METODIKA DESIGN PATTERNS FIRST A VYHODNOCOVÁNÍ STUDENTSKÝCH ÚLOH

Rudolf Pecinovský
Amaio Technologies, Inc., Třebohostická 14, 100 00 Praha 10;
Katedra informačních technologií VŠE Praha
rudolf@pecinovsky.cz

ABSTRAKT:
Jednou z poměrně neoblíbených činností při výuce programování je vyhodnocování domácích úkolů a samostatně řešených úloh. Jedná se totiž o činnost časově náročnou a přitom většinou nepříliš zajímavou. Příspěvek ukazuje, jak je možno při výuce podle metodiky *Design Patterns First* vyhodnocování odevzdaných úloh do značné míry automatizovat. Nejprve shrnuje základní charakteristiku metodiky *Design Patterns First* a ukazuje, proč použití této metodiky usnadňuje zadávání a zejména pak vyhodnocování studentských úloh. Současně seznamuje s „mikroknihovnou“, kterou autor pro automatizované vyhodnocování odevzdaných programů používá. V další části pak na příkladu tří úloh ze vstupního kurzu programování v jazyku Java ukazuje, jak je možno vyhodnocování úloh výrazně zefektivnit.

KLÍČOVÁ SLOVA:
Design Patterns First, Object First, OOP, výuka programování, objektově orientované programování.

1 ÚVOD

Příprava zadání studentských úloha a zejména pak jejich následné vyhodnocování patří pro mnohé z nás k těm méně příjemným složkám výukového procesu. Budou-li mít všichni studenti shodné zadání, hrozí nebezpečí, že si navzájem předají jeho řešení, které pak ti bystřejší z nich ještě několika hromadnými záměnami formálně modifikují, aby shodnost programů nebila tak do očí. Budou-li mít různá zadání, přidělváme si práci s jejich následných vyhodnocováním.

Standardní vyhodnocovací postup bývá poměrně pracný a často nudný: vyučující musí každý program spustit, předložit mu nějaká testovací data a ověřit, že program reaguje podle požadavků. Už jenom samotné zjištění faktu, jestli odevzdané řešení pracuje podle požadavků zadání, tak většinou vyžaduje věnovat každému z řešení čas počitatelný v minutách. Stáhnout čas věnovaný jednomu řešení na nějaký dostatečně malý zlomek minuty se nám podaří spíše výjimečně.

Někteří vyučující se proto snaží vymýšlet úlohy, které se budou následně dobře vyhodnocovat, ale tato snaha většinou končí u toho, že úloha někam vypíše jakési hodnoty, které pak „zkontrolujeme očima“. Neustále zůstává nutnost každý jeden program spustit a někam se podívat, k jakým výsledkům dospěl. Doba vyhodnocování proto bývá lineární funkcí počtu vyhodnocovaných prací.

Mnohé jistě napadlo, že by si programátor měl přece umět často opakovanou a víceméně rutinní činnost zautomatizovat. Problém je v tom, že při klasických podobách zadání, která známe z nejrůznějších učebnic a kurzů, je takováto automatizace rozumně řešitelná pouze u zadání vyžadujících vytvoření jednoho zdrojového souboru s programem používajícího pouze standardní vstup a výstup. Jakmile budeme po studentech chtít něco složitějšího, začneme mít s automatizací vyhodnocování odevzdaných řešení problémy.

Jak jsem však naznačil v anotaci, při použití metodiky *Design Patterns First* dostáváme již na počátku výuky do rukou prostředky, které nám umožní relativně snadno automatizovat i řešení poměrně složitých zadání. Podívejme se proto na tyto prostředky a zmíněnou metodiku podrobněji

2 ZÁKLADNÍ PRINICIPY METODIKY *DESIGN PATTERNS FIRST*

Metodika *Design Patterns First* vznikla proto, že ostatní metodiky výuky nebyly schopny dodržet některá základní pedagogická pravidla, především pak pravidlo označované jako *Pravidlo ranního ptáčete* (*Early Bird Pedagogical Pattern*), které vyžaduje takové uspořádání výuky, aby se klíčová témata probírala co nejdříve.

2.1 Rohraní (`interface`)

Jedním ze základních principů moderního programování je např. pravidlo *programovat proti rozhraní a ne proti implementaci*. Při jeho aplikaci v jazyku Java se intenzivně využívá programová konstrukce **`interface`**, jejíž výklad však ostatní metodiky zařazují až někam ke konci výuky, takže studenti pak nemají dostatek příležitostí si používání dané konstrukce osvojit a už vůbec ne je zažít a naučit se je zakomponovávat do svých návrhů.

Metodika *Design Patterns First* zařazuje výklad této konstrukce do počátečních hodin výuky. Časné začlenění výkladu rozhraní poskytne studentům nejen dostatek příležitostí a prostoru se s ním „důvěrně seznámit“, ale současně poskytne vyučujícímu při zadávání úloh rozsáhlé možnosti, které jsou při používání ostatních metodik větší část kurzu nedosažitelné.

2.2 Návrhové vzory

Dalším výrazným rysem současného programování je intenzivní používání návrhových vzorů. Návrhové vzory se do povědomí širší programátorské veřejnosti dostaly až v druhé polovině devadesátých let díky vydání publikace [3], avšak jejich akceptace proběhla nesmírně rychle a v současné době se jejich používání zařadilo mezi klíčové programátorské dovednosti.

Co nejčasnější začlenění výkladu návrhových vzorů a jejich intenzivní používání jak v demonstračních příkladech, tak v úlohách zadávaných k samostatnému zpracování, se proto stalo druhým klíčovým principem popisované metodiky. Tento princip považovali autoři dokonce za natolik důležitý, že dal název celé metodice.

2.3 Vývoj řízený testy

Třetí zásadou, kterou se tato metodika snaží studentům vštěpovat od samého počátku výuky, je důležitost testů a výhodnost jejich definice ještě před tvorbou vlastního programu. Na počátku výuky proto vyučující zadává studentům úlohy prostřednictvím testů výsledných řešení. Později pak bývá prvním etapou řešení příprava testů a další etapa řešení je odstartována až po jejich odsouhlasení vyučujícím.

Při tomto přístupu se tak studenti velmi záhy setkají s výhodami programování, jehož jediným cílem je vyhovění předem připraveným testům. To zvyšuje pravděpodobnost, že si tento progresivní způsob práce osvojí a budou jej používat i ve své budoucí praxi.

3 VYUŽITÍ TĚCHTO PRINCIPŮ PŘI NÁVRHU A VYHODNOCOVÁNÍ ÚLOH

Podívejme se nyní, jak se včasný výklad výše popsaných principů může promítnout do návrhu úloh zadávaných za domácí úkol nebo k samostatnému řešení v hodinách. Začnu svým oblíbeným výrokem, který studentům velmi často opakuji: *Program, který skoro chodí, je jako letadlo, které skoro létá*. Jinými slovy: u odevzdaných prací má smysl hodnotit pouze zpracování těch funkcí, které student rozchodil. Není důležité, jak rozsáhlou část zadání student rozpracoval. Jediné, co může student při předávání práce obhajovat, je ta část zadání, kterou dotáhl do funkční podoby.

V počátečních hodinách zadávám studentům úlohy tak, že dostanou testovací třídu s předem připravenou sadou testů. Jejich úkolem je definovat třídu, která implementuje předem zadané rozhraní (**`interface`**) a jejíž instance projdou testy v obdržené testovací třídě.

Tím, že instance vytvořené třídy projdou připravenými testy, sice studenti prokáží svoji schopnost vyřešit zadanou úlohu, ale vyhodnocování odevzdaných řešení vyučujícímu příliš neusnadní. Zůstává stále povinnost „prohnat“ každé z odevzdaných řešení předepsanými testy a ověřit, nakolik je splňuje.

Vyhodnocení odevzdaných programů bychom mohli zautomatizovat tak, že bychom studentům předepsali způsob pojmenovávání odevzdaných tříd a připravili program, který takto pojmenované třídy natáhne, vytvoří jejich instance, a chování vytvořených instancí vyhodnotí.

Jistým problémem je podivuhodná neschopnost studentů dodržet jakékoliv konvence, které nekontroluje překladač nebo testovací program. Museli bychom tedy do testovacího programu zabudovat nějaký mechanizmus, který bude vedle funkčnosti programu testovat i dodržení předepsaných konvencí. Přiznejme si však, že tento úkol je poměrně snadno řešitelný.

Automatické vyhodnocení odevzdaných řešení se však výrazně zjednoduší ve chvíli, kdy jedním z požadavků úlohy bude implementace nějakého rozhraní. Pak totiž nebudeme muset řadu věcí kontrolovat my, protože je za nás zkontroluje překladač, a my se budeme moci soustředit pouze na vyhodnocení vlastního řešení.

Pro automatické vyhodnocování úloh používám mikroknihovnu obsahující jedno rozhraní a jednu třídu:

- Rozhraní testujících tříd `ITest<I>` charakterizuje třídy, v nichž definujeme testy odevzdaných řešení. Rozhraní požaduje po třídě, která je implementuje, aby její instance implementovaly metodu `testuj(I)`. Ta ve svém parametru obdrží testovanou instanci třídy implementující rozhraní `I`, kterou má otestovat, prověří ji a zapíše někam zprávu o výsledku tohoto testování.
- Vyhledávací třídu `Tester`, jejíž instance mají za úkol vyhledat všechny objekty, které je třeba otestovat, a předat je k otestování. Konstruktor této má dva parametry:
  - instanci testující třídy, která je schopna obdržené objekty otestovat, a
  - class-objekt rozhraní, které mají implementovat testované instance.

  Vyhledávání instancí k otestování spustíme zavoláním metody `prověř(?)`, které předáme buď lokaci balíčku s testovanými třídami, nebo class-objekt libovolné třídy, která se v tomto balíčku nachází (může jí být např. testující třída). Metoda projde zadaný balíček a jeho podbalíčky, vyhledá v nich všechny třídy implementující zadané rozhraní, od každé vytvoří její instanci (v současné verzi pracuje pouze s instancí vytvořenou bezparametrickým konstruktorem), a tu pak předá instanci testující třídy, kterou obdržel konstruktor této vyhledávací instance.

Vytvoření testovací třídy je tak jedinou činností, která zabere trochu času. Studenti umístí svá řešení do zadaného balíčku (případně do jeho definovaných podbalíčků) a nám už jen zbývá spustit připravený program, který vše automaticky prověří a vyhodnotí. Vlastní vyhodnocení bývá většinou otázkou několika vteřin, v případě většího počtu studentů a složitějšího vyhodnocování nejvýše několika málo minut.

4 VARIANTNÍ ZADÁNÍ A IDENTIFIKACE AUTORA

Je známou skutečností, že jednotně zadaná úloha studenty přímo vybízí k tomu, aby ti zkušenější předávali výsledná řešení svým méně zkušeným kolegům. Proti tomuto oblíbenému nešvaru je však možno se jednoduše bránit tím, že každý student dostane zadání, které se bude od ostatních zadání drobně lišit. Společné pro všechny zadání bude pouze implementované rozhraní.

Na první pohled se může zdát, že různost zadání bude ztěžovat vyhodnocení správnosti výsledného řešení. Při vhodně volených úlohách je ale tato komplikace zanedbatelná. Jednou z možností, jak různost zadání řešit, je přiřadit každému zadání nějaký identifikátor. V povinně implementovaném rozhraní pak deklarujeme metodu, která vrátí identifikátor řešeného zadání. Naši testovací třídu pak můžeme doplnit pomocnou třídou, jejíž metodě předáme identifikátor zadání, a ona nám vrátí sadu testů, prostřednictvím níž máme dané řešení prověřit.

Obdobně můžeme řešit identifikaci autora. Povinně implementované rozhraní může deklarovat metodu `getAuthor()`, která vrátí jméno nebo nějaký jiný standardizovaný identifikátor autora. Prostřednictvím tohoto identifikátoru můžeme autora nejen identifikovat, ale také zjistit zadání, které mu bylo přiděleno (např. zavoláním výše zmíněné metody).

5 PŘEBÍRÁNÍ CIZÍCH ŘEŠENÍ

Nyní na chvíli odbočím od původního tématu a dotknu se otázky přebírání řešení od kolegů, resp. objednávání si hotových řešení od přátel či dokonce za úplatu. Přiznejme si, že zabránit studentům přebírat cizí řešení, je poměrně obtížné a u domácích úkolů prakticky nemožné. Zdá se mi proto výhodnější se s touto možností smířit a upravit způsob odevzdávání úloh tak, aby přiměl studenty, aby si odevzdávané řešení alespoň prostudovali.

Já to řeším tak, že veřejně vyhlásím, že je mi jedno, jestli student odevzdávané řešení vytvoří nebo si je nechá vytvořit (když už to neumím zabezpečit), ale budu po něm chtít, aby se v odevzdaném řešení vyznal tak, jako kdyby je napsal sám. Při odevzdávání proto na studenta čeká nějaký drobný úkol jak odevzdávanou úlohu modifikovat.

Vysvětlím studentům, že praktický každý programátor jednou za čas použije ve svém programu řešení někoho jiného, ale pouze nezodpovědný jedinec začlení do svého programu modul, jehož fungování nechápe, ale bude za něj ručit a zabezpečit je i při budoucích modifikacích programu.

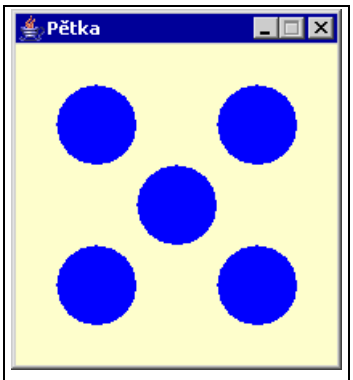
Zkušenost ukázala, že řada studentů se domnívá, že když si nechá program udělat a z vysvětlování autora získá pocit, že pochopila jeho funkci, tak že pak budou schopni program samostatně upravit. Všichni ale víme, že tomu tak není. Každoročně se pokaždé vyskytne několik studentů, kteří přinesou vypracovanou úlohu, a když jim potom drobně upravím zadání nebo naopak pokazím odevzdané řešení, s údivem zjistí, že program, o němž se ještě včera večer domnívali, že mu rozumějí, je pro ně nyní záhadným bludištěm, které nejsou schopni modifikovat ani opravit.

6 PŘÍKLADY

Vraťme se ale k zadávání a vyhodnocování úloh a podívejme se na několik příkladů takovýchto zadání.

6.1 Pětka

Prvním úlohou je vytvoření jednoduchého grafického objektu sestávajícího z několika jiných grafických objektů. Takovýmto grafickým objektem může být např. podoba „pětky“ na běžné házecí kostce používané při různých deskových hrách. Studenti dostanou tři „třídy“:



Obr. 1: *Pětka*

- Rozhraní, které má jejich řešení implementovat a které deklaruje jednoduché metody na změnu pozice a rozměru vytvořené instance plus výše zmíněné metody vracející identifikační kód autora a/nebo jeho jméno.
- Knihovní třídu pro generování zadání, jež nabízí metodu, které dodáme identifikační kód studenta a ona nám vrátí jeho verzi zadání.
- Testovací třídu, která otestuje, že vytvořená třída správně implementuje všechny metody deklarované v rozhraní.

Knihovní třída pro generování verzí je jednoduchá: na základě heš-kódu zadaného identifikátoru studenta doplní do předpřipraveného zadání barvy objektů, rozměry výsledného obrazce, jeho počáteční pozici a typ použitých obrazců (elipsy, obdélníky, trojúhelníky). Odchylky mezi jednotlivými zadáním jsou sice jen drobné, ale pro daný účel dostatečné.

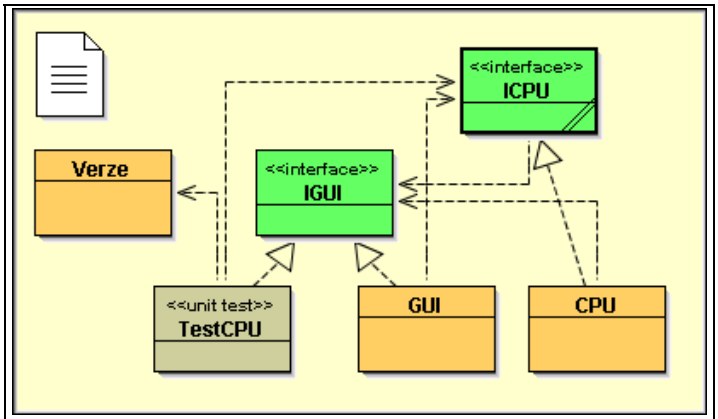
6.2 Kalkulačka

Jako druhý příklad uvedu svoji oblíbenou kalkulačku. Protože jsem se zde o této úloze při různých příležitostech již několikrát zmiňoval (nejpodrobněji je daná úloha vysvětlována asi v [7]), projdu její zadání jen ve stručnosti.

Studenti mají za úkol vytvořit součást projektu, který bude pracovat jako jednoduchá kalkulačka. Jejich konkrétním úkolem je definovat třídu, která by implementovala rozhraní `ICPU` a realizovala v celém projektu výpočetní jednotku.

Ostatní části projektu připraví vyučující. Mezi nimi je nejenom třída `GUI`, která má na starosti grafické uživatelské rozhraní, ale také třída `TestCPU`, která ověří, že výsledná úloha splňuje podmínky zadání, a třída `Verze`, která slouží jako generátor zadání a současně testovacích kroků k danému zadání.

V tomto případě se zadání negeneruje pouze na základě identifikačního kódu studenta, ale studenti si mohou zvolit hladinu obtížnosti, na kterou si ještě troufají. Následně je jim přiděleno některé ze zadání dané hladiny obtížnosti. (Za vyřešení obtížnějšího zadání získají samozřejmě více bodů.)



Obr. 2: *Diagram tříd projektu Kalkulačka*

Když student zná pořadové číslo svého zadání, může se zeptat třídy `Verze`, co vše má podle daného zadání jeho CPU umět. Pak příslušné funkce naprogramuje. Přitom má neustále k dispozici třídu `TestCPU`, kterou může kdykoli požádat, aby ověřila, nakolik je jeho řešení správné. Instance testovací třídy postupuje tak, že se studentovy třídy zeptá na číslo realizovaného zadání, pak se zeptá třídy `Verze`, co vše musí dané zadání umět a současně ji „poprosí“ o sadu testovacích kroků k ověření správného vyřešení úlohy. Pak se na studentovu CPU obrátí v roli GUI, předkládá jí jednotlivá „uživatelská“ zadání (stisky tlačítek) a testuje, jestli odpovědi CPU, tj. texty, které se mají vykreslit na displej, odpovídají požadovaným.

Po vyřešení úlohy studenti odevzdají svá řešení (pouze třídu implementující rozhraní `ICPU`) na zadané místo, já je všechna zkopíruji do balíčku, v němž už mám připravené ostatní třídy projektu spolu se svojí vyhledávací a testovací sadou. Testovací třídu pak požádám, aby ve spolupráci s vyhledávací třídou hromadně prověřila všechna řešení. Během minuty je všech 60 řešení otestovaných a protokol uložen do zadaného textového souboru.

Studenti, jejichž řešení jsou prověřena, mohou přijít na obhajobu, kde dostanou přidělenou nějakou další funkci, o kterou mají za úkol svoji CPU doplnit, a když se úkolu úspěšně zhostí, je jim přidělen patřičný počet bodů.

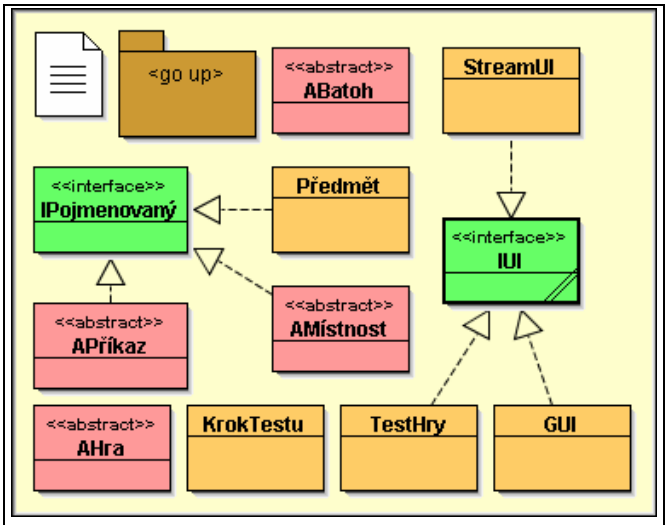
6.3 Hra

V pozdějších etapách výuky mají studenti za úkol vytvořit samostatnou aplikaci, která by realizovala konverzační hru, při níž bude hráč procházet jakýmisi virtuálním světem, zadávat počítači příkazy a dozvídat se jeho reakce na ně. Zadání je tedy poměrně obecné a mohli bychom obtížně hledat způsob, jak jednotně otestovat funkčnost vytvořených programů, aniž bychom museli trávit věky bušením příkazů do klávesnice.

První věcí, která nám pomůže v našem snažení po automatizaci testování, je vytyčení jistých mantinelů, v nichž se budou muset všechny hry pohybovat. Definoval jsem proto rámec, do nějž musí studenti své hry začlenit (viz obr. 3).

Diagram tříd je tentokrát poněkud složitější, a to přesto, že v něm oproti minulému diagramu nejsou zobrazeny běžné závislosti, ale pouze implementace rozhraní. Ani studenti již ve svých řešeních nevystačí s jedinou třídou, ale budou v nich muset použít několik vzájemně spolupracujících tříd.

Máme-li se po stažení všech řešení v obdržených projektech rozumně vyznat, musí každý student umístit své řešení do samostatného balíčku a tím i do samostatné složky. Balíček pak bude pojmenován identifikátorem studenta. Na jménech tříd uvnitř balíčku již nezáleží a studenti si je mohou pojmenovat naprosto po svém.



Obr. 3: *Diagram tříd rámce pro hru*

Máme tedy rámec, který definuje mantinely. Potřebujeme ale sadu testů, které můžeme těžko vytvořit centrálně, když ani nevíme, jakou hru se každý ze studentů bude snažit naprogramovat. Studenti proto dostanou nejprve za úkol definovat scénář své budoucí hry. Tento scénář zapíší jako posloupnost testovacích kroků, tj. jako kontejner instancí třída `KrokTestu`. Instance třídy `KrokTestu` budou obsahovat následující informace:

- příkaz, který uživatel (testovací program) na počátku tohoto testovacího kroku zadá a na nějž bude program v daném testovacím kroku reagovat,
- zprávu, kterou hra na daný příkaz v daném okamžiku odpoví (v různých situacích může na stejný příkaz odpovídat různě),
- místnost (nebo její ekvivalent), do které se po zpracování příkazu uživatel dostane,
- východy z cílové místnosti, tj. do kterých místností je možno se z této místnosti dále přesunout,
- předměty, které se v cílové místnosti nalézají,
- aktuální obsah uživatelova batohu.

Instance třídy `TestHry` má dvě metody. Metoda `testTestů(KrokTestu)` simuluje průběh hry pouze tím, že prochází zadané kroky testu a vypisuje stavy hry po jednotlivých příkazech. Tato metoda slouží především vyučujícímu, aby mohl zhodnotit, nakolik je předložený scénář úměrný požadavkům předmětu a schopnostem studenta.

Po odsouhlasení scénáře se studenti pustí do vytváření svého programu – hry. Jejich úkolem je definovat hru tak, aby se chovala přesně tak, jak si ve scénáři předepsali. K otestování funkčnosti hry slouží metoda `TestHry(AHra)`, která se hry zeptá na scénář a podle tohoto scénáře začne hru testovat.

Aby studenti nemohli scénář v průběhu vývoje libovolně měnit, používá vyučující jinou variantu testu, která se neptá hry na scénář, ale zeptá se jí na jméno autora a podle něj najde příslušný scénář v databázi dříve odevzdaných scénářů. Pokud proto student v průběhu vývoje hry dospěje k názoru, že by potřeboval scénář poněkud změnit, musí tuto změnu nejprve konzultovat s vyučujícím.

Závěrečný test odevzdaných řešení je opět jednoduchý: vyučující zkopíruje na svůj disk všechny odevzdané balíčky, instance vyhledávací třídy vyhledá všechny třídy implementující třídu `AHra` z rámce, vytvoří jejich instance a předhodí je jednu po druhé testovací třídě k otestování. Ta pak prověří každou hru pomocí dříve odevzdaného scénáře.

7 ZÁVĚR

Příspěvek ukázal, že vhodné využití rozhraní (`interface`) nám dává do rukou velice mocný nástroj pro zadávání, ale zejména pak pro automatizované vyhodnocování samostatně řešených studentských úloh. Předvedl, jak je možno zadávat studentské úlohy tak, aby si studenti při jejich řešení maximálně osvojili návyky, které budou ve své následující praxi potřebovat, a současně tak, aby si vyučující maximálně ulehčil vyhodnocování odevzdaných řešení.

Příspěvek současně seznámil s první verzí mikroknihovny pro automatizaci vyhodnocování odevzdaných řešení. Její použití může velmi výrazně zkrátit čas potřebný pro toto vyhodnocení.

Použití metodiky *Design Patterns First*, která zařazuje výuku rozhraní na samý začátek výuky, umožňuje využívat všechny tyto výhody po celou dobu kurzu.

LITERATURA

[1] 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)

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

[3] 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.

[4] PECINOVSKÝ Rudolf: *Návrhové vzory*, Computer Press, © 2007. ISBN 80-251-????-?.

[5] PECINOVSKÝ Rudolf: Aplikace metodiky „Design Patterns First“. *Objekty 2005 – sborník příspěvků devátého ročníku konference*, VŠB, Ostrava 2005. ISBN 80-213-1568-7.

[6] 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.

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

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

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

[10] 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.

[11] *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