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.
```