný sázecí stroj. Pak už jej pouze zásobuje texty, které sazeč za pomoci dodaného stroje sází.

***Stavitel* odpoutává řídicí a výkonné objekty**

412. Řekl bych, že mi to začíná být jasné. Jestli to dobře chápu, tak cílem vzoru *Stavitel* je umožnit operativně definovat nové výkonné objekty, aniž bychom museli měnit kód řídícího objektu.

Přesně tak. Návrhový vzor *Stavitel* odpoutává třídu řídících objektů (to byl např. náš stavbyvedoucí či sazeč) od tříd výkonných objektů (to byli naši stavitelé a sázecí stroje) a umožňuje tak jejich nezávislé změny.

Vzor *Stavitel* se nesoustřeďuje na dodání hotových objektů, ale na proces jejich tvorby

Při té příležitosti bych tě ještě chtěl upozornit na jednu zvláštnost tohoto návrhového vzoru. Na rozdíl od ostatních návrhových vzorů zodpovědných za vytváření nových objektů se tento vzor nezabývá jednorázovým vytvořením a „dodáním“ objektu, jak tomu bylo v případě tovární metody či klonu, ale zabývá se vlastním procesem vytváření požadovaného objektu.

Důsledkem předchozí skutečnosti je, že návrhový vzor *Stavitel* umožňuje mnohem citlivější ovlivňování konstrukčního procesu.

Kdy vzor použít:

– Algoritmus tvorby nezávislý na vnitřní reprezentaci

413. Můžeš tedy shrnout, kdy je vhodné tento návrhový vzor použít?

Použijeme jej v situacích, kdy:

- Algoritmus tvorby požadovaných objektů je nezávislý na jejich vnitřní reprezentaci, tj. když způsob tvorby objektu nezávisí na tom, z jakých konkrétních stavebních kamenů tento objekt vytváříme.

– Objekty se budují v několika fázích

- Vytvářené objekty se budují postupně v několika fázích. Řídící objekt pak funguje jako takový supervizor, který dohlíží na správnou posloupnost vyvolání jednotlivých fází výstavby objektu. Iniciuje a často i validuje jednotlivé fáze a v případě potřeby hlásí vzniklé chyby. S tím se setkáme zejména tehdy, mají-li vytvářené objekty složitou strukturu.

– Řídící objekt jako správce zdrojů

- Při vytváření objektů jsou používány systémové zdroje, kterých je jen omezené množství. Řídící objekt pak může fungovat jako centrální správce těchto zdrojů.

Výsledkem je většinou složitě strukturovaný objekt

V našem příkladu s tvorbou formátovaného textu vysazeného v nějakém tvaru je výsledkem jakýsi zdánlivě homogenní objekt (textový řetězec). Většinou to ale bývá spíš naopak; výsledkem práce stavbyvedoucího a jeho stavitele bývá nějaký složitě strukturovaný objekt. Proto se také návrhový vzor *Stavitel* používá často ve spojení s návrhovým vzorem *Strom*.

Implementace

414. Řekl bych, že z předchozího výkladu účelu vyplynula i potřebná implementace.

Zopakování principů

Pro jistotu bych hlavní principy tohoto návrhového vzoru ještě jednou zopakoval. O vytváření objektu se při aplikaci tohoto návrhového vzoru dělí dva objekty:

– Řídící objekt

– Výkonný objekt

- vlastní vytváření objektů má na starosti řídící objekt,
- vytváření jednotlivých částí má na starosti výkonný objekt.

Řídící objekt ovládá výkonný objekt

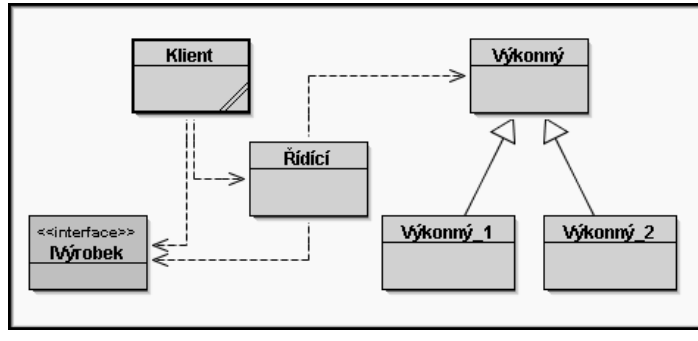
Řídící objekt dostane přidělen výkonný objekt, s jehož pomocí požadovaný objekt vytvoří. Postupně výkonnému objektu říká, jakou část vytvářeného objektu má vytvořit, a ten ji vytvoří.

Výkonných objektů bývá několik

Výkonných objektů bývá více. Každý z nich vytváří požadované části jinak. Všechny výkonné objekty však implementují stejné rozhraní, v němž jsou deklarovány metody, které může řídící objekt volat.

Výkonné objekty mají společného rodiče či rozhraní

Společné rozhraní výkonných objektů nemusí být nutně definováno jako `interface`. Může to být i abstraktní či konkrétní třída (tak je to i na obr. 24.1), která bude definovat implicitní verze metod, aby se třídy výkonných objektů mohly omezit pouze na definice těch metod, které chtějí překrýt.



Obrázek 24.1
Diagram tříd návrhového vzoru Stavitel

415. Kdysi jsem někde diagram tříd stavitele zahlédl a nebyla v něm vůbec třída zastupující výrobek, ale zato v něm byly třídy zastupující produkty jednotlivých výkonných objektů.

Proč se občas kreslí vzor v diagramu jinak

Jak jsem již řekl v odpovědi na otázku 412, těžištěm zájmu návrhového vzoru *Stavitel* není vlastní vytvářený objekt, ale proces jeho tvorby. Někteří autoři (včetně GoF) tuto skutečnost zdůrazňují tím, že vytvářený objekt, resp. jeho rozhraní, do diagramu tříd vůbec nezahrnou.

Já se však domnívám, že všechny ty tanečky se dělají právě proto, abychom nakonec vytvořili požadovaný objekt, a proto by tento objekt měl mít své zastoupení i v diagramu tříd návrhového vzoru.

Na druhou stranu ale musím přiznat, že v mnoha případech jsou jednotlivé druhy vytvářených objektů natolik rozdílné, že se těžko hledá jakési společné rozhraní. Pokud bys měl toto rozhraní definovat pouze formálně, tak pak je lepší je nedefinovat vůbec. Klient ví, co si objednal, a lze tedy předpokládat, že bude také vědět, jak s tím zacházet.

416. Ještě jsi mi nevysvětlil, proč ve svém diagramu nemáš produkty jednotlivých výkonných objektů.

Výkonné objekty mohou být schopny vytvořit celý objekt, při jehož sestavování jim bude řídící objekt radit, jak postupovat. Na druhou stranu mohou být schopny vytvořit pouze části výsledného objektu a ponechat jeho sestavení na řídícím objektu.

Příklad z GoF

V GoF je např. ukázka programu vytvářejícího bludiště. Výkonné objekty jsou schopny vytvářet různé druhy bludišť a řídící objekt jim říká, které místnosti mají vytvořit a jak je propojit. Podle mne by se na podstatě návrhového vzoru nic nezměnilo, kdyby výkonné objekty nebyly schopny vrátit celé bludiště, ale pouze poskytovaly řídícímu objektu jednotlivé místnosti a propojovaly je. Celé bludiště by pak vrátil klientovi řídící objekt.

Vzory nejsou dogma

Vím, že existuje řada puristů, kteří neuznávají jiné návrhové vzory než přesně ty, které byly publikovány v GoF. Doporučoval bych ti ale, abys nebral návrhové vzory jako dogma a spíš se z nich snažil pochopit duch jejich konstrukce, protože pak budeš moci využít jejich výhod i v případě, kdy požadavky a okrajové podmínky tvé aplikace nebudou přesně odpovídat požadavkům a okrajovým podmínkám vzorů publikovaných v GoF.

417. Přivítal bych, kdybys ještě jednou shrnul, co vše se může při použití tohoto návrhového vzoru měnit a jak se s tím vypořádat.

Dobrá, a aby to bylo jasnější, zkusím to hned demonstrovat na výše zmíněném příkladu se sazečem textů.

Co se může měnit

Při použití návrhového vzoru *Stavitel* jsi připraven poměrně jednoduše reagovat na potřebu volby následujících parametrů:

– Části a vlastnosti vytvářených objektů

- Z jakých částí se budou vytvářené objekty skládat a jaké vlastnosti budou tyto části či celé objekty mít (dům může mít volitelný počet pater).

V uvedeném příkladu sem patří to, jaké formátovací možnosti chceš svým programem podporovat, tj. zda budeš umět vedle tučnosti či ležatosti textu měnit např. i velikost a druh písma, zarovnání odstavců a další formátovací charakteristiky.

– Způsob zadání požadavků na vytvářený objekt

- Způsob, jakým zadáváš a interně uchováváš popis požadavků na vytvářený objekt (např. jak zadáš, že chceš, aby vytvářený dům byl pětipatrový).

  V uvedeném příkladu bychom sem zahrnuli požadované formátování textu. Musíš umět nějakým způsobem zadat, které části textu budou vysazeny tučně či kurzivou, jak se pozná hranice odstavců a další potřebné charakteristiky.

– Druh vytvářeného objektu

- Požadovaný druh vytvářeného objektu (např. jestli má být dům z cihel, panelů či ze dřeva).

  V uvedeném příkladu sem patří požadovaný druh výsledného dokumentu, tj. budeš-li chtít, aby byl výsledek ve formátu TXT, HTML, RTF či nějakém dalším.

První charakteristika, tj. možnosti vytvářeného objektu (u nás množina formátovacích možností), ovlivní definici společného rozhraní definující schopnosti stavitelů.

Druhá charakteristika, tj. způsob zadávání požadavků (u nás způsob zadání formátování textu), ovlivní množinu metod, které bude muset umět řídící objekt (v našem případě sazeč) nabídnout.

Poslední charakteristika, tj. druh vytvářeného objektu (u nás druh vytvářeného dokumentu), pak ovlivní množinu výkonných objektů (v našem případě množinu sázecích strojů), které budeme muset umět řídícímu objektu poskytnout.

Příklad

418. O tom sázecím programu jsi toho již tolik napovídal, že je nejvyšší čas ukázat, jak jsi jej naprogramoval.

Základní idea Dobrá. Ukážu ti příklad generátoru jednoduše formátovaných textů, o němž jsem před chvílí hovořil. Připomínám, že autor (klient), který bude vytvářet své texty, bude tyto texty dělit do odstavců a bude označovat, které části textu mají být vysazeny tučně a které kurzivou. Víc formátování mu pro zjednodušení nepovolíme. (Pro maximální zjednodušení dokonce nedovolíme ani současné nastavení tučného i ležatého písma – to si můžeš v případě zájmu doplnit sám.)

Práce programu Takto předpřipravené texty předá autor sazeči. Konstruktoru sazeče bude jako parametr předán odkaz na sázecí stroj, pomocí nějž bude vytvářený sazeč schopen nechat celý text vysadit do příslušného výstupního tvaru.

Sazeč × autor Sazeč musí být dohodnut s autorem, jakým způsobem mu autor označí, co je odstavec a které části textu má nechat vysadit tučně nebo kurzivou. Může být ale s různými autory dohodnut na různých způsobech předávání informací. Pro každou reprezentaci musí definovat vlastní metodu, která tuto reprezentaci sázeného textu přijme a zpracuje.

Způsoby zadávání textu

419. Koncepce je jasná, teď mne zajímá spíš ta implementace.

Dva způsoby reprezentace textu

Říkal jsem, že řídící objekt může být se svými klienty „dohodnut“ na různých způsobech předávání jejich požadavků. Náš sazeč bude autorům nabízet dva způsoby zadání formátovaného textu: objektový a textový.

Objektová reprezentace – typ `Text`

V objektové reprezentaci bude každá část textu reprezentována jako instance neměnné hodnotové třídy `Text` (viz výpis 24.1). Tato instance si bude kromě vlastního textu pamatovat i způsob formátování svého textu. Veškerý text příslušné dané instance bude mít jednotný způsob formátování.

Typ formátování textu a vlastní text jsou definovány jako vlastnosti instance, na které se mohou ostatní objekty kdykoliv zeptat, ale které již není možno po vytvoření instance měnit (říkal jsem, že třída je neměnná).

Typ formátování

Způsob formátování jsem definoval jako výčtový typ, který je zanořený do třídy `Text`. Tento výčtový typ je ale definován jako veřejný, a proto jej mohou bez problému využívat i instance ostatních tříd.

Výpis 24.1: Definice tříd `Text` a `Text.Typ`

```
package rup.česky.vzory._24_stavitel.text;

/*******************************************************************************
 * Instance třídy <code>Text</code> jsou části textu s homogenním formátováním.
 * Své formátování si části pamatují.
 */
public class Text
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final Typ    typ;   //Formátování dané části textu
    private final String text;  //Vlastní text


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*******************************************************************************
     * Vytvoří novou část se zadaným textem a formátem.
     */
    public Text( Typ typ, String text )  {
        this.typ  = typ;
        this.text = text;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Vrátí text dané části.
     */
    public String getText()  {
        return text;
```

```
    }


    /*************************************************************************
     * Vrátí formát dané části.
     */
    public Typ getTyp()  {
        return typ;
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /**  Instance třídy <code>Typ</code> deklarují možné druhy textu. */
    public enum Typ { BĚŽNÝ, TUČNÝ, ŠIKMÝ };

}
```

Odstavce

Části textu uchovávané v instancích třídy `Text` vytvářejí odstavce, tj. instance třídy `Odstavec`. Instance třídy `Odstavec` jsou iterovatelné. Jejich iterátor vrací postupně jednotlivé části textu (tj. instance třídy `Text`) tvořící daný odstavec.

Použití iterovatelného pole

Ve zdrojovém kódu třídy `Odstavec` (viz výpis 24.2) si všimni, že zodpovědnost za správné vytvoření iterátoru přehazuje `Odstavec` spolu se zodpovědností za jeho správnou funkci na instanci třídy `IterovatelnéPole` (její definici najdeš ve výpisu 16.1 na straně 206), o které jsme si povídali v souvislosti s návrhovým vzorem *Iterátor* (viz kapitolu *Moc se mi v tom nehrab (Iterátor – Iterator)* na straně 203).

Výpis 24.2: Definice třídy `Odstavec`

```
package rup.česky.vzory._24_stavitel.text;

import java.util.Iterator;
import rup.česky.vzory._16_iterátor.IterovatelnéPole;


/*******************************************************************************
 * Instance třídy <code>Odstavec</code> představují odstavce textu,
 * které mohou být složeny z různě formátovaných částí.
 */
public class Odstavec implements Iterable<Text>
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final IterovatelnéPole<Text> pole;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*************************************************************************
     * Vytvoří odstavec složený ze zadaných částí textu.
     * @param text  Části textu tvořící odstavec
     */
    public Odstavec( Text... text )
    {
```

```
        //Zodpovědnost za korektní průchod polem textů
        //přehazuje na instanci třídy IterovatelnéPole
        pole = new IterovatelnéPole<Text>( text );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Vrátí iterátor vracející postupně jednotlivé části textu
     */
    public Iterator<Text> iterator()
    {
        return pole.iterator();
    }

}
```

Celek

Celý sázený text je sazeči předán jako posloupnost odstavců, přesněji jako jednorozměrné pole instancí třídy `Odstavec`.

Textová reprezentace

Říkali jsme si, že vedle objektové reprezentace, kterou jsme si právě popsali, může být text předán sazeči i v textové podobě doplněné formátovacími značkami. Všechny formátovací značky budou uvozeny znakem ¤ (obecný znak měny, nebo chceš-li „sluníčko“) následovaným znakem definujícím danou značku:

Formátovací značky

- ¤B Začátek tučného textu
- ¤b Konec tučného textu
- ¤I Začátek šikmého textu (kurzivy)
- ¤i Konec šikmého textu
- ¤p Konec odstavce

Omezení

Aby byl program co nejjednodušší, neumožňuje zadat znak ¤. Teoreticky by jej bylo sice možno zadávat jako dvojici těchto znaků (obdobně jako je to v Javě se znakem \), ale to by vyžadovalo nestandardní ošetření a já jsem chtěl program maximálně zjednodušit, protože i tak je už tato kapitola výrazně delší, než je průměr délek kapitol v knize. Všemi „zdokonalovacími úpravami“, o kterých se tu občas zmiňuji, by se kód nejen prodloužil, ale navíc by se zašuměl drobnostmi, které sice dělají výsledek dokonalejší, ale odvádí pozornost čtenáře od toho, co se mu tu snažím vysvětlit.

Sázecí stroje

420. Řekl bych, že ty nejdůležitější variace se týkají druhu vytvářených objektů. Pověz mi něco o nich.

Po sazeči budeme chtít, aby výsledný dokument vysadil v zadaném tvaru, který bude některým z několika možných výsledných tvarů. K tomu, aby byl sazeč schopen vytvořit dokument v požadovaném tvaru, dostane k dispozici sázecí stroj.

Rozhraní ISázecíStroj

Požadované schopnosti sázecích strojů, které budou převádět dodané texty do jednotlivých výstupních tvarů, deklaruje rozhraní `ISázecíStroj`.

Výpis 24.3: Definice rozhraní `ISázecíStroj`

```
package rup.česky.vzory._24_stavitel.text;

/*******************************************************************************
 * Instance rozhraní <code>ISázecíStroj</code> jsou schopny převádět zadaný text
 * do svého textového formátu včetně jednoduchého formátování.
 */
public interface ISázecíStroj
{
//== DEKLAROVANÉ METODY ========================================================

    /*******************************************************************************
     * Konvertuje běžný text.
     *
     * @return Text po konverzi.
     */
    public String běžný( String text );


    /*******************************************************************************
     * Konvertuje tučný text.
     * @return Text po konverzi.
     */
    public String tučný( String text );


    /*******************************************************************************
     * Konvertuje šikmý text, tj. text označovaný jako kurziva (italic).
     * @return Text po konverzi.
     */
    public String šikmý( String text );


    /*******************************************************************************
     * Konvertuje celý odstavec.
     * @return Text po konverzi.
     */
    public String odstavec( String text );


    /*******************************************************************************
     * Konvertuje celý dokument.
     * @return Text po konverzi.
     */
    public String dokument( String text );

}
```

Tři druhy sázecích strojů

Definoval jsem tři druhy sázecích strojů. Jeden vytváří texty ve formátu HTML, druhý ve formátu RTF a třetí generuje obyčejné texty bez formátování. Počítej ale s tím, že sázecí stroje pro vytváření dokumentů ve formátu HTML a RTF jsou schopny vytvořit pouze zjednodušené verze obou formátů, které jsou na mezi toho, co jsou příslušné čtecí programy ochotny akceptovat. Naším cílem totiž není generovat dokonalé texty, ale vysvětlit si funkci návrhového vzoru *Stavitel*. Proto také sazeč neukládá výsledné dokumenty přímo do souborů, ale pouze je vrací jako textové řetězce.

Rozhraní sázecích strojů již známe, takže se můžeme podívat, jak jsou jednotlivé sázecí stroje definovány. Jak jsem již řekl, sázecí stroje jsou vytvořeny tři:

- StrojHTML
- Instance třídy `StrojHTML` vytvářejí texty ve zjednodušené verzi formátu HTML. Zdrojový kód najdete ve výpisu 24.4.
- StrojRTF
- Instance třídy `StrojRTF` vytvářejí texty ve zjednodušené verzi formátu RTF. Zdrojový kód najdete ve výpisu 24.5.
- StrojTXT
- Instance třídy `StrojTXT` vytvářejí prosté neformátované textové dokumenty. Zdrojový kód najdete ve výpisu 24.6.

Všechny tři sázecí stroje vystačí s implicitními konstruktory, protože se vše potřebné dozvědí až ve chvíli, kdy budou požádány o nějakou službu.

Výpis 24.4: Definice třídy `StrojHTML`

```
package rup.česky.vzory._24_stavitel.text;


/*******************************************************************************
 * Instance třídy <code>StrojHTML</code> jsou schopny převádět zadaný text
 * do formátu HTML včetně jednoduchého formátování.
 */
public class StrojHTML implements ISázecíStroj
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Konvertuje běžný text tak, že znaky s kódem nad 127 převede
     * na jejich 7bitovou html reprezentaci.
     *
     * @return Text po konverzi.
     */
    public String běžný( String text )
    {
        StringBuilder sb = new StringBuilder();
        for( int i=0;   i < text.length();   i++ ) {
            char c = text.charAt(i);
            if( c < 128 )
                sb.append( c );
            else
                sb.append('&').append('#').append((int)c).append(';');
        }
        return sb.toString();
    }


    /*******************************************************************************
     * Konvertuje tučný text.
     * @return Text po konverzi.
     */
    public String tučný( String text )
    {
        return "<b>" + běžný(text) + "</b>";
    }
```

```
    /*****************************************************************************
     * Konvertuje šikmý text, tj. text označovaný jako kurziva (italic).
     * @return Text po konverzi.
     */
    public String šikmý( String text )
    {
        return "<i>" + běžný(text) + "</i>";
    }


    /*****************************************************************************
     * Konvertuje celý odstavec.
     * @return Text po konverzi.
     */
    public String odstavec( String text )
    {
        return "<p>" + text + "</p>";
    }


    /*****************************************************************************
     * Konvertuje celý dokument.
     * @return Text po konverzi.
     */
    public String dokument( String text )
    {
        return "<html>" + text + "</html>";
    }

}
```

Výpis 24.5: Definice třídy `StrojRTF`

```
package rup.česky.vzory._24_stavitel.text;

import java.io.UnsupportedEncodingException;


/*******************************************************************************
 * Instance třídy <code>PísařRTF</code> jsou schopny převádět zadaný text
 * do formátu RTF včetně jednoduchého formátování.
 */
public class StrojRTF implements ISázecíStroj
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Konvertuje běžný text tak, že znaky s kódem nad 127 převede
     * na jejich 7bitovou html reprezentaci.
     *
     * @return Text po konverzi.
     */
    public String běžný( String text )
    {
        //Převádí text do kódování středoevropských Windows
        final String sada = "Windows-1250";
```

```
        byte[] znaky;
        try {
            znaky = text.getBytes( sada );
        } catch (UnsupportedEncodingException ex) {
            throw new RuntimeException ("\nNepodporovaná znaková sada: "+sada);
        }
        StringBuilder sb = new StringBuilder();
        for( byte b : znaky ) {
            if( b > 0 )   //tj. kód znaku < 128
                sb.append( (char)b );
            else
                sb.append('\\').append('\'')
                  .append(String.format("%02x", b));
        }
        return sb.toString();
    }


    /**************************************************************************
     * Konvertuje tučný text.
     * @return Text po konverzi.
     */
    public String tučný( String text )
    {
        return "\\b " + běžný(text) + "\\b0 ";
    }


    /**************************************************************************
     * Konvertuje šikmý text, tj. text označovaný jako kurziva (italic).
     * @return Text po konverzi.
     */
    public String šikmý( String text )
    {
        return "\\i " + běžný(text) + "\\i0 ";
    }


    /**************************************************************************
     * Konvertuje celý odstavec.
     * @return Text po konverzi.
     */
    public String odstavec( String text )
    {
        return text + "\\par ";
    }


    /**************************************************************************
     * Konvertuje celý dokument.
     * @return Text po konverzi.
     */
    public String dokument( String text )
    {
        return "{\\rtf1\\ansi\\ansicpg1250 " + text + "}";
    }

}
```

Výpis 24.6: Definice třídy `PisarTXT`

```
package rup.česky.vzory._24_stavitel.text;

import java.io.UnsupportedEncodingException;


/*******************************************************************************
 * Instance třídy <code>StrojTXT</code> jsou schopny převádět zadaný text
 * do formátu TXT a ignorovat přitom nastavené formátování
 * s výjimkou rozdělení dokumentu do odstavců.
 */
public class StrojTXT implements ISázecíStroj
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    public String běžný   ( String text )  { return text; }
    public String tučný   ( String text )  { return text; }
    public String šikmý   ( String text )  { return text; }
    public String odstavec( String text )  { return text + "\n"; }
    public String dokument( String text )  { return text; }
}
```

Definice sazeče

421. Takže teď už zbývá jen sazeč.

Třída `Sazeč` je završením celého návrhu. Její instance mají za úkol analyzovat vstupní texty a předávat je sázecímu stroji k převedení do požadovaného formátu. Výsledek obdržený od sázecího stroje pak předají zákazníkovi, jenž jim dodal text ke zpracování.

Výpis 24.7: Definice třídy `Sazeč`

```
package rup.česky.vzory._24_stavitel.text;

import rup.česky.vzory._24_stavitel.text.Text.Typ;

import static rup.česky.vzory._24_stavitel.text.Text.Typ.*;


/*******************************************************************************
 * Instance třídy <code>Sazeč</code> mají na starosti vysazení textu.
 * Aby mohly vytvořit text v požadovaném výstupním tvaru,
 * dostanou k ruce instanci rozhraní <code>ISázecíStroj</code>,
 * která umí generovat text v požadovaném výstupním tvaru.
 */
public class Sazeč
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final ISázecíStroj stroj;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /***************************************************************************
     * Vytvoří novou instanci sazeče, která ihned dostane k ruce sázecí stroj,
```

```
     * jenž je schopen vysazovat dodané texty do svého formátu.
     *
     * @param stroj Sázecí stroj, který je schopen vysadit text
     *              v požadovaném formátu
     */
    public Sazeč( ISázecíStroj stroj )
    {
        this.stroj = stroj;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /****************************************************************************
     * Převede text zadaný jako pole odstavců do řetězce ve tvaru
     * odpovídajícím sázecímu stroji daného sazeče.
     *
     * @param text  Posloupnost odstavců tvořících sázený dokument
     * @return Posloupnost znaků vytvořeného dokumentu
     *         uložitelná do souboru odpovídajícího typu
     */
    public String vysaď( Odstavec... text )
    {
        //Builder, v němž bude skládán výsledný dokument
        StringBuilder sbd = new StringBuilder();

        for( Odstavec odst : text ) {
            //Builder, v němž bude skládán daný odstavec
            StringBuilder sbo = new StringBuilder();
            for( Text txt  : odst ) {
                String s = txt.getText();
                switch( txt.getTyp() )
                {
                    case BĚŽNÝ:   s = stroj.běžný( s );   break;
                    case TUČNÝ:   s = stroj.tučný( s );   break;
                    case ŠIKMÝ:   s = stroj.šikmý( s );   break;
                    default:
                        throw new IllegalArgumentException(
                            "Nepodporovaný typ textu: " + txt.getTyp() );
                }
                sbo.append( s );    //Přidá zpracovaný text do odstavce
            }
            //Přidá další odstavec do dokumentu
            sbd.append( stroj.odstavec( sbo.toString() ) );
        }
        return stroj.dokument( sbd.toString() );
    }


    /****************************************************************************
     * Převede zadaný text doplněný formátovacími značkami do řetězce ve tvaru
     * odpovídajícím sázecímu stroji daného sazeče.
     * <p>
     * Formátovací značky jsou uvozeny znakem ¤ ("sluníčko", znak měny),
     * za nímž následuje znak specifikující formátování:
     * <ul><li> B - Začátek tučného písma </li>
     *     <li> b - Konec tučného písma </li>
     *     <li> I - Začátek šikmého písma </li>
```

```
     *     <li> b - Konec šikmého písma </li>
     *     <li> p - Konec odstavce</li>
     * </ul>
     * Znak ¤ není možno zadat. Bylo by jej sice možno přidat
     * např. jako zdvojení znaku ¤¤, ale v zájmu maximální jednoduchosti
     * programu bylo od této možnosti upuštěno.
     *
     * @param text  Převáděný text doplněný formátovacími značkami
     * @return Posloupnost znaků vytvořeného dokumentu
     *         uložitelná do souboru odpovídajícího typu
     */
    public String vysaď( String text )
    {
        StringBuilder
            sbd = new StringBuilder(),  //Builder pro dokument
            sbo = new StringBuilder();  //Builder pro odstavce
        int TL = text.length();
        Typ řez = BĚŽNÝ;

        for( int start=0, i=0;   i < TL;   i++ ) {

            while( text.charAt( i++ )  !=  '¤' );
            char znak = text.charAt(i); //Znak za znakem ¤
            řez = ověřŘez( znak, řez, i, text );

            //Zpracujeme text až k přečtené formátovací značce
            String s = text.substring( start, i-1 );
            switch( znak )
            {               //Není-li mezi poslední formátovací značkou
                            //a značkou konce odstavce žádný text
                            //Nemusíme se jej snažit přidávat
                case 'p':   if( start == i-1 ) break;
                case 'B':
                case 'I':   s = stroj.běžný( s );   break;
                case 'b':   s = stroj.tučný( s );   break;
                case 'i':   s = stroj.šikmý( s );   break;
                default:    CHYBA( i, text );
            }
            sbo.append( s );    //Přidáme text do odstavce
            start = i+1;        //S dalším čtením začneme za značkou
            if( znak != 'p' )
                continue;       //Není-li konec odstavce, čteme další text

            //Již byl uzavřen odstavec - přidáme jej do dokumentu
            sbd.append( stroj.odstavec( sbo.toString() ) );
            sbo = new StringBuilder(); //Vyprázdníme builder pro příští odstavec
        }
        return stroj.dokument( sbd.toString() );
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ======================================

    /**************************************************************************
     * Ověří, jestli zadaná formátovací značka byla zadána korektně.
     *
     * @param Přečtený formátovací znak
     * @param typ  Současný typ formátování
```

```
     * @param i    Index, kde byla nalezena značka (pouze pro chybové hlášení)
     * @param text Zpracovávaný text (pouze pro chybové hlášení)
     *
     * @return Typ následujícího textu
     */
    private Typ ověřŘez( char c, Typ typ, int i, String text )
    {
        if( typ == BĚŽNÝ ) {
            //Běžný formát lze změnit na libovolný jiný či v něm uzavřít odst.
            if( c == 'I' )
                typ = ŠIKMÝ;
            else if( c == 'B' )
                typ = TUČNÝ;
            else if( (c != 'p') )
                //Buďto se jednalo o uzavírací formátovací znak
                //nebo o zcela neznámý formátovací znak
                CHYBA( i, text );
        }else if( ((typ == TUČNÝ)  &&  (c == 'b'))  ||
                  ((typ == ŠIKMÝ)  &&  (c == 'i')) ) {
            //Uzavření aktuálně nastaveného formátování
            typ = BĚŽNÝ;
        }else
            //Cokoliv jiného způsobí chybu
            CHYBA( i, text );
        return typ;
    }


    /*************************************************************************
     * Vyhodí výjimku a v její doprovodné zprávě se pokusí ukázat,
     * kde byla v zadávaném textu nalezena.
     */
    private void CHYBA( int i, String text )
    {
        throw new IllegalArgumentException(
            "\nNepodporovaný typ textu: " +  text.substring( 0, i-1) +
            "\n" + text.substring( i-1, i+1) +
            "\n" + text.substring( i+1 ) );
    }

}
```

Testovací autor

422. Ještě by to chtělo nějak ověřit, že vše funguje.

Pro otestování celého mechanismu slouží třída `Autor`. Ta nemá žádné instance a jejím účelem je opravdu pouze test funkčnosti sazeče.

Autor vypisuje řetězce obdržené od sazeče do standardního výstupu, kde si je můžeš prohlédnout.

Výpis 24.8: Definice testovací třídy `Autor`

```
package rup.česky.vzory._24_stavitel.text;

import static rup.česky.vzory._24_stavitel.text.Text.Typ.*;
```

```
/*******************************************************************************
 * Třída <code>Autor</code> slouží k testování správné funkce
 * instancí třídy <code>Sazeč</code>.
 */
public class Autor
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Předpřipravený vzorek testovacího textu, který bude autor tvořit. */
    private static final Odstavec[] strukturovaný =
    {
        new Odstavec(
            new Text( TUČNÝ, "Různé druhy textu:" )
            ),
        new Odstavec(
            new Text( BĚŽNÝ, "Prostý text je zcela bez příznaků, " ),
            new Text( TUČNÝ, "tučný text" ),
            new Text( BĚŽNÝ, " a " ),
            new Text( ŠIKMÝ, "ležatý text" ),
            new Text( BĚŽNÝ, " je třeba označit." )
        ),
        new Odstavec(
            new Text( BĚŽNÝ, "Náš konvertor by měl rozeznávat jednotlivé " ),
            new Text( TUČNÝ, "odstavce" ),
            new Text( BĚŽNÝ, " a uvnitř odstavců také " ),
            new Text( ŠIKMÝ, "různé druhy zvýraznění" ),
            new Text( BĚŽNÝ, "." )
        ),
    };

    private static final String řetězcový =
        "¤BRůzné druhy textu:¤b¤p" +
        "Prostý text je zcela bez příznaků, ¤Btučný text¤b a ¤Iležatý text¤i "+
        "je třeba označit.¤pNáš konvertor by měl rozeznávat jednotlivé "+
        "¤Bodstavce¤b a uvnitř odstavců také ¤Irůzné druhy zvýraznění¤i.¤p";

//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================
    private Autor() {}

//== TESTY =====================================================================

    public static void test()
    {
        Sazeč html = new Sazeč( new StrojHTML() ),
              rtf  = new Sazeč( new StrojRTF()  ),
              txt  = new Sazeč( new StrojTXT()  );

        System.out.println(
            "============================================================\n\n"+
            "HTML strukturovaný:\n" + html.vysaď( strukturovaný ) +
       "\n\n============================================================\n\n"+
            "RTF strukturovaný:\n" + rtf.vysaď( strukturovaný )  +
       "\n\n============================================================\n\n"+
            "TXT strukturovaný:\n" + txt.vysaď( strukturovaný )  +
         "\n============================================================"+
         "\n============================================================\n\n"+
```

```
            "HTML řetězcový:\n" + html.vysaď( řetězcový ) +
        "\n\n==========================================================\n\n"+
            "RTF řetězcový:\n" + rtf.vysaď( řetězcový )  +
        "\n\n==========================================================\n\n"+
            "TXT řetězcový:\n" + txt.vysaď( řetězcový )  +
          "\n==========================================================" );
    }
    /** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {  test();  }
}
```

Možná rozšíření

423. Proč třída `Autor` neukládá výsledné texty rovnou do souborů?

Protože pak by bylo zbytečné, aby sazeč vytvářel celý řetězec, který autorovi vrátí.

Kdybychom chtěli udělat program užitečnější, mohli bychom definovat ve třídě `Sazeč` metody, které budou posílat vytvořený text přímo do výstupního proudu. Pak by ale bylo vhodné koncipovat třídu `Sazeč` jinak. Nevytvářela by řetězec představující celý dokument, ale posílala by své výstupy do výstupního proudu průběžně.

Pak by se ale měly obdobně chovat i sázecí stroje. Jejich metody by neměly zpracovávat celé bloky textu, ale pouze vracet příslušné prefixy a sufixy (předpony a přípony). Místo metody `tučný(String)` by mohly nabízet např. dvojici metod `zapniTučný()` a `vypniTučný()` a obdobně by se mohly „zapínat“ a „vypínat“ nejenom bloky nějak formátovaného textu, ale i celé odstavce a celý dokument.

Budeš-li se chtít procvičit, můžeš si podobnou verzi sám definovat. Svůj výtvor si pak můžeš porovnat s verzí, kterou najdeš mezi doprovodnými programy.

Shrnutí – co jsme se naučili

- Účelem vzoru *Stavitel* je umožnit využití stejného konstrukčního postupu pro různé vnitřní reprezentace konstruovaných objektů.
- Návrhový vzor stavitel se nesoustřeďuje na vlastní vytvoření objektu, ale na postup jeho tvorby.
- V návrhovém vzoru vystupují vedle *řídícího objektu* rovněž *výkonné objekty*.
- Řídící objekt má na starosti dodržení správného postupu vytvoření požadovaného objektu.
- Výkonné objekty mají na starosti vytvoření některých detailů podle pokynů řídícího objektu.
- Pro různé druhy vytvářených objektů (např. pro jejich různé vnitřní reprezentace) se používají různé výkonné objekty, tj. výkonné objekty, které jsou instancemi různých tříd.
- Aby mohly být výkonné objekty jednotně používány, implementují společné rozhraní, případně mají společného rodiče.

- Každý výkonný objekt je typicky schopen vytvořit kteroukoliv z částí požadovaných řídícím objektem. Jednotlivé výkonné objekty se liší pouze v tom, jak tuto část vytvoří.
- V některých případech jsou výkonné objekty schopny podle návodu řídícího objektu vytvořit i celý požadovaný objekt.
- Návrhový vzor *Stavitel* použijeme:
  - je-li algoritmus tvorby objektu závislý na použité vnitřní reprezentaci,
  - mají-li vytvářené objekty složitější vnitřní strukturu, a budují-li se proto v několika fázích,
  - je-li vhodné použít řídící objekt současně jako správce zdrojů používaných při tvorbě daných objektů.
- Návrhový vzor *Stavitel* patří mezi vzory uvedené v GoF.

KAPITOLA 25

Bude toho víc (Abstraktní továrna – Abstract Factory)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Definuje rozhraní pro tvorbu celé rodiny souvisejících nebo závislých objektů a tím odděluje klienta od vlastního procesu vytváření objektů.

[1] **Definice v GoF:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes. – Poskytuje rozhraní pro tvorbu rodin souvisejících nebo závislých objektů, aniž by byly předem specifikovány jejich konkrétní třídy.

Účel

424. Jak si mám představit *rozhraní pro tvorbu celé rodiny objektů*?

Objekty se společnou charakteristikou

Řada aplikací nabízí možnost výběru jakési základní charakteristiky následně vytvářených objektů. Tato charakteristika se přitom nevztahuje na jedinou třídu, ale na celou skupinu tříd, jejichž instance chceme následně vytvářet.

Příklad: look-and-feel

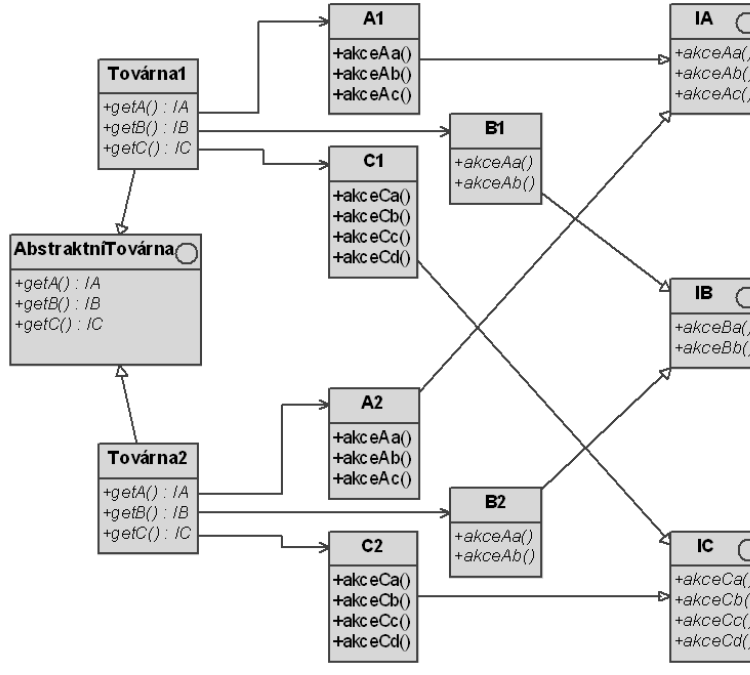
Abych ti to přiblížil, uvedu často citovaný příklad vzhledu uživatelského rozhraní, označovaného často anglickým termínem *look-and-feel*. V Javě si např. můžeš vybrat, jestli budeš používat nativní vzhled operačního systému, na němž právě pracuješ, nebo standardní vzhled Javy pojmenovaný *Ocean*, případně některý z dalších vzhledů.

Chování

Od chvíle, kdy si daný vzhled vybereš, bude subsystém mající na starosti kreslení GUI dostávat pro kreslení jednotlivých komponent (tlačítka, posuvníky, základní panely oken, …) instance sady tříd implementujících daný vzhled. Když svůj názor na požadovaný vzhled aplikace změníš, aplikace začne subsystém zásobovat instancemi zcela jiných tříd – těch, které implementují nově požadovaný vzhled.

Řešení: abstraktní tovární třída

Jednou z možností, jak takovouto funkcionalitu implementovat, je právě použití návrhového vzoru *Abstraktní továrna*. Podívej se na obrázek 25.1, na němž je diagram tříd popisující principiální schéma tohoto návrhového vzoru. Je sice trochu složitější než diagramy, s nimiž jsme se doposud setkali, ale doufám, že z něj vše pochopíš.



Obrázek 25.1
Principiální schéma návrhového vzoru Abstraktní továrna

425. Přivítal bych, kdybys ho se mnou prošel.

Příklad v diagramu

Pojďme. Rozhraní `AbstraktníTovárna` deklaruje společné vlastnosti továren, které budou vytvářet instance tří typů deklarovaných rozhraními `IA`, `IB` a `IC`. Na obrázku toto rozhraní implementují dvě tovární třídy označené jako `Továrna1` a `Továrna2`.

Práce s instancí abstraktní továrny

Klient nějakým způsobem získá instanci té správné abstraktní továrny. Která to bude, to záleží na tom, budeš-li chtít získávat instance prvního nebo druhého druhu. Pak tuto tovární instanci postupně žádá o aktuálně potřebné instance jednotlivých rozhraní. V celém programu pak tyto instance vystupují jako instance svých implementovaných rozhraní.

426. Teorie je to asi krásná, ale přivítal bych něco konkrétnějšího.

Vysvětlení obrázku

Zkusím to. Představ si, že `AbstraktníTovárna` představuje továrny vytvářející jednotlivé komponenty GUI. `Továrna1` se specializuje např. na komponenty vytvářející systémový vzhled GUI, `Továrna2` na komponenty vytvářející nějaký tvůj oblíbený vzhled. Jednotlivá rozhraní uvedená v pravé části diagramu pak zastupují jednotlivé komponenty: okna, tlačítka, textová pole, posuvníky atd.

Ty se rozhodneš pro nějaký vzhled GUI a podle toho zvolíš továrnu. Té si pak říkáš o jednotlivé komponenty a podle toho, kterou továrnu sis vybral, dostáváš buď komponenty pro rozhraní se systémovým vzhledem nebo naopak komponenty pro rozhraní s tvým oblíbeným vzhledem.

427. Jenomže když vytvářím GUI, tak si o komponenty neříkám nějaké továrně, ale získávám je přímo prostřednictvím volání konstruktoru dané komponenty.

Proč není používání továren ve swingu vidět

To je proto, že každá komponenta má svůj model, který je na jejím vzhledu nezávislý, a svůj vzhled, který jediný můžeš ovlivnit. Když budeš měnit *look and feel*, tak nezměníš chování tlačítek, zaškrtávacích políček či posuvníků, ale změníš pouze jejich vzhled.

Když proto požádáš konstruktor o komponentu, nedozvíš se nic o tom, kde konstruktor k informacím o rámcovém vzhledu komponenty přišel. Proto se také nedozvíš, že někde v hloubi je abstraktní továrna, která má na starosti tvorbu některých objektů týkajících se vzhledu komponent.

428. Tohle už sice bylo lepší, ale přece jen bych tě požádal, abys zkusil něco ještě konkrétnějšího.

Příklad: virtuální svět

Dobrá. Představ si, že máš nějaký virtuální svět, v němž jsou domy, stromy a auta. Schopnosti každého z objektů jsou popsány rozhraními `IDům`, `IStrom` a `IAuto` (dejme tomu, že všechny můžeš umístit na zadané souřadnice, strom umí navíc růst a auto se plynule přesunout).

Tři pohledy

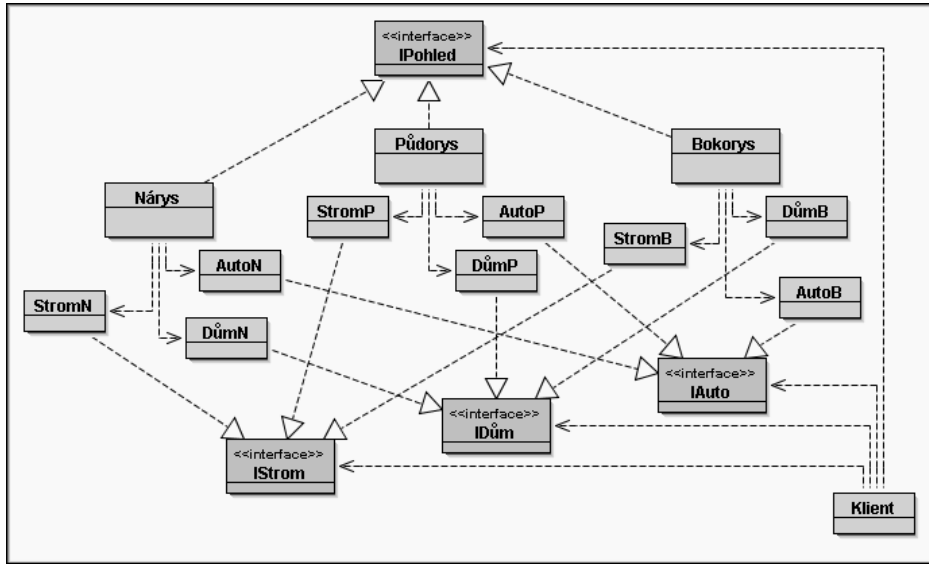
Na tento virtuální svět se můžeš dívat buď shora (půdorys), zepředu (nárys) anebo z boku (bokorys). Při každém pohledu vypadá příslušný objekt jinak a při umísťování na zadané souřadnice se vykreslí na jiném místě plátna.

Celý systém tříd, které vykreslují zmíněné objekty, si můžeš prohlédnout na obrázku 25.2. Všechny tovární třídy implementují rozhraní `IPohled`, které deklaruje, že budou umět na požádání vrátit instance rozhraní `IAuto`, `IDům` a `IStrom`.

Rozhraní IPohled

Rozhraní `IPohled` bychom mohli nahradit abstraktní třídou, která by měla jednoduchou tovární metodu, jež by dodala „tovární instanci“ (tj. instanci schopnou dodávat instance vyjmenovaných rozhraní) požadovaného typu (tj. pro nárys, resp. bokorys, resp. půdorys). Budeme ale předpokládat, že tyto „tovární instance“ umíme získat od zbytku aplikace, a spokojíme se proto s tím, že jejich společným rodičem bude rozhraní.

Jak si můžeš domyslet z obrázku, klient se bude k těmto „továrním instancím“ obracet jako k instancím rozhraní `IPohled` a k jimi vytvořeným instancím jako k instancím příslušných rozhraní. Celá složitost toho, že v různých pohledech slouží k vykreslování různé třídy, může zůstat před klientem skryta.



Obrázek 25.2
Diagram tříd objektů virtuálního světa

429. Začínám to pomalu chápat. Instance abstraktní továrny, nebo jak ty říkáš „tovární instance“, jsou vlastně taková „sdružení továrních metod“ vedených nějakou jednotnou ideou.

Příklad z politiky

Tak nějak. Když už jsi začal o těch idejích, zkusím ti to předvést na příkladu ze života. Představ si, že abstraktní továrnou ekvivalentní našemu rozhraní `IPohled` z minulého příkladu je `PolitickáStrana`. Má různé dceřiné třídy: `ODS`, `SNK`, `Zelení`, `KDU`, `ČSSD` atd. (to jsou ekvivalenty našich pohledů). Každá jejich instance, tj. každá konkrétní politická strana, musí být schopna vygenerovat ze svého středu instance implementující rozhraní `Premiér`, `Ministr`, `Hejtman`, `Starosta` a další.

Ve volbách se vybere některá politická strana, tj. instance rozhraní `PolitickáStrana`, a tu pak okolnosti žádají, aby generovala osoby na příslušné funkce. Podle toho, která politická strana dosadí do příslušné funkce svoji instanci, bude vypadat chování dané

instance. Dodá-li ministra financí instance třídy `ODS`, je zřejmé, že se jeho chování bude výrazně lišit od chování ministra financí dodaného instancí třídy `ČSSD`.

Implementace

430. Tak to byl opravdu příklad ze života. A řekl bych, že i leccos naznačuje. Například to, jak je (alespoň teoreticky) jednoduché vyměnit tovární instanci (= stranu) a následně vyměnit staré instance (ministry, hejtmany atd.) za nové a tím změnit chování celého programu, tj. státu.

Vliv abstraktní továrny na chování systému

Abstraktní továrna umožňuje relativně jednoduše ovlivňovat chování celého systému. Stačí ve vhodný okamžik jenom vyměnit instanci (v našem posledním příkladu zvolit stranu, v předchozím příkladu změnit úhel pohledu) a pak už naprogramované mechanismy tuto změnu postupně zanesou do celého systému.

Ten příklad s politickými stranami ale také ukazuje jedno nebezpečí, kterého se musíš v programu vyvarovat. Sám víš, jaké to přináší problémy, polovinu funkcí obsadí instance jedné strany a polovinu instance strany druhé. Ve svých programech musíš ohlídat, aby se při výměně instance abstraktní továrny opravdu vyměnily instance na všech postech, jinak tvůj program dopadne stejně jako naše politická scéna.

Nevýhoda: špatně rozšiřitelné rozhraní

431. Ještě na něco si mám dávat pozor?

Použití abstraktní továrny s sebou přináší jeden nepříjemný důsledek: špatně se rozšiřuje množina rozhraní, jejichž instance mohou instance abstraktní továrny generovat.

Příklad: virtuální svět

Odpoutám se od politické scény a vrátím se k našemu virtuálním světu. Máš-li navržený nějaký systém s řadou použití abstraktní továrny určené pro realizaci nejrůznějších náhledů na svět, bude se ti špatně přidávat další komponenta – např. osoba. Nemůžeš totiž pouze jednoduše rozšířit množinu metod deklarovaných rozhraním `IPohled` o metodu vracející instanci rozhraní `IOsoba`, ale musíš také „obejít" všechny je implementující třídy, v každé implementovat tovární metodu pro vytvoření dané komponenty a spolu s ní také vytvořit třídu realizující danou komponentu s respektováním příslušného náhledu.

Na počátku si proto musíš vždy důkladně rozmyslet, které druhy výkonných prvků budeš potřebovat, abys pak nemusel jejich spektrum někdy později měnit.

432. To zní dost nepříjemně. Zvlášť když si pomyslím, že málokdy je programátor v počátečních fázích návrhu schopen myslet na všechny eventuality. Nedalo by se to nějak obejít?

Použití univerzální tovární metody

Jistá možnost by tu byla. Snižuje ale robustnost celého programu. Abstraktní továrna nemusí deklarovat pro každý vytvářený typ instance speciální metodu, ale může definovat jednu univerzální metodu, která se ve svém parametru dozví, čí instanci má vytvořit a vrátit.

Vada: syntaktická chyba přejde v běhovou

Nepříjemnou věcí ovšem je, že toto řešení stěhuje chybu, která byla původně syntaktická, a projevila se proto hned při překladu, mezi chyby sémantické, na které se přijde až za běhu programu.

433. Když nad tím tak přemýšlím, tak si říkám, že implementace abstraktní továrny budou nejspíš jedináčci.

Jedináčkovství továrních instancí

Ano. Není třeba, aby každá z továren měla několik instancí, když nakonec musí stejně všechny dělat totéž – vracet instance deklarovaných rozhraní odpovídající duchu dané továrny.

Časté používání vzoru *Prototyp*

Navíc tovární metody v těchto továrních instancích často používají k vytváření požadovaných instancí prototypy, které jsou v řadě situací nejjednodušším řešením svěřeného úkolu. Pamatuj na to, až budeš někdy tento návrhový vzor aplikovat.

Příklad

434. Chtělo by to nějaký jednoduchý příkládek.

Praktické aplikace abstraktní továrny jsou rozsáhlé

S příkladem je to trochu horší. Aplikace tohoto návrhového vzoru, s nimiž jsem se setkal v praxi, byly pro takovýto vstupní kurz příliš složité. Příklady, které jsem potkal v různých učebnicích a kurzech, byly zase takové primitivní AHA-příklady, které mne nijak neoslovily a připadaly mi spíše jako takové příklady pro příklady.

Použijeme příklad s virtuálním městem

Rozhodl jsem se proto, že nejlepší bude dotáhnout příklad s virtuálním městem. Je sice ve skutečnosti ještě trochu složitější než na schematickém obrázku 25.2, ale na druhou stranu příklady na vzor *Abstraktní továrna*, které by nebyly čistými AHA-příklady, jsou bohužel složitější.

Skutečný diagram tříd doprovodného programu si můžeš prohlédnout na obrázku 25.3. Aby se v něm dalo alespoň trochu vyznat, nechal jsem zobrazovat jenom šipky dědičnosti a zobrazování šipek závislostí jsem potlačil, protože v té změti čar už by ses těžko orientoval.

435. No, je pěkně nahuštěný. Pojď mi jej tedy po kouscích rozebrat.

Jdeme na to. Protože jsem ale dělal tento příklad jako poslední a stránky, na nichž jsem se s nakladatelem domluvil, jsem už vyčerpal[1], nebudu ti tu předvádět celý program, ale zobrazím pouze klíčová rozhraní. Budeš-li se zajímat o další třídy, můžeš si je najít mezi doprovodnými programy.

Jádrem je třída `Svět`

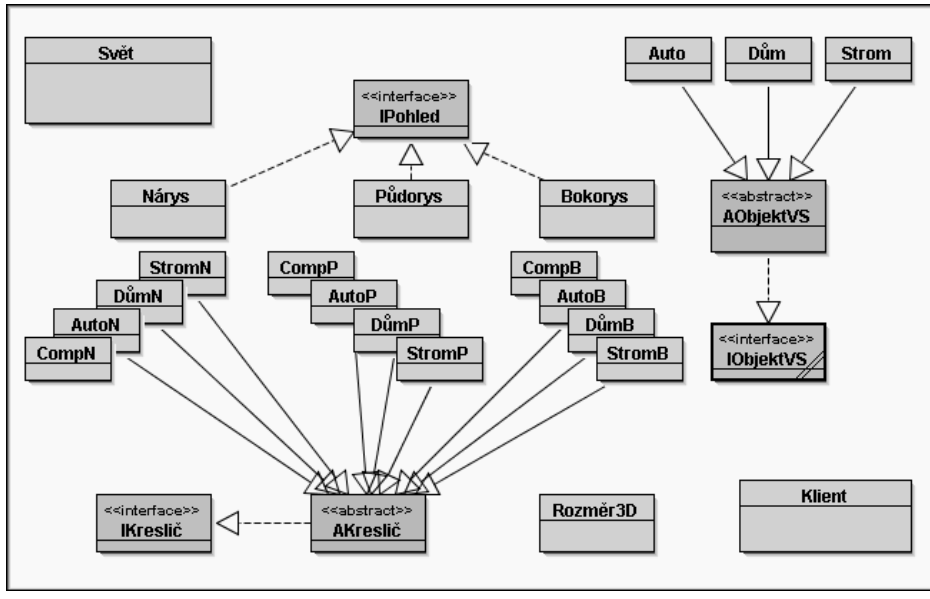
Jádrem celé aplikace je třída `Svět`, jejíž instance je jedináček a zároveň pracuje jako abstraktní továrna. Ve světě se vyskytují objekty, které jsou instancemi rozhraní `IObjektVS` (= objekt virtuálního světa). Abstraktní třída `AObjektVS` je společným rodičem existujících tříd objektů, jimiž jsou prozatím třídy `Auto`, `Dům` a `Strom`. Budeš-li mít ale chuť, můžeš si přidat vlastní.

Aplikace je skromná, aby zůstala malá

Protože se jedná pouze o demonstrační aplikaci, je rozhraní ve svých požadavcích skromné. Kdybys definoval podobnou aplikaci, která by měla opravdu simulovat svět pro nějakou hru, bylo by asi výrazně bohatší.

---

[1] Přiznejme si, že nebýt tohoto omezení, byla by kniha nakonec tak tlustá, že by si ji nikdo nekoupil.



Obrázek 25.3
Diagram tříd simulace virtuálního světa

Výpis 25.1: Rozhraní `IObjektVS` definující vlastnosti objektů vyskytujících se v simulovaném virtuálním světě

```
package rup.česky.vzory._25_abstraktní;

import rup.česky.tvary.IPosuvný;


/*******************************************************************************
 * Instance rozhraní <code>IObjektVS</code> představují objekty virtuálního
 * světa spravované instancí třídy {@link Svět}.
 */
public interface IObjektVS extends IPosuvný
{
//== DEKLAROVANÉ METODY ========================================================

    /*---------------------------------------------------------------------------
     * Vrátí svého kresliče.
     */
    public IKreslič getKreslič();


    /*---------------------------------------------------------------------------
     * Vrátí přepravku, jež je instancí třídy <code>Rozměr3D</code>
     * a obsahuje všechny tři rozměry daného objektu.
     */
    public Rozměr3D getRozměr3D();
```

```
    /*************************************************************************
     * Nastaví rozměry objektu.
     */
    public void setRozměr3D( int dx, int dy, int dz );


    /*************************************************************************
     * Nastaví rozměry objektu.
     */
    public void setRozměr3D( Rozměr3D rozměr );


    /*************************************************************************
     * Přizpůsobí zobrazování objektu novému pohledu.
     *
     * @param továrna  Továrna na instance realizující zobrazení
     *                 v aktuálně nastaveném pohledu
     */
    public void nastavPohled( IPohled továrna );

}
```

436. Proč je v projektu definováno rozhraní `IObjektVS`, když už je tam třída `AObjektVS`, která společné rozhraní všech objektů definuje také.

Proč je vedle třídy `AObjektVS` definováno i rozhraní

To máš sice pravdu, ale toto řešení ti umožní časem přidat i takové objekty, které by nebylo moudré definovat jako potomky třídy `AObjektVS`. Vzpomeň si např. na knihovnu kontejnerů – ta je definována obdobně.

437. Dobře, dobře, už jsem zapomněl. Tak pokračuj další skupinou.

Chování a zobrazování je vhodné oddělit

Objekty našeho virtuálního světa mají jisté vlastnosti a schopnosti. Mezi nimi je i schopnost zobrazit se na displeji. Není ale moudré směšovat vlastní simulaci chování objektu a jeho zobrazování, protože tím je příliš provážeš a ztížíš si budoucí úpravy kterékoliv z obou složek (podrobněji si na toto téma budeme povídat v kapitole *Každý chvilku tahá pilku (Model-Pohled-Ovládání – Model-View-Controller)* na straně 425).

Zobrazování mají na starosti kresliče

Zobrazování objektů mají proto na starosti speciální kresliče, což jsou instance tříd implementujících rozhraní `IKreslič`. Všimni si, že kreslič toho zdánlivě moc neumí: rozhraní po něm požaduje pouze tři metody:

- zděděnou metodu `nakresli(Kreslítko)`, která bude vyvolána v okamžiku, kdy bude mít kreslič za úkol objekt vykreslit;
- pomocnou metodu `getČelníSouřadnice()`, která vrací hodnotu, podle níž svět pozná, který objekt je v pozadí, a má být proto kreslen dříve, a který je naopak v popředí, a má být proto kreslen později;
- metodu `objektZměněn()`, pomocí níž okolí kresliči oznamuje, že se s objektem něco stalo (např. se pohnul), takže si musí „vykorespondovat“ nové parametry, aby jej byl opět schopen správně kreslit. Díky této metodě se nemusí objektu ptát na jeho parametry při každém jeho překreslování, ale zeptá se pouze tehdy, pokud se doopravdy něco změnilo.

Výpis 25.2:

```
package rup.česky.vzory._25_abstraktní;

import rup.česky.tvary.IKreslený;

/*******************************************************************************
 * Instance rozhraní <code>IKreslič</code> jsou zodpovědné za správné
 * vykreslení svých majitelů při daném pohledu na virtuální svět
 */
public interface IKreslič extends IKreslený
{
//== DEKLAROVANÉ METODY ========================================================

    /*******************************************************************************
     * Vrátí hodnotu souřadnice, podle níž se posuzuje,
     * který objekt bude zobrazován dříve.
     */
    public int getČelníSouřadnice();


    /*******************************************************************************
     * Upozorňuje kreslič, že některé parametry objektu se změnily,
     * takže si je bude muset při příštím vykreslování zjistit znovu.
     */
    public void objektZměněn();

}
```

438. Těch kresličů je ale celá řada. Jak objekt pozná, který je v danou chvíli ten pravý?

Proč tolik tříd

Jak jsme si již řekli, na svět se můžeme dívat různými pohledy a v každém pohledu budou objekty vypadat výrazně jinak. Není moudré nacpat všechny možnosti do jediné třídy – proto je těch kresličů tolik.

Jak objekt získá svůj kreslič

Aplikace je navržena tak, aby pro každý pohled a každou třídu existoval speciální kreslič. Když se instance třídy `Svět` dozví, že je třeba zprostředkovat jiný pohled na svět, obvolá všechny své objekty (tuto techniku si vysvětlíme podrobněji v kapitole *Až se to stane, dám ti vědět (Pozorovatel – Observer)* na straně 375) a požádá je, aby aktualizovaly své kresliče. K tomuto účelu jim dodá instanci rozhraní `IPohled`, jež obsahuje tovární metody pro všechny kresliče. Každý objekt pak zavolá metodu, která mu vrátí konkrétní kreslič.

Rozdělení zodpovědnosti

Jinými slovy: instance třídy `Svět` je zodpovědná za dodání správné sady kresličů, které jsou schopny kreslit objekty v zadaném pohledu, a objekty virtuálního světa jsou pak zodpovědné za to, že si ze sady kresličů pro daný pohled vyberou kreslič, který bude schopen kreslit právě je.

Výpis 25.3:

```
package rup.česky.vzory._25_abstraktní;

import java.util.Comparator;
```

```
/**********************************************************************
 * Rozhraní <code>IPohled</code> definuje sadu metod pro tvorbu objektů
 * vystupujících ve virtuálním světě.
 */
public interface IPohled
{
//== KONSTANTY ================================================================
//== METODY ===================================================================

    /**********************************************************************
     * Vytvoří na zadaných souřadnicích nové auto.
     */
    public IKreslič kresličAuto( Auto auto );


    /**********************************************************************
     * Vytvoří na zadaných souřadnicích nový dům.
     */
    public IKreslič kresličDům( Dům dům );


    /**********************************************************************
     * Vytvoří na zadaných souřadnicích nový Strom.
     */
    public IKreslič kresličStrom( Strom strom );


    /**********************************************************************
     * Vrátí komparátor, který porovná instance, aby bylo možno určit,
     * která se má kreslit dříve.
     */
    public Comparator<IObjektVS> komparátor();


    /**********************************************************************
     * Vrátí pohled, v němž všechny vytvořené kresliče pracují.
     */
    public Svět.Pohled getPohled();

}
```

439. Na obrázku 25.2 na straně 318 implementovala každá sada kresličů jiné rozhraní. Nyní všechny třídy kresličů implementují společné rozhraní `IKreslič`. Proč?

Proč implementují společné rozhraní

Protože je v aplikaci už tak hromada tříd a já jsem ji chtěl alespoň trochu zjednodušit. U abstraktní továrny nejde o to, aby každá tovární metoda vracela instanci jiného typu, ale aby každá vracela instanci specializovanou pro nějakou činnost. Na tomto příkladu vidíš, že instance vracené různými továrními metodami mohou dělat různé věci, i když jsou instancemi téhož rozhraní.

440. Co tam dělá ten komparátor?

Účel metody vracející komparátor

Původně jsem se domníval, že objekty virtuálního světa budou tovární instanci žádat o své kresliče a vlastní svět si od ní bude naopak brát komparátor, který mu umožní jednotlivé objekty seřadit tak, aby se ty zadní kreslily dříve a ty přední později.

Protože v každém pohledu jsou objekty jinak seřazeny, myslel jsem si, že definuji pro každý objekt jeho vlastní komparátor, který pak bude svět používat. Pak jsem ale přišel na lepší řešení (možná jenom na řešení, které se mi více líbilo), takže ten komparátor už není potřeba. Nicméně jsem jej v rozhraní ponechal, abys viděl, jak různé mohou jednotlivé instance být.

441. Jakému řešení jsi dal nakonec přednost?

Alternativní řešení

Zdálo se mi, že informace vhodné pro třídění se od jednotlivých objektů získávají příliš těžko, tak jsem nakonec doplnil rozhraní `IObjektVS` o metodu vracející aktuální kreslič (kresliče totiž nejlépe vědí, co je při daném pohledu vepředu a co vzadu) a instancím rozhraní `IKreslič` jsem přidal metodu, která vrací číslo, podle nějž je možno jednotlivé objekty pro kreslení seřadit.

Nyní tedy stačí světu jediný komparátor, který se vždy instance zeptá na její kreslič a toho pak na příslušnou hodnotu. Podle takto získaných hodnot vrátí informaci o tom, který z porovnávaných objektů bude při daném pohledu více vpředu a který více vzadu.

442. Vidím, že ta aplikace je opravdu složitá. Jak se spouští?

Jak aplikaci spustit

Podle velikosti obrazovky nastavíš ve třídě `Svět` rozměry jednoho políčka a počet políček, která budou tvořit virtuální svět, a pak už jen vytváříš objekty, které se samy iniciativně přihlašují k zobrazování ve virtuálním světě.

Hlavní program spouštějící celou aplikaci najdeš ve třídě `Klient`. V její testovací metodě můžeš zadávat jednotlivé objekty, střídat pohledy a pozorovat, jak se bude aplikace chovat.

Shrnutí – co jsme se naučili

- Návrhový vzor *Abstraktní továrna* řeší problém, kdy je třeba vybrat mezi několika sadami tříd, jejichž instance budou vystupovat v programu.
- Typickým použitím *Abstraktní továrny* je volba celkového vzhledu (*look and feel*) GUI.
- Instance vracené jednotlivými metodami mohou být jak různého, tak stejného typu.
- Při implementaci *Abstraktní továrny* se často používá vzor *Tovární metoda*.
- Jednotlivé používané *„tovární instance“* bývají definovány jako jedináčci.
- Návrhový vzor *Abstraktní továrna* neumožňuje jednoduše přidávat další rozhraní, jejichž instance budou „tovární instance“ dodávat; řešení s univerzální tovární metodou snižuje robustnost systému.
- Při implementaci *Abstraktní továrny* se často používá vzor *Prototyp*.
- Návrhový vzor *Abstraktní továrna* patří mezi vzory uvedené v GoF.

ČÁST 6

Zjednodušujeme program

V této části se soustředíme na návrhové vzory, které nám umožní zjednodušit původně příliš složitý návrh.

KAPITOLA 26

Příliš mnoho druhů tříd (Dekorátor – Decorator)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Přidává další přídavnou funkcionalitu k objektu tak, že objekt „zabalí" do jiného objektu, který má na starosti pouze přidanou funkcionalitu, a zbytek požadavků deleguje na „zabalený" objekt. Tím umožňuje přidávat funkčnost dynamicky a zavádí flexibilní alternativu k dědění.

[1] **Definice v GoF:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. – Dynamicky připojuje k objektu další odpovědnost (funkčnost). Dekorátor poskytuje flexibilní alternativu k rozšiřování funkčnosti prostřednictvím dědičnosti.

Účel

443. Přiznám se, že mne název kapitoly poněkud mate. Když je druhů tříd příliš mnoho, tak proč je mám ještě navíc nějak zdobit?

To ale není zdobení navíc. Návrhový vzor *Dekorátor* používá ono „zdobení" k tomu, aby počet tříd snížil.

Jak lze zdobením snížit počet tříd

444. Jak mohu snížit počet tříd tím, že je ozdobím?

Dekorátor použiješ tehdy, je-li velký počet požadovaných tříd důsledkem velkého množství kombinací různých vlastností, které mají dané třídy mít.

Příklad: kavárna

V knize [15] je použití dekorátoru uváděno na příkladu kavárny, v níž si můžeš objednat různé druhy kávy. Jednotlivé druhy kávy jsou přitom určeny svými přísadami. Různé kávy mají různé kombinace přísad.

Nemá smysl vytvářet pro výrobu každého druhu kávy speciální třídu. Stačí několik základních tříd pro výrobu výchozích druhů (turecká, překapávaná, instantní, bez kofeinu, …) a pak už jen výsledek zdobíš, tj. přidáváš ingredience (mléko, cukr, vaječný likér, …).

445. Počkej, počkej. Tohle jsi mi vykládal u šablonové metody.

Tehdy (viz kapitolu *Já to umím, upřesni jen detaily (Šablonová metoda – Template Method)* na straně 239) jsme řešili stejný problém jako nyní, ale zvolili jsme jinou taktiku, než jakou se ti chystám předvést teď.

446. V čem se liší?

Odlišnost dekorátoru od šablonové metody

V tom, že u šablonové metody byl předem definován nějaký algoritmus (příprava nápoje) a potomci pouze konkretizovali některé detaily tohoto základního postupu.

Naproti tomu dekorátor předpokládá, že příslušný objekt (v našem případě nápoj) již někdo vytvořil (např. za pomoci tovární metody), a tváří se, že vytváří nový objekt, který původnímu objektu dodává nějakou funkčnost nebo naopak nějakou jeho funkčnost mění tak, aby lépe odpovídala nějakým speciálním podmínkám.

V našem případě s nápoji proto definuješ pro dodatečné přidání každé ingredience speciální třídu, jejímuž konstruktoru předáš instanci, k níž je třeba ingredienci dodat. Získáš tak novou instanci s dodanou ingrediencí.

447. A nemohl bych prostě definovat metodu, která by danou ingredienci dodala?

Proč nestačí definovat novou metodu

U té kávy ti může připadat, že bychom mohli nové vlastnosti doplňovat zavoláním příslušné metody, ale v praxi většinou každá nová vlastnost citelně mění schopnosti a reakce výsledného objektu.

Dekorátor umožňuje operativně přidávat vlastnosti

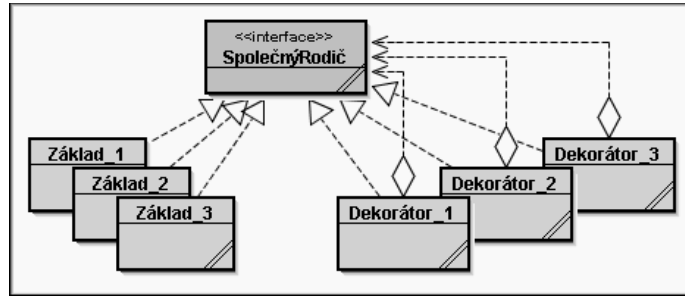
Navíc velmi často nemůžeš dopředu odhadnout, jaké všechny vlastnosti budeš v budoucnu od daného objektu vyžadovat. Dekorátor ti umožňuje kdykoliv v budoucnu doplnit nově požadované vlastnosti, aniž bys kvůli tomu musel jakkoliv měnit třídu definující výchozí objekt. Vše, na co jsou původní objekty zvyklé, zůstane zachováno a klienti s nestandardními požadavky budou používat původní objekty ozdobené dekorátorem.

448. Hm. Zkus mi to ukázat na nějakém příkladu, který je blíže praxi.

Příklad: GUI

Dekorování se používá např. u objektů, které jsou součástí GUI. Vezmi si prosté okno, které pouze zabírá nějakou oblast obrazovky. Odekoruješ-li je např. titulkovou lištou a rámečkem, musí se začít jinak počítat souřadnice objektů uvnitř okna. Odekoruješ-li je posuvníkem, změní se jeho chování ještě mnohem výrazněji.

449. Už to začínám chápat. Takže všechna okna budou mít nějakého společného rodiče, jehož instance budou u těch dekoračních tříd předávány jako parametry jejich konstruktorů – asi tak jako na obr. 26.1.



Obrázek 26.1
Struktura tříd při implementaci návrhového vzoru Dekorátor

Jak to funguje

Přesně tak. Konstruktor dekorátoru převezme dekorovaný objekt jako svůj parametr, obalí jej a „ozdobí“ svojí funkčností. Objekt dekorátoru se pak stará o věci související s dodanou funkčností a veškeré ostatní požadavky deleguje na onen obalený objekt, aby pak vrátil případné výsledky jako svoje vlastní.

450. Vidím, že dekorátor je pěkný vyčuránek. Slízne smetanu a zbytek nechá oddřít svého otroka.

Spíše bych to viděl jako symbiózu: každý z nich dělá to, co umí. Když něco umí ten druhý lépe, nechá to na něm.

Společný rodič je často třída

Ještě bych jenom dodal, že na obrázku 26.1 definuješ společného rodiče jako rozhraní, ale často je tento rodič definován jako abstraktní třída, protože toho bývá dost společného, co by mohl pro všechny své potomky implementovat.

Někdy se definuje společný rodič dekorátorů

Kromě toho bývá občas zvykem definovat společného rodiče dekoračních tříd, ale obecně to není nutné. Definici se společným rodičem dekorátorů najdeš např. v GoF a společného rodiče dekorátorů používá i standardní knihovna proudů v Javě, která je další typickou ukázkou použití tohoto návrhového vzoru.

Společný rodič dekorátorů je většinou adaptér definující implicitní verze metod

Definuješ-li společného rodiče dekorátorů, pak většinou funguje jako adaptér (viz návrhový vzor *Adaptér* na str. 261), který dekorátorům umožňuje definovat opravdu jen jejich přidanou funkčnost a předem pro ně připraví předání řízení obalenému objektu, tj. při volání některé ze svých metod zavolá stejnojmennou metodu obaleného objektu a předá jí obdržené parametry, a vrátí-li tato metoda nějaký výsledek, vrátí jej svému volajícímu programu.

Metody, jejichž prostřednictvím dekorátor realizuje jím přidanou funkčnost, doplní či překryje, ostatní metody od svého „dekoračního rodiče“ zdědí.

Příklad: knihovna proudů

Jako příklad bych mohl uvést knihovnu proudů, která se musí vypořádat s tím, že v různých situacích potřebuješ číst data z různých zdrojů a různými způsoby s nimi pracovat. Kdybychom uvažovali všechny kombinace, dostali bychom nepřehlednou paletu tříd, která by zabírala zbytečně moc místa v paměti a skoro nikdo by se v ní nevyznal.

Příklad dalších výhod

To by ale nebyly všechny nevýhody. Jak jsem říkal, programovat máš neustále tak, abys byl kdykoliv připraven program upravit. Představ si, že bys do takto pojaté knihovny potřeboval přidat další vlastnosti. Pak bys musel ke každé z dosavadních tříd přidělat „parťáka“ vybaveného danou vlastností. Šeredná představa.

Implementace

451. Přestaň už teoretizovat a ukaž mi, jak se takový dekorátor konkrétně aplikuje. Ukaž to třeba na té knihovně proudů, kterou jsi naposled zmiňoval.

Co vše musí umět knihovna proudů

Dobře. Podívej se na obr. 26.2. Na něm jsem se snažil zobrazit, co všechno by měly umět třídy, které slouží ke čtení nebo zápisu dat. Šipky na obrázku jsou sice obousměrné, ale při konkrétní operaci se pohybuješ pouze jedním směrem: při čtení shora dolů, při zápisu zdola nahoru.

A: Čtení

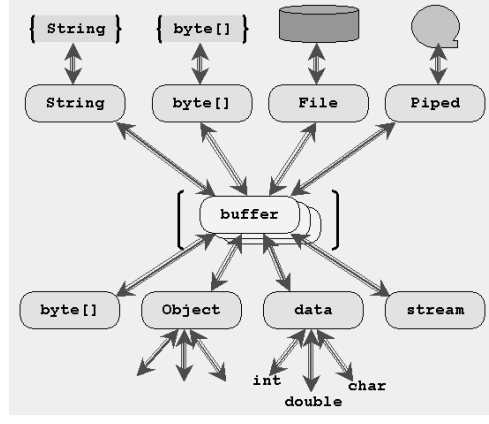
Podívejme se nejprve na čtení. Při něm načítáme data z některého ze zdrojů uvedených na obrázku 26.2 nahoře (to je jen výběr, program může používat i další zdroje dat) a potřebujeme je předat některému z cílů uvedených na obrázku dole.

– číst data z různých zdrojů

Podle situace potřebujeme číst data jednou ze souboru, jindy se nám hodí, když můžeme jako zdroj dat použít pole bajtů nebo textový řetězec, a občas potřebujeme číst data posílaná jiným programem nebo jinou částí našeho programu (zdroj dat pak označujeme jako datovod nebo „trubku“ [pipe]). Sám si jistě vymyslíš ještě další možné zdroje, které jsem již na obrázku neuváděl.

– načítat data do různých cílů

Na druhém konci toku dat je náš program a cíle, kam data směřují. S nimi je to podobné jako se zdroji: je jich celá řada. Jednou nám vyhovuje, když data přečteme do nějakého pole bajtů, ale jindy bychom je potřebovali načítat přímo do příslušných proměnných anebo prostě jenom přečtená data přeposílat dál jinému programu.



Obrázek 26.2
Koncepce čtení a zápisu dat

– pomocné operace cestou od zdroje k cíli

Při načítání či ukládání dat ale občas potřebujeme vykonávat řadu pomocných operací:

- jedná-li se o znaková data, může se nám hodit převod mezi různými kódováními,
- pracujeme-li s diskem, rádi bychom jistě využili vyrovnávací paměť, která všechny operace o několik řádů zrychlí,
- při čtení z komprimovaných souborů potřebujeme přečtená data nejprve dekomprimovat,
- … a tak bych mohl vyjmenovávat ještě dlouho.

B: Zápis

Zcela obdobné je to i se zápisem, pouze budeš číst obrázek odspodu nahoru. Data, která potřebuješ zapsat, máš připravena v různých zdrojích, potřebuješ s nimi provádět různé pomocné operace (kódovat, komprimovat, ukládat do vyrovnávacích pamětí), abys je nakonec odeslal k některému z řady možných cílů.

Násobení počtu tříd

Když bys řešil problém klasicky, potřeboval bys

$$počet\text{-}zdrojů \times počet\text{-}možných\text{-}kombinací\text{-}mezioperací \times počet\text{-}cílů$$

různých tříd, protože bys pro každou kombinaci zdroj-(posloupnost mezioperací)-cíl potřeboval speciální třídu: jednu, která by četla ze souboru a ukládala do proměnné, další, která by četla ze souboru s využitím vyrovnávací paměti a ukládala do proměnné, další, která by četla ze souboru s využitím vyrovnávací paměti a aplikací nějaké filtrace a ukládala výsledek do proměnné, další, která by opět četla ze souboru, ale ukládala pro změnu do pole bajtů, a tak bych mohl vyjmenovávat dlouho.

452. Tak už nevyjmenovávej a přejdi k té implementaci.

Dva univerzální rodiče: vstup × výstup

Java definuje dva univerzální rodiče: třídu `java.io.InputStream`, která je rodičem všech tříd určených pro čtení dat, a třídu `java.io.OutputStream`, která je rodičem všech tříd určených pro zápis.

Základní třídy přímo komunikují se zdroji a cíli dat

Základní třídy jsou zastoupeny třídami, které umějí přímo komunikovat s nějakým zdrojem nebo cílem dat. Omezím-li se na vstupní proudy, tak jsou to třídy (název balíčku budu vynechávat, všechny najdeš tamtéž) `ByteArrayInputStream`, která čte z pole bajtů, `FileInputStream` pro čtení ze souborů spravovaných operačním systémem a `PipedInputStream` pro čtení dat posílaných jinou částí programu, nejčastěji jiným vláknem.

Dekorátory dodávají:

Tyto třídy ti sice data načtou, ale takto načtená data ti z nejrůznějších důvodů nemusí vyhovovat.

– čtení s bufferem

Je-li čtení pomalé, mohlo by pomoci použití vyrovnávací paměti – dekoruješ vstupní proud tak, že vytvoříš novou instanci proudu `BufferedInputStream` a odkaz na původní proud předáš konstruktoru jako parametr. Získáš tak proud „ozdobený“ vyrovnávací pamětí.

– přímé čtení dat

Nelíbí-li se ti, že načítáš pouze bajty, zatímco ty bys chtěl číst přímo hodnoty načítaných proměnných? Dekoruj proud tak, že vytvoříš instanci třídy `DataInputStream`, jejímuž konstruktoru předáš odkaz na svůj dosavadní (možná již dekorovaný) proud. Získáš tak proud „ozdobený“ schopností převádět posloupnost bajtů na konkrétní data.

Podobně i znakové proudy

Obdobné by to bylo s výstupními proudy a naprosto stejné by to bylo i se znakovými proudy, které jsou potomky tříd `Reader` a `Writer`.

Je na tobě, zda budeš vše deklarovat v jediném příkazu anebo si celou operaci rozdělíš na kroky. Při rozdělené deklaraci bys bufferovaný výstup do souboru deklaroval např.:

```
FileWriter     fw = new FileWriter     ( "C:/výstup.txt" );
BufferedWriter bw = new BufferedWriter ( fw );
```

Naproti tomu při jednorázové deklaraci bys tentýž výstup deklaroval např.:

```
BufferedWriter bos = new BufferedWriter (
                         new FileWriter ("C:/výstup.txt") );
```

Je-li dekorátorů více, může záležet na tom, v jakém pořadí je vzájemně zanořuješ. Špatné pořadí zanoření může způsobit, že se jednotlivá rozšíření navzájem „pohádají“. Musíš si vždy přečíst dokumentaci, aby ses pak nedivil, proč se ti program chová tak divně.

453. Takže instance dekoračních tříd mají dekorovanou instanci definovanou jako svůj atribut, a kdykoliv se po nich chce např. nějaké čtení, požádají o načtení tento atribut, načtená data po svém zpracují a předají volající metodě?

Vidím, že bys to mohl vykládat místo mne.

454. A jak je to s implementací jednotlivých metod. To musí každý dekorátor definovat všechny metody znovu, aby příslušný požadavek předal svému atributu? Dokážu si představit, že v řadě případů požadavek opravdu jen předá a vrátí obdržený výsledek. To mi pak připadá jako zbytečně moc psaní.

Rodiče dekorátorů

Máš pravdu. Proto také proudy definují třídy `FilterInputStream`, resp. `FilterOutputStream`, resp. `FilterReader`, resp. `FilterWriter`, které fungují jako adaptéry: nabízejí předpřipravený atribut a předdefinované verze metod, které pouze předají požadavek svému atributu a volající metodě předají obdržený výsledek.

Potomci těchto tříd pak mohou překrýt pouze ty metody, které musí obdržená data ještě nějakým způsobem zpracovávat. Jsou to právě ty adaptéry, o nichž jsem hovořil v odpovědi na otázku 450.

Příklad

455. Řekl bych, že je nejvyšší čas si nějaký takový dekorátor naprogramovat.

Mám tu pro tebe hned několik příkladů dekorátorů textových proudů. Konstruktoru prvního z nich zadáš jako parametry proudy, do nichž má posílat výstup, a vše, co do něj pošleš, přepošle do každého ze zadaných proudů. Je to tedy vlastně takový násobič výstupu, který využiješ např. tehdy, budeš-li chtít poslat data nejenom na standardní výstup, ale také třeba do souboru. Jeho zdrojový text najdeš ve výpisu 26.1.

`MultiWriter` zapisuje do více proudů současně

Protože `MultiWriter` zapisuje ne do jednoho, ale hned do několika proudů, nemůže být definován jako potomek třídy `FilterWriter` (přesněji mohl by být, ale protože by musel překrýt všechny metody, nemělo by to smysl). Musí tedy definovat všechny potřebné

metody sám (ono jich naštěstí u znakových proudů tolik není – u bajtových proudů by to bylo horší). Další příklady však již výhod schopného předka využít mohou.

ASCIIWriter
převádí na 7bitový java-kód

První z nich je třída `ASCIIWriter` (její zdrojový kód najdeš ve výpisu 26.2), jejíž instance převádí zapisované znaky do sedmibitového kódu, v němž jsou všechny znaky s kódem vyšším než 127 převáděny na posloupnost `\uXXXX`, kde znaky `X` zastupují hexadecimální číslice.

Tato konverze je ve standardní knihovně definována na několika místech. Autoři jednotlivých tříd se však mezi sebou nedohodli, takže každý z nich definoval potřebnou konverzi ve svých třídách po svém.

ASCIIReader
čte 7bitový java-kód

Posledním z ukázkových dekorátorů je třída `ASCIIReader` (její zdrojový kód najdeš ve výpisu 26.3), která naopak čte ze „sedmibitového" proudu a převádí konvertované znaky zpět na jeden znak s příslušným kódem.

Uvedená dvojice dekorátorů realizuje konverze, o něž by bylo třeba požádat program `native2ascii.exe`, který je sice součástí JDK, ale přímo z programu se vyvolává špatně. Předvedené dekorátory však pracují pouze se znaky, které je možné zakódovat prostřednictvím 16 bitů. Nebude proto fungovat s rozšířenou sadou znaků, podporovanou od verze 5.0[1].

ASCIIRWTest
demonstruje použití

Aby ses mohl také podívat, jak se takové dekorátory používají, najdeš ve výpisu 26.4 definici třídy `ASCIIRWTest`, která testuje funkčnost obou dekorátorů.

Výpis 26.1: Třída `MultiWriter` přeposílající vystupující znaky do zadané skupiny výstupních znakových proudů

```
package rup.cesky.vzory._26_dekorátor;

import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.StringWriter;
import java.io.Writer;

/*******************************************************************************
 * Instance třídy MultiWriter představují výstupní proudy,
 * které posílají vystupující data synchronně na několik míst.
 *
 * @author    Rudolf PECINOVSKÝ
 * @version   0.00.000,  0.0.2006
 */
public class MultiWriter extends Writer
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Seznam proudů, do nichž budou posílána vystupující data. */
    private Writer[] writers;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================
```

[1] Podrobnosti o této rozšířené sadě znaků a její podpoře se můžeš dočíst v [31].

```
    /*****************************************************************************
     * Vytvoří nový násobný vystupující proud přeposílající vystupující data
     * do všech proudů zadaných jako parametry.
     * @param writers  Seznam proudů, do nichž budou synchronně zasílána
     *                 vystupující data.
     */
    public MultiWriter( Writer... writers ) {
        this.writers = writers.clone();
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Zapíše na výstup znak odpovídající spodním 16 bitům zadaného kódu.
     * @param c    16bitový kód zapisovaného znaku
     * @exception  IOException  Nastane-li nějaká chyba
     */
    @Override
    public void write( int c ) throws IOException {
        for( Writer w : writers )
            w.write(  c  );
    }


    /*****************************************************************************
     * Zapíše na výstup zadanou část zadaného pole znaků.
     *
     * @param  cbuf  Pole znaků se zapisovanými znaky
     * @param  off   Index prvního zapisovaného znaku
     * @param  len   Počet zapisovaných znaků
     * @exception    IOException  Nastane-li nějaká chyba
     */
    @Override
    public void write(char[] cbuf, int off, int len) throws IOException {
        for( Writer w : writers )
            w.write(  cbuf, off, len );
    }


    /*****************************************************************************
     * Zapíše na výstup zadanou část zadaného řetězce.
     * @param  str   Řetězec se zapisovanými znaky
     * @param  off   Index prvního zapisovaného znaku
     * @param  len   Počet zapisovaných znaků
     * @exception    IOException  Nastane-li nějaká chyba
     */
    @Override
    public void write(String str, int off, int len) throws IOException{
        for( Writer w : writers )
            w.write(  str, off, len );
    }


    /*****************************************************************************
     * Spláchne proud.
     * @exception    IOException  Nastane-li nějaká chyba
```

```
     */
    @Override
    public void flush() throws IOException {
            for( Writer w : writers )
                w.flush();
    }


    /**************************************************************************
     * Zavře proud.
     * @exception    IOException  Nastane-li nějaká chyba
     */
    @Override
    public void close() throws IOException {
        for( Writer w : writers )
            w.close();
    }

//== TESTY A METODA MAIN ======================================================

    /**************************************************************************
     * Testovací metoda.
     * @throws IOException
     */
     public static void test() throws IOException {
         Writer sw = new StringWriter(),
                os = new OutputStreamWriter( System.out ),
                fw = new FileWriter( "Y:/SMAZAT.txt");

         MultiWriter mw = new MultiWriter( sw, os, fw );
         mw.write( "Pokusný text posílaný do mnoha stran" );
         mw.flush();
         System.out.println("\n===");
         System.out.println("SW: " + sw );
         //Pozor se zavíráním, máme-li v seznamu standardní výstup.
         mw.close();
     }
     /**************************************************************************
      * @param args Parametry příkazového řádku - nepoužívají se
      * @throws IOException */
     public static void main( String[] args ) throws IOException  {  test();  }
}
```

Výpis 26.2: Dekorátor `ASCIIWriter` převádí vystupující znaky do sedmibitového java-kódu

```
package rup.česky.vzory._26_dekorátor;

import java.io.FilterWriter;
import java.io.IOException;
import java.io.Writer;

/*******************************************************************************
 * Instance třídy ASCIIWriter slouží k převodu libovolných znaků
 * na sedmibitový javový unicode, v němž znaky s kódem větším než 127 zapisují
 * jako posloupnost znaků &#92;uABCD, kde ABCD je čtyřmístný kód daného znaku
 * zapsaný v šestnáctkové soustavě.
 * Třída řeší pouze znaky dvoubajtového unicodu.
```

```
 * <p>
 * V případě použití bufferovaného výstupu musí být instance této třídy
 * definovány jako obálka bufferovaného writeru,
 * protože při převodu rozbíjí případné bufferování.
 * <p>
 * V zájmu zvýšené efektivity třída překrývá i metody {@link #write(int)}
 * a {@link #write(String,int,int)}, které by překrývat nemusela. Rodičovská
 * třída je však obě implementuje prostřednictvím volání metody
 * {@link #write(char[],int,int)}, což je zbytečně neefektivní.
*/
public class ASCIIWriter extends FilterWriter
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Sada speciálních znaků, před něž má být vkládáno zpětné lomítko. */
    private String speciální = "\\";


//== NESOUKROMÉ METODY TŘÍDY ===================================================

//##############################################################################
//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří konverzní writer, který převádí znaky s kódem nad 128
     * na příslušné javové unikódové sekvence typu &#92;uABCD,
     * kde ABCD je čtyřmístný kód daného znaku zapsaný v šestnáctkové soustavě.
     * @param writer obalovaný writer
     */
    public ASCIIWriter( Writer writer ) {
        super( writer );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Nastaví sadu znaků, před nimiž se bude ignorovat zpětné lomítko.
     * @param speciální Sada znaků zadaná jako řetězec
     */
    public void setSpeciální( String speciální ) {
        this.speciální = "\\" + speciální;
    }


    /*****************************************************************************
     * Vrátí sadu znaků, před nimiž se bude ignorovat zpětné lomítko.
     * @return speciální Řetězec se sadou znaků
     */
    public String getSpeciální() {
        return this.speciální;
    }


    /*****************************************************************************
     * Zapíše na výstup znak odpovídající spodním 16 bitům zadaného kódu.
     * @param c    16bitový kód zapisovaného znaku
     * @exception  IOException  Nastane-li nějaká chyba
```

```
     */
    @Override
    public void write( int c ) throws IOException {
        převeď( c );
    }


    /*************************************************************************
     * Zapíše na výstup zadanou část zadaného pole znaků.
     * @param  cbuf  Pole znaků se zapisovanými znaky
     * @param  off   Index prvního zapisovaného znaku
     * @param  len   Počet zapisovaných znaků
     * @exception    IOException  Nastane-li nějaká chyba
     */
    @Override
    public void write(char[] cbuf, int off, int len) throws IOException {
        for( int i=off;   i < off+len;   i++ )
            převeď( cbuf[i] );
    }


    /*************************************************************************
     * Zapíše na výstup zadanou část zadaného řetězce.
     * @param  str   Řetězec se zapisovanými znaky
     * @param  off   Index prvního zapisovaného znaku
     * @param  len   Počet zapisovaných znaků
     * @exception    IOException  Nastane-li nějaká chyba
     */
    @Override
    public void write(String str, int off, int len) throws IOException {
        int SL = off+len;
        for( int i=off;   i < SL;   i++ )
            převeď( str.charAt( i ) );
    }


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ========================================

    /*************************************************************************
     * Převede zadaný znak do 7bitového kódování a zapíše jej prostřednictvím
     * obaleného writeru. Při převodu se všechny znaky s kódem menším než 128
     * jednoduše zapíší a místo znaků s kódem větším než 127 se zapíše
     * posloupnost znaků &amp;#92;uABCD, kde ABCD je čtyřmístný kód daného znaku
     * zapsaný v šestnáctkové soustavě.
     * @param c   Převáděný znak
     * @exception  IOException  Dojde-li při zápisu k jakékoliv chybě
     */
    private void převeď( int c ) throws IOException {
        if( c < 128 ) {
            //Prvních 128 znaků pouze kopíruje na výstup
            if( speciální.indexOf( (char)c ) > 0 )
                //Výjimkou jsou speciální znaky, před něž vloží zpětné lomítko
                out.write( '\\' );
            out.write( c );
        } else {
            //Zbylé znaky převede na příslušné 6znakové posloupnosti
            char[] cc = new char[6];
            //První dva znaky budou zpětné lomítko a písmeno u
```

```
            cc[0] = '\\';
            cc[1] = 'u';
            //Zbylé čtyři znaky jsou šestnáctkový kód znaku
            for( int i=5;   i > 1;   i– ) {
                cc[i] = Character.forDigit( c%16, 16 );
                c /= 16;
            }
            //Pošlu na výstup celou šestici znaků najednou
            out.write( cc );
        }
    }
}
```

Výpis 26.3: Dekorátor `ASCIIReader` převádí znaky ze 7bitového java-kódu do kódování Unicode

```
package rup.česky.vzory._26_dekorátor;

import java.io.FilterReader;
import java.io.IOException;
import java.io.Reader;


/*******************************************************************************
 * Instance třídy ASCIIReader slouží k převodu libovolných znaků
 * zapsaných v sedmibitovém javovém unicodu, tj. znaků zapsaných ve tvaru
 * &amp;#92;uABCD, kde kde ABCD je čtyřmístný kód daného znaku
 * zapsaný v šestnáctkové soustavě, na normální znaky.
 * <p>
 * Převod řeší pouze znaky dvoubajtového unicodu, takže kódy znaků tříbajtového
 * unicodu převede na odpovídající dva pseudoznaky dvoubajtového unicodu.
 * <p>
 * Dvouznakové sekvence začínající zpětným lomítkem nenásledovaným znakem
 * <b>u</b> se při převodu nahradí znakem následujícím za zpětným lomítkem.
 * <p>
 * V případě použití bufferovaného vstupu musí být instance této třídy
 * definovány jako obálka bufferovaného readeru,
 * protože při převodu rozbíjí případné bufferování.
*/
public class ASCIIReader extends FilterReader
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Hodnota signalizující,
     * že při předchozím čtení nebylo přečteno zpětné lomítko. */
    private static final int NIC = -2;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ===============================================

    /** Znak přečtený po zpětném lomítku.  */
    private int poLomítku = NIC;

    /** Sada speciálních znaků, před nimiž má být zpětné lomítko. */
    private String speciální = "\\";
```

```
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /**************************************************************************
     * Vytvoří konverzní reader.
     * @param reader Obalovaný reader
     */
    public ASCIIReader( Reader reader )  {
        super( reader );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Nastaví sadu znaků, před nimiž se bude ignorovat zpětné lomítko.
     * @param speciální Sada znaků zadaná jako řetězec
     */
    public void setSpeciální( String speciální ) {
        this.speciální = "\\" + speciální;
    }


    /**************************************************************************
     * Vrátí sadu znaků, před nimiž se bude ignorovat zpětné lomítko.
     * @return speciální Řetězec se sadou znaků
     */
    public String getSpeciální() {
        return this.speciální;
    }


    /**************************************************************************
     * Přečte znak
     * @return     Kód přečteného znaku nebo -1 v případě, že už není co číst
     * @exception  IOException  Nastala-li nějaká vstupně-výstupní chyba
     */
    @Override
    public int read() throws IOException {
        return přečtiZnak();
    }


    /**************************************************************************
     * Přečte znaky do zadané části pole. Zablokuje své vlákno až do doby,
     * dokud nebudou všechny znaky načteny nebo dokud nebude dosaženo
     * konce proudu.
     * @param   cbuf  Cílové pole znaků
     * @param   off   Index prvního zapisovaného znaku
     * @param   len   Maximální počet znaků k načtení
     *
     * @return  Počet skutečně přečtených znaků nebo -1, bylo-li dosaženo
     *          konce proudu před načtením prvního znaku.
     * @exception  IOException  Nastala-li nějaká vstupně-výstupní chyba
     */
    @Override
    public int read(char[] cbuf, int off, int len) throws IOException {
        for( int i=off;   i < off+len;   i++ ) {
            int znak = přečtiZnak();
```

```
            if( znak < 0 )  {
                //vstupní proud skončil
                int přečteno = i-off;    //Počet přečtených znaků
                //Nepřečetl-li jsem nic, musím vrátit -1
                return (přečteno == 0)  ?  -1  :  přečteno;   //==========>
            }
            cbuf[i] = (char)znak;
        }
        return len;
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /**************************************************************************
     * Metoda přečte ze vstupního proudu další znak a vrátí jeho kód.
     * Rozpozná, kdy je znak zakódován jako posloupnost &#92;uABCD,
     * kde ABCD je čtyřmístný kód daného znaku - v takovém případě
     * vrátí zakódovaný znak.
     * Přečte-li zdvojené zpětné lomítko, vrátí jedno zpětné lomítko.
     *
     * @return Kód přečteného znaku
     * @throws IOException  Dojde-li při čtení k nějaké chybě
     */
    private int přečtiZnak() throws IOException  {
        //Byl-li minule přečten dopředu znak za lomítkem, nebudu nyní číst
        //a vrátím onen znak za zpětným lomítkem.
        if( poLomítku != NIC ) {
            //Při minulém čtení bylo přečteno zpětné lomítko a další znak
            //Vrátím onen zapamatovaný další znak
            int ret = poLomítku;
            poLomítku = NIC;    //Nyní již zpětné lomítko přečtené nebylo
            return ret;         //==========>
        }
        //Nebylo - přečtu další znak
        int c = in.read();
        if( c == '\\' ) {
            //Přečetli jsme zpětné lomítko - je třeba zjistit, co je za ním
            int c2 = in.read();
            if( c2 == 'u' ) {
                //Za lomítkem je znak u => následuje 4místný kód znaku
                for( int i=0;   i < 4;   i++ ) {
                    c = 16*c + Character.digit( in.read(), 16 );
                }
                return( c );   //==========>
            } else if( speciální.indexOf( c2 ) >= 0 ) {
                //Za zpětným lomítkem je speciální znak => vrátím jej
                return c2;   //==========>
            } else {
                //Následoval jiný znak - při příštím čtení jej vrátím
                poLomítku = c2;
                //Nyní vracím zpětné lomítko
                return '\\';   //==========>
            }
        } else
            return c;
    }
}
```

Výpis 26.4: Definice třídy `ASCIIRWTest` sloužící k otestování dekorátorů pracujících se 7bitovým java-kódem

```
package rup.česky.vzory._26_dekorátor;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.StringReader;
import java.io.StringWriter;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;


/*****************************************************************************
* Třída ASCIIRWTest slouží k testování dekorátorů ASCIIWriter a ASCIIReader.
 */
public class ASCIIRWTest
{
//== KONSTRUKTORY A TOVÁRNÍ METODY ==========================================
    private ASCIIRWTest()  {}

//== TESTY ==================================================================

    private static final String TEXT =
                                ": Příliš žluťoučký kůň úpěl ďábelské ódy\n";

    /*************************************************************************
     * Přečte soubor NázevTřídy.ctxt
     * a zapíše přečtený obsah na standardní výstup.
     *
     * @throws IOException Pokud se při čtení vyskytne chyba
     */
    private static String přečtiCtxt( Class třída, String název, String spec )
            throws IOException
    {
        InputStream is = třída.getResourceAsStream( název );
        if( is == null )
            throw new RuntimeException( "Ve složce s class-souborem třídy " +
                třída + " není připraven soubor " + název );
        ASCIIReader abfr = new ASCIIReader(
                                new BufferedReader(
                                    new InputStreamReader(
                                        třída.getResourceAsStream(název) )));
//        Reader         reader = new InputStreamReader( is );
//        BufferedReader bfr    = new BufferedReader( reader );
//        ASCIIReader    abfr   = new ASCIIReader( bfr );
        System.out.println("===== Název čteného souboru: " + název);
        System.out.println("      Speciální znaky: \"" + spec + "\"");
        abfr.setSpeciální( spec );
        int znak;
        StringBuilder sb = new StringBuilder();
        while( (znak = abfr.read()) >= 0 ) {
```

```
            sb.append( (char)znak );
            System.out.print( (char)znak );
        }
        abfr.close();
        System.out.println( "\n===== Konec čtení ctxt souboru\n" );
        return sb.toString();
    }


    /*****************************************************************************
     * Přečte soubor NázevTřídy.ctxt
     * a zapíše přečtený obsah na standardní výstup.
     * @throws URISyntaxException
     * @throws IOException
     *
     * @throws IOException Pokud se při čtení vyskytne chyba
     */
    private static void zapišCtxt( String vstup,
            Class třída, String název, String spec )
            throws URISyntaxException, IOException
    {
        URL url = třída.getResource( "" );
        System.out.println( "===== URL: " + url );
        URI uri = url.toURI();
        File složka = new File( uri );
        System.out.println( "      Výstupní složka: " + složka );

        String přípona;
        if( (spec != null)  &&  !spec.equals( "" ) )
            přípona = "CZ+";
        else
            přípona = "CZ";
        File výstup = new File( složka, název + přípona );
        System.out.println( "      Výstupní soubor: " + výstup );
        System.out.println( "      Speciální znaky: \"" + spec + "\"" );

        //Následuje zápis konvertovaného ctxt souboru
        StringReader   sr   = new StringReader( vstup );
        BufferedWriter bfw  = new BufferedWriter( new FileWriter( výstup ) );
        ASCIIWriter    abfw = new ASCIIWriter( bfw );
        abfw.setSpeciální( spec );
        int znak;
        try {
            while( (znak = sr.read()) >= 0 ) {
                System.out.print( (char)znak );
                abfw.write( (char)znak );
            }
        } finally {
            bfw.close();
        }
        System.out.println( "\n===== Konec zápisu konvertovaného souboru\n" );
    }

    /*****************************************************************************
    * Testovací metoda slouží zároveň jako demonstrace možnosti použití
    * instancí této třídy.
    * @throws Exception Nepodaří-li se IO operace
    */
```

```
    public static void test()  throws Exception {
        //Nejprve budeme číst ze Stringu, do nějž zapíšeme konverzním writerem
        StringWriter sw = new StringWriter();
        sw.write( "\\1.Původní" + TEXT );
        ASCIIWriter   aw = new ASCIIWriter( sw );
        aw.write( "\\2.Konvert" + TEXT );
        aw.close();
        System.out.println("\fBudeme číst následující text:\n" +
            sw.toString() + "\n===== Přečteno:\n" );

        StringReader sr;
        ASCIIReader  ar;

        //Čtení po znaku
        sr = new StringReader( sw.toString() );
        ar = new ASCIIReader( sr );
        int znak;
        while( (znak = ar.read()) >= 0 ) {
            System.out.print( (char)znak );
        }
        ar.close();
        System.out.println( "\n===== Konec čtení z řetězce\n" );

        //Čtení do pole
        char[] cc = new char[100];
        sr = new StringReader( sw.toString() );
        ar = new ASCIIReader( sr );
        int počet;
        while( (počet = ar.read( cc, 0, cc.length )) > 0 ) {
            for( int i=0;   i < počet;   System.out.print( cc[i++] ) );
            System.out.println( "==#==" );
        }
        ar.close();
        System.out.println( "\n===== Konec čtení do pole\n" );

        Class třída = ASCIIRWTest.class;
        String název = třída.getSimpleName() + ".ctxt";

        //Přečte a při čtení vytiskne obsah vlastního ctxt souboru
        //Poté jej zapíše do souboru a paralelně tiskne, co zapisuje
        String přečteno;
        přečteno = přečtiCtxt( třída, název, "" );

        přečteno = přečtiCtxt( třída, název, ": =");
        zapišCtxt( přečteno,   třída, název, "" );

        přečteno = přečtiCtxt( třída, název, ": =");
        zapišCtxt( přečteno,   třída, název, ": =");

    }
    /*****************************************************************************
     * @param args Parametry příkazového řádku - nepoužívají se
     * @throws Exception Nepodaří-li se IO operace
     */
    public static void main (String[] args)   throws Exception
    { test(); }
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Dekorátor* umožňuje snížit kombinatoricky rostoucí počet tříd.
- Každá dodaná funkčnost je implementována jako „ozdobení“ (dekorování) jiného objektu.
- Základní třídy i dekorátory mají společného rodiče.
- Pro každé rozšíření funkčnosti je definována samostatná třída – dekorátor.
- Konstruktor dekorátoru přebírá dekorovaný objekt jako parametr a „obalí“ jej vlastními schopnostmi.
- Dekorátor většinou přidává metody, které se starají o jím přidanou funkčnost. Může ale také pouze překrýt metody svých předků a definovat je pro daný účel efektivněji.
- Požadavky nesouvisející s dodatečnou funkčností předá dekorátor dekorovanému objektu a vrátí obdržený výsledek.
- Dekorátory mají někdy ještě vlastního rodiče se společnou implementací. Ten pak bývá koncipován jako adaptér nabízející implicitní funkčnost: delegování metod na obalený objekt a vracení obdržených výsledků.
- Standardní knihovna používá dekorátory např. v knihovně proudů a v knihovnách pro tvorbu GUI.
- Návrhový vzor *Dekorátor* patří mezi vzory uvedené v GoF.

KAPITOLA 27

Horký brambor (Řetěz odpovědnosti – Chain of Responsibility)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Umožňuje, aby zaslaný požadavek zpracoval jiný objekt než ten, kterému byl zadán. Doporučuje objekty zřetězit tak, aby objekt, který není schopen požadavek zpracovat, mohl požadavek předat dalšímu objektu v řetězu.

[1] **Definice v GoF:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. – Umožňuje více objektům zpracovat požadavek, čímž se vyhne svázání odesilatele požadavku s jeho adresátem. Zřetězí cílové objekty, které posílají požadavek dále řetězem, dokud není zpracován.

Účel

456. Po dlouhé době se mi zdá, že již ze stručné charakteristiky tuším, o co by mohlo v daném návrhovém vzoru jít. Není mi ale zcela jasné, k čemu může být tento návrhový vzor dobrý.

Minimalizace provázanosti

Jednou ze zásad, o kterých jsme si na počátku říkali, bylo, že je vhodné minimalizovat vzájemnou provázanost (závislost) jednotlivých tříd a jejich instancí. Uvolnění této vazby zprostředkovává např. rozhraní (`interface`), protože obracíš-li se na instanci rozhraní, nemáš většinou představu o tom, kdo by se za ním mohl skrývat.

Někdy se špatně určuje, komu zprávu poslat

Stále tu ale zůstává nutnost vědět, komu svoji zprávu posíláš. Správně bys ji měl poslat tomu, kdo na ni bude reagovat. To ale jde v některých situacích velice těžko odhadnout.

Příklad: nápověda

Vezmi si třeba nápovědu k aplikaci. O nápovědu se typicky žádá stiskem klávesy **F1**. Jenomže koho o ni požádáš, tj. komu má systém zaslat tvoji zprávu?

457. Přece aplikaci.

To by nebylo nejlepší řešení. Nápověda aplikace je rozsáhlá a ty většinou nechceš nápovědu k celé aplikaci, ale pouze k tomu prvku, na který právě ukazuješ (asi jsem měl varovat, že hovořím o kontextové nápovědě).

458. V tom případě ji pošlu tomu prvku, který má zrovna fokus, tj. prvku, na který ukazuji nebo v němž se zrovna pohybuji.

Aktivní prvek nebývá ten, který umí odpovědět

Co když ale tento prvek žádnou konkrétní nápovědu přiřazenu nemá? Představ si, že jsi otevřel dialogové okno a uvědomil sis, že vlastně přesně nevíš, k čemu jednotlivé volby slouží. Zmáčkneš proto **F1** a požadavek se předá tlačítku **OK**, protože to má právě fokus. Jenomže k tomuto tlačítku většinou žádná nápověda přiřazena není.

459. No jo, to bych se pak nic nedozvěděl.

Řešení: zavedení řetězu odpovědnosti

Právě. Tohle je právě situace, kdy je vhodné použít návrhový vzor *Řetěz odpovědnosti*. Tlačítko `OK` nemá přiřazenu žádnou nápovědu, tak požadavek přehraje na další komponentu v řetězu, kterou bude např. karta v dialogovém okně. Ta buď umí zobrazit nápovědu, nebo opět přehraje požadavek dál – nejspíš celému dialogovému oknu. No a kdybys zapomněl vybavit dialogové okno nápovědou, tak to přehraje tvůj požadavek na celou aplikaci, která už ti snad něco přece jenom prozradí.

460. No jo, to je chytré. A používá se to i jinde než u dialogových oken a jejich nápovědy?

Aplikace ve stromových strukturách

S aplikací tohoto návrhového vzoru se velmi často setkáš ve stromových strukturách. Když se nějakého uzlu na něco zeptáš a on ti nebude umět odpovědět, tak přehraje tvůj požadavek na svůj rodičovský uzel, nebude-li to vědět ani on, přehraje to opět na svého rodiče a tak to poběží dál, až to v případě neschopnosti kohokoliv odpovědět skončí někde v kořeni celého stromu.

Osvobozuje žadatele od nutnosti znát adresáta

Tady bych se vrátil k otázce uvolnění vazeb, o níž byla zmínka na začátku. Použitím návrhového vzoru *Řetěz odpovědnosti* osvobozuješ žadatele od nutnosti vědět, komu má správně svoji žádost poslat. Prostě ji pošle někomu, kdo bude zrovna po ruce,

a bude očekávat, že nebude-li oslovený objekt znát odpověď, bude umět zařídit, aby se tvůj požadavek dostal k někomu, kdo jej zodpoví.

Možnost dynamické změny odpovědnosti

Navíc můžeš při použití tohoto vzoru měnit odpovědnost za správnou reakci za chodu programu. Prostě se najednou objeví někdo, kdo už ví, co odpovědět, takže zprávu nebude posílat dál a prostě ji zodpoví.

Na druhou stranu by měl být tazatel připraven i na to, že se oslovený objekt nebude umět se svými následníky dohodnout (tj. nenajde se nikdo, kdo by uměl zprávu zpracovat), a zpráva zůstane nakonec nezpracována.

461. Trochu mi to připomíná mechanismus zpracování výjimek.

Je to něco podobného – i tam metoda vyhodí výjimku, aniž by nutně věděla, kdo ji zachytí. Hlavní rozdíl je přitom ten, že ono zřetězení, tj. určení, komu nezpracovanou výjimku metoda předá, nedefinuje programátor přímo, ale zabezpečuje je virtuální stroj na základě informací získaných ze zásobníku návratových adres.

462. Umím si zdůvodnit, proč je v názvu návrhového vzoru slovo *řetěz*. Proč ale řetěz odpovědnosti?

Proč je v názvu *řetěz odpovědnosti*

Protože instance, kterou oslovíš, je zodpovědná za to, že na tvoji zprávu správně zareaguje. Ona ale tuto svoji odpovědnost přehraje na další instanci v řetězu navzájem na sebe napojených instancí.

Implementace

463. Jestli jsem to dobře pochopil, tak jednotlivé objekty v řetězu nemusí být stejného typu. Stačí, když všechny implementují rozhraní deklarující zadanou zprávu.

Instance v řetězu mohou být různých typů

Bystrý kluk. Přesně tak. V diagramu tříd může být i řada různých typů objektů – pak je ovšem vhodné aplikovat vzor *Strom* (viz kapitolu *Bloudění strukturou (Strom – Composite)* na straně 271). Každý objekt totiž musí znát svého následníka v řetězu, kterému bude přeposílat zprávu, na niž nebude znát odpověď.

Tato propojení jednotlivých členů řetězu mohou být definovaná jednoúčelově právě kvůli optimálnímu zpracování příslušné zprávy. V řadě situací už ale má struktura jakési přirozené vazby, které můžeme při implementaci vzoru využít.

464. Jaké vazby?

Přirozené vazby ve struktuře

Například ve stromu bývá zvykem, že každý uzel zná svého rodiče. Pošle-li mu proto někdo zprávu, na kterou nebude umět reagovat, může se z potíží vylhat tím, že správu přepošle svému rodiči v blahé naději, že ten bude chytřejší.

Nemusí to být strom

Strukturou, kterou bloudí požadavek ke zpracování, ale nemusí být vždy strom. Může jí být stejně dobře i seznam nebo nějaká jiná datová struktura, jejíž prvky uchovávají vzájemné odkazy na své kolegy, kterým budou přeposílat zprávy, na jejichž zpracování samy nestačí.

Pokud ale v použité struktuře žádné přirozené vazby nejsou, budeš v ní muset potřebné vazby definovat sám.

Příklad

465. Neměl bys nějaký příkládek na použití tohoto návrhového vzoru?

Ukážu ti takový AHA-příklad. S použitím tohoto návrhového vzoru v trochu praktičtějším programu si počkej, až si budeme povídat o návrhovém vzoru *Interpret*. V závěrečném příkladu v podkapitole *Příklad: Aritmetické výrazy* na straně 488 se tento vzor používá pro substituce proměnných výrazem. To se ti bude určitě líbit.

Popis úlohy

Vraťme se ale k našemu AHA-příkladu. Představ si, že máš řadu úloh, které postupně vznikají a je třeba je řešit. Každé úloze přidělíš jeden proces, který bude mít její vyřešení na starosti.

Procesy jsou ale drahá záležitost a především jejich vytváření je pomalé a náročné. Proto je výhodnější, když nemusíš vytvářet procesy stále nové, ale pokud už nějaký proces vytvoříš, tak jej po použití ponecháš připravený pro příště.

O těchto možnostech jsme se bavili již v kapitole *Konečný počet instancí (Fond – Pool)* na straně 151. Nyní si ukážeme dvě náhražky fondu. Místo samostatného fondu použijeme náhradní řešení přímo v programu.

Podívej se nejprve na ony symbolické úlohy. Každá úloha ví, jak se jmenuje a jak je obtížná, přičemž obtížnost úlohy je současně časem, který bude třeba k jejímu vyřešení.

Výpis 27.1: Definice třídy `Úloha`, jejíž instance představují symbolické úlohy, které je třeba vyřešit

```
package rup.česky.vzory._27_řetěz;

import java.util.Random;
import rup.česky.společně.IO;
import rup.česky.společně.SynchroTisk;


/*******************************************************************************
 * Instance třídy <code>Úloha</code> představují úlohy,
 * které budou zadávány procesům k řešení.
 * Každá úloha má svoji obtížnost, která říká,
 * jak dlouho bude procesu trvat její zpracování.
 */
public class Úloha
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Prodleva mezi vygenerováním jednotlivých úloh. */
    private static final int PRODLEVA = 200;

    /** Největší možná obtížnost úlohy = maximální doba jejího trvání. */
    private static final int  NEJDELŠÍ = 1000;

    /** Generátor náhodných obtížností vytvářených úloh. */
    private static final Random náhoda = new Random();


//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================================
```

```
    /** Počet dosud vytvořených úloh. */
    private static int počet = 0;   //Počet doposud vytvořených úloh


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Pracnost úlohy = doba, po kterou bude zpracovávána. */
    private final int pracnost = náhoda.nextInt( NEJDELŠÍ );

    /** Identifikátor úlohy. */
    private final String ID = "Úloha_" + ++počet + "-" + pracnost;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří novou úlohu s náhodnou obtížností.
     */
    public Úloha()
    {
        IO.čekej( PRODLEVA );
        //Zaznamená na výstupu, kdy byla úloha vytvořena a jakou má obtížnost
        SynchroTisk.tiskni( String.format(
            "%TH:%<TM:%<TS,%<TL *** Vytvořena: %s",
             System.currentTimeMillis(), this ) );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Vrátí pracnost daného procesu.
     */
    public int getPracnost()
    {
        return pracnost;
    }


    /*****************************************************************************
     * Vrátí identifikátor úlohy, který obsahuje její "rodné číslo" a obtížnost.
     */
    public String toString()
    {
        return ID;
    }

}
```

466. Ta je opravdu jednoduchá, tu pochopím i já. Takže co budeme s těmi úlohami dělat?

Simulace procesů

Budeme je řešit. Připravil jsem dvě třídy představující symbolické procesy, které dostanou vyřešení zadané úlohy na starost. První z nich se jmenuje `ProcesSolo` a simuluje klasický proces. Její instance umí prozradit, zda je proces volný (tj. zda právě neřeší nějakou úlohu), jak dlouho mu bude případné řešení aktuální úlohy ještě trvat,

a především pak umí zadanou úlohu zpracovat. Než mu ji ale předáš ke zpracování, musíš si nejprve ověřit, že je doopravdy volný, protože jinak ti vyhodí výjimku.

Výpis 27.2: Definice třídy `ProcesSolo`, jejíž instance představují procesy schopné řešit zadané úlohy

```
package rup.česky.vzory._27_řetěz;

import rup.česky.společně.IO;
import rup.česky.společně.SynchroTisk;

/*******************************************************************************
 * Instance třídy <code>ProcesSolo</code> představují procesy,
 * které o sobě nevědí a jejichž spouštění řídí volající program.
 */
public class ProcesSolo
{
//== PROMĚNNÉ ATRIBUTY TŘÍDY ===================================================

    /** Počet doposud vytvořených procesů. */
    private static int počet = 0;


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Identifikátor procesu s jeho "rodným číslem". */
    private final String ID = "Proces_" + ++počet;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Kdy skončí zpracovávání právě zpracovávané úlohy. */
    private volatile long skončím;

    /** Je-li proces zrovna volný, tj. nezpracovává-li právě žádnou úlohu. */
    private volatile boolean volný = true;

    /** Zpracovávaná úloha. */
    private volatile Úloha úloha;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /***************************************************************************
     * Vytvoří nový proces a oznámí to.
     */
    public ProcesSolo()
    {
        SynchroTisk.tiskni( "Vytvářím proces " + this );
    }



//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /***************************************************************************
     * Vrátí identifikační řetězec s "rodným číslem" procesu.
```

```
     */
    public String toString()
    {
        return ID;
    }


    /*************************************************************************
     * Vrátí informaci o tom, je-li proces volný.
     * Není-li volný, vypíše zprávu, v níž oznámí, kolik času mu zbývá do
     * vyřešení úlohy a kterou úlohu řeší.
     */
    public boolean volný()
    {
        if( !volný )
            zpráva( "xxx", "provádí (" + zbývá() + ")" );
        return volný;
    }


    /*************************************************************************
     * Vrátí počet milisekund, které zbývají do vyřešení úlohy.
     */
    public int zbývá()
    {
        return Math.max( 0, (int)(skončím - System.currentTimeMillis()) );
    }


    /*************************************************************************
     * Zpracuje zadanou úlohu.
     *
     * @param úloha  Úloha, o jejíž zpracování je proces žádán
     */
    public void zpracuj( final Úloha úloha )
    {
        //Proces nesmí být znovu požádán, dokud ještě pracuje na minulé úloze
        if( !volný )
            throw new IllegalStateException( this +
                " je vytížen, do dokončení práce zbývá " + zbývá() + " ms" );

        volný = false;
        this.úloha = úloha;
        final int pracnost = úloha.getPracnost();
        skončím = System.currentTimeMillis() + pracnost;
        zpráva( "+++", "začal" );

        //Úlohu vykonává v samostatném vlákně
        new Thread( toString() ) {
            public void run() {
                IO.čekej( úloha.getPracnost() );
                zpráva( "---", "dokončil" );
                volný = true;
            }
        }.start();
    }
```

```
//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ============================================

    /*****************************************************************************
     * Vypíše zprávu s aktuálním systémovým časem, činností, kterou
     * proces oznamuje, a názvem zpracovávané úlohy.
     */
    private void zpráva( String značka, String akce )
    {
        SynchroTisk.tiskni( String.format(
            "%TH:%<TM:%<TS,%<TL %s %s %s zpracování: %s",
            System.currentTimeMillis(), značka, this, akce, úloha ) );
    }


}
```

467. Musím přiznat, že metodu `zpracuj(Úloha)` jsem tak zcela nepochopil. Proč má parametr definovaný jako `final`?

Proč je parametr definovaný jako `final`

Protože s tímto parametrem pracuje instance anonymní třídy definované uvnitř metody, a ty, jak jistě víš, nemohou pracovat s lokálními proměnnými, ale pouze s lokálními konstantami.

Jinak jsem si myslel, že definice zase tak složitá není. Instance procesu si zapamatuje tři věci:

- že je obsazená, tj. že už na něčem pracuje, nemůže proto do vyřešení dané úlohy přijímat další zakázky,
- kdy bude s řešením hotova, aby mohla odpovídat na otázku, jak dlouho jí to bude ještě trvat,
- jakou úlohu právě řeší – to bude vypisovat ve svých zprávách.

Pak vypíše zprávu o tom, že proces začal řešit zadanou úlohu, spustí vlákno, v němž bude úlohu řešit, a ukončí svoji činnost. Úloha se zatím bude řešit v paralelně běžícím vlákně – třeba na jiném procesoru.

468. Vidím, že to vlákno toho moc nedělá.

Proč vlákno nic nedělá

Samozřejmě, vždyť je to jenom simulace. Vlákno si zjistí, jak dlouho má danou úlohu řešit a na tuto dobu se uspí (můžeš si myslet, že zadané úlohy řeší ve spánku). Když zadaná doba uplyne, vlákno vypíše zprávu o vyřešení úlohy, označí svůj proces opět za volný a ukončí svoji činnost.

469. Říkal jsi, že první proces je klasický. V čem je ten druhý neklasický?

V čem je proces neklasický

Neklasičnost těchto procesů spočívá v tom, že instance třídy `ProcesŘetěz`, které tyto procesy simulují, jsou zřetězeny. Třída tak vlastně nepřímo definuje fond, do kterého ale nemusíš použité instance vracet, protože se do něj vrátí samy hned poté, co pro tebe splní zadanou úlohu.

Aby měla vznik svých instancí pod kontrolou, nezveřejňuje konstruktor, ale místo něj dává uživatelům k dispozici pouze jednoduchou tovární metodu. Kdo bude chtít

zpracovat nějakou úlohu, požádá třídu o instanci a tu pak pověří zpracováním své úlohy.

Žadatel se nemusí starat o to, jestli tato instance není náhodou zrovna zaměstnaná řešením nějaké jiné úlohy. Ponechá na ní, aby zařídila, aby se v takovém případě o řešení jeho úlohy někdo postaral.

470. To by šlo zařídit např. tak, že by třída vracela pouze úlohy, které jsou v danou chvíli volné.

Proč je třeba vytvářet nové procesy

To je sice pravda, ale v některých případech by to mohlo vést k vytváření příliš mnoha procesů. Stačí, aby se našlo víc takových, kteří o proces požádají „do foroty" a pak jim dlouho trvá, než pro něj připraví úlohu, o jejíž zpracování jej požádají. Za tu dobu mohl ten proces vyřešit několik jiných úloh.

471. Máš pravdu, to mne nenapadlo. Jak to tedy řeší?

Jak je řešeno poskytování procesů

Podívej se do výpisu 27.3. Pokud instance, která je požádána o zpracování úlohy, zrovna pracuje na jiné úloze, zapamatuje si, že už ji podruhé požádal, a předá žádost další instanci v řetězu. Ta udělá samozřejmě totéž.

Dopadne to tak, že se buď v řetězu najde instance, která je volná a o zadanou úlohu se může postarat, nebo se požadavek kruhem vrátí zase k instanci, která byla požádána jako první. Ta si ale všimne, že už tu tento požadavek byl, takže jí dojde, že žádný z procesů v řetězu nejspíš není volný, a vytvoří proto nový proces, který zpracováním dané úlohy pověří.

Výpis 27.3: Definice třídy `ProcesŘetěz`, jejíž instance představují vzájemně zřetězené procesy tvořící jednoduchý fond

```
package rup.česky.vzory._27_řetěz;


/*******************************************************************************
 * Instance třídy <code>ProcesŘetěz</code> představují procesy,
 * které o sobě vědí a jsou si schopny požadavek na zpracování úlohy
 * vzájemně předat, aniž by tím obtěžovaly žadatele.
 */
public class ProcesŘetěz extends ProcesSolo
{
//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================================

    /** Kandidát příštího oslovení. */
    private static ProcesŘetěz naŘadě = new ProcesŘetěz();


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ===============================================

    /** Následník procesu v řetězci. */
    private ProcesŘetěz další;

    /** Byla-li již tato instance požádána o vyřešení dané úlohy.
     *  Je-li instance žádána znovu, znamená to, že všechny ostatní jsou také
     *  zaneprázdněny, a je proto třeba vytvořit nový proces. */
```

```
    private boolean požádán = false;


//== NESOUKROMÉ METODY TŘÍDY ================================================

    /**************************************************************************
     * Vrátí instanci, kterou je možno oslovit.
     */
    public static ProcesŘetěz getInstance()
    {
        return naŘadě;
    }

//== KONSTRUKTORY A TOVÁRNÍ METODY ==========================================

    /**************************************************************************
     * Vytvoří nový proces a zařadí jej do řetězu před proces,
     * který je současným kandidátem oslovení.
     */
    private ProcesŘetěz()
    {
        if( naŘadě == null ) {
            //První instance ještě nebyla vytvořena - atribut bude
            //inicializován až po návratu z konstruktoru.
            //První vytvořená instance bude svým vlastním následníkem
            další = this;
        }else{
            //Instance se vloží za tu, která je na řadě
            další        = naŘadě.další;
            naŘadě.další = this;
        }
    }

//== NESOUKROMÉ METODY INSTANCÍ =============================================

    /**************************************************************************
     * Zpracuje zadanou úlohu. Je-li zrovna zaměstnán řešením jiné úlohy,
     * předá požadavek dalšímu procesu v řadě. Pokud se mu požadavek
     * po oběhnutí celého řetězu vrátí, vytvoří nový proces,
     * který pak zpracováním dané úlohy pověří.
     *
     * @param úloha  Úloha, o jejíž zpracování je proces žádán
     */
    public void zpracuj( final Úloha úloha )
    {
        if( požádán ) {
            //Už je žádán podruhé, tj. byly požádány všechny procesy
            //v řetězu a všechny odmítly =>
            //je třeba vytvořit další proces a tomu úlohu předat ke zpracování
            new ProcesŘetěz().zpracuj( úloha );
        }else{
            if( volný() ) {
                super.zpracuj( úloha );
            } else {
                požádán = true; //Pro případ, že by žádost oběhla řetěz
                další.zpracuj( úloha );
                požádán = false;    //Žádost byla zpracována
            }
```

```
        }
    }

}
```

472. Jenomže nikam neuloží odkaz na vytvořený proces, takže ho správce paměti při nejbližší příležitosti odstraní. To se nekamarádí s tím, že jsi říkal, že vytváření procesů je drahá věc a neměli bychom s ním hýřit.

Proč není proces odstraněn

Instance, která iniciuje vytvoření nového procesu, opravdu nikam získaný odkaz neukládá a prostě jej zahazuje. Ona jej totiž nikam ukládat nemusí, protože o to se postará již konstruktor daného procesu, který vytvořenou instanci začlení do řetězu, takže na ni přirozeně odkazuje její předchůdce v řetězu.

473. Aha. Tohle je vlastně ta struktura, ve které musíš jednotlivé vazby definovat sám, protože procesy mezi sebou žádné přirozené vazby nemají.

Ano. Navíc je to struktura, v níž se pořadí jejích členů dynamicky mění. Po každém přidání nového člena je potřeba některou vazbu přerušit a jinou vytvořit.

474. Takže už zbývá jenom vyzkoušet, jak oba druhy procesů pracují.

K tomu jsem připravil třídu `Systém`, která ukazuje, jak je možné každou z definovaných tříd použít.

Metoda `cyklem(int)` ukazuje, jak je možné vyhledávat volné procesy v cyklu. Jak vidíš, je trochu složitější a její druhou nevýhodou je, že je příliš vázána na používané procesy. Sama si je vytváří, udržuje je v seznamu a kolem dokola obíhá, aby zjistila, který z nich je ten pravý.

Metoda `řetězem(int)` naproti tomu ukazuje, jak jednoduše lze celou problematiku vyřešit, využijeme-li aplikace návrhového vzoru *Řetěz odpovědnosti* a ponecháme zodpovědnost za spuštění toho správného procesu na instanci, kterou oslovujeme, a na jejích spolupracovnících.

Výpis 27.4: Definice třídy `Systém` simulující systém, který vytváří úlohy a předává je svým procesům ke zpracování

```
package rup.česky.vzory._27_řetěz;

import java.util.ArrayList;
import java.util.List;
import rup.česky.společně.IO;

import rup.česky.společné.SynchroTisk;


/*******************************************************************************
 * Instance třídy <code>Systém</code> slouží k otestování funkce
 * obou druhů procesů.
 */
public class Systém
{
//== NESOUKROMÉ METODY TŘÍDY ===================================================
```

```
/*****************************************************************************
 * Metoda má za úkol předat zadaný počet úloh ke zpracování
 * nezávislým procesům. Aby se předání úloh ke zpracování příliš
 * nezdržovalo, vytvoří si fond, v němž uchovává všechny procesy,
 * a o zpracování vždy požádá ten, který je volný.
 * Není-li ve fondu žádný volný proces, vytvoří nový fond,
 * zařadí jej do fondu a pověří jej zpracováním aktuální úlohy.
 *
 * @param úloh  Počet úloh, které se mají jednotlivým procesům
 *              postupně přidělit
 */
public static void cyklem( int úloh )
{
    List<ProcesSolo> fond = new ArrayList<ProcesSolo>();
    fond.add( new ProcesSolo() );   //Proces pro první úlohu
    int procesů = fond.size();      //Aktuální počet procesů

    int p  = 0,  //Index aktuálně oslovovaného procesu
        p0 = 0;  //Index prvního osloveného procesu umožňuje poznat,
                 //kdy jsme již oslovili všechny a je třeba vytvořit nový
    for( int i=0;   i < úloh;   i++ ) {
        Úloha úloha = new Úloha();
        //Uloha je vytvořena, musíme najít proces, který ji zpracuje
        for(;;) {
            //Aby mohl proces úlohu dostat ke zpracování, musí být volný
            if( fond.get(p).volný() ) {
                fond.get(p).zpracuj( úloha );   //Je volný - dostane ji
                break;  //Úloha je zpracovávána, můžem vzít další
            }
            //Oslovený proces byl obsazen, oslovíme další
            p = (p+1) % procesů;  //Fond procházím kolem dokola
            if( p == p0 ) {       //Oslovili jsme již všechny
                //Všechny procesy jsou obsazeny, musíme vytvořit nový
                ProcesSolo ps = new ProcesSolo();
                fond.add( ps ); //Přidáme jej do fondu
                procesů++;      //Počet potřebujeme pro rotaci indexu
                ps.zpracuj( úloha ); //Proces je nový, takže neobsazený
                break;  //Úloha je zpracovávána, můžem vzít další
            }
        }
        //O zpracování další úlohy požádáme další proces v řadě
        p = (p+1) % procesů;
        //Zapamatujeme si, kde jsme začali, abychom poznali,
        //kdy už jsme všechny procesy oslovili
        p0 = p;
    }
}


/*****************************************************************************
 * Metoda má za úkol předat zadaný počet úloh ke zpracování zřetězeným
 * procesům. Nestará se o to, který z procesů je obsazený a který volný -
 * to si procesy vyřídí mezi sebou samy.
 *
 * @param úloh  Počet úloh, které se mají jednotlivým procesům
 *              postupně přidělit
 */
```

```
    public static void řetězem( int úloh )
    {
        for( int i=0;   i < úloh;   i++ ) {
            Úloha proces = new Úloha();
            ProcesŘetěz.getInstance().zpracuj( proces );
        }
    }


//== TESTY ======================================================================

    /** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {
        final int POČET = 20;
        cyklem( POČET );
        IO.čekej( 1000 );
        SynchroTisk.tiskni( "=====  SKONČIL CYKLUS  =====\n\n\n" +
                            "=====  ZAČÍNÁ  ŘETĚZEC  =====" );
        řetězem( POČET );
    }
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Řetěz odpovědnost* ukazuje možné řešení úlohy, vyžadující zaslání zprávy některému ze skupiny objektů, aniž by bylo předem známo, který z nich je v daný okamžik ten pravý.
- Využívá se v případě, kdy není dopředu jasné, koho oslovit.
- Skupina instancí, z nichž bude některá reagovat, je navzájem propojena odkazy do řetězu.
- Neumí-li oslovený objekt reagovat, předá požadavek dalšímu v řetězu s nadějí, že on to dokáže.
- Řetěz umožňuje dynamicky měnit konečného adresáta dané zprávy.
- Instance v řetězu mohou být různých typů – pak je vhodné aplikovat vzor *Strom*.
- Návrhový vzor *Řetěz odpovědnost* patří mezi vzory uvedené v GoF.

KAPITOLA 28

Až se to stane, dám ti vědět (Pozorovatel – Observer)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Zavádí vztah mezi objekty (pozorovateli) reagujícími na změnu stavu (pozorovaného) objektu nebo na jím sledované události. Pozorovatelé se u pozorovaného objektu přihlásí a ten je pak na každou změnu svého stavu či výskyt události upozorní.

[1] **Definice v GoF:** Define a one-to many dependency between objects so that when one objects change state, all its dependents are notified and updated automatically. – Zavádí vztah 1:N mezi objekty; když jeden objekt změní stav, všechny na něm závislé objekty jsou na tuto změnu upozorněny a automaticky se aktualizují.

Účel

Dva způsoby čekání na událost:

– Oslík ve filmu Shrek

– lůžkový vlak

475. Při čtení stručné charakteristiky mne napadlo, k čemu je dobré, když objekt někoho upozorňuje na to, že se jeho stav změnil.

Vysvětlím ti to na příkladu: jestli jsi viděl druhý díl Shreka, v němž Shrek na začátku odjíždí se svojí ženou k jejím rodičům do Země za sedmero horami a řekami. Jede s nimi Oslík, který se celou cestu každou chvíli ptá: „Už tam budem?“

To je jeden způsob, jak čekat na změnu stavu. Druhý způsob znají cestující z lůžkových vlaků: při nástupu řeknou průvodčímu, kam jedou, a před příjezdem do cílové stanice je průvodčí dojde vzbudit a upozornit je, že za chvíli dorazí do cíle své cesty.

Tento druhý způsob popisuje princip činnosti, na němž je založen návrhový vzor *Pozorovatel*.

476. Chápu, že ten druhý způsob je daleko výhodnější – nemusíš se o nic starat, protože ti někdo včas řekne, že máš vystoupit. Můžeš mi přiblížit pár situací, kdy se tento návrhový vzor používá?

Použití v GUI

Jedním z typických je použití v uživatelském rozhraní. Program se pořád nevyptává, jestli již uživatel nestiskl tlačítko v dialogovém okně, ale přihlásí se u tlačítka jako posluchač, a až tlačítko někdo stiskne, tak tlačítko oznámí všem přihlášeným posluchačům, že bylo stisknuto, a ti na to mohou zareagovat.

477. Tím chceš naznačit, že u jednoho tlačítka může být přihlášeno několik posluchačů, kteří budou všichni reagovat na jeho stisk?

Může. A co víc: u tlačítka (a obecně u většiny grafických komponent v Javě) může být přihlášeno několik skupin posluchačů. Jedni budou čekat na jeho stisk, druzí na příjezd či odjezd myši, třetí na jeho aktivaci a deaktivaci. K tlačítku se může přihlásit celkem 12 různých skupin posluchačů.

478. Pochopil jsem z toho, že tento návrhový vzor je v grafických knihovnách hojně používaný. Pochopil jsem dobře, že pozorovatele a posluchače mám považovat za synonyma?

Pozorovatel a posluchač jsou synonyma

Ano. Návrhový vzor se sice jmenuje *Pozorovatel* (*Observer*), ale tvůrci knihovny AWT označili tyto objekty jako posluchače (listener). Nicméně se jedná o aplikaci téhož návrhového vzoru.

479. Stavem tlačítka může být to, jestli je stisknuté nebo puštěné, ale odhaduji, že bych takto mohl nastavit reakci nejenom na změnu stavu, ale obecně na jakoukoliv událost.

Lze reagovat na cokoliv

Správně. Proto je na využití návrhového vzoru *Pozorovatel* postaveno celé *událostmi řízené programování*. Vezmi si např. třídu `SprávcePlátna` z doprovodné knihovny. Ta jej také využívá hned několikrát: jednou při překreslování objektů na plátně, podruhé při reakci na stisk tlačítka na klávesnici a potřetí při reakci na nějakou událost myši.

To, že někdo stiskl tlačítko na klávesnici nebo myši, bychom nejspíš nepovažovali za změnu stavu plátna, ale pouze za událost, která by mohla s děním na plátně nějak souviset – např. by mohla ovlivnit chování některých nakreslených objektů.

480. Zatím jsi říkal, že může být u jednoho pozorovaného objektu přihlášeno několik posluchačů. Může to být i obráceně, tj. že by byl jeden posluchač přihlášen u několika pozorovaných objektů?

Může. Tato konstrukce je použita v příkladu k návrhovému vzoru *Prostředník*, jehož zdrojový kód najdeš ve výpisu 29.1 na straně 390. Není to ale nic tak zásadního, abys kvůli tomu musel odbíhat. Prozatím stačí, když budeš vědět, že se to občas používá, a až si budeme o tom příkladu povídat, tak ti to připomenu.

Implementace

S využitím předpřipravených tříd ve standardní knihovně

481. K čemu může být návrhový vzor *Pozorovatel* dobrý, jsem už asi pochopil. Teď by mne zajímalo, jaká jsou doporučení pro jeho implementaci.

Jednou z možností je využít standardní knihovny, která definuje třídu `java.util.Observable` a rozhraní `java.util.Observer`. Toto řešení má sice své mouchy, ale na druhou stranu je lze použít téměř mechanicky:

– Pozorovaný = `Observable`

- třídu, jejímiž instancemi budou pozorované objekty, definuješ jako potomka třídy `Observable` a zařídíš, aby ve správnou chvíli obvolával přihlášené pozorovatele;

– Pozorovatel = `Observer`

- třídy, jejichž instance budou vystupovat jako pozorovatelé, necháš implementovat rozhraní `Observer` a definuješ v nich metodu `update` specifikující reakci na očekávanou událost;

– Pozorovatelé se přihlásí u pozorovaného

- necháš přihlásit pozorovatele u pozorovaného objektu, aby je pozorovaný objekt mohl „obvolávat“ v okamžiku, kdy nastane událost, na kterou čekají.

482. Jaké metody deklaruje rozhraní `Observer`?

Co deklaruje `Observer`

Rozhraní `Observer` deklaruje jedinou metodu

```
public void update(Observable o, Object arg)
```

jejímž prvním parametrem je pozorovatelný objekt, který „obvolává“ své pozorovatele, a druhým parametrem je objekt (nejčastěji přepravka či pole), v němž tento objekt může svým pozorovatelům předávat předem domluvené informace.

483. Proč se pozorovaný objekt předává svým pozorovatelům jako parametr. Vždyť se u něj přihlásili, tak jej musí znát.

Proč se pozorovaný předává jako parametr

To je sice pravda, ale toto řešení umožňuje, aby byl jeden pozorovatel přihlášený u několika pozorovaných objektů současně a podle prvního parametru zjistil, který z pozorovaných objektů mu oznamuje změnu stavu.

484. Ještě mi přibliž význam druhého parametru.

K čemu je druhý parametr

Pozorovatel čeká na změnu stavu pozorovaného objektu nebo na nějakou událost, která s ním souvisí. Možných stavů či událostí ale může být víc. V druhém parametru proto může pozorovaný objekt podrobněji specifikovat, co se vlastně stalo.

– strategie push (tlačná)

Předávání informací v parametru je jedním z možných způsobů jejich předání. Protože při něm pozorovaný objekt tyto informace pozorovanému objektu vnucuje, bývá tato strategie označována slovem `push` (tlačit).

– strategie pull (tažná)

Druhou možností je, že pozorovaný objekt pozorovatelům nic nepředá a bude předpokládat, že budou-li se potřebovat něco dozvědět, sami si o potřebnou informaci řeknou, tj. zavolají některou z jeho metod. Protože od něj musí tuto informaci teprve „vyloudit", bývá tato strategie označována slovem `pull` (táhnout).

V dalším textu budu tyto strategie označovat stejně jako remorkéry, tj. jako tlačnou a tažnou.

485. A která je lepší?

Proč je tažná lepší

Z hlediska snadnějších budoucích úprav je výhodnější strategie tažná, protože nespojuje tak těsně pozorovaný objekt s jeho pozorovateli. Když bude po úpravě programu některý z pozorovatelů vyžadovat dosud nepředávané informace, stačí pouze rozšířit definici třídy pozorovaného objektu o příslušnou metodu a z daného pozorovatele tuto metodu zavolat.

Při používání tlačné strategie bychom museli navíc změnit definici přepravky, v níž informace předáváme. Navíc tlačná strategie často zbytečně vnucuje pozorovatelům řadu informací, o které tito pozorovatelé nestojí, protože je mohou využít pouze v některých situacích.

486. Je vůbec někdy tlačná strategie výhodná?

Kdy je tlačná výhodná

Při použití tlačné strategie totiž odpadnou volání metod pozorovaného objektu. Je trochu efektivnější v případě, kdy se předávanou sadu informací potřebují pozorovatelé dozvědět téměř pokaždé a kdy neočekáváme, že by se definice pozorovatelů a pozorovaných objektů v budoucnu měnila.

Kromě toho můžeme použít kombinovanou strategii, při níž pozorovaný objekt „dotlačí" často používané informace a méně často používané informace z něj budou pozorovatelé v případě potřeby „tahat".

Nevýhody:

487. Říkal jsi, že řešení ze standardní knihovny má své mouchy.

– Pozorovaný je potomek `Observable`

Jeho největší nevýhodou je to, že vyžaduje, aby třída pozorovaných objektů byla definována jako potomek třídy `Observable`, což se ti někdy nehodí, protože tvá třída je již potomkem někoho jiného.

– `Observable` neimplementuje žádné rozhraní

Třída `Observable` navíc neimplementuje ani žádné rozhraní, které by umožňovalo definovat tvoji vlastní implementaci zcela po svém.

– `Observable` není definována jako abstraktní

Kromě toho se domnívám, že ji její původní autor zapomněl definovat jako abstraktní, protože její instance sice vytvořit jdou, ale jako pozorovatelné objekty jsou nepoužitelné, takže ti opravdu nezbývá, než vytvořit jejího potomka.

488. Proč?

Proč `Observable` vyžaduje potomka

„Obvolávání" pozorovatelů se děje jako výsledek spolupráce tří metod:

- `protected void setChanged()`
  Nastaví příznak označující, zda je proč „obvolávat" pozorovatele.
- `protected void clearChanged()`
  Shodí příznak označující, zda je proč „obvolávat" pozorovatele.

- `public void notifyObservers(Object arg)`
  `public void notifyObservers()`
  Je-li nastaven výše zmíněný příznak, „obvolá“ všechny přihlášené pozorovatele, tj. zavolá jejich metodu `update(Observable,Object)`.

Při této koncepci budou pozorovatelé „obvoláni“ pouze tehdy, bude-li nahozen příznak registrující, že se opravdu něco stalo. Problém je ale v tom, že metodu `setChanged()`, která jediná může tento příznak nastavit, mohou volat pouze potomci třídy `Observable`.

Každý pozorovatelný objekt spolupracující s instancemi rozhraní `Observer` proto musí být nutně potomkem třídy `Observable`.

489. To by snad bylo možné obejít pomocí vnitřních tříd, ne?

Obejití pomocí vnitřních tříd

Samozřejmě, můžeš to udělat obdobně, jako to dělají kontejnery s iterátory: definují tovární metodu, která vrátí instanci vnitřní třídy, implementující potřebné rozhraní.

Možností je několik. Jednou z nich je definovat u objektu např. jednoduchou tovární metodu

```
public Observable getObservable()
```

která bude při volání metody

```
public void update(Observable o, Object arg)
```

předávat v druhém parametru odkaz na vlastní pozorovatelný objekt.

Druhou možností je rozšířit třídu pozorovatelných objektů o metody pro přihlášení, resp. odhlášení, pozorovatelů a definovat v ní atribut odkazující na instanci třídy `Observable`, která by pak všechny pozorovatelné objekty „obvolávala“ a opět předávala v druhém parametru odkaz na vlastní pozorovaný objekt.

Nevýhody

Nevýhodou obou těchto řešení je, že pozorovatelé v prvním parametru svých aktualizačních metod dostanou odkaz na instanci třídy `Observable`, která jim nic významného nepřináší a navíc je soukromým atributem pozorovaného objektu.

Shrnuto a podtrženo, nevyhovují-li ti omezení, která na tebe tato koncepce klade, bude asi nejlepší si definovat svoji vlastní sadu, která na tebe nebude taková omezení klást. Navíc budeš moci v této definici využít i parametrizovaných typů, takže ti pak odpadnou některá přetypování.

Přiznejme si, že této nevýhody si zřejmě byli vědomi i tvůrci standardní knihovny, a proto uvedenou dvojici v celé standardní knihovně ani jednou nepoužili (hovořím o části knihovny Java SE 6, jejíž zdrojové kódy jsou součástí JDK).

Příklad

490. Máš nějaký příklad s vlastním řešením pozorovatele?

Příklad: Správce Plátna

V doprovodné knihovně tvarů jich najdeš hned několik. Nejpozorovanější objekt je `SprávcePlátna`, protože jej mohou pozorovat hned čtyři druhy pozorovatelů:

- Objekty, které chtějí být zobrazovány na plátně, se u správce plátna hlásí některou z metod `přidejXxx`.

- Objekty, které chtějí reagovat na příkazy zadávané z klávesnice, se hlásí voláním metody `přihlasKlávesnici(java.awt.event.KeyListener)`.
- Objekty, které chtějí reagovat na příkazy zadávané myší, se hlásí voláním metody `přihlasMyš(java.awt.event.MouseListener)`.
- Objekty, které chtějí reagovat na případnou změnu velikosti kroku plátna, se hlásí voláním metody `přihlasPřizpůsobivý(IPřizpůsobivý)`.

  Jen pro úplnost dodám, že přizpůsobivé objekty jsou takové, které jsou schopny reagovat na změnu kroku plátna tak, aby byla zachována jejich relativní pozice a velikost vůči mřížce, tj. aby např. obrazec, který zabírá druhé a třetí políčko ve čtvrtém řádku, zabíral toto políčko i po změně velikosti kroku a tím i rozteče čar tvořících mřížku ohraničující jednotlivá pole.

– Jen výňatek zdrojového kódu

Každá z uvedených metod zařadí přihlašující se objekt do seznamu objektů, které „obvolává“ v okamžiku, kdy k očekávané události dojde. Ve výpisu 28.1 si můžeš prohlédnout kódy některých metod souvisejících s implementací funkcí implementovaných prostřednictvím návrhového vzoru *Pozorovatel*. Není tam ukázán kompletní zdrojový kód třídy `SprávcePlátna`, protože má okolo 1 200 řádků, z nichž značná část s těmito funkcemi vůbec nesouvisí.

– Chybí i metody pro přidávání a překreslení

Není tam ukázán ani zdrojový kód metod souvisejících s přidáváním tvarů do správy, ani kód překreslující celé plátno, protože obsahuje řadu konstrukcí, které by potřebovaly podrobnější vysvětlení, ale které s implementací návrhového vzoru nijak nesouvisí. Kdyby tě zajímal kompletní zdrojový kód této třídy, můžeš si jej najít mezi doprovodnými programy.

– Přihlašování posluchačů klávesnice a myši jen zprostředkuje

Ve vztahu k přihlašování posluchačů klávesnice a myši slouží `SprávcePlátna` pouze jako prostředník aplikující návrhový vzor *Zástupce*. Přihlašovací požadavek předá oknu a to pak tvé posluchače upozorňuje na příslušné události.

– Kompletní je jen přizpůsobování změně kroku

Jedinou kompletně popsanou aplikací tohoto návrhového vzoru je tedy přihlašování přizpůsobivých posluchačů. Jejich upozorňování na změnu kroku plátna má na starosti metoda `setKrokRozměr(int,int,int)`, která po novém nastavení rozměrů plátna zjistí, zda se změnil jeho krok, a pokud ano, tak všechny přihlášené přizpůsobivé posluchače na tuto skutečnost upozorní.

Abys měl implementaci tohoto vzoru doopravdy kompletní, přidal jsem ve výpisu i zdrojový kód rozhraní `IPřizpůsobivý`, které mimo jiné obsahuje i vnořený adaptér.

Výpis 28.1: Výběr deklarací a metod třídy `SprávcePlátna` souvisejících s funkcemi implementovanými prostřednictvím vzoru Pozorovatel

```
package rup.česky.tvary;

// ...  Vynechané importy a dokumentace

public class SprávcePlátna
{
    /** Aplikační okno animačního plátna. */
    private final JFrame okno;

    /** Seznam přihlášených přizpůsobivých předmětů. */
    Set<IPřizpůsobivý> přizpůsobivý = new LinkedHashSet<IPřizpůsobivý>();
```

```
/** Zda se mají přizpůsobiví upozorňovat na změny rozměru pole. */
private boolean hlásitZměnyRozměru = true;


/*****************************************************************************
 * Přihlásí posluchače událostí klávesnice.
 *
 * @param posluchač  Přihlašovaný posluchač
 */
public void přihlasKlávesnici( KeyListener posluchač )
{
    okno.addKeyListener( posluchač );
}


/*****************************************************************************
 * Odhlásí posluchače klávesnice.
 *
 * @param posluchač  Odhlašovaný posluchač
 */
public void odhlasKlávesnici( KeyListener posluchač )
{
    okno.removeKeyListener( posluchač );
}


/*****************************************************************************
 * Přihlásí posluchače událostí myši.
 *
 * @param posluchač  Přihlašovaný posluchač
 */
public void přihlasMyš( MouseListener posluchač )
{
    okno.addMouseListener( posluchač );
}


/*****************************************************************************
 * Odhlásí posluchače myši.
 *
 * @param posluchač  Odhlašovaný posluchač
 */
public void odhlasMyš( MouseListener posluchač )
{
    okno.removeMouseListener( posluchač );
}


/*****************************************************************************
 * Přihlásí posluchače změny velikosti pole.
 *
 * @param  posluchač  Přihlašovaný posluchač
 * @return  Informace o tom, byl-li posluchač doopravdy přidán -
 *          false oznamuje, že posluchač už byl přihlášen,
 *          a nebylo jej proto třeba přidávat.
 */
public boolean přihlasPřizpůsobivý( IPřizpůsobivý posluchač )
{
```

```
        return přizpůsobivý.add( posluchač );
    }


    /*****************************************************************************
     * Odhlásí posluchače změny velikosti pole.
     *
     * @param  posluchač  Odhlašovaný posluchač
     * @return  Informace o tom, byl-li posluchač doopravdy odebrán -
     *          false oznamuje, že posluchač už nebyl přihlášen,
     *          a nebylo jej proto třeba odebírat.
     */
    public boolean odhlasPřizpůsobivý( IPřizpůsobivý posluchač )
    {
        return přizpůsobivý.remove( posluchač );
    }


    /*****************************************************************************
     * Nastaví, zda se mají přihlášeným posluchačům hlásit změny
     * velikosti kroku, a vrátí původní nastavení.
     *
     * @param hlásit  Požadované nastavení (true=hlásit, false=nehlásit).
     * @return Původní nastavení
     */
    public boolean hlásitZměnyRozměru( boolean hlásit )
    {
        boolean ret = hlásitZměnyRozměru;
        hlásitZměnyRozměru = hlásit;
        return ret;
    }


    /*****************************************************************************
     * Nastaví rozměr plátna zadáním bodové velikosti políčka a
     * počtu políček ve vodorovném a svislém směru.
     * Při velikosti políčka = 1 se vypíná zobrazování mřížky.
     *
     * @param  krok    Nová bodová velikost políčka
     * @param  pŠířka  Nový počet políček vodorovně
     * @param  pVýška  Nový počet políček svisle
     */
    public synchronized void setKrokRozměr( int krok, int pŠířka, int pVýška )
    {
        //Ověření korektnosti zadaných parametrů
        Dimension obrazovka = Toolkit.getDefaultToolkit().getScreenSize();
        if( (krok   < 1)  ||
            (pŠířka < 2)  ||  (obrazovka.width  < šířkaBodů) ||
            (pVýška < 2)  ||  (obrazovka.height < výškaBodů) )
        {
            throw new IllegalArgumentException(
                "\nŠpatně zadané rozměry: " +
                "\n  krok =" + krok  + " bodů," +
                "\n  šířka=" + pŠířka + " polí = " + pŠířka*krok + " bodů," +
                "\n  výška=" + pVýška + " polí = " + pVýška*krok + " bodů," +
                "\n  obrazovka= " + obrazovka.width  + "×" +
                                    obrazovka.height + " bodů\n" );
        }
```

```
        šířkaBodů = pŠířka * krok;
        výškaBodů = pVýška * krok;

        okno.setResizable( true );
        vlastníPlátno.setPreferredSize( new Dimension( šířkaBodů, výškaBodů ) );
        okno.pack();
        okno.setResizable( false );

        obrazPlátna = vlastníPlátno.createImage( šířkaBodů, výškaBodů );
        kreslítko = new Kreslítko( (Graphics2D)obrazPlátna.getGraphics() );
        kreslítko.setPozadí( barvaPozadí );

        int starý      = this.krok;
        this.krok      = krok;
        this.šířkaPolí = pŠířka;
        this.výškaPolí = pVýška;

        připravČáry();

        if( hlásitZměnyRozměru  &&  (krok != starý) )
        {
            nekreslit++; {
                for( IPřizpůsobivý ip : přizpůsobivý )
                    ip.krokZměněn( starý, krok );
            }nekreslit--;
        }
        okno.setVisible(true);
        překresli();
        okno.toFront();
    }
```

Výpis 28.2: Definice rozhraní `IPřizpůsobivý` deklarující požadavky na instance, které se chtějí přizpůsobovat změně kroku `SprávcePlátna`, spolu s definicí příslušného adaptéru

```
package rup.česky.tvary;

import rup.česky.tvary.IHýbací;


/*******************************************************************************
 * Rozhraní IPřizpůsobivý je určeno pro instance, které chtějí být schopny
 * reagovat na velikosti kroku a tím i políčka aktivního plátna.
 * Kdykoliv se změní velikost pole aktivního plátna, plátno to oznámí
 * všem přihlášeným přizpůsobivým posluchačům.
 */
public interface IPřizpůsobivý
{
//== DEKLAROVANÉ METODY ========================================================

    /***************************************************************************
     * Přihlásí-li se instance u správce plátna
     * jako přizpůsobivý posluchač, zavolá aktivní plátno tuto její metodu
     * po každé změně kroku a tím i velikosti jeho pole.
     *
     * @param starý     Původní velikost kroku.
```

```
     * @param nový      Nově nastavená velikost kroku.
     */
    public void krokZměněn( int starý, int nový );


//== VNOŘENÉ TŘÍDY ==============================================================

    /**************************************************************************
     * Třída <code>IPřizpůsobivý.Adaptér</code>
     * definuje implicitní implementaci metody {@link #krokZměněn(int, int)}
     * pro třídy, které implementují rozhraní rozhraní {@link IHýbací}.
     */
    public static abstract class Adaptér extends AHýbací
                                         implements IPřizpůsobivý
    {
    //== NESOUKROMÉ METODY TŘÍDY ================================================

        /**********************************************************************
         * Přizpůsobí velikost a pozici svěřeného objektu
         * nové velikosti kroku plátna.
         *
         * @param ih        Přizpůsobující se objekt
         * @param starý     Původní velikost kroku.
         * @param nový      Nově nastavená velikost kroku.
         */
        public static void krokZměněn( IHýbací ih, int starý, int nový )
        {
            double K = (double)nový / starý;
            Pozice p = ih.getPozice();
            Rozměr r = ih.getRozměr();
            ih.setPozice( (int)(K * p.x),     (int)(K * p.y)     );
            ih.setRozměr( (int)(K * r.šířka), (int)(K * r.výška) );
        }


    //== KONSTRUKTORY A TOVÁRNÍ METODY ==========================================

        /**********************************************************************
         * Vytvoří v zadané pozici novou instanci zadaného rozměru.
         *
         * @param x       x-ová souřadnice počátku, x=0 má levý okraj plátna
         * @param y       y-ová souřadnice počátku, y=0 má horní okraj plátna
         * @param šířka   Šířka vytvářeného objektu v bodech
         * @param výška   Výška vytvářeného objektu v bodech
         */
        protected Adaptér( int x, int y, int šířka, int výška ) {
            super( x, y, šířka, výška );
        }


    //== NESOUKROMÉ METODY INSTANCÍ =============================================

        /**********************************************************************
         * Přihlásí-li se instance u správce plátna
         * jako přizpůsobivý posluchač, zavolá aktivní plátno tuto její metodu
         * po každé změně kroku a tím i velikosti jeho pole.
         *
         * @param starý     Původní velikost kroku.
```

```
         * @param nový      Nově nastavená velikost kroku.
         */
        public void krokZměněn( int starý, int nový ) {
            krokZměněn( this, starý, nový );
        }
    }
}
```

491. Myslím, že to přizpůsobování změnám kroku pro základní informaci o použití tohoto vzoru stačí.

Příklad: Multipřesouvač

Kdyby ses chtěl seznámit s další aplikací tohoto vzoru, podívej se v knihovně tvarů na třídu `Multipřesouvač`. Její instance má „dovézt" svěřené objekty do požadovaných cílových pozic. Je-li přesouvaný objekt instancí třídy implementující rozhraní `IMultiposuvný`, bude po dopravení do cíle vyvolána jeho metoda `přesunuto()`. Na této třídě je zajímavé to, že „pozorování" je pouze vedlejší činností doprovázející hlavní náplň jejích posuvných metod a zvyšující jejich užitnou hodnotu.

Zdrojový kód této třídy tu ale neuvedu, protože má přes 400 řádek, z nichž s naším vzorem souvisí jenom hrstka, která je navíc zanořena v okolním kódu zabezpečujícím požadovanou funkčnost a navíc vyžaduje jistou obeznámenost s programováním vláken. Pokud bys měl přesto zájem si jej prohlédnout, tak tě i tady odkážu na zdrojový kód v doprovodných programech.

Shrnutí – co jsme se naučili

- Návrhový vzor *Pozorovatel* bývá někdy označován jako vzor *Posluchač*.
- Vzor ukazuje, jak zabezpečit, aby se objekt včas dozvěděl o události, na kterou čeká, aniž by musel obtěžovat jejího iniciátora neustálými dotazy.
- Ve standardní knihovně je pro aplikaci tohoto vzoru připraveno rozhraní `java.util.Observer` a třída `java.util.Observable`.
- Metoda `update(Observable, Object)` deklarovaná rozhraním `Observer` zavádí první parametr proto, aby mohl být jeden pozorovatel přihlášen u několika pozorovaných objektů. Druhý parametr slouží jako schránka na případné parametry.
- Při aplikaci návrhového vzoru *Pozorovatel* můžeme použít tažnou nebo tlačnou strategii:
  - při tažné strategii si pozorovatel říká pozorovanému objektu o potřebné parametry,
  - při tlačné strategii předá pozorovaný objekt pozorovateli všechny parametry při volání upozorňovací metody.
- Použití implementace ze standardní knihovny vyžaduje definici potomka třídy `Observable`.
- Návrhový vzor *Pozorovatel* patří mezi vzory uvedené v GoF.

KAPITOLA 29

Telefonní ústředna (Prostředník – Mediator)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Odstraní vzájemné vazby mezi řadou navzájem komunikujících objektů tím, že zavede objekt, který bude prostředníkem mezi komunikujícími objekty. Tím zruší přímou vzájemnou závislost komunikujících objektů a umožní upravovat chování každého z nich nezávisle na ostatních.

[1] **Definice v GoF:** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. – Definuje objekt, který zapouzdří způsob interakce skupiny objektů. *Prostředník* prosazuje snížení provázanosti tím, že brání objektům v explicitní vzájemné komunikaci a umožňuje jim tak nezávisle měnit své interakce.

Účel

492. Musím přiznat, že mi ta stručná charakteristika opravdu navozuje představu telefonní ústředny, která zabezpečí, že i když vyměním telefonní aparát, nebo se dokonce přestěhuji, tak dokud nezměním telefonní číslo, všichni se ke mně dovolají a já zase naopak k nim.

Hlavní účel

Přesně tak. Hlavním účelem tohoto návrhového vzoru je zpřetrhat explicitní závislosti mezi komunikujícími objekty. Pokud spolu budou objekty komunikovat přímo, bude muset na každou změnu definice toho prvého reagovat i druhý objekt. Máš-li těch objektů ale celou řadu a komunikují-li každý s každým, pak představa, že bys musel kvůli každé změně jednoho z nich prověřit všechny ostatní, není z nejpříjemnějších.

493. Stejně mi ale připadá, že se zavedením dalšího objektu program zbytečně zesložití. Vždyť před tím jsem jednoduše poslal zprávu cílovému objektu a hned jsem se dozvěděl, co on na to. Podle tohoto návrhového vzoru ji musím místo toho poslat nějakému prostředníku, ten ji za mne pošle dál, obdrží odpověď a tu mi vrátí. Není to zbytečně složité?

Zjednodušení komunikace na příkladu Správce Plátna

Není. Vezmi si jako příklad třídu `SprávcePlátna`. Ta funguje jako prostředník mezi všemi objekty, které jsou na plátně nakresleny. Aby byl obraz na plátně vždy správně vykreslen, musí se po každé změně některého z nakreslených objektů plátno překreslit. Pokud se třeba některý z objektů zasunul pod jiný, musí nakreslit nejprve sebe a pak požádat objekt, pod nímž je zasunut, aby se překreslil.

Teď si představ, že by si musel každý objekt pamatovat ostatní objekty na plátně, aby jim dal v případě změny své pozice či vzhledu vědět, aby se ve vhodnou chvíli překreslily.

Zavedení prostředníka celou situaci zjednoduší. Prostředník si udržuje přehled o tom, komu kdy prozradit, že se něco změnilo, a požádat jej o to, aby se překreslil.

494. Počkej, počkej, V minulé kapitole jsi říkal, že `SprávcePlátna` implementuje návrhový vzor *Pozorovatel*.

Prostředník a *Pozorovatel*

Samozřejmě. Pro přeposílání obdržených zpráv bývá u prostředníků často použit návrhový vzor *Pozorovatel*. Pozorovaný objekt v něm má za úkol prozradit přihlášeným pozorovatelům, že došlo k nějaké události. U návrhového vzoru *Prostředník* je pozorovaným objektem právě náš prostředník a událostí, o níž budou pozorovatelé zpraveni, je zpráva nějakého objektu, kterou je třeba přihlášeným pozorovatelům předat.

`SprávcePlátna` navíc zabezpečí, že zpráva bude předána jednotlivým pozorovatelům ve správném pořadí, aby se opravdu překreslil nejprve ten spodní a teprve pak ten horní. Teď si představ, že by něco podobného musel umět každý z objektů, které se mohou na plátně pohnout či jinak změnit svoji podobu.

Kdy použít vzor *Prostředník*

495. Přiznám se, že mne z takové představy obchází kopřivka. Pochopil jsem, že někdy jsou pravidla pro vzájemnou komunikaci objektů natolik složitá, že nejjednodušším řešením je opravdu definice nějakého inteligentního prostředníka.

– Složitá komunikace objektů

Právě. V řadě situací je vzájemná komunikace objektů poměrně složitá a nemá zcela průzračnou strukturu. Zavedením prostředníka je pak možné celou situaci docela výrazně zprůzračnit.

– Komunikující objekty je třeba použít samostatně jinde

Druhou situací, kdy je použití prostředníka výhodné, je komunikace objektů, které bys potřeboval samostatně použít v jiném programu. Kdyby byly tyto objekty přímo napojeny na jiné objekty v tom prvém programu, těžko by se přenášely. Zavedením prostředníka vazby zpřetrháš a můžeš objekt přenést s tím, že mu v cílovém programu připravíš podobného prostředníka.

Výhoda: komunikaci lze upravit zavedením dceřiné třídy

Zavedení prostředníka má ještě jednu výhodu: potřebuješ-li nějak upravit chování celé skupiny komunikujících objektů, stačí v řadě případů definovat pouze dceřinou třídu prostředníka a v ní příslušné změny naprogramovat. Žádný z komunikujících objektů se o tom vůbec nemusí dozvědět. Jedinou změnou, kterou by objekty mohly zaznamenat, je, že dostanou odkaz na jiného prostředníka, na nějž se mají se svými požadavky obracet.

Zaslání zprávy = volání metody

Implementace

496. Řekl bych, že jsem princip a účel tohoto návrhového vzoru pochopil. Prozradíš mi ještě nějaké tipy k jeho implementaci?

Někdy je vhodné, aby vysílač předal odkaz na sebe

Zaslání zprávy prostředníkovi se realizuje jako volání jeho metody, jejímiž parametry jsou součásti zprávy.

Jednou z možností, jak předávání zpráv precizovat, je předat prostředníkovi v parametru odkaz na sebe. Prostředník si pak zjistí, které objekty by zpráva od daného objektu mohla zajímat, a obešle je.

Zadání adresáta nebývá potřeba

Druhou možností je zadávat mezi parametry i případné adresáty, ale to se většinou nepoužívá, protože bývají většinou známi a zřejmí.

Při implementaci se často používá vzor *Pozorovatel*

Jak jsem již řekl, při implementaci prostředníka se velmi často používá návrhový vzor *Pozorovatel*. Prostředník se od někoho dozví, že se stalo něco, co by mohlo přihlášené pozorovatele zajímat, a všem přihlášeným pozorovatelům proto rozešle zprávu o nastalé události.

Příklad

497. Copak sis pro mne připravil u tohoto návrhového vzoru za příklad?

Třída `SprávceSouborů`

O jednom příkladu jsem ti již říkal – je jím knihovna *Tvary*, v níž instance třídy `SprávceSouborů` slouží do jisté míry také jako prostředník. Kdykoliv někdo dospěje k názoru, že se vzhled plátna změnil (např. proto, že daný objekt změnil svoji pozici nebo rozměr), pošle správci plátna zprávu `překresli()` a tím zároveň nepřímo posílá doporučení k překreslení i ostatním zobrazeným objektům.

498. O tvarech jsi se zmiňoval již na konci minulé kapitoly. Neměl bys ještě něco dalšího?

Popis funkce příkladu

Ukážu ti prográmek, který používám při výkladu tvorby GUI za pomoci knihovny *swing*. Je to jednoduchá aplikace, která se podívá do zadané složky a nabídne ti zobrazení všech grafických souborů (přesněji souborů s příponami .gif, .jpg, .jpeg či .png), které tam najde.

Jak jsem říkal, příklad slouží k výuce tvorby GUI, takže názvy nalezených souborů jsou pak uvedeny v dialogovém okně jako popisky tlačítek, stavy přepínače, položky rozbalovacího seznamu a příkazy nabídky. Po zadání požadovaného obrázku prostřednictvím libovolného z nabídnutých ovládacích prvků program daný obrázek nakreslí.

Vlastnosti skupin

Všechny tyto skupinové ovládací prvky mají jedno společné: zadáš-li kterýkoliv prvek ze skupiny, nastavení dříve zadaného prvku se zruší a příznak nastavení (stisknuté tlačítko, označení stavu přepínače, zobrazení vybrané položky v seznamu) se aplikuje na zadaný prvek.

Úkol

Tím, že volby jsou v několika zdánlivě nesouvisejících skupinách, je třeba zabezpečit, aby se zadání realizované pomocí jedné skupiny ovládacích prvků promítlo i do ostatních skupin, tj. aby se po zadání vybraného obrázku v seznamu promáčklo jeho tlačítko a aby se příslušně nastavil stav přepínačů v okně i v jeho nabídce.

Použité řešení

Aplikace řeší tento problém tak, že definuje vnořenou třídu `Sada`, jejíž instance obsahují všechny ovládací prvky, jejichž aktivace zabezpečí vykreslení daného obrázku. Každý z těchto prvků patří do jiné skupiny. Všechny ovládací prvky sdílí jednoho společného posluchače událostí, který zjistí, jaký soubor chce uživatel zobrazit, a předá tuto informaci všem skupinám.

499. To je ten příklad, o němž ses zmiňoval, když jsem se ptal, zda může být jeden pozorovatel přihlášen u několika pozorovaných objektů?

Pozorovatel přihlášený u více pozorovaných

Ano. Třída `SpolečnýPosluchač` má jedinou instanci, která poslouchá za všechny přihlášené – to je ten společný posluchač, o němž jsem před chvilkou hovořil. Tato instance funguje jako prostředník, přes nějž libovolný ovládací prvek oznamuje svým kolegům ze stejné sady, že tento prvek je vybrán, a že se proto mají i jeho kolegové nastavit jako vybraní.

Výpis 29.1: Definice třídy `GUI_OBR` zobrazující obrázky ze zadané složky

```
package rup.česky.vzory._29_prostředník;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FilenameFilter;
import java.util.*;
import javax.swing.*;


/*****************************************************************************
 * Instance třídy GUI_OBR představují grafické aplikace pro testování práce
 * s grafikou, obrázky a posluchači událostí.
 * Tato verze ukazuje všechny obrázky v zadané složce a přidává možnost
 * zadávat tisknuté obrázky z rozbalovacího seznamu nebo přepínače.
 */
public class GUI_OBR
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ============================================

    private JFrame    okno;         //Aplikační okno.
```

```
    private JMenuBar  mnbNabídka;   //Hlavní nabídka
    private JLabel    lblObrázek;   //Vykreslovaný obrázek

    //Mapa obrázků a s nimi spojených ovládacích prvků indexovaná jejich názvy
    private Map<String, Sada> sady =
                                new LinkedHashMap<String,Sada>();

    //Společný posluchač pro všechny přepínané ovládací prvky
    private ActionListener společnýPosluchač = new SpolečnýPosluchač();

    //Skupiny jednotlivých přepínacích ovládacích prvků
    private ButtonGroup    bgrPovely   = new ButtonGroup(),
                           bgrPřepnutí = new ButtonGroup(),
                           bgrTlačítka = new ButtonGroup();

    private JComboBox comboBox = new JComboBox();


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /*****************************************************************************
     * Vytvoří instanci aplikace, která ze zadané složky zjistí všechny
     * soubory se zobrazitelnými obrázky a připraví aplikační okno
     * umožňující zadat zobrazení obrázků různými způsoby.
     */
    private GUI_OBR( File složka )
    {
        okno = new JFrame( "Zkouška" );     //Vytvoří aplikační okno
        okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Připraví sady s obrázkem a jeho volicími prvky
        vytvořSady( složka );

        hlavníNabídka();    //Vytvoří hlavní nabídku

        panelComboBox();    //(S) Přidá panel s rozbalovacím seznamem
        panelTlačítek();         //(Z) Přidá panel tlačítek
        panelPřepínačů();   //(C) Přidá panel přepínače
                            //(V) je rezervován pro žurnál
        panelObrázků();     //(J) Vytvoří panel s obrázky

        okno.pack();        //Optimalizuje velikost vytvořeného okna
        okno.setVisible( true );
    }


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ==========================================

    /** Množina přípon označujících formáty grafických souborů,
     *  které umí Java nakreslit.
     *  Atribut je deklarován až zde, protože s ním pracuje pouze
     *  následující metoda a nikdo jiný. Definice tohoto atributu nahrazuje
     *  definice statických lokálních proměnných z jiných jazyků.
     *
     *  Protože přípony jsou statickým atributem, je nutno je naplnit
     *  statickou metodou v deklaraci nebo
     *  ve statickém inicializačním bloku - to je použito zde. */
    private static final Set<String> přípony = new HashSet<String>();
```

```
    static {
        přípony.add( "png" );
        přípony.add( "gif" );
        přípony.add( "jpg" );
        přípony.add( "jpeg" );
    }
    /*************************************************************************
     * Vrátí seznam všech souborů s příponami zadanými v atributu
     * <code>přípony</code>, které se nachází v zadané složce.
     */
    private static File[] zjistiSoubory( File složka )
    {
        File[] ret = složka.listFiles( new
            //Filtr vybírající soubory, které bude program umět nakreslit
            FilenameFilter()
            {
                public boolean accept(File dir, String name) {
                    String přípona = name.substring( name.lastIndexOf('.')+1 )
                                          .toLowerCase();
                    //Akceptujeme pouze soubory s příponami ze zadané množiny
                    return přípony.contains( přípona );
                }
            }
        );
        System.out.println( "Soubory:" );
        for( File f : ret )
            System.out.println( "   " + f );
        return ret;
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /*************************************************************************
     * Připraví v aplikačním okně hlavní nabídku (nabídkovou lištu).
     */
    private void hlavníNabídka()
    {
        mnbNabídka = new JMenuBar();    //Vytvoření prázdné nabídkové lišty
        připravNabídkuObrázky();
//        připravNabídkuNápověda();
        okno.setJMenuBar( mnbNabídka ); //Přidání nabídkové lišty do okna
    }


    /*************************************************************************
     * Vytvoří rozbalovací seznam s názvy obrázků určených k zobrazení
     * a umístí jej do severního panelu aplikačního okna.
     */
    private void panelComboBox()
    {
        for( Sada s : sady.values() )  {
            comboBox.addItem( "S: " + s.název );
        }
        comboBox.addActionListener( společnýPosluchač );
        okno.add( comboBox, BorderLayout.NORTH );
    }
```

```
/*****************************************************************************
 * Připraví do jižního panelu aplikačního okna panel,
 * v němž se budou zobrazovat jednotlivé obrázky.
 */
private void panelObrázků()
{
    //Zavádíme pro obrázek vlastní panel,
    //aby byly malé obrázky vycentrovány
    JPanel panel = new JPanel( new FlowLayout( FlowLayout.CENTER ) );
    lblObrázek = new JLabel();         //Obrázek bude uložen v návěští
    panel.add( lblObrázek );
    okno .add( panel, BorderLayout.SOUTH );
}


/*****************************************************************************
 * Vytvoří přepínač, jehož polohy budou odpovídat jednotlivým obrázkům,
 * a umístí jej do centrálního panelu aplikačního okna.
 */
private void panelPřepínačů()
{
    JPanel jpanel = new JPanel( new FlowLayout( FlowLayout.CENTER ) );
    Box bpanel = Box.createVerticalBox();
    for( Sada s : sady.values() )  {
        bgrPřepnutí.add( s.přepnutí );
        bpanel     .add( s.přepnutí );
    }
    jpanel.add( bpanel );
    okno.add( jpanel, BorderLayout.CENTER );
}


/*****************************************************************************
 * Připraví tlačítka, jejichž popisky budou odpovídat jednotlivým obrázkům,
 * a umístí je do západního panelu aplikačního okna.
 */
private void panelTlačítek()
{
    Box panel = Box.createVerticalBox();
    for( Sada s : sady.values() )  {
        bgrTlačítka.add( s.tlačítko );
        panel      .add( s.tlačítko );
    }
    okno.add( panel, BorderLayout.WEST );
}


/*****************************************************************************
 * Připraví položku hlavní nabídky, jejíž povely budou odpovídat
 * jednotlivým obrázkům.
 */
private void připravNabídkuObrázky()
{
    JMenu mnuObrázky = new JMenu( "Obrázky" );   //Položka v hlavní nabídce
    for( Sada s : sady.values() ) {
        JRadioButtonMenuItem rbmi = s.povel;
        bgrPovely .add( rbmi );
```

```
            mnuObrázky.add( rbmi );
        }
        mnbNabídka.add( mnuObrázky );
    }


    /*****************************************************************************
     * Nechá zjistit zobrazitelné soubory v adresáři a ke každému z nich
     * vytvoří sadu obsahující ikonu obrázku a ovládací prvky, které
     * spustí její zobrazení: tlačítko, položku seznamu, položku přepínače,
     * položku hlavní nabídky
     */
    private void vytvořSady( File složka )
    {
        new Sada();
        File[] soubory = zjistiSoubory( složka );
        for( File soubor : soubory ) {
            new Sada( soubor );
        }
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    /*****************************************************************************
     * Instance třídy SpolečnýPosluchač představuje společného posluchače
     * všech prvků ovládajících zobrazování obrázků.
     */
    private class SpolečnýPosluchač implements ActionListener
    {
        /*************************************************************************
         * {@inheritDoc}
         * @param ae
         */
        public void actionPerformed( ActionEvent ae )
        {
            String zdroj0 = ae.getActionCommand();
            if( zdroj0.equals( "comboBoxChanged" ) )
                zdroj0 = (String)comboBox.getSelectedItem();
            String zdroj = zdroj0.substring( 3 );
            if( !sady.containsKey( zdroj ) )
                throw new RuntimeException( "\nNeznámá událost: " + ae );
            Sada sada = sady.get( zdroj );
            lblObrázek.setIcon( sada.obrázek );
            sada.povel   .getModel().setSelected( true );
            sada.přepnutí.getModel().setSelected( true );
            sada.tlačítko.getModel().setSelected( true );

            comboBox.removeActionListener( this );
            comboBox.getModel().setSelectedItem( zdroj );
            comboBox.addActionListener( this );

            okno.pack();
            lblObrázek.repaint();
        }
     }
```

```
/*****************************************************************************
 * Instance třídy Sada jsou jednoduché přepravky
 * sloužící ke sdružení souvisejících informací a objektů.
 * Třída musí být vnitřní, protože přihlašuje ke všem prvkům
 * společného posluchače, který je atributem instance.
 */
private class Sada
{
    final String               název;
    final Icon                 obrázek;
    final JToggleButton        tlačítko;
    final JRadioButtonMenuItem povel;
    final JRadioButton         přepnutí;


    /*************************************************************************
     * Vytvoří sadu odpovídající počátečnímu prázdnému obrázku.
     */
    Sada() {
        this( "NIC", new ImageIcon() );
    }

    /*************************************************************************
     * Vytvoří sadu odpovídající obrázku uloženému v zadaném souboru.
     * @param  soubor  Soubor s obrázkem
     */
    Sada( File soubor ) {
        this( soubor.getName().
                    substring( 0, soubor.getName().lastIndexOf('.') ),
              new ImageIcon( soubor.getAbsolutePath() )
        );
    }

    /*************************************************************************
     * Vytvoří novou sadu se zadaným názvem a obrázkem.
     * K této dvojici vytvoří všechny potřebné ovládací prvky
     * pro spuštění zobrazení daného obrázku.
     *
     * @param  název    Název obrázku
     * @param  obrázek  Obrázek určený k zobrazování
     */
    private Sada( String název, Icon obrázek )
    {
        this.název   = název;
        this.obrázek = obrázek;

        tlačítko = new JToggleButton( "T: " + název );
        tlačítko.addActionListener( společnýPosluchač );

        povel = new JRadioButtonMenuItem( "N: " + název );
        povel.addActionListener( společnýPosluchač );

        přepnutí = new JRadioButton( "P: " + název );
        přepnutí.addActionListener( společnýPosluchač );

        sady.put( název, this );
    }
}
```

```
//== TESTY A METODA MAIN ======================================================
    /*****************************************************************************
     * Testovací metoda očekávající v parametru název složky, v níž jsou
     * umístěny soubory s obrázky. Nebude-li složka zadána, bude ji program
     * hledat v podložce OBR složky, v níž se nachází class soubor třídy.
     *

     * @param args  Parametry příkazového řádku s názvem složky

     */
     public static void test( String... args ) {
         File složka;
         if( args.length > 0 )
             složka = new File( args[0] );
         else
            try {
//                 složka = new File( GUI_OBR.class.getResource( "Zdroje" )
//                                                  .toURI() );
                java.net.URL url = GUI_OBR.class.getResource("Zdroje");
                java.net.URI uri = url.toURI();
                složka = new File( uri );
            } catch( Exception ex ) {
                throw new RuntimeException( ex );
            }

         new GUI_OBR( složka );
     }
    /** @param args  Parametry příkazového řádku. */
     public static void main( String[] args )  {  test( args );  }
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Prostředník* doporučuje zjednodušit vzájemnou komunikaci mezi mnoha objekty zavedením prostředníka, na kterého se budou všichni obracet a který jejich zprávy předá požadovanému adresátu.
- Pro přeposílání obdržených zpráv bývá u prostředníků často použit návrhový vzor *Pozorovatel*.
- Návrhový vzor *Prostředník* je výhodné aplikovat:
  - je-li vzájemná komunikace objektů složitá nebo
  - je-li třeba použít komunikující objekty také samostatně.
- Oddělením komunikujících objektů docílíme toho, že je můžeme měnit nezávisle na sobě.
- Modifikaci režimu komunikace lze zabezpečit např. definováním dceřiné třídy prostředníka.
- Posílaná zpráva je definována jako metoda prostředníka, jejímiž parametry jsou součásti zprávy.
- Součástí zprávy posílané prostředníku může být i odkaz na příjemce či odesilatele.
- Návrhový vzor *Prostředník* patří mezi vzory uvedené v GoF.

ČÁST 7

Já se přizpůsobím

Tato část uzavírá celou knihu probráním návrhových vzorů, které předem počítají s možnými budoucími změnami zadání a definují architekturu aplikace a jejich částí tak, aby bylo možno případné další modifikace realizovat co nejsnadněji.

KAPITOLA 30

Příště to může být jinak (Most – Bridge)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Zavedením rozhraní odděluje abstrakci zprostředkovávající nějakou službu od implementace této služby, takže lze obě dvě měnit nezávisle na sobě.

[1] **Definice v GoF:** Decouple an abstraction from its implementation so that the two can vary independently. – Odděluje abstrakci od implementace, takže lze obě dvě měnit nezávisle na sobě.

Účel

500. Musím se přiznat, že té stručné charakteristice vzoru vůbec nerozumím.

Jestli tě to potěší, nejsi sám, koho tato formulace zaskočila. Když jsem ji četl poprvé v GoF (tam je ještě o něco kryptičtější – viz poznámka pod čarou na str. 399), měl jsem naprosto stejný pocit. Říkal jsem si, že význam každého slova znám, ale větu naprosto nechápu. Když jsem si pak četl dále příslušnou kapitolu o tomto vzoru, chvíli mi trvalo, než jsem terminologii autorů akceptoval.

Později jsem pochopil, že nejsem sám. Podobný problém měli evidentně i autoři mnohých učebnic. Někteří se k tomu dokonce přiznali (např. autor skvělé příručky [18]), jiní (např. autor neméně dobré [17]) alespoň čtenářům vysvětlili, jak že to vlastní autoři GoF s tou abstrakcí a implementací mysleli.

Jak chápat charakteristiku

501. A jak to tedy mysleli?

Popisovali situaci, kdy klient komunikuje se třídou, která je pouze zprostředkovatelem nabízené služby. Mohli bychom říci, že je její abstrakcí (nemusí to být ale nutně abstraktní třída nebo rozhraní). Vlastní realizaci (tj. implementaci) služby má na starosti jiná třída. Odtud také plynou použité termíny:

Abstrakce je zprostředkovatel

Implementace je realizátor

- třídu, jejíž objekty zprostředkovávají služby, označili jako *abstrakci,*
- třídu, která tyto služby doopravdy poskytuje, označili jako *implementaci.*

Protože řada autorů ve svých textech týkajících se tohoto návrhového vzoru oba termíny bez dalšího vysvětlení používá, pokusím se vám je alespoň částečně vštípit a budu je v dalším textu používat i v této kapitole. Aby vám však bylo vždy jasné, že se neodvolávám na běžný význam těchto slov, ale na jejich terminologický význam související s návrhovým vzorem *Most*, tak budu při těchto „připomenutích" uvádět oba termíny kurzivou.

Obě třídy jsou odděleny rozhraním

Pod oddělením těchto tříd si pak musíš představit to, že *abstrakce* nebude volat metody *implementace* přímo, ale bude se na *implementaci* obracet prostřednictvím nějakého rozhraní, které bude konkrétní *implementaci* zastupovat, resp. které bude ona implementovat.

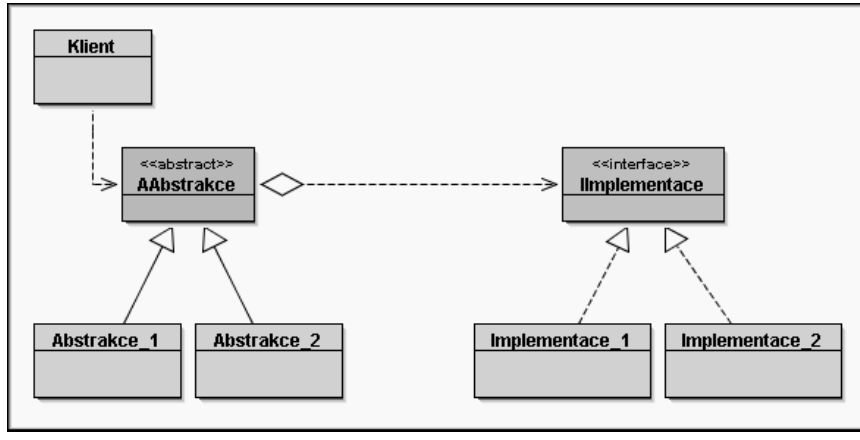
502. Už to začínám pomaličku chápat. Tím, že se *abstrakce* neobrací přímo na *implementaci*, ale pouze na její rozhraní, tak ji nijak neovlivní, když vyměním jednu *implementaci* za jinou.

Správně. To je také hlavní důvod aplikace tohoto návrhového vzoru. Je to vlastně realizovaná zásada, že nemáme programovat proti konkrétní *implementaci*, ale pouze proti jejímu rozhraní.

Když ti zobrazím diagram tříd s několika *abstrakcemi*, kterým navíc zavedu společného rodiče, a několika *implementacemi* schovanými za společným rozhraním (viz obr. 30.1), mohl by ti při troše představivosti připomínat most.

Definice několika verzí abstrakce schovaných za společným rozhraním však není typická, naopak, je spíše výjimečná. Je to vlastně aplikace návrhového vzoru *Zástup-*

ce na třídu označovanou jako *abstrakce*. Doplnil jsem ji do diagramu pouze proto, aby pak obrázek více připomínal most.



Obrázek 30.1
Diagram tříd charakterizující návrhový vzor Most

503. Takže ona *implementace* je nyní vlastně za dvěma rozhraními: prvním je *abstrakce* a druhým je rozhraní vložené mezi *abstrakci* a *implementaci*.

Já bych to tak neviděl. *Abstrakce* zde většinou funguje jako adaptér a konvertuje služby nabízené *implementací* do podoby, která je pro klienta mnohem výhodnější. Druhé rozhraní (na obrázku `IImplementace`) pak abstrakci a její implementaci „osvobozuje“ od vzájemné závislosti.

504. Kde se v praxi tento vzor uplatní.

Uplatnění: návrh ovladačů

Typické uplatnění je např. při návrhu ovladačů. Tvá třída (klient) potřebuje komunikovat s nějakými zařízeními a činí tak prostřednictvím ovladače. Pro tento účel definuješ obecný ovladač (*abstrakce*), jenž komunikuje s konkrétním ovladačem daného zařízení (*implementace*). Je-li komunikace s ovladačem realizována pomocí návrhového vzoru *Most*, můžeš velice jednoduše ovladač inovovat anebo jej při výměně zařízení dokonce vyměnit za jiný, aniž bys musel nějak výrazněji zasahovat do své aplikace (klienta).

505. Mohl bys uvést nějaký příklad?

Příklad: JDBC

Typickými, často uváděnými zástupci jsou např. ovladače JDBC zprostředkovávající komunikaci mezi tvojí aplikací a nějakou externí databází. V tomto případě je *abstrakcí* objekt typu `Connection`, který se tváří, že tě k databázi připojí a zprostředkuje ti s ní komunikaci, ale skutečným realizátorem (*implementací*) této komunikace je ovladač, který musíš nejprve zprovoznit, aby se na něj mohl objekt typu `Connection` začít obracet.

Díky tomu, že při implementaci tohoto mechanismu byl použit návrhový vzor *Most*, můžeš při každém běhu programu použít jinou databázi Máš-li několik ekvivalentních databází, můžeš např. v příkazovém řádku své aplikace zadat název ovladače, *abstrakce* (tj. objekt `Connection`) se jeho prostřednictvím k databázi připojí, a už jedeš.

Když to hodně zjednoduším a vynechám kontroly, mohla by hlavní metoda tvé aplikace vypadat např. následovně:

```
public static void main( String[] args ) {
    String driver= args[0];   //Třída ovladače
    String url   = args[1];   //Adresa databáze
    String jméno = args[2];   //Uživatelské jméno
    String heslo = args[3];   //Uživatelské heslo
    Class.forName(driver);    //Zprovoznění ovladače
    Connection con = DriverManager.getConnection(url, jméno, heslo);
    MojeAplikace.start( con );
}
```

506. Jestli jsem to dobře pochopil, tak jedním z důvodů použití vzoru může být i to, že budeme chtít na počátku programu určit, jaká implementace bude tentokrát ta pravá, a tu pro abstrakci připravit, aniž by mohla zjistit, že při každém spuštění programu pracuje s jinou implementací.

Vzor umožňuje definovat implementaci v konfiguračním souboru

Správně, bystrej kluk. Při použití návrhového vzoru *Most* můžeme tu pravou implementaci definovat např. v konfiguračním souboru. Změní-li se podmínky nebo získá-li uživatel třídu, která bude v jeho aplikaci pracovat efektivněji, stačí vyměnit informace v konfiguračním souboru a při příštím spuštění bude abstrakce komunikovat s novou implementací.

507. A co kdybych se rozhodl změnit implementaci za chodu programu?

Změna implementace za chodu

Takovéto situace řeší jiný návrhový vzor označovaný jako *Strategie*, který budeme probírat v kapitole *Vyberte si, jak to chcete (Strategie – Strategy)* na straně 415.

Implementace

508. K čemu je *Most* dobrý a jak se používá, jsi mi již vysvětlil. Odhaduji, že jeho implementace bude relativně triviální.

Postup

Bude, ostatně jako u většiny vzorů:

– rozhraní *implementace*

– definice *implementace*

– jak klient získá odkaz na server

– používat služeb

1. Definuješ rozhraní charakterizující vlastnosti dané *implementace*. Nemusíš je vždy definovat jako `interface`, někdy je výhodné použít abstraktní třídu.
2. Definuješ *implementaci* jako třídu implementující toto rozhraní, resp. jako dceřinou třídu dané abstraktní třídy.
3. Definuješ způsob, jakým *abstrakce* získá odkaz na instanci tohoto rozhraní, tj. na instanci konkrétní *implementace*.
4. Používáš v *abstrakci* služeb dané *implementace* oslovované prostřednictvím jejího rozhraní.

Při tomto řešení můžeš poměrně svobodně vyměňovat v *abstrakci* používané *implementace* a současně používat konkrétní *implementaci* v různých *abstrakcích*.

Příklad

509. Takže už jen zbývá ukázat vše na nějakém průzračném příkladu.

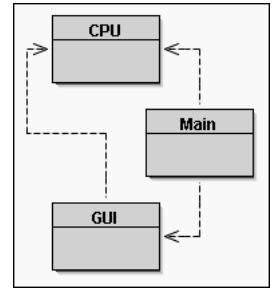
Kalkulačka – typické řešení:

Jeden bych tu měl. Některé učebnice programování (z těch v seznamu je to např. [39]) a kurzy používají při výkladu příklad kalkulačky, na němž si mají žáci procvičit, co se naučili. Všechny učebnice a kurzy, s nimiž jsem se setkal, používaly řešení se dvěma či třemi třídami (viz obr. 30.2):

– GUI
- Třída reprezentující uživatelské rozhraní (GUI) výsledné kalkulačky – tu připravil vyučující.

– CPU
- Třída reprezentující CPU – tu měli za úkol naprogramovat studenti.

– Hlavní třída aplikace
- Případně ještě hlavní třída aplikace, která pouze odstartovala její běh.



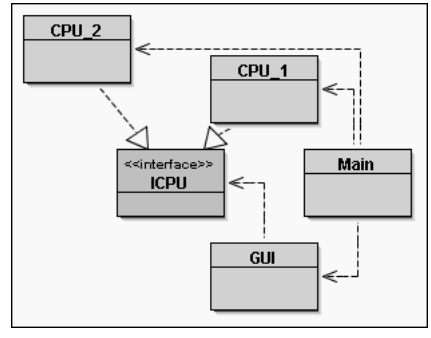
Obrázek 30.2
Diagram tříd klasicky koncipovaného kalkulátoru

Nevýhody

K tomuto řešení se museli vyučující uchýlit proto, že se drželi zaužívané praxe, při níž se rozhraní probírá až na konci kurzu. Jeho nevýhodou bylo, že chtěl-li vyučující připravit několik různých zadání, musel pro každé z nich definovat jeho vlastní verzi GUI. A chtěl-li pak po studentovi na zkoušce, aby své řešení jakkoliv modifikoval, musel na příslušnou modifikaci opět připravit i své GUI.

Využití *Mostu*

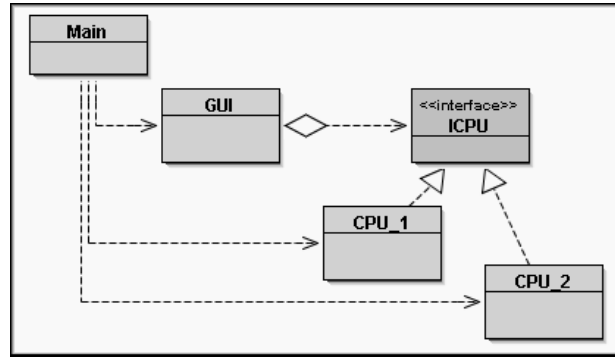
S využitím *Mostu* bychom mohli řešení upravit: definovat dostatečně univerzální rozhraní pro procesorové jednotky (*implementace*) a nechat toto rozhraní každého studenta implementovat. Všechny CPU by pak mohla obsluhovat jedna třída realizující potřebné GUI (*abstrakce*) – viz obrázek 30.3.



Obrázek 30.3
Diagram tříd řešení využívajícího vzor Most

510. Počkej, počkej. Tento obrázek se obrázku 30.1 vůbec nepodobá. A chybí mi v něm klient. Překresli jej, ať vidím, že je to doopravdy *Most*.

No s trochou prostorové představivosti bys to v něm mohl vidět, ale budiž. Obrázek 30.4 ti už připadá „mostovější"? Kdybys měl problémy se šipkami závislostí vedoucích od třídy `Main` k třídám `CPU_1` a `CPU_2`, tak ty tam jsou proto, že třída `Main` od nich musí nejprve získat jejich instanci, aby ji pak mohla předat konstruktoru třídy `GUI`.



Obrázek 30.4
Diagram tříd z obr. 30.3 překreslený tak, aby více připomínal diagram z obrázku 30.1

Třídu `Main` jsem tam ale nakreslil pouze proto, aby ti připomněla původní obrázek návrhového vzoru *Most*. Skutečným klientem totiž není třída `Main`, ale uživatel, který s aplikací „rozmlouvá".

511. Teď už je to lepší. Toho společného rodiče abstrakcí si již domyslím.

Kdy má smysl schovat GUI za rohraní

V první etapě je třída `GUI` opravdu osamocená. Pokud bychom tento příklad použili při výuce, tak bychom mohli v první etapě dát studentům za úkol definovat vlastní CPU, avšak v druhé etapě, až budou trochu pokročilejší, naučit je definovat vlastní GUI. Pak bychom mohli všechna GUI „schovat" za společné rozhraní nazvané např. `IGUI` a obrázek by se začal podobat tomu původnímu ještě víc.

512. Už to dál nerozebírej a pochlub zdrojovým kódem.

Rozhraní `ICPU`

Rozhraní `ICPU` předepisuje implementaci tří metod – viz výpis 30.1.

Výpis 30.1: Definice rozhraní `ICPU`

```
package rup.česky.vzory._30_most;

/*******************************************************************************
 * Rozhraní ICPU definuje metody, které musí poskytovat každá CPU
 * realizující výpočetní jednotku kalkulačky
 * a spolupracující s instancí rozhraní {@link IGUI}.
 *
 * @author  Rudolf Pecinovský
 * @version 0.00.000, 0.0.2006
 */
public interface ICPU
{
```

```
    /*****************************************************************************
     * Vrátí přepravku s požadovaným rozměrem klávesnice (počet řádků a sloupců)
     * a bodovou velikostí tlačítek.
     *
     * @return Požadovaná přepravka
     */
    public Klávesnice getRozměry();


    /*****************************************************************************
     * Převezme text popisku na stisknutém tlačítku a vrátí text,
     * který má být zobrazen v reakci na stisk daného tlačítka.
     * Objektu smí být zadán pouze některý z příkazů, které předal
     * jako návratovou hodnotu po volání metody
     * {@link #getPopisky()}.
     *
     * @param popisek Popisek na stisknutém tlačítku
     * @return Obsah displeje po provedení příkazu vyvolaného stiskem tlačítka
     */
    public String stisknuto( String popisek );

}
```

První slouží k tomu, aby se mohlo grafické rozhraní správně připravit, druhá již ovlivňuje vlastní chod celé aplikace.

Třída Klávesnice

Informací, které vrací metoda `getRozměry()`, je přeci jenom více, takže je pro ně definována speciální přepravka – třída `Klávesnice` – viz výpis 30.2. (Tato třída není zobrazena v diagramu na obr. 30.3, protože je pouze pomocná a nesouvisí přímo s návrhovým vzorem.)

513. To je třída, o níž ses zmiňoval v kapitole o přepravkách, když jsem se ptal, jak udělat neměnnou přepravku, která obsahuje pole.

Nemodifikovatelné popisky

Správně. Tato přepravka obsahuje dvourozměrné pole popisků kláves. Aby nebylo možno měnit tyto popisky za běhu, jsou přístupné pouze prostřednictvím metody `popisek(int,int)`, které zadáš řádek a sloupec a ona ti vrátí popisek na klávese v zadaném řádku a sloupci.

Výhodou tohoto řešení je, že když takovou přepravku někomu poskytneš, může se na její obsah spolehnout a nemusí si je nikam kopírovat pro případ, že by jej někdo v budoucnu změnil.

Výpis 30.2: Definice třídy `Klávesnice`

```
package rup.česky.vzory._30_most;

import rup.česky.debug.ToString;

/*******************************************************************************
 * Přepravka uchovávající rozměry klávesnice předpokládané jejím CPU.
 */
public class Klávesnice
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================
```

```
    public final int sloupců;        //Počet sloupců kláves
    public final int řádků;
    public final int znakůDispleje;

    //Pole musí být soukromé, aby hodnoty zůstaly neměnné
    private final String[][] popisky;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří přepravku se všemi hodnotami nulovými,
     * tj. s definicí všech rozměrů ponechanou na GUI.
     *
     * @param popisky  Popisky kláves v řádku a sloupci klávesnice
     *                 odpovídající umístění textu v poli
     */
    public Klávesnice( String[][] popisky ) {
        this( 0, popisky );
    }


    /*****************************************************************************
     * Vytvoří přepravku, v níž jsou uloženy požadované rozměry
     * klávesového pole spolu s rozměry tlačítek a velikostí displeje.
     * Zadáním nulového rozměru žádáme GUI, aby si jej dopočítalo samo.
     *
     * @param znakůDispleje  Požadovaný počet zobrazitelných znaků na displeji
     * @param popisky  Popisky kláves v řádku a sloupci klávesnice
     *                 odpovídající umístění textu v poli
     */
    public Klávesnice( int znakůDispleje, String[][] popisky )
    {
        if( (popisky == null)         ||  (popisky.length == 0)  ||
            (popisky[0].length == 0)  ||  (znakůDispleje < 0)    )
        {
            throw new IllegalArgumentException(
                      "Znaků displeje=" + znakůDispleje +
                      ToString.array2o( "\nPopisky", popisky ) );
        }
        this.znakůDispleje = znakůDispleje;

        this.řádků = popisky.length;
        int s = popisky[0].length;
        for( int r=1;   r < this.řádků;   r++ )
            if( s < popisky[r].length )
                s = popisky[r].length;
        this.sloupců = s;
        this.popisky = new String[řádků][sloupců];

        for( int r=0;   r < this.řádků;   r++ ) {
            System.arraycopy( popisky[r], 0,
                              this.popisky[r], 0, popisky[r].length );
            java.util.Arrays.fill( this.popisky[r], popisky[r].length,
                                   this.sloupců,    "" );
        }
//        System.out.println("\nKlávesnice: " + this );
    }
```

```
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Vrátí popisek na klávese v zadaném řádku a sloupci klávesnice.
     */
    public String popisek( int řádek, int sloupec ) {
        return popisky[řádek][sloupec];
    }


    /*****************************************************************************
     * Vrátí text s požadovaným počtem znaků na displeji
     * a popisky na jednotlivých klávesách
     */
    public String toString() {
        return ToString.array2o( "Klávesnice: Znaků displeje=" + znakůDispleje +
                                 ", Popisky ", popisky );
    }

}
```

514. Musím říct, že tato třída na obyčejnou přepravku nevypadá. Ten její konstruktor je nějak komplikovaný.

Klávesnice vykonává i pomocné práce

Tato přepravka vykonává ještě nějaké přípravné práce, které mají usnadnit práci instancím, jež budou vytvořenou přepravku používat. Konstruktor proto zjišťuje, jestli uživatel zadal použitelné pole s popisky, a pokud ano, tak toto pole dorovná do obdélníkového.

515. Kdo tyto přepravky používá?

Použití třídy Klávesnice

GUI požádá CPU o její požadavky na vytvořené dialogové okno. CPU vrátí instanci přepravky `Klávesnice`, v níž zadá případný požadavek na počet znaků zobrazitelných na displeji a dvojrozměrné pole popisků na jednotlivé klávesy. Tím, že konstruktor `Klávesnice` doplní v případě potřeby tyto popisky prázdnými řetězci na obdélníkové pole, usnadní práci CPU, která pak může „bez přemýšlení“ sázet jednotlivé popisky jeden za druhým do příslušného panelu (přesněji předávat je jeho správci rozvržení).

516. GUI slouží v našem příkladu jako abstrakce.

Třída `GUI`

Ano, už jsem to říkal. Ale oproti typickým aplikacím tohoto návrhového vzoru nekomunikuje tato aplikace se zbytkem programu, ale přímo s uživatelem.

Výpis 30.3: Definice třídy `GUI`

```
package rup.česky.vzory._30_most;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import rup.česky.vzory._30_most.ICPU;
import rup.česky.vzory._30_most.Klávesnice;


/*******************************************************************************
 * Třída GUI definuje grafické uživatelské rozhraní
 * pro obecnou, předem neznámou kalkulačku.
 *
 * @author    Rudolf Pecinovský, Lubos Pavlicek
 * @version   0.00.000,  0.0.2003
 */
public class GUI
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final int ŠÍŘKA  = 50;   //Minimální šířkaKlávesy
panelTlacitek
    private static final int VÝŠKA  = 35;   //Minimální výškaKlávesy
panelTlacitek
    private static final int MEZERA =  5;   //Mezera mezi tlačítky
    private static final int DISPLEJ= 12;   //Implicitní rozměr displeje


//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================================

    private final ICPU       cpu;       //CPU, která bude obsluhována
    private final JFrame     okno;      //Aplikační okno
    private final JTextField displej;   //Displej kalkulačky

    private final JPanel panelTlacitek = new JPanel();;

    private final ActionListener obsluhaTlačítka = new
        ActionListener() {
            /** @param event  Událost, na kterou metoda reaguje */
            public void actionPerformed(ActionEvent event) {
                text = cpu.stisknuto( event.getActionCommand() );
                překresli();
            }
        };


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ===============================================

    private String text;            //Text zobrazovaný na displeji


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /***************************************************************************
     * Vytvoří grafické uživatelské rozhraní kalkulačky,
     * jejíž CPU dostane jako parametr.
     *
     * @param cpu  CPU kalkulačky, pro niž vytváří GUI
```

```
     */
    public GUI( ICPU cpu ) {
        this.cpu = cpu;
        this.text = "";

        displej = new JTextField();
        displej.setHorizontalAlignment(JTextField.RIGHT);
        displej.setEditable(false);
        displej.setBackground(Color.WHITE);

        JPanel panelDisplay = new JPanel();
        panelDisplay.add(displej);

        okno=new JFrame();

        okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        okno.getContentPane().add( panelDisplay, BorderLayout.NORTH );
        okno.getContentPane().add( panelTlacitek, BorderLayout.CENTER );

        nastavRozměryOkna();
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /****************************************************************************
     * Nastavuje rozměr klávesového pole spolu s rozměry tlačítek.
     * Zadáním nulového rozměru požádáme GUI, aby si jej dopočítalo samo.
     * Metoda je definována jako finální, aby byla použitelná v konstruktoru.
     */
    public final void nastavRozměryOkna() {
        okno.setVisible( false );
        Klávesnice k = cpu.getRozměry();

        displej.setColumns( Math.max( k.znakůDispleje, DISPLEJ ) );

        připravPanelTlačítek( k );
        okno.pack();
        okno.setVisible( true );
        překresli();
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /****************************************************************************
     * Připraví panel tlačítek, přičemž popisky tlačítek
     * zjistí dotazem na obsluhovanou CPU.
     */
    private void připravPanelTlačítek( Klávesnice k ) {
        panelTlacitek.removeAll();
        panelTlacitek.setLayout(
                        new GridLayout( k.řádků, k.sloupců, MEZERA, MEZERA ) );
        for (int r=0;   r < k.řádků;   r++) {
            for( int s=0;   s < k.sloupců;   s++ ) {
                String popisek = k.popisek( r, s );
                Component c;
                if( popisek.equals("") ) {
```

```
                    c = new JLabel( "" );
                } else {
                    JButton jb = new JButton( popisek );
                    jb.addActionListener(obsluhaTlačítka);
                    c = jb;
                }
                panelTlacitek.add( c );
            }
        }
    }

    /*************************************************************************
     * Nastaví nový text na displeji a tím iniciuje jeho překreslení.
     */
    private void překresli() {
        displej.setText( text );
    }
}
```

517. Ve zdrojovém kódu píšeš, že metoda `nastavRozměryOkna()` je definována jako finální, aby byla použitelná v konstruktoru. K tomu by ale přece stačilo, aby byla deklarována jako soukromá.

Proč je metoda `nastavRozměryOkna()` finální

To je pravda, ale v tomto případě je metoda připravena na rozšíření, při němž bude instance třídy pracovat s CPU, která bude měnit svoji povahu, a bude proto chtít jednou za čas zcela změnit vzhled kalkulačky.

Podrobněji si o této možnosti budeme povídat v příští kapitole. Zatím se smiř se tím, že potřebuji mít metodu definovanou jako veřejnou, a protože neplánuji tvorbu potomků, kteří by zrovna tuhle metodu překryli, definuji ji jako konečnou (jinými slovy: byl jsem líný definovat její tělo jako soukromou metodu, na niž se tato metoda odvolává). Kdybych si později svůj úmysl rozmyslel, upravil bych ji tak, jak to bývá v těchto případech obvyklé.

518. No dobrá. Tak už mi ukaž nějakou CPU.

Třída `Binární`

Ve výpisu 30.4 najdeš zdrojový kód jednoduché binární kalkulačky (nic jednodušší-ho jsem nevymyslel) definované ve třídě `Binární`. V doprovodných příkladech pak najdeš zdrojový kód třídy `Reálná`, která definuje kalkulačku pracující s reálnými čísly a nabízející mnohem bohatší paletu funkcí.

Binární kalkulačka, jejíž zdrojový text je přiložen, převádí stisky kláves okamžitě na čísla, která na displeji zobrazuje (pozor, záporná čísla jsou záměrně zobrazena přesně, jak jsou reprezentována v počítači, tj. jako dvojkový doplněk). Reálná kalkulačka, jejíž zdrojový kód je v balíku doprovodných programů, ukazuje jiný přístup k řešení: stisky číselných kláves převádí na příslušné znaky, z nichž skládá textový řetězec, jejž převádí na číslo až ve chvíli, kdy s ním potřebuje pracovat.

Obě CPU komunikují s uživatelem přes instance třídy `GUI`. Kdybys definoval nějakou svoji vlastní kalkulačku, která by implementovala rozhraní `ICPU`, mohla by nějaká instance `GUI` zprostředkovat komunikaci s uživatelem i jí.

Výpis 30.4: Definice třídy `Binarni`

```
package rup.česky.vzory._30_most;

/*******************************************************************************
 * Instance třídy Binární představují CPU jednoduché binární kalkulačky,
 * která umí pouze sčítat, odečítat a mazat naposledy zadaná čísla.
 */
public class Binární implements ICPU
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private int     minule  = 0;
    private int     nyní    = 0;
    private char    operace = 0;


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * {@inheritDoc} */
    public String příkaz( String popisek ) {
        //V následujícím přepínači jsou předvedeny různé možnosti řešení
        //reakce na vstup do konkrétní větve
        char povel = popisek.charAt(0);
        switch( povel )
        {
            //Několik návěští uvozujících společnou akci
            case '0':
            case '1':
                    nyní = 2*nyní + (povel-'0');
                break;

            //Zavolání funkce a přímé opuštění metody
            case '+':
            case '-':
                return plusMinus( povel );

            //Konkrétní akce následované příkazem break
            case 'C':
                nyní /= 2;
                break;

            //Zavolání funkce následované příkazem break
            case '=':
                spočti();
                break;

            //Nestandardní ukončení vyvoláním výjimky
            default:
                throw new IllegalArgumentException(
                        "Neplatná operace: " + popisek );
        }
        return Integer.toBinaryString( nyní );
    }


    /*******************************************************************************
```

```
     * {@inheritDoc} */
    public String[] getPopisky()
    {
        return new String[]  { "1", "+", "C",      "0", "-", "="  };
    }


    /**************************************************************************
     * {@inheritDoc} */
    public Klávesnice getRozměry() {
        return new Klávesnice();
    }

//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Připraví registry pro následující součet, resp. rozdíl hodnot.
     */
    private String plusMinus( char povel ) {
        if( operace != 0 )
            spočti();
        minule  = nyní;
        nyní    = 0;
        operace = povel;
        return Integer.toBinaryString( nyní );
    }


    /**************************************************************************
     * Aplikuje zapamatovanou operaci na zapamatované hodnoty.
     */
    private void spočti() {
        switch( operace )
        {
            case '+':  nyní = minule + nyní;   break;
            case '-':  nyní = minule - nyní;   break;
            default:   throw new IllegalArgumentException(
                                        "Neplatná operace: " + operace );
        }
        operace = 0;
    }


//== TESTY =============================================================================

    /**************************************************************************
     * Testovací metoda.
     */
    public static void test() {
        new GUI( new Binární() );
    }
    /** @param args Parametry příkazového řádku - nepoužité. */
    public static void main( String[] args )  {  test();  }
}
```

519. Ve stručné charakteristice je psáno, že návrhový vzor *Most* umožňuje nezávisle měnit abstrakci a implementaci. Prozraď mi, kdy by se mi mohla hodit záměna GUI z nějakého pádnějšího důvodu, než že to používané není dostatečně krásné.

Kdy je rozumné měnit abstrakci

Např. tehdy, pokud bys chtěl danou CPU testovat. V mém zadání je např. vedle rozhraní `ICPU` a třídy `GUI` definována testovací třída, která se vydává za GUI, posílá CPU simulované stisky kláves a kontroluje, že odpovědi CPU jsou korektní.

Student přinese své řešení, já na něj pošlu testovací třídu (na to řešení), ta se jej zeptá, které zadání řeší, a testovací třída pak otestuje, zda toto zadání vyřešil přesně podle požadavků.

Přiznám se, že jsem ve své lenoře došel tak daleko, že všechna odevzdaná řešení umístím do složek podle kroužků, resp. tříd, do nichž studenti patří, pak zavolám speciální testovací třídu, která projde všechny složky, zjistí, které soubory jsou class-soubory tříd implementujících rozhraní `IGUI`, příslušné třídy zavede a jejich instance výše popsaným způsobem otestuje.

Zdrojové kódy těchto testovacích tříd tu vypisovat nebudu, protože s daným návrhovým vzorem přímo nesouvisí, ale můžeš si je prohlédnout mezi doprovodnými programy.

Shrnutí – co jsme se naučili

- Návrhový vzor *Most* řeší situaci, kdy klient komunikuje se třídou, která je pouze zprostředkovatelem (*abstrakcí*) nabízené služby. Vlastní realizaci (tj. *implementaci*) služby má na starosti jiná třída.
- Návrhový vzor *Most* doporučuje vložit mezi *abstrakci* a *implementaci* rozhraní. Tím je od sebe oddělí a umožní je nezávisle na sobě měnit.
- Aplikace vzoru *Most* umožní definovat implementaci např. v konfiguračním souboru.
- Návrhový vzor *Most* se uplatňuje např. při návrhu ovladačů.
- Při implementaci tohoto vzoru je třeba:
  - definovat oddělující rozhraní,
  - definovat implementaci,
  - definovat, jak abstrakce získá odkaz na implementaci,
  - definovat klienta,
  - vše rozběhnout.
- Návrhový vzor *Most* patří mezi vzory uvedené v GoF.

KAPITOLA 31

Vyberte si, jak to chcete (Strategie – Strategy)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Definuje množinu vyměnitelných objektů (algoritmů) řešících zadanou úlohu (tj. objektů se stejným rozhraním) a umožňuje mezi nimi dynamicky přepínat.

[1] **Definice v GoF:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. – Definuje rodinu algoritmů, zapouzdří každý z nich a nastaví je jako záměnné. *Strategie* umožňuje klientům zaměňovat používané algoritmy.

Účel

520. Jestli jsem dobře pochopil stručnou charakteristiku, tak mi *Strategie* umožní ovlivnit to, jakým způsobem se bude úloha řešit.

K čemu je vzor *Strategie*

Dalo by se to tak říci, ale obecnější by bylo tvrzení, že návrhový vzor *Strategie* nám umožňuje jednoduše ovlivnit, co a jak se bude řešit. Navíc nám umožňuje tento výběr v budoucnu velice jednoduše rozšířit.

521. Jako obyčejně tě požádám o příklad.

Příklad: Diamanty

Možná si vzpomeneš na kapitolu *Příliš mnoho instancí (Muší váha – Flyweight)*, v níž jsme jako příklad vytvářeli hru s diamanty (výpisy kódu začínají na str. 175). Když jsme tento příklad řešili, měli různí studenti různé názory na to, jak by bylo vhodné řešit kontrolu vzniku trojic, uprázdnění desky a její setřesení.

Kdybychom chtěli navrhovaná řešení porovnat, mohli jsme je všechna „schovat za rozhraní“ a zkoušet, jak bude aplikace běhat s různými řešeními.

522. To mi připomíná návrhový vzor *Most*.

Rozdíl *Most* × *Strategie*

Máš pravdu. Princip je podobný. Rozdíl je v tom, že *Strategie* předpokládá, že se budou použité postupy měnit za chodu programu.

Příklad: Kalkulačka

Zkusím ti dát jiný příklad – a vezmu ho přímo od toho mostu. V kapitole *Příště to může být jinak (Most – Bridge)* jsme si jako příklad ukazovali program realizující kalkulačku. Díky aplikaci vzoru *Most* jsme mohli spustit pokaždé jinou kalkulačku. Kalkulačka, kterou jsme však spustili, měla své CPU neměnné po celou dobu běhu aplikace.

Výměna CPU za chodu

Teď si ale představ, že bych chtěl definovat několik různých CPU, např. jednu pro běžná čísla, druhou pro zlomky a třetí pro komplexní čísla. Tyhle CPU bych samozřejmě potřeboval měnit za chodu, obdobně jako to dělají kalkulačky, které si koupíš v krámě.

523. No, to je už zajímavější. A to je jediná odchylka od *Mostu*?

Strategie se zaměřuje na chování

Druhou odchylkou je to, že návrhový vzor *Most* hovoří o tom, jak vypadá architektura dané části aplikace, kdežto návrhový vzor *Strategie* se zaměřuje na to, jak se tato část aplikace chová.

Implementace

524. Dobře. Tak mi tedy ukaž, jak bych měl přemýšlet. Jak bych tedy mohl navrhnout onu kalkulačku, která bude za chodu dynamicky vyměňovat CPU?

Výměna CPU – marnotratná verze

Nejprve ti ukážu trochu marnotratnější řešení, které jsem použil ve svých kroužcích. Protože každý student musel stejně připravit komplexní CPU, tak každá CPU zpracovává vstup čísel po svém. Kdybys chtěl efektivnější řešení, vyčlenil bys zadávání a editaci čísel a jednotlivým CPU bys nechal na starosti pouze zpracování zadaných čísel. To si můžeš zkusit za domácí úkol.

525. No dobře, třeba to na tom jednodušším příkladu snáze pochopím.

To doufám. Takže náš základní problém je, jak zařídit, abys mohl v průběhu práce s kalkulačkou přepínat mezi jednotlivými CPU.

526. Na panelu kalkulačky bych definoval sadu tlačítek, jejichž stiskem bych zadal, v jakém režimu má kalkulačka pracovat.

Jasně, ale důležité je, aby se GUI nemusela o žádné přepínání starat. Aby pro ně bylo tlačítko jako tlačítko. Pošle jeho popisek CPU a ta mu prozradí, co má nakreslit na displej.

527. Jak ale může CPU přepínat mezi různými CPU?

Vyčleníme jednu speciální CPU, která bude v aplikaci fungovat jako přepínač. Ta bude odchytávat stisky tlačítek požadujících přechod do jiného režimu, tj. výměnu CPU. Ostatní tlačítka předá aktivní CPU obdobně, jako by jí je předalo GUI.

Výměna CPU – efektivnější verze – problémy

528. A v té efektivnější verzi by se mohla postarat o převod stisků číselných kláves na čísla.

Mohla, ale nebylo by to tak jednoduché, jak se tu snažíš naznačit. Muselo by se totiž změnit rozhraní, které musí implementovat jednotlivé CPU, aby byly schopny přijmout celé číslo a ne pouze jednotlivé číslice.

Kromě toho bys musel jednotce pro vstup čísel nějak oznámit, ve které části displeje se nachází zadávané číslo, protože některé z kalkulaček (např. zlomková, komplexní, vektorová, …) budou na displeji zobrazovat těch čísel několik.

529. No jo, no jo. Pochopil jsem, že by to bylo složitější, než jsem si na začátku myslel. Pokorně se vracím k plně funkčním CPU a jejich přepínači.

Přepínač CPU

Přepínač by byl naprosto jednoduchý: měl by atribut, v němž by uchovával odkaz na aktuální CPU, které by posílal popisky stisknutých tlačítek (samozřejmě s výjimkou těch přepínacích) a od které by přebíral texty na displej, které by beze změny předával `GUI`.

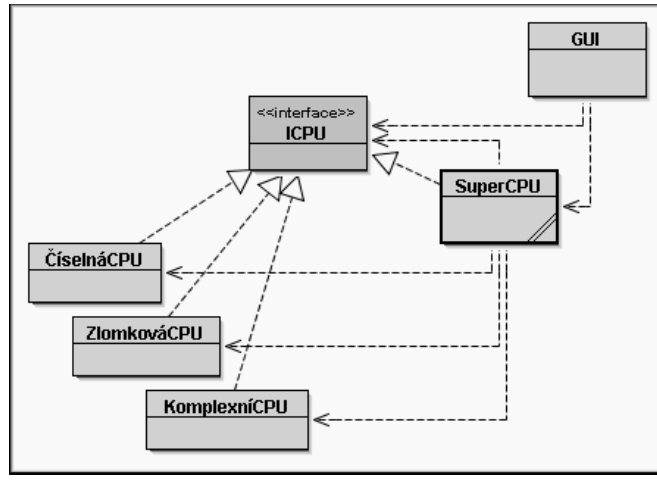
Při stisku přepínacího tlačítka by jenom vyměnil obsah tohoto atributu a následně by vše pokračovalo s jinou CPU.

530. Ten přepínač by se vlastně choval k jednotlivým CPU jako GUI.

Přepínač je současně CPU i GUI

Máš pravdu, byl by to takový hermafrodit. Vůči GUI by vystupoval jako taková super CPU „všechno vím, všechno znám“, kdežto vůči jednotlivým CPU by vystupoval jako GUI. Celou koncepci si můžeš prohlédnout na obrázku 31.1.

Chceš-li však získat představu o obecné podobě architektury návrhového vzoru *Strategie*, tak si odmysli to, že třída `SuperCPU` implementuje stejné rozhraní jako jednotlivé CPU a že toto rozhraní oslovuje i třída `GUI`, která se `SuperCPU` komunikuje a předává jí požadavky na změnu strategií zpracování, tj. na změnu CPU. To je pouze specialita našeho příkladu.



Obrázek 31.1
Koncepce kalkulačky schopné přepínat režimy

531. Když na to tak koukám, tak mi to připomíná návrhový vzor *Stav*. Jediný rozdíl je, že se jednotlivé stavy mezi sebou nepřepínají samy, ale přepíná je `SuperCPU`.

Podobnost se vzorem *Stav*

Návrhovému vzoru *Stav* (povídali jsme si o něm v kapitole *Příliš mnoho rozhodování (Stav – State)* na straně 221) je to opravdu blízko. Hlavní rozdíl ale spočívá v tom, že v návrhovém vzoru *Stav* jsou jednotlivé stavy vnitřní záležitostí vícestavového objektu a klient (v našem případě `GUI`) se o změně stavu ani nedozví. Naproti tomu v návrhovém vzoru *Strategie* se předpokládá, že si klient změnu algoritmu explicitně objedná.

532. Ještě jednou se vrátím k návrhovému vzoru *Stav*. Tehdy jsme si říkali, že místo rozhraní může být společným rodičem jednotlivých stavů i abstraktní třída. Mohl by jejich společným rodičem být ve *Strategii* přepínač?

Společný rodič jako abstraktní třída

Samozřejmě. Občas se to tak i používá a přepínač pak současně implementuje implicitní strategii, která se použije tehdy, nemá-li klient žádné speciální přání.

Společného rodiče definovaného jako abstraktní třídu využiješ např. tehdy, pokud by se tento návrhový vzor sloučil se *Šablonovou metodou*, která by byla definována v přepínači, a volal by metody, jež by jednotlivé strategie implementovaly po svém.

Nevýhody

Nevýhodou tohoto řešení je ale zbytečně těsná vazba jednotlivých strategií na přepínač. Přitom snahou moderního programování je právě všechny vazby co nejvíce uvolnit, aby případná změna jedné třídy neovlivnila příliš mnoho jiných tříd. Můžeš ale definovat společného abstraktního rodiče mimo přepínač a vazbu tak uvolnit.

533. Tak mne napadá, že při koncepci z obrázku 31.1 by šly nové CPU přidávat téměř za chodu. Pokud by je měl přepínač uloženy v nějakém dynamickém kontejneru, mohl by k nim kdykoliv přidat další.

To je pravda, i když jsem se s tím ještě nesetkal. Dokážu si ale představit, že by se to mohlo někdy hodit.

Příklad

534. Tak už mne nenapínej a ukaž mi, jak jsi naprogramoval tu `SuperCPU`.

Proč řešení neodpovídá obrázku 31.1

Nejprve bych měl poznamenat, že řešení, které tu předvedu, neodpovídá přesně obrázku 31.1, protože jsem chtěl maximálně využít hotových tříd, které jsem ti ukazoval jako příklad aplikace návrhového vzoru *Most* a jejichž zdrojové kódy najdeš ve výpisech 30.1 až 30.4, začínajících na straně 404. Modifikoval jsem proto možnosti původních tříd prostřednictvím jejich potomků, které jsem pak v tomto příkladu použil.

Rozhraní `ISubCPU`

Nejprve bych ti představil rozhraní `ISubCPU`, které je potomkem rozhraní `ICPU` (viz výpis 30.1 na straně 404), jež jsem ti představoval, když jsme hovořili o aplikaci návrhového vzoru *Most* při návrhu jednoduché kalkulačky.

Rozhraní `ISubCPU` přidává metodu `getPopisek()`, která vrací popisek, jenž bude umístěn na klávese, jejíž stisk bude zapínat danou kalkulačku.

Výpis 31.1: Rozhraní `ISubCPU`, jehož instance představují CPU pracující v rámci nějaké super-CPU

```
package rup.česky.vzory._31_strategie;

import rup.česky.vzory._30_most.ICPU;

/*******************************************************************************
 * Instance třídy <code>ISubCPU</code> představují jednotlivé CPU
 * pracující v rámci nějaké "SuperCPU", která mezi nimi vhodně přepíná.
 */
public interface ISubCPU extends ICPU
{
    /***************************************************************************
     * Vrátí popisek na tlačítku přepínajícím na danou kalkulačku.
     *
     * @return Požadovaný popisek
     */
    public String getPopisek();
}
```

Třída `ReálnáSub`

Současně s tímto rozhraním jsem definoval potomky tříd `Binární` (viz výpis 30.4 na straně 411) a `Reálná` a pojmenoval je `BinárníSub` a `ReálnáSub`. Obě rozšiřují schopnosti svých rodičů právě o implementaci rozhraní `ISubCPU`. Protože jsou obě rozšíření prakticky stejná, uvádím pouze definici třídy `ReálnáSub`.

Výpis 31.2: Třída `ReálnáSub`, jejíž instance pracují jako reálné kalkulačky zabudované uvnitř nějaké `SuperCPU`

```
package rup.česky.vzory._31_strategie;

import rup.česky.vzory._30_most.Reálná;


/*******************************************************************************
 * Instance třídy <code>ReálnáSub</code> jsou použité jako CPU
 * pracující s reálnými čísly a jsoucí součástí nějaké {@link SuperCPU}.
```

```
 */
public class ReálnáSub extends Reálná implements ISubCPU
{
//== NESOUKROMÉ  METODY INSTANCÍ ===============================================

    /*****************************************************************************
     * Vrátí popisek na tlačítku přepínajícím na danou kalkulačku.
     */
    public String getPopisek() {
        return "Dbl";
    }

}
```

Třída SuperGUI

Zároveň jsem definoval i potomka třídy `GUI` (viz výpis 30.3 na straně 407), který počítá s tím, že bude obsluhovat nějakou instanci třídy `SuperCPU` (nebo jejího potomka). Tento potomek přidá k funkčnosti svého rodiče jediné: cpu (přesněji super-cpu), s níž bude komunikovat, se představí zavoláním její metody `setGUI(SuperGUI)`, v jejímž parametru předá odkaz na sebe. Učiní tak proto, aby `SuperCPU` mohla požádat `GUI` o změnu vzhledu okna poté, co uživatel požádá o změnu kalkulačky.

Výpis 31.3: Třída `SuperGUI` vytvářející GUI pro `SuperCPU`

```
package rup.česky.vzory._31_strategie;

import rup.česky.vzory._30_most.GUI;


/*******************************************************************************
 * Instance třídy <code>SuperGUI</code> definuje grafické uživatelské rozhraní
 * pro obecnou, předem neznámou kalkulačku.
 * Komunikuje s instancí třídy {@link SuperGUI}, které předá odkaz na sebe,
 * aby ji mohla super-cpu žádat o změnu vzhledu.
 */
public class SuperGUI extends GUI
{
//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří grafické uživatelské rozhraní kalkulačky,
     * jejíž CPU dostane jako parametr.
     * V rámci své konstrukce předá této CPU odkaz na sebe.
     *
     * @param cpu  CPU kalkulačky, pro niž vytváří <code>SuperGUI</code>
     */
    public SuperGUI( SuperCPU cpu ) {
        super( cpu );
        cpu.setGUI( this );
    }

}
```

Třída SuperCPU

Vlastní třída `SuperCPU` je poměrně jednoduchá. Její konstruktor dostane ve svých parametrech pole CPU, mezi kterými bude přepínat. Každé se zeptá na jí požadované rozložení kláves, které doplní tlačítky pro přepínání mezi jednotlivými CPU. Tuto novou

podobu přepravky si zapamatuje, aby mohla při přepnutí na danou CPU nastavit příslušné rozmístění kláves.

Reakce na stisk tlačítka

Když GUI oznámí, že někdo stiskl nějaké tlačítko, podívá se, jestli to není tlačítko na přepínání mezi CPU, a pokud je, tak požádá GUI o příslušný layout (= rozložení kláves) a nastaví pro danou CPU i její poslední stav displeje. Jedná-li se o běžné, tj. nepřepínací tlačítko, přepošle je aktuální CPU a nechá na displeji zobrazit od ní obdržený výsledek.

Nestandardnost ovlivňování klienta

Toto ovlivňování klienta (z hlediska programu je GUI vůči `SuperCPU` klientem) není sice u návrhového vzoru *Strategie* běžné, ale schválně jsem původní statický příklad, v němž všechny podřízené CPU sdílely jedno společné klávesové pole, upravil tak, aby si je mohla každá přizpůsobit po svém. Opět jsem ti na něm chtěl ukázat, že to, že se něco obvykle nedělá, ještě neznamená, že se to dělat nesmí.

Výpis 31.4: Třída `SuperCPU` sloužící jako přepínač mezi jednotlivými „jednoduchými" CPU

```
package rup.česky.vzory._31_strategie;

import java.util.LinkedHashMap;
import java.util.Map;

import rup.česky.debug.ToString;
import rup.česky.vzory._30_most.GUI;
import rup.česky.vzory._30_most.ICPU;
import rup.česky.vzory._30_most.Klávesnice;


/*******************************************************************************
 * Instance třídy Binární představují CPU jednoduché binární kalkulačky,
 * která umí pouze sčítat, odečítat a mazat naposledy zadaná čísla.
 */
public class SuperCPU implements ICPU
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final int          jednotek;
    private final ISubCPU[]    poleCPU;
    private final String[]     názvy;
    private final Klávesnice[] klávesy;
    private final String[]     displej;
    private final Map<String,Integer> indexCPU =
                                    new LinkedHashMap<String,Integer>();


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Index aktuální CPU. */
    private int icpu;

    private SuperGUI gui;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří super-cpu schopnou přepínat mezi cpu obdrženými jako parametry.
```

```
     */
    public SuperCPU( ISubCPU... poleCPU )
    {
        if( (poleCPU == null)  ||  (poleCPU.length == 0) )
            throw new IllegalArgumentException(
                ToString.array1o( "\nDegenerované pole CPU", poleCPU) );
        jednotek = poleCPU.length;
        this.poleCPU = poleCPU.clone();
        displej = new String[ jednotek ];
        názvy   = new String[ jednotek ];
        icpu = 0;
        for( int i=0;   i < poleCPU.length;   i++ ) {
            String název = poleCPU[i].getPopisek();
            názvy[i] = název;
            indexCPU.put( název, i );
            displej[i] = "";
        }
        klávesy = připravKlávesnice();
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * {@inheritDoc} */
    public String stisknuto( String popisek ) {
        if( indexCPU.containsKey( popisek ) ) {
            icpu = indexCPU.get( popisek );
            gui.nastavRozměryOkna();
        } else {
            displej[icpu] = poleCPU[icpu].stisknuto( popisek );
        }
        return displej[icpu];
    }


    /**************************************************************************
     * {@inheritDoc} */
    public Klávesnice getRozměry() {
        return klávesy[icpu];
    }


    /**************************************************************************
     * Zapamatuje si GUI, s nímž komunikuje, aby je mohl žádat o rozložení
     * odpovídající cpu, na kterou se přepne.
     */
    public void setGUI( SuperGUI gui ) {
        this.gui = gui;
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================================

    /**************************************************************************
     * Pro každou obsluhovanou CPU si zapamatuje její požadavky
```

```
     * na rozložení kláves a doplní je tlačítky pro přepínání
     * mezi jednotlivými CPU.
     */
    private Klávesnice[] připravKlávesnice() {
        Klávesnice[] klávesy = new Klávesnice[ jednotek ];
        for( int i=0;   i < jednotek;   i++ ) {
            Klávesnice původní = poleCPU[i].getRozměry();
            int sloupců = Math.max( původní.sloupců, jednotek );
            String[][] popisky = new String[ původní.řádků+1 ][ sloupců ];

            int s=0;
            for( String název : indexCPU.keySet() ) {
                popisky[0][s++] = název;
            }
            for( ;   s < sloupců;   popisky[0][s++] = "" );

            for( int r=0;   r < původní.řádků;   r++ ) {
                String[] řádek = popisky[ r+1 ];
                for( s=0;   s < původní.sloupců;   s++ )
                    řádek[s] = původní.popisek( r, s );
                for( ;   s < sloupců;   řádek[s++] = "" );
            }
            klávesy[i] = new Klávesnice( popisky );
        }
        return klávesy;
    }


//== TESTY ====================================================================

    /*****************************************************************************
     * Testovací metoda.
     */
    public static void test() {
        new SuperGUI( new SuperCPU( new BinárníSub(),
                                    new ReálnáSub() ) );
    }
    /** @param args Parametry příkazového řádku - nepoužité. */
    public static void main( String[] args )  {  test();  }
}
```

535. Třída `SuperCPU` nepočítá s tím, že by bylo možno přidávat za chodu další CPU. Šlo by ji upravit?

Úprava pro dynamické přidávání CPU

Samozřejmě. Stačilo by zaměnit konstantní pole za proměnná a přidat metodu, která přidá novou CPU a znovu zavolá metodu `připravKlávesnice()`, aby rozšířila první řádek o tlačítko právě přidané CPU. Pokud bys měl chuť, můžeš si toto rozšíření udělat za domácí cvičení.

Shrnutí – co jsme se naučili

- Návrhový vzor *Strategie* ukazuje, jak je možno zabezpečit přepínání použitých algoritmů za chodu programu.
- Oproti vzoru *Most* se nezabývá výměnou části programu, ale přepínáním mezi několika částmi, které jsou (většinou trvalou) součástí programu. Vzor se totiž nezaměřuje na architekturu systému, ale na jeho chování.
- Oproti vzoru *Stav* není přepínání interní záležitostí modulu, ale pracuje na objednávku klienta.
- Společným „rodičem" jednotlivých stavů nemusí být jen rozhraní, ale může jím být i třída implementující implicitní verze metod.
- Tento společný rodič může aplikovat i vzor *Šablonová metoda*.
- Návrhový vzor *Strategie* patří mezi vzory uvedené v GoF.

KAPITOLA 32

Každý chvilku tahá pilku (Model-Pohled-Ovládání – Model-View-Controller)

- **Účel**
- **Implementace**
- **Příklad: Reversi (Othello)**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Odděluje část programu mající na starosti předzpracování příkazů uživatele (tj. zjištění, co od programu vlastně chce) od částí zabezpečujících logiku programu, která uživatelovy příkazy zpracovává, a části, která má na starosti zobrazení výsledků.

[1] MVC (Model/View/Controller) consist of three kind of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. MVC decouples them to increase flexibility and reuse. – MPO (Model/Pohled/Ovládání) sestává ze tří druhů objektů. *Model* představuje objekt aplikace, *Pohled* prezentaci na obrazovce a *Ovládání* definuje způsob, jak uživatelské rozhraní reaguje na vstup od uživatele. MPO odděluje tyto objekty a tím zvyšuje flexibilitu a znovupoužitelnost celého řešení.

Účel

536. Řekl bych, že tento návrhový vzor je pouze aplikací principů, o nichž jsme již hovořili.

Proč jsme vzor zařadili

Máš pravdu. Tuto kapitolu jsem do knihy zařadil hlavně proto, že se v různých knihách a kurzech tento vzor často zmiňuje. Musím sice přiznat, že MPO (nebo dáváš-li přednost angličtině, tak MVC) je spíše architektonický vzor uplatňující se při celkovém návrhu architektury programu, ale přesto jsem si říkal, že by se o něm měla kniha specializovaná na výklad návrhových vzorů zmínit.

537. Takovéto rozhodnutí nechám na tobě. Když už ses tedy rozhodl jej sem zařadit, tak jej trochu rozeber.

Funkce vzoru

Jak jsem již popsal ve stručné charakteristice, tento vzor explicitně odděluje část programu zabývající se vstupem a předzpracováním zadávaných příkazů od části reakcí na zadané příkazy a části mající na starosti výstup výsledků. V čisté verzi má každou z těchto činností na starosti samostatný objekt, resp. skupina objektů (mohli bychom říci samostatný modul).

538. Přiznám se, že jsem se s něčím podobným již setkal, ale většinou byla celá komunikace s uživatelem, tj. jak vstup, tak výstup, naprogramována v jedné třídě.

Co zahrnout do vstupu

Záleží na tom, co nazýváš vstupem a co výstupem. Do vstupu nemůžeš zahrnovat vykreslování tlačítek, nabídek a dalších ovládacích prvků, ale zahrnuješ tam naopak reakce na události vyvolané prostřednictvím těchto prvků (stiskem tlačítka, zaškrtnutím políčka atd.).

U jednodušších programů bývají opravdu jak vykreslování GUI, tak definice reakcí na události definovány v jedné třídě. I tak ale můžeš definici obou činností oddělit alespoň do samostatných metod a nedefinovat reakci na události v rámci anonymní třídy definované uvnitř metody specifikující zobrazení dialogového okna.

Důležité je, že tímto oddělením můžeš snáze zabezpečit, aby aplikace reagovala jednotně nezávisle na tom, zadá-li uživatel svůj příkaz z klávesnice, myší nebo prostřednictvím nějakého hlasového vstupu.

Co zahrnout do výstupu

Na druhou stranu do části programu zpracovávající výstup (lepší by snad bylo říci: do části prezentující model) bychom neměli striktně zařazovat pouze vlastní vykreslení dodaných hodnot, ale někdy je výhodné do ní zařadit i transformaci výsledků do podoby, která bude předkládána uživateli. Je přitom jedno, zda bude předkládána opět prostřednictvím GUI v nějakých jeho komponentách nebo zda bude ukládána do souboru či tisknuta na tiskárnu. Navíc může uživatel chtít obdržet výsledky jednou v tabulkové podobě a jindy třeba graficky.

Co je to model

Třetí částí je tzv. *model*, který zahrnuje vnitřní reprezentaci zpracovávaných dat a celou aplikační logiku, která tato data zpracovává.

539. V minulých kapitolách jsme si ukazovali různé definice kalkulačky. Mohl bys mi ukázat jednotlivé moduly na nich?

Rozdělení na příkladu kalkulaček

Naše kalkulačky zrovna patří mezi ty jednoduché aplikace, u nichž je zbytečné vše rozdělovat na tři zcela oddělené části. Mají proto sloučený modul zabývající se předzpracováním příkazů uživatele s modulem vykonávajícím tyto příkazy.

Aby to ale zase nebylo tak jednoduché, tak při detailnějším rozboru zjistíš, že je tu tento návrhový vzor aplikován jak ve třídě `GUI`, tak ve třídách implementujících rozhraní `ICPU`.

540. Co to tu na mne zase hraješ? Jak je implementovaný v obou třídách?

Základní úkoly třídy `GUI`

Třída `GUI` má na starosti vykreslení aplikačního okna, převzetí požadavků uživatele a jejich předání instanci `ICPU` a naopak převzetí výstupu CPU a jejich zobrazení. Zdánlivě se tedy stará pouze o vstup a výstup.

Svůj model má každé tlačítko

Když se ale začneš vrtat v tom, jak jsou naprogramována jednotlivá tlačítka, tak zjistíš, že i ona mají svůj „modul" analyzující požadavky uživatele, svůj model a svůj vykreslovací „modul". `GUI` tlačítku pouze naznačí, co chce nakreslit, a nechá na něm, jak to zařídí. Obdobně reakce na události dostanou od tlačítka informaci o nastalé události předzpracovanou.

Samostatnou kapitolou je pak model tlačítek. Požadavky na něj definuje rozhraní `javax.swing.ButtonModel` a implementují je dvě třídy: jedna definuje model pro tlačítka, která se po stisku vrátí do původní polohy, a druhá pro tlačítka, která zůstanou stisknuta a pro návrat do původní polohy je musíš stisknout znovu.

541. No dobrá, tlačítka mají svůj model. Jenomže z toho, co jsi říkal, jsem pochopil, že třídy implementující `ICPU` mají pro změnu své vlastní předzpracování vstupu a výstupu.

Samozřejmě. CPU dostala uživatelův povel jako polotovar, ze kterého musela nejprve odvodit, co po ní uživatel vlastně chce. Po této analýze provedla požadovanou činnost a výsledek musela pro změnu zase převést do tvaru, s nímž si třída `GUI` uměla poradit. CPU tak měla vlastní zpracování vstupu, model i přípravu výstupu.

Třídy implementující rozhraní `ICPU` však byly tak jednoduché, že pro jednotlivé operace nevyčleňovaly žádné moduly, ale pouze metody, anebo dokonce jen části kódu.

Třída `Binární`

Ve třídě `Binární` (viz výpis 30.4 na straně 411) měla vstup na starosti metoda `stisknuto(String)` a zpracování zadaných příkazů metody `plusMinus(char)` a `spočti()`. Rozdělení ale nebylo úplně čisté, protože reakce na stisk číslicové klávesy či smazání posledního znaku byla natolik jednoduchá, že jsem pro ni nedefinoval separátní metodu a vyřešil jsem ji ihned v rámci metody `stisknuto(String)`.

Třída `Reálná`

Ve třídě `Reálná` (přesněji `rup.česky.vzory._30_most.Reálná` – ta nemá zdrojový kód uveden v knize a najdeš jej pouze mezi doprovodnými programy) je vše vyřešeno čistěji. Zpracování vstupu má na starosti kaskáda tvořená metodou `stisknuto(String)` následovanou jednou z metod `jednoznak(String)`, `dvojznak(String)` a `trojznak(String)`. Ty rozhodnou, co se bude dělat, a pověří dalším zpracováním některou z metod oprašujících model.

V obou třídách je pak začleněno i předzpracování informace pro zobrazení na displeji, protože třídy zabývající se výstupem požadují hotový řetězec připravený k zobrazení.

542. Předpokládám, že účelem je totéž, co u většiny vzorů: umožnit co nejjednodušší změnu kterékoliv z částí.

Hlavní důvod rozdělení: usnadnění změn

Máš pravdu. Hlavním důvodem pro oddělení výše zmíněných tří činností je usnadnění změny kterékoliv z nich. Výše uvedený přístup zabezpečil, že všechny CPU mohly sdílet společné GUI nezávisle na tom, co a jak přesně chtěla ta která CPU na displeji zobrazit (říkal jsem si třeba, že binární kalkulačka zobrazuje záporná čísla jako dvojkový doplněk, naopak reálná průběžně zobrazuje obsah paměti a poslední zadanou operaci).

Obdobně je to i s použitím tohoto návrhového vzoru v jiných situacích. Vymyslel jsi lepší algoritmy na zpracování dat? Prima. Inovuješ model, a zachováš-li komunikační rozhraní se zbylými dvěma složkami, uživatel se o tom vůbec nedozví.

Dostal jsi požadavek zobrazovat data v různých formátech (např. jednou jako tabulku hodnot, podruhé jako sloupcový graf, potřetí jako koláčový graf, počtvrté jako mapu s barvami označujícími hodnotu pro příslušnou oblast)? Definuješ čtyři různé pohledy a s využitím vzoru *Strategie* necháš na uživateli, aby si vybral, který je pro něj v dané chvíli optimální.

Potřebuješ vedle vstupu z klávesnice doplnit i vstup ze souboru nebo z webového rozhraní? Definuješ tři vstupní moduly a zbylé dvě části aplikace nemusí vůbec zajímat, odkud jsi obdržená data získal.

Požadavky na změnu vstupu a výstupu patří k nejčastějším

Důležitost využívání tohoto návrhového vzoru tkví v tom, že právě požadavky na modifikace vstupu a výstupu patří k těm nejčastějším, a je proto nesmírně důležité, aby vazby mezi moduly obstarávajícími vstup a výstup a modulem řešícím klíčové úlohy aplikace byly co nejvolnější. Aby změna kteréhokoliv z nich pokud možno neovlivnila zbylé dva.

543. Řekl bych, že uvedené důvody jsou dostatečně pádné. Přesto se ale zeptám: existují ještě další?

Měl bych ještě jeden, i když to není tak úplně nový důvod, ale spíš rozvinutí těch výše zmíněných.

Aplikace vzoru usnadňuje přenositelnost mezi platformami

Oddělíš-li dobře jednotlivé složky aplikace, můžeš získat aplikaci, která je velice snadno přenositelná mezi platformami. Všechny aplikace, které jsem ti ukazoval jako příklady, byly naprogramovány na platformu Java SE (Standard Edition). Aplikace, které pro zobrazení výsledků používají knihovnu *Tvary* (tj. třídu `SprávcePlátna` & spol.), lze prostou výměnou této knihovny převést na platformu Java ME (Micro Edition), takže si je pak budeš moci pouštět třeba na svém mobilu.

Implementace

544. Prozradíš mi něco o implementaci tohoto vzoru?

Vzor je obecným doporučením

Tento návrhový vzor je takovým obecným doporučením, které je možné implementovat různými způsoby.

Konkrétní vazby nespecifikuje

Vzor např. nijak nespecifikuje, jak přesně má komunikace mezi jeho jednotlivými částmi probíhat. V některých aplikacích posílá model zobrazovacímu modulu informace o tom, kdy a jak jej má zobrazit, jindy si naopak zobrazovací modul o tyto informace sám říká modelu.

Vstup a výstup se často slučují

Jak jsem již uváděl, v řadě jednodušších aplikací se někdy zobrazovací a ovládací část slučují do jedné třídy, protože použité grafické uživatelské rozhraní k tomu přímo vybízí.

Stručně řečeno: nenajdeš žádná přesná implementační doporučení, pouze obecné zásady doporučující tyto tři činnosti oddělit, abys pak mohl s minimální námahou změnit kteroukoliv z nich, aniž bys tím příliš ovlivnil zbylé dvě.

Často se aplikuje vzor *Pozorovatele*

Často se používá aplikace vzoru *Pozorovatel*. *Pohled* se přihlásí u *modelu* jako jeho pozorovatel a *model* jej v případě jakékoliv změny na tuto skutečnost upozorní. *Pohled* si pak od *modelu* vyžádá potřebné informace a aktualizuje výstup.

Jak ale jistě odhadneš, takovéto řešení má smysl pouze tehdy, pracuje-li aplikace s více pohledy. Je-li v aplikaci pouze jeden *model* a jeden *pohled*, lze klasické přihlašování nahradit např. předáním parametru v konstruktoru.

Příklad: Reversi (Othello)

545. Tak to jsi mi toho opravdu moc neřekl. Nenašel by se tam alespoň nějaký ilustrační příklad?

Příklad aplikace: knihovna swing

Pěkným příkladem je např. knihovna swing. Každá z jejích komponent, která slouží ke vstupu údajů (tlačítka, nabídky, zaškrtávací políčka, …), má uvedené tři součásti definované odděleně. Při studiu knihovny tak např. zjistíš, že příkazy nabídky i tlačítka mají společný model a liší se pouze modulem zodpovědným za jejich zobrazení.

Na druhou stranu klasické tlačítko a tlačítko, které zůstane po stisku zamáčknuté, mají stejnou zobrazovací část, avšak liší se drobně ve svém modelu.

Abys ale neřekl, že jsem tě zcela odbyl, připravil jsem pro tebe program hrající známou hru Reversi, nazývanou někdy také Othello. V následujících výpisech budou drobné vynechávky, k nimž se vrátíme v kapitole *Zpátky na stromy (Pamětník – Memento)*, začínající na straně 467. Tam si ukážeme, jak je možné do programu zabudovat možnost návratu do předchozího stavu a opětné obnovení stavu, z nějž ses do předchozího stavu vracel.

546. Neříkej, co mi budeš vysvětlovat bůhví kdy, a pověz mi raději, co sis pro mne připravil na teď.

Pravidla hry

Nejež se, už se k tomu chystám. Pro jistotu nejprve připomenu pravidla hry:

– Rozměr plochy

- Reversi se hraje na šachovnici o typickém rozměru 8 × 8 polí.

– Výchozí postavení

- Ve výchozí pozici jsou na čtyřech středových polích kameny obou hráčů uspořádány tak, že kameny každého hráče jsou na jiné diagonále.

– Střídavé tahy

- Hráči pokládají na šachovnici střídavě své kameny.

– Kam lze položit kámen

- Kámen lze položit pouze na pole, které sousedí se soupeřovým kamenem či souvislou řadou soupeřových kamenů, za níž je opět kámen hráče. Po položení kamene se všechny soupeřovy kameny mezi právě položeným kamenem a jeho protějškem za soupeřovými kameny stávají majetkem hráče, který kámen položil.

– Nemůže-li hráč táhnout

- Nemůže-li hráč položit žádný kámen tak, aby vyhověl předchozímu pravidlu, nemůže hrát a soupeř hraje znovu.

– Kdy je konec

- Hra končí v okamžiku, kdy už není možné položit další kámen.

– Kdo je vítěz

- Vítězem je ten, kdo při ukončení hry vlastní více kamenů.

547. Pravidla znám. Ukaž, jak jsi hru naprogramoval.

Diagram tříd

Podívej se na diagram tříd na obrázku 32.1. Možná ti bude připadat složitý, ale to je částečně proto, že jsem do aplikace přidal několik rozhraní, která umožní jednoduše vyměňovat konkrétní implementaci dané funkčnosti.

Jádro: třída `Reversi`

Jádrem celé aplikace je třída `Reversi`, která celou aplikaci spouští. Třída `Reversi` je jediná, která ví, kdo se skrývá za rozhraními `IModel`, `IPohled` a `IKurzor`, protože vytváří jejich instance, které pak předává ostatním třídám.

Rozhraní `IModel`

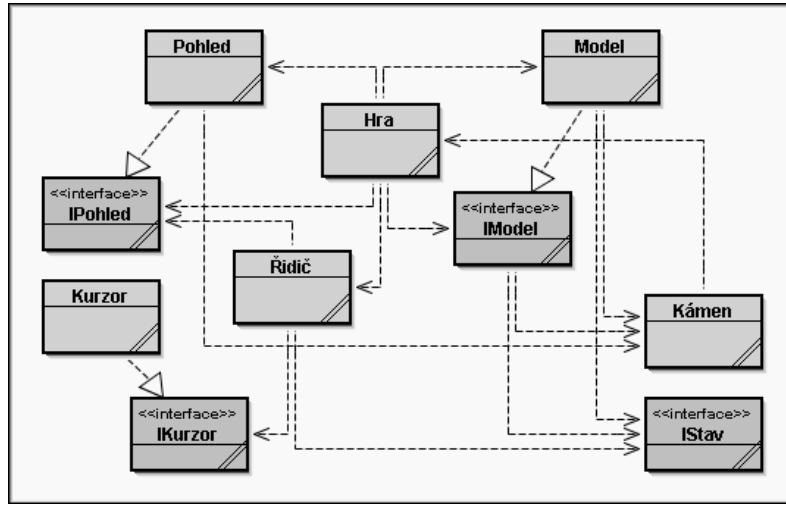
Instance rozhraní `IModel` má na starosti realizaci vlastní logiky hry. Nestará se vůbec o to, jak bude aplikace komunikovat s uživatelem – ji zajímá pouze to, kam se protihráč rozhodl umístit svůj kámen.

Rozhraní `IPohled`

Zobrazení aktuálního stavu hry má na starosti instance rozhraní `IPohled`. Ta si řekne instanci modelu o potřebné informace a na jejich podkladě vše nakreslí.

Třída `Řidič`

Poslední důležitou třídou je `Řidič`, ta má na starosti zabezpečení správné reakce na požadavky uživatele.



Obrázek 32.1
Diagram tříd hry Reversi

548. Proč jsi neschoval za rozhraní také toho řidiče?

Proč `Řidič` není rozhraní

Protože se na něj s výjimkou instance třídy `Reversi` nikdo neobrací. `Reversi` vše inicializuje a pak předá řízení průběhu hry řidiči. Ten komunikuje s uživatelem a říká ostatním částem hry, co mají dělat.

549. Vedle rozhraní `IModel` a `IPohled` tam vidím i rozhraní `IKurzor`, o němž ses doposud nezmiňoval.

Rozhraní `IKurzor` a třída `Kurzor`

Třídu `Kurzor` jsem schoval za rozhraní proto, že její implementace je závislá na způsobu zobrazování. Když se proto vymění způsob zobrazování (např. když nahradím využívání správce pláten nějakým elegantním dialogovým oknem), budu muset změnit i třídu realizující kurzor.

550. A proč jsi tedy nezačlenil její funkci pod rozhraní `IPohled`?

Proč je kurzor samostatný

Protože každá entita v programu má mít pouze jeden úkol. Instance rozhraní `IPohled` mají na starosti zobrazování stavu hry. Naproti tomu instance třídy `IKurzor` mají na starosti zobrazování komunikace s uživatelem. Proto jsem je rozdělil.

`Kurzor` slučuje všechny části v jedné třídě

V souvislosti se třídou `Kurzor` bych tě ale chtěl upozornit na jinou věc. `Kurzor` je třída, která souvisí se všemi třemi částmi: převádí ovládací příkazy uživatele na jejich zobrazení (pohyb kurzoru) a přitom připravuje informace, které budou ovlivňovat následné chování modelu (pozici, kam uživatel položil další kámen). Všechny tři složky jsou tam poměrně provázané. Protože se ale jedná o jednoduchou třídu, zůstává vše i při této provázanosti poměrně přehledné. Považoval jsem proto za rozumnější ponechat vše pohromadě a nerozbíjet třídu na jednotlivé části.

551. Nezmiňoval ses ještě o třídě `Kámen`.

Třída `Kámen`

Instance třídy `Kámen` jsou nositelkami informací o konkrétním políčku. Tato třída by měla fungovat jako výčtový typ se čtyřmi druhy polí: okrajovým polem za hranicemi hrací plochy, polem s kamenem hráče, polem s kamenem počítače a prázdného pole.

Proč `Kámen` není výčtový typ

Jako výčtový typ jsem ji nedefinoval proto, že jsem chtěl, aby si prázdné pole pamatovalo i svoji sílu, tj. počet kamenů, které se obrátí v případě, kdy na ně hráč položí svůj kámen. Protože se mi v její definici nechtělo vyjmenovávat všechny počty jedním tahem obrátitelných kamenů, svěřil jsem výrobu těchto instancí programu a třídu definoval podle návrhového vzoru *Originál*. Její zdrojový kód si můžeš prohlédnout ve výpisu 32.1.

Třída `Kámen.Typ`

Ve výpisu si všimni, že třída definuje veřejný vnořený výčtový typ `Kámen.Typ` definující možné typy kamenů. Kamene se pak můžeš zeptat na jeho typ zavoláním metody `getTyp()`. Tímto obratem se sloučily všechny typy prázdných kamenů pod jednu typovou střechu, takže se pak v programu relativně snadno testují.

Výpis 32.1: Definice třídy `Kámen`, jejíž instance udržují informace o svém políčku

```
package rup.česky.vzory._32_mvc.reversi;


/*******************************************************************************
 * Instance třídy <code>Kámen</code> představují kameny na hracím poli,
 * přičemž mezi tyto kameny se počítají i prázná pole a neviditelná okrajová
```

```
 * pole ležící těsně za hranicemi hrací desky.
 * Prázdná pole si pamatují svoji sílu, tj. kolik kamenů by se obrátilo,
 * kdyby hráč položil na dané pole svůj kámen.
 * Instance této třídy se nezajímají o vzhled kamenů při jejich zobrazení,
 * slouží primárně k realizaci logiky hry.
 */
public class Kámen
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    public static final Kámen HRÁČ    = new Kámen( Typ.HRÁČ );
    public static final Kámen POČÍTAČ = new Kámen( Typ.POČÍTAČ );
    public static final Kámen OKRAJ   = new Kámen( Typ.OKRAJ );
    public static final Kámen PRÁZDNÝ = new Kámen( 0 );

    private static final int POČTŮ = Math.max( 4 * (Reversi.STRANA - 3),
                                               3 * (Reversi.STRANA - 2) );

    private static final Kámen[] počet = new Kámen[ POČTŮ ];
    static{ počet[0] = PRÁZDNÝ; }


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final int síla; //Síla pole s daným kamenem
    private final Typ typ;  //Typ kamene


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří nový kámen zadaného typu. Síla všech typů kamenů s výjimkou
     * kamenů typu <code>PRÁZDNÝ</code> je rovna -1.
     * Pro vytváření kamenů typu <code>PRÁZDNÝ</code> slouží
     * jiný konstruktor.
     *
     * @param typ Typ vytvářeného kamene
     */
    private Kámen( Typ typ )
    {
        this.typ = typ;
        síla     = -1;
    }


    /*****************************************************************************
     * Vytvoří kámen typu  <code>PRÁZDNÝ</code> o zadané síle. Silou kamene
     * se přitom rozumí počet kamenů, které se obrátí, když na danou pozici
     * položí hráč svůj kámen.
     *
     * @param síla  Síla vytvářeného kamene
     */
    private Kámen( int síla )
    {
        this.typ  = Typ.PRÁZDNÝ;
        this.síla = síla;
```

```
    }


    /*************************************************************************
     * Vrátí kámen oznamující zadanou sílu svého pole,
     * tj. počet obrácených soupeřových kamenů po tahu na danou pozici.
     */
    public static Kámen getInstance( int síla )
    {
        if( počet[síla] == null )
            počet[síla] = new Kámen( síla );
        return počet[síla];
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*************************************************************************
     * Vrátí "sílu" pole, na němž leží daný kámen.
     * @return Síla pole s kamenem
     */
    public int getSíla()
    {
        return síla;
    }


    /*************************************************************************
     * Vrátí typ daného kamene.
     */
    public Kámen.Typ getTyp()
    {
        return typ;
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ===================================================

    /*************************************************************************
     * Výčtový typ, jehož instance reprezentují jednotlivé typy kamenů,
     * které se mohou vyskytovat na hrací desce.
     */
    public static enum Typ {
        /** Položený kámen hráče.                          */ HRÁČ,
        /** Položený kámen počítače.                       */ POČÍTAČ,
        /** Kameny na polích těsně za okrajem hrací plochy. */ OKRAJ,
        /** Symbolický kámen představující prázdé pole.    */ PRÁZDNÝ;
    }

}
```

552. Každý druh kamene má definovanou jedinou instanci. Na desce ale může přece být víc kamenů stejného typu.

Kámen implementuje vzor *Muší váha*

Aplikoval jsem návrhový vzor *Muší váha*, o němž jsme hovořili v kapitole *Příliš mnoho instancí (Muší váha – Flyweight)*, začínající na straně 171.

553. Posledním typem, o němž ses doposud nezmínil, je rozhraní `IStav`. Na něm je zajímavé, že nemá žádnou třídu, která by je implementovala. K čemu tedy je?

Rozhraní IStav

Toto rozhraní a části kódu, které s ním souvisejí, budu prozatím ignorovat. Vrátím se k němu až v příkladu ke kapitole *Zpátky na stromy (Pamětník – Memento),* začínající na straně 467. Tam si budeme ukazovat, jak s jeho pomocí doplnit aplikaci o funkce *zpět* (*undo*) a *znovu* (*redo*).

Třída `Reversi` a jí vytvářené instance

554. Dobře, ještě chvíli to vydržím. Vysvětli mi tedy, jak funguje to, o čem jsi nyní ochoten hovořit.

Co má na starost

Třída `Reversi` je až trapně jednoduchá. Má jedinou starost: vytvořit instance všech složek, které budou následně realizovat vlastní hru, a tím vše spustit. Její zdrojový kód najdeš ve výpisu 32.2.

Stejně jednoduchá jsou i rozhraní `IModel` (výpis 32.3), `IPohled` (výpis 32.4) a `IKurzor` (výpis 32.5), jejichž instance třída vytváří a předává vytvářené instanci třídy `Řidič` (výpis 32.6), která pak přebírá řízení hry na svá bedra.

Výpis 32.2: Definice třídy `Reversi`, která iniciuje celou aplikaci

```
package rup.česky.vzory._32_mvc.reversi;

import rup.česky.tvary.SprávcePlátna;


/*******************************************************************************
 * Instance třídy <code>Reversi</code> představují hlavní třídu aplikace
 * sloužící k hraní hry Reversi, resp. Othello, s počítačem.
 */
public class Reversi
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    public static final int STRANA = 8;     //Počet polí vodorovně/svisle
    public static final int POLE   = 20;    //Rozměr jednoho pole

    private static final SprávcePlátna SP = SprávcePlátna.getInstance();


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /***************************************************************************
     * Vytvoří hru pro hrací desku zadaných rozměrů.
     *
     * @param strana Počet polí desky vodorovně/svisle
     * @param pole   Vodorovný a svislý rozměr polí desky
     */
    private Reversi( int strana, int pole )
    {
        IModel  model  = new Model ( strana );
        IPohled pohled = new Pohled( strana, pole,  model );
        IKurzor kurzor = new Kurzor( strana, pole );
        Řidič   řidič  = new Řidič ( model, pohled, kurzor );
```

```
    }


//== TESTY ====================================================================

    /*****************************************************************************
     * Testovací metoda.
     */
    public static void test()
    {
        Reversi hra = new Reversi( STRANA, POLE );
    }
    /** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {  test();  }
}
```

Výpis 32.3: Definice rozhraní `IModel` deklarujícího požadavky na třídu, jejíž instance budou mít na starosti vlastní logiku hry

```
package rup.česky.vzory._32_mvc.reversi;

import rup.česky.tvary.Pozice;


/*******************************************************************************
 * Instance rozhraní <code>IModel</code> obsahují vlastní logiku hry.
 * Nezabývají se tím, jak bude průběh hry zobrazen, a starají se pouze
 * o prinicipiální ošetření jednotlivých typů situací.
 */
public interface IModel
{
    /*****************************************************************************
     * Vrátí kámen na zadané pozici.
     *
     * @param x Vodorovná souřadnice pozice, levý okraj má x=1
     * @param y Svislá souřadnice pozice, horní okraj má y=1
     * @return  Kámen na zadané pozici
     */
    public Kámen getKámen( int x, int y );


    /*****************************************************************************
     * Na zadanou pozici umístí kámen hráče a vrátí informaci o tom,
     * má-li hráč po tahu počítače ještě nějaký dostupný tah.
     * Umístění kamene počítače nepotřebuje speciální metodu,
     * protože počítač nedělá chyby, a není jej proto nutno tak hlídat.
     *
     *
     * @param pozice Pozice pole, kam umístit kámen
     *               levý horní okraj má pozici [1;1]
     * @return Má-li hráč ještě k dispozici nějaký tah
     */
    boolean setKámen(Pozice pozice);


    /*****************************************************************************
     * Vrátí aktuální stav hry jako instanci rozhraní {@link IStav}.
```

```
     */
    IStav getStav();


    /*****************************************************************************
     * Nastaví stav hry zadaný prostřednictvím parametru.
     */
    void setStav(IStav stav);


    /*****************************************************************************
     * Vrátí řetězec uvádějící, kolik kamenů má právě hráč a kolik počítač.
     */
    String stavSlovy();


}
```

Výpis 32.4: Definice rozhraní `IPohled` deklarujícího požadavky na třídy, jejichž instance budou mít na starosti zobrazení stavu hry

```
package rup.česky.vzory._32_mvc.reversi;

/*******************************************************************************
 * Instance rozhraní <code>Pohled</code> jsou objektem zodpovědným
 * za správné zobrazení informací reprezentovaných modelem.
 *
 * @author    Rudolf PECINOVSKÝ
 * @version   0.00.000,  0.0.2007
 */
public interface IPohled
{
    /*****************************************************************************
     * Zobrazí aktuální pozici hry.
     */
    void zobraz();
}
```

Výpis 32.5: Definice rozhraní `IKurzor` deklarujícího požadavky na třídy, jejichž instance budou mít na starosti vizualizaci komunikace s uživatelem

```
package rup.česky.vzory._32_mvc.reversi;

import rup.česky.tvary.Pozice;


/*******************************************************************************
 * Instance rozhraní <code>IKurzor</code> představují objekty
 * zprostředkovávající vizualizaci komunikace s uživatelem,
 * tj. zobrazí uživateli, na které pole se chystá položit svůj kámen.
 */
public interface IKurzor
{
    /*****************************************************************************
     * Vrátí aktuální deskovou pozici kurzoru.
     * Políčko v levém horním rohu desky má souřadnice [1;1].
```

```
     */
    public Pozice getPozice();


    /*************************************************************************
     * Posune kurzor o jedno políčko vpravo;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u levého okraje.
     */
    public void posunVpravo();


    /*************************************************************************
     * Posune kurzor o jedno políčko vlevo;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u pravého okraje.
     */
    public void posunVlevo();


    /*************************************************************************
     * Posune kurzor o jedno políčko dolů;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u horního okraje.
     */
    public void posunDolů();


    /*************************************************************************
     * Posune kurzor o jedno políčko vzhůru;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u dolního okraje.
     */
    public void posunVzhůru();

}
```

Výpis 32.6: Definice třídy `Řidič`, jejíž instance přebírá po inicializaci od třídy `Reversi` řízení celé hry

```
package rup.česky.vzory._32_mvc.reversi;

import java.awt.event.KeyAdapter;
import java.util.ArrayList;
import java.util.List;
import rup.česky.společně.IO;
import rup.česky.tvary.SprávcePlátna;

import static java.awt.event.KeyEvent.*;


/*****************************************************************************
 * Instance třídy <code>Řidič</code> převezmou po inicializaci řízení celé hry,
 * komunikují s uživatelem a na základě této komunikace řídí ostatní objekty.
 */
public class Řidič extends KeyAdapter
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final SprávcePlátna SP = SprávcePlátna.getInstance();
```

```
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final IModel  model;
    private final IPohled pohled;
    private final IKurzor kurzor;

// ... Deklarace atributů umožňujících návrat o několik kroků zpět


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /****************************************************************************
     * Vytvoří instanci, která bude řídit hru.
     */
    public Řidič( IModel model, IPohled pohled, IKurzor kurzor )
    {
        this.model  = model;
        this.pohled = pohled;
        this.kurzor = kurzor;

// ... Kód umožňující pozdější návrat o pár kroků zpět

        SP.přihlašKlávesnici( this );
    }

//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /****************************************************************************
     * Ošetření události stisk klávesy.
     * Jedná-li se o opakovaný stisk téže klávesy před jejím puštěním,
     * ignoruje jej. Jinak spustí akci příslušnou dané klávese.
     * Stisk klávesy Esc program ukončí.
     *
     * @param  ke   Událost klávesnice, na kterou je třeba reagovat
     */
    public synchronized void keyPressed( java.awt.event.KeyEvent ke )
    {
        int kc = ke.getKeyCode();
        switch( kc )
        {
            case VK_LEFT:
                kurzor.posunVlevo();
                break;

            case VK_RIGHT:
                kurzor.posunVpravo();
                break;

            case VK_UP:
                kurzor.posunVzhůru();
                break;

            case VK_DOWN:
                kurzor.posunDolů();
                break;

            case VK_ENTER:
```

```
                if( !model.setKámen( kurzor.getPozice() ) )
                    konecHry();
// ... Kód řešící uložení aktuálního stavu hry pro případ pozdějšího návratu
                break;

// ... Bloky reagující na požadavky po vrácení o krok zpět,
//     anebo opětné přejití do stavu o krok vpřed

            default:
        }
        pohled.zobraz();
    }


//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ================================

    /**************************************************************************
     *
     */
    private void konecHry()
    {
        pohled.zobraz();
        IO.zpráva( "Konec hry\n" + model.stavSlovy() );
        System.exit( 0 );
    }

}
```

Třída `Kurzor`

555. K rozhraním opravdu není co dodat a třída `Řidič` byla doopravdy jednoduchá. Teď by byl asi čas na `Kurzor`, s nímž `Řidič` komunikuje.

Co má na starosti

Instance třídy `Kurzor` (výpis 32.7) realizují pohyby kurzoru podle povelů uživatele zprostředkovaných řidičem. Aby bylo ovládání snadnější, implementují „válcový" pohyb, kdy se kurzor po opuštění hrací desky v jednom směru vzápětí objeví na opačném kraji – tuto funkčnost ale třídě předepisuje již implementované rozhraní.

Řidič kurzor nehlídá

Ve výpisu řidiče si všimni, že řidič kurzoru říká, kam s ním chce uživatel pohnout, ale nijak se nezajímá o jeho aktuální pozici. Na tu se zeptá až ve chvíli, kdy se uživatel rozhodne položit na pozici kurzoru svůj kámen.

Výpis 32.7: Definice třídy `Kurzor` vizualizující komunikaci s uživatelem

```
package rup.česky.vzory._32_mvc.reversi;

import rup.česky.tvary.APosuvný;
import rup.česky.tvary.Barva;
import rup.česky.tvary.IKreslený;
import rup.česky.tvary.Kreslítko;
import rup.česky.tvary.Pozice;
import rup.česky.tvary.Čtverec;

import static rup.česky.tvary.SprávcePlátna.SP;
```

```
/*******************************************************************************
 * Instance třídy <code>Kurzor</code> představuje kurzor označující
 * pozici, na kterou se hráč chystá umístit svůj kámen.
 * Tato třída sdružuje jak kód zabezpečující zobrazování průběhu,
 * tj. konkrétně zobrazení své instance, tak kód spolupracující s logikou hry,
 * které prozrazuje souřadnice pozice, na kterou chce uživatel položit svůj kámen.
 */
public class Kurzor implements IKurzor, IKreslený
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final APosuvný obraz;   //Obraz kurzoru na plátně
    private final int strana;       //Políčkový rozměr hrací desky
    private final int pole;         //Bodová velikost jednoho pole desky
    private final int posun;        //Překrytí obrazu kurzoru přes okraj pole
    private final int šířka;        //Šířka obrazce kurzoru v bodech


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private int x, y;   //Políčkové souřadnice kurzoru


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*******************************************************************************
     * Vytvoří nový kurzor a přihlásí jej u klávesnice.
     *
     * @param deska  Deska, jejíž obsah bude pomocí kurzoru ovládat
     */
    public Kurzor( int strana, int pole )
    {
        this.pole   = pole;
        this.strana = strana;
        posun  = pole / 4;          //Kurzor bude o 1/4 přesahovat přes
        šířka  = pole + 2*posun;    //okraj pole

        x = (strana / 2);           //Počáteční souřadnice kurzoru
        y = (strana / 2);           //budou uprostřed hrací desky

        //Kurzor bude zobrazen jako průsvitný čtverec
        obraz = new Čtverec( x*pole - posun, y*pole - posun,
                             šířka, Barva.KOUŘOVÁ );
        x++; y++;   //Změna políčkových souřadnic plátna na souřadnice desky
        SP.přidej( this );  //Zobrazení kurzoru na plátně
        //Předpokládáme, že pracovní deska je již u správce pláten přihlášena,
        //a že se proto bude kurzor kreslit nad ní.
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Vrátí aktuální deskovou pozici kurzoru.
     * Políčko v levém horním rohu desky má souřadnice [1;1].
     */
    public Pozice getPozice()
    {
```

```
        return new Pozice( x, y );
    }


    /*************************************************************************
     * Posune kurzor o jedno políčko vpravo;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u levého okraje.
     */
    public void posunVpravo()
    {
        if( x < strana ) {
            x++;
            obraz.posunVpravo();
        } else {
            x = 1;
            obraz.setPozice( -posun, obraz.getY() );
        }
    }


    /*************************************************************************
     * Posune kurzor o jedno políčko vlevo;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u pravého okraje.
     */
    public void posunVlevo()
    {
        if( x > 1 ) {
            x--;
            obraz.posunVlevo();
        } else {
            x = strana;
            obraz.setPozice( (strana-1)*pole - posun, obraz.getY() );
        }
    }


    /*************************************************************************
     * Posune kurzor o jedno políčko dolů;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u horního okraje.
     */
    public void posunDolů()
    {
        if( y < strana ) {
            y++;
            obraz.posunDolů();
        } else {
            y = 1;
            obraz.setPozice( obraz.getX(), -posun );
        }
    }


    /*************************************************************************
     * Posune kurzor o jedno políčko vzhůru;
     * pokud by měl kurzor opustit hrací desku, zobrazí jej u dolního okraje.
     */
    public void posunVzhůru()
    {
```

```
        if( y > 1 ) {
            y--;
            obraz.posunVzhůru();
        } else {
            y = strana;
            obraz.setPozice( obraz.getX(), (strana-1)*pole - posun );
        }
    }


    /*****************************************************************************
     * Nakreslí kurzor zadaným kreslítkem.
     */
    public void nakresli( Kreslítko kreslítko )
    {
        obraz.nakresli( kreslítko );
    }

}
```

Třída `Pohled` a její vnořené třídy

556. Obávám se, že nejtěžší bude model, takže bych si teď nechal vysvětlit třídu `Pohled`.

Co má na starosti

Těžiště práce této třídy (zdrojový kód najdeš ve výpisu 32.8) je v metodě `nakresli(Kreslítko)`, která se postupně ptá modelu na typy kamenů na jednotlivých polích a vzápětí je na tato pole nakreslí.

Kreslení kamenů

Kameny jsou logicky součástí modelu, a proto se samy kreslit neumí. Třída `Pohled` proto pro jednotlivé typy kamenů definuje vlastní vnitřní třídy, jejichž instance vystupují jako obrazy těchto kamenů, které se umí nakreslit na plátno.

Třída `ObrazHP`

Kameny hráče a počítače se liší pouze svojí barvou, takže jejich obrazy jsou instancemi téže třídy `ObrazHP` a liší se pouze tím, jakou barvu dostane do vínku jejich konstruktor.

Třída `ObrazPrázdný`

Obrazy prázdných polí jsou pak instancemi třídy `ObrazPrázdný`. Nejjednodušší to má instance pole, které nemá žádnou sílu – ta nedělá nic. Ostatní se snaží doprostřed políčka zobrazit číslo představující hypotetickou sílu daného pole, tj. počet soupeřových kamenů, které se otočí po položení kamene hráče na dané pole.

Výpis 32.8: Definice třídy `Pohled`, jejíž instance mají na starosti zobrazení aktuálního stavu hry

```
package rup.česky.vzory._32_mvc.reversi;

import rup.česky.tvary.Barva;
import rup.česky.tvary.IKreslený;
import rup.česky.tvary.Kreslítko;
import rup.česky.tvary.SprávcePlátna;
import rup.česky.tvary.Čtverec;
import rup.česky.tvary.Text;


/*****************************************************************************
```

```
 * Instance třídy <code>Pohled</code> je objektem zodpovědným
 * za správné zobrazení informací reprezentovaných modelem.
 */
public class Pohled implements IPohled, IKreslený
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    private static final SprávcePlátna SP = SprávcePlátna.getInstance();
    private static final String FONT = "SansSerif";


//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final int    strana;     //Políčkový rozměr hrací plochy
    private final int    pole;       //Bodový rozměr jednoho pole
    private final IModel model;      //Zobrazovaný model

    private final ObrazHP  hráč;         //Kámen představující hráče
    private final ObrazHP  počítač;      //Kámen představující počítač
    private final ObrazPrázdný prázdný;  //Kámen představující prázdné pole

    /** Posun popisku označujícího sílu pole oproti souřadnicím pole. */
    private final int posunTextu;
    private final int výškaPísma;   //Výška písma v bodech


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /***************************************************************************
     * Vytvoří novou instanci pohledu, která vyčistí plátno a
     * připraví jednotlivé typy kamenů.
     *
     * @param strana Políčkový rozměr hrací plochy
     * @param pole   Bodový rozměr jednoho pole
     * @param model  Zobrazovaný model
     */
    public Pohled( int strana, int pole, IModel model )
    {
        this.strana = strana;
        this.pole   = pole;
        this.model  = model;
        this.posunTextu = pole / 4;
        this.výškaPísma = pole / 2;

        hráč    = new ObrazHP( Barva.ČERVENÁ );
        počítač = new ObrazHP( Barva.ZELENÁ  );
        prázdný = new ObrazPrázdný();

        SP.odstraňVše();
        SP.setKrokRozměr( pole, strana, strana );
        SP.přidej( this );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /***************************************************************************
     * Zobrazí aktuální pozici hry.
```

```
     */
    public void zobraz()
    {
        SP.překresli();
    }


    /**************************************************************************
     * Za pomoci dodaného kreslítka vykreslí obraz své instance
     * na zadných souřadnicích.
     *
     * @param x           Vodorovná souřadnice kresleného obrazce
     * @param y           Svislá souřadnice kresleného obrazce
     * @param kreslítko   Kreslítko, kterým se instance nakreslí na plátno
     */
    public void nakresli( Kreslítko k )
    {
        for( int x=1;   x <= strana;   x++ ) {
            int xx = (x-1) * pole;
            for( int y=1;   y <= strana;   y++ ) {
                Kámen kámen = model.getKámen( x, y );
                int yy = (y-1) * pole;
                switch( kámen.getTyp() )
                {
                    case HRÁČ:
                        hráč.nakresli( xx, yy, k );
                        break;

                    case POČÍTAČ:
                        počítač.nakresli( xx, yy, k );
                        break;

                    case PRÁZDNÝ:
                        prázdný.nakresli( kámen.getSíla(), xx, yy, k );
                        break;

                    default:
                        throw new IllegalStateException(
                            "Nepodporovaný typ: " + kámen.getTyp() );
                }
            }
        }
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    /**************************************************************************
     * Instance třídy <code>ObrazHP</code> představují obrazy kamenů
     * hráče a počítače.
     */
    private class ObrazHP
    {
        Čtverec čtverec;

        /**********************************************************************
         * Vytvoří obraz kamene některého z hráčů definovaný jeho barvou.
         * Tento obraz bude vykreslen na všech pozicích,
```

```
         * na nichž bude odpovídající kámen umístěn.
         */
        ObrazHP( Barva barva )
        {
            this.čtverec = new Čtverec( 0, 0, pole, barva );
        }

        /**************************************************************************
         * Nakreslí dodaným kreslítkem kámen na zadané pozici hrací desky.
         */
        void nakresli( int x, int y, Kreslítko k )
        {
            čtverec.setPozice( x, y );
            čtverec.nakresli( k );
        }
    }


    /******************************************************************************
     * Instance třídy <code>ObrazPrázdný</code> představují obrazy kamenů
     * na polích, na něž ještě nebyl umístěn kámen hráče či počítače.
     * Má-li pole nenulovou sílu, instance ji zobrazí.
     */
    private class ObrazPrázdný
    {
        Text text[] = new Text[ strana*4 ];


        /**************************************************************************
         * Vytvoří obraz prázdného pole, který bude vykreslen na všech pozicích,
         * na nichž ještě nebude umístěn kámen některého z hráčů.
         */
        ObrazPrázdný() {
            text[0] = getText( 0 );
        }


        /**************************************************************************
         * Nakreslí dodaným kreslítkem na zadané pozici hrací desky
         * kámen prázdného pole s informací o síle příslušného pole.
         */
        void nakresli( int síla, int x, int y, Kreslítko k )
        {
            if( síla == 0 )
                return;
            Text text = this.text[síla];
            if( text == null ) {
                text = this.text[ síla ] = getText( síla );
            }
            text.setPozice( x+posunTextu, y+posunTextu );
            text.nakresli( k );
        }

        /**************************************************************************
         * Metoda zastupující statickou metodu třídy <code>ObrazPrázdný</code>.
         * Úkolem metody je vytvořit instanci zobrazitelného čísla.
         */
        private Text getText( int síla )
```

```
        {
            Text text = new Text( 0, 0, String.valueOf( síla ) );
            text.setFont( FONT, Text.OBYČEJNÝ, výškaPísma );
            return text;
        }
    }

}
```

Třída `Model`

557. Řekl bych, že na předchozích programech bylo opravdu dobře vidět, jak lze ty tři části programu oddělit. Pořád jsme ale před sebou tlačili tu největší chuťovku: definici třídy `Model`. Už se na ni těším.

Nejjednodušší řešení

Definoval jsem v ní to nejjednodušší možné řešení, kdy program opravdu pouze zjišťuje, jakou má každé políčko sílu, aby pak položil svůj kámen na políčko s největší silou. Program proto snadno porazíš.

Lze snadno vyměnit za lepší

Na druhou stranu právě tím, že jsme oddělili model, pohled a řízení, tak je velmi snadné nahradit stávající model nějakým chytřejším, který bude vědět, jak je důležité obsadit rohová políčka nebo jak je naopak nebezpečné pokládat kámen na předposlední políčko před okrajem desky, a případně jej dovybavit i schopností podívat se na několik tahů dopředu, jaké důsledky by mohlo mít položení kamene na to či ono políčko.

558. Tak už se nerozplývej a ukaž, jak jsi to naprogramoval.

Atribut `deska`

Model má atribut `deska`, což je dvojrozměrné pole kamenů představující hrací desku a kameny na ní položené.

Pole `deska` je větší než hrací plocha

Abych nemusel při analýze desky neustále testovat, kdy už desku opouštím, je toto pole na každé straně o jeden řádek a sloupec delší než vlastní hrací deska. Do těchto okrajových buněk jsou pak vloženy speciální okrajové kameny. To výrazně zjednodušuje výpočet síly polí.

Metoda `seKámen(Pozice)`

Těžiště práce celého modelu je v metodě `seKámen(Pozice)`, která umisťuje hráčův kámen na zadanou pozici na desce.

1. Kontrola

Nejprve zkontroluje, jestli na tuto pozici smí hráč svůj kámen vůbec umístit. Pole, na něž smí hráč umístit svůj kámen, musí být prázdné a musí mít nenulovou sílu, aby po položení kamene hráče došlo k otočení alespoň jednoho kamene soupeře.

2. Obrať kameny

Je-li vše v pořádku, tak kámen hráče umístí a všechny ovlivněné kameny soupeře obrátí.

3. Tah počítače

Poté najde tah pro počítač, a pokud takový existuje, umístí na něj kámen počítače.

4. Kam může hráč táhnout

Nezávisle na tom, jak dopadl počítač (tj. jestli pro něj vůbec existoval tah), spočte optimální tah pro hráče. Není to proto, aby hráči radil, ale proto, aby zjistil, jestli má hráč vůbec kam táhnout.

5. Kdy hra končí

Nemůže-li hráč táhnout, ale počítač před tím táhnout mohl, znovu se pokusí najít tah pro počítač. Cyklus trvá tak dlouho, dokud nenastane situace, kdy bude mít hráč kam táhnout nebo kdy naopak nebude moci táhnout ani jeden. To, jestli může hráč ještě

táhnout anebo jestli již neexistuje žádný tah a hra končí, pak metoda vrátí jako svoji návratovou hodnotu.

Metoda najdiTah()

Vlastní hledání tahu v metodě `najdiTah()` je jednoduché. Projde se celá deska a u všech prázdných polí se zjistí jejich síla. Ta se zjistí tak, že se z daného pole podívá počítač všemi směry, a najde-li na sousedním poli kámen protihráče (případně celou řadu jeho kamenů) následovaný kamenem aktuálního hráče, spočte „zarámované“ kameny a připočte je k celkové síle pole. Silou pole je pak součet sil ve všech osmi směrech.

559. Tak to bylo opravdu podrobné. Teď už mi nezbývá než program projít a na vše, o čem jsi povídal, se ještě jednou podívat.

Výpis neobsahuje undo a redo

Podívej se, její zdrojový kód najdeš ve výpisu 32.9. Z výpisu jsem pouze vyjmul metody sloužící k uchování aktuálních stavů pro umožnění vrácení tahů a jejich případné opětné provedení. Jak jsem již řekl, touto problematikou se budeme zabývat v kapitole *Zpátky na stromy (Pamětník – Memento)* na straně 467.

Výpis 32.9: Definice třídy `Model`, jejíž instance mají na starosti vlastní logiku hry; ve výpisu nejsou zobrazeny metody související s vracením a opětným prováděním posledních tahů

```
package rup.česky.vzory._32_mvc.reversi;

import java.util.BitSet;
import java.util.Random;
import rup.česky.tvary.Pozice;
import rup.česky.tvary.Směr8;
import rup.česky.tvary.SprávcePlátna;


/*******************************************************************************
 * Instance třídy <code>Model</code> obsahují vlastní logiku hry.
 * Nezabývají se tím, jak bude průběh hry zobrazen, a starají se pouze
 * o principiální ošetření jednotlivých typů situací.
 */
public class Model implements IModel
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Generátor pseudonáhodných čísel pro znáhodnění výběru polí
     *  o stejné váze. */
    private static final Random rnd = new Random();


//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================================

    private final int       strana;  //Políčkový vodorovný/svislý rozměr desky
    private final Kámen[][] deska;   //Interní reprezentace hrací desky
    private final int       polí;    //Počet polí na hrací desce


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /*****************************************************************************
     * Vytvoří nový model hry na desce o zadaném vodorovném/svislém rozměru.
     * Tento model přijímá informace od uživatelského rozhraní a
```

```
     * modifikuje vnitřní reprezentaci hry, kterou uživatelské rozhraní
     * zobrazí nebo jinak zveřejní.
     *
     * @param strana Políčkový vodorovný/svislý rozměr desky
     */
    public Model( int strana )
    {
        rup.česky.společně.IO.rodičNa(250,0);
        this.strana = strana;
        this.polí   = strana * strana;
        this.deska  = new Kámen[strana+2][strana+2];
        //Nastavíme okrajové kameny kolem hracího pole
        for( int xy=0;   xy < strana+1;   xy++ ) {
            deska[0][xy] =
            deska[xy][0] =
            deska[xy][strana+1] =
            deska[strana+1][xy] = Kámen.OKRAJ;
        }
        //Do hracího pole umístíme prázdné kameny
        for( int x=1;   x <= strana;   x++ ) {
            Kámen[] sloupec = deska[x];
            for( int y=1;   y <= strana;   y++ )
                sloupec[y] = Kámen.PRÁZDNÝ;
        }
        //Doprostřed pole umístíme kameny hráče a počítače
        int s2 = strana/2;
        int s1 = s2 + 1;
        deska[s2][s2] =
        deska[s1][s1] = Kámen.HRÁČ;
        deska[s2][s1] =
        deska[s1][s2] = Kámen.POČÍTAČ;
        najdiTah( Kámen.HRÁČ );
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /****************************************************************************
     * Vrátí kámen na zadané pozici.
     *
     * @param x Vodorovná souřadnice pozice, levý okraj má x=1
     * @param y Svislá souřadnice pozice, horní okraj má y=1
     * @return  Kámen na zadané pozici
     */
    public Kámen getKámen( int x, int y )
    {
        return deska[x][y];
    }


    /****************************************************************************
     * Na zadanou pozici umístí kámen hráče a vrátí informaci o tom,
     * má-li hráč po tahu počítače ještě nějaký dostupný tah.
     * Umístění kamene počítače nepotřebuje speciální metodu,
     * protože počítač nedělá chyby, a není jej proto nutno tak hlídat.
     *
     * @param pozice Pozice pole, kam umístit kámen
     *               levý horní okraj má pozici [1;1]
```

```
     * @return Má-li hráč ještě k dispozici nějaký tah
     */
    public boolean setKámen( Pozice pozice )
    {
        Pozice tah;
        int x = pozice.x;
        int y = pozice.y;

        //Nahradit je možno pouze prázdné pole => je třeba zajistit, abychom
        //jiné druhy polí neoslovovali. Tyto "nevhodné" druhy polí
        //lze snadno poznat podle jejich záporné síly.
        if( (deska[x][y].getSíla() <= 0) ) {
            tah = najdiTah( Kámen.HRÁČ );  //Přepočte znovu síly polí
            return (deska[tah.x][tah.y].getSíla() > 0);
        }
        deska[x][y] = Kámen.HRÁČ;                 //Umístí kámen hráče
        // Překlopí všechny hráčem zajmuté kameny
        obraťKameny( x, y, Kámen.HRÁČ, Kámen.POČÍTAČ );

        boolean počítačTáhl, hráčMáTah;
        do{ //Nemůžeme přebírat výsledky mechanicky. Je třeba zabezpečit,
            //řešení nestandardních situací, kdy jeden z hráčů nemůže táhnout
            tah  = najdiTah( Kámen.POČÍTAČ );  //Najde protitah počítače

            SprávcePlátna.getInstance().překresli();
            rup.česky.společně.IO.čekej( 500 );

            //Zjistí, může-li počítač hrát
            if( počítačTáhl = (deska[tah.x][tah.y].getSíla() > 0) ) {
                //Může hrát - provede doporučený tah
                deska[tah.x][tah.y] = Kámen.POČÍTAČ;   //Umístí kámen počítače
                obraťKameny( tah.x, tah.y, Kámen.POČÍTAČ, Kámen.HRÁČ );
            }
            tah = najdiTah( Kámen.HRÁČ );       //Spočte sílu polí pro hráče
            hráčMáTah = (deska[tah.x][tah.y].getSíla() > 0);
        //Nemůže-li hráč hrát, nabídne opět tah počítači
        }while( !hráčMáTah   &&   počítačTáhl  );
        return hráčMáTah;
    }

// ... Metody pro získání a nastavení stavu související s možností vracení tahů

    /*************************************************************************
     * Vrátí řetězec uvádějící, kolik kamenů má právě hráč a kolik počítač.
     */
    public String stavSlovy()
    {
        int hráč=0, počítač = 0;
        for( int x=1;   x <= strana;   x++ )
            for( int y=1;   y <= strana;   y++ ) {
                Kámen kámen = deska[x][y];
                    if( kámen == kámen.HRÁČ )
                        hráč++;
                    else if( kámen == Kámen.POČÍTAČ )
                        počítač++;
            }
        return "Hráč: " + hráč + ",  Počítač: " + počítač;
    }
```

```
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ================================================

    /**************************************************************************
     * Najde optimální tah zadaného hráče a při té příležitosti
     * přepočte sílu všech volných polí.
     *
     * @param hráč Hráč, pro kterého se hledá optimální tah
     * @return Optimální tah = tak na pole s největší silou
     */
    private Pozice najdiTah( Kámen hráč )
    {
        Pozice tah = new Pozice( 0, 0 );
        int    max = 0;
        Kámen  soupeř = (hráč == Kámen.HRÁČ)  ?  Kámen.POČÍTAČ
                                              :  Kámen.HRÁČ;
        //Projde všechna volná pole, vyhodnotí je a vrátí jedno z těch,
        //která se ukázala jako nejsilnější
        for( int x=1;   x <= strana;   x++ ) {
            for( int y=1;   y <= strana;   y++ ) {
                if( deska[x][y].getTyp() != Kámen.Typ.PRÁZDNÝ )
                    continue;
                int síla = sílaPole( x, y, hráč, soupeř );
                if( (síla > max)    ||
                   ((síla > 0)  &&  (síla == max)  &&  (rnd.nextInt(2) == 0)) )
                {
                    max = síla;
                    tah = new Pozice( x, y );
                }
            }
        }
        return tah;
    }


    /**************************************************************************
     * Zjistí sílu zadaného prázdného pole a
     * všechny kameny ovlivňované tímto polem překlopí.
     *
     * @param x       Vodorovná souřadnice pole, jehož sílu zjišťujeme
     * @param y       Svislá souřadnice pole, jehož sílu zjišťujeme
     * @param hráč    Kámen hráče, pro nějž zjišťujeme sílu pole
     * @param soupeř  Kámen soupeře
     * @return Informace o síle zadaného pole
     */
    private void obraťKameny( int x, int y, Kámen hráč, Kámen soupeř )
    {
        analyzuj( x, y, hráč, soupeř, true );
    }


    /**************************************************************************
     * Zjistí sílu zadaného prázdného pole a umístí do něj prázdný kámen,
     * který si bude zjištěnou sílu pamatovat.
     *
     * @param x       Vodorovná souřadnice pole, jehož sílu zjišťujeme
     * @param y       Svislá souřadnice pole, jehož sílu zjišťujeme
```

```
     * @param hráč    Kámen hráče, pro nějž zjišťujeme sílu pole
     * @param soupeř  Kámen soupeře
     * @return Informace o síle zadaného pole
     */
    private int sílaPole( int x, int y, Kámen hráč, Kámen soupeř )
    {
        return analyzuj( x, y, hráč, soupeř, false );
    }



    /**************************************************************************
     * Zjistí sílu zadaného prázdného pole
     * a podle požadavku zadaného v parametru <code>překlop</code>
     * do něj buď umístí prázdný kámen, který si bude zjištěnou sílu pamatovat,
     * nebo všechny kameny ovlivňované tímto polem překlopí.
     *
     * @param x       Vodorovná souřadnice pole, jehož sílu zjišťujeme
     * @param y       Svislá souřadnice pole, jehož sílu zjišťujeme
     * @param hráč    Kámen hráče, pro nějž zjišťujeme sílu pole
     * @param soupeř  Kámen soupeře
     * @param překlop <code>true</code> mají-li se kameny na ovlivněných
     *                polích překlopit, <code>false</code> má-li se pouze
     *                zjistit síla pole a umístit na něj příslušný kámen
     * @return Informace o síle zadaného pole
     */
    private int analyzuj( int x, int y, Kámen hráč, Kámen soupeř,
                          boolean překlop )
    {
        int součet = 0;
        for( Směr8 směr : Směr8.values() ) {
            if( směr == Směr8.ŽÁDNÝ )
                continue;   //Směr ŽÁDNÝ přeskočíme
            součet += sílaVeSměru( směr, x, y, hráč, soupeř, překlop );
        }
        if( !překlop )
            deska[x][y] = Kámen.getInstance( součet );
        return součet;
    }


    /**************************************************************************
     * Zjistí pro pole na zadaných souřadnicích jeho sílu v zadaném směru
     * pro zadaného hráče.
     *
     * @param směr    Směr, pro nějž se síla pole zjišťuje
     * @param x       Vodorovná souřadnice pole, jehož sílu zjišťujeme
     * @param y       Svislá souřadnice pole, jehož sílu zjišťujeme
     * @param hráč    Kámen hráče, pro nějž zjišťujeme sílu pole
     * @param soupeř  Kámen soupeře
     * @param překlop <code>true</code> mají-li se kameny na ovlivněných
     *                polích překlopit, <code>false</code> má-li se pouze
     *                zjistit síla pole a umístit na něj příslušný kámen
     * @return Síla zadaného pole v zadaném směru
     */
    private int sílaVeSměru( Směr8 směr, int x, int y,
                             Kámen hráč, Kámen soupeř, boolean překlop )
    {
```

```
        int dx = směr.dx();
        int dy = směr.dy();
        int xx = x + dx;
        int yy = y + dy;
        int ss = 0;
        while( deska[xx][yy] == soupeř )
        {
            ss++;
            xx += dx;
            yy +=dy;
        }
        if( deska[xx][yy] == hráč ) {
            if( (ss > 0)  &&  překlop ) {
                int ještě = ss; //Počet kamenů k překlopení
                do {
                    xx -= dx;
                    yy -= dy;
                    deska[xx][yy] = hráč;
                }while( --ještě > 0 );
            }
            return ss;
        }else{
            return 0;
        }
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

// ... Definice tříd souvisejících s operacemi vracení tahů
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Model-Pohled-Ovládání* (MPO) je sice v GoF zmíněn (je tam rozebírán na třech stranách), ale nepatří mezi vzory oficiálně uvedené v GoF. Je totiž považován za skupinu vzorů.
- Vzor doporučuje rozdělit části programu zabývající se předzpracováním požadavků uživatele, vlastní logikou programu a prezentací získaných výsledků do samostatných modulů.
- Do části zabývající se prezentací výsledků patří nejenom vlastní GUI, ale často i kód, který data pro tuto prezentaci připravuje.
- Aplikace vzoru umožňuje snadnou implementaci zadávání požadavků různými způsoby (klávesnice, myš, pero, hlas, …) stejně jako různé možnosti prezentace výsledků (tabulka, graf, mluvené slovo, …).
- Další výhodou takovéhoto rozdělení je usnadnění případných budoucích změn.
- Změny požadavků na způsob zadávání požadavků a prezentaci výsledků patří k nejčastějším.
- Aplikace vzoru usnadňuje přenositelnost mezi platformami.
- Vzor je spíše obecným doporučením, které jednotlivé vazby nijak konkrétně nespecifikuje.
- V jednodušších aplikacích se některé části často slučují.

KAPITOLA 33

Tohle ještě neumíš (Návštěvník – Visitor)

- **Účel**
- **Implementace**
- **Příklad**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Umožňuje definovat pro skupinu tříd nové operace jejich instancí, aniž by bylo nutno jakkoliv měnit kód těchto tříd.

[1] **Definice v GoF:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. – Reprezentuje operaci, která má být provedena na prvcích dané datové struktury. *Návštěvník* umožňuje definovat nové operace, aniž by bylo třeba měnit třídy objektů, nad nimiž jsou operace prováděny.

Účel

560. Jestli jsem to dobře pochopil, tak tento vzor umožňuje definovat pro nějakou skupinu tříd nové metody jejich instancí někde mimo tyto třídy. To taky jde?

Vzor je těžký na pochopení

Když to uděláš šikovně, tak ano. Musím ale dopředu upozornit, že tento návrhový vzor je asi ze všech vzorů nejsložitější a řada programátorů má s jeho pochopením problémy.

561. O to lépe jej budeš muset vysvětlit. Tak povídej!

Určení

Návrhový vzor *Návštěvník* je určen pro situace, kdy máš poměrně stabilní skupinu tříd s relativně nestabilní množinou metod. Jinými slovy: máš skupinu tříd, u níž nepředpokládáš, že do ní budeš přidávat nebo z ní ubírat její členy, avšak obáváš se, že v průběhu života programu budeš muset jejím třídám přidávat nějaké metody.

562. To přece dělám každou chvíli.

Ano, ale děláš to tak, že upravuješ kód těchto tříd. Představ si ale, že dané třídy jsou součástí nějaké uzavřené knihovny a jejich kód již nemůžeš měnit. Jsou-li však instance těchto tříd schopny přijmout návštěvníky, můžeš tyto metody definovat ve svých vlastních třídách.

563. Ty tu na mne šiješ nějakou boudu. Jak mohu u sebe definovat nové metody pro instance tříd, jejichž zdrojový kód nemohu nijak ovlivnit?

Jak definovat nové metody pro třídy s nezměnitelným kódem

Právě aplikací návrhového vzoru *Návštěvník*. Rozšiřované třídy ale musí použití tohoto návrhového vzoru podporovat a být pro přijetí návštěvníků připraveny. Dokážou to tím, že implementují rozhraní, jež deklaruje metodu schopnou přijmout návštěvníka.

Pak pro každou metodu, kterou chceš instance dané skupiny tříd naučit, definuješ zvláštní třídu návštěvníků. Budeš-li chtít provést metodu definovanou návštěvníkem, pošleš oslovované instanci návštěvníka (tj. zavoláš její metodu na přijímání návštěvníků) a necháš je, ať spolu potřebnou činnost realizují.

564. Už chápu, proč tento vzor někteří programátoři nechápou. Já ho zatím také moc nechápu. Stále to vypadá jako děsná magie.

Dobře, zkusíme to znovu, ale tentokrát již hned na nějakém příkladu. Při procházení implementace vzoru možná celý princip pochopíš.

Implementace

Příklad:

565. Dobře, ale vyber něco doopravdy jednoduchého.

Příjem návštěvníka objekty

Jasně. Představ si např. moji knihovnu tvarů kreslených prostřednictvím správce plátna. Všechny třídy související s kreslením tvarů jsou v této knihovně definovány jako potomci třídy `APosuvný`. Ta definuje metodu

```
public Object přijmi( INávštěvník návštěvník, Object param )
```

Nelze použít dědění

Tato metoda má zajímavou vlastnost: chceš-li ji plnohodnotně implementovat, musí ji každá třída překrýt vlastní verzí, přestože jsou všechny definice totožné.

566. Proboha! Další magie! Proč?

Za chvíli ti to vysvětlím, Teď se na chvíli smiř s tím, že tuto metodu nesmí třídy jenom zdědit, ale musí ji překrýt vlastní verzí.

567. No dobrá, pokračuj.

Rozhraní implementované návštěvníky

Všechny třídy návštěvníků musí implementovat rozhraní `INávštěvník`. Jedině tak totiž mohou být jejich instance přijaty výše zmíněnou metodou jako návštěvníci. Rozhraní `INávštěvník` (viz výpis 33.1) požaduje po implementujících třídách implementaci série metod

```
public Object aplikujNa( Ttt navštívený, Object param )
```

kde za `Ttt` dosazuješ postupně všechny navštívitelné třídy. Pro každou z navštěvovaných tříd tak musí mít návštěvník definovánu speciální metodu. Tato metoda má za úkol splnění příslušného úkolu určeného instanci navštívené třídy.

Výpis 33.1: Rozhraní `INávštěvník`, které musí implementovat všichni návštěvníci

```
package rup.česky.tvary;


/*****************************************************************************
 * Instance rozhraní INávštěvník slouží k rozšíření schopností
 * navštívených objektů.
 */
public interface INávštěvník
{
//== DEKLAROVANÉ METODY ========================================================

    public Object aplikujNa( AHýbací     navštívený, Object param );
    public Object aplikujNa( APosuvný    navštívený, Object param );
    public Object aplikujNa( Čára        navštívený, Object param );
    public Object aplikujNa( Čtverec     navštívený, Object param );
    public Object aplikujNa( Elipsa      navštívený, Object param );
    public Object aplikujNa( Kruh        navštívený, Object param );
    public Object aplikujNa( Obdélník    navštívený, Object param );
    public Object aplikujNa( Obrázek     navštívený, Object param );
    public Object aplikujNa( Text        navštívený, Object param );
    public Object aplikujNa( Trojúhelník navštívený, Object param );

//== VNOŘENÉ TŘÍDY ============================================================

// ...  Vynechaná definice adaptéru

}
```

Základní „fígl"

Celý fígl spočívá v tom, umět zařídit, aby navštívená instance spustila svoji verzi metody návštěvníka, protože ta je pro ni optimalizovaná.

Společná podoba definice přijímací metody

Jestli si vzpomínáš, tak jsem říkal, že všechny navštěvované třídy definují metodu `přijmi(INávštěvník,Object)`, a že jí dokonce definují stejně. Jejich definice je následující:

```
public Object přijmi( INávštěvník návštěvník, Object param ) {
    return návštěvník.aplikujNa( this, param );
}
```

Podstata onoho fíglu spočívá právě v tom, že každá třída definuje svoji vlastní verzi této metody. Tím, že prvním předávaným parametrem je instance této třídy, je v definici jednoduše zařízeno, že se zavolá správná verze návštěvníkovy metody optimalizovaná pro instance dané třídy.

568. Magie, magie, magie. Ale myslím, že to začínám trochu chápat. (Za chvíli ze mne bude také čaroděj.) Abych to pochopil ještě lépe, ukaž mi definici nějaké návštěvnické třídy a popiš mi, jak to vše probíhá.

Adaptér návštěvníka

Dobře. Nejprve ti ale musím prozradit, že rozhraní `INávštěvník` má (jako většina mých rozhraní) definovanou vnořenou třídu `Adaptér`, která poskytuje implicitní implementaci návštěvníka těm, kteří si chtějí ušetřit práci. Její definici najdeš ve výpisu 33.2.

Výpis 33.2: Třída `INávštěvník.Adaptér` s implicitní implementací návštěvníka

```
// ...  Vynechané části definice

//== VNOŘENÉ TŘÍDY ============================================================

    /*****************************************************************************
     * Třída <code>INávštěvník.Adaptér</code>
     * definuje adaptér pro třídy, které z nějakého důvodu musí implementovat
     * rozhraní {@link INávštěvník},
     * avšak nechtějí implementovat všechny jeho metody.
     */
    public static class Adaptér implements INávštěvník
    {
    //== NESOUKROMÉ METODY INSTANCÍ ===============================================

        public Object aplikujNa( APosuvný navštívený, Object param) {
            throw new UnsupportedOperationException();
        }

        public Object aplikujNa( Čára navštívený, Object param ) {
            return aplikujNa((APosuvný)navštívený, param );
        }
        public Object aplikujNa( Text navštívený, Object param ) {
            return aplikujNa((APosuvný)navštívený, param );
        }
        public Object aplikujNa( AHýbací navštívený, Object param )  {
            return aplikujNa((APosuvný)navštívený, param );
        }

        public Object aplikujNa( Obrázek navštívený, Object param ) {
            return aplikujNa((AHýbací)navštívený, param );
        }
        public Object aplikujNa( Trojúhelník navštívený, Object param){
            return aplikujNa((AHýbací)navštívený, param );
        }

        public Object aplikujNa( Elipsa navštívený, Object param ) {
            return aplikujNa((AHýbací)navštívený, param );
        }
        public Object aplikujNa( Kruh navštívený, Object param ) {
            return aplikujNa((Elipsa)navštívený, param );
        }
```

```
        public Object aplikujNa( Obdélník navštívený, Object param ) {
            return aplikujNa((AHýbací)navštívený, param );
        }
        public Object aplikujNa( Čtverec navštívený, Object param ) {
            return aplikujNa((Obdélník)navštívený, param );
        }
    }
```

Dědičnost v adaptéru

Než půjdeme dále, chtěl bych tě upozornit na to, že tento adaptér počítá s tím, že by rodičovská třída mohla definovat verzi metody, která by platila i pro potomka. Při jeho použití proto nemusíš definovat všechny metody, ale stačí ti pouze ty, v nichž se realizace metody pro potomka liší od realizace metody pro předka.

Příklad: Návštěvník Plocha

Této možnosti využívá i třída `NávštěvníkPlocha`, jejímž úkolem je spočítat plochu navštívených instancí a jejíž definici i s testem najdeš ve výpisu 33.3. Tato třída definovala svoji vlastní verzi metod pouze pro instance tříd `Čára`, `Elipsa`, `Obdélník`, `Obrázek` a `Trojúhelník`. Pro ostatní využila definice zděděné od třídy `INávštěvník.Adaptér`.

Dvě skupiny zděděných metod

Funkce těchto zděděných metod bychom mohli rozdělit do dvou skupin: metody pro instance tříd `Kruh` a `Obdélník` přebírají rodičovskou verzi, která opravdu něco spočítá, kdežto ostatní metody vedou nakonec k vyhození výjimky `UnsupportedOperationException`.

Výpis 33.3: Definice třídy `NávštěvníkPlocha`, jejíž instance počítají plochu navštívených instancí

```
package rup.česky.vzory._33_návštěvník;

import java.util.List;

import rup.česky.tvary.APosuvný;
import rup.česky.tvary.SprávcePlátna;
import rup.česky.tvary.Elipsa;
import rup.česky.tvary.IKreslený;
import rup.česky.tvary.INávštěvník;
import rup.česky.tvary.Kruh;
import rup.česky.tvary.Obdélník;
import rup.česky.tvary.Text;
import rup.česky.tvary.Trojúhelník;
import rup.česky.tvary.Čtverec;
import rup.česky.tvary.Čára;


/*******************************************************************************
 * Instance třídy NávštěvníkPlocha představují návštěvníky,
 * kteří mají za úkol zjistit velikost plochy navštíveného obrazce.
 */
public class NávštěvníkPlocha extends INávštěvník.Adaptér
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    public Double aplikujNa( Čára c, Object param ) {
        return new Double( 0 );
    }
```

```
    public Double aplikujNa( Obdélník o, Object param ) {
        return new Double( o.getŠířka() * o.getVýška() );
    }


    public Object aplikujNa( Obrázek obr, Object param ) {
        return new Double( obr.getŠířka() * obr.getVýška() );
    }


    public Double aplikujNa( Elipsa e, Object param ) {
        return new Double( e.getŠířka() * e.getVýška() * Math.PI / 4 );
    }


    public Double aplikujNa( Trojúhelník t, Object param ) {
        return new Double( t.getŠířka() * t.getVýška() * 0.5 );
    }



//== TESTY A METODA MAIN ================================================

    /*************************************************************************
     * Metoda nakreslí několik objektů, pak požádá správce plátna o jejich
     * seznam a každý z objektů v seznamu navštíví, aby s ním provedla
     * požadovanou akci - výpočet jeho plochy.
     */
    public static void test2()
    {
        SprávcePlátna SP = SprávcePlátna.getInstance();
        SP.přidej( new Čtverec()     );
        SP.přidej( new Obdélník()    );
        SP.přidej( new Elipsa()      );
        SP.přidej( new Kruh()        );
        SP.přidej( new Trojúhelník() );
        SP.přidej( new Text( "Xxx" ) );

        //Návštěvník, který umí spočítat plochu navštíveného tvaru
        NávštěvníkPlocha np = new NávštěvníkPlocha();

        //Spočteme plochu všech objektů nakreslených na plátně
        List<IKreslený> tvary = SP.seznamKreslených();
        for( IKreslený ik : tvary ) {
            try {
                System.out.println( ik + " - plocha " +
                                    ((APosuvný)ik).přijmi( np, null ) );
            }catch( Exception ex ) {
                System.out.println( ik + " - INSTANCE TYPU " + ik.getClass() +
                    " - PLOCHU NENÍ MOŽNO SPOČÍTAT" );
            }
        }
    }
    /** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {  test2();  }
}
```

569. Proč metody `NávštěvníkPlocha` vracejí hodnoty typu `Double`, když implementované rozhraní vyžaduje vracení hodnoty typu `Object`?

Metoda potomka může vracet potomka typu vraceného metodou předka

Od verze 5 Java akceptovala v syntaktických pravidlech skutečnost, že instance potomka se může kdykoliv vydávat za instanci předka, a umožnila, aby překrývající metody vracely hodnotu typu, který je potomkem typu vraceného překrytou metodou. V řadě případů se tak ušetří přetypování.

Třída `NávštěvníkPlocha` této skutečnosti využívá. Výhodou tohoto přístupu je navíc to, že překladač zkontroluje, že všechny metody opravdu vracejí hodnotu typu `Double`.

570. Co kdybychom nechtěli, aby metody vyhazovaly výjimky? To by mělo stačit, kdybychom definovali pro instance `APosuvný` verzi, která vrací nulu.

Třeba. K tomu by stačilo pouze změnit typ prvního parametru u metody pro čáry, která také vrací nulu. Jak vidíš, vhodným použitím adaptéru si v podobných případech můžeš ušetřit dost práce.

571. Takže teď ještě zbývá popsat, jak přesně bude probíhat test.

Průběh testu:

Dobře. Připrav se ale na to, že to bude hutné a bude vyžadovat dost úsilí celý postup sledovat.

– příprava

Začátek vynechám a zastavím se až u cyklu, který prochází seznam kreslených objektů vrácený správcem plátna.

Prvním obdrženým objektem bude čtverec, protože jsme jej i na plátno přidávali jako první.

– voláme `přijmi()`

Začneme skládat vystupující text. Při tom potřebujeme zavolat metodu `přijmi(INávštěvník,Object)`, avšak její implementaci rozhraní `IKreslený`, za jehož instanci obdržený objekt stále považujeme, neslibuje.

Přetypujeme tedy získaný objekt na `APosuvný` – to můžeme, protože víme, že všechny tvary jsou potomky této třídy, nenastane zde žádný problém. Pak zavoláme metodu `přijmi(INávštěvník,Object)` čtverce.

– ta volá `aplikujNa (Čtverec)`

Ta zavolá metodu `aplikujNa(Čtverec,Object)` návštěvníka. To, že to bude právě tato přetížená verze, je důsledkem toho, že metoda je volána z těla metody definované ve třídě `Čtverec` a jejím prvním parametrem je `this`. Kdybychom byli líní a rozhodli se ve všech třídách použít pouze zděděné verze této metody, volala by se pokaždé přetížená verze `aplikujNa(APosuvný,Object)`.

To, která verze metody se zavolá, rozhoduje totiž překladač v době příkladu podle deklarované verze parametru. Jaký bude jeho skutečný typ za běhu, totiž nemůže překladač vědět ani odhadnout. Proto také musíme poctivě v každé třídě překrýt rodičovskou verzi této metody novou verzí, a to přesto, že celou překrývanou metodu včetně těla stačí pouze zkopírovat.

– ta volá `aplikujNa (Obdélník)`

Takže zpět do programu. Jeho „krokování“ jsme opustili po zavolání metody `aplikujNa(Čtverec,Object)`.

Tuto metodu dědí náš návštěvník od adaptéru. Jeho verze místo sebe volá metodu `aplikujNa(Obdélník,Object)`.

– ta spočte a vrátí obsah

Ta spočítá obsah a vrátí jej metodě, která ji zavolala, tj. metodě `aplikujNa(Čtverec,Object)`.

– `aplikujNa(Čtverec)` udělá totéž

Ta bez přemýšlení vrátí obdržený výsledek své volající metodě, tj. metodě `přijmi(INávštěvník,Object)`.

– stejně tak `přijmi()`

Ta obdržený výsledek také ihned vrátí a vrácený výsledek se použije v testovací metodě při sestavování řetězce, který se bude tisknout.

– získaná hodnota se vytiskne

Obdobně se budou zpracovávat i další nakreslené objekty.

572. Uff, to byla dávka. Ještě si to dvakrát přečtu a možná to budu i chápat. Mám teď ale jiný dotaz. Stále ses ještě nezmínil o tom, k čemu je dobrý druhý parametr, do nějž jsme prozatím pokaždé dávali jenom prázdný odkaz `null`.

Význam druhého parametru

Ten je určen pro případ, kdy by návštěvníkovy metody potřebovaly nějaké parametry. Pokud by parametrů bylo více, můžeš si vybrat, zda je budeš předávat v poli anebo v přepravce. Není-li jich proměnný počet, přimlouval bych se za přepravku, protože se pak uplatní typová kontrola.

Příklad

573. Nemáš nějaký příklad?

Příklad: `NávštěvníkObrys`

Mám. Připravil jsem třídu `NávštěvníkObrys`, která obkreslí navštívený objekt čárou zadané barvy a tloušťky. Tato barva a tloušťka jsou právě ony parametry, které je třeba návštěvníkově metodě předat. Definoval jsem pro ně přepravku `PObrys`, v níž cestují až k místu, kde budou doopravdy použity. Její definici najdeš ve výpisu 33.4.

Třída `NávštěvníkObrys` (definici najdeš ve výpisu 33.5) pak přidává na plátno rámy, které se dokonce dokážou přizpůsobovat souřadnicím a rozměrům svých vzorů. Metoda návštěvníka je definována pro všechny hýbací objekty, přičemž obdélník, trojúhelník a elipsa mají ještě své vlastní definice. Z posuvných objektů je definována pouze pro text, protože u něj se dá zjistit jeho rozměr, který je k orámování potřeba.

Výpis 33.4: Třída `PObrys` definuje přepravku pro parametry návštěvníkovy metody

```
package rup.česky.vzory._33_návštěvník;

import rup.česky.tvary.Barva;

/*****************************************************************************
 * Instance třídy PObrys představují přepravky uchovávající data
 * požadovaná k vytvoření obrysu.
 */
class PObrys
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Požadovaná tloušťka obrysových čar v bodech.*/ public final int   síla;
    /** Požadovaná barva obrysových čar.           */ public final Barva barva;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /***************************************************************************
```

```
     * Vytvoří novou přepravku na hodnoty obdržené jako parametry.
     * @param síla  Požadovaná tloušťka obrysových čar v bodech
     * @param barva Požadovaná barva obrysových čar.
     */
    public PObrys( int síla, Barva barva ) {
        this.síla  = síla;
        this.barva = barva;
    }
}
```

Výpis 33.5: Třída `NávštěvníkObrys` definuje návštěvnickou metodu, která doplní objekt obrysem zadané barvy a tloušťky

```
package rup.česky.vzory._33_návštěvník;

import java.awt.Rectangle;
import java.awt.font.FontRenderContext;
import java.util.ArrayList;
import java.util.List;

import rup.česky.společně.IO;
import rup.česky.tvary.*;

import static rup.česky.tvary.SprávcePlátna.SP;


/*******************************************************************************
 * Instance třídy TestNávštěvník představují ...
 */
public class NávštěvníkObrys extends INávštěvník.Adaptér
{
//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*******************************************************************************
     * Obkreslí zaujatou oblast obdélníkovým rámem zadané barvy a síly;
     * bude použita pro všechny {@link AHýbací} objekty s výjimkou
     * instancí tříd {@link Elipsa} a {@link Trojúhelník},
     * které mají svoji vlastní obsluhu.
     * @param a     Obkreslovaný objekt
     * @param param Přepravka s tloušťkou a barvou obrysu
     * @return      Objekt, který bude obrysem zadaného objektu
     */
    public Object aplikujNa( final AHýbací a, Object param ) {
        return obkreslovač( a, (PObrys)param,
            new IObrys(){
                public void nakresli( Kreslítko k, int x, int y, int s, int v,
                                      Barva b )
                {
                    k.kresliRám( x, y, s, v, b );
                }
            }
        );
    }


    /*******************************************************************************
     * Obkreslí elipsu eliptickým rámečkem zadané barvy a tloušťky.
```

```
     * Uvnitř obkreslovaného rámu se bohužel mohou vyskytnout
     * prosvítající body původní elipsy.
     * @param e     Obkreslovaný objekt
     * @param param Přepravka s tloušťkou a barvou obrysu
     * @return      Objekt, který bude obrysem zadaného objektu
     */
    public Object aplikujNa( final Elipsa e, Object param ) {
        return obkreslovač( e, (PObrys)param,
            new IObrys(){
                public void nakresli( Kreslítko k, int x, int y, int s, int v,
                                      Barva b )
                {
                    k.kresliOvál( x, y, s, v, b );
                }
            }
        );
    }


    /*************************************************************************
     * Obkreslí trojúhelník rámečkem zadané barvy a tloušťky.
     * U šikmých čar však nebývá tloušťka konstantní - to by vyžadovalo
     * složitější výpočet vrcholů rámu.
     * @param t     Obkreslovaný objekt
     * @param param Přepravka s tloušťkou a barvou obrysu
     * @return      Objekt, který bude obrysem zadaného objektu
     */
    public Object aplikujNa( final Trojúhelník t, Object param ) {
        return obkreslovač( t, (PObrys)param,
            new IObrys(){
                public void nakresli( Kreslítko k, int x, int y, int s, int v,
                                      Barva b )
                {
                    int[][] points = getVrcholy( x, y, s, v, t.getSměr() );
                    k.kresliPolygon( points[0], points[1], b );
                }
            }
        );
    }


    /*************************************************************************
     * Obkreslí text obdélníkovým rámečkem zadané barvy a tloušťky.
     * Aby nezakryl rámovaný text, nechává (na rozdíl od ostatních tvarů)
     * rám růst od testu ven.
     * @param x     Obkreslovaný objekt
     * @param param Přepravka s tloušťkou a barvou obrysu
     * @return      Objekt, který bude obrysem zadaného objektu
     */
    public Object aplikujNa( final Text x, Object param ) {
        Rectangle r = x.getFont()
                       .getStringBounds( x.getNázev(),
                                  new FontRenderContext( null, false, false ) )
                       .getBounds();//FontRenderContext frc)
        final int s = ((PObrys)param).síla;
        final AHýbací ah =
            new AHýbací( x.getX() -  s,  x.getY() -  s,
                      r.width  + 2*s, r.height + 2*s )
```

```
            {
                 public void nakresli( Kreslítko k ) {}
                 public int getX() { return (x.getX() -  s); }
                 public int getY() { return (x.getY() -  s); }
            };
        SP.přidejNad( x, ah );
        return obkreslovač( ah, (PObrys)param,
            new IObrys(){
                public void nakresli( Kreslítko k, int x, int y, int s, int v,
                                      Barva b )
                {
                    k.kresliRám( x, y, s, v, b );
                }
            }
        );
    }


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ============================================
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ =========================================

    /**************************************************************************
     * Společná metoda pro zarámování objektu rámem zadané barvy a tloušťky.
     * Protože různé obrazce vyžadují různé tvary rámu, využívá vzor
     * <i>Příkaz</i> a nechává si jako parametr dodat objekt, který umí
     * v potřebné pozici a velikosti vykreslit rám požadovaného tvaru.
     */
    private IKreslený obkreslovač( final AHýbací ah, final PObrys param,
                                   final IObrys obrys )
    {
        IKreslený ret = new IKreslený() {
            public void nakresli( Kreslítko k ) {
                int x = ah.getX();
                int y = ah.getY();
                int s = ah.getŠířka();
                int v = ah.getVýška();
                for( int i = param.síla;
                     (--i >= 0)  &&  (s > 0)  &&  (v > 0);
                     x++, y++, s-=2, v-=2 )
                {
                     obrys.nakresli( k, x, y, s, v, param.barva );
                }
            }
        };
        SP.přidejNad( ah, ret );
        return ret;

    }


    /**************************************************************************
     * Pomocná metoda pro výpočet vrcholů trojúhelníku.
     * Vrátí matici se souřadnicemi vrcholů daného trojúhelníku.
     *
     * @return Požadovaná matice
     */
    private int[][] getVrcholy( int gx, int gy, int gš, int gv, Směr8 směr )
    {
```

```
        int[] x = null;
        int[] y = null;

        switch( směr )
        {
            case VÝCHOD:
                x = new int[]{ gx, gx+gš,     gx    };
                y = new int[]{ gy, gy+(gv/2), gy+gv };
                break;

            case SEVEROVÝCHOD:
                x = new int[]{ gx, gx+gš, gx+gš };
                y = new int[]{ gy, gy,    gy+gv };
                break;

            case SEVER:
                x = new int[]{ gx,    gx+(gš/2), gx+gš };
                y = new int[]{ gy+gv, gy,        gy+gv };
                break;

            case SEVEROZÁPAD:
                x = new int[]{ gx,    gx, gx+gš };
                y = new int[]{ gy+gv, gy, gy    };
                break;

            case ZÁPAD:
                x = new int[]{ gx,        gx+gš, gx+gš };
                y = new int[]{ gy+(gv/2), gy,    gy+gv };
                break;

            case JIHOZÁPAD:
                x = new int[]{ gx, gx,    gx+gš };
                y = new int[]{ gy, gy+gv, gy+gv };
                break;

            case JIH:
                x = new int[]{ gx, gx+(gš/2), gx+gš };
                y = new int[]{ gy, gy+gv,     gy    };
                break;

            case JIHOVÝCHOD:
                x = new int[]{ gx,    gx+gš, gx+gš };
                y = new int[]{ gy+gv, gy+gv, gy    };
                break;

            default:
                throw new IllegalStateException(
                    "Instance ukazuje do nedefinovaného směru" );
        }
        return new int[][] { x, y };
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    /**************************************************************************
     * Společné rozhraní definující vlastnosti objektů,
     * které budou sloužit jako obrysy svých vzorů.
```

```
     */
    private interface IObrys {
        /**************************************************************************
         * Nakreslí zadaným kreslítkem požadovaný obrys zadaného rozměru,
         * na zadaných souřadnicích a zadanou barvou.
         * @param k Dodané kreslítko
         * @param x Vodorovná souřadnice
         * @param y Svislá souřadnice
         * @param s Šířka obrazce obrysu
         * @param v Výška obrazce obrysu
         * @param b Barva obrysu
         */
        public void nakresli( Kreslítko k, int x, int y, int s, int v,
                              Barva b );
    }


//== TESTY A METODA MAIN ======================================================

    /**************************************************************************
     * Testovací metoda.
     */
    public static void test()
    {
        SprávcePlátna SP = SprávcePlátna.getInstance();
        SP.přidej( new Obdélník(   0,   0, 100, 50 ) );
        SP.přidej( new Elipsa  ( 150,   0, 100, 50 ) );
        SP.přidej( new Trojúhelník(0,  50, 100, 50 ) );
        SP.přidej( new Čtverec ( 100,  50,  50     ) );
        SP.přidej( new Kruh    ( 200,  50,  50     ) );
        SP.přidej( new Text    (  50, 150, "Xxx Yy") );

        NávštěvníkObrys no = new NávštěvníkObrys();
        PObrys po = new PObrys( 3, Barva.ČERNÁ );

        //Musím udělat kopii seznamu, protože jinak by se iterátor zlobil,
        //že mu v průběhu iterace přidávám do seznamu nové prvky
        List<IKreslený> tvary = new ArrayList<IKreslený>(SP.seznamKreslených());
        List<IKreslený> rámy  = new ArrayList<IKreslený>();

        for( IKreslený ik : tvary ) {
            APosuvný ap = (APosuvný)ik;
            try {
                rámy.add( (IKreslený)ap.přijmi( no, po ) );
            }catch( Exception ex ) {
                System.out.println( ap + " - INSTANCE TYPU " + ap.getClass() +
                    " - PLOCHU NENÍ MOŽNO ORÁMOVAT\n" );
                ex.printStackTrace();
            }
        }
        IO.zpráva( "Výchozí pozice" );
        for( IKreslený ik : tvary ) {
            APosuvný ap = (APosuvný)ik;
            ap.setPozice( ap.getX() + 50,  ap.getY() + 50 );
        }
        IO.zpráva( "Po rozstřelu" );
        for( IKreslený ik : rámy ) {
            SP.odstraň( ik );
```

```
        }
        IO.zpráva( "Po odrámování" );
    }
    /** @param args Parametry příkazového řádku - nepoužívané. */
    public static void main( String[] args )  {  test();  }
}
```

574. Proč mají tentokrát návštěvníkovy metody deklarován první parametr jako `final`?

První dvě jej tak mají definován pouze proto, aby „držely basu“ se zbylými dvěma. Ty používají tento parametr uvnitř definice anonymní třídy, a tam, jak jistě víš, je možno používat pouze lokální konstanty, nikoliv lokální proměnné.

Shrnutí – co jsme se naučili

- Návrhový vzor *Návštěvník* umožňuje definovat nové operace instancí, aniž by bylo nutno měnit kód jejich tříd.
- Hodí se v situacích, kdy se počet tříd již nemění, ale mění se počet metod, které musí umět.
- Pro každou metodu je třeba definovat zvláštní třídu návštěvníka.
- Instance třídy, která má spolupracovat s návštěvníkem, musí definovat metodu pro příjem návštěvníka.
- V těle metody je jediný příkaz: předání řízení metodě návštěvníka.
- Návštěvník musí implementovat rozhraní, které předepisuje přetíženou verzi metody pro každou navštěvovanou třídu.
- Aby překladač zvolil správnou přetíženou verzi, nesmí se navštěvované třídy spolehnout na dědění, ale každá musí definovat svoji vlastní, byť stále stejnou verzi přijímací metody.
- Pro rozhraní návštěvníka je možné definovat adaptér, který může usnadnit definici návštěvníkových metod.
- Při aplikaci návštěvníkovy metody zavoláme přijímací metodu instance, pro niž chceme metodu provést. Instance v ní zavolá svoji verzi návštěvníkovy metody, který vše provede místo instance a vrátí jí výsledek, který instance předá, jako by jej připravila sama.
- Návrhový vzor *Návštěvník* patří mezi vzory uvedené v GoF.

KAPITOLA 34

Zpátky na stromy (Pamětník – Memento)

- **Účel**
- **Implementace**
- **Příklad: Reversi s návraty**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]
Zabezpečuje uchovávání stavů objektů, aby je bylo v případě potřeby možno uvést do původního stavu. Přitom zabezpečuje, aby při uchovávání stavu nebylo narušeno ukrytí implementace.

[1] **Definice v GoF:** Without violating encapsulation, capture and externalize an object's internal state so, that the object can be restored to this state later. – Zachytává a ukládá stav objektů, aniž by narušil jejich zapouzdření. Tím umožňuje pozdější návrat objektů do uloženého stavu.

Účel

575. Tak tentokrát se nám relativní délka podkapitol asi prohodí. Většinou jsi mi sáhodlouze vysvětloval účel návrhového vzoru, abychom pak byli s implementací hotovi raz dva. Teď to vidím obráceně: k čemu je to dobré, mi je asi jasné, a bude mne docela zajímat, jak se to dá zařídit.

Pro jistotu bych ale účel tohoto návrhového vzoru přece jenom zopakoval pro případ, že by ti některé souvislosti utekly.

Účel vzoru

Jak jsi jistě pochopil, základním účelem tohoto návrhového vzoru je poskytnout doporučení, jak uchovávat stavy objektů, aby bylo možno později systém převést zpět do některého z předchozích stavů funkcemi označovanými v českých programech většinou slovy *zpět* (anglicky *undo*) a *znovu* (anglicky *redo*).

Hlavní problém

Základním problémem ukládání stavů je to, že při něm objekt potřebuje někam předat informace o svém vnitřním stavu, tj. informace, které se jinak snaží podle základů ukrývání implementace přede všemi maximálně zatajit. Návrhový vzor *Pamětník* se snaží řešit právě tento problém.

Implementace

576. Vidíš, to mne nenapadlo. A jak jej řeší?

Zástupci klíčových tříd

Než se rozpovídám o řešení, projděme si situaci, ve které by použití tohoto návrhového vzoru připadalo v úvahu. Pro větší názornost nebudu tentokrát hovořit o obecných třídách, ale ukážu ti vše na konkrétní situaci hry, u níž si chceme zapamatovat klíčové momenty, k nimž bychom se chtěli vrátit. Mějme tři typy:

- `Hra` je třída, jejíž instance umí nějakým způsobem archivovat svůj stav.
- `Správce` je třída, jejíž instance má na starosti uchovávání informací o stavech objektů a převádění objektů zpět do zapamatovaných stavů.
- `IPamětník` je rozhraní, o jehož instanci požádá instance třídy `Správce` instanci třídy `Hra` v okamžiku, kdy se uživatel rozhodne zapamatovat aktuální stav.

`IPamětník` může být jen značkovací

První zajímavou věcí je, že rozhraní `IPamětník` může být pouze značkovací, tj. nemusí mít definovanou žádnou metodu. Slouží totiž pouze k tomu, aby `Hra` předala instanci `Správce` do úschovy informace, do nichž nikomu nic není.

Pamětník je trezor

Instance rozhraní `IPamětník` je tedy obdoba přepravky, která však má všechny svoje datové i funkční členy přísně soukromé. Je to takový malý trezor, který slouží pouze k tomu, aby se tajné informace uložily někde, odkud je bude možno v případě potřeby opět získat.

Pamětník je většinou vnitřní třída

Jak si jistě domyslíš, nejlepším řešením je definovat třídu implementující rozhraní `IPamětník` jako vnitřní třídu třídy `Hra`. Z hlediska instancí třídy `Hra` budou instance této třídy běžné přepravky, avšak z hlediska okolního světa to budou trezory.

Komunikace hry s pamětníkem

Třída `Hra` pak bude definovat dvě metody. Metodu

```
public IPamětník getStav()
```

která vrátí instanci rozhraní `Pamětník`, v níž budou uloženy všechny informace potřebné k zrekonstruování aktuálního stavu, a metodu

```
public void setStav(IPamětník)
```

která ve svém parametru obdrží pamětníka stavu, do kterého se má hra vrátit, a na základě uložených informací se do něj doopravdy vrátí.

577. U her jsem ale často zvyklý, že si nepamatují nějaké definované stavy, ale že si pamatují posloupnost tahů, takže se můžeš vracet tah za tahem.

Kdy si pamatovat stavy a kdy tahy

Záleží na tom, o jakou hru se jedná. Jsou hry, v nichž je ukládání jednotlivých tahů rozumným řešením (např. většina deskových her), ale na druhou stranu jsou i takové, kde bys ukládáním každého tahu zbytečně spotřebovával paměť a hlavně návrat do nějaké definované pozice by trval zbytečně dlouho. Příkladem mohou být různé komunikační hry, kde bloudíš nějakým virtuálním světem a snažíš se dopátrat informací, které ti umožní splnit zadaný úkol.

578. No dobře. V čem by se situace změnila, kdyby se jednalo o hru a obecně o libovolný program, v němž by bylo výhodné si pamatovat každý krok (napadají mne např. textové editory).

V takovém případě máš dvě možnosti:

Kdy uchovávat výsledné pozice

- V některých aplikacích je výhodné si pamatovat opravdu jednotlivé požadavky (např. tahy ve hře), přičemž podle směru, kterým by ses průběhem seance pohyboval, bys buďto vyvolával požadavky (tahy) inverzní (zpět – undo) anebo naopak požadavky uložené (znovu – redo). Typickým příkladem takové aplikace by mohly být např. šachy.
- V jiných aplikacích je naopak výhodnější si místo provedených akcí pamatovat inkrementální změny stavu (např. u textových editorů si budeš pamatovat co přibylo, resp. ubylo).

Pamatování mezistavů může být drahé

Nesmíš ale zapomínat na to, že zejména u některých aplikací je pamatování si mezistavů poměrně drahou operací (většinou spotřebuje hodně paměti), takže není vhodné s takovýmito možnostmi příliš hýřit. Určitě víš, že řada programů ti proto nabízí, aby sis sám zvolil, kolik posledních kroků si mají pamatovat.

Příklad: Reversi s návraty

579. Řekl bych, že teď už zbývá pouze nějaký ukázkový příklad, na němž bys mi ukázal nějakou vzorovou implementaci.

Ukážu ti, jak je možno doplnit kód rozebíraný v podkapitole *Příklad: Reversi (Othello)* na straně 429 jako příklad aplikace návrhového vzoru *Model-Pohled-Ovládání*.

Tehdy jsem součásti, které souvisely s možností zapamatování si jednotlivých stavů hry, přeskakoval a odkazoval tě na tuto podkapitolu. Nyní se k nim vrátím a vše si ukážeme.

580. Sem s ním, už se nemohu dočkat.

Nepoužitelné uchovávání tahů

Reversi jsou typickým příkladem hry, u které nemusí být optimální pamatovat si jednotlivé tahy, protože ze zadaného tahu a dosažené cílové situace se nedá poznat, jaká byla výchozí situace – leda by sis pokaždé celou hru k poslednímu tahu přehrál. Mohli bychom proto považovat za rozumnější pamatovat si výsledné situace po jednotlivých tazích.

Potřebujeme stav desky

Situaci hry kompletně popisuje stav hrací desky, tj. uspořádání kamenů na desce. Nebylo by však moudré ukládat celou desku, protože bychom tak ukládali řadu informací, které nezbytně nepotřebujeme.

Nepotřebujeme ji ukládat celou

Proč bychom měli ukládat odkazy na kameny na jednotlivých polích, když nám stačí vědět, na kterých polích jsou kameny hráče a na kterých kameny počítače. Každá tato informace obnáší jediný bit, kdežto odkaz na nějaký objekt zabere podle typu procesoru čtyři nebo osm bajtů.

Uchováváme v instancích třídy `BitSet`

Ve třídě `Model` (rozšíření najdeš ve výpisu 34.1) jsem pro uchování stavu definoval vnořenou třídu – přepravku, kterou jsem nazval `Stav` a která měla dva atributy, jimiž byly instance třídy `java.util.BitSet`; jedna udržovala informace o umístění kamenů hráče a druhá o umístění kamenů počítače.

Rozhraní `IStav` a třída `Model.Stav`

Třída `Stav` implementovala rozhraní `IStav`, které podle doporučení nedeklarovalo žádné metody, a bylo tedy pouhým značkovacím rozhraním (jistě budeš souhlasit, že jeho zdrojový kód není třeba ukazovat). Nikdo z těch, kteří model o stav požádali, tak neměl šanci zjistit, jaké informace si model do instance stavu uložil. Žadatel dostal pouze objekt, který mohl někde uchovat, aby jej pak mohl modelu při žádosti o reaktivaci daného stavu předat.

Doplněné metody třídy `Model`

Ve třídě `Model` (a současně i v rozhraní `IModel`) pak kromě této vnořené přibyly dvě metody: metoda `getStav()` vracející instanci rozhraní `IStav` a metoda `setStav(IStav)` nastavující podle obdržené instance rozložení kamenů na hrací desce.

Výpis 34.1:

```
// ...  Vynechané části definice

    /*****************************************************************************
     * Vrátí aktuální stav hry jako instanci rozhraní {@link IStav}.
     */
    public IStav getStav()
    {
        return new Stav();
    }


    /*****************************************************************************
     * Nastaví stav hry zadaný prostřednictvím parametru.
     */
    public void setStav( IStav stav )
    {
        if( ! (stav instanceof Stav) )
            throw new IllegalArgumentException(
                "Parametr není instancí požadované třídy" );
        ((Stav)stav).setStav();
```

```
        najdiTah( Kámen.HRÁČ );
    }

// ...  Vynechané části definice


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    /****************************************************************************
     * Instance třídy <code>Stav</code> představují jednotlivé stavy hry.
     * Implementací rozhraní {@link IStav} se nezavazují k definici žádných
     * metod, pouze tím deklarují, že uchovávají stav.
     * Všechny definované metody proto slouží pro interní potřebu
     * instancí vnější třídy.
     */
    private class Stav implements IStav
    {
        /** Atributy udržující informaci o pozicích kamenů hráče a počítače. */
        BitSet hráč, počítač;


        /****************************************************************************
         * Vytvoří novou instanci a uloží do ní aktuální stav hry.
         */
        private Stav()
        {
            hráč    = new BitSet( polí );
            počítač = new BitSet( polí );
            int i = 0;
            for( int x=1;   x <= strana;   x++ ) {
                Kámen[] sloupec = deska[x];
                for( int y=1;   y <= strana;   y++, i++ ) {
                    Kámen kámen = sloupec[y];
                    if( kámen == kámen.HRÁČ )
                        hráč.set( i );
                    else if( kámen == Kámen.POČÍTAČ )
                        počítač.set( i );
                }
            }
        }


        /****************************************************************************
         * Zapíše zapamatovaný stav (tj. příslušně umístí kameny) na herní desku.
         */
        private void setStav()
        {
            int i = 0;
            for( int x=1;   x <= strana;   x++ )
                for( int y=1;   y <= strana;   y++, i++ )
                    deska[x][y] = (hráč   .get( i ))  ?  Kámen.HRÁČ
                                : (počítač.get( i ))  ?  Kámen.POČÍTAČ
                                                      :  Kámen.PRÁZDNÝ;
        }

    }
}
```

581. Když už jsi šetřil, tak proč jsi místo instancí třídy `BitSet` nepoužil rovnou čísla typu `long`?

Proč není použit `long`

Souhlasím, že tím bych ještě více ušetřil paměť, a dokonce i zrychlil program, ale takto mi to připadalo pro řadové programátory pochopitelnější, protože moje zkušenost říká, že průměrný programátor si s bitovými operacemi příliš netyká. Přiznám se ale, že když bych vytvářel program pouze pro svoji soukromou potřebu a ne pro výuku, tak bych po nich asi sáhl.

582. Co dalšího se zavedením možnosti návratu a znovuprovedení tahů změnilo?

Nutná rozšíření třídy `Řidič`

Vedle třídy `Model` bylo třeba upravit už jen definici třídy `Řidič` (přehled jejích rozšíření najdeš ve výpisu 34.2), která musela připravit nějaké úložiště pro zapamatované stavy a rozšířit své schopnosti o reakci na klávesy žádající o návrat minulého stavu, resp. o vrácení se do stavu, z nějž jsme se před chvílí vraceli.

Tah blokuje návrat vpřed

Ve výpisu si všimni, že vynechám-li nemožnost návratu před počáteční stav, tak se do minulých stavů můžeš vracet bez omezení. Zpátky vpřed se však můžeš vrátit pouze tehdy, pokud jsi od návratu do minulého stavu neprovedl žádný tah. Provedením tahu se možnost jakékoliv akce typu *„odvolávám, co jsem odvolal“* ruší.

Výpis 34.2: Části definice třídy `Řidič`, které rozšiřují možnosti instancí o schopnost návratu minulých stavů a opětné vyvolání stavů, z nichž jsme se vraceli

```
package rup.česky.vzory._32_mvc.reversi;

// ...  Vynechané importy

public class Řidič extends KeyAdapter
{
// ...  Vynechané deklarace

    private final List<IStav> stavy = new ArrayList<IStav>();


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private int index;      //Index posledního zaznamenaného stavu
    private int poslední;   //Index posledního platného stavu


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří instanci, která bude řídit hru.
     */
    public Řidič( IModel model, IPohled pohled, IKurzor kurzor )
    {
        this.model  = model;
        this.pohled = pohled;
        this.kurzor = kurzor;
```

```
        this.stavy.add( model.getStav() );  //Zapamatuj si výchozí stav
        index = poslední = 0;

        SP.přihlašKlávesnici( this );
    }

//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Ošetření události stisk klávesy.
     * Jedná-li se o opakovaný stisk téže klávesy před jejím puštěním,
     * ignoruje jej. Jinak spustí akci příslušnou dané klávese.
     * Stisk klávesy Esc program ukončí.
     *
     * @param  ke   Událost klávesnice, na kterou je třeba reagovat
     */
    public synchronized void keyPressed( java.awt.event.KeyEvent ke )
    {
        int kc = ke.getKeyCode();
        switch( kc )
        {
// ...  Vynechaný kód pro posuny nahoru, dolů, vlevo a vpravo

            case VK_ENTER:    //Provedení tahu
                if( !model.setKámen( kurzor.getPozice() ) )
                    konecHry();
                IStav stav = model.getStav();
                poslední = ++index;
                if( stavy.size() > index )
                    stavy.set( index, stav );
                else
                    stavy.add( stav );
                break;

            case VK_BACK_SPACE:   //Návrat do minulého stavu
                if( index > 0 ) {
                    model.setStav( stavy.get( --index ) );
                }
                break;

            case VK_INSERT:       //Návrat do stavu, z nějž jsme se vraceli
                if( index < poslední ) {
                    model.setStav( stavy.get( ++index ) );
                }
                break;

            default:
        }
        pohled.zobraz();
    }


}
```

Shrnutí – co jsme se naučili

- Vzor ukazuje, jak je možno uchovávat informace o stavech objektů pro potřeby pozdějšího nastavení dřívějších stavů.
- Hlavním problémem je nutnost maximálního skrytí informací o instanci, jejíž stav chceme uložit.
- Instance uchovávající stav mohou implementovat pouhé značkovací rozhraní.
- Vlastní pamětníci bývají většinou definováni jako vnitřní třídy.
- Je třeba si rozmyslet, kdy uchovávat raději informace o provedených akcích a kdy výsledné pozice.
- Místo celých pozic je možno uchovávat jen jejich inkrementální změny.
- Návrhový vzor *Pamětník* patří mezi vzory uvedené v GoF.

KAPITOLA 35

Tak si to naprogramuj sám (Interpret – Interpreter)

- **Účel**
- **Implementace**
- **Příklad: Aritmetické výrazy**
- **Shrnutí – co jsme se naučili**

Stručná charakteristika vzoru[1]

Definuje reprezentaci gramatických pravidel jazyka pomocí systému tříd a současně definuje interpret tohoto jazyka pomocí metod instancí těchto tříd.

[1] **Definice v GoF:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. – Pro daný jazyk definuje reprezentaci jeho gramatiky spolu s interpretem, který používá tuto reprezentaci k interpretaci vět v daném jazyku.

Účel

583. Jako obyčejně se musím přiznat, že jsem za stručné charakteristiky vůbec nic nepochopil.

Tím se netrap. Stručná charakteristika je určena pro člověka, který již daný návrhový vzor zná, aby si daný vzor a jeho zákonitosti lépe vybavil.

Kdy vzor použijeme

Jde v podstatě o to, že v řadě případů bývá výhodné řešit některé problémy tak, že si navrhneš vlastní specializovaný programovací jazyk a v něm pak řešení problému popíšeš. Tvůj program pak bude tento popis interpretovat a tím daný problém vyřeší.

Návrhový vzor *Interpret* nabízí poměrně jednoduchý způsob, jak implementovat interpretaci programů jednoduchých jazyků. Každou konstrukci jazyka definovat jako samostatnou třídu a výskyty této konstrukce v programu pak definovat jako instance dané třídy. Celý program se tak přirozeně rozbalí do stromové struktury, kterou lze poměrně velmi jednoduše interpretovat.

584. Zní to sice hezky, ale dokud nějaký takový jazyk neuvidím, nebudu pořádně vědět, o čem hovoříš.

Vzor hovoří o vnitřní reprezentaci jazyka

Než si ukážeme implementaci interpretu na nějakém jazyku, musím tě upozornit, že návrhový vzor *Interpret* nehovoří o klasických programovacích jazycích, které znáš. Jde o vnitřní reprezentaci daného jazyka a vůbec se přitom nemluví o nějakém textovém zápisu.

Klasický textový program nejprve zpracovává lexikální analyzátor

Potřebuješ-li interpretovat nějaký program zapsaný v některém z běžných jazyků, musíš zdrojový text předhodit nejprve *lexikálnímu analyzátoru* (používá se pro něj občas termín *scanner*), který text projde a najde v něm jednotlivé symboly jazyka[1] – identifikátory různých entit (názvy proměnných, funkcí, tříd atd.), klíčová slova a další důležité prvky (závorky, operátory atd.).

Po něm nastoupí syntaktický analyzátor

Posloupnost těchto symbolů je pak předána *syntaktickému analyzátoru* (používá se pro něj občas termín *parser*), který v ní odhaluje větší celky (např. celý podmíněný příkaz, definici cyklu, definici metody, definici třídy atd.) a vytvoří vnitřní reprezentaci programu – většinou jako nějakou stromovou strukturu.

Další v řadě je sémantický analyzátor

Ve třetí etapě se dostává ke slovu *sémantický analyzátor*, který prochází onu vnitřní reprezentaci a vytváří mezikód pro vlastní interpret. U jednodušších implementací už nic nevytváří a zadaný program i interpretuje.

Vzor Interpret doporučuje vnitřní reprezentaci a způsob její interpretace

Návrhový vzor *Interpret* neříká vůbec nic o lexikální ani syntaktické analýze jazyka. On dokonce ani nepočítá explicitně s tím, že by byly dané programy zapisovány v textové podobě (tím ale neříkám, že to nejde – na konci si to ukážeme). Soustředí se především na to, jakou lze zvolit vnitřní reprezentaci navrženého jazyka a jak lze program převedený do této vnitřní reprezentace jednoduše interpretovat.

Na složitější jazyky je lepší použít klasické postupy

Pro nějaké rafinovanější jazyky je koncepce návrhového vzoru *Interpret* zbytečně těžkopádná – pro ně je potřeba použít některé z komplexnějších řešení. Pro jednodu-

[1] V literatuře bývají tyto symboly někdy označovány jako *lexémy* nebo *tokeny*.

ché pomocné jazyky, s nimiž se v programech často setkáváme, však návrhový vzor *Interpret* nabízí velmi jednoduché a poměrně efektivní řešení této úlohy.

Implementace

585. Začínám tušit, o čem ta stručná charakteristika vlastně mluví. Teď by to ale chtělo všechno ukázat na nějakém příkladu.

Definice gramatiky analyzátoru regulárních výrazů

Pokusím se ti nejprve vysvětlit principy celého návrhového vzoru na příkladu inspirovaném příkladem v GoF. Představ si, že máme následovně definovanou gramatiku tvorby regulárních výrazů (teď nehovořím o regulárních výrazech ze standardní knihovny, ale o nějakých našich vlastních):

```
Výraz           ::= Literál | Volba | Posloupnost | Opakování |
                    '(' Výraz ')'
Volba           ::= Výraz '|' Výraz
Posloupnost     ::= Výraz '&' Výraz
Opakování       ::= Výraz '*'
Literál         ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

586. Gramatiky jsem nikdy nestudoval, takže mi budeš muset předchozí definici vysvětlit.

Jak číst definici gramatiky

Identifikátory na levé straně přiřazovacího operátoru `::=` označují různé prvky jazyka. Na pravé straně je pak definice příslušného prvku.

Znak `|` („svislítko“) odděluje jednotlivé možnosti, složené závorky sdružují svůj obsah tak, že vůči okolí vystupuje jako jeden objekt, a hvězdička označuje, že se předchozí prvek může libovolněkrát opakovat, přičemž libovolněkrát znamená i nulakrát. Má-li se do programu napsat přímo nějaký znak, je v definici uveden mezi apostrofy.

587. No, možná, že bych to již dokázal rozebrat, ale přesto bych přivítal, kdybys mi raději ty definice řádek za řádkem přeložil do srozumitelštiny.

Rozbor popsané gramatiky

Co bych pro tebe neudělal. Projděme si tedy předchozí definici gramatiky příkaz za příkazem.

– Výraz

První příkaz říká, že *Výraz* je buď *Literál* nebo *Volba* nebo *Posloupnost* nebo *Opakování* anebo *Výraz* uzavřený v kulatých závorkách. Každý z vyjmenovaných prvků je pak dále definován.

– Volba

Volba se zapíše jako dva výrazy oddělené znakem `|` a znamená, že porovnávaný text musí odpovídat buď výrazu vlevo od „svislítka“ nebo výrazu vpravo.

– Posloupnost

Posloupnost zapisuješ jako dva výrazy oddělené znakem & a vyžaduje, aby v porovnávaném textu byl text odpovídající výrazu vlevo následován textem odpovídajícím výrazu vpravo.

– Opakování

Opakování zapíšeš tak, že za výraz, který se může opakovat, napíšeš hvězdičku. V porovnávaném textu se pak může libovolněkrát opakovat text odpovídající výrazu před hvězdičkou, přičemž i zde budeme definovat, že libovolněkrát může být i nulakrát.

– Literál

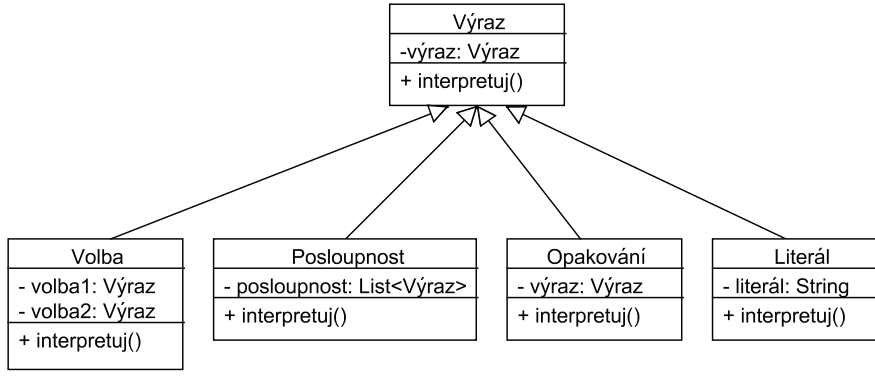
Literál je pak libovolná posloupnost malých písmen anglické abecedy.

588. Dejme tomu, že jsem tvůj výklad jednotlivých prvků regulárního výrazu pochopil. Co teď?

Převedení gramatiky do systému tříd

Začneme tím, že si tuto definici převedeme do tříd. Každému prvku předchozí syntaktické definice bude odpovídat jedna třída. Návrhový vzor *Interpret* reprezentuje gramatická pravidla vazbami mezi třídami. Prvky jazyka jsou definovány jako třídy. Abychom mohli definovat metody realizující požadované chování prvků, definujeme instanční atributy odpovídající prvkům na pravé straně definice, s nimiž budou tyto metody pracovat.

Předchozí gramatika tedy vede na definici pěti tříd: třídy `Výraz` a jejích potomků `Volba`, `Posloupnost`, `Opakování` a `Literál`. Jejím výchozím prvkem je `Výraz`, koncovým symbolem je `Literál`. Diagram tříd této reprezentace je znázorněn na obr. 35.1.



Obrázek 35.1
Diagram tříd interpretu popsané gramatiky

589. Proč nemá svoji vlastní třídu také výraz v závorce?

Proč nemá výraz v závorce svoji třídu

Závorka pouze označuje, jakou část výrazu je třeba chápat jako jeden prvek, protože je rozdíl, jestli napíšeš `a | (b & c)` nebo `(a | b) & c`. Když už víš, co s čím patří dohromady, tak můžeš příslušný výraz vytvořit a na závorku zapomenout.

Se závorkou bychom museli počítat pouze tehdy, pokud bychom chtěli vytvořit i překladač našich výrazů do vnitřní podoby určené pro náš interpret. Protože ale budeme prozatím překládat výrazy ručně (tj. na překladač si budeme hrát my), tak se o ní v programu zmiňovat nemusíme, protože při překladu automaticky zmizí.

590. Tak mi tedy ukaž nějaký takový ruční překlad.

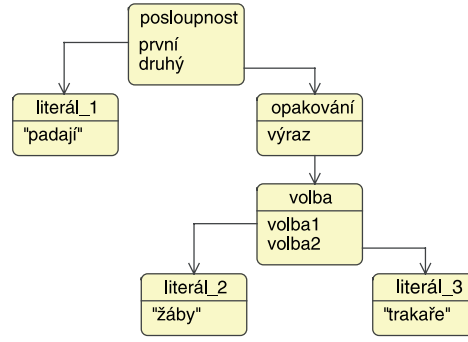
Překládaný výraz

Každý výraz odpovídající předchozí syntaxi je reprezentovatelný jako skupina navzájem propojených objektů. Představ si, že dostaneš za úkol, aby tvůj program kontroloval, zda nějaké texty odpovídají výrazu:

```
padají  &  ( žáby | trakaře ) *
```

„Ruční" překlad výrazu

Tomuto výrazu by odpovídal objekt, jehož strukturu si můžeš prohlédnout na obrázku 35.2. Tento objekt bude přestavovat interpret, který je schopen ověřit, že zadaný text odpovídá výrazu reprezentovanému daným interpretem.



Obrázek 35.2
Diagram objektů odpovídající výrazu „padají (žáby | trakaře)"

591. Vždyť to není žádný objekt, ale celý strom objektů.

Výraz je převeden na strom objektů

Samozřejmě. Náš interpret, tj. objekt `výraz`, se se svým úkolem vypořádá tak, že analyzovaný text předá svým „podřízeným", kteří vyhodnotí jednotlivé podvýrazy a oznámí mu výsledek, pak se s ním pochlubí.

Abych byl přesnější:

1. Ty předáš analyzovaný text jako parametr nějaké metody objektu `výraz`.
2. Ten předá text objektu `posloupnost`.
3. Ten se zeptá objektu `literál_1`, zda začátek textu odpovídá požadovanému, tj. zda text začíná slovem „padají".
4. Bude-li tato domněnka potvrzena, zeptá se ještě objektu `opakování`, zda je zbytek textu požadovaným opakováním.
5. Objekt `opakování` se bude opakovaně ptát objektu `volba`, následuje-li v textu některý z volitelných výrazů. Až objekt `volba` (přesněji jeho podřízené objekty `literál_2` a `literál_3`) další výraz nenajde, oznámí volajícímu objektu, tj. objektu `posloupnost`, že s opakováním skončil.
6. Po tomto zkontrolování jednotlivých částí vrátí objekt `posloupnost` výsledek vyhodnocení objektu `výraz`.
7. Objekt `výraz` se v případě úspěšnosti testu ještě podívá, jestli byl vyhodnocen opravdu celý výraz, a pak požadovanou informaci vrátí.

592. Princip je mi asi jasný, ale nejsem si jist, že bych to dokázal také zapsat v kódu. Začni třeba s tím, jak provedu překlad a vyrobím tu vnitřní reprezentaci, o které jsi mluvil.

Jak vytvořit objekt odpovídající výrazu

Začneme tedy tím, že si ukážeme, jak bychom vytvořili objekt odpovídající výrazu `padají & (žáby | trakaře)*`. Ukážu ti dvě možnosti – nejprve tu „ukecanější":

```
Literál     literál_1   = new Literál( "padají" );
Literál     literál_2   = new Literál( "žáby"   );
Literál     literál_3   = new Literál( "trakaře");
Volba       volba       = new Volba( literál_2, literál_3 );
```

```
Opakování   opakování   = new Opakování( volba );
Posloupnost posloupnost = new Posloupnost( literál_1, opakování );
Interpret   interpret   = new Výraz( posloupnost );
```

Zkrácený zápis

V předchozí ukázce jsme „cestou" definovali řadu pomocných proměnných, které byly proměnnými na jedno použití. Zkušenější programátoři jsou často líní psát tolik deklarací a tak se snaží bez pomocných proměnných obejít:

```
Interpret interpret = new Interpret(
          new Posloupnost(
              new Literál( "padají" ),
              new Opakování(
                  new Volba(
                      new Literál( "žáby"   ),
                      new Literál( "trakaře") ) ) ) );
```

Řadě lidí ale připadá takovýto zápis nepřehledný, a jak jsme si říkali již několikrát, přehlednost a srozumitelnost programu jsou důležitější než pár ušetřených mikrosekund strojového času nebo znaků v programu.

Pokud ti však porozumění takovémuto zápisu nedělá potíže a nebude-li dělat potíže ani tvým spolupracovníkům, nic ti nebrání jej používat. Za sebe totiž musím říct, že mně připadá ten druhý výraz rozhodně přehlednější.

593. Že by to byla křišťálová studánka, to se říct nedá. Při troše snahy se v tom ale dá vyznat. Mám tedy interpret konkrétního výrazu. Co s ním?

Použití vytvořeného interpretu

Co sníš, odhadnout nedokážu, ale mohu ti ukázat, jak můžeš použít právě vytvořený interpret. K otestování vytvořeného interpretu bychom mohli použít následující kód:

```
String [] texty = {
          //Výrazy, které mají vyhovovat
              "padají žáby žáby trakaře",
              "padají",
              "padají trakaře trakaře trakaře",
          //Výrazy, které nemají vyhovovat
              "padají padají žáby" ,
              "padají žáby padají" ,
              ""
          };
for( String s : texty ) {
    System.out.println( s + "  –  "+ interpret.analyzuj(s) );
}
```

Definice jednotlivých částí interpretu

594. Jak se interpret připraví a použije, to již tuším. Teď by mě ale zajímalo, jak jej vlastně naprogramuji. Zatím jsi se stále choval tak, jako by už byly všechny třídy dávno hotové.

Potřeboval jsem ti nejprve ukázat, jak to všechno funguje, abychom se pak mohli zamýšlet nad tím, jak to zařídit, aby všechno právě takto fungovalo. Teď se tedy vrhneme na definice jednotlivých tříd našeho interpretu.

Rozhraní `IVýraz`

Nejprve ti ukážu, jak jsem definoval rozhraní `IVýraz`. Všimni si, že je v něm zanořená třída, která slouží jako obálka na analyzovaný text doplněný o pozici, kam se analyzátor doposud dostal. Třída je definovaná jako soukromá v rámci balíčku, takže na

ni interpret vidí, a třídy představující jednotlivé speciální podoby výrazu ji dokonce při přihlášení se k implementaci daného rozhraní zdědí. V instanci této přepravky si pak budou předávat informaci o tom, jak jsou s analýzou daleko.

Výpis 35.1: Rozhraní `IVýraz` implementované třídami, jejichž instance představují vnitřní reprezentaci části regulárního výrazu

```
package rup.česky.vzory._35_interpet.regulární;


/*******************************************************************************
 * Rozhraní <code>IRegVýraz</code> definuje požadavky na instance reprezentující
 * jednotlivé speciální druhy regulárního výrazu.
 * Kromě toho definuje jako svoji vnořenou třídu přepravku,
 * která slouží instancím tříd, jež toto rozhraní implementují,
 * k vzájemné výměně informací.
 */
public interface IRegVýraz
{
//== METODY ====================================================================

    /*******************************************************************************
     * Vyhodnotí, zda text v parametru <code>zdroj</code> počínaje pozicí,
     * na niž ukazuje index, odpovídá příslušnému regulárnímu výrazu.
     * Odpovídá-li, přesune referenční bod za tento text.
     *
     * @param zdroj Přepravka s analyzovaným textem a indexem počátečního znaku
     * @return <code>true</code> pokud následující text odpovídá požadavkům
     */
    public boolean vyhodnoť( Zdroj zdroj );


//== VNOŘENÉ TŘÍDY =============================================================

    /*******************************************************************************
     * Přepravka určená pro třídy implementující rozhraní <code>IRegVýraz</code>
     * k předávání informací o postupu při vyhodnocování zadaného výrazu.
     * <p>
     * Třída není neměnná, protože atribut <code>pozice</code> nemůže být
     * konstantní. Bezpečnosti je dosaženou omezením přístupových práv
     * na "package private".
     */
    static class Zdroj
    {
        /** Analyzovaný text. */
        final String text;

        /** Pozice, k níž je text prověřen. */
        int pozice = 0;

        /*******************************************************************************
         * Vytvoří instanci přepravky s analyzovaným textem
         * a indexem nastaveným na počátek tohoto textu.
         *
         * @param text Text určený k analýze
         */
        Zdroj( String text ) {
```

```
            this.text = text;
        }
    }
}
```

595. Jestli jsem to dobře pochopil, tak instance tříd z obr. 35.1 budou umět vyhodnotit svoji část výrazu a jako štafetový kolík si budou předávat instanci přepravky.

Třída `Interpret`

Dalo by se to tak říci. Ty vytvoříš instanci třídy `Interpret` reprezentující daný regulární výraz a pak už jí jen předhazuješ výrazy ke kontrole. Definici třídy interpret a její metody `analyzuj(String)` si můžeš prohlédnout ve výpisu 35.2. Všimni si, že metoda vrátí `true` až tehdy, když se přesvědčí, že byl zanalyzovaný opravdu celý zadaný text.

Součástí výpisu je i závěrečná testovací metoda, která na příkladu svých testů ukazuje, jak je možno tento interpret použít. Její tělo jsme si ale již ukazovali, takže tě nemá čím překvapit.

Výpis 35.2: Definice třídy `Interpret`, která zastřešuje celou funkčnost

```
package rup.česky.vzory._35_interpet.regulární;


/*****************************************************************************
 * Instance třídy <code>Interpret</code> představují přeložené regulární výrazy,
 * které jsou schopny zanalyzovat, zda zadaný text
 * vyhovuje příslušnému regulárnímu výrazu.
 */
public class Interpret
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

        /** Vnitřní reprezentace regulárního výrazu. */
        private final IRegVýraz výraz;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /***************************************************************************
     * Vytvoří instanci reprezentující zadaný regulární výraz.
     *
     * @param výraz  Regulární výraz, který chceme reprezentovat
     */
    public Interpret( IRegVýraz výraz ) {
        this.výraz = výraz;
    }


//== ABSTRAKTNÍ METODY =========================================================
//== NESOUKROMÉ  METODY INSTANCÍ ===============================================

    /***************************************************************************
     * Zanalyzuje zadaný text a vrátí informace o tom,
     * vyhovuje-li regulárnímu výrazu reprezentovanému daným interpretem.
     *
     * @param text  Analyzovaný text
     * @return <code>true</code> vyhovuje-li zadaný text danému výrazu
```

```
     */
    public boolean analyzuj( String text) {
        IRegVýraz.Zdroj zdroj = new IRegVýraz.Zdroj( text );
        boolean ret = výraz.vyhodnoť( zdroj );  //Výraz je vyhodnocen
        if( !ret )              //Neodpovídá
            return false;
        //Odpovídá li, je třeba zkontrolovat, zda byl vyhodnocen celý řetězec
        int i = zdroj.pozice - 1;
        while( (++i < text.length())  &&
               (Character.isWhitespace( text.charAt(i) )) );
        return i == text.length();
    }


//== TESTY ================================================================

    /*************************************************************************
     * Vytvoří interpret výrazu "padají ( žáby | trakaře )"
     * a vyhodnotí několik textů, nakolik výrazu vyhovují
     */
    public static void test() {
        Interpret interpret = new Interpret(
          new Posloupnost(
              new Literál( "padají" ),
              new Opakování(
                  new Volba(
                      new Literál( "žáby"   ),
                      new Literál( "trakaře") ) ) ) );
        String [] texty = {
            //Výrazy, které mají vyhovovat
                "padají žáby žáby trakaře",
                "padají",
                "padají trakaře trakaře trakaře",
            //Výrazy, které nemají vyhovovat
                "padají padají žáby" ,
                "padají žáby padají" ,
                "",
            //Test reakce na oddělovače
                "padajížábyžábytrakaře",
                "padají",
                "      padají  trakaře  \n     trakaře   trakaře     ",
        };
        for( String s : texty ) {
            System.out.println( s + "  –  "+ interpret.analyzuj(s) );
        }
    }
    /** @param args Parametry příkazového řádku */
    public static void main( String... args ) { test(); }
}
```

596. Definice třídy `Interpret` byla z tvého předchozího popisu tak trochu odhadnutelná. Zajímala by mne ale definice dalších tříd.

Třídy Posloupnost a Volba

Ukážu ti definici třídy `Posloupnost`. Velice podobná je i třída `Volba` a právě kvůli té podobě ji tu ani neukazuji – kdyby ses o ni zajímal, najdeš ji mezi zdrojovými soubory.

Výpis 35.3: Definice třídy `Posloupnost`, jejíž instance zjišťují, zda následující text obsahuje dva zadané regulární výrazy za sebou

```
package rup.česky.vzory._35_interpet.regulárni;


/*******************************************************************************
 * Instance třídy <code>Posloupnost</code> dokážou vyhodnotit,
 * zda v předloženém textu následuje pasáž odpovídající dvěma za sebou
 * následujícím regulárním výrazům zadaným jejímu konstruktoru.
 */
public class Posloupnost implements IRegVýraz
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Regulární výrazy, s nimiž se text porovnává. */
    private final IRegVýraz první, druhý;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*******************************************************************************
     * Vytvoří instanci kontrolující, zda další text odpovídá posloupnosti
     * dvou zadaných regulárních výrazů.
     *
     * @param první Regulární výraz, jemuž má odpovídat úvodní část textu
     * @param druhý Regulární výraz, jemuž má odpovídat následující část textu
     */
    public Posloupnost( IRegVýraz první, IRegVýraz druhý ) {
        this.první = první;
        this.druhý = druhý;
    }


//== NESOUKROMÉ  METODY INSTANCÍ ===============================================

    /*******************************************************************************
     * Vyhodnotí, zda následující text v parametru <code>zdroj</code>
     * odpovídá posloupnosti zadaných regulárních výrazů.
     * Odpovídá-li, přesune referenční bod za příslušný text.
     *
     * @param zdroj Přepravka s analyzovaným textem a indexem počátečního znaku
     * @return <code>true</code> pokud následující text odpovídá požadavkům
     */
    public boolean vyhodnoť( Zdroj zdroj ) {
        return první.vyhodnoť(zdroj)  &&  druhý.vyhodnoť(zdroj);
    }
}
```

597. Ty jsi sice naznačoval, že to bude jednoduché, ale že to bude takhle jednoduché, to jsem opravdu nečekal. Ty ostatní třídy jsou také takhle jednoduché? Co třeba cyklus?

Jak je zjednodušen cyklus

No, přiznejme si, že právě cyklus jsem v tomto úvodním příkladu v zájmu jednoduchosti trochu odbyl. Instance třídy, jejíž definici si můžeš prohlédnout ve výpisu 35.4, patří mezi ty „žravé“, které „sežerou“ maximální část předloženého textu. To ale

někdy nemusí být správně, protože pak může být jako nevyhovující vyhodnocen i výraz, který ve skutečnosti zadanému regulárnímu výrazu vyhovuje.

598. Tak teď jsi mne zaskočil. Jak ji může vyhodnotit jako nevyhovující, když vyhovuje.

Ukázka špatně vyhodnocených výrazů

Záleží na definici regulárního výrazu. Definuješ-li např. regulární výraz:

```
a * & abc
```

nebude mu text `aaabc` vyhovovat. Pokud bys proto upravil definici do tvaru:

```
a * & bc
```

tak by jí sice předchozí text vyhověl, avšak této definici by pro změnu vyhověl i text `bc`, který podle první definice vyhovět nemá.

599. Jak bych to měl tedy řešit?

Možné řešení

Např. tak, že by metody nevracely jediné řešení (tj. jednu pozici), ale vracely by celou kolekci možných řešení, tj. kolekci indexů, kde končí řetězce, které vyhovují doposud zpracované části řetězce. Metody vyhodnocující zbytek řetězce by pak analyzovaly první člen kolekce, a pokud by vedl k odmítnutí výrazu, tak by zkusily další člen a výraz by vyhodnotily jako nevyhovující až tehdy, pokud by žádný z členů nevedl k řešení.

Nechtěl jsem ale do programu zanášet informační šum, protože jsem si říkal, že by ty kolekce odváděly pozornost od vlastního principu řešení. Můžeš si to zkusit vyřešit jako domácí cvičení.

600. Přiznávám, že takovéto zesložitění programu si rád odpustím. Navíc bych k němu neměl tu správnou motivaci, protože třídy `Pattern` a `Matcher` z balíčku `java.util` v pohodě zvládají i mnohem složitější výrazy, tak se přeci nebudu vnucovat se svým primitivním interpretem, který navíc vyžaduje převést regulární výraz do objektové podoby. Když už jsme ale na třídy `Pattern` a `Matcher` narazili, tak by mne zajímalo, jestli jsou tyto třídy navržené podle vzoru *Interpret?*

Jsou, i když je to tam trošičku složitější než v našem předchozím příkladu. Třída `Pattern` obsahuje překladač, který převede zadaný regulární výraz do objektové vnitřní podoby – instance třídy `Pattern`.

Instance třídy `Matcher` jsou pak analyzátory aplikující zadaný regulární výraz (tj. instanci třídy `Pattern`) na konkrétní text. Tyto instance jsou tedy interprety programu uchovávaného v instanci třídy `Pattern`.

601. Aha – tak abychom se vrátili k našemu ukázkovému programu.

Třída `Opakování`

Také si myslím. Ukážu ti tedy pouze onu jednoduchoučkou, i když ne zcela dokonalou verzi a budu přepokládat, že si ono rozšíření o návrat celé kolekce hodnot dokážeš domyslet.

Výpis 35.4: Definice třídy `Opakování`, jejíž instance zjišťují, zda se v analyzovaném textu opakuje zadaný výraz

```
package rup.česky.vzory._35_interpet.regulární;


/*******************************************************************************
```

```
 * Instance třídy <code>Opakování</code> vyhledá v analyzovaném textu případná
 * opakování zadaného regulárního výrazu a přesune další vyhodnocování
 * až za ně. Protože se zadaný regulární výraz může v analyzovaném textu
 * vyskytnout i nulakrát, vrací její vyhodnocovací metoda vždy <code>true</code>.
 */
public class Opakování implements IRegVýraz
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** Výraz, jehož výskyty máme přeskakovat. */
    private final IRegVýraz výraz;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří instanci přeskakující ve vyhodnocovaném textu případné
     * (i opakované) výskyty zadaného regulárního výrazu.
     *
     * @param výraz Výraz, jehož následující výskyty přeskakujeme
     */
    public Opakování( IRegVýraz výraz)
    {
        this.výraz = výraz;
    }


//== NESOUKROMÉ  METODY INSTANCÍ ===============================================

    /*****************************************************************************
     * Přesune referenční bod ve vyhodnocovaném textu až za případná opakování
     * zadaného regulárnímu výrazu.
     *
     * @param zdroj Přepravka s analyzovaným textem a indexem počátečního znaku
     * @return Vždy <code>true</code>
     */
    public boolean vyhodnoť( Zdroj zdroj ) {
        while( výraz.vyhodnoť( zdroj ) );
        return true;
    }
}
```

602. Zatím všechny objekty řešily svůj analytický úkol tak, že jej přehodily na někoho jiného. Protože jediná třída, kterou jsi mi ještě neukázal, je `Literál`, tak si říkám, že to bude jediný objekt, který bude opravdu pracovat.

Třída Literál

Máš pravdu. Instance třídy `Literál` jsou v našem interpretu jediné, které zpracovávaný řetězec opravdu čtou. Jejich úkolem při analýze je pouze zjistit, jestli následující řetězec je či není tím řetězcem, na který čekají.

Objeví-li literál na vstupu svůj řetězec, přesune aktivní pozici za něj. Zjistí-li, že na aktuální pozici je jiný řetězec, nechá aktuální pozici nezměněnou, aby tam svého oblíbence mohl najít někdo jiný.

Výpis 35.5: Zdrojový kód třídy `Literál`

```
package rup.česky.vzory._35_interpet.regulární;


/*******************************************************************************
 * Instance třídy <code>Literál</code> testují, jestli vstupní text
 * nepokračuje jejich řetězcem.
 */
public class Literál implements IRegVýraz
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final String jméno;


//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private int délka;


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*******************************************************************************
     * Vytvoří objekt testující přítomnost zadaného řetězce
     * na aktuální pozici analyzovaného textu.
     *
     * @param jméno Porovnávaný (vyhledávaný) řetězec
     */
    public Literál( String jméno )
    {
        this.jméno = jméno;
        this.délka = jméno.length();
    }


//== NESOUKROMÉ  METODY INSTANCÍ ===============================================

    /*******************************************************************************
     * Vyhodnotí, zda následující text za aktuální pozicí
     * v parametru <code>zdroj</code> není zadaným literálem.
     * Je-li, přesune aktuální pozici za něj.
     *
     * @param zdroj Přepravka s analyzovaným textem a indexem počátečního znaku
     * @return <code>true</code> pokud následující text odpovídá požadavkům
     */
    public boolean vyhodnoť( Zdroj zdroj ) {
        int start=zdroj.pozice-1;
        String text = zdroj.text;
        while( (++start < text.length())  &&
               (Character.isWhitespace( text.charAt(start) )) );
        if( start < text.length() ) {
            int i=start, j=0;   //Potřebujeme se ptát na j po proběhnutí cyklu
            for( ;
                 (j < délka)  &&  (zdroj.text.charAt(i) == jméno.charAt(j));
                 i++, j++ );
```

```
            if( j == délka ) {
                zdroj.pozice = start + j;
                return true;
            }
        }
        zdroj.pozice = start;
        return false;
    }
}
```

Bílé znaky se ignorují

Jak si můžeš v programu všimnout, před vlastním čtením znaků, které by mohly být součástí onoho očekávaného řetězce, se nejprve přeskočí všechny případné bílé znaky. V zájmu jednoduchosti jsem ale nezaváděl povinné oddělovače, takže se můžeš přesvědčit, že řetězce bez mezer budou stejně jako řetězce s řadou bílých znaků mezi literály označeny jako vyhovující.

Příklad: Aritmetické výrazy

603. Jsem rád, že jsi první příklad interpretu hodně zjednodušil. Možná by ale přece jenom neškodilo ukázat něco složitějšího.

O moc složitější být nemůže, protože nemohu věnovat celou knihu jenom interpretu. I tak si myslím, že to bude největší příklad v celé knize. Je ale už poslední, tak jej můžeme vzít současně jako rozlučkový.

Mohli bychom si zkusit definovat takovou jednoduchou „kalkulačku“, která umí pracovat s konstantami a proměnnými, dokáže sčítat a násobit, přičemž je schopna dát přednost násobení před sčítáním.

Oproti minulé verzi, kdy jsme vnitřní reprezentaci programu vyráběli sami, si ale tentokrát vytvoříme program, který bude číst textovou podobu výrazů a tyto výrazy bude převádět do požadované vnitřní stromové podoby. Abych ale tento analyzátor maximálně zjednodušil, bude náš interpret umožňovat používání pouze jednopísmenových názvů konstant a proměnných.

604. I tak to vypadá zajímavě a mohl by to být takový dobrý příklad na rozlouče-nou. Tak jakpak jsi definoval povolené tvary výrazů?

Definice gramatiky, kterou bude vytvářený interpret schopen zpracovávat, je následující:

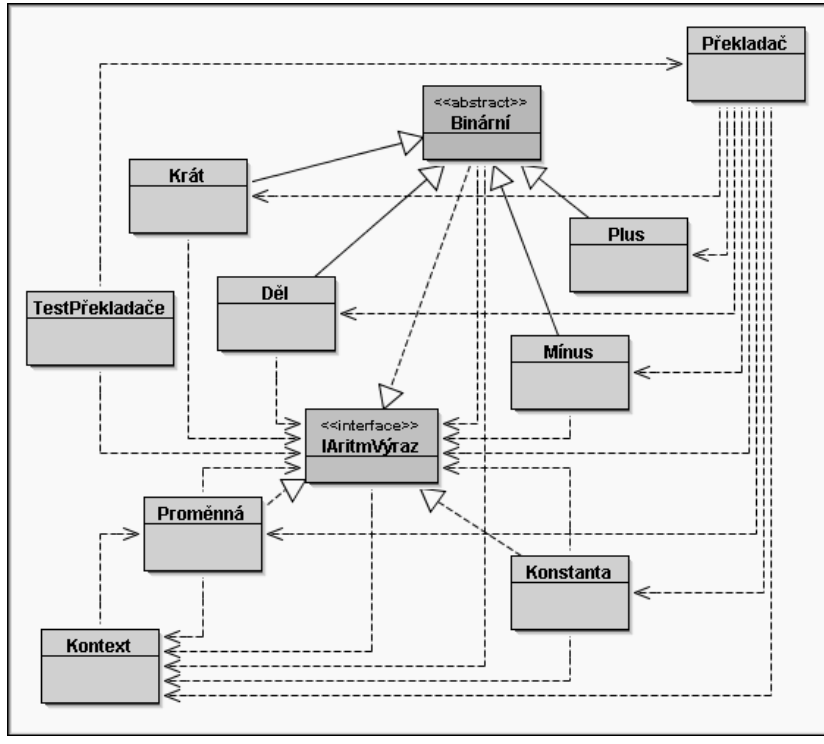
```
Konstanta ::= '#' Číslo '#'
Proměnná  ::= Identifikátor
Poločlen  ::= Proměnná  |  Konstanta          |  '(' Výraz ')'
Člen      ::= Poločlen  |  '+' Poločlen       |  '-' Poločlen
Součin    ::= Člen      |  Součin '*' Člen    |  Součin '/' Člen
Součet    ::= Součin    |  Součet '+' Součin  |  Součet '-' Součin
Výraz     ::= Součet
```

605. Vypadá pouze o maličko složitější než ta, kterou jsme doposud rozebírali.

Zdání klame. Tato definice zavádí proměnné, které je možno substituovat jakýmkoliv výrazem (je za ně možno dosadit výraz). Po této substituci se bude v místech, v nichž je použita daná proměnná, vyhodnocovat vždy celý substituovaný výraz.

606. Ale ale, to vypadá zajímavě. Tak se chlub!

Začal bych diagramem tříd celé aplikace, který si můžeš prohlédnout na obr. 35.3.



Obrázek 35.3
Diagram tříd interpretu aritmetických výrazů

607. Chybí mi v něm třída pro Poločlen, který gramatika definuje, a rozcházejí se mi i další názvy.

Říkal jsem si, že i takto je už diagram tříd poměrně složitý, a že by proto bylo rozumné nahradit aplikaci znaménka minus obyčejným součinem. Nebude-li u poločlenu žádné znaménko nebo bude-li u něj znaménko plus, prostě prohlásíme poločlen za člen.

Instance tříd `Plus`, `Minus`, `Krát` a `Děl` pak realizují konkrétní operaci v součtu, resp. součinu.

Rozhraní `IAritmVýraz`

608. No, dejme tomu. To si ještě rozmyslím, až dojdeme na místo, kde ten poločlen a ty členy zpracováváš. Takže kde začneš s výkladem?

Co požaduje

Začneme od rozhraní `IAritmVýraz`, které všechny třídy implementují (samozřejmě s výjimkou překladače a testu). Toto rozhraní požaduje po implementujících třídách tři věci:

- aby jejich instance uměly vytvořit svoji kopii,

- aby ve svém výrazu uměly všechny výskyty zadané proměnné nahradit zadaným výrazem a
- aby poté, co jim dodáme hodnoty jednotlivých proměnných, dokázaly svůj výraz spočítat a vrátit jeho hodnotu.

Výpis 35.6: Rozhraní `IAritmVýraz` specifikující vlastnosti vyhodnotitelných aritmetických výrazů

```
package rup.česky.vzory._35_interpet.kalkulačka;

/**************************************************************************
 * Instance třídy <code>IAritmVýraz</code>
 * představují vyhodnotitelné aritmetické výrazy.
 */

public interface IAritmVýraz extends Cloneable
{
//== DEKLAROVANÉ METODY ========================================================

    /**************************************************************************
     * Vrátí kopii daného výrazu.
     *
     * @return Kopie daného výrazu
     */
    public IAritmVýraz kopie();


    /**************************************************************************
     * Všechny výskyty proměnné zadané svým názvem nahradí ve svém výrazu
     * výrazem zadaným jako parametr a vrátí výraz po provedené substituci.
     *
     * @param proměnná Název proměnné nahrazované zadaným výrazem
     * @param výraz    Výraz nahrazující zadanou proměnnou
     * @return Výsledný výraz vzniklý z daného výrazu po nahrazení (sustituci)
     *         zadané proměnné zadaným výrazem
     */
    public IAritmVýraz nahraď( String proměnná, IAritmVýraz výraz );


    /**************************************************************************
     * Přiřadí proměnným hodnoty uložené v zadaném kontextu, po tomto
     * přiřazení výraz vyhodnotí a vrátí získanou hodnotu.
     *
     * @param ctx Kontext obsahující informace o proměnnných
     * @return    Výsledek vyhodnocení daného výrazu
     */
    public double vyhodnoť( Kontext ctx );

}
```

Třída `Kontext`

609. Na tu substituci se vysloveně těším. Teď by mne ale zajímalo, co je to ten kontext, který ve třetí metodě předáváš výrazu, který se má spočítat.

K čemu je

`Kontext` je třída, jejíž instance uchovávají informace o hodnotách jednotlivých proměnných. Představ si třeba, že jsi dělostřelec a máš takto naprogramovaný vzoreček,

který ti v závislosti na rychlosti a směru větru, tlaku vzduchu a vzdálenosti cíle spočítá, jak máš zamířit. Když se změní kterýkoliv z parametrů (nebo všechny současně), zadáš jejich nové hodnoty do kontextu, který předáš vyhodnocovací metodě, a ta ti vyhodnotí vzoreček pro nové hodnoty svých proměnných.

610. To je sympatické. A jak ty hodnoty proměnných nastavím?

Jak nastavit hodnoty proměnných

Instance třídy `Kontext` (viz výpis) rozlišují proměnné pouze podle jejich názvu. Budeš-li proto chtít inicializovat nějakou proměnnou, musíš danému kontextu zadat řetězec ve tvaru *NázevProměnné* = *PožadovanáHodnota*, např. `"x=7"`. Inicializační hodnoty proměnných přitom můžeš zadat jak při volání konstruktoru daného kontextu, tak kdykoliv později.

Více inicializací v jednom řetězci

Aby se hodnoty proměnných lépe zadávaly, naprogramoval jsem příslušné metody tak, že můžeš v jednom řetězci zadat několik inicializací. Jednotlivé inicializace pak musíš oddělovat čárkou nebo středníkem – např. `"a=5, b=6, c=7"`. Na mezerách vložených do okolí rovnítek, čárek a středníků přitom nezáleží.

611. Jestli jsem to dobře pochopil, tak proměnná vystupuje ve výrazu jenom jako název a to, jakou má hodnotu, se výraz dozví na poslední chvíli z dodaného kontextu.

Přesně tak. Můžeš mít dokonce několik předpřipravených kontextů s různými sadami proměnných a jejich počátečních hodnot a ve vhodnou chvíli předhodit výrazu tu správnou sadu k vyhodnocení.

612. Proč je metoda `nastav(String...)` definována jako finální a ostatní metody ne?

Proč je `nastav(String…)` konečná

Metoda `nastav` je volána konstruktorem a v něm bys určitě neměl používat překrytelné metody. Kdybys chtěl někdy později definovat potomka kontextu a tuto metodu v něm překrýt, mohl bys kód upravit tak, že tělo metody vystěhuješ do nějaké soukromé (a tím nepřekrytelné) metody, kterou bude volat konstruktor i metoda `nastav(String...)`. Pak ji můžeš překrývat, aniž by sis koledoval o malér.

613. Proč pro mapu používáš třídu `TreeMap` a ne daleko běžnější `HashMap`?

Proč je použita `TreeMap`

Protože při použití `HashMap` budou proměnné v množině vracené metodou `getProměnné()` seřazené náhodně, což se mi nelíbilo, ale byl jsem líný vymýšlet nějakou efektivnější metodu řazení proměnných podle abecedy před jejich tiskem.

Ve skutečné aplikaci bych ale asi opravdu dal přednost třídě `HasMap` a proměnné seřadil až těsně před jejich případným tiskem. Mnou použité řešení je sice bezpracné, ale pokud by ses často obracel na kontext, abys do něj něco přidal, tak by mohlo zdržovat. Zatím mne toto řešení nezdržovalo, tak jsem jeho optimalizaci neřešil.

Výpis 35.7: Definice třídy `Kontext` uchovávající hodnoty použitých proměnných

```
package rup.česky.vzory._35_interpet.kalkulačka;

import java.util.Set;
import java.util.TreeMap;
import java.util.Map;

/*******************************************************************************
```

```
 * Instance třídy <code>Kontext</code> představují sady proměnných
 * s přiřazenými hodnotami.
 */
public class Kontext
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    public static final Double DOUBLE_0 = 0.0;


//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================================

    private final Map<String, Double> hodnoty = new TreeMap<String, Double>();


//== KONSTRUKTORY A TOVÁRNÍ METODY ============================================

    /**************************************************************************
     * Vytvoří nový, prázdný kontext.
     */
    public Kontext()
    {
    }


    /**************************************************************************
     * Vytvoří nový kontext a vloží do něj inicializované proměnné
     * zadané v parametru.
     *
     * @param proměnné Textové řetězce obsahující názvy proměnných a jejich
     *                 hodnoty. Podrobnosti v metodě {@link nastav(String...)}.
     *
     */
    public Kontext( String... proměnné )
    {
        nastav( proměnné );
    }


//== NESOUKROMÉ METODY INSTANCÍ ===============================================

    /**************************************************************************
     * Vrátí množinu instancí třídy {@link Map.Entry} obsahujících
     * informace o všech proměnných v zadaném kontextu, přičemž
     * každá z instancí v množině obsahuje název proměnné a její hodnotu.
     */
    public Set<Map.Entry<String,Double>> getProměnné()
    {
        return hodnoty.entrySet();
    }


    /**************************************************************************
     * Vrátí hodnotu proměnné se zadaným názvem.
     *
     * @param název Název proměnné, jejíž hodnotu zjišťujeme
     * @return      Hodnota zadané proměnné
     */
```

```
public double hodnota( String název )
{
    Double hodnota = hodnoty.get( název );
    if( hodnota == null )
        throw new IllegalStateException(
            "\nProměnná " + název + " neexistuje");
    return hodnota.doubleValue();
}


/*****************************************************************************
 * Vrátí hodnotu zadané proměnné.
 *
 * @param proměnná Proměnná, jejíž hodnotu zjišťujeme
 * @return Hodnota zadané proměnné
 */
public double hodnota( Proměnná proměnná )
{
    return hodnota( proměnná.getNázev() );
}


/*****************************************************************************
 * Nastaví zadaným proměnným zadané počáteční hodnoty.
 * Požadovaná nastavení se zadávají jako textové řetězce ve tvaru<br>
 * <code><i>NázevProměnné</i> = <i>hodnota</i></code><br>
 * V jednom řetězci je možné inicializovat více proměnných.
 * Jednotlivé inicializace se v takovém případě oddělují čárkou nebo
 * středníkem. Případné mezery v okolí rovnítek, čárek a středníků
 * se ignorují.
 *
 * @param inicializace  Pole incializačních řetězců
 */
public final void nastav( String... inicializace )
{
    for( String dávka : inicializace ) {
        String[] sada = dávka.split( " *[,;] *" );
        for( String přiřazení : sada ) {
            String[] ph = přiřazení.split( " *= *" );
            přiřaď( Double.parseDouble( ph[1] ), ph[0] );
        }
    }
}


/*****************************************************************************
 * Přiřadí proměnným zadaným svým názvem zadanou hodnotu.
 *
 * @param název    Název proměnné, které přiřazujeme hodnotu
 * @param hodnota  Přiřazovaná hodnota
 */
public void přiřaď( double hodnota, String... název )
{
    Double h = Double.valueOf( hodnota );
    for( String s : název )
        hodnoty.put( s, h );
}
```

```
    /*****************************************************************************
     * Přiřadí zadaným proměnným nulovou hodnotu.
     *
     * @param proměnné Názvy nulovaných proměnných oddělené bílými znaky
     */
    public void nuluj( String... proměnná )
    {
        for( String s : proměnná )
            hodnoty.put( s, DOUBLE_0 );
    }

}
```

Konstanty a proměnné

614. Kontext jsem pochopil Co přijde nyní?

Podíváme se na konstanty a proměnné. Konstanty jsou jednodušší, tak je proberu jako první.

Třída Konstanta

Konstanta má život jednoduchý, protože svoji hodnotu dostane přidělenu již při svém vzniku. Při kopírování se nemusí o nic starat a vzhledem ke své podstatě nemá ani co substituovat, takže jí stačí vrátit svoji kopii. Zkrátka nuda.

Výpis 35.8: Definice třídy `Konstanta`

```
package rup.česky.vzory._35_interpet.kalkulačka;

/*******************************************************************************
 * Instance třídy Konstanta představují konstanty vystupující
 * v aritmetických výrazech.
 */

public class Konstanta implements IAritmVýraz
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    private final double hodnota;   //Hodnota dané konstanty


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří novou konstantu se zadanou hodnotou.
     *
     * @param hodnota Hodnota vytvářené konstanty
     */
    public Konstanta( double hodnota ) {
        this.hodnota = hodnota;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * {@inheritDoc}
```

```
     */
    public IAritmVýraz kopie()
    {
        return new Konstanta( hodnota );
    }


    /*************************************************************************
     * Protože v konstantě nelze nahradit proměnnou výrazem,
     * vrátí po "aplikované substituci" kopii sebe sama.
     */
    public IAritmVýraz nahraď( String proměnná, IAritmVýraz výraz )
    {
        return new Konstanta( hodnota );
    }


    /*************************************************************************
     * Nezávisle na kontextu vrací konstanta vždy svoji hodnotu.
     */
    public double vyhodnoť( Kontext ctx )
    {
        return hodnota;
    }


    /*************************************************************************
     * Vrátí řetězcovou podobu hodnoty výrazu.
     */
    public String toString()
    {
        return String.valueOf( hodnota );
    }


}
```

615. Proměnné jsou také nuda.

Proměnné jsou přece jenom o maličko zajímavější. Při svém vyhodnocování se musí na svoji hodnotu zeptat kontextu a trochu práce mají i se svojí substitucí.

V čem jsou proměnné zajímavější

Budeš-li chtít po nějaké proměnné, aby místo jiné proměnné substituovala nějaký výraz, tak se na tebe vykašle a stejně jako konstanta vrátí svoji kopii, protože opravdu není co substituovat.

Jediná situace, kdy se může dít něco zajímavějšího, nastane v okamžiku, kdy proměnnou požádáš, aby substituovala sama sebe. Pak proměnná vrátí místo sebe výraz, kterým ji chceš ve výrazech nahradit.

616. Přiznám se, že tohle moc nechápu.

Mechanismus substituce

Jde o to, že všechny výrazy, které sestávají z více částí, řeší nahrazování proměnné výrazem tak, že o ně požádají svoje části (vzpomeň si na návrhový vzor *Řetěz odpovědnosti*, který jsme probírali v kapitole *Horký brambor (Řetěz odpovědnosti – Chain of Responsibility)* na straně 361). Požadavek propadá stromem, jenž je reprezentací

daného výrazu, tak dlouho, až dopadne na konstantu nebo proměnnou. Konstanty a jinak pojmenované proměnné pošlou do substituovaného výrazu svoji kopii, takže se v něm nic nezmění. Substituovaná proměnná však místo sebe pošle substituční výraz a udělá to na všech místech, kde se v původním výrazu vyskytovala. Výraz se tím transformuje na jiný, v němž je místo všech výskytů oné proměnné zadaný výraz.

617. To se mi líbí. A už také chápu, proč je metoda definována právě takto.

Výpis 35.9: Definice třídy `Proměnná`

```
package rup.česky.vzory._35_interpet.kalkulačka;

/*******************************************************************************
 * Instance třídy Proměnná představují proměnné
 * vystupující v aritmetických výrazech.
 */

public class Proměnná implements IAritmVýraz
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    private final String název; //Název proměnné


//== KONSTRUKTORY A TOVÁRNÍ METODY =============================================

    /*****************************************************************************
     * Vytvoří novou proměnnou se zadaným názvem.
     *
     * @param název   Název vytvářené proměnné
     */
    public Proměnná( String název )
    {
        this.název  = název;
    }


//== NESOUKROMÉ METODY INSTANCÍ ================================================

    /*****************************************************************************
     * Vrátí název dané proměnné.
     *
     * @return Název dané proměnné
     */
    public String getNázev()
    {
        return název;
    }


    /*****************************************************************************
     * {@inheritDoc}
     */
    public IAritmVýraz kopie()
    {
        return new Proměnná( název );
    }
```

```
    /*****************************************************************************
     * {@inheritDoc}
     * Má-li se nahradit daná proměnná, vrátí nahrazující výraz,
     * má-li se nahradit jiná proměnná, na výrazu (proměnné) se náhradou
     * nic nezmění, a proto vrátí sebe sama.
     */
    public IAritmVýraz nahraď( String názevProměnné, IAritmVýraz výraz )
    {
        if( názevProměnné.equals( název ) )
            //Je to nahrazovaná proměnná => nahradí se
            return výraz.kopie();
        else
            //Není to nahrazovaná proměnná => bude dále vystupovat ve výrazu
            return new Proměnná( název );
    }


    /*****************************************************************************
     * Vrátí hodnotu uloženou pro danou proměnnou v kontextu.
     */
    public double vyhodnoť( Kontext ctx )
    {
        return ctx.hodnota( název );
    }


    /*****************************************************************************
     * Vrátí název dané proměnné.
     */
    public String toString()
    {
        return název;
    }

}
```

Binární operátory

618. Konstanty a proměnné jsme probrali, takže z výrazů nám již zbývá pouze třída `Binární` a její potomci.

Čím se potomci liší

Správně. Všechny třídy reprezentující binární operace mají společného rodiče, protože díky němu mohou většinu kódu sdílet. Jediné, co mají vlastní, je vedle konstruktoru metoda `vyhodnoť(double,double)`, pomocí níž rodič implementuje požadovanou metodu `vyhodnoť(Kontext)`.

619. Proč rodič definuje atribut `operátor`, když potřebnou operaci stejně provádějí potomci?

K čemu je atribut `operátor`

Atribut `operátor` neslouží k vyhodnocování výrazů, ale k vytvoření textové podoby reprezentovaného výrazu. Všechny třídy jsem totiž oproti nezbytnému minimu trochu rozšířil. Vedle vlastní práce s výrazem jsem je vybavil i schopností zobrazit textovou podobu kódu, který reprezentují.

Textová podoba reprezentovaného výrazu

U tříd reprezentujících binární operátory nebude tato textová podoba nejspíš přesně ta, kterou jsi zadal, protože při výrobě kódu instance pro jistotu vkládají na nebezpečná místa závorky, ale měl by to být kód ekvivalentní.

K čemu jsou dobré

Touto vlastností jsem výrazy vybavil proto, aby se ti lépe analyzovaly různé stromy vzniklé překladem zadaného textu. Když tedy debugger požádáš, aby ti u každého uzlu stromu ukázal i jeho textovou reprezentaci, budeš průběžně vždy vědět, kterou část zadaného textu daná instance reprezentuje.

Vytváří se až na požádání

Protože lze ale předpokládat, že většinou nebudeš tuto textovou podobu reprezentovaného výrazu potřebovat, definoval jsem ji tak, že se začne vytvářet až v okamžiku, kdy o ni poprvé požádáš. Když už se pak jednou vytvoří, tak si ji instance zapamatuje v atributu `kód` pro případ, že by o ni byla žádána znovu.

620. Chápu. V kódu vidím, jak binární operátory rozdělují vyhodnocení výrazu mezi své operandy. Jenom je mi divné, proč metody `kopie()` a `nahraď(String,IAritmVýraz)` napřed instance klonují, aby v nich pak něco upravovaly. Já bych je definoval tak, že bych zavolal konstruktor, kterému bych předal kopie či substituce svých operandů a vrátil od něj získaný odkaz.

K tomu, abys mohl něco podobného udělat, bys ale potřeboval, aby třída nebyla abstraktní. Od abstraktní třídy nemůžeš vytvořit instanci, takže nechceš-li tuto metodu svěřit potomkům (a smířit se s tím, že se pak bude duplikovat kód), tak ti nezbývá, než se „zbaběle“ uchýlit k systémovému `clone()`, který ti vrátí instanci příslušného potomka.

Výpis 35.10: Definice třída `Binární`, která je společným rodičem všech výrazů reprezentujících binární operace

```
package rup.česky.vzory._35_interpet.kalkulačka;

import rup.česky.společně.UEX;

/*******************************************************************************
 * Instance třídy <code>Binární</code> představují výraz
 * reprezentující binární operaci.
 */
abstract public class Binární implements IAritmVýraz, Cloneable
{
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================================

    /** Levý operand.  */ protected IAritmVýraz levý;
    /** Pravý operand. */ protected IAritmVýraz pravý;
    /** Textová podoba kódu, jehož reprezentací je daný výraz. */
    private String kód;
    private char   operátor;


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*******************************************************************************
     * Vytvoří nový výraz binární operace.
     *
```

```
     * @param levý  Levý operand
     * @param pravý Pravý operand
     */
    public Binární( char operátor, IAritmVýraz levý, IAritmVýraz pravý )
    {
        this.operátor = operátor;
        this.levý  = levý;
        this.pravý = pravý;
    }


//== ABSTRAKTNÍ METODY ==========================================================

    /**************************************************************************
     * {@inheritDoc}
     */
    abstract protected double vyhodnoť( double levý, double pravý );


//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /**************************************************************************
     * {@inheritDoc}
     */
    public Binární kopie()
    {
        Binární kopie;
        try { kopie = (Binární)this.clone();
        } catch (CloneNotSupportedException ex) {throw new UEX(ex);}
        kopie.levý  = levý .kopie();
        kopie.pravý = pravý.kopie();
        return kopie;
    }


    /**************************************************************************
     * {@inheritDoc}
     */
    public IAritmVýraz nahraď( String proměnná, IAritmVýraz výraz )
    {
        Binární subst;
        try { subst = (Binární)this.clone();
        } catch (CloneNotSupportedException ex) {throw new UEX(ex);}
        subst.levý  = levý .nahraď( proměnná, výraz );
        subst.pravý = pravý.nahraď( proměnná, výraz );
        subst.kód   = null;
        return subst;
    }


    /**************************************************************************
     * {@inheritDoc}
     */
    public double vyhodnoť( Kontext ctx )
    {
        double levý  = this.levý .vyhodnoť( ctx );
```

```
        double pravý = this.pravý.vyhodnoť( ctx );
        return vyhodnoť( levý, pravý );
    }


    /*****************************************************************************
     * Vrátí řetězcovou podobu hodnoty výrazu.
     */
    public String toString()
    {
        if( kód == null )
            kód = "(" + levý + ") " + operátor + " (" + pravý + ")";
        return kód;
    }

}
```

621. Nachytal jsi mne na švestkách. Pojď mi raději ukázat její potomky.

Ukážu ti pouze jednoho, protože jsou si všichni velmi podobní.

Výpis 35.11: Definice třídy `Plus` reprezentující součet dvou operandů

```
package rup.česky.vzory._35_interpet.kalkulačka;

/*******************************************************************************
 * Instance třídy <code>Plus</code> představují součet dvou výrazů.
 */

public class Plus extends Binární
{
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /*****************************************************************************
     * Vytvoří nový výraz představující součet zadaných výrazů.
     *
     * @param levý  Levý operand
     * @param pravý Pravý operand
     */
    public Plus( IAritmVýraz levý, IAritmVýraz pravý )
    {
        super( '+', levý, pravý );
    }


//== NESOUKROMÉ METODY INSTANCÍ =================================================

    /*****************************************************************************
     * {@inheritDoc}
     */
    protected double vyhodnoť( double levý, double pravý )
    {
        return levý  +  pravý;
    }
}
```

Třída Překladač

622. Takže už zbývá jen třída překladač. Co ta dělá?

Třída `Překladač` je schopna přeložit zadaný text do vnitřní reprezentace a vrátit vytvořený výraz k následné interpretaci.

623. To je to, co jsme dělali v minulém příkladu ručně. Tak to mi budeš muset vysvětlit podrobněji.

Já bych řekl, že když se podíváš na její kód, tak vše poměrně rychle pochopíš. Jediné, co by tě mohlo zaskočit, je štafetový kolík představovaný instancí vnořené třídy `Zdroj`. Protože jsme ale podobný štafetový kolík využívali i při rozboru tříd testujících regulární výrazy, řekl bych, že s ní nebudeš mít problémy.

624. Tak nevím, jestli mi opravdu tak věříš, anebo se už těšíš na konec a jsi líný mi to vyložit. Pojď to projít alespoň stručně.

Opravdu se domnívám, že program je poměrně srozumitelný, protože jednotlivé metody přesně interpretují definici gramatiky uvedenou v dokumentačním komentáři (jestli tě při jejím čtení obtěžují HTML značky, najdeš ji také na str. 488).

Výpis 35.12: Knihovní třída `Překladač` pro převod textového zápisu programu do vnitřního tvaru

```
package rup.česky.vzory._35_interpet.kalkulačka;

import java.util.Map;

/*******************************************************************************
 * Instance třídy Překladač představují lexikální a syntaktické analyzátory,
 * které převedou text do stromové struktury vnitřní reprezentace programu.
 * <p>
 * Definice gramatiky použitého jazyka je:<br>
 * <b><i>Konstanta</i> ::= #</b><i>číslo</i><b>#</b><br>
 * <b><i>Proměnná</i> ::=</b> <i>identifikátor</i><br>
 * <b><i>Poločlen</i> ::=</b> <i>Proměnná</i> | <i>Konstanta</i> |
 *                        '<b>(</b>' <i>Výraz</i> '<b>)</b>' <br>
 * <b><i>Člen</i> ::=</b> <i>Poločlen</i> | '<b>+</b>'<i>Poločlen</i> |
 *                       '<b>-</b>'<i>Poločlen</i> <br>
 * <b><i>Součin</i> ::=</b> <i>Člen</i> |
 *                          <i>Součin</i> '<b>*</b>' <i>Člen</i> |
 *                          <i>Součin</i> '<b>/</b>' <i>Člen</i> <br>
 * <b><i>Součet</i> ::=</b> <i>Součin</i> |
 *                          <i>Součet</i> '<b>+</b>' <i>Součin</i> |
 *                          <i>Součet</i> '<b>-</b>' <i>Součin</i> <br>
 * <b><i>Výraz</i> ::=</b> <i>Součet</i>
 */

public class Překladač
{
//== NESOUKROMÉ METODY TŘÍDY ==================================================

    /***************************************************************************
```

```
     * Převede zadaný text na objekt typu <code>IAritmVýraz</code>,
     * který je vnitřní reprezentací zadaného kódu.
     *
     * @param text Text, který je třeba přeložit
     * @return Vnitřní reprezentace zadaného kódu
     */
    public static IAritmVýraz přeložVýraz( String text )
    {
        Zdroj zdroj = new Zdroj( text );
        return přeložVýraz( zdroj );
    }


//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    /**************************************************************************
     * Soukromý konstruktor zabraňuje vytváření instancí.
     */
    private Překladač() {}


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ============================================

    /**************************************************************************
     * Přečte další platný znak ze zadaného zdrojového kódu.
     *
     * @param zdroj  Přepravka obsahující všechny potřebné informace
     *               o zpracovávaném zdrojovém programu
     * @return Další znak ze zadaného zdrojového kódu
     *         není-li ve zdrojovém kódu další znak, vrátí nulu
     */
    private static char dalšíZnak( Zdroj zdroj )
    {
        char znak;
        for( int i = zdroj.pozice;   i < zdroj.text.length();   i++ ) {
            znak = zdroj.text.charAt( i );
            if( ! Character.isWhitespace( znak ) ) {
                zdroj.pozice = i+1;
                zdroj.naŘadě = znak;
                return znak;
            }
        }
        return 0;
    }


    /**************************************************************************
     * Přeloží text začínající na aktuální pozici zdroje
     * a vrátí objekt představující daný výraz.
     *
     * @param zdroj  Přepravka obsahující všechny potřebné informace
     *               o zpracovávaném zdrojovém programu
     * @return Vnitřní reprezentace přeloženého výrazu
     */
    private static IAritmVýraz přeložVýraz( Zdroj zdroj )
    {
        return přeložSoučet( zdroj );
    }
```

```
/*****************************************************************************
 * Přeloží text začínající na aktuální pozici zdroje
 * a vrátí objekt představující daný výraz.
 *
 * @param zdroj  Přepravka obsahující všechny potřebné informace
 *               o zpracovávaném zdrojovém programu
 * @return Vnitřní reprezentace přeloženého výrazu
 */
private static IAritmVýraz přeložSoučet( Zdroj zdroj )
{
    IAritmVýraz součet = přeložSoučin( zdroj );
    if( zdroj.naŘadě == '+' )
        součet = new Plus( součet, přeložSoučin( zdroj ) );
    else if( zdroj.naŘadě == '-' )
        součet = new Minus( součet, přeložSoučin( zdroj ) );
    return součet;
}


/*****************************************************************************
 * Přeloží text začínající na aktuální pozici zdroje
 * a vrátí objekt představující daný výraz.
 *
 * @param zdroj  Přepravka obsahující všechny potřebné informace
 *               o zpracovávaném zdrojovém programu
 * @return Vnitřní reprezentace přeloženého výrazu
 */
private static IAritmVýraz přeložSoučin( Zdroj zdroj )
{
    IAritmVýraz součin = přeložČlen( zdroj );
    if( zdroj.naŘadě == '*' )
        součin = new Krát( součin, přeložČlen( zdroj ) );
    else if( zdroj.naŘadě == '/' )
        součin = new Děl( součin, přeložČlen( zdroj ) );
    return součin;
}


/*****************************************************************************
 * Přeloží text začínající na aktuální pozici zdroje
 * a vrátí objekt představující daný výraz.
 *
 * @param zdroj  Přepravka obsahující všechny potřebné informace
 *               o zpracovávaném zdrojovém programu
 * @return Vnitřní reprezentace přeloženého výrazu
 */
private static IAritmVýraz přeložČlen( Zdroj zdroj )
{
    int start = zdroj.pozice;
    char znak = dalšíZnak( zdroj );     //Další nebílý znak
    if( znak == '-' ) {
        znak = dalšíZnak( zdroj );
        return new Krát( new Konstanta( -1 ), přeložPoločlen( zdroj ) );
    }else if( znak == '+' ) {
        znak = dalšíZnak( zdroj );
    }
```

```
        return přeložPoločlen( zdroj );
    }


    /*****************************************************************************
     * Přeloží text začínající na aktuální pozici zdroje
     * a vrátí objekt představující daný výraz.
     *
     * @param zdroj  Přepravka obsahující všechny potřebné informace
     *               o zpracovávaném zdrojovém programu
     * @return Vnitřní reprezentace přeloženého výrazu
     */
    private static IAritmVýraz přeložPoločlen( Zdroj zdroj )
    {
        IAritmVýraz člen = null;
        char znak = zdroj.naŘadě;
        if( znak == '#' ) {
            člen = přeložKonstantu( zdroj );
        }else if(  Character.isJavaIdentifierStart( znak ) ) {
            člen = přeložProměnnou( zdroj );
        }else if( znak == '(' ) {
            člen = přeložVýraz( zdroj );
            if( zdroj.naŘadě != ')' )
                CHYBA(zdroj);
        }else
            CHYBA(zdroj);
        dalšíZnak( zdroj );
        return člen;
    }


    /*****************************************************************************
     * Očekává ve zdrojovém programu konstantu (literál)
     * a vrátí instanci, která tuto konstantu představuje.
     */
    private static Konstanta přeložKonstantu( Zdroj zdroj )
    {
        int konec = zdroj.pozice;
        while( zdroj.text.charAt( konec++ ) != '#' );
        double hodnota = Double.parseDouble(
                                zdroj.text.substring(zdroj.pozice, konec-1) );
        zdroj.pozice = konec;
        return new Konstanta( hodnota );
    }


    /*****************************************************************************
     * Očekává ve zdrojovém textu název proměnné
     * a vrátí instanci, která tuto proměnnou představuje.
     */
    private static Proměnná přeložProměnnou( Zdroj zdroj )
    {
        int počátek, konec = počátek = zdroj.pozice-1;
        String text = zdroj.text;
        while( (++konec < text.length())  &&
               Character.isJavaIdentifierPart( text.charAt(konec) ) );
        Proměnná p = new Proměnná(zdroj.text.substring(počátek, konec));
        zdroj.pozice = konec;
```

```
        return p;
    }


    /*****************************************************************************
     * Společná reakce na jakoukoliv chybu odhalenou v překládaném výrazu.
     * Metoda je ale vyvolávána pouze v reakci na nejmarkantnější chyby.
     */
    private static void CHYBA(Zdroj zdroj)
    {
        throw new IllegalStateException( "\nZadaný výraz je neplatný:\n" +
            zdroj.text.substring( 0, zdroj.pozice ) +
            " -¤- " +
            zdroj.text.substring( zdroj.pozice ));
    }


//== VNOŘENÉ A VNITŘNÍ TŘÍDY ================================================

    /*****************************************************************************
     * Instance třídy <code>Zdroj</code> představují zdrojový text
     * překládaného výrazu včetně pozice, kam až se v něm překladač dostal,
     * a znaku, který uvozuje následující část výrazu.
     */
    private static class Zdroj
    {
        String text;
        char   naŘadě = 0;
        int    pozice = 0;

        Zdroj( String text ) {
            this.text = text;
        }

        String kód( int start ) {
            return text.substring( start, pozice );
        }
    }
}
```

Použití interpretu v programu

625. Přiznávám, dalo se to číst. Takže už zbývá jenom ukázat, jak lze celý tento komplex jednoduše používat v programu.

To demonstruje třída `TestPřekladače`. V její metodě `test()` se můžeš podívat, jak se v kontextu přiřazují hodnoty proměnným.

První výrazy jsou předány zpracovávacím metodám v textové podobě. Metody je převedou do vnitřní reprezentace, kterou nechají vyhodnotit, a výsledek vytisknou.

Poslední vyhodnocovaný výraz ukazuje možnosti substituce. Při přípravě vzorečku jsem si musel vzpomenout, jak jsme vyhodnocovali odpor elektrických obvodů sestavených z kombinací sériového a paralelního spojení odporů zadané velikosti a jak se nám v průběhu výpočtu vzorečky nechutně zesložiťovaly. S použitím našeho interpretu by stačilo pojmenovat bloky a napsat vzoreček pro výpočet odporu obvodu

sestaveného z těchto bloků. V dalších etapách bychom za proměnnou představující jednotlivé bloky dosazovali vzorečky pro výpočet odporu daného bloku.

Výpis 35.13: Třída `TestPřekladače` předvádějící použití výše definovaného interpretu

```
package rup.česky.vzory._35_interpet.kalkulačka;

import java.util.Map;


/*****************************************************************************
 * Třída <code>TestPřekladače</code> slouží jako testovací třída pro ověření
 * funkce třídy <code>Překladač</code> a tříd s ní spolupracujících.
 */

public class TestPřekladače
{
//== KONSTRUKTORY A TOVÁRNÍ METODY ==============================================

    private TestPřekladače() {}

//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ============================================

    /***************************************************************************
     * Vyhodnotí zadaný kód,
     * do nějž za proměnné dosadí hodnoty ze zadaného kontextu.
     * Výsledek vypíše na standardní výstup.
     */
    private static void zpracuj( String výraz, Kontext kontext )
    {
        zpracuj( Překladač.přeložVýraz( výraz ), kontext );
    }


    /***************************************************************************
     * Vyhodnotí zadaný aritmetický výraz,
     * do nějž za proměnné dosadí hodnoty ze zadaného kontextu.
     * Výsledek vypíše na standardní výstup.
     */
    private static void zpracuj( IAritmVýraz výraz, Kontext kontext )
    {
        double výsledek   = výraz.vyhodnoť( kontext );
        StringBuilder sb = new StringBuilder();
        sb.append("Výsledkem výrazu ").append(výraz).append(" pro:");
        for( Map.Entry<String,Double> mesd : kontext.getProměnné() ) {
            sb.append( "\n   " )
              .append( mesd.getKey() )
              .append( " = " )
              .append( mesd.getValue() );
        }
        sb.append( "\nje hodnota " ).append( výsledek ).append( "\n\n" );
        System.out.println( sb );
    }


//== TESTY A METODA MAIN ========================================================
```

```
    /*****************************************************************************
     * Testovací metoda.
     */
    public static void test()
    {
        Kontext ctx = new Kontext();
        String zdroj;

        ctx.přiřaď( 3, "X" );
        ctx.přiřaď( 4, "Y" );

        zpracuj( "((X + Y) * (X - Y))", ctx );
        zpracuj( "X * X  +  Y * Y", ctx );
        zpracuj( "#3.14# * X * Y", ctx );
        zpracuj( "-#5# * -Deset -Padesát",
                 new Kontext( "Deset=10, Padesát=50") );

        IAritmVýraz výraz = Překladač.přeložVýraz( "(x + y) / (x*y)" );
        IAritmVýraz subst = Překladač.přeložVýraz( "a + b" );
        IAritmVýraz nový  = výraz.nahraď( "x", subst );
        Kontext kontext = new Kontext( "y=2", "a=7, b=3" );
        zpracuj( nový, kontext );
    }
    /** @param ppr Parametry příkazového řádku - nepoužité */
    public static void main(String[]ppr){ test(); }/*-*/
}
```

Shrnutí – co jsme se naučili

- Návrhový vzor *Interpret* ukazuje, jak je možno relativně jednoduše implementovat interpret vlastního jazyka.
- Vzor se zabývá pouze vnitřní reprezentací programu a její interpretací. Neřeší převod textového zápisu do této reprezentace.
- Vzor doporučuje převést gramatická pravidla do systému tříd. Instance těchto tříd pak budou představovat části programu.
- Vzor je velmi efektivní při implementaci jednodušších gramatik. U složitějších gramatik je výhodnější sáhnout po klasických postupech.
- Vzor umožňuje snadnou realizaci i poměrně rafinovaných operací, mezi něž patří např. substituce proměnné výrazem apod.
- Přeložené programy je možné spouštět nad různými sadami hodnot jejich proměnných uložených v instanci nějakého kontextu.
- Návrhový vzor *Interpret* patří mezi vzory uvedené v GoF.

ČÁST 8

Přílohy

- PŘÍLOHA A **Základy jazyka UML**
- PŘÍLOHA B **Seznam doporučené a nedoporučené literatury**

PŘÍLOHA A

Základy jazyka UML

- **Jazyk UML**
- **Diagramy tříd**

Jazyk UML

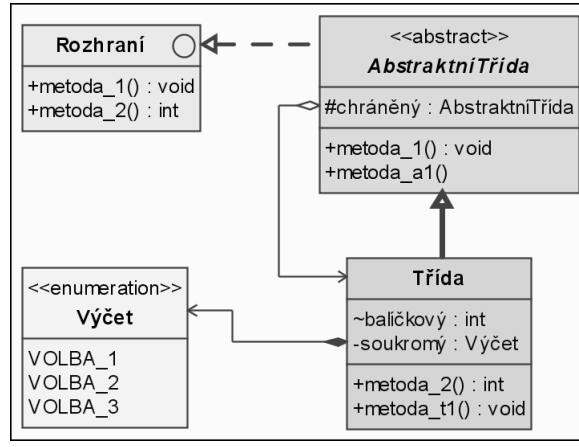
Jazyk UML je duchovním dítětem tří autorů: Gradyho Booche, Jamese Rumbaugha a Ivara Jacobsona. Každý z nich vytvořil v průběhu 80. let vlastní systém pro grafický zápis různých etap vývoje programových systémů. Na počátku devadesátých let si začali jeden od druhého nápady půjčovat a nakonec se dohodli, že své systémy sloučí v systém jediný, který nazvali *Unified Modeling Language* (sjednocený modelovací jazyk), zkráceně UML.

Jazyk UML se skládá z mnoha grafických prvků, které se dají podle pravidel jazyka vzájemně kombinovat do podoby diagramů. UML definuje 13 diagramů:

- Diagramy chování – Behavior diagrams
  - Diagram případů užití – Use case diagram
  - Stavový diagram – State chart (state machine) diagram
  - Diagram činností – Activity diagram
- Strukturální diagramy – Structure diagrams
  - Diagram tříd – Class diagram
  - Diagram objektů – Object diagram
  - Diagram komponent – Component diagram
  - Kombinovaný strukturální diagram – Composite structure diagram
  - Diagram balíčků – Package diagram
  - Diagram nasazení – Deployment diagram
- Interakční diagramy– Interaction diagrams
  - Diagram spolupráce – Collaboration diagram
  - Diagram komunikací – Communication diagram
  - Sekvenční diagram – Sequence diagram
  - Přehledový diagram interakcí – Interaction overview

Diagramy tříd

V této knize jsou použity pouze diagramy tříd. I když se nazývají diagramy tříd, slouží k zobrazení všech datových typů, tj. tříd, rozhraní i výčtových typů. Do diagramu přitom zakreslujeme většinou pouze ty typy, které jsou pro pochopení dané části programu důležité.



Obrázek A.1
Jednoduchý diagram tříd

Datové typy

V diagramu tříd je každý datový typ (třída, rozhraní, výčtový typ) zobrazen jako obdélník, který může být vodorovně rozdělen na několik částí. Je-li rozdělen, mají jednotlivé části následující význam:

- Název datového typu doplněný případným stereotypem (textem uzavřeným mezi «francouzské uvozovky») označujícím nějakou speciální vlastnost daného typu (např. že se jedná o interface)
- Atributy (nemusí být všechny, stačí uvést jen ty, které jsou v daný okamžik zajímavé)
- Metody (opět stačí uvést jen ty, které jsou v daný okamžik zajímavé)
- Vyhazované výjimky

Je-li zobrazena některá z částí, musí být v zájmu jednoznačnosti zobrazeny i všechny části nad ní, byť by v nich nebyl zobrazen žádný údaj.

Atributy

Atributy se zobrazují v sekci pod názvem třídy. U každého atributu se nejprve zobrazuje symbol jejich dostupnosti (přístupu):

– `private`

\+ `public`

\# `protected`

~ implicitní přístup (`package private`)

Vlastní atribut je pak uváděn v pascalské syntaxi, tj. název atributu následovaný dvojtečkou a jeho typem.

Metody

Metody zobrazujeme v sekci pod atributy. Obdobně jako u atributů u nich nejprve uvádíme dostupnost. Následuje název metody, seznam parametrů a případně dvojtečka a za ní typ návratové hodnoty. Není-li však pro objasnění funkce metody typ návratové hodnoty nutný, nemusíme jej uvádět.

Vztahy mezi datovými typy

Mezi datovými typy mohou existovat nejrůznější vztahy. Ty se zobrazují čárami mezi datovými typy, které mezi sebou mají nějaký vztah. Tyto čáry mohou být nahrazeny šipkami vedoucími od závislých typů k typům, na nichž tyto typy závisí.

Čeština nemá pro objekt, na němž je jiný objekt závislý, žádné příhodné označení. Budu jej proto v dalším textu nazývat *primární*. Všechny závislosti mezi typy znázorňuji v knize šipkami, které vedou od závislého typu k primárnímu typu, tj. ukazují na primární typ.

V obrázcích v této knize se můžete setkat s následujícími vztahy mezi dvěma datovými typy:

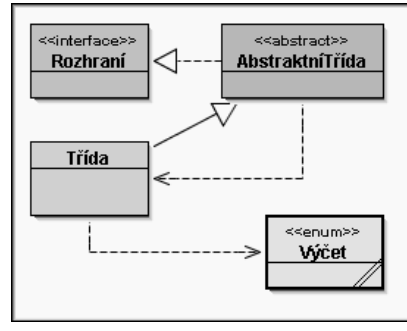
- *Závislý typ používá metody či instance primárního datového typu.*
  Tuto závislost znázorňujeme jednoduchou čárkovanou šipkou.
- *Závislý typ má atributy, které jsou instancemi primárního typu.*
  K patě šipky zmíněné v předchozím bodě přidáme kosočtverec. V tomto případě ještě rozlišujeme prázdný a vyplněný kosočtverec.
  - Prázdný, nevyplněný kosočtverec naznačuje volnější vazbu, při níž není odkazovaný objekt nedělitelnou součástí svého majitele. Zánik majitele (instance závislého typu) tak nemusí mít za důsledek zánik odkazované instance (zrušení instance reprezentující knihu nevede ke zrušení instance reprezentující jejího vydavatele).
  - Vyplněný kosočtverec symbolizuje pevnou vazbu, při níž je odkazovaný objekt integrální součástí svého majitele (např. motor je součástí auta) a jeho samostatná existence bez majitele nemá v daném programu smysl. Z toho také plyne, že se zánikem svých majitelů většinou automaticky zanikají i příslušné vlastněné objekty.
- *Závislý datový typ je potomkem primárního datového typu.*
  Tuto závislost znázorňujeme plnou šipkou s nevyplněnou trojúhelníkovou hlavou. Je-li několik typů potomkem společného rodiče, mohou se jednotlivé šipky na konci spojit a „dorazit“ ke společnému rodiči jako šipka jediná.
- *Závislý datový typ je implementací jiného datového typu.*
  I tuto závislost znázorňujeme šipkou s prázdnou trojúhelníkovou hlavou, avšak na rozdíl od šipky dědičnosti není implementační šipka plná, ale je čárkovaná.

Diagramy tříd v prostředí BlueJ

V začátečnických kurzech programování používám vývojové prostředí *BlueJ*, jehož integrální součástí je i editor zjednodušených UML diagramů. Nutno přiznat, že tato zjednodušení vedou často k přehlednějším diagramům, než jaké byste získali při použití profesionálních či profesionálně se tvářících editorů.

BlueJ především zobrazuje ve svých diagramech pouze názvy tříd. Jejich atributy ani metody nezobrazuje, ty si musíte najít ve zdrojovém kódu nebo dokumentaci dané třídy (tu však prostředí vytváří automaticky).

Diagram tříd z obrázku A.1 by vývojové prostředí *BlueJ* nakreslilo (po drobné změně uspořádání tříd) podle obrázku A.2.



Obrázek A.2
Diagram tříd nakreslený v prostředí BlueJ

PŘÍLOHA B

Seznam doporučené literatury

- **Co číst**
- **Rejstřík**

V následujícím přehledu literatury jsou uváděny pouze publikace a články, na něž se v textu odvolávám, spolu s několika klíčovými publikacemi z daných oblastí.

Co číst

Softwarové inženýrství

[1] SCHWALBE, Kathy. *Řízení projektů v IT*. Brno: Computer Press, © 2007. 720 s. ISBN 978-80-251-1526-8.

[2] McCONNELL, Steve. *Odhadování softwarových projektů*. Computer Press, © 2006. 320 s. ISBN 80-251-1240-3

Jazyk UML

Česky

[3] SCHMULLER, Joseph. *Myslíme v jazyku UML*. Praha: Grada, © 2001. 360 s. (Překlad [7]) ISBN 80-247-0029-8.

[4] FOWLER, Martin. *UML bez záhad*. Grada, © 2007. 192 s. (Překlad [5]) ISBN 978-80-247-2061-3

Anglicky

[5] FOWLER, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition*. Addison-Wesley Professional, © 2003. 192 s. ISBN 0-321-19368-7.

[6] GORMAN, Jason. *UML for Java Developers – Cass Diagrams*. PDF soubor s prezentací volně stáhnutelný z adresy http://www.parlezuml.com/tutorials/umlforjava/java_class_ba

[7] SCHMULLER, Joseph. *SAMS Teach Yourself UML in 24 Hours*. SAMS Publishing, © 1999. 422 s. (Přeloženo v [3]) ISBN 0-672-31636-6.

[8] BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language Reference Manual*. Addison-Wesley, © 1999. 550 s. ISBN 0-201-30998-X.

[9] BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language User Guide*. Addison-Wesley, © 1997. 482 s. ISBN 0-201-57168-4.

Návrhové vzory

Česky

[10] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů*. Grada, © 2003. 386 s. (Překlad [16]) ISBN 80-247-0302-5.

[11] ECKEL, Bruce. *Myslíme v jazyku C++ díl 2*. (10. kapitola se zabývá návrhovými vzory) Grada, © 2005. 396 s. (Překlad [14]) ISBN 0-201-30998-X.

[12] KRAVÁL, Ilja. *Design Patterns v OOP se zaměřením na JAVU, C# a Delphi*. Elektronická verze ke stažení na adrese **http://www.objects.cz**.

Anglicky

[13] ECKEL, Bruce. *Thinking in patterns*. Koncept knihy. HTML verzi lze stáhnout na adrese **http://www.bruceeckel.com**

[14] ECKEL, Bruce. *Thinking in C++ volume 2*. (10. kapitola se zabývá návrhovými vzory) Addison-Wesley, © 1995. 396 s. (Přeloženo v [11]) ISBN 0-201-30998-X. HTML verzi lze stáhnout na adrese **http://www.bruceeckel.com.**

[15] FREEMAN, Eric; FREEMAN, Elisabeth. *Head First Design Patterns*. O'Railly, © 2004. 640 s. ISBN 0-596-00712-4

[16] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, © 1995. 396 s. (Přeloženo v [10]) ISBN 0-201-30998-X.

[17] METSKER, Steven, John. *Design Patterns Java Workbook*. Addison-Wesley – Pearson Education, © 2002. 476 s. ISBN 0-201-74397-3.

[18] SHALLOWAY, Alan; TROTT, James R. *Design Patterns Explained 2nd edition*. Addison-Wesleey, © 2005. 430 s. ISBN 0-321-24714-0.

Webová sídla

[19] Stránka zaměřená na návrhové vzory s řadou demonstračních příkladů: http://home.earthlink.net/~huston2/dp/patterns.html

Objektově orientované programování

Česky

[20] BECK, Kent. *Programování řízené testy*. Grada, © 2004. 204 s. (Překlad [25]) ISBN 80-247-0901-5

[21] FOWLER, Martin. *Refaktoring. Zlepšení existujícího kódu*. Grada, © 2003. 394 s. (Překlad [26]) ISBN 80-247-0299-1

[22] KRAVÁL, Ilja. Design Patterns v OOP se zaměřením na Javu, C# a Delphi. 138 s. Vydána pouze elektronicky v září 2002. Není sice volně ke stažení, ale lze si o ni napsat na adresu objects@objetst.cz.

[23] MERUNKA, Vojtěch. *Datové modelování*. Alfa Publishing, © 2006, ISBN 80-86851-54-0

[24] PAGE-JONES, Meilir. *Základy objektově orientovaného návrhu v UML*. Grada, © 2001. 368 s. (Překlad [27]) ISBN 80-247-0210-X

Anglicky

[25] BECK, Kent. *Test Driven Development By Example*. Addison-Wesley, © 2003. 220 s. (Přeloženo v [20]) ISBN 0-321-14653-0

[26] FOWLER, Martin. *Refactoring. Improving the Design of Existing Code*. Addison--Wesley, © 2000. 430 s. (Přeloženo v [21]) ISBN 0-201-48567-2.

[27] PAGE-JONES, Meilir. *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley, © 2000. 458 s. (Přeloženo v [24]) ISBN 0-201-69946-X.

Java

Česky

[28] BLOCH, Joshua. *Java efektivně – 57 zásad softwarového experta*. Grada, © 2002. 230 s. (Překlad [41]) ISBN 80-247-0416-1

[29] ECKEL, Bruce. *Myslíme v jazyku Java – knihovna programátora.* Grada, © 2000. 432 s. (Překlad 1. poloviny 2. vydání [35]) ISBN 80-247-9010-6

[30] ECKEL, Bruce. *Myslíme v jazyku Java – knihovna zkušeného programátora.* Grada, © 2000. 470 s. (Překlad druhé poloviny 2. vydání [35]) ISBN 80-247-0027-1

[31] PECINOVSKÝ, Rudolf. *Java 5.0 – Novinky jazyka a upgrade aplikací.* CP Books, a. s., © 2005. 152 s. ISBN 80-251-0615-2

[32] PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java 5.0.* Grada, © 2004. 602 s. ISBN 80-247-0941-4

[33] SPELL, Brett. *Java Programujeme profesionálně.* Computer Press, © 2002. 1 022 s. (Překlad [40]) ISBN 80-7226-667-5

Anglicky

[34] GOSLIG. *Java Language Specification, 3rd edition.* Addison-Wesley Professional, © 2005, 688 s. ISBN 0-321-24678-0. (Kniha je k dispozici on-line na adrese **http://java.sun.com/docs/books/jls/index.html**, kde si můžete stáhnout i její PDF nebo HTML verzi.)

[35] ECKEL, Bruce. *Thinking in Java, 4th Edition.* Prentice Hall PTR, © 2005. 1 408 s. (2. vydání přeloženo v [29]+[30]) ISBN 0-131-87248-6. HTML verzi 3. vydání je možno stáhnout na adrese **http://www.bruceeckel.com**

[36] HORSTMANN, Cay. *Core Java, Volume I – Fundamentals.*

[37] HORSTMANN, Cay. *Core Java, Volume II – Advanced.*

[38] HORSTMANN, Cay. *Big Java.* John Wiley & Sons, © 2002. ISBN 0-471-40248-6

[39] KÖLLING, Michael: *Object First with Java and BlueJ.* Addison-Wesley, © 2005.

[40] SPELL, Brett. *Professional Java Programming.* Peer Information, © 2000, 1 111 s. (Přeloženo v [33]) ISBN 1-861003-82-X

[41] BLOCH, Joshua. *Effective Java – Programming Language Guide.* Addison-Wesley Professional, © 2001. 252 s. (Přeloženo v [28]) ISBN 0-201-31005-8

Jednotlivé články

Česky

[42] PECINOVSKÝ, Rudolf. *Výčtové typy.* Portál Java.cz. © 2005. Adresa: **http://www.java.cz/detail.do?articleId=1292**

Anglicky

[43] HAGGAR, Peter. *Double-checked locking and the Singleton pattern.* Portál IBM © 2002. **http://www-128.ibm.com/developerworks/java/library/j-dcl.html**

Rejstřík

Ch

I

J

K

L

M

N

O

P

U

V

W

Z

Další knihy z nabídky nakladatelství Computer Press

K1369 *320 stran*

Steve McConnell :

Odhadování softwarových projektů

Jak správně určit rozpočet, termíny a zdroje

499 Kč

Jeden z největších problémů softwarových projektů je překračování jejich odhadované doby trvání, rozpočtu i dalších zdrojů. Na vině přitom nemusí být jen neefektivní tým, ale také obtížně stanovitelný plán. Zkušený autor popisuje metody a možnosti, které vedou nejen k přesnějšímu odhadování softwarových projektů, ale také k lepší sledovatelnosti stavu projeku. Inspirujte se ve 120 užitečných tipech a ušetřete statisíce!

K1388 *720 stran + CD*

Kathy Schwalbe:

Řízení projektů v IT Kompletní průvodce

990 Kč

Řízení projektů je jedním z nejdůležitějších, přitom však nejobtížnějších manažerských úkolů – a v oblasti informačních technologií to platí dvojnásob. Proslulá kniha zkušené autorky se zabývá celým životním cyklem softwarového projektu a přináší neocenitelné rady, návody, postupy a znalosti z praktického vývoje, které zlevní, urychlí a zefektivní průběh a dodání vašich projektů.

K1325 *1400 stran*

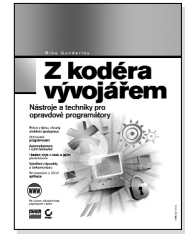
Christian Nagel:

C# 2005

Programujeme profesionálně

1390 Kč

Kniha zkušeného autorského týmu vás kompletně připraví na programování aplikací v jazyce C# a na práci v prostředí Microsoft Visual C# 2005. Dozvíte se i všechny podrobnosti o fungování architektury .NET. Kniha nevyžaduje žádné vstupní znalosti C# nebo platformy .NET. Na svých stránkách vám ukáže, jak programovat veškeré druhy aplikací, a otevře vám pohled na řadu souvisejících technologií.



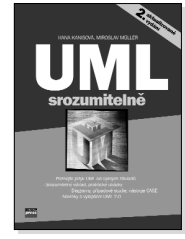
K1339 *288 stran*

Mike Gunderloy:

Z kodéra vývojářem Nástroje a techniky pro opravdové programátory

369 Kč

Kniha zkušeného autora je šitá přímo na míru programátorům a šéfům softwarových projektů. Po jejím přečtení se zvýší nejen kvalita vámi vytvářených programů, ale i vaše schopnost pracovat v týmu: v knize se dozvíte o defenzívním programování, o systémech pro správu i generování zdrojového kódu, testování, instalačních systémech, metodikách, dokumentaci, využívání výhod prostředí IDE, ale též o správném plánování projektů.



K1347 *176 stran*

Hana Kanisová, Miroslav Müller:

UML srozumitelně

2. aktualizované vydání

199 Kč

Aktualizovaná publikace seznamuje průvodcovským stylem a pomocí mnoha příkladů a diagramů s jazykem UML i souvisejícími technikami pro vývoj softwaru: od správy požadavků přes modelování a tvorbu případu užití až po mapování tříd do tabulek relačních databází. Použití popisovaných technik vždy ilustruje v nástroji CASE. Kniha obsahuje vysvětlení všech novinek a změn v UML 2.0.

K0702 *1040 stran*

Brett Spell:

Java

Programujeme profesionálně

890 Kč

Máte-li za sebou základy Javy, můžete v ní profesně vyspět s jednou ze světově nejúspěšnějších knih. V širokém záběru témat si osvojíte bohatou jazykovou výbavu Javy i tvorbu různého typu aplikací. Z obsahu: tvorba uživatelského rozhraní, zpracování událostí, použití moderních komponent... „Je to výborná kniha, kterou ocení všichni programátoři používající Javu." Chip 3/03