# METODIKA VÝUKY PROGRAMOVÁNÍ NA ROZCESTÍ

**Rudolf Pecinovský**

*The paper points out the differences between the taught programming and the programming used in the daily praxis. In the first part it overviews the main changes in programming passed through last 30 years. In the second part it discussed some other differences between the nowadays programming and the programming in the eighties. The third part shows that in many schools all these changes have a very little impact on methodology of teaching. In these schools the students are still prepared for programming in eighties. The last part suggests the most important changes, which should be made in the methodology, and introduces the methodology "Design Patters First", which tries to respect all the needs of modern programming.*

## Nejprve trocha historie

Za těch téměř 30 let, co učím programování, prošla tato disciplína několik revolucemi.

### 1.1 Programování opouští univerzity a specializované firmy

První z nich nastala na počátku osmdesátých let s nástupem mikropočítačů. Tehdy programování opustilo univerzity a specializované firmy a začaly se mu věnovat masy nadšenců jako svému hobby. Výuka programování se postupně začala prosazovat na středních a posléze i na některých základních školách. Mnozí z učitelů (troufám si tvrdit, že na počátku dokonce většina), kteří v této době začali učit programování, byli nadšení samouci opojení možností naučit počítač poslouchat jejich příkazy. Živelný způsob programování, který tehdy v komunitě amatérských programátorů převládal, připomínal stav, který ve světě profesionálního programování panoval až do počátku sedmdesátých let.

### 1.2 Přechod ke strukturovanému programování

Abychom mohli naučit děti v zájmových kroužcích správným návykům, převzali jsme tenkrát ze Standfordu Pattisův svět robota Karla (viz např. [6]), upravili jej tak, aby byl aplikovatelný i při výuce dětí, doplnili jej příslušnou metodikou a s její pomocí začali prosazovat výuku strukturovaného programování.

Přechod od živelného ke strukturovanému programování bychom mohli označit za druhou programátorskou revoluci. Programátorský svět před ní a po ní se výrazně lišily.

### 1.3 Objektově orientované programování

Za hranicemi však na počátku osmdesátých let zažíval programátorský svět jinou revoluci. Po prvních velice slibných zkušenostech s objektově orientovaným programová-

ním v jazyce *Smalltalk* začaly v druhé polovině sedmdesátých let vznikat objektově orientované dialekty dalších významných programovacích jazyků. Průlomovým se stalo uvedení jazyk C++ na počátku osmdesátých let. Tento jazyk vyvedl objektově orientované programování ze světa univerzit mezi řadové profesionální programátory. Ti v něm dostali do rukou jazyk, který jim umožnil psát nadále programy v jejich starém dobrém „céčku“, ale současně jim umožnil experimentovat s třídami a objekty. Postupně odhalovali jeho výhody a zjišťovali, že jak může tato nová technologie výrazně zvýšit produktivitu jejich práce.

Osmdesátí léta byla léty experimentů a omylů. Špičkoví programátoři postupně odhalovali, že pouhé používání tříd a objektů může sice výrazně zvýšit produktivitu programování, ale jejich špatné použití vede u rozsáhlejších aplikací k obrovským problémům. Tyto problémy sice autoři oněch problémových aplikací často sváděli na použití OOP, ale skutečnost byla jiná: hlavním problémem bylo **špatné použití OOP**.

Naučíte-li člověka, který v životě neviděl auto, jezdit v automobilu se zamknutou zpátečkou, bude sice nadšený, protože to pro něj bude mnohem pohodlnější než chodit pěšky, ale jeho nadšení opadne ve chvíli, kdy se dostane do slepé uličky, ve které se nebude umět otočit. Problémem těch, kteří pomlouvají OOP je právě většinou to, že se v něm ještě nenaučili zařadit jinou rychlost než zpátečku.

## 1.4 Návrhové vzory

Druhá polovina osmdesátých a počátek devadesátých let byly dobou hledání a objevování skutečného OOP. Symbolickou tečkou za tímto obdobím hledání bylo vydání knihy [5], jejíž autoři definovali některé klíčové zásady OOP a uvedli mezi programátorskou veřejnost koncept *návrhových vzorů*. Kniha se stala rázem bestsellerem a na špičce žebříčku nejprodávanějších a nejvíce citovaných knih se drží dodnes.

Vydání knihy [5] a především pak myšlenka návrhových vzorů odstartovala novou revoluci – revoluci zavádění návrhových vzorů do dalších a dalších oblastí programování. V naší republice se však tato revoluce poněkud opozdila. Na vině je částečně to, že překlad oné slavné publikace se překladateli příliš nepovedl, ale především to, že značná část učitelů považuje OOP stále za jakýsi dočasný výstřelek či za magii, která se může učit pouze na vysokých školách.

Věřte, že mám vyzkoušeno, že přístupně vysvětlené OOP chápou bez problémů i 10leté děti. Dokonce si troufám tvrdit, že je chápou lépe než dospělí či jejich starší spolužáci, kteří již mají své uvažování zkažené strukturovaným programováním, a kupodivu často lépe než zkušení profesionální programátoři, kteří se do našich kurzů přijdou přeškolovat z klasického strukturovaného programování na programování objektově orientované (podrobněji viz např. [11]).

## 1.5 Java

V témže roce, kdy byla vydána zmiňovaná kniha o návrhových vzorech, byl uveden také programovací jazyk Java. V univerzitních kruzích vzbudil okamžitě značné nadšení. Konečně zde byl jednoduchý programovací jazyk, který vycházel vstříc všem zásadám moderního, tj. objektově orientovaného programování. Jeho programy byly navíc platformně nezávislé, takž nebyl problém je začít vyvíjet pod jedním operačním systémem, pokračovat pod druhým a spustit je pod třetím.

Během několika let se Java stala hlavním jazykem vstupních kurzů programování na většině světových univerzit a její jednoduchost přivedla k jejímu zavedení do výuky i značnou část středních škol (teď hovořím o světě, ne o nás).“

Pozice Javy jako vstupního programovacího jazyka prozatím vypadá neotřesitelná. Na loňské konferenci *Innovation Technology in Computer Science Education* charakterizoval její pozici jeden přednášející slovy: „Nyní, po deseti letech zkušeností s používáním Javy ve výuce, bychom dokázali navrhnout vhodnější jazyk pro výuku. Všechny firmy po nás ale chtějí programátory ovládající Javu, tak se o návrh a zavedení nějakého alternativního jazyka pro výuku programování ani nesnažíme.“

### 1.6 Současnost

V současnosti je v plném kvasu další revoluce, při níž se navzájem ovlivňují (a přebírají své konstrukce) klasicky koncipované a skriptovací programovací jazyky. Na její výsledky si však ještě budeme muset chvíli počkat. Jedna věc je však již dnes jistá: v této revoluci nejde o nahrazení OOP nějakou jinou technologií, ale o optimální využití těch jeho vlastností, které byly doposud trochu opomíjeny.

## Změny preferencí

Neměnily se pouze technologie programování, měnily se i úkoly, které museli programátoři řešit, a současně s nimi se měnily i požadavky zaměstnavatelů na vlastnosti a schopnosti jejich programátorů.

V [7] jsem uváděl stručnější verzi tabulky 1, v níž jsem se snažil vystihnout odchylky programování přelomu 70. a 80. let, kdy jsem se učil programovat já, a programování současného. Dovolím si ji zde znovu připomenout a o některých změnách se rozhovořit podrobněji.

**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<br>Klíčovou úlohou je návrh architektury systému |
| Při vývoji programů se kladla váha především na jejich efektivitu | Při vývoji programů se klade váha především na jejich spravovatelnost a modifikovatelnost |
| Metodika vývoje programů počítala s pevným zadáním | Zadání většiny vyvíjených projektů se v průběhu vývoje neustále mění |

## 1.7 Algoritmizace

Současní programátoři při řešení současných úloh již většinou nemusí znát do hloubky taje algoritmizace, protože tyto znalosti při své práci prakticky nikdy nepoužijí. Když se pak jednou za čas objeví nějaký složitější problém, na jehož řešení nevystačí dostupné knihovny, bývá beztak výhodnější najmout si na jeho řešení specialistu.

Současné programátorským firmám již nevadí, že jejich nastupující zaměstnanec nezná rozdíl mezi tříděním metodou *QuickSort* a *HeapSort*, nebo že neví, jak nejlépe implementovat seznam. Je spíš zajímá, nakolik má nastupující programátor zažité OOP a zda umí používat návrhové vzory. Optimalizované metody pro třídění vektorů stejně jako různé implementace seznamů najde v knihovně.

Stejně tak již firmám neimponují programátoři, kteří dokáží navrhovat perfektně optimalizovaná řešení, pokud jejich programy nejsou současně přiměřeně čitelné pro jejich kolegy. Jejich skvěle optimalizované programy totiž bude muset někde udržovat i poté, co onen geniální programátor odejde jinam.

## 1.8 Respektování zákazníka

Při výuce programování si musíme ujasnit, zda chceme vychovávat specialisty, kteří budou vytvářet vysoce efektivní programy, ale nebudou ochotni akceptovat zákazníkovy vrtochy, protože by pak museli zbořit svojí původní dokonalou koncepci, anebo zda budeme chtít vychovávat programátory, kteří se budou schopni programováním doopravdy živit a kteří budou nejen ochotní, ale také schopni svůj program kdykoliv upravit podle průběžně se měnících požadavků zákazníka.

Ze své konzultační praxe jsem se dozvěděl již o třech firmách, které musely rozpustit svůj tým špičkových programátorů z MFF UK a nahradit je programátory z méně elitních škol. Ti sice možná nebyli tak geniální, ale na druhou stranu byli ochotni přizpůsobovat svá řešení neustále se měnícím potřebám zákazníků. Takoví programátoři (i při stejných platech) vydělají dané firmě daleko víc, a proto si je také jinak považuje.

## 1.9 Modifikovatelnost kódu

Specialisté budou stále potřeba, avšak zdaleka ne v takovém množství, jaké se v současné době snaží naše školy vyprodukovat. Naopak programátorů schopných přiměřeně efektivně řešit standardní úkoly současnosti produkují naše školy stále žalostně málo. Kolik škol např. učí, jak vyvíjet projekt, jehož zadání se v průběhu vývoje neustále mění a upřesňuje? Kolik škol učí, jak navrhnout programy tak, aby jejich další úprava vyvolaná předem očekávatelnými změnami zadání byla relativně bezbolestná? A tak bych mohl pokračovat ještě dlouho.

Uvědomme si, že jedinou věcí, na kterou se můžeme při vývoji programů spolehnout, je vědomí, že brzy bude všechno jinak. Firmy nyní hledají programátory, kteří by uměli pracovat s nejčastěji používanými rámci a kteří dokáží do výsledného programu relativně rychle a efektivně zapracovat nečekanou změnu v zadání. Proto musejí být jejich programy přehledné, protože pak může potřebné změny vložit do programu i kdokoliv jiný než jeho tvůrce.

## 1.10 Refaktorace

Učebnice programování, které jsem studoval v mládí, vysvětlovaly, že každému programu musí předcházet velice důkladná analýza, protože každé opomenutí v analýze se

pak velice prodraží. Čím později chybu odhalíme, tím dražší bude její oprava, přičemž tato závislost byle prezentována ne jako lineární, ale jako exponenciální.

V průběhu let se však přišlo na to, že nezávisle na tom, jak důkladně bude provedena analýza, se vždy objeví něco, kvůli čemu se bude zadání měnit. Vznikly proto metodiky refaktorace hotových programů, které tento problém řeší (viz např. skvělá [a také skvěle přeložená] publikace [4]). Tyto metodiky ukazují, jak je třeba programovat, aby oprava chyby či změna programu vyvolaná změnou zadání byla levná nezávisle na době, kdy je potřeba ji udělat.

### 1.11 Testování

Další oblastí, v níž se pohled na správné programovací postupy výrazně změnil, je testování programů. Donedávna připadalo všem přirozené, že program se nejprve napíše a pak se otestuje. Problém byl ale v psychologii programátorů a v psychologii jejich manažéru. Jak se termín odevzdání blížil a případně jak se následně vzdaloval, byl na programátory vyvíjen stále větší tlak, aby dílo co nejdříve odevzdali i za cenu toho, že nebude zcela otestované.

K tomu přistupovala další drobnost: programátor, který vytvořil program, většinou netestoval, nakolik výsledný program vyhovuje původnímu zadání, ale testoval, zda funkce, které naprogramoval, pracují podle jeho předpokladů. Ty se však občas od předpokladů zákazníka dost lišily.

Řešení tohoto problému nabízí metodika *Programování řízené testy* (viz [2]), která doporučuje nejprve napsat testy prověřující požadovanou funkčnosti programu, a teprve pak vytvářet příslušný program. Programátor se pak celou dobu snaží o jediné: aby my začaly chodit testy. Praxe ukazuje, že důsledná aplikace této metodiky opět výrazně zvyšuje produktivitu. Navíc řada firem dává tyto testy odsouhlasit zákazníkovi a začleňuje je do smluv, aby zákazníkům zabránila průběžně „drobně“ měnit zadání.

## 2 Co na to výuka

Programování se dramaticky vyvíjí, požadavky softwarových firem se mění, ale výuka programování (alespoň z mého pohledu) na většině míst stagnuje. Přiznejme si, že na minimálně většině našich středních škol a i na řadě vysokých škol se programování stále učí podle požadavků z první poloviny 80. let.

### 2.1 Výuka algoritmizace a soutěže v programování

Těžiště výuky spočívá ve výuce algoritmizace a školy soutěží v tom, čí studenti dosáhnou lepších výsledků na různých olympiádách a programátorských soutěžích. Přiznejme si, že programátorské soutěže jsou na tom obdobně jako testy inteligence. Ty také neměří nic jiného, než schopnost řešení testů inteligence. Všichni víme, že korelace naměřeného IQ s úspěšností v praktickém životě není tak velká, jakou by ji rádi viděli autoři IQ testů.

Absolventi, kteří dokáží skvěle řešit různé algoritmické lahůdky na soutěžích, mohou mít velké problémy v praxi, kde je zaměstnavatel postaví před obludný program vytvořený jejich předchůdci a zadá jim úkol doplnit program nějakou další funkcí nebo odladit nějakou jeho problematickou pasáž.

## 2.2 Metodika programování

Dalším problémem je vyučovaná metodika programování. Všechny větší projekty jsou dnes programovány objektově. Prakticky žádná významnější softwarová firma si již nedovolí řešit jen trochu rozsáhlejší projekt neobjektově. OOP přináší vyšší produktivitu, větší spolehlivost výsledných programů a jejich snazší udržovatelnost a modifikovatelnost. Navíc je naprostá většina velkých projektů nasazována spuštěna na aplikačních serverech, které objektovou orientovanost obhospodařovaných programů vyžadují.

Přesto řada našich škol i nadále srdnatě vypouští strukturované, objektovým programováním netknuté programátory v blahé naději, že se jejich absolventi na trhu práce skvěle uplatní. Pokud se škola rozhodne zařadit výuku objektového programování, dělá to většinou až na konci kurzu, kdy už studenti nemají čas přednesenou látku dostatečně procvičit a už vůbec ne ji zažít.

Nedávno jsem hledal inspiraci pro svoji knihu a procházel jsem na internetu zveřejněné kurzy programování našich vysokých škol. Zjistil jsem, že většina z nich učí v prvním semestru strukturované programování, přesněji algoritmizaci. Jak vtipně prohlásil jeden vyučující na FEL: „Slovo objekt je v prvním semestru považováno za sprosté slovo."

O objektových programových konstrukcích se začnou zmiňovat až v druhém semestru. Při procházení zveřejněných přednášek a cvičení jsem však ke svému údivu zjistil, že někteří přednášející (a to i na renomovaných školách) ještě stále učí objektové programování „se zařazenou zpátečkou". Nelze se pak divit, že tito absolventi jsou svými zaměstnavateli následně vysíláni do našich přeškolovacích kurzů, aby se zde dozvěděli, jak je to s tím objektově orientovaným programováním doopravdy.

Řada studií už dávno prokázala, že přeškolení strukturovaného programátora na programátora objektově orientovaného trvá minimálně rok, u zkušenějšího programátora 18 měsíců (musí se toho více odnaučovat). To platí i ve výuce. OOP vyžaduje naprosto jiný přístup k řešení problému a jiné uvažování. Když už se žáci naučí programovat strukturovaně, odmítají následně změnit způsob svého uvažování. Jejich programy proto často nejsou objektově orientovanými programy, ale pouze strukturovanými programy používajícími třídy. A to je citelný rozdíl.

## 2.3 Styl programování

Problémem však není jen vyučovaná metodika programování, ale i vyučovaný styl programování. Na řadě škol razí v kurzech programování zásadu, že na programu není důležité to, jak je napsán, ale především to, jestli pracuje dostatečně optimálně. Když jsem nedávno procházel vzorové příklady, které předkládají studentům vyučující na některých našich vysokých školách považovaných všeobecně za špičkové, uvědomil jsem si, že kdyby mi takový program odevzdal student u zkoušky, měl by značné problémy. Řešení úlohy tvořené jedinou cca 50řádkovou procedurou s proměnnými a, b, c, d, e, f opravdu nepovažuji za vhodný demonstrační příklad kurzu programování.

Našim studentům vždy na přednáškách říkám: „Způsob, jak učíme programování my a jak je učí některé jiné školy, se v lecčems výrazně liší. Rozdíl spočívá v tom, že tamty školy možná učí své studenty skvěle programovat, kdežto my se vás snažíme naučit si programováním skvěle vydělávat." A evropský výzkumný projekt REFLEX prováděný Střediskem vzdělávací politiky nám dává za pravdu – absolventi naší fakulty mají největší průměrný plat ze všech absolventů všech testovaných fakult v naší republice (viz. [14]).

## 3 Jak by měla vypadat metodika výuky

Přestanu již plakat nad tím, co není dokonalé, a pokusím se připomenout některé zásady a principy, které by měla správná metodika výuky programování respektovat. Nebudu je vyjmenovávat všechny, protože jsem na toto téma už několikrát hovořil a texty jsou snadno dostupné např. na mých webových stránkách. Další pedagogické zásady se můžete dozvědět např. ve [3] nebo v [13]. Zmíním se o nich proto jen velmi stručně a navíc se zmíním jen o těch, o nichž se domnívám, že jsou v největším rozporu se současně používanou metodikou.

### 3.1 Princip ranního ptáčete

Jeden z důležitých pedagogických vzorů říká, že klíčová témata máme zařazovat pokud možno na počátek kurzů, aby studenti měli dostatek času a příležitostí je zažít a procvičit (viz např. [3]). Jedním ze základních pravidel moderního programování zmiňovaným již v [5], je zásada, že je třeba programovat proti rozhraní a ne proti implementaci. Další velmi důležitým prvkem je používání návrhových vzorů.

Podíváte-li se na běžné kurzy programování, zjistíte, že zmínku o rozhraní zařazují až někam ke konci kurzu a o návrhových vzorech se většinou nezmiňují vůbec.

### 3.2 Pracujme s obludami

Jak jsem již řekl, typickou úlohou začínajícího programátora je přidat k nějakému rozsáhlému programu další funkci nebo opravit nějakou problémovou pasáž. Programátor proto musí umět se rychle orientovat v cizím programu a znát techniky, jak upravit jednu jeho část aniž by ovlivnil práci jiných částí.

Stejně tak by měl umět přidávat nové funkce takovým způsobem, aby tím neznemožnil případné jejich další modifikace nebo přidávání dalších funkcí.

### 3.3 Nejprve architektura, potom kód

Typickým přístupem začátečníků postavených před vyřešení nějakého problému je to, že začnou okamžitě přemýšlet na úrovni jednotlivých instrukcí. Tento problém jsme řešili i při výuce podle starších metodik. Vzhledem k tomu, že značnou část strukturovaných programů lze považovat jako v programovacím jazyce zapsané posloupnosti příkladů, vedoucí k vyřešení problému.

Objektové programy mají ale trochu jinou filozofii. Mohli bychom je charakterizovat jako množinu objektů a zpráv, které si mezi sebou posílají, zapsanou v nějakém programovacím jazyce. Způsob myšlení nutný při návrhu objektového programu se od klasického liší stejně zásadně, jako se liší jejich definice.

Seznámíme-li studenty s objektovými konstrukcemi až poté, co už zažili jiný způsob uvažování a návrhu programu, velmi těžko je budeme přeučovat na takový způsob, který moderní programování vyžaduje.

### 3.4 Život je neustálá změna

Jak jsem již řekl, současné programování se vyznačuje tím, že zadání se v průběhu vývoje projektu několikrát změní. Na tuto skutečnost však studenty ve svých kurzech prakticky vůbec nepřipravujeme.

Potřebná příprava by měla sestávat ze dvou složek:

za prvé bychom je měli seznámit s nástroji a technikami, které jim tuto úlohu významně usnadní, a za druhé bychom jim měli také.

- Měli bychom je seznámit s nástroji a technikami, které jim tuto úlohu významně usnadní. Musíme proto zahrnout do výkladu návrhové vzory a základní refaktorační techniky.
- Měli bychom jim poskytnout příležitost pracovat na projektu s průběžně pozměňovaným zadáním a ukázat jim, jak v takovýchto případech postupovat.

### 3.5 Vhodné používání testů zvyšuje produktivitu

Většina studentů opouští kurzy programování s pocitem, že testy se musí dělat jenom proto, že to po nich někdo chce. Nenaučili se, že vhodná (a včasná) definice testů jim může výrazně zvýšit produktivitu.

Používání jednotkových testů by se mělo stát nedílnou součástí naší výuky. U jednodušších příkladů můžeme studentům předložit předpřipravené testovací třídy s metodami testujícími jednotlivé části jejich řešení, u složitějších můžeme být příprava testů součástí první etapy řešení, přičemž etapu vlastního řešení mohou studenti zahájit až po schválení odevzdaných testů.

### 3.6 Shrnutí: respektujme praxi

Podle mého názoru řada výše zmíněných problémů je do značné mír způsobena nedostatečným kontaktem vyučujících s firmami, které jsou v přední linii současného softwarového dění a jejich pracovníci znají nejenom své současné potřeby, ale mají i představu o tom, kam se bude programování ubírat v nejbližších letech.

Úplně vzorová byla reakce jednoho učitele na workshopu o výuce programování, který byl součástí konference *Objekty 2007*. Dotyčný vyučující vysvětloval, že je třeba nejprve důkladně probrat algoritmizaci, protože bez toho přeci není možné programovat. Vystoupili postupně dva zástupci větších softwarových firem, kteří potvrdili, že jejich programátoři opravdu žádné složitější algoritmické problémy neřeší, protože všechny klíčové problémy jsou vyřešené v knihovnách a těžiště práce programátorů leží jinde. Daný vyučující se však nedal zviklat a prohlásil, že to tak není., že bez hluboké znalosti algoritmizace nic nenaprogramují.

## 4 Metodika *Deign Patterns First*

Přestanu již plakat nad současným stavem výuky programování a prozradím vám dvě metodiky, které se tento problém snaží řešit.

Na přelomu století se ve světě začala prosazovat metodiky *Object First*, která začíná výukou práce s objekty a teprve po čase přidává algoritmické konstrukce (viz např. [1]).

Metodika *Object First* vykročila správným směrem, ale zůstala na půli cesty. Učila totiž studenty pracovat s objekty, ale výklad rozhraní ponechávala od pozdějších kapitola a výklad návrhových vzorů vynechala úplně.

Zmíněné nedostatky se spolu s několika dalšími drobnostmi snaží řešit metodika *Design Patterns First* (viz [7], [8], [9]). Tato metodika se snaží respektovat všechny výše zmíněné principy, zásady a požadavky. Byla vyzkoušena v různých druzích kurzů počínaje

zájmovými útvary pro žáky základních škol, přes standardní výuku na středních a vysokých školách až po kurzy profesionálních programátorů.

Její příjemnou výhodou je i skutečnost, že včasným zapojením objektových principů s vhodným výběrem úloh dosáhne toho, že případní pokročilejší studenti nemohou využít svých předchozích znalostí a výchozí úroveň žáků se tak sjednotí.

Druhým příjemným důsledkem používání této metodiky je možnost automatizace vyhodnocování studentských úloh a z toho plynoucí výrazná úspora času vyučujícího.

## 5 Závěr

Příspěvek shrnul významné předěly, které prodělalo programování v posledních 30 letech, a ukázal, že tyto zásadní změny se v řadě případů nepromítly ve výuce. Shrnul základní požadavky, které by měla respektovat metodika výuky programování, aby absolventi kurzů vycházeli do praxe co nejlépe připraveni na úkoly, které je budou v praxi čekat. Na závěr představil dvě metodiky výuky, které se snaží respektovat zmíněné požadavky a trendy: Metodiku *Object First*, která ukazuje, že je výuku možné začít prací s objekty, a především pak metodiku *Design Patterns First*, která možnosti naznačené předchozí metodikou dále dopracovává a ukazuje, že spolu s objekty je možné na začátek výuky zařadit i výklad rozhraní a návrhových vzorů.

## Literatura

Texty článků, které jsem psal, si můžete stáhnout na adrese http://vyuka.pecinovsky.cz


[1] Barnes D. J., Kolling M. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2002, ISBN: 0-13-044929-6

[2] BECK, Kent. *Test Driven Development By Example*. Addison-Wesley © 2003. 220 s. ISBN 0-321-14653-0. (Překlad: BECK, Kent. *Programování řízené testy*. Grada © 2004. 204 s. ISBN 80-247-0901-5)

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

[4] FOWLER, Martin. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, © 2000. 430 s. ISBN 0-201-48567-2. (Překlad: *Refaktoring. Zlepšení existujícího kódu.* Grada, © 2003. 394 s. ISBN 80-247-0299-1)

[5] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. ISBN 0-201-30998-X (Překlad: *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.* Praha: Grada, © 2003. 386 s. ISBN 80-247-0302-5.)

[6] PATTIS Richard E.: *Karel The Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, 1981. ISBN 0-471-59725-2.

[7] PECINOVSKÝ, Rudolf: *Aplikace metodiky Design Patterns Fist*. Objekty 2006 – sborník příspěvků jedenáctého ročníku konference, ČZU Praha, 2006. ISBN 80-213-1568-7.

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

[9] 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.

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

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

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

[13] *The Pedagogical Patterns Project.* http://www.pedagogicalpatterns.org/

[14] Výzkumný projekt *REFLEX - flexibilita odborníků ve společnosti znalostí*. Středisko vzdělávací politiky. http://svp.pedf.cuni.cz/index.php?id=8

## Autoři

Ing. Rudolf Pecinovský, CSc.
Amaio Technologies, Inc.
VŠE Praha, Fakulta informatiky a statistiky, Katedra informačních technologií
rudolf@pecinovsky.cz