Začlenění návrhových vzorů do výuky programování

Rudolf Pecinovský[1]

[1]Amaio Technologies Inc., 100 00 Praha 10, Třebohostická 14
rudolf@pecinovsky.cz

Abstrakt. Příspěvek nejprve stručně seznamuje se šesti hlavními současnými přístupy k výuce základů programování. Poté se soustředí na přístup, při němž celá výuka začíná prací s objekty a důslednou aplikací zásad objektově orientovaného programování. Kritizuje polovičatost doposud publikovaných učebních textů a naznačuje, jak je možno deklarované zásady aplikovat důsledněji. V druhé, klíčové části příspěvku ukazuje na konkrétních příkladech, jak lze postupovat při výuce, při které se od počátku učí doopravdy objektově orientované programování a předvádí, jak je možno prakticky od prvních hodin začlenit do výuky programování seznámení s návrhovými vzory a možnostmi jejich použití. V třetí části srovnává dva přístupy k zadání zkušebního příkladu: klasický a skutečně objektově orientovaný. Ukazuje, že opravdu objektový přístup je výhodnější jak pro studenty, tak pro vyučujícího. V poslední, čtvrté části pak zmiňuje návrhové vzory, které by si také zasloužily zařazení do vstupních kurzů, avšak v předchozím textu na ně nedošlo.

Klíčová slova: OOP, návrhové vzory, výuka programování, vstupní kurzy programování, přístup „nejdříve objekty“, přístup „object first“

1 Úvod

Stejně jako jiné oblasti činnosti, i programování se neustále vyvíjí. Bohužel, metodika výuky programování nesleduje tyto trendy vždy dostatečně dobře a většinou se za nimi výrazně opožďuje. Učitelé proto často připravují žáky na styl programování, který byl progresivní před 15 až 25 lety, ale v současné době je již dávno překonaný. O to překonanější bude v době, kdy budou jejich studenti vstupovat do praxe.

1.1 Přehled používaných přístupů k výuce

V současné době se na různých místech učí programování podle různých metodik, které jsou rozdělovány do následujících šesti skupin:

Nejdříve hardware (Hardware-first)
Zastánci tohoto přístupu 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 realizačního 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 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.

Jednou z velkých nevýhod takto koncipovaných kurzů je pak to, že se jedná především o kurzy syntaxe a nikoliv o kurzy programování. Vyučující přednáší a cvičí tak, jakoby předpokládali, že umění programovat přijde se znalostí syntaxe jako vedlejší efekt. Nepřijde.

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 studenti si pak někdy připadají 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 je proto třeba koncipovat tak, aby k tomuto zahlcení nedošlo.

Nejdříve zeširoka (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 výchozími znalostmi studentů.

1.2 Současná převažující podoba přístupu „Nejdříve objekty“

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 výuka práce s objekty probírá až v závěru kurzu, nepřináší zdaleka tak dobré výsledky jako výuka, při které výuky prací s objekty naopak začíná. Postupně se proto rozšiřují řady vyučujících, kteří se snaží koncipovat svoji výuku tak, aby se jejich žáci na počátku výuky s objekty nejenom seznámili, ale aby s nimi od samého začátku výuky také pracovali.

Typickým, a pravděpodobně také nejznámějším představitelem výukového textu připraveného pro tento typ výuky je [1]. Přiznejme si však, že i tato kniha řeší problém pouze částečně. Žáci sice od samého začátku pracují s objekty a navrhují programy, v nichž se učí definovat třídy tak, aby byly jednoduché, kompaktní a minimálně provázané, ale zůstávají pouze u definice jednoduchých tříd. S existencí rozhraní se seznámí až v závěru kurzu a navíc jsou zde rozhraní prezentována do jisté míry jako náhražka násobné dědičnosti. O návrhových vzorech, které jsou považovány za jeden z pilířů současného programování, nepadne ani slovo. Stejně tak autoři nijak explicitně nezdůrazňují základní pravidlo, že programovat se má proti rozhraní tříd a ne proti jejich implementaci. Pak by totiž museli rozhraní jako druh datového typu prezentovat zcela jinak a především daleko dříve.

V [1] i v nejrůznějších dalších „Object-First kurzech“ však bývá opominuta řada neméně důležitých zásad, které by bylo třeba žákům vštěpovat již od samého začátku výuky. Řada těchto zásad je poměrně pregnantně vysvětlena např. v [4], avšak tato učebnice předpokládá, že čtenář již prošel základním kurzem jazyka Java. Při jejím pročítání si však bystrý čtenář musí uvědomit, že výklad řady z vysvětlovaných zásad již mohl a měl být součástí vstupního kurzu programování.

1.3 Snahy o zavedení návrhových vzorů do výuky – killer examples

Základním problémem současné metodiky (alespoň jak jej mnozí cítí) je nedostatek názorných příkladů, na kterých by bylo možno již v začátečnických kurzech demonstrovat současné trendy a především pak použití návrhových vzorů. Na přelomu století se proto skupina vysokoškolských učitelů dohodla, že budou pořádat pravidelné semináře nazvané *„Killer Examples“ for Design Patterns and Objects First*, na kterých se pokusí představit tzv. „killer examples“, což by měly být právě příklady, které jsou na jednu stranu dostatečně jednouché, aby je bylo možno použít i ve vstupních kurzech programování, avšak na druhou stranu budou dostatečně „složité“, aby se na nich dalo přirozeně demonstrovat použití návrhových vzorů.

Když jsem procházel příklady prezentovanými na těchto seminářích, připadala mi většina z nich pro vstupní kurzy s minimální hodinovou dotací stále příliš (možná bych mohl říci zbytečně) složitá. Zdá se mi, že ani po těch čtyřech letech, po něž se tyto semináře konají, se nepodařilo vymyslet ty správné „killer examples“ (zatím mne nenapadl ten správný český překlad).

Ve svém příspěvku bych se proto pokusil předvést několik příkladů, které možná nebudou oněmi pravými „killer examples“, ale alespoň demonstrují, že návrhové vzory a takové principy, jako programování proti rozhraní a další, lze aplikovat i při řeše-

ní velice jednoduchých příkladů, které lze studentům vstupních kurzů programování předložit již na prvních cvičeních.

2 Návrhové vzory použité na samém počátku výuky

2.1 Rozdílná vstupní úroveň studentů

Jedním z problémů počátečních hodin výuky je rozdílná úroveň studentů. Někteří z nich se s programováním setkávají poprvé, jiní již mají za sebou řadu programů. Drobnou výhodou je, že převážná většina těch, kteří se považují za zkušené programátory, má zkušenosti pouze se strukturovaným programováním, protože objektově orientované programování se většinou na středních školách ani v zájmových kroužcích neučí. Pokud na ně tedy včas „vybafneme“ s dostatečně objektovými příklady, máme šanci relativní vstupní úroveň studentů ve vztahu k probírané látce částečně sjednotit.

Nutnost onoho „včasného vybafnutí“ je o to větší, že v opačném případě se studenti s předchozími neobjektovými programátorskými zkušenostmi v prvních hodinách často snaží znevažovat některé předváděné postupy, protože je ze své zkušenosti dokáží naprogramovat zdánlivě jednodušeji či efektivněji. Neuvědomují si však, že OOP vyžaduje přece jenom poněkud odlišný styl řešení problémů, a že to, co se nenaučí na jednoduchých příkladech, jim bude chybět při řešení příkladů složitějších.

O tom, jak se programátoři s předchozími neobjektovými zkušenostmi pokouší interpretovat přednášenou látku za pomoci svých dosavadních znalostí, jsem již hovořil v [8], [9] a [11]. V kurzech profesionálních programátorů jsem si ověřil, jak důležité je těmto programátorům co nejdříve „podříznout větev“ jejich dosavadních znalostí, na které „sedí“, a co nejdříve jim předložit příklady, na něž jejich dosavadní znalosti nestačí. Obdobný postup lze aplikovat i na studenty středních a vysokých škol.

2.2 Testovací třída

Na první hodině/cvičení se studenti většinou teprve seznamují s objekty a třídami, s vývojovým prostředím, které budou po zbytek kurzu používat. Prohlédnou si strukturu nějakého předem připraveného programu a ujasní si vzájemné závislosti a interakce jednotlivých tříd a jejich instancí. Protože je program předem připravený, nemusí být triviálně jednoduchý (alespoň z jejich pohledu). Při té příležitosti si studenti ujasní, že vše, co v programu vystupuje, je objekt, včetně toho, co by v běžném životě za objekt nepovažovali – např. vlastnosti jako je barva nebo 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.

Na vysokých školách se dá na tyto úvodní hrátky navázat ještě v prvním cvičení konstrukcí prázdné třídy, kterou postupně doplníme o konstruktor, jenž nakreslí některý z obrázků, které byly dříve kresleny interaktivně. Poté si předvedeme možnosti definice přetížených verzí konstruktorů a seznámíme se s klíčovým slovem **this**.

Pokračujeme definicí metod, které mají s obrázkem manipulovat (např. jej mají přesunout). Ukážeme si, že k takovýmto operacím potřebujeme zavést atributy a současně si předvedeme, jak se svými atributy pracují instance grafických tříd v používaném projektu.

2.3 Návrhový vzor *Knihovní třída* (*Library class*)

Definujeme si třídu `Světlo` a její metody `rozsviť()` a `zhasni()`. Pak si řekneme, že bychom mohli naučit naše světlo blikat. Seznámím je proto s pomocnou třídou `P` a její metodou `čekej(int)`, která na zadanou dobu zastaví provádění programu. Poté definujeme metodu `blikni()`, která světlo rozsvítí, chvíli počká, zhasne je a zase chvíli počká.

Při té příležitosti jim prozradím, že třída `P` nemá žádné instance, protože je její metody ke své činnosti nepotřebují. Všechny jsou proto definovány jako metody třídy (statické metody) a třída má svůj konstruktor definován jako soukromý, aby její instance vůbec vyrobit nešlo. Vysvětlíme si, že třída `P` je typickým reprezentantem knihovní třídy a zopakujeme si, jaké vlastnosti knihovní třída má.

2.4 Zavedení rozhraní

Jednou z možností, jak studentům co nejdříve představit úlohy, v jejichž řešení jim neobjektové zkušenosti příliš nepomohou, je vyložit co nejdříve vedle tříd také rozhraní a předkládat pak úlohy, při jejichž řešení je využijí.

V knize [7] jsem zaváděl rozhraní až po ukončeném výkladu základních vlastností tříd (atributy, metody, statické a nestatické členy, referenční a hodnotové datové typy atd.). Obdobnou posloupnost jsem prezentoval i v [8], [9], [10] a [11].

Postupem doby jsem ale dospěl k závěru, že to je příliš pozdě. Zkušenosti s kurzy dětí i dospělých ukázaly, že rozhraní je vhodné vyložit hned poté, co se žáci seznámí se základní syntaxí použitého programovacího jazyka. Stačí opravdu základní znalosti o definicích tříd, jejich konstruktorů, atributů a metod.

Seznámení s rozhraními jsem v současné době předřadil před výklad řady rysů tříd. Studenti se tak seznámí s tímto druhem datového typu na samém počátku kurzu a budou se s ním setkávat ve většině svých následných úloh.

Velkou výhodou včasného zavedení rozhraní je, že se pak daleko snáze vymýšlí automatizované kontroly odevzdaných úkolů. Příklady takovýchto automatizovaných kontrol si ještě ukážeme.

2.5 Návrhový vzor *Služebník*

V současné době seznamuji studenty s rozhraním hned po prvních definicích vlastních tříd. Protože v tuto chvíli ještě studenti neprobrali cyklus, navrhnu jim náhradní řešení, které bude využívat objektových možností jazyka.

Teoretický úvod

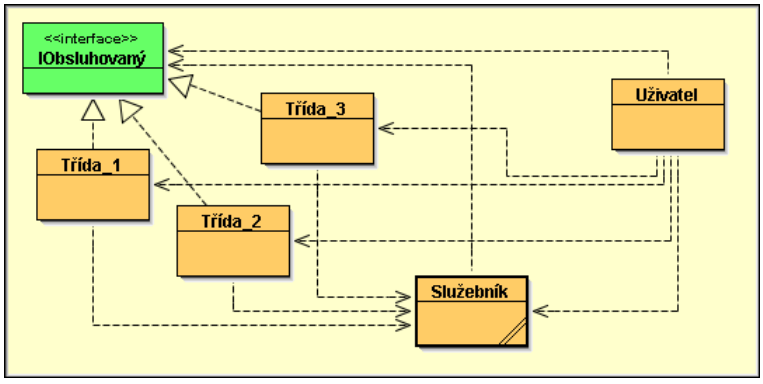
Seznámím je s návrhovým *Služebník* a prozradím jim, že tento vzor použijeme ve chvíli, kdy potřebujeme vybavit skupinu tříd novou schopností. Vysvětlím jim, že doplňovat kód do každé ze tříd nebývá optimální, protože jednou z důležitých programátorských zásad je, nepoužívat stejný kód na různých místech programu. Při jeho případné pozdější modifikaci (a je jedno, jestli jsme v něm našli chybu nebo jestli si zákazník objednal úpravu chování programu) si pak totiž nemusíme pamatovat, kde všude byl daný kód použit, a všechna místa navštívit a kód na nich shodně opravit. Stačí nám totiž opravit kód na jediném místě. Tím snížíme pracnost a naopak zvýšíme spolehlivost dané opravy.

Jednou z možností, jak tento problém řešit, je definovat třídu – služebníka, která bude obsahovat potřebné metody. V místě, kde mají naše instance provést požadovanou činnost, pak mohou pouze zavolat příslušnou metodu služebníka a předat mu obsluhovanou instanci jako parametr.

Aby mohl služebník danou instanci správně obsloužit, mívá většinou nějaké požadavky na její schopnosti. Vedle třídy služebníka proto definujeme také rozhraní (nazvěme jej pracovně `IObsluhovaný`), které požadované schopnosti blíže specifikuje. Služebník pak definuje metodu, jejímž parametrem je instance tohoto rozhraní.

Výhodou tohoto přístupu je také to, že nemusíme třídu služebníka spolu s potřebným rozhraním definovat sami, ale můžeme je získat od někoho jiného. To se nám hodí zejména tehdy, pokud víme o tom, kde hotové řešení seženeme, nebo pokud nedokážeme danou úlohu vyřešit sami a musíme si její řešení někde objednat.

Vysvětlíme si, že služebník může být sice definován jako knihovní třída, ale většinou bývá definován jako instance – často jako jedináček.



Obr. 1. Diagram tříd návrhového vzoru *Služebník*.

Realizace
Po tomto teoretickém úvodu nabídnu studentům hotovou třídu `Opakovač` s metodami `opakujKrát(int)` a `opakujVteřin(double)` doplněnou rozhraním `IAkce` (pro lepší orientaci studentů přidávám ke všem identifikátorům rozhraní standardně prefix `I`) vyžadujícím implementaci metody `akce()`.

Upravíme pak definici třídy `Světlo` tak, aby implementovala rozhraní `IAkce`, a doplníme metodu `akce()`. Ukážeme si, že tuto metodu lze definovat jednouše tak, že zavolá dříve definovanou metodu `blikni()`. Poté definujeme metodu `blikej(int)`.

Následně definujeme třídu `Přejezd` simulující výstražná světla na železničním přejezdu, která bude mít (mimo jiné) metodu `blikni()`, jež rozsvítí levé a zhasne pravé světlo, chvíli počká, zhasne levé a rozsvítí pravé světlo a opět chvíli počká.

Za domácí úkol dostanou studenti doplnit třídu `Přejezd` o schopnost dlouhodobého blikání a definovat vlastní třídu `Semafor`, která bude schopna simulovat standardní semafor řídící provoz na silnicích.

V profesionálních kurzech řeší převážná většina kurzantů úkol tak, jak je požadováno, tj. s využitím rozhraní. Mezi žáky v kroužcích i mezi studenty na VŠ se však vždy najde dost takových, kteří již vědí, jak se programuje cyklus, a rozhodnou se ušetřit si práci tím, že místo implementace rozhraní použijí rovnou cyklus. Na ty mám připraven malý podraz: v některé z dalších hodin rozšíříme schopnosti vytvořených přejezdů a semaforů zavedením dalších služebníků. Instance, které ve svých metodách nevyužívají opakovače, protože řeší opakování vlastním cyklem, nebudou schopny některé z nových akcí realizovat. Za chvíli se k těmto zadáním dostaneme.

2.6 Návrhový vzor *Přepravka* (*Messenger*)

Abychom si návrhový vzor *Pomocník* zopakovali a upevnili, doplníme doposud definované třídy o schopnost pohybu. Společně se zamyslíme nad množinou zpráv, na něž musí umět reagovat instance, které budou chtít využívat služebníka, jenž s nimi bude plynule pohybovat.

Při definici metod pro zjišťování a nastavování pozice narazíme na problém, že pozice je definována dvěma hodnotami, kdežto metoda smí vrátit pouze jednu hodnotu. Tento problém lze řešit dvěma způsoby: buď definovat několik metod, z nichž každá vrátí jednu z požadovaných hodnot, nebo definovat novou třídu, jejíž instance budou sloužit jako přepravky pro skupiny předávaných hodnot. Pro každou hodnotu bude definován jeden atribut, přičemž atributy přepravky budou (na rozdíl od běžné praxe) deklarovány jako veřejné.

Definujeme proto třídu `Pozice`, jejíž instance budou sloužit jako přepravky při předávání informací o pozicích různých objektů. Poté definujeme rozhraní `IPosuvný`, jehož instance budou implementovat metody `getPozice()` a `setPozice(Pozice)` a upravíme definici tříd `Světlo` a `Přejezd`, aby jejich instance byly posuvné.

Poté si studenti stáhnou předpřipravenou třídu `Přesouvač` a vyzkouší si, jak se jejich instance plynule přesouvají po plátně. Ti, kteří ve svých třídách použili při implementaci blikání opakovače, si mohou dokonce vyzkoušet, že jejich instance jsou schopny se posouvat a přitom blikat. Instance těch, kteří definovali blikání prostřednictvím cyklu, tuto schopnost ovládat nebudou.

2.7 `AktivníPlátno` a událostmi řízené programování

Již při počátečním hraní si s objekty jsme zjistili, že přesouvané objekty odmazávají objekty, které přes ně v jejich výchozí pozici přesahují. Při plynulém přesouvání pak odmazávají vše, přes co cestou přejedou. (Nejde o to, že bychom nemohli naprogramovat plátno tak, aby se objekty neodmazávaly, ale takto můžeme před studenty předestřít pádný důvod pro změnu koncepce zobrazování.)

Abychom tento nepříjemný stav odstranili, nahradíme dosavadní třídu **`Plátno`** představující pasivní plátno, na něž se všechny obrazce kreslí, třídou **`AktivníPlátno`**. Její instance však již není skutečným plátnem, na které se kreslí, ale pouze takovým manažerem, který rozhoduje o tom, kdy se má co na plátno nakreslit. Tato instance má skutečné plátno schované a zobrazení instance na tomto plátně zprostředkuje tak, že instanci poskytne kreslítko, jímž se daná instance na ono soukromé plátno nakreslí. Jinak, než dodaným kreslítkem se na toto plátno nic nakreslit nedá, takže tímto způsobem **`AktivníPlátno`** zabezpečí, že se na plátno nakreslí pouze ten obrazec, který samo určí, a v okamžiku, který určí.

Budeme-li chtít od této chvíle nějaký objekt nakreslit na plátno, musíme jej nejprve přihlásit u manažera – aktivního plátna, aby jej přidal mezi spravované objekty, o jejichž vykreslovaní se stará.

Aby bylo **`AktinvíPlátno`** ochotno přijmout někoho do své správy, musí o něm vědět, že se bude umět na požádání nakreslit dodaným kreslítkem. Vedle aktivního plátna je proto definováno ještě rozhraní **`IKreslený`**, jež vyžaduje implementaci metody **`nakresli(Kreslítko)`**, po jejímž zavolání se příslušná instance dodaným kreslítkem nakreslí. Metoda pro přidání dalšího objektu mezi spravované je pak deklarována **`nakresli(Kreslítko)`**.

Objekt, který bude svěřen do správy aktivního plátna, však nikdy nebude vědět, kdy bude o své vykreslení požádán. **`AktivníPlátno`** totiž spustí posloupnost překreslování pokaždé, když někdo vyvolá jeho metodu **`překresli()`**. Spravované objekty však dopředu nevědí, kdy to bude. V tuto chvíli se studenti poprvé setkávají s událostmi řízeným programem. Pokaždé, když se někdo domnívá, že se situace na plátně změnila (např. když se nějaký objekt posunul nebo změnil svoji velikost), požádá **`AktivníPlátno`**, aby vše překreslilo, a to pak oslovuje jednotlivé nakreslené objekty, předává jim aktuální kreslítko a žádá je, aby se překreslily.

2.8 Návrhový vzor *Pozorovatel* (*Observer*)

`AktivníPlátno` je vzorovou implementací návrhového vzoru *Pozorovatel*, a to hned dvakrát. O prvním významu jsem již hovořil. Kromě objektů, které chtějí být nakresleny na plátně, nabízí **`AktivníPlátno`** možnost evidovat tzv. přizpůsobivé objekty, tj. objekty, které chtějí mít svoji pozici i rozměr neustále v korelaci s velikostí kroku aktivního plátna.

`AktivníPlátno` totiž kreslí na plátno čtvercovou síť, která uživatelům umožňuje přesněji odhadnout pozice a rozměry nakreslených obrazců. V některých aplikacích je výhodné, aby se zobrazované objekty uměly přizpůsobit změně velikosti kroku aktivního plátna, tj. vzdálenosti čar této sítě.

Třídy, jejichž instance se mají přizpůsobovat změnám kroku aktivního plátna, musí implementovat rozhraní **`IPřizpůsobivý`**, jež vyžaduje od je implementujících tříd implementaci metody **`krokZměněn(int,int)`**. Přizpůsobivé instance opět vystupují jako pozorovatelé, kteří se zavoláním metody **`přidejPřizpůsobivý(IPřizpůsobivý)`** přihlásí u aktivního plátna a to pak při každé změně svého kroku zavolá u všech přihlášených přizpůsobivých instancí jejich metodu **`krokZměněn(int,int)`**.

Návrhový vzor *Pozorovatel* využívá i třída **`Multipřesouvač`** a její rozhraní **`IMultiposuvný`**, jež je potomkem rozhraní **`IPosuvný`** vyžadujícím navíc implementaci metody **`přesunuto(Multipřesouvač)`**. Instance třídy **`Multipřesouvač`** (která je mimochodem jedináček) může přesouvat několik instancí rozhraní **`IPosuvný`**. Je-li přesouvaný objekt instancí rozhraní **`IMultiposuvný`**, vyvolá **`Multipřesouvač`** po přesunutí objektu do požadované cílové pozice jeho metodu **`přesunuto(Multipřesouvač)`**, které se předá jako parametr. Instance se v této metodě může rozhodnout o svém dalším přesunu a opět o něj **`Multipřesouvač`** požádat.

Poznámka. Možná bude někomu připadat předchozí mechanizmus jako rekurzivní volání. Není tomu tak. Instance se přihlásí u multipřesouvače o přesun a tím svoji metodu ukončí. Až bude příště multipřesouvač posouvat svěřené objekty o kousek k jejich zadaným cílovým pozicím, přemístí i staronově zařazenou instanci.

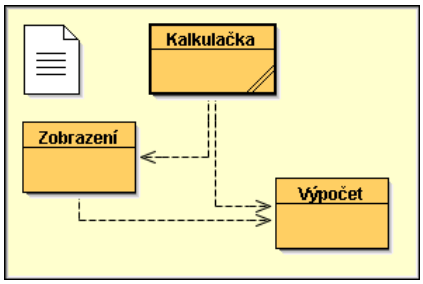
3 Návrh úprav stávajících příkladů

Prozatím jsem uváděl pouze příklady pracující s grafickými objekty. Přiznám se, že ty patří k mým oblíbeným, protože je na nich názorně vše vidět a vysvětlované principy při jejich použití „lépe tečou studentům do hlavy“. Nesmíme se však omezovat pouze na tento druh příkladů, protože pak studenti mohou podlehnout dojmu, že se OOP týká pouze práce s grafickými objekty.

Vyučující předkládají studentům různé druhy příkladů, přičemž část projektu často navrhnou sami a část projektu nechávají řešit studenty. Běžně zadávané projekty však mívají několik společných nevýhod:

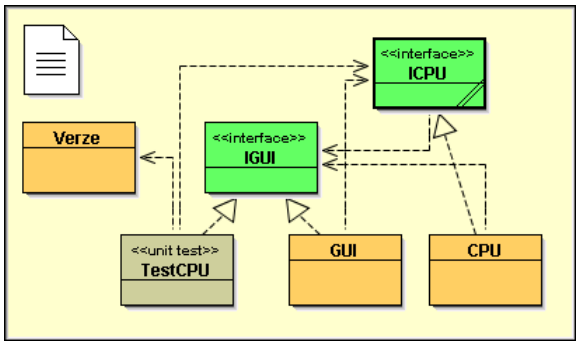
- Požadují po studentech především vyřešení algoritmických částí řešení a objektové pozadí jim poněkud uniká.
- Vyučujícímu zabere zbytečně moc času testování správnosti řešení požadované funkčnosti.
- Chce-li vyučující prověřit pochopení celého zadání studentem a požádá-li studenta o jakékoliv rozšíření, musí kvůli tomu měnit i třídu, kterou sám navrhl (nebo o její změnu požádat studenta).

Jedním z oblíbených zadání je návrh kalkulačky. Tento příklad uvádí již [1] a můžete se s ním setkat na řadě škol. Jeho diagram tříd většinou odpovídá diagramu na obr. 2, přičemž třídy **`Kalkulačka`** a **`Zobrazení`** definuje vyučující a student má za úkol navrhnou pouze předem zadané metody třídy **`Výpočet`**.



Obr. 2. Diagram tříd klasicky navrženého příkladu s kalkulačkou.

Takto navržená aplikace má výše popsané nevýhody. Pro vyučujícího by bylo mnohem výhodnější, kdyby definoval zadání podle diagramu tříd na obr. 3.



Obr. 3. Diagram tříd příkladu s kalkulačkou využívajícího rozhraní.

Toto zadání je sice na první pohled složitější, ale to je opravdu pouze na první pohled. Ve skutečnosti je pro vyučujícího mnohem výhodnější, protože mu šetří hodně práce. Svým způsobem je jednodušší i pro studenty (tedy alespoň pro ty, kteří chápou význam a možnosti použití rozhraní). Zkusím je proto popsat podrobněji:

Rozdělení na CPU a GUI

Kalkulačka je obdobně jako v předchozím zadání rozdělena na centrální procesorovou jednotku (`CPU`) a grafické uživatelské rozhraní (`GUI`). Na rozdíl od předchozího zadání však nejsou tyto části definovány primárně jako třídy, ale jsou zavedeny jako rozhraní. To přináší řadu výhod.

Tím, že `CPU` a `GUI` zastupují ve vzájemné komunikaci jimi implementovaná rozhraní `ICPU` a `IGUI`, si osvobodíme ruce k tomu, abychom mohli svobodně vyměňovat třídy reprezentující kteroukoliv z obou částí.

Rozhraní `IGUI` deklaruje pouze dvě přetížené metody `setRozměr`; první umožní zadat rozměr klávesnice (tj. počet sloupců a řádků kláves) a druhá vedle rozměru umožní zadat i bodovou velikost jejích tlačítek (to kdyby studenti používali funkce s delšími popisky).

Rozhraní `ICPU` deklaruje také dvě metody: metodu `getOperace(IGUI)`, která vrátí seznam popisků na klávesách kalkulačky přičemž ve svém parametru dostane odkaz

na aktuální `GUI`, a metodu `příkaz(String)`, která převezme jako parametr popisek na stisknuté klávese a vrátí textový řetězec, jenž se má vypsat na displeji.

To, že zde odpadla hlavní třída `Kalkulačka`, je pouze vedlejší efekt, protože vývojové prostředí *BlueJ*, které na kurzech používáme, takovou třídu nepotřebuje. Pro účely testování ji s výhodou zastoupí přípravek (fixture) v testovací třídě `TestCPU`. Pro pozdější úpravu na samostatně spustitelnou aplikaci ji můžeme kdykoliv doplnit.

Popis činnosti

Kalkulačku spustíme tak, že vytvoříme novou instanci `GUI`, přičemž jejímu konstruktoru předáme jako parametr odkaz na instanci použité `CPU`. Při vytváření své instance nejprve konstruktor `GUI` požádá svůj parametr o seznam popisků na klávesách, a pak vytvoří klávesnici, na jejíž klávesy postupně umístí obdržené popisky. Klávesy, pro něž obdrží prázdný popisek, přitom přeskočí.

`CPU` může v reakci na žádost o seznam operací zadat `GUI` požadovaný rozměr klávesnice a případně i jejích tlačítek. Musí to však učinit před tím, než vrátí požadovaný seznam popisků.

Význam třídy Verze

Instance třídy `Verze` uchovávají informace o jednotlivých zadáních. Při vytváření instance zadáme konstruktoru číslo požadované verze a konstruktor vytvoří instanci, která si pamatuje seznam požadovaných operací dané verze (ten vrátí po zavolání metody `getOperace()`) a seznam testovacích kroků, jimiž je možno příslušnou `CPU` otestovat. Tento seznam, resp. jeho iterátor získáme zavoláním metody `getTesty()`.

Snadné testování

Třídu `GUI` zobrazující podobu kalkulačky na obrazovce můžeme pro účely testování snadno nahradit třídou `TestCPU`, která se bude při komunikaci s testovanou `CPU` vydávat za plnohodnotnou `GUI` (`CPU` nemá šanci poznat, zda se něco doopravdy zobrazuje) a při té příležitosti její chování kompletně otestuje.

Každý student definuje svoji vlastní `CPU`, kterou je možno kdykoliv přidat do projektu a kompletně otestovat.

Testování je velice jednoduché: vytvoříme přípravek obsahující instanci testované `CPU` a instanci třídy `Verze` odpovídající příslušnému zadání. Pak už jen spustíme testy a *BlueJ* nám oznámí, jak dopadly.

Snadné ověřování znalostí

Rozhodne-li se vyučující prověřit pochopení studenta zadáním nějaké rozšiřující funkce, stačí studentu, aby jeho metoda `getOperace()` vrátila seznam bohatší o popisek pro nově přidanou funkci a aby naprogramoval reakci na stisk příslušné klávesy. `GUI` samo automaticky upraví vzhled kalkulačky podle obdrženého seznamu.

4 Další návrhové vzory, které by měly být probrány ještě ve vstupním kurzu

V předchozích dvou kapitolách jsem nastínil možnosti zařazení výkladu některých objektově orientovaných rysů (a především pak návrhových vzorů) do prvních hodin vstupního kurzu programování. Ve zbylých hodinách bychom neměli opomenout výklad následujících návrhových vzorů:

4.1 *Neměnné objekty* (*Immutable objects*)

V začátečnických kurzech i učebnicích často postrádám výklad toho, že datové typy můžeme dělit na odkazové (referenční) a hodnotové, tj. na typy, u nichž shodu objektů odvozujeme ze shody jejich odkazů a typy, u nichž shodu instancí odvozujeme ze shody jejich hodnot a v Javě ji zjišťujeme voláním metody `equals(Object)`.

Řada učebnic se sice zmíní, že instance některých datových typů (např. typu `String`) je třeba porovnávat prostřednictvím jejich metody `equals()`, avšak tuto skutečnost nijak dále nerozebírají. To, že hodnotové datové typy rozdělujeme ještě na proměnné (mutable) a neměnné (immutable) již většinou naprosto pominou a nejvýše upozorní, že instance datového typu `String` jsou neměnné a naznačí některé důsledky této skutečnosti.

Pokud se autor o dělení na proměnné a neměnné typy zmíní, tak tento výklad již většinou nedoprovodí výkladem toho, jaké jsou výhody a nevýhody neměnných objektů a především jak neměnné objekty definovat. Vynechám-li svoji učebnici, tak mezi texty, do nichž jsem měl možnost nahlédnout, jsem rozumně podrobný rozbor této otázky našel pouze v [2], resp. [3].

Autoři běžných kurzů většinou předpokládají, že na to jejich žáci přijdou sami. Nepřijdou. Mojí totiž tolik starostí s pochopením nového světa, že na odhalování některých detailů již nemají volnou mentální kapacitu.

4.2 *Jedináček* (*Singleton*)

Z tříd, o nichž jsem se zmiňoval v kapitole 2, jsou jako jedináčci definovány `Plátno`, `AktivníPlátno` a `Multipřesouvač`. S třídami, jejichž instance jsou jedináčci, se tedy studenti seznámí. Bylo by proto nanejvýš vhodné, aby se takové třídy naučili také definovat sami.

4.3 *Výčtový typ* (*Enumerated Type*)

Výčtový typ řeší problém definice třídy s předem známým počtem předem známých instancí. Java 5.0 jej zavedla vedle tříd a rozhraní jako samostatný druh datového typu. Bylo by vhodné seznámit studenty nejenom s jeho definicí a používáním, ale také s tím, jak jeho ekvivalent definovat v předchozích verzích Javy.

4.4 Prázdný objekt (*Null Object*)

Bývá častým zvykem, že metody vracejí v nejrůznějších okrajových situacích prázdný odkaz `null`. Metody, které takové metody volají, pak musejí ošetřovat, zda jim byl vrácen plnohodnotný objekt nebo prázdný odkaz.

V řadě situací je vhodné definovat v aplikaci nebo třídě objekt nazvaný např. `NULL`, který by byl vracen místo prázdného odkazu. Řada metod se tak zjednoduší, protože již nebudou muset testovat prázdnost vráceného odkazu. Je však třeba, aby měl onen prázdný objekt definovány správné metody.

4.5 *Iterátor* (*Iterator*)

S iterátorem se studenti seznámí při probírání kolekcí. Bylo by vhodné jim při té příležitosti prozradit, že se jedná o návrhový vzor, a rozebrat situace, kdy je výhodné jej použít.

4.6 *Tovární metoda* (*Factory Method*)

Metody `iterator()`, které poskytují všechny kolekce ve standardní knihovně, realizují návrhový vzor *Tovární metoda*. Na příkladu iterátoru lze také studentům tento návrhový vzor dostatečně průzračně vysvětlit.

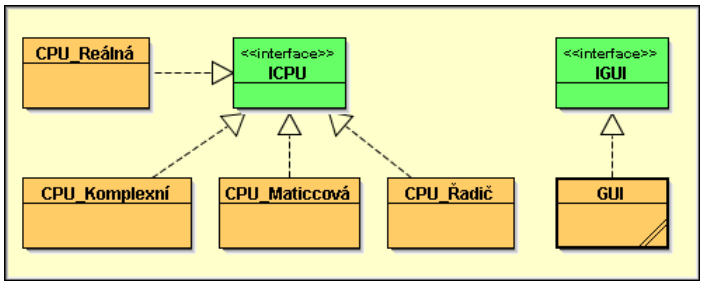
4.7 *Zástupce* (*Proxy*)

Zástupce je další z jednoduchých vzorů, s nímž je vhodné studenty seznámit již ve vstupním kurzu. Ukázka zadání, jehož řešení využívá tento vzor, je např. v [7]. Rád bych ale časem nalezl nějaké lepší příklady, ve kterých by studenti tento vzor nejenom potkali, ale bylo by v nich pro ně užitečné tento návrhový vzor také použít.

4.8 *Stav* (*State*) a *Strategie* (*Strategy*)

Další návrhové vzory, které si pro svoji jednoduchost zaslouží být vyloženy již v úvodním kurzu, jsou vzory *Stav* a *Strategie*. Vzor *Stav* je v [7] demonstrován na příkladu šipek (v současných kurzech nahrazuji šipky roboty), které se chovají odlišně podle směru, do nějž jsou natočeny (jinak se kreslí, pohybuje a zatáčí šipka či robot směřující na východ a jinak šipka či robot směřující na jih).

Pro demonstraci návrhového vzoru *Strategie* mi připadá optimální rozšířit před chvílí probíraný příklad s kalkulačkou. Navrhneme kalkulačku, která má několik CPU – jednu pro výpočty s reálnými čísly, druhou pro komplexní čísla, třetí např. pro maticové výpočty. Vše pak bude dirigovat řadič (také CPU), který rozlišuje reakci na přepínací příkaz (po něm změní obsah vnitřní proměnné odkazující na aktuální CPU) a jinak na ostatní příkazy (ty beze změny přepošle aktivní CPU) – viz obr. 4.



Obr. 4. Diagram tříd kalkulačky demonstrující návrhový vzor *Strategie*.

Na webu najdeme řadu dalších jednoduchých příkladů, v nichž se tyto návrhové vzory s úspěchem využijí.

4.9 *Příkaz* (*Command*)

Návrhový vzor *Příkaz* byl použit (spolu s několika dalšími vzory) už v příkladu s kalkulačkou. U něj by bylo vhodné poukázat na jeho příbuznost s návrhovým vzorem *Služebník* – liší se vlastně pouze v tom, jakými úvahami se k výslednému schématu dostaneme.

4.10 … a další

V předchozím výčtu jsem jmenoval pouze vzory, s nimiž se studenti ve vstupních kurzech bezpečně setkají (nebo by se s nimi setkat měli). Vedle toho je řada dalších vzorů, které nejsou tak složité, aby je nebylo možno použít v příkladech řešených ve stupních kurzech. Hlavním problémem je podle mne nalezení těch správně jednoduchých příkladů, na nichž se toho ale studenti hodně naučí.

5 Další náměty k zamyšlení

- Metodikou výuky ve vstupních kurzech se u nás hlouběji skoro nikdo nezabývá – většina vyučujících směřuje ve svém bádání k „vyšším metám“. Lepší vstupní kurzy jim však „přihrají“ lépe připravené studenty. Bylo by proto vhodné nové trendy v této oblasti alespoň sledovat a vyzkoušet.
- Neměli bychom se omezovat pouze na bádání nad vysokoškolskými kurzy programování. Programování se stále častěji učí na středních a dokonce i na základních školách. Nebudeme-li motivovat středoškolské učitele, aby změnili paradigma, v němž žáky vychovávají, budou do vysokoškolských kurzů programování stále přicházet strukturovaní (a to v lepším případě) programátoři, které zde bude nutno nejprve odnaučit mnohému z toho, co se dříve pracně naučili, a poté je naučit zcela novému, objektovému přístupu k řešení problémů.
- Při sledování současných trendů v programování bychom se neměli úzce orientovat na pouhé OO paradigma. OOP je sice vynikající metodika pro tvorbu a

správu rozsáhlých programových systémů. Vedle ní ale existuje ještě lepší metodika: *Ať to udělá někdo jiný*. Vedle umění navrhnout správný algoritmus či použít správný návrhový vzor bychom měli naše studenty naučit také schopnosti odpovědět si na otázku *Jaký hotový program nebo přípravek mohu při řešení svého problému využít?*

- Nepodařilo se mi vymyslet, jak terminologicky odlišit rozhraní představující množinu zveřejněných vlastností třídy a jejích objektů (tj. rozhraní jako protiváhu k implementaci) od představujícího druhu datového typu (tj. od rozhraní jako protiváhy k třídě). Tato snadná záměna termínů zbytečně komplikuje výklad, protože vyučující musí často používat termín střídavě v obou významech a to ztěžuje výklad i jeho chápání. V předjavovských dobách tento problém neexistoval, ale zavedením datových typů typu *interface* tento problém vyvstal a považuji jej za docela nepříjemný.

6 Závěr

Přes deklaraci principu „object-first" je začlenění výuky principů objektově orientovaného programování v kurzech a učebnicích, s nimiž jsem měl možnost se seznámit, pouze částečné. Domnívám se, že tato skutečnost je do značné míry dána nedostatkem příkladů, na nich by bylo možno návrhové vzory a další principy objektově orientovaného programování demonstrovat. Příspěvek ukázal některé možnosti a především pak příklady, které mohou být inspirací pro prohloubení „objektovosti" současných vstupních kurzů.

Vzhledem k tomu, že je tato problematika stále ještě v počátečních etapách vývoje, bylo by vhodné uspořádat obdobné semináře, jako jsou v textu zmiňované semináře *„Killer Examples" for Design Patterns and Objects First*, na nichž by si vyučující mohli vyměňovat své zkušenosti. Tyto semináře mohou být i virtuální nebo mohou být přidruženy k nějaké (např. této) konferenci.

Nezanedbatelnou je i nutnost osvěty mezi středoškolskými učiteli, kteří své studenti stále vychovávají podle dávno překonaných metodik, takže je vysokoškolské kurzy musí nejprve odnaučit mnohému z toho, co se pracně naučili. Ale to je již téma na samostatný příspěvek.

Reference

1. BARNES David J.; KÖLLING Michael: *Objects First with Java: A Practical Introduction Using BlueJ – 2nd edition*. Prentice Hall, 2005, ISBN: 0-13-124933-9.
2. BLOCH, Joshua. *Effective Java – Programming Language Guide*. Addison-Wesley Professional © 2001. 252 s. (Přeloženo [3]). ISBN 0-201-31005-8
3. BLOCH, Joshua. *Java efektivně – 57 zásad softwarového experta*. Praha: Grada © 2002. 230 s. (Překlad [2]). ISBN 80-247-0416-1
4. FREEMAN Eric; FREEMAN Elisabeth: *Head First Design Patterns*. O'Reilly, © 2004. ISBN 0-596-00712-4.

5. GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. (Přeloženo: [6]) ISBN 0-201-30998-X.
6. GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.* Praha: Grada, © 2003. 386 s. (Překlad [5]). ISBN 80-247-0302-5.
7. PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java 5.0*, Grada, 2004. ISBN 80-247-0941-4.
8. 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.
9. 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.
10. PECINOVSKÝ Rudolf: Proč a jak učit OOP žáky základních a středních škol. *Žilinská didaktická konferencia*, 2004, Žilina.
11. PECINOVSKÝ Rudolf: Jak efektivně učit OOP. *Tvorba softwaru 2005 – sborník přednášek*. ISBN 80-86840-14-X.