



Rebecca M. Riordan

# Vytváříme relační databázové aplikace



- Teorie i praxe návrhu relačního databázového systému
- Normalizace, tabulky, relace, dotazy, aplikací logika, uživatelské rozhraní
- Příklady v Microsoft Accessu a SQL Server

Rebecca M. Riordan

# Vytváříme relační databázové aplikace

---

Computer Press  
Praha  
2000



Rebecca M. Riordan

# Vytváříme relační databázové aplikace



- Teorie i praxe návrhu relačního databázového systému
- Normalizace, tabulky, relace, dotazy, aplikační logika, uživatelské rozhraní
- Příklady v Microsoft Accessu a SQL Serveru



VŠECHNY CESTY  
K INFORMACÍM

**Microsoft®**

# Vytváříme relační databázové aplikace

Rebecca M. Riordan

Copyright © Computer Press® 2000. Vydání první. Všechna práva vyhrazena.  
Vydavatelství a nakladatelství Computer Press®,  
Hornocholupická 22, 143 00 Praha 4, <http://www.cpress.cz>

ISBN 80-7226-360-9

Prodejní kód: K0393

**Překlad:** David Krásenský

**Jazyková korektura:** Rostislav Brzobohatý

**Vnitřní úprava:** Tomáš Doležal

**Sazba:** Tomáš Doležal

**Rejstřík:** Pavlina Bauerová

**Obálka:** Petr Chládek

**Komentář na zadní straně obálky:** Ivo Magera

**Technická spolupráce:** Jiří Matoušek, Petr Klíma

**Odpovědný redaktor:** Ivo Magera

**Vedoucí technické redakce:** Martin Hanslian

**Vedoucí knižní redakce:** Ivo Magera

**Vedoucí produkce:** Kateřina Vobecká

Authorized translation from English language edition Designing Relational Database Systems.

Original copyright: © Microsoft Press/Rebecca M. Riordan, 1999.

Translation: © Computer Press, 2000.

Autorizovaný překlad z originálního anglického vydání Designing Relational Database Systems.

Originalní copyright: © Microsoft Press/Rebecca M. Riordan, 1999.

Překlad: © Computer Press, 2000.

**Žádná část této publikace nesmí být publikována a šířena žádným způsobem a v žádné podobě bez výslovné svolení vydavatele.**

**Veškeré dotazy týkající se distribuce směřujte na:**

**Computer Press Brno**, náměstí 28. dubna 48, 635 00 Brno-Bystrc,  
tel. (05) 46 12 21 11, e-mail: [distribuce@cpress.cz](mailto:distribuce@cpress.cz)

**Computer Press Bratislava**, Hattalova 12/A, 831 03 Bratislava, Slovenská republika,  
tel.: +421 (7) 44 45 20 48, e-mail: [distribucia@cpress.sk](mailto:distribucia@cpress.sk)

Nejnovější informace o našich publikacích naleznete na adrese: <http://www.cpress.cz/knihy/bulletin.html>.  
Máte-li zájem o pravidelné zasílání bulletinu do Vaší e-mailové schránky, zašlete nám jakoukoli, i prázdnou zprávu na adresu [bulletin@cpress.cz](mailto:bulletin@cpress.cz).



<http://www.vltava.cz>

Nejširší nabídka literatury, hudby, MP3,  
multimediálního softwaru a videa za  
bezkonkurenční ceny.



Vaše dotazy, vzkazy, náměty, připomínky ke knižní produkcii  
Computer Press přijímá 24 hodin denně naše horká linka:  
[knihy@cpress.cz](mailto:knihy@cpress.cz)

# OBSAH

PŘEDMLUVA . . . . .	ix
PODĚKOVÁNÍ . . . . .	xii
ÚVOD . . . . .	xiii

## 1. část Teorie relačních databází

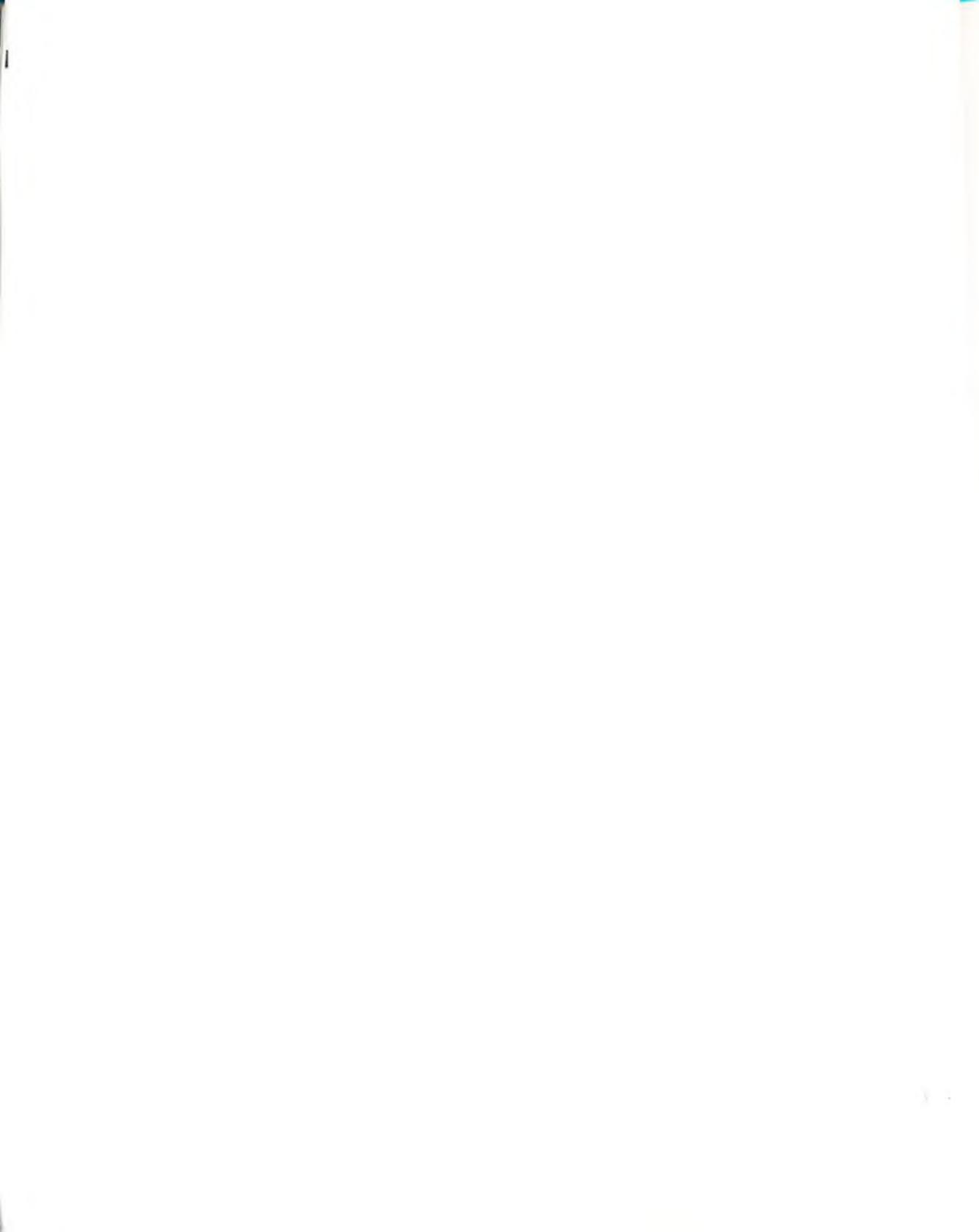
<b>1. ZÁKLADNÍ POJMY</b>	<b>3</b>
Co je to databáze? . . . . .	4
Databázové nástroje . . . . .	6
Relační model . . . . .	8
Terminologie relačních databází . . . . .	10
Datový model . . . . .	11
<b>2. STRUKTURA DATABÁZÍ</b>	<b>23</b>
Základní principy . . . . .	27
První normální forma . . . . .	32
Druhá normální forma . . . . .	34
Třetí normální forma . . . . .	35
Další normalizace . . . . .	37
<b>3. VZTAHY A RELACE</b>	<b>43</b>
Terminologie . . . . .	43
Modelování vztahů . . . . .	45
Vztahy typu jedna k jedné . . . . .	47
Vztahy typu jedna k více . . . . .	51
Vztahy typu více k více . . . . .	52
Unární vztahy . . . . .	53
Ternární vztahy . . . . .	54
Vztahy s předem známou kardinalitou . . . . .	57
<b>4. DATOVÁ INTEGRITA</b>	<b>59</b>
Omezení integrity . . . . .	59
Implementace datové integrity . . . . .	66
<b>5. RELAČNÍ ALGEBRA</b>	<b>77</b>
Hodnoty Null a třihodnotová logika (ještě jednou) . . . . .	78
Relační operátory . . . . .	79
Množinové operátory . . . . .	85
Speciální relační operátory . . . . .	89

## 2. část Návrh relačních databázových systémů

<b>6. PROCES NÁVRHU</b>	<b>97</b>
Modely životního cyklu .....	.97
Proces návrhu databáze .....	.101
Poznámka o metodologických a standardech návrhu .....	.102
<b>7. DEFINICE PARAMETRŮ SYSTÉMU</b>	<b>103</b>
Stanovení cílů systému .....	.103
Sestavení kritérií návrhu .....	.107
Určení záběru systému .....	.111
<b>8. DEFINICE PRACOVNÍCH PROCESŮ</b>	<b>115</b>
Vymezení stávajících pracovních procesů .....	.116
Analýza pracovních procesů .....	.120
Dokumentace pracovních procesů .....	.122
Uživatelské scénáře .....	.124
<b>9. MYŠLENKOVÝ DATOVÝ MODEL</b>	<b>127</b>
Identifikace datových objektů .....	.127
Definice vztahů .....	.132
Analýza oborů hodnot .....	.139
Normalizace .....	.141
<b>10. DATABÁZOVÉ SCHÉMA</b>	<b>143</b>
Architektury systémů .....	.143
Součásti databázového schématu .....	.157
Bezpečnost .....	.160
<b>11. PŘEDNESENÍ NÁVRHU</b>	<b>165</b>
Komu je sdělení určeno a jeho význam .....	.165
Struktura dokumentu .....	.166
Stručné shrnutí pro vedoucí pracovníky .....	.166
Přehled systému .....	.168
Pracovní procesy .....	.169
Myšlenkový datový model .....	.170
Databázové schéma .....	.171
Uživatelské rozhraní .....	.171
Speciální nástroje .....	.174

### **3. část Návrh uživatelského rozhraní**

<b>12. ROZHRANÍ JAKO DŮLEŽITÝ PROSTŘEDNÍK</b>	<b>175</b>
Efektivní rozhraní .....	175
Modely rozhraní .....	178
Úrovně uživatelů .....	180
Jak zapojit uživatele .....	181
Minimalizace zatižení paměti .....	183
Jak být konzistentní .....	184
<b>13. ARCHITEKTURY UŽIVATELSKÉHO ROZHRANÍ</b>	<b>187</b>
Podpora pracovních procesů .....	187
Architektury dokumentů .....	189
<b>14. REPREZENTACE ENTIT V NÁVRHU FORMULÁŘU</b>	<b>199</b>
Jednoduché entity .....	199
Relace typu jedna k jedné .....	202
Relace typu jedna k více .....	202
Hierarchie .....	206
Relace typu více k více .....	208
<b>15. VÝBĚR VHODNÝCH OVLÁDACÍCH PRVKŮ WINDOWS</b>	<b>211</b>
Reprezentace logických hodnot .....	212
Reprezentace množiny hodnot .....	213
Reprezentace číselných a datumových údajů .....	218
Reprezentace textových hodnot .....	220
<b>16. ZACHOVÁNÍ DATOVÉ INTEGRITY</b>	<b>223</b>
Třídy omezení integrity .....	224
Vestavěná omezení .....	225
Aplikační omezení .....	231
<b>17. VYTVAŘENÍ SESTAV</b>	<b>235</b>
Třídění, vyhledávání a filtrování dat .....	236
Vytváření standardních sestav .....	240
Vytváření ad hoc sestav .....	245
<b>18. NA POMOC UŽIVATELŮM</b>	<b>251</b>
Pasivní mechanismy pomoci .....	253
Reaktivní mechanismy pomoci .....	257
Proaktivní pomoc .....	262
Školení uživatelů .....	263
<b>SLOVNÍČEK POJMŮ</b> .....	<b>265</b>
<b>LITERATURA</b> .....	<b>273</b>
<b>O AUTORCE</b> .....	<b>275</b>
<b>REJSTŘÍK</b> .....	<b>277</b>



# PŘEDMLUVA

Konečně se vám dostává do rukou kniha, která překlenuje propast mezi akademickou teorií databází a skutečným vytvářením databází v reálném světě. Tato knížka současně podává základní informace, které budete při práci s databázemi potřebovat – od návrhu až po implementaci – najdete zde tedy nejnovější aktuální rady pro rychlé vytváření dokumentace k databázovým projektům, pro návrh grafického uživatelského rozhraní databázových aplikací a pro školení uživatelů a dokumentaci.

Jestliže při své práci vytváříte skutečné databáze z reálného světa, které řeší skutečné problémy z reálného světa, najdete zde podrobné, přesné a velice praktické informace o umění tvorby databázových řešení. Součástí knihy je i řada reálných příkladů, postavených na běžných implementacích jazyka SQL. Po knize zajisté sáhnou i ostřlení databázoví „borci“, kteří již „všechno ví“ – pro ně je velice dobrým osvězením paměti a připomenutím aktuální podoby teorie a praxe.

Napadlo vás někdy třeba, co je to takové theta-spojení tabulek? Nebo kde se asi používá? Jestliže jste nikdy neabsolvovali teoretický základ formálního školení o databázích a teď přemýšlite, co vám asi tak uniklo, nebo jestliže chcete jenom svoje spolupracovníky ohromit přesnými teoretickými znalostmi z oblasti databází, pak vězte, že tato kniha obsahuje i velké množství praktických informací a technik, které vyplývají z relační teorie – tedy informací, o jaké jste nezavadili třeba i během několika let praktických zkušeností.

Absolvovali jste pečlivou, akademickou a teoretickou přednášku, jejíž látku byste ale nyní rádi aplikovali na příklady z reálného světa? Příští týden má proběhnout převod podnikové databáze do nového prostředí a databázový systém, který vám byl navržen, neimplementuje operátor dělení relací, jejž ovšem nutně potřebujete pro jistou klíčovou sestavu. Zde v knize přesně zjistíte, co různé oblíbené implementace jazyka SQL podporují a nepodporují, a naučíte se postupy pro překonání oné „propasti“ mezi teorií a praxí.

Zabezpečit, že hotový návrh systému skutečně řeší zadaný problém z reálného světa, je často mnohem obtížnější a také důležitější než samotná specifika dané implementace. Druhá část knihy proto vychází z bohatých zkušeností, které Rebecca načerpala z výstavby kriticky důležitých databázových aplikací pro různé zákazníky po celém světě. Najdete zde tudíž praktické informace a zásady pro práci se zákazníky, které aplikaci pomohou, aby vyhovovala jejich skutečným potřebám a nikoli tomu, co si zákazník myslí, že potřebuje. Kniha tak zaplňuje důležitý prostor mezi návrhem a implementací aplikace a porovnává například výhody a nevýhody jednovrstvé, dvouvrstvé a n-vrstvé architektury, jakož i nejnovějších inovací z internetového prostředí. Dále hovoří o administraci, bezpečnosti a auditu systému; všechny tyto otázky se až příliš často podceňují a při řešení projektu se odkládají, až je příliš pozdě.

Kniha ale nekončí u pouhého návrhu databází. Najdete zde i praktické a pragmatické rady pro správu životního cyklu projektu (dozvítě se i o jiných často skloňovaných problémech) a pro vývoj a přednesení neustále se vyvíjejícího návrhu uživatelům a programátorům. Dále kniha obsahuje výklad různých moderních uživatelských rozhraní, určených pro databázové aplikace, společně s rozbořem jejich pro a proti; ukáže vám, jak se určité ovládací prvky systému Microsoft Windows dobře mapují na konkrétní datové typy. Kapitola 16 je jedním z nejvíce realistických a pragmatických pojednání zaměřených na uživatele, jaké jsem kdy viděl. Z tohoto textu doslova září autorčiny bohaté zkušenosti z oblasti výstavby databázových aplikací. A jako poslední, avšak neméně významné téma kniha vynikajícím způsobem rozebírá možnosti pomoci a zaškolení uživatelů pro práci s hotovou aplikací; na pomoc si bere jak moderní techniky (například bublinová nápověda, ToolTips), tak i staré známé, vyzkoušené postupy, jako je akustická kontrola (zvukové upozornění) a horké klávesy.

Podtrženo a sečteno, tato kniha je praktické, pragmatické a znalostmi nabité pojednání o vytváření reálných databázových aplikací v dnešním překotném světě vývoje softwaru a je napsána zkušenou autorkou, která čerpá ze svých vlastních zkušeností. Smysl má proto pro každého, kdo se pokouší vytvářet nějaké databázové aplikace. Poučení však přináší i těm, kteří chtějí přenést teorii databázových systémů do praxe. A abych nezapomněl – kniha je napsána zábavným, osvěžujícím stylem, díky němuž se opravdu snadno a příjemně čte – to je něco, co bych si u nějaké knihy o databázích v životě nedokázal představit!

*Michael Mee*

# PODĚKOVÁNÍ

Na titulní stránce této knihy je sice uvedeno moje jméno, nedá se ale říci, že by ji napsal jeden jediný člověk – během jejího vzniku ji podporovalo tolik lidí, že by s přehledem zaplnili jednu menší vesnici. Bez pomoci těchto lidí by se kniha nejenže neměla šanci dostat na pulty knihkupectví, díky nim je ale navíc mnohem lepší, než jaká by byla bez přispění jejich obrovské laskavosti.

Takže teď hezky postupně vyjádřím svoji vděčnost těmto lidem:

Na prvním místě jmenuji svoji rodinu: je to můj manžel Mark Riordan, který je z nějakého záhadného důvodu neustále přesvědčen, že jsem prostě schopna udělat cokoli; dále můj otec Harlow Wright, který mi pomáhal dodávat potřebnou sebedůvštu; a ze všeho nejvíce děkuji své manince, Dianě Wrightové – díky ní a díky jejímu zdravému nadhlídku jsem dokázala brát všechno s úsměvem.

Mike Mee mi jako první vnuknul myšlenku na sepsání knížky a po jejím dokončení byl tak laskavý a napsal předmluvu.

Eric Stroo dokázal zachovat ledový klid i vůči takovému chaotickému a strašlivě nedisciplinovanému, poprvé příscímu autorovi, jako jsem byla já.

A toto jsou lidé, kteří měli odvahu čist moje rukopisy již v prvních fázích prací na knize a dodali do nich svoje bohaté zkušenosti: Dev Ashish, Jim Ferguson, Kim Jacobson a Annette Marcová.

Alice Turnerová, moje skvělá editorka, pochopila, co jsem vlastně tím či oním textem myslela, uvedla mé nejasné úvahy na pravou míru, dala mým zamotaným materiálům pevnou strukturu a řád, a přes to všechno, nad rámec svých povinností, mi dodávala silnou morální podporu.

Rob Nance dokázal ze stohu načívaných skic doslova vykouzlit smysluplné ilustrace.

Keri Hardwick mě zachránil za pět minut dvanáct. Přítel v nouzi...

Kromě toho děkuji pravidelným přispěvatelům internetové diskusní skupiny comp.databases.ms-access, jejichž trpělivost, velkorysost a technická genialita mě neustále ohromují.

A nakonec se k něčemu dobrovolně přiznám: najdete-li v knize nějaké chyby, pak vězte, že ty jsou bezvýhradně MOJE.

*Děkuji vám všem.  
Rebecca Riordanová*

U

R  
k  
a  
t  
c  
c

I  
:

# ÚVOD

Relační databáze jsou docela zálužná stvoření. Pochopit ostatní typy komerčního softwaru je nekonečně snazší. Textové procesory jsou ve skutečnosti jenom jakési hodně chytré psací stroje, a každému je nad slunce jasnější, že díky klávesce Backspace můžeme zapomenout na ty různé bílé opravné laky. Tabulkové procesory nám předkládají také jakýsi způsob myšlení, který je všem důvěrně známý – tedy nejen účetním; elektronická pošta neboli e-mail je zase pro pochopení modelu téměř shodná se systémem klasické papírové pošty.

Databáze jsou něco jiného. V ostatních typech softwaru najdeme totiž docela dobrou analogii s reálným světem. Někdy, jako například u pracovní plochy systému Windows, se tato analogie hledá o něco hůře, jinak jsou ovšem analogie dobré; dostanete se v ní odtud tam. Relační databáze jsou ale zcela umělé. Jistým způsobem se podobají geometrii: můžeme pomocí nich vytvářet modely reálného světa, samy o sobě je ale v tomto reálném světě nenajdeme. Schválně, kdy jste si třeba s přítelkyní jen tak nalili sklenku vína a šli se dívat na geometrické útvary na hladině jezera?

Pozor, teď tady hovořím o databázích, nikoli o tabulkách. Příkladů tabulek najdeme v reálném světě celou řadu, od telefonního seznamu třeba až po slovník. Ale relační databáze? Ani jednu. Ouvej. Neuvidíme je ani v odlesku měsíce na hladině jezera. Kartotéky v univerzitní knihovně, které obsahují jmenný, názvový a předmětový katalog, nemají od databáze příliš daleko; stále jsou to ale jen jisté oddělené množiny dat, jež spolu spojuje pouze laskavá osoba knihovníka.

Tato kniha pojednává o návrhu databázových systémů. Mým záměrem je podat vám jako čtenářům veškeré znalosti, které budete potřebovat při „přetavení“ chaotických a složitých situací z reálného světa do efektivního, funkčního návrhu databáze. Ani po přečtení knihy nebude moci pozorovat ty odlesky databází na hladině jezera, ale pokud jsem v ní odvedla dobrou práci, budete schopni navrhovat a implementovat třeba relační model ryb, racků a účinku planktonu na ně.

Celá kniha je rozdělena do tří částí. První z nich rozebírá základní principy relačního modelu. Zde se tedy skrývá ona nepříjemná, teoretická látka. Nebojte se, dále už to tak hrozné nebude. Druhá část se pouští do procesu analýzy a návrhu databází – řekneme si zde, co musíme udělat, abychom se od situace v reálném světě dostali do spolehlivého návrhu databázového systému. V poslední, třetí části knihy hovoříme o něčem, co je z pohledu uživatele u databázového systému nejdůležitější, a sice o uživatelském rozhraní.

I když v těch následujících několika stovkách stránek budeme pochopitelně hovořit o otázkách implementace, knížka není v žádném případě nějakou kuchařkou „jak programovat“. Najdete zde několik málo příkladů s programovým kódem, ty jsem se ovšem snažila omezit na minimum a pochopit byste je měli i přesto, že jste třeba nikdy s žádným programovacím jazykem nepracovali. Konkrétní příklady databází vycházejí z ukázkové databáze Northwind, která se dodává společně s Microsoft Accessem. (Verze databáze Northwind, dodávaná s SQL Serverem 7.0, je také velice podobná.) Po dokončení této knihy již budete znát většinu toho, co potřebujete do začátků tvorby databází, přičemž podrobnější výklad programátorského stylu najdete v různých knihách a zdrojích uvedených v seznamu literatury. A můžete si být jisti, že vaše datová architektura stojí na pevných základech a ani později v době řešení projektu vás nezavede do problémů.

1

# ČÁST

TEORIE RELAČNÍCH DATABÁZÍ

Z

C  
a  
P  
v  
S  
i  
:

# ZÁKLADNÍ POJMY

Co je to tedy za záhadnou stvůru, ta relační databáze? Stručně řečeno je to nástroj pro efektivní a spolehlivé ukládání informací a manipulace s nimi. Pod pojmem „efektivita“ a „spolehlivost“ zde přitom rozumíme ochranu dat před nahodilou ztrátou či poškozením; data nesmí dále spotřebovávat více prostředků (atž už počítačových, nebo i lidských), než kolik je nezbytně nutné, a musíme je být schopni načíst rozumným způsobem při přijatelné rychlosti. Samotná databáze je pak implementací relačního modelu, který představuje jistý způsob popisu určitých aspektů reálného světa a který vychází z množiny pravidel, zformulované poprvé Dr. E. F. Coddem na konci šedesátých let.

Teoreticky bychom mohli relační databázi naprogramovat „na zelené louce“, z ničeho; v reálu však budeme za normálních okolností využívat služeb vhodného systému pro správu databází (Database Management System, DBMS; česky se někdy uvádí pojem „systémy řízení báze dat“, ŠŘBD, často se ale říká jen „databázový systém“, my však v této knize pojmy „databázový systém“ a „systém pro správu databází“ rozlišujeme, pozn. překl.). Tomuto systému se dále často říká systém pro správu relačních databází (Relational Database Management System, RDBMS), z technického hlediska však systém pro správu databází, který má být považován za relační, musí vyhovět souboru zhruba 300 pravidel, jímž ale podle mých vlastních poznatků plně nevyhovuje ani jeden komerčně dostupný systém. V této knize budeme pracovat se dvěma systémy pro správu databází, a sice se systémem Microsoft Access a Microsoft SQL Server.

Řekla jsem, že relační databáze je fyzickou implementací relačního modelu (tedy datového modelu). Tyto dva pojmy je přitom velice důležité vzájemně rozlišovat. Ve fázi návrhu je téměř nemžné zcela ignorovat veškerá omezení konkrétního prostředí zvoleného pro implementaci, pravotní model by však měl být podle zásad správné praxe naopak co „nejčistší“. Asi teď namítnete, že při implementaci musí člověk poměrně často dělat různé kompromisy, například z důvodu rychlosti výsledného systému – během datového modelování však tato rozhodnutí mužeme, ba dokonce bychom je *měli* zcela prominout. Příkladem takového postupu je ukládání vypočtených polí (například SoučetObjednávek) do základních tabulek; v návrhu relačního systému je to jeden z *největších* prohřešků, zatímco v praxi se jedná o zcela běžnou techniku. V implementaci si tedy dělejte, co uznáte za vhodné, v modelu se však vypočtené pole objevit nesmí.

## Co je to databáze?

Databázová terminologie je skoro stejně ošemetná, jako pojem „objektově orientované programování“. Slovem „databáze“ můžeme totiž označit prakticky cokoliv – od jedné, jednoduché množiny dat, jako je například telefonní seznam, až po složitou množinu různých nástrojů, jako je třeba SQL Server – mezi těmito dvěma extrémy se pochopitelně nachází řada dalších možností. Tato určitá nepřesnost nemusí být nutně na závadu – takový je prostě náš přirozený jazyk – pro naše exaktní účely je však nevhodná, a proto se pokusím naopak zavést do našeho vyjadřování jistou preciznost. Vztahy mezi jednotlivými pojmy, o kterých budeme dále hovořit, znázorňuje obrázek 1-1.

Relační databáze samy o sobě sice v reálném světě nemají žádnou analogii, úkolem většiny z nich ale je modelovat jeho určitý aspekt. Tuto modelovanou část reálného světa nazývám *prostor problému*. Každý prostor problému je ze své podstaty chaotický a komplikovaný – pokud by složitý nebyl, asi bychom si k němu nevytvářeli model. Pro úspěch každého našeho projektu je ale životně důležité omezit navrhovaný databázový systém na konkrétní, dobře definovanou množinu objektů a jejich vzájemných vztahů; jen tak můžeme provádět rozumná rozhodnutí ohledně záběru výsledného systému.

Pod pojmem *datový model* budu rozumět myšlenkový (konceptuální) popis prostoru problému. Sem patří definice entit, jejich atributů (entitu může být například Zákazník a jeho atributy bude tvořit Jméno a Adresa) a omezení entity (JménoZákazníka nemůže být například prázdné). Datový model zahrnuje také popis vztahů mezi entitami a veškerá omezení platná pro tyto vztahy – jednaku nadřízenému tak například nesmí přímo podléhat více než pět podřízených pracovníků. Datový model však ještě nemá žádnou souvislost s fyzickým rozložením výsledného systému.

Definice fyzického rozvržení systému – tedy seznam implementovaných tabulek a pohledů – představuje takzvané *databázové schéma* neboli stručně jen *schéma*. Vzniká z myšlenkového modelu, a to převedením do fyzické reprezentace, kterou je již možné implementovat ve zvoleném systému pro správu databází. Všimněte si, že i schéma je stále myšlenkové (čili symbolické), nikoli fyzické. Schéma není nic víc, než jen opět datový model, tentokrát vyjádřený v pojmech, pomocí kterých jej popisujeme vůči databázovému stroji – tvoří ho tedy tabulky, spouště a další podobné entity. Jednou z výhod každého databázového stroje je, že se nikdy nemusíme zabývat fyzickou implementací; to znamená, že veškeré B-stromy a jejich listové úrovně můžete do značné míry ignorovat.

Jakmile databázovému stroji vysvětlíme, jak mají data podle naší představy vypadat (at už pomocí programového kódu, nebo v nějakém interaktivním grafickém prostředí, jaké má například Microsoft Access), vytvoří stroj určité fyzické objekty (obvykle je umístí na určité místo pevného disku, nemusí to ale být pravidlem), do nichž posléze začneme ukládat data. *Databází* zde budu nazývat sjednocení takto vytvořené struktury a vlastních dat. To znamená, že databázi tvoří fyzické tabulky, dále definované pohledy, dotazy a uložené procedury a konečně pravidla, jejichž „vynucováním“ bude databázový stroj zajišťovat ochranu dat.



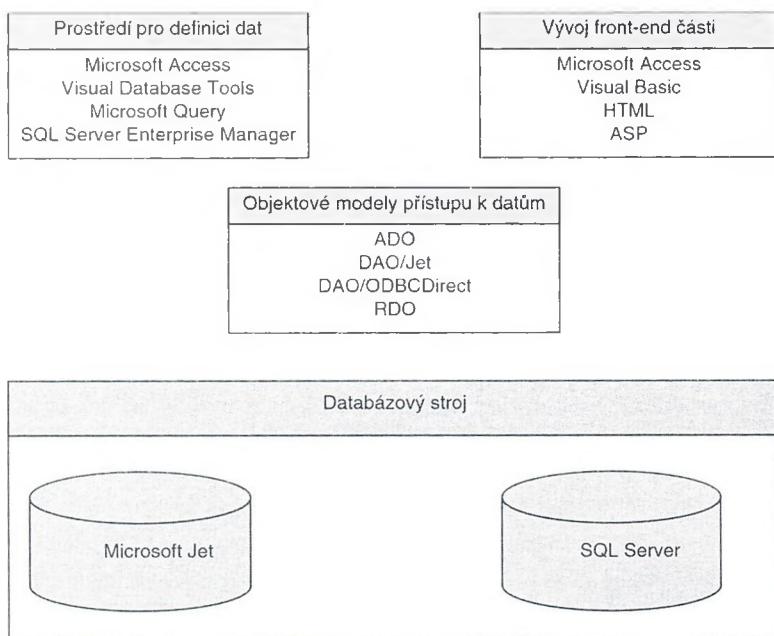
Obrázek 1-1 Terminologie relačních databází

Pojem „databáze“ přitom *nezahrnuje* samotnou *aplikaci*, která se skládá z formulářů a sestav, s niží posléze pracuje uživatel, ani neobsahuje různé softwarové součásti, jež propojují obě části, front-end a back-end, tedy například různý middleware nebo Microsoft Transaction Server. Slovo „databáze“ ve svém významu neobsahuje také databázový stroj. Soubor .mdb z Microsoft Accessu je tedy databází, zatímco Microsoft Jet je databázový stroj. Uvedený soubor typu .mdb může ve skutečnosti obsahovat nejen databázi, ale i další věci – například formuláře a sestavy – o tom si ale něco více povíme později.

Pro všechny tyto komponenty – tedy pro aplikaci, databázi, databázový stroj a vrstvu middleware dají – budu souhrnně používat pojem *databázový systém*. Součástí takto definovaného databázového systému je tedy veškerý software a veškerá data, která společně tvoří reálný, provozní systém.

## Databázové nástroje

Tato kniha se sice zaměřuje spíše na otázky návrhu než na samotnou implementaci, avšak abstraktní teorie sama o sobě k ničemu není, pokud nevíme, jak ji aplikovat. V knize budeme proto hovořit o výstavbě relačních databází pomocí nástrojů firmy Microsoft. Takových nástrojů existuje celá řada a Microsoft je podle všeho uvádí na trh jeden za druhým; řekneme si tedy, co jsou tyto nástroje zač, k čemu slouží a jak do sebe vzájemně zapadají. Veškeré nástroje, o kterých zde budeme hovořit, znázorňuje obrázek 1-2. Nejsnazší je přitom tyto nástroje vysvětlovat v souvislosti s převodem systému z podoby abstraktního modelu do živého, provozního systému; přesně to budeme totiž jako vývojáři dělat. A tímto způsobem jsou jednotlivé nástroje také seskupeny v následujícím obrázku.



Obrázek 1-2 Databázové nástroje, o kterých se v knize hovoří

## Databázové stroje

Na nejnižší úrovni se nacházejí databázové stroje. Někdy se jim říká „back-end“, tedy doslova „zadní konec“, „zadní část“ systému; to je ale poněkud nepřesné, protože pojem back-end znamená ve skutečnosti část konkrétní fyzické architektury, jak si ukážeme v kapitole 10. Tyto databázové stroje představují mechanismy, které provádějí vlastní fyzickou manipulaci s daty – uklá-

dají je tedy na disk a po vyžádání je opět načítají. My budeme hovořit o dvou databázových strojích, a sice o databázovém stroji Microsoft Jet a o SQL Serveru. Možná vás teď překvapuje, že se mezi těmito dvěma stroji nenachází také Microsoft Access. Tento systém – Microsoft Access – totiž pro manipulaci s daty uloženými v souborech typu .mdb využívá služeb databázového stroje Jet, přičemž se ale dokáže připojit k libovolným datům, uloženým v jakémkoli zdroji dat ODBC, a dále s nimi manipulovat; může tedy pracovat i s daty SQL Serveru. Access používal databázový stroj Jet vždycky, i když Microsoft jej nedával k dispozici jako samostatnou jednotku; tato situace se změnila až s příchodem Microsoft Visual Basicu verze 3. Microsoft Access 97 začal podporovat mechanismus ODBC Direct a Microsoft Access 2000 podporuje SQL Server, takže Microsoft fakticky začal od databázového stroje Jet odpojovat veškeré front-end nástroje. Osobně očekávám, že tento trend bude pokračovat i v dalších verzích. (Ale to je jenom můj názor a nechejte si ho raději pro sebe.)

Databázový stroj Jet a SQL Server jsou sice značně rozdílné, na druhé straně však mají také něco společného: oba jsou to vynikající nástroje pro ukládání dat a manipulaci s nimi. Podstatný rozdíl mezi nimi spočívá v jejich architektuře a v okruhu problémů, pro jejichž řešení jsou určeny. Microsoft Jet je typem „stolního“ databázového stroje a hodí se pro malé a střední systémy. SQL Server pracuje oproti tomu v architektuře klient/server a slouží zejména pro středně velké až rozsáhlé systémy; je rozšířitelný až na tisíce uživatelů, kteří současně pracují s kriticky důležitými aplikacemi. (Tuto charakteristiku ale prosím nechápejte tak, že by se databázový stroj Jet hodil pouze pro triviální systémy.) Na rozdíly mezi oběma těmito databázovými stroji budeme v celé této knize často narážet, přičemž výhody a nevýhody obou architektur rozebírá kapitola 10.

## Objektové modely přístupu k datům

Jak Microsoft Access, tak i Visual Basic nabízí jednoduchý mechanismus pro přímé svázání ovládacích prvků formuláře se zdrojem dat; to znamená, že my jako programátoři nemusíme komunikovat přímo s databázovým strojem. Z různých důvodů, o nichž si také něco řekneme, však takový postup není úplně vždycky možný nebo vhodný. V těchto případech je nejfektivnějším postupem manipulace s daty v programovém kódu pomocí vhodného datového objektového modelu.

Datový objektový model, neboli přesněji řečeno objektový model přístupu k datům, je jakési „lepidlo“, které k sobě pojí programové prostředí a databázový stroj. Každý objektový model definuje jistou množinu objektů, s jejichž vlastnostmi a metodami můžeme v programovém kódu manipulovat. Microsoft nabízí (v současné době) celkem tři objektové modely přístupu k datům: je to Data Access Objects (DAO), které se dodává ve dvou podobách, a sice DAO/Jet a DAO/ODBC Direct, dále Remote Data Objects (RDO), které se používá zejména pro přístup ke zdrojům dat typu Open Database Connectivity (ODBC), a konečně Microsoft ActiveX Data Objects (ADO), které mají v nejbližší budoucnosti nahradit jak DAO, tak i RDO.

Nejstarším z těchto tří objektových modelů je přitom DAO; to je nativní (přirozené) rozhraní k databázovému stroji Jet. Rozhraní RDO je podobné DAO, je ale optimalizováno pro přístup ke zdrojům dat ODBC, jako je například SQL Server a Oracle. Poslední z nich, ADO, má výrazně méně obsáhlou hierarchii objektů než ostatní dvě (skládá se z pouhých čtyř základních objektů) a nabízí jistá výrazná rozšíření modelu – podporuje například odpojené množiny záznamů a takzvané datové útvary.

Opět připomínám, že tato kniha se zaměřuje zejména na otázky návrhu, nikoli na samotnou implementaci; rozdíly mezi jednotlivými modely a jejich výhody či nevýhody zde proto nijak dopodrobna rozebírat nebudeme. Nejlepším zdrojem informací o této problematice je kniha od Williama Vaughna, *Hitchhiker's Guide to Visual Basic and SQL Server* (tato kniha vyšla v roce 1998 i v českém překladu u vydavatelství Computer Press, a to pod názvem *Visual Basic pro SQL Server*; pozn. překl.) a různé další dokumenty, které najdete na webovém sídle firmy Microsoft. (K objektovým modelům existují mimochodem i jiné alternativy, jako je například knihovna Visual Basic Library for SQL Server – VBSQL – a rozhraní OLE DB, o nich zde však hovořit nebudeme.)

### Vývojové nástroje pro front-end části

Databázové stroje Microsoft Jet a SQL Server pro nás tedy zabezpečují fyzické aspekty manipulace s daty, k tomu jim ale musíme nějakým způsobem umět říci, jakou mají mít data strukturu. V této oblasti nabízí Microsoft celou řadu mechanismů, my se ale budeme věnovat jen dvěma z nich: je to Microsoft Access a nástroje Microsoft Visual Database Tools. Ostatní metody nabízí zhruba stejné možnosti, já však osobně preferuji tyto dvě. Samozřejmě, že jakmile porozumíte nezbytným základům, mužete si již ke své práci vybrat libovolný vhodný nástroj.

Strukturu databáze je rovněž možné definovat pomocí programového kódu; i to si v této knize ukážeme, i když za normálních okolností něco takového nedoporučuji. Práce s interaktivními nástroji je totiž mnohem rychlejší, snazší a zábavnější, takže uvedené operace mají smysl snad jen v případě, že bychom potřebovali měnit strukturu dat za běhu aplikace (osobně nemám vůči těmto postupům naprostou žádnou důvěru, snad jen s výjimkou dočasných tabulek).

Jakmile je fyzická definice nové databáze dokončena, přejdeme k nástrojům, pomocí nichž vytvoříme formuláře a sestavy, se kterými bude uživatel v aplikaci komunikovat. Primárně budeme hovořit o dvou takových nástrojích: je to Microsoft Access a Visual Basic. V kapitole 10 se ještě zlehka dotkneme internetových prohlížečů, avšak již teď předesíláme, že samotný jazyk HTML je mimo rámec této knihy.

## Relační model

Relační model vychází z jistého souboru základních matematických principů odvozených z teorie množin a predikátové logiky. Na poli datového modelování se o jejich aplikaci poprvé pokusil Dr. E. F. Codd, a to koncem šedesátých let; Codd se posléze stal výzkumným pracovníkem firmy IBM a svoje výsledky publikoval v roce 1970. Relační model definuje způsob, jakým je možné data reprezentovat (tedy strukturu dat), způsoby jejich ochrany (neboli integritu dat) a dále operace, které můžeme nad daty provádět (manipulace s daty).

Relační model nepředstavuje pochopitelně jedinou metodu pro ukládání dat a manipulaci s nimi. Mezi jinými možnostmi jmenujme například hierarchické, síťové a hvězdicové modely. Každý z těchto modelů má své zastánce a každý z nich má pro jistou říšdu úloh nesporné přednosti. Relační model se tak například příliš dobře nehodí pro práci s hierarchickými daty, pro které byl speciálně navržen hvězdicový model. Díky své vysoké efektivitě a flexibilitě je ale relační model zdaleka nejoblíbenější technikou tvorby databází, a proto se mu také věnujeme v naší knize. Tento model přitom implementují oba probírané databázové stroje, tedy Microsoft Jet a Microsoft SQL Server.

Obecně se dá říci, že relační databázové systémy vykazují následující charakteristické vlastnosti:

- Veškerá data se pomyslně dají reprezentovat v pravidelně uspořádaných strukturách s řádky a sloupců, kterým se říká *relace*.
- Všechny hodnoty v databázi jsou *skalární*. To znamená, že v každé konkrétní pozici řádku a sloupce dané relace se nachází jedna (a to přesněji právě jedna) hodnota.
- Operace v databázi se provádějí vždy nad celou relací a jejich výsledkem je opět jiná celá relace; tomuto mechanismu se říká *uzávěr*.

Pokud jste někdy pracovali s databázemi systému Microsoft Access, zajisté v této „relaci“ poznáváte „množiny záznamů“ (recordset), případně slovy SQL Serveru „výslednou množinu“ (result set). Dr. Codd však při formulaci relačního modelu zvolil pojmenování „relace“, protože ten byl vůči jiným pojímům relativně „volný“ a neměl jiné, zavádějící významy, jako třeba slovo „tabulka“. Ve velice vztížitá je totiž jedna nesprávná představa, a sice že se relační model jmenuje podle jistých vztahů (neboli „relací“), které jsou definovány mezi tabulkami. Název relačního modelu je však ve skutečnosti odvozen z relací, na kterých je přímo postaven.

Všimněte si, že pro tento model stačí pouze *pomyslná* reprezentace dat v relaci; relační model tedy vůbec neurčuje, jak mají být data fyzicky implementována. Toto oddělení myšlenkové (konceptuální) a fyzické reprezentace dat se nám již dnes zdá být téměř samozrejmé, avšak před oněmi zhruba třiceti lety, kdy programování databází znamenalo obvykle přímé zapisování strojového kódu pro fyzickou manipulaci se zařízeními pro ukládání dat, bylo důležitou, ba dokonce průlomovou inovací.

Relace ve skutečnosti nepotřebují vůbec žádnou fyzickou reprezentaci. Určitá množina záznamů se tak může opravdu mapovat na reálnou, fyzickou tabulku, uloženou v určitém místě na disku, stejně tak ale může vyjadřovat pouze jisté sloupce, vybrané z deseti různých tabulek a doplněné o několik vypočtených polí – která se ovšem nikam fyzicky neukládají. Relací může být cokoliv, co je uspořádáno do struktury (formátu) řádků a sloupců a co obsahuje skalární hodnoty. Existence určité relace je tedy zcela nezávislá na její fyzické reprezentaci.

Díky principu uzávěru – tedy že za relace považujeme myšlenkově nejen základní tabulky, ale i výsledky definovaných operací – můžeme výsledky jedné operace použít jako vstupní hodnoty jiné operace. To znamená, že jak v databázovém stroji Jet, tak i v SQL Serveru mohou být výsledky jednoho dotazu základem jiného dotazu. To znamená, že vývojáři databází mají v ruce funkci, která se podobá podprogramům, neboli procedurám z procedurálního programování: je to možnost zapouzdření složitých nebo často prováděných operací, které pak můžeme v případě potřeby kdysi znova využít.

Takto můžeme například vytvořit dotaz se jménem DotazPlnéJméno, který spojí veškeré atributy, jež reprezentují různé části jména člověka (tedy křestní jméno, příjmení a titul) a vrátí jedno vypočtené pole se jménem PlnéJméno. Poté vytvoříme druhý dotaz, mezi jehož zdroje uvedeme dotaz DotazPlnéJméno a v němž budeme s vypočteným polem PlnéJméno pracovat úplně stejně, jako s kterýmkoli skutečným polem některé ze základních tabulek. To znamená, že výpočet plného jména nemusíme ve druhém dotazu zapisovat znovu.

Požadavek, podle něhož musí být všechny hodnoty v relaci pouze skalární, může být někdy tak trochu zrádný. Výraz „jediná hodnota“ je totiž zcela nutně subjektivní a je závislý na sémantice použitého datového modelu. Jako příklad si uvedeme údaj Jméno, který může v jednom datovém modelu definovat jedinou hodnotu, zatímco v jiném prostředí bude rozdělen do tří hodnot, tedy „Titul“, „Křestní jméno“ a „Příjmení“, a třeba ve třetím datovém modelu může obsahovat ještě „Druhé jméno“ a „Šlechtický titul“. O žádném z těchto datových modelů se přitom nedá s absolutní platností říci, že by byl správnější nebo méně správný; vhodnost konkrétního datového modelu závisí na datech, která pomocí něj máme reprezentovat.

## Terminologie relačních databází

Obrázek 1-3 ukazuje relaci, v níž jsou označena formální jména jejích základních komponent. Jestliže jste již něco o návrhu relačních databází slyšeli, pak si zajistě všimnete, že tato relace není v normální formě. To zde ovšem není nikterak na závadu; přestože nemá normální formu, vyhovuje podmínkám relace, protože je uspořádána do řádků a sloupců, a všechny její hodnoty jsou skalární.

Vektor souřadnic	Atribut		Záhlaví
			Tělo
Jméno Dodavatele:Jméno Společnosti	Výrobek:Jméno Výrobku	Jednotková Cena:Měna	
Specialty Biscuits, Ltd.	Teatime Chocolate Biscuits	230,00 Kč	
Specialty Biscuits, Ltd.	Sir Rodney's Marmalade	2 025,00 Kč	
Specialty Biscuits, Ltd.	Sir Rodney's Scones	250,00 Kč	
Specialty Biscuits, Ltd.	Scottish Longbreads	312,50 Kč	
PB Knäckebrot AB	Gustaf's Knäckebrot	525,00 Kč	
PB Knäckebrot AB	Tunnbröd	225,00 Kč	
Refrescos Americanas LTDA	Guaraná Fantástica	112,50 Kč	
Heli Süßwaren GmbH & Co. KG	NuNuCa Nuß-Nougat-Creme	350,00 Kč	
Heli Süßwaren GmbH & Co. KG	Gummibärchen	780,75 Kč	
Heli Süßwaren GmbH & Co. KG	Schoggi Schokolade	1 097,50 Kč	
Plutzer Lebensmittelgrößmärkte AG	Rössle Sauerkraut	1 140,00 Kč	
Plutzer Lebensmittelgrößmärkte AG	Thüringer Rostbratwurst	3 094,75 Kč	
Plutzer Lebensmittelgrößmärkte AG	Wimmers gute Semmelknödel	831,25 Kč	

Obrázek 1-3 Komponenty relace

Celá struktura se tedy, jak jsme si řekli, nazývá *relace*. Jednotlivý řádek dat je *vektor hodnot* nebo *souřadnic* (anglicky tuple). Každý řádek je přitom ve skutečnosti n-rozměrným vektorem, adjektivum „n-rozměrný“ se ale většinou vynechává. Počet řádků souřadnic neboli hodnot v relaci určuje její *kardinalitu*. V našem příkladu má daná relace kardinalitu 13. Každý sloupec ve vektoru souřadnic se pak nazývá *atribut*. Počet atributů v relaci se nazývá *stupeň*. Relace z našeho příkladu má tedy stupeň 3.

Relace se dále dělí do dvou hlavních částí, těmi jsou *záhlaví* a *tělo*. Jednotlivé vektory hodnot tvoří tělo relace, zatímco záhlaví tvoří – inu, záhlaví, tedy jakoby nadpisy sloupců. Všimněte si, že popisek každého atributu se skládá ze dvou částí, oddělených dvojtečkou, například *Jednotková Cena:Měna*. První část tohoto popisu vyjadřuje jméno (název) atributu, zatímco druhá popisuje

jeho doménu, neboli obor hodnot. U *oboru hodnot* se na chvíli zastavíme -- ten totiž vyjadřuje „druh“ dat, která atribut reprezentuje – v tomto případě se jedná o peněžní (měnové) údaje. Obor hodnot však *není* totéž, co datový typ. Této otázce se věnujeme v další části textu. Určení oboru hodnot se v záhlaví často neuvádí.

Tělo relace se skládá z neuspořádané množiny nula nebo více vektorů hodnot. Na tomto místě zdůrazním několik důležitých věcí. Za prvé, relace jako taková je neuspořádaná. To znamená, že čísla řádků *nemají* v relaci smysl. Relace nemá sama o sobě žádné vestavěné pořadí. Za druhé, podmírkám relace vyhovuje i prázdná relace, která neobsahuje ani jeden řádek hodnot. Za třetí, relace je v podstatě množina. A prvky množiny jsou již ze své podstaty jednoznačně identifikovatelné. Podtrženo a sečteno, tedy dostáváme, že pokud má být tabulka současně relací, musí mít každý její řádek jednoznačnou identifikaci a tabulka nesmí obsahovat opakování záznamy (dva nebo více stejných řádků).

Pokud jste již někdy pracovali s dokumentací k Microsoft Accessu nebo k SQL Serveru, asi vás teď napadne otázka, proč jste tato slova nikdy předtím neslyšeli? Jsou to totiž jisté formální pojmy, které se používají v technické (teoretické) literatuře; Microsoft je tedy jako pojmy nepoužívá. Rozhodla jsem se je v knize uvést, abyste o nich věděli a abyste se kvůli nim nedostali do rozpáku třeba na nějaké slavnostní recepci (tedy alespoň třeba na recepcích pořádaných na počest n-rozměrných vektorů hodnot třetího stupně). Nezapomeňte, že relace jsou čistě pomyslnou, konceptuální konstrukcí; jakmile je realizujeme neboli instancujeme v databázi, stanou se z nich v databázovém stroji Microsoft Jet *množiny záznamů*, respektive v SQL Serveru *výsledné množiny*; pro atribut přitom jak Microsoft Jet, tak i SQL Server používá shodně pojem pole a vektor souřadnic (hodnot) je – také v obou z nich – jednoduše *záznam*. Pojmy si odpovídají v podstatě vzájemně jednoznačně, zapamatujte si ale, že relace jsou jen pomyslná, teoretická struktura, zatímco množiny záznamů a výsledné množiny jsou již fyzické objekty.

## Datový model

Nejabstraktnější částí celého návrhu databáze je *datový model*; to je totiž myšlenkový (pojmový) popis daného prostoru problému. Datové modely se vyjadřují pomocí entit, atributů, domén (oborů hodnot) a vztahů. Zbývající část této úvodní kapitoly věnujeme proto všem uvedeným součástem datového modelu.

### Entity

Podat přesnou formální definici pojmu entita je dosud obtížné, jeho podstata je ale docela intuitivní a přímo se nabízí: *entita* je cokoliv, o čem v systému potřebujeme uchovávat nějaké informace.

Při zahájení prací na návrhu datového modelu není sestavení prvního seznamu entit nijak obtížné. Jestliže vy sami nebo vás zákazník hovoříte o příslušném prostoru problému, používáte jako podstatná jména a slovesa většinu vhodných kandidátů na entity. „Zákazníci kupují výrobky. Zaměstnanci prodávají výrobky. Dodavatelé nám prodávají svoje výrobky.“ Z těchto vět vybereme slova „zákazníci“, „výrobky“, „zaměstnanci“ a „dodavatelé“ – to jsou všechno zcela zřejmě entity.

Události, které v našem stručném „rozhovoru“ reprezentují slovesa „prodávat“ a „kupovat“, jsou také entity – i když toto tvrzení v sobě skrývá určitou léčku. Za prvé, sloveso „prodávat“ zde vyjadřuje ve skutečnosti dvě zcela rozdílné události, a sice prodej našeho výrobku zákazníkovi a ná-

kup cizího výrobku pro naši organizaci. V tomto příkladu je to na první pohled zcela zřejmé, přesto je však více než snadné učinit chybu, zejména pokud se v daném prostoru problému nepohybujeme příliš jistě.

Druhá záludnost je přesně opačná než první: někdy totiž dvě různá slovesa (v našem příkladu je to sloveso „kupovat“ z první věty a sloveso „prodávat“ ze druhé věty) popisují fakticky jednu a tu též událost, tedy nákup určitého našeho výrobku zákazníkem. Opět platí, že pokud nejsme s prostorem daného problému dostatečně dobře obeznámeni, nemusí to být úplně jasné. Tato chyba se přitom velice často odhaluje obtížněji než první. Jestliže totiž náš klient popisuje něco, co se na první pohled jeví jako jedna a ta stejná událost, pomocí různých sloves, může to opravdu znamenat zcela rozdílné typy událostí. Pokud je například náš klient krejčím a řekne ve dvou větách „zákazník si koupí oblek“ a „zákazník si objedná oblek“, znamená to sice v obou případech fakticky prodej obleku, avšak v prvním případě se jedná o hotovou konfekci, zatímco ve druhém si zákazník odnáší oblek šitý na míru. To jsou již velice odlišné procesy, které musíme odlišným způsobem také modelovat.

Kromě rozhovorů s klientem je při sestavování seznamu entit užitečné probrat veškeré dokumenty, které se v daném prostoru problému používají. Vhodnými zdroji kandidátů na entity jsou tak například různé vstupní formuláře, výkazy, sestavy a manuály pracovních postupů. S těmito dokumenty je však nutné zacházet velice opatrně. Tištěné dokumenty vykazují totiž vysoký stupeň setrvačnosti: zejména tisk vstupních formulářů je dosti nákladný, takže se často nemění ani při zásadnějších změnách pracovních pravidel a postupů. Jestliže tedy v průběhu své analýzy „zakopnete“ o entitu, na kterou v průběhu rozhovorů nikdy nepřišla řeč, nesmíte z toho ihned usuzovat, že se o ní klient jednoduše zapomněl zmínit. V takovém případě je totiž velice pravděpodobné, že se jedná o zastaralou položku, která se již v organizaci vůbec nepoužívá. A to si musíte sami ověřit.

Sestavený první seznam vhodných kandidátů na entity je nutno prověřit a zjistit, jestli je úplný a konzistentní. Opět musíme tedy vyhledat opakovány entity a dále takové dvojice (nebo n-tice) různých entit, které se v našem seznamu tváří jako jediná entita. Při této kontrole nám velice pomůže myšlenka takzvaných podtypů (subtypů) entit. Pokud se nyní vrátíme zpět k našemu příkladu s krejčím, pak můžeme říci, že obě položky „konfekce“ a „na míru“ reprezentují sice nákup určitého oblečení, avšak jako různé typy nákupu. Jinými slovy, Prodej a Ušití jsou podtypy jedné entity Nákup.

Atributy, které jsou pro oba typy Nákupu společné, přiřadíme v tomto případě nadtypu, tedy Nákupu, zatímco atributy platné jen pro konkrétní podtyp – tedy pro typ Konfekce nebo NaMíru – vyučleníme do tohoto podtypu. Tímto způsobem můžeme s oběma typy událostí zacházet podle toho, jak se nám to hodí – buďto jako s obecným Nákupem (například při výpočtu celkových tržeb), nebo jako s konkrétním speciálním typem Nákupu (například při porovnávání objemů obou kategorií).

Někdy přijdeme na to, že podtypy entity ve skutečnosti nemají žádné speciální atributy; v takovém případě je vhodnější nadefinovat TypProdeje (nebo TypZákazníka, nebo TypKdovíČeho) jako atribut příslušného společného nadtypu a zjištěné podtypy vůbec jako samostatné entity nemodelovat. V našem příkladu s krejčovským salónem tak můžeme například u zakázky šité na míru sledovat vybranou barvu a látku, zatímco u konfekčního prodeje nás zajímá výrobce oděvu. V takovém případě budeme obě entity modelovat skutečně pomocí podtypů. Pokud by nám ale stačilo

vědět jenom to, jestli byl prodaný oblek konfekcí nebo jestli byl šitý na zakázku, stačilo by zavést atribut TypProdeje, který by se také snáze implementoval.

Jednotlivé podtypy se obvykle vzájemně vylučují, rozhodně to ale nemusí platit vždycky. Uvažujme například databázi zaměstnanců. Všichni zaměstnanci mají určité atributy společné (například datum nástupu do zaměstnání, oddělení a telefonní linku); někteří z nich pak budou prodejci (ti mají definovány speciální atributy procento provize a stanovený plán prodeje), a jen někteří budou členy podnikového fotbalového mužstva. Pochopitelně to ale neznamená, že by prodejce nemohl hrát třeba hokej (a tím spíše, že by nemohl být členem podnikové fotbalové jedenáctky).

Většina entit modeluje objekty nebo události ve fyzickém světě: to jsou zákazníci, výrobky nebo telefonní objednávky. Těmto entitám se říká *konkrétní entity*. Některé entity však mohou modelovat také abstraktní pojmy (myšlenky). Těm se přirozeně říká *abstraktní entity* a zřejmě nejběžnějším příkladem je entita, která modeluje určitý vztah mezi jinými entitami – je to například vztah, na základě něhož je za jistého klienta odpovědný jeden konkrétní obchodní zástupce, nebo že je určitý student zapsán do určitého kursu.

Někdy nám stačí namodelovat pouhou existenci takového vztahu. Jindy potřebujeme k existujícímu vztahu evidovat různé doplňující informace, jako je například datum vytvoření vztahu nebo jeho určité charakteristické vlastnosti. Vztah mezi jaguáry a kojoty je například soupeřivý, zatímco vztah mezi jaguárem a králíkem je predátorský; pokud budeme někdy stavět zoologickou zahradu s volným výběhem, budou nás právě takové vztahy určitě zajímat.

Jestli se vztahy bez žádných atributů mají také modelovat jako samostatné entity, to je otázka určité diskuse. Osobně si myslím, že takovýto postup nic dobrého nepřináší, naopak komplikuje proces odvození databázového schématu z datového modelu. Z výkladu si ale odneste ponaučení, že vztahy jsou v datovém modelu stejně důležité jako samotné entity.

## Atributy

Navrhovaný systém bude o každé entitě zaznamenávat a sledovat určité skutečnosti. Těmto skutečnostem (údajům) se říká atributy dané entity. Pokud náš systém obsahuje například entitu Zákazník, budeme nejspíše potřebovat znát jeho jméno a adresu, a možná také jeho obor podnikání. Jestliže modelujeme entitu Volání Služby, budeme chtít vědět, jaký zákazník volal, kdy se tento hovor uskutečnil, kdo jej vyřídil a jestli se problém zákazníka vyřešil nebo ne.

Určování atributů daného modelu je sémantický proces. To znamená, že se budeme rozhodovat podle významu dat a podle způsobu jejich využití. Podívejme se opět na jeden běžný příklad – tentokrát to bude adresa. Stačí adresu zákazníka modelovat jako jedinou entitu (Adresa), nebo jako množinu několika entit (Ulice, ČísloDomu, Město, Okres, PSČ)? Většina návrhářů (včetně mne samotné) má již v podvědomí zabudován obecný princip, podle něhož se se strukturovanými daty snáze pracuje, takže každý téměř automaticky rozdělí adresu do množiny atributů; to ovšem nemusí být vždy nutně správné a zcela určitě ani nevhodnější.

Představme si například nějakou místní amatérskou hudební společnost. Adresy svých členů bude tato potřebovat jen proto, aby se jednou za čas mohly vytisknout adresní štítky. Protože všichni členové společnosti bydlí v jednom stejném městě, nemá smysl na adresu vůbec pohlížet jako na cokoli jiného než jednolity údaj: stačí ji tedy chápát jako jediný, víceádkový blok textu, který se na povel „vychrlí“ jako celek.

Co ale třeba zásilkový obchod, který veškerou svoji činnost zajišťuje prostřednictvím Internetu? Z důvodů důvodů potřebuje taková společnost vědět, v jakém státě každý jednotlivý zákazník bydlí. Údaj o státě by se zajisté dal zjistit i z onoho jednolitého textového pole, s jakým pracovala například naše hudební společnost, taková operace však již není tak snadná, a proto má smysl modelovat přinejmenším stát do samostatného atributu. Jak ale máme zacházet se zbytkem adresy? Má se skládat z několika atributů a pokud ano, z jakých? Uvědomte si, že i když adresy ve Spojených státech (a určitě i v České republice, pozn. překl.) odpovídají jistému poměrně standardnímu formátu, modelovat je není až tak jednoduché, jak by se na první pohled mohlo zdát.

Nyní si nejspíše řeknete, že množina atributů, kterou jsme si uvedli před chvílí – {Ulice, ČísloDomu, Město, Okres, PSČ} – musí přece stačit. Najednou ale přijde na to, že někdy systém narazí na adresu s číslem bytu, P. O. boxem, firemní nebo třeba armádní adresou? A co budete dělat s adresou „C/O“ na jméno někoho jiného? Svět je navíc čím dál menší, ale rozhodně ne méně komplikovaný, takže co se stane, až získáte prvního zákazníka mimo republiku? V takovém případě nejenže budeme muset doplnit údaj o státě a vhodně upravit poštovní směrovací číslo, ale možná také budeme muset změnit uspořádání atributů v adrese. Ve většině evropských států se například číslo domu zapisuje za jméno ulice. To není špatné a při pořizování dat se to může docela dobře sledovat; kolik uživatelů ale ví, že taková adresa 4/32 Griffen Avenue, Bondi Beach, Austrálie, znamená byt číslo 4, dům číslo 32 (a ne třeba naopak)?

Pointa zde ovšem není taková, že se adresy nějak obtížně modelují (i když to je samo o sobě pravdu), nýbrž že si dopředu nikdy nemůžeme stanovit nějakou jednoznačnou zásadu, jakým způsobem se mají data toho kterého typu modelovat. Pro mezinárodní zásilkovou službu tak budeme muset navrhnut nějaké dosti složité schéma, které je ale pro místní hudební společnost zcela nevhodné a zbytečné.

O známém francouzském malíři Henri Matisse se říkalo, že obraz prohlásil za dokončený až v okamžiku, kdy do něj nic dalšího nemělo smysl ani přidávat, ani ubírat. O návrhu entit se dá říci prakticky totéž. A jak zjistíte, že jste tohoto bodu dosáhl? S odpověď vás bohužel nepotěší: to se s jistotou nedá poznat nikdy. Ani za současného stavu technologií neexistuje způsob, jak sestavit prokazatelně správný návrh databáze. Dá se dokázat, že daný konkrétní návrh má určitou chybu, nepřítomnost chyb však dokázat neumíme. Nevina se prostě těžko dokazuje. Jak z toho ven? Přesná pravidla pochopitelně neexistují, můžeme si ale říci určité vhodné strategie.

První strategie zní takto: začněte požadovaným výsledkem a snažte se neudělat návrh složitější, než jaký musí nezbytně být.

Na jaké otázky má vaše databáze odpovídat? V našem prvním příkladu s hudební společností potřebujeme znát odpověď na jedinou otázku: „Kam mám této osobě adresovat poštu?“ Stačil nám tedy model s jediným atributem. Druhý příklad se zásilkovým obchodem již musí být schopen podat odpověď i na druhou otázku: „V jakém státě tato osoba bydlí?“. Zde tedy musíme nadefinovat jinou strukturu.

Nesmíte ale zapomenout, že výsledný model musí být natolik flexibilní, aby dokázal odpovídat nejen na otázky, které mu uživatelé budou klást hned teď, ale také na otázky, jaké se dají do budoucna jistým způsobem předvídat. Osobně bych se třeba docela klidně vsadila, že lidé z naší amatérské hudební společnosti za námi přijdou rok po implementaci systému zpátky a požádají

nás, jestli by se nedaly adresy trácit podle poštovního směrovacího čísla – aby mohli u pošty získat adresu za hromadné podání zásilek.

Budte také připraveni na otázky, na které by se uživatelé určitě rádi zeptali, jen kdyby mohli; to platí zejména v případě, že se snažíme o automatizaci nějakého ručního systému. Představte si reditele velké knihovny, který chce zjistit, kolik ze čtyř miliónů knih ve sbírkách bylo vydáno v Chicagu před rokem 1900. V klasickém systému by vás postavil před kartotéky a nechal vás pěkně dlouho se trápit. V dobré navrženém databázovém systému je však i takový dotaz zcela triviální.

Jednou ze známk opravdu dobrého návrháře je důkladnost a tvořivost, s jakou se od uživatelů pokouší zjistit takovéto potenciální otázky. Nezkušení analytici často skončí s tím, že uživatelé prostě nevědí, co vlastně chtějí. Samozřejmě že to nevědí; v tom jim máte pomoc vy – musí na to přijít společně s vámi.

V tomto místě se ale opět skrývá jedna léčka. Cenou za vysokou flexibilitu bývá totiž často složitost výsledného systému. Obecně totiž platí něco, co jsme si ukázali na příkladu s adresami: čím více způsoby chceme data rozkládat a skládat, tím více různých výjimek musíme ošetřit; a přesně v tomto místě se nám kdesi za horizontem ztrácí původní cíl dobrého výsledného systému.

To mě přivádí ke strategii číslo dvě: hledejte výjimky (výjimečné situace). Tato strategie má přitom dvě stránky: za prvé je důležité umět identifikovat všechny výjimky a za druhé, systém je třeba navrhnut tak, aby co nejvíce výjimek dokázal zvládnout bez zbytečného „obtěžování“ uživatele. Co tato věta znamená, si ukážeme na dalším příkladu: osobní jména.

Jestliže budeme pomocí našeho systému vytvářet korespondenci, je v něm velice důležité umět správně sestavit jméno. (Konkrétní příklad: pokud mi do domu dojde pošta, o které nevím a kterou jsem si ani nevyžádala, a bude na ní uvedeno *Pan R. M. Riordan*, ani mě nenapadne jí otevřít.) Většina jmen je docela jasných. Třeba takové jméno Slečna Jana Zlobilová se skládá z položek Titul, Jméno a Příjmení – nemám pravdu? Ne, nemám. (To vás asi napadlo, že.) Tak za prvé: křestní jméno a příjmení (respektive jejich podoba) je závislá na jakési národní kultuře. Přesnější by tedy bylo říkat Křestní jméno a Rodné jméno. Co teď ale uděláme s pánum, který se jmenuje Sir James Pedlington Smythe, Lord Dunstable? Je zde Rodné jméno celý výraz Pedlington Smythe, nebo je Pedlington jeho Druhé jméno? A co bude asi znamenat přídomek Lord Dunstable? A známý zpěvák Sting? To je Křestní jméno nebo Rodné jméno? A co se stane s někým, koho označujeme jako Umělec Kterému Se Dříve Říkal Prince? A zajímá nás to vůbec ještě?

Poslední otázka není až tak prostořeká, jak se na první pohled zdá. Dopis, na jehož adrese bude uvedeno jméno Sir James Pedlington Smythe, asi nikoho neurazí. Dotyčný gentleman se ale nejmenuje Sir Smythe – je to přece Sir James, nebo možná ještě Lord Dunstable. Teď ale uvažujte reálně: kolik z vašich klientů má šlechtický titul? Pokud byste naší místní společnosti amatérských hudebníků vytvořili takový systém pro evidenci členů, jehož obrazovku vidíme na obrázku 1-4, asi by vám moc nepoděkovali:

Zapamatujte si tedy, že flexibilita jde zpravidla na úkor vyšší složitosti. I když jsem před chvílí říkala, že je velice důležité snažit se odchytit co nejvíce výjimek, naopak je velice rozumné některé z nich vypustit jako nepravděpodobné, protože jejich zpracování by bylo příliš nákladné.

Správně od sebe rozlišit entity a atributy bývá někdy obtížné. Vhodným příkladem jsou opět adresy; konkrétní rozhodnutí musí být – jak jinak – zase závislé na daném prostoru problému. Ně-

Obrázek 1-4 Příklad složité obrazovky pro zadávání adresy

kteří návrháři tak zastávají postup vytvořit pro adresu jedinou entitu, do které snadno uložím všechny adresy, modelované v systému. Tento přístup má z pohledu implementace jisté nepřehlédnutelné výhody, jako je lepší zapouzdření a možnost opětovného využití programového kódu. Z pohledu návrhu systému mám však k němu vážné výhrady.

Je například dosti nepravděpodobné, že by se adresy zaměstnanců a zákazníků používaly přesně stejným způsobem. Hromadné sdělení zaměstnancům se tak například bude posílat spíše prostřednictvím interní elektronické pošty (podnikového intranetu), než klasickou „papírovou“ poštou. Je-li tomu opravdu tak, znamená to, že i pravidla a požadavky na oba typy adres musí být různé. Ona poměrně hloupá zadávací obrazovka z obrázku 1-4 se může pro adresy zákazníků opravdu dobré hodit, pokud je však v systému pro adresu definována jen jediná entita, jsme tutéž obrazovku nutni používat i pro adresy zaměstnanců, kde je zbytečně složitá.

## Domény

Ze začátku této kapitoly si možná vzpomínáte, že v záhlaví relace je pro každý atribut uvedena dvojice údajů JménoAtributu:JménoDomény. Dále jsem vám řekla, že definice domény neboli oboru hodnot popisuje *typ* dat, která daný atribut reprezentuje. Přesněji řečeno, obor hodnot tvoří množina všech přípustných platných hodnot, které smí určitý atribut obsahovat.

Obory hodnot se často zaměňují s datovými typy, toto vyjádření je ale nepřesné. Datový typ je totiž fyzický pojem, zatímco obor hodnot je logický. „Číslo“ je tedy datový typ, zatímco slovo „Věk“ představuje již obor hodnot. Uvedeme si ještě jiný příklad: atributy JménoUlice a Příjmení můžeme oba vyjádřit pomocí textového pole neboli řetězcové hodnoty, přestože se ale zcela zřejmě jedná o dva různé *typy* textového pole; oba atributy tak náleží do různých domén, mají jiný obor hodnot.

Pojem oboru hodnot má také širší význam než pojem datového typu. Definice oboru hodnot obsahuje totiž přesnější popis dat, která se považují za platná. Vezměme si například obor hodnot DosaženéVzdělání, které vyjadřuje univerzitní vzdělání určité osoby. V databázovém schématu můžeme tento atribut definovat jako Text[4]; to ale neznamená, že by mohl obsahovat libovolný čtyřznakový řetězec, protože se do něj smí zapisovat třeba jen hodnoty z množiny {MUDr, JUDr, RNDr, Mgr, Ing}.

Samořejmě ne všechny domény se dají definovat prostým výčtem svých členů. Domény neboli obor hodnot Věk obsahuje například řádově stovku různých hodnot, pokud se týká žijících osob, ale i desetitisíce hodnot, hovoříme-li o muzejní expozici (takový obor hodnot bychom asi „překřtili“ na Stáří). V takovém případě je vhodné definovat obor hodnot pouze jako jistou množinu pravidel, pomocí nichž můžeme u každé konkrétní hodnoty určit, jestli do množiny platných hodnot spadá nebo ne.

To znamená, že například obor hodnot VěkOsoby můžeme vyjádřit jako „nezáporné celé číslo z intervalu 0 až 120“, zatímco StáříExponátu bude mít obor hodnot „libovolné celé číslo větší než nula“.

„Aha“, říkáte si teď asi. Jako bych vás slyšela. „Obor hodnot je v podstatě spojení datového typu a validačního pravidla, které ověřuje platnost hodnoty.“ Je pravda, že s takovýmto názorem se příliš mýlit nebudete. Zapamatujte si ale, že validační pravidla jsou součástí datové integrity, nikoli

#### Objednávky

Cílo objednávky	Zákazník	Cílo prodeje	Datum objednávky
10248 Wolski Zajazd	5	04.07.1996	
10249 Toms Spezialitäten	6	05.07.1996	
10250 Hanari Carnes	4	08.07.1996	
10251 Victuailles en stock	3	08.07.1996	
10252 Suprèmes délices	4	09.07.1996	

#### Zaměstnanci

Cílo zaměstnance	Príjmení	Jméno	Funkce
1 Davolio	Nancy	Obchodní zástupce	
2 Fuller	Andrew	Viceprezident pro obchod	
3 Leverling	Janet	Obchodní zástupce	
4 Peacock	Margaret	Obchodní zástupce	
5 Buchanan	Steven	Obchodní ředitel	
6 Suyama	Michael	Obchodní zástupce	

Obrázek 1-5 Relace Zaměstnanci a Objednávky

součástí popisu dat. Validaci pravidlo poštovního směrovacího čísla může například pracovat s atributem Okres, zatímco obor hodnot atributu PSČ bude „řetězec o pěti číslicích“.

Všimněte si, že ve všech těchto definicích se nějakým způsobem odvoláváme na typ uložených dat (tedy číslo nebo řetězec). To na první pohled vypadá přímo jako datový typ, ve skutečnosti tomu tak ale není. Datové typy jsou totiž, jak jsem již řekla, fyzické; to znamená, že je vnitřně definuje a implementuje samotný databázový stroj. Bylo by tedy hrubou chybou definovat při vývoji

datového modelu nějaký obor hodnot jako varchar(30) nebo Long Integer, protože to jsou již specifické popisy, platné pro konkrétní databázový stroj.

Jestliže má pro libovolné dvě domény (obory hodnot) smysl vzájemně porovnávat atributy na nimi definované (odtud vyplývá i možnost provádění různých relačních operací, jako je například spojení tabulek, o němž budeme hovořit v kapitole 5), pak o nich říkáme, že jsou *typově kompatibilní*. Máme-li například definovány dvě relace z obrázku 1-5, je naprosto smysluplné propojit je podmínkou  $\text{ČísloZaměstnance} = \text{ČísloProdejce}$  – takto můžeme například získat seznam faktur, které vystavil daný zaměstnanec. Domény atributů  $\text{ČísloZaměstnance}$  a  $\text{ČísloProdejce}$  jsou tedy typově kompatibilní. Pokus o propojení relací s podmínkou  $\text{ČísloZaměstnance} = \text{ČísloObjednávky}$  zřejmě nepovede k žádné rozumné odpovědi – a to i kdyby byly tyto dvě domény definovány nad stejným datovým typem.

Ani databázový stroj Microsoft Jet, ani SQL Server však naneštěstí nemá vestavěnou silnou podporu domén; oba znají pouze datové typy. A ani v rámci jednotlivých datových typů neprovádí žádný z obou uvedených databázových strojů silné typové kontroly; to znamená, že oba klidně potichu převedou i nesprávné údaje. Jestliže bychom například atribut  $\text{ČísloZaměstnance}$  naefektivovali v tabulce *Zaměstnanci* z ukázkové databáze Northwind programu Microsoft Access jako dlouhé celé číslo (Long) a atribut *SoučetFaktury* z tabulky *Faktury* jako peněžní hodnotu (Currency), mohli bychom klidně vytvořit dotaz, který by obě tabulky propojil na základě kritéria  $\text{WHERE } \text{ČísloZaměstnance} = \text{SoučetFaktury}$ ; databázový stroj Microsoft Jet by nám v takovémto případě klidně bez jakékoli chyby vrátil seznam zaměstnanců, jejichž identifikátor  $\text{ČísloZaměstnance}$  se rovná celkové hodnotě nějaké faktury. Oba atributy nejsou sice typově kompatibilní, to však databázový stroj Jet neví.

Proč se tedy vůbec doménami zabývají? Protože jsou to velice užitečné nástroje pro návrh databází, jak si ukážeme v kapitole 2. „Jsou tyto dva atributy zaměnitelné?“ „Existují nějaká pravidla, která platí pro jeden atribut, nikoli však pro druhý?“ To všechno jsou při návrhu datového modelu velice důležité otázky, přičemž odpověď na ně pomáhá hledat právě analýza domén neboli oborů hodnot.

## Vztahy

Kromě atributů jednotlivých entit musíme v datovém modelu určit také vztahy, definované mezi různými entitami. Na myšlenkové (konceptuální) úrovni jsou *vztahy* jednoduše jistými asociacemi mezi entitami. Z tvrzení „Zákazníci nakupují výrobky“ tak vyplývá existence jistého vztahu mezi entitami *Zákazníci* a *Výrobky*. Entity, zapojené do určitého vztahu, se nazývají *účastníky*. Počet účastníků označujeme jako *stupeň* vztahu. (Stupeň vztahu je tedy částečně podobný, avšak nikoli stejný jako stupeň relace; ten totiž vyjadřuje počet atributů.)

Drtivá většina vztahů je binárních, tedy se dvěma účastníky – jako například vztah „Zákazníci nakupují výrobky“, nezbytně nutné to ale zdaleka není. Běžné jsou i ternární vztahy, tedy vztahy se třemi účastníky. Ze dvou binárních vztahů „Zaměstnanci prodávají výrobky“ a „Zákazníci nakupují výrobky“ implicitně vyplývá ternární vztah, vyjádřený větou „Zaměstnanci prodávají výrobky zákazníkům“. Na základě původních dvou binárních vztahů však nejsme schopni zjistit, který zaměstnanec prodal který výrobek kterému zákazníkovi; to dokáže ošetřit pouze ternární vztah.

Speciálním případem binárního vztahu je entita, která se účastní vztahu se sebou sama. Těmto vztahům se často říká vztahy typu „soupiska materiálu“ (výrobek obsahuje součásti, které se mohou skládat z dalších součástí) a nejčastěji se používají k reprezentaci hierarchických struktur. Obvyklým příkladem je vztah mezi zaměstnanci a jejich nadřízenými: určitý zaměstnanec může sám o sobě *být* nadřízeným a zároveň může *mít* nadřízeného.

Vztah mezi libovolnými dvěma entitami může být typu jedna k jedné, jedna k více, nebo více k více. Vztahy typu jedna k jedné jsou poměrně vzácné a nejčastěji představují vztah mezi entitou určitého nadtypu a podtypu. Vrátíme-li se k našemu předchozímu příkladu, můžeme si uvést vztah mezi entitou zaměstnance a entitou podrobných údajů o tomto zaměstnanci jako prodejci; tento vztah je zcela zřejmě typu jedna k jedné.

Vztahy typu jedna k více jsou zřejmě nejobvyklejší. Jedna faktura může obsahovat položky více výrobků. Jeden prodejce může vytvořit mnoho faktur. To jsou příklady vztahů typu jedna k více.

Poslední typ vztahu, více k více, není sice až tak obvyklý, přesto ani on není žádnou vzácností a najdeme pro něj řadu dobrých příkladů. Každý zákazník nakupuje mnoho výrobků a o každý výrobek se zajímá mnoho zákazníků. Podobně třeba každý učitel vyučuje mnoho studentů a každý student chodí do hodin k mnoha učitelům. Vztahy typu více k více se v relačním modelu nedají implementovat přímo; docela zřetelně se však nabízí jejich nepřímá implementace, jak uvidíme v kapitole 3.

Účast dané entity v určitém vztahu může být úplná nebo částečná. Jestliže entita nemůže existovat bez účasti v nějaké relaci, pak musí být její účast vždy *úplná*, jinak je *částečná*. Informace o prodejci tak například nemohou logicky vůbec existovat, pokud k němu není definován odpovídající Zaměstnanec. Opačná implikace však již neplatí. Zaměstnanec může být klidně něčím jiným, než jen prodejce, takže záznam o zaměstnanci může existovat i bez odpovídajícího záznamu informací o prodejci. Z toho vyplývá, že účast entity Zaměstnanec ve vztahu je částečná, zatímco účast entity Prodejce je úplná.

Podstatné je zde přitom zajistit, aby specifikace částečné nebo úplné účasti ve vztahu platila pro veškeré instance dané entity v libovolném okamžiku. Společnost může například docela běžně změnit dodavatele určitého výrobku. Pokud bychom ale účast entity Výrobky ve vztahu „Dodavatel prodávají své výrobky“ definovali jako úplnou, nebylo by možné odstranit určitého dodavatele, aniž bychom zároveň odstranili i podrobné informace o odpovídajících výrobcích.

## Diagramy entit a vztahů

Model entit a vztahů, který popisuje data jako entity, atributy a relace (vztahy), zavedl poprvé Peter Pin Shan Chen v roce 1976. Současně navrhl metodu jeho zobrazení do diagramů, které nazval diagramy entit a vztahů (Entity Relationship Diagrams, E/R diagram), a které se záhy rychle rozšířily a staly se obecně uznávanými. V E/R diagramech se entity popisují pomocí obdélníků, atributy se vyjadřují jako elipsy (ovály) a vztahy se znázorňují pomocí kosočtverců, jak vidíme na obrázku 1-6.

Povaha vztahů mezi entitami (tedy jedna k jedné, jedna k více nebo více k více) se vyjadřuje různými způsoby. Řada lidí používá pro znázornění variant jedna a více symboly 1 a M, respektive 1 a  $\infty$  (to je nekonečno). Já osobně používám techniku, které bychom mohli říkat „muř nožky“; vidíme ji opět na obrázku 1-6 a podle mého názoru je názornější.

Velkou výhodou E/R diagramů je, že se velice snadno nakreslí a přitom jsou dobře srozumitelné. Atributy však v praxi kreslím obvykle samostatně, protože ty se nacházejí na jiné úrovni podrobnosti.

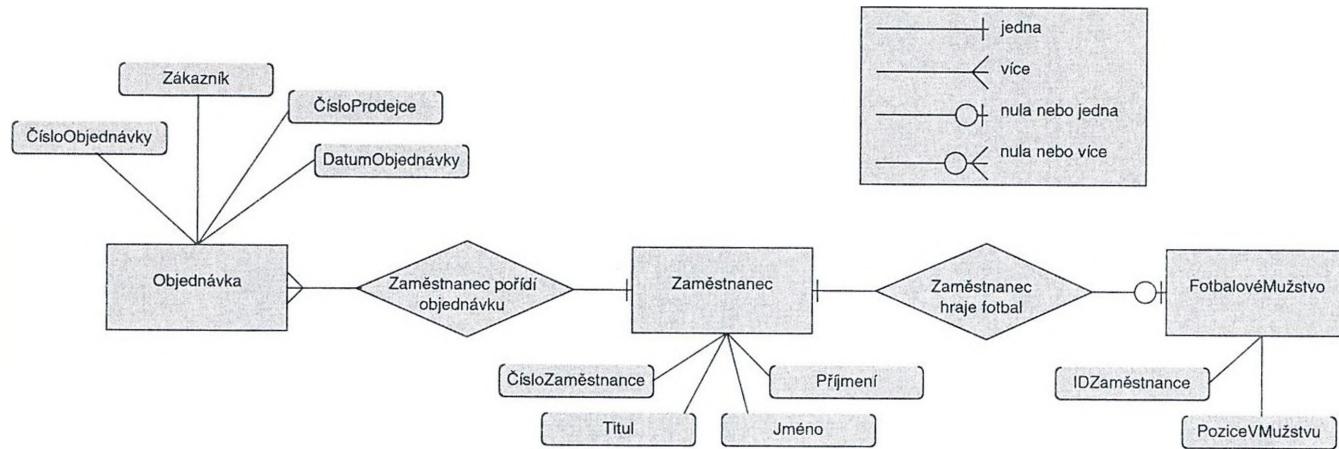
Člověk totiž obvykle uvažuje buďto o entitách v modelu a o vztazích mezi nimi, nebo o atributech dané entity, málokdy ale o obojím současně.

### **Stručné shrnutí**

V této kapitole jsme si řekli něco o komponentách databázového systému a položili jsme tak základy pro celý zbytek knihy. Začali jsme popisem takzvaného *prostoru problému*, který tvoří jistá dobře definovaná část reálného světa. *Myšlenkový (pojmový, konceptuální) datový model* je pak popis daného prostoru problému, provedený s pomocí *entit*, *atributů* a *domén* (oborů hodnot), definovaných v tomto prostoru problému. Fyzické rozmístění datového modelu se nazývá *databázové schéma* a realizuje se (instanciuje se) v takzvané *databázi*. Posledním pojmem je *databázový stroj*, který na žádost *aplikace* a jejím jménem provádí fyzické manipulace s databází. Aplikace se pak skládá z formulářů a sestav, s nimiž může uživatel pracovat.

Následující kapitola rozebírá podrobněji strukturu databáze a věnuje se také principům normalizace.

Obrázek 1-6  
ER diagram – diagram entity a vztahů



**S**

Ta  
mo  
bu  
je

Re  
ce  
rá  
da  
je

CE

C

# STRUKTURA DATABÁZÍ

2

Tato kapitola hovoří o prvním aspektu návrhu relačních databázových modelů: je to struktura samotných relací. Nejdůležitější úkol této fáze návrhu je docela jednoduchý: zabezpečit, že model bude schopen odpovědět jakoukoli otázku, kterou mu je možné rozumně položit. Druhým cílem je minimalizace veškerých redundancí a problémů s nimi spojených.

Redundance je špatná nejen proto, že plýtvá drahocennými prostředky, ale také proto, že nám docela výrazně komplikuje život. Uvažujme například množinu záznamů, která je uvedena na obrázku 2-1 a která reprezentuje faktury v určité společnosti. (Pro tento příklad budeme předpokládat, že je tato množina záznamů přímo základní tabulkou, která je uložena někde v databázi, nejedná se tedy o výsledek dotazu.)

Číslo objednávky	Jméno Zaměstnance	Zaměstnán od	Linka	Firma	Název výrobku	Množství
10254	Buchanan Steven	17.10.1993	3453	Tokyo Traders	Longlife Tofu	21
10248	Buchanan Steven	17.10.1993	3453	Leka Trading	Singaporean Hokkien Fried Mee	10
10248	Buchanan Steven	17.10.1993	3453	Formaggi Fortini s.r.l.	Mozzarella di Giovanni	5
10254	Buchanan Steven	17.10.1993	3453	Refrescos Americanas LTDA	Guaraná Fantástica	15
10248	Buchanan Steven	17.10.1993	3453	Cooperativa de Quesos 'Las Cabras'	Queso Cabrales	12
10254	Buchanan Steven	17.10.1993	3453	Ma Maison	Pâté chinois	21
10258	Davolio Nancy	01.05.1992	5467	Formaggi Fortini s.r.l.	Mascarpone Fabiofi	6
10258	Davolio Nancy	01.05.1992	5467	New Orleans Cajun Delights	Chef Anton's Gumbo Mix	65
10258	Davolio Nancy	01.05.1992	5467	Exotic Liquids	Chang	50
10255	Dodsworth Anne	15.11.1994	452	Exotic Liquids	Chang	20
10255	Dodsworth Anne	15.11.1994	452	Gai pâturage	Raclette Courdavault	30
10255	Dodsworth Anne	15.11.1994	452	Svensk Sjöföda AB	Inlagd Sill	25
10255	Dodsworth Anne	15.11.1994	452	Pavlova, Ltd.	Pavlova	35
10253	Leverling Janet	01.04.1992	3355	Karkki Oy	Maxilaku	40
10251	Leverling Janet	01.04.1992	3355	PB Knäckebrod AB	Gustaf's Knäckebrod	6
10251	Leverling Janet	01.04.1992	3355	Pasta Buttini s.r.l.	Ravioli Angelo	15
10251	Leverling Janet	01.04.1992	3355	New Orleans Cajun Delights	Louisiana Fiery Hot Pepper Sauce	20
10253	Leverling Janet	01.04.1992	3355	Formaggi Fortini s.r.l.	Gorgonzola Telino	20
10253	Leverling Janet	01.04.1992	3355	Aux joyeux ecclésiastiques	Chartreuse verte	42
10256	Leverling Janet	01.04.1992	3355	G'day, Mate	Perth Pasties	15
10256	Leverling Janet	01.04.1992	3355	Plutzer Lebensmittelgroßmarkte AG	Original Frankfurter grüne Soße	12
10252	Peacock Margaret	03.05.1993	5176	Specialty Biscuits, Ltd.	Sir Rodney's Marmalade	40
10261	Peacock Margaret	03.05.1993	5176	Specialty Biscuits, Ltd.	Sir Rodney's Scones	20

Obrázek 2-1 Tato množina záznamů obsahuje redundantní data

Vidíme, že hodnoty data nástupu ZaměstnánOd a jeho telefonní Linka jsou zde uvedeny pro každého zaměstnance několikrát. Toto prosté tvrzení má ovšem několik zásadních důsledků. Za prvé to znamená, že při každém pořizování nové faktury musíme znova zadávat údaje i do těchto dvou polí. A samozřejmě s každým dalším pořízením stejného údaje se zvyšuje pravděpodobnost chyby. Jak například z množiny záznamů na obrázku 2-2 poznáte, jestli pracovník Steven Buchanan nastoupil ve skutečnosti v roce 1989 nebo 1998? Za druhé, takto postavená struktura neumožňuje sa-

mostatné uložení data nástupu a telefonního čísla zaměstnance, dokud tento zaměstnanec neuskuteční svůj první prodej. A za třetí, jestliže faktury za určitý kalendářní rok přesuneme do archivu a z kmenové databáze je odstraníme, ztratí se s nimi i potřebné informace o datu nástupu a telefonním čísle.

Cíl objednávky	Jméno Zaměstnance	Zaměstnán od	Linka	Firma	Název výrobku	Množství
10254 Buchanan Steven		17.10.1989	3453	Tokyo Traders	Longlife Tofu	21
10248 Buchanan Steven		17.10.1998	3453	Leka Trading	Singaporean Hokkien Fried Mee	10
10248 Buchanan Steven		17.10.1998	3453	Formaggi Fortini s.r.l.	Mozzarella di Giovanni	5

Obrázek 2-2 Opakování údaje mohou vést k nekonzistencím

Jestliže jsou redundantní údaje uloženy ve více než jedné relaci, pak se všechny tyto problémy (kterým se obvykle říká *aktualizační anomálie*) ještě zhoršují. Podívejme se na příklad v obrázku 2-3. (Opět zde předpokládáme, že se jedná o základní tabulky, nikoli o výsledky dotazů.) Jestliže se zde změní telefonní číslo společnosti Around the Horn, nejenže je musíme změnit v množině záznamů Zákazníci, ale také nesmíme zapomenout aktualizovat každý výskyt tohoto čísla v množině záznamů Faktury.

Relace Zákazníci

Kód zákazníka	Firma	Telefon
ALFKI	Alfreds Futterkiste	030-0074321
ANATR	Ana Trujillo Emparedados y helados	(5) 555-4729
ANTON	Antonio Moreno Taquería	(5) 555-3932
AROUT	Around the Horn	(71) 555-7788
BERGS	Berglunds snabbköp	0921-12 34 65

Relace Faktury

Cíl objednávky	Firma	Telefon
10835 Alfreds Futterkiste	030-0074321	
10952 Alfreds Futterkiste	030-0074321	
11011 Alfreds Futterkiste	030-0074321	
10625 Ana Trujillo Emparedados y helados	(5) 555-4729	
10702 Ana Trujillo Emparedados y helados	(5) 555-4729	
10759 Ana Trujillo Emparedados y helados	(5) 555-4729	
10926 Ana Trujillo Emparedados y helados	(5) 555-4729	
10308 Antonio Moreno Taquería	(5) 555-3932	
10365 Antonio Moreno Taquería	(5) 555-3932	
10507 Antonio Moreno Taquería	(5) 555-3932	

Obrázek 2-3 Stejně opakování údaje se mohou vyskytovat i v několika množinách záznamů

Tyto operace pochopitelně nejsou tak složité a není jich totik, aby se nedaly správně provést. Problém je ale nezapomenout na všechno. A i když třeba *vy* sami nikdy nic nezapomínáte, jak teď zajistíte, že bude veškeré redundancy přesně znát i jiný programátor, který bude zajišťovat údržbu hotového systému někdy po půl roce, neřkuli že si bude pamatovat (či vůbec přesně vědět), jak je má správně ošetřit? Redundancí i výsledným problémům je proto mnohem, mnohem lepší hned od začátku předcházet.

Při těchto úvahách si ovšem musíme být jisti, že je daný redundantní atribut opravdu redundantní. Uvažujme příklad na obrázku 2-3. Na první pohled vás asi napadne, že atributy JednotkováCena v těchto dvou relacích jsou jednoduše redundantní. Oba ale ve skutečnosti reprezentují dvě různé hodnoty. První atribut JednotkováCena z relace Výrobky reprezentuje aktuální prodejní cenu.

## Relace Výrobky

Číslo výrobku	Název výrobku	Jednotková cena
11	Queso Cabrales	525,00 Kč
12	Queso Manchego La Pastora	950,00 Kč
14	Tofu	581,25 Kč
17	Alice Mutton	975,00 Kč
18	Cárnarvon Tigers	1 562,50 Kč
20	Sir Rodney's Marmalade	2 025,00 Kč
22	Gustaf's Knäckebrot	525,00 Kč
26	Gumbär Gummibärchen	780,75 Kč
27	Schoggi Schokolade	1 097,50 Kč
28	Rossle Sauerkraut	1 140,00 Kč
29	Thüringer Rostbratwurst	3 094,75 Kč
30	Nord-Ost Matjeshering	647,25 Kč
32	Mascarpone Fäboli	800,00 Kč

## Relace Objednávky

Číslo objednávky	Název výrobku	Jednotková cena	Množství	Datum objednávky
10248	Queso Cabrales	525,00 Kč	12	04.07.1996
10248	Singaporean Hokkien Fried Mee	350,00 Kč	10	04.07.1996
10248	Mozzarella di Giovanni	870,00 Kč	5	04.07.1996
10249	Manjimup Dried Apples	1 325,00 Kč	40	05.07.1996
10249	Tofu	581,25 Kč	9	05.07.1996
10250	Manjimup Dried Apples	1 325,00 Kč	35	08.07.1996
10250	Jack's New England Clam Chowder	241,25 Kč	10	08.07.1996
10250	Louisiana Fiery Hot Pepper Sauce	526,25 Kč	15	08.07.1996
10251	Louisiana Fiery Hot Pepper Sauce	526,25 Kč	20	08.07.1996
10251	Gustaf's Knäckebrot	525,00 Kč	6	08.07.1996
10251	Ravioli Angelo	487,50 Kč	15	08.07.1996

Obrázek 2-4 Zdánlivě totičné údaje ve skutečnosti nemusí být vůbec redundantní

Druhý atribut JednotkováCena v relaci Objednávky však vyjadřuje historický údaj o prodejně ceně v okamžiku realizace daného prodeje. Tofu má například uvedenu v relaci Objednávky jednotkovou cenu JednotkováCena 18,60 dolarů, zatímco v relaci Výrobky je již za 23,25 dolarů. Tofu se dnes tedy sice prodává za 23,25 dolarů, avšak na skutečnosti, že se někdy v minulosti prodalo za 18,60 dolarů, to již nic nemění. Oba atributy jsou definovány nad stejným oborem hodnot, jsou však logicky rozdílné.

Číslo zaměstnance	Formální oslovení	Jméno	Příjmení	Funkce
1	Slečna	Nancy	Davolio	Obchodní zástupce
2	Doktor	Andrew	Fuller	Viceprezident pro obchod
3	Slečna	Janet	Leverling	Obchodní zástupce
4	Pani	Margaret	Peacock	Obchodní zástupce
5	Pan	Steven	Buchanan	Obchodní ředitel

Číslo zaměstnance	Plné jméno
1	Slečna Nancy Davolio, Obchodní zástupce
2	Doktor Andrew Fuller, Viceprezident pro obchod
3	Slečna Janet Leverling, Obchodní zástupce
4	Pani Margaret Peacock, Obchodní zástupce
5	Pan Steven Buchanan, Obchodní ředitel

Obrázek 2-5 Provést zřetězení informací je snadné, vyčlenit je ale zpět ze složeného pole je však již obtížné

Schopnost datového modelu odpovědět pokládané otázky závisí z velké části na jeho úplnosti (zádný databázový systém zcela zřejmě nedokáže poskytovat data, která sám neobsahuje) a je částečně na jeho struktuře. Jak snadno se dá na otázky odpovídat, to ale již téměř výhradně závisí právě na struktuře datového modelu. Základní princip můžeme vyjádřit větou, že je velice snadné vzájemně spojit atributy a relace, zpátky je oddělit, to je však již velice obtížné. Tento princip dokresluje obrázek 2-5.

Nyní se vrátíme k našemu příkladu se jmény z kapitoly I a budeme uvažovat dvě relace uvedené na obrázku 2-5. Sloupec PlněJméno zde snadno odvodíme z horní relace, a to následujícím příkazem:

```
FormálníOslovení & " " & Jméno & " " & Příjmení & _  
    , " & Funkce
```

Pokud bychom ale z pole PlněJméno ve druhé tabulce chtěli zjistit jen Příjmení, museli bychom s řetězcem provádět takovéto manipulace:

```
Function NačtiPříjmení(PlněJméno) As String  
    Dim Příjmení As String  
        ' odříznout pole Funkce  
    Příjmení = Left(PlněJméno, InStr(PlněJméno, ",") - 1)  
        ' odříznout pole FormálníOslovení  
    Příjmení = Right(Příjmení, Len(Příjmení) - _  
        InStr(Příjmení, " "))  
        ' odříznout pole Jméno  
    Příjmení = Right(Příjmení, Len(Příjmení) - _  
        InStr(Příjmení, " "))  
    NačtiPříjmení = Příjmení  
End Function
```

Takovýto postup by byl také velice nepříjemně citlivý vůči jakýmkoli změnám obsahu pole PlněJméno; ze jména „Jiří Včelař Kotas“ se tak například vrátí „Včelař Kotas“, zatímco my bychom zájisté chtěli vrátit pouze „Kotas“. Vytvoření seznamu ve formátu Příjmení, Jméno by bylo velice obtížné.

Druhým důležitým principem při vytváření datového modelu je vyhnout se situacím, kdy odpověď na jednu otázku vyžaduje vyhodnocení stejných informací z několika různých polí; jen tak se později dají odpovědi na dotazy sestavit skutečně efektivně. Uvažujme například relace zobrazené na obrázcích 2-6 a 2-7, které modelují studenty a jimi zapsané předměty. Pokud máme v první relaci odpovědět na otázku: „Kteří studenti mají letos zapsanou biologii?“, musíme hledat hodnotu „Biologie“ hned v šesti polích. Celý příkaz SELECT jazyka SQL by vypadal takto:

```
SELECT ČísloStudenta FROM Zápisu WHERE Semestr1 = "Biologie"  
    OR Semestr2 = "Biologie" OR Semestr3 = "Biologie"  
    OR Semestr4 = "Biologie" OR Semestr5 = "Biologie"  
    OR Semestr6 = "Biologie";
```

U druhé struktury nám ke stejnemu dotazu stačí prohledat jen jedno pole, Předmět:

```
SELECT ČísloStudenta FROM Zápisu WHERE Předmět = "Biologie";
```

K cíli vedou oba uvedené postupy, druhý z nich je ale zcela samozřejmě jednodušší, snáze se o něm uvažuje a je méně náchylný k chybám při programování.

## Relace Zápisy

CíloStudenta	Jméno	Příjmení	Semestr1	Semestr2	Semestr3	Semestr4	Semestr5	Semestr6
1	Nancy	Davolio	Biologie	Francouzština	Angličtina	Historie	Chemie	Tělocvik
2	Andrew	Fuller	Tělocvik	Biologie	Francouzština	Angličtina	Chemie	Historie
3	Janet	Leverling	Tělocvik	Biologie	Francouzština	Angličtina	Chemie	Historie
4	Margaret	Peacock	Francouzština	Tělocvik	Historie	Angličtina	Biologie	Chemie
5	Steven	Buchanan	Biologie	Francouzština	Tělocvik	Historie	Angličtina	Chemie
6	Michael	Suyama	Francouzština	Tělocvik	Biologie	Historie	Angličtina	Chemie
7	Robert	King	Historie	Francouzština	Angličtina	Biologie	Chemie	Tělocvik

Obrázek 2-6 V této struktuře se určité dotazy dosť obtížně odpovídají

## Relace Zápisy

CíloStudenta	Jméno	Příjmení	Semestr	Předmět
1	Nancy	Davolio	1	Biologie
2	Andrew	Fuller	1	Tělocvik
3	Janet	Leverling	1	Tělocvik
4	Margaret	Peacock	1	Francouzština
5	Steven	Buchanan	1	Biologie
6	Michael	Suyama	1	Francouzština
7	Robert	King	1	Historie
1	Nancy	Davolio	2	Francouzština
2	Andrew	Fuller	2	Biologie
3	Janet	Leverling	2	Biologie
4	Margaret	Peacock	2	Tělocvik
5	Steven	Buchanan	2	Francouzština
6	Michael	Suyama	2	Tělocvik
7	Robert	King	2	Francouzština
1	Nancy	Davolio	3	Angličtina
2	Andrew	Fuller	3	Francouzština
3	Janet	Leverling	3	Francouzština

Obrázek 2-7 Tato struktura má sice více záznamů, vede však ke snazší formulaci dotazů

U datového modelování si tedy zapamatujte jen dvě věci: vyhýbejte se redundancím a navrhněte datový model vždy tak, aby z něj bylo možné data snadno načítat. Zbytek již představuje jakousi snahu o formalizaci těchto dvou základních principů. Pokud jste ale již k nějakému datovému modelování přičichli, zajisté víte, že jakkoli jsou tyto principy jednoduché, někdy se velice obtížně aplikují. Je to jako kancelářské svorky na papíry: jakmile je uvidíme, zdá se být odpověď dokonale snadná, jakmile ale poprvé dostaneme na stůl větší hromadu volných listů papíru, dostaneme se do problémů.

**Základní principy**

Principy normalizace, jež ve zbytku této kapitoly rozbehříme, jsou jisté nástroje, které řídí strukturu dat podobným způsobem, jakým „řídí“ kancelářská svorka listy papíru. Jednotlivé normální formy (budeme hovořit celkem o šesti) specifikují pro strukturu relací určitá pravidla, která jsou

od jedné úrovně ke druhé přísnější. Každá forma je rozšířením předchozí a vždy navíc zabraňuje určitým aktualizačním anomaliím.

Pamatujte si, že normální formy nejsou žádným univerzálním, povinným předpisem pro vytvoření „jediného správného“ datového modelu. Datový model může být klidně normalizovaný, a přesto se v něm nepodaří odpovědět na všechny otázky, které mu budeme klást, nebo sice odpoví, ale natolik pomalým a hluopým způsobem, že jakýkoli databázový systém, jenž nad ním postavíme, bude zcela nepoužitelný. Jestliže je ale datový model normalizovaný – to znamená, jestliže dodržuje pravidla relační struktury – je vysoce pravděpodobné, že bude jeho výsledkem efektivní, funkční databáze. Než se ale pustíme do vlastní normalizace, musíme si ještě vysvětlit několik základních pojmu.

### Bezztrátová dekompozice

V relačním modelu můžeme jednotlivé relace spojovat nejrůznějšími způsoby, a to prostřednictvím propojení atributů. Při vytváření plně normalizovaného datového modelu odstraňujeme redundance, přičemž rozdělujeme původní relace takovým způsobem, abychom nové relace mohli opět jakkoli zpětně spojit beze ztráty informace. To je princip *bezztrátové dekompozice*. Z relace uvedené na obrázku 2-8 můžeme například tímto postupem získat dvě relace, které ukazuje obrázek 2-9.

Cílo objednávky	Datum objednávky	Dodat dne	Firma	Adresa	Město	PSC
10248	04.07.1996	01.08.1996	Wolski Zajazd	ul. Filtrowa 68	Varsava	01-012
10249	05.07.1996	16.08.1996	Toms Spezialitäten	Luisenstr. 48	Münster	44087
10250	08.07.1996	05.08.1996	Hanari Carnes	Rua do Paço, 67	Rio de Janeiro	05454-876
10251	08.07.1996	05.08.1996	Victuailles en stock	2, rue du Commerce	Lyon	69004
10252	09.07.1996	06.08.1996	Suprêmes délices	Boulevard Tirou, 255	Charleroi	B-6000
10253	10.07.1996	24.07.1996	Hanari Carnes	Rua do Paço, 67	Rio de Janeiro	05454-876

Obrázek 2-8 Nenormalizovaná relace

#### Relace Zákazníci

Kód zákazníka	Firma	Adresa	Město	PSC
ALFKI	Alfreds Futterkiste	Obere Str. 57	Berlin	12209
ANATR	Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	Mexico D.F.	05021
ANTON	Antonio Moreno Taquería	Mataderos 2312	Mexico D.F.	05023
AROUT	Around the Horn	120 Hanover Sq.	Londýn	WA1 1DP
BERGS	Berglunds snabbköp	Berguvsvägen 8	Lulea	S-958 22
BLAUS	Blauer See Delikatessen	Forsterstr. 57	Mannheim	68306
BLONP	Blondel pere et fils	24, place Kléber	Štrasburg	67000
BOLID	Bólido Comidas preparadas	C/ Araquil, 67	Madrid	28023

#### Relace Faktury

Cílo objednávky	KódZákazníka	Datum objednávky	Dodat dne
10248 WOLZA		04.07.1996	01.08.1996
10249 TOMSP		05.07.1996	16.08.1996
10250 HANAR		08.07.1996	05.08.1996
10251 VICTE		08.07.1996	05.08.1996
10252 SUPRD		09.07.1996	06.08.1996

Obrázek 2-9 Relaci z obrázku 2-8 je možné rozdělit do těchto dvou relací bez ztráty jakékoli informace

S pomocí těchto dvou relací se již zbavíme redundantních adres, na druhé straně však můžeme stále vyhledat adresu zákazníka jednoduše prostřednictvím kódu KódZákazníka, který je uložen jak v množině záznamů Zákazníci, tak i v množině záznamů Faktury.

## Kandidátní klíče a primární klíče

V kapitole 1 jsem nadefinovala tělo relace jako neuspořádanou množinu nula nebo více vektorů souřadnic (hodnot) a poukázala jsem, že podle definice je každý člen množiny jedinečný. Má-li být tato podmínka splněna, musí u každé relace existovat určitá kombinace atributů, která jednoznačně identifikuje každý jednotlivý vektor souřadnic. Tato množina jednoho nebo více atributů se nazývá *kandidátní klíč*.

Daná relace může mít i více než jeden kandidátní klíč, avšak jednoznačnou identifikaci jednotlivých vektorů hodnot musí definovat vždy každý z nich; to znamená, že kandidátní klíč musí být schopen v jakémkoli daném okamžiku jednoznačně identifikovat všechny možné vektory hodnot,

Číslo kategorie	Název kategorie	Popis
1	Nápoje	Nealkoholické nápoje, káva, čaj, pivo
2	Koření	Sladké a pikantní omáčky, čalamády, pomazánky a koření
3	Cukrovinky	Zákusky, cukroví a bonbóny
4	Mléčné výrobky	Sýry
5	Obilné výrobky	Chleby, sušenky, těstoviny a cereální výrobky
6	Maso / Drůbež	Masné výrobky
7	Plodiny	Sušené ovoce a luštěnin
8	Mořské produkty	Ryby a dary moře

Obrázek 2-10 Kandidátní klíče musí být irreducibilní, takže jednoduchý klíč ČísloKategorie podmínkám vyhovuje, zatímco složený klíč {ČísloKategorie, NázevKategorie} již nikoli

nikoli pouze vektory hodnot z jisté vybrané množiny. Mimochodem, zároveň musí platit i opačné tvrzení: jestliže mají dva vektory souřadnic stejnou hodnotu kandidátního klíče, pak musí reprezentovat stejnou entitu. Z toho ovšem vyplývá, že kandidátní klíč nelze zjistit pohledem na konkrétní množinu dat. Jestliže je určité pole (nebo kombinace polí) pro danou množinu vektorů hodnot jedinečné, nemusí to nutně znamenat, že je vhodným kandidátním klíčem; nemáme totiž žádnou záruku, že bude jedinečné pro *všechny* vektory hodnot – a to je podmínka pro kandidátní klíč. Opět zde platí, že musíme správně pochopit sémantiku neboli význam sestavovaného datového modelu.

Vratíme se nyní zpět k relaci Faktury na obrázku 2-9. Identifikátor zákazníka KódZákazníka je v tomto příkladu jedinečný; je ovšem krajně nepravděpodobné, že by zůstal jedinečný i nadále – a zcela určitě to není záměrem! Bez ohledu na podobu konkrétního vzorku dat nám zde totiž sémantika modelu říká, že toto pole ve skutečnosti *není* kandidátním klíčem.

Každá relace musí mít již podle své definice alespoň jeden kandidátní klíč: triviálním kandidátním klíčem je množina všech atributů, ze kterých se skládá vektor souřadnic. Kandidátní klíče se přitom skládají buďto z jediného atributu (to je *jednoduchý klíč*), nebo z více atributů (potom hovoříme o *složeném klíči*). Kandidátní klíč musí ovšem splňovat ještě jednu doplňující podmítku: musí být irreducibilní neboli neredukovatelný (to znamená, že jakákoli podmnožina této množiny klíčů již nevyjadřuje jednoznačnou identifikaci). Z toho ale vyplývá, že množina všech atributů není *nutně* správným kandidátním klíčem. V relaci uvedené na obrázku 2-10 je například samotný atri-

but ČísloKategorie kandidátním klíčem, avšak množina {ČísloKategorie, NázevKategorie}, ačkoli je jedinečná, již kandidátním klíčem není, protože atribut NázevKategorie je zbytečný.

Někdy – i když ne příliš často – může mít jedna relace i několik možných kandidátních klíčů. V tomto případě bývá zvykem stanovit jeden z kandidátních klíčů za *primární klíč* a ostatní kandidátní klíče uvažovat jako *náhradní (alternativní) klíče*. Tento výběr primárního klíče z kandidátních klíčů zde můžeme provést libovolným způsobem a na logické úrovni nemá smysl se jím zabývat. (Nezapomeňte, že datový model je zásadně čistě abstraktní.) Proto osobně používám pojem „kandidátní klíč“ pouze na úrovni datového modelu, zatímco pojem „primární klíč“ ponechávám vyhrazen pro vlastní implementaci; tím zdůrazňuji rozdíl mezi datovým modelem a jeho fyzickou implementací.

### Poznámka

*Pokud je jediný možný kandidátní klíč příliš těžkopádný – vyžaduje například příliš mnoho pole nebo je příliš rozsáhlý – můžeme pro primární klíč použít speciální datový typ, kterým vytvoříme umělé klíče s jednoznačnými hodnotami, vygenerovanými systémem. Takový datový typ zná jak databázový stroj Microsoft Jet (zde se nazývá automatické číslo – pole AutoNumber), tak i Microsoft SQL Server (ten jej označuje datovým typem Identity) a pole jím definovaná jsou velice užitečná – tedy za předpokladu, že se nesnažíme dát jíme jakýkoli význam. Jsou to jen jakési značky. Ne máme u nich záruku, že se budou generovat přesně po řadě a vůbec nad jejich generováním máme jen minimální kontrolu; pokud se rozhodneme dát jíme jakýkoli jiný význam, zaděláme si na mnohem horší problémy, než jaké tímto postupem vyřešíme. Považujme je tedy jen za jisté jednoznačné umělé klíče, nic víc.*

Výběr kandidátních klíčů je sice sémantický proces, nemyslete si ale, že vhodným kandidátním klíčem musí být běžné atributy, pomocí nichž identifikujeme entitu v reálném světě. Člověka jako jednotlivce (osobu) charakterizujeme například v reálném životě zpravidla pomocí jména; prostě pohled do každého telefonního seznamu nás ale rychle přesvědčí, že jména budou jen sotva jednoznačnou identifikací.

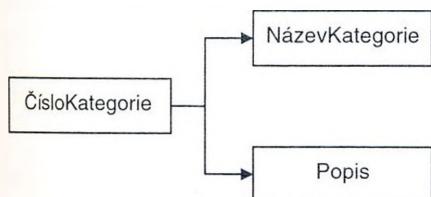
V kombinaci s nějakým jiným atributem (nebo s množinou atributů) musí samozřejmě i jméno na konec definovat kandidátní klíč; vhodná kombinace se ale zjišťuje velice obtížně. Jednou jsem pracovala v kanceláři se zhruba dvacetí dalšími lidmi, z nichž dva se jmenovali stejně – Larry Simon a třetí skoro stejně – Lary Simon. Říkalo se jim „malý Larry“, „Němec Larry“ a „světovlasý Larry“; to znamená, že jsme si vytvořili jakýsi kandidátní klíč z tělesné výšky, národnosti a barvy vlasů – a pochopitelně jména; tento kandidátní klíč by ale byl v jakékoli databází určitě nepoužitelný. V takovýchto situacích je proto nejlépe opustit definovaná pole a identifikovat jednotlivé vektory hodnot pomocí umělého, systémem generovaného identifikátoru, jako je třeba automatické číslo v Microsoft Accessu nebo pole Identity v SQL Serveru – ale ještě jednou pozor, nesnažte se tomuto číslu dávat jakýkoli jiný význam!

## Funkční závislost

Pojem *funkční závislosti* je při úvahách nad datovými strukturami také velice užitečným nástrojem. Jestliže je dán vektor hodnot T se dvěma množinami atributů {X<sub>1</sub>, ..., X<sub>n</sub>} a {Y<sub>1</sub>, ..., Y<sub>n</sub>} (tyto množiny nemusí být nutně disjunktní) a pro každou platnou hodnotu vektoru X existuje pouze jedna platná hodnota vektoru Y, pak říkáme, že množina Y je funkčně závislá na množině X.

V relaci, kterou jsme viděli na obrázku 2-10, má například každý vektor souřadnic se stejnou hodnotou množiny {ČísloKategorie} i stejnou hodnotu množiny {NázevKategorie, Popis}. To znamená, že můžeme říci, že atribut ČísloKategorie *funkčně určuje* množinu {NázevKategorie, Popis} (neboli tato množina je na atributu funkčně závislá). Všimněte si, že v opačném směru funkční závislost zdaleka platit nemusí: jestliže známe hodnotu {NázevKategorie, Popis}, neurčíme z ní jednoznačně hodnotu atributu {ČísloKategorie}.

Funkční závislosti mezi množinami atributů můžeme znázornit diagramem, jehož příklad vidíme na obrázku 2-11. V textu můžeme funkční závislosti vyjádřit zápisem  $X \rightarrow Y$ , který čteme slovy „X funkčně určuje Y“.



Obrázek 2-11 Diagramy funkční závislosti jsou v drtivé většině názorné a nepotřebují další komentář

Funkční závislost je velice zajímavá i po teoretické stránce, protože představuje mechanismus pro vytvoření něčeho, co začíná připomínat matematické modelování. Můžeme například rozebírat reflexitu a tranzitivitu funkční závislosti.

V praktické aplikaci pak funkční závislost znamená pohodlný způsob vyjádření něčeho, co je na první pohled zřejmé: pro každou relaci existuje určitá množina atributů, jejichž hodnoty jsou pro každý vektor souřadnic jednoznačné, přičemž ze znalosti jejich hodnot je možné odvodit hodnoty ostatních, nejednoznačných atributů.

Pokud je množina {X<sub>i</sub>} kandidátním klíčem, musí být na ní funkčně závislé veškeré ostatní atributy {Y<sub>j</sub>}; to vyplývá přímo z definice kandidátního klíče. Jestliže množina {X<sub>i</sub>} není kandidátním klíčem a funkční závislost není triviální (tedy jestliže {Y<sub>j</sub>} není podmnožinou {X<sub>i</sub>}), pak relace nutně obsahuje určitou míru redundance a musíme tedy provést další normalizaci. V jistém slova smyslu se přitom dá říci, že úkolem datové normalizace je zajistit, že na schématu podobném obrázku 2-11 budou všechny šipky *vycházet z kandidátních klíčů*.

## První normální forma

O relaci říkáme, že je v první normální formě, pokud jsou všechny její atributy definovány nad skalárními obory hodnot (domény). To je zároveň nejjednodušší i nejobtížnější myšlenka datového modelování. Samotný princip se skutečně přímo nabízí: každý atribut vektoru hodnot musí obsahovat jen jednu jednotlivou, skalární hodnotu. Co ale můžeme považovat za takovou jednotlivou hodnotu? V relaci na obrázku 2-12 vidíme, že atribut Položky obsahuje zcela zřejmě několik hodnot, a celá relace není tudíž v první normální formě. Problém ale zdaleka není vždy tak jasné.

Číslo objednávky	Kód Zákazníka	Datum objednávky	Položky	Objednáno celkem
1	CACTU	01.01.1999	3 Zaanse koeken, 1 Tarte au sucre	897,00 Kč
2	BSBEV	01.05.1998	4 Mozzarella di Giovanni	1 392,00 Kč
3	SUPRD	05.02.1999	3 Ravioli Angelo, 6 Tofu	1 980,60 Kč

Obrázek 2-12 Atribut Položky v této relaci není skalární

Na určité problémy spojené s určením, jestli je daný atribut skalární, jsme narazili již v kapitole 1 v souvislosti s modelováním jmen a adres. Dalším dosti ošidným oborem hodnot je datum. Údaje typu datum se totiž skládají ze tří přesně definovaných komponent: je to den, měsíc a rok. Měli bychom je tedy do tabulky uložit jako tři různé atributy, nebo z nich sestavit jedinou složenou hodnotu? Správná odpověď však ani zde, jako vždy, není jednoznačná – určíme ji pouze odpovídajícím posouzením sémantiky modelovaného prostoru problému.

Jestliže se v daném systému používá datum výhradně, nebo alespoň prvořadě jako jediná hodnota, pak jej můžeme považovat za skalární veličinu. Pokud ale systém bude potřebovat často manipulovat i s jednotlivými komponentami datového údaje, bude možná lepší je uložit do samostatných atributů. Někdy nás přitom například nezajímá den, ale pouze měsíc a rok. Jindy je pro nás významný měsíc a den, nikoli však již rok. To není tak obvyklé, někdy se to ale stát může.

Datová aritmetika představuje dosti obtížné operace, takže nejčastěji budeme zřejmě využívat atributy definované nad datovým typem DateTime, u něhož snadno předáme většinu té nevděčné práce našemu zvolenému vývojovému prostředí. Při porovnání jediné komponenty se však můžeme dostat snadno do problémů. To platí zejména v případě, že ignorujeme časovou složku daného pole. Jestliže například do pole DateCreated přiřadíme výsledek funkce jazyka VBA Now(), která vrací datum i čas, a poté se pokusíme tuto hodnotu porovnat s výsledkem funkce Date(), jež vrací pouze datum, můžeme dostat nesprávné výsledky. Čas třeba uživateli vůbec nemusíme zobrazit, a přesto se do dat ukládá; údaj vyjádřený jako „1/1/1999 12:30:19 AM“ pak zcela zřejmě není totéž co „1/1/1999“.

Dalším místem, kde lidé velice často narážejí na problémy s neskalárními hodnotami, jsou různé kódy a příznaky. V řadě společností se například různým obchodním případům či referencím přiřazují čísla, která jsou ve skutečnosti vypočtenou hodnotou. Takový kód může mít formát REF0010398 a znamená třeba, že se jedná o první případ otevřený v březnu 1998. Zde asi nemá smysl pokoušet se měnit podniková pravidla a ani není vhodné pokoušet se v datovém modelu posléze manipulovat s jednotlivými komponentami rozděleného referenčního čísla.

Mnohem snazší je totiž uložit všechny čtyři hodnoty samostatně: {ČísloRef, ČísloPřípadu, Měsíc, Rok}, tedy zvlášť číslo dané reference, číslo otevřeného případu, měsíc a rok. Jaké tím získáme výhody? Určit číslo dalšího případu nebo zjistit celkový počet případů otevřených v daném roce je záležitostí jednoduchého dotazu nad jediným atributem, při němž již nemusíme provádět žádné další manipulace. To má významné důsledky i pro rychlosť zpracování aplikace, a to zejména v prostředích architektury klient/server, protože vyjmout hodnotu zprostředka atributu se často nedá na databázovém serveru, takže v takovém případě musíme přenést každý jednotlivý záznam na klienta a zpracovávat jej lokálně.

Problematickým typem neskalárního atributu je také bitový příznak. V klasických prostředích konvenčního programování se množina booleovských hodnot docela běžně ukládá do jednotlivých bitů uceleného slova, nad nímž se posléze provádí různé testy a kontroly s pomocí bitových operací. O tento postup se do značné míry opírá například i programování v rozhraní API Windows. V konvenčním programování je to navíc zcela rozumné; v relačních datových modelech však nikoli. Nejenže takový postup porušuje pravidla pro první normální formu, ale dokonce něco takového ve skutečnosti ani nemůžeme udělat, protože verze jazyka SQL v databázovém stroji Jet, ani v SQL Serveru bitové operátory nedefinují. Mohli bychom si tedy v databázové aplikaci nadefinovat nějaké uživatelské funkce, avšak pouze v Microsoft Accessu (jazyk Microsoft Visual Basic nepodporuje uživatelské funkce v dotazech); takovýto dotaz by se navíc musel zpracovávat výhradně lokálně.

Toto je přitom naneštěstí takové omezení, s jakým se setkáme často jen z historických důvodů. Vyvíjíme-li ale novou aplikaci a máme-li tedy na výběr, je vhodné se tomuto postupu vynhnout a do jediného atributu zásadně nekódovat více než jednu informaci. I při spolupráci se staršími systémy můžeme data před započetím operací rozbalit a do množiny záznamů si uložit obě podoby.

Ještě na jeden typ neskalární hodnoty si musíme v relaci při kontrole podmínek první normální formy dávat pozor: je to opakování skupina hodnot. Na obrázku 2-13 vidíme opět jakousi relaci Faktury. Zde se kdosi kdysi rozhodl, že si zákazník nesmí koupit najednou více než pět položek zboží. Osobně pochybuji, že něco takového dotyčný výběc konzultoval s manažerem prodeje. Zcela vážně, takovéto omezení je téměř určitě umělé a zavádí je systém, nikoli povaha problému. A každé umělé systémové omezení je velké zlo, a v našem případě je také hned na první pohled nesprávné.

Číslo objednávky	KódZákazníka	Položka1	Množství1	Položka2	Množství2	Položka3	Množství3	Položka4	Množství4	Položka5	Množství5
1 ANTON		Queso Cabrales	4 Tofu	3 Ravioli Angelo	1 0	0 0					0
2 BLAUS		Louisiana Fiery Hot Pepper Sauce	2 0	0 0	0 0	0 0					0

Obrázek 2-13 Tento datový model omezuje počet položek zboží, které si zákazník smí současně nakoupit

Další příklad opakování skupiny vidíme na obrázku 2-14. Odtud není až tak na první pohled zřejmé, že se jedná o chybu, přičemž nad podobným modelem byla úspěšně implementována řada skutečných systémů. Při bližším pohledu ale zjistíme, že se opět jedná v podstatě o variaci struktury z obrázku 2-13, a že tedy tento postup vede ke stejným problémům. Představte si dotaz, pomocí něhož bychom chtěli zjistit, které výrobky překročily v libovolném měsíci prvního čtvrtletí plánovaný objem prodeje o více než 10 procent.

Název výrobku	Rok	PlánLeden	SkutLeden	PlánÚnor	SkutÚnor	PlánBřezen	SkutBřezen
Aniseed Syrup	1999	1 000,00 Kč	1 300,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč
Chai	1999	4 000,00 Kč	2 000,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč
Chang	1999	3 000,00 Kč	8 022,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč
Chef Anton's Cajun Seasoning	1999	7 000,00 Kč	7 300,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč
Chef Anton's Gumbo Mix	1999	3 000,00 Kč	3 231,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč	0,00 Kč

Obrázek 2-14 | Toto je opakování skupina

## Druhá normální forma

Relace je ve druhé normální formě, pokud je v první normální formě a navíc všechny její atributy jsou závislé na celém kandidátním klíči. Na obrázku 2-15 je tak například klíčem množina {NázevVýrobku, JménoDodavatele}, avšak pole TelefonníČísloDodavatele je závislé pouze na poli JménoDodavatele, nikoli na celém složeném klíči.

Název výrobku	JménoDodavatele	Název kategorie	TelefonníČísloDodavatele
Perth Pasties	G'day, Mate	Maso / Drůbež	(02) 555-5914
Tourtiere	Ma Maison	Maso / Drůbež	(514) 555-9022
Pâté chinois	Ma Maison	Maso / Drůbež	(514) 555-9022
Uncle Bob's Organic Dried Pears	Grandma Kelly's Homestead	Plodiny	(313) 555-5735
Tofu	Mayumi's	Plodiny	(06) 431-7877
Rössle Sauerkraut	Plutzer Lebensmittelgroßmärkte AG	Plodiny	(069) 992755
Manjimup Dried Apples	G'day, Mate	Plodiny	(02) 555-5914
Longlife Tofu	Tokyo Traders	Plodiny	(03) 3655-5011
Ikura	Tokyo Traders	Mořské produkty	(03) 3655-5011
Konbu	Mayumi's	Mořské produkty	(06) 431-7877
Carnarvon Tigers	Pavlova, Ltd.	Mořské produkty	(03) 444-2343
Nord-Ost Matjeshering	Nord-Ost-Fisch Handelsgesellschaft mbH	Mořské produkty	(04721) 8713
Inlagd Sill	Svensk Sjöföda AB	Mořské produkty	08-123 45 67

Obrázek 2-15 | Všechny atributy v relaci musí být závislé na celém klíči

### Relace Výrobky

Cílo výrobku	Název výrobku	Kategorie
1	Chai	Nápoje
2	Chang	Nápoje
3	Aniseed Syrup	Koření
4	Chef Anton's Cajun Seasoning	Koření
5	Chef Anton's Gumbo Mix	Koření
6	Grandma's Boysenberry Spread	Koření
7	Uncle Bob's Organic Dried Pears	Plodiny

### Relace Dodavatelé

Cílo dodavatele	JménoDodavatele	TelefonníČísloDodavatele
1	Exotic Liquids	(171) 555-2222
2	New Orleans Cajun Delights	(100) 555-4822
3	Grandma Kelly's Homestead	(313) 555-5735
4	Tokyo Traders	(03) 3555-5011
5	Cooperativa de Quesos 'Las Cabras'	(98) 598 76 54
6	Mayumi's	(06) 431-7877
7	Pavlova, Ltd.	(03) 444-2343
8	Specialty Biscuits, Ltd.	(161) 555-4448
9	PB Knackebrot AB	031-987 65 43

Obrázek 2-16 | Tyto dvě relace jsou již ve druhé normální formě

Jíž jsme si ukázali, že takový model vede k redundancím a dále že výsledkem redundancí mohou být velice nepříjemné problémy při údržbě systému. Lepší model vidíme tedy na obrázku 2-16.

Logicky vzato se jedná o požadavek nepokoušet se v jediné relaci vyjádřit dvě různé entity, Výrobky a Dodavatelé. Jestliže obě entity ve výsledném vyjádření od sebe oddělíme, nejenže se tím zbabíme nežádoucí redundancy, ale zároveň definujeme mechanismus pro uložení informací, které by nám jinak unikly. V příkladu na obrázku 2-16 můžeme například pořídit informace o Dodavateli ještě před tím, než od něj získáme informace k jakémukoli výrobku. V první relaci bychom si nic podobného dovolit nemohli, protože žádná komponenta primárního klíče nesmí být prázdná.

Další oblastí, v níž se lidé dostávají se druhou normální formou do problémů, je záměna omezení (omezujících podmínek), která jsou náhodou v daném případě vždy pravdivá, s podmínkami, které jsou (a mají být) pravdivé skutečně vždy bez ohledu na konkrétní data. V relaci na obrázku 2-17 tak například předpokládáme, že každý dodavatel má pouze jednu adresu, což může být dnes sice pravda, do budoucna to ale již platit nemusí.

Cíllo dodavatele	Adresa	Region	PSC
1 49 Gilbert St.		EC1 4SD	
2 P O. Box 78934	LA	70117	
3 707 Oxford Rd.	MI	48104	
4 9-8 Sekimai		100	
5 Calle del Rosal 4	Asturias	33007	
6 92 Setsuko		545	
7 74 Rose St.	Victoria	3058	
8 29 King's Way		M14 GSD	
9 Kaloadaqatan 13		S-345 67	
10 Av. das Americanas 12.890		5442	
11 Tiergartenstraße 5		10785	
12 Bogenallee 51		60439	
13 Frahmredder 112a		27478	
14 Viale Dante, 75		48100	
15 Hattevegen 5		1320	
16 3400 - 8th Avenue	OR	97101	
17 Brovallavägen 231		S-123 45	
18 203, Rue des Francs-Bourgeois		75004	
19 Order Processing Dept.	MA	02134	
20 471 Serangoon Loop, Suite #402		0512	
21 Lyngbysild		2800	
22 Verkoop		9999 ZZ	
23 Valtakatu 12		53120	
24 170 Prince Edward Parade	NSW	2042	
25 2960 Rue St. Laurent	Québec	H1J 1C3	

Obrázek 2-17 Dodavatel může mít i více než jednu adresu

## Třetí normální forma

O relaci říkáme, že je ve třetí normální formě, pokud je ve druhé normální formě a navíc všechny její neklíčové atributy jsou vzájemně nezávislé. Jako příklad si vezměme společnost, která má v každém státě Unie pouze jednoho obchodního zástupce. V relaci uvedené na obrázku 2-18 proto vidíme, že atributy Region a Prodejce jsou vzájemně závislé, žádný z těchto dvou atributů však není přípustným kandidátním klíčem dané relace.

CíloObjednávky	Firma	Region příjemce	Prodejce
10410 Bottom-Dollar Markets	BC	Leverling, Janet	
10389 Bottom-Dollar Markets	BC	Peacock, Margaret	
10411 Bottom-Dollar Markets	BC	Dodsworth, Anne	
10466 Comércio Mineiro	SP	Peacock, Margaret	
10290 Comércio Mineiro	SP	Callahan, Laura	
10414 Família Arquibaldo	SP	Fuller, Andrew	
10347 Família Arquibaldo	SP	Peacock, Margaret	
10366 Família Arquibaldo	SP	Dodsworth, Anne	
10423 Gourmet Lanchonetes	SP	Suyama, Michael	
10268 GROSELLA-Restaurante	DF	Callahan, Laura	
10253 Hanari Carnes	RJ	Leverling, Janet	
10250 Hanari Carnes	RJ	Peacock, Margaret	
10394 Hungry Coyote Import Store	OR	Davolio, Nancy	
10375 Hungry Coyote Import Store	OR	Leverling, Janet	
10495 Laughing Bacchus Wine Cellar	BC	Leverling, Janet	
10482 Lazy K Kountry Store	WA	Davolio, Nancy	
10307 Lonesome Pine Restaurant	OR	Fuller, Andrew	
10317 Lonesome Pine Restaurant	OR	Suyama, Michael	
10441 Old World Delicatessen	AK	Leverling, Janet	

Obrázek 2-18 Atributy Region a Prodejce jsou sice vzájemně závislé, žádný z nich však nemůže být kandidátním klíčem

Třetí normální formu bychom mohli vyžadovat opravdu i puntičkářsky ad absurdum. Ve většině případů můžeme například poštovní směrovací číslo neboli hodnotu atributu PSČ určit podle města a oblasti, tedy na základě hodnot City a Region; to znamená, že přísně vzato relace z obrázku 2-19 není ve třetí normální formě.

Firma	Adresa	Město	Region	PSČ
Alfreds Futterkiste	Obere Str. 57	Berlín		12209
Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	Mexico D.F.		05021
Antonio Moreno Taquería	Mataderos 2312	Mexico D.F.		05023
Around the Horn	120 Hanover Sq	Londýn		WA1 1DP
Berglunds snabbköp	Berguvsvägen 8	Lulea		S-958 22
Blauer See Delikatessen	Forsterstr. 57	Mannheim		68306
Blondel pere et fils	24, place Kléber	Štrasburg		67000
Bólido Comidas preparadas	C/ Araquil, 67	Madrid		28023
Bon app'	12, rue des Bouchers	Marseille		13008
Bottom-Dollar Markets	23 Tsawassen Blvd.	Tsawassen	BC	T2F 8M4
B's Beverages	Fauntleroy Circus	Londýn		EC2 5NT
Cactus Comidas para llevar	Cerrito 333	Buenos Aires		1010
Centro comercial Moctezuma	Sierras de Granada 9993	Mexico D.F.		05022

Obrázek 2-19 Striktě vzato, tato relace není ve třetí normální formě

Dvě relace, které vidíme na obrázku 2-20 jsou tedy z technického hlediska třetí normální formy správnější; fakticky nám ale takový model přináší jen jedinou výhodu, kterou je možnost automatického vyhledání poštovního směrovacího čísla při zadávání nových záznamů; uživateli tak ušetříme několik úhozů kláves. To není zcela bezvýznamné, na druhé straně ale stejnou funkci můžeme implementovat i vhodnějšími způsoby; zde totiž při každém odkazu na adresu obchodního zástupce musíme provést operaci spojení relací, která znamená jistou nezanedbatelnou úroveň režie.

Firma	Adresa	Město	Region
Bottom-Dollar Markets	23 Tsawassen Blvd.	Tsawassen	BC
Comércio Mineiro	Av. dos Lusiadas, 23	São Paulo	SP
Família Arquivaldo	Rua Oros, 92	São Paulo	SP
Gourmet Lanchonetes	Av. Brasil, 442	Campinas	SP
Geal Lakes Food Market	2732 Baker Blvd.	Eugene	OR
GROSELLA-Restaurante	5\$ Ave. Los Palos Grandes	Caracas	DF
Hanari Carnes	Rua do Paço, 67	Rio de Janeiro	RJ
HILARION-Abastos	Carrera 22 con Ave. Carlos Soublette #8-35	San Cristóbal	Táchira
Hungry Coyote Import Store	City Center Plaza	Elgin	OR
Hungry Owl All-Night Grocers	8 Johnstown Road	Cork	Co. Cork
Island Trading	Garden House	Cowes	Isle of Wight
Laughing Bacchus Wine Cellars	1900 Oak St.	Vancouver	BC
Lazy K Kountry Store	12 Orchestra Terrace	Walla Walla	WA

Město	Region	PSC
Anchorage	AK	99508
Vancouver	BC	V3F 2K1
Tsawassen	BC	T2F 8M4
San Francisco	CA	94117
Caracas	DF	1081
Boise	ID	83720
Cowes	Isle of Wight	PO31 7PJ
Barranquimeto	Lara	3508
Butte	MT	59801
Albuquerque	NM	87110
I de Margarita	Nueva Esparta	4980
Portland	OR	97219
Elgin	OR	97827
Eugene	OR	97403
Portland	OR	97201
Montreal	Québec	H1J 1C3
Rio de Janeiro	RJ	02389-673
Rio de Janeiro	RJ	02389-890
Rio de Janeiro	RJ	05454-876
Resende	SP	08737-363
São Paulo	SP	05432-043
Campinas	SP	04876-786
São Paulo	SP	05442-030
São Paulo	SP	05487-020

Obrázek 2-20 Tyto dvě relace již jsou ve třetí normální formě

To, jestli a jakým způsobem třetí normální formu implementovat, přitom opět – stejně jako každé jiné rozhodování v procesu datového modelování – závisí na sémantice daného modelu. Samostatnou entitu vytvoříme jen tehdy, pokud je nová entita pro datový model významná, nebo pokud se její data budou často měnit, případně pokud bezpečně víme, že tím při implementaci získáme nějaké technické výhody. Poštovní směrovací čísla se mohou měnit, avšak rozhodně ne často; navíc ve většině systémů nejsou až tak životně důležitá.

## Další normalizace

První tři normální formy byly součástí původní Coddovy formulace relační teorie a v drtivé většině případů se ani ničím jiným nemusíte zabývat. Zapamatujte si tuhle „kouzelnou formulku“, kterou jsem se naučila na vysoké škole: „Klíč, nic než klíč a celý klíč, pomáhej mi Codd.“ (Nepřipo-

mírná vám to nic? Jak přisahají svědkové u soudu? „Pravdu, nic než pravdu a celou pravdu, pomáhej mi Bůh...“ Pozn. překl.)

Další normální formy – tedy Boyce/Coddova, čtvrtá a pátá – slouží pro speciální případy, z nichž většina je ale dosti vzácných.

### Boyce/Coddova normální forma

Boyce/Coddova normální forma se pokládá za jistou variaci třetí normální formy. Používá se pro speciální případy relací s více kandidátními klíči. Boyce/Coddovu normální formu lze přitom aplikovat pouze při splnění následujících podmínek:

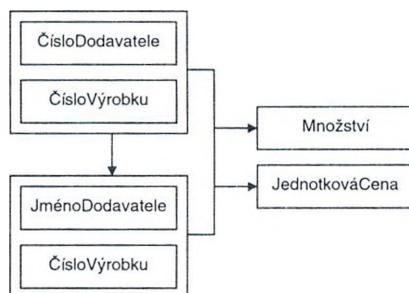
- Relace musí mít dva nebo více kandidátních klíčů.
- Nejméně dva z kandidátních klíčů musí být složené.
- Kandidátní klíče se v některých atributech musí překrývat.

Nejsnáze Boyce/Coddovu normální formu pochopíme s pomocí funkčních závislostí. Boyce/Coddova normální forma v podstatě říká, že *mezi* kandidátními klíči nesmí být žádná funkční závislost. Uvažujme například relaci z obrázku 2-21. Tato relace je zřejmě ve třetí normální formě (vycházíme z předpokladu, že jména dodavatelů JménoDodavatele jsou jedinečná), přesto stále ve významu může obsahovat redundanci.

Cílo dodavatele	Jméno dodavatele	Číslo výrobku	Množství	Jednotková cena
5 Cooperativa de Quesos 'Las Cabras'		11	12	140,00 Kč
6 Mayumi's		72	5	85,00 Kč
14 Formaggi Fortini s.r.l		14	16	34,80 Kč
19 New England Seafood Cannery		815	22	566,00 Kč
20 Leka Trading		26	34	99,00 Kč
21 Lyngbysild		33	2	81,00 Kč
24 G'day, Mate		11	83	9,50 Kč
29 Forêts d'érabiles		8	10	12,00 Kč

Obrázek 2-21 Tato relace je ve třetí normální formě, nikoli však v Boyce/Coddově normální formě

Tato relace má dva kandidátní klíče, {ČísloDodavatele, ČísloVýrobku} a {JménoDodavatele, ČísloVýrobku}, mezi nimiž je funkční závislost, jak vidíme z obrázku 2-22.



Obrázek 2-22 Diagram funkční závislosti pro relaci z obrázku 2-21

Jak vidíte, mezi položkami {ČísloDodavatele} → {JménoDodavatele} je funkční závislost, a to je pořušení pravidel Boyce/Coddovy normální formy. Správný datový model vidíme na obrázku 2-23.

#### Relace Dodavatelé

Číslo dodavatele	Jméno dodavatele
24	G'day, Mate
29	Forêts d'éables
14	Formaggi Fortini s.r.l.
21	Lyngbysild
6	Mayumi's
20	Leka Trading
5	Cooperativa de Quesos 'Las Cabras'
19	New England Seafood Cannery

#### Relace Výrobky

Číslo dodavatele	Císlo výrobku	Množství	Jednotková cena
24	11	83	9,50 Kč
29	8	10	12,00 Kč
14	14	16	34,80 Kč
21	33	2	81,00 Kč
6	72	5	86,00 Kč
20	26	34	99,00 Kč
5	11	12	140,00 Kč
19	815	22	566,00 Kč

Obrázek 2-23 Tento model představuje plně normalizovanou verzi obrázku 2-21

Porušení pravidel Boyce/Coddovy normální formy můžeme poměrně snadno předcházet – stačí si správně uvědomit, o čem daná relace logicky vypovídá. Jestliže je na obrázku 2-21 uvedena relace Výrobky, pak v ní nemají co dělat informace o dodavatelích (a samozřejmě i naopak).

### Čtvrtá normální forma

Čtvrtá normální forma představuje teoretický základ jednoho docela zřejmého principu: v jedné relaci se nesmí spojovat nezávislé opakování skupiny. Všechno si opět ukážeme na příkladu. Společnost Northwind Traders prodává pod svou vlastní značkou zboží, které vyrábí různí dodavatelé v různých velikostech balení. Všichni dodavatelé jsou přitom schopní dodávat balení všech velikostí. Zcela nenormalizovaná verze relace Výrobky může vypadat zhruba jako na obrázku 2-24.

Název výrobku	Jméno dodavatele	Množství v jednotce
Chai	Exotic Liquids	8 oz, 16 oz, 32 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delight	8 oz, 16 oz, 32 oz
Pavlova	Pavlova, Ltd.	8 oz, 16 oz, 32 oz

Obrázek 2-24 Nenormalizovaná relace

Prvním krokem v normalizaci této relace je odstranění neskalárního atributu VelikostBalení; výsledkem bude relace, kterou vidíme na obrázku 2-25.

Relace na obrázku 2-25 je překvapivě v Boyce/Coddově normální formě, protože všechno závisí na „celém klíči“. Zcela zřejmě zde ale máme problémy s redundancí, takže udržovat datovou integritu může být docela nepříjemný problém. Řešení těchto problémů spočívá v myšlence *dvojic vícehodnotové závislosti* a ve čtvrté normální formě.

Název výrobku	Jméno dodavatele	Množství v jednotce
Chai	Exotic Liquids	8 oz
Chai	Exotic Liquids	16 oz
Chai	Exotic Liquids	32 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	8 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	16 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	32 oz
Pavlova	Pavlova, Ltd.	8 oz
Pavlova	Pavlova, Ltd.	16 oz
Pavlova	Pavlova, Ltd.	32 oz

Obrázek 2-25 Upravená verze relace z obrázku 2-24 je v Boyce/Coddově normální formě

Dvojici vícehodnotové závislosti představují dvě vzájemně nezávislé množiny atributů. Na obrázku 2-24 je takovouto vícehodnotovou závislostí {NázevVýrobku} → {VelikostBalení | JménoDodavatele}, kterou čteme následovně: „Pole NázevVýrobku současně určuje pole VelikostBalení a JménoDodavatele“. Čtvrtá normální forma v podstatě říká, že vícehodnotové závislosti musíme vyčlenit do samostatné relace, jako je tomu na obrázku 2-26. Formální definice pak zní, že relace je ve čtvrté normální formě, pokud je v Boyce/Coddově normální formě, a navíc všechny vícehodnotové závislosti jsou zároveň funkčními závislostmi z kandidátních klíčů.

Název výrobku	Množství v jednotce
Chai	8 oz
Chai	16 oz
Chai	32 oz
Chef Anton's Cajun Seasoning	8 oz
Chef Anton's Cajun Seasoning	16 oz
Chef Anton's Cajun Seasoning	32 oz
Pavlova	8 oz
Pavlova	16 oz
Pavlova	32 oz

Název výrobku	Množství v jednotce
Chai	8 oz
Chai	16 oz
Chai	32 oz
Chef Anton's Cajun Seasoning	8 oz
Chef Anton's Cajun Seasoning	16 oz
Chef Anton's Cajun Seasoning	32 oz
Pavlova	8 oz
Pavlova	16 oz
Pavlova	32 oz

Obrázek 2-26 Relace, které obsahují vícehodnotové závislosti, je třeba rozdělit

Ke čtvrté normální formě si musíme říci ještě jednu důležitou věc: do hry totiž vstupuje pouze v případě, že atributy mají více hodnot. Pokud by každý výrobek v našem příkladu měl pouze jednu možnou velikost balení nebo jediného dodavatele, nemělo by čtvrtou normální formu vůbec smysl uvažovat. Podobně pokud dvě množiny atributů nejsou vzájemně nezávislé, pak to nejspíš znamená, že relace poruší již pravidla druhé normální formy.

## Pátá normální forma

Pátá normální forma se týká poměrně vzácného případu *spojené závislosti*. Spojená závislost vyjadřuje cyklické omezení: „pokud je Entita1 spojena s Entitou2, Entita2 je spojena s Entitou3 a Entita3 je spojena zpětně s Entitou1, pak všechny tři entity musí být *mutně* součástí stejněho vektoru hodnot“.

Nyní se tuto větu pokusíme „přeložit do češtiny“. Jestliže například {Dodavatel} dodává {Výrobek}, dále {Zákazník} si objedná výrobek {Výrobek} a dodavatel {Dodavatel} dodává něco zákazníkovi {Zákazník}, pak v podstatě {Dodavatel} dodává {Výrobek} zákazníkovi {Zákazník}. V reálném životě je samozřejmě takováto úvaha nesprávná. Jeden {Dodavatel} může totiž zákazníkovi {Zákazník} dodávat cokoliv, nikoli pouze jeden {Výrobek}. O spojené závislosti hovoříme jen tehdy, pokud je zároveň definováno další omezení, které již uvedené úvaze dává smysl.

V takovéto situaci nestačí definovat jedinou relaci s atributy {Dodavatel, Výrobek, Zákazník}, a to z důvodu problémů při aktualizaci. Jestliže totiž u relace na obrázku 2-27 vložíme například vektor hodnot {„Ma Maison“, „Aniseed Syrup“, „Berglunds snabbköp“}, musíme zároveň vložit i druhý vektor hodnot, {„Exotic Liquids“, „Aniseed Syrup“, „Berglunds snabbköp“}, protože do modelu se nově dostává vazba mezi hodnotami „Aniseed Syrup“ a „Berglunds snabbköp“.

Dodavatel	Výrobek	Zákazník
Exotic Liquids	Aniseed Syrup	Alfreds Futterkiste
Exotic Liquids	Chef Anton's Cajun Seasoning	Berglunds snabbköp

Obrázek 2-27 Tato relace není v páté normální formě

Problém vyřešíme rozložením inkriminované relace do tří různých relací (DodavatelVýrobek, VýrobekZákazník a DodavatelZákazník); tím ale současně vznikají nové problémy, protože při opětovném vytvoření původní relace musíme provést spojení všech tří nových relací. Dočasné spojení pouze dvou relací povede k nesprávným informacím.

To je ovšem z pohledu návrháře systému dosti svízelná situace, protože neexistuje žádný rozumný způsob, jak zajistit spojení tří tabulek; jediná možnost spočívá ve využití bezpečnostních omezení. Navíc, pokud by si uživatel vytvořil dočasnou výslednou množinu, byly by v ní uvedené údaje *zdánlivě* správné; je ovšem velké riziko, že uživatel takovouto chybu prostým pohledem nezjistí.

Takováto cyklická propojená závislost je ale naštěstí natolik vzácná, že komplikace s ní spojené můžeme ve většině běžných situací bezpečně ignorovat. Jestliže je ignorovat nemůžeme, pak nezbývá než postavit databázový systém takovým způsobem, při němž integritu spojení přes více relací zajistíme explicitně.

### Stručné shrnutí

V této kapitole jsme hovořili o struktuře databází z pohledu procesů normalizace. Základním principem celé normalizace je přitom odstraňování redundancí mechanismem bezztrátové dekompozice – to znamená dělením relací bez jakékoli ztráty informací. Formálně tento princip vyjadřuje takzvané normální formy. Nejčastěji se setkáme s první, druhou a třetí normální formou, které bychom mohli shrnout do „kouzelné“ formulky: „Klíč, nic než klíč a celý klíč, pomáhej mi Codd.“ Zbývající tři normální formy se používají pouze ve výjimečných případech.

Následující kapitola se bude věnovat logickému propojení relací, které modelují vztahy mezi entitami.

Kapitola 2 rozebírala proces normalizace datového modelu, jehož součástí je analýza entit v prostoru problému a jehož výsledkem je návrh množiny relací, které efektivně a správně zachycují všechny relevantní údaje. Relace jsou však jen jednou z mnoha součástí datového modelu. Stejně důležité jsou i vazby (asociace) mezi relacemi a omezení, definovaná nad těmito vazbami. V této kapitole budeme tedy hovořit o modelování vztahů. Podobně jako pro definování relací i zde platí, že základní principy se nabízejí docela přímo – stačí správně porozumět sémantice neboli významu datového modelu. Určité speciální případy však do modelu vztahů příliš dobře nezapadají; o některých z nich si povíme později.

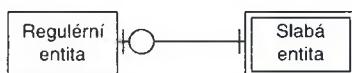
## Terminologie

Vztahy podléhají určité základní terminologii. (Na tomto místě malé vysvětlení: autorka rozlišuje *vztahy*, definované mezi entitami v reálném světě, a *relace*, definované v podstatě jako tabulky v implementované databázi. Pojmem „relace“ se však zpravidla v databázových systémech popisují právě ony „vztahy“ mezi tabulkami a většina tvrzení, kterými zde budeme popisovat vztahy, se prakticky beze změny promítá i do těchto běžných databázových relací. Pozn. překl.) Entity, svázané určitým vztahem, se nazývají *účastníky* vztahu a počet účastníků definujeme jako *stupeň* vztahu. Drtivá většina vztahů je *binárních*, tedy o dvou účastnících, běžné jsou ale také *unární vztahy* (relace svázaná se sebou sama, tedy vztah o jednom účastníku) a někdy se setkáme i s *ternárními vztahy* (tedy se vztahy o třech účastnících). I většina příkladů v této kapitole představuje binární vztahy. O unárních a ternárních vztazích jako speciálních případech si povíme v pozdější části této kapitoly.

Účast dané entity ve vztahu můžeme klasifikovat jako *úplnou účast* nebo jako *částečnou účast*; konkrétní typ účasti závisí na tom, jestli může entita existovat i bez účasti ve vztahu. Máme-li například definovány dvě entity Zákazník a Objednávka, je účast entity Zákazník v jejich vzájemném vztahu jen částečná, protože informace o Zákazníkovi můžeme zadat i bez toho, aniž by musel hned podat první objednávku. Entita Objednávky vykazuje naopak úplnou účast, protože objednávku může podat pouze konkrétní zákazník.

Podle stejného principu se někdy entity také klasifikují jako *slabé* (to jsou entity, které musí mít ve vztazích vždy úplnou účast), respektive *regulérní* (může se vztahu účastnit i částečně). Slabé entity mohou tedy existovat pouze ve vztahu s nějakou jinou entitou, zatímco regulérní entity mohou existovat i samy o sobě, izolovaně. Tato klasifikace se promítá i do metody vytváření entit a vztahů (E/R diagramů), jak ji původně popsal Chen.

Vztahy a entity do nich zapojené můžeme tedy klasifikovat třemi různými způsoby: buďto jako úplnou nebo částečnou účast, jako povinné a volitelné, nebo jako slabé a regulérní entity. Osobně se mi z nich nejvíce líbí klasifikace na povinné a volitelné (nepovinné) entity. V rozsáhlých nebo složitých systémech však může být explicitní vyjádření účinku vztahu na entitu docela užitečné. Jestli je daná entita slabá nebo regulérní, to můžeme v E/R diagramech vyjádřit pomocí notace uvedené na obrázku 3-1.

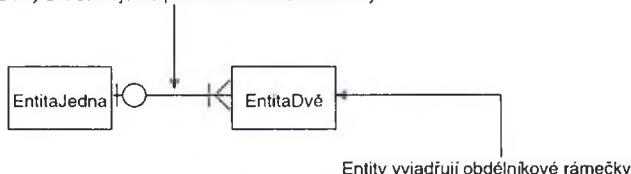


Obrázek 3-1 Touto notací můžeme vyjádřit rozdíl mezi slabou a regulérní entitou

Někdy je dále užitečné rozdělit vztahy na typy „Je“ a „Má“. Základní myšlenka je docela jednoduchá: buďto můžeme říci, že A je B, nebo že A má B. Zaměstnanec je například členem Fotbalového Týmu; stejný Zaměstnanec pak má(n) Adresu. Slovesa „je“ a „má“ ovšem pochopitelně po jazykové stránce velice často k popisu vztahu nestačí. Zaměstnanec tak například „nemá“ Objednávku Prodeje, on ji „vytvoří“. Protože ale zcela bezpečně neplatí, že Zaměstnanec je Objednávka Prodeje, bez velkého přemýšlení zvolíme verzi „má“.

Klasifikace účasti ve vztahu indikuje současně *volitelnost* vztahu, tedy jestli se daná entita musí nebo nemusí účastnit daného vztahu. Volitelnost (nepovinnost) je ovšem docela zálužná, protože databázové stroje ve své implementaci tento okruh problémů vůbec neřeší; podrobněji si o celé záležitosti povíme v kapitole 4 při výkladu implementace datové integrity.

Vztahy znázorňujeme pomocí čar mezi obdélníky.



Jednoduchá čára znamená „jedna“:

„Muří nožka“ znamená „více“:

Kroužek označuje „nepovinnost“  
(můžeme jej také číst jako „nula“):

Symboly se také dají kombinovat; tento  
například znamená „jedna nebo více“:

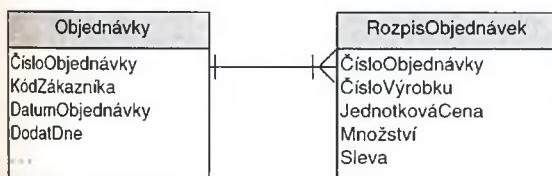
Obrázek 3-2 Pomocí této notace budeme vyjadřovat volitelnost a kardinalitu vztahů

Maximální počet instancí jedné entity, které můžeme asociovat s jednou instancí jiné entity, nazývám *kardinalitou* vztahu. (Všimněte si, že pojmy stupeň a kardinalita mají u vztahů a u relací poněkud odlišný význam.) Existují celkem tři obecné typy kardinality, a sice jedna k jedné, jedna k více a více k více.

Kardinalitu a volitelnost vztahů vyjadřuji pomocí notace, kterou vidíte na obrázku 3-2. Podle mého názoru je „notace muřichů nožek“ (zavedli jsme si ji v kapitole 1) nejvýmluvnější a klientům se nejlépe vysvětluje. Dají se samozřejmě použít i jiné postupy, vždy ale musíme zvolit takový, jaký konkrétnímu případu nejlépe vyhovuje.

## Modelování vztahů

Jakmile zjistíme, že mezi dvěma entitami existuje určitý vztah, musíme jej vhodně namodelovat. K tomu stačí zahrnout příslušné atributy z jedné relace (takzvané *primární relace*) i do druhé relace (které zde říkáme *cizí relace*, případně nevlastní relace), jak to ukazuje obrázek 3-3.



Obrázek 3-3 Relaci modelujeme zahrnutím vhodných atributů z primární relace (Objednávky) do cizí relace (RozpisObjednávek)

Vidíte, že tento diagram vypadá oproti formálnímu E/R diagramu z obrázku 1-6 na straně 21 poněkud odlišně. Za prvé, atributy zde neuvádíme jako samostatné objekty. Na této úrovni návrhu se totiž zajímáme především o vztahy mezi entitami, nikoli o jejich přesnou skladbu. Atributy v diagramu podle mého názoru zbytečně rozptylují a komplikují jej.

Za druhé, vztahy samy o sobě nemají žádný název. Podle mého názoru je označování vztahů názvy či nějakými popisy zcela zbytečné a někdy může být i zavádějící – popis vztahu se totiž liší podle směru, ve kterém jej čteme (učitelé tak *vyučují* studenty, ale studenti *se učí* od učitelů). Já tedy vztahy žádným popisem označovat nebudu, někdy ale vyznačím alespoň atribut, pomocí kterého budeme relaci implementovat v databázovém schématu. To je užitečně například tehdy, pokud má primární entita více než jeden kandidátní klíč a my potřebujeme explicitně ukázat, který z nich se má použít.

Jak jsem již řekla, diagramy tohoto stylu jsou podle mých vlastních zkušeností velice užitečné zejména při práci s klientem; kromě toho se velice snadno nakreslí, a to buďto od ruky, nebo pomocí některého z nástrojů pro tvorbu diagramů, jako je například Visio Professional nebo Micrografx Flowcharter 7. Určité nástroje pro tvorbu diagramů nabízejí ale také Microsoft Access, Microsoft SQL Server a Microsoft Visual Basic; tyto nástroje můžete tedy použít namísto zde uvedených postupů, nebo naopak ve spojení s nimi.

Okno relací programu Microsoft Access (pro soubor databázového stroje Jet), respektive databázové diagramy (u databáze implementované v SQL Serveru) má však jednu podstatnou výhodu:

takto vytvořené diagramy se stanou přímou součástí databáze a automaticky odrážejí změny v ní provedené. To je ale zároveň také největší nevýhoda těchto nástrojů. Nemůžete v nich navrhovat abstraktní diagramy; vždy musíte nejprve vytvořit fyzické tabulky. Proto zde také hrozí nebezpečí že již v raných fázích procesu návrhu, tedy před dokončením implementačního modelu, vklouzne do něčeho, co dokonce *vypadá* jako vlastní implementace.

Já ve své práci často používám jak abstraktní diagramy, tak i konkrétní diagramy svázané s databází. Abstraktní diagramy vytvářím přitom v raných fázích celého návrhu; po dokončení myšlenkového neboli konceptuálního návrhu, když již dokumentuji fyzické databázové schéma, přecházím k vhodnému nástroji Microsoft.

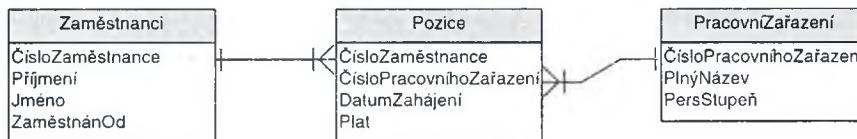
Uvedený postup při modelování relace samozřejmě neznamená, že bychom prostě *jakékoli* atributy vzali z primární relace a zkopiérovali je do cizí relace; musíme zvolit takové atributy, které jednoznačně identifikují primární entitu. Jinými slovy, do cizí relace zde doplňujeme takové atributy, které tvoří kandidátní klíč primární relace. Nebude tedy žádným překvapením, když teď řeknu, že opakovane atributy se v cizí relaci nazývají *cizí klíč* (anglicky foreign key, v české verzi Microsoft Access se setkáte s pojmem nevlastní klíč, pozn. překl.). V příkladu uvedeném na obrázku 3-3 jsme tak do relace RozpisObjednávek doplnili atribut ČísloObjednávky, který je kandidátním klíčem relace Objednávky. Relace Objednávky je zde tedy primární relací, zatímco relace RozpisObjednávek je cizí relací.

### Poznámka

*Dvojice kandidátní klíč/cizí klíč, která danou relaci modeluje, nemusí nutně přímo představovat primární klíč primární tabulky; dobré zde poslouží libovolný kandidátní klíč. V zásadě používáme takový kandidátní klíč, který je ze sémantického pohledu nejvíce smysluplný.*

Primární a cizí relaci zde nemůžeme zvolit zcela libovolně. V prvé řadě ji určuje kardinalita příslušné relace, a za druhé – máme-li jakékoli pochybnosti – sémantika datového modelu. Jestliže jsou například dány dvě relace se vztahem typu jedna k více, bude relace na straně „jedna“ vždy primární relací, zatímco na straně „více“ se nachází vždy cizí relace. To znamená, že kandidátní klíč z relace na straně „jedna“ doplníme jako cizí klíč do relace na straně „více“. O této otázce budeme hovořit podrobněji při rozboru jednotlivých typů vztahů ve zbytku této kapitoly.

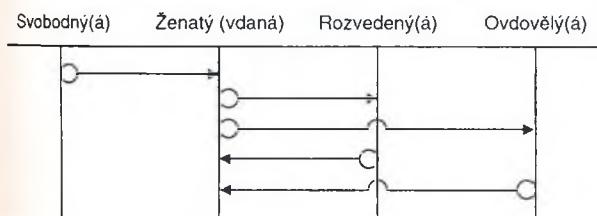
Někdy potřebujeme modelovat nejen samotnou skutečnost existence daného vztahu, ale také určité jeho vlastnosti, například dobu trvání nebo datum zahájení. V tomto případě je užitečné vytvořit pro reprezentaci vztahu speciální abstraktní relaci, jejímž příkladem je na obrázku 3-4 relace Pozice.



Obrázek 3-4 Abstraktní relace mohou modelovat vlastnosti vztahů

Tato technika ovšem jistým způsobem komplikuje výsledný datový model, takže bychom mohli snadno podlehnout pokušení zahrnout atributy vztahu přímo do jedné z účastnických relací. Jestliže ale datový model obsahuje velké množství atributů, nebo hodně relací s atributy, může být tato „úniková cesta“ dosti těžkopádná. Mnohem důležitější je ale to, že s pomocí samostatné entity vztahu je možné snadno sledovat i historii vztahu. Z modelu uvedeného na obrázku 3-4 můžeme tak například zjistit historii pracovních pozic určitého zaměstnance společnosti; pokud by ale Pozice byla definována přímo jako atribut relace Zaměstnanci, nebylo by něco takového možné.

Abstraktní entity relací jsou zajímavé také právě v situacích, kdy potřebujeme sledovat změny určitého vztahu v čase. Na obrázku 3-5 vidíme příklad diagramu stavů a přechodů, který popisuje platné (legální) změny rodinného stavu člověka.



Obrázek 3-5 V diagramu stavů a přechodů se zobrazuje množina platných změn ve stavu určité entity; v tomto případě se jedná o změny rodinného stavu jednotlivce

Pochopit diagramy stavů a přechodů není nikterak obtížné. Každá ze svislých čar znázorňuje určitý stav a platné přechody mezi nimi vyjadřují šipky. Člověk může například přecházet ze stavu ženatý do stavu rozvedený a naopak, ze stavu rozvedený do stavu svobodný však již není návratu. Jestliže tedy nyní potřebujeme modelovat pouze aktuální rodinný stav člověka, dokážeme zajistit, že se budou provádět jen platné změny, i bez implementace nějaké abstraktní entity vztahu. Pokud nás ale bude zajímat, že se Jeníšek a Mařenka Chalupníkovi vzali v roce 1953, potom se rozvedli v roce 1972, Mařenka se znova provdala v roce 1975, ale v roce 1986 ovdověla, pak skutečně nezbývá než pro sledování těchto změn vytvořit abstraktní entitu vztahu.

## Vztahy typu jedna k jedné

Zřejmě nejjednodušším typem vztahu jsou vztahy typu jedna k jedné. O vztahu entit X a Y říkáme, že je typu jedna k jedné, pokud platí, že každou libovolnou instanci entity X je možné asociovat pouze s jedinou instancí entity Y. Typu jedna k jedné bude většina vztahů typu „je“; jinak bychom ale příklady vztahů typu jedna k jedné hledali dosti obtížně. Jestliže mezi určitými entitami zvolíme vztah typu jedna k jedné, musíme mít jistotu, že buďto konkrétní vyjádřený vztah platí neustále, nebo pokud se změní, že nás jeho dřívější hodnoty nezajímají. Řekněme například, že modelujeme kancelářské prostory v určité budově. Budeme-li předpokládat, že v každé kanceláři sedí pouze jeden člověk, dostaváme tím vztah typu jedna k jedné, jak vidíme na obrázku 3-6.

Takový vztah mezi zaměstnancem a jeho kanceláří je ale platný pouze v jednom konkrétním časovém okamžiku. Postupem času dostanou jednu kancelář přidělenu různí zaměstnanci. (Podobně



Obrázek 3-6 Toto je vztah mezi entitami Kancelář a Zaměstnanec, který je typu jedna k jedné

se může změnit i samotné uspořádání kanceláří v budově, to je však již poněkud jiný problém. Jestliže použijeme vztah typu jedna k jedné, jaký vidíme na obrázku 3-6, dostáváme tím jednoduchý, jasný model budovy, nemáme ale žádnou možnost, jak zjistit historii obsazení kanceláří. Třeba nás to ani nezajímá. Pokud například navrhujeme systém pro vnitropodnikovou poštu, chceme vědět, do jaké kanceláře máme Janě Dolejší poslat poštu právě dnes, a ne kam bychom ji byli bývali posílali někdy před třemi měsíci. Jestliže ale vytváříme systém pro správce budovy, nemůžeme tyto informace o historii ztratit, protože správce se bude například systému dotazovat, jak často se nájemci kanceláří mění.

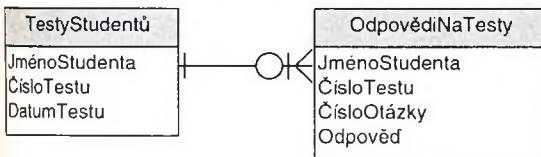
Vztahy typu jedna k jedné jsou sice v reálném světě poměrně vzácné, jako abstraktní pojem jsou však docela běžné a také velice užitečné. Nejčastěji slouží ke snížení počtu atributů v relaci, případně modelují podřídu určité entity. U databázového stroje Jet platí totiž omezení nejvýše 255 polí pro jednu tabulku – a v SQL Serveru je tento počet omezen dokonce na 250. Já osobně mám ale jistou nedůvěru – ba dokonce velkou nedůvěru – vůči jakémukoli datovému modelu, který se do téhoto omezení nevejde. Několikrát za život jsem se ale skutečně setkal a se systémy (obvykle to bylo v medicíně nebo v přírodních vědách), jejichž entity měly více než 255 „pravých“ atributů. V těchto případech již nezbývá nic jiného, než vytvořit novou relaci s jistou libovolně zvolenou množinou atributů a mezi touto novou relací a původní, řídící relací vytvořit vztah jedna k jedné.

Další oblastí problémů, které si *zdánlivě* žádají o překročení fyzických omezení velikosti tabulek je modelování různých testů a dotazníků. Test s libovolným (různým) počtem otázek nás tak může svádět k tomu, abychom odpovědi jednoho člověka (studenta) modelovali podle obrázku 3-7.

Test
JménoStudenta
DatumTestu
Odpověď1
Odpověď2
OdpověďN

Obrázek 3-7 Tato struktura se skutečně někdy používá pro modelování testů a dotazníků, není však právě ideální

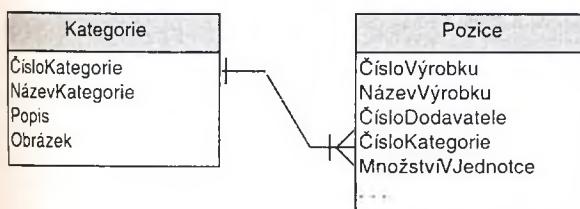
Uvedená struktura se docela snadno implementuje, obvykle to ale není nejlepší řešení. Atributy odpovědí zde totiž tvoří opakovovanou skupinu, a relace tudíž není ani v první normální formě. Lepší model vidíme na obrázku 3-8.



Obrázek 3-8 Druhá struktura se implementuje o něco obtížněji, pro modelování testů a dotazníků je však jednoznačně vhodnější

## Podtřídy entit

Mnohem zajímavější využití vztahů typu jedna k jedné představuje vytváření podtříd entit; to je myšlenka, „vypůjčená“ z oblasti objektově orientovaného programování. Abychom si mohli ukázat výhody vytváření podtříd entit, podíváme se nejprve na klasickou implementaci. V ukázkové databázi Northwind aplikace Microsoft Access je každý výrobek přiřazen do jisté kategorie výrobků, jak to vidíme na obrázku 3-9.

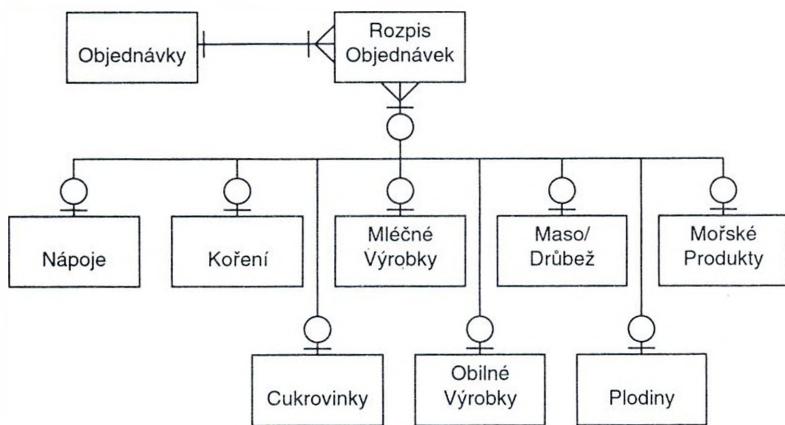


Obrázek 3-9 V databázi Northwind je každý výrobek přiřazen do určité kategorie výrobků

Díky relaci Kategorie můžeme nyní výrobky v různých výstupních sestavách podle kategorií sestupovat; v některých případech náš prostor problému skutečně nic jiného vyžadovat nebude. Při takovém návrhu budeme ale moci s výrobkem pracovat pouze jako s obecným výrobkem, nikoli jako s instancí jeho konkrétní kategorie. Atributy, definované pro Výrobky, má totiž u sebe uložen *jakýkoli* výrobek, bez ohledu na svůj typ. To ale příliš dobře neodpovídá realitě z našeho prostoru problému – Nápoje mají již ze své podstaty jiné atributy než třeba Koření.

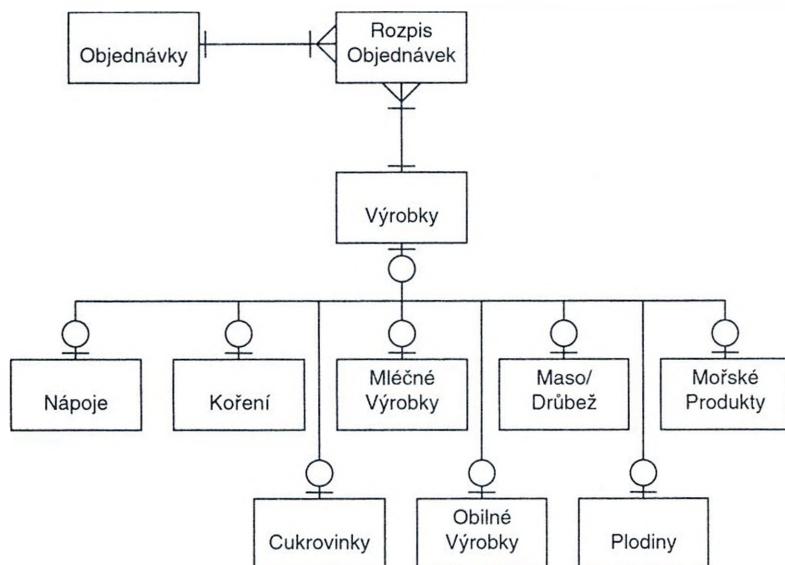
Možná vás proto napadne navrhnut se seznam výrobků v databázi Northwind podle obrázku 3-10. V tomto modelu můžeme skutečně dosti dobře uložit veškeré specifické informace u každého typu výrobku, pracovat s výrobkem jen jako s obecným výrobkem je ale zároveň výrazně obtížnější.

Představte si nyní například, jak bychom museli kontrolovat, jestli uživatel zadal správný kód výrobku: „Jestliže kód existuje v relaci x, nebo v relaci y, nebo v relaci...“. Takový složený test je zhruba stejně ošklivý, jako dotaz s opakovánou skupinou z kapitoly 2. U této struktury se navíc můžete velice snadno dostat do problémů s integritou – uvažujte, co se stane, jestliže se jisté atributy týkají pouze jedné kategorie výrobků (třeba budeme mít atribut JednotekVBalení, který bude platit pro Nápoje, nikoli však pro MléčnéVýrobky), a nyní změníme kategorii určitého výrobku. Co v takovéto situaci uděláme? Zahodíme staré hodnoty? Co když ale uživatel provedl změnu nechtěně, omylem, a posléze se ji bude snažit vrátit zpět?



Obrázek 3-10 Tento model dokáže dobře zachytit atributy platné jen pro danou kategorii

Vytvoření podtříd z entity výrobků spojuje výhody obou těchto „světů“. Při tomto řešení můžeme stále dobře zachytit informace platné pouze pro jisté konkrétní kategorie výrobků, přičemž současně neztrácíme možnost pracovat s výrobkem jako s obecným typem výrobku, pokud to potřebujeme. Odstranění informací, které již nejsou potřeba, můžeme klidně odložit až do okamžiku, kdy bezpečně víme, že je už opravdu potřebovat nebudeme. Model s podtřídami entit ukazuje obrázek 3-11.



Obrázek 3-11 Tento model nabízí s pomocí podtříd entity možnosti obou předcházejících obrázků

Na tomto místě musíme ale říci, že i když je vytváření podtříd entit pro jisté typy problémů datového modelování docela elegantním řešením, někdy se docela pěkně nepříjemně implementuje. Vezměme jako příklad sestavu, která obsahuje podrobné informace o výrobcích. Taková sestava musí obsahovat nějaké podmíněné zpracování, při němž se zobrazí jen ta pole, která odpovídají aktuální podtřídě. To samozřejmě není žádný nepřekonatelný problém, přece jen to ale znamená určitou práci. Ve většině případů nedoporučují datový model „ohýbat“ jen proto, abychom usnadnili život programátorům. Na druhé straně ale zcela zřejmě nemá smysl komplikovat model podtřídami, když je třeba, využijeme jen k seskupení kategorií v nějaké sestavě; v takové situaci úplně stačí (a navíc je mnohem rozumnější) použít strukturu uvedenou na obrázku 3-9.

Identifikovat primární a cizí relaci bývá ve vztahu jedna k jedné někdy dosti ošidné; toto rozhodnutí musí jednoznačně vycházet ze sémantiky datového modelu. Jestliže tedy v konkrétním případu zvolíme strukturu s podtřídami entit, stane se obecná entita primární relací a jednotlivé podtřídy budou tvořit cizí relace.

### Poznámka

*V této situaci bývá často vhodným kandidátním klíčem pro podtřídy také cizí klíč (nevlastní klíč), který příslušná podtřída vyžaduje. Podtřídy mají jen zřídka svoje vlastní identifikátory.*

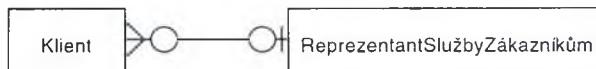
Jestliže ale na druhé straně pomocí vztahu typu jedna k jedné obcházíme omezení počtu polí, nebo jestliže modelujeme entity, v jejichž prostoru problému se skutečně nachází pravý vztah typu jedna k jedné, pak není volba primární relace takto jednoznačná. Vhodnou primární relaci musíme v takovém případě zvolit na základě pochopení daného prostoru problému.

V této situaci nám také může pomoci volitelnost vztahu. Pokud je daný vztah volitelný (nepovinný) jen na jedné straně (a musíme říci, že jsem v životě neviděla vztah, který by byl volitelný na obou stranách), prohlásíme relaci na nepovinné straně za cizí relaci. Jinými slovy, pokud je jedna z entit slabá a druhá regulérní, stane se regulérní entita primární relací vztahu a slabá entita představuje cizí relaci.

## Vztahy typu jedna k více

Nejběžnější typ vztahů mezi entitami představují vztahy typu jedna k více, v nichž můžeme jednu instanci první entity asociovat s nula, jednou nebo více instancemi druhé entity. Ve většině normalizačních postupů, které jsme si ukazovali v kapitole 2, se přitom mezi relacemi vytvářejí právě vztahy typu jedna k více.

Dobře identifikovaný vztah typu jedna k více přináší s sebou jen málokdy nějaké problémy. Je ale velice důležité správně popsat na každé straně vztahu jeho volitelnost (nepovinnost). Lidé se běžně domnívají, že volitelná může být pouze strana „více“ takového vztahu, to ale zdaleka neplatí. Uvažujme například vztah, který vidíme na obrázku 3-12.



Obrázek 3-12 Tento vztah může být volitelný v obou směrech

Vztah mezi entitami Klient a ReprezentantSlužbyZákazníkům je zde v obou směrech volitelný. Českým řečeno to znamená asi toto: „ReprezentantSlužbyZákazníkům může mít nula nebo více klientů. Naopak, ReprezentantSlužbyZákazníkům daného klienta, pokud mu byl přidělen, musí být v relaci ReprezentantSlužbyZákazníkům uveden“. Jestliže ale stranu „jedna“ vztahu typu jedna k více ne definujeme jako volitelnou (nepovinnou), má to doslova zásadní důsledky pro implementaci i pro využitelnost systému. Podrobněji budeme o těchto otázkách hovořit v kapitole 14; prozatím zde ale stačí, když pochopíte, že relační teorie sama o sobě nevyžaduje, aby byla strana „jedna“ vztahu jedna k více povinná.

Identifikovat primární a cizí relaci je ve vztahu typu jedna k více velice snadné. Entita na straně „jedna“ tvoří vždy primární relaci; její kandidátní klíč se zkopiřuje do relace na straně „více“, z níž se tím pádem stává cizí relace. Kandidátní klíč primární relace je velice často součástí kandidátního klíče cizí relace na straně „více“, sám o sobě však jednoznačně identifikovat vektory hodnot cizí relace nikdy nedokáže. Kandidátní klíč cizí relace se z něj může stát pouze po spojení s jedním nebo více vhodnými atributy.

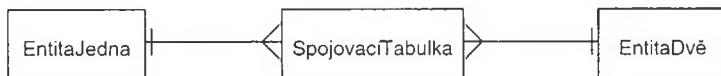
## Vztahy typu více k více

V reálném světě najdeme spoustu příkladů vztahů typu více k více. Studenti mohou mít zapsáno několik různých přednášek a podobně jedné přednášky se může účastnit více různých studentů. Každý zákazník nakupuje v řadě obchodů, a jeden obchod má mnoho zákazníků – nebo alespoň jeho majitel doufá, že bude mít! V relační databázi se však vztahy typu více k více přímo implementovat nedají. Modelují se proto s pomocí speciálního „prostředníka“, speciální mezilehlé relace, která má s každým z obou původních účastníků vztah jedna k více. Celou situaci vidíme na obrázku 3-13. Takový prostředník neboli mezilehlá relace se obvykle nazývá *spojuvací tabulka*, i když při práci na úrovni datového modelu hovoříme o relacích, nikoli o tabulkách.

Tento vztah typu více k více...



... se modeluje následovně:



Obrázek 3-13 Vztahy typu více k více se řeší pomocí mezilehlé (spojuvací) tabulky

Určení primárních a cizích relací již přímo vyplývá z toho, že vztah typu více k více modelujeme jako dva vztahy typu jedna k více. Jak jsme si totiž řekli, ve vztahu jedna k více je relace na straně „jedna“ vždy primární relací. To znamená, že každá z původních entit se v jednom z nových vztahů stane primární relací a spojovací tabulka bude pro oba vztahy cizí relací, přičemž z obou původních relací převezme jejich kandidátní klíče.

Spojovací tabulky obsahují nejčastěji pouze kandidátní klíče dvou původních účastníků; jinak jsou ale ve skutečnosti pouze speciálním případem abstraktních entit vztahu, o nichž jsme hovořili dříve. A jako takové mohou podle potřeby obsahovat jakékoli doplňující atributy.

## **Unární vztahy**

Všechny vztahy, o nichž jsme zatím hovořili, byly binární vztahy se dvěma účastníky. Unární vztahy mají oproti tomu jen jednoho účastníka – relace je tedy spojena se sebou sama. Klasickým příkladem unárního vztahu je vazba Zaměstnance a Nadřízeného. Zaměstnanec má totiž ve většině případů za Nadřízeného dalšího zaměstnance, který sám o sobě mívá zpravidla také nadřízeného.

Unární vztahy se modelují stejným způsobem jako binární vztahy – kandidátní klíč primární relace se tedy přidá do cizí relace. Jediný rozdíl zde spočívá v tom, že primární a cizí relace jsou jedna a tataž relace. Jestliže je tedy kandidátním klíčem relace Zaměstnanci atribut ČísloZaměstnance, deklarovaný v oboru hodnot (doméně) ČísloZaměstnance, můžeme do relace doplnit atribut nazvaný třeba ČísloNadřízeného, který deklarujeme také v oboru hodnot ČísloZaměstnance, jak to vidíme na obrázku 3-14.



Obrázek 3-14 Unární vztah vzniká při spojení relace se sebou samou

Unární vztahy mohou mít přitom libovolnou kardinalitu. Unární vztahy typu jedna k více implementují různé hierarchie, jako je například již uvedená organizační hierarchie Zaměstnanců a jejich Nadřízených. Unární vztahy typu více k více se musí, podobně jako jejich binární protějšky, modelovat pomocí spojovací tabulkы. Na straně „jedna“ mohou být unární vztahy opět volitelné, jak jsme ostatně viděli i na obrázku 3-14. Nejvyšší šéf (tedy ředitel) většiny organizací již nad sebou nikoho nemá. (Akcionáři se nepočítají, ledaže bychom je považovali za nezávislou část datového modelu.)

## Ternární vztahy

Ternární vztahy mají obvykle tvar *X dělá Y pro Z*; podobně jako i běžné vztahy typu více k více se v relační databázi přímo modelovat nedají. Na rozdíl od vztahů více k více však pro jejich modelování neexistuje nějaký jednoznačný předpis.

Na obrázku 3-15 vidíme, že sýr (výrobek) Mozzarella di Giovanni, kterou nakoupil zákazník Vins et alcools Chevalier, dodávají dva dodavatelé – Formaggi Fortini s.r.l. a Forêts d'éables; zde ale nemáme žádnou možnost, jak zjistit, který z nich nám prodal konkrétní sýr, jež jsme posléze prodali společnosti Vins et alcools Chevalier. V tomto datovém modelu se tedy uvedený ternární vztah ztratil. Dodavatelé však nedodávají jen „nějaké“ výrobky; dodávají totiž výrobky, které si nakoupí konkrétní zákazník.

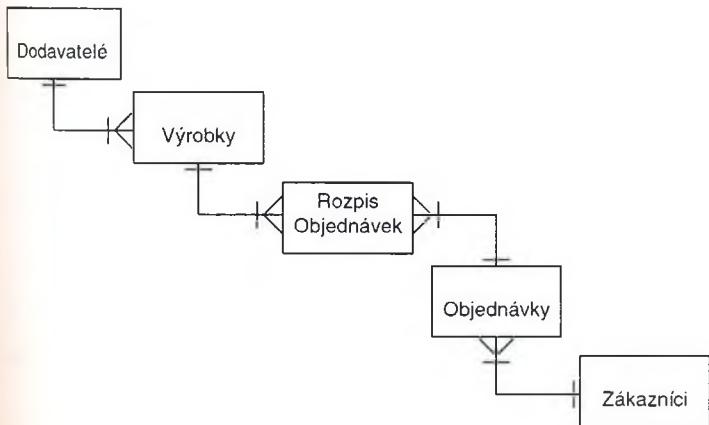
Kód zákazníka	Firma	
VINET	Vins et alcools Chevalier	
Kód zákazníka	Číslo objednávky	Název výrobku
VINET	10248	Queso Cabrales
VINET	10248	Singaporean Hokkien Fried Mee
VINET	10248	Mozzarella di Giovanni
Název výrobku	Firma	
Mozzarella di Giovanni	Formaggi Fortini s.r.l.	
Mozzarella di Giovanni	Forêts d'éables	

Obrázek 3-15 Z těchto relací nepoznáme, od kterého dodavatele pochází sýr, jež u nás zakoupila společnost Vins et alcools Chevalier

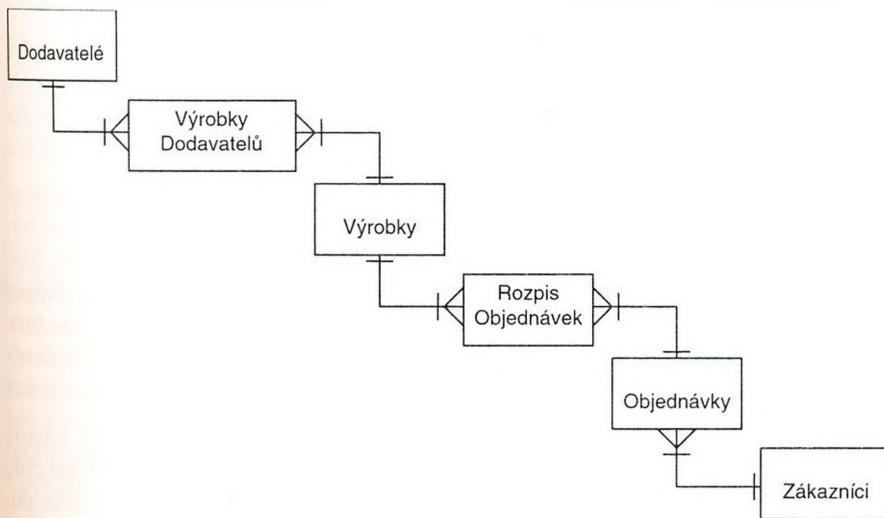
Abychom problém správně pochopili, podíváme se nejprve na model vztahů v obvyklejším prostoru problému, který vidíme na obrázku 3-16.

V tomto diagramu nám každý výrobek dodává jediný stálý dodavatel a ternární vztah tudíž udržovat nemusíme – jestliže známe určitý výrobek, víme také, kdo jej dodává. Na obrázku 3-17 však již každý výrobek může dodávat více různých dodavatelů, potřebný ternární vztah se zde ale ztrácí.

Klíčem k řešení problému je prověřit směr vztahů typu jedna k více. Z libovolné dané entity na straně „více“ určíme jednoznačně odpovídající entitu na straně „jedna“. Podle určené položky re-



Obrázek 3-16 Toto je typický řetězec vztahů s entitami, které se účastní procesů spojených s objednávkou



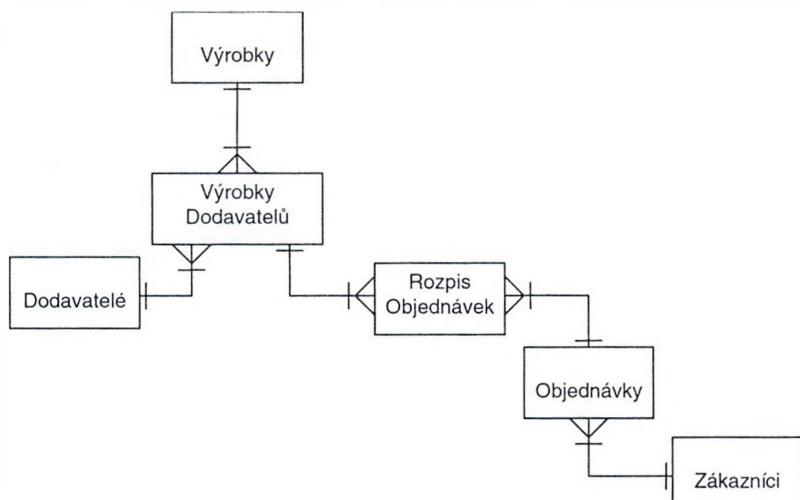
Obrázek 3-17 V tomto modelu se potřebný ternární vztah ztrácí

lace *RozpisObjednávek* na obrázku 3-16 tak zjistíme, ke které položce *Objednávky* náleží; jestliže známe *Objednávku*, máme již také *Zákazníka*. Proces funguje také na druhou stranu: jestliže známe položku relace *RozpisObjednávek*, zjistíme odtud *Výrobek* a posléze i *Dodavatele*.

Opačný postup ale již neplatí. Pokud se nám podaří identifikovat pouze entitu na straně „jedna“ daného vztahu, nemůžeme na straně „více“ vybrat jen jedinou odpovídající entitu. A přesně to je problém obrázku 3-17. Jestliže známe položku relace *RozpisObjednávek*, můžeme zjistit *Výrobek*, avšak z *Výrobku* nezjistíme, se kterou entitou *VýrobkyDodavatelů* je spojen.

Tuto holou skutečnost můžeme snadno shrnout do následujícího pravidla: v řetězci vztahů nemůžeme více než jednou změnit směr z „jedna do více“ na „více do jedna“. V řetězci na obrázku 3-16 se směr skutečně mění jen jednou, a to u relace RozpisObjednávek. U obrázku 3-17 bychom ale již potřebovali změnit směr dvakrát, a sice poprvé u relace RozpisObjednávek a podruhé u relace VýrobkyDodavatelů.

Řešením je vyloučit z celého řetězce entitu Výrobky, jak to vidíme na obrázku 3-18.



Obrázek 3-18 Tento model již ternární vztah zachová

Nyní se směr řetězce vztahů mění skutečně jen jednou, a to u relace RozpisObjednávek, přičemž veškeré vztahy jsou zde zachovány. Všimněte si ale, že entitu Výrobky jsme z modelu neodstranili. V praxi se totiž dá dost dobře očekávat, že si zákazníci nebudou objednávat Výrobky Dodavatelů, nýbrž jen Výrobky; zachováním relace Výrobky nám vzniká vhodné rozhraní k této funkci.

V konkrétním prostoru problému se pochopitelně o ternární vztahy nemusíme vůbec zajímat nebo třeba v případě potřeby najdeme jiný, lepší způsob sledování vztahu, jako je například číslo šarže na balení. Tento konkrétní příklad naprostoto vůbec modelovat nemusíme. Je totiž velice důležité si uvědomit, že model z obrázku 3-18 není nějak sám o sobě lepší nebo přesnější než model uvedený na obrázku 3-17. Opět platí, že si musíme vždy zvolit takový model, který nejlépe vyjadřuje sémantiku daného prostoru problému.

## Vztahy s předem známou kardinalitou

Někdy budeme na straně „více“ určitého vztahu typu jedna k více přesně znát minimální, absolutní (přesný) nebo maximální počet vektorů hodnot. Ve škole má jeden den třeba pět vyučovacích hodin, kostra dospělého člověka obsahuje 233 kostí a golfový hráč smí v průběhu hry použít maximálně 14 holí.

Taková situace vždy svádí k onomu pohodlnému modelování, kdy kandidátní klíč každého vektoru hodnot prohlásíme za atribut relace na straně „jedna“, jak to ukazuje obrázek 3-19; tento postup má ale dva dosti významné problémy. Za prvé takto vzniká opakovaná skupina a za druhé, tento model je nespolehlivý.

CíleStudenta	ČísloStudenta	KresmJméno	Jméno	Příjmení	Jméno	Semestr1:PředmětVýuky	Semestr2:PředmětVýuky	Semestr3:PředmětVýuky	Semestr4:PředmětVýuky
	1	Nancy	Davolio	Biology	Francoúzština	Angličtina	Historie		
	2	Andrew	Fuller	Tělocvik	Biology	Francoúzština	Angličtina		
	3	Janeł	Leverling	Tělocvik	Biology	Francoúzština	Angličtina		
	4	Margaret	Peacock	Francoúzština	Tělocvik	Historie	Angličtina		
	5	Steven	Euchanan	Biology	Francoúzština	Tělocvik	Historie		
	6	Michael	Suyama	Francoúzština	Tělocvik	Biology	Historie		
	7	Robert	King	Histone	Francoúzština	Angličtina	Biology		

Obrázek 3-19 Je lákavé pokusit se modelovat známou kardinalitu takto, je to však nerozumné

Opakovou povahu atributů maskují na obrázku 3-19 jejich různé názvy, všechny jsou ale definovány nad stejným oborem hodnot, PředmětVýuky. Pamatujte si, že kdykoli narazíte na několik atributů, definovaných nad jedním stejným oborem hodnot, je poměrně pravděpodobné, že se pod různými názvy atributů ve skutečnosti skrývají například pouze hodnoty kategorií nebo typů.

Kromě tohoto tak říkajíc teoretického problému trpí ale podobné struktury ještě jedním nešvarem: jsou nespolehlivé. Dejme tomu, že společnost ve své interní politice stanoví, že například pod každého nadřízeného smí spadat nejvýše pět přímých podřízených; takto definovaná politika se ale zdaleka nemusí plně promítat do praxe. Jestliže takovou politiku (či její pravidlo) naopak začleníme do datového modelu, implementujeme ji jako neprůstřelné, neměnné systémové omezení. Vsadím se, že v takovém případě hned při prvotním naplnění datových tabulek přijdete alespoň na jednoho nějakého šéfa, kterému přímo podléhá šest lidí. Co se v takovém případě stane? Musí někdo z nich ihned přejít k jinému šéfovi? Nebo jednoho šéfa zadáme do systému dvakrát (a možná ho i dvakrát zaplatíme)? Anebo vás prostě programátor, kterému zazvoní ve tři hodiny ráno telefon s prosbou o technickou podporu, bude muset velice neslušně proklínat?

Taková omezení kardinality se musí vždy implementovat jako systémová omezení; nikdy se nesmí stát přímou součástí struktury samotných relací. Navíc, jak si podrobněji řekneme v kapitole 16, s implementací různých omezení kardinality je třeba zacházet velice opatrně a vždy si musíme důkladně promyslet možné dopady do použitelnosti výsledného systému.

### Stručné shrnutí

V této kapitole jsme podrobně zkoumali vztahy mezi entitami. Podívali jsme se na všechny jednotlivé typy binárních vztahů – tedy na vztahy typu jedna k jedné, jedna k více a více k více – a ukázali jsme si, jak se reprezentují v datovém modelu; jedna z relací se vždy definuje jako primární, ježíž kandidátní klíč se stane součástí druhého účastníka vztahu, takzvané cizí (nevlastní) relace. Kromě toho jsme hovořili o speciálních případech unárních a ternárních vztahů a řekli jsme si, jak se i ony dají v datovém modelu reprezentovat.

Nyní jsme se tedy již seznámili se všemi základními komponentami datového modelu: s entitami, jejich atributy a se vztahy mezi nimi. V kapitole 4 přejdeme k tématu datové integrity a budeme se zabývat mechanismy, které pomáhají udržovat konzistenci databáze.

# DATOVÁ INTEGRITA

4

Vytvoření modelu entit v daném prostoru problému a definice vztahů mezi nimi je jen jednou z mnoha součástí procesu datového modelování. Zároveň totiž musíme zachytit i pravidla, pomocí nichž hotový databázový systém zajistí, že skutečná fyzická data v něm uložená budou správná, nebo alespoň přinejmenším věrohodná. Jinými slovy, modelovat musíme také *datovou integritu* (integritu dat).

Na tomto místě je důležité si říci, že opravdu zaručit dokonalou *přesnost* dat je velice obtížné, ne-li téměř nemožné. Vezměme například záznam objednávky, podle něhož si jistá Marie Kovářová dne 15. července 1999 zakoupila 17 rámových pil. Databázový systém dokáže ověřit, že je pro něj Marie Kovářová známým zákazníkem, že společnost skutečně prodává rámové pily a že 15. července 1999 byla podána objednávka. Může také zkontolovat, jestli má Marie Kovářová dostatečný úverový rámec pro zaplacení 17 rámových pil. Něco ale již ověřit *nedokážeme*: nevíme, jestli si paní Kovářová ve skutečnosti objednala 17 rámových pil a ne třeba 7 nebo 1, anebo třeba 17 šroubováků. Systém by snad ještě mohl odvodit, že na objednávku rámových pil od osoby jako jednotlivce je počet 17 kusů příliš velký a v tomto smyslu by mohl upozornit uživatele. I takové upozornění by se ale v systému implementovalo dosti nákladně a vynaložené úsilí by neodpovídalo výsledkům.

Podstatná věc, kterou vám zde chci říci, je asi toto: žádný systém nedokáže nikdy ověřit, že Marie Kovářová skutečně podala přesně takovou objednávku, jaká je v něm zaznamenaná; může ověřit jen to, že objednávku *mohla takto podat*. Totéž pochopitelně platí pro jakýkoli záznamový systém; dobré navržený databázový systém nám přitom udělá rozhodně lepší službu než průměrný manuální záznam – když nic jiného, pomocí aplikace vhodných pravidel zabezpečí určitou konzistenci dat. Žádný databázový systém a ani žádný návrhář databázových systémů však nedokáže garantovat, že data v databázi jsou skutečně pravdivá; umí pouze zajistit, že *mohou* být pravdivá. Zajištění této věrohodnosti dat znamená, že data musí vyhovovat jistým *omezením integrity* (omezujícím podmínkám), která pro ně byla definována.

## Omezení integrity

Někteří lidé říkají omezením integrity aplikační pravidla (business pravidla). Aplikační pravidlo představuje ale mnohem širší pojem: zahrnuje veškerá omezení, definovaná v systému, nikoli pouze omezení, která se týkají integrity dat. Zejména bezpečnost systému – ta definuje, kteří uživatelé smí provádět které operace a za jakých okolností – je součástí administrace systému, nikoli datové integrity. Jako taková je ale bezpečnost zcela nepochybně aplikačním požadavkem a tvoří ji jedno nebo více aplikačních pravidel. O otázkách bezpečnosti databází budeme hovořit v kapitole 8, kde budeme rozebírat administraci systému.

Datová integrita se implementuje na několika různě podrobných úrovních. Doménová, přechodová a entitová omezení definují pravidla pro údržbu integrity jednotlivých relací. Omezení referenční integrity zajišťují zachování potřebných vztahů mezi relacemi. Dalším typem jsou omezení databázové integrity, která kontrolují databázi jako celek, a omezení transakční integrity, která řídí způsob manipulace s daty buďto v jedné databázi, nebo i mezi několika databázemi.

## Doménová integrita

Jak jsme si řekli již v kapitole 1, doména neboli obor hodnot je množina všech přípustných hodnot určitého daného atributu. Omezení doménové integrity – kterému se obvykle stručně říká *doménové omezení* – je pak pravidlo, jež definuje tyto platné hodnoty. K úplnému popisu některých domén může být pochopitelně nezbytné definovat i více než jedno doménové omezení.

Doména neboli obor hodnot není totéž co datový typ, takže s pokusem definovat domény jen pomocí fyzických datových typů můžeme docela pěkně pohorjet. Hrozí zde totiž nebezpečí, že povolené hodnoty zbytečně mnoho omezíme – zvolíme například celočíselný typ integer jen proto, že se nám zdá dostatečný, a ne proto, že je 255 skutečně největší povolenou hodnotou dané domény.

Z toho ovšem také vyplývá, že datové typy mohou být v datovém modelu docela šikovnou pomůckou, a proto je prvním krokem při definici doménových omezení v systému často právě výběr logického datového typu. Pod pojmem *logický datový typ* zde přitom rozumíme „datum“, „řetězec“, „obrázek“ a podobně, nic přesnějšího. Datum je zřejmě nejhodnější příklad, na kterém si ukážeme výhody takového postupu. Nedoporučuji tedy definovat DatumTransakce jako typ DateTime, který vyjadřuje právě fyzickou reprezentaci. Jestliže ale DatumTransakce nadefinujeme pouze jako „datum“, můžeme se soustředit na popis podmínek, podle nichž se má jednat o „datum mezi datem zahájení podnikání a dnešním datem včetně“, a můžeme pominout všechna ostatní krkolomná pravidla třeba o přestupných ročích.

Jestliže jsme si zvolili správný logický datový typ, je vhodné dále definovat měřítko a přesnost číselného typu, respektive maximální délku řetězcové hodnoty. To se dosti blíží popisu fyzického datového typu, zde bychom se ale měli stále nacházet na logické úrovni. Samozřejmě není problém, pokud si třeba výraz „řetězcová hodnota o nejvýše 30 znacích“ zkrátíte zrovna na char(30). Čím abstraktnější bude ale popis domén v datovém modelu, tím více manévrovacího prostoru si ponecháváte na pozdější dobu a tím méně budete na systém klást původně nezamýšlená omezení.

Jako další aspekt doménové integrity musíme zvážit, jestli smí daná doména obsahovat také neznámé nebo neexistující hodnoty. Zpracování těchto hodnot je dosti diskutabilní a my se k nim budeme vracet v souvislosti s různými aspekty návrhu databázových systémů. Prozatím je důležité si uvědomit, že je podstatný rozdíl mezi neznámou a neexistující hodnotou a že je skutečně často (i když ne vždy) možné určit, že jedna z nich nebo obě budou součástí domény.

První z obou myšlenek, tedy že „neznámá“ a „neexistující“ hodnoty jsou různé, neznamená na logické úrovni příliš mnoho problémů. (Nezapomeňte opět, že datový model je vždy *logickou* konstrukcí.) Můj otec například nemá druhé jméno, zatímco druhé jméno mých sousedů prostě neznám. To jsou dvě rozdílná tvrzení. Některé implementační otázky nás zatím opravdu nemusí zajímat, tento logický rozdíl je ale opravdu docela jasně pochopitelný.

Druhou myšlenku můžeme vyjádřit takto: jestliže určíme, jestli má doména obsahovat také neznámé a neexistující hodnoty, pak se musíme také rozhodnout, jestli bude systém některé z nich akceptovat. Pokud se vrátíme zpět k našemu příkladu s typem DatumTransakce, zde je docela možné, aby datum transakce bylo neznámé; jestliže ale transakce jednou proběhla, musela proběhnout v nějaký pevný časový okamžik a její datum nemůže být tedy neexistující. Jinými slovy, datum transakce musí být definováno, my jej pouze nemusíme znát.

Nyní bychom si mohli říci, že jakoukoli hodnotu můžeme prostě neznát, takže i každá hodnota může být neznámá. To ale není příliš smysluplné tvrzení. Zde totiž definujeme něco jiného: nezájímá nás to, jestli určitá hodnota může být sama o sobě neznámá, nýbrž to, jestli se smí jako neznámá také uložit. V konkrétním případě tak třeba nemá smysl ukládat data s neznámou hodnotou, nebo určitou entitu nedokázeme bez znalosti příslušné hodnoty identifikovat. V obou popsáncích situacích bychom záZNAM, který má v určeném poli neznámou hodnotu, nedovolili vložit do databáze.

Toto rozhodnutí se přitom nedá vždy provést přímo na úrovni domény, rozhodně je ale vhodné se o to alespoň pokusit, protože takto si usnadníme další práci. Konkrétní rozhodnutí do jisté míry závisí také na tom, nakolik jsou naše domény (obory hodnot) obecné neboli generické. Jako příklad si uvedeme doménu Jméno, nad kterou definujeme atributy KřestníJméno, DruhéJméno, Příjmení a JménoFirmy. Všechny tyto atributy bychom mohli zrovna tak definovat nad samostatnými doménami; jedna společná, obecnější doména má však určité výhody, protože dokáže na jednom místě postihnout veškerá společná pravidla (a v tomto případě je jich více než dost). Na druhé straně již ale takto nemůžeme na úrovni domény stanovit, jestli jsou přijatelné i prázdne a neznámé hodnoty; tyto vlastnosti nezbývá než popsat na úrovni jednotlivých entit.

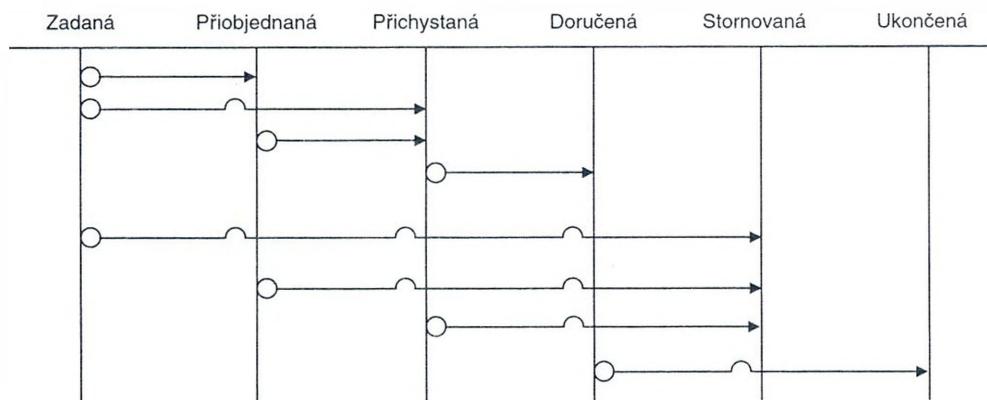
Poslední myšlenkou doménové integrity je, že množinu hodnot reprezentovaných určitou doménou je třeba popsat co nejkonkrétněji. Naše doména nazvaná DatumTransakce tak například není jednoduše množinou všech přípustných datumů; je to množina datumů ode dne, kdy naše společnost zahájila činnost, do dnešního dne. Můžeme ji přitom dále omezit a zakázat v ní třeba neděle, státní svátky a jiné dny, kdy firma nepracuje.

Někdy se dá doména popsat přímo seznamem platných hodnot. Doménu Víkend například úplně popisuje množina {"sobota", „neděle“}. Jindy je jednodušší zapsat jedno nebo více pravidel, která určují členství hodnoty v doméně, jako tomu bylo například u domény DatumTransakce. Oba postupy jsou v zásadě přípustné, i když jisté metodologie návrhu mohou přímo předepisovat konkrétní metodu popisu omezení. Každopádně je ale důležité zachytit veškerá omezení co nejpečlivěji a nejúplněji.

## Přechodová integrita

Omezení přechodové integrity definují stav, kterými může vektor hodnot právoplatně přecházet. Diagram stavů a přechodů, uvedený na obrázku 4-1, ukazuje například stav, jimiž může procházet objednávka.

Zde bychom tedy pomocí omezení přechodové integrity mohli například zajistit, že se stav určité objednávky nezmění ze „Zadané“ přímo na „Ukončenou“ bez průchodu ostatními mezilehlými stavů, nebo že se stav stornované objednávky již nikdy nezmění.



Obrázek 4-1 Tento diagram ukazuje stavy, kterými smí procházet objednávka

Stav entity kontroluje obvykle jediný atribut. V takovém případě můžeme přechodovou integraci považovat za jistý speciální typ doménové integrity. Někdy ale platnost přechodů kontroluje několik atributů nebo dokonce několik relací. A protože přechodová omezení mohou být definována na libovolné úrovni podrobnosti, je vhodné je při přípravě datového modelu považovat za samostatný typ omezení.

Změna stavu zákazníka z „Normálního“ na „Přednostního“ může být například povolena jen v případě, že jeho úvěrový limit dosahuje předepsané minimální hodnoty a že s naší společností obchoduje více než rok. Velikost úvěrového limitu zde bude nejspíše kontrolovat určitý atribut relace Zákazníci, zatímco délka období, po které zákazník s naší společností obchoduje, nemusí být nikde explicitně uložena, takže ji třeba budeme muset vypočítat podle data nejstaršího záznamu tohoto zákazníka v relaci Objednávky.

### Entitová integrity

*Entitová omezení* zabezpečují integritu entit modelovaných v daném systému. Na nejjednodušší úrovni můžeme za takovéto entitové omezení považovat existenci primárního klíče: ten totiž zajišťuje pravidlo, podle něhož „každou entitu musí být možné jednoznačně identifikovat“.

Toto je v jistém slova smyslu jediné *pravé omezení entitové integrity*; všechna ostatní jsou již technicky vzato pouze omezeními integrity, definovanými na úrovni entity. Tato omezení, definovaná na úrovni entity, mohou přitom kontrolovat jeden atribut, několik atributů, nebo relaci jako celek.

Integritu jednotlivého atributu modelujeme především jeho definicí v určité doméně (oboru hodnot). Atribut v dané relaci tak zdědí veškerá omezení integrity definovaná pro jeho doménu. Na úrovni entity pak můžeme tato zděděná omezení podle potřeby dále zpřísnit, nemůžeme je však v žádném případě zpět uvolnit. Dá se tedy říci, že entitové omezení může specifikovat podmínku množiny definované doménovým omezením, nikoli však jeho nadmnožinu. Jestliže například atribut DatumObjednávky nadefinujeme nad doménou DatumTransakce, můžeme u něj dál stavovit, že datum objednávky musí spadat do aktuálního roku, zatímco DatumTransakce povoluje jakékoli datum mezi okamžikem zahájení podnikání a dnešním dnem. Ani entitové omezení však

nesmí v atributu DatumObjednávky povolit budoucí datum (byť z letošního roku), protože to již zakazuje omezení definované v doméně daného atributu.

Podobně můžeme například v atributu JménoSpolečnosti, definovaném nad doménou Jméno, zakázat prázdné hodnoty, přestože je doména Jméno povoluje. I toto omezení popisuje opět užší, přísněji definovanou množinu platných hodnot, než jaká je určena v samotné doméně.

### Poznámka

*Návrháři databází často popisují platnost (povolení) prázdných a neznámych hodnot raději na úrovni entity než na úrovni domény. Některí návrháři by mohli dokonce namítat, že tato omezení mají smysl pouze na úrovni entity. Tento názor je do jisté míry oprávněný, osomně ale doporučuji nadefinovat doménu co nejúplněji. Zcela jistě je pravda, že pokud uvažujeme o prázdných a neplatných hodnotách již na úrovni domény, ničemu to neradí a proces celé specifikace (i implementace) si tak můžeme jen zjednodušit.*

Kromě zúžení intervalu hodnot jednoho určitého atributu může entitové omezení zasahovat také do několika různých atributů. Příkladem takového omezení je například požadavek, podle něhož musí být DatumOdeslání pozdější nebo rovno jako DatumObjednávky. Entitová omezení se však nemohou odvolávat na jiné relace. To znamená, že jako entitové omezení nemůžeme definovat například mechanismus, který definuje ProcentoSlevy zákazníka (což je atribut relace Zákazníci) podle hodnoty CelkovýProdej tohoto zákazníka (ten totiž vychází z více záznamů relace RozpisObjednávek). Omezení, která jsou závislá na několika relacích, jsou vždy omezení na úrovni databáze; za chvíli si o nich v této kapitole povíme více.

S omezeními, která zasahují do několika atributů, zacházejte opatrně; někdy totiž mohou známenat, že datový model není plně normalizován. Jestliže omezujeme nebo vypočítáme hodnotu jednoho atributu podle hodnoty druhého atributu, je nejspíše všechno v pořádku. Entitové omezení, které říká, že „Stav zákazníka nesmí být ‘Přednostní’, pokud jeho záznam není alespoň jeden rok starý“, je definováno správně. Pokud ale hodnota jednoho atributu přímo určuje hodnotu jiného – tedy pokud bychom například řekli, „Jestliže je záznam zákazníka alespoň jeden rok starý, pak nechť Stav = ‘Přednostní’“ – dostáváme tím funkční závislost a porušujeme podmínky třetí normální formy.

### Referenční integrita

V kapitole 3 jsme si ukazovali dekompozici relací, jejímž smyslem bylo minimalizovat možné redundancy, a hovořili jsme o cizích (nevlastních) klíčích, které implementují vazby mezi relacemi. Jestliže se tyto vazby přeruší, bude systém přinejlepším nespolehlivý a přinejhorším zcela nepoužitelný. Tyto vazby udržují a ochraňují takzvaná *omezení referenční integrity*.

Dá se říci, že v podstatě existuje jen jediné omezení referenční integrity: cizí klíče se nesmí stát sirotky. Jinými slovy, žádný řádek v cizí tabulce nesmí obsahovat takovou hodnotu cizího klíče, která nemá odpovídající záznam v primární tabulce. Vektorům hodnot, které obsahují takovéto cizí klíče bez odpovídajícího kandidátního klíče v primární relaci, se nazývají *sirotci (sirotčí entity)*. K vytvoření sirotka může dojít jedním ze tří následujících způsobů:

- Do cizí (nevlastní) tabulky se přidá vektor hodnot s klíčem, který neodpovídá žádnému kandidátnímu klíči v primární tabulce.
- Kandidátní klíč v primární tabulce se změní.
- Záznam primární tabulky, na který se sirotek odkazuje, se odstraní.

Máme-li tedy zachovat integritu daného vztahu, musíme správně ošetřit všechny tyto uvedené případy. První případ, tedy přidání neodpovídajícího cizího klíče, je obvykle přímo zakázán. Uvědomte si ale, že se zde nepočítá s neznámými a neexistujícími hodnotami. Pokud je daný vztah deklarován jako volitelný, můžeme do cizí relace zapsat libovolný počet neznámých a neexistujících hodnot, a to bez narušení pravidel referenční integrity.

K situaci, která je druhou z možných příčin vzniku sirotčí entity – tedy ke změně hodnoty kandidátního klíče v odkazované tabulce – by nemělo docházet příliš často. Já dokonce silně doporučuji veškeré změny kandidátních klíčů pokud možno zcela zakázat. (Mimochedem, toto by mohlo být jedno entitové omezení: „Kandidátní klíče se nesmí měnit.“) Jestliže ale příslušný datový model povoluje změnu kandidátního klíče, musíme zajistit promítnutí odpovídající změny i v hodnotách cizích klíčů. Tomuto mechanismu se říká *kaskádovitá aktualizace*. Databázové stroje Microsoft Jet i Microsoft SQL Server nabízejí přitom mechanismus, pomocí něhož se kaskádovité aktualizace implementují velice snadno.

Poslední příčinou vzniku sirotka je odstranění vektoru hodnot, který obsahuje primární entitu. Jestliže někdo odstraní například záznam Zákazníka, co se stane s objednávkami tohoto zákazníka? Podobně jako u změny kandidátního klíče můžeme být i zde přísní a odstranění vektoru hodnot primární relace, na který se odkazuje nějaká cizí relace, jednoduše zakázat. To je zcela jistě nejčistší řešení – pochopitelně pokud si to v daném systému můžeme dovolit (pokud je to pro ně „rozumné“ omezení). Jestliže něco takového není možné, můžeme opět využít mechanismu databázového stroje Jet, respektive SQL Serveru, který operaci rozšíří v kaskádě; zde se operace pochopitelně nazývá *kaskádovité odstranění*.

V tomto případě máme ale ještě jednu, třetí možnost, která se ovšem o něco obtížněji implementuje. Každopádně se nedá implementovat automaticky. Můžeme totiž všechny závislé záznamy přeřadit jinam. To není zcela vždy žádoucí, někdy se tomu ale nevyhneme. Řekněme například, že firma ZákazníkA koupí firmu ZákazníkB. V takovémto případě může mít smysl odstranit veškeré záznamy ZákazníkaB a objednávky ZákazníkaB přeřadit k ZákazníkoviA.

Dalším speciálním typem omezení referenční integrity je podmínka maximální kardinality, o kterou jsme hovořili v kapitole 3. Pravidla jako například „jeden nadřízený smí mít nejvýše pět přímých podřízených“ se v datovém modelu definují právě jako omezení referenční integrity.

## Databázová integrity

Nejobecnější formou omezení integrity je takzvané *databázové omezení*. V databázových omezeních se odkazujeme na více než jednu relaci: „Stav zákazníka nesmí být ‘Přednostní’, pokud během posledních 12 měsíců neuskutečnil alespoň jeden nákup.“ Většina databázových omezení má právě takový tvar.

Veškerá omezení integrity je vhodné vždy definovat co nejúplněji, a ani databázová integrity nemá žádnou výjimkou. Dávejte si ale pozor a nezaměňujte databázové omezení se specifikací pracov-

ního procesu. *Pracovní proces* je něco, co se provádí v databázi, jako například přidání nové objednávky, zatímco omezení integrity je pravidlo, které se týká obsahu databáze. Pravidla, která definují úkoly prováděné s pomocí databáze, jsou omezení pracovních procesů, nikoli databázová omezení. Pracovní procesy, jak uvidíme v kapitole 8, mohou mít také zásadní dopady do datového modelu, nesmí se ale stát jeho přímou součástí.

Z daného aplikačního pravidla není ale vždy zcela zřejmé, jestli se jedná o omezení integrity, nebo o pracovní proces (anebo ještě o něco úplně jiného). Rozdíl ani nemusí být nijak zásadně důležitý. Pokud na tom v podstatě nezáleží, můžete pravidlo implementovat na tom místě, kde je to pro vás nejvhodnější. Jestliže se přímo nabízí vyjádřit pravidlo jako databázové omezení, nadefinujte jej přesně tak. Pokud je tento postup obtížný (což bývá často, a to i když je dané pravidlo jasné omezením integrity), přesuňte jej do uživatelského rozhraní front-end, kde jej implementujete procedurálně.

Jestliže je ale pravidlo na druhé straně velice nestálé a podléhá častým změnám, bude možná snazší jej udržovat jako součást databázového schématu, kde se jediná jeho změna okamžitě promítne do všech systémů, které se na ně odkazují (doufejme, že je tato změna *nenaruš*).

## **Transakční integrita**

Poslední formou databázové integrity je *transakční integrita*. Omezení transakční integrity ovládají způsoby přípustné manipulace s databází. Transakční omezení jsou na rozdíl od ostatních omezení procedurálního charakteru, a proto jako takové nejsou součástí datového modelu.

Transakce úzce souvisejí s pracovními procesy. Jejich základní myšlenky jsou ve skutečnosti zcela transparentní, protože jeden daný pracovní proces se může skládat z jedné nebo více transakcí a naopak. Pracovní proces si můžeme představit jako jistou abstraktní konstrukci („přidání objednávky“) a transakci jako fyzickou konstrukci („aktualizace tabulky RozpisObjednávek“) – toto přirovnání není sice úplně korektní, přesto je ale docela užitečné.

Transakce se obvykle definuje jako „logická jednotka práce“; taková definice je ale podle mého názoru k ničemu. Transakce je totiž v podstatě jistá nedělitelná skupina určitých operací (úkonů), které se budто musí všechny společně dokončit, nebo se nesmí dokončit žádná z nich. Před zahájením transakce a po jejím ukončení musí být v databázi splněna veškerá definovaná omezení integrity, *během* zpracování transakce však mohou být některá z nich dočasně porušena.

Klasickým příkladem transakce je převod peněz z jednoho bankovního účtu na druhý. Jestliže se příslušná částka odečte z ÚčtuA, ale poté se systému nepodaří ji přičíst na nový ÚčetB, znamená to, že se peníze po cestě ztratily. Problém má zcela zřejmě následující řešení: jestliže druhý z překazů selže, musí se zrušit i první operace. V databázové hantýrce se říká, že se operace „vrátí zpět“ (roll back).

Do jedné transakce může být zapojeno několik různých záznamů, několik různých relací, a dokonce i několik různých databází. Přesněji řečeno platí, že všechny operace, prováděné nad databází, jsou transakcemi. Transakci je tedy i pouhá aktualizace jediného existujícího záznamu. Tyto transakce na nízké úrovni však naštěstí za nás provádí transparentním způsobem samotný databázový stroj, takže na této úrovni podrobnosti se jimi obvykle zabývat nemusíme.

Prostředky pro údržbu transakční integrity nabízí jak databázový stroj Jet, tak i SQL Server; v obou případech jsou jimi příkazy BEGIN TRANSACTION, COMMIT TRANSACTION a ROLLBACK TRANSACTION. Jak možná sami očekáváte, implementace v SQL Serveru je z těchto dvou systémů robustnější a je lépe schopna se zotavit z hardwarových výpadků i z jistých typů softwarových havárií. To jsou ale již implementační otázky, které jsou mimo rámec této knihy. Z pohledu návrháře je důležité správně zachytit a specifikovat transakční závislosti, tedy závislosti oněch neslavných „logických jednotek práce“.

## Implementace datové integrity

Zatím jsme se v našem výkladu zaměřili na vhodné zachycení předloženého prostoru problému na abstraktní úrovni, v myšlenkovém datovém modelu. V této části textu si řekneme několik slov k určitým otázkám, které souvisí s vytvořením fyzického modelu prostoru problému: budeme se věnovat databázovému schématu. Přechod z jedné úrovni do druhé má zde především podobu změněné terminologie – z relací se stávají tabulky a atributy se nazývají pole – výjimkou je však datová integrita. Její mechanismy se nikdy nemapují tak jasně a přesně, jak bychom si přáli.

### Neznámé a neexistující hodnoty (ještě jednou)

Někde ze začátku této kapitoly jsem tak trochu v dobrém rozmaru prohlásila, že u domén (oborů hodnot) a atributů se máme pokusit zjistit, jestli povolují prázdné nebo neznámé hodnoty, přičemž o možné implementaci výsledných omezení bychom ještě uvažovat neměli. Jakmile se ale dostaneme do databázového schématu, problému implementace (a musím předeslat, že to je problém) se nevyhneme.

O onom známém „problému chybějící informace“ se přitom vědělo již dávno před zavedením nějakého relačního modelu. Jak máme v systému říci, že jedna určitá informace buďto chybí (zákazník má příjmení, my ho ale neznáme), nebo vůbec neexistuje (zákazník nemá druhé jméno)? Většina relačních databází, včetně databází stroje Microsoft Jet a databází SQL Serveru, zavádí pro oštětření chybějících a neexistujících hodnot prázdné hodnoty Null.

Považovat hodnotu Null za nějaké spásné řešení tohoto problému je poněkud přehnané, protože má sama o sobě řadu problémů. Někteří databázoví experti hodnoty Null zcela zavrhují. C. J. Date říká, že hodnoty Null „rozbijejí celý model“ a ani já sama si už nepamatuji, kolikrát jsem o nich slyšela, že jsou prostě jedno velké „zlo“. Jakékoli poznámky o složitosti ošetření hodnot Null či kajícné rady, jak se jimi nenechat zaskočit, vedou nakonec ke slovům jako: „Dobrá. Raději je ne používejte. Ublížily by vám.“

Škola „hodnoty Null jsou zlo“ doporučuje pro označení neznámých nebo neexistujících hodnot (případně obojích) použít speciální hodnoty z vhodné domény. Tento postup bych asi nazvala *přístup se smluvou hodnotou*. Přístup se smluvou hodnotou s sebou ale přináší hned několik problémů. Za prvé, v řadě případů opravdu zvolená hodnota je opravdu pouze smluvou. Datum 9. 9. 1900 neznamená samo o sobě neznámý údaj; to jen my jsme se takto dohodli na její interpretaci. Osobně nechápu, v čem má být tato metoda lepší než hodnoty Null. Prázdná hodnota Null je samozřejmě také jistá smluvená hodnota, nedá se však splést s ničím jiným a navíc má tu výhodu, že ji přímo podporuje relační model a většina relačních databázových strojů.

Druhý problém, díky kterému je smluvená hodnota podle mě zcela nepoužitelná, je její dopad na referenční integritu. Vezměme například volitelný vztah mezi relacemi Zákazníci a ReprezentantSlužbyZákazníkům (RSZ). Jestliže je zákazníkovi nějaký RSZ přidělen, pak musí být uveden v tabulce reprezentantů RSZ. U postupu se smluvenou hodnotou musíme do tabulky RSZ doplnit speciální záznam se zvolenou, smluvenou hodnotou, která bude označovat nepřiřazeného RSZ, jak to ukazuje obrázek 4-2.

#### Zákazníci

Kód zákazníka	Firma	RSZ
ALFKI	Alfreds Futterkiste	Nancy Davolio
ANATR	Ana Trujillo Emparedados y helados	Laura Callahan
ANTON	Antonio Moreno Taqueria	Robert King
AROUT	Around the Horn	Steven Buchanan
BERGS	Berglunds snabbköp	NEPŘIDĚLEN
BLAUS	Blauer See Delikatessen	Laura Callahan
BLONP	Blondel pere et fils	Margaret Peacock
BOLID	Bólido Conidas preparadas	Steven Buchanan
BONAP	Bon app'	Nancy Davolio
BOTTM	Bottom-Dollar Markets	Nancy Davolio
BSBEV	B's Beverages	Laura Callahan
CACTU	Cactus Comidas para llevar	Robert King
CENTC	Centro comercial Moctezuma	NEPŘIDĚLEN
COMMI	Comércio Mineiro	NEPŘIDĚLEN
CONSH	Consolidated Holdings	Michael Suyama
DRACO	Drachenblut Delikatessen	Robert King
DUMON	Du monde entier	Laura Callahan
EASTC	Eastern Connection	Laura Callahan
ERNSH	Ernst Handel	Janet Leverling
FAMIL	Familia Arquibaldo	Andrew Fuller
FISSA	FISSA Fabrica Inter. Salchichas S.A.	Robert King
FOJUG	Folies gourmandes	Janet Leverling
FOLKO	Folk och fa HB	Steven Buchanan
FRANK	Frankenversand	Andrew Fuller
FRANR	France restauration	Robert King
FRANS	Franchi S.p.A.	NEPŘIDĚLEN

#### RSZ

Cílo zaměstnance	Příjmení	Jméno
1 Davolio	Nancy	
2 Fuller	Andrew	
3 Leverling	Jane	
4 Peacock	Margaret	
5 Buchanan	Steven	
6 Suyama	Michael	
7 King	Robert	
8 Callahan	Laura	
9 Dodsworth	Anne	
10 NEPŘIDĚLEN		

Obrázek 4-2 U smluvené hodnoty musíme z důvodu údržby referenční integrity doplnit „falešný“ záznam

A teď mi řekněte, kolik RSZ v dané společnosti pracuje? O jednoho méně, než kolik je RSZ uvedených v tabulce, protože jeden z nich představuje falešný záznam. Ouvez. Jaký je průměrný počet zákazníků na jednoho zaměstnance RSZ? Počet záznamů v tabulce Zákazníci minus počet záznamů, které odpovídají reprezentantovi (RSZ) „NEPŘIDĚLEN“, děleno počtem záznamů v tabulce RSZ minus jedna. Dvakrát ouvez.

Smluvené hodnoty mají ale svůj smysl například při generování výstupních sestav. Za hodnoty Null můžeme například dosadit text „Neznámý“ a namísto prázdných hodnot vypsat „Nedefinovaný“. To je samozřejmě zcela jiná situace a zcela jiná záležitost, než ukládání těchto smluvených hodnot přímo do databáze, kde se „tlučou“ s mechanismy datové manipulace, jak jsme si již ukázali.

Hodnoty Null jsou možná jistým zlem a zcela nepochybně jsou ošklivé, pro ošetření neznámých a neexistujících hodnot ale bohužel představují nejlepší dostupný nástroj. Pokuste se tedy celý problém ještě jednou důkladně promyslet, najděte alternativy, pokud jsou rozumné, a jestliže rozumné nejsou, pak nezbývá než se smířit i s jistými obtížemi hodnot Null.

Jedním z problémů hodnot Null je, že s výjimkou domén deklarovaných jako řetězcové nebo textové datové typy mohou být nuteny sloužit dvojím způsobem. Pole deklarované s datovým typem DateTime dokáže přijímat pouze datumové údaje nebo hodnoty Null. Jestliže odpovídající atribut povoluje podle své definice jak neznámé, tak i neexistující hodnoty, přičemž oba tyto typy reprezentuje jediná hodnota Null, nemáme žádnou možnost, jak zjistit, jestli hodnota Null v daném konkrétním záznamu vyjadřuje „neznámou“ nebo „neexistující“ hodnotu. U řetězcových a textových datových typů se s tímto problémem nesetkáme, protože jako prázdnou hodnotu můžeme snadno použít prázdný řetězec o nulové délce, takže hodnota Null vyjadřuje pouze neznámou hodnotu.

V praxi se ale tento problém nevyskytuje až tak často, jak by se dalo očekávat. Neexistující hodnoty povolují i některé netextové domény, ve kterých můžeme tím pádem hodnotu Null vždy jednoznačně interpretovat jako neznámý údaj. Jestliže doména neexistující hodnotu neakceptuje, můžeme pro její vyjádření často zvolit jinou rozumnou alternativu. Upozorňuji vás ale, že na tomto místě doporučuji jistou skutečnou hodnotu, nikoli smluvnou hodnotu. Relace Výrobky má sice například atribut Váha, v jiném atributu VoláníSlužby, který zcela zřejmě žádnou váhu nemá, však již nulovou hodnotu použít můžeme. (U řady číselných polí je nula skutečně vhodným kandidátem pro vyjádření prázdného údaje, vždy se ale také použít nedá.)

Druhý a mnohem závažnější problém hodnot Null je, že dosti komplikují manipulace s daty. Operace logického porovnání jsou s nimi složitější a sestavení určitých typů dotazů je opravdu komplikované. Podrobněji se na tento problém podíváme v kapitole 5.

S hodnotami Null proto nezacházím nijak lehkomyslně a pokud k nim najdeme nějakou rozumnou, smysluplnou alternativu, doporučuji ji rozhodně vzít. Jak jsem již ale někde řekla (a rozhodně to stojí za zopakování), nesnažte se datový model zbytečně ohýbat jen proto, abyste programátorům usnadnili život. Promyslete jej opravdu důkladně a pokud zjistíte, že váš systém prostě hodnoty Null potřebuje, použijte je.

## Reakce na porušení omezení

Při definici databázového schématu musíme nejen určit, jakým nejfektivnějším způsobem se může dané omezení integrity implementovat, ale také rozhodnout, jakou operaci má databázový stroj provést v případě, že dojde k porušení tohoto omezení. Ve většině případů databáze samozřejmě inkriminovaný příkaz jednoduše odmítne provést a vyvolá chybu. Někdy může ale databáze vykonat určitá nápravná opatření, po jejichž přijetí je již původně požadovaná změna přijatelná. Příkladem takového opatření může být přiřazení implicitní hodnoty do atributu, který nepovoluje prázdné hodnoty, nebo provedení kaskádovité aktualizace či kaskádovitého odstranění, které zachová potřebnou referenční integritu. O možných reakcích na porušení platných omezení budeme podrobněji hovořit v části 3 této knihy.

## Deklarativní a procedurální integrita

Relační databázové stroje podporují dva typy mechanismů integrity: je to deklarativní a procedurální integrita. Podpora *deklarativní integrity* je explicitně definována („deklarována“) jako součást databázového schématu. Určitou podporu deklarativní integrity nabízí přitom jak databázový stroj Microsoft Jet, tak i SQL Server. Deklarativní integrita je vhodná zejména pro implementaci datové integrity. Proto je dobré ji používat pokud možno všude.

SQL Server podporuje také *procedurální integritu*; definuje totiž speciální procedury, nazývané *sponště*(triggers), které se vyvolávají („sponštějí“) automaticky při každém vložení, aktualizaci, případně odstranění záznamu. Databázový stroj Jet však žádné sponště, ani jinou formu procedurální integrity nenabízí. Jestliže v něm nějaké omezení integrity nemůžeme implementovat pomocí mechanismů deklarativní integrity, musíme je implementovat v uživatelské front-end části.

Ve zbytku této kapitoly se budeme věnovat specifikum mapování různých omezení integrity, definovaných v datovém modelu, na fyzické databázové schéma.

## Doménová integrita

SQL Server podporuje definici vlastních domén neboli oborů hodnot v jisté omezené míře. Tuto podporu představují takzvané uživatelem definované datové typy (user-defined data types, UDDT). Pole definovaná nad uživatelsky definovaným datovým typem zdědí jak typovou deklaraci, tak i doménová omezení definovaná pro daný UDDT.

Další neméně důležitá věc je, že SQL Server nepovoluje operace porovnání mezi polí deklarovanými nad dvěma různými uživatelsky definovanými datovými typy, a to i když jsou oba tyto typy založeny nad stejným systémovým datovým typem. Přestože tedy například domény JménoMěsta a JménoSpolečnosti definujeme shodně jako 30znakový řetězec neboli char(30), operaci porovnání JménoMěsta = JménoSpolečnosti SQL Server prostě odmítne vykonat. Uvedené chování můžeme potlačit pouze explicitním vyvoláním konverzní funkce, tedy zápisem porovnání JménoMěsta = CONVERT(char(30), JménoSpolečnosti); obecně je ale dobré si porovnání polí definovaných nad různými doménami opravdu důkladně rozmyslet, protože často ani ze sémantického hlediska nemají smysl.

Uživatelsky definované datové typy můžeme vytvořit buďto pomocí nástroje SQL Server Enterprise Manager, nebo prostřednictvím systémové uložené procedury sp\_addtype. Při deklaraci UDDT se v každém z těchto dvou případů zadává jméno neboli datový typ a příznak, jestli typ akceptuje hodnoty Null. Po vytvoření uživatelsky definovaného datového typu můžeme k němu dále definovat implicitní hodnoty a validační pravidla. Pod pojmem *pravidlo* se přitom v SQL Serveru rozumí jistý logický výraz, který popisuje hodnoty přípustné pro daný uživatelsky definovaný datový typ (respektive pro pole, jestliže pravidlo není svázáno s UDDT, ale s polem). *Implicitní hodnota* pak není nic jiného než jistá „standardní“ hodnota, kterou systém vloží do pole v případě, že uživatel nestanovil hodnotu explicitně a pole by tím pádem muselo zůstat prázdné.

Svázání pravidla nebo implicitní hodnoty s uživatelsky definovaným datovým typem probíhá ve dvou po sobě jdoucích krocích. Nejprve musíme vytvořit pravidlo či implicitní hodnotu, a poté je svážat s uživatelsky definovaným datovým typem (respektive s polem). I v tomto případě můžeme postup ospravedlnit oblíbenou větou, „to není chyba, to je vlastnost systému“: jednou definované pravidlo či implicitní hodnota se totiž posléze dá využít i kdekoli jinde. Mně osobně ale tento po-

stup nepřípadá bůhvíjak pohodlný a podle mých vlastních zkušeností se tyto objekty opětovně využívají opravdu jen zřídka. Při definici tabulky můžeme totiž v SQL Serveru deklarovat implicitní hodnoty a omezení typu CHECK přímo jako součást definice tabulky. (Omezení typu CHECK mají podobou funkci jako pravidla, jsou ale silnější.) Jestliže ale deklarujeme uživatelsky definovaný datový typ, žádný takový postup o jednom kroku bohužel použít nemůžeme; zde nám nezbývá než zůstat u starší metodologie „nejprve vytvoř, potom svaž“. Zda Microsoft přidá do některé z budoucích verzí SQL Serveru možnost deklarace omezení typu CHECK přímo s novým uživatelsky definovaným datovým typem, v to můžeme jen ve skrytu duše tajně doufat.

Druhý způsob implementace datové integrity, i když jaksi odložené, spočívá ve vyhledávací tabulce. Tento postup můžeme použít jak u databázového stroje Microsoft Jet, tak i v SQL Serveru. Jako příklad uvažujme doménu StátyUSA. Teoreticky bychom mohli vytvořit pravidlo, do něhož bychom ručně zapsali všechny paděsát států Unie. Prakticky vzato by ale takový postup byl cistí krátkolinný, a to zejména v databázovém stroji Jet, kde bychom museli pro každé pole deklarované s touto doménou zapisovat pravidlo znovu. Je tedy mnohem, mnohem jednodušší vytvořit speciální vyhledávací tabulkou StátyUSA a zformulovat novou podmíinku, že hodnoty daného pole mají být omezeny jen na hodnoty uložené v této tabulce; tuto podmíinku pak zajistíme pomocí mechanismů referenční integrity.

## Entitová integrity

Entitová omezení mohou v databázovém schématu kontrolovat jednotlivá pole, několik polí, anebo tabulku jako celek. Jisté mechanismy pro zajištění integrity na úrovni entity přitom nabízí jak databázový stroj Microsoft Jet, tak i SQL Server. Není žádným překvapením, že i zde poskytuje SQL Server bohatší funkce; tentokrát ale rozdíl není až tak propastný, jak by leckdo možná očekával.

Na úrovni jednotlivých polí představuje nejzákladnější omezení integrity samozřejmě definovaný datový typ. Databázový stroj Microsoft Jet i SQL Server mají k dispozici široké spektrum datových typů, které shrnuje následující obsáhlá tabulka:

Logický datový typ Datový typ	SQL Serveru Datový typ	Stroje Microsoft Jet	Interval hodnot	Velikost pro uložení
Celé číslo	Int	Dlouhé celé číslo	Celá čísla od -2 147 483 648 do 2 147 483 647	4 bajty
	Smallint	(není)	Celá čísla od -32 768 do 32 767	2 bajty
	Tinyint	Celé číslo	Celá čísla od 0 do 255	1 bajt
Pakovaný dekadický (přesný číselný)	Decimal	Číslo (různé typy)	Celá čísla a zlomky od $-10^{38}$ do $10^{38}$	2-17 bajtů
Číslo s pohyblivou řádovou čárkou (přibližný číselný typ)	Float (přesnost 15 číslic)	Dvojitá přesnost	Přibližná reprezentace čísel od $-1,79 \cdot 10^{-308}$ do $1,79 \cdot 10^{308}$	8 bajtů
Interval kladných hodnot: od $2,23 \cdot 10^{-308}$ do $1,79 \cdot 10^{308}$				

Logický datový typ Datový typ	SQL Serveru Datový typ	Stroje Microsoft Jet	Interval hodnot	Velikost pro uložení
			Interval záporných hodnot: od -2,23E <sup>108</sup> do -1,79E <sup>108</sup>	
- Real		Jednoduchá přesnost	Přibližná reprezentace čísel od -3,40E <sup>18</sup> do 3,40E <sup>18</sup>	4 bajty
			Interval kladných hodnot: od 1,18E <sup>18</sup> do 3,40E <sup>18</sup>	
			Interval záporných hodnot: od -1,18E <sup>18</sup> do -3,40E <sup>18</sup>	
Žnakový (s pevnou délkou)	Char	(není)	V databázovém stroji Jet maximálně 255 znaků, v SQL Serveru 7.0 maximálně 8000 znaků (ve starších verzích 255 znaků)	1 bajt na každý deklarovaný znak
Žnakový (s proměnnou délkou)	Varchar	Text	V databázovém stroji Jet maximálně 255 znaků, v SQL Serveru 7.0 maximálně 8000 znaků (ve starších verzích 255 znaků)	1 bajt na každý uložený znak
Peněžní údaj	Money	Měna	Čísla s přesným vyjádřením na čtyři desetinná místa, v intervalu od -922 337 208 685 477,5808 do 922 337 208 685 477,5807	8 bajtů
	Smallmoney	(není)	Čísla s přesným vyjádřením na čtyři desetinná místa, v intervalu od -214 748,3648 do 214 748,3647	4 bajty
Datum a čas	Datetime	Datum/čas	V SQL Serveru datum od 1. ledna 1753 do 31. prosince 9999, v databázovém stroji Jet od 1. ledna 100 do 31. prosince 9999	8 bajtů
	Smalldatetime	(není)	Od 1. ledna 1900 do 6. června 2079	4 bajty
Binární (s pevnou délkou)	Binary	(není)	Maximálně 8000 bajtů	Počet deklarovaných bajtů plus 4 bajty
Binární (s proměnnou délkou)	Varbinary	(Podporováno pouze u propojených tabulek)	Maximálně 8000 bajtů	Počet skutečně uložených bajtů plus 4 bajty
Dlouhý textový údaj	Text	Memo	Znakové údaje o délce až 2 GB v SQL Serveru, respektive do 1 GB v databázi Microsoft Jet	Množství skutečně uložených dat plus 16 bajtů
Velký binární objekt (Binary Large Object, BLOB)	Image	Objekt OLE	Binární data o délce až 2 GB v SQL Serveru, respektive do 1 GB v databázi Microsoft Jet	Množství skutečně uložených dat plus 16 bajtů
Booleovská (logická) hodnota	Bit	Ano/Ne	0 nebo 1	1 bajt, v SQL Serveru se ale bitové sloupce v tabulce sloučují, takže 8 a méně sloupců tohoto typu zabírá v SQL Serveru celkem 1 bajt

V SQL Serveru, jak jsme si před chvílí řekli, můžeme přitom deklarovat pole také nad uživatelsky definovaným datovým typem. Tako deklarované pole zdědí od datového typu jeho nadefinované povolení hodnot Null, implicitní hodnoty a pravidla; ve vlastní definici pole můžeme ale všechny tyto vlastnosti předefinovat. Logicky vzato by se omezení z uživatelsky definovaného datového typu měla v definici pole jen dále zúžit; SQL Server ale ve skutečnosti v takovém případě nahradí jednoduše v popisu pole původní definici UDDT za novou podobu. Je tedy možné v definici pole zadat povolení hodnot Null, přestože původní uživatelsky definovaný datový typ, s nímž je pole deklarováno, hodnoty Null zakazuje.

Povolení hodnot Null u daného pole můžeme snadno určit jak v SQL Serveru, tak i v databázovém stroji Jet. Při definici sloupce v SQL Serveru stačí zapsat klíčové slovo NULL nebo NOT NULL, případně zatrhnout odpovídající zaškrťávací políčko v SQL Server Enterprise Manageru.

U databázového stroje Jet je ekvivalentem příznaku Null pole Je nutno zadat. Kromě toho databázový stroj Jet definuje příznak Povolit nulovou délku, který určuje, jestli dané pole typu Text nebo Memo povoluje také prázdné řetězce („“). V SQL Serveru se podobné omezení dá implementovat pomocí omezení typu CHECK.

Implicitní hodnotu nového pole nadefinujeme v databázovém stroji Jet jednoduše nastavením odpovídající vlastnosti. V SQL Serveru případně při vytváření pole vlastnost Default, případně k poli navážeme systémovou implicitní hodnotu; postup jsme si popsali u uživatelsky definovaných datových typů. Deklarovat implicitní hodnotu jako součást definice tabulky je ale rozhodně čistší a v případě, že implicitní hodnotu nedeklarujete (nebo nemůžete deklarovat) na úrovni domény, vám tento postup osobně vřele doporučuji.

A nakonec ještě databázový stroj Jet i SQL Server umožňují zavedení specifických omezení pro entity. Databázový stroj Jet nabízí ke každému poli dvě vlastnosti, Ověřovací Pravidlo a OvěřovacíText. V SQL Serveru je možné společně s definicí pole deklarovat omezení typu CHECK, případně s polem později svázat systémová pravidla. Vhodnější metodou jsou přitom omezení typu CHECK.

Validační pravidla databázového stroje Jet a omezení typu CHECK z SQL Serveru se na první pohled zdají být stejná, přesto jsou ale mezi nimi určité významné rozdíly. Oba mechanismy mají podobu logického výrazu a žádný z nich se nesmí odkazovat na jiné tabulky či sloupce. Validiční pravidlo v databázovém stroji Jet musí být ale u přípustné hodnoty rovno True, zatímco omezení CHECK v SQL Serveru se nesmí vyhodnotit na False. To je opravdu drobný, přesto však významný rozdíl: pro omezení typu CHECK je přijatelný výsledek True nebo Null, zatímco validační pravidlo „projde“ pouze s hodnotou True.

V SQL Serveru se navíc pro jediné pole dá definovat i několik omezení typu CHECK. Na jedno pole v SQL Serveru je možné přesněji řečeno aplikovat jedno pravidlo a libovolný počet omezení typu CHECK, zatímco u databázového stroje Jet má pole jedinou vlastnost Ověřovací Pravidlo. Hodnota vlastnosti OvěřovacíText představuje mimochodem v databázovém stroji Jet text, který se uživateli předá jako chybová zpráva. Microsoft Access zobrazí tento text v okně se zprávou; v Microsoft Visual Basicu a v dalších programových prostředích je pak k dispozici jako položka kolekce Errors.

Entitová omezení, která se odkazují na několik polí ze stejné tabulky, se v databázovém stroji Jet implementují jako tabulková validační pravidla, zatímco v SQL Serveru jako tabulková omezení typu CHECK. Tato omezení na úrovni tabulky fungují přesně stejným způsobem jako jejich protějšky definované na úrovni jednotlivého pole; jediný rozdíl spočívá v místě jejich deklarace.

Nejdůležitějším omezením entitové integrity je požadavek, podle něhož musí být možné jednoznačně identifikovat každou jednotlivou instanci entity. Uvědomte si, že toto je fakticky jediné *pravé* pravidlo entitové integrity; ostatním pravidlům se přesněji říká omezení integrity, platná na úrovni entity. Databázový stroj Jet a SQL Server podporují omezení jednoznačnosti do značné míry stejným způsobem, navenek je ale tato podpora zcela odlišná. Oba stroje implementují tato omezení s pomocí indexů, ty jsou však v SQL Serveru před uživatelem skryty. Jestli přitom máme explicitně vytvořit index (v databázovém stroji Jet), nebo nadeklarovat omezení (SQL Server), to je v podstatě jen technický detail.

Dalším důležitým rozdílem mezi omezeními jednoznačnosti a primárními klíči spočívá v tom, že jedinečné indexy mohou obsahovat hodnoty Null, zatímco primární klíče nikoli. Určité rozdíly mezi oběma databázovými stroji spočívají také právě ve způsobu ošetření hodnot Null v jedinečných indexech. Databázový stroj Jet tak nabízí vlastnost Ignorovat hodnoty Null, která zakazuje přidávat do indexu záznamy s hodnotami Null v indexovaných sloupcích. Tyto záznamy jsou tedy součástí tabulky, v indexu se však neobjeví. SQL Server nic podobného neumožňuje.

V SQL Serveru smí dále tabulka obsahovat pouze jeden záznam s hodnotou Null v indexu. Takové řešení nelze logicky podporovat, protože znamená, že záznamy s hodnotami Null se považují za stejné, a to samozřejmě neplatí. Hodnota Null není rovna ničemu jinému a není rovna ani jiné hodnotě Null.

Je zajímavé, že ani v databázovém stroji Jet, ani v SQL Serveru není definice primárního klíče v tabulce povinná; tabulka dokonce ani nemusí mít omezení jednoznačnosti. Jinými slovy, v obou těchto systémech můžeme klidně vytvořit tabulku, která podle našeho pojetí není relací; vektory hodnot v relaci musí být totiž možné jednoznačně identifikovat, zatímco u tabulky to nutně není. Proč bychom takovou tabulku měli chtít vytvořit, to mi poněkud uniká, přesto se ale domnívám, že je dobré o této možnosti vědět – člověk nikdy neví, kdy ji bude potřebovat.

SQL Server nabízí také procedurální mechanismus integrity na úrovni entity, který databázový stroj Jet nezná. Jsou to takzvané *spouště* (triggers), což jsou malé jednotky programového kódu (přesněji řečeno kód v jazyce Transact-SQL), které se vyvolávají automaticky při vzniku jisté definované události. Pro každou z těchto událostí – INSERT, UPDATE a DELETE – je možné definovat i několik spouští a podobně i jedna spoušť může sloužit několika událostem.

## Referenční integrita

Zatímco ve své podpoře entitové integrity se databázový stroj Jet a SQL Server v podstatě neliší, v oblasti referenční integrity využívají zcela odlišné přístupy. V SQL Serveru je tak možné definovat *omezení cizího (nevlastního) klíče* přímo jako součást definice tabulky. Omezení cizího klíče zavádí odkaz na kandidátní klíč jiné tabulky, která se zde nazývá primární tabulka. Po vytvoření tohoto odkazu již SQL Server odmítá veškeré operace vložení záznamů, kterým neodpovídá žádný záznam primární tabulky, a tím zabraňuje ve vytvoření sirotčích záznamů. Hodnoty Null nejsou ve sloupcích cizího klíče zakázané, i když pokud je sloupec součástí primárního klíče tabulky (velice

často je tomu právě tak), dá se jim zabránit. SQL Server zabraňuje také v odstranění záznamu primární tabulky, jemuž odpovídají nějaké hodnoty cizího klíče.

Databázový stroj Jet podporuje mechanismus referenční integrity v databázi prostřednictvím *objektu Relation*. Zde Microsoft zvolil poněkud nešťastnou terminologii – objekt Relation databázového stroje Jet je totiž fyzickou reprezentací vztahu mezi dvěma entitami. (Na rozdíl od autorky nemusíte v pojmu „relace“ spatřovat žádný rozpor – tento pojem se v relačních databázích používá zcela běžně. Pozn. překl.) Nezaměňujte proto objekt Relation s logickými relacemi definovanými na úrovni datového modelu.

Nejjednodušší způsob vytvoření objektů Relation spočívá v jejich návrhu přímo v uživatelském rozhraní Microsoft Accessu (pomocí příkazu Relace z nabídky Nástroje); dají se ale také vytvořit v programovém kódu. Dvě tabulky, které se účastní relace, definujeme u objektu Relation modelu Data Access Objects (DAO) ve vlastnostech Table a ForeignTable, zatímco kolekce Fields popisuje propojená pole obou tabulek.

Způsob, jakým bude databázový stroj udržovat u dané relace referenční integritu, charakterizuje její vlastnost Attributes. Tato vlastnost nabývá několika možných hodnot, které uvádí následující tabulka:

Konstanta vlastnosti Attributes	Popis
dbRelationUnique	Vztah je typu jedna k jedné.
dbRelationDontEnforce	Vztah se nijak nezajišťuje, není definována referenční integrita.
dbRelationInherited	Vztah existuje v jiné než aktuální databázi, která definuje obě propojené tabulky.
dbRelationUpdateCascade	Aktualizace se budou promítat v kaskádě.
dbRelationDeleteCascade	Odstranění se budou promítat v kaskádě.

Všimněte si atributových příznaků dbRelationUpdateCascade a dbRelationDeleteCascade. Jestliže je nastaven (zapnut) první, aktualizační příznak, provede databázový stroj Jet při každé změně odkažovaného pole automatickou aktualizaci všech odpovídajících záznamů v cizí tabulce. Podobně odstraňovací příznak znamená, že se z cizí tabulky automaticky odstraní odpovídající záznamy. SQL Server žádné srovnatelné automatické příznaky nemá, chování kaskádovitých operací je však možné poměrně snadno implementovat pomocí spouští.

## Ostatní typy integrity

Říkali jsme si, že v datovém modelu definujeme ještě tři další typy integrity: je to databázová integrita, přechodová integrita a transakční integrita. Z toho některá přechodová omezení jsou natolik jednoduchá, že se dají snadno deklarovat jako validační pravidla. Většina z nich, a kromě toho také všechna databázová a transakční omezení, se však musí implementovat procedurálně. V databázovém SQL Serveru to znamená napsat odpovídající spouští. A protože databázový stroj Jet spouště nepodporuje, musí se v něm uvedená omezení implementovat v uživatelské front-end části.

## Stručné shrnutí

V této kapitole jsme hovořili o modelování a implementaci datové integrity. Tři z popsaných typů omezení integrity – doménová, přechodová a entitová omezení – přitom kontrolují jednotlivé relace, zatímco omezení referenční integrity zabezpečují zachování jistých vztahů *mezi* relacemi. Posledním typem jsou databázová omezení a transakční omezení, která kontrolují podobu databáze jako celku.

Datová integrita se v databázovém schématu implementuje pomocí vhodné kombinace deklarativní a procedurální integrity. Deklarativní integritu deklarujeme explicitně jako součást schématu a pro implementaci je vhodnější. Ne všechna omezení se ale dají implementovat jako deklarativní integrita; v takovém případě nezbývá než použít mechanismy procedurální integrity.

V kapitole 5 budeme zkoumat relační algebru a operace, které se dají nad relacemi v databázi provádět.



# RELAČNÍ ALGEBRA

5

V předchozích kapitolách jsme hovořili o definici jistého speciálního typu relace, které se říká *základní relace* a která již dostane nějakou fyzickou reprezentaci v databázi. Relační model podporuje také vytvoření několika typů odvozených relací. Za *odvozenou relaci* považujeme přitom takovou relaci, která je definována na základě jiných relací, nikoli přímo na základě atributů. Relacemi, na něž se odvozená relace odkazuje, mohou být jak základní relace, tak i jiné odvozené relace, a to v libovolné kombinaci.

Základní relaci reprezentuje v databázovém schématu tabulka. Odvozenou tabulku tvoří v Microsoft SQL Serveru *pohledy* a u databázového stroje Jet *dotazy*. Já zde pro zjednodušení budu používat výraz „pohled“, protože to je standardní pojem z oblasti relačních databází. Obecným pojmem „množina záznamů“ budu rozumět pohled nebo dotaz.

Pohledy se definují s pomocí takzvaných relačních operátorů, kterým se věnuje právě tato kapitola. Microsoft Access, respektive SQL Server Enterprise Manager, nabízí pro definici pohledů speciální grafické rozhraní. Pohledy je možné definovat také pomocí příkazů SELECT jazyka SQL.

Zkratka SQL (která se často vyslovuje „síkwel“) znamená Structured Query Language, strukturovaný dotazovací jazyk. Jedná se o standardní jazyk určený pro vyjadřování relačních operací. Určitý svůj dialekt jazyka SQL podporuje jak databázový stroj Jet, tak i SQL Server. Samozřejmě že tento dialekt není u obou systémů stejný. To by bylo až příliš jednoduché. Rozdíly mezi oběma implementacemi však naštěstí příliš významně nezasahují do relační algebry, o níž tato kapitola pojednává. Já budu uvádět příklady z obou dialektů.

Příkaz SELECT jazyka SQL je velice silný nástroj a je také docela složitý. Jeho plný výklad je mimo rámec této knihy. Pro účely našeho výkladu jej zredukujeme na základní strukturu, která má následující syntaxi:

```
SELECT <SeznamPolí>
  FROM <SeznamMnožinZáznamů>
    <TypSpojení>JOIN <SpojovacíPodmínka>
  WHERE <VýběrováKritéria>
  GROUP BY <SeznamPolíKseskupení>
  HAVING <VýběrováKritéria>
  ORDER BY <SeznamPolíKseřazení>
```

Na prvním řádku této klausule SELECT vidíme <SeznamPolí>. Tento seznam obsahuje jedno nebo více polí, které budou součástí výsledné množiny záznamů. Pole mohou být buďto přímo obsažena v podkladových množinách záznamů, nebo mohou být vypočtena. <SeznamMnožinZáznamů> v klausuli FROM je, jak zajisté tušíte, seznam tabulek a pohledů, na nichž je příkaz SELECT

založen. Klausule SELECT a FROM jsou zároveň jedinými dvěma klausulemi, které jsou v příkazu povinné; všechny ostatní jsou nepovinné.

Klausule JOIN definuje vztah mezi množinami záznamů uvedenými v seznamu <SeznamPolí>. O operacích spojení tabulek budeme hovořit podrobněji dál v této kapitole. Klausule WHERE popisuje logický výraz, <Výběrová Kritéria>, který omezuje data zahrnutá ve výsledné množině záznamů. I o tomto mechanismu omezení výsledné množiny si povíme víc později.

Další klausule, GROUP BY, spojuje do jediného záznamu takové záznamy, které mají v polích ze zadaného seznamu stejné hodnoty. Klausule HAVING dále omezuje navrácená pole seskupená pomocí klausule GROUP BY. Poslední klausule ORDER BY definuje třídění množiny záznamů, a to podle polí uvedených v seznamu <SeznamPolíKSeřazení>.

## Hodnoty Null a třihodnotová logika (ještě jednou)

Většina operací relační algebry pracuje s *logickými operátory*. To jsou operátory, které obvykle vrací *booleovský výsledek*, čili hodnotu True nebo False. Říkám zde „obvykle“, protože díky hodnotám Null se nám celá situace v relačním modelu tak trochu komplikuje.

Hodnota Null představuje třetí hodnotu ke klasické dvouhodnotové množině booleovských hodnot; to znamená, že musíme pracovat se třemi hodnotami – True, False a Null. Není tedy žádným překvapením, že se souboru těchto hodnot a operátorů říká *třihodnotová logika*. Pravdivostní tabulky třihodnotové logiky pro standardní logické operátory uvádí tabulka 5-1.

AND	True	False	Null
True	True	False	Null
False	False	False	Null
Null	Null	Null	Null

OR	True	False	Null
True	True	True	Null
False	True	False	Null
Null	Null	Null	Null

XOR	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

Tabulka 5-1 Pravdivostní tabulky operátorů And, Or a XOR ve třihodnotové logice

Jak vidíte, hodnota výrazu Null op cokoliv je pro jakýkoliv logický operátor op rovna Null. To obecně platí i pro operátory logického porovnání, jak to vidíme v tabulce 5-2.

=	True	False	Null
True	True	False	Null
False	False	True	Null
Null	Null	Null	Null

≠	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

Tabulka 5-2 Pravdivostní tabulky operátorů rovno a nerovno ve třihodnotové logice

V SQL Serveru jsou z jistých důvodů, které snad alespoň pro jeho návrháře mají nějaký smysl, definovány dále jakási „rozšíření“ normálních logických operátorů. Jestliže je tak volba ANSI\_NULLS vypnuta, vyhodnotí se výraz Null = Null jako pravdivý (True), zatímco výraz Null = <hodnota>, kde <hodnota> je cokoli jiného než Null, se vyhodnotí jako nepravdivé (False). (Toto zvláštní chování souvisí zcela nepochybně s problémem povolení jen jediné hodnoty Null v jedinečných indexech typu UNIQUE.)

Pro práci s hodnotami Null nabízí SQL Server dva speciální unární operátory – IS NULL a IS NOT NULL. Tyto operátory pracují přesně v souladu s běžným očekáváním. Jejich pravdivostní tabulku uvádí tabulka 5-3; zápis <hodnota> zde opět označuje cokoli jiného než Null.

Operand	Is Null	Is Not Null
<hodnota>	False	True
True	False	True
False	False	True
Null	True	False

Tabulka 5-3 Pravdivostní tabulky funkcí IS NULL a IS NOT NULL

## Relační operátory

Náš výklad relační algebry zahájíme čtyřmi typy relačních operátorů: je to restrikce (omezení), projekce (promítání), spojení a dělení (oddělení). První dva pracují jen s jednou množinou záznamů, i když touto množinou záznamů může být samozřejmě i pohled založený na libovolném počtu jiných množin záznamů. Operátor spojení je z celého relačního modelu zřejmě nejdůležitější a de-

finuje způsob propojení dvou množin záznamů. Poslední operátor, dělení, se používá poměrně zřídka, představuje ale šikovnou možnost, jak zjistit, které záznamy z jedné množiny záznamů odpovídají všem záznamům ve druhé množině záznamů.

Všechny tyto operátory jsou implementovány v určité podobě příkazu SELECT jazyka SQL. Dají se mezi sebou kombinovat libovolným způsobem, pochopitelně pokud dodržíme systémová omezení například ohledně maximální délky a složitosti příkazu.

## Restrikce

Operátor restrikce (omezení množiny záznamů) vrátí jen ty záznamy, které vyhovují stanovenému výběrovému kritériu. V příkazu SELECT jej implementuje klausule WHERE, kterou ukazuje následující příklad:

```
SELECT * FROM Zaměstnanci WHERE Příjmení = "Davolio";
```

V databázi Northwind tento příkaz vrátí záznam zaměstnankyně Nancy Davoliové, protože ona je jedinou osobu, která má v tabulce příjmení shodné se zadáným řetězcem. (Hvězdička v sekci, kde by se měl nacházet <SeznamPolí>, je speciální zkratkou a zastupuje výraz „všechna pole“.)

Výběrová kritéria, specifikovaná v klausuli WHERE, mohou být libovolně složitá. Jednotlivé logické výrazy spojujeme pomocí operátorů AND a OR. Tyto výrazy se posléze vyhodnocují pro každý jednotlivý záznam v původní množině záznamů a pokud celý výraz vrátí pro daný záznam hodnotu True, stane se tento záznam součástí výsledné množiny záznamů. Jestliže výraz vrátí u záznamu hodnotu False nebo Null, pak se záznam do množiny záznamů nepromítá.

## Projekce

Zatímco restrikce neboli omezení vyřizne z výsledné množiny záznamů jistý „horizontální“ výřez, projekce neboli promítání představuje vertikální řez – z původní množiny záznamů vrací pouze vybranou podmnožinu polí.

Jazyk SQL vyjadřuje tuto jednoduchou operaci v sekci <SeznamPolí> příkazu SELECT a do výsledků zahrne jen ta pole, která v této sekci uvedeme. Pomocí následujícího příkazu můžeme například vytvořit telefonní seznam zaměstnanců:

```
SELECT Příjmení, Jméno, Linka
FROM Zaměstnanci
ORDER BY Příjmení, Jméno;
```

Vzpomeňte si, že klausule ORDER BY provádí uspořádání dat; v tomto případě se tedy výsledky operace setřídí abecedně podle pole Příjmení a následně podle pole Jméno.

## Spojení

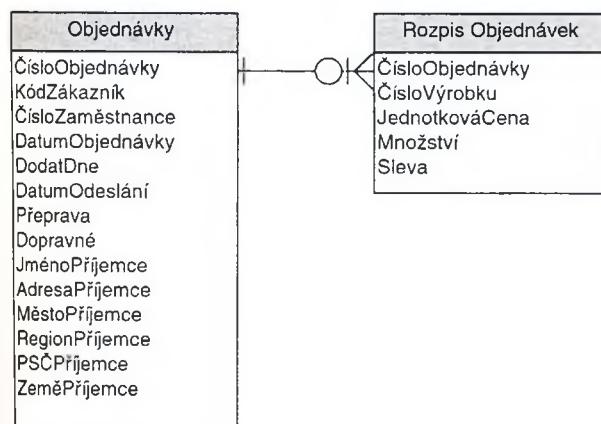
Operace spojení (v databázi též spojení tabulek, join) jsou zřejmě nejběžnějším případem relačních operací. Pro relační model jsou zajisté přímo životně důležité – pokud by se data v relacích nemohla podle potřeby zpětně sestavit, nemohli bychom je ani dekomponovat. A přesně toto provádí operátor spojení: propojí množiny záznamů na základě porovnání jednoho nebo více polí.

Operace spojení se v příkazu SELECT implementují pomocí klausule JOIN. Podle typu porovnání hodnot svázaných polí a podle způsobu zpracování výsledků tohoto porovnání je můžeme rozdělit do několika kategorií, na které se nyní společně podíváme.

### **Spojení na rovnost**

Jestliže se porovnání v operaci spojení provádí na základě operátora rovnosti, nazýváme je *spojení na rovnost* (equi-join). Z této operace se vrátí pouze ty záznamy, které mají v zadaných polích vzájemně odpovídající hodnoty.

Vezměme například relace uvedené na obrázku 5-4. To je typický příklad propojení tabulek, které jsou výsledkem procesu normalizace dat. Primárním klíčem tabulky Objednávky a cizím (nevlastním) klíčem tabulky RozpisObjednávek je zde pole ČísloObjednávky.



Obrázek 5-4 Tyto tabulky se dají opětovně spojit pomocí operátora JOIN

Opětovné spojení (a následnou denormalizaci) těchto dvou tabulek provede následující příkaz SELECT:

```

SELECT Objednávky.ČísloObjednávky, Objednávky.KódZákazníka,
       [Rozpis Objednávek].ČísloVýrobku
  FROM Objednávky
 INNER JOIN [Rozpis Objednávek]
    ON Objednávky.ČísloObjednávky = [RozpisObjednávek].ČísloObjednávky
 WHERE (((Objednávky.ČísloObjednávky)=10248));
  
```

Výsledkem tohoto příkazu je množina záznamů, kterou vidíme na obrázku 5-5.

KódZákazníka	Číslo objednávky	Číslo výrobku
VINET	10248	11
VINET	10248	8
VINET	10248	14

Obrázek 5-5 Tato množina záznamů je výsledkem spojení tabulek Objednávky a RozpisObjednávek

**Poznámka**

Jestliže tento dotaz spusťte v aplikaci Microsoft Access 2000 nad databází Northwind, nezobrazí se ve výsledné množině KódZákazníka, nýbrž jméno zákazníka. V Accessu mohou totiž pole zobrazovat něco jiného, než co je v tabulce skutečně uloženo; stačí při definici tabulky nadeklarovat vyhledávací ovládací prvek. To je velká výhoda pro interaktivní práci s Microsoft Accessem, na naše příklady to ale působí jako studená sprcha.

**Přirozená spojení**

Speciálním případem spojení na rovnost jsou takzvaná *přirozená spojení*. Operace přirozeného spojení musí vyhovět následujícím podmínkám:

- Operátorem porovnání v operaci spojení musí být rovnost.
- Spojení se musí účastnit všechna společná pole obou tabulek.
- Do výsledné množiny záznamů se smí zapsat pouze jedna množina společných polí.

Na přirozených spojeních není v podstatě nic výjimečného. Nechovají se žádným speciálním způsobem a databázový stroj je nepodporuje žádným zvláštním způsobem. Jedná se pouze o nejobvyklejší formu spojení; jsou tedy natolik častá, že se pro ně vymyslelo speciální označení.

Databázový stroj Jet však při vytvoření přirozeného spojení, které vyhovuje jistým speciálním podmínkám, přece jen dělá něco zvláštního. Jestliže je mezi dvěma tabulkami definován vztah typu jedna k více a společná pole, zahrnutá do výsledného pohledu, pocházejí z tabulky na straně „více“ příslušného vztahu, pak databázový stroj provede něco, čemu se říká *úprava řádku* (Row Fix-Up) nebo *automatické vyhledávání* (AutoLookup). Jakmile pak přejdeme kurzorem do polí použitých ve spojovacím kritériu, nabídne databázový stroj Jet automaticky seznam společných polí – docela příjemný trik, který programátorovi usnadní život.

**Theta-spojení**

Technicky vzato jsou theta-spojeními všechna spojení, jestliže je ale operátorem porovnání v operaci spojení rovnost, pak se toto spojení podle jistých běžných zvyklostí nazývá spojení na rovnost, nebo jen jednoduše spojení. Operace spojení, založená na libovolném jiném operátoru porovnání (tedy  $<>$ ,  $>$ ,  $>=$ ,  $<$ ,  $<=$ ), je pak *theta-spojení*.

Theta-spojení jsou v praxi velice vzácná, jsou ale vhodná pro řešení jistých typů problémů. Tyto problémy, které se řeší pomocí theta-spojení, souvisí většinou s vyhledáním záznamů, jejichž hodnota je například větší než průměr či součet hodnot, nebo záznamů, které v určité hodnotě spadají do zadlého intervalu.

Uvažujme například, že vytvoříme dva pohledy, z nichž první obsahuje průměrný počet prodaných jednotek za každou kategorii výrobků, a druhý popisuje celkový počet jednotek prodaných od každého jednotlivého výrobku. Situaci vidíme na obrázku 5-6. Jak tyto dva pohledy vytvořit, to si povíme v této kapitole o něco později; prozatím je prostě berme za hotové.

#### PrůměryVýrobkůVKategoriích

Název kategorie	Průměrně prodáno
Nápoje	678
Koření	442
Cukrovinky	608
Mléčné výrobky	915
Obilné výrobky	652
Maso / Drůbež	700
Plodiny	598
Mořské produkty	640

#### SoučtyVýrobků

Číslo výrobku	Název výrobku	Prodáno celkem
1	Chai	828
2	Chang	1057
3	Aniseed Syrup	328
4	Chef Anton's Cajun Seasoning	453
5	Chef Anton's Gumbo Mix	298
6	Grandma's Boysenberry Spread	301
7	Uncle Bob's Organic Dried Pears	763
8	Northwoods Cranberry Sauce	372
9	Mishi Kobe Niku	95
10	Ikura	742
11	Queso Cabrales	706
12	Queso Manchego La Pastora	344
13	Konbu	891
14	Tofu	404

Obrázek 5-6 Tyto pohledy se dají propojit s pomocí theta-spojení

Následující příkaz SELECT, který je postaven na operátoru porovnání  $>$ , sestaví v rámci každé kategorie seznam výrobků s nejvyšším objemem prodeje:

```
SELECT DISTINCTROW PrůměryVýrobkůVKategoriích.NázevKategorie,
    SoučtyVýrobků.NázevVýrobku
FROM PrůměryVýrobkůVKategoriích
INNER JOIN SoučtyVýrobků
ON PrůměryVýrobkůVKategoriích.ČísloKategorie = SoučtyVýrobků.ČísloKategorie
AND SoučtyVýrobků.ProdánoCelkem > [PrůměryVýrobkůVKategoriích].[PrůměrněProdáno];
```

Výsledky ukazuje obrázek 5-7.

Kategorie	Název výrobku
Koření	Chef Anton's Cajun Seasoning
Koření	Louisiana Fiery Hot Pepper Sauce
Koření	Vegie-spread
Koření	Siroop d'éitable
Koření	Gula Malacca
Koření	Grandma's Boysenberry Spread
Koření	Original Frankfurter grüne Soße
Nápoje	Chartreuse verte
Nápoje	Chang
Nápoje	Guaraná Fantástica
Nápoje	Chai
Nápoje	Côte de Blaye
Nápoje	Outback Lager
Nápoje	Rhönbräu Klosterbier
Nápoje	Lakkalikööri
Nápoje	Steeleye Stout

Obrázek 5-7 Tato množina záznamů je výsledkem theta-spojení

V tomto případě bychom pohled mohli definovat také pomocí restrikce s klausulí WHERE. Po opuštění pohledu SQL přitom Microsoft Access ve skutečnosti celý dotaz přepíše do následující podoby:

```
SELECT DISTINCTROW PrůměryVýrobkůVKategoriích.NázevKategorie,
    SoučtyVýrobků.NázevVýrobku
FROM PrůměryVýrobkůVKategoriích
INNER JOIN SoučtyVýrobků
ON PrůměryVýrobkůVKategoriích.ČísloKategorie = SoučtyVýrobků.ČísloKategorie
WHERE (((SoučtyVýrobků.ProdánoCelkem) > [PrůměryVýrobkůVKategoriích].[PrůměrněProdáno]));
```

Technicky vzato se přitom pomocí restrikce dají vyjádřit všechna spojení, tedy jak spojení na rovnost, tak i theta-spojení. (V databázové terminologii říkáme, že theta-spojení není nedělitelným nebo *atomickým operátorem*.) V případě theta-spojení je tato formulace navíc téměř vždy vhodnější, protože databázový stroj dokáže provádění takto sestaveného dotazu lépe optimalizovat.

## Vnější spojení

Všechna spojení, která jsme si zatím ukazovali, byla takzvaná *vnitřní spojení* (inner join); to znamená, že vracela jen ty záznamy, pro něž se spojovací podmínka vyhodnotí jako pravdivá (True). Všimněte si, že to není úplně přesně totéž jako navracení jen těch záznamů, u nichž si určená pole vzájemně odpovídají (přestože obvykle se vnitřní spojení popisuje právě takto). „Odpovídají si“ totiž znamená rovnost, a jak již víme, ne všechny operace spojení jsou založeny na rovnosti.

Relační algebra podporuje ještě další typ spojení, a sice vnější spojení (outer join). Takové vnější spojení vrátí všechny záznamy, které by vrátilo odpovídající vnitřní spojení, plus všechny záznamy z jedné nebo obou výchozích množin záznamů. Namísto chybějících (tedy „vzájemně neodpovídajících“) hodnot se vypíší hodnoty Null.

Vnější spojení se dále rozlišují jako levé, pravé a plné vnější spojení; konkrétní typ závisí na tom, ze které množiny záznamů se mají dosazovat ony záznamy „navíc“. A teď poslouchejte: já jsem se ve škole učila, že *levé vnější spojení* vrací zásadně všechny záznamy z množiny záznamů uvedené

na straně „jedna“ vztahu jedna k více, zatímco *pravé vnější spojení* vrací všechny záznamy ze strany „více“. U databázového stroje Jet i u SQL Serveru se však „směr“ vnějšího spojení rozlišuje podle pořadí, v jakém jsou výchozí množiny záznamů uvedeny v příkazu SELECT. Oba následující příkazy vrátí tudíž z množiny záznamů X všechny záznamy a z množiny záznamů Y jen ty z nich, pro které se *<podmínka>* vyhodnotí jako pravdivá (True):

```
SELECT * FROM X LEFT OUTER JOIN Y ON <podmínka>
SELECT * FROM Y RIGHT OUTER JOIN X ON <podmínka>
```

*Plné vnější spojení* (full outer join) pak vrátí všechny záznamy z obou zadaných množin záznamů, přičemž vzájemně spojí ty záznamy, u nichž se zadaná *<podmínka>* vyhodnotí jako pravdivá. SQL Server podporuje plná vnější spojení v podmínce FULL OUTER JOIN:

```
SELECT * FROM X FULL OUTER JOIN Y ON <podmínka>
```

Databázový stroj Jet plná vnější spojení přímo nepodporuje, můžeme je ale nahradit sjednocením levého vnějšího spojení a pravého vnějšího spojení. O operacích sjednocení si povíme v další části výkladu.

## Dělení

Poslední relační operací je dělení. Operátor *relačního dělení* (pro rozlišení od běžného matematického dělení) vrátí všechny záznamy z jedné množiny záznamů, jejichž hodnoty se shodují se všemi odpovídajícími hodnotami ve druhé množině záznamů. Máme-li například množinu záznamů, která zobrazuje kategorie výrobků dodávaných jednotlivými dodavateli, bude výsledkem relačního dělení seznam dodavatelů, kteří dodávají výrobky ve všech kategoriích.

Tato situace není nikterak neobvyklá, její řešení ale není triviální, protože příkaz SELECT jazyka SQL operaci relačního dělení přímo nepodporuje. Stejněho výsledku jako s relačním dělením můžeme dosáhnout pomocí řady různých postupů. Nejsnazší způsob spočívá v přeformulování původního požadavku.

Protože dotaz „vypiš všechny dodavatele, kteří dodávají výrobky ve všech kategoriích“, se zpracovává obtížně, zkusíme namísto něj vytvořit jiný dotaz: „vypiš všechny dodavatele, u nichž se počet dodávaných kategorií výrobků rovná počtu všech kategorií výrobků“. To je příklad rozšířených operací, o kterých si ještě v této kapitole povíme více. Vždy se ale tato metoda použít nedá; v takovém případě můžeme relační dělení implementovat pomocí souvztažných poddotazů. Souvztažné dotazy jsou již ale mimo rámec této knihy. Proto budete-li je někdy potřebovat, podívejte se prosím do literatury uvedené v seznamu na konci knížky.

## Množinové operátory

Další čtyři operátory relační algebry vycházejí z klasické teorie množin. Byly však z pochopitelných důvodů mírně upraveny – my totiž pracujeme s relacemi, nikoli s nerozlišenými množinami.

## Sjednocení

*Relační sjednocení* je v podstatě „zřetelením“ dvou množin záznamů, čili jejich spojením za sebou. Dá se říci, že je to víceméně jakási relační verze sečítání. Výsledek sjednocení množiny záznamů A a množiny záznamů B je stejný, jako kdybychom všechny záznamy z množiny A přidali do množiny záznamů B.

Jako příklad uvažujme, že pro nějaké hromadné rozesílání pošty potřebujeme seznam všech jmen a adres známých v naší databázi. V databázi Northwind jsou adresy obsaženy ve dvou množinách záznamů, a sice v tabulkách Zákazníci a Zaměstnanci; obojí adresy můžeme tedy snadno sloučit pomocí operace sjednocení. V tomto případě budeme potřebovat následující příkaz UNION:

```
SELECT Firma AS PlnéJméno, Adresa, Město, PSČ
FROM Zákazníci
UNION SELECT [Jméno] & " " & [Příjmení] AS PlnéJméno,
             Adresa, Město, PSČ FROM Zaměstnanci
ORDER BY PlnéJméno;
```

Všimněte si, že pole Firma jsme zde přejmenovali na „PlnéJméno“ a pole Jméno a Příjmení z tabulky Zaměstnanci jsme spojili pomocí operace zřetězení. Výsledné pole se také jmenuje „PlnéJméno“. To ovšem neznamená, že by se u dotazu se sjednocením musela všechna pole v <SeznamuPolí> každého jednotlivého příkazu SELECT jmenovat stejně; musí jich být ale stejný počet a musí mít stejné (nebo alespoň vzájemně kompatibilní) typy. Výsledky výše uvedeného příkazu v Microsoft Accessu vidíme na obrázku 5-8.

PlnéJméno	Adresa	Město	PSČ
Alfreds Futterkiste	Obere Str. 57	Berlín	12209
Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	Mexico D.F.	05021
Andrew Fuller	908 W. Capital Way	Tacoma	98401
Anne Dodsworth	7 Houndstooth Rd.	Londýn	WG2 7LT
Antonio Moreno Taquería	Mataderos 2312	Mexico D.F.	05023
Around the Horn	120 Hanover Sq	Londýn	WA1 1DP
Berglunds snabbköp	Berguvsvägen 8	Luleå	S-958 22
Blauer See Delikatessen	Forsterstr. 57	Mannheim	68306
Blondel père et fils	24 , place Kléber	Štrasburg	67000
Bólido Comidas preparadas	C/ Araquil, 67	Madrid	28023
Bon app'	12, rue des Bouchers	Marseille	13008
Bottom-Dollar Markets	23 Tsawassen Blvd.	Tsawassen	T2F 8M4
B's Beverages	Fauntleroy Circus	Londýn	EC2 5NT

Obrázek 5-8 Příkaz UNION sloučuje záznamy z obou určených tabulek

## Průnik

Operátor průniku vrátí ty záznamy, které jsou pro obě původní množiny záznamů společné. Jedná se v podstatě o operaci „najdi duplicity“ – přesně tak se také nejčastěji používá. Průnik se implementuje pomocí operace vnějšího spojení.

Jako příklad uvažujme seznamy klientů společnosti, které jsme zdědili z několika různých starších systémů; tuto situaci znázorňuje obrázek 5-9.

Následující příkaz SELECT vrátí potřebné duplicitní záznamy:

```
SELECT DuplicitníZákazníci1.*
FROM DuplicitníZákazníci1
LEFT JOIN DuplicitníZákazníci2
ON (DuplicitníZákazníci1.KódZákazníka = DuplicitníZákazníci2.KódZákazníka)
AND (DuplicitníZákazníci1.Firma = DuplicitníZákazníci2.Firma)
WHERE (((DuplicitníZákazníci2.KódZákazníka) IS NOT NULL));
```

Výsledky tohoto příkazu vidíme na obrázku 5-10.

DuplicitníZákazníci1

Kód zákazníka	Firma
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel pere et fils
BOLID	Bólido Comidas preparadas

DuplicitníZákazníci2

Kód zákazníka	Firma
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
FAMIA	Familia Arquibaldo
FISSA	FISSA Fábrica Inter Salchichas S.A.
FOLIG	Folies gourmandes
FOLKO	Folk och få HB
FRANK	Frankenversand

Obrázek 5-9 Tabulky ze starších systémů obsahují často duplicitní údaje

Kód zákazníka	Firma
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn

Obrázek 5-10 Vnější spojení, kombinované s operátorem IS NOT NULL, provádí operaci průniku

## Rozdíl

Zatímco operace průniku dvou množin záznamů provádí příkaz „najdi duplicity“, *operátor rozdílu* zajistuje „hledání sirotků“. Relační rozdíl dvou množin záznamů tvoří záznamy, které náleží jen do jedné množiny záznamů, do druhé však nikoli.

Jako příklad budeme uvažovat stejně dvě množiny záznamů jako na obrázku 5-9; tentokrát ale v příkazu SELECT vrátíme všechny *neshodné* záznamy:

```
SELECT DuplicitníZákazníci1.*
FROM DuplicitníZákazníci1
LEFT JOIN DuplicitníZákazníci2
ON (DuplicitníZákazníci1.KódZákazníka = DuplicitníZákazníci2.KódZákazníka)
AND (DuplicitníZákazníci1.Firma = DuplicitníZákazníci2.Firma)
WHERE (DuplicitníZákazníci2.KódZákazníka IS NULL);
```

Operace vnějšího spojení v tomto příkazu vrátí všechny záznamy ze dvou seznamů. Jak si zajisté vzpomínáte, doplní vnější spojení do polí, které nemají ve druhé tabulce odpovídající záznam, hodnoty Null. Klausule WHERE musí omezit navrácené záznamy jen na tyto záznamy bez odpovídajících hodnot, a proto obsahuje operátor IS NULL.

Pokud se vám to nyní zdá nějaké složité, zkuste celou operaci provést ve dvou na sebe navazujících krocích. Nejprve vytvořte vnější spojení, a to ve formě pohledu, a poté takto vytvořený pohled omezte pomocí klausule WHERE. Celý postup ukazuje obrázek 5-11.

Krok 1: vytvoření vnějšího spojení

Kód zákazníka	Firma
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel père et fils
BOLID	Bólido Comidas preparadas

```
SELECT DuplicitníZákazníci1.*  
FROM DuplicitníZákazníci1  
LEFT JOIN DuplicitníZákazníci2  
ON (DuplicitníZákazníci1.KódZákazníka = DuplicitníZákazníci2.KódZákazníka);
```

Krok 2: omezení výsledků na ty záznamy, kde je KódZákazníka rovno Null

Kód zákazníka	Firma
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel père et fils
BOLID	Bólido Comidas preparadas

```
SELECT DuplicitníZákazníci1.*  
FROM DuplicitníZákazníci1  
LEFT JOIN DuplicitníZákazníci2  
ON (DuplicitníZákazníci1.KódZákazníka = DuplicitníZákazníci2.KódZákazníka)  
WHERE (DuplicitníZákazníci2.KódZákazníka IS NULL);
```

Obrázek 5-11 Operace relačního rozdílu se dá vykonat ve dvou krocích

## Kartézský součin

Posledním množinovým operátorem je kartézský součin. Tato operace funguje podobně jako její protějšek v teorii množin; to znamená, že kartézský součin dvou množin záznamů zkombinuje každý jednotlivý záznam první množiny záznamů s každým jednotlivým záznamem druhé množiny záznamů.

Kartézský součin (neboli stručně jen „součin“) dvou množin záznamů vrací příkaz SELECT, v němž klausule JOIN chybí. Následující příkaz tak vrátí všechny možné kombinace zákazníků a reprezentantů služby zákazníkům (RSZ):

```
SELECT JménoZákazníka, JménoRSZ FROM Zákazníci, RSZ
```

Kartézský součin je v některých případech vhodný pro analytické účely, případně slouží pro generování různých dočasných výsledků, s nimiž se dále manipuluje. Nejčastěji jsou ale důsledkem omylu. To takhle jednoduše v Návrháři dotazů Microsoft Accessu zapomenete natáhnout čáru spojení tabulek a hurá, kartézský součin je na světě. Je to opravdu úžasné snadné, takže se nelekňete, když se to napoprvé (nebo desetkrát napoprvé) čirou náhodou stane i vám.

## Speciální relační operátory

Od okamžiku prvotní formulace relačního modelu byla navržena řada různých rozšíření relační algebry. My se zde společně podíváme na ta z nich, která jsou již obecně uznávaná: jsou to souhrnné operace, rozšíření a přejmenování. Dále si řekneme něco o třech rozšířených, která ve svých produktech nabízí Microsoft: je to transformace, součtové hodnoty a datová krychle.

### Souhrnné operace

Souhrnný operátor provádí přesně to, co od něj nejspíše budeme očekávat: vytváří záznamy se souhrnnými údaji, seskupené podle určitých polí. Tato operace je velice užitečná v řadě běžných situací, kdy potřebujeme zkoumat data na vyšší úrovni abstrakce, než s jakou jsou uložena v databázi.

Implementaci souhrnné operace představuje v příkazu SELECT klausule GROUP BY. Z tohoto příkazu se vrátí jeden záznam pro každou různou hodnotu v zadaném poli nebo polích. Jestliže do klausule zadáme více než jedno pole, vytvoří se vnořené skupiny. Uvažujme například takovýto příkaz:

```
SELECT Kategorie.NázevKategorie, Výrobky.NázevVýroku,
       SUM([RozpisObjednávek].Množství) AS SoučetMnožství
  FROM (Kategorie INNER JOIN Výrobky ON Kategorie.ČísloKategorie =
        Výrobky.ČísloKategorie)
    INNER JOIN [RozpisObjednávek]
      ON Výrobky.ČísloVýroku = [RozpisObjednávek].ČísloVýroku
 GROUP BY Kategorie.NázevKategorie, Výrobky.NázevVýroku;
```

Tento příkaz vrátí jeden záznam pro každý výrobek obsažený v databázi Northwind; výsledky budou seskupeny podle kategorií výrobků a budou obsahovat tři pole: NázevKategorie, NázevVýroku a SoučetMnožství – to je celkový počet prodaného množství každého jednotlivého výrobku. Celý výsledek vidíme na obrázku 5-12.

Všechna pole, uvedená v části <SeznamPolí> příkazu SELECT, se u klausule GROUP BY musí buďto promítnout do <SeznamuPolíKSeskupení>, nebo musí tvořit argument některé z agregačních funkcí jazyka SQL. *Agregační funkce jazyka SQL* vypočítávají přitom souhrnné údaje pro výsledné záznamy. Nejčastěji se používají agregační funkce AVERAGE, COUNT, SUM, MAXIMUM a MINIMUM.

Agregační výpočty jsou dalším místem, kde nám mohou ušetřit jistou ránu hodnoty Null. Tyto hodnoty totiž v souhrnné operaci vystupují a tvoří zde samostatnou skupinu. Agregační funkce je však ignorují. Problém může tedy obvykle nastat jen v případě, že je některé z polí uvedených v <SeznamuPolíKSeskupení> zároveň parametrem agregační funkce.

Název kategorie	Název výrobku	SoučetMnožst
Cukrovinky	Gumbär Gummibärchen	753
Cukrovinky	Chocolade	138
Cukrovinky	Maxilaku	520
Cukrovinky	NuNuCa Nuß-Nougat-Creme	318
Cukrovinky	Pavlova	1158
Cukrovinky	Scottish Longbreads	799
Cukrovinky	Schoggi Schokolade	365
Cukrovinky	Sir Rodney's Marmalade	313
Cukrovinky	Sir Rodney's Scones	1016
Cukrovinky	Tarte au sucre	1083
Cukrovinky	Teatime Chocolate Biscuits	723
Cukrovinky	Valkoinen suklaa	235
Cukrovinky	Zaanse koeken	486
Koření	Aniseed Syrup	328
Koření	Genen Shouyu	122
Koření	Grandma's Boysenberry Spread	301
Koření	Gula Malacca	601
Koření	Chef Anton's Cajun Seasoning	453
Koření	Chef Anton's Gumbo Mix	298
Koření	Louisiana Fiery Hot Pepper Sauce	745
Koření	Louisiana Hot Spiced Okra	239

Obrázek 5-12 Klausule GROUP BY vráci souhrnné údaje

## Rozšíření

Operátor rozšíření slouží k definici virtuálních polí, které se vypočítají z konstant a hodnot uložených v databázi, ale které se samy o sobě fyzicky nikam neukládají. Virtuální pole vytvoříme přímou definicí v <SeznamuPolí> příkazu SELECT:

```
SELECT [JednotkováCena]*[Množství] AS RozšířenáCena
FROM [RozpisObjednávek];
```

Výpočet, který nové virtuální pole definuje, může být libovolně složitý. Celý postup je opravdu velice jednoduchý a rychlý, takže uložení vypočteného pole do databáze zpravidla nemá žádné opodstatnění.

## Přejmenování

Posledním z běžně používaných operátorů je přejmenování. Operaci přejmenování můžeme přitom provést buďto v některé množině záznamů ze <SeznamuMnožinZáznamů>, nebo u jednotlivých polí v <SeznamuPolí>. V databázovém stroji Jet má přejmenování množin záznamů následující syntaxi:

```
SELECT <JménoPole> AS <AliasPole>
FROM <JménoTabulky> AS <AliasTabulky>
```

V SQL Serveru se klíčové slovo „AS“ zapisovat nemusí, takže píšeme rovnou:

```
SELECT <JménoPole> <AliasPole> FROM <JménoMnožinyZáznamů> AS <AliasMnožinyZáznamů>
```

Přejmenování je velice užitečné například při definici pohledu se spojením na sebe sama, jak vidíme v následující ukázce kódu:

```
SELECT Nadřízený.Jméno, Employee.Jméno
FROM Zaměstnanci AS Zaměstnanec
INNER JOIN Zaměstnanci AS Nadřízený
ON Zaměstnanec.ČísloZaměstnance = Nadřízený.ČísloZaměstnance;
```

Pomocí této klausule snadno logicky odlišíme oba výskyty stejného pole.

## Transformace

Příkaz TRANSFORM je prvním z rozšíření relační algebry od firmy Microsoft, kterým se zde budeme zabývat. Tento příkaz přebírá výsledky souhrnné operace (s klausulí GROUP BY) a otočí je o 90 stupňů (transponuje). Operaci se častěji říká *křížový dotaz* (crosstab query) a podporuje ji pouze databázový stroj Jet; v SQL Serveru nebyla (zatím) implementována.

Syntaxe příkazu TRANSFORM vypadá následovně:

```
TRANSFORM <AgregačníFunkce>
SELECT <SeznamPolí>
FROM <SeznamMnožinZáznamů>
GROUP BY <SeznamPolíKSkupení>
PIVOT <ZáhlaviSloupců> [IN (<SeznamHodnot>)]
```

Klausule TRANSFORM <AgregačníFunkce> zde definuje souhrnná data, z nichž se naplní výsledná množina záznamů. Příkaz SELECT musí obsahovat klausuli GROUP BY a nesmí obsahovat klausuli HAVING. Podobně jako u každé jiné klausule GROUP BY i zde může <SeznamPolíKSkupení> obsahovat více polí. (Seznamy <SeznamPolí> a <SeznamPolíKSkupení> bývají v příkazu TRANSFORM téměř vždy totožné.)

Klausule PIVOT identifikuje pole, jehož hodnoty se použijí jako záhlaví sloupců. Implicitně zapíše databázový stroj Jet tyto sloupce z množiny záznamů zleva doprava v abecedním pořadí. Nepovinná klausule IN však umožňuje zadání jmen sloupců, které budou uvedeny v pořadí definovaném <SeznamemHodnot>.

Níže uvedený příkaz TRANSFORM podává v podstatě stejně informace jako náš předchozí příklad s operací souhrnu, jehož výsledky jsme viděli na obrázku 5-12.

```
TRANSFORM Count(Výrobky.ČísloVýrobku) AS PočetVýrobkůČísla
SELECT Dodavatelé.Firma
FROM Dodavatelé
INNER JOIN (Kategorie INNER JOIN Výrobky
    ON Kategorie.ČísloKategorie = Výrobky.ČísloKategorie)
ON Dodavatelé.ČísloDodavatele = Výrobky.ČísloDodavatele
GROUP BY Dodavatelé.Firma
PIVOT Kategorie.NázevKategorie;
```

Výsledek této operace TRANSFORM ukazuje obrázek 5-13.

Firma	Cukrovinky	Kofein	Meso / Drůbež	Mléčná výrobky	Močkové produkty	Nápoje	Oblíbené výrobky	Plodiny
Aux joyeux ecclésiastiques						3		
Bigfoot Biowholes						3		
Cooperativa de Quesos Las Cabras'				2				
Escargots Nouveaux					1			
Exotic Liquids						1		
Forets d'érables	1	1						
Formaggi Fortini s.r.l.					3			
Gai pâturage					2			
G'day, Mate				1			1	1
Grandma Kelly's Homestead			2					
Heli Süßwaren GmbH & Co. KG	3							
Karkki Oy	2					1		
Leka Trading		1				1	1	
Lyngby Silic					2			
Ma Maison				2				
Mayumi's		1				1		1
New England Seafood Cannery						2		
New Orleans Cajun Delights		4						
Nord-Ost-Fisch Handelsgesellschaft mbH						1		
Norske Meierier					3			
Pasta Buttini s.r.l.		1						2
Pavlova, Ltd	1	1	1			1	1	
PB Knäckebrot AB							2	
Plutzer Lebensmittelgroßmarkte AG			1	1			1	1
Refrescos Americanas LTDA							1	
Specialty Biscuits, Ltd	4							
Svensk Sjöföda AB						3		
Tokyo Traders				1		1		1
Zaanse Snoepfabriek	2							

Obrázek 5-13 Příkaz TRANSFORM otočí výsledky o 90 stupňů

## Součtové hodnoty

Souhrnný operátor, který je v příkazu SELECT implementován pomocí klausule GROUP BY, generuje záznamy se souhrnnými údaji. Klausule ROLLUP představuje logické rozšíření této operace, protože definuje součtové hodnoty.

Klausule ROLLUP je k dispozici pouze v SQL Serveru a je implementována jako rozšíření klausule GROUP BY:

```
SELECT Kategorie.NázevKategorie, Výrobky.NázevVýrobku,
       SUM([RozpisObjednávek].Množství) AS SoučetMnožství
  FROM (Kategorie INNER JOIN Výrobky
        ON Kategorie.ČísloKategorie = Výrobky.ČísloKategorie)
        INNER JOIN [RozpisObjednávek]
        ON Výrobky.ČísloVýrobku = [RozpisObjednávek].ČísloVýrobku
 GROUP BY Kategorie.NázevKategorie, Výrobky.NázevVýrobku WITH ROLLUP;
```

Výslednou množinu záznamů ukazuje tentokrát obrázek 5-14.

Dostali jsme tedy opět stejnou množinu záznamů jako na obrázku 5-12 ze strany 79, je ale doplněna o jisté řádky navíc: tyto řádky s hodnotami Null (jeden z nich vidíme na obrázku) obsahují součtové hodnoty za celou skupinu nebo podskupinu. Z obrázku tedy vyplývá, že se prodalo celkem 8 137 jednotek nápojů.

Kategorie	Název výrobku	Prodáno celkem
Nápoje	Côte de Blaye	623
Nápoje	Guaraná Fantástica	1125
Nápoje	Chai	828
Nápoje	Chang	1057
Nápoje	Chartreuse verte	793
Nápoje	Ipoh Coffee	580
Nápoje	Lakkalikööri	981
Nápoje	Laughing Lumberjack Lager	184
Nápoje	Outback Lager	817
Nápoje	Rhönbräu Klosterbier	1155
Nápoje	Sasquatch Ale	506
Nápoje	Steeleye Stout	883
Nápoje		8137
Koření	Genen Shouyu	122
Koření	Grandma's Boysenberry Spread	301
Koření	Gula Malacca	601
Koření	Chef Anton's Cajun Seasoning	453
Koření	Chef Anton's Gumbo Mix	298
Koření	Louisiana Fiery Hot Pepper Sauce	745
Koření	Louisiana Hot Spiced Okra	239

Obrázek 5-14 Operátor ROLLUP sčítá souhrnné hodnoty

## Datová krychle

Operátor datové krychle, CUBE, je k dispozici také pouze v SQL Serveru a také se zapisuje jako rozšíření klausule GROUP BY. Klausule CUBE vypočítá v podstatě součet každého sloupce uvedeného v <SeznamuPolíKSkupení> podle každého jiného sloupce. V jistém slova smyslu se tak podobá operátoru ROLLUP, avšak zatímco operátor ROLLUP vytváří součtové hodnoty pro každý jednotlivý sloupec zadáný v <SeznamuPolíKSkupení>, operátor CUBE vytváří součtová data pro další, doplňující skupiny.

Jestliže máme například v <SeznamuPolíKSkupení> zapsána tři pole – A, B a C – pak vrátí operátor CUBE následujících sedm agregačních údajů:

- Celkový počet hodnot C.
- Celkový počet hodnot C, seskupený podle A.
- Celkový počet hodnot C, seskupený podle C v rámci A.
- Celkový počet hodnot C, seskupený podle B v rámci A.
- Celkový počet hodnot C, seskupený podle B.
- Celkový počet hodnot C, seskupený podle A v rámci B.
- Celkový počet hodnot C, seskupený podle C v rámci B.

## Stručné shrnutí

V této kapitole jsme hovořili o úpravách základních relací s pomocí různých relačních operátorů a ukázali jsme si několik příkladů, implementovaných v jazyce SQL. Zároveň jsme se zde podívali ještě jednou na problém hodnot Null a na tříhodnotovou logiku.

Mezi standardní relační operátory patří operátory restrikce (omezení) a projekce (promítání), které slouží k výběru jisté podmnožiny z jedné množiny záznamů. Operátory spojení, sjednocení, rozdílu a kartézského součinu definují způsob sloučení dvou množin záznamů. Všechny tyto operátory, s výjimkou operátoru rozdílu, se dají implementovat pomocí příkazu SELECT jazyka SQL. Rozdíl se někdy dá napsat jako příkaz SELECT, někdy však vyžaduje jiné postupy, které jdou již mimo rámec této knihy.

Zmínili jsme se zde také o několika speciálních operátorech. Operátor souhrnných výpočtů a operátor rozšíření provádějí nad daty jisté výpočty. Operátor přejmenování popisuje záhlaví sloupců, které se vypíší při zobrazení výsledné množiny záznamů. Operátory TRANSFORM, ROLLUP a CUBE jsou pak speciálními rozšířeními jazyka SQL, které ve svých produktech implementuje firma Microsoft; každý z nich nabízí speciální možnosti vytváření souhrnů a prohlížení dat.

Tímto přehledem relační algebry jsme završili první část knížky. Teorie relačních databází je poměrně komplikovaná, takže bychom samozřejmě našli ještě spoustu problémů a drobností, na které jsem v knize, určené jako úvod do problematiky, prostě nenašla místo. Seznámila jsem vás ale se vsemi nejdůležitějšími součástmi této teorie. Ve zbytku knihy se pustíme do praktických aspektů návrhu databázových systémů a uživatelských rozhraní.

**2**

# **ČÁST**

**NÁVRH RELAČNÍCH  
DATA BÁZOVÝCH SYSTÉMŮ**



V první části této knihy jsme se zabývali základními principy návrhu relačních databází. Struktura dat je ale pouze jednou z mnoha součástí databázového systému – je to samozřejmě velice důležitá součást, přesto však je jedna z mnoha. Na začátku druhé části se proto podíváme na některé zbývající aspekty návrhu databázových systémů.

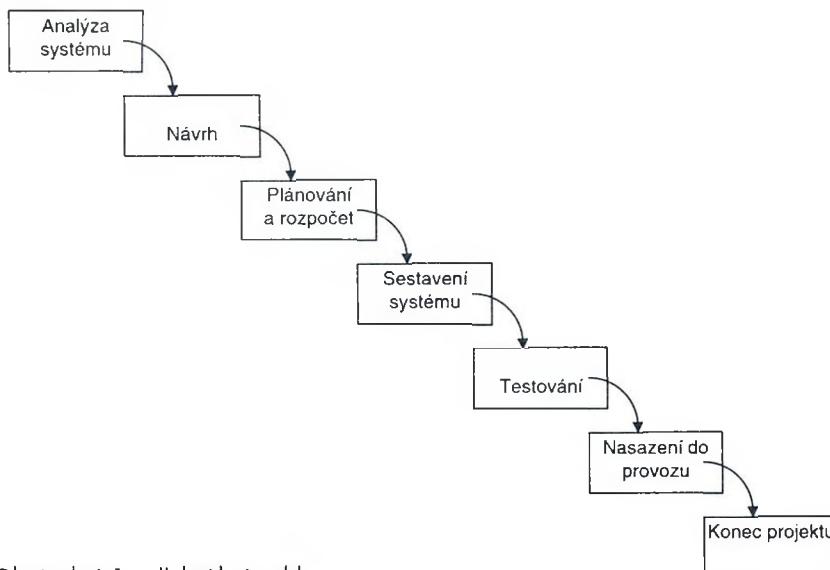
Tato část bude tedy rozebírat většinu činností spojených s analýzou a návrhem databázových systémů, tedy například definici systémových parametrů a pracovních procesů, myšlenkový databázový model a databázové schéma. Návrh uživatelského rozhraní je natolik složité téma, že se mu věnujeme v samostatné, třetí části knihy.

Zde tedy budu hovořit pouze o analýze a návrhu databázových systémů; vlastní implementace reálné aplikace je již mimo rámec této knihy. Analýza a návrh se ale nedají od zbytku celého procesu oddělit a nemohou existovat samy o sobě, izolovaně; pro začátek si proto řekneme něco málo o životním cyklu projektů.

## Modely životního cyklu

Kdysi dávno postupovali systémoví analytici při vývoji systémů metodou známou jako „vodopádový model“. Ten existuje v několika různých verzích; jednu z nich, relativně jednoduchou, ukazuje obrázek 6-1.

Celý proces začíná tedy analýzou systému, které se někdy říká analýza požadavků, protože se zaměřuje na to, co zákaznická organizace a její uživatelé od systému potřebují. Jakmile je analýza systému dokončena a odsouhlasena, vytvoří se podrobný návrh celého systému. Po této fázi následuje plánování a tvorba rozpočtu projektu; poté již zbývá systém sestavit, otestovat a nasadit do provozu.



Obrázek 6-1 Vodopádový model

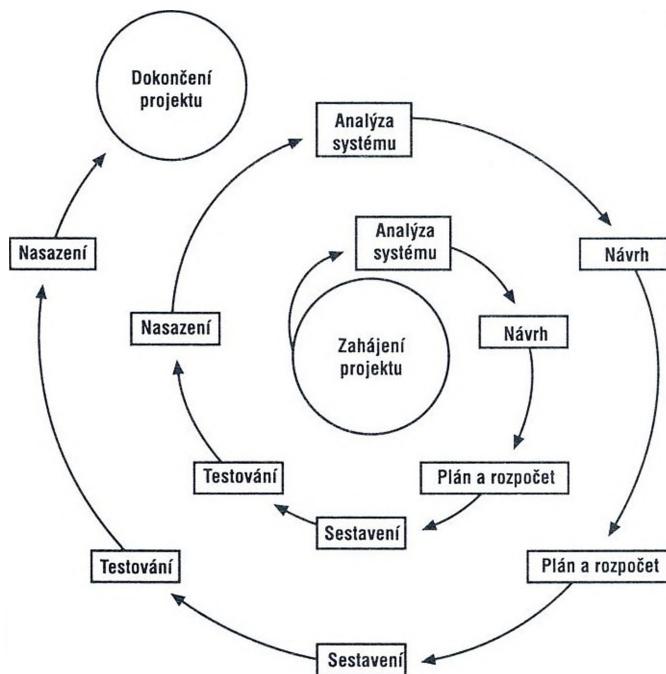
Vodopádový model našemu oku zajisté lahodí. Každá činnost se pěkně dokončí a odsouhlasí ještě před započetím další, přičemž model sleduje rozumnou kontrolu nad rozpočtem, nasazením pracovníků a časem. Jestliže se vám podaří vodopádový model nasadit včas s dodržením původního finančního plánu, budou vás zákazníci mít upřímně rádi.

Tento model v sobě ale skrývá jeden zásadní problém – realita se většinou nedá tak pěkně „nalinikovat“. Model předpokládá, že během zpracování určitého úkolu máme pro něj k dispozici veškeré potřebné informace a nedovoluje pozdější vstup žádných dalších informací do projektu. Tato situace je ovšem, snad jen s výjimkou opravdu malých systémů, mimořádně nepravidelně.

Vodopádový model neuvažuje také v průběhu řešení projektu žádné změny aplikačních požadavků. Předpokládat, že systém, který vyhovuje potřebám organizace někdy na začátku projektu, bude vyhovovat i na konci dvouletého nebo tříletého vývoje, je ale čiré bláznovství. Jestliže se pokusíte do provozu nasadit systém, který je k ničemu, asi vás žádný klient nebude mít rád, přestože jinak dodržíte časový i finanční plán celého projektu.

Na tomto místě si ale musíme říci, že veškeré činnosti, které jsme identifikovali ve vodopádovém modelu, jsou stanoveny naprosto správně. Vynecháním libovolné z nich si ve vývojovém projektu zaděláváme na téměř jistý malér. Problémem tohoto modelu je tedy jeho linearita, neboli česky řečeno chybnný předpoklad, podle něhož do jednou ukončené fáze vývoje nebudeme muset nikdy vstupovat znova.

Pro řešení uvedených problémů vodopádového modelu bylo navrženo několik alternativních modelů životního cyklu projektu. Spirálový model tak v podstatě předpokládá několik iterací (opakování) činností z vodopádu; záběr každé další iterace se oproti předchozí rozšiřuje, až se dostaneme k dokončení projektu. Schéma spirálového modelu ukazuje obrázek 6-2.

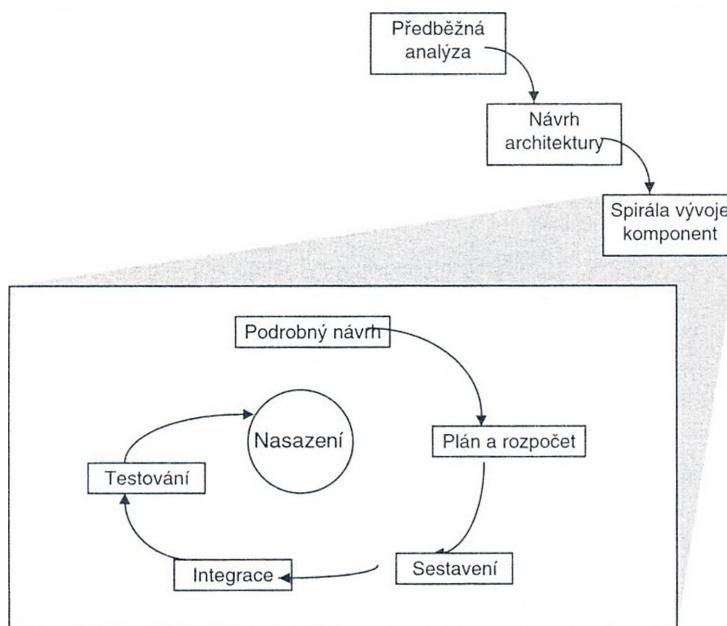


Obrázek 6-2 Spirálový model

Problém spirálového modelu spočívá v tom, že pokud se jím budeme striktně řídit, vezmeme celkovou strukturu a záběr projektu v úvahu až příliš pozdě, takže hrozí určité (a to nikoli bezvýznamné) riziko, že při pozdějších iteracích přijdu výsledky dřívější práce vniveč. Já jsem ze spirálového modelu měla vždy ten pocit, že je nejlepší cestou jen k překračování rozpočtu projektu a k uvádění vývojářů v zoufalství. Takováto situace je velice nebezpečná zejména právě pro databázové projekty, kde i zdlánlivě malé rozšíření záběru systému může vést k zásadním změnám v sémantice dat a změna databázového schématu si může vyžádat neočekávané změny v celém systému.

Já osobně proto u větších systémů doporučuji jiný model (a také jej ve své práci používám), ktere mu se říká inkrementální vývojový model nebo evoluční vývojový model; vidíme jej na obrázku 6-3.

V tomto modelu, který je do značné míry jistou variací na předcházející spirálový model, se nejprve provede předběžná analýza celého systému, nikoli pouze jeho části. Po této etapě následuje návrh architektury, a to opět celého systému. Cílem návrhu architektury je přitom zejména definovat jednotlivé komponenty systému, které se již budou implementovat víceméně nezávisle, a popsat komunikace a vzájemné závislosti mezi těmito komponentami. Podrobný návrh a implementace každé z komponent se pak provádí podle libovolného vhodného vybraného modelu. Já jsem do obrázku 6-3 zanesla v této fázi spirálový model, protože znamená pro návrh a implementaci vyšší flexibilitu.



Obrázek 6-3 Inkrementální vývojový model

Všimněte si, že spirála obsahuje v tomto schématu jeden úkol navíc, a sice integraci. Ve spirálovém modelu je integrace komponent samozřejmě také obsažena (implicitně), podle mých vlastních zkušeností je ale právě v inkrementálním vývojovém modelu integrace o něco složitější. To je také jeden z důvodů, proč při vývoji komponent používám raději spirálový model. Jestliže podrobný návrh komponenty odložíme až na okamžik, kdy se skutečně pustíme do jejího vývoje, pak do ní můžeme zapracovat veškeré poznatky, které jsme získali během integrace předcházejících komponent a snad se můžeme vyhnout i problémům, s jakými jsme se při integraci setkali.

Inkrementální vývojový model předpokládá, že se každý rozsáhlější systém dá dekomponovat nebo rozložit do určitých samostatných komponent; to ale samozřejmě pro všechny systémy nemusí nutně platit. Výsledkem tohoto postupu může být také velké množství pomocného kódu. Dejme tomu, že například obrazovka pro pořizování dat má podle našich úval volat jistou komponentu Microsoft ActiveX, která provede hledání podle kódu zákazníka a po nalezení vykoná určitou operaci; tuto komponentu ActiveX ale ještě nemáme vytvořenou. Vývojovému týmu pak nezbývá nic jiného, než provizorně napsat maketu komponenty, která se dá alespoň bezchybně zavolat. A složitější systémy mohou obsahovat i dosti významné množství takového pomocného programového kódu.

Kromě toho zde hrozí jisté riziko, že se příslušnou komponentu ActiveX nikdy nepodaří implementovat; třeba se nevejde do sešněrovaného rozpočtu, anebo jednoduše později usoudíme, že nemá smysl a z návrhu ji zcela vypustíme. Pokud s něčím takovým vůbec nepočítáte, můžete nakonec v hotovém systému distribuovat také prázdné makety komponent, jejichž jediným smyslem

je zabránit v havárii nějakých ostatních komponent. To není právě elegantní řešení a vysvětlovat je programátorovi, který dostal „za krk“ údržbu systému, také nebude nic příjemného.

Analýza systému a návrh architektury se provádí již na začátku projektu, a proto je zde určité riziko, že později zastarají a budou neaktuální; to je ovšem, jak si zajisté vzpomínáte, jednou z hlavních nevýhod vodopádového modelu. Z toho důvodu je důležité uvedené dva kroky – a zejména pak analýzu požadavků, protože u té je větší „šance“ na případné změny – ještě před zahájením podrobného návrhu jednotlivých komponent opravdu důkladně prověřit a zrevidovat. Často se takto můžeme dočkat velice příjemného překvapení, jak snadno se dají změny v požadavcích promítнуть do změn ještě nenavržených komponent – někdy dokonce stačí změnit jen pořadí, v jakém budeme nové komponenty vyvíjet – takže žádná část naší dříve provedené práce nepřijde vniveč.

Inkrementální vývojový model má ale několik důležitých výhod. Na začátku projektu nadefinujeme totiž jen „celkový pohled“ na nový systém, takže riziko promarněné práce je v něm opravdu minimální. Protože rozsáhlé projekty se rozkládají do menších komponent, dají se projekty těchto menších jednotlivých komponent lépe udržovat a spravovat. A díky rozdělení systému do komponent můžeme některé základní funkce systému nabídnout uživatelům již v časných stádiích vývoje projektu. Takto začíná systém dosti brzo přinášet kýzené ovoce a současně se do něj dostává mechanismus, pomocí něhož do dalších vývojových prací přirozenou cestou zapojíme i samotné uživatele.

## Proces návrhu databáze

V každém zvoleném modelu obecného vývoje musíme provést určité činnosti spojené s analýzou a návrhem. Nezáleží na tom, jestli je provedeme postupně nebo opakovaně, jestli pracujeme nad celým systémem nebo v jediné jednotlivé komponentě, jestli používáme formální nebo neformální postupy; v každém projektu musíme každým z těchto pěti kroků alespoň jednou projít.

### Definice parametrů systému

V ideálním případě by měly práce na každém projektu začít jasnou definicí cílů, kterých chceme dosáhnout, a stanovením, proč jich chceme dosáhnout; pak už by nám měl být souzen pouze úspěch. Většina projektů ale žádnou takovouto definici před zahájením prací nemá; k tomu tedy slouží právě tato první fáze procesu návrhu. Cíle projektu říkají, „proč“ jej řešíme. Na základě cílů si uvědomíme, „co“ máme udělat a stanovíme tak záběr budoucího systému. Jakmile porozumíme cílům projektu a jeho záběru, dokážeme již reálně určit kritéria návrhu a víme tedy, „jak“. Všechny tyto aspekty rozebíráme podrobněji v kapitole 7.

### Definice pracovních procesů

Databázové systémy zabezpečují na první pohled jen operace spojené s ukládáním a načítáním dat, ve skutečnosti však většina z nich podporuje jeden nebo více pracovních procesů. Uživatelé neukládají data jen tak pro nic za nic; chtějí je nějakým způsobem využívat. Správně porozumět pracovním procesům, které mají naše data podporovat, je životně důležité pro správné pochopení sémantiky datového modelu. O pracovních procesech pojednává podrobněji kapitola 8.

## Sestavení myšlenkového datového modelu

Myšlenkový (konceptuální) datový model nedefinuje pouze jistou množinu tabulkových struktur, ale zejména využití dat v celém systému. Do datového modelu tak spadá nejen logický datový model, ale také popis způsobu práce s daty v jednotlivých pracovních procesech. O myšlenkovém datovém modelu hovoříme v kapitole 9.

## Příprava databázového schématu

V databázovém schématu se myšlenkový datový model převádí do podoby fyzických pojmu. Do databázového schématu tak náleží zejména popis tabulek, které budou v systému implementovány, a fyzická architektura dat. Fyzickým architekturám a databázovým schématům se podrobň věnujeme v kapitole 10.

## Návrh uživatelského rozhraní

Jestliže je uživatelské rozhraní hloupé, zmatečné nebo arogantní, nebude zřejmě systém úspěšný bez ohledu na jakkoli ohromnou technickou výkonnost. Uživatelské rozhraní je totiž pro většinu uživatelů totéž co systém. O návrhu uživatelského rozhraní hovoří část 3 této knihy.

## Poznámka o metodologiích a standardech návrhu

Osobně nejsem velkým příznivcem nějakých zaručených postupů pro návrh počítačových systémů, které jsou rozpracované do jednotlivých kroků a do nejmenších detailů. Podle mých zkušeností mohou skutečně dobrému návrhu někdy i překážet, protože analytici se záhy starají spíše o zaškrťávání nějakých dotazníků, než o to, aby opravdu porozuměli potřebám uživatele.

U velkých systémů, jejichž návrhu se účastní velké množství analytiků a několik programátorských týmů, je však zcela zřejmě nezbytné zavést určité společné procedury pro řízení celého projektu. Existuje několik propracovaných metodologií, z nichž u většiny je k dispozici nějaký automatizovaný podpůrný nástroj. Ne, nebudu zde žádnou takovou metodologii ani žádný nástroj doporučovat. Za prvé je to otázka spíše jakési víry (někdy i fanatické), než otázka rozumového zdůvodnění, a za druhé, podobně jako je tomu třeba i u konvence pojmenování proměnných, i zde je důležitá spíše samotná skutečnost existence nějaké metodologie než konkrétní vybraná z nich.

Na druhé straně ale plně chápu, že příprava dokumentů pro návrh systému může člověka odrazovat, alespoň v několika prvních systémech, které takto navrhujete. Kapitola 11 rozebírá proto celý proces obecně.

# DEFINICE PARAMETRŮ SYSTÉMU

7

„Systém je navržen k tomu, aby dělal *něco*, ne *všechno*.“ – Robert Hall, výzkumný pracovník Severoamerického letectva.

Říká se, že Bob Hall pronesl tuto větu před viceprezidentem, který se již kdoví pokolikáté pokoušel rozšířit záběr jeho projektu, a že ho téměř stála zaměstnání. (V návrhu systémů se také, podobně jako všude jinde, vyplatí vědět, s kým právě hovoříte.) Věta je nicméně jednou z nejpravdivějších poznámek, jaké jsem na téma procesu návrhu systémů kdy slyšela. Jestliže má být projekt opravdu úspěšný, musíte nejet být schopni dobře popsat, čeho se snažíte dosáhnout, ale také projektu stanovit určité rozumné hranice, za které již nepůjdete. Pokud nedokážete s přijatelnou jistotou říci, „*toto uděláme, ale tolik dělat nebudeme*“, mohu vás ubezpečit, že vás *rozhodně* čekají velké problémy.

Tento proces definice systému se skládá ze tří následujících kroků:

- Stanovení cílů, a to nikoli pouze cílů systému, ale i projektu jako celku.
- Sestavení kritérií návrhu systému, na jejichž základě budeme posuzovat veškeré kompromisy, které v průběhu celého návrhu a implementace učiníme, a pomocí nichž budeme posléze hodnotit úspěch či selhání systému.
- Určení záběru systému – o co se budeme snažit a co v systému dělat nebudeme.

## Stanovení cílů systému

Definovat (nebo přesněji řečeno zjistit) cíl a záběr či rozsah systému by mělo být docela jednoduché. Někdy, máme-li notnou dávku štěstí, to jednoduché opravdu je. Někdy je cíl přímo součástí prvotního zadání. Častěji je ale definice záběru systému dosti komplikovaným procesem, ve kterém se spojují formální analytické postupy, nezbytné kompromisy při návrhu a více než troška diplomacie.

Nejdůležitějším faktorem při určování záběru systému i návrhových kritérií, podle kterých se bude hotový systém hodnotit, je obvykle základní cíl vývojového projektu. Tento cíl je konec konců také prvotním *důvodem*, proč se systém vůbec implementuje. Je zřejmé, že dokud jasně nepochopíme, čeho se vlastně snažíme dosáhnout, nemůžeme provádět žádné informované rozhodnutí ohledně jakéhokoli jiného aspektu daného projektu.

Nezaměňujte ale pojmy „cíl“ a „zadání“. Na začátku většiny projektů stojí stručný popis vyvíjeného systému a předběžný rozpočet. Zadání tedy může znít „automatizovat současný systém pořizování objednávek“ a na jeho realizaci budeme mít půl druhého roku a milion korun. Věta „automatizo-

vat současný systém pořizování objednávek" vyjadřuje ale pouze zadání, není to cíl. Cílem je důvod, nebo přesněji řečeno množina důvodů, pro které se projekt řeší.

Hovořit o stanovení „cílů“ systému je ve skutečnosti poněkud zavádějící. Drtivá většina systémů má mnoho různých cílů, ať už hmatatelných nebo pomyslných, přičemž jejich zjištování znamená často mravenčí, až detektivní práci. Proč se vůbec snažíme systém pořizování objednávek automatizovat? Máme celý proces urychlit? Zvýšit jeho přesnost? Snížit náklady? Zlepšit povědomí společnosti mezi jejími zákazníky, uživateli? Udelat radost řediteli? Za cíle systému můžeme nejspíše označit všechny tyto náměty a kromě nich i řadu dalších.

Rozhodně tím ale nemyslím, že byste se měli vrhnout na lidi ve firmě a zaútočit na jejich soukromí, nebo se začít vyptávat na dívčerné firemní informace; takto zjištěné cíle by určitě do značné míry spadaly do kategorie „do toho vám nic není“. Pro analýzu systému nepotřebujete vědět, že se šéf oddělení nebojí brouzdat na divokých vlnách Internetu. Nemusíte vědět ani to, že pokud se má systém zaplatit, musí zkrátit průměrnou dobu zpracování jedné objednávky z 10 minut na 2 minuty.

V podstatě ani nemusíte rozumět oněm prázdným frázím, jaké s oblibou používají lidé z oblasti prodeje a marketingu, třeba „umístění výrobků na trh“ a „vyjít vstří očekáváním uživatelů“. To naštěstí neznamená, že byste se museli vrátit do školy a všechno teď dohonit; stačí se klienta ještě jednou zeptat. Každý pod touto obecnou proklamací míní něco trochu jiného, takže i proto se zeptat opravdu musíme.

Problém těchto obtížně uchopitelných cílů spočívá zejména v tom, že se často velice obtížně převádí na měřitelná kritéria návrhu. Někdy stačí uživatele rozumně „natuknout“ a už dokážeme tento neměřitelný, neuchopitelný cíl převést na něco měřitelného. Cíl, vyjádřený větou „vyjít vstří očekáváním uživatelů“, poukazuje například obvykle na nějaký problém ve službě zákazníkům; takový problém se již buďto snadno převede na určité měřitelné kritérium, nebo se naopak zcela vypustí.

Jestliže má náš klient například problémy s dodržováním dodacích lhůt a má důvod se domnívat, že příčinou tohoto problému jsou nereálné, nesplnitelné dodací lhůty, které zákazníkům slibují obchodní zástupci ve snaze získat zákazníka, pak má systém pořizování objednávek přímý vztah k cíli korigovat nereálná očekávání uživatelů. Můžeme tedy například určitým způsobem omezit délku časového intervalu mezi datem převzetí objednávky a příslušeným datem dodání zboží. Pokud jsou ale problémy spojeny s řízením kvality ve výrobním procesu, znamená to, že objednávkový systém nedokáže nijak pomoci a pak je vaší povinností na to klienta upozornit. To vůbec neznamená, že by projekt nemělo smysl řešit; všichni jeho účastníci musí ale pochopit, že takto stanovený cíl nedokáže systém pro pořizování objednávek *přímo* ovlivnit.

Takto snadno se ale samozřejmě nedají zdaleka převést všechny neměřitelné cíle. Příkladem je třeba oblíbené „umístění firmy na trhu“. Dejme tomu, že vás klient osloví s požadavkem vytvořit webové sídlo, které by „společnost umístilo na technologickou špičku v oboru“. Takto formulované cíle budí samy o sobě nic neznamenají, anebo k jejich naplnění postačí již samotná existence navrhovaného systému. Jestli je tomu tak či onak, to zjistíme docela snadno; stačí se klienta zeptat, jak poznáte, že jste požadovaného cíle dosáhli. Obvykle je docela velká šance, že pokud se podaří splnit i ostatní cíle ohledně výkonnosti systému, splní se i onen neměřitelný abstraktní cíl.

Dalším typem výroků, které musíme podrobit důkladnému rozboru, jsou cíle vyjádřené obecnými pojmy jako například „zlepšit“ a „zredukovat“. Velice běžnými, a zároveň také velice vágními cíli jsou například požadavky „zlepšit efektivitu“ a „zvýšit produktivitu“. Jak zjistíte, že jste těchto cílů dosáhlí? Jeden můj klient mi vyprávěl velice krásnou (a téměř určitě neskutečnou) příhodu o významu měřitelnosti v požadavcích na pracovníky. Říkal mi, že jakémusi mladému obchodníkovi řekl, že jeho úkolem bude „propagovat zboží a služby naší společnosti“. Mladík tedy otevřel dveře kanceláře a zvolal: „Všichni musí ty naše krámy kupovat, protože jsou fakt betálny.“ Obávám se, že něco takového neměl jeho nadřízený na mysli.

Pokud nemáte velký smysl pro černý humor, nebo jej alespoň neuplatňujete u svých bankovních účtů (neboli pokud si chcete rozumně vydělat), musíte během úvodní analýzy rozhodnout o mře, v jaké je určité obecné zlepšení ve skutečnosti požadováno. Zlepšit efektivitu o kolik? Zvýšit produktivitu z čeho na kolik? V tomto rozhodování se ale skrývá ještě jedna léčka. Je velice snadné říci, že cíle musí být objektivně a přímo měřitelné; rozhodně se dá souhlasit například s tím, že cíl „zkrátit dobu zpracování faktury z 10 minut na 3 minuty“ je lépe zformulovaný než obecné „zlepšení efektivity“. První tvrzení v sobě ale předpokládá, že známe současnou dobu nutnou ke zpracování faktury; zjistit přesný údaj může být dosti obtížné.

Náklady na přesné prozkoumání problému mohou často převýšit i riziko spojené s případnou chybou. V našem malém příkladu s fakturami nejspíše není nutné vyslat do kanceláří tým analytiků se stopkami, kteří přesně zjistí, jak dlouho zpracování jedné faktury vlastně trvá, i když i to jsem nejednou viděla. Před několika lety jsem pracovala na projektu, v němž jisté ministerstvo utratilo navíc 50 000 dolarů za ověření, jestli byl oprávněný nákup jistého krabicového grafického softwaru s doporučenou prodejnou cenou 2 500 dolarů.

Řešení převodu těchto obecně formulovaných požadavků do podoby měřitelných kritérií návrhu spočívá v definici jistého měřítka a myšlence pojmu „dost dobré“. Jestliže je vaše firma nebo vaše vlastní kariéra na implementaci nového počítačového systému závislá, pak si musíte být naprosto, naprosto jistí, co děláte. Jestliže navrhujete systém malého rozsahu, který nebude mít žádný zásadní dopad na fungování společnosti, pak si můžete dovolit být jakoby méně pečliví. Pokud se vrátíme k našemu příkladu s fakturami, zde stačí vědět, že například průměrný pracovník dokáže za současného stavu zpracovat denně zhruba 25 faktur. K tomu nemusíme provádět žádnou bühlíjak podrobnou analýzu; to nám je téměř jistě s dostatečnou přesností schopen povědět vedoucí příslušného oddělení; proto si vás ostatně zavolali. Jsem si ale jista, že třeba na takovém ministerstvu obrany se dělá sáhodlouhý průzkum, jehož výsledkem je například objednávka šroubováku za 400 dolarů.

Vždy také stojí se zeptat, proč se vůbec má daného zlepšení dosáhnout. Dejme tomu, že se například zpracování určité agendy neúměrně zpožduje a stále se hromadí nevyřízené faktury. Šéf takového oddělení má dvě možnosti: buďto se pokusit celý proces urychlit, nebo zaměstnat další pracovníky. A jestliže víte o nedostatcích a hromadění nevyřízených faktur a jestliže znáte odlhadované zvýšení objemu prodeje, můžete snadno zjistit, jak velké zlepšení je skutečně potřeba.

Číslo, ke kterému takovým prostým výpočtem dospějete, se přitom může lišit od údaje, jenž původně vyslovil váš klient. Je zřejmé, že pokud klient požaduje zkrácení času zpracování faktur na polovinu, pak pro dosažení tohoto cíle musíte udělat všechno, co je ve vašich silách. Jestliže ale víte, že pro potřeby daného systému bohatě stačí dosáhnout zkrácení o 25 procent, pak máte ur-

čitý vyjednávací prostor a můžete se s klientem bavit o tom, jestli se máte snažit za každou cenu o zlepšení číselného ukazatele rychlosti zpracování, nebo jestli je důležité zvážit také spolehlivost systému a pohodlí při práci s ním.

Klient vám rozhodně nebude úmyslně klást nesmyslné či nerozumné požadavky, nebude vás záměrně uvádět v omyl nebo vám klást za vinu věci, které nejsou vaši chybou. Součástí vaší práce je ale také pomocí klientovi se rozhodnout, v čem jim navrhovaný databázový systém může a nemůže pomoci. Lidé z počítačové branže si více než často naříkají, jak by jim usnadnilo život, kdyby klienti věděli, co chtějí. Klienti ale opravdu *vědí*, co chtějí; pouze tyto svoje potřeby nedokáží vyjádřit v řeči počítačového systému. A to je vaši úlohou.

Jinou variací na toto téma je klient, který vám předloží hrst náčrtků nějakých obrazovek a ukázkových sestav, jejichž implementace je nebo není možná. V tomto případě nám klient přednáší již řešení, nikoli samotný problém; na nás je pak prozkoumat skutečné důvody, které leží za takovým amatérským „návrhem systému“, přičemž nesmíme sklouznout k závěru, že je dotyčný budто pitomec, nebo že tohle není jeho práce, anebo že vzal práci za špatný konec; to všechno ovšem vyžaduje jistou dávku taktu.

Jediné, co vám v těchto případech mohu doporučit, je zkusit to. Jestliže klient vašim otázkám stále zatvrdí, pokuste se jakoby mezi řádky pronést něco jako „pokud mi pomůžete pochopit vaše podnikatelské prostředí, budu vám i já schopen lépe pomoci“. Pokud vám ani toto nepomůže, nezbývá než systém implementovat s tím, co máte, anebo z projektu úplně vycouvat (což, jak naprostě chápou, není úplně vždycky možné). Nejlepší, co snad ještě můžete udělat, je posoudit důkladně předložený návrh, a pokud v něm najdete nějaké podstatné chyby, probrat je s klientem ve smyslu: „Toto udělat nemůžu, ale mohl bych udělat tohle a tamto. Co tedy nejlépe vypadá vašim potřebám?“

Výše popsaný proces „vypátrání“ cílů neplatí jen pro databázové systémy. Od ostatních počítačových systémů se databázové systémy liší především tím, že téměř jako vedlejší produkt vytvářejí jakýsi balík dat o vlastní organizaci. Tento balík dat, ať už je to seznam odběratelů nebo evidence faktur, může mít pro organizaci velkou hodnotu a může jít i mimo rámec pracovních procesů, které přímo podporuje.

To samozřejmě *neznámená*, že bych se vám snažila vnitit myšlenku, podle které bychom na každý projekt měli pohlížet jako na příležitost k vytvoření nějakého celopodnikového úložiště dat. Tvrdím zde pouze tolik, že data, která jsou součástí systému, mají určitou hodnotu i pro ostatní části podniku nebo pro jiné procesy ve stejně jeho části a že tuto hodnotu je vhodné zvlášť posoudit. Data, která v systému shromáždíme, se třeba mohou velice dobře využít i v jiném oddělení podniku, i když takové věci se stávají mnohem méně, než by se na první pohled mohlo zdát.

Jestliže například fakturační systém v sobě udržuje seznam zákazníků, zatímco oddělení prodeje potřebuje seznam jejich adres pro zaslání nové nabídky, může být skutečně vhodné jim tento seznam zpřístupnit. Pokud nic jiného, může společný seznam ušetřit nezáživnou „úřední“ práci spojenou s opisováním všech jmen a adres. To ovšem neznámená začlenit do fakturačního systému veškeré funkce pro hromadnou korespondenci a pro práci s adresami.

Jediný důvod, proč zde o těchto možnostech hovořím – i když s rizikem určité megalománie – je, že někdy může být vhodné *mírně* upravit datové struktury, aby vyhovovaly i jinému účelu. O těchto postupech si více povíme v kapitole 9, přesto mi ale již nyní dovolte přednест alespoň stručný příklad.

Vzpomeňte si na náš výklad o atomických hodnotách; tehdy jsem vám řekla, že adresa může být, v závislosti na sémantice daného systému jednoduše nějakým jednolitym blokem dat, který se někde vytiskne na adresní štítek. Pokud ale data mohou mít smysl i pro někoho jiného – ovšem jen za podmínky rozdělení jednotlivých atributů – je rozumné zatížit současný systém jistou malou režii navíc a vyhnout se tak nutnosti opakování dat v jiném systému.

Pečlivě si ale ověřte, že tato režie navíc je opravdu malá a že sdílení společných dat je opravdu proveditelné. Viděla jsem (a k mé ostudě se musím přiznat, že jsem i sama implementovala) systémy, do kterých se musely zadávat celé skupiny údajů, jež neměly žádný přímý vztah k danému pracovnímu procesu, a to jen proto, že by někde jinde někdy jindy někomu jinému mohly být možná užitečné. Něco takového se velice snadno udělá i neúmyslně, takže pokud někdy budete hovořit o „možnostech budoucího růstu“, dávejte si pozor, jestli ve skutečnosti nemáte na mysli „přidávání zbytečné režie navíc“.

Po stanovení první množiny cílů můžeme přejít k dalším činnostem v procesu analýzy: je to sestavení kritérií návrhu a určení záhlbu systému. Nepodlehněte ale mylnému přesvědčení, že cíle jsou jednou provždy pevně dané. Vždy musíte být připraveni cíle systému v pozdějších fázích projektu znova přehodnotit.

U každého projektu, který trvá více než několik týdnů, si můžete být jisti, že aplikační požadavky budou podléhat změnám. Objem prodeje se radikálně liší od původních odhadů, sloučení s jinou společností znamená zvýšení počtu zaměstnanců, a nikoli jejich snížení; také při jakýchkoli externích událostech se může dost dložit stát, že budeme muset přehodnotit cíle projektu. Znamená to, že během doby řešení dlouhého projektu si musíme s klientem neustále ověřovat, jestli se nic podstatného nějak drasticky nezměnilo.

I u projektů s poměrně kratší dobou trvání můžete ale v pozdějších fázích řešení zjistit, že některé z původních cílů jsou nesmyslné nebo nedosažitelné. Vzpomeňte si na jeden z velkých problémů klasického vodopádového modelu: v něm se mlčky předpokládá, že v každém okamžiku přesně víme všechno, co právě potřebujeme. V reálu však svůj pohled na potřeby systému budeme v průběhu celého řešení projektu neustále rozšiřovat a znalosti o něm prohlubovat. Při zapracování nových poznatků budeme muset často přehodnotit cíle systému, i když to třeba znamená jenom ověření, po kterém řekneme: „Uf, všechno je v pořádku a naše cíle jsou i nadále platné.“

## Sestavení kritérií návrhu

Jakmile do určité rozumné míry pochopíme měřitelné a neměřitelné cíle projektu, můžeme se pustit do sestavení kritérií návrhu. Samozřejmě že ne vždy jde všechno tak pěkně po sobě, takže v praxi budeme hodnotící kritéria návrhu sbírat již v průběhu definice cílů. Pro přesnost našeho výkladu budeme ale předpokládat, že jsme nejprve vytvořili seznam cílů projektu a nyní potřebujeme sestavit seznam kritérií, podle kterých budeme posuzovat úspěch či selhání našeho projektu.

Kritéria návrhu s cíli projektu úzce souvisí. Jestliže nám cíle projektu řeknou, kam se máme ubírat, pak nás kritéria návrhu dokáží upozornit, že jsme již dorazili do cíle. Každé z kritérií návrhu systému by přitom mělo přímo podporovat jeden nebo více z definovaných cílů systému. Pokud přesto najdete nějaké důležité kritérium, které zdánlivě žádnému konkrétnímu cíli neodpovídá, znamená to téměř jistě, že seznam cílů projektu není úplný.

Přiřadit každé jednotlivé kritérium určitému cíli, který podporuje, není úplně nezbytně nutné, je to ale každopádně užitečné cvičení, a to i pro zkušené analytiky. Jedno z největších nebezpečí každého projektu je totiž to, že nevíme, co přesně nevíme. To platí zejména v případě, že se nacházíme na pozici externího konzultanta, a nemáme tudíž příliš velké zkušenosti (pokud vůbec nějaké) s činnostmi prováděnými v dané organizaci. Vzájemně si neodpovídající cíle a kritéria velice dobře poukazují na to, že jsme dosud neporozuměli všemu, čemu rozumět máme.

Kritéria návrhu mají obvykle jednu z následujících tří podob:

- Přímo měřitelné požadavky, jako například: „Vytisknout odloženou sestavu za méně než dvě hodiny.“
- Kritéria prostředí, jako například: „Pracovat ve stávající lokální síti LAN.“
- Obecné strategie návrhu, jako například: „Poskytovat uživatelům kontextově závislou návodě.“

Přesná definice těchto kategorií není bůhvýjak podstatná, z této trojice je ale docela dobře jasné, do jaké míry rozumíme analyzovanému systému. Většina kritérií by měla spadat do první nebo druhé kategorie. Pokud zjistíte, že máte vytvořeny pouze obecné strategie návrhu, rozhodně se s vámi vsadím, že ve skutečnosti dostatečně dobře nechápete problém, o jehož řešení se snažíte.

### **Poznámka**

*Ať už si zvolíte jakákoli kritéria, pokud jich jednou dosáhnete, zastavte. Znamená to, že váš projekt je dokončen. Můžete oslavovat. Tato věta ovšem není tak docela triviální, jak by se na první pohled mohla zdát. Dovolte mi jednoduchý příklad. Dejme tomu, že optimalizujete určitý úsek programového kódu. Podle stanovených kritérií návrhu musí daná funkce vypočítat jistou hodnotu za méně než 10 sekund. Vám se podaří funkci upravit tak, že výpočet provede za 9 sekund, ale přitom bezpečně víte, že pokud byste vyzkoušeli o něco málo jiný postup, dokázali byste výsledný čas ještě o polovinu srazit. Já vám ale říkám, nedělejte to. A pokud to přesto udělat musíte, udělejte to na své vlastní triko. Po splnění všech kritérií návrhu již nesmíte pokračovat v práci, protože jinak byste celý projekt nikdy nedokončili.*

*Jedinou přípustnou výjimkou z tohoto pravidla jsou výzkumné a vývojové projekty; tyto typy projektů mají ale zcela jiné cíle, a tudíž i jiná kritéria. Návrhové kritérium výzkumného a vývojového kritéria nebude znít třeba „provést výpočet za méně než 10 minut“, ale spíše bude něco jako „zjistit optimální metodu pro výpočet toho a toho“. U takovéhoto kritéria nemůžeme nikdy s jistotou říci, že jsme je v jednom daném řešení splnili, protože nevíme, jestli je řešení právě optimální; můžete tedy projekt zkoumat donekonečna, či přesněji řečeno, dokud vám nedojde čas nebo peníze (podle toho, co dřívě).*

Ve fázi sestavování kritérií návrhu je velice důležité neupnout se na nějaký konkrétní typ návrhu či architektury. Můžete si být opravdu jisti, že z důvodu podpory rozšiřitelnosti systému použijete Microsoft Transaction Server, to je ale architekturní rozhodnutí, nikoli kritérium návrhu. Za kritérium návrhu je možné pokládat například podmínu, že systém musí být natolik rozšiřitelný, aby byl schopen podporovat x uživatelů.

Budete-li někdy na pochybách, pak si uvědomte, že podle kritérií návrhu se rozhoduje, jestli byl celý projekt úspěšně dokončen. V tomto případě se sami sebe zeptejte: „Jestliže budeme používat Microsoft Transaction Server, znamená to, že je systém dokončen?“ Možné to sice je, na druhé straně se ale dá prakticky určitě říci, že ze samotného zapojení Microsoft Transaction Serveru to nepoznáme. Jestliže ale odpovíme „ano“ najinou otázku, „pokud jsme schopní rozšíření na x uživatelů, jsme hotovi?“, může to třeba znamenat i to, že jsme vyhověli i dalším kritériím návrhu.

### **Přímo měřitelná kritéria**

O důležitosti rozpoznání objektivně měřitelných kritérií jsem již hovořila. Jestliže se vám podaří popsat měřitelná kritéria, vyplýne z nich zcela přirozeně řada kritérií návrhu. Pokud je například cílem systému zkrátit čas zpracování o 50 procent a doba zpracování v současném stavu je 10 minut, můžeme zcela zřejmě zformulovat kritérium návrhu, „umožnit dokončení zpracování za 5 minut nebo méně“.

Rozlišit měřitelné cíle od přímo měřitelných kritérií bývá někdy dosti obtížné. Nemyslím si ale, že by tento rozdíl byl bůhvjdak důležitý. A pokud nějakou podmínu zahrnete jak mezi cíle projektu, tak i do seznamu kritérií návrhu, neznamená to, že by na vás přišla nějaká specifikační policie a zatkla by vás. Jestliže se vám ale nepodaří některý měřitelný cíl podpořit jedním nebo více kritérií návrhu, je to určitě jakési varování.

Dejte si také pozor, abyste při stanovování kritérií návrhu nezabíedli do příliš zbytečných podrobností. V určitém systému může například platit, že nutnou podmínkou pro dokončení jistého procesu během jedné minuty je vykonání daného dotazu za méně než 10 sekund. To je ale opravdu pouze problém implementace a my se stále nacházíme ve stádiu návrhu, kdy o projektu nemáme tolik vědomostí, abychom mohli provádět seriózní rozhodnutí o implementaci.

### **Kritéria prostředí**

Drívou většinu systémových omezení, vyplývajících z prostředí, představuje stávající operační prostředí a starší, dosud provozované systémy, s nimiž musíme spolupracovat. Jen poměrně málokdy navrhujeme nový systém úplně s čistým stolem. Ve většině případů bude mít náš klient již nějaké existující hardwarové a softwarové prostředí, přičemž od nového systému očekává, že v něm bude fungovat.

Další důležitý zdroj kritérií systémového prostředí souvisí pouze s databázovými systémy: je to předpokládaný objem zpracovávaných dat. Kdysi, když jsem začínala jako nezávislý konzultant, jsem si přivodila docela pěknou ostudu v souvislosti s nabídkou systému sledování prodeje v regionální pobočce jistého obchodu s počítačovým hardwarem. Po úvodním rozboru požadavků jsem totiž předložila nabídku malého systému, který by se dal napsat v Microsoft Accessu 2.0, pročež mi bylo taktně naznačeno, že klient hodlá sledovat prodej za celou společnost, která má zhruba 500 regionálních poboček s objemem prodeje v desítkách miliónů dolarů – což je zcela zřejmě

daleko nad možnosti Microsoft Accessu. Já jsem se totiž mylně domnívala, že bude stačit sledovat pouze prodej v jedné pobočce. Ouvej. (Nemusím vám zdůrazňovat, že tu zakázku jsem samozřejmě nedostala.)

V souvislosti s objemy dat je třeba se zaměřit na dvě otázky. První z nich představuje hrubý, absolutní objem dat v daném okamžiku a druhým faktorem je předpokládaný růst. Knihovna má například několik miliónů svazků, ale denně do svých sbírek přidává jen několik málo záznamů. V systému pro práci s objednávkami se oproti tomu přidává denně několik stovek záznamů, po dokončení daného prodejního případu se ale příslušné záznamy odsunou do archivu, takže absolutní počet záznamů nikdy nepřekročí tisícovku. To jsou zřejmě dvě zcela rozdílná schémata objemu dat, která tím pádem vyžadují i rozdílné strategie při návrhu.

Podpora očekávaného objemu dat je jednou z mála situací, kdy se dá při návrhu ospravedlnit jisté předimenzování systému. Obecně se dá vyslovit následující pravidlo: plánujte vždy nejméně o 10 procent vyšší kapacitu, než kolik je nejvyšší údaj vyslovený samotným klientem. U menších systémů doporučuji tuto kapacitu navíc posílit na 20 až 25 procent. Při větších objemech dat to již není až tak velký problém. Dobře navržený systém architektury klient/server dokáže podporovat 100 000 záznamů stejně dobře, jako jich podporuje 10 000. Systém postavený na Microsoft Accessu, který běží v lokální síti LAN a který byl původně navržen pro několik tisíc záznamů, se však již stěží podaří rozšířit na milióny záznamů, ať je postaven sebelépe.

Dalším důležitým primárním zdrojem kritérií systémového prostředí je počet uživatelů, které musí systém podporovat. Ve většině systémů je definována více než jedna kategorie uživatelů, přičemž přesné požadavky musíme stanovit u každé z nich. V systému pro zpracování objednávek bude zcela zřejmě růžková skupina uživatelů pořizovat objednávky. Druhá skupina uživatelů se bude pravděpodobně dotazovat na stav objednávek a třetí skupina může z celé databáze vytvářet třeba různé sestavy. Každá z těchto tří skupin potřebuje od systému jiný typ podpory, který musíme tím pádem popsat vždy v samostatných kritériích návrhu.

Při stanovování kritérií musíte dále správně rozlišovat mezi uživateli, kteří jsou k systému pouze připojeni, a uživateli, kteří se systémem skutečně pracují. V databázovém stroji Jet platí například omezení, podle něhož smí být v každém libovolném okamžiku k databázi připojeno nejvíce 255 uživatelů. To znamená, že až 255 lidí může mít naši databázi současně otevřenou. Zároveň to ovšem neznamená, že by těchto 255 lidí mohli databázi současně aktualizovat.

## **Obecné strategie návrhu**

Některé cíle projektů se nedají příliš snadno převést do podoby jednoduchých číselných měřitelných veličin. Jedním z cílů projektu může být například „zlepšit přesnost pořizování dat“. Taková podmínka patří zcela zřejmě k cílům, které se velice obtížně kvantifikují. Náklady na zjištění počtu chyb vzniklých při pořizování dat nejspíše v této situaci převyšší výhodu stanovení kritéria, kterým změříme náš úspěch.

To ale neznamená, že bychom měli cíle uvedeného typu zcela ignorovat; je třeba je stanovit jako strategie návrhu, nikoli jako měřitelná kritéria. V našem příkladu může být kritériem návrhu „kdekoliv to je možné, zlepšit přesnost pořizování dat pomocí seznamů, ze kterých bude uživatel vybírat platnou hodnotu“, nebo třeba „snížit výskyt úvěrových výjimek pomocí implementace vhodných kontrol úvěrů, které se budou provádět ještě před přijetím faktury do systému“.

Podobně jako u měřitelných kritérií návrhu, ale ani zde není dobré jít příliš dopodrobna. Zatím ještě neprovádíme návrh systému, nýbrž pouze definujeme kritéria, podle kterých budeme posuzovat jeho úspěch. Ve výše uvedených příkladech jsme něco dělali „kdekoli to je možné“ a prováděli jsme „vhodné kontroly úvěrů“; konkrétní náplň těchto podmínek odložíme až do pozdějších fází návrhu projektu, když již budeme lépe chápout požadavky systému.

Je třeba se ovšem také vyhýbat různým „emotivním“ kritériím. Taková věta, „systém musí být uživatelsky přátelský“, zní docela přesvědčivě a vypadá dobře – konec konců sotva kdoby rád pracoval se systémem, který by byl uživatelsky *nepřátelský* či zlý. Jako kritérium je ale k ničemu. Zjistit, jestli systém naplňuje kritérium „systém musí vyhovovat zásadám Windows pro návrh softwaru, *Windows Interface Guidelines for Software Design*“, je docela dobré možné. Jestli je ale určitý systém uživatelsky přátelský, to je až příliš často věcí názoru a věcí určité debaty. A smyslem kritérií návrhu je mimo jiné vymýtit emotivní spory, a ne v nich přilévat olej do ohně.

## Určení záběru systému

Jakmile bezpečně víte, proč vlastně daný systém implementujete, máte již dobrý základ pro rozhodování, které funkce rozumně spadají do záběru systému a které nikoli. Podobně jako kritéria návrhu by přitom i jednotlivé funkce v dosahu (záběru) systému měly přímo podporovat stanovený cíl.

Řekněme například, že jedním z cílů projektu je zvýšení efektivity procesu zpracování objednávek prodeje. Tisk faktur tento cíl přímo podporuje a zcela jasně spadá do záběru navrhovaného systému. Sestavení katalogu výrobků však již na druhé straně nás cíl přímo nepodporuje, a proto leží mimo záběr systému. To přitom platí i v případě, že katalog vytvářejí stejně lidé jako faktury a že třeba se systémem pro pořizování objednávek využívají významné množství společných dat.

V těchto situacích má někdy smysl opravit definici cílů systému. Systém z našeho posledního příkladu by se tak například mohl z pouhého „Pořizování objednávek“ změnit na systém „Podpora prodeje“ a podmínka „Vytvářet katalogy výrobků“ by mohla být jedním z jeho cílů. Pomocí úpravy formulace cílů můžeme také vhodně ověřit, nakolik úplnou definici cílů se nám na začátku podařilo sestavit.

Tento postup v sobě skrývá ale také jistá nebezpečí. Definovat cíle nesmíme s pomocí požadovaného záběru systému. To je něco jako zapřáhat vůz před koně. Ze své vlastní zkušenosti vím, jak lákavé může být rozšířit záběr systému, aby se do něj vešly nějaké snadné či jinak zajímavé funkce. Vím také, jak trapné je, když vám uživatel položí otázku: „A proč bych to vlastně měl chtít?“ a vy mu na ni nedokážete uspokojivě odpovědět.

Pravidlo, podle něhož se funkce, která přímo nepodporuje žádný cíl, nachází mimo záběr systému, má smysl porušit ve dvou specifických situacích. První případ vyjadřuje situace, kdy má daná funkce pro uživatele určitou významnou hodnotu, která zcela zřetelně převáží významnou hodnotu spojenou s implementací. Dobrým příkladem může být třeba výše uvedený cíl vytvoření katalogu výrobků.

V tomto případě je zřejmě vhodnější a bezpečnější nesnažit se rozšiřovat cíle systému, ale zahrnout příslušné funkce pod obecnou kategorii záběru „Funkce s přidanou hodnotou“. Z této charakteristiky je jasné, že daná funkce není naprostě nezbytná, takže pokud někdy později zjistíme,

že její implementace stojí mnohem více, než se na první pohled zdálo, případně pokud nám prostě dojde čas nebo peníze, můžeme ji s klidným svědomím vyškrtnout.

Druhá situace, ve které je třeba zahrnout do systému funkce, které přímo nepodporují žádný jeho cíl, je, pokud klient na implementaci dané funkce trvá. Jako analytikovi je vám třeba úplně jasné, že vytvoření telefonního seznamu zaměstnanců nemá naprosto žádnou souvislost se zpracováním objednávek, jestliže si jej ale náš klient žádá, má ho mít. Můžete při diskusi s ním poukázat, že tato funkce nepodporuje žádné ze zadaných cílů systému, vašim úkolem je ale uspokojit požadavky klienta, a ne je definovat a vnucovat mu je.

### **Analýza nákladů a přínosů**

Nad stanovenými funkcemi má někdy smysl provést analýzu nákladů a přínosů. To platí zejména u systémů, které se skládají z několika různých komponent. Pořadí, v jakém budeme jednotlivé komponenty implementovat, určíme s pomocí relativního poměru nákladů ku přínosům. Jestliže uvažujeme o rozšíření záhlbu systému a o implementaci funkcí s přidanou hodnotou, pak může analýza nákladů a přínosů sloužit také jako kontrola reálnosti požadavků.

Mezi jinak sobě rovnými komponentami je třeba implementovat jako první ty komponenty, které mají nejvyšší přínosy oproti vynaloženým nákladům. Touto strategií dosáhneme nejlépe kýženého „rychlého zisku“, takže systém začne co nejdříve přinášet ovoce a začne se vyplácet. To platí zejména pro dlouhodobé projekty. Jestliže dokážeme základní funkce implementovat za poměrně krátkou dobu, dostaneme solidní základ pro hodnocení dalšího vývoje a snížíme tím riziko neočekávaných změn v podnikovém prostředí, se kterými by systém mohl být později nepoužitelný.

Pokud se systém dokáže začít vyplácet v relativně raných fázích vývoje, může se například otevřít prostor i pro financování nějakých těch „bezvadných a zajímavých, nikoli však nezbytně nutných“ funkcí, které jsme byli nutenci při definici záhlbu systému nemilosrdně vyškrtnout. Čas od času nastane ale pochopitelně i opačná situace. Klient tak třeba zjistí, že nějaké částečně implementované funkce již dostatečně naplňují jeho cíle a vývoj dalších komponent odsune někam do nekonečna.

Provedení formální analýzy nákladů a přínosů podléhá samozřejmě pravidlu o nákladech na přípustný omyl, které převyšují náklady na zjištění odpovědi. Analýzy nákladů a přínosů nebývají nikterak obtížné, občas jsou ale časově náročné; je přitom zřejmé, že nemá smysl strávit dva dny analýzou systému, který pak napišeme za jediný den. V řadě situací je proto více než dostatečná i neformální, intuitivní analýza.

Analýza nákladů a přínosů se dá provádět mnoha různými způsoby, její základní princip je však přesto velice jednoduchý. Odhadované přínosy funkce vydělíme odhadovanými náklady na implementaci dané funkce; tím dostaneme jistou číselnou hodnotu. Čím vyšší je tato hodnota, tím vyšší relativní hodnotu má komponenta ve srovnání vůči ostatním komponentám.

### **Poznámka**

*Připomínám, že zde hovoříme výhradně o odhadovaných nákladech a o odhadovaných přínosech. Kolik funkce skutečně stojí, to s jistotou můžeme říci až po její implementaci; její přínosy pak dokážeme přesně ocenit až po určité době provozu systému.*

Jistá základnost analýzy nákladů a přínosů spočívá v určení společných měrných jednotek. Veškeré náklady se totiž musí stanovovat ve stejných jednotkách a podobně i veškeré přínosy se musí také stanovovat ve stejných jednotkách, i když jednotky použité pro měření nákladů a výnosů nemusí být nutně stejné. Náklady na implementaci můžeme například vyjádřit časem, zatímco hodnotu přínosů vyčíslíme v peněžních jednotkách (například v korunách). Výsledný poměr porovnáváme s jinými hodnotami, které se vypočetly také nad stejnými jednotkami, takže porovnání je skutečně v pořádku.

Odhadované náklady na vytvoření systému či komponenty můžeme obvykle měřit časem nebo penězi, tedy buďto v hodinách, nebo v korunách; v podnikatelském prostředí se skutečně hodnoty mezi těmito dvěma veličinami naprosto normálně převádí. Protože ale význam obou těchto pojmu je poněkud široký, je někdy vhodné vyjadřovat náklady pomocí nějaké odvozené hodnoty, například v „jednotkách práce“. Takto zabráníme možné záměně analýzy nákladů a přínosů za cenovou nabídku či plán implementace.

Stanovit společnou měrnou jednotku výnosů může být více problematické. Dejme tomu, že podle našich odhadů dosáhneme automatizací daného pracovního procesu výsledného zvýšení efektivity o 20 procent a snížení chybosti o 50 procent. Zvýšení efektivity práce o 20 procent můžeme poměrně snadno převést na hodiny ušetřeného času a tím pádem i na peněžní vyjádření (je-li to nutné). Vyjádřit pomocí jedné hodnoty i zlepšenou přesnost však již tak snadné nebude. Můžeme například odhadnout náklady na vyhledání a odstranění chyb (opět v časových nebo peněžních jednotkách). Takto ale nedokážeme odpovědět na jiné neměřitelné, ale přesto také velice reálné přínosy, jako je například zlepšení dobrého jména společnosti, které vyplývá ze správného zpracování objednávky zákazníka.

V těchto situacích můžeme využít více odhadů přínosů. Dejme tomu, že budeme přínosy odhadovat v jednotkách „uspořené koruny“, „utržené koruny“ a „neměřitelné výhody“. Poté vypočteme tři poměry nákladů a přínosů, pro každou hodnotu jeden. Takto se ovšem samozřejmě mírně komplikuje porovnání výsledných hodnot – pro vás samotné i pro klienta může být obtížné rozhodnout, jestli má mít funkce s hodnotami  $3/6/2$  vyšší prioritu než třeba funkce s hodnotami  $6/2/3$ . V takovémto případě je třeba hodnoty určitým způsobem normalizovat a odvodit z nich jediný odhad poměru nákladů a přínosů.

Hodnoty v každé jednotlivé kategorii přínosů můžeme například sečít – pokud jsou ovšem všechny zhruba stejně důležitosti. Ano, přiznejme si, že je to trošku sčítání hruštiček a jablíček, ale v tomto okamžiku je přijatelné je prostě považovat za ovoce. Někdy je vhodnější vypočítat průměr. Já obvykle dělám obojí.

Pokud však jednotlivé kategorie nemají pro klientskou organizaci stejný význam – a tak je tomu nejčastěji – přiřadíme každé z kategorií určitou relativní hodnotu a tímto faktorem důležitosti vynásobíme hodnoty v příslušné kategorii. Budeme-li například dále uvažovat naše tři kategorie přínosů, můžeme se rozhodnout, že kategorie „uspořené koruny“ není až tak důležitá, zatímco „neměřitelné výhody“ jsou velice důležité a „utržené koruny“ jsou ještě dvakrát důležitější než „neměřitelné výhody“. Kategorii „uspořené koruny“ přiřadíme proto modifikátor 1, kategorii „neměřitelné výhody“ modifikátor 2 a kategorii „utržené koruny“ modifikátor 4. Výsledné údaje shrnuje následující tabulka.

	Utržené koruny (mod. 4)	Uspořené koruny (mod. 1)	Neměřitelné výhody (mod. 2)	Celkem	Vážený součet	Průměr	Vážený průměr
Funkce 1	3	6	2	11	22	3,6	7,3
Funkce 2	6	2	3	11	32	3,6	10,6

Podobně jako je často vhodnější přiřadit jednotlivým kategoriím relativní modifikátory, je dobré i odhadu přínosu stanovovat relativně, nikoli absolutně. Často je tak například velice obtížné říci, že funkce X má neměřitelné přínosy 3. Obvykle se ale dá odhadnout, že třeba funkce X bude mít pravděpodobně dvakrát vyšší neměřitelné přínosy než funkce Y a že funkce Y a funkce Z budou mít zhruba stejné neměřitelné přínosy.

Analýza nákladů a přínosů může být také velice užitečným nástrojem pro podchycení zamýšlených přínosů systému. Představuje jednoduchý způsob porovnání relativních hodnot různých komponent. Je to ale stále jen jistý nástroj, který je navíc postaven výhradně na odhadech; tyto odhadu pochopitelně nesmíte mylně považovat za absolutní hodnoty. I když má například funkce X poměr přínosu 12 a funkce Y má poměr přínosu 2, může přesto být vhodnejší (nebo dokonce nezbytně nutné) implementovat jako první funkci Y.

Výsledky analýzy nákladů a přínosů je třeba posuzovat ve spojení s dalšími faktory, jako jsou například závislosti systému. I tyto faktory musíme v průběhu vývoje, kdy postupně zpřesňujeme naše chápání systému, znova revidovat. Odhadu přehodnocujte vždy před započetím práce na každé komponentě. Zkušenosti se *skutečnými* náklady a přínosy předchozích komponent se mohou totiž promítnout do změn v odhadech budoucích komponent.

### Stručné shrnutí

V této kapitole jsme se zabývali činnostmi, které souvisí se správným pochopením systému na začátku projektu. Nejprve musíme stanovit cíle systému a poté je převést na měřitelná kritéria návrhu, ze kterých budeme moci rozhodnout o úspěchu nebo selhání projektu. Zároveň musíme popsat záběr systému – tedy stanovit hranice systému a říci, co bude a co nebude jeho součástí.

Všechny tyto činnosti představují jakýsi nultý krok; jsou to věci, které musíme udělat ještě *před* tím, než se pustíme do vlastního návrhu. V následující kapitole budeme důkladně rozebírat první krok procesu návrhu: je to definice pracovních procesů, které bude systém svými činnostmi podporovat.

# DEFINICE PRACOVNÍCH PROCESŮ

8

Řada databázových systémů je sice navržena s tím, že budou jednoduše sloužit k ukládání a načítání nějaké množiny dat, úkolem většiny z nich je ale napomáhat v provádění jedné nebo více činností. Tyto činnosti jsou *pracovní procesy*, které bude systém podporovat. Pracovní proces není přitom nic jiného než množina jednoho nebo více diskrétních úkonů, jež společně reprezentují jistou činnost, která má pro danou organizaci svůj smysl. Příkladem pracovního procesu mohou být například výrazy „Zpracuj objednávku prodeje“ nebo „Vyhledej telefonní číslo zákazníka“, i když oba se nacházejí na naprosto rozdílné úrovni složitosti.

*Úkon* (úkol) je diskrétní operace, tedy jeden krok v provádění daného pracovního procesu. Proces zpracování objednávky prodeje se může například skládat z úkonů „Zaznamenat objednávku prodeje“, „Ověřit úvěr zákazníka“, „Ověřit dostupnost zboží na skladě“ a „Odeslat objednané zboží“. Proces vyhledání telefonního čísla zákazníka se však oproti tomu bude nejspíše skládat jen z jediného úkonu, „Vyhledej záznam požadovaného zákazníka“.

Správně rozlišit mezi úkonom a činností (aktivitou) může být někdy obtížné; hranice mezi oběma pojmy není pevně daná a můžeme ji v podstatě vymezit jakkoli. Vymezení této hranice se do jisté míry podobá rozhodnutí, co v datovém modelu reprezentuje skalární hodnota; jediný atribut z jednoho modelu tak můžeme v jiném modelu rozložit do několika elementárních atributů. Činnost, kterou považujeme v jednom systému za jediný úkon, může ve druhém systému tvořit celý proces, který na nižší úrovni podrobnosti rozložíme do několika samostatných úkonů. A podobně jako u datového modelování musí i zde příslušné rozhodnutí vycházet ze sémantiky daného prostoru problému.

Některé systémy se pro analýzu pracovních procesů příliš dobře nehodí. Nástroje pro tvorbu ad hoc sestav například žádný konkrétní proces nepodporují, protože podporují určité typy činností. V těchto případech je vhodnější navrhnout uživatelské scénáře. O nich si něco povíme v závěru této kapitoly.

## Vymezení stávajících pracovních procesů

Definice záběru systému je ve skutečnosti prvním krokem analýzy podnikových procesů, protože právě z této definice určíme procesy, které potřebujeme analyzovat. Pořadí, v jakém budeme různé procesy v řešeném systému zkoumat, obvykle není důležité. I když hodláte podle plánu implementovat některé komponenty systému před jinými (jestliže tedy postupujete metodou implementálního vývoje), je vhodné před započetím vlastní implementace provést alespoň hrubou analýzu všech pracovních procesů, které bude systém podporovat. Takto totiž snadno zjistíte veškeré závislosti mezi procesy, které mohou ovlivnit pořadí, v jakém bude nutné komponenty implementovat.

### Hovory s uživateli

Po identifikaci pracovních procesů, které se nacházejí v požadovaném záběru systému, je dalším úkolem podchycení veškerých aktivit, jež momentálně slouží k jejich provádění. V tomto okamžiku ještě nemusíte podrobně zjišťovat, co představuje pouze úkon a co je snad samostatný pracovní proces, který vyžaduje další analýzu. Stačí, když si zavoláte někoho, kdo je vám schopen a ochoten říci: „Takže my dostaneme od prodejce takový dokument, a napřed se podíváme, jestli je správně vyplněný. Pokud je všechno v pořádku, jdeme do kartotéky zákazníku a …“ Pokládejte lidem hodně otázek, ptejte se na všechno a všechny odpovědi si zapisujte. Vyžádejte si také kopii všech formulářů a sestav, které se používají buďto jako vstup do procesu, nebo které jsou naopak jeho výsledkem. Vaším cílem je pochopit, co se zde děje; zatím se nesnažíme o analýzu procesu.

Mimochodem, řada lidí nazývá tuto fázi analýzy „rozhovory s uživatelem“. Osobně bych zde raději viděla slovo „diskuse“ nebo nějaký více neutrální pojem. Člověk si ani neuvědomí, nakolik počítáče nahánějí lidem strach a to i lidem, kteří s nimi již dnes pracují. Mnozí lidé se neustále obávají, že počítáče mají pouze nahradit jejich práci a „rozhovory s uživatelem“ mylně chápou asi takto: „Rozhodujeme se, kdo dostane padáka.“ To platí zejména ve velkých organizacích, kde nemůžeme hovořit se všemi a kde lidé ani neví, kdo přesně jste a jaký je váš úkol.

Pokud je to jen trochu možné (vždy tomu tak opravdu není), pokuste se raději hovořit s lidmi, kteří dané pracovní procesy přímo provádějí, než s jejich nadřízenými či vedoucími. Nadřízení mají totiž podle mých zkušeností velice často zkreslené představy a pracovní procesy pod svým vedením příliš idealizují. Zpovídání člověka, který danou činnost vykonává dnes a denně, je pro tyto účely mnohem vhodnější, protože vám dokáže říci i o různých běžných problémech a překážkách, s nimiž se neustále potýká. Samozřejmě že je nutné si popovídат i s nadřízenými, protože ti zase nejlépe chápou, *proč* se ten který úkon provádí a znají i různé souvislosti a řeknou vám, co by se stalo, kdyby se tyto úkony neprováděly nebo kdyby se prováděly nesprávně, a jaké by to mělo důsledky.

Při těchto rozhovorech se snažte vyhledávat v analyzovaných procesech různé výjimky. Jestliže například uživatel řekne: „Kontrolujeme úplnost vyplněné objednávky“, pak se nesmíte zapomenout zeptat také na to, co se stane v případě, že objednávka *není* úplná. Možná ji dotyčný pracovník prostě vrátí zpět; vy se ale přesto musíte zeptat, jestli se třeba nesnaží chybějící informace vyhledat sami. Takto můžete v navrhovaném systému usnadnit běh celého procesu. Ve skutečnosti je u každé činnosti, kterou uživatel provádí, nutné se zeptat, jaké mohou při jejím zpracování nastat chyby a co se v takovém případě dělá.

Nás zde ovšem zajímají především databázové systémy, takže musíme věnovat velkou pozornost zejména tomu, jaké údaje se v procesu využívají. S jakými informacemi daný proces pracuje? Odkud se tyto informace berou? Jakou mají podobu? Co se stane, když nejsou k dispozici nebo když nejsou ve správném tvaru? Z odpovědi na tyto a podobné otázky sestavíme hrubý materiál, který bude základem pro myšlenkový datový model, o němž budeme hovořit v následující kapitole.

Řada pracovních procesů se přitom skládá z úkonů, které provádějí různí lidé (jednotlivci). Vy tedy musíte při analýze zcela zřejmě hovořit pokud možno se všemi lidmi zapojenými do procesu. Toto doporučení platí ale také pro osoby, jejichž úkoly spočívají zvlášť mimo dosah systému. Vezměme například ještě jednou náš výše popsaný proces zpracování objednávky prodeje. Úkon „Odeslat objednané zboží“ může ve skutečnosti znamenat něco jako „Odeslat objednávku do expedičního oddělení“; co se děje v expedičním oddělení, to je již mimo rámec našeho systému. Přesto je vhodné pohovořit i s lidmi v expedici a ověřit si, jestli dostávají všechny informace, jež ke své práci potřebují, a jestli je dostávají ve formě, která je pro ně užitečná a smysluplná.

Podobně jestliže se v určitém okamžiku daného procesu vytiskne nějaká sestava, musíme vyhledat osobu, která je jejím příjemcem, a zjistit, co s ní dále dělá. Divili byste se, kolik různých páprů se v organizacích potuluje sem a tam fakticky bez žádného rozumného důvodu. (Možná to pro vás žádné překvapení není.) Anebo, což je častější případ, sestava má nějaký rozumný důvod a má svůj smysl, obsahuje ale nesprávné informace, případně obsahuje sice správné informace, ale v nevhodné podobě, takže se prostě někde jinde založí a nevyužívá se k tomu, k čemu byla původně určena.

## **Identifikace úkonů**

Po vykonání rozhovorů se všemi lidmi, kteří momentálně provádějí práce, jejichž provoz má námi navrhovaný systém podporovat, bychom již měli příslušným činnostem docela solidně rozumět. Dalším krokem je tedy uspořádat všechny takto získané informace do množiny úkonů. Klíčem je zde identifikace aplikačních (věcných) pravidel, která pro dané procesy musí platit.

Na začátku této kapitoly jsem definovala pojem „úkon“ jako nějakou diskrétní operaci. Nyní již můžeme přesněji říci, co v tomto kontextu znamená slovo „diskrétní“. Znamená totiž dvě věci: za prvé, operace má přesně definovaný začátek a konec a za druhé, před zahájením úkonu a po jeho dokončení je splněna platnost všech souvisejících aplikačních pravidel. Během zpracování úkonu se však přesto pravidla smí porušit – pochopitelně jen dočasně.

Aplikační pravidlo (věcné pravidlo, business pravidlo) zde není nic jiného, než jisté omezení, které vychází z oboru definovaného daným problémem; není to tedy například omezení odvozené od datového typu. Podmínka „Žádná objednávka nesmí mít datum odeslání rovno 36. dubna“ není tedy aplikačním pravidlem, protože je závislá pouze na oboru hodnot datumu. (A konec konců je to docela hloupě nadefinované pravidlo.) Podmínka „Žádná objednávka nesmí mít datum odeslání před datem objednávky“ již vychází z našeho prostoru problému; závisí totiž na způsobu, jakým organizace provádí svoji podnikatelskou činnost. Je to tedy, nebo by alespoň mohlo být, platné aplikační pravidlo. Pojem „aplikáční – business pravidlo“ se mimochodem používá i v případě, že daná organizace není ve své podstatě žádný podnikatelský subjekt. Databáze, která bude evidovat například sbírku starých šavlí, bude také podléhat aplikačním čili „business“ pravidlům.

Drtivá většina aplikačních pravidel souvisí se způsobem zpracování dat. Příklady aplikačních pravidel, která přímo souvisí s daty, jsou třeba podmínky „Poštovní směrovací číslo zákazníka nesmí být prázdné“ nebo „Datum vystavení faktury musí být pozdější nebo rovné datum odeslání“. Další pravidla, jako například „Jestliže zákazník s danou objednávkou překročí svůj úvěrový limit, je nutné vyžádat souhlas manažera prodeje“, nevedou přímo k omezení nějaké datové hodnoty, i když uplatnění takového pravidla je spuštěno také určitou datovou hodnotou.

Nebojte se; zjištění aplikačních pravidel, která se týkají určitého pracovního procesu, není tak obtížné, jak by se na první pohled mohlo zdát. Přesně k nim totiž směrují všechny ty otázky typu „K jaké chybě zde může dojít a co se stane, když k ní opravdu dojde?“ V tomto okamžiku se přitom ještě nemusíte nijak dopodrobna zajímat o přesnou podobu pravidel. Jejich přesná podoba spadá do sestavení myšlenkového datového modelu, k němuž se dostaneme později. Zde nám stačí jediné: seskupit různé identifikované činnosti takovým způsobem, abychom mohli mít jistotu, že aplikační pravidla mohou být na obou stranách každé operace splněna.

Podívejme se na malý příklad. Řekněme, že náš seznam činností, které se provádějí při zpracování objednávky prodeje, obsahuje následující úkony:

1. Zkontrolovat úplnost vyplněné objednávky prodeje.
2. Vzít si kartu s údaji o zákazníkovi, jedná-li se o existujícího zákazníka.
3. Zaznamenat informace o odeslání zboží.
4. Vyplnit podrobné informace rozpisu objednávky.
5. Novému zákazníku přiřadit číslo zákazníka.
6. Ověřit dostupnost zboží na skladě.
7. Ověřit úvěrový limit zákazníka.
8. Připravit objednané zboží.
9. Zahálit zboží.
10. Připravit potřebné přepravní dokumenty.

První věci, které si na tomto seznamu asi všimnete, je zdánlivě náhodné pořadí jednotlivých operací. To je v pořádku. Na tomto místě se pouze snažíme pochopit, jak se věci momentálně provádějí. Výslednou podobu procesu „vyčistíme“ později. Některé body obsahují také podle všeho jinou úroveň podrobnosti. I na to se za chvíličku podíváme. Zatím ale musíme jen identifikovat (popsat) úkony.

Bod číslo 1, „Zkontrolovat úplnost vyplněné objednávky prodeje“, má zcela jasné, diskrétní začátek i konec. Operace se provádí v okamžiku přijetí nové objednávky a ukončí se ve chvíli, kdy celý dokument prošel požadovanou kontrolou. Na začátku procesu můžeme předpokládat, že je zaručena platnost všech aplikačních pravidel, takže v tom není žádný problém. Jestliže těmto aplikačním pravidlům nevyhovuje původně zadaný dokument s objednávkou prodeje, jednoduše se zamítne. Víme tedy, že pravidla budou platit i po dokončení úkonu, kdy zpracování procesu přejde k dalšímu kroku. Bod 1 tedy odpovídá definici úkonu.

K bodu číslo 2, „Vzít si kartu s údaji o zákazníkovi“, náleží pouze jediné aplikační pravidlo: osoba, která vyžaduje kartu s údaji o zákazníkovi, musí k ní mít přístup. Zde můžeme předpokládat, že k záznamům o zákaznících má přístup každý, kdo provádí tento pracovní proces, takže jako kritérium nemá toto pravidlo smysl. Operace začíná po zkontolování objednávky prodeje; protože ale s kartou údajů o zákazníkovi může dotyčný pracovat i v dalších operacích, nemá diskrétní konec. Není to tedy úkon; je to jeden z kroků v úkonu.

Karta (záznam) s údaji o zákazníkovi se ve skutečnosti využívá v bodu 3 („Zaznamenat informace o odeslání zboží“) a v bodu 5 („Novému zákazníku přiřadit číslo zákazníka“); odtud již na první pohled vidíme, že oba body náleží ke stejnemu úkonu. Skutečně je zřejmé, že body 2 až 5 můžeme seskupit do jediného úkonu, který nazveme třeba „Záznam objednávky“. Součástí záznamu objednávky je v tomto případě zcela jasně i bod číslo 4, „Vyplnit podrobné informace rozpisu objednávky“. Nebudte ale překvapeni, když zjistíte, že některé úkony se provádějí současně. V manuálním systému může někdo skutečně docela snadno vyplňovat třeba dva formuláře zároveň. Na tom, že oba formuláře logicky patří k různým, samostatným úkonům, zde nezáleží.

Nyní jsme tedy vytvořili nový úkon, který se skládá ze čtyř diskrétních kroků. Začíná v okamžiku, kdy byla provedena kontrola úplnosti původního dokumentu a končí se zaznamenáním celé objednávky. Tady ale nastává určitý problém. Náš klient nechce dovolit zákazníkům podávat takové objednávky, které převyšují jejich úvěrový limit, kontrola úvěrového limitu se ale provádí až v bodu 7, „Ověřit úvěrový limit zákazníka“, tedy *poté*, co uživatel zkontoval, jestli je objednané zboží na skladě. Dokud se však nepotvrdí, že objednávka nepřekračuje úvěrový limit zákazníka, nemůžeme s jistotou říci, že aplikační pravidla platí na obou stranách úkonu. Součástí úkonu „Záznam objednávky“ musí být tudíž i bod 7.

Bod číslo 6, „Ověřit dostupnost zboží na skladě“, není ale logicky vzato součástí záznamu objednávky. Tento bod vyjadřuje ve skutečnosti jistý diskrétní úkon, který začíná po dokončení úkonu „Záznam objednávky“ a končí potvrzením, že je potřebné objednané zboží opravdu na skladě. Bod číslo 6 je tedy samostatným úkonem. Neříkejte, že jsem vás nevarovala.

V takových situacích si nezapomeňte ověřit, *proč* se jednotlivé úkony provádějí zdánlivě mimo pořadí. Nejčastěji je to věc jistého pohodlí, někdy jsou ale tyto nekonzistence důsledkem jistých operačních omezení, které mají významný dopad na procesy našeho systému.

Jestliže se v tomto případě ověřuje dostupnost zboží na skladě jako první jen z toho důvodu, že je to řekněme logisticky snazší, pak můžeme změnit pořadí operací. Zrovna tak ale můžete zjistit, že mezi procesem pořizování objednávek a provozním procesem skladu či výrobním procesem dochází k nějaké interakci, kvůli které se množství zboží na skladě musí nutně ověřovat okamžitě; takovou interakci již musíme v systému odpovídajícím způsobem zohlednit.

Všimněte si, že zatím nemusíme nijak podrobně popisovat aplikační pravidla, která platí pro určitý úkon. Stačí, když do jisté rozumné míry bezpečně víme, že ať budou pravidla jakákoli, zůstane jejich platnost zachována. V tomto případě stačí tedy předpokládat, že se objednávka, která nevyhovuje všem definovaným pravidlům, prostě zamítne; podrobně tedy všechna pravidla zatím zkoumat nemusíme.

Pokud bychom později přišli na to, že se objednávka, která není kryta dostatečným množstvím zbožím na skladě, nesmí vůbec přijmout, musel by se úkon „Ověřit dostupnost zboží na skladě“

stát skutečně jedním z kroků již popsaného úkonu „Záznam objednávky“. Jak uvidíme v následující části textu, přesunout činnosti mezi různými úkony a mezi různými logickými úrovněmi je docela jednoduché. Podstatné je zde tedy pouze správně identifikovat jednotlivé činnosti.

Analýza posledních třech bodů našeho seznamu – tedy bodů 8, 9 a 10 – závisí na předpokládaném záběru systému. Jestliže má být součástí systému i odeslání objednaných výrobků, pak je musíme podrobit odpovídajícímu zkoumání. Pokud ale, což bude asi pravděpodobnější, záběr systému končí s předáním hotové objednávky do expedičního oddělení, můžeme všechny tyto tři body shrnout do jediného úkonu, „Odeslat objednávku do expedičního oddělení“.

My ale naštěstí víme, že i po tomto okamžiku bude probíhat několik dalších činností; tyto informace pochopitelně není důvod zahazovat. Bližším zkoumáním zjistíme, že každý z těchto tří zbyvajících bodů představuje samostatný úkon. Poslední bod seznamu, „Připravit potřebné přepravní dokumenty“, může dokonce představovat celý další pracovní proces. Všechny tyto úkony však již nesporně leží mimo předpokládaný záběr projektu, takže je do naší analýzy uvedeme jen jako kroky; takto nezahodíme, co již víme, jinak se jimi ale zabývat nebudeme. Náš upravený seznam úkonů vypadá tedy takto:

Úkon 1: Zkontrolovat úplnost vyplněné objednávky prodeje

Úkon 2: Záznam objednávky

Krok 1: Vzít si kartu s údaji o zákazníkovi

Krok 2: Zaznamenat informace o odeslání zboží

Krok 3: Vyplnit podrobné informace rozpisu objednávky

Krok 4: Novému zákazníku přiřadit číslo zákazníka

Krok 5: Ověřit úvěrový limit zákazníka

Úkon 3: Ověřit dostupnost zboží na skladě

Úkon 4: Odeslat objednávku do expedičního oddělení

Krok 1: Připravit objednané zboží

Krok 2: Zabalit zboží

Krok 3: Připravit potřebné přepravní dokumenty

Během procesu uspořádání činností do úkonů a kroků přijdete určitě na spoustu věcí, kterým nebudete rozumět tak dobře, jak jste si původně mysleli. V takovém případě není nic snazšího, než se vrátit k uživateli a vyžádat si od něj další vysvětlení. Procesy je ale každopádně třeba revidovat ve spolupráci s uživateli. Často uživateli stačí, když vidí daný proces popsaný na papíře, a hned od něj dostanete další informace – třeba si vzpomene na kroky, o nichž se zapomněl zmínit, nebo vám řekne o výjimkách, na které jste se vy sami zapomněli zeptat.

## **Analýza pracovních procesů**

Nyní již tedy máme docela jasnou představu o tom, jak se všechno dělá za současného stavu. Dále musíme prozkoumat pracovní procesy a najít v nich možná další vylepšení. Jak jsem již řekla, vět-

šina organizací má dosti velkou setrvačnost. To platí jak pro podnikové procesy, tak i pro práci s dokumenty. Nasazení nového počítačového systému pak představuje vynikající příležitost k „prořezání suchých větví“.

Jedno staré pravidlo efektivity říká, že bychom žádný kus papíru neměli zpracovávat více než jednou. (Často se také říká, jednou pořízená informace se nesmí pořizovat znova a musí se dále využívat. Pozn. překl.) Podobně jako řada ostatních není ani toto pravidlo vždy úplně praktické, není ale neobvyklé, když v analýze přijdeme na to, že se změnou uspořádání různých úkonů dají pracovní procesy podstatně zjednodušit, takže si jednotlivé osoby či procesy nemusí různé části práce přehazovat sem a tam. Dokud jsme do své práce přímo ponořeni, nedokážeme z jistého důvodu dost dobře jednotlivé pracovní procesy popsat, jako třeba „nejprve udělám A a předám to tobě, ty uděláš B, vrátíš mi to zpátky a já udělám C“, jakmile ale veškeré činnosti popíšeme na papíře, vše se rázem vyjasní.

Při takové analýze musíme mít velice dobře jasno ohledně vzájemné závislosti úkonů. V každém určitém procesu najdeme některé úkony, které jsou závislé na jiných; to znamená, že takové úkony se musí provádět v jistém konkrétním pořadí. Objednávku nemůžeme například předat do Expedice, dokud ji nepořídíme (nezaznamenáme do systému). Pořadí ostatních, nezávislých úkonů již může být libovolné. Nezáleží například na tom, jestli číslo nového zákazníka přiřadíme před nebo po zadání podrobných informací o odeslání zboží.

Velice důležité je podívat se také na závislosti mezi daty. Některé úkony jsou odpovědné za vytváření informací (jako jsou například čísla zákazníků), které se využívají v jiných úkonech. Jeden můj klient používal například podobný proces zpracování objednávek prodeje, jaký jsme si popsal v předcházející části textu, avšak číslo zákazníka a počáteční úvěrový limit neurčovalo oddělení prodeje, které bylo odpovědné za celé zpracování objednávek, nýbrž účetní oddělení.

Původně tedy pracovní procesy tohoto klienta vypadaly následovně:

**Úkon 1: Zkontrolovat úplnost vyplněné objednávky prodeje**

**Úkon 2: Záznam objednávky**

Krok 1: Vzít si kartu s údaji o zákazníkovi

Krok 2: Zaznamenat informace o odeslání zboží

Krok 3: Vyplnit podrobné informace rozpisu objednávky

**Úkon 3: Provést kontrolu úvěrového rámce zákazníka**

Krok 1: Ověřit reference zákazníka

Krok 2: Ověřit úvěrový limit zákazníka

Krok 3: Novému zákazníku přiřadit číslo zákazníka

**Úkon 4: Zkompletovat objednávku**

Krok 1: Ověřit dostupnost zboží na skladě

Krok 2: Odeslat objednávku do expedičního oddělení

Úkon 3 zde provádělo účetní oddělení, které pak potvrzenou objednávku vrátilo zpět do oddělení prodeje jen v případě, že kontroly úvěrového rámce dopadly v pořádku. Při tomto postupu nastával ale samozřejmě problém, že počáteční pořízení dat již v této fázi dávno proběhlo. Z toho vyplývá nejen to, že v případě zamítnutí objednávky přichází provedená práce vniveč, ale také to znamená určitou další práci navíc, protože neplatné objednávky se musí periodicky odstraňovat. Pracovníci pořizování dat byli tím pádem zbytečně zapojeni také do sporů mezi účetním oddělením a obchodními zástupci, kteří pochopitelně chtěli podané objednávky uspokojit (a vyinkasovat odpovídající provizi). Změnou pořadí úkonů, kdy se kontrola úvěrového rámce provedla ještě *před* předáním objednávky pořizovačům dat, se jednak odstranila neefektivita, jednak zmizely i nepříjemné spory.

Kromě prozkoumání závislostí mezi jednotlivými úkony je třeba prověřit také úkony, které se již nadále nebudou provádět. Odhalit tyto úkony se mállokdy podaří hned na první pohled; jestliže budeme ale pozorně sledovat tok dat skrče jednotlivé procesy, najdeme někdy datové položky, které byly vytvořeny pro jiný úkon či proces, který je již ve skutečnosti nepotřebuje. Méně často se nám podaří odhalit celé zbytečné úkony nebo i jen kroky v rámci některého z nich. Většina lidí se dokáže zbytečné a obtížné práci docela úspěšně bránit; celou situaci však mohou zakrývat různé interakce mezi pracovními procesy. Nejobvyklejším příkladem těchto zbytečných operací je tvorba sestav, které již ničemu neslouží.

To, co jsme si teď řekli, samozřejmě nelze chápat jako ospravedlnění rozsáhlých změn v podnikatelské praxi vašeho klienta. Zrovna tak vám radím: jestliže vás požádá pratetička Gertruda o převedení háčkovacích vzorů na počítač, nesnažte se jí vnucovat, aby si také změnila způsob jejich uspořádání. Ve skutečnosti se přitom neefektivity v pracovních procesech podaří najít jen dosti zřídka. Vaším úkolem je ale pomoci klientům, aby mohli dělat svoji práci lépe; součástí tohoto úkolu je i kontrola pracovních procesů.

## Dokumentace pracovních procesů

Podobně jako pro ostatní části procesu návrhu platí i pro analýzu pracovních procesů, že množství času stráveného s analýzou a s formální dokumentací by mělo být úměrné složitosti systému. K analýze jednoduchého systému pro sledování jmen a telefonních čísel lidí bude například úplně stačit hodinový rozhovor a nějaké stručné poznámky pro pozdější práci.

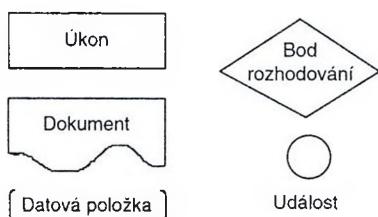
I pro nejjednodušší projekty však přesto doporučuji uskutečnit alespoň dvě setkání – ledaže jste vy sami návrhářem i klientem v jedné osobě. Smyslem druhého setkání je ověřit si, jestli správně rozumíte potřebám klienta, a potvrdit faktickou správnost všeho, čemu jste porozuměli a co jste naplánovali udělat.

Složitější projekty si žádají třeba i několik týdnů rozhovorů s desítkami různých lidí, čemuž odpovídá také složitá a formální dokumentace. K dokumentaci jednoduchých procesů stačí třeba strukturovaný seznam úkonů a jejich kroků, jaký jsme třeba v této kapitole sestavili u předchozího příkladu. U složitějších procesů doporučuji nakreslit si názorný obrázek.

Pro dokumentaci vztahů mezi daty existuje v celém oboru uznaný standard, a sice diagramy entit a vztahů (Entity Relationship Diagrams, E/R diagramy). Diagramy pracovních procesů již nejsou natolik konzistentní. Používané metodologie tvorby diagramů bývají obvykle úzce spjaty s kon-

krétními analytickými postupy. Proto pokud některou z těchto metodologií dobře znáte a jste na ni zvyklí, není důvod cokoliv měnit. Smyslem je zde pochopit jisté informace a umět je vyjádřit. Užitečnými a použitelnými nástroji jsou zde rozhodně diagramy toku dat a diagramy kvality procesů. Prosazovat nějaké speciální postupy pro tvorbu diagramů mi vždy připadaly poněkud hloupé, i když lidé to často dělají.

Jestliže zde nějaká formální, standardní technika fakticky chybí, máme prostor k vytvoření své vlastní. Budeme k tomu potřebovat celkem pět symbolů, které budou po řadě vyjadřovat úkon, dokument, datovou položku, bod rozhodování a událost, jako je například počátek a konec (počáteční a koncový bod) úkonu. Jednotlivé symboly, které používám já sama při své práci, ukazuje obrázek 8-1.

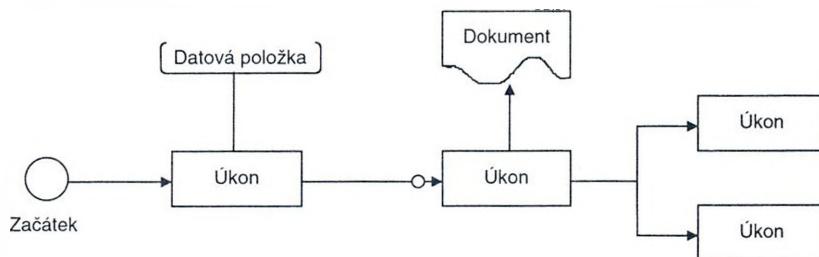


Obrázek 8-1 Symboly pracovních procesů

Jestliže je k dokončení určitého úkonu potřeba jen několik málo kroků, zapíši je přímo do bloku Úkon. Pokud je ale těchto kroků více, takže by se do symbolů pohodlně nevešly, rozvinu každý takový úkon do samostatného diagramu, v němž blok typu Úkon reprezentuje jeden krok. Někdy pro lepší srozumitelnost použiji stínování nebo tučný obrys, pomocí něhož naznačím, že daný úkon provádí nějaká externí skupina, jako tomu bylo u účetního oddělení, které provádělo kontrolu úvěrového rámce pro nového zákazníka.

Symbol datové položky může reprezentovat buďto jediný atribut, jako je například číslo zákazníka, nebo celou entitu, jako je například zákazník se všemi svými údaji. Pokud danou datovou položku přímo vytváří určitý úkon, můžeme tuto skutečnost opět vyznačit pomocí stínování nebo tučného obrysu. Někteří analytici také ve svých diagramech naznačují, že úkon určitou datovou položku „spotřebovává“; to znamená, že tuto položku využívá jen tento úkon, ale již žádné další úkony stejného procesu. Upřímně řečeno, málokdy mám pocit, že by taková informace byla k něčemu dobrá, takže raději diagramy zbytečně nepřeplňuji.

Jakmile se rozhodnete pro určité symboly, které budete používat (důrazně vám doporučuji vybrat si spíše symboly, které se snadno nakreslí, než složité symboly s určitým vnitřním významem), potřebujete je ještě nějakým způsobem usporádat. Já používám k vyznačení závislostí šipky; rozvětvená čára mi indikuje, že se příslušné úkony dají provádět v libovolném pořadí. Prázdný kroužek na čáře znamená, že se jedná o nepovinný (volitelný) úkon, podobně jako je tomu i u E/R diagramů. Všechna tato propojení ukazuje názorně obrázek 8-2.



Obrázek 8-2 Propojení jednotlivých pracovních procesů

Jestliže postupem času přijdete na to, že vaše konkrétní procesy jsou příliš složité, než aby se daly rozumně zachytit pomocí této jednoduché techniky, pak vám doporučuji podívat se po některé z propracovanějších metodologií, jež popisuje kterákoli dobrá učebnice systémové analýzy a návrhu. Několik takových učebnic je uvedeno v seznamu doporučené literatury.

## Uživatelské scénáře

Alternativou k formální analýze pracovních procesů je identifikace takzvaných uživatelských scénářů. Takový uživatelský scénář se skládá ze dvou komponent: první tvoří množina jednoho nebo více uživatelských profilů, které popisují různé typy uživatelů navrhovaného systému, a druhou jsou scénáře používání pro každý uživatelský profil; scénářem je přitom slovní popis způsobu, jakým bude uživatel s navrhovaným systémem pracovat – zde tedy najdeme činnosti, které má uživatel podle očekávání v novém systému provádět.

Uživatelské scénáře mohou sice zachytit stejně informace jako při analýze pracovních procesů, obvykle se ale mají snahu zaměřovat spíše na způsob, jakým bude s navrhovaným systémem pracovat jeho uživatel, než na konkrétní kroky určité transakce. Z důvodu tohoto silného zaměření na spolupráci s uživatelem bývá někdy obtížné navrhnout takový uživatelský scénář, který by nepředjímal nějakou konkrétní podobu uživatelského rozhraní.

Smyslem uživatelského scénáře je ale soustředit se na cíle a očekávání uživatele systému; jako takový je uživatelský scénář tudíž užitečný zejména pro systémy, které musí podporovat velké množství nahodilých (ad hoc prováděných) činností. Analytik může s jeho pomocí snadno rozberat typy úkonů, které budou uživatelé potřebovat vykonávat, přičemž se nemusí zabývat detailními mechanismy procesů, jejichž definice dosud není hotova.

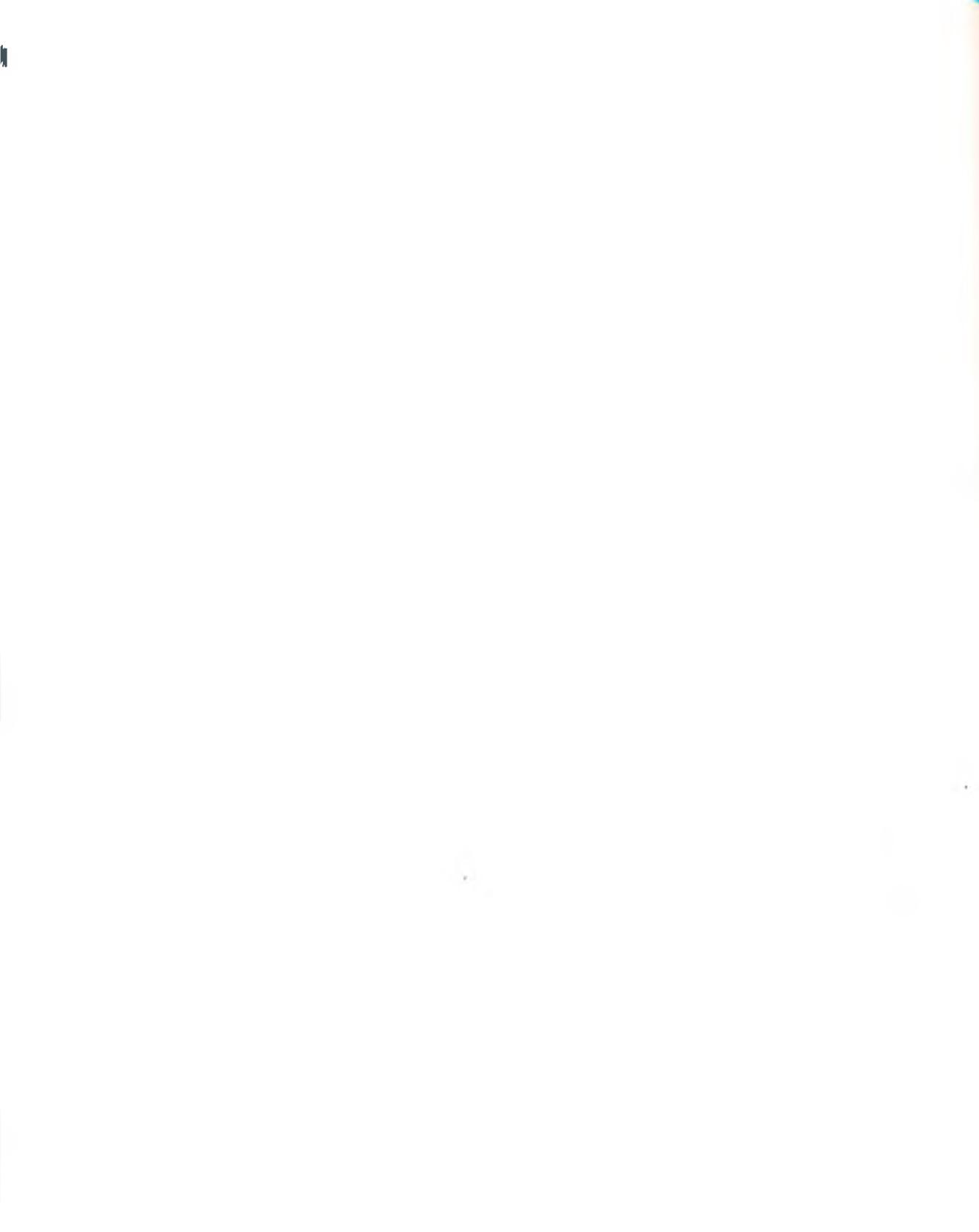
I poměrně jednoduchý scénář, jako například „Obchodní zástupce bude s pomocí systému sledovat stav objednávek podaných zákazníkem ve všech fázích celého procesu, od prvotního pořízení dat přes odeslání zboží, fakturaci až po konečné zaplacení“ dostatečným způsobem vysvětluje, jak bude uživatel se systémem pracovat. Současně nás ale nenutí provádět žádná rozhodnutí ohledně podrobného návrhu funkcí uživatelského rozhraní.

Vývoj uživatelských scénářů a analýza pracovních procesů se samozřejmě vzájemně nevylučují. Analýza pracovních procesů je užitečným nástrojem pro rozbor samotných procesů, zatímco u uživatelských scénářů se návrhář může zaměřit na to, jak budou se systémem pracovat jednotlivé typy uživatelů. U většiny systémů jsou tyto dva okruhy zhruba stejně důležité. Jestliže je projekt

natolik velký, že se v něm vyplatí i rozsáhlejší analýza, má zajisté smysl provést oba typy analýzy. Uživatelské scénáře mohou totiž obvykle přímo vycházet z informací, které získáme při analýze pracovních procesů, takže se nemusíme pouštět do dalších rozhovorů či další analýzy.

### Stručné shrnutí

V této kapitole jsme si ukázali, jak porozumět činnostem, které má navrhovaný systém podporovat. Můžeme k tomu využít analýzu pracovních procesů o různém stupni formálnosti, vytvořit uživatelské scénáře, nebo obojí. V kapitole 9 budeme pokračovat návrhem myšlenkového (konceptuálního) datového modelu: to je logický popis způsobu vytvoření dat, jejich strukturování a využívání v systému.



V této fázi celého procesu návrhu bychom již měli docela přesně vědět, čeho hodláme dosáhnout. Nadefinovali jsme záběr systému, sestavili jsme množinu kritérií návrhu a provedli analýzu pracovních procesů. Nyní je tedy na čase začít s vytvářením datového modelu.

Vzpomeňte si, že myšlenkový neboli konceptuální datový model obsahuje popis jednotlivých entit, jejich atributů a vztahů mezi nimi. Není to totéž co databázové schéma, které popisuje fyzické rozvržení tabulek. K vytvoření databázového schématu nemáme zatím dostatek informací. Nejprve musíme pochopit strukturu uživatelského rozhraní a architekturu, v níž budeme systém implementovat, a pak se teprve pustíme do jeho návrhu.

## Identifikace datových objektů

V předchozích fázích návrhu analýzy jsme sbírali nebo vytvářeli určité zdrojové dokumenty. To znamená jednak dokumenty, které nám poskytne samotný klient – ukázky vstupních formulářů, tištěných sestav atd. – a jednak dokumentace k pracovním procesům, které jsme si připravili sami. Prvním krokem ve vytvoření datového modelu je proto opětovné proštěpení těchto dokumentů a sestavení seznamu všech dat (informací), se kterými bude systém pracovat.

Začneme některým pracovním procesem. Dá se říci, že v podstatě nezáleží na tom, kterým z procesů začneme; já si ale obvykle vyberu některý z centrálních procesů, protože do těchto nejdůležitějších procesů bývá zapojena většina entit. Většinu pracovních procesů odstartuje nějaký papírový dokument, jako je například objednávka prodeje, kterou obchodní zástupce předá pracovníkovi pořizování objednávek. Někdy je spouště určitá jiná událost; v takovém případě bývá jedním z prvních úkonů vyplnění nějakého formuláře. Jestliže budeme pokračovat v našem příkladu zpracování objednávek z kapitoly 8, ukázkový formulář objednávky prodeje může vypadat třeba jako formulář na obrázku 9-1.

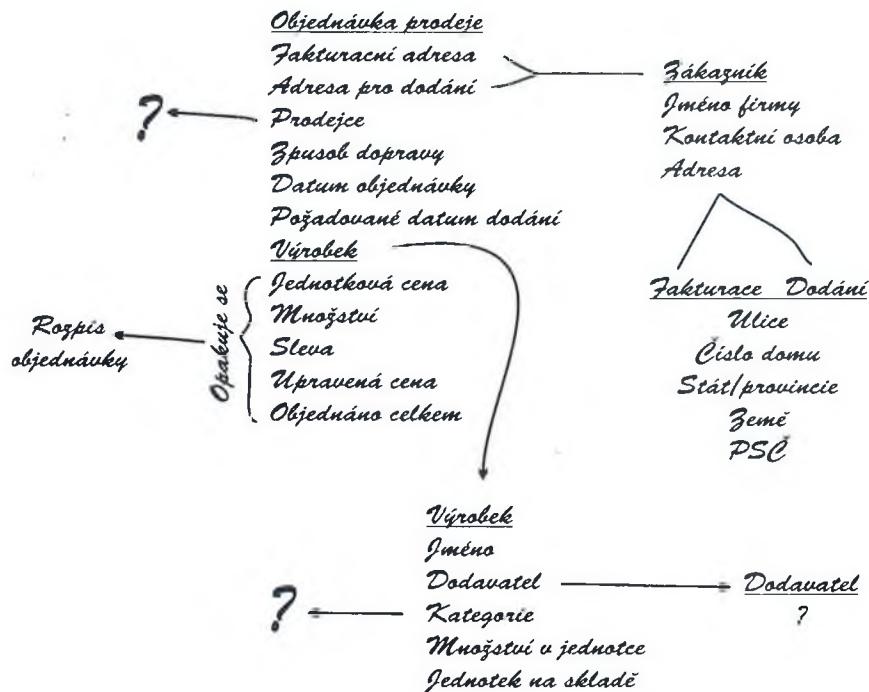
		OBJEDNÁVKA PRODEJE					
One Pier Lots Way, Tivua Pointe WA 48165 Tel/fax: 1-700-555-1234 / fax: 1-700-555- 6533		Datum: 21-VI-2000					
Příjemce:	Wolski Zajazd	Příjemce:	Wolski Zajazd				
ul. Filtrowa 68	ul. Filtrowa 68						
Varsáva 01-012	Varsáva 01-012						
Polsko	Polsko						
<b>Čísločka:</b>	<b>Zákazník:</b>	<b>Prodejce:</b>	<b>Objednávka:</b>	<b>Dodal dle:</b>	<b>Údaje do:</b>	<b>Dopravce:</b>	
10248	WOLZA	Steven Buchanan	4.7.1996	1.8.1996	16.7.1996	Federal Shipping	
<b>Číslo výrobku: Hlavní výrobek:</b>		<b>Množství:</b>	<b>Cena:</b>	<b>Sleva:</b>	<b>Výsledná cena:</b>		
11 Queso Cabrales		12	350,00 Kč	0%	4 200,00 Kč		
42 Singaporean Hokkien Fried Mee		10	245,00 Kč	0%	2 450,00 Kč		
72 Mozzarella di Giovanni		5	607,50 Kč	0%	3 037,50 Kč		
		<b>Mezidovadlo:</b>	9 687,50 Kč				
		<b>Doprava:</b>	809,50 Kč				
		<b>Celkem:</b>	10 497,00 Kč				

Obrázek 9-1 Většinu pracovních procesů spouští určitý papírový dokument, jako je například tato objednávka prodeje

Najděte si příklad tohoto prvního bloku dat a sepište si všechny informace, které v sobě obsahuje. Zatím se nesnažte tyto informace klasifikovat jako entity a atributy, pouze je napište na papír. Poznamenejte si případné opakované skupiny údajů; dále prověřte všechny datové položky, které podle vámi provedené analýzy pracovních procesů chybí. První pokus o sestavení seznamu datových položek může vypadat třeba tak, jako na obrázku 9-2.

<i>Objednávka prodeje</i>
<i>Fakturační adresa</i>
<i>Adresa pro dodání</i>
<i>Prodejce</i>
<i>Způsob dopravy</i>
<i>Datum objednávky</i>
<i>Požadované datum dodání</i>
<i>Výrobek</i>
<i>Jednotková cena</i>
<i>Množství</i>
<i>Sleva</i>

Obrázek 9-2 První verze seznamu datových položek formuláře Objednávka prodeje



Obrázek 9-3 První verze definice entit a atributů formuláře Objednávka prodeje

Po sestavení tohoto prvního seznamu se již můžeme pustit do zkoumání jednotlivých entit, atributů a vztahů. U každé položky seznamu musíme nejprve určit, jestli se jedná o objekt nebo o nějakou skutečnost, která se o objektu zaznamenává. Z objektů se posléze stanou entity a ze zaznamenaných skutečností atributy této entity. Výsledky analýzy budou nejspíše vypadat zhruba jako na obrázku 9-3.

Jak vidíte, položky Fakturační adresa a Adresa pro dodání jsme identifikovali jako fakta, která náleží nově přidané entitě Zákazník. Zatím ještě ale není na první pohled zřejmé, jestli obě adresy patří pouze k entitě Zákazník, pouze k entitě Objednávka prodeje, nebo k oběma entitám. Stejný princip platí i pro položky Jednotková cena, která se může objevit v entitě Rozpis objednávky. Podobně jako je aktuální běžná prodejní cena výrobku logicky odlišná od ceny, za kterou se konkrétní výrobek skutečně prodá, je i aktuální adresa pro fakturaci a adresa pro dodání určitému klientovi logicky odlišná od adresy, na kterou byla zaslána tato faktura a *tato objednávka*.

Jestli má být adresa pro dodání a adresa pro fakturaci také atributem entity Zákazník, to je do jisté míry věc názoru; závisí přitom také na povaze systému a jeho uživatelů. Pokud tyto atributy do entity Zákazník zahrneme, můžeme je v průběhu procesu pořizování objednávek nastavit jako implicitní hodnoty, takže se zkrátí čas potřebný pro pořízení objednávky a sníží se riziko chyb při zápisu dat. Současně tím ale vzniká navíc jistá režie, protože příslušné hodnoty musíme v entitě Zákazník zvlášť udržovat. Jestliže má většina zákazníků dané organizace více fakturačních adres

(jsou to například maloobchodní řetězce nebo jiné firmy s mnoha pobočkami), bude tato režie více než nepříjemná; v takovém případě je vhodnějším řešením zadat adresu pro dodání jednoduše v průběhu pořizování objednávky.

Někdy má smysl vydlat se zlatou střední cestou. Do entity Zákazník přidáme tak například několik atributů s adresou pro dodání, nevyžadujeme však vyplnění všech zároveň. Jestliže má zákazník jen jednu adresu, může ji systém použít jako implicitní hodnotu. Pokud je k němu definováno několik různých adres, může systém uživateli nabídnout seznam, z něhož si dále vybere. Další možnost je implementovat určité podmíněné zpracování, kdy například jako implicitní hodnotu nabídneme naposledy zadanou adresu, ale zároveň umožníme v případě potřeby výběr jiné adresy ze seznamu.

Za zvážení dálé stojí, jestli se má entita Zákazník v procesu pořizování objednávek přímo aktualizovat. Pokud zákazník nemá žádnou adresu, případně pokud uživatel zadá novou adresu, může se systém automaticky zeptat, jestli se tato adresa má přidat k záznamu daného zákazníka. Toto jsou ovšem již ve skutečnosti otázky uživatelského rozhraní aplikace, nikoli otázky datového modelu; uživatelské rozhraní je ale s datovým modelem natolik úzce spjato, že nikdy není možné je od sebe úplně oddělit.

Datová položka Prodejce podle všeho naznačuje, že by se k ní někde měla vázat entita Zaměstnanec, zatím ale nevíme, jak má vypadat, a proto jsme k ní také zapsali otazník. S touto entitou budou nejspíše pracovat také další dokumenty a procesy. V takovém případě k ní můžeme doplnit potřebné atributy. Není-li tomu tak, můžeme se rozhodnout, že položku Prodejce ponecháme definovánu jako atribut entity Objednávka prodeje. Nezapomeňte přitom, že všechna tato rozhodnutí musí vycházet ze sémantiky daného systému. Pokud budeme údaj o zaměstnanci využívat pouze k výpisu jména na Objednávce prodeje, nemá naprostě žádný smysl vytvářet pro entitu Zaměstnanec nějakou složitou vstupní obrazovku s údaji o nadřízeném, oddělení a telefonním čísle.

V položce Výrobek poznáváme samostatnou entitu, přičemž skupina položek, kterou jsme v prvním diagramu odhalili jako opakovou skupinu (Výrobek, Jednotková cena, Množství a Sleva), tvoří entitu Rozpis objednávek. U entity Výrobek jsme popsali jakousi prvotní množinu atributů, zřejmě podle nějakého jiného zdrojového dokumentu; v tomto seznamu jsme dále našli entity Dodavatel a Kategorie (výrobců), i když ani k nim zatím opět nemáme žádné další podrobné informace.

Na tomto místě ještě definujeme jednotlivé entity na myšlenkové (konceptuální) úrovni, takže nás zatím nezajímá, jestli se různé atributy, jako je například atribut Upravená cena z entity Rozpis objednávek nebo atribut Jednotek na skladě z entity Výrobek, budou do tabulek přímo ukládat jako hodnoty, nebo jestli se budou v případě potřeby vypočítávat. Zatím ve skutečnosti ani nevíme, jestli se vůbec dají nějak vypočítat. To zjistíme později.

Zajímavým údajem je atribut Způsob dopravy. V mnoha formulářích objednávek najdeme skupinu zahrávacích „chlívečků“, které označují zvolený způsob dopravy, například „Česká pošta“, „Obchodní balík“, „Železniční spěšnina“, „Kusová zásilka“, „DHL“ a podobně. Jsou to entity nebo atributy? Přijde na to. (Uhádli jste, že?) Kolik máme možností? Pokud jsou více než dvě, asi je sotva budeme schopni pohodlně modelovat jako jediný atribut.

Do jaké míry jsou jednotlivé varianty stabilní? Na tomto místě pracujeme nejspíše s nějakými externími poskytovateli služeb. Bude klientská organizace tyto poskytovatele často střídat nebo pří-

dávat další? Nakolik musí být schopna organizace reagovat na různé speciální způsoby doručování? Zamítne firma prodejní případ, kvůli kterému by si musela najmout himálajského šerpu pro vynesení zboží na Mount Everest? Jestliže zní odpověď ne, musí být schopen i náš model se všemi těmito variantami počítat.

Pokud budeme způsob dopravy modelovat jako samostatnou entitu, bude uživatel moci jednotlivé položky (jednotlivé způsoby) kdykoli měnit a stejně tak bude moci kdykoli přidávat nové, cennou, kterou za tuto flexibilitu platíme, je ale složitější datový model i uživatelské rozhraní. Rozdíl může spočívat v několika málo stisknutích kláves navíc – uživatel například namísto zatržení vhodného zaškrávacího políčka myši musí vybrat položku z rozbalovacího seznamu. Těchto několik málo stisků kláves – nebudeste-li dostatečně opatrní – může ale také přispět ke krkolomnému, pomalému rozhraní.

Jestliže tedy společnost musí pracovat i s různými speciálními způsoby dopravy, je třeba pečlivě zvážit, jakým způsobem s nimi budeme počítat také my. Musíme najít velice dobrý kompromis: na jedné straně potřebujeme dostatečnou flexibilitu, kdy musíme být schopni ošetřit všechny rozumné případy, na druhé straně ale zároveň nesmíme na uživatele uvalit zbytečnou zátěž navíc. V našem případě by bylo zřejmě nevhodnějším řešením doplnění nepovinného atributu Speciální instrukce; s tímto novým atributem pak ale musíme počítat jak v datovém modelu, tak i ve všech procesech nového systému.

Tato rozhodnutí mohou až nečekanými způsoby ovlivnit různá omezení systému. V našem případě sice organizace přesně ví, jakým způsobem má zboží zákazníkovi odeslat, nyní již ale nestačí prostě vyžádat zadání způsobu dopravy. Systém musí vědět, že atributy Způsob dopravy a Speciální instrukce nesmí být *oba* zároveň prázdné; to je trochu složitější pravidlo, které musíme v modelu implementovat na jiné úrovni.

Jestliže přitom vlastní doručení (doprava) zboží tvoří přímou součást systému, je vhodné (a zřejmě i nezbytné) považovat Způsob dopravy za samostatnou entitu; další mimořádné způsoby dopravy však již mohou pro systém znamenat dosti významný objem režie navíc. Jak ale máme zachytit podrobné informace o způsobu dopravy, který fakticky dopředu přesně neznáme? Zde máme v zásadě dvě možnosti: buďto můžeme vytvořit obecnou entitu způsobu dopravy, která bude obsahovat atributy známé u většiny způsobů dopravy, jako je například podací číslo a čas doručení, anebo můžeme postihnout jen známé způsoby dopravy a speciální metody ponechat speciálnímu zpracování – je to výjimečná situace, která se musí ošetřit vně našeho systému.

Zde totiž hrozí nebezpečí přílišného zkomplikování systému; uživatel by jej tím pádem mohl vnímat jako příliš složitý a obtížně by se v něm orientoval. Je až příliš snadné nechat se unést ohromnými funkcemi, které můžeme v systému nabídnout, a zapomenout na režii a pracnost, která je s těmito funkcemi spojena. Ano, nabídnout implicitní hodnoty je dobré ovšem *za předpokladu*, že se dají snadno udržovat a že se tato údržba provádí pravidelně (nejlépe jako vedlejší produkt nějaké jiné činnosti). Pokud umožníme podávání dotazů na doručení zboží i recepčnímu, je to sice přijemné, na druhé straně ale uvažujme – stojí za to zadávat u tisíců objednávek podrobné informace o dopravě, které ve skutečnosti budou sloužit jen pro pět zákazníků, kteří takový dotaz položí?

Při tomto rozhodování o způsobu návrhu si musíme pouze důkladně promyslet veškeré jejich důsledky. Dokud jsme ale v takové té první fázi, kdy si říkáme: „Není to báječné? Tolik nám to uše-

trí času", snadno spoustu věcí přehlédneme. Jakmile kdekoli v systému podchytíte určitý údaj, přemýšlejte o tom, jestli by se nedal ještě někde využít; systém by jej mohl definovat například jako implicitní hodnotu nebo jako omezení. Jestliže *tak jako tak* zadáváme podrobné informace o způsobu dopravy, proč by je neměl mít k dispozici i ten recepční?

Úvahy je ale třeba vést i opačným směrem: kdekoli nějakou informaci potřebujeme, musíme zvážit, kde se bude vytvářet a jak se s ní bude manipulovat. Obecně je vhodnější nabídnout uživateli seznam voleb než prosté textové pole. Zároveň nám tak ale vznikají náklady spojené s vytvořením a údržbou tohoto seznamu, přičemž pro údržbu musíme navrhnut také vhodné rozhraní. Při rozhodování o struktuře datového modelu musíme proto zvažovat všechny tyto věci a musíme mezi nimi najít odpovídající rovnováhu.

Naším cílem jistě bude od uživatele nikdy nepožadovat dvojí zadávání jednoho stejného údaje. Současně ale nechceme uživatele nutit k tomu, aby vstával od stolu a kdoví kde sháněl požadované informace jen proto, že je daný úkol vyžaduje. O této problematice budeme hovořit podrobnej v části 3, správně rozpoznat okamžik vytvoření jednotlivých údajů a okamžik jejich využití je však životně důležitým prvním krokem celého procesu.

## Definice vztahů

Po rozebrání všech zdrojových dokumentů máme již v ruce hrubý popis entit v daném prostoru problémů a jejich atributů. Zhývají ještě dva úkoly: mezi těmito entitami navázat odpovídající vztahy a podívat se na atributy a omezení každé jednotlivé entity.

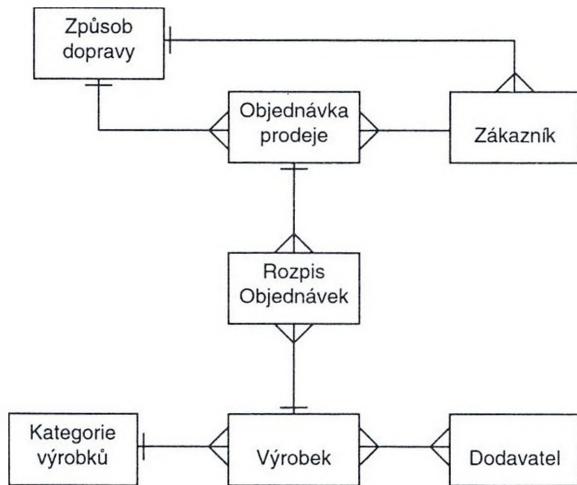
Teoreticky bychom sice mohli nejprve nadefinovat atributy, podle mého názoru je ale nejjednodušší začít právě vztahy; některé z nich vedou totiž k vytvoření nových, samostatných entit a některé si vyžádají doplnění dalších atributů k již identifikovaným entitám.

Máte-li alespoň částečně podobný styl práce jako já, bude výsledkem prvního „průchodu“ zdrojových dokumentů hromada ručně psaných poznámek, prošpikovaných různými šípkami a vzájemnými odkazy typu „viz strana 12“, které nejspíše nikdo jiný nebude schopen dešifrovat. Prvním krokem při definici vztahů bude tedy nějaké „rozumné“ a pěkné uspořádání těchto poznámek. Můžeme tedy začít vytvořením první verze diagramu entit a vztahů (E/R diagramu) našeho datového modelu. (A pokud jsou vaše poznámky natolik chaotické a nečitelné, že se i *vý sami* obáváte, že byste je třeba za čtrnáct dní nebyli schopni rozluštit, můžete již v této fázi popsat identifikované atributy jednotlivých entit.)

Začneme tím, že si vybereme některou entitu, obvykle jednu ze základních entit systému, a poté do modelu přidáme další entity, které mají k vybrané první entitě nějaký vztah. Současně s popisem vztahů můžeme nadefinovat i jejich povahu (tedy jestli je vztah typu jedna k jedné, jedna k více nebo více k více), povinné to ale není – můžeme nakreslit obyčejnou čáru, která znamená, že mezi danými entitami *je* nějaký vztah, k jehož analýze se máme vrátit později. Já obvykle potřebnou analýzu provádí rovnou, třeba ale zrovna pro vás bude jednodušší popsat nejprve všechny entity a teprve poté je blíže analyzovat.

První náčrt E/R diagramu pro náš příklad se zpracováním objednávek vidíme na obrázku 7-4. Tento příklad je opravdu jednoduchý, a diagram je tudíž i docela dobře čitelný. (Zde jsme se tedy rozhodli, že Prodejce bude pouze atributem entity Rozpis objednávek, nikoli samostatnou entitou.)

tou.) Při práci na složitějším příkladu musíme někdy nakreslit i několik diagramů, z nichž každý popisuje pouze jistou podmnožinu dat. V takovém případě je vhodné kreslit diagramy s pomocí určitých automatizovaných mechanismů. Jinak je dosti obtížné dosáhnout toho, aby si vzájemně odpovídaly.

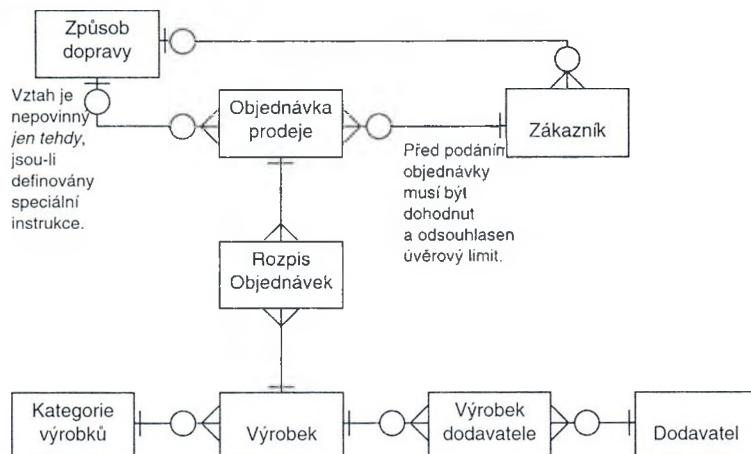


Obrázek 9-4 První verze E/R diagramu aplikace Zpracování objednávek

Nad připraveným hrubým E/R diagramem již můžeme začít analyzovat jednotlivé vztahy podrobněji. U každého vztahu tak musíme určit:

- Kardinalitu vztahu
- Volitelnost každého z účastníků
- Veškeré atributy vztahu
- Případná omezení vztahu

Obrázek 9-5 ukazuje E/R diagram zpracování objednávek ještě jednou, a to po uvedených úpravách.



Obrázek 9-5 E/R diagram zpracování objednávek po provedené analýze vztahů

### Kardinalita vztahu

Možná jste vztahy mezi entitami vyznačili hned v prvním načernutém obrázku, jako jsem to udělala i já na obrázku 9-4. Pokud ne, je teď právě načase. I když jste třeba vztahy v obrázku vyznačili, je vhodné původní rozhodnutí nyní znova prověřit, abychom tak získali úplnější obrázek celého datového modelu.

Jestliže někde přijdeme na vztah typu více k více, musíme do modelu doplnit spojovací entitu, která bude z obou stran propojena vztahem typu jedna k více. V našem modelu je typu více k více vztah mezi entitami Výrobek a Dodavatel, takže k němu musíme vytvořit spojovací entitu Výrobek dodavatele. Všimněte si, že i mezi entitami Objednávka prodeje a Výrobek je vztah typu více k více, tentokrát již ale roli spojovací entity zastupuje entita Rozpis objednávek.

### Volitelnost vztahu

Nyní jsme tedy nadefinovali typ vztahu mezi veškerými dvojicemi entit a dále musíme zvážit, jestli je pro některého z účastníků (případně pro oba) tento vztah volitelný (nepovinný). V našem příkladu se podívejme na vztah mezi entitami Zákazník a Způsob dopravy – ten je volitelný v obou směrech, takže zákazník nemusí mít definovaný žádný implicitní způsob dopravy a naopak způsoby dopravy mohou existovat i přesto, že je žádný zákazník nevyužívá.

Vztah mezi entitami Kategorie výrobků a Výrobek je oproti tomu nepovinný jen v jednom směru. K určité Kategorii výrobků nemusí být přiřazen žádný Výrobek, naopak ale každý Výrobek musí patřit do nějaké Kategorie výrobků.

Mezi entitami Objednávka prodeje a Způsob dopravy je definován ještě složitější vztah. Způsob dopravy může existovat nezávisle na jakémkoliv Objednávce prodeje, takže strana Objednávky prodeje je v tomto vztahu vždy nepovinná. Ze strany Způsobu dopravy je ale vztah nepovinný jen za podmínky, že daná Objednávka prodeje obsahuje speciální instrukce. To je důležité omezení, které musíme do diagramu zvlášť poznamenat.

## Atributy vztahu

Ve většině situací nám stačí si o určitém vztahu mezi dvěma entitami zaznamenat pouze fakt, že skutečně existuje. Potřebujeme tedy například vědět, že určitý Zákazník podal jistou Objednávku prodeje a to je *všechno*, nic jiného již vědět nepotřebujeme. Někdy je ale nutné znát o vztahu další, doplňující údaje – například kdy vztah začal a jak dlouho trval. Tyto údaje jsou již atributy samotného vztahu, nikoli atributy některého z účastníků.

Jestliže má určitý vztah nějaké atributy, musíme jej vždy modelovat jako samostatnou entitu. V našem příkladu se zpracováním objednávek bychom mohli například jednoho Dodavatele každé položky zboží (každého výrobku) označit za preferovaného, tedy mu dát stav „Přednostní dodavatel“. Protože zde máme již mezi entitami Výrobek a Dodavatel vytvořenou spojovací entitu, můžeme atribut Přednostní dodavatel doplnit jednoduše do ní. Pokud by spojovací entita chyběla, museli bychom vytvořit novou, speciální entitu, která by vyjadřovala atributy vztahu.

## Doplňující omezení vztahu

Nakonec se musíme zamyslet nad tím, jestli k danému vztahu nepatří ještě nějaká další, doplňující omezení. Jaký minimální a maximální počet záznamů může být definován na straně „více“ vztahu typu jedna k více? Musí být před zavedením vztahu splněny nějaké podmínky? Jsou naopak podmínky, za kterých vztah existovat *musí*?

V našem příkladu je jedním takovým omezením požadavek, podle něhož je vztah mezi Objednávkou prodeje a Způsobem dopravy nepovinný jen tehdy, pokud byly definovány Speciální instrukce. Dalším příkladem podobného omezení je pravidlo, podle něhož zákazník nesmí podat objednávku, dokud pro něj nebyl potvrzen a odsouhlasen úvěrový limit. V diagramu označujeme toto pravidlo opět pomocí zapsané poznámky. Pokud je přitom v datovém modelu definováno příliš mnoho omezení, nebo pokud jsou natolik složitá, že se jen těžko dají zachytit v prosté poznámce, můžeme je dokumentovat i jinde. Do diagramu je ale každopádně vhodné poznačit, že takové omezení existuje.

## Rozbor entit

Začíná se nám již docela pěkně rýsovat celkový obrázek všech entit v systému a jejich vzájemných vazeb. Je tedy načase začít s podrobnou analýzou každé jednotlivé entity. U každé entity musíme zjistit následující informace:

- Vztah mezi entitou a daným prostorem problému
- Pracovní procesy, které entitu vytvářejí, modifikují, používají a odstraňují
- Případné další entity, se kterými může entita komunikovat nebo na kterých je závislá
- Aplikační pravidla a omezení náležící k entitě
- Atributy dané entity

## Vztah mezi entitou a prostorem problému

Rozpoznat vztahy mezi entitou a daným prostorem problému je zpravidla docela jednoduché. „Entita Zákazník modeluje jednotlivce a organizace, které nakupují naše výrobky.“ Největší problém zde podle mého názoru spočívá v tom, že poměrně často vymyslíme větu, která je bezna-

dějnou tautologií. Příkladem je výrok „Entita Zaměstnanci modeluje zaměstnance naší organizace“, který nám zřejmě ve své podstatě nic neříká.

Pokud se v nějaké entitě modeluje vztah, vznikají tím poněkud záludnější vazby, protože taková entita se nemapuje přímo do prostoru problému. „Dodavatel může dodávat více různých výrobků a jeden daný výrobek můžeme odebírat od libovolného počtu různých dodavatelů. Tento vztah modeluje entita Výrobek dodavatele, která kromě toho popisuje také stav Přednostního dodavatele, definovaného pro libovolného daného dodavatele a určitý konkrétní výrobek.“

Někdy se určité věci v prostoru problému – zřejmě nejlepším příkladem je zde Objednávka prodeje – modelují v datovém modelu pomocí jedné nebo více logických entit. Takovým entitám pak říkáme *složené entity*. Písemný dokument objednávky prodeje reprezentuje v našem modelu entita Objednávka prodeje a *zároveň* entita Rozpis objednávek.

Pro účely dokumentace mi ale většinou připadá jednodušší popisovat složené entity jako jediný objekt. Například: „Entity Objednávka prodeje a Rozpis objednávek reprezentují jednu objednávku podanou určitým Zákazníkem. Entita Objednávka prodeje modeluje přitom samotnou objednávku, zatímco entita Rozpis objednávek vyjadřuje jednotlivé objednané výrobky.“

### **Pracovní procesy, které ovlivňují entitu**

Místo, kde se jednotlivé datové položky využívají, jste možná již identifikovali v rámci dříve provedené analýzy pracovních procesů. Přesto je vhodné tyto informace zahrnout i do dokumentace k dané entitě. Jestliže totiž budeme potřebovat někdy později změnit strukturu entity, například do ní doplnit nějaký atribut, máme tímto způsobem vytvořeno jediné místo, které popisuje veškeré dotčené pracovní procesy.

Identifikovat procesy, které s entitou přímo pracují, je také obvykle poměrně jednoduché. Postihnout procesy, které do entity zasahují nepřímo, již může být pracnější. Na první pohled nemusí být tak například zcela zřejmé, že proces pořizování objednávek může změnit implicitní způsob dopravy daného zákazníka, nebo že atribut „Speciální bonus“, definovaný pro kategorii výrobků, může ovlivnit výši slevy, a tudíž i celkovou hodnotu objednávky prodeje. A přesně toto jsou ony typy interakcí a vzájemných vazeb, které, nejsou-li řádně a pečlivě zdokumentovány, jsou pro programátora údržby přímo noční můrou.

Většina analytiků dokumentuje tyto vzájemné vazby v analýze pracovních procesů; to má zcela zřejmě smysl v případě, že se změny týkají samotných procesů. Někdy však změny zasahují i do samotného modelu, a to buďto přímo, z důvodu změn v aplikačním prostředí, nebo nepřímo, protože si určité zásahy do modelu vyžaduje například změna v některém existujícím procesu. V tomto případě je mnohem snazší projít si dokumentaci ke konkrétní entitě, než kdybychom se museli prohrazbovat všemi pracovními procesy a složitě z nich zjišťovat, kterého z nich se změna může týkat.

Informace o pracovních procesech je vhodné v dokumentaci k entitám prokládat křížovými odkazy. Podobně jako i všechny jiné křížové odkazy může být i takový model náročnější na implementaci i na údržbu, z dlouhodobého hlediska nám však usnadní život.

## Interakce mezi entitami

E/R diagramy jsou vynikající nástroje, na druhé straně však dokáží zobrazit jen poměrně málo informací. Jestliže entity v modelovaném systému spolu složitě vzájemně komunikují a mají složité interakce, které se v tomto diagramu nedají snadno vyjádřit, je důležité je dokumentovat v rámci popisu jednotlivých entit. I když třeba do diagramu doplníme textové poznámky, veškeré interakce, které nejsou z poznámek okamžitě jasné, bychom měli dále rozvinout.

Pokud je model natolik složitý, že jej musíme vyjádřit pomocí většího počtu diagramů entit a pokud se v něm určitá entita objevuje v několika diagramech současně, je vhodné uvést v popisu této entity seznam všech ostatních entit, které jsou s touto entitou nějakým způsobem svázány. To platí zejména pro entity, které definují množinu platných hodnot, z nichž vybíráme na několika různých místech. Na entitu Formální oslovení, která bude obsahovat položky jako „Pan“, „Paní“, „Slečna“, „Profesor“ apod., se budeme například odvolávat třeba v desítky různých míst. Jestliže tedy potřebujeme v této entitě provést nějakou zásadní změnu, je velice užitečné mít všechny odkazované entity sepsané na jednom místě.

Obecně se ale dá říci, že E/R diagram nabízí již sám o sobě dostačující prostředky pro dokumentaci interakce mezi jednotlivými entitami. Další, doplňující informace musíme dokumentovat jen u dosť výjimečných případů, jejichž příklad jsme si uvedli v předcházejícím odstavci.

## Aplikační pravidla a omezení

Další nezbytnou součástí dokumentace každé entity jsou informace o veškerých omezeních definovaných pro ni na úrovni entity. Každá entita podléhá samozřejmě základnímu „omezení“, podle něhož musí mít jednoznačnou identifikaci; ze svých zkušeností považuji za vhodné popsat atributy, které budou sloužit jako primární klíč tabulky, již v popisu entity.

Dokumentovat je také třeba případná omezení, která se odkazují na několik atributů, jako byla v našem příkladu podmínka „atributy Způsob dopravy a Speciální instrukce nemohou být zároveň prázdné“; i pro tuto dokumentaci je tedy vhodný okamžik právě nyní.

## Atributy

Poslední věcí, které musíme ke každé entitě dokumentovat, je seznam atributů a jejich oborů hodnot (domén). Při sestavování seznamu atributů začneme nejspíše prvotním seznamem, který jsme vytvořili během rozboru zdrojových dokumentů, a poté k nim doplníme cizí (nevlastní) klíče potřebné pro zajištění referenční integrity.

Kromě toho musíme ověřit, jestli má každá entita alespoň jeden kandidátní klíč, který může jednoznačně identifikovat jednotlivé její instance. V databázovém schématu se z tohoto kandidátního klíče stane primární klíč tabulky. Nezapomeňte přitom, že primární klíče nemohou obsahovat prázdné hodnoty Null. Jako klíč nemůžeme proto vždy použít nějaký stávající atribut či kombinaci stávajících atributů. V takovém případě je třeba do entity doplnit libovolný systémem generovaný jednoznačný identifikátor.

Takový „umělý“ identifikátor bude v našem příkladu zřejmě potřebovat entita Zákazník. Budeme-li předpokládat, že zákazníkem může být buďto společnost (firma), nebo jednotlivec (osoba), můžeme z prvotní analýzy dostat seznam atributů, jaký je uveden na obrázku 9-6.



Obrázek 9-6 Seznam atributů entity Zákazník

I když zde třeba pomineme skutečnost, že jména nemohou být jednoznačná, přesto ještě zblývá jeden problém. Pokud je zákazníkem jednotlivec, bude prázdné Jméno firmy. Jestliže je zákazníkem společnost, bude naopak prázdné Jméno osoby. Vždycky je tedy něco prázdné. To znamená, že již z toho důvodu nemůžeme tato pole použít jako kandidátní klíč, i kdybychom třeba bezpečně věděli, že jednoznačně identifikují záznamy – a ani to bezpečně nevíme.

Tyto úvahy mě přivádějí k dalšímu problému s entitou Zákazník. Jména zákazníků totiž nejsou jednoznačná. V našem případě ani celý seznam atributů nezaručuje jednoznačnost, protože dvě osoby jako jednotlivci mohou mít nejen stejné jméno, ale mohou bydlet i na stejně adrese. Ze jména Standa Procházka nepoznáme, že se ve skutečnosti jedná o Standu Procházku ml., který se svým otcem Standou Procházkou st. bydlí v jednom domě se stejnou adresou.

Standu Procházka starší a Standa Procházka mladší jsou zcela nepochybně dva různí lidé, takže je musíme reprezentovat ve dvou různých záznamech. K atributům, pomocí nichž bychom je snad mohli vzájemně rozpoznat a jednoznačně identifikovat, však nemáme přístup. Dovedete si představit, že by se vás obchodník při podání objednávky potravin dotazoval na poměry, ve kterých bydlíte? „Promiňte, pane, ale nemáte náhodou nějakého příbuzného, který s vámi bydlí a který se jmenuje stejně jako vy? Jenom si to potřebuji ověřit pro náš počítačový systém.“ To zajisté není zrovna ideální služba zákazníkům.

Tento zdánlivě neřešitelný problém má ale naštětí docela jednoduchou odpověď: je to atribut Číslo zákazníka. A jestliže v dané organizaci ještě neexistuje žádný způsob přidělování čísel zákazníků, můžeme pro ně využít automatické mechanismy databázového stroje Microsoft Jet, respektive Microsoft SQL Serveru (jedná se o datový typ Automatické číslo, respektive Identity).

Jestliže se ale rozhodnete pro takový nahodile zvolený identifikátor, nezapomeňte v systému po psat i jiný, alternativní způsob identifikace zákazníků. Odmítanou objednávku jen proto, že zákazník zapomněl svoje přidělené číslo zákazníka, je rozhodně nepřípustné. Můžeme se tak zákazník zeptat: „Jste ten Standa Procházka ze Žižkova, nebo Standa Procházka z Bohnic?“ Žádat jej, aby se pokusil najít nějakou starou fakturu a poté nám zavolal znovu, to zdaleka tak dobré zajisté není.

Entita Zákazník vhodně ilustruje také druhý důvod, pro který se doporučuje systémem generovaný, nahodilý identifikátor. I kdybychom totiž mohli předpokládat, že spojení jména a adresy bude k jednoznačné identifikaci zákazníka pro naše účely stačit, museli bychom při svázání každé další entity kopírovat příliš velké množství polí. Nezapomeňte, že každý primární klíč dané entity tvoří zároveň cizí klíč ve všech jiných entitách, které se na původní entitu odkazují. A zajisté se mnou budete souhlasit, že je mnohem efektivnější kopírovat jediný atribut, než jich opakovat třeba pět nebo šest.

## Analýza oborů hodnot

V seznamu atributů na obrázku 9-6 jsme viděli zápis tvaru Atribut:Doména. Řada analytiků možnost definice domény neboli oboru hodnot ignoruje a popisuje atributy přímo pomocí jejich datového typu a odpovídajícího omezení. Jestliže se tedy i vy sami rozhodnete tento krok analýzy vypustit, budete v docela „dobré společnosti“. Není to sice *správné*, přesto se ale nedá očekávat, že by vám to někdo vytknul jako chybu.

Proč tedy třeba já sama provádím analýzu oborů hodnot a proč ji doporučuji i vám? Důvod je docela jednoduchý: šetří další práci a nabízí nám doplňující informace. A cokoli je snazší a lepší, je podle mého názoru vhodné zvážit. Další výhodou tohoto kroku analýzy je, že je také technicky správný.

Podívejme se na jeden příklad: o attributech Jméno firmy a Jméno osoby tak obrázek 9-6 neříká nic jiného, než že množinu svých platných hodnot odvozuje od domény Jméno.

Nyní můžeme například následujícím způsobem definovat doménu Jméno:

„Je to řetězec jednoho nebo více slov se správně zapsanými velkými písmeny, o maximální délce 75 znaků. V řetězci jsou povoleny pouze abecední znaky a interpunkční znaménka tečka (.) a čárka (,).“

Jednu doménu neboli obor hodnot stačí definovat jen jednou, a poté se na ni můžeme v celém systému libovolně mnohokrát odvolávat. Všechna tato omezení bychom samozřejmě mohli nadefinovat pro každý odpovídající atribut, ale proč bychom se takto namáhali? Navíc, jestliže jsou tyto atributy definovány nad stejnou doménou, pak bezpečně víme, že je můžeme logicky porovnat. Pokud bychom je definovali přímo, nemuselo by to být hned na první pohled zřejmé.

Vyhledat zákazníka, který má uvedenu stejnou hodnotu v poli Jméno firmy a Jméno osoby, není sice právě nejužitečnější operace, ale rozhodně je to přinejmenším možné a snad i smysluplné. O porovnání atributů jména firmy a čísla zákazníka, které mohou mít v podstatě náhodou stejnou strukturu a omezení, se to již říci nedá.

Technicky můžeme doménu definovat jako „množinu hodnot, ze kterých může daný atribut vybírat svoji hodnotu“. To je sice teoreticky úplně jasné, ale jak vlastně můžeme doménu neboli obor hodnot definovat? V podstatě zde musíme popsat tři základní věci:

- Datový typ domény
- Případná omezení množiny hodnot přípustná v daném datovém typu
- Nepovinně jakékoli formátování, příslušné k definované doméně.

## Výběr vhodného datového typu

Prvním krokem při definici domény neboli oboru hodnot je výběr základního datového typu, který bude v databázovém schématu tuto doménu reprezentovat. To je tedy jediné místo, kde je velice praktické porušit pravidlo úplného oddělení databázového schématu od myšlenkového datového modelu.

Datový typ je zde jakýmsi zkráceným popisem množiny hodnot. Zatímco výraz „Celé číslo“ není sám o sobě doménou (pokud nemodelujeme nějaké matematické teorie), hodnoty domény „Množství“ jsou ale oproti tomu téměř určitě celá čísla. Nedoporučuji ale vázat se příliš na specifiku konkrétního databázového stroje. V tomto okamžiku můžeme totiž ještě cílový databázový stroj změnit.

„Datovým typem“ domény může být také jiná doména. Dejme tomu, že již máme nadefinován obecný datový typ Datum, který například říká, že se všechny datumové údaje v systému musí naházet po dni 1. ledna 1900 a budou formátovány pomocí čtyřciferného roku. Doménu Datum události pak můžeme naprosto správně definovat jako „Datum, které nenastalo dříve než 23. října 1982 (den zahájení podnikání).“

## Omezení množiny hodnot

Dalším krokem, který následuje po rozhodnutí o základním datovém typu domény, je rozhodnout, jaké hodnoty ze základní množiny tohoto datového typu budeme v doméně považovat za platné. Někdy určíme platné hodnoty nejsnáze pomocí určitého pravidla, například: „Množství musí být kladné celé číslo.“

Někdy je jednodušší vypsat u domény seznam platných hodnot. „Oblast musí nabývat jedně z těchto hodnot: Severozápad, Severovýchod, Střed, Jih.“ V tomto případě zahrneme téměř jistě doménu do datového modelu jako samostatnou entitu. To je mnohem pohodlnější řešení, než kdybychom museli uvedené hodnoty s každým odkazem na doménu znova opisovat; entita současně umožňuje jednotlivé hodnoty snadno změnit, a to i po implementaci systému.

Jedinou možnou výjimkou z tohoto pravidla je situace, kdy doménu tvoří opravdu malý počet různých hodnot, které se v žádném případě nebudou měnit. Řekněme například, že modelujeme určitý dotazník či zkoušku; v rámci tohoto modelu nadefinujeme doménu Odpověď, která se skládá ze dvou hodnot, „True“ a „False“ (pravda a nepravda). Uvedené dvě hodnoty nemá žádný smysl modelovat v samostatné entitě. Doména nemůže nabývat žádné jiné hodnoty a při implementaci by odkaz na tabulku znamenal téměř určitě větší problémy než přímý zápis pravidla.

Pomocí entity budeme modelovat také domény, k jejichž definici potřebujeme více než jeden atribut. Nejlepším příkladem je zřejmě atribut Region (neboli v americkém prostředí stát unie). Jestliže musíme v systému počítat s více různými zeměmi, nemůžeme přímo zjistit, jestli je daný stát platný, aniž bychom si nejprve zjistili zadanou zemi.

Jestliže se daný zákazník nachází v Austrálii, bude například řetězec „New South Wales“ platným regionem (státem), zatímco „Alabama“ nikoli. V tomto případě musí vyhledávací entita naší domény obsahovat oba atributy, tedy Region i Země. Tento příklad však již není striktně vzato definicí domény a v E/R modelu se vyjadřuje pomocí povinných vztahů. Situaci však můžeme chápout jako určitou složenou doménu a podle toho s ní také zacházet.

Smyslem zde konec konců není nic jiného, než zjednodušit identifikaci domén (oborů hodnot) v systému; pomocí vhodné úpravy definice těch domén, které se v datovém modelu vyskytují opakováně, si ušetříme čas a zároveň snížíme riziko chyb.

Ve specifikaci domény musíme dálé říci, jestli atributy definované nad danou doménou mají připouštět prázdné hodnoty Null, řetězce nulové délky, případně obojí. Tyto charakteristiky je vhodné deklarovat explicitně i v případě, že modelujeme určitou množinu hodnot pomocí systémové entity; pak totiž povolení hodnot Null můžeme určit podle vztahu mezi dvěma entitami.

Provedení datové analýzy a identifikace seznamu atributů každé jednotlivé entity jsou dva iterativní procesy, které spolu úzce souvisí. V praxi asi přijdete na to, že je nejfektivnější definovat domény současně s vytvářením seznamu atributů. Jestliže je již doména určitého atributu definována, stačí jednoduše zapsat její jméno. V opačném případě můžeme novou doménu ihned definovat podle vzniklého příkladu.

Během tohoto procesu můžeme zjistit, že pro určité atributy platí kromě omezení definovaných pro celou doménu ještě další, doplňující omezení. To je zcela v pořádku a není to nikterak neobvyklé. Tako můžeme nadefinovat například doménu Datum události, která vyjadřuje datum, v němž může vzniknout libovolná událost. V této doméně omezíme datum na časový interval po zahájení podnikatelské činnosti. V příkladu Objednávek prodeje bychom tak nad doménou Datum události mohli nadefinovat jak Datum objednávky, tak i Datum odeslání. Atribut Datum odeslání musí být ale *zároveň* po Datu objednávky. To je již omezení na úrovni entity a jako takové je musíme zapsat do popisu entity.

Při definici omezení domény (a konec konců i dalších omezení atributů) je třeba být co nejvíce konkrétní, *přičemž zároveň nesmíme ohrozit použitelnost systému*. Podrobněji budeme o tomto tématu hovořit v části 3 této knihy, na tomto místě si proto alespoň řekneme, že čím přesněji určíme doménu definujeme, tím vyšší stupeň pomoci můžeme nabídnout uživateli. Jestliže ale nedopatřením vypustíme z domény některé správné, důležité hodnoty, zkomplikujeme uživateli život a celý systém může být nakonec zcela nepoužitelný.

## Definice formátu

Pro doménu je často také vhodné, i když ne nezbytně nutné, specifikovat odpovídající formát. Jestliže tak například řekneme, že veškeré datumové údaje se mají zobrazovat ve formátu DD-MMM-RRRR, nemusíte již tento formát nikde znova opakovat.

## Normalizace

Možná vás překvapuje, že jsem se v tomto výkladu datového modelování nikde nezmínila o normalizaci datového modelu. Podle mých vlastních zkušeností to jde nějak samo sebou: jestliže začnete jednotlivými údaji a informacemi, které posléze uspořádáte do entit, budete nejspíše opakovány skupiny a vztahy typu více k více odstraňovat již v průběhu analýzy a přímo tak dostanete datový model ve třetí normální formě.

Na druhé straně ale ověřit si, jestli výsledný datový model vyhovuje podmínkám normality, není rozhodně vůbec na závadu, zejména pro vás jako začátečníky. Nezapomeňte, že každá entita v modelu musí být závislá (a jednoznačně definovaná) na „klíči, celém klíči a na ničem jiném než na klíči“.

### **Stručné shrnutí**

V této kapitole jsme se zabývali sestavením myšlenkového (konceptuálního) datového modelu navrhovaného systému. Uvedený proces začíná prostudováním všech zdrojových materiálů, v nichž se snažíme rozpoznat elementy dat, které se v systému používají, a které posléze uspořádáme do množiny entit. Dále zjistíme vztahy mezi entitami a provedeme analýzu každé jednotlivé entity a jejích atributů.

V následující kapitole budeme tento myšlenkový model převádět do podoby fyzického databázového schématu, která se již implementuje nad zvoleným konkrétním databázovým strojem.

V poslední kapitole jsme rozebírali myšlenkový neboli konceptuální datový model, který definuje logickou strukturu dat. Nyní tedy v této kapitole přejdeme k databázovému schématu, které již popisuje fyzickou strukturu dat. Nezapomeňte ale přitom, že i databázové schéma je ještě logickou konstrukcí; při návrhu databázového schématu tak budeme fyzickou strukturu dat popisovat v dosti abstraktních pojmech. Za vlastní fyzickou reprezentaci dat již odpovídá databázový stroj, takže se o ni v podstatě nemusíme zajímat.

## Architektury systémů

Před zahájením vlastního popisu databázového schématu musíme provést určitá rozhodnutí ohledně architektury, kterou modelovaný systém vyžaduje. Literatura však naneštěstí používá pojem „architektura“ ve významu dvou různých (i když vzájemně propojovaných) modelů. Já budu tyto dva modely rozlišovat, takže jednomu z nich říkám *architektura programového kódu* a druhému *architektura dat*. Ještě jednou vás ale upozorňuji, že takto jsem si je pojmenovala jen já; v materiálech jiných autorů je nenajdete.

### Architektury programového kódu

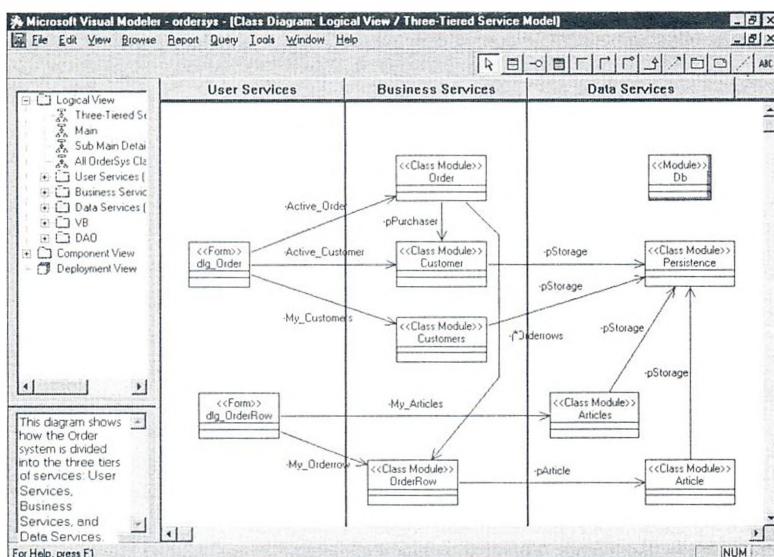
Modelu, kteremu zde říkám „architektura programového kódu“, se v literatuře říká různě: „aplikativní model“, „vrstvený přístup“ a „model služeb“. Architektura programového kódu popisuje způsob, jakým je logicky strukturován kód systému. Struktura programového kódu je do značné míry záležitostí vlastní implementace, a jako taková leží tudíž mimo rámec této knihy. Architektura kódu však může rozhodovat, ve kterém místě databázového schématu se budou implementovat omezení datové integrity; proto se jí zde budeme částečně věnovat, i když poněkud povrchně.

V počítačovém pravěku měly systémy monolitickou architekturu: byly to obrovské bloky programového kódu s minimální strukturou. Každý, kdo měl někdy tu smůlu, že byl nucen se pokusit o modifikaci (nebo dokonce o pochopení) nějakého takového monolitického systému libovolné úrovně složitosti, se již asi nikdy rád nepodívá na talíř špaget. Programátoři začali ale postupně do tohoto chaosu vnášet jistý řád a začali programový kód různým způsobem strukturovat do diskrétních komponent – do podprogramů (subrutin), modulů nebo objektů, podle možností konkrétního programovacího jazyka. Tento přístup má ale také jistý problém: zřejmě s ním sice nevytvoríme špagety, ale máme docela „šanci“ vytvořit jakési „tortellini“ – vzájemně nezávislé kousky programového kódu, které *nějakým* způsobem vzájemně komunikují a spolupracují, nikdo ale netuší, jak.

Aby se tyto moderní „těstoviny“ daly vůbec nějak rozumně zvládnout, snaží se řada vývojářů komponenty uspořádat do služeb, někdy nazývaných vrstvy, které provádějí různé úkony na jisté diskrétní logické úrovni. Jednotlivé vrstvy se dají uspořádat nespočetně mnoha různými způsoby. Nyní se společně podíváme na dva nejběžnější typy, a sice na třívrstvý a čtyřvrstvý model. Anglicky se kupodivu říká three-tiered model a four-layered model; neptejte se mě ale, proč se vrstvám říká jednou „tiers“ a podruhé „layers“. Osobně si myslím, že je to jenom kvůli „matení nepřitele“. (Ještě že česky říkáme „tiers“ i „layers“ shodně vrstvy. Pozn. překl.)

## Třívrstvý model

Ve třívrstvém modelu se jednotlivé komponenty rozdělují na Uživatelské služby, Aplikační služby a Datové služby. Tento model podporuje například nástroj Microsoft Visual Modeler – grafický modelovací nástroj, který je integrován v Microsoft Visual Studiu 6.0 a který vidíme na obrázku 10-1.



Obrázek 10-1 Microsoft Visual Modeler podporuje třívrstvý model

Komponenty programového kódu, které prezentují (zobrazují) informace uživateli a které reagují na jeho aktivity, se v třívrstvém modelu zařazují do vrstvy Uživatelských služeb. V této vrstvě se tedy skrývá celé uživatelské rozhraní. Vrstva Aplikačních služeb odpovídá za zajištění platnosti aplikačních pravidel a za ověření (verifikaci) vstupu zadанého uživatelem. Komponenty vrstvy Aplikačních služeb komunikují jak s vrstvou Uživatelských služeb, tak i s vrstvou Datových služeb. Za údržbu dat jsou pak odpovědné komponenty programového kódu z vrstvy Datových služeb, které komunikují pouze s vrstvou Aplikačních služeb.

Třívrstvý model je tedy docela jasný a Microsoft Visual Modeler je zajisté šikovný nástroj, jeho možnosti využití při skutečném vývoji jsou ale podle mého názoru poněkud problematické. Vždycky se totiž najdou určité typy funkcí, které se nedají jednoznačně přiřadit do žádné kon-

krétní vrstvy. Řekněme například, že určitou datovou položku potřebujeme před zobrazením uživateli odpovídajícím způsobem naformátovat. Rodné číslo osoby je třeba v databázi uloženo jako řetězec devíti číslic, zobrazovat se má ale ve formátu 99-99-99/9999. Patří toto formátování do vrstvy Uživatelských služeb, nebo do vrstvy Datových služeb? Našli bychom důvody pro každou z těchto dvou vrstev. Podobně se můžeme ptát na správu transakcí: patří do vrstvy Aplikačních služeb nebo Datových služeb? Při návrhu složitých systémů s hierarchicky uspořádanými daty a s datovými útvary nám tato rozhodování mohou docela pěkně zamotat hlavu.

Na tom, do které vrstvy tyto typy funkcí umístíte, v podstatě příliš nezáleží, tedy pokud jste opravdu důslední; přesně zde je ale zároveň slabé místo modelu. Jestliže se totiž musíte přizpůsobit určitým zvenčí definovaným konvencím – jako například „formátování patří do vrstvy Uživatelských služeb a vytváření hierarchických datových množin je součástí vrstvy Aplikačních služeb“ – narůstá v modelu velká režie, která dokáže převážit veškeré jeho výhody.

## Čtyřvrstvý model

Rozdělením architektury programového kódu nikoli do tří, nýbrž do čtyř vrstev se zbavíme řady problémů třívrstvého modelu. Ve čtyřvrstvém modelu, kterému se často říká také „vrstvený přístup“, se jednotlivé komponenty programového kódu rozdělují na vrstvu Uživatelského rozhraní, vrstvu Datového rozhraní, vrstvu Transakčního rozhraní a vrstvu Rozhraní externího přístupu. Jejich vzájemné vztahy ukazuje obrázek 10-2.

Vrstva Uživatelského rozhraní odpovídá vrstvě Uživatelských služeb z třívrstvého modelu. Tato vrstva je odpovědná za veškerou komunikaci s uživatelem; to znamená, že prostřednictvím různých objektů oken prezentuje potřebné informace uživateli, reaguje na změny stavu těchto objektů, jako je například změna velikosti formuláře, a konečně iniciuje požadavky uživatele.

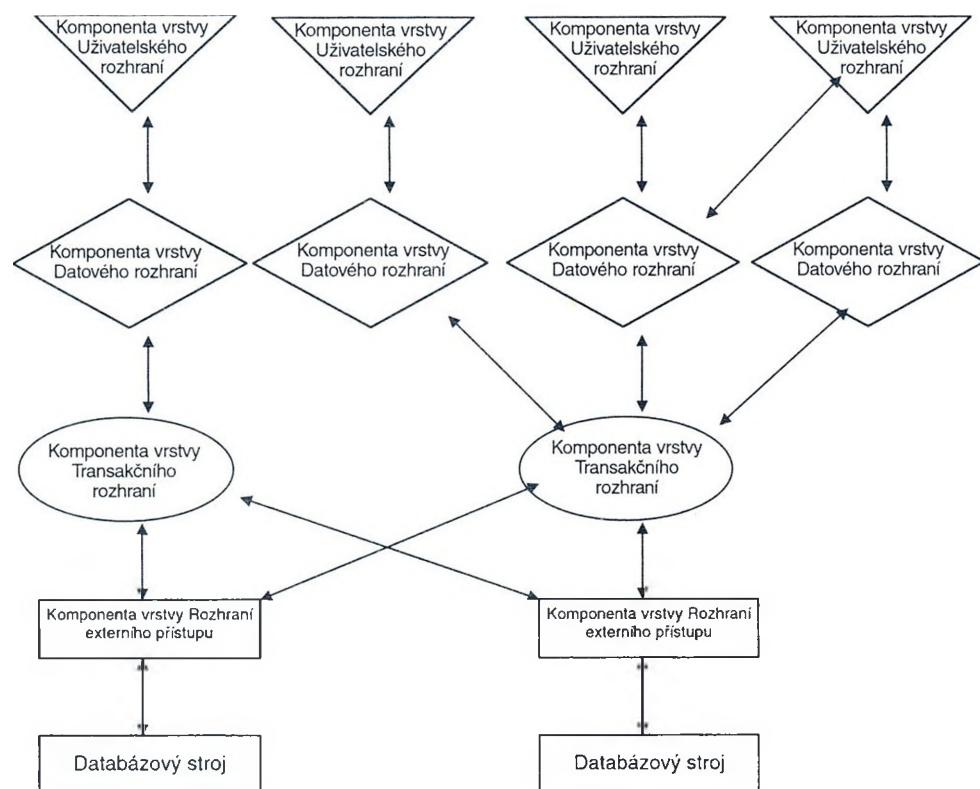
Druhá vrstva, vrstva Datového rozhraní, zajišťuje údržbu dat v paměti (nemá tedy na starosti trvalou správu dat, kterou zajišťuje vrstva Rozhraní externího přístupu a samotný databázový stroj, jak si ukážeme). Explicitně obsahuje většinu funkcí, jejichž zařazení by v třívrstvém modelu bylo sporné, tedy například formátování dat a vytváření virtuálních množin záznamů. (*Virtuální množina záznamů* existuje pouze v paměti a trvale se nikde neukládá.)

Drívá většina komponent vrstvy Datového rozhraní je úzce svázána s určitou konkrétní komponentou vrstvy Uživatelského rozhraní. Teoreticky však komponenta vrstvy Datového rozhraní může podporovat i několik různých komponent vrstvy Uživatelského rozhraní, jak ostatně vidíme na obrázku 10-2. Systém může tak například obsahovat formulář Správa údajů o zákazníkovi, který popisuje informace jen o jednom zákazníkovi, a formulář Přehled zákazníků s mnoha různými zákazníky. Oba tyto formuláře reprezentují entitu Zákazník, takže mohou sdílet jistý společný programový kód pro formátování a třeba pro ověření platnosti Čísla zákazníka; přesně to jsou funkce vrstvy Datového rozhraní.

Z fyzického pohledu odpovídá komponenta vrstvy Uživatelského rozhraní obvykle formuláři Microsoft Visual Basicu nebo Microsoft Accessu, přičemž s nimi spojená komponenta Datového rozhraní se zpravidla implementuje v modulu formuláře. Jisté procedury mohou být ale společné i pro několik formulářů; v takovém případě musíme procedury zcela přirozeně implementovat ve společném, sdíleném modulu.

Vrstva Datového rozhraní je odpovědná za validaci nebo ověření platnosti dat, neodpovídá však již za vlastní aplikační procesy. Součástí vrstvy Datového rozhraní bude tak například programový kód, který ověří, jestli je Číslo zákazníka, zadané v objednávce, v systému známé. Programový kód, který zabezpečuje určité definované pořadí událostí, tedy který například zabrání v expedici zboží pro zákazníka bez potvrzení jeho úvěrového limitu, však již náleží do vrstvy Transakčního rozhraní.

Vrstva Transakčního rozhraní koordinuje práci s daty v celé aplikaci. Komponenty této úrovně jsou odpovědné za sestavení a spuštění dotazů, za načítání informací z vrstvy Rozhraní externího přístupu, za zajištění aplikačních procesů a za zpracování chyb a narušení, které jím vrstva Rozhraní externího přístupu ohlásila.



Obrázek 10-2 Čtyřvrstvý model

Komponenty ve vrstvě Transakčního rozhraní se budou opakováně využívat častěji než komponenty z vrstvy Datového rozhraní. Komponenty vrstvy Transakčního rozhraní jsou vhodnými kandidáty pro implementaci jako objekty, jak ve Visual Basicu, tak i v Microsoft Accessu. Objekt Zákazník může například definovat metodu Aktualizuj, která se vyvolává z několika různých komponent vrstvy Datového rozhraní. Všimněte si přitom, že vrstvy Datového rozhraní by měla být (alespoň v ideálním případě) zcela nezávislá na komponentách vrstvy Uživatelského rozhraní, takže aktualizované hodnoty musíme ve volání uvést explicitně. Jinými slovy, volání bude mít následující tvar:

MůjZákazník.Aktualizuj KódZákazníka, JménoZákazníka

Metoda *Aktualizuj* vytvoří poté odpovídající příkaz UPDATE, předá jej do vrstvy Rozhraní externího přístupu ke zpracování a nakonec ošetří případné chybové stavy – buďto je přímo vyřeší, nebo je předá zpět v řetězci volání k zobrazení ve vrstvě Uživatelského rozhraní.

Vrstva Rozhraní externího přístupu je odpovědná za komunikaci mezi naší aplikací a externími zdroji dat. V databázových systémech tak komponenty programového kódu na této úrovni zajistují komunikaci s databázovým strojem. To znamená, že provádějí dotazy a jejich výsledky (včetně případných chybových zpráv) předávají zpět po řetězci volání komponent.

V ideálním případě bychom měli procedury na této úrovni navrhnout takovým způsobem, aby byly transakce izolovány od specifických konkrétního zvoleného databázového stroje. Aplikaci, která byla podle původního záměru postavena pro databázový stroj Microsoft Jet, by tak bylo teoreticky možné převést na platformu databázového stroje Microsoft SQL Server pouhou náhradou vrstvy Rozhraní externího přístupu za jinou. Skutečnost však již tak snadná není.

Vzpomeňte si, že za sestavování dotazů je odpovědná vrstva Transakčního rozhraní; vrstvy Rozhraní externího přístupu je pouze vykonává. Vzhledem k poměrně významným syntaktickým rozdílům mezi různými implementacemi jazyka SQL je vytvoření transakční vrstvy jen málokdy tak snadné. K vytvoření stejné množiny záznamů, kterou jsme nad databázovým strojem Microsoft Jet vygenerovali pomocí jediného příkazu (dotazu) TRANSFORM, potřebujeme tak v SQL Serveru posloupnost několika příkazů.

Pokud dokážeme všechny možné dotazy, které bude aplikace za svého běhu vykonávat, určit již dopředu, můžeme je zahrnout přímo do databázového schématu, čímž se problémům spojeným se syntaxí zcela vyhneme. Na tomto místě nám přitom mohou velice dobře pomoci parametrické dotazy. To znamená, že v aplikaci nemusíme třeba příkaz SQL pro výběr zákazníka se jménem Honzátko sestavovat za běhu – stačí řetězec se jménem „Honzátko“ předat do nějakého předem vytvořeného, již existujícího dotazu jako parametr.

Postihnout všechny potřebné dotazy takto dopředu ale naneštěstí není vždy možné, a to zejména pokud uživateli nabízíme v aplikaci možnost sestavování jednorázových ad hoc dotazů. V tomto případě je téměř nemožné vrstvu Transakčního rozhraní zcela izolovat. (Musíme se přiznat, že se mi vlastně nepodařilo dosud najít nějaké „konečné řešení“ tohoto problému: jestliže na nějaké přijdete, budu vám neskonale vděčná, pokud se o ně se mnou podělít.) Kromě toho to znamená, že do komponent vrstvy Datového rozhraní musíme zapsat navíc jistý podmínkový kód, jako například:

```

dbsStroj = mojeData.JménoStroje
Select Case dbsStroj
    Case "SQLServer"
        ' sestavit dotaz v dialekту SQL Serveru
    Case "Jet"
        ' sestavit dotaz pro databázový stroj Microsoft Jet
    Case Else
        ' vrátit do vrstvy Datového rozhraní chybu "neznámý stroj"
End Select

```

Pokud má daná aplikace podle zadání podporovat jen jeden jediný databázový stroj, může být lákavé sloučit vrstvu Transakčního rozhraní a vrstvu Rozhraní externího přístupu do jediné. To vám ale nedoporučuji. Navrhnut vrstvu Rozhraní externího přístupu zabere sice určitý čas, není to ale nikterak obtížné a jakmile tuto vrstvu napišeme, ušetříme si ve zbytku systému stovky zbytečných řádků programového kódu. Navíc, jestliže jednou napišeme komponentu, která komunikuje například s SQL Serverem verze 7.0 prostřednictvím rozhraní ADO 2.0, nemusíme psát její funkce již nikdy znova. Zcela bez úprav ji můžeme využít v jakémkoli jiném systému, který někdy později vytvoříme. Novou komponentu vrstvy Rozhraní externího přístupu budeme potřebovat v jediném případě: pokud se změní podkladový databázový stroj nebo objektový model.

### **Architektury programového kódu a databázové schéma**

Zvolená architektura programového kódu ovlivňuje databázové schéma ve dvou základních oblastech: je to izolace vrstvy Rozhraní externího přístupu (respektive vrstvy Datových služeb v třívrstvém modelu) a validace dat. O izolování vrstvy Rozhraní externího přístupu od změn v databázových strojích jsme již hovořili – v takovém případě musíme dopředu rozhodnout o používaných dotazech a zahrnout je přímo do databázového schématu. Vedlejším efektem tohoto přístupu je mimo jiné zlepšení rychlosti zpracování, a to někdy i dosti významné. Validace (ověření platnosti) dat je o něco závludnější otázka. „Co“ a „jak“ si o validaci dat řekneme podrobněji v kapitole 16. Zde se pokusíme odpovědět na otázky „kdy“ a „kde“.

Někteří návrháři prosazují začlenění veškerých funkcí pro validaci dat přímo do samotného databázového stroje. Tento přístup není zdaleka nesmyslný: veškerá omezení datové integrity a aplikacní pravidla se implementují v jediném místě, kde se podle potřeby dají také poměrně snadno upravit. Uvedený postup má ale naneštěstí i svoje problémy.

Tak za prvé, určitá pravidla se na úrovni databázového stroje implementovat *nedají*. Protože například databázový stroj Microsoft Jet nezná spouště, nemůžeme v něm zajistit pravidlo, podle kterého se po vytvoření záznamu nesmí změnit hodnota jeho primárního klíče. Ani v SQL Serveru, který má v této oblasti většinou mnohem bohatší možnosti, se zdaleka ne všechna pravidla dají implementovat přímo v databázovém stroji.

Za druhé, při validaci dat je nutné čekat na okamžik odeslání dat do databázového stroje, což může zhoršit praktickou použitelnost systému. Obecně je totiž nutné validovat data co nejdříve, nejlépe ihned po jejich zadání. V některých případech to znamená zahájit validaci již s okamžikem stisku první klávesy; jako příklad si uvedlme číselné pole, do kterého uživatel nesmí zadávat písmena a jiné abecední znaky. V ostatních případech musíme data validovat po opuštění vstupního pole, respektive po zadání posledního z jisté definované sekvence polí; druhý jmenovaný

případ platí pro složitější situace, kdy například potřebujeme zajistit pravidlo, podle něhož musí být PožadovanéDatum Dodání pozdější nebo rovno než DatumObjednávky.

Odesílat data do databázového stroje s každou jednotlivou stisknutou klávesou je přímo žalostně pomalé, a to i v samostatné (izolované) aplikaci, která běží na jediném počítači. Jestliže navíc mechanismus validace vyžaduje komunikaci neboli výměnu informací s databázovým strojem přes lokální síť, nebo dokonce – pánbůh chraň – přes rozlehlou síť či Internet, poběží aplikace prakticky určitě tak pomalu, že se uživatelé ještě rádi vrátí ke staříčké papírové kartotéce (pokud ji mezitím neodvezli na skládku).

Jediným řešením je tedy odeslat data do databázového stroje k ověření (validaci) až po dokončení celého záznamu. V tomto okamžiku však již uživatel věnuje pozornost zpravidla jinému, dalšímu úkolu. Ohlásit problém, který vznikl při zadávání dat někdy před 10 minutami, již nemá smysl – uživatele to jen zbytečně mate a ruší jej v další práci.

Pokud má systém mít co nejlepší odezvu a zároveň má být co nejlépe použitelný, musíme validaci dat implementovat přímo v aplikaci. Jestliže přitom danou databázi využívá pouze jediná aplikace a požadavky na validaci jsou relativně stabilní, mohli bychom se rozhodnout celou validaci dat implementovat *zájbrazdě* na úrovni aplikace a na validaci v databázovém stroji zcela zapomenout. Tím bychom se zajisté zbavili určité dvojí práce, tento postup je ale na druhé straně poměrně nebezpečný.

Jestliže někdy v budoucnu postavíme nad stejnou databází nějakou další aplikaci, můžeme i s dobrým úmyslem nechtěně podkopat integritu databáze – i zde se potvrzuje, že dobrými úmysly je dlážděna také cesta do pekel. Zranitelná je ale i databáze, kterou nikdy nebudeme sdílet s žádnou jinou aplikací; hrozí jí nebezpečí od uživatelů, kteří s daty manipulují pomocí různých ad hoc nástrojů, jako je Microsoft Access nebo SQL Server Enterprise Manager. Těmto zásahům můžeme zabránit pomocí přísného bezpečnostního modelu, kterým zakážeme veškeré změny v datech, kromě změn provedených samotnou aplikací; za bezpečnost zde ovšem platíme nepříjemným omezením přístupu k datům.

Z těchto všech důvodů se tedy domnívám, že nejlepší je implementovat validaci dat zároveň jak v aplikaci, tak i v databázovém schématu. Microsoft Access to dělá automaticky: jestliže pro určité pole nadefinujeme validační pravidlo na úrovni tabulky a poté toto pole přetáhneme na svázany formulář, zdědí formulář i původně definované validační pravidlo.

Ve starších verzích Microsoft Accessu před verzí 2000 je ale tato funkce naneštěstí zároveň dobrým příkladem problému s opakovánou validací. Jestliže po vložení pole do svázанého formuláře změníme příslušné validační pravidlo na úrovni tabulky, ve formuláři se tyto změny nepromítnou. Microsoft Access 2000 již pravidlo aktualizuje, ve Visual Basicu však tento problém zůstává. Jestliže se na toto pole budeme odkazovat z několika různých formulářů (ať už ve stejné aplikaci nebo v několika různých aplikacích), musíme validační pravidla v každém formuláři upravit ručně (nebo lépe v komponentách vrstvy Datového rozhraní, které jednotlivé formuláře podporují).

S ohledem na tento problém zjišťují někteří vývojáři validační pravidla až za běhu, dotazem vůči databázovému stroji. Tento postup znamená jistý nezanedbatelný objem režie; jestliže se ale budou validační pravidla měnit poměrně často, mohou tuto režii zcela vyvážit snazší úpravy pravidel, která se nacházejí na jediném místě.

Validační pravidla můžeme z databázového stroje načít při prvotním spuštění aplikace, při zavedení formuláře, nebo před aktualizací každého jednotlivého záznamu. Osobně doporučuji je zjistovat při zavedení formuláře. Pokud totiž budete tyto operace provádět hned po spuštění aplikace, můžete zároveň načítat i zcela irrelevantní informace, které se týkají formulářů, s nimiž uživatel nakonec vůbec nebude pracovat. Jestliže si je zjistíte vždy s aktualizací každého záznamu, budete si absolutně jisti, že pracujete vždy s těmi nejaktuálnějšími pravidly; na druhé straně to ale znamená časově náročné dotazování schématu s každým jednotlivým záznamem – a upřímně řečeno, kolik systémů může být až tak moc proměnlivých?

Pravidlo se pochopitelně může změnit právě v okamžiku, kdy má uživatel otevřený příslušný formulář; jestliže nyní uživatel náhodou zadal data, která původnímu pravidlu vyhovují, zatímco nové poruší, odmítne je samotný databázový stroj, takže k žádné velké škodě nedojde. Kromě toho, už samotná myšlenka „rozvrtání“ databázového schématu v živém systému, se kterým právě někdo pracuje, si koleduje o průšvih.

### **Architektury dat**

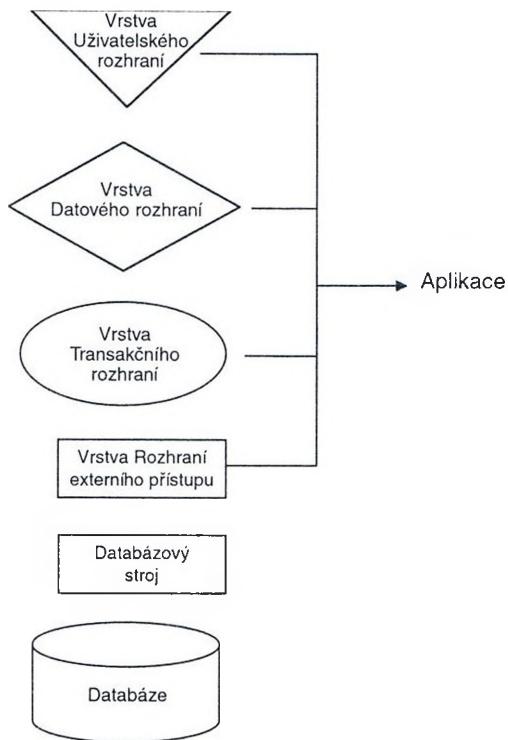
Kromě struktury programového kódu v novém systému se musíme rozhodnout také pro vhodnou architekturu dat. Z kapitoly 1 si zajisté vzpomínáte, že každý databázový systém se skládá z řady diskrétních komponent: jednak je to samotná aplikace, dále databázový stroj a databáze. (Celé schéma ukazuje obrázek 1-1 na straně 5.) Ve čtyřvrstvém modelu kódu můžeme tuto strukturu dále rozčlenit; výsledek vidíme na obrázku 10-3.

Při volbě vhodné architektury dat pro novou aplikaci se musíme rozhodnout, kde budou jednotlivé vrstvy existovat. Teoreticky může být každá vrstva (nebo dokonce každá jednotlivá komponenta) umístěna na jiném počítači a mohou vzájemně komunikovat přes nějakou síť. Na druhém konci této pomyslné stupnice se nachází řešení, při němž jsou všechny komponenty umístěny na jediném, samostatném počítači. V praxi se pak používá několik málo víceméně standardních konfigurací, které se v různých situacích ukazují být efektivní; všem těmto architekturám se nyní budeme věnovat podrobněji.

### **Jednovrstvá architektura**

Každé logické skupině komponent v architektuře dat se říká „vrstva“ (tier). Nejjednodušší architekturou je samozřejmě jednovrstvý systém, v němž se veškeré komponenty nacházejí v jedné jediné logické vrstvě; nejjednodušší verzí jednovrstvé datové architektury je pak samostatný (izolovaný) systém.

V samostatném, izolovaném systému se všechny komponenty nacházejí na jediném počítači a jsou tím pádem dostupné jen tomu uživateli, který na tomto počítači fyzicky pracuje. Počítač může být sice „náhodou“ připojen k počítačové síti nebo k Internetu, databázový systém však pro jiné uživatele dostupný není. Veškeré zpracování probíhá na tomto lokálním počítači a veškerá data jsou uložena lokálně, takže jediným omezením výkonu systému jsou fyzické možnosti tohoto počítače – rychlosť jeho procesoru a velikost paměti. Samostatné systémy se zpravidla vyznačují velkou spotřebou paměti; to je zároveň jedním z důvodů, proč se obvykle dává přednost jiným konfiguracím.



Obrázek 10-3 Databázový systém se skládá ze šesti diskrétních vrstev

Většina samostatných systémů používá databázový stroj Microsoft Jet. Na jediném počítači bychom docela určitě mohli implementovat i systém s SQL Serverem, z jistých důvodů, které si podrobně rozebereme za chvíli, je však dosti diskutabilní, jestli by se takový systém dal ještě považovat za jednovrstvý. Výjimkou je stroj Microsoft Data Engine (MSDE), který je distribuován s Microsoft Accessem 2000. MSDE je něco jako známý „semafor služby SQL Serveru“, běží však na jednovrstvé architektuře.

Poměrně běžnou variací jednovrstvé architektury je síťová databáze. V tomto modelu je databáze (nebo alespoň některá její část) fyzicky umístěna na vzdáleném počítači připojeném přes síť, ale veškeré zpracování probíhá výhradně lokálně.

### Poznámka

*Nesnažte se umístit na síťový disk samotnou aplikaci. Je to sice také teoreticky možné, rozhodně vám to ale nemohu doporučit, protože tím vzniká neúměrné zatížení sítě. Namísto toho ponechejte aplikaci na lokálním počítači a k datům umístěným na síti přistupujte prostřednictvím propojených tabulek.*

Síťová databáze – kterou můžeme vytvořit pouze s pomocí databázového stroje Microsoft Jet, nikoli v SQL Serveru – umožňuje přistupovat k datům současně více uživatelům. Maximální počet uživatelů, kteří mohou současně pracovat s databází stroje Microsoft Jet, je teoreticky roven 255. V praxi dosažený maximální počet však závisí na tom, co uživatelé s daty provádějí, a zejména také na celkové efektivitě systému. Dvacet lidí s hbitými prsty, kteří do systému velice rychle pořizují data, znamenají zcela zřejmě větší zátěž než třeba padesát jiných lidí, kteří si prohlížejí výsledky prodeje a promýšlejí strategie dalšího prodeje výrobků.

Snížení míry zatížení sítě má největší přímý dopad na databázové schéma. Nezapomeňte totiž, že se veškeré zpracování odehrává stále na lokálním počítači; s počítačem, na kterém je uložena databáze, tak v jistém slova smyslu pracujeme jen jako s jakýmsi vzdáleným pevným diskem. Doba odezvy při komunikaci přes síť je ale zpravidla mnohem pomalejší než přímý přístup k datům na lokálním disku. Síť má navíc dosti omezenou kapacitu, o kterou musí vzájemně soupeřit všichni uživatelé systému. To znamená, že je vhodné co nejvíce snížit objem informací zasílaných po síti tam a zpět. Opět se jedná z velké části o implementační záležitost; jestliže se budete implementací zabývat vy sami, doporučuji vám podívat se do literatury uvedené v seznamu na konci knihy.

Rychlosť komunikace po síti mohou ale přímo ovlivnit dva důležité aspekty databázového schématu: jednak je to umístění databázových objektů a dále používání vhodných indexů. Už jsem se zmínila o tom, jak je důležité mít objekty uživatelského rozhraní uloženy zásadně lokálně. Kromě toho je vhodné zkopirovat na počítač uživatele také některá data, která se příliš často nemění.

Poměrně stabilní bude například seznam výrobků, na který se zároveň uživatelé budou zpravidla velice často odkazovat. Jestliže tabulka Výrobky není příliš rozsáhlá (několik megabajtů je ještě přijatelných, pokud dosahuje gigabajtové velikosti, nemá již tato úvaha smysl), můžeme ji zkopirovat zvlášť na počítač každého uživatele. Ve většině situací se tím sníží objem provozu na síti a zlepší se výkon systému. Samozřejmě musíme zároveň definovat odpovídající mechanismus pravidelné aktualizace dat; to ale není žádný velký problém.

Dobrými kandidáty pro lokální uložení jsou také seznamy poštovních směrovacích čísel, seznamy okresů, států a třeba regionálních poboček organizace; jsou totiž obvykle poměrně malé a stabilní. Zároveň se na ně v aplikaci provádějí časté odkazy. (Tabulku, kterou využije systémový administrátor jen jednou do roka, zcela zřejmě nemá smysl kopírovat na všechny pracovní stanice uživatelů.) Objednávky prodeje nebo seznamy zákazníků či studentů oproti tomu k lokálnímu uložení obvykle vhodné *nejsou*. Informace v takových množinách záznamů se velice rychle mění, přičemž základním smyslem síťové databáze je právě sdílení jedné společné neaktuálnější verze množiny záznamů.

Druhým místem, kde můžeme databázové schéma ovlivnit výkonnost sítě, jsou vhodné indexy. Index si můžeme představit jako jakousi miniaturní tabulku, v níž se udržuje jisté definované pořadí. Obsahuje jen ta pole, která jsou nezbytná pro uspořádání záznamů, a navíc jistý „ukazatel“ na záznam ve skutečné tabulce; strukturu indexu názorně ukazuje obrázek 10-4.

Všimněte si, že s pojmem „ukazatel“ zde zacházíme poněkud volně. Tento ukazatel neznamená totéž co pojmy paměťový ukazatel nebo objektový ukazatel, které se používají v oblasti programování. Fyzická implementace indexů ve skutečnosti téměř vůbec neodpovídá modelu z výše uvedeného obrázku, pro vysvětlení je přesto tento model docela dobrý. Pokud se vám *chce* se nimrat

**Výrobky - Tabulka**

Cílo výrobku	Název výrobku	Dodavatel
65	Louisiana Fiery Hot Pepper Sauce	New Orleans Cajun Delights
53	Perth Pasties	G'day, Mate
60	Camembert Pierrot	Gai pâturage
12	Queso Manchego La Pastora	Cooperativa de Quesos 'Las Cabras'
18	Cainarvon Tigers	Pavlova, Ltd.

**Index: Výrobky podle názvu**

- Camembert Pierrot
- Carnarvon Tigers
- Louisiana Fiery Hot Pepper Sauce
- Perth Pasties
- Queso Manchego La Pastora

Obrázek 10-4 Index je jakási miniaturní tabulka, jejíž záznamy jsou uloženy v jistém definovaném pořadí

v těchto záležitostech hlouběji, doporučuji vám začít například s knihou *Microsoft Jet Database Engine Programmer's Guide*.

Změna skutečného, fyzického pořadí záznamů základní tabulky je ve většině situací příliš časově náročná, a tudíž nežádoucí; databázový stroj Microsoft Jet proto při uspořádání tabulky ve skutečnosti řadí jen indexový soubor. Taková operace je (obvykle) podstatně rychlejší; díky popsánému řešení můžeme navíc k tabulce snadno přistupovat v několika různých uspořádáních současně, protože nad jednou tabulkou je možné pochopitelně udržovat i více než jeden index.

SQL Server definuje jistý speciální druh indexu, kterému se říká *clusterován index* a který řídí skutečné, fyzické uspořádání dat. Každá tabulka může mít tím pádem jen jeden clusterován index.

Z pohledu výkonu systému jsou indexy opravdu velice důležité: v řadě případů může totiž databázový stroj Microsoft Jet provádět žádanou operaci jen s pomocí indexu, a nemusí tudíž zasahovat do samotné základní tabulky. Tato skutečnost vede často k dosti znatelnému urychlení i u samostatných systémů (protože i čtení dat z lokálního disku zabere určitý čas), v síťově orientovaných řešeních může být dosažené urychlení přímo zásadního významu.

Jako příklad uvažujme tabulku Zákazníci se 100 000 záznamy, z nichž každý zabírá 1 500 bajtů. Aplikace potřebuje v této tabulce vyhledat určitý konkrétní záznam, dejme tomu záznam zákazníka Stavební společnost Tibelka, jejíž ID Zákazníka je roven STSTIHE. Vykonáme tedy následující příkaz:

```
SELECT * FROM Zákazníci WHERE IDZákazníka = "STSTIHE"
```

Pokud by v tabulce nebyl deklarován ani index, ani primární klíč, musel by databázový stroj Microsoft Jet po zadání takového dotazu načíst postupně všech 100 000 záznamů a zjistit, který z nich odpovídá zadané podmínce. To znamená, že by se přes síť posílalo nejméně 150 MB dat. Pokud je ovšem pole ID Zákazníka indexované, a to buďto explicitně, nebo pokud je deklarováno jako primární klíč, stačí databázovému stroji Jet načíst pouze index, který bude mít nejspíše velikost několika málo kilobajtů, a podle něj již může rychle vyhledat správný záznam základní tabulky.

Indexy se mohou postarat opravdu o velmi výrazné urychlení aplikace, samozřejmě ale ani ony nejsou zadarmo. Práce s indexy znamená totiž jistou režii navíc: při každém přidání nového záznamu nebo při aktualizaci stávajícího musí Microsoft Jet odpovídajícím způsobem upravit i všechny indexy příslušné tabulky. Obvykle je tato režie poměrně zanedbatelná, jestliže ale začneme v jedné určité tabulce používat příliš mnoho indexů, mohou mít na výkonnost systému dosti nepříznivý dopad. V extrémním případě by čas nezbytný pro údržbu indexů mohl i převyšit čas, který pomocí nich ušetříme.

## Dvouvrstvé architektury

V dvouvrstvé architektuře se databáze a databázový stroj nacházejí na vzdáleném počítači. Obě tyto součásti systému mohou být buďto na jednom stejném počítači, nebo na dvou různých počítačích. Databáze může být ve skutečnosti rozprostřena i mezi několik různých fyzických počítačů; z logického pohledu bude systém přesto stále dvouvrstvý. Tuto architekturu můžeme využít pouze v systému Microsoft SQL Serveru nebo u jiného databázového serveru, jako je například Oracle; databázový stroj Microsoft Jet ji nepodporuje.

Na první pohled se možná zdá, že mezi síťovou databází a dvouvrstvým systémem není příliš velký rozdíl. Budto použijeme Microsoft Jet lokálně, nebo SQL Server na vzdáleném počítači – vždyť je to skoro totéž? Podstatný rozdíl zde spočívá v tom, že u síťové databáze se veškeré zpracování provádí na lokální pracovní stanici, zatímco ve dvouvrstvém systému jsou potřebné činnosti rozděleny mezi dva různé procesory. Pracovní stanice je odpovědná za správné ošetření interakce s uživatelem a vzdálený počítač se stará o přístup k datům. SQL Server tak provádí veškeré manipulace s daty (to znamená, že provádí i dotazy) a výsledky „naservruje“ klientské pracovní stanici. Z toho důvodu se dvouvrstvé databázové systémy označují častěji jako architektura typu *klient/server*.

Abychom si celý rozdíl mezi síťovou databází a dvouvrstvým systémem ještě lépe ujasnili, vrátíme se k následujícímu příkazu jazyka SQL, o němž jsme hovořili v souvislosti s významem indexů:

```
SELECT * FROM Zákazníci WHERE IDZákazníka = "STSTIHE"
```

V síťové databázi provede veškeré zpracování databázový stroj Jet: to znamená, že nejprve načte index (pokud nějaký existuje), vyhledá podle něj správný záznam a poté jej převezmé. V systému s architekturou klient/server odešle aplikace oproti tomu uvedený příkaz ke zpracování na SQL Server a jako výsledek dostane správný záznam (respektive záznamy). (Celá situace je samozřejmě ve skutečnosti výrazně složitější, protože se provádí mnohem více kroků, pro pochopení ale tento jednoduchý model stačí.)

U takového jednoduchého požadavku může být rychlosť zpracování v obou architekturách natakně dobrá, že rozdíl mezi síťovou databází a databází typu klient/server ani nepostřehneme. Systém klient/server může být dokonce za jistých okolností pomalejší. U aplikací se složitými dotazy a s mnoha aktivními uživateli vede ale systém klient/server k výraznému zlepšení celkové výkonnosti a odezvy.

Odezva aplikace se v systému klient/server může zlepšit díky tomu, že zatímco server pilně počítá a připravuje výsledky příkazu, může se pracovní stanice věnovat něčemu jinému, například reagovat na další požadavky uživatele. Platí přitom i opačná úvaha: jestliže je pracovní stanice zaprázdlněna a reaguje na povely uživatele (nebo třeba jen čeká), než uživatel něco udělá, aby na

jeho činnost mohla rychle reagovat), má databázový server volné ruce a může klidně zpracovávat jiné požadavky. SQL Server je mnohem propracovanější než databázový stroj Microsoft Jet v řadě různých ohledů, systémy typu klient/server dokáží ale podporovat větší počet uživatelů než síťová databáze právě díky této lepší schopnosti odezvy. Je to docela pěkný příklad situace, kdy je celek větší než prostý součet jeho částí.

Má-li systém architektury klient/server správně pracovat a má-li mít vůbec smysl, musíme co nejvíce objem zpracování dat přesunout na server. Zatímco v síťové databázi je často vhodné ukládat si dotazy lokálně, ve dvouvrstvém systému klient/server by měly zůstat na serveru.

Jestliže se budete připojovat k SQL Serveru prostřednictvím Microsoft Accessu, dávejte si dobrý pozor na příkazy, které se budou provádět lokálně. Příkaz SELECT, jenž obsahuje uživatelsky definovanou funkci, bude například vykonávat samotný Access a nebude se předávat na SQL Server, protože SQL Server volání funkcí v příkazech SELECT nepodporuje (tyto funkce ale nepodporuje ani databázový stroj Microsoft Jet, takže dotaz, který obsahuje uživatelsky definovanou funkci, není možné vyvolat například z Visual Basicu, přestože je dotaz uložen v souboru .mdb).

Z pohledu implementace je třeba při výstavbě databází architektury klient/server pochopitelně zvážit mnohem více různých otázek. Opět vás odkáží na řadu vynikajících knih, které se tomuto tématu věnují a z nichž některé jsou uvedeny v seznamu literatury na konci této knížky.

## **N-vrstvé architektury**

Rozdelením provozní zátěže zpracování mezi dva systémy můžeme ve dvouvrstvém systému, je-li správně implementován, výrazně zlepšit celkovou výkonnost a odezvu aplikace. Podobné výhody může mít i další rozdelení zátěže mezi více systémů. Jestliže se nyní vrátíme k naší čtyřvrstvé architektuře programového kódu, kterou jsme viděli na obrázku 10-3 (strana 151), pak můžeme říci, že se na další mezilehlé systémy zpravidla distribuují komponenty vrstvy Transakčního rozhraní a komponenty vrstvy Rozhraní externího přístupu.

Podle všeho však naneštěstí složitost takové implementace roste exponenciálně. Při přechodu na tři nebo více logických vrstev se musíme více a více potýkat s problémy, jako je konektivita, bezpečnost, správa procesů apod. Protože složitost takových systémů si často vyžaduje doplnění dalších serverů různého typu, jako je například Microsoft Transaction Server, říká se jim obvykle „n-vrstvé systémy“. (S fyzickými vrstvami tedy zcela zřejmě přestaváme počítat u čísla tří, podobně jako s manželkami.)

Uvedená složitost implementace je ale naštěstí jen – otázkou implementace. Vývojový tým třeba před těmito otázkami uteče a vrhne se na kariéru pletení proutěných košíků, návrh databáze však n-vrstvá architektura sotva ovlivní. Logické úrovně architektury kódů je třeba velice přísně rozlišovat, avšak databázové schéma, které správně funguje ve dvouvrstvém prostředí, se docela klidně může beze sebemenší změny rozšířit i na n-vrstev.

## **Internetové a intranetové architektury**

Provoz databáze přes Internet nebo intranet je v podstatě speciálním případem n-vrstvé architektury. Konkrétní podoba použité technologie může být různá – jako přenosový protokol slouží HTTP a uživatelským rozhraním je spíše Internet Explorer než Microsoft Access – z logického pohledu jsou ale tyto architektury velice podobné.

Nejdůležitější rozdíl mezi databázovým systémem provozovaným na Internetu a systémem instalovaným v klasickém prostředí je ten, že Internet je anonymní prostředí (neudržuje žádný vnitřní stav neboli identitu uživatele). V běžném prostředí klient/Server si tak aplikace při svém prvním spuštění vyžádá od uživatele jméno a heslo a poté se s pomocí těchto informací připojí k SQL Serveru. Nově ustavené spojení (za předpokladu, že je zadáno uživatelské jméno a heslo správné) již server obvykle udržuje aktivní po celou dobu trvání seance. Protože tedy spojení s databází je trvalé, server bezpečně ví, kdo je jeho klientem, a jestliže klient odešle nějaký požadavek, může se podle toho odpovídajícím způsobem zařídit. Této informaci „vím, kdo jsi“ se říká *identita* (anglicky state, mohli bychom říci také vnitřní stav aplikace) a udržuje ji databázový server.

V systému, provozovaném v prostředí Internetu, si však již databázový server žádné informace o identitě neudržuje. Aplikace musí s každým novým požadavkem vůči databázovému serveru znovu ustavit spojení a musí znova prokázat svoji totožnost. Jakmile databázový server na zasláný požadavek odpoví, zapomene na aplikaci, která mu jej původně podala.

Tento malý objem režie, spojené s nutností vytvořit pro každý požadavek nové spojení, má ve většině situací jen velmi malý dopad na výkonnost databázového systému a v žádném případě neovlivňuje databázové schéma. Anonymní („bezstavová“) povaha Internetu má však ještě jeden významný důsledek, který již má podstatný dopad do aplikace a může si vynutit i změny v databázovém schématu.

Cílem většiny internetových aplikací je takzvaný *tenký klient*. To znamená, že aplikace má na klientu provádět co nejmenší část zpracování, zpravidla pouze činnosti spojené s uživatelským rozhraním. Uvažujme ale dotaz, který vrací velké množství záznamů – více záznamů, než kolik se dá rozumně zobrazit na jediné obrazovce. V klasické aplikaci uložíme takovou výslednou množinu do cache, a to buďto na klientu, nebo na serveru.

V internetové aplikaci se ale výsledky na straně serveru do žádné cache uložit nedají, protože server jednoduše neví, kam bude posílat další dávku výsledků. Pokud je ukládáme do cache na straně klienta, musí u něj běžet také odpovídající komponenty pro zpracování dat (tedy komponenty vrstev Transakčního rozhraní a komponenty Rozhraní externího přístupu) – a tyto komponenty rozhodně nejsou „tenké“. Jsou opravdu docela „vypasené“.

Rozhraní ActiveX Data Objects (ADO) nabízí pro ošetření těchto situací speciální mechanismus, kterému se říká *stránkování*. Aplikace si ještě před odesláním dotazu může ve vlastnosti PageSize objektu množiny záznamů *Recordset* stanovit počet záznamů, které se mají vrátit najednou. Jestliže tedy například do této vlastnosti přiřadíme hodnotu 14, budou se záznamy vracet v dávkách po čtrnácti. Vlastnost AbsolutePage pak určuje konkrétní požadovanou stránku, zatímco vlastnost PageCount vrací celkový počet stránek. Tento mechanismus funguje podobně jako klausule TOP N standardního příkazu SELECT jazyka SQL, zde však máme navíc možnost si říci o „prostředních N“ záznamů.

Na straně serveru stránkování znamená, že se při každém požadavku uživateli o novou stránku musí znova provádět původní dotaz. U dotazů s rozumně rychlou odezvou to není žádný problém. Složitý dotaz, jehož výpočet je poměrně pomalý, to však problémy znamená, a to dosti velké. Aplikace, ve které musí jeden uživatel čekat několik minut na odpověď, je přijatelná jen s velkými výhradami. Jestliže s aplikací pracuje několik tisíc uživatelů, kteří musí na každých dal-

ších pět záznamů čekat několik minut, zajisté ji brzy vyhodí do popelnice – nic jiného si ostatně ani nezaslouží.

Jestliže se v internetové aplikaci setkáme se složitým dotazem, máme v zásadě několik možností, jak jej vyřešit. První, a ve většině případů také nejvhodnější řešení spočívá v dokonalé optimalizaci dotazu. Pokuste se vytvořit dočasné tabulky, denormalizovat data a vůbec udělat cokoliv, čím se dá doba odezvy aplikace srazit na přijatelnou úroveň.

Pokud taková optimalizace není možná, nemáme někdy opravdu jinou možnost než přesunout určité komponenty pro zpracování dat na klienta a vytvořit tak fakticky „tučného“ (překrmeného) klienta. V této architektuře již můžeme výsledky dotazu ukládat do cache na straně klienta, takže v jistém slova smyslu napodobíme prostředí podobné síťové databázi (s několika málo komplikacemi).

Obecně ale platí, že tučný klient se hodí spíše do prostředí intranetu než Internetu, nemluvě o architektuře Microsoft Distributed Networked Architecture (DNA). Řada lidí má zcela pochopitelnou averzi ke stahování komponent s programovým kódem, ve veřejně dostupné aplikaci je však tato situace méně pravděpodobná. Jestliže se jí přesto nevyhnete, můžete jen doufat, že nabízíte natoží hodnotný obsah, že převáží i odpor uživatelů.

## Součásti databázového schématu

Po dokončení myšlenkového (konceptuálního) datového modelu a po rozhodnutí vhodné systémové architektury již máme většinu informací, které potřebujeme k sestavení databázového schématu. Databázové schéma je přitom popis datových objektů obsažených v budoucí databázi. Kromě datové architektury samostatné aplikace pak databázové schéma definuje také místo, kde se každý jednotlivý objekt bude nacházet.

Jestliže budeme nový systém implementovat v Microsoft Accessu, bude databázové schéma obsahovat definici jednotlivých tabulek, dotazů a vztahů. Nebude však již popisovat formuláře, sestavy a komponenty programového kódu z navrhovaného systému, přestože i ty jsou uloženy v souboru .mdb databáze. U systému implementovaného v Microsoft SQL Serveru obsahuje databázové schéma definice tabulek, pohledů, uložených procedur a spouští v databázi.

### Definice tabulek a vztahů

Definice fyzických tabulek v databázovém schématu se odvozuje přímo od myšlenkového datového modelu. Z entit se stávají tabulky a z jejich atributů se vytvářejí pole tabulek. Většinou se jedná opravdu o jednoduchý, přímý převod. Jediným místem, kde si musíme dávat trošku větší pozor, jsou omezení, vztahy a indexy.

### Omezení

V rámci definice myšlenkového datového modelu jsme definovali také omezení entit, atributů a domén (oborů hodnot). Jestli budeme tato omezení implementovat v databázovém schématu, to záleží (jak jsme ostatně viděli) na zvolené systémové architektuře. A jak jsem již řekla, někteří vývojáři implementují raději všechna omezení u čtyřvrstvého modelu pouze ve vrstvě Datového rozhraní a ve vrstvě Transakčního rozhraní, respektive ve vrstvě Aplikačních služeb u třívrstvého modelu.

Já doporučuji ve většině situací implementovat odpovídající omezení na obou úrovních. Jestliže tedy budete se mnou souhlasit a rozhodnete se omezení zahrnout do samotné databáze, budete je definovat právě jako součást databázového schématu. O implementaci datové integrity jsme podrobně hovořili v kapitole 4, nyní je ale načase krátké opakování.

Z většiny omezení domén a omezení atributů se v databázovém schématu stávají omezení na úrovni pole; v Microsoft Accessu jsou to pak obvykle ověřovací (validační) pravidla. Microsoft Access podporuje také omezující klausuli CHECK, známou z SQL Serveru; tu využijeme v případě, že se rozhodneme vytvořit databázi nikoli pomocí DAO nebo pomocí uživatelského rozhraní Accessu, ale pomocí příkazů jazyka SQL.

Omezení na úrovni entity se zpravidla převádějí na omezení tabulek, a to opět buďto jako validační pravidla, nebo jako omezení typu CHECK jazyka SQL. Omezení entitové integrity, podle něhož musíme být schopni jednoznačně identifikovat každou jednotlivou instanci dané entity, implementujeme pomocí primárního klíče definovaného pro každou tabulkou.

Ať už budeme databázi implementovat v SQL Serveru nebo nad databázovým strojem Microsoft Jet, někdy přijde na to, že některá z omezení, definovaných v myšlenkovém datovém modelu, se nedají implementovat jako součásti definice tabulky. V SQL Serveru můžeme taková omezení zajistit pomocí spouští (triggers). Databázový stroj Jet však spouště nepodporuje, takže v něm musíme tato omezení implementovat jako součást aplikace.

## Vztahy

O způsobech modelování vztahů mezi entitami v relační databázi jsme hovořili v kapitole 3 a poté ještě jednou v kapitole 9. Prvním krokem je vždy zahrnout do cizí (nevlastní) relace nějaký jednoznačný identifikátor z primární relace. Na úrovni databázového schématu to znamená, že do cizí tabulky přeneseme z primární tabulky všechna pole jejího primárního klíče.

Někteří vývojáři se na tomto místě zastaví a referenční integritu implementují raději v samotné aplikaci, než by tyto činnosti ponechali na databázovém stroji. Já při své práci – podobně jako u všech ostatních mechanismů validace v databázích – dělám obojí: referenční integritu validuji jednak v aplikaci s ohledem na použitelnost v systému, jednak i v databázovém stroji kvůli bezpečnosti. Myslím si, že kdybych byla mužem, asi bych nosila opasek i šle zároveň.

## Indexy

Význam indexů pro celkový výkon systému jsme již rozebírali před chvílí. Alespoň jeden index by měla mít každá tabulka, přínejmenším index, který vytvoří databázový stroj automaticky při deklaraci primárního klíče. Kromě toho je vhodné vytvořit index na každém poli nebo kombinaci polí, na základě nichž budeme provádět operace spojení tabulek. U tabulky, která ve vztahu reprezentuje primární relaci, to obvykle není žádný problém, protože pro spojení se používají pole, jež tvoří přímo primární klíč. Jestliže ale v tabulce, která představuje cizí relaci, netvoří pole použité (použitá) pro spojení celý primární klíč, musíme do ní zpravidla doplnit další indexy.

Pokud je pole cizího klíče součástí primárního klíče, netvoří ale *celý* klíč, definují samostatný index na cizím klíči. Tabulka RozpisObjednávek bude mít například nejspíše primární klíč (ČísloObjednávky, ČísloVýrobku). I když pro spojení této tabulky s hlavní tabulkou Objednávky mů-

žeme ve většině případů (tedy alespoň ve většině případů, které si já sama dokáži představit) použít přímo index primárního klíče, je přece jen vhodnější vytvořit pro jistotu ještě samostatný index jen na poli ČísloObjednávky.

Indexovat je také třeba veškerá pole, na základě kterých se bude provádět třídění (řazení) dat. Seznam zákazníků se tak například zpravidla řadí podle jmen zákazníků a objednávky se třídí podle datumu, i když v žádném z těchto dvou případů se tříděné pole obvykle neúčastní primárního klíče, ani nevytváří operaci spojení tabulek. Vytvoření indexu na těchto polích vedle k usnadnění a zefektivnění procesu třídění.

Vytváření indexů se dá ale také docela dobře přestřelit, takže budte opravdu opatrní. Nezapomeňte, že s údržbou každého jednotlivého indexu je spojena jistá režie – je poměrně malá, při větším počtu indexů se ale samozřejmě sčítá. Každé pole, podle kterého budeme tabulku často třídit, je třeba indexovat; pomocí klausule ORDER BY jazyka SQL se ale dají záznamy setřídit i bez indexu.

Praktický maximální počet indexů v dané tabulce závisí ve skutečnosti na tom, jak často se tabulka bude aktualizovat. (Index znamená režii pouze při přidávání nového záznamu nebo při takové aktualizaci, kdy se změní některé z polí indexu.) Takovou tabulku, jako jsou například Objednávky, bude systém aktualizovat víceméně neustále, takže bych nedoporučovala udržovat více než nějakých 10 až 15 indexů, a to včetně primárního klíče a včetně indexů, které slouží pro podporu spojení tabulek. Více indexů by se na druhé straně dalo ospravedlnit třeba v tabulce Výrobky, která se nejspíše bude aktualizovat méně často, ale jejíž údaje bude systému využívat mnoha různými způsoby. Jako vždy se tedy musíte rozhodovat podle toho, jak se s daty bude v systému pracovat.

## Pohledy a dotazy

Jak Microsoft Access, tak i SQL Server nabízí určitý mechanismus pro uložení příkazů SELECT jazyka SQL. Tento uloženým příkazům se v SQL Serveru říká *pohledy*, zatímco v Microsoft Accessu se nazývají *dotazy*. (Já jim zde budu říkat dotazy, protože je to obvyklejší pojem.) Uložený dotaz běží ve většině případů rychleji než samostatně vytvořený příkaz SELECT; úplně vždycky to sice neplatí, onou příslušenou výjimku zde ale tvoří natolik výjimečné situace, že je můžeme v podstatě pominout a uvedené tvrzení můžeme považovat za obecně platné pravidlo.

Při rozhodování o dotazech, které zahrneme do databázového schématu, můžeme začít tím, že se v myšlenkovém datovém modelu pokusíme vyhledat složené entity. Připomínám, že složená entita je jedna logická entita, kterou z důvodu efektivity implementujeme pomocí dvou nebo více tabulek. Do modelu je tak třeba zahrnout zejména dotazy, které denormalizují veškeré složené entity. Většinu těchto entit budou tvořit tabulky ve vztažích typu jedna k více, jako jsou například tabulky Objednávky a RozpisObjednávek; z některých složených entit se ale také vytvářejí podtřídy pomocí vztahů typu jedna k jedné – i pro ně bychom měli vytvořit příslušné dotazy.

Uživatelé systému budou velice často potřebovat vyhledávat určité záznamy v primárních entitách – budou například hledat určitého zákazníka nebo určitou objednávku; toto hledání představuje druhý okruh dotazů, které je vhodné začlenit do databázového schématu. Všechny tyto obvyklé případy vyhledávání ošetříme nejlépe pomocí parametrického dotazu, do něhož může uživatel za běhu aplikace zadat charakteristiku konkrétního hledaného záznamu.

Někdy budeme potřebovat pro danou entitu nadefinovat více než jeden vyhledávací dotaz. Uživatel bude tak například hledat Objednávky podle známého údaje DatumObjednávky, KódZákazníka, nebo ČísloObjednávky. Pro každý z těchto typů hledání musíme opět vytvořit samostatný parametrický dotaz.

Na druhé straně ale uživatelé určitě nebudou prohledávat úplně každou tabulkou. Databázové schéma bude například obsahovat tabulkou se seznamem okresů České republiky. Tyto vyhledávací tabulky jsou rozhodně velice užitečné, je ale velice nepravděpodobné, že by některý uživatel potřeboval někdy vyhledat záznam určitého okresu.

Po potřebných dotazech je také dobré se poohlédnout mezi formuláři a sestavami, které budeme v aplikaci implementovat. Zde budeme potřebovat dotazy pro propojení polí a také pro podporu různých vyhledávání, jako je například vyhledávání v kombinovaném poli. Jestliže bude systém obsahovat vzájemně závislé formuláře, budeme muset i pro ně vytvořit parametrické dotazy. Příkladem může být dialog, který zobrazuje podrobné informace o Zákazníkovi a který se vyvolá z formuláře pro pořizování objednávek.

V závislosti na pracovních procesech daného systému je třeba do databázového schématu zahrnout také dotazy (respektive v SQL Serveru uložené procedury), které provádí různé operace. Jestliže například víme, že se v systému budou pravidelně archivovat objednávky nebo hromadně aktualizovat ceny výrobků, bude celý proces mnohem efektivnější, pokud jej bude podporovat odpovídající dotaz nebo uložená procedura, než kdybychom v něm museli vytvářet příkazy za běhu.

Další akční dotazy doplníme nejspíše do databázového schématu během implementace. Na rozdíl od indexů neznamená implementovaný dotaz či uložená procedura v podstatě žádnou režii, takže rozhodně neváhejte a doplňte do databázového schématu všechno, co je potřeba.

Nezapomeňte také, že vývoj systémů nemusí být v žádném případě přesně lineární proces. Zatímco změny v samotných tabulkách během implementace mohou způsobit značné problémy (a čím dál se v procesu vývoje nacházíme, tím jsou tyto problémy horší a horší), přidávání dalších dotazů do schématu je velice jednoduché a také velice žádoucí.

## Bezpečnost

Jakmile pochopíme pracovní procesy v systému a sestavíme myšlenkový datový model, musíme dále zjistit administrativní požadavky na systém. Tyto administrativní požadavky nemusí mít nutně přímý dopad do databázového schématu, jsou to ale nieméně aplikační pravidla, kterým se ostrý systém musí přizpůsobit.

Administrativní požadavky jsou v jistém slova smyslu jakési „meta-požadavky“, protože se týkají *samotného* systému, nikoli prostoru problému, který tento systém modeluje. Spadají do dvou různých kategorií: první z nich tvoří bezpečnostní požadavky, které určují, kdo smí k systému přistupovat, zatímco do druhé řadíme požadavky dostupnosti, z nichž se odvozují takové věci, jako například po jakou dobu musí systém běžet (například 24 hodin denně po sedm dní v týdnu, nebo jen v běžné pracovní době) a jak budou uživatelé zálohovat data. Dostupnost systému je téměř výhradně otázkou implementace, takže se zde budeme zabývat pouze bezpečností.

Implementace vhodného bezpečnostního schématu může být dosti složitá věc. Jak pro Microsoft Access a databázový stroj Jet, tak i pro SQL Server jsou však naštěstí tyto procesy velice dobře do-

kumentovány. Ještě příjemnější je, že návrh databáze je něco jiného než vlastní implementace, takže v této fázi stačí zvážit jen základní logické bezpečnostní vazby – principy bezpečnosti na logické úrovni jsou docela jednoduché.

## Úrovně bezpečnosti

Nejprve musíte zjistit, jakou úroveň bezpečnosti bude systém vyžadovat. Všimněte si, že zde hovoříme o zabezpečení dat, nikoli systémového kódu; to je již otázkou implementace. Určité mechanismy pro ochranu programového kódu před nedbalým nebo úmyslným poškozením nabízí jak Microsoft Access, tak i jazyk Visual Basic. Na nejnižší úrovni bezpečnosti se nachází zcela nezabezpečený systém, v němž smí k databázi přistupovat kdokoli a kdykoli. Taková „ochrana“ je zcela zřejmě nejsnazší na implementaci i na administraci, protože v podstatě znamená, že nemusíme nic zvláštního dělat.

Pokud ale mají data pro danou organizaci určitou hodnotu (a tu většinou mají), je implementace zcela nezabezpečeného systému příliš lehkomyслná. Přesto může mít svůj smysl – jestliže má například nás klient implementované vhodné schéma sítové bezpečnosti, které dokáže přístup k datům již samo omezit, nemusíme bezpečnostní opatření v databázi duplikovat.

Další úrovni je *bezpečnost na sdílené úrovni*. Na této úrovni případně celé databázi jedno heslo; každý, kdo toto heslo zná, má k systému plný přístup. I tento model se docela snadno implementuje a administruje; jedinou podmírkou je pravidelná změna hesla. Bezpečnost na sdílené úrovni v řadě situacích skutečně postačuje.

*Bezpečnost na úrovni uživatelů* je sice pracnější jak co se týče implementace, tak i administrace, poskytuje však nad databází nejpřesnější kontrolu. V bezpečnosti na úrovni uživatelů může administrátor přiřadit každému jednotlivému uživateli přesná oprávnění ke každému objektu: „Pepa může přidávat a editovat informace do entity Zákazníci, ale entitu Objednávky si smí pouze prohlížet. Máňa smí přidávat a editovat informace do entity Zákazníci i do entity Objednávky. Ani Pepa, ani Máňa však nesmí záznamy žádného typu odstraňovat.“

Ríkat tomuto typu bezpečnosti „bezpečnost na úrovni uživatelů“ je ale ve skutečnosti poněkud zavádějící. Bezpečnostní oprávnění je sice možné přidělit jednotlivým uživatelům, dají se ale také přidělit obecným rolím, které posléze „obsadíme“ uživateli jako jednotlivci. To je již mnohem efektivnější mechanismus implementace bezpečnosti, protože znamená mnohem menší nároky na administraci.

V tomto modelu musíme nejprve pojmenovat typy uživatelů systému – identifikujeme tak systémové administrátory, pracovníky pro pořizování objednávek, obchodní zástupce a podobně – a poté zjistíme, jaká bezpečnostní oprávnění má mít každá z rolí pro každý jednotlivý objekt v systému. To ale neznamená, že bychom museli nutně přiřazovat bezpečnostní oprávnění přímo pro datové objekty; ve skutečnosti to dokonce není ani vhodné. Rozhodneme se například, že obchodní zástupci mají mít možnost přidávat, editovat i odstraňovat záznamy v tabulce Zákazníci, zároveň ale nechceme, aby mohli zasahovat do tabulky přímo. Přidělíme jim tedy oprávnění pro práci s formulářem Údržba zákazníků, nikoli však pro samotnou tabulku. Takto budeme mít jistotu, že uživatel nemůže třeba neúmyslně obejít speciální zpracování definované ve formuláři pro údržbu.

Často potřebujeme dát lidem do rukou oprávnění k prohlížení jen určité části dat. V tabulce Zaměstnanci povolíme například prohlížení polí Jméno a Linka každému, zatímco prohlížení pole Mzda umožníme jen vedoucím pracovníkům. Podobně můžeme obchodním zástupcům dovolit prohlížet si Objednávky podané jejich vlastními klienty, nikoli však klienty jiného obchodníka. Obě tyto situace vyřešíme pomocí dotazů; příslušným uživatelům pak přidělíme oprávnění k těmto dotazům a zamítneme jim přístup k podkladovým tabulkám.

## Audit

Někdy musíme nejen kontrolovat přístup uživatelů k datům, ale potřebujeme také vědět, co uživatelé v systému skutečně provedli. Tyto požadavky mohou být velice různé. V některých organizacích stačí nadřízeným sledovat, kdo se přihlásil k systému a kdy. Jiní vyžadují podrobné záznamy o tom, kdo kdy provedl které změny. A ostatní chtějí něco mezi tím.

Jak budeme požadavky na audit neboli sledování v systému modelovat, to záleží na jejich přesné podobě. Pokud nám stačí sledovat, kdo všechno se systémem pracoval, nadefinujeme třeba jen jednoduchou entitu s atributy UživatelskéJméno, ČasPřihlášení a ČasOdhlášení. Při přihlášení uživatele tak vytvoříme záznam, který posléze s jeho odhlášením aktualizujeme.

Někdy potřebujeme také vědět, kdo daný záznam pořídil (přidal do tabulky). Toto sledování můžeme zajistit přímo v primární entitě, a to pomocí jednoho či dvou doplňujících atributů: Vytvořil, případně ČasVytvoření.

Sledovat odstranění záznamů bývá složitější. Obecně zde máme několik možností. Můžeme tak uživatelům zcela zabránit ve skutečném odstraňování záznamů a do záznamu přidat namísto toho příznak Odstraněn, který případně doplníme o atributy Odstranil a ČasOdstranění. Při tomto postupu můžeme záznamy před jejich fyzickým odebráním z databáze ještě například zkopirovat do archivního souboru.

Při druhé variantě sice operace odstranění povolíme, veškeré potřebné informace zapíšeme ale současně do souboru s protokolem, podobně jako jsme si zaznamenávali přihlášené uživatele. Samozřejmě i zde musíme vytvořit vhodné doplňující atributy. Konec konců asi nemá příliš velký smysl vědět, že někdo určitý záznam odstranil, aniž bychom zároveň nějakým způsobem dokázali rekonstruovat jeho *původní* podobu.

Jestliže potřebujeme vědět dopodrobna, kde a jaké změny byly provedeny, musíme do modelu přidat ke každé tabulce ještě jednu speciální tabulkou pro audit. V této tabulce můžeme sledovat jak uživatele, který změnu provedl, tak i datum jejího provedení a změněný záznam s původními a novými hodnotami.

Chcete-li některé z těchto funkcí pro sledování neboli audit implementovat v databázi Microsoft Jet, musíte zároveň uživatelům zakázat jakýkoli přímý přístup k tabulkám, protože tak by mohli velice snadno obejít veškerá zavedená bezpečnostní opatření. V SQL Serveru nic takového dělat nemusíme; tento stroj podporuje totiž databázové spouště, které se obejít ani potlačit nedají.

Ať už ale tyto požadavky na sledování a audit systému implementujeme jakkoli, musíme také zvážit, jakým způsobem se informace při něm získané budou využívat, kdo je bude využívat a za jakých okolností. Přístup k tabulkám s údaji o auditu je tak zcela zřejmě nutné omezit.

Zároveň budeme možná muset do systému doplnit pracovní procesy auditu. Mají mít systémoví administrátoři možnost vrátit provedené změny zpět? Je třeba generovat sestavy o využití systému?

Podle mých zkušeností tvoří většinu požadavků na audit jakési „pojistky“, přičemž informace vzešlé z tohoto sledování se používají jen za výjimečných okolností. V takovém případě nemusíme třeba pracovní procesy v systému vůbec rozšiřovat. Systémový administrátor může zkoumat data manuálně, s pomocí interaktivních funkcí Microsoft Accessu nebo SQL Server Enterprise Manažeru; ve stejném nástroji vykoná i potřebné operace.

## Stručné shrnutí

Tato kapitola se zabývala převodem myšlenkového (konceptuálního) datového modelu do podoby fyzického databázového schématu. Začali jsme výkladem dvou architektur pro definování struktury programového kódu v systému – třívrstvého modelu a čtyřvrstvého modelu – a to zejména z pohledu možných dopadů zvolené architektury kódu na databázové schéma.

Dále jsme rozebrali několik možných architektur dat. V jednovrstvém systému se jak aplikace, tak i data nacházejí na stejném logickém počítači. Jednovrstvá aplikace může být buďto jako samostatná, izolovaná aplikace, nebo mohou být její data uložena na síti, jsou ale přístupná jen jednomu uživateli. Síťové aplikace, implementované v databázovém stroji Jet jsou z logického pohledu také jednovrstvé; k datům však v nich může přistupovat současně i větší počet uživatelů.

Dvouvrstvé aplikace neboli aplikace architektury klient/server můžeme implementovat pomocí SQL Serveru. V této logické architektuře provádí veškeré manipulace s daty serverový počítač, zatímco klient je odpovědný za komunikaci (interakci) s uživatelem. Základní principy dvouvrstvých aplikací můžeme dále rozšířit na tři nebo více počítačů; v takovém případě hovoříme o n-vrstvé aplikaci.

Nakonec jsme se věnovali převodu myšlenkového datového modelu na databázové schéma. Tento proces je většinou poměrně přímočarý, protože jediné nové informace, které do něj vstupují, tvoří definice indexů a dotazů, jež bude navrhovaná databáze implementovat. Posledním tématem, na které jsme se v této kapitole podívali, byly důsledky bezpečnostních požadavků pro databázové schéma; zároveň jsme si řekli něco málo o návrhu bezpečnostního schématu na logické úrovni.

V následující kapitole se stručně podíváme na některé otázky související s přednesením čili sdělením našeho návrhu nového systému klientům a ostatním členům vývojového týmu.



Snad jen s výjimkou velice jednoduchých databázových systémů, které píšete sami pro sebe, můžete výsledky své práce umět také přednést neboli sdělit jiným lidem. Všimněte si prosím, že zde říkám „sdělit“, nikoli „zdokumentovat“. Všechna dokumentace světa je úplně k ničemu, pokud je komplikovaná a nečitelná. Stejně jako se cizímu jazyku nenaučíte tak, že si přečtete slovník, nemůžete ani projekt pochopit pouhým prohlédnutím nějakých tabulek dat.

Pro tyto úkoly jsou velice důležité základní dovednosti psaného projevu; jestliže tedy o těchto svých schopnostech byť jen trochu pochybujete, kupte si nějakou dobrou učebnici nebo se přihlaste do vhodného kursu. *Ujišťuji* vás, že to nebude promarněný čas. Pokud si totiž nejste jisti v kramflecích se spisovnou gramatikou, budou se vaši klienti právem ptát, čemu jinému ještě nerozumíte tak dobře, jak byste měli. (A to je konec kázání.)

## Komu je sdělení určeno a jeho význam

Porozumět „obecenstvu“, kterému je naše sdělení určeno, je důležité v každém písemném projevu, zvláště však při přednesení návrhu systému; čtenáři tohoto našeho díla bude totiž nejspíše hned několik různých typů čtenářů se zcela různými požadavky.

Abyste správně pochopili, kdo všechno bude výsledný dokument číst, a tím pádem i co všechno do dokumentu patří, si musíte nejprve položit otázku, čeho se vy sami a vaši čtenáři pokoušíte dosáhnout. Klient potřebuje získat jistotu, že jste jeho požadavkům porozuměli, a získat určité záruky, že mu výsledný systém pomůže dosáhnout jeho cílů. Nepotřebuje (nebo alespoň nechce) pochopit veškeré podrobnosti týkající se implementace vlastního systému. Má-li být ale dokument zároveň základem dalšího vývoje, potřebuje v něm vývojový tým vidět přesně ty podrobnosti, které klient očima jen zběžně přelétne.

Někdy je nevhodnějším řešením připravit několik různých dokumentů: jeden pro klienta a druhý pro vývojový tým. Tento postup je velice dobrý zejména v iterativním vývojovém modelu, protože mu velice blízce odpovídá.

Já v těchto situacích napíši obvykle několik dokumentů:

- Specifikaci požadavků, která je určena primárně pro klienta a ve které pomocí (relativně) ne-technických pojmu vyjadřuji, jakým způsobem chápu analyzovaný systém.
- Specifikaci architektury, kterou si bude číst jak klient, tak i vývojový tým, ale která je určena zejména pro vývojáře; rozebírá podrobně vzájemné závislosti a komunikace mezi komponentami.
- Ke každé komponentě samostatnou Technickou specifikaci určenou pro potřeby vývojového týmu.

U jednodušších systémů zpravidla plně dostačuje jediný dokument. Musíte do něj pouze zpracovat potřeby každé skupiny čtenářů a příslušné informace podat v takové formě, která jím bude snadno srozumitelná.

## **Struktura dokumentu**

Konečná struktura dokumentu závisí především na jeho záběru a určení a také na vašem vlastním vkusu – pochopitelně pokud nepracujete ve firmě, která vyžaduje dodržení přesných standardů pro dokumentaci. Dokument může být tak jednoduchý nebo naopak tak složitý, jak jen budete chtít; neexistuje pro něj žádný standardní formát. V této kapitole vám dám jisté obecné zásady, zároveň ale předpokládám, že si je upravíte podle konkrétní potřeby.

Jestliže se držíte mých doporučení ohledně analýzy systémů, pak zjistíte, že se požadovaný dokument docela dobře rozpadá na několik dobře definovaných částí, které se (nikoli náhodou) poměrně přesně kryjí s předchozími kapitolami části 2 této knihy. Do dokumentu je dále vhodné začlenit Úvod či Stručné shrnutí určené pro vedoucí pracovníky.

## **Stručné shrnutí pro vedoucí pracovníky**

V rozsáhlejších organizacích dohlíží často na vývoj projektů nějaký řídící výbor. Potřebnou odpovědnost za celkový dozor nad projektem nebo nad jeho rozpočtem má obvykle určitý pracovník ve vedoucí pozici, a to i u malých projektů. Jestliže má nad vaším projektem někdo konkrétní takový dozor, je vhodné zahrnout do dokumentu nějaké Shrnutí pro vedení, určené právě jemu.

Tito lidé se obvykle bůhvíjak náruživě nezajímají o podrobnosti celého systému. Mají jen několik konkrétních otázek, na které chtějí slyšet odpověď; čím efektivněji je odpovíme, tím lépe. Lidé ve vedoucích pozicích se zpravidla ptají na takovéto věci:

- Jakými problémy se navrhovaný systém zabývá?
- Je to nejlepší a nákladově nejfektivnější řešení?
- Jaká jiná řešení byla zvážena?
- Jak dlouho potrvá implementace celého systému?
- Kolik to bude stát?
- Jaká jsou se systémem spojena rizika?

Odpověď na první otázku je velice jednoduchá, pokud se nám podařilo dobře nadefinovat cíle a záběr systému. Stačí je zde pouze uvést znovu. Obvykle se snažím být co nejpreciznější a uvádět zde explicitně také všechny věci, které se mohly do systému začlenit, ale byly zamítнутý. Když nic jiného, je to dobrá pojistka proti pozdějším námitkám.

Jestliže jste v pozici externího konzultanta, může druhá a třetí otázka, které se týkají jiných řešení, zůstat bez odpovědi (prostě proto, že to nemůžete vědět). Máte-li však k témtu informacím přístup, je vhodné zde alespoň stručně nastínit základní principy ostatních řešení a uvést důvody jejich zamítnutí. Šéfové se rádi ujišťují, že byly opravdu důkladně zváženy veškeré další alternativy.

Odpovědi na druhou a třetí otázku musí být ale každopádně stručné. Pokud jste například velice důkladně prověřili dostupné aplikace typu krabicového softwaru a zamítli je ve prospěch vlastního řešení, máte zřejmě k dispozici velice podrobné informace o nákladech, funkcích, další podpoře atd. Sem ale tyto informace nepatří; napište je do přílohy. Celé shrnutí pro vedoucí pracovníky by prakticky vzato nemělo přesáhnout jen několik málo stránek.

Podat rozumnou odpověď na otázky ohledně časových a finančních nákladů může být zálužné a téma vždy je to nepříjemné. Jestliže ale uvážíte, co se snažíte u šéfů prosadit a co mají zakoupit, doberete se odpovědi mnohem snáze. Při vývoji malého systému máte nejspíše o odpověďích velice dobrou představu; vycházejí z definovaného záběru systému. U rozsáhlých složitých systémů v této fázi ještě nevíte, kolik času a peněz si vyžádá; to je ale naprostě v pořádku. Podstatné je, že víte, jaký je další krok; nic jiného již na tomto místě nepotřebujete.

Jestliže dokážete realisticky říci něco jako: „Zatím není možné odhadnout skutečnou dobu trvání či náklady na úplnou implementaci, očekáváme ale, že se vejde někde do intervalu x až y. K dokončení následující fáze projektu bude ale stačit pouze z a její výsledek bude...“, pak jste rozhodně na dobré cestě. Jenom se musíte ujistit, že onen „výsledek, který bude...“ je hmatatelný a že má pro organizaci sám o sobě určitou hodnotu. Lidé podle mých vlastních zkušeností velice neradi vydávají peníze na nějaký blíže nedefinovaný „další výzkum“.

Přišla jsem také na to, že jedním z nejefektivnějších způsobů, jak si získat důvěru klienta, je pokusit se vhodným způsobem odpovědět na poslední otázku ohledně případných rizik. Promyslete si, co by se mohlo špatného stát a venujte tomu určitou práci. Našli jste určité dosud, nevyřešené technické otázky? Budou věci trvat déle, než se očekává? Kde? Kdy? Proč? Budete klidně trochu paranoidní.

Nyní se ale vrátíme zpátky na zem, do reality. Jak pravděpodobně jsou tyto situace? Ano, je možné, že celý náš vývojový tým (respektive jeho podstatnou část) sklálí chřípková epidemie, nebo že se naše kancelář zřítí pod padajícím letadlem. To ale není příliš pravděpodobné. Docela pravděpodobně naopak je to, že se během řešení projektu stane něco nepříznivého; vedoucí pracovníci chtějí mít jistotu, že jsme zvážili všechny možné a pravděpodobné problémy a že jsme si připravili určité havarijní plány. Nemusíte zde uvádět úplně všechno, stačí třeba jen v několika málo odstavcích rozvést dva nebo tři možné problémy, je ale důležité, abyste zvážili i otázky rizika.

Externím konzultantům pak nedoporučuji vyhýbat se zde možným problémům, díky kterým byste nemuseli úkol dokončit. První otázka, kterou si zcela zřejmě položí každý, kdo si vás má najmout a zaplatit, je, jestli jste schopni předloženou práci vůbec udělat. Ve shrnutí pro vedoucí pracovníky zřejmě není právě vhodné se chlubit svými referencemi, je však třeba naopak se dotknout ri-

zika nesplnění úkolu. Podrobné smluvní ošetření rizik je čistě na vás; v tomto ohledu vám nebudu doporučovat nic zvláštního. Pouze vám mohu sdělit vlastní zkušenosť, podle níž jasnou zmínkou o těchto problémech zvýšíte svoji podnikatelskou důvěryhodnost.

## Přehled systému

Ať už jste do dokumentu začlenili formální Shrnutí pro vedoucí pracovníky, nebo méně strukturovaný Úvod, první částí správně napsaného dokumentu musí být vždy Přehled systému. To je jediná část dokumentu, která je stejná pro všechny jeho čtenáře. Celkový záběr projektu musí pochopit jak vývojový tým, tak i cílový klient. Rozdíl mezi oběma typy čtenářů je opravdu minimální.

Jestliže jste se rozhodli nevytvořit žádné formální Shrnutí pro vedoucí pracovníky, pak musejete některé z informací, které by se v této části jinak objevily, zapsat právě do Přehledu systému. A i když Shrnutí v dokumentu připravíte, můžete v Přehledu systému některé otázky rozebrat podrobněji. Některé z rozebíraných otázek – jako je například porovnání alternativních řešení nebo řízení rizika – mohou být přitom natolik rozsáhlé, že si zaslouží samostatnou část dokumentu.

Hlavním cílem této části dokumentu je ale podat jakýsi „celkový obrázek“ podoby systému. Zde by tedy mělo stačit vysvětlit základní parametry systému – cíle a záběr systému a dále kritéria návrhu.

O přednesení cílů a záběru systému v podstatě téměř není co si říci. Pokud jste cílům a záběru systému správně porozuměli, stačí je napsat do dokumentu, a to takovými slovy, aby jim rozuměli předpokládaní čtenáři. Pokud jim nerozumíte, vratíte se zpět a pokuste se je pochopit.

Opět vám doporučuji být co nejpřesnější a nejkonkrétnější a uvést zde zejména ty oblasti, které se již nacházejí mimo rámec systému, nebo které je možné implementovat později. Napište sem všechno, co by bylo myslitelné do systému zavést, přestože jste se třeba rozhodli danou oblast ze záběru systému vypustit. Věnujte určitý čas popisu oblastí, které by se podle vašeho názoru *mohly* do systému zahrnout, ale o kterých se nikdy explicitně nehovořilo. Přesně v tomto okamžiku se totiž musíte ujistit, že se vy sami a váš klient pohybujete na jedné lodi a že pracujete se stejnými výchozími předpoklady.

Jestliže jste navíc vytvořili k systému analýzu nákladů a přínosů, je třeba ji také do dokumentu zahrnout, i když ne nutně právě do této části. Jistým způsobem je to otázka stylu a vкусu, ale já osobně nerada vkládám do hlavního dokumentu stránky a stránky plné tabulek. Dokument bude mnohem čitelnější, pokud do jeho hlavní části zapíšeme jen souhrnné informace – nejvýše jednu nebo dvě tabulky – a zbytek přesuneme do příloh.

Totéž platí pro dokumentaci cílů a záběru systému. V této fázi projektu můžete mít připravenou podrobnou analýzu funkcí, důkladně provázanou s podporovanými cíli a upravenou podle jejich významu. To je velice dobrá věc, kterou v celém projektu rozhodně využijeme. Pokud je ale tabulka delší než zhruba na jednu stránku, nepatří do hlavního textu dokumentu, ale opět do přílohy. Do těla dokumentu vložte pouze souhrnnou tabulku s textovým popisem a na podrobnější informace odkažte čtenáře do odpovídající přílohy.

## Pracovní procesy

Nejlepší způsob, jakým můžeme přednест čtenářům pracovní procesy systému, závisí na způsobu jejich zachycení. Jestliže jste z nich sestavili jakousi osnovu, můžete je do textu dokumentu přímo zahrnout. Pokud jste si v rámci analýzy připravili diagramy pracovních procesů, vložíte je do dokumentu také. Nezapomeňte k nim ale doplnit vysvětlení použitých symbolů. A každopádně také vysvětlete, co myslíte pod pojmy „proces“, „úkon“, „aktivita“, „operace“ a podobně, které v textu a v diagramu používáte.

Bez ohledu na konkrétní podobu této dokumentace pracovních procesů napište do výsledného dokumentu jejich textový, slovní popis. Příprava takového „slobového cvičení“ je především dobrou kontrolou formálního popisu. Budete překvapeni, jak často najdete tímto způsobem různé chyby.

Za druhé je velice důležité, aby čtenář tyto pracovní procesy skutečně posoudili. Různé osnovy či body a diagramy lidé zpravidla jen zběžně přelétou a dostatečně je nepochopí. Jestliže do dokumentu zahrnete jak grafickou, tak i slovní verzi popisu, přinutíte čtenáře si všechno přečíst a opravdu materiálu porozumět; dvě podoby informací pomohou vysvětlit jedna druhou.

Jestliže navrhujete určité změny pracovních procesů, nezapomeňte do skriptu zahrnout jak současný stav, tak i novou verzi; ve slovním popisu pak veškeré změny zvlášť zvýrazněte. Samozřejmě musíte také vysvětlit, proč navrhované změny zlepší pracovní toky nebo proč a jak vyřeší určité problémy. Pokud takto explicitně přednesete v dokumentu všechny navrhované změny, přijde v tomto slovním popisu často i sám klient na něco, co jste v prvotní analýze přehlédli.

Dokumentace pracovních procesů je jednou z oblastí, ve kterých mohou potřeby různých typů čtenářů našeho dokumentu kolidovat. Vývojový tým zde bude hledat výklad podaný v technických pojmech: transakce se potvrzuji a datové položky se aktualizují. Klienti zase na druhé straně očekávají, že pracovní procesy popíšete těmi slovy, jaké jsou oni sami zvyklí používat. To mohou být shodou okolností stejně pojmy jako náš počítačový slovník, obvykle tomu tak ale není.

Budete-li při psaní dokumentu někde na pochybách, dovolte si malou nepřesnost raději na straně klienta. Prakticky vzato vývojový tým rozhodně mnohem snáze pochopí text napsaný pomocí výrazových prostředků klienta, než naopak. Pokud se někde technickým pojmem nevyhnete, nezapomeňte je v dokumentu odpovídajícím způsobem nadefinovat.

Tohle všechno se ovšem naneštěstí snadno říká, ale mnohem hůře udělá. Jestliže se ve své práci plně věnujete počítačům, velice snadno zapomenete, že některé z těchto pojmu opravdu nemusí být běžnému člověku srozumitelné.

Já si při psaní dokumentů udržuji seznam takových pojmu, jako je například „transakce“, a dokonce i „soubor“ a dokument před konečným odevzdáním dokumentu klientovi ještě jednou zreviduji. Není těžké jej sestavit například s pomocí funkce Hledat textového procesoru. Bezradný klient je jen zřídka spokojeným klientem.

## Myšlenkový datový model

Výsledkem dokončené prvotní analýzy systému je nejspíše množina diagramů entit a vztahů (E/R diagramů), seznam domén neboli oborů hodnot použitých v systému a určité poznámky ohledně omezení (omezujících podmínek) v datech. Tyto údaje se dají poměrně snadno uspořádat do formátu, v němž je budeme prezentovat. Zcela vhodnou a postačující dokumentací modelu je analýza entit, u složitějších entit případně ilustrovaná příslušným E/R diagramem. Analýzu domén neboli oborů hodnot pak ve svých dokumentech mám ve zvyku pojmit v samostatné části textu jako jakýsi „slovníček pojmu“ a poté se na něj v modelu vhodně odvolávám.

Toto je jedno z míst dokumentace, kde se několika stránkám tabulek nejspíše nevyhneme. Stejně tak zde nic nenaděláme s tím, že se jedná v podstatě o technický výklad; klient je však nemůže pominout. Vaším úkolem je snažit se o to, aby byl celý proces pokud možno bezbolestný.

Za prvé, používejte co nejméně technických pojmu a nepište jich do dokumentu více, než skutečně potřebujete. Slovům jako „tabulka“, „pole“ a „záznam“ se zde zřejmě nevyhnete; na slova „entita“, „relace“ a „atribut“ ale raději zapomeňte. Vím, že ony „méně technické“ pojmy jsou také méně přesné. I tak se ale svému přesnému výkladu poměrně dobře blíží. Dále nezapomeňte nadefinovat veškeré pojmy, které v dokumentu používáte (ne abyste místo slovníčku pojmu dali uživatelům krátkou odbornou lekcí z návrhu databází).

Všechno to vypadá možná na první pohled složitě, v praxi to však velký problém není. Jakmile v dokumentu vysvětlíte, že každá tabulka reprezentuje v podstatě nějakou „věc“ a její pole představují „údaje a informace“ o této věci, je již klient obvykle připraven na zbytek textu. Někdy se vás klient zeptá na nějaký detail, například proč se s poštovními směrovacími čísly pracuje jako se znakovými poli, já se ale takovými věcmi zabývám raději neformalně až v momentě, kdy vzniknou.

Jestliže má dokument sloužit zároveň jako technická specifikace, musíte do něj zahrnout také veškeré technické údaje a podrobnosti, které bude potřebovat vývojový tým. Já se ve svých dokumentech snažím tyto podrobné informace vést odděleně od hlavních tabulek; obvykle je zapisuju do podnadpisu nižší úrovně ke každé entitě. Klientům zpravidla netrvá dlouho si uvědomit, že „těmhle věcem“ prostě rozumět nemusí. V takových situacích klienta obvykle požádám o kontrolu úplnosti seznamu atributů a o kontrolu přesnosti analýzy domén, přičemž zbytek může zcela ignorovat.

Teoreticky by měl klient zkontolovat i vztahy mezi entitami, v praxi jsem se však opravdu jen málokdy setkala s tím, že by klient zrovna v této oblasti dokázal „vychytat“ nějaké chyby či místa, která jsem nesprávně pochopila; navíc se vždy jednalo výhradně o situaci, kdy jsem s ním model rozebírala osobně. Pro klienta jako laika je tato problematika opravdu příliš vzdálená.

Klienta je ale dobré požádat o kontrolu velikosti polí a jejich datových typů; to přináší rozhodně lepší výsledky. Jestliže máte ovšem nějaké vážnější nejasnosti, je dobré si i zde naplánovat s odpovědnými pracovníky raději osobní setkání. Někdy je to únavné, to uznávám, tyto věci musí být ale naprostě správné, a proto se něčemu takovému často nevyhneme.

Druhá možnost je zvýraznit problémové okruhy. Člověk, kterému dáte k posouzení 50 stránek různých tabulek, již zpravidla stěží dokáže udržet pozornost, takže pokud v dokumentu zdůrazníte, co od něj potřebujete zkontolovat a potvrdit, dostanete určitě lepší výsledky.

## Databázové schéma

Samotné databázové schéma vzniká z velké části jen přímým převodem myšlenkového (konceptuálního) datového modelu, a proto i dokumentace databázového schématu vypadá obvykle do značné míry podobně jako dokumentace datového modelu.

Informace obsažené v databázovém schématu jsou velice důležité pro členy vývojového týmu; protože však obsahují jen málo nových informací, pro klienta jsou prakticky nepodstatné. Proto jestliže nevytváříte pro klienta a pro vývojáře samostatné dokumenty, přesuňte specifikace tabulek a dotazů raději do přílohy. Architekturu dat a bezpečnostní specifikace však klient musí potvrdit.

Já dokumentuji datovou architekturu obvykle pomocí jisté kombinace diagramů a slovního popisu. To je tedy další případ, kdy se uvedené dvě formy sdělení vzájemně podporují a vhodně se doplňují. Bezpečnostní požadavky můžeme dokumentovat pomocí jednoduchého slovního popisu, i když někdy nám pomůže malý náčrtek či obrázek – zejména u složitějších bezpečnostních struktur.

## Uživatelské rozhraní

Je velice dobré si připravit alespoň stručný dokument s návrhem uživatelského rozhraní. Tento návrh musíme ovšem s klientem rozebrat ještě před zahájením vlastních prací na uživatelském rozhraní, o nichž hovoříme ve zbytku této knihy. U malých systémů však naopak sestavení takového návrhu může celý proces návrhu naprostě zbytečně zdržovat.

I když se třeba nemusíme pouštět zrovna do nějakého formálního návrhu uživatelského rozhraní, bývá velice často dobré načrtnout pář ukázkových obrazovek, označených jako „NÁVRH“ nebo „VZOR“. Pomocí takových ukázkových obrazovek si uživatel snáze představí navrhovaný systém i vizuálně. Musíte být ovšem opatrní. Přestože třeba stokrát řeknete, že tyto obrazovky jsou jen předběžný návrh a že se v ostrém systému mohou změnit, i tak vzbuzují v uživateli určitá očekávání. Jestliže pak hotový, ostrý systém vypadá výrazně jinak než tento návrh, pak v podstatě veškerá vaše dobré míněná iniciativa jen zbytečně kalí vodu.

Očekávání, která u uživatele prvotním návrhem vzbudíme, představují problém obvykle jen tehdy, pokud se funkce, pro něž jsme navrhli ukázkovou obrazovku, z projektu nakonec vypustí. Často krát jsem se setkala s uživateli, kteří v konečné verzi systému očekávali všechny obrazovky navržené při analýze prvotních požadavků, přestože později sami viděli a odsouhlasili specifikaci rozhraní, ve které již zcela chybely. Naučila jsem se proto uvádět do dokumentace i tyto vypuštěné obrazovky – nejlépe do samostatné části textu, nazvané třeba „Funkce vypuštěné ze systému“. Vytvořit docela stručný popis vypuštěných funkcí a uvést důvody, proč je systém nebude podporovat, není až tak pracné; je to opět docela dobrá pojistka proti nečekaným námitkám.

Nadefinované uživatelské rozhraní musíme dále přednест uživateli. K tomuto přednesení můžeme využít následující dva mechanismy: prototyp a specifikaci rozhraní. Já si téměř vždy připravuji obojí – když nic jiného, tak sestavení nefunkčního prototypu je pro mě nejsnazším způsobem, jak si připravit ilustrace do dokumentů, na jejichž základě pak stavím strukturu konečné specifikace.

## Prototypování rozhraní

Nejlepším nástrojem pro přednesení návrhu rozhraní nového systému uživateli je podle mého názoru prototyp rozhraní. Řada uživatelů, zejména pak uživatelů s malými zkušenosťmi v oblasti práce s počítači, si nedokáží z několika obrazovek vytištěných na papíře dobře představit, jak vlastně bude hotový systém vypadat a jak bude fungovat. Jestliže jim dáme do rukou prototyp rozhraní, nemusí si nic složitě představovat a odvozovat.

Prototypy mají různé podoby a velikosti, přičemž se dají využít k řadě různých účelů. Jednou z nejjednodušších forem prototypu je řada vzájemně propojených maket obrazovek a nabídek, které modelují tok běhu ostrého systému. Prototyp rozhraní neobsahuje v zásadě žádný programový kód – stačí do něj vytvořit pouze kód, který je potřeba k propojení obrazovek. Formuláře a obrazovky obsahují veškeré zamýšlené ovládací prvky, ty však nejsou svázány s daty, ani nejsou jinak funkční. Ani příkazy nabídek nic nedělají – snad jen s výjimkou příkazů, které zobrazí určitý dialog. (Upřímně řečeno, toto není až tak úplně pravda. Já obvykle přidávám k nabídkám doplňkové položky, které říkají třeba: „Tento příkaz bude dělat to a ono. Tato funkce není v prototypu implementována.“)

Jedinou výjimkou z pravidla „zóny bez programového kódu“ je situace, kdy je fyzické zobrazení závislé na konkrétních datech. Řekněme například, že navrhujeme obrazovku pro pořizování a edityování informací o zákazníkovi; podrobné informace na obrazovce se ale liší podle toho, jestli je daným zákazníkem osoba (jednotlivec) nebo firma (společnost). Viditelnost některých ovládacích prvků se třeba v takovém případě rozhodně měnit podle výběru z nějakého přepínače. A takové chování musíme implementovat již v prototypu.

Prototyp rozhraní sestavují obvykle přímo v tom front-end nástroji, pomocí kterého budu vytvářet i hotový systém. Tento postup je rychlý a efektivní, skrývá v sobě ale jisté nebezpečí. Zde totiž sestavujeme prototyp, nikoli celý systém, takže si můžeme docela klidně usnadnit život různými zkratkami a „fintami“, které jsou ale v provozním, ostrém kódru naprostě nepřijatelné. Zároveň je ale velice těžké odolat pokušení převzít navržený prototyp jako základ ostrého systému.

Konec konců, všechny ty obrazovky a nabídky jsou již sestaveny, takže proč bychom ztráceli čas jejich novým vytvářením – proč bychom je nemohli jednoduše hned využít? Chyba lávky. Tyto obrazovky a nabídky jsou zatím jen *prototypované*, nejsou ještě sestavené. Jestliže prototyp rozhraní skutečně převezmete za základ ostrého systému, podstupujete riziko, že v něm ponecháte i ony zkratky, které v ostrém systému nemají co dělat, *ujišťuji* vás, že se vrátí a budou vás po nocích strašit.

Někteří návrháři právě z tohoto důvodu doporučují provádět návrh rozhraní pomocí jiného nástroje – nejlepší je nevytvářet je pomocí programového prostředí, jako je Microsoft Access nebo Microsoft Visual Basic, ale pomocí nějakého kreslicího nástroje. Vezměte jakýkoli nástroj, který se vám nejlépe hodí. Pro mne to znamená takový programový nástroj, se kterým pracují dnes a denně. Pro vás je to třeba nějaký kreslicí nástroj, nebo dokonce prezentační nástroj, jako je Microsoft PowerPoint. (Měla jsem jednoho klienta, který si obrazovky všech mých prototypů pečlivě přenesl do PowerPointu, kde je využil pro své interní prezentace.) Důležité je si i zde jasně uvědomit, co děláte – zatím se nacházíte ve fázi dokumentace návrhu, *nikoli* výstavby systému.

## Specifikace rozhraní

Prototyp rozhraní je zajisté vynikající nástroj, protože dává uživateli alespoň nějakou představu o práci budoucího systému. Jeho možnosti jsou ale zcela záměrně omezené. Dobrou specifikaci rozhraní proto ani zdaleka nemůže nahradit. (Opačné tvrzení zde ovšem neplatí. Pečlivě sestavená specifikace rozhraní často prototyp vůbec *nepotřebuje*.)

Podobně jako dokumentace datového modelu musí i specifikace uživatelského rozhraní obsahovat technické informace, které klient nemůže zcela pominout. Na tomto místě platí proto stejná doporučení – snažte se technickou terminologii používat co nejméně a pokud možno se pokusete všude od hlavního textu dokumentu oddělit skutečné technické záležitosti, které může klient s přehledem ignorovat.

Jestliže máte vytvořen prototyp rozhraní, je již sestavení specifikace rozhraní docela jednoduché. Já tuto specifikaci sestavují z bitmapového obrázku každé jednotlivé obrazovky, slovního popisu jejich významu a dále tabulkového výpisu ovládacích prvků, k nimž uvádím zdroj dat (pokud je nějaký) a případné zpracování, které formulář provádí. Pokud jste prototyp uživatelského rozhraní nevytvořili, můžete základní podobu rozhraní popsát jiným způsobem; zbývající informace jsou ale stále stejné.

Ve většině systémů je vhodné stručně popsát také tok řízení systému. Pokud bude navrhovaný systém podporovat řadu různých pracovních procesů, sestavte do dokumentu pro každý pracovní proces odpovídající model toku řízení obrazovek. Nejjednodušší způsob spočívá v doplnění diagramů pracovních procesů o textové poznámky.

## Správa pozdějších změn

Dokumentace návrhu projde nejspíše sérií několika revizí a teprve poté se konečně ustálí. Až tento dokument (nebo množina dokumentů) dosáhne stabilního stavu – ale rozhodně ještě před tím, než se pustíme do další fáze projektu – je třeba je podřídit mechanismu řízení změn.

Všimněte si, že „řízení změn“ neznamená nutně „zmrazení specifikace“; domnívám se totiž, že něco takového není reálné ani pro opravdu malé systémy. Jestliže již předem počítáte s nevyhnutelnými změnami, dlouhodobě si rozhodně usnadníte život.

Řízení změn je možné realizovat několika různými způsoby. Při změnách se obvykle bráním ediací (úpravě) příslušných dokumentů; mnohem snazší je podle mého názoru vydat změnové listy, které fungují jako doplněk původního dokumentu. Pokud jsou změny opravdu obsáhlé, má smysl celý dokument, nebo alespoň jeho některé části, přepsat a zcela nahradit; něco takového se ale stává dosti zřídka.

Dokážeme-li zavést nějaké centrální umístění hlavní specifikace, má smysl dokument upravovat nebo opatřit poznámkami. V textovém procesoru, jako je například Microsoft Word, můžeme tyto úpravy provést pomocí revizních nástrojů. Takový dokument pak můžeme umístit do sdíleného sítového adresáře, nebo jej zveřejnit na podnikovém intranetu.

Jediný problém, se kterým jsem se setkala při umístění dokumentu do centrálního místa, je otázka, jak zajistit, že lidé budou skutečně pracovat vždy s aktuální verzí dokumentu, a nebudou si jej pouze tisknout a ručně čmárat do vytiskných kopií. Zavedení vůbec nějakého plánu řízení změn je ale obvykle mnohem důležitější než jeho podrobná implementace.

## Speciální nástroje

Součástí balíku programovacích nástrojů od Microsoftu jsou dvě utility, které jsou užitečné pro vytváření udržování dokumentace návrhu: je to Microsoft Visual SourceSafe a Microsoft Visual Component Manager. První z nich, Visual SourceSafe, je určen především pro řízení změn ve zdrojovém kódu během trvání vývojového projektu; stejný nástroj však můžeme použít i pro správu dokumentů spojených s vyvíjeným projektem.

Já s výhodou využívám nástroj Visual SourceSafe, a to zejména pro údržbu dokumentace v projektech, na kterých pracuje více různých vývojářů. Díky procesu kontroly dokumentů před povolením úprav zde uživatelé nemohou přepsat výsledky práce někoho jiného a zároveň mají k dispozici vždy nejnovější verze všech dokumentů.

Druhý ze jmenovaných nástrojů, Visual Component Manager, je front-end rozhraní k Microsoft Repository. Dodává se společně s verzí Enterprise Edition v prostředí Microsoft Visual Studio. Tento nástroj je určen pro administraci dokumentů a komponent programového kódu pro jejich zveřejnění, nikoli pro správu během vývoje. Visual Component Manager umožňuje vytvořit pro každý projekt samostatnou databázi; mně se na něm líbí zejména to, že nabízí pohodlnou správu všech souvisejících dokumentů na jediném místě.

Jestliže daný projekt vyvíjíme v prostředí Microsoft Visual Studio, můžeme pomocí stejné databáze projektu zároveň administrovat veškeré komponenty programového kódu; všechny informace máme pěkně pohromadě. Visual Component Manager však naneštěstí není integrován s Microsoft Accessem, i když mechanismus Repository, na němž je postaven, se dá rozšiřovat a teoreticky by se dal odpovídajícím způsobem upravit.

### Stručné shrnutí

V této kapitole jsem shrnula některé zásady pro přednesení návrhu systému klientovi a vývojovému týmu. Můžete je brát třeba jako názor jednoho vývojáře. Mně a mé práci zde popsané strategie plně vyhovují, vy si je ale upravte podle potřeby konkrétních pracovních postupů a podle požadavků vašich klientů – zde to platí oproti jiným rozebíraným oblastem dvojnásob.

Nacházíme se sice na konci druhé části této knihy, přesto jsme zatím diskusi o procesu návrhu databází nedokončili. V části 3 přejdeme ke snad nejdůležitější komponentě každé aplikace: je to uživatelské rozhraní.

# 3

## ČÁST

NÁVRH UŽIVATELSKÉHO  
ROZHRANÍ



# ROZHRANÍ JAKO DŮLEŽITÝ PROSTŘEDNÍK

12

Po dokončení analytických úkolů, které jsme si popsali v předcházející části knihy, bychom již měli velice dobře vědět, co všechno má navrhovaný systém dělat. Ve třetí části knihy se budeme věnovat některým otázkám, které souvisí s vytvářením rozhraní systému.

V kapitole 12 začneme přehledem obecných postupů pro návrh rozhraní a různými modely, které je třeba zvážit. Do této kapitoly se však vešlo jen omezené množství informací. Pro další zdokonalení v návrhu uživatelského rozhraní se proto obrátěte na jiné zdroje. Několik vynikajících knih na toto téma je uvedeno v přiloženém seznamu literatury a další vám zcela nepochybňě nabídou v jakémkoli knihkupectví technické literatury.

## Efektivní rozhraní

Uživatel *považuje* za systém přímo uživatelské rozhraní systému; všechno ostatní jsou prostě „prkotiny“, které může s ledovým klidem ignorovat. Dobrý návrh uživatelského rozhraní je tudíž pro úspěch nebo pád projektu životně důležitý. Postavte jej správně a uživatelé vám odpustí sem tam nešťastně zvolenou implementaci. Navrhniť jej nevhodně a už se ani nebudou zajímat, nakolik efektivní programový kód se v systému skrývá.

Jistou ironií osudu je, že pokud uživatelské rozhraní navrhnete *správně*, skoro si toho nikdo ani nevšimne. Opravdu elegantní rozhraní jsou totiž tak říkajíc neviditelná. Nikdo si ale nemusí všimnout ani toho, že je rozhraní postaveno nesprávně. Rozhraní řady počítačových systémů, a zejména pak databázových systémů, jsou skutečně natolik hloupá, že se váš systém stane dalším z řady průměrných, tuctových, ničím nevynikajících počítačových systémů, na jaké jsou již uživatelé zvyklí.

Jestliže si tedy rozhraní v podstatě ani nikdo nevšimne, proč bychom se jím měli tolík trápit? Naopak: proč ne? Je to přece konec konců naše práce. Vím, že vás teď budu asi poučovat jako maminka kdysi v mládí, ale když už se vůbec pouštíte do návrhu počítačových systémů, copak nemá smysl snažit se je navrhovat nejlépe, jak umíte? Navrhnout opravdu efektivní rozhraní znamená mnohem více práce, než jen prostě k databázi „připlácenout“ nějaké formuláře či jiný front-end. Implementace skutečně efektivního rozhraní bývá často také pracnější, i když to nutně nemusí být pravda. Takové rozhraní může mít kromě toho opravdu velké přínosy, které zdaleka nejsou z kategorie „jedinou odměnou je čestný pocit z dobře vykonané práce“. Efektivní uživatelské rozhraní minimalizuje dobu, kterou uživatel potřebuje ke zvládnutí systému, a dobu, kterou my potřebujeme k jeho implementaci. Zlepšení produktivity práce je v hotovém, implementovaném systému o to větší, pokud jej uživatel nemusí dálé pracně přizpůsobovat svým potřebám. Obě tyto otázky se obvykle promítají do navrhovaných cílů systému. Rozhodně pak mají vliv na neslavnou základní úroveň požadavků.

Efektivní rozhraní, které dobře naplňuje očekávání uživatele a dobře podporuje pracovní procesy, zároveň minimalizuje potřebu další externí dokumentace, která je vždy nákladná. A i když třeba uživatelé nějak vědomě nepostřehnou, jak úžasné má váš systém uživatelské rozhraní, určitě jim neunikne, že tento systém pracuje podle všeho lépe než systém, který navrhla firma UživatelijsouBlbci, s. r. o. To již může ovlivnit vaši základní úroveň při návrhu dalšího projektu.

Co tedy dělá rozhraní efektivním? Podle mého je efektivní takové rozhraní, které pomáhá uživatelům splnit jejich úkoly a jinak se jim nepletí do života. Efektivní rozhraní neklade na uživatele žádné svoje vlastní požadavky. *Nikdy* nenutí uživatele, aby hráli podle jeho pravidel; hraje podle pravidel stanovených uživatelem. K tomu, aby uživatel mohl s efektivním rozhraním pracovat, se nemusí učit hromady nějaké nezajímavé dokumentace. A nakonec, efektivní rozhraní se nikdy nechová neočekávaným způsobem.

Na všechny tyto tři principy se v této kapitole podíváme podrobněji. Nejprve si ale stručně řekneme něco málo o modelech, které je vhodné zvážit při návrhu uživatelských rozhraní.

## Modely rozhraní

Alan Cooper ve své vynikající knize *About Face: The Essentials of User Interface Design* popisuje způsoby, jakými uživatelé uvažují o systémech (a způsoby, jakými systémy uvažují o uživatelích) pomocí třech základních modelů: je to mentální (duševní) model, symbolický model (manifestační model, model projevů) a implementační model. Tyto tři různé způsoby vnímání systémů jsou mimo jiné velice užitečné pro rozhodování o návrhu jejich uživatelského rozhraní.

*Mentální model* uživatele popisuje, co si uživatel *myslí*, že se v systému děje. Tento model velice často neodpovídá tomu, co se v něm děje ve skutečnosti, nijak to ale nevadí. Mohu mít třeba velice vágní představu, že moje tělo při výrobě energie „spaluje“ potravu zhruba stejným způsobem, jakým se v automobilovém motoru spaluje benzín. Vím, že skutečné procesy probíhají fakticky úplně jinak, ale to mě nezajímá. Vím jenom, že pokud chci, aby pracoval motor mého auta, musím do něj nalít benzín, a pokud má pracovat moje tělo, musím mu zajistit přísun potravy. Potud mi tedy můj mentální model plně vyhovuje.

Totéž platí i pro počítačové systémy. Ať už píší na stařičkém psacím stroji nebo na počítači v textovém procesoru, vždy píší tím způsobem, že stisknu klávesu se znakem a tento znak se objeví v textu. Mentální model psaní textu je tedy stejný. Ve skutečnosti ale pochopitelně probíhá něco úplně jiného. To, co probíhá ve *skutečnosti*, je již záležitostí *implementačního modelu*. Veškeré ony základní operace, kdy se v psacím stroji pohybují nějaké páčky a v počítači běží nějaký programový kód, jsou součástí implementace. O to se uživatel nezajímá – a ani jej nesmíme nutit se o to zajímat.

Uživatelské rozhraní je *symbolický model*, který se nachází na pozici mezi mentálním modelem uživatele a implementačním modelem vývojáře. Chcete-li, můžeme jej charakterizovat jako model procesu, jakým se systém *jeví* (projevuje) uživateli. Jedním z cílů návrhu uživatelského rozhraní je skrýt co nejvíce detailů implementačního modelu. Ideální systém přesně odpovídá mentálnímu modelu uživatele. Této *přesné shody* s mentálním modelem uživatele však často není možné dosáhnout. Čím více se jí ale dokážeme přiblížit, tím lépe.

Pokud se o počítače zajímáte natolik, že jste se rozhodli přečíst si tuto knihu, znamená to mimojiné, že váš mentální model neodpovídá mentálnímu modelu uživatelů. Můj mentální model práce s textovým procesorem vypadá tak, že zmáčknutí vybranou znakovou klávesu a ASCII kód tohoto znaku se uloží na nějaké místo v paměti RAM. Tento model je dosti vzdálený od skutečného implementačního modelu, velice dobře ale připomíná způsob, jakým si chod věcí představuje průměrný úředník.

Zde se ovšem skrývá jedno velké nebezpečí návrhu rozhraní: i když třeba nejste do prací na implementaci systému přímo zapojeni, téměř určitě o nich alespoň něco víte. Budět se tedy musíte naučit na veškeré implementační detaily dočasně zapomenout, nebo si některého uživatele odchytit jako „pokusného králíka“ a požádat jej, aby vám popsal svůj mentální model.

Nejlepší je vykonat s pomocí prototypu systému formální testy použitelnosti. Provádět tyto testy opravdu důkladně sice někdy nelze, přesto však stojí za to se jakkoli pokusit alespoň o nějakou analýzu použitelnosti. Jestliže jste prototyp systému sestavili, dejte jej do rukou několika uživatelům a nechejte je si s ním pohrát. Zeptejte se, co si myslí, že se v systému děje. Určitě se dočkáte překvapení. Pokud žádný prototyp vytvořen nemáte, můžete se ptát na podobný systém, nebo dokonce použít „papírovou maketu“. S poslední jmenovanou technikou jsem sice nezaznamenal příliš mnoho úspěchů. Přišla jsem totiž na to, že pokud má uživatel na papíře vedle sebe interaktivní obrazovku a sestavu, je z toho často zmaten.

Jakmile sesbíráte tolik informací, kolik jen můžete, popřemýšlejte, jestli implementační model (o který nám zde ve skutečnosti běží) někde náhodou nenaruší mentální model uživatele a poté se pokuste případné problémy vyřešit. Používáte nesprávnou terminologii? Vždy pracujte se stejnými pojmy, jaké znají i vaši uživatelé. Nenutíte náhodou uživatele k tomu, aby říkali „editovat zážnam“, když oni chtějí říci „změnit adresu“? To může být nejen problém terminologie, ale také problém ve struktuře systému, o níž budeme podrobněji hovořit v následující kapitole.

## Úrovně uživatelů

Nedovedu si představit, že by se někdo úmyslně rozhodl postavit systém, který není „uživatelsky přátelský“. Použitelnost systému nemusí mít v zadání příliš vysokou prioritu, sotva kdo by ale vytvářel nějaký „uživatelsky nepřátelský“ systém. Problém je v tom, že výraz „uživatelsky přátelský“ je jedním z oněch pěkných pojmu, které byly sice vytvořeny s dobrým úmyslem, ale které v podstatě příliš mnoho neznamenají, takže nějakou rozumnou definici musíte hledat kdekoliv.

Uživatelsky přátelské neboli uživatelsky příjemné rozhraní se často definuje jako rozhraní, které se „snadno naučí“ nebo se kterým se „snadno pracuje“. Jestliže zatím ponecháme stranou otázku, co to přesně znamená „snadno“, musíme se ptát: „Snadno pro koho?“ Se systémem, který se začátečník snadno naučí, se expertovi nemusí nutně být výhodou dobré pracovat. Nejlépe je proto zvážit potřeby všech úrovní uživatelů a každému z nich se přizpůsobit jinou podobou rozhraní.

### Začátečník

Každý z nás je někdy v něčem začátečníkem. Jen málo lidí zůstane na této úrovni navždy – většinou se přes pozici „nováčka“ dostanou do fáze mírně pokročilého, nebo celý váš systém zahodí a pustí se do nějakého jiného. Proto nesmíte do systému zapracovat takovou podporu začátečníků, která by pokročilým uživatelům překážela.

Začátečník se nejprve musí dozvědět, *co* vlastně váš systém dělá, a teprve poté se může začít učit, *jak* s ním má pracovat. Nejlepší je uvádět tyto informace někde mimo samotný hlavní systém. U jednoduchých systémů plně postačuje například úvodní dialog s popisem systému. (Nezapomeňte ale implementovat určitou funkci, pomocí které se dá dialog později trvale odstranit.) Pro složitější systémy je vhodné vytvořit například ukázku s průvodcem.

Elektronická nápověda se pro začátečníky příliš nehodí. Takový člověk třeba ani neví, že existuje, nebo s ní neumí pracovat. Několikrát jsem ale měla úspěch s elektronickým průvodcem uživatele, který se spouštěl prostřednictvím odkazu v úvodním dialogu a v nabídce Nápověda. Pokud má mít takový průvodce pro začátečníka nějaký smysl, musí být orientovaný na úkony. Začátečníka nezajímá, co znamená nějaká „položka nabídky“; on se přece chce naučit, jak má vytvořit fakturu.

### Pokročilý

Většina uživatelů většiny systémů spadá do prostřední kategorie, tedy do kategorie pokročilých. Takový pokročilý uživatel již ví, *co* všechno systém umí, někdy si ale neumí vzpomenout, *jak* to dělá. To je zároveň skupina, kterou musíte v uživatelském rozhraní přímo podporovat. Rozhraní systému Microsoft Windows však naštěstí nabízí řadu různých nástrojů, které těmto uživatelům pomáhají.

Jedním z nejlepších způsobů, jak částečně pokročilému uživateli připomenout možnosti a funkce systému, je dobré navržený systém nabídek. Rychlým pohledem na dostupné položky nabídek uživatel snadno zjistí, jaké funkce má momentálně k dispozici a zároveň se může rovnou pustit do odpovídajícího úkolu.

Druhou vynikající úrovní podpory pokročilých uživatelů je elektronická (online) nápověda. Vytváření elektronické nápovědy je již mimo rámec této knihy. Na tomto místě však alespoň připomenu, že většina uživatelů bude do nápovědy přistupovat především pomocí rejstříku. To znamená, že rejstřík musí být pokud možno co nejúplnejší.

## Expert

Uživatelé-experti vědí, co mají udělat a jak to mají udělat. Zajímá je tedy hlavně to, jak požadované operace provést *rychle*. Tato skupiny uživatelů bude proto tím spokojenější, čím více zabudujete do systému různých zkratek. Uživatelé-experti jsou podle mých zkušeností obvykle orientovaní na klávesnici, takže pokud se chcete této skupině zavděčit, nezapomeňte do systému zabudovat možnost rychlého procházení pomocí klávesnice a horkých kláves.

Experti dále oceňují možnost dalšího uživatelského přizpůsobení pracovního prostředí. Vytvořit takové funkce může být ale opravdu nákladné, takže nejprve je nutné zvážit, jaké jsou vlastně jejich přínosy. Jestliže se *opravdu* rozhodnete implementovat v systému jistou úroveň uživatelského přizpůsobení, byť se třeba jedná jen o možnost uspořádání oken na obrazovce, nezapomeňte provedené změny udržovat v platnosti mezi různými seancemi. Nic uživatele nenaštve víc, než když si po každém spuštění programu musí všechna okna uspořádávat znovu.

## Jak zapojit uživatele

Někde za pojmem „uživatelsky přátelský“ se v té naší pěkné, ale v podstatě bezcenné terminologii pohybuje také pojem „uživatelsky orientovaný“. Co to znamená? Na rozdíl od pojmu „uživatelsky přátelský“ má tento výraz přece jen nějaký praktický význam, i když i ten je poměrně vágňí. Systém je uživatelsky orientovaný, pokud vždy reaguje na požadavky uživatele a pokud *nikdy* nevnucuje určitý způsob práce.

Tento princip si zřejmě nejsnáze vysvětlíme na příkladu. Jeden můj známý, vynikající vývojář, jehož si opravdu velice vážím, popisoval metodu, pomocí které zajistí, že uživatelé zadávají data přesně v tom pořadí, jaké je pro systém nejhodlnější. Na formuláři nejprve uzamkne všechny ovládací prvky s výjimkou prvního; jakmile do něj uživatel zadá požadovaný údaj, odemkne druhý ovládací prvek, pak třetí a tak dále.

Uvedený postup je ale nejen strašně vzdálený od uživatelsky orientovaného systému, ale nakonec ani nemůže fungovat. Uživatelé totiž nezadávají data jen v jednom stejném pořadí, a velice často k tomu mají svůj důvod – určitý údaj například není možné vůbec zjistit, nebo je to v daném okamžiku příliš pracné. Pokud je výše popsaným způsobem donutit něco zadat, opravdu něco zadají. Napiši do systému jakýkoli nesmysl, který aplikace přijme. V podstatě jste je tedy obvinili, potrestali, a nakonec jste nedosáhli vůbec ničeho.

O této otázce budeme hovořit podrobněji v kapitole 16, kde se budeme zabývat datovou integritou. Umělé vynucování datové integrity je primárním způsobem, jakým databázové systémy bojují s uživateli o kontrolu. Druhý způsob spočívá v přílišné modalitě (orientaci na módy neboli režimy práce). *Mód* neboli režim práce je jistá systémová podmínka, která omezuje možnosti uživatele. Databázové systémy tak znají klasické módy přidávání, editace a prohlížení dat. Systém, v němž se uživatel musí z režimu prohlížení záznamu vrátit nejprve do hlavní nabídky, přestože hodlá editovat ten stejný záznam, je přímo směšně neefektivní. Pokud jej budeme nutit před zahájením úprav vybrat určitou položku nabídky nebo klepnout na nějaké tlačítko, také to není o moc lepší.

Řada vývojářů ale naneštěstí tuto modalitu bere za samozřejmou, za jakési „zvykové právo“. Snad jako marný pokus o ochranu uživatelů před neúmyslně provedenými změnami, nebo snad proto, že položky nabídek Přidat, Upravit a Zobrazit se v systémech používají už dobrých 20 let, zachová-

vávají toto pojetí navěky i v systému Windows, kde zcela není na místě. Rozhodně vám proto doporučuji pracovat s tím, že uživatel ví, co dělá. Pokud chce uživatel změnit záznam, nechejte jej změnu provést. Nenuťte uživatele k tomu, aby si od vás musel vyžádat nějaké zvláštní povolení.

Jestliže ale uživateli máme nabídnout takovou svobodu, musíme jim současně nabídnout jakousi bezpečnostní záchrannou síť. V Microsoft Accessu i v Microsoft Visual Basicu se docela snadno daří implementovat operace vrácení zpět („undo“) o více úrovních. Do programu můžeme také doplnit volbu nabídky Vrátit se k naposledy uloženému stavu, která umožní uživateli zrušit veškeré změny v aktuálním záznamu.

Osobně velice nerada žádám uživatele o potvrzení změn například při uložení záznamu, i když v některých situacích je takový postup ospravedlnitelný. Pro většinu uživatelů je totiž celá myšlenka nějakého „ukládání“ naprostě cizí. Vzpomeňte si na jejich mentální model. Uživatel prostě udělal nějakou změnu a vy se ho najednou ptáte, jestli tuto změnu opravdu chce provést – to je přece zmatené. Většina lidí si navíc na toto chování poměrně rychle zvykne a jakmile zjistí, co to přesně znamená, naučí se v každém dialogu s žádostí o potvrzení zcela automaticky, bezmyšlenkovitě klepnout na tlačítko „OK“. Dialog je tedy nakonec v podstatě k ničemu.

Takové potvrzovací zprávy či žádosti jsou tedy dalším příkladem, kdy na uživatele klademe zbytečné požadavky, aniž bychom něčeho smysluplného dosáhli. V několika málo případech, kdy jsem potvrzovací zprávy do systému skutečně implementovala (obvykle to bylo jen na výslově přání klienta), jsem zároveň vytvořila mechanismus, pomocí něhož se dá vypnout (tato volba bývá zpravidla začleněna do dialogu Vlastnosti nebo Možnosti).

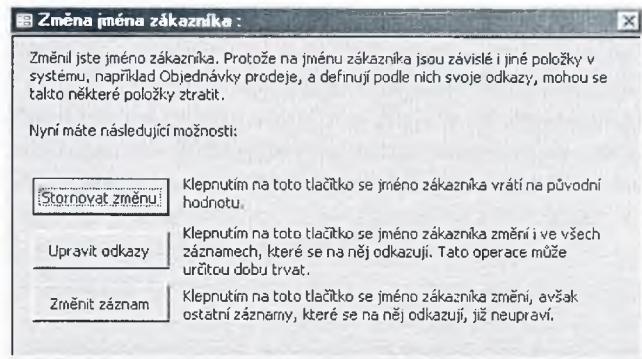
Některé rozsáhlé změny se však vrátit zpět nedají, alespoň ne snadno. Neúmyslná změna hodnoty jednoho pole v záznamu se dá docela lehce opravit. Odstranění všech záznamů z tabulky již ale zpět nevrátíme. Proto jestliže uživatelům umožníme provádět i věci, které považujeme za opravdu nebezpečné, je nejlepší nabídnout jim možnost vrácení celé operace zpět. Pokud to není možné, vyžádáme si od nich rozhodně definitivní potvrzení.

Zde ale musíte dosti citlivě nadefinovat, co je to „opravdu nebezpečné“, a uživateli musíte podat dostatečně podrobné informace o důsledečích jimi zvolené operace. Okno se zprávou, uvedené na obrázku 12-1, uživatele pouze vyděsí.



Obrázek 12-1 Toto okno se zprávou je přímo ukázkově k ničemu

Druhé okno se zprávou, které vidíme na obrázku 12-2, je již mnohem lepší. Nejenže vysvětluje celou situaci pomocí pojmu známých uživateli, ale zároveň mu nabízí i jiné možnosti, než OK a Storno (které si ostatně průměrný uživatel vysvětluje jako „jsem idiot“ a „jsem idiot, pokorně se omlouvám“).



Obrázek 12-2 Toto okno se zprávou vysvětluje důsledky akce, kterou uživatel požaduje, a nabízí mu několik možností řešení

## Minimalizace zatížení paměti

Lidé si nepamatují věci příliš dobře – ostatně to je jeden z důvodů, proč používáme počítače. Z toho vyplývá, že u dobrého uživatelského rozhraní by uživatel neměl být nucen si zapamatovat více, než kolik je opravdu nezbytně nutné. Určitě se musí naučit (a zapamatovat si), co všechno systém umí. Potřebují si také zapamatovat některé věci ohledně práce se systémem. Vždy je ale nutné množství informací, které si uživatel musí zapamatovat, snížit na minimum.

Jedním z možných přístupů je dodržovat standardy a konvence rozhraní systému Windows. Jednu množinu takových konvencí definuje kniha *The Windows Interface Guidelines for Software Design*. Ke změnám těchto zásad zpravidla nebude mít důvod, a zejména pak změny nesmíme provádět zcela svévolně. Produkty rodiny Microsoft Office – zejména pak Microsoft Access – definují další de facto standardy. Ani od těchto standardů se neodchylujte, ledaže byste k tomu měli opravdu dobrý důvod. Jinak své uživatele asi nepotěšíte.

Dejme tomu, že máte báječné navigační ikony a že jste přesvědčeni, že jsou *mnohem* lepší než implicitní „video“ tlačítka v Microsoft Accessu. Co ale chudáci uživatelé. Ti se jednou naučili, že se stiskem tlačítka dostanou na nový záznam. A vy je teď nutíte si pamatovat, že zrovna ve *vašem* programu (ale nikde jinde) musí kliknout na tu neuvěřitelně rozkošnou fotku s kočičkou a kotátky.

Zásady pro tvorbu rozhraní v systému Windows mají ale samozřejmě také své hranice, a zdaleka tudíž nedokáží postihnout veškeré situace. Ve většině případů je možné obecné zásady rozšířit směrem k potřebám konkrétního problému. Pokud se někde musíte rozhodnout, snažte se těchto zásad držet. V další části textu si o významu konzistence povíme něco více.

Mělo by být také jasné, že pokud se snažíte co nejvíce zmenšit množství informací, které si uživatel musí zapamatovat, neměl by jej systém nutit zadávat stejnou informaci dvakrát. Již jsme si říkali, jak se dají v této situaci využít implicitní hodnoty. To znamená, že pokud má uživatel postupně pracovat s několika různými formuláři, je důležité přenášet také veškeré relevantní informace.

Uživatele bychom navíc nikdy neměli žádat o přímé zadání informace, která se dá rozumně vybrat z nějakého seznamu. Počítače jsou přímo notoričtí puntičkáři, zatímco uživatelům je úplně jedno, jestli byl záznam zákazníka Standy Procházky pořízen jako „Standa Procházka“, „S. Procházka“, nebo třeba „Standa Procházka ml.“ Nezapomeňte ale, že jsem řekla něco o „rozumném“ výběru. Aby uživatel musel čekat, než mu systém do kombinovaného pole naplní 65 000 záznamů, to zcela určitě rozumné není. Rozumné je nabídnout mu funkci filtrování seznamu, pomocí níž si vybere menší podmnožinu, ze které se již dá vybírat mnohem snáze.

Jestliže uživateli nabídnete nějaký seznam k výběru, přidejte k němu co nejvíce doplňujících informací, a to zejména pokud jednotlivé položky seznamu nejsou nutně jednoznačné. Uživatel si tak určitě nebude pamatovat, že „Pepa Kolář“ bydlí v Liberci, zatímco „Pepík Kolář“ třeba v Karlovačově Řečici. Seznamy a kombinovaná pole mohou v Microsoft Accessu obsahovat i více sloupců dat. Ve Visual Basicu stačí odpovídající pole spojit pomocí operace zjednocení.

V obou uvedených prostředích je vhodné zobrazit další podpůrné informace například v kontextové nabídce. Po technické stránce je nejlepší zobrazit tyto informace příkazem, například příkazem „Detaily“ z kontextové nabídky. (Všimněte si, že za názvem tohoto příkazu neuvádíme symbol výpustku neboli tří tečky, přestože zobrazuje dialog. Zapsat zde výpustek je poměrně běžnou chybou.) Pokud však podrobné informace nejsou nijak obsáhlé, můžeme je vypsat přímo do kontextové nabídky. Tím uživateli ušetříme alespoň nějaký čas.

A nakonec, uživatele bychom nikdy neměli nutit si zapamatovat nějaké složité, obskurní schéma kódování. Jedinečnost záznamu můžeme v tabulce například zajišťovat pomocí automatického, systémem generovaného čísla; o jeho existenci by se však uživatel ve většině případů vůbec neměl dozvědět. Nemá-li číslo zároveň nějaký vnější význam, například číslo faktury, vůbec jej nezobrazujte.

Řada organizací má vypracovány seznamy zkratky, které vyjadřují různé věci, jako například kategorie výrobků či oblasti prodeje. Jestli se tyto zkratky mají nebo nemají v systému používat, o tom rozhodují podle jednoduchého pravidla: záleží na tom, jestli je lidé používají i v běžném hovoru. Vyvarujte se proto řešení, kdy sice lidé hovoří o „jihovýchodním regionu“, ale vy je v systému nutíte zadávat zkratku „JVR“. V systému používejte zkratku „JVR“ jen tehdy, pokud ji pracovníci skutečně používají při normálním rozhovoru mezi sebou. I v takovém případě bych ale doporučovala povolit do pole oblasti zadání testu „JVR“ i „jihovýchod“.

## **Jak být konzistentní**

Konzistence v uživatelském rozhraní znamená mnohem více, než jen mít nabídky Soubor a Úpravy na správném místě. Důsledný neboli konzistentní musí být také způsob, jakým systém komunikuje s uživatelem. Při návrhu databázových systémů si musíme dát pozor na tyto tři oblasti: jakým způsobem se uživatel pohybuje mezi záznamy v tabulce, jak složité entity se v něm reprezentují a jak uživatel zahajuje operaci přidávání a editace záznamů.

Většina systémů se skládá z řady formulářů, které reprezentují jednotlivé nejdůležitější entity systému. Konkrétní systém může tak například obsahovat formulář Zákazníci, formulář Výrobky a formulář Objednávka prodeje. Zde se musíme rozhodnout, jakým způsobem se uživatelé budou pohybovat v příslušné podkladové množině záznamů. Nemáme-li nějaký opravdu pádný důvod

pro zavedení několika různých mechanismů, je třeba si zvolit jen jednu metodu a tu důsledně dodržovat v každém z formulářů.

Svázané formuláře v Microsoft Accessu i ve Visual Basicu zobrazí implicitně první záznam podkladové množiny záznamů. Obsahují přitom navigační tlačítka, pomocí kterých se uživatel může v množině záznamů dále pohybovat. Toto implicitní rozhraní můžeme z různých důvodů změnit. V aplikaci tak můžeme například zobrazit vždy jen jeden záznam a pro výběr zobrazovaných záznamů zvolit jiný mechanismus – třeba samostatný formulář pro hledání nebo ovládací prvek typu kombinované pole, z něhož si uživatel vybere konkrétní záznam. Příklad druhé z popsaných technik uvádí obrázek 12-3, který jsem převzala z ukázkové databáze Access Developer Solutions.

Obrázek 12-3 Kombinované pole může být někdy užitečným mechanismem, pomocí kterého si uživatel vybere požadovaný záznam, je třeba jej ale používat konzistentně

Změna implicitního navigačního mechanismu je v pořádku – ve skutečnosti jsou k tomu opravdu často dobré důvody – jestliže si ale vybereme jeden určitý mechanismus, měli bychom jej používat ve všech formulářích našeho systému. Je zcela nesprávné a pro uživatele matoucí ponechat třeba na formuláři Zákazníci standardní navigační tlačítka, zatímco ve formuláři Objednávka prodeje vytvoříme tlačítko Hledat objednávku a formulář Výrobky zobrazíme v podobě tabulkového listu.

Výjimkou z tohoto pravidla je systém, který obsahuje několik kategorií různých formulářů. Takto se můžeme například rozhodnout ošetřit formuláře pro údržbu vyhledávacích tabulek jiným způsobem, než běžné tabulky pro pořizování údajů v primárním entitách. V takovémto případě uživateli žádným nežádoucím způsobem nezatěžujeme – pochopitelně pokud je uživatelské rozhraní v rámci každé z kategorií konzistentní.

Druhá oblast, v níž je třeba zachovat konzistenci, je reprezentace složitých entit. Konzistentním způsobem je tak nutné uživateli prezentovat entity, které se modelují jako více než jedna tabulka ve vztahu jedna k více (klasickým příkladem je entita Objednávka prodeje). Je poměrně hluoupé

vypsat řádky rozpisu objednávky prodeje do tabulkového listu, zatímco z více kontaktních osob stejného zákazníka vytvoříme ovládací prvek seznamu.

V této oblasti se ale konzistence naneštěstí udržuje mnohem obtížněji, zejména pokud se týká více než dvou množin záznamů. Jediný formulář, na kterém je jeden tabulkový list s kontakty zákazníků, ve druhém jsou adresy a třetí obsahuje uspořádané výrobky, asi moc pěkný nebude. V takových situacích je třeba zapojit vlastní nápaditost a tvůrčí schopnosti. Z jednotlivých tabulkových listů tak můžeme například vytvořit samostatné záložky, nebo je umístit na samostatné vyskakovací formuláře. Můžete se také rozhodnout pro *dve* metody zobrazování a ty pak důsledně dodržovat.

Poslední oblastí, kde je nutné si dávat pozor na konzistenci, jsou mechanismy pro vytváření a editaci záznamů. Ty musí být absolutně konzistentní, a to v celém systému. Jestliže jeden formulář umožňuje editaci přímo na místě, zatímco druhý formulář vyžaduje od uživatele explicitní přechod do editačního módu klepnutím na tlačítko Uprav, případně volbou z nabídky, bude systém uživateli opravdu mást. Zavádění nějakých explicitně přepínaných editačních módů samozřejmě vůbec není dobré, jak jsme si již ukázali.

Někdy máme důvod určité záznamy po uložení uzamknout – to znamená, že uživateli neumožníme přímou editaci na místě. Změnit objednávku umožníme například uživateli jen před jejím odesláním; po provedení operace odeslání se již z objednávky stává historický záznam, který nemá smysl měnit. Někteří vývojáři implementují v takovémto případě mechanismus, který v okamžiku, kdy uživatel chce záznam editovat, kontroluje, jestli již byla objednávka odeslána. To je přijatelné, pokud se ovšem stejným způsobem chovají *všechny* formuláře.

Vhodnější řešení je proto editaci na místě normálně umožnit, ale při zobrazení již odeslané objednávky všechna pole na formuláři uzamknout a zabránit tak úpravám. Implementace takového mechanismu je o něco málo složitější; tuto práci však nakonec vykonává systém, nikoli uživatelé, a to se nepočítá. A navíc takto můžeme implementovat editaci na místě v celém zbytku systému, kde se uzamykání historických záznamů neplatí.

### **Stručné shrnutí**

V této kapitole jsme se zabývali některými základními principy návrhu uživatelských rozhraní. Začali jsme popisem tří modelů rozhraní systému: mentální model uživatele (co si uživatel *myslí*, že se v systému děje), implementační model systému (co se *skutečně děje*) a symbolický model (co systém uživateli *ukazuje* o své činnosti).

Dále jsme se věnovali třem kategoriím uživatelů: začátečníci, pokročilí a experti. Stručně jsme zde rozbrali speciální požadavky každé z těchto kategorií a řekli jsme si, jak těmto požadavkům nejlépe vyhovět v uživatelském rozhraní systému. Nakonec jsme si řekli tři nejdůležitější pravidla pro návrh uživatelského rozhraní: zapojit do systému uživatele, minimalizovat zatížení paměti a být konzistentní.

V následující kapitole přejdeme k celkové architektuře systému a konečně se podíváme na některé „hmatačelné“ součásti návrhu uživatelského rozhraní.

Při návrhu uživatelského rozhraní nového systému je třeba jako první provést rozhodnutí o celkové struktuře rozhraní – to znamená, zvolit vhodnou *architekturu uživatelského rozhraní*. V této kapitole budeme hovořit o několika víceméně standardních architekturách, které jsou popsány v knize *The Windows Interface Guidelines for Software Design* a implementovány v populárních softwarových produktech.

Můžete si také navrhnout svoji vlastní architekturu uživatelského rozhraní; protože se ale opět jedná o únik od stávajících standardů, měli byste k tomu mít opravdu pádný důvod. Nezapomeňte, že konzistence, a to nejen v rámci jedné aplikace, ale i mezi různými aplikacemi, usnadňuje uživatelům život.

## Podpora pracovních procesů

Nejdůležitější princip při rozhodování o struktuře budoucího rozhraní můžeme vyjádřit zhruba takto: konkrétní zvolená architektura musí vycházet z pracovních procesů, které má systém podporovat, nikoli ze struktury dat. Sledujte, jaké úkoly se uživatelé snaží plnit, a strukturu systému přizpůsobte s ohledem na podporu těchto aktivit.

Tato chyba se udělá opravdu snadno: podíváte se do diagramu entit a vztahů (E/R diagramu) modelovaného systému, najdete v něm třeba entitu „Zákazníci“, takže vytvoříte formulář Zákazníci pro vytváření a editaci záznamů o zákaznících. Poté navrhnete formulář Objednávky, který se na tabulku Zákazníci odkazuje v režimu pouze pro čtení. Protože se snažíte vytvořit uživatelsky přijemný systém, umožněte uživateli vybrat jméno zákazníka ze seznamu; zbytek formuláře jednoduše zapолните poli s informacemi z tabulky Zákazníci. Výsledný formulář může vypadat třeba jako na obrázku 13-1, který pochází z ukázkové databáze Northwind.

Výrobek:	Cena:	Množství:	Sleva:	Výsledná cena:
► Original Frankfurter grüne Soße	325,00 Kč	2	20%	520,00 Kč
Raclette Courdavault	1 375,00 Kč	15	0%	20 625,00 Kč

Zobrazit výrobky měsíce | Tisk faktury | Mezisoučet: 21 145,00 Kč  
Dopravné: 1 738,25 Kč  
Celkem: 22 883,25 Kč

Záznam: 14 | 1 | 1 | ► | ▶ | ▶\* | z 830

Obrázek 13-1 Formulář Objednávky z ukázkové databáze Northwind

Tento formulář není na první pohled tak úplně špatný. Zkuste se ale zamyslet nad úkolem, který stojí před uživatelem: tento uživatel zadává obchodníku prodeje. Otevře formulář Objednávky a v něm jenom zjistí, že je zákazník pro systém zatím neznámý. Musí tedy formulář opustit, přejít v systému někam jinam, zadat údaje o novém zákazníkovi, a pak se *konečně* vrátit k pořízení vlastní objednávky. Jestliže přitom uživatel původní formulář Objednávky neuzavřel, nesmí zapomenout po návratu stisknout třeba klávesovou kombinaci Shift+F9 (nebo něco jiného, co je zhruba stejně „jasné“) a obnovit seznam zákazníků. Jak bolestné.

Někteří vývojáři řeší tento problém pomocí události NotInList kombinovaného pole, ve které může uživatel zadat nové jméno. Při vyvolání této události se zobrazí zpráva s dotazem, jestli se má přidat nový zákazník, a pokud uživatel přidání nového záznamu potvrdí, otevře se formulář Zákazníci.

Toto řešení ušetří uživateli alespoň několik málo stisků kláves, stále však uživatele nutí k tomu, aby přerušil rozdělanou práci (tedy zadávání objednávky) a začal dělat něco jiného (údržba seznamu zákazníků). Mnohem lepší by bylo, kdybychom uživateli umožnili dokončit aktuální úkol bez přerušení. Jestliže uživatel zadá takové jméno zákazníka, které se v nabízeném seznamu ne nachází, zůstanou pole, jež by se ve formuláři za normálních okolností vyplnila údaji o zákazníkovi, prázdná. (Již z toho je mimochodem naprostě jasné, že se jedná o nového zákazníka; nemá tedy smysl uživatele strašit nějakým pípáním a zobrazováním nějakých zbytečných dialogů.) Jakmile uživatel tato pole vyplní, může systém potichu přidat nového zákazníka někde na pozadí.

Jestliže se všechna pole do záznamu nového zákazníka dají zjistit z formuláře Objednávky, jsme v tomto okamžiku hotovi. Pokud obsahuje tabulka zákazníků ještě nějaká další pole (jak tomu ostatně velice často je), můžeme se rozhodnout položit po dokončení objednávky prodeje uživateli dotaz, jestli hodlá tyto informace ihned doplnit. (Nezapomeňte ale počkat, až uživatel skončí práci na objednávce – vyrušovat je přece neslušné.) Toto rozhodnutí bude vycházet z okolností, za kterých se objednávka pořizuje – jinými slovy, z příslušných pracovních procesů.

Pokud objednávku zadává obchodní zástupce, který s novým zákazníkem přímo osobně jedná, je okamžik po převzetí objednávky skutečně tím správným místem pro pořízení uvedených informací. „Protože jste náš nový zákazník, poprosím vás nyní ještě o pár doplňujících informací..“ Podobně pokud se doplňující informace o zákazníkovi (nebo alespoň některé z nich) nacházejí na formuláři Objednávky, který slouží k zadávání dat do systému, pak je uživatel může snadno zadat ještě v okamžiku, dokud zákazník stojí před ním, a ušetříme mu tak nervy při pozdějším hrabání se v papírech.

Jestliže má ale uživatel před sebou svazek objednávek, které musí do systému pořídit, a doplňující informace o zákazníkovi *nemá* při tomto pořizování k dispozici, pak by takový dialog v 99 procentech případů jednoduše zrušil a žádné informace by v něm nezadal. Žádat od uživatele neustále zadání informací, které nemá (a třeba ani nemůže mít), je k ničemu – dokonce jej to obtěžuje. Lepší by tedy bylo označit záznamy s neúplnými údaji nějakým příznakem, aby se k nim uživatel mohl vrátit později, kdy již potřebné informace má k dispozici.

Při ukončení formuláře Objednávky se již můžeme uživateli zeptat, jestli chce přejít přímo na údržbu Zákazníků (nebo jak se tento proces v systému jmenuje). Má to ovšem smysl *jen* tehdy, pokud se dá předpokládat, že tyto informace bude vyhledávat a zadávat stejný člověk (jednotlivec), a to v dalším kroku po zadání objednávky.

## Architektury dokumentů

Architektury uživatelského rozhraní můžeme rozdělit do dvou základních skupin, a sice podle toho, jestli se v aplikaci zobrazuje jen jedno okno – takovým aplikacím se říká rozhraní jednoho dokumentu (Single Document Interface, SDI) – nebo jestli aplikace obsahuje základní okno, v rámci něhož se dají otevírat další okna – pak hovoříme o rozhraní více dokumentů (Multiple Document Interface, MDI).

Žádný z těchto dvou stylů rozhraní není sám o sobě lepší nebo uživatelsky přátelštější než druhý. Obě mají svoje výhody a nevýhody, které si v této části textu stručně naznačíme. Konkrétní zvolená architektura uživatelského rozhraní musí opět vycházet z pracovních procesů, které navrhovaný systém bude podporovat.

### Rozhraní jednoho dokumentu

Rozhraní jednoho dokumentu (Single Document Interface, SDI) – jak zajisté tušíte – nabízí uživateli pouze jedno jediné hlavní okno. Doplňující informace může zobrazovat v dalších dialozích. Model typu SDI se hodí pro systémy, které mají pracovat vždy pouze s jednou logickou entitou (tu však v databázi může reprezentovat libovolný počet fyzických tabulek). Pomocí rozhraní typu SDI můžeme například navrhnout jednoduchý systém pro údržbu informací o zaměstnancích.

Architektura SDI má řadu výhod, protože s jediným oknem může uživatel opravdu snadno manipulovat a snadno jej také sleduje. Vyhovuje metodě návrhu uživatelského rozhraní orientovaného na dokumenty, které doporučují „největší žijící“ odborníci na uživatelské rozhraní v samotném Microsoftu.

Systémy SDI se nejsnáze vytvářejí ve Visual Basicu. V Microsoft Accessu se přímo implementovat nedají, protože tam jsou všechny formuláře obsaženy přímo v okně aplikace Access. Dojem roz-

hraní typu SDI však můžeme i v Microsoft Accessu dosáhnout maximalizací hlavního formuláře systému hned po spuštění aplikace; dále musíme odstranit tlačítko pro minimalizaci. V Accessu 2000 můžeme navíc stanovit, že se okno formuláře má zobrazit v hlavním panelu systému Windows. Při troše pečlivosti může být tedy i systém napsaný v Microsoft Accessu pro všechny naše účely aplikací typu SDI.

### **Aplikace typu sešit**

Architektura rozhraní typu sešit (sešitové rozhraní) je speciálním typem rozhraní jednoho dokumentu SDI. V sešitu se různé pohledy na data nezobrazují v samostatných oknech, nýbrž na záložkách jediného okna. Nejlepším příkladem sešitové aplikace je zřejmě Microsoft Excel.

Výhodou této architektury rozhraní je pro uživatele bezpečný kontext, v němž ale není omezen jen na jeden formulář. Implementace tohoto rozhraní s přijatelnou dobou odezvy však může být poněkud obtížná. Nezbytným předpokladem je snažit se o odpovídající výkon aplikace během celé její implementace. V takovém případě je sešit velice užitečný jednak pro zobrazování různých pohledů na jeden objekt, jednak pro prezentaci pohledů na úzce související množinu objektů, které uživatel nepotřebuje mezi sebou vzájemně porovnávat.

V sešitu můžeme například na jedné záložce zobrazit souhrnnou sestavu měsíční úrovně prodeje, na druhé záložce koláčový diagram s údaji o prodeji podle kategorie a na třetí záložce třeba sloupcový diagram ročních objemů prodeje. Všechny tyto informace zde skutečně souvisí, přesto však můžeme oprávněně očekávat, že si uživatelé tyto sestavy nebudou prohlížet společně jako skupinu. Jednotlivé sestavy spolu tedy souvisí, vzájemně však nejsou srovnatelné, takže si je uživatel opravdu nejspíše nebude zobrazovat všechny současně.

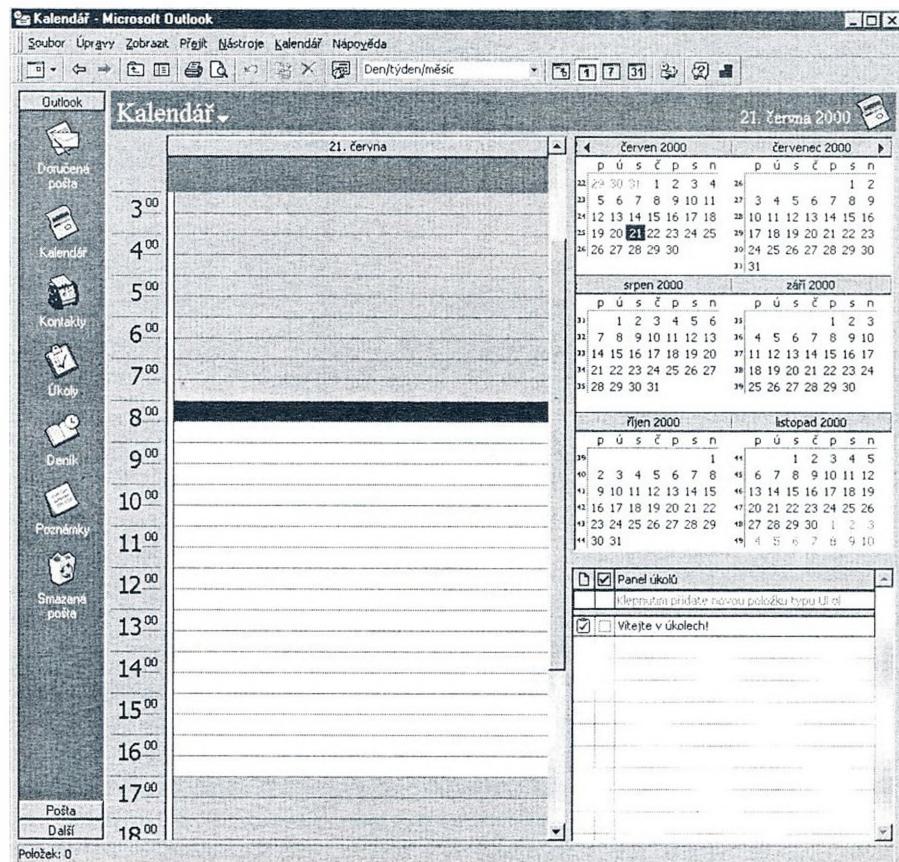
Záložky v sešitu mají jisté implicitní pořadí, které nám může být užitečné. Řekněme, že například v systému podporujeme určitý pracovní proces složený z řady diskrétních úkolů. Tyto úkoly se přitom velice často (i když ne nutně vždy) vykonávají v jistém konkrétním pořadí. Pro takové úkoly můžeme vhodně využít záložky sešitu. Pořadí záložek přesně odpovídá pořadí provádění úkolů a uživatel se tak nemusí přizpůsobovat ničemu novému.

Sešit se ale na druhé straně obvykle nehodí pro oddělení pracovních procesů – které představují zpravidla jasné odlišené aktivity – nebo pro prezentaci informací, které se musí přímo porovnávat. Nezapomeňte, že ze sešitu je v každém daném okamžiku vidět vždy pouze jedna karta, takže pro uživatele by takové řešení znamenalo nezádoucí nároky na jejich krátkodobou paměť.

### **Rozhraní ve stylu Microsoft Outlook**

Dalším speciálním typem architektury uživatelského rozhraní je něco, čemu zde říkám „rozhraní ve stylu Microsoft Outlook“; aplikace Outlook byla totiž první, kde jsem tento typ rozhraní viděla. V tomto stylu rozhraní je okno aplikace rozděleno do dvou podoken, z nichž jedno obsahuje sadu ikon a druhé obsahuje dokumenty. Příklad vidíme na obrázku 13-2.

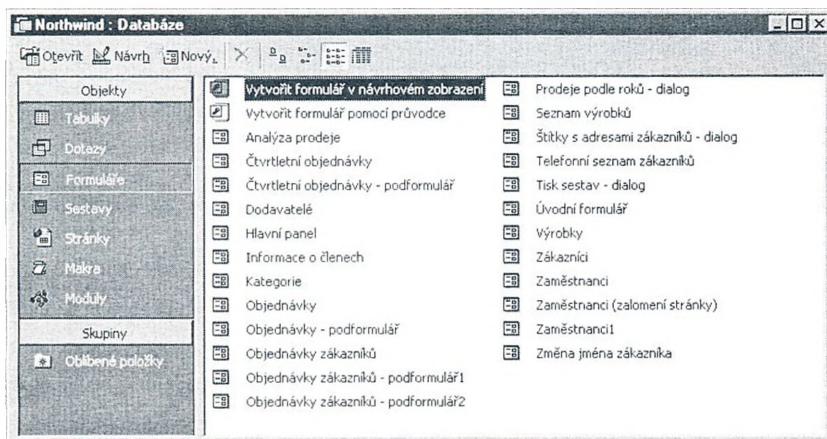
Osobně mám rozhraní tohoto stylu v oblibě zejména pro aplikace, které podporují více různých pracovních procesů. Podokno s ikonami v levé straně okna nabízí uživateli bezprostřední kontext; navíc může být rozděleno na panely, ve kterých se ikony dají seskupit do jednotlivých funkčních oblastí.



Obrázek 13-2 V rozhraní stylu Microsoft Outlook je okno aplikace rozděleno do dvou podoken

Ani Microsoft Access, ani Microsoft Visual Basic však naneštěstí nemají žádné vestavěné ovládací prvky, pomocí kterých by se rozhraní uvedeného stylu dalo implementovat, přestože je zná například okno Databáze programu Microsoft Access 2000 (viz obrázek 13-3). Ovládací prvky ActiveX ve stylu Microsoft Outlook, které do obou prostředí integrovat můžeme, jsou však k dispozici od různých dodavatelů třetích stran.

Jestliže se v aplikaci pro toto rozhraní skutečně rozhodnete, je vhodné umožnit uživateli skrytí podokna ikon, jak to umožňuje i Microsoft Outlook. Panel s ikonami je velice dobrým navigačním nástrojem; jakmile však uživatel otevře dokument, s nímž hodlá pracovat, zůstane po určitou dobu u něj a panel najednou zbytečně zabírá drahocennou plochu na obrazovce.



Obrázek 13-3 Okno Databáze programu Microsoft Access 2000 má rozhraní stylu Microsoft Outlook

#### **Bozhraní více dokumentů**

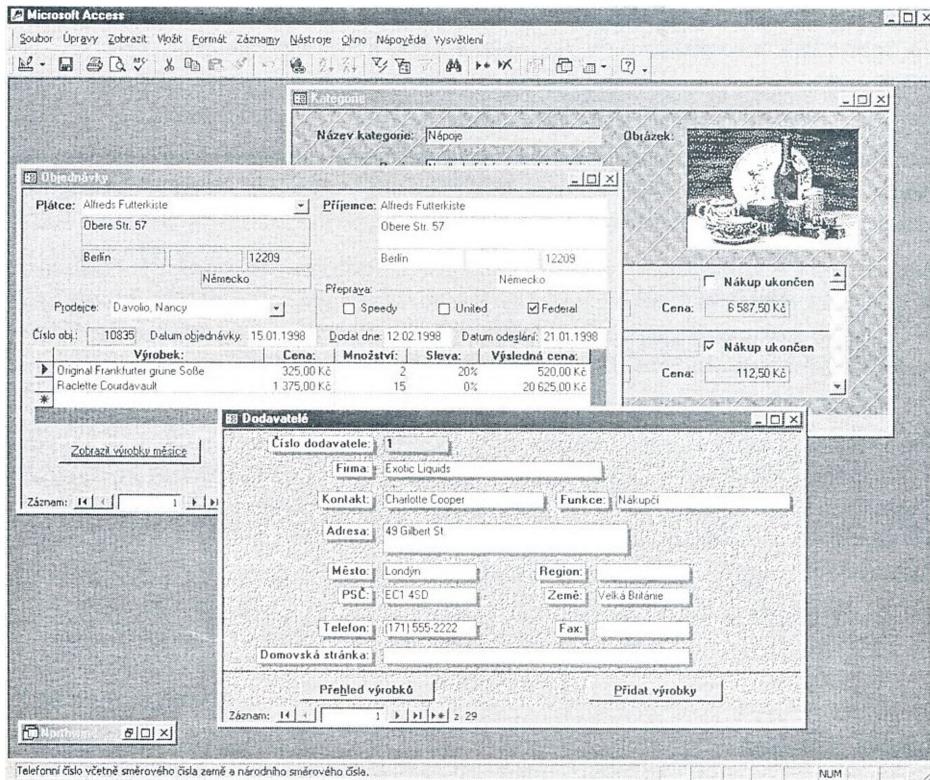
Druhá většina databázových systémů využívá nějakou verzi rozhraní více dokumentů, MDI. U rozhraní typu MDI se přitom v rámci jednoho primárního okna dá navíc otevřít několik synovských oken (potomků).

Každé jednotlivé synovské okno může obsahovat jiný typ informací; okno tak může mít jako své potomky například formulář se zákazníky a formulář pro zobrazení objednávek. Synovská okna mohou zobrazovat také různé pohledy na stejné informace, jako je například běžný formulář informací o zákaznících a sestava s objemy prodeje pro daného zákazníka. Druhá možnost je zobrazení několika instancí stejného typu informací; jeden formulář zákazníka může například ukazovat informace o panu Kovářovi a ve druhé instance stejného formuláře budou obsaženy informace o zákazníkovi Bednářovi.

Podobně jako u rozhraní SDI existuje také několik různých typů struktury aplikací MDI, z nichž každá vyhovuje jiným požadavkům. My zde budeme hovořit jen o těch nejdůležitějších strukturách, uvedené konfigurace však zdaleka nejsou vyčerpávající – a navíc se ani nemusí nutně vzájemně vylučovat.

### **„Klasická“ architektura MDI**

Jak tedy vypadá klasická struktura aplikace typu MDI? Primární nebo hlavní okno aplikace obsahuje několik synovských oken stejného typu nebo různých typů. Dobrým příkladem takové aplikace je Microsoft Word – okno aplikace Word může obsahovat najednou několik otevřených dokumentů. Počet synovských oken je (alespoň podle toho, co sama znám) omezen pouze velikostí dostupné paměti.



Obrázek 13-4 Kontejnerový model v aplikacích typu MDI může často zmást uživatele

Ve Wordu je možné pomocí spouštěcího přepínače /n, zadávaného na příkazovém řádku, otevřít automaticky se spuštěným aplikace okno nového dokumentu. Podobnou funkci můžete implementovat i ve své vlastní aplikaci s touto architekturou uživatelského rozhraní. To znamená, že můžete nadefinovat například přepínač, který po spuštění aplikace otevře formulář Objednávky, připravený k pořizování dat.

Nezapomeňte ale v takovém případě doplnit do aplikace mechanismus, kterým se uvedená funkce zase vypne; jinak by funkce zbytečně dráždila uživatele, pro které fakticky nemá smysl. Zároveň musíte být při implementaci opatrní, abyste nesmyslně nezobrazovali chybové zprávy takovému uživateli, který okno okamžitě uzavře, nebo dokonce – co je ještě horší – přidá do databáze prázdné záznamy.

Největší problém rozhraní typu MDI je určitá nekonzistence kontejnerového modelu. Rodičovské okno obsahuje totiž synovská okna, která jsou v něm otevřena, avšak samotná aplikace nemusí nutně obsahovat objekty, jež tato okna reprezentují. Tento problém je zpravidla horší u aplikací jako je Microsoft Word, kde jsou dokumenty skutečně samostatnými objekty souborového systému. V databázových aplikacích je uživatel od těchto složitých aspektů souborového systému obvykle mnohem lépe izolován. Opravdu efektivně to ale někdy i malé, jednoduché databázové aplikace nedokáží.

Podívejme se například na obrázek 13-4. Dále předpokládejme, že se uživatel mezi jednotlivými okny přepíná sem a tam a že v každém z oken visí nějaká nepotvrzená změna.

Jestliže nyní uživatel vybere příkaz Uložit z nabídky Soubor, co se vlastně stane? Potvrdí se pouze změny v okně Dodavatelé. My to víme, protože si dokážeme uvědomit, že příkazy nabídky pracují vždy pouze nad aktuálním oknem; ví to ale i náš uživatel? Není rozumné naopak očekávat, že pokud požádáme aplikaci „Northwind“ o uložení, uloží se v ní jednoduše veškeré změny bez ohledu na to, kde jsme je skutečně provedli?

Knihovna *The Windows Interface Guidelines for Software Design* říká, že do nabídky Soubor je možné doplnit volbu Uložit vše, která uloží veškeré nepotvrzené změny ve všech otevřených oknech. To je nepochybně jisté použitelné řešení, přinejlepším je to ale jen kompromis. Uživatel si stále musí uvědomit, že mezi aplikací MDI a objekty, nad nimiž tato aplikace pracuje, je podstatný rozdíl.

Tento rozdíl je ale součástí implementačního modelu, nikoli mentálního modelu uživatele; zároveň je to pro uživatele na pochopení jedna z nejobtížnějších myšlenek. I poměrně dobře znalí uživatelé často přesně nechápou, co se kde ukládá. I já sama jsem často zmatena z Microsoft Wordu a neuvedomuj si přesně, jaké formátování je uloženo spolu s dokumentem a jaké se nachází v šabloně, a to s tímto produktem dosti intenzivně pracuji po řadu let.

Přesto všechno mají ale i klasické aplikace MDI své místo na slunci. Pro většinu aplikací, ve kterých je třeba mít možnost otevření několika oken současně, jsou stále tím nejlepším dostupným řešením.

### Rozhraní s přepínacím panelem

Aplikace s přepínacím panelem (hlavním panelem) zobrazí při svém spuštění jakýsi centrální formulář, jaký vidíme například na obrázku 13-5. Většina tlačítek na takovémto formuláři je svázána s určitým formulářem nebo sestavou.



Obrázek 13-5 Při spuštění aplikace se zobrazí přepínací panel

Strukturu přepínacího panelu mají mimo jiné i databáze vygenerované pomocí průvodce Nová databáze v Microsoft Accessu; uvedená struktura se proto u databázových aplikací, vyvíjených v tomto prostředí, stala běžným typem rozhraní. I ve Visual Basicu by byla implementace podobného chování docele snadná.

Musím přiznat, že jsem jistým způsobem proti přepínacím panelům zaujatá. Něco mi na nich připomíná staříčký DOS a připadají mi tak hloupá. Moc se mi také nelibí, že podporují onu nechvalně známou strukturu nabídeku „Hledej záznam/Edituj záznam/Tiskni záznam“, která je pro uživatele až neuvěřitelně krkolemná.

Rozhraní s přepínacím panelem může být ale vhodné v případě, že má jedna stejná aplikace podporovat několik různých pracovních procesů, přičemž se nechceme pouštět do složité implementace rozhraní ve stylu Microsoft Outlook, nebo pokud musíme být schopni v této aplikaci otevřít několik oken současně. Elegantním mechanismem pro navedení uživatele přes celou aplikaci je právě přepínací panel na nejvyšší úrovni, který pro každý jednotlivý pracovní proces obsahuje jedno tlačítko.

Při návrhu aplikací s přepínacím panelem je ale důležité si zapamatovat jednu zásadu: podobně jako u každé jiné architektury rozhraní i zde musíme strukturovat přepínací panel podle pracovních procesů, nikoli podle dat. Svoje tlačítko by tak v přepínacím panelu měla mít každá činnost či každá operace, kterou bude uživatel potřebovat vykonávat, *nikoli* každý formulář a každá sestava v systému.

## Rozhraní projektu

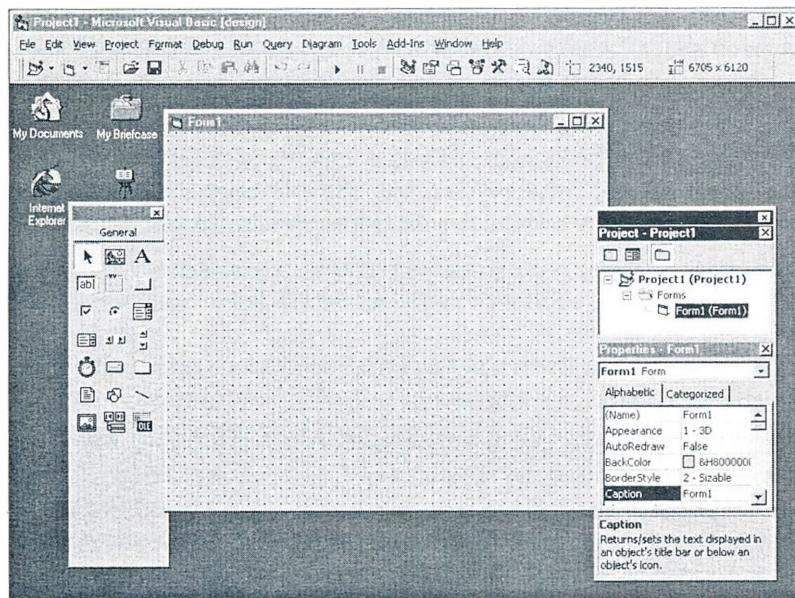
Rozhraní typu projektu považuji jistým způsobem za aplikaci přepínacího panelu, která neobsahuje žádné okno. Okno „Projekt“ (které se samozřejmě nemusí přesně takto jmenovat) nabízí namísto toho mechanismus pro nezávislé otevírání oken na pracovní ploše. Dobrým příkladem projektového rozhraní je rozhraní typu SDI ve Visual Basicu, které vidíme na obrázku 13-6.

Tato nezávislá okna se po svém otevření zobrazí na hlavním panelu systému a uživatel s nimi může pracovat skutečně nezávisle. Současně je ale ovládá i okno Projekt. Při minimalizaci nebo uzavření okna Projekt se tak stejná operace provede i se sekundárními okny.

Rozhraní projektového typu odstraňuje kontejnerový model a vyhýbá se tím oné nekonzistence, jakou známe z aplikací MDI. Na druhé straně zde ale vzniká jiná nekonzistence: je to onen zvláštní vztah mezi oknem Projekt a (zdlánlivě) nezávislými sekundárními okny.

Uvažujme následující scénář: uživatel si poklepáním na nějakou ikonu v okně Projekt otevře sekundární okno a poté se snaží uvolnit prostor na pracovní ploše a minimalizuje okno Projekt. Oujej, nové okno, které uživatel právě otevřel, zmizí také. Ano, dá se kdykoli obnovit z hlavního panelu – jak krásně ale mate uživatele.

Pokud se vám myšlenka s oknem projektu líbí, neberu vám je, ale osobně si myslím, že oproti klasickému oknu projektu je vhodnější model, jaký ukazuje okno Databáze v programu Microsoft Access. Okno Databáze má totiž stejné výhody jako okno projektu, nemá však zároveň ony nepříjemné vedlejší účinky, protože je součástí klasické aplikace typu MDI.



Obrázek 13-6 Visual Basic v režimu SDI má rozhraní typu projektu

## Průvodci

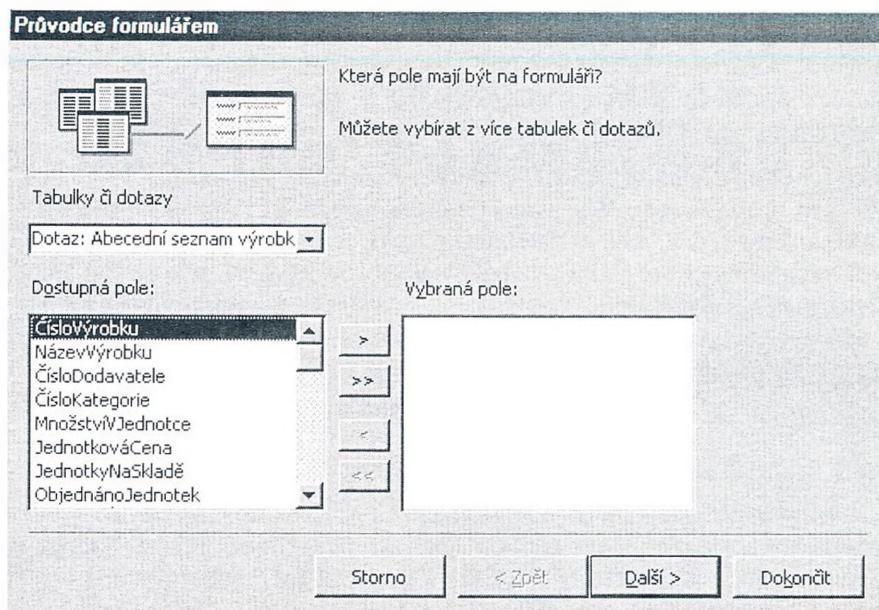
Posledním typem architektury rozhraní, které hraje svou dosti důležitou roli v databázových aplikacích, je průvodce (anglicky wizard, doslova „kouzelník“). Průvodce se skládá z posloupnosti stránek, které se postupně zobrazují v jistém dialogu. Příklad průvodce vidíme na obrázku 13-7.

Průvodce se nejčastěji používá u úkonů, které se provádějí poměrně málo často, jako je například instalace nebo konfigurace hardwaru. Podobně se ale zcela jistě hodí i pro podporu málo častých pracovních procesů v databázových aplikacích.

Pomocí průvodců můžeme také podporovat složité pracovní procesy. Jestliže se určitý pracovní proces skládá z řady různých úkonů, které se musí (nebo alespoň mohou) provádět v jistém definovaném pořadí, může být vhodnou variantou rozhraní právě průvodce. Průvodci jsou užiteční zejména v případě, že pracovní proces obsahuje velké množství podmíněných úkolů: „Jestliže platí podmínka a, proved krok 3 a poté krok 4; platí-li podmínka b, přejdi na krok 6!“

U složitých pracovních procesů se ale průvodce dá využít jen tehdy, pokud se jednotlivé úkoly mohou provést přesně v tom pořadí, v jakém je průvodce nabízí. To není až tak časté, jak bychom mohli očekávat. Pokud se úkoly nemusí zadávat v žádném konkrétním pořadí, není průvodce nejvhodnějším řešením, nebo by alespoň neměl být *jediným* rozhraním k dokončení úkolu.

Rozhodneme-li se využít průvodce v některé naší databázové aplikaci, musíme si pečlivě promyslet, jak a kdy se data, která uživatel do průvodce zadá, budou fyzicky ukládat. Obvykle platí totiž takový model, že uživatel může kdykoli během práce průvodce stisknout tlačítko Storno a sy-



Obrázek 13-7 Aplikace Microsoft Access 2000 využívá průvodců velmi často

stém se vrátí do stejného stavu, v jakém byl průvodce spuštěn – to znamená, že se musí zahodit veškerá data, která uživatel mezitím do průvodce zadal.

V závislosti na konkrétní aplikaci může být přitom vhodné ukládat data buďto po dokončení každé jeho jednotlivé stránky, nebo dát uživateli alespoň možnost i při stisku tlačítka Storno uložit dříve zadané údaje. Netvrídíme, že by některý z těchto dvou postupů musel být nezbytně nutným pravidlem; pokud se ale v průvodci pořizuje větší množství dat, má tento postup (v některé z variant) svoje opodstatnění.

Druhá možnost je nabídnout uživateli dočasné pozastavení běhu průvodce a jeho pozdější pokračování. To je pro uživatele velice pohodlné řešení, na implementaci je ale o něco složitější, protože v něm musíme vyřešit určitý mechanismus pro dočasné uložení dat, musíme zjistit, kde uživatel průvodce zastavil, a musíme být schopni od tohoto místa později pokračovat dále.

### Stručné shrnutí

V této kapitole jsme hovořili o různých variantách struktury formulářů v systému. Na nejvyšší úrovni si můžeme zvolit mezi systémem jednoho dokumentu SDI, v němž se veškerá data zobrazují v jediném okně, a systémem více dokumentů MDI, který pracuje s několika okny.

Systémy typu SDI mají dvě podoby: jsou to pracovní sešity a rozhraní stylu Microsoft Outlook. Každé z nich je výhodné pro určité situace. Více variant existuje přitom pro aplikace typu MDI, kde známe jednak klasické aplikace MDI, dále architekturu přepínacího panelu, kterou podporují například aplikace vytvořené s pomocí Průvodce databází v Microsoft Accessu, a konečně architekturu projektu, jejímž názorným příkladem je Visual Basic. Posledním typem architektury jsou průvodci, kteří se hodí zejména pro podporu málo častých úkolů a složitých pracovních procesů.

V následující kapitole přejdeme od uspořádání množiny formulářů do aplikace ke struktuře jednotlivých formulářů a ukážeme si, jak rozhodnout o rozvržení formuláře na základě struktury entit v datovém modelu.

# REPREZENTACE ENTIT V NÁVRHU FORMULÁŘŮ

14

V této kapitole si řekneme, jak definovat strukturu jednotlivých formulářů podle způsobu, jakým jsou entity v nich zobrazované reprezentovány v datovém modelu. (V několika posledních kapitolách jsme se tak intenzivně zaměřili na pracovní procesy, že jste asi trochu nechápali, proč jsme se vůbec učili nějaké datové modelování.) Rozhodnutí o tom, jaká data patří na každý jednotlivý formulář, musí vycházet z pracovních procesů, konkrétní rozvržení formuláře a výběr ovládacích prvků na něm však již určují skutečné datové struktury, které formulář reprezentuje.

První rozhodování, jež musíme při návrhu provést, je zjistit, jakým způsobem se formulář mapuje do modelu entit a vztahů (E/R modelu). Reprezentuje tento formulář jedinou entitu, dvě entity ve vztahu jedna k jedné, dvě entity ve vztahu jedna k více, nebo více než dvě entity? Každý z těchto uvedených typů struktur se hodí pro jisté typy rozvržení formuláře, o nichž budeme v této kapitole hovořit. Podobně jako architektury rozhraní z kapitoly 13 představují ale i tato rozvržení jen jakési obecné zásady, ne nějaká přísná pravidla. Jsou to však rozvržení, která se podle mých zkušeností nejpřirozeněji hodí pro určité typy datových struktur.

## Jednoduché entity

Prvním typem entity, které se na formuláři nejsnáze reprezentuje, je jednoduchá (prostá) entita, jež představuje jedinou tabulku v databázi. Pokud se taková entita účastní nějakého vztahu, nachází se buďto na straně „více“, nebo se druhý účastník vztahu na formuláři neobjeví.

Podívejme se například na obrázek 14-1, který ukazuje entitu Zákazníci a její vztahy.



Obrázek 14-1 Tento E/R diagram ukazuje entity svázané s entitou Zákazníci

Entita Zákazníci se v tomto diagramu nachází u všech vztahů na straně „více“, pouze s výjimkou vztahu s entitou Objednávky. Někde v aplikaci budeme mít nejspíše formulář pro údržbu informací o zákaznících. (To přitom platí i v případě, že se data do systému nejprve pořizují jako větší produkt nějaké jiné činnosti.) Entita Objednávky do tohoto formuláře Zákazníků zřejmě patřit nebude. Objednávky jsou od zákazníků logicky oddělené, takže je uživatelé budou spravovat na nějakém jiném místě systému.

Entita Zákazníci je tedy na straně „více“ všech entit, zahrnutých na formuláři údržby zákazníků, a proto s ní můžeme pracovat jako s jednoduchou entitou. Z toho také vyplývá jednoduché rozvržení formuláře, které se přímo nabízí. Na formuláři nebudeme potřebovat ani podřízené formuláře, ani mřížky. Stačí pro každé jednotlivé pole zvolit ovládací prvek, který se pro ně nejlépe hodí (o tom si více povíme v následující kapitole), „hodit“ jej na odpovídající místo formuláře a jsme hotovi.

Samozřejmě že ovládací prvky nebudeme na formulář nějak *pohazovat*, ale že je pěkně uspořádáme, vytvoříme okraj v šířce sedmi bodů, mezi jednotlivými prvky vytvoříme mezeru čtyřmi body a nejdůležitější z nich umístíme do levého horního rohu, jak to popisuje kniha *The Windows Interface Guidelines for Software Design*, nemám pravdu? Základní myšlenku jste ale zajisté pochopili. Vytvořit takový formulář není nijak složité a nic zvláštního k tomu nepotřebujeme – tedy pokud se nám nepodaří zcela vyčerpat prostor formuláře.

Vyčerpat dostupný prostor se nám ale také může docela klidně podařit. Do formuláře nebo se stav v Microsoft Accessu můžeme během doby jeho života vložit nejvýše 754 ovládacích prvků (to je technické omezení, do něhož se počítají i ovládací prvky, které později odstraníme) a do formuláře ve Visual Basicu nejvýše 254 ovládacích prvků (přičemž pole ovládacích prvků se zde počítají za jediný ovládací prvek).

Prakticky vzato ale není dobré dávat na jeden formulář více než 25 až 30 ovládacích prvků nebo skupin ovládacích prvků, s nimiž musí uživatel pracovat najednou. (Všimněte si, že zde říkám „ovládacích prvků nebo skupin ovládacích prvků“; skupinové pole se třemi nebo čtyřmi tlačítky voleb přepínáče zde tedy považujeme za jediný logický ovládací prvek.) Co tedy uděláme, když nějaká jednoduchá entita obsahuje třeba 75 atributů? V takovém případě musíme zobrazit vždy pouze část údajů.

Nejsnazší způsob usporádání formuláře, který reprezentuje entitu s příliš mnoha atributy, než kolik je možné pohodlně zobrazit najednou, spočívá v rozdělení polí do určitých skupin nebo kategorií. Já obvykle začínám výběrem takových atributů, které jasně identifikují entitu. Mezi tyto atributy řadím přitom nejen atributy kandidátního klíče, ale také další popisné atributy, podle nichž uživatel bezpečně pozná, že pracuje s tou správnou konkrétní instancí entity.

U entity Zákazníci bude například do této skupiny identifikačních atributů spadat nejspíše jméno a adresa (včetně veškerých podrobných údajů) a možná i atribut Obchodní zástupce. V entitě Výrobky to budou například atributy Kategorie výrobku, Název a Popis. Tato skupina atributů by tedy měla být umístěna na nepřehlédnutelném místě v horní části formuláře a měla by být vždy viditelná.

Zbývající atributy pak rozdělíme do několika skupin, v nichž se nacházejí vždy logicky spojené atributy, které je třeba prohlížet společně. U entity Zákazníci můžeme například takto do jedné

skupiny přiřadit atributy, které vyjadřují údaje pro prodej – jako je standardní výše slevy a doba splatnosti faktury – a do druhé skupiny informace o kontaktech – nákupčího, obchodního zástupce a podobně. V entitě Výrobky můžeme podobně do jedné skupiny zahrnout technické specifikace výrobcu a do druhé skupiny informace související s balením zboží.

Při uspořádání atributů do skupin máme v zásadě několik možností. Na hlavním formuláři tak můžeme vytvořit ovládací prvek se záložkami, kde každou skupinu atributů umístíme na samostatnou záložku. Toto řešení používám třeba já osobně nejčastěji. Ovládací prvky typu záložka nabízí totiž uživateli nejlepší kontext – uživatel hned na první pohled ví, že má k dispozici ještě nějaké další informace a o jaké informace se vlastně jedná. Pokud je ale skupin více než pět nebo šest, je již ovládací prvek se záložkami pro uživatele nezvládnutelný.

V tomto případě je vhodné vyčlenit některé nebo i všechny skupiny do vedlejších, podřízených formulářů. Podřízený formulář má smysl také v situaci, kdy jednotlivé skupiny obsahují více atributů, než kolik se dá rozumně umístit na jedinou záložku. Uživatel může tyto vedlejší formuláře otevírat pomocí příkazových tlačítek na hlavním formuláři, který má tím pádem strukturu podobnou přepínacímu panelu. Jakmile má ale formulář příliš mnoho příkazových tlačítek, není již opět rozumně zvládnutelný. V takovém případě je vhodnější otevírat podřízené formuláře pomocí příkazů z nabídky.

Nezapomeňte ale i na podřízeném formuláři poskytnout uživateli potřebný kontext. Uživatel musí vždy bezpečně vědět, ke které instanci dané entity zobrazené informace náleží. Nejsnáze přitom entitu identifikujeme tak, že v podřízeném formuláři zopakujeme určité údaje z hlavní obrazovky. Obvykle zde ovšem nemusíme opakovat celou identifikační skupinu atributů – z této skupiny zde stačí uvést takové údaje, které podřízený formulář jednoznačně svazují s hlavním formulářem.

Druhá možnost je vytvořit podřízené formuláře jako modální. To je jedna z mála situací, v nichž se dají v systému ospravedlnit modální formuláře, protože tak můžeme uživateli snáze udržet potřebný kontext. Modální formuláře však na druhé straně uživatele vždy jistým způsobem omezují; tomuto omezení je vhodné se pokud možno vyhnout. Osobně používám toto řešení jen v nejvyšší nouzi, kdy už skutečně na formuláři nemám místo ani na těch několik málo atributů, které definují hlavní kontext.

Namísto několika vedlejších oken bychom teoreticky mohli sekundární, doplňující informace zobrazit v několika podformulářích hlavního formuláře (ve Visual Basicu by to byly rámečky nebo ovládací prvky typu Frame). Tento přístup se mi ale příliš nelíbí, protože se při něm opět obtížně udržuje kontext uživatele. Musíme totiž vytvořit určitý mechanismus, pomocí něhož zjistíme, který podformulář je zobrazen. Jestliže budeme zobrazení ovládat pomocí nabídky, nemáme již možnost na formuláři vyznačit, že jsou k dispozici také další informace. Pokud na formuláři vytvoříme určitý ovládací prvek, například množinu „rádiových“ tlačítek přepínače, nemusí být zase uživateli hned na první pohled jasné, proč se zobrazení formuláře neustále mění. Osobně považuji za vhodnější řešení použít ovládací prvek se záložkami – je to přece jen jakýsi zavedený, standardní mechanismus a je to rozhodně lepší než vymýšlet nějaké svoje vlastní řešení.

## Relace typu jedna k jedné

K formulářům, které reprezentují takové entity, mezi nimiž je vztah typu jedna k jedné, můžeme ve většině případů přistupovat úplně stejně jako k formulářům, které reprezentují jedinou, jednoduchou entitu. Snadno vytvoříme dotaz, v němž spojíme odpovídající pole z obou tabulek, a s jeho výsledkem pak pracujeme jako s jedinou jednoduchou entitou. Jestliže má tento výsledek více atributů, než kolik se dá rozumně zobrazit na jediném formuláři, uplatníme stejně postupy jako u jednoduché entity.

Pokud se primární entita účastní více různých vztahů typu jedna k jedné, bývá nejpřirozenější rozvržení formuláře určené již povahou těchto vztahů. Jestliže mohou vztahy existovat vedle sebe současně, můžeme výsledek považovat za jednu obrovskou výslednou množinu a zobrazit je na jednom formuláři nebo na několika formulářích, opět tedy stejným způsobem jako u jednoduchých entit.

Často se ale dotčené vztahy vzájemně vylučují; jeden Výrobek může být tak například Nápoj, nebo Sýr, rozhodně však nebude obojím. V takovémto případě je vhodné nadefinovat několik podformulářů nebo rámečků – pochopitelně pokud na to máme ve formuláři místo. Zde se nemusíte bát, že by si uživatel musel nějak zvlášť uvědomovat, jestli jsou v daném kontextu k dispozici další informace; žádné další informace zde totiž nejsou. Ovládací prvek se záložkami by byl ve skutečnosti v této situaci přímo zavádějící.

Pokud na formuláři není dostatek místa, musíme opět ovládací prvky vhodně seskupit a použít některý z postupů, které jsme probírali u jednoduchých entit. Jestliže ovládací prvky, které reprezentují atributy podtířid, umístíme na některou záložku, je velice dobré pokusit se pro popisek záložky najít nějaký dostatečně obecný výraz. Uživatele by zbytečně mátlo, pokud by se textový popis záložky (nebo třeba popis příkazového tlačítka) s průchodem tabulkou neustále měnil.

Jestliže má uživatel možnost procházet primární množinou záznamů, snažím se ze stejného důvodu nezobrazovat atributy podtířidy v ovládacím prvku se záložkami hned na jeho první záložce. Jednotlivé ovládací prvky každé podtířidy se totiž liší, takže zobrazení formuláře by opět při procházení záznamů nevhodně „poskakovalo“. Pokud se nám podaří všechny atributy podtířid dát jinam než na první záložku, bude zobrazení stabilní a povede tak zároveň i k urychlení běhu aplikace, protože při procházení záznamů se nemusí zbytečně znova přepočítávat parametry zobrazení.

## Relace typu jedna k více

V řadě formulářů potřebujeme zobrazit entity, které jsou zapojeny do nějakého vztahu typu jedna k více. Stanovit nejlepší rozvržení těchto formulářů je docela snadné; stačí si zapamatovat jediné pravidlo, podle něhož formulář zobrazuje vztah typu jedna k více, nikoli více k jedné. Při modelování složitějších vztahů si celou situaci často představíme snáze pomocí záznamů než instancí entit. V tomto případě musí tedy záznam na straně „jedna“ příslušného vztahu řídit zobrazení záznamů na straně „více“, a ne naopak. Jestliže přesto zkuste ovládat záznam na straně „jedna“ podle záznamu na straně „více“, brzy se do toho zamotáte (a to nejen vy sami, ale i celý systém a jeho uživatelé).

Jestliže tedy zobrazení formuláře správně určuje záznam na straně „jedna“, zbývá nám rozhodnout se, jak se budou zobrazovat záznamy na straně „více“. Zde máme v zásadě dvě možnosti: můžeme je zobrazit buďto všechny najednou, nebo postupně jeden po druhém.

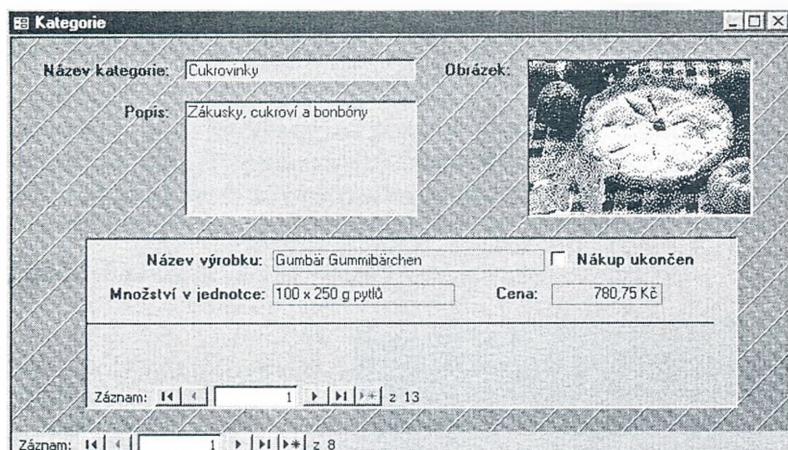
Pro konkrétní variantu se rozhodneme z velké části podle toho, kolik podrobných informací musíme uživateli ke každému záznamu na straně „více“ zobrazovat. Pokud nám stačí zobrazit jen několik málo polí, můžeme obvykle zobrazit všechny záznamy najednou. Příkladem je formulář z databáze Northwind, uvedený na obrázku 14-2; zde se čtyři pole každého záznamu na straně „více“ zobrazují v podformuláři.

Obrázek 14-2 Na tomto formuláři se záznamy ze strany „více“ vztahu typu jedna k více zobrazují na podformuláři typu nekonečný formulář

Svislý posuvník na pravé straně podformuláře indikuje, že v daném okamžiku nejsou viditelné všechny záznamy. Z pohledu návrhu přesto i toto řešení považujeme za zobrazení typu „všechny najednou“. V následující kapitole si řekneme o několika dalších způsobech zobrazování více záznamů.

Osobně mám raději zobrazení stylu „všechno najednou“, protože takový formulář nabízí uživateli největší možný kontext. Pokud máte problémy s nedostatkem místa, vždycky můžete doplňující informace zobrazit na nějakém vedlejším formuláři, který se vyvolá nějakým speciálním mechanismem. Úplně vždycky ale tento typ rozvržení formuláře není vhodný nebo použitelný, přestože třeba potřebujeme zobrazit několik záznamů ze strany „více“ najednou. Zde musíme řešit hned dva problémy: za prvé musíme dát uživateli jasné najevo, že existuje více záznamů, a dále mu poskytnout mechanismus pro pohyb mezi záznamy.

Formulář na obrázku 14-3 je stejný jako formulář z obrázku 14-2, podformulář zde ale nezobrazuje nekonečný formulář, nýbrž jen jediný záznam.



Obrázek 14-3 Na tomto formuláři se záznamy ze strany „více“ vztahu typu jedna k více zobrazují jeden po druhém

Voliče (selektory) pro výběr záznamů ve spodní části podformuláře, které mají tvar „video tlačítek“ (dalo by se říci jakoby ovládacích tlačítek videa nebo magnetofonu), jsou v Microsoft Accessu implicitním mechanismem pro navigaci mezi záznamy. Obecně doporučuji držet se implicitních možností, kde to je jen trochu možné; podívejme se ale na výsledek: na tomto formuláři jsou nad sebou poskládány hned dva voliče záznamů, přičemž rozlišit, který z nich patří do podformuláře a který ovládá primární množinu záznamu, můžeme jen podle jejich přesné pozice na formuláři. To rozhodně neznamená žádný konec světa, optimální řešení to ale zdaleka také není.

Někdy můžeme z primárního (hlavního) formuláře voliče záznamů odstranit. U formuláře, jako je na obrázku 14-3, bychom takto mohli postupovat za podmínky, že uživateli nabídnete dobré rozhraní pro hledání kategorií výrobků. Uživatel většinou tak jako tak nebudou podobným formulářem pro údržbu procházet po stránkách, takže dobré navržená funkce pro vyhledávání jim bude zcela určitě plně postačovat.

Druhá možnost je nahradit jednu ze dvou sad voličů záznamu jiným mechanismem pro procházení – například příkazovými tlačítka nebo uživatelským posuvníkem. Zpracovat pohyb záznamů pomocí jiné techniky může být ale někdy obtížné, takto zavádíme do rozhraní nežádoucí nekonzistence. Ani taková nekonzistence nemusí být nežádoucí, pokud si zvolíme jeden přístup k řešení a toho se dále důsledně držíme.

Na straně „jedna“ zobrazeného vztahu se tak můžeme rozhodnout pro pohyb mezi záznamy například pomocí tlačítka panelu nástrojů Vpřed a Zpět, jako třeba v Microsoft Internet Exploreru, zatímco v záznamech na straně „více“ se budeme přesouvat pomocí voličů záznamu typu video tlačítka. Takový postup můžeme opět použít jen tehdy, pokud se ho budeme důsledně držet v celé aplikaci. I formuláře, které reprezentují jednoduché entity a žádné záznamy na straně „více“ tak nemají, by tudíž měly používat tlačítka Vpřed a Zpět.

Někdy budeme na formuláři zobrazovat primární tabulku, se kterou je svázání několika vztahů typu jedna k více, přičemž na straně „více“ téhoto vztahu potřebujeme zobrazit více než jednu z příslušných tabulek. Těmto situacím se snažím pokud možno vždy vyhnout. Návrh formulářů, kde

se na straně „více“ nachází několik tabulek, bývá chaotický a jeho implementace bývá ještě chaotičejší; někdy ale opravdu nemáme na vybranou. Jestliže tedy opravdu musíte na jediném formuláři zobrazit několik vztahů typu jedna k více, je nejsnazším řešením oddělit jednotlivé strany „více“ těchto vztahů na ovládacím prvku se záložkami. Takto má totiž uživatel k dispozici zcela jasný kontext. Současně se tím aplikace může urychlit, protože záznamy pro jednotlivou záložku stačí načíst až v okamžiku zobrazení ovládacího prvku se záložkami.

Jedné situaci se ale musíte snažit vyhýbat opravdu co nejpečlivěji: zobrazit současně několik ovládacích prvků s více záznamy tak, že jsou všechny zároveň viditelné. Takové zobrazení je totiž nejen pomalé, ale hlavně ošklivé; ze zobrazení na jediném formuláři se navíc zdánlivě podbízí, že mezi entitami na straně „více“ je nějaký vztah, který ve skutečnosti neexistuje. Na obrázku 14-4 tak naprostě není jasné, jestli zobrazená telefonní čísla patří k dané společnosti, nebo k jednotlivým kontaktům.

The screenshot shows a Windows application window titled "Firma". The form contains the following fields:

- Číslo firmy:** 1
- Název firmy:** Spojené topinkováče s.r.o.
- Adresa:** Vysmahlá 12  
areál Tepláren
- Kontakty:** Steven Buchanan  
Nancy Davolio  
Andrew Fuller  
Janet Leverling  
Margaret Peacock
- Telefonní čísla:** (206) 555-1234  
(425) 555-8080  
(206) 555-9482  
(206) 555-8122  
(206) 555-3412

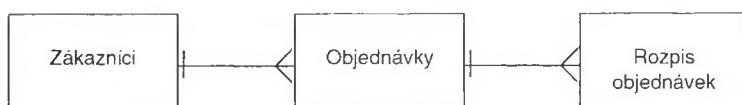
At the bottom, there is a navigation bar with buttons for back, forward, and search, followed by the text "z 5".

Obrázek 14-4 Na tomto formuláři není jasné, jestli telefonní čísla patří k dané společnosti, nebo k jednotlivým kontaktům

## Hierarchie

Za hierarchii bychom v podstatě mohli považovat každý vztah typu jedna k více, uvedený pojednání se ale obvykle používá jen pro takové vztahy, které se skládají ze tří nebo více množin záznamů; je to tedy, chcete-li, jakýsi vztah typu jedna k více k více. Hierarchické vztahy se v datových modelech vyskytují docela často, ve formulářích se ale zpravidla explicitně vyjadřovat nemusí.

Na obrázku 14-5 vidíme například hierarchii o třech úrovních; tvoří ji vztah entity Zákazníci, Objednávky a Rozpis objednávek:



Obrázek 14-5 Na tomto E/R diagramu vidíme hierarchii o třech úrovních

Při tomto typu struktury bude mít většina aplikací formulář Zákazníci, který – pokud vůbec nějaké informace o objednávkách zobrazuje – bude obsahovat pouze souhrnné údaje. Datové položky z entit Objednávky a Rozpis objednávek pak bude zobrazovat samostatný formulář Objednávky. Formulář Objednávky se bude přitom samozřejmě odkazovat i na formulář Zákazníci, zobrazovat bude ale pouze vztah typu jedna k více mezi entitami Objednávky a Rozpis objednávek. Jen málokdy se setkáme s aplikací, která bude skutečně potřebovat zobrazit podrobné informace ze všech tří tabulek.

Těžko říci, jestli je tato snaha vyhýbat se zobrazování hierarchických vztahů vedena nedostatkem skutečné potřeby, nebo obtížností jejich rozumného zobrazení. Dejme tomu, že klient požaduje mít možnost prohlížet si objednávky přímo z formuláře pro údržbu zákazníků. Tuto funkci bychom mohli zabezpečit poměrně snadno: záznamy z tabulky Objednávky (ta se nachází v prostřední úrovni) bychom zobrazili jeden po druhém a záznamy z tabulky Rozpis objednávek všechny najednou. V podstatě by to znamenalo vytvořit z formuláře na obrázku 14-2 (na straně 203) podformulář a vložit jej do formuláře pro údržbu Zákazníků.

Zobrazení záznamů z tabulky Objednávky jeden po druhém však v těchto situacích opravdu není vhodné. Uživatel se spíše bude ptát: „Kolik objednávek tento zákazník podal?“ nebo: „Jaká je jejich průměrná cena?“, než aby se ptal na konkrétní objednané výrobky. A odpověď na tyto docela běžné otázky by bylo dosti těžkopádné, pokud bychom záznamy z tabulky Objednávky zobrazovali jeden po druhém.

Můžeme se tedy rozhodnout zobrazit na formuláři Zákazníci pouze souhrnné informace o objednávkách a uživateli, který požaduje podrobnější údaje, nabídnout mechanismus pro otevření podřízeného (vedlejšího) formuláře Objednávky. Tento přístup ve stylu „souhrn a detaily“ má ale naštěstí také určité dosti závažné nevýhody.

Jestliže si uživatel chce prohlédnout rozpis objednávek a potřebuje z něj zjistit, jaké výrobky si daný zákazník objednává nejčastěji, musí podstoupit pracné otevření druhého formuláře. A protože na formuláři Objednávky se bude obvykle zobrazovat objednávka jen v zobrazení jednoduchého formuláře, je také dosti obtížné porovnat výrobky objednané v různých časových okamžicích.

Donedávna byl jedinou schůdnou alternativou ovládací prvek typu stromového zobrazení. Tyto ovládací prvky jsou výbornými nástroji, avšak množství dat, které se v každé jednotlivé úrovni hierarchie dá zobrazit, je u nich omezeno, protože veškerá data se musí zobrazit na jediném řádku. Pro nás příklad by díky tomuto omezení byl strom zcela nepoužitelný. Jen si zkuste představit, jak by se na jediný řádek vešly třeba všechny informace o zákazníkovi.

S vydáním verze Microsoft Access 2000 a Visual Basicu 6 je ale naštěstí všechno mnohem snazší. Oba tyto vývojové nástroje nabízí zajímavý mechanismus pro zobrazení vnořených dat ve formátu více záznamů. V Accessu 2000 podporují prezentaci hierarchických dat ve formátu osnovy takzvané vnořené datové listy, které vidíme na obrázku 14-6.

Cílo dodavatele	Firma	Kontaktní osoba	Funkce	Adresa	Město	
1 Exotic Liquids	Charlotte Cooper	Nákupčí	49 Gilbert St	Londýn		
2 New Orleans Cajun Delights	Shelley Burke	Vedoucí nákupu	P. O. Box 78934	New Orle		
Číslo výrobku	Název výrobku	Kategorie	Množství v jednotce	Jednotková cena	Jednotky na skladě	Objed.
4 Chef Anton's Cajun Seasoning	Koření	48 x 6 oz sklenic	550,00 Kč	53		
Číslo objednávky	Jednotková cena	Množství	Stavba			
10309	440,00 Kč	20	0%			
10326	440,00 Kč	24	0%			
10336	440,00 Kč	18	10%			
10339	440,00 Kč	10	0%			
10344	440,00 Kč	35	0%			
10464	440,00 Kč	16	20%			
10511	550,00 Kč	50	15%			
10527	550,00 Kč	50	10%			
10533	550,00 Kč	50	5%			
10606	550,00 Kč	20	20%			
10635	550,00 Kč	10	10%			
10636	550,00 Kč	25	0%			
10654	550,00 Kč	12	10%			
10704	550,00 Kč	6	0%			
10726	550,00 Kč	25	0%			
10846	550,00 Kč	21	0%			
10913	550,00 Kč	30	25%			
10950	550,00 Kč	5	0%			
11000	550,00 Kč	25	25%			
11077	550,00 Kč	1	0%			
*	0,00 Kč	1	0%			
5 Chef Anton's Gumbo Mix	Koření	36 krabice	533,75 Kč	0		
65 Louisiana Fiery Hot Pepper Sauce	Koření	32 x 8 oz lahvi	526,25 Kč	76		
66 Louisiana Hot Spiced Okra	Koření	24 x 8 oz sklenic	425,00 Kč	4		
*	(omatické číslo)				0	
3 Grandma Kelly's Homestead	Regina Murphy	Obchodní zástupce	707 Oxford Rd.	Ann Arbo		
4 Tokyo Traders	Yoshi Nagase	Vedoucí oddělení marketingu	9-8 Sekimai	Tokio		
5 Cooperativa de Quesos 'Las Cabras'	Antonio del Valle Saavi	Zástupce pro export	Calle del Rosal 4	Oviedo		
Číslo výrobku	Název výrobku	Kategorie	Množství v jednotce	Jednotková cena	Jednotky na skladě	Objed.
11 Queso Cabrales	Mléčné výrobky	1 kg balení	525,00 Kč	22		
12 Queso Manchego La Pastora	Mléčné výrobky	10 x 500 g balení	950,00 Kč	86		
*	(omatické číslo)			0		
6 Mayumi's	Mayumi Ohno	Pracovník marketingu	92 Setsuko	Osaka		
7 Pavlova, Ltd.	Ian Devling	Vedoucí oddělení marketingu	74 Rose St	Melbourn		
8 Specialty Biscuits, Ltd.	Peter Wilson	Obchodní zástupce	29 King's Way	Manches		
9 PB Knäckebrot AB	Lars Peterson	Prodejce	Kaleodagatan 13	Goteborg		
10 Refrescos Americanas LTDA	Carlos Diaz	Vedoucí oddělení marketingu	Av das Americanas 12 890	Sao Paul		
11 Heli Süßwaren GmbH & Co. KG	Petra Winkler	Vedoucí prodeje	Tiergartenstraße 5	Berlin		

Obrázek 14-6 Vnořené datové listy v Microsoft Accessu 2000 zobrazují hierarchická data

Vnořené datové listy nejsou bůhvíjak půvabné, použitelné ale jsou a fungují. Dají se vnořovat až do osmi úrovní hloubky, na každé úrovni se ale dá jako vnořená zobrazit jen jediná množina záznamů. To znamená, že například nemůžeme vytvořit vnořený datový list, který by na stejně úrovni zahrnoval entity Adresy i Objednávky. Stejný typ hierarchického zobrazení jako vnořený datový list nabízí ve Visual Basicu verze 6 ovládací prvek typu hierarchické mřížky (Hierarchical Flexgrid), který ale navíc umožňuje i vnoření několika množin záznamů na každé úrovni.

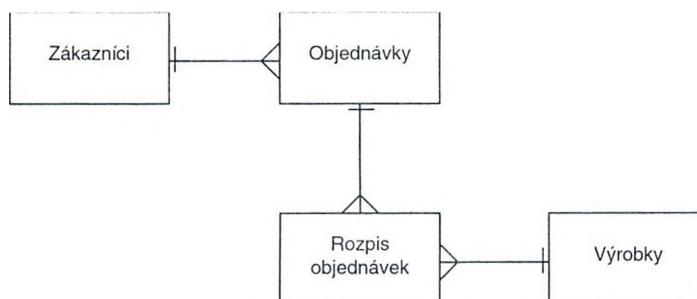
Data zobrazená v ovládacím prvku typu Hierarchical Flexgrid jsou ale naneštěstí určena pouze pro čtení. Pokud uživatel potřebuje data editovat, musíme je svázat s další, doplňující množinou ovládacích prvků, které mohou být buďto na stejném formuláři, nebo na druhém, sekundárním. Toto řešení může tedy zcela zřejmě vést k poněkud nešťastnému uživatelskému rozhraní – vždy je vhodnější umožnit uživatelům editaci dat přímo na místě.

Zobrazení dat v režimu pouze pro čtení je ale často skutečně přijatelné. Klienti zpravidla používají hierarchické zobrazení dat pro různé přehledy, analýzy a pro management, zatímco údržbu dat zabezpečují jiné, oddělené pracovní procesy.

## Relace typu více k více

Posledním typem logických vztahů, které můžeme ve formulářích potřebovat zobrazit, jsou vztahy typu více k více. V databázi reprezentují tyto vztahy tři nebo více tabulek.

V drtivé většině případů můžeme se vztahy typu více k více pracovat stejně, jako by to byly běžné vztahy typu jedna k více. Na obrázku 14-7 vidíme například vztah typu více k více mezi tabulkami Zákazníci a Výrobky, přičemž v roli spojovacích tabulek mezi nimi vystupují dále tabulky Objednávky a Rozpis objednávek.



Obrázek 14-7 Mezi tabulkami Zákazníci a Výrobky je vztah typu více k více

Na tomto obrázku může být docela rozumným požadavkem zobrazit jak všechny výrobky, které si daný zákazník kdy objednal (v tomto případě považujeme za stranu „jedna“ vztahu tabulku Zákazníci), nebo naopak všechny zákazníky, kteří si objednali daný výrobek (pak se na straně „jedna“ nachází tabulka Výrobky). Použijeme zde stejné prezentační postupy, jako u každého jiného formuláře typu jedna k více. Jediné, co musíme vyřešit, je otázka, jestli do formuláře zahrnout také údaje ze spojovacích tabulek a jak ošetřit případné duplicity na straně „více“.

Většina spojovacích tabulek obsahuje pouze primární klíče z obou stran vztahu více k více. Jak jsme si ale řekli v kapitole 3 a jak jsme dále viděli na obrázku 14-7, někdy má i samotný vztah svoje vlastní atributy, které modelujeme jako součást spojovací tabulky. Jestliže potřebujeme tyto atributy zahrnout do navrhovaného formuláře, zobrazíme je na straně „více“.

Pokud tak v našem příkladu máme ve formuláři zobrazit objednané výrobky pro každého jednotlivého zákazníka (to znamená, že se tabulka Zákazníci nachází na straně „jedna“), bude datum objednávky – což je pole z tabulky Objednávky – zcela jasně informací příslušnou k výrobku, nikoli k zákazníkovi. „Zákazník X si objednal výrobek Y dne 15. dubna a výrobek Z 18. dubna...“ Pokud bychom informace zobrazovali opačným způsobem, tedy pokud by se na straně „jedna“ nacházela tabulka Výrobky, bylo by datum objednávky naopak součástí informací o zákazníkovi: „Výrobek X si zakoupil zákazník Y dne 15. dubna a zákazník Z 18. dubna...“ V tomto případě totiž tabulku Zákazníci považujeme za stranu „více“.

Je docela pravděpodobné, že na straně „více“ zobrazeného vztahu budou duplicity (opakování hodnoty), nebo alespoň částečné duplicity. Musíme se tedy rozhodnout, jestli zobrazit každou jednotlivou položku samostatně, nebo jestli uvést pouze souhrnné informace. Jestliže například ke každému výrobku vypisujeme seznam zákazníků, kteří si jej zakoupili, můžeme vypsat zákazníka znovu ke každému jeho nákupu daného výrobku, nebo můžeme vypisovat každého zákazníka jednou a zobrazit k němu vždy celkový počet nákupů, kdy si objednal tento výrobek, a celkové (a možná i průměrné) objednané množství.

Na tomto místě budete ale opatrní, abyste nevypisovali zcela totožné, duplicitní údaje. Nemá smysl vypsat 27krát jméno stejného zákazníka, když k němu nenabízíme žádné doplňující informace, jako je například datum objednávky. Jediným důvodem, proč by uživatel mohl chtít tyto informace vidět, je zjištění počtu objednávek, které daný zákazník podal; v takovém případě by ale veškeré výpočty měla provádět samotná aplikace, a ne je nechávat na nešťastném uživateli.

Zpracování vztahu typu více k více jako vztahu typu jedna k více sice drtivě většině požadavků vyhovuje, někdy ale potřebujeme úplné zobrazení vztahu. Manažer produktů, jenž si prohlíží zákazníky, kteří si zakoupili určitý výrobek, tak bude například potřebovat vědět, jaké další výrobky si tito zákazníci objednali, a podle těchto údajů jim sestaví „nabídkový balíček“.

Relativně jednoduchým řešením je zde ošetřit vztahy jako hierarchie a zobrazit je pomocí vnořeného datového listu nebo mřížky typu Flexgrid. Tento postup v sobě ale skrývá jedno nebezpečí: není z něj jednoznačně jasné, jaké další informace reprezentuje. Jestliže do seznamu výrobků vložíme například vnořený datový list, na kterém jsou uvedena pouze jména zákazníků, nemusí být na první pohled zřejmé, jestli se jedná o jména zákazníků, kteří si tento výrobek zakoupili, nebo jestli se ve skutečnosti jedná o dodavatele, kteří jsou pro nás „zdrojem“ výrobku.

Jestliže se obáváte, že je hierarchické zobrazení zbytečně složité a matoucí, nebo jestliže váš klient obvykle nepožaduje hierarchické informace, může být vhodnější podat příslušné informace v sekundárním okně, jehož význam se již uživateli objasní snáze. V sekundárním okně můžeme kromě přímo zobrazených prostých údajů (nebo místo nich) vypsat také souhrnné informace.

Manažer produktů, který zjišťuje, jaké další výrobky si ještě zákazník zakoupil, může chtít seznam výrobků, zakoupených zákazníky, kteří si zakoupili původní výrobek, uspořádat třeba podle procenta překrývání poptávky – všichni zákazníci, kteří si zakoupili například hodinky, si koupili také

holínky, ale jen 10 procent z nich si koupilo ještě hmoždinky. Tyto souhrnné informace můžeme samozřejmě umístit i na hlavní formulář; protože ale potřebný výpočet je dosti složitý (a tudíž i časově náročný), je vhodnější zobrazit je jen na vyžádání v sekundárním okně – tedy pokud nejsou potřeba příliš často.

Uvedené dvě techniky pro zobrazování všech údajů ve vztahu typu více k více se přitom vzájemně nevylučují. Na hlavním formuláři tak můžeme vytvořit seznam zákazníků pro každý výrobek v hierarchickém zobrazení, zatímco v sekundárním okně, které si uživatel prohlíží jen na vyžádání, nabídneme zobrazení „10 nej- zákazníků“. Jako vždy musíme při konkrétním rozhodnutí vycházet z toho, jakým způsobem se daný formulář bude používat.

### **Stručné shrnutí**

Zatímco v kapitole 13 jsme si řekli, jak můžeme uspořádat formuláře podle úkolů, které budou uživatelé provádět, v této kapitole jsme zkoumali, jakou strukturu ovládacích prvků na formuláři zvolit, a to v závislosti na struktuře zobrazených entit.

Konkrétní zvolenou strukturu formulářů budou určovat především entity, jež budou tyto formuláře reprezentovat, a pokud se má na formuláři zobrazit více než jedna entita, pak také jejich vzájemný vztah. Dále bude strukturu formuláře určovat počet zobrazovaných atributů; z praxe je totiž známo, že není vhodné, aby bylo na formuláři najednou viditelných více než 25 až 30 ovládacích prvků či skupin ovládacích prvků.

V následující kapitole přejdeme na další úroveň návrhu uživatelského rozhraní: budeme se zabývat výběrem jednotlivých ovládacích prvků pro reprezentaci různých typů dat.

# VÝBĚR VHODNÝCH OVLÁDACÍCH PRVKŮ WINDOWS

15

V kapitole 14 jsme si ukazovali různé způsoby definice struktury formulářů, při nichž jsme vycházeli ze vztahů mezi entitami, které tyto formuláře reprezentují. Nyní si v této kapitole řekneme, jak k určitém logickým typům dat vybírat odpovídající vhodné ovládací prvky.

Pro výběr ovládacích prvků platí dva základní principy. Nejdůležitější je zvolit vždy takový ovládací prvek, který nejlépe odpovídá způsobu, jakým uživatel o příslušném údaji uvažuje – tedy jinými slovy, ovládací prvek musí odpovídat mentálnímu modelu uživatele. Za druhé, hodnoty zadávané uživatelem musíme vždy omezit na nejužší možný interval.

Lidé, kteří pracují s databázemi, si často představují data jako textové hodnoty. Jestliže otevřete množinu záznamů v aplikaci Microsoft Access jako tabulkový list, který vidíme na obrázku 15-1, vidíme zde skutečně všechna pole v textové podobě. Ve skutečnosti je ale datového typu Text pouze pole Jméno zákazníka. Číslo zákazníka je však již typu Automatické číslo, Datum první objednávky je Datum/čas, Úvěrový limit je typu Měna a Přednostní zákazník je typu Ano/ne (Boolean).

Stav zákazníků : Tabulka					
Jméno zákazníka	Číslo zákazníka	Datum první objednávky	Úvěrový limit	Přednostní zákazník	
Alfreds Futterkiste	10060	4.9.1998	5 000,00 Kč	ne	
Ana Trujillo Emparedados y helados	10061	15.6.1999	2 000,00 Kč	ne	
Antonio Moreno Taquería	10062	11.1.1999	10 000,00 Kč	ano	
Around the Horn	10063	7.12.1999	15 000,00 Kč	ano	
Berglunds snabbköp	10064	31.1.1998	30 000,00 Kč	ano	
Blauer See Delikatessen	10065	3.1.2000	5 000,00 Kč	ne	
Blondel pere et fils	10066	5.9.1999	20 000,00 Kč	ano	
Bólido Comidas preparadas	10067	18.8.1999	10 000,00 Kč	ano	
Bon app'	10068	28.2.2000	25 000,00 Kč	ano	
Bottom-Dollar Markets	10069	26.7.1999	10 000,00 Kč	ano	
*	0		0,00 Kč	ne	

Obrázek 15-1 Tento datový list zobrazuje několik různých datových typů v podobě textových hodnot

Jako systémoví návrháři si však při manipulaci s daty samozřejmě uvědomujeme jejich různé datové domény (obory hodnot) a nesnažíme se spojit datum s logickou hodnotou typu Boolean. Člověk nicméně často podléhá pokušení zobrazovat hodnoty i různých typů stejným způsobem, obvykle do textových polí. Tento postup samozřejmě funguje – do textového pole se dá vypsat údaj libovolného datového typu – uživateli však nenabízí žádné zvláštní výhody, ani pohodlí. Textové pole považujte proto vždy pouze za onu příslušenou poslední šanci a použijte je jen v případě, kdy se skutečně nehodí žádný jiný, konkrétnější typ ovládacího prvku.

Jestliže se podíváme na pole deklarované nad doménou Datum/čas, můžeme v něm vidět řetězec znaků formátovaných určitým speciálním způsobem, uživatelé v něm však rozhodně vždy uvidí datum. Jejich mentální proces při změně datumového údaje není zdaleka stejný jako při změně textového řetězce. Například uživatel jméno zákazníka, v němž udělá chybu, řekne si: „Místo toho písmene K ve jméně 'Karle' má být M.“ Pokud má ale uživatel změnit datumový údaj, řekne si spíše něco jako: „Tohle by mělo být od pondělka za týden.“ Proto jestliže zde vybereme takový ovládací prvek, který podporuje způsob, jakým uživatel o datech přemýší, usnadníme mu tím život.

Uživateli usnadníme život také omezením hodnot, které smí zadávat. Omezení dat není totéž co validace (neboli ověření platnosti) dat, o které budeme hovořit v následující kapitole. Validace dat je kontrola, kterou systém provádí až po zadání údaje a pomocí které si ověřuje, jestli uživatel zadal „rozumnou“, použitelnou hodnotu. Omezením vstupu dat však uživateli přímo od počátku zabráníme v zadání nesmyslné hodnoty.

Musíme tedy začít tím, že uživateli nejprve omezíme typ dat, jaká smí do pole zadávat. Vzhledem k výběru ovládacích prvků můžeme data rozdělit do čtyř skupin: logické údaje, množiny hodnot, čísla a datum a text. O každé z těchto skupin si v této kapitole povíme podrobněji.

### Poznámka

*V této knize ani při nejlepší vůli nemohou podrobně rozebrat onu zdánlivě nekonečnou řadu různých ovládacích prvků ActiveX, které nabízejí dodavatelé z třetích stran. Svůj výklad jsem proto omezila jen na ty ovládací prvky, které nabízí Microsoft Visual Basic a Microsoft Access. Pochopitelně vám ale nebráním v tom, abyste se seznámili s čímkoliv, co je na trhu k dispozici. (Dobrým místem na začátek je webové sídlo firmy Microsoft.) Kolem sebe zajistě najdete spoustu nepotřebného baraburdí, ale také hodně zajímavých a užitečných věcí – a nakonec třeba zjistíte, že několik stovek dolarů za vhodné nástroje od třetích stran rám může ušetřit týden dlouhého vývoje, testování a bezesných nocí.*

## Reprezentace logických hodnot

Logické hodnoty můžeme sice zobrazit do textového pole, není to ale příliš vhodný postup. Tabulka Zákazníci může například obsahovat pole se jménem Úvěr potvrzen, které deklarujeme jako hodnotu typu Boolean (neboli v Microsoft Accessu jako datový typ Ano/ne). Toto pole bychom mohli zobrazit pomocí textového pole a poté při validaci dat kontrolovat, jestli uživatel zadal správně „Ano“, „Ne“, nebo třeba „Yes“, „No“, „True“ či „False“. Takovým povolením nijak neomezeného zadávání dat si ale jenom koledujeme o to, aby uživatelé začali zkoušet zadávat nesmysly jako „Nevím“, „Ani náhodou“ a podobně. Pokud tedy netoužíte právě po přebírání a dalším zpracování a interpretaci takovýchto hodnot, je lépe usnadnit uživateli život ovládacím prvkem, který je omezen jen na dvě možné hodnoty.

Microsoft Access i Visual Basic přitom skutečně nabízí hned dva ovládací prvky, které jsou pro tuto situaci vhodnější: jsou to zaškrťávací políčka a přepínací políčka a vidíme je na obrázku 15-2.



Obrázek 15-2 Každý ovládací prvek typu zaškrťávací políčko a přepínací políčko zde zobrazuje jednu logickou hodnotu

Zaškrťávací políčka jsou většině lidí důvěrně známá a pro reprezentaci většiny logických hodnot jsou skutečně ideální. Přepínací tlačítka se tak často nepoužívají. Hodí se spíše pro booleovské hodnoty, které jsou „zapnutý“ nebo „vypnutý“, než pro hodnoty, které se dají vyjádřit jako „ano“ a „ne“. Problémem přepínacích tlačítek ale je, že ve svém „vypnutém“ stavu se nedají rozlišit od běžných příkazových tlačítek. A protože uživatel zpravidla očekává, že stiskem tlačítka spustí provedení určité akce (činnosti), může se bát stisknout tlačítko s popisem „Úvěr potvrzen“, protože si bude myslet, že se jím spouští proces potvrzení úvěru a nenapadne jej, že pouze jednoduše indikuje jeho dokončení. Z toho důvodu používám přepínací tlačítka nejraději ve skupinách, podobně jako tlačítka přepínačů.

Co se týče tlačítek přepínačů (říká se jim také „rádiová“ tlačítka, protože připomínají přepínače vlnových rozsahů na starých rozhlasových přijímačích), nepokoušejte se prosím pomocí nich reprezentovat jednotlivé logické hodnoty. V programovém prostředí Microsoft Windows vám v tom pochopitelně nic nemůže zabránit, je to ale zcela zbytečné (úplně stejně dobře poslouží obyčejné zaškrťávací políčko) a navíc i nevhodné. Tlačítka přepínačů mají totiž vyjadřovat množinu vzájemně se vylučujících variant. Jediné tlačítko přepínače vypadá nejen osaměle, ale tím pádem i docela podivně.

Ještě horší ale je, že pokud byste pomocí tlačítek přepínačů reprezentovali několik logických hodnot ve stejné oblasti formuláře, uživatel by se mohl mylně (ale po právu) domnívat, že tyto volby spolu nějakým způsobem souvisí a že z nich může vybrat vždy pouze jednu. Zároveň by očekával, že po výběru jedné z těchto voleb zmizí výběr u ostatních voleb domnělé skupiny. A uživatelé vždycky rozladí, když se počítačový systém chová jinak, než by on sám očekával.

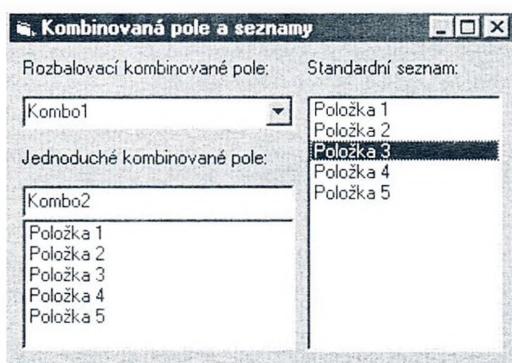
## Reprezentace množiny hodnot

Řada ovládacích prvků umožňuje zobrazit na formuláři určitou množinu hodnot. Konkrétní ovládací prvek si volíme z velké části podle toho, jestli potřebujeme jako výsledek vybrat jedinou hodnotu (přestože pochází z jisté množiny hodnot), nebo jestli připouštíme několik hodnot.

### Výběr jediné hodnoty z množiny hodnot

Určitá hodnota bývá často pro dané pole přípustná jen za podmínky, že se zároveň nachází v jistém seznamu. Hodnota pole Číslo zákazníka v záznamu tabulky Objednávky bude například platná tehdy a jen tehdy, pokud toto číslo existuje v kmenové tabulce Zákazníci. Jestliže zde budeme chtít vstup uživatele omezit jen na množinu přípustných hodnot, nabídnete mu seznam Zákazníků a umožníme mu vybrat z něj jednoho konkrétního zákazníka.

V této situaci se nejčastěji používají ovládací prvky typu kombinované pole a seznam (pole se seznamem), které vidíme na obrázku 15-3. Funkce obou těchto ovládacích prvků se v jistých ohledech liší; nejdůležitější rozdíl spočívá v tom, že do kombinovaného pole může uživatel zadávat i novou hodnotu, která se v seznamu nenachází. Jak jsme si již řekli na jiném místě, pomocí této funkce můžeme někdy vytvářet také záznamy propojené entity. I když ale zadávání nových záznamů v daném prostředí nevyužijeme, může se nám hodit druhá funkce kombinovaného pole, a sice možnost vyhledání hodnoty přímým zápisem počátečních znaků. Řada lidí totiž raději píše na klávesnici a hledat někde na stole myš, nebo se dokonce dívat na obrazovku, je pro ně nepohodlné. Z toho důvodu preferuji ve většině situací kombinovaná pole.



Obrázek 15-3 Visual Basic nabízí celkem tři různé typy kombinovaných polí

Ve Visual Basicu můžeme vhodnou konfigurací kombinovaného pole určit, jestli má být seznam zobrazen neustále, nebo jen na vyžádání. Microsoft Access zná oproti tomu pouze rozbalovací kombinovaná pole. (V seznamech ve Visual Basicu se mohou vedle každé jednotlivé hodnoty zobrazovat navíc zaškrťovací políčka; Microsoft Access tuto funkci nepodporuje.)

Kníha *The Windows Interface Guidelines for Software Design* říká, že prostá (přímo zobrazená) kombinovaná pole a seznamy by měly mít takovou velikost, aby se v nich mohlo zobrazit tři až osm položek. Jestliže na formuláři není pro ovládací prvek o takové velikosti místo, je vhodnější použít kombinované pole s rozbalovacím seznamem.

Kombinované pole s rozbalovacím seznamem je lepší také v případě, že po výběru položky již zobrazení seznamu nemá smysl, i když je třeba na formuláři dostatek místa pro trvalé zobrazení seznamu. Pokud například na formuláři Objednávky nabízíme seznam zákazníků, stačí uživateli vybrat jednoho konkrétního požadovaného zákazníka; poté jej již nezajímá, kteří další zákazníci jsou v systému známí.

Někdy je užitečné, pokud má uživatel možnost vidět hned na první pohled seznam přípustných hodnot, zejména pokud se samotný seznam nebo hodnota přiřazovaná do záznamu často mění. V knihovnickém systému, kde se například jednotlivé knižní tituly zařazují do předmětových kategorií definovaných v seznamu, který se často aktualizuje a upřesňuje, bude uživatel například chtít vidět i aktuální seznam, z něhož zjistí, jestli nebudou knihu lépe charakterizovat nějaká nová kategorie.

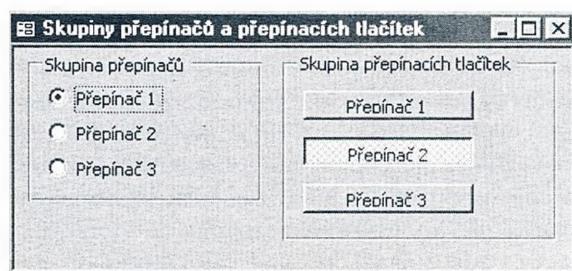
Mimořádně užitečnou funkcí jak ve Visual Basicu, tak i v Microsoft Accessu je možnost zobrazit jinou hodnotu, než jaká je fyzicky uložena v tabulce. Jestliže je například primárním klíčem tabulky Zákazníci nějaké počítadlo (automatické číslo) nebo hodnota typu Identity, uložíme tuto hodnotu zároveň do tabulky Objednávky jako cizí (nevlastní) klíč, zobrazovat ji ale uživateli nemá smysl. Ve formuláři Objednávky tak můžeme snadno zobrazit jména zákazníků (z tabulky Zákazníci), z nichž si teprve uživatel bude vybírat.

Ve Visual Basicu stačí do vlastnosti DataField a ListField příslušného ovládacího prvku zadat jinou hodnotu. V Microsoft Accessu použijeme v seznamu více sloupců, z nichž jeden bude obsahovat jméno zákazníka a druhý číslo zákazníka; ovládací prvek pak svážeme se sloupcem pro číslo zákazníka v tabulce Objednávky a sloupcem s číslem zákazníka přiřadíme nulovou šířku, takže jej fakticky skryjeme.

Možnost zobrazovat v kombinovaných polích Microsoft Accessu více sloupců je mimořádně užitečná a vždy mi přišlo docela škoda, že něco podobného ve Visual Basicu není možné. Při zobrazení seznamu zákazníků je například docela šikovně vypsat vedle každého zákazníka ještě nějaké další, doplňující informace, které jej blíže identifikují, například město, v němž zákazník bydlí. Ve Visual Basicu neexistuje jiná možnost, než zřetězením hodnot vytvořit vypočtené pole, které pak použijeme ve vlastnosti ListField ovládacího prvku seznamu. (Zde autorka nemá tak docela pravdu: kombinovaná pole o více sloupcích ve Visual Basicu existují – stačí do vlastnosti ColumnCount přiřadit počet sloupců a poté jednotlivé položky definovat pomocí vlastnosti List, která se tváří jako pole. Autorka má zřejmě na mysli propojení s databází. Pozn. překl.) Toto řešení však z nějakého záhadného důvodu nefunguje vždy tak dobré jako kombinovaná pole v Microsoft Accessu, zejména pokud jsou některé hodnoty prázdné.

Seznamy a kombinovaná pole jsou zajisté velice užitečné, pokud však seznam obsahuje příliš velké množství hodnot, již tak praktické nejsou. U několika stovek položek (neřku-li více) musíme seznam nějakým způsobem omezit. Uživatel tak může vybrat například počáteční písmeno jména zákazníka, jemu příslušnou oblast prodeje, nebo třeba okres, a poté již seznam na základě provedeného výběru odfiltrujeme.

Pokud je seznam naopak velice krátký – tedy pokud nemá více než pět nebo šest položek – a pokud jsou jeho hodnoty navíc pevné, je vhodnější použít namísto kombinovaného pole či seznamu raději skupinu tlačítek přepínače. Skupinu tlačítek přepínače můžeme přitom implementovat buďto pomocí tlačítek přepínače, nebo pomocí přepínacích tlačítek. Obě možnosti vidíme na obrázku 15-4; běžněji se používají tlačítka přepínačů.



Obrázek 15-4 Tlačítka přepínačů nebo přepínací tlačítka uzavřená ve skupinovém poli mohou zobrazovat krátké seznamy s pevnými hodnotami

Skupinu voleb přepínačů je možné vygenerovat i za běhu, alespoň ve Visual Basicu, rozhodně to ale není vhodné. Změna rozvržení formuláře na obrazovce – tedy i přidávání a odstraňování voleb jakoby z ničeho nic – může uživatele vyvést z míry. Jestliže tedy seznam voleb není pevně daný (nebo alespoň pevně daný do další verze aplikace), použijte raději seznam nebo kombinované pole. Velikost těchto ovládacích prvků se při přidávání či odstraňování položek ze seznamu nemění, takže pro uživatele neznamená změna seznamu žádné nepohodlí, ani žádnou nejistotu.

## Výběr množiny hodnot

Jestliže potřebujeme vybrat určitou množinu hodnot, pohybujeme se ve vztahu typu jedna k více na straně „více“. Na prvním místě, jak jsme již viděli v kapitole 14, se přitom v této situaci musíme rozhodnout, jestli potřebujeme záznamy zobrazovat a vybírat všechny najednou, nebo postupně jeden po druhém.

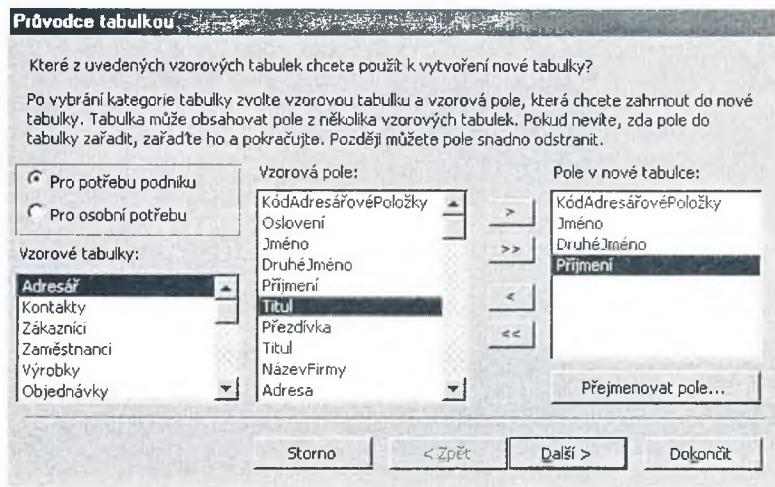
V Microsoft Accessu můžeme pro zobrazení a výběr jednotlivých záznamů na straně „více“, jeden po druhém, použít podformulář v zobrazení jednoduchého formuláře. Stačí odpovídajícím způsobem na definovat vlastnosti LinkChildFields a LinkMasterFields a Access již většinu práce udělal za nás. Ve Visual Basicu je s vytvořením takového podformuláře o něco více práce (ano, je to spíš o hodně více práce), na opaktu však získáme nad podformulářem o něco vyšší stupeň kontroly. Každopádně je tento postup velice dobrý v případě, že potřebujeme načíst hodnoty několika polí, a zejména pokud potřebujeme pracovat s různými typy ovládacích prvků.

Potřebujete-li u každého záznamu shromáždit od uživatele více různých hodnot a zároveň potřebujete zobrazit několik záznamů současně, můžete v Microsoft Accessu použít datový list (viděli jsme jej na obrázku 14-2 na straně 203), respektive ve Visual Basicu ovládací prvek typu mřížka (Grid), případně hierarchickou mřížku Microsoft Hierarchical FlexGrid. Tyto mechanismy umožňují kromě textových polí ještě několik dalších typů ovládacích prvků. Upřímně řečeno, mřížky ve Visual Basicu se ale implementují poněkud obtížně a hierarchické mřížky Microsoft Hierarchical FlexGrid umožňují pouze zobrazení dat, nikoli však jejich editaci.

Naštěstí ale jak Microsoft Access, tak i Visual Basic nabízí jisté alternativy, u kterých máme nad zobrazením formulářem lepší kontrolu. Ovládací prvek podformuláře v Microsoft Accessu tak podporuje zobrazení typu nekonečného formuláře, v němž si můžeme prohlížet několik záznamů najednou. Ve Visual Basicu verze 6 je nově zaveden ovládací prvek DataRepeater, který má v podstatě stejně funkce, i když je implementován odlišným způsobem. Oba uvedené ovládací prvky se hodí pro případ, kdy potřebujeme zobrazit více záznamů a zároveň potřebujeme použít některý z ovládacích prvků – jako je například skupina přepínačů – který ovládací prvek typu datový list nebo mřížka nepodporuje. U obou ovládacích prvků může přitom každý záznam zabírat i více než jeden řádek.

Dalším ovládacím prvkem, který může být za jistých okolností vhodný pro zobrazení více záznamů, je ovládací prvek typu stromového zobrazení. Tento ovládací prvek se nejčastěji používá pro zobrazení hierarchických dat ve struktuře osnovy, dají se ale pomocí něj zobrazit také vybrané podrobnosti ke každému záznamu. Pro editaci těchto podrobných informací však již stromová struktura není efektivní; na druhé straně je ale užitečná pro kontrolování záznamů. Podobně jako v Průzkumníku systému Microsoft Windows můžete i zde zobrazit do jiné části formuláře podrobné informace k vybranému záznamu.

A konečně v situacích, kdy uživatel potřebuje vybrat ze seznamu množinu položek, může být dobré použít propojenou dvojici seznamů. Tato struktura, kterou vidíme v seznamech Vzorová pole a Pole v nové tabulce na obrázku 15-5, se často používá v průvodcích. (Níže uvedená obrazovka pochází z Průvodce tabulkou v Microsoft Accessu 2000.) Uživatel takovým propojeným seznamem velice snadno porozumí, na druhé straně vás ale upozorňuji, že jejich implementace tak snadná není.



Obrázek 15-5 Průvodce tabulkou v Microsoft Accessu 2000 nabízí uživateli pomocí propojených seznamů výběr polí do nové tabulky

Struktura s takovou dvojicí seznamů je pohodlná pro prvotní pořizování dat, při následné editaci a zobrazování dat však již vhodná není – když pro nic jiného, tak proto, že na formuláři zabírá příliš mnoha místa. Proto se tato technika využívá nejčastěji v průvodcích, kde každou část operace pořizování dat můžeme ošetřit na samostatné obrazovce. Po dokončení prvotního zadání dat však již uživatele nejspíše nebude zajímat úplný seznam vybraných a nevybraných hodnot; dále je již tedy efektivnější využít některý z výše popsaných ovládacích prvků, nebo dokonce jediný seznam s možností vícenásobného výběru.

Výběr v seznamech s možností vícenásobného výběru však pracuje poměrně zvláštním způsobem, díky němuž mohou být tyto seznamy nebezpečné. Jestliže uživatel klepne na novou položku myší bez současného stisku klávesy Ctrl, může si tak velice snadno nedopatřením zrušit výběr u všech pracně zvolených položek. Jestliže je navíc tento výběr svázaný s daty, může být vyřešením operací přidávání záznamů, jejich odstraňování a opětovných výběrů téměř neřešitelný problém. Vhodnějším řešením je tedy druhý, normální seznam s možností výběru jen jediné položky, ve kterém se zobrazují jen položky vybrané v první seznamu; snad jen k němu doplníme textové pole či kombinované pole pro přidávání nových položek.

## Reprezentace číselných a datumových údajů

Číselné a datumové hodnoty se nejčastěji prezentují v textových polí. Jako vždy je ovšem lepší číselné hodnoty, které může uživatel zadávat, pokud možno všude omezit. Pokud je ale interval přípustných hodnot příliš široký, nemusí se nám to podařit.

Určitou kontrolu nad zadáváním číselných údajů v textovém poli nabízí vlastnost InputMask v Microsoft Accessu, respektive ovládací prvek MaskedData ve Visual Basicu – přinejmenším zde můžeme zakázat zadávání abecedních (nečíselných) znaků. Access dokáže navíc zadaný údaj poměrně inteligentně interpretovat; to je již ale ona validace „s křížkem po funuse“, proti které existují i lepší možnosti omezení vstupních dat.

Visual Basic verze 6 nabízí dva nové ovládací prvky pro zadávání kalendářových údajů, a sice ovládací prvek MonthView a ovládací prvek DateTimePicker; oba vidíme na obrázku 15-6. Microsoft Access 2000 má pak ovládací prvek typu kalendář, který vypadá podobně jako ovládací prvek MonthView z Visual Basicu.

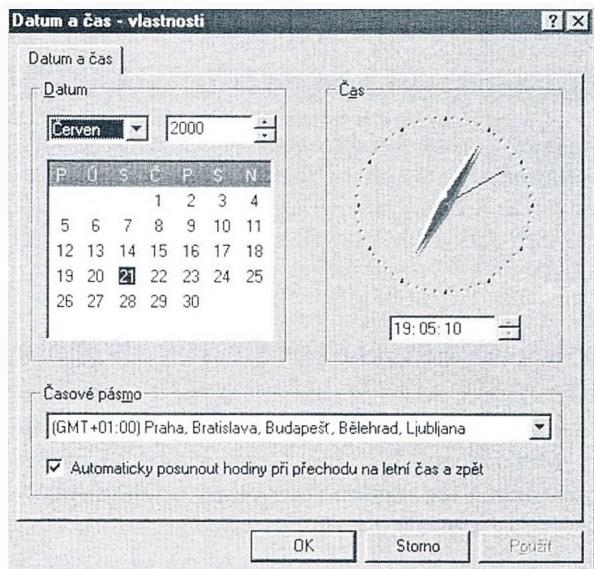


Obrázek 15-6 Ovládací prvky MonthView a DateTimePicker

Tyto dva ovládací prvky jsou v podstatě kalendářovými ekvivalenty seznamu a kombinovaného pole – ovládací prvek DateTimePicker zobrazuje kalendář jen na vyžádání, zatímco v ovládacím prvku MonthView je zobrazen vždy. Oba ovládací prvky však dokáží zpracovat pouze datum, nikoli zároveň datum a čas. To může vést při vazbě na pole typu Datum/čas k jistým nepříjemným komplikacím, takže při implementaci si musíme počítat opatrně.

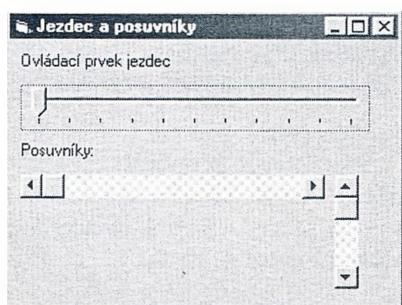
Visual Basic a Microsoft Access nabízí také ovládací prvek Microsoft UpDown (kterému se někdy říká spinner, česky číselník), pomocí něhož se dájí zadávat jak číselné hodnoty, tak i údaje typu Datum/čas. Většina uživatelů zná ovládací prvek typu UpDown například z nastavení data a času v systému Windows, jak to vidíme na obrázku 15-7.

Ovládací prvky typu UpDown jsou užitečné zejména v situacích, kdy se příslušná podkladová hodnota nemůže zvětšovat stejnomořně – určité datum se například smí vybírat jen z pracovních dnů, nebo číselná hodnota musí být zaokrouhlena na nejbližší celou stovku.



Obrázek 15-7 V systému Windows se ovládací prvky typu UpDown používají k nastavení data a času

Poslední skupinu ovládacích prvků v jazyce Visual Basic, které můžeme využít pro zadávání číselných údajů, jsou jezdce a posuvníky; ukazuje je obrázek 15-8.



Obrázek 15-8 Ovládací prvky typu jezdec a posuvník mají v databázových aplikacích jen omezené možnosti využití

Tyto ovládací prvky mají ale v databázových aplikacích poměrně omezené možnosti využití. Jedná se v podstatě o jisté grafické mechanismy pro nastavení číselné hodnoty a jako takové by se teoreticky daly použít pro zadání libovolného číselného údaje. (Posuvníky sice nejspíše nebudeeme považovat za číselné nástroje – to není jejich symbolický model – přesto ale skutečně na pozadí vrací do aplikace číselnou hodnotu.)

Osobně je považuji za nejužitečnější v případech, kdy má uživatel porovnat určitou množinu hodnot, u nichž není tak důležitá přesná hodnota, jako jejich relativní pozice – to znamená, že „hodnota tohoto pole je důležitější než hodnota tohoto pole“. Několikrát jsem například použila posuvník pro kontrolu shody, kdy si uživatel při hledání konkrétního záznamu mohl nastavit relativní důležitost nebo požadovanou toleranci porovnávaných polí.

## Reprezentace textových hodnot

Nakonec jsme se tedy dostali k oněm ovládacím prvkům „poslední instance“, tedy k textovým polím. Textová pole ale nejsou nic zlého: jenom se používají přespíši. Většina dat v typické databázi je skutečně textových – a pro jejich reprezentaci se textová pole velice dobře hodí. Výjimkou je pole, které tvoří cizí (nevlastní) klíč; v takovém případě je obvykle vhodnější použít kombinované pole nebo seznam, jak jsme si řekli před chvílí.

Základní principy výběru ovládacích prvků platí stejnou měrou i pro textová pole: používejte je v těch situacích, kde odpovídají mentálnímu modelu uživatele, a zároveň v nich zavádějte jakékoli použitelné techniky pro omezení vstupu dat.

Omezení vstupu dat definuje v Microsoft Accessu vlastnost InputMask ovládacího prvku textového pole a ve Visual Basicu speciální ovládací prvek MaskedTextBox. Do obou těchto vlastností zapisujeme vzorek, který řídí požadovanou podobu zadávaných dat. Rodné číslo osoby v České republice bude mít například vzorek #####/### (šest číslic, lomítko a čtyři číslice).

Vstupní masky mají v jistých situacích svůj smysl; tyto situace jsou ale poměrně vzácné a naopak můžeme pomocí nich zadávání dat velice snadno omezit příliš. Dobrým příkladem jsou telefonní čísla: mohli bychom podlehnout pokušení nadefinovat pro jejich hodnoty vstupní masku – předtím se ale musíme přesvědčit, že systému stačí akceptovat jen telefonní čísla ze stejného státu. Telefonní čísla v zahraničí mají totiž často jiný formát – nemluvě o mezinárodní předvolbě, která se k nim zpravidla zapisuje. I v jedné zemi musíme uvažovat jisté varianty – například podnikové linky a předčíslí.

Nezapomeňte, že i při definování vstupní masky je naším cílem usnadnit uživateli život. Pokud jim zabráníme v zadání naprostě správných, platných – jenom neočekávaných – údajů, život jim naopak pouze zkompplikujeme.

Kromě standardních textových polí podporují Microsoft Access i Visual Basic pro zadávání textu také ovládací prvek Microsoft RichTextBox, který umožňuje kombinaci různých řezů a stylů písma. Implementace ovládacího prvku typu RichTextBox je složitější než práce s jinými ovládacími prvky, protože u něj musíme nadefinovat také rozhraní pro nastavení vlastností textu.

U ovládacího prvku RichTextBox je ale složitější nejen potřebné rozhraní, ale také manipulace s údaji v něm zapsanými, protože má přímo v sobě vloženo formátování textu. Z toho důvodu doporučuji používat ovládací prvek typu RichTextBox jen v situacích, kdy tyto jeho doplňující funkce znamenají pro uživatele nějaký významný přínos. Navíc je třeba tyto ovládací prvky omezit jen na takové hodnoty, s nimiž se bude pracovat zásadně jako s bloky textu, které se někde zobrazí, ale nikdy se podle nich nebude vyhledávat, ani se nebudou spojovat s jinými hodnotami.

Ovládací prvek typu RichTextBox může být užitečný například pro správu předdefinovaných vzorových odstavců textu, jejichž kombinací sestavíme standardizovaný dopis. Vlastní odstavce, které se vloží do výsledného dokumentu, budou uloženy jako bloky formátovaného textu, uživateli si je ale vybere na základě popisných informací, případně podle různých kategorií – tedy na základě informací uložených v podobě standardního, neformátovaného textu.

### Stručné shrnutí

V této kapitole jsme rozbehrali různé typy ovládacích prvků, které můžeme v aplikacích využít pro zobrazování a naopak pro načítání informací. Konkrétní zvolené ovládací prvky musí vždy odpovídat mentálnímu modelu dat uživatele a zadávání dat je třeba pokud možno omezit jen na množinu přípustných hodnot.

V následující kapitole se podíváme, co se dá dělat v případě, že omezení dat není možné provést; budeme zde tedy hovořit o implementaci mechanismů validace dat.



Představte si, že jste vlastníkem malé výrobní společnosti a že se velice usilovně snažíte přetáhnout zákazníka od jiného, většího a lépe zavedeného konkurenta. Nakonec zákazníka přesvědčíte, že to s vámi zkusí, ale jeho podmínkou je, abyste mu doručili zboží do 48 hodin. Ověříte si ve výrobě, jestli to zvládnou (jen tak tak), takže otevříte láhev šampaňského.

O deset minut později přichází první problém. Vypadá to, že účetní objednávku nepodepíše, dokud nebude potvrzen úvěrový rámec zákazníka – a to zabere zhruba asi týden. Bez podepsané objednávky se ale nemůže rozjet výroba. Co teď uděláte? Začnete asi účetního přesvědčovat, že si daná situace vyžaduje menší „ohnutí“ zavedených pravidel. Pokud to účetnímu nestačí, vysvětlíte mu, že požadavek ověření úvěru před dokončením objednávky je vaše pravidlo, a protože i firma je konec konců vaše, můžete toto pravidlo kdykoli podle potřeby porušit. A když ani to nepomůže, prostě toho „idiota“ vyhodíte z práce a potřebné papírování si zajistíte sami.

Moc dobře vím, že i účetní jsou jenom lidé a že mají také svoje rodiny a problémy jako my ostatní, takže scénář, který jsem právě načrtla, asi není příliš pravděpodobný. Lidé prostě tak tvrdě neodmítnou vykonat příkaz svého šéfa (nebo alespoň ne příliš často). Počítačové systémy to ale zcela tvrdošíjně dělají, a to zcela normálně. Dupnou si svou malou elektronickou nohou, elektronicky vám vynadají a odmítnou vyplnit docela rozumný požadavek – vše ve jménu „zachování datové integrity“. Jestliže jsem vám někde na začátku této knihy říkala, co to datová integrita je, jak se modeluje a kde se implementuje, nyní vám řeknu něco opravdu smutného: zachování datové integrity vůbec není důležité. (Teď se na chvíli zastavím a počkám, až vaše protestné mručení přestane.) Ne, tím opravdu neříkám, že máte pomínout *validaci* dat. Tím jenom říkám, že zachování datové integrity v databázi je mnohem méně důležité než pomáhat uživatelům; jen podle toho musíme systém navrhovat. Data musí být v systému *nakonec* platná; to ale neznamená, že by musela být naprostě správná a platná již v okamžiku zadání.

Ted se i vy na chvíli zastavte a přemýšlejte, proč vlastně ten který databázový systém vyvíjíte: jeho úkolem je pomáhat lidem v nějaké práci. Validace dat, která pomáhá lidem v práci, je v pořádku; podporuje cíle systému. Taková validace dat, která zabraňuje lidem provádět úkoly v takovém pořadí, jaké má pro ně smysl, nebo jim zabraňuje v provedení něčeho docela rozumného, pouze neočekávaného, je však již špatná. Je to opravdu tak jednoduché. Systém *nikdy* nesmí uživateli zabranovat v zadání dat jen proto, že nejsou úplná nebo že jsme s nimi při návrhu systému nepočítali. Něco takového se ale samozřejmě někdy snadno řekne, ale hůře udělá. V této kapitole se podívaláme na různé typy omezení integrity a řekneme si, jak v systému dostat do vzájemné rovnováhy ochranu dat proti nahodilým chybám a zachování spolehlivosti a použitelnosti systému.

## Třídy omezení integrity

V kapitole 4 jsme si omezení integrity rozdělili do šesti různých typů, a to podle jejich logické úrovně v relačním modelu. V této kapitole budeme používat jiné rozdělení. Všechna omezení integrity si zde rozdělíme do dvou základních tříd: jsou to *vestavěná omezení* a *aplikaciční omezení* (tém se říká také *aplikaciční pravidla*).

Vestavěná omezení řídí fyzickou strukturu dat a odvozují se přímo z relačního modelu. Pravidlo, které zakazuje uživateli odstranit z tabulky Zákazníci záznam, k němuž existují nějaké svázány záznamy v jiných tabulkách, je příkladem takového vestavěného omezení, protože referenční integrita je funkcí relačního modelu. Jestliže uživatel odstraní záznam zákazníka, aniž by zároveň odstranil i ostatní na něm závislé záznamy, dojde k destabilizaci databáze. Pokud by se do databáze někdy později přidal jiný záznam, jehož primární klíč má náhodou stejnou hodnotu jako primární klíč záznamu odstraněného zákazníka, mohly by se původně osiřelé záznamy objednávek nesprávně asociovat se záznamem nového zákazníka. K něčemu takovému může docela snadno dojít například tehdy, pokud bychom hodnotu primárního klíče odvozovali ze jména zákazníka.

Kvůli osiřelým záznamům však databáze může vracet falešné (nesprávné) výsledky i v případě, že se primární klíč nikdy znova nepoužije. Dotazy, které počítají celkové množství výrobků prodaných za určité časové období, vrací například různé výsledky podle toho, jestli je tabulka Objednávky spojena s tabulkou Zákazníci nebo ne. Při sestavování podrobného výpisu všech počtu každého výrobku, prodaných každému jednotlivému zákazníkovi, se tabulky Zákazníci a Objednávky obvykle spojují pomocí operace přirozeného spojení. Do výpočtů souhrnných údajů o prodeji budou zahrnuty jen ty záznamy z tabulky Objednávky, které mají v tabulce Zákazníci nějaký odpovídající záznam. K osiřelým záznamům v tabulce Objednávky neexistuje žádný odpovídající záznam tabulky Zákazníci, takže ty se výpočtu neúčastní. U souhrnných přehledů prodeje, při jejichž sestavování je tabulka Objednávky propojena jen s tabulkou Výrobky, se však osiřelé záznamy do výsledků provedených výpočtu naopak promítnou. Na otázku: „Kolik zboží jsme prodali v červnu?“ dá systém tudíž dvě různé odpovědi závislé na přesné formulaci otázky – a to je zřejmě zcela nepřijatelné.

Aplikační omezení vycházejí na druhé straně z daného prostoru problému. Pravidlo, které zakazuje uživateli odstranit záznam zákazníka a v kaskádě s ním i odpovídající záznamy z tabulky Objednávky, pokud všechny odpovídající objednávky nebyly dosud uspokojeny nebo stornovány, je příkladem takového aplikačního omezení.

Aplikační omezení tak v podstatě říkají, „tohle tady neděláme“, zatímco vestavěná omezení říkají, „tohle se vůbec nedá udělat“. V praxi však hranice mezi vestavěnými omezeními a aplikačními omezeními není vždy zcela jasná a vlastně ani vůbec není důležitá. Jediné, co je zde skutečně důležité, že se jedna třída omezení datové integrity odvozuje přímo z prostoru problému. Tato omezení implementujeme v podstatě pro pohodlí uživatele, takže pokud je naopak pro uživatele výhodné a pohodlné je obejít, můžeme je bezpečně ignorovat. Odstranění zákazníka, který má rozpracované nějaké objednávky, není v žádném případě výhodné – pro firmu by to mohla být docela klidně menší katastrofa. Přidat do určitého oddělení šestého zaměstnance, přestože podle jistého aplikačního pravidla má jednomu vedoucímu přímo podléhat nejvýše pět zaměstnanců, však

může být výhodné. Naopak velice nevýhodné a nepohodlné by bylo, kdybychom tohoto zaměstnance do systému zadat nemohli.

Závěr je takový, že ani odmítnutím aplikáčních omezení nemůžeme ohrozit stabilitu či spolehlivost databáze. Jestliže uživatel odstraní záznam zákazníka a zároveň odstraní s ním spojené rozpracované objednávky, ztratí systém jisté dosud důležité údaje, k ohrožení žádných jiných dat však již nedochází. Příslušné záznamy tabulek Zákazníci a Objednávky se dají obnovit nebo pořídit znova a všechno bude zase v pořádku.

Systém se musí k oběma třídám omezení datové integrity – tedy k vestavěným omezením a k aplikáčním omezením – stavět odlišným způsobem. Vestavěná omezení se nedají ignorovat, aniž bychom ohrozili spolehlivost dat. Aplikáční omezení se oproti tomu dají – a často i musí – na základě úsudku samotného uživatele potlačit. V následujících částech textu rozebereme tedy každou ze tří omezení integrity podrobněji.

## Vestavěná omezení

Validační pravidla první třídy, kterým zde říkám vestavěná omezení, ovládají fyzickou strukturu databáze. Do této třídy tak náleží pravidla, která definují typ, formát a délku dat, dále přípustnost prázdných hodnot Null, omezení intervalu a konečně omezení entitové integrity a referenční integrity.

### Datový typ

Za předpokladu, že pro datové položky zvolíme odpovídající typy ovládacích prvků, se uživatelé do konfliktu s omezeními datového typu za normálních okolností nedostávají. Neznám nikoho, kdo by se pokoušel do pole Částka splátky zadat datum, nebo kdo by zapisoval do zaškrťávacího políčka text – ledaže se dotyčný nedívá pořádně na obrazovku a podaří se mu stisknut klávesu Enter dvakrát po sobě. Někdo se ale může snažit třeba do textového pole, které očekává zadání číselné hodnoty, zapsat text „čtrnáct“; právě proto je tedy velice důležité zvolit pro zadávání údajů správné ovládací prvky, jak jsme si říkali v předcházející kapitole.

### Formát

Formátování dat neznamená obvykle žádný velký problém, zejména pokud můžeme údaj zadáný uživatelem po opuštění příslušného pole vhodně přereformátovat (zejména v Microsoft Accessu je to velice jednoduché), nebo pokud můžeme pro zadání údaje uživatelem definovat odpovídající vstupní masku. Budte ale opatrní a dejte si pozor, abyste formát nenadefinovali jako příliš omezující. Jestliže zadáný údaj přesně neodpovídá formátu, který jste pro něj definovali, je lepší pokusit se jej naformátovat, jak nejlépe umíme, ale poté uživateli umožnit jeho případnou úpravu. Pokud neplatný formát znamená přímo neplatná data, je takový postup nepoužitelný – tato situace je ale poměrně vzácná. Uživatel, který zadává telefonní číslo 9-9999-9999-99, může třeba docela klidně pracovat s nějakým novým, úžasným telefonním systémem.

### Délka

omezení délky dat – zejména pak délky znakových polí – jsou neustálým zdrojem problémů. I když jsme sebevíce velkorytí, můžeme si být téměř jisti, že se dříve nebo později objeví napro-

sto správný, platný údaj, který je ale příliš dlouhý a do pole se nevejde. Někdy můžeme tomuto problému předejít tak, že použijeme pole znakového typu (Character), alokujeme pro ně maximální možnou délku pole (tedy 255 znaků) a nastavíme u něj typ proměnné délky. V Microsoft Accessu mají přitom proměnnou délku všechna pole. V SQL Serveru musíme nastavit typ pole explicitně, a to na VARCHAR. Oba databázové stroje alokují v tomto případě prostor jen pro skutečně uložené znaky, takže se žádným místem neplýtvá.

Znaková pole s proměnnou délkou se ale pro všechny situace nehodí. Za prvé, někdy si údaj přímo vyžaduje jistou pevnou délku. Příkladem může být rodné číslo osoby, které má vždy délku 10 znaků. Pokud uživatel přesto zadá rodné číslo o 11 znacích, nepředstavuje omezení délky žádný problém; údaj je jednoduše neplatný. Povolit zápis takového údaje by nemělo žádný smysl. Za druhé, díky způsobu, jakým SQL Server zpracovává aktualizace záznamů se znakovými poli o proměnné délce, mohou tato pole vést k pomalejšímu výkonu systému. Ve většině případů je toto zpomalení prakticky nepostřehnutelné, avšak v aplikacích, kde je rychlosť zpracování životně důležitá a ve kterých se data často aktualizují, je lépe použít pole s pevnou délkou. (Při *přidávání* dat se rychlosť zpracování nijak neliší; jediný rozdíl spočívá v jejich *aktualizaci*.)

A nakonec, povolením velice dlouhých polí můžeme sice zlepšit použitelnost systému, někdy ji ale můžeme naopak dosti výrazně omezit. Formátování obrazovek a sestav může být díky dlouhým polím dosti ošklivé (zejména pak formátování sestav, protože v nich se data nedají rolovat) a vyhledávání záznamů, které obsahují jistou konkrétní informaci, může být přímo utrpením. Jestliže tedy není vhodné povolit u určitého údaje dlouhé hodnoty, pokuste se nadefinovat nějaká pravidla či konvence pro ošetření dat, která se do pole nevejdou. Jestliže je například skutečné jméno zákazníka delší, než je vyhrazený prostor, můžeme ve spolupráci s uživateli zavést jistá rozumná pravidla pro zkrácení jména; v angličtině vypustíme například členy, jako například „The“, slovo „společnost“ zkrátíme na „spol.“ a všechny znaky za slovem „společnost“ vypustíme. Takto zajistíme, že se jméno „Jedna opravdu, ale opravdu dlouhá společnost, s ručením omezeným“, bude zadávat vždy jako „Opravdu, opravdu dlouhá spol.“, a ne jednou jako „Opravdu, opravdu dlouhá s. r. o.“, a podruhé jako „Opravdu dlouhá společnost“.

## Hodnoty Null

Chybějící hodnoty jsou další oblastí, kde mohou mít uživatelé problémy s vestavěnými omezeními. O prázdných hodnotách Null jsme již dosti podrobně hovořili; při tom jsem se vám snažila vysvětlit, že je dobré hodnoty Null povolit tam, kde příslušná hodnota může být v reálném světě skutečně neznámá, nebo alespoň tam, kde taková data nebudou vůči systému zcela irrelevantní. Jestliže se ale rozhodnete se této dobře míněné rady nedržet, musíte navrhnut způsob, jakým uživatel pomůže ubohému uživateli, který se pokouší vytvořit nový záznam a ještě nemá k dispozici všechny požadované údaje.

Otázkou je, *když* jsou data potřeba. Z analýzy pracovních procesů systému víme, že před dokončením určitého daného úkolu musí být známy jisté údaje. To ale neznamená, že musí být hned při prvotním vytvoření záznamu známy *veškeré* údaje, které vyžadují *všechny* možné úkoly v systému. Před odesláním mzdly zaměstnanci musíme například znát číslo jeho bankovního účtu. Pole s informacemi o jeho bankovním spojení nesmí tedy být prázdné v den výplaty. Jen proto, že nový zaměstnanec nemá po ruce číslo svého bankovního účtu, nemá smysl bránit uživateli při prvotním pořízení záznamu v zadání ostatních informací o zaměstnanci, ani v provádění dalších úkonů.

Uživatel potřebuje třeba vytvořit služební průkaz či nějakou bezpečnostní kartu, bez které se nový zaměstnanec nedostane do budovy. Pro tento úkol informace o bankovním účtu zajisté nepotřebuje. Nakonec někdy informace o bankovním spojení potřebují uživateli určitou rozumnou implicitní hodnotu. Systém nemá důvod se za toto hněvat; časem se přece všechno srovná – konec konců o to se postará i sám zaměstnanec!

Jestliže se rozhodnete prázdné hodnoty Null v poli nepřipouštět, byť jen dočasně, je nejjednodušším způsobem ošetření uvedeného omezení dát uživateli určitou rozumnou implicitní hodnotu. Jednou z možností definice takové implicitní hodnoty je nadeklarovat příslušný údaj jako implicitní hodnotu v databázovém schématu. Druhou možností je nabídnout uživateli výběr z množiny hodnot – například „Neznámá“, „Nemá smysl“, nebo „Upřesní se později“. Někdy může systém vypočítat rozumnou implicitní hodnotu také za běhu, případ od případu.

## Intervaly

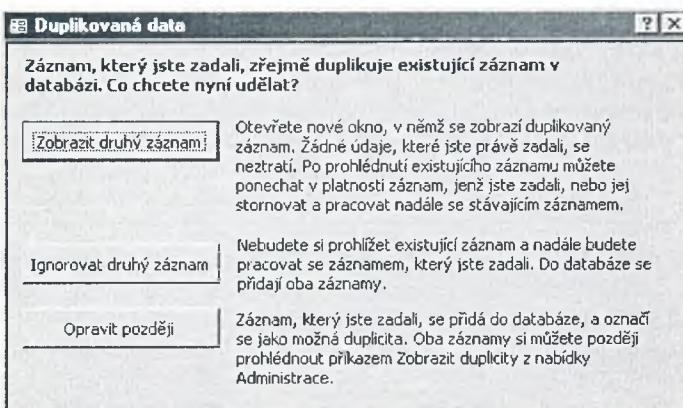
Definované intervaly představují – kromě povolení hodnot Null – další omezení na úrovni atributů, se kterými mají uživatelé často problémy. Některá omezení intervalu jsou přitom u konkrétního datového typu explicitní – do krátkého celého čísla se dá například uložit číselná hodnota nejvýše do 255. Jestliže je intervalové omezení přímo určeno datovým typem, nezbývá nám v podstatě nic jiného, než toto omezení vysvětlit uživatelům. Pokud se do pole musí ukládat i větší hodnoty, musíme v databázovém schématu určit jiný datový typ s širším intervalem povolených hodnot. Rozhodně nedoporučují pokusit se v aplikaci určovat datový typ za běhu.

Jestliže je přitom intervalové omezení v databázovém schématu definováno jako validační pravidlo nebo jako omezení typu CHECK, případně pokud není omezení implementováno v databázovém schématu, ale až v samotné aplikaci, nejedná se možná o vestavěné omezení, ale spíše o aplikativní pravidlo a my máme pro případné změny mnohem větší manévrovací prostor. O aplikativních pravidlech budeme hovořit podrobněji v další části výkladu.

## Omezení entitové a referenční integrity

Z předcházejících kapitol si zajisté vzpomínáte, že entitová omezení zajíšťují možnost jednoznačné identifikace každého jednotlivého záznamu tabulky, zatímco omezení referenční integrity zakazují záznamům odkazovat se ve stejném nebo v jiné tabulce na záznamy, které neexistují. Proto musíme pečlivě zvážit, jak entitová omezení a omezení referenční integrity zpracovat, aniž bychom uživatele zbytečně omezovali nebo na něj kladli zbytečné nároky. Entitová omezení a omezení referenční integrity se nejčastěji ošetří pomocí seznamu platných položek, ze kterého uživatel vybere konkrétní hodnotu. To ale, jak jsme si ukázali v poslední kapitole, není zcela vždy možné – nejčastěji proto, že by výsledný seznam byl příliš dlouhý, a tudíž prakticky obtížně použitelný. Jestliže omezení vstupu od uživatele pomocí seznamu není možné nebo není vhodné, musíme platnost dat ověřit co nejdříve po zadání a v případě problémů ihned upozornit uživatele. Záznam, který je podle všeho duplikátem existujícího záznamu v databázi, je typickým problémem entitové integrity. Nejlepší reakcí systému je nabídnout uživateli zobrazení tohoto stávajícího záznamu – nebo alespoň příslušných polí – a umožnit mu posouzení, jestli je nový záznam skutečně duplicitní; příklad upozornění ukazuje obrázek 16-1.

Při zobrazování stávajícího záznamu si ale dávejte pozor, abyste nepřepsali data, která uživatel právě zadal do nového záznamu. Existující záznam vypište proto do samostatného okna a rozhod-



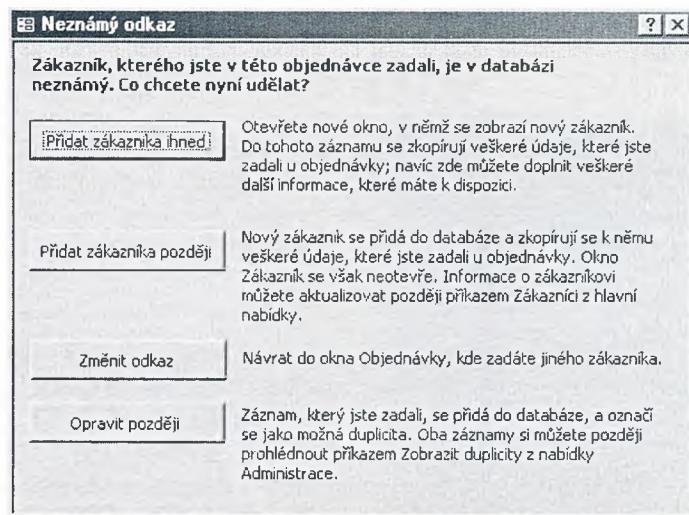
Obrázek 16-1 V tomto dialogu může uživatel rozhodnout, jestli je nový záznam skutečně duplicitní

nutí, jestli se má nový záznam přepsat, ponechejte na uživateli, jak to vidíme na obrázku 16-1. Všimněte si také, že v dialogu na obrázku 16-1 může uživatel pokračovat v zadávání dat i bez prohlédnutí podezřelého duplicitního záznamu. Uživatel již třeba o druhém záznamu mezičím ví; v takovém případě nemá smysl jej v práci vyrušovat více, než je absolutně nezbytně nutné. (Opakujte po mně: „Počítač to *neví* nejlépe, počítač to *neví*...“)

Dalším problémem je uživatel, který se pokouší ponechat prázdne některé z polí primárního klíče. Podmínka, podle níž nemůže pole primárního klíče obsahovat prázdnou hodnotu, je podle mého názoru tím nejlepším argumentem, proč jako primární klíč použít raději automatickou, systémem definovanou hodnotu. Jestliže za primární kľíč prohlásíme pole typu Automatické číslo (respektive v SQL Serveru pole typu Identity), může si uživatel dělat s přirozenými poli primárního klíče, co chce, aniž by ohrozil entitovou nebo referenční integritu. Pokud se pole typu Automatické číslo (respektive Identity) do daného systému nehodí, musíme zajistit, že uživatel zadá do každého pole primárního klíče nějakou hodnotu. Jednoduše do téhoto polí zkopirovat implicitní hodnotu zcela zřejmě nemůžeme, protože implicitní hodnota je vždy jedna stejná, a tudíž ji tabulka akceptuje jen jednou. Systém by *mohl* nějakou implicitní hodnotu vypočítat také za běhu, případ od případu, ale jestliže už takto uvažujeme o nějaké v podstatě libovolně zvolené hodnotě, můžeme klidně použít přímo pole typu Automatické číslo. Jedinou další možností je vyzádat si nějakou přípustnou hodnotu od uživatele.

Vznik problému s referenční integritou obvykle znamená, že se uživatel pokouší odkazovat se na záznam, který neexistuje. Takový nesprávný odkaz může vzniknout náhodou – uživatel zapíše například nesprávné jméno – nebo může být víceméně záměrný. Uživatel si třeba neuvědomí, že odkazovaný záznam neexistuje, nebo se ještě nedostal k zadání jeho dat. Jednu možnost ošetření této situace naznačuje dialog na obrázku 16-2. Zde má uživatel celkem čtyři možnosti: buďto systém přidá hned nový záznam a vyžádá si od uživatele okamžité zadání podrobných informací, nebo systém přidá nový záznam, který ale uživatel upraví později, nebo uživatel změní nesprávný odkaz, anebo odkaz opraví později.

Všimněte si, že uživateli se zde nenabízí možnost potlačení narušeného omezení. Toto omezení se totiž potlačit vůbec nedá, protože v takovém případě by byla ohrožena stabilita databáze, jak jsme si již ukázali.



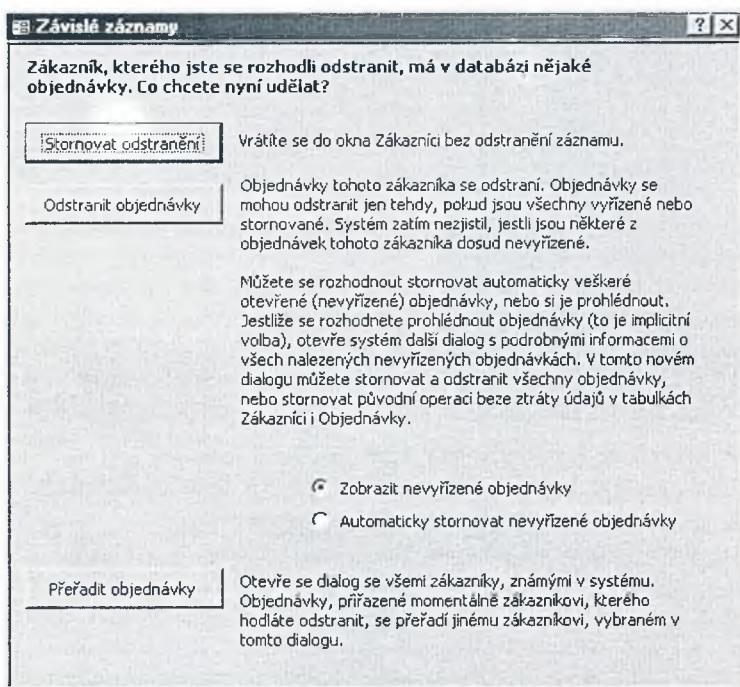
Obrázek 16-2 V tomto dialogu se uživatel může rozhodnout, jak má ošetřit odkazovaný záznam, který (zatím) neexistuje

Jestliže nemůžeme uživateli nabídnout v dialogu seznam platných hodnot, protože je jich příliš mnoho, můžeme v něm namísto toho zobrazit seznam položek, které zadanému údaji blízko odpovídají. Pro takové vyhledávání existuje řada různých algoritmů, od jednoduchého operátora LIKE jazyka SQL až po hledání SOUNDEX.

V některých situacích není nutné dialog s informacemi o validaci referenční integrity zobrazit vůbec. Ve skutečnosti je obvykle vhodné se tomuto zobrazení opravdu vyhnout, protože v podstatě uživatele ruší při práci. Jestliže se dál s rozumnou mírou jistoty předpokládat, že uživatel bude chtít vždy přidávat nové záznamy a podrobné informace upraví později, a pokud mu zároveň nabídnete odpovídající prostředky pro vrácení operací systému zpět (jestliže systém „tipuje“ nesprávně), může systém potichu přidat nový záznam na pozadí, bez zbytečného povyku. Jakmile uživatel zadávání záznamu dokončí, můžete například do stavového řádku vypsat zprávu o přidání záznamu, případně zprávu zobrazit v dialogu (pokud potřebujeme informovat uživatele podrobněji). Ať už se ale rozhodneme pro jakýkoli postup, nikdy nepřerušujte uživatele v zadávání záznamů, pokud to není nevyhnutelně nutné; je to vůči uživateli zbytečně neslušné a nepříjemné.

Druhý případ, kdy uživatel může narušit pravidla referenční integrity, nastává (kromě odkazu na záznam, který ve skutečnosti neexistuje) při pokusu o odstranění nebo změnu odkazovaného záznamu. Uživatel se například pokouší odstranit záznam zákazníka, který má nevyřízené objednávky, nebo změnit hodnotu pole Číslo výrobku, na nějž existuje odkaz v tabulce Rozpis objednávek.

Databázový stroj Microsoft Jet podporuje operace kaskádovité aktualizace a odstranění, které obsluhují aktualizace a odstraňování odkazovaných záznamů bez zásahu uživatele. V SQL Serveru můžeme stejně funkce implementovat pomocí spouští. Pokud má kaskádovité promítání operaci aktualizace a odstranění v dané aplikaci smysl, představují zdaleka nejideálnější řešení. Jestliže ale uživateli nabídnete možnost rozhodnout se o dalším postupu sám, musíme mu vysvětlit důsledky každé z variant, jak to vidíme na obrázku 16-3.



Obrázek 16-3 Tento dialog vysvěluje důsledky jednotlivých nabízených variant a umožňuje uživateli zvolit si další postup

Pomocí tohoto dialogu může uživatel operaci odstranění buďto stornovat, nebo ji promítnat v kaskádě do druhé tabulky, nebo přeřadit závislé záznamy k jinému zákazníkovi. Všimněte si, že uvedený dialog upozorňuje na to, že se otevřené (nedokončené) objednávky nedají odstranit (to je aplikační pravidlo) a nabízí mu možnost stornovat objednávky (v podstatě toto pravidlo obejít), nebo si je prohlédnout.

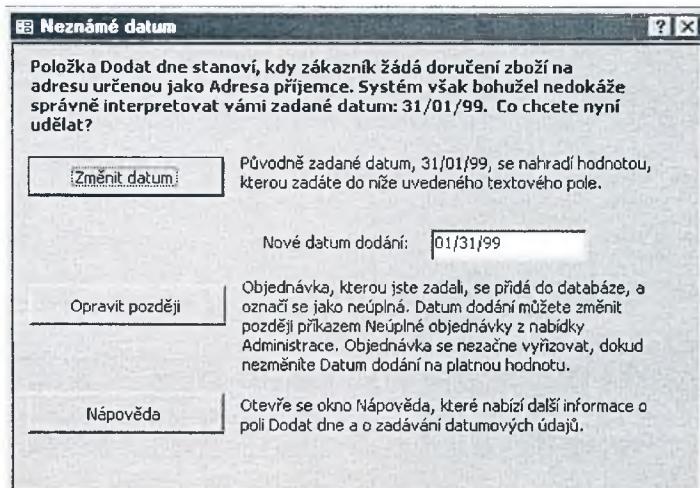
Z validačního dialogu můžeme také otevřít záznamy otevřených (rozpracovaných) objednávek, nebo dokonce všech objednávek daného zákazníka. Cílem je poskytnout uživateli vždy co nejvíce informací, aby se mohl rozhodnout opravdu kvalifikovaně na základě úplných informací.

## Aplikační omezení

V předcházející části textu jsme hovořili o vestavěných omezeních, jejichž úlohou je chránit strukturu a integritu databáze. S daty, která porušují některé vestavěné omezení, již uživateli v podstatě nemáme jak pomoci. Data musí vyhovovat požadavkům relačního modelu – alespoň nakonec musí jeho podmínky splňovat. Jestliže ale data narušují aplikační pravidlo (aplikativní omezení), je to úplně něco jiného.

Aplikační omezení (lépe se jim říká „aplikativní pravidla“) se neodvozují, jak jsem již řekla, ze samotného relačního modelu, ale z prostoru problému. Nezapomeňte, že datový model není nic jiného, než zjednodušený model určité části reálného světa. Jako návrháři systému se v modelu snažíme co nejvěrněji zachytit všechny aspekty daného prostoru problému, ale ani při nejlepší vůli se nám to nemusí vždy podařit. A i když je třeba model hned při své první implementaci ve všech ohledech dokonalý, mohou se aplikativní podmínky změnit za běhu. Systém, který má být úspěšný, musí být schopen elegantně zvládnout i neočekávané situace.

Obecně existují dva důvody, proč se uživatelé snaží do systému zadávat údaje, na které není připraven: buďto nevhodný údaj zadá nedopatřením (nahodile), anebo úmyslně, protože reálně vzniklá situace neodpovídá implementovanému modelu systému. (Ve skutečnosti jsou tři důvody, pokud ještě počítáme úmyslnou sabotáž; to je ale mimořádně *nepravděpodobný* případ a aplikativní pravidla v žádném případě nejsou vhodným místem pro jeho ošetření.)



Obrázek 16-4 Tento dialog vysvěluje chybu při zadávání dat bez nějakého ultimáta

### Nesprávné zadání z nedopatření

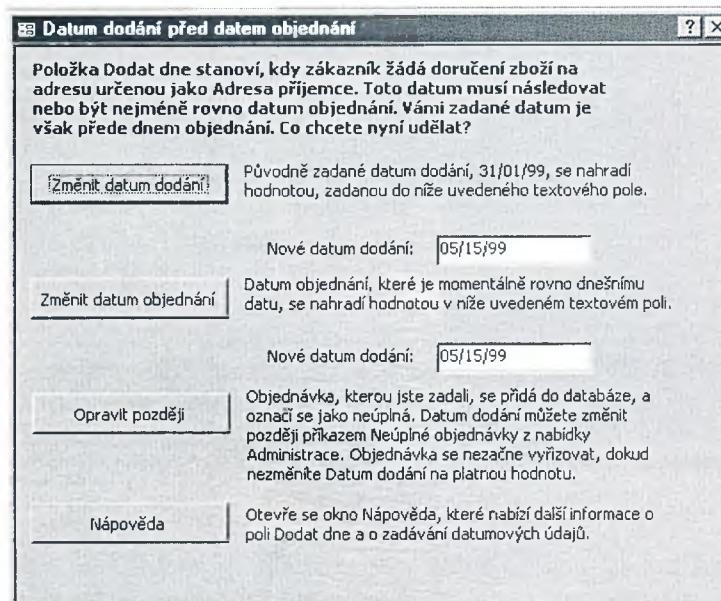
Jestliže uživatel zadá do systému neočekávaný údaj nahodile, nedopatřením, je naší jedinou stárostí co nejvíce mu usnadnit nápravu vzniklého problému. To přinejmenším znamená, že bychom měli uživateli přesně vysvětlit podstatu problému a říci mu, co s ním má udělat. Pokud možno, je třeba také nabídnout vhodné alternativy, které může systém přímo implementovat. Jestliže například uživatel zadá do pole datumu měsíc a den obráceně (zde uvažujeme americký způsob zadá-

vání data, pozn. překl.), může se vyvolat dialog uvedený na obrázku 16-4. Tento dialog se docela rozumně pokouší logicky odvodit, co chtěl uživatel správně zadat, a nabízí uživateli možnost zápisu správného údaje jediným klepnutím myši.

Jestliže uživatel zadá takto nahodile nesprávný údaj, může se jednat skutečně o omyl, nebo to znamená, že uživatel správně nechápe, co po něm systém vlastně chce. Dialog z obrázku 16-4 se proto snaží vysvětlit uživateli význam pole Datum odeslání, zároveň ale obsahuje i tlačítko Nápověda, po jehož klepnutí se uživateli dostane podrobnějších informací. O pomoc uživatelům budeme podrobněji hovořit v kapitole 18.

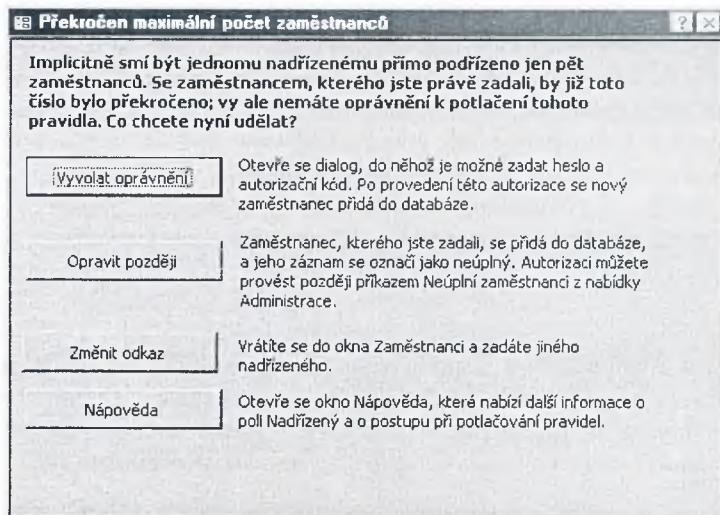
### Rozpor reality a modelu systému

Druhým důvodem, proč může uživatel zadávat do systému neočekávané údaje, je skutečně vzniklá reálná situace, která neodpovídá navrženému modelu systému. Uživatel se například pokouší napravit situaci, kdy je například objednávka již vyřízena a odeslána, ale dosud nebyla zadána do systému. Datum odeslání této objednávky bude dřívější než datum pořízení objednávky – zde tedy dochází k porušení aplikačního pravidla. Jestliže bychom uživateli nenabídli nějakou rozumnou radu, zadal by jakékoli nesmyslné datum odeslání, jaké by byl systém schopen akceptovat; tím by ale došlo k ohrožení integrity databáze. Je tedy dobré pokusit se vymyslet co nejvíce různých výjimek, na jaké může systém narazit, a uživatelům pomoci je konzistentně vyřešit. Na obrázku 16-5 vidíme, jak může systém reagovat na zadání Data odeslání, které je starší než zapsané Datum objednávky. Tento příklad zároveň předpokládá, že implicitní hodnotou Data objednávky je aktuální systémové datum a že uživatel nemůže toto pole obvykle editovat.



Obrázek 16-5 Tento dialog umožňuje uživateli zadání objednávky se zpětnou platností

Takto jednoduše potlačit či „obejít“ se ale nedají všechna aplikační pravidla (nebo to alespoň není vhodné). Některá aplikační pravidla se skutečně potlačit nedají – například proto, že odpovídající situace nemůže reálně nastat, nebo z důvodu jistých zákonného omezení. Někdy ale nestojí otázka, „zdali“ je možné aplikační pravidlo porušit, ale „kdo“ je může porušit. Na obrázku 16-6 se uživatel například pokusil zadat informace o šestém zaměstnanci, který je přímým podřízeným jednoho stejného vedoucího, takže porušil aplikační pravidlo, podle něhož nesmí mít žádný vedoucí pracovník pod sebou více než pět přímých podřízených.



Obrázek 16-6 Tento dialog říká, že aplikační pravidlo smí porušit jen osoba s odpovídajícím oprávněním

Aktuální uživatel zde nemá oprávnění k potlačení pravidla. Systém ale nabízí sekundární dialog, do něhož může někdo jiný s odpovídajícím oprávněním – zřejmě vedoucí nebo manažer – zadat potřebné heslo a autorizační kód. Protože lidé s takovým oprávněním nemusí být „na zavolání“ vždy po ruce, nabízí první dialog uživateli také možnost uložení aktuálního záznamu a podává další instrukce, podle kterých se dá autorizační kód zadat později.

Možnost potlačení aplikačních pravidel může výrazně zlepšit použitelnost systému, avšak za cenu složitějšího datového modelu. V našem posledním příkladu musíme do tabulky Zaměstnanci přidat speciální pole s autorizačním kódem a do odpovídajícího omezení integrity zahrnout odkaz na tento kód. Dále musíme vytvořit mechanismus, kterým při odložení autorizace na pozdější dobu prohlásíme záznam za neplatný. První možnost je doplnit do tabulky Zaměstnanci jedno pole typu Boolean, které indikuje, že záznam (zatím) není platný. Podle tohoto příznaku můžeme později, kdy je již uživatel připraven, vyřešit nevyřízené problémy, vyhledat neplatné záznamy, nebo při tvorbě sestav záznamy filtrovat. Někdy je také užitečné vědět, z jakého důvodu je záznam neplatný; v takovém případě doplníme pole s kódem – například „SV“ znamená, že chybí „souhlas vedoucího“, nebo „KU“ si žádá „kontrolu úvěru“. Tento druhý postup nabízí při zpracování záznamů vyšší flexibilitu. Jestliže ale uvedená pole s kódem mohou pro libovolný záznam nabývat příliš velkého

počtu různých hodnot, může být toto řešení pro uživatele složité; pak nezbývá než přesunout kódy možných problémů do druhé tabulky a s první tabulkou ji svázat vztahem typu jedna k více.

Změny potřebné pro prohlášení záznamů za neplatné se mohou zdát na první pohled docela jasné, a z velké části jsou opravdu jasné; na druhé straně se ale informace o neplatných záznamech mohou v systému rozšířit dosti neočekávanými způsoby. Mají se neplatné záznamy zahrnovat například do různých sestav? O zpracování těchto sestav se můžeme rozhodovat podle polí s kódem neplatnosti, protože jen z něj se dozvím, které záznamy se mají z určité sestavy či dotazu vypustit.

Jestliže neplatné záznamy do sestav a dotazů *zahrneme*, musíme zároveň uvážit, jestli se mají nějakým způsobem také označit jako nespolehlivé, například jestli se mají seskupit pod samostatné záhlaví. Pokud se naopak rozhodneme, že neplatné záznamy, nebo alespoň neplatné záznamy určitého typu, do sestavy zahrnovat nebude, musíme se pak také rozhodnout, jestli se mají ze sestav vyloučit také případné s nimi asociované záznamy. Jestliže například nastane výše uvedený problém s nepotvrzeným záznamem šestého zaměstnance, má se ze sestav vyloučit jen záznam tohoto jednoho zaměstnance, nebo vůbec veškeré záznamy podřízené stejněmu vedoucímu? Všechny tyto otázky musíme tedy zvážit ještě před tím, než v systému vůbec povolíme možnost potlačení aplikačního pravidla. Tato možnost sice zlepšuje použitelnost systému, zvýšená pracnost složitějšího systému však za to třeba nestojí. V jednoduchých systémech může být lepší takový záznam, který nevyhovuje aplikačnímu pravidlu, zcela odmítnout a nechat výjimku ošetřit lidem vně počítačového systému. Pokud se ovšem rozhodnete pro toto řešení a ponecháte ošetření výjimečných případů mimo navrhovaný systém, nezapomeňte to opět uživateli říci v dialogu pro validaci dat. Nenechávejte uživatele bezradně na pochybách a napovězte mu, jak postupovat dále.

### **Stručné shrnutí**

V této kapitole jsem zaujala na první pohled neobvyklý, zcela jasně neortodoxní postoj, podle kterého databázové systémy nemusí nutně zajišťovat omezení databázové integrity. Zabývali jsme se zde dvěma typy omezení: prvním jsou vestavěná omezení, která ovládají fyzickou strukturu dat a odvozují se přímo z relačního modelu, zatímco druhou skupinu tvoří aplikační omezení, která vycházejí ze zadlého prostoru problému. Ukázali jsme si, že povolit uživateli potlačení vestavěných omezení je nebezpečné, zatímco u aplikačních omezení to často má smysl – záznamy, které aplikačním omezením nevyhovují, můžeme označit za neplatné a vyřešit je později. Podívali jsme se také na několik příkladů možného prohlášení záznamů za neplatné. V následující kapitole si ukážeme různé aspekty vypisování výstupních dat, jejichž správu systém zajišťuje.

Z faktů uložených v databázovém systému se stávají informace až poté, co je zkombinujeme určitým smysluplným způsobem; do té doby jakoby žádný velký smysl neměly. V této kapitole se tedy podíváme na různé otázky spojené s podáváním těchto smysluplných kombinací faktů – neboť informací – uživatelům.

## Poznámka

*Pod pojmem „vytváření sestav“ (možná by bylo lépe říci něco jako „vypisování dat“) v této kapitole nemyslím jen vytváření tištěných sestav. Toto slovo zde používám v obecnějším slova smyslu, a sice ve smyslu jakéhokoli poskytování informací, které vycházejí z dat uložených v databázi. Tyto podávané informace mohou mít podobu tištěné sestavy, ale zrovna tak se mohou vypisovat jen ve formuláři nebo v množině záznamů, zobrazené jako datový list.*

Kdysi klávno, kdy počítače stály mnohem více než zaměstnanci a kdy strojový čas počítačů byl mimořádně vzácným a cenným prostředkem, produkovaly databázové systémy v pravidelných intervalech velice málo sestav. Jestliže jste ze systému potřebovali cokoli speciálního, mohli jste si o tom leda nechat zdát. Běžné dodací lhůty v průměrném výpočetním středisku dosahovaly až několika let, takže pokud jste potřebovali sestavu „přesně jako je tahle, jenom aby byla seřazena podle Zákazníků a posílaná podle Státu“, bylo jednodušší zadat to jako úkol sekretářce, která ji sepsala na stroji ručně.

Dnes jsou již počítače naopak velice laciné, jsou doslova všude, a velice vzácným a cenným prostředkem jsou naopak sekretářky. Uživatel musí mít proto možnost říci přímo počítačovému systému: „Chci zrovna takovou sestavu, jako je tahle, ale...“, a to znamená, že naše pozice jako databázového návrháře je o něco obtížnější. Ale přece je to lepší, než dávat nějaké těsnopisné diktáty, ne?

K funkcím pro tvorbu sestav můžeme v databázovém systému přistupovat jedním ze dvou možných způsobů: buďto se pokusíme všechny možné sestavy odhadnout dopředu, anebo uživateli umožníme jejich vytváření na vyžádání, podle potřeby. Většina systémů vyžaduje přitom jistou kombinaci obou těchto přístupů.

Sestavy, které se dají předvídat a očekávat již při návrhu systému, můžeme vytvořit v průběhu implementace. Těmto typům sestav říkám *standardní sestavy*. Po dokončení implementace systému můžeme dále uživatelům nabídnout určitý mechanismus pro vytváření jejich vlastních sestav. Ty budu nazývat *ad hoc sestavy*. V této kapitole budeme hovořit o obou uvedených typech sestav, ze všeho nejdříve se ale podíváme na mechanismy pro třídění, vyhledávání a filtrování dat, které budeme při vytváření sestav potřebovat.

## Třídění, vyhledávání a filtrování dat

Při zapisování příkazů jazyka SQL je třídění, vyhledávání a filtrování dat docela jednoduché: stačí odpovídající kritéria zapsat do klausule WHERE nebo do klausule ORDER BY. Jestliže ale chceme tuto možnost nabídnout i uživateli, musíme mezi ním a logikou příkazu SELECT jazyka SQL vytvořit určitý „nárazník“.

Problém zde ale spočívá v tom, že logika jazyka SQL dosti špatně odpovídá logice přirozeného jazyka. Jestliže si manažer prodeje vyžádá například seznam všech distributorů své společnosti, kterí jsou ze státu Wyoming a Florida, pak po právu očekává, že výsledný seznam bude obsahovat distributory ze státu Wyoming a distributory ze státu Florida. Do odpovídajícího příkazu SELECT však v jazyce SQL musíme zapsat tuto klausuli: WHERE Region = „Wyoming“ OR Region = „Florida“ – ano, operátor je zde OR, „nebo“. Jazykově je sice správné říci „a“, ve formální logice je však správným výrazem spojka „nebo“. To je pro uživatele dosti velký problém – jazyk SQL je totiž natolik blízký přirozenému (anglickému) jazyku, že je zároveň beznadějně matoucí.

Samozřejmě že můžeme naučit uživatele, jak v příkazu SELECT jazyka SQL vytvářet výběrová kritéria. Opravdu vám alespoň o jednom velice uznávaném návrháři databází, který to skutečně pravidelně dělá, a podle všeho velice úspěšně. Osobně si ale myslím, že je lépe nabídnout více intuitivní rozhraní a ušetřit tak jak sebe sama, tak i tvůrce uživatelské dokumentace a (co je nejdůležitější) samotné uživatele od horkých chvil spojených s přímým používáním jazyka SQL.

Na tento úkol ale naštěstí nejsme sami. Microsoft Access nabízí příklady rozhraní pro konstrukci klausul jazyka SQL s kritérií. Zkopírovat některý z těchto mechanismů uživatelského rozhraní přímo do navrhovaného systému sice úplně vždycky dost dobře nejde, jsou však alespoň dobrým místem pro začátek; přesně tak je využívám i já. Rozhraní Microsoft Visual Basicu sice žádný z těchto mechanismů nepodporuje, při troše snahy se zde ale většina z nich dá implementovat v programovém kódu.

### Třídění dat

Microsoft Access nabízí vynikající rozhraní pro třídění dat, které představují příkazy Seřadit vzestupně a Seřadit sestupně. Uživatel klepne myší na ovládací prvek, podle něhož chce data seřadit, a poté vybere odpovídající příkaz z nabídky Záznamy, nebo klepne na tlačítko panelu nástrojů. Sotva bychom si asi dokázali vymyslet nějaké jednodušší rozhraní.

### Filtrování podle výběru

Co se týče filtrování dat, je nejjednodušším rozhraním v Microsoft Accessu příkaz Filtrovat podle výběru a jeho „nerozlučný společník“ Filtrovat mimo výběr. Rozhraní těchto dvou příkazů se do značné míry podobá příkazům Seřadit vzestupně a Seřadit sestupně. Jestliže uživatel buďto vybere

obsah pole na formuláři, nebo do něj umístí kurzor a poté zvolí příkaz Filtrovat podle výběru, od-filtruje Microsoft Access v podkladové množině záznamů daného formuláře jen ty záznamy, jejichž hodnota se přesně shoduje s hodnotou vybraného pole. Jinými slovy, sestaví podmítku WHERE jazyka SQL ve tvaru WHERE <jméno pole> = <hodnota ovládacího prvku>.

Jestliže uživatel vybere jen začátek obsahu pole, omezí Access množinu záznamů na ty záznamy, jejichž hodnota příslušného pole začíná vybranými znaky. Pokud tedy například uživatel vybere první tři znaky názvu výrobku „Chartreuse verte“, bude výběr ekvivalentní zápisu následující klauzule WHERE jazyka SQL: WHERE [Název výrobku] LIKE „Cha\*“. V ukázkové databázi Northwind vrátí přitom tento dotaz záznamy, které obsahují výrobky názvů Chartreuse verte, Chai a Chang.

Vybere-li uživatel v poli jakékoli jiné znaky, vrátí Microsoft Access všechny záznamy, které obsahují vybrané znaky kdekoli v tomto poli. To znamená, že pokud uživatel vybere ve výše uvedeném příkladu znaky „ar“, bude ekvivalentní klausule WHERE jazyka SQL vypadat takto: WHERE [Název výrobku] LIKE „\*ar\*“. Po spuštění v ukázkové databázi Northwind vrátí tento dotaz celkem 10 záznamů, které vidíme na obrázku 17-1.

	Cislo výrobku	Název výrobku	Dodavatel
+	24	Guaraná Fantástica	Refrescos Americanas LTDA
+	58	Escargots de Bourgogne	Escargots Nouveaux
+	73	Röd Kaviar	Svensk Sjöföda AB
+	39	Chartreuse verte	Aux joyeux ecclésiastiques
+	7	Uncle Bob's Organic Dried Pears	Grandma Kelly's Homestead
+	32	Mascarpone Fabioli	Formaggi Fortini s.r.l.
+	72	Mozzarella di Giovanni	Formaggi Fortini s.r.l.
+	62	Tarte au sucre	Forêts d'érables
+	18	Carnarvon Tigers	Pavlova, Ltd.
+	20	Sir Rodney's Marmalade	Specialty Biscuits, Ltd.
*	(Automatické číslo)		

Obrázek 17-1 Tyto záznamy se vrátí v případě, že uživatel vybere v poli Název výrobku formuláře, případně na datovém listu, znaky „ar“ a zadá příkaz Filtrovat podle výběru

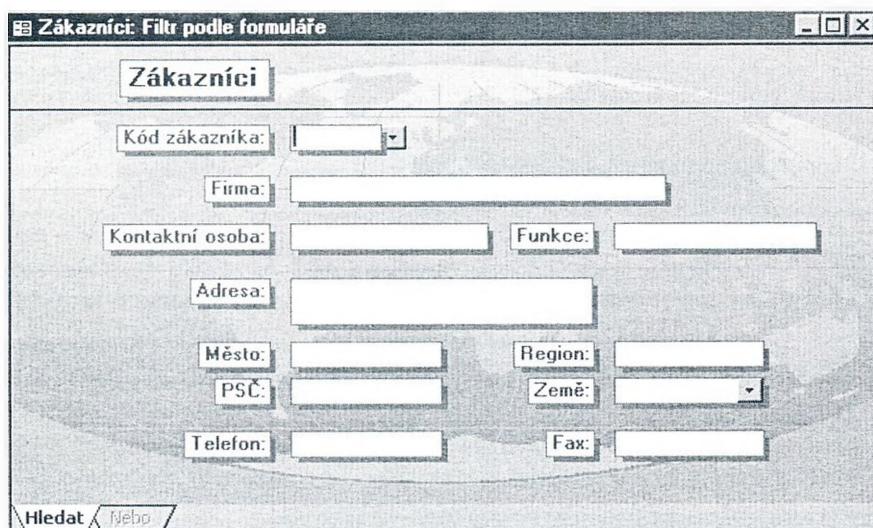
Poznámky

Při zvolení příkazu *Filtrovat podle výběru* však nemůžeme počítat s tím, že by Microsoft Access skutečně vykonával příkaz *SELECT* jazyka SQL. Osobně se domnívám, že tomu tak skutečně je, zaručit se to ale nedá. Asi nikdo nevidí do toho, čeho všechno jsou oni „eskamotéři“ ve vývojovém týmu Microsoft Accessu schopni.

## Filtrování podle formuláře

Příkaz Filtrovat podle výběru je jednoduché, elegantní rozhraní, které se uživatel velice snadno naučí; zároveň je ale omezeno jen na filtrování podle jediného pole. Uživatel může dalšími vhodně zvolenými příkazy Filtrovat podle výběru datovou množinu dále zúžit, takový postup je ale poměrně těžkopádný. Pro uživatele, kteří požadují rozhraní pro filtrování s bohatšími možnostmi, nabízí Microsoft Access příkaz Filtrovat podle formuláře. Obrázek 17-2 ukazuje formulář Zákazníci z ukázkové databáze Northwind poté, co v něm uživatel zvolil příkaz Filtrovat podle formuláře z nabídky Záznamy.

Pomocí okna Filtrovat podle formuláře může uživatel nastavit filtrované hodnoty i pro několik různých polí na několika záložkách původního okna. Hodnoty na záložce Hledat se spojí po-



Obrázek 17-2 Příkaz Filtrovat podle formuláře umožňuje uživateli filtrování informací podle více různých polí

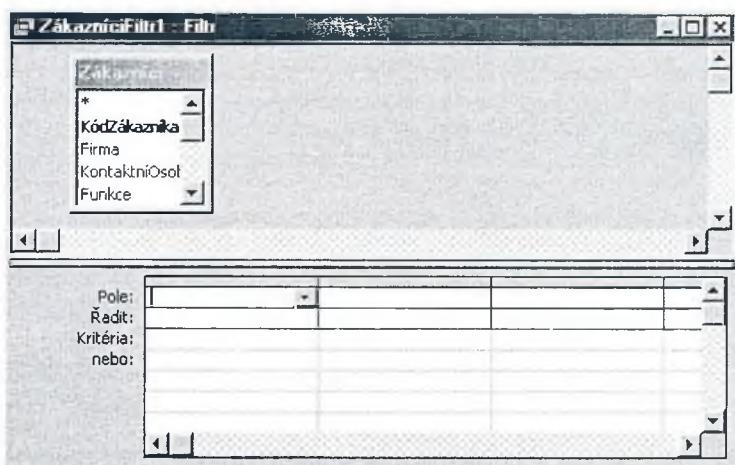
mocí logického operátora AND, zatímco hodnoty uvedené na záložce Nebo se propojí logickým operátorem OR. Ani při tomto postupu jsme se tedy nezbavili onoho rozporu mezi jazykovým a formálním logickým významem spojek „A“ (AND) a „NEBO“ (OR), obě pojetí se však již postupně sbližují.

Implicitně se v okně Filtrovat podle formuláře zobrazuje každé textové pole zdrojového formuláře jako kombinované pole, které obsahuje všechny aktuální hodnoty daného pole. Toto chování můžeme vypnout, jestliže do vlastnosti Vyhledání filtru ovládacího prvku textového pole přiřadíme hodnotu Nikdy. Jestliže okno Filtrovat podle formuláře zobrazuje kombinované pole, musí se hodnota v nalezených odpovídajících záznamech přesně shodovat s hodnotou, kterou zde uživatel vybral. Pokud se v daném místě okna nachází normální textové pole – tedy pokud jsme do vlastnosti Vyhledání filtru ovládacího prvku textového pole přiřadili hodnotu Nikdy – může uživatel buďto zadat hodnotu, ke které se bude hledat přesná shoda, nebo může zapsat výraz jako například LIKE „CHA\*“ nebo IS NOT NULL. Konkrétní rozhodnutí, jestli se v okně Filtrovat podle for-

můláře mají zobrazit kombinovaná pole, záleží přitom na velikosti podkladové množiny záznamů (uživatel nejspíše nebude chtít čekat, než Microsoft Access naplní kombinované pole se 100 000 položkami) a na flexibilitě, kterou uživatelé od systému požadují.

## Pokročilé filtrování a třídění

Příkaz Filtrovat podle formuláře, respektive rozhraní na něm postavené, pro většinu situací plně postačuje. Jestliže ale naši uživatelé dobře znají návrhář dotazů v Microsoft Accessu, případně jestliže uživatelům pomocí vlastnosti Vyhledání filtru umožňujeme v okně Filtrovat podle formuláře výběr hodnot z kombinovaného pole, a zároveň jim chceme nabídnout možnost zadávat do kritérií výrazy, bude pro nás zřejmě zajímavé okno Microsoft Accessu Rozšířený filtr či řazení, které vidíme na obrázku 17-3.



Obrázek 17-3 Toto okno Microsoft Access zobrazí, jestliže uživatel vybere příkaz Rozšířený filtr či řazení z otevřeného formuláře Zákazníci v ukázkové databázi Northwind

Okno Rozšířený filtr či řazení nabízí určitou podmnožinu z funkcí, které známe z návrhového zobrazení dotazu: umožňuje tak fakticky pouze zadávání klausulí WHERE a ORDER BY příkazu SELECT. To znamená, že v tomto okně nemá uživatel možnost změnit základní strukturu navrácené množiny záznamů, protože nemůže změnit seznam polí, ani připojit další množiny záznamů.

Okno Rozšířený filtr či řazení je jedním z rozhraní pro filtrování, o jejichž napodobení bych se ve Visual Basicu nepokoušela. Udělat by se to pochopitelně dalo, zdaleka by to ale nebylo jednoduché. Proto jestliže tyto funkce v navrhovaném systému nezbytně potřebujete, vyberte si raději za vývojový nástroj Microsoft Access, případně si najděte vhodný produkt od některé z třetích stran, který do systému zapojíte.

## Microsoft English Query

Jestliže si za databázový stroj navrhovaného systému zvolíte Microsoft SQL Server, je vhodné uvažovat o implementaci Microsoft English Query. Tento nástroj nabízí nejen rozhraní pro třídění a filtrování, ale také zejména plnohodnotné rozhraní k databázi v přirozeném (anglickém) jazyce.

Při implementaci Microsoft English Query v konkrétní aplikaci musíme nadefinovat mapování jazyku našeho prostoru problému na databázové schéma a vytvořit něco, čemu Microsoft English Query říká „aplikační soubor“. Vytvoření tohoto aplikačního souboru není nijak zvlášť obtížné, na druhé straně to ale není ani příliš triviální. Podobně jako při vytvoření dobrého rejstříku k souboru nápovědy to znamená strávit spoustu času vymýšlením slov, kterými se uživatelé budou nejspíše snažit pojmenovat různé entity a atributy našeho databázového schématu.

Po vytvoření aplikačního souboru je již integrace Microsoft English Query do databázové aplikace docela snadná. Naše databázová aplikace předá jednoduše dotaz (lépe řečeno otázku), zadáný uživatelem v přirozeném anglickém jazyce, ke zpracování do stroje Microsoft English Query a jako výsledek dostane odpovídající příkaz jazyka SQL. Tedy, taková je alespoň teorie. Ve skutečnosti může aplikace dostat nazpět namísto příkazu SQL chybovou zprávu, která říká, že uživatel (to je to stvoření s neuvěřitelnou fantazií, jakou mohou mít jen uživatelé) zformuloval otázku takovým způsobem, že mu stroj nerozumí.

Ve správném prostředí je Microsoft English Query skutečně vynikajícím nástrojem. Pokud má systém velice složité databázové schéma a pokud s ním bude pracovat velké množství začínajících uživatelů, kteří mu budou klást různé dotazy, může být rozhraní přirozeného jazyka, jaké poskytuje Microsoft English Query, opravdu ideálním řešením.

## Vytváření standardních sestav

Téměř v každém databázovém systému najdeme určitý počet sestav, které můžeme nadefinovat již dopředu. Většinu těchto standardních sestav odhalíme již v průběhu analýzy pracovních procesů, přesto však stojí za to sednout si na chvíli s uživateli nad databázové schéma a podívat se, jestli by pro ně neměly smysl i jiné sestavy.

### Přehledové sestavy a podrobné sestavy

Pro každou entitu v systému je vhodné nadefinovat jak přehledovou sestavu, tak i podrobnou sestavu. Přehledová sestava je v podstatě prostý seznam všech instancí dané entity – neboli seznam všech záznamů v tabulce. Někdy stačí tento seznam jednoduše uspořádat v abecedním pořadí. Častěji je ale musíme určitým způsobem seskupit. Seznam zákazníků můžeme například seskupit podle státu, regionu nebo přiděleného obchodního zástupce. Jestliže se přitom dá očekávat, že zdrojová tabulka dané sestavy bude mít více než několik stovek záznamů, je dálé vhodné nabídnout uživateli možnost tisku jen vybrané podmnožiny záznamů. Obchodní zástupce si bude nejspíše chtít prohlížet pouze seznam jeho vlastních zákazníků, ne seznam všech zákazníků v systému.

Zatímco přehledová sestava (výpis) zobrazuje pro každou instanci dané entity jen několik málo informací, podrobná sestava ukazuje oproti tomu všechny podrobné informace o konkrétní entitě (nebo alespoň většinu z nich). Opět je přitom obvykle vhodné nabídnout uživateli nějakou metodu pro výběr záznamů, které se mají tisknout. Vhodným mechanismem pro

tento výběr je velice často seznam s možností vícenásobného výběru, protože v něm uživatel není omezen jen na spojité výběry.

Mějte ale na paměti ona praktická omezení použitelnosti seznamů. Jestliže určitá tabulka obsahuje několik tisíc záznamů, musíme uživateli nabídnout sadu několika ovládacích prvků typu seznam, pomocí kterých progresivně zúží množinu záznamů. Druhá možnost je použít namísto seznamu ovládací prvek jiného typu. Můžeme například vytvořit textové pole s podobnou funkcí, jako je například textové pole, které v dialogu Tisk aplikace Microsoft Word definuje rozsah tisku. S takovým polem musíme samozřejmě v aplikaci implementovat také rozbor intervalu záznamů, jejichž počátek a konec vymezují pomlčky, respektive jednotlivých záznamů oddělených čárkami, to ale není nikterak obtížné.

## Souhrnné sestavy

Druhý typ sestav, které bychom mohli označit za sestavy typu „rozděl a panuj“ a které obsahují různým způsobem zkombinované a vzájemně porovnané souhrnné údaje, je na implementaci o něco složitější, pro uživatele je ale často také mnohem užitečnější. Příkladem tohoto typu sestav je procentní rozdělení prodeje podle jednotlivých regionů či prodejců, nebo počet zákazníků, kteří si zakoupili výrobky každé z kategorií.

Souhrnné sestavy se také velice dobře hodí pro grafickou reprezentaci; grafické sestavy se přitom velice snadno implementují například v nástroji Microsoft Graph, případně v různých nástrojích od třetích stran. Osobně ale doporučuji podávat grafické informace společně s textovými daty jako jejich doplněk, nikoli jako náhradu místo nich. Souhrnná sestava o prodeji v textové podobě se asi nebude být vždy pohodlně prohlížet, na druhé straně ji ale můžeme velice snadno exportovat do vhodného nástroje pro statistickou analýzu nebo do tabulkového procesoru, jako je například Microsoft Excel, a zde s daty dálé manipulovat.

## Sestavy založené na formuláři

Kromě sestav odvozených z databázového schématu jsou dobrým zdrojem řady užitečných sestav také různé formuláře v systému. Osobně považuji možnost tisku většiny formulářů v každém databázovém systému za samozřejmost. Implementovat tisk formuláře je pro nás jako vývojáře docela snadné a pro uživatele, který si potřebuje výsledky své práce rychle vytisknout a zkontrolovat nebo je ukázat někomu jinému, jsou mimořádně užitečné.

Někdy stejně informace jako formulář obsahuje odpovídající podrobná sestava dané entity; v takovém případě nemusíme vytvářet novou formulářovou sestavu, ale můžeme přímo využít stávající podrobnou sestavu. Nejčastěji ale podrobná sestava entity obsahuje navíc určité doplňující informace, nebo jsou v ní některé informace formátovány jiným způsobem než na formulářové sestavě. Vytvořit sestavu založenou na formuláři je ale natolik snadné a laciné, že obvykle nabízíme uživatelům jak normální podrobnou sestavu, tak i formulářovou sestavu. Sestava založená na formuláři se vytiskne implicitně po klepnutí na tlačítko tisku v panelu nástrojů, zatímco k podrobné sestavě se dostane prostřednictvím vhodného příkazu nabídky (případně také zvláštním tlačítkem na panelu nástrojů).

## Rozhraní pro spouštění sestav

Nabídnout do uživatelského rozhraní sestavy není nijak složité. Příkaz Vytisknout sestavu můžeme uživateli dát k dispozici celkem třemi různými způsoby: prostřednictvím příkazu nabídky, tlačítkem z panelu nástrojů, nebo příkazovým tlačítkem na formuláři. Z příkazu pro tisk sestavy musí být ale uživateli vždy jasné, kterou sestavu bude tisknout.

V nabídce se informace o tisknuté sestavě vypíše velice snadno: jméno sestavy použijeme přímo jako položku nabídky Sestavy. Jméno sestavy můžeme zapsat také do bublinové nápovědy (ToolTip) tlačítka na panelu nástrojů, respektive do textového popisku příkazového tlačítka, zde je ale vhodné napsat před jménem sestavy ještě sloveso „Vytisknout“ nebo slovo „Tisk“. Jako položka nabídky Sestavy je například výraz „Seznam zákazníků“ přijatelný, v odpovídající bublinové nápovědě na panelu nástrojů či v titulku příkazového tlačítka by ale měl být text „Vytisknout Seznam zákazníků“. Tento postup doporučují zásady pro tvorbu rozhraní v systému Windows, z textu je ale zároveň uživateli jasné, že systém sestavu přímo vytiskne a že ji třeba neotevře v běžném okně.

Kromě toho, *kde* má být sestava v systému pro uživatele dostupná (tedy jestli to má být v nabídce, panelu nástrojů nebo v příkazovém tlačítku), musíme také zvážit, *jak* bude dostupná. Databázový systém obsahuje často desítky a někdy i stovky sestav, a uvedení takového počtu položek v jedné obrovské nabídce o mnoha sloupcích je zcela zřejmě nerozumné. Seznam všech dostupných sestav se ale naštěstí dá obvykle velice snadno omezit jen na ty z nich, které mají v aktuálním kontextu daného uživatele smysl. Uživatel si nejspíše nebude chtít někde uprostřed pořizování objednávek prodeje tisknout třeba telefonní seznam zaměstnanců.

Jestliže si přesto myslíte, že má v systému smysl nabídnout uživateli všechny sestavy najednou, můžete vytvořit dialog, v němž budou sestavy rozčleněny do kategorií a z něhož si uživatel může vybrat tisk libovolné z nich. Tento postup je vhodný také v případě, že uživatelé budou potřebovat tisknout i několik sestav najednou. Pokud jim umožníme v takovém dialogu vybrat současně libovolný počet sestav, stačí jim na tlačítko tisku klepnout jen jednou a dále se již mohou věnovat své práci.

U sestav, které se často tisknou dohromady v jedné dálvce – jako jsou například různé měsíční přehledy a uzávěrky – používám určitou variantu uvedené metody s dialogem. Nabídnou tedy uživateli dialog, v němž jsou implicitně vybrány všechny sestavy, které se obvykle v dálvce tisknou. Nyní může uživatel k tomuto seznamu doplnit další sestavy, jež tiskne jen příležitostně, případně některou ze standardních sestav vypustit, a teprve poté odešle celou dálvku na tiskárnu.

Uvedení každé jednotlivé sestavy v dálvce znamená – oproti tisku dálvky v jediném příkazu nabídky – pro uživatele jistou režii navíc, z praxe jsem ale zjistila, že tato zvýšená flexibilita uživatelům rozhodně stojí za to jedno klepnutí myší navíc. Často se stává, že se určitá jedna sestava z dálvky musí vytisknout znova, například kvůli chybě tiskárny, nebo jednoduše proto, že na ni někdo rozlil kávu. Pokud jsou všechny sestavy uvedeny v jednom společném dialogu, kde je můžeme tisknout nejen v dálvce, ale i jednotlivě, nemusíme kvůli takovým případům zavádět pro každou sestavu zvláštní položku nabídky, nebo dokonce – což je ještě horší – tisknout kvůli problémům v jedné sestavě celou dálvku znova.

## Ošetření chyb tiskárny

Systém musí v každém případě umět vhodným způsobem ošetřit případné chyby tiskárny nebo tišku sestavy; některé z poměrně běžných situací při tisku se tím pádem mohou stát dosti komplikovanými. Uživatel bude chtít například vytisknout všechny faktury, které dosud nebyly vytiskeny. Tento docela běžný požadavek je zároveň pro systém jedním z nej obtížnější zvládnutelných, protože systém neví, jestli byla ta která konkrétní sestava skutečně vytiskena. Systém pouze ví, že sestavu odeskal do tiskové fronty, a to pochopitelně zdaleka není totéž.

Některí návrháři systémů ošetrují možné chyby tiskárny pomocí dialogu, který zobrazí po odeslání sestav na tiskárnu a ve kterém žádají uživatele o potvrzení úspěšného vytisknutí sestav. Tato metoda samozřejmě funguje, pro uživatele to ale znamená přerušit práci až do okamžiku, kdy má vytisknou sestavu fyzicky v ruce. A jestliže se tisková úloha s požadovanými sestavami dostane náhodou zrovna za něčí 1000stránkový manuál, může uživatel čekat poměrně dlouho. Většina sestav se navíc skutečně vytiskne hned napoprvé správně, takže toto zdržení je asi tak v 99 procentech případů zbytečné.

Já osobně ošetruji proto problémy s tiskem jako výjimky – ostatně ničím jiným ani nejsou. Pokud se vrátíme k našemu poslednímu příkladu, kde potřebujeme vytisknout pouze nevytištěné sestavy, musíme každopádně doplnit do odpovídající tabulky jedno pole navíc. Jestliže si systém žádá od uživatele potvrzení o úspěšném vytisknutí sestavy, stačí nadefinovat pole typu Ano/ne nebo Boolean. Snadno ale můžeme uložit do tabulky také datum tisku nebo číslo tiskové úlohy. Poté můžeme do dialogu doplnit příkaz, pomocí kterého bude uživatel zaznamenávat problémy s tiskem buďto po jednotlivých úlohách, nebo třeba po jednotlivých fakturách (sestavách).

Jestliže se daná sestava netiskne více než jednou denně, můžeme jako příznak použít aktuální datum. Bezpečnější je ale vždy vygenerovat pro každou tiskovou úlohu jedinečný číselný identifikátor a uložit namísto datumu den. Dojde-li k nějakým problémům, vybere uživatel jednoduše inkriminovanou tiskovou úlohu a systém vrátí do polí s číslem tiskové úlohy původní hodnotu Null. Záznamy s hodnotou Null se automaticky přičlení k následující tiskové úloze. Druhá možnost je zobrazit všechny záznamy z problémové tiskové úlohy a ponechat opětovný tisk konkrétních vybraných sestav na uživateli.

Jak ale uživatel pozná určitou tiskovou úlohu? Do zápatí sestavy bychom například zapsat číslo tiskové úlohy; já ale raději používám jiný postup. Vytvořím systémovou tabulkou, do které uložím číslo sestavy, číslo tiskové úlohy, datum vytisknutí sestavy a jméno uživatele (pokud se dá zjistit), kdo tiskovou úlohu vyvolal. Potom již mohu uživateli nabídnout seznam tiskových úloh s textovým popisem a nemusím je nutit pamatovat si nějaké nesmyslné číslo.

Někdy má ale zahrnutí sestavy do další tiskové dávky mnohem závažnější souvislosti. V účetních systémech se například určité sestavy generují v rámci měsíčních uzávěrek. Po ukončení a uzavření měsíce se provádí inicializace některých hodnot; v takovém případě již třeba neexistuje způsob (nebo alespoň neexistuje žádný snadný způsob), jak sestavy vygenerovat znova. Osobně se ale domnívám, že takové „akční“ sestavy svědčí spíše o nevhodné strategii návrhu.

S ohledem na tuto nespolehlivost tiskových operací se snažím generování sestav a aktualizace záznamů (s výjimkou aktualizací záznamů „sestava vytisknuta“) velice přísně oddělovat – a vám vůbec doporučuji totéž. Jestliže pracovní procesy nesmlouvavě, absolutně vyžadují, aby byla aktualizace ně-

jakých záznamů pevně spojena s tiskem sestavy, pak je nejbezpečnější pozastavit další zpracování v systému až do okamžiku, kdy osoba, která tisk sestavy vyvolala, potvrdí její úspěšné dokončení.

Potvrzení o úspěšném dokončení tiskové úlohy se dá také provést na pozadí, takže nemá smysl systém do vlastního potvrzení zastavovat. Můžeme například ve stavovém rádku vytvořit ikonu, která po klepnutí myši zobrazí potvrzovací dialog a dokončí aktualizaci tabulek. Nezapomeňte ale opět uživateli vhodnou zprávou vysvětlit, co má udělat.

### **Automatický tisk a tisk na vyžádání**

Další otázkou při vytváření standardních sestav je, jestli se mají tisknout jen na vyžádání, automaticky, nebo obojím způsobem. Já se držím jednoho obecného, jednoduchého pravidla a všechny sestavy nechávám tisknout jen na vyžádání. Jedinou výjimku jsem ochotna učinit u sestav, které jsou zcela jednoznačnou součástí pracovního procesu, jako je například faktura vytvořená ihned po zadání objednávky prodeje. Taková automatická sestava se dá ale tisknout i na vyžádání – zejména s ohledem na ošetření oněch otravných problémů s tiskárnou.

Nezapomeňte, že i sestavy spojené s určitým pracovním procesem – například faktury – se dají vytisknout jedna po druhé nebo dohromady v dávce. Faktury tak můžeme tisknout jednotlivě ihned při pořízení odpovídající objednávky prodeje, nebo z nevytištěných faktur vytvořit dávku a vytisknout je po ukončení procesu pořizování dat. Ve své praxi jsem přitom použila oba uvedené postupy. Jestliže mají uživatelé k počítaci připojenou lokální tiskárnu, volím zpravidla jednotlivý tisk faktur, zatímco u síťové tiskárny se kloním spíše k dávkovému tisku. Nejlepší je pak umožnit přesnou konfiguraci uživateli – nastavení tiskáren podléhají poměrně často změnám.

Sestavy, které se tisknou v pravidelných intervalech – například každý týden nebo na konci každého měsíce – generují někteří návrháři v systému automaticky, já ale raději i tyto sestavy nechám tisknout na vyžádání. Implementace automatického generování sestav v systému je doslova složitá. Systém si musí sám spočítat, kdy se má sestava vytisknout (to mimo jiné znamená, že musí počítat například s výkendy a svátky), a dále musí sledovat, jestli již byla sestava ve správný den skutečně vytisknuta. Pokud se systémem pracuje současně více lidí, musí zároveň stanovit, kteří uživatelé nebo která kategorie uživatelů smí tiskovou úlohu spustit, a ošetřit případ, kdy se v požadovaný den do systému nepřihlásí žádný uživatel této kategorie.

Jestliže všechny tyto problémy spojené s automatickým generováním sestav porovnáme s druhou možností, kdy uživatel pohodlně vybere z nabídky příkaz Tisk týdenních sestav, je výsledek jednoznačný. Takový příkaz (případně dialog s informacemi o problémech při tisku sestav) musíme přitom ve vhodné nabídce vytvořit tak jako tak, protože uživatel musí mít pro případ problémů s tiskem vždy možnost opětovného vygenerování sestav.

Kupodivu se dá říci, že není příliš pravděpodobné, aby uživatelé zapomínali tisknout týdenní, měsíční nebo čtvrtletní sestavy. Vůbec ale není na škodu, pokud uživateli nabídnete možnost nadefinovat interval, po kterém se – kdyby náhodou zapomněl – sestava vyvolá automaticky. Jestliže aktuální časový interval vypíšeme uživateli jako implicitní hodnotu a umožníme mu tuto hodnotu změnit, může uživatel opravit případné přehlédnutí. Pomocí tohoto postupu se sestavy, které se tisknou v pravidelných intervalech, mohou vytisknout i před koncem aktuálního časového intervalu. Zkontrolovat plnění plánovaného rozpočtu ještě v průběhu projektu, kdy se dá s potenciálním překročením nákladů ještě něco udělat, je rozhodně více než užitečné.

## Vytváření ad hoc sestav

Pracovní procesy, jejichž podporu má databázový systém zajišťovat, jsou někdy opravdu velice dobře definované. V takovém případě můžeme docela snadno stanovit již předem všechny sestavy, které musí navrhovaný systém generovat. Častěji však musíme uživatelům nabídnout určitou vyšší úroveň flexibility a umožnit mu konfiguraci sestav, nebo dokonce i návrh svých vlastních sestav.

Jakou flexibilitu bude systém potřebovat, to zjistíme vždy z konkrétních potřeb uživatelů, které se systémem od systému liší. Přesný stupeň flexibility se může pohybovat od jednoduchých úprav, kdy uživatelům umožníme v předem definovaných sestavách zadání doplňujících kritérií filtrování, až po plnohodnotné nástroje pro návrh sestav.

### Návrháři sestav

Jestliže můžeme v systému využít vhodný komerčně dostupný nástroj, jako je například Microsoft Access nebo některý z ovládacích prvků od třetích stran, pak je implementace nástroje pro tvorbu sestav opravdu snadná. (Nové nástroje Microsoft Data Reports, která se dodávají společně s Visual Basicem verze 6, nejsou příliš vhodné, protože na počítači každého uživatele vyžadují přítomnost vývojového prostředí.)

Návrháři sestav dělají uživateli při návrhu požadovaných sestav v podstatě neomezenou volnost. I za tuto flexibilitu musíme ale naneštěstí něčím zaplatit. Zde ovšem ani tak nemluvíme o prvotních nákladech na pořízení nástroje pro tvorbu sestav, i když pokud bychom museli nainstalovat plnohodnotnou kopii Microsoft Accessu stovkám uživatelů, kteří jej jinak ani nepotřebují, zajisté by tyto náklady významné byly.

Mnohem významnější náklady spočívají totiž v tom, že musíme všechny uživatele vyškolit pro tvorbu sestav v příslušném důmyslném a složitém nástroji. Uživatelé tak musí nejen vědět, jak sestavu s pomocí návrháře rozvrhnout, ale také musí alespoň zhruba rozumět databázovému schématu, aby v něm dokázali přistupovat k datům, která potřebují. A to je – i s odpovídajícím školením, kterému věnujeme jistý čas – za hranicemi možností průměrného uživatele. Tito lidé navíc *mají* svoji práci a vytváření uživatelských sestav do této práce zajisté nepatří.

### Uživatelský návrh sestav

Pro uživatele je celý proces návrhu vlastních sestav často pohodlnější, pokud jim nedáme k dispozici jenom běžný komerční nástroj, ale pokud vhodnou uživatelskou utilitu pro návrh sestav implementujeme přímo v systému. Teoreticky bychom mohli vytvořit takový nástroj pro návrh sestav, který by se svojí flexibilitou vyrovnal návrháři sestav Microsoft Accessu a který by byl navíc ušit na míru potřebám našeho systému. Udělat něco takového by ale bylo neobyčejně pracné i časově náročné a ve většině situací to ani nemá smysl. Mnohem obvyklejším řešením je proto nabídnout uživateli jistou množinu předem definovaných sestav a umožnit mu zvolit si konkrétní data, která budou v sestavě zobrazena.

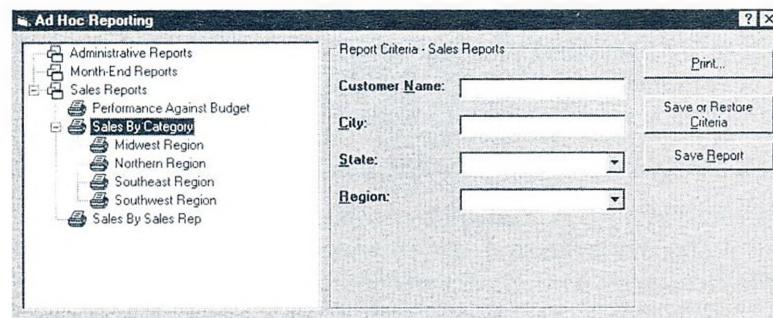
Průvodce Microsoft Access Report Wizard se nachází zřejmě ještě hodně blízko implementačnímu modelu uživatelské tvorby sestav, přesto je ale příkladem dobrého postupu pro vytvoření předem definovaných tvarů sestav. Průvodce Report Wizard nabízí bohaté možnosti formátování, zároveň ale nechává stranou složité aspekty návrhu sestav a umožňuje uživateli výběr ze seznamu před-

definovaných rozvržení a stylů sestav. Jakmile přitom průvodce Report Wizard vygeneruje sestavu, může s ní uživatel dále manipulovat pomocí běžného rozhraní Microsoft Accessu.

Jestliže může uživatel definovat rozvržení (základní tvar) a styl sestavy samostatně, má nad sestavou vyšší stupeň kontroly. Druhou možností je rozvržení a styl zkombinovat a celý postup tak dále zjednodušit. Většina uživatelů si žádá poměrně jednoduché „uživatelské“ sestavy, které jsou ve skutečnosti jen mírně upravenými variantami standardních sestav. Jinými slovy, uživatelé nejčastěji potřebují sestavy, které jsou „zrovna jako tahle, ale...“, přičemž za slovíčkem „ale“ většinou následuje nějaké kritérium pro řazení nebo filtrování.

Já proto při své práci používám zcela jinou „taxonomii“ než průvodce Microsoft Access Report Wizard. Každou ad hoc sestavu rozdělím do dvou komponent, z nichž jednu nazývám „formát“ a druhou „kritéria“. Formát v sobě zahrnuje komponenty rozvržení a stylu sestavy a zároveň definuje pole, která se mají tisknout (pojem můžeme také rozšířit na tabulku či dotaz, na němž je sestava založena). Za běhu se tak ve skutečnosti podle kritérií stanovených uživatelem modifikuje objekt Report v Microsoft Accessu, respektive objekt Data Report ve Visual Basicu. Tato kritéria specifikují řazení a filtrování, které se má na formát aplikovat. Osobně nabízím uživateli téměř vždy možnost uložení a opětovného využití kritérií. Jak něco takového funguje, to si ukážeme za chvíli. Někdy je také vhodné umožnit uživateli definici různých úrovní seskupení, to ale obvykle pokládám za zbytečné.

Uživatelé zde tedy specifikují pouze dvě komponenty, takže pro generování uživatelských sestav používám vhodný dialog. Obecnou strukturu tohoto dialogu ukazuje obrázek 17-4. (Jestliže se rozhodnete postupovat takto i ve svém systému, musíte samozřejmě ukázku upravit, aby odpovídala jeho uživatelskému rozhraní.)



Obrázek 17-4 Pomocí této základní struktury nabízím uživateli funkce vytváření ad hoc sestav

Dialog můžeme rozdělit na tři části. V jeho levém podokně se nachází ovládací prvek stromového zobrazení, který nabízí uživateli výběr ze seznamu formátů. Namísto ovládacího prvku stromového zobrazení bychom mohli použít také běžný seznam, nebo dokonce jen skupinu tlačítek přepínače. V našem příkladu jsou jednotlivé formáty seskupeny do několika kategorií. Toto seskupení je užitečné zejména u většího počtu sestav. Ve skupinách formátů „Administrativní sestavy“, „Měsíční sestavy“ a „Sestavy prodeje“ tak například uživatel mnohem snáze najde, co právě potřebuje.

Třetí úroveň hierarchie z obrázku 17-4 tvoří „sestavy“. V tomto kontextu již za sestavu považujeme jistou kombinaci uložených kritérií a formátu. Jestliže uživatel zadał například v kritériích formátu pro sestavu prodeje Region „jihozápad“, může hotovou sestavu uložit jako „Prodej v regionu jihozápad“. Zejména u specifikací složitějších kritérií může toto uložení ušetřit uživateli poměrně zajímavé množství času.

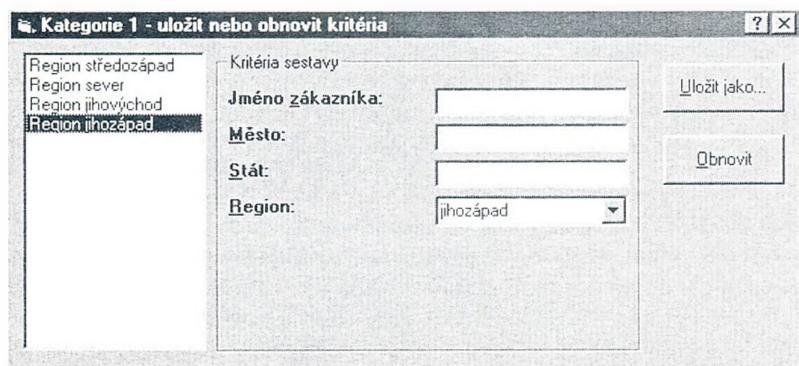
V prostředním podokně dialogu zadává uživatel kritéria pro řazení a filtrování. Někdy se dá v podokně kritérií použít pro všechny formáty jedna stejná množina ovládacích prvků, i když třeba musíme zablokovat jisté ovládací prvky, které pro vybraný formát nemají smysl. A někdy potřebujeme u každého formátu pro definici kritérií zcela jinou množinu ovládacích prvků. Já se při své práci vydávám obvykle zlatou střední cestou. Příklad jsme viděli na obrázku 17-4: zde jsme formáty sestav rozdělila do tří kategorií, přičemž konkrétní množinu ovládacích prvků pro definici kritérií definuje právě příslušná kategorie.

Pravé podokno dialogu na obrázku 17-4 obsahuje množinu příkazových tlačítek. Tlačítko Tisk vyvolá pomocný dialog, v němž může uživatel nadefinovat další možnosti tisku, jako je například počet kopií a použitá tiskárna. Jestliže žádné takové volby nejsou k dispozici, je vhodnější nabídnout uživateli přímo v dialogu dvě příkazová tlačítka, Tisk a Náhled (Ukázka před tiskem), a ušetřit mu tak nutnost vyvolání dalšího dialogu.

Příkazové tlačítko Uložit nebo obnovit kritéria zobrazí jiný dialog, jehož příklad vidíme na obrázku 17-5. Tento jednoduchý dialog nabízí stejné ovládací prvky pro zadávání kritérií, jaké jsme již viděli v prostředním podokně hlavního formuláře pro vytváření uživatelských sestav. Kritéria, která již uživatel uložil, jsou uvedena v seznamu po levé straně dialogu. Vybráním položky z tohoto seznamu se příslušné kritérium promítnete do prostředního podokna dialogu. Tlačítko Uložit jako... se nejprve uživatele zeptá na název a poté uloží zvolené kritérium, zatímco tlačítko Obnovit nače kritérium do hlavního formuláře.

Takovéto ukládání kritérií se implementuje poměrně jednoduše. Stačí pro každou kategorii vytvořit odpovídající tabulku a do prostředního podokna uvést ovládací prvky pro jednotlivá pole. V prostředí s více uživateli se přitom musíme rozhodnout, jestli mají být uložená kritéria k dispozici všem uživatelům, nebo jestli bude mít každý jednotlivý uživatel svoji vlastní sadu kritérií. Pokud jsou kritéria společná (sdílená) pro všechny, je třeba uložit tabulky do hlavní databáze společně s ostatními sdílenými daty. Jestliže má každý uživatel svoje vlastní kritéria, musíme tabulky uložit lokálně, ve front-end databázi (případně v lokální databázi, pokud je hlavní aplikace napsána v jiném prostředí než v Microsoft Accessu).

Uvedené dva postupy se ovšem nemusí vzájemně vylučovat. Snadno můžeme nabídnout uživatelům jak společná, tak i privátní kritéria – stačí do tabulky kritérií uložit také jméno uživatele, respektive zobrazit sjednocení (UNION) sdílených a lokálních tabulek. Já obvykle ukládám do spo-

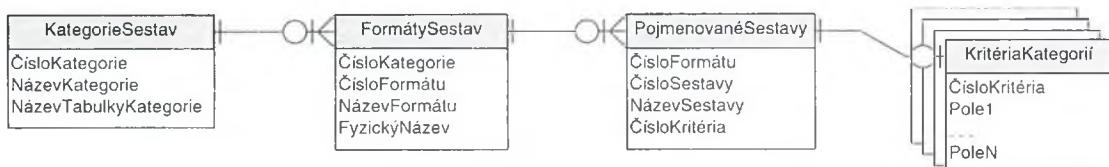


Obrázek 17-5 Tento dialog, který se vyvolá z hlavního formuláře pro vytváření sestav, umožňuje uživateli uložení a opětovné obnovení kritérií sestavy

lečné tabulky jméno uživatele, protože takto může systém snáze podporovat i „toulavé“ uživatele, kteří budou k systému přistupovat z několika různých počítačů.

Kritéria můžeme v tomto mechanismu ukládat také podle kategorií. Uložená kritéria se tudíž dají aplikovat na libovolnou sestavu v této kategorii (za předpokladu, že sestavy mají společné specifikace kritérií).

Někdy je ale vhodnější propojit kritéria s určitým formátem sestavy; přesně to dělá na obrázku 17-4 (na straně 246) poslední příkazové tlačítko. Implementace je opět docela jednoduchá. Budete potřebovat tabulku pro každý typ kritérií (jestliže jste implementovali ukládání kritérií, můžete znova využít již vytvořené tabulky) a dále tabulku, která propojí formáty sestav s kritérii. Celou strukturu ukazuje obrázek 17-6.



Obrázek 17-6 Tato struktura tabulek umožňuje uživateli ukládání kritérií a sestav, a navíc umožňuje za běhu přidávání dalších sestav do systému

První tabulka, KategorieSestav, obsahuje pole s názvem NázevTabulkyKategorie. Z tohoto pole systém pozná konkrétní tabulku kritérií, která obsahuje kritéria pro tuto kategorii formátu. Jestliže mají všechny sestavy v systému stejnou strukturu kritérií, pak toto pole není potřeba. Na druhé straně, pokud mají všechny formáty jinou strukturu kritérií, musí být toto pole uvedeno v tabulce FormátySestav.

Struktura tabulek, kterou jsme viděli na obrázku 17-6, umožňuje konfiguraci všech dostupných sestav za běhu; k tomu slouží také pole NázevFormátu a FyzickýNázev v tabulce FormátySestav.

tav. Pokud jsou formáty sestav definovány v systému nezávisle, tedy jako objekty v databázi (jako je tomu například u sestav aplikace Microsoft Access), respektive jako samostatné soubory (takový mechanismus implementuje řada různých nástrojů pro tvorbu sestav od třetích stran), můžete přidávat sestavy do systému kdykoli pomocí nepřímého přístupu. V takovém případě bude seznam formátů v dialogu vycházet z tabulky FormátySestav; nebudeme jej do aplikace programovat napevno.

Při přidávání nového formátu sestavy stačí pouze vytvořit objekt formátu (v Microsoft Accessu vytvoříme sestavu, jinde to může být samostatný soubor operačního systému) a poté doplnit odpovídající záznam do systémové tabulky FormátySestav. Nový formát bude v systému k dispozici automaticky, aniž by kdokoli musel jakýmkoli způsobem zasahovat do jeho základních funkcí. Pole NázevFormátu obsahuje text, který se zobrazí uživateli, zatímco FyzickýNázev definuje skutečný název objektu, případně jeho cestu k souboru, pokud jsou objekty uloženy externě.

Metoda vytváření uživatelských sestav, kterou jsme si zde právě vysvětlili, je sice pro uživatele jednodušší než plnohodnotný nástroj pro návrh sestav, v některých aplikacích je ale naopak tak říkajíc kanónem na vrabce. Uživateli by třeba úplně stačilo, kdyby mohlo k nějaké předem definované standardní sestavě tu a tam specifikovat doplňující kritéria filtrování. Tato jednoduchá kritéria můžeme často zabezpečit pomocí několika málo ovládacích prvků, kterými uživatel nadefinuje kritéria přímo v dialogu tisku; nemusíme tak složitě implementovat plnohodnotné rozhraní pro tvorbu uživatelských sestav, jaké jsme si právě ukázali.

## Standardní dopisy

Speciálním typem uživatelské sestavy je standardní dopis. Někdy je text takového standardního dopisu pevně dán, častěji si ale uživatel potřebuje vybrat z určitých předdefinovaných odstavců, jejichž spojením teprve dopis vznikne. Ať už je ale text dopisu pevně dán nebo ne, osobně se domnívám, že databázová sestava každopádně není právě nejlepším prostředkem pro vytváření korespondence.

Možnosti formátování databázových sestav se sice neustále zdokonalují a rozšiřují, sotva se ale někdy vyrovnaní textovým procesorem, které jsou pro tyto úkoly speciálně určeny. Navíc, většina uživatelů potřebuje mít možnost doplňovat do dopisu ještě před tiskem další text; dovolit uživateli přímo manipulovat s databázovou sestavou však není vhodné, protože veškeré změny ve standardní struktuře by se provedly jako trvalé.

Jména a adresy osob, a dejme tomu i předem definované odstavce, je rozhodně možné mít uloženy v databázi. Jestliže ale tyto údaje přivedeme do textového procesoru, jako je například Microsoft Word, nabídneme tím uživateli mnohem bohatší možnosti formátování textu i dalších manipulací s ním, a to ještě před tiskem dopisu.

Implementovat odesílání standardních dopisů do textového procesoru je ale naštěstí čím dál jednodušší. Již dávnou jsou pryč doby, kdy jsme museli pracně odesílat příkazové řetězce dynamické výměny dat (Dynamic Data Exchange, DDE) do nějaké nespolupracující, špatně dokumentované aplikace. V dnešním Microsoft Wordu můžeme například tabulkou či dotaz Microsoft Accessu přímo určit za zdroj hromadné korespondence. Poté můžeme dokument spojit s daty z databázové aplikace pomocí jazyka Visual Basic for Applications (VBA) a výsledný dokument zobrazit uživateli k dalším úpravám, nebo jej přímo odeslat na tiskárnu.

V některých případech potřebujeme sledovat, jaké standardní dopisy byly v systému vytvořeny. Jestliže uživatelům neumožňujeme další úpravy dopisů před tiskem, pak nemá smysl ukládat v databázi přímo jejich text. Stačí pouze zaznamenat skutečnost, že daný dopis byl odeslán, případně ještě datum a jméno uživatele, který jej odeslal. Na druhé straně, jestliže uživatel vytváří dopisy pouhou kombinací předem definovaných odstavců textu, můžeme celý proces modelovat s pomocí složené entity, jakou vidíme na obrázku 17-7.



Obrázek 17-7 Do této struktury můžeme ukládat dopisy, zahrnuté do určitého vytvořeného dopisu

Jestliže v systému umožníme uživatelům před tiskem dopisu ještě další úpravy, je vhodnější neukládat text dopisu do databáze, protože ani Microsoft Access, ani Microsoft SQL Server neumí příliš efektivně pracovat s velkými objemy textu. Lepší je uložit do databáze pouze umístění a název fyzického souboru s dokumentem. To znamená, že systém musí být schopen zvládnout přesunutí, přejmenování i odstranění souboru; obvykle přitom požádá o pomoc uživateli.

### **Stručné shrnutí**

V této kapitole jsme se zaměřili na různé aspekty podávání informací, které vycházejí z dat uložených v databázi. To obvykle znamená vytváření různých tištěných sestav; informace můžeme ale uživateli poskytovat také v podobě formuláře či množiny záznamů, kterou zobrazíme jako datový list.

Na začátku kapitoly jsme rozebrali postupy, které Microsoft Access nabízí pro řazení (třídění) a filtrování dat. V souvislosti s těmito postupy jsme si ukázali řadu užitečných příkladů funkcí, jaké můžeme v aplikaci implementovat, a to i v jiném prostředí, než je Microsoft Access.

Dále jsme hovořili o různých typech standardních sestav, které může systém uživateli nabízet; věnovali jsme se tak přehledovým a podrobným sestavám, souhrnným sestavám a konečně sestavám založeným na formulářích systému. Poté jsme si ukázali, jak je možné vytváření sestav integrovat do uživatelského rozhraní systému, a probrali jsme některé otázky ošetření chyb při vytváření sestav. O něco podrobněji jsme se dále věnovali vytváření ad hoc sestav; ukázala jsem vám metodu, pomocí které nabízím tyto funkce uživateli. Nakonec jsme se dotkli vytváření standardních dopisů, které vznikají spojením databázových funkcí pro správu dat a funkcí pokročilého formátování textu, známých ze speciálních textových procesorů.

V následující kapitole přejdeme k tématu pomoci uživatelům a podíváme se, jaké konkrétní formy pomoci uživatelům můžeme zpracovat do uživatelského rozhraní našeho databázového systému.

Všechno, o čem jsme hovořili ve třetí části této knihy, se v jistém slova smyslu dá zařadit pod jeden pojem, „pomoc uživatelům“. Naším cílem při návrhu uživatelského rozhraní a výběru vhodné struktury oken, dialogů a ovládacích prvků by mělo být vždy pomáhat uživatelům systému v zabezpečení jejich běžných úkolů. Ve velké části výkladu problematiky datové integrity v kapitole 16 jsme například hovořili o mechanismech, kterými může systém pomocí uživatelům zabránit v různých nešťastných náhodách, ale které uživateli nesmí (a ani nemusí) nijak překážet. V této kapitole se podíváme na konkrétnější, explicitní formy pomoci uživatelům, které můžeme do uživatelského rozhraní aplikace začlenit.

V kapitole 12 jsme hovořili o třech kategorích lidí, kteří budou se systémem pracovat. Začátečníci potřebují vědět, *co* systém umí dělat. Pokročilý uživatel chce vědět, *jak* v systému provádět určité operace, a experti se již zajímají o to, jak tyto úkony vykonat *rychle*. Pro jednotlivé úrovně uživatelů jsou vhodné různé typy pomoci. Úvodní obrazovka a ukázka s průvodcem je velice užitečná pro začátečníky, zatímco pokročilým uživatelům již překáží a experty přímo rozčíluje.

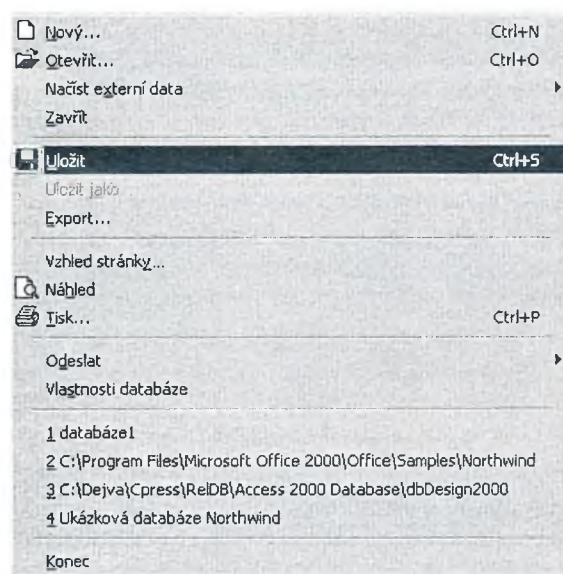
Proto musíme zvážit nejen to, jaké podpůrné mechanismy jednotlivým typům uživatelů nabídнемe, ale také to, jakým způsobem mohou tyto různé mechanismy existovat vedle sebe. Úvodní dialog, který zobrazujeme při spuštění aplikace a který je určen zejména pro začátečníky, může například obsahovat zaškrťávací políčko „Tento dialog již znova nezobrazovat“, pomocí něhož se pokročilí uživatelé tohoto dialogu zbaví. Bublinové nápovědy typu ToolTips a zprávy na stavovém řádku, jejichž úkolem je podporovat středně pokročilé uživatele, již obvykle nepředstavují žádný problém ani pro velmi pokročilé uživatele („experty“), pro jistotu je ale vhodné nabídnout uživateli možnost vypnout i bublinové nápovědy a zprávy na stavovém řádku. O možnostech přizpůsobení uživatelského rozhraní pro různé typy uživatelů budeme v této kapitole hovořit podrobněji.

Kromě toho, jak budou moci jednotlivé podpůrné mechanismy existovat vedle sebe, musíme také zvážit, jak bude systém pomáhat uživateli v přechodu z jedné úrovně znalostí na druhou. Zde se můžeme spoléhat třeba jen na dokumentaci k systému, ta ale – z důvodu, které rozebereme o něco později – není příliš efektivní. Vhodnější je pomocí uživatelům přímo ze samotného uživatelského rozhraní.

Většina systémů byť i jen o průměrné složitosti nabízí ke každému příkazu hned několik cest. Uložení změn v záznamu můžeme například provést buďto pomocí příkazu Uložit z nabídky Soubor, nebo klepnutím na příslušné tlačítko panelu nástrojů, anebo stiskem klávesové kombinace Ctrl+S. Každé z těchto cest se říká vektor příkazu, přičemž různé *vektory příkazu* jsou vhodné pro různé úrovně uživatelů.

Začátečníci a mírně pokročilí uživatelé mají nejradiji nabídky, odkud snadno zjistí, které příkazy jsou k dispozici; mírně pokročilí uživatelé využívají ve větší míře také tlačítek na panelech nástrojů a experti pracují s klávesovými zkratkami (horkými klávesami). V přechodu z jedné úrovni na druhou uživatelům proto nejvíce pomůže, jestliže budeme v celém systému důsledně zobrazovat ke každému příkazu všechny tyto vektory.

Na obrázku 18-1 vidíme standardní (implicitní) nabídku Soubor v Microsoft Accessu 2000. Všimněte si, že u příkazu Uložit jsou uvedeny všechny jeho příkazové vektory. Další možností je přístup pomocí klávesové zkratky Alt-S-U, který zde naznačuje podtržené písmeno U ve slově Uložit. (Písmeno S v názvu nabídky Soubor je podtrženo také.) V nabídce vidíme také ikonu, pomocí které vyvoláme příkaz z panelu nástrojů, a klávesovou zkratku Ctrl-S. Takovéto uživatelské rozhraní, v němž jsou v každé nabídce uvedeny všechny příkazové vektory, již samo o sobě pomáhá uživateli naučit se rychlejší postupy provádění jednotlivých úkolů.



Obrázek 18-1 Ve standardní nabídce Soubor aplikace Microsoft Access 2000 jsou u příkazu Uložit uvedeny všechny jeho příkazové vektory

## Poznámka

*Microsoft Visual Basic nenabízí na svých vestavěných nabídkách ikony z panelů nástrojů; existují však různé ovládací prvky ActiveX od třetích stran, které již ikony v nabídkách podporují. Při vytváření nabídek v Microsoft Accessu 2000 můžeme ikony nadefinovat; výběr vestavěných ikon je ale tak malý, že musíme prakticky vždy nakreslit vhodnou ikonu pomocí Editoru tlačitek. Klávesové zkratky, které se v nabídce zobrazují automaticky, se dají vytvořit velice snadno jak ve Visual Basicu, tak i v Microsoft Accessu.*

Se zobrazením několika příkazových vektorů najednou není žádný problém, protože se jedná o pasivní mechanismus uživatelské pomoci. Veškerou pomoc uživatelům můžeme přitom rozdělit na *pasivní mechanismy*, které jsou vestavěnou součástí uživatelského rozhraní, dále na *reaktivní mechanismy*, které se vyvolávají v reakci na určitý úkon uživatele, a na *proaktivní mechanismy*, které se snaží jistým způsobem předvídat potřeby uživatele. Všem těmto jednotlivým mechanismům se nyní budeme v této kapitole věnovat podrobněji; celou kapitolu završíme stručným výkladem problematiky materiálů ke školení a uživatelského přizpůsobení.

## Pasivní mechanismy pomoci

Mezi pasivní mechanismy pomoci uživatelům zahrnujeme veškerá vodítka, ukazovátka a vysvětlení, která jsou součástí uživatelského rozhraní aplikace a která navádí uživatele v provádění požadovaného úkolu. Pasivní mechanismy pomoci nevyžadují od uživatele – na rozdíl od reaktivních mechanismů – žádnou činnost; odtud také vyplývá jejich název.

Pasivním mechanismem pomoci uživatelům jsou veškeré popisy ovládacích prvků, názvy nabídek, a dokonce i popisy záhlaví formulářů a dialogů; proto je třeba názvy těchto elementů volit opravdu s rozvalou. V zásadě je vhodné volit takové názvy, které co nejzřetelněji popisují prováděné operace nebo zadávané údaje.

Kromě názvů a popisů existuje celá řada dalších mechanismů pasivní pomoci. My se zde budeme podrobněji věnovat jen třem z nich: jsou to mnemonické přístupové klávesy, bublinová návodčida a stavový řádek.

### Mnemonické přístupové klávesy

Mnemonické (mnemotechnicky definované) přístupové klávesy nabízí uživateli přímý přístup k funkcím a přímou pasivní pomoc. Protože přístupové neboli horké klávesy umožňují uživatelům rychlé procházení systémem, nabízí také přímý přístup k funkcím. A protože jsou v popisech ovládacích prvků vždy podtržené, představují zároveň i přímou pasivní pomoc. Metoda přístupu „Alt-podtržený znak“ je důvěrně známá každému uživateli, který má byť i jen částečné zkušenosti se systémem Microsoft Windows.

Mnemonické přístupové klávesy je vhodné definovat pro všechny položky nabídek a pro všechny ovládací prvky. Konkrétní písmeno, zvolené jako přístupový znak, vybíráme přitom podle následující priority:

1. První písmeno položky nabídky nebo popisu ovládacího prvku.
2. Rozlišující souhláska položky nabídky nebo popisu ovládacího prvku.
3. Samohláska v položce nabídky nebo v popisu ovládacího prvku.

Definovat přístupové klávesy je v Microsoft Accessu i ve Visual Basicu triviální: stačí před zóna přístupové klávesy v popisu ovládacího prvku (respektive položky nabídky) zapsat znak & (ampersand). Podtržení písmene v popisu a zpracování přístupu již v obou prostředích zajistí samotný systém. (Mimochodem, jestliže potřebujeme v popisu namísto podtrženého znaku zobrazit znak ampersand, zapíšeme dva ampersandy za sebou. Popis „Pivo & rum“ se tak zobrazí jako „Pivo \_rum“, zatímco „Pivo && rum“ se zobrazí správně jako „Pivo & rum“.)

### Bublinová návodě

Bublinovou návodě neboli tlačítkový tip (ToolTip) pro tlačítko Uložit na panelu nástrojů Formulářové zobrazení Microsoft Accessu ukazuje obrázek 18-2. Na tom ale přece není nic zvláštního, že? Celý princip bublinové návodě je docela jednoduchý: tato návodě v podstatě funguje jako popis tlačítka na panelu nástrojů a dalších ovládacích prvků, které jiný popis nemají.

#### Poznámka

*U jiných ovládacích prvků než je tlačítko panelu nástrojů se tlačítkový tip – ToolTip – nazývá návodě ovládacího prvku (control tip). Návodě typu ToolTip a control tip fungují naprostě stejně, takže je pro jednoduchost budu v této části výkladu zahrnovat pod společný pojem bublinová návodě.*



Obrázek 18-2 Bublinová návodě označuje význam ovládacích prvků, které nemají jiný textový popis

Bublinová návodě je sice možná docela jednoduchá až nezáživná, na použitelnost systému má ale naprostě zásadní vliv. Jestliže jste se někdy rozhodli reprezentovat funkce systému pomocí ikon, pak zajisté dobře víte, jak těžké je vytvořit takovou ikonu, která by sama o sobě nesla nějakou informaci. Zvolit si vhodnou ikonu pro příkaz „Uložit“ asi není příliš obtížné (alespoň pokud jsme někdy viděli ikonu s disketou, kterou známe z Microsoft Office); co ale třeba tlačítko „Otevřít formulář Zákazníci“? Tady můžeme nakreslit malého panáčka; jakou ikonu použijeme ale pro otevření formulářů Zaměstnanci a Dodavatelé? Naše vizuální metafore jsou najednou v úzkých.

Díky bublinové nápovědě se již nemusíme tak úzkostlivě snažit hledat ikony, které by samy dobré vysvětlovaly svůj význam. Většina lidí si dokáže poměrně dobře asociovat obrázek s nějakou myšlenkou pomocí malého příběhu či krátkého textu – přesně tak si člověk může pamatovat jména lidí – a bublinová nápověda jim k tomu dává větší šanci.

Dejme tomu, že tedy formulář Zákazníci budeme v Microsoft Accessu reprezentovat pomocí malé ikony s rybičkou. (Na co jen ti návrháři Accessu myslí?) Že tato rybička reprezentuje zrovna formulář Zákazníci, na to uživatel hned při prvním pohledu na panel nástrojů nepřijde; bublinová nápověda vysvětlí ale význam ikony velice rychle. Jestliže tuto asociaci ikony a jejího významu vhodně podpoříme tím, že obrázek použijeme při každé odvolávce na zákazníky – tedy v nabídkačích, formulářích i v dokumentaci – zvykne si uživatel zcela bez problémů.

Implementovat bublinovou nápovědu v Microsoft Accessu nebo ve Visual Basicu je docela jednoduché: stačí nastavit odpovídající vlastnost. Text vlastnosti ToolTip by přitom měl být krátký – pokud možno jen jedno nebo dvě slova. Nezapomeňte, že bublinová nápověda funguje jako popisek: jejím smyslem je připomenout uživateli, co daný ovládací prvek reprezentuje; nemá za úkol *učit* uživatele, jak s ním pracovat.

Jestliže má daný ovládací prvek stejnou funkci jako položka nabídka, použijte pro bublinovou nápovědu také stejné slovo či frázi jako pro příkaz nabídka. Pokud ovládací prvek žádné položce nabídka neodpovídá, zvolte takové podstatné jméno nebo sloveso, které co nejlépe vystihuje funkci ovládacího prvku a vhodně jej odlišuje od ostatních ovládacích prvků. Máme-li například panel nástrojů se třemi tlačítky, které postupně otevírají formulář Zákazníci, formulář Dodavatelé a formulář Zaměstnanci, můžeme do bublinové nápovědy zapsat třeba jen slovo „Zákazníci“, „Dodavatelé“ a „Zaměstnanci“. V jiném panelu nástrojů, kde bude například jedno tlačítko otevírat formulář Zákazníci a druhé bude provádět tisk seznamu zákazníků, musíme obě tlačítka v bublinové nápovědě rozlišit, takže do nich případně text „Otevřít formulář Zákazníci“ (případně „Údržba Zákazníků“) a „Tisk Zákazníků“.

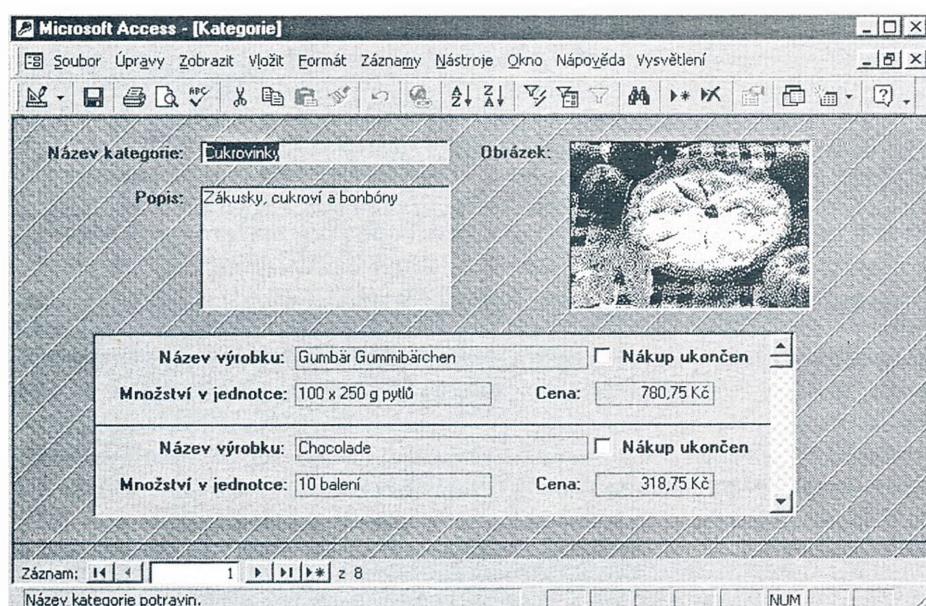
Ještě jedna poznámka k obrázkům: při návrhu vhodného obrázku se budeme snažit vštípit uživateli nějakou myšlenku, nějakou představu. I zde, podobně jako ve všech ostatních oblastech návrhu uživatelského rozhraní, musíme být ale důslední (konzistentní). Jestliže určitý obrázek reprezentuje v systému entitu, pak jej musíme použít na každém místě, kde se na toto entitu odkazujeme; totéž platí i pro obrázek, který vyjadřuje určité sloveso. Já si při své práci vybírám pro každou z těchto kategorií – tedy pro entity a slovesa – množinu vhodných obrázků a podle potřeby je i kombinuji. Pokud se například rozhodnu asociovat operaci odstranění s obrázkem ve tvaru písmene X či přeškrtnutí a tabulkou Zákazníci s obrázkem lidského profilu, vyznačím operaci „Odstranit zákazníka“ složením obou obrázků, jak to vidíme na obrázku 18-3.



Obrázek 18-3 Jestliže obrázky v systému důsledně asociováme s entitami a slovesy, může být jejich spojení velice výmluvné. Pokud v celém systému konzistentně označuje přeškrtnutí tvaru X operaci odstranění a obrázek lidského profilu znamená tabulku Zákazníci, bude většině uživatelů hned na první pohled jasné, že tato ikona znamená příkaz „Odstranit zákazníka“

## Stavový řádek

Stavový řádek je dalším dosti běžným mechanismem poskytování pasivní pomoci uživatelům. Zobrazuje se při spodním okraji okna a může obsahovat různé informace o zapnutém režimu (módu) aplikace, jako je například stav klávesy Num Lock, stav klávesy Caps Lock a rozšířený výběrový mód. Ve stavovém řádku se uživateli vypisují také různé informace. Stavový řádek aplikace Microsoft Access vidíme na obrázku 18-4. Všimněte si přitom, že formulář Kategorie je zde v Accessu maximalizován, takže stavový řádek na spodním okraji okna je zdánlivě součástí formuláře. Ve skutečnosti je ale tento stavový řádek součástí okna samotného Microsoft Accessu.



Obrázek 18-4 Stavový řádek se nachází u spodního okraje hlavního okna

V Microsoft Accessu se dají různé popisné informace zobrazovat do stavového řádku velice snadno, a to zvlášť pro každý jednotlivý ovládací prvek formuláře. Stačí zvolený text přiřadit do vlastnosti Text na stavovém řádku ovládacího prvku. Jestliže u ovládacího prvku svázaného s datovým polem nenedefinujeme vlastnost Text na stavovém řádku explicitně, objeví se ve stavovém řádku hodnota Popis tohoto pole (kterou určíme v Návrhovém zobrazení tabulky). Text stavového řádku však, pokud vím, nelze v Microsoft Accessu nastavit z programového kódu, takže se pomocí něj nedá uživateli zobrazovat ani rozšířený, doplňující popis položek nabídky či jiné zprávy obecného charakteru. (To není pravda; každý ovládací prvek je objektem, takže stačí do jeho vlastnosti – pozor, v programovém kódu se názvy vlastností zapisují anglicky, takže nepíšeme OvládacíPrvek.TextNaStavovémŘádku, ale OvládacíPrvek.StatusBarText – přiřadit zvolenou hodnotu. Nedá se ale nadefinovat nějaký „obecný“ text stavového řádku, který se zobrazí jinde než na ovládacích prvcích – formulář již například vlastnost StatusBarText nemá. Pozn. překl.)

Ve Visual Basicu můžeme text zobrazený ve stavovém řádku ovládat explicitním přiřazením vlastnosti Text odpovídajícího panelu v ovládacím prvku StatusBar. Stavový řádek zde tedy slouží nejen k popisu aktivního ovládacího prvku, jako je tomu v Microsoft Accessu; můžeme do něj v celém systému vypisovat i nejrůznější vysvětlovací zprávy. Nevýhodou této vyšší flexibility je, že text stavového řádku musíme vždy definovat v programovém kódu; ve Visual Basicu navíc nelze žádným způsobem propojit text stavového řádku s ovládacím prvkem a nechat jej zobrazit automaticky. Uznávám, že toto ruční přiřazování textu může být dosti pracné, vzhledem k bohatším funkcím stavového řádku je to ale na druhé straně poměrně přijatelná cena.

Největší výhodou stavového řádku je, že se uživateli nijak nevnuceje. Text ve stavovém řádku nevyžaduje, na rozdíl třeba od okna se zprávou, žádnou reakci z klávesnice, ani jej uživatel nemusí nijak explicitně odstraňovat či potvrzovat. V Microsoft Accessu (a ve Visual Basicu také, pokud tu funkci v systému uživateli nabídnete) se uživatel může dokonce rozhodnout stavový řádek vůbec nezobrazovat.

Ve Visual Basicu můžeme do stavového řádku vypisovat uživateli různé doplňující informace po-pisného charakteru, případně jej můžeme informovat o procesech, které probíhají na pozadí. Do stavového řádku můžeme také zapisovat chybové zprávy, i když zde musíme počítat se situací, kdy stavový řádek není viditelný. Používat text stavového řádku pro informace, které uživatel *musí* vidět, je dosti nebezpečné. I když má třeba uživatel stavový řádek zobrazen, nemusí si všimnout, že je v něm uvedena nějaká důležitá zpráva.

Zobrazení zprávy do stavového řádku má ale svůj smysl. Jestliže například záznamy, které porušují omezení integrity, do databáze neodmítneme zapsat, pouze jejich platnost pozastavíme (jak jsem vám mimo jiné doporučovala v kapitole 16), je velice vhodné vysvětlit uživateli celou situaci ve stavovém řádku a nabídnout mu zde také možnost řešení. Obsah „problémových“ polí mám někdy ve zvyku vypisovat červenou barvou, a jakmile uživatel toto pole vybere, zobrazím popis problému do stavového řádku a naznačím možnost jeho nápravy. Prostor stavového řádku je ale omezen, a proto také nabízím – obvykle po stisku vhodné funkční klávesy – zobrazení dialogu s podrobnějšími instrukcemi.

## Reaktivní mechanismy pomoci

Reaktivní mechanismy pomoci se na rozdíl od pasivních mechanismů, o nichž jsme hovořili před chvílí, zobrazují pouze jako odpověď na určitou akci provedenou uživatelem. Pokročilí uživatelé a experti využívají přitom reaktivní mechanismy pomoci častěji než začátečníci, protože ti je zpravidla neumí vyvolat, nebo třeba dokonce neví, že existují. Proto se tyto mechanismy také příliš nehodí pro pomoc typu „K čemu vůbec tohle a tamto je?“, kterou začátečníci potřebují nejvíce.

Většinu reaktivní pomoci představuje určitá podoba elektronické online nápovědy. Pro poskytování tohoto typu online pomoci existuje několik přístupů, přičemž my se v této části výkladu zaměříme na dva z nich: je to klasická online nápověda a novější tipy „Co je to?“ Jistým způsobem se za reaktivní pomoc uživatelům dají také označit chybové zprávy (přestože se o nich takto obvykle neuvažuje), kterým se budeme věnovat na konci této části kapitoly.

## Online návod

Klasická elektronická (online) návod se představuje v podstatě pouze převodem tištěné dokumentace do počítačové podoby. Jako ostatně při každém převodu nějakého objektu či aktivity z reálného světa do počítače přitom i tato návod některé věci usnadňuje a jiné naopak ztěžuje. Online návod je oproti tištěné dokumentaci dostupnější a možnost zobrazit si související materiál pouhým klepnutím myši je zajisté velkou výhodou. Na druhé straně, online návodou se o hodně obtížněji listuje a nemůžeme si ji vzít s sebou jen tak na čtení do vlaku nebo ke kávě.

Pečlivým návrhem návodného systému můžeme určité nevýhody online návodu oslavit. (I tak je ale elektronická návod přenosná jen do té míry, do jaké je přenosný i počítač.) Návrh a sestavování systémů návodu je velice rozsáhlé téma, jehož většina spadá mimo rámec této knihy. Zde vám proto mohu dát pouze určité obecné zásady, vyzdvihnout některé zvláštnosti vytváření návodu pro databázové systémy a pro podrobnější informace vás odkaži na doporučenou literaturu.

První a nejdůležitější věc, která platí pro návrh každého systému elektronické návody, bez ohledu na to, nakolik úzce je s vlastní aplikací spjata, je, že ji nikdy nesmíme považovat za *integrální* součást uživatelského rozhraní. To znamená, že systém musí vždy stát jíž sám o sobě na svých vlastních nohou a nikdy nesmí do online návodu (a konec konců ani do žádné jiné dokumentace) uživatele při plnění určitého úkolu nutit.

Nezapomeňte totiž, že uživatelé-začátečníci třeba nerozumí, co všechno vlastně systém online návodu obsahuje, takže je ani nenapadne v nesnázích stisknout onu notoricky známou klávesu F1. Online návodu chápejte tedy jako podporu či doplnění ostatní pomoci uživatelům, kterou systém sám o sobě nabízí, nikoli jako její náhradu. Vývojáři se až příliš často domnívají, že stačí vytvořit systém online návodu a aplikace již nemusí být tak názorná a nemusí se sama vysvětlovat. A tento velice častý omyl vede podle mých zkušeností k nepěknému a prakticky nepoužitelnému softwaru.

Na druhém místě musíme při návrhu online návodu zvážit, jaký typ podpory v ní budeme poskytovat. Témata online návodu můžeme zhruba rozdělit do dvou typů: první jsou orientovaná na úkoly a druhá jsou orientovaná na funkce. Témata orientovaná na úkoly uživateli říkají, jak zajistit určitý úkol, jako je například tisk faktury nebo naplánování schůzky. Funkčně orientovaná téma podávají podrobné informace o určité konkrétní funkci (jako je například příkaz Tisk v nabídce Soubor) nebo o určitém ovládacím prvku (jakým může být třeba textové pole KódZákazníka na formuláři). Dá se říci, že tyto dva typy témat návodu přibližně odpovídají Průvodci uživatelů a Referenční příručce z tištěné dokumentace.

V podpoře databázových systémů má své místo jak návod orientovaná na úkoly, tak i návod orientovaná na funkce. Jestliže systém podporuje velké množství různých pracovních procesů, nebo jestliže jsou podporované pracovní procesy doslova složité, může být pro uživatele velkou pomocí právě návod orientovaná na úkoly, která jim podává jakousi „mapu“ potřebných pracovních postupů. I přesto je ale důležité nabídnout uživateli jistou podporu a vedení v celém systému. Zobrazit uživateli pouze nějaký abecední seznam formulářů, jaký známe z okna Databáze aplikace Microsoft Access, a spolehat se, že jen online návod jím řekne, v jakém vzájemném pořadí se musí vyplnit, je zajisté zcela nepřijatelné. (Tedy, nepřijatelné to je alespoň, pokud budete pracovat pro mne.)

Témata online návodů orientovaná na úkoly by v ideálním případě neměla obsahovat žádný myšlenkový nebo úvodní materiál. Nezapomeňte, že tato návod je vložený místem, kde bychom měli uživatele učit, co všechno systém umí. Jediným úkolem online návodů orientované na úkoly je podat přehled, „jak“ určitý postup funguje; „co“ a „proč“, to zde vysvětlovat nemusíme.

Složitější problémy je vhodné rozdělit do několika na sebe navazujících témat návodů; výsledná návod tak bude na obrazovce počítače čitelnější. Jestliže je i vysvětovaný pracovní proces složitý a jestliže se při něm dá postupovat různými směry, je nejlépe nesnažit se vysvětlit všechny možné postupy v jediném tématu. Do hlavního tématu zahrneme pouze nejjednodušší nebo neobvyklejší postup a na téma variantních postupů vytvoříme odpovídající odkazy.

Zatímco téma návodů orientovaná na úkoly se soustředí na to, „jak“ se systémem pracovat, téma orientovaná na funkce hovoří o tom, „co“ se má udělat a „proč“. V databázových systémech se přitom většina funkčně orientovaných témat bude týkat datových položek a ovládacích prvků, nikoli funkcí jako takových. Jen málokdy budeme muset v návodu databázového systému podávat taková téma, jako například v návodu Microsoft Accessu, kde se dozvím třeba přesnou syntax funkce Mid\$().

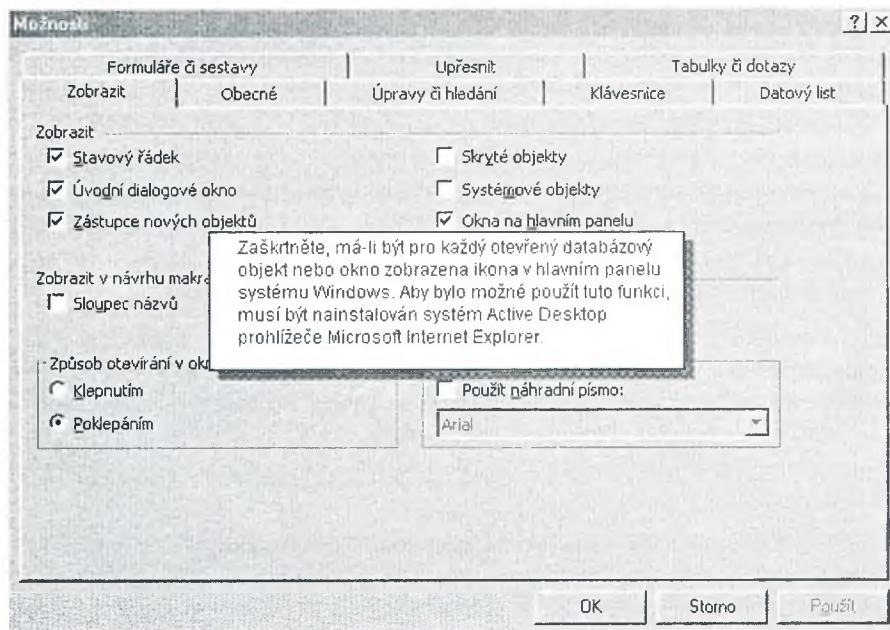
Témata návodů databázových systémů tak musí vysvětlit význam každé entity a atributu v systému a příslušná omezení k nim definovaná. Někdy v nich také potřebujeme uživateli vysvětlit způsob práce s různými typy ovládacích prvků, které systém využívá – například jak procházet ovládací prvek stromového zobrazení nebo jak vybrat datum pomocí ovládacího prvku typu kalendář.

Při plánování těchto datově orientovaných témat je důležité si rozmyslet, proč budou uživatelé tuto návodu hledat. Jestliže si například uživatel prohlíží formulář Objednávky a nachází se na textovém poli s popiskem „Požadované datum dodání“, zřejmě opravdu nebude mačkat klávesu F1 jen proto, že nechápe, že se jedná o datum, kdy zákazník žádá doručení zboží. Pokud v příslušném tématu návodů podáváte uživateli přesně takové vysvětlení (a nic jiného), je horší než k ničemu – pro uživatele je to zbytečná ztráta času, která je navíc jen rozčilí.

Proč tedy *asi* uživatel klávesu F1 stiskl? Možná nerozumí, proč je datum doručení již vyplněno – v tématu návodů můžeme tedy vysvětlit implicitní hodnotu a nabídnout možnost jejího přepsání. Anebo mu třeba zákazník řekl, že chce doručit zboží „kdykoli od prvního následujícího měsíce“ – v takovém případě zde uživateli musíme říci, že má vyplnit nejdřívější požadované datum, případně jiné datum určené pravidly daného aplikacního prostředí. Čím lépe se zamyslíme nad otázkami, na které se uživatel může skutečně ptát, tím efektivnější bude celý systém návodů.

### **Tipy „Co je to?“**

Tipy „Co je to?“ (What's This Help) jsou do značné míry úkolově orientovaným typem online návodů; této orientaci na úkoly neodpovídá pouze způsob jejich vyuvolání. Návod typu „Co je to?“ se totiž vyuvolává klepnutím na ikonu s otazníkem v záhlaví okna, po němž ještě musíme klepnout na požadovaný ovládací prvek okna. Obrázek 18-5 ukazuje návod typu „Co je to?“ pro zaškrťávací políčko Okna na hlavním panelu v dialogu Možnosti aplikace Microsoft Access 2000.



Obrázek 18-5 Tipy nápovědy „Co je to?“ se v dialogu zobrazí po klepnutí na ikonu s otazníkem v záhlaví okna a po následném klepnutí na ovládací prvek

Osobně se mi myšlenka nápovědy „Co je to?“ velice líbí, protože je velice dobře integrována s uživatelským rozhraním systému. Občas jsem viděla, jak i začínající uživatelé „zakopli“ o tipy „Co je to?“. Nikdy jsem ale začátečníky neviděla, že by jen tak „náhodou“ vyvolali stiskem F1 online nápovědu, a dovedu si představit, že by se jí mohli snad i polekat, kdyby se jim najednou na obrázovce začala objevovat různá podivná okna.

U jednoduchých systémů může dobře zvolená množina tipů „Co je to?“ úplně stačit jako celá nápověda v systému, zejména pokud systém nepodporuje příliš mnoho složitých pracovních procesů. Témata nápovědy „Co je to?“ musí být ale poměrně stručná, protože se zobrazují vždy nad daným otevřeným oknem a nedá se jimi rolovat. Potřebujete-li v nápovědě vysvětlit složitější omezení, musíte tipy „Co je to?“, ve kterých na delší text není místo, doplnit jedním nebo více vhodnými tématy běžné online nápovědy. (V takovém případě ale nezapomeňte na konec textu nápovědy „Co je to?“ doplnit upozornění jako „Pro podrobnější informace stiskněte F1“.)

Téma nápovědy „Co je to?“ se snaží odpovědět na otázku – jak již název těchto tipů napovídá – „Co znamená tato věc?“ Tip nápovědy „Co je to?“ si můžete představit jako nějaký speciální, dlouhý titulek ovládacího prvku, který tvoří celý odstavec textu. Ani v nápovědě „Co je to?“ se nám zpravidla z důvodů prostorových omezení nepodaří vymyslet takový text, který by dokázal odpovídat na všechny možné otázky uživatelů. Při troše šikovnosti a při troše přemýšlení svedeme ale s nápovědou „Co je to?“ více než s nějakým byť i dlouhým popiskem ovládacího prvku.

Charakterizovat údaj „Požadované datum dodání“ jako „datum, kdy zákazník žádá doručení zboží“ je v tipech „Co je to?“ zhruba stejné zlo jako v běžné online nápovědě. Snažte se přinejmenším říci, že se jedná o „nejdřívější datum, kdy zákazník žádá doručení zboží“. Implicitně je toto datum tři dny po datu objednání, klepnutím do pole a zapsáním nové hodnoty však toto datum můžete změnit. Pro podrobnější informace stiskněte F1.“

## Akustická kontrola

Akustická kontrola – tedy vyjádření určitého stavu systému pomocí nějakého počítačového tónu – je velice účinným mechanismem pomoci uživatelům; návrháři systémů dokáží ale jeho možnosti využít dobré i zle. Dialog s textem „Píp, teď jsi tomu dal“, který se v systému zobrazí při zjištění „uživatelské chyby“, je výborným příkladem, kdy akustické upozornění slouží nedobrým účelům. Uživatelé nemají rádi, když je někdo příliš upozorňuje na jejich vlastní chyby, přičemž pípnutí upozorní na problém nejen uživatele, který u počítače přímo sedá, ale také jeho okolí. Kromě toho, implicitní (standardní) pípnutí je *opravdu* dosti nepříjemný zvuk.

Jinak postavené akustické tóny však mohou – a umí – být naopak velice dobrým nástrojem. Než by počítač na uživatele při nesnázích nějak nepříjemně zakvílel, je určitě vhodnější pokusit se vydat nějaký jemný zvuk (podle mě je ideální takové malé zamíření, jako když přede kočka) naopak v případě, že je všechno v pořádku. Dejme tomu, že systém při pořizování dat kontroluje s opuštěním odpovídajícího ovládacího prvku každé jednotlivé pole. Jestliže data vyhoví všem relevantním omezením, může systém vydat jemný zvuk „je to v pořádku“. (Tento zvuk ale nesmí být nijak hlasitý.) Jestliže systém narazí na nějaký problém, nevydá žádný zvuk, ale namísto toho zobrazí zprávu ve stavovém rádku. Mrazivé ticho je zde dostatečným upozorněním na vzniklý problém a uživatel se zajisté ihned podívá na obrazovku. (Jiní – včetně mne samotného – by s tímto názorem mohli zajisté polemizovat – pro řadu lidí je příjemnější systém, který přímo upozorní na chybu, než systém, který neustále „přede jako kočka“, a na chybu v podstatě „upozorní“ dosti nevýrazně. Neberme ale autorce její názor... Pozn. překl.)

Nejlepší analogií této techniky pozitivní kontroly (za kterou jsem hluboce zavázána Alanu Coopersovi) je klávesnice. Klávesnice totiž se stiskem každé jednotlivé klávesy docela lehounce klepne. Tento zvuk si nejspíše při práci s počítačem ani neuvědomíte, ale jakmile jej přestanete slyšet, hned si toho všimnete a začnete se pídit po přičinách problému.

Příklad s klávesnicí by měl být také dostatečnou odpovědí na vaše případné obavy, že by se v takové místnosti plné pořizovačů dat nedalo pro zimě různých zvuků pracovat. Jak jsem již ale řekla, „spokojený“ zvuk systému „jsem v pohodě“ nemusí být nijak výrazný, a přesto splní svůj smysl. Mechanismus pozitivní kontroly jsem úspěšně použila například v systému určeném pro telefonní zákaznické centrum, kde pracovalo přes 100 uživatelů v jedné obrovské místnosti.

## Chybové zprávy

Většina lidí nepovažuje chybové zprávy za prostředek pomoci uživatelům – a to je docela škoda. Je ovšem pravdou, že většina chybových zpráv uživatelům *nepomáhá*, ale spíše je trestá. Chybové zprávy však můžeme chápát nikoli pouze jako příležitost k upozorňování na chyby uživatele, ale jako vhodný důvod k pomoci uživatelům. Systém se v podstatě sám dostal do problémů a uživatel musí pomoci se z nich dostat zase ven.

Slušně vychovaný člověk neodpovídá na žádost o pomoc žádnými záhadnými větami, ani nic dalšího nevyžaduje. Slušně vychovaný člověk si také nedomýší, že si za problémy může jen jeho protějšek. Slušně vychovaný člověk vysvětlí problém, jak nejlépe umí, zdvořile požádá o pomoc, nesnaží se nijak zbytečně vnucovat a trpělivě vysvětlí důsledky své rady.

Stejně by se měl chovat i „slušně vychovaný“ počítačový systém. Počítačový systém si ve skutečnosti nezasluhuje takovou úctu jako lidé, a proto musí být více úslužný. (Klidně by se před uživatelem mohl i plazit po kolenou.) Při vydání „chybové“ zprávy je tak systém povinen:

- Vysvětlit zřetelně nastalou situaci, a to takovými slovy, kterým uživatel rozumí.
- Požádat uživatele zdvořile o pomoc, a nedávat najevo, že chybu udělal uživatel.
- Nežádat uživatele zbytečně o cokoliv, co může systém docela rozumně udělat sám.
- Popsat případné důsledky veškerých operací, které uživatel může provést.

Někdy může být systém zcela beznadějně zmaten, a díky různým vnějším faktorům, jako je například vyčerpání volné paměti nebo havárie pevného disku, může občas nastat situace, kdy systém nedokáže bez zásahu uživatele vůbec pokračovat. Při takové události nemáme v podstatě jinou možnost, než zobrazit uživateli zprávu o chybě. Zde můžeme přitom nejen vyžádat si pomoc, kterou systém potřebuje, ale také uživateli podat informace, které *on* sám potřebuje k tomu, aby se systém od podobných problémů v budoucnu vyvaroval.

Pokud vzniklou situaci vysvětlíme jasně, bez nějaké nesrozumitelné hantýrky, bude uživatel moci vyřešit problém přímo. A pokud uživatel hned také pochopí podstatu problému, může mu v budoucnu předcházet – samozřejmě za předpokladu, že se dané situaci dá předcházet. Zadání neplatného formátu dat se předcházet dá; haváriím disku obvykle ne.

Jestliže zmatenému a rozladěnému uživateli nenabídneme jen nějaké jakoby náhodně vybrané varianty, ale vysvětlíme mu i jejich důsledky, máme více než dobrou šanci, že systém od něj nakonec dostane tu nejrozumnější možnou odpověď. Uvědomte si, že pro vás samotné se hodně věcí může zdát zřejmých, ale pro uživatele již tak jasné nejsou. Nesnažte se v chybových zprávách mluvit přespříliš, ale na druhé straně se nebojte mluvit jasně.

Zformulovat zprávu zdvořile a nijak se uživateli nevnucovat je v podstatě jen záležitostí dobrého vychování. Vzpomínáte si, jak vám maminka říkala, že na med chytíte více much než na oct? Jestliže budou uživatelé vnímat váš systém jako příjemný, zdvořilý a užitečný, zajisté vám prominou jednu podivnou obecnou chybu ochrany, kterou jste prostě nemohli odhalit. (Nikdo není dokonalý.)

## Proaktivní pomoc

Kategorie pasivních a reaktivních mechanismů pomoci uživatelům jsou poměrně správně chápané, i když s růstem našich znalostí interakce mezi člověkem a počítačem a s příchodem nových implementačních nástrojů jsou stále důmyslnější. Poslední kategorie, proaktivní pomoc, je do značné míry novinkou a i dnes ji jen málo systémů implementuje.

Princip proaktivní pomoci uživatelům je docela jednoduchý: systém sleduje činnosti uživatele a snaží se proaktivně pomáhat, tedy předvídat jeho práci a navrhovat možnosti efektivnější práce se systé-

mem, případně přebíráním některých úkolů od uživatele do svých vlastních rukou. Zřejmě nejviditelnějším zástupcem proaktivní pomoci je Pomocník Microsoft Office, který uživateli nabízí tipy podle toho, jakou práci v aplikaci provádí.

Jedním typem proaktivní pomoci, která si v současné době získává stále větší pozornost, je inteligentní agent. Takový inteligentní agent je softwarová komponenta, na kterou může uživatel postoupit různé úkoly. Inteligentní agenti jsou obvykle implementováni jako speciální webové uživatelské rozhraní, které dokáže odpovídat na požadavky jako například „Najdi mi nejlepší cenu této položky“, nebo „Navrhni mi nějakou knihu, kterou bych si rád přečetl“. Takové agenty však samozřejmě nic neomezuje jen na webové prostředí. V systému můžeme například implementovat agenta, který studentům pomáhá sestavit si rozvrh přednášek podle jejich osobních preferencí – „Nedávej mi žádnou matematiku na dopoledne a jazykové přednášky mi naplánuj na večer.“

Microsoft nabízí dva nástroje pro vytváření rozhraní založených na animovaných postavičkách, a sice Pomocníka Microsoft Office a Microsoft Agenta. Onu známou poskakující kancelářskou sponku, která v Microsoft Office reprezentuje Pomocníka (konkrétní podobu si ale uživatel může změnit), většina lidí asi zná; ne každý si ale uvědomuje, že Pomocník Office má i programové rozhraní. Pomocník Office je ovšem k dispozici pouze v aplikacích Microsoft Office a s běžovým strojem Microsoft Accessu se distribuovat nedá.

Jestliže pracujete v prostředí Microsoft Accessu (případně v jiném vývojovém prostředí, které využívá ovládací prvky Microsoft ActiveX, jako je i Visual Basic), můžete si z webového sídla firmy Microsoft stáhnout nástroje Microsoft Agent SDK. Tyto nástroje mají výrazně bohatší možnosti než Pomocník Microsoft Office.

Microsoft Agent je velice pěkná hračka. Kromě postaviček, definovaných přímo v SDK, si můžete navrhnut a animovat i svoje vlastní. Microsoft Agent podporuje dále mechanismus rozpoznávání řeči (samozřejmě anglického jazyka, pozn. překl.), který skrývá pro databázové aplikace přímo fascinující možnosti. Hrozně ráda bych viděla třeba rozhraní Microsoft Agenta k nástroji Microsoft English Query, který je součástí Microsoft SQL Serveru verze 7.0 a který nabízí zpracování dotazů SQL zadávaných v přirozeném jazyce. Představte si, kdyby před uživatele předstoupila celá databáze v podobě postavičky Microsoft Agenta.

Nepropadejte ale přílišnému nadšení. Ani Pomocník Microsoft Office, ani Microsoft Agent nenaší totiž žádnou přímou podporu mechanismů, pomocí nichž bychom mohli monitorovat činnosti uživatele a reagovat na ně. Nabízí pouze jednoduchá rozhraní pro důmyslnou komunikaci postaviček s uživatelem, ale implementovat samotnou inteligenci inteligentního agenta již musíme sami na vlastní pěst.

## Školení uživatelů

Požadavky na školení a zaučení uživatelů se u databázových systémů nijak neliší od ostatních typů softwaru. K prvnímu školení uživatelů stačí tištěná dokumentace, výuka formou přednášky, nebo instruktáž na počítači. A samozřejmě ani zde se jednotlivé možnosti nevylučují.

Jestliže se rozhodnete implementovat určité mechanismy školení pomocí počítače, nezapomeňte si předem ujasnit záběr celého projektu a jeho posluchače. Začátečníky zajímá především to, co vůbec systém umí, a teprve druhotně chtějí vědět, jak se to v systému udlélá.

Záběr školení uživatelů na úrovni začátečníků může být různý a závisí na složitosti systému a na povoleném rozpočtu. Řadu systémů stačí vysvětlit jen na jedné či dvou úvodních obrazovkách. Ve složitějších systémech využijeme více výukových mechanismů – například názorné ukázky, nebo dokonce formální počítačové školení se cvičeními, testy a kvízovými ještě.

Pokročilé uživatele již zajímá, jak se ten který úkol v systému provádí, takže jejich potřebám v řadě případů dobré odpovídá nápověda orientovaná na úkoly. Hranice mezi „nápovědou“ a „školením“ ve skutečnosti často vůbec nejsou nijak pevně dány. Ve složitých systémech však někdy musíme implementovat i pro pokročilé uživatele stejný typ formálního počítačového školení jako pro začátečníky.

Bez ohledu na konkrétní podobu a záběr implementovaného počítačového školení je třeba je vždy do jisté rozumné míry oddělit od vlastního systému. Uživatel by rozhodně měl mít možnost využít výuku přímo z uživatelského rozhraní systému, nejlépe položkou v nabídce Nápověda. Přítomnost školicích materiálů by ale v žádném případě neměla překážet normálnímu chodu systému.

### **Stručné shrnutí**

V této kapitole jsme rozebrali tři typy explicitní pomoci uživatelům: je to pasivní pomoc uživatelům, která je přímo vestavěná do uživatelského rozhraní systému, dále reaktivní pomoc, která se spouští v reakci na určitou činnost uživatele, a konečně proaktivní pomoc, kterou iniciuje samotný systém.

Začali jsme přehledem tří základních typů pasivní pomoci uživatelům – mnemonickými přístupovými klávesami, bublinovou nápovědou ToolTip a stavovými řádky. Dále jsme pokračovali mechanismy reaktivní pomoci, která je obvykle implementována v podobě online nápovědy nebo tipů „Co je to?“, může ale zahrnovat také akustickou kontrolu a měly by do ní patřit chybové zprávy. Nakonec jsme se stručně dotkli nové oblasti proaktivní pomoci a školení uživatelů.

# SLOVNÍČEK POJMŮ

## **abstraktní entita**

Entita, která slouží pouze k modelování vztahů mezi jinými entitami.

## **ad hoc sestava**

Sestava, kterou si konfiguruje sám uživatel po implementaci celé aplikace.

## **agregační funkce**

Funkce jazyka SQL, která vrací souhrnné hodnoty.

## **aktualizační anomálie**

Problémy při manipulaci s daty, jejichž příčinou je nevhodně navržený datový model.

## **alternativní klíč**

Viz náhradní klíč.

## **aplikace**

Formuláře a sestavy, se kterými pracuje uživatel.

## **aplikační omezení**

Omezení odvozené z prostoru daného problému.

## **aplikační pravidlo**

Omezení integrity, které je důsledkem prostoru daného problému, nikoli samotné relační teorie.

## **atribut**

Sloupec v relaci.

## **bezeztrátová dekompozice**

Rozdělení relací takovým způsobem, že je možné je zpětně rekonstruovat beze ztráty informací.

## **binární vztah**

Vztah (relace) se dvěma stranami.

## **booleovský výraz**

Výraz, jehož výsledkem je buďto pravda (True), nebo nepravda (False).

## **cizí relace (nevlastní relace)**

Relace, do které vstupuje primární klíč druhého účastníka relace.

**databáze**

Sjednocení databázového schématu a uložených dat.

**databázová aplikace**

Formuláře a sestavy, se kterými pracuje uživatel.

**databázové omezení**

Omezení integrity, které se odkazuje na více relací.

**databázové schéma**

Fyzické rozvržení tabulek v databázi.

**databázový systém**

Společný pojem pro databázovou aplikaci, databázový stroj a databázi.

**datová integrity**

Pravidla, pomocí nichž databáze zajišťuje alespoň částečnou správnost a věrohodnost dat.

**datový model**

Myšlenkový popis prostoru problému v relačních pojmech.

**deklarativní integrity**

Metoda definice omezujících podmínek integrity, při níž se explicitně deklaruje jako součást definice tabulky.

**dělení relaci**

Operace spojení tabulek, která vrátí z jedné množiny záznamů takové záznamy, jejichž hodnotám odpovídají všechny hodnoty ve druhé množině záznamů.

**doména**

Viz obor hodnot.

**doménové omezení**

Viz omezení oboru hodnot.

**dotaz**

Odvodená relace v systému Microsoft Access.

**entita**

Cokoli, o čem potřebujeme v systému uchovávat informace.

**equi-join**

Viz spojení na rovnost.

**full outer join**

Viz plné vnější spojení.

**inner join**

Viz vnitřní spojení.

**jednoduchý klíč**

Kandidátní klíč složený z jediného atributu.

**junction table**

Viz spojovací tabulka.

**kandidátní klíč**

Jeden nebo více atributů, které jednoznačně identifikují relaci.

**kardinalita relace**

Počet řádků v relaci.

**kardinalita relace**

Maximální počet instancí jedné entity, které se mohou účastnit dané relace.

**kartézský součin**

Relační operace, jejímž výsledkem je spojení každého jednotlivého záznamu z jedné množiny s každým jednotlivým záznamem z druhé množiny.

**kaskádovitá aktualizace**

Automatická aktualizace entit v cizí relaci pro případ, kdy se změní odpovídající entita v primární relaci.

**konkrétní entita**

Entita, která modeluje objekt nebo událost z reálného světa.

**left outer join**

Viz levé vnější spojení.

**levé vnější spojení (left outer join)**

Vnější spojení tabulek, které vrací všechny záznamy z první množiny záznamů uvedené v příkazu SELECT.

**množina záznamů (recordset)**

Obecný pojem, který v Microsoft Accessu označuje fyzickou reprezentaci určité relace.

**náhradní (alternativní) klíč**

Kandidátní klíč relace, který se nepoužívá jako primární klíč tabulky.

**nevlastní relace**

Viz cizí relace.

**obor hodnot (doména)**

Množina hodnot, ze kterých je možné vybírat daný atribut.

**odvozená relace**

Virtuální relace, která je definována s pomocí jiných relací.

**omezení entity**

Omezení integrity, které zajišťuje platnost entit modelovaných v daném systému.

**omezení integrity**

Pravidlo datové integrity.

**omezení oboru hodnot (doménové omezení)**

Omezení integrity, které určuje interval možných hodnot.

**outer join**

Viz vnější spojení.

**pasivní pomoc uživatelům**

Mechanismus pomoci uživatelům, který je vestavěnou součástí uživatelského rozhraní.

**plné vnější spojení (full outer join)**

Vnější spojení tabulek, které vrací všechny záznamy obou účastníků.

**pohled**

Odvozená relace v Microsoft SQL Serveru.

**pole**

Fyzická reprezentace atributu v databázi.

**pracovní proces**

Něco, co se má provádět pomocí databázové aplikace.

**pravé vnější spojení (right outer join)**

Vnější spojení tabulek, které vrací všechny záznamy ze druhé množiny záznamů uvedené v příkazu SELECT.

**primární klíč**

Kandidátní klíč relace, který slouží k jednoznačné identifikaci záznamů tabulky.

**primární relace**

Relace, jejíž primární klíč je uložen i u druhého účastníka vztahu.

**proaktivní pomoc uživatelům**

Mechanismus pomoci uživatelům, který se snaží předem odhadovat potřeby uživatele.

**procedurální integrita**

Metoda zajištění datové integrity, při níž vytvoříme speciální procedury, které se automaticky provádějí při aktualizaci, vložení nebo odstranění záznamu.

**prostor problému**

Část reálného světa, kterou má daná databázová aplikace modelovat.

**průnik relaci**

Relační operace, která vrátí ty záznamy, jež jsou dvěma množinám záznamů společné.

**přirozené spojení**

Speciální typ spojení na rovnost, při němž se celá operace spojení tabulek provádí na základě rovnosti a účastní se jí všechna společná pole, přičemž ve výsledné množině je zahrnuta pouze jedna množina společných polí.

## **reaktivní pomoc uživatelům**

Mechanismus pomoci uživatelům, který se spouští určitou operací provedenou samotným uživatelem, například neplatným zadáním nebo vyžádáním tématu online nápovědy.

## **record**

Viz záznam.

## **recordset**

Viz množina záznamů.

## **referenční integrita**

Soubor takových omezení integrity, která zajišťují zachování platnosti jistých vztahů mezi entitami.

## **regulérní entita**

Entita, která může existovat i bez účasti v některém vztahu.

## **relace**

Logická konstrukce, v níž jsou data uspořádána do řádků a sloupců.

## **right outer join**

Viz pravé vnější spojení.

## **rozdíl relaci**

Relační operace, která vrátí všechny záznamy z jedné množiny záznamů, jímž neodpovídá záznam ve druhé množině záznamů.

## **sirotek (entita sirotek)**

Entita, která nemá vztah k žádné jiné entitě v primární relaci vztahu.

## **sjednocení relaci**

Spojení dvou množin záznamů.

## **schéma**

Fyzické rozvržení (rozmístění) tabulek v databázovém systému.

## **skalární hodnota**

Jednotlivá, neopakována hodnota.

## **slabá entita**

Entita, která existuje jen tehdy, pokud se účastní daného vztahu.

## **složená entita**

Jedna entita prostoru problému, kterou modelují dvě nebo více relací.

## **složený klíč**

Kandidátní klíč složený ze dvou nebo více atributů.

## **spojení na rovnost (equi-join)**

Spojení mezi dvěma tabulkami postavené na podmínce rovnosti.

**spojovací tabulka (junction table)**

Tabulka, která v databázi vyjadřuje určitou relaci.

**standardní sestava**

Sestava, kterou je možné definovat a implementovat jako součást databázové aplikace.

**stupeň relace**

Počet sloupců v relaci.

**stupeň vztahu**

Počet účastníků vztahu.

**tabulka**

Fyzická instanciace určité relace definované v databázovém schématu.

**tělo relace**

Vektory souřadnic (vektory hodnot), z nichž se daná relace skládá.

**ternární relace**

Relace se třemi účastníky.

**theta-spojení**

Technicky vzato jakákoli operace spojení tabulek LŘF "Sčítání" ě. aqN ZWÁřO :i^R+00f#C:\n%1"l

Program Files\PROGRA~1\{IsAdobeADOBE,1\}(\@?Photoshop 5.0 CZ\PHOTOS~1\0CZ(2ZWé\$§3 Photoshop.exe\PHOTOSHOP.EXE:37YÜn)SYSTEMC:\\"\\KNI-B0693\CP\Program Files\Adobe\Photoshop 5.0 CZ\Photoshp.exe@\..\..\..\..\Program Files\Adobe\Photoshop 5.0 CZ\Photoshp.exe)"C:\Program Files\Adobe\Photoshop 5.0 CZ" založená na operátoru porovnání. Normálně se ale tento pojem omezuje jen na spojení tabulek s operátory jinými než rovnost.

**transakční integrita**

Omezení integrity, které kontroluje platnost několika operací v databázi.

**tříhodnotová logika**

Model logického vyhodnocení, který pracuje s hodnotami (uznává je jako výsledky logických operací) True, False a Null (pravda, nepravda, neznámá hodnota).

**tuple**

Viz vektor souřadnic.

**typově kompatibilní obory hodnot**

Obory hodnot, které se dají logicky porovnávat.

**účastník**

Entita, která je spojena určitým vztahem s jinou entitou.

**úkol, úloha**

Diskrétní krok v pracovním procesu.

**unární vztah (relace)**

Asociace mezi entitou a sebou sama.

### **uzávěr**

Princip, podle něhož je výsledkem všech operací na dané relaci vždy další relace, s níž je možné dále manipulovat.

### **vektor příkazu**

Cesta k provádění příkazu v uživatelském rozhraní, například položky nabídky nebo tlačítka z panelu nástrojů.

### **vektor souřadnic, vektor hodnot (tuple)**

Řádek v relaci.

### **vestavěné omezení**

Omezení, které řídí fyzickou strukturu databáze.

### **vnější spojení (outer join)**

Spojení tabulek, které vrací všechny záznamy odpovídajícího vnitřního spojení plus všechny záznamy některého z účastníků (případně obou).

### **vnitřní spojení (inner join)**

Spojení tabulek, které vrací jen ty záznamy, pro něž je daná operace pravdivá (její výsledek je True).

### **vztah**

Asociace mezi dvěma nebo více entitami (rozlišuj od relace).

### **záhlaví relace**

Definice atributu a oboru hodnot na začátku relace.

### **základní relace**

Relace, která je instanciována jako tabulka v databázi.

### **záznam (record)**

Fyzická reprezentace vektoru souřadnic.



# LITERATURA

## Část I – Teorie relačních databází

Date, C. J.: *An Introduction to Database Systems*.

Sedmé vydání Reading: Addison-Wesley Publishing Company, 1999.

Date, C. J. a Huge Darwen: *Foundation for Object/Relational Databases*:

The Third Manifesto. Reading, Mass.: Addison-Wesley Publishing Company, 1998.

Fleming, Candace C. a Barbara von Halle: *Handbook of Relational Database Design*.

Reading, Mass.: Addison-Wesley Publishing Company, 1989.

Teorey, Toby J.: *Database Modeling & Design*.

Třetí vydání San Francisco: Morgan Kaufmann Publishers, 1999.

## Část II – Návrh relačních databázových systémů

Gill, Tom a Susannah Finzi: *Principles of Software Engineering Management*,

Reading, Mass.: Addison-Wesley Publishing Company, 1988.

Haught, Dan a Jim Ferguson: *Microsoft Jet Database Engine Programmer's Guide*.

Druhé vydání Redmond, Wash.: Microsoft Press, 1997.

McConnell, Steve: *Rapid Development*.

Redmond, Wash.: Microsoft Press, 1996.

Pressman, Roger S.: *Software Engineering: A Practitioner's Approach*.

Třetí vydání New York: McGraw-Hill, 1992.

Sommerville, Ian: *Software Engineering*.

Šesté vydání Reading, Mass: Addison-Wesley Publishing Company, 1996.

Soukup, Ron: *Inside Microsoft SQL Server 6.5*.

Redmond, Wash.: Microsoft Press, 1997.

(Vyšlo i v českém překladu, Mistrovství v Microsoft SQL Serveru 6.5, Computer Press, Brno 1998.)

### Část III – Návrh uživatelského rozhraní

Cooper, Alan: *About Face: The Essentials of User Interface Design.*  
Foster City, Cal.: IDG Books Worldwide, 1995.

Heckel, Paul: *The Elements of Friendly Software Design.*  
New York: Warner Books, 1991.

Mandel, Paul: *The Elements of User Interface Design,*  
New York: John Wiley & Sons, 1997.

Microsoft Corporation: *The Windows Interface Guidelines for Software Design.*  
Redmond, Wash.: Microsoft Press, 1998.

Shneiderman, Ben: *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Reading, Mass.: Addison-Wesley Publishing Company, 1980.

# O AUTORCE

## **Rebecca M. Riordanov**

Za svých 17 let bohatých zkušeností v oboru si získala mezinárodní uznání v oblasti návrhu a implementace počítačových systémů, které jsou technicky důkladné a spolehlivé a které efektivně splňují požadavky jejich klientů.

Pracuje jako nezávislý konzultant a specializuje se na návrh databázových systémů a systémů pro podporu pracovních procesů. Za svoji podporu v internetových diskusních skupinách získala v roce 1998 od Microsoftu velice ceněný status MVP.

Rebecca bydlí se svým manželem a dvěma kočkami v holandském Amsterodamu. Její adresa je [rebeccar@ibm.net](mailto:rebeccar@ibm.net).

||

# REJSTŘÍK

## A

abstraktní entity, 13  
ad hoc sestavy, 236, 245  
aktualizace, 226  
aktuální anomálie, 24  
akustická kontrola, 261  
analýza nákladů a přínosů, 112  
    pracovních procesů, 120  
aplikace, 5  
aplikáční omezení, 224, 231  
    pravidla, 137, 224  
architektura dat, 143, 150  
architektury dokumentů, 189  
    programového kódu, 143  
    systému, 143  
atomický operátor, 84  
atribut, 10  
atributy, 13, 137  
    vztahu, 135  
audit, 162  
automatické vyhledávání, 82  
automatický tisk, 244

## B

bezpečnost, 160  
    na sdílené úrovni, 161  
    na úrovni uživatelů, 161  
bezztrátová dekompozice, 28  
binární vztah, 43  
booleovský výsledek, 78  
Boyce/Coddova normální forma, 38  
bublinová návodčka, 254

## C

cíle systému, 103  
cizí relace, 45  
**C**  
čtvrtá normální forma, 39  
čtyřvrstvý model, 145

## D

další normalizace, 37  
databázová integrita, 64  
databázové nástroje, 6  
    omezení, 64  
    schéma, 4, 148, 157, 171  
    stroje, 6  
datová integrita, 59, 223  
    krychle, 93  
datový model, 4, 11, 170  
    typ, 225  
definice formátu, 141  
    parametrů systému, 101  
    pracovních procesů, 101  
tabulek, 157  
    vztahů, 132, 157  
deklarativní integrita, 69  
dělení, 85  
délka, 225  
diagram entit, 19  
    vztahů, 19  
dokument, 166  
dokumentace pracovních procesů, 122  
doménová integrita, 60, 69  
doménové omezení, 60

domény, 16  
 doplňující omezení vztahu, 135  
 dotazy, 159  
 druhá normální forma, 34  
 dvouvrstvé architektury, 154

**E**

entitová integrita, 62, 70  
 omezení, 62  
 entity, 11  
 expert, 181

**F**

filtrování dat, 236  
 podle formuláře, 238  
 podle výběru, 236  
 formát, 225  
 front-end části, 8  
 funkční závislost, 31

**H**

hierarchie, 206  
 hodnoty Null, 226  
 hodnoty, 9

**C H**

chybové zprávy, 261  
 chyby tiskárny, 243

**I**

identifikace datových objektů, 127  
 úkonů, 117  
 identita, 156  
 implementační model, 179  
 indexy, 158  
 interakce mezi entitami, 137  
 internetové architektury, 155  
 intervaly, 227  
 intranetové architektury, 155

**J**

jednoduché entity, 199  
 jednoduchý klíč, 29  
 jednovrstvá architektura, 150

**K**

kandidátní klíč, 29  
 kardinalita, 10  
 vztahu, 45, 134  
 kartézský součin, 88  
 kaskádovitá aktualizace, 64  
 kaskádovité odstranění, 64  
 klasická architektura MDI, 192  
 konkrétní entity, 13  
 kritéria návrhu, 107  
 prostředí, 109

**L**

logické hodnoty, 212  
 operátory, 78  
 logický datový typ, 60

**M**

mentální model, 178  
 měřitelná kritéria, 109  
 Microsoft Access, 3  
 English Query, 240  
 SQL Server, 3  
 mnemonické přístupové klávesy, 253  
 množinové operátory, 85  
 model systému, 232  
 modely rozhraní, 178

**N**

náhradní (alternativní) klíč, 30  
 nástroje, 174  
 návrh, 97  
 databáze, 101  
 návrháři sestav, 245  
 neexistující hodnoty, 66  
 nesprávné zadání, 231

normalizace, 141  
n-vrstvé architektury, 155

**O**  
objektové modely, 7  
odvozená relace, 77  
omezení, 137, 157  
    cizího klíče, 73  
    entitové integrity, 227  
    integrity, 59  
    množiny hodnot, 140  
    referenční integrity, 63, 227  
online návod, 258  
ovládací prvky Windows, 211

**P**  
pasivní mechanismy pomoci, 253  
pátá normální forma, 41  
podrobné sestavy, 240  
podřídy entit, 49  
pohledy, 159  
pokročilé filtrování, 239  
pokročilý, 180  
pomoc uživatelům, 251  
porušení omezení, 68  
pracovní proces, 116, 169, 187  
pravidlo, 69  
primární klíč, 29  
primární relace, 45  
proaktivní mechanismy, 253  
    pomoc, 262  
procedurální integrity, 69  
projekce, 80  
prostor problému, 135  
prototypování rozhraní, 172  
průnik, 86  
první normální forma, 32  
přehled systému, 168  
přehledové sestavy, 240  
přechodová integrity, 61  
přejmenování, 90  
přidávání dat, 226  
přirozená spojení, 82

**R**  
reaktivní mechanismy, 253  
    pomoci, 257  
referenční integrity, 63, 73  
relace, 9  
    typu jedna k jedné, 202  
    typu jedna k více, 202  
    typu více k více, 208  
relační dělení, 85  
    model, 8  
    operátory, 79  
reprezentace množiny hodnot, 213  
restrikce, 80  
rozbor entit, 135  
rozdíl, 87  
rozhraní, 177  
    jednoho dokumentu, 189  
    projektu, 195  
    s přepínacím panelem, 194  
    ve stylu Microsoft Outlook, 190  
    více dokumentů, 192  
rozpor reality, 232  
rozšíření, 90

**S**  
sdílení, 165  
sestavy, 235  
    založené na formuláři, 241  
shrnutí, 166  
schéma, 4  
sjednocení, 85  
skalární hodnoty, 9  
složené entity, 136  
složený klíč, 29  
součtové hodnoty, 92  
souhrnná operace, 89  
souhrnné sestavy, 241  
souřadnice, 10  
specifikace rozhraní, 173  
spojená závislost, 41  
spojení, 80  
    na rovnost, 81  
spouštění sestav, 242

správa změn, 173  
 standardní dopisy, 249  
     sestavy, 240  
 stavový řádek, 256  
 stránkování, 156  
 strategie návrhu, 110  
 struktura databází, 23  
 stupeň vztahu, 18  
 symbolický model, 179

**Š**

školení, 263

**T**

tělo, 10  
 tenký klient, 156  
 ternární vztah, 43, 54  
 textové hodnoty, 220  
 theta-spojení, 82  
 tisk na vyžádání, 244  
 transakční integrita, 65  
 transformace, 91  
 třetí normální forma, 35  
 třídění, 239  
     dat, 236  
 třídy omezení integrity, 224  
 tříhodnotová logika, 78  
 typy integrity, 74

**U**

účastník vztahu, 43  
 úkon, 115  
 unární vztahy, 53

uzávěr, 9  
 uživatel, 180  
 uživatelské rozhraní, 171, 187  
 uživatelské scénáře, 124  
 uživatelský návrh sestav, 245

**V**

validace dat, 223  
 vestavěná omezení, 224, 225  
 vnější spojení, 84  
 volitelnost vztahu, 44, 134  
 výběr jediné hodnoty, 213  
     množiny hodnot, 216  
 vyhledávání dat, 236  
 vývojové nástroje, 8  
 vztah mezi entitou, 135  
 vztahy, 18, 158  
     s předem známou kardinalitou, 57  
     typu jedna k jedné, 47  
     typu jedna k více, 51  
     typu více k více, 52

**Z**

záběr systému, 111  
 začátečník, 180  
 záhlaví, 10  
 základní principy, 27  
     relace, 77  
 zapojení uživatele, 181  
 zatížení paměti, 183





Rebecca M. Riordan

# Vytváříme relační databázové aplikace

Relační databáze jsou základem informačních systémů menších firem i velkých institucí, neobejdou se bez nich internetové obchodní aplikace a jsou ideálním řešením evidence dat i v malém měřítku pro domácí potřebu. Naučit se převádět reálné informace, které nás obklopují, do relačního schématu, a to optimálně s ohledem na cíle navrhované aplikace a požadavky uživatelů, znamená umět „uvážovat relačně“. Návrhem tabulek a relací přitom úspěšný návrh aplikace nekončí: je třeba připravit dotazy pro přímé výběry dat či jako zdroj formulářů, výstupních sestav a webových stránek, zajistit integritu dat na požadovaném stupni důslednosti, zabalit aplikaci do atraktivního, přitom snadno ovladatelného a předvídatelného uživatelského rozhraní, ošetřit potenciální chybové situace... Tomu všemu učí publikace *Vytváříme relační databázové aplikace*, zaplňující takto dosud překvapivě zející mezera na domácím knižním trhu.

Vhodným čtenářem této knihy jsou všichni uživatelé, programátoři a budoucí databázoví vývojáři a implementátoři, kteří již získali základní představu o relačních databázích, ale nemají jistotu, jak ji řádně uplatnit při návrhu a tvorbě vlastních databázových aplikací. Najdou v ní srozumitelným jazykem vysvětlené klíčové pojmy, jako jsou první tři normální formy či různé typy relací, poznají jednotlivé fáze návrhu, jímž se realita převádí do da-

tabázového schématu, pochopí zásady pro volbu polí, datových typů, relací, tabulek a dotazů a uvědomí si úlohu vhodné koncipovaného uživatelského rozhraní.

Kapitoly knihy jsou rozděleny do tří vyhraněných částí:

- 1. Teorie relačních databází** – co je databáze a datový model, principy normalizace, vztahy a relace, datová integrita, relační algebra.
- 2. Návrh relačních databázových systémů** – proces návrhu databáze, myšlenkový datový model, databázové schéma, přednesení a ohlájení návrhu.
- 3. Návrh uživatelského rozhraní** – rozhraní jako důležitý prostředník, různé architektury uživatelského rozhraní, reprezentace entit v návrhu formulářů, výběr vhodných ovládacích prvků, zachování datové integrity, vytváření sestav, mechanismy na pomoc uživatelům.

Ohlasy na původní vydání v angličtině vyzdvihují jedinečně zvládnuté podání z podstaty složité relační teorie, na ukázkách z reálného prostředí, diagramech i příhodách z praxe.

Příklady v knize jsou ilustrovány na produktech Microsoft Access a SQL Server, návrh databázových systémů je však téma s neménou platností v jakémkoli systému, ať už jde o MySQL na Linuxu nebo třeba Oracle.



**Vydalo vydavatelství  
a nakladatelství**

**Computer Press®**  
Hornocholupická 22,  
143 00 Praha 4,  
<http://www.cpress.cz>

**Distribuce:**

**Computer Press Brno**,  
náměstí 28. dubna 48,  
635 00 Brno-Bystrc,  
tel. (05) 46 12 21 11,  
fax: (05) 46 12 21 12,  
e-mail: [distribuce@cpres.cz](mailto:distribuce@cpres.cz)

**Computer Press Bratislava**,  
Hattalova 12,  
831 03 Bratislava, SR,  
tel.: +421 (7) 44 45 20  
44 25 17  
fax: +421 (7) 44 45 20  
e-mail: [distribucia@cpres.cz](mailto:distribucia@cpres.cz)

Publikaci lze objednat  
také na adresu  
<http://www.vltava.cz>

**ISBN 80-7226-360-9  
PRODEJNÍ KÓD: KO393**



9 788072 2636

dopravná cena

**Microsoft®**



VŠECHNY CESTY  
K INFORMACÍM