MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Principy objektově orientované analýzy

DIPLOMOVÁ PRÁCE

Vojtěch Vild

Brno, 2006

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce

Mgr. Jiří Ráček, Phd.

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu práce Mgr. Jiřímu Ráčkovi, Phd. za podporu a za jeho flexibilitu a ochotu snášet netypické termíny našich konzultací.

Shrnutí

Moje diplomová práce je přehledového charakteru a klade si za cíl pomoci začínajícímu analytikovi se zorientovat v komplikovaném moderním objektově orientovaném světě softwaru. Nejedná se pouze o popisnou práci, ale o práci, která hledá jádro analytické profese. Práce čtenáře seznamuje s problémy dnešní doby a na jejich základě definuje požadavky jak na tvorbu software tak na nároky kladené na analytickou profesi. Práce ukazuje význam objektového programování a databází. Vysvětluje, proč je analýza důležitá a popisuje základní analytickou disciplínu modelování. Seznamuje čtenáře s typy modelů, dále pak s objektově orientovanými metodikami, nástroji a porovnává je. Zdůrazňuje úlohu UML. Dále představuje různé typy projetků a dává je do souvislosti s probíranými nástroji. Na závěr práce odhaluje analytické principy a přímo ukazuje, jak je možné je v praxi aplikovat.

Klíčová slova

objektová orientace, databáze, analýza, systém, model, UML, CASE, normální formy

Obsah

Kapitola 1

Úvod

1.1 Smluvní zadání

Cílem diplomové práce je specifikovat současné potřeby trhu s informačními a komunikačními technologiemi, identifikovat základní typy IT projektů a porovnat metody používané pro jejich rešení. Zaměřit se zejména na použité modelovací nástroje, postupy při tvorbě modelu a jejich vztah k celkovému životnímu cyklu SW produktu. Posoudit vhodnost současných metod, metody porovnat, najít společné rysy a na jejich základě předpovědět budoucí vývoj analýzy IT.

1.2 Struktura práce

Práce je stavěna logickým způsobem a každá kapitola navazuje na předchozí kapitolu. Abstrahuje záměrně od technických detailů. Pohlíží na modelování z nadhledu a hledá samotné jádro analytické profese.

Ve druhé kapitole se pokusíme identifikovat náš současný stav společnosti »kde jsme«, vyjmenujeme klíčové charakteristiky dnešní doby a pokusíme se výsledné poznání dát do souvislosti s informačními technologiemi.

Ve třetí kapitole se pokusíme podívat na nutné podmínky, které je třeba vybudovat pro úspěšné řešení IT projektů. Řekneme si, proč je v dnešní době tolik důležitý objektově orientovaný přístup a jaký je význam a přínos relačních databází.

Ve čtvrté kapitole se budeme věnovat analytické profesi v IT a ukážeme, proč je analýza tak důležitá.

V páté kapitole budeme obecně hovořit o jednom z významných pilířů analýzy, a to o modelování systémů. Řekneme si, k čemu je to

dobré a jaké jsou klíčové modely analytika.

V následující kapitole si ukážeme, jak a proč je vhodné při modelování využívat CASE systémy, a uděláme si stručný přehled poskytovaných CASE systémů na trhu.

Sedmá kapitola představuje typické IT projekty. Pomocí již uvedených modelů zdůrazníme jejich specifika a charakteristiky.

Osmá kapitola pojednává o různých metodikách vývoje softwaru, porovnává je a opět modeluje jejich charakteristiky a životní cykly.

Devátá kapitola je nejdůležitější a je pointou a přínosem této práce. Seznamuje čtenáře s analytickými principy, které jsou více než jen klasickou kuchařkou. Tyto principy prezentují, jakým způsobem by se měla ubírat moderní analýza.

Závěr již jen shrne získané poznatky.

Kapitola 2

Charakter dnešní doby

2.1 Masmédia a Internet

Zavedením masmédií a rozšířením internetu nastalo zrušení izolace jednotlivých systémů (kultury, územní celky apod.). Z podnikové perspektivy maj díky tomu potenciální zákazníci velmi snadný přístup k informacím, a proto mohou objektivně volit mezi jednotlivými nabídkami. Následkem toho je velmi intenzivní konkurence. Trhy se otevřely a každý podnik nyní čelí přílivu mezinárodní konkurence z celého světa – a má-li být úspěšný, měl by být stále nejlepší, protože jen nejvyšší kvalita, nejlepší služby a nejnižší cena určují normy pro ostatní.

Díky zrušené izolaci se do kontaktu dostaly nejrůznější systémy a jejich interakce sebou nese naprosto neočekávané důsledky. Dnešní svět je možné docela výstižně charakterizovat spojením »neustálá změna«. Známý Descartův démon říká, že pokud by na světě existoval dostatečně silný intelekt, byl by schopný díky analýze všech současných rysů světa a interakci všech jeho systémů »spočítat« budoucnost. O tom, zda je toto tvrzení pravdivé či nikoliv, můžeme polemizovat, nicméně jisté je, že takový intelekt neexistuje. Díky efektu motýlích křídel navíc ani není možné s určitou jistotou předpovědět více, než jaké bude zítra počasí. Díky divoké interakci nejrůznějších systémů plyne neustálá a nepředvídatelná změna business podmínek a nutí podniky být velmi flexibilní, aby se byly schopné vhodně a rychle přizpůsobovat novému trhu. Jak se v současné době izolace i nadále zmenšuje a zvyšuje se dostupnost informací, zkracuje se i doba životních cyklů produktů i času, který mají firmy na jejich vývoj.

Cykly změn probíhají v neustále kratší periodě. Není tedy již ani

zaručeným receptem být velkým a ještě větším. Největší podniky si s sebou sice nesou dlouholeté osvědčené principy, ale v případě, že tyto principy přestávají být efektivní (a to je v období divokých změn běžné), těžko se v takovém kolosu mění. Problémem je totiž jejich rozsáhlá byrokracie (bez které by se pochopitelně velká organizace neobešla), protože brzdí a znesnadňuje jakékoli inovace a nová řešení. Naproti tomu začínající podniky bez této zátěže dokáží na měnící se požadavky trhu svižně reagovat a dokáží i daleko rychleji a bez zbytečných průtahů nabízet zákazníkům nové generace výrobků či služeb.

Změna se netýká jen zákazníků a konkurence, jak jsme ukázali ve dvou předchozích bodech. Změnila se i ona sama - stala se všeprostupující, neustálou a nepředvídatelnou. Každý z konkurentů firmy, kterých je díky globalizaci daleko více po celém světě, může kdykoli zavést na trh nové produkty (což není v dnešní inovativní době žádný div, technologie se nikdy nevyvíjela takovou rychlostí jako dnes). Je tedy životně důležité, aby byl podnik připraven na rychlou reakci.

2.2 Zhýčkaní zákazníci

Dnes již nestačí nabídnout přijatelný výrobek a dobrou cenu a zákazníci přicházejí. Nestačí dokonce soutěžit jen na základě ceny, přidává se i hledisko kvality, služeb, záruk, širokého výběru a individuálních řešení. Dnešní zákazník chce víc než jen přijatelný výrobek. Dnešní doba je obdobím luxusu, zákazníci již nepožadují pouze uspokojení potřeb, chtějí **dokonalé** uspokojení potřeb, je obdobím, kdy je dokonalost na dosah ruky a každý ji chce. Dnešní člověk se už jen tak s něčím nespokojí. Prvním důvodem, který vedl ke změně postavení zákazníků a prodejců, bylo postupné nasycení trhu. Dnes již lidé (u rozvinutých zemích) mají hmotné statky, které potřebují ke spokojenosti (dům, auto, ledničku, počítač...), takže jen čas od času inovují. Jejich nároky tím ale pochopitelně rostou – nejsou hnáni nutností kupovat, mohou si proto vybírat řešení na míru svým individuálním požadavkům, se službami, které potřebují. Tento přístup si mohou dovolit i díky mezinárodní konkurenci, která na trh dodává mnohem větší sortiment výrobků, než bylo dříve obvyklé. Sílu dává kupujícím i snadný přístup k informacím, které je možné využít ke

srovnání cen a jiných podmínek různých výrobců. Z toho plyne, že podniky, které nejsou radikálně prozákaznicky orientované, nemají na současném trhu šanci.

2.3 Vzrůstající složitost světa

Vlivem rozvoje technologie (stále intenzivnějším právě z důvodu prolomením izolace – díky videokonferencím a Internetu spolu mohou »reagovat« vědecké špičky a tímto urychlovat další vývoj, což zapříčiňuje další intenzivní změnu) se dostáváme do stavu, kdy již není možné být pouze specialista. Všechno souvisí se vším. V současnosti již neplatí to co před 200 lety, že člověk se v mládí naučil, co potřeboval (naučil se být třeba truhlářem) a celý život z toho těžil. Dnešní svět je složitější a člověk i podnik se uplatní pouze tehdy, pokud neustále investuje do svého rozvoje a sebevzdělávání se a využívá mezioborových znalostí.

Díky rostoucí složitosti už často není v podniku nikdo, kdo by rozuměl všem procesům, které v podniku probíhají... Při zavádění softwaru do podniků často musí předcházet silná analytická etapa, která nejprve důkladně zmapuje podnik a až následně je možná implementace.

2.4 IT jako zdroj konkurenční výhody

IT obecně rozšiřuje možnosti a umožňuje zákazníkům poskytovat nové a kvalitní služby a také umožňuje do jisté míry zefektivnit práci snížením nákladů. Schopnost snižování nákladů je ovšem obecně u IT přeceňována.

Podnik, který nevyužívá IT, nemůže již v dnešní době účinně konkurovat, protože není schopen poskytovat poptávané a dnes již standardní služby. S příchodem IT nastal zlom v následujících oblastech (viz. zdroj [4] DP Petra Lidmana):

> Sdílené databáze – pořádek v datech. Dříve mohly být informace v jednu dobu přístupné jen na jednom místě. Díky sdíleným databázím již ale mohou být všude, kde je to potřebné. To eliminuje potřebu řešit činnosti sekvenčně, protože již není

nutné předávat si »složku s údaji«. Složky navíc obsahovaly nekonzistence, které vznikaly přepisováním dat a následným udržováním několika nezávislých formulářů.

Expertní systémy – podpora řadových pracovníků. Běžní zaměstnanci vybavení kvalitním expertním systémem mohou dnes provádět daleko kvalifikovanější činnosti, než jakým by odpovídala úroveň jejich znalostí. Bylo by ovšem chybou pohlížet na expertní systémy jako na náhradu za opravdové experty. Jejich výjimečné schopnosti a především zkušenosti nedokáže žádný počítačový systém nahradit (především ve výzkumu a rozvoji v oboru).

Telekomunikační sítě a bezdrátový přenos dat – podpora decentralizace. Již samozřejmou pravdou se stalo, že části podniku umístěné daleko od ústředí musí být řízeny odděleně, jako autonomní organizace, protože v opačném případě by každé čekání na rozhodnutí neúměrně prodlužovalo dobu řešení daného úkolu. Díky moderním telekomunikačním sítím mohou pobočky disponovat stejnými informacemi jako ústředí a mohou se účastnit videokonferenčních jednání. Výhody centralizace se tedy zmenšují a přínosy z decentralizace jsou naopak výraznější. Od rozšíření bezdrátového přenosu dat již není nutné respektovat ani fyzická umístění kanceláří – přistupovat k informacím, komunikovat a řešit problémy je možné odkudkoli z terénu v rámci dosahu signálu.

Vysoký výkon výpočetní techniky a datové sklady – nová zbraň manažerů. Jedním z přínosů dramatického zvýšení výkonu počítačů a dostupnosti informací je možnost revidovat plány průběžně. Dříve bylo nutné postupně shromažďovat údaje o prodejích, cenách, poptávce a nabídce atd. a jednou za čas z nich sestavit nový výrobní plán. Dnes je možné automaticky z odpovídajících zdrojů zásobovat informacemi výkonný počítač, který plány upravuje průběžně tak, aby odpovídaly současnosti, ne historii.

2.5 Pozice IT

Méně než jedna čtvrtina IT projektů končí podle plánu, aniž by nebyl překročen rozpočet nebo plánovaná doba projektu, takže je často v podnicích cítit zklamání z IT a nedůvěra vůči IT odborníkům. Bohužel dnešní doba je stále ještě plná počítačových šarlatánů a každý, kdo umí PHP, se považuje za odborníka. Bohužel projekty těchto nezkušených individuí dopadají, jak dopadají. Jednotlivé softwarové firmy mají ještě malou tradici, takže pro potenciálního zákazníka neznalého oboru je velmi obtížné rozlišit šarlatána od skutečného odborníka se zkušenostmi.

IT je mladou technologií a není to tak dlouho, co se vedli diskuze, zda-li je programování spíše vědou nebo uměním. Odpověď je ale jiná. Je to inženýrská disciplína podobně jako stavitelství. K tomu by se samozřejmě měl vázat vysoký stupeň formalizace, účinné a jednoznačné metodiky, zásadní role norem všeho druhu. A záruka na dodané dílo... Inženýři nikdy nic nevymýšlí od nuly. Místo toho sestavují řešení podle přesného zadání z předem prověřených stavebních bloků (komponent), na které je spolehnutí. Za absurdní bychom dnes považovali postavení velké budovy bez architektonického nákresu a bez náležité dokumentace. Přesto je dnes tento stav u informačních systémů zcela běžný, a to i za situace, kdy všichni vědí, že rekonstrukce budovy je jev řídký, zatímco rekonstrukce informačního systému je na denním pořádku.

Přes obrovský pokrok informačních technologií i reklamu a sliby tvůrců informačních systémů je však denní realita více než neuspokojivá. Umožňují snad informační systémy našich podniků bezpapírový chod? Mohou se manažeři spolehnout na informace, poskytované systémem na podporu rozhodování a nehledají podklady pro svou práci stále ještě v papírových výstupech (skladové inventury apod.)? Je v současné databázové době vůbec možné, aby existoval tak absurdní systém, jako je systém daňového přiznání, kde občan vyplňuje na různých úřadech opakovaně stejné údaje?

Vývoj aplikací je pod stále vyšším tlakem uživatelů na rychlost jejich uvedení do provozu se současným požadavkem na udržení a zvyšování kvality. Dnešní aplikace musí být také v daleko větší míře integrovány s jinými systémy, které již v provozu existují, se kterými nová aplikace komunikuje. Nutností je i snižování nákladů

na údržbu aplikací. Zkušenosti ukazují, že řešením této situace je zvýšení míry opakovaného použití částí aplikací a přírůstkový postup dodávky systému.

Nebuďme ale pesimisté. Každý obor včetně IT musí nejprve uzrát. Velké technologie (jako třeba parní stroj nebo automobil) byly také prvních 40 let ztrátové a trpěly velkými neduhy čerstvě zrozené technologie. Podle statistiky je období, kdy začne IT sklízet své plody, již za dveřmi. Komponentově orientovaný vývoj a zavedení norem do procesu vývoje je pravděpodobný a logický směr vývoje IT. Snad jednou, stejně jako stavař umí najít příslušné dodavatele oken, dveří a dalších stavebních komponent a postavit nebo rekonstruovat z nich budovu, bude podobným způsobem i systémový integrátor schopný vystavět software.

Moderní techniky na zvyšování kvality a produktivity při vývoji SW produktu jsou:

Iterativní vývoj (Model vodopád je v dnešní době na projekty větší než malé naprosto nepoužitelný, protože jeho zkostnatělost nemá nejmenší šanci se vyrovnat s měnícími se požadavky již během vývoje. Dále neposkytuje zákazníkovi žádné »hmatatelné« výstupy, čímž ohrožuje projekt na poli diplomacie. V současnosti je možné software vyvíjet pouze iterativním způsobem například dodržování milestonů.)

Správa požadavků

Komponentová architektura – CBD (Společnost Gartner Group v roku 2000 vydala prohlášení, že přechod firem na CBD zvýší do roku 2004 jejich produktivitu a dobu »odezvy« pět až desetkrát. Koncept CBD tkví v tom, že firma místo toho, aby vždy vytvářela novou aplikaci tak říkajíc od nuly, použije již vytvořené moduly (komponenty), které nyní spojí do kompletního systému. S rostoucí knihovnou komponent pak klesá doba a náročnost tvorby systému. Komponenty se samozřejmě mohou s každým novým projektem vylepšovat.)

Vizuální modelování (Vizuální modelování umožňuje v srozumitelné, názorné a jednoduché formě vyjádřit podstatné z modelované domény. Díky své přehledné vizuální podobě podporuje komunikaci v týmu a týmovou práci.)

Kontrola kvality

Řízení změn

Kapitola 3

Charakter IT projektů

3.1 Zákon rostoucí entropie – všudypřítomný chaos

Každý program se skládá z obsluhujícího kódu (»binárka«) a z dat, se kterými pracuje. Na samotném počátku byly IT projekty jednoduché, a proto se používalo strukturované procedurální programování a data se prostě četla a ukládala do souborů. Je to jistě nejpřímější a nejjednodušší přístup. Jenže v dnešní době je již nedostačující. Je tomu tak proto, že velikost projektů IT přerostla únosnou mez a tento přístup je od určité velikosti projektu velmi neefektivní.

Softwarový projekt středního a velkého rozsahu představuje stejně jako jakýkoliv jiný větší projekt živnou půdu pro chaos. Každý takový projekt podléhá přirozené entropii a není možné nikdy předvídat, kdy, kde a v jaké formě se chaos vyskytne (což souvisí se známým efektem motýlích křídel – jedna takřka nepostřehnutelná drobnost může u velkého projektu spustit strašlivou lavinu průšvihů).

Pokud například u procedurálně psaného programu bylo náhle třeba něco předělat nebo se ukázalo, že něco není možné plánovaným způsobem implementovat a implementovalo se ad-hoc náhradním způsobem (což je typický zásah chaosu), stal se kód velmi nečitelným a neudržovatelným. Podobně to dopadalo se soubory. Jakmile se projektu trochu dotknul chaos, informace v nich se staly často redundantní, neplatné apod.

V boji proti entropii existuje řada manažerských postupů, ale i na úrovni vývojových prací bylo nezbytné objevit nový přístup. Tím se na úrovni programu stalo objektové programování a na úrovni dat databázový přístup. Oba tyto principy přinesly do vývoje softwaru abstrakci jako prostředek v boji proti entropii.

Proto považuji za základní prvek při tvorbě středních a větších

systémů použití:

objektově orientované analýzy a objektově orientovaného programování (OOP)

používání relačních (objektových) databází

3.2 Abstrakce

Výrazným prvkem OOP a databázového přístupu je skutečnost, že nutí vývojáře program si předem rozmyslet! Při OOP je vývojář na rozdíl od procedurálního programování doslova nucen k provedení návrhu, protože musí dekomponovat »svět« do tříd. U databáze musí nejprve nadefinovat tabulky, než je může v programu používat.

V realitě totiž od určité velikosti projektu platí, že přestává čistě intuitivní přístup přinášet nejlepší výsledky. Podobně byl například v dopravním průmyslu zaveden kruhový objezd (podle mého názoru se jedná o jednu z nejgeniálnějších abstrakcí v dopravního průmyslu aplikovanou).

Výhody správných abstraktních řešení:

Zvyšuje čitelnost

Zvyšuje flexibilitu (odolnost proti změnám)

Zlepšuje znovupoužitelnost

Je možné přehledně vizualizovat

Nevýhody abstraktních řešení:

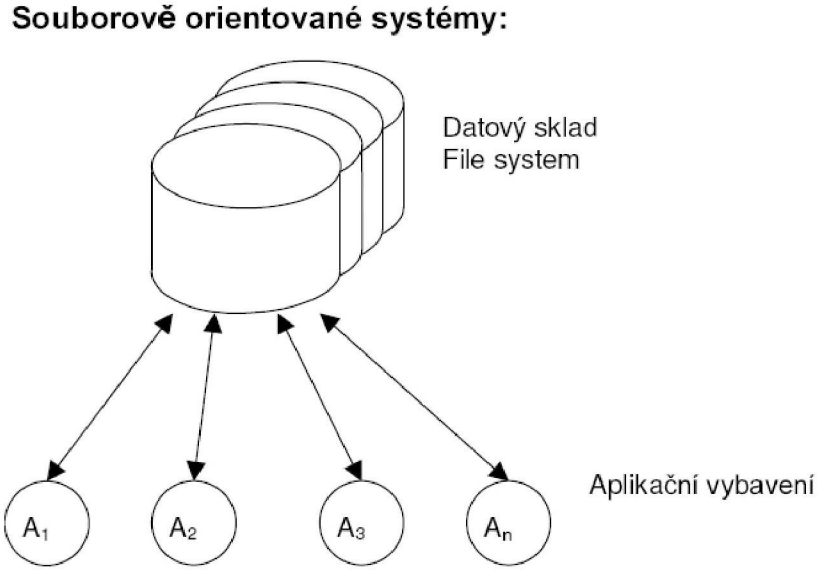
Velká konstantní cena za její zavedení. (Například program vypisující »Hello Word!« v plně objektově orientovaném jazyku je v porovnání se strukturovaným jazykem neskutečně složitý.)

Objekty a databáze mají větší režii než jejich předchůdci. (Což obvykle příliš nevadí, ale je to nepříjemné například u jádra operačního systému.)

Z toho vyplývá, že OOP a databáze nejsou příliš vhodné na velmi malé aplikace, protože se nevyužije jejich přínos v abstrakci, naopak jsou obtížněji implementovatelné kvůli velké konstantní ceně za jejich zavedení. Mohlo by se jednat o příslovečné použití »kanónu na vrabce«.

3.3 Databáze nebo souborový systém?

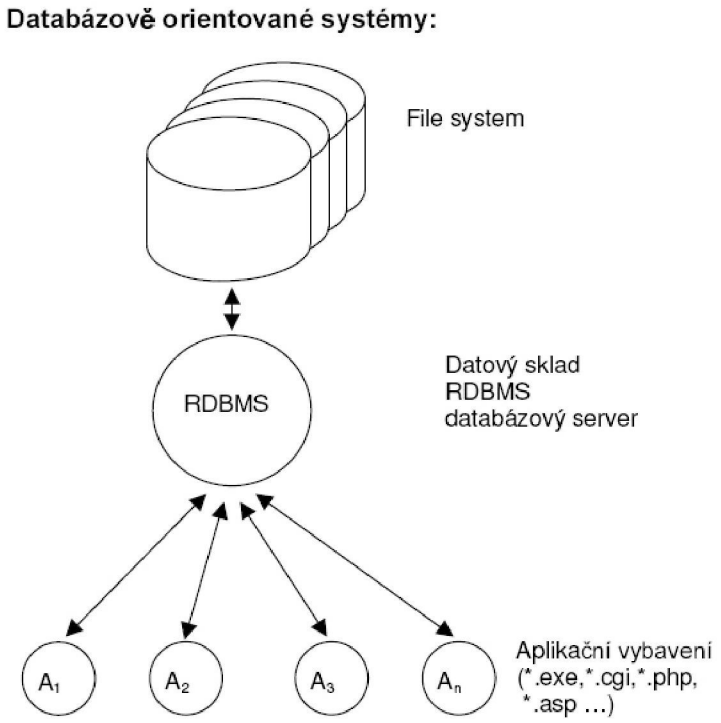
Následující obrázky porovnávají starý souborově orientovaný přístup a databázový přístup (viz. zdroj [14] Materiály k výuce Architektura relačních databázových systémů) .



Obrázek 3.1: ERD před aplikováním principu flexibility

Každý databázový systém je určen svým systémem řízení báze dat (SŘDB). Jedná se o spustitelnou aplikaci nad souborovým systémem, která umí:

Interpretovat SQL pro definici dat a pro manipulaci s daty.



Obrázek 3.2: ERD po aplikování principu flexibility

Přistupovat k jednotlivým záznamům podle jména tabulky a hodnoty primárního klíče. (Proto je časová složitost přístupu logaritmická vzhledem k počtu tabulek a počtu záznamů v tabulce a nikoliv konstantní – systém předem neví, kde je fyzický soubor s daty, tuto odpověď zná až SŘBD poté, co projde příslušný strom (obvykle B+ strom). Logaritmická časová složitost je daň za aplikační nezávislost.)

Zajišťovat integritní omezení.

Podpora transakčního zpracování a multiuživatelského přístupu.

SŘBD má integrovanou podporu pro procedurální jazyk, určený pro implementaci triggerů a uložených procedur.

textbfTyp systému	textbfsouborově orient.	textbfdatabázově orient.
přístup	konstantní	logaritmický
aplikační nezávislost	NE	ANO
zajištění integrity dat	NE	ANO
transakční zpracování	NE	ANO
dokumentovatelnost	obtížná	snadná

Tabulka 3.1: tabulka porovnání souborově orientovaných a databázově orientovaných systémů

3.4 Objektová orientace nebo procedurální?

Procedurální přístup

Nejprve je nutné získané požadavky systematizovat. Vytvoří se seznam funkčních požadavků, seznam událostí a reakcí na ně a seznam výstupů. Poté se začnou modelovat vlastní entity a procesy organizace.

Objektový přístup

Získávají se znalosti budoucího použití systému. Jsou vytvářeny use case a identifikováni aktoři. Výstupem jsou pak jednotlivé diagramy use case (nebo pouze jeden diagram u jednodušších systémů). Dalším krokem je pak analýza sledu kroků v každém use case.

Objektová metodika má své nesporné výhody. Těmi hlavními jsou třídy a jejich zodpovědnost, zdůrazňování iteračního a přírůstkového životního cyklu projektu a názornost vizualizace. Znázornění systému pomocí tříd, které vznikají zobecněním objektů reality, umožní vývojáři realitu lépe pochopit. Na druhou stranu je to jasný a přirozený prostředek pro zachycení struktury reality, srozumitelný i ostatním lidem. Další nespornou výhodou objektové metodiky je fakt, že implementace programů probíhá v objektových jazycích. Přechod od analýzy k implementaci je pak samozřejmě jednodušší. Tento přechod pak také podporují programové nástroje pro podporu návrhu informačních systémů.

3.5 Principy OOP

Abstrakce

Objektově orientovaný přístup se od tradičních postupů liší především ve způsobu myšlení a pohledu na reálný svět. Jde o způsob nahlížení na software pomocí abstrakce reálného světa, který zjednodušuje složitý problém na přijatelnou velikost. Hlavní a největší silou objektově orientovaného přístupu je dekomponování složité reality do přijatelné množiny abstrakcí reálného světa – tříd. Zaměřuje se na podstatné vnitřní vlastnosti objektů a ignoruje jeho nedůležité vlastnosti. To znamená, že klade důraz na to, co objekt je a co by měl dělat, a ne na to, jak by to měl dělat. Smyslem abstrakce je rozdělení odpovědnosti za jednotlivé úkoly do jednotlivých tříd. Objektové modelování, programování a pohled na svět také více odpovídá přirozenému lidskému uvažování.

Třída slouží jako šablona pro tvorbu objektů (neboli instancí třídy). Tyto instance reprezentují konkrétní položky reálného světa, které umí vykonávat stejné »věci«, sdílí stejné vlastnosti, ale odlišují se jejich hodnotou. Objekt (instance třídy) popisuje konkrétní část problémové oblasti. Stane-li se nám, že námi navržená třída má pouze jednu smysluplnou instanci, je zde cosi podezřelého – neznamená to náhodou návrat k procedurálnímu programování?

Výhodou objektů je oproti strukturovanému programování to, že objekty logicky a přirozeně sdružují funkcionalitu s datovou složkou. V důsledku jsou objektově orientované systémy více kohezní a tedy i lépe udržovatelé a přehledné.

Zapouzdření

Zapouzdření vychází z abstrakce. Jeho smyslem je odstínit externího uživatele objektu od implementačních detailů konkrétního objektu.

Zapouzdřením je myšleno to, že objekt má zodpovědnost za určité jeho dovednosti a chová se vůči okolnímu světu jako černá skříňka. Jsou známy pouze jeho veřejné metody, které jsou řádně zdokumentované, a ty dává k dispozici, ostatní metody a atributy má schované – soukromé. Externí uživatel objektu přeci nemá vůbec žádný zájem na tom, aby poznal konkrétní implementační strukturu daného ob-

jektu.

Zapouzdření výrazně zvyšuje znovupoužitelnost.

Polymorfismus

Polymorfismus znamená mnohotvárnost, jde ruku v ruce s abstrakcí a sjednocuje manipulaci s různými objekty. Díky polymorfismu můžeme voláním jedné metody způsobovat rozdílné chování různých objektů.

Polymorfismu je dosahováno používáním rozhraní. Některé zdroje uvádí jako další princip objektově orientovaného programování dědičnost. Dědičnost je v zásadě pouze berlička k dosažení polymorfismu. Navíc používání dědičnosti je často přeceňováno a nadužíváno (protože je pohodlné). Použití dědičnosti je také nebezpečné a dědím-li z předka, kterého neznám (kterého jsem neprogramoval), porušuji principy defensivního programování.

Polymorfismus je zdrojem velké flexibility objektově orientovaných systémů díky tomu, že umožňuje jednotný přístup k objektům.

3.6 Je použití OOP a databází samospasitelné?

Samozřejmě není. Samo objektové programování nezaručuje dobře navržený kód, je to pouze nástroj na programování velkých aplikací, který při správném použití nese kvalitní výsledky.

Stejně tak samo použití databáze není samospasitelné. Pouze dobře konstruovaná databáze přinese pozitivní efekt a to pouze v případě, že její implementaci předcházela kvalitní analýza. O problému kvalitní analýzy pojednává následující kapitola a dále i zbytek této práce.

Kapitola 4

Analýza

4.1 Je analýza nezbytná?

Jak jsem zmiňoval, používání objektově orientovaného přístupu a databázového přístupu začíná být smysluplné až od určité velikosti projektu. U analýzy jsou sice jisté paralely, ale ne tak docela. Analýza je zásadní způsob, jak vytvořit kvalitní softwarový produkt libovolné velikosti.

Nad projekty větší než malé již není možné sednout a začít kódovat. Má to sice nádech romantiky, ale v dnešním světě je to zcela neefektivní. Je tomu tak proto, že každý větší systém obsahuje mnoho souvislostí. Podcenění jádra takovéhoto systému je fatální, protože odstranit chybu, která se stala ve fázi analýzy, je v každé následující fázi dražší. Slavný databázový návrhář Michael Hernandez jednou trefně podotknul, že na poctivou analýzu je vždycky málo času. Ale na rozsáhlou a drahou opravu chyb způsobených odbytou analýzou se čas najde vždy. Ze zkušenosti mnoha projektů vyplývá, že rozdíl v ceně za nápravu chyby, rozpoznané včas během fáze analýzy, je menší o tři řády než chyba rozpoznaná až ve fázi implementace. Zároveň je známo, že v úvodních analytických fázích má původ až 75% všech chyb systému.

Z toho jasně plyne, že bychom se měli snažit úvodní fáze analýzy provádět velmi pozorně. Snaha ušetřit prostředky (peníze a čas) v úvodních fázích vývoje ve prospěch návrhu a implementace je zásadně nesprávná a v konečném důsledku je mnohem dražší. Bohužel podceňování analýzy a přeceňování implementační fáze je typické.

4.2 Úloha analýzy

Analýza je úvodní zamyšlení nad problematikou. Jako je pro architekta budovy zásadní uživatelský pohled – snaží se navrhnout takovou stavbu, aby co nejlépe splňovala svoji funkci v závislosti na procesech, které budou v budově probíhat, a zároveň nepřesahovala jeho finanční možnosti – stejně tak je tento pohled důležitý pro analytika systému. Analýza je tedy uživatelský pohled na systém, zatímco návrh je již vývojářský pohled na systém.

Cílem analýzy je:

získání specifikace zadání (Uvědomit si co chci dělat, zjistit, co chce zákazník.)

získání nadhledu nad danou oblastí a konzistentní pohled (Což je nezbytné pro správná implementační rozhodnutí.)

eliminování složitosti problémové oblasti do uchopitelné podoby pomocí modelů (Jak jsem již zmiňoval, dnešní doba je charakteristická obrovskou složitostí.)

4.3 Získání specifikace zadání

Myšlenka, že zákazník předem zná požadavky na systém je velmi idealistická a hraničí s troufalým optimismem. Představa, že zákazník si předem nadiktuje, co chce a my mu to naprogramujeme, neodpovídá bohužel realitě. Nejen, že zákazník má obvykle pouze mlhavou představu o tom, co od systému požaduje, ale také vůbec netuší, že se jeho požadavky začnou měnit s tím, jak poznává schopnosti IT a dodávaného software během vývoje. Představa, že se »zadrátuje« to, co nám zákazník řekl, že chce, a pak bude zákazník spokojený, již neplatí. Prací analytika je během konzultací zjistit, co zákazník **doopravdy potřebuje**, uvědomit ho o tom (případně ho přesvědčit) a dodat mu to.

Dále je nutné počítat s tím, že požadavky se podle měnícího se businessu budou časem i nadále vyvíjet (a to dokonce už během vývoje daného software). Šanci mají pouze flexibilní (přizpůsobivé) aplikace. Toho je možné docílit dodržením rozšířeného principu abstrakce, viz kapitola Analytické principy.

Základními problémy při specifikaci požadavků je dále také fakt, že zadavatel a uživatel systému nejsou totožní. Přitom zadavatel se snaží o maximální efektivitu vynaložených prostředků, kdežto uživatel vyžaduje maximální pohodlí (maximální podporu jeho konkrétní práce). Dalším problémem je, že informační systém je obvykle určen pro větší okruh uživatelů, přičemž každý z nich má své vlastní požadavky na něj.

4.4 Získání nadhledu nad danou oblastí a konzistentní pohled

Analýza systematizuje získané požadavky a vytváří v mysli analytika nadhled nad danou problematikou. Bez nadhledu není možné dělat správná rozhodnutí. Totéž platí i u implementace systémů.

Analytická profese s sebou nese několik výrazných úskalí. Hlavním úskalím je to, že obvykle jediný analytik má nadhled nad aplikací (na rozdíl od programátorů, zákazníka nebo manažera – ten má nadhled jiného druhu). Analytik je přirozeně pod neustálým tlakem ze strany programátorů i zákazníka, kteří na analytika naléhají, aby konečně odevzdal podklady pro programování. Pokud ovšem analýzu odbudeme, pravděpodobně se nestihneme propracovat k jádru analytického modelu a systém bude špatně navržený, neflexibilní apod.

Další úskalí je, že analytik nikdy nemůže se stoprocentní jistotou tvrdit, že dosáhl jádra analytického modelu. Spíše než exaktní vědou, je práce analytika uměním, které využívá přirozeného citu a zkušenosti.

4.5 Eliminování složitosti problémové oblasti pomocí modelu

Model je něco, co reprezentuje realitu v její zjednodušené formě, kde zůstávají pouze její podstatné a zásadní rysy a nepodstatné rysy a detaily jsou zanedbány. Slouží k přemostění veliké složitosti reality do té míry, abychom mohli realitu »uchopit«, přemýšlet o ní a rozhodovat. O tvorbě modelu systému pojednává následující kapitola.

4.6 Důležitost analýzy v závislosti na velikosti projetku

Úloha analýzy, stejně jako význam OOP a databází, opět roste s velikostí projektu, ani ne tak na nezbytnosti jejího použití (analýza je nutná vždy, ať už v jakékoliv formě), ale spíše v kvalitativních požadavcích kladených na jednotlivé analytiky. U malých projektů je ještě možné, aby byl programátor sám sobě analytikem. S rostoucí velikostí projektu totiž narůstá:

složitost projektu (Je tomu tak proto, že každý větší systém obsahuje mnoho souvislostí – navíc souvislosti v systému jsou složitější než u malých systémů.)

komplikovanější a nejasnější požadavky (Jejich získání je mnohem náročnější.)

rostoucí dekompozice a následně mnohem náročnější fáze integrace jednotlivých komponent (Starou pravdou je, že dílčí řešení nemusí být špatné, ale celek tvoří více než jen prostý součet jeho částí. A právě důležitou úlohou analytika je zajistit, aby jednotlivé části zapadly do jednoho konzistentního, jednoduchého a elegantního celku.)

Kapitola 5

Model systému

Tvorba modelu pomáhá lépe porozumět systému tím, že ukáže shrnutá pozorování v přehledné formě. Jak jsem se již zmínil, modelování je v podstatě určité zjednodušování reality, kde zdůrazníme zásadní rysy a nepodstatné rysy ignorujeme. Je-li model příliš přesný, ztrácí svůj smysl, je-li příliš nedokonalý, může být zdrojem chyb. Záleží pouze na zkušenostech a přirozeném citu analytika, aby odhadl správnou míru.

Modelujeme proto, abychom získali nadhled nad řešenou problematikou a byli jsme schopni zvolit nejlepší možné řešení. K modelování neodmyslitelně patří tvorba diagramů. Diagramy jsou základní abstrakcí, díky které je schopen analytik systematicky přemýšlet o svojí práci a umožňuje věcně komunikovat s ostatními analytiky, dalšími členy týmu, ale i zákazníkem. Jak řekl klasik, jeden obrázek může mít hodnotu tisíce slov.

Diagramy slouží k zachycení modelu systému. Nezávisle na tom, jestli budeme programovat strukturovaně nebo objektově, se pro popis modelu systému objevují tři typy diagramů. Je tomu tak proto, že každý systém má svou statickou složku (v objektovém světě objekty), svou dynamickou složku (procesy) a dále je nutné dát oba tyto světy do souvislosti (objekty vystupující v procesech).

5.1 Statický model

Statický model popisuje stabilní základ systému. Osobně jej považuji ve většině projektů za nejdůležitější, protože zachycuje právě pevnou strukturu systému, o kterou se opírá všechna jeho funkcionalita. Statický model je vždy strukturální a vztahový model. Jeho účelem je prezentovat vztahy jednotlivých entit v systému.

Příkladem statického modelu je model hierarchie podřízenosti v podniku, diagram balíčků, ERD nebo model tříd.

Objektově orientované systémy mají (pochopitelně) jako svůj základní stavební prvek objekty. Předpisem (jakousi šablonou nebo tovární třídou) pro určitou skupinu objektů je třída. Proto zásadní roli v objektově orientované analýze hraje model tříd.

Dnes, kdy jsou změny všeprostupující a neustálé, je vytvoření kvalitního modelu tříd obtížné, protože se může stát, že ještě před dokončením projektu zastará. Jakými principy se řídit při návrhu modelu tříd, bude podrobně popsáno v kapitole o Analytických principech.

Datové modelování a jádro analytického modelu

Když jsem se začínal zajímat o analýzu a začal jsem se setkávat s jinými analytiky, velmi mě překvapilo, jak velký důraz je u analytiků kladen na zvládnutí analytické disciplíny datového modelování. Postupem času jsem si uvědomil, že datové modelování je jádro každé analýzy (tím nemám na mysli pouze nejčastější aplikaci datového modelování pro návrh databáze nebo návrh modelu tříd), protože každá analýza popisuje systém a každý systém je specifický souvislostmi v daném systému. A právě datové modelování je základní souvislostní mapa v každém systému. Jeden datový model popíše statickou-vztahovou strukturu systému lépe než desítky stran textu a umožní analytikovi se na systém podívat objektivněji, z nadhledu – tolik ceněné schopnosti analytika.

Hlavní analytikova práce v objektově orientované analýze je nalezení jádra analytického modelu tříd a prosazení jeho implementace. Jádro analytického modelu není možné matematicky definovat, ale všichni významní analytici potvrzují jeho existenci se slovy, že jádro analytického modelu je elegantní, poměrně jednoduchý, dlouhodobě stabilní, flexibilní a snadno implementovatelný model systému. K nalezení jádra potřebuje ovšem analytik čas a zkušenosti.

5.2 Specifikační model

Specifikační model zachycuje v uspořádané podobě požadavky kladené na systém. Pomáhá nám uvědomit si, co vlastně chceme, a definuje naše zadání.

Příkladem specifikačního modelu je například seznam událostí, diagram funkční hierarchie v strukturované analýze nebo model případů užití v objektově orientované analýze.

5.3 Dynamický model

Dynamický model popisuje chování systému v čase a dává tak do souvislosti statický a specifikační pohled. Dynamické modely popisují chování systému a hrají zásadní úlohu při implementaci. Slouží programátorovi často jako předpis pro práci.

Příkladem dynamického modelu je ve strukturované analýze DFD (data flow diagram) nebo matice CRUD, v objektově orientované analýze to je například stavový diagram nebo sekvenční diagram.

Základním poznatkem při modelování objektově orientovaných systémů je, že systém se skládá z procesů. Základním diagramem pro popis procesu je stavový diagram (nebo jeho odlehčená verze diagram aktivit). Tento stavový diagram reprezentuje šablonu procesu. Konkrétní proces je jeho instancí a probíhá s drobnými odchylkami od této definované šablony. Jako je základním modelem popisujícím statický model model tříd, hraje mezi dynamickými modely dominantní roli právě stavový diagram.

Stavový diagram umožňuje zachytit jednotlivé etapy nějakého procesu (v UML je speciální verzí stavového diagramu diagram aktivit) nebo jednotlivé významné stavy nějakého objektu. Modelování stavů je zásadní konceptuální záležitost.

5.4 Aplikace modelů při tvorbě software

Abych zdůraznil zásadní roli právě modelu tříd a stavového diagramu, podívejme se na cíle analýzy informačního systému.

Životní cyklus systému se zasazuje do prostředí reálného světa, přičemž systému přisuzujeme cíle. A abychom se ve spletitém systému

vyznali, dělíme jej na části. Ty se vymezovaly podle různorodých pravidel, až jsme dospěli k dnešním objektům a procesům jako základním stavebním kamenům systému.

Pro modelování objektů je základním prostředkem diagram tříd a pro modelování procesů je základním prostředkem stavový diagram (nebo případně jeho zjednodušená verze diagram aktivit).

Pomocí diagramu tříd a stavového diagramu je možné namodelovat esenciální část systému. Přestože existují i jiné kvalitní analytické diagramy, po zbytek práce se pro jednoduchost (a také kvůli prezentování síly těchto dvou diagramů) omezím pouze na tyto dva základní prvky analytického aparátu (nebo jejich odlehčené verze, pro model tříd to bude ERD – entity relationship diagram – a pro stavový diagram to bude diagram aktivit) a budu jimi ilustrovat některé následující prvky práce. V příloze DP je popsána notace obou používaných diagramů.

Kapitola 6

Metodiky řešení IT projektů

6.1 Co je to metodika?

Postup tvorby analýzy a návrhu informačního systému popisuje mnoho metodik. Zabývají se jak nejvhodnějším postupem (který bývá často velmi obdobný), tak modely, které je vhodné vytvořit.

Zmíněné metodiky návrhu a tvorby informačního systému jsou v duchu iteračního a přírůstkového životního cyklu. Rozpracovávají jeho jednotlivé fáze, a to specifikaci požadavků, analýzy, designu, implementace, testování a nasazení. Svými kroky jsou si tedy podobné, liší se však hloubkou metodiky a používanými notacemi pro popis systému.

6.2 Objektově orientované metodiky 1. generace

Objektově orientované metodiky 1. generace se vyznačují mnoha neduhy právě zrozené a nedozrálé technologie. V současnosti se používají obvykle pouze ze setrvačnosti. Jejich největším problémem je, že všechny mají odlišnou notaci. Dále na sobě ještě nesou známku ještě čerstvě opuštěného strukturálního přístupu.

Schlear-Mellorova

Metodika Shlaera a Mellora patří k prvním přístupům k objektově orientovaným metodologiím. Byla prezentovaná již v roce 1988 v jejich knize Object-Oriented Systems Analysis – Modeling the World in Data and Object Lifecycles: Modeling the World in States (1988).

Coad-Yourdonova

Coad a Yourdon prezentovali svoji metodiku ve své knize Object-Oriented Analysis and Object-Oriented Design (1990).

Autoři zdůrazňují úlohu objektové orientace zejména kvůli zlepšení komunikace mezi analytiky a experty na problémovou oblast, zvýšení konzistence mezi analýzou, designem a programováním, znovupoužitelnosti analýz, designu a programových výsledků atd.

Analýza zahrnuje pět vrstev nazývaných SOSAS (podle počátečních písmen termínů reprezentujících každou fázi). Ve fázi designu je těchto pět vrstev transformováno do čtyř komponent.

V Coad-Yourdonově metodice je nepříjemný posun v terminologii (jde o pojmy objekt a třída). Objekt je abstrakcí entit, zapouzdřuje jejich atributy a služby (čili z pohledu UML se jedná o třídu). Třída je pak popisem jednoho nebo více objektů se stejnou množinou atributů a služeb, přičemž zahrnuje i popis způsobu tvorby nového objektu ve třídě (jedná se tedy o jakýsi kontejner).

Fusion

Metodika Fusion byla vyvinuta u Hewlett-Packard Derekem Colemanem, který vedl tým ve Velké Británii (1990).

Cílem bylo vytvořit jednoduchou, přitom však komplexní metodu návrhu informačního systému. Tato metodika vznikla kombinací myšlenek jiných metod, a to Booche, Jacobsona, Rumbaugha a dalších. Zapracovány byly hlavně ty myšlenky, které měly největší využití v praxi. Metodika Fusion je zaměřena na analýzu a návrh a kombinuje několik vlastností předchozích metod.

OMT (Rumbaugh)

Metodiku popsal James Rumbaugh v knize Object-Oriented Modeling and Design (1991).

Obsahuje celou řadu myšlenek a přístupů, které jsou důležité pro analytiky a designery. OMT byla a je velmi oblíbenou metodikou analýzy a návrhu, dokazuje to i fakt, že mnoho nových metodik vychází právě také z ní (např. Select Perspective). OMT zahrnuje jak notaci (jednotlivé použitelné diagramy), tak i popis objektově oriento-

vaného vývoje informačního systému.

OOSE (Jackobson)

Jacobsonova metodologie (Objectory) byla publikována v knize Object-Oriented Systems Engineering (1992).

Snaží se podpořit celý životní cyklus vývoje softwarového produktu. Jednodušší verzí Objectory je OOSE (Object-Oriented Software Engineering), kterou Jacobson vytvořil jakožto metodologii pro tvorbu aplikací. Přináší do objektově orientované analýzy Use Case Diagram.

OOAD (Booch)

Grady Booch popsal svůj přístup k OOAD ve své knize Object-Oriented Design with Applications (1994).

Tato metodika se skládá ze čtyř hlavních aktivit a šesti notací. Pokrývá oblasti analýzy požadavků a analýzu reality, avšak hlavní důraz je kladen na design. Z metodiky vyzařuje i silný vztah přímo k programování.

6.3 UML

Používání odlišných notací působilo problémy na trhu, proto tato metodologická válka logicky vyústila sjednocením notací metodik OOAD, OMT a OOSE v roce 1996, a tak vzniklo UML. Samotní autoři UML uvádějí (Booch, Rumbaugh a Jackobson), že UML není odklonění od jejich specifických metodik, nýbrž že se jedná o jejich následníka. UML se tedy snaží tyto různé myšlenky uspořádat do soudržné struktury. UML je v dnešní době dominantní notací v oblasti modelování analýzy a designu.

Základní myšlenka jazyka UML je podle jejich autorů tato: Zaměřením se na podstatné rysy při současném ignorování jiných vlastností je podstatou abstrakce a je klíčem k poznávání a komunikaci. Důvody pro to jsou následující. Každý komplexní systém je nejlépe uchopitelný pomocí malé skupiny relativně nezávislých pohledů na systém. Žádný jednotný pohled není dostatečný. Každý model by

DIAGRAM	Statický	Specifikační	Dynamický
Diagram případů užití		ANO	
Diagram tříd	ANO		
Stavový diagram			ANO
Diagram aktivit			ANO
Diagram spolupráce			ANO
Sekvenční diagram			ANO
Diagram komponent	ANO		
Diagram nasazení	ANO		

Tabulka 6.1: diagramy UML

měl být vyjádřen na různé úrovni výstižnosti. Nejlepší modely mají přímé spojení s realitou. Základní cíle UML jsou:

Poskytnout uživatelům použitelný vizuální modelovací jazyk pro vytváření a výměnu smysluplných modelů.

Poskytnout mechanismy rozšiřitelnosti a specializace pro rozšíření základních konceptů (princip stereotypů).

Vytvořit specifikaci nezávislou na konkrétních programovacích jazycích a procesech vývoje a analýzy, pouze s omezením na objektově orientovaný přístup.

Poskytnout formální základ pro pochopení modelovacího jazyka.

Podporovat rozvoj objektových nástrojů sjednocením notací a vytyčením standardu.

Podporovat vývojové koncepty jako jsou komponenty, frameworky a návrhové vzory.

Zahrnout bohaté zkušenosti jejich tvůrců.

6.4 Objektově orientované metodiky současnosti

Všechny moderní metodiky současnosti v objektově orientovaném světě jsou založené na UML. UML je ovšem pouze notace, nikoliv

metodika. Tímto se dostáváme k metodikám druhé generace, které jsou založené na UML. Jejich společným rysem je také rostoucí orientace na vytváření systémů komponent a používání distribuovaných systémů.

Z těchto metodik druhé generace v této práci uvádím metodiku Select Perspective (podpořená nástrojem Select Enteprise) a Rational Unified Process (podpořená nástrojem Rational Rose). Nabízejí totiž komplexní postup, jak přejít od poznávání systému k jeho pochopení až k vytvoření podkladů vhodných pro implementaci. Začínají analýzou business processes, následuje vymezení systému, okolí a požadovaných funkcí (use cases). Obsahuje modelování struktury prvků systému (definovaní tříd a vazeb mezi nimi v diagramu tříd) a modelování stavů, kterými tyto prvky systému v průběhu svého životního cyklu procházejí. Následuje znázornění aplikační logiky, toho, jak probíhá komunikace mezi objekty (diagramy interakcí a spolupráce). Vše je zakončeno návrhem komponent v duchu komponentového vývoje (CBD). Pro analýzu a návrh databáze se nabízí využití ERD.

Select Perspective

Metodika Select Perspective je velmi pěkně popsána v knize Stručný úvod do jazyka UML (viz. zdroj [6]).

Vychází z kombinace OMT (James Rumbaugh) a OOSE (Ivar Jacobson). Současná verze metodiky nabízí podporu jednotlivých fází komponentově orientovaného vyvoje (Component Based Development – CBD), tj. tvorby komponent, řízení komponent a tvorby aplikací z komponent.

Select Perspective zahrnuje kromě UML i modelování firemních procesů a datové modelování. Jejich tvrzení je, že samotné UML nestačí, je třeba ho doplnit o tyto dva rozšířené přístupy. Základní CASE Select Enerprise podporující metodiku Select Perspective například diagram aktivit vůbec neobsahuje.

Select Perspective staví na sedmi základních principech:

> Vazba k oboru podnikání (Linkage to Business), Select Perspective začíná znázorněním procesů a z těch pak vychází další analýzy.

Komponentově orientovaný vývoj.

Integrované modelovací techniky, a to modelování business processes, datové modelování a část UML.

Iterativní, inkrementální vývoj.

Paralelní vývoj, v souvislosti s CBD může paralelně pracovat několik týmů vyvíjejících jednotlivé komponenty.

Architektura založená na službách (Service-Based Architekture), částí Select Perspective jsou Perspective Patterns, což jsou připravené rámce pro technickou a business architekturu.

Programové nástroje pro návrh systému, zprávu komponent, automatické generování kódu atd.

RUP (Rational Unified Process)

Metodika RUP je pěkně popsána v knize UML a unifikovaný proces vývoje aplikací (viz. zdroj [5]).

RUP byla vytvořena autory UML (Booch, Rumbaugh a Jackobcon), a proto si tato metodika vystačí čistě s UML (krom možnosti modelování relačních databází v ERD místo modelu tříd). Metodiku RUP autoři popisují v knize The Unified Software Development Process (1995).

Základní východiska RUP jsou:

Iterativní vývoj softwaru (Develop Software Iteratively). Klasický vodopádový model (sekvenční fáze vývoje) na dnešní rozsáhlé systémy nelze bez problémů použít. V každé fázi vývoje se pozornost obrací na klíčové problémy, výsledkem fáze jsou snadno testovatelné produkty (zřetelné cíle, spustitelný software). Iterativnost celého procesu dává možnost snadno promítnout změny do systému.

Snadná správa požadavků (Manage Requirements). RUP jasně definuje prostředky a způsob dokumentace požadované funkcionality – model jednání a modely objektové interakce jsou zde důležitým prostředkem.

Využití komponent (Use Component-Based Architectures). Využití stávajících komponent (částí s jasně definovaným účelem a rozhraním) je značným urychlením procesu vývoje, popřípadě znamená urychlení vývoje v budoucnu, pokud komponentu vyvíjíme sami.

Důraz na vizualizaci vyvíjeného softwaru (Visually Model Software). Moderní nástroje poskytují vhodný (vizuální) pohled na systém a dovolují využít další, snadněji ovladatelné produkty (různá grafická IDE). UML je ideální jazyk pro zachycení modelů a následné počáteční generování kódu.

Testování (Verify Software Quality). Testování je nedílnou součástí vývoje softwaru a RUP se snaží objektivními technikami podporovat testování během celého procesu vývoje.

Snadné zapracování změn (Control Changes to Software). Iterativní vývoj vyvolává množství změn, které je nutné ihned promítnout do systému – RUP poskytuje vhodné řídící nástroje (od obecných premis až po konkrétní kroky), které pomohou udržet vše v přijatelných mezích a pod kontrolou.

Když se pozorněji podíváme, všimneme si, že metodiky Select Perspective a RUP jsou si velmi podobné. Kdybychom sledovali jiné moderní metodiky vývoje objektově orientovaného software, mohli bychom vysledovat jakousi společnou bázi.

Metodiku tvoří skupina integrovaných modelů, které popisují aktivity, úlohy a techniky vývoje, včetně metod zabezpečení kvality software a konfiguračního i projektového managementu. Možná, že se nabízí pronést kacířské prohlášení, že se o metodikách mluví víc, než si zasluhují. Není jádro všech metodik totožné? Nejde nakonec pouze o různé diagramy aktivit složené ze stejných činností, pouze poskládané v jiném pořadí?

Kapitola 7

Softwarová podpora analýzy

7.1 Prezentace modelů

Diagramy je možné samozřejmě kreslit rukou na kus papíru, ale je to velmi neefektivní (především z důvodu editace a aktualizace) a především je rukou velmi pracné vytvořit diagram, aby jej bylo možné prezentovat před zákazníkem nebo ostatními členy týmu. Naštěstí na trhu existují kvalitní a neustále rozvíjené CASE a CABE systémy.

Výběr vhodného CASE systému (Computed Aided Software Engeneering) je ideálním řešením, pokud se chystáme vytvářet softwarový produkt. Jestliže je naším záměrem spíše než návrh softwarového produktu vytvoření komplexního a úplného modelu podnikových procesů, je potřeba použít specializovaný produkt na business modelování nabízející možnost hierarchického rozpadu procesů, hierarchii podřízenosti, katalog pracovních míst apod. Tyto produkty bývají označovány jako CABE (Computed Aided Business Engeneering). CASE systémy mívají sice obvykle jistou základní podporu pro BPR (business process modelling), ale tato podpora je u takto specializovaných projektů obvykle nedostačující.

Přestože jsou tyto prostředky poměrně drahé (ať už CASE nebo CABE), samy o sobě nestačí. Nejsou totiž ničím jiným, než pouze nástroji. Nástroj může využít svůj potenciál až v rukou dovedného uživatele.

Systém CASE (CABE) se skládá ze tří základních složek:

grafický (a textový) procesor

funkční obsluha

databáze

Grafický (a textový) procesor

Základní vlastností každého CASE nebo CABE systému, která by nás při zvažování jeho pořízení měla zajímat, je grafická podpora zamýšlených typů diagramů a jejich kvalitní vizualizace. Dále by měl mít možnost diagramy exportovat do obrázků různých používaných formátů a generovat dokumentaci k diagramu (například ve formě HTML).

Funkční obsluha

Další velkou výhodou CASE (CABE) systémů je možnost generování fyzického implementačního skriptu z logického modelu (například Together dovoluje vygenerovat z diagramu tříd skutečné třídy podle zvoleného programovacího jazyka, dále Power designer dovoluje vygenerovat z fyzického datového modelu SQL skript pro vygenerování databáze apod.). To je vhodné zejména proto, že je tímto způsobem snadné udržovat konzistenci mezi jednotlivými modely (v tuto chvíli mám na mysli logický a fyzický model). To analytik ocení především ve chvílích, kdy například dolaďuje databázi. Stačí, když změnu učiní pouze v diagramu v CASE a ta se automaticky přenese do fyzického modelu. Bude-li analytik dopisovat změny do obou modelů ručně, hrozí při opomenutí nekonzistence obou modelů. Toto souvisí také s Principem odstranění redundance, uvedeném v příslušné kapitole této DP.

Databáze

Databáze je zcela určitě nejdůležitějším rysem CASE nebo CABE systému. Rozhoduje o tom, jestli budeme skutečně schopni v diagramech zachytit vše, co budeme potřebovat a co budeme chtít. Každý prvek nebo vztah, který graficky zobrazíme, se se svými charakteristikami uloží do databáze. Databáze poskytuje jistá usnadnění naší práce a pomáhá nám nedopouštět se triviálních chyb, například nás varuje, abychom nepoužívali stejný název pro různé prvky. Zásadní věcí, kterou databáze daného CASE nebo CABE přímo ovlivňuje, je to, zdali nás nenutí při modelování lhát. Pochopitelně nástroj, který nás nutí lhát není vhodný. Při pokusu o odhalení kvality této databáze

hledejme výjimečné situace, které nastávají pouze zřídkakdy. Schopnost nástroje vypořádat se s těmito výjimečnými situacemi ukáže použitelnost nástroje v reálných aplikacích. Jsou to otázky typu:

> Je možné v ERD odkazovat z asociativní tabulky na další asociativní tabulku? (což například Power Designer nesplňuje)
>
> Je možné zavést na úrovni konceptuálního modelování v ERD vícenásobnou dědičnost? (což například Oracle Designer nesplňuje)
>
> ...apod.

Databáze dále rozhoduje, jestli umí CASE pracovat se souvislostmi mezi jednotlivými modely.

7.2 CASE systémy

Tyto nástroje hrají důležitou roli v etapách analýzy, návrhu a částečně implementace softwarového projektu. Nyní se pokusíme porovnat jednotlivé CASE systémy na základě toho, které modely podporují a podle jejich funkcionality. Porovnání jejich databází, a tedy schopností modelovat výjimečné situace, je bohužel za hranicí této práce. Porovnáme pouze špičkové CASE, které podporují jak fázi analýzy, návrhu, tak fázi implementace. Všechny obsahují nástroje pro správu verzí a repository a jsou na dnešním trhu široce rozšířené. Dále všechny obsahují již zmiňovanou funkcionalitu Round trip engineering, což znamená, že umožňují provádět změny ve vygenerovaném kódu, aniž by se narušila vazba mezi vygenerovaným kódem a návrhem systému. Jestliže programátor provede změnu v kódu (např. přidá ve třídě další metodu), zanese se tato změna automaticky do diagramu, kde je tato třída modelována (a naopak) – toto je velmi důležité, protože jinak by hrozilo, že budeme mít model a kód nekonzistentní (viz. kapitola Analytické principy).

Select Enterprise

Poskytuje přímou podporu pro metodologii Select perspektive. Protože metodika Select perspective má vlastní diagramy na BPM, ne-

podporuje UML diagram aktivit. Obsahuje prostředí pro kreslení obecných diagramů (něco podobného jako v MS Visio), takže je možné v tomto CASE vizualizovat diagramy, pro které nebyl primárně určen (DFD, FH apod.).

Rational Rose

Poskytuje přímou podporu pro metodologii RUP. Zajímavostí je možnost vyjádřit diagramy ve třech různých notacích, a to nejen v UML, ale i OOAD nebo OMT (kupodivu některé firmy stále ještě tyto »stařík«y používají).

Together

Poskytuje plnou podporu pro objektově orientované metodiky. Mimo UML má ještě podporu notace podle Coad-Yourdonovy metody.

Power Designer

Poskytuje částečnou podporu pro objektově orientované metodiky a také částečnou podporu pro procedurální metodiky. Z diagramů UML podporuje jenom část diagramů (například vůbec nepodporuje stavový diagram nebo sekvenční diagram). Jeho výhodou je, že má velkou podporu databází při vytváření skriptů pro jejich založení (například i pro některé freewarové databáze jako PostgreSQL nebo MySQL).

Oracle Designer

Poskytuje pouze plnou podporu pro strukturální metodiky. Primární úlohou tohoto CASE je podpora tvorby databázové aplikace.

Legenda k tabulkce porovnání CASE systémů:

UML reprezentuje podporu pro všechny UML diagramy,

ERD reprezentuje podporu pro entity-relationship diagram, který se používá na modelování databází,

CASE systém	**Rational Rose**	**Together**	**Select Enterprise**	**Power designer**	**Oracle designer**
UML	ANO	ANO	téměř úplné	částečné	NE
ERD	ANO	NE	ANO	ANO	ANO
DFD	NE	NE	Nepřímo	NE	ANO
FD+CRUD	NE	NE	Nepřímo	NE	ANO
EPC+PH	NE	NE	ANO	ANO	ANO
generátor kódu	ANO	ANO	ANO	ANO	ANO
reverz kódu	ANO	ANO	ANO	ANO	ANO
generátor DBS	ANO	NE	ANO	ANO	ANO
reverz DBS	ANO	NE	NE	ANO	ANO
multidim. Model	NE	NE	NE	ANO	ANO
generátor dok.	HTML	HTML	MS WORD	HTML	Slabý

Tabulka 7.1: tabulka porovnání CASE systémů

DFD značí podporu klasického data-flow diagramu, který se používal během strukturované analýzy jako dynamický model,

FD je diagram funkční dekompozice (hierarchie), používá se v některých strukturovaných metodikách,

CRUD je matice používaná ve strukturované analýze pro ověření základní funkcionality nad každou entitou v systému,

EPC je diagram procesních řetězců, používá se během BPM (business process modelling),

PH je diagram hierarchie procesů (BPM),

generátor kódu a reverz kódu je funkcionalita podporující již zmiňovaný Round trip engineering pro model tříd,

generátor DBS a reverz DBS je funkcionalita podporující již zmiňovaný Round trip engineering pro ERD (nebo případně model tříd popisující relační databázi),

multidim. Model určuje zdali má CASE podporu pro technoogii OLAP pro tvorbu datových skladů,

generátor dokumentace ukazuje jakým způsobem umí CASE vytvářet k diagramům dokumentaci.

Správný CASE vhodný pro vývoj objektově orientovaného podnikového software by měl podporovat:

modely UML pro zachycení architektury software

základní možnosti pro zachycení procesního modelování (Vývoj podnikového software je nutné do jisté míry vázat přímo na konkrétní podnikovou strukturu, navíc procesní modelování dává analytické informace do kontextu a často slouží jako vhodný výchozí bod projektu.)

generátory kódu a databáze včetně reverzních generátorů k zajištění Round-trip engeneeringu

7.3 CABE systémy

Pro modelování struktury a cílů organizace se používají nástroje pro business engineering (CABE – Computer Aided Business Engineering).

CABE jsou specializované produkty pro podporu BPM (business process modelling). Tyto systémy jsou velmi užitečné v projektech jako je BPR (business process reingeneering), zavádění workflow management systému nebo tvorby informační strategie. Porovnání těchto systémů je pouze velmi rámcové a rozhoduje především to, jaké modelovací techniky daný CABE podporuje.

Aris Toolset

Velmi kvalitní a populární CABE, který pokrývá všechny (nebo většinu) potřeby projektů BPR, zavádění workflow nebo tvorby informační strategie. Podporuje všechny složky BPM.

CABE systém	Aris Toolset	QPR ProcessGuide
EPC a PH	ANO	ANO
ORGS, ORGP a KPM	ANO	slabé
IDM	ANO	slabé
FD	ANO	NE

Tabulka 7.2: tabulka porovnání vybraných CABE systémů

QPR ProcessGuide

Jednodušší CABE, omezený trochu na procesní modelování. Další složky BPM modeluje pouze velmi rámcově.

FirstStep Designer, Provision Workbench, Corporate Modeler

Další CABE systémy, v této práci se těmito nástroji nebudeme podrobněji zabývat.

Legenda k tabulkce porovnání CABE systémů:

EPC a PH jsou procesní modely (diagram procesních řetězců a diagram procesní hierarchie).

ORGS, ORGP a KPM jsou diagramy pro modelování organizační struktury podniku (organizační struktura, schéma podřízenosti, katalog pracovních míst).

IDM je ideální datový model.

FD je diagram funkční dekompozice (hierarchie).

CABE nástroje by měly dále podporovat zachycení a modelování následujících oblastí:

Globální cíle podniku, tj. možnost jejich zachycení, napojení cílů podniku na modelované procesy (hlavní procesy v podniku mají vést k naplňování jeho cílů).

Organizační struktura, tj. možnost jejího znázornění, možnost mapování procesů na organizační strukturu (za procesy zodpovídají jejích vlastníci, kteří zastávají zároveň nějakou pozici v organizační struktuře).

Topologie, tj. možnost zachycení geografické struktury podniku, prvků okolí podniku a možnost vazby mezi prvky okolí a procesy.

Hierarchie procesů, tj. možnost jejího zachycení, možnost definovat subsystémy a kontrola konzistence vzhledem k organizační struktuře.

Ohodnocení procesů, tj. možnost procesy hodnotit i možnost definice vlastních metrik pro jejich hodnocení.

Vzorové šablony procesů, tj. referenční modely. Referenční modely se zabývají popisem funkcí a procesů podniků. Popis funkcí a procesů se pohybuje na dost obecné úrovní (nejsou zachyceny rozmanitosti podniků, modely jsou pro obecné podniky).

7.4 Další kritéria pro výběr vhodného nástroje

Kromě již zmiňovaných parametrů v tabulce je vhodné při výběru CASE nebo CABE sledovat:

Podporuje nástroj integraci s ostatními nástroji potřebnými pro projekt?

Je nástroj přístupný modifikacím? (Neexistuje nástroj, který by vyhověl všem projektům a všem nárokům uživatelů. Každý projekt vyžaduje určité úpravy nástrojů. Nástroj by tedy měl obsahovat nějaký nástroj, kterým je možné ho upravovat.)

Podporuje nástroj integraci s ostatními nástroji potřebnými pro projekt?

Podporuje autorizovaný přístup uživatelů (úroveň přístupu podle oprávnění)?

Podporuje nástroj kontrolu konzistence v rámci diagramu? (například v diagramu datových toků nedovolí nakreslit datový tok z datového skladu opět přímo do datového skladu, bohužel některé nástroje mají podobné omezení například na propojení dvou asociativních entit u ERD, což bohužel není žádoucí.)

Podporuje kontrolu konzistence mezi jednotlivými diagramy v rámci celého projektu a v rámci používané metodiky? (např. zda je pro každý Use Case definován sekvenční digram.)

Jaká je cena nástroje?

Jakou technickou a uživatelskou podporu nástroj má?

Kapitola 8

Typy projektů v IT

8.1 Rozložení složitosti

Dnes je možné vysledovat několik hlavních proudů a nejvíce poptávaných softwarových projektů. Každý z nich je charakterizován jiným rozložením složitosti (tedy toho, v jaké oblasti IT bude potřeba klást největší důraz a kde bude potřeba mít nejkvalitnější pracovní personál).

Rozložení složitosti nás bude zajímat na úrovni:

Databázová úroveň – aplikace obsahuje množství dat, která jsou často v komplikovaném vztahu. Navíc je pravděpodobné, že aplikační oblast je nestabilní, proto je nutné dělat databáze na obecnější úrovni. Typickým zástupcem je informační systém podniku.

Algoritmická úroveň – aplikace obsahuje netriviální algoritmy, často založené na komplikovaných matematických a fyzikálních modelech. Typickým zástupcem je řídící systém.

Grafická úroveň – na aplikaci jsou požadovány vysoké grafické nároky (buď na 3D nebo 2D grafiku). Typickým zástupcem je simulátor nebo hra.

Projekty v podnikové sféře

Společným rysem na úrovni podnikových projektů je jejich veliká komplikovanost a složitost na úrovni databáze. Graficky nebo algoritmicky jsou většinou spíše jednodušší (na rozdíl třeba od počítačových her, které potřebují často silný grafický engine, nebo řídících

systémů, které mají pod sebou poměrně malou databázi, ale mohou obsahovat často velmi složité algoritmické řešení).

Je bohužel mimo rozsah této práce porovnávat i jiné typy projektů (operační systémy, grafické aplikace a hry...), proto se v další části této kapitoly zaměříme pouze na projekty v podnikové sféře.

8.2 Typy projektů v podnikové sféře

Operační informační systém (OIS)

OIS je klasický informační systém podporující každodenní chod podniku. Někteří manažeři se mylně domnívají, že hlavním smyslem zavedení operačního systému je snížení nákladů. Zavedení OIS jen zřídka kdy ušetří náklady (naopak jeho zavedení stojí podnik mnoho peněz). Hlavním účelem je získání konkurenční výhody poskytnutím nové služby svým zákazníkům (online systém rezervací, prodeje, poskytování informací, informační terminály apod.). Dále slouží jako stabilní základ pro manažerský informační systém (MIS) a systém workflow, které dále mohou výrazně rozšířit jeho vliv na prosperitu podniku.

Pro implementaci OIS je jako základní výbava výkonná relační databáze, vývojové prostředí a kvalitní CASE.

Zavádění strategického (manažerského) informačního systému (MIS)

Jako nadstavba nad OIS je MIS. Hlavním cílem projektu MIS je dát podniku strategickou výhodu proti ostatním podnikům. Manažerský informační systém by měl sloužit k analýzám současných dat a dat z minulosti a na jejich základě odhadovat budoucí trendy. Slouží tedy pro podporu strategického plánování podniku.

Na implementaci MIS je často vhodné použít technologii OLAP (multidimenzionální databáze). Váže se k technologiím jako DataWerehousing, DataMining, KnowledgeDiscovery. Jako matematický aparát používá matematickou statistiku. Je-li na implementace MIS použita klasická relační databáze, obsahuje narozdíl od dobře navržené OIS databáze mnoho dopočítaných dat. U manažerských informačních systémů ovšem platí poněkud jiná pravidla při jejich konstrukci

než pro klasické OIS a jejich popis přesahuje rámec této práce.

Zavádění workflow management systému do podniku

Primárním účelem zavedení systému workflow je podpora a automatizace vykonávání procesů ve firmě. Hlavním efektem workflow systému je nejen díky analýze zavedení pořádku ve firmě (zavedení systému workflow obvykle předchází reengeneering obchodních procesů firmy nebo alespoň jejich revize), ale také dramatické snížení nákladů (eliminace středního managementu, zrychlení průběhu některých procesů) a zvýšení efektivity procesů díky jejich správnému návrhu a automatizaci a tedy i získání konkurenční výhody zefektivněním procesů.

Na implementaci workflow systému je nezbytné použít kvalitní CABE systém, který slouží pro definování procesů. S těmito definicemi pak dále pracuje výkonné jádro workflow.

Zavádění workflow systému obvykle předchází projekt BPM a BPR, případně IST.

BPM, BPR, IST

Velmi zjednodušeně se jedná o projekty týkající se optimalizace procesů v podniku. BPM (business process modelling) je metoda velmi komplexního zachycení vazeb a struktur v podniku (za pomoci procesního, funkčního, datového modelování a modelování organizačních struktur v podniku) a je nezbytným výchozím bodem pro BPR (business process reengeneering) a IST (definice informační strategie podniku).

BPR je radikální přestavba procesů v podniku, kde je zachován pouze vstup a výstup procesu, ale celé jádro procesu je vybudováno znovu.

Pro implementaci těchto projektů není ani tak důležitý CASE systém (žádný software zatím nevyvíjíme), ale kvalitní CABE systém.

8.3 Softwarový balík nebo řešení přímo na zakázku?

Manažeři podniků musí před zavedením podnikového software volit mezi zásadním rozhodnutím. Softwarový balík nebo řešení na

Typ projektu	**Zakázka**	**Customizovaný**
cena zavedení	vysoká	střední
cena údržby	velmi vysoká	střední
doba trvání zavedení	vysoká	střední
konkurenční výhoda	ANO	spíše NE

Tabulka 8.1: tabulka porovnání zakázkového projektu a customizace

zakázku? Není záměrem této práce důkladně prozkoumat tuto problematiku, proto jsem se omezil na stručné porovnání v tabulce (viz. výše).

Kapitola 9

Analytické principy

9.1 Je analýza inženýrskou disciplínou nebo uměním?

Obecně lze říci, že jednotlivé metodiky se pokouší o převod komplikovaného paralelního modelování do jednotlivých sekvenčních kroků. To je jistě krok správným směrem a připomíná to již zmiňovanou inženýrskou metodu (metodika se snaží převádět komplikovanou analýzu na sled snadno pochopitelných sekvenčních kroků). Ale myslím, že profesionální analytik by měl jít více do hloubky problému. Nejen proto, že IT je stále ještě nevyzrálé dítě, ale proto, že takovýto přístup zbavuje analytika do jisté míry nadhledu (což je jedna ze tří základních analytických vlastností, jak jsem uváděl dříve). Jako příklad vezměme normální formy, které se používají pro návrh databází. Je to sestava pravidel, které při jejich aplikování dopomohou k poměrně kvalitní databázi. Je samozřejmě možné je bezmyšlenkovitě mechanicky aplikovat a pravděpodobně dostaneme na výstupu »vcelku«dobrou databázi. A to je právě kámen úrazu. Analytická profese je totiž spíše než exaktní vědou uměním. A přestože existuje například teorie malířské a fotografické kompozice, zeptejte se některého umělce, co si o ní myslí... Znalost těchto pouček může být užitečná, ale je nutné si uvědomit, že jsou jenom omezeným odrazem pravdy. Například za každou normální formou se v hloubi skrývá jeden nebo více mocných analytických principů, které vychází přímo z logiky. Schopnost logického uvažování by měla být nejdůležitější schopností každého analytika a pouze manuální zvládnutí nějaké metody považuji za nedostatečné.

Dále uvedené analytické principy budeme prezentovat zejména na tvorbě datového modelu ERD. Ve skutečnosti je ale vhodné je

používat i v dalších oblastech vývoje SW – jako například návrh uživatelského rozhraní, návrh modelu tříd apod. Ovšem během tvorby datového modelu je jejich dopad nejvýraznější.

Není vždy důležité všechny následující principy dodržet. Jsou spíše vodítkem a kdykoliv zjistíme, že některý z principů porušujeme, měli bychom si uvědomit, proč to děláme. Občas může být z různých důvodů vhodné něco udělat trochu »nečistě«z důvodů zvýšení výkonnosti apod. Měli bychom mít ale stále na mysli, že každé místo, kde se odkloníme od analytických principů a které uděláme »natvrdo«, je krásným potenciálním místem pro příslovečné »zamávání motýlích křídel«. Následující principy jsou v podstatě rozšířením myšlenek, které stojí za normálními formami pro tvorbu databází. Pro normální formy také platí, že nemá smysl je za každou cenu dodržet. Opět může být občas výhodné je porušit, ale musíme vědět proč.

V předchozích kapitolách jsme se bavili mimo jiné o hledání jádra analytického modelu. Pravdou sice je, že nikdy přesně nevíme, zdali jsme jej už dosáhli či nikoliv, ale podle mého názoru o něm určitě můžeme říct to, že splňuje všechny následující principy (protože dodržování následujících principů přirozeně přispívá k »čistému«návrhu):

Triviální princip pravdivého zobrazení

Princip odstranění redundance

Princip jednotné sémantiky

Rozšířený princip abstrakce pro dosažení flexibility

Princip osvědčených řešení

9.2 Triviální princip pravdivého zobrazení

O tom, co je pravda, se zde bavit nebudeme. Nechme ji na intuitivní úrovni jako něco abstraktního, ale pouze s jediným výskytem, a přijměme fakt, že komunikace slouží k přibližování se k dané pravdě. A to je právě úkolem analytika. Skrze komunikaci se dostatečně přiblížit pravdě.

Pravdou máme na mysli skutečné a reálné zachycení vztahů v reálném světě v modelu aplikace (a tedy následně i v aplikaci). Pokud naše aplikace (v tomto případě především model tříd nebo model databáze) nebude odpovídat pravdě (jak informační silou, tak především odpovídajícími funkčními závislostmi), nebude do ní možné vyplnit správná data.

U abstrakce je zcela běžné, že některé aspekty reality úmyslně zanedbáme, ale nikdy bychom neměli pravdu zkreslit (což znamená v podstatě lhát). V jednoduchosti to znamená například to, že pokud něco je entitní množina, tak s tím je třeba zacházet jako s entitní množinou (a nikoliv atributem – což odpovídá 1.NF podle Codda). Pokud je něco vlastností entity nebo vztahu, tak to taky nechat jako atribut u dané vlastnosti nebo vztahu (což v podstatě říká 2.NF). Kdykoliv se dopustíme zkreslení, hrozí podobné anomálie jako při porušení 1.NF nebo 2.NF, což je popsáno v příloze této práce.

Význam konzultací

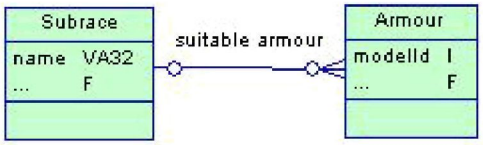
Smysl konzultací je právě v tom, že se nám podaří dostat se dostatečně blízko k pravdě daného odvětví a budeme ji schopni zachytit pomocí modelu. Dobrý analytik musí být schopen objektivního přístupu a znát prostředky přirozeného jazyka, jak co nejvíce vytěžit z omezeného prostoru konzultace. Obsáhnout takto celou aplikační oblast je ovšem velmi obtížné a přestože existují velmi zajímavé komunikační postupy, které k tomu dopomáhají, v této práci se jimi zabývat z nedostatku prostoru nebudeme.

Proč je těžké tento princip dodržet, když je tak triviální

Také je typické, že hlavní programátor to tom, co uvidí náš model řekne, že je to celé příliš složité. Je sice možné, že toto tvrzení může být dost dobře příznakem, že jsme ještě neobjevili jádro analytického modelu, ale také je velmi dobře možné, že aplikační oblast je sama o sobě složitá, takže ani její, řekněme, model tříd nemůže být jednoduchý. Jako analytici můžeme být nuceni realitu deformovat a nemístně zjednodušovat, čímž deformujeme pravdivé zobrazení reality v našem modelu a to se nám projeví v důsledku tak, že aplikace neodpovídá reálným požadavkům.

Příklady

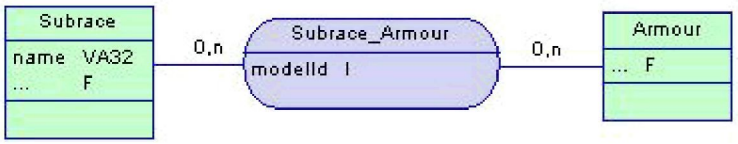
Jako příklad jsem zvolil část raálné databáze pro podporu vývoje počítačové hry. Typickým případem je ve spěchu vytvoření vztahu se špatnou násobností. Mezi tabulkou Subrace (která reprezentuje různé tvory, například různé mimozemské protivníky, ale také i ženu a muže) a Armour opravdu »obvykle« platí vazba 1-n, ale jsou celkem výjimečné případy, kdy zbroj sdílí více ras (například žena a muž). Jestliže nemáme připravenou strukturu na tento typ vztahu, budeme při používání této databáze nadefinovat v tabulce Armour extra zbroje mužské a ženské, které mají ovšem stejné parametry (data se stávají redundantní). A hrozí nám tedy nekonzistence dat. Navíc v tomto systému měla tabulka Armour ještě dva předky, takže možnosti, kde může nastat nekonzistence se ještě rozrostly.



Obrázek 9.1: ERD před aplikováním principu pravdivého zobrazení

9.3 Princip odstranění redundance

Redundance je typický rys hojně se vyskytující v datech i kódu dnešních aplikací. Redundance znamená, že máme v kódu některá nadbytečná místa (stejná informace nebo stejná část kódu). To co v nich je uložené, je buď uložené i někde jinde nebo je z něčeho jiného odvoditelné. Zdánlivě je na redundanci špatné to, že naše aplikace je kvůli redundanci větší, než by musela být. To ale není hlavní problém redundance. Mnohem větším problémem je to, že hrozí, že se naše aplikace stane nekonzistentní (ať už v kódu nebo v datech).



Obrázek 9.2: ERD po aplikování principu pravdivého zobrazení

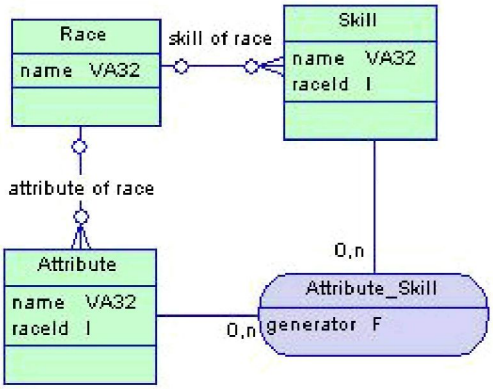
Každou novou změnu v kterékoliv z postižených oblastí totiž musíme zanést na několik míst najednou (ať už v kódu nebo v datech). To přináší obrovskou zbytečnou režii ke každé změně (a tedy i náklady na údržbu). Pokud tak neučiníme (a u středních nebo větších systémů si buďme jisti, že to občas opomeneme), tak se stává náš kód/data nekonzistentní. Nekonzistence sebou nese špatné chování aplikace, nelogické a nesmyslné výpisy apod.

Datovou redundanci je možné odstranit dodržováním 3. normální formy.

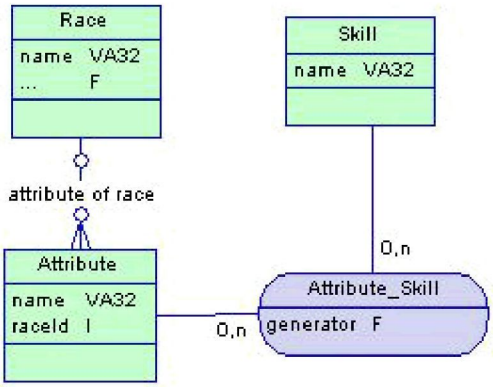
Redundanci v kódu aplikace je možné odstranit opět dodržením 3. normální formy u dekompozice na třídy, dále používáním rodičovských tříd a abstraktních předků pro společný kód potomků (pokud jsme odpůrci dědičnosti, tak můžeme společný kód umístit do speciální třídy, kterou pak z ostatních agregujeme). U procedurálního programování odstraňuje redundanci používání procedur.

Příklady

Ukázka je opět ze světa počítačových her a prezentuje, že odstraňování redundance je důležité nejen uvnitř tabulek (jako to učí 3.NF), ale také v rámci celé databáze. Ve zvoleném příkladu je nadbytečná vazba z tabulky Skill do Race, protože ji je možné odvodit z jiných vazeb. Pokud by tato vazba zůstala, mohlo by se stát, že by se data stala nekonzistentní.



Obrázek 9.3: ERD před aplikováním principu odstranění redundance



Obrázek 9.4: ERD po aplikování principu odstranění redundance

9.4 Princip jednotné sémantiky

Jednotnou sémantikou mám na mysli například to, že o každém sloupci databáze můžu s klidným svědomím říci, k čemu slouží, jaké hod-

noty se v něm uchovávají a co vyjadřují. Často můžeme být svědky, že se v jednom sloupci databáze uchovávají naprosto rozdílná data, která mění svoji sémantiku v závislosti na některých zvláštních podmínkách... Jednotnost implikuje jednotný programátorský přístup a jednotné zacházení (částečně tento princip totiž souvisí s polymorfismem). Pokud tento princip nedodržíme, budeme muset mít zbytečně obsáhlou a komplikovanou dokumentaci a programátorům se bude daný systém velmi špatně programovat.

Tento princip částečně souvisí s dodržením 4. normální formy, protože takto konstruovaná databáze vůbec neumožní programátorovi, aby dostal data, která nemají smysl.

9.5 Rozšířený princip abstrakce pro dosažení flexibility

Jak již bylo zmíněno, dnešní svět se mění závratnou rychlostí každou minutu. Aplikace postavená pouze na současný stav (»natvrdo«napsaná) nemá na vypořádání se se změnami dnešního světa sebemenší šanci.

Velkým a zajímavým úkolem analytika je zvolit správnou míru abstrakce aplikace. Každá abstrakce sebou nese práci navíc, ale při správné konstrukci je jí možné použít pro dosažení flexibility, protože používá jednotný přístup. Toto rozhodnutí je samozřejmě různé projekt od projektu, ale obzvlášť u velkých informačních systémů, u kterých se předpokládá životnost mnoho let, je použití tohoto principu klíčové.

Smysl tohoto principu je dobře vidět na databázích. Cílem těchto abstrakcí je převést nový požadavek na změnu na databázi tak, že nebudeme muset měnit její strukturu (to znamená přidat novou tabulku, sloupec nebo vazbu). Místo toho bude stačit přidat řádek do některé již existující tabulky (tedy změnu zachytíme a ošetříme pouze datově, nemusíme nic měnit na struktuře databáze, nic do kódu dopisovat, ladit ani kompilovat).

Další myšlenkou uváděného principu je to, že ani při použití nejvýkonnějších analytických metod používaných na přemostění rozdílného vidění vývojáře a zákazníka nikdy nebude výsledek stoprocentní. Proto od určitého bodu nepoužívejme tyto metody, zrušme

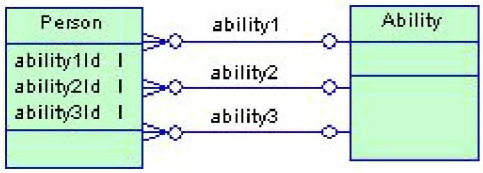
přemostění a udělejme systém tak obecně, aby si zákazník mohl do systému vložit cokoliv ho napadne.

9.5.1 Vazební entity místo tvrdých odkazů

Tato jednoduchá abstrakce podporuje zakládání vazebních tabulek namísto realizování tvrdých odkazů. Je tomu tak proto, že si obvykle nemůžeme být jisti, zdali nám bude počet tvrdých odkazů stačit (ať už jejich počet dimenzujeme na jakýkoliv). Dalším důvodem proti tvrdým odkazům je to, že se s nimi na programové úrovni nešikovně pracuje.

Sekundární výhodou vazební tabulky je to, že umožní levně přidat atribut popisující vazbu mezi kernel entitami (přidání atributu je vždy docela levné na rozdíl od změny struktury tabulek a vazeb mezi nimi). To ovšem souvisí s obecnou datovou vazbou.

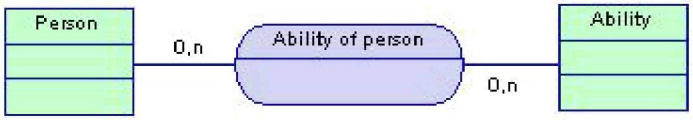
Příklady



Obrázek 9.5: ERD před aplikováním principu flexibility

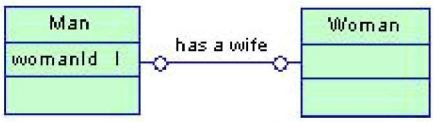
9.5.2 Obecná datová vazba

Předpokládáme-li měnící se počet vazeb mezi entitami nebo si chceme nechat možnost evidovat u vazby nějakou informaci, měli bychom použít obecnou datovou vazbu (viz zdroj [5] Datové modelování v příkladech). Typem vazby můžeme vždy bezpečně popsat význam vazby.



Obrázek 9.6: ERD po aplikování principu flexibility

Příklady



Obrázek 9.7: ERD před aplikováním principu flexibility – obecná vazba

Obrázek 9.8: ERD po aplikování principu flexibility – obecná vazba

9.5.3 Obecný atribut

Vysokou mírou abstrakce je zavedení obecného atributu. Jestliže víme, že daná aplikační oblast neustále přirozeně generuje například nové podtypy některé kernel entity (například vznikají velmi často nové podtypy zaměstnanců viz. příklad), je velmi nevhodné a neobyčejně úmorné pokaždé přidávat do databáze nový podtyp a přepsat kód pro práci s daným podtypem. Tento problém je možné velmi elegantně vyřešit obecnou vazbou na úrovni atributů.

Abstrakce atributů umožní v daném případě definovat a nastavovat nové podtypy a editovat jejich nové atributy pouze zadáním nového záznamu do databáze a na úrovni kódu nemusíme již udělat ani čárku. Do jisté míry to souvisí s polymorfismem – jsme schopni jednotně programově zacházet se všemi podtypy entity. Je ovšem pravda, že uvedenou datovou strukturu je třeba důmyslně implementovat, protože její přímočará implementace u velkého množství instancí je velmi neefektivní.

Příklady

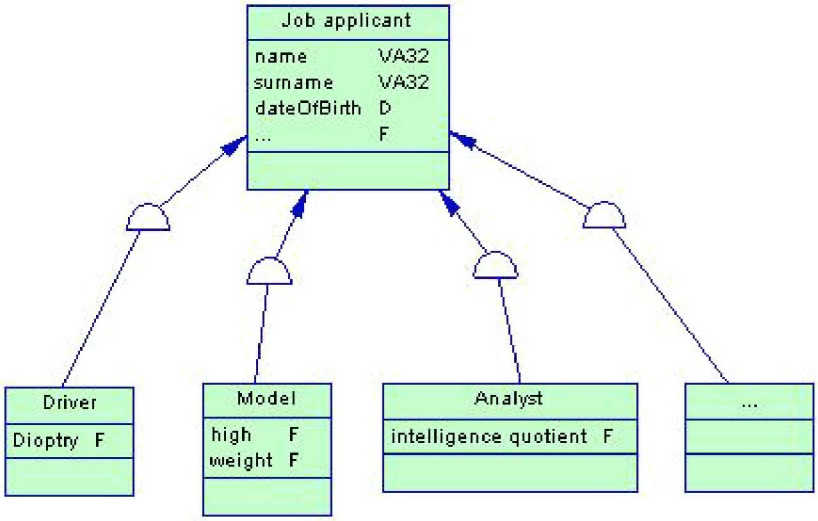
9.5.4 Obecná kategorizace

Další silnou abstrakcí používanou v universálním modelování (viz. zdroj [10] Universální modelování a konstrukce IS) je obecná kategorizace. Myšlenka této abstrakce spočívá v tom, že zákazník nikdy neví (což samozřejmě nepřizná), jakým způsobem hodlá data třídit. Flexibilní konstrukce obecné kategorizace řeší tento problém a ještě umožňuje zavádět další úrovně kategorií.

Příklady

9.6 Princip osvědčených řešení

Poslední princip je lehce úsměvný. Jde o slavný krok stranou a souvisí s dnešním trendem komponentového vývoje softwaru (a nejen softwaru, jde o základní inženýrský postup). Přestože komponentový způsob vývoje softwaru je myšlen na úrovni skládání různých již hotových částí aplikací dohromady, je možné podobný myšlenkový postup použít i pro tvorbu modelů. Velká část reality se neustále

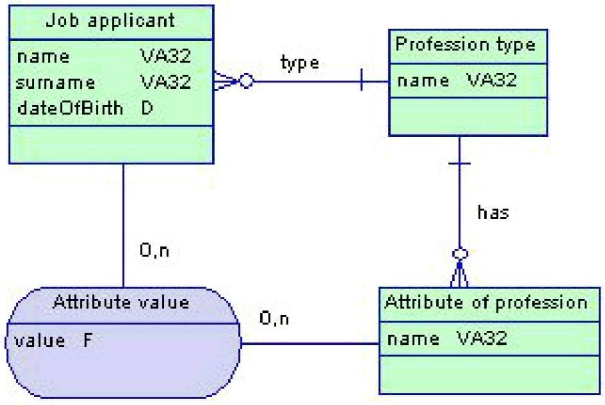


Obrázek 9.9: ERD před aplikováním principu flexibility – obecný atribut

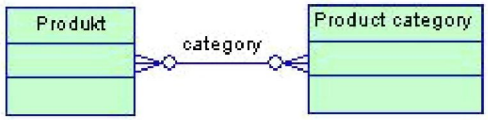
opakuje a je zajisté nesmyslné neustále vyvíjet (a modelovat) všechno znovu »od píky«... Dnešní doba již poskytuje pečlivě vymyšlené komponenty modelů, které jsou navíc prověřené mnoha lety praxe a mnoha úspěšnými projekty.

Analytické a návrhové vzory

Analytické i návrhové vzory přináší do aplikace určitou úroveň abstrakce (což vždy znamená trochu práce navíc), ovšem se správně použitou abstrakcí přichází i jednotný programový přístup a schopnost obsáhnout i nové, dosud neznámé požadavky na software. Dále se jedná o osvědčená a mnoha projekty prověřená řešení a pouze naivní analytik si myslí, že dokáže vše vymyslet lépe a rychleji než ostatní v oboru. Vzpomeňme tvrzení společnosti Gartner Group v úvodu této práce, že používáním komponent se efektivita zvyšuje 5-10krát.



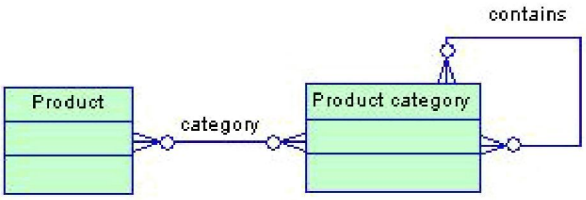
Obrázek 9.10: ERD po aplikování principu flexibility – obecný atribut



Obrázek 9.11: ERD před aplikováním principu flexibility – obecná kategorizace

Jistě v každém projektu existuje mnoho prostoru velmi specifického pro daný projekt. Ale také jistě není žádný důvod, abychom standardní problémy neřešili standardním způsobem.

(viz. zdroj [12] Datové modelování v příkladech nebo zdroj [20] Návrh programů pomocí vzorů)



Obrázek 9.12: ERD po aplikování principu flexibility – obecná kategorizace

9.7 Kde jsou principy zbytečné...

Pravdou je, že ne vždy potřebujeme flexibilní obecný systém. Existuje řada projektů, kde víme, že poté, co software vyvineme a prodáme, ho nebudeme dále vyvíjet a udržovat (například počítačová hra). Je očividné, že nedodržování výše uváděných principů bude mít v těchto projektech menší dopad.

Při vývoji takového jednoúčelového softwaru platí zdánlivě jiné zásady než při vývoji klasického softwaru. Pravdou je, že u jednoúčelového softwaru si můžeme dovolit psát trochu »natvrdo«, ale je to hra s ohněm. Přestože pro takovýto projekt není třeba plánovat budoucí rozvoj aplikace, musíme aplikaci navrhnout natolik dobře, aby unesla alespoň změny a nové požadavky, které se objeví v rámci vývoje aplikace. Proto doporučuji i v takovýchto aplikacích analytické principy dodržovat (alespoň se o to v rozumné míře snažit).

Kapitola 10

Závěr

Zdá se, že pro střední a velké projekty již souborově orientovaným systémům a procedurálnímu programování dávno odzvonilo (snad s výjimkou programování jádra operačního systému). Stejně tak »hurá stylu« při řešení softwarového projektu.

Novým a logickým trendem v IT se očividně pomalu ale jistě stává orientace na tvorbu komponentového softwaru a orientace na služby. Snad již bude tvorba software více podobná jiným průmyslovým odvětvím, jako třeba stavitelství, a na jeho konstrukci se bude nadále využívat inženýrských postupů.

Notací budoucnosti je zcela jistě UML a metodiky založené na nich. Metodiky budou vždy potřeba z důvodu zjednodušení procesu analýzy, ale jejich nebezpečím je přílišné zmechanizování analytické tvorby.

Doufám, že se v této práci podařilo ukázat, že za poučkama a metodikama stojí hluboké analytické principy, které jsou samotným jádrem analytické profese. Přestože IT směřuje k průmyslové komponentové výrobě, je stále důležité (a možná o to důležitější), aby si analytici udrželi dvě spíše umělecké vlastnosti. Nadhled a cit.

Literatura

[1] PETER SOKOLOVSKY: *Tvorba a reengeneering objektově orientovaných obchodních procesů*, SCIENCE 1999

[2] JOSEF FIALA, JAN MINISTR: *Průvodce analýzou a modelováním procesů*, 2003

[3] ANTONÍN CARDA, RENÁTA KUNSTOVÁ: *WORKFLOW Nástroj manažera pro řízení podnikových procesů*, Grada Publishing 2003

[4] PETR LIDMAN: *Diplomová práce – Business Process Modelling*, Fakulta Informatiky MU Brno 2003

[5] JIM ARLOW, ILA NEUSTADT: *UML a unifikovaný proces vývoje aplikací*, Computer Press Brno 2003

[6] HANA JANUSOVÁ, MIROSLAV MULER: *UML srozumitelně*, Computer Press 2004

[7] JOSEPH SCHMULLER: *Myslíme v jazyku UML*, Grada Publishing 2001

[8] JAROSLAV KRÁL: *Informační systémy*, SCIENCE 1998

[9] ZBYŇEK DIVIŠ: *Diplomová práce – Manažerské informační systémy a jejich úloha v řízení podniku*, Fakulta Informatiky MU Brno 2003

[10] ZDENKO STANÍČEK: *Universální modelování a konstrukce IS*, 2003

[11] ZDENKO STANÍČEK: *Materiály k výuce Datového modelování 1, Datového modelování 2 a Řízení implementace IS*, 2001
Dokument dostupný na url
http://www.fi.muni.cz/~stanicek

[12] LUBOR ŠAŠERA, ALEŠ MIČOVSKÝ, JURAJ ČERVEŇ: *Datové modelování v příkladech*, Grada Publishing 2001

[13] MICHAEL J. HERNANDEZ: *Návrh databází*, Grada publishing 2006

[14] MILAN DRÁŠIL: *Materiály k výuce Architektura relačních databázových systémů*, 2001
Dokument dostupný na url
http://www.fi.muni.cz/usr/qdrasil

[15] JIŘÍ POLÁK, VOJTĚCH MERUNKA, ANTONÍN CARDA: *Umění systémového návrhu*, Grada publishing 2003

[16] JAMES MARTIN, JAMES J. ODELL: *Object-oriented analysis and design*, 1992

[17] KARLA PETRLÍKOVÁ: *Diplomová práce – Objektově orientované metody návrhu systémů*, Fakulta Informatiky MU Brno 2005

[18] MICHAL BENEŠ: *Diplomová práce – Prehled OO notací a metodik*, Katedre informacních technologií Vysoké školy ekonomické KU˙Praha 2002
Dokument dostupný na url
http://objekty.vse.cz/Objekty/MetodikyANotace

[19] JIŘÍ SOCHOR: *Materiály k výuce Analýza a návrh systémů a Objektové metody návrhu informačních systémů*, 2001
Dokument dostupný na url
`http://www.fi.muni.cz/~sochor/PA103/Slajdy/`

[20] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES: *Návrh programů pomocí vzorů*, Grada publishing 2003

[21] TOMÁŠ PITNER: *Java, začínáme programovat*, Grada publishing 2002

[22] JOSHUA BLOCH: *Java efektivně, 57 zásad softwarového experta*, Grada publishing 2002

Dodatek A

Definice normálních forem

A.1 1. normální forma

Definice

Relace R se nachází v 1.NF, pokud všechny komponenty n-tice jsou atomické (tj. nejsou opět relací). V záznamech tabulky se nevyskytují opakující se skupiny položek (Coddova definice).

Výklad definice

První normální forma se snaží odstranit z tabulky vícesložkové a vícehodnotové sloupce do samostatných sloupečků.

Anomálie při porušení

Není možné zaručit konzistenci dat pomocí referencní integrity (ale je to možné obejít pomocí triggerů).

Komplikované a neefektivní SQL dotazy (špatně se daná data třídí, špatně se v nich vyhledává, často je třeba data zbytečně parsovat v aplikační logice).

Není možné sloupec rozumně indexovat.

Doporučení

Vždy dodržet. V některých velmi malých aplikacích, kde nám zásah entropie příliš nehrozí můžeme například adresu prohlásit za atomický atribut, přestože není. Je to z toho důvodu, že normalizace adresy je kvůli vnitřním závislostem poměrně komplikovaná (například PSČ a město – údaje jsou úmyslně redundantní, aby mohla pošta

doručit zásilku i když některý údaj chybí) a u takto malých aplikací to za to nestojí.

A.2 2. normální forma

Definice

Relace R je v 2.NF, pokud je v 1.NF a jestliže je každý atribut plně závislý na klíči (nikoliv parciálně).

Výklad definice

Dbejme na to, aby tabulka měla vždy primární klíč.

Nedávejme do asociativních tabulek sloupce, co patří do kernel tabulek.

Anomálie při porušení

Nemáme-li primární klíč, hrozí nám duplicita záznamů a tím pádem i nekonzistence dat. Bez primárního klíče může být způsob, jak identifikovat daný záznam v tabulce, komplikovaný. (Mimochodem primární klíč automaticky tvoří index, takže přístup k záznamu v tabulce probíhá v logaritmické casové složitosti.) Primární klíč navíc slouží jako prostředek pro vytváření vztahů, bez primárního klíče jen steží dosáhneme referenční integrity.

Data ve vazební tabulce jsou duplicitní (tedy opět hrozí nekonzistence) a při smazání posledního záznamu ve vazební tabulce informaci ohledně základní tabulky ztratíme zcela (viz. příklad – pokud přestanou námi evidovaní dodavatelé dodávat v nějaký časový okamžik určitý výrobek, tak po zanesení této změny do databáze přijdeme i o informaci, jak se daný výrobek jmenuje).

Doporučení

Vždy dodržet. Pouze v extrémních případech je možné zvážit cílené porušení z důvodu zvýšení výkonnosti – vyhneme se takto jedné operaci JOIN (například při JOIN s obrovskou tabulkou).

A.3 3. normální forma

Definice

Relace R je ve 3.NF, jestliže je v 2.NF a neexistuje atribut (který není složkou klíče), který by byl tranzitivně závislý na klíči relace R.

Výklad definice

Nenechávejme v databázi redundantní data.

Anomálie při porušení

Databáze obsahuje redundantní data. Opět se nám muže stát, že při opomenutí editace závislé hodnoty zároveň s měněnou hodnotou budeme mít data v databázi nekonzistentní.

Doporučení

Snažit se dodržet, přestože u velkých systémů to bude stát značné úsilí. V některých odůvodněných případech je možné ponechat, ale je třeba prostředky databáze zajistit konzistenci dat (triggery a integritními omezeními).

A.4 4. normální forma

Definice

Relace R je ve 4.NF, jestliže pro každou netriviální multizávislost X platí, že X je nadmnožinou nějakého klíče schématu R.

Výklad definice

V tabulkách databáze by se neměla vyskytovat nevyplněná pole (způsobené podmínečnými závislostmi). Najdeme-li takováto místa v databázi, zaveďme podtyp dané tabulky.

Anomálie při porušení

Na úrovni databáze téměř žádné (krom lehce vyšší paměťové náročnosti na pevném disku). Důležité je ke 4.NF přistupovat spíše jako k psychologickému standardu, který zdůrazňuje, že pokud v některé tabulce zůstává nějaké pole nevyplněné, není to správné a měli bychom se snažit údaj dohledat.

Doporučení

Výhodou dodržování 4.NF je u obrovských tabulek jejich dekompozice na únosnou míru. Ovšem u malých tabulek (mám na mysli s málo sloupci) může způsobit slepé dodržování 4.NF přílišnou dekompozici tabulek a tedy nejen zdlouhavé dotazování, ale i komplikované vyplňování. Záleží na osobním stylu analytika a specifických požadavcích na aplikaci, do jaké míry by měla být 4.NF dodržena.

A.5 5. normální forma

Definice

Výklad definice

Do tabulky není možné přidat nový sloupec (skupinu sloupců) tak, aby se vlivem skrytých závislostí rozpadla na několik dílčích tabulek.

Anomálie při porušení

Doporučení

5.NF je spíše teoretická záležitost, během návrhu databáze není třeba se s ní zatěžovat.