VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FIT
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Úvod do softwarového inženýrství

IUS

Studijní opora

Autoři: Ing. Bohuslav Křena, Ph.D.
Ing. Radek Kočí, Ph.D.

21. prosince 2010

Obsah

Kapitola 1

Úvod

0:20

1.1 Slovo autorů

Vážení studující, dostává se vám do rukou studijní opora k předmětu Úvod do softwarového inženýrství, který je vyučován v prvním ročníku bakalářského studijního programu Informační technologie na Fakultě informačních technologií Vysokého učení technického v Brně.

Studijní opory jsou určeny pro distanční a kombinovanou formu studia, kde plní roli přednášek kvůli specificky omezeným možnostem studia pracujících studujících. Když jsme tuto oporu v roce 2006 vytvářeli, domnívali jsme se, že může být užitečnou pomůckou i pro studující v prezenční formě studia (např. pro zopakování látky před zkouškou nebo pro nahrazení přednášky v případě nemoci). Později jsme si uvědomili, že zpřístupnění studijní opory může mít i své negativní dopady – studijní opora může u některých studujících vyvolat dojem, že nemusí chodit na přednášky, protože se vše podstatné naučí ze studijní opory těsně před zkouškou (dlouhodobý přínos na poslední chvíli a často bez hlubšího pochopení naučených informací je pochopitelně minimální). Vzhledem k poměrně vysoké účasti studujících na našich přednáškách toto riziko považujeme za únosné.

Dobrý předmět by se měl průběžně vyvíjet a s aktualizací přednášek studijní opora zastarává. To se stalo i této opoře, jak jsme zaznamenali v loňském roce mimo jiné i z reakcí studujících v anonymních dotaznících předmětu. Vlastně by nás to nemělo překvapit, protože zkušenosti z českých i zahraničních univerzit poskytujících distanční vzdělávání ukazují, že studijní oporu je nutné aktualizovat alespoň jednou za tři roky. Aktualizace opory je bohužel proces časově náročný. Pokud bychom vyšli z předpokladu, že čas, který je přednášející schopen a ochoten každým rokem věnovat na aktualizaci předmětu, je pevně daný, pak paralelní aktualizace přednášek a studijní opory zákonitě snižuje tempo aktualizace celého předmětu (hrubým odhadem na polovinu). Zda jsou studijní opory v konečném důsledku skutečně prospěšné pro prezenční studium, bude zřejmě vyžadovat hlubší zamyšlení.

Pro množství jiných úkolů, jejichž splnění bylo naší povinností, jsme se v průběhu semestru k aktualizaci této opory bohužel nedostali. Poskytujeme vám proto tuto oporu ve stavu, v jakém se právě nachází, přestože sami víme o celé řadě nedostatků, a doufáme, že i tak některým z vás pomůže. Nespoléhejte však, prosíme, na ni jako na jediný zdroj informací. Pro složení zkoušky znalost opory nemusí stačit! Zkouška totiž zjišťuje komplexní zvládnutí látky prezentované na přednáškách a schopnosti získané poznatky samostatně a tvůrčím způsobem aplikovat (porovnejte Studijní a zkušební řád VUT v Brně, článek 12, odst. 3).

Opora obsahuje výběr podstatných částí našich přednášek z akademického roku 2006/2007, které jsme doplnili o podrobné komentáře i o prvky vyžadované pro samostudium. Jedná se především o zavedení grafických symbolů (tzv. piktogramů) pro označení specifických částí textu. Význam jednotlivých piktogramů je v tabulce 1.1. Oproti klasickým skriptům zde navíc najdete otázky a odpovědi, které vám pomohou ověřit, že jste probíranou látku dobře zvládli.

	Čas potřebný pro studium		Otázka, příklad k řešení
	Cíl		Počítačové cvičení, příklad
	Definice		Příklad
	Důležitá část		Reference
	Rozšiřující látka		Správné řešení
	Obtížná část		Souhrn
			Slovo tutora, komentář
			Zajímavé místo

Tabulka 1.1: Význam používaných piktogramů

Na tomto místě chceme poděkovat prof. Ing. Márii Bielikové, CSc., která nám poskytla své přednášky z roku 2003/04, které jsme použili jako základ svých přednášek a tudíž i nepřímo základ této studijní opory. Naše poděkování patří také doc. Ing. Jaroslavu Zendulkovi, CSc., který nám poskytl materiály k přednášce o strukturované analýze.

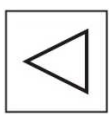
Autoři

1.2 Cíle předmětu a struktura opory

Cílem předmětu je, aby studující získali přehled v oblasti tvorby rozsáhlých softwarových systémů, aby se seznámili s procesem tvorby softwaru a s etapami jeho životního cyklu a aby se naučili používat základní modely jazyka UML.

Udává se, že maximální rozsah studijní opory je 60 stran, protože větší rozsah značně snižuje odhodlání studentů se studijní oporou *prokousat*. Protože tato studijní opora bude sloužit především studentům prezenční formy studia, dovolili jsme si tento rozsah překročit. I tak jsme nemohli do této opory zařadit řadu témat, kterým se v přednáškách věnujeme. Na jednu stranu jsme tím čtenáře ochudili o jistě zajímavá témata, na druhou stranu jsme vybrali pouze to, co považujeme za opravdu podstatné.

Probíraná látka je organizována následovně. Nejdříve se seznámíme se základními pojmy softwarového inženýrství a se specifickými vlastnostmi softwaru, které ke vzniku softwarového inženýrství vedly (kapitola 2). Dále probereme jednotlivé etapy vývoje softwaru a jejich možné uspořádání v životním cyklu softwaru (kapitola 3). Jako příprava pro detailnější seznámení s jednotlivými etapami vývoje softwaru pak slouží kapitola 4 věnující se objektové orientaci, protože právě na objektové orientaci je založena většina moderních technik softwarového inženýrství. Zbývající část opory se podrobněji věnuje jednotlivým etapám vývoje softwaru, a to v pořadí, ve kterém se v životním cyklu nejčastěji objevují. Kapitoly 5 a 6 se zabývají specifikací a analýzou požadavků na vyvíjený software. Kapitola 7 se věnuje metodám a technikám používaným pro architektonický i podrobný návrh.



V kapitole 8 pak najdete z pohledu softwarového inženýrství nejdůležitější postřehy pro implementaci a testování software. Někomu se může zdát, že implementaci a testování je v předmětu Softwarové inženýrství i v této opoře vyhrazen poměrně malý prostor. Máme pro to dva poměrně zásadní důvody. Za prvé implementaci softwaru se věnuje řada dalších předmětů a za druhé na implementaci a testování žádný programátor nezapomene, zatímco k *ošizení* analýzy požadavků nebo návrhu se řada programátorů nechá snadno *svést*. Oporu uzavírá pár poznámek k řízení softwarových projektů v kapitole 9.

Jak jsme již avizovali výše, do této opory se nevešlo vše, co je obsaženo v přednáškách. Jedná se například o agilní metodologie, provoz a servis, ochranu intelektuálního vlastnictví či o etický kodex softwarového inženýra.

Kapitola 2

Softwarové inženýrství

2:00

V této kapitole se seznámíme se základními pojmy softwarového inženýrství, s jeho stručnou historií a se specifickými vlastnostmi softwaru, které ke vzniku softwarového inženýrství vedly.

2.1 Definice softwarového inženýrství

Softwarové inženýrství je systematický přístup k vývoji, nasazení a údržbě softwaru.

Software engineering is the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software; that is, the application of engineering to software. (IEEE Standard Computer Dictionary, 1990)

Ale pozor! Softwarové inženýrství není programování. Jestliže jste se zatím setkali s vývojem jenom opravdu malých programů, zřejmě se vám zdá toto tvrzení minimálně podezřelé. Uvidíte však, že programování (nebo též implementace) v rámci větších softwarových projektů představuje pouze malý zlomek z celkové doby i z celkových nákladů projektu. Programování tedy není softwarové inženýrství, ale jenom jedna z mnoha jeho částí.

Cíle SW inženýrství

Software je dnes všude okolo nás a svěřujeme mu důležité informace, své peníze a někdy dokonce i své zdraví či životy. Proto jsou kladeny stále větší požadavky na jeho správnost, spolehlivost a bezpečnost. Na druhou stranu od počítačů očekáváme stále vyšší funkčnost a pokud možno za co nejméně peněz. Tyto požadavky však jdou v principu proti sobě – jestliže například zvýšíme funkčnost produktu, vzroste tím jeho složitost a bude proto obtížnější a dražší dosáhnout požadované spolehlivosti. Odpovědí na potřebu levně vyvíjet bezpečný a současně užitečný software je právě softwarové inženýrství.

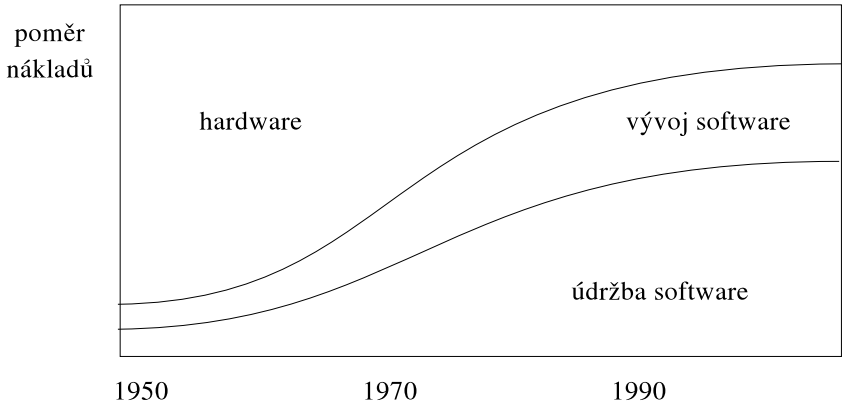
Softwarové inženýrství zahrnuje tři hlavní oblasti:

1. Management projektu – metodologie pro řízení životního cyklu projektu a postupy pro dosažení požadovaného výsledku v požadovaném čase. (Efektivní práce s časem znamená i úsporu nákladů.)
2. Techniky – pro analýzu, návrh, programování, testování a údržbu softwaru.
3. Výchova softwarového inženýra – poskytuje mu základní bázi znalostí, učí ho využívat tyto znalosti v praxi a motivuje ho k vyhledávání nových informací.

2.2 Historie softwarového inženýrství

Můžete si myslet, že historie určitého oboru či odvětví není pro praxi užitečná a že v oblasti informačních technologií, kde probíhá bouřlivý vývoj, to platí dvojnásob. Je sice pravdou, že některé vědomosti z oblasti IT rychle zastarávají. Na druhou stranu však základní principy, na kterých IT stojí, zůstávají v platnosti dlouhá desetiletí. Kromě toho znalost věcí minulých vám pomůže pochopit současné dění a také lépe předvídat budoucí vývoj. Nepřeskakujte proto tuto podkapitolu jenom kvůli jejímu názvu!

Vývoj softwarového inženýrství byl výrazně ovlivněn změnou poměru mezi pořizovací cenou hardwaru, cenou softwaru a náklady vynaloženými na provoz (údržbu softwaru). Přibližný vývoj tohoto poměru je znázorněn na obrázku 2.1.



Obrázek 2.1: Vývoj poměru nákladů mezi HW a SW

V počátcích počítačového věku (padesátá a šedesátá léta minulého století) bylo hlavní nákladovou složkou pořízení hardwaru. Není se ani co divit, když si uvědomíme, že se jednalo o sálové počítače s velkým příkonem a vážící i desítky tun. Výpočetní výkon těchto počítačů byl z dnešního pohledu směšný a jejich spolehlivost byla mizivá. Na těchto počítačích běžely velice jednoduché programy v tzv. dávkovém režimu – programátor program vyděroval na děrné štítky, ty přinesl do výpočetního střediska a za nějaký čas, třeba i druhý den, dostal výsledek. Software byl tedy jednoduchý, neinteraktivní, většinou jednoúčelový a pracoval s ním jeho tvůrce.

Jak se však počítače vyvíjely, hardware byl stále menší, levnější a rychlejší. Programy se naopak začaly zvětšovat (jejich funkčnost i složitost), na jejich vývoji pracovalo více lidí a jakmile se staly interaktivními, začali je používat i běžní (tedy počítačově méně gramotní) uživatelé. V dnešní době pořízení hardwaru je ve firemní sféře malá položka, zatímco koupě nebo vývoj softwaru a zejména pak jeho údržba *spolkne* mnohem více peněz.

Pokud tedy chceme za použití softwaru celkově ušetřit, je lépe věnovat více úsilí vývoji softwaru, než pak řešit problémy způsobené nekvalitním softwarem.

Nyní si projdeme jednotlivá období a seznámíme se s tím nejdůležitějším, co na se na poli softwarového inženýrství objevilo.

2.2.1 60. léta

V šedesátých letech se začaly objevovat problémy s vývojem rozsáhlejších programů. Tyto problémy vyústily v zavedení pojmů *softwarové inženýrství* a *softwarová krize* na konferencích v letech 1968 a 1969.

Softwarová krize

Softwarová krize se projevovala (a stále projevuje) neúnosným prodlužováním a prodražováním projektů, nízkou kvalitou výsledných produktů, problematickou údržbou a nízkou produktivitou práce programátorů. Hledání jednoduchého a účinného řešení na existující problémy vedlo k zavedení strukturovaného programování.

O trvání softwarová krize a o míře její závažnosti se můžeme přesvědčit například ze studie Standish Group Report, která byla zveřejněna v roce 1995 v USA a ze které vybíráme následující údaje (tabulky 2.1, 2.2 a 2.3).

Překročení nákladů o	Projektů
méně než 20 %	15,5 %
21 - 50 %	31,5 %
51 - 100 %	29,6 %
101 - 200 %	10,2 %
201 - 400 %	8,8 %
více než 400 %	4,4 %

Tabulka 2.1: Překračování nákladů SW projektů

Překročení času o	Projektů
méně než 20 %	13,9 %
21 - 50 %	18,3 %
51 - 100 %	20,0 %
101 - 200 %	35,5 %
201 - 400 %	11,2 %
více než 400 %	1,1 %

Tabulka 2.2: Překračování času vývoje SW projektů

Výsledná funkčnost	Projektů
méně než 25 %	4,6 %
25 - 49 %	27,2 %
50 - 74 %	21,8 %
75 - 99 %	39,1 %
100 %	7,3 %

Tabulka 2.3: Dosažená funkčnost SW projektů

Průměrný SW projekt tedy v porovnání s původním plánem stál o **89 %** více, jeho vývoj trval **2,22 krát** déle a výsledný produkt poskytuje pouze **61 %** funkčnosti. Můžeme tedy říci, že průměrný softwarový projekt byl téměř 7 krát horší, než se původně plánovalo.

2.2.2 70. léta

Snahy o překonání softwarové krize vedly počátkem sedmdesátých let k výzkumu tzv. *dobrých* programovacích praktik, tedy postupů, které se v programátorské praxi osvědčily. Při vývoji softwaru se začíná brát v úvahu lidský faktor, je zkoumán životní cyklus tvorby softwaru a zavádí se i podpora pro řízení tvorby softwaru a jeho kvality (začaly se vytvářet procedury zaměřené na systematické testování programů). S tím souvisí i zavedení nových technik programování v týmech (například zavedení vedoucího programátora).

Jsou objeveny výhody návrhu shora-dolů, které umožňují postupné zjemňování systému, což vede k modulárnímu programování a ke vzniku nových programovacích jazyků (C, Pascal). Jednoduché techniky strukturovaného programování jsou nahrazeny celou metodologií – strukturovanou analýzou a návrhem.

Snahy o návrh různých formálních metod pro automatickou syntézu programů nebyly příliš úspěšné – nepodařilo se je úspěšně použít pro programy přesahující několik řádků. Naproti tomu snahy o zavedení abstrakce a modulární dekompozice jako návrhových technik byly úspěšnější a vedly k zavedení důležitého pojmu *abstraktní datový typ*.

Koncem sedmdesátých let dochází k dalšímu rozvoji metodologií, přičemž se začínají formovat datově a procesně orientované metody a je kladen větší důraz na úvodní etapy životního cyklu softwaru, tedy na specifikaci a analýzu požadavků a na návrh. Principy softwarového inženýrství se začínají široce používat v počítačovém průmyslu.

2.2.3 80. léta

V osmdesátých letech se začaly používat specializované programy jako podpůrný prostředek pro vytváření programů – vznikla první vývojová programovací prostředí a první CASE (Computer Aided Software Engineering) nástroje.

Vývoj pokračuje na poli programovacích jazyků – vyvíjí se funkcionální, logické i imperativní programování, objevují se nová programovací paradigmata (např. objektově orientované programování nebo paralelní programování) – i na poli metodologickém – vyvíjejí se formální metody pro specifikaci a návrh větších programů, je zaměřena pozornost na etapy nasazení a údržby softwaru (systémy na podporu údržby verzí a pro řízení konfigurací SW systémů).

2.2.4 90. léta

Ve snaze zvýšit efektivitu vytváření softwarových systémů se rozšiřuje prototypování a vývoj softwaru založený na znovupoužitelnosti (angl. *reusability*) a komponentách. Začíná se sledovat kvalitu/vyspělost softwarového procesu a softwaru použitím metrik.

Vedoucí úlohu přebírá objektově orientované programování – rozšiřuje se jazyk Java a objektová orientace se dostává i do fází specifikace požadavků a návrhu (např. návrhové vzory).

Objevují se nové výzvy pro softwarové inženýrství, například aplikace technik znalostních systémů a umělé inteligence, využití internetu pro distribuovaný vývoj aplikací, fenomén *open source* softwaru.

2.2.5 Současnost

Jedním z významných milníků poslední doby je rozšíření jazyka UML (Unified Modelling Language) do praxe. V současnosti je důraz při vývoji softwaru kladen

zejména na podporu zákazníků, servis a údržbu softwaru, *outsourcing* (tj. zajišťování některých neklíčových operací a funkcí externími pracovníky) se stává běžnou záležitostí.

Protože se klasické metodologie rozrostly a staly se z nich do značné míry byrokratické kolosy, které vývoj zejména menších systémů neúnosně prodražují, můžeme se dnes stále častěji setkat s tzv. lehkými metodologiemi (angl. *ligh-weight metodologies*), které mají flexibilnější životní cyklus a zahrnují techniky jako párové programování (dva programátoři používají jeden počítač) či psaní testů ještě před implementací.

Vědecká komunita v současnosti věnuje značné úsilí rozvoji formálních metod pro simulaci, analýzu a verifikaci systémů, aby zvýšila bezpečnost softwarových systémů vzhledem k jejich stále rostoucímu nasazení v kritických oblastech.

Shrnutí

Jak rostla složitost vyvíjených softwarových systémů, objevovalo se stále více problémů s vývojem softwaru – softwarové projekty se neúnosně prodražovaly, zpožďovaly a výsledný produkt nesplňoval očekávání zadavatelů. Tento stav nazýváme softwarovou krizí.

Odpovědí na softwarovou krizi je softwarové inženýrství, které se s ní snaží bojovat na několika frontách. Postupně se vyvíjejí nové programovací jazyky, aby se v nich rychleji programovalo, aby v programech dělali programátoři méně chyb a hlavně aby se programy lépe udržovaly. Z pohledu řízení softwarových projektů a týmů je zkoumán životní cyklus softwarového projektu a jeho jednotlivé etapy, vznikají celé metodologie pro vývoj softwaru i pro řízení jeho kvality.

I přes poměrně nízkou úspěšnost současných softwarových projektů považujeme softwarové inženýrství za velice úspěšné a užitečné, protože díky němu se dnes mohou vyvíjet nesrovnatelně složitější systémy než dříve.

2.3 Softwarový produkt

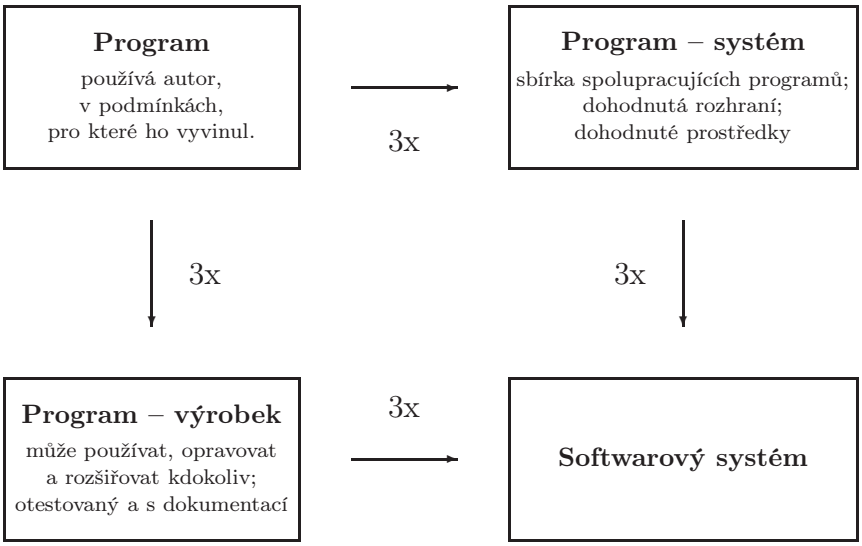
Než se dostaneme k podrobnějšímu zkoumání vlastností softwaru, zkuste si odpovědět na zdánlivě jednoduchou otázku, proč vlastně software vytváříme. Vaši odpověď pak můžete porovnat s tím, jak jsme se s touto otázkou vypořádali my.

Software vytváříme tehdy, kdy řešení určitého problému není možné bez použití počítačů (jedná se například o předpověď počasí či o bankomaty). Software také slouží jako prostředek pro používání nových technologií (například publikování na počítači či návrh součástek). Software také může zlepšit služby zákazníkům (např. informační systém knihovny, jízdní řády na internetu) nebo může ušetřit náklady (např. efektivnějším řízením skladu).

Pokud členové jedné komunity vytvářejí software (ve smyslu výrobku) pro členy jiné komunity (pro uživatele), nazveme takový software softwarovým produktem. Softwarový produkt však nejsou jenom počítačové programy, nýbrž softwarový produkt zahrnuje i požadavky, specifikace, popisy návrhu, zdrojové texty programů, testovací data, příručky, manuály, ...

Softwarový produkt

Objasnění vztahu mezi počítačovým programem a softwarovým systémem je na obrázku 2.2. Nejméně problematický je pochopitelně program, který autor vytvořil sám a pro svoji potřebu. Nemusí se proto zabývat dokumentací ani přílišným testováním, protože když narazí na chybu, může si ji sám opravit.



Obrázek 2.2: Vztah mezi programem a softwarovým systémem

Složitější situace nastává, když se nejedná o jeden program, nýbrž o sbírku spolupracujících programů, kde každý program vytváří někdo jiný. Zde se už musí vývojový tým dohodnout na rozhraní, které budou jednotlivé programy používat pro komunikaci, a už je nutné vytvářet alespoň neformální dokumentaci. Vytvořit programový systém je asi třikrát náročnější než vytvoření izolovaného programu.

Od jednoho programu se však můžeme dát i jinou cestou. Pokud bude chtít autor nabídnout svůj program ostatním uživatelům (a to ať za účelem svého zisku nebo z altruistických pohnutek), musí ho důkladněji otestovat, musí napsat dokumentací (uživatelský manuál) a také musí počítat s tím, že program bude spouštěn na počítačích s různou konfigurací. Vytvoření programu jako výrobku je velice zhruba asi třikrát náročnější než vytvoření obdobného programu.

Systém komunikujících programů určený pro uživatele pak nazveme softwarovým systémem. Jeho vytvoření je pak devětkrát náročnější než vytvoření programu.

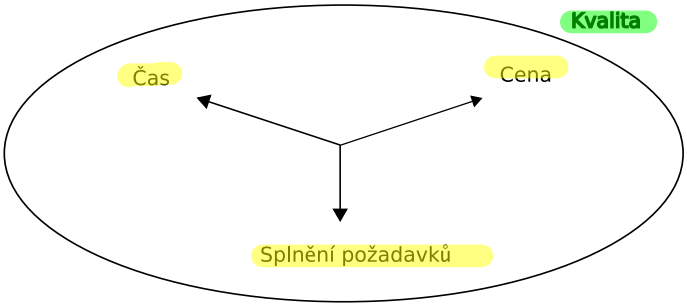
Neberte tato čísla dogmaticky – určitě se budou výrazně lišit případ od případu. Slouží nám zde pouze pro objasnění základních myšlenek a ne jako statisticky průkazné hodnoty.

2.3.1 Kvalita

Zde se seznámíme pouze s definicí kvality, protože více pro pochopení dalšího výkladu nebudeme potřebovat. Řízením kvality softwarového produktu se podrobněji budeme zabývat až v závěrečné kapitole 9.

Kvalita je souhrn vlastností a charakteristik výrobku, procesu nebo služby, které ukazují jeho schopnost splnit určené nebo odvozené potřeby. *(ISO 8402)*

Kvalita však není definovaná jako absolutní míra, nýbrž jako stupeň splnění požadavku či potřeb. Na obrázku 2.3 je kvalita výrobku znázorněna schematicky – a je určena cenou výrobku, časem potřebným pro jeho výrobu a splněním požadavků na něj kladených.



Obrázek 2.3: Kvalita výrobku

Pokud bude cílem vyvinout nespolehlivý software, pak čím bude tento software méně spolehlivý, tím bude kvalitnější.

2.3.2 Vlastnosti softwarového produktu

Různé vlastnosti softwaru mají různou důležitost v závislosti na typu vyvíjeného softwaru. Softwarové výrobky můžeme podle typu rozdělit na dvě základní kategorie – generické a zákaznické.

Generický software

Generický software je vyvinut (vyroben) a je pak prodáván libovolnému zájemci – je též znám pod označením *krabicový software* (angl. *off-the-shelf*) a jeho typickými představiteli jsou operační systémy, textové a tabulkové editory, multimediální přehrávače, internetové prohlížeče, antivirové programy, jednoduché účetní programy, počítačové hry a mnoho dalších. Generický software musí být velice dobře otestován před vypuštěním do prodeje, protože dodatečné opravy jsou velice drahé vzhledem k tomu, že je software obvykle používán velkým počtem uživatelů.

Zákaznický software

Zákaznický software je *šitý na míru* pro konkrétního zákazníka. Vývoj zákaznického softwaru obvykle potřebují větší firmy, které používají software pro určitou specializovanou oblast (pro kterou vhodný krabicový software neexistuje). Cena zákaznického softwaru je v porovnání s krabicovým softwarem o několik řádů vyšší, protože zákaznický software je prodán pouze jednou, v lepším případě nejvýše několika málo zákazníkům. Zákaznický software může být vyvinut dvojím způsobem – buď si ho firma objedná u softwarové firmy nebo zaměstná programátory, kteří systém vyvinou a pak se budou starat o jeho údržbu. Typickým příkladem zákaznického softwaru jsou systémy pro řízení výroby, pro řízení leteckého provozu, pro státní administrativu, pro armádu či pro mobilní sítě.

Vlastnosti z hlediska použití

Vlastnostmi z hlediska použití nemáme na mysli funkcionalitu, ale ty vlastnosti softwaru, které jsou potřeba až tehdy, když je software používán. Jsou to:

Správnost – míra, do jaké software vyhovuje specifikaci.

Spolehlivost – pravděpodobnost, že software bude v daném čase vykonávat zamýšlenou funkci.

Efektivnost – splnění kritérií na využití zdrojů počítačového systému, na čas potřebný na realizaci a dalších kritérií spojených se samotným vývojem (např. náklady).

Použitelnost – úsilí, které je nutné vynaložit na to, aby se dal software používat.

Bezpečnost – míra odolnosti vůči neoprávněným zásahům do systému.

Vlastnosti z hlediska přenosu

Přenositelnost – úsilí, které je nutné pro přenos softwaru z jedné platformy na jinou.

Znovupoužitelnost – míra, do jaké je možné jednotlivé části softwaru znovu použít v dalších aplikacích.

Interoperabilita – úsilí, které je potřebné k zajištění spolupráce systému s jinými systémy.

Vlastnosti z hlediska změn

Udržovatelnost – úsilí, které je potřeba vynaložit na další vývoj a údržbu softwaru podle měnících se potřeb zákazníka a také v důsledku měnícího se okolí (např. změna legislativy).

Pružnost (modifikovatelnost) – úsilí nutné pro modifikaci výrobku při provozu (např. zvýšení jeho funkcionality).

Testovatelnost – úsilí nutné pro testování vlastností softwaru, např. zda se chová správně.

Dokumentovanost – míra, do které jsou všechna rozhodnutí při vývoji zdokumentována a kontinuita dokumentace v průběhu všech etap vývoje.

2.3.3 Problémy při vývoji softwaru

Zatím jsme si definovali problémy při vývoji softwaru z pohledu vnějšího. Pokud bychom použili příměr z medicíny, tak prodražování a zpožďování softwarových projektů i špatná funkčnost výsledného produktu jsou vlastně jenom příznaky. A podobně jako se lékaři snaží pochopit podstatu nemoci, aby s ní mohli účinněji bojovat, tak i my se v této části pokusíme najít vnitřní příčiny problémů, se kterými se setkáváme při vývoji softwaru.

Nevyhnutelné problémy

Složitost – žádné dvě části nejsou stejné; složitost je zdrojem dalších problémů jako např. komunikace v týmech; je náročné pochopit všechny možné stavy systému; problémy s úpravami a rozšířeními, apod.

Přizpůsobivost – když se něco změní, měl by se přizpůsobit software a ne naopak.

Nestálost – mění se okolí a mění se i software (nejde o nahrazení novým); přibývají požadavky na úspěšně používaný software; software přežívá hardwarové prostředky.

Neviditelnost – neexistuje přijatelný způsob reprezentace softwarového výrobku, který by pokryl všechny aspekty; dokonce ani nejsme schopni určit, co v dané reprezentaci chybí.

Syndrom

90% hotovo

Ilustrací neviditelnosti je tzv. *syndrom 90% hotovo*. Při posuzování hotové části se totiž nevychází z hotového, ale z odpracovaného (např. podle plánu). Pokud tedy manažer průměrného projektu (a již dříve jsme si uváděli, že průměrný projekt trval 2,22 krát déle, než se původně plánovalo) poprvé od programátorů uslyší, že je už 90 % práce uděláno, může si být jist, že software není hotov ani z poloviny a že hlášení *90 % hotovo* uslyší v průběhu řešení projektu ještě mnohokrát.

Další problémy

Kromě problémů, se kterými se při vývoji softwaru budeme potýkat vždy, existuje celá řada dalších problémů, se kterými se setkat nemusíme. My zde probereme takové, které se vyskytují poměrně často nebo jsou jinak důležité, s vědomím, že seznam těchto problémů nemůže být nikdy úplný.

Specifikace požadavků — při specifikaci požadavků je nezbytná komunikace s uživatelem (zadavatelem) systému. Ta bývá problematická hned z několika hledisek. Pokud uživatel nemá jasnou a ucelenou představu o výsledném softwarovém systému (a u velkých softwarových systémů je to prakticky vždy), bývá formulace požadavků nejasná a neúplná. Při formulaci požadavků se velice často používá přirozený jazyk (protože je přirozeným komunikačním prostředkem uživatele), který v sobě obsahuje nejednoznačnost. Požadavky mohou být rozporné a mohou se měnit (jak se postupně vyvíjí představa uživatele o výsledném systému). Softwarový inženýr (analytik) se sice vyzná *v počítačích*, ale jeho znalosti z oblasti, pro kterou je software vyvíjen, mohou být poměrně malé, z čehož pak plynou problémy s validací požadavků i s plánováním samotného projektu.

Náchylnost softwaru k chybám — hodně chyb se projeví až při provozu softwarového systému (a ne při vývoji). Odstraňování chyb vede k návratu v etapách vývoje softwaru.

Práce v týmu — jedním z hlavních zdrojů chyb v programech jsou komunikační problémy. Práce na velkých softwarových projektech se těžko organizuje a plánuje (např. v produktivitě práce mezi jednotlivými programátory jsou extrémní odchylky – až 1:20). Problémy plynoucí z práce v týmu ilustrují následující dva postřehy Freda Brookse:

Přidáním dalších pracovníků do zpožděného projektu se tento projekt ještě více zpozdí.

Napsání překladače Algolu zabere 6 měsíců nezávisle na tom, kolik ho vytváří programátorů.

x+y

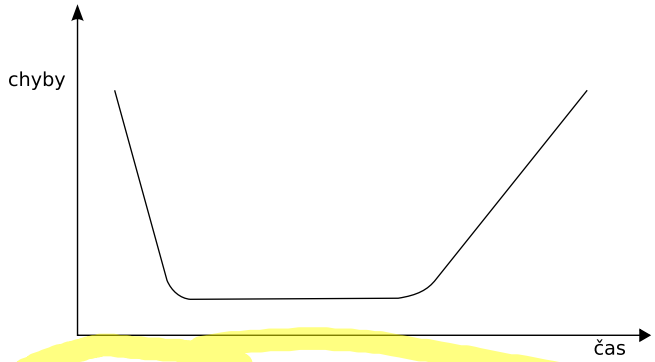
Nízká znovupoužitelnost při tvorbě softwaru – v procesu tvorby softwaru je málo standardů a většinou se software tvoří od začátku. S každým programem se tak vymýšlí už vymyšlené, zatímco jenom málo produktů se sestavuje z už existujících součástí.

Problém míry – metody použitelné na řešení malých problémů se nedají přizpůsobit na řešení velkých (složitých) problémů.

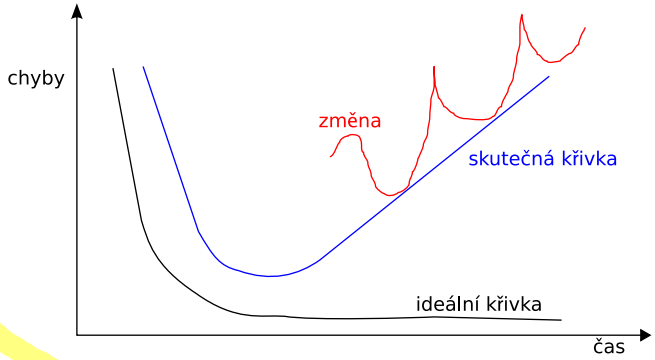
Tvorba dokumentace – čím je softwarový projekt větší, tím vyžaduje více dokumentace (roste jak její kvantita tak i rozmanitost). A čím je dokumentace rozsáhlejší a rozmanitější, tím se hůře udržuje její aktuálnost (vzhledem ke stavu softwaru), úplnost i konzistence. *Ve velkých vojenských softwarových projektech připadalo 400 anglických slov na každý příkaz v programovacím jazyce Ada.*

x+y

Způsob stárnutí softwaru – software se sice fyzicky neopotřebuje, ale přidáváním nových funkcí ve spojení s častými opravami chyb vede k postupné degradaci struktury a ke snižování spolehlivosti softwarových systémů.



Obrázek 2.4: Typická chybová křivka hardwaru



Obrázek 2.5: Typická chybová křivka softwaru

Porovnání křivek stárnutí hardwaru a softwaru je na obrázcích 2.4 a 2.5. Na ose x je vynesen čas a na ose y počet chyb za jednotku času (tedy intenzita poruch). Typická chybová křivka hardwaru má tvar podobný vaně a proto je také nazývána vanová křivka a je platná nejenom pro počítačový hardware, ale prakticky pro všechny stroje, nástroje i přístroje. Když je stroj nový, objeví se u něj několik poruch (často ještě z výroby), které je potřeba opravit (*dotáhnout šroubky*). Pak stroj pracuje poměrně spolehlivě po celou dobu své životnosti. Jakmile je stroj opotřeben, začne se pozvolna objevovat více a vážnějších poruch. V tomto stádiu už není efektivní stroj opravovat a je lepší koupit stroj nový. Jediným problémem je však rozpoznat, kdy se jedná o běžné poruchy a kdy už se jedná o poruchy způsobené opotřebením (to ale není náplní tohoto předmětu).

U softwaru bychom mohli očekávat, že po počátečním odstranění chyb by měla intenzita chyb neustále klesat (v grafu je tento průběh označen jako *ideální křivka*). Úpravy softwaru kvůli měnícímu se prostředí i opravy nalezených chyb však způsobují zanášení nových chyb do softwaru, takže *skutečná křivka* má bohužel rostoucí tendenci. Jakmile se intenzita problémů stává neúnosná, je čas na vytvoření nového systému (úpravy stávajícího systému by situaci ještě zhoršily).

V této souvislosti bychom vás však rádi varovali před *syndromem druhého systému*. Jestliže někdo vytvoří opravdu úspěšný softwarový systém, dříve nebo později ho bude potřeba nahradit systémem novým. Nezřídka se stává, že dříve rozvážný autor povzbuzen úspěchem prvního systému chce nový systém udělat opravdu dokonalý a přidá do něj všechno možné i nemožné. Nový systém je pak

příliš složitý, nepřehledný a neefektivní. *Systém totiž není dokonalý, když k němu nelze nic přidat, ale tehdy, když z něho nelze nic odstranit.* Dalším rizikem při návrhu druhého systému je nepoužití nových technologií. Autor má totiž tendenci použít technologie použité v původním systému – autor je jednak dobře zná a navíc se již osvědčily.

2.3.4 Příčiny zastavení softwarových projektů

Výše jsme probrali řadu problémů, se kterými se při vývoji softwarových systémů můžete setkat. Některé jsou nezbytné, jiným se dá vyhnout, některé jsou kritické, jiné méně důležité. Nyní si pokusíme udělat obrázek o závažnosti jednotlivých problémů, které vedly k zastavení softwarových projektů, podle analýzy více jak 350 firem a 8000 aplikací. Číslo v závorce udává procento zastavených projektů kvůli danému problému.

- neúplnost nebo nejasnost požadavků (13,1 %)
- nedostatek zájmu a podpory ze strany uživatele (12,4 %)
- nedostatek zdrojů, tj. podhodnocený rozpočet a krátké termíny (10,6 %)
- nerealistické očekávání (9,9 %)
- malá podpora od vedení dodavatele nebo odběratele (9,3 %)
- změna požadavků a specifikace (8,7 %)
- nedostatečné plánování (8,1 %)
- vyvíjený systém už není potřeba (7,5 %)
- ...

2.3.5 Důležité faktory pro úspěch softwarových projektů

Cílem předmětu IUS není ani tak naučit studenty problémy při vývoji softwaru řešit, ale důrazně je upozornit na problémy, se kterými se při vývoji softwaru mohou setkat. Pokud si totiž možné problémy na začátku projektu neuvědomíte, jejich dopad bude katastrofální. Když si ale možné problémy uvědomíte předem, budete jejich řešení hledat tak dlouho, dokud nějaké přijatelné nenajdete. Takové řešení může být sice i dosti drastické – projekt odmítnete hned na začátku, ale je to pořád lepší, než když po vynaložení spousty času a peněz zjistíte, že projekt nelze dokončit.

Sledování tohoto cíle nás proto dosud vedlo k prezentaci různých problémů. Nyní uděláme výjimku a uvedeme několik faktorů, které naopak k úspěšnosti softwarových projektů přispívají:

- zájem, zapojení uživatelů,
- podpora vedení zákazníka,
- jasně definované požadavky,
- dobré plánování,
- realistické očekávání,
- správná dekompozice úlohy,
- kompetentnost zúčastněných.

Shrnutí

Vytvořit softwarový produkt (sbírku komunikujících programů určenou pro uživatele) je výrazně náročnější, než vytvořit program, protože softwarový produkt vyžaduje důslednější testování i více dokumentace. Softwarové produkty mohou být buď generické (krabicový software) nebo dělané na zakázku. Podle toho pak nabývají důležitosti různé vlastnosti softwaru.

Kvalita libovolného výrobku a tedy i softwarového produktu je určena třemi základními faktory – cenou, časem a splněním požadavků. Tyto faktory jdou však v principu proti sobě a proto se při tvorbě softwaru musíme vyrovnat s řadou problémů. Vždy se potkáme se složitostí, přizpůsobivostí, nestálostí a neviditelností. Dále pak se můžeme setkat například s problémy při specifikaci požadavků, s problémy plynoucími z práce v týmech nebo s problémy při tvorbě dokumentace.

Kontrolní otázky

2.1 Co je to softwarové inženýrství?

2.2 Jaký je vztah mezi softwarovým inženýrstvím a programováním?

2.3 Jak se projevuje softwarová krize?

2.4 Jaký je rozdíl mezi programem a softwarovým systémem?

2.5 Čím je určena kvalita výrobku?

2.6 Jaký je rozdíl mezi zákaznickým a generickým softwarovým výrobkem?

2.7 Co je to správnost softwaru?

2.8 Co je to použitelnost softwaru?

2.9 Co je to udržovatelnost softwaru?

2.10 Jaké jsou čtyři nevyhnutelné problémy při tvorbě softwaru?

2.11 Co je to syndrom 90 % hotovo?

2.12 Jaká je typická křivka stárnutí softwaru?

Kapitola 3

Modely životního cyklu softwaru

1:10

V této kapitole se seznámíme s procesem tvorby softwaru. Charakterizujeme si jednotlivé etapy životního cyklu softwaru a jejich nejběžnější uspořádání do modelů životního cyklu softwaru.

3.1 Proces vývoje softwaru

V procesu vývoje softwaru se potřeby uživatele transformují na požadavky na software, na jejichž základě je vytvořen návrh systému, podle něhož je softwarový systém implementován a otestován a nakonec je předán uživateli.

Softwarový proces

Softwarový proces definuje kdo, kdy a co má dělat, aby bylo dosaženo požadovaného cíle.

Při vývoji softwaru vstupují do hry tři strany. První stranou je zákazník, který sponzoruje vývoj softwaru (má na zavedení softwaru zájem) a specifikuje požadavky na software. Druhou stranou je dodavatel, který systém vyvíjí. Posledním aktérem jsou uživatelé, kteří upřesňují požadavky na systém a kteří systém budou používat.

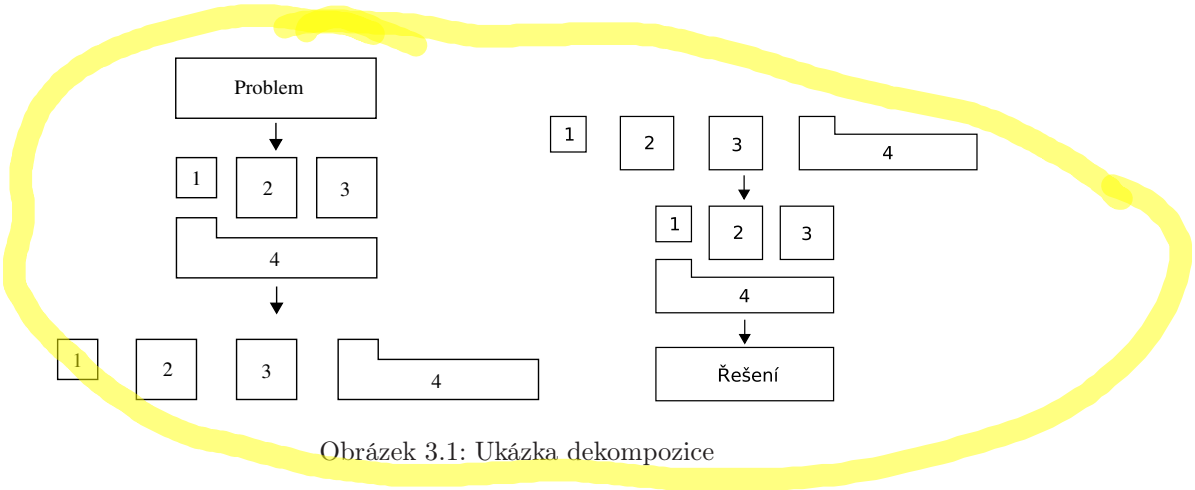
Uživatelé nemají vždy zájem na zavedení nového softwarového systému (na rozdíl od zákazníka, tj. jejich nadřízených). Typickým důvodem je, že se uživatelé nového systému bojí – budou se muset naučit něco nového a nejsou si jisti, že to zvládnou. Také se mohou obávat, že po zavedení systému se ukáže, že není potřeba tolik lidí a že někteří z nich budou propuštěni. A to nemusí být jenom planá obava, protože skutečnou motivací vedení pro vytvoření nového softwarového systému může být právě snížení nákladů snížením počtu pracovníků ...

Když uživatelé nemají zájem na úspěchu projektu, budou se snažit jeho vývoj všemožně brzdit, například vymýšlením nesmyslných požadavků (v úvodních fázích) či neobjektivní kritikou softwaru (při přebírání). V takovém případě je dodavatel v nezáviděníhodné situaci a úspěch projektu je značně nejistý.

3.1.1 Dekompozice

Nyní se seznámíme s dekompozicí – jedním z přístupů, který je hojně využíván při vytváření složitějších systémů a který je příkladem použití pravidla *Rozděl a panuj!* v softwarové praxi. Dekompozice spočívá v rozložení (angl. *decomposition*) složitějšího problému na jednodušší podproblémy, jejichž zvládnutí je lehčí (viz. obrázek 3.1).

Dekompozice přináší lépe zvládnutelné podsystémy, umožňuje soustředit pozornost jen na jeden podsystém, probírat dílčí problémy bez rušivých vlivů a umožňuje vyvíjet jednotlivé podsystémy nezávisle na sobě. Na druhou stranu rozložení systému na podsystémy vyžaduje, aby bylo zavedeno vhodné rozhraní mezi



Obrázek 3.1: Ukázka dekompozice

podsystémy a je pak také nutno věnovat pozornost integraci všech podsystémů a otestování celého systému. Výhody dekompozice však v drtivé většině případů převládají a skutečně velké systémy se bez dekompozice nedají zvládnout.

3.1.2 Životní cyklus softwaru

Existuje mnoho přístupů k vývoji softwaru. Podstaty těchto přístupů jsou vyjádřeny v modelech životního cyklu softwaru, přičemž rozdíly mezi jednotlivými modely spočívají v posloupnosti jednotlivých vývojových etap.

DEF

Model životního cyklu SW

Model životního cyklu softwaru definuje etapy vývoje softwaru a pro každou etapu dále definuje nutné činnosti a její vstupy a výstupy.

Etapy odrážejí činnosti spojené s vývojem softwaru, které jsou následující (číslo v závorce udává přibližný podíl dané etapy na celkově vynaloženém úsilí):

- analýza a specifikace požadavků (8 %),
- architektonický a podrobný návrh (7 %),
- implementace (12 %),
- integrace a testování (6 %),
- provoz a údržba (67 %).

Všimněme si, že nejvíce úsilí (= času a peněz) je spotřebováno ve fázi provozu a údržby. Věnovat dostatečnou pozornost úvodním etapám (tedy specifikaci požadavků a návrhu) se vyplatí, vzhledem k pozdějším úsporám při provozu a údržbě. Počáteční etapy bohužel svádějí k *úsporám* a k častému ověření přísloví, že *lenoch se vždycky nejvíc nadře.*

3.2 Etapy životního cyklu softwaru

Nyní si vymezíme a stručně charakterizujeme etapy při vývoji softwaru. Jednotlivým etapám (zejména těm úvodním) se budeme podrobně věnovat prakticky v celé zbývající části této studijní opory.

Analýza a specifikace požadavků je úvodní etapou při vývoji softwaru, ve které se zabýváme požadavky zákazníka (zadavatele a uživatelů) na software – získáváme je, analyzujeme, definujeme a specifikujeme je (tedy z neformálních, nejasných, neúplných a rozporuplných požadavků zákazníka se snažíme získat strukturované, jasné, úplné a konzistentní požadavky). Pozornost by v této etapě měla být věnována pouze požadavkům samotným a nikoliv jejich realizaci (té se věnují další etapy).

Součástí této etapy by měla být studie vhodnosti (proveditelnosti), která odpoví na důležitou otázku, zda má smysl se do projektu vůbec pouštět. Dalším možným výsledkem této etapy může být identifikace možných rizik a jejich analýza. Nezbytným výstupem této etapy je plán akceptačního testování (testy, na jejichž základě zákazník software převezme – akceptuje). Chybějící specifikace akceptačních testů je *zárukou* problémů při přebírání *hotového* produktu zákazníkem.

Architektonický návrh navazuje na analýzu požadavků a slouží k ujasnění koncepce systému a k jeho dekompozici. Dekompozice vyžaduje vymezení funkcionality jednotlivých podsystémů a definování vztahů mezi nimi (tedy definici komunikačního rozhraní).

Podobně jako se při analýze požadavků plánuje akceptační testování, tak se při architektonickém návrhu plánuje testování celého systému (tedy správnost integrace jednotlivých podsystémů do jednoho celku). Je vhodné také naplánovat postup nasazení systému do provozu, a to včetně zaškolování uživatelů.

Při kritickém zamyšlení (zvláště když vezmeme v úvahu údaje o úspěšnosti *softwarových projektů z předchozí kapitoly) se nám může zdát zbytečné vytvářet plán nasazení systému do provozu již při architektonickém návrhu, protože tento plán se ve většině případů stejně bude výrazně měnit. Je to jistě správný postřeh. Na druhou stranu při architektonickém návrhu mají lidé v paměti informace nutné pro vytvoření tohoto plánu (takže jeho vytvoření je levnější než později) a také nejsou v časovém stresu (projekt je teprve v počátcích), takže vytvoří dobrý (ideově čistější) plán. Obecněji, plány se nevytvářejí proto, aby se přesně dodržovaly, ale aby poskytovaly základní představu o budoucím vývoji projektu a aby sloužily pro odhalování problémů při vývoji a tak umožnily včasnou a adekvátní změnu plánu a upřesnění odhadů.*

Podrobný návrh se soustřeďuje na podrobnou specifikaci softwarových součástí, na výběr algoritmů realizujících požadované funkce, na stanovení logické a fyzické struktury dat a na způsoby ošetřování chybových a neočekávaných stavů.

Při podrobném návrhu se také plánují práce na implementaci součástí, s čímž souvisí vytvoření požadavků na lidské zdroje a odhad doby trvání a nákladů na projekt. Výstupem detailního návrhu by měl být také návrh testů součástí, a to včetně testovacích dat.

Implementace a testování součástí zahrnuje programovou realizaci softwarových součástí, vypracování dokumentace k součástem a otestování implementovaných součástí.

Integrace a testování systému Po implementaci a otestování jednotlivých součástí (podsystémů) je nutné součásti spojit do jediného celku. Nastupuje zde testování celého systému, při kterém se objeví chyby, které nebylo možné odhalit izolovaným testováním samostatných součástí systému. Objevené chyby jsou samozřejmě opravovány, čímž se vlastně vracíme k předchozí etapě – k implementaci a testování součástí. Testování součástí nelze vynechat s odkazem na to, že už součásti

otestovány byly, protože opravou nalezených chyb se do programu mohou zanést chyby nové (což není rozhodně vzácným jevem).

Akceptační testování a instalace spočívá v otestování systému uživatelem. Na základě akceptačního testování se pak zákazník rozhodne buď systém převzít nebo při objevení závažnějších nedostatků softwaru je převzetí odloženo do doby, než dodavatel nedostatky opraví. Po akceptování systému uživatelem (zákazníkem) začíná instalace softwarového systému u zákazníka a školení uživatelů.

Až při této etapě tvůrce softwaru ocení pečlivost při analýze a specifikaci požadavků nebo, což je bohužel častější případ, začne litovat, že se zpracování požadavků nevěnoval důkladněji. Například pokud chybí plán akceptačního testování, uživatel může převzetí systému (a tím třeba zaplacení systému) oddalovat, aniž má dodavatel proti tomu účinný nástroj.

Tato etapa má pochopitelně zcela odlišnou podobu pro různé typy softwarových produktů. Například u generického softwaru si zákazník může software koupit ještě před tím, než si ho nainstaluje a vyzkouší. V takovém případě je ve výhodě spíše dodavatel. Dnes však bývá zvykem, že zákazník ještě před koupí generického softwaru si chce software vyzkoušet a tvůrci softwaru mu to v rámci konkurenčního boje umožňují. Akceptační testování tak vlastně rozhoduje o úspěchu či neúspěchu softwaru.

Provoz a údržba vyžaduje průběžné řešení problémů, které s nasazením používáním softwaru souvisejí. Tato etapa také zahrnuje opravy nalezených chyb, rozšiřování softwaru o nové funkce či přizpůsobování softwaru měnícím se požadavkům okolí.

3.3 Modely životního cyklu softwaru

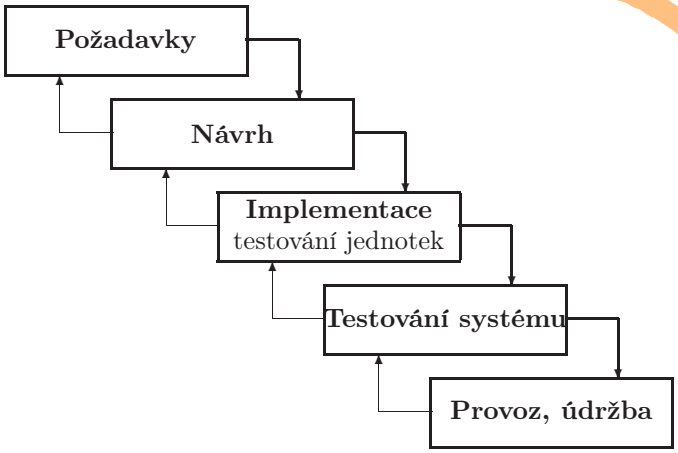
V předchozí sekci jsme se stručně seznámili se základními etapami vývoje softwaru. Každý softwarový projekt je unikátní a jednotlivé etapy i jejich uspořádání se proto liší projekt od projektu. Přesto však můžeme nalézt řadu společných rysů, které lze popsat právě v modelu životního cyklu softwaru.

Model životního cyklu softwaru definuje jednotlivé kroky (etapy), které je nutné vykonat, a jejich časovou následnost. Nedefinuje ale délku trvání kroků ani jejich rozsah. Každá etapa musí být dobře definována a musí vytvářet reálné výstupy, aby bylo možné vyhodnotit její správnost.

3.3.1 Vodopádový model

Základním modelem životního cyklu softwaru je tzv. vodopádový model. V tomto modelu jsou jednotlivé etapy seřazeny za sebou — následující etapa začíná až po ukončení etapy předcházející. S vodopádovým modelem životního cyklu softwaru jsme se již vlastně seznámili, protože podle něj jsou seřazeny jednotlivé etapy v předchozí sekci. Vodopádový model je tedy velice přirozený a zřejmě i nejstarší model životního cyklu softwaru. Jeho grafické znázornění je na obrázku 3.2.

Základním problémem při použití vodopádového modelu je uživatel, který není schopen předem stanovit všechny požadavky. Když pak v závěrečných fázích projektu získá spustitelnou verzi softwaru, vznese celou řadu nových požadavků, případně opraví původní. Místo toho, aby se software nasadil a používal, dostáváme se tak v životním cyklu prakticky na začátek a je nutné provést novou analýzu požadavků, často drastickou úpravu návrhu, implementaci a testování. To je problematické, protože se s tím v životním cyklu nepočítalo — analytici či návrháři



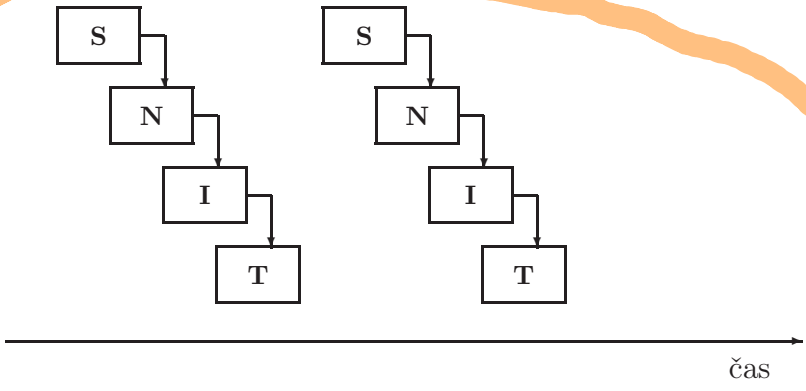
Obrázek 3.2: Vodopádový model životního cyklu softwaru

už mohou pracovat na jiném projektu, změny v návrhu, které by bylo vhodné kvůli novým požadavkům udělat, je těžké prosadit, protože to bude znamenat nákladné změny implementace (v krajním případě to může znamenat až zahození celé implementace). Ve snaze minimalizovat nutné změny implementace pak degraduje struktura softwaru ještě před tím, než se software vůbec nasadí. Vodopádový model také neodpovídá vývoji reálných projektů, kdy se jednotlivé etapy překrývají nebo jdou dokonce v jiném pořadí.

Zdá se, že vodopádový model má vážné nedostatky, nicméně jeho důležitost plyne z toho, že z něho vycházejí ostatní modely životního cyklu softwaru. Vodopádový model lze ale použít i v praxi. Je každopádně lepší než neřízený chaos a v případě, že jsou požadavky předem jasné a dobře definované, může vést dokonce i k úspěchu projektu.

3.3.2 Iterativní model

Iterativní model se snaží odstranit hlavní problém vodopádového modelu životního cyklu softwaru (uživatel vidí spustitelnou verzi softwaru příliš pozdě) rozdělením procesu vývoje softwaru do iterací (viz obrázek 3.3, kde S = specifikace požadavků, N = návrh, I = implementace a T = testování).



Obrázek 3.3: Iterativní model životního cyklu softwaru

Každou iteraci můžeme chápat jako instanci vodopádového modelu. Po každé iteraci má tak uživatel k dispozici spustitelnou verzi softwaru (s neúplnou funkcionalitou), která mu pomůže upřesnit požadavky na software, které jsou zapracovány v další iteraci.

Výhodou iterativního modelu je možnost upřesňovat požadavky uživatelem, aniž to naruší životní cyklus softwaru. Vývoj softwaru je ale nutné rozdělit do iterací, což může vést k návrhu horší struktury systému, než kdyby byly všechny požadavky známy na začátku.

3.3.3 Inkrementální model

Inkrementální model je velice podobný iterativnímu modelu. Na základě specifikace celého systému se stanoví ucelené části systému, které se vytvářejí a odevzdávají uživateli postupně. Struktura systému může být navržena lépe než při použití iterativního modelu, aniž pozdější upřesnění požadavků uživatelem má negativní dopady jako u vodopádového modelu.

3.3.4 Spirálový model

Spirálový model do procesu vývoje softwaru zavádí prototypování a klade značný důraz na analýzu rizik. Jednotlivé etapy se ve vývoji softwaru opakují, vždy na vyšším stupni zvládnuté problematiky. Opakování etap nám může připomínat iterativní nebo inkrementální model životního cyklu. Zde však uživatel nedostává k vyzkoušení verzi softwaru s omezenou funkcionalitou, ale pouze prototyp.

Prototyp se od verze s omezenou funkcionalitou liší tím, že prototyp se po použití zahodí a software (nebo další prototyp) se vytváří znovu. To přispívá k jednoduššímu a ideově čistějšímu systému. Cenou za to je práce spotřebovaná na tvorbu prototypů.

DEF

Prototyp

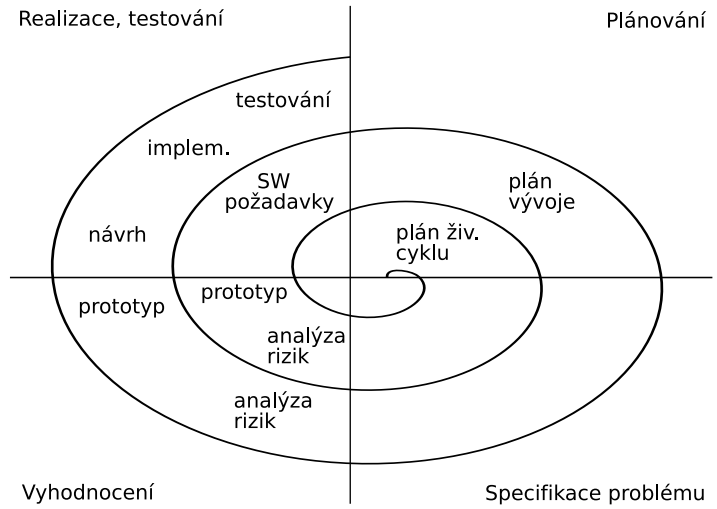
Při použití spirálového modelu pro vývoj softwaru existují spustitelné verze už od začátku, takže chyby jsou odhaleny co nejdříve, což snižuje cenu jejich odstraňování. Spirálový model je však náročný na řízení a lze jen těžko získat celkový přehled o projektu. Ukázka spirálového modelu životního cyklu softwaru je na obrázku 3.4.

3.3.5 Rational Unified Process

V porovnání s dosud probranými modely životního cyklu Rational Unified Process (RUP) není pouze koncepcí, ale je použitelný pro řízení reálných softwarových projektů. RUP je výsledkem výzkumu řady velkých firem koordinovaný firmou Rational. RUP je založen na vývoji softwarového produktu iteračním způsobem (po každé iteraci je k dispozici spustitelný kód). RUP klade důraz na vizualizaci softwarového systému (pomocí jazyka UML – Unified Modelling Language), na průběžnou kontrolu kvality produktu, na řízení změn a na využívání existujících komponent.

3.3.6 Agilní metodologie

Metodologie založené na klasických modelech životního cyklu softwaru se vyvíjely dlouhou dobu a postupem času zmohutněly. Při vytváření zejména menších softwarových systémů však tyto klasické metodologie nejen že ke kvalitě výsledného



Obrázek 3.4: Ukázka spirálového modelu

produktu nepřispěly, ale naopak ji ještě snížily. Dodržování řady spíše byrokratických pravidel těchto metodologií totiž vývoj prodražilo (a cena je jeden z atributů určujících kvalitu produktu), aniž to přineslo přiměřený efekt. Odpovědí na tyto problémy byl vývoj agilních metodologií, které kladou hlavní důraz na člověka jako určující faktor pro kvalitu výsledného produktu.

Shrnutí

Vývoj softwaru je popsán modelem životního cyklu softwaru, který definuje jednotlivé etapy a jejich pořadí. V každém softwarovém projektu lze najít následující etapy:

- analýzu a specifikaci požadavků,
- architektonický a podrobný návrh,
- implementaci,
- integraci a testování,
- provoz a údržbu.

Základním modelem životního cyklu softwaru je vodopádový model. Ten lze úspěšně použít, pokud jsou přesné požadavky na software známy již na začátku. V opačném případě je lepší použít některý z modelů, které počítají s pozdějším upřesňováním požadavků, jako jsou iterativní model, inkrementální model nebo spirálový model. Tyto modely jsou však konceptuální a praktické softwarové projekty se jimi přesně neřídí.

Naproti tomu existují i praktičtěji zaměřené a méně restriktivní modely. My jsme si uvedli stručnou charakteristiku RUP, se kterým se budeme setkávat i v dalších kapitolách této opory, a agilní metodologie.

Kontrolní otázky

3.1 Kteří aktéři vstupují do procesu tvorby softwaru?

3.2 Proč se při vývoji softwaru uplatňuje dekompozice?

3.3 Jaké jsou nevýhody dekompozice?

3.4 Co je to model životního cyklu softwaru?

3.5 Vyjmenujte etapy při vývoji softwarového systému.

3.6 Která etapa obvykle spotřebuje nejvíce zdrojů?

3.7 Jaké jsou hlavní úkoly analýzy a specifikace požadavků?

3.8 Co je náplní architektonického návrhu?

3.9 Jaký je rozdíl mezi testováním systému a akceptačním testováním?

3.10 Nakreslete vodopádový model životního cyklu softwaru.

3.11 Kdy lze úspěšně použít vodopádový model pro vývoj reálného softwaru?

3.12 Jaká je základní myšlenka iterativního modelu?

3.13 Jaký je účel prototypu při vývoji softwaru?

Kapitola 4

Objektová orientace

2:00

V této kapitole se dozvíte

Cílem této kapitoly je seznámení se základními principy objektově orientovaného přístupu k tvorbě softwaru a programování. Všimneme si význačných rysů objektově orientovaných jazyků a seznámíme se s nejrozšířenějším jazykem, používaným v objektově orientovaných metodách návrhu – s jazykem UML.

4.1 Objektově orientované modelování a návrh

Výhodou objektově orientovaných přístupů k návrhu systému je především stabilita navrhovaných prvků a jednoduchost jejich změn. Klasické metodologie založené pouze na vnímání funkcí, které systém musí zajistit, jsou nestabilní z pohledu neustále se měnících požadavků uživatelů, zatímco metodologie založené na objektech vykazují větší míru odolnosti. Proto se vývoj systémů zaměřuje na objektově orientovanou analýzu, návrh a také implementaci. V devadesátých letech bylo vyvinuto mnoho metodologií objektově orientované analýzy a návrhu. Každá z nich poskytuje různé metody a nástroje a definuje vlastní (různorodé) procesy tvorby systémů. Tento stav se ukázal jako nevyhovující, protože nezahrnuje komunikaci návrhářů, programátorů a ostatních zainteresovaných osob na vývoji aplikace. Proto se vyvíjela technologie, která by umožnila sjednotit jednotlivé procesy vývoje, pokud možno pro celou návrhovou etapu. Tato snaha vyústila v definici jazyka UML [2] (Unified Modelling Language) v roce 1995. UML ovšem sám o sobě nenabízí metodologii tvorby či návrhu systémů, ale nabízí pouze prostředky pro unifikovanou tvorbu modelů různých aspektů navrhovaného systému. UML je součástí různých metodologií návrhu, např. RUP.

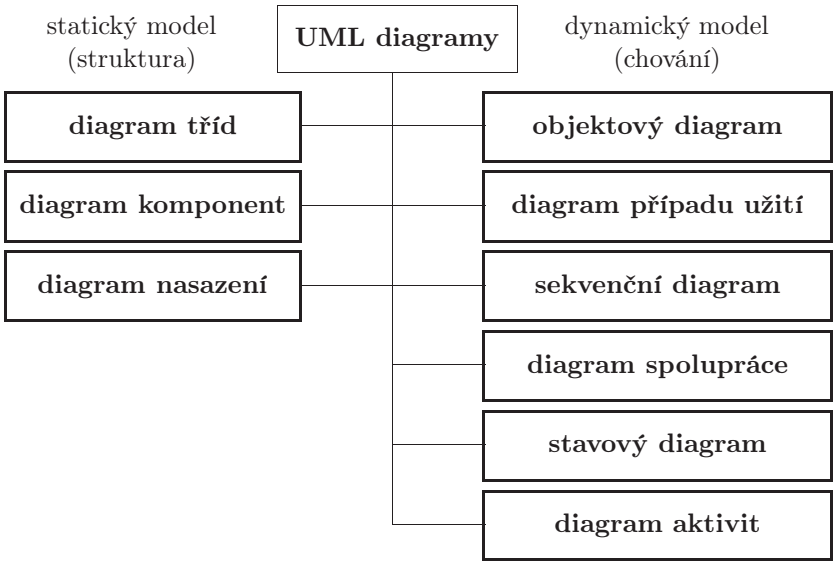
Objektově orientovaný návrh nutně neimplikuje implementaci navrhovaného systému v objektově orientovaném jazyce. Pro složitější systémy však přináší objektově orientovaný návrh značné výhody i v případech neobjektových implementací.

4.1.1 Jazyk UML

UML operuje s pojmem *pohled* (*view*). Pohled systému je projekce systému na jeden z jeho relevantních aspektů. Taková projekce se zaměřuje na příslušný aspekt a ignoruje ostatní. Jak jistě vyplývá z podstaty věci, je vhodné vytvářet několik různých pohledů na tentýž systém. Pohledy nad systémem jsou pak modelovány prostřednictvím vhodných nástrojů (modelů) poskytovaných UML. Můžeme rozlišit tyto základní pohledy:

- *Strukturální pohled (structural view)* popisuje vrstvu mezi objekty a třídami, jejich asociace a možné komunikační kanály.
- *Pohled chování (behavior view)* popisuje, jak systémové komponenty (objekty) interagují, a charakterizuje reakce na vnější systémové operace.
- *Datový pohled (data view)* popisuje stavy systémových komponent (objekty) a jejich vazby.
- *Pohled rozhraní (interface view)* je zaměřeno na zapouzdření systémových částí a jejich potenciální použití okolím systému.

Jazyk UML nabízí několik základních diagramů. My se zaměříme na jednotlivé aspekty UML diagramů v těch částech učebního textu, které budou mít vazbu na daný typ diagramu v analýze a návrhu softwarových systémů. Nyní si pouze uvedeme jejich přehled (viz obrázek 4.1).



Obrázek 4.1: UML diagramy

4.2 Základy objektové orientace

Přestože mnoho uživatelů používá objektově orientované technologie (metody návrhu, programovací jazyky), výsledný efekt není takový, jak jej objektová technologie prezentovala a jaké objektová technologie skutečně nabízí. Za klíčový důvod těchto neúspěchů lze považovat nepochopení základního významu objektové technologie a odpoutání se od vžitých představ tvorby výpočetních modelů. V tomto pojetí je objekt většinou chápán pouze jako jakási struktura, která umožňuje částečně modularizovat vyvíjenou aplikaci. Princip objektově orientované technologie je však radikálně odlišný a dal by se charakterizovat jako přiblížení vyjadřovacích prostředků strojů myšlení a chápání světa člověkem. Základním principem objektově orientovaného návrhu je myslet v reálných objektech (tj. uvědomit si *jaké* objekty máme, *jak* se chovají, *co* je třeba o nich vědět), nikoli v datových strukturách a procedurách. Na reálný svět můžeme nahlížet jako na množinu objektů, které reagují na vnější podněty, a některé z nich mohou vyvíjet i vlastní aktivity. Objekt vytvořený v objektově orientovaném programovacím prostředí je pak abstrakcí reálného objektu a modelování nějaké části reálného světa je vytváření abstrakcí nad objekty dané části reality.

Objektově orientované přístupy lze považovat za soubor objektově orientovaných prostředků (třídy, objekty atd.), jejich vzájemných vazeb a metodologie. Velmi podstatné je uvědomit si (tak jako i v jiných technologiích), že objektově orientované přístupy nejsou *cílem* našeho snažení, ale *prostředkem* k dosahování

cílů. Velmi důležitou podmínkou správného použití objektových přístupů je nejen zvládnutí vyjadřovacích prostředků, ale především metod jejich použití a nasazení na řešení konkrétních problémů.

4.2.1 Základní pojmy objektové orientace

Objektově orientované systémy jsou většinou charakterizovány následujícími základními pojmy:

DEF

Objekt (Object) spojuje data a funkcionalitu do uzavřené jednotky. Na objekt můžeme pohlížet jako na černou skříňku (*black-box*), jejíž vnitřní realizace je nám skryta. Objekt má stav, chování a identitu. Stav objektu je množina podmínek, ve kterých může objekt existovat. Změny stavu reflektují chování objektu. Chování objektu definuje, jak objekt reaguje na vnější či vnitřní podněty. Chování je popsáno množinou operací (metod).

- Součástí definice objektu jsou především *proměnné objektu a metody* (tj. operace objektu). Proměnné reprezentují datové složky objektu, metody implementují chování objektu.
- *Atribut a proměnná objektu*. Atribut reprezentuje vlastnost objektu a je definován množinou hodnot, kterých může nabývat. Atribut není proměnná objektu (i když tak bývá většinou implementován). Atribut může být složený z více proměnných objektu nebo může být vypočítaný (např. objem kvádru je součinem velikostí jeho tří hran).
- *Stav objektu* je reprezentován množinou hodnot jeho atributů (resp. aktuální hodnoty všech atributů reprezentují aktuální stav objektu).
- *Rozhraní objektu* je definováno množinou operací, které objekt nabízí. Obsahuje pouze deklarace metod, tj. definuje, jakým zprávám objekt rozumí a tedy co umí dělat. Neříká jak to dělá, to je otázkou implementace metod objektu.
- *Komunikace objektů (Message passing)*. Objekty mezi sebou (případně objekt sám se sebou) komunikují prostřednictvím zasílání zpráv. Zpráva je příjemcem (*receiver*) chápána jako požadavek na provedení indikovaného chování (metody) a návrat výsledku akce volajícímu objektu (*sender*). Zpráva obsahuje identifikátor příjemce, název operace (metody), která se má vykonat a argumenty operace.
  Ukázka zasílání zpráv v jazycích Java a Smalltalk. Příjemce zprávy je identifikován proměnnou *obj*, název operace je *pridej* a operace má jeden argument – celé číslo.

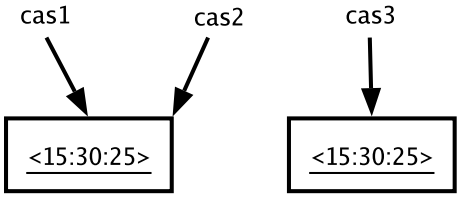
  x+y

  Java: `obj.pridej(100);`
  Smalltalk: `obj pridej: 100.`
- *Identita objektu* znamená, že každý objekt je jedinečný, bez ohledu na jeho stav. Dva objekty mohou být stejné (tj. mají stejné chování a v daném okamžiku stejný stav), ale jde o dva různé objekty. Je třeba rozlišovat test shodnoti objektu (např. dva objekty reprezentující stejné datum) a test identity (zda se nejedná o tentýž objekt).

Příklad objektu. Pro lepší reprezentaci uvedených pojmů si nyní ukážeme jednoduchý

x+y

příklad. Předpokládejme, že máme objekt, reprezentující čas v intervalu 0hod:0min:0sec – 23hod:59min:59sec. *Atributem* objektu je čas, *rozhraní* objektu tvoří dvě operace – čtení času a nastavení času. Obě operace pracují s textovou reprezentací času. Atribut je reprezentován *proměnnou objektu* reprezentující počet sekund od času 0:0:0, tzn. že může nabývat hodnot 0 – 86400. Obě operace tedy musí provádět konverze řetězce z a na příslušné celé číslo. Uvažujme situaci na následujícím obrázku – máme dva objekty reprezentující stejný čas a 3 reference na tyto objekty (*cas1*, *cas2* a *cas3*).



Test shodnosti a identity pak vypadá následovně (použijeme dva typy zápisu jako ukázku – syntaxe jazyků Java a Smalltalk):

	Java	Smalltalk	výsledek testu
shodnost	$cas1.equals(cas2)$	$cas1 = cas2$	true
	$cas2.equals(cas3)$	$cas2 = cas3$	true
identita	$cas1 == cas2$	$cas1 == cas2$	true
	$cas2 == cas3$	$cas2 == cas3$	false

Java používá metodu *equals*, protože = je operátor přiřazení. Ve Smalltalku je := operátor přiřazení.

Abstrakce (Abstraction). Vytvářený systém objektů a tříd je abstrakcí řešeného problému. Jinými slovy, doménu řešeného problému analyzujeme a klasifikujeme do abstraktních datových struktur, v našem případě do objektů. Klasifikace je založena na rozpoznávání podobností v řešené problematice. Smyslem abstrakce je zjednodušit pohled na celý systém bez ztráty jeho významu. Každému objektu pak odpovídá určitá zodpovědnost za řešení identifikované části problému – objekt je abstrakcí části řešeného problému.

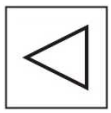
Zapouzdření (Encapsulation). Hlavní vlastností zapouzdření je ukrytí implementačních detailů objektu. Jedinou možností, jak může okolí objektu modifikovat jeho stav nebo přistupovat k interním informacím, je prostřednictvím operací, resp. zasíláním zpráv.

Polymorfismus (Polymorphism). Polymorfismus lze přeložit jako mnohotvarost, schopnost výskytu v mnoha formách. V objektově orientované technologii polymorfismus znamená výskyt různých typů chování na základě jedné zprávy. Jinak řečeno, my můžeme objektům zasílat stejné zprávy, aniž známe jejich typ (tj. třídu), a zpráva bude zpracována příslušným způsobem právě podle typu objektu. Polymorfismus mimo jiné umožňuje ukládat do proměnných různé objekty v různých okamžicích.

- *Časná vazba* znamená, že při zasílání zprávy je implementace příslušné operace (metoda) vybrána staticky v době kompilace.
- *Pozdní vazba* znamená, že při zasílání zprávy je implementace příslušné operace (metoda) vybrána dynamicky až za běhu programu. Pro dosažení polymorfismu musí jazyk podporovat volání metod prostřednictvím pozdní vazby.

Dědičnost (Inheritance). Organizuje a usnadňuje polymorfismus a zapouzdření objektů tím, že umožňuje definovat a vytvářet objekty na základě již existujících objektů – ty pak sdílejí (a rozšiřují) jejich chování bez nutnosti reimplementace. Vyjadřuje hierarchický vztah mezi objekty.

Nejznámější je dědičnost tříd (viz kap. 4.2.2), avšak existují jazyky, které nemají třídy a řeší problém dědičnosti jiným způsobem (viz kap. 4.2.3).

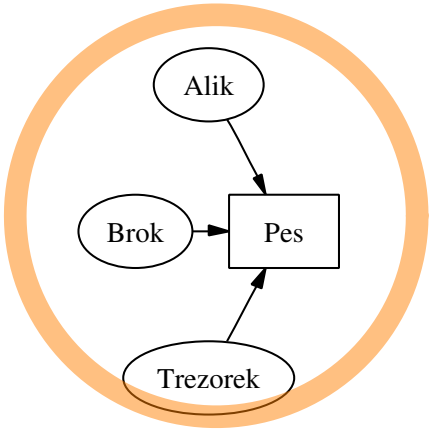


4.2.2 Třídně založené jazyky

Třídně založené jazyky představují takový styl objektově orientovaného programovaní, který definuje *třídy* objektů. Třída je generická definice pro množinu podobných objektů. Třída je tedy chápána jako množina objektů, které mají stejné chování a stejnou množinu atributů. Objekt je instancí své třídy, tzn. že se všemi instancemi sdílí chování definované třídou a jeho stav je na ostatních instancích nezávislý. Třída je představována speciálním objektem, který definuje atributy podobných objektů, chování podobných objektů a speciální operace (především tvorbu instancí, vyhledávání metod apod.) Typickými příklady jsou jazyky Smalltalk, C++, Java nebo C#.

Třída (Class). Třída je generická definice pro množinu podobných objektů. Třída zajišťuje specifikace atributů a chování objektů, jde tedy o abstrakci nad objekty. Objekt je instance příslušné třídy. Třídu lze chápat jako množinu svých instancí nebo jako vzor (šablonu) pro vytváření objektů.

Ukázka vztahu objektů, jejich klasifikace do třídy Pes a vztah těchto objektů ke své třídě.



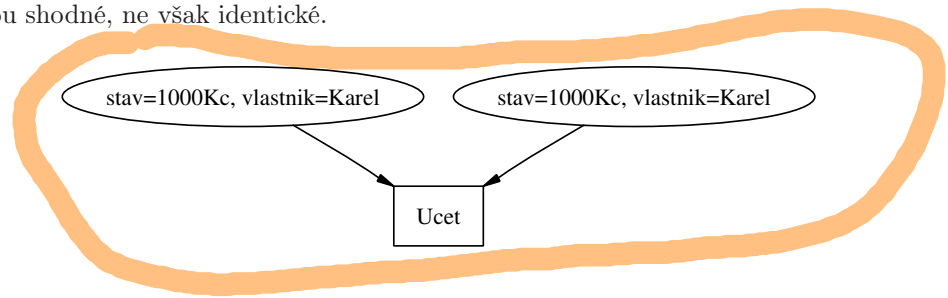
Dědičnost. Podstatnou vlastností třídně založených jazyků je vyjádření hierarchie objektů prostřednictvím dědičnosti tříd. Dědičnost vyjadřuje vztah generalizace–specializace mezi třídami, tj. možnost vytvářet speciálnější varianty existujících tříd. Tato speciální (odvozená) třída dědí (sdílí) atributy a metody nadřazené třídy, které může modifikovat, případně přidávat nové vlastnosti. Každý jazyk může definovat různé speciality spojené s dědičností, např.:

- Smalltalk a Java pracují s jednoduchou dědičností (tj. třída může být specializací pouze jedné nadřazené třídy) a definují společného předka pro všechny

třídy (kořen pomyslného stromu dědičnosti) – třídu Object, takže každá (i prázdná) třída disponuje základními operacemi.

- C++ umožňuje vícenásobnou dědičnost (tj. třída může být specializací více nadřazených tříd) a nedefinují společného předka, takže prázdná třída nemá žádné chování.

Identita objektu nesouvisí s třídou! Dva objekty téže třídy se stejným stavem jsou shodné, ne však identické.



4.2.3 Prototypově založené jazyky

Prototypově založené jazyky představují takový styl objektově orientovaného programovaní, který pracuje pouze s *objekty*. V těchto jazycích existuje vždy minimálně jeden počáteční objekt, který slouží jako prototyp ostatním objektům. Nové objekty se vytvářejí klonováním jiných, již existujících objektů. Typickými příklady jsou jazyky Self či JavaScript.

Klonování. Nové objekty se vytvářejí klonováním již existujících objektů. V těchto jazycích vždy existuje alespoň jeden objekt, který může sloužit jako prototyp pro nové objekty.

Delegování. Podstatnou vlastností prototypově založených jazyků je vyjádření dědičnosti objektů prostřednictvím delegování – každý objekt může určit množinu jiných objektů (nadobjektů), na které deleguje zprávu, které sám nerozumí. Objekty tím mohou sdílet chování s jinými objekty.

4.2.4 Typování

Smyslem typování je dát elementům (hodnoty v paměti, proměnné, atp.) sémantický význam. Na nejnižší úrovni je každá hodnota sekvence bitů. Pokud hodnotě můžeme přiřadit typ, můžeme s danou hodnotou pracovat na vyšší úrovni (víme, jaké operace je možné provést atp.) Typy také umožňují kontrolu, zda používaná operace je povolena v daném kontextu. V objektově orientovaném prostředí se může třída považovat za typ objektu (objekt je instancí třídy, která definuje operace apod.) Většinou se rozlišuje mezi staticky typovanými a dynamicky typovanými jazyky.

Staticky typované jazyky. U těchto jazyků dochází k typové kontrole v době kompilace. Mezi staticky typované jazyky se řadí např. C++ a Java.

Dynamicky typované jazyky. U těchto jazyků dochází k typové kontrole v době běhu programu. Mezi dynamicky typované jazyky se řadí např. Smalltalk, Lisp, Python atd. Tyto jazyky se vyznačují tím, že jejich součástí je virtuální

stroj, který interpretuje buď přímo program (skriptovací jazyky), nebo předem zkompilovaný a optimalizovaný mezikód (někdy také označovaný jako byte-kód).

Ukázka možného chování statických a synamických systémů na následujícím příkladu, napsaného v pseudokódu.

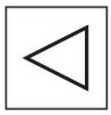

```
var x;          // (1)
x := 5;         // (2)
x := "hi";      // (3)
```

U staticky typovaných systémů je řádek č. 3 ilegální, neboť proměnné x již byl přiřazen typ celé číslo a nelze do něj nyní přiřadit řetězec. U dynamicky typovaných systémů je tato operace povolena, neboť nepožaduje u proměnné x typovou konzistenci. Naproti tomu následující ukázka

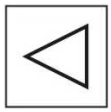
```
var x;          // (1)
x := 5;         // (2)
5 / "hi";       // (3)
```

by u dynamicky typovaných systému vyvolala chybu za běhu programu, neboť operace dělení není definována nad celými čísly a řetězci.

Dynamická kontrola typů probíhá u všech jazyků. Jako dynamicky typované se ovšem označují pouze ty jazyky, které nemají statickou typovou kontrolu. Některé staticky typované jazyky (Java, C++) umožňují dynamické přetypování, čímž částečně obcházejí statickou typovou kontrolu (tímto způsobem se typová kontrola přenáší až do běhu programu).

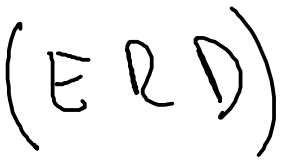


Někdy se můžete setkat s dělením na silně a slabě typované jazyky. Význam těchto pojmů však není jednoznačný, existují i protichůdné definice (viz např. [1]).



4.3 Vztahy mezi třídami

Pro zobrazení tříd a statických vztahů mezi nimi slouží diagram tříd z jazyka UML [2]. Mezi vztahy patří zejména:



- asociace (association)
- závislost (dependency)
- zobecnění (generalization)
- realizace (realization)

Každá relace (vztah) je, nebo může být, řiditelná. Řiditelnost je zobrazena šipkou a označuje primární objekt v relaci. Např. u vztahu zobecnění směr šipky značí směr zobecňování, tj. postup od speciálnějšího k obecnějšímu. U závislosti používání směr šipky označuje objekt, který je používán, na kterém závisí druhý objekt ve vztahu.

4.3.1 Asociace

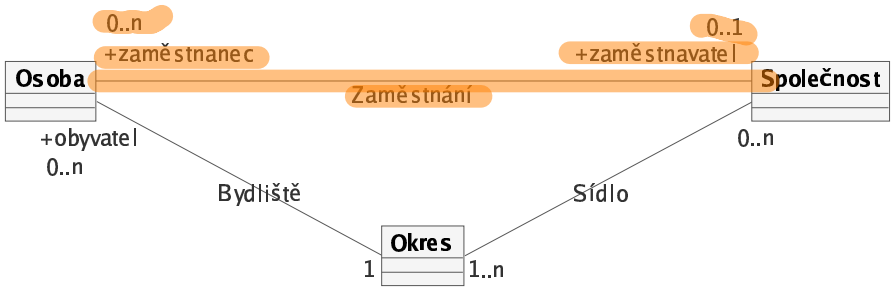
Asociace slouží k zachycení vztahů a informací mezi třídami z několika různých perspektiv. Vztahy vyjadřují zejména:

- Identifikaci vztahu mezi třídami.

  - Asociace má svůj název, název může být sloveso nebo podstatné jméno. Např. na obrázku 4.2 je znázorněna asociace *Zaměstnání*, kterou můžeme také číst jako *osoba je zaměstnána ve společnosti* nebo *společnost zaměstnává osobu*.
  - V objektově orientovaném návrhu dáváme přednost podstatnému jménu.
- S instancí jaké třídy může být svázána instance jiné třídy (každé spojení váže instanci jedné třídy s instancí druhé třídy).
- Kolik instancí třídy A může být svázáno s instancí třídy B (násobnost vztahu).
  - Počet spojení se v čase může měnit. Omezení na počet spojení se vyjadřuje násobnostmi (mohutnostmi).
  - Násobnost se většinou vyjadřuje těmito symboly:
    * 0, 1, n, konkrétní číslo
    * místo n se někdy používá *
    * Např. u asociace Zaměstnání (mezi třídou Osoba a Společnost) na obrázku 4.2 znamená násobnost 0..n u třídy Osoba, že s jedním objektem třídy Společnost může být svázáno 0 až n (tj. neomezeně) instancí třídy Osoba.
  - Násobnost vyjadřuje záměry návrhu (např. zda člověk může mít maximálně jedno zaměstnání nebo více).
- Jakou roli hraje objekt ve vztahu (např. zaměstnanec, zaměstnavatel apod.)

Ukázka diagramu tříd a asociací mezi třídami. Asociace se graficky znázorňuje čárou mezi asociovanými třídami. Násobnost a role se uvádějí u čáry blíže k objektu, ke kterému se váží. Např. instance třídy Osoba hraje ve vztahu k instanci třídy Společnost roli zaměstnace (a naopak, instance třídy Společnost hraje roli zaměstnavatele). Každý zaměstnanec může mít 0..1 zaměstnavatelů (tj. žádného nebo jednoho) a každý zaměstnavatel může mít 0..n zaměstnanců (tj. žádného nebo neomezený počet).

x+y



Obrázek 4.2: Asociace mezi třídami.

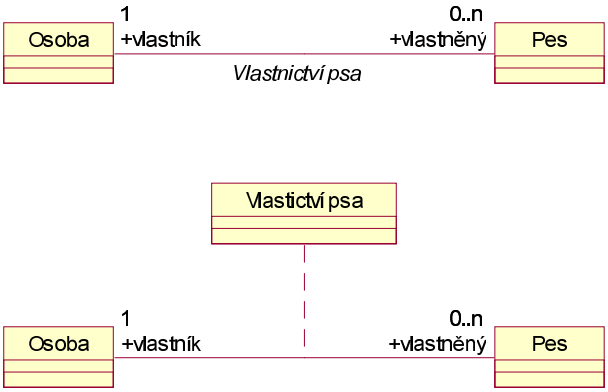
Stupeň asociace vyjadřuje, kolik tříd participuje v jedné asociaci. Nejběžnějším

DEF

typem je binární asociace (tj. mezi dvěma třídami), mohou se vyskytnout i asociace vyšších stupňů (ternární atd.) Asociace vyšších stupňů se většinou převádějí na binární asociace, pokud to není možné, provádí se povýšení asociace na třídu.

Povýšení asociace na třídu. V objektově orientovaném návrhu lze asociaci povýšit na třídu. Povýšení se dělá v případech, kdy chceme asociaci přiřadit parametry, které nelze (nebo to není vhodné) přiřadit objektu v asociaci. Jiným případem povýšení je eliminace vícenásobných asociací.

Následující příklad ukazuje asociaci *Vlastnictví psa* mezi osobou a psem. Asociaci chceme přiřadit atributy (od kdy je pes vlastněn apod.) Tyto atributy nelze včlenit do třídy *Osoba*, neboť objekt osoba může vlastnit více psů (přesněji řečeno, tyto informace takto včlenit *lze*, ale toto řešení se nám nemusí líbit z hlediska návrhu). Obdobná situace je i u třídy Pes. Pokud vložení atributů vztahu do jedné z asociovaných tříd odporuje koncepci návrhu, můžeme vztah povýšit na třídu (tj. vztah je realizovaný třídou).



Obrázek 4.3: Povýšení asociace na třídu.

4.3.2 Kompozice a agregace

Kompozice a agregace jsou speciálními typy asociace, které říkají, že jeden objekt obsahuje (je složen) z jiných objektů. Těmto typům asociace se také říká asociace celek/část. Rozdíl mezi agregací a kompozicí pak spočívá v "síle" vztahu. Pro obě varianty platí:

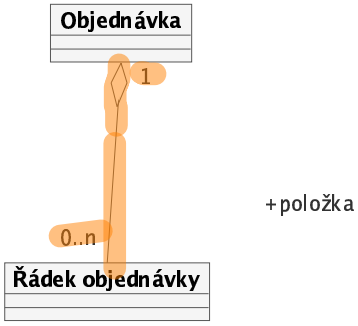
- Objekt "část" nemůže obsahovat (být složen či seskupen) objekt "celek", ve kterém je obsažen.
- Role objektu "část" se zobrazuje na konci asociační linie. Role objektu "celek" se neoznačuje (je zřejmá).
- Asociace celek/část nemá název (většinou nemá smysl).

Agregace (seskupení, aggregation) vyjadřuje slabší vztah mezi objektem reprezentujícím celek a objekty reprezentující části celku. Objektu reprezentujícímu celek se říká agregační (seskupený) objekt, jeho částem pak konstituční(tvořící) objekty. Vlastnosti agregace:

- Seskupený objekt může existovat bez svých konstitučních objektů.
- Konstituent (konstituční objekt) může být součástí více seskupení.
- Implicitní násobnost se nedá předpokládat.
- Agregace bývají homeometrické (tj. konstituenti patří do téže třídy).

Ukázka agregace. Celá objednávka sestává z jednotlivých řádků (položek) objednávky. Položky objednávky nejsou pevně svázány s celou objednávkou, mohou se dynamicky vytvářet a zanikat, aniž by se tím objekt objednávka stával neplnohodnotným.

x+y



Obrázek 4.4: Agregace tříd.

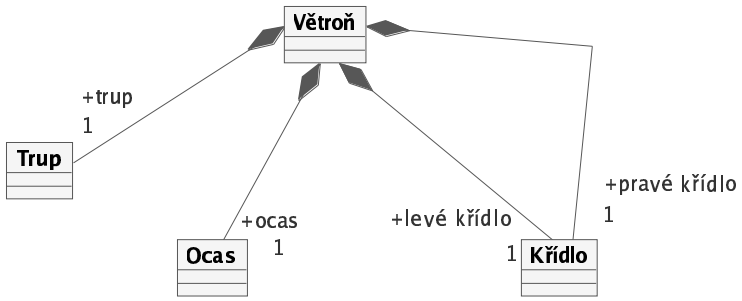
Kompozice (složení, composotion) vyjadřuje silnější vztah mezi objektem reprezentujícím celek a objekty reprezentující části celku. Objektu reprezentujícímu celek se říká kompozitní (složený) objekt, jeho částem pak komponentní (složkové) objekty. Vlastnosti kompozice:

DEF

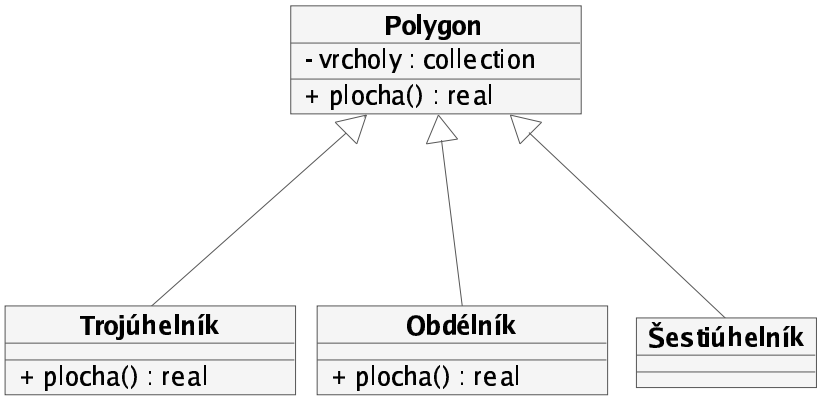
- Složený objekt nemůže existovat bez svých komponent. Složený objekt většinou řídí životní dráhu svých komponent (tj. komponenty vznikají a zanikají společně se složeným objektem).
- Komponentní objekt může být součástí pouze jedné kompozice.
- Implicitní násobnost každé složky je 1.
- Kompozice bývají heterometrické (tj. komponenty patří do různých tříd).

Ukázka kompozice. Větroň se skládá (zjednodušeně) z trupu, ocasu a pravého a levého křídla. Pokud některé z těchto součástí chybí, nejedná se o plnohodnotného větroně – z pohledu objektové abstrakce takový objekt nemůže plnit svou funkci.

x+y



Obrázek 4.5: Kompozice tříd.



Obrázek 4.6: Zobecnění tříd.

4.3.3 Zobecnění

Zobecnění zachycuje vazbu mezi třídami, kterou již známe – dědičnost. Zobecnění se zobrazuje šipkou s prázdným trojúhelníkem. Směr šipky udává směr zobecnění, tedy od speciální třídy k její nadtřídě.

Příklad. Diagram zobrazuje jednu obecnější třídu a její 3 specializace. Obecnější třída `Polygon` má metodu `plocha`, která počítá plochu (obsah) polygonu. Protože se jedná o obecný typ polygonu, je i algoritmus výpočtů obecný pro všechny polygony. Odvozené třídy `Trojúhelník` a `Obdélník` jsou speciální variantou polygonu a algoritmus výpočtu plochy je jednodušší - má tedy smysl předefinovat metodu `plocha`. Třída `Šestiúhelník` v našem návrhu nepředefinováva metodu `plocha` a pro výpočet plochy tudíž používá metodu nadřazené (obecné) třídy `Polygon`.

Modifikaci metody v podřízené (speciální) třídě se říká **přepisování (overriding)**. V některých jazycích se můžete setkat také s pojmem **přetěžování (overloading)**. Přetěžování je technika vícenásobné definice metody stejného jména v jedné třídě (příp. více funkcí stejného jména u procedurálních jazyků). Přetěžování

se většinou realizuje změnou typu či počtu argumentů metody.

Ukázka přetěžování metod v jazyce Java. Jde o dvě metody stejného jména převádějící peníze na zadaný účet. První varianta má dva argumenty: kam a kolik se má převést. Druhá varianta převádí celý zůstatek účtu.

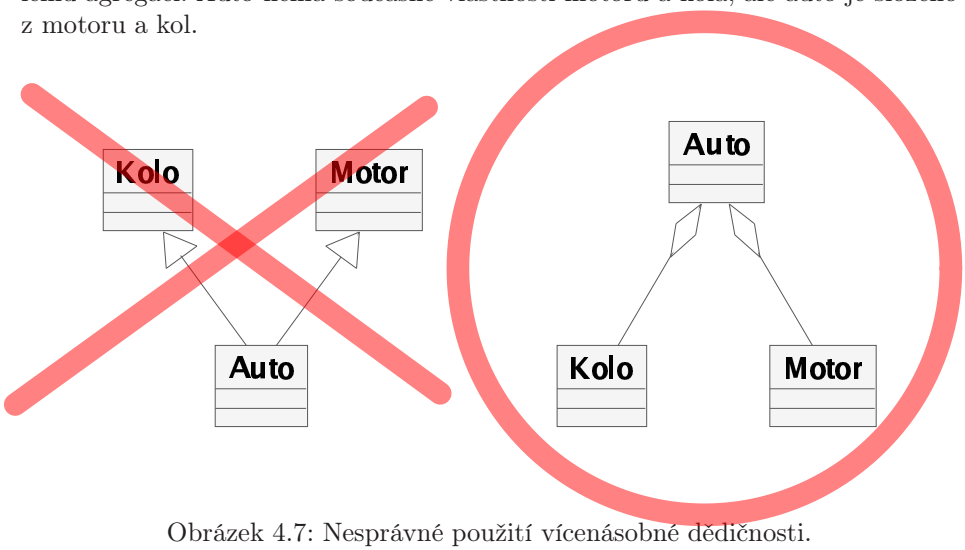
```
prevedNa(Ucet u, int castka);
prevedNa(Ucet u);
```

Jazyk Smalltalk přetěžování nezná, v tomto případě by se jednalo o dvě metody s různými jmény.

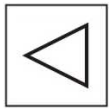
```
preved: castka na: u.
prevedNa: u.
```

Vícenásobná dědičnost umožňuje zjednodušení návrhu, jejím nesprávným použitím se však dá návrh zatemnit a zkomplikovat, neboť s sebou nese problémy se sémantikou (nejednoznačnost). Problémem nejednoznačnosti je především konflikt jmen. Pokud třída dědí dvě nadřazené třídy, které obě definují stejnou metodu, není jednoznačné, která metoda se má volat (tuto situaci lze explicitně ošetřit). Vícenáslobná dědičnost se dá obejít skládáním objektů (tříd). V některých případech je využití vícenásobné dědičnosti výhodné – znakem dobrého vývojáře je mj. i to, zda dokáže odhadnout, kdy je vhodné místo agregace použít vícenásobnou dědičnost.

Ukázka nesprávného použití vícenásobné dědičnosti a řešení daného problému agregací. Auto nemá současně vlastnosti motoru a kola, ale auto je složeno z motoru a kol.



Obrázek 4.7: Nesprávné použití vícenásobné dědičnosti.

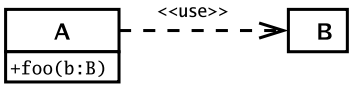


Jazyk Self disponuje vícenásobnou dědičností na úrovni objektů (tj. vícenásobným delegováním). Každý objekt může definovat (a za běhu i měnit) seznam objektů, které jsou mu "nadřazeny" (tj. seznam objektů, na které lze delegovat zprávy). Problém konfliktu jmen je řešen prioritou přístupu k nadřazeným objektům.

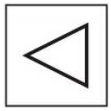
4.3.4 Závislost

Závislost vyjadřuje jiné různé vztahy mezi objekty či třídami. Typ závislosti se v diagramech označuje pomocí stereotypů. Nejběžnějším vztahem závislosti je používání (stereotyp `«use»`) – proto se závislost bez stereotypu považuje za používání. Závislost používání říká, že objekt třídy A (klient) sice neobsahuje objekt třídy B (dodavatel), tedy není jeho složkou, ale přesto jej potřebuje. Typickými příklady jsou:

- operace třídy A potřebuje argument třídy B
- operace třídy A vrací objekt třídy B
- operace třídy A používá třídu B, např. vytváří jeho dočasnou instanci



Jen přehledově si ukážeme jiné typy závislostí:



- `«instantiate»` – klient je instancí dodavatele
- `«trace»` – obecná vazba mezi elementy, které jsou na různé úrovni abstrakce
- `«refine»` – upřesnění, např. dvě třídy, jedna z nich je optimalizovaná na výkon
- `«friend»` – řízené narušení zapouzdření, klient má přístup k privátním prvkům dodavatele (je lepší nepoužívat, z nejrozšířenějších jazyků je podporována pouze jazykem C++).

4.3.5 Realizace

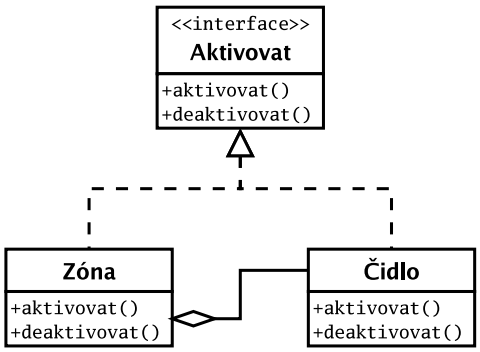
Realizace je vztah mezi třídou a rozhraním, který říká, že třída implementuje všechny operace z daného rozhraní. Libovolný objekt pracující s tímto rozhraním pak umí pracovat i s jeho implementačními třídami.

Rozhraní je speciální typ třídy, která specifikuje pouze množinu operací, ne však jejich implementaci. Rozhraní odděluje specifikaci funkčnosti (tj. co má třída umět) od implementace (tj. jak toho má dosáhnout), což umožňuje flexibilnější modelování, neboť omezuje počet vazeb mezi třídami. Dost často nás pouze zajímá, co objekt umí, tedy jakým zprávám rozumí – v tom případě můžeme pracovat pouze s rozhraním. Na místě rozhraní pak lze použít libovolnou třídu, která toto rozhraní implementuje (realizuje).

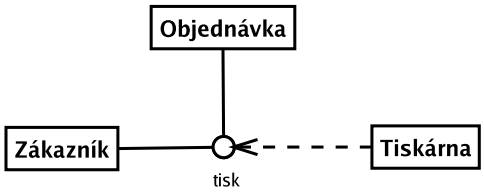
Ukázka znázornění rozhraní. Rozhraní `Aktovovat` definuje dvě operace. Třídy `Zóna` a `Čidlo` implementují rozhraní `Aktivovat`.

Ukázka alternativního znázornění rozhraní. Rozhraní `tisk` je implementováno třídami `Zákazník` a `Objednávka`. Rozhraní `tisk` používá třída `Tiskárna` – může tedy používat instance tříd `Zákazník` a `Objednávka`, aniž bychom tuto vazbu museli explicitně vyjadřovat.

Zdroje



Obrázek 4.8: Rozhraní.



Obrázek 4.9: Rozhraní – alternativní notace.

1. Definice silně typovaných jazyků, http://en.wikipedia.org/wiki/Strongly-typed_programming_language, *únor 2006*

2. Jazyk UML, http://www.uml.org, *únor 2006*

Kontrolní otázky

4.1 Stručně charakterizujte rozdíl mezi objektově orientovaným a strukturovaným přístupem k analýze a návrhu.

4.2 Stručně charakterizujte pojem abstrakce.

4.3 Stručně charakterizujte pojem polymorfismus.

4.4 Stručně charakterizujte pojem zapouzdření.

4.5 Stručně charakterizujte pojem dědičnost.

4.6 Stručně charakterizujte pojmy časná vazba, pozdní vazba a přepisování.

4.7 Vysvětlete pojmy identita objektu a shodnost objektů.

4.8 Čím se vyznačují třídně založené jazyky? Jaké třídně založené jazyky znáte?

4.9 Čím se vyznačují prototypově založené jazyky? Jaké prototypově založené jazyky znáte?

4.10 Stručně charakterizujte staticky typované jazyky.

4.11 Stručně charakterizujte dynamicky typované jazyky.

4.12 Vyjmenujte a stručně charakterizujte základní vztahy mezi třídami.

4.13 Identifikujte vztahy mezi třídami na obrázku 4.2 a vyjmenujte informace o jednotlivých vztazích.

Kapitola 5

Analýza a specifikace požadavků

2:00

V této kapitole se dozvíte

Cílem této kapitoly je seznámení se s prvotními etapami životního cyklu softwaru – jsou to specifikace požadavků a analýza požadavků. Budeme se věnovat technice specifikace požadavků s využitím diagramů případů užití z jazyka UML.

5.1 Činnosti spojené s vývojem softwaru

Vývoj softwaru sestává typicky z několika podstatných kroků (etap). Tyto etapy se mohou přizpůsobovat konkrétním požadavkům softwaru, použité metodologii vývoje softwaru apod., avšak jejich rámec a význam zůstává stejný. Empiricky bylo zjištěno, že jednotlivé základní etapy pokrývají určitý podíl v životě softwaru. Tyto etapy a jejich podíl jsou znázorněny v následující tabulce:

analýza a specifikace požadavků	**8 %**
architektonický a podrobný návrh	**7 %**
implementace	**12 %**
integrace (např. se stávající částí SW) a testování	**6 %**
provoz a údržba	**67 %**

Jak vyplývá z uvedené tabulky, vlastní vývoj softwaru spotřebuje asi 33% z celkové doby životního cyklu softwaru. Tato doba obsahuje i integraci a testování, což se dá považovat za zkušební provoz vyvíjené aplikace. Asi dvě třetiny doby životního cyklu představují provoz a údržba softwaru, tedy plnění úkolů, pro které byl software vytvořen. Pochopitelně existují aplikace běžící mnohem déle než jsme si uvedli – uvedená čísla je nutné brát jen jako orientační průměr pro většinu aplikací. Pokud se zaměříme na dobu strávenou během vlastního vývoje (tj. cca jedna třetina), zjistíme, že analýza a návrh zabírají více času než implementace softwaru. Skutečně lze říci, že dobrý návrh a (především) analýza jsou základem úspěšného projektu. Různé metodiky dokonce doporučují zvýšit poměr návrhu a testování na úkor implementace (viz např. pravidlo 40-20-40).

Softwarové inženýrství je věda snažící se vnést řád a rigoróznost do oblasti vývoje softwaru, tedy do oblasti, ve které se pracuje s nestabilními prvky, jakou jsou člověk a různé sociální skupiny (vývojový tým, uživatelé atd.) V této oblasti lze predikovat pouze na základě zkušeností s chováním a vývojem různých situací, přičemž tyto odhady nejsou nikdy přesné. Různé metodiky životního cyklus se snaží tyto nejistoty eliminovat a dovést softwarový projekt alespoň k nějakému úspěchu. Existují přístupy, které striktně neoddělují fáze specifikace, návrhu a testování, ale považují je za činnosti, které je nutné provádět souběžně (viz např. agilní metodologie).

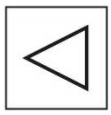
5.2 Analýza a specifikace požadavků

Cílem analýzy a specifikace požadavků je stanovení služeb, které zákazník požaduje od systému a vymezení podmínek jeho vývoje a provozu. Smyslem této etapy je především

- *Získat, analyzovat a definovat požadavky.* Jako úplně první krok je nutné, s využitím vhodných prostředků, zjist, o jaký softwarový produkt má zákazník zájem, vymezit jeho funkcionalitu atd. V této fázi je kladen důraz na požadavky uživatele, nikoli na to, jak jich lze toho docílit!
- *Transformace neformálních požadavků uživatele do strukturovaného popisu požadavků.* Požadavky jsou od zákazníka většinou získány ve formě neformálního seznamu či popisu, který je *nutné* převést do částečně formálního, strukturovaného popisu. Nutnost transformace vyplývá z potřeby získat tak jednoznačné zadání jak je to jen možné. Tím lze předejít potenciálním budoucím potížím ve stylu *"my jsme původně chtěli něco úplně jiného"*. Specifikace se pochopitelně časem mění, důležité je mít pevný výchozí bod a veškeré změny také dokumentovat.
- *Provedení studie vhodnosti, identifikace a analýza rizik.* Pokud známe požadavky zákazníka, měli bychom provést i analýzu rizik, tj. odhadnout, zda jsme schopni požadovaný produkt vytvořit za předpokládanou cenu, v předpokládané době apod. Jinak řečeno, měli bychom, na základě dostupných informací, řici, zda je tento projekt v našich silách.
- *Plánování akceptačního testování.* Součástí smlouvy se zákazníkem by měly být i plány akceptačního testování, tj. plány jak se bude výsledný produkt testovat při přebírání zákazníkem, jaké výsledky testů budou považovány za úspěšné atd.

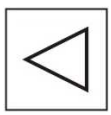
Řekli jsme si, že je v této etapě kladen důraz na požadavky, nikoli na realizaci. Při provádění studie vhodnosti však musíme mít na zřeteli i způsob řešení, jaký by byl pro daný typ projektu vhodný. Stejný problém (požadavek) může mít více alternativ řešení, přičemž každá alternativa s sebou vždy nese nějaké výhody i nevýhody, které musí být brány v potaz. Také je dobré si uvědomit, že náš produkt bude používat více různých uživatelů, kterí mohou mít různé názory na to, jaké má být např. uživatelské rozhraní.

Pro zajímavost si uvedeme, ve kterých etapách a s jakou pravděpodobností dochází k odhalení chyb, které se neodhalily při specifikaci požadavků:



při návrhu	65%
při implementaci	2%
při testování	30%
v provozu	3%

Náklady na opravu chyb ve specifikaci rostou tím více, čím později (v pozdějších fázích vývoje) jsou odhaleny. Následující tabulka uvádí odhad nákladů podle toho, v jaké etapě došlo k odhalení chyby (můžeme vidět, že mezi etapami je nárůst asi trojnásobný):



Etapa	Náklady (člověko-hodiny)
Specifikace	2
Návrh	5
Implementace	15
Akceptační testování	50
Údržba	150

5.3 Typy požadavků

Požadavky na software mohou být různorodé a nemusí vždy souviset pouze s tím, co je nutné naprogramovat. Důležitou vlastností každého požadavku je měřitelnost. Jen u měřitelných požadavků jsme schopni určit, zda výsledná aplikace tyto požadavky splňuje či nesplňuje. Součástí definice požadavku, který je více subjektivní (typicky uživatelské rozhraní) by tedy měla být i specifikace hodnocení míry splnění požadavku. Požadavky můžeme rozlišit do několika základních kategorií.

- *Funkcionální požadavky.* Základní typ požadavků určující co má vyvíjený systém dělat, jaká je jeho funcionalita. Příkladem funcionálních požadavků je výpočet mezd, rezervace letenek apod.
- *Požadavky na provoz systému.* Definují podmínky, za jakých má aplikace pracovat, např. kolik uživatelů k ní bude souběžně přistupovat, jaká doba odezvy se považuje za akceptovatelnou apod.
- *Požadavky na výsledný systém.* Definují především požadavky související s vývojem a nasazením systému – počítačové vybavení (aplikace musí běžet na zadaném procesoru se zadanou pamětí), programové vybavení (požadavek na operační systém, programovací jazyk), spolehlivost, odolnost vůči chybám, přenositelnost, bezpečnost atd.
- *Požadavky na vývojový proces.* Definují především požadavky zákázníka na dodržování norem během vývoje a předávání systému.
- *Požadavky na rozhraní.* Jaké jsou požadavku na rozhraní systému se svým okolím, ať už se jedná o uživatelské rozhraní, nebo např. rozhraní na jiné součásti systému.
- *Externí požadavky.* Požadavky vyplývající z charakteru aplikace v návaznosti např. na legislativní požadavky (soulad se zákonem v oblasti ochrany informací apod.) či požadavky etické.

5.4 Specifikace požadavků

5.4.1 Metody získávání informací

Kvalitní získávání informací o problémové oblasti a požadavcích snižuje riziko vytvoření systému, který nebude vyhovovat požadavkům uživatele. Důležitá je motivace ze strany zákazníka (uživatele) – zákazník se musí podílet na definici cílů a měl by v prvotních fázích intenzivně spolupracovat s analytiky. Jedno z umění dobrého analytika spočívá ve správné motivaci zákazníka (uživatele) na spolupráci. Existuje celá řad metod, jak získávat informace, my si zde uvedeme ty nejběžnější:

- interview (orientační, strukturované)

- dotazníky
- studium dokumentů
- pozorování prací u zákazníka
- přímá účast na pracech zákazníka
- analýza existujícího softwarového systému

5.4.2 Problémy při specifikaci požadavků

Největší problémy při specifikaci požadavků pramení z podstaty získávání informací, tj. komunikací se zákazíkem. Pro komunikaci se zákazníkem se používá přirozený jazyk, který není jednoznačný. Vzniká tedy problém srozumitelnosti specifikovaných požadavků, neboť různí lidé mohou rozumět stejné větě jinak. Důležitým faktorem je také kontext, ve kterém se věty pronášejí – co nám je jasné nyní, nemusí být jasné za měsíc (zapomeneme v jakých souvislostech byly věty řečeny). Problémy plynoucí z použití přirozeného jazyka mohou být tyto:

- *Vyřazení* – Neuvedení entity v textu specifikace. V době specifikace byla tato entita implicitní (alespoň každému zúčastněnému připadalo jasné o koho/co se jedná), avšak s odstupem času se tato implicitní informace může vytratit.
  - Příklad: *Používají systém k výpůjčkám knih.*
  - Kdo systém používá? Přímo klienti knihovny, pouze personál, nebo skutečně všichni?
- *Deformace, zkreslení* – Informace může vyvolávat zavádějící výklad, a to např. tím, že informace není úplná nebo se opět předpokládá implicitní podmínka, která není zřejmá dalším lidem.
  - Příklad: *Čtenáři si nemohou půjčit další knihu, dokud nevrátí knihy s prošlou výpůjční lhůtou.*
  - Pokud čtenář knihu nevrátí, má možnost ji zaplatit a znovu si půjčovat další knihy!
- *Zobecnění* – Věta je příliš obecná a ve svém důsledku nepravdivá.
  - Příklad: *Každý, kdo si chce vypůjčit knihu, musí mít průkazku.*
  - Pokud půjde o výpůjčky mezi knihovnami, pak není průkazka nutná. Lepší formulace by tedy měla znít např. takto: *Každý zákazník knihovny, který se chce vypůjčit knihu, musí mít průkazku.*

Jedním z možných způsobů jak předcházet případným nedorozuměním je používat *slovníček pojmů*. Slovníček pojmů zachycuje terminologii pro daný projekt (např. termíny používané v obchodním jazyce), řeší synonyma (tj. definuje množinu slov, které se v daném kontextu považují za synonyma), řeší homonyma (definuje jednoznačný význam daného slova) apod.

Další problémy vznikají z následujících příčin:

- Zákazník není schopen přesně formulovat požadavky. Zákazník nemá na začátku projektu přesnou představu, co vlastně požaduje, jaký by měl navrhovaný systém být a co vše by měl umět. Dost často nemá představu o náročnosti vytváření jednotlivých prvků systému – to, co se mu jeví jako "hračka", může být dosti komplikované a naopak.

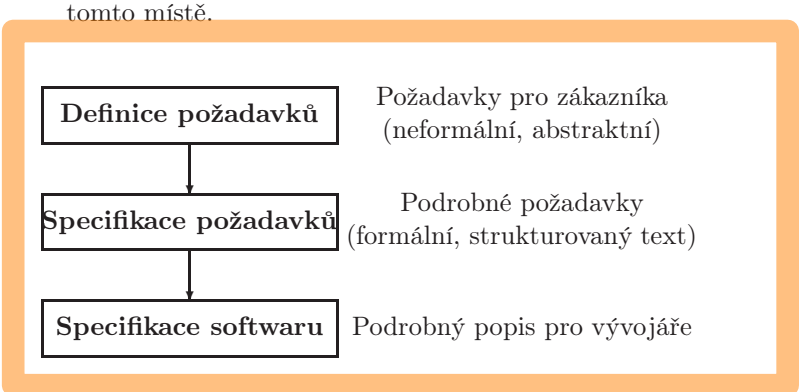
- Zákazník se většinou neorientuje v problematice vývoje softwaru, vývojář (analytik) se nemusí orientovat v doménové problematice – problém různorodé terminologie. Vhodné řešení je vyčlenění člověka (příp. více lidí) od zákazníka, který je odborník na problémovou doménu a orientuje se v problematice vývoje softwaru, a který spolupracuje s vývojovým týmem. Ve vývojovém týmu by zase měl být specialista na řešenou problematiku, případně je možné specialistu–analytika na uvedený projekt najmout.
- Rozhodování, jaké požadavky už nezačleňovat do specifikace – tj. problém nalezení vhodné míry toho, co vše je reálné zprovoznit. V ideálním případě obsahuje specifikace nejmenší množinu toho, co je nezbytně nutné pro provoz systému.

Pro úspěšné zvládnutí etapy analýzy a specifikace požadavků jsou důležité komunikační schopnosti a zkušenosti analytika. Na závěr je velmi vhodné nechat si výsledek zkontrolovat a potvrdit druhou stranou (tj. zákazníkem) – vyhnete se potenciálním problémům v budoucnu!

5.4.3 Kroky při specifikaci požadavků

Etapa specifikace požadavků obsahuje několik kroků, které jen vhodné vykonat. My si je v této kapitole ve stručnosti přiblížíme.

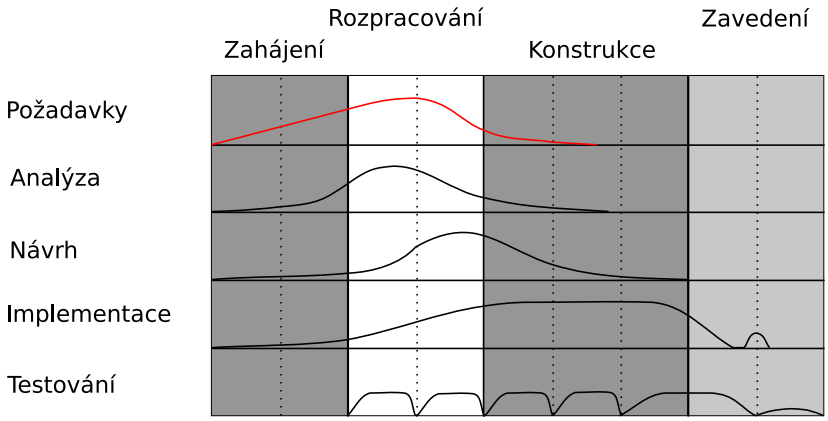
- *Studie vhodnosti.* Prvotní odhad, zda jsme schopni splnit požadované zadání v navrhovaném rozsahu, čase a s jakými náklady. Součástí studie vhodnosti by měl být i odhad počtu pracovníků, jaké technologie by se měly nasadit apod. Důležitou podmínkou studie vhodnosti je její *rychlé* a *levné* provedení. Cílem je určit, zda je účelné participovat na daném projektu.
- *Analýza požadavků.* Analýza představuje zkoumání požadavků zákazníka, zkoumání současného stavu (již existující podobné systémy, komponenty, řešení atd.)
- *Definování požadavků.* Transformace informací z analýzy požadavků do dokumentu určeného především pro uživatele a zákazníka.
- *Specifikace požadavků.* Transformace informací z analýzy požadavků a definice požadavků do pokud možno formálního, strukturovaného dokumentu určeného pro vývojáře. Tento dokument se soustřeďuje se na software, nikoli na proces jeho tvorby! Velmi často se vytváří paralelně s architektonickým návrhem.
- *Specifikace softwaru.* Dokument popisující architekturu vyvíjeného softwaru. Tento dokument je výstupem etapy návrhu, pro úplnost jej uvádíme i na tomto místě.



5.5 Prostředky pro specifikaci požadavků v RUP

Jak můžeme vidět na obrázku 5.1, specifikace požadavků je první etapa v životním cyklu softwaru. RUP pracuje v iteracích, po první iteraci se souběžně s etapou specifikace aktivuje etapa analýzy požadavků atd. To umožňuje rychle reagovat na nalezené nedostatky v pokročilejších etapách a modifikovat podle nich specifikaci požadavků (ve spolupráci se zákazníkem). Výstupem etapy specifikace požadavků jsou většinou

- diagramy případů užití,
- detaily případů užití,
- specifikace (strukturovaný text),
- slovníky pojmů.



Obrázek 5.1: RUP model vývoje softwaru – etapa specifikace požadavků.

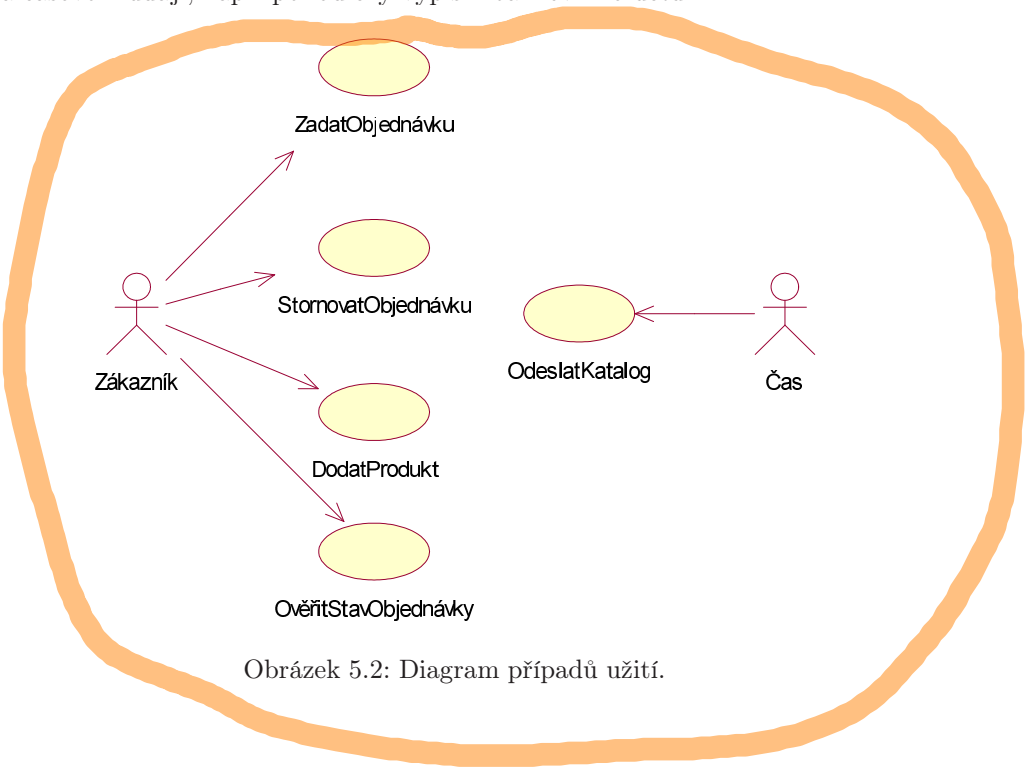
5.5.1 Případ užití

Pro zachycení požadavků se v RUP používají diagramy případů užití (*use cases*), které jsou součástí modelovacího jazyka UML. Klíčovými aktivitami metodiky RUP ve fázi specifikace požadavků jsou nalezení případů užití a jejich účastníků a nalezení detailů vybraných případů užití. Jeden případ užití je chápán jako funkce, kterou systém vykonává jménem jednotlivých účastníků nebo v jejich prospěch. Každý případ užití má svůj *název*, *jednoznačný identifikátor* a *specifikaci* (viz kapitola 5.5.2). Diagramy užití znázorňují:

- hranice navrhovaného/zkoumaného systému,
- účastníky analyzovaných případů užití,
- analyzované případy užití a
- interakce mezi aktéry a případy užití.

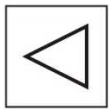
V laboratořích můžete použít software Rational Rose. Podívejte se na učební text zaměřený na jazyk UML 2.0, který máte k dispozici na privátních stránkách předmětu IUS!

Ukázka diagramu případu užití. Na obrázku 5.2 je ukázka diagramu případu užití, který zachycuje několik případů užití např. internetového obchodu zákazníkem (tj. případ užití *Zadat objednávku* apod.) Můžeme si všimnout speciálního aktéra *Čas* – tímto způsobem lze zachycovat případy, jejichž provedení závisí na časovém údaji, např. periodický výpis z bankovního účtu.

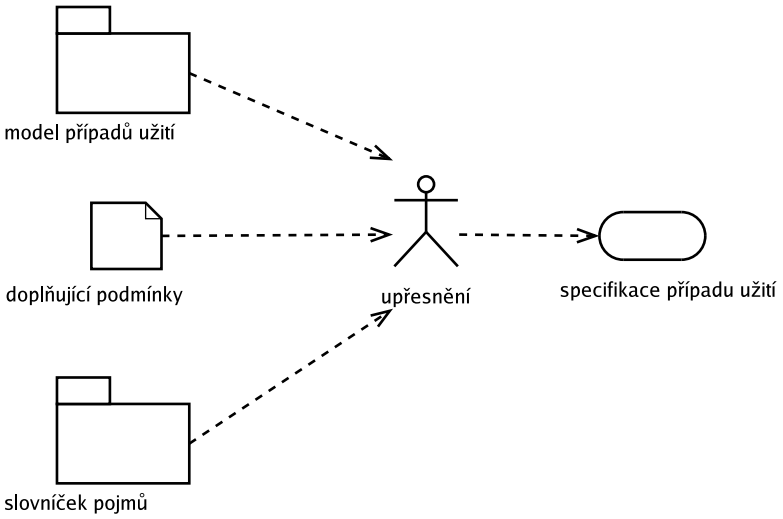


Obrázek 5.2: Diagram případů užití.

5.5.2 Detail případu užití



Samotné diagramy případů užití pouze ukazují, jaké akce je možné v systému vykonávat a kdo je může vykonávat. V mnoha případech je však tato informace málo konkrétní, nedá se z ní poznat co vše se děje a za jakých podmínek. Pro konkretizaci (specifikaci) případu užití existuje *Detail případu užití.* Pro jeho zobrazení neexistuje standard, většinou se však používá tabulka. Každá specifikace případu užití má své *vstupní podmínky, tok událostí* a *výstupní podmínky.* Následující diagram schématicky znázorňuje přechod od případu užití k jeho specifikaci – vstupními daty jsou diagram případu užití, dodatečné informace (většinou ze specifikace požadavků) a případně slovníček pojmů pro řešení nejednoznačností.



Ukázka specifikace případu užití. Budeme uvažovat případ placení daně z přidané hodnoty. Účastníky jsou čas a finanční úřad, vstupní podmínka je dosažení konce fiskálního čtvrtletí. Pokud je vstupní podmínka splněna, aktivuje se tok událostí. Ten v našem případě spočívá v těchto krocích – systém určuje výši daně z přidané hodnoty a následně zasílá elektronicky platbu finančnímu úřadu. Uvedený případ užití je velmi jednoduchý a nezahrnuje různé varianty možného vývoje. Pro tyto účely lze použít Specifikaci s alternativními kroky (viz následující příklad).

```
x+y
```

název
identifikátor

účastníci

stav před

kroky

stav po

Případ užití: Platit daň z přidané hodnoty
ID: UC1
Účastníci: Čas finanční úřad
Vstupní podmínky: 1. Je konec fiskálního čtvrtletí?
Tok událostí: 1. Případ užití začíná na konci fiskálního čtvrtletí. 2. Systém určuje výši daně z přidané hodnoty, kterou je třeba odvést státu. 3. Systém odesílá elektronickou platbu finančnímu úřadu.
Následné podmínky: 1. Finanční úřad přijímá daň z přidané hodnoty ve správné výši.

Ukázka specifikace případu užití s alternativními kroky. Budeme uvažovat případ zobrazní košíku v internetovém obchodu. V uvedené specifikaci vidíme dva možné typy alternativ. První je vyjádřena podmínkou v hlavním toku událostí (viz podmínku `KDYŽ`). Pokud je podmínka splněna, vykoná se specifická větev hlavního toku programu a pak se pokračuje dále. Pokud podmínka splněna není, tato větev se přeskočí. Druhý typ vyjádření alternativy je definování alternativních toků. Alternativní tok je aktivován kdykoli v průběhu hlavního toku, pokud jsou splněny jeho podmínky. Např. v naší ukázce je definován alternativní tok *"Zákazník může kdykoliv opustit košík."*, který může nastav kdykoli v průběhu hlavního toku.

x+y

Případ užití: Zobrazit košík
ID: UC11
Účastníci: Zákazník
Vstupní podmínky: 1. Zákazník je přihlášen do systému.
Tok událostí: 1. Případ užití začíná volbou "zobrazit obsah košíku". 2. KDYŽ je košík prázdný: 2.1 Systém oznámí Zákazníkovi, že košík neobsahuje žádné položky. 2.2 Případ užití končí. 3. Systém zobrazí seznam všech položek v nákupním košíku zákazníka včetně ID, názvu, množství a ceny každé položky.
Následné podmínky:
...
Alternativní tok 1: 1. Zákazník může kdykoliv opustit obrazovku košíku.
Následné podmínky:
Alternativní tok 2: 1. Zákazník může kdykoliv opustit systém.
Následné podmínky:

5.5.3 Pokročilé techniky modelování případů užití

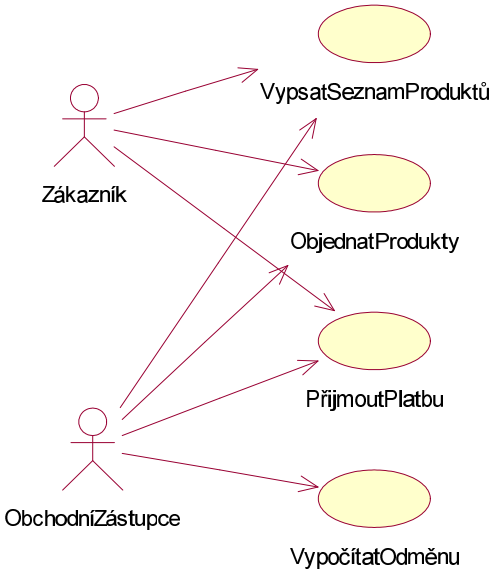
Základní diagram případu užití může být rozšířen o pokročilé techniky umožňující (při dobrém použití) lepší přehlednost a čitelnost zejména rozsáhlých modelů. Mezi pokročilé techniky patří:

- zobecnění účastníka
- zobecnění případu užití
- relace `«include»`
- relace `«extend»`

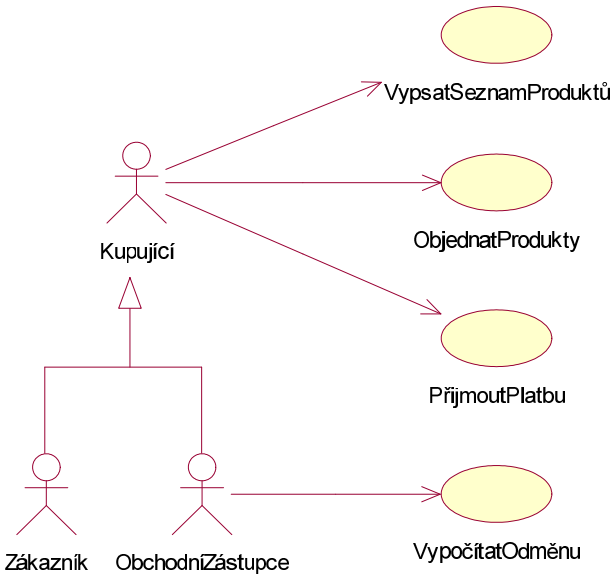
Zobecnění účastníka

Podobně jako v objektově orientovaném prostředí představuje zobecnění pomůcku pro vyjádření vztahu generalizace/specializace mezi účastníky a tím umožňuje zpřehlednit model (odstraní se nadměrné množství interakcí zobrazovaných v modelu).

Ukázka zobecnění účastníka. Na následujícím obrázku je model případů užití objednávkového systému, ve kterém vystupují dva aktéři – zákazník a obchodní zástupce. Oba aktéři mají možnost získat seznam produktů, objednat produkty a provést platbu. Protože obchodní zástupce má větší obrat, náleží mu odměna určená podle odebraného množství. Tedy z pohledu modelu případů užití existuje pouze jeden rozdíl mezi zákazníkem a obchodním zástupcem – obchodní zástupce si může vypočítat odměnu ze zisku.

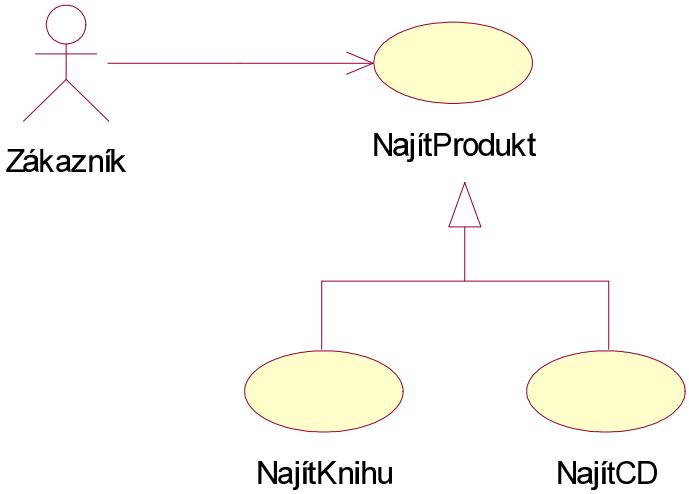


Můžeme tedy eliminovat počet zobrazovaných interakcí tím, že definujeme zobecněného aktéra (viz následující obrázek). Definujeme obecnějšího aktéra *Kupující*, který může získat seznam produktů, objednat produkty a provést platbu. Jeho speciálnější variantou je *Zákazník* mající stejné postavení a *Obchodní zástupce*, který může navíc vypočítat odměnu.



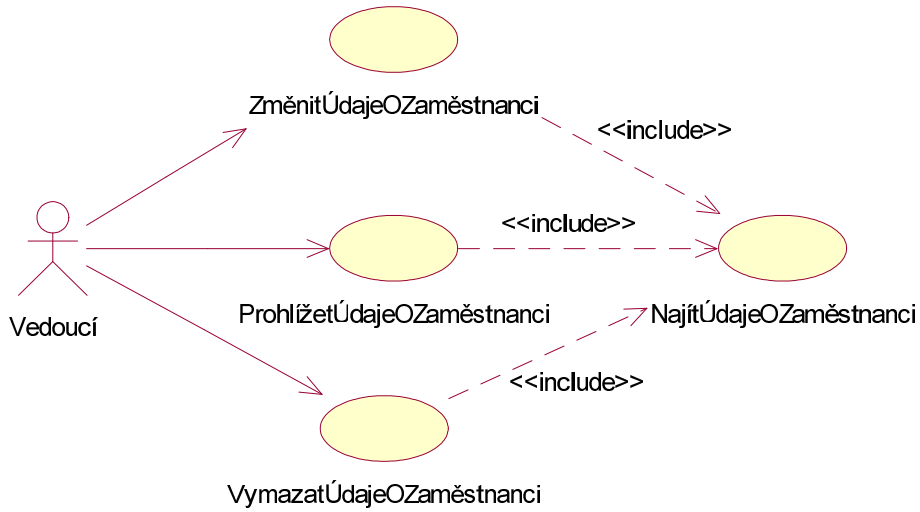
Zobecnění případu užití

Obdobný případ jako zobecnění aktéra. Pokud jeden či více aktérů používá více případů užití, které jsou svou funkcionalitou blízké, je možné zakreslovat tyto případy ve vztahu generalizace/specializace. Např. na následujícím obrázku zákazník může vyhledávat knihy a CD. Tyto funkce systému jsou si podobné, pouze pracují s jinými daty a mohou mít odlišné podmínky. Můžeme tedy vytvořit speciální případ *Najít produkt* a jeho speciální varianty. Aktér participuje pouze se speciálním případem, což znamená, že může také participovat na všech jeho speciálnějších variantách.



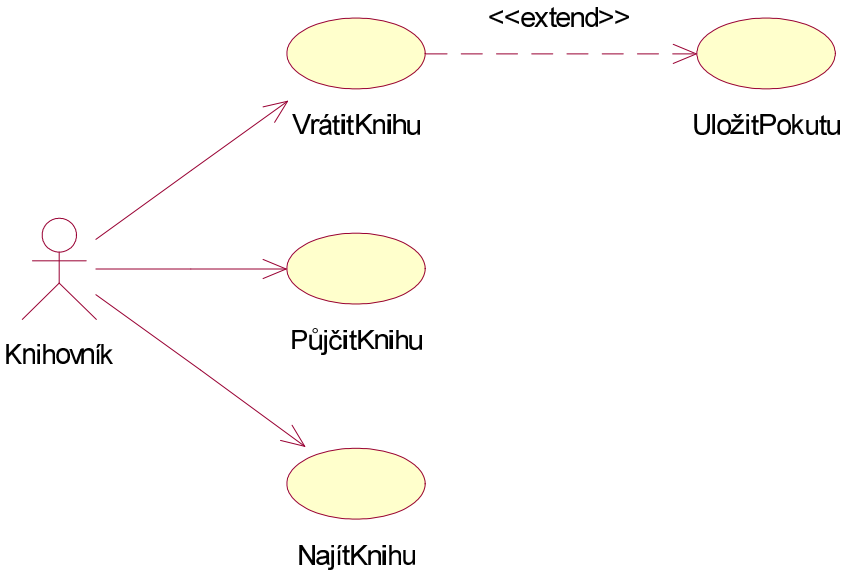
Relace «include»

Tato relace vyjadřuje situaci, kdy jeden případ užití zahrnuje v sobě jeden či více jiných případů užití. Příklad uvedený na následujícím diagramu zobrazuje případy změnit, prohlížet a vymazat udáje o zaměstnanci. Všechny tyto případy obsahují funkci pro vyhledání udajů o zaměstnanci, která je vždy stejná. Můžeme ji tedy vyčlenit jako speciální případ, který ostatní případy využívají (obsahují v sobě).



Relace «extend»

Tato relace vyjadřuje situaci, kdy jeden případ užití může rozšiřovat funkcionalitu jiných případů užití. Oproti relaci include, kde se vložený případ musí vždy použít, případ v relaci extend je nepovinný. Příklad uvedený na následujícím diagramu obsahuje případ *VrátitKnihu*. Součástí tohoto případu může být speciální varianta, kdy čtenář knihu např. ztratí a musí zaplatit pokutu. Tato situace nenastává vždy. V diagramu případů užití ji můžeme modelovat případem *UložitPokutu*, který rozšiřuje případ *VrátitKnihu*.



5.6 Prostředky pro analýzu v RUP

Výstupem etapy analýzy požadavků jsou většinou tyto diagramy v jazyce UML

- analytické třídy
- objektové diagramy
- analytické balíčky
- realizace případů užití
  - diagramy analytických tříd
  - diagramy interakce
- diagram aktivit

5.6.1 Analytické třídy

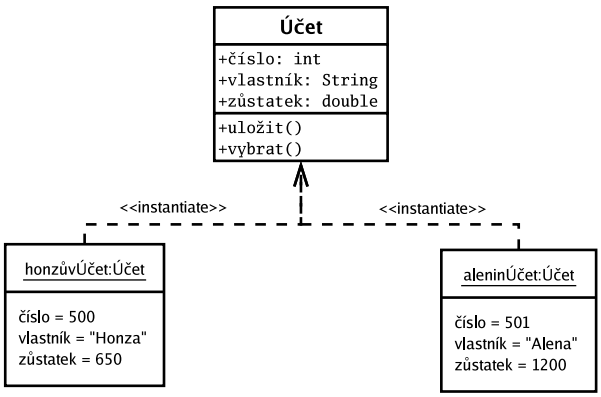
Analytické třídy mapují pojmy problémové domény na abstraktní entity modelu – třídy. Diagramy tříd a vztahy mezi třídami jsme rezebírali v kapitole 4.3, a proto se jimi tady nebudeme detailněji zabývat. Analytická třída není návrhová třída, tj. analytická třída slouží pouze pro identifikaci entit v řešené problematice a vztahů mezi nimi. Během etapy návrhu jsou analytické třídy upřesňovány do jedné či více návrhových tříd. Analytická třída

- obsahuje jen nejpodstatnější atributy a operace
- obsahuje minimální množinu odpovědností
- obsahuje minimum vazeb na jiné (*analytické*) třídy

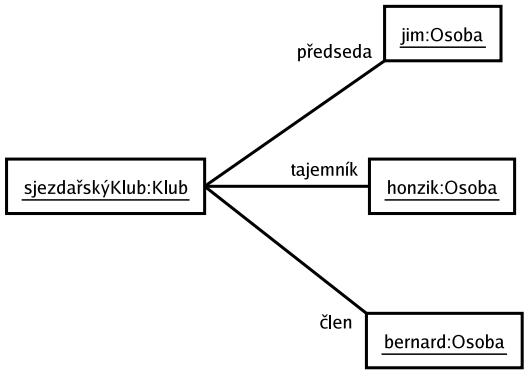
5.6.2 Objektové diagramy

Objektové diagramy patří mezi diagramy dynamické. Zachycují konkrétní instance tříd a jejich vazby v určitém čase či podmínce. V některých případech je vhodné zobrazit konkrétní vazby mezi objekty, zejména pro zdůraznění význačného aspektu nebo jako pomůcku při hledání tříd v řešené problematice a vazeb mezi nimi. Vazby mezi objekty se mohou v průběhu měnit a jejich abstrakce je zachycena právě v diagramu tříd.

Následující obrázek znázorňuje třídu `Účet` a její dvě instance `honzůvÚčet` a `aleninÚčet`. Vztah třída/objekt je naznačen závislostí `instantiate`. Jak můžeme vidět, definuje třída `Účet` tři instanční proměnné (proměnné objektu) a dvě metody (`uložit` a `vybrat`). Identifikace objektu sestává ze dvou částí – jména třídy, které je instancí (uvádí se za dvojtečku) a případného jména objektu (pokud objekt má jméno, uvádí se před dvojtečkou).



Další ukázka zobrazuje vztahy mezi čtyřmi objekty. Objekt `sjezdařskýKlub` (instance třídy `Klub`) má několik členů – `jim`, `honzik` a `bernard` (instance třídy `Osoba`). Uvedený vztah mezi objekty je znázorněn asociací. Můžeme vidět, že např. `jim` hraje v tomto vztahu, v aktuálním čase či podmínce, úlohu předsedy klubu.

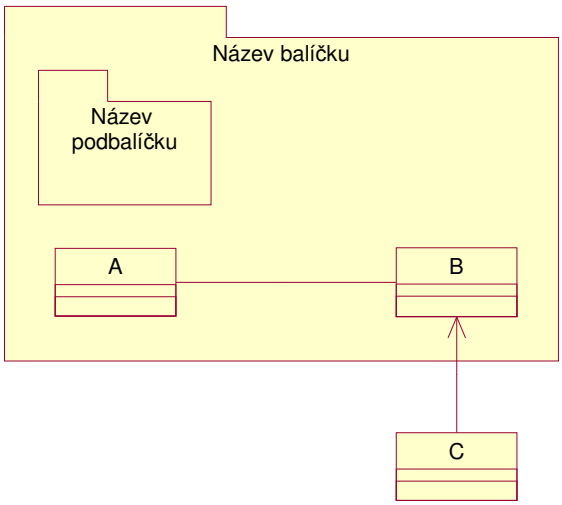


5.6.3 Analytické balíčky

Analytické balíčky souvisejí s dekompozicí problému – umožňují souběžnou práci na více částech (balíčcích) v etapě návrhu. Analytické balíčky seskupují sémanticky související elementy a definují hranice tohoto seskupení. Více balíčků můžeme řešit

souběžně. Balíček také poskytuje zapouzdření prostoru jmen (tj. názvy elementů musí být v rámci balíčku jedinečná, mezi balíčky však může docházet k duplicitám) a definuje viditelnost zapouzdřených elementů – *veřejné* (*public*), *soukromé* (*private*) a *chráněné* (*protected*). Analytické balíčky mohou obsahovat

- případy užití
- analytické třídy
- realizace případů užití
- další balíčky

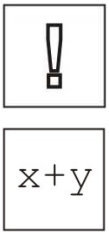


5.6.4 Diagramy interakce

Rozlišujeme dva typy diagramů interakce mezi třídami a jejími instancemi, které se od sebe liší kladeným důrazem buď na strukturální vazby nebo na časovou posloupnost vykonávaných akcí.

Diagram spolupráce (collaboration diagram)

Reprezentuje strukturální relace mezi objekty. Pro ilustraci si uvedeme jednoduchý příklad. Vyjdeme z následujího diagramu tříd:



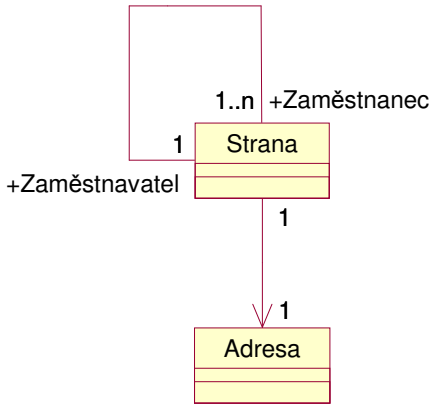


Diagram tříd zobrazuje třídu `Strana`, jejíž instance jsou v zájemném vztahu, ve kterém hrají buď roli zaměstnavatele nebo roli zaměstnace. Každý zaměstnavatel či zaměstnanec pak má přidělenu jednu adresu. V diagramu spolupráce zobrazujeme každou roli zvlášť. Elementy se označují identifikátorem s touto syntaxí a sémantikou:

- /NázevRole:NázevKlasifikátoru(Třídy)

Na obrázku tedy můžeme vidět roli Zaměstnavatel (instance třídy Strana) mající vazbu 1..N na instance třídy Strana s rolí Zaměstnanec. Každá taková instance má navíc vazbu 1..1 s instancí třídy Adresa (bez uvedení role).

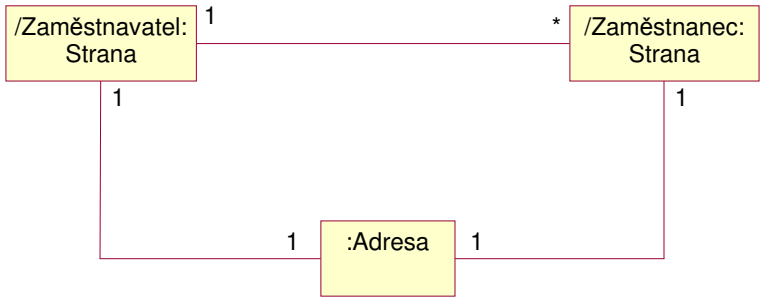


Diagram konkrétní spolupráce

Diagram konkrétní spolupráce zobrazuje skutečně instance příslušných tříd a jejich vazby. Oproti předchozí variantě je k popisu přídán další (nepovinný) parametr – název objektu:

- názevObjektu/NázevRole:NázevKlasifikátoru(Třídy)

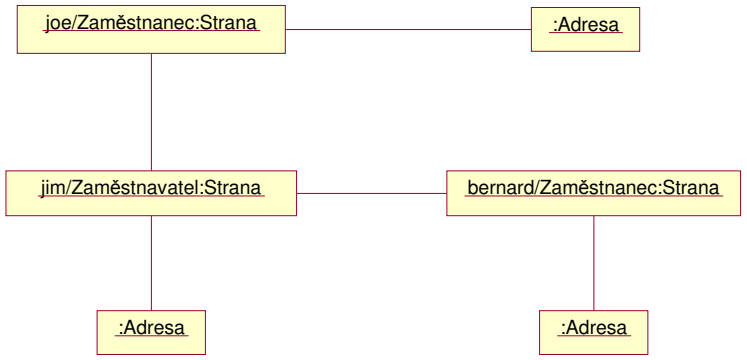


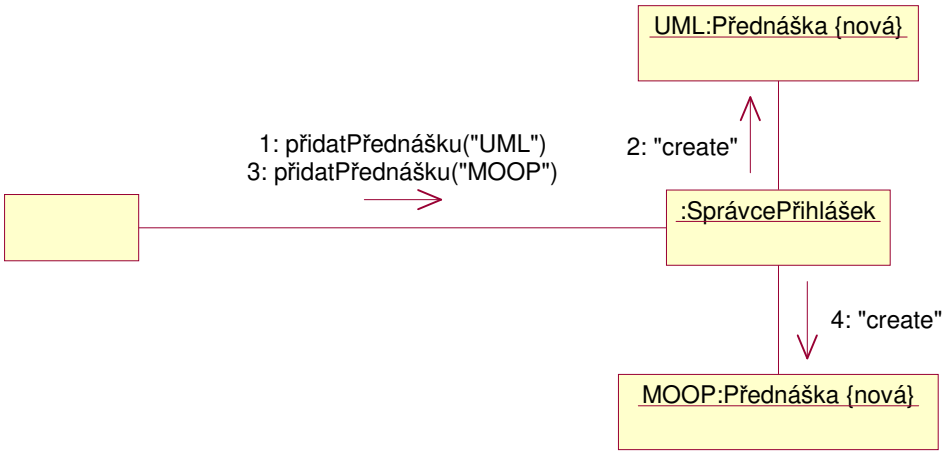
Diagram spolupráce s označením posloupnosti zpráv

V diagramu spolupráce můžeme zachytit zprávy zasílané mezi třídami (objekty) a jejich pořadí. Jsme tedy schopni sledovat vazby mezi objekty na základě zasílaných zpráv (viz následující příklad). V příkladu vyjdeme z digramu tříd a detailu případu užití – přidání přednášky do systému.

Případ užití: Přidat přednášku
ID: UC11
Účastníci: Kvestor
Vstupní podmínky: 1. Kvestor je přihlášen do systému.
Tok událostí: 1. Kvestor zadá příkaz "přidat přednášku". 2. Systém přijme název nové přednášky. 3. Systém vytvoří novou přednášku.
Následné podmínky: 1. Nová přednáška byla přidána do systému.

Diagram tříd:

Následující diagram spolupráce zachycuje tuto situaci: aktér zasílá zprávu *přidatPřednášku("UML")* (pořadí 1) správci přihlášek, který v reakci vytvoří instanci třídy `Přednáška` (viz zpráva 2: create). U objektu "UML:Přednáška" můžeme vidět stereotyp *{nová}* – tím můžeme naznačit, že se jedná o objekt, který dosud v systému nebyl. Obdobně pro přidání přednášky "MOOP".

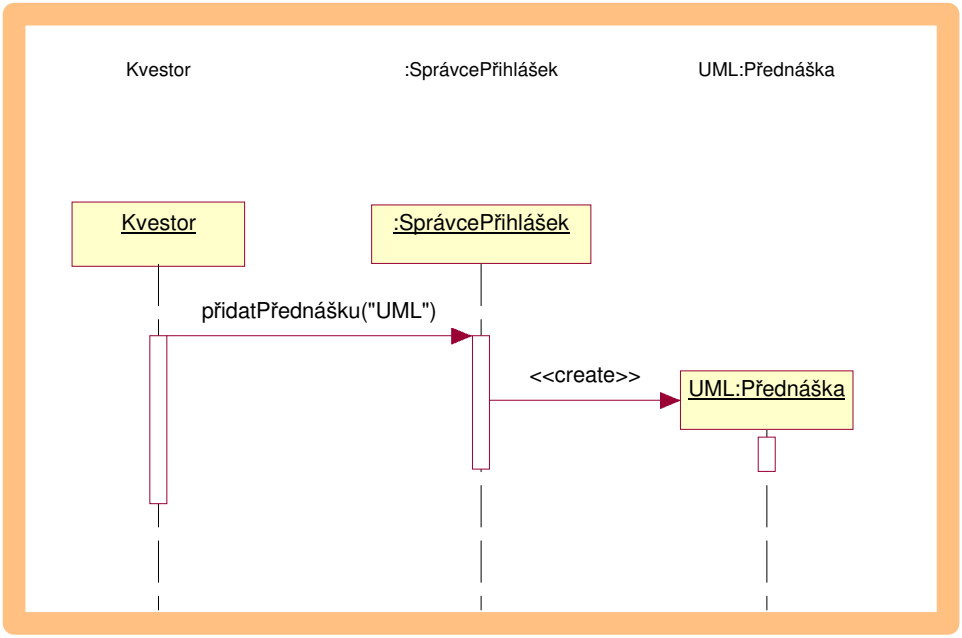


5.6.5 Sekvenční diagramy (sequence diagrams)

Reprezentují časově orientovanou posloupnost předávání zpráv mezi objekty. Sekvenční diagramy bývají přehlednější než diagramy spolupráce a uživatelé (zákazník) jim lépe porozumí. Každý objekt má svou časovou osu, která se zakresluje ve vertikálním směru shora dolů. Podle pozice zasílané zprávy na časové ose pak můžeme určit chronologii zasílání zpráv. Sekvenční diagram na obrázku 5.3 zobrazuje přidání přednášky do systému (viz příklad na diagram spolupráce).

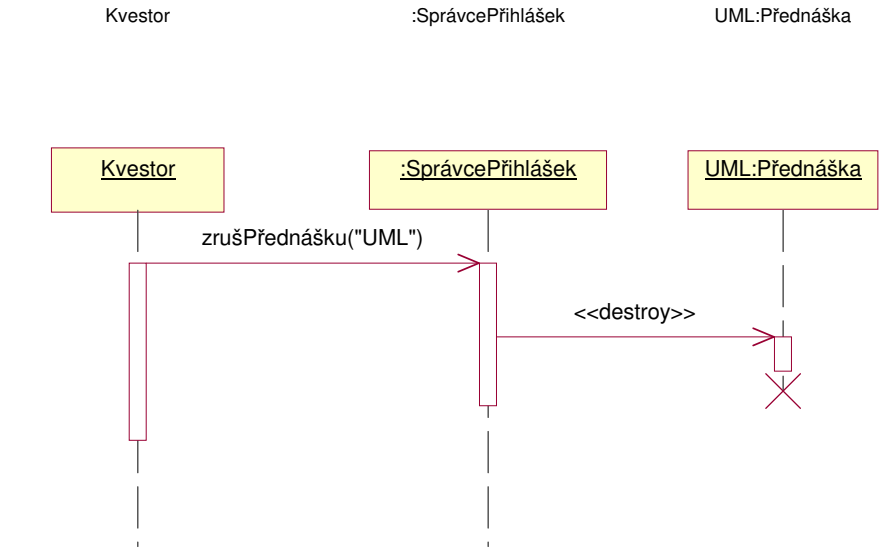
Sekvenční diagram na obrázku 5.4 zobrazuje zrušení přednášky. Ukončení života objektu je znázorněno křížem na konci časové osy objektu.

Sekvenční diagramy umožňují definovat omezení (constraints) či význačný stav objektu. Na následujícím obrázku jsou definovány 2 kontrolní body A a B a

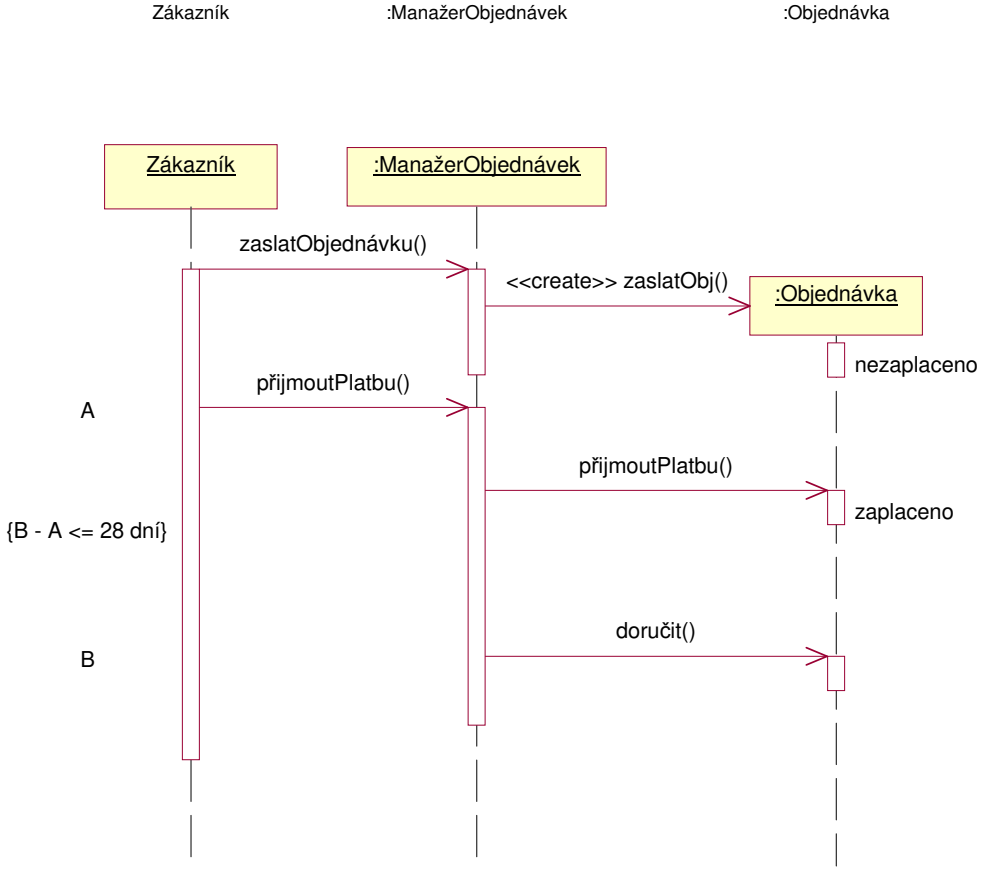


Obrázek 5.3: Sekvenční diagram – přidání přednášky

podmínka říkající, že rozdíl mezi A a B musí být menší než 28 dní. V tomto konkrétním případu toto omezení říká, že od doby příjmu platby manažerem až po odeslání objednaného zboží nesmí uběhnout více než 28 dní. Význačné stavy jsou zachyceny textově přímo u časové osy objektu po provedení žádané operace. Např. po vytvoření objednávky je tato ve stavu nezaplacená, v okamžiku příjmutí platby je ve stavu zaplacená.

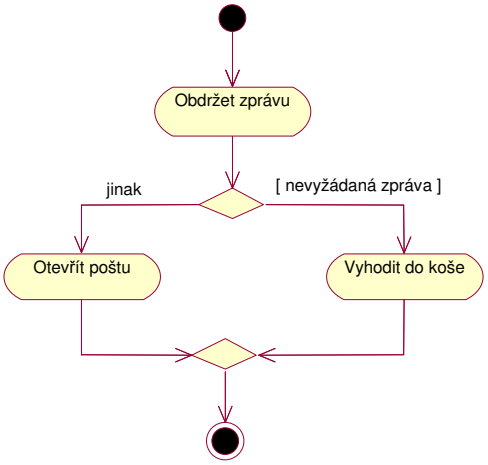


Obrázek 5.4: Sekvenční diagram – přidání přednášky



5.6.6 Diagramy aktivit

Diagramy aktivit reprezentují objektově orientované diagramy toků a lze je připojit k libovolnému modelovanému elementu (třída, balík, apod.) Diagram aktivit je zvláštním případem stavového automatu, který je určen především na modelování manažerských procesů. Diagram aktivit obsahuje *počátek a konec*, *stavy aktivity* a *přechody mezi stavy*. Na obrázku 5.5 můžeme vidět ukázku diagramu aktivity, který zobrazuje stavy a události spojené s příjmem dopisy (velmi jednoduchý model).



Obrázek 5.5: Diagram aktivity

Kontrolní otázky

5.1 Co je cílem analýzy a specifikace požadavků?

5.2 Jaké znáte typy požadavků?

5.3 Jaké jsou problémy při specifikaci požadavků?

5.4 Co je diagram případu užití a k čemu slouží?

5.5 Co je detail případu užití?

5.6 K čemu slouží diagramy analytických tříd a objektové diagramy?

5.7 Jaké znáte diagramy interakce, jaký je mezi nimi rozdíl?

Kapitola 6

Strukturovaný přístup k analýze

1:30

V této kapitole se dozvíte

Cílem této kapitoly je seznámit se se strukturovaným přístupem k analýze, zejména s datovým modelováním. Základním diskutovaným modelem bude ER diagram (Entity Relationship Diagram).

6.1 Základní modely strukturované analýzy

Úvodem si vysvětlíme základní rozdíl mezi objektovým a strukturovaným přístupem k analýze a návrhu obecně. Strukturovaná analýza a návrh chápe systém jako kolekci funkcí (procesů) operujících nad daty, tj. pracujeme (navrhujeme) strukturu dat a funkce pracující s těmito daty. Naproti tomu objektově orientovaná analýza a návrh chápe systém jako kolekci vzájemně komunikujicích objektů, tj. data a funkce pracující s těmito daty jsou spolu svázány.

Funkční modelování. Funkční (nebo také procesní) modelování ukazuje funkce systému, toky dat mezi systémem a okolím, toky dat mezi funkcemi systému a data ukládaná v systému. Základním modelem je diagram datových toků DFD (Data Flow Diagram). Jednotlivé procesy (funkce) bývají blížeji specifikovány pomocí tzv. minispecifikací, které definují funkcionalitu procesů.

Datové modelování. Datové modelování ukazuje entity aplikační domény zpracovávané systémem a statické vztahy mezi nimi (typicky perzistentní data ukládaná v databázi). Jde o význačný model v datově intezivních aplikacích. Datové modelování má také zásadní význam pro návrh databází. Základním modelem je diagram entit a vztahů ERD (Entity Relationship Diagram).

Datový slovník. Obsahuje specifikace prvků modelů, notace pro specifikaci informačního obsahu prvků DFD a ERD.

Modelování dynamického chování. Pro modelování dynamického chování systémů se používá stavový diagram, který zachycuje stavy, ve kterých se systém (či jeho část) může nacházet, přechody mezi stavy, události vyvolávající přechody a reakce na přechody mezi stavy.

6.2 Data Flow Diagram (DFD)

6.2.1 Základní princip

Diagram datových toků (DFD) je technika používaná při strukturované analýze a návrhu pro specifikaci chování systému. DFD zachycuje funkce, které má systém nabízet (funkcionalitu systému). Z tohoto pohledu se DFD podobá diagramům

případů užití (use case diagrams – UCD) z objektově orientované analýzy. Oproti UCD však DFD zachycuje také *datové sklady* a *toky dat* mezi funkcemi systému a datovými sklady. Z tohoto pohledu je tedy DFD blíže návrhu než UCD. Stejně jako jsou případy užití z UCD doplněny svými detaily pro bližší specifikaci své funkcionality, jsou funkce z DFD doplněny *minispecifikacemi*. Minispecifikace popisují danou funkci, a to většinou ve formě strukturovaného textu.

DFD je hierarchický model, tzn. že každá funkce může být dále modelovaná opět DFD. Většinou se celý systém modeluje tak, že se definuje na nejvyšší úrovni celý systém a jeho okolí, a tento systém je dále rozvíjen na nižších úrovních DFD. Součástí DFD jsou tedy funkce systému (zachycují se jako ovály), datové sklady, aktéři a datové toky mezi funkcemi a datovými sklady (příp. aktéry). Bližší popis DFD si ukážeme na příkladu v kapitole 6.2.2.

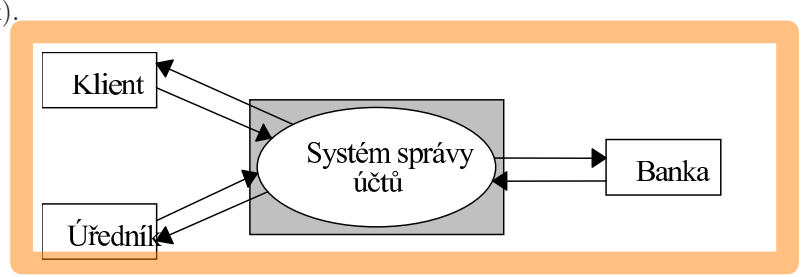
6.2.2 Ukázka funkčního modelování

Uvažujme následující minispecifikaci systému. Provádíme analýzu systému správy účtů banky. Každý účet má jednoznačné číslo, dále je potřeba znát jméno a adresu majitele účtu. Kromě majitele mohou s účtem disponovat i další jím určené osoby. O těch je třeba znát stejné údaje jako o majiteli. Každá z disponujících osob může mít stanoven limit pro výběr z daného účtu. S účty manipuluje úředník banky na základě příkazu osoby oprávněné s účtem disponovat. Na účet lze provádět vklad, z účtu lze provádět výběr a lze převádět částky na jiné účty v téže nebo jiné bance. Musí být k dispozici informace, kdo příkaz adal a který úředník ho provedl. Systém musí poskytovat prostředky pro správu informací o klientech banky, musí umožňovat vytvářet a rušit účty, zadávat příkazy, importovat příkazy pro převody z jiných bank a naopak exportovat příkazy pro převody na účty v jiných bankách. Systém musí být schopen tisknout měsíční výpisy z účtů a řadu dalších tiskových sestav.

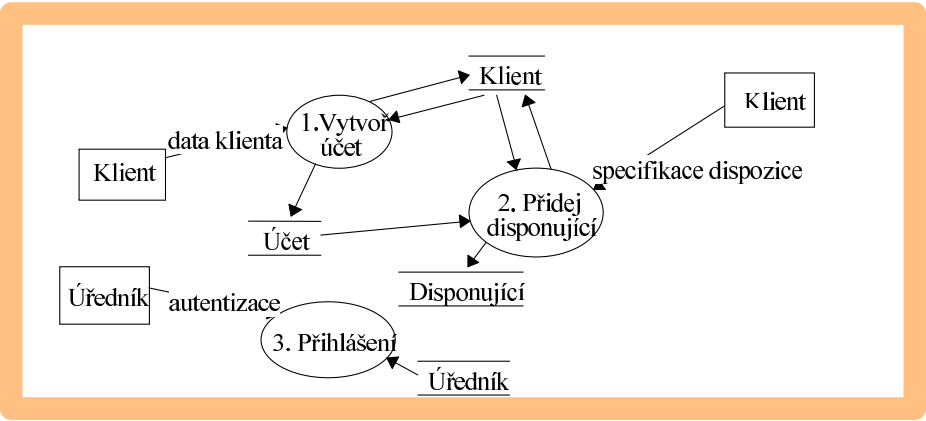
x+y

Prvním krokem je identifikace vlastního systému a jeho okolí (účastníků). Můžeme identifikovat *klienta* banky, *úředníka* a *banku* (kvůli mezibankovním převodům). Vlastní systém, který budeme modelovat, je systém správy účtů (viz následující obrázek).

x+y

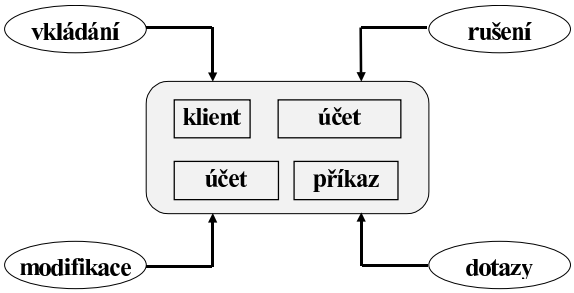


Na další úrovni popíšeme detailněji systém správy účtů, tj. popíšeme jeho základní funkce. Pro naše účely jsme identifikovali tři funkce: *vytvoření účtu*, *přidání disponujícího* a *přihlášení k systému* (viz následující obrázek). Vidíme, že kromě funkcí a aktérů jsou zobrazeny i datové sklady a toky mezi aktéry, funkcemi a sklady. Proberme si blíže funkci *Přidej disponující*. Funkce musí získat nejprve informace o klientovi (autentizace apod.) – viz datový tok mezi funkcí a skladem *Klient*. Na základě informací o klientovi získá informace o jeho účtu (viz datový tok mezi funkcí a skladem *Účet*. Následně na základě informací o účtu a specifikace nového disponenta se vytvoří záznam o novém disponujícím (viz datový tok mezi funkcí a skladem *Disponující*.



6.3 Entity Relationship Diagram

Entity Relationship Diagram (ERD) slouží k modelování dat aplikační domény a jejich vztahů "v klidu". ERD je síťový model popisující návrh uložených dat v systému na vyšší úrovni abstrakce. ERD je zcela odlišný od diagramu datových toků (DFD), který modeluje funkce vykonávané navrhovaným systémem. ERD modeluje data, která potřebujeme v systému uchovávat a vztahy mezi těmito daty.



6.3.1 Základní pojmy datového modelování

Nejprve si vysvětlíme základní pojmy, které se vyskytují v datovém modelování.

- *Entita.* Entita je "věc" reálného světa (věc bývá často označována jako objekt). Každá taková entita musí být rozlišitelná od jiných entit (musí být jedinečná). Příkladem entity může být konkrétní klient banky (každého klienta lze od ostatních odlišit), konkrétní účet v bance (každý účet je jedinečný) apod.

- *Entitní množina.* Entitní množina je množina entit téhož typu, které sdílí tytéž vlastnosti (atributy). V datovém modelování se entity téhož typu klasifikují do entitiní množiny, která je pak zachycena v ERD. V návaznosti na předchozí příklad máme entitní množiny *Klient* a *Účet* definující atributy uvedených entit.

- *Vztah.* Vztah vyjadřuje asociaci mezi entitami, tj. situaci, kdy dvě (případně více) entity spolu logicky souvisejí. Např. klient banky (s číslem klienta K999) vlastní účet (s číslem účtu U100). Mezi těmito dvěma entitami je vztah.

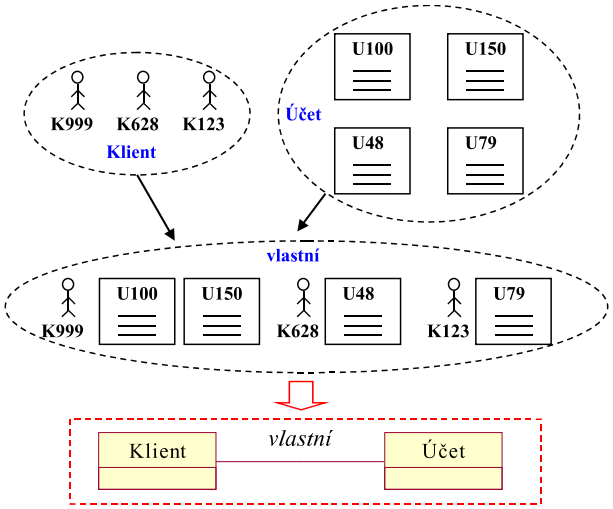
- *Vztahová množina.* Vztahová množina je množina vztahů téhož typu, které sdílí tytéž vlastnosti. V návaznosti na předchozí příklad existují entitní

DEF

Jak vidno, pojem objekt není omezen pouze na objektově orientované přístupy a metody. Objektem lze označit (a také se tak děje) téměř cokoliv. Je proto nutné si uvědomit, že ne vždy, když se mluví o objektu, má se na mysli objektová orientovanost.

Vidíme, že vztah entita – entitní množina je velmi podobný vztahu objekt – třída v objektově orientovaných přístupech.

množiny *Klient* a *Účet*, mezi nimiž existuje vztahová množina vyjadřující vlastnictví účtu.



Obrázek 6.1: Tvorba ER diagramu

Význam uvedených pojmů názorně vysvětluje obrázek 6.1. Na něm je zachyceno 7 entit – 3 klienti banky (K999, K628, K123) a 4 účty (U100, U150, U48, U79). Mezi entitami existují tyto vztahy – klient K999 vlastní účty U100 a U150, klient K628 vlastní účet U48 a klient K123 vlastní účet U79. Na základě těchto faktů můžeme klasifikovat entity do dvou entitních množin – *Klient* a *Účet*, mezi kterými existuje vztah vlastnictví účtu klientem banky (klient *vlastní* účet).

6.3.2 Atribut entity

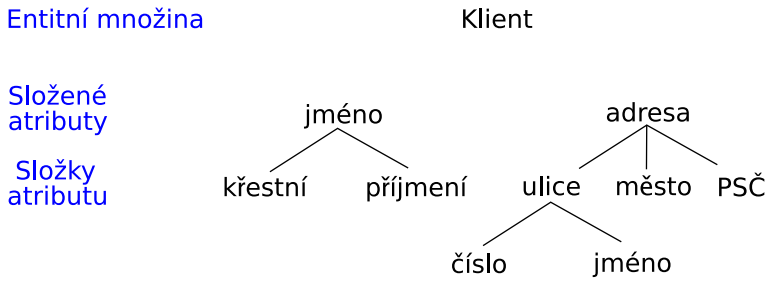
Atribut je vlastnost entity, která nás v kontextu daného problému zajímá. Atribut je definován jako prvek entitní množiny (např. u entitní množiny Klient nás zajímá čísloKlienta, jméno, příjmení, adresa apod.) Atributy můžeme charakterizovat podle různých kritérií:

- jednoduché (simple) a složené (composite) atributy (viz obrázek 6.2)
  - jednoduchý atribut je reprezentován jednoduchým (primitivním) datovým typem (např. integer, string atd.)
  - složený atribut (jak už název napovídá) je složen z jednoduchých, příp. i jiných složených atributů

Jednoduché atributy jsou podmínkou pro tzv. 1. normální formu databázové tabulky.

- jednohodnotové (single-valued) a vícehodnotové (double-valued) atributy
  - vícehodnotový atribut může nabývat více různých hodnot *současně* – např. atribut telefon může mít více čísel
  - lze omezit minimální a maximální počet hodnot, kterých může atribut současně nabývat
- prázdné atributy
  - mohou nabývat speciální hodnoty NULL

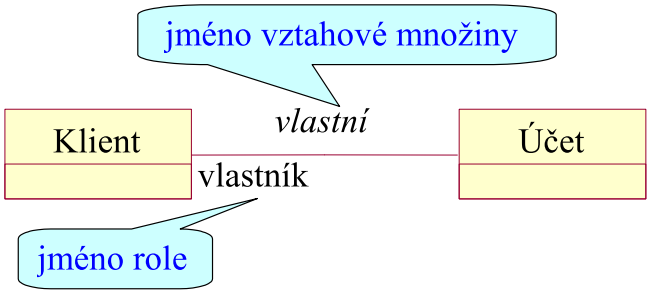
  - vhodné pro zachycení situace, kdy víme, že existuje reálná hodnota atributu, ale v jistých okamžicích může být neznámá (prázdný atribut tak zastupuje chybějící hodnotu)
  - vhodné pro zachycení situace, kdy nevíme, zda existuje reálná hodnota atributu (prázdný atribut tak zastupuje neznámou hodnotu)
- odvozené atributy
  - atribut, který není definován entitní množinou přímo – jeho hodnotu lze odvodit od jiných atributů nebo vztažených entit
  - např. věk lze odvodit z atributu datumNarození a informace o aktuálním datu



Obrázek 6.2: Typy atributů

6.3.3 Vztah mezi entitami

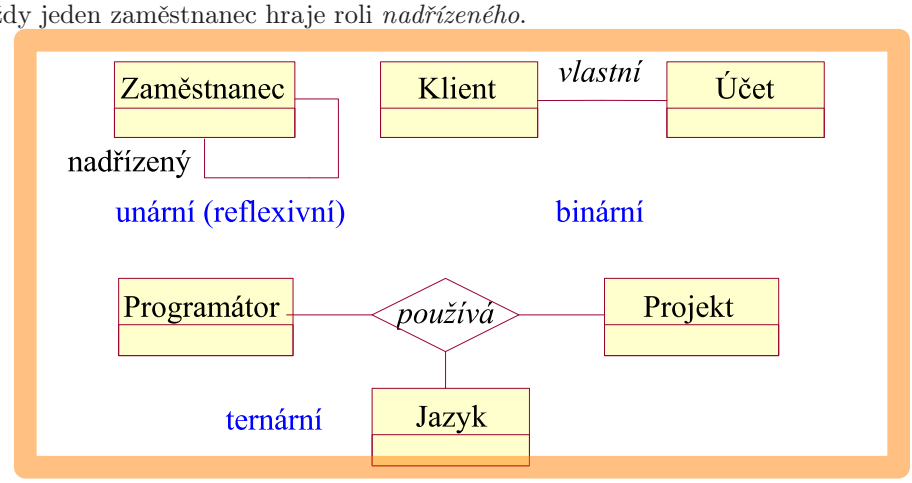
Vztah (vztahová množina) je vedle entity (entitní množiny) dalším důležitým prvkem zachycovaným v ERD. Vztahová množina je graficky zachycena asociací (spojem) mezi příslušnými entitami. Každá vztahová množina definuje své jméno, členství entity v daném vztahu, kardinalitu a název role, kterou příslušná entita hraje v určitém vztahu (viz obrázek 6.3). Obrázek 6.3 zachycuje vztahovou množinu *vlastní* mezi entitními množinamy *Klient* a *Účet*, která říká, že každá entita typu *Klient* je ve vztahu s nějakou entitou (příp. více entitami) typu *Účet*, přičemž entita typu *Klient* hraje v tomto vztahu roli *vlastníka*. Název vztahové množiny i jméno role vyjadřují sémantiku (význam) vztahu.



Obrázek 6.3: Parametry vztahů

Stupeň vztahu

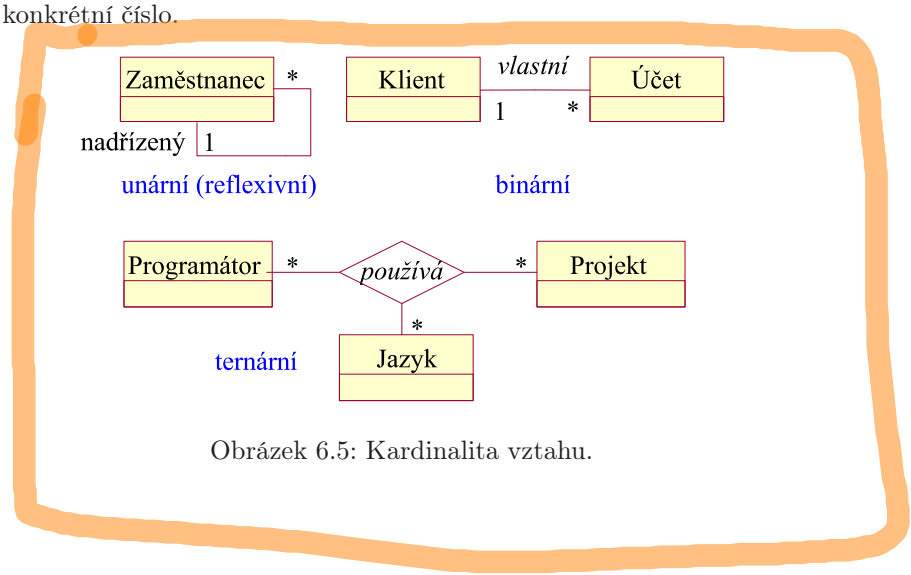
Stupeň vztahu vyjadřuje kolik entitních množit je zapojeno do jedné vztahové množiny (viz obrázek 6.4). Nejběžněji se používají vztahy unární (reflexivní) a binární. Ternární vztahy (příp. i vyššího stupně) se vyskytují zřídka – komplikují návrh a většinou se dají převést na vztahy binární. Unární vztah uvedený na obrázku 6.4 vyjadřuje, že zaměstnanci mohou být navzájem ve vztahu, ve kterém vždy jeden zaměstnanec hraje roli *nadřízeného*.



Obrázek 6.4: Stupeň vztahu

Kardinalita vztahu

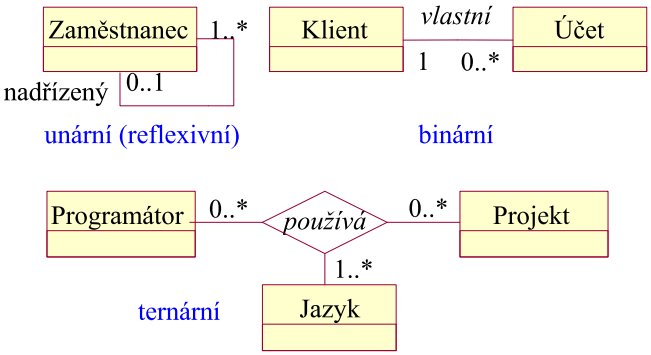
Kardinalita (*cardinality*) je maximální počet vztahů daného typu (vztahové množiny), ve kterých může participovat jedna entita. Na obrázku 6.5 jsou naznačeny různé kardinality pro nám již známé ukázkové vztahové množiny. Takže např. entita typu *Klient* vlastní (může vlastnit) více účtů (kardinalita je naznačena znakem $*$ na příslušné straně vztahu). Naproti tomu entita typu *Účet* má nejvýše jednoho vlastníka (kardinalita je naznačena znakem 1 na příslušné straně vztahu). Typické hodnoty, které se u kardinality uvádějí jsou: 1 a M (příp. $*$), pokud je třeba, lze uvést konkrétní číslo.



Obrázek 6.5: Kardinalita vztahu.

Členství vztahu

Členství (*membership*) nebo také účast (*participation*) je minimální počet vztahů daného typu (vztahové množiny), ve kterých musí participovat jedna entita. Typické hodnoty, které se u členství jsou: 1 (povinné) a 0 (volitelné). Na obrázku 6.6 jsou naznačena různá členství pro nám již známé ukázkové vztahové množiny. Takže např. entita typu *Klient* nemusí vlastnit žádný účtů (členství je naznačeno znakem 0 na příslušné straně vztahu). Naproti tomu entita typu *Účet* má minimálně jednoho vlastníka (členství je naznačeno znakem 1 na příslušné straně vztahu). Ve skutečnosti, pokud na uvedený obrázek pohlížíme jako na úplný ER diagram (tj. včetně členství a kardinality), pak klient nemusí vlastnit žádný účet nebo může vlastnit více účtů a účet musí mít právě jednoho vlastníka.

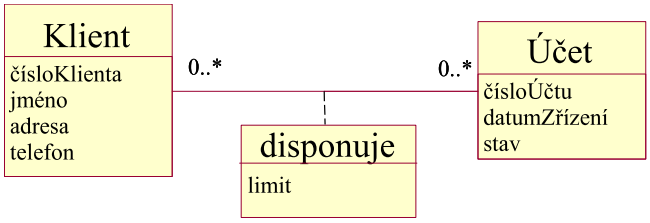


Obrázek 6.6: Členství / účast

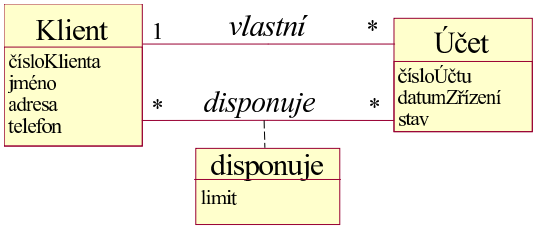
Atributy vztahu

Vztahu můžeme přiřadit atributy. Atributy vztahu vytváříme v okamžiku, kdy nelze (nebo z jistých důvodů nechceme) tyto atributy přiřadit některé vázané entitě. V tomto případě povýšíme vztahovou množinu na entitní množinu, které pak příslušné atributy přiřadíme (viz obrázek 6.7). Na obrázku 6.7 existuje vztahová množina *disponuje* mezi entitami typu *Klient* a *Účet*, která je typu M:N (tzn., že klient může disponovat více účty a učet může mít více disponentů). U každého takového vztahu (*disponuje*) chceme uchovávat limit, se kterým může disponent manipulovat na daném jednom účtu. Klient může disponovat více účty a účet může mít více disponentů, nelze tedy tento atribut přiřadit ani entitní množině *Klient* ani entitiní množině *Účet*, protože by nebylo jasné, k jakému vztahu se tento atribut váže. Musíme proto vztahovou množinu povýšit na entitní množinu a atribut definovat zde. V tomto případě pak každý vztah koresponduje s jednou entitou typu *disponuje*.

Více vztahů mezi entitami. Na následujícím příkladu si ukážeme, že mezi stejnými entitami může existovat více různých vztahů, které mohou vyjadřovat různou sémantiku. Mějme opět entitní množiny *Klient* a *Účet*, mezi jejímiž entitami může nalézt tyto dva vztahy: klient *může vlastnit* více účtů (každý účet má právě jednoho vlastníka) a klient může disponovat více účty (účet může mít více disponentů). Rozlišujeme tedy vztah vyjadřující možnost omezené manipulace s účtem (převody, výběry atd.) a vztah vyjadřující vlastnictví účtu (vlastník účtu pak má plná práva manipulace s účtem). V našem ER diagramu je toto zaznačeno dvěma vztahovými množinami – *vlastní* a *disponuje*.



Obrázek 6.7: Atributy vztahu



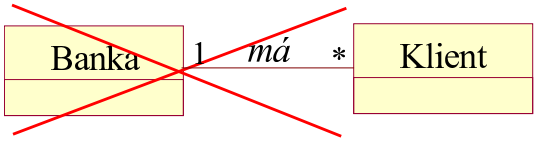
6.3.4 Tvorba ERD

V této části si shrneme nejpodstatnější aspekty spojené s tvorbou ERD a podmínky, které je vhodné dodržovat.

Názvy. Názvy (jména) musí být srozumitelné a musí vyjadřovat význam entitiních a vztahových množin. Pro entitiní množiny se používají podstatná jména v jednotném čísel, pro vztahové množiny většinou slovesa či předložky. V případě, že je jméno vztahové množiny jasné ze jmen entitních množin, pak není nutné toto jméno uvádět. Např. *Klient—vlastní—Účet*, nebo *Výdaj—za—Projekt*.

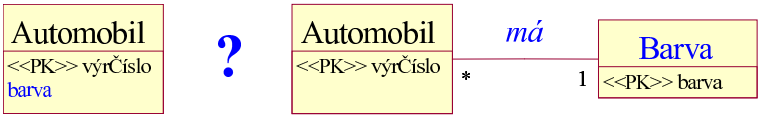
Identifikátor (primární klíč). Každá entita musí být jednoznačně identifikovatelná – ve strukturovaném návrhu k tomu slouží *primární klíče* definované u entitních množin. Hodnota každého primárního klíče musí být v rámci jedné entitní množiny unikátní a minimální. Jednoznačným identifikátorem je jednoduchý nebo složený atribut. Je dobré si uvědomit, že unikátnost hodnoty je vyžadována jen v rámci vyvíjeného systému, nikoliv celého vesmíru!

Reprezentace systému v ERD. Celkový systém by *neměl být zahrnut* do ERD (viz obrázek). Pokud modelujeme banku, nemá žadný význam do ERD zahrnovat entitní množinu *Banka*, protože *to celé, co modelujeme, je banka!* Modelování entitní množiny *Banka* by mělo význam v okamžiku, kdy potřebujeme mít reprezentaci *jiných* bank, např. z důvodu mezibankovních transakcí apod.

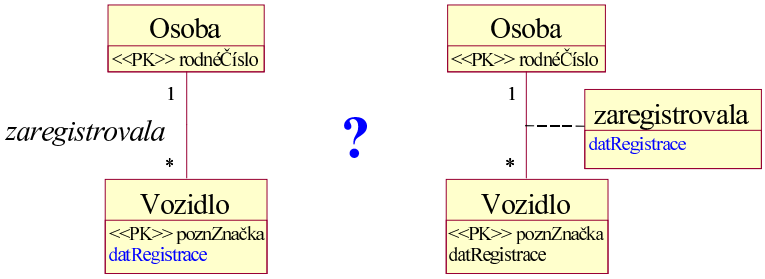


Použít entitní množinu nebo atribut? Někdy můžeme řešit otázku, zda některou věc modelovat jako atribut entity, nebo zda raději vytvořit novou entitní množinu. Např. zda barva automobilu bude atribut automobilu, nebo

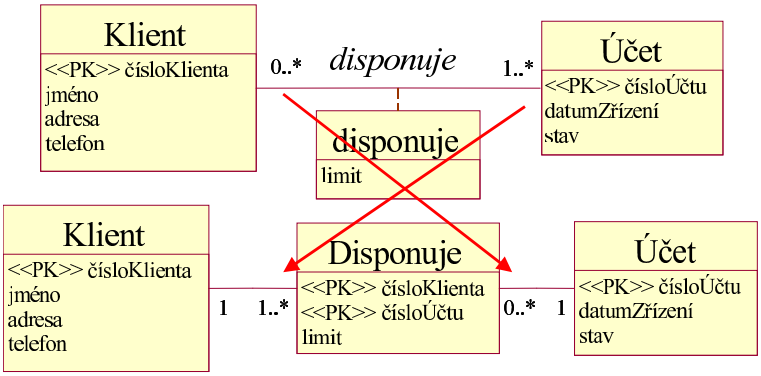
samostatná entitní množina (viz obrázek). Pokud s tímto atributem souvisejí další informace (např. odlišnost technologického postupu, cena, atp.), pak je lepší tuto vlastnost modelovat jako zvláštní entitní množinu. Existuje pravidlo: Je-li hodnota atributu důležitá, i když neexistuje žádná entita s touto hodnotou jako vlastností, pak bychom ji měli modelovat jako entitu.



Modelování atributů ve vztahu 1:N. Při modelování atributů vztahu se rozhodujeme, zda povýšit vztahovou množinu na entitní množinu nebo atribut přiřadit některé entitě ve vztahu. Např. řešíme otázku, jak uchovat informaci o datumu registrace vozidla (viz obrázek). Pokud budeme tuto informaci modelovat jako atribut entity *Vozidlo*, ztrácíme možnost sledování historie registrací vozidla (vždy se uchová informace pouze o poslední registraci). Pokud chceme uchovat i historii registrací, musíme uvedenou vztahovou množinu povýšit na entitní množinu a atribut datum registrace modelovat jako atribut této entitní množiny (viz Atributy vztahu na straně 68).

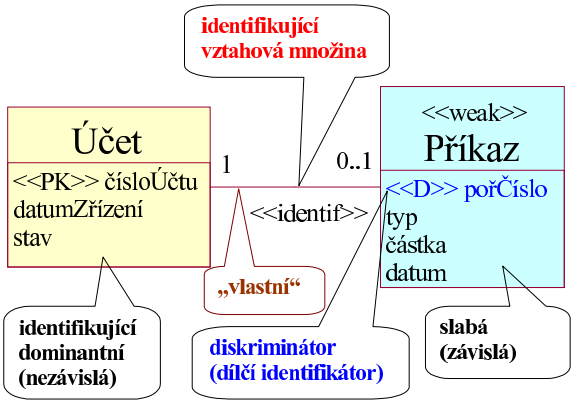


Náhrada vztahů M:N vazební entitní množinou. Vztahy typu M:N se těžko převádějí na schéma relační databáze, proto se transformují na vztahy typu 1:N pomocí vazební entitní množiny. Jedná se v podstatě o realizaci vztahové množiny povýšené na entitní množinu. Princip je znázorněn na následujícím obrázku. Původní vztahová množina je nahrazena dvěma vztahovými množinami, a to mezi první a vazební entitní množinou a mezi vazební a druhou entitní množinou, přičemž členství a kardinalita se přesunou podle naznačených šipek. U původních entitních množit se členství a kardinalita nahradí za hodnotu 1. Primární klíč vazební entitní množiny je tvořen primárními klíči první a druhé entitní množiny (*čísloKlienta* a *čísloÚčtu*). Každé původní vazbě pak odpovídá jedna vazební entita s novými vazbami na původní entity.



6.3.5 Slabé (weak) entitní množiny

Až doposud jsme se zabývali tzv. *silnými (strong)* entitními množinami. Silná entitní množina se vyznačuje tím, že její identifikátor (primární klíč) je tvořen vlastními atributy. Vedle silných entitních množin mohou existovat ještě *slabé (weak)* entitní množiny, jejichž identifikátor je tvořen i atributy jiných silných entitních množin. Slabá entitní množina je tedy závislá na nějaké jiné (silné) entitní množině (viz obrázek 6.8).

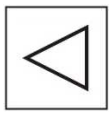


Obrázek 6.8: Slabá entitní množina

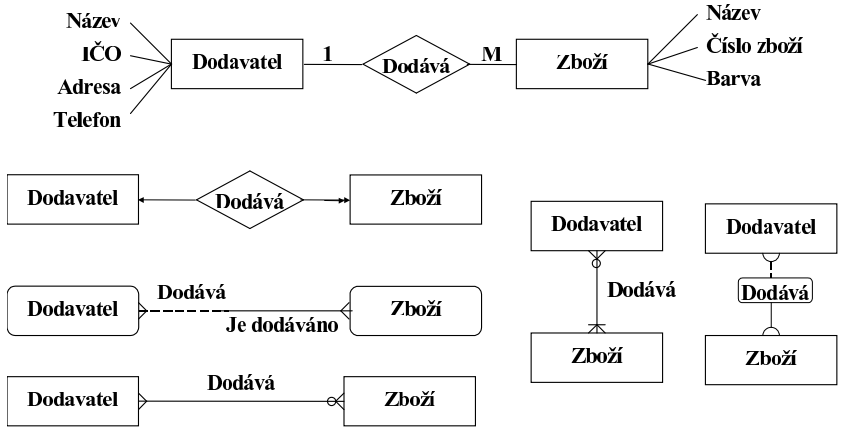
Identifikátor slabé entitní množiny. Identifikátor je tvořen dominantním identifikátorem nezávislé entitní množiny + diskriminátorem slabé (závislé) entitní množiny. Slabá množina existenčně závisí na silné (identifikující, dominantní) množině. Pokud zanikne silná entita, zanikají spolu s ní i na ní závislé slabé entity. Na obrázku 6.8 je naznačen příklad, kde příkaz je chápán jako slabá entita vázaná na nějaký účet. Když zanikne účet (jeho reprezentace jako entity), zaniknou i všechny příkazy svázané s tímto účtem. Příkaz je identifikován jako *čísloÚčtu* + *pořadovéČíslo* příkazu.

Slabá nebo silná entitní množina? Entitu modelujeme jako slabou, když tato entita kompletně zmizí při odstranění odpovídající identifikující entity. Cokoliv s atributem, který je jednoznačný, by nemělo být modelováno jako slabá entitní množina. Opět existuje pravidlo: Jsme-li na pochybách, modelujeme jako silnou entitní množinu.

6.3.6 Alternativní notace ERD



Kromě notace ER diagramů, kterou jsme si uváděli v našem textu, existují i notace jiné. Na obrázku 6.9 jsou tyto alternativy stručně naznačeny.



Obrázek 6.9: Alternativní notace ERD

Kontrolní otázky

6.1 Jaké jsou základní modely strukturované analýzy?

6.2 Charakterizujte DFD.

6.3 Charakterizujte ERD.

6.4 Definujte pojmy entita a entitiní množina

6.5 Definujte pojmy vztah a vztahová množina.

6.6 Vysvětlete pojmy stupeň, kardinalita a členství vztahu.

6.7 Co je primární klíč a co pro něj musí platit?

Kapitola 7

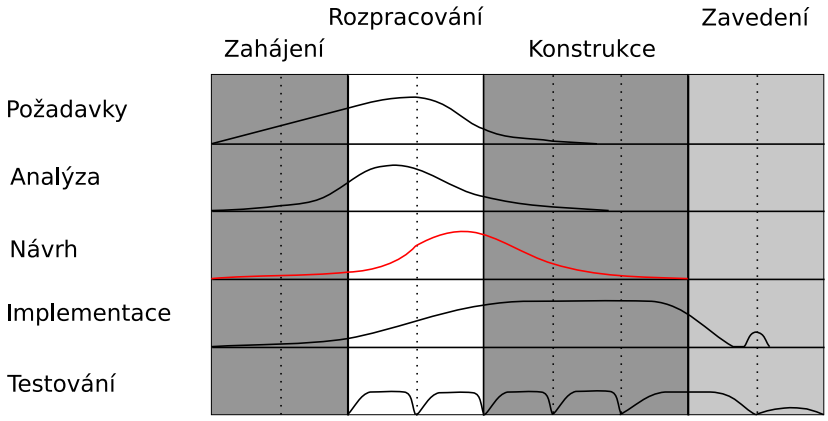
Návrh

0:50

7.1 Prostředky pro návrh v RUP

Návrh je po specifikaci a analýze požadavků další etapou ve vývoji softwaru. Dobrý návrh je podmínkou úspěšné tvorby softwaru a minimalizuje náklady spojené s implementací a posléze i údržbou softwarového produktu. V této kapitole se zaměříme na prostředky podporující návrh v metodice RUP. Výstupem etapy návrhu jsou většinou tyto diagramy:

- doplněné (upřesněné) analytické diagramy, zejména
  - diagramy interakce
- diagramy upřesňující vybrané analytické diagramy, zejména
  - návrhové třídy (upřesňují *analytické třídy*)
  - návrhové podsystémy (vycházejí z *analytických balíčků*)
  - stavové diagramy
  - rozhraní

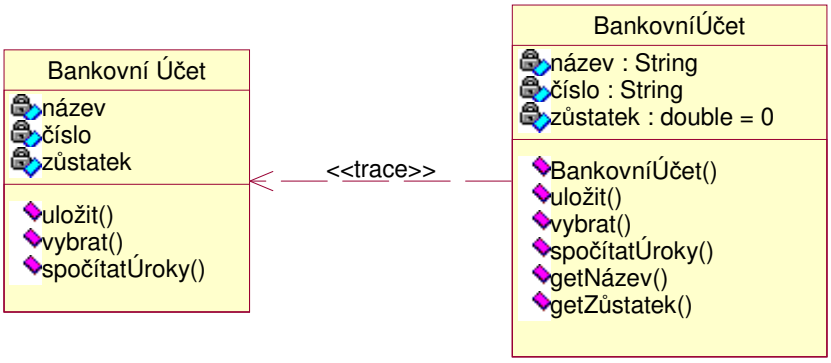


Obrázek 7.1: Návrh v kontextu RUP

7.2 Návrhové třídy

Návrhové třídy vycházejí z analytických tříd, kterým doplňují implementačně důležité vlastnosti (proměnné, metody) tak, že návrhové třídy lze přímo implementovat. V diagramu návrhových tříd se také uplatňují komponenty (třídy) z doménového řešení, tj. z např. knihovny použitého programovacího jazyka, vrstva

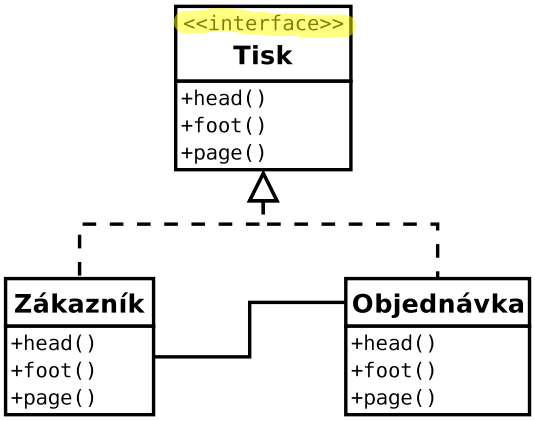
aplikačního serveru, se kterým aplikace (její třídy) komunikuje, knihovny grafického uživatelského rozhraní apod.



Obrázek 7.2: Návrhové třídy

7.3 Rozhraní

Rozhraní odděluje specifikaci funkčnosti od její implementace. Rozhraní pouze deklaruje *metody* potřebné pro splnění specifikované odpovědnosti objektu. Používání rozhraní umožňuje flexibilnění modelování (implementaci) a omezuje počet vazeb mezi třídami či objekty.



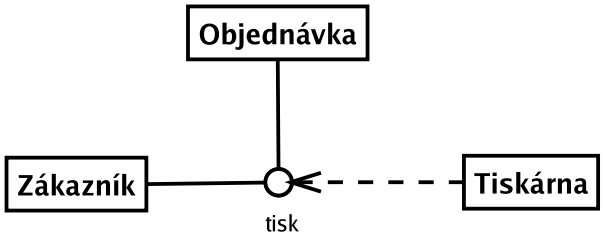
Obrázek 7.3: Rozhraní *Tisk*

Příklad použití rozhraní. Na obrázku 7.3 můžeme vidět rozhraní pro tisk, které specifikuje metody *head()* pro získání záhlaví stránky, *foot()* pro získání zápatí stránky a *page()* pro získání obsahu stránky. Toto rozhraní implementují dvě třídy – třída *Zákazník* a třída *Objednávka*. Na obrázku 7.4 je znázorněna alternativní notace rozhraní a implementujících tříd (v tomto případě se neznázorňují metody nabízené rozhraním). Na tomto diagramu je také zobrazena třída *Tiskárna*, která *používá* rozhraní *Tisk*. Tím, že jsme použili rozhraní místo přímých vazeb na příslušné třídy, jsme zjednodušili návrh v tom smyslu, že není nutné modifikovat třídu

Tiskárna při vkládání nových tříd, které mohou být také tištěny. Tyto nové třídy prostě implementují rozhraní *Tisk*, takže třída *Tiskárna* umí používat instance těchto tříd. Požití rozhraní si můžeme znázornit i prostřednictvím zdrojového kódu (se syntaxí programovacího jazyka Java). Třída tiskárna obsahuje metodu *print*, která může vypadat nějak takto:

```
public void print(Tisk document) {
    printHead(document.head());
    ...
}
```

Jako argument má objekt, který se bude tisknout. Tento objekt je typu *Tisk*, tedy argumentem této metody může být libovolný objekt, jehož třída implementuje rozhraní *Tisk*.



Obrázek 7.4: Rozhraní *Tisk* – alternativní notace

Příklad použití rozhraní. Obrázek 7.5 demonstruje další využití rozhraní při deklaraci podsystémů. Podsystém deklaruje rozhraní, která jediná jsou přístupná svému okolí. Toto okolí může k podsystému přistupovat pouze skrz zveřejněné rozhraní, což umožňuje lepší koordinaci při implementaci podsystémů (zjednodušuje kontrolu správnosti).

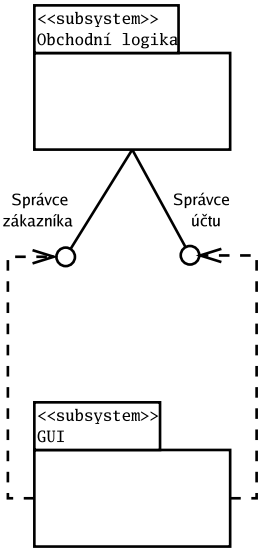
7.4 Upřesňování analytických modelů

Během návrhu dochází k upřesňování analytických modelů, a to zejména analytických tříd (viz kapitola 7.2) a diagramů interakce.

Diagramy interakce jsou doplňovány o nové třídy, které vyplynuly z transformace analytického modelu tříd na návrhový model tříd. Dále jsou doplněny o implementačně specifiké prvky, např. jaký konstruktor se použije pro inicializaci objektu, lze si uchovávat objekty a těm následně zasílat zprávy apod. Ukázka upřesněného sekvenčního modelu je na obrázku 7.6 – tento diagram vychází z analytického diagramu na obrázku 5.3.

7.5 Stavové diagramy

Stavové diagramy (statecharts) představují významný prvek v metodologiích, které se opírají o modelování a následné testování těchto modelů. V jednoduchém přiblížení si můžeme statové diagramy představit jako síť stavů, událostí a přechodů mezi stavy. Jde tedy o specifický druh stavového automatu. V rámci seznamování s jazykem UML jsme již poznali jeden typ stavového automatu, a to diagram aktivit. Diagram aktivit je především určen pro modelování manažerských



Obrázek 7.5: Rozhraní a podsystémy

procesů a postihuje chování bez ohledu na účastnící se objekty (tj. nespecifikuje odpovědnosti objektů a tedy ani jejich konkrétní chování). Naproti tomu stavový diagram modeluje životní cyklus jednoho reaktivního objektu.

Reaktivní objekt je takový objekt, který reaguje na vnější události, tzn. že je schopen na vnější podnět a na základě svého stavu provést akci. Výsledkem akce může být změna stavu reaktivního objektu či vygenerování dalších událostí. Životní cyklus reaktivního objektu je modelován jako řada stavů, přechodů a událostí. Chování objektu je tedy důsledkem předchozího chování (následný stav závisí na aktuálním stavu).

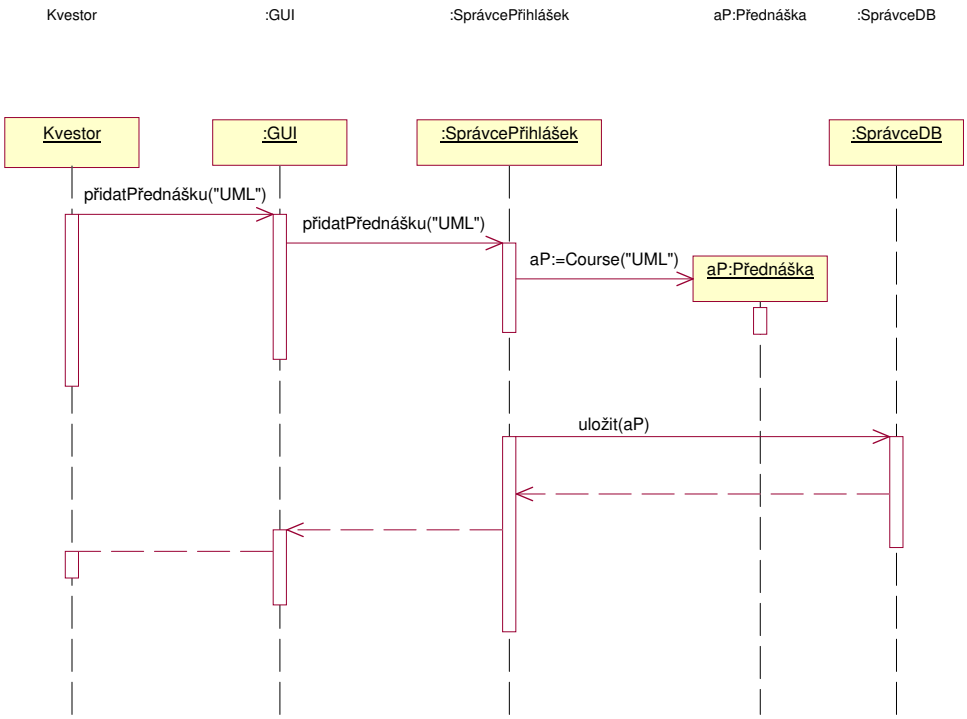
Reaktivním objektem se v tomto kontextu může myslet *třída* (resp. popis chování instancí třídy), *případ užití* nebo *systém* (*podsystém*). Nejčastější použití stavových diagramů je však pro modelování chování instancí tříd (tj. jeden stavový diagram modeluje chování jedné třídy).

Ukázka prvků stavového diagramu. Obrázek 7.7 zobrazuje základní prvky stavového diagramu: *stav*, *přechody mezi stavy* a *události*. Události mohou být součástí stavu nebo přechodu. S událostí je vždy spojená akce – v případě stavu jsou definované tři základní události:

x+y

- *vstup (entry)* definuje akce provedené při vstupu do stavu
- *výstup (exit)* definuje akce provedené při opouštění stavu
- *aktivita (do)* definuje akce prováděné během setrvání ve stavu

Kromě těchto událostí může pochopitelně objekt v daném stavu reagovat i na další (uživatelské) události. V případě přechodu podmiňuje událost provedení tohot přechodu. Kromě výskytu události může být přechod také podmíněn stavovou podmínkou (např. že proměnná objektu má nějakou hodnotu). Tyto dvě podmínky přechodu se dají navzájem kombinovat a může být uvedena pouze jedna z nich. Kromě přechodu do nového stavu lze definovat i akci, která se provede během přechodu.



Obrázek 7.6: Upřesňování analytických modelů

Ukázka stavového diagramu. Obrázek 7.8 popisuje chování objektu *Účet* pomocí stavového diagramu. Stejně jako diagram aktivit má i stavový diagram definovaný počáteční stav. Účet může být ve dvou základních stavech – solventní (zůstatek $>= 0$) nebo přečerpaný (zůstatek < 0). V obou stavech lze ukládat a vybírat částky peněz. V případě změny stavové podmínky (viz přechody a proměnná objektu *zůstatek*) dojde k přechodu do nového stavu.

x+y

Ukázka stavového diagramu s událostí WHEN. Obrázek 7.9 vychází z předchozího příkladu, který jsme obohatili o debetní limit – účet nemůže být přečerpaný více než povolená částka. Pokud bude proměnná *zůstatek* menší než *debetníLimit* (viz událost *WHEN*), provede se akce *uvědomitÚředníka*.

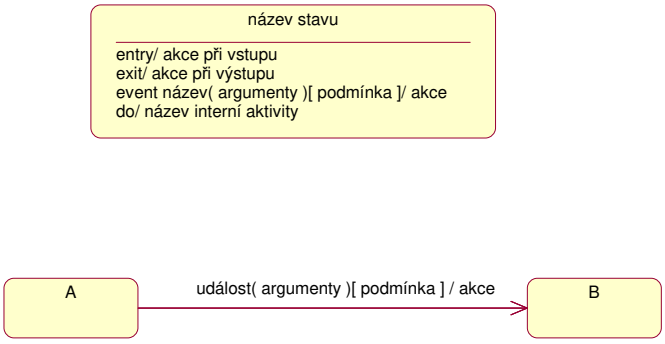
x+y

Ukázka sekvenčního složeného stavu s událostí AFTER. Stavy lze také hierarchicky vnořovat do sebe – složený stav pak obsahuje další stavy a přechody mezi nimi. Obrázek 7.10 popisuje chování při vytáčení čísla. Složený stav *VytáčeníČísla* reprezentuje chování při bezchybném pokusu o vytočení čísla. Tento stav pak lze při různých chybát či uplynutí časového intervalu opustit. Např. pokud ve stavu *ČekáníNaOznamovacíTón* nepřijde po 20 sekundách *oznamovací tón*, přejde se do stavu *NeníPřipojeno*. Nebo pokudn kdykoliv ve složeném stavu *VytáčeníČísla* přijde událost *storno*, přejde se do stavu *NeníPřipojeno*.

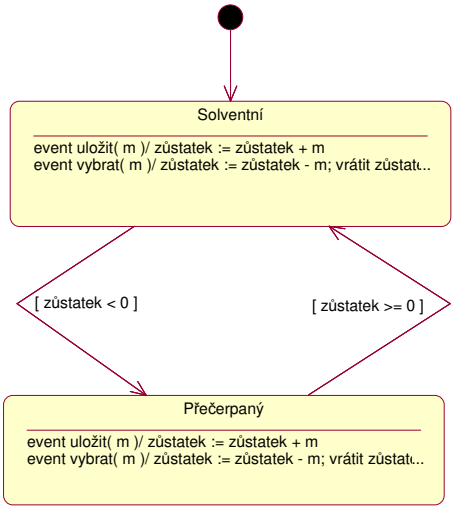
x+y

Kontrolní otázky

7.1 Co je to návrhová třída a diagram návrhových tříd?



Obrázek 7.7: Prvky stavového diagramu

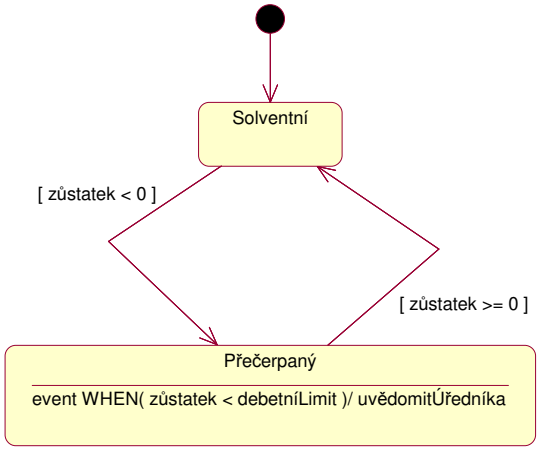


Obrázek 7.8: Stavový diagram – událost volání.

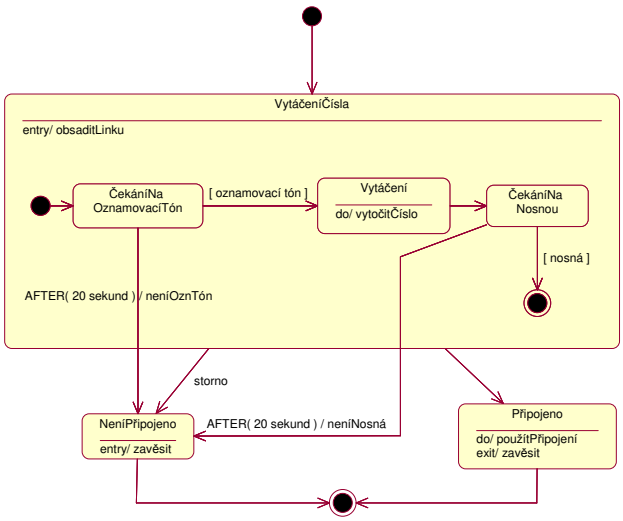
7.2 Co je to rozhraní, čím se vyznačuje?

7.3 Čím se vyznačuje etapa návrhu s využitím prostředků jazyka UML?

7.4 Čím se vyznačují stavové diagramy (statecharts) jazyka UML?



Obrázek 7.9: Stavový diagram – událost změny (WHEN)



Obrázek 7.10: Stavový diagram – sekvenční složený stav a událost změny (AFTER)

Kapitola 8

Implementace a testování

1:20

V této kapitole se budeme věnovat implementaci a testování. Implementací softwaru se zabývá celá řada dalších předmětů a proto se zde omezíme jenom na několik poznámek k výběru programovacího jazyka a ke strategiím implementace.

O něco větší pozornost zde bude věnována testování, ale kvůli šířce této oblasti nelze v prostoru, který máme k dispozici, popsat více než terminologii a obecné principy testování. Věříme však, že základní orientace v této oblasti vám pomůže alespoň v nalezení relevantních zdrojů informací.

Stejně tak se domníváme, že se znalostí principů softwarového inženýrství zvládnete i nasazení a údržbu softwaru, přestože se této etapě téměř nevěnujeme. Nástroje a techniky, které se používají při provozu softwaru, se jednak značně liší podle typu softwaru a chyby při údržbě softwaru nemají většinou takový dopad na úspěch celého projektu, jako například chyby při specifikaci požadavků.

8.1 Implementace softwaru

Implementace softwaru je transformace návrhu jednotlivých modulů a jejich vzájemných vazeb do programové realizace.

Podíl implementace na celkovém objemu prací v životním cyklu softwaru se postupně snižuje. Přispělo a přispívá k tomu řada faktorů, například zavedení programovacích jazyků s vyšší úrovní abstrakce, vizuální programování, využívání integrovaných vývojových prostředí (tzv. IDE – Interactive Development Environment), nástroje pro generování (kostry) aplikací, prostředky pro spolupráci aplikací (tzv. midleware), objektový přístup či zvýšení znovupoužitelnosti pomocí komponent.

8.1.1 Výběr programovacího jazyka

Při výběru vhodného programovacího jazyka hrají roli následující kritéria:

- zkušenosti programátorů s daným jazykem,
- vhodnost jazyka pro příslušnou aplikaci,
- dostupnost podpůrných prostředků pro daný jazyk,
- rozšířenost jazyka,
- možnosti přenositelnosti,
- použitelnost na vybraném hardwaru,
- rozsah dostupných knihoven (zvyšuje možnosti znovupoužití),
- cena vývojového prostředí,

- možnost dodržení standardů pro kontrolu jakosti,
- strategie organizace,
- požadavky zákazníka.

Výběr programovacího jazyka také značně závisí na velikosti softwarového projektu. Pro malý projekt jsou důležité vlastnosti jazyka samotného — jazyk by měl být jednoduchý a srozumitelný, zdrojové texty by měly být čitelné a snadno udržovatelné. U velkých projektů se kromě vlastností jazyka také zajímáme o podporu práce v týmech — tedy o prostředky pro abstrakci (procedury, funkce, generické typy údajů), o mechanizmy pro plánování práce a o schopnosti vývojového prostředí (podpora týmové práce, efektivní kompilace, správa verzí či podpora integrace).

8.1.2 Strategie implementace

Jak jsme si řekli výše, implementace spotřebuje jen malou část úsilí potřebného pro vývoj softwaru a tento podíl navíc dlouhodobě klesá, zatímco nejvíce zdrojů se spotřebuje při provozu a údržbě. **Programy se proto nevytvářejí tak, aby se lehce psaly, ale aby se lehce četly a modifikovaly!**

Při implementaci se dá postupovat dvěma základními způsoby. Při implementaci **zdola-nahoru** se nejdříve implementují části na nejnižších úrovních a ty se pak postupně spojují do větších celků. Základní výhodou tohoto přístupu je, že odladěné moduly nižších úrovní lze přímo použít při vytváření modulů vyšší úrovně. Mezi nevýhody pak patří to, že se chyby v logice projeví až v etapě integračního testování a že je systém možné předvádět až po jeho úplném dokončení. Je také potřeba vytvářet speciální moduly pro generování údajů pro otestování modulů na nižších úrovních.

Při implementaci **shora-dolů** se nejdříve vytvoří moduly na nejvyšší úrovni a pak se postupně dokončují části na nižších úrovních. Výhodou tohoto přístupu je, že systém lze brzy demonstrovat a že nejzávažnější chyby jsou odhaleny včas. Nevýhodou je, že se logika systému ověřuje několikrát a že jsou pro testování potřeba speciální moduly, které simulují práci podsystémů. Tento přístup se nedá použít, pokud je požadována implementace některých modulů nejnižší úrovně (např. výstupních sestav) na začátku. Testování jednotlivých modulů při přístupu zdola-nahoru je jednodušší než testování logiky celého systému.

V praxi se používá kombinace obou přístupů.

8.2 Validace a verifikace programu

V průběhu vytváření softwaru je nezbytné ověřovat, že vytváříme správný výrobek (validace = ověření, že vyvíjený software splňuje potřeby uživatele) a že tento výrobek vytváříme správně (verifikace = ověření, zda software vyhovuje specifikaci).

Ověřujeme různé vlastnosti softwaru, například jeho správnost, spolehlivost, efektivnost či bezpečnost. Chtěli bychom však upozornit čtenáře na úskalí ověřování těchto vlastností. Vezměme si takovou správnost. Je přirozené očekávat, že správnost je dostatečným a současně nevyhnutelným předpokladem úspěšného softwaru. Ale není tomu tak, protože správnost (tj. míra, do jaké software vyhovuje specifikaci) je jenom jednou – byť velice důležitou – vlastností softwaru.

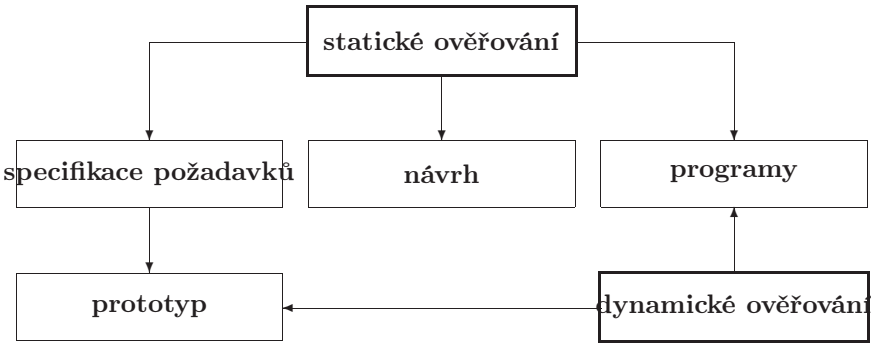
Představme si, že zadavatel chce program na jednoduché výpočty (simulace kalkulačky) a při objednávce takové drobnosti opomene stanovit termín dodání a

maximální cenu. Jestliže se pak dodavatel rozhodne vytvořit *dokonalý* software, implementuje do kalkulačky všechny možné i nemožné funkce a po pěti letech vývoje dodá zákazníkovi vynikající vědeckou kalkulačku za dvacet milionů korun, zákazník nadšen rozhodně nebude.

Naopak ne vždy je stoprocentní správnost softwaru nezbytná. Představme si software na výpočet mezd, při jehož implementaci si programátoři zjednodušili práci tím, že u jednoho zaměstnance připustí maximálně deset dětí, na které může uplatnit daňový odpočet. Jenže v podniku s dvaceti tisíci zaměstnanců, který si tento software objednal, pracuje jeden zaměstnanec, který má dětí jedenáct. Software sice nesplňuje požadavky, ale je v zásadě použitelný. Vedoucí mzdové účtárny každý měsíc spočítá ručně výplatu toho jednoho pracovníka, zatímco software spočítá výplatu všem ostatním. Tato chyba softwaru stojí podnik deset minut práce mzdové účetní měsíčně, zatímco její oprava by mohla stát i desítky tisíc korun. Z ekonomického hlediska se tedy nevyplatí tuto chybu opravovat. Stoprocentní správnost softwaru je navíc prakticky nedosažitelná a uživatelé jsou na chyby v softwaru celkem zvyklí.

8.2.1 Typy ověřování

Rozlišujeme dva typy ověřování – statické a dynamické. Statické ověřování nevyžaduje běh programu a lze ho proto provádět v libovolné etapě vývoje softwaru. Dynamické ověřování neboli testování odvozuje vlastnosti softwaru na základě výsledků běhu programu nebo prototypu s vybranými vstupy. Toto rozdělení ilustruje obrázek 8.1.



Obrázek 8.1: Statické a dynamické ověřování

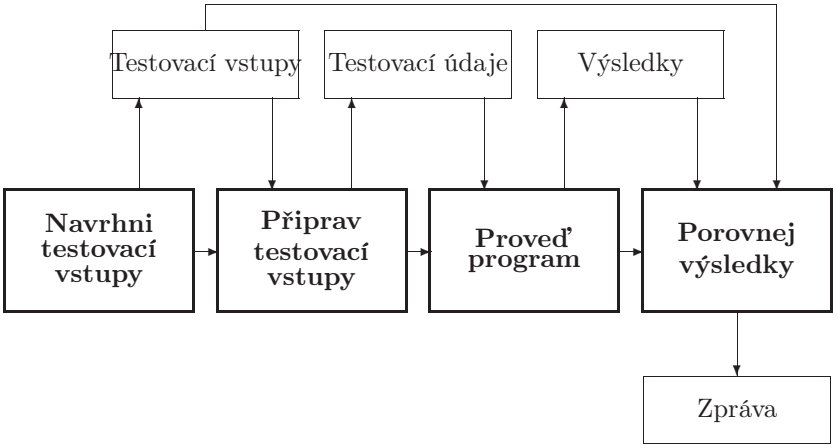
Při statickém ověřování se procházejí různé dokumenty vytvořené během vývoje softwaru (a to včetně zdrojových textů programů). Prohlídka dokumentů může mít různou úroveň formálnosti – od ad hoc prohlídky, přes párové programování až po formální inspekci. Efektivnost prohlídky se zlepšuje použitím seznamů otázek a vytvořením statistik o typech nalezených chyb a jejich počtu.

8.3 Testování

Cílem testování je odhalit chyby během vývoje softwaru. Test, který neodhalí nesprávné chování systému, je neúspěšný.

Na obrázku 8.2 vidíme typický proces testování systému. Nejdříve jsou navrženy testovací vstupy, což je v podstatě množina dvojic *vstupní data/očekávaná výs-*

tupní data. Vstupní data (testovací údaje) jsou pak zadány programu na vstup, čímž získáme výsledky. Ty pak porovnáme s očekávanými výstupními daty a zjistíme, zda jsme odhalili chybu v programu (chyba může být výjimečně i v samotných testovacích vstupech).



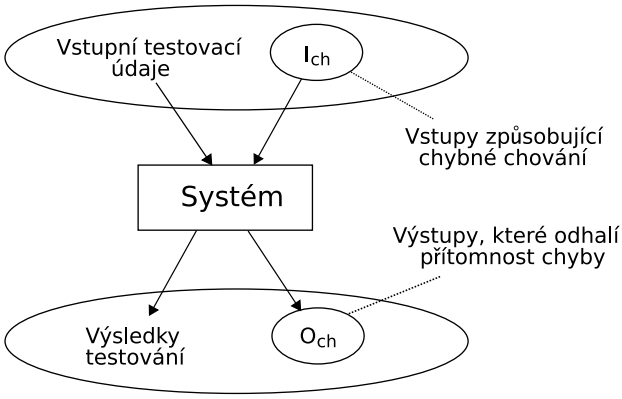
Obrázek 8.2: Proces testování

Celý proces testování vypadá na první pohled jednoduše. Jsou tu však dva zádrhely. Když výsledky neodpovídají očekávaným hodnotám (našli jsme tedy chybu), není vždy jednoduché najít ve zdrojových textech programu příčinu. Principiálně horším problémem však je výběr testovacích vstupů. Množina testovacích vstupů bývá příliš rozsáhlá, a proto můžeme vybrat jenom zlomek z nich. Testování proto nemůže prokázat, že v programu nejsou chyby, ale může pouze ukázat, že tam chyby jsou. Na to, jak vybrat vhodnou množinu testovacích vstupů, se hned podíváme.

8.3.1 Výběr testovacích vstupů

Na obrázku 8.3 vidíme testovaný systém a množinu všech jeho testovacích údajů. Její podmnožina I_{ch} obsahuje všechny vstupní data, která způsobí chybu. Při návrhu testovacích vstupů se snažíme vybrat takové testovací vstupy, pro které je pravděpodobnost příslušnosti do množiny I_{ch} vysoká.

Při výběru testovacích vstupů postupujeme podle testovacího kritéria, které určuje, jaké podmínky musí splňovat množina testovacích vstupů (např. můžeme chtít, aby množina testovacích vstupů pokrývala všechny příkazy v programu). Při výběru testovacích vstupů bychom se měli snažit, aby jednotlivé testovací vstupy odhalily různé chyby – snažíme se, aby byla množina testovacích vstupů co nejmenší, a pro odhalení chyby stačí, když bude odhalena jedním testem. Také bychom se měli snažit, aby testy z vybrané množiny testovacích vstupů odhalily všechny chyby. To je však prakticky nemožné, protože velikost této množiny musí být přijatelná — výsledky testování musíme získat v rozumném čase, abychom je mohli využít, a navíc i příprava a zpracování testů stojí určité úsilí. Hledáme proto vždy kompromis mezi cenou testů a množstvím chyb, které odhalí.



Obrázek 8.3: Množina testovacích vstupů

8.3.2 Testovací kritérium

Při výběru testovacích vstupů vycházíme ze samotného programu (jak uvidíme později, některá testovací kritéria dostupnost zdrojových textů programu nevyžadují), z jeho specifikace — snažíme se vlastně ověřit, že program vyhovuje specifikaci) — a ze zvoleného testovacího kritéria, jak ukazuje obrázek 8.4.

Obrázek 8.4: Aplikace testovacího kritéria

DEF

Spolehlivost a platnost testovacího kritéria

Testovací kritérium K je spolehlivé, když všechny množiny testovacích vstupů splňující kritérium K odhalí stejné chyby. Nezáleží tedy na tom, kterou množinu testovacích vstupů vybereme, protože vždy odhalíme stejné chyby.

Testovací kritérium K je platné, když pro každou chybu v programu existuje množina testovacích vstupů, která splňuje kritérium K a která chybu odhalí.

Snažíme se, aby testovací kritérium bylo spolehlivé i platné, protože když množina testovacích vstupů vybraná podle tohoto kritéria neodhalí žádné chyby, tak v programu žádné chyby nejsou. Bylo bohužel dokázáno, že neexistuje algoritmus, který určí platné kritérium pro libovolný program.

Nyní se seznámíme s některými testovacími kritérii, podle nichž se pak nazývá i samotné testování.

Náhodné testování

Při náhodném testování vybíráme testovací vstupy náhodně například pomocí generátoru náhodných čísel. Jeho nevýhodou je, že výběr okrajových hodnot je málo pravděpodobný, zatímco výskyt programátorských chyb je pro tyto hodnoty

velice vysoký. Náhodné testování proto není vhodné používat samostatně, ale je vhodným doplněním jiného testovacího kritéria.

Funkcionální testování

Při funkcionálním testování se vychází pouze ze specifikace a neuvažuje se vnitřní struktura programu. Funkcionální testování je také známo pod označením *metoda černé skříňky* (angl. *black box*).

Zde doporučujeme, aby si čtenář zopakoval partie matematiky, které se věnují relacím. Doufáme však, že i bez jejich znalosti čtenář pochopí následující informace.

Při funkcionálním testování se využívá rozdělení vstupů/výstupů do tříd ekvivalence. Každý vstup/výstup patří právě do jedné z tříd ekvivalence, pro které je chování systému z určitého hlediska identické (např. pokud se při daném vstupu zjistí chyba, tak stejnou chybu je možné odhalit použitím jiného vstupu z dané třídy ekvivalence). Když už jsou vstupy rozděleny do tříd ekvivalence, vybere se z každé třídy ekvivalence průměrná hodnota nebo medián, hranice třídy a několik náhodných hodnot.

Jako příklad si vezmeme funkci $ABS(x)$, která má vracet absolutní hodnotu z celého šestnáctibitového čísla x. Podle znalosti této funkce identifikujeme dvě třídy ekvivalence, a to intervaly $< -32\,768, -1 >$ a $< 0, 32\,767 >$. Pak testovací vstupy budou následující (pro zjednodušení zápisu zde vlastně uvádíme pouze testovací údaje): $\{-16\,384, -32\,768, -1, -28\,967, -18\,776, 16\,383, 0, 32\,767, 17\,137, 11\,960\}$

Strukturální testování

Při strukturálním testování (jak už název napovídá) se při návrhu testovacích vstupů vychází z vnitřní struktury programu. Tato metoda se také nazývá jako *metoda bílé skříňky* (angl. *white box*). Protože se jedná o testování konkrétní implementace, snažíme se pokrýt různé struktury programu – kritéria tak mohou být založena na tocích řízení (pokrytí cest v programu, pokrytí rozhodovacích bloků nebo podmínek či pokrytí příkazů) nebo na tocích dat.

Pro *otestování* úplnosti množiny testovacích vstupů se používá tzv. *mutační testování*. Když je množina testovacích vstupů vybrána, udělá se v programu úmyslně několik chyb a spustí se testování pomocí vybrané množiny testovacích vstupů. Podle toho, kolik těchto úmyslně vložených chyb je zachyceno, se dá odhadnout, jak úspěšné bude testování (tedy kolik procent chyb v softwaru po testování zůstane).

Jako příklad si vezmeme následující implementaci funkce ABS.


```
int ABS ( int x ) {
    if ( x > 0 )
        return x;
    else
        return -x;
}
```

Pak například množina testovacích vstupů $\{-10, 10\}$ vyhovuje testovacímu kritériu pokrytí příkazů, tzn. budeme mít jistotu, že každý příkaz programu bude při testování proveden alespoň jednou.

V tomto případě intuitivně cítíme, že množina testovacích vstupů navržená metodou černé skříňky je lepší než množina navržená pomocí kritéria pokrytí příkazů. U složitějších programů by však lepších výsledků mělo dosahovat strukturální testování, protože pro výběr testovacích vstupů můžeme využít více informací, a tak množina testovacích vstupů může být výrazně menší při dostatečném otestování programu. S výhodou lze zkombinovat i více testovacích kritérií dohromady.

8.3.3 Strategie testování

Podobně jako při implementaci máme několik možností, jak při testování postupovat. Volba strategie testování často závisí na strategii zvolené pro implementaci.

Při **testování zdola-nahoru** (angl. *bottom-up testing*) se nejdříve testují programové části na nejnižší úrovni abstrakce a pak se pokračuje testováním částí na vyšší úrovni. Testování zdola-nahoru je vhodné použít tehdy, pokud více modulů nižší úrovně představuje obecné moduly, které jsou často používané jinými částmi systému.

Při **testování shora-dolů** (angl. *top-down testing*) se postupuje přesně opačně než při testování zdola-nahoru. Začínáme tedy testovat na nejvyšší úrovni abstrakce a postupujeme až k jednotlivým modulům. Při tomto přístupu musíme řešit problémy se simulací modulů na nižších úrovních a také tento přístup vede k omezené znovupoužitelnosti modulů.

Jednofázové testování (angl. *big-bang testing*) spočívá v samostatném otestování jednotlivých modulů a jejich následné integraci do celého systému. Tato strategie je vhodná pouze pro malé systémy, neboť je náročná identifikace místa chyby při integraci a také lze těžko rozlišit chyby v rozhraní modulů od ostatních chyb.

Jak jsme už několikrát viděli, v praxi se často používá kombinace různých teoretických přístupů. Stejně je tomu i u testování, kdy tzv. **sendvičové testování** (angl. *sandwich testing*) kombinuje strategii testování zdola-nahoru a shora-dolů. Moduly se pak rozdělí do dvou skupin podle jejich převládajícího účelu na logické, které se testují metodou shora-dolů, a funkční, které se testují metodou zdola-nahoru.

Zvláštní strategií testování je **testování porovnáváním** (angl. *comparison testing*), kdy pro testování máme k dispozici několik verzí systému. Může se jednat o prototyp a funkční verzi nebo o verze pro různé operační systémy nebo i o několik nezávisle vyvíjených verzí pro vysoce spolehlivé systémy. Hlavní myšlenkou testování porovnáváním je předpoklad, že jestliže různé verze dávají stejné výsledky, pravděpodobně všechny verze pracují správně. Toto testování však neodhalí situace, kdy jsou v různých verzích tytéž chyby. A ani nezávislý vývoj několika verzí nemůže odhalit chyby ve specifikaci, protože testování slouží pro ověření, že implementace vyhovuje specifikaci.

8.3.4 Zvláštní typy testování

Testovat lze řadu vlastností softwarového produktu — funkčnost, robustnost, přípustný objem dat, dobu odezvy, paměťové nároky, bezpečnost (jak drahé je proniknout do systému), zotavení, dokumentaci, ... My se zde však podíváme na některé zvláštní typy testování, se kterými se můžeme v praxi poměrně často setkat.

Regresní testování

Regresní testování spočívá v opakovaném vyhodnocení testů po každé změně softwarového systému. Regresní testování je zvláště užitečné při opravách systému, kdy se programátor může přesvědčit, že zjištěná chyba byla skutečně opravena,

ale hlavně se může přesvědčit, že její opravou do systém nezanesl chyby nové. Regresní testování musí být automatizované, protože se spouští opakovaně. Ruční provádění regresních testů by totiž vývoj softwaru neúměrně prodražilo.

Akceptační testování

Akceptační testování provádí uživatel na reálných datech. Na jeho základě uživatel určí, zda dodaný produkt splňuje zadání. Pokud není produkt zákazníkem akceptován, dostane dodavatel seznam vlastností, které je nutné ještě opravit. Jestliže zákazník software akceptuje, další změny systému již představují údržbu systému.

Alfa testování

Pro generické softwarové výrobky, kde není možné provést akceptační testy u každého zákazníka (je jich příliš mnoho a jenom málo z nich je ochotno se testování věnovat), se *akceptační testování* provádí jiným způsobem. V první fázi se provádí tzv. **alfa testování**, kdy si vývojový tým pozve do svého sídla vybraného uživatele. Vývojáři pak sledují jeho práci se softwarem a zaznamenávají si objevené chyby (které pak pochopitelně opraví). Při alfa testování se obvykle objevují chyby, které plynou z odlišného způsobu práce uživatele, který software zatím nezná a ani nebývá IT profesionálem, a vývojáře, který software zná dokonale.

Beta testování

Beta testování provádí uživatelé u sebe. Při beta testování se objevují problémy, které plynou z komunikace programu s neznámým prostředím uživatelova počítače. Výsledkem testování je zpráva uživatele, na jejímž základě vývojový tým upraví software a uvolní ho do prodeje.

Alfa i beta testování se samozřejmě může provádět i opakovaně — oprava chyby zjištěné při alfa testování je levnější než kdyby se stejná chyba objevila při beta testování nebo dokonce při provozu.

Kontrolní otázky

8.1 Podíl implementace na celkovém objemu prací klesá, roste nebo se nemění?

8.2 Co je to validace programu?

8.3 Co je to verifikace programu?

8.4 Postačuje správnost softwarového produktu pro jeho úspěšnost?

8.5 Je správnost softwarového produktu nezbytná pro jeho úspěšnost?

8.6 Jaký je rozdíl mezi statickým a dynamickým ověřováním?

8.7 Jaký je vztah mezi dynamickým ověřováním a testováním?

8.8 Co je cílem testování?

8.9 Jakým způsobem se při testování odhalení chyb dosahuje?

8.10 Lze testováním odhalit všechny chyby?

8.11 K čemu slouží testovací kritérium?

8.12 Co je to metoda černé skříňky?

8.13 Co je to metoda bílé skříňky?

8.14 Kdo provádí akceptační testování?

8.15 Jaký je rozdíl mezi alfa a beta testováním?

Kapitola 9

Řízení softwarových projektů

1:20

V této závěrečné kapitole si stručně probereme řízení softwarových projektů, které je neodmyslitelnou součástí všech etap vývoje softwaru. Na řízení softwarových projektů se zde podíváme ze tří pohledů – nejdříve z pohledu softwarového inženýra (nebo chcete-li vedoucího vývojového týmu), pak se na řízení podíváme očima manažera a nakonec si uvedeme několik poznámek k řízení kvality.

Přestože jistě intuitivně chápeme, co projekt znamená, pokusme se ho zde definovat co nejpřesněji.

Projekt

Projekt je časově ohraničené úsilí, které se vyvíjí s cílem vytvořit jedinečný výsledek (např. výrobek nebo službu).

Každý projekt je tedy *časově ohraničený)* — má jednoznačný začátek a konec. Konec projektu je dosažen tehdy, když jsou dosaženy stanovené cíle projektu nebo když se ukáže, že těchto cílů dosáhnout nelze. Každý projekt je z definice *jedinečný* — výsledek projektu se nějak liší od výsledků podobných projektů.

9.1 Pohled softwarového inženýra

Nejdříve se na řízení softwarového projektu podíváme očima vedoucího vývojového týmu, který má na starosti technickou realizaci projektu. Vedoucí vývojového tým musí mít rozsáhlé technické znalosti, značné komunikační dovednosti a musí se také *vyznat v lidech*.

9.1.1 Parametry softwarového projektu

V každém softwarovém projektu existují čtyři základní proměnné, které lze ovlivňovat – cena, čas, rozsah a kvalita. Tyto parametry jsou vzájemně svázány. Když například snížíme náklady, bude to mít za následek pokles kvality a zmenšení rozsahu projektu, nebo když budeme chtít zkrátit čas dodání projektu, povede to ke snížení kvality, rozsahu a pravděpodobně vzroste i cena projektu. Nicméně se to nesmí přehnat ani druhým směrem – například navyšováním ceny nad určitou mez se kvalita projektu nezvýší a ani čas dodání se nezkrátí. Stejně tak neúměrné prodlužování projektu vede jen k oddálení zpětné vazby a tím i pozdnímu odhalení problémů.

Zajímavým parametrem je kvalita. Nízká kvalita sice šetří počáteční náklady, ale o to více bude dodatečných nákladů v budoucnu. Pro dosažení vyšší kvality jsou sice nutné vyšší počáteční náklady, nicméně vyšší kvalita pak výrazně šetří náklady na provoz a údržbu. Co se týče rozsahu projektu, lze zde zopakovat postřehy uvedené dříve, a to, že menší projekty lze vyvíjet rychleji, levněji a s lepší kvalitou než projekty rozsáhlejší.

Jde tedy o to, aby nastavení těchto čtyř proměnných odpovídalo konkrétnímu projektu. Žádný z účastníků však nemůže definovat všechny proměnné současně. Zákazník například může definovat cenu a čas vývoje a vývojový tým pak určí rozsah a kvalitu projektu.

9.1.2 Sestavení vývojového týmu

Správně sestavený vývojový tým připomíná družstvo v kolektivním sportu – i při vývoji softwaru je týmový duch důležitější než kvalita nespolupracujících individualistů. Probereme si tedy nejčastější problémové typy lidí, se kterými se může vedoucí vývojového vývojového týmu setkat, a pokusíme se načrtnout i nejlepší způsob zacházení s nimi.

Beran příliš důrazně prosazuje variantu, kterou považuje za nejlepší. *Beran* může prosadit horší řešení díky své neústupnosti a ne silou svých argumentů. Nejnebezpečnější však je, když se v týmu sejdou dva berani. Jejich neustálé soupeření odčerpává týmu neúměrné množství energie i na vyřešení méně důležitých problémů. Nejvhodnějším řešením takové situace je rozdělit odpovědnost mezi oba *berany* – každému vymezit oblast, ve které rozhoduje a za kterou také nese odpovědnost.

Slabý článek je člen týmu, který v porovnání s ostatními je slabší – má méně znalostí, nižší inteligenci, komunikační problémy nebo není spolehlivý či pracovitý. Nebezpečí *slabého článku* spočívá v tom, že může znehodnotit úsilí celého týmu (podobně jako nejslabší článek řetězu určuje jeho pevnost). Nejlepším řešením je takového člověka vůbec do týmu nepřijmout. To však není vždy možné (a i kdyby bylo, tak se nejslabším článkem stane zase někdo jiný). Vhodným řešením je zařadit *slabý článek* na méně náročnou a méně důležitou pozici (například při testování či při psaní dokumentace může udělat méně škody než při specifikaci a analýze požadavků).

Lidé s ***dělnickou mentalitou*** sice mají dostatečné zkušenosti, ale snaží se *přežít* pracovní dobu s vynaložením co nejmenší námahy. S lehčími případy se může vedoucí týmu smířit, těžší by se měl snažit propustit. Vedoucí by jim měl přidělovat dobře měřitelnou práci (i zde je výborným kandidátem psaní dokumentace) a odvedenou práci pravidelně kontrolovat.

Snaživec odvádí na své pozici dobrou práci, ale chce na prestižnější místo, na které nemusí mít předpoklady. Řešením je kompromis – najít pozici, která by snaživce (alespoň částečně) uspokojila, ale kde by nemohl ohrozit projekt.

Při sestavování týmu hraje důležitou roli také jeho velikost, což není nic jiného než hledání kompromisu mezi cenou a časem. Menší tým totiž pracuje efektivněji, protože nepotřebuje tolik času na vnitřní komunikaci. Větší tým však může projekt dokončit dříve a díky většímu objemu znalostí a zkušeností může najít lepší řešení.

Situaci vedoucího vývojového týmu ilustruje průzkum, jak programátoři tráví svůj pracovní čas:

psaní programů	*13%*
čtení programů a příruček	*6%*
komunikace týkající se práce (konzultace, porady, . . .)	*42%*
ostatní (včetně osobních věcí)	*39%*

Ve fakultní knihovně máte k dispozici knihu *Co programátory ve škole neučí* od Petra Palety, ve které najdete zábavnou formou podané užitečné postřehy z praxe softwarového inženýra.

9.2 Management projektů

Management

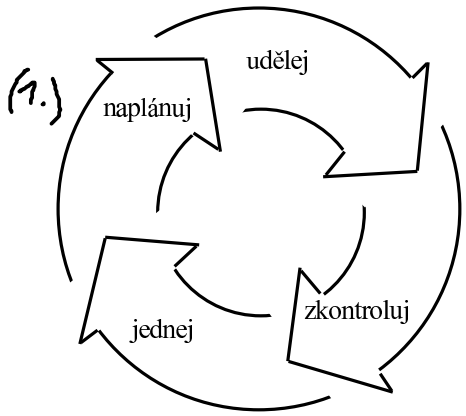
Management je proces koordinace činností skupiny lidí, který realizuje jednotlivec nebo skupina lidí za účelem dosažení stanovených cílů. Tyto cíle se nedají dosáhnout jenom prací jednotlivce.

Pro manažery nejsou nezbytné hluboké technické znalosti z oblasti, kterou řídí, ale především umění jednat s lidmi.

9.2.1 Demingův manažerský cyklus

Manažerské procesy by měly probíhat v nekonečné smyčce zvané Demingův manažerský cyklus (je na obrázku 9.1), který se skládá z následujících kroků:

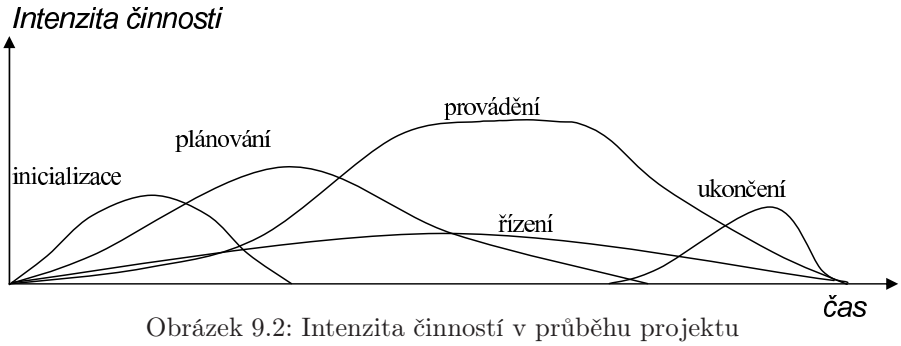
1. Naplánuj (Plan) – navrhni nové nebo uprav stávající manažerské činnosti pro zlepšení výsledků.
2. Udělej (Do) – proveď plán a zjisti jeho účinnost.
3. Zkontroluj (Check) – zhodnoť dosažené výsledky.
4. Jednej (Act) – rozhodni, které změny jsou potřeba pro další zlepšení procesu řízení.



Obrázek 9.1: Demingův manažerský cyklus

9.2.2 Etapy projektu z pohledu managementu

Intenzitu jednotlivých činností v průběhu typického projektu vidíme na obrázku 9.2.



Obrázek 9.2: Intenzita činností v průběhu projektu

Inicializace je úvodní etapou projektu a jejím cílem je rozpoznání, že projekt může začít, a také získání všech relevantních informací potřebných pro plánování

projektu. Tato etapa může podle typu a velikosti projektu trvat několik dnů až několik měsíců. Součástí této etapy bývá studie vhodnosti, která má odpovědět na otázku, zda je rozumné se do projektu vůbec pouštět, jinými slovy zda očekávané přínosy projektu jsou větší než předpokládané náklady na projekt a zda rizika s projektem spojená nejsou příliš vysoká.

Plánování se snaží snížit neurčitost, zlepšit výkonnost, zajistit lepší pochopení cílů projektu a vytvořit základnu pro sledování a řízení práce. Při plánování se definují požadavky na zdroje, požadavky na práci a definuje se kvalita a kvantita práce. Plánování je intenzivní hlavně v počátečních fázích projektu, v průběhu provádění a řízení se plány upravují podle potřeby. Plánování obvykle představuje malou část z celkově vynaložené práce, ale může představovat pro projekt značné riziko. To však neznamená, že je nutné vytvářet *vyčerpávající* plány, ale naopak – plánování by mělo být tak podrobné, jak je to nezbytné, a ne tak, jak je to možné.

Řízení spočívá ve sběru a rozšiřování informací v průběhu celého projektu. Sleduje se zejména aktuální stav projektu a na základě jeho porovnání s plánem se provádí změny a odhaduje další průběh projektu. Dobré řízení se preventivními opatřeními snaží problémům předcházet, než je později řešit.

Provádění není nic jiného než realizace plánu projektu a ze všech etap spotřebuje nejvíce času a peněz. Manažer projektu koordinuje a usměrňuje provádění činností z plánu – přiděluje úkoly, stanovuje priority, rozděluje pravomoci, sleduje postup prací a rozhoduje o umístění důležitých zdrojů. Zatímco výsledkem ostatních etap jsou dokumenty potřebné pro projekt samotný, výsledkem provádění je dosažení cíle projektu (např. výrobek či služba).

Při **ukončení** se ukončují kontrakty s dodavateli a zákazníkům se dodávají výrobky či poskytují služby. Je také užitečné zaznamenat nové poznatky, zkušenosti a poučení pro budoucí projekty. V zásadě pouze v této etapě se projeví specifické vlastnosti softwarových projektů – údržba softwaru (na rozdíl třeba od údržby mostu či budovy) představuje významný podíl úsilí a peněz na celém projektu. Manažeři proto musí vyčlenit dostatečné personální a finanční prostředky na údržbu softwaru.

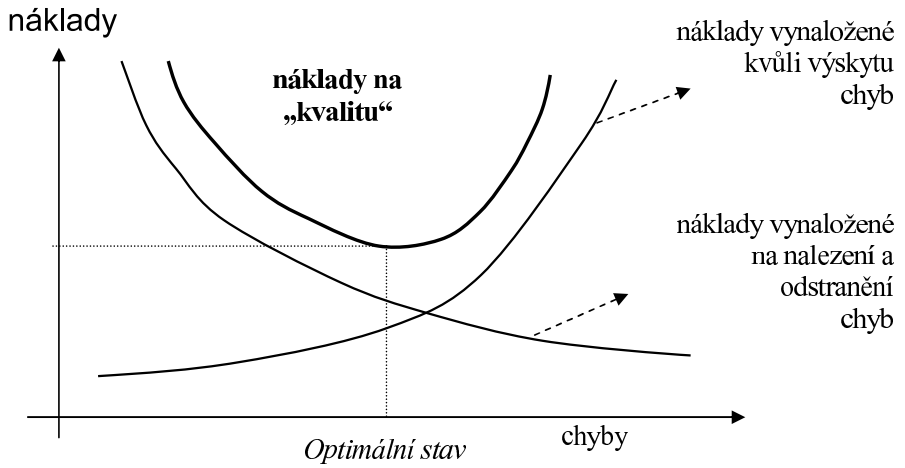
9.3 Řízení kvality softwarových projektů

Obvyklý postup při tvorbě softwaru spočívá v co nejrychlejší implementaci programu *neznámé kvality* a rychlém testování s cílem najít a odstranit chyby a nedostatky. V žádném jiném technickém oboru se nevytvářejí výrobky nekontrolovatelné kvality, přičemž by se spoléhalo na testování.

Kvalita určitého softwarového výrobku se může jevit různě pro různé aktéry, kteří se se softwarem dostanou do styku. Pro uživatele může kvalita odpovídat jednoduchosti obsluhy systému a spolehlivému a efektivnímu provádění jednotlivých funkcí systému. Pro tvůrce a údržbáře systému kvalita odpovídá čitelným a modifikovatelným programům a srozumitelné a přesné dokumentaci. Pro manažera kvalita obvykle odpovídá dodání výrobku včas, v rámci rozpočtu a dohodnutých požadavků (tento pohled na kvalitu jsme zavedli i my – viz obrázek 2.3 na straně 13).

U softwaru občas svádí jeho kvalitu ztotožňovat s jeho funkcionalitou. Není to ale správné, protože systém s řadou funkcí může mít nízkou kvalitu (například příliš chyb) a naopak systém s pouze základními ale dobře otestovanými funkcemi budeme považovat za kvalitní. Z toho by se tedy dalo usuzovat, že kvalitní software nesmí obsahovat chyby a že software bez chyb je kvalitním softwarem. Není to

však pravda, protože absence chyb v softwaru není dostačující ani nezbytnou podmínkou pro kvalitní software. Kromě toho v každém trochu složitějším softwaru nějaké chyby jsou.



Obrázek 9.3: Optimální kvalita softwaru

Vztah mezi počtem chyb v softwaru a jeho kvalitou osvětluje obrázek 9.3. Na ose x je vynesen počet chyb v softwaru a na ose y náklady. Pokud budeme v softwaru tolerovat velké množství chyb, náklady na jeho vývoj budou nízké. Jakmile však budeme tyto chyby chtít odstranit, bude nás to stát další úsilí navíc a náklady na vývoj softwaru proto porostou. Čím méně chyb v softwaru budeme tolerovat, tím bude odhalování chyb nákladnější. Tento průběh je v grafu vyjádřen klesající křivkou. Dalším zdrojem nákladů spojených se softwarem jsou dodatečné náklady způsobené chybami softwaru. Když bude v softwaru chyb málo, budou se projevovat pouze výjimečně a náklady spojené na odstraňování jejich následků budou nízké. Jak však poroste počet chyb v softwaru, porostou i tyto náklady. Tato závislost je znázorněna v grafu rostoucí křivkou. Celkové náklady na použití softwaru jsou dány jednak náklady na jeho vývoj a náklady vynaložené na odstraňování následků chyb softwaru – celkovým nákladům v grafu odpovídá horní křivka ve tvaru písmene U. Nejvyšší kvalita (tedy nejnižší celkové náklady za použití softwaru) je tedy někde uprostřed. V praxi však není jednoduché odhadnout, kdy jsme se do tohoto stavu dostali a kdy je tedy vhodné přerušit testování a opravu chyb.

9.3.1 Normy pro systém zajištění kvality

Pro softwarové produkty se vychází z předpokladu, že pokud má organizace *kvalitní* proces tvorby výrobku (tj. softwaru), budou i její výrobky kvalitní. Tento přístup se používá hlavně proto, že v softwarovém inženýrství není jednoduché měřit kvalitu programů pomocí nějaké výstupní kontroly.

ISO 9000

V evropských zemích se pro řízení a zajištění kvality používá mezinárodní soustava norem ISO 9000, kterou definovala mezinárodní organizace pro normy ISO (International Standards Organisation). Tato soustava norem je sice primárně určena pro hromadnou výrobu, ale protože obsahuje všeobecné požadavky platné pro libovolnou oblast výroby, lze je použít i pro softwarový průmysl.

Zavedení ISO 9000 (tedy získání certifikátu) má pro organizaci své výhody i nevýhody. Normy ISO 9000 mohou sloužit jako slušný základ pro dobře fungující

výrobu, dávají zákazníkovi určitou představu o organizaci a zvyšují konkurenceschopnost a zlepšují jméno organizace. Důraz norem ISO 9000 je však kladen především na kontrolu a ne na kvalitou samotnou, takže snadno to sklouzne k byrokratickému přístupu. Zavedení ISO 9000 je poměrně drahé a časově náročné, což se musí projevit i na ceně výrobků, takže potom skutečná kvalita výrobků (tj. cena × čas × funkce) může i klesnout.

`http://www.iso.ch/`

9.4 Měření v softwarovém inženýrství

Bez měření v softwarovém inženýrství by nebylo možné softwarové projekty řídit. Nicméně si musíme být vědomi toho, že měření není nikdy dokonalé a že zavedením určité metriky ovlivníme vlastní projekt. Vysvětlíme si to na následujícím příkladu.

Jsme manažeři softwarového projektu a naším úkolem je ohodnotit práci programátorů podle jejich výkonů. Nejdříve použijeme metodu, která se nám již dříve osvědčila pro hodnocení dělnických profesí – budeme hodnotit výkony programátorů podle doby, kterou strávili v práci. Hned po prvním měsíci zjistíme, že to není správné řešení. Programátoři se jednak cítí sledováním doby strávené v práci omezováni a navíc intuitivně cítíme, že naměřené hodnoty naprosto neodpovídají skutečně odvedeným výkonům. (Pozorný čtenář už samozřejmě ví, proč tomu tak je: produktivita jednotlivých programátorů se diametrálně liší a navíc programátoři značnou část pracovní doby věnují soukromým záležitostem.)

Uznáme proto, že programátor není dělník, a druhý měsíc zkusíme jiné řešení – budeme výkony měřit jako počet řádků programu, které programátor napsal. Brzy však za námi přijde vedoucí vývojového týmu s tím, že programátoři změnili styl psaní programů a nyní vkládají spoustu zbytečných prázdných řádků a podmínky přepisují na více řádků. To je samozřejmě nepříjemné a tak se rozhodneme, že místo počtu řádků budeme hodnotit počet znaků, o které se zvětšil soubor se zdrojovým textem programu. Zanedlouho je však vedoucí vývojového týmu opět v naší kanceláři s tím, že programátoři začali psát zbytečně rozsáhlé komentáře a že se ve skutečnosti postupuje pomaleji než dříve.

Zkusíme tedy ještě jeden pokus – budeme měřit, o kolik se zvětšil zdrojový kód, ovšem po vypuštění komentářů a oddělovačů. Nejdříve se to zdá být dobré, ale po dvou měsících zjistíme, že kvalita zdrojových textů výrazně poklesla – zdrojové texty obsahují mnoho nedostupného kódu, identifikátory proměnných jsou zbytečně dlouhé, kód není prakticky komentován a řadu procedur a funkcí by bylo možné napsat jednodušeji a efektivněji. Měření výkonů programátorů nakonec vzdáme a hodnocení jednotlivých programátorů ponecháme na subjektivním posouzení vedoucím vývojového týmu.

Po uvedení tohoto příkladu vás jistě napadne, že metriky vlastně nejsou v softwarovém inženýrství potřeba. Jsou, ale musí se používat rozumně. Ve výše uvedeném příkladu jsou totiž reakce programátorů na zavedení metriky dovedené ad absurdum, aby se o to více ozřejmily principy. Ale i náš příběh můžeme s trochou nadsázky skončit *happy endem*.

Vedoucí vývojového týmu použije kombinaci všech výše uvedených metrik – nenápadně sleduje, jak dlouho zůstávají programátoři v práci, občas zkontroluje, zda programátoři skutečně pracují, měří počet řádků zdrojového textu programu,

porovnáním velikosti programu bez komentářů a s komentáři ověřuje, zda jsou programy dostatečně komentované, a namátkovým nahlédnutím do zdrojových textů také rychle pozná kvalitu práce jednotlivých programátorů. Kromě toho, že má přehled o tom, co kdo skutečně udělal, je to člověk zkušený a ostatními programátory respektovaný, takže s jeho z pohledu manažerů subjektivním *hodnocením jsou všichni spokojeni a kvalita vytvářených zdrojových textů se výrazně zlepšila.*

Měření může být buď přímé (tj. hodnotu získáme přímým sledováním atributu, např. počet řádků programu) a nepřímé (tj. hodnotu odvodíme z jiných atributů, které lze měřit přímo, např. udržovatelnost lze určit jako čas potřebný pro odstranění chyby. I přesně naměřené hodnoty mohou mít různé interpretace, například počet chyb nalezených za jednotku času může reprezentovat kvalitu testování nebo (ne)spolehlivost programu. Pro úspěch projektu je důležitá dohoda na kritériích přijetí projektu a tato kritéria musí být měřitelná.

Představme si, že v podmínkách pro přijetí softwaru zákazníkem je uvedeno, že uživatelské rozhraní musí být přátelské. *Při sestavování podmínek pro přijetí softwaru se nám to může zdát jako smysluplný požadavek na software. Když však budeme software předávat zákazníkovi, on ho nepřijme s tím, že uživatelské rozhraní není uživatelsky přátelské, aniž je ochoten nebo schopen to blíže vysvětlit. A protože nemůžeme přátelskost uživatelského rozhraní nijak změřit, musíme software upravovat tak dlouho, dokud se uživateli nebude zdát dostatečně přátelský.*

Když však v podmínkách pro přijetí softwaru je uvedeno, že Operátoři musí být schopní začít pracovat s libovolnou funkcí programu do 30 sekund od usednutí k terminálu, *výše uvedená situace nenastane. V případě neshody se zákazníkem posadíme vyškoleného operátora k terminálu, zadáme mu, aby provedl vybrané funkce, a se stopkami v ruce měříme čas. Ověření, že software splňuje tuto měřitelnou podmínku, navíc může provést ještě před předáním softwaru zákazníkovi nebo dokonce ještě i před dokončením softwaru.*

Doufáme, že jsme čtenáře dostatečně přesvědčili o tom, že měření je pro softwarové inženýrství důležité, a že se již nemůže dočkat, až uvedeme další metriky.

9.4.1 Metriky pro software

Pro odhad času a nákladů a pro měření produktivity se používá jako metrika počet řádků zdrojového textu programu (LOC = Lines Of Code nebo kKLOC = kilo LOC), pro dokumentaci se používá počet stránek.

Pro měření spolehlivosti se používá střední doba mezi výpadky systému (MTBF = Mean Time Between Failures), která je součtem střední doby opravy (MTTR = Mean Time To Recovery) a střední doby do následujícího výpadku (MTTF = Mean Time To Failure). Dostupnost (pravděpodobnost, že v daném čase program pracuje správně) pak můžeme nepřímo určit jako podíl MTTF/MTBF.

Pro měření složitosti softwaru se dá použít jako metrika počet souborů, velikost programu (počet příkazů, řádků, ...), počet větvení, hloubka vnoření řídících struktur, počet cyklů, a podobně. Je však potřeba tyto metriky interpretovat velice opatrně – mohou například sloužit pro odhalení problémových (příliš složitých) modulů. Pro dokumentaci lze použít jako metriku průměrnou délku věty.

Metriky se zavádějí i pro studijní opory. Kromě počtu stran studijní opory se sleduje i průměrný počet vět v souvětí a počet cizích slov. Vám jako uživatelům studijní opory je jistě jasné, že dlouhá a složitá souvětí ani množství cizích slov k pochopení probírané látky nepřispívají.

Velice důležitým parametrem softwarového produktu je počet chyb. Počet chyb však nelze určit přímo, protože všechny chyby v softwaru nikdy neznáme. Můžeme

však počet chyb odhadnout buď podle intenzity jejich výskytu nebo pomocí chybovosti, což je průměrný počet chyb v jenom kLOC. Chybovost můžeme určit tak, že si projdeme jenom malou část kódu ale zato důkladně, takže v něm najdeme prakticky všechny chyby. Celkový počet chyb pak získáme jako součin chybovosti a celkové velikosti zdrojových kódů softwaru.

Pro měření udržovatelnosti můžeme použít střední dobu potřebnou na opravu chyby, střední dobu potřebnou pro pochopení logiky modulu či střední dobu na nalezení příslušné informace v dokumentaci.

Pro vyjádření času vynaloženého na vývoj systému se používají člověko-měsíce nebo v případě větších projektů člověko-roky. Například 24 člověko-měsíců znamená, že na projektu pracoval jeden člověk dva roky nebo dva lidé jeden rok nebo tři lidé osm měsíců nebo čtyři lidé půl roku, atd.

Shrnutí

Řízení softwarových projektů není exaktní věda a tak manažeři potřebují především komunikační dovednosti a přirozenou inteligenci, zatímco bez detailních technických znalostí z oboru projektu se mohou docela dobře obejít. V této kapitole jsme si uvedli několik poznámek k řízení softwarových projektů, a to z pohledu vedoucího vývojového týmu, z pohledu manažera i z pohledu řízení kvality. Vedoucí vývojového týmu je součástí vývojového týmu a má zásadní vliv na atmosféru uvnitř týmu i na jeho efektivitu a produktivitu. Úkolem manažerů je zajistit všechno tak, aby se vývojový tým mohl věnovat řešení technických problémů. Zavedení metod pro řízení kvality obvykle sníží riziko neúspěšnosti projektů a obvykle i ušetří náklady v pozdějších fázích projektu (zavedení kvality ovšem zvýší náklady v počátečních fázích projektu). V evropském kontextu se pro řízení kvality používají normy ISO 9000. Aby bylo možné projekty řídit, musí být k dispozici metriky. Uvedli jsme si několik nejčastěji používaných softwarových metrik a upozornili na to, že se získanými výsledky je potřeba nakládat velice opatrně.

Kontrolní otázky

9.1 Co je to projekt?

9.2 Jaké jsou čtyři proměnné, které můžeme u softwarového projektu ovlivňovat?

9.3 Pracuje efektivněji menší nebo větší vývojový tým? A proč?

9.4 Kdy a proč se provádí studie vhodnosti?

9.5 Jaký je vztah mezi počtem chyb v softwaru a jeho kvalitou?

9.6 Proč se zavádí normy pro řízení kvality do vývoje softwaru?

9.7 Jaký je rozdíl mezi přímým a nepřímým měřením?

9.8 Jaká metrika se používá pro měření spolehlivosti softwarových systémů?

9.9 Měří se dostupnost přímo nebo nepřímo?

Odpovědi na kontrolní otázky

1 Úvod

2 Softwarové inženýrství

2.1 Co je to softwarové inženýrství?

Softwarové inženýrství je systematický přístup k vývoji, nasazení a údržbě softwaru.

2.2 Jaký je vztah mezi softwarovým inženýrstvím a programováním?

Programování je jedna z oblastí, kterou se softwarové inženýrství zabývá.

2.3 Jak se projevuje softwarová krize?

Softwarová krize se projevuje neúnosným prodlužováním a prodražováním projektů, nízkou kvalitou výsledných produktů, problematickou údržbou a nízkou produktivitou práce programátorů.

2.4 Jaký je rozdíl mezi programem a softwarovým systémem?

Program vytvořil autor pro svoji potřebu, zatímco softwarový systém je sbírka komunikujících programů určená pro uživatele. Vytvoření softwarového systému je výrazně náročnější než vytvoření programu (komunikační rozhraní, důkladné testování, dokumentace).

2.5 Čím je určena kvalita výrobku?

Kvalita výrobku je určena jeho cenou, časem potřebným pro jeho výrobu a splněním požadavků na něj kladených.

2.6 Jaký je rozdíl mezi zákaznickým a generickým softwarovým výrobkem?

Zákaznický software je vytvářen podle požadavků konkrétního zákazníka a je obvykle určen pouze pro něho. Jeho cena je tedy značně vysoká. Generický (nebo též krabicový) software je nejdříve vyvinut a pak prodáván libovolným zájemcům.

2.7 Co je to správnost softwaru?

Správnost je míra, do jaké software vyhovuje specifikaci.

2.8 Co je to použitelnost softwaru?

Použitelnost je úsilí, které je nutné vynaložit na to, aby se dal software používat.

2.9 Co je to udržovatelnost softwaru?

Udržovatelnost je úsilí, které je potřeba vynaložit na další vývoj a údržbu softwaru podle měnících se potřeb zákazníka a také v důsledku měnícího se okolí.

2.10 Jaké jsou čtyři nevyhnutelné problémy při tvorbě softwaru?

Složitost, přizpůsobivost, nestálost a neviditelnost.

2.11 Co je to syndrom 90 % hotovo?

Odhady aktuálního stavu projektu nevycházejí ze skutečně hotového, ale pouze z odpracovaného. Čím je větší rozdíl mezi plánem a skutečností, tím jsou takové odhady více zavádějící.

2.12 Jaká je typická křivka stárnutí softwaru?

Viz obrázek 2.5 na straně 16.

3 Modely životního cyklu softwaru

3.1 Kteří aktéři vstupují do procesu tvorby softwaru?

Zákazník – sponzoruje vývoj softwaru a specifikuje požadavky na software, **dodavatel** – vyvíjí software, **uživatelé** – upřesňují požadavky a software používají.

3.2 Proč se při vývoji softwaru uplatňuje dekompozice?

Dekompozice umožňuje rozložit složitý systém na podsystémy, jejichž zvládnutí je snadnější a které navíc mohou být vyvíjeny nezávisle.

3.3 Jaké jsou nevýhody dekompozice?

Při dekompozici je nutné navíc definovat rozhraní mezi podsystémy a podsystémy je po jejich dokončení potřeba integrovat a otestovat jejich vzájemnou spolupráci.

3.4 Co je to model životního cyklu softwaru?

Model životního cyklu softwaru definuje etapy vývoje softwaru a pro každou etapu dále definuje nutné činnosti a její vstupy a výstupy.

3.5 Vyjmenujte etapy při vývoji softwarového systému.

1. Analýza a specifikace požadavků
2. Architektonický a podrobný návrh
3. Implementace
4. Integrace a testování
5. Provoz a údržba

3.6 Která etapa obvykle spotřebuje nejvíce zdrojů?

Nejvíce úsilí (času a peněz) je obvykle věnováno provozu a údržbě. To samozřejmě neplatí pro neúspěšné projekty, které se do provozu vůbec nedostanou...

3.7 Jaké jsou hlavní úkoly analýzy a specifikace požadavků?

Získat, analyzovat, definovat a specifikovat požadavky na software, provést studii vhodnosti a identifikovat rizika a provést jejich analýzu, naplánovat akceptační testování.

3.8 Co je náplní architektonického návrhu?

Ujasnit si koncepci systému, provést jeho dekompozici, naplánovat testování celého systému a postup jeho nasazení.

3.9 Jaký je rozdíl mezi testováním systému a akceptačním testováním?

Testování systému provádí dodavatel a jeho cílem je odhalit v systému chyby, zatímco akceptační testování provádí zákazník a jeho cílem je zjistit, zda systém vyhovuje jeho potřebám (odpovídá specifikaci požadavků).

3.10 Nakreslete vodopádový model životního cyklu softwaru.

Viz obrázek 3.2 na straně 23.

3.11 Kdy lze úspěšně použít vodopádový model pro vývoj reálného softwaru?

Vodopádový model lze použít v praxi tehdy, když jsou všechny požadavky na software známy předem a v průběhu vývoje se nemění.

3.12 Jaká je základní myšlenka iterativního modelu?

Rozdělením vývoje softwaru do iterací se dosáhne toho, že uživatel uvidí spustitelnou verzi softwaru výrazně dříve než u vodopádového modelu, takže případné problémy jsou objeveny dříve a tudíž opraveny levněji.

3.13 Jaký je účel prototypu při vývoji softwaru?

Cílem prototypu je odhalit chyby ve specifikaci požadavků co nejdříve, aby způsobily co nejmenší škody. Prototyp je po použití zahozen.

4 Objektová orientace

4.1 Stručně charakterizujte rozdíl mezi objektově orientovaným a strukturovaným přístupem k analýze a návrhu.

Strukturovaná analýza a návrh chápe systém jako kolekci fukcí operujících s daty. Objektově orientovaná analýza a návrh chápe systém jako kolekci vzájemně komunikujících objektů, které zapouzdřují data a funkcionalitu.

4.2 Stručně charakterizujte pojem abstrakce.

Abstrakce spočívá v klasifikaci entit řešeného problému do abstraktních struktur. Klasifikace je založena na rozpoznávání podobností. Smyslem abstrakce je zjednodušení pohledu na systém bez ztráty jeho smyslu.

4.3 Stručně charakterizujte pojem polymorfismus.

Výskyt různých chování (implementací chování) na základě stejné zprávy.

4.4 Stručně charakterizujte pojem zapouzdření.

Ukrytí implementačních detailů objektu. K objektu lze přistupovat jen prostřednictvím zpráv.

4.5 Stručně charakterizujte pojem dědičnost.

Sdílení implementace objektů. Objekty mohou sdílet chování a proměnné jiných objektů – toto chování mohou předefinovat či přidat další funkcionalitu. Dědičnost organizuje a usnadňuje polymorfismus a zapouzdření.

4.6 Stručně charakterizujte pojmy časná vazba, pozdní vazba a přepisování.

Jedná se o techniky výběru implementace operace, tj. která metoda se vyvolá při zaslání zprávy objektu. Časná vazba metody vybírá již při překladu, pozdní vazba metody vyhledává dynamicky, za běhu programu. Pozdní vazba je nutná pro dosažení polymorfismu.

4.7 Vysvětlete pojmy identita objektu a shodnost objektů.

Shodnost objektů definuje relaci mezi dvěma objekty, která je založena na porovnání jejich stavů – dva objekty jsou shodné, pokud je shodný jejich stav. Identita objektu vyjadřuje skutečnost, že každý objekt je jedinečný bez ohledu na třídu a stav – dva objekty se stejným stavem jsou shodné, ale ne identické (jedná se o dva různé objekty).

4.8 Čím se vyznačují třídně založené jazyky? Jaké třídně založené jazyky znáte?

Zavádějí pojem třída, což je generická definice pro množinu podobných objektů – objekty jsou pak instancemi tříd. Instance stejné třídy sdílejí chování – třída tedy implementuje chování podobných objektů. Dědičnost je vyjádřena dědičností tříd – chování je sdíleno mezi třídami.

4.9 Čím se vyznačují prototypově založené jazyky? Jaké prototypově založené jazyky znáte?

Pracují přimo s objekty – nové objekty se vytvářejí klonováním již existujících. Dědičnost je vyjádřena delegováním mezi objekty – objekt může určit množinu objektů, kterým deleguje pro něj neznámé zprávy. Vždy existují nějaké počáteční objekty (prototypy), které lze použít pro vytváření nových či pro sdílení základní funčnosti objektů.

4.10 Stručně charakterizujte staticky typované jazyky.

Typová kontrola probíhá během překladu (kompilace). Typickými představiteli jsou jazyky C++ či Java.

4.11 Stručně charakterizujte dynamicky typované jazyky.

Typová kontrola probíhá až za běhu programu. Typickými představiteli jsou jazyky Smalltalk, Python či Self. Tyto jazyky se vyznačují tím, že jejich součástí je virtuální stroj.

4.12 Vyjmenujte a stručně charakterizujte základní vztahy mezi třídami.

Asociace – identifikuje funkční vztahy mezi třídami, odráží reálnou vazbu mezi reprezentovanými objekty (např. podnik zaměstnává několik pracovníků). Objekty asociovaných tříd jsou implementovány jako proměnné objektu. Závislost – zachycuje vztahy mezi třídami, které vyplývají z podstaty modelování objektů (např. argumentem metody je objekt dané třídy). Zobecnění (dědičnost) – vyjadřuje sdílení chování mezi třídami (třída může být speciálnější variantou jiné třídy). Realizace – vztah mezi třídou a rozhraním, vyjadřuje skutečnost, že třída implementuje (realizuje) jisté rozhraní.

4.13 Identifikujte vztahy mezi třídami na obrázku 4.2 a vyjmenujte informace o jednotlivých vztazích.

Zaměstnání – Osoba má roli zaměstnance a může mít nejvýše jednoho zaměstnavatele. Společnost má roli zaměstnavatele a může zaměstnávat 0 až N zaměstnanců. Bydliště – Osoba má roli obyvatele a má právě jedno bydliště (objekt třídy Okres). Okres může mít 0 až N obyvatel. Sídlo – Společnost alespoň v jednom Okrese, může však mít více sídel. Okres může hostit 0 až N Společností.

5 Analýza a specifikace požadavků

5.1 Co je cílem analýzy a specifikace požadavků?

Stanovení služeb, které zákazník od systému požaduje a vymezení podmínek jeho provozu a vývoje.

5.2 Jaké znáte typy požadavků?

Fukcionální požadavky, požadavky na provoz systému, požadavky na výsledný systém, požadavky na vývojový proces, požadavky na rozhraní, externí požadavky.

5.3 Jaké jsou problémy při specifikaci požadavků?

Největší problémy pramení z používání přirozeného jazyka: vyřazení entity z textu specifikace, zkreslení skutečného významu (nejasný kontext, neúplná informace apod.) a zobecnění (výrok nemusí platiti vždy). Další problémy souvisejí např. s tím, že zákazník není schopen přesně vyjádřit požadavky (protože je často na začátku nezná), nebo protože se zákazník neorientuje v problematice vývoje softwaru a vývojář v problematice řešeného problému.

5.4 Co je diagram případu užití a k čemu slouží?

Je to diagram z jazyka UML. Slouží ve fázi specifikace pro nalezení případů užití systému (jednotlivých funkcí) a účastníků, kteří na případech participují. Zo-

brazuje hranice systému, případy užití, aktéry (účastníky) a interakce mezi aktéry a případy.

5.5 Co je detail případu užití?

Podrobnější popis funcionality jednoho případu užití, často ve formě speciální tabulky.

5.6 K čemu slouží diagramy analytických tříd a objektové diagramy?

Objektové diagramy pomáhají identifikovat třídy a vazby mezi nimi. Anlytické třídy pak klasifikují objekty do základních tříd a vazeb – zaměřují se pouze na model reality bez nutné návaznosti na implementaci (to je pak součástí návrhových tříd). Analytická třída obsahuje jen nejpodstatnější atributy a operace.

5.7 Jaké znáte diagramy interakce, jaký je mezi nimi rozdíl?

Diagram spolupráce (collaboration diagram) – reprezentuje strukturální relace mezi objekty a vazby mezi nimi na základě zasílání zpráv. Sekvenční diagram (sequence diagram) – reprezentuje časově orientovanou posloupnost předávání zpráv mezi objekty.

6 Strukturovaný přístup k analýze

6.1 Jaké jsou základní modely strukturované analýzy?

Funkcionální model (Data Flow Diagram – DFD), datový model (Entity Relatioship Diagram – ERD) a model dynamického chování (State Transition Diagram – STD).

6.2 Charakterizujte DFD.

Jde o funcionální model zachycující funkcionality systému – funkce, aktéry, datové sklady a datové toky.

6.3 Charakterizujte ERD.

Datový model zachycující statická data a relace mezi nimi.

6.4 Definujte pojmy entita a entitiní množina

Entita vyjadřuje věc reálného světa, která musí být rozlišitelná od jiných entit. Entita má své vlastnosti. Entitní množina je množina entit téhož typu, které sdílí tytéž vlastnosti.

6.5 Definujte pojmy vztah a vztahová množina.

Vztah je asociace mezi dvěma (nebo více) entitami, které spolu logicky souvisejí (např. klient banky vlastní účet). Vztah může mít vlastnosti (atributy). Vztahová množina množina vztahů téhož typu (tj. mezi stejnými entitními množinami.

6.6 Vysvětlete pojmy stupeň, kardinalita a členství vztahu.

Stupeň vyjadřuje počet entitních množin zapojených do jedné vztahové množiny (většinou jsou binární, mohou být ternární apod.) Kardinalita vyjadřuje maximální počet vztahů daného typu (v dané vztahové množině). Členství vyjadřuje povinnost účasti entity v daném vztahu (tj. zda vztah mezi entitami musí nebo nemusí existovat).

6.7 Co je primární klíč a co pro něj musí platit?

Primární klíč je jednoznačný identifikátor entity. Hodnota primárního klíče musí být v rámci entitní množiny jedinečná a minimální (podmínka minimální hodnoty je nutná – pokud by nebyla, mohl by být primární klíč tvořen všemi atributy entity).

7 Návrh

7.1 Co je to návrhová třída a diagram návrhových tříd?

Třída vycházející z analytické třídy, kterou upřesňuje. Návrhovou třídu lze přímo implementovat v programovacím jazyce. V diagramu návrhových tříd se uplatňují třídy z doménového řešení (knihovny apod.)

7.2 Co je to rozhraní, čím se vyznačuje?

Rozhraní specifikuje množinu operací (metody) pro splnění požadované odpovědnosti objektu. Objekt implementující rozhraní pak může být použit všude tam, kde se předpokládá toto rozhraní, což umožňuje flexibilnější modelování a programování.

7.3 Čím se vyznačuje etapa návrhu s využitím prostředků jazyka UML?

V etapě návrhu dochází především k doplnění a upřesňování analytických diagramů, např. diagramů tříd a interakce.

7.4 Čím se vyznačují stavové diagramy (statecharts) jazyka UML?

Stavové diagramy modelují chování reaktivních objektů, tj. objektů, které reagují na vnější události (může jít o třídy, případy užití či podsystémy). Nejčastěji se používají pro modelování chování tříd (resp. instancí tříd). Stavové diagramy bývají často používané v metodologiích založených na testování či simulaci modelů – stavové diagramy lze (s případnými drobnými modifikacemi) dobře simulovat nebo verifikovat.make

8 Implementace a testování

8.1 Podíl implementace na celkovém objemu prací klesá, roste nebo se nemění?

Podíl implementace na celkovém objemu prací postupně klesá. Přispívá k tomu softwarové inženýrství například zavedením nových programovacích jazyků s vyšší mírou abstrakce nebo novými technikami pro zvýšení znovupoužitelnosti (objekty, komponenty, ...).

8.2 Co je to validace programu?

Validace je ověření, že vyvíjený software splňuje požadavky zákazníka (tzn. vytváříme správný výrobek).

8.3 Co je to verifikace programu?

Verifikace je ověření, že vyvíjený software vyhovuje specifikaci (tzn. vytváříme výrobek správně).

8.4 Postačuje správnost softwarového produktu pro jeho úspěšnost?

Správnost softwarového produktu nepostačuje, protože správnost je jenom jednou z vlastností softwarového produktu. Dalšími důležitými vlastnostmi softwaru jsou také čas dodání a náklady vynaložené na jeho vývoj.

8.5 Je správnost softwarového produktu nezbytná pro jeho úspěšnost?

Správnost softwarového produktu nezbytná není a navíc absolutní správnost je prakticky nedosažitelná. Každý trochu složitější software má řadu nedostatků a přesto jej používáme.

8.6 Jaký je rozdíl mezi statickým a dynamickým ověřováním?

Statické ověřování nevyžaduje spustitelný program a lze ho proto provádět v libovolné etapě životního cyklu softwaru. Dynamické ověřování naopak vyžaduje běh

programu a je proto možné až tehdy, kdy je k dispozici spustitelná verze programu nebo alespoň jeho prototyp.

8.7 Jaký je vztah mezi dynamickým ověřováním a testováním?

Testování je pouze jiný název pro dynamické ověřování. Obvykle se jeden z těchto výrazů vybere podle souvislostí.

8.8 Co je cílem testování?

Cílem testování je odhalit chyby v softwaru.

8.9 Jakým způsobem se při testování odhalení chyb dosahuje?

Při testování se na vstup systému přivedou testovací údaje a porovnají se výsledky s očekávanými hodnotami. Pokud se získané výsledky liší od očekávaných (resp. správných), byla objevena chyba.

8.10 Lze testováním odhalit všechny chyby?

Obvykle ne, protože množina všech možných testovacích vstupů je příliš rozsáhlá nebo ani není konečná.

8.11 K čemu slouží testovací kritérium?

Testovací kritérium stanovuje podmínky, které musí množina testovacích vstupů splňovat. Účelem testovacího kritéria je vybrat přijatelně velkou množinu testovacích vstupů, pomocí které však lze chyby ještě odhalit.

8.12 Co je to metoda černé skříňky?

Termínem *metoda černé skříňky* se označuje funkcionální testování, při kterém se při výběru testovacích vstupů vychází pouze ze specifikace softwaru a neuvažuje se jeho vnitřní struktura.

8.13 Co je to metoda bílé skříňky?

Termínem *metoda bílé skříňky* se označuje strukturální testování, při kterém při výběru testovacích vstupů vychází z vnitřní struktury softwaru.

8.14 Kdo provádí akceptační testování?

Akceptační testování provádí uživatel (zákazník) na reálných datech.

8.15 Jaký je rozdíl mezi alfa a beta testováním?

Alfa testování provádí vybraní uživatelé generického softwaru v sídle vývojového týmu, zatímco beta testování provádí uživatelé u sebe.

9 Řízení softwarových projektů

9.1 Co je to projekt?

Projekt je časově ohraničené úsilí, které se vyvíjí s cílem vytvořit jedinečný výsledek (např. výrobek nebo službu).

9.2 Jaké jsou čtyři proměnné, které můžeme u softwarového projektu ovlivňovat?

Cena, čas, kvalita a rozsah.

9.3 Pracuje efektivněji menší nebo větší vývojový tým? A proč?

Efektivněji pracuje menší tým, protože nepotřebuje tolik času na vnitřní komunikaci.

9.4 Kdy a proč se provádí studie vhodnosti?

Studie vhodnosti bývá součástí inicializace projektu a jejím cílem je určit, zda je rozumné se do projektu pouštět (tj. zda očekávané přínosy projektu jsou větší než předpokládané náklady a zda rizika projektu nejsou příliš vysoká).

9.5 Jaký je vztah mezi počtem chyb v softwaru a jeho kvalitou?

Součástí kvality softwaru je i jeho cena. Zcela bezchybný software by byl příliš drahý, stejně tak příliš mnoho chyb v softwaru není přijatelné (každá chyba něco stojí). Optimální je proto kompromis – odstraňovat chyby, dokud je jejich odstranění levnější než jejich následky. Vztah mezi kvalitou a počtem chyb ilustruje obrázek 9.3 na straně 92.

9.6 Proč se zavádí normy pro řízení kvality do vývoje softwaru?

Zvýšení kvality softwarového produktu šetří náklady v pozdějších fázích projektu (používání a údržba). Zvýšení kvality však vyžaduje vyšší náklady v počátečních fázích projektu. Normy pro řízení kvality (např. ISO 9000) mohou být zavedeny i z marketingových důvodů.

9.7 Jaký je rozdíl mezi přímým a nepřímým měřením?

Při přímém měření získáváme hodnotu atributu jeho přímým sledováním (např. počet řádků programu). Při nepřímém měření hodnotu odvodíme z jiných atributů (např. chybovost = počet chyb / počet řádků).

9.8 Jaká metrika se používá pro měření spolehlivosti softwarových systémů?

Pro měření spolehlivosti se používá střední doba mezi výpadky systému (MTBF – Mean Time Between Failures).

9.9 Měří se dostupnost přímo nebo nepřímo?

Dostupnost (pravděpodobnost, že systém v daném čase pracuje správně) se měří nepřímo jako podíl střední doby do dalšího výpadku a střední doby mezi výpadky.