VÝUKA PROGRAMOVÁNÍ PODLE METODIKY DESIGN PATTERNS FIRST

Rudolf Pecinovský
Amaio Technologies, Inc., Třebohostická 14, 100 00 Praha 10,
rudolf@pecinovsky.cz

ABSTRAKT
V současné době převládá ve světě ve výuce programování na vysokých školách metodika *Objects First*. Řada autorů však poukazuje na to, že při tomto přístupu se řada vyučujících příliš dlouho soustřeďuje na návrh jednoduchých aplikací, v nichž navíc neuplatňují řadu klíčových principů moderního programování, mezi nimi především návrhové vzory. Příspěvek seznamuje s upravenou verzí metodiky *Objects First* nazvanou *Design Patterns First*. Ukazuje, že při vhodně zvolených příkladech je možné seznamovat žáky s návrhovými vzory a řadou dalších principů moderního programování od samého počátku výuky.

1 ÚVOD

Mají-li si žáci nějakou látku důkladně osvojit, není možné začít s jejím výkladem až někdy před koncem příslušného kurzu. Tato zásada je obzvlášť důležitá u předmětů, u nichž nestačí se látku pouze naučit, ale je nutné si ji postupně osvojit vyřešením řady praktických úloh. Do této kategorie předmětů patří i výuka programování. To byl také důvod zavedení metodiky *Objects First*, která bere v úvahu, že v současné době se naprostá většina aplikací programuje objektově a chce studenty na toto programové paradigma co nejlépe připravit. V současné době je tato metodika převládající metodikou univerzitních vstupních kurzů programování a postupně se prosazuje i na řadě škol středních.

Přestože tento přístup převládá, mnozí cítí, že je na něm ještě stále co vylepšovat; že nestačí začít pouze výkladem objektů a práce s nimi, ale že by bylo třeba zahrnout do výuky i další aspekty moderního programování. Na přelomu století se proto skupina vysokoškolských učitelů dohodla, že budou pořádat pravidelné semináře nazvané *„Killer Examples" for Design Patterns and Objects First*, na kterých se pokusí představit tzv. „killer examples", což by měly být právě příklady, které jsou na jednu stranu dostatečně jednouché, aby je bylo možno použít i ve vstupních kurzech programování, avšak na druhou stranu budou dostatečně „složité", aby se na nich dalo přirozeně demonstrovat použití návrhových vzorů. Příklady, které názorně demonstrují užitečnost návrhových vzorů a jejich použití.

Prozatím však (alespoň v mně dostupné literatuře) nikdo neuvažoval o tom, že by návrhové vzory bylo možno přednášet od samého počátku vstupních kurzů. Chtěl bych zde proto ukázat, že při vhodném výběru příkladu to možné je.

2 ZÁSADY SPRÁVNÉHO PROGRAMOVÁNÍ

Opomíjení se však netýká pouze návrhových vzorů. Stejně jsou postiženy o další z klíčových zásad, kterými se řídí moderní programování, a které je vhodné vštěpovat studentům od samého počátku výuky. Připomeňme si nejdůležitější z nich (možná, že byste do seznamu přidali i další):

- Neustále dbát na důsledné zapouzdření
- Zapouzdřit části kódu, které by se mohly měnit.
- Programovat proti rozhraní a ne proti implementaci.
- 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 (návrhu řízený odpovědnostmi – responsibility driven design).
- Minimalizovat vzájemnou provázanost (coupling) entit.
- Vyhýbat se duplicitám kódu.
- Využívat návrhové vzory.

Jakkoliv jsou však tyto zásady pro správný návrh kódu klíčové, ve výuce programování se s výkladem (a tím spíš prosazováním) značné části z nich většinou nesetkáme. Jediná zásada, jejíž dodržování bývá vyžadováno prakticky všude, je zásada důsledného zapouzdření. Ostatní zásady většina učebnic a kurzů většinou ani nezmíní. Máme-li však studenty dobře připravit na návrh komplexních aplikací, měli bychom jim dodržování těchto zásada vštěpovat již od samého počátku výuky a ne až v některých nadstavbových kurzech anebo dokonce spoléhat na to, že se je naučí dodržovat sami.

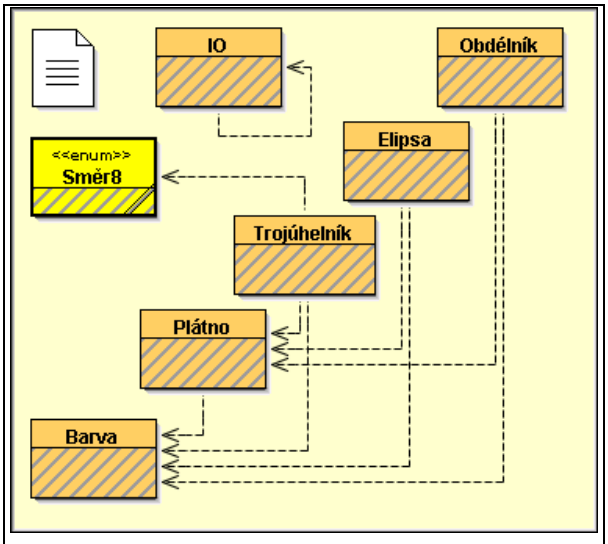
3 METODIKA *„DESIGN PATTERNS FIRST“*

V následující části bych chtěl předvést možnou náplně prvních několika cvičení a ukázat příklady, na nichž studenty seznamujeme se zásadami moderního programování včetně používání návrhových vzorů.

Jednou z důležitých vlastností metodiky je to, že studenti nemají většinou za úkol vytvářet nějaké samostatné, a proto nutně nesmírně jednoduché programy, ale že mají rozšiřovat funkčnost nějakého rozsáhlého (alespoň pro ně), předem připraveného projektu. Náplň výuky se tak mnohem více blíží potřebám praxe, protože převážnou většinou programátorských úkolů je právě rozšíření funkčnosti nějakého stávajícího projektu vytvořeného navíc většinou někým jiným.

3.1 Úvodní hodina

Na úvodní hodině se studenti seznámí s vývojovým prostředím (používáme *BlueJ*), v němž si pak otevřou předem připravený projekt obsahující 8 tříd různých vlastností. V tomto projektu se seznámí s třídami, objekty, posíláním zpráv a dalšími stavebními kameny objektových programů (viz obr. 1). Při té příležitosti se dozví, že v programech považujeme za objekt opravdu všechno včetně toho, co bychom v běžném životě za objekt nikdy neprohlásili – např. vlastnosti.



Obr. 1: *Diagram tříd úvodního projektu*

Při příležitosti předvádění rozdílného chování některých tříd jim prozradíme, že v objektovém programování používáme ekvivalenty matematických vzorců, které definují osvědčené postupy, jak navrhnout program řešící určitý často se vyskytující problém. V naší úloze jsou tímto „problémem“ např. počty instancí jednotlivých tříd:

- Třída `IO` slouží pouze jako přepravka pro často používané metody, které nejsou reakcemi nějaké instance na jí posílanou zprávu. Je proto vybudována podle vzoru *Knihovní třída*.
- Třída `Plátno` musí mít nejvýše jednu instanci, aby bylo zaručeno, že se budou všechny obrazce kreslit na jednom plátně. Při její konstrukci byl proto použit návrhový vzor *Jedináček*. O její instanci nežádáme konstruktor, ale metodu konstruovanou podle návrhového vzoru *Jednoduchá tovární metoda*.

- Třída `Směr8` definuje předem známý počet předem známých instancí – je proto vybudována jako *Výčtový typ*.
- Třída `Barva` sice neomezuje počet vytvořených instancí, ale hlídá si, aby nevznikly dvě instance reprezentující stejnou barvu. Při její konstrukci byl proto použit vzor *Originál*.
- Třídy `Elipsa`, `Obdélník` a `Trojúhelník` jsou běžnými objektovými typy a tvorbu svých instancí nijak neovlivňují.

Účelem tohoto seznámení je zanést do mysli studentů informaci o tom, že i v programování existují ekvivalenty matematických vzorců. Informaci, na kterou pak hned v další hodině navážeme.

Současně jim prozradíme, že třída `IO` je jednou z možných implementací návrhového vzoru *Fasáda*, jehož účelem je zjednodušit komplikovaný systém a umožnit uživatelům (programátorům) požadujícím jednoduché operace práci se zdánlivě jednodušším systémem.

Aby studenti zažili vytváření instancí a posílání zpráv, dostanou za úkol definovat testovací třídu, která vytvoří obrazec a po potvrzení uživatele jej pak přesune na nové místo.

3.2 První „textový“ program – interface, návrhový vzor *Služebník*

Na druhé hodině nejprve vytvoříme prázdnou třídu `Kruh`, kterou pak doplníme o konstruktor kreslící kruh a poté doplníme konstruktory umožňující zadat polohu, rozměr a barvu vytvořeného kruhu. Přejmenujeme ji na `Světlo` a doplníme postupně metody `rozsviť()`, `zhasni()` a `blikni()`.

Nyní se rozhodneme doplnit metodu `blikej(int)`, jejíž parametr definuje, kolikrát má světlo bliknout. Abych eliminoval snahy pokročilejších studentů o použití cyklu, přidám požadavek, aby světlo nezastavilo kvůli svému blikání ostatní činnosti, tj. aby na zablikání světla nečekala celá aplikace. Protože ani ti nejpokročilejší studenti většinou ještě s vlákny nepracovali, neznají prostředky, pomocí nichž by bylo možno úlohu splnit.

Seznámíme je proto s návrhovým vzorem *Služebník* (viz [11], [9]) a při té příležitosti také s programovou konstrukcí `interface`, která služebníku umožňuje deklarovat své požadavky na obsluhované objekty, tj. požadavky, kterým musí objekty požadující svoji obsluhu vyhovět, aby je byl služebník ochoten obsloužit.

Pak studentům nabídneme předem připravenou třídu `Opakovač`, jejíž instance budou služebníci, a interface `IPříkaz`, který deklaruje služebníkovy požadavky na obsluhované instance. Vyzkoušíme si aplikaci návrhového vzoru a ověříme, že tímto způsobem se nám opravdu podařilo rozblikat světla, aniž by jejich blikání nějak viditelně zdržovalo ostatní činnost programů.

Vzápětí aplikujeme právě poznaný návrhový vzor ještě jednou, když se rozhodneme naučit naše světlo a ostatní grafické objekty plynule se přesouvat po plátně. Nyní již mohou studenti spolupracovat a sami specifikovat služebníkovy požadavky na obsluhované objekty. Pak opět „vytáhneme z rukávu“ třídu `Přesouvač`, jejíž instance budou oni hledaní služebníci, a interface `IPosuvný`, který formalizuje služebníkovy požadavky.

3.3 Dědičnost rozhraní, návrhový vzor *Pozorovatel*

Přesouvač z minulé hodiny nepracoval stejně jako opakovač, tj. neřešil svůj úkol v samostatném vlákně. Při přesouvání objektu přesouvačem se další příkaz začal provádět až poté, co byl přesun ukončen. To zabraňovalo přesouvání více objektů najednou.

Problém je v tom, že kdybychom přesouvali objekt v samostatném vlákně, neuměl by jednoduše poznat, zda už se dostal do cíle a může provádět v cíli požadovanou činnost. Aby to zjistil, musel by se neustále ptát: „Už jsem na místě?“ (obdobně jako Oslík v 2. dílu Shreka). Prozradíme jim, že obdobný problém řeší návrhový vzor *Pozorovatel*, v němž se pozorovatel (v našem případě přesouvaný objekt) přihlásí u pozorovaného objektu (v našem případě přesouvač), aby mu řekl, že nastala očekávaná událost.

Definujeme třídu `Multipřesouvač`, jejíž instance dokáže přesouvat řadu objektů současně. Požaduje však, aby objekty implementovaly interface `IMultiposuvný`, který je potomkem `IPosuvný`, a přidává metodu `přesunuto()`. Tu multipřesouvač zavolá v okamžiku, kdy přesouvaný objekt vítězně přesune do cíle.

Při plynulých přesunech vyvstane problém odmazávání nepohyblivých objektů, přes které „přejede“ některý plynule přesouvaný objekt. Vysvětlíme, že i tento problém můžeme řešit pomocí návrhového vzoru *Pozorovatel*. Dosavadní třídu `Plátno`, jejíž instance poskytovala na obrazovce prostor, na němž se všechny objekty kreslily, nahradíme třídou `AktivníPlátno`, která již nebude klasickým plátnem, ale bude fungovat jako manažer, jenž bude překreslování objektů na plátně organizovat. Každý, kdo bude chtít zobrazován na plátně, se u něj bude muset přihlásit, a on mu vždy řekne, kdy se má znovu nakreslit.

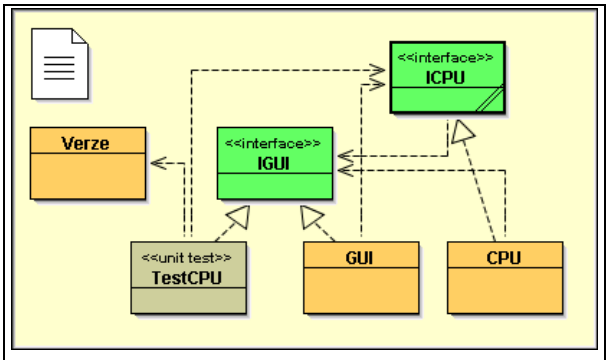
3.4 Referenční a hodnotové typy, jednoduché kontejnery, návrhový vzor *Iterátor*

V další hodině se seznámíme studenty s rozdílem mezi referenčními a hodnotovými objektovými typy a vysvětlíme jim princip a účel definici neměnných hodnotových typů. Ukážeme jim příklad takové definice na typu `Zlomek`.

Poté jim předvedeme práci s kontejnery typu množina a seznam a vysvětlíme význam jejich typových parametrů. Probereme také hlavní zásady, jež je nutné dodržet při definicích překryvů metod `equals(Object)` a `hashCode()`. Vysvětlíme, že má-li kontejner dbát na zapouzdření, nesmí nikoho pustit ke svému internímu úložišti a využívá proto iterátor. Ukážeme si jeho použití a seznámíme je s jeho vlastnostmi a omezeními.

3.5 Návrhový vzor *Most*, programování proti rozhraní

Nyní již studenti znají (přesněji měli by znát) mnohé ze zásad OOP a je nejvyšší čas vyzkoušet jejich aplikaci. Předvedeme jim rámec projektu *Kalkulačka*, ve kterém uvidí příklad jednoduché binární kalkulačky sestávající ze dvou tříd: třídy `GUI` realizující uživatelské rozhraní a třídy `CPU` implementující aritmetickou jednotku. Třída `CPU` implementuje interface `ICPU`. Při definici třídy `GUI` není známo, jaká instance bude představovat CPU vytvářené kalkulačky. Ví jenom, že bude implementovat rozhraní `ICPU`. Odkaz na příslušnou instanci obdrží až konstruktor `GUI` jako parametr.



Obr. 2: *Diagram tříd projektu Kalkulačka*

Vysvětlíme studentům, že takovéto řešení je jednou z možných implementací návrhového vzoru *Most*, který umožňuje, aby třída nebyla závislá na tom, s jakou implementací nějakého rozhraní pracuje. Současně jim vysvětlíme, jak se na tomto návrhovém vzoru názorně projevuje výhoda zásady neprogramovat proti implementaci, ale programovat proti rozhraní.

Jejich první semestrální prací je definovat předem zadanou verzi CPU, která bude počítat s hodnotami některého hodnotového objektového typu. Při předvádění své práce pak mohou vidět, jak je systém schopen akceptovat novou třídu, kterou do té chvíle neznal a o které pouze věděl, že bude implementovat předem známé rozhraní.

3.6 Mapy, pole, návrhový vzor *Příkaz*

Vysvětlíme studentům podstatu kontejnerů typu mapa (slovník) a pak probereme klasické jednorozměrné pole jako speciální případ statické mapy, jejíž prvky jsou indexovány celými čísly. Přitom studenty upozorníme, že pole jsou na rozdíl od map z kontejnerové knihovny pouze statická, ale na druhou stranu je práce s nimi přímo zabudována ve virtuálním stroji a v instrukčních souborech procesorů, takže je mnohem efektivnější.

Seznámíme je s metodami pro třídění polí a ukážeme jim, jak lze prostřednictvím vhodně definovaných instancí implementujících interface `Comparator` zadávat různé druhy třídění.

3.7 Zanořené třídy, návrhový vzor *Tovární metoda*, implementace iterátoru

Vysvětlíme studentům jaký je účel vnořených, vnitřních a lokálních tříd a seznámíme je s jejich implementací a zvláštnostmi. Pak předvedeme několik příkladů jejich použití ve standardní knihovně, především pak implementaci iterátoru. Při té příležitosti je seznámíme s rozhraním `Iretable` a prozradíme jim, že instance tříd implementujících toto rozhraní mohou využívat zkrácené verze cyklu `for`. Současně jim také vysvětlíme, že popsaný mechanizmus je jednou z možných implementací vzoru *Tovární metoda*.

3.8 Abstraktní třídy, dědění od abstraktních tříd, návrhový vzor *Stav*

Zavedeme abstraktní třídy jako konstrukci, která v sobě slučuje vlastnosti klasických tříd a interfejsů: od tříd přebírají schopnost implementace, tj. definice atributů a konkrétních metod, od interfejsů schopnost deklarovat abstraktní, tj. neimplementované metody. Vysvětlíme rozdíl mezi děděním rozhraní, které se uplatní při implementaci tohoto rozhraní, a děděním implementace, které se uplatní při dědění od abstraktní třídy. Znovu zopakujeme, že konkrétní (tj. ne abstraktní) potomci tříd dědí implementované metody a musí sami implementovat deklarované abstraktní metody.

Prozatím se vyhneme výkladu překrývání neabstraktních virtuálních metod a řekneme studentům, že je o této možnosti poučíme později.

3.9 Návrhové vzory *Zástupce*, *Strategie*, *Řetěz odpovědnosti*

Před úplným probráním dědičnosti tříd seznámíme studenty ještě s několika často používanými vzory, při jejichž implementaci dědičnost tříd nepotřebují. Použití probíraných vzorů předvedeme jak na speciálních příkladech, tak na příkladech ze standardní knihovny.

3.10 Dědičnost tříd, její pravidla a úskalí, návrhové vzory *Adaptér* a *Šablona*

Vstupní kurz programování uzavřeme kompletním probráním dědičnosti tříd včetně překrývání metod demonstrovaným jak na speciálních příkladech, tak na příkladech ze standardní knihovny. Vysvětlíme všechny klady a zápory této konstrukce a upozorníme také na situace, kdy se nesmí používat překrytelné metody a také na situace, kdy je třeba vytváření potomků, resp. definici překryvších metod zakázat a ukážeme prostředky, které je možno k danému účelu použít.

Znovu zdůrazníme, že moderní programování dává před dědičností přednost vkládání objektů a že dědičnost je vhodné použít jen v jednoznačně „dědičných“ případech.

4 ZÁVĚR

Příspěvek ukázal, že metodiku *Object First* je možné ještě dále vylepšit a ve vstupních kurzech programování ještě více prohloubit seznamování začátečníků s moderními trendy programování.

Popsal jsem rámcový postup výuky podle metodiky *Design Patterns First* a předvedl řadu příkladů a postupů, na nichž jsem se snažil ukázat, že při vhodně volených příkladech je možné studenty seznamovat s návrhovými vzory prakticky od samého počátku kurzu. Moje zkušenosti ukazují, že uplatněním popsané metodiky je možné vštípit studentům zásady moderního programování mnohem efektivněji a pevněji. Studenti absolvující tyto kurzy se dokáží výrazně lépe orientovat ve složitých objektových programech, které mají doplnit o nějakou drobnou funkčnost, což bývá jedním z nejčastějších úkolů absolventů našich kurzů v praxi.

Vedlejším, nicméně pro vyučujícího velmi příjemným efektem popsané metodiky je, že ve vstupních kurzech velice rychle sjednotí úroveň studentů, protože ani ti zkušenější studenti nebudou většinou umět použít jiné konstrukce, než ty přednášené.

LITERATURA

[1] *4th "Killer Examples" for Design Patterns and Objects First workshop* http://www.cse.buffalo.edu/faculty/alphonce/KillerExamples/OOPSLA2005/

[2] BARNES David J.; KÖLLING Michael: *Objects First with Java: A Practical Introduction Using BlueJ – 2nd edition*. Prentice Hall, 2005, ISBN 0-13-124933-9.

[3] BLOCH, Joshua. *Effective Java – Programming Language Guide*. Addison-Wesley Professional © 2001. 252 s. ISBN 0-201-31005-8. (Český překlad: *Java efektivně – 57 zásad softwarového experta*. Praha: Grada © 2002. 230 s. ISBN 80-247-0416-1)

[4] *Computing Curricula 2001, Computer Science Volume.* http://www.sigcse.org/cc2001/

[5] DAHL Ole-Johan; NYGAARD Kristen: *How Object-Oriented Programming Started*, http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

[6] FREEMAN Eric; FREEMAN Elisabeth: *Head First Design Patterns*. O'Reilly, © 2004. ISBN 0-596-00712-4.

[7] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. (Český přelad: *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.* Praha: Grada, © 2003. 386 s.

[8] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš: Let's Modify the Objects First Approach into Design Patterns First, *Proceedings of the Eleventh Annual Conference on Innovation and Technology in Computer Science Education,* University of Bologna 2006.

[9] PECINOVSKÝ Rudolf: Začlenění návrhových vzorů do výuky programování. *Objekty 2005 – sborník příspěvků devátého ročníku konference*, VŠB, Ostrava 2005. ISBN 80-248-0595-2.

[10] PECINOVSKÝ Rudolf: Jak efektivně učit OOP. *Tvorba softwaru 2005 – sborník přednášek*. ISBN 80-86840-14-X.

[11] PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java 5.0*, Grada, 2004. ISBN 80-247-0941-4.

[12] PECINOVSKÝ Rudolf: Jak při výuce Javy opravdu začít s objekty. *Objekty 2004 – sborník příspěvků devátého ročníku konference*, ČZU, Praha 2004. ISBN 80-248-0672-X.

[13] PECINOVSKÝ Rudolf: Proč a jak učit OOP žáky základních a středních škol. *Žilinská didaktická konferencia*, 2004, Žilina.

[14] PECINOVSKÝ Rudolf: Výuka objektově orientovaného programování žáků základních a středních škol, *Objekty 2003 – sborník příspěvků osmého ročníku konference*, VŠB, Ostrava 2003. ISBN 80-248-0274-0.

[15] Shalloway, A., Trott, J. A. *Design Patterns Explained – A new Perspective on Object-Oriented Design (2nd edition)*. Addison-Wesley, 2004. ISBN 0-321-24714-0.

[16] *The ACM Java Task Force – Project Rationale*, Second Public Draft (February 23, 2006), ke stažení na adrese http://www-cs-faculty.stanford.edu/~eroberts//jtf/rationale/rationale.pdf