MODERNÍ PROGRAMOVACÍ TECHNIKY

Rudolf Pecinovský
ICZ a.s., Hvězdova 1689/2a, 140 00 Praha 4,
Katedra informačních technologií VŠE Praha
rudolf@pecinovsky.cz

ABSTRAKT:
Umění programovat zahrnuje celou řadu znalostí a dovedností. Vedle všeobecně uznávané nutné znalosti syntaxe použitého programovacího jazyka musí programátor zvládnout ještě celou řadu dalších. Spektrum těchto nadstavbových znalostí a dovedností se však v průběhu doby mění. Příspěvek se zamýšlí na znalostmi a dovednostmi, které vyžaduje současné moderní programování. Probírá požadovanou úroveň znalostí návrhových vzorů, refaktorace, jednotkových testů, vývojových prostředí, systémů pro správu verzí, optimalizačních nástrojů a dalších programů a technik.

KLÍČOVÁ SLOVA:
Výuka programování, objektově orientované programování, návrhové vzory, refaktorace, jednotkové testy, profilace, správa verzí.

1 ÚVOD

Když jsem se zhruba před 30 lety začal učit programovat, dostal jsem od svého vyučujícího postupně několik knih vysvětlujících syntaxi jednotlivých jazyků a měl jsem za úkol vytvořit v každém z nich dva jednoduché programy. Žádní jiná znalost, než znalost syntaxe použitého jazyka, se tehdy od programátora neočekávala.

Časem se mi do rukou dostala kniha nazvaná *Programovací techniky*, z níž jsem se dozvěděl, jak je možno rychle třídit pole a soubory, jak se vytvářejí spojové seznamy a řadu dalších užitečných dovedností. Následně se mi dostala do rukou kniha popisující principy strukturovaného programování a vysvětlující, jak programovat přehledně, efektivně a spolehlivě a především mne seznámila s posledními trendy v programování. Tyto knihy mi ukázaly rysy programování, které se v tehdejší době na školách většinou neučily, a přiměly mne, abych se začal pídit po dalších informacích podobného druhu.

Vyzbrojen znalostmi z těchto a jim podobných knih jsem brzy začal být mezi spolupracovníky (byť nezaslouženě) považován za mága, který umí vyřešit úlohy, s nimiž si běžní smrtelníci nevědí rady. V tehdejší době totiž opravdu nebylo zvykem, že by programátor uměl něco víc, než syntaxi použitého jazyka a jazyka pro komunikaci s operačním systémem.

Tento stav platil v podstatě až do revoluce. Pak si ale zaměstnavatelé začali rychle uvědomovat, že k dosažení rozumné produktivity práce musí programátoři ovládat ještě celou řadu dalších dovedností. Bohužel, tak jako se mne ve škole nikdo nesnažil naučit dovednosti, které byly užitečné v tehdejší době, tak se řada škol nesnaží naučit své studenty dovednostem, které jsou potřebné nyní (pravda, jsou výjimky, ale je jich poskrovnu). A obdobně jako se tehdy mnozí studenti museli učit principy práce počítačů, které v době výuky již dávno dosloužily, tak se i dnes některé školy snaží vybavit své studenty vědomostmi, jejichž použití absolventům v jejich budoucí kariéře nejspíše nehrozí.

Nejmarkantnější je tento problém u absolventů, kteří nastupují do zaměstnání po ukončení bakalářského studia. Přitom právě tito absolventi by měli být vybaveni pro běžné problémy praxe. Opak je všinou pravdou. V průběhu studia se dozvěděli spoustu teorie, ale o tom, jak doopravdy vypadá současné programování, jim toho jejich alma mater prozradila pramálo.

Připomeňme si některé dovednosti požadované po současných programátorech, které bychom ve smyslu výše zmíněné příručky mohli označit jako *Moderní programovací techniky*.

2 PROGRAMÁTORSKÉ DOVEDNOSTI

2.1 Znalost syntaxe použitého programovacího jazyka

Mnohé možná překvapí, že znalost používaného programovacího jazyka není pro řadu firem tak důležitá, jak by se na první pohled zdálo. Programovací jazyk je totiž většinou definován pouze malou množinou pravidel, která nebývá tak obtížné se naučit, zejména jedná-li se o jazyk vycházejícího ze stejného paradigmatu. Po vhodném zaškolení může nový programátor být během několika týdnů plnoprávným členem týmu.

2.2 Znalost používané platformy, jejích konvencí a knihoven

Na rozdíl od znalosti syntaxe programovacího jazyka bývá znalost knihoven daleko větším problémem. Současné knihovny jsou rok od roku rozsáhlejší, což činí přechod mezi platformami stále obtížnější. Knihovny jsou si navíc čím dál podobnější, což paradoxně problematiku přechodu ještě zvyšuje, protože si programátor neumí vždy hned uvědomit nuance mezi oběma přístupy a často dlouho používá v novém prostředí své zažité zvyky.

Pokud podlehnutí zaužívaným stereotypům způsobí syntaktickou chybu, je to ještě dobré. Jakmile je však rozdíl mezi oběma koncepcemi nenápadnější a projeví se až ve specifické situaci za běhu, bude odhalení takovéto chyby pro většinu zúčastněných pěkně zapeklitým oříškem.

2.3 Schopnost programovat v daném paradigmatu

Jak jsem řekl, neznalost používaného jazyka nebývá často považována za klíčový problém. Rozumně (i když obtížněji) řešitelná je i neznalost dané platformy. Mnohem náročnějším problémem však bývá změna paradigmatu. Uvádí se, že přeškolení strukturovaného programátora na objektově orientovaného trvá podle jeho zkušeností 12 až 18 měsíců – čím zkušenější, tím je doba přeškolení delší.

Různá paradigmata vyžadují zásadně odlišné přístupy k řešení problémů a setrvačnost vedoucí k aplikaci zažitých postupů přináší řadu problémů při komunikaci v týmu. Nový programátor je proto „pro jistotu“ po dobu své konverze pověřován úkoly, které nemusí zdaleka odpovídat jeho schopnostem. To zákonitě vede k dalším napětím uvnitř týmu.

S programátory, kteří se přeškolují ze strukturovaného paradigmatu na objektově orientované se pravidelně setkávám ve svých kurzech. A opakovaně se zde setkávám i s jejich údivem, proč se ve škole museli učit strukturované paradigma, když vývoj a údržba programu podle objektového paradigmatu je mnohem jednodušší a efektivnější.

Speciální kategorií jsou pak programátoři vyškolení podle zásady, že objektově programovat znamená používat třídy a dědičnost. Jejich převedení do skutečného objektově orientovaného světa bývá v mnohých případech poměrně náročné.

2.4 Schopnost algoritmizace složitějších problémů

Schopnost algoritmizace složitých úloh považují mnozí učitelé stále za jednu z hlavních programátorských dovedností. Poptáte-li se však u programátorských firem, zjistíte, že tuto dovednost po svých programátorech většinou nepožadují, nezjišťují a většinou ani nijak zvlášť neohodnocují. Naprostá většina řešených úloh totiž žádné větší algoritmické dovednosti nevyžaduje. Tuto situaci navíc posiluje skutečnost, že v současných programech jsou metody, které obsahují více příkazů (většinou se uvádí hranice mezi 10 a 20 příkazy) považovány za podezřele složité a začíná se uvažovat o tom, zda jejich složitost není způsobena tím, že dělají několik věcí najednou. Pro definici tak krátkých metod většinou není třeba žádných hlubokých algoritmických studií.

Pro firmu bývá výhodnější začlenit do svého týmu jednoho či dva specialisty na zapeklité problémy a řadové programátory takovýmito případy vůbec neobtěžovat. Pokud se totiž při řešení projektu objeví skutečně algoritmicky náročný problém, řadový programátor jej stejně

nevyřeší a musí jej přenechat specialistovi na algoritmické lahůdky. Takovýmto specialistou určitě nebude absolvent bakalářského studia a přitom právě na bakalářském stupni je výuka algoritmizace nejrozsáhlejší.

2.5 Návrh architektury programu

Samostatnou kapitolou je návrh architektury programu. Řada škol se snaží naučit své studenty řešit různé algoritmické oříšky, ale zapomíná je naučit, jak dekomponovat řešený problém na sadu jednodušších. Problém řady absolventů standardně pojatých kurzů bychom mohli charakterizovat známým rčením, že pro stromy nevidí les.

Postavíte-li je před problém, nedokáží jej rozebrat na koncepční úrovni, ale ihned spadnou o dvě úrovně níž a začnou popisovat konkrétní programovou realizaci některých funkcí. Když je zarazíte a pokusíte se je vrátit znovu na koncepční úroveň, záhy opět spadnou zpět ke kódu. Po druhém nebo třetím návratu pochopí, že v danou chvíli je úroveň kódu úrovní zakázanou, ale v koncepční hladině se pohybovat neumějí a často vůbec netuší, za který konec problém uchopit.

2.6 Návrhové vzory

S návrhem architektury souvisí i znalost návrhových vzorů, která je mnohými firmami považována za naprosto klíčovou. Návrhové vzory jsou v současné době sjednocujícím činitelem všech platforem postavených na objektovém paradigmatu. Programátoři se sice mohou „do krve pohádat“ o tom, zda je lepší C#, Java, PHP, Python či Ruby, ale všem jim je jasné, kdy je vhodné použít návrhový vzor *Jedináček*, *Interpret* či *Tovární metoda*.

Používání návrhových vzorů prohlubuje akceptaci objektového paradigmatu. Pokud do týmu, který se již s návrhovými vzory dávno sžil, přijde programátor, který se s nim nikdy nesetkal, připadá si jak v jiném světě. Nejenom že nechápe jednotlivé termíny, ale často nechápe ani důvody, které k aplikací té které programové konstrukce vedly. Odtud je už jenom krůček k tvorbě programů, které budou kolidovat s koncepcí, na níž je postaven zbytek systému.

Řada našich univerzit nabízí kurz návrhových vzorů jako výběrový kurz pro pokročilé studenty. Bohužel, při procházení publikovaných textů jsem se neubránil dojmu, že předmět je většinou chápán jako defilé návrhových vzorů bez nějaké výraznější snahy vysvětlit základní myšlenky, na kterých jsou návrhové vzory postaveny.

Jak jsem navíc řekl, tyto kurzy jsou uváděny jako výběrové. Škol, které by výklad návrhových vzorů spolu s výkladem všech hlavních rysů skutečně objektově orientovaného programování zařadily již do úvodních, povinných kurzů programování, je jako šafránu.

2.7 Refaktorace

Teorie refaktorace se objevila jako důsledek poznání, že nezávisle na tom, jak důkladná bude analýza problému, stejně se v průběhu vývoje zadání změní. Refaktorace proto ukazuje, jak upravovat program, abychom s minimální námahou a minimálním rizikem uvedli zdrojový kód do stavu, v němž se nám budou výrazně snadněji zanášet následné změny.

Umění refaktorace spočívá ve znalosti kritických vlastností programů (používá se pro ně termín *pachy v kódu*), které zvyšují pravděpodobnost budoucích problému, a ve znalosti postupů, kterými lze tyto pachy odstranit. Současné vývojové nástroje mají naštěstí většinu těchto postupů již integrovánu a umějí je na požádání spustit. Programátor ale stále musí vědět, které části programu si o refaktoraci „koledují“ a kterou jejich vlastnost je třeba eliminovat.

Jedním z problémů zařazení refaktorace do výuky je setrvačnost učitelů. Na řadě škol by totiž moderní programování označilo značnou část demonstračních programů používaných v úvodních kurzech programování za *páchnoucí* a vyžadujících refaktorační zásah. Učitelé, kteří by měli pokročilé studenty těmto technikám učit, si však nemohou dovolit „pomlouvat“ programy svých kolegů přednášejících v nižších semestrech, a proto asi dané téma raději moc nerozvířují.

2.8 Jednotkové testování

Zrod knihovny *SUnit* a jejích následovníků na konci devadesátých let minulého století způsobil programátorskou revoluci. Situaci výstižně charakterizoval Martin Fowler, který řekl: „Ještě nikdy v historii programování neovlivnilo tak málo řádek kódu tak výrazně celý programátorský svět.“ Vývojové prostředí, které v současné době nemá zabudovanou podporu pro příslušnou verzi knihovny typu *xUnit* nemá šanci na úspěch.

Jednotkové testování vychází vstříc myšlence *Programování řízeného testy*, které tvrdí, že bychom měli nejprve napsat testy a teprve pak testovaný program. Tato převratná myšlenka přinesla výrazné zvýšení produktivity programátorské práce a současně také zvýšení spolehlivosti vyvinutých programů.

Knihovny *xUnit* pak nabízej prostředky k tomu, jak většinu testování zautomatizovat, aby programátora neobtěžovalo, ale naopak mu umožňovalo rychleji nacházet a opravovat případné chyby v kódu.

2.9 Kolektivní vývoj a používání správy verzí

Používání systému pro správu verzí je v současné době nedílnou součástí práce většiny programátorských týmů. Přináší možnost efektivního návratu ke starším verzím, bezpečných oprav minulých verzí neovlivňujících verze současné, sdílení kódu skupinou programátorů a další cenné východy.

Na rozdíl od dříve popisovaných technik se používání systémů pro správu verzí dá poměrně rychle a snadno naučit. Nicméně jejich používání při přípravě projektů, které jsou integrální součástí výuky, považuji za nanejvýš žádoucí. Aby však výuka práce s těmito systémy byla opravdu efektivní, musí jít ruku v ruce s vývojem projektů v týmu, při kterém si účastníci postupně uvědomí různé výhody a nevýhody týmové spolupráce i používání systému na správu verzí.

2.10 Začlenění programu do existujících rámců

Většina úloh, které studenti na školách řeší, spočívá ve vytvoření nějaké samostatné aplikace. S takovýmto typem úlohy se však ve své budoucí praxi setkají spíše výjimečně. V praxi dostane čerstvý absolvent většinou za úkol vytvořit nějakou menší součást začleněnou do rozsáhlejšího rámce, jehož vytvoření je navíc za hranicemi jeho aktuálních programátorských schopností. Řešení takovéhoto typu úloh se však většinou na škole neučí. V praxi proto mívají absolventi často problémy s pochopením požadavků, které na ně takovýto typ úlohy klade.

2.11 Optimalizační nástroje

Průměrný student podléhá často mladické iluzi, že vhodnými zásahy do kódu vytvářený program výrazně zefektivní. Neuvědomuje si však, že část kódu, kterou tak pracně optimalizuje, nemusí mít na celkovou efektivitu programu nijak výrazný vliv. Přitom ruční optimalizace vede většinou ke snížení přehlednosti programu a zvýšení jeho náchylnosti k nejrůznějším chybám. Pro zjištění, které části programu výrazně ovlivňují celkovou dobu zpracování, slouží v praxi specializované nástroje – tzv. profilery. Studenti by se s těmito nástroji měli v průběhu výuky seznámit a uvědomit si, jak falešné někdy bývají představy o vlivu některých částí programu na jeho celkovou efektivitu.

2.12 Měření kvality programu

Při rozhovoru s vedoucími programátorských týmů, kteří k nám posílají své „ovečky“ na přeškolení či doškolení se neustále setkáváme s nářky na to, co si různí vyučující představují pod kvalitním programem. Pro firmu, která se živí vývojem softwarových projektů, začíná být životně důležitou otázka spravovatelnosti vyvíjených programů a tím i jejich přehlednosti a srozumitelnosti.

Většina studentů však odchází ze škol s přesvědčením, že u programu je vedle funkcionality nejdůležitější jeho efektivita a případně elegance řešení. O nutnosti tvorby přehledného kódu často odmítají diskutovat. Neuvědomují si, že to, že je kód přehledný, přinese firmě ve svém důsledku v řadě případů větší příjmy, než celý původně vyvinutý program.

3 ZÁVĚR

Příspěvek nastínil některé z oblastí, jejichž znalost je pro současné programátory velmi důležitá a které jsou na řadě škole zařazovány do výuky sporadicky nebo vůbec. Snažil se ukázat, proč je znalost těchto technik v praxi důležitá a proč by bylo vhodné zvážit posílení jejich intenzivnějšího zařazení do výuky.

Podle vyjádření softwarových firem se jako obzvláště problematické jeví především odborné profily středoškoláků a absolventů bakalářského studia, které jejich školy nevybavují znalostmi potřebnými pro současnou praxi, ale spíše znalostmi, které byly důležité v době, kdy studovali jejich učitelé.

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] 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., ISBN 80-247-0302-5)

[3] KERIEVSKY Joshua: *Refactoring to Patterns*, Addison Wesley, © 2005, 368 stran, ISBN 0-321-21335-1

[4] METSKER Steven John: *Design Pattersn Java Workbook.* Addison-Wesley, © 2002, 476 stran, ISBN 0-201-74397-3.

[5] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila: Order of Explanation Should be Interface – Abstract Classes – Overriding, *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education*, University of Dundee 2007.

[6] PECINOVSKÝ Rudolf: *Návrhové vzory*, Computer Press, © 2007, 528 stran. ISBN 978-80-251-1582-4.

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

[8] 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.

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

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

[11] PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java 5.0*, Grada, © 2004, 602 stran, ISBN 80-247-0941-4.

[12] STELTING Stephen, MAASSEN Olaf: Applied Java Patterns, Sun Microsystems Press, © 2002, 576 stran, ISBN 0-13-093538-7

[13] 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.