

Objektové programování

naučte se pravidla
objektového myšlení

Ondřej Čada

- ❑ Základy objektového pohledu na programování
- ❑ Shrnutí rozdílů mezi běžnými objektovými jazyky
- ❑ Základní paradigmatata objektového designu
- ❑ Distribuované objekty a vzdálené zpracování



Upozornění pro čtenáře a uživatele této knihy

Všechna práva vyhrazena. Žádná část této tištěné či elektronické knihy nesmí být reprodukována a šířena v papírové, elektronické či jiné podobě bez předchozího písemného souhlasu nakladatele. Neoprávněné užití této knihy bude **trestně stíháno**.

Používání elektronické verze knihy je umožněno jen osobě, která ji legálně nabyla a jen pro její osobní a vnitřní potřeby v rozsahu stanoveném autorským zákonem. Elektronická kniha je datový soubor, který lze užívat pouze v takové formě, v jaké jej lze stáhnout s portálu. Jakékoliv neoprávněné užití elektronické knihy nebo její části, spočívající např. v kopírování, úpravách, prodeji, pronajímání, půjčování, sdělování veřejnosti nebo jakémkoliv druhu obchodování nebo neobchodního šíření je zakázáno! Zejména je zakázána jakákoliv konverze datového souboru nebo extrakce části nebo celého textu, umístování textu na servery, ze kterých je možno tento soubor dále stahovat, přitom není rozhodující, kdo takovéto sdílení umožnil. Je zakázáno sdělování údajů o uživatelském účtu jiným osobám, zasahování do technických prostředků, které chrání elektronickou knihu, případně omezují rozsah jejího užití. Uživatel také není oprávněn jakkoliv testovat, zkoušet či obcházet technické zabezpečení elektronické knihy.





Copyright © Grada Publishing, a.s.

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

V knize použité názvy programových produktů, firem apod. mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

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)
© Grada Publishing, a.s. 2011

Obsah

Úvod	13
1 Základy objektového pohledu	15
1.1 Programování je co?	16
1.2 Trocha historie nikoho nezabije	16
1.2.1 Sekvenční programování	17
1.2.2 Strukturované programování	17
1.2.3 Objektové programování	19
1.2.4 A co dále, po objektech?	20
1.3 Jiná rozlišovací kritéria	20
1.3.1 Události nebo stavy?	21
1.3.2 Interpret nebo překladač?	21
1.3.3 Malá odbočka k orákulu	22
2 Z čeho a jak objekty stavět	23
2.1 Vlastnosti objektů reálného světa	24
2.2 Komunikace mezi objekty	25
2.3 Implementace objektů v počítačovém systému	25
2.3.1 Třídy	26
2.3.2 Je třída objekt?	26
2.3.3 Instance	27
2.3.4 Metařídy	27
2.3.5 Instanční proměnné a metody	27
2.3.6 Zprávy	29
2.4 Slavné pojmy	30
2.4.1 Dědičnost	31

2.4.2	Zapouzdření	32
2.4.3	Polymorfismus	34
2.4.4	Polymorfismus konkrétních služeb	35
3	Ze života objektu	37
3.1	Typy objektů podle doby jejich života	37
3.2	Dynamické objekty	38
3.2.1	Podpora přímo v jazyku	38
3.2.2	Podpora na úrovni tříd	39
3.2.3	Automatická správa paměti	39
3.3	Automatické objekty	41
3.4	Statické objekty	42
3.5	Trvalé objekty	43
4	Objekty a typy	47
4.1	Typy	47
4.1.1	Typy typů	48
4.1.2	Pozdní vazba	49
4.2	Typy objektů	50
4.2.1	Objekty jsou dynamické	51
4.2.2	Třída však může být typem	51
4.2.3	Ukaž, co umíš...	52
4.3	Jak poznat kachnu?	53
5	Objektový návrh	57
5.1	Objekty mají chování, nejen data!	57
5.1.1	Klasický neobjektový návrh	58
5.1.2	Objektový návrh	61

5.2 MVC	63
5.2.1 Model	64
5.2.2 Vzhled	64
5.2.3 Řízení	65
5.3 Ukaž co umíš – podruhé	66
5.3.1 Zjištění třídy objektu	66
5.3.2 Pozor na „řádoby polymorfni“ metody	68
5.4 Osamělec	69
5.5 Abstraktní a sdružené třídy	70
5.5.1 Abstraktní třídy	71
5.5.2 Sdružené třídy	71
5.5.3 Vícenásobné rozhraní	73
5.5.4 Primitivní metody	74
5.6 Notifikace	75
5.6.1 Princip funkce	75
5.6.2 Jednoduchá ukázka	76
5.6.3 Princip použití	78
5.7 Klonování	78
5.7.1 Jednoduché klonování instancí	79
5.7.2 Měnitelné a neměnné objekty	80
5.7.3 Klonování objektových sítí	82
5.8 Inicializace na vyžádání	82
5.9 Jak se vyhnout dědictví	84
5.9.1 Rozšíření tříd	85
5.9.2 Delegace	86
5.9.3 Mechanismus akce/cíl	88
5.9.4 Mám kachnu a nebojím se ji použít!	89
6 Problémy objektového návrhu	91

6.1 Nesnesitelná křehkost dědění	92
6.1.1 Technické příčiny	92
6.1.2 Principiální problémy	92
6.1.3 Řešení	94
6.2 Je kružnice elipsou, nebo elipsa kružnicí?	95
6.2.1 Implementační hledisko	95
6.2.2 Strukturální hledisko	98
6.2.3 Řešení	99
6.3 Objekt všemohoucí	101
6.4 Příliš mnoho tříd umožnilo programátora... ..	101
7 Zástupné objekty	103
7.1 Základy	104
7.1.1 Vkládání objektů	104
7.1.2 Přesměrování zpráv	104
7.1.3 Využití zástupných objektů	106
7.1.4 Vícenásobná dědičnost	107
7.2 Záchytné objekty	108
7.2.1 Nejjednodušší varianta	109
7.2.2 Omezené služby ihned	111
7.2.3 „Sebenahrazující se“ objekty	113
7.3 Budoucí objekty	119
7.3.1 Prostředky Objective C pro spolupráci s procesy	120
7.3.2 Implementace budoucího objektu	123
8 Objektové vazby	125
8.1 Pojmenované atributy	126
8.1.1 Atributy	126
8.1.2 Přístup k atributům podle jména	127

8.1.3	Využití pojmenovaných atributů	128
8.1.4	Pojmenované relace 1:N...	129
8.1.5	...a přístup k jejich prvkům	131
8.1.6	Řazení jmen	132
8.1.7	Agregační funkce	133
8.1.8	Ověření správnosti atributů	135
8.2	Sledování změn objektů	135
8.2.1	Princip funkce	136
8.2.2	Pohled pozorovatele	136
8.2.3	Pohled modelu	137
8.3	Řídicí objekty a objektové vazby	138
8.3.1	Kam se nám ztratilo řízení?	139
8.3.2	Postavení řídicího objektu	139
8.3.3	Standardní řídicí objekty	141
8.3.4	Základní princip implementace	142
8.4	Podpora vazeb v GUI	143
8.4.1	Přístup k vazbám	144
8.4.2	Převodníky hodnot	144
8.4.3	Zástupné hodnoty	145
9	Distribuované objekty	147
9.1	Jednoduchá ukázka	148
9.1.1	Rozhraní služebního objektu	148
9.1.2	Implementace	149
9.1.3	Použití	150
9.1.4	Distribuované objekty	151
9.1.5	Server	151
9.1.6	Klient	152
9.2	Jak to celé funguje?	152
9.2.1	Zástupný objekt na každém rohu...	152

9.2.2	ORB	154
9.2.3	Vyhledání partnerů	155
9.2.4	Prvotní navázání spojení	156
9.3	Speciální případy a triky	157
9.3.1	Předávání neobjektových dat	157
9.3.2	Předávání objektů	158
9.3.3	Využití rozhraní či protokolu pro vyšší efektivitu	159
10	Nepřímé zasílání zpráv	161
10.1	O co se jedná?	162
10.1.1	Základy	162
10.1.2	Nepřímé zasílání zpráv a pole	163
10.1.3	Další služby pro práci s polem	165
10.1.4	Řetězení metazpráv i běžných zpráv	166
10.1.5	Zástupný objekt	168
10.1.6	Hlídaní přístupových práv	169
10.1.7	Ošetření výjimek	170
10.1.8	Ověření, zda příjemce zprávu akceptuje	170
10.1.9	Paralelní zpracování	171
10.2	Vnitřní mechanismus nepřímého zasílání zpráv	173
10.2.1	Časovač	173
10.2.2	Pole	174
10.2.3	Řetězení zpráv	174
10.2.4	Ostatní	175
10.3	Ukázka jednoduché implementace	176
10.3.1	Rozsah služeb a rozhraní	176
10.3.2	Implementace metazpráv	177
10.3.3	Zástupné objekty metazpráv	178
10.3.4	Zástupný objekt „each“	179
10.3.5	Zástupný objekt „afterDelay:“	180
10.3.6	Zástupný objekt „collect“	181

10.3.7 Zástupný objekt „selectToArray:“	182
10.3.8 Zástupný objekt „ifResponds“	184
10.4 Simulace bloků	184
10.4.1 Použití	185
10.4.2 Implementace	186
Slovníček zkratk a pojmů	187
Rejstřík	199

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 knihovнами 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 zaslání zpráv, kde nabízíme i plně funkční ukázkou 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 hezký a čistě navržený systém s plně dynamickou objektovou podporou. Snad jedinou chybou *Ruby* je trochu nešťastný standard pojmenovávání běžně užívaných metod jeho knihovnických 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 knihovnických 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 něž 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.

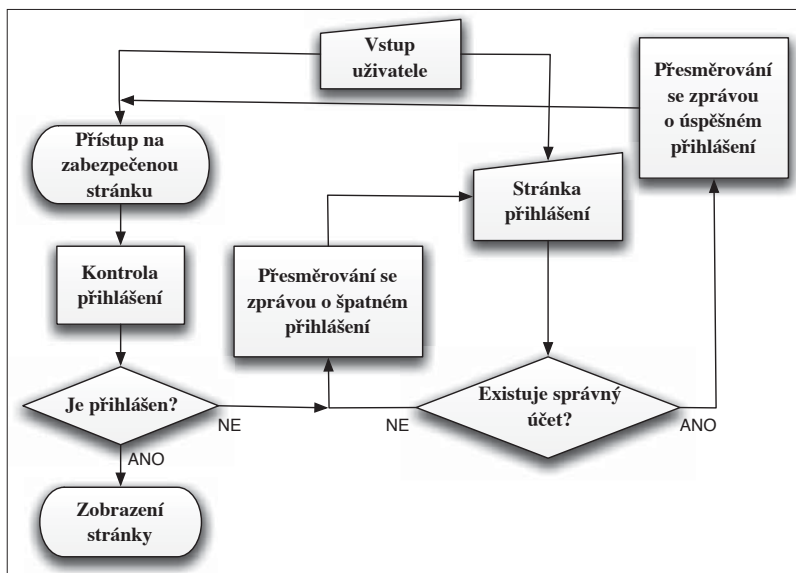
Je ovšem samozřejmě již při návrhu třeba počítat i s tím, jaký programovací jazyk – a jaké objektové knihovny – budou ve třetí, implementační fázi použity: to proto, že některé jazyky mohou do značné míry omezit volbu mechanismů, jimiž se při návrhu rozhodneme ten který problém řešit. Knihovny naopak 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 něž se chvilku zdržíme, programování *sekvenční*.

1.2.1 Sekvenční programování

Nejprve se programovalo prostě tak, že v patřičném kódu – at 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 přeskokách 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*.

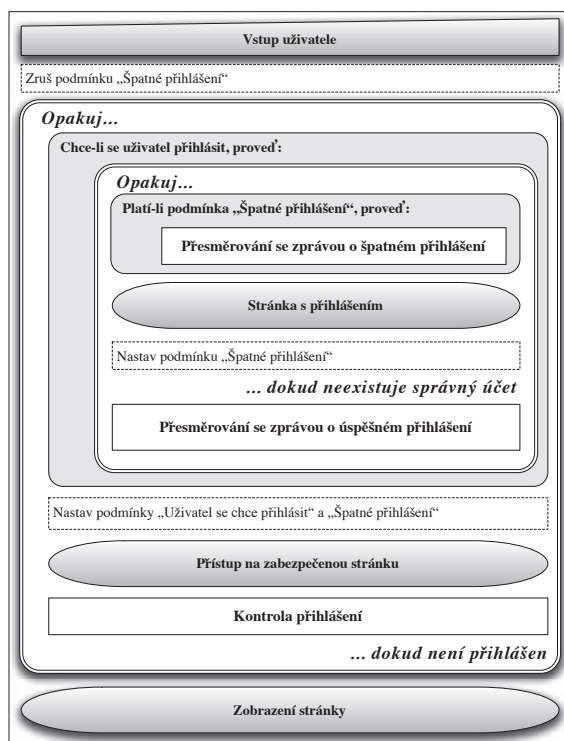
1.2.2 Strukturované programování

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 pracnou 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, proved vnořený blok A; jinak proved 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 sekvencním programování, ale bezproblémová, neboť neobsahovala žádné podmínky ani přeskočky. Ilustraci tohoto přístupu vidíme na obrázku 1.2.



Obrázek 1.2: Struktogram reprezentující tyž 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 rizikem 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 objektivý princip tzv. „zapouzdření“, uvidíme, že vlastně jde opět o totéž: ko-

1. Základy objektivého pohledu

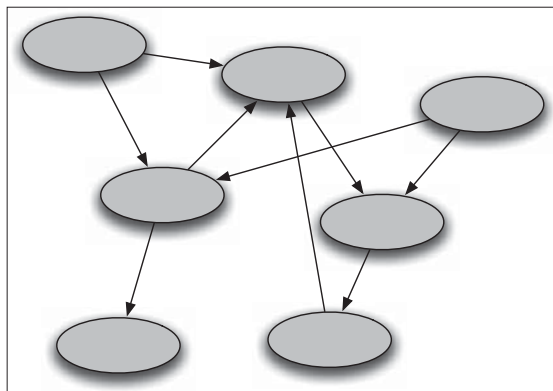
reální definice „rozhraní“, jež zde hierarchický stavební blok, tam objekt, nabízí, spolu s nezávislostí tohoto „rozhraní“ na vnitřní implementaci, umožňuje bezpečně a bezproblémově dodatečně úpravy.

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 reprezentovat 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 nenásvědčuje, 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 kapitoly je snad vhodné se zmínit o tom, že vedle výše uvedeného pohledu sekvencí, strukturovaného či objektového existuje řada dalších kritérií,

1. Základy objektového pohledu

podle nichž lze programování dělit; ačkoli pro naši knihu tyto rozdíly nejsou podstatné, vyplatí se ukázat si několik nejčastěji užívaných dělicích kritérií malinko blíže.

1.3.1 Události nebo stavy?

Při programování aplikací existují obecně dvě základní koncepce:

- *stavové* programování: při něm je systém naprogramován tak, aby podle potřeb a vnějších požadavků přecházel mezi různými stavy; v každém z nich nabízí nějaké tomuto stavu odpovídající funkce;
- naproti tomu při *událostmi řízeném* programování je systém (v principu) neustále v téže stavu – totiž „čekání na následující událost“; jakmile se dočká, obslouží ji způsobem, který dané události odpovídá, a opět čeká na další.

Prvý si – ve velmi hrubém přiblížení – můžeme představit jako firmu, jež má několik zástupců, z nichž každý mluví jedním cizím jazykem. Je-li zrovna u okénka ten, který umí anglicky, nemá německy mluvící zákazník šanci; může se jediné pokusit o *změnu stavu*, tj. vyžádat si změnu zástupce.

V takovéto analogii pak bude modelem událostmi řízeného systému zástupce, který umí více jazyků: mohou na něj na přeskáčku mluvit zákazníci anglicky, německy, španělsky i řecky, a on bude s to každý požadavek korektně obsloužit, aniž by kvůli tomu bylo zapotřebí „měnit stav“.

Je poměrně zřejmé, že – v naprosté většině případů – je druhý přístup flexibilnější, a proto se také v současnosti běžně užívá (hovoříme už zase o tvorbě aplikací pro samočinné počítače, nikoli o jazykových schopnostech cestáků). S objektovým či neobjektovým programováním to ale nemá zhora nic společného – být je asi pravda, že s využitím objektových prostředků je implementace událostmi řízeného systému poněkud snazší než bez nich.

1.3.2 Interpret nebo překladač?

Programovací jazyky jsou v zásadě dvou typů:

- *interpretované*, kde existuje speciální aplikace, jež čte náš zdrojový program a rovnou jej provádí (tedy jej „interpretuje“);
- *překládané*, kde existuje jiná speciální aplikace, jež náš zdrojový program „přeloží“ – totiž převede do formy, již dokáže interpretovat přímo počítač, samotný jeho hardware (tedy konkrétně procesor).

Interpretované jazyky tedy nelze použít tam, kde není k dispozici interpret; překládané naopak – po překladači – mohou pracovat kdekoli.

V praxi nejsou rozdíly příliš podstatné; kromě jiného také proto, že řada moderních interpretů jazyk průběžně překládá; velmi běžné jsou také kombinované přístupy, kdy je nutný překladač – stejně jako ve druhém případě –, jenž ale nepřekládá do formy vhodné přímo pro hardware, nýbrž do speciální pomocné formy, již jiný speciální program interpretuje (zcela typickým zástupcem tohoto přístupu je Java).

Opět rozlišení na překládané a interpretované jazyky nikterak nesouvisí s tím, zda jde o objektový či neobjektový systém; objektové jazyky však vzhledem k obecně větší flexibilitě obvykle mívají i v „překládané“ podobě jisté rysy systému interpretovaného.