Aplikace metodiky Design Patterns Fist

Rudolf Pecinovský[1]

[1]Amaio Technologies Inc., 100 00 Praha 10, Třebohostická 14
VŠE, Fakulta informačních technologií, 130 67, Praha 3, nám W. Cuhurchilla 4
rudolf@pecinovsky.cz

Abstrakt. Efektivita výuky programování je závislá nejenom na zvolené metodice, jež má optimalizovat předávání informací a získávání dovedností, které bude student potřebovat v praxi, ale i na kvalitě příkladů, na nichž jsou studentům vysvětlované konstrukce a postupy demonstrovány, a na výběru úloh, jež mají studenti samostatně řešit. Příspěvek shrnuje důvody, které vedly k zavedení metodiky *Design Patterns First* a následně se zabývá zásadami, na něž je třeba pamatovat při návrhu doprovodných příkladu a samostatně řešených úloh. Předvádí několik úloh, které autor při aplikaci této metodiky používá, a předkládá další náměty na příklady.

Klíčová slova: OOP, návrhové vzory, výuka programování, vstupní kurzy programování, object first, design patterns first

1 Úvod

1.1 Trocha historie

Programování se od svého vzniku vyvíjí stále bouřlivěji. V padesátých letech bylo považováno téměř za magii, kterou je schopno provozovat jen několik zasvěcenců. V průběhu šedesátých let se z něj postupně stávala inženýrská disciplína přístupná stále většímu okruhu lidí. Programování se začalo učit na univerzitách. Výuka programování se pomalu začala etablovat jako samostatná vědní disciplína, jejíž význam narůstal se zvyšujícím se objemem znalostí, které chtěli učitelé svým svěřencům předat.

Na přelomu šedesátých a sedmdesátých let byla při ACM (Association for Computing Machinery) založena organizace SIGCSE – Special Interest Group on Computer Science Education. Řada učitelů sice učila nadále intuitivně, ale paralelně vznikaly i první systematicky pojaté metodiky výuky. Konference SIGCSE se staly fórem, kde si učitelé mohli předávat své zkušenosti.

S příchodem osobních počítačů v osmdesátých letech nastal obrovský boom. Programování vtrhlo mezi mládež a řadové uživatele a začalo se učit nejenom na technických univerzitách, ale i na řadě středních škol, a to dokonce i v naší republice. V průběhu devadesátých let pak již definitivně opustilo oblast jakéhosi technického umění pro zasvěcené a stalo se běžnou inženýrskou disciplínou. Půjčím-li si názvosloví z jiného obory, mohli bychom říci, že přešlo od alchymie k chemii.

Změnilo se nejenom vnímání této disciplíny zbytkem populace, ale změnila se i disciplína sama. Klíčové rozdíly mezi programováním v druhé polovině sedmdesátých let, kdy jsem se učil programovat já, a programováním současným, jsem se pokusil shrnout v tabulce 1.

Tabulka 1. Programování dříve a nyní

Dříve	Nyní
Řada úloh stále čekala na vyřešení	Většina běžných úloh je vyřešena a řešení jsou dostupná v komponentách či knihovnách
Programy pracovaly samostatně, navzájem příliš nespolupracovaly	Nové program jsou téměř vždy součástí rozsáhlejších aplikací a rámců
Klíčovou úlohou programátora byl návrh algoritmů a základních datových struktur	Důležitější než znalost algoritmů se stává znalost knihoven a aplikačních rámců, v nichž jsou potřebné algoritmy a datové struktury připraveny Klíčovou úlohou je návrh architektury systému

1.2 Současný stav

Se změnou programů a úloh k naprogramování se samozřejmě výrazně změnila i používaná technologie. Dříve převládalo strukturované programování, které se narodilo v roce 1968 publikací slavného článku [1] a poměrně velmi rychle se prosadilo.

Objektově orientované programování, které dnes převládá, se sice narodilo o něco dříve než strukturované (jeho zrod je spojován se vznikem jazyka Simula 67), ale prosazovalo se mnohem pomaleji. Dlouho totiž trvalo, než se jeho propagátorům podařilo přesvědčit programátorskou veřejnost (a především programátorské firmy) o jeho výhodných vlastnostech. Intenzivněji se proto začalo prosazovat až v průběhu osmdesátých let a zcela ovládlo programátorský svět v průběhu let devadesátých.

Velkým impulsem k upevnění jeho dominantního postavení bylo na jedné straně používání návrhových vzorů a na druhé straně příchod jazyka Java a jeho masové nasazení ve vstupních kurzech programování na převážné většině univerzit. Nezanedbatelnou úlohu sehrály i nové metodiky vývoje programů označované souhrnně jako agilní a nesmíme zapomenout ani na rychle se prosadivší techniky automatického testování. O všem se na této konferenci v minulých ročnících hovořilo.

Podíváte-li se dnes na světové konference o výuce programování, zjistíte, že se už dávno nediskutuje o tom, zda k výuce používat Javu – používají ji téměř všichni. Hovoří se pouze o tom, jak programování co nejlépe naučit. Chce-li někdo doporučit, aby se ve vstupních kurzech programování masové využíval jiný jazyk, musí toto své rozhodnutí zdůvodnit. I tak však tyto příspěvky nemívají příliš velký ohlas.

Už se také dávno nediskutuje o tom, který z možných přístupů k výuce použít. Prakticky všichni vyučující se již shodli na tom, že mají-li si studenti osvojit způsob myšlení potřebný pro vývoj OO programů, musí si jej osvojovat od samého začátku výuky a ne na něj přecházet až někdy v jejím průběhu. V opačném případě budou mít problémy s potřebným „mentálním kotrmelcem“ a navíc tímto přeučováním ztratí zbytečně mnoho času. Jak bylo vtipně poznamenáno v [19]: *Object First! is a mantra for the first programming based computer science course.*

Obdobně panuje všeobecný souhlas s co nejčasnějším začleněním výkladu návrhových vzorů. Problém je pouze v tom, jak výklad návrhových vzorů do výuky začlenit a jaké použít příklady, na nichž by bylo možno tento výklad co nejlépe demonstrovat. Tím se také zabývá řada příspěvků na současných konferencích SIGCSE. Součástí těchto konferencí se posledních pět let staly dokonce semináře nazvané původně *"Killer Examples" for Design Patterns and Objects First workshop* a přejmenované letos na stručnější *"Killer Examples" for Design Patterns*. Tyto semináře byly i inspirací pro náš letošní seminář.

2 Metodika *Design Patterns First*

Jak jsem řekl, panuje všeobecná shoda o nutnosti co nejčasnějšího začlenění návrhových vzorů do výuky, nicméně návodů a ověřených postupů, které by ukazovaly, jak lze návrhové vzory začlenit do výuky již od prvních lekcí, zatím moc nenajdete.

Jednou z takovýchto metodik je metodika *Design Patterns First*, která vychází z doposud převažující metodiky *Object First* (viz např. [7], [9]), avšak mění uspořádání výkladu jednotlivých konstrukcí tak, aby bylo možno začít vykládat návrhové vzory vzápětí po prvních pokusech o vytvoření vlastního kódu (viz [11], [12]).

Obě metodiky se snaží respektovat známou zásadu, že u předmětů, u nichž nestačí se látku pouze naučit, ale je třeba si ji osvojit vyřešením řady úloh, je nutné začít učit základní myšlenkové postupy hned od začátku výuky. Jinak totiž nebudou mít žáci dost příležitostí si tyto principy osvojit, nehledě na to, že by se museli často odnaučovat leccos z toho, co se před tím naučili (viz např. [7], [14], [16]).

Metodika *Object First* doporučuje začít hned na počátku výuky prací s objekty, avšak nelpí již na tom, aby se současně s třídami vysvětloval od samého začátku také pojem rozhraní a aby se od samého začátku začleňoval do výuky i výklad návrhových vzorů. Domnívám se, že hlavním důvodem odsunutí výkladu rozhraní do pozdějších lekcí byl nedostatek dostatečně jednoduchých příkladů, na nichž by bylo možno vykládanou látku demonstrovat a které by bylo možno zadávat k samostatnému vypracování (ostatně proto se také konají semináře *"Killer Examples" for Design Patterns*).

Metodika *Design Patterns First* reaguje na zkušeností, že zejména studenti, kteří se před vstupem do kurzů OOP potkali se strukturovaným programováním, mají velké problémy s pochopením pojmu rozhraní v celé jeho komplexnosti a s jeho přirozeným začleněním do svých úvah při návrhu programů. Problémy se špatným chápáním rozhraní pak logicky vedou i k problémům s pochopením funkce a účelu návrhových vzorů. Chceme-li proto začlenit výklad návrhových vzorů na počátek výuky, musíme tam s ním začlenit i výklad pojmu rozhraní. Tato metodika proto studenty seznamuje s oběma pojmy vzápětí poté, co se naučí samostatně definovat jednoduchou třídu.

Aby bylo možno výklad takto uspořádat, bylo nutno připravit dostatečně jednoduché příklady, na nichž je možno již na počátku výuky vysvětlit koncepci a účel rozhraní tak, aby žáci cítili jeho zavedení jako opodstatněné. Současně s výkladem konstruktu rozhraní je pak možno zařadit i použití prvního návrhového vzoru.

V letošním školním roce se pokouším tuto metodiku paralelně aplikovat na Vysoké škole ekonomické v Praze na fakultě informatiky a statistiky (viz [3] – zde se ji pokouším aplikovat již druhým rokem) a na Střední odborné škole informatiky a podnikání v Praze (viz [4] – tam zkouším aplikaci této metodiky letos poprvé).

2.1 Prozatímní zkušenosti z VŠE

Při výuce na VŠE se musím alespoň rámcově synchronizovat s ostatními vyučujícími, kteří prozatím učí podle metodiky *Object First* lehce upravené o časnější zařazení výkladu rozhraní. Výhodou této synchronizace je možnost porovnání reakcí studentů po skončení semestru a jejich výsledků v následujících letech.

Studenti na VŠE se dělí do dvou zhruba stejně početných skupin. První skupina bere výuku programování jako nutné zlo potřebné k tomu, aby se probojovala do druhého ročníku, druhá skupina má o programování zájem a počítá s tím, že jeho znalost ve své budoucí profesi využije.

Na dvě skupiny se studenti dělí i při hodnocení výkladu. Jedni jsou poněkud rozčarováni tím, že nedostávají přesné návody jak to či ono řešit, druzí naopak oceňují, že je koncepce výuky bližší tomu, s čím se setkají ve své budoucí praxi.

Na porovnání výsledků v následujících letech je zatím příliš brzy.

2.2 Předběžné zkušenosti z SOŠIP

Na SOŠIP učím OOP ve třetím a ve čtvrtém ročníku. Výklad oproti VŠE nijak podstatně neměním, pouze probíhá poněkud pomaleji a je více prokládán příklady, které studenti nejprve řeší jako domácí úkol, pro nějž si v následující hodině předvedeme vzorové řešení.

Studenti čtvrtého ročníku již rok programovali a při prvním představení nového stylu programování byli mnozí z nich zmateni skutečností, že existuje úplně jiné programování, než jaké se doposud učili. Projevil se u nich známý problém, že vše, o čem se na hodině přednášelo, cítili potřebu interpretovat prostřednictvím toho, co se naučili v předchozích letech.

Studenti třetího ročníku jsou na tom poněkud lépe. Ve druhém ročníku se učili programovat pouze v Baltíkovi a nestačili ještě získat a zažít návyky, které by jim bránily akceptovat OOP.

Jak jsem již předeslal, zde výuka teprve začala, takže je na jakékoliv rozsáhlejší zkušenosti ještě brzy. Zatím lze pouze sledovat, že studenti, kteří neberou studium jako nutné zlo, vítají, že již po poměrně krátké době mohou dělat poměrně zajímavé programy, a naznačují, že jejich subjektivně pociťované pokroky v tomto předmětu jsou výrazně vyšší než u paralelně probíhajících předmětů zabývajících se jinými jazyky a přednášenými podle jiných metodik.

3 Zásady návrhu demonstračních příkladů a samostatně řešených úloh

Výuka programování je těžko uskutečnitelná bez dobrých příkladů, na nichž studentům předvádíme funkci a použití jednotlivých vysvětlovaných konstrukcí, ale současně jim s jejich pomocí také nepřímo vštěpujeme zásady správného programování a poskytujeme vzory, které budou moci ve své praxi aplikovat.

Ještě důležitější než výběr vhodných demonstračních příkladů je volba samostatně řešených úloh. Na nich totiž závisí efektivita celé výuky. Cílem výuky nemá být

pouze to, aby byl student schopen vyřešit nějaký jednoduchý AHA program. Naším cílem by mělo být postupně připravovat studenty na řešení úloh, s nimiž se setkají ve své budoucí praxi. A problémy, s nimiž se programátor potkává při řešení těchto úloh se často výrazně liší od těch, s nimiž se většinou setkával v průběhu výuky.

Jak jsem již naznačil v historickém úvodu, doba programování, jež jsme mohli prezentovat jako zvláštní druh technického umění, již z větší části skončila. Nastoupila doba programování, které se zařadilo mezi ostatní technologie.

Typickou dnešní úlohou již není vyřešit nějaký zapeklitý algoritmický problém, protože většina algoritmických problémů, s nimiž se můžeme setkat, jej již dávno vyřešena a naprogramována v nejrůznějších knihovnách. Typickou dnešní úlohou je sestavit a rozchodit poměrně rozsáhlý program, který musí navíc spolupracovat s dalšími, ještě rozsáhlejšími programy. Přitom je třeba tento program rozchodit v rozumném čase a s rozumnými náklady. Takové úlohy musíme naše studenty naučit řešit. Řešení algoritmických chuťovek je většinou výhodnější přenechat specialistům.

Projděme si hlavní zásady, kterými bychom se měli řídit při návrhu a přípravě úloh, na nichž bychom chtěli demonstrovat probíranou látku nebo které bychom chtěli studentům zadat k samostatnému vypracování. Řada z dále uvedených požadavků byla již publikována (viz např. [7], [12], [18], [22]), některé z nich jsou nové, často inspirované závěry či myšlenkami některých dalších prací.

Následující zásady bychom měli dodržovat nejenom při aplikaci metodiky *Design Patterns First*, ale většina z nich je aplikovatelná i na výuku podle jiných metodik včetně metodik, které nezačínají výukou objektů, tím méně návrhových vzorů.

3.1 Úlohy musejí být zajímavé

Všichni s touto zásadou hluboce souhlasí, ale stále najdeme řadu kurzů, v nichž se zejména v počátečních hodinách řeší úlohy typu *Sečtěte zadaná čísla a vytiskněte výsledek*, v pozdějších lekcích úlohy typu *Najděte nevětšího společného dělitele* atd. Studenti tak zbytečně získávají dojem, že programování je nudná disciplína a v době, kdy se konečně začnou objevovat zajímavější úlohy, již tento názor těžko mění.

Výše zmíněné úlohy měly jednu společnou výhodu: stačilo je pouze zadat. Naproti tomu u příkladů (a zejména pak u příkladů určených pro první hodiny výuky), které by mohly být zajímavé, musí učitel připravit značnou část projektu, aby to, co zbude, bylo nejenom zajímavé, ale také dostatečně jednoduché. V tom nám ale mohou pomoci různé rámce, o nichž se zmiňuje zásada 3.3.

3.2 Nezadávat projekty, jež je třeba vytvářet zcela od počátku

Mezi učiteli je velice oblíbené zadávání úloh, které studenti musejí vyřešit zcela od počátku. Takové úlohy však budou málokdy splňovat zásadu 3.1, protože zejména na začátku výuky nemáme při zadávání zajímavých úloh na čem stavět. Navíc, jak jsem již zmínil výše, takovéto úlohy nejsou zástupci typických zadání, s nimiž se studenti v praxi setkají.

Z výukového hlediska jsou mnohem výhodnější projekty, jejichž větší část je předem připravena a studenti mají za úkol definovat jenom zlomek celého projektu. V počátečních stádiích je dokonce výhodné, když mají za úkol pouze doplnit těla me-

tod v předpřipraveném zdrojovém kódu třídy. V dalších fázích pak mohou doplňovat celé metody a poté i celé třídy či skupiny tříd.

Úlohy, v nichž mají studenti za úkol připravit pouze část projektu, jsou pro studenty většinou zajímavější a také jim více přibližují typické situace jejich budoucí praxe, kdy budou mít (alespoň zpočátku) většinou za úkol doplnit nějaký existující projekt o novou funkčnost. Přitom složitost upravovaného projektu bude velmi často přesahovat aktuální schopnosti jeho modifikátorů.

Jedinou nevýhodou těchto úloh je, že jejich příprava je výrazně pracnější. Pracnost zadávání však být může podstatně snížena využitím rámců, o němž hovoří zásada 3.3.

3.3 Využívat rámců (frameworks)

Vymýšlení úloh „na zelené louce“ je nesmírně pracné a občas vyžaduje od učitele více času, než kolik má v daném období právě k dispozici. Pracnost přípravy projektu se výrazně sníží, bude-li mít učitel k dispozici nějaký vhodný rámec. Publikace [5] definuje rámec jako *sadu spolupracujících tříd, která definuje znovupoužitelný základ pro specifickou třídu programů*. Tato sada tříd může tvořit knihovnu (případně část knihovny), anebo se může jednat o celý projekt, který je v rámci řešení různých úloh doplňován o další funkčnost.

Naší třídou programů jsou úlohy, které předkládáme studentům k samostatnému řešení a po nichž chceme, aby splňovaly jisté požadavky (např. ty, které zde postupně rozebírám). Ze známějších rámců bych jmenoval např. modul s implementací robota Karla, knihovnu funkcí pro čaroděje Baltazara doplňující vývojové prostředí firmy SGP nebo miniknihovničku tříd pro používání grafiky, jejíž vývoj a následné použití je součástí výkladu v publikaci [15].

Řada dalších rámců a knihoven byla publikována na nejrůznějších konferencích. Jejich autoři však většinou podlehli představě, že publikovaný rámec musí být relativně univerzální a kompletní, a proto jsou jejich rámce podle mého názoru pro použití v prvních lekcích většinou neúměrně složité a potřebují další vrstvu odstínění (návrhový vzor *Fasáda* ☺).

Tvorba dalších rámců by mohla být poměrně vděčným námětem pro nejrůznější bakalářské a disertační práce. Takto vytvořené rámce mohou být dokonce vyvinuty paralelně pro platformy J2SE i J2ME, takže program vyvinutý a odladěný na stolním počítači může být vzápětí nahrán do mobilního telefonu, kde se s ním mohou žáci chlubit svým kamarádům (a tím zvyšovat svoji motivaci).

Vznikla i celá vývojová prostředí založená na nějakém rámci. Příkladem může být např. prostředí *Greenfoot* (viz [21]), které je však postavené na intenzivním využívání dědičnosti, před jejímž nadměrným používáním se v našich kurzech snažíme naopak varovat, a proto je přes jeho nespornou atraktivnost nepoužíváme.

Jiným zajímavým projektem je *Alice* (viz [20]). Toto vývojové prostředí bude asi mnohým připomínat známého Baltíka s tím rozdílem, že k zadávání příkazů se nepoužívají ikony, ale příkazy budoucího programu mají podobu formulářů se vstupními poli, kam zapisujeme volání metod, podmínky a další potřebné elementy.

Toto prostředí je vhodné pro učitele na středních školách, kteří nechtějí učit čisté objektové programování, ale chtěli by na ně žáky zajímavým způsobem připravit. Těžiště práce v tomto prostředí je, stejně jako u nového Baltíka, v definici metod.

Výhodou prostředí *Alice* je neexistence syntaxe, jejíž dodržení by bylo nutno sledovat. Obecně je toto prostředí považováno za vhodný předstupeň pro budoucí přechod na jazyk Java případně C++.

3.4 Umožnit tvorbu smysluplných programů již v prvních hodinách výuky

Máme-li naplnit tuto zásad, nesmíme čekat, až studenti získají dostatečné množství znalostí, aby mohli vytvářet složitější programy, ale musíme pro ně připravit i takové programy, které budou moci zpracovávat hned po prvních hodinách výuky.

Podaří-li se nám vymyslet úlohy, na jejichž vyřešení by jejich znalosti stačily a které by jim navíc připadaly zajímavé, výrazně tak od samého začátku podnítíme jejich zájem o předmět. Ve snaze dodržet tuto zásadu nám mohou výrazně pomoci rámce či vývojová prostředí zmiňovaná v bodě 3.3.

3.5 Projekty musí být dostatečně složité

Studenti, kteří si zvyknout na řešení jednoduchých úloh, bývají v praxi zaskočení složitostí projektů, které mají rozšířit či upravit. Je proto nanejvýš vhodné pracovat od samého počátku výuky s projekty, jejichž složitost výrazně překračuje míru aktuálních znalostí a dovedností studentů. (Je to jistá variace na téma: *Čím hloupější je uživatel, tím chytřejší musí být počítač.*)

Složitost celého projektu, jehož část mají studenti za úkol doplnit či upravit, má ještě jednu vedlejší výhodu: po vyřešení zadané parciální úlohy získají studenti pocit, že vlastně naprogramovali celý velký projekt, a tak také svoji práci často prezentují svým rodičům či kamarádům. To přirozeně výrazně zvyšuje jejich motivaci.

I složité úlohy je ale vhodné připravovat tak, aby se k nim mohli studenti v dalších etapách vracet a na základě nově nabytých znalostí posoudit části, které mohli dříve pouze používat, ale v žádném případě je nemohli navrhnout. Ještě lepších výsledků pak dosáhneme, když se nám podaří vymyslet takové projekty, k nimž se budou moci studenti vracet proto, aby je postupně doplňovali o další funkčnost.

Budou-li se s odstupem vracet ke svým dřívějším programům, které budou muset upravovat a vylepšovat, sami si mimo jiné postupně uvědomí i nutnost dodržování řady konvencí, protože jinak se pro ně stanou po čase jejich vlastní programy nečitelné a budou si zbytečně přidělávat práci jejich nutnou analýzou.

3.6 Projekty by měly přirozeně obsahovat návrhové vzory

Chceme-li, aby studentům přešly zásady moderního programování do krve, nestačí o nich jen přednášet, ale měli by se s jejich naplňováním setkávat i v zdáních svých úloh. Chceme-li, aby si osvojili práci s návrhovými vzory, měly by být návrhové vzory v zadávaných úlohách použity. Při dostatečně složitých projektech rozebíraných v bodu 3.5, však bývá tato zásada většinou přirozeně splněna.

Jak je správně poznamenáno v [22], hlavním problémem při snaze naučit studenty používat návrhové vzory nebývá složitost návrhových vzorů, ale samoúčelnost jejich použití. Chceme-li přesvědčit studenty o užitečnosti použití návrhových vzorů, musíme úlohy koncipovat tak, aby z nich užitečnost jejich použití pokud možno „odkapávala“.

Není důležité, abychom zaváděli vzory v celé jejich komplexnosti. V řadě případů stačí smysluplně použít zjednodušenou variantu vzoru. Smysluplné použití je daleko důležitější než seznámení s přesnou podobou vzoru. Studenti musí vstřebat základní ideu celé koncepce návrhových vzorů, aby se pro ně v další praxi stalo používání návrhových vzorů přirozenou součástí vývojového procesu a aby jim bylo především jasné, jaký je základní účel a hlavní výhoda jejich používání.

3.7 Projekty zadávané k samostatnému zpracování by měly obsahovat jednoduché vzorové řešení nebo alespoň jeho náznak

Studenti mívají v některých případech značné problémy s pochopením toho, co se od nich vlastně chce. Proto je vhodné jim připravit nějaké triviální vzorové řešení, které by jim mohlo sloužit jako prvotní inspirace a možná i jako výchozí bod k jejich vlastním řešením.

Toto vzorové řešení je vhodné použít i u samostatně řešených úloh, i když je mnozí studenti jednoduše převezmou a použijí jako základ vlastního řešení. Přiznejme si, že studium cizích programů je pro většinu programátorů jedním z velice cenných zdrojů poučení. Navíc můžeme těžko ověřit, jestli student vyvinul celé řešení zcela samostatně nebo jestli mu s ním někdo pomáhal.

Já řeším podobné problémy tak, že studentům dopředu říkám, že se nebudu pídit po tom, jak získali řešení, ale budu vyžadovat, aby se v odevzdaném programu vyznali tak dobře, jako kdyby jej naprogramovali sami. Při předávání svého řešení pak mají za úkol modifikovat celý program (např. přidáním další funkce), případně opravit v programu chybu, kterou jsem do něj bez jejich vědomí zanesl.

Už se mi několikrát stalo, že student sice neuměl dost dobře vysvětlit, proč tu či onu část programu naprogramoval právě takto, ale kdykoliv jsem za jeho zády program jakkoliv „poškodil“, bleskově se v něm zorientoval, chybu našel a opravil. Pochopil jsem z toho, že se dotyčný (stejně jako mnozí jiní začátečníci) ještě neumí zcela přesně vyjádřit, ale že se v programu vyzná, a odevzdanou práci jsem mu proto uznal.

3.8 Součástí odevzdané práce musí být i testy funkčnosti

Jednou ze zásad, kterou se snažíme studentům vštěpovat, je důležitost testů a apel na jejich tvorbu před započetí práce na vlastním kódu. Studenti by se měli naučit své práce nejenom vytvářet, ale také testovat.

U prvních úloh můžeme doplnit zadání vlastní třídou testující funkčnost odevzdaného řešení, aby si studenti hned od počátku uvědomili, nakolik se jim problém zjednodušuje, pokud budou testy hotové ještě před tím, než začnou pracovat na řešení úkolu a zdání se tak smrskne na požadavek vytvořit program vyhovující testům.

Bezpodmínečné vyžadování jednoznačného splnění všech testů zdůrazní důležitost a význam odevzdávání plně funkčního programu. Studentům, kteří oponují, že jejich program testy téměř prošel, namítám: *Program, který skoro chodí, je jako letadlo, které skoro létá*. O tom, že program musí bezezbytku projít všemi testy se prostě nediskutuje; je to nutná podmínka k přijetí odevzdaného programu.

U dalších úloh bychom měli po studentech vyžadovat, aby definovali vlastní testy a při odevzdávání práce návrh těchto testů obhájili a předvedli, že jejich řešení těmito testy projde.

Ještě lepším řešením je rozdělit odevzdávání práce do dvou etap: v první etapě budou mít za úkol připravit, odevzdat a obhájit testy svého budoucího řešení a v druhé etapě pak bude jejich úkolem vytvořit program, který těmito testy projde.

Testy se tak stanou neoddělitelnou součástí jejich práce a jako takové budou i hodnoceny. Navíc toto dělené odevzdávání práce umožní vyučujícímu s dostatečným předstihem upozornit studenty na nedotaženost jejich řešení nebo naopak na potenciální budoucí problémy, které jim při vývoji příliš složitě pojatého řešení hrozí.

3.9 Projekty by měly využívat GUI

Demonstrační příklady i samostatně řešené úlohy by měly využívat přiměřeně komfortní GUI, a to přesto, že nechceme-li v dané etapě kurzu učit studenty přímo práci s GUI, měli bychom je od ní osvobodit, aby se mohli soustředit na vlastní problém.

Studenti jsou na jistou úroveň komfortu obsluhy aplikací zvyklí a jejich motivaci výrazně zvyšuje skutečnost, že i jejich začátečnické aplikace jsou schopny jakýsi srovnatelný komfort nabídnout.

Není-li cílem výuky přímo tvorba GUI (ta se většinou učí až v pokročilejších kurzech), mělo by být GUI součástí té části projektu, kterou připravuje vyučující, nebo by měla být studentům poskytnuta nějaká knihovna, která bude na jednu stranu dostatečně jednoduchá, aby jí dokázali snadno zvládnout a pracovat s ní, avšak na druhou stranu bude dostatečně bohatá na to, aby pokryla jejich očekávatelné požadavky.

V tomto směru mohou být naším užitečným pomocníkem správně vybrané rámce, které buď danou funkci obsahují, anebo jsou určené k řešení úloh, jež se bez ní obejdou.

3.10 Projekty by měli nabízet prostor pro vlastní tvořivost

Příliš striktně zadané projekty s jednoznačnými řešeními studenty dostatečně nemotivují nehledě na to, že je příliš vybízí k tomu, aby řešení jednouše převzali od svých spolužáků. Projekty, v nichž mohou studenti projevit vlastní tvořivost, je daleko víc zaujmou a navíc mají daleko větší zábrany projekt jednoduše opsat, protože s rostoucí odlišností jednotlivých řešení současně roste i pravděpodobnost, že takováto kopie bude odhalena.

Obzvlášť velký úspěch mají mezi studenty projekty, v nichž mohou svému vyučujícímu nebo spolužákům, kteří jejich projekty testují, připravit nějaké překvapení.

U projektů s vysokou variabilitou řešení zdánlivě neúměrně rostou nároky na jejich následné testování. Využijme-li však možnosti prvotního odevzdání testů následované odevzdáním úlohy, která musí těmto testům vyhovět tak, jak to bylo rozebráno v oddílu 3.8, můžeme s minimálním úsilím úspěšně otestovat i úlohy s obrovskou variabilitou (viz např. diskuse u příkladu v pasáži 4.7 Adventura).

3.11 Projekty by neměly být zašuměné

Podle bodu 3.5 by sice projekty měly být složité, avšak na druhou stranu je třeba dbát na to, aby nebyly složité zbytečně, tj. abychom nenutili studenty vkládat rozsáhlejší pasáže, které by zadávaly více méně mechanicky a nic podstatného by se při jejich zadávání nenaučili.

Ve snaze přiblížit zadávané úlohy co nejvíce reálnému životu používají autoři některých učebnic a kurzů projekty, v nichž je kód, jehož vytvořením se studenti doopravdy něco naučí, pouze malým zlomkem kódu, který musí vytvořit. Veškerý takový kód považuji za šum, který na jednu stranu odvádí pozornost studentů od probíraných konstrukcí a na druhou stranu díky své pracnosti snižuje jejich motivaci.

Řešení, která budou splňovat zásady 3.1 a 3.5 (zajímavé a složité) bude určitě obsahovat značnou část kódu, která nepodporuje přímo procvičování probírané látky. Tuto část kódu je proto třeba „vytěsnit" do té části projektu, kterou připravuje vyučující, a nezatěžovat s ní studenty. Navíc každé odhalení toho, že jim vyučující nějakým obratem ušetřil práci, kvitují studenti s povděkem (tedy alespoň ti, kteří to zaregistrují) a nepřímo jim tak zvýšíme motivaci ke zdárnému dokončení úkolu.

4 Příklady úloh

4.1 Testovací třída

Při aplikaci metodiky *Object First* nezadávali vyučující první hodinu žádný domácí úkol (leda instalovat doma *BlueJ* či jiné použité prostředí), protože si neuvědomovali, že od verze 1.5 *BlueJ* podporuje interaktivní definici testovacích tříd a metod. Zaslání takto definované třídy je ideální příležitostí, jak si mohou studenti doma procvičit práci s třídami, jejich instancemi a současně i definici testovacích tříd a metod spolu s definicí testovacích přípravků.

4.2 Světlo – Přejezd – Semafor

Příklad s blikajícím světlem, který vede k zavedení návrhového vzoru *Služebník*, byl představen již několikrát (viz [11], [12], [13]). Jeho základní výhodou je, že studentům, kteří již mají s programováním nějaké zkušenosti, neumožňuje použít cyklus, protože pak instance nepracují tak, jak pracovat mají.

Na základě tohoto příkladu definujeme na hodině ještě třídu *Přejezd*, která simuluje chování výstražných světel u železničních přejezdů a za domácí úkol pak studenti dostanou definici semaforu, který bude procházet jednotlivými stavy bez ohledu na ostatní dění.

4.3 UFO

Příklad, který v publikaci [15] uzavírá kapitolu, v níž se čtenáři seznamují se syntaxí definice tříd a jejich atributů a metod. Tento příklad ukazuje, jak je možno připravit zadání, v němž studenti dostanou předpřipravenou třídu s metodami s prázdnými těly a dokumentačními komentáři, které specifikují požadované chování dané metody.

4.4 Výtahy

Příklad, na němž je v publikaci [15] předváděno řešení složité úlohy postupnou definicí metod, které krok za krokem rozšiřují schopnosti vytvářených objektů. Celé řeše-

ní je opět postaveno na předem připravených testech, které jsou v průběhu návrhu programu postupně jeden za druhým „zprovozňovány“. Navíc se jedná o příklad, jenž je možno dále rozšiřovat – viz [8].

4.5 Auto a dopravní hřiště

Příklad, který používám u středoškoláků a zájmových kroužků, kde je třeba poněkud intenzivněji procvičovat probranou látku. Studenti mají za úkol definovat vlastní auto s dvěma světly, s nimiž již mají zkušenosti z tvorby semaforu.

Své auto budují postupně. Nejprve mají za úkol definovat jenom konstruktory, poté dostanou za úkol implementovat zadané rozhraní a tím pádem doplnit některé metody. Postupně je vylepšují tak, aby bylo na konci schopno reagovat na povely zadávané z klávesnice (čtení klávesnice obstarává část programu definovaná učitelem; auta musí pouze umět reagovat na zprávy vyvolané stisky jednotlivých tlačítek). Když mají svá auta takto připravena, mohou s nimi začít „závodit“ na připravené trati.

Inspirací je pro ně v každém kroku vzorové řešení jednoduchého auta. Cílem zadání je, aby si postupně procvičili práci s atributy, implementaci rozhraní a ujasnili si, jak pomocí implementace zadaného rozhraní dosáhnou toho, že jejich auto bude schopno komunikovat se zbytkem projektu a časem i s ostatními auty.

V budoucnu mám v plánu vytvořit se studenty postupně celé dopravní hřiště, kde by jejich auta jezdila podle zadaných pokynů a přitom respektovala dopravní značení a semafory. „Učitelská část“ projektu je ale zatím pouze ve stavu zrodu a čeká buď na časový prostor, nebo na nějakého diplomanta.

4.6 Kalkulačka

I tento projekt je popsaný již v [11], [12], [13], takže se o něm opět zmíním jen stručně. Studenti se na něm učí používání algoritmických konstrukcí a základní práci s jednoduchými kontejnery.

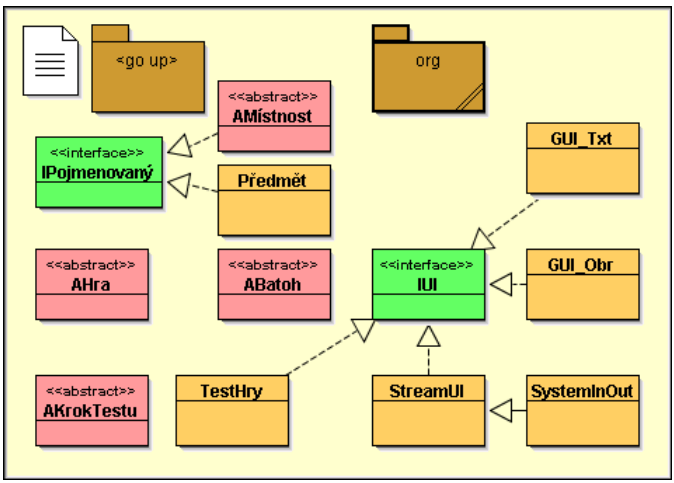
Všechna řešení implementují shodné rozhraní. Studenti dostanou předem testovací třídu, jejíž instance jejich programu zeptá na jeho schopnosti a pak je otestuje. To umožňuje vyučujícímu definovat třídu, která ve složce projektu vyhledá všechny soubory představující přeložené třídy implementující dané rozhraní, a všechny je hromadně otestuje nezávisle na tom, jaké zadání ta která třída řeší. Studenti si tak mohou znovu uvědomit, jak mocným nástrojem rozhraní je.

4.7 Adventura

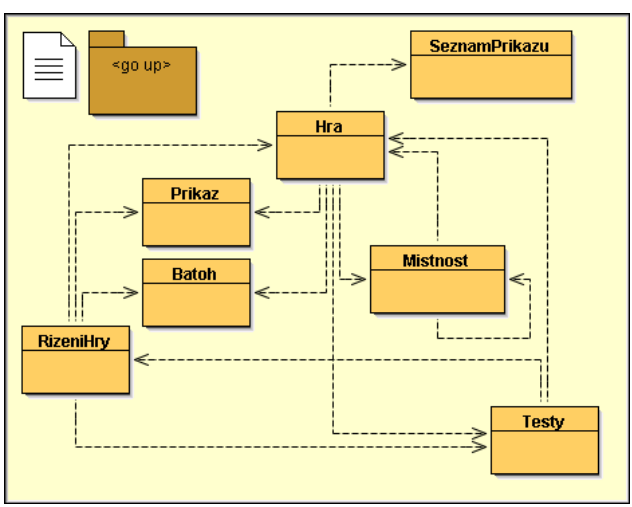
Zadání je variací na původní zadání publikované v [9] a [10]. Úkolem studentů je vytvořit komunikační hru, při níž je úkolem uživatele dosáhnout jakéhosi cíle (osvobození princezny, nalezení pokladu, vyloupení banky, …). Uživatel zadává textové příkazy, na které program reaguje. Cestou prochází různými „místnostmi“ a sbírá užitečné předměty, které mu mají pomoci v dosažení požadovaného cíle.

Mnou používané zadání se od zadání publikovaného v [9] a [10] liší v tom, že studenti nemají za úkol vytvořit zcela vlastní aplikaci, ale mají za úkol vytvořit ve vlastním balíčku aplikaci, která vyhovuje předem zadanému rámci, tj. aplikaci, jejíž

klíčové třídy jsou potomky předem zadaných abstraktních tříd a implementují předem zadaná rozhraní (viz obr. 1).



Obr. 1. Rámec, do kterého mají studenti začlenit své řešení



Obr. 2. Vzorové řešení velmi jednoduchého scénáře

Účelem této modifikace je opět možnost poskytnutí rozumně komfortního GUI pro komunikaci uživatele s hrou a především pak možnost jednotného testování všech aplikací (a z toho plynoucí prudká úspora času vyučujícího).

Tato úloha je právě z těch, které jsou rozděleny na dvě části: testy a vlastní řešení. První část úlohy je definice sady testů, v nichž studenti nabídnou k oponentuře scénář své hry. Procházení hrou musí být definováno jako sada testovacích kroků představujících jednotlivé akce uživatele a požadované reakce jejich aplikace. Každý z testovacích kroků musí být instancí třídy, která je potomkem abstraktní třídy *AKrokTestu*, v níž je specifikována forma, jakou se jednotlivé informace zadávají.

Při předkládání testu k oponentuře mohu odevzdávané testy „předhodit“ metodě, která na základě těchto testů nasimuluje testovaný průběh hry. Při diskusi nad navrhovaným průběhem mohu studenty upozornit např. na to, že svůj úkol pojali příliš jednoduše, anebo naopak příliš složitě.

V druhé etapě pak mají studenti za úkol navrhnout odevzdat balíček s řešením, které musí vyhovovat dříve odevzdaným testům. Při vývoji celé hry mohou používat libovolné z řady uživatelských rozhraní, která jsou součástí základního rámce. Všechny třídy realizující uživatelská rozhraní přitom implementují rozhraní *IUI*, takže pro vlastní hru mezi nimi není žádný viditelný rozdíl. Ten pozná až uživatel. Některá z připravených rozhraní komunikují s uživatelem v grafickém okně, jiná prostřednictvím standardního vstupu a výstupu, další pořizují žurnál průběhu hry do souboru.

Důležitou vlastností rámce je, že za uživatelské rozhraní se vydává i třída *TestHry*, která celou aplikaci testuje. Studenti tedy mohou v průběhu vývoje libovolně přecházet mezi jednotlivými rozhraními a svým testy. Takto koncipované zadání opět výrazně zjednodušuje testování a vyhodnocování studentských prací při jejich odevzdání.

Součástí zadání je i balíček *org*, v němž mohou studenti najít vzorové řešení velice jednoduché hry a v němž si také mohou ujasnit některé z požadavků zadání.

5 Závěr

Příspěvek ukázal základní pohnutky k zavedení metodiky *Design Patterns First* a seznámil s jejími klíčovými vlastnostmi a doporučeními. V druhé části pak nastínil problematiku zadávání příkladů a samostatně řešených úloh a předvedl několik úloh, které autor používá ve výuce. Ukázal, že výklad rozhraní i návrhových vzorů lze bez větších těžkostí zařadit na samý začátek vstupních kurzů programování a předvedl i některé výhodné vlastnosti takovéhoto postupu, které zjednoduší přejímání řešení od studentů a jejich testování.

Reference

[1] DIJKSTRA Edsger W.: Go To Statement Considered Harmful, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148

[2]

[3] Doprovodná stránka k výuce na VŠE: *http://vyuka.pecinovsky.cz/vse*

[4] Doprovodná stránka k výuce na SOŠIP: *http://vyuka.pecinovsky.cz/sosip*

[5] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. (Přeloženo: [6]) ISBN 0-201-30998-X.

[6] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.* Praha: Grada, © 2003. 386 s. (Překlad (nepovedený) [5]). ISBN 80-247-0302-5.

[7] KÖLLING Michael, ROSENBERG John: Guidelines for Teaching Object Orientation with Java. *The Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001)*. Canterbury, 2001.

[8] NESLON Chris, WELLS Barbara: Developing an Elevator Control System. *"Killer Examples" for Design Patterns and Objects First Worshop* OOPSLA 2002. *http://www.cse.buffalo.edu/faculty/alphonce/KillerExamples/OOPSLA2002*

[9] PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš: Zkušenosti s přístupem object-first v úvodním kurzu programování. *Objekty 2005 – sborník příspěvků devátého ročníku konference*, VŠB Ostrava, 2005. ISBN 80-248-0595-2.

[10] PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš: *Úvod do Javy*. Vysoká škole ekonomická v Praze, Nakladatelství Oeconommica © 2005. ISBN 80-245-0963-6.

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] PROULX Viera K., RAAB Jeff, RASALA Richard: Objects from the Beginning – with GUIs. *ITiCSE 2002*. ACM, ISBN 1-58113-499-1.

[20] *Vývojové prostředí Alice* – domovská stránka *http://www.alice.org*.

[21] *Vývojové prostředí Greenfoot* – domovská stránka *http://www.greenfoot.org*

[22] WICK Michael R.: Teaching Design Patterns in CS1: a Closed Laboratory Sequence based on the Game of Life. *SIGCSE'05*, ACM, ISBN 1-58113-997-7.

Annotation

The efficiency of education of programming significantly depends not only on selected methodology, however also on quality of examples we use for demonstration of explained construction, and on selection of tasks, which should students solve independently. The paper collects reasons leading to introducing the methodology *Design*

Patterns First. Then it discuses problems the rules, which we should bear in mind by selection of assignments. It shows some examples and tasks, which the author uses in his lessons based on the *Design Patterns First* methodology.