Tvorba učebnic a kurzů programování

Rudolf Pecinovský

ICZ a.s., Na Hřebenech II 1817, 140 00 Praha 4
VŠE Praha, Nám. W. Churchilla 4, 130 67 Praha 3
rudolf@pecinovsky.cz

Abstrakt. Článek rozebírá různé přístupy k tvorbě učebnic a kurzů programování a programovacích jazyků. Seznamuje se jejich výhodami a nevýhodami a ukazuje, čím se liší skutečné učebnice a kurzy programování od učebnic a kurzů, které pouze seznamují s nějakým programovacím jazykem a jeho knihovnami. Specifikuje některé zásady, které vhodné respektovat při návrhu obsahu učebnic a kurzů učících programování. Připomíná pedagogickou zásadu ranního ptáčete, podle níž se má výklad uspořádat tak, aby se nejdůležitější věci učily opravdu co nejdříve. Současně doporučuje nesoustředit se zbytečně na prvky, jejichž tvorbu postupně přebírají různé automatizované systémy, a naopak učit od samého začátku také zásady architektury. V závěru seznamuje s různými přístupy k tvorbě doprovodných programů a rozebírá jejich výhody a nevýhody. Současně upozorňuje na program, který může výrazně pomoci při tvorbě programů, které studenti v průběhu kurzu postupně vyvíjejí a zdokonalují.

1 Úvod

Je všeobecně známo, že živé jazyky a programovací jazyky mají mnoho společného. Budeme-li se chtít naučit nějaký živý jazyk, můžeme si dojít do knihkupectví, kde najdeme učebnice daného jazyka, učebnice jeho gramatiky a překladové slovníky. Pokud narazíte na učebnici psaní v daném jazyce, bývá to většinou učebnice, která se vás snaží naučit, jak psát obchodní dopisy.

Existují samozřejmě i učebnice, které vás učí doopravdy psát – mohli bychom je označit za učebnice budoucích spisovatelů. Tyto učebnice se vás však nesnaží naučit psát v nějakém konkrétním jazyce, ale naopak předpokládají, že daný jazyk již znáte a potřebujete se naučit něco z „vyšší školy psaní“. Tyto „učebnice psaní“ se většinou zaměřují na specifické druhy textů. Jedny vám vysvětlí, jak správně psát odborné příručky, další se vás snaží naučit psát povídky či romány, jiné vysvětlují, jak psát scénáře k filmům či divadelní hry.

Všechny výše zmíněné druhy učebnic živých jazyků mají jedno společné: již v titulu jasně deklarují, co se vás budou snažit naučit a opravdu se o to snaží. U programovacích jazyků je tomu bohužel jinak. Podíváme-li se do knihkupectví s oddělením knih o výpočetní technice, najdeme zde určitě řadu knih s názvy typu „Učebnice programování v jazyku xyz“. Začneme-li se však zajímat o jejich obsah, zjistíme, že převážná většina z nich učebnicemi programování není. Mohli bychom je nazvat učebnicemi kódování v daném programovacím jazyce, anebo prostě učebnicemi daného

programovacího jazyka. Převedeno do terminologie učebnic živých jazyků, jednalo by se o slovníky a učebnice gramatiky daného jazyka, v lepším případě o učebnice daného jazyka. Pouze výjimečně bychom je mohli označit za učebnice programování odpovídající výše zmiňovaným učebnicím psaní.

Tento článek se zabývá základními vlastnostmi, které by měla mít učebnice, abychom ji mohli označit za učebnici programování. Při té příležitosti seznamuje s nástrojem, který tvorbu takovéto učebnice usnadní.

Stejné vlastnosti jako učebnice by měly mít i kurzy, které se vydávají za kurzy programování. Cokoliv, co bude proto v následujícím článku řečeno o učebnicích, platí v nezměněné míře i pro kurzy.

2 Nevhodné vlastnosti učebnic a kurzů programování

Zůstaneme-li u předchozí paralely, mohli bychom přirovnat učebnice programování k učebnicím psaní. Různá programovací paradigmata bychom pak mohli přirovnat k jednotlivým autorským stylům či žánrům.

Rozhodneme-li se vytvořit učebnici psaní, určitě se nebudeme soustředit na detaily. Gramatická pravidla budeme zmiňovat spíše okrajově, a když už se o nich zmíníme, budeme nejspíše vysvětlovat, jak nám pomohou dosáhnout kýženého efektu. Obdobné to bude se slovní zásobou a dalšími „detailními“ vlastnostmi jazyka.

V učebnicích psaní se budeme naopak soustředit na celkovou koncepci vytvářeného díla a na to, jak dosáhnout kýženého efektu. Budoucí autoři detektivek se budou dozvídat, jak u čtenáře zvyšovat napětí, jak vhodně zakomponovat do textu informace, které se na počátku zadají bezvýznamné, ale na konci se ukáží jako klíčové a řadu dalších obdobných dovedností. Obdobné to bude i dalšími druhy literatury.

Koncepce učebnic je v rozporu s vyhlašovanými zásadami

Podívejme se nyní na učebnice, které o sobě prohlašují, že jsou učebnicemi programování. Zjistíme, že se většinou soustřeďují právě na detail a snaží se čtenáři ukázat, jak z jednotlivých detailních znalostí poskládat jednoduché programy. Vysvětlují, co to jsou proměnné a detailně rozebírají jejich vlastnosti. Poté probírají jednotlivé programové konstrukce (příkazy, rozhodování, cykly) a učí čtenáře vytvářet jednoduché metody. Jedná-li se o učebnice objektových jazyků, tak nakonec vysvětlí pojem třídy a předvedou, jak má každá třída definovánu svoji sadu metod, případně, jak může díky dědičnosti několik tříd své metody sdílet.

Bohužel, pouze výjimečně se tyto učebnice povznesou nad tuto základní úroveň a pokusí se čtenáři vysvětlit také něco o architektuře programů a zásadách jejího návrhu. To ponechávají učebnicím pro pokročilé kurzy, případně se domnívají, že se to čtenář při tvorbě svých programů přirozeně naučí jako vedlejší efekt. Většina čtenářů však již žádnou učebnici pokročilého programování číst nebude a bude odkázána na samostudium při návrhu svých programů, při němž se poučuje z vlastních chyb a často pracně odhaluje zákonitosti, které je měli učitelé naučit již v počátku výuky.

Vezměme si ponaučení z historie a podívejme se, kolik úsilí stálo a jak dlouho trvalo, než ti nejlepší programátoři objevili zásady, které se pak následně začaly obje-

vovat v učebnicích pro pokročilé. Přitom ale většina těchto zásad patří právě do učebnic pro začátečníky a často navíc do jejich úvodních lekcí.

Učebnice, které nechtějí zůstat u pouhého výkladu kódování, často vysvětlují, že velké projekty bývá většinou nejvýhodnější navrhovat metodou shora dolů, při níž definujeme základní cíl a dekomponujeme jej na jednotlivé podcíle. Na každý z podcílů použijeme rekurzivně totéž pravidlo, až se po několika krocích dostaneme do stavu, kdy jsou podcíle tak jednoduché, že není problém je naprogramovat.

To je však pro mnohé čtenáře pouhá teorie, protože jim jejich učebnice celou dobu předváděla postup právě opačný: jak z jednoduchých komponent stavět větší celky. K zažití obráceného postupu by potřebovali mnohem více příkladů a hlavně mnohem více času. To jim však učebnice většinou nenabídnou, protože pokud tyto zásady vysvětlují, tak povětšinou v rámci závěrečných shrnutí.

Problematické zvládnutí nových poznatků

Nevhodnost výše popsaného postupu je navíc umocněna tím, že jakmile opustíme pubertu, ztratíme schopnost přijmout nové informace, aniž bychom se je podvědomě snažili ihned sladit s tím, co už víme. Pokud přijatá informace neodpovídá zcela našim znalostem, podvědomě si ji upravíme tak, aby se s nimi dostala do souladu.

Tato skutečnost ale často brzdí zvládnutí vyšších hladin abstrakce, pokud si před tím studenti důkladně osvojili hladiny nižší. Jakmile jim jejich dosavadní znalosti a zkušenosti naznačí jakousi podobnost nově přijímaného poznatku s poznatky dříve osvojenými, okamžitě podvědomě nový poznatek upraví tak, aby jej mohli bez problému interpretovat prostřednictvím dosavadních poznatků. Neuvědomují si přitom, že si vlastně zapamatovali poněkud jinou informaci, než jakou se jim snažil vyučující předat.

Toto podvědomé upravování přijímaných fakt je tím intenzivnější, čím větší zkušenosti student má. Začínajícího programátora tak některým dovednostem naučíme výrazně snáze než starého praktika. Přechod zkušených programátorů na nové paradigma je proto velmi často pouze formální: programátoři sice začnou programovat v novém programovacím jazyku, ale jejich architektonické myšlení pokračuje stále ve starých kolejích. Výstižně to vystihl článek [2], který tvrdí, že „*Real Programmer can write FORTRAN programs in any language*“.

3 Jak stavět učebnice a kurzy programování

V [1] najdeme 14 pedagogických zásad, kterými bychom se měli při návrhu našich učebnic a kurzů řídit. První uvedenou (a jednou z nejdůležitějších) je zásada ranního ptáčete: „*Organize the course so that the most important topics are taught first. Teach the most important material, the “big ideas”, first (and often). When this seems impossible, teach the most important material as early as possible.*“

Chceme-li tedy své studenty (a čtenáře) naučit uvažovat o programu v abstraktních pojmech bez toho, že by se nechávali strhávat úvahami o některých podružných detailech, musíme je tomu učit od samého začátku, a ne s tím přijít někdy ke konci učebnice či kurzu.

Kdykoliv si všimneme, že by mohlo být pochopení některého důležitého principu nepříznivě ovlivněno předchozími zkušenostmi, měli bychom uspořádat výklad tak, abychom studenty (čtenáře) nejprve naučili tento důležitý princip, a teprve pak vysvětlovali rysy, jejichž předčasná znalost by mohla ovlivnit přijetí onoho důležitého principu.

Ilustrujme si to na dvou příkladech.

Příklad 1: Návrhový vzor Stav

Budeme-li chtít studentům ukázat, že liší-li se chování objektu v závislosti na jeho stavu, je v řadě případů výhodnější definovat pro každý stav samostatnou třídu a v ní definovat chování objektu nacházejícího se v daném stavu. Chování celého vícestavového objektu pak popisuje třída definovaná podle návrhového vzoru *Stav* a spolupracující s výše popsanými jednostavovými třídami.

Budeme-li studenty seznamovat s tímto návrhovým vzorem až poté, co už budou suverénně používat podmíněné příkazy a přepínače, bude jim zažité klasické, neobjektové řešení připadat jednodušší a přirozenější. Budou se mu proto snažit dávat přednost i v případech, kdy by použití vzoru *Stav* bylo výrazně výhodnější.

Seznámíme-li je s tímto vzorem ještě před tím, než jim vysvětlíme podmíněné příkazy a přepínače, bude to pro ně nějakou dobu jediné řešení, jak realizovat potřebné rozhodování. Až se později naučí používat podmíněné příkazy, budou už mít návrhový vzor *Stav* zažitý a nebude jim připadat tak extravagantní. Budou již vědět, jaké jsou jeho výhody a budou se proto rozhodovat mnohem lépe.

Příklad 2: Dědičnost

Obdobná situace je i v případě dědičnosti. Příliš časný výklad dědičnosti do jisté míry zablokuje pozdější akceptaci alternativních řešení některých problémů. Výhodnější je seznámit studenty nejprve s návrhovým vzorem *Dekorátor* a předvést jim jeho použití na příkladech, v nichž je jeho použití výrazně výhodnější než použití klasické dědičnosti. Když se později seznámí s dědičností, budou už mít za sebou několik projektů výhodně využívajících dekorátor a budou daleko méně náchylní k používání dědičnosti v situacích, kdy její použití není optimální.

Navíc výklad dědičnosti založený na znalosti návrhového vzoru *Dekorátor* je pro studenty většinou výrazně pochopitelnější, protože jsou základní vlastnosti dědičnosti vysvětleny s použitím konstrukcí, které už znají.

Časný výklad architektury

Zásadu ranního ptáčete bychom mohli aplikovat i v obecnější rovině. Sledujeme-li vývoj nástrojů pro vývoj programů, nemůžeme si nevšimnout, jak neustále přibývá nástrojů, které umějí zakódovat nejrůznější obraty. Přicházejí programovací jazyky se stále mocnějšími příkazy, které realizují funkce, jejichž kód donedávna zabíral téměř celou stránku výpisu.

Při uvažování této skutečnosti bychom se měli snažit neprotěžovat zbytečně výuku dovedností, od jejichž aplikace moderní nástroje své uživatele postupně osvobozují. Měli bychom se soustředit spíše na vštípení zásad, které studenti uplatní při návrhu

architektury. Tyto zásady bychom jim přitom neměli vštěpovat až v nějakých nadstavbových kurzech. Jejich výklad bychom měli zařazovat od samého počátku našich kurzů. Jinými slovy: kurzy bychom měli koncipovat tak, abychom mohli tyto zásady vykládat co nejdříve.

Z toho ovšem vyplývá, že by výklad neměl být postaven na jednoduchých AHA příkladech, ale že bychom měli se studenty od počátku pracovat na nějakém rozsáhlejším projektu, k němuž budeme přidávat jednoduché části případně některé z jeho částí upravovat. Tak se studenti současně naučí pracovat v režimu, který je mnohem bližší tomu, s nímž se setkají ve své pozdější praxi.

5. Doprovodné programy

Zvláštní kapitolou jsou doprovodné programy. Řada učebnic a kurzů se omezuje na jednoduché AHA programy, z nichž sice studenti pochopí základní funkci vysvětlované konstrukce, ale v řadě případů nezískají rozumnou představu o tom, jak tyto konstrukce správně použít v programu. Takovéto programy se proto hodí pro výuku studentů, kteří se potřebují pouze naučit pracovat v novém programovacím jazyce, ale programovat již umějí a probírané paradigma znají, takže si použití probíraných konstrukcí v praxi umějí odvodit a domyslet.

Jiné učebnice naopak preferují výuku na relativně složitých praktických programech. Studenti tak sice uvidí zasazení probírané látky do širšího kontextu demonstračního programu, ale pochopení okolního programu je někdy stojí zbytečně velké úsilí. Takovýto druh doprovodných programů se proto hodí pro studenty, kteří znají základy programování a potřebují se naučit pracovat s novými knihovnami a frameworky.

Stále nám však zbývají studenti, kteří do světa programování teprve vstupují. Ti potřebují demonstrační programy dostatečně jednoduché, aby se nemuseli k vykládané konstrukci probojovávat záplavou okolního „šumu“, ale na druhou stranu dostatečně složité, aby viděli, jak danou konstrukci použít v praxi. Při návrhu doprovodných programů pro tyto studenty se můžeme vydat dvěma směry:

Samostatné programy pro každou konstrukci

Z pohledu autora je nejjednodušší demonstrovat pro každou konstrukci její použití v samostatném programu. Problém je, že doprovodné programy budou zejména v počátečních kapitolách opravdu velmi jednoduché, takže to budou spíše AHA-příklady. Protože se však jedná o programy z počátečních kapitol, bude mít student v dalších programech určitě dostatek příležitostí se s probíranou konstrukcí setkat a ujasnit si její možná použití.

Postupně vylepšované programy

Z hlediska studenta je výhodnější, když je látka demonstrována na menším množství postupně vylepšovaných programů. Při studiu použití nové konstrukce pak student

nemusí analyzovat okolní program tak podrobně, protože jej nejspíše zná z předchozích lekcí. Může proto pracovat i s relativně složitými programy, aniž by jej tak výrazně rušil „šum“ okolního kódu.

Druhý přístup je výhodnější, protože umožňuje navíc vedle probíraných konstrukcí průběžně učit zásadám, které jsou důležité při tvorbě a následné údržbě reálných projektů: přehlednosti kódu, opakovanému použití dříve napsaných automatických testů, používání konstrukcí usnadňujícím následné modifikace a údržbu a řadě dalších zásad.

Nepříjemnou vlastností tohoto přístupu je, že bychom měli mít pro každou součást programu tolik zdrojových souborů, kolik je jednotlivých podob dané součásti v průběhu vývoje celého projektu a neustále ověřovat jejich vzájemnou konzistenci. Pokud bychom si navíc v některé pozdější fázi vývoje daného doprovodného programu uvědomili, že jsme měli v některé z předchozích lekcí vysvětlit něco jinak, museli bychom vhodně modifikovat větší množství zdrojových souborů a stále přitom dbát na jejich vzájemnou konzistenci a také na to, abychom v některém z nich nepoužili konstrukci, která v dané lekci ještě nebyla probraná.

Tento problém lze naštěstí řešit programově. Příkladem programu, který nám může s touto úlohou pomoci, je program *Cumulant*, jehož účelem je právě podpořit definici projektů, které se postupně vylepšují a kumulují tak další a další funkce. Program definuje speciální preprocesorové komentářové příklady, které nám umožní definovat v jednom souboru více verzí zdrojových souborů pro jednotlivé verze projektu. Současně můžeme ve zvláštním souboru definovat, jak se ze základní sady zdrojových souborů mají vybírat soubory, z nichž budou (po příslušné konverzi) sestavovány projekty definující jednotlivé fáze postupně budovaného programu.

6. Závěr

Příspěvek analyzoval různé přístupy k psaní učebnic a k návrhu osnov kurzů programování a programovacích jazyků. Rozdělil učebnice a kurzy na ty, které se soustředí pouze na výuku jazyka a jeho knihoven, a na ty, které se opravdu snaží naučit čtenáře a kurzanty programovat. V dalším textu se pak soustředil na učebnice a kurzy snažící se učit programování.

Pro tuto skupinu učebnic a kurzů specifikoval některé zásady, které je při návrhu jejich obsahu vhodné respektovat. Doporučil dodržovat pedagogickou zásadu ranního ptáčete a uspořádat výklad tak, aby se nejdůležitější věci učily opravdu co nejdříve. Současně doporučil nesoustředit se zbytečně na prvky, jejichž tvorbu postupně přebírají různé automatizované systémy, a naopak učit od samého začátku také zásady architektury.

V závěru seznámil s různými přístupy k tvorbě doprovodných programů a rozebral jejich výhody a nevýhody. Současně upozornil na program, jenž může výrazně pomoci při návrhu doprovodných programů, které studenti v průběhu kurzu postupně vyvíjejí a zdokonalují.

Poděkování

Tento příspěvek vznikl za podpory grantu *Výzkum metodik výuky programování a možností jejich zlepšení* vypsaného nadací RPF.

Literatura

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

[2] POST Ed: *Real Programmers Don't Use Pascal*, Datamation 1983, For downoad at http://www.ee.ryerson.ca/~elf/hack/realmen.html.

[3] PECINOVSKÝ Rudolf: *Cumulant – Assistant supporting creation of teaching project stepwise cumulating functionality of the developed program.* Proceeding of Objects 2011. ISBN 978-80-554-0432-5.