Současné trendy v metodice výuky programování

RUDOLF PECINOVSKÝ
Vysoká škole ekonomická Praha, Fakulta informatiky a statistiky; Amaio Technologies, Inc;
Tel.: +420 603 330 090, e-mail: rudolf@pecinovsky.cz

Úvod

Programování doznalo v posledních 25 letech řadu naprosto zásadních změn. Měnily se nejenom používané programovací jazyky, ale měnila se i filozofie celého programování. Dávno již skončily doby, kdy programátoři vyvíjeli většinou samostatné programy a potřebovali se vedle syntaxe použitého jazyka naučit především dovednostem, jak správně algoritmizovat ten či onen problém. Nyní se musí místo toho naučit začleňovat své programy do existujících systémů, ovládat řadu nejrůznějších technologií a standardů a zvládat rozsáhlé knihovny.

Ať si to již připouštíme nebo ne, většina obtížných algoritmických problémů je nyní již vyřešena a jejich řešení jsou zařazena do běžně dostupných knihoven. Současné programování již dávno opustilo doby, kdy jsme je mohli zařazovat mezi umění, a přesunulo se do kategorie technologií. Základní dovedností, kterou musí současný programátor ovládat, není vymýšlení nových postupů, ale především správná aplikace postupů dříve vymyšlených.

Dosavadní historie vývoje nejrůznějších programovacích nástrojů a knihoven však přinesla řadu nových poznatků a zkušeností. Jedněmi z nejdůležitějších bylo docenění významu znovupoužitelnosti a snadné modifikovatelnosti dříve vytvořených programů. Znovupoužitelnost a snadná modifikovatelnost jsou v současné době klíčovými zaklínadly všech zkušených programátorů a manažerů softwarových projektů.

Optimální by bylo, kdyby se stejně jako vlastní programování vyvíjela i metodika jeho výuky. Bohužel, jak sami jistě víte, metodika výuky se ve většině oblastí za vývojem daného předmětu výrazně opožďuje. Sebekriticky si přiznejme, že mnozí učitelé často připravují žáky na styl programování, který možná byl před 20 lety progresivní, ale v současné době je již dávno překonaný. O to překonanější bude v době, kdy budou jejich studenti vstupovat do praxe.

V současné době se při vývoji prakticky všech velkých projektů používá objektově orientované programování. Prakticky všechny vysoké školy a s nimi i značná část středních škol proto učí své studenty programovat objektově. Na špičkových vzdělávacích pracovištích a konferencích pedagogů se již neřeší otázka, zda učit objektově orientované programování, ale jak je učit. Tomuto tématu bych chtěl věnovat i tento příspěvek.

Světový standard pro vstupní kurzy programování

Vstupní kurzy programování bývají ve světě označovány zkratkou CS1 (Computer Science 1) a na ně navazující kurzy pak zkratkou CS2. Tyto kurzy přitom přestávají být výsadou univerzit, ale postupně se stávají součástí učebních plánů i na řadě středních škol. Jejich jednotné označení a relativně jednotné požadavky na výstupní znalosti jejich absolventů usnadňují celosvětovou diskusi o jejich náplni i o metodikách výuky.

Od šedesátých let se jednou za čas sejde skupina odborníků z vědeckých společností IEEE-CS (Computer Society of the Institute for Electrical and Electronic Engineers – Computer Science) a ACM (Association for Computing Machinery) a specifikují požadavky na osnovy těchto kurzů. Tato kurikula vyšla postupně v letech 1968, 1978, 1991 a poslední ([3]) pak na konci roku 2001.

Jak začít

Mají-li si žáci nějakou látku důkladně osvojit, není možné začít s jejím výkladem až někdy před koncem příslušného kurzu. Tato zásada je obzvlášť důležitá u předmětů, u nichž nestačí se látku pouze naučit, ale je nutné si ji postupně osvojit vyřešením řady praktických úloh. Do této kategorie předmětů patří i výuka programování.

Jak jsem řekl, požadavky kladené v současné době na absolventy těchto kurzů jsou shrnuty v doporučení [3]. Toto doporučení zmiňuje několik základních přístupů k počátečním etapám výuky:

Nejdříve hardware (Hardware-first)
Jeho zastánci tvrdí, že k tomu, aby studenti dokázali správně programovat, musí nejprve vědět, jak je počítač konstruován, protože jedině tak si mohou představit, jak bude jejich program prováděn. Výuka začíná výkladem spínacích obvodů, konstrukcí registrů a aritmetických jednotek, a teprve poté pokračuje výkladem konstrukce programů ve strojovém kódu a následně ve vyšších programovacích jazycích. Tato koncepce se uplatní pouze v několika speciálních oborech, protože většinou, zejména pak při tvorbě rozsáhlých aplikací, je považováno za optimální, je-li programátor od hardwaru co nejvíce odstíněn.

Nejdříve algoritmy (Algorithms-first)
Tento přístup nevyužívá k výkladu některého z existujících jazyků, ale vykládá základní algoritmy za použití pseudokódu. Studenti se nejprve učí základní principy, aniž by se zdržovali laděním nějakých programů. Zkušenost však ukazuje, že právě absence této zpětné vazby a nemožnost si vše vyzkoušet je pro studenty silně demotivující.

Nejdříve příkazy (Imperative-first)
Klasická, a jak odhaduji, u nás stále nejpoužívanější metodika výuky. Při ní se studenti nejprve seznámí s klasickými programovými konstrukcemi a teprve pak s případnou objektově orientovanou nadstavbou. Zkušenost však ukazuje, že takto připravovaní studenti se nesžijí s objektově orientovaným paradigmatem tak

dobře, jako studenti, kteří začali výuku hned prací s objekty, což je vzhledem k současnému významu OOP považováno za velký handicap tohoto přístupu.

Nejdříve funkce (Functional-first)
Tento přístup zavedli v osmdesátých letech v MIT. Jeho výhodou je sjednocení počáteční úrovně studentů, protože se zde setkají s jazykem, jehož filozofie je výrazně jiná než filozofie jazyků hlavního proudu. Tato odlišnost ale na druhou stranu mnohé ze studentů demotivuje, protože se nechtějí učit něco, co pak ve své praxi přímo nepoužijí.

Nejdříve objekty (Objects-first)
Tato koncepce vychází ze skutečnosti, že OOP je zdaleka nejpoužívanější metodikou programování a mají-li si je studenti opravdu osvojit, musí se s ním setkávat od samého počátku výuky.

Nevýhodou tohoto přístupu je, že objektově orientované jazyky bývají koncipovány jako komplexní a pokud vyučující začne studenty seznamovat s daným jazykem v jeho plné šíři, připadají si studenti jejich složitostí zcela zahlceni. Je přitom jedno, zda jde o složitost vlastního jazyka, jak je tomu např. v případě jazyka C++, nebo o složitost standardní knihovny, jak je tomu v případě jazyka Java.

Kurzy vedené tímto přístupem je proto třeba koncipovat tak, aby k tomuto zahlcení nedošlo. Přes tuto drobnou nevýhodu je přístup *Object-first* v současné době nejrozšířenější.

Nejdříve celkový přehled (Breadth-first)
Zastánci této koncepce tvrdí, že by se studenti měli nejprve seznámit s problematikou počítačové vědy v co největší šířce, a teprve pak se soustředit na takové detaily, jakým je např. programování. Studenti prošedší takovýmito kurzy přistupují k řešení problémů z většího nadhledu a jsou jej často schopni chápat v celé jeho šíři. Kritici však této koncepci vytýkají, že odkládá výuku programování a tím i na ni navazující předměty o jeden až dva semestry, což není vždy vyváženo lepšími znalostmi studentů.

Strukturované versus objektově orientované programování

Než se rozhovořím o tom, jak co nejlépe učit OOP, pokusím se nejprve přiblížit těm, kteří ještě nejsou v OOP zcela doma, hlavní rozdíly mezi objektově orientovaným programováním a metodikami, které mu předcházely – především modulárním a strukturovaným programováním.

Předchozí metodiky se soustředily na vymýšlení postupů jak vyřešit zadanou úlohu. Byla-li úloha složitější, rozdělila se na několik úloh jednodušších, které se pak vývojáři snažili řešit pokud možno samostatně. Základním stavebním kamenem těchto programů jsou procedury a funkce, které mívají v průměru několik desítek příkazů.

Objektově orientovaní programátoři postupují poněkud jinak. Chápou svůj program jako simulaci reálného či virtuálního světa, a proto se pokouší nejprve tento svět popsat. Zjišťují, jaké se v tomto světě nacházejí objekty, jaké mají tyto objekty vlastnosti a jak spolu komunikují. Veškeré dění pak převádějí na posílání zpráv mezi objekty. (Usedá-li např. nějaká osoba ve hře na židli, pošle židli zprávu o tom, že se ji zatěžuje svoji vahou a židle na tuto zprávu odpoví bud tichým mlčením, nebo hlasitým zhroucením se.) Vlastní program pak vytvářejí jako popis těchto objektů a zpráv, které si objekty mezi sebou navzájem posílají.

Na zprávy, které danému objektu posílají jiné objekty, reaguje objekt zavoláním příslušných metod, což jsou části kódu definující reakci objekty na zaslání dané zprávy. Metody jsou ekvivalenty klasických procedur a funkcí. Na rozdíl od nich však bývají výrazně jednodušší – mívají většinou od jednoho do deseti příkazů. Metody, které by měly více než 20 příkazů, jsou v dobrých programech spíše výjimečné.

Délka metod je ale pouze vedlejší důsledek jiné filosofie přístupu k řešení problému – jiného paradigmatu. Hlavní výhody objektového přístupu jsou:

Zmenšení sémantické mezery
Jedním z důležitých přínosů OOP je zmenšení sémantické mezery mezi „lidským“ a „programátorským“ popisem problému a jeho řešení. Popis funkce objektového programu je daleko bližší popisu, jakým si daný problém a jeho řešení popíšou mezi sebou lidé. Tvůrci program pak daleko lépe chápou vztah programu k simulované skutečnosti a mohou funkci programu snáze vysvětlit i jeho zadavateli. Dochází proto k mnohem méně nedorozuměním mezi zadavatelem programu a jeho řešiteli.

Zapouzdření implementace
To, že program by měl co nejméně prozrazovat o tom, jak to dělá, že umí to, co umí, hlásaly knihy, zabývající se správnou metodikou programování, již od šedesátých let minulého století. Autoři předobjektových metodik a programovacích jazyků však důležitost dodržování této zásady dostatečně nedocenili, takže jejich metodiky a jazyky pro dosažení tohoto cíle příliš prostředků nenabízely. Naproti tomu OOP staví zapouzdření implementace ve stupnici hodnot na jednu z nejvyšších příček a nabízí proto i řadu prostředků, jak je zabezpečit.

Modifikovatelnost kódu
V dobách, kdy se prosazovaly jednotlivé předobjektové metodiky, si tvůrci programů neuvědomovali nutnost tvorby modifikovatelného kódu. Teprve v průběhu posledních 20 let se ukazovalo stále jasněji, že jedinými programy, které nebudou nikdy modifikovány, jsou programy, které jsou tak špatné, že si zákazníci raději koupí nový, dražší program, než aby žádali autory stávajícího programu o jakékoliv vylepšení. Prakticky každý program, který je jen trochu dobrý, dozná v průběhu svého života řadu změn a vylepšení. Objektově orientované jazyky proto, na rozdíl od svých předchůdců, nabízejí řadu prostředků, které umožňují vytvořit program tak, aby jeho pozdější úpravy vyžadovaly jen minimální úpravy jeho zdrojového kódu.

Nutnost modifikovatelnosti kódu stoupla zejména tehdy, když programátorům začalo docházet, že nezá-

visle na tom, jak pečlivě a dokonale bude provedena počáteční analýza, stejně se bude kód měnit. Nezáleží na tom, jestli byla v analýze chyba nebo jestli zákazník přišel s novou specifikací toho, co se má vlastně naprogramovat. Důležité je, že původní řešení je třeba upravit a že tato oprava musí vyjít co nejlevněji.

Znovupoužitelnost hotových částí
Ti starší z nás ještě zažili doby, kdy svět nebyl ještě zaplaven nejrůznějšími programy řešícími ten či onen problém. Značná část řešených problémů byla řešena poprvé a tvůrci programů nemívali příliš často příležitost zjednodušit si práci tím, že použijí jiný program, který řeší podobnou problematiku nebo alespoň její část. Postupem času sice vznikly nejrůznější knihovny, ale při jejich používání se museli programátoři omezit na řešení nabídnutých autory daných knihoven a většinou neměli žádnou možnost je upravovat, vylepšovat a přizpůsobovat jejich vlastnosti svým požadavkům.

OOP se řídí heslem: „Kolo není třeba pokaždé znovu vynalézt, ale stačí je jen znovu použít!“ Přineslo několik konstrukcí, které na jedno stranu výrazně usnadňují nerůznější modifikace chování kódu převzatého z jiných zdrojů a na druhou stran také konstrukce umožňující definovat kód tak, aby byl co nejlépe znovu použitelný a usnadňoval případné modifikace.

Návrhové vzory
Starší programovací jazyky neumožňovaly nijakou výraznou typizaci používaných postupů. Veškerá typizace se omezovala na algoritmy řešící tu či onu třídu úloh nebo na obecné metodologie. Oblast definice architektury celého systému a jeho jednotlivých částí zůstávala odkázána zcela na zkušenosti a intuici architekta. Nové prostředky a možnosti, které přineslo OOP, umožnily definici tzv. *návrhových vzorů*, které bychom mohli přirovnat k programátorské verzi matematických a fyzikálních vzorečků: kdo je zná, nemusí je odvozovat a je proto s řešením mnohem dříve hotov a navíc dělá méně chyb.

Shrnutí
Každá z výše uvedených přednosti vede k efektivnějšímu vývoji a spolehlivějšímu, robustnějšímu a snáze spravovatelnému kódu. Produktivita vývoje a kvalita výsledného kódu byly hlavními důvody, proč všechny významné programátorské firmy v průběhu osmdesátých let minulého století přešly na OOP a proč si v současné době téměř nikdo netroufne vyvíjet opravdu veliký projekt jinak než objektově.

Oblíbené námitky proti výuce OOP
Na téma výuky OOP jsem publikoval již řadu článků. Pro vás byly asi nejdostupnější články na serveru *Česká škola*. Ti z vás, kteří tento server sledují, si možná vzpomenou, jak vášnivě někteří učitelé obhajovali „staré dobré strukturované programování“. Námitky proti zaváděné výuky OOP do základních a středních škol byly hlavně následující:

OOP je pouze módní vlna
OOP má delší historii než strukturované programování. Hnutí za strukturované programování odstartoval Dijkstrův článek *Go To Statement Considered Harmful* přednesený v roce 1968. Prvním objektově orientovaným jazykem ale byl jazyk *Simula I*, který se narodil v letech 1962 až 1965 (viz např. [4]) a do své nejslavnější verze *Simula 67* byl dotažený o dva roky později.

Strukturované programování se však ujalo rychleji, protože jeho zásady byly mnohem jednodušší a pro průměrného programátora pochopitelnější, takže se mezi programátorskou veřejností snáze prosazovaly. Naproti tomu skutečné výhody OOP byly v průběhu času teprve odhalovány a přiznejme si, že řada programátorů programujících v OOP jazycích (a bohužel i řada vyučujících, kteří OOP přednáší) zásady správného OOP nepochopila dodnes.

Objektové programování se začalo výrazněji prosazovat v osmdesátých letech s příchodem jazyka C++ a zcela ovládlo pole v devadesátých letech, zejména pak po příchodu jazyka Java. Za celou tu dobu jeho pozice stále sílí a v dohledu není zatím ani náznak nějaké ještě lepší alternativy.

OOP vytváří nespolehlivé programy
Oblíbenou námitkou kritiků bývá věta: „Podívejte se, jak nespolehlivá jsou objektová *Windows* a porovnejte je třeba s programem XYZ, který není naprogramován objektově.“

Tady lze pouze namítnout, že tak rozsáhlý program jako je operační systém *Windows* není v lidských silách bez použití OOP s rozumnými náklady vůbec naprogramovat. Programy této míry obludnosti jsou bez OOP s akceptovatelnými náklady a akceptovatelnou spolehlivostí a udržovatelností nenaprogramovatelné.

Každopádně firmy, kterým na spolehlivosti objednaných programů maximálně záleží, např. banky, již ani jiný, než objektově vytvořený program neakceptují.

OOP je pro naše studenty příliš obtížné
Moje zkušenosti ukazují, že OOP je pro děti snáze stravitelné než pro dospělé, zkušené programátory. Ti mají s jeho naučením mnohem větší problémy.

Kroužky, v nichž jsem učil, prošla řada 12letých dětí a žádné nemělo s pochopením OOP nějaké zvláštní problémy. Zato dospělí, které přeškoluji z klasického programování na objektově orientované, zápasí s objektovou orientovaností mnohem více. Obdobné problémy měly i děti, které přišly do mého kroužku již se znalostí klasického programování, a museli se proto nejprve odnaučit některé ze svých dosavadních nevhodných návyků.

Základ úspěchu dětí tkví v tom, že děti nemají nutkavou potřebu zasadit nové poznatky do kontextu stávajících znalostí. Děti jsou totiž zvyklé se učit něco, s čím se doposud nepotkaly a k čemu teprve dodatečně přiřadí potřebný kontext.

Naproti dospělí tuto potřebu mají a vše, co o OOP prohlásím, se zpočátku pokoušejí interpretovat prostřednictvím takového programování, s nímž mají své zkušenosti. Základem jejich úspěšné a efektivní výuky

je proto uvést je co nejdříve do prostoru, v němž nebudou moci své dosavadní zkušenosti použít, a budou nuceni (stejně jako děti) vybudovat vše od základů.

Většina absolventů bude beztak programovat jen webové stánky a tam OOP nepotřebuje
Podíváte-li se na nabídky pracovních míst, zjistíte, že programátoři, kteří nezvládají objektově orientované programování, jsou stále hůře uplatnitelní. Nebudeme-li se dívat pouze na současný stav, ale pokusíme-li se nahlédnout do doby, kdy budou naši absolventi odcházet do praxe, bude uplatnění neobjektového programátora téměř vyloučené.

Dosavadní zkušenosti

Moje zkušenosti s výukou OOP pocházejí ze čtyř zdrojů:

- zájmové kroužky programování navštěvované dětmi ve věku od 11 do 16 let,
- studenti VŠE v předmětu základy programování,
- dospělí, kteří se rozhodnou naučit programovat,
- profesionální programátoři přecházející z klasického programování na programování objektově orientované.

Se všemi uvedenými kategoriemi studentů mám obdobné zkušenosti: ti, kteří ještě nikdy neprogramovali, mají sice na počátku více problémů s napsáním byť jednoduchých programů, protože jim zpočátku dělá potíže syntaxe. Objektová technologie je však nezaskočí, protože se ji nesnaží interpretovat na základě neobjektových zkušeností (jak také, když žádné nemají) a naopak lépe odpovídá jejich dosavadním návykům z reálného světa.

Naproti tomu ti, kteří již programovali a chtějí pouze zvládnout novou technologii, nemívají problémy se syntaxí, ale zase jim chvilku trvá, než se naučí, že při OOP musejí myslet trochu jinak. Úpěnlivě se drží svých dosavadních zvyklostí a některé nové zásady, které si s jejich dosavadními zkušenostmi protiřečí, si osvojují problematicky.

Osvědčila se mi proto metodika, kterou s oblibou charakterizuji slovy *co nejdříve jim podříznout větev dosavadních zkušeností, na které sedí*. Tato metodika doporučuje studenty co nejdříve uvádět do situací, které s jejich dosavadními zkušenostmi řešit nezvládnou a nezbude jim, než použít objektové technologie.

Tento postup se osvědčil i na vysoké škole, kde toto časné „podříznutí větve“ do jisté míry sjednocuje úroveň studentů, kteří ještě nikdy neprogramovali se studenty, kteří přišli s poměrně bohatými (strukturovanými) programátorskými zkušenostmi.

Současná trendy

Jak jsem již řekl, objektově orientované programování je hlavním proudem v programování již téměř 20 let. Přibližně stejně staré jsou i první práce, které ukazovaly, že výuka, při níž se práce s objekty probírá až v závěru kurzu, nepřináší zdaleka tak dobré výsledky jako výuka, při které prací s objekty naopak začíná.

V druhé polovině devadesátých let se na konferencích začaly houfně objevovat příspěvky, které přinášely první zkušenosti s umístěním výkladu objektu na počátek výuky a objevily se dokonce i vývojové nástroje, které tento způsob výuky vysloveně předpokládaly. Mezi nimi je pravděpodobně nejznámější vývojové prostředí specializované na výuku – *BlueJ*. Jednou z jeho velkých výhod je, že umožňuje studentům přemýšlet přímo nad diagramem tříd a „nezahání“ je hned k řádkům napsaného kódu.

V posledních letech se proto metodika *Nejdříve objekty* stala nejrozšířenější metodikou vstupních kurzů programování na vysokých školách a stále více se prosazuje i na školách středních.

Podíváte-li se na současná témata příspěvků v různých konferencích či dokonce na témata celých konferencí, zjistíte, že stále sílí názor, že samotné umístění výkladu objektů a práce s nimi na počátek výuky nestačí. Mají-li se zásady objektově orientovaného programování „vrýt studentům důkladně pod kůži“, musí se je od samého začátku učit dodržovat.

Jedním z častých témat odborných debat je zařazení návrhových vzorů (design patterns) v co nejčasnějším stádiu výuky. Na toto téma jsou dokonce pořádány celé konference a pracovní setkání. Nejznámější z nich je nejspíš seminář *„Killer Examples“ for Design Patterns and Objects First* pořádaný v San diegu, který se loni konal již počtvrté. Termínem *killer examples* jsou zde označovány příklady, které jsou schopny „zabít starou metodiku“, protože na nich všichni pochopí, kudy vede ta jediná správná cesta.

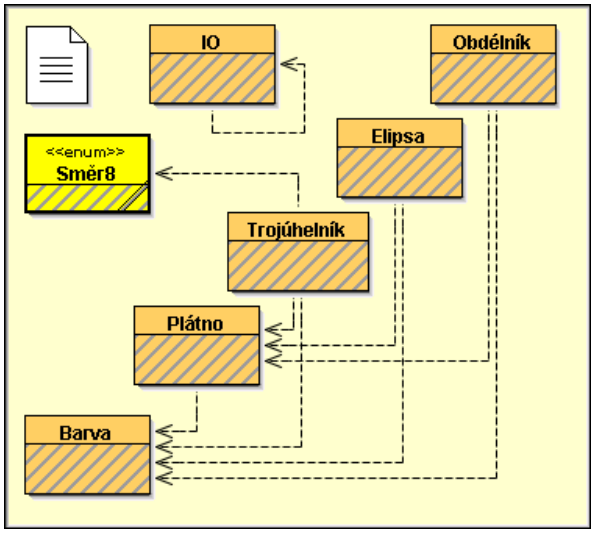
Metodika „Nejdříve návrhové vzory“

(Podklady pro workshop)
Moje zkušenosti ukazují, že tyto trendy jsou nanejvýš opodstatněné. Proto jsem metodiku, kterou jsem publikoval v [9] dále upravoval. Výklad konstrukce `interface` a návrhových vzorů jsem stále přesouval blíže k počátku kurzu, až se dostaly na jeho úplný začátek (viz [6]).

Jednou z klíčových vlastností metodiky je to, že studenti nemají za úkol vytvářet nějaké samostatné, a proto nutně nesmírně jednoduché programy, ale že mají rozšiřovat funkčnost nějakého rozsáhlého (alespoň pro ně), předem připraveného projektu.

Náplň výuky se tak mnohem více blíží potřebám praxe, protože převážnou většinou programátorských úkolů je právě rozšíření funkčnosti nějakého stávajícího projektu vytvořeného navíc většinou někým jiným.

První hodina: objekty, testovací metody a význam návrhových vzorů
Na první hodině/cvičení se studenti s objekty a třídami většinou teprve seznamují. Díky použitému vývojovému prostředí mohou přímo oslovovat jednotlivé třídy v projektu i jednotlivé vytvořené instance. Prohlédnou si strukturu nějakého předem připraveného projektu (viz obr. 1) a ujasní si vzájemné závislosti a interakce jednotlivých tříd a jejich instancí. Osvojí si pojmy třída a instance a naučí se vytvářet instance tříd v projektu.



Obr. 1: *Diagram tříd úvodního projektu*

V této etapě je důležité, aby studenti pochopili, že v programech je za objekt považováno opravdu vše, včetně toho, co by v běžném životě za objekt nepovažovali – např. různé vlastnosti, vztahy, činnosti apod. My se to snažíme našim studentům přiblížit tím, že hned v prvním projektu, v němž pracují s grafickými tvary, narazí na třídy `Barva` a `Směr`.

Používáme-li vhodné interaktivní prostředí, jakým je např. stále populárnější prostředí *BlueJ*, mohou studenti hned na počátku vytvářet nejenom instance tříd, které již v projektu existují, ale mohou definovat i svoji vlastní třídu – konkrétně testovací třídu, která si ve formě jednotlivých testů zapamatuje operace, jež v projektu s jeho třídami a jejich instancemi prováděli. Vytvoření vlastní testovací třídy, jejíž testy např. nakreslí nějaké zajímavé obrázky, může být při výuce na základních a středních školách prvním domácím úkolem.

Současně se můžeme seznámit i s prvními návrhovými vzory. Takto na počátku nemůžeme samozřejmě vysvětlovat konstrukci těchto vzorů, ale můžeme je seznámit s typickými úlohami, jež návrhové vzory řeší. V našem příkladu s geometrickými obrazci upozorňujeme na to, že se v něm vyskytuje třída, která nemá žádnou instanci (pomocná třída `IO`), třída mající právě jednu instanci (`Plátno`), třída mající předem známý počet předem známých instancí (výčtový typ `Směr`), třída, která sice nemá počet instancí omezen, ale sama jej řídí (třída `Barva`, která neumožní vytvořit dvě ekvivalentní instance) a třídy, které mohou mít libovolný počet operativně vytvářených instancí (geometrické třídy `Obdélník`, `Elipsa` a `Trojúhelník`).

Druhá hodina: první textový program, návrhový vzor *Přepravka*

Druhá hodina je věnována tvorbě prvních textových programů a návrhovému vzoru *Přepravka*.

Nejprve na úvodní hrátky s objekty navážeme konstrukcí prázdné třídy (pojmenujeme ji příhodně `Prázdná`), kterou postupně doplníme o konstruktor, jenž nakreslí kruh, které jsme minulou hodinu kreslili interaktivně. Poté si předvedeme možnosti definice přetížených verzí konstruktorů a seznámíme se s klíčovým slovem **this**.

Nyní přejmenujeme naši prázdnou třídu na `Světlo` a pokračujeme definicí metod, které mají naše světlo rozsvítit a zhasnout. Ukážeme si jak definovat potřebné atributy a současně si předvedeme, jak instance naší třídy se svými atributy pracují. Doplníme metodu `blikni()`, která naše světlo na předem definovanou dobu rozsvítí a po jejím uplynutí je zase zhasne.

Pokračujeme definicí parametrických metod: metody `setPozice(int,int)`, která obrazec přesune do nové pozice a metodami `getX()` a `getY()`, které oznamují, kde se obrazec nachází. Ukážeme si, že k definici přístupových metod není třeba vždy definovat explicitní atributy, ale že je možné hodnoty těchto atributů na poslední chvíli spočítat.

V tuto chvíli upozorníme na těžkopádnost práce s hodnotami, které mají více složek a na nemožnost definice metody vracející více hodnot. Seznámíme studenty s návrhovým vzorem *Přepravka* a definujeme třídu `Pozice` a v našich třídách obrazců zavedeme metody `getPozice()` a `setPozice(Pozice)`, které s přepravkami pracují.

Za domácí úkol dostanou studenti vytvořit přepravku pro uchování informací o rozměru obrazců a doplnění metod, které budou vracet, resp. nastavovat rozměr objektu. Současně dostanou za úkol definovat třídu, jejíž instance by vykreslily půdorys auta, které má v rozích umístěna světla.

Třetí hodina: interface, návrhové vzory *Služebník*, *Pozorovatel*

Ve třetí hodině chceme po studentech, kteří ještě neznají podmínku ani cyklus, aby pro světlo definovali metodu `blikej(int)`, po jejímž vyvolání světlo požadovaný počet krát zabliká.

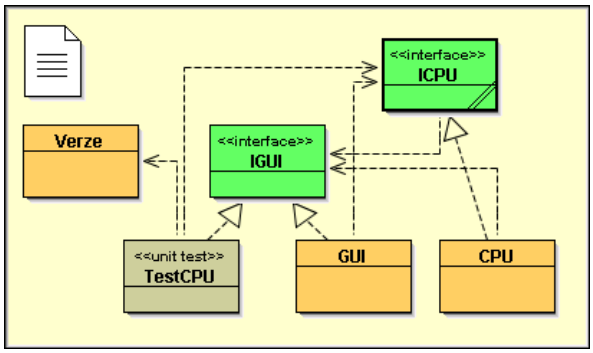
Ti zkušenější sice navrhnou použití cyklu, avšak většinou neumí vyřešit doplňující úlohu, aby světlo bylo schopno provádět při blikání další činnosti – např. aby auto s blikajícími světla dále jelo. Seznámíme je proto s návrhovým vzorem *Služebník* a vysvětlíme princip jeho použití. Definujeme pak rozhraní `IAkční`, jehož instance implementují metodu `akce()` a poskytneme studentům třídu `Opakovač`, která nabízí metody umožňující opakovat akci svého akčního parametru.

V dalším výkladu upozorníme na nevýhody současného projektu, v němž pohybující se obrazce přemazávají obrazce, přes něž se přesouvají. Otevřeme proto nový projekt, v němž je dosavadní třída `Plátno` nahrazena třídou `AktivníPlátno`, která není klasickým, pasivním plátnem, ale je dispečerem, který řídí překreslování obrazců, jež mu byly svěřeny do péče.

S aktivním plátnem vstupuje do našich programů návrhový vzor *Pozorovatel*, který implementuje pravidlo známé jako Hollywoodský princip: „Nevolejte nám, zavoláme vám.“ Ukážeme, jak se instance – pozorovatelé u aktivního plátna přihlašují a jak pak takové přihlášení ovlivní jejich další život.

Čtvrtá hodina: kalkulačka, návrhové vzory *Most* a *Strategie*

V další hodině studentům předvedeme program, který realizuje jednoduchou kalkulačku. Ukážeme jim, jak vhodnou definicí architektury aplikace mohou dosáhnout toho, že jedno grafické uživatelské rozhraní (GUI) může spolupracovat s několika CPU – např. s celočíselnou, reálnou, zlomkovou atd. (viz obr. 2). Jako první semestrální práci pak studenti dostanou za úkol definovat vlastní verzi CPU kalkulačky.



Obr. 2: *Diagram tříd projektu Kalkulačka*

Duch výuky

V průběhu celé výuky je třeba studenty neustále seznamovat s hotovými programy a ukazovat jim na nich, jak se v současné době programuje. Je potřeba, aby se co nejdříve naučili „přemýšlet objektově“ a vytvářet své programy podle stejných zásad, podle nichž byly vytvořeny programy, s nimiž se setkávají na hodinách.

Závěr

Objektově orientované programování je v současné době zdaleka nejrozšířenějším paradigmatem. Vývoj rozsáhlejších programů je jinými technologiemi prakticky neřešitelný.

Nejrozšířenější metodikou výuky objektově orientovaného programování je metodika *Object First*, která studentům umožní se od samého počátku výuky pohybovat v objektovém světě. Tuto metodiku je ale výhodné nahradit metodikou *Design Patterns First*, která poskytuje studentům mnohem intenzivnější seznámení s moderními programovacími technikami a postupy.

Literatura

[1] *4th "Killer Examples" for Design Patterns and Objects First workshop* http://www.cse.buffalo.edu/faculty/alphonce/KillerExamples/OOPSLA2005/

[2] BARNES David, KÖLLING Michael *Objects First With Java: A Practical Introduction Using BlueJ (2nd edition)*. Prentice Hall, 2004. ISBN 0-131-24933-9.

[3] *Computing Curricula 2001, Computer Science Volume.* http://www.sigcse.org/cc2001//; PDF verze: http://www.sigcse.org/cc2001/cc2001.pdf

[4] DAHL Ole-Johan; NYGAARD Kristen: *How Object-Oriented Programming Started*, http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

[5] FREEMAN Elisabeth, FREEMAN Eric *Head First Design Patterns*. O'Reilly, 2004. ISBN 0-596-00712-4.

[6] 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.

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

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

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

[10] PECINOVSKÝ Rudolf: Jak při výuce Javy opravdu začít s objekty. *Objekty 2004 – sborník příspěvků devátého ročníku konference*, ČZU, Praha 2004. ISBN 80-248-0672-X.

[11] PECINOVSKÝ Rudolf: Proč a jak učit OOP žáky základních a středních škol. *Žilinská didaktická konferencia*, 2004, Žilina.

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

[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.

[14] *The ACM Java Task Force – Project Rationale*, Second Public Draft (February 23, 2006), ke stažení na adrese http://www-cs-fa-ulty.stanford.edu/~eroberts//jtf/rationale/rationale.pdf