Metodika *Design Patterns First* v roce 2010

Rudolf Pecinovsky[1]

[1]Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky,
Katedra informačních technologií
ICZ, Na Hrebenech II 1718/10, 140 00 Praha 4
rudolf@pecinovsky.cz

Abstrakt: Metodika *Design Patterns First* se narodila v roce 2004 a od té doby se neustále vyvíjí a (doufejme) vylepšuje. Tato metodika vychází z metodiky *Object First*, kterou doplňuje o důslednou aplikaci zásady, že nejdůležitější témata se mají přednášet co nejdříve. Od začátku proto studenty seznamuje s koncepcí rozhraní a návrhových vzorů. Článek shrnuje stav této metodiky v roce 2010. Podrobně popisuje aktuálně používaný postup výuky. Ukazuje, co vše tato metodika doporučuje probrat v konverzačním režimu použitého vývojového prostředí *BlueJ* a znovu zopakovat v následujícím textovém režimu. Podrobně také vysvětluje novou koncepci výkladu dědičnosti postavenou na analogii s návrhovým vzorem *Dekorátor*.

Klíčová slova: Design Patterns, návrhové vzory, Design Patterns First

1 Úvod

Metodika *Design Patterns First* se narodila v roce 2004 v reakci na nedostatky jiných, v té době používaných metodik programování. Především nám vadilo, že naprostá většina učebnic nedodržovala pedagogickou zásadu ranního ptáčete, která říká: „*Organizujte výklad tak, abyste nejdůležitější témata přednášeli co nejdříve.*“ (*Organize the course so that the most important topics are taught first* – [2]) Jenom tak budou míst studenti dostatek času a příležitostí, aby si tyto zásady osvojili a aby je pak přirozeně aplikovali ve své budoucí praxi. Metodika *Design Patterns First* se navíc snaží od počátku učit studenty agilním technikám vývoje programů ([3]).

Současné programování se řídí řadou důležitých zásad, které bychom studentům měli vštípit. Mezi nejdůležitější patří:

- Programovat proti rozhraní, a ne proti implementaci.
- Využívat při řešení úloh ověřených návrhových vzorů.
- Neodkládat testování až na dobu, kdy bude úloha vyřešena, ale průběžně testovat vyvíjený program.
- Dávat přednost skládání před dědičností, používat dědičnost pouze v odůvodněných případech.

Podívejme se, jak metodika doporučuje uspořádat výklad, aby se výše uvedené zásady naplnily:

2 Začátek v interaktivním režimu

Metodika *Design Patters First* vychází ze starší metodiky *Object First*, jejíž autoři přišli s geniální myšlenkou nabídnout studentům vývojové prostředí, které bude umožňovat práci v interaktivním režimu, v němž student přijímá roli jednoho z objektů programu a komunikuje s ostatními objekty prostřednictvím posílání zpráv. Celá výuka OOP pak začíná právě v tomto interaktivním režimu, v němž se studenti seznamují se základními vlastnostmi objektů, aniž by museli napsat jediný řádek kódu.

Obáváme se však, že autoři kurzů a učebnic, které se touto metodikou řídí, nedocenili výukový potenciál interaktivního režimu a opouštějí jej proto zbytečně brzy – dříve, než studenty seznámí se všemi klíčovými vlastnostmi objektů. Studenti se pak musejí současně učit syntaktická pravidla použitého jazyka a pravidla objektově orientovaného programování, což je často zbytečně rozptyluje a mate.

Metodika *Design Patterns First* se snaží vykládat vždy pouze jedinou oblast a nemíchat proto výuku objektových rysů s výukou syntaxe. Doporučuje proto setrvat u práce v interaktivním režimu delší dobu a probrat v něm všechny klíčové rysy OOP včetně některých, které se v ostatních učebnicích programování vysvětlují až mnohem později – např. konstrukce `interface` a struktury některých návrhových vzorů.

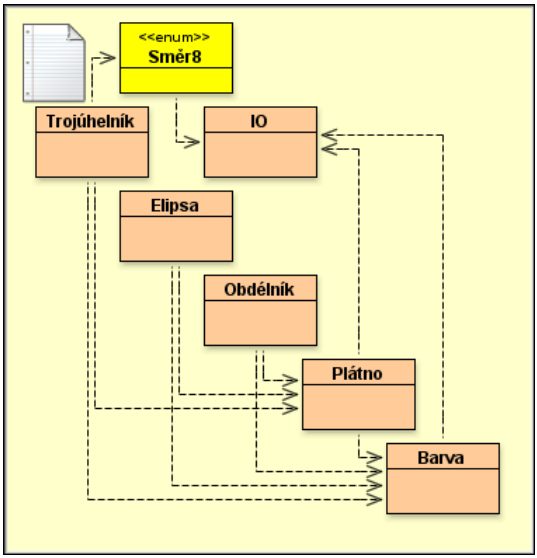
2.1 Základní seznámení s objekty

Začátek výkladu je velmi blízký postupu doporučovanému metodikou *Object First* a byl již popsán v pracech [5], [6], [7] a [8]. I zde se pro vstup do světa programování používá vývojové prostředí *BlueJ*, jehož prostřednictvím se studenti v úvodní hodině seznámí s objekty, třídami a posíláním zpráv. V úvodním projektu jim ukážeme, že objekty mohou reprezentovat nejenom grafické útvary a další entity, které běžný člověk zahrnuje pod hromadný název objekt. Vysvětlíme jim, že v programech mohou objekty představovat i abstraktní pojmy a obecně cokoliv, co můžeme nazvat podstatným jménem.

V prvních lekcích používáme projekt umožňující tvorbu jednoduchých obrázků sestavených ze tří základních geometrických tvarů: obdélníků, elips a trojúhelníků (viz obr. 1). Jako představitelé tříd, jejichž instance reprezentují abstraktní pojmy, zde slouží třídy `Barva` a `Směr8`.

V reakci na tvrzení, že objekt je vše, co můžeme nazvat podstatným jménem, se nás téměř vždy někdo zeptá, zda je třída také objekt. Vysvětlíme studentům, že třída je zvláštní druh objektu. Třídy jsou jediný druh objektů, který umí vytvářet jiné objekty – své instance. Samy třídy přitom nejsou (alespoň v jazyku Java) instancemi žádné třídy tříd.

Při tomto výkladu nám výrazně pomáhá vývojové prostředí *BlueJ*, které pracuje s třídami stejně jako s jejich instancemi. Oba druhy objektů jsou reprezentovány obdélníky, v jejichž místní nabídce je zobrazen seznam zpráv, které jim můžeme poslat. Díky takto pojatému výkladu se jednoduše vyhneme problémům, které studenti mívají při pozdějším zavádění atributů a metod třídy (statických atributů a metod).



Obr. 1: *Úvodní projekt*

2.2 Definice testovací třídy

Po prvních pokusech s interaktivním posíláním zpráv objektům studentům ukážeme, jak lze prováděné akce uchovat. Využijeme přitom toho, že prostředí *BlueJ* si zapamatovává prováděné akce a je schopno z nich na požádání vytvořit kód, který zabuduje do označené testovací třídy.

Studenti se naučí definovat v *BlueJ* testovací třídu, její testovací přípravek a testovací metody, které s tímto přípravkem pracují. Naučí se tak vytvářet své první programy, aniž by museli vědět cokoliv o použitém jazyku a jeho syntaxi.

Současně jim ukážeme, kde na disku najdou zdrojový i přeložený kód vytvořené třídy. Vysvětlíme jim, že až dostanou nějaký domácí úkol, budou odevzdávat soubor se zdrojovým kódem. Současně probereme konvence pro vytváření názvů tříd s jejich budoucími domácími úkoly a zdůrazníme jim, že musejí přidělit třídě správný název hned při její definici. Ukážeme jim, co způsobí dodatečné přejmenování odesílaného zdrojového souboru, které mají mnozí v oblibě.

2.3 Prohloubení výkladu objektů

V dalším výkladu prohlubujeme znalosti o posílání zpráv. Probranou látku si studenti vždy hned procvičí na definici testovací třídy, kterou průběžně doplňují o další testy, v jejichž definicích používají probrané konstrukce. Nejprve takto probereme zprávy vracející hodnotu včetně hodnot objektových typů a zprávy s parametry opět včetně parametrů objektových typů.

V dalším výkladu využijeme toho, že *BlueJ* pracuje s třídami velmi podobně jako s jejich instancemi. Jediným rozdílem je, že třídy jsou umístěny v diagramu tříd

kdežto instance ve speciálním zásobníku odkazů. Znovu studentům připomeneme, že třída je pouze zvláštním druhem objektu. Doposud jsme třídám posílali pouze zprávy žádající o vytvoření jejich instance. Nyní si ukážeme, že jim můžeme stejně dobře posílat i obyčejné zprávy.

Současně si ukážeme, že zprávy poslané třídě, mohou mít dopad na chování všech jejích instancí nezávisle na tom, zda byla daná instance vytvořena před nebo po odeslání příslušné zprávy.

2.4 První domácí úkol

Jakmile se studenti naučí posílat zprávy s parametry, mohou dostat svůj první domácí úkol: mají definovat testovací přípravek, jehož objekty vytvoří na plátně nějaký obrázek. Současně mají definovat dvě testovací metody, které tento obrázek nějakým jednoduchým způsobem animují. Vytvořený zdrojový kód pak odevzdají.

Od této hodiny pak již mohou každou další hodinu dostat domácí úkol, v němž předvedou, nakolik pochopili látku přednášenou na hodině. Abychom posílili zpětnou vazbu, připravili jsme speciální program, který studentům umožní zkontrolovat svá řešení. Program zkontroluje požadované charakteristiky navrženého řešení (např. že testovací přípravek definuje nejméně čtyři objekty, že testovací třída definuje nejméně dvě testovací metody apod.). Obdobný testovací program pak budou mít k dispozici pro domácí úkol na každé následující hodině.

2.5 Průběžné testy

Domácí úkoly, které studenti dostávají, poskytují pouze kontrolu toho, zda se student doma látce opravdu věnoval. Mají však malou vypovídací hodnotu o míře pochopení probrané látky, protože nemáme zaručeno, že je student vypracoval samostatně.

Vlastní ověření míry zvládnutí probrané látky poskytne až krátký test na počátku hodiny. V něm mají studenti za úkol zapracovat do doneseného domácího úkolu drobnou úpravu. Teprve po odevzdání takto upraveného programu je domácí úkol považován za odevzdaný. Tento přístup poskytuje mnohem lepší zpětnou vazbu o stupni zvládnutí probrané látky.

2.6 Atributy a vlastnosti objektů

Po zvládnutí definice testovacích tříd si vysvětlíme, že k tomu, aby objekt mohl na obdržené zprávy správně reagovat, potřebuje znát svůj aktuální stav. Informace o svém stavu si objekt pamatuje v atributech. Ukážeme si, jak nám *BlueJ* umožňuje zobrazovat hodnoty atributů. Při té příležitosti se seznámíme i s atributy třídy a ukážeme si, že třída zprostředkovává přístup k těmto atributům všem svým instancím. Jakmile kdokoliv změní hodnotu statického atributu, všechny instance ihned pracují s novou hodnotou.

Současně studentům vysvětlíme rozdíl mezi atributy a vlastnostmi objektů. Definujeme, že vlastnost je charakteristika, jejíž hodnotu můžeme od objektu zjistit

nebo ji u něj nastavit. Naproti tomu atribut je interní údaj sloužící k tomu, aby daný objekt mohl úspěšně plnit svoji funkci. Ukážeme jim, že některé vlastnosti jsou přímo spojeny s odpovídajícími atributy, kdežto jiné toto přímé spojení nemají. Stejně tak jim ukážeme i atributy, které slouží opravdu pouze pro vnitřní potřebu daného objektu, který nám neposkytuje žádnou možnost zjistit a/nebo nastavit jejich hodnotu.

2.7 Konstrukce interface

V současnosti používaný další postup se od návrhu v Pecinovský, Pavlíčková & Pavlíček (2006) poněkud liší. Nyní již po úvodním výkladu práce s objekty a posílání zpráv nepřecházíme hned k psaní kódu, ale setrváváme v interaktivním režimu a prohlubujeme znalosti o objektovém programování.

Po procvičení různých variant posílání zpráv včetně jejich důsledků na hodnoty atributů studentům vysvětlíme dva aspekty každého objektu: jeho rozhraní a jeho implementaci. Povíme jim, že současné programování prosazuje zásadu, že se má programovat proti rozhraní a ne proti implementaci.

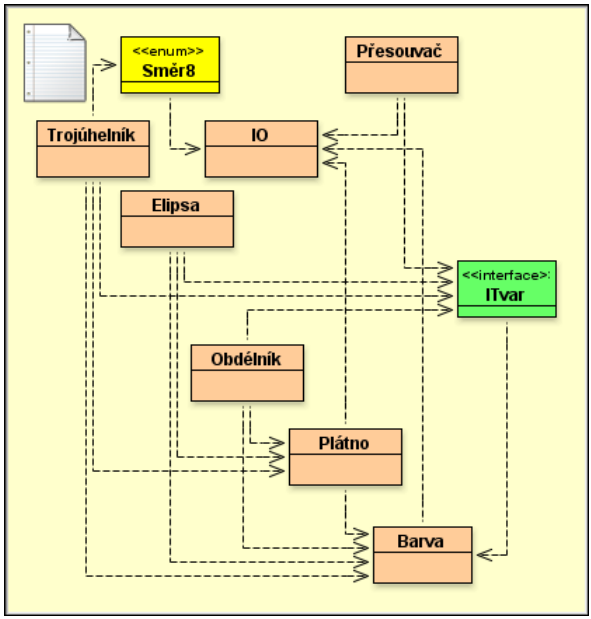
Při té příležitosti studentům představíme nový druh datového typu: `interface`. Vysvětlíme jim, že je to datový typ, který má definováno pouze rozhraní a žádnou implementaci – jedná se tedy a jakousi formalizovanou podobu rozhraní zapsanou prostředky jazyka. Protože takovýto typ nemá žádnou implementaci, nemůže mít ani žádné instance. Místo toho se třídy mohou přihlásit k tomu, že implementují metody deklarované v daném *interfejsu* a tím získají právo vydávat svoje instance za instance implementovaného *interfejsu*. Přitom jim současně prozradíme, že kdykoliv se někde mluví o instancích *interfejsu*, hovoří se ve skutečnosti o instancích tříd, které tento *interfejs* implementují.

Pro další výklad uděláme dohodu, že obecné rozhraní a jeho formalizovanou podobu odlišíme terminologicky. Budeme-li hovořit o programové konstrukci, budeme používat termín *interfejs*, v prvním pádě pak většinou dáme přednost jeho anglické podobě `interface`. Budeme-li hovořit o obecné vlastnosti objektů, budeme používat termín *rozhraní*. Termín rozhraní pak budeme používat i v situacích, kdy nemůže dojít k nedorozumění.

Po tomto spíše teoretickém úvodu se vrátíme znovu k našim experimentům s objekty. Doplníme projekt o předem definovaný `interface`, který skupina tříd z projektu přirozeně implementuje, a o třídu, jejíž metody pracují s parametry, které jsou instancemi tohoto *interfejsu*.

Ukážeme jim, že dokud třída k implementaci příslušného *interfejsu* explicitně nepřihlásí, nemůžeme její instance předat jako parametry metodám nově přidané třídy. Není důležité, zda objekt implementuje metody vyžadované příslušným *interfejsem*, ale zda se mateřská tohoto objektu třída k implementaci daného *interfejsu* veřejně hlásí.

Předvedeme studentům, jak toto veřejné přihlášení se třídy k implementaci *interfejsu* realizovat, tj. jak v diagramu tříd jednoduše definovat implementaci *interfejsu* třídou. Pak vyzkoušíme, že po explicitně definované implementaci můžeme instance dané třídy používat jako instance implementovaného *interfejsu*.



Obr. 2: *Úvodní projekt po zavedení interfejsu*

2.8 Seznámení s prvními návrhovými vzory

Nyní již studenti umějí vše potřebné pro to, abychom je mohli podrobněji seznámit s ideou návrhových vzorů (doposud jsme o nich hovořili spíše platonicky). Vysvětlíme studentům termín *návrhový vzor* a v aktuálním projektu jim současně ukážeme, kde všude v něm jsou návrhové vzory použity. Předvedeme jim (stále jsme v interaktivním režimu) aplikace návrhových vzorů *Tovární třída*, *Jedináček*, *Výčtový typ*, *Originál* a *Prototyp*. Na příkladu třídy přidané při výkladu konstrukce `interface` jim vysvětlíme základní principy a použití návrhového vzoru *Služebník* (je to ve skutečnosti pouze trochu jinak nahlížený vzor *Příkaz*).

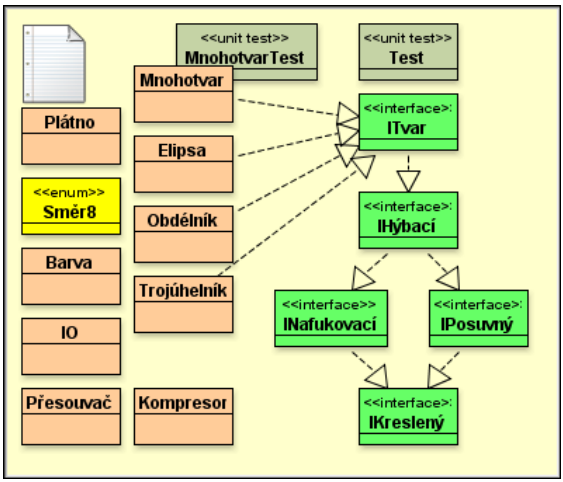
Někteří kritici se občas ptají, proč se metodika jmenuje *Design Patters First*, když se návrhové vzory neprobírají na počátku. Když metodika vznikla, seznamovali jsme studenty s návrhovými vzory opravdu již na první hodině a ukazovali jsme jim, kde jsou v úvodním projektu použity. Ukázalo se, že je však výhodnější zařadit výklad až ve chvíli, kdy mají studenti dostatečné znalosti pro pochopení i těch složitějších vzorů a zejména ve chvíli, kdy mohou začít návrhové vzory sami aktivně používat.

2.9 Dědičnost interfejsů

Pokračujeme výkladem dědičnosti *interfejsů* a z ní plynoucích důsledků. Zavedeme do projektu několik dalších rozhraní vytvářejících dědickou struktur a ukážeme, jak se dědičnost *interfejsů* zobrazuje v diagramu tříd. Na příkladech pak předvedeme, že

třída implementující potomka *interfejsu* je automaticky považována za třídu implementující jeho předka.

Vysvětlíme studentům význam dědičnosti rozhraní a ukážeme jim, jak lze využitím dědičnosti minimalizovat požadavky kladené na třídy, jejichž instance potřebujeme používat jako parametry různých užitečných metod.



Obr. 3: *Přeuspořádaný úvodní projekt při předvádění dědičnosti interfejsů*

2.10 Další návrhové vzory

Projekt, na němž se studenty od počátku pracujeme, je navržen tak, aby všechny prováděné akce byly pro studenty maximálně pochopitelné. To s sebou (záměrně) nese i některé nepříjemné důsledky. Jedním z nich je, že grafické objekty se před změnou velikosti nebo pozice v původní pozici nejprve smažou. Toho nyní využijeme a začneme studenty připravovat na úpravu, která tyto nepříjemné vlastnosti odstraní.

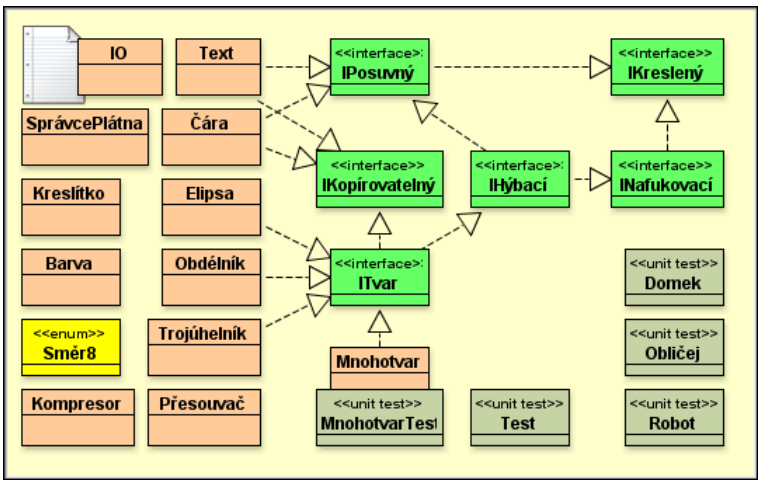
Vysvětlíme jim, co by se muselo změnit, aby objekty, jejichž část byla při přesunu či změně rozměru jiných objektů odmazána, opět získaly svoji původní podobu. Ukážeme jim hlavní problémy naivního řešení a seznámíme je s návrhovými vzory *Prostředník* a *Pozorovatel*, které nám umožní vyřešit uvedený problém elegantně.

Otevřeme nový projekt, v němž je původní plátno nahrazeno správcem plátna. Každý objekt, který chce být zobrazen na plátně, se musí nejprve přihlásit u správce plátna. Správce je však ochoten akceptovat pouze objekty implementující rozhraní `IKreslený`. To vyžaduje implementaci metody, po jejímž zavolání se daný objekt nakreslí dodaným kreslítkem.

Správce plátna pak rozhoduje, kdy danou metodu daného objektu zavolá a tím i kdy se kterýkoliv ze spravovaných objektů nakreslí. Studenti se tak poměrně záhy setkají s událostmi řízeným programováním. (Připomínám, že stále ještě nenapsali ani řádek kódu.)

Po tomto úvodním seznámení předvedeme studentům, jak je třeba upravit definici testovacích přípravků v jejich třídách. Ukážeme si, že po této drobné úpravě pak

všechny metody pracují obdobně jako před tím. Jediným rozdílem je kvalitnější zobrazení při změnách pozic a velikostí objektů na plátně.



Obr. 4: *Upravený projekt využívající Správce plátna*

3 Opakování v textovém režimu

Výkladem rozhraní, jeho dědičnosti a jeho použití v některých návrhových vzorech končí interaktivní část výuky. V další části se znovu vrátíme k úvodnímu projektu s jednoduchým plátnem a začneme se studenty pracovat v textovém režimu. V něm si postupně zopakujeme vše, co jsme probírali v interaktivním režimu, a současně je naučíme, jak jednotlivé probírané konstrukce zapsat ve zdrojovém kódu. Tím se dosáhne toho, že si studenti ještě více upevní znalosti získané v první etapě a navíc si je při opakování mohou zasadit do kontextu následně získaných znalostí.

3.1 Problematika konstruktorů

Začneme tím, že jim na příkladu velmi jednoduché třídy vysvětlíme, jak mohou definovat novou třídu. Poté jejich znalosti prohlubujeme tak, aby mohli v textovém režimu definovat třídu, jejíž instance po vytvoření nakreslí stejný obrázek, jako nakreslila jejich testovací třída, kterou vytvářeli v interaktivním režimu, přesněji řečeno jaký se nakreslil při vytváření jejího testovacího přípravku.

Při rozšiřování vlastností vytvářené třídy vyvstane potřeba definovat několik konstruktorů. Probereme proto se studenty význam a funkci konstruktorů. Seznámíme je s problematikou přetěžování, naučíme je vytvářet více konstruktorů jedné třídy a předvedeme jim, jak mohou konstruktory delegovat odpovědnost za prvotní inicializaci vytvářeného objektu na některého ze svých kolegů.

Při tomto výkladu se velmi osvědčuje nevysvětlovat konstruktor jako metodu, která se jmenuje stejně jako její třída, ale pojmout jej naopak jako metodu, jejíž

skutečný název odporuje pravidlům tvorby identifikátorů, a proto se s ní pracuje, jako kdyby byl její název prázdný řetězec (podrobnosti viz Pecinovský R. (2010)).

3.2 Zavedení atributů

Po definici konstruktorů začneme vytvořenou třídu doplňovat o první metody. Přitom zjistíme, že si k jejich úspěšné definici potřebujeme pamatovat některé informace použité při konstrukci objektů. Doplníme tedy definici o deklarace potřebných atributů a doplníme tělo neobecnějšího konstruktoru o inicializaci těchto atributů.

Postupně vytvářenou třídu stále vylepšujeme a doplňujeme metody s dalšími vlastnostmi probranými v předchozím interaktivním režimu.

3.3 Implementace rozhraní

Po zopakování základních konstrukcí a vysvětlení jejich zápisu ve zdrojovém kódu programu přejdeme k implementaci rozhraní. Ukážeme studentům, jak se implementace rozhraní zapisuje ve zdrojovém kódu a jak můžeme její zápis usnadnit využitím možností nabízených prostředím *BlueJ*. Připomeneme jim, jak se implementací vhodných rozhraní zvyšuje použitelnost třídy a jak se současně zvyšuje její schopnost využívat možností nabízených různými služebníky.

Znovu opustíme jednodušší projekt a vrátíme se k dokonalejšímu projektu se správcem plátna. Vysvětlíme studentům základy refaktorace a ukážeme jim, jak v tomto vylepšeném projektu zprovozní své třídy.

Připomeneme studentům, že i rozhraní každého má dvě složky: signaturu a kontrakt. Vysvětlíme jim, že doposud jsme probírali pouze zápis signatury, ale ještě jsme se nenaučili zapisovat kontrakt. Prozradíme jim, že se k zápisu kontraktu používají dokumentační komentáře a naučíme je tyto komentáře vytvářet. Od této chvíle budou muset být všechny jejich odevzdávané programy vybaveny komentáři.

3.4 Prohlubování znalostí

V dalším výkladu postupně prohlubujeme dosavadní znalosti objektů a práce s nimi a přidáváme znalosti další. Vedle témat, s nimiž se můžeme setkat i v ostatních učebnicích zahrneme i výklad návrhového vzoru *Přepravka* a ukážeme studentům, jak s jeho pomocí definovat metody, které vracejí několik hodnot současně.

Na závěr druhého bloku studentům vysvětlíme princip zavádění tříd a seznámíme je s konstruktorem třídy – statickým inicializačním blokem. Vysvětlíme jim současně i vlastnosti instančního inicializačního bloku, aby chápali chování programu v případě, kdy při definici konstruktoru třídy zapomenout uvést modifikátor `static` nebo když se jim při úpravách programu ztratí hlavička metody a v kódu zůstane pouze její tělo.

4 Složitější konstrukce

V průběhu druhého bloku výkladu jsme stále pracovali v kořenovém balíčku. Následující blok začne zavedením konceptu balíčků. V dalším výkladu se oddělí hotové třídy sloužící jako knihovní od tříd vytvářených při výuce. Každý další studentský projekt proto bude využívat více balíčků.

4.1 Zavedení balíčků

Během předchozího výkladu se stal používaný projekt již poměrně složitým. Budeme-li do něj přidávat další třídy a rozhraní, bude problematické jej přehledně zobrazit. Máme tedy pro studenty dostatečnou motivaci k tomu, abychom tento složitý projekt rozdělili na několik menších, vzájemně spolupracujících „projektů".

Vysvětlíme studentům koncepci balíčků a ukážeme jim, jak lze náš dosavadní projekt rozdělit do několika balíčků tak, aby v každém byly třídy a rozhraní, které spolu logicky souvisejí a aby se současně minimalizovali vzájemné vazby mezi jednotlivými balíčky.

Jmennou konvenci, kterou jsme doposud používali pro názvy odevzdávaných tříd s domácími úkoly, upravíme pro názvy balíčků. Od tohoto okamžiku studenti odevzdávají své domácí úkoly vždy jako balíček.

4.2 Další návrhové vzory – dekorátor

V dalších lekcích řešenou úlohu postupně doplňujeme o další zadání a tím celý projekt vylepšujeme a současně i zesložiťujeme. Jedno z takovýchto rozšiřujících zadání vede na řešení, v němž by absolvent klasicky pojatých kurzů použil dědičnost tříd, ale zkušení programátoři vědí, že pro danou úlohu je mnohem vhodnější použít návrhový vzor *Dekorátor*. Studenty proto s tímto vzorem seznámíme a podle tohoto vzoru dané zadání vyřešíme.

Účel tohoto příkladu je dvojí. Za prvé se studenti včas seznámí s alternativními možnostmi řešeni problémů, takže až poznají dědičnost, nebudou ji bezhlavě aplikovat na řešení všech problémů. Za druhé nám pak znalost tohoto vzoru poslouží při pozdějším výkladu dědičnosti tříd.

4.3 Algoritmické konstrukce a knihovna kolekcí

Jedinou algoritmickou konstrukcí, kterou jsme v dosavadním výkladu používali, byla posloupnost příkazů. Veškeré obraty, které by „strukturovaný programátor" řešil zavedením podmíněných příkazů či cyklů jsme doposud vždy dokázali řešit použitím objektových konstrukcí. Takovýto způsob řešení je však vhodný pouze na omezenou množinu problémů.

V dalším výkladu tak studenty postupně seznamujeme s jednotlivými algoritmickými konstrukcemi a jejich použitím. Paralelně jim představujeme nejdůležitější kontejnery z knihovny kolekcí. Před tím je ale musíme seznámit s

koncepcí generických datových typů a ukázat jim, jak příslušná pravidla aplikovat právě při práci s kolekcemi.

4.4 Pole

V závěru tohoto bloku seznámíme studenty s klasickými poli. Vysvětlíme jim, že pole můžeme chápat jako speciální typ seznamu a že do jazyka bylo zavedeno proto, že má přímou podporu v instrukčních souborech většiny současných mikroprocesorů. Naučíme studenty pracovat s poli, přebírat je jako parametry a vracet jako návratové hodnoty. Současně jim ukážeme, jak převádět kolekce na pole a naopak jak převádět pole na seznamy.

Při práci s poli studentům představíme i metody s proměnným počtem parametrů. Ukážeme jim, jak s takovými parametry pracovat a jak definovat vlastní metody tohoto druhu.

5 Dědičnost

Doposud jsme v žádném z definovaných programů nepoužili dědičnost. Její výklad si ponecháváme na závěr základního kurzu.

5.1 Tři druhy dědičnosti

Nejprve studentům vysvětlíme, že obdobně jako rozhraní mohou i třídy mít svoje potomky a předky. Nejprve však studenty seznámíme s tím, že v objektovém programování rozeznáváme tří druhy dědičnosti (Lalond W. & Pugh J. (1991), Pecinovský R. (2010)): dědičnost rozhraní, dědičnost implementace a nativně chápanou dědičnost. Vysvětlíme jim, že ve správně navrženém programu musejí být všechny tři typy dědičnosti v souladu. Každý nesoulad ohrožuje stabilitu a robustnost vytvářeného programu.

Prozatím jsme se setkávali pouze s dědičnosti rozhraní. V dalších lekcích začneme probírat základní pravidla dědičnosti implementace.

5.2 Abstraktní třídy a rodičovský podobjekt

Začneme s příklady, při nichž nebudeme potřebovat překrývat zděděné metody. Seznámíme studenty s pojmem *abstraktní třída* a vysvětlíme jim, že je to hybrid mezi třídou a rozhraním. Od třídy přebírá schopnost definovat metody včetně jejich implementace, od rozhraní pak přebírá možnost zavést abstraktní metody, tj. metody, které jsou pouze deklarovány, avšak nejsou implementovány. Spolu s výhodami však přebírá i některá omezení: od rozhraní přebírá neschopnost vytvořit vlastní instanci, od tříd pak nutnost mít pouze jediného rodiče. Tuto nutnost zdůvodníme na příkladu dědění od dvou rodičů se společným předkem.

Dědičnost tříd přitom vysvětlujeme jako překladačem implementovaný návrhový vzor *Dekorátor*, v němž je odkaz na dekorovaný objekt uložen do atributu nazvaného `super` (podrobněji viz Pecinovský R. (2010)). Při takto chápané dědičnosti je řada syntaktických konstrukcí pouze logickým důsledkem dříve poznaných pravidel a studenti se je nemusí učit jako něco nového.

Při řešení příkladů studentům předvedeme, jak lze do společné rodičovské třídy vytknout společné atributy a shodně definované metody a naopak metody, jejichž definice se u jednotlivých potomků liší, deklarovat jako abstraktní. Spolu pak upravíme knihovnu správce plátna tak, abychom pro grafické třídy definovali v odůvodněných případech jejich společné rodiče, do nichž se vytknou příslušné atributy a metody. Při tomto převodu se ukáže, že v daném případě je vhodné vytvořit dvě abstraktní rodičovské třídy, přičemž jedna bude potomkem druhé. Studenti se tak naučí definovat i složitější stromy dědičnosti.

5.3 Dědičnost tříd jako automatizovaná implementace vzoru dekorátor

Po výkladu dědičnosti, při níž není potřeba překrývat rodičovské metody, plynule pokračujeme výkladem plnohodnotné dědičnosti tříd. Vysvětlíme studentům mechanizmus překrývání metod a poukážeme na paralelu s obdobným problémem řešeným při použití návrhového vzoru *Dekorátor*.

V dalším výkladu studenty seznámíme s úskalími virtuálních metod. Vysvětlíme jim, proč je nesmějí používat v konstruktoru a jak může při nevhodné definicí překrývající metody dojít k přeplnění zásobníku.

Zvláštní pozornost věnujeme nevhodným návrhům dědičnosti. Na příkladech ukážeme, kdy nevhodný návrh překrývajících metod porušuje kontrakt předka a dochází tak nesouladu mezi třemi aspekty dědičnosti.

Znovu připomeneme úlohu řešenou před výkladem dědičnosti a ukážeme si, že řešení prostřednictvím dekorátoru je výhodnější. Vysvětlíme studentům, že by vždy měli přemýšlet nejprve nad možností skládání a po dědičnosti sáhnout až v okamžiku, kdy se všechna ostatní řešení ukáží jako ne zcela vhodná.

6 Závěr

Článek shrnuje současný stav metodiky *Design Patters First*. Připomíná základní myšlenky, na nichž je metodika postavena. Podrobně vysvětluje doporučovaný postup výkladu a připomíná, na která témata se v jednotlivých fázích výkladu soustředit. Ukazuje, co všechno je možno probrat při používání interaktivního režimu, v němž studenti přímo oslovují objekty projektu, aniž by napsali jediný řádek kódu. Vysvětluje, jak lze v tomto režimu zahrnou do výkladu i problematiku rozhraní včetně jejich dědičnosti, která je důležitým výchozím bodem pro navazující výklad návrhových vzorů.

V další části článek probírá metodiku výkladu v textovém režimu, v němž si studenti opakují znalosti získané při práci v interaktivním režimu a přitom se učí syntaxi jazyka a zápis různých konstrukcí do kódu. V poslední části jsou pak

podrobně probrána doporučení pro výklad dědičnosti tříd, kterým základní kurz končí.

7 Poděkování

Tento článek byl zpracován za podpory prostředků institucionální podpory na dlouhodobý koncepční rozvoj vědy a výzkumu na Fakultě informatiky a statistiky VŠE v Praze.

8 Literatura

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

[2] Bergin J.: Fourteen Pedagogical Patterns. *Proceedings of Fifth European Conference on Pattern Languages of Programs.* (EuroPLoP™ 2000) Irsee 2000.

[3] Buchalcevová A.: Where in the curriculum is the right place for teaching agile methods? *Proceedings 6th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2008).* Prague : Copyright, 2008, p. 205–209. ISBN 978-0-7695-3302-5.

[4] Lalond W., Pugh J.: *Subclassing ≠ Subtyping ≠ IsA*. Journal of Object-Oriented Progrtamming. Vol. 3, No. 5. 1991.

[5] Pecinovský R.: *Myslíme objektově v jazyku Java 5.0*, Grada, 2004. ISBN 80 247 0941 4.

[6] 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.

[7] 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. Available from: http: vyuka.pecinovsky.cz

[8] Pecinovský R.: *Early Introduction of Inheritance Considered Harmful*. Objekty, Hradec Králové.

[9] Pecinovský R.: *Using the methodology Design Patterns First by prototype testing with a user.* Proceedings of IMEM, Spišská Kapitula.

[10] Pecinovský R.: *How to improve understanding of object constructs using a slightly modified explanation*. Available at: http: vyuka.pecinovsky.cz

Abstract: Methodology *Design Patterns First* has born in the 2004 and it continuously developed and (hopefully) improved. This methodology is based on the methodology *Object First* which it fulfills by the consistent application the early bird rule telling, that the most important subjects has to be explained at first. It therefore introduce from very beginning the concept of interfaces and design patterns. The paper summarizes the state of this methodology in the year 2010. It describes in detail the currently used method of explanation. It shows what subjects are recommended for explanation using the interactive mode of the IDE *BlueJ* and subsequent revision in the text mode. The paper also explains the now concept of the explanation of inheritance, the concept based on the analogy with the design pattern *Decorator*.