Objektově orientované programování? Co to je?

RUDOLF PECINOVSKÝ[1]
[1] ICZ a.s. Hvězdova 2a, 140 00 Praha 4; VŠE, nám. W. Churchilla 4, 130 67 Praha 3;
Tel.: +420 603 330 090, e-mail: rudolf@pecinovsky.cz;

Anotace

Příspěvek nejprve stručně probere historii OOP a ukáže, jak se postupně v programátorském světě prosazovalo. Poté vysvětlí základní principy moderního objektově orientovaného programování, a v závěru předvede aplikaci těchto principů při návrhu programu.

Úvod

Za posledních 10 let jsem se mnohokrát dostal do diskusí o vhodnosti či nevhodnosti výuky OOP na základních a středních školách. V řadě případů byly tyto diskuse více vášnivé než věcné, protože ne všichni hovořili o tomtéž. Řada zanícených oponentů nakonec musela přímo či nepřímo přiznat, že ono pomlouvané OOP ve skutečnosti nezná a ví o něm spíše z doslechu. Pokusím se proto v tomto příspěvku vysvětlit základní vlastnosti a rysy objektového paradigmatu a na jejich základě pak ukázat, proč by se mělo OOP učit i ve vstupních kurzech programování na základních a středních školách.

Trocha historie

Nejprve bych si dovolil zopakovat pár všeobecně známých historických faktů. Základní myšlenky, na nichž je OOP postaveno, se objevily na počátku 60. let. Do syntaxe programovacího jazyka pak byly poprvé začleněny v jazyku *Simula 67*. Jak název jazyka napovídá, jednalo se o jazyk vyvinutý pro programování diskrétních simulací. Později si bystré hlavy uvědomily, že každý program je ve skutečnosti simulací nějakého reálného či virtuálního světa a že by proto mohlo být užitečné aplikovat tyto myšlenky na všechny programy. Oprávněnost této myšlenky potvrzuje i současná převaha objektového paradigmatu.

Na počátku 70. let se v laboratořích firmy Xerox v Palo Alto vznikla skupina vývojářů, která začala vyvíjet koncepčně zcela nový programovací jazyk nazvaný *Smalltalk*. Tento jazyk stavěl na myšlenkách jazyka *Simula 67* a dále je rozvíjel. *Simula 67* přinesla základní myšlenky, ale objektově orientované paradigma stejně jako termín *objektově orientované programování* přinesli právě autoři jazyka *Smalltalk*.

Díky jazyku *Smalltalk* se OOP rozšířilo na univerzity a mezi vědci zabývající se počítačovou vědou. Programátoři pracující na softwarových zakázkách jej však používali jen výjimečně. Změna nastala na konci 70. letech, když Bjarne Stroustrup dostal v Bellových laboratořích za úkol naprogramovat simulaci rozsáhlých telefonních soustav. Znal jazyk *Simula*, ale ten mu připadal na tak rozsáhlý projekt příliš pomalý. Na druhou stranu jazyk *C*, který se tehdy postupně stával standardním jazykem systémových programátorů, byl sice dostatečně rychlý, ale byl zase příliš nízkoúrovňový, takže by vývoj tak rozsáhlého systému zabral zbytečně moc práce.

Stroustrup se rozhodl pro dvoufázové řešení: definoval upravenou verzi jazyka C, kterou nazval *C with classes* (C s třídami) a vytvořil soustavu maker, která převáděla program z tohoto jazyka do jazyka C. Získal tak sílu jazyka *Simula* a rychlost jazyka *C*. Tento jazyk se ukázal být velmi efektivním, takže se začal rychle rozšiřovat. V roce 1983 byl jazyk přejmenován na *C++* a byl vytvořen překladač, který překládal programy bez mezistupně.

Jazyk *C++* nabýval rychle na popularitě. Umožňoval totiž programátorům pracovat dál ve svém oblíbeném jazyce *C*, a přitom kdykoliv využít výhod, které nabízela objektově orientovaná nadstavba. Tyto konstrukce umožňovaly výrazně zefektivnit vývoj řady programů, takže je programátoři začali používat stále častěji až se bez nich zanedlouho nedokázali obejít.V 80. letech se proto objektově orientované programování začalo šířit jako lavina.

Jak ale všichni víme, nic není dokonalé. Programátoři používající objektově orientované konstrukce velmi často pochopili pouze syntaxi těchto konstrukcí, ale ne jejich skutečný význam. Často je proto používali ve svých programech nevhodně a ve chvíli, když program nabyl jisté „obludnosti“, začali se dostávat do potíží. A jak je u lidí zvykem, neobviňovali z těchto potíží vlastní nešikovnost, ale principy OOP.

Na konci 80. let proto řada z nich OOP opustila a vrátila se ke „starému dobrému“ strukturovanému programování. V té době se ale na odborných konferencích začaly množit příspěvky, které poukazovaly na toto špatné chápání zásad OOP a ukazovaly, jak by měl programátor „přemýšlet“ a programovat, aby byl jeho program doopravdy objektově orientovaný a aby mu používání OO paradigmatu přineslo onen slibovaný nárůst efektivity vývoje i spolehlivosti vytvořených programů.

Tyto hlasy se ale objevovaly většinou jen na konferencích a způsob práce řadových programátorů ovlivňovaly jen sporadicky. Zásadní změnu přineslo až vydání knihy *Design Patterns – Elements of Reusable Object-Oriented Software* [5] v roce 1995. Tato kniha ukázala, jak je třeba při vývoji objektově orientovaných programů přemýšlet a současně přinesla 23 ověřených doporučení na řešení typických programátorských problémů.

Návrhové vzory se staly rázem hitem a jako houby po dešti se začaly množit knihy, které se snažily vysvětlit danou problematiku i řadovým programátorům a paralelně i další knihy, které přinášely vzory z dalších oblastí programování.

Co je to OOP

Základní charakteristika

Opusťme nyní historii, vraťme se ke kořenům a vysvětleme si, co to vlastně je *Objektově orientované programování* a jak se vývoj objektově orientovaných programů liší od vývoje programů strukturovaných.

Jak jsem již řekl kapitole o historii, každý program je simulací reálného či virtuálního světa:

- Vytváříme-li program pro účtárnu, simulujeme v něm vytvoření faktury, její odeslání zákazníkovi, komunikaci mezi zákazníkem a účtárnou a její (doufejme) závěrečné proplacení.
- Vytváříme-li textový editor, simulujeme virtuální svět dokumentu, jeho stránek, odstavců, písmen obrázků a dalších náležitostí.

A tak bychom mohli pokračovat pro každý program, který vytváříme.

Dobrá, přijmeme tedy tezi, že program je simulací reálného či virtuálního světa. Svět, jak jej známe, sestává z objektů. Má-li být proto naše simulace co nejpřesnější, bylo by vhodné, kdyby program uměl pracovat s objekty.

V reálném světě jsou všechny děje důsledkem toho, že spolu objekty navzájem interagují – jeden objekt působí na druhý a ten na to reaguje. Interakce objektů simulujeme v objektově orientovaných programech zasíláním zpráv.

V reálném světě se posadíme na židli a v závislosti na naší váze a kvalitě židle nás židle unese či neunese. Současně se v závislosti na typu podložky může židle, na níž usedáme, zabořit.

V objektovém programu pošle objekt představující nás objektu představujícímu židli zprávu o tom, kolik kg se na židli usazuje. Objekt židle zhodnotí, jestli usednuvšího člověka unese a pokud ano, pošle své podložce zprávu o změně zatížení svých nohou. Podložka židli odpoví, zda se zaboří a o kolik. Židle vše vyhodnotí a pošle „usedajícímu objektu“ informaci o tom, jak jeho akce dopadla.

Objektově orientovaný program versus strukturovaný program

Z předchozího začíná vysvítat (alespoň v to doufám), čím se objektově orientovaný program liší od klasického strukturovaného programu:

- Ve strukturovaném programu bychom výše zmíněnou situaci s usazováním se osoby na židli řešili definicí funkce[1], které předáme jako parametry usazovanou osobu, židli a případně i podložku, na níž židle stojí. Funkce nám pak vrátí informaci o výsledku této akce, podle nějž se další program zařídí.
- V objektově orientovaném programu bychom dané osobě (přesněji objektu, který ji představuje) poslali zprávu, aby se usadila na židli. Dále by vše probíhalo tak, jak jsem před chvílí popsal.

Když tedy popsaný případ trochu zobecníme, dostaneme :

- *Strukturovaný program* je (přesněji řečeno většina lidí jej tak chápe) v nějakém jazyce zapsaný *postup řešení zadané úlohy*. (Není to sice úplně přesné, ale pro většinu programů takováto definice vyhovuje.)
- *Objektově orientovaný program* je v nějakém jazyce zapsaná *množina objektů a zpráv, které si tyto objekty mezi sebou předávají*.

Způsob návrhu strukturovaného a objektově orientovaného programu se od sebe liší stejně zásadně, jako se od sebe liší jejich definice:

- Při návrhu strukturovaného programu vymýšlíme postupy a datové struktury, na něž budeme tyto postupy aplikovat.
- Při návrhu objektově orientovaného programu se snažíme odhalit účastníky a následně pak definovat zprávy, které si budou vzájemně posílat.

Podívá-li se na tyto dva popisy člověk schopný dostatečné abstrakce, uvidí, že oba hovoří, byť každý jinými slovy, v podstatě o tomtéž. Z hlediska zákazníka, který si objednává vývoj programu, to však vůbec není totéž. Při komunikaci se strukturovaným programátorem se daleko výrazněji projeví tzv. **sémantická mezera**, což je termín postihující rozdíl mezi tím, jak popisuje problém zákazník – budoucí uživatel a jak jej popisuje programátor.

Prakticky všichni strukturovaní programátoři, kteří přicházejí do mých přeškolovacích kurzů, mají velký problém s popisem svého řešení jazykem uživatele. Z toho ale zákonitě vyplývá, že jim uživatel nemůže poskytnout kvalitní zpětnou vazbu a odhalit, kde takový se programátor ve svých předpokladech o požadovaném chování programu mýlí.

Vyjadřování objektově orientovaných programátorů je zákazníkovu chápání mnohem bližší. To má dva důsledky:

- Programátor musí dělat méně zobecňujících kroků, aby se dostal na úroveň abstrakce, kterou od něj požaduje používané paradigma, a tím sníží pravděpodobnost, že některý z těchto zobecňujících kroků nebude zcela korektní.
- Při popisu použitého řešení se programátor může vyjadřovat pro zákazníka mnohem srozumitelněji, aniž by se přitom nějak výrazně vzdaloval od popisu, který používá při komunikaci s počítačem. Zákazník mu proto může poskytnout mnohem lepší zpětnou vazbu a dříve odhalit případnou chybnou interpretaci zadání.

Jak jsem již naznačil, problém není v tom, že by strukturovaní programátoři neuměli mluvit. Problém je především v těch několika hladinách abstrakce mezi zadáním a výsledným kódem.

[1] Dohodněme se, že procedura je pouze speciálním případem funkce – je to funkce která nic nevrací. Starší jazyky procedury a funkce rozlišovaly, současné jazyky již mezi nimi žádný rozdíl nedělají.

Objekty

Opusťme nyní na chvíli obecné povídání o objektově orientovaném paradigmatu a podívejme se blíže na jeho stavební kameny. Jak jsme si řekli při seznamování se základních charakteristikou OOP, objektové paradigma předpokládá, že simulovaný svět sestává z objektů. Proto jsou také objekty nejen v názvu tohoto paradigmatu, ale také v centru jeho zájmu

Většina lidí vnímá objekty jako něco, co se dá uchopit. Otevřete-li však např. Encyklopedii Britanica, najdete v ní mnohem obecnější definici: *Objekt je cokoliv, co můžeme vnímat našimi smysly nebo co ovlivňuje vnímání našimi smysly*. Když se nad touto definicí zamyslíte, zjistíte, že podle ní může být objektem prakticky cokoliv.

Obdobné chápe objekt i OOP. Zobecňuje naši laickou definici objektu a prohlašuje za objekt cokoliv, co můžeme nazvat podstatným jménem. To, že stůl, židle, počítač auto či člověk jsou objekty, pochopí každý. Řada programátorů však má problémy se vstřebáním faktu, že objektem jsou i

- vlastnosti (barva, směr, délka, krása),
- události (připojení, přerušení, počítání),
- stavy (připravenost, běh,),
- děje (spouštění, komunikace, výpočet)

a další charakteristiky, které obecně považujeme za abstraktní.

Akceptaci tohoto zobecněného chápání objektů většině programátorů usnadní, když si uvědomí, že každá výše popsaná abstraktní charakteristika je reprezentována nějakou množinou údajů (barva má několik barevných složek, směr může být charakterizován úhlovou odchylkou od severu, připojení můžeme popsat adresou partnera, použitým protokolem, rychlostí komunikace a případnými dalšími charakteristikami). Tuto sadu charakteristik zabalíme do jakési datové struktury, kterou označíme *objekt*. Když si tuto skutečnost programátoři uvědomí, přestanou mít většinou s chápáním zobecněného objektu problémy.

Stav – atributy

Objekt může obsahovat jiné objekty. Tyto objekty tvoří jeho **atributy**. Hodnoty atributů definují stav objektu a ovlivňují jeho vlastnosti.

To, které atributy objektu definujeme, záleží na tom, k čemu chceme objekt použít. Vytváříme-li auto pro nějaké závody, definujeme pro něj nějaký vzhled a jízdní vlastnosti. Vytváříme-li auto pro aplikaci, která má sloužit k technické výuce v autoškolách, budeme se při definici jeho atributů soustředit na jednotlivé komponenty, jejichž účel a funkci chceme studentům vysvětlit.

Schopnosti – metody

Říkali jsme si, že práce OO programu spočívá v tom, že objekty si navzájem posílají různé zprávy. Na zaslanou zprávu oslovený objekt nějak reaguje. Tuto reakci definujeme v kódu, který bývá označován jako metoda. Programátoři proto většinou neříkají, že objekt posílá druhému objektu zprávu, ale že volá jeho metodu.

Činnost metody, tj. reakce osloveného objektu na zprávu, záleží na tom, kým a jak byla zpráva odeslána, a na tom, v jakém stavu se objekt v okamžiku obdržení zprávy nacházel. Ostatně i my budeme např. na zprávu o tom, že venku prší, reagovat jinak, když bude venku +5° C a když tam bude +35° C.

Třídy

Víme, že objekty se v okolním světě často opakují. Protože jsou všichni programátoři líní, snaží se jim autoři jazyků jejich práce usnadnit. Jedním z výrazných usnadnění je zavedení tříd. Třídu můžeme chápat jako šablonu, která definuje, jak se budou vytvářet objekty, které označíme jako instance dané třídy. Tato šablona definuje, jaké budou mít její instance atributy a jaké budou mít metody.

Shodnost atributů jednotlivých instancí ale neznamená shodnost jejich hodnot. Můžeme např. definovat třídu `Auto`, jejíž instance budou mít atribut `motor`. Každá instance však bude mít svůj vlastní motor a tyto motory se navíc mohou vzájemně lišit výkonem, spotřebou, provedením apod.

Třída ale může definovat i vlastní atributy a metody a ty pak její instance sdílejí. Je to obdobné, jako když máte doma každý svůj kartáček na zuby a ručník, ale všichni sdílíte společnou vanu. Když instance změní hodnotu některého svého atributu, ostatní instance se o tom nedozvědí. Pokud však změní hodnotu atributu třídy, budou to hned všechny instance vědět, protože s ní tento atribut sdílejí.

Návrh jednoduchého programu

Začátečníkům v OOP se doporučuje, aby si před návrhem programu podrobně popsali řešený problém. Pak v popisu podtrhají podstatná jména a tím získají doporučení, jaké mají definovat objekty, resp. třídy objektů. Pak si podtrhají všechna slovesa a získají tak doporučení, jaké mají definovat metody.

Přiznejme si však, že na druhou část většinou nedojde, protože v okamžiku, kdy je jasné, které třídy a objekty budou v programu vystupovat, dokáží i začátečníci poměrně rychle odvodit, které metody bude potřeba definovat, a to i bez pomoci výše zmíněné berličky s podtrhanými slovesy.

Při návrhu OO programů se však výsledky této předběžné analýzy nepřevádí hned do kódu, ale zaznamenávají se nejprve v grafickém jazyku UML (Unified Modeling Language). Ten definuje sadu diagramů, v nichž se zakreslují informace o projektu v jednotlivých fázích jeho vývoje od úvodních rozhovorů se zákazníkem až po výsledné rozmístění částí rozsáhlejšího programu na jednotlivých počítačích.

Pro naše účely se využívá diagram tříd, v němž se zobrazují jednotlivé třídy a případné závislosti mezi nimi. Třídy jsou zde představovány obdélníky, rozdělenými vodorovně na tři části:

- do horní části se píše tučně název třídy,
- do střední části se zapisují definované atributy a
- do spodní části se zapisují definované metody.

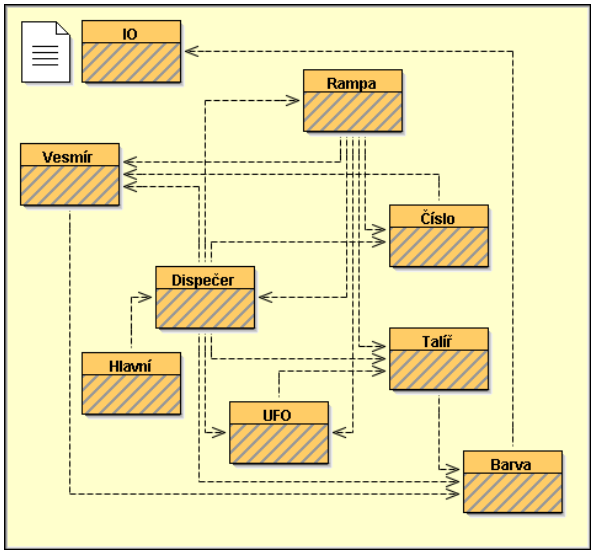
Často, zejména v začátečnických kurzech, se však používá zjednodušený diagram tříd, v němž se pro zvýšení přehlednosti zapisují do obdélníku pouze názvy tříd. Ukažme si vše na následujícím jednoduchém programu převzatém z učebnice [9].

UFO

Naprogramujte **hru**, při níž **hráč** ovládá **UFO** pohybující se ve **vesmíru**. Cílem hráče je dopravit UFO ze startovací **rampy** na přistávací rampu. Hráč může ovládat několik UFO současně. Abychom mohli jednotlivá UFO odlišit, je na **talíři** každého z nich napsané **číslo**. Čísly jsou označeny i jednotlivé přistávací rampy.

V předchozím zadání jsou zvýrazněna podstatná jména. Každé z nich označuje objekt, který je reprezentován třídou. Jedinou výjimkou je hráč, který není reprezentován objektem, ale osobou sedící z klávesnicí.

Při úvahách nad projektem ještě vyplyne, že mají-li se UFO pohybovat plynule, musí být do programu zařazen ještě někdo, kdo ovládá animaci. Tento objekt je nazván `Dispečer`. Kromě toho byla do výsledného projektu ještě třída označená `Hlavní`, která má na starosti spuštění celého programu. Výsledný diagram tříd je na obrázku 1.



Obrázek 1
Zjednodušený diagram tříd projektu UFO

V předchozím diagramu zobrazují šipky vzájemné závislosti tříd. Pokud instance jedné třídy používají ke své činnosti instance jiné třídy, jsou na ní závislé. To se v diagramu znázorní čárkovanou šipkou směřující od závislé třídy ke třídě, na níž tato třída závisí.

Diagram tříd názorně zobrazí rozdělení celého projektu na jednotlivé třídy. Každá třída má přidělen jednoznačný úkol, za nějž jsou ona a její instance zodpovědné. Definice jednotlivých tříd se pak dá rozdělit mezi členy týmu. Každý vyhotoví svoji část programu a při příští schůzce pak dají vše dohromady a prověří vzájemnou spolupráci jednotlivých tříd a jejich instancí.

Další rysy objektových programů

Rozhraní a implementace, zapouzdření

Při vývoji objektově orientovaných programů se klade velký důraz na to, aby jednotlivé části programu nemohly využívat své „znalosti“ o tom, jak je protější část naprogramována. U všech entit (objekty, třídy, metody, …) se proto rozlišují dvě charakteristiky:

- *Rozhraní*, které definuje, co o dané entitě ví okolní program.
- *Implementace*, která specifikuje, jak je dosažené správné funkčnosti dané entity.

Jedním z nejdůležitějších pravidel správného objektově orientovaného programování je zásada *programovat proti rozhraní a ne proti implementaci*.

Moderní programovací se snaží neponechávat dodržování této zásady pouze na programátorovi, ale snaží se její dodržení kontrolovat a případně také vynucovat.

Se skrýváním implementace souvisí nejdůležitější rys objektově orientovaného programování, kterým je **zapouzdření**. Zapouzdřením označujeme dvě věci:

- Umístění dat (atributů) a kódu (metod), který s těmito daty pracuje, pohromadě do definice třídy. Programátor tak má přehled o vlastnostech a stavu zpracovávaných dat a dělá proto mnohem méně chyb.
- Znemožnění ostatním částem programu manipulovat s těmito daty jakýmkoliv jiným způsobem, než prostřednictvím metod vlastníka těchto dat, tj. metod příslušného objektu. Jinými slovy: k datům objektu je možné přistupovat pouze způsobem definovaným v rozhraní daného objektu.

Důsledné zapouzdření všech částí programu dramaticky zvyšuje efektivitu vývoje i spolehlivost výsledného programu. Všechny další rysy jazyka jsou proto často posuzovány podle toho, nakolik podporují nebo naopak narušují optimální zapouzdření.

Dědičnost

Jednou z konstrukcí, které nabízí většina objektových programovacích jazyků, je možnost definice dědičnosti. Dědičnost představuje speciální druh skládání, při němž je do objektu vložen jiný objekt, tzv. rodičovský podobjekt, a nový majitel tohoto vloženého objektu přebírá jeho rozhraní a případně k němu přidává své vlastní rysy. Vložený objekt, resp. jeho třída, je pak označen za rodiče (odtud také název *rodičovský podobjekt*) a jeho majitel za potomka.

Dědičnost přináší elegantní způsob, jak si ušetřit programování. Potomci totiž dědí všechny dostupné metody svých rodičů, takže jim stačí, když sami definují pouze ty, jejichž rodičovská definice jim nevyhovuje, a ty, které rodič vůbec nemá.

Dědičnost je však typickou konstrukcí, která narušuje zapouzdření, a to v obou jeho rysech:

- Potomkům bývá často umožněno pracovat přímo s daty svého rodiče, takže přestává platit, že kód je blízko zpracovávaných dat se všemi z toho plynoucími důsledky.
- Rodič musí často prozradit potomkovi něco o své implementaci, protože jinak by potomek nebyl schopen definovat své metody bezchybně. To opět zvyšuje pravděpodobnost chyb.

Obecná zásada proto říká, že dědičnost máme použít pouze tehdy, když jakékoliv jiné řešení je výrazně těžkopádnější.

Kromě toho platí další zásada: má-li program zůstat stabilní, musí být potomek vždy speciálním případem předka. Obecně bychom mohli prohlásit, že *instance potomka musí být vždy schopna se plnohodnotně vydávat za instanci svého předka*. Bohužel, tato zásada bývá v řadě rádobyobjektových programů porušována. A co je ještě horší, její porušení najdeme v ukázkových příkladech na přednáškách kurzů objektového programování i na řadě univerzit včetně těch, které se vydávají za špičkové a prestižní.

Interface

Jak jsem již uvedl v souvislosti se zapouzdřením, v objektově orientovaném programování se důsledně odděluje rozhraní a implementace. Programovací jazyk Java dokonce zavedl speciální konstrukci **interface**, která definuje pouze rozhraní bez jakékoliv implementace. *Interface* své metody pouze deklaruje, avšak nijak je neimplementuje. To ponechává na třídách, které se přihlásí k jeho implementaci. Takové třídy pak mohou své instance vydávat za instance implementovaného *interface*-u.

Protože *interface* nedefinuje žádnou implementaci, nemůže mít ani žádné instance. Požaduje-li některá část programu instancí *interface*-u, musí ji zastoupit instance nějaké třídy, která daný *interface* implementuje.

Mohli bychom říci, že implementovaný *interface* se chová podobně jako rodič – implementující třída také přebírá jeho rozhraní. Protože však instance implementující třídy nepřebírají žádný rodičovský podobjekt, nemůže dojít ani k jednomu z obou výše uvedených problémů, s nimiž se setkáváme u dědičnosti.

Moderní techniky aplikované v OOP

Návrhové vzory

Jak jsem již uvedl v historickém přehledu, současné programování výrazně ovlivnil příchod návrhových vzorů. Jejich používání zvyšuje efektivitu vývoje, spolehlivost a robustnost výsledných programů, jejich spravovatelnost a umožňuje mnohem rychlejší reakce na změny zadání.

Návrhové vzory bychom mohli označit za programátorskou verzi matematických vzorečků. Jenom se do nich nedosazují čísla, ale třídy, objekty a v některých případech metody. Znalost návrhových vzorů přináší několik výhod:

- Vývoj se zrychluje, protože programátoři nemusí v řadě případů vymýšlet vlastní dostatečně dokonalé a efektivní řešení.
- Díky tomu se vývoj i zkvalitňuje, protože odpadá možnost vzniku chyby při vývoji tohoto speciálního řešení.
- Vývoj se dále zlevňuje, protože části programu navržené s využitím návrhových vzorů je možné mnohem snadněji použít i v dalších programech.
- Zlepšuje se i komunikace mezi členy týmu. Když programátor řekne že někde použil jedináčka či most, všichni vědí, co od daného řešení mohou čekat a kde na ně číhají jaká úskalí a nemusí si vše vzájemně sáhodlouze vysvětlovat.

Refaktorace

Když jsem se v polovině sedmdesátých let začal učit programovat, učilo se, že před začátkem jakéhokoliv programování se musí nejprve provést důkladná analýza, abychom uprostřed programování neobjevili, že budeme muset něco naprogramovat jinak. Platilo totiž, že cena opravy s dobou jejího odhalení velice rychle narůstá.

Zkušenost z 80. a zejména pak z 90. let ukázala, že nezávisle na tom, jak dokonalá bude analýza, program se bude s nejvyšší pravděpodobností měnit – jednou kvůli dodatečné změně v zadání, jindy kvůli novým technologiím. Byly proto vypracovány metody, jak program upravit, aby takovéto pozdní zásahy přišly co nejlevněji. Tyto techniky úpravy bývají označovány jako *refaktorace kódu*. V průběhu let se natolik osvědčily, že jsou nyní integrální součástí všech lepších vývojových prostředí.

Automatizované testování

Dalším velkým přínosem objektového programování je zavedení automatizovaných jednotkových testů. Nové knihovny umožnily navrhovat testy mnohem efektivněji, než tomu bylo v minulém století, a tyto testy pak automaticky spouštět.

Řada nových metodik dokonce prosazuje tzv. *programování řízené testy*, které vyžaduje, aby programátor nejprve napsal testy vytvářeného programu, a teprve pak začal programovat. Jeho cíl se tím velmi zjednoduší (a tím stoupne jeho produktivita): jeho úkolem nyní totiž bude pouhé zprovoznění předem připravených testů.

Podle této metodiky programátor spouští automatizované testy po každé, byť nepatrné, změně programu. Úspěšné proběhnutí testů mu umožní se spoléhat na doposud napsaný kód a programovat v další etapě o to efektivněji. Naopak havárie testů oznámí, že chybu do

programu zanesla některá z naposledy zanášených změn. Spouštíte-li jednotkové testy dostatečně často, bude těchto změn jenom velice málo a nemělo by dělat problém si pamatovat, co vše se zaměnilo, a snadno tak odhalit, kde je chyba.

Závěr

Jak jsem již několikrát upozornil, OOP přináší velký nárůst produktivity vývoje, spolehlivosti výsledných programů i možnosti vzájemné spolupráce jednotlivých programů. Říkali jsme si dále o tom, že OOP umožňuje psát programy tak, abychom mohli co nejefektivněji reagovat na měnící se zadání a dodatečné požadavky našich zákazníků.

OOP vyžaduje zcela jiný přístup k řešení problémů. Tento přístup často snáze osvojí ti, kteří doposud nikdy neprogramovali, než ti, kteří mají bohaté programátorské zkušeností a s nimi související zažité stereotypy. Obzvlášť patrné je to u dětí, pro něž je způsob uvažování vyžadovaný OOP mnohem přirozenější než způsob vyžadovaný klasickým strukturovaným paradigmatem.

Literatura

[1] BECK, Kent. *Programování řízené testy*. Grada © 2004. 204 s. (Překlad [2]) ISBN 80-247-0901-5

[2] BECK, Kent. *Test Driven Development By Example*. Addison-Wesley © 2003. 220 s. (Přeloženo v [1]) ISBN 0-321-14653-0

[3] FOWLER, Martin. *Refaktoring. Zlepšení existujícího kódu.* Grada, © 2003. 394 s. (Překlad [4]) ISBN 80-247-0299-1

[4] FOWLER, Martin. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, © 2000. 430 s. (Přeloženo v [3]) ISBN 0-201-48567-2.

[5] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. ISBN 0-201-30998-X.

[6] PAGE-JONES, Meilir. *Základy objektově orientovaného návrhu v UML*. Grada © 2001. 368 s. (Překlad [7]) ISBN 80-247-0210-X

[7] PAGE-JONES, Meilir. *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley © 2000. 458 s. (Přeloženo v [6]) ISBN 0-201-69946-X.

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

[9] PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java.* Grada, © 2008. 576 s. ISBN 978-80-247-2653-3