

# Jak při výuce Javy opravdu začít s objekty

Rudolf Pecinovský<sup>1</sup>

<sup>1</sup>Amaio Technologies, Inc., Třebohostická 14, 100 00, Praha 10  
rudolf@pecinovsky.cz

**Abstrakt.** Příspěvek seznamuje s metodikou výuky programování, při které studenti pracují s objekty od samého počátku kurzu a ne až po předešlé zabývající se klasickými strukturovanými konstrukcemi. Ty se podle této metodiky probírají až v závěrečných etapách výuky. Příspěvek vysvětluje nevýhody dosavadního standardního přístupu a ukazuje, jak jej lze s využitím současných vývojových nástrojů změnit. Předvádí, jak je třeba volit příklady, a uvádí i zkušenosti z dvouletého používání předvedené metodiky v kurzech profesionálních programátorů i kroužcích dětí základních a středních škol.

**Klíčová slova:** OOP, Java, výuka programování, metodika

## 1 Motivace

Při prohlídce současných učebnic programování a na internetu zveřejněných prezentací přednášek univerzitních kurzů zjistíme, že všichni vyučující sice hovoří o důležitosti objektově orientovaného přístupu, ale naprostá většina začne výkladem primitivních datových typů a klasických strukturovaných konstrukcí. Studenti se tak na počátku seznamují se zcela jiným paradigmatem, než je to, které by si měli v průběhu kurzu osvojit.

Nepříjemnou vlastností takto koncipovaného výkladu je, že studenti na počátku vstřebávají základy strukturovaného programování, ty v jejich mysli zakoření, takže objektové rysy, s nimiž se seznámí v další části kurzu, se pak snaží naroubovat na tento strukturovaný kmen. Osvojení objektového myšlení jim pak většinou trvá výrazně déle, než kdyby s objekty pracovali hned a nové poznatky usazovali do čistého, předchozími programátorskými zkušenostmi nepoznamenaného prostoru.

Když jsem před několika lety začal školit Javu, vyšel jsem ve svých kurzech ze známé učebnice [12], která byla koncipována právě takto. Pozoroval jsem pak, že přestože frekventanti kurzů bez problémů zvládli syntaxi a teoretické základy použití objektových rysů jazyka, jejich programy příliš objektově orientované nebyly.

Tehdy jsem se seznámil s učebnicí [9], jejíž autoři vyvinuli vývojové prostředí *BlueJ*, které jim umožnilo začít výuku experimenty s třídami a objekty a hned první vytvářené programy koncipovat jako doopravdy objektové.

Změna filozofie výkladu, s níž autoři této učebnice přišli a kterou vysvětlili v řadě článků (viz např. [6] – texty tohoto a řady dalších článků najdete na adrese [2]), mi připomněla počátek osmdesátých let, kdy jsme začali zavádět metodiku Karel. Ta také převracela tehdy běžnou posloupnost výkladu, při níž se procedury a funkce probíraly až po předchozím výkladu datových typů, podmíněných příkazů a cyklů.

Vývojové prostředí *BlueJ* nyní umožňovalo (stejně, jako tehdy prostředí robota Karla) převrátit zaužívanou posloupnost výkladu a doopravdy začít výkladem toho, co

by měli studenti zvládnout především, a to je práce s třídami a objekty. Nejenom teoretickým výkladem vysvětlujícím, co to jsou objekty, ale praktickými experimenty následovanými tvorbou programů realizujících funkce, které studenti poznali při předchozích experimentech.

Při hlubším studiu možností, které tento způsob výkladu nabízel, jsem byl čím dále tím více přesvědčen, že autoři [9] přišli s geniální myšlenkou, avšak nevyužili plně její potenciál. Zkusil jsem proto dotáhnout naznačené myšlenky směrem, o němž jsem se domníval, že využije potenciál původního nápadu efektivněji.

Novou metodiku jsem vyzkoušel v kroužcích programování navštěvovaných žáky základních a středních škol. Po těchto „testech na dětech“ jsem ji začal používat ve svých kurzech pro profesionální programátory a podle reakcí posluchačů soudím, že úspěšně (viz např. [3]).

## 2 Základní charakteristika

Metodika postupně procházela řadou úprav. Poslední stav je publikován v [4]. Snažil jsem se přitom, aby veškerý výklad i doprovodné programy dodržovaly pravidla uvedená v [8], z nichž bych zdůraznil:

- Co nejdříve umožnit tvorbu programů.
- Příklady musí být zajímavé.
- Příklady musí vyžadovat aktivní použití nových poznatků.
- Řešené problémy by neměly být z hlediska studentů triviální.
- Řešení nesmí být příliš zašuměná, tj. část programu, v níž se používá probíraná konstrukce, nesmí být „ztracená“ ve zbytku programu.
- Studenti se musí naučit programy nejen vytvářet, ale také ladit.

Metodika vychází ze zásady, že studenti by měli nejprve pochopit základní vlastnosti objektů a tříd a „ručně si osahat“, jak vlastně takový objektově orientovaný program pracuje. Vyzkoušet reakce tříd a objektů na zaslání zpráv, pochopit základní vlastnosti odkazů na objekty, ujasnit si rozdíl mezi zprávami posílanými třídě a instancí, získat představu o „vnitřní struktuře“ objektu atd.

Po těchto osobních zkušenostech se začínou seznamovat se zápisem základních objektových rysů, s nimiž před tím experimentovali. Chování, které v první, experimentální etapě pozorovali, se nyní učí naprogramovat. Naučí se vytvářet třídy, jejichž instance budou schopny spolupracovat s instancemi ostatních tříd v dané aplikaci. Dozvědí se, jak definovat reakce tříd i jejich instancí na zaslání zpráv a jak zařídit, aby si instance i třídy pamatovaly svůj stav a mohly podle něj reagovat.

Ve třetí etapě začínou rozšiřovat svoje znalosti objektových rysů a konstrukcí. Paralelně se seznámí s několika základními návrhovými vzory (knihovná třída, přepravka, tovární metoda, jedináček, služebník, stav, zástupce).

Nejprve si prohloubí svoje vědomosti o třídách a jejich instancích a poté se naučí pracovat s rozhraními a vyzkoušejí si různé možnosti jejich použití. Přitom objeví některé problémy a dozvědí se, jak je lze řešit pomocí dědičností rozhraní.

Poté, co zvládnou a alespoň částečně zažijí práci s rozhraními, přejdeme k další konstrukci, kterou je dědičnost tříd. Výklad se však neomezuje na pouhé seznámení

s mechanismem dědičnosti, ale probírá dědičnost opravdu zevrubně a po počátečním předvedení jejích základních rysů se soustřeďuje především na probrání jejích nebezpečných vlastností od narušení zapouzdření až k různým omezením, o nichž se běžné kurzy většinou bohužel nezmiňují. Tato omezení jsou totiž častým zdrojem zálužných začátečnických chyb, které nepoučení začátečníci odhalují s velkými obtížemi.

Ti, kteří, přišli s předchozími zkušenostmi ze strukturovaného světa, začnou v průběhu této etapy výuky pomalu připouštět, že objektivě orientované programování vyžaduje naprosto jiný způsob uvažování. Protože stále nemají k dispozici prostředky, na které byli z dřívějších dob zvyklí a jimiž by mohli některé objektivě konstrukce obejít, musí využívat pouze přednesené objektivě konstrukce, nezbude jim proto, než začít při řešení zadaných úloh přemýšlet opravdu objektivě.

Jakmile studenti vstřebají základy objektivě paradigmatu, můžeme je seznámit i s klasickými konstrukcemi, jakými jsou podmíněné příkazy a cykly. Zavedení těchto konstrukcí v takto pozdní fázi výuky zabezpečí, že je studenti budou používat opravdu pouze k dosažení požadované funkčnosti metod a nebudou jimi nahrazovat výhodnější použití objektivě konstrukcí, které by už měli mít v této době zažité.

První běh uzavírá výklad práce s dynamickými kontejnery, následovaný výkladem polí. Toto řazení ale již není v současných učebnicích výjimečné, protože pro začátečníky je pochopení práce se základními dynamickými kontejnery mnohem jednodušší než pochopení všech aspektů práce s klasickými poli.

Celý výklad je neustále veden na příkladech. Snažil jsem se, aby tyto příklady nebyly jen jednoduchými *aha-příklady*, na nichž si studenti pouze ověří, že daná konstrukce pracuje tak, jak bylo vyloženo, ale aby to byly příklady, které opravdu něco řeší a mají viditelný a pokud možno zajímavý výstup.

Mnohé z úloh jsou koncipovány tak, aby se při jejich analýze přímo nabízela řešení, která se následně ukáží jako chybná. Studenti pak tato řešení většinou použijí, protože se na první pohled zdají být těmi nejlepšími možnými. Na těchto řešeních pak studentům demonstrují nejčastější začátečnické chyby i způsoby, jak těmto chybám předejít.

Hlavním účelem takového postupu je dosáhnout osobního prožitku studenta s tímto druhem chyby a jejím vyřešením. Takovýto prožitek zanechá v studentech daleko lepší povědomí o možných úskalích některých konstrukcí než pouhý výklad s taxativně vyjmenovanými potenciálními problémy.

### 3 První zkušenosti

Postupy vyzkoušené v kroužcích navštěvovaných 10letými až 16letými dětmi jsem posléze aplikoval v kurzech pro profesionální programátory pořádaných naší firmou. K mému překvapení měli profesionální programátoři, které jsem učil v kurzech, daleko větší problém s přijetím objektivě paradigmatu, než před tím děti v kroužcích. V kurzech si navíc nejrychleji osvojovali objektivě uvažování většinou ti, kteří před tím nikdy neprogramovali. Tato skutečnost pouze potvrdila známý fakt, že předchozí zkušenosti s klasickým, strukturovaným programováním jsou při výuce objektivě orientovaného programování spíše přítěží.

Čím je programátor zkušenější, tím obtížnější je pro něj opustit zaužívaná strukturovaná paradigmatu a přejít na objektivě orientovaný způsob uvažování. Na zkuše-

ných programátorech je vidět, jak se první jeden až dva dny týdenního kurzu snaží veškeré nové informace interpretovat prostřednictvím toho, s čím pracovali doposud. Nepomáhá žádné přesvědčování o tom, aby na tyto své zkušenosti zapomněli hned na začátku. Nemohou překročit svůj stín a dokud budou schopni interpretovat nové informace klasickými prostředky, naučené paradigma neopustí.

V průběhu druhého až třetího dne začnou zjišťovat, že už s dosavadními zkušenostmi nevystačí, a začnou pomalu akceptovat myšlenku, že zadané úlohy je opravdu výhodnější a rychlejší řešit tak, že opustí všechny doposud užívané postupy a přijmou za své objektové řešení. Jeden programátor charakterizoval tento přechod slovy: „Za první tři dny kurzu jsem pochopil daleko víc, než za předchozí půlrok intenzivního samostudia.“

Zde se potvrdilo, že zmínky o konstrukcích, které programátoři znají ze své předchozí praxe, tj. o podmíněných příkazech, cyklech a klasických polích, je vhodné odsunout až do závěrečných lekcí. Čím déle nemohou programátoři používat tyto konstrukce jako alternativu k objektovému řešení, tím větší je u nich šance, že konečně přestanou interpretovat objektové konstrukce strukturovanými prostředky, osvojí si objektově orientovaný způsob řešení problémů a časem jej přijmou za vlastní.

Při klasickém způsobu výuky, o němž jsem se zmiňoval v úvodu, je kladen důraz na výklad syntaxe a sémantiky jednotlivých konstrukcí jazyka. Tento způsob „nepodřízne“ zavčas pod programátorem „větev jeho dosavadních zkušeností“. Zkušenosti programátoři pak často odcházejí z takovýchto kurzů se znalostí příslušných konstrukcí jazyka, avšak bez schopnosti vytvářet skutečně objektově orientované programy.

Předkládaná metodika se snaží tyto problémy reflektovat a upravit postup výuky tak, aby účastníky kurzů přiměla na nové paradigma přejít.

## 4 Důležitost použitého prostředí

Říkal jsem, že studenti by si měli před vlastním programováním vyzkoušet, jak vlastně objektové programy pracují, a zkusit s nimi chvíli experimentovat. K takovýmto experimentům je ale potřeba prostředí, které uživateli umožní přímo oslovovat jednotlivé třídy a jejich instance. Učíme-li programovat v jazyku Java, máme naštěstí takovéto prostředí k dispozici – je jím prostředí *BlueJ*, jehož hlavní rysy jsou popsány v tomto sborníku v článku *BlueJ – vývojové prostředí pro výuku jazyka Java*.

*BlueJ* nám umožňuje předvést, jak takový objektově orientovaný program funguje. S jeho pomocí můžeme demonstrovat, jak se vytvářejí objekty jako instance jednotlivých tříd, jak objektům posílat zprávy, jak můžeme posílat zprávy i celé třídě, ukázat, že si objekty i celá třída mohou ukládat informace do atributů, vysvětlit rozdíl mezi primitivním a objektovým datovým typem (to je sice specialita Javy, ale vysvětlit se musí) a řadu dalších rysů a vlastností tříd, objektů a objektově naprogramovaných aplikací.

Když pak v dalších etapách vytvářejí studenti své vlastní programy, mohou je v průběhu vývoje pomocí stejného mechanismu zkusit a ověřit si tak, že se svými třídami a jejich instancemi mohou komunikovat naprosto stejně, jako před tím komunikovali s předpřipravenými třídami a jejich instancemi.

## 5 Jak volit a připravovat příklady

Chceme-li učit podle této metodiky, nestačí pouze vymyslet příklady a nechat je studenty naprogramovat. Chceme-li odložit výklad podmíněných příkazů a cyklů až do závěrečných etap, museli bychom zadávat jen opravdu primitivní úlohy, na kterých by se toho pak studenti moc nenaučili.

Mají-li studenti při řešení úloh postupně vstřebávat objektové paradigma, musíme jim nabízet úlohy složitější. Ty ale studenti nemohou řešit celé, protože k tomu nemají potřebné znalosti a zkušenosti. Je jim proto potřeba připravit polotovary, které k dosažení plné funkčnosti vyžadují doplnění předem známých tříd. Úkolem studentů je pak tvorba těchto tříd a jejich začlenění do celkového projektu.

Při takto koncipovaných příkladech můžeme navíc daleko dříve zahrnout do výkladu témata, která bychom jinak museli zařadit až na závěr kurzu nebo dokonce do následných kurzů. Typickým příkladem takovýchto témat je např. událostmi řízené programování nebo některé návrhové vzory. Právě tato náročnější témata ale umožňují studentům lépe pochopit objektové paradigma, naučit se „myslet objektově“. U primitivních příkladů totiž nebývá nutnost jejich „objektového řešení“ tak zřejmá.

Při zadávání úloh se mi osvědčilo doplňovat sadu předpřipravených tříd i „prázdny“ verzemi tříd, které mají studenti za úkol doplnit. Zdrojový kód těchto prázdných tříd obsahuje definice prázdných verzí všech metod uvedených v zadání a dokumentační komentáře třídy a všech předdefinovaných metod. Studenti tak dostanou zadání ve tvaru, kdy je požadovaná funkce třídy a jejich metod popsána přímo u míst, kde má student za úkol tuto funkčnost realizovat.

Dostanou-li studenti takto předpřipravené definice tříd, většinou zadání lépe pochopí. Navíc je daleko pravděpodobnější, že vybaví dokumentačními komentáři i ty metody, jejichž definice není explicitně vyžadována zadáním a není proto předpřipravena v definici prázdné třídy, nicméně pro vyřešení úlohy je definovat musí. Neučiní-li tak, budou tyto metody v okolním textu příliš „svítit“.

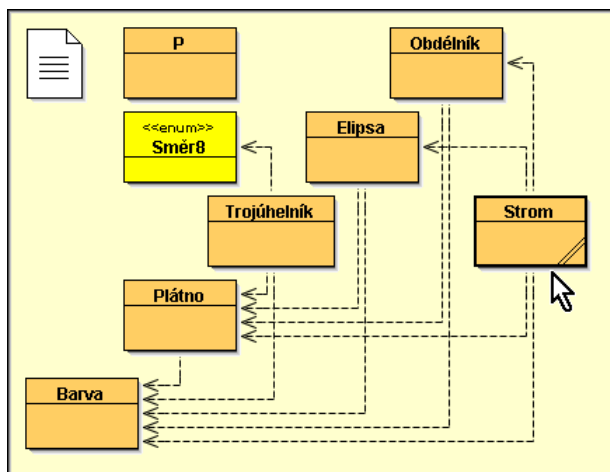
Předpřipravené prázdné třídy doplňují i příslušnými testovacími třídami, které umožňují průběžně kontrolovat postup řešení. V skrytu duše přitom doufám, že když si programátor zvykne na pohodlí programování zabezpečeného testy, bude předem připravené testy používat i ve své další praxi, i když si je pak bude muset napsat sám.

Prázdny a testovací třídy jsou především reakcí na lenost průměrného studenta, který je většinou mnohem ochotnější se pustit do řešení úlohy, která se na první pohled zdá být téměř vyřešená. Jak mi prozradili frekventanti mých kurzů, pro mnohé z nich je přítomnost prázdných tříd a je doplňujících testovacích tříd velice důležitým motivačním faktorem při rozhodnutí, zda úlohu zkusit vyřešit. (Studenti v našich placených kurzech domácí úkoly povinně řešit nemusí, pro jejich řešení se rozhodují zcela dobrovolně.)

## 6 Příklady příkladů

Jak jsem již uvedl, veškerá výuka v mých kurzech je postavena na příkladech. Zavedení každé další konstrukce je motivováno snahou vyřešit úlohu, která s dosavadními znalostmi vyřešit nešla, a pokud ano, tak obtížně.

## 6.1 Základní tvary, Strom, návrhový vzor Knihovni třída



Obr. 1: Diagram tříd vstupního projektu s definicí třídy `Strom`

Na počátku se studenti seznámí s projektem obsahujícím několik tříd reprezentujících jednoduché grafické tvary a několik podpůrných tříd. Po počátečních experimentech mají za úkol definovat třídu reprezentující nějaký složený grafický obrazec (na obr. 1 je to třída `Strom`), určit, jaké bude mít atributy a definovat metody, které budou ekvivalenty metod, s nimiž se seznámili při experimentech s grafickými obrazy.

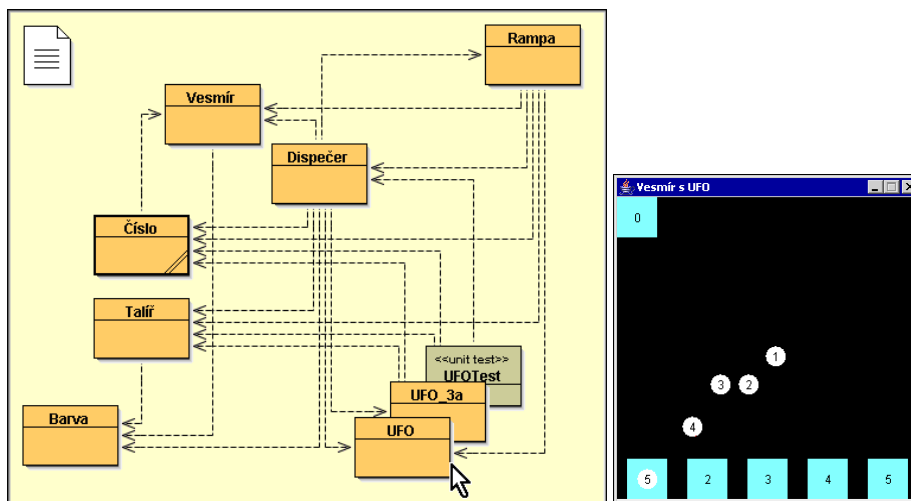
Jak je vidět z obrázku, studenti pracují hned od počátku na projektu, který je z jejich pohledu poměrně složitý. Navíc řeší problémy (konkrétně vykreslení a pohyb vytvořených obrazců), které by bez pomoci předdefinovaných tříd nezvládli. Vysvětlují jim, že v praxi také budou muset většinou doplňovat novou funkčnost do existujícího projektu a pouze výjimečně budou mít šanci vytvořit něco zcela nového.

U tohoto projektu bych chtěl současně upozornit na to, že bychom měli studentům hned od počátku vkládat do projektů třídy, jejichž instance by v běžném chápání za objekty nepovažovali. V předchozím diagramu jsou to třídy `Barva` a `Směr8`. Studenti si tak hned od počátku uvědomují, že v OO programech je objektem opravdu všechno, tedy i např. vlastnosti.

## 6.2 UFO

V závěrečném samostatném projektu první etapy mají studenti za úkol definovat třídu reprezentující UFO ve vesmíru ovládaném dispečerem. UFO sestává z talíře, jenž konstruktor obdrží jako parametr, a čísla, jež musí vytvořit na základě pořadí předaného jako druhý parametr.

Úkolem je naprogramovat konstruktor a sadu jednoduchých metod, které definují reakci instancí třídy `UFO` na stisky kláves a na zprávy okolních tříd. Vytvoří tak jednoduchou hru, ve níž hráč vždy požádá dispečera o přistavení dalšího UFO, které se pak pokusí dopravit do hangáru.



Obr. 2: Diagram tříd a aplikační okno projektu UFO

Hra je zajímavá mimo jiné tím, že hráči ovládají klávesami tah motorů a tím zrychlení UFO. Vypnutím motorů proto dosáhnou pouze toho, že UFO přestane zrychlovat. Navíc je koncipována tak, že mohou mít ve vesmíru několik pohybujících se UFO současně a zadávat, které z nich chtějí v daném okamžiku ovládat.

Naprogramování třídy UFO složité není. Zvládnutí řízení UFO prostřednictvím ovládání tahu motorů však pro některé hráče složité je. Tím, že není zcela jednoduché hru zvládnout, stoupne zejména v očích začínajících programátorů její zdánlivá složitost a tím i jejich víra ve vlastní schopnosti a současně představa, jak složitý projekt jsou již schopni zvládnout.

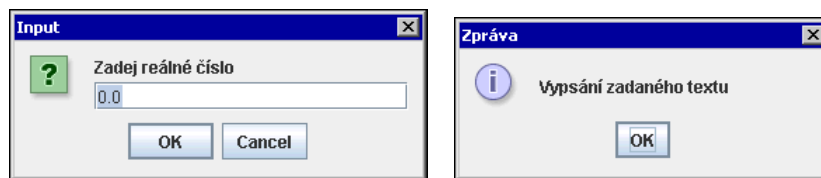
UFO je první z projektů, v nichž studenti (spolu)vytvářejí program řízený událostmi. Takových bude v průběhu kurzu naprostá většina.

UFO je zároveň kompletní aplikace, takže nabízí dobrou příležitost vysvětlit ukládání aplikací do JAR souborů a demonstrovat, jak je takto uložený program samostatně spustitelný na jiných počítačích s instalovaným JRE. Jsou-li k dispozici počítače pracující na různých operačních systémech, je možno navíc předvést nezávislost vytvořeného na použitém operačním systému.

### 6.3 Vstupy a výstupy

*BlueJ* umožňuje obejít se v řadě situací bez znalosti práce se vstupy a výstupy, protože uživatel může zadávat příslušné hodnoty jako parametry v dialogových oknech otvíraných při volání metod. Nicméně v řadě úloh je takovýto způsob zadávání hodnot nevhodný.

Základní projekt s grafickými tvary proto obsahuje třídu `P`, která je pomocnou knihovni tříd obsahující základní sadu užitečných metod, mezi nimi i velice jednoduché metody pro okenní vstup a výstup. Před dalšími tématy proto v tuto chvíli studenti s těmito metodami seznámím a naučím je dané metody používat.



**Obr. 3:** Dialogová okna pro jednoduchý okenní vstup a výstup

Vzápětí si ale ukážeme i možnosti tisku na standardní výstup a standardní chybový výstup, protože ty jsou v řadě případů výhodnější než zobrazení výsledků v sérii dialogových oken. Standardní vstup v úvodním kurzu vynechávám.

#### 6.4 Návrhové vzory Přpravka (Messenger), Tovární metoda a Jedináček, výčtové typy

Na závěr první etapy, v níž se studenti učí vytvářet nové třídy a definovat jejich konstruktory, atributy a metody, si povíme o návrhových vzorech a seznámíme se s prvními z nich. Vysvětlíme si, jak lze definovat třídu, jejíž instance bude jedináček, a ukážeme si i základní využití továrních metod. S implementací obou vzorů se studenti již setkali, takže jim nebudou připadat nijak abstraktní.

Dalším z probraných vzorů je Přpravka označovaná v [11] jako Messenger. Studenti se naučí jak s pomocí tohoto vzoru jednoduše obejít nemožnost vrácení několika hodnot současně a definují tři třídy, které tento vzor implementují.

Součástí bloku týkajícího se návrhových vzorů je i seznámení s objektovými výčtovými typy. Studenti se dozvědí nutná syntaktická pravidla pro jejich definici a naučí se tyto typy navrhovat a používat.

#### 6.5 Aktivní plátno, Přesouvač, Kompresor a Výtah, návrhový vzor Služebník

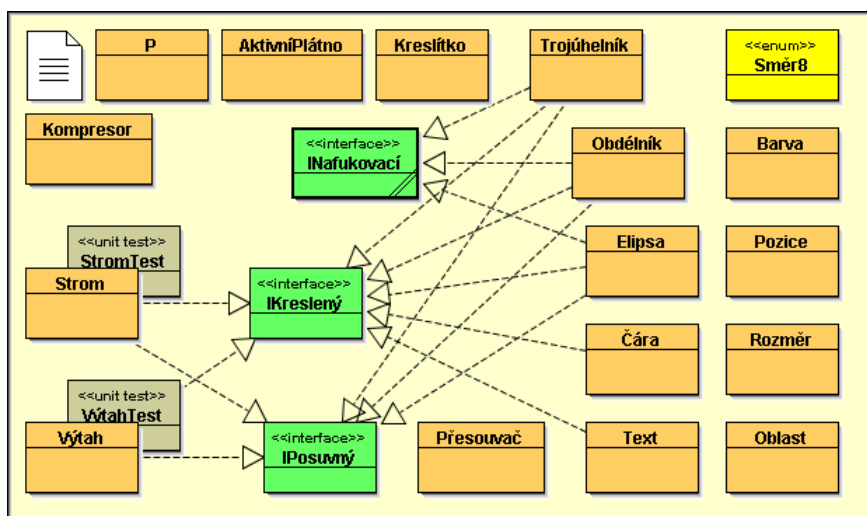
Po zvládnutí první etapy se v další etapě začnou studenti seznamovat s polymorfismem a dědičností. Vrátime se opět k základním geometrickým tvarům, ale nahradíme třídu `Plátno` třídou `AktivníPlátno`, jejíž instance (jedináček) se chová jako manažer, který dostane do správy obrazce, jež se mají zobrazit na plátně, a postará se o to, aby se tyto obrazce vždy správně vykreslily a aby se při nejrůznějších posunech a dalších aktivitách vzájemně nepřemazávaly. Na obrazce přebírané do správy klade jedinou podmínku: musí se umět nakreslit zadaným kreslítkem, tj. musí implementovat rozhraní `IKreslený`.

Při přechodu od původní verze plátna k aktivnímu plátnu se změni požadavky na třídy, které chtějí být vykreslovány na plátně. Změny v předpřipravených třídách jsou již v novém projektu připraveny, avšak změny v třídě `strom`, kterou si studenti zkopírují z minulého projektu, projdeme krok za krokem spolu. Studenti se tak učí, jak postupovat v situaci, kdy se změni podmínky, které má jejich třída splňovat.

V tomto projektu také studentům vysvětlím základy refaktorování a vyzkoušíme si je na jedné úpravě třídy `strom`, která je na jednorázové řešení příliš složitá.

Poté studenty seznámím se vzorem Služebník a předvedu jim třídu `Presouvac`, která je schopna přesouvat instance rozhraní `IPosuvny` po plátně. Na příkladu této třídy se naučí, jak specifikovat požadavky na rozhraní na základě požadavků na spektrum funkcí služebníka. Vzápětí si vše vyzkouší na definici třídy `Kompresor`, jejíž instance změní velikost zadaného parametru, jenž musí implementovat rozhraní `INafukovací`.

Partii zabývající se rozhraními zakončuje samostatný příklad, v němž mají studenti za úkol naprogramovat třídu `Vytah`, která má na aktivním plátně simulovat provoz výtahu. Výtah musí být schopen dojet pro zadaného pasažéra, nechat jej nastoupit, převést jej do jiného patra a tam jej nechat vystoupit.



Obr. 4: Diagram tříd závěrečné verze projektu Rozhrani

## 6.6 Balíčky

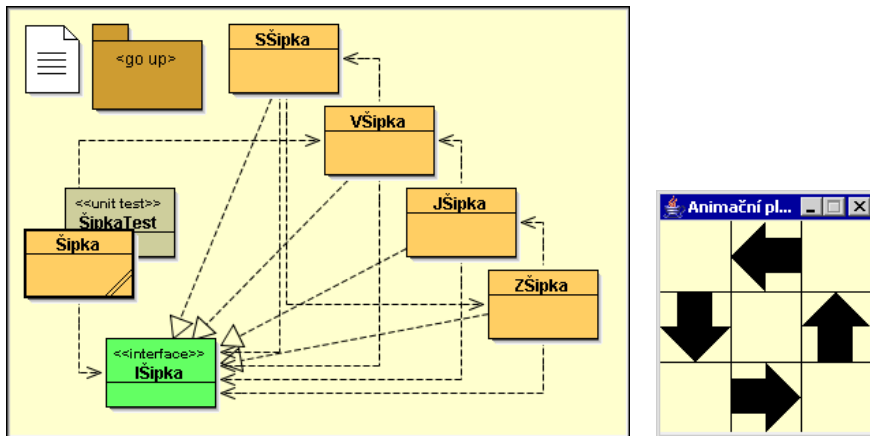
Na konci pasáže o rozhraních již začne být projekt dostatečně složitý, aby mělo smysl začít uvažovat o jeho rozdělení do několika balíčků. Vysvětlíme si proto výhody takového rozdělení o problémy, s nimiž se budou studenti potýkat při práci s balíčky v prostředí *BlueJ*.

Ve všech dalších projektech již budou třídy týkající se práce se základními geometrickými tvary umístěny v samostatném balíčku. Většina úloh pak bude umístěna v samostatných balíčcích a do balíčku s geometrickými tvary se vrátíme pouze tehdy, budem-li chtít jeho třídy doplnit nebo vylepšit.

## 6.7 Šipky a návrhový vzor Stav

Při použití služebníků se objeví potřeba, aby parametr implementoval více rozhraní současně. Vysvětlíme si proto dědičnost rozhraní a ukážeme si, jak s její pomocí uvedený problém vyřešit.

Poté probereme návrhový vzor Stav a ukážeme si jeho aplikaci na příkladu šipek, jejichž chování závisí na směru, do nějž jsou natočeny. Protože jsme v tuto chvíli ještě neprobrali podmíněné příkazy ani přepínače, je aplikace návrhového vzoru Stav jedinou možností, jak úlohu vyřešit.

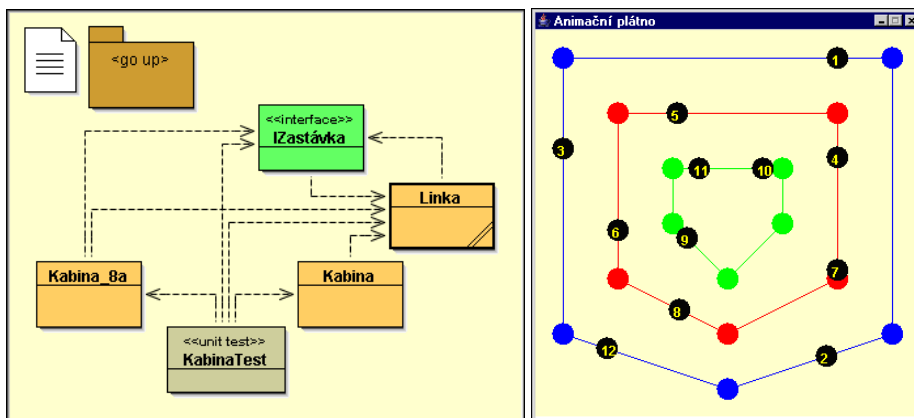


Obr. 5: Diagram tříd a aplikační okno projektu Šipky

### 6.8 Multipřesouvač a Kabina, návrhový vzor Zástupce

Pasáž o dědičnosti rozhraní zakončujeme úlohou, při níž mají studenti za úkol naprogramovat třídu `Kabina`, jejíž instance představují kabiny pohybující se po uzavřených linkách. Při té příležitosti se seznámíme s návrhovým vzorem Zástupce a ukážeme si, jak její třída `Linka` aplikuje při předávání odkazu na zastávky.

Poté jim představím třídu `Multipřesouvač`, jejíž instance (opět jedináček) je schopna přesouvat více objektů současně a u objektů implementujících rozhraní `IMultiposuvný` navíc zavolat po dovezení instance do požadované pozice její metodu `přesunuto()`, ve které si instance může např. vybrat nový cíl a nechat se do něj opět přesunout.



Obr. 6: Diagram tříd a aplikační okno projektu Kabina

Takto mohou studenti s minimálním úsilím vytvořit aplikaci, která viditelně provádí několik úloh současně a dále si upevní své povědomí o možnostech událostmi řízeného programování, s nimiž se setkali již při tvorbě třídy `UFO` a následně pak při přípravě tříd spolupracujících s aktivním plátnem.

## 6.9 Čtverec, Kruh, XObdélník, Terč – vhodnost špatných příkladů

Po dědičnosti rozhraní se začneme věnovat dědičnosti tříd. Nejprve si na malém ilustračním projektu vysvětlíme její základní principy a pak odvodíme třídu `čtverec` jako potomka třídy `obdélník`<sup>1</sup>. Ukážeme si přitom, jak malé úsilí stačí k návrhu nové plnohodnotné třídy. Studenti si pak sami obdobně odvodí třídu `kruh` jako potomka třídy `Elipsa`.

Při té příležitosti si vysvětlíme, že principem odvození dceřiné třídy není převzetí atributů a metod od rodiče, ale definice podmnožiny instancí, které mají oproti ostatním instancím rodičovské třídy některé speciální vlastnosti. Současně si ukážeme příklady špatně definované dědičnosti.

V následujícím výkladu odvodíme od třídy `obdélník` třídu `xobdélník` představující obdélník přeškrtnutý svými úhlopříčkami, který má navíc jiný referenční bod, vůči němuž se počítají jeho souřadnice. Při jeho vývoji si ukážeme řadu rysů dědičnosti, na něž je třeba při vývoji dceřiných i rodičovských tříd pamatovat. Takto poučení pak mají studenti za úkol samostatně odvodit od třídy `kruh` třídu `terč`, jejíž instance zobrazí na plátně tři soustředné kruhy přeškrtnuté záměrným křížem.

Na konci kapitoly se sice studenti dozvědí, že tyto třídy jsme vůbec neměli definovat jako potomky, ale měli jsme raději využít skládání, nicméně k předvedení záluďnosti dědičnosti tříd slouží dobře. Vyzkoušel jsem, že dokud si studenti tyto záluďné vlastnosti dědičnosti neosahají, neuloží je do paměti dostatečně pevně. Pouhá zmínka o tom, že si při tvorbě rodičovských a dceřiných tříd mají dát na to či ono pozor, k dostatečnému upevnění potřebných poznatků nestačí.

## 6.10 Zpětná kabina

Po dokončení zápasu s `xobdélník` a `terč` se vracíme k projektu s kabinami a od třídy `kabina` odvodíme třídu `zpětnáKabina`, jejíž instance mají trochu jiný tvar a navíc jezdí po linkách v opačném směru.

Projekt slouží k odhalení dalších nebezpečných vlastností dědičnosti, konkrétně nutnosti zákazu použití překrytných metod v definici konstruktoru. Ukážeme si, jak se s tímto zákazem vypořádat a přitom se seznámíme s konečnými třídami a metodami a znovu si připomeneme výhodné vlastnosti továrních metod.

---

<sup>1</sup> Definice potomka, který nemůže plnohodnotně reagovat na všechny zprávy, sice není zcela optimální, avšak na druhou stranu nabízí možnost záluďnosti tohoto druhu potomků dostatečně zřetelně převést a ukázat různé možnosti jejich řešení.

## 6.11 Dokončení návrhu knihovny tvarů

Výklad dědičnosti tříd pokračuje definicí společného rodiče skupiny tříd, které mají společné vlastnosti. Domnívám se totiž, že vzhledem k tomu, že při definici rodiče je třeba používat jiné myšlenkové postupy než při definici dceřiných tříd, je vhodné probrat definice společných rodičů skupiny evidentně příbuzných tříd samostatně a upozornit na její specifika.

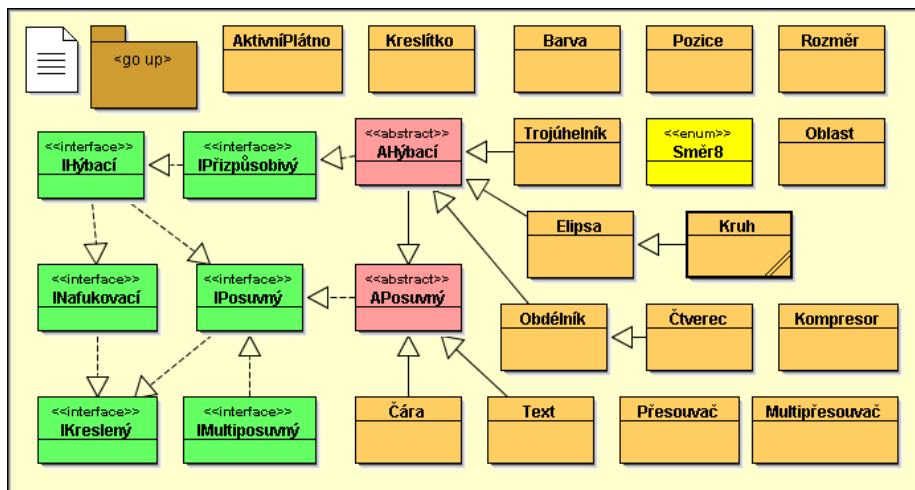
Vrátíme se proto do balíčku se základními geometrickými tvary, kde definujeme nejprve třídu **Posuvný** jako společného rodiče všech tříd, jejichž instance jsou schopny posunu, a poté studenti samostatně definují třídu **Hýbací** jako společného rodiče tříd, jejichž instance jsou schopny jak posunu, tak změny své velikosti.

Definice společných rodičů zákonitě vyústí v potřebu zavedení abstraktních tříd. Vysvětlíme si jejich podstatu a vlastnosti a předvedeme si, co vše získáme tím, že společné rodičovské třídy, které doposud nevyhovovaly zcela našim požadavkům, převedeme na abstraktní.

Na závěr si pak ukážeme, jak lze jednoduše změnit vlastnosti celé větve dědičného stromu. Definujeme rozhraní **IPřizpůsobivý**, které budou implementovat třídy, jejichž instance se budou umět přihlásit jako posluchači aktivního plátna, aby mohly zareagovat na změnu velikosti jeho pole a přizpůsobit se jí.

Ukážeme si, jak prostou implementací tohoto rozhraní rodičovskou třídou získají požadovanou vlastnost všechny její dceřiné třídy aniž bychom museli jakkoliv měnit jejich kód – stačí je pouze přeložit.

Na závěr těchto úprav prohlásíme balíček za dovedený do konečného stavu a ukážeme si, jak jej lze jednoduše definovat jako knihovnu, kterou budeme moci používat v dalších projektech.



Obr. 7: Diagram tříd závěrečné verze balíčku `rup.česky.tvary`

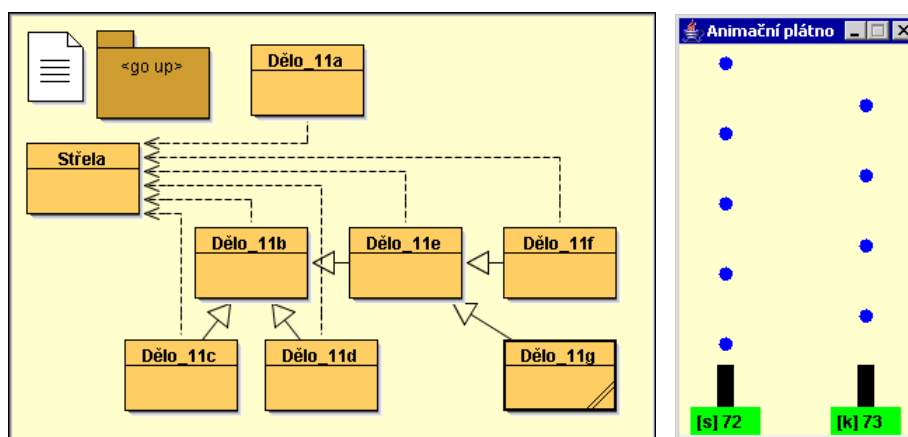
## 6.12 Střelba, návrhový vzor Adaptér

Uzavření dědičnosti tříd přechází výklad do své závěrečné fáze, v níž probíráme klasické strukturované konstrukce a kontejnery.

Začínám výkladem podmíněného příkazu, jehož použití demonstruji na ovládání klávesnice, které používáme při v definicích postupně vylepšovaných tříd, jež mají sloužit jako základ budoucí hry, v níž budou děla sestřelovat nalétávající letadla.

Při výkladu ošetřování klávesnice zároveň probereme návrhový vzor Adaptér, jehož použití standardní knihovna při práci s ošetřením událostí klávesnice nabízí.

Diagram tříd na obr. 8 ukazuje posloupnost tříd, které vznikají při postupném zdokonalování našeho původního návrhu. Takováto posloupnost vzniká při mém výkladu poměrně často. Používám ji proto, abychom se mohli kdykoliv vrátit k minulému stavu projektu a připomenout si, které úpravy jsme dělali a proč.



Obr. 8: Diagram tříd a aplikační okno projektu Děla

## 6.13 Balónky

Po podmíněných příkazech nastupuje výklad cyklů, jejichž použití si demonstrujeme na simulaci padajících balónek. Studenti programují simulaci volného pádu balónku jako cyklus jeho postupných překreslování v nových pozicích. Po úspěšné realizaci pádu naprogramují i jeho odraz a poté odraze s postupně se zmenšující amplitudou odskoku.

Při programování simulace balónek se sice studenti naučí pracovat s cykly, ale vlastní úloha působí nedotaženě, protože případný druhý balónek musí se svým pádem čekat, až doskáče jeho předchůdce. Na závěr výkladu o cyklech se proto studenti seznámí se základními informacemi o vláknech a naučí se naprogramovat pohybující se balónky tak, aby byly schopny skákat všechny současně.

## 6.14 Molekuly, návrhový vzor Iterátor

Po výkladu cyklů pokračujeme dynamickými kontejnery, konkrétně množinami a po nich seznamy. Použití množin demonstrujeme na simulaci náhodného pohybu skupiny molekul v uzavřeném prostoru.

Animace molekul způsobem použitým v předchozím příkladu při animaci balónek je při větším počtu molekul velmi neefektivní. Ukážeme si proto, jak lze tuto animaci zefektivnit, a pro animaci definujeme speciální třídu **Animátor**.

Následně si ukážeme, že kvůli animátoru musí třída **Molekula** zmírnit přístupová práva k některým svým atributům. Seznámíme se proto s možností vnoření tříd a vysvětlíme si, jak je možno použít vnoření třídy animátoru k zachování zapouzdření implementace třídy **Molekula**.

Při generování skupiny náhodně rozmístěných molekul i při následné simulaci jejich pohybu vystačíme s novou podobou cyklu `for` zavedenou v Javě 5.0. Abychom si ukázali i situace, kdy s tímto cyklem nevystačíme a musíme použít klasický cyklus s iterátorem, rozšíříme zadání o vývěvu, jež bude pohlcovat molekuly, které se dostanou do jejího dosahu. Vysvětlíme si proto funkci návrhového vzoru Iterátor a definujeme upravenou třídu **Molekula** s vnořenou třídou **Animátor**, které budou simulovat pohyb molekul s prostorem s vývěvou, která každou molekulu, která se dostane do jejího dosahu, z tohoto prostoru (a tím i z příslušné množiny) odstraní.

V závěrečném projektu této části doplníme prostor s molekulami ještě o „porodnici“, která pokaždé, když se v její oblasti nenachází žádná molekula, vygeneruje novou molekulu. V prostoru se tak nastaví dynamická rovnováha, při které molekuly stále mizí ve vývěvě a na druhou stranu se stále rodí v porodnici.



**Obr. 9:** Aplikační okno prostoru s molekulami, vývěvou a porodnicí

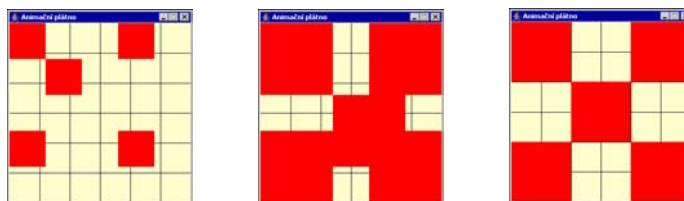
## 6.15 Mnohotvar

Použití seznamů si demonstrujeme při vývoji třídy **Mnohotvar**, jejíž instance představují grafické útvary, které mohou sestávat z několika jednodušších tvarů, přičemž po těchto jednodušších tvarech požadujeme pouze, aby implementovali rozhraní **IHýbací**. Mohou to tedy být jak základní primitivní tvary, tak nějaký ze složených tvarů, které jsme vytvářeli na počátku celého kurzu, anebo to může být dokonce i jiný mnohotvar.

Na příkladu mnohotvaru si ukazujeme, jak může být občas výhodné definovat jako vnořenou třídu soukromou přepravku pro skupinu údajů, které si daná instance musí pamatovat. Vlastní seznam pak definovat ne jako seznam ukládaných instancí

(v našem případě jednoduchých tvarů), ale jako seznam instancí vnořené přepravky, v nichž si může instance vnější třídy pamatovat kromě odkazu na ukládanou instanci i potřebné pomocné údaje.

Na obrázku 10 si můžete prohlédnout, jak vypadá pravidelný mnohotvar složený z pěti čtverců po výrazném zmenšení a následném zvětšení. Levý obrázek ukazuje zvětšený mnohotvar nepoužívající zaokrouhlování vypočtených souřadnic, ale pouze jejich ořezávání. Prostřední používá zaokrouhlování a pravý ukládá místo pouhých jednoduchých tvarů celou přepravku i s některými pomocnými údaji.



**Obr. 10:** Aplikační okna jednotlivých verzí Mnohotvaru po testu změny velikosti

### 6.16 Vyjádření čísel slovy a Pascalovy trojúhelníky

Práci s jednorozměrnými poli demonstrujeme na příkladu převedení čísla na jeho textovou podobu uváděnou při vyjádření čísla slovy – např. 213 = dvě stě třináct.

Práci s vícerozměrnými poli si pak vysvětlujeme při definici třídy `Pascal`, jejímiž instancemi jsou Pascalovy trojúhelníky zadané velikosti.

### 6.17 Roboti

V kroužku se v závěru první etapy výkladu stala velice oblíbenou soutěž o vytvoření nejschopnějšího robota. Každý člen má při ní za úkol definovat třídu, jejíž instance představují roboty, kteří jsou schopni pobíhat po dvorku a sbírat značky.

Na hodině pak vypouštíme roboty na společná dvorek ve skupinách po čtyřech a tvůrci tříd soutěží, čím robot nasbírá v daném časovém limitu více značek a nejlépe znemožní sbírání značek ostatním přítomným robotům.

Přiznám se, že v závěru kurzů již většinou není na podobné hraní si čas, takže tuto aplikaci na kurzech většinou vynecháváme.

### 6.18 Displej



**Obr. 11:** Aplikační okno projektu Displej

Na závěr celého kurzu si předvedeme vývoj složitější aplikace sestávající ze tří navzájem propojených tříd, které dohromady realizují simulaci sedmisegmentového displeje.

je. Vývoj této aplikace je vlastně přípravou na druhý běh kurzu, ve kterém se studenti učí vyvíjet složitější programy a současně se dozvědí řadu dalších informací o technologii – probereme výjimky, vlákna, práci se soubory a reflexi, podrobně pohovoříme o knihovně kontejnerů a povíme si i řadu podrobností o vnitřní reprezentaci výčtových typů a o práci s parametrizovanými datovými typy. Současně si ukážeme základy tvorby GUI.

## 7 Porovnání uvedené metodiky s metodikou prezentovanou v [9]

Tuto pasáž neberte jako kritiku přístupu srovnávané publikace, ale spíše jako představení dvou různých koncepcí a námět k zamyšlení nad tím, jaké uspořádání výkladu je při výuce vstupních kurzů programování výhodnější.

Autoři knihy [9] začínají stejně jako já (přesněji řečeno já začínám stejně jako oni), tj. experimenty s třídami a jejich instancemi následovanými vytvořením jednoduché třídy. Hned od první vytvářené třídy se však odpoutají od práce s grafickými objekty a snaží se vytvářet jakoby praktické aplikace. Já se naopak snažím v celém vstupním kurzu stavět na grafických příkladech, protože mi připadají daleko názornější a lépe smyslově uchopitelné. Kromě toho se domnívám, že studenti mají v této etapě dostatek starostí s pochopením a přijmutím základů OOP a nechci jim proto v každé lekci předkládat programy a úlohy pracující ve zcela jiném prostředí.

Autoři [9] zaměřují svůj další výklad na některé zásady návrhu složitějších objektových programů, jakými jsou např. provázanost (coupling) a soudržnost (cohesion) tříd a metod (těmto otázkám se já věnuji až v druhém kurzu, a proto zde nebyly zmíněny). Při jejich výkladu tak mohou používat pouze vzájemnou závislost tříd do které nezasahuje dědičnost ani implementace rozhraní. Tím se výklad zjednodušuje.

Moje zkušenosti však naznačují, že návrh složitějších aplikací, v nichž je třeba dbát na minimální provázanost a maximální soudržnost tříd a metod, je vhodné přesunout až do pozdějších fází výkladu, kdy mají studenti základní konstrukce zažité a mohou se proto od nich odpoutat a soustředit se na architekturu projektu. Proto ve svých kurzech seznamuji nejprve s rozhraními, pak ještě chvíli s rozhraními a poté s dědičností tříd. Návrh složitějších projektů a s ním související témata odkládám až do druhého kurzu.

Výhodou takového uspořádání je navíc to, že studenti se mohou při procházení vzorových řešení v prvním kurzu setkat s množstvím správně (doufejme) koncipovaného kódu a řada zásad jim tak alespoň částečně přejde do krve i bez jejich předběžného teoretického zdůvodnění. Navíc si odložením tohoto výkladu udělám prostor, který mohu věnovat výkladu jednodušších návrhových vzorů, o nichž se domnívám, že napomáhají lepšímu pochopení objektového paradigmatu.

Takovéto uspořádání je nezbytností zejména v kroužcích navštěvovaných dětmi, u nichž je návrh struktury složitějších programů tím největším problémem. Po zkušenostech z kroužků jsem však udělal pár opatrných sond mezi profesionálními programátory a zjistil jsem, že v jejich kurzech je potřeba takového uspořádání možná méně do očí bijící, avšak o nic méně důležitá.

S dědičností tříd seznamuje [9] až přibližně v polovině výkladu a s rozhraními dokonce až ve třech čtvrtinách. (Navíc je rozhraní prezentováno nejprve jako náhražka

násobné dědičnosti tříd – to jsem tedy opravdu zamrkal.) Takto pozdní umístění vede k tomu, že autorům nezbyvá prostor na podrobný výklad možností, výhodných vlastností a nebezpečných rysů obou konstrukcí.

Moje zkušenosti naznačují, že výklad rozhraní je třeba umístit výrazně před výklad dědičnosti tříd, protože jenom tak je možno objasnit účel a funkci rozhraní dostatečně pregnantně bez rušivých vlivů alternativních možností. Domnívám se také, že před přistoupením k výkladu dědičnosti tříd by měli mít studenti používání rozhraní nejprve dostatečně zažitě.

Kromě toho se domnívám, že při výkladu dědičnosti tříd je třeba zdůraznit nejenom nejčastější chyby při jejich použití v návrhu (např. že dceřiná třída není skutečným podtypem rodičovské třídy), ale také řadu dalších záležitostí dědičnosti, jakými jsou např. narušení zapouzdření, zákaz používání překrytných metod v konstruktorech apod.

Na přístupu [9] se mi nelíbí také to, že výklad podmíněných příkazů a cyklů proběhne někde v první třetině výkladu jakoby mimochodem při řešení úloh. Možná jsem odchovanec staré školy, ale zdá se mi, že podrobnému výkladu těchto konstrukcí je třeba věnovat téměř stejnou pozornost jako výkladu konstrukcí objektových.

## 8 Závěr

Jak jsem již řekl, uvedenou metodiku používám jak v kurzech pro profesionální programátory, tak v kroužcích navštěvovaných dětmi ze základních a středních škol. V kroužcích si nové nápady vždy vyzkouším a ověřím, a ty, které se osvědčí aplikuji v kurzech.

Použití této metodiky se osvědčilo nejenom při výuce naprostých začátečníků, ale i při doškolení a přeškolení profesionálních programátorů, kteří mají bohaté zkušenosti ze strukturovaného programování v klasických či skriptovacích jazycích. Díky ní se daří frekventanty těchto kurzů nejenom naučit syntaktická a sémantická pravidla jazyka, ale vštípit jim zároveň i základy objektově orientovaného uvažování. Frekventanti nyní po absolvování kurzů vědí nejenom jak objektové konstrukce použít, ale také kdy po nich při návrhu architektury programu sáhnout.

Při prezentaci [8] jeden z oponentů namítal, že výklad podle této metodiky není možné stihnout ve vyhrazeném čase. Upřesnil bych proto, že v dětských kroužcích s jednou 90minutovou lekcí týdně probereme uvedenou látku za necelý rok. Kurzy pro ty, kteří ještě nikdy neprogramovali, trvají 8 pracovních dnů, kurzy pro programátory přecházející na Javu z jiných programovacích jazyků jsou pětidenní.

## Reference

1. Webová stránka prostředí BlueJ: <http://www.bluej.org>
2. Odborné články a publikace o prostředí BlueJ a metodice výuky programování: <http://www.bluej.org/about/papers.html>

3. Jirka Hradil blog, červen 01. 2004: Školení v Javě – rychlý start pro začátečníky? [http://www.hradil.cz/2004/06/kolen-v-jav-rychl-start-pro-zatenky\\_01.html](http://www.hradil.cz/2004/06/kolen-v-jav-rychl-start-pro-zatenky_01.html)
4. Pecinovský R.: *Myslíme objektivě v jazyku Java 5.0*, Grada, 2004. ISBN 80-247-0941-4.
5. Van Haaster, K. and Hagan, D.: Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool. *Information Science + Information Technology Education Joint Conference*. Rockhampton, QLD, Australia, June 2004.
6. Kölling, M., Quig, B., Patterson, A. and Rosenberg, J.: The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, No 4*. Dec 2003.
7. Patterson, A., Kölling, M. and Rosenberg, J.: Introducing Unit Testing With BlueJ. *Proceedings of the 8th conference on Information Technology in Computer Science Education (ITiCSE 2003)*. Thessaloniki, 2003.
8. Pecinovský R.: Výuka objektivě orientovaného programování žáků základních a středních škol. *Objekty 2003 – Sborník příspěvků osmého ročníku konference*. Ostrava 2003. ISBN 80-248-0274-0.
9. Barnes D. J., Kölling M. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2002, ISBN: 0-13-044929-6.
10. Nourie, D.: Teaching Java Technology With BlueJ. Online article at <http://java.sun.com/features/2002/07/bluej.html>, July 2002.
11. Eckel B.: Thinking in Patterns. Electronic book at <http://64.78.49.204>.
12. Eckel B.: *Thinking in Java 2nd Edition*. Prentice Hall, 2000. Český překlad vyšel ve dvou dílech: *Myslíme v jazyku Java – knihovna programátora*. Grada, 2000. ISBN 80-247-9010-6 a *Myslíme v jazyku Java – knihovna zkušeného programátora*. Grada, 2000. ISBN 80-247-0027-1.
13. Kölling, M.: Teaching Object Orientation with the Blue Environment. *Journal of Object-Oriented Programming, Vol. 12 No. 2, 14-23*. 1999.
14. Kölling, M.: *The Design of an Object-Oriented Environment and Language for Teaching*. PhD Thesis, Basser Department of Computer Science, University of Sydney, 1999.

## Annotation

### *How to Really Start With Objects by Teaching Java*

The paper introduces such methodology of teaching programming, where students work with object from very beginning and not after some prologue which deals with classical programming structures. According to this methodology, classical structures are presented in the last third or quarter of the course. The paper explains disadvantages of today's standard methodology and it shows how it can be changed by usage of the IDE *BlueJ*. It demonstrates how the examples should be chosen and it presents the two years experience from the professional courses as well as from children programming clubs.