METHODOLOGY *ARCHITECTURE FIRST*

Rudolf PECINOVSKÝ

Abstract: The set of tasks solved by computers increases all the time. There are also programming tasks among them. The area, which still resists to automation, is a design of a good architecture. However, most of the current methodologies do not reflect this trend and teach primarily how to write a program in some programming language. The methodology *Architecture First* turns it up and starts with teaching the architecture. The paper introduces this methodology and explains its basic principles.

Key words: architecture, Architecture First, Design Patterns First, education, methodology, OOP, programming

METODIKA *ARCHITECTURE FIRST*

Resumé: Množina úloh, které již není třeba programovat, protože je umí naprogramovat nějaký generátor kódu, se stále rozšiřuje. Oblastí, která automatizaci stále vzdoruje a ještě nějakou dobu jí vzdorovat bude, je návrh architektury programu. Většina používaných metodik výuky však stále klade důraz především na výuku kódování. Příspěvek seznamuje se základními principy metodiky výuky *Architecture First*, která ukazuje, jak vychovávat programátory, které automaty ještě dlouho nenahradí.

Klíčová slova: architektura, metodika *Architecture First*, metodika *Design Patterns First*, metodika výuky, OOP, výuka programování

1 Úvod

Oblast úloh, které programátor nemusí řešit, protože je za něj umí vyřešit nějaký hotový program, se stále rozšiřuje. Nejprve jsme přešli od strojového kódu k vyšším programovacím jazykům, pak jsme začali využívat rozsáhlé knihovny, v současné době se stále častěji uplatňují nejrůznější generátory kódu. Oblastí, která však automatizaci stále vzdoruje, je návrh kvalitní architektury.

S obdobným problémem se setkáváme i při rozhovorech se zástupci firem. Ti si stěžují, že většinu absolventů škol může zaměstnat tak nejvýše jako kodéry a že jejich týmům chybějí dobří architekti.

Řada autorů již ve svých pracech ukázala, že styl programování, který se studenti naučí jako první, na dlouhou dobu ovlivňuje jejich práci a způsob návrhu programů ([5], [9], [12], [18]). To ale neplatí pouze o změně paradigmatu, např. při přechodu ze strukturovaného programování na programování objektově orientované.

Stejný efekt pozorujeme i ve chvíli, když studenty, kteří už umějí vytvořit středně složité programy, začneme učit, jak navrhovat architekturu těchto programů. Když takovýto student dostane nějaké zadání, většinou začne poměrně záhy přemýšlet nad tím, jak by tu či onu funkcionalitu zakódoval, a architekturu celého projektu podřizuje způsobu jeho zakódování do použitého programovacího jazyka.

Nabízí se proto myšlenka začít respektovat pedagogickou zásadu ranního ptáčete a upravit výuku programování tak, abychom studenty začali nejdříve učit to nejdůležitější – návrh architektury. Pro zakódování navrženého programu pak využít nějaký generátor kódu a zařadit výuku kódování (tj. zápisu programů v nějakém programovacím jazyce) až ve chvíli, kdy složitost našich programů překročí možnosti dostupných generátorů kódu.

2 Současný stav

Většina používaných metodik začíná výuku programování výukou zápisu programu v kódu nějakého programovacího jazyka. Metodika *Object First* sice přišla s myšlenkou začít výuku vysvětlením základního chování objektů v diagramu tříd ([1], [8]), avšak její autoři si bohužel neuvědomili genialitu a dosah této myšlenky umožňující pokračovat ve výkladu architektury, a ve svých učebnicích velice rychle sklouzávají ke klasickému přístupu.

Autoři této metodiky vyvinuli vývojové prostředí *BlueJ*, které je vybaveno jednoduchým generátorem kódu, a umožňuje proto zůstat se studenty delší dobu v hladině architektury a definici jednodušších programů nechat na bedrech používaného vývojového prostředí ([1], [10]). Bohužel, potenciál tohoto prostředí ve svých učebni-

cích prakticky nevyužívají a po základním architektonickém úvodu sklouzávají do úrovně kódu.

3 Základní principy metodiky *Architecture First*

Opakované stížnosti podniků na znalosti standardních absolventů škol vedly autora příspěvku k tomu, že vyvinul metodiku výuky *Architecture First* [1], která zavedené postupy obrací. Metodika se řídí pedagogickou zásadou ranního ptáčete (early bird pattern), která říká ([3], [4]): *"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."*

Rozhodneme-li se učit především tvorbu architektury, protože víme, že následné kódování přebírá na svá bedra z větší a větší míry počítač, měli bychom základní architektonické zásady učit co nejdříve, a principy kódování učit až následně jako jeden ze způsobů realizace navržené architektury.

Metodika *Architecture First* proto nejprve se studenty probírá základní principy budování architektury objektově orientovaných programů, a teprve poté, co tyto principy studenti vstřebají, pokračuje výkladem způsobu, jak lze navržený program zakódovat.

Někteří z vás možná namítnou, že takto postupuje řada kurzů. Hlavní rozdíl je ale v tom, že většina kurzů zůstává při svém výkladu architektury v teoretické rovině, kdežto metodika *Architecture First* využívá při počátečním výkladu interaktivních schopností použitého vývojového nástroje, a zejména pak jeho schopnosti vytvořit program realizující předvedenou činnost. Studenti tak mají možnost jednodušší návrhy ihned realizovat, aniž by se museli rozptylovat pravidly syntaxe použitého programovacího jazyka.

Výuka v úvodních kurzech programování aplikujících tuto metodiku proto probíhá ve čtyřech etapách ([14]):

1. První etapa probíhá v interaktivním režimu, kdy veškerý kód vytváří generátor, jenž je součástí použitého vývojového prostředí.
2. Druhá etapa přechází do textového režimu, v němž si studenti opakují látku první etapy a učí se zapsat programy, které v první etapě vytvářel zmíněný generátor.
3. Ve třetí etapě se pak studenti seznamují s náročnějšími konstrukcemi, které jsou za hranicemi schopností použitého generátoru.
4. Ve čtvrté etapě se studenti seznámí se základními algoritmickými konstrukcemi a naučí se je používat ve svých programech.

4 První etapa

Jak bylo řečeno, v první etapě pracujeme se studenty v interaktivním režimu, v němž student vystupuje jako jeden z objektů programu. Tento „objekt" posílá ostatním objektům (včetně objektu vývojového prostředí) zprávy. Prostřednictvím zasílání zpráv student aktivuje ostatní objekty, sdělí jim, co mají dělat, a ukáže tak vývojovému prostředí, jak se má navržený program chovat. Vývojové prostředí pak na požádání vytvoří program (definuje metodu), který předvedenou činnost zopakuje. Student vlastně pracuje v podobném režimu, v jakém se v některých programech vytvářejí jednoduchá makra.

Protože se studenti v první etapě vůbec nezabývají zakódováním navrženého programu, nejsou rozptylování syntaktickými pravidly použitého jazyka a mohou se soustředit především na vysvětlovanou architekturu a probírané architektonické principy. To nám umožnuje již na začátku výuky vysvětlit a názorně předvést, jak fungují takové základní konstrukce, jakými jsou skládání objektů, implementace rozhraní, dědění rozhraní i dědění implementace, potřeba zavedení abstraktních tříd a jejich základní vlastnosti a některé další konstrukce.

5 Co v první etapě vykládáme

První etapu začínáme výkladem všeobecné povahy objektů, při němž studentům vysvětlíme, že v OOP je objektem vše, co můžeme nazvat podstatným jménem. V důsledku toho zařadíme mezi objekty nejenom takové abstraktní pojmy, jako spojení, přerušení, výpočet, barva, směr či krása, ale současně si vysvětlíme, že objektem je i třída, přestože to řada učebnic popírá. V tom nám pomáhá vývojové prostředí, v němž můžeme předvést, že s objektem třídy se v interaktivním režimu pracuje naprosto stejně, jako s objekty jejích instancí.

Nosným tématem první etapy je práce s rozhraními a interfejsy, která je základem moderního programování, a přitom s nimi značná část absolventů škol pracovat neumí. Umějí implementovat definovaný interfejs, ale neumějí správně odhad-

[1] Metodika byla původně zveřejněna pod názvem *Design Patterns First* [17], ale v průběhu dalšího vývoje si autor uvědomil, že časný výklad návrhových vzorů je pouze důsledkem celkového zaměření na časný výklad architektury, a proto metodiku přejmenoval.

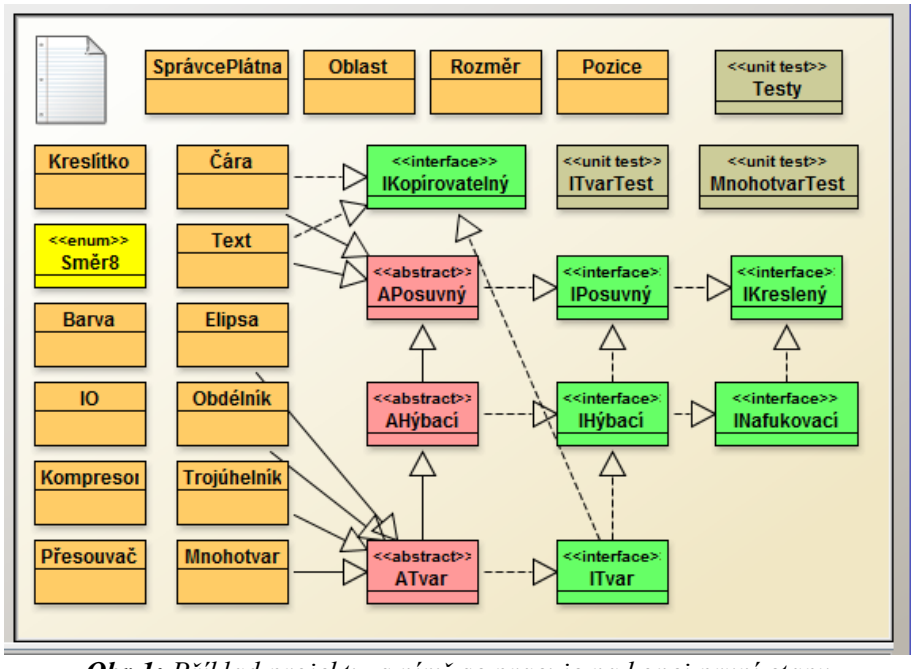
nout, kdy by měli v projektu definovat interfejs vlastní. Navíc si často neuvědomují, že rozhraní je daleko obecnější pojem než interfejs. První etapa se jim proto snaží přiblížit skutečný význam a použití obou pojmů v současných programech.

Jakmile studenti pochopí význam pojmů rozhraní a interfejs, je logickým pokračováním výklad návrhových vzorů ([15]), a to jak těch, které s interfejsem nepracují (*Knihovní třída*, *Jedináček*, *Prázdný objekt*, *Přepravka*, *Výčtový typ)*, tak samozřejmě i těch, v nichž je použití interfejsu (případně jeho náhražky v jazycích, které interfejs nezavádějí) jejich základním stavebním kamenem (*Služebník*, *Prostředník*, *Pozorovatel/Posluchač/Vydavatel-Předplatitel*).

Během této interaktivní etapy můžeme vysvětlit a předvést dokonce i některé architektonické principy (programování proti rozhraní, minimalizace provázanosti, inverze řízení). Studenti se zde naučí pracovat s rozhraním, uvědomí si jeho primární účel (degradovaný v řadě učebnic na náhražku násobné dědičnosti). Naučí se vnímat jeho funkci v rámci architektury projektu.

Na konci etapy vysvětlujeme rozdíly mezi třemi typy dědění a na výklad dědění typů, který jsme jako jediný doposud používali, navážeme výkladem dědění implementace zahrnujícím i výklad abstraktních tříd a jejich opodstatnění v programu.

Na konci první etapy pracujeme s projektem, jehož složitost řádově odpovídá složitosti projektu na obrázku 1.



Obr 1: *Příklad projektu, s nímž se pracuje na konci první etapy*

6 Práce v interaktivním režimu

Práce v interaktivním režimu, při něm jakoby ignorujeme nutnost zakódování navržených konstrukcí, má své výhody:

- Studenti začátečníci se nerozptylují syntaktickými pravidly použitého jazyka a mohou se soustředit na vysvětlované architektonické konstrukce.
- Pokročilí studenti mají výrazně ztíženou možnost překlopit své uvažování do hladiny kódu a musejí uvažovat v architektonických termínech.

Je zajímavé pozorovat, jaký vliv má na pokročilé studenty nutnosti používat generátor kódu. Na počátku je jim totiž jasné, jak by tu či onu konstrukci zapsali a snaží se proto generátor kódu obcházet. Cokoliv, co vyučující řekne, si ihned přeloží do kódu a jistou dobu proto jedou na jakési paralelní rovině, protože jejich dosavadní zkušenost jim občas vnucuje modifikovat obdrženou informaci, takže si pak zapamatují něco jiného, než co jim vyučující říkal [5].

S pokračujícím výkladem se však studenti postupně učí vnímat program jinýma očima a uvědomují si, že skutečně objektově orientovaný pohled na zpracovávaný program je přece jenom poněkud odlišný od toho, na jaký byli ze své dosavadní praxe zvyklí.

Převod studentů zvyklých na klasické strukturované programování (byť v objektově orientovaném jazyce) bychom mohli přirovnat

k postupu, který na přelomu 70. a 80. let minulého století použil Richard E. Pattis, když svým studentům zvyklým z domova na programování v jazyku Basic, představil robota Karla s jeho omezenými možnostmi [11].

7 Kurzy pro manažery

Výklad základních architektonických principů s demonstracemi a praktickými ukázkami v interaktivním režimu je použitelný i pro manažery, kteří chtějí získat základní informace o tvorbě programů, aby je pak jejich či dodavatelovi programátoři nemohli, jak se lidově říká, „opít rohlíkem“ ([16]). S výhodou se používá i u programátorských týmů pracujících podle agilních metodik, které doporučují, aby se členem vývojového týmu stal zástupce zákazníka ([2]). Typický zástupce zákazníka se sice nechce programovat, ale na druhou stranu bývá ochoten komunikovat s týmem v prostředí diagramů objektů, případně diagramů tříd, a oponovat navrhované chování programu na úrovni, které je mnohem blíže kódu, než tomu bývá bez použití interaktivního režimu.

8 Druhá etapa

V programátorských kurzech přecházíme po vysvětlení základních architektonických konstrukcí do druhé etapy výuky, v níž studenti znovu procházejí předchozí látku a učí se zakódovat programy, které za ně v první etapě kódoval generátor v použitém vývojovém prostředí. V této etapě se studenti učí syntaktická pravidla a základní programové konstrukce použitého jazyka, aniž by se přitom museli rozptylovat návrhem požadovaného programu. Program již mají navržen z první etapy a nyní si pravidla jeho návrhu pouze opakují.

Toto opakování je velice potřebné, protože studenti si v první etapě často správně nezapamatují přesný význam některých vysvětlovaných pojmů. Tento problém se projeví zejména u studentů s předchozími programátorskými zkušenostmi, kteří si (jak již bylo výše zmíněno) zapamatují řadu předávaných informací zkresleně, protože daná informace neodpovídá přesně jejich předchozím zkušenostem získaným při práci v jiném paradigmatu, a jejich mozek ji proto podvědomě přizpůsobí dosavadním zkušenostem [5].

Základním pravidlem druhé etapy ale je, aby studenti při práci s kódem nezapomněli na architektonický přístup k řešení problému a neutopili se v kódu, jak to bývá u většiny programátorů obvyklé. Důležité je, aby studenti nadále dokázali program nejprve navrhnout v architektonické hladině a teprve následně přejít k jeho kódování.

V druhé etapě se ale neproberou všechny konstrukce z první etapy. V této etapě ještě nevysvětlujeme dědění implementace, protože jeho příliš časný výklad svádí studenty k zneužívání této konstrukce a po výkladu dědění implementace si studenti navíc obtížně osvojují programové konstrukce, které by mohly dědění implementace nahradit.

Druhá etapa proto končí výkladem práce s balíčky (jmennými prostory), po němž je vhodně přejít na profesionální vývojové prostředí.

Postupy i metodiky výuky použité v první a druhé etapě jsou důsledně aplikovány v učebnici [14].

9 Třetí etapa

Jak bylo naznačeno, ve třetí etapě opouštíme výukové prostředí a přecházíme na profesionální prostředí, které nám umožňuje s přiměřenou námahou navrhovat rozsáhlejší projekty. Navíc se studenti seznámí s některým z prostředí, která se v praxi opravdu používají.

V této etapě se dále prohloubí výklad architektonických principů, který je nyní už prokládán výkladem konkrétní realizace potřebných programových konstrukcí. Pokračuje se dalšími návrhovými vzory, především pak těmi, které nabízejí alternativní řešení úloh, jež bývají většinou řešeny jinak, i když toto „jiné“ řešení není optimální. Mezi těmito vzory vyčnívají především návrhové vzory *Stav*, *Adaptér* a *Dekorátor*.

Návrhový vzor *Stav* je třeba studentům představit dříve, než se seznámí s podmíněným příkazem. Pokud se totiž studenti naučí nejprve pracovat s podmíněným příkazem, odmítají používat návrhový *Stav* i v situacích, kdy je jeho použití mnohem vhodnější. Pokud obrátíme pořadí výuky, umějí daleko lépe odhadnout, který postup je v dané situaci vhodnější.

Návrhový vzor *Dekorátor* je pro změnu vhodné vysvětlit před tím, než se seznámí s děděním implementace. Včasné vysvětlení tohoto vzoru má dvě výhody:

- Znalost návrhového vzoru *Dekorátor* výrazně napomáhá pochopení různých vlastností dědění implementace, protože jim lze vždy ukázat ekvivalentní chování odpovídajícího dekorátoru a studenti pak mnohem lépe pochopí, proč se vysvětlovaná konstrukce chová právě takto.
- Pokud si před probíráním dědičnosti studenti tento vzor dostatečně osvojí, jsou pro ně pak dědičnost a dekorace dvě více méně rovnocenné varianty řešení a mohou se mezi nimi

kvalitněji rozhodnout. Při obráceném postupu výkladu studenti častěji volí dědičnost i v situacích kdy její použití není vhodné.

Po probrání výše uvedených vzorů můžeme přistoupit k vysvětlení dědění implementace se všemi jeho vlastnostmi a záludnostmi. Takto pozdě zařazený výklad dědění implementace umožní, aby studenti dostatečně zažili, jak lze různé problémy řešit bez něj, a přijali dědění implementace jako další užitečnou konstrukci a ne jako klíčovou konstrukci, které se vše přizpůsobuje.

Navíc nám dříve vysvětlený vzor *Dekorátor* umožní emulovat chování dědění implementace za pomoci dříve vysvětlených a osvojených konstrukcí, takže pak studenti nepokládají některé rysy dědění za nějakou skrytou magii.

Díky tomuto přístupu můžeme studentům názorně vysvětlit a na příkladech názorně předvést i některé nestandardní rysy a vlastnosti dědění implementace, jejichž probrání se standardně koncipované učebnice většinou vyhýbají.

Návrhový vzor *Adaptér* nám po probrání dědění implementace umožní studentům předvést možné řešení různých úloh, které se v praxi relativně často vyskytují, ale učebnice se jim moc nevěnují – např. jak lze v projektu definovat dvě relativně nezávislé hierarchie dědičnosti.

Vedle dědění implementace se ve třetí etapě vysvětlují i lambda-výrazy, které se nacházejí na pomezí mezi algoritmickými a architektonickými konstrukcemi. Po seznámení s lambda-výrazy se studenti seznámí se základními kolekcemi a naučí se s nimi pracovat s využitím interních iterátorů a lambda výrazů.

I ve třetí etapě se, stejně jako ve druhé, snažíme, aby se studenti soustředili především na architekturu navrhovaných aplikací. Příklady řešené v této etapě nevyžadují návrh nějakých složitých algoritmů, ale řeší se většinou především v architektonické rovině. Jedinou algoritmickou konstrukcí, která se v prvních třech etapách výuky používá, je posloupnost příkazů. Výklad zbylých algoritmických konstrukcí je součástí čtvrté etapy.

Rozdíl mezi klasickým a zde probíraným přístupem je patrný zejména při porovnání programů navržených při výuce podle metodiky *Architecture First* s programy, které řeší stejný problém podle klasických postupů, a které jsou proto v řadě případů složitější a obtížněji modifikovatelné.

10 Čtvrtá etapa

Jak už bylo naznačeno, až do této chvíle se studenti nesetkali s algoritmickými konstrukcemi, jakými jsou podmíněné příkazy, cykly a rekurze. Při řešení dosavadních úloh je totiž nepotřebovali, přestože to v řadě případů byly úlohy netriviální.

Ve čtvrté etapě se začneme trochu více soustředit na kód a tyto základní algoritmické konstrukce si vysvětlíme. Studenti se postupně seznámí s alternativní možností řešení dosavadních úloh prostřednictvím algoritmických konstrukcí. Protože už mají přechozí, „architektonické“ řešení již (alespoň částečně) osvojené, mohou se v praxi kvalifikovaně rozhodnout, které řešení je v té které situaci výhodnější.

Výhodou přístupu metodiky *Architecture First* je to, že při obráceném postupu, tj. při prvotním seznámení s kódem a algoritmickými konstrukcemi a teprve následném seznamování a architektonickou alternativou řešení různých úloh, nepovažují studenti obě alternativy za rovnocenné. Jakmile jednou zvládnou algoritmické konstrukce, budou jim architektonické alternativy připadat ve školních příkladech zbytečně složité, a nesáhnou proto po nich ani v příkladech, kdy je jejich volba výhodná.

Postupy i metodiky výuky použité v třetí a čtvrté etapě jsou důsledně aplikovány v učebnici [13].

11 Grafické zobrazení algoritmů

Ve čtvrté etapě se setkáme i s potřebou použít poněkud složitější algoritmy než ty, s nimiž se studenti setkali v průběhu první až třetí etapy, v nichž se výklad soustředil především na architekturu navrhovaných programů.

Většina současných učebnic programování se soustřeďuje především na výklad syntaxe jazyka a používání dostupných knihoven. Algoritmické dovednosti programátorů přitom nijak nerozvíjí.

Přiznejme si, že v současné době, v níž se stále množí nejrůznější frameworky a generátory kódu, začíná programování připomínat práci se stavebnicí, při níž je důležité pouze umět vhodně propojit jednotlivé prvky. Jednou za čas se však i řadový programátor setká s poněkud složitějším algoritmem, který musí pochopit. K tomu je vhodné použít nějaký grafický nástroj.

Dříve se algoritmy zakreslovaly prostřednictvím vývojových diagramů, které ale sváděly programátory k používání nestrukturovaných konstrukcí a dalších programátorských obratů, jež v současné době považujeme za nečisté.

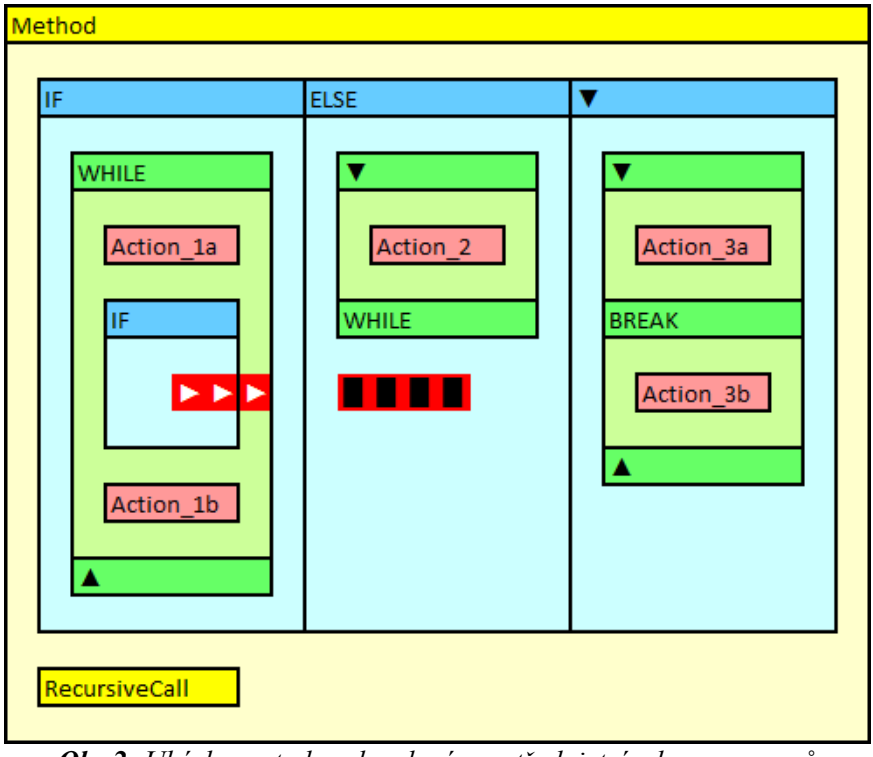
Později je proto začaly (alespoň na některých univerzitách a v některých firmách) nahrazovat Nassi-Schneidermanovy diagramy. Ty sice bránily používání konstrukcí, které jsou v rozporu se zásadami strukturovaného programování, ale zase zaváděly používání šikmých čar, které poněkud komplikovaly zápis výrazů, podle nichž se rozhoduje.

V současné době se většinou dává přednost používání UML diagramů v nichž se k zápisu algoritmů používají diagramy aktivit, které jsou odvozeny z původních vývojových diagramů se všemi jejich nectnostmi.

Metodika *Architecture First* doporučuje používat pro zobrazení složitějších algoritmů kopenogramy ([7] – viz obrázek 2). Algoritmy se sice dále zapisují v kódu použitého jazyka, ale vývojové prostředí je vybaveno zásuvným modulem (pluginem), který na požádání zobrazí kopenogram označené metody ([6]). Výhodou kopenogramů oproti vývojovým diagramům a diagramům aktivit je jejich jednoznačná preference strukturovaných konstrukcí. Kopenogramy sice umožňují v případě potřeby zobrazit i nestrukturované konstrukce, ale porušení zásad strukturovaného programování je v nich jasné a zřetelné.

Výhodou kopenogramů oproti Nassi-Schneidermanovým diagramům je absence šikmých čar a jednoznačný, předem daný význam jednotlivých barev, který výrazně zvyšuje vypovídací schopnost diagramů i při pohledu ze vzdálenosti, z níž ještě není možné přečíst obsah jednotlivých bloků.

Zobrazení kopenogramu zadané metody využijeme především při vysvětlování složitějších algoritmických konstrukcí a při hledání chyb ve složitějších metodách.



Obr 2: *Ukázka metody zakreslené prostřednictvím kopenogramů*

12 Úlohy, které ještě čekají na řešení

V diskusi k první etapě výuky byly zmíněny problémy, které mají zkušenější studenti se změnou své orientace na nové paradigma. S ještě intenzivnějšími problémy se setkáváme u profesionálních programátorů, kteří přecházejí do přeškolovacích kurzů objektově orientovaného programování. Bohužel se ukazuje, že obdobné problémy mají i studenti, kteří sice prošli kurzy objektově orientovaného programování, ale nenaučili se objektově myslet, takže je firmy posílají do našich přeškolovacích kurzů.

Obě tyto skupiny vytvářejí typické strukturované programy v objektově orientovaných jazycích s občasným, často i formálním, použitím objektových konstrukcí. Když je to potřeba, umějí implementovat interfejs, ale mají velký problém odhadnout, ve kterých situacích by bylo výhodné navrhnout interfejs vlastní.

Jako účastníci přeškolovacích kurzů považují tito programátoři počáteční práci v interaktivním

režimu za zbytečné zdržování. Následně pak ale přiznávají, že si až dodatečně uvědomili, jak moc se toho v této etapě naučili.

Dalším společným rysem značné části současných programátorů je to, že nedokáží přemýšlet v hladině architektury. Typický programátor dostane zadání a jeho mozek okamžitě přepne do hladiny kódu a on začne přemýšlet, jak by zadání zakódoval. Jejich architektonické myšlení je proto svázáno s jejich schopností kódovat a ve svých programech proto používají pouze takové konstrukce, o nichž dopředu vědí, jak je zakódovat. Sémantická mezera, kterou se snaží objektově orientované programování odstranit nebo alespoň výrazně zmenšit, je u těchto programátorů většinou zachována v původní, „předobjektové" velikosti.

Hlavním výzkumným úkolem budoucího vývoje metodiky *Architecture First* je proto nalezení způsobů, jak co nejefektivněji přeškolit na objektové paradigma ty, kteří již mají s programováním své zkušenosti a domnívají se, že OOP je pouze nějaký nový název pro to, co oni už dávno znají.

13 Závěr

Článek nastínil některé problémy, s nimiž se potýká současná výuka programování, a seznámil se základními charakteristikami metodiky *Architecture First*, která se snaží těmto problémům předcházet a vychovávat programátory, kteří budou mít co nejlepší uplatnění v budoucí praxi.

Metodika rozděluje výuku programování do čtyř etap, přičemž první etapa je užitečná i pro ty, kteří nechtějí pracovat jako programátoři – např. manažeři objednávající softwarová řešení nebo zákazníci, kteří se podle agilních metodik stávají členy programátorských týmů.

První tři etapy výuky podle této metodiky se soustřeďují především na architektonické řešení problémů. V první etapě se vše řeší čistě v architektonické rovině a o vytvoření programu na základě navržené architektury se stará generátor kódu, který je součástí vývojového prostředí. Druhá etapa opakuje látku probranou v první etapě, avšak studenti se v ní učí zakódovat vše, co za ně v první etapě vytvářel generátor. Třetí etapa pokračuje výkladem architektonických konstrukcí, jejichž realizace byla za hranicemi možností použitého generátoru kódu. Čtvrtá etapa uzavírá výuku výkladem algoritmických konstrukcí, přičemž složitější algoritmy jsou zobrazovány prostřednictvím kopenogramů.

Metodika *Architecture First* je důsledně aplikována v učebnicích [14] (první a druhá etapa) a [13] (třetí a čtvrtá etapa).

14 Literatura

[1] BARNES D. J., KÖLLING M: *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall 2005. ISBN 0-13-124933-9.

[2] BECK K., ANDERS C.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley 2004. ISBN 0-321-27865-8.

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

[4] BERGIN, J.: *Pedagogical Patterns: Advice For Educators*. CreateSpace Independent Publishing Platform 2012. ISBN 1-4791-7182-4.

[5] DRIVER R., BELL, B.: *Students' thinking and learning of science: A constructivist view*. School Sci. Rev. 1986 pp. 443–456.

[6] FIALA M.: *Vytvořte editor kopenogramů.* Diplomová práce VŠE 2012.

[7] KOFRÁNEK J., PECINOVSKÝ R., NOVÁK P.: Kopenograms – Graphical Language for Structured Algorithms. *Proceedings of the 2012 International Conference on Foundation of Computer Science. WorldComp* 2012 Las Vegas. CSREA Press. ISBN 1-601-32211-9.

[8] KÖLLING, M., ROSENBERG, J.: Guidelines for Teaching Object Orientation with Java, *Proceedings of the 6th conference on Information Technology in Computer Science Education* (ITiCSE 2001), Canterbury, 2001.

[9] ENTWISTLE, N. 2007. Conceptions of learning and the experience of understanding: Thresholds, contextual influences, and knowledge objects. In *Reframing the Conceptual Change Approach in Learning and Instruction*. S. Vosniadou, A. Baltas, and X. Vamvakoussi Eds., Chapter 11, Elsevier, Amsterdam, The Netherlands.

[10] KÖLLING, M., *Teaching Object Orientation with the Blue Environment*, Journal of Object-Oriented Programming, Vol. 12 No. 2, 14-23, 1999.

[11] PATTIS R. E.: *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*. John Wiley & Sons, 1981.

[12] PEA, R. D.: *Language-independent conceptual "bugs" in novice programming*. J. Educ. Comput. Res. 2, 1. 1986.

[13] PECINOVSKÝ R.: *Java 8 – Učebnice objektové architektury pro mírně pokročilé*. Grada 2013.

[14] PECINOVSKÝ R.: *Java 7 – Učebnice objektové architektury pro začátečníky*. Grada 2012. ISBN 978-80-247-3665-5.

[15] PECINOVSKÝ R.: *Návrhové vzory – 33 vzorových postupů pro objektové programování*. Computer Press, © 2007, 528 s. ISBN 978 80 251 1582 4.

[16] PECINOVSKÝ R.: Using the methodology Design Patterns First by prototype testing with a user. *Proceedings of IMEM*, Spišská Kapitula.

[17] 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.

[18] SMITH III, J. P., DISESSA, A. A., ROSCHELLE. J.: *Misconceptions reconceived: A constructivist analysis of knowledge in transition.* J. Learn. Sci. 3 1993, 115–163.