JAK EFEKTIVNĚ UČIT OOP

Rudolf Pecinovský

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

Abstrakt
Objektově orientované programování vyžaduje diametrálně odlišný přístup k analýze a syntéze programu, než před ním převažující programování strukturované. Většina učebních textů a kurzů programování však tuto skutečnost nebere v úvahu a je nadále koncipována především jako kurzy, ve kterých se jejich čtenáři, resp. frekventanti učí především syntaxi použitého jazyka. Opravdu objektový přístup k vývoji programu se proto studenti učí až v nadstavbových kurzech, kdy ale již mají zažitu řadu zlozvyků, které se musejí postupně odnaučovat. Příspěvek shrnuje zkušenosti s výukou snažící se vštípit žákům co nejdříve zásady OOP a ukazuje, jak je třeba změnit přístup k výuce, aby absolventi kurzů byli schopni programovat doopravdy objektově. Postupně popisuje praktickou aplikaci jednotlivých zásad, které je vhodné při výuce programování dodržet, a shrnuje zkušenosti s takto pojatou výukou.

1 CO DĚLÁME NEŠIKOVNĚ

Podíváte-li se do obsahu učebnic programování pro začátečníky či projdete-li si obsahy nejrůznějších začátečnických kurzů programování, zjistíte, že přestože o sobě všechny tyto učebnice a kurzy prohlašují, že učí své čtenáře a frekventanty programovat, učí je většinou ve skutečnosti pouze syntaxi vysvětlovaného jazyka v bláhové naději, že jako vedlejší efekt této výuky se student naučí také programovat.

Učebnice a kurzy proto kladou důraz především na výklad nejrůznějších syntaktických pravidel a jsou doprovázeny většinou jen AHA-příklady sloužícími pouze k demonstraci vysvětlované konstrukce bez nějaké větší snahy začlenit danou konstrukci do celkového kontextu.

Kdybychom použili paralelu z matematiky, styl výuky těchto učebnic a kurzů bychom mohli přirovnat k výuce matematiky, ve které se učí pouze úpravy výrazů a řešení rovnic, ale vůbec se žákům nezadávají slovní úlohy, při jejichž řešení by se naučili přednášenou látku aplikovat.

Obdobné je to i s výukou základních programovacích paradigmat. S nástupem vyšších programovacích jazyků většina vyučujících (znám i výjimky) pochopila, že je výhodnější učit nejprve základní principy programování v některém z vyšších programovacích jazyků, a teprve v další etapě začít případně učit assembler. Přijali tento postup přesto, že assembler používá méně programových konstrukcí (alespoň u těch prvních to tak bylo), takže na první pohled vypadá programování v assembleru jednodušší.

Po nástupu dalších paradigmat se však převažující styl výuky už nezměnil. Částečně asi z konzervativizmu, ale podle mne především proto, že by změna stylu výuky vyžadovala přípravu řady nejrůznějších pomocných programů, a na to většinou nebývá čas. Kromě toho je při tomto způsobu výuky mnohem obtížnější i vymýšlení příkladů.

1.1 Strukturované paradigma

Podívejme se např. na výuku strukturovaného programování, které bylo ve vstupních kurzech po dlouhou dobu nejčastěji probíraným paradigmatem. Všichni asi budou souhlasit s tím, že jednou ze základních dovedností, které si musí strukturovaný programátor osvojit, je dekompozice problému na jednodušší podproblémy a správný návrh procedur a funkcí. Nezaujatý pedagog by proto řekl, že by bylo vhodné výukou dekompozice a s ní souvisejícího návrhu procedur a funkcí začít, aby měl žák dostatek času si vše v průběhu další výuky neustále procvičovat a zvládl si tak celou problematiku co nejlépe osvojit a zažít.

Jak ale víme, většina učebnic a kurzů tuto zásadu neakceptovala. Značný prostor věnovala tomu, aby naučila žáky správně vytvářet „střeva“ budoucích procedur, a o vlastním návrhu procedur a funkcí se zmínila až na konci celého výkladu. Na výklad dekompozice už pak často nezbyl prostor a odkládal se proto až do učebnic a kurzů pro pokročilé.

V tomto trendu pokračuje i značná část současné středoškolských kurzů, které nejen že povětšinou ignorují nástup objektového programování a snaží se omezit na programování strukturované, ale navíc je učí tak, že poměrně dlouho nemají žáci šanci naprogramovat něco, co by jim připadalo alespoň trochu zajímavé.

V tomto trendu pokračuje i značná část současné středoškolských kurzů, které nejen že povětšinou ignorují nástup objektového programování a snaží se omezit na programování strukturované, ale navíc učí tak, že poměrně dlouho nemají žáci šanci naprogramovat něco, co by jim připadalo alespoň trochu zajímavé.

Nevýhodou takto koncipovaných učebnic a kurzů je, že se žáci naučí definovat části programu jako procedury opravdu jenom tehdy, kdy je třeba danou činnost vyvolávat na mnoha místech. Ignorují však skutečnost, že nový podprogram můžeme definovat také jen kvůli zpřehlednění stávajícího. Jejich podprogramy jsou pak obrovské a nepřehledné a žáci opouštějící takovéto kurzy většinou nejsou schopni vytvořit jen trochu složitější program.

Se zásadami tvorby správných programů se pak tito absolventi seznamují většinou postupně prostřednictvím poznávání jiných programů. Tento proces je však velice pomalý a řada programátorů přesvědčených o vlastní neomylnosti si možnost jiného přístupu velice dlouho nepřizná.

Ti z vás, kteří se setkali s metodikou robota Karla, si ale jistě vzpomenou, že obrácený postup výkladu je nejenom možný, ale přináší i své výsledky. Tato metodika učí žáky nejprve vytvářet procedury a dekomponovat program na jednodušší celky, a teprve pak je seznamuje s podmínkami, cykly a dalšími programovými konstrukcemi. Programátoři, kteří se učili podle této metodiky, si zásady správného programování procvičovali od samého počátku výuky, takže na konci měli již mnohé z těchto zásad zažité „pod kůží“.

1.2 Objektově orientované programování

Situace se opakuje i při výuce objektově orientovaného programování. Většina programátorů uznává, že moderní programování je a ještě dlouho bude objektově orientované. (Středoškolští učitelé programování sice odmítají tento fakt přiznat, ale to je zase jiná pohádka.) Podíváte-li se však do učebnic a osnov programátorských kurzů, opět zjistíte, že zůstává pouze u deklarací. Na počátku výkladu se sice žáci dozvědí o důležitosti objektově orientovaného přístupu, vlastní výklad pak začne výkladem primitivních datových typů a klasických strukturovaných konstrukcí. Studenti se tak na počátku seznamují se zcela jiným paradigmatem, než je to, které by si měli v průběhu kurzu osvojit.

Ve stejném duchu pak učebnice i kurzy často pokračují. Seznámí studenta se syntaxí definice třídy, vysvětlí mu fungování dědičnosti, prozradí mu existenci rozhraní. V řadě učebnic a kurzů jsem se dokonce setkal s tvrzením, že rozhraní v Javě, C# či Visual Basicu .NET bylo zavedeno především proto, aby bylo možno alespoň částečně implementovat násobnou dědičnost, kterou tyto jazyky nezavádí.

Studenti se v takovýchto učenicích a kurzech většinou vůbec nic nedozví o pravidlech, jak správně navrhovat objektový program. Prošla mi rukama řada učebnic a prohlížel jsem plány řady kurzů. Téměř nikde jsem si nevšiml, že by se v průběhu výuky OOP studenti něco dozvěděli o soudržnosti a provázanosti tříd a metod a už vůbec ne o návrhových vzorech. Podrobně se jim vysvětlují zásady dědičnosti tříd, ale vyučující přitom zapomene zdůraznit, že současné programování dává před dědičností tříd přednost skládání a používání rozhraní.

Nemůžeme ale tvrdit, že učíme objektově orientované programování, když neprozradíme téměř nic z jeho filosofie. Programovat objektově neznamená používat třídy, ale vytvářet programy, které mají objektově orientovaného ducha.

Obdobně neopovažuji za optimální, když se kurzy návrhových vzorů soustředí především na vyjmenování 23 základních vzorů publikovaných v [1], ale o zásadách, kterými se autoři těchto návrhových vzorů řídili, se nijak podrobně nezmiňují. Jako čestnou výjimku bych uvedl učebnici [2], jejíž autoři seznamují čtenáře nejenom s vlastními návrhovými vzory, ale průběžně jim také vštěpují hlavní zásady OOP.

2 ZKUSME TO JINAK

Přiznám se, že když jsem začal učit OOP, trpěly moje učebnice a kurzy obdobnými neduhy. Také jsem učil především syntaxi a bláhově očekával, že studenti pochopí ducha programování mezi řádky a že si po nástupu do praxe potřebné návyky postupně osvojí. Bohužel jsem si musel přiznat, že přestože frekventanti kurzů bez problémů zvládli syntaxi a teoretické základy použití objektových rysů jazyka, jejich programy příliš objektově orientované nebyly.

Před několika lety jsem se ale seznámil s učebnicí [3], jejíž autoři vyvinuli vývojové prostředí *BlueJ*, které jim umožnilo začít výuku experimenty s třídami a objekty a hned první vytvářené programy koncipovat jako doopravdy objektové. Vývojové prostředí *BlueJ* umožňuje (stejně, jako kdysi prostředí robota Karla) převrátit zaužívanou posloupnost výkladu a doopravdy začít výkladem toho, co by měli studenti zvládnout především, a to je práce s třídami a objekty. Umožňuje dokonce začít nejenom teoretickým výkladem vysvětlujícím, co to jsou objekty, ale doprovodit jej hned praktickými experimenty následovanými tvorbou vlastních programů realizujících funkce, které studenti poznali při předchozích experimentech.

Při hlubším studiu možností, které tento způsob výkladu nabízel, jsem byl čím dále tím více přesvědčen, že autoři [3] přišli s geniální myšlenkou, avšak nevyužili plně její potenciál. Mezi mé hlavní výtky patří to, že strukturované konstrukce vykládají jen tak mimochodem v rámci výkladu něčeho jiného, takže žák, který si potřebuje něco připomenout či ujasnit, potřebnou informaci jen velmi obtížně hledá. Kromě toho mi na tomto výkladu vadilo, že rozhraní probírá až po dědičnosti a navíc je v prvním plánu prezentuje především jako náhradu násobné dědičnosti a nijak se nezmiňuje o klíčové roli, kterou v současném programování rozhraní bezesporu má.

Zkusil jsem proto dotáhnout naznačené myšlenky směrem, o němž jsem se domníval, že využije potenciál původního nápadu efektivněji. Novou metodiku jsem vyzkoušel v kroužcích programování navštěvovaných žáky základních a středních škol. Po těchto „testech na dětech“ jsem ji začal používat ve svých kurzech pro profesionální programátory a podle reakcí posluchačů přicházejících na pokračovací a nadstavbové kurzy soudím, že úspěšně (viz např. [5]).

2.1 Principy, na nichž je metodika postavena

Předkládaná metodika je publikována v [6] a je postavena na několika zásadách, které byly podrobněji vysvětleny např. v [7]:

- *Od začátku vštěpovat žákům zásady moderního programování*
  Chceme-li, aby se žákům vryly zásady moderního programování pod kůži, musíme je vysvětlovat od samého začátku, a ne je napřed učit programovat trochu jinak, a pak je vše přeučovat.
- *Co nejdříve umožnit tvorbu programů*
  Programováním přitom nemyslím jenom klasické psaní kódu, ale i přímé zadávání instrukcí nějakému ovládanému subjektu.
- *Nepředbíhat, tj. nepoužívat prvky jazyka, které ještě nebyly vyloženy*
  Nejznámějším porušením této zásady je známý program *Hello World*, který sice bývá umístěn na počátek učebnice, ale používá konstrukce vysvětlené až v její hloubi.
- *Informace je třeba předávat po malých soustech*
  Jinými slovy: výklad hojně prokládat příklady, při jejichž řešení mají žáci možnost látku zažít. Nová informace nesmí vytlačit starou, tj. nesmí přijít dřív, dokud stará nezakoření.

- *Příklady musí být zajímavé*
  Je-li příklad zajímavý, chtějí jej žáci řešit sami od sebe bez nucení. K tomuto účelu je výhodné zkusit naprogramovat nejrůznější hry a programy realizující různé animace.
- *Studenti by se měli co nejvíce seznamovat s hotovými, vzorově vyřešenými programy*
  To má několik výhod: za prvé uvidí, jak má takový program vypadat a za druhé nám to umožní zadávat i složitější úlohy, než by v danou chvíli zvládli sami.
- *Studenti se musí naučit programy nejen vytvářet, ale také ladit*
  Budeme-li jenom obcházet studenty a opravovat za ně jejich chyby, tak je pro samostatné programování moc dobře nepřipravíme. Studenti se musí naučit své chyby nacházet a opravovat sami.
- *Doprovodné příklady musí vyžadovat aktivní použití nových poznatků*
  Jinými slovy: příklady by pokud možno neměly být drilové, ale raději takové, při nichž musí žáci trochu „zapnout hlavu".
- *Řešení nesmí být příliš zašuměná*
  Řešení problému, který demonstruje použití vysvětlované konstrukce, by nemělo zabírat pouze malou část celkového řešení. Chceme-li s žáky řešit rozsáhlejší problém kvůli nějakému malému, ale zajímavému jádru, je výhodné jim „omáčku" předem naprogramovat, aby se pak mohli soustředit na to, co je chceme naučit, ale aby na druhou stranu vnímali začlenění vysvětlované konstrukce do celku a příklad nedegradoval na nějaký AHA-příklad.
- *Předkládat řešení netriviálních problémů*
  Největší slabinou řadového absolventa není neznalost programových konstrukcí, ale neschopnost řešit složité úlohy. V životě se přitom s jinými úlohami téměř nesetká. Musíme je proto naučit řešit složité úlohy již v rámci výuky.

2.2 Jak začít

Tady narážíme na velký problém: abychom studenty naučili napsat byť jednoduchý objektový program, museli bychom jim toho vysvětlit tolik, že by to většinu z nich přestalo cestou bavit. Musíme na to oklikou, při níž si studenti s objekty nejprve „hrají" a když pochopí jejich základní vlastnosti a seznámí se s nejdůležitějšími termíny, tak si zkusí nějaký ten vlastní objekt naprogramovat.

Metodika vychází ze stejných počátečních premis, z jakých jsem vycházel při tvorbě metodiky výuky za pomoci Karla a později Baltíka. Platí známé pravidlo: čím nezkušenější je uživatel, tím chytřejší musí být počítač. Je proto třeba připravit studentům předem nějaké prostředí (případně i několik), v nichž budou i jednoduché programy provádět zajímavé věci.

Ani objektové programování proto nezačínám učit na zelené louce (ač je to u učebnic programování stále nepsaný standard), ale připravím studentům prostředí, ve kterém se nejprve pohybují a pro něž později své programy tvoří. Při veškeré výuce využíváme toho, že studenti nemusí hned od počátku vytvářet kompletní programy, ale že stačí, když vytvoří pouze drobný doplněk do nějakého již existujícího světa. Doplněk, který viditelně rozšíří možnosti tohoto světa a zvýší tak jeho „kvalitu". I když toho žáci na počátku vědí ještě velice málo, už mohou vytvářet relativně efektní (a hlavně pro ně zajímavé) programy.

Vlastní vstup do světa objektově orientovaného programování proto probíhá ve třech na sebe navazujících etapách:

1. V první, prohlížecí a hrací etapě se studenti seznámí s pojmy třída a objekt a prohlédnou si strukturu nějakého předem připraveného programu. Ujasní si vzájemné závislosti a interakce jednotlivých tříd a jejich instancí. Protože je program předem připravený, nemusí být triviálně jednoduchý (alespoň z jejich pohledu).

   V této etapě si studenti především ujasní to, že vše, co v programu vystupuje, je objekt, včetně toho, co by v běžném životě za objekt nepovažovali – např. vlastnosti jako je barva nebo směr.

2. V druhé, příkazové etapě si vyzkouší vytvořit instance tříd a zkusí s těmito instancemi pracovat. Volají jejich metody, analyzují jejich chování a zkoumají hodnoty jejich atributů. Hlavním cílem této etapy je ujasnění si základního vztahu mezi třídou a její instancí (objektem). Studenti si sami vyzkouší, že jedna třída může mít (většinou) řadu instancí. Zjistí, že nemohou volat metody instancí, dokud žádná instance neexistuje, ale že metody třídy mohou volat i před vznikem prvé instance. Seznámí se s atributy instancí a tříd a ujasní si jejich vliv na vlastnosti a následné chování těchto instancí a tříd.

   V pozdějších fázích výuky se takto (tj. hraním si s hotovým programem) seznámí i se základními projevy polymorfizmu a ověří si, že není možné vytvářet instance abstraktních tříd ani rozhraní, i když je možné jejich objekty předávat jako parametry.

3. V třetím etapě začnou studenti programovat. Postupně vytvářejí vlastní třídu a její definici upravují tak, aby třída a její instance uměly totéž, co jiné třídy a instance v daném projektu. Základem úspěchu přitom je to, že jejich nově vytvářené třídy mohou od počátku spolupracovat s ostatními třídami v projektu včetně těch předefinovaných.

   OOP nám v tomto snažení vychází velmi vstříc, protože umožňuje, aby studenti již na samém počátku vytvářely programy, které se přirozeně zapojí do předem připravené větší aplikace a rozšíří tak její možnosti a schopnosti. Navíc hned od počátku pracují se složitými (alespoň pro ně) programy. „Inteligence“ vytvářených programů je pak logicky mnohem vyšší, než bývá v úvodních lekcích zvykem.

3 VLASTNÍ VÝUKA PROGRAMOVÁNÍ

Jakmile studentům proniknou základy objektově orientovaného přístupu „do krve“, můžeme se s nimi vydat do dalších oblastí a postupně před nimi odkrývat další a další oblasti objektově orientovaného programování. Podívejme se nyní na mnou používaný postup.

3.1 Jednoduché metody a atributy

Studenti se nejprve naučí naprogramovat to, s čím se v předchozí etapě seznámili, tj. jednoduché metody a atributy. Od samého počátku přitom pracují i s atributy obsahujícími odkazy na instance objektových typů. Postupně se dozvědí, jak předávat metodám parametry a jak definovat metody vracející hodnotu. Přitom pracují s hodnotami jak primitivních, tak objektových datových typů.

3.2 Automatizované testy, TDD

Po osvojení si základních syntaktických pravidel se studenti naučí vytvářet automatizované testy, aby mohli všechny své budoucí programy průběžně testovat. Seznámí se s metodikou vývoje programů řízeného testy a vyzkouší si vše na svých dosavadních programech.

V průběhu další výuky nemusejí studenti vždy vytvářet testovací třídy svých tříd. Často dostanou testovací třídy předpřipravené a mají za úkol definovat svoji třídu tak, aby těmto testům vyhověla. Získají tak zkušenosti s vývojem řízeným testy a budou proto ochotnější tuto metodiku použít i u programů, které vytvářejí na zelené louce.

3.3 Návrhové vzory

Poté se dozvědí, co to jsou návrhové vzory a seznámí se s několika nejjednoduššími návrhovými vzory, s jejichž implementacemi buď již pracovali a nebo vzápětí začnou pravidelně pracovat (knihovní třída, tovární metoda, jedináček, přepravka). Pro zjednodušení přitom terminologicky nerozlišuji klasické návrhové vzory (např. jedináček) od idiomů (např. knihovní třída či jednoduchá tovární metoda). Cílem je zafixovat žákům vzory řešení a ne je škatulkovat.

V průběhu zbytku kurzu se budou postupně seznamovat s dalšími a dalšími návrhovými vzory. Návrhové vzory totiž nejsou podle této metodiky vysvětlovány zvlášť v nějakých nadstavbových kurzech, ale od chvíle, kdy se žáci seznámí se základy syntaxe, se návrhové vzory stanou integrální součástí výuky, celou ji prostupují a neustále se na ně odvolávám.

3.4 Rozhraní

Další velká pasáž je věnována rozhraním a možnostem jejich použití v programu. Celé moderní programování dává přednost používání rozhraní před předčasným používáním tříd a jejich dědičnosti. Drtivá většina učebnic (bohužel včetně [3]) však žákům vysvětlí nejprve dědičnost tříd a teprve pak je seznámí s rozhraními. Navíc často autoři a instruktoři prezentují rozhraní jako něco, co má alespoň částečně nahradit některé nedostatky dědičnosti – např. neexistenci násobné dědičnosti.

Pokud se učebnice a kurzy soustředí především na syntaxi, je to do jisté míry jedno. Rozhodnete-li se ale žáky učit doopravdy programovat, je nanejvýš vhodné výklad rozhraní výkladu tříd předřadit.

Rozhraním se věnujeme opravdu dlouho a ukazujeme si nejrůznější příklady jejich použití. Studenti se přitom seznámí s několika dalšími návrhovými vzory (služebník, zástupce, stav) a naučí se poznávat situace, kdy architektura budovaného programu s výhodou využije zavedení správného rozhraní.

3.5 Strukturované programové konstrukce

V učebnici [6] jsem v tomto místě pokračoval výkladem dědičnosti tříd. Zkušenosti posledního roku ale ukazují, že bude možná výhodnější seznámit studenty v tomto okamžiku s klasickými programovými konstrukcemi, tj. s podmínkami, přepínači a cykly. Přiznávám, že v této otázce ještě nemám jasno.

3.6 Práce s kontejnery a poli

Po výkladu základních programových konstrukcí pokračujeme výkladem práce s kontejnery. Začínáme prací s dynamickými kontejnery, protože je jednodušší a univerzálnější. Žáci se zde mohou v klidu seznámit s pojmem kontejneru a osvojit si jeho základní vlastnosti, aniž by se museli zabývat indexy, které mnohé z nich zpočátku poněkud matou.

Následně pak probereme i práci s poli jako statickými kontejnery a ukážeme si, v jakých situacích je lepší volit dynamické kontejnery a kdy je výhodnější sáhnout po polích. Ukážeme si také, že klasické pole je vlastně mapa, v níž jsou jako klíče použita celá čísla.

3.7 Dědičnost tříd

Jak jsem řekl v podkapitole o strukturovaných konstrukcích, v současné době se přikláním k tomu, prodloužit co nejvíce práci se samotnými rozhraními a vložit výklad dědičnosti tříd až na závěr základního kurzu. Tuto koncepci ale zatím teprve připravuji a zkouším na dětech v kroužcích. Základní běhy profesionálních kurzů prozatím stále běží podle metodiky publikované v [6]. Budu vděčen za jakýkoliv názor na toto uspořádání.

3.8 Výjimky

Pokud bychom přesunuli výklad dědičnosti tříd až za výklad strukturovaných konstrukcí, museli bychom přesunout i výklad výjimek, protože koncepce výjimek je na dědičnosti postavena.

3.9 Závěrečný příklad

V dosavadním průběhu kurzu studenti vždy vytvářeli své programy jako součást nějakého rozsáhlejšího projektu. Přitom jim byly průběžně vštěpovány zásady, podle kterých se mají takové projekty navrhovat. V závěru základního kurzu mají studenti za úkol navrhnout a realizovat vlastní projekt, který nebude součástí nějakého předpřipraveného většího celku, ale navrhnou jej celý. Z předpřipravených tříd mají k dispozici pouze knihovní třídy a jejich instance.

4 REAKCE FREKVENTANTŮ KURZŮ

Zavádění a průběžnému vylepšování uvedené metodiky se věnuji již tři roky. Díky charakteru našich kurzů se mi podařilo za tu dobu nasbírat řadu zkušeností.

4.1 Charakteristika našich kurzů

Při zavádění a testování nové metody výuky mají pro mne jako metodika kurzy pořádané naší firmou oproti kurzům na různých školách několik výhod:

- Naše kurzy jsou intenzivní celodenní, takže nám studenty nic nerozptyluje a můžeme během týdenního či čtrnáctidenního kurzu probrat látku jednoho až dvou semestrů (záleží na typu školy, s níž se porovnáváme).
- Kurzy jsou týdenní až čtrnáctidenní, takže zkušenosti, které vyučující na škole získá za pololetí či za rok můžeme my získat za jeden či dva týdny a program dalších kurzů podle těchto zkušeností upravit.
- Protože nechceme látku pouze odpřednášet, ale chceme ji frekventanty doopravdy naučit, přijímáme do kurzů maximálně 4 posluchače, kterým se pak lektor intenzivně věnuje. Vyučující tak naváže s posluchači mnohem těsnější kontakt, než je v běžných kurzech obvyklé.
- Každý půlden teoretické výuky je následován samostatnou prací, kde studenti ukáží na řešení zadané úlohy, nakolik si doposud probranou látku osvojili a jsou ji schopni samostatně aplikovat. První, jednodušší úlohu dostávají po obědě, druhou pak řeší jako noční domácí úkol. Tím získáváme obdobnou zpětnou vazbu jako školy.

To vše tu neříkám proto, abych se chlubil, ale proto, abych vás přesvědčil, že závěry, o nichž budu za chvíli psát, nemám vycucané z prstu, ale jsou podložené poměrně intenzivní zkušeností z výuky řady posluchačů, jejichž práci a způsoby uvažování jsem měl možnost pozorovat z daleko větší blízkosti, než jak je tomu zvykem v běžných, masových kurzech.

4.2 Typy frekventantů

Do našich kurzů objektového programování v jazyku Java se hlásí především dva druhy zájemců:

- Ti, kteří nikdy neprogramovali a pokud se je ve škole někdo snažil něco z programování naučit, úspěšně to zapomněli (alespoň to o sobě tvrdí a hlásí se proto do kurzů, které jsou o pár dnů delší a dražší). Nejčastěji to bývají administrátoři nebo technici, kteří cítí, že by si jako programátoři mohli vydělat víc.
- Ti, kteří již mají s programováním své zkušenosti a vytvořili již několik programů. Doposud však programovali pouze strukturovaně, byť často používali jazyky, které práci s objekty umožňují (Visual Basic, PHP) nebo dokonce OOP plnohodnotně implementují (C++, Delphi). Do kurzu se hlásí proto, že zjistili, že jim dosavadní znalosti nestačí a chtějí-li si udržet či zlepšit své současné postavení, potřebují se naučit programovat objektově.
- Pomalu začíná sílit i třetí skupina: zájemci, kteří již programují objektově (alespoň si to o sobě myslí), ale chtěli by si své znalosti a dovednosti prohloubit.

4.3 Zkušení programátoři

Pro mnohé z vás bude asi překvapivé, že se zkušenými programátory je to těžší. Odmítají totiž na počátku přistoupit na to, že by OOP vyžadovalo zcela jiný přístup k řešení problému, jiné myšlení. Podle nich je programování jen jedno – to které se naučili. Vždyť jejich programy fungovaly, takže musely být naprogramované správně! Objektové technologie jsou podle nich jen další rozšiřující konstrukcí na úrovni *if* nebo *while*.

V kurzu se proto nějakou dobu snaží interpretovat předváděné objektové konstrukce pomocí prostředků, na jejichž používání byli doposud zvyklí. Dokud se jim to daří (a moje metodika je proto upravená tak, aby to moc dlouho netrvalo), domnívají se, že se jenom naučí novou syntaxi a pár nových obratů.

V průběhu druhého dne školení však začnou ztrácet půdou pod nohama a nejpozději třetí den přiznají, že OOP opravdu vyžaduje zcela jiný způsob myšlení a snaží se rychle zpětně přeinterpretovat vše, co jsme do té doby probrali a tím také změnit způsob, jakým si to uložili

do paměti. (Jak prohlásil jeden z posluchačů: „Za první tři dny kurzu jsem se naučil víc, než za předchozí půlrok intenzivního samostudia.“)

4.4 Začátečníci

Se začátečníky je život jiný. Ty není třeba nic odnaučovat, takže nenastává ona mezifáze, kdy musí znovu přehodnocovat to, co si už jednou nějak zasadili do paměti. Začínající účastníci našich kurzů mívají jiné problémy: počátky výuky vnímají jako nezávazné hraní a odmítají si přiznat, že když v této fázi něčemu zcela neporozumí, bude jim to v dalších fázích chybět. V dalších fázích výuky se proto musí občas vrátit zpět a něco si ještě jednou ujasnit a především pak si připomenout terminologii.

5 ZÁVĚR

Ve svém příspěvku jsem se vás pokusil seznámit s důvody, které mne vedly k vytvoření nové metodiky výuky programování, se základní koncepcí této výuky a se zkušenostmi s její aplikací v profesionálních kurzech.

V tomto příspěvku je popsána pouze metodika stojící za základním během kurzu. Na tento kurz navazují dva nadstavbové kurzy:

- Kurz prohloubení práce s jazykem Java, který vychází z toho, že základy OOP již studenti zvládli a můžeme se v něm proto soustředit na výklad nejrůznějších specialit jazyka a ukazovat jejich aplikaci v nejrůznějších projektech.
- Kurz prohloubení znalostí a dovedností při objektově orientovaném programování. V tomto kurzu probereme zbylé návrhové vzory a seznámíme se s dalšími zásadami, které je třeba při návrhu objektově orientovaných programů dodržovat.

Úvodní kurz i nadstavbový kurz objektového programování jsou primárně koncipovány jako jazykově nezávislé a je proto možno je přizpůsobit pro výklad většiny moderních objektově orientovaných jazyků. Stačí pouze převést do příslušného jazyka projekty, do kterých budou studenti doplňovat zadané součásti.

LITERATURA

[1] GAMMA E.; HELM R.; JOHNSON R.; VLISSIDES J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. Český překlad vydalo v roce 2003 nakladatelství Grada pod názvem *Návrh programů pomocí vzorů – Stavební kameny objektově orientovaných programů* (ISBN 80-247-0302-5).

[2] FREEMAN E.; FREEMAN E.: *Head First Design Patterns*. O'Reilly, 2004. ISBN 0-596-00712-4.

[3] BARNES D. J.; KÖLLING M. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2002, ISBN: 0-13-044929-6.

[4] VAN HAASTER K.; HAGAN, D.: *Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool*. Information Science + Information Technology Education Joint Conference. Rockhampton, QLD, Australia, June 2004.

[5] Jirka Hradil blog, červen 01. 2004: *Školení v Javě – rychlý start pro začátečníky*? *http://www.hradil.cz/2004/06/kolen-v-jav-rychl-start-pro-zatenky_01.html*

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

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