Mýty ve výuce programování a metodika *Design Patterns First*

Rudolf Pecinovský

ICZ a.s., 140 00 Praha 4, Hvězdova 1689/2a
VŠE Praha, 130 67 Praha3, Nám. W. Churchilla 4
rudolf@pecinovsky.cz

Abstrakt. Příspěvek shrnuje dosavadní stav metodiky *Design Patterns First*. Nejprve připomíná (a současně vyvrací) některé oblíbené mýty rozhlašované pedagogy, kteří jimi zdůvodňují svoji neochotu učit objektově orientované programování. Poté představuje metodiku. Seznamuje s myšlenkami, na jejichž základě vznikla a popisuje postup výuky a výhody jejího používání. Ukazuje, jaké spektrum příkladů umožňuje včasné zavedení metodiky použít a jak včasné zavedení rozhraní umožňuje zefektivnit vyhodnocování studentských úloh.

1. Ohlédnutí

1.1 Metodiky programování

V historii programování se neustále objevovaly a objevují různé metodiky jak programovat a jak programování učit. Většina z nich je na počátku odmítána, protože nutí programátory a učitele měnit zažité zvyklosti a často navíc naznačuje, že programování zdaleka není takové umění, za jaké by je mnozí rádi vydávali.

První boj sváděli zastánci vyšších programovacích jazyků. Odpor programátorů k jejich používání byl velký. Všichni svorně tvrdili, že program nikdy nedokáže přeložit zdrojový kód tak dobře jako člověk. Tvůrci jazyka Fortran dokonce zašli ve snaze přimět programátory k jeho používání tak daleko, že překladač simuloval běh některých částí programu, aby pak mohl co nejefektivněji přidělit registry procesoru.

Jedna z nejznámějších „bitev" probíhala v sedmdesátých letech mezi zastánci a odpůrci strukturovaného programování. O této době pojednává řada úsměvných článků, z nichž asi nejznámější (alespoň v našich zemích) je ironický dopis nazvaný „*Opravdoví programátoři nepoužívají Pascal*" ([17]).

Necelý rok před Dijkstrovým článkem „*Go To Statement Considered Harmful*" ([3]), který inicioval výše zmíněnou „bitvu" se objevil programovací jazyk *Simula 67* vyvinutý původně pro simulaci procesů ([2],[9]). Ten přišel s několika zajímavými nápady, které oslovily další badatele, až se díky efektu sněhové koule staly dostatečně viditelné na to, aby se probojovaly i do zorného pole běžných programátorů.

Objektově orientované programování, které na těchto myšlenkách staví, je dnes hlavním proudem programování. Prakticky všechny významné projekty jsou dnes programovány objektově – tedy alespoň to o nich jejich autoři tvrdí. Když ale začnete

jejich programy studovat, zjistíte, že mnozí programátoři považují za objektový jakýkoliv program naprogramovaný v jazyce, který objektově orientované programování podporuje. Neuvědomují si, že program nedělá objektově orientovaným používání tříd a dědičnosti, ale že OOP vyžaduje výrazně odlišný přístup k řešení problému.

K tomu, aby byl program doopravdy objektově orientovaný a mohl využívat všech výhod, které OOP nabízí, je potřeba začít při vývoji programů také objektově myslet. Teprve pak se začnou projevovat výhody, které s sebou OOP přináší. Dokud programátoři nezačnou objektově myslet, tak si na OOP jen hrají.

1.2 Metodiky výuky

Obdobně jako s programováním je to i s jeho výukou. I zde probíhaly (a probíhají) bitvy mezi zastánci staršího přístupu a modernějším pojetím. Většinou bylo moderní pojetí tak dlouho znevažováno, dokud nezačalo učit dostatečné množství mladých učitelů, kterým bylo toto pojetí bližší než jejich starším kolegům. V současné době se např. bojuje o to, zda stále začínat výukou algoritmických konstrukcí, a nebo hned od počátku pracovat s objekty, třídami a rozhraními.

U výuky programování přibývá ještě další problém: je třeba rozlišovat mezi výukou programovacího jazyka a vlastní výukou programování. Je to obdobné, jako u výuky jazyků, kde je také rozdíl mezi výukou gramatiky a výukou tvorby literárních děl. Když se jednou naučíte psát dobře povídky, učebnice, básně nebo jakýkoliv jiný druh psaného textu, budete to umět ve všech jazycích, které ovládáte (samozřejmě s přihlédnutím k hloubce znalostí daného jazyka).

2. Mýty o výuce programování

Při debatách o výuce programování se neustále setkáváme s řadou mýtů. Pokusím se některé z nich připomenout a snad i vyvrátit.

2.1 Netřeba se starat o požadavky zaměstnavatelů, protože nejsou našimi zákazníky

Při návrhu koncepce výuky musíme mít na zřeteli požadavky, které na naše absolventy budou klást jejich zaměstnavatelé. Na konferenci *Informatika XXI./2008* mi v tomto bodě jeden účastník oponoval, že se nehodlá řídit požadavky firem, protože firmy nejsou jeho zákazníky. Řekl bych, že tím vyslovil názor nezanedbatelné části vyučujících. Přiznám se, že tento názor zcela nechápu.

Podle mne by jedním z cílů výuky mělo být co nejlepší uplatnění absolventů v praxi. Jak ale chceme dosáhnout tohoto cíle, když budeme potřeby praxe ignorovat, protože budoucí zaměstnavatelé nejsou našimi zákazníky?

Chceme-li, aby naši absolventi našli dobré uplatnění na trhu práce a aby je jejich zaměstnavatelé neposílali záhy po nástupu do „Pecinovského přeškolovacích kurzů“ (případně do jejich firemních ekvivalentů), měli bychom je od samého začátku učit programovat tak, aby pak už žádné přeškolení nepotřebovali.

2.2 Studenty musíme učit především algoritmickému myšlení

Jedno z dalších oblíbených pořekadel praví: „Studenty musíme naučit především algoritmickému myšlení, a proto musí výuka začínat základy algoritmizace“. Nikdo z těchto oponentů ale zatím přesně nespecifikoval, co to vlastně je to *algoritmické myšlení*. Když se ale začnete detailně vyptávat na to, co si přesně pod oním algoritmickým myšlením představují, zjistíte, že se vlastně jedná o způsob uvažování, který je vlastní objektově orientovanému programování. Spojovat cosi, co někdo nazval algoritmické myšlení, s algoritmizací je obdobně logické jako spojovat podvodníky s potápěči, protože ti přece pracují pod vodou.

OOP přineslo do programování další výrazné zmenšení sémantické mezery mezi způsobem popisu problému běžným člověkem a způsobem jeho popisu v analýza a následně v programu. Když začátečníky učíme nejprve základy algoritmizace, vnucujeme jim transformaci běžného popisu problému do primitivního tvaru vyžadovaného programovacími jazyky šedesátých a počátku sedmdesátých let minulého století. Problém je v tom, že když si jednou tento způsob uvažování osvojí, již těžko se vracejí k přirozeněji formulovanému způsobu řešení daného problému.

Jedním z největších problémů „algoritmicky vychovaných programátorů“ je např. jejich neschopnost zůstat při hovoru o programu na hladině abstrakce, které rozumí i zadavatel, a nesklouzávat neustále na úroveň kódu. U objektových programátorů je tato tendence neporovnatelně slabší, protože při návrhu programu myslí v jiných kategoriích – v takových, kterým často rozumí i zákazník.

2.3 Objektové orientované programování je jen dočasný módní výstřelek

Výrok uvedený v nadpisu této podkapitoly prohlašují lidé, kteří donedávna o OOP vůbec neslyšeli. To, že o něm neslyšeli ale ještě neznamená, že neexistovalo. Jak je obecně známo (a jak jsem i v úvodu naznačil), OOP je ve skutečnosti o nějaký rok starší než strukturované programování, které tito oponenti prosazují.

Objektově orientované programování dostalo při svém zrodu do vínku dva nezanedbatelné handicapy:

- Má poněkud větší nároky na rychlost procesoru a množství paměti.
- Chvíli trvalo, než programátoři správně pochopili, co jim vlastně nové paradigma nabízí a jak tyto možnosti co nejlépe využít.

V současné době však jsou již oba handicapy dávno překonány. Výkonnostní parametry současných počítačů jsou tak vysoké, že drobná režie spojená s některými objektově orientovanými konstrukcemi je již nepodstatná a je bohatě vyvážena dalšími vlastnostmi: rychlejším vývojem, stabilnějším a robustnějším výsledným programem a především pak mnohem snáze udržovatelným kódem. (A navíc kódem, který je mnohem snáze opětně použitelný v jiných aplikacích.)

Trochu větším problémem jsou zažité zvyky, které vznikly v době, kdy bylo běžné chápání významu a použití některých konstrukcí výrazně odlišné od současného. Dnes se již ví, že toto chápání vedlo ke vzniku problematických vazeb uvnitř programu. Nicméně na některé školy tato znalost ještě zřejmě nepronikla a učí zde programové konstrukce dále podle již dávno překonaného modelu.

2.4 OOP vede k vytváření nespolehlivých programů – podívejte se na *Windows*

V tomto mýtu se spojilo hned několik špatných úvah najednou. První, nejviditelnější je, že se podle jednoho programu posuzuje celá metodologie. Na této úvaze je navíc scestné to, že *Windows* mají objektově orientované uživatelské rozhraní, ale samy dlouho nebyly naprogramované objektově.

V současné době však již samozřejmě objektově programovány jsou, protože OOP je metodika, která umožňuje vytvářet rozsáhlé a komplikované programy nejrychleji a nejspolehlivěji. Ostatně podívejte se na banky jako na organizace, kterým životně záleží na spolehlivosti jejich softwaru. Prakticky vše, co se pro ně programuje, se programuje povinně objektově.

Navíc bychom si měli přiznat, že programy takové „obludnosti" jakou vykazují současné operační systémy, kancelářské balíky a další aplikace, se už dost dobře jinak než objektově v rozumném čase a spolehlivosti naprogramovat nedají. Neobjektové programování se proto v současné době používá především při vývoji ovladačů a programů pro jednočipové počítače (avšak i sem již OOP proniká). V rozsáhlejších aplikacích jsou neobjektově programována již pouze jádra některých z nich.

2.5 Aby se studenti naučili správně programovat, musejí nejprve pochopit základní principy práce počítače

Tento názor je vyhlašován hlavně jako protiváha ke snaze začínat výuku OOP přímo objekty. Obávám se, že stejně jako většina řidičů netuší, jak přesně pracuje jejich automobil, aniž by jim to při úspěšném řízení automobilu nějak zásadně vadilo, tak i většina programátorů znalosti o „podhoubí" svých programů nikdy nevyužije. Většina programátorů dokonce nevyužije ani hlubší znalosti algoritmizace a konstrukce složitějších datových struktur.

Občas se dokonce ukazuje, že tyto znalosti jsou kontraproduktivní. Programátoři, kteří vědí, do jaké podoby je program přeložen a jak je pak tento přeložený program spouštěn, mívají často neodolatelné nutkání pomoci překladači a upravit program do efektivněji zpracovávané podoby. Ukazuje se však, že značná část projektů, jejichž vývoj byl předčasně ukončen, zkrachovala právě kvůli takovýmto předčasným snahám o optimalizaci, protože programátoři v zájmu optimalizace vytvářeli kód, který obsahoval řadu skrytý chyb a navíc byl právě kvůli těmto snahám špatně čitelný a modifikovatelný, takže se jej nedařilo v požadovaném termínu odladit. A to už vůbec nahovořím o tom, že současný vývoj optimalizačních překladačů pokročil tak daleko že „ruční optimalizace" se s ním dá jen těžko srovnat.

2.6 OOP je pro studenty příliš abstraktní a obtížněji je chápou

Jeden z oblíbených pedagogických mýtů vychází z přesvědčení, že když něco špatně chápe učitel, budou to ještě hůře chápat studenti. Toto tvrzení je pravdivé pouze v případě, kdy studentům vykládá látku učitel, který ji sám nechápe. Zkušenosti s výukou objektově orientovaného programování na základní škole ale ukazují, že děti žádné problémy s chápáním OOP nemají. Ony totiž hned pochopí, že přirozeně popisuje jim známou skutečnost a nemají problém s takovýmto popisem pracovat.

Problémy nastávají až v okamžiku, kdy potřebujeme naučit objektově programovat někoho, kdo už před tím programoval neobjektově. Zkušenosti velkých firem (viz např. [18]) ukazují, že přeškolení klasického programátora na programátora objektově orientovaného trvá v průměru 6 až 18 měsíců. Navíc čím je programátor zkušenější, tím je přeškolení delší (a dražší), protože se musí zbavovat hlouběji zakořeněných návyků. Zkušení programátoři se totiž snaží nejprve interpretovat přednášené konstrukce pomocí konstrukcí, které již znají, takže jim dlouho trvá, než konečně pochopí, že tudy cesta nevede. Pro začátečníky je totiž dobře vysvětlené objektové myšlení mnohem přirozenější než násilný převod zadání do klasických strukturovaných schémat.

Před odevzdáním rukopisu prvního vydání knihy [16] do tisku jsem dával tento rukopis číst řadě lidí, abych získal zpětnou vazbu a text ještě případně upravil. Docela mne tehdy překvapilo, že oslovení teenageři neměli s chápáním textu prakticky žádné problémy, kdežto oslovení učitelé programování na základních a středních školách s ním docela zápasili a u řady pasáží požadovali důkladnější vysvětlení. Dospěl jsem tehdy k názoru, že jedním z jejich velkých handicapů je právě jejich znalost strukturovaného programování, která ovlivňuje jejich chápání čteného textu, protože si v textu domýšlejí „mezi řádky“ věci, které text netvrdí a velmi často ani nejsou pravda.

3. Metodika *Design Patterns First*

Metodika *Design Patterns First* vznikla jako reakce na problémy, s nimiž se v praxi setkáváme u absolventů vyučovaných podle dosavadních metodik. Tito studenti jsou dobře proškoleni v oblastech, které nikdy nepoužijí, ale chybí jim znalosti a především praktické zkušenosti v oblastech, s nimiž se budou denně setkávat. Metodika se současně snaží dodržovat zásady správné výuky publikované např. v [1], [6], [11].

3.1 Výchozí teze

Metodika *Design Patterns First* vychází z následující posloupnosti základních tezí o současném programování a jeho výuce:

1. Principiální myšlenky je vhodné vysvětlit studentům co nejdříve, aby měli dostatek času na jejich osvojení a dostatek příležitostí na jejich procvičení. (Pedagogická zásada ranního ptáčete – viz např. [1]).
2. Současné programování čelí rostoucím požadavkům na vysokou efektivitu vývoje důrazem na snadnou modifikovatelnost a vysokou znovupoužitelnost vytvořených programů.
3. K dosažení vysoké efektivity vývoje snadno modifikovatelných a znovupoužitelných programů výrazně napomáhá používání návrhových vzorů ([4], [15]). Ty se v poslední době stávají jedním z klíčů úspěšného vývoje programů.
4. Abychom studenty co nejlépe naučili pracovat podle zásad moderního programování, měli bychom zanést návrhové vzory do výuky co nejdříve.
5. Návrhové vzory důsledně uplatňují zásadu, že se nemá programovat proti implementaci, ale proti rozhraní.

6. Java, používaná jako programovací jazyk většiny vstupních kurzů programování zavádí konstrukci `interface`, kterou bychom mohli považovat za formalizaci zápisu rozhraní (případně části rozhraní) třídy.

7. Máme-li ve výuce co nejdříve používat návrhové vzory a další aspekty moderního programování, měli bychom co nejdříve vysvětlit programovou konstrukci `interface`.

V současné době převažuje ve světě metodika výuky programování označovaná jako *Object First* (viz např. [7], [10]). Její velkou předností je, že učí studenty pracovat s objekty ještě dříve, než začnou psát vlastní kód. Mají tedy možnost si osvojit základního ducha objektově orientovaných programů ještě před tím, než je začnou rozptylovat záludnosti syntaxe (viz např. [16]).

Nevýhodou metodiky *Object First* (přesněji nevýhodou stávajících učebnic) je ale to, že po dlouhou dobu zůstává u výuky objektů a práce s nimi a `interface` vykládá až téměř na závěr výuky, kdy už studenti nemají příliš mnoho šancí jej zažít.

3.2 První cvičení

Metodika *Design Patterns First* proto přebírá od metodiky *Object First* myšlenku začít kurz s vývojovým prostředím, které umožní pracovat z počátku v interaktivním režimu. V něm se studenti naučí pracovat s objekty a posílat jim zprávy. Současně získají první povědomí o tom, že objektem může být doopravdy vše, co můžeme označit podstatným jménem. Ukážeme jim, že i třída je objekt, a to objekt, který umí vytvářet své instance. Současně jim umožníme nahlédnout pod pokličku objektů a ukážeme jim atributy jako místa, kam si objekty ukládají informace o svém stavu.

Veškerý výklad včetně ukázek postupů mají studenti k dispozici ve formě flashových animací, takže si pokud něco z tohoto úvodního bombardování novými koncepcemi a termíny nezapamatují, mohou si vše v klidu domova zopakovat a vyzkoušet.

V závěru prvního cvičení si studenti vytvoří svoji první třídu. I tu však vytvářejí v interaktivním režimu. Využíváme toho, že prostředí *BlueJ*, které používáme, je schopno definovat metody testovací třídy obdobným způsobem, jakým jsou v kancelářských programech definována jednoduchá makra, tj. předvedením postupu, který je třeba naprogramovat a který pak dané prostředí samo naprogramuje.

Za domácí úkol pak mají vytvořit vlastní třídu, jejíž instance budou představovat obrázek složený nejméně za čtyř různých objektů: domek, obličej, auto …

3.3 Druhé cvičení

Na druhém cvičení se studenti naučí psát zdrojový kód třídy a vytvoří své první konstruktory a následně i řadové metody. Naučíme je využívat skrytý parametr `this` a vysvětlíme princip přetěžování metod.

Následně studentům vysvětlíme, jak je k plnohodnotné funkci objektu třeba zavést atributy a ukážeme jim, jak atributy definovat a používat. Při té příležitosti vysvětlíme rozdíl mezi atributem a vlastností objektu a zdůrazníme výhodnost (či spíše povinnost) definovat atributy jako soukromé.

Ke konci hodiny jim ukážeme, jak je třeba definovat metody, které vracejí několik hodnot najednou a seznámíme je s jednoduchým návrhovým vzorem *Přepravka*, který jim ukáže, jak zařídit, aby jejich metody mohly vracet několik hodnot současně.

Použití návrhového vzoru *Přepravka* demonstrujeme na příkladu zjišťování a nastavování pozice a rozměrů objektů. Přitom se ale ukáže, že výchozí projekt, s nímž studenti pracují, je navržen poněkud prostoduše. Objekt, který se má přesunout, se musí nejprve v původní pozici smazat. Tím ale často poškodí vzhled jiných zobrazených objektů, protože je přitom také odmaže. Tak si připravíme motivaci pro následující cvičení, kdy studentům ukážeme, jak tuto nevýhodu elegantně odstranit.

Za domácí úkol mají studenti naprogramovat třídu, kterou vytvářeli v minulém domácím úkolu interaktivně, definovat v ní předepsané přetížené verze konstruktorů a metody pro zjišťování a nastavování rozměru a pozice.

3.4 Třetí cvičení

Na třetím cvičení si představíme návrhové vzory *Prostředník* a *Pozorovatel*, které nabízejí způsob, jak se vypořádat s problémem odmazávání objektů, s nímž jsme se potýkali na konci předchozí hodiny. Koncepce těchto návrhových vzorů je však postavena na doposud neznáme programové konstrukci – na `interface`. Vysvětlíme jim její význam pro současné programování a ukážeme, jak nám tato konstrukce umožní vyřešit naše dosavadní problémy.

Ukážeme jim pak, jak jsou zmíněné návrhové vzory implementovány v novém projektu, který již netrpí neduhy předchozího a při změnách pozice a rozměru zobrazených objektů již nic neodmazává. Při té příležitosti je seznámíme s událostmi řízeným programováním a naučíme je psát programy, které nebudou muset neustále zjišťovat, zda ta či ona událost nastala, ale budou se umět dohodnout se zdrojem události, aby je na nastalou událost upozornil v okamžiku jejího vzniku a ony mohly zareagovat.

Domácí úkol na této hodině je jednoduchý: upravit definici svých tříd (grafického objektu a testovací třídy), tak, aby plnohodnotně pracovaly s novou verzí knihovny grafických objektů

3.5 Čtvrté cvičení

Ve čtvrtém cvičení prohlubujeme chápání pojmu `interface` a ukazujeme studentům další příklady úloh, které lze elegantně řešit vhodným použitím rozhraní.

Seznámíme studenty s návrhovým vzorem *Služebník* a předvedeme jim, jak je možno stávající objekty elegantně doplnit o funkčnost, kterou dříve někdo naprogramoval. Poté si ukážeme „služebníky“ s komplexnějšími požadavky na obsluhované objekty a předvedeme si, jak můžeme vyjít těmto požadavkům vstříc implementací několika rozhraní a dědičností rozhraní.

Studenti tak poznají další možné použití konstrukce `interface` a pomalu si začnou uvědomovat, ve kterých situacích je vhodné tuto konstrukci do programu začlenit. Začnou pomalu chápat, kde všude lze užitečně využít základní skutečnost, že prostřednictvím rozhraní může objekt deklarovat svoje požadavky na své budoucí „spolupracovníky“ a jak ji využít při návrhu architektury programu.

Při zadání domácího úkolu je seznámíme s novým služebníkem (a jím vyžadovaným rozhraním), který zprostředkuje ovládání svěřeného objektu prostřednictvím klávesnice. Studenti mají za úkol implementovat zadané rozhraní a zprovoznit ovládání změny pozice a velikosti jejich grafických objektů z klávesnice.

3.6 Páté cvičení

Na dalším cvičení procvičujeme algoritmické konstrukce a práci s kontejnery, konkrétně s množinami a seznamy. Studenti se naučí ukládat do seznamu pozice, kterými prošel jejich objekt při ovládání z klávesnice, a po ukončení „klávesnicového řízení" pak „přehrát" celou absolvovanou cestu. Ukážeme jim různé problémy, které mohou v průběhu řešení této úlohy vzniknout. Naučí se přitom pracovat s iterátory a uvědomí si některá jejich omezení, na které je třeba myslet při jejich používání.

3.7 Další cvičení

Na následujících cvičeních pak postupně doprobereme základní jazykové konstrukce. Při všech ukázkách a domácích úkolech neustále posilujeme chápání návrhových vzorů a především rozhraní jako základního nástroje pro zefektivnění vývoje podle současných měřítek. Ukazujeme, že dobře navržený program je schopen akceptovat poměrně velké změny zadání bez neustálých dominových efektů vedoucích na nutnost předělání podstatné části programu po každé drobné změně.

Ukazujeme jim, že architektura projektu založeného na rozhraních usnadní nejenom reakce na neustále se měnící požadavky zákazníka, ale že umožní i snadnou výměnu celých modulů a vzájemnou zastupitelnost kompatibilních modulů.

V neposlední řadě pokračujme v požadavku důsledného testování vytvořených programů. Ukazujeme studentům, jak vytvářet jednotkové testy, a učíme je, jak je možno pomocí zástupných objektů (mock objects) vyzkoušet spolupráci s ostatními moduly projektu dlouho před tím, než budou tyto objektu naprogramovány.

Součástí požadavků je u všech domácích úkolů nejenom odevzdání vytvořené třídy, ale současně i odevzdání příslušné testovací třídy, která prověří, že objektu pracuje skutečně tak, jak bylo požadováno.

4. Efekty uvedeného postupu

4.1 Efekty pro studenta

Hlavním cílem celé koncepce je prohloubit chápání objektové podstaty současného programování a především pak chápání významu rozhraní.

Programátoři, kteří přicházejí do mých přeškolovacích kurzů, většinou znají syntaxi jazyka a vědí, jak mají implementovat rozhraní. Neumějí však ve svých projektech odhalit místa, kdy by sami měli rozhraní definovat a jsou pak překvapeni, čeho všeho se dá takto dosáhnout a jak se pak mnohá řešení zjednoduší.

Metodika *Design Patterns First* se snaží vychovat programátory, kterým jsou tyto postupy vlastní a kteří budou schopni navrhovat programy snadno upravitelné, rozšiřitelné a spravovatelné.

Klíčovým cílem celé metodiky je optimalizovat spektrum získaných znalostí a dovedností z hlediska budoucí praxe absolventů. Typickou úlohou programátora většinou není vyřešit nějakou „jednomužně zvládnutelnou“ úlohu na zelené louce. Bývá spíše postaven před nějaký rozsáhlý projekt, na jehož vývoji se podílí více lidí a má pouze doplnit či upravit některou jeho část. Metodika se proto snaží studenty naučit tomu, co budou při takto koncipovaných úlohách potřebovat.

Včasné zavedení konstrukce `interface` nám (mimo jiné) umožní snadno zadávat kolektivní úkoly, na nichž se podílí několik studentů současně. Můžeme proto studentům ukázat, jak postupovat v situacích, kdy má jejich modul spolupracovat s moduly, které ještě nejsou hotové a jak takovou budoucí spolupráci testovat ještě před tím, než je spolupracující modul hotov.

4.2 Efekty pro učitele

Jednou z nejnepříjemnějších součástí výuky je vyhodnocování domácích úloha. Je to činnost relativně mechanická, pracná a velmi únavná. Včasné zavedení rozhraní umožňuje koncipovat zadání úloh tak, aby bylo možno velmi snadno a jednoduše automatizovat vyhodnocování odevzdaných řešení. Při vhodně definovaném zadání bývá příprava vyhodnocovacího programu otázkou půl hodiny. Začlenění připraveného programu do vyhodnocovacího rámce, jeho spuštění a následné vyhodnocení všech odevzdaných prací pak zabere pouze několik málo minut prakticky nezávisle na počtu studentů a odevzdaných prací ([14]).

5. Závěr

Příspěvek se pokusil vyvrátit některé mýty, kterými mnozí učitelé často argumentují při zdůvodňování toho, proč nechtějí učit objektově orientované programování, anebo proč trvají na tom, že před vysvětlením objektových konstrukcí musí nejprve probrat klasické, strukturované programování.

Ukázal, že tento postup je velmi často kontraproduktivní, protože se studenti nejprve učí způsobu uvažování, který pak musejí při přechodu na skutečné objektové programování opustit. Výsledkem je pak buď zbytečná pracnost osvojení si OOP, anebo naopak přijetí mylného předpokladu, že programovat objektově znamená jenom používat třídy a objekty.

V závěrečné části seznámil s metodikou *Design Pattern First*, která se snaží učit programování tak, aby si absolventi maximálně osvojili duch současného objektově orientovaného programování a našli tak co nejlepší uplatnění v praxi. Ukázal, že klíčovou podmínkou takto pojaté výuky je brzké začlenění výkladu návrhových vzorů a její pravidelné používání v řešených programech. To ale vyžaduje včasné zavedení konstrukce `interface`. Vedlejším efektem aplikace této metodiky pro učitele může být mimo jiné výrazné snížení pracnosti vyhodnocování řešení domácích úloh.

Literatura

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

[2] DAHL Ole Johan, NYGAARD Kristen: Class and subclass declarations. NCC document. (As presented at the IFIP Working Conference on Simulation Programming Languages, Oslo May 1967).

[3] DIJKSTRA Edsger W.: Go To Statement Considered Harmful, Communications of the *ACM*, Vol. 11, No. 3, March 1968, pp. 147-148, Elektronickou verzi lze získat např. na http://www.ecn.purdue.edu/ParaMount/papers/dijkstra68goto.pdf

[4] GAMMA E.; HELM R.; JOHNSON R.; VLISSIDES J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. ISBN 0-201-63361-2.

[5] GODDARD, Doug. 1994. Is it really object oriented?. *Data Based Advis.* 12, 12 (Dec. 1994), 120-123.

[6] KÖLLING Michael: The problem of teaching object-oriented programming , Part 1: Languages. Journal of Object-Oriented Programming, 11(8): 8-15, 1999

[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 Workshop OOPSLA 2002. http://www.cse.buffalo.edu/faculty/alphonce/KillerExamples/OOPSLA2002

[9] NYGAARD Kristen: SIMULA, An Extension of ALGOL to the Description of Discrete Event Networks. In *Proceedings, IFIP Congress62*, pp.520-522. North-Holland Publ. Comp.

[10] 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.

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

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

[13] 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.

[14] PECINOVSKÝ Rudolf: Metodika *Design Patterns First* a vyhodnocování studentských úloh. *Tvorba softwaru 2007 – sborník přednášek*. ISBN 978-80-248-1427-8.

[15] PECINOVSKÝ Rudolf: Návrhové vzory – 33 vzorových postupů pro objektové programování, Computer Press 2007, ISBN 978-80-251-1582-4.

[16] PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java – Kompletní učebnice programování pro naprosté začátečníky, 2. aktualizované a rozšířené vydání*. Grada 2008. ISBN 978-80-247-2653-3.

[17] POST Ed: *Real Programmers Don't Use Pascal*, Datamation 1983, Stručná charakteristika s odkazy na http://en.wikipedia.org/wiki/Real_Programmers_Don%27t_Use_Pascal, český překlad např. na http://pluto.pslib.cz/kerslage/folklor/pojidaci.kolacu.html.

[18] STROUSTRUP Bjarne: *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994

Annotation

First the paper reminds some favorite myths about the object oriented programming presented by teachers who need to justify their unwillingness to teach it. It rebuts these myths and introduces the *Design Patterns First* methodology. It shows that the key condition of modern programming teaching is early introduction of design patterns and their regular use in presented examples. It needs early introduction of `interface`, too. This concept allows the students to master a broad range of the tasks which would be otherwise too difficult at the beginning of the course. The paper shows some of them. Simultaneously it shows how the early introduction of the `interface` simplifies the automation of the of the students' tasks evaluation.