Metodika Design Patterns First

ING. RUDOLF PECINOVSKÝ, CSC.
ICZ a.s., Praha 4, Na Hřebenech II 10
VŠE Praha, Fakulta informatiky a statistiky, katedra informačních technologií, Praha 3, nám. W. Churchilla 4
tel. + 420 603 330 090, e-mail: rudolf@pecinovsky.cz

ABSTRAKT
Naprostá většina učebnic a kurů objektově orientovaných jazyků zařazuje výklad dědičnosti vzápětí po seznámení s objekty. Studenti jsou pak jejími možnostmi opojeni a použijí ji i tam, kde je to nevhodné. Přednáška naznačí některé takové situace, ukáže, jak výklad dědičnosti nahradit výkladem vzoru *Dekorátor*, jehož použití bývá v takových situacích výhodnější. Poté vysvětlí, jak na základě znalostí vzoru dekorátor vyložit dědičnost a její vlastnosti tak, aby je studenti co nejlépe pochopili a správně si osvojili její používání.

1. ÚVOD

Objektově orientované programování je při vývoji rozsáhlejších aplikací již delší dobu naprosto dominantním paradigmatem. Do školních osnov však výuka OOP proniká pomalu a těžkopádně. Navíc, podíváme-li se do obsahu učebnic objektově programovaných jazyků a do osnov jejich kurzů, zjistíme, že většina z nich učí nejprve základní algoritmické konstrukce a objektové konstrukce uvádí spíš jako nadstavbu nad algoritmickým základem. Takto je pak chápou i jejich čtenáři a absolventi a z toho plyne i řada problémů, které mají při vývoji programů v praxi.

Na přelomu století se objevila metodika výuky *Object First*, která prosazuje zařazení práce s objekty hned na začátek kurzu. Tato metodika však soustřeďuje pouze na přeskupení výkladu objektů a tříd a přitom opomíjí, jak důležitou roli hraje v současném programování konstrukce *rozhraní* ([7], [16], [17], [18]). Navíc ignoruje neméně důležitý prvek, kterým jsou *návrhové vzory* (tamtéž). Tuto nevýhodu se snaží napravit metodika *Design Patterns First*, která zavádí rozhraní hned v prvních hodinách výuky a od začátku kurzu také studenty postupně seznamuje s různými návrhovými vzory (viz [22], [23], [24]).

Metodika však naznačuje pouze celkovou strategii výuky, tj. koncepci kurzů a rámcové uspořádání látky. V mnoha případech je však důležitá nejen strategie, ale i taktika, tj. způsob, jakým budeme jednotlivé konstrukce přednášet. Jak začátečníci, tak zkušenější programátoři mívají problémy s osvojením některých objektově orientovaných konstrukcí. V následujícím textu se pokusím ukázat, jak upravit styl výkladu tak, aby začátečníci tyto konstrukce lépe chápali a pokročilejší začali odstraňovat některé své špatné návyky plynoucí často z jejich špatného pochopení.

Soustředíme-li se na jazyk Java, který je ve vstupních kurzech objektově orientovaného programování nejpoužívanější[1], ukazuje se, že upravit nebo přímo změnit výklad a pojetí je vhodné u následujících témat:

- Vysvětlení pojmu objekt
- Rozdílné chápání tříd a objektů
- Rozhraní
- Konstrukce objektů a konstruktory
- Klíčové slovo this
- Dědičnost tříd

V následujícím textu se postupně zaměřím na každé z výše zmíněných témat a ukážu, jaká úprava výkladu se mi osvědčila při snaze zlepšit u studentů pochopení vykládané látky.

2. VŠECHNO JE OBJEKT

Učebnice a kurzy, s nimiž jsem měl možnost se setkat, se stavějí k výkladu termínu *objekt* dvojím způsobem:

- většina textů vysvětluje termín objekt na objektech z okolního reálného světa,
- zbylé tento termín nijak zvlášť nevysvětlují a předpokládají, že se jedná o pojem všeobecně známý, který žádné zvláštní vysvětlení nepotřebuje.

V obou případech však výklad zůstává u příkladů fyzických objektů a nijak se nezmiňuje o tom, že v programech vystupují jako objekty i abstraktní pojmy (krása, velikost, směr, připojení, přerušení, výpočet…). Když se potom studenti s takovýmito objekty ve své další praxi setkají, často nějakou dobu zápasí s akceptací toho, že by abstraktní pojem mohl být v programu reprezentován jako objekt.

V této souvislosti se ukázalo výhodné vysvětlit studentům, že v objektovém programování se za objekt považuje vše, co je možno označit podstatným jménem. Do této kategorie pak automaticky zapadají i výše

[1] S podobnými problémy se ale můžeme setkat u učebnic a kurzů prakticky všech objektových programovacích jazyků.

zmínění abstraktní termíny. V tomto okamžiku sice budou někteří posluchači zmateni, protože mají problémy s pochopením abstraktního pojmu jako objektu. Zde jim proto vysvětlíme, že v programu je každý pojem zastupován skupinou údajů, která daný pojem charakterizuje (krása může mít svoji stupnici, směr může být učen názvem světové strany nebo odchylkou od předem daného směru, připojení má definovaný protějšek, protokol a některé další parametry, atd.). Z hlediska programu je pak objektem právě tato skupina údajů.

Studenti velmi rychle pochopí, že obdobně, jako definují skupinu údajů reprezentující auto, židli či jiný objekt reálného světa, mohou definovat i údaje, které budou charakterizovat abstraktní pojmy jako krása, směr apod. Aby jim tato možnost přešla co nejrychleji do krve, je nutno objekty tohoto typu používat od samého začátku kurzu. Jako vhodní kandidáti takovýchto abstraktních objektů se zde jeví některé charakteristiky grafických objektů – např. barvy nebo směry.

Nestačí však studentům pouze toto chápání objektů oznámit, ale je třeba, aby se objekty tohoto druhy vyskytovaly hned od počátku kurzu v příkladech, aby si na tento druh objektů mohli studenti zvyknout.

3. TŘÍDY VERSUS OBJEKTY

V klasicky koncipovaných učebnicích jsou třídy často vysvětlovány jako jakési abstrakce popisující vlastnosti skupin objektů, které označujeme jako jejich instance (několik z uvedených učebnic svorně charakterizuje třídy výrokem typu: *classes serve as blueprints or templates for the objects that the program uses*). Někdo pak k tomuto základnímu vysvětlení ještě připojí dodatek, že třídy můžeme chápat jako továrny na objekty a jsou schopny tyto objekty na požádání vytvořit.

Zkušenost však ukazuje, že po takto pojatém výkladu řada studentů nějakou dobu zápasí s pochopením rozdílu mezi třídami a objekty a zejména pak s významem statických atributů a metod.

Ukázalo se, že v tomto směru pomáhá, když studentům vysvětlíme, že i třída je objekt (když je objektem vše, co můžeme označit podstatným jménem, musíme jako objekt akceptovat i třídu). Vysvětlíme jim, že třída je speciální druh objektu, který má oproti ostatním objektům dvě zvláštní vlastnosti:

- Třídy jsou jediné objekty, jež umějí vytvářet jiné objekty. Třídou vytvořené objekty pak označujeme jako instance dané třídy. Pokud některý z pokročilejších studentů namítne, že i jiné objekty umějí vytvářet objekty, vysvětlíme mu, že tyto objekty musí o vytvoření nového objektu nejprve požádat nějakou třídu a teprve takto vytvořený objekt pak mohou vydávat za svůj výtvor.
- Objekt třídy není instancí žádné třídy tříd (hovořím o jazyku Java, v jiných jazycích – např. ve Smalltalku – to může být jinak). V programu jej musíme vždy oslovovat přímo. V Javě je v některých situacích objekt třídy zastupován objektem, jenž je instancí třídy `Class`.

Je až s podivem, jak tento drobný posun ve výkladu přispívá u řady studentů k pochopení pojmu třídy a odbourání problémů s jeho následným použitím.

Tomuto vysvětlení nahrává i prostředí BlueJ, které ve vstupních kurzech programování používáme. V tomto prostředí se totiž s třídami pracuje téměř stejně jako s objekty. Třídy i jejich instance jsou reprezentovány obdélníky, přičemž zprávy, které jim je možno posílat, tj. metody, které je možno volat, jsou uvedeny v jejich místní nabídce. Jediným rozdílem mezi třídami a objekty je v prvním přiblížení to, že obdélníky zastupující třídy jsou umístěny v diagramu tříd, kdežto obdélníky zastupující objekty jsou umístěny v zásobníku odkazů. Studentům proto připadá chápání tříd jako speciálního druhu objektů naprosto přirozené.

Zavedení třídy jako speciálního druhu objektu nám pomáhá i při vysvětlování dalších témat:

- Student pak nemají problémy s pochopením rozdílu mezi atributy a metodami třídy a instance, a můžeme je proto bez problému používat téměř od začátku kurzu.
- Studenti o něco snadněji chápou i pravidla pro zavádění tříd a v dalším průběhu kurzu i definici dědičnosti.

4. KONSTRUKTOR A PARAMETR `this`

Dalším tématem, s jehož pochopením a následným použitím mívají studenti občas problémy, jsou konstruktory a použití klíčového slova `this`. Prakticky všechny učebnice vycházejí z původního popisu v [28], v němž bylo uvedeno: *"Constructor is identified by having the same name as its class."* Tento výklad nijak nerozebírá, zda konstruktor je či není metoda.

V pojetí konstruktoru se pak učebnice liší. [1] a [9] např. vysvětlují, že konstruktor není metoda a nesmí proto nic vracet. Při takto pojatém výkladu je pak za vrácení odkazu na vytvořenou instanci zodpovědný operátor new. (Ten sice v Javě není ve skutečnosti operátorem — [8] jej ve výčtu operátorů neuvádí — ale řada autorů jej za operátor označuje.)

Většina autorů však definuje konstruktory jako speciální metody, které se jmenují stejně jako třída, v níž jsou definovány, a nemají explicitně definován typ návratové hodnoty. Předávání zodpovědnosti za inicializaci proměnné mezi konstruktory prostřednictvím konstrukce `this(…)` pak vysvětlují jako speciální obrat. To, proč reflexe udává pro konstruktor identifikátor `<init>` a proč jej tak (většinou) označuje i debugger, pak většinou

vůbec nevysvětlují (případný výskyt ignorují – viz [4], str. 927), a pokud se o této skutečnosti zmíní (viz např. [10]), tak nijak nekomentují to, že před tím tvrdili něco jiného.

Zamyslíme-li se nad syntaxí definic konstruktorů, zjistíme, že konstruktory můžeme stejně dobře vydávat za metody, jejichž názvem je prázdný řetězec a které vracejí odkaz na instanci své třídy. Toto pojetí je navíc bližší skutečné implementaci. Ukázalo se, že i při výkladu je výhodnější se přiblížit se tomu, jak to skutečně ve virtuálním stroji probíhá, a prezentovat konstruktory jako metody, které mají některé zvláštní vlastnosti:

- Konstruktory se interně jmenují `<init>`, což odporuje pravidlům pro identifikátory, a proto je v programu deklarujeme jako bezejmenné metody, nebo přesněji jako metody, jejichž identifikátorem je prázdný řetězec.
- Konstruktory lze použít *pouze* k inicializaci čerstvě alokované paměti, a proto je může volat pouze „operátor“ `new` nebo jiný konstruktor téže třídy, který tak deleguje svoji zodpovědnost za vytvoření nové instance na volaný konstruktor. Toto „delegující volání“ musí být prvním příkazem volajícího konstruktoru proto, aby se zaručilo, že v případném dalším těle delegujícího konstruktoru se bude pracovat s plnohodnotným, inicializovaným objektem.
- Konstruktor musí vždy vracet odkaz na vytvořenou instanci. Tento odkaz obdrží od toho, kdo jej volá (tj. od „operátoru“ `new` nebo delegujícího konstruktoru), jako hodnotu skrytého parametru `this`. Protože je předem známo, jakou hodnotu konstruktor vrací, příslušný příkaz `return this;` se v definici konstruktoru neuvádí, a překladač jej za nás doplní.

Při takto pojatém výkladu konstruktoru je pak pro studenty mnohem logičtější i výklad dalších probíraných syntaktických obratů, které vysvětlujeme následovně:

- Konstrukce nového objektu probíhá ve dvou fázích:
  - V první fázi se zavolá operátor `new`, kterému se předá jako parametr název třídy, jejíž instanci se chystáme vytvořit. Operátor z něj odvodí velikost paměti, kterou má pro daný objekt vyhradit (a vynulovat) a současně i některé informace, které potřebuje pro správnou alokaci objektu. (Později jim vysvětlíme, že tímto dodatkem je doplnění odkazu na tabulku virtuálních metod.) Součástí alokace paměti je i její vynulování a inicializace konstant vyhodnotitelných v době překladu.
  - V druhé fázi se zavolá bezejmenná metoda – konstruktor, které operátor `new` předá ve skrytém parametru `this` odkaz na právě alokovanou paměť, a za něj se pak případně doplní hodnoty ostatních parametrů. Úkolem konstruktoru je tuto paměť inicializovat a uvést ji tak do stavu, v němž bude správně reprezentovat příslušný objekt.

  Průběh operace bychom mohli zdůraznit rozepsáním výrazu pro konstrukci objektu do dvou řádků:

  ```
  new NázevTřídy     //volání operátoru new
  (/*parametry*/)    //volání konstruktoru
  ```
- Jak jsme řekli, konstruktor je možno použít pouze k inicializaci čerstvě alokované paměti. Pokud je proto volán jiným konstruktorem, musí být toto volání prvním příkazem jeho těla, před nímž nesmí předcházet žádný jiný příkaz, a dokonce ani otevření bloku příkazů (tj. otevírací složené závorky).
- Předává-li jeden konstruktor zodpovědnost za prvotní inicializaci objektu jinému konstruktoru (tj. deleguje-li na něj svoji zodpovědnost), tj. volá-li jeden konstruktor jiný konstruktor, musí toto volání kvalifikovat parametrem `this` obdobně, jako kdyby se např. obracel na svůj atribut nebo volal svoji metodu. Při volání konstruktoru se však za `this` nepíše tečka, takže místo volání

  ```
  this.(/*parametry*/)
  ```

  píšeme jenom

  ```
  this(/*parametry*/)
  ```

Při takto pojatém výkladu si studenti snáze zapamatují syntaktickou konstrukci pro předání zodpovědnosti za prvotní inicializaci na druhý konstruktor a současně je jim pak hned jasné, co se jim snaží debugger sdělit uvedením identifikátoru `<init>`.

Na takto pojatý výklad pak logicky naváže i výklad funkce statických a instančních bloků, zařazení inicializačních deklarací atributů do těla konstruktoru a nakonec i výklad konstrukce potomka, kterému musí předcházet konstrukce předka. Všechno do sebe zkrátka logicky zapadá.

Při výše popsaném výkladu funkce konstruktoru někteří studenti občas namítnou, že objekt vlastně nevytváří konstruktor, ale operátor `new`, který alokuje potřebnou paměť. Zde se jich s oblibou ptám, kdo podle nich vytváří např. džbánek. Všichni svorně odpovědí, že hrnčíř. Já jim pak vysvětlím, že paměť alokovaná operátorem `new` je z hlediska programu obdobou hlíny, kterou dostane hrnčíř. Vlastní vytvarování džbánku je pak úkolem hrnčíře obdobně, jako je inicializace objektu úkolem konstruktoru.

5. ROZHRANÍ

Řada učebnic vysvětluje konstrukci `interface` jako náhražku násobné dědičnosti. Funkce této konstrukce je však mnohem důležitější a v současném programování přímo klíčová. V roce 1995 formulovali autoři [7] zásadu

Programming to an Interface, not an Implementation (programování proti rozhraní, ne proti implementaci), která se stala nosným motem současného programování. Konstrukce `interface` v Javě a dalších jazycích (C#, Groovy, Scala, Visual Basic.NET, …) není náhražkou čehosi, ale je to naopak klíčová konstrukce, která umožňuje formálně deklarovat rozhraní oddělené od jakékoliv implementace a zvýšit tak pravděpodobnost toho, že uživatel daného objektu bude opravdu využívat pouze rozhraní objektu a nebude zbytečně využívat jeho implementačních detailů.

Většina učebnic vyjmenovaných v úvodu probírá rozhraní až v závěru části věnované programovým konstrukcím a ani v dalším výkladu se nijak nesnaží naučit své čtenáře něco více než implementovat nějaké existující rozhraní. Pouze [9] a [11] se pokouší předvést, kdy je vhodné zavést do programu vlastní rozhraní.

Metodika *Design Patterns Fist* doporučuje zařadit výklad rozhraní hned do prvních hodin výuky. Rozhraní dokonce vysvětluje a učí je používat ještě před tím, než studenti začnou psát svůj vlastní kód ([16]). Poskytuje tak studentům dostatek času a příležitostí, aby se s rozhraním naučili pracovat.

Kromě toho se snaží naučit studenty nejenom implementovat rozhraní, které již existuje, ale ukazuje jim na příkladech řadu situací, kdy je vhodné zařadit rozhraní do vlastního návrhu. Kromě toho zadává studentům úlohy, jejichž součástí je i definice vlastních rozhraní. Na konci kurzu pak studenti dokáží mnohem lépe odhadnout situace, které je nejvýhodnější řešit zavedením vlastního rozhraní, a dokáží potřebná rozhraní navrhnout a začlenit do projektu.

6. DĚDĚNÍ

Problém s výukou dědičnosti spočívá především v tom, že se většinou učí příliš brzy – některé učebnice ji dokonce vysvětlují prakticky současně s vysvětlením pojmu objekt a třída. Při takto časném výkladu dědičnosti si pak studenti často zapamatují především to, že si díky dědičnosti mohou ušetřit psaní nějakého kódu.

Pominu nyní otázku, že chceme-li, aby si studenti co nejlépe osvojili objektové paradigma, měli bychom dědičnost vysvětlovat až poté, kdy studentům vysvětlíme konstrukt `interface` a naučíme je rozhraní implementovat. Tato problematika byla již rozebírána v článcích [22], [21], [19] seznamujících s metodikou výuka *Design Patterns First*. Přístup, při němž se vykládá nejprve rozhraní a teprve pak dědičnost tříd, najdeme z uvedených učebnic pouze v [9] a [11] a [29]. Nicméně i v uvedených případech je dědičnost vysvětlována prakticky vzápětí po výkladu rozhraní a studenti navíc nedostanou příležitost řešit úlohy, v nichž je použití dědičnosti spíše na obtíž.

Vraťme se ale k dědičnosti tříd. Všechny jmenované učebnice zmiňují, že instance potomka by měly být pouze speciálními případy instancí předka. Ne všechny však kladou na toto pravidlo stejný důraz. Většina je pouze na počátku výkladu zmíní a v dalším textu již pouze ukazuje, jak lze využitím dědičnosti ušetřit psaní kódu. To také bývá hlavní informace, kterou si studenti o dědičnosti zapamatují. Jedním z důvodů je i to, že učebnice neuvádějí příklady špatně navržené dědičnosti, aby čtenáře včas varovaly před špatným návrhem.

Při výkladu dědičnosti tříd bychom si navíc měli uvědomit, že se v ní promítají tři aspekty dědičnosti ([13]):

- dědičnost typů,
- dědičnost implementace a
- přirozeně chápaná dědičnost.

Protože převážná většina kurzů a učebnic tyto tři aspekty nezmiňuje, dovolím si je připomenout:

- Dědění typů bychom mohli označit také jako děděné rozhraní. Potomek při něm od rodiče přebírá veškerou jeho signaturu i kontrakt. V důsledku toho mohou instance potomka vystupovat kdykoliv v roli instance předka. Při dědění tříd však překladač zabezpečí pouze převzetí (zdědění) signatury předka. O dodržení kontraktu se musí postarat programátor.
- Dědění implementace hovoří o tom, že potomek přebírá od předka veškerou jeho implementaci (to zajistí překladač). Implementaci, která se potomku nehodí, překryje implementací vlastní a případně doplní další. Tady právě hrozí největší nebezpečí, že při překrývání resp. doplňování dalších členů programátor poruší kontrakt předka.
- Přirozeně chápaná dědičnost hovoří o tom, kdy potomka chápeme jako speciální případ předka. Tady můžeme narazit v případě, kdy je toto chápání v rozporu s implementací a může proto někdy u programátora vyvolat falešné představy a očekávání. Příkladem ze standardní knihovny Javy jsou např. mapy, které sice intuitivně chápeme jako množiny dvojic (klíč, hodnota), ale nejsou tak implementovány.

Ze jmenovaných učebnic tuto skutečnost explicitně zmiňuje pouze [1]. Při čtení ostatních si tento fakt musíme odvodit sami. Dokud tyto tři aspekty dědičnosti řádně neprobereme a nevysvětlíme studentům, že ke správné funkci programu je nutné, aby byly všechny (nebo alespoň první dva) v souladu, můžeme opakovaně očekávat, že si studenti neuvědomí případný rozpor ve svých projektech a návrzích. Z toho vyplývají i následující doporučení:

1. Odložit výkladu dědičnosti na co nejpozdější dobu

Především bychom měli odložit výklad dědičnosti na co nejpozdější dobu, abychom měli dostatek prostoru na procvičení použití rozhraní. Studenti by se v této době měli naučit nejenom rozhraní implementovat, ale také správně rozpoznat jistou množinu typických situací, v nichž je výhodné navrhnout vlastní rozhraní.

V tomto období je také vhodné seznámit studenty s návrhovým vzorem *Dekorátor* a připravit s nimi alespoň jeden projekt, v němž je použití tohoto vzoru mnohem výhodnější než použití klasické dědičnosti tříd. Představení tohoto vzoru má dva důvody:

- Pokročilejším studentům, kteří se již dědičnost tříd ovládají, ukážeme, že použití dědičnosti není vždy optimálním řešením, a tím je nepřímo přimějeme k pozornějšímu sledování výkladu.
- Připravíme si půdu pro následující výklad dědičnosti tříd.

2. Vysvětlit dědičnost jako automatizovanou realizaci vzoru *Dekorátor*

V další etapě studentům vysvětlíme, že při dědění tříd probíhá dědění implementace způsobem, který je velmi podobný tomu, s nímž se seznámili při aplikaci návrhového vzoru *Dekorátor*. Jedná se vlastně o aplikaci tohoto vzoru, při níž s přebíraným (dekorovaným) objektem přebíráme (dědíme) současně i jeho rozhraní. Pro dekorovaný objekt překladač připraví atribut nazvaný `super` a současně zabezpečí automatické delegování volaných metod na metody dekorovaného objektu.

Objekt `super` označíme jako *rodičovský podobjekt* daného objektu. Na rozdíl od dekorátoru jej však konstruktor nepřebírá jako parametr, ale vytvoří si jej sám zavoláním jeho „bezejmenné metody“ – konstruktoru prostřednictvím příkazu

```
super (/*prametry*/)
```

Přidáme přitom upozornění, že rodičovský podobjekt musí být vytvořen jako první, a proto konstruktor, který nedeleguje zodpovědnost za inicializaci vytvářené proměnné na některého ze svých kolegů, musí začínat tímto příkazem. Jedinou výjimkou je situace, kdy bychom chtěli volat bezparametrický konstruktor předka – jeho volání nemusíme uvádět explicitně, protože je za nás překladač doplní sám. Proto jsme jej také doposud nemuseli používat – všechny naše dosavadní třídy byly totiž potomky třídy `Object`, která má pouze bezparametrický konstruktor. Současně ukážeme, že tento příkaz můžeme do kterékoliv z našich dosavadních tříd doplnit.

Ukazuje se, že takto koncipovaný výklad je pro řadu studentů mnohem pochopitelnější než výklad klasický. Řada programátorů, kteří k nám chodí na zdokonalovací kurzy, nám po tomto výkladu dokonce prozradila, že teprve nyní pochopili některé vlastnosti dědičnosti, které jim byly doposud nejasné.

Na závěr výkladu je třeba studentům znovu připomenout, že překladač se postará pouze a zdědění implementace a signatury. O to, aby kontrakt potomka nekolidovalo s kontraktem předka, se musí postarat programátor.

7. APLIKACE V PRAXI

Výklad podle metodiky *Design Patterns First* dodržující všechna výše zmíněná doporučení byl použit v učebnici [16]. Tato učebnice je prvním dílem chystaného kompletu a je v ní zcela vynechán výklad dědičnosti. Ten bude zahrnut až do druhého dílu.

Naopak je zde ukázáno „správné“ řešení úlohy, kterou řada začátečníků prošedších klasicky koncipovanými kurzy OOP začne okamžitě řešit za pomoci dědičnosti, přestože právě v tomto příkladu je použití dědičnosti poněkud nevhodné a zbytečně komplikuje celé řešení.

Učebnice byla koncipována jako příručka pro žáky středních škol, případně pro žáky vyšších ročníků základních škol. Prošla rozsáhlou oponenturou, přičemž mezi lektory byli začínající i pokročilí programátoři a tři učitelé programování na různých stupních škol.

8. ZÁVER

Článek stručně seznámil se zásadami výuky podle metodiky *Design Patterns Fist*. Současně poukázal na běžné způsoby výkladu některých konstrukcí objektově orientovaného programování v jazyku Java, jejichž zvládnutí působí občas některým studentům problémy a naznačil možné úpravy, které podle zkušeností vedou k lepšímu pochopení daných konstrukcí a v následné etapě i k lepšímu návrhu studentských programů.

Doporučil při výkladu objektů zahrnout i objekty představující abstraktní pojmy. Následně při výkladu tříd pak doporučuje prezentovat třídu také jako objekt, i když s některými speciálními vlastnostmi. Při výkladu konstruktorů pak radil pojmout konstruktor jako metodu, jejímž názvem je prázdný řetězec, a ukázal, jak se po této drobné změně stane řada dalších konstrukcí logickým pokračováním. Na závěr se zabýval dědičností. Navrhl posunout výklad dědičnosti tříd až za výklad rozhraní, při němž se (mimo jiné) studenti seznámí i s návrhovým vzorem *Dekorátor*. Tento vzor pak může posloužit k přiblížení funkce dědičnosti tříd a usnadní pochopení některých konstrukcí, jakými je překrývání metod nebo nutnost volat konstruktor předka. Při výkladu dědičnosti tříd navíc doporučil seznámit studenty s třemi aspekty dědičnosti a nutností jejich vzájemného souladu.

LITERATURA

[1] ARNOLD K., GOSLING J., HOLMES D.: *The Java™ Programming Language, Fourth Edition*. Addison Wesley Professional 2005. ISBN 0-321-34980-6.

[2] BARNES D., KÖLLING M.: *Objects First With Java: A Practical Introduction Using BlueJ (2nd Edition)*. Prentice Hall 2004. ISBN 978-0-131-24933-2.

[3] BERGIN, Joseph: Fourteen Pedagogical Patterns. *Proceedings of Fifth European Conference on Pattern Languages of Programs.* (EuroPLoP™ 2000) Irsee 2000.

[4] DEITEL H. M., DEITEL P. J.: *Java How to Program, 7th Edition*. Prentice Hall 2007, ISBN 978-0-132-22220-4.

[5] ECKEL B.: *Thinking in Java (3rd Edition)*. Prentice Hall 2007, ISBN 978-0-131-00287-2-6.

[6] FAIN Y.: *Java Programming for Kids, Parents and Grandparents.* Smart Data Processing 2004. ISBN 0-9718439-5-3. http://www.csd.abdn.ac.uk/~tnorman/teaching/CS1014/information/JavaKid8x11.pdf

[7] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. ISBN 0-201-30998-X.

[8] GOSLING J., JOY B., STEELE G., BRACHA G.: *Java™ Language Specification, Third Edition.* Addison Wesley 2005. ISBN 978-0-321-24678-3. http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[9] HORSTMAN C. S.: *Big Java (3rd Edition)*. John Wiley and Sons 2007. ISBN: 978-0-470-10554-2.

[10] HORSTMAN C. S., CORNELL G.: *Core Java™, Volume I – Fundamentals (8th Edition)*. Prentice Hall PTR 2007. ISBN 978-0-132-35476-9

[11] HORSTMAN C. S.: *Java Concepts for Java 5 and 6.* John Wiley and Sons 2007. ISBN 978-0-470-10555-9.

[12] HORTON I.: *Beginning Java 2*. Wrox 2002. ISBN 978-0-76454-365-4.

[13] LALOND W., PUGH J.: *Subclassing ≠ Subtyping ≠ IsA*. Journal of Object-Oriented Progrtamming. Vol. 3, No. 5, 1991.

[14] LIANG Y. D.: *Introduction to Java Programming: Comprehensive Version (6th Edition).* Prentice Hall 2006. ISBN 978-0-132-22158-0.

[15] MORELLI R., WALDE R.: *Java, Java, Java, Object-Oriented Problem Solving (3rd Edition)*. Prentice Hall 2006, ISBN 978-0-131-47434-5.

[16] PECINOVSKÝ R.: *OOP – Naučte se myslet a programovat objektově.* Computer Press 2010. ISBN 978-80-251-2126-9.

[17] PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java*. Grada 2008. 576 s. ISBN 978-80-247-2653-3

[18] PECINOVSKÝ, Rudolf. *Návrhové vzory – 33 vzorových postupů pro objektové programování*. Computer Press, © 2007, 528 s. ISBN 978-80-251-1582-4.

[19] PECINOVSKÝ R.: *Early Introduction of Inheritance Considered Harmful*. Objekty 2009, Hradec Králové.

[20] PECINOVSKÝ R. *Using the methodology Design Patterns First by prototype testing with a user.* Proceedings of IMEM 2009, Spišská Kapitula.

[21] PECINOVSKÝ R., PAVLÍČKOVÁ J.: *Order of explanation should be Interface – Abstract classes – Overriding.* Proceedings of 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'2007). Dundee, ACM Press.

[22] PECINOVSKÝ R., PAVLÍČKOVÁ J., PAVLÍČEK L.: *Let's Modify the Objects-First Approach into Design-Patterns-First*. Proceedings of 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'2006). Bologna, ACM Press, ISBN 1-59593-346-8.

[23] PECINOVSKÝ Rudolf: Výuka programování podle metodiky Design Patterns First. *Tvorba softwaru 2006 – sborník přednášek*. ISBN 80-248-1082-4.

[24] PECINOVSKÝ Rudolf: Aplikace metodiky Design Patterns First. *Objekty 2006 – sborník příspěvků desátého ročníku konference*, ČZU, Praha 2006. ISBN 80-213-1568-7.

[25] SCHILDT H.: *Java: A Beginner's Guide, Third Edition*. McGraw-Hill Osborne Media 2005. ISBN 0-07-223189-0

[26] SCHILDT H.: *Java: The Complete Reference, J2SE 5 Edition*. McGraw-Hill Osborne Media 2004. ISBN 978-0-07-223073-4.

[27] SIERRA K., BARTES B.: *Head First Java, 2nd Edition*. O'Reilly Media 2009. ISBN 978-0-596-00920-5.

[28] STRUSTRUP B.: *The C++ Programming Language, 2nd Edition.* Addison-Wesley Publishing Company 1991. ISBN 0-201-53992-6.

[29] ZAKHOUR S., HOMMEL S., ROYAL J., RABINOVITCH I., RISSER T., HOEBER M.: *The Java Tutorial: A Short Course on the Basics, 4th Edition.* Prentice Hall PTR 2006. ISBN 978-0-321-33420-6.

[30] ZUKOWSKI J.: *Mastering Java 2.* Sybex 2002. ISBN 978-0-7821-4022-4.