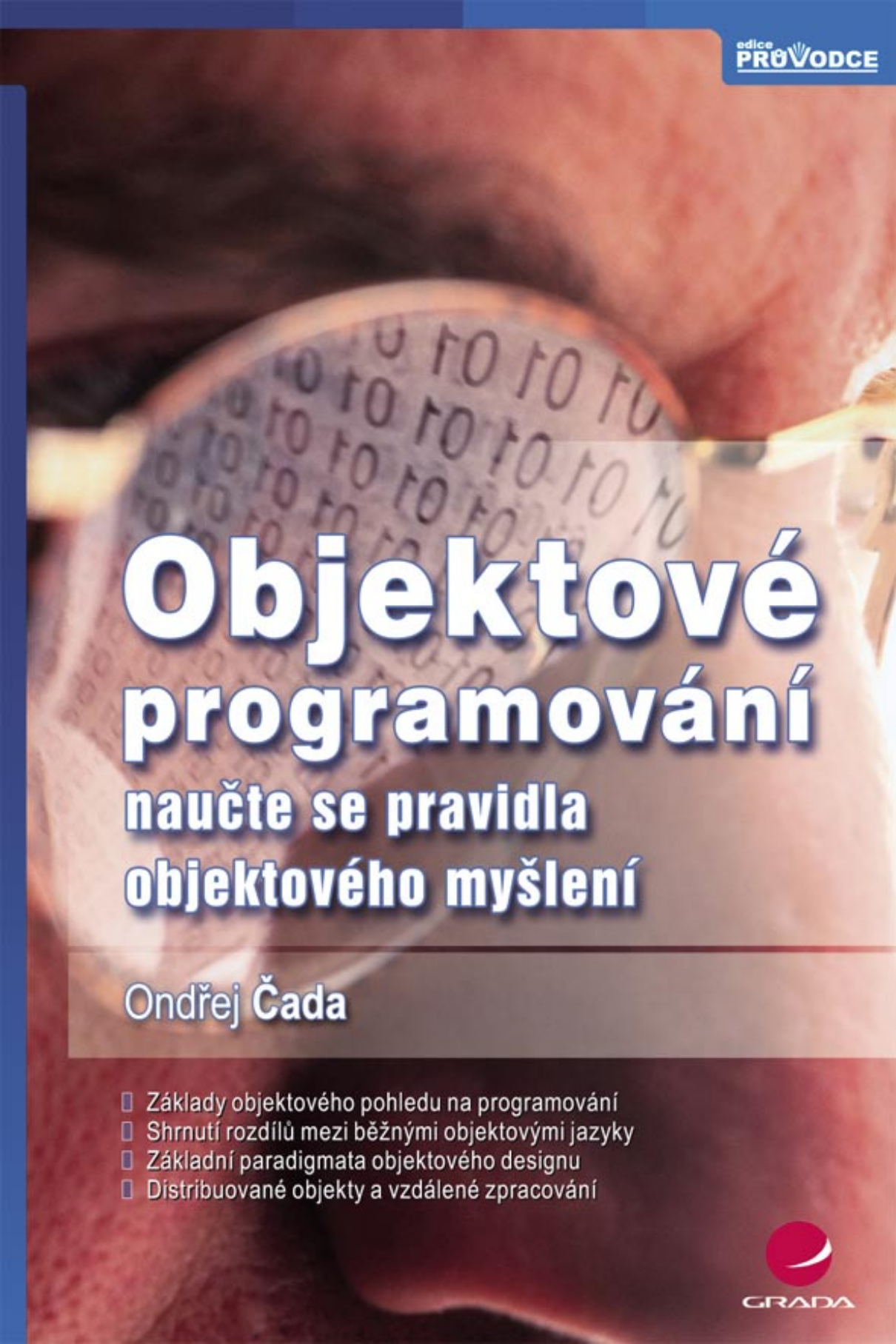

edice
PROVODCE
Objektové programování
naučte se pravidla objektového myšlení
Ondřej Čada
Základy objektového pohledu na programování
Shrnutí rozdílů mezi běžnými objektovými jazyky
Základní paradigmata objektového designu
Distribuované objekty a vzdálené zpracování
GRADA

Upozornění pro čtenáře a uživatele této knihy

Objektové programování

naučte se pravidla objektového myšlení

Ondřej Čada

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
jako svou 3648. publikaci

Odpovědný redaktor Tomáš Vild
Sazba Tomáš Vild
Návrh a grafická úprava obálky Vojtěch Kočí
Počet stran 200
První vydání, Praha 2009

Cover Photo © fotobanka allphoto

Vytiskly Tiskárny Havlíčkův Brod, a.s.
Husova ulice 1881, Havlíčkův Brod
ISBN 978-80-247-2745-5 (tištěná verze)
ISBN 978-80-247-6699-7 (elektronická verze ve formátu PDF)

Obsah

Úvod

V této knize se zaměříme na základy objektového programování jako takového. Nepůjde však o konkrétní učebnici toho či onoho objektového programovacího jazyka s těmi či oněmi konkrétními knihovnami tříd; namísto toho si vysvětlíme řadu obecných vzorů a mechanismů, jichž lze s výhodou využívat v podstatě kdekoli.

Samozřejmě s jistým omezením daným možnostmi a flexibilitou konkrétního prostředí a programovacího jazyka: kupříkladu v C++, jež nabízí velmi omezené a nedokonalé služby pro práci s objekty, jsou možnosti využití standardních vzorů značně limitovány.

Kniha je psána pro všechny úrovně čtenářů, od úplných začátečníků, jimž přinese základní představu o struktuře a funkci objektového systému a o službách, jež jsou s ním spojeny, přes mírně pokročilé, již se zde seznámí s řadou standardních principů a mechanismů, usnadňujících objektové programování, stejně jako s některými nejběžnějšími chybami a problémy – a samozřejmě také s ukázkami toho, kterak se chybám vyhnout a problémy řešit. Zkušení programátoři zde pak najdou řadu poměrně podrobných komentovaných příkladů, ilustrujících vhodná objektová řešení mnoha běžných úloh – až po značně pokročilé mechanismy jako je kupříkladu popis distribuovaných objektů nebo nepřímé zasílání zpráv, kde nabízíme i plně funkční ukázku jednoduché implementace.

Úroveň výkladu je dostatečně podrobná na to, aby textu porozuměl kdokoli, kdo již má alespoň základní programátorské zkušenosti: obecnou problematiku algoritmizace a programování jako takového kniha ovšem nepokrývá.

Podobně také není součástí knihy detailní výklad žádného konkrétního objektového programovacího jazyka, a text knihy ani takovou znalost nevyžaduje. Je však zapotřebí, aby měl čtenář alespoň nejzákladnější zkušenosti s nějakým programovacím jazykem a jeho základními mechanismy do té míry, aby dokázal porozumět jednoduchým a detailně komentovaným příkladům v jiných jazycích: nebudeme explicitně popisovat ani zcela obecné principy jako např. podmínkový příkaz, příkaz cyklu či programový blok, nebo pojmy „proměnná“ či „typ“.

Ačkoli zběžně se zmíníme o řadě různých objektových programovacích jazyků, příklady budeme poměrně důsledně uvádět v následujících třech:

- Kdekoli si vystačíme s jejími poměrně omezenými službami a limitovanou podporou objektového systému, budeme pro příklady používat *Javu*. To proto, že jde o jazyk poměrně dobře srozumitelný i těm, kdo jej důkladně neznají, jeho konstrukce jsou velmi intuitivní – a navíc lze také předpokládat, že Javu bude nejspíš běžně používat většina čtenářů této knihy.

- Tam, kde statický ne-tak-docela-objektový systém Javy pro potřeby konkrétního příkladu nestačí, použijeme objektový programovací jazyk *Ruby*. Jeho obliba a míra používání v současnosti právem stoupá: jde o jazyk přehledný a snadno pochopitelný, umožňující čtenáři soustředit se na vlastní problém a neřešit okrajové technické nepodstatné záležitosti. Přitom se jedná o velmi hezky a čistě navržený systém s plně dynamickou objektovou podporou. Snad jedinou chybou Ruby je trochu nešťastný standard pojmenovávání běžně užívaných metod jeho knihovních tříd – věci jako „to_s“ či „<<“ (nemluvě ani o „=~“) nejsou bez podrobnějšího výkladu příliš zřejmé.

- Čím dále budeme pokračovat směrem ke konci knihy k náročnějším a složitějším příkladům, tím častěji se setkáme s *Objective C* a jeho nejběžnějšími knihovnami tříd *Cocoa*. Jeho obecnou nevýhodou z koncepčního hlediska – ale samozřejmě o to silnější výhodou při praktickém programování – je to, že vzhledem ke zpětné kompatibilitě s klasickým neobjektovým programovacím jazykem C nese zátěž statického typového systému a neobjektových typů. Právě u složitějších příkladů však tato nevýhoda relativně zaniká ve srovnání s tím, že Objective C je obecně snadno čitelné a srozumitelné hlavně díky dobře navrženým jménům tříd a metod standardních knihoven – např. účel zprávy „description“ je zřejmý i bez dalšího výkladu (na rozdíl od Rubyovského zhruba ekvivalentního „to_s“); podobně např. můžeme srovnat zprávu Cocoa „addObject:“ s Rubyovskou zprávou „<<“. Další výhodou právě pro složitější příklady je míra praktické použitelnosti: ukážeme-li si řešení problému v Objective C s jeho komplikacemi, danými statickým typovým systémem a řadou neobjektových typů a konstrukcí, dokážeme pak týž problém snadno vyřešit i v jakémkoli jiném plně objektovém programovacím jazyce. Konečně pak není zanedbatelné ani to, že Objective C je primárním programovacím jazykem významné platformy Apple, a zároveň je plně přenositelné a platformně nezávislé v rámci projektu GNU (to se týká i standardních knihoven tříd Cocoa, jichž je valná většina – byť ne zcela všechny – v přenositelné podobě volně přístupná v rámci projektu GNUStep).

Všechny příklady jsou podrobně popsány, a každá jazyková konstrukce či využití některé ze standardních knihovních služeb jsou napoprvé detailně vysvětleny.

1.

Základy objektového pohledu

V úvodní kapitole se trochu blíže podíváme na programování jako takové a možné přístupy; vymezíme si pojem „objektové programování“ a pro lepší kontext se seznámíme s některými dalšími běžně užívanými pojmy. Zběžně také nahlédneme do historie a vytyčíme si rozsah principů, jimiž se vlastně v této knize chceme zabývat.

1.1 Programování je co?

Ačkoli pojem „programování“ – jehož objektové variantě je tato kniha věnována – se používá poměrně široce v nejrůznějších kontextech (*programujeme* videorekordér, aby nám nahrál požadovaný pořad; *programujeme* termostat, chceme-li mít v domě teplo; řada psychologů hovoří o tom, že *programujeme* sami sebe), my se v této knize samozřejmě soustředíme pouze na programování ve smyslu „tvorby aplikací pro samočinné počítače“.

V tomto rámci se již ale nebudeme omezovat na nejužší význam, totiž „vlastní psaní zdrojového kódu“; budeme se zabývat minimálně trojicí vzájemně se ovlivňujících činností, jež všechny lze při troše dobré vůle do pojmu *programování* zahrnout:

- *Analýza* problému je prvním krokem, bez nějž se programování (samočinných počítačů, ale vlastně nejen těch) neobejde. Již zde je důležité zvolit pohled, odpovídající paradigmatu, v jehož rámci chceme problém řešit – v našem případě tedy půjde o „objektovou analýzu“.
- Následující *návrh* („design“) řešení je asi tou vůbec nejdůležitější fází (ano, mnohem důležitější, než následné psaní zdrojového kódu!), a právě zde je co nejlepší pochopení objektových principů rozhodující. V současnosti třeba není nikterak neobvyklé, že tradiční „strukturovaný“ (vizte níže) návrh vede k tomu, že programátor, jenž sám sebe považuje za programátora objektového, pouze používá technických prostředků objektových systémů – ale programy píše přesně podle hierarchických metod přístupu strukturovaného. Pak mnohdy také argumentuje, že objektové programování žádné zásadní výhody nepřináší – jemu samozřejmě nikoli, když jej ve skutečnosti nevyužívá!
- Vlastní *implementace* – často právě nazývaná „programováním“ v nejužším smyslu slova, to opravdické „psaní zdrojového kódu“ – už je do jisté míry nepodstatná. V této fázi je zapotřebí co nejelegantněji a nejefektivněji převést výsledky návrhu do prostředků konkrétního programovacího jazyka, ale již při ní nepadají žádná zásadní rozhodnutí. My si v knize ukážeme řadu konkrétních příkladů v několika různých objektových programovacích jazycích; hlavní těžiště však je právě v objektovém návrhu.

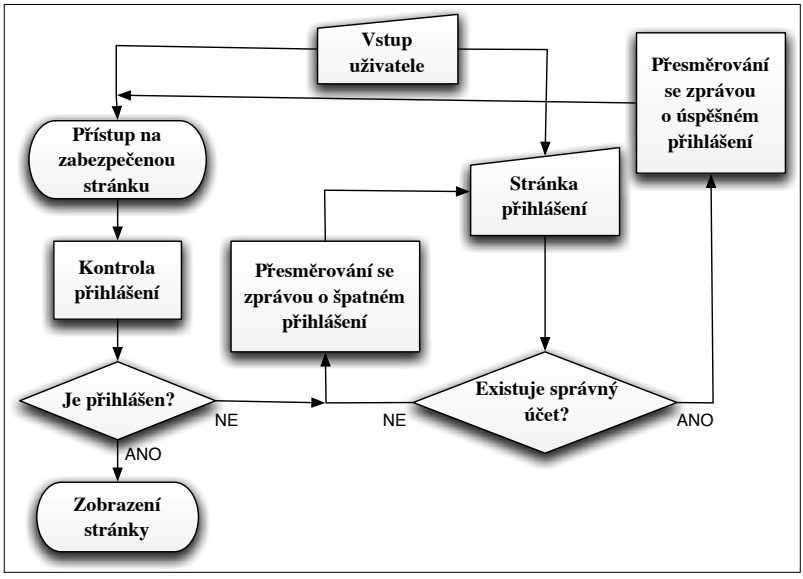
Je ovšem samozřejmě již při návrhu třeba počítat i s tím, jaký programovací jazyk – a jaké objektové knihovny – budou ve třetí, implementační fázi použity: to proto, že některé jazyky mohou do značné míry omezit volbu mechanismů, jimiž se při návrhu rozhodneme ten který problém řešit. Knihovny naopak mohou návrh značně usnadnit – máme-li např. k dispozici plnohodnotný systém distribuovaných objektů (jak si jej popíšeme v deváté kapitole), navrhují se objektové systémy založené na koncepci klient/server nesrovnatelně snáze, než bez něj.

1.2 Trocha historie nikoho nezabije

Ponecháme-li stranou počítačový pravěk, zahrnující přepojování drátů v Eniacu stejně jako prosvětlená tlačítka na čelních panelech zařízení poněkud mladších – ne že by nešlo o velmi zajímavé záležitosti, leč s naším tématem to pohříchu souvisí jen zcela okrajově – bude prvním milníkem, u nějž se chvilku zdržíme, programování *sekvenční*.

1.2.1 Sekvenční programování

Nejprve se programovalo prostě tak, že v patřičném kódu – ať již jím byl assembler, BASIC nebo FORTRAN – programátor psal sekvenci příkazů, jež počítač prováděl jeden po druhém. Existovaly samozřejmě vždy nějaké triky, jak sekvenci narušit a řízení předat jinam – jinak by to bylo zhola nepoužitelné; byly vždy ale velmi nepohodlné a z hlediska pozdějších úprav programu krajně nešikovné. V podstatě všechny byly založeny na jednoduchých podmínkách a přeskocích na jiné místo v oné výše zmíněné sekvenci příkazů – celé bychom si to pak mohli v grafické podobě představit zhruba tak, jak ukazuje obrázek 1.1.



Obrázek 1.1: Vývojový diagram přihlášení k webové aplikaci

Na něm vidíme tzv. *vývojový diagram*; v počátcích programování velmi oblíbený a dodnes – bohužel – ne zcela zapomenutý způsob, kterak zobrazovat algoritmy sekvenčního typu – i se všemi jejich nešvary.

S řešením těchto problémů přišlo tzv. programování *strukturované*.

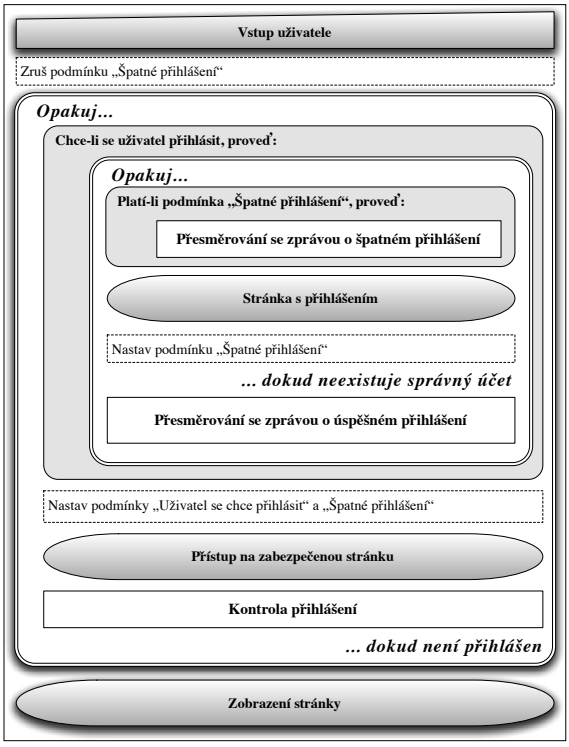
1.2.2 Strukturované programování

Záhy si programátoři uvědomili, že předávat řízení sem tam v jediné dlouhé sekvenci příkazů (nebo síti prvků vývojového diagramu, což je v principu totéž) je značně nepohodlné; zvláště pak dodatečné úpravy a změny takto sestaveného kódu jsou pravou noční můrou. Právě v dobách sekvenčního programování vzniklo rčení: „Každý program obsahuje alespoň jednu závažnou chybu; při její opravě vždy zavlečeme do programu nejméně dvě nové.“ I pokusili se tuto sekvenci rozbít a nahradit ji striktně hierarchickou strukturou „stromovitého“ typu.

V ní již neexistovalo „předání řízení jinam“ (a mnohé strukturované programovací jazyky skutečně zašly tak daleko, že odpovídající příkaz – obvykle pojmenovaný „go to“ – vůbec neobsahovaly); namísto toho bylo pouze možné využívat a vzájemně

skládat hotové stavební bloky typu „platí-li podmínka P, proveď vnořený blok A; jinak proveď vnořený blok B“, případně “dokud platí podmínka P, prováděj opakovaně vnořený blok A“.

Tyto vnořené bloky samy mohly být sestaveny z podobných modulů; tím nejjednodušším blokem pak byla prostá sekvence příkazů – sice stejná, jako v sekvenčním programování, ale bezproblémová, neboť neobsahovala žádné podmínky ani přeskoky. Ilustraci tohoto přístupu vidíme na obrázku 1.2.



Obrázek 1.2: Struktogram representující týž algoritmus

I přesto, že v této variantě – tzv. *struktogramu* – jsme se neobešli bez pomocných podmínek (to je známé zlo striktně strukturovaných algoritmů), je struktogram pořád oproti vývojovému diagramu poněkud přehlednější. Hlavní výhodou je to, že v něm nikde nejsou skoky „z jedné strany na druhou“: všechny jeho části jsou zcela jasně ohraničené a uzavřené samy v sobě. Nejinak tomu bylo i s programováním, založeným na obdobných principech.

Hlavní rozdíl oproti sekvencím ovšem spočívá v tom, že dodatečné úpravy strukturovaně napsaného systému jsou mnohem snazší a bezpečnější, s jen minimálním risikem vnesení nových chyb a problémů. Jednotlivé funkční bloky totiž mají jasně definované vstupy a výstupy, a pokud tuto úmluvu dodržíme, můžeme uvnitř bloku bezpečně provádět jakékoli úpravy, změny a zdokonalení.

Stojí za to si tento princip zapamatovat. Až si budeme níže popisovat objektový princip tzv. „zapouzdření“, uvidíme, že vlastně jde opět o totéž: ko-

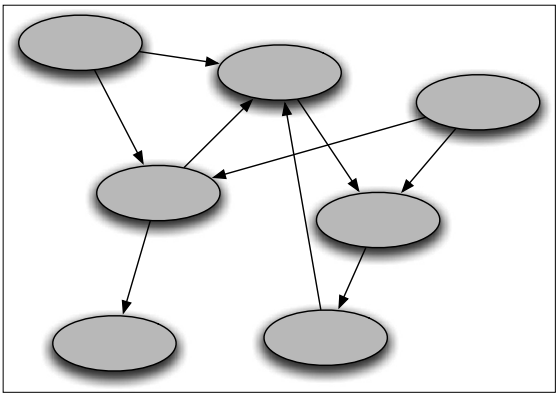
rektní definice „rozhraní“, jež zde hierarchický stavební blok, tam objekt, nabízí, spolu s nezávislostí tohoto „rozhraní“ na vnitřní implementaci, umožňuje bezpečné a bezproblémové dodatečné úpravy.

Strukturovaný přístup byl velmi životaschopný, a zvláště u programovacích jazyků, jež se nesnažily o dokonalou koncepční čistotu – asi nejlepším příkladem je slavný jazyk C –, sloužil skvěle desítky let. V podstatě slouží skvěle dodnes: i dnes je naprostá většina programovacích jazyků více či méně založena na principech strukturovaného programování a jeho hierarchicky vkládaných blocích.

Hlavní nevýhodou strukturovaného programování se ukázala být jeho rigidní hierarchie: ačkoli tak lze bezpochyby řešit velmi značné množství problémů, málo platné, reálný svět není hierarchický. Spíše je volnou sítí vzájemně komunikujících prvků, jež se někdy staví do částečných a nedokonalých hierarchií, jindy zase ne – podle momentální potřeby a možností. A právě modelem reálného světa je programování *objektové*.

1.2.3 Objektové programování

Objektové programování jde ještě o krůček dále než programování strukturované: to rozbilo striktní sekvenci příkazů do poněkud flexibilnější hierarchie, kdežto programování objektové rozbíjí i tuto hierarchii. Snaží se tedy v tomto smyslu nastavit zrcadlo reálnému světu: je založeno na volné síti vzájemně komunikujících *objektů*, jež se podle potřeby mohou – ale nemusí – stavět do libovolných struktur, včetně hierarchické. Takovou objektovou strukturu vidíme na obr. 1.3.



Obrázek 1.3: Síť vzájemně komunikujících objektů

Objektový přístup se prvně ve významnější míře objevil již před desítkami let v dnes již víceméně zapomenutém jazyce Simula 67; jen o pár let později byl excelentně implementován v nezapomenutelném systému Smalltalk. Od té doby jsou služby, ve větší či menší míře podporující objektové programování, k dispozici ve většině programovacích jazyků: některé z nich jsou skvělé, jiné naopak objektovému programování spíše brání (odstrašujícím příkladem zde může být velmi nešťastně navržený „objektový“ systém jazyka C++).

Poměrně značné množství literatury a zdrojů, věnovaných objektovému přístupu, omílá mantru „dědičnost, zapouzdření, polymorfismus“. Nevěřte jim! Jedná se sice o velmi důležité principy – a také se jednomu každému z nich budeme v této knize

ještě dost podrobně věnovat –, ale žádný z nich (snad do jisté míry vyjma polymorfismu) a ani všechny dohromady nejsou tím základním a hlavním principem, na němž je idea objektového programování založena: tím je právě a především

- volná síť vzájemně komunikujících objektů.

Vše ostatní jsou doplňky a služby, zvyšující pohodlí programátora (dědičnost) či bezpečnost a spolehlivost celého objektového systému (zapouzdření).

Podobně jako strukturované programování do jisté míry zachovalo sekvence příkazů – jako elementární bloky ve své hierarchii –, ani objektový přístup nezavrhuje hierarchické struktury: již jsme se ostatně zmínili o tom, že většina současných programovacích jazyků – včetně těch objektových – do jisté míry strukturovaný přístup podporuje. V objektových systémech nalézáme hierarchii hned na několika místech:

- jako přímé dědictví strukturovaných programovacích jazyků obvykle objektové systémy využívají tradičního „strukturovaného“ kódu pro popis konkrétního chování objektů. Typický případ je, že programátor stanoví „dostane-li tento objekt takovou a takovou zprávu, provede se následující úsek programu“;
- sama objektová struktura – přinejmenším u složitějších aplikací – bývá hierarchická: to, co na jedné úrovni abstrakce lze representovat jediným objektem, je při podrobnějším pohledu ve skutečnosti celá samostatná „vnořená“ síť objektů, jež spolupracují na implementaci celkové úlohy;
- navíc se objevují nové hierarchie, dané technickými prostředky toho kterého objektového systému – zcela typickou zde bývá hierarchie tříd podle dědičnosti.

1.2.4 A co dále, po objektech?

Je zajímavá otázka, zda vůbec a ano-li, kdy se objeví další koncepce, odlišná od současného objektového pohledu.

Možné je samozřejmě cokoli; prozatím tomu ale nic nenasvědčuje, a podle osobního názoru autora této knihy to ani nenastane (leda až se objeví skutečně universálně použitelný deklarativní systém, založený na umělé inteligenci) – objektové systémy se samozřejmě budou zdokonalovat a měnit, avšak jejich základní paradigma již zůstane bez principiálních změn.

To proto, že objektové programování je přímým modelem reálného světa; zdá se proto celkem pravděpodobné, že nelze vymyslet nic principiálně lepšího.

Z koncepčního hlediska je do jisté míry sporné, zda objekty programovacích jazyků skutečně *mají* modelovat objekty reálného světa; protiargumenty v zásadě tvrdí, že počítačové systémy řeší problémy odlišného typu, než s jakými se setkáváme v realitě. Praxe nicméně je taková, že (a) objektové systémy byly pro simulaci původně navrženy a do značné míry reálnému světu odpovídají, (b) ty, jež tak činí, slouží programátorům v průměru lépe, než prostředí, jež se reálnému světu podobají méně.

1.3 Jiná rozlišovací kritéria

Na konci této úvodní kapitolky je snad vhodné se zmínit o tom, že vedle výše uvedeného přístupu sekvenčního, strukturovaného či objektového existuje řada dalších kritérií,