Objektové modelování a UML v praxi 2000

***Motto knihy:* Dělat věci jednoduchými není vůbec jednoduché**

© **Ilja Kraval**, vydáno leden roku 2001, revidováno červen 2001

kontakt: http://www.objects.cz/ , mailto:objects@objects.cz

Licenční ujednání

- Toto autorské dílo je vytvořeno v elektronické podobě. Jeho šíření, distribuce a pozměňování podléhá autorskému zákonu.
- Tento dokument je vydán jako licencovaný dokument a kromě autora díla RNDr. Ilji Kravala není žádnému jinému subjektu poskytnuto právo na šíření, pozměňování a distribuci tohoto díla.
- Tento dokument je vydán jako single licence. Oprávnění ke čtení a studiu má pouze majitel licence a nesmí být zpřístupněn nikomu jinému, než majiteli licence.
- Majitelství single licence neopravňuje majitele licence k dalšímu šíření tohoto autorského díla, k jeho pozměňování anebo k další distribuci.
- Autor neodpovídá za případné změny učiněné jinou osobou v tomto dokumentu.

Tato e-kniha byla vytvořena pomocí nástrojů WORD a VISIO od firmy Microsoft.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Úvodní slovo autora

Tato elektronická kniha navazuje na předešlé papírové knihy typu "paperback" vydané nakladatelstvím Computer Press v r. 1998 a které byly věnovány problematice objektově orientovaného programování a komponentní technologie. Kromě toho také tato e-kniha navazuje na skripta o UML publikované na *Serveru objektových technologií* v roce 1999 (http://www.objects.cz/).

Předešlé „paperback“ knihy byly nazvány „Základy OOP za pomoci MS Visual Basic 5.0“ a „Základy komponentní technologie COM“ a jsou stále v nabídce v Internetového knihkupectví Vltava (http://www.cpress.cz/ , Vltava). Obě dvě knihy a také skripta se setkaly s velmi příznivým ohlasem. Podle spousty obdržených mailů lze také konstatovat, že se také rozšířil počet příznivců OOP, UML a komponentní technologie v naší republice.

Poznámka: V návaznosti na uvedené knihy a skripta vyšla ještě další skripta o XML, VB, ASP atd., která si můžete objednat na http://www.objects.cz/

Od vydání knih v Computer Pressu uplynuly již dva roky a od prvních skript o UML více než rok a nutno poznamenat, že se za tu dobu opět „něco změnilo“. Především se svět tvorby SW posunul opět o velký krok kupředu, což lze považovat za skutečnost již samu o sobě jako dobrý důvod pro napsání nové knihy.

Na straně druhé předešlé dvě knihy jsem tvořil jako zaměstnanec „jedné jediné“ softwarové firmy. Neměl jsem tehdy žádné nebo minimální zkušeností s tím, jak zavádět OOP a UML ve firmách. Poslední dva roky se věnuji konzultační činnosti při zavádění OOP a UML u mnoha softwarových firem pod hlavičkou své vlastní firmy. Současně spolupracuji jako externí spolupracovník při tvorbě analytických modelů informačních systémů tvořených v UML u těch firem, které již na OOP a UML přešly.

Takto jsem získal kontakt s desítkami lidí, kteří potřebují poradit v oblasti OOP, UML a COM. Při těchto konzultacích a školeních jsem postupně zjišťoval, jak vlastně přemýšlejí ti, kteří se učí OOP a UML, jakých chyb se dopouštějí pracovníci, kteří začínají s OOP a UML. Zjistil jsem, kde a jak se tvoří omyly a mýty o OOP a UML, ověřil jsem si, jak postupovat při zavádění OOP a naučil se, jak vysvětlovat a kde začínat s OOP a UML. Současně jsem nabyl určitých zkušeností, jak co nejrychleji tvořit informační systémy pomocí OOP a UML.

Poznámka: A také jsem se setkal s nezdary se zavedením OOP a UML ve firmách (naštěstí ojediněle) a z těch jsem se poučil nejvíce...

Zajímavé na všech konzultacích ve firmách, které zaváděly OOP a UML bylo to, že na začátku ve všech firmách panovalo velké rozladění z předešlého způsobu tvorby SW bez objektů. Pochopitelně toto rozladění doprovázela obrovská touha po změně technologie tvorby SW k „něčemu lepšímu, systematičtějšímu, průhlednějšímu a stabilnějšímu“. Pracovníci většinou snili o lepším způsobu tvorby SW, než jaký se doposud ve firmě používal. Při seznámení se s OOP a UML najednou s překvapením zjistili, že tento přístup je právě splněním onoho snu. A o tom je mimo jiné i tato kniha.

Onen trýznivý stav ve firmách před zaváděním OOP a UML je možné charakterizovat stručně těmito slovy:

- tvoříme software těžkopádně
- vyrábíme SW s chybami
- výsledný informační systém je nestabilní

- náklady spojené s dodatečnou údržbou jsou obrovské
- obtížně a velmi nepříjemně dokumentujeme výsledky práce
- ani nevíme, jak zapsat analýzu a design, (kód ten snad ano...)
- pokud dojde ke změnám v zadání, obtížně tyto změny zavádíme anebo nejsme schopni tyto změny uhlídat
- tvoříme SW bez žádného „chytrého řízení ve firmě“
- týmy pracují bez koncepce, tj. pracovníci v týmech „bojují každý sám za sebe“ a ve větších firmách toto platí i mezi týmy
- stále opakujeme práci s minimálním re-use
- a jiné negativní projevy, vedoucí ke špatným vztahům mezi pracovníky a tedy k znechucení pracovníků...

Všechny tyto symptomy a ještě mnoho dalších „negativních“ se zahrnují v tvorbě SW pod jeden stručný pojem: **Nekvalita tvorby SW**.

Zní to jako v pohádce o zázračném receptu, ale opravdu přechod na objektové programování a poté na objektové modelování pomocí UML (a následně také přechod na objektové kódování), to je řeč o přechodu firmy na mnohem vyšší kvalitu tvorby softwaru se všemi důsledky z toho plynoucími: Z hlediska zaměstnanců se jedná o příjemnější způsob tvorby SW a z hlediska firmy se jedná o celkové výrazné snížení nákladů díky zvýšení kvality SW. Důsledkem přechodu na UML je rychlejší a kvalitnější tvorba SW a velmi výrazné snížení nákladů na údržbu po implementaci softwaru.

Poznámka: V jednom zahraničním článku jsem si všiml jednoho zajímavého postřehu z amerického prostředí SW firem, že „špatná SW firma se pozná podle spousty obalů od picc v rohu místnosti“, zřejmá to narážka na večerní a noční práci programátorů...

Nutno podotknout, že i když ve valné většině případů byla moje konzultační činnost ve firmách završena kladným výsledkem, přece jen (a i to se může stát) v několika málo případech se požadovaný výsledek, tj. přechod na OOP, nedostavil. Firma ani po konzultacích a velké snaze „nepřekročila svůj Rubikon strukturálního přístupu“ a nepřiblížila se k OOP a komponentám, natož k objektovému modelování a setrvala v původním „zastaralém“ stavu. Rozborem těchto několika málo neúspěšných projektů jsem dospěl k určitým důležitým závěrům, mimochodem podobným jako jsou všeobecná doporučení v oblasti řízení projektů, které mne vedly k vyvarování se určitých postupů při zavádění OOP a UML ve firmách. Takže i záporné výsledky se staly velkým poučením dokonce větším, než kladné výsledky. Tyto projekty se tak zařadily mezi nejpoučnější zdroje informací o tom, jakých chyb se při přechodu na OOP a UML vyvarovat a o tom je také tato e-kniha.

Na druhé straně jsem v „objektově vyspělejších“ firmách působil jako externí spolupracovník a podílel se přímo na tvorbě modelů v UML několika informačních systémů také jako analytik. Dá se říci, že v rámci těchto konzultací jsem se podílel na tvorbě spousty modelů UML, a měl jsem tedy možnost pracovat s objektovými modely chápanými jako modely horší a modely lepší. Současně se mi ukázalo, jak se vyvarovat nejčastějších chyb a omylů, a tedy jak se vyhnout složitým a neschůdným řešením a to hned na začátku návrhu systému.

Po určité době strávených konzultační činností jsem uvážil, že zkušenosti a poznatky by bylo vhodné zúročit v nějakém souhrnném dokumentu. Proto se vám dostává nyní do rukou první vydání knihy v elektronické podobě věnované nejmodernějším metodám tvorby informačních systémů v objektovém prostředí pomocí komponent s podporou objektového modelování za podpory syntaxe UML.

Tato kniha je napsána z velmi praktického pohledu. Nazval jsem tuto knihu *„Objektové modelování a UML v praxi 2000“* proto, že převážná část dokumentu je vskutku věnována praktickému použití metod objektového modelování a použití UML . Jsou v ní zodpovězeny všechny nejčastější otázky,

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

vyvráceny nejčastější omyly a je zde upozorněno na největší hrubé chyby při zvládání objektového přístupu, stejně tak při používání komponent a při zavádění objektového modelování pomocí UML.

Samozřejmě nevyhrazuji si právo na tvrzení, že vše co napíši, bude takříkajíc „stoprocentně správné“. To, co předkládám, je souhrn praktických poznatků (podložených několika málo teoretickými tvrzeními) z konzultací a tvorby SW ve firmách, které na OOP a na objektové modelování přecházely.

Doufám, že vám, čtenáři bude tato kniha číst plynule a příjemně a že po jejím přečtení se také vy zařadíte mezi ještě větší stoupence objektového přístupu a mezi praktické uživatele UML.

Komu je určena tato kniha

Tato kniha je určena především všem pracovníkům činným v oblasti tvorby SW, tj. analytikům, vedoucím projektů, programátorům, testerům atd. ale i uživatelům SW v podnicích, které spolupracují s firmami tvořící software objektově orientovaným přístupem.

Zvládnutí této knihy podá čtenáři ucelený obraz o objektově orientovaném přístupu při tvorbě SW, o modelovacím jazyce UML a o tvorbě SW pomocí komponent.

Poznámka:

V žádném případě tato kniha neslouží jako specifikace jazyka UML a popisuje hlavně praktické zkušenosti s objektovým modelováním a použitím UML. Syntaxe UML je zde mnohde využívána, ale není všude přesně specifikována podle poslední verze, kterou je 1.3. V mnoha částech této knihy proto není dodržena stoprocentně syntaxe jazyka UML verze 1.3. Protože syntaxe jazyka UML 1.3 je poměrně rozsáhlou (a zajímavou) oblastí, připravuje se publikace zaměřená přímo na specifikaci UML 1.3. Kniha bude mít název ***„Stručná specifikace UML 1.3 s praktickým výkladem“***

Co je a co není UML

UML značí zkratku pro *Unified Modeling Language*, což by se dalo přeložit jako *„Unifikovaný modelovací jazyk“*. Tento název již sám o sobě napovídá, že UML patří mezi jazyky, tedy má svou vlastní syntaxi. Jinak řečeno UML lze chápat jak jako dohodu nad vyjadřovacími prostředky pro objektové modelování.

Pro programátora není samozřejmě pojem „jazyk“ nic nového a patří k nejpodstatnějším pojmům z jeho branže. Přece jen při výběru působnosti programátora má podstatný význam právě znalost určitého programovacího jazyka.

Avšak na rozdíl od programovacího jazyka není UML jazykem nějakého konkrétního programovacího prostředí (C++, Visual Basic apod.). Pomocí UML lze mimo jiné vytvářet modely informačních systémů a programových produktů univerzálně pouze s jedním omezením, informační systém je tvořen pomocí objektově orientovaného přístupu. UML je tedy obecnějším jazykem než jazyky programovací, protože umožňuje zapsat model jakéhokoliv informačního systému, který je tvořen objektově. Z tohoto hlediska je znalost syntaxe UML chápána jako obecnější, než znalost nějakého konkrétního programovacího jazyka. Znalost UML lze přirovnat k esperantu v tvorbě informačních systémů.

Poznámka: Způsob popisu aplikace pomocí UML je tedy z hlediska programování jazykově nezávislý, ale to neznamená, že výsledný model zapsaný v UML bude nezávislým na prostředí, ve kterém bude implementován. Ve výsledném modelu informačního systému se ve fázi designu v modelu vyskytují objekty a třídy platné pouze v daném prostředí (např. PictureBox, Label, atd. z VB 6.0 apod.).

UML se však používá nejenom pro modelování v oblasti tvorby softwaru. Protože základním pojetím UML je objektově orientovaný přístup, lze tento modelovací jazyk použít všude tam, kde lze aplikovat pohled na problematiku, což nemusí být pouze tvorba SW. Podotkněme, že takovýchto oblastí

možných použití UML může být mnoho. Jako příklad mohu uvést použití UML při modelování procesů v podnicích, v bankách, úřadech apod., tj. v problémové oblasti logistiky firem a organizací. Při popisu a poté optimalizaci chodu podniků a institucí lze použít objektově orientovaný přístup a tedy následně logicky lze použít pro tvorbu modelů těchto podniků a institucí jazyk UML. Objekty zde však již nevyjadřují nějaké části informačního systému, ale vyjadřují přímo nějaké objekty v podniku resp. instituci (objekty „oddělení“, „pracovník“, „vedoucí“, „zpráva“, „oběžník“ atd.). Pomocí UML se modelují nejenom existující vztahy v podniku, ale také vztahy v podniku zatím neexistující, tedy vztahy možné a žádoucí, což je synonymum pro optimalizaci podniku pomocí syntaxe UML.

Avšak hlavním cílem zvládnutí UML je naučit se modelovat informační systémy napsané objektově standardními prostředky, což má za následek, že pokud UML použijeme, potom našemu vyjádření modelu porozumí každý, kdo zná UML.

Na straně druhé je třeba zdůraznit, že UML jako jazyk v žádném případě není metodikou resp. návodem pro tvorbu SW. V syntaxi UML tedy nejsou uváděny postupy jak SW tvořit, nebo zásady jak řídit projekty, jak zavádět metodiky ve firmě. Protože však tato oblast (oblast řízení projektů) je velmi důležitá, tvůrci UML doporučují určité zásady jako doplněk k UML (o těchto zásadách bude řeč v kapitole o řízení projektů). Přitom se však zdůrazňuje, že UML je pouze jazykem a tedy jenom dohodu nad syntaxí zápisu modelů a „nic víc“.

Z uvedené definice že „UML je pouze jazyk a nic víc“ by se mohlo vcelku logicky vyvodit, že samotné zavedení UML tedy samo o sobě neřeší problematiku řízení projektů ve firmě resp. zavádění metodik, pracovních postupů apod. To je sice pravda, ale problém řízení projektů je pochopitelně mnohem složitější, než pouhé používání UML. Avšak už samotné zavedení UML ve firmě bez dalších zásahů do řízení může vést k pozitivním změnám v řízení projektů, protože správné použití syntaxe UML nutí tvůrce SW postupovat určitými kroky vyžadovanými syntaxí UML. Obecně totiž platí, že již používání nového moderního nástroje (což UML určitě je) vede automaticky k zavedení určitých lepších pracovních postupů, než jaké vyžadovaly předešlé nástroje.

Pokud se ve firmě nezavedou žádné metodiky, neexistují tím pádem samozřejmě žádné závazné dokumenty řízení. Avšak již sama nutnost použití notace UML a následnost tvorby modelů v UML si určitý metodický postup při tvorbě informačního systému vyžaduje sama o sobě. Tedy pouhá povinnost zavedení notace UML může již sama o sobě kvalitativně změnit chod ve firmě, aniž by byly zaváděny formální závazné postupy (i když vřele doporučuji zavést i závazné postupy). Problematice řízení projektů při tvorbě SW v objektovém prostředí bude věnována samostatná kapitola.

Zkušenosti mne vedly k zajímavému praktickému doporučení: Zaveďte ve firmě UML a nutnost tvorby metodik vyplyne na povrch sama od sebe, stačí tuto nutnost pouze správně podchytit.

Jak bylo řečeno, základním kriteriem pro použití UML jako modelovacího jazyka je objektově orientovaný přístup. V další kapitole jsou vysvětleny základy OOP spolu s nejčastějšími kladenými otázkami při použití tohoto moderního přístupu pro tvorbu SW.

Re-use v informačních systémech

Pojmu re-use věnujeme celou úvodní samostatnou kapitolu, protože patří k základním pojmům v programování. Osobně považuji za jeden z nejpříznivějších důsledků zavedení OOP a UML maximálně a efektivně pojatý re-use, který nemá ve strukturálním programování obdobu.

Co se vlastně chápe pod pojmem re-use? Většinou se re-use spojuje pouze s re-usem zdrojového kódu. Zdrojový kód se namísto opakování „re-usne“, tj. opětovně použije. Avšak re-use je třeba chápat jako obecnější přístup než pouze pro znovu použitelnost zdrojového kódu. Re-use budeme chápat jako něco, co obecně souvisí s opakováním se ve vývoji „něčeho“ a může to být „cokoliv“. Základní smysl re-use je „zabránit opakování“, tj. zabránit zdvojení informace a tím také zdvojení práce.

Obecně re-use patří k základním poznatkům teorie programování. Platí základní obecné pravidlo při zavedení libovolného re-use:

Pokud se vyžaduje neopakování něčeho a shoda pojmů něčeho, vede to osamostatnění tohoto určitého pojmu, resp. k osamostatnění části systému a vytvoření nové samostatné entity a k vytvoření odkazů na tento pojem.

Uveďme si klasické příklady:

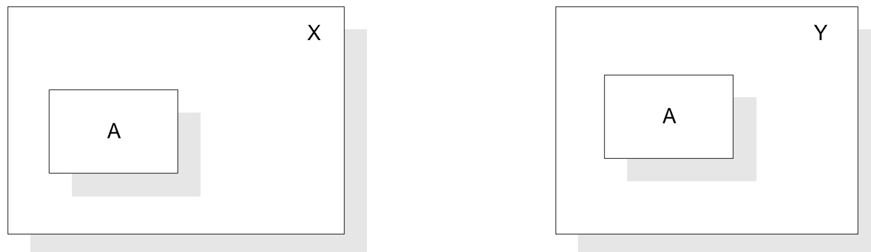
- pokud se opakuje kód procedury, můžeme buď přenášet kód pomocí schránky (clipboardem), což není v žádném případě doporučováno, nebo zavedeme „nový mechanismus“ s voláním procedury s novým názvem procedury, která je tak sdílena. Do ní umístíme opakující se kód procedury. Duplicitu kódu odstraníme „jednou funkcí volanou z různých míst". Ve strukturálním programování tedy deklarujeme jednu společnou funkci pro všechny případy jejího volání. Takováto funkce je definována pouze jednou, definice se vyskytuje pouze na jednom místě a její definice se pokaždé neopakuje.
- jiný příklad se vyskytuje v teorii datových modelů. Pokud se ve dvou tabulkách vyskytnou stejné sloupce stejného analytického významu, jedná se opět o duplicitu kódu, přesněji řečeno o redundanci dat, kterou bychom museli programem ošetřovat. Vyžadujeme-li shodu těchto sloupců, normalizační proces nad databází provede „vyvedení“ sloupců ven do samostatné tabulky a zpětné provázání společných údajů do obou původních tabulek. Vznikl tak nový pojem, nová entita, zde speciálně nová tabulka s novým analytickým významem tabulky. Tento proces tvorby entit souvisí s normalizací databáze.
- dalším příkladem je zavedení třídy v OOP (viz dále v samostatné kapitole). Při definici objektů se mohou definice opakovat, což vede k duplicitě kódu. Z toho důvodu se zavádí nový pojem třída, která spojuje všechny definice objektů stejných vlastností. Zpětně je třída jako kopyto definicí provázána k objektům tak, že daný objekt patří do určité třídy a je z ní generován, resp. pomocí ní definován. Pomocí třídy je dosaženo shody definicí objektů stejných vlastností.
- jiným příkladem je zavedení pojmu dědění (podrobněji viz další kapitoly): Po zjištění, že mohou existovat stejné vlastnosti ve dvou nebo více třídách, zavede se dědění z třídy předka. Všechny společné vlastnosti se spojují do společné jedné třídy. Kód je sdílen pomocí dědění.
- ...tak bychom mohli pokračovat dále.

Postup re-use je tedy vždy stejný: Najdi „*to, co se opakuje*“, vytrhni „*to, co se opakuje*“ do samostatné entity, a provaž tuto novou entitu zpět do místa, odkud byla vytržena. Vznikne tak sdílení dané entity.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Tuto operaci re-use znázorněme graficky:

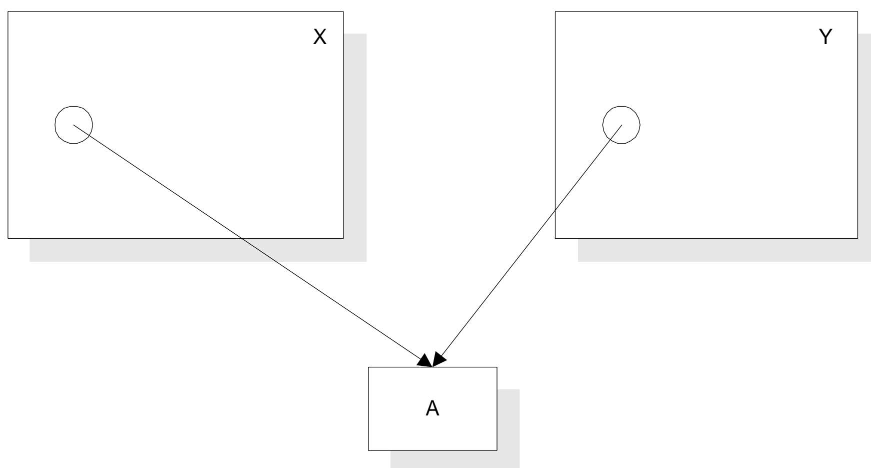
Nechť se v systému opakuje ve dvou částech informace, což můžeme znázornit symbolicky takto:



obrázek 1 : Ve dvou částech systému se opakuje informace A

Provedeme operaci re-use: Vytrhněme informaci A jak z X, tak z Y, osamostatněme ji, a poté ji znovu provážeme do bodů, odkud byla vytržena. Je třeba podotknout, že chceme-li tento postup „vytržení“ a „provázání zpět“ provést, musí být zavedena nějaká zvláštní operace tohoto provázání (přesněji řečeno interakce) mezi prvky typů entit A, X, a Y. V předešlém výčtu pro použití re-use mezi třídami, mezi tabulkami, funkcemi, musí být zavedeny interakce dědění mezi třídami, join mezi tabulkami, volání mezi funkcemi apod.

Operací vytržení a zpětného provázání vznikne nový stav „po provedení re-use“ znázorněný na dalším obrázku:



obrázek 2 : Po provedení re-use je A osamostatněno a z bodů v X a Y je odkazováno na A

Otázkou je, proč dbát na to, aby náš systém co nejvíce používal re-use? Použití re-use má tři základní příznivé aspekty:

1. **Neopakování se prací při samotném vývoji**. Je vcelku pochopitelné, že zavedení re-use (a to nejenom ve vlastní práci zaměstnance, ale v celé firmě), má za následek, že „se neopakují již provedené práce“, což vede pochopitelně k urychlení vývoje a tím snížení nákladů. Je někdy až překvapivé, jak je tato jednoduchá poučka v některých firmách s lehkým srdcem porušována.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Přitom re-use patří k prvním zásadním krokům, které drasticky snižují náklady, zvyšují kvalitu a urychlují vývoj. Většinou je pro vedení firmy přímo šokující, jakých překvapivě kladných výsledků se dá dosáhnout pomocí re-use.

2. **Neopakování prací při opravách a změnách v systému**. Je zřejmé, že pokud se některé části systému opakují (například kód), potom při provedené změně musíme samozřejmě projít všechny opakující se části a tyto části pokaždé znovu opravovat. Údržba systému se potom stává opakujícími se stejnými úkony oprav téhož na různých bodech.
3. **Maximální zvýšení přehlednosti systému, dosažení čistoty a elegance systému, docílení srozumitelného a jasného rozvrstvení systému**. Nepoužití re-use mimo jiné vede ke zvýšení neuspořádanosti systému. Například v předešlém odstavci se při změnách a nepoužití re-use nejenom tyto změny opakují, ale musí se vést evidence "kde všude tyto opakující se změny musíme provést“. V některých případech se při ztrátě této evidence postupuje metodou „změna se týká té části procesu, která po změně spadne do erroru“. Údržba systému s maximálním re-use je mnohem průhlednější. Pokud se má něco opravit, je zřejmé, kde tato se tato oprava nachází. Naopak při ztrátě re-use se údržba systému stává dost napínavou detektivkou. Osobně ztrátu re-use považuji za automatickou ztrátu elegance a přehlednosti systému. Software má také svůj „umělecký dojem“ ve škále elegance a ten je mimo jiné vyjádřen úrovní použitého re-use.

V OOP při správném použití UML se dosahuje maximálního možného re-use již na úrovni prvních modelů analýzy (tj. již v Use Case modelu). Celá objektová technologie je prodchnuta neustálou snahou o maximální re-use, protože re-use je základním symptomem „čistoty“ vytvořeného softwaru. Existují optimalizační metody, které re-use narušují, ale tyto postupy se chápou jako dodatečné úpravy (tj. jako optimalizační kroky).

Poznámka 1: Klasickým příkladem narušení čistoty re-use je zavedení indexu u tabulky. Index není nic jiného, než opakující se „informace v tabulce již obsažená“, avšak uspořádána tak, aby práce s tabulkou byla rychlejší.

Poznámka 2: Existuje ještě jedna situace vedoucí k úmyslné ztrátě re-use a tou je snaha o následné „znepřehlednění kódu“ pro znesnadnění práce hackera. Například kód funkce se úmyslně opakuje pro nějakou verifikaci, tj. nevolá se centrálně jedna zavedená funkce, aby hacker musel provést také všechny změny ve všech opakováních kódu funkce. V každém případě se tím samozřejmě pouze ztíží hackování (zamezit nelze nikdy). Při hackování se vždy jedná o hru s pravidly vnutit hackerovi takovou pozici, kdy se mu z hlediska vlastních nákladů (například ceny jeho času apod.) neoplatí hackovat.

Problematice re-use jsme v této kapitole věnovali zvláštní pozornost proto, že se tento pojem bude prolínat jako červená niť všemi kapitolami věnovaným objektovému modelování pomocí UML.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Základy OOP

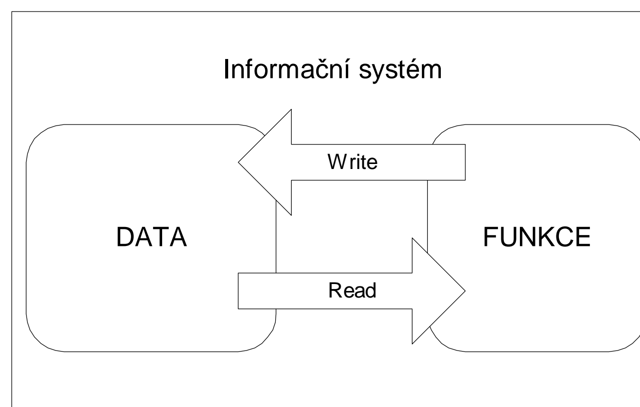
Zastaralý strukturální přístup k tvorbě SW

Velmi mnoho firem používá pro tvorbu SW zastaralý strukturální přístup. Vysvětleme si nejprve, čím je tento přístup charakteristický a jak se projevuje ve svých důsledcích.

V informačním systému tvořeném strukturálně se všechny prvky dají rozdělit do dvou základních oblastí: **DATA** a **FUNKCE**.

1. Oblast **DATA**. Do této oblasti spadají všechny zavedené proměnné (jako například proměnné typu `string`, `long`, `record`, `type` apod.), a také všechny údaje uložené na disku ve formátu dat, jako například soubor, file, tabulka, sloupec, řádek atd. Pod daty máme tedy na mysli jakoukoliv uchovanou („podrženou“) informaci buď pouze v paměti počítače (dočasnou proměnnou, která po vypnutí počítače mizí i se svým obsahem) anebo informaci uschovanou perzistentně, tj. perzistentní data (například na disku, na disketě, na pásce). Základní vlastností a tedy jediným „uměním“ dat je „podržet“ informaci ať perzistentně anebo dočasně. Je zřejmé, že rozdělení na perzistentní a neperzistentní data není z našeho hlediska pojetí systému až tak důležité, jedná se o data buď uschovaná i po vypnutí systému anebo držená pouze v době běhu systému. Pokud by teoreticky systém nemohl být nikdy vypnut a nikdy nemohl „spadnout“ (což je samozřejmě tvrzení z oblasti sci-fi), a pokud bychom měli k dispozici neomezenou velikost operační paměti, potom by se požadavek na ukládání dat stal nadbytečným.
2. Oblast **FUNKCE**. Do této oblasti spadají prvky jako tzv. výkonné části informačního systému, které vykonávají nějakou činnost. Jsou nazývány programy, funkce, procedury, skripty atd. Prvky z této oblasti jsou tedy identifikovatelnými posloupnostmi činností informačního systému a jsou to ty prvky, které mění obsahy prvků z první oblasti – z oblasti dat. Funkce pracují s daty dvěma směry: Buď provádějí zápis do dat anebo čtení z dat.

Uvedené rozdělení ukazuje následující obrázek:



obrázek 3: Strukturálně “ne-objektově” pojatý informační systém

Pro pochopení strukturálního pojetí a jeho nedostatků je velmi důležité si uvědomit, že toto dělení je konečné a úplné. Ve strukturálním pojetí neexistuje nic mimo tyto dvě oblasti, tj. mimo data a funkce.

Má to jeden velmi nepříjemný důsledek, který vyplývá z té prosté skutečnosti, že náš okolní svět se skládá z objektů, tedy „realita je objektově orientovaná“. Pokud navrhuji informační systém pomocí strukturálního pojetí, musím navrhovat pouze funkce a data. To vede k nutnosti převést objektový model reálného světa do úplně odlišné řeči funkcí a dat v programování. Existuje tedy transformace ve smyslu:

problém k řešení (který je sám o sobě objektově orientovaný) převedeme do funkcí a dat

Jinak řečeno pokud máme vytvořit informační systém strukturálně, potom musíme ryze objektově orientovanou realitu okolního světa přetransformovat nějakým myšlenkovým procesem do dvou oblastí, do funkcí a do dat. Tento postup považujeme za „programátorské řešení problému“. Proces této transformace je mnohdy dost náročný a je chápán jako analýza systému ve strukturálním pojetí.

Je to však transformace z objektů do funkcí a dat v systému nutně provedená ***navíc a z hlediska pojetí OOP úplně zbytečně***. Ve strukturálním pojetí se jí však nevyhneme, protože je vyžadována samou podstatou strukturálního pojetí, kde nic jiného než funkce a data neexistují. Nutnost převádět objekty reality do funkcí a dat činí analýzu a návrh systému ve strukturálním pojetí mnohem složitější než v OOP přístupu, protože se vyžaduje přetavit jinak běžné objekty reality do nějakých funkcí a do nějakých dat. Základní problém spočívá v tom, že samo naše uvažování je v podstatě objektově orientované a realitu vidíme objektově.

Zajímavé na tom všem je ta skutečnost, že programátor dlouho pracující strukturálně považuje uvedený postup „navrhni z objektů funkce a data“ za natolik běžný a za natolik zažitý postup, že si dotyčný ani neuvědomuje, že tento pohled je vlastně velmi zvláštní. Okolní svět, který máme za úkol v informačním sytému nějak postihnout, se totiž v žádném případě nechová tak, že by byl tvořen dvěma okruhy, okruhy DATA a FUNKCE. Realita je totiž jiná než „funkce a data“ (je objektová).

Poznámka: V knize o COM uvádíme velmi ilustrativní příklad na strukturální pojetí. Představme si, že uvažujeme strukturálně a že máme vyřešit následující situaci: Jdeme po ulici a potřebujeme zjistit, kolik je hodin. Ve strukturálně pojatém reálném světě bychom zavedli funkci například `GETTIME` vracející aktuální čas. Ve chvíli požadavku bychom „někam nahoru zavolali“ funkci `GETTIME` a ta nám vrátí požadovaný čas. V normálním světě (který je objektově orientovaný) se podíváme na hodinky, které jsou objektem kompetentním za vydání času, a tento čas získáme dotazem od nich. Všimněme si, že v tomto případě je v systému někdo kompetentní za určitou funkcionalitu. Ve strukturálním pojetí je tímto kompetentním pouze sám systém a jeho funkce. Celá systém se chová jako jeden velký objekt jehož metodami jsou funkce.

Strukturální přístup s sebou přináší jednu velkou zátěž pro přerod u dlouholetých „strukturálních programátorů“. Tato zátěž není ničím jiným než uvedenou naučenou oklikou transformace od objektů k datům a funkcím. Při tvorbě informačních systémů ve strukturálním pojetí musí být pracovníci nejenom znalí, ale i zkušení v návrhu funkcí a dat. Z hlediska programátorů se tato zkušenost tvorby funkcí a dat chápe jako schopnost tvorby informačního systému, která není „normálnímu smrtelníkovi“ známa. V této schopnosti navrhovat funkce a data mnohdy programátor-analytik chápe své poslání a svou profesní odlišnost od ostatních pracovníků z jiných profesí. „Správný a zkušený“ strukturální programátor při pohledu na jednoduchý systém objektové reality světa ihned vidí funkce a data a je schopen systém strukturálně navrhnout, jinak řečeno tato „oklika“ je pro něj hračkou (což pro neznalého neplatí). Programátor-analytik vidí tuto schopnost navrhnout funkce a data jako náplň své práce, protože ztotožňuje návrh informačního systému s touto transformací.

Problém přerodu pracovníků od strukturálního programování na OOP spočívá ve zvyku vidět program jako funkce a data a „nijak jinak“. Vzniká tak u pracovníků dvojí pohled na daný problém, jeden „jak to je“ a druhý „jak vypadá program“. Existuje jeden vyjadřovací prostředek pro realitu (neformálně objektový) a druhý je vyjadřovací prostředek pro program (formální jako funkce a data). Pokud je

pracovník obdařen díky zkušenostem možností rychlého a automatického pohledu na model informačního systému ve tvaru funkce-data, tak tato výhoda se paradoxně stává velkou překážkou při přechodu na OOP, protože tohoto pohledu se programátor musí bezpodmínečně zbavit.

Příklad

Představme si programátora, který chce uživateli napsat jednoduchý program evidující firmy a faktury a nic víc. Rozhovor vypadá nějak takto:

Programátor: „Takže budou tam firmy... a co se eviduje u firem?“

Uživatel: „IČO, DIČ.. Název, adresa, bankovní spojení.“

Programátor: „A faktury?“

Uživatel: „Odběratele, to je jedna z těch firem, pak řádky faktury, tam bude dodané zboží, množství...“

atd.

Všimněte si, že uživatel a programátor spolu hovoří v ***pojmech*** a nikoliv v datech a funkcích. To, co se požaduje evidovat popisují ***vlastními slovy problémové domény***. Problém je v tom, že:

- ve strukturálním pojetí se nyní musí provést transformace z těchto pojmů do funkcí a dat
- v UML se tyto pojmy přímo zrcadlí určitým jednoduchým procesem do objektů a poté do tříd

Zkušený objektový programátor v uvedených větách za pojmy již „vidí objekty v informačním systému“, zkušený strukturální programátor „vidí data a funkce“.

V mnoha případech jsem se setkal s jednou velkou překážkou při tvorbě informačních systémů, kdy bylo vyžadováno vytvořit objektově orientované modely v UML a na straně poskytovatele analytických poznatků, tj. v roli uživatele v předešlém rozhovoru, se nacházela „silná“ osobnost velmi dobře znalá strukturálního programování. Nastala velmi nepříjemná situace: Paradoxně tato osoba nebyla schopna popsat daný problém v jednoduchých pojmech, jak to vyžaduje UML. Důvod je zřejmý – poskytovatel informace automaticky převáděl své jinak normální obyčejné a srozumitelné poznatky vyjádřené „selským rozumem“ ihned do řeči strukturálního programování, protože je zvyklý takto uvažovat. Rozhovor pak vypadá jako příklad nějak takto:

Analytik OOP: „Takže faktura obsahuje co?“

Uživatel – strukturální programátor: „Obsahuje číslo faktury a IČO firmy a dále obsahuje...“

Analytik OOP (skočí mu do řeči): „A proč má faktura IČO firmy?“

Uživatel – strukturální programátor: „Přes něj se dostane do firmy. To je klíč. Ale může tam být taky idečko firmy.“ (poznámka: Idečko je chápáno jako systémový identifikátor řádku v tabulce, Autoincrement, Identity apod.)

Analytik OOP: „Takže faktura vidí svou firmu... To je partner, dodavatel nebo odběratel, že?“

...jinde podobně:

Uživatel – strukturální programátor: „Řádek faktury obsahuje kód zboží...“

Analytik OOP (skočí mu do řeči): „A proč kód zboží?“

Uživatel – strukturální programátor: „Přes něj se dostane do objednaného zboží, co je na řádku. To je klíč. Ale může tam být taky idečko zboží.“

Za pozornost stojí jedna zajímavá skutečnost: Všimněte si, že uživatel označený jako „strukturální programátor“ v předešlém rozhovoru pendluje mezi dvěma vyjadřovacími prostředky – mezi strukturálními vyjadřovacími prostředky a mezi „pojmovými prostředky“, blízkými k objektovému

popisu, aniž by si to sám uvědomoval: Například použije větu: „...faktura obsahuje IČO firmy resp. idečko firmy a tím je zprostředkována vazba...“ je větou ze strukturálního pojetí. Naopak věta: „...se dostane do objednaného zboží“ patří již do OOP, protože „objednané zboží“ je již správným samostatným pojmem.

Důležitý závěr vyplývající z existence transformace do dat a funkcí ve strukturálním programování

Z uvedeného je patrný, jeden důležitý závěr: Pojetí informačních systémů je v objektově orientovaných systémech zapsaných pomocí UML mnohem bližší normálnímu lidskému uvažování, než ve strukturálním programování. Tato skutečnost je velmi důležitá zejména pro tvorbu analýzy, protože v UML se tímto získává vysoká analytická přehlednost v informačním systému (zrcadlení se skutečností), získá se transparence systému a v neposlední řadě také vysoká logická elegance navrženého systému.

Coad-Yourdonova škola modelování ve strukturálním programování

Také ve strukturálním pojetí se samozřejmě vyvíjely různé metody, jak postupně zdokonalovat zápisy modelů, tj. zápisu prvků modelů v oblasti dat a funkcí nad nimi pracujícími.

Určitým vyvrcholením těchto škol se stala Coad-Yourdonova metoda modelování, mimochodem chápána jako jeden z nejvyšších předstupňů objektového modelování. Stojí za zmínku, že sami tvůrci této metody ji po určité době opustili a stali se velkými zastánci objektového modelování.

Základními prvky modelů Coad-Yourdonovy metody strukturálního přístupu jsou tři typy diagramů:

- *Entity RelationShips Diagram* (*ERD)*,
- *Data Flow Diagram (DFD)*
- *Data Structure Diagram (DSD)*

Diagramy ERD

Diagramy ERD zavádějí tzv. entity a vztahy mezi nimi. Je zřejmé, že protože zůstáváme na poli strukturálním, tak pod entitami je zde myšlen nějaký datový útvar (nikoliv objekt).

Zavedení entity v ERD znamená zavedení nějakého datového útvaru (skupiny dat), která může mít vztah k jiné skupině dat. Vyjádření těchto vztahů se svými vlastnostmi (například kardinalita, tj. násobnost) udává vztah mezi těmito entitami, tj. vztah mezi daty.

V prvé řadě je třeba podotknout, že existuje velmi mnoho způsobů, jak realizovat vztah mezi daty. Existují souborové systémy, existují stromové databáze a také nejrozšířenější relační databáze. V relační databázi je vztah mezi dvěma skupinami dat realizován na základě shody hodnot dat, například shodou klíče. Mnohdy se zapomíná díky časté a neustálé práci s relačními databázemi, že v jiných typech databází je tento vztah realizován jinak a dokonce pro některé případy mnohdy výhodněji a optimálněji, než v relační databázi.

Příklad

Ve jedné firmě, kde jsem měl možnost několik roků pracovat jako zaměstnanec, se vyvinul vcelku zajímavý bankovní informační systém, ve kterém se přistupovalo k datům přes „vlastnoručně“ tvořené soubory (v Pascalu typ record, binární čtení, vyvážené stromy apod.). Protože se jednalo o řešení „detailně šité na míru“, ve kterém se použily binární stromy, optimalizační techniky v assembleru apod., systém pracoval vcelku dost „neobvykle rychle“. Měl však jiná omezení, která vedla ke změně technologie a současně k zavedení relační databáze. Použití relační databáze však vyžadovalo další

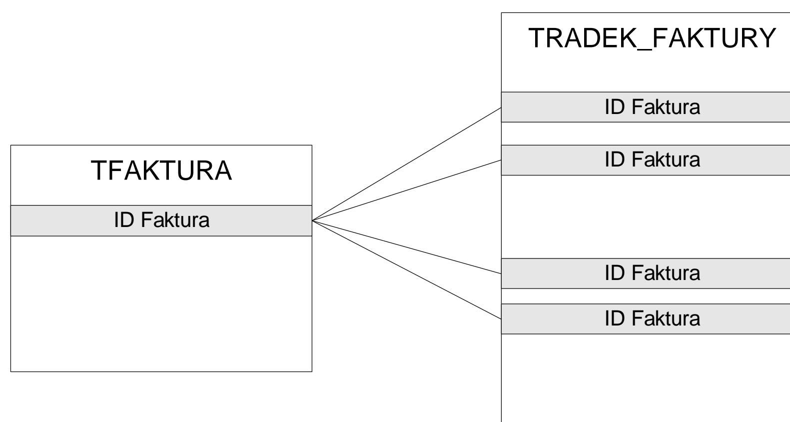
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

zavedené optimalizační techniky, protože její použití vedlo paradoxně oproti předchozímu souborovému řešení ke zpomalení práce s daty.

Uveďme na ukázku jednoduchou ukázku datové struktury realizované pomocí shody hodnot klíčů v relační databázi:

- data hlavičky faktury `TFAKTURA` mají jeden sloupec `ID_FAKTURA` jako vlastní klíč tzv. `primary key`,
- data řádku faktury `TRADEK_FAKTURY` mají cizí klíč `ID_FAKTURA`.

Pro jednu skupinu dat hlavičky faktury existuje N skupin dat řádků faktury, které mají stejnou hodnotu klíče `ID_FAKTURA`. Získání dat hlavičky faktury spolu s daty řádků (JOIN) je potom dán klauzulí `WHERE` se shodou klíče `ID_FAKTURA` v obou datových entitách.



obrázek 4 : Vztah data faktury a data řádků faktury

Tento jinak triviální příklad zde uvádím proto, abyste si povšimli určitých formulací, které jsou zde úmyslně zvoleny. U každého analytického pojmu (faktura, řádek faktury) předřazuji slovo data, tedy nikoliv „faktura“, ale „data faktury“ a nikoliv „řádek faktury“, ale „data řádku faktury“.

Pomocí ERD vyjádříme datové entity a vztahy mezi nimi v celém systému a získáváme tak celkový datový obraz celého systému i se vztahy mezi daty. Skupiny dat jsou mezi sebou propojeny nějakým algoritmem (shodou klíčů v tabulkách apod.).

Poznámka: Pokud je v pozadí objektové aplikace relační databáze, je pro tvorbu informačního systému i v objektovém prostředí nutné zavést ERD diagramy. Pokud objekty odkládají své informace v perzistenci do relační databáze, potom doplňkem všech použitých modelů UML musí být jako další diagram navržen diagram ERD, který popisuje vztahy mezi daty v relační databázi. Na rozdíl od Coad-Yourdonovy školy je však tento diagram až „sekundárním diagramem“ a také vzniká až na základě požadavků objektů pro ukládání dat do databáze. Samozřejmě datový diagram ERD není součástí UML – jedná se o diagram speciálního případu řešení aplikace při odkládání dat do relační databáze.

Ve strukturálním programování (a Coad-Yourdonově škole) je ERD jedním z „primárních modelů“ informačního systému.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

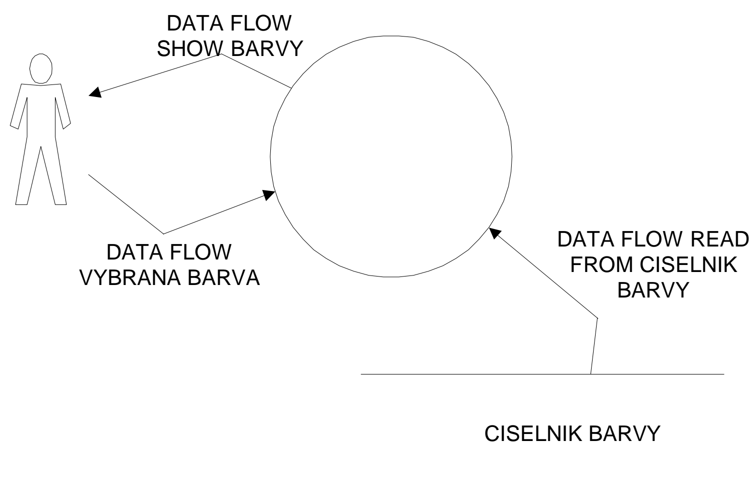
Diagramy DFD

Dalším diagramem, který se často používá ve strukturálním programování, je tzv. *Data Flow Diagram*, ve zkratce DFD. Tento diagram vyjadřuje tzv. toky dat v informačním systému. Protože základními dvěma okruhy jsou funkce a data, tento diagram zobrazuje toky dat mezi procesy (funkcemi) a daty a (např. uložištěmi dat) a okolím. Datové toky se chápou jako přenosy dat a v tomto pojetí se zobrazuje přenos dat od uložiště dat přes procesy až k vnějšímu prvku systému a naopak.

Procesy (odpovídající funkce) se v této škole zobrazují kružnicí (resp. elipsou), datové toky šipkami, uložiště dat dvěma rovnoběžkami a vnější prvek pomocí obrázku „panáčka“.

Příklad

Navrhujeme systém evidence aut. Uveďme pro ilustraci klasickou situaci: V uložišti dat existuje číselník barev a úkolem obsluhy je vybrat jednu z barev pro dosazení do evidovaného auta.



obrázek 5 : Ukázka Data Flow Diagramu

V uvedeném příkladu jsme zobrazili tři toky dat (chápané také v určité posloupnosti, tj. sekvenci).

První tok označený jako DATA FLOW READ FROM CISELNIK BARVY reprezentuje „načítání“ číselníku barev a je specifikován SQL příkazem například nějak takto

```
SELECT id_barva, nazev FROM BARVY
```

Druhý tok označený jako „SHOW“, ukazuje nutnost zobrazení načtených dat, například pomocí GUI prvku provázaného na prvý datový tok. Tok zobrazení obsahuje například `nazev, id_barva,` ale položka `id_barva` v tomto toku je uschována jako hidden pole (nezobrazuje se, ale je jej třeba pro navázání vybrané barvy).

Třetí tok znamená výběr a reprezentuje tok konkrétního identifikátoru barvy (například tento datový tok obsahuje položku `id_barva`). Obsluha vybrala podle položky `nazev`, ale zpět putuje tok `id_barva`. Tento identifikátor se poté dosazuje do nějaké jiné struktury, například do dat `Auta` jako hodnota tohoto cizího klíče.

Je třeba zdůraznit, že data flow diagramy nejsou součástí UML diagramů, i když jsou (v rámci strukturálního programování) velmi srozumitelné a přehledné. Patří však ke klasickým diagramům strukturálního pojetí aplikace, protože zobrazují velmi efektivně vztahy mezi funkcemi a daty.

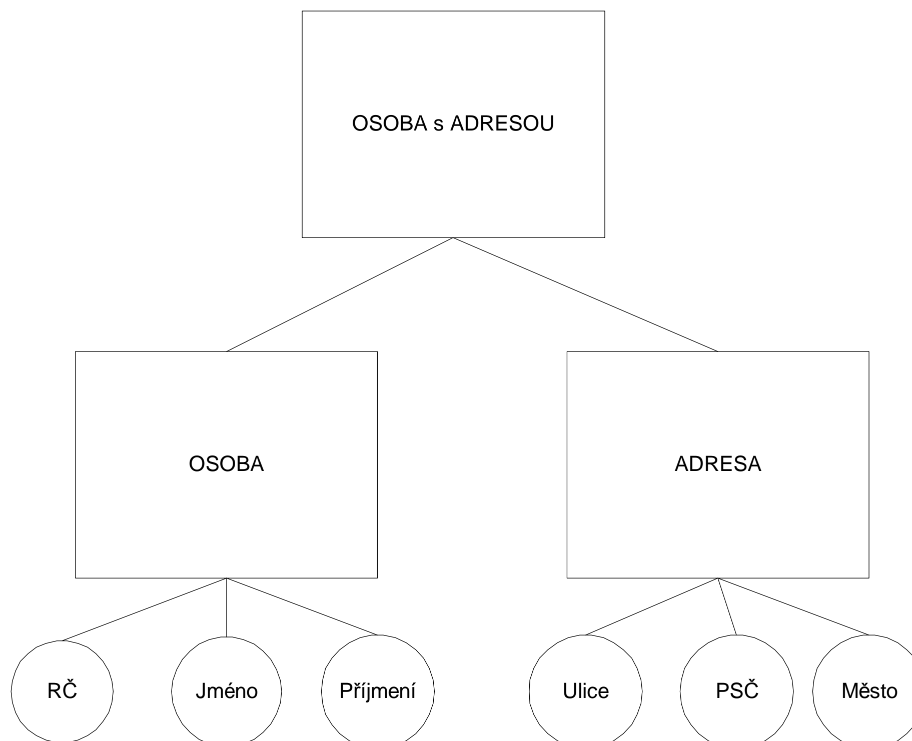
Jejich použití v objektovém prostředí však postrádá smyslu, jak si ukážeme v dalších kapitolách. Jiné diagramy UML totiž popisují přesně vztahy spolupráce mezi objekty. V dalších kapitolách si vysvětlíme, proč diagramy toků dat postrádají v OOP a UML smysl.

Diagramy DSD

Jako poslední diagram uvádím málo používaný DSD diagram, což je zkratka pro *Data Structure Diagram*. Tento diagram umožňuje provádět „dekompozici“ jinak složitějších a obecněji chápaných struktur dat na jemnější struktury dat až po úroveň tabulek a polí (sloupců). Tuto dekompozici provádí jednoduchým zápisem pomocí rozkladu. Pomocí Data Structure Diagramu můžeme například vyjádřit skladbu datových toků nebo uložených dat.

Příklad

Na následujícím diagramu je znázorněna entita Osoba s adresou, která je složena ze dvou entit:



obrázek 6 : Ukázka DSD diagramu. Skalár (dále nedělitelná infomrace) se v diagramu se může označit kolečkem

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Pomocí tři typů diagramů používaných v Coad-Yourdonově škole lze nakonec popsat celý systém pojatý strukturálně velmi přesně a výstižně. Osobně tuto školu považuji za určité vyvrcholení ne-objektového pojetí aplikací.

Poznámka: Měl jsem osobní zkušenost pracovat (ještě před přechodem na OOP) v jednom velmi silném CASE nástroji této školy (název tohoto nástroje je „CASE WESTMOUNT"). Opravdu zápisy v tomto prostředku jsou přesné a výstižné.

Je pochopitelné, že mezi všemi diagramy ERD, DFD a DSD musí existovat konzistence. Například nelze vytvořit datový tok obsahující položky, které nejsou obsaženy v uložišti dat apod. Vztah mezi ERD a datovými toky musí také souhlasit, stejně tak mezi entitami vyjádřenými v DSD a ERD. Díky tomu lze kontrolovat (provádět „check") mezi diagramy.

Přes všechny výhody této škol sami její tvůrci tuto opustili a přešli na objektové modelování založené na OOP.

Základní vztahy mezi DATY a FUNKCEMI

Pro lepší pochopení a snadnější přechod k objektově orientovaného přístupu je vhodné si uvést základní možné vztahy mezi daty a funkcemi ve strukturálním programování.

Viditelnost proměnné

Pokud deklarujeme proměnnou resp. ji zavedeme v určité části kódu, může se stát, že v jiné části kódu není tato proměnná dostupná, jinak řečeno není viditelná. V tom případě kompilátor příslušného jazyka resp. samotný program nahlásí chybu o neznámé proměnné, například chybové hlášení *„Variable not defined"* anebo v jiném jazyce *„Unknown Identifier"*. Pokud kompilátor takovouto chybu nenahlásí a lze s proměnnou pracovat, potom budeme hovořit jako o proměnné v dané části kódu viditelné. Takto budeme chápat viditelnost proměnných a indikátorem této viditelnosti je právě hlášení kompilátoru ve stylu *„Variable not defined"*.

Sdílení proměnné dvěma a více funkcemi

Pokud existují dva procesy, resp. dvě části kódu programu, které mají stejnou proměnnou jako viditelnou, potom oba procesy tuto proměnnou mohou sdílet. Funkce nemusí být spuštěny současně, ale třeba postupně, nehovoříme zde o paralelních procesech.

Viditelnost a sdílení proměnných jsou dva pojmy blízké, což není nic nového. Mají-li dva procesy sdílet tutéž proměnnou, potom oba procesy musí danou proměnnou „vidět" a tedy musí být v dosahu jejich viditelnosti.

Znovu-použitelnost proměnné

Pod tímto pojmem nemáme na mysli znovu-použitelnost kódu, ale znovu použitelnost proměnné a budeme jí rozumět takovou situaci, kdy při rozšíření problému o nutnost použít novou proměnnou nemusíme deklarovat tuto novu proměnnou, protože tato nová proměnná se automaticky rodí s tímto novým problémem.

Klasickým příkladem znovu použitelné proměnné jsou lokální proměnná procedury. Pokud zavoláte proceduru, která má v sobě deklarovánu proměnnou jako lokální proměnnou této procedury, automaticky se tvoří nová proměnná této procedury. Dokonce při rekurzi se proměnné z jednotlivých úrovní rekurze od sebe odlišují, i když mají tentýž název a jsou deklarovány pouze jednou.

Ve smyslu zda je nutné deklarovat novou proměnnou anebo ne a tedy „znovu-použít" existující proměnnou budeme chápat znovu použitelnost proměnné.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Viditelnost proměnných a sdílení proměnných v procedurálním programování

Ve strukturálním programování existují určité prvky „encapsulace" tj. zapouzdření, ale pouze mezi procedurami. Zapouzdření se projevuje v tom, že procedury lze skládat a tak znovu používat z knihoven. Při volání procedur se nezajímáme, jak jsou tyto uvnitř poskládány. Bez obav můžeme „opravit" vnitřek procedury a vyměnit jej za jiný vnitřek. Při nezměněné hlavičce procedury se navenek z hlediska volající procedury nic nestane. Běžně se této vlastnosti používá ke zdokonalení kódu (například náhrada částí procedur v kódu assembler apod.) anebo k jinému rozložení volání funkcí uvnitř dané funkce.

Jinak samozřejmě tato vlastnost zapouzdření procedur vede k tomu, že program napsaný strukturálně je vůbec opravitelný, v opačném případě by se musel celý vyměnit.

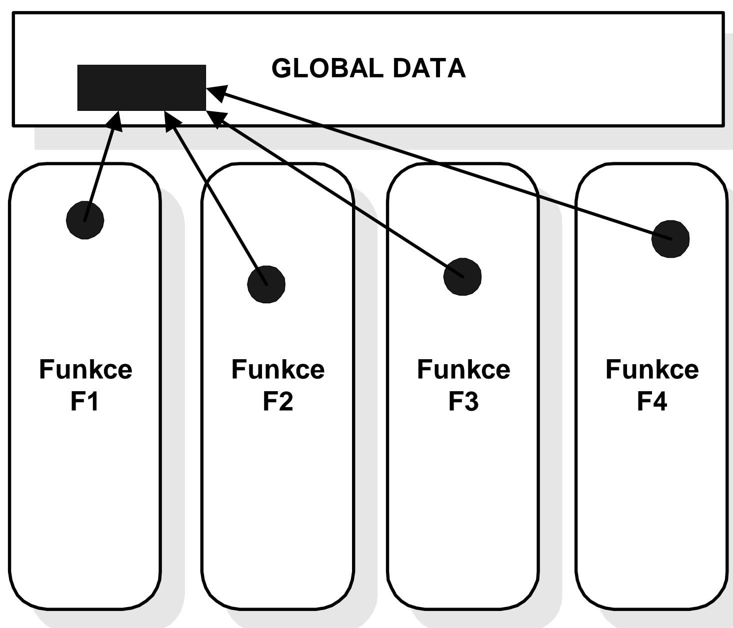
Procedury se takto chovají jako „uzavřené černé skříňky". Této skutečnosti se v procedurálním programování bohatě využívá a toto využití vede k vysoké (ale neúplné) flexibilitě při tvorbě programu. Totéž, co platí o procedurách (možnost uzavřeného skládání) totiž neplatí o proměnných. Nemožnost použít zapouzdření i pro proměnné úzce souvisí se vztahy možné viditelnosti proměnných z procedur.

Základním problémem procedurálního přístupu je možná poloha proměnné vůči proceduře. Proměnné ve strukturálním programování se obecně mohou vyskytovat v těchto základních polohách, kdy jsou procedurou viditelné:

- Proměnná je obecně globální proměnnou viditelnou z libovolné procedury
- Proměnná je přístupná a viditelná procedurám pouze daného modulu
- Proměnná je vstupním resp. výstupním parametrem procedury
- Proměnná je lokální proměnnou dané procedury

Tyto základní čtyři polohy při procedurálním programování používáme a nutno si uvědomit, že se jedná o jediné možné použitelné polohy viditelnosti. To je dost důležitý závěr: Jinou volbu, než právě tyto čtyři polohy v procedurálním programování již nemáme k dispozici. Pokud tedy zavádíme novou proměnnou v procedurálním programování, musíme si vybrat jednu z těchto čtyř. Žádnou jinou již nemáme na výběr.

Ukažme si nejprve, jak vypadá celá aplikace v procedurálním programování se svými proměnnými. K tomu nám poslouží následující obrázek, který znázorňuje u jedné aplikace polohu globálních proměnných vůči procedurám, přičemž aplikace je napsaná pomocí přístupu procedurálního programování:



obrázek 7 : Možné vztahy mezi funkcemi a daty ve strukturálním programování

V předchozím obrázku je znázorněna ta skutečnost, že pokud je proměnná zavedena jako globální, je „volatelná" z různých funkcí a dochází tak k jejímu sdílení.

Každá z poloh viditelnosti, lokální, globální a vstupně výstupní, má své výhody a nevýhody.

Stavy v systému a globální proměnné

Pokud použijete globální proměnnou (data), potom tato proměnná je schopna držet svoji informaci i při volání jiné funkce a to po skončení běhu předešlé volané funkce. Tato vlastnost je nejdůležitější vlastností globálně viditelné proměnné. Díky této vlastnosti globální proměnná reprezentuje nějaký stav v informačním systému.

Pro stav je charakteristické to, že jakmile je získána nějaká hodnota tohoto stavu, je již pro další zpracování absolutně nezajímavé, jak bylo tohoto stavu dosaženo.

Poznámka: Samozřejmě sama hodnota je závislá na tom, co proběhlo, ale podstatné je, že pokud již bylo hodnoty nějak dosaženo, je poté již jedno, jak bylo této hodnoty dosaženo..

Příklad

Pokud je v globální proměnné uschována informace *„Aktuální soubor otevřen*" typu `Ano-Ne`, potom tato informace je čitelná a sdílená funkcemi a každá z těchto funkcí (samozřejmě která to potřebuje) se může na začátku svého toku rozhodnout nějak takto:

```
IF NOT „Aktuální soubor otevřen" THEN ...

    ELSE . . .

ENDIF
```

V příkladu je v sekvenci rozhodování `IF` úplně jedno, zda byl soubor otevřen nějakou funkcí F1 nebo nějakou funkcí F2. Důležitý je stav držený touto proměnnou a její obsah (tj. historie otevření je po otevření nezajímavá).

Protože informace v globální proměnné setrvává i po skončení funkce, reprezentuje tato informace určitý stav, které tyto funkce mohou měnit (v uvedeném příkladu pomocí funkcí pro otevření souboru) anebo se podle ní řídit (v příkladu rozhodování „pokud není soubor otevřen...“). Zavedení globální proměnné mezi funkcemi tedy vede ke sdílení informace, čímž umožňuje jinak složité scénáře toku funkcí rozdělit do částečných scénářů odpovídajících stavům. V každém z těchto scénářů (scénářem rozumíme posloupnost volání funkcí) zahajujeme činnost na základě výchozího stavu, který tato globální proměnná drží ve svém obsahu a v tomto scénáři se nezajímáme o předešlé scénáře, které k tomuto stavu vedly. Tím se řešení může rozpadnout na posloupnosti dosažení hodnot stavů

Sam získaná hodnota stavu v globální proměnné tedy uzavře kapitolu získání této hodnoty, což je obrovská výhoda tohoto přístupu – předešlá historie je nezajímavá a tedy zapouzdřena z časového hlediska předešlého vývoje.

Globální viditelnost proměnných (dat) má kromě této výhody ve strukturálním programování také několik obrovských nevýhod. O první nevýhodě pohovoříme nyní a druhé velké nevýhodě se budeme věnovat v jedné z příštích kapitol, až probereme pojem objekt (tam totiž vynikne uvedený problém více).

Velkou nevýhodou globální viditelnosti proměnných je ta skutečnost, že pokud používáme jednu proměnnou pro „držení“ nějaké informace nezávisle na historii volání funkcí (tedy v poloze viditelnosti globální), a přitom vznikne nový problém téhož typu (například nepotřebujeme pracovat s jedním souborem, ale potřebujeme otevřít dva soubory), potom nastane problém, protože naše funkce totiž počítá v syntaxi pouze s jednou proměnnou. Například je zavedena proměnná *„Aktuální soubor otevřen“* a potřebujeme pracovat se dvěma a více soubory najednou.

Tento problém odpadá u lokálních proměnných. Lokální proměnná pokaždé, když pracujeme s danou funkcí, tak se proměnná alokuje a může být „automaticky“ používána. Pokud bychom uvedenou proměnnou udávající *„soubor otevřen“* zavedli v lokální viditelnosti, nepotřebovali bychom z hlediska syntaxe zavádět novou proměnnou pro druhý soubor. Oproti tomu ztratila by se díky lokálnosti schopnost reprezentovat stav viditelný několika funkcemi v časové posloupnosti volání různých funkcí za sebou.

Většinou se tento problém potřeby nové proměnné beze změny syntaxe ve strukturálním programování řeší pomocí zavedení nějaké řady proměnné a odvoláváme se na index.

Příklad

Pokud bychom chtěli pracovat s N soubory a u každého zaznamenat jeho stav„Soubor otevřen“, můžeme zavést namísto jedné proměnné řadu proměnných „Soubor otevřen[i]“ a odvolávat se na index [i]. Vlastnosti souboru jsou tak identifikovány pomocí indexu. Pokud zavedeme další vlastnosti souboru A, B, C atd., potom zavedením několika řad indexu A, B, C přiřazujeme každému souboru skupinu vlastností pomocí indexu s danou hodnotou. Index tak reprezentuje identifikaci, „čí jsou to vlastnosti“. Tím se již blížíme k objektovému náhledu, zde je identifikátorem objektu číslo i.

Všimněme si, že nejlepší variantou by byla nějaká nová zvláštní viditelnost proměnných taková, že by slučovala obě výhody – proměnná by byla na jedné straně „globální“ a na straně druhé „lokální“ a to „nějak současně“. Pokud byste chtěli tento problém řešit pomocí strukturálního programování, je to možné pomocí dynamicky alokovaných řad, ale zapomeňte na to. Problém plně řeší objektově orientovaný přístup.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Objektově orientovaný přístup

Ve strukturálním pojetí existují dva základní okruhy prvků systému, pouze DATA a FUNKCE.

„Čistý“ objektově orientovaný přístup tyto dva okruhy nezná a zavádí úplně jinou filosofii tvorby informačního systému. Základním prvkem systému je objekt.

Namísto prvků DATA a FUNKCE se zavádí oproti těmto dvěma pojmům úplně odlišný prvek, tzv. objekt.

Vysvětlíme si nyní, co je chápáno pod jedním objektem v OOP.

Poznámka: Tedy zapomeňme na chvíli na data a funkce zavedené ve strukturálním programování. Ale povzdech následuje: Vím ze zkušenosti při konzultacích, jak je to pro strukturální programátory obtížné!

Základní vlastností objektového prostředí je existence struktur v programu (kódu anebo v binárním tvaru) majících vlastnosti *objektu*. Pro pochopení rozdílu mezi strukturálním programování a OOP je třeba si uvědomit, že vlastnosti objektů jsou něčím kvalitativně novým a diametrálně odlišným od vlastností proměnných a funkcí ve strukturálním programování. Právě pochopení této základní odlišnosti je základem pro pochopení OOP.

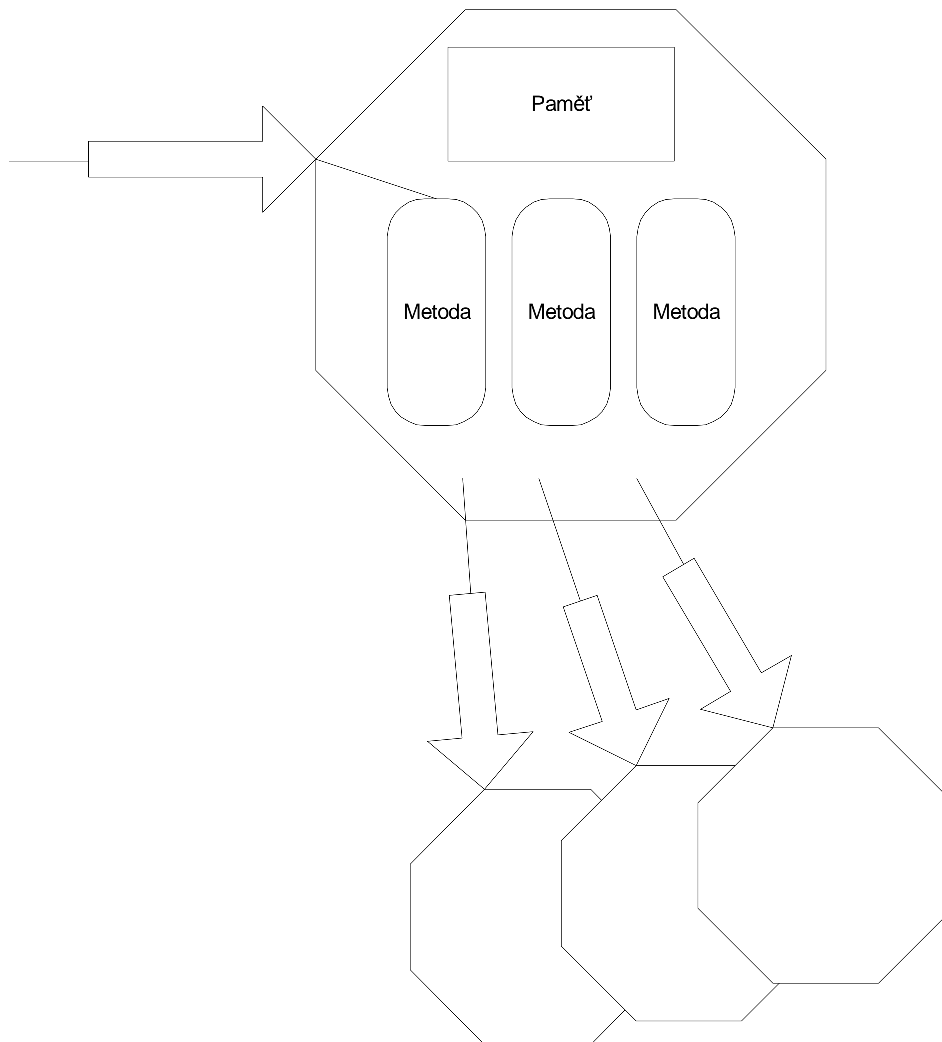
Prostředí resp. jazyk nebo technologie splňuje objektově orientovaný princip, pokud dovoluje vytvářet struktury nazývající se objekty a tyto nové struktury mají následující vlastnosti.

Objekt je definován jako v programu uzavřená struktura, která:

- obsahuje *vnitřní paměť,* tj. má vlastnost si něco pamatovat. Tato vnitřní paměť se někdy nazývá *atributy* objektu. Důležité je, že vnitřní paměť objektu je zvnějšku objektu nepřístupná. Je jeho soukromou záležitostí, co si objekt pamatuje a jak.
- obsahuje *metody objektu*, což jsou procedury nebo funkce, resp. obecněji posloupnost kódu programu, které vykonávají nějakou činnost nad vnitřní pamětí objektu a pouze nad ní. Metody objektu jsou zvnějšku také neviditelné a nepřístupné. Nemůžeme metodu objektu zavolat přímo. Vnitřní paměť a vnitřní metody téhož objektu jsou vůči sobě plně viditelné stejně jako při globální viditelnosti. Metoda objektu má v dosahu své viditelnosti vnitřní paměť a naopak. Jinými slovy, metoda objektu je to, co je schopno pracovat s vnitřní pamětí objektu a nic jiného. Můžeme si představit, že metoda objektu je jedinou možností, jak zpracovat vnitřní paměť objektu.
- je strukturou, která je nějakým mechanismem *schopna přijmout a zpracovat zprávu zvnějšku.* V každém jazyce a použité technologii je tato schopnost zpracovat zprávu implementována nějak jinak. Mechanismus zpracování zprávy je takový, že každý objekt v sobě obsahuje tzv. *protokol zpráv*, což je přiřazení *zprávy versus metoda objektu.* Můžeme si to představit jako převodník mezi zprávou a metodou. Každá *zpráva* v protokolu zpráv má přiřazenu právě jednu *metodu objektu.* Přijmout a zpracovat zprávu pro objekt znamená, že objekt v protokolu zpráv nalezne odpovídající zprávu, k ní nalezne odpovídající přiřazenou metodu a spustí ji se vstupními přijatými parametry. Po vykonání metody vrátí zprávě výstupní parametry. Jedinou možností, jak spolupracovat s objektem, je *poslat mu zprávu.* Jinými slovy při použití objektu jako uživatel zvně se nezajímáme o vnitřní strukturu objektu, o uspořádání metod a atributy (které stejně nevidíme), ale o „reakce“ objektu na zprávy. Každá zpráva může obsahovat (nést s sebou) tzv. vstupní a výstupní parametry, což mohou být opět objekty. Množina zpráv, kterou může objekt přijmout a tedy kterou může uživatel objektu použít, se nazývá *interface objektu.*

- může *obsahovat jiné objekty*, kterým je schopen poslat zprávy v rámci běhu své metody a tak řídit jejich činnost. Tímto vznikají sekvence zasílání zpráv od objektu k objektům a vzniká tak tok činnosti programu.

Tyto čtyři základní postuláty jsou základními postuláty OOP, jsou implementovány jak v kódu OOP, tak například v komponentní technologii. Uvedené čtyři postuláty lze znázornit pomocí obrázku:



obrázek 8 : Struktura objektu: Objekt je schopen přijmout zprávu, je schopen vyvolat metodu podle přijaté zprávy, obsahuje paměť a další objekty

Důležité je, že tento seznam je pro další odvozování vlastností prvků v OOP dostačující a je tedy axiomatický (jsou to principy OOP).

Pojetí aplikace viděné jako objekty je podstatně odlišné od strukturálního. Pro mnoho pracovníků v oblasti tvorby SW je dost obtížné opustit naučený pohled na systém jako DATA + FUNKCE a přejít na objektové programování. Cílem této knihy je mimo jiné napomoci čtenáři překlenout tuto hranici pomocí objektového modelování.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

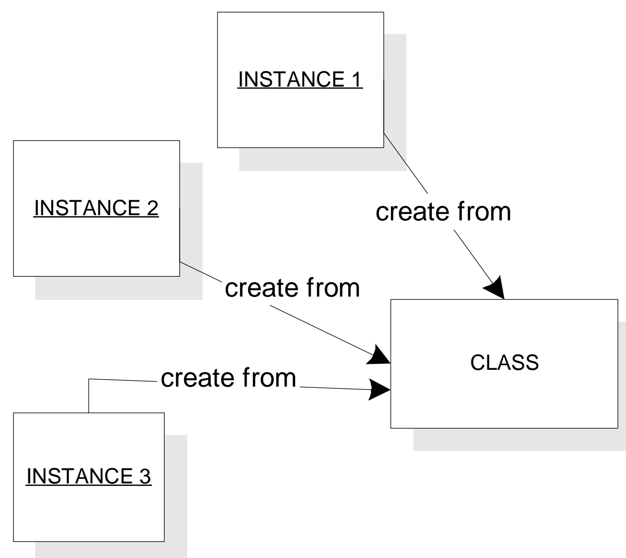
Třída jako objekt

Všimněme si, že v postulátech obecné teorie OOP se hovoří o vlastnostech objektu jako takového a není v nich žádná zmínka o nějaké třídě. Důležité je vědět, že v „obecné teorii OOP" by bylo možné, avšak nikoliv praktické, hovořit o objektu bez třídy. Z hlediska čistě teoretického lze tedy deklarovat a zavést jeden objekt „přímou definicí". Stačí definovat jeho vlastnosti, tj. jaké má nový objekt atributy, jaké má mít metody, jaký má protokol zpráv a z jakých objektů je složen. Tento způsob tvorby objektů „deklaruj jeden objekt po druhém" je podle teorie OOP možný, má však jednu nevýhodu:

Pokud budeme deklarovat „objekt úplně stejných vlastností" podruhé a lišící se pouze názvem, potom se v definici budeme opakovat. Vzpomeňme nyní na princip operace re-use!

Z toho důvodu, aby se definice nemusela opakovat, se zavádí nový objekt *Třída*. Je to takový objekt, který napomáhá vzniknout jako kopyto (jako forma) novým objektům stejných vlastností. Pokud tedy definujeme nový objekt, stačí v této definici uvést, z jaké je *Třídy.* Další je již dáno vztahem mezi objekty. Stačí tedy definovat *jednou* kopyto pro budoucí objekty a můžeme jich poté definovat kolik chceme a nebudeme se v definici opakovat, pouze se odkážeme, že objekt je z této třídy, čímž je dáno, jak je definován.

Znamená to, že pojem třída je objektem, který napomáhá definovat nové objekty jako jejich forma, jako jejich kopyto, jako jejich šablona, jako jejich roditel . Vzniká tímto nový vztah mezi objekty a to mezi objekty, které jsou třídami, vůči jiným objektům, které z těchto tříd pocházejí. Objekty pocházející z dané třídy (definované a tvořené pomocí této třídy) se nazývají instance této třídy. Vztah mezi instancemi a třídou „pochází z" můžeme znázornit takto:



obrázek 9 : Třída sdílí definice objektů – tři objekty pocházejí ze stejné třídy a mají tedy stejné vlastnosti, nikoliv však informační obsah

Všimněme si, že zavedení třídy a její použití pro tvorbu instancí je jedním z uplatnění již uvedeného principu re-use: Opakuje se definice objektů, tedy tuto definici vytrhněme a osamostatněme, čímž vznikne třída (jako objekt). Poté ji provážeme k těmto instancím podle předešlého obrázku, tj. vznikne vazba „instance pochází ze třídy".

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Poznámka: Pokud bychom chtěli s trochou nadsázky přirovnat vztah třídy a jejích instancí, mohli bychom použít přirovnání ze života hmyzu. V mraveništi existuje matka královna, která nechává „vznikat mravencům". Oba prvky systému, jak matka královna, tak mravenec, jsou objekty, ale existuje mezi nim vztah v tom smyslu, že matka tvoří podle vnitřně zadaného kódu mravence. V OOP matku královnu reprezentuje objekt třídy a mravence zastupují instance. Existuje pouze ten rozdíl, a to, že pro každý druh mravence (tj. mravenec dělník, mravenec voják atd.) existuje v této paralele v OOP pro každý „druh instance" zvláštní třída, patřící do jednoho stromu dědičnosti.

Poznámka:

Třída, jejímiž instancemi jsou třídy, se nazývá metaclass.

Metody a atributy třídy

Jak bylo řečeno, třída v čistém OOP prostředí je (jak jinak) objektem. Avšak objekt-třída má zvláštní postavení oproti objektům, které vystupují jako instance této třídy. Objekt-třída vystupuje v roli „roditele" objektů – svých instancí. Přitom princip takového zrodu nového objektu musí splňovat principy OOP a je velmi jednoduchý:

Třída jako objekt musí mít povinně mezi svými zprávami (a následně mezi metodami) alespoň jednu zprávu, která jejím zavoláním po objektu-třídě vyžaduje vydání nového objektu, tj. nové instance. Tedy pošleme zprávu objektu-třídě a dostaneme nový objekt-instanci. Nazvěme jednu z těchto zpráv žádajících instanci například jako `New`. Výstupním parametrem této zprávy je instance třídy jako objekt z dané třídy. V pseudo-syntaxi objektového jazyka, kde pro poslání zprávy zvolíme tečkovou syntaxi, bychom mohli takovýto zrod nové instance napsat takto:

```
MyObject = MyClass.New
```

Poznámka: Tečková syntaxe je již specialitou námi zavedeného pseudo-jazyka. Existují jazyky, které nepoužívají tečkovou syntaxi, ale jinou syntaxi poslání zprávy objektu, například pomocí rezervovaného slovo INVOKE:

```
MyObject = MyClass INVOKE New
```

a jiné jazyky používají ještě i jinou syntaxi, například:

```
MyObject := MyClass new
```

Podotkněme, že zpráva, kterou můžeme třídu požádat o nový objekt, nemusí být v dané třídě pouze jedinou. V protokolu zpráv objektu-třídy může být takovýchto zpráv hned několik a to „na výběr" podle potřeb dané situace použití třídy pro zrod instance.

Například může existovat jedna zpráva dávající nový prázdný objekt a může existovat druhá zpráva, která také dává nový objekt, avšak tento objekt není prázdný, ale s již nastavenými implicitními

hodnotami atributů. První zprávu například nazvěme `NewEmpty`*, druhou zprávu vracející vyplněný objekt například* `NewWithImplicitValues` *apod.*

Objekt-třída musí pro své poslání zrodu instancí logicky v sobě „nějak" obsahovat dvě části své definice. Na straně jedné je třída objektem, tedy obsahuje definice svých atributů, zpráv a metod.

Tuto část definice třídy nazvěme jako sekce `CLASS OBJECT SECTION`.

Na straně druhé jakmile požádáme o novou instanci (například metodou `new`), tato volaná třída musí nějak „vědět", jak má nový objekt ve své struktuře vypadat. Proto existuje druhá část definice ve třídě, nazvěme ji jako sekce `INSTANCE OBJECT SECTION`, do které se třída „podívá" jako do šablony, když rodí nový objekt-instanci. Tato sekce je vlastně onou sudičkou, která dává nové instanci požadované vlastnosti. Definice celé třídy se tedy skládá ze dvou sekcí, jedna definuje vlastnosti třídy jako objektu, druhá definuje vlastnosti instancí ze třídy vznikajících.

Poznámka: V každém (čistém) objektovém jazyce jsou pochopitelně tyto sekce nazývány nějak jinak. Například v CACHÉ je definována jako CLASS PARAMETRES a OBJECT BEHAVIOR, podobně v OO Cobolu jako sekce CLASS-OBJECT a OBJECT-STORAGE SECTION apod.

K čemu potřebujeme metody třídy a atributy třídy?

Třída jako objekt musí mít alespoň jednu metodu definovanou v `CLASS OBJECT SECTION`, která poskytuje novou instanci, jinak nelze hovořit o třídě. K čemu však další metody a atributy třídy v této sekci?

Ukazuje se jako velmi výhodné používat metody třídy a atributy třídy definované `CLASS OBJECT SECTION` pro ty informace, které jsou spojené s touto třídou jako roditele instancí. Klasickým příkladem je uživatelský název třídy, nebo držení kolekce resp. počtu instancí, zamykání celé třídy apod. Objekt třídy deklarovaný v `CLASS DEFINITON SECTION` lze tedy využít pro účely evidence všeho, co je společné pro všechny instance (z hlediska instancí se jedná o meta-informace).

Třída jako objekt a problém slepice nebo vejce...

Jak bylo řečeno, třída jako objekt slouží k tvorbě instancí, tedy ke tvorbě dalších jiných objektů. Ale jak je to s třídou? Potřebuje třída pro svůj vznik další třídu? Jak vlastně vznikne třída? Pokud by třída zase potřebovala nějakou třídu ke svému vzniku, tak s takovouto rekurzivní úvahou se samozřejmě zacyklíme.

Třída jako objekt se vymyká vztahu třída-instance v tom smyslu, třída nepochází z nějaké třídy. V objektovém prostředí je zavedení nové třídy tvořeno tak, že rovnou a přímo deklarujeme danou třídu (tj. její `CLASS OBJECT SECTION` a `INSTANCE OBJECT SECTION`) jako jeden objekt syntaxí daného prostředí. Můžeme si to v daném prostředí představit tak, že v prostředí existuje volba pro tvůrce systému „`Přidat třídu`", kde vyplníme `CLASS OBJECT SECTION` a `INSTANCE OBJECT SECTION`. Objekt třídy v té chvíli „automaticky" v systému existuje a můžete se na něj odvolat. Některé třídy jako objekty jsou v těchto prostředích většinou již zavedeny jako systémové (například `Window`, `Peristence`, `Collection`, `String` apod.) a náš nový objekt-třída se k těmto přiřadí.

Třída v „méně čistých OOP" prostředích

Některé jazyky jako Visual Basic, Visual C++, Delphi (Object Pascal) nemají třídu zavedenu přímo jako objekt pomáhající tvorbě nových objektů (tj. třída zde není objektem), ale z historických důvodů je pojem třída zaveden odlišně. Základní myšlenka „třída = kopyto pro objekty" je však zachována.

Třída je v těchto jazycích zavedena jako obdoba ***typu proměnné***. Znamená to, že základní funkce třídy je tímto dodržena. Jedná se také o „kopyto" neboli „formu" pro budoucí objekty dané třídy, tzv.

instance třídy. Využívá se tedy klasického vztahu typu proměnné a proměnné daného typu, který je v těchto jazycích z historických důvodů velmi dobře propracován.

Postup při zavedení nové třídy je takový, že nejprve musíme deklarovat třídu jako obdobu typu proměnné (např. class module, form module, apod. v Basicu, class v jiných jazycích) a teprve potom můžeme vytvářet instance z těchto tříd jako proměnné z tohoto typu. Jedná se však o určitý ústupek od teoreticky pojaté čistoty OOP.

I v těchto jazycích, kde je třída typem proměnné, jsou zavedeny pojmy „metoda třídy“ a „atribut třídy“, například pomocí rezervovaného slova `static`. Jedná se samozřejmě o dost zvláštní a na první pohled trochu nepochopitelnou konstrukci: Třída je zavedena jako typ proměnné a tento typ proměnné má své metody a své atributy. Pokud si však uvědomíme, jak je v čistém OOP třída zavedena, a o jaký ústupek se u těchto jazyků jedná (třída by měla být a už není objektem), můžeme i tuto konstrukci vcelku dobře přijmout jako logickou. Na úrovni třídy je z hlediska instancí uschována meta-informace.

Třída jako vyšší forma abstrakce

Pro pochopení tříd je důležité si uvědomit, že zavedením třídy vzniká nová vyšší abstrakce. Pokud zavedeme třídu jako kopyto pro budoucí instance třídy, tak vlastně definujeme vlastnosti pro **každou instanci, tedy definujeme vlastnosti pro ty instance, které zatím v systému neexistují, ale mohou existovat**.

A to je již jiná situace, než když mluvíme o jedné jediné instanci.

Příklad

Pokud napíšeme v definici třídy osob, že nějaká osoba z této třídy (tj. instance této třídy) bude mít jméno a příjmení, tak jsme vlastně zavedli obecně, že každá osoba (i ta, která zatím neexistuje) bude mít jméno a příjmení. Tato věta je abstraktnější než že „pan Novák má jméno a příjmení“, což je „hmatatelný a přímo ověřitelný fakt“.

Musíme při modelování přesně rozlišovat, zda hovoříme o konkrétní instanci anebo o třídě těchto konkrétních instancí. Pokud při objektovém modelování známe vlastnosti nějaké konkrétní instance (tj. „pan Novák“) a potřebujeme zavést třídu (tj. „Osoba“), tak tento krok nemusí být až tak jednoduchý, jak by se nyní mohlo zdát. Tento proces totiž odpovídá **zobecnění daného pojmu od konkrétního pojmu k abstraktnímu pojmu**.

Polymorfismus

Na postulátech OOP (stejně jako na všech postulátech) je výhodné to, že z nich lze odvodit další vlastnosti objektů. Podobně je tomu u polymorfismu. Setkal jsem se s mnohými způsoby vysvětlení polymorfismu, některé z nich byly velmi nepřesné a mnohdy zbytečně složité. Přitom vysvětlení polymorfismu je jednoduché.

Představme si tu situaci, kdy dva různé objekty mají ve svém protokolu stejnou zprávu, ale každý z nich na ni reaguje jinou metodou. Znamená to, že dva objekty mají ve svém protokolu zpráv stejnou zprávu, ale každý z nich má k nim přiřazenu jinou metodu. Pokud je možné oběma objektům poslat tutéž zprávu, každý z nich vyvolá jinou metodu.

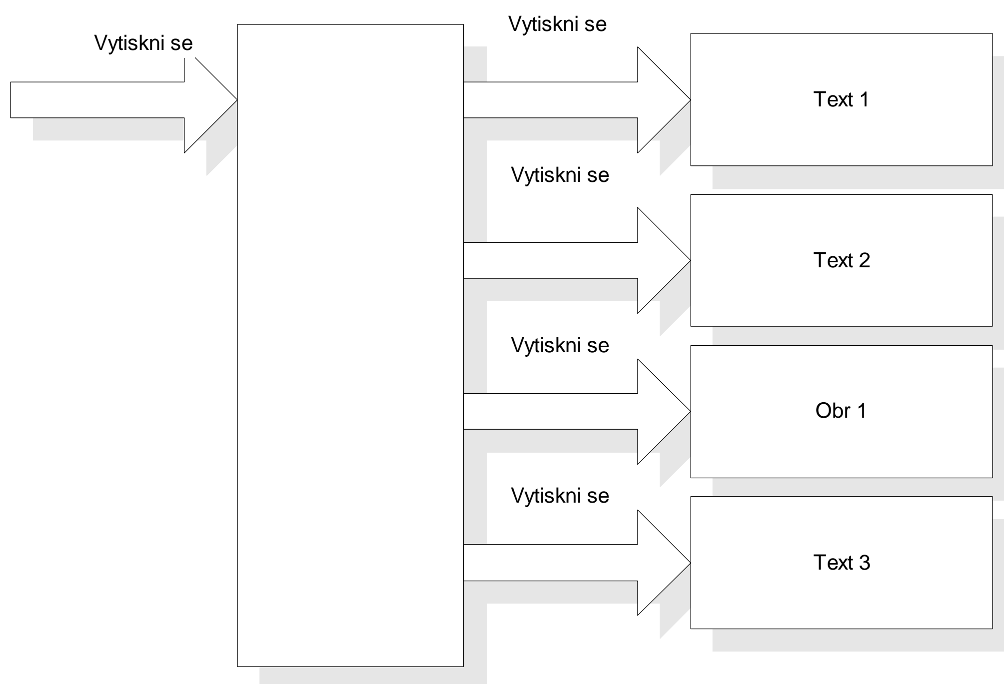
Tuto situaci různého chování objektů na stejnou zprávu nazýváme *polymorfismus*. Uvědomme si, že polymorfismus je v „normálním“ životě natolik běžný, že si jej ani neuvědomujeme. Například představme si N jedinců, kteří rozumí určité zprávě, kterou jim předáme, ale každý na ni reaguje jinou metodou. Takto se přece chová valná většina z nás!

Jako příklad použití polymorfismu uveďme: Stránku jako objekt mohou skládat objekty tvořené z *Textu* a tvořené z *Obrázku,* tj. celá *Stránka* je objektem „kontejner“, který obsahuje několik *Textů* a několik *Obrázků*. Jak objekt *Text*, tak objekt *Obrázek*, mají tutéž zprávu *Vytiskni se*, ale každý z nich tak činí jinak, tj. jinou metodou, jedna metoda je metodou pro tisk textu, druhá metoda je metodou pro

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

tisk bitové mapy. Protože obsluha může poskládat obsah *Stránky* libovolně, není dopředu jasné, jak bude *Stránka* složena, tj. z kolika bude mít objektů *Obrázek* a kolik bude mít objektů *Text*.

Pokud chceme stránku vytisknout, nabízí se nám pomocí polymorfismu elegantní a jednoduché řešení: Každému objektu na *Stránce* se v pošle zpráva *Vytiskni se*. V teorii OOP je polymorfismus zaveden jako důsledek existence protokolu zpráv a má jednoduché vyjádření: „Různé chování různých objektů na tutéž zprávu“.



obrázek 10 příklad vysvětlující podstatu polymorfismu: Stejná zpráva, různé metody

Připomeňme jenom, že polymorfismus vyplynul sám od sebe od konstrukce objektu, která zní: Objekt může přijmout zprávu a na základě ní vyvolat metodu.

Poznámka: V některých jazycích polymorfismus souvisí přímo s kompatibilitou mezi dvěma objekty, které mají stejnou zprávu s různou metodou. V těchto jazycích není až tak jednoduché „dosadit“ namísto objektu Textu objekt Obrázek, protože se jedná o proměnné dvou různých typů. Jedná se zde opět o nepříjemný důsledek zavedení třídy jako typu proměnné a instance-objektu jako proměnné tohoto typu. Řešení se provádí v zásadě dvěma způsoby: Buď pomocí dědění a přepisování metod od předka k potomkovi (virtual metody) anebo pomocí implementace interfacu a vyplňování metod za tímto interfacem. Technologicky se jedná o snahu zavést mechanismus poslání stejné zprávy a vyvolat jinou metodu. Uvedený převodník, který toto umožňuje, je zaveden v obou případech stejně pomocí tzv. virtuální tabulky metod.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Dědění v OOP

V jedné z předešlých kapitol jsme zavedli operaci re-use a již poprvé ji také použili. Při zavedení pojmu třída se „napomohlo" definici nových objektů sdílením definice tvorby objektů pomocí třídy a zavedla se nová operace vztahu mezi instancemi a její třídou, kdy instance pochází z dané třídy.

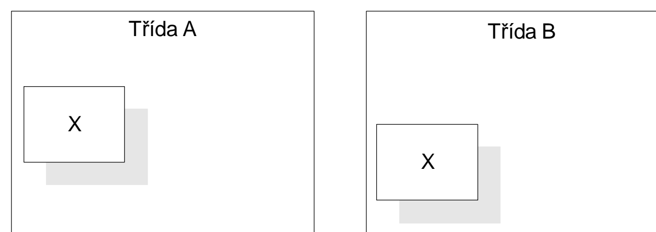
Představme si, že začneme používat třídy pro zavádění objektů a při této práci najednou zjistíme, že některé části definic budoucích objektů se v některých různých třídách budou opakovat. Takovéto opakování se definic ve třídách samozřejmě odporuje teorii re-use.

Podle principu re-use tedy opakující se části definic vyčleníme bokem do zvláštní třídy a zpětně definice tříd provážeme nějakou interakcí. Toto „provázání mezi třídami v definicích" znamená, že jedná třída používá ke své definici jinou třídu, čímž dojde ke sdílení definice. Jak známo, musí existovat operace mezi třídami, která tento re-use mezi třídami pro definici objektů zavádí. Tato interakce se v tomto případě re-use nazývá dědění. Je zřejmé, že tato interakce je jednosměrná – jedna třída (tzv. potomek) se odvolává na jinou třídu (tzv. předka) a tu použije ke své definici.

Pro objekt-instanci vzniklou z dané třídy tak vzniká trochu nová situace: Vlastnosti této instance budou dány jednak definicí třídy, ze které pochází, a také všemi definicemi tříd, na které se daná třída odvolává děděním jako potomek na své předky. Protože vztah dědění je transitivní, instance bude mít rekurzivně také vlastnosti předků daných předků.

Poznámka: Vlastnost transitivnost operace znamená následující vztah: Pokud platí vztah z A do B, přitom platí z B do C, potom platí také z A do C.

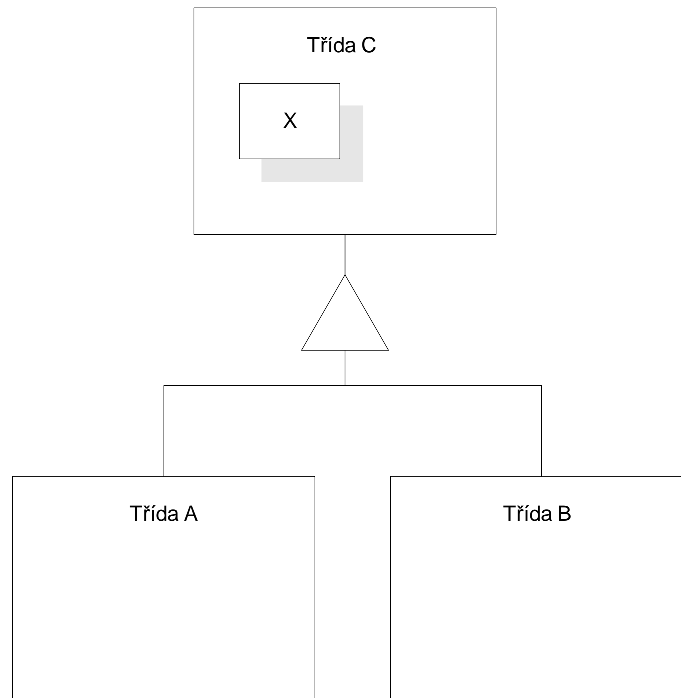
Zavedení vztahu dědění ukazují následující obrázky:



obrázek 11 Redundance X při definicích tříd A a B s některými společnými vlastnostmi

Redundance v definicích tříd se odstraní děděním:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 12 Dědění zavádí re-use mezi třídami sdílením předka, třída A i třída B sdílí definici X

Objekt a modelování informačních systémů

Všimněme si několika podstatných skutečností souvisejících s objektovým programováním a modelováním informačních systémů.

Objektová reference jako jeden ze základních pojmů objektového modelování

Zavedení objektové reference a její pochopení patří k základním pojmům nutným pro zvládnutí objektového modelování. Podstata analýzy, designu a tvorby informačních systému v objektovém prostředí je mimo jiné založena právě na správném pochopení této základní charakteristiky objektu.

Věnujme se proto blíže pojmu objektová reference.

Objektová reference a identifikace objektu

Jako čtvrtý bod definice struktury objektu je uvedena vlastnost objektu „mít přístupné“ jiné objekty a těm zasílat zprávy. Tedy každý objekt může obsahovat další své podřízené objekty, což odpovídá rekurzivnímu pohledu jako na „krabičky v krabičkách“.

Z hlediska této části definice objektu můžeme tedy mezi dvěma objekty nalézt vztah jakési nadřízenosti a podřízenosti v tom smyslu, že jeden objekt má „nějak“ přístupné služby druhého objektu. Jinak řečeno může jej požádat o službu zasláním zprávy. Funkcionalita systému je potom chápána jako posloupnost zasílání zpráv z jednoho objektu do druhého a pochopitelně následně „hlouběji“ z tohoto podřízeného objektu do dalších jemu podřízených objektů. Celý informační systém se ve své dynamické podobě chová jako skupina struktur posílajících si zprávy.

Každý objekt musí být jednoznačně ***identifikován***, aby mohl být nadřízeným objektem osloven při zaslání zprávy. Pochopitelně sama identifikace přímo implikuje dvě základní charakteristiky:

- možnost oslovit „ten správný objekt“

- možnost s daným objektem opakovaně pracovat, tj. využít skutečnosti, že je to tentýž objekt jako před tím oslovený

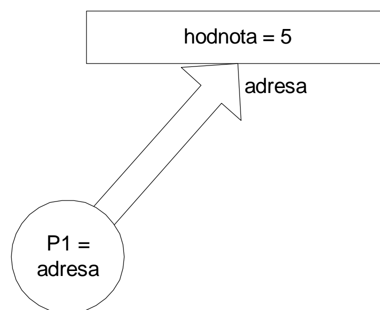
Z hlediska sémantiky se v OOP prostředí jednoznačná identifikace provádí pomocí názvu objektu. Je zřejmé, že v dané určité konkrétní technologii tvorby SW musí být tato vlastnost „odkázání se na objekt a poslat mu zprávu" nějakým způsobem zavedena.

V konkrétních OOP technologiích je toto oslovení pomocí názvu objektu realizováno v implementaci z 90% pomocí proměnné typu ukazatel na alokovanou paměť, kde se oslovený objekt nachází. Pomocí tohoto přístupu znamená slovní spojení „mít k dispozici objekt A" znamená „držet" v proměnné s názvem objektu hodnotu ukazatele na objekt, tj. „ukazovat si" na konkrétní místo v paměti a tam se nachází oslovovaný objekt A. Jinak řečeno hodnota schovaná za identifikátorem objektu reprezentuje ukazatel na objekt. To je však implementační pohled, z hlediska modelování hovoříme na abstraktnější úrovni pouze o identifikaci objektu pomocí jednoznačného názvu objektu a nestaráme se o to, jak konkrétně je toto oslovení pomocí názvu provedeno.

V konzultacích pro získání určité představy o tom, jak propojení mezi objekty konkrétně funguje, se mi osvědčilo použít tohoto uvedeného „implementačního" pohledu za pomoci ukazatelů na objekty, kdy nadřízený objekt si drží ukazatel na svůj vnitřní objekt a tím jej identifikuje a může mu poslat zprávu. Existují sice i jiné technologie propojení objektů (vynucené okolnostmi, například když druhý objekt se nachází na jiném stroji) , ale pro názornou představu, na kterou je zvyklý praktický programátor pracující s ukazateli, to bohatě stačí.

Příklad (pro neznalé pojmu proměnná typu ukazatel na proměnnou)

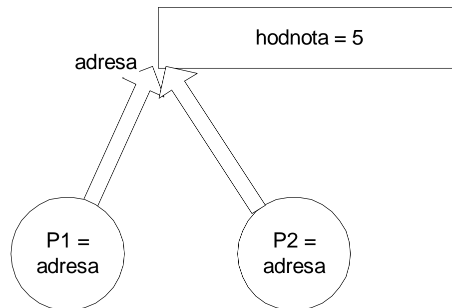
Pro představu spolupráce objektů si vysvětleme, jak v programování funguje práce s proměnnou typu ukazatel. Pokud máme v proměnné typu ukazatel nějakou hodnotu, tak tato hodnota není hodnotou proměnné, kterou potřebujeme znát, ale v proměnné typu ukazatel se vyskytuje adresa paměti a teprve na místě, kam ukazuje tato adresa, se nachází obsah proměnné. Na následujícím obrázku je proměnná typu ukazatel `P1`, která obsahuje hodnotu `adresa`, tato hodnota adresa označuje místo v paměti, kde je hodnota integer `5`. Přes proměnnou ukazatele se dostaneme k hodnotě proměnné. Je zřejmé, že změna hodnoty `adresa` způsobí, že proměnná typu ukazatel si začne ukazovat na jiné místo v paměti (jiná reference).



obrázek 13 Proměnná ukazatel a hodnota v paměti

Zajímavá situace nastane, pokud existují dvě (a více proměnných typu ukazatel), které si ukazují na totéž místo v paměti. V následujícím příkladu si dvě proměnné typu ukazatel ukazují na totéž místo v paměti. Platí `P1 = P2 = adresa`.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



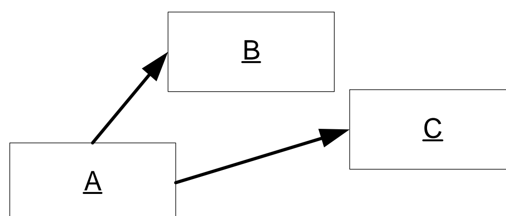
obrázek 14 Dvě proměnné typu ukazatel se sdílenou hodnotou adresy

V této situaci je zajímavé, že k hodnotě na dané adrese můžeme přistoupit ze dvou míst v programu – jak pomocí proměnné `P1`, tak pomocí proměnná `P2`. Pokud přepíšeme na daném místě, tj. na dané adrese, hodnotu pomocí přístupu přes ukazatel `P1` (například změníme hodnotu `5` na hodnotu `6`), tak tuto změnu následně poznáme také při čtení hodnoty přes ukazatel `P2`. Je to způsobeno sdílením na dané adrese. Existuje určitá vcelku vhodná představa „fungování“ objektové reference pomocí ukazatelů na objekt. Pokud si v předešlém příkladu zaměníme číselnou proměnnou (ve které je umístěna hodnota `5`) objektem, můžeme hovořit o proměnné typu ukazatel na objekt. Jeden objekt si ukazuje na druhý objekt „přes ukazatel“. Můžeme tedy v této představě „technologicky“ ztotožnit objektovou referenci s proměnnou typu ukazatel na objekt.

Poznámka: Tato představa je velmi vhodná pro analytické úvahy, kdy si objekty takto představujeme, avšak tato představa nemusí být úplně přesná. Do problému totiž vstupuje navíc ještě pojem interface objektu. Je třeba si uvědomit, že klient objektu „vidí“ jeho interface a díky tomu může poslat objektu zprávu. Uživatel tedy nevidí přímo objekt, ale jeho interface.

Naše současné poznatky již mohou vést k prvním modelům. Nejedná se pochopitelně o přesné modely UML, ale mohou mít velmi dobrou vypovídací schopnost. Znázorněme „přístupnost“ objektové reference od nadřízeného objektu k podřízenému jednoduchou šipkou:

Poznámka: Následující diagram není přesně podle notace UML. Nejblíže k němu má tzv. „Object diagram“. O syntaxi UML bude pojednáno v navazujících kapitolách o UML.



obrázek 15 : Objekt A má zpřístupněny služby dvou objektů B a C

Jednotlivé obdélníky značí objekty a uvnitř jsou uvedeny názvy povinně s podtrženým názvem značící, že obdélník označuje objekt a nikoliv třídu.

Na uvedeném obrázku má objekt s názvem A „podřízeny“ dva objekty s názvy B a C. Znamená to, že objekt s názvem A obsahuje dvě objektové reference, jednu referenci na objekt B a druhou referenci na objekt C. Přes tyto reference může požádat o služby jak objekt B, tak objekt C.

Vnější objekt se může ve svém životním cyklu, například těsně po svém zrodu, nacházet v takovém stavu, kdy objektová reference v něm vložená neukazuje na žádný objekt („prázdný ukazatel“). V tom případě hovoříme o nenaplněné objektové referenci. Pokud objektová reference ukazuje na konkrétní objekt, potom hovoříme o naplněné objektové referenci. Naplnění objektové reference se může i změnit. Do určitého okamžiku si objekt ukazuje na jeden objekt, poté dojde ke změně této reference a objekt si poté ukazuje na jiný objekt.

O jedné časté chybě při chápání objektové reference

Předešlý obrázek má v sobě ukrytu jednu záludnost, na kterou musím upozornit. Z obrázku je zřetelně patrné, jak vypadá vzájemná struktura propojení objektů (a to i se směrem) od nadřízeného k podřízeného objektu.

Tento vztah je zdánlivě podobný vztahu datovému, kdy přes klíč jsou provázána data z různých tabulek. Avšak existuje tu jeden velmi podstatný rozdíl vyplývající z vlastnosti zapouzdření objektu. Pokud někdo používá objekt A, potom tento objekt má v sobě tyto objektové reference ukryty jako svoje vnitřní záležitosti a dotyčný uživatel objektu A o těchto referencích nic neví. Tedy skutečně tento pohled odpovídá doslova přirovnání uzavřené krabičky obsažené v jiné uzavřené krabičce.

V relačních datových vztazích tomu tak není. Provázání přes klíč umožňuje (a nejenom to, dokonce vyžaduje) použít zavedenou relaci mezi daty, tedy uživatel nerozlišuje mezi nějakou „vnitřní“ a „vnější“ strukturou. Existují dvě datové struktury stojící **vedle sebe** a ty jsou provázané přes klíč.

U objektů je tomu přesně obráceně: Pokud držím A, tj. „držím“ tento objekt (mám objektovou referenci), musím požádat jeho o funkcionalitu posláním zprávy. Dovnitř A nemohu díky zapouzdření vidět, tedy vztah těchto struktur A na jedné straně a B, C na straně druhé odpovídá nikoliv spojení, ale **vždy pohledu vnořování do sebe**.

Důležité je si uvědomit, že podřízený objekt je nutno vidět jako „skrytou strukturu nějak vloženou nebo odkázanou pomocí objektové reference“ uvnitř nadřízeného objektu. Rozdíl mezi vnořením objektů a skládáním dat je velmi významný a souvisí s pojmem „skalár“ a s procesem „normalizace entit“.

Představme si, že bychom v rozporu s teorií databází mohli tabulky v relační databázi skládat jako do sebe nořící se struktury. Můžeme namalovat takovou „tabulku“, kde jeden sloupec je vlastně celým řádkem další tabulky, tj. neobsahuje jeden sloupec, ale N sloupců. Vznikne tak příklad popisu následující nenormalizované entity (všimněte si dvojité závorky na konci struktury):

```
A = [ a1, a2, a3, [ b1, b2 ] ]
```

tj. jako bychom chtěli zavést čtvrtý sloupec `a4` vnitřně složený ze dvou sloupců `b1` a `b2`

Nebo zapsáno takto

`A = [a1, a2, a3, B ],` kde `B = [ b1, b2 ]`

Příklad

Ve strukturálním programování se při řešení určitého projektu Personalistiky dozvíme z analýzy, že „Zaměstnanec firmy má N dětí“ a je třeba to evidovat. V prvním přiblížení můžeme napsat

Zaměstnanec = (Údaje, Děti), kde víme, že Dětí je N. Z hlediska teorie relačních databází se jedná samozřejmě o chybnou konstrukci, která je „hrubě nenormalizovaná“.

Tuto „vnořenou“ konstrukci samozřejmě teorie databází nedovoluje. Pokud bychom chtěli obdobu této konstrukce napsat v databázové teorii, potom bychom napsali něco v tomto smyslu:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

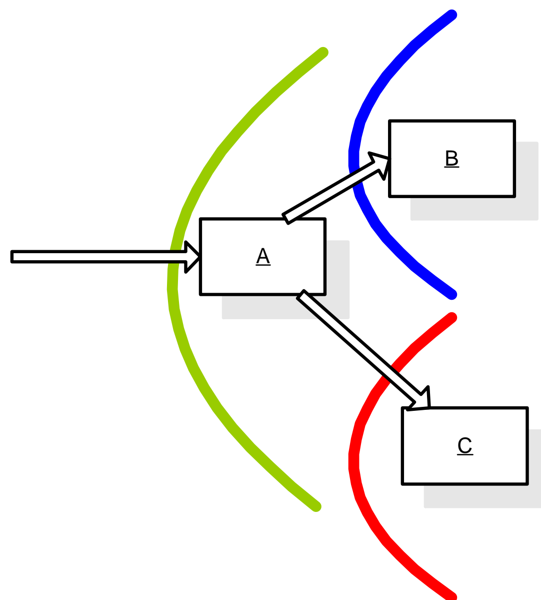
```
A = [ a1, a2, a3]
B = [ b1, b2, a1 (je cizí klíč) ]
```

Prvky `a1, a2, a3, b1, b2` považujeme za dále nedělitelné informace, tedy skaláry.

Naopak v objektové teorii je vnoření objektů a tedy „neskalárnost" při skládání základní vlastností.

Informační prostor objektu

Princip skládání objektů jako do sebe zapouzdřených vnořených struktur přináší další důležitý důsledek: Rozvrstvení systému podle různé viditelnosti informací v rámci objektů. Díky zapouzdření totiž vnější objekt nezasahuje do kompetencí vnitřního objektu a stejně tak objekt nad tímto uvažovaným objektem nezasahuje do kompetencí jeho. Předešlý obrázek si lze z hlediska rozvrstvení informace představit i takto:



obrázek 16 : Rozvrstvení systému podle viditelnosti v rámci objektů

Na obrázku je patrné vymezení informační oblasti „vnitřku" objektu A. Tato oblast je „shora" vymezena vnější slupkou objektu A (zelená slupka), přičemž „zdola" je vymezena zapouzdřením nižších objektů (modrá a červená slupka). Systém se takto „skládá ve vrstvách" objektového pohledu, protože můžeme rekurzivně použít stejný pohled na vrstvy u spodních objektů směrem dolů (oblast B je poté viděna ve stejné pozici náhledu jako oblast A) a objektů nahoru (uživatel objektu A je opět vrstvou pro dalšího uživatele nad ním).

Vrstvy jsou od sebe izolovány, tj. „dovnitř vrstev" uživatelem objektu vidět. Z tohoto důvodu se na každé této vrstvě řeší pouze problém dané vrstvy a co je důležité: Nic víc. O příznivých důsledcích takového pohledu na systém bude pojednáno v kapitolách o stabilitě systému a růstu náročnosti jeho řešení při nárůstu počtu analytických entit v systému.

V každé vrstvě takto viděné se (jak vyplývá z definice objektu) nachází vnitřní paměť objektu a vnitřní metody s možností posílání zpráv vnitřním objektům do „nižší vrstvy".

Na obrázku jsou znázorněny šipky reprezentující „vidění" objektových referencí s následným mechanismem zasílání zpráv. Mechanismus tohoto zasílání je podstatou dynamiky informačních systémů:

1. Vnější uživatel posílá zprávu objektu (na obrázku například zpráva objektu A).

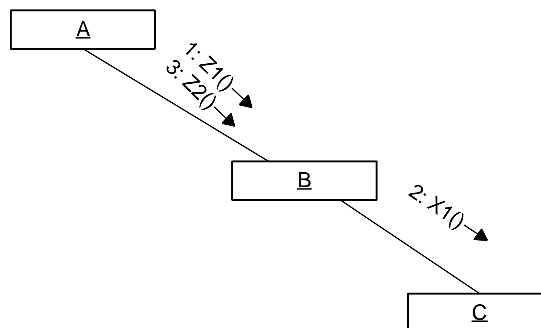
2. Na základě protokolu zpráv (převodníku zpráva-metoda) je vyvolána vnitřní metoda v objektu A.
3. Tato metoda může pracovat s vnitřní pamětí objektu resp. poslat zprávu některému z vnitřních objektů vložených objektovou referencí do objektu A (tj. v rámci dané metody se posílá zpráva do B nebo C). Poslání zprávy do vnitřního objektu však není nic jiného, než poslání zprávy. Co se děje uvnitř objektů B nebo C není pro oblast A důležité. Objekt A může přijmout výstupní parametry zasílané zprávy.

Jedním z důležitých úkolů objektového modelování je dostatečně popsat tuto dynamiku na základě analýzy a poté designu. Teoreticky vzato popis všech objektů v systému se vztahy vnoření spolu se zprávami můžeme považovat za celkový a úplný popis dynamické funkcionalitu systému.

Vztah objektu a jeho klienta a analytická nutnost dodržení anonymity klienta

Pokud celý informační systém navrhneme pomocí objektů, můžeme si jednotlivé činnosti tohoto systému představit vždy jako postupné delegování zpráv od objektu k objektu. Každý „volající" objekt posílající zprávu je vlastně uživatelem jiného „volaného" objektu.

Dohodněme se na zobrazení zpráv pomocí šipek od objektu k objektu podél linku objektů. Pohled na určitou část systému potom může vypadat například takto:



obrázek 17 : Příklad spolupráce objektů v systému

Poznámka: V UML syntaxi tohoto diagramu odpovídá tzv. Collaboration diagram, jehož použití a prvky si ukážeme v jedné z kapitol o UML.

Na daném obrázku jsou znázorněny tři objekty A, B, C a jejich linky (beze směru). Podél linku jsou znázorněny i zprávy, které si posílají objekty. Ze směru „kdo komu posílá zprávu" lze logicky usoudit „kdo koho musí vidět". Zprávy jsou označeny šipkami podél linků, u každé šipky je nejprve pořadí zprávy v rámci diagramu (pořadí v diagramu je 1, 2, 3) a poté název zprávy.

Z uvedeného diagramu vyplývá, že celá sekvence zaslání zpráv vypadá takto:

1. Objekt A posílá zprávu Z1 objektu B
2. Objekt B poté posílá X1 zprávu objektu C

3. Až tato sekvence skončí a vrátí se řízení objektu A, tento posílá ještě jednu zprávu Z2 objektu B

Je zřejmé, že objekt A je uživatelem objektu B a objekt B je uživatelem objektu C. Tento vztah „uživatel objektu" budeme dále nazývat klientem objektu.

Představme si, že bychom pomocí všech takovýchto a podobných diagramů nalezli všechny případy, kdy jsou objektu A zaslány zprávy v celém informačním systému. Pro zavedení objektů je však důležitější třída, ze které objekt pochází. Sám objekt A pochází z nějaké třídy, například třída CA, viz předešlá kapitola o významu třídy. Musíme provést revizi jako vyhotovení seznamu všech zpráv pro všechny objekty z téže třídy. Tím nalezneme všechny zprávy, které jsou třeba k tomu, aby objekt A a všechny objekty z téže třídy CA mohly fungovat. Můžeme tak definovat třídu v sekci `INSTANCE OBJECT SECTION`. Dostaneme tak seznam všech nutných zpráv objektu A, současně všech nutných zpráv objektů zavedených přes třídu CA. Jakékoliv vynechání kterékoliv zprávy v třídě CA by znamenalo, že část našeho systému by přestala fungovat. Touto revizí jsme zjistili seznam všech zpráv (a k nim metod) libovolného objektu ze třídy CA.

Když budeme tento objekt navrhovat (tj. psát pro něj třídu), tak sice jsme vyšli z kontextu jeho celkového použití v systému, ale navrhujeme jej tak, aby jeho funkcionalita nezávisela na tom, kdo ji bude používat. Objekt pouze plní svoje funkce a nestará se to, kdo a kdy bude jeho funkcionalitu volat. Dodržování tohoto principu je pro začátečníky v OOP poměrně dost náročné.

Tento princip se nazývá anonymita klienta a značí, že objekt poskytuje služby „a nic víc", o samotném klientovi neví vůbec nic.

Příklad na nedodržení anonymity klienta

Uvedu jeden příklad chybné úvahy, která je založena na narušení anonymity klienta. V jednom mailu se objevil následující dotaz:

```
Mám několik metod objektu pro vytvoření zálohy, komprimaci atd. a jednu
velkou metodu, která tyto jednotlivé metody spojí do jedné, aby uživatel
mohl zavolat pouze tuto metodu a nemusel se zdržovat hledáním jednotlivých
dílčích metod. V rámci této 'velké' metody je však třeba zobrazovat hlášky.
Mohl by jste mi prosím alespoň naznačit, jak by měla struktura objektu
vypadat? Vím, že to určit nejde, ale alespoň dle Vašich zkušeností s
tvorbou objektů. Napadlo mě řešit tento problém pomocí vyvolání událostí.
Poté by vše fungovalo. Nikde však není zaručeno, že klient událost ošetří.
Nebo nechat na uživateli, aby si toto zpracoval podle sebe?
```

Všimněte si jedné dost důležité věty tohoto dotazu. Tazatel vcelku logicky tvrdí:

```
...Nikde však není zaručeno, že klient událost ošetří...
```

To opravdu nikde není zaručeno. Pokud však naprogramuji takovýto objekt a nastane nějaká událost, kterou by uživatel objektu měl odchytit, tak tuto událost vyvolám a poskytnu ji uživateli objektu případně i s informacemi a dále se nestarám o to, zda ji uživatel odchytí nebo ne. Navrhovaný objekt je pouze „hromádkou služeb" a nestará se o to, co a jak činí uživatel objektu. Tento pohled vytváří objekt mnohem flexibilnějším a stabilnějším, protože de facto ošetřuji všechny možné stavy, kterými objekt může projít, a víc mne nezajímá. Pokud jej navrhnu z pohledu jeho funkcionalit a nikoliv z pohledu starosti o uživatele, tak se objekt stává opravdu samostatnou vcelku „inteligentní" součástkou, která se chová velmi rozumně. K tomuto problému se budeme velmi často vracet v různých příkladech.

Agregace a běžná asociace mezi objekty

Podřízenost objektů vůči nadřízeným je realizována pomocí objektové reference. Objekt je tímto jednoznačně identifikován a lze mu od vnějšího objektu poslat zprávu. Obecně tuto referenci mezi objekty nazývá UML jako „link" mezi objekty. Syntaxe UML nerozlišuje směr tohoto vztahu mezi objekty (pozor – řeč je o objektech, tento směr „kdo koho vidí" zavádí až Class model, tj. model tříd).

Podřízenost jednoho objektu vůči druhému vyjádřená objektovou referencí může být analyticky v objektovém modelování chápána dvojím způsobem, buď jako **agregace** anebo jako **běžná asociace**. V obou případech se však jedná o vztah dvou objektů, z nichž druhý podřízený objekt má svou objektovou referenci vloženu do prvního objektu.

Agregace

Pokud vnější objekt analyticky obsahuje vnitřní objekt jako svou část, tj. vnitřní objekt analyticky skládá vnější objekt, potom hovoříme o agregaci. Analyticky chápáno se tedy jedná o vztah skládání pojmů z částí, kde části skládají celek.

Příklady agregací:

Objekt Faktura obsahuje objekty Řádky Faktury

Objekt Osoba obsahuje objekt Bydliště

apod.

Pro agregaci je důležitá ta skutečnost, že objekt nadřízený obsahuje objekt podřízený jako svou část, a proto je tento nadřízený objekt zodpovědný za zrod a zničený své části.

Běžná asociace

V případě tzv. běžné asociace se jedná také o vložení objektové reference na vnitřní objekt do vnějšího objektu, avšak toto vložení má jiný charakter, než předešlý případ agregace. Při běžné asociaci vnější objekt získává objektovou referenci na vnitřní objekt zvenku (dočasně) k použití. Při běžné asociaci na rozdíl od agregace objektová reference neukazuje na objekt, který skládá vnější objekt, ale na objekt, který je do vnějšího objektu pouze dočasně dosazen. Důvodem tohoto dosazení je poskytnout možnost nadřízenému objektu využít funkcionality právě tohoto objektu.

Protože při běžné asociaci vnější objekt pouze získává objektovou referenci na vnitřní objekt, není to „jeho" objekt, ale pouze „půjčený" objekt a tedy vnější objekt není (a nesmí být) zodpovědný za zrod a zničený svého asociovaného objektu. Tento objekt totiž není jeho vlastnictvím.

Představme si, že vnější objekt získá objektovou referenci na svůj podřízený objekt a tento objekt nechá zničit, což je chybou. V tom případě provedl operaci nad objektem, který není v jeho kompetenci a tedy někomu jinému (kdo tento objekt drží jako svou agregaci) tento objekt zničil.

Agregace a běžná asociace z hlediska syntaxe UML verze 1.3

Kromě uvedených pojmů agregace a běžná asociace se v UML zavádí pojem asociace (nezaměňovat s běžnou asociací).

Asociace reprezentuje jakýkoliv vztah objektové reference do druhého objektu a agregace je zvláštní případ asociace, kdy vložený objekt je chápán jako část vnějšího objektu. Existují tedy tři pojmy: asociace (jakýkoliv vztah), který se dělí na agregaci a běžnou asociaci.

Poznámka: V některých případech se stává, že přívlastek „běžná" se u pojmu „běžná asociace" vynechá a hovoří se pouze o asociaci a přitom se má na mysli běžná asociace.

Naplnění agregace a naplnění běžné asociace

Pod naplněním agregace nebo běžné asociace máme na mysli takový proces, kdy před tímto procesem není objektová reference naplněna a po jeho skončení naplněna je. V tomto ohledu se jedná o proces, kdy se mění stav objektové reference z nenaplněné na naplněnou. K tomuto procesu dochází u agregace a běžné asociace odlišnými scénáři:

V případě agregace se při tomto procesu zrodí nový vnitřní agregovaný objekt. Tento nový objekt se alokuje, začne žít v paměti a získá se objektová reference na tento objekt. Ta se dosadí do objektové reference u dané agregace vložené do vnějšího objektu.

V případě běžné asociace je nutno si uvědomit, že jediným způsobem, jak dostat „cokoliv“ dovnitř objektu, je poslat objektu zprávu (viz definice objektu jako totálně uzavřená struktura) a to, co má „putovat“ dovnitř, se musí poslat pomocí vstupních parametrů této zprávy. Znamená to, že hodnota objektové reference, která se má naplnit jako běžná asociace, musí tuto hodnotu reference získat zvenku pomocí vstupního parametru zprávy poslané objektu. Logicky z toho vyplývá, že existence běžná asociace vyžaduje existenci alespoň jedné zprávy, která „přinese“ hodnotu objektové reference zvně a pomocí metody ji dosadí do prázdné připravené reference. Takovýchto zpráv může být více a nemusí být pouze jedna.

Poznámka: Pokud použijeme představu existence objektové reference pomocí ukazatelů na objekt, tak si můžeme vysvětlit naplnění objektové reference u agregace a běžné asociace takto:

***Agregace**: Nadřízený objekt požádá prostředí o zrod objektu, který je jeho součástí (tento objekt do té doby není zrozen). Prostředí po zrodu objektu vrátí objektu ukazatel na zrozený objekt a ten si jej dosadí do objektové reference.*

***Běžná asociace**: Asociovaný objekt již existuje. Nadřízený objekt přijme zprávu, jejímž vstupním parametrem je ukazatel na tento existující objekt. Nadřízený objekt si tento ukazatel dosadí do objektové reference.*

Podle tohoto hlediska lze tedy zkontrolovat, zda je návrh asociace anebo agregace správně implementován. Agregace implikuje zrod (a zánik) objektu, což je v kompetenci nadřízeného objektu, asociace implikuje existenci zprávy objektu se vstupním parametrem objektové reference.

Property v OOP

Některé OOP jazyky zavádějí do své syntaxe tzv. *Property* jako zajímavý a efektivní prvek syntaxe. Zavedení *Property* velmi zpřehledňuje zápis kódu a proto je s oblibou používán. Jako analytik jej velmi doporučuji, protože jeho používání vede k velmi silnému analytickému vyjádření významu kódu, tj. sám kód se stává analyticky velmi přehledný a čitelný. Přitom použití *Property* ponechává princip OOP o zapouzdření zachován a nedojde jeho použitím k narušení tohoto principu.

Jak již bylo řečeno, jednotlivé prvky vnitřní paměti objektu jsou zvnějšku neviditelné. Navenek však samotný objekt může přijímat zprávy, na které reaguje. Některé z těchto zpráv mají zvláštní charakter a to z hlediska pohledu na daný objekt. Uvedené „zvláštní“ zprávy lze reprezentovat tak, jako by objekt svému uživateli vykazoval specifické vlastnosti (angl. *Property*), jako jsou „barva“, „stav“, „font“ apod. objektu. Jinými slovy, některé skupiny zpráv pro objekt se jeví jako skutečné vlastnosti, stavy apod., daného objektu.

Například objekt může pomocí zprávy přijmout stav *Zavřen* anebo *Otevřen* a tento vnějšně viditelný stav (pomocí zpráv) reprezentuje navenek jeho vlastnost, tj. *Property*.

Tyto zvláštní zprávy objektu jsou zvláštní pouze tím, že vypadají zvně jako nastavení anebo čtení těchto vlastností objektu. V angličtině se pro metody spouštěné těmito zprávami mnohdy používá předpon *Set* a *Get*.

Pokud jsou v objektu zprávy typu „*Set* vlastnost" a „*Get* vlastnost" (*Set* a *Get*), můžeme v syntaxi daného jazyka, který umí *Property*, nahradit tyto zprávy pomocí jednoho *Property*. Změní se tak pouze syntaxe zaslání těchto zpráv objektu na přehlednější způsob s obdobou přiřazení proměnné, ale významem pro objekt se stále jedná o totéž a to zaslání zprávy objektu:

původní zpráva nastavení: **`Objekt.SET_vlastnost_A ( Parametr )`**

tentýž zápis pomocí property: **`Objekt.Vlastnost_A = Parametr`**

a podobně:

původní zpráva čtení: **`Výstup = Objekt.GET_Vlastnost_A`**

tentýž zápis pomocí property: **`Výstup = Objekt.Vlastnost_A`**

Důležité je vědět, že oba zápisy, jak pomocí zprávy, tak pomocí *Property*, jsou jedno a totéž, tedy pokud v syntaxi jazyka zavedete *Property*, potom je to totéž, jako by jste zavedli dvě zprávy pro objekt *Set* a *Get.* Ten jazyk, který podporuje syntaxi pro zavedení *Property*, nám nabízí pouze jinou syntaxi pro danou funkci *Set* a *Get*.

Zmatením při prvním použití *Property* je v tom, že v použití *Property* je skryto zaslání zprávy a tedy je v něm skryto volání metody objektu. Pod jednoduchou syntaxí vypadající jako přiřazení je skryto volání metody. Předem podotkněme, že není nutné používat *Property*, ale mnohdy je to výhodné. Samotné zavedení syntaxe *Property* totiž zpřehlední kód tím, že se začne objevovat pseudo-syntaxe vyjadřující velmi přesně již kódem analytickou podstatu problému. Pokud nepoužíváte *Property*, můžete dojít díky složeným závorkám k velmi nepřehledným konstrukcím.

Násobnost objektových referencí a její realizace pomocí kolekcí

V předešlých kapitolách jsme se věnovali vazbě objektů, kdy nadřízený objekt drží objektovou referenci na podřízený objekt. V některých případech však z analýzy vyplyne, že by vnější objekt potřeboval držet referenci nikoliv na jeden objekt, ale na celý seznam objektů. Jinak řečeno, z analýzy vyplyne nutnost zavést vztah 1:N a nikoliv 1 : 1, ať už agregaci anebo asociaci.

Například objekt Faktura potřebuje držet Řádky faktury.

Tento vztah se realizuje pomocí tzv. kolekcí (collections), což jsou objekty umožňující držet N objektů. Kolekce jsou velmi užitečné objekty v OOP díky své vlastnosti držet seznam objektů.

Kolekce jsou objekty, které se chovají jako seznamy objektů. Znamená to, že ten, kdo drží objektovou referenci na tuto kolekci, drží seznam objektů a má zpřístupněn seznam objektů, tj. má nějak zpřístupněno N objektových referencí. Důležité je si uvědomit, že zpřístupnění k těmto referencím je pouze pomocí zprávy a tedy funkcionalita kolekce při vydání prvku seznamu je zásadně skryta.

Z hlediska analytického je pro zavedení seznamu objektů tj. kolekce, důležitá jedna okolnost. Prvky seznamu vystupují analyticky jako rovnocenné prvky ve stejné roli. Můžeme hovořit o seznamu prvků, přičemž prvek první má stejný analytický význam , jako prvek druhý, třetí atd. Například práce se

„seznamem osob“ znamená funkcionalitu nad těmito prvky, kde je jedno, zda hovořím o prvním nebo i-tém prvku kolekce (tj. N osob).

Poznámka: Nezaměňujme pojem collection zde jako obecná kolekce v OOP přímo s rezervovaným slovem některých jazyků, jako například Collection z Visual Basicu,což je speciálním případem kolekce v OOP.

V OOP existuje několik druhů kolekcí, které se liší svými vlastnostmi. Uveďme si tyto vlastnosti:

Indexované a neindexované kolekce

V indexované kolekci lze přistupovat k prvkům podle pořadí. Je to podobné jako u konvenční „array“ nebo „table“ struktury. V ne-indexovaných kolekcích nejsou její elementy drženy v pořadí a jsou drženy podobně, jako v množině.

Automaticky a manuálně rostoucí kolekce

Další vlastností, podle které lze rozlišovat typy kolekcí, je schopnost kolekce růst automaticky s přidáváním prvků anebo zda je vyžadována nutnost manuálního růstu. V druhém případě při manuálním růstu kolekce musí mít v tom případě zprávu pro změnu maximálního možného počtu prvků.

Většinou se v OOP používají automaticky rostoucí kolekce, i když samozřejmě režie na obsluhu přidávání prvků kolekce je o něco pomalejší oproti kolekcím, kde je maximální počet prvků stálý a pouze speciálním příkazem (zprávou) se mění tento maximální počet.

Na první pohled by se mohlo zdát, že rozlišení kolekcí na automaticky rostoucí a manuálně rostoucí je technologickým problémem, ale nemusí to být vždy pravda. Pokud z analýzy vyplyne, že kolekce musí mít omezený počet prvků, potom samozřejmě můžeme použít kolekci s omezeným počtem prvků.

Unikátnost prvků v kolekci

V některých kolekcích se zavádí mechanismus rozlišení prvků této kolekce podle tzv. klíče. Každý objekt vložený do kolekce doprovází klíč, což je „datový“ prvek s hodnotou jednoznačně a unikátně se odlišující od hodnoty ostatních klíčů. Přidání objektu do kolekce se tak děje po dvojicích Objekt + Klíč. Pochopitelně nejsou povoleny duplicity klíče při vkládání.

Příklad

Kolekce může přijmout zprávu pro přidání prvku do kolekce, nazvěme tuto zprávu `Add`. Přitom současně s tímto prvkem můžeme u kolekce s klíčem přidat také klíč. Metoda (zpráva) kolekce deklarována jako `Add( aObject As CNeco, akey As .... )`

a použití například takto:

```
Mycol.Add MyObject, MyObject.ID
```

Dvojice „objekt + klíč“ v přidání do kolekce a vnitřní mechanismy v kolekci umožňují dostat objekt zpět z kolekce nejenom podle indexu vložení, ale také podle klíče: Dej prvek vložený s klíčem s určitou hodnotou klíče.

Poznámka: Ve VB deklarovaná kolekce, tj. objekt založený na třídě `Collection`, je tímto typem kolekce.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Násobné indexy v kolekci

Předešlou vlastnost klíčování kolekcí můžeme rozšířit na obecnější vlastnost zavedení násobných indexů v kolekcích (mulitiindex). Podobně jako v teorii databází můžeme k dané tabulce zavádět indexy pro účely sortování, rychlejšího vyhledání, kontroly unikátnosti apod., můžeme obecně v OOP také k prvkům kolekce přidávat indexy, jejichž význam je úplně stejný, jako u tabulek v relační databázi.

Index u tabulky v databázi je vlastně pomocná struktura postavená „vedle tabulky" s realizovanou vazbou do této tabulky. Pomocí této pomocné struktury se například lépe vyhledává, lépe sortuje apod. V každém případě práce s indexy má charakter optimalizace a tedy jako každá optimalizace, která funguje vždy na principu „něco za něco", vede i zavedení k určitému „znečištění" jinak čisté teorie. V případě indexu u tabulky se jedná o duplikaci informace (jakási obdoba porušení normalizace), protože informace se nachází jak v řádcích tabulky, tak ještě jednou v indexu. Ono „něco za něco" v tomto případě znamená sice získat rychlost, ale na druhé straně nutnost obsluhovat obě „totožné" informace a hlídat jejich konzistenci. Pokud přidáváme záznam do tabulky při zapnutém indexu, musí se přitom (nějak) obsloužit také odpovídající index.

Podobně můžeme uvnitř kolekce (zdůrazňuji vzhledem k základnímu pravidlu OOP, kterým je zapouzdření, že indexy jsou umístěny „uvnitř kolekce"!) zavést nějaké „pomocné struktury", které umožní rychlejší vyhledávání, sortování apod. Tato struktura-index má přímý vztah k prvkům kolekce. Tím můžeme realizovat indexy v kolekcích podobně jako v databázi. Je zřejmé, že v tomto případě je předešlá vlastnost unikátnosti přes klíč pouze speciálním případem použití indexu, kdy klíč může být zaveden jako jeden z indexů s požadavkem unikátnosti. Zapouzdření kolekce přímo implikuje uschování mechanismu fungování indexů „dovnitř kolekce", čímž získáváme „neomezené možnosti", jak algoritmy indexování zdokonalovat i u již existujících a již používaných kolekcí.

Typy kolekcí v OOP

V literatuře se můžete setkat s nejčastější používanou definicí následujících typů kolekcí:

Bag	Ne-indexovaná, automaticky rostoucí, duplikace povoleny
Array	Indexovaná, manuálně rostoucí, duplikace povoleny
OrderedCollection	Indexovaná, automaticky rostoucí, duplikace povoleny
Set	Ne-indexovaná, automaticky rostoucí, duplikace nepovoleny
Dictionary	Indexovaná, automaticky rostoucí, duplikace nepovoleny

V každém objektovém programovacím vývojovém prostředí jsou nějak kolekce zavedeny, a nutno podotknout, že se někdy názvy těchto kolekcí v drobnostech liší od předešlé definice. Například ve VB existují dvě „základní" kolekce ze tříd `Collection` (přímo ve VB) a `Dictionary` (například v knihovně scriptu).

Praktické výhody OOP a jejich využívání

Velmi mnoho diskusí v konzultačních cvičeních bylo vedeno na téma: A je OOP a objektové modelování při použití UML opravdu takovou výhodou? Nejedná se pouze o nějaké teoretické plané filozofování?

Musím nejprve odpovědět jednoznačně, že výhody OOP, UML a použití komponentní technologie nejsou žádnou chimérou a jejich výhody se projevují přímo v praxi a také se mi to při mnohonásobných konzultacích ve firmách plně ověřilo.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Povězme si nyní o dvou základních vlastnostech objektově orientovaného přístupu. Právě tyto dvě vlastnosti posouvají tvorbu SW objektovým způsobem oproti strukturálnímu přístupu na diametrálně odlišnou a mnohem vyšší kvalitativní úroveň.

Zapouzdření objektů a konzistence vnitřních stavů objektu

V jednom z konzultačních cvičení byla položena otázka, zda je opravdu nutné zavádět v konkrétním objektově orientovaném jazyce zapouzdření objektů. Většina objektově orientovaných jazyků totiž umožňuje zavést atributy ve viditelnosti `Public` a v tom případě tento atribut může být takto vyveden mimo objekt. Jako příklad můžeme zavést atribut objektu pomocí class modulu ve VB 6.0 takto:

```
Public MyAtribut as String
```

a tímto tento atribut zpřístupnit mimo objekt. Nemuseli bychom v tom případě zavádět žádné metody měnící hodnoty tohoto atributu a vnější prvek by si mohl například číst nebo zapisovat rovnou do atributu.

Nač tedy používat metody objektu a například proč zavádět zdlouhavé `Property`? Nebylo by mnohdy výhodnější zavést uvedený přístup zpublikování atributu (například pro zvýšení flexibility programu, zvýšení rychlosti apod.)? Je tato konstrukce `Public atribut` v OOP vůbec povolena? K čemu vede její porušení, tedy nejedná se o pouhé teoretizování a filozofování v rámci nějaké „čisté nebo nečisté“ teorie OOP?

Předem si odpovězme na tyto otázky a poté si zdůvodníme odpověď:

Opravdu tato konstrukce `Public atribut` není v „čistém“ OOP povolena a dokonce je přímo zakázána.

Narušení zapouzdření objektu vede narušení dvou základních „kladných“ vlastností v OOP a těmi jsou ***konzistence vnitřních stavů objektu*** a ***úplnost informace objektu***. Rozeberme si tento problém v následující kapitole podrobněji z hlediska první vlastnosti nazvanou jako konzistence vnitřních stavů objektu .

Konzistence vnitřních stavů objektu

Protože možné hodnoty atributů reprezentují možné stavy objektu a protože jedinou možností, jak změnit hodnotu atributu, je vyvolat metodu objektu, tak z toho plyne, že jedinou možností jak ***změnit vnitřní stav objektu je změnit jej přes jeho metodu a nijak jinak***.

Tento závěr je velmi důležitý. Ve svém důsledku znamená, že výčtem všech metod objektu také dostaneme výčet všech možností, jak měnit atribut a tedy jak měnit stav objektu. Tímto je sám objekt (a nikdo jiný) plně odpovědný za změny svých stavů. Změny stavů objektů jsou determinovány pouze jeho metodami (což neznamená, že je stav objektu determinován plně, kdo ví, kdo a kdy bude tyto metody volat!).

Tuto plnou odpovědnost objektu sama za sebe a tedy odpovědnost za změny svých vnitřních stavů budeme dále nazývat ***konzistence vnitřních stavů objektu***.

Je pochopitelné, že narušení zapouzdření objektu znamená ztrátu konzistence vnitřních stavů objektu. Pokud povolíme změnu atributu nějakou funkcí mimo objekt, potom samozřejmě existuje „něco“ mimo objekt, co mění stav tohoto objektu bez běhu metody objektu. Objekt může změnit svůj stav, aniž by byl volán a niž by spustil některou ze svých metod. Z hlediska svého chování se objekt začne jevit jako „nelogický blázen“ anebo jako „nadpřirozený jev“. Sám objekt je v klidu, nic se s ním neděje, ale jeho vnitřní stavy se z ničeho nic změní. Takový objekt je nevyzpytatelný a podotkněme, že nevyzpytatelnost není dobrým základem pro tvorbu IS.

Klasický příklad narušení konzistence vnitřních stavů

Většina současných systémů založených na OOP patří z hlediska perzistence dat do tzv. hybridních systémů. Znamená to, že o perzistenci dat se v pozadí stará relační databáze. V OOP má relační databáze mnohem podřadnější význam než ve strukturálním programování, kde je ERD ústředním

diagramem. V OOP se databáze chová vůči žijícím objektům business vrstvy pouze jako úschovna zavazadel. V určitém okamžiku jsou objekty datové vrstvy požádány od objektů business vrstvy o „odložení dat“ podle nějakého algoritmu. V jiném určitém okamžiku jsou objekty datové vrstvy požádány o „vydání těchto dat“ podle téhož algoritmu, jak byly do úschovny, tj. do databáze, vloženy. Podotkněme, že v relační databázi je tímto algoritmem shoda hodnot klíčů mezi tabulkami. Existují i jiné efektivnější algoritmy „uložení a vyložení souvisejících zavazadel“, které zaručují mnohem vyšší rychlost a flexibilitu, například algoritmus tzv. post-relační databází založený na rychlých stromových strukturách.

Zaveďme jako jeden z atributů tzv. identifikátor objektu o označme jej OID (object identifier). Nechť tento atribut má hodnotu stejnou jako ID v tabulce zavedené jako Autoincrement (IDENTITY, SERIAL, AUTOINCREMENT apod.). Přes tuto hodnotu může daný objekt požádat objekt datové vrstvy o vydání nebo o změnu dat. Podobně jiné business objekty provázané s naším objektem a žádající o podobné operace mohou náš objekt požádat o vydání ID pro nutnou vazbu mezi daty jako hodnoty pro cizí klíč (agregované a asociované objekty).

Objekt má data odložena v databázi, avšak to je pouze jeden z možných stavů objektu. Objekt také nemusí mít svůj datový obraz v databázi (tj. nemusí mít svá zavazadla v úschovně), například těsně po svém zrodu. Učiňme dohodu, že objekt, který nemá svůj obraz v DB, bude mít OID rovno –1. Naopak ten objekt, který bude mít svá data v DB, bude mít OID rovno přímo hodnotě ID v tabulce (tj. kladná hodnota).

Scénář zavedení nového objektu do systému může potom vypadat například takto:

- Zrod objektu. Tehdy je OID implicitně rovno -1
- Vyplnění formuláře uživatelem a poté přesun údajů do objektu.
- Zavolání metody INSERT objektu, objekt požádá objekt datové vrstvy o INSERT s návratovou hodnotou IDENTITY (vrací se ID)
- Dosazení navráceného IDENTITY do OID a pokud se operace nezdařila, je OID rovno -2

Všimněme si, že v tomto scénáři se OID mění podle stavu objektu. Buď je OID rovno -1 (objekt ještě není uložen), nebo je OID = -2 (nezdar) anebo je rovno ID v tabulce databáze a v tom případě objekt má svá data odložena.

A nyní se zeptejme: Má smysl toto OID povolit jako `Public`? Pokud tak učiníme, povolíme dosud neznámým funkcím zasáhnout do jinak uzavřených scénářů nabytí hodnoty OID a dovolíme tak narušit konzistenci stavů objektu. Každý analytik by samozřejmě „vyskočil až ke stropu“, kdyby někdo chtěl OID učinit `Public`, vždyť mechanismus chodu OID musí být determinován uzavřenými scénáři chování objektu a ničím jiným. Narušení zapouzdření (a tedy povolení změny atributu zvně) naruší jinak logické chování objektu a následně silně destabilizuje systém. Pokud povolíme atributy jako `Public`, potom ztrácíme kontrolu nad objekty a také nad vývojem systému.

Zapouzdření vede k logickému chování objektů a ke konzistenci stavů, přitom porušení zapouzdření vede k nedeterminovanému chování objektů a v konečném důsledku ke ztrátě stability systému. Systém se zapouzdřenými objekty drží ve svých objektech konzistenci stavů. Proto je takovýto systém mnohem stabilnější a vyvíjí se mnohem snadněji.

Narušení zapouzdření objektu v konečném důsledku znamená ztrátu konzistence stavů objektu. Objekt začne měnit svoje stavy, aniž by v něm proběhly jakékoliv procesy. Takovéto chování objektu není ničím jiným, než změnou stavu bez vnitřního chování, tj. nelogické chování, jako například když řekneme, že „jablko je shnilé, aniž by shnilo“.

Vyhledávání chyb a možnosti změny v programu (flexibilita) se při nedodržení konzistence stavů výrazně zmenšují. Z konzultací jsou mi dobře známy případy, kdy se provádějí změny v programu způsobem: „Nejprve prohledej systém a urči, koho se to týká. Pokud to nejde, tak po provedené změně dělej testy a čekej, co začne padat resp. chovat se jinak, a tam to oprav!“. Slovy klasika takovýto způsob oprav zdá se mi poněkud nešťastným.

Zapouzdření objektů naštěstí nezná takovýto způsob oprav systému. Daný problém změn je díky zapouzdření a konzistenci stavů vymezen pouze v rámci daného objektu (čemuž odpovídá v programu obsah třídy) a to dokonce pouze v oblasti jeho působnosti (netýká se dokonce podřízených objektů pod ním, kterým pouze deleguje činnosti pomoci zpráv). Tento závěr je v konečném důsledku tou příčinou, proč dobře napsaný program v OOP je ve svém vývoji mnohem více stabilnější než program napsaný strukturálně. Objekt se v tomto prostředí nechová „zázračně“, ale kauzálně. Jak známo, kauzalita je pro programy velmi žádoucí a naopak „zázračné“ chování není příliš vhodné pro vývoj a stabilitu informačních systémů. Jak často jsem se setkal s programy napsanými strukturálně se zázračným a nevypočitatelným chováním!

Zapouzdření a úplnost informace objektu

Existuje ještě druhý velmi podstatný závěr plynoucí z vlastnosti zapouzdření a tím je **úplnost informace objektu**. Tato vlastnost souvisí s analytickou přesností a úplností pojmů, které skládají další pojmy a tak tvoří systém.

Zajímavé je, že samotné úvahy o úplnosti informace objektu se budou jevit (jak si ukážeme) jako velmi jednoduché a triviální, ale přitom tyto jednoduché zásady nejsou již z principu ve strukturálním programování dodržovány a málokdo si to uvědomuje. Ve strukturálním programování neexistuje ekvivalent této vlastnosti a tím dochází k zajímavým efektům zalepování a přilepování IS, k tvorbě nepřehledných „lepenců systému“, k nedělitelným molochům a na druhé straně k rozbíjení pojmů, jejich tříštění a jejich nejednotnosti.

Vysvětlení úplnosti informace objektu

Představme se, že jako analytik popisujete nějaký pojem, který bude figurovat jako entita v informačním systému. Jako klasický příklad zvolme pojem Faktura. Můžeme provést následující stručný popis: *Faktura „vidí“ Partnera (tj. Dodavatele resp. Odběratele), obsahuje Datum vydání, Datum dodání, obsahuje Řádky faktury atd...*

Takto bychom popsali informaci Faktury. Je zřejmé, že tímto popisem vymezujeme oblast a hranice pojmu Faktury, tj. to co do ní patří a naopak také pochopitelně tímto popisem automaticky vymezujeme, co do Faktury nepatří. Všimněme si například, že sama Faktura nevidí Zboží faktury, ale Zboží je přístupné a viděno (například) až Řádkem faktury. Podobný popis a úvahy proběhnou v myšlenkách analytika. Konkrétním výrazem těchto na první pohled abstraktních úvah je potom některý z diagramů objektového modelování zapsaný pomocí UML (například objektový model a model tříd).

Přitom se může zdát jako velmi triviální následující myšlenka: Když buduji pojem Faktura, který má v OOP díky povaze objektů poté i svůj přesný obraz v objektech, tak tímto pojmem mám na mysli vše, co má tento pojem obsahovat. Samozřejmě současně s tímto pojmem nemám na mysli to, co tento pojem obsahovat nemá. Celá Faktura jako objekt je opět složena z jiných pojmů, následně objektů (do kterých v této úvaze nyní nevcházím, například Řádky faktury apod.) a tím tento pojem Faktury vytváří určitou oblast svého existenčního vymezení. Jednoduše řečeno: „Faktura je jako objekt toto a toto“ a tím automaticky není „vše ostatní“. Když se řekne Faktura, je zřejmé, že tím mám na mysli celý pojem Faktury. Podobně v OOP, když „držím“ objekt Faktura, mám vše, co s sebou Faktura nese. Tato vlastnost se projeví přímo v kódu, například takto požádám Fakturu o IČO Odběratele přes property:

```
txtICO.Text = EditovanaFaktura.Odberatel.ICO
```

Je to velmi triviální úvaha, ale jak často není takováto jednoduchá zásada v objektovém programování (natož strukturálním) dodržována! Tato myšlenka je vyjádřením úplnosti pojmu Faktury.

Připomeňme, že ve strukturálním programování složeném z dat a funkcí nemá tato myšlenka úplnosti resp. neúplnosti pojmu smysl. Program je totiž složen z funkcí a dat a nikoliv z objektů. Určitou obdobou se může jevit zavedení modulárního programování, tj. zavést moduly (někdy nazývané knihovny), v kterých jsou něco jako „informace“ daného pojmu ve tvaru funkcí a dat zavedeny a

odloučeny od ostatních částí systému. Avšak strukturální modulární programování má svá omezení daná zejména tím, že velmi těžko vytvoříte několikanásobnou instanci jednoho modulu (několik instancí od sebe identifikovaných). Pokud například vytvoříte modul „Řádek faktury“, velmi obtížně se v tomto modulárním prostředí tvoří „Řádky faktury“, tj. několik instancí modulu vedle sebe. Modul je totiž zaveden pouze jako jeden, čímž vznikají problémy s multi-instancemi. Souběžně s tím vzniká problém identifikace těchto entit ze stejného modulu (jeden řádek, druhý řádek, faktura číslo X, faktura číslo Y atd.). Tento problém se musí řešit pomocí datových vztahů. Podotkněme, že pokud zavedete modul s možností několika instancí, přecházíte tím do oblasti OOP a tento modul odpovídá možnému implementačnímu zavedení třídy.

Klasickým příkladem je rozdíl mezi BAS a CLS modulem ve Visual Basicu. Můžeme BAS modul chápat jako případ třídy s možností vytvořit pouze jednu instanci, kterou tímto nemusíme identifikovat od jiné instance a tedy nemusíte tuto instanci oslovovat názvem (vznikají tak systémové funkce jako metody objektu systému).

Zaveďme název pro právě vypozorovanou a jednoduchou vlastnost objektů a nazvěme ji ***úplnost informace objektu***. Definujme tuto vlastnost tak, že ***objekt obsahuje všechny informace včetně funkcionality, které vytvářejí daný pojem objektu a nic víc***.

Narušení zapouzdření a neúplnost informace objektu

Samotné narušení zapouzdření objektu automaticky vede k porušení vlastnosti úplnosti informace objektu, protože samo o sobě se tím předpokládá, že objekt není úplný.

Představme si, že vytvoříte objekt s `Public` atributem. Pochopitelně to činíte proto, aby něco mimo objekt mohlo s tímto atributem pracovat, jinak byste zavedli tento atribut jako `Private` atribut. Předešlá věta není ničím jiným, než synonymem pro narušení úplnosti informace objektu. Předpokládáme totiž, že **existuje něco mimo objekt, co má s tímto atributem pracovat.** To je přesný opak k pojmu úplnosti informace objektu, kde se předpokládá, že vše, co s vnitřkem objektu pracuje, je uvnitř objektu.

Uveďme si klasický příklad na úplnost a neúplnost informace objektu:

Pokud si přilinkujete do svého programu pomocí Automation technologie objekt aplikace Excel, dostanete tak k dispozici **úplný** Excel objekt, tj. úplný ve své funkcionalitě. Není nic mimo něj, co byste si ještě museli přilinkovat. Vytvoříte jednu instanci a máte k dispozici celý Excel. Pokud pracujete s jedním listem spreadsheetu Excelu, potom dostanete k dispozici všechnu jeho informaci, podobně když „držíte“ jednu buňku atd. Každý objekt obsahuje svou informaci a nic mimo něj. Pokud zavedete funkci mimo objekt pracující s něčím v objektu, potom zavádíte něco, co do objektu fyzicky nepatří a mělo by tam patřit.

Nepříznivé důsledky narušení úplnosti informace objektu pro vývoj IS

Existují dva základní nepříznivé důsledky nedodržení vlastnosti úplnosti informace objektu a podotkněme, že se jedná o vlastnosti příznačné pro strukturálně napsané programy :

- Systém se stává silně nepřehledným, nestabilním a současně neodolným vůči změnám. Jakákoliv změna vede k následným změnám (bůhví kde) mimo změněnou oblast
- Systém vytváří lepence a molochy dále již nedělitelné. Pokud chcete dodat pouze část systému, zjistíte, že musíte k zachování funkcionality dodat další a další části, které s tímto problémem zdánlivě nesouvisejí, ale musejí být také dodány. Díky rozbití pojmů a jejich nevyhranění se funkcionalita „rozprskla“ přes celý systém. Velmi obtížně se v takovém prostředí zavádějí komponenty (problém „siamských dvojčat“, kdy nelze „odoperovat“ od sebe části systému propojené podobně jako siamská dvojčata)

Pokud používáme OOP, potom díky zapouzdření mají pojmy „ostré“ kontury. Je pouze otázkou kvality objektové analýzy, zda vytvářené pojmy (budoucí objekty) obsahují to, co obsahovat mají a neobsahují to, co obsahovat nemají.

Ve strukturálním programování se již samou podstatou problému setkáváme s otázkou, co vlastně reprezentuje daný pojem s vymezenou funkcionalitou. Strukturální programování vede díky public datům automaticky k problémům neúplnosti informace objektů (protože samy objekty jako uzavřené struktury neexistují) s nepříznivými důsledky uvedenými v předešlém výčtu.

První kroky k modelování ještě než zahájíme v UML

Pro vysvětlení objektového modelování představme si, že bychom vytvořili informační systém jinou technologií, než je běžné pomocí SW, například by se jednalo o obdobu písemné kartotéky.

Každá informace v kartotéce je uložena na kartičce. Na této kartičce jsou jednak nějaké údaje a také algoritmy nad těmito údaji. Pokud čteme informaci z této karty (jak údaje, tak algoritmy), tak nemáme možnost v té chvíli „vidět“ jinou kartu. V této technologii však existuje možnost odskoku na jinou kartu a požádat tuto kartu o funkcionalitu, tj. na kartě, kterou čteme je možné vidět „odkaz“ (například pomocí „nitě“ apod.) na jinou kartu. Tento odkaz na jinou kartu je dvojí povahy:

- Buď je karta, na kterou se ukazuje, součástí karty nadřízené karty, například vnitřní karta je „vlepena“ do vnější karty,
- Anebo si na tuto druhou kartu pouze ukazuje, tj. odkazovaná karta je vlepena jinde a pouze se na ni propojíme „nití“.

Tyto dva způsoby odkazů odpovídají dvěma způsobům vazeb mezi objekty: Agregace a běžná asociace. Dohodněme se, že agregaci v těchto kartičkách budeme značit černě vyplněným bodem a asociaci šipkou.

Kromě toho existuje také možnost vytvořit takovéto propojení na skupinu karet (seznam karet), což odpovídá realizaci vazby 1:N, což znázorníme násobným namalováním několika kartiček.

Důležitý princip „této hry“ je v tom, že pokud jsme v daném algoritmu na určité kartě, tak nevidíme nic jiného, než to, co je napsáno na této kartě a také vidíme, jaké jsou na této kartě odkazy na jiné karty, ale nic víc. Do odkazovaných karet nevidíme, dokud do nich „neskočíme“.

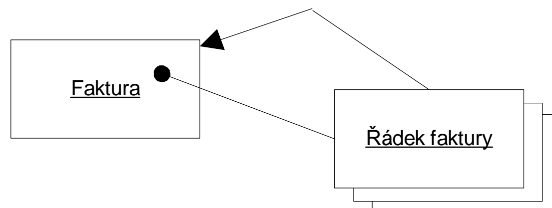
Například z těchto našich úvah vyplývá, že podstatou vazby mezi objekty je jednosměrnost a pokud chceme provést obousměrnou vazbu, musíme provést obě vazby mezi kartičkami tam i zpět („skok tam a skok zpět“).

Je zřejmé, že každá kartička s nabízenou funkcionalitou odpovídá jednomu objektu. Pokud bychom chtěli zavést třídu, tak ji můžeme chápat jako kopyto pro tvorbu zatím nevyplněných kartiček (formulář pro prázdnou kartičku).

V tomto modelu si můžeme znázornit různé vztahy mezi objekty velmi efektivně, srozumitelně a přehledně.

Příklad: Faktura obsahuje N Řádků faktury. Každý řádek faktury vidí svou Fakturu, kam patří. Pokud jsme v informačním prostoru Faktury (na její kartičce), vidíme kromě jejích údajů na ní napsaných (atribut Datum splatnosti apod.) také objektovou referenci na seznam Řádků faktury jako vztah agregace. Pokud se nacházíme v informačním prostoru jednoho Řádku faktury, tak vidíme (kromě jiného) jednu objektovou referenci na Fakturu, je to ta Faktura, která drží seznam řádků, kam patří náš řádek.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 18 : Vztah Faktury a Řádků faktury

Je zde třeba zdůraznit vcelku triviální fakt: Na předešlém diagramu nehledejme žádné složitosti. Vyjadřuje velmi jednoduchou skutečnost, že nějaká faktura obsahuje (jako svou agregaci) seznam řádků faktury a každý řádek faktury zná svou fakturu, kam patří (jako svou běžnou asociaci).

Je velmi praktické vytvářet si takovéto pomocné diagramy a z vlastní zkušenosti mohu potvrdit, že tyto pomocné diagramy se stávají základem pro další „skutečně modely v syntaxi UML“.

Konceptuální diagram jako pomocný diagram

Uvedený diagram neodpovídá žádnému diagramu UML, nejblíže má k tomuto vyjádření tak zvaný *Object diagram*, který však nevyznačuje směr a nerozlišuje rozdíl mezi agregací a běžnou asociací. Pro naši „vnitřní potřebu“ budeme tento typ diagramu dále nazývat **konceptuálním diagramem**. Připomeňme a přitom zdůrazněme, že tento diagram není součástí UML. Z hlediska UML se jedná o modifikovaný ***Object Diagram***, na rozdíl od něj navíc vyznačujeme směr a rozlišujeme agregaci a běžnou asociaci.

Námi zavedený konceptuální diagram budeme používat v časných stádiích analýzy, kdy hledáme pojmy (koncepty) pro vyjádření chodu a činností v systému. V dalších fázích již konceptuální diagram můžeme odložit, protože nalezneme vyjádření v některém z Class diagramů, ze kterého lze konceptuální diagram poté odvodit.

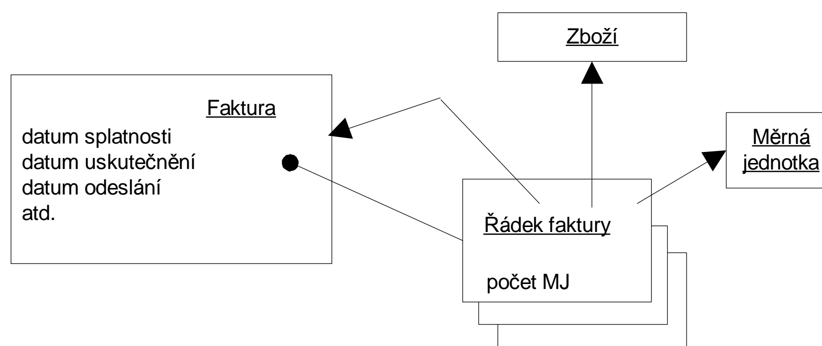
Poznámka: I když tento diagram nepatří do syntaxe v UML a je námi zavedený jako „navíc“, při konzultacích a v analýzách systémů se mi tento diagram velmi osvědčil v prvních stádiích analýzy, zejména při odhalování pojmů a objektů v IS. Z tohoto hlediska jej považuji za velmi přínosný. Po nalezení daných pojmů a vytvoření Class diagramu můžeme tento diagram odložit a dále nepoužívat. Ale při zahájení prací v analýze jej považuji dokonce za nezbytný! Důvod je prostý: Při tvorbě tohoto diagramu se pracuje pouze s pojmy (podobně jako s rolemi), ale nehledají se zobecnění ve třídách.

První model jako klasický ukázkový příklad na re-use v OOP

Pomocí již zavedeného konceptuálního diagramu můžeme začít modelovat.

Jako první příklad si ukážem, jak vypadá klasický re-use v objektu Faktura a Řádky Faktury pomocí námi zavedeného konceptuálního diagramu. Spolu se slovním vyjádřením budeme malovat také diagram, který těmto pojmům odpovídá.

Jak víme, tak jedna Faktura (jedná se o „nějakou" konkrétní určitou Fakturu nikoliv o třídu) obsahuje Řádky Faktury. Řádek Faktury si ukazuje na jedno konkrétní Zboží ze Seznamu Zboží. Dále si Řádek faktury ukazuje na jeden objekt Měrná jednotka ze Seznamu měrných jednotek a obsahuje Počet měrných jednotek jako svůj údaj. Namalujme tento jednoduchý konceptuální diagram odpovídající těmto větám:



obrázek 19 : První model pro Řádek faktury

Poznámka: V modelu není řešena otázka následných změn faktury vzhledem k historii ostatních entit. Pokud někdo změní Zboží, projeví se to také v již vydaných Fakturách.

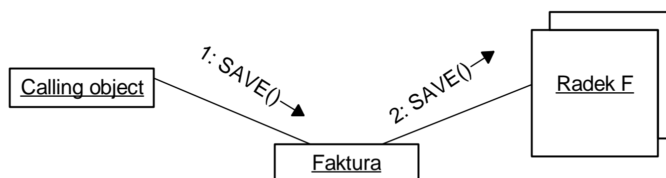
Všimněme si na tomto ukázkovém diagramu několika zajímavých skutečností:

Diagram vyjadřuje velmi přesně vztahy mezi pojmy. Pokud diagram vyjádříme pouze větami, tak faktura obsahuje řádky faktury, každý řádek vidí „svou fakturu", dále vidí jednu měrnou jednotku a jedno zboží. Každý pojem z této struktury je odpovědný za svůj obsah, například Měrná jednotka obsahuje informace jako Zkratku měrné jednotky (například „kg"), Název měrné jednotky („kilogram") apod. Zboží obsahuje Druh, Název, Jednotkovou cenu apod. Každý z pojmů je kompetentní za informace ze svého okruhu působnosti. Všimněme si, že podrobnosti typu kg, kilogram, druh apod. (tj. „údaje u vnitřků pojmů") nemáme zatím v diagramu vyjádřeny, resp. se uvádějí pouze jako příklad, co může daný pojem obsahovat. Navíc každý pojem může dále obsahovat další pojmy, které tu zatím nemáme znázorněny. Tento diagram tak vyjadřuje abstraktnější rovinu pouze v pojmech. Víme, že pojmy obsahují podrobnosti, ale ty nás v této chvíli příliš nezajímají (ale víme, že tam tyto detaily jsou). Je možné, že některé podrobnosti přibudou a jiné detaily ubudou, ale vztahy mezi pojmy jsou v této rovině abstrakce již takto vyjádřeny. Právě tato skutečnost spočívající ve vymezení pojmů, je pro objektové modelování podstatnou. Tento pohled je velmi blízký „normálnímu" pohledu člověka při abstraktním uvažování, když se vymezují určité pojmy a když se vymezují vztahy mezi nimi.

Je třeba si uvědomit, že se v tomto případě se vztahy mezi pojmy nechápou pouze jako vztahy mezi „mrtvými" informacemi. Každá z „kartiček" (pojmů, objektů apod.), nejenom že nese své nějaké údaje, ale také ovládá nějaké umění (například umí se uložit, tj. požádat úschovnu zavazadel o odložení dat).

Příklad: Nazvěme zprávu pro fakturu pro uložení jako Save. Uložení celé faktury potom může znamenat sekvenci poslání zpráv podle obrázku (podle jednoho z možných scénářů):

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 20 : Sekvence zpráv Save

Například v tomto modelu „zavolat“ *Save* pro Fakturu (poslat jí zprávu) znamená provést uložení vlastních údajů a poté zavolat metodu, tj. poslat zprávu Save všem řádkům faktury (například v cyklu apod.)

Dalším důležitým zastavením u tohoto příkladu je úvaha spočívající v otázce: Jak se vlastně „naplní“ správná objektová reference na správný objekt, tj. v našem příkladu reference z Řádku faktury na tu „správnou“ Měrnou jednotku nebo na to „správné“ Zboží?

Za prvé je třeba připomenout, že vztah od řádku faktury směrem k měrné jednotce anebo zboží je běžnou asociací. Znamená to, že řádek faktury musí mít „umění“ přijmout objektovou referenci zvně (tj. musí mít zprávu pro příjem této reference) a to jak pro zboží, tak pro měrnou jednotku. Většinou se tato reference zavádí jako property (viz předešlé kapitoly o property), například v pseudokódu VB:

```
Property Set MyZbozi (a_Zbozi As CZbozi)
    Set mMyZbozi = a_Zbozi
End Property
```

Druhou věcí, kterou je třeba si v této souvislosti uvědomit, je ta skutečnost , že k naplnění těchto referencí může dojít několika různými scénáři podle kontextu (tj. na základě souvislostí a okolnostech), kde a kdy je třeba tyto objekty naplnit.

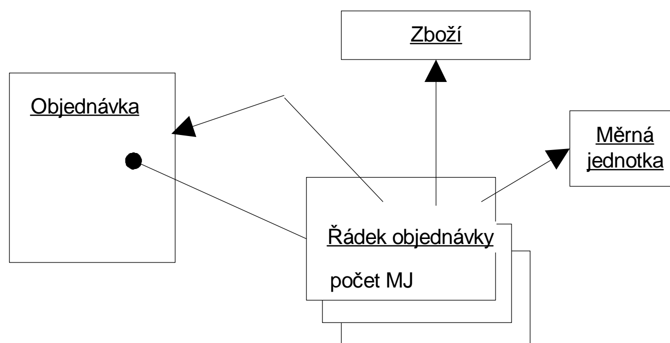
Příklad: Pro osvětlení této situace, tj. naplnění reference podle kontextu, uveďme naplnění referencí u nového řádku faktury v kontextu formuláře vyplněném obsluhou. V tom případě se

- Obsluze zobrazí seznam zboží,
- Obsluha vybere zboží a tím vybere jednu konkrétní objektovou referenci na jedno konkrétní zboží. To je to „správné“ Zboží.
- Tato konkrétní reference se stává vstupním parametrem zmíněné metody property `MyZbozi`.

Dalšími možnými kontexty jsou naplnění faktury z textového souboru, z databáze apod.

Pro náš příklad pro nalezení re-use je důležitá následující úvaha. Představme si, že v další části systému se naleznou další vztahy mezi dalšími novými pojmy. Například při práci s Objednávkou se

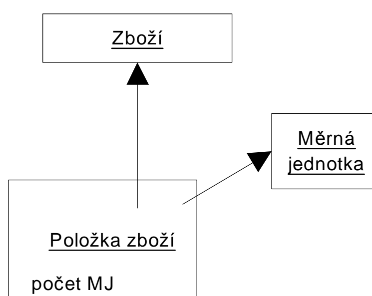
zjistí, že Objednávka obsahuje Řádky objednávky a každý řádek (kromě svých vlastních informací) vidí jednu Měrnou jednotku a jedno Zboží. Je pochopitelné, že jeden konkrétní řádek objednávky se liší od jednoho konkrétního řádku faktury, ale podobnost pojmů je v tomto příkladu veliká, porovnejte následující a předchozí obrázek:



obrázek 21 : Řádky objednávky

Tato podobnost není náhodná, avšak z hlediska operace re-use zatím nevyužitá. Existuje několik možných způsobů (minimálně dva), jak v tomto případě zavést re-use, ukažme si ten jednodušší a později, až probereme vztah generalizace a specializace, také druhý způsob.

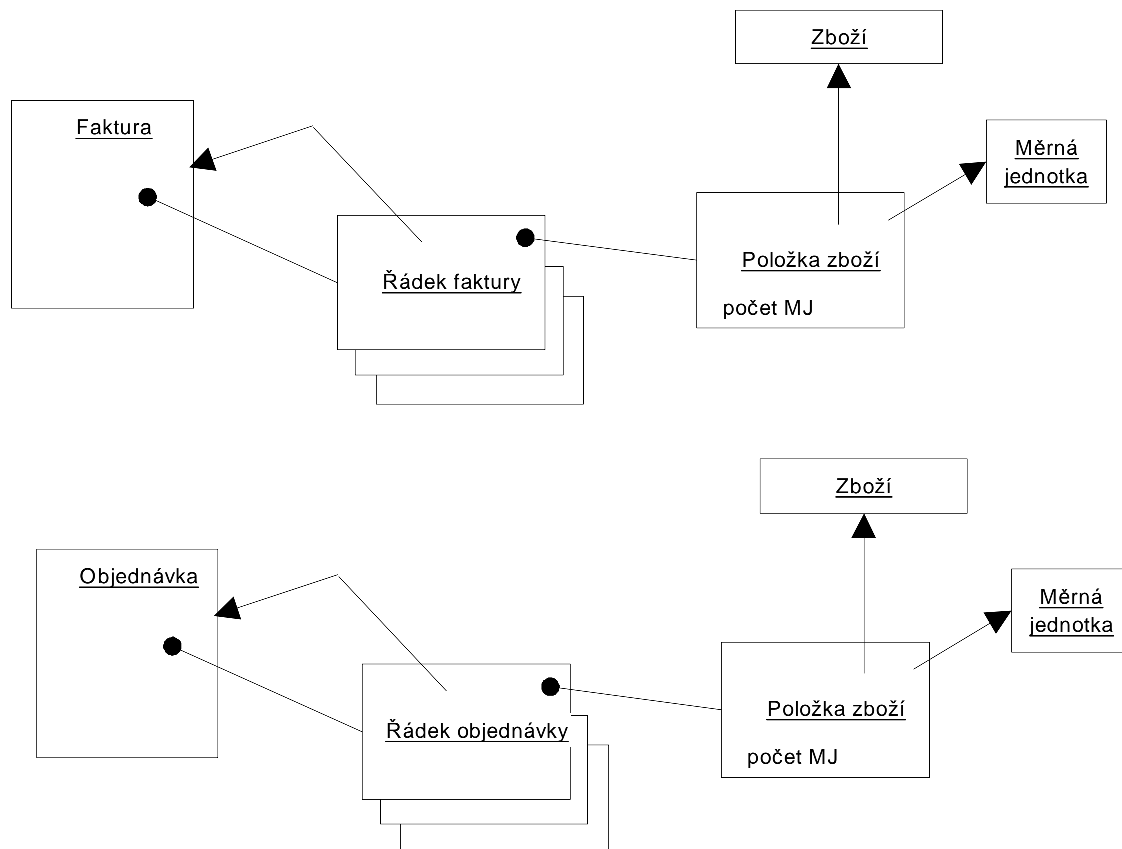
Zaveďme nový pojem: Nazvěme jej například Položka zboží. Představme si pod ním jeden „balíček zboží", který obsahuje Počet měrných jednotek, ukazuje si na Měrnou jednotku a ukazuje si na své Zboží. Zavedení této Položky zboží odpovídá nově zavedenému pojmu, který do této chvíle neexistoval. Jeho vymezení vede k znovu použití tohoto pojmu všude tam, kde je třeba. Návrh v našem konceptuálním modelu se můžeme změnit takto:



obrázek 22 : Zavedení Položky zboží – "balíček zboží"

a tuto položku zboží „vložíme" jak do Řádku Faktury, tak do Řádku Objednávky:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 23 Položka zboží se používá jak v kontextu faktury, tak objednávky

Předešlý obrázek zřetelně ukazuje na jeden z možných re-use. Všimněme si pojmu Položka zboží v obou modelech, jak pro Řádek objednávky, tak pro Řádek faktury. Zavedení tohoto pojmu Položka zboží již v konceptuálním diagramu se chápe jako shoda dvou pojmů znovu použitelných v různých kontextech, tj. v různých souvislostech, zde například jako balíček zboží jak v Řádku objednávky, tak v Řádku faktury. Použití tohoto pojmu se nabízí i jinde, například v dopravě podniku ve Zboží na voze nebo ve Zboží ve skladu apod. Samozřejmě z hlediska objektů je Položka zboží v Řádku objednávky jiným objektem, jinou instancí, jinou „kartičkou“, než je Položka zboží v Řádku faktury, ale mají úplně stejné vlastnosti (obsahuje Počet měrných jednotek, vidí Měrnou jednotku a vidí Zboží). Tato vlastnost se přenáší až do zavedených tříd, protože obě Položky zboží v obou případech patří do téže třídy.

Uvedenou Položku zboží (možná rozšířenou o další pojmy, jako obaly, druhy obalu, jazyk na obalu atd.) můžeme použít ve velmi mnoha případech a dosadit ji všude, kde třeba. Uveďme příklady použití, kde se všude může Položka zboží dosadit:

- Položka zboží v řádcích různých dokladů jako jsou: řádek expedičního příkazu, řádek objednávky, řádek faktury atd.
- Položka zboží na skladě, pohyby zboží na skladě, atd.
- Položka zboží na voze v dopravě, na výstupu z firmy atd.
- aj.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

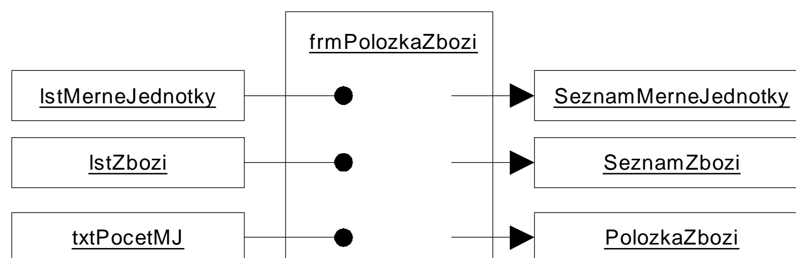
Re-use se projevuje v našem návrhu velmi jednoduše. Umění všech položek zboží jako „balíčku zboží" je stejné. Například naučíme (pomocí třídy) Položku zboží zprávě a také metodě Save, která následuje hned po vyplnění všech vazeb (je naplněna reference na Měrnou jednotku a Zboží) a současně je vyplněn Počet měrných jednotek. Velmi často se opakující se scénář je poté následující:

1. Nadřízený objekt (například Řádek faktury, Řádek objednávky atd.) nechá vzniknout novému „svému" objektu Položka zboží.
2. Obsluze se zobrazí GUI prvek (formulář, OCX apod.), na něm se zobrazí Seznam zboží, Seznam měrných jednotek a editace Počtu měrných jednotek.Obsluha vybere Zboží, vybere Měrnou jednotku, zadá editací Počet měrných jednotek
3. Až dojde k ukládání nadřízeného objektu (Řádku objednávky, Řádku faktury atd.), tak tento nadřízený objekt pouze pošle zprávu Save objektu své Položce zboží.

Přímo v nějaké konkrétní implementaci se tento re-use může projevit například takto:

Systém je navržen ve VB 6.0. Editace nové položky zboží probíhá přes formulář `frmPolozkaZbozi`, který se zobrazuje v modálním módu, obsahuje dva prvky `Listview` (jeden pro výběr Měrné jednotky a druhý pro výběr Zboží) a jedno editační pole `TxtBox` pro počet měrných jednotek. Tento formulář má asociaci na tři objekty, na Seznam měrných jednotek, na Seznam zboží a na editovanou Položku zboží. První dva objekty (Seznam měrných jednotek a Seznam zboží) jsou „zdrojem" pro zobrazení a třetí objekt je naopak objektem, který bude „přijímat" informaci (do něj se dosadí vybrané objektové reference a zadaná hodnota Počtu měrných jednotek).

Symbolicky můžeme náš návrh namalovat takto:



obrázek 24 : Implementace návrhu pro formulář editace položky zboží ve VB 6.0

V uvedeném diagramu (který už není analytické povahy, protože obsahuje implementační detaily) jsou znázorněny GUI objekty: Formulář, List zboží, List Měrných Jednotek a Editační pole pro Počet měrných jednotek, a jsou zde znázorněny také tři business objekty: Seznam měrných jednotek, Seznam zboží a Položka zboží.

Všechny objekty mezi sebou spolupracují určitým způsobem a to tak, že oba seznamy nabízejí pro oba Listy nějakým způsobem údaje pro zobrazení seznamů, přičemž pokud obsluha vybere určitý řádek a odsouhlasí výběr, potom seznam dodává odpovídající vybraný objekt. Tento objekt se dosadí do objektu Položky zboží přes odpovídající property objektové reference. Obsluha také provede editaci Počtu měrných jednotek, které se dosazují do položky zboží. Část pseudo-kódu po odsouhlasení stiskem buttonu OK obsluhou může například vypadat (bez kontrol) takto:

...část kódu formuláře po stisku OK (bez kontroly)

```
With PolozkaZbozi
    .PocetMJ = txtPocetMj.text
    Set .Zbozi = KolekceZbozi.Item(ListZbozi.SelectedItem.Tag)
    Set .MernaJednotka = KolekceMJ.Item(ListMJ.SelectedItem.Tag)
    Unload Me
End With
```

V pseudokódu jsme symbolicky označili vybraný prvek Listu jako `SelectedItem` a současně označili vazbu mezi prvky v Listech k prvkům v business kolekcích přes hodnotu `Tag`, tj. hodnotou přiřazenou ke každému prvku Listu a která odpovídá klíčům v business kolekcích.

Re-use se projeví v tom, že ten, kdo programuje nadřízený objekt obsahující Položku zboží (například navrhujeme Řádek faktury, Řádek objednávky apod.), již vůbec neprogramuje podřízenou Položku zboží a to ani ve scénářích jako je tento, které jsou již připraveny. Založení a editace nové Položky zboží podléhá nyní tomuto scénáři:

Jsme v novém Řádku faktury. Nechejme vytvořit novou Položku zboží agregovanou v tomto Řádku faktury. Zavolejme formulář `frmPolozkaZbozi`, do kterého dosadíme za editovanou Položku zboží právě vytvořenou Položku zboží. Tím „končíme“ s Položkou zboží a ostatní je re-use.

Jinak řečeno každý, kdo potřebuje editovat položku zboží, zavolá pouze `frmPolozkaZbozi` zahájením scénáře například `frmPolozkaZbozi.Show vbModal`. Protože je tento postup vždy stejný, bude se obsluze objevovat vždy jeden a ten samý formulář ať už pracuje s Položkou zboží v Řádku faktury, Položkou zboží ve skladu, Položkou zboží při manipulaci na voze atd.

K tomuto příkladu několik poznámek:

1. *Není vždy vhodné používat modální okna pro editaci entit. Pokud nechceme použít „samostatný formulář“ pro editaci položky zboží a přitom použít re-use (hovoříme stále o implementaci ve VB 6.0 a nikoliv VB 7.0!), mohli bychom s výhodou využít technologii OCX a vytvořit „část formuláře“ jako komponentu, kterou lze dosadit do jiných formulářů.*
2. *V tomto příkladu jsme chtěli poukázat na sílu re-use v OOP. Mnoho diskusí se vede nad uvedenou spoluprací mezi GUI prvky a business prvky systému, tj. jak například uvedený GUI prvek získá údaje pro zobrazení, jak se naplní apod., protože tam se může při nesprávném scénáři naplnění (o kterém tady nebyla řeč) objevit kritické místo. Tomuto problému se budeme věnovat v dalších kapitolách v různých modelech a přístupech při tzv. optimalizaci aplikace.*

Re-use versus RAD

RAD je zkratkou pro tzv. *„Rapid Aplication Development“*. Jedná se o zvláštní druh vývoj SW aplikace, kdy se pomocí poměrně dost sofistikovaných nástrojů jednoduchým způsobem aplikace vyvine velmi rychle „naklikáním“ GUI prvků na formuláře. Tyto GUI prvky mají přímou vazbu do dat přes nějaký zavedený objekt datového zdroje, například *Datasource*. Vývojové prostředí tak provádí automaticky určité kroky, které se jinak provádějí ručně.

Výhodou tohoto postupu je bezesporu rychlost. Nevýhodou je však mnohem menší re-use mezi již hotovými prvky, například re-use mezi částmi systému, resp. z projektu do projektu apod.

Představme si, že bychom měli na výběr ze dvou možných technologií tvorby softwaru:

- první z nich je charakterizován velmi velkou rychlostí tvorby prvků pomocí určitých „chytrých“ automatizovaných mechanismů vývojového prostředí (to je případ RAD), avšak pokud vytvoříme

určitý celek, nemůžeme jej takřka vůbec znovu použít. Pokud se něco opakuje být s malými odlišnostmi, musíme takovýto podobný prvek znovu vyvinout, tedy nemůžeme provést re-use naší práce. Nutno však podotknout, že pomocí automatických procesů vyvíjíme dílčí části velmi rychle. Tento způsob vývoje je tedy charakterizován tím, že ve vyvíjené aplikace není zaveden takřka žádný re-use, ale tato nevýhoda je vyvážena možností poměrně rychle vytvořit opakující se část aplikace. Samozřejmě, pokud se cokoliv v aplikaci dodatečně změní resp. nalezne chyba, musíme tuto chybu anebo změnu opravovat a zavádět tolikrát, kolikrát jsme (i když rychle) část aplikace znovu tvořili.

- druhý možný způsob vývoje spočívá v možnosti tvorby systému mnohem pracněji, protože nepoužíváme automat, ale pracujeme v tvorbě SW doslova „na míru" a kód píšeme „ručně". Avšak máme k dispozici tak maximální re-use, že žádnou práci neprovádíme ***nikdy dvakrát*** (poznámka: nikdy, pokud jsme důslední, pokud umíme OOP a pokud umíme řídit projekty). Systém je z tohoto hlediska čistý, vše je tvořeno pouze jednou. Software je přehlednější, průhlednější a stabilnější, flexibilnější na opravy atd. Je zřejmé, že maximální re-use postupně urychluje vývoj aplikace, protože čím bohatší máme již hotovou knihovnu, tím rychleji se nám tvoří další prvky z této knihovny odvozené.

Otázka zní: Který z těchto dvou přístupů je lepší?

Samozřejmě neexistuje jednoznačná odpověď. Základním kriteriem pro posouzení této otázky je jednak koncepce firmy, jednak koncepce projektů, a také velikost projektů. Podívejme se na tuto otázku z hlediska nákladů a zisků, což je pro firmy to základní hledisko.

- Postup pomocí RAD je výhodný tam, kde se vyvíjejí jednoduché a co do počtu entit malé aplikace (například do 10-20 tabulek apod.). Jedná se tedy o aplikace a projekty firmy vystačující s malým re-use. Pokud si sami pro sebe vytvoříte malou aplikaci s pěti tabulkami a třemi formuláři, tak bude plně dostačující vytvořit tuto aplikaci pomocí RAD. Z hlediska nákladů není také zanedbatelná otázka zaučení a nutná kvalifikace pracovníků. Pro vývoj RAD vystačí průměrný až podprůměrný programátor, který se rychle orientuje v daném automatizovaném vývojovém prostředí.
- Oproti tomu druhý (OOP) přístup je nezbytný tam, kde narůstá velikost a složitost problému. Pokud budete vyvíjet aplikaci (ať už desktop anebo na síti) s větším počtem entit (například 25 a více), zaručeně začnete mít s přístupem RAD problémy. Počne se projevovat velmi nepříjemný „synergetický" efekt nárůstu problémů s nárůstem entit v RAD. Ověřil jsem si v praxi (a je to též uváděno v zahraniční literatuře), že ve vývoji pomocí RAD platí zajímavé „synergetické pravidlo": S větším počtem entit neroste v RAD pracnost systému lineárně (tj. úměrně počtu), ale mnohem, mnohem rychleji, takřka exponenciálně. Znamená to, že nemůžete u RAD použít jednoduchou metodu přidávání entit bez následných problémů. Pokud máte v systému 20 tabulek a rozšíříte je na 30, nepřibude vám 1/3 práce, ale práce přibude mnohem více. Dokonce od určitého momentu „synergetický" efekt vztahu mezi entitami přeroste u desítek a stovek tabulek do velmi silných problémů. Potom je velmi složité a (dokonce až nemožné) systém udržovat ve velmi dobrém stabilním chodu. Oproti tomu nevýhodou použití OOP a UML je určitá „zdánlivá pomalost výsledků" v počátcích vývoje, což je nepříjemný fakt znervózňující zejména vedoucí pracovníky. Nejprve se totiž hmatatelné výsledky jakoby nedostavují.

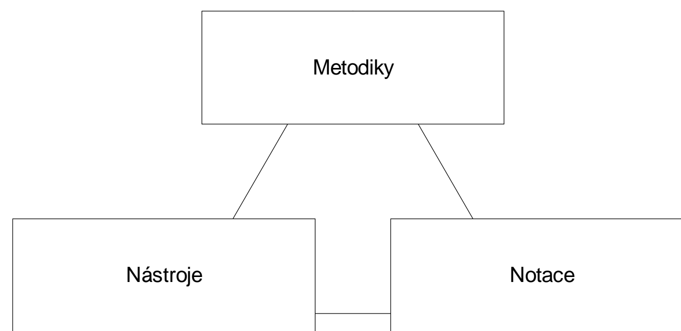
Poznámka: V jedné bratislavské firmě tvořící systémy pro leasingové společnosti, kde jsem konzultoval při zavádění OOP, UML a COM, se postupovalo podle doporučení pro vedení projektu a přechodem na OOP podle syntaxe UML. Zajímavý byl jeden postřeh a následné zjištění těchto pracovníků: Projekt se v OOP rozjíždí jako lokomotiva: Nejprve „pomalu", protože se hned nekóduje, ale provádí analýza v UML, vytváří se modely v UML, navrhují se vzory scénářů pro spolupráce objektů v opakujících se situacích atd. Poté se projekt „běží" z hlediska výstupů stále větší rychlostí.

Paradoxně v projektech tvořených pomocí RAD resp. strukturálně je situace přesně opačná: Projekt na začátku velmi rychle podává výsledky (a to dokonce přímo v kódu). Následná euforie je brzy vystřídána skepsí, projekt se postupně komplikuje a zpomaluje a to mnohdy až k úplnému zadrhnutí. Z hlediska pracnosti s dalším vývojem pracovníci pracují stále s vyšším výkonem, ale přitom práce stále přibývá.

Dost velkou nevýhodou použití OOP, přesněji překážkou, může být v kvalifikaci pracovníků. Pro firmu je z tohoto hlediska mnohem jednodušší nasadit RAD než OOP. Tedy určitou brzdou přechodu na OOP se může stát průměrnost programátorů, kteří z neznalosti nejsou schopni využít všech výhod re-use v OOP a tedy ztrácejí výhody tohoto přístupu. Právě velmi častým jádrem diskusí o výhodách OOP jsou neznalosti, které jsou způsobeny nevyužitím výhod re-use v OOP. Přístup pomocí OOP je totiž paradoxně z pohledu jednotlivostí pracnější, ale z pohledu celého projektu méně pracné, než je tomu naopak u RAD.

Základní trojúhelník tvorby SW

V každé SW firmě musí nejvyšší managment vyřešit základní otázku vztahu tří pojmů, které tvoří tři pilíře tvorby SW ve firmě. Tento vztah vyjadřuje následující obrázek:



obrázek 25 : Trojúhelník tří pilířů tvorby SW ve firmě

Jinak řečeno, každý management SW firmy by měl vyřešit tyto tři základní otázky:

1. Jak budou ve firmě zavedeny a aplikovány metodiky tvorby SW, jinak řečeno závazné pracovní postupy ve firmě (pojem „metodiky“ jsem zvolil pro volný překlad z anglicky „processes“). To je pilíř „Metodiky“ na obrázku.
2. Jaká bude používána notace (syntaxe) pro tvorbu SW. V programovacím jazyce je odpověď na tuto otázku jasná, protože můžete použít pouze notaci daného jazyka. Pro modelování doporučuji použít notaci UML.
3. Jaké nástroje bude firma pro tvorbu SW potřebovat (z angl. „tools“), například Visual Studio, jaké CASE nástroje apod.

Zásady pro přechod firmy od strukturálního programování k OOP a UML

V této kapitole jsou popsány zásady přechodu firem na OOP a UML.

Změny pohledů na kvalitu SW na přelomu století

V posledních dvou desetiletích se v souvislosti s informačními technologiemi hovoří o tzv. krizi softwaru. I když se na první pohled nemusí zdát současná situace v SW „kritickou“, zkusme se zamyslet nad jednoduchou otázkou: Co kdyby produkty architektury a stavitelství jako produkty dvou oborů lidské činnosti vykazovaly stejné negativní vlastnosti, jako mají softwarové produkty? Tato představa je hrozivá:

- Výstavba budov by se prováděla „ad hoc“. Přímo ve stavbě a takříkajíc na místě by se dodělávaly požadavky vzniklé jakýmsi pochybným nesystematickým postupem. Budovy by vznikaly bez plánu a pouze ústním zadáním vedoucího stavby na základě rozhovorů s uživatelem.
- Budovy by se vylepšovaly „lepením“ dalších funkcionalit bez nějakých rozumných plánů, tedy dodělávky by se prováděly podle schématu: „jak si zedník smyslí, tak to postaví“.
- Některá budova by „sem tam někdy spadla“ anebo by vykazovala fatální chyby. Přitom nejsmutnější na této situaci je to, že takovýto „error“ příliš častého pádu budov resp. fatálních nedostatků by se vůbec nepovažoval za nějakou tragédii. Budova by se prostě opravila „servisním zásahem“ a jede se dál. Dokonce by se tento postup považoval za běžný a stal by se součástí smlouvy mezi zákazníkem a dodavatelem.

Krize SW se projevuje i tím, že jsme si příliš zvykli na to, že software vykazuje takovéto velmi negativní vlastnosti a není na tom tedy nic podivného. Avšak v porovnání s jinými obory je na tom SW velmi špatně.

Na druhé straně i v oboru SW se začíná situace měnit a tato změna se postupně projevuje i v České republice. Na začátku 90. let po sametové revoluci nastal velký „boom“ výroby softwaru, přičemž nároky uživatelů na kvalitu byly velmi nízké. Uvedený postup byl ve své době možný a vzhledem k vysoké poptávce na rychlost dodávky se dá považovat také za v té době jediný možný postup. Avšak všimněme si, jak dnes vzrůstají požadavky na kvalitu softwaru ze strany uživatelů. Stává se již problémem uplatnit na trhu nekvalitní software, protože existují technologie, které se takovýmto rychlým a vlastně laickým postupům výroby SW brání. Trh SW se postupně mění a žádá vyšší kvalitu.

Musím na tomto místě zdůraznit, že vedení firem, která ignorují tento nástup kvality do výroby SW a tedy nástup nabídky takovýchto kvalitních a stabilních produktů, které vyžadují minimum pozdějších servisních zásahů, si vytvářejí do budoucna vážný problém pro svou firmu nikoliv pouze technologický, ale zejména nechávají vzniknout problémům obchodním. Požadavky zákazníků se totiž začínají také v ČR rapidně měnit a to směrem nahoru k požadavku na vyšší kvalitu SW.

Kategorie SW z hlediska požadavků na kvalitu

Ne každý softwarový výtvor musí splňovat zvyšující se požadavky na kvalitu. Podle tohoto hlediska můžeme software rozdělit do následujících kategorií:

- entusiastické programy. Takovéto programy vznikají na základě místních požadavků nějakých nadšenců, například studentů, astronomů, apod. Většinou se jedná o software pro vlastní potřebu. Protože pole působnosti takového softwaru je malé a software není rozsáhlý, a protože dokonce většinou je sám tvůrce také uživatelem, nejsou požadavky na kvalitu nikterak vysoké.
- komerční konzumní software typu balíček. Sem patří různé desktopové aplikace jako editory, spreadsheety, „office“ programy – tj. kancelářské programy, hry, antivirové programy, utility apod. Zde se již nároky na kvalitu zvyšují z důvodu distribuce softwaru mezi zákazníky. Je pochopitelné, že požadavek na opravu vytvořeného softwaru již distribuovaného mezi zákazníky může znamenat enormní zvýšení nákladů (distribuce service packů apod.). Firma vyrábějící software bez chyb získává obrovskou konkurenční výhodu oproti firmám chybujícím.
- business systémy, podnikové systémy. Sem patří systémy instalované v podnicích, organizacích apod. většinou na zakázku a na klíč podle smlouvy mezi dodavatelem a odběratelem (ekonomické systémy podniků, banky, pošty, podniky, dopravní systémy, spedice apod.). Nároky na kvalitu jsou

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

ještě vyšší než u softwaru v předešlém odstavci. Vztah mezi zákazníkem a výrobcem je užší, důsledky havárií, nestabilit a změn mohou mít z hlediska nákladů velmi nepříznivé důsledky, protože dodavatel se může stát na dodávaném SW závislým z hlediska ekonomických výsledků. Pád systému resp. závady v něm mají pro uživatele ekonomicky katastrofické následky včetně poškození dobrého jména odběratele (například v bance apod.).

- software kritický pro zdraví a životy lidí, kritický pro normální chod společnosti. Patří sem software pro nemocnice, pro řízení letecké dopravy, software pro kosmonautiku, software v jaderných elektrárnách, ministerstvo obrany, velitelská stanoviště, raketové systémy, ale také burza apod. Selhání takovéhoto systému samozřejmě nezpůsobí pouze zvýšení nákladů, ale může mít za následek lidskou nebo společenskou katastrofu.

V čem spočívá krize SW a jak se jí vyhnout

Sami můžete posoudit, do které kategorie patří software vyráběný vaší firmou. Ve většině případů, kdy jsem působil ve firmách jako externí konzultant a externí analytik, se jednalo o firmy vyrábějící buď konzumní software typu balíček anebo to byli dodavatelé business systémů pro organizace a podniky.

Zajímavé na mých zkušenostech je to, že každá firma se ve svém vývoji setkala s tímto problémem krize SW na „vlastní kůži“. Vývoj těchto firem byl velmi podobný: Na začátku vznikl žádaný software pomocí rychlého vývoje bez nějakých úvah o jeho kvalitě nebo jeho nekvalitě. Firma se díky tomuto softwaru velmi dobře etablovala na trhu. Postupným zvyšováním požadavků na systém však začaly přibývat problémy. Nakonec se firma dostala do situace popsané zde jako krize SW s těžko udržitelným softwarem.

Paradoxně hlavním problémem krizového stavu softwaru je jeho nadměrná (tj. zbytečná) složitost. Jednoduché postupy jeho tvorby (například RAD) neznamenají, že se ve svém výsledku jedná o jednoduše pojatý a průhledný software. Problém nynějších softwarových inženýrů, kteří používají rychlé postupy tvorby SW, je v tom, že neumějí dělat věci jednoduchými, dokonce se věci stávají mnohem více složitějšími, než jak si to tyto věci vyžadují. V tomto paradoxu se ukazuje, že platí základní motto této knihy, které neplatí jenom při tvorbě SW. Technologie, která sama o sobě vyžaduje postup „tvořit SW jako jednoduchý SW“, je ukryta v objektově orientovaném přístupu. Jak ukazuje praxe, prvním krokem, jak se vyhnout nebo odstranit krizi, je přechod firmy na technologii OOP s použitím komponentní technologie a samozřejmě následně přechod firmy na objektové modelování (nejlépe pomocí UML).

Ovšem v tomto kroku přechodu je ukryto velké nebezpečí a úskalí. Každá změna je riziko a dokonce platí, že každá změna, i změna k lepšímu, je vždy nejprve změnou k horšímu. Přechod na OOP, komponentní technologii a UML ve firmě není totiž pouze otázkou zvládnutí této technologie pracovníky firmy. To je požadavek nutný, nikoliv postačující. Přechod na OOP, COM a UML je projektem jako každý jiný projekt a musí být projektem řízeným shora vedením firmy.

Otázkou je, jaké kroky musí vedení firmy minimálně učinit?

Analýza, design a kódování

Doporučuji, aby v projektech tvorby SW byly zavedeny minimálně tři fáze vývoje. Osobně ze zkušeností v konzultacích doporučuji jako nezbytné oddělit a tím dodržet určitou posloupnost od sebe oddělených prací minimálně ve třech fázích.

- Analýza
- Design (též Návrh)
- Kódování

Podotkněme, že existují další fáze: Testování, nasazení apod., ale těmi se nebudeme zabývat. Tyto tři fáze jsou fáze pouze jako extrakt náplně „čistého vývojového oddělení“.

Musíme zde zdůraznit, že toto rozdělení se ukazuje jako nezbytné z praktického hlediska a firma, která toto rozdělení neprovede, naráží na neskonalé problémy. Velmi důležitým pravidlem pro toto rozdělení je požadavek, aby každá z těchto fází byla reprezentována existujícími dokumenty, které se musí bezpodmínečně vyhotovit a odevzdat. Platí pravidlo, že dokument „ležící“ pouze v něčí hlavě neexistuje. Na tyto tři fáze je nutno se podívat ze třech úhlů pohledu:

- Vývojové dokumenty nutně projdou těmito fázemi a každá z fází dává dalšímu dokumentu jiný a nový obsah. Celý software ve všech třech pohledech lze rozdělit do tří základních pohledů, na analytické dokumenty, na dokumenty návrhu a dokumenty kódu (tj. zdrojový kód). Výsledek, tj. software, se tak jeví ve třech různých rovinách pohledu: V rovině analytické, v rovině designu a jako samotný kód. Každý z těchto pohledů má jiné hledisko. Rozdělení tak vysoce zprůhlední výsledný software, protože jej vidíme v různých abstrakcích a to od nejvyšší analytické až po nejhlubší „implicitní“ v kódu. Výhodou použití UML je to, že velmi jednoduše můžeme zavést odpovídající formalismy pro dokumenty. Stačí pouze určit, které modely UML budou vytvořeny v analýze a které v designu. Samo UML nabízí notaci pro tvorbu analytických a design dokumentů.
- Protože v každém projektu musí dokumenty softwaru projít těmito fázemi, požaduje se automaticky, aby ve firmě existoval závazný postup, který zaručuje dodržení těchto fází. Rozdělení na fáze má tedy význam z hlediska metodik ve firmě. Vlastně je to onen první krok k opakovanému „re-use“ ve způsobu řízení projektů. Jinak řečeno, je to první krok k určení obecných postupů „co se v kterém okamžiku má udělat“. Protože UML nabízí závaznou notaci pro tyto dokumenty, část práce na metodikách, tj. jak tvořit tyto dokumenty, můžeme pouze z UML převzít. Většinou se tak děje nasazením některého z CASE nástrojů, jehož použití si vynutí určité postupy.
- Vývojové dokumenty procházející těmito fázemi musí být tvořeny někým, kdo vystupuje v roli tvůrce tohoto dokumentu. Znamená to, že v projektu musí být vymezeny určité role pracovníků. Z hlediska minimálního rozdělení na tří fáze musí tedy existovat analytik, designér a programátor. Současně musí být určen způsob předávání dokumentů mezi pracovníky v těchto rolích.

Rozdělení na fáze tedy zavádí automaticky tři základní výhody:

- přehlednost systému s pohledem na vrstvu analytickou, na vrstvu designu a na samotný kód
- re-use mezi projekty se stejnou analýzou, ale různou technologií (výměna DB apod.)
- zavedení prvních závazných postupů ve firmě
- zavedení rolí v projektu se svými povinnostmi

Náplň analýzy

Smyslem a posláním analýzy je vyhotovit dokumenty pomocí UML a příbuzných dokumentů UML, které:

- vysvětlí podstatu a esenci problému
- vytvoří modely informačního systému na úrovni problémové domény
- zjednoduší složitější pohledy
- zobecní pohledy na části systému, tj. najde jejich zobecnění

Platí důležité pravidlo, že ve fázi analýzy se neptáme a nepostihujeme „jak“ bude daný systém fungovat konkrétně v daném a předem daném prostředí. Popisujeme systém pouze na úrovni problémové domény. Existuje pomůcka pro dodržení tohoto pravidla: V analýze se nesmí objevit pojmy implementační z daného prostředí (tabulka, sloupec, apod.), ale objevují se v ní pouze pojmy z problémové domény (faktura, řádek faktury). Toto pravidlo má velmi blízko k objektovému pohledu na vytvářený software, jak ukazuje následující příklad.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Příklad chybné věty z analýzy:

...Z tabulky TABKLIENT se načtou údaje ID, IČO, Název a DIČ a tyto se zobrazí obsluze pomocí Listview, obsluha vybere řádek v ListView s daným IČO, Názvem a DIČEM a odchytí se tak ID...

Správná formulace pouze v pojmech problémové domény:

...Obsluze se zobrazí Seznam Klientů (IČO, Název a DIČ) a obsluha vybere Klienta ...

Prací analytika je také hledat zobecnění a „analytickou optimalizaci“. Například se má řešit matematický problém bodu, přímky, kružnice, elipsy, hyperboly a paraboly. Analytik by měl odhalit, že všech 6 případů lze řešit pomocí jedné rovnice druhého řádu. Podobně by měl najít zobecnění mezi entitami apod.

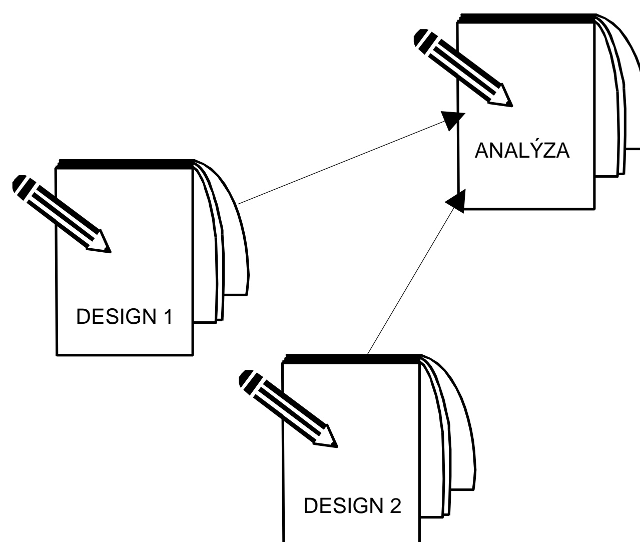
Výsledkem analýzy jsou dokumenty ve tvaru modelů UML (bude pojednáno konkrétně které), a tyto modely obsahují pouze pojmy z dané problémové domény. Dokumenty jsou nejenom výchozími pro další zpracování, ale jsou čitelné jak pro vývojáře, tak pro uživatele, který nemusí znát vůbec nic z teorie programování.

Analytické dokumenty jsou nezávislé na implementačním prostředí a jsou platné pro každé prostředí, protože popisují pouze podstatu problému (zodpovídají otázku pouze „o co jde“ a nikoliv „jak to jde“).

Náplň designu

Práce v designu navazuje na práci v analýze. Analytický dokument se stává výchozím a je chápán zadání pro dokument designu. Designér již řeší otázku, jak bude analýza realizována konkrétně v daném prostředí. Designér převezme analytický dokument a na základě něho vytvoří dokument designu, který na analýzu navazuje a popíše tak „realizaci analýzy“.

Výsledkem je celkový dokument, který „v sobě“ obsahuje analýzu a přitom popisuje realizaci systému v jednom ze zvolených konkrétních prostředí. V jiném prostředí se vezme tatáž analýza a provede se její „implementace“ do jiného prostředí. Analytická podstata problému je však stejná.



obrázek 26 : Stejná analýza, dva různé designy,kde šipky označují závislost dokumentů mezi sebou.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Rozdělení na analytické a design dokumenty odpovídá již zmíněnému postupu re-use sdílení. V tomto případě se operace re-use týká dokumentů analýzy. Nebudeme přece vytvářet několikrát analýzu při řešení stejného problému pro různá prostředí.

Důležité je (a to uvidíme při modelování v UML), že fáze designu navazuje v OOP na fázi analýzy bezprostředně „převzetím" analytického dokumentu a jeho rozšířením o design. Práce na designu tedy není převzetím pouze nějakých „abstraktních myšlenek" z analýzy jako podkladů, ale jedná se o přímé použití prvků modelu v analýze pro design.

Příklad: V analýze se zavede třída jako class CKlient. Tato třída bude existovat „stejně" i v designu, designér pouze rozšíří její funkcionalitu resp. provede dědění apod., aby se tato třída mohla dobře chovat i v daném prostředí (například ukládat se do relační databáze stojící v pozadí apod.).

Vztah mezi modely designu a analýzy je v čistém OOP vztahem závislosti, kde design „obaluje" analýzu funkcionalitou daného prostředí.

Přechod firmy na OOP z hlediska času a nákladů

Přesto (nebo právě proto), že jsem velkým zastáncem OOP přístupu, musím upozornit na velké nebezpečí vyplývající z přechodu na OOP. Pokud firma přechází na jakoukoliv jinou technologii, musí to být krok vážený a naplánovaný. Jakékoliv „hurá" projekty se firmě určitě vymstí.

Jeden z nepříjemných faktorů takového projektu je bezesporu čas, který je třeba obětovat a kterého není nikdy při tvorbě softwaru dostatek. Určitá oběť musí být z hlediska času dána z podstaty věci, protože nelze přejít na nový způsob programování ze dne na den.

V literatuře se uvádí následující statistický odhad: Při přechodu pracovníka ze strukturálního na OOP přístup je třeba počítat přibližně s 6 - 9 měsíci jeho přechodu na objektové myšlení. Z vlastní zkušenosti mohu potvrdit, že se jedná o vcelku správný statistický údaj, avšak samozřejmě se jedná o údaj pouze statistický s nutnými odchylkami. Znám případy, kdy přechod na OOP proběhl u pracovníka řádově v týdnech a znám případy, kdy neproběhl vůbec.

Poznámka: Z konzultací jsem vypozoroval jednu závažnou překážku, která může pracovníkovi úplně zabránit v přechodu na OOP a tou je tzv. ***programátorská ješitnost****. Danému pracovníkovi vybavenému touto negativní vlastností je při vysvětlování „vždy všechno jasné" a ze zásady nikdy „nepřizná", že něčemu nerozumí. Neuvědomuje si, že pokud něčemu nerozumí, tak chyba není povinně v něm. Pokud máme v týmu vypěstovanou takovouto atmosféru arogantní ješitnosti (většinou tvořenou několika málo jedinci), velmi těžko budeme přijímat nové poznatky včetně OOP. Dotyčnému lze pouze poradit, aby se zbavil této vlastnosti, protože velmi silně brání hlavně jeho rozvoji. Spousta informací jdoucí kolem něj mu prostě uteče.*

Pro přechod pracovníka na OOP myšlení je také velmi důležité, zda v daném týmu pracuje alespoň jeden pracovník, který již objektově pracoval. Velmi rychlý přechod pracovníka na OOP je v tom případě, pokud celý tým pracuje objektově a on je zařazen jako nový člen tohoto týmu. Pokud daný pracovník přejde na OOP, musíme počítat také s jeho seznámením se použitím objektů ve vývojovém prostředí jazyka. K tomu je zapotřebí přibližně 1 měsíc.

Není bez zajímavosti udávaná informace o porovnání zvýšených nákladů na první projekt v OOP. Je pochopitelné, že přechod na OOP způsobí zvýšení nákladů díky zavádění nových technologií do té doby ve firmě nepoužívaných. Podle údajů v literatuře je statistický odhad nárůstu nákladů na první projekt v OOP o 30% vyšší, než u téhož projektu pojatém v původní technologii. Musím poznamenat, že podle mých poznatků a odhadů se toto číslo jeví jako příliš optimistické a spíše bych jej zvedl na hranici 50%. V každém případě (ať už bereme optimistický nebo spíše pesimistický odhad) je třeba počítat s určitými ekonomickými ztrátami u prvního projektu jako daň za zavedení nové technologie.

V dalších následných projektech se však náklady rapidně snižují. Uvádí se údaj snížení nákladů od druhého a dalšího projektu přibližně o 30%. Jedná se o jednoduchý důsledek ekonomického využití re-use v OOP. Pokud provádíme re-use v plné míře, další projekty se stále více stávají pouhými „skládačkami“, ale postupně vždy jeden projekt za druhým více a více využívá této vlastnosti. Postupné snižování nákladů tedy souvisí s budováním objektové a komponentní knihovny ve firmě, v použití externích komponent apod. Snížení nákladů v „již zajeté“ firmě pracující objektově a komponentně je opravdu velmi vysoké a je určitě oproti strukturálnímu a to určitě pod hranici 50% ! V těchto úsporách navíc není započítána silná konkurenční výhoda v menší poruchovosti softwaru.

Největších úspor v nákladech při použití objektové a komponentní technologie se dociluje těmito základními faktory:

- možnost použití velmi silného re-use. Nutno podotknout, že OOP a použití UML je na re-use silně zaměřeno (v čistém OOP a UML se jedná o maximální re-use, jaký je v programování teoreticky možný)
- zvýšení přehlednosti a transparence systému a tedy vyvarování se fatálních chyb vzniklých „vývojářským chaosem“ v systému
- zvýšení kvality softwaru a následné prudké snížení nákladů po dokončení SW díky malé chybovosti, vysoké flexibilitě atd. (uvádí se, že většina nákladů ve strukturálním programování je soustředěna ex post až po implementaci softwaru u zákazníka)

Doporučení firmě v projektech při přechodu na OOP

Pokud firma přechází na OOP, lze jí poradit určitá doporučení:

- je třeba používat externích zkušeností jako školení, literatura atd.
- je třeba nastartovat pilotní projekt v OOP
- je třeba používat externistů při zahajovaní prací
- je vhodné, aby v týmu pracovali zaměstnanci již seznámení s OOP
- je třeba vyškolit všechny účastníky projektu, včetně vývojářů, testerů, vedoucích projektů atd.

K těmto obecným doporučením přiložím ještě určité zkušenosti z praxe. Velmi důležitý je pilotní projekt. Není vhodné, aby tento pilotní projekt založený v OOP byl pouze jakousi hračkou, „nezávazným projektem“ apod. Tlak na jeho dokončení včetně použití OOP a UML je opravdu žádoucí a nutný.

Poznámka: V jedné firmě se rozhodli, že založí souběžně dva projekty řešící tentýž problém, tedy dvě varianty, jak objektovou a strukturální vedle sebe. Samozřejmě přechod na OOP bolí, hlavně když vidíte vedle sebe jakousi možnou náhražku ve strukturálním programování. V daném projektu došlo k zajímavému efektu: Protože objektová analýza může vyjádřit problematiku pomocí UML velmi přesně a důsledně, byli účastníci projektu velmi spokojeni s analýzou učiněnou pomocí OOP a UML. Protože se však vyskytly problémy přechodu na OOP (díky určitým chybám v koncepcích), začalo se od OOP ustupovat a rozhodlo se pokračovat sice s analýzou v OOP, ale sám program byl napsán jako datová aplikace (RAD). Avšak analýza v OOP byla velmi důsledná a podrobná, tj. problematika v ní velmi podrobně rozepsaná do velkého počtu entit a tak při její realizaci pomocí RAD nastaly problémy, popsané v předešlé kapitole (složité vazby v SQL, jejich provázání atd.). Zjistilo se, že takto napsaná aplikace je velmi komplikovaná a proto se „začala optimalizovat“, jinak řečeno zjednodušovat. Ale tato optimalizace nebyla pouhou změnou designu, ale zjednodušila se analýza, a celý problém se převedl na dohody s uživatelem „zda mu to takto stačí“. Pilotní projekt v OOP nakonec nebyl realizován v původním rozsahu.

Velmi žádoucími a nezastupitelnými jsou samozřejmě konzultace externistů, resp. použití externích zdrojů, školení, knihy apod. Musím však upozornit na jedno velké svým způsobem paradoxní úskalí,

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

které jsem také vypozoroval ve firmách přecházející na OOP a UML. Toto úskalí spočívá v tom, že pracovník, který si přečte základy OOP, UML a COM a pochopí souvislosti a hlavně výhody tohoto přístupu, navíc pokud si udělá nějaké příklady a „ono to chodí“, tak tento pracovník propadne nesprávnému dojmu, že „na tom přechodu na OOP a UML vlastně nic není“.

Problém tkví v tom, že něco jiného je „přejít na OOP a UML jako chápající jedinec“ a něco jiného je takovýto přechod pro firmu. To, co dotyčný vidí, je pouze cíl „takto bychom to měli dělat jako firma“. Ale mnohem důležitější je samotná cesta celé firmy k tomuto cíli a ta musí být velmi dobře připravená. Takže nestačí jenom „pochopit OOP a UML“, ale je třeba zavést celý **projekt s plánem, harmonogramem a konkrétními kroky** přechodu na OOP, UML, COM pro celou firmu. Každému, kdo někdy řídil lidi pod sebou, je tento rozdíl zřejmý.

Důsledkem takového nepochopení ze strany pracovníků je sice správný tlak „zespodu vůči vedení“, ale také nebezpečí určité netrpělivosti od těchto pracovníků a vyžadování „hurá“ projektů přechodu na OOP (před kterými důrazně varuji).

Stupně řízení projektů ve firmě a kvalita řízení projektů

Spolu s přechodem, na OOP je spojena otázka, jak bude firma jako celek řídit svoje projekty v objektovém prostředí, tj. jak budou fungovat metodiky ve firmě. Nejprve je třeba rozlišit, v jakém stupni je jejich úroveň použití, čemuž odpovídá úroveň a kvalita projektového řízení.

Rozlišují se tyto stupně metodologií ve firmách (psáno anglicky s překladem):

1. ***Initial*** - inicializační stav firmy. Ve firmě neexistují žádné metodiky. Neexistují ani napsané metodiky, a to ani pouze metodiky vyslovené. Každý projekt se řídí „případ od případu“
2. ***Reproducible*** - opakovatelné postupy při řízení v projektech. V tomto případě již lze alespoň přibližně stanovit plán projektu. Metodiky jako „návodky“ však nejsou nijak stanoveny a spíše jedná o určité zvyky a návyky ve firmě zavedené.
3. ***Defined*** - existují definované pracovní postupy, tj. existují dokumenty přístupné zaměstnancům jako metodiky, které se dodržují. Ve firmě musí být zavedeny mechanismy, které použití metodik udržují při životě. Z vlastní zkušenosti mohu potvrdit, že není až tak složité metodiky zavést, jako je udržet v platnosti a v používání. Pro jejich udržení při životě se vyžaduje mravenčí práce s malými kroky a nikoliv zavedení metody „čínských skoků“.
4. ***Framed*** - použití metodik se začíná měřit. Znamená to, že pracovní postupy ve firmě jsou měřitelné a vyhodnocují se podle metrik softwaru.
5. ***Optimized*** - poslední nejvyšší stadium. Pracovní postupy se nejenom vyhodnocují (měří), ale hledají se lepší metodiky, tj. stav firmy se z hlediska řízení optimalizuje

Většina firem, kde jsem prováděl konzultace, se nacházela ve stavech buď mezi 1 až 2 nebo 2 až 3. Zatím jsem se v ČR nesetkal s firmou, která by dosáhla stupně 5.

Vztah řízení projektů k UML a doporučení daná od tvůrců UML

Samo použití OOP spolu s UML samozřejmě přináší tak trochu jiný styl do práce ve firmě. I když, jak zdůrazňují sami tvůrci UML, tento modelovací jazyk není návodem k metodikám, existují určitá doporučení daná tvůrci UML. Jedná se o jakýsi „doprovodný návod“ k použití UML v projektech pro vedoucího projektu.

Vývojový proces v projektu by měl být podle tvůrců UML:

- řízen Use Case modelem
- zaměřen na architekturu
- měl by být iterativní a inkrementální

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Rozeberme tato doporučení. Nejprve je třeba zdůraznit, že tyto rady tvůrců UML se netýkají problému „jak modelovat v UML“, tj. samotných vývojářů, ale tato doporučení jsou směrována k vedoucím projektů ve stylu „jak řídit projekt v objektovém a komponentním prostředí při použití UML“.

Projekt řízený Use Case modelem

První rada se týká použití tzv. Use Case modelu v řízení projektu. Samotnému Use Case modelu bude věnována celá kapitola (jako každému z modelů UML). Tam si také toto doporučení probereme velmi podrobně. Zde si pouze všimněme, že Use Case model se nepoužívá pouze pro čisté „vývojové“ práce (jako součást celkového modelu softwaru), ale používá se také vedoucím projektu pro řízení projektu a stává se tak důležitým dokumentem používaným pro metodické řízení v projektu.

Navrhovat a dodržovat správnou architekturu

Druhé doporučení ve znění „*vedoucí, zaměřte se na architekturu*“ neznamená zaměřit se na nějaký návrh tabulek apod. Pod architekturou mají autoři UML na mysli rozvrstvení systému a jeho rozložení do rozdělitelných celků. Jedná se vlastně o přímý důsledek správného použití objektového a komponentního přístupu. Tvůrci UML doporučují, aby se také řízení projektu zaměřilo na toto rozvrstvení systému, tj. na objekty, na komponenty a na skupiny objektů a skupiny komponent, tzv. vrstvy. Jednoduše řečeno, autoři UML doporučují vedoucím projektům jednoduchou a srozumitelnou zásadu: Pro vedoucího projektu je mnohem snazší a efektivnější řídit práce nad „několika oddělitelnými částmi systému“ než nad „jedním velkým, zašmodrchaným a provázaným molochem, kde všechno souvisí se vším“. Zásadou objektového modelování je pochopitelně rozložení na vrstvy, tj. na objekty a komponenty a na vrstvy objektů a vrstvy komponent (což budeme probírat v kapitole Komponentní model a Třívrstvová architektura). Zde se zdůrazňuje, že toto rozložení má velký kladný vliv na řízení projektu, samozřejmě za podmínky, že se tento pohled z hlediska řízení nezanedbává.

Poznámka: Velmi často jsem se setkal se systémy, které byly takříkajíc „nedělitelné“. Jejich provázanost jednotlivých částí byla tak silná, že nemělo smysl hovořit o jednotlivých částech systému, i když jakési pomyslné části měly své pojmenování. Například bylo nutné dodávat „celou databázi“, i když zákazník požadoval pouze částečnou funkcionalitu. V žádném případě si nelze představit pod správným a čistým rozvrstvením, že se vezme již existující moloch, a ten se za pomoci tužky pomyslně rozdělí na menší části. Problematika rozložení na části není takto jednoduchá a má svá pravidla. Takovéto jednoduché rozložení systému na menší části je spíše „zbožným přáním“ a pouze pomyslným dělením, než skutečným rozložením na vrstvy. Je to stále jeden moloch „počmáraný“ pomyslnými hranicemi.

V případě dobře navrženého systému s průhlednými vrstvami (v dalších částech uvedeme ukázky) se systém vyznačuje těmito vlastnostmi:

- systém, i když je rozsáhlý, je jednoduchý, protože to, co je složité a rozdělitelné na menší celky, se stává jednoduchým
- systém je srozumitelný, protože co je jednoduché, je srozumitelné
- systém je elegantní, protože co je nepřehledné, je škaredé a neelegantní
- systém má dobře definované hladiny abstrakce, uvedený pohled ve vrstvách abstrakce odpovídá systematickému pohledu přirozeného lidského chápání
- systém má jasně definované interfacy, tj. jsou zřetelně vidět rozdělení a následně definované propojení částí u každé abstraktní úrovně. Pohled na jednoduché interfacy nás zbavují v první abstrakci zbytečných detailů „za těmito interfacy“

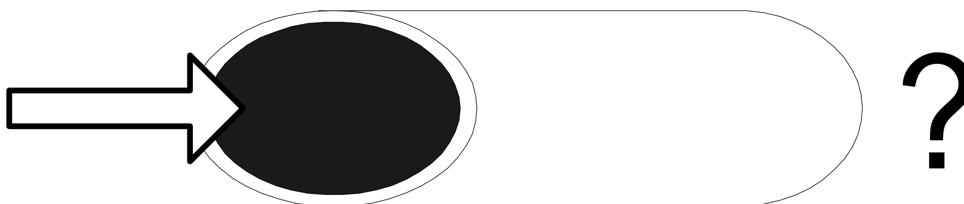
Pokud systém nemá dobrou a logickou architekturu a stává se nedělitelným molochem, potom se projekt velmi špatně řídí. V tom případě v systému opravdu „všechno souvisí se vším“. Pokud chcete

odladit nějaký modul, potom se dostáváte do začarovaného kruhu jiných modulů. Žádný z modulů není nikdy ukončen, protože jeho funkcionalita je závislá na funkcionalitě nesmyslně propojených modulů.

Metody vývoje SW

Třetí doporučení se týká způsobu vývoje z hlediska řízení prací v projektu. Existuje několik možných způsobů řízení projektu, některé lepší a některé horší. Uveďme si nejdůležitější z nich.

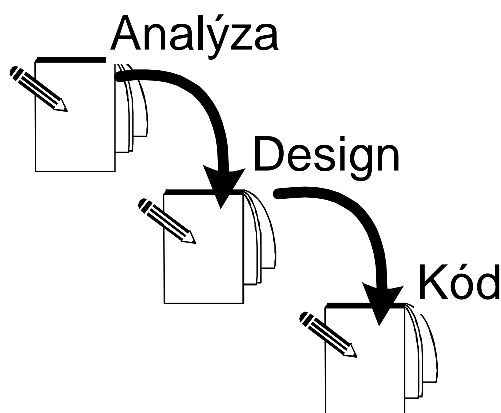
První z nich se nazývá příznačně jako „Metoda tunel“



obrázek 27 : Metoda řízení projektu tunel

Její princip je velmi primitivní a bohužel také dost rozšířený: Na počátku projektu se vstupuje do „tunelu“, tj. do neznáma, a vedení vydává pokyny podle momentální situace. Vytváří se tak dost silný tlak na realizaci projektu. Úkolem je „nalézt“ konec tunelu a projekt zdárně ukončit. Projekt je řízen pouze operativně a nikoliv s nějakou koncepcí.

Další metodou je „Metoda vodopád“, u které už můžeme vypozorovat určitou koncepci: V projektu je stanoveno, že musí vzniknout analytické dokumenty, poté design dokumenty a poté dokumenty kódování (zdrojový kód).



obrázek 28 Metoda řízení projektu typu vodopád

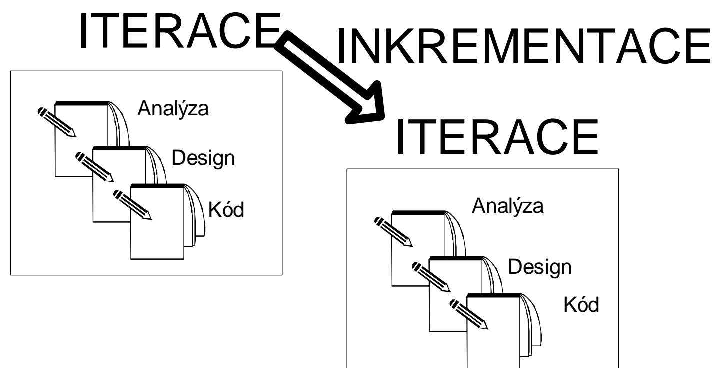
Tento postup zní vcelku logicky, má však jeden háček: Nejprve musí vzniknout „celá“ analýza, potom „celý“ design a poté „celé“ kódování. Protože se dokumenty předávají z jedné fáze do druhé „odshora dolů“, nazýváme tuto metodu jako „Metoda vodopád“. Je vcelku pochopitelné, že tato metoda je mnohem lepší, než předešlá „chaotická“ metoda tunel (která je oproti tomu větším dobrodružstvím),

ale má své nevýhody. Největší nevýhodou je její těžkopádnost a obtížné řízení lidských zdrojů. Nejprve všichni „dělají analýzu“, potom „všichni dělají design“ a potom „všichni kódují“.

S největším nepochopením jsem se v konzultacích setkal u použití třetí, tzv. iterativní a inkrementální metody, kterou budeme označovat jako I+I metoda. Její výjimečnost spočívá v tom, že je v plném rozsahu použitelná pouze v objektovém prostředí, je to tedy určitá výsada OOP. Většinou pracovníci, kteří nemají zkušenosti s OOP, nepochopí její princip do důsledku a ten, kdo používá OOP, většinou automaticky začne tuto metodu používat (aniž by o ní slyšel), protože její použití je v OOP velmi přirozené. Tvůrci UML v tomto třetím doporučení vyslovují radu, aby vedoucí projektu této metodě věnovali mimořádnou pozornost a zakomponovali ji obecně do řízení projektu.

Podstatou této metody je provedení analýzy, designu a kódu v jedné relativně malé iteraci v jedné relativně malé části systému. Provede se něco jako „probublání“ určité vymezené části systému od analytického dokumentu až do konečného kódu, aniž by se čekalo na výsledky „celé analýzy“, „celého designu“ a „celého kódu“ od jiných částí systému. Tento jeden krok tvorby od analýzy přes design po kód je jednou iterací.

Poté se provede „rozšíření prací“ o další část systému (tj. inkrementace) a provede se druhá iterace. Takto se postupuje dále až do cíle. Dá se říci, že metoda I+I je vlastně složení velmi mnoha menších vodopádů částí systémů:



obrázek 29 Metoda iterace a inkrementace

V čem většinou spočívá nepochopení metody I+I? Většinou se argumentuje tím, že tento postup je přece použitelný i ve strukturálním programování. Je to sice možné, ale je to oproti OOP mnohem obtížnější. Základní nepochopení této metody spočívá v tom, že rozšiřování systému v inkrementaci nespočívá pouze v přidávání entit (tj. objektů, přesněji tříd) a vazeb, ale v přidávání funkcionalit objektů spolu s novými stále konzistentními vnitřními stavy objektů.

Pokud objekt získá určité hodnoty svých atributů, které odpovídají nějakému stavu, potom je z hlediska další funkcionality jedno, jak se do tohoto stavu dostal. Můžeme tedy zahájit jednu iteraci od tohoto stavu a přitom objekt se mohl do tohoto stavu dostat nějak jinak (třeba simulací). Představme si, že za tím účelem vytvoříme dočasnou simulační metodu, která objekt nastaví do určitého stavu (například metoda simulace naplnění z databáze apod.).

Od nastavení tohoto stavu lze provést jednu iteraci až po požadovaný další stav. Připomenu, že ve strukturálním programování postaveném na datovém modelu, se začíná vývoj od datových entit a tabulek (od struktury dat resp. ERD). V objektovém programování se začíná od objektu v určitém stavu a poté se může provést další vývoj funkcionality objektu. Na rozdíl od strukturálního programování můžeme zahájit proces I+I v libovolné iteraci. V metodě I+I se doporučuje v iteracích začít v kritických místech systému.

Příklad: Uvedu klasický příklad z praxe. V projektu firmy se měl vyvinout systém, který měl optimalizovat jízdy popelářských vozů městem. Byly zadány ulice, body svozu odpadu, garáže, skládky atd. Existuje nějaký optimalizační algoritmus, který podle počtu vozů a rozmístění ulic apod. určí optimální trasy vozů městem (a tím se firmě pro svoz odpadu sníží náklady). Chceme vyvinout a odzkoušet v dané iteraci pouze metodu pro optimalizaci. Je třeba zdůraznit, že tím bychom měli začít pro ověření algoritmu, vždyť jaký by to byl „průšvih“, kdybychom naučili systém spoustě věcí, například načítání všech entit z databáze, jejich ukládání atd. a potom se ukázal algoritmus optimalizace (pro který to vše činíme) jako mylný a skončili bychom s celým již naprogramovaným systémem. V metodě **I+I** můžeme nejprve vytvořit tu část systému v objektech, které vytvoří stav těsně před optimalizací. To odpovídá tvorbě objektů křižovatek, tvorbě objektů úseků mezi křižovatkami, ulicím, garážím, atd. Vazby a hodnoty atributů musíme naplnit nějakým jednoduchým simulačním programem (nejlépe „doslova ručně kódem“). Po tomto naplnění se objekty nacházejí ve stejném stavu (drží stejné vazby a atributy), jako by byly naplněny z databáze a to je startovací bod pro náš další vývoj. Provedeme vývoj této iterace, tj. vyvineme optimalizaci, odzkoušíme ji a poté simulační kód odložíme. Máme tak objekty naučeny optimalizaci. V tomto příkladu třeba jiný pracovník v jiné iteraci vyřeší naplnění objektů z databáze. Pro něj je konečným bodem výsledku jeho iterace naším startovacím bodem. Výsledky prací nás a kolegy mohou na sebe navázat. Tvůrci UML doporučují jednoduchou věc: vedoucí projektu by měli řídit práci mezi pracovníky tímto způsobem.

Důležitá výjimka z metody I+I

Existuje však jedna výjimka z pravidla metody **I+I**, a tou je tvorba Use Case modelu, který by měl vyhotoven v celé své celistvosti najednou a pokud možno hned na začátku projektu. O této výjimce budeme podrobně pojednávat v kapitole o Use Case modelu.

Role pracovníků v projektu

Základní čtyři role v projektu

Zavedení tří fází, kterými jsou analýza, designu a kódování (ve smyslu metody **I + I**), s sebou přináší další důležitý efekt a tím je nutnost rozlišovat tři role neboli funkce v projektu. Podle povahy dokumentů jsou to role:

- role analytika
- role designéra
- role programátora.

Je vcelku pochopitelné, že analytik je zodpovědný za vyhotovení analytických dokumentů projektu, designér za dokumenty návrhu a programátor za kód. Analytik i designér používají syntaxi UML k tvorbě dokumentů, případně designér ještě syntaxi a prostředky daného prostředí. V případě použití relační databáze používá designér také již zmíněný ERD diagram.

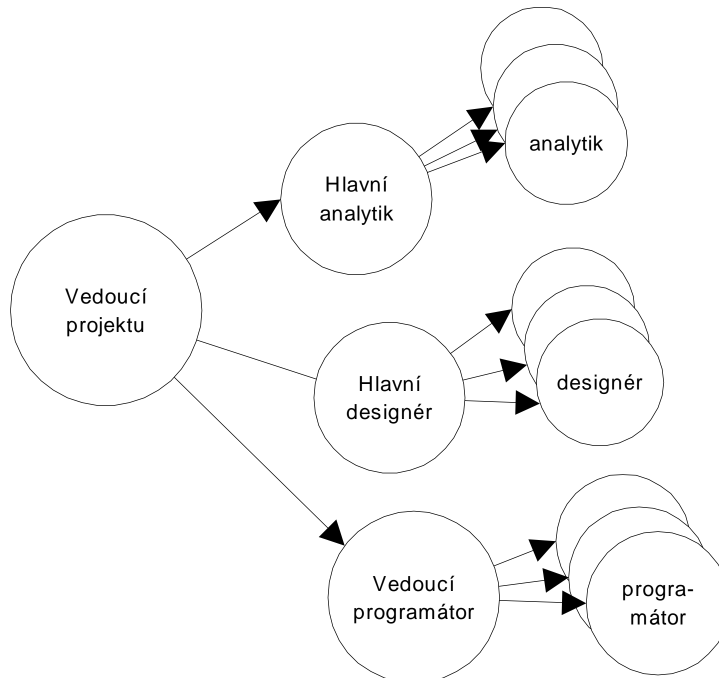
Programátor pro tvorbu svých dokumentů (zdrojový kód) využívá syntaxi daného programovacího jazyka, což neznamená, že programátor nepotřebuje znát UML. Minimálně by měl být velmi dobrým čtenářem UML, protože 90% všech dokumentů, které obdrží jako podklady ke své práci, jsou psány v UML.

Kromě těchto tří rolí existuje ještě jedna nezastupitelná role pracovníka a tím je role vedoucího projektu, který je postaven nad tyto tři role a řídí práce z vyšší úrovně nad těmito rolemi. V žádném případě však vedoucí projektu není pouhý řídící pracovník a pouze manažer. Je to „vývojář“ jako každý jiný, pouze s pravomocemi řídícího pracovníka. Měl by to být člověk také velmi dobře znalý UML, protože by mě být schopen číst dokumenty analýzy a designu.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Většinou však již ve středním projektu nemůže celou práci obsáhnout v dané roli jeden pracovník a tedy vyžaduje se zavést několik „paralelních“ stejných rolí stojících vedle sebe, například několik programátorů, u složitějších projektů několik analytiků a případně několik designérů, kteří mají spolupracovat. Jak bude vypadat skladba rolí v tomto případě?

Důležité je to, že tři vrstvy (analýza, design a kódování) musí zůstat zachovány. Uvnitř těchto vrstev vzniknou vnitřní vrstvy jako další stupeň řízení, čímž vzniknou nové role: hlavní analytik (a jemu podřízený analytik), hlavní designér (a jemu podřízený designér), vedoucí programátor (a jemu podřízený programátor), viz obrázek:



obrázek 30 Role v projektu s více pracovníky

Velmi důležité je dodržet zásadu, že za analytické dokumenty je zodpovědný hlavní analytik, za dokumenty designu hlavní designér a za kód vedoucí programátor, čemuž odpovídá pohled na vrstvy v řízení projektu.

Poznámka: Pokud bychom tuto strukturu přirovnali ke kompetencím objektů, tak například hlavní analytik (podobně hlavní designér, vedoucí programátor) je pro vedoucího projektu „interfecem“ odpovědným za dodání dokumentů. Je záležitostí „vnitřního uspořádání práce v další vrstvě“, jak tohoto docílí. Samozřejmě hlavní analytik musí být vybaven patřičnými pravomocemi vůči svým podřízeným, tj. analytikům. Má právo práci jak zadávat, kontrolovat, a ale také podřízené odměňovat případně „trestat“.

V této souvislosti upozorním na jeden velmi důležitý a praktický poznatek, který bývá mnohdy zanedbáván. Role v nejnižší vrstvě řízení, tj. analytiků, designérů a programátorů jako podřízených svým vedoucím (hlavnímu analytikovi, hlavnímu designérovi a vedoucímu programátorovi) odpovídá spíše roli „asistentů“ než úplně samostatných pracovníků. Vztah mezi hlavním analytikem a podřízeným analytikem je takový, že je to hlavní analytik, který tuto analýzu tvoří jako celek (i když je rozsáhlá), a jeho podřízení v projektu jsou spíše pomocníci, kteří mu na jeho žádost a na jeho pokyny dodávají požadované dokumenty. Není to tak, že analytici tvoří samostatně analytické dokumenty a

hlavní analytik tyto dokumenty pouze „kompletuje“ do jednoho balíku. Jeho role je mnohem náročnější a složitější. Nechává přepracovat dokumenty podle svých představ (tj. tak, aby vypadaly, jako by je dělal sám), někdy je sám upravuje a teprve poté dává dohromady do „jednoho balíku dokumentů“. Je na vedoucím, kolik práce přenechává svým podřízeným a co po nich vyžaduje. Tento princip „jedné hlavy s několika asistenty“ je v projektu velmi důležitý.

Poznámka: Všimněte si, že většina úspěšných projektů a to nejen v tvorbě SW (například Wolkswagen, Seat, Ford apod.), jsou ty, v níž vystupuje silná osobnost, která do se do projektu „položí svou duší“. Spolupracovníci této osobnosti pomáhají realizovat její vize.

Jako zvláštní pozice je mezi těmito rolemi role vedoucího programátora. Pokud použijete navrhovanou strukturu rolí v projektu a projekt se realizuje metodou **I+I**, tak programátor již nemá tak volné pole působnosti. Vedoucí programátor se v tomto rozložení prací chová spíše jako „vedoucí dílny“ tj. jako mistr v dílně, která dostává plány (např. UML dokumenty) a realizuje je v kódu pomocí svých podřízených.

Možnosti sdílení osob v rolích projektu, jejich omezení a doporučení

Velmi častou otázkou v konzultacích je dotaz, zda lze z kapacitních důvodů obsadit dvě role jednou osobou případně provádět přesuny v rolích. Odpověď je jednoduchá: Ano, lze, a dokonce je to z hlediska vývoje i žádoucí. Přece jen i přes použití **I+I** metody existuje v určitých fázích projektu větší koncentrace práce v analýze, jindy v designu a poté v kódování. Mnohdy je třeba pracovníky dočasně přesunout na jiné práce, tedy v rámci projektu řídit lidské zdroje..

Nutno však podotknout, že existují určitá pravidla, která je třeba při těchto přesunech a obsazení dvou a více rolí jednou osobou dodržet. Existuje totiž jedno reálné nebezpečí při použití „pendlování“ pracovníků v rolích. Pokud obsadíte dvě role jednou osobou, vystavujete projekt riziku, že se poruší jinak striktní oddělení vrstev analýzy, designu a kódování od sebe. Sdílení osob v rolích totiž zavádí nový zvláštní mechanismus, který umožňuje obejít předávání dokumentů mezi dvěmi osobami. Jedna daná osoba sdílí svou vlastní „paměť a znalosti“ z jiné oblasti. Vyplývá z toho, že existuje možnost, aby se teoreticky nemusela předávka dokumentů uskutečnit se zdůvodněním: „Proč mám psát sám sobě analýzu, když ji mám v hlavě“. Mnohdy se takto ve firmě postupuje z důvodu urychlení prací. Obecně se tomuto jevu snažíme zamezit, protože dochází k porušení hlavní zásady řízení projektu: Co není zdokumentováno, to neexistuje. Při sdílení osob v rolích je třeba docílit úplného „schizofrenního chování“ pracovníků v rolích: Teď jsem analytik a teď jsem designér, předávám sám sobě analýzu jako existující dokument. V praxi je však takovéto zavedení dost obtížné, protože se jeví tato činnost jako nadbytečná.

Zásadní doporučení obsazení rolí

Z důvodu možného obejití předávky dokumentů mezi analýzou a designu ze zásady nedoporučuji, aby hlavní analytik a hlavní designér byla jedna a tatáž osoba. Oddělení těchto dvou rolí i fyzicky jako dvou osob má za následek snadnější dodržování zásady vyhotovení zvlášť analýzy a zvlášť designu, což považuji za velmi důležité. Jinak řečeno, považuji za důležité, aby existovala analýza a tedy ve firmě existence samostatné osoby hlavního analytika.

Pokud není zbytí a je třeba ušetřit lidské zdroje, přičemž je nutno sloučit některé role, tak jako rozumný ústupek se mi jako vcelku dobrá kombinace z praktických zkušeností osvědčilo obsazení rolí, kdy hlavní analytik je současně i vedoucím projektu a designér jako druhá osoba současně i vedoucím programátorem. Vedoucí projektu má jako hlavní analytik velmi dobrou představu o tom, co je „cílem“. Jako vedoucí projektu realizuje manažersky svou analytickou vizi. Přitom je jako hlavní analytik povinen analýzu vyhotovit pro vedoucího designéra. Vedoucí designér a vedoucí programátor jako jeden pracovník bývá obsazen člověkem tzv. „programátorem - fanatikem“ velmi dobře znalým daného prostředí. Je to on, kdo jako technolog přímo v praxi realizuje myšlenky z analýzy a v implementaci je zavádí do života. Z hlediska svých kapacit tento pracovník „pendluje“ mezi tvorbou

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

designu, zadáváním a kontrolou prací programátorů a také sám jako jeden z nejlepších programátorů mnohdy programuje (bez toho totiž programátor – fanatik trpí abstinenčními příznaky) . Vedoucí projektu a současně analytik je zaměřen na abstraktní roviny projektu a designér a hlavní programátor bývá technokratem.

Samozřejmě je třeba zdůraznit, že optimální rozdělení rolí je, pokud to situace umožňuje, vůbec role mezi sebou nesdílet.

Základní koncepty UML

Než přistoupíme k vysvětlování jednotlivých modelů a diagramů UML, bylo by účelné popsat základní koncepty, které se celým UML prolínají. Mnoho věcí při vysvětlování konkrétních modelů a diagramů by bylo společných a proto nyní soustředíme společné koncepty do jednoho místa.

Modely v UML

Účelem použití UML je vytvořit modely informačního systému resp. softwaru. Každý model popisuje daný informační systém jako zvláštní druh pohledu na něj.

V této souvislosti položme si jednoduchou otázku: Co vlastně tvoří informační systém? Je informační systém pouze zdrojovým kódem a následně implementovaným zkompilovaným strojovým kódem?

Určitě bychom s takto omezenou definicí nesouhlasili! Nemůžeme informační systém chápat pouze takto omezeně. Každý model, který popisuje daný informační systém, se stává pojmově jeho součástí, tedy informační systém se skládá ze všech svých modelů a dokumentů včetně zdrojového kódu. Díky takovému náhledu na informační systém jako celého komplexu se tento systém lépe a kvalitněji vyvíjí a také díky vzniklé dokumentaci snáze udržuje.

Uvedený přístup různých modelů můžeme přirovnat k stavbě domu a vyhotovení jeho plánů. Existuje plán stavby stěn domu, plán rozvodů elektřiny domu, plán rozvodů vody domu, plán rozvodů odpadu domu apod. Každý z těchto plánů, tj. různých modelů domu, popisuje dům svým specifickým způsobem a dohromady vytvářejí všechny vyhotovené modely domu.

Elementy modelu v UML

UML používá celkem 9 různých modelů a všechny si zde podrobně popíšeme. Každý model používá ke svému vyjádření prvky modely, tzv. ***elementy modelu***. Tyto konkrétní elementy modelu dohromady vytvářejí daný model. Kromě pojmu model se můžeme v UML setkat i s pojmem diagram UML. Mezi modelem a diagramem existuje přímý vztah: Každý model je možné v UML vyjádřit graficky pomocí vizuálních elementů a jedno takovéto vyjádření modelu v grafické podobě se nazývá diagramem UML. V této souvislosti budeme používat oba pojmy, jak model, tak diagram.

Pojem model a metamodel

V teorii modelování se zavádí zvláštní pojem „metamodel“, který je třeba sice znát, avšak pro praktické použití nemá až takový význam. Konkrétně v UML se zavádí pojem ***metamodel UML***. Pod tímto pojmem se skrývá „model modelu UML“.

Nutno podotknout, že pokud vám teď napoprvé není jasná podstata pojmu metamodelu jako „model modelu“, tak to není nic výjimečného. Lidská mysl totiž není uzpůsobena na myšlenkové vnořování do sebe a do meta-úrovní. Je obtížné vyjádřit a chápat abstrakci abstrakce podobně jako násobné negace.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Metamodel UML, tedy model modelu UML je vlastně modelování samotného UML. Metamodel je modelem samotné syntaxe UML. Vlastně celé naše vysvětlení UML není nic jiného, než popis metamodelu UML. Například zavedený pojem „element v UML“ je paradoxně prvkem elementem metamodelu UML, stejně jako třída, objekt, asociace atd. jsou elementy metamodelu UML.

Poznámka: Pokud budete pracovat v SW firmě, která bude vytvářet informační systém pomocí CASE nástroje v UML, potom využíváte syntaxi UML pro tvorbu vašeho systému. Zajímavá situace nastane, pokud budete SW firmou, která bude vyvíjet CASE nástroj pro UML a k modelování tohoto nástroje použijete UML. V tom případě vám začnou ve vašem modelu CASE nástroje vznikat prvky metamodelu UML.

Existuje standard UML zavedený organizací OMG (www.omg.org). Zajímavé na této specifikaci UML a tak trochu paradoxní je to, že samo UML je zde vysvětleno pomocí UML. Z hlediska čistě matematického bychom to nazvali jako definici kruhem.

Tato kniha není v žádném případě učebnicí s přesnou specifikací UML poslední verze 1.3. Popisuje sice základní konstrukce tohoto jazyka s ohledem na objektové modelování, avšak přesný standard UML verze 1.3 je mnohem více teoreticky přesnější a bude mu proto věnována samostatná publikace.

Modely v UML a jejich pozice v projektu

Jak bylo řečeno, v UML existuje 9 modelů (a tedy 9 typů diagramů jako jejich grafické vyjádření). V této kapitole si ukážeme pozice těchto modelů z hlediska vývojového procesu, tj. vysvětlíme si jejich poslání a smysl ve fázích tvorby analýzy a designu. V názvech se budu držet hlavně anglických názvů s vlastním možným překladem (zatím jsem nenašel příručku a slovník češtiny pro UML, určitě zajímavý úkol pro některou z kateder VŠ ve spolupráci s češtináři):

- **Use Case diagram**, překládán jako diagram užitných případů, diagram případů užití nebo diagram užitných činností apod. Popisuje systém jako hierarchický seznam případů užití, resp. užitných činností, neboli Use Casů. Dokumenty Use Case diagramu se tvoří pouze v analýze a to hned na jejím počátku.
- **Sequence diagram**, sekvenční diagram. Diagram popisuje spolupráci objektů pomocí znázornění sekvence zaslaných zpráv. Jedna sekvence zaslání zpráv se také nazývá scénář. Vyskytuje se jak v analýze, tak v designu.
- **Class diagram**, diagram tříd (nepoužíváme z historických důvodů překlad „třídní diagram“). Diagram popisuje třídy v systému a jejich vztahy. Vyskytuje se jak v analýze, tak v designu.
- **Object diagram** nebo také Instance diagram. Diagram vyjadřuje vztahy mezi objekty a je odvoditelný z class diagramu. Vyskytuje se jak v analýze, tak v designu.
- **Collaboration diagram**, diagram spolupráce objektů. Vyjadřuje podobně jako Sequence diagram zasílání zpráv mezi objekty. Tyto dva diagramy Sequence a Collaboration diagram jsou zastupitelné. Vyskytuje se jak v analýze, tak v designu.
- **State Chart diagram**, stavový diagram. Vyjadřuje stavy objektů a přechody mezi stavy. Vyskytuje se jak v analýze, tak v designu.
- **Activity diagram,** diagram aktivit. Popisuje aktivity systému. Vyskytuje se jak v analýze, tak v designu.
- **Component diagram,** komponentní nebo komponentový diagram. Popisuje komponenty v systému, jejich vztahy. Vyskytuje se pouze v designu.

- **Deployment diagram,** diagram rozmístění zdrojů. Popisuje rozmístění strojů, jejich HW parametry, rozmístění procesů, zařízení, síť atd. Vyskytuje se pouze v designu.

Pokud je v systému v pozadí relační databáze, tak k je třeba vyhotovit ještě **ERD diagram**, který není součástí UML, protože ERD popisuje speciální ukládání dat v určitém prostředí, v relační databázi.

Poznámka: Jako poslední „model" bych ještě přidal zdrojový kód, který je také obrazem systému a jeho tvorba by se měla řídit určitými pravidly, samozřejmě není součástí UML.

Některé z modelů jsou zastupitelné. Osobně se mi osvědčilo používat prakticky takovouto „minimální" sestavu modelů:

1. Vždy zavést Use Case Model, Class model, Component a Deployment modely. Tyto modely musí být vytvořeny v celém rozsahu
2. Sequence model doporučuji používat v popisu určitých složitých scénářů a v zavedení vzorů (design patterns podle standardního chápání spolupráce ve vzoru). Při zavedení tohoto diagramu není potřeba zavádět Collaboration diagram jako jeho alternativní diagram.
3. Object diagram doporučuji zavést pro osvětlení některých vztahů mezi objekty, které nemusí být zřetelně viditelné z Class diagramu (ale třeba podotknout, že každý vztah z Object diagramu je odvoditelný z Class diagramu). Class diagram je totiž svým pojetím o úroveň abstrakce výše, než Object diagram (Object diagram ukazuje „konkrétní příklad vztahu instancí", kdežto Class diagram zobecňuje vztah mezi třídami těchto instancí)
4. V případě potřeby doporučuji zavést Stavový (State Chart) diagram pro vystižení stavů objektů a to hlavně v analýze.

Elementy v UML

Každý model a diagram v UML je složen z elementů. Nutno zdůraznit, že také interakce mezi elementy je elementem, například asociace mezi třídami je model elementem.

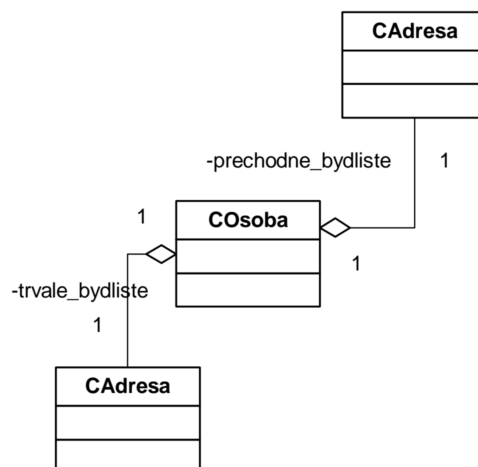
Elementy se dělí na tzv. **model elementy** a **visual elementy**. Model elementy vyjadřují esenci modelu. Visual elementy zobrazují model elementy graficky (obdélník, šipka, panáček apod.).

Pokud například provedeme v CASE nástroji report modelu, tj. výpis z modelu, nezobrazí se visual elementy, ale vypíší se vlastnosti model elementů (názvy tříd apod.).

Daný model element vyskytující se v modelu může mít několik visual elementů, které jej znázorňují, v několika nebo v jednom diagramu. Je tedy dovoleno použít pro jeden model element několik visual elementů a to dokonce několik visual elementů k jednomu model elementu v jednom diagramu pro zvýšení přehlednosti.

Příklad: Jedna třída CAdresa je znázorněna na diagramu pomocí dvou visual elementů. V modelu tedy existuje pouze jedna tato třída s názvem CAdresa, ale na tomto diagramu dva její visual elementy:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 31 : Dva visual elementy pro CAdresa (mají různou pozici), avšak jeden model element CAdresa

Společné mechanismy v modelech UML

Existuje několik společných mechanismů ve všech diagramech UML, které si nyní uvedeme a nemusíme je tak opakovat u každého diagramu zvlášť. V každém diagramu lze vždy zavést:

Tagged value

Pod tagged value máme na mysli dvojici „Název + hodnota“ (podobně jako v HTML resp. XML existuje TAG jako taková dvojice). UML umožňuje k libovolném elementu přiřadit vlastní pojmenovanou hodnotu a tím elementu přiřadit vlastnost s hodnotou.

Stereotypy

Mechanismus zavedení stereotypu umožňuje tvůrci modelu zavést vlastní kategorizaci elementů. Můžeme tedy zavést tak zvaný stereotyp a ten přiřadit k elementu. Tím je element vlastním způsobem kategorizován.

Příklad: Zavedeme stereotyp ActiveX DLL a každé komponentě typu ActiveX DLL přiřadíme tento stereotyp

Notes - poznámky

Ke každému elementu lze přiřadit poznámku, popis. Při tvorbě modelu by se nemělo na tuto „povinnost“ zapomínat, i když se tak běžně děje a tak u nepopsaného elementu po určité době se někdy nepodaří znovu odhalit původní záměry autora.

Constraints

Ke každému elementu lze přiřadit omezující podmínky (podobně jako v databází k tabulce apod.). Sémanticky není tato podmínka nijak omezena a můžeme použít vlastní jazyk (např. IF X > 0 apod.).

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Podmínkou je samozřejmě srozumitelnost. Doporučuje se použít tzv. Object Constraint Language - OCL, což je specifikace zvláštního jazyka pro tyto účely vhodného. Upozorňuji pouze, že jenom studium tohoto jazyka by znamenalo několik měsíců usilovné práce.

Dependencies

Závislosti mezi elementy jsou v UML sledovanou záležitostí a pro modelování se jedná o dost podstatnou záležitost. Pod závislostí máme na mysli tu situaci, kdy jeden element ke své existenci potřebuje druhý element a tedy pokud jeden element ze systému vyjmeme, druhý element jej bude postrádat. U každého vztahu mezi elementy bychom mohli tento vztah sledovat.

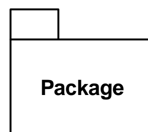
Vztah „Dependency“ může vyplynout jako důsledek jiného vztahu, který přímo implikuje tuto vlastnost. Například pokud jeden objekt z třídy CA má podřízen a používá objekt z třídy CB, tak mezi třídami CA a CB existuje vztah dependency v tom smyslu, že CA potřebuje CB „ke svému životu“. Není třeba tedy v tomto případě tento odvozený vztah dependency explicitně uvádět.

Packages v UML

V modelování pomocí UML je zaveden obecný mechanismus „grupování“ elementů do jednotlivých oblastí, které se nazývají package (balík, svazek). Package je opět model elementem, tedy prvkem modelu. Umožňuje tvůrci modelu „zabalit“ část systému. Umístěním prvků do package (přičemž do package lze umístit i package) vytváříme „části systému“ od sebe logicky oddělené.

Package patří mezi společné mechanismy UML, tedy libovolný element v libovolném modelu můžeme zařadit do package. Pokud nezavedeme v modelech žádný package, potom se celý systém chápe jako jeden „velký package“, tzv. „Top Pakcage“. Takovýto návrh s jedním package však není příliš vhodný. Zavedení několika package v modelu totiž přímo vede k logickému a přehlednému členění systému.

Pro package se v UML používá visuální prvek ve tvaru folderu:



obrázek 32 : Visual prvek pro package je ve tvaru folderu

Pravidla pro tvorbu a užití packages

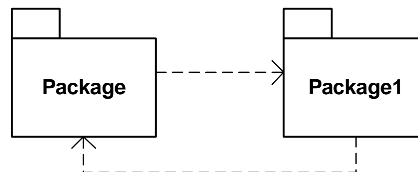
Důležité je, že členění elementů do package má svá pravidla. Každý element patří do nějakého package, odkud pochází. Pokud je jeden element ve vztahu dependency vůči jinému elementu z jiného package, implicitně z toho vyplývá také vztah dependency mezi těmito package. Tedy také tyto dva package jsou na sobě závislé, tj. i ony vstupují do vztahu dependency. Toto dependency však vzniklo díky dependency mezi vnitřními elementy z těchto package jako dependency odvozené.

Příklad:

V package P1 se nachází třída CA, v package P2 se nachází třída CB. Objekty z třídy CA používají ke své činnosti objekty z třídy CB. CA je tedy v dependency na CB a tím také package P1 je v dependency na P2.

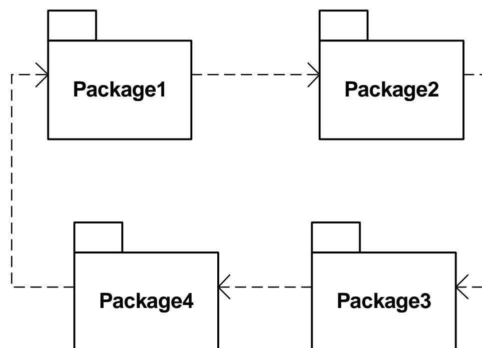
Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Pro vztah dependency mezi package však platí jedno významné omezení, které vede právě ke správnému rozvrstvení systému. Oním omezením je zamezení efektu zvaného jako **circular reference**. Ve vztahu dependency nesmí nikdy nastat circular reference, neboli kruhová reference, tj. dependency uzavřené do kruhu:



obrázek 33 Typická kruhová reference mezi dvěma package (zakázaný vztah)

Circular reference by neměla nastat pouze mezi dvěma package podle předešlého obrázku, ale neměl by nastat jakýkoliv kruh z dependency (kruh ve smyslu teorie grafu), například:



obrázek 34 : Circular reference složitější povahy (zakázaný vztah)

Vztah mezi dvěma package má totiž velmi podobnou povahu, jako linkování knihoven mezi sebou, známé již ve strukturálním modulárním programování. Ovšem v UML se nejedná o kód, ale na abstraktnější úrovni o část modelu, o package. Jeden package (část modelu) lze pouze „přilinkovat“ k druhé části modelu a tak vznikají větší celky modelu skládáním package.

Díky tomuto mechanismu se model rozvrstvuje podobně, jako když „obalujeme“ menší balíčky většími balíčky. Je pochopitelné, že by nemělo dojít k circular reference, protože potom nejde o přilinkování jedné části k druhé, ale vzájemný vztah provázanosti mezi sebou.

S trochou nadsázky lze použít protimluv: „Dva package provázané do kruhu nejsou dva package, ale je to jeden package“. Jinak řečeno Circular reference v dependency může nastat pouze mezi elementy uvnitř package.

Je to právě chyba circular reference mezi package, která v modelu tříd vede k chybě siamských dvojčat, kdy nelze od sebe oddělit dvě jinak logicky oddělitelné skupin tříd do komponent. Bude rozebráno v komponentním modelu.

Poznámka k přesnému vysvětlení ohledně syntaxe UML 1.3. Je třeba poznamenat, že sama syntaxe modelování v UML ***nezakazuje*** použít circular reference ve vztahu dependency mezi dvěma package a každý CASE nástroj vám v UML takovýto vztah povolí zavést. Ale použití circular reference v dependency mezi package vede automaticky k efektu vzájemné provázanosti těchto dvou package jako dvou siamských dvojčat. Důsledkem je ta skutečnost, že oba package spolu provázané „se

chovají jako jeden package“. Například pokud exportujete z modelu do modelu jeden package , musíte exportovat i druhý a stejně tak naopak, pokud exportujete druhý, musíte exportovat i prvý, tj. oba package „vždy jdou spolu jako jeden package“.

Z toho důvodu budeme nadále používat modely s package vždy bez circular reference.

Význam packages v UML

Přiznám se, že jsem dlouhou dobu podceňoval používání mechanismu packages v UML. Považoval jsem doporučení používat package obdobné radám typu „dělejte si pořádek na pracovním stole, bude se vám lépe pracovat“. S postupem času jsem přišel k názoru, že mechanismus packages patří k těm nejdůležitějším v UML.

Z praxe jsem dospěl k názoru, že používání packages má v UML dva hlavní důvody:

- vytváření obdoby knihoven (modulů) v UML
- zavedení komponentní technologie v systému

První bod souvisí obecně s prací v UML. Je vhodné zavést package z důvodu zpřehlednění modelů, přehledné práce s prvky modelu apod. Použití packages má velký význam například v mechanismech exportu a importu částí modelů z jednoho projektu do druhého, tj. ve sdílení částí modelů mezi projekty. Právě package reprezentuje onu sdílenou část modelu, se kterou lze v mechanismu re-use mezi modely pracovat. Zavedením packages pracujeme s přesně definovanými balíky stejně jako s knihovnami v programování a tyto balíky můžeme sdílet (linkovat).

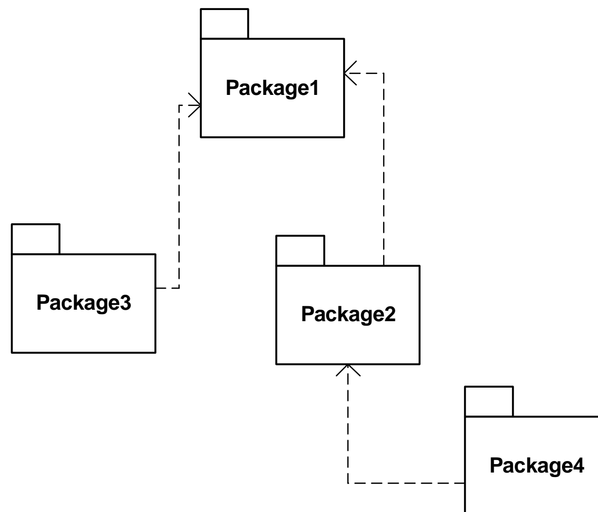
Poznámka: Opět připomeňme využití mechanismu re-use: „vytrhni, zpětně provaž a potom sdílej“. Zde se tato operace týká packages

Druhý bod použití packages souvisí s návrhem tříd a návrhem komponent. Zatímco předešlý bod se týkal jakéhokoliv modelu v UML (export a import můžeme provést v kterémkoliv modelu), existuje ještě speciální význam použití packages v modelu tříd a v modelu komponent. Zavedení packages v modelu tříd je prvním krokem ke správnému návrhu komponent. O tomto problému budeme hovořit podrobně v příslušné kapitole komponentní technologie.

Nadbytečný link mezi package a granulita systému

Mechanismus vazby mezi package si lze představit jako mechanismus „narůstajících“ balíčků od těch „nejsdílenějších“ až po celý model, například takto:

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 35 : Příklad diagramu s package

Všimněme si blíže vztahů mezi těmito package. Nejprve je zřejmé, že každý package obsahuje nějaké elementy, které jsou na sobě závislé a vztahy šipek označují, který package je ve vztahu dependency vůči druhému package. Je vidět, že Package1 je jakýmsi jádrem modelu. Package2 má dependency na Package1 a také Package3 má dependency na Package1.

Zajímavý je pohled na Package1 ze strany obou Package2 a Package3, které jsou na něm závislé. Obecně platí, že vztah mezi package typu dependency pochopitelně vůbec neinformuje o tom, který element z jednoho package je závislý na kterém konkrétním elementu z druhého package. Dva package buď závislé jsou, tj. existuje alespoň jeden element z jednoho package, který je závislý na jednom anebo více elementech z druhého package, anebo závislé nejsou, tj. ani jeden prvek v jednom package není závislý na žádném z prvků druhého package..

Z uvedeného je zřejmé, že ve vztahu dependency mezi Package2 a Package1 můžeme uvnitř Package1 rozlišit ty elementy, které nejsou pro Package2 vůbec zajímavé a na ty, které zajímavé jsou a vedou k tomuto vztahu dependency. Znamená to, že přilinkování Package1 k Package2 je sice nutné, ale teoreticky by se nemusel linkovat celý package. Tu část, která je nadbytečná pro Package2, budeme nazývat nadbytečným linkem.

Příklad

Umístíme do jednoho package všechny elementy související se všemi číselníky. Vznikne tak package všech číselníků. Pro daný package „evidence něčeho“ budeme však potřebovat pouze jeden číselník. Pro tento package lze všechny ostatní a tedy nepotřebné číselníky chápat jako nadbytečný link, ke kterému však dojít muselo (linkuje se vždy celý package).

Nadbytečný link není chybou modelu, avšak pokud je příliš velký, zvyšuje se riziko vnášení chyb do těch částí systému, kterých se problém vůbec netýká.

Na druhou stranu snaha po maximální čistotě a příliš velké zamezování nadbytečného linku způsobí vysokou granulitu systému, která může vést k velkému počtu package. Takovýto přístup vede v komponentním modelu k příliš velkému počtu komponent.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Kdy zahájit rozdělování systému do package

Rozdělování do package souvisí s celkovým pohledem na systém jako na vrstvy a nevždy se podaří tento náhled získat hned v úvodních fázích analýzy. Největší význam má tvorba packages v class modelu, protože rozložení tříd do package implikuje tvorbu komponent. Každá komponenta je v konečném důsledku reprezentována jedním nebo skupinou package, jinak řečeno rozhraní komponent nikdy neprochází napříč package. Tímto package ze tříd skládají komponenty. Protože package nevytvářejí circular reference, tak ani komponenty nemají následně jako vzniklé ze package v žádném svém vztahu dependency žádný circular reference.

Doporučuji vytvářet package v okamžiku, kdy se začínají rýsovat vrstvy systému, což nastává při odhalování více elementů a vztahů mezi nimi. V některých případech se však package odhalí dříve. V každém případě nalézání package souvisí se zvláštním specifickým pohledem na systém jako na vrstvy.

Příklad na rozvrstvení aplikace do package

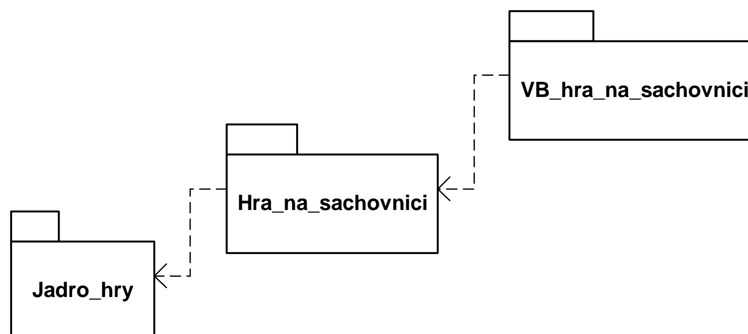
Představme si, že máme řešit aplikaci hru v šachy na síti mezi hráči jako klienty hry. Jaké budou v modelování package, do kterých budou spadat třídy? Zajímají nás hlavně třídy a package tříd.

Nejhlubším jádrem, tedy nejvnitřnější vrstvou, bude skupina entit, které drží stav hry a to nezávisle na jakémkoliv zobrazení. Můžeme si představit tyto entity jako zachycující stav hry podobně jako ve hře „poslepu", kdy se nic nezobrazuje, ale podstata hry je podchycena. Příkladem takového stavu bude kolekce již učiněných tahů zapsaná jako pohyby na šachovnici „odkud - kam" a současný stav hry na šachovnici (tj. stavy všech políček s obsazením figurek). Tato vrstva může být chápána jako „čisté šachy". Z tohoto jádra lze zobrazit například současnou pozici hry jako výpis typu: černý - e2, d7, Kd8, bílý Ke5 atd. a současně celý průběh hry. Nazvěme tuto nejvnitřnější vrstvu jako „Jádro hry".

Druhou vrstvou, která obaluje tuto vrstvu, bude vrstva, která na jedné straně spolupracuje s vnitřní vrstvou (využívá funkcionalit Jádra hry), ale sama již drží informace pro zobrazení. V této vrstvě budou takové informace, jako velikost a barva šachovnice, rozměry, typ figurek apod. Ale pozor, tato vrstva ještě nic nezobrazuje. Nazvěme tuto vrstvu jako „Hra na šachovnici". Tato vrstva například již obsahuje třídu pro šachovnici, ale drží pouze její údaje pro zobrazení, nezobrazuje ji však.

Nejvrchnější vrstva bude již čistě implementační. Rozhodneme se, že obrazovky budou napsány ve Visual Basicu. Tato vrstva si přilinkuje vrstvu „Hra na šachovnici", která obsahuje také nejvnitřnější vrstvu „Jádro hry". Nazvěme tuto nejvrchnější vrstvu jako „VB Zobrazení hry". V této vrstvě již existují PictureBoxy, Labely apod., které jsou obsahem tříd této vrstvy. Například třída „VB Šachovnice" obsahuje prvky pro zobrazení (PictureBoxy, Labely, bitové mapy atd.), které ovládá, současně má v sobě objekt Šachovnice z vrstvy „Hra na šachovnici" a tedy s údaji, „co" šachovnici zobrazit.

Jednoduchý vztah těchto packages ukazuje následující obrázek:



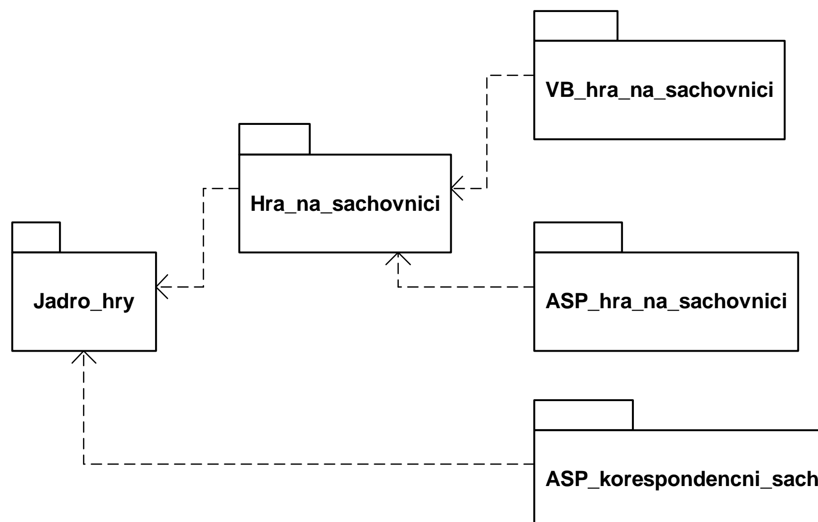
obrázek 36 : Rozložení package v šachu

V každé vrstvě je problém striktně oddělen a vrstvy jsou průhledné. Můžeme odladit velmi dobře Jádro hry a tak mít velmi dobrý základ pro další použití. Pokud odladíme velmi dobře i další vrstvu, tak se poslední package týká vskutku pouze zobrazení ve VB.

Současně máme možnost provádět různé modifikace a všimněte si, že s maximálním re-use v modelu. Například vznikne další požadavek:

1. Chceme vyvinout aplikaci pouze jako korespondenční šach bez šachovnice
2. Aplikace poběží pod ASP
3. Obě kombinace, tj. ASP, ale i korespondenčně

Model se změní takto:



obrázek 37 : Rozšíření modelu o další modifikace

Jak vidět, takovéto rozšíření je v tomto diagramu velmi přehledné s maximálním re-use. Například je zřejmé, že v package „ASP Korespondenční šach" bude formulář, který obsahuje seznam již učiněných tahů, např. v HTML pomocí TAGu TABLE, a editační pole INPUT typu TEXT pro nový tah, pokud je hráč na tahu (obdoba chatu). Po odeslání tahu přes ASP se například přes REQUEST objekt převezme tah, zavolá se Jádro hry a snažíme se mu předat tento tah ve své nejjednodušší podobě.

Podobně tomu u VB hry s šachovnicí, jenom musíme vyvinout jiné GUI, obdobu listboxu a edit pole, odchytit pohyb na šachovnici a ten převést na stejný tah tak, jak mu rozumí jádro.

Poznámka: Všimněme si na tomto příkladu i jednoho zajímavého faktu. Mohli bychom dva package „Jádro hry" a „Hra na šachovnici" dát do jednoho package. V tom případě by si package „ASP Korespondenční šach" linkoval tento společný package. Není to fatální chyba, pouze klasický příklad nadbytečného linku, protože tento package si k sobě linkuje celý balík i se šachovnicí, kterou nepotřebuje a využívá pouze určitou část přilinkovaného package.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Specifikace požadavků

Pokud jste měli tu možnost a přečetli si skripta o objektovém modelování a UML vydaná na našem serveru v roce 1999, tak jste si určitě všimli, že v úvodu se pojednává podrobně o dokumentu ***Specifikace požadavků***. Přitom, a to je možná překvapení, tento typ dokumentu není součástí UML, což nebylo v předešlých skriptech příliš zdůrazněno.

Otázkou je, proč tento dokument není součástí objektového modelování pomocí UML?

Vysvětlení je prosté: Specifikace požadavků ve své podstatě není modelem systému. Je to soupis požadavků, ale tento soupis není přímo modelem systému. Postupem času, zejména získáváním praxe, jsem si ověřil, že specifikace požadavků má sice svůj význam, nikoliv však pro modelování systému jako takového. Jeho význam spočívá v oblasti řízení projektu.

Poznámka: Z hlediska použití samotných modelů UML má nejblíže ke specifikaci požadavků tak zvaný Use Case model.

Protože specifikace požadavků nepatří mezi modely UML (není to jeden z obrazů IS), nepodřizuje se pochopitelně žádné syntaxi. Mnohdy se však tento dokument ve firmách používá, doporučuje se jej používat a proto o něm tady hovoříme. Setkal jsem se v praxi s dvěma osvědčenými způsoby, jak zavádět práci se specifikací požadavků na systém. Oba způsoby lze za určitých předpokladů, které zde uvedeme, v zásadě velmi doporučit a to zejména ve firmách, které pracují v týmech.

Specifikace požadavků jako zadání ze strany vedení projektu vůči realizátorům projektu

Zadávání prací od vedení projektu vůči analytikům skrývá v sobě určitý zvláštní paradox a protimluv, který se musí velmi praktickou cestou vyřešit. Tento paradox spočívá v tom, že když už analytici mají zadánu práci a tedy pracují, tak v té chvíli „odhalují" pomocí analytických modelů funkcionalitu systému, tj. vytvářejí analýzu pomocí modelů, ve kterých se popisuje „co bude systém provádět, až bude naprogramován".

Na druhé straně představme si, že se před touto prací vyžaduje, aby existovalo zadání ze strany vedoucího projektu vůči analytikům. V té chvíli je otázkou, co má toto zadání obsahovat a jak má být úplné?

Je jasné, že toto zadání nemůže podat úplný soupis požadavků ve smyslu všech požadavků na veškerou funkcionalitu systému, protože ta se má teprve odhalovat a ta je předmětem analytických prací. Ze strany vedení projektu není smyslem jejich práce, aby se nahradila práce analytiků psaním rozsáhlých a podrobných zadání.

Proto se mnohdy zavádí ***Specifikace požadavků*** jako zvláštní typ dokumentu, který reprezentuje takovéto zadání od vedení k analytikům. Neexistuje žádná syntaxe, tj. dohoda nad tvorbou takovéhoto dokumentu. Je třeba si uvědomit, že takto pojatá specifikace požadavků, tj. jako zadání od vedoucího projektu vůči hlavnímu analytikovi, **nemůže být ze své podstaty úplná**, ale musí být sepsána pouze v obecnější rovině. Dokument tak obsahuje pouze základní prvky funkcionalit. Zadání se stává pouze výchozím bodem pro další práci.

Protože nemůže být toto zadání úplné, nemá smysl toto zadání porovnávat s výsledky práce analytiků jedna ku jedné. Rozsah analytického dokumentu je mnohem více obsažný. O mnoho větší význam než toto úvodní zadání má pro další zdárný průběh projektu následně vytvořený model zvaný Use Case model, jak si uvedeme dále.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Postup vedoucího projektu při tvorbě specifikaci požadavků

Osobně doporučuji jako osvědčený takovýto postup: Vedoucí projektu vypracuje stručnou specifikaci požadavků, nebo-li zadání, které se chápe jako rámcový dokument. Hlavní analytik (při konzultacích také s vedoucím projektu) vyhotoví co nejdříve úplný Use Case model (o tomto modelu viz dále). Důležité je, že tento model již podléhá všeobecným standardům UML na jeho tvorbu, musí mít určitou formu a určitou syntaxi. Tento model je po odsouhlasení vedoucím projektu „nejvyšším dokumentem pro další práci“. Use Case model tak plně přebírá funkci zadání pro další tvorbu jiných modelů. Důležité je hlídat konzistenci modelů, protože odsouhlasený Use Case model je prvním existujícím modelem systému, stává se „závazným“ a jeho změny podléhají přísným pravidlům, viz dále v kapitolách o použití Use Case modelu pro řízení projektu.

Řízení evidence požadavků ve firmě

Existuje ještě druhá velká oblast, kam spadá práce s požadavky. Mnohdy bývá pro firmu velmi výhodné, aby si vedla evidenci požadavků na své produkty a tvořené systémy zvlášť jako speciální vnitrofiremní agendu. Vždy, když je vznesen nějaký požadavek od kohokoliv na funkcionalitu systému (například od uživatele, od analytika, od vedoucího projektu apod.), tak se tento požadavek ve firmě zaeviduje, a poté se s ním začne „nějak pracovat“. Vzniká tak nová logistická vnitrofiremní agenda jako řízená evidence požadavků na SW produkty firmy.

Například se eviduje:

- kdy požadavek vznikl,
- kdo požadavek vznesl,
- důležitost požadavku,
- ve které verzi se bude realizovat,
- vztah k modelům (hlavně k Use Case modelu, tj. kterého Use Case se týká apod.),
- aj.

Existují aplikace, které tuto a podobné evidence požadavků podporují. Pomocí řízené evidence požadavků vznikají specifikace požadavků podle předešlého odstavce systematicky a efektivněji. Pro firmu to samozřejmě znamená zavedení vyšší kvality do metod řízení projektů.

Tento přístup pomocí softwarem řízené evidence požadavků ukazuje názorně, jaká je vlastně poloha požadavků na systém z hlediska modelování. Specifikace požadavků je jakýmsi „informačním backgroundem“ pro modelování a požadavky se evidují zvlášť od modelů.

Use Case model

Use Case Model, tj. model užitných činností, též překládán jako model užitných případů, se začíná tvořit v úplně počáteční fázi analýzy. I když se současně s modelem aplikace popisuje okolí aplikace a tím se také vymezuje hranice aplikace (co do systému patří a co už ne), největší důraz se klade hlavně na nalezení všech užitných činností systému.

***Hlavním posláním Use Case modelu je získat celkový seznam všech užitných činností systému*.**

V této souvislosti se samozřejmě naskýtá otázka: Co to je jedna užitná činnost systému, jeden Use Case?

Element Use Case, užitná činnost systému, případ užití

Prvek Use Case neboli užitná činnost v systému, specifikuje jeden prvek funkcionality systému. Nejlépe si uvědomíme význam užitné činnosti na základě požadavku na její zrod.

Na počátku jedné užitné činnosti existuje jeden prvek informační nerovnováhy mezi okolím a naším systémem. Tato jedna nerovnováha je charakterizována jako (+ / -) nebo (- / +) ve vztahu okolí versus informační systém. Tato nerovnováha vede k požadavku na určitou funkcionalitu systému, která tuto nerovnováhu vyrovnává.

Příklad: Obsluha drží „v ruce“ novou fakturu a chce ji zadat do systému. V okolí existuje faktura (+), která není v systému (-). Obsluha po zadání nové faktury do systému vyrovnává tento deficit (do stavu 0 / 0).

Jiný příklad: Naopak může nastat situace, kdy něco je obsaženo v systému a není dostupné v okolí (- / +). Obsluha potřebuje získat přehled o bonitě klienta banky (-) a údaj je v systému (+). Po vyžádání informace se deficit vyrovná (0 / 0).

Celý systém se v takovémto pohledu skládá ze samých užitných činností vyrovnávajících takovéto deficity, tj. celková činnost sytému je neustálé takovéto vyrovnávání informačních deficitů. Každá takováto činnost charakterizuje určité užití systému okolím. Z hlediska funkcionality je jedna užitná činnost systému chápána jako to, co vede v konečném důsledku ke splnění určité požadované funkcionality, tj. vyrovnání deficitu informací okolí versus systém.

Jakmile je nerovnováha informace vyrovnána, systém setrvává v jakémsi IDLE stavu v očekávání dalších požadavků.

Syntaxe UML pro jeden Use Case v modelování

Jeden model element Use Case je jednoznačně identifikován mezi ostatními Use Casy svým názvem. Visuálním elementem pro jeden Use Case je elipsa.

Obrázek:

obrázek 38 Visual prvkem Use Case je elipsa

Užitné činnosti a princip jejich vyhledávání pomocí hierarchie

Bylo by možné si sednout a sepisovat užitné činnosti na jednu hromadu (například při rozhovoru a konzultaci s uživatelem). Na co by se vzpomnělo, to by se zapsalo. Při tomto postupu nejenom že riskujeme, že některou z činností zapomeneme, ale takovýto seznam by byl také velmi nepřehledný.

Poznámka: Je třeba však podotknout, že pokud by byl úplný, mohl by být takovýto seznam Use Casů považován za technicky přijatelný, protože teoreticky obsahuje celý popis systému z hlediska užitných činností a to je vlastně celý Use Case model.

Aby se zabránilo nepřehlednému vyhledávání užitných činností, zavádí se hierarchické členění Use Casů mezi sebou.

Postup je v úvaze následující: Postupujeme odshora dolů. Při tomto pohledu shora je celý systém chápán jako souhrn všech užitných činností, které má systém obsahovat. Tato nejhořejší úroveň se rozpadá logicky na několik úrovní dalších podřízených souhrnů užitných činností o úroveň níže. A tak postupujeme stále dolů rozkladem užitných činností.

V praxi jsem se setkal se dvěma možnými přístupy k tvorbě hierarchie, oba jsou podle mne možné a nelze podle mne rozhodnout, který z nich je „ten vhodnější".

První z nich používá k tvorbě hierarchie Use Casů mechanismus package. Zavádí se jednotlivé package Use Casů s dekompozicí odshora dolů, přičemž každý package je chápán jako skladba dalších packagů resp. Use casů. Tímto mechanismem se Use case model hierarchicky rozčlení podle skladby daných packagů až po ty „nejnižší" package, které obsahují již pouze nejnižší Use Casy.

Poznámka: Podle přesné syntaxe UML je tento proces tvorby hierarchie pomocí package správný.

Druhý způsob nepoužívá mechanismus package. Tento přístup rozlišuje Use Casy, které jsou „hierarchickou skladbou" jiných Use Casů, což však z hlediska UML syntaxe není úplně přesné, protože UML nezná interakci typu hierarchického skládání mezi Use Casy. Avšak tato metoda je často v praxi používána. Podle ní existuje tedy rozklad samotných Use Casů směrem dolů. V tomto postupu rozeznáváme Use Casy, které mají povahu souhrnu užitných činností (tj. Use Casy, které se dále člení na další, vlastně se jedná o synonymum pro package) a oproti tomu ty Use Casy, které jsou posledními, tj. listy tohoto hierarchického stromu. Právě tyto poslední listy jsou činnostmi, která vyrovnávají „jeden deficit", kdežto užitné činnosti nad nimi jsou chápány jako souhrny (skupiny) takovýchto užitných činností.

Je na každém z nás, který z těchto dvou způsobů tvorby si vybere. Pracoval jsem s oběma a nevidím v nich žádný velký rozdíl. V dalším budeme popisovat hierarchický rozklad modelu druhým způsobem, tj. rozkladem samotných Use Casů a nikoliv použitím mechanismu package.

V každém případě při postupu rozkladu odshora dolů je třeba dodržet určité zásady. Pokud se určitá užitná činnost rozpadá hierarchií na několik užitných činností, tak musí platit pravidlo úplnosti:

Souhrn všech užitných činností na spodnější úrovni hierarchie zahrnuje všechny činnosti v souhrnu nad nimi a tedy žádná z nich nesmí nechybět. Při dekompozici odshora až dolů tedy provádíme kontrolu úplnosti na každé úrovni rozkladu.

Pro vysvětlení: Nechť užitná činnost, která ještě není listem užitných činností, tj. má být dekomponována, se jmenuje A. Popisem takovéto činnosti může být například tato věta: „Užitná činnost A je souhrn všech činností, které zahrnují A“. Například užitná činnost se nazývá „Práce s fakturou“, její popis je: „všechny činnosti, které zahrnují práci s fakturou“. Tato užitná činnost se například rozpadla na Use Casy „Nová faktura“, „Editace neodeslané faktury“ a „Odsunutí faktury do archivu“ aj. Otázka kontroly úplnosti zní: Jsou opravdu nižší Use Casy souhrnem všech činností „Práce fakturou“? Nechybí některý (například „Import faktury“ apod.). Tímto postupem je zabezpečena úplnost celého Use Case modelu.

Při vyhledávání nižších úrovní buď konzultujeme s uživatelem systému anebo s někým znalým problematiky. Většinou se kladou otázky typu:

- Jaké jsou úkoly aplikace?
- Bude se z okolí přidávat, ubírat nebo měnit informaci v systému a jaké?
- Bude okolí potřebovat informace o změnách v systému a jaké?
- Jaké změny v okolí systému povedou k přísunu informací do systému?
- Jaké činnosti jsou třeba k administraci systému?
- apod.

Tyto otázky a následné odpovědi vedou k vyhledávání odpovídajících užitných činností, k jejich rozkladu odshora až dolů a nalezení všech listů Use Case modelu.

Hierarchie v Use Case

Jak bylo řečeno, v praxi existují při objektovém modelování dva způsoby vyjádření hierarchie v Use Case modelu. Jeden používá pro UML jednotného a společného mechanismu tvorby package, v tomto případě z Use Casů, druhý zavádí Use Case „na vyšší úrovni a nižší úrovni“ a tím je tvořena hierarchie přímo mezi Use Casy.

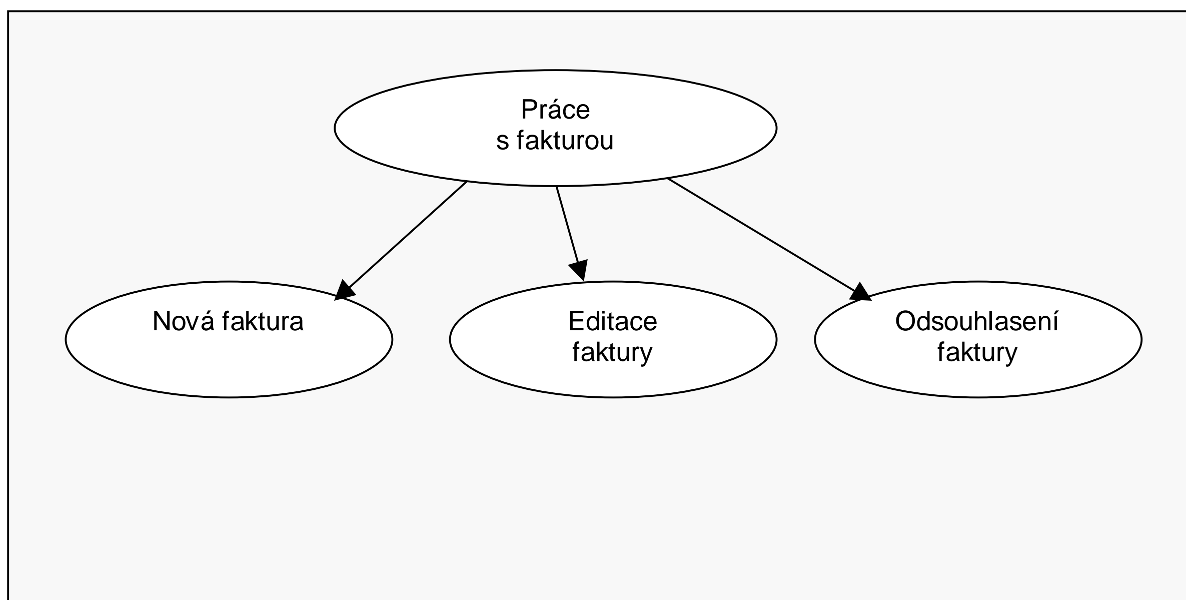
První způsob pomocí package je z hlediska syntaxe UML přesnější, ale v praxi se mnohdy používá i druhý způsob, protože se jeví jako velmi přehledný.

V této kapitole je popsán druhý způsob, kdy existují Use Case v hierarchii na vyšší a nižší úrovni (existují Use Case jako souhrny činností na vyšší úrovni).

Mezi jednotlivými Use Casy se mohou v uvedeném přístupu vyskytovat jednosměrné interakce vztahy mezi Use Casy jako hierarchie. Tyto vztahy jsou v UML chápány jako další model elementy a jejich visual elementem je šipka a jednou z těchto interakcí je již zavedená hierarchie od elementu Use Case „nahoře“ k elementu „dole“ (poznámka: tj. nikoliv přesně podle syntaxe UML 1.3, protože ta nezná tento typ interakce mezi elementy Use Case).

Tento vztah je vcelku zřejmý z předešlé kapitoly a slouží k přehlednému rozkladu užitných činností v hierarchickém členění. V diagramu nejlépe znázorníme vztah hierarchie šipkou bez žádného dalšího popisu:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 39 : Rozklad hierarchie Use Casů

Je pochopitelné, že na každé úrovni hierarchického členění lze přidávat další elementy „vedle sebe" na stejné úrovni. Například dalším elementem v předešlém obrázku by bylo „odeslání faktury" apod.

Druhý způsob je sice podle UML přesný, ale není až tak přehledný: Skupinu „vyšších" Use Casů vyjádříme jako package Use Casů a rozklad provádíme pomocí package mezi sebou (package skládá package), až na poslední úrovni je package složen z Use Case. Tedy to, co je v předešlém přístupu Use Case na vyšší úrovni, je zde reprezentováno jako package.

Poznámka: Specifikace UML 1.3 například tento postup pomocí package přímo nabízí i pro modelování business procesů v jednom ze svých standardních profilů, blíže bude pojednáno v připravované publikaci „Stručná specifikace UML 1.3".

Důležité zásady pro tvorbu Use Case modelu

Mějte vždy na paměti, že hlavním cílem Use Case modelu je nalézt úplně všechny užitné činnosti jako listy hierarchického stromu modelu a provést poté jejich konzistentní popis. Ostatní pracovní postupy zavedené v tvorbě Use Case modelu, jako jsou vyhledávání a tvorba hierarchie Use Casů, určení prvků okolí systému apod., jsou **pouze pomocnými metodami** (i když někdy nezbytnými) na cestě jedinému cíli: Získat všechny Use Casy jako listy hierarchie.

Konzistence Use Case modelu a „zlaté Use Casy"

Při tvorbě Use Case modelu postupujeme odshora dolů postupným logickým rozkladem (buď pomocí package anebo vztahu mezi Use Case jako hierarchie). Pokud přidáváme nový odhalený Use Case, tak jako analytici musíme dobře sledovat ***konzistenci modelu***. Jeden nový Use Case s sebou přináší nutnost zavést jeden nebo více dalších jiných Use Casů v jiných částech modelu (doslova na druhém konci systému), které jsou nutné k tomu, aby vůbec tento náš nový Use Case mohl fungovat. Problém je v tom, že pokud v novém Use Case existuje nějaká informace, se kterou musí nový Use Case pracovat, tak pochopitelně musí existovat jiný Use Case, který s touto informací nějak nakládá „z druhé strany", který nám tuto informaci nějak připravuje pro nově zaváděný Use Case.

Například když potřebujeme pro fakturu vybrat Odběratele, tak musí existovat jeden a více Use Casů, které pracují s Odběrateli. Činnosti musí nějak tento seznam připravit s veškerým nutným komfortem, jako editace prvků apod. Pokud na tyto Use Case pozapomeneme, tak z pochopitelných důvodů nebude systém moci správně fungovat, protože seznam nebude použitelný.

Při dobrém sledování konzistence se začnou objevovat další a další Use Casy nutné pro obsluhu nově přidávaných Use Casů a model tak „potěšeně roste“. Z hlediska významu Use Case (nikoliv z hlediska nutné funkčnosti) můžeme tedy rozdělit Use Casy na zlaté Use Casy a „obslužné“ Use Casy. Zlaté Use Casy jsou ty, pro které systém vůbec existuje a jsou důležité z obchodního hlediska. Obslužné Use Casy musí být zavedeny jenom z toho důvodu, aby zlaté Use Casy mohly fungovat. Toto rozdělení je však pouze z hlediska pohledu obchodního významu funkcionality, nikoliv z hlediska samotné funkcionality. Aby systém mohl dobře fungovat, musí obsahovat všechny Use Casy, jak všechny zlaté, tak všechny obslužné.

Samozřejmě, stejně jako ve všech oblastech lidské činnosti, i zde platí obdoba jakéhosi všeobecně platného Murphyho zákona, který můžeme v souvislosti s tímto postupem vypozorovat:

Nepodstatných činností, které jsou však nezbytné pro to, aby mohly být fungovat věci podstatné, je více než 99% ze všech činností dohromady.

Opravdu, většina Use Casů má povahu „okolních podpůrných tanečků“ nutných pro to, aby několik málo Use Casů, mohlo fungovat. Při tvorbě Use Case modelu zahajujeme vždy naše úvahy u zlatých Use Casů. Tyto Use Casy začnou rodit pomocné obslužné Use Casy, které však také musí v systému existovat.

Postup je tedy takový, že vytvoříme nějaký „zlatý“ Use Case a postupně od něj „odbíháme“ pro tvorbu nových obslužných Use Casů.

Poznámka: Je třeba podotknout, že tuto radu nemusíme příliš zdůrazňovat, protože většinou takto postupujeme intuitivně. Pouze jedinci obzvlášť nadaní schopností všechno zesložiťovat si najdou nějakou neschůdnější cestu a začnou tvořit model u nejnepodstatnějších a vedlejších Use Casů.

Postup rozšiřování Use Case modelu při sledování konzistence odpovídá požadavku na úplnost Use Case modelu a má velký význam pro použití Use Case modelu při řízení projektu (bude pojednáno na konci kapitoly o Use Case modelu).

Omyl ohledně nejednoznačnosti rozkladu hiearchie

Častým omylem je přeceňování samotného „významu hierarchie“ Use Case modelu, ať už použijeme rozklad pomocí package anebo rozklad samotných Use Casů. (Dokonce z hlediska syntaxe UML jsou Use Case pouze ty, které jsou v našem pojetí listové a vztah hierarchie zaveden pomocí package).

Hierarchie je určitým pohledem analytika na daný systém a ten je hodně subjektivní. Je třeba si uvědomit, že pokud daný systém bude zkoumat nějaký analytik, s největší pravděpodobností u téhož systému navrhne jinou hierarchii rozkladu užitných činností než jiný analytik. Výsledek rozkladu totiž závisí na pohledu každého z nás.

Avšak úplná libovůle zde neplatí: Oba Use Case modely téhož systému se musí shodovat v celkovém výčtu seznamu koncových listů a **tento seznam musí být vždy stejný, protože ten je pro Use Case model podstatný**. Hierarchická cesta k listům Use Casů je pouze různým úhlem pohledu. Po tvorbu hierarchie lze tedy vyžadovat dvě základní podmínky:

1. musí být zabezpečena úplnost všech listů modelu (model je tzv. konzistentní)

2. hierarchie (ať je tvořena pomocí package anebo rozkladem Use Casů) musí být přehledná a musí odpovídat přirozenému a logickému pohledu na systém

Omyl ztotožnění hierarchie Use Case modelu s rozkladem na komponenty

Při dekompozici odshora dolů jsem při konzultacích u svých klientů vypozoroval jednu velmi zavádějící myšlenku: Zdá se, jako by hierarchie v Use Case modelu na první pohled vytvářela současně s tímto rozkladem také rozklad na části systému, moduly, komponenty, knihovny apod. Někteří pracovníci se pokusí rozčlenit systém na moduly a komponenty podle hierarchie v Use Case modelu a podle toho rozdělují systém a také práci mezi členy týmu.

Zde musím upozornit na jedno úskalí: Při tomto ztotožnění buďme však velmi opatrní a doporučuji se mu raději vyhnout, protože může být velmi zavádějící. O tom, jak hledat správně části systému jako jsou komponenty, bude pojednáno ve zvláštní kapitole. Problém je totiž v tom, že jednotlivé Use Casy samy o sobě ve své hierarchii určitě nevytvářejí dobrý přehled o členění systému na moduly nebo na komponenty. Jinak řečeno systém zobrazený v pohledu Use Casů není dobrým vodítkem pro členění na komponenty nebo moduly a proto jej nedoporučuji použít pro členění systému na komponenty.

Mnohdy si tvůrce systému neuvědomí, že jeden Use Case, tj. list našeho stromu, realizovaný jako scénář, může „proběhnout“ několika komponentami a předávat kompetence z jedné komponenty do druhé. Právě v tom spočívá záludnost použití Use Casů pro návrh komponent. Vztah mezi Use Casem, tj. listem stromu a komponentami je obecně 1 : N, protože pro realizaci daného Use Casu se použije buď jedna anebo hned několik komponent. Například v Use Casu „Nová Faktura“ se vybírá Odběratel (a ten třeba patří do komponenty Firmy), vybírá se u řádku faktury resp. položky zboží dané Zboží (to patří do komponenty Zboží) atd. Znamená to, že v tomto Use Casu se použije hned několik komponent. Na příkladu je také patrné, že důležitější pro návrh komponent jsou pojmy použité v Use Casech (později přecházejí pojmy na třídy) než samotné Use Casy.

Popis Use Casu

Každý model element typu Use Case má dvě podstatné vlastnosti: Název Use Casu pro jeho identifikaci (mezi Use Casy jednoznačná) a Popis Use Casu, umisťovaný do Note, (Note má každý model element, viz společné mechanismy v UML).

Pozice důležitosti popisu u Use Casu je však v tomto modelu o mnoho vyšší, než je pozice popisu u jiných typů elementů v jiných modelech. Zatímco u jiných typů model elementů má popis (tj Note - poznámka) spíše význam vysvětlující, v Use Case modelu patří ke stěžejním. V Use Casu je získání popisu oproti tomu jedním z hlavních cílů ztvárnění Use Casu. Tedy je to popis Use Casu, který je podstatnou vlastností Use Casu a na něj je zaměřena hlavní pozornost analytika.

V popisu Use Casu je jednoduchými slovy popsána daná užitná činnost. Analytik vysvětlí stručně a pouze v pojmech problémové domény, co se odehraje, až je nakonec splněn požadavek na funkcionalitu. Zdůrazňuji zde „pouze slovně a pouze s pojmy problémové domény“. V žádném případě se v popisu nesmí objevit pojmy z programování, z použité technologie apod. Důležitá zásada je ta, že uvedený popis napíšeme stejně, jako by ho psali uživatelé plně neznalí programování. Uvedený popis musí být v pojmech na takové abstraktní úrovni, která zaručí platnost uvedeného popisu v libovolné technologii (třeba i v evidenci na papíře). Nezávislost tohoto popisu na použité technologii musí být důsledně dodržována, jinak se Use Case stane řešením pro určitou platformu, pro určitý speciální případ, a navíc, v konečném důsledku se může takovýto specializovaný popis stát pro uživatele velmi nesrozumitelným.

Poznámka: V některých vysvětleních použití Use Case modelu (například při použití Rational Rose) jsem se setkal se speciálním názvem takovéhoto popisu jako event-flow.

Popis Use Case a pojmové programování a usnadnění přechodu na OOP

Jedním z hlavních pozitivních důsledků dodržení zásady „vyhni se v Use Casu technologickým podrobnostem a použij popis pouze pomocí pojmů problémové domény“ je snadnější přechod na

objektové programování. Platí nepsané pravidlo, že kdo tuto zásadu poruší, nedospěje do OOP a tedy ani do UML. Uvedený přístup zachování abstrakce a ponechání popisu pouze v rovině pojmů problémové domény totiž plně odpovídá objektovému pohledu. Jednotlivé použité pojmy v popisu Use Case se stávají kandidáty na objekty a to odpovídá objektovému pojetí.

V čem je přínos pro objektový pohled? Na vysvětlenou porovnejme tyto dvě části různých popisů Use Casu:

První, bližší objektovému pohledu, zachovává správnou úroveň abstrakce:

…Obsluze se zobrazí seznam firem (Název firmy, IČO). Obsluha vybere firmu a ta se dosadí do Editované faktury…

Druhý popis, a podotkněme, že chybný, je napsán zkušeným programátorem, který nepopisuje Use Case pomocí pojmů problémové domény, ale pomocí programátorských pojmů. Autor nabízí určité „technologické řešení“, které však nutně zavádí do strukturálního programování:

…Z tabulky Firem funkce načte údaje IČO, Název a Id firmy. Obsluze se z těchto údajů zobrazí Grid IČO, Název a hidden Id. Obsluha vybere jeden row v Gridu podle Iča a Názvu a tím vybere Id. Dané Id se dosadí jako cizí klíč do chyceného záznamu faktury.

Všimněte si podstatného rozdílu mezi oběma pohledy. První je abstraktnější, jednodušší, čistší a odpovídá objektovému myšlení. Druhý pohled je kostrbatý, zbytečně složitý, poplatný dané technologii (databázím a ERD). Navíc je zřejmé, že druhému popisu nebude uživatel (a mnohdy ani kolega programátor) rozumět.

Poznámka: Představme si, že bychom měli jako konzultanta paní účetní, která v životě neprogramovala a té musíme vysvětlit, co je to cizí klíč..

Pokud se blíže podíváte na první popis, všimněte si, že popisuje esenci problému. Pro mnohé programátory bývá jejich navyklý pohled z hlediska implementace funkcí, proměnných a tabulek, SQL příkazů atd. příliš velkou překážkou pro jejich přechod na objektové myšlení. Z praxe jsem při konzultacích získal jeden důležitý poznatek, který svědčí o tom, jak je toto místo, které nyní čtete, pro přechod na OOP kritické: Každý, kdo byl nakonec od určitého okamžiku schopen napsat Use Case v pojmech problémové domény, tak se od tohoto okamžiku začal překlápět na objektové myšlení.

Element interakce „uses“ mezi Use Case a první vyhledávání re-use v systému

Mezi dvěma Use Casy může nastat z analýzy vypozorovaná jednosměrná interakce, která se nazývá podle starší syntaxe UML 1.2 jako „**uses**“, podle nové verze UML 1.3 jako „**include**“. Vysvětlíme si interakce nejprve podle syntaxe verze UML 1.2 a poté uvedeme rozdíl ve verzi UML 1.3. Jedná se totiž o určitý „pojmový skok“, který je třeba dobře pochopit, proto vyjdeme z „historicky starší verze“ UML 1.2.

Představme si, že píšete nějaký Use Case tj. jeho popis a najednou získáte dojem: „To jsem už jednou prožil, tedy přesněji řečeno psal“. A opravdu zjistíte, že celá jedna pasáž v jednom Use Case odpovídá celé jedné již napsané pasáži v jiném již existujícím Use Case. Jedná se o shodu opakováním. Můžete sice použít metodu přenosu pomocí schránky (clipboardu), ale to určitě není šťastné řešení…

Provedeme proto úpravu typu re-use: Společnou pasáž vyjmeme z obou Use Casů a provážete ji zpět na místo, odkud je „volána“. Vznikne tak nový „sdílený“ Use Case, který je v re-use vůči oběma původním Use Casům. V určitém bodě se použije „odskok“ na jiný Use Case.

Poznámka: Podobně se volá funkce ve strukturálním programování. Uvedená sdílená pasáž se již neopakuje a je obsažena (dokonce se stejnou syntaxí „include“ nebo „uses“ v C++ nebo Delphi).

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Pokud provedete zásah do sdíleného Use Case, projeví se změna pochopitelně díky sdílení v obou Use Casech.

Visual elementem uses resp. include je šipkou spolu se stereotypem označeným takto: <<include>>, ve starší verzi UML jako <<uses>>. Jedná se tedy o vyznačení stereotypu

Příklad

Jako klasický příklad použití uses resp. include uveďme použití této interakce při výběru ze seznamu entit a daná entita se v seznamu nevyskytuje. Potom je žádoucí, aby se mohlo z Use Casu výběru odskočit do Use Casu „Nový item seznamu“, přidat prvek a poté se vrátíme zpět. Například v užitné činnosti Nový úvěr se vybírá Ručitel, který se má dosadit do nového Úvěru. V činnosti existuje následující sekvence vět:

„…Obsluha vybere Osobu ze seznamu Osob. Pokud se daná Osoba v seznamu nevyskytuje, potom se volá užitná činnost Nová osoba, která se poté dosadí do nového Úvěru“.

Mezi užitnými činnostmi Nový úvěr a Nová osoba se nalezl vztah uses resp. include. Znamená to, že užitná činnost Nová osoba je užita (tj. uses tedy includována) do užitné činnosti Nový úvěr, což si můžeme představit podobně jako u volání funkce.

Je třeba hned na začátku zdůraznit, že vztah uses resp. include je „zapouzdřen“ dovnitř užitné činnosti, která používá jiný Use Case. Znamená to, že když Nový úvěr má uses resp. include na Novou osobu, tak tento vztah je uschován dovnitř Nového úvěru a „zvenku viděno“ o tomto vztahu nevíme nic. Častou chybou při chápání vztahu uses resp. include je to, že vnější činnost se mylně chápe pouze jako ta část Use Case, která je pouze na straně „bez include“ a vedle stojící existuje includovaný. Správně se musí Nový úvěr chápat jako celý i s činností nová osoba „dohromady“.

Co je hlavní důvod hledání re-use v Use Case modelu?

Pokud nalezneme vztah mezi dvěma Use Casy typu uses resp. include a tím zavedeme re-use, tak si vcelku pochopitelně ušetříme práci při tvorbě tohoto modelu. To je jeden příznivý důsledek použití re-use, ale ušetřit si práci v Use Case modelu není ten hlavní důvod, proč re-use vyhledáváme.

Ušetření práce při psaní je pěkné, ale mnohem důležitější je nalezení re-use v systému obecně jako výsledek analytického hledání. Pokud totiž nalezneme re-use už v Use Case modelu, bude se tento re-use prolínat do dalších modelů a to je to podstatné.

Při této příležitosti je třeba připomenout, že obecně nepoužití re-use tam, kde je třeba, není fatální chybou. Je to sice vážná chyba, protože opakování v systému má nepříznivé důsledky (viz předešlé kapitoly o zavádění re-use v IS), ale systém je stejně funkční jak před zavedením re-use při opakování elementů v systému, tak po jeho odhalení a zavedení.

Element interakce mezi Use Case typu „Extends“

Interakci typu uses (verze UML 1.2) resp. include (verze UML 1.3), která byl popsána v přešlých kapitolách, lze chápat jako „odskok“ do sdíleného Use Case. Tímto způsobem řešíme re-use jako sdílení Use Casů jejich vzájemným voláním. Existuje ještě jeden možný typ vztahu mezi Use Casy, který také řeší re-use, a tím je vztah zvaný **extends**.

Důležitá poznámka: Zde v této kapitole je vysvětlen tento vztah extends pomocí syntaxe starší verze UML 1.2 a poté v další kapitole přejdeme na verzi 1.3. Důvodem je historická posloupnost verzí v chápání tohoto vztahu.

Je zajímavé, že když jsem při konzultacích vysvětloval Use Case, tak většina z těch, kteří již UML alespoň částečně znali, se přiznávali, že vztah extends jim není až tak jasný. Tato nejasnost souvisí s úvahami při přechodu na OOP.

Hned úvodem k vysvětlení extends je třeba říci, že rozdíl mezi extends a uses je z hlediska teorie modelování sice „podstatný", ale z praktického hlediska (lépe řečeno pragmatického hlediska), záměnu uses a extends nepovažuji v Use Case modelu za fatální chybu.

Poznámka: Teď samozřejmě vyslovuji tento poznatek na základě praktických zkušeností, pokud tyto články čte některý z expertů na UML, určitě souhlasit nebude, nechť tedy promine...

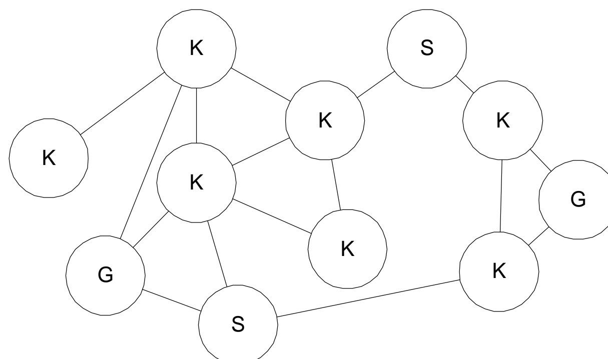
Jinak řečeno, tvrdím, že záměna extends za uses při tvorbě Use Case modelu je sice chybou, ale domnívám se, že to není fatální chyba modelu. Z čistě praktického hlediska doporučuji následující postup šetřící čas a náklady:

Nejprve všechny re-use zapišme pomocí uses a neuvažujme o extends. Postupujme podle předešlé kapitoly. Po určité době projděme tato uses a posuďme, zda se u daného vztahu uses se nejedná o chybné určení typu a změňme typ uses na typ extends.

V žádném případě nedoporučuji věnovat nějaký čas diskusi, zda se jedná o interakci typu uses anebo interakcí typu extends. Největší chybou by bylo povolit několikahodinovou diskusi N pracovníků, zda uvažovaný vztah spadá pod extends nebo pod uses (podle UML 1.3 pod include).

Vztah extends zavedený ve verzi UML 1.2 si vysvětlíme na velmi ilustrativním „učebnicovém" příkladu přímo z praxe. Vraťme se opět k tvorbě softwaru, který měl za úkol řešit úlohu optimalizace svozu odpadu: Po městě jezdí popelářské vozy, jezdí od jedné adresy k druhé a sváží odpadky. Úkolem je v grafu tohoto svozu nalézt pro všechny vozy ještě lepší cestu (a možná nejlepší) tak, aby se snížily nebo úplně minimalizovaly náklady. Celá síť města je matematicky vyjádřena jako graf se svými uzly a úseky (resp. hranami grafu).

Z analýzy vyplynulo, že uzlem grafu může být buď křižovatka, nebo garáž nebo skládka. Uzel je vlastně to, co ukončuje úsek z každé strany a může být trojího typu, buď je uzel křižovatkou, nebo garáží anebo skládkou. Vozy vyjíždějí z garáží, jezdí přes úseky a křižovatky, odvážejí odpad na skládky a vracejí se do garáží. Jako příklad může sloužit následující obrázek:



obrázek 40 Uzly jsou označeny počátečním písmenem svého typu – Křižovatka, Skládka, Garáž

V příkladu se zaměřme na užitné činnosti nazvané *Přidání nové křižovatky*, *Přidání nové garáže* a *Přidání nové skládky*. U všech třech případů se jedná vlastně o přidání jednoho kolečka na předešlém obrázku, ale každé z těchto přidaných koleček má podle svého typu tak trochu jiné vlastnosti.

Poznámka: Jedná se o typické Use Casy, které nejsou „zlaté", ale podpůrné. Zlatým Use Casem je výpočet optimalizace cest vozů..

Začněme popisovat tyto užitné činnosti, například takto:

Užitná činnost Přidání nové křižovatky

Obsluha určí polohu (X, Y). (například pomocí myši na grafické mapě města). Obsluha zadá název křižovatky (editací), popis křižovatky (editací). Po odsouhlasení se křižovatka uloží.

Užitná činnost přidání nové garáže

Obsluha určí polohu (X, Y). (například pomocí myši na grafické mapě města). Obsluha zadá název garáže (editací), popis garáže (editací) a kapacitu garáže (editací) Po odsouhlasení se garáž uloží.

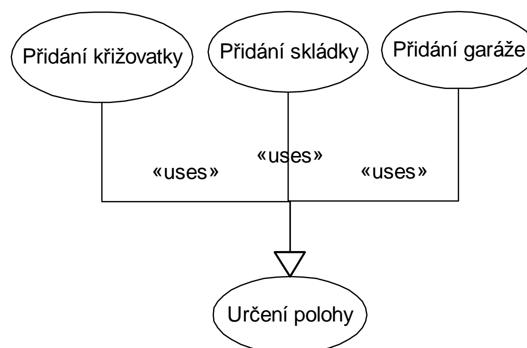
Užitná činnost přidání nové skládky

Obsluha určí polohu (X, Y). (například pomocí myši na grafické mapě města). Obsluha zadá název skládky (editací), Po odsouhlasení se sládka uloží.

Všimněme si úvodní věty opakující se ve všech třech užitných činnostech:

Obsluha určí polohu (X,Z). (například pomocí myši na grafické mapě města)

Zde se jedná se o „signál k posouzení re-use". Protože se tato užitná činnost opakuje, mohli bychom jako první přiblížení zvolit řešení tohoto re-use pomocí zavedení vztahu uses resp. include. Každá z těchto tří užitných činností by měla „odskok" uses resp. include na nějakou sdílenou užitnou činnost. Nazvěme tuto společnou činnost v tomto prvním přiblížení k řešení (které není úplně přesné) jako *Určení polohy*.



obrázek 41 První ne úplně přesné přiblížení k řešení

Podotkněme, že ponechat řešení takto je sice chybou, nikoliv však fatální.

Můžeme však nalézt ještě přesnější a tedy „správnější" pohled: Vraťme se k obrázku ukazující mapku města. Každé kolečko na obrázku (ať se jedná o křižovatku, garáž nebo skládku) je chápáno jako jeden Uzel grafu. Existují tedy nikoliv tři, ale čtyři pojmy: Uzel, Křižovatka, Garáž a Skládka. Mezi

těmito pojmy existuje vztah. Zatímco Křižovatka, Skládka a Garáž stojí jako pojmy„vedle sebe“, mezi Uzlem a ostatními existuje nějaká závislost, kterou vyjadřuje následující věta:

Křižovatka je chápána jako „Uzel a něco navíc speciálního pro Křižovatku“ , …totéž pro Skládku, a totéž pro Garáž.

Pokud se podíváme blíže, co činí Uzel Uzlem (kolečko na obrázku bez ohledu na to, zda je to K, S nebo G), tedy jaké má Uzel vlastnosti, tak Uzel je nositelem polohy (a možná ještě něco dalšího specifického pouze pro Uzel). Tedy jak Křižovatka, tak Skládka, tak Garáž mají Polohu, protože jsou viděny i jako Uzly (jsou viděny jako kolečka na obrázku). Vraťme se k posledním dvěma větám proloženým tiskem a porovnejme je. Větu:

Obsluha určí polohu (X,Z). (Poznámka: například pomocí myši na grafické mapě města)

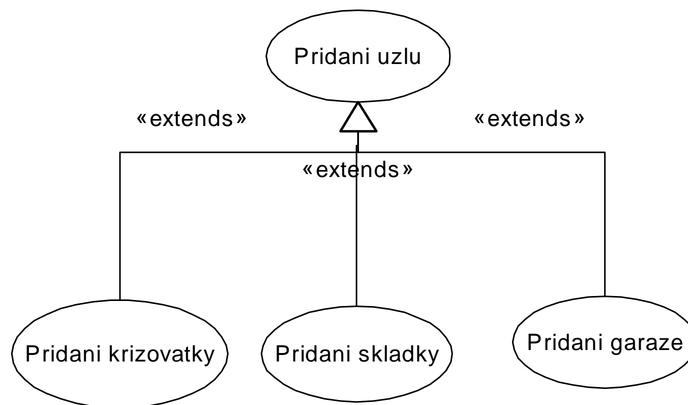
můžeme nyní vidět nikoliv jako *Udání polohy*, ale jako samotné *Přidání uzlu* (bez ohledu na to, zda se jedná o Křižovatku, Garáž nebo Skládku). Zavedeme tedy užitnou činnost *Přidání uzlu* a změníme pohled:

Přidání Křižovatky je chápáno jako Přidání uzlu a něco navíc

Přidání Skládky je chápáno jako Přidání uzlu a něco navíc

Přidání Garáže je chápáno jako Přidání uzlu a něco navíc

Tedy **celé** *Přidání Křižovatky* rozšiřuje neboli specializuje obecnější *Přidání Uzlu*, podobně pro *Přidání Skládky*. Vztah extends je směrem od *Přidání uzlu* k *Přidání křižovatky*, *Garáže* a *Skládky*.



obrázek 42 Přesnější vyjádření re-use v předešlém příkladu

Všimněme si, že je dodržena zásada, že ten prvek, který potřebuje druhý prvek, tak si na něj ukazuje.

Zdůrazněme, že celá užitná činnost *Přidání křižovatky* se musí číst i spolu se všemi užitnými činnostmi, které ona sama extenduje.

Rozdíl mezi extends a uses si nejlépe uvědomíme na uvedeném příkladu při této úvaze: Pokud bychom vztah extends nezavedli a ponechali pouze uses resp. include, tak bychom vlastně otočili myšlenku v tom směru, že Přidání Křižovatky **obsahuje** Přidání Uzlu. Pokud použijeme extends, tak **rozšiřujeme** (specializujeme) Přidání uzlu o Přidání křižovatky. Protože čteme tento vztah od užitné činnosti Přidání křižovatky směrem k Přidání uzlu, přesnější pro pochopení by byl výraz: Přidání křižovatky je **speciálním případem** Přidání uzlu. A to je podstata vztahu extends ve verzi UML 1.2.

Osobně doporučuji nehloubat nad rozdíly extends a uses resp. include, protože to znamená vyhledávat vztah obdobné vztahům generalizace a specializace již v Use Casech a to je mnohdy dost obtížné. Určitě čas a námaha strávená nad tímto rozlišováním nestojí za to… Možná později, až se začnou tvořit class diagramy, je dobré vrátit se k Use Case modelu a předělat odpovídající vztahy, které to vyžadují, z uses na extends.

Použití extends pro rozšíření volitelné činnosti, extension point

Vztah extends se v UML 1.2 používá nejenom pro případ uvedený jako specializace, ale také jako možnost přidání činnosti, která je volitelná a rozšiřuje tak volitelně původní činnost. Tato místa extenze, kde dochází k rozšíření činnosti, lze označit jako tzv. extension point a přiřadit je k rozšiřované činnosti.

Rozdíl mezi uses a extends v tomto pojetí (odpovídá nyní UML 1.2) je následující:

- uses určuje povinně „obsažení“ činnosti jedné v druhé
- extends buď „rozšiřuje“ činnost o volitelnou činnost anebo specializuje činnost

Připomeňme, že takto byly interakce zavedeny v UML 1.2.

Interakce mezi Use Case podle UML 1.2 a UML 1.3

Nejprve kde se nyní nacházíme ve vysvětlení: Jsme v pozici UML 1.2, známe dvě interakce uses a extends mezi se Use Casy a víme, kdy je použít. První vyjadřuje „obsahuje“, druhý vyjadřuje „rozšiřuje“ a to buď specializací anebo volitelnou činností.

Bohužel přechod od verze UML 1.2 k verze UML 1.3 zde není pouze kosmetickou úpravou, ale je poněkud drastický, i když na první pohled se to tak nemusí jevit.

Důležité je, že ve verzi UML 1.2 jsou obě interakce uses a extends zavedeny (tak trochu překvapivě) jako vztah *Generalization*, tedy patří do vztahu chápaného jako vztah gen - spec. Vztah *Generalization* má v UML 1.2 své stereotypy, které jej rozdělují do kategorií a těmito stereotypy jsou (abecedně):

«extends»
«inherits»
«private»
«subclass»
«subtype»
«uses»

Všimněme si, že tímto se stereotyp „uses“ a „extends“ dostal do „stejné rodiny“, jako je třeba „inherits“ používaný jako stereotyp v generalizaci mezi třídami. Znamená to, že vztah mezi Use Casy je zde v UML 1.2 při těchto interakcích vždy chápán jako generalizace, pouze volbou stereotypu rozlišujeme, o jaký typ generalizace se jedná. U „extends“ použitém v UML 1.2 by se v určitých případech tato pozice dala pochopit (viz předešlý příklad s popelářskými vozy), ale „uses“ použitém jako „obsahuje“ resp. „extends“ použitém jako „volitelná činnost“ určitě nikoliv.

Přechod od UML 1.2 k 1.3 znamenal právě opravu této podivné konstrukce s generalizací následujícím způsobem:

1. Obě dvě interakce byly vyjmuty z rodiny generalizací a byly zavedeny jako dědicové jiného typu interakce: jako dědicové elementu *Relationship*. První z nich „uses“ navíc dostala jiné jméno „include“. Znamená to, že existují dvě interakce jako asociace mezi Use Casy podobně jako asociace mezi třídami. Je zřejmé, že tato konstrukce lépe odpovídá pozici a užití těchto

interakcí, nyní se chápou ve stejné pozici, jako je například asociace mezi třídami (mají své konce asociací apod.)

2. Samotné Use Casy jsou ve verzi UML 1.3 zavedeny jako tzv. *Generalizable element*, tj. mezi Use Casy lze „zvlášť“ ještě zavést vztah generalizace a specializace.

V konečném důsledku to znamená, že v Use Case modelu podle UML 1.3 můžete zavést celkem tři interakce mezi Use Casy:

- include, vyjadřuje vztah „obsahuje“
- extends, vyjadřuje vztah „rozšiřuje“ spolu s extension points, avšak nikoliv specializací (volitelná činnost)
- generalization, vyjadřuje generalizaci a specializaci mezi Use Casy (vše, co je v obecnějším, se dědí)

Přitom první dva body jsou důsledkem zavedení nových typů relace mezi Use Casy a třetí je důsledkem vlastnosti Use Casu „možnost býti poděděn“.

Podle UML 1.3 by se tedy příklad s popelářskými vozy řešil pomocí nikoliv pomocí extends, ale pomocí vztahu generalizace a specializace.

V UML 1.2 by se musela zvolit interakce „generalizace“ se stereotypem extends.

Některá důležitá doporučení pro psaní Use Casů pro správný přechod na OOP

Základem psaní popisu Use Case jsou pojmy problémové domény (Faktura, Objednávka atd.), se kterými se „něco odehrává“. Podobně jako je Use Case model celý sám o sobě úplný, tak jednotlivé pojmy, které používáme, musí být také úplné a tím také přesné. Představme si, že v rámci Use Casu se zakládá nový výrobek, který je daného typu výrobku. Porovnejte tyto zápisy, odlišné části jsou tučně:

*„Obsluze se zobrazí seznam typů výrobků. Obsluha vybere typ výrobku a ten se dosadí do nového výrobku. Obsluze se zobrazí **seznam barev**. Obsluha vybere jednu barvu a ta se dosadí do nového výrobku…“*

anebo

*„Obsluze se zobrazí seznam typů výrobků. Obsluha vybere typ výrobku a ten se dosadí do nového výrobku. Obsluze se zobrazí **seznam barev povolených pro vybraný typ výrobku**. Obsluha vybere jednu barvu a ta se dosadí do nového výrobku…“*

a do třetice:

*„Obsluze se zobrazí seznam **všech** typů výrobků. Obsluha vybere typ výrobku a ten se dosadí do nového výrobku. Obsluze se zobrazí **seznam všech barev**. Obsluha vybere jednu barvu a ta se dosadí do nového výrobku…“*

Všimněte si, že texty jsou zdánlivě takřka stejné a liší se jenom v „maličkostech“, které jsou však podstatné:

V prvním textu není specifikováno, o jaký seznam barev, ze kterého se vybírá, se vlastně jedná.

V druhém textu je výslovně uvedeno, že poté, co je typ výrobku vybrán, existuje pro něj seznam barev, ze kterých lze vybrat.

Ve třetím textu se popis analyticky od druhého liší podstatně – jedná se o jiný Use Case ve smyslu, že neexistuje žádné omezení barev versus typ výrobku.

Je zřejmé, že tato „hra se slovíčky“ musí být v analýze velmi přesná a zde se takříkajíc „láme chleba“ ohledně toho, jak bude systém složitý. Druhý a třetí zápis jsou totiž diametrálně odlišnými. Pokud totiž

přijmeme analytický model vedoucí ke vztahu mezi typem výrobku a barvou výrobku, tak vlastně zavádíme seznam povolených barev pro daný typ výrobku. Tím však nejenom že dojde v tomto Use Case k zúžení seznamu podle typu výrobku (což je pro uživatele výhodou), ale na druhé straně „někde" musí existovat Use Case, který bude do systému obsluhovat administraci pro typy výrobku odpovídající barvy.

Co se týče první formulace uvedená jako „ani tak ani tak", mohli bychom ji odmítnout jako nepřesnou, protože není de facto vymezeno, zda se jedná o omezený seznam barev anebo všech barev. Avšak pokud přijmeme dohodu, že když nebude řečeno jinak, máme na mysli vždy seznam všech entit, tak bychom mohli tuto formulaci za této podmínky o dohodě přijmout. V každém případě je nutno však na přesnost pojmů velmi důsledně dbát.

Zásada omezeného slovníku popisu Use Casů

Při psaní slohových cvičení, článků, knih, románů atd., se vyžaduje používat bohatou slovní zásobu a pokud možno autor se snaží neopakovat pojmy a používá proto synonyma. Při tvorbě popisů v Use Case modelu dodržujeme zásadu přesně opačnou: Synonyma jsou naopak přísně zakázána a slovník musí být velmi chudý a strohý. Pokud se uvedený pojem v modelu již vyskytuje, musí být použit, jinak porušíme tu nejhlavnější a nejdůležitější zásadu pro možnost použití re-use.

Použití stejných formulací jako vzoru

Druhou zásadou je použití stejných formulací u stejných situací. Existují určité typy scénářů, které se opakují. Pokud na ně narazíme, musíme použít stejných formulací. Nejedná se o nic jiného, než o intuitivní zavedení a použití určitého „vzoru" v popisech Use Casů a o určitý způsob re-use ve slovním vyjadřování.

Příklad: Naplnění asociace výběrem objektu obsluhou ze seznamu zobrazených objektů budeme vždy formulovat stejně a to podle následujícího vzoru

Obsluze se zobrazí seznam „něčeho". Obsluha vybere „něco" a to se dosadí do „něčeho jiného".

Příklad na výběr barvy auta v evidenci aut:

Obsluze se zobrazí seznam barev. Obsluha vybere barvu a ta se dosadí do auta.

Je pravdou, že potom není popis Use Casu žádné „počteníčko". Věty jsou stále na jedno brdo, ale to je účel. Enormně se tak zrychluje tvorba Use Case a vytvoří se určitý návyk slovníku, který vede k rychlejšímu pochopení čtenáře „o čem je v Use Casu řeč".

Základní vzory popisů užitných činností při akcích obsluhy

V souvislosti se vzory popisů Use Casů je třeba uvést, že 90% popisů užitných činností jsou ty, které komunikují s okolím pomocí formulářů, tj. kterou doprovází činnost obsluhy z formulářem a u nich se zase v 90% jedná buď o:

- výběr pojmu ze seznamu a dosazení pojmu do jiného pojmu
- zadání nějakého pojmu
- přidání nebo vymazání pojmu v seznamu

První případ odpovídá naplnění asociace, například vybrání itemu z číselníků a jeho dosazení do entity („očíselníkování" něčeho). Druhý bod odpovídá editaci prvku anebo změně celého pojmu, třetí odpovídá práci se seznamem a jeho itemem.

Jednotlivé vzory popisů pro tyto situace jsou následující:

- pro výběr pojmu ze seznamu a dosazení tohoto pojmu do jiného se použije vzor: *„…Obsluze se zobrazí seznam „něčeho“. Obsluha vybere „něco“ a to se dosadí do „něčeho jiného“.* Doporučuji při zobrazení seznamu uvést v závorce „jaké informace se zobrazí“ případně podle čeho je seznam sortován.

Příklad na dosazení ze seznamu: *Obsluze se zobrazí Seznam firem (IČO, Název, uspořádáno podle IČO nebo Názvu). Obsluha vybere Firmu a ta se dosadí do Nové Faktury.*

- při zadání pojmu se použije jednoduchá formulace: *Obsluha zadá něco.* Můžeme tím vyjádřit jednak editaci atributu ve tvaru jednoduchého pole (můžeme doplnit slůvkem editací), anebo se jedná o změnu údajů celého vnořeného objektu. Pokud se jedná o vnořený objekt, musí následovat v popisu vazba na include pomocí slovního spojení „a dále viz užitná činnost“, protože se jedná o zadání vícero pojmů.

Příklady na zadání pojmů:

…Obsluha zadá rodné číslo Osoby (editací)… je bez odvolání se na jinou užitnou činnost a znamená jednoduchou editaci.

…Obsluha zadá Bydliště osoby, viz činnost Editace adresy. Je s odvoláním se na užitnou činnost a jedná se o celou činnost nikoliv pouze editace jednoho pole. Teprve uvnitř činnosti Editace adresy bude *…Obsluha zadá Ulici (editací)…*

Element „Actor“ v Use Case modelu

Součástí Use Case modelu je také element zvaný **Actor**. Úmyslně jsem tento prvek umístil až na jedno z posledních míst mezi ostatními elementy, protože taková je skutečně jeho pozice.

Jeden model element Actor vyjadřuje jeden prvek okolí systému, který se systémem nějak komunikuje, používá jej apod. **Actor není prvkem systému** a reprezentuje kohokoliv (nebo cokoliv) mimo systém, kdo nějak komunikuje se systémem a je s ním v interakci:

Actor může pouze přijímat nebo dávat do systému informace

V předešlé větě je vyjádřen hlavní význam pojmu actor v modelu. Pokud vyjmenujeme všechny actory, tak vyjmenujeme celé okolí systému, mimo jiné tak vymezíme jeho hranice.

V zavedení elementu Actor v UML se vyskytuje určitý protimluv, rozpor a nutno podotknout, že tento rozpor vede k velkému nedorozumění a ve svém důsledku může vést k velké chybě při chápání Use Case modelu, jak jsem si ověřil v praxi.

Každý model v UML (a nejedná se pouze o Use Case model) je souhrnem elementů, které popisují daný systém. Model popisuje systém a tedy ze zásady nepopisuje „nic mimo systém“. Z tohoto hlediska se actor jeví jako element, který je v takto pojatém čistém modelu prvkem cizím. Actor totiž existuje mimo systém a proto „teoreticky vzato“ by jej čistý model neměl zavádět. Zavedení actora se tak stává v určitém smyslu zavádějící a vede u začínajících pracovníků k chybám. Základní příčinou této chyby je tzv. porušení anonymity klienta systému, k čemuž právě zavádění actorů svádí.

Poznámka: V předešlých skriptech o UML jsem (díky doporučením uváděným v literatuře) přeceňoval ve výkladu význam actora. Praxe ukazuje, že tato chyba může vést k fatálnímu zádrhelu v tvorbě Use Case modelu, čehož jsem byl několikrát svědkem

Existuje jedna zásada, kterou je třeba při vyhledávání actorů dodržet a tím vymezit jejich postavení v informačním systému:

Hlavním posláním actorů je nalezení a neopomenutí žádných reálně existujících užitných činností.

Je vcelku logické, že systém je z hlediska pohledu Use Case modelu beze zbytku určen jeho užitnými činnostmi bez ohledu na to, jaké jsou zvoleny actory. Vyhledávání actorů je tak pouze doplňkovou činností, která vede k nalezení všech užitných činností. Jsou to však užitné činnosti, které tvoří systém a nikoliv actoři!

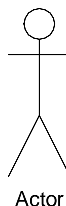
Pokud vytvoříte Use Case model a určíte v něm nepřesně actory a určíte přesně užitné činnosti, což je velmi častý (a mnohdy výhodný praktický případ), potom výsledný systém naprogramovaný podle tohoto modelu bude správný. Pokud dobře určíte actory a nepřesně určíte užitné činnosti, potom systém bude naprogramován chybně.

Actora vyhledáváme hlavně proto, abychom mohli vyjmenovat všechny činnosti, které musí systém obsahovat. Někdy je velmi účelné actory uvádět přesně, protože jím nemusí být pouze živá osoba (obsluha, manažer, ředitel, aj.), ale také jiné systémy (viděné jako externí systémy), datové přípojky apod. V tomto případě vymezení actorů ukazuje, co je předmětem řešení aplikace a co nikoliv, tj. co se považuje za externí prvek. Mnohdy se tato hranice právě u systémů zanedbává a vznikají tak nedorozumění. Například pokud je datová přípojka na port COM vymezena jako actor, znamená to, že tato přípojka není předmětem řešení a s aplikací pouze komunikuje. Při vyhledávání actorů jako externích systémů musíme být velmi opatrní na to, abychom tohoto actora identifikovali na správné abstraktní úrovni.

Příklad: Pro náš systém existuje jiný externí systém, který se jmenuje „Ekonomický systém podniku" a který dodává data do naší aplikace pomocí souboru, který se jmenuje IMPORT.DAT umístěný na určité cestě na disku. Tento soubor obsahuje data v určitém formátu. Otázka zní: Co je actorem, když budeme tvořit Use Case model: Soubor, údaje v něm, cesta ke souboru nebo externí systém?

V žádném případě actorem není uvedený soubor, ani cesta, ale sám externí systém. V popisu činnosti tedy nebude figurovat soubor. Bude existovat užitná činnost Import z „Ekonomického systému podniku" a v ní se uvede, jaké údaje se budou při exportu předávat, ale pozor: pouze slovy problémové domény. Doporučuji v příloze Use Case uvést implementační podrobnosti současného stavu (datové formáty atd.).

Základním visuálním elementem Actora je „panáček", ale používají se i jiné symboly, jako PC pro externí systémy apod.



obrázek 43 Visual element actora

Současně s označením se do modelu také uvádí stručný a výstižný popis actoru.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

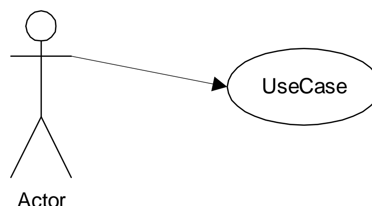
Jak se vyhledává actor

Většinou při konzultaci s uživatelem aplikace (resp. při představě, kdo ji bude užívat) se zjišťuje:

- Co je to za organizaci, která systém bude užívat?
- Jaké role mají zaměstnanci pracující se systémem?
- Kdo má zájem na splnění nějaké funkcionality systému?
- Kdo bude systém administrovat?
- Kdo má prospěch z používání systému?
- Kdo přistupuje k systému aby do ní vložil informaci, změnil informaci nebo vymazal informaci?
- Jaké existují zdroje dat z okolí systému?
- Jací existují spotřebitelé dat z okolí systému?

Element modelu interakce „komunikace Actor - Use Case“

Jako poslední model element si uvedeme nutnou interakci mezi Use Casem a Actorem. Visual elementem je spojnice případně se šipkou pro vyznačení směru:



obrázek 44 Interakce mezi Use Casem a okolím, v tomto případě jednosměrná

Zavedení této interakce umožňuje znázornit, se kterou užitnou činností daný actor interaguje.

Příklady z praxe na chyby při zavádění actorů

Příklad na kontext actora v Use Casu a zbytečné diskuse

Při analýze jednoho bankovního informačního systému se řešil Use Case „Založení účtu klientovi na přepážce“. Na první pohled jednoduchá otázka „Kdo je vlastně actorem…“ dostala mezi členy týmu zajímavé rozměry. Rozvinula se vášnivá diskuse:

Jedni tvrdili že actorem je přece klient, protože řeč je o klientovi. Druzí nesouhlasili a tvrdili, že actorem je obsluha na přepážce, protože je to ona, která komunikuje se systémem a klient stojí až kdesi vzadu za přepážkou: „Přece mi nechcete tvrdit, že klient sedí u počítače v bance!“. A paradoxní na tom všem je, že tato diskuse trvala více než dva dny…

Teď, s odstupem času již vím, v čem byla ta největší chyba a asi tušíte správně: Hlavní chybou tohoto problému bylo to, že se vůbec dopustilo, aby tato diskuse trvala tak dlouho. Správně jí mělo být

vyhrazeno 5 minut a možná jen 5 sekund. Ono je to v podstatě jedno, zda zvolíte řešení A nebo B, i když samozřejmě čistější je řešení, kdy actorem je v tomto případě klient. Klienta bychom zvolil z toho důvodu,že účet se zakládá v kontextu klienta. To, že mu v tom pomáhá obsluha na přepážce, je implementační záležitost logistiky daného problému a z hlediska abstrakce analýzy je tento detail nepodstatným. Pokud bychom totiž danou úvahu „kdo vlastně reálně komunikuje se systémem" rozvinuli v tomto směru ad absurdum, tak bychom mohli dospět k závěru, že reálným actorem při zakládání účtu je „device keyboard". Správně actorem je ten prvek okolí, v jehož kontextu se užitná činnost provádí. Například pokud bychom uvažovali nad užitnou činností Přihlášení obsluhy, tak je jasné, že v tom případě zvolíme jako actora Obsluhu na přepážce.

V předešlém příkladu je z hlediska modelování nejdůležitější, že jsme objevili Use Case „Založení účtu klientovi" a kdo je actorem je v tomto případě vedlejší. V dalším vývoji projektu, při tvorbě dalších modelů (class model apod.) a nakonec při kódování má význam jen a pouze užitná činnost a nikoliv actor. Ten se v další práci nijak neprojeví, protože pro systém je actor vždy anonymním klientem systému! Proto by hlavní analytik měl utnout tyto diskuse a sám rozhodnout (…když je to jedno, tak má dobrou a výhodnou příležitost uplatnit svou autoritu „bez rizika následků")

Anonymita klienta systému, přístupová práva a kontext actora

V praxi existuje přímo ukázkový příklad na chybu nedodržení anonymity klienta systému. Tato chyba, jak si ukážeme, má přímo fatální důsledky na tvorbu modelu, tedy stručně řečeno vás tato chyba spolehlivě zavede do slepé uličky a tvorba modelu se „zadrhne". Díky tomuto příkladu a tedy na této chybě si ještě lépe vysvětlíme pojem anonymity klienta.

V jedné firmě se vyvíjel rozsáhlý systém, který vyžadoval (stejně jako většina jiných podobných systémů) řešit také přístupová práva. Jedná se o všeobecně známou agendu: Existují různí uživatelé, kteří patří do různých rolí a podle příslušností do daných rolí se jim buď nějaká činnost povoluje nebo zakazuje, resp. jinak modifikuje (editace, čtení něčeho apod.).

V této firmě se učinila ta chyba, že se začali zavádět actoři, kteří „pochopitelně" souviseli s danými rolemi. Úvaha je jednoduchá: Přece ten, kdo užívá systém, je actorem a jeho role jako uživatele souvisí s jeho právy. Například vedoucí účetní, mzdová účetní, ředitel, atd. jsou actoři a současně rolemi v přístupových právech. Zní to sice logicky, ale je to chybná úvaha! Přidání dalšího uživatele systému s dalšími právy tímto vedlo paradoxně k přidání actora. A ejhle – má se snad měnit Use Case model s dalšími právy a dalšími rolemi práv? To je nesmysl...

Pro modelování nejsou důležitá nějaká práva a role, ale seznam užitných činností, to mějme na paměti. Celkový seznam užitných činností, který potřebujeme jako konečný výsledek své práce, vzniká vyhledáváním užitných činností a to různými pomocnými postupy. Mezi tyto pomocné postupy patří i vyhledávání actorů.

Uveďme ilustrativní příklad osvětlující, jak se vyhledávají užitné činnosti pomocí actorů. Někdo z analytiků si vzpomene: A co vedoucí v podniku, nebude on potřebovat sestavy? Odpověď analytiků zjištěna jako „ano", tedy odhalili jsme díky actorovi, že bude existovat alespoň jedna nebo více užitných činností související s výstupem sestav pro vedoucího. Ale tím „role actora končí"…

Co se týče přístupových práv, na začátku (resp. v jiných bodech) se určí, zda přihlášený uživatel má právo na pokračování a pokud ne, tak „má smůlu". Ale toto určení přístupových práv nesouvisí s nalézáním actorů! Výsledkem chybného postupu, kdy nakonec role splývaly s actory, se dospělo až do „mrtvého bodu", kdy přidání role znamenalo zásah do analytického modelu s přidáním actora. Navíc se actoři začali mezi sebou „bít" o pozice vůči užitným činnostem. Tam se samozřejmě s modelováním skončilo, protože to vše je nesmysl…

Problém anonymity actora spočívá v tom, že sama funkcionalita systému se de facto nestará o to, kdo konkrétně na druhé straně „za zrcadlem", tedy v okolí systému, sedí za klávesnicí. Důležité je nalézt potřebné užitné činnosti a tedy kontexty použití systému. K tomu jedinému, tj. k nalezení kontextů použití systému, slouží nalézání actorů. Následně se naleznou užitné činnosti a to je to, co hledáme.

Uvedený problém ještě více vynikne v tzv. procesu autentikace, Jedná se o proces, který má toto poslání: „Ověřuje se, že ten, kdo se za někoho vydává, je skutečně tím, za koho se vydává“. Všimněme si trochu kostrbaté formulace v předešlé větě, která je však záměrem. Nemůžeme prostě napsat, že ten na druhé straně je opravdu oním na druhé straně. Musí se nejprve za někoho vydávat, třeba zadat uživatele a heslo, a teprve potom se toto ověřuje. Samo ověřování je užitnou činností jako každá jiná. Mnohdy nemusí být autentikace žádnou „legrací“, například na Internetu se jedná o velmi složitý (a oboustranný) proces.

Anonymita klienta spočívá v tom, že musí existovat proces (užitná činnost) uvnitř systému, která provede autentikaci, například pomocí hesla, vnějším zařízením apod. Ale po tomto ověření není klient stoprocentně tím, za koho se vydává, ale vždy anonymním klientem, který se autentikoval a „dopadlo to dobře“.

*Poznámka: V souvislosti s přístupovými právy vás upozorním na jeden připravovaný projekt naší firmy. Jedná se o agendu Přístupových práv napsanou pomocí OOP, UML a COM a vyjde v edici **Open Model & Open Source.** V této edici budou vycházet otevřené aplikace spolu s modelem v UML a otevřeným kódem. Nejenom, že můžete tuto agendu zapojit do systému, ale také ji celou do detailů prostudovat a to od analýzy v UML až po kód. Sledujte stránky http://www.objects.cz. Tuto agendu jako soustavu modelů v UML a kódu ve Visual Basicu a C++ připravujeme s vydáním na první polovinu roku 2001.*

Použití Use Case modelu v procesu řízení vývoje projektu

Zvláštní vlastnosti Use Case modelu oproti ostatním modelům UML

Úplnost modelu ve dvou rovinách

Use Case model patří k nejdůležitějším modelům UML se zvláštním postavením. Jeho zvláštnost spočívá v tom, že pokud máme k dispozici Use Case model systému, tak vlastně držíme v ruce celý a úplný abstraktní popis systému.

Jak bylo řečeno, vyžaduje se, aby Use Case model byl úplný. Vlastnost „úplnosti“ je zde chápána ve dvou rovinách:

- model je úplný, protože v něm ***nechybí žádná z užitných činností*** (jinak je model chápán jako chybný model)
- model na rozdíl od jiných některých modelů popisuje systém tak, že kdokoliv, kdo jej vlastní, je schopen ***pouze na základě tohoto modelu*** navrhovat další modely a nakonec systém naprogramovat

Podotkněme, že ne každý model v UML má tuto vlastnost úplnosti. Existují pouze dva, po kterých je to striktně vyžadováno a které tuto vlastnost mají: Use Case model a Class model.

Co tyto dvě roviny úplnosti vlastně znamenají prakticky? Je zřejmé, že Use Case model musí obsahovat všechny Use Casy, tj. je úplný podle prvního požadavku. Také víme, jak se toho druhu úplnosti dosahuje: metodou hierarchického rozkladu a hledáním actorů.

Druhá vlastnost úplnosti znamená, že Use Case model je natolik úplným a výstižným popisem systému, že v konečném důsledku z něj lze vytvořit dalším modelováním a kódováním naprogramovaný systém. V Use Case modelu je totiž celý systém uschován jako jeho první úplná abstraktní extrakce.Systém už de facto existuje ve své podobě Use Case modelu a „zbývá jej jenom zrealizovat a naprogramovat“.

Ne každý model se vyznačuje touto „axiomatickou“ vlastností, že se od něj dá systém v konečném důsledku naprogramovat. Například stavový model nemá tuto vlastnost: Pokud byste dostali do rukou

stavový model, dostali byste sice dost výmluvnou informaci, ale těžko byste podle něj tvořili další modely a kód. Stavový diagram se používá jako vodítko pro vysvětlení složitých situací v životě objektů. Navíc je technicky obtížné a tedy není účelné pomocí stavového diagramu popsat všechny stavy všech objektů v systému.

Srozumitelnost modelu

Kromě toho, že je Use Case model úplný a kromě toho, že podle něj lze naprogramovat celý systém, tak navíc je ještě dobře srozumitelný pro všechny účastníky projektu. Protože model musí být bezpodmínečně napsán pouze v pojmech problémové domény (viz předešlé kapitoly), rozumějí mu nejen vývojáři, ale je srozumitelný i pro uživatele, který už nemusí například rozumět Class modelu. Věta:

Obsluha vybere ze seznamu Druhu zboží Druh zboží a dosadí jej do editovaného Zboží

bude uživateli velmi dobře srozumitelná a může ji buď zamítnout anebo odsouhlasit. Díky těmto vlastnostem se Use Case model dostává do zvláštní výjimečné pozice oproti ostatním modelům.

Rychlost tvorby modelu

V porovnání s jinými modely se Use Case tvoří až překvapivě rychle. Největší „zádrhel" při psaní Use Case modelu není ani tak v hledání formulací, tj. v hledání toho „jak to napsat", ale v hledání toho „co se má vlastně napsat". Jinak řečeno, a to si sami můžete ověřit praxí, pro psaní Use Case modelu jsou největší brzdou nedostatečné informace v analýze a jejich doplňování

Příklad: Při jedné konzultaci padl ze strany pracovníků SW firmy návrh na dobu trvání konzultace v délce 3 dny a měl se přitom projednávat vznik a tvorba Use Case modelu. Upozornil jsem, že to je na první konzultaci o Use Case příliš dlouhá doba a že tvorba Use Case modelu má spíše charakter rychlých koleček konzultací s uživatelem, než že by se jednalo o jedno dlouhé kolečko. Avšak snaha byla „udělat toho co nejvíc" a pracovníci na délce tří dnů trvali. První den jsme během asi 10 hodin probrali dekompozici poměrně rozsáhlého systému a během druhého dne jsem při popisu Use Casů začali narážet na problémy typu: „...tak to ještě nevíme". Třetí den nebylo o čem jednat. Problém je v tom, že Use Case je sice hodně abstraktní v tom, že se nezabývá implementačními podrobnostmi, ale je velmi konkrétní v popisu systému a jsou proto třeba velmi dobré znalosti analytického charakteru.

Existují dva základní okruhy neznalostí při tvorbě Use Case modelu: Jako první jsou neznalosti zásadní, protože to jsou neznalosti z problémové domény jako takové. Prostě „nevíme, co se má vlastně v dané situaci udělat". Musíme samozřejmě tyto neznalosti odstranit například pomocí konzultanta a experta na danou problémovou doménu.

Příklad: Přijímá se zásilka výrobků a před tím přišel seznam avizovaných výrobků, jaké výrobky mají do závodu přijít . Co se stane, když v seznamu výrobků, které byly dodány, chybí výrobek, který byl předběžně avizován? Anebo naopak, co se stane, když výrobek není avizovaný a přijde? Co se stane, když seznam avízovaných výrobků chybí úplně?

Druhý typ neznalosti se netýká ani tak problematiky jako takové, ta je dobře známa, ale jde spíše o otázku, jak by se nejlépe daný informační systém v dané situaci použil a to „s co nejmenší námahou jeho tvorby". Jedná se tedy spíše o hledání kompromisů, co systém bude podporovat a co již ne (co uživatel přijme jako řešení a co již nikoliv). Tyto otázky souvisí s otázkou nákladů a také cenou za projekt, což by si měl zákazník uvědomit.

Příklad: Obsluha zadává nový výrobek do výroby. Vybere typ výrobku a poté barvu tohoto výrobku. Barvy jsou obecně omezeny typem výrobku. Budeme v systému také toto omezení podporovat, tj. bude existovat omezení barva versus typ výrobku, a tedy pro vybraný výrobek se zobrazí k výběru pouze jeho barvy, anebo bude obsluha vybírat ze všech barev a je tedy pouze na ní, aby se nespletla? Pokud totiž přijmeme maximálnější variantu s omezením barvy pro typy výrobku, tak jsme si tímto přidali práci, musíme samozřejmě zavést nové Use Casy, které budou umět administrovat toto přiřazení, tj. k danému typu výrobku přidávat jeho povolenou barvu a odejmout jeho povolenou barvu. To je samozřejmě z hlediska projektového řízení „práce navíc".

K tomu ještě musíme upozornit uživatele, že každé omezení tohoto typu sice snižuje možnou chybu obsluhy, ale na druhé straně zvyšuje „neflexibilitu" v tom smyslu, že když tento seznam povolených barev typu výrobku nebude správně zadán, musí se opravit, jinak obsluha nebude moci tento výrobek dát do výroby. Tento problém více vynikne v tom případě, pokud obsluha zadávající výrobky do výroby nemá přístupová práva k administraci povolených barev k typům výrobků.

V každém případě při dobré znalosti problémové domény a při rychlém řešení situací „co v systému bude a co ne", je rychlost tvorby Use Case modelu překvapivě enormní. Z vlastní zkušenosti mohu potvrdit, že „když je vše jasné", tak spíše bolí prsty od psaní než hlava od přemýšlení. Velká rychlost tvorby je způsobena tím, že se v Use Case modelu nezatěžujeme tím, **jak** se má daná věc realizovat, ale popisujeme danou věc přímo. Popisujeme přímo věc bez implementačních podrobností a nestaráme se o to, zda to „bude chodit anebo ne, proč to nechodí a anebo zda to navrhnout takto nebo takto". To je až předmětem další fáze vývoje zvané jako design.

Příklad jestliže má „obsluha vybrat ze seznamu Druhů zboží", tak se vůbec „nešpekuluje" nad tím, jak se tento výběr konkrétně provede, zda se seznam vejde do paměti anebo ne apod. Podstata je, že obsluha vybere ze seznamu Druhů zboží a tak se to napíše. Díky tomu je popis psán velkou rychlostí, protože se vystihuje samo abstraktní jádro problému. Samozřejmě nepopisujeme implementační podrobnosti,. Ale věci typu „podle čeho je seznam seřazen, jaká jsou omezení apod." apod. spadají do analýzy a tedy se do Use Case musí zahrnout!

Časová náročnost tvorby Use Case modelu

Mohu z praxe uvést následující údaje:

- pro malý systém se Use Case model tvoří řádově dny,
- pro středně velký systém se Use Case model tvoří přibližně 3-4 týdny
- pro velké systémy záleží na rozložení týmu analytiků a jejich efektivním řízení. Pro velmi rozsáhlé systémy může vyhotovení celého Use Casu trvat přibližně 2-4 měsíce, pokud pracuje celý tým analytiků souběžně a dobře pod jednotným a zkušeným vedením

Další zvyšování času s rozsáhlostí systému není relativně příliš markantní, protože délka trvání je poté spíše otázkou kvality řízení rozšířeného týmu analytiků, než problémem rozšíření systému. Pro větší systémy musí být do tvorby zapojeno více pracovníků a pak je to jen otázka jejich efektivního řízení.

Použití Use Case modelu v projektu

Předešlé vlastnosti dává Use Case model oproti ostatním modelům do výrazně jiného postavení. Ostatně na to nás již upozornili již tvůrci UML v doporučení pro řízení projektu, které znělo: *Projekt má být řízen Use Case modelem.* V praxi se mi nejenom že tato skutečnost plně ověřila, ale navíc mohu

potvrdit, že Use Case model se používá v projektu nejenom pro řízení projektu, ale slouží jako dokument pro řešení dalších dost problematických situací.

Zapojení Use Case modelu do projektu velmi výrazně urychlují ty práce v projektu, které se přímo netýkají tvorby kódu jako takového, ale jsou pro projekt anebo pro firmu jako celek nezbytnými. Tuto myšlenku rozvedeme dále, protože má, jak jsem si ověřil v praxi, dalekosáhlé příznivé důsledky.

Použití Use Case modelu pro vývoj

Tento druh použití Use Casu zde uvedeme pro úplnost, protože je to vlastně triviální tvrzení: Kvůli vývoji SW původně Use Case vznikl. Vývojáři mohou rozpracovat další modely v jeho návaznosti a nakonec vytvořit kód.

Použití Use Case modelu pro vedení projektu

Když se podíváme na všechny vlastnosti Use Case modelu uvedené v předešlé kapitole, tak se nepodivíme tomu, že každý správný vedoucí projektu rád získá Use Case model systému projektu, který má vést.

Poznámka: Měl jsem tu příležitost pracovat jako vedoucí projektu a hlavní analytik v jedné osobě a mohu potvrdit, že Use Case model je pro vedoucího projektu nepostradatelným dokumentem. Ten, kdo nepoznal vedení projektu podporované Use Case modelem, tak ten nepoznal ten „správný a pěkný prožitek" z vedení projektu (a většinou zná ten záporný prožitek vedoucí až k žaludečním vředům).

Dokud nebyl hotov Use Case model, měl jsem takříkajíc „neklidné spaní". Jakmile se dokončil (a například zjistilo se, že systém je opravdu oproti očekávání více rozsáhlý), tak jsem si oddechl – rozsáhlost problému byla odhalena a dokonce i určena do detailů.

Oproti tomu řídit projekt bez Use Case modelu znamená řídit něco, o čem nemám ani potuchy. Takovéto řízení je plné tzv. „jobovek" (Jobových zvěstí), které vedoucímu chodí od podřízených pracovníků: A toto je ještě třeba udělat, a toto chybí atd...

Opravdu není nad to, když jako vedoucí vidíme systém před sebou ve své úplnosti, a to dokonce v modelu úplném, čitelném a přehledném.

Jistě znáte ty „správné odhady" pracnosti projektu počítané na člověkoměsíce, člověkoroky apod. Pokud nemáte před sebou Use Case model, tak se jedná o hádání z křišťálové koule. Netvrdím, že se při použití Use Case modelu přesně trefíte do správných čísel, ale určitě váš odhad bude při pohledu na Use Case model alespoň podložen jeho rozsáhlostí.

Bobtnání projektu jako největší mor a jak mu zamezit

Největší nemoc, ba přímo rakovina všech projektů, je „bobtnání projektu". Znáte tu situaci: Projekt jede „na plné obrátky", ale stále přibývají další a další agendy, útvary, a tedy další práce. Neustálé rozšiřování projektu, tedy jeho bobtnání, nakonec vede k tomu, že se v projektu nedaří plnit plánované termíny. Navíc projekt ztrácí své kontury, začíná se doslova „rozplizávat" a z původně na začátku dobře plánovaného projektu se stává snůška organizačních opatření, protože další a další požadavky na systém rostou jako houby po dešti.

Každý vedoucí projektu by se měl procesu bobtnání projektu vyvarovat. Nutno podotknout, že se ho také mnohdy obává, ale bohužel až jeho důsledků: Pro vedoucího projektu se efekt bobtnání projektu jeví jako „časovaná bomba plná zmíněných Jobových zvěstí". V určitých intervalech za ním přijde podřízený a řekne mu, že je třeba ještě udělat „neočekávaně" to nebo ono, protože „to jinak fungovat nebude" a samozřejmě je třeba to dořešit. Pokud vedoucí projektu není masochistou, tak takovýto styl práce mu spíše způsobí žaludeční vředy než požitek z vedení projektu.

Jedním z hlavních úkolů vedoucího projektu je ve spolupráci s hlavním analytikem zamezit a ještě lépe úplně zabránit procesu bobtnání projektu. Jinak se projekt dostane do synergeticky nestabilního stavu a nakonec „exploduje“ přesně podle závěrů teorie katastrof. Hlavní důvody bobtnání projektu jsou uvedeny v následujících kapitolách.

Bobtnání projektu díky chybné analýze systému

Nejčastějším důvodem nepříjemného bobtnání projektu je neúplný anebo vůbec chybějící Use Case model. Pokud vedoucí projektu nemá žádný Use Case model k dispozici, tak samozřejmě může akorát tak hádat, co jej ještě v projektu čeká. To je důvod, proč vedoucí projektu, který jednou „zažije“ projekty s použitím Use Case modelu, jej začne vždy vyžadovat i v dalších projektech.

Musím však upozornit na jinou a to velmi častou chybu neúplného Use Case modelu, která souvisejí s určitým pohledem vedoucích pracovníků na projekt. Velmi často dochází u těchto pracovníků k zúžení pohledu pouze na „zlaté Use Casy“ projektu. Většinou vedoucí pracovník pouze v těchto zlatých Use Casech vidí funkcionalitu daného systému (a ostatní je pro něj nezajímavý „balast“). Co si v té chvíli neuvědomuje, je to, že systém potřebuje ještě dalších 99% tohoto balastu, tj. podpůrných Use Casů, které musí zabezpečit funkcionalitu oněch zlatých Use Casů. Navíc tyto vedlejší Use Casy mohou být mnohem složitější než oněch několik pár „zlatých Use Casů“, protože uvedené zlaté Use Casy jsou „zlaté“ nikoliv díky složitosti, ale díky obchodu. Pokud není hotový ***celý*** Use Case model, tak potom tomuto vedoucímu těžko vyvracíte jeho názor, že jak on sám říká: „Však na tom nic není“.

Poznámka: Vedoucí projektu GSM Bankingu by mohl prohlásit: Vždyť na tom nic není. Na jedné straně klient napíše SMS zprávu a ta se v bance provede. Že existuje ještě dalších 150 činností okolo (včetně evidence klientů, telefonů, bezpečnost atd.), tak to je v té chvíli opominuto...

V souvislosti s neúplným Use Casem buďte musím upozornit na jednu záludnost. Buďte opatrní na tvorbu „plánovaných neúplností“ Use Case modelu ve smyslu: Tady je ještě nějaký Use Case nehotov, tj. „tam jsou lvi“, víme o tom a tuto část Use Case modelu doděláme později. Tedy jinak řečeno, máte perfektní přehled o určitých částech systému, ale existují neprobádané oblasti. V tom případě se nenechte ukolébat tím, že budoucí změny se týkají pouze této oblasti, kam jste zatím nevstoupili. Největší záludnost tohoto postupu spočívá v tom, že budete muset zasahovat a rozšiřovat také už hotové a existující Use Casy. Tato neúplnost se totiž netýká pouze neznámé nezpracované oblasti, ale také již hotových oblastí, což může být na první pohled překvapivé. Uvědomme si, že každá užitná činnost potřebuje ke své funkcionalitě pojmy, které zase potřebují ke své vlastní funkcionalitě jiné Use Casy umístěné na druhém konci systému. Rozpracování nehotových Use Casů může proto vést k dalšímu rozšíření Use Casů v již hotových oblastech.

Příklad: Existuje oblast Use Casů „Neznámá“, kterou zpracujeme a po její zpracování se díky ní rozroste užitná činnost „Administrace“, kterou jsme jinak považovali za uzavřenou. Nová oblast totiž potřebuje ke své činnosti další užitné činnosti v Administraci.

Je nyní jasné, že první zásadou vedoucího projektu a následně vedoucího analytika je **vytvořit Use case model pokud možno hned na začátku projektu**. Projekt lze tedy rozdělit na dvě hlavní fáze: Projekt před dokončením Use Case modelu a projekt po dokončení Use Case modelu. Tento krok rozdělení na dvě fáze považuji za nezbytný a jeho nedodržení má pro řízení projektu fatální důsledky.

Příklad: Po rozpracování Use Casu a jeho posouzení se může dokonce stát, že se projekt tak říkajíc „odhouká“ a není nic lepšího, než když se tak stane v této fázi, kdy na projektu pracovalo několik málo analytiků po nepříliš dlouhou dobu. Jistě dovedeme pochopit (a někteří z nás to zažili na vlastní kůži), k jakým ztrátám pro firmu to vede, když se projekt musí zastavit díky bobtnání projektu někde v půlce cesty, když jsou například některé agendy ztvárněny dokonce i v kódu.

Uvedený postup vyhotovení Use Case modelu hned na začátku způsobí to, že si bobtnání projektu „užijí" pouze analytici tvořící tento model, což je mimochodem jedna z náplní jejich práce, za kterou jsou placeni. Díky vyhotovení Use Case modelu dojde k bobtnání Use Case modelu pouze při jeho tvorbě, což je mimochodem přirozená vlastnost tohoto modelu a jediný možný způsob, jak tento model vyvinout. Tedy ve fázi tvorby Use Case modelu se všechny práce soustřeďují na jeho tvorbu.

Poznámka: Mimochodem tento požadavek na tvorbu Use Case modelu hned na začátku a v plnosti znamená, že tento model se vymyká pravidlu I + I, tj. inkrementální a iterativní vývoj.

V mnoha případech se postupuje tak, že se Use Case model odkládá a „jde se rovnou programovat", z toho důvodu, že „Use Case model nelze napsat, protože ještě přesně nevíme, jak to bude". Tedy Use Case model nebo jeho části se vynechají jenom proto, že při tvorbě modelu nejsou známy analytické informace související s daným informačním systémem. Jedná se o jednu z největších chyb v projektu. Problém se pouze odloží a bohužel se tento postup později vymstí i s úroky. Namísto toho, aby se informace o tom „jak to tam má chodit" v té chvíli získaly, vynechá se řešení Use Case modelu se všemi důsledky. Vždyť jak lze naprogramovat systém, když nevíme takovou podstatnou věc, jakou je „co se vlastně má naprogramovat". Tímto se obejde se problém tvorby Use Case modelu a zdánlivě se práce urychlí (něco se přece jen programovat začne...), avšak tento problém se později vrátí a to ještě v hrozivější podobě.

Jedna praktická rada: Pokud narazíte na nějaké nevyjasněnosti analýzy v Use Case modelu, tak buďte rádi, že jste je odhalili v této fázi a ne až v programování. Zde se musí práce plně soustředit na řešení těchto problémů, zde se nejintenzivněji provádějí konzultace s uživateli, s externími spolupracovníky. Výsledky konzultací se ihned zapisují do Use Case modelu. Až je Use Case model hotov, tak těchto konzultací rapidně ubude.

Příklad: Setkal jsem se v praxi se zajímavou situací, kdy v jedné firmě váhali přijmout moje doporučení vyhotovit celý Use Case model. Zdůvodnění znělo vcelku logicky: „Systém je příliš rozsáhlý, bude nás to stát hodně času a navíc, všichni pracovníci se v problematice dobře orientují a to, jak to má chodit, tedy známe velmi dobře". Nakonec doporučení přijali a Use Case vyhotovili. Tvorba celého modelu Use Case, jinak poměrně dost rozsáhlého systému obřích leasingových společností se všemi doprovodnými agendami, trvalo odhadem asi 6-8 týdnů. Nyní na tento dokument nedají dopustit, protože se stal pro všechny velmi žádaným čtením (není nad to přečíst si, jak to má chodit). Navíc při psaní jednotlivých Use Casů zjistili, že v některých případech jejich znalost toho, jak to má být, nebyla až tak detailní: Use Case model ihned odhalil všechny tyto skryté neznámé detaily.

V konzultacích bývá častou otázkou, zda je technicky vůbec možné vyhotovit Use Case model v celku a to dokonce hned na začátku. Mohu potvrdit, že ano a že jedinou brzdou jeho tvorby je právě řešení nevyjasněností a nikoliv jeho technické ztvárnění. Pokud se podíváte na čas nutný pro tvorbu Use Case modelu (uvedeno například v předešlých kapitolách), tak určitě „námaha" stojí za to. Navíc při tvorbě Use Case modelu programátoři a designéři pracují na jiných projektech. Připomenu, že u velkých projektů se musí do Use Case modelu zapojit další pracovníci než pouze jeden analytik.

Bobtnání projektu díky rostoucím požadavků uživatelů a jak mu zamezit

Z praxe je velmi dobře známé bobtnání projektu díky neustále rostoucím požadavkům uživatele. Projekt po každé konzultaci s uživatelem naroste o další předtím zapomenuté požadavky a tento proces nemá konce kraje.

Navíc tento postup má i svou nepříjemnou obchodní stránku: Někteří uživatelé využívají svého postavení a snaží se , aby softwarové firma provedla na danou zakázku co nejvíce práce „zadarmo" a snaží se vměstnat do dodatečných úprav softwaru další a další funkcionality.

Jediný způsob, jak zamezit tomuto velmi nepříjemnému procesu (přece jen „zákazník je zákazník") spočívá v zavedení Use Case modelu do procesu komunikace s uživatelem. Use Case model se tak po malých úpravách zapojí do „obchodního procesu" a stává se východiskem dohody mezi dodavatelem a odběratelem.

Je zřejmé, že pokud je Use Case model hotov a odsouhlasen zákazníkem (a má svou právní váhu), tak lze jednoduše určit, zda požadavek na systém je oprávněn anebo nikoliv. Proces práce s požadavky zde musí vést ke správnému přístupu k tzv. verzování systému. Požadavky, které nejsou předmětem dané dohody, se dostávají do vyšších verzí.

Bobtnání projektu díky touze po dokonalosti

Z vlastní zkušenosti mohu potvrdit, že následující příčina bobtnání projektu má pro vedoucího projektu z hlediska psychiky nejhorší emotivní dopady (lidově řečeno „člověk by vraždil").

Představte si, že je prosazeno vytvoření Use Case modelu, který se zdárně dokončí. Na něj navazují další modely, uživatel je odsouhlasí, další jeho připomínky putují po dohodě do příští verze atd. Stručně řečeno „opratě projektu jsou pevně v rukou" a najednou zjistíte, že v týmu se systém tvoří tak, že funkcionalita systému nesouhlasí s dohodnutým Use Case modelem. A když pátráte po tom, kde je příčina, zjistíte, že některý z podřízených pracovníků, například programátor, se rozhodl, že zná lepší řešení a v touze po dokonalosti systém tak zvaně „vylepší". Dobře míněná snaha vede k velmi nepříjemnému důsledku: Další modely a možná i kód nesouhlasí s Use Case modelem, a celý projekt se dostává do nestabilního stavu.

Samozřejmě není žádoucí touhu po dokonalosti potlačovat hned od začátku nějakými administrativními zásahy. Je dobře, že členové týmu mají zájem na tom, aby systém „byl lepší". Avšak jakákoliv změna musí proběhnout jako řízený proces, tj. dotyčný pracovník je povinen svůj návrh předat hlavnímu analytikovi a ten rozhodne: Buď přeřadí tento návrh do seznamu požadavků na příští verzi anebo se rozhodne jej zařadit do stávající verze, samozřejmě po poradě s vedoucím projektu. Ale z praxe doporučuji být velice opatrný na rozšiřování funkcionality. Vzpomeňme, že přidání jednoho Use Casu může způsobit zrod několika dalších podpůrných Use Casů. Systém se nerozroste pouze o námi přidané Use Casy, ale sám ještě nabobtná v jiných částech. Proto doporučuji používat metodu „pokud není třeba, neměnit Use Case model dané schválené verze".

Změny v Use Case modelu díky změnám analytické podstaty problému

V některých případech nastává změna i díky tomu, že nějaká skutečnost problémové domény se změní. Například se změní zákony, prováděcí předpisy apod. V tom případě si moc nepomůžeme a je třeba změnit Use Case model a následně další modely (a nikoliv pouze kód!). Ve vztahu k uživateli je důležitý vztah vyplývající z obchodních dohod, zda tyto změny spadají do uzavřené smlouvy anebo nikoliv.

Výhody použití Use Case modelu ve firmě

Use Case model se s výhodou používá v následujících oblastech

Použití Use Case modelu vedoucím projektu pro řízení projektu

O tomto způsobu použití bylo pojednáno v předešlých kapitolách. Uvádíme tuto skutečnost ve výčtu pro úplnost.

Použití Use Case modelu pro tvorbu dodatku ke smlouvám se zákazníkem

Use Case model se může stát velmi dobrým východiskem pro popis specifikaci produktu při sepsání smlouvy se zákazníkem v té části smlouvy, kde je třeba tuto specifikaci popsat. Přece jen je dobré podchytit ve smlouvě, co má dodávaný systém vlastně umět. Pokud tak neučiníme anebo tak učiníme pouze povrchně, sepsaná smlouva stojí více méně na vodě a po určité době se můžete s tímto zákazníkem přít, jak to vlastně bylo ve smlouvě myšleno.

Pochopitelně není řešením „vzít Use Case model tak jak je a přilepit jej ke smlouvě“. Use Case model by měl podléhat určitým pravidlům utajení, protože pokud jej dostane do rukou konkurence, dostane vlastně celý váš systém jako na dlani. Stačí však vzít tento model a pomocí určitých jeho pasáží lze specifikaci produktu vytvořit velmi lehce. Navíc uživatel velmi dobře rozumí slovníku Use Case modelu a tedy nemusíme měnit formulace.

Použití Use Case modelu pro seznámení členů týmu s problémem

Všechny práce na systému, ať už se jedná o vytvoření následných modelů, návrhu, kódování atd., se velmi urychlí díky tak jednoduché skutečnosti, jakou je ta, že se kdokoliv z týmu může seznámit s Use Case modelem a přečíst si „o co vlastně jde“. Mohu potvrdit z vlastní praxe, že dobrý a kvalitní Use Case model se stává v týmu velmi žádaným dokumentem.

Použití Use Case modelu v testování

Ve velmi mnoha firmách se na dotaz „jak testujete“ odpovídá „každý sám po sobě“. To však není testování. Programátor má mít odevzdanou práci sám po sobě prověřenu, o tom by nemělo být pochyb. Tato důsledná kontrola vlastních výtvorů není v žádném případě procesem testování!

Testování je proces již nezávislý na tvůrci a mělo by se konat systematicky podle tzv. testovacích plánů.

V testech se pochopitelně mimo jiné zkoumá bezchybnost fungování systému jak v normálních, tak v tzv. mezních situacích (bezchybnost v mezních situacích se lidově nazývá „blbovzdornost systému“). Testovací plány jsou nezbytné nejenom z toho důvodu, že je třeba výsledek testu někam zapsat, ale také proto, že tester se musí nějak seznámit s tím, co se vlastně má testovat. Pro tvorbu testovacích plánů při zkoumání funkcionality má opět nezastupitelnou roli Use Case model.

Poznámka: Testy existují ve dvou rovinách: testy analytické funkcionality, pro které vytváří testovací plány analytické oddělení a testy technologie (zátěž, kritické body systému, krajní meze naplnění, rychlost apod.), které navrhuje designér.

Použití Use Case modelu v obchodním oddělení

Jistě jako vývojáři znáte ty pracovníky, kteří jsou vůči vývojářům z popisu práce nejvíce otravní: Jsou to pracovníci obchodního oddělení a to zejména, když se blíží nějaký veletrh. Neustále dorážejí otázkami, co ten náš systém vlastně umí, co mají nabízet, jaké má výhody atd.

Pokud má firma k dispozici Use Case model systému, tak se tato frekvence otravnosti obchodníků vůči vývojářům výrazně zmenší. Nejprve jsou odkázáni na tento dokument a teprve poté mohou analytici s obchodníky diskutovat o obchodních materiálech. Protože obchodníci jsou již znalí problematiky, tak tato diskuse se však již zaměří na to, o čem má skutečně být, tj. jaké jsou hlavní výhody, co v materiálech zdůraznit (a co naopak nezdůrazňovat), apod.

Poznámka: Samozřejmě obchodník nevezme Use Case model, nezkopíruje jej a nevydá jej jako obchodní dokument. O určitém stupni utajení Use Case modelu byla již řeč.

Jednoduchá tvorba Use Case modelu bez použití CASE nástroje

Ukážeme možnost tvorby Use Case modelu bez použití žádného CASE nástroje a to velmi jednoduchým způsobem a dokonce, což je zajímavé, pro některá užití je tento způsob výhodnější, než

tvorba tohoto modelu pod CASE nástrojem. Pro tvorbu Use Case modelu v nejjednodušší podobě lze použít přímo editor Word.

Nejprve si vytvořte šablonu:

- Založte novou šablonu – soubor typu `DOT`.
- V této šabloně vytvořte formáty nadpisů od „Nadpis 1“ až po „Nadpis 7“. Snažte se nadpisy od sebe odlišit velikostí písma (viz například nadpisy v tomto dokumentu).
- Můžete navíc vytvořit i číslování nadpisů kapitol, pokud vám vyhovuje v textu vidět kam která podkapitola patří. Word tuto funkcionalitu podporuje.
- Dejte šabloně jméno (například `UseCase.dot` apod.) a uložte soubor mezi šablony

Pokud mát problémy s tvorbou této šablony a vyhovuje vám formát tohoto dokumentu, můžete použít tento dokument pro tvorbu šablony. Udělejte si kopii tohoto dokumentu, v této kopii smažte všechen text a uložte jej jako šablonu s příponou DOT.

Při tvorbě Use Case modelu postupujte takto:

- Zvolte v menu Soubor / Nový (pozor nikoliv kliknutí na button s ikonou nového prázdného dokumentu v Commandbaru)
- Vyberte šablonu pro Use Case model.
- Editujte nový dokument.

Při editaci dokumentu se řídíte těmito zásadami:

- Každá úroveň nadpisu odpovídá úrovni hierarchii kompozice Use Case modelu
- Pokud si zapnete zobrazení dokumentů Zobrazit / Osnova, můžete provádět velmi efektivně úpravy v hierarchii členění Use Casů.
- Pro re-use použijte pouze include a to tak, že provedete křížový odkaz přes nadpisy. Vložení křížového odkazu se provádí volbou Vložit / Křížový odkaz a v comboboxu Typ odkazu zvolte nadpisy. Text v nadřízeném Use Casu potom vypadá takto: „…viz užitná činnost *křížový odkaz“*

Takto zapsaný Use Case model má jednu velkou výhodu: Je velmi dobře čitelný jako „normální kniha“.

Závěrečné poznámky: Use Case model a jeho význam pro firmu

Je zajímavé, že v každé firmě, kde jsem měl možnost spolupracovat při zavádění OOP, UML a komponentní technologie, nakonec vždy zavedli Use Case model a většinou jej používají dodnes. Ať už se rozhodnete pro přechod na objektové technologie nebo ne, zavedení Use Case modelu má vždy příznivé důsledky. Platnost tohoto modelu není omezena pouze na OOP.

Jaké jsou nejčastější chyby při tvorbě Use Case modelu?

- pracovníci vzdají tvorbu modelu pro nedostatek informací z analýzy a přejdou rovnou na tvorbu již známých částí systému jen proto, aby bylo aspoň něco hotového. Je samozřejmé, že z důvodu uklidnění zákazníka můžete zhotovit prototypy již zanalyzovaných částí, ale **nesmíte přerušit anebo dokonce ukončit intenzivní práce na Use Case modelu, pokud není celý hotov**. Neznalost analýzy a neúplnost modelu je velmi nebezpečnou časovanou bombou, která je schopná celý projekt nakonec zbortit.

- analytik jako bývalý programátor není schopen opustit implementační pohled a vyjadřovat se pouze v abstraktních (ale přesných) pojmech. Rada: opravovat jeho Use Case model, tak dlouho, dokud si nezvykne.

Sequence model

(zde v této knize překládáno jako sekvenční model)

Pokud se podíváme na definiční vlastnosti objektu, zjistíme, že jednou z těchto vlastností je právě zaslání zprávy od objektu k objektu. Posláním Sequence modelu je zobrazit posloupnost zaslání zpráv mezi objekty. Existují však různé mechanismy poslání zprávy mezi objekty.

- V 99% případech se v OOP jedná o jednoduché zaslání zprávy mezi objekty realizované v syntaxi daného jazyka tzv. přímým oslovením druhého objektu například tečkovou syntaxí. Jeden objekt v rámci výkonu nějaké metody posílá zprávu objektu `A`, tato zpráva se jmenuje například `UdělejProMneNěco`:

  ```
  A.UdělejProMneNěco
  ```

  Zkráceně se toto zaslání zprávy nazývá zavolání metody (které je však v objektu „uschováno“ až za poslání zprávy). Zpráva může mít výstupní a vstupní parametry. V tom případě se může zapsat poslání zprávy takto:

  ```
  Result = A.UdělejProMneNěco ( InputPar )
  ```

  V některých případech dochází k předání parametrů odkazem, tj. členy v `InputPar` mohou mít povahu vstupně-výstupních parametrů. Je velmi důležité si uvědomit, že parametry zprávy (jak vstupní, tak výstupní) mohou být objektovými referencemi! Tedy objekty si mohou předávat odkazy na objekty, tedy jeví se to jako předání objektů.

  Tato skutečnost souvisí s rozdílem mezi strukturálním a objektovým pojetím a mimo jiné dává odpověď na otázku, proč nemůže být z principu DFD diagram součástí UML a tento diagram postrádá v OOP smysl. Pokud použijete například jako vstupní parametr objektovou referenci, nemá smysl hovořit o toku dat, ale o obecném toku informace (pojmu). Objekt je totiž zapouzdřen a specifikace dat by znamenala přejít na strukturální programování.

- **asynchronní a synchronní zaslání zprávy**: při asynchronním zaslání zprávy objekt posílá zprávu druhému objektu asynchronně a nečeká na ukončení běhu činnosti druhého objektu, což se například zprostředkuje přes nějakého prostředníka – objekt Message Serveru, který zprávu převezme a poté ji předá adresátovi. Posílající objekt tedy nečeká na odpověď, odešle zprávu (podobně jako poštu) například do Message Serveru a ten se jako správný pošťák postará o doručení adresátovi. Všimněte si, že poslání zprávy od objektu k objektu se vlastně rozpadne na sekvenci několika poslání zpráv mezi objekty: od objektu A k Message Serveru, od Message Serveru k objektu B.

Sequence model vyjadřuje sekvence zasílání zpráv mezi objekty a používá k tomu následující syntaxi.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

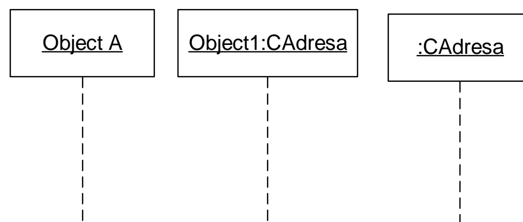
Element „Objekt“ v Sequence modelu

Objekt se v sekvenčním diagramu označuje pomocí obdélníku s názvem objektu uvnitř, přičemž tento název musí být podtržen. Od obdélníku je vedena svislá čára vyznačující časovou osu bez měřítka. Tato časová osa souvisí s posloupnostmi činností objektu čteno odshora dolů, avšak informace o čase je pouze relativní (podává pouze informaci o tom, co probíhá dřív a co později):

V názvu objektu (který musí být podtržen) se používá buď

- jenom název objektu bez třídy
- název objektu s názvem třídy, odkud objekt pochází (pokud jsme si jisti, že tento objekt z této třídy skutečně pocházet bude) , oddělovačem je dvojtečka
- anonymní objekt, kdy nás nezajímá název objektu, ale pouze název třídy, název objektu se v tom případě vynechá a zůstane pouze dvojtečka a název třídy.

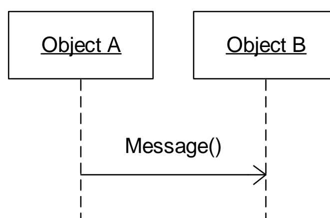
Všechny tři způsoby zobrazuje jako příklad následující obrázek:



obrázek 45 Objekt bez vyznačení třídy, objekt s vyznačenou třídou a anonymní objekt

Element „Zpráva“ („Message“) v Sequence modelu

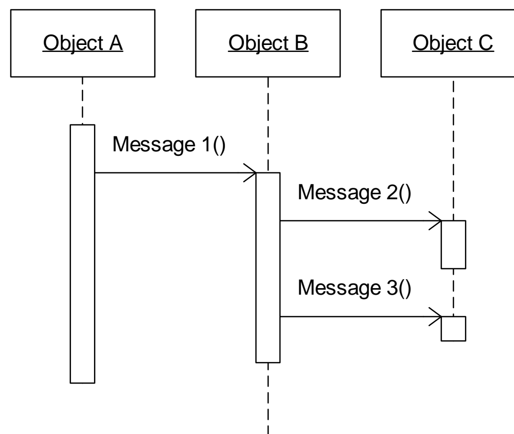
Zpráva od jednoho objektu k druhému se v sekvenčním diagramu znázorňuje pomocí šipky od časové osy jednoho objektu, který zprávu posílá, k objektu, který zprávu přijímá.



obrázek 46 Zaslání zprávy od objektu k objektu

Zasílání několika zpráv v sekvenci se vyznačuje posloupností šipek umístěnými v časové posloupnosti odshora dolů podle své časové posloupnosti. Současně se může vyjádřit také „časová délka běhu činnosti (aktivity objektu)“ objektu pomocí rozšířené časové osy.

V diagramu se pomocí zpráv znázorňuje *sekvence* zpráv dané užitné činnosti. Vzniká tak scénář zpracování (běhu činnosti) jako velmi přehledný diagram sekvence spolupráce objektů



obrázek 47 Scénář sekvence zpráv

Tento diagram čteme tak, že v rámci činnosti objektu A se pošle zpráva objektu B, který tím spouští svou aktivitu a v rámci této aktivity pošle objektu C nejprve zprávu Message 2 a potom zprávu Message 3.

Jednoduchá a asynchronní zpráva

Je důležité rozeznávat dva základní typy zpráv rozlišených podle povahy zpracování při jejich průběhu a to *zprávu jednoduchou* a *zprávu asynchronní*.

Jednoduchá zpráva je takovou zprávou, u které po odeslání zprávy vysílající objekt předá řízení toku činnosti přijímajícímu objektu a sám čeká na zpracování odeslané zprávy. Vysílající objekt pokračuje ve své činnosti až po ukončení činnosti zpracování a ukončení činnosti v objektu přijímajícího zprávu. Tato situace je běžná u jednoho toku (nitě, threadu) zpracování v systému aplikace, kdy se sice předává tok aplikace z jednoho objektu do druhého, ale stále se jedná o jeden a tentýž tok (jedna a ta samá niť).

Visual prvkem synchronní zprávu je šipka:

obrázek 48 Synchronní zpráva

Message call

Zvláštním případem synchronní zprávy je tzv. zpráva „message call", která reprezentuje přímo případ synchronního vyvolání metody volaného objektu. Jedná se o nejčastější případ použití zprávy, kdy v kódu syntaxi jazyka se přímo píše volání metody. Visual prvkem je plná šipka.

Asynchronní zpráva

Pokud vysílající objekt pošle druhému objektu zprávu a pokračuje dále ve své činnosti a tedy **nečeká na průběh činnosti** v přijímajícím objektu, potom se hovoříme o asynchronní zprávě. V tom případě přijímající objekt zpracovává zprávu a souběžně běží proces v objektu, který zprávu vyslal a vždy tak vzniknou alespoň dva paralelní toky zpracování . Při tomto způsobu zaslání zprávy buď dochází opět k synchronizaci anebo jedna z činností objektu končí bez ohledu na průběh činnosti u druhého objektu. Visual elementem model elementu asynchronní zprávy je „poloviční" šipka:

obrázek 49 Asynchronní zpráva

Vztah mezi zprávami a metodami objektů

Pokud má objekt schopnost přijímat určitou zprávu, znamená to, že na tuto zprávu objekt spustí některou z metod. Toto přiřazení zpráva-metoda se nazývá protokol zpráv, seznam zpráv se nazývá interface objektu.

V sekvenčním diagramu se nyní hovořilo o zprávách (šipky mezi objekty). Představme si, že máme u daných objektů určeny třídy, ke kterým objekty patří. Potom můžeme na základě zpráv, které objekt musí umět přijmout, určovat jaké metody musí objekt znát a tedy můžeme budovat samotnou třídu těchto objektů. Znamená to, že u těch objektů, u kterých končí „šipka", tam vznikne nějaká metoda.

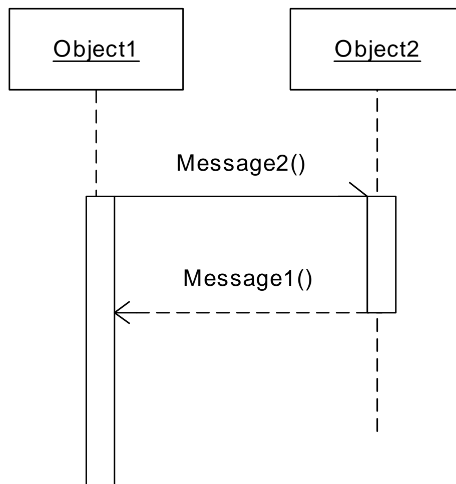
Vstupní a návratové parametry zprávy

Metoda objektu je přiřazena k dané zprávě. Protože obecně v OOP zpráva zasílaná objektu má své parametry jak vstupní, tak výstupní, tak tyto parametry se poté stávají vstupními a výstupními parametry dané metody. Parametry se mohou se vyznačit do sekvenčního diagramu. Vstupní argumenty se umisťují vždy do závorky za název metody.

V syntaxi návratových parametrů existují dva způsoby zápisu, jak se znázorňují vracející se hodnoty.

První z nich je zápis obdobný funkci, kde se vstupní parametry označí podobně jako u funkce resp. procedury. Označí se parametry jako vstupní nebo výstupní. Tento způsob se běžně používá u zpráv typu message call, kdy je zřejmé, že k návratu hodnot dojde v okamžiku ukončení procedury.

U asynchronních zpráv by nemuselo být zřejmé, kdy dojde k návratu hodnot a je proto třeba toto vyznačit. Používá se k tomu syntaxe tzv. návratové zprávy, která „vrací hodnoty". Návrat se tedy označí pomocí tečkované šipky:



obrázek 50 Návratová zpráva, return message

Vztah Sequence modelu a Use Case modelu

Zatímco Use case model jako první model projektu může být použit ve strukturálně pojatém prostředí, sekvenční diagram je diagramem pracujícím výhradně s objekty. Znamená to, že tento diagram již vyhledává kompetence objektů za činnosti v systému, čímž se pochopitelně stává tento diagram pro tvorbu mnohem obtížnější než Use Case model, který se tvoří velmi rychle. V Use Case modelu jsme sice použili pojmy, ale stále ještě se nejedná o objekty a tedy v popisu máme sice omezenou, ale přece jen jakousi volnost.

Většinou se v literatuře u UML dočtete, že je vhodné hned po dokončení Use Case modelu zahájit práce na sekvenčním modelu, protože mezi nimi existuje velmi jasný a důležitý vztah:

Každý Use Case, který je listem hierarchie (tj. každý nejspodnější Use Case vyrovnávající jeden deficit informace + / -) lze chápat v jiném pohledu také jako určitý scénář spolupráce objektů mezi sebou. Lze jej tedy vyjádřit jako jeden nebo jako několik navazujících sekvenčních diagramů.

Postup uváděný v literatuře je následující: Projděme popisy listů v Use Casech a v nich pojmy. Pokud jsou tyto pojmy přesné, stávají se kandidáty na objekty (nikoliv třídy!). Spolupráce mezi objekty – zasílání zpráv je poté vyjádřeno jedním nebo více sekvenčními diagramy. Tedy doporučuje se při tvorbě sekvenčního modelu aplikace vycházet z Use Case modelu, a to tak, že každý jeden Use Case (užitná činnost) je převeden na nějakou sekvencí zasílání zpráv, tj. na jeden nebo více sekvenčních diagramů. Tímto způsobem se systém popíše pomocí sekvenčních diagramů.

Avšak tento postup má jeden háček: Mohu potvrdit, že převést všechny Use Casy na sekvenční diagramy u středních a rozsáhlejších systémů je bezesporu utopií a není to ani technicky možné. Sekvenční diagram je totiž v popisu mnohem detailnější než Use Case model. Problém je v tom, že jeden rychle napsaný odstaveček v Use Case modelu může vést k několika stránkám v sekvenčního diagramu.

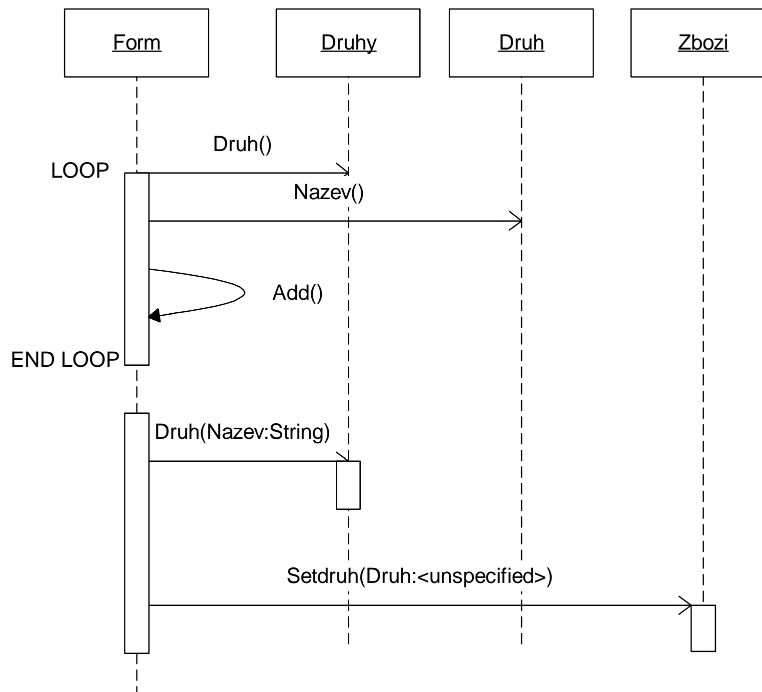
Pro sekvenční diagram je velmi důležitá ta okolnost, že zřetelně popisuje chování aplikace a současně vymezuje kompetence mezi objekty. Vymezení všech sekvenčních diagramů v systému vede vlastně k odhalení všech metod objektů a jejich posloupnosti volání. To, že jeden objekt posílá zprávu druhému objektu znamená, že tyto dva objekty jsou určitě ve vzájemném vztahu. Sekvenční diagramy napomáhají nejen odhalovat chování aplikace a tím metody objektů, ale také napomáhají spolu modelem tříd dobře odhalovat kompetence objektů za jednotlivé činnosti (kdo co bude konat sám a k čemu použije jiný objekt a sám se této činnosti vyhne).

Příklad na překlopení části jednoho Use Casu do sekvenčního diagramu

Představme si, že v Use Case modelu se vyskytuje následující pasáž:

Obsluze se zobrazí seznam Druhů zboží, Obsluha vybere jeden Druh Zboží a ten se dosadí do editovaného Zboží.

Jak bude vypadat analytický návrh v sekvenčním diagramu? Existuje několik možných návrhů, předkládám jako příklad jeden, ten nejjednodušší:



obrázek 51Možný scénář dosazení vybraného druhu zboží do zboží

V tomto scénáři má objekt formuláře Form znázorněny dvě činnosti. První činnost přebírá od seznamu Druhy cyklem všechny Druhy a od každého vezme název pro zobrazení.

Poznámka: přesnější by bylo, že si je přidává některý z prvků obrazovky, například pro Listbox apod.

Druhá činnost vznikla po výběru daného prvku. Název je v tomto případě unikátní pro vyhledání Druhu v seznamu Druhů. Formulář si vyžádá daný druh (který byl vybrán) a ten se dosadí do zboží.

Poznámka:

V diagramu je použit constraint LOOP a END LOOP, který vyznačuje začátek a konec smyčky. Tento constraint je tvořen vlastním textem (je možné použít například OCL)

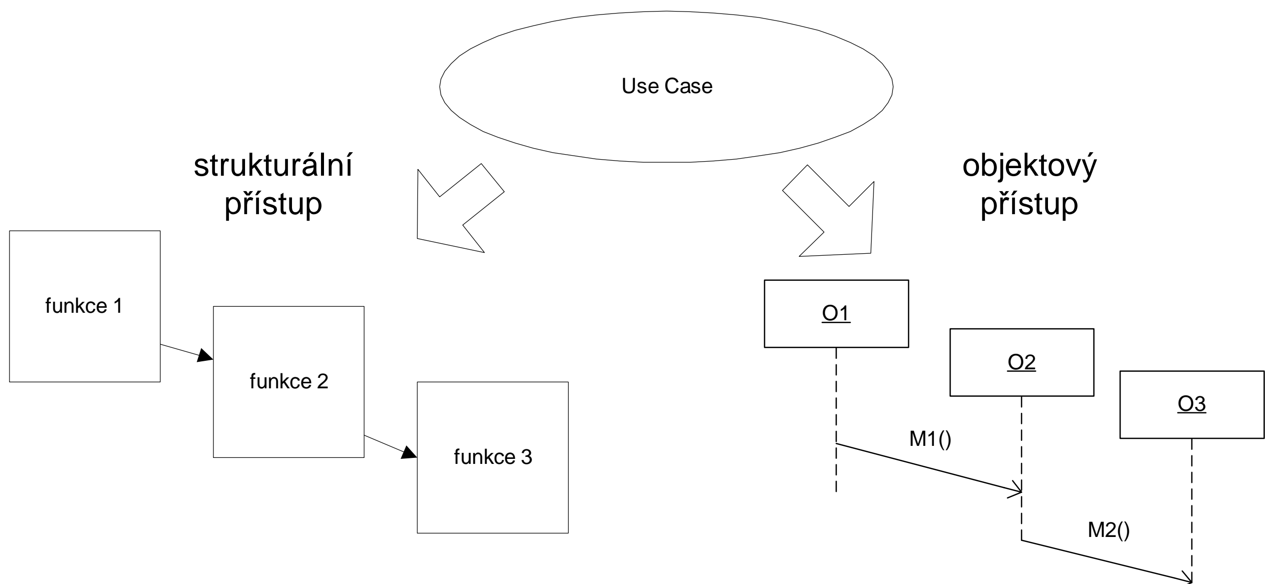
Všimněme si, jak se daný jednoduchý popis rozvinul do poměrně složité sekvence zpráv objektů mezi sebou. Osobně nedoporučuji již v prvních fázích takového překlápění vyhledávat třídy, tato činnost je totiž ještě náročnější.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Sekvenční diagram a strukturální programování

Pokud bychom nepoužili objektový přístup, ale strukturální, nemohli bychom pochopitelně zavést sekvenční diagram. Tento diagram totiž zavádí objekty a rozděluje mezi nimi toky činnosti. Oproti tomu Use Case diagram lze použít i ve strukturálním programování.

Uvedený rozdíl vystihuje následující obrázek



obrázek 52 Přechod od Use Case k funkcím ve strukturálním programování a k objektům v OOP

Levá strana obrázku ukazuje přechod od Use Case modelu k funkcím strukturálního programování . Pravá strana naopak zobrazuje přechod od téhož Use Case modelu k objektům. Všimněte si určité podoby obou stran: Na jedné straně vidíme volání funkcí a na straně druhé posílání zpráv mezi objekty, oba obrázky si jsou velmi podobné. Postupné „volání objektů mechanismem zpráv“ je na rozdíl od volání funkcí doplněno o kompetence objektů, které nahrazují kompetence funkcí.

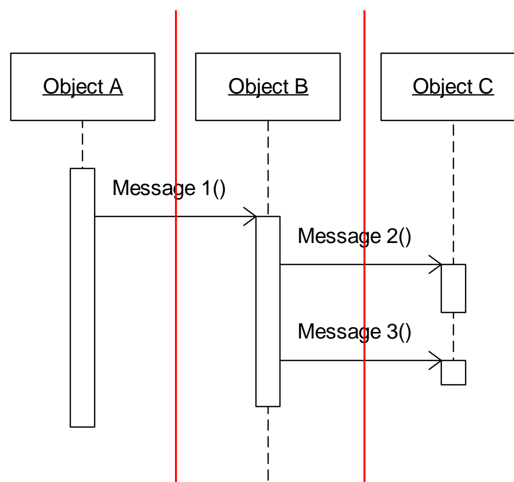
Základní rozdíl v chápání funkcí a objektů zde není znázorněn: Každý objekt O1, O2 a O3 si sebou nese své atributy, které jsou „při něm“ a se kterými může tento objekt pracovat.

Zapouzdření objektů a praktický pohled na sekvenční diagram

Při práci se sekvenčními diagramy je třeba se naučit určitému praktickému pohledu, který odpovídá principu OOP. Sekvenční diagram poskytuje přehlednou posloupnost scénáře zasílání zpráv. Uvedená přehlednost diagramu však může svádět k nesprávnému pohledu na sekvenci zpráv. Scénář zpráv znázorněný v diagramu jakoby se jevil otevřený, protože jej vidíme vcelku.

Je třeba si představit, že každý sekvenční diagram je rozdělený svisle na oblasti dané svislými čarami životních cyklů objektů a vstup do této vrstvy se děje pouze šipkami, tj. zprávami. Co se děje dále za touto vrstvou je pro toto rozhraní vrstev neznámou. Tedy spolupráci vidíme jako rozbalenu, ale tato spolupráce se provádí přes tyto hranice.

Znázorněme si následující scénář, který vidíme na obrázku jako „pěkně a přehledně otevřen“



obrázek 53 Vrstvy mezi objekty

V předešlém diagramu jsou červeně znázorněny informační oblasti jednotlivých objektů. Soustřeďme se na objekt B a na chvíli si odmysleme dvě vnější oblasti, zůstane pouze střed vymezený dvěma červenými čárami. Co nám vymezený informační prostor říká? Objekt „umí přijmout synchronní zprávu Message 1()“. V rámci aktivity (metody) po příjmu této zprávy pošle zprávu objektu C, a poté pokračuje ve zpracování a pošle ještě jednu (jinou) zprávu objektu C. Potom vrací řízení zpět tomu objektu, který mu poslal zprávu Message 1 (protože se jedná o synchronní zprávu). Při tomto pohledu je třeba si pro správné pochopení OOP uvědomit, že objekt B neví nic o existenci objektu A. To je pro něj anonymní klient (viz předešlé kapitoly o anonymitě klienta). Objekt B pouze umí přijmout zprávu, a tento diagram ukazuje jedno toto využití: Jeho „umění přijmout Message 1“ je dosazeno do prostředí objektu A, který toto umění využívá. Z druhé strany objekt B je klientem objektu C (a B je anonymním klientem pro C).

Objekt A, který využívá službu objektu B, vůbec neví, jak je tento příkaz (služba) naplněna. To, že je uvnitř cokoliv provedeno posláním zprávy do C, je pro A neznámou skutečností a vidíme to pouze díky tomuto diagramu. Uvedený pohled „na vrstvy“ znamená právě možnost fázování prací na projektu – diagram se postupně rozšiřuje o další vrstvy. Například analýza končí zprávou Save pro objekt, která se nepopisuje. Jak se provede uložení implementačně, je věcí dalšího postupu v designu. Znamená to, že pokračování tohoto diagramu „doprava“ (rozepsání zpráv Message 2 a Message 3) může být kdykoliv doplněno.

Reálné použití sekvenčních diagramů v projektu

Jak bylo řečeno, v literatuře se doporučuje takový postup pro tvorbu sekvenčního diagramů, který vychází z Use Case modelu. Jednotlivé Use Casy vyjadřující jednu činnost vyrovnání nerovnováhy mezi okolím a systémem se dále zpracují. Slovní popis Use Casu se pomocí objektů a zpráv „překlopí“ do sekvenčního diagramu. Znamená to, že se naléznou z popisu Use Casu odpovídající objekty (není povinné nyní hledat třídy) a odpovídající zprávy objektů, které ve své sekvenci realizují daný popis Use Casu.

Pokud bychom vzali **celý** Use Case model a snažili se jej překlopit do sekvenčního diagramu, bylo by to je sice teoreticky pěkný postup, ale tento postup je prakticky nerealizovatelný.

Poznámka: Osobně jako vedoucí projektu jsem byl svědkem následující situace: V projektu středně velkého systému byl relativně velmi rychle vyhotoven Use Case model. Tento Use Case model byl

dotažen až do všech svých listů a byl zhotoven asi za jeden pracovní týden. Jednomu pracovníkovi bylo dáno za úkol vypracovat z tohoto Use Case modelu sekvenční diagramy. Bohužel jednalo se o služebně nejmladšího pracovníka v týmu, takže mu nějakých pár dnů trvalo, než se odvážil ozvat a říci svému nadřízenému, že je to nereálný úkol, protože není ani v polovině Use Case modelu a přitom tvoří 150. stránku diagramů.

V čem tkví příčina této nemožnosti vytvořit úplný sekvenční diagram? Je velký rozdíl v povaze Use Case modelu a sekvenčního modelu. Zatímco popis UseCase modelu je velmi hutný, sekvenční diagram je již velmi detailní a podrobný (i když i on se ve fázi analýzy týká pouze objektů business logiky). Tento rozdíl se projeví ve velmi rozdílné pracnosti vyhotovení obou modelů. Není opravdu možné použít metodu překlopení Use Case modelu vcelku na úplný sekvenční model.

Přesto použití sekvenčního modelu velmi doporučuji a považuji jej za jeden z těch „nezbytných modelů“. Existují dvě významné oblasti, kde je vhodné sekvenční diagram použít.

Použití sekvenčního diagramu pro specifické a složité sekvence

Opustíme maximalistický požadavek vyhotovit pro každý list Use Case modelu odpovídající sekvenční diagram a popíšeme pouze ty situace, kdy je velmi ***žádoucí jej vytvořit***.

Je nutné vytvořit ty scénáře spolupráce objektů, které jsou velmi specifické a jejich vynechání by mohlo vést k nějakým nedorozuměním. Jedná se o scénáře nepříliš známé, velmi speciální, zvláštní apod.

Příklad: V jednom projektu Internet Bankingu jsem řešili problém přihlášení klienta a autentikace pomocí vnějšího zařízení (zařízení ActiveCard pro vyhotovení podpisu podobné kalkulačce) ve spolupráci s bezpečnostním serverem v pozadí. Bylo třeba vyjádřit oboustrannou autentikaci klienta a serveru navzájem a problém vytvoření bezpečného zašifrovaného kanálu na Internetu pomocí SSL.

Takovýto scénář je dobré popsat pomocí několika následných sekvenčních diagramů, takže nemůže dojít k nějakým kolizím v chápání. Sekvenční diagram velmi názorně ukázal spolupráci objektů při navázání spojení, při oboustranné autentikaci, při vyhotovení MAC podpisu, vytvoření SSL kanálu atd.

Použití sekvenčních diagramů pro zavedení vzorů scénářů

V praxi se velmi osvědčilo použít sekvenční diagram pro vyjádření vzorů spolupráce mezi objekty.

Poznámka: Máme na mysli vzory tak, jak jsou definovány podle E.Gamma, R. Helma, R. Johnsona a J. Vlissidese v knize Design Patterns, ISBN 0-201-63361-2

V tomto případě se daný sekvenční diagram nechápe jako konkrétní diagram spolupráce mezi konkrétními objekty, ale jako vzor (šablona), do které se teprve dosazují konkrétní objekty a konkrétní zprávy. V takto pojatém sekvenčním diagramu se použité názvy objektů v diagramu nechápou jako konkrétní názvy konkrétních objektů, ale jako **role** objektů, do kterých se teprve konkrétní objekty dosadí. Tím se šablona konkrétně implementuje pro daný případ. V některých případech jsou části sekvenčního diagramu napsány pouze symbolicky. Například zpráva mezi objekty jako „properties“ znamená množinu zpráv pro převzetí všech nutných property od objektu. Takovéto symbolické názvy musí být ve vzoru (anebo pro všechny vzory všeobecně) popsány.

Použití sekvenčních modelů jako vzorů považuji velmi důležité hlavně pro řízení projektu. Vede to k velmi silnému sjednocení stylu práce návrhářů a programátorů (a navíc i testerů). Dokonce u

konkrétního kódu, který vytvořil pracovník na základě vzoru sekvenčního modelu, lze zkontrolovat, zda tento kód odpovídá danému vzoru a odhalit tak případné chyby.

Pro jeden a tentýž analytický problém může existovat několik různých scénářů, které daný problém realizují v implementaci a provádí se výběr scénáře pro optimalizaci daného problému. Správný postup je takový, že designér by měl vybrat jeden ze scénářů (vzorů spolupráce) a podle něj navrhnout řešení pro konkrétní objekty. Pokud se narazí na problém optimalizace, musí buď vybrat jiný scénář anebo vytvořit nový vzor a tím se výběr vzorů rozšíří.

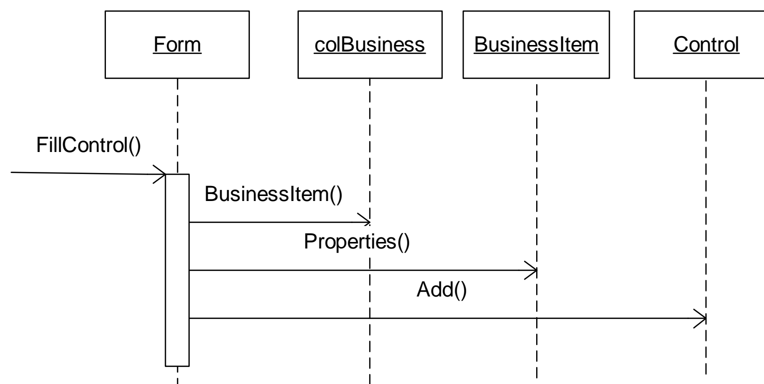
Příklad na použití sekvenčního diagramu jako vzoru

Název vzoru: „Vzor pro zobrazení kolekce objektů do formuláře, klasický scénář"

Popis: V pozadí formuláře Form existuje kolekce business vrstvy colBusiness se svými prvky, BusinessItem. BusinessItem má svoje Properties, které je třeba zobrazit. K zobrazení se použije Control přidávající prvky jako string (resp. jako vlastní prvky, objekty obsahující stringy).

Formulář má svou metodu FillControl, která si cyklem vyžádá od každého BusinessItemu jeho Properties a ty se přidají do Controlu.

Sekvenční diagram:



Poznámka: Scénář není vhodný pro „hluboké" kolekce

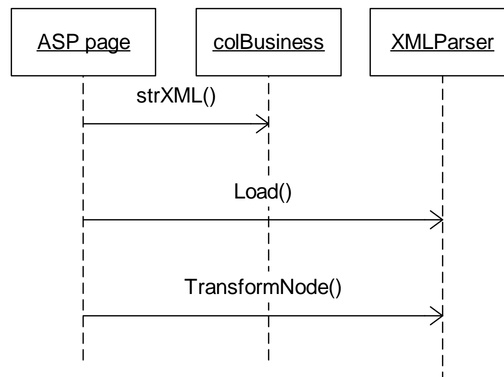
Druhý příklad na sekvenční model jako vzor

Název: Zobrazení kolekce do ASP stránky pomocí XML

Popis: Business kolekce umí vydat ze sebe string jako XML řetězec. Pomocí XSL šablony se převede na požadovaný tvar zobrazení do ASP stránky.

Scénář jako diagram:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



Poznámka: Scénář je vhodný pro ASP stránky.

Je samozřejmé, že by bylo možné zavést další scénáře, například s načítáním pouze omezené množiny itemů pro zobrazení „od do" itemů a toto načítání ovládat událostmi posunů po Controlu apod.

Model tříd, Class model

Jak již bylo řečeno, třída je chápána jako „kopyto" pro objekty, ze kterých objekty vznikají. V čistém OOP prostředí má třída povahu objektu. Z hlediska důležitosti v pořadí modelů patří Class model k těm „nejdůležitějším". Každý OOP jazyk je totiž postaven na pojmu třída a kódování programů se provádí právě pomocí tříd. Pochopitelně ke spolupráce objektů dochází na úrovni instancí, avšak vlastnosti objektů jsou jim dávány „do vínku" vždy a zásadně ve třídách.

Důležitost Class modelu spočívá právě v tom, že **je modelem výchozím pro kódování**. Protože každý objekt patří k nějaké třídě (žádný OOP jazyk „nepracuje bez tříd"), tak kódování programu znamená de facto kódovat třídy.

Class model a stupeň abstrakce ve tvorbě SW

Zavedení tříd zvyšuje míru abstrakce ve tvorbě SW. Je velký rozdíl v našem pohledu na systém, jestli modelujeme na úrovni instancí anebo na úrovni tříd. Jakmile přejdeme z úrovně instancí do tříd, tak jsme zavedli vyšší abstrakci. Vyjadřujeme se v obecnějších pojmech, než u instancí.

Při tvorbě class modelu velmi doporučuji používat jako doplňkový Object diagram resp. konceptuální diagram, který není součástí UML. Problém je v tom, že mnohdy vztahy hůře viditelné v class modelu se okamžitě osvětlí ve vztahu uvedeném jako příklad vztahu instancí.

Musí být Class model úplný?

Je zřejmé a zdá se logické, že Class model by měl být jako výchozí model pro kódování úplný a měl by obsahovat všechny svoje prvky, protože jinak „jak by programátor mohl napsat svůj kód bez neúplných nebo chybějících tříd…"

Ale je třeba podotknout, že představa „vytvoříme nejprve úplný Class model se všemi metodami a podle něj kód" není podobně jako u sekvenčního modelu příliš reálná a prakticky je obtížně schůdná.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Takovýto maximalistický požadavek by vedl k velkým a zbytečným zdržením v projektu, i když většina prvků v Class modelu je bezesporu nezastupitelná. Jak se tedy postupuje pragmaticky při tvorbě Class modelu?

Je opravdu nezbytné zhotovit v Class modelu v analýze a to v každém případě:

- zavést všechny třídy
- zavést všechny vztahy mezi třídami
- zavést všechny atributy
- zavést některé důležité metody v business vrstvě

Poznámka: Je pochopitelné, že nic jiného než Class model v analytické vrstvě nevyjádří tu skutečnost, že Osoba má atributy Rodné číslo, Jméno a Příjmení.

Avšak v reálné tvorbě SW v konkrétním projektu se mnohé metody a atributy hlavně ve fázi designu tvoří až ve vývojovém prostředí, kde se kóduje a pokud třeba, tak se zpětně se zde navržené prvky přenášejí zpět do Class modelu metodou reverse-engineering.

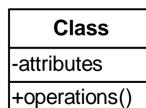
Příklad: Jako poznatek potvrzující a vysvětlující tuto „pragmatickou praxi" uvedu, že se těžko v někde setkáte s tím, že by Class model obsahoval všechny třídy všech navržených formulářů, protože jejich návrh podléhá jednoduchým přímým pravidlům tvorby ve vývojovém prostředí (například ve VB 6.0).

Je tedy otázkou, které prvky class modelu můžeme vynechat? Jsou to ty, které jsou odvoditelné z jiných informací, například ze vzorů a stereotypů. Například nemusíme u kolekcí daného typu znázorňovat metodu Add, pokud bude někde v modelu uvedeno, že ji bude mít každá kolekce tohoto typu apod.

Model „element" Class v UML

Visuálním elementem třídy, tedy class, je symbol obdélník s názvem třídy, přičemž tento název není podtržen. Název třídy jednoznačně třídu identifikuje a je unikátní.

Další částí visual elementu Class mohou být vyznačeny názvy atributů a názvy metod budoucích instancí, přičemž tyto názvy jsou jedinečné v rámci dané třídy.



Viditelnost atributů a operací, operace a metody třídy

Ke každému atributu resp. operaci lze přiřadit tzv. vlastnost viditelnosti, která je podle UML buď

• public – libovolný vnější Model Element vidí tento Model Element (atribut resp. operaci).

• protected - libovolný dědic Model Elementu vidí tento Model Element.

• private – pouze uvnitř Model Element, v jeho částech je viditelný

Na diagramu se značí private jako -, public jako +, protected jako #

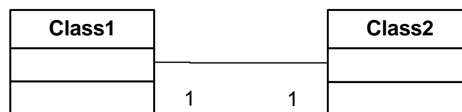
Pokud se jedná o atribut třídy resp. metodu třídy, název atribut anebo této metody je podtržen.

Model element „asociace“

Asociace

Vztah mezi třídami zvaný v syntaxi UML jako ***asociace*** je chápán jako abstrakce konkrétního vztahu mezi budoucími instancemi (abstrakce pro budoucí objektovou referenci).

Visual elementem asociace je spojnice mezi dvěma třídami s doplňkovými informacemi (o kterých bude dále řeč).



obrázek 54 Asociace

V syntaxi UML se agregace chápe jako zvláštní případ asociace, tj. asociace je obecnější pojem a agregace je chápána jako zvláštní případ asociace. Pro asociaci, která není agregací, budeme používat český název „běžná asociace“.

Název asociace

Asociace má jako model element svůj unikátní název, kterým se odlišuje od ostatních asociací. Je třeba však podotknout, že význam tohoto názvu, i když se jedná o jeho jednoznačný identifikátor, není z hlediska modelování až tak důležitý. Název asociace slouží pouze k identifikaci tohoto model elementu a mnohdy se ve visual prvcích modelování jeho zobrazení vynechává.

Poznámka: Sama existence názvu asociace vyplývá z obecného pravidla, že každý model element má svůj název, tedy i asociace. Většinou se název model elementu uvádí v reportech apod.

Agregace a kompozice

Daná asociace buď agregací je anebo nikoliv, což znamená, že je vztahem celek - část anebo nikoliv. Pokud je asociace agregací, může se jednat podle syntaxe UML buďto o tzv. shared agregaci anebo o kompozici (tj. unshared agregaci). Kompozice je silnější forma agregace.

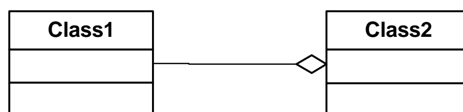
U shared agregace UML dovoluje, aby daný prvek z dané třídy mohl být ve vztahu k několika třídám agregujícím jako agregovaný (sdílená část). Daný prvek se tedy může u shared agregace vyskytovat jako část pro několik jiných prvků. Oproti tomu u kompozice se jedná o unikátní „ownership“ tj. majitelství. Navíc kompozice je natolik silná agregace, že nadřízený prvek nemá smysl bez svého podřízeného prvku.

Poznámka: Z předešlého výkladu objektového modelování jsme zavedli pouze běžnou asociaci a agregaci. Zavedení kompozice toto dělení „zjemňuje v tom smyslu“, že jsme v předešlých úvahách uvažovali o agregaci jako vždy kompozici a to ostatní bylo běžnou asociací. UML dovoluje ještě něco mezi tím, tj. shared agregaci, kdy daný prvek může být částí několika celků. Osobně nedoporučuji příliš rozlišovat, zda se jedná u agregace o shared agregaci anebo o kompozici. Pro objektové modelování je podle mého názoru a podle poznatků z praxe plně dostačující rozlišovat mezi agregací

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

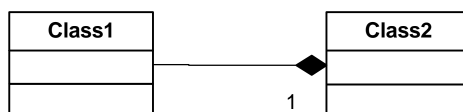
obecně (bez rozdílu shared nebo unshared) a běžnou asociací. Příliš dlouhá diskuse nad „shared nebo unshared“ je podle mého názoru pro projekt časovou a tedy i finanční ztrátou.

Visual prvkem agregace je kosočtverec (diamont), pokud není vyplněn, znázorňuje shared agregaci:



obrázek 55 Agregace zvaná v UML jako shared

a pokud vyplněn je, jedná se o kompozici:



obrázek 56 Agregace zvaná v UML jako kompozice - unshared

Směr asociace

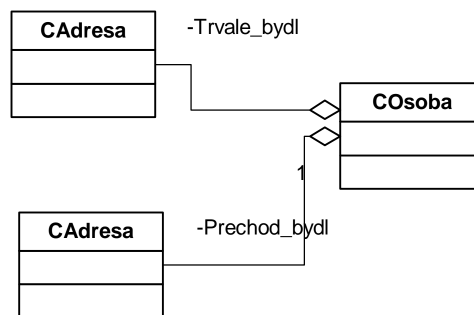
U asociací lze rozlišovat směr, který se v diagramu může také vyznačit šipkou:

Pokud šipku nezavedeme, je asociace chápána jako obousměrný vztah (jako dva jednosměrné vztahy vůči sobě).

Role

U asociace může být vyznačena na každé její straně tzv. role třídy v dané asociaci. Role znamená, jak bude vlastně budoucí instance z dané třídy vůči druhé vystupovat, jakou „roli bude hrát“. Role se ve visual prvku vyznačuje na straně u třídy, která vstupuje do vztahu vůči druhé (na druhé straně, než kde vznikne jedna nebo více instancí třídy). V některých případech se role vynechává, protože je z kontextu použití třídy jasné, o jakou roli se jedná. Jindy je velmi významné roli uvést:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 57 Role v asociacích (agregace)

Některé generátory kódu z CASE nástrojů UML používají role jako názvy generovaných instancí a pokud tedy používáme generování kódu tímto způsobem, doporučuje se role umístit, jinak se názvy rolí vygenerují automatem.

Násobnost vazby mezi třídami

V rámci asociace se také zavádí násobnost vazby na jedné i druhé straně asociace. Násobnost (mulitiplicity) značí, jaká je násobnost vztahu. Nejčastější údaje použité u násobnosti jsou

jedna a pouze jedna	1	
jedna nebo žádná	0..1	
od určitého čísla po určité číslo	M..N	(například od 2..4)
od nuly po libovolné kladné číslo	*	
od nuly po libovolné kladné číslo	0..*	
od jedné po libovolné kladné číslo	1..*	
apod.		

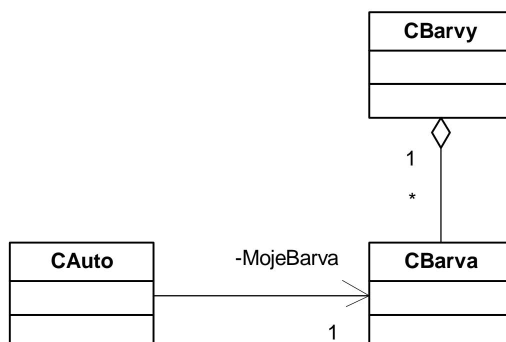
Důležité pravidla modelování u agregace a běžné asociace

1. Každá běžná asociace vede **ke sdílení objektových referencí**. Na jedné straně objekt, který je sdílen, se vyskytuje někde jako součást objektu a tedy jako objekt agregovaný, na straně druhé se objektová reference na něj používá **ještě jednou v běžné asociaci.** V agregaci se objekt rodí a v běžné asociaci se dosazuje.
2. Běžně asociovaný objekt nesmí být zrozen, natož zničen objektem, který jej k sobě asociuje. Na tuto operaci nemá nadřízený objekt, který obsahuje běžnou asociaci na jiný objekt, právo.

Klasický případ agregace a asociace: Číselníky

Jedním z příkladů asociace a agregace je použití „číselníků“, které přiřazují entitám vlastnosti z omezené množiny hodnot. Většinou tyto vlastnosti obsahují pouze id, kód a text. Jeden číselník je tedy jedním seznamem takovýchto vlastností (například barva apod.), tedy Seznam agreguje tyto objekty. Jiné objekty, které je používají pro vyjádření svých vlastností, jej asociují

V tomto příkladu si Auto ukazuje na svou Barvu:



obrázek 58 Použití číselníku jako klasická běžná asociace

Model element asociativní třída

Ze zkušenosti mohu potvrdit, že se zavedením asociativní třídy se začátečníci v objektovém modelování dopouštějí velmi často chyb.

Co je to asociativní třída

Asociativní třída je asociací, která je zároveň třídou. Vztah mezi třídami vyžadující další informaci na této asociaci vede k zavedení třídy.

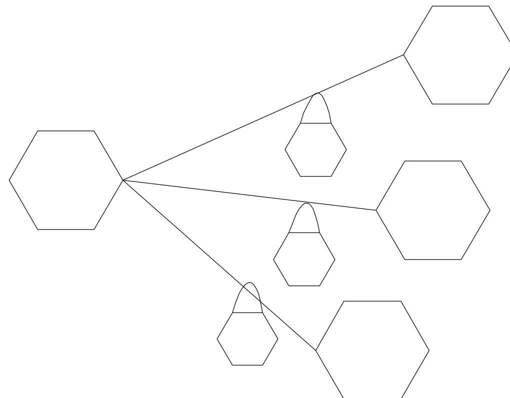
Kdy je bezpodmínečně nutné zavést asociativní třídu?

Nutnost zavést tzv. asociativní třídu nastává **vždy v těchto případech**:

1. Při běžné asociaci mezi třídami * ku * (jinak řečeno M ku N).
2. Sama vazba mezi objekty (realizace asociace) by měla nést nějakou informaci, která patří k této vazbě
3. V kombinaci obou bodů 1 a 2 (velmi častý případ).

Jako další uveďme, že je vhodné, i když ne nutné, zavést asociativní třídu i pro nalezenou běžnou asociaci 1 ku N (pozor nikoliv agregaci). Avšak musím poznamenat, že v praxi se málokdy tato vazba běžná asociace 1 ku N vyskytuje.

Při zavedení asociativní třídy si můžeme představit, že s každým propojením mezi dvěma objekty potřebujeme zavést informaci, tj. potřebujeme „pověsit kartičky“ (objekty) na samotné odskoky mezi objekty.



obrázek 59 Analytická situace vedoucí k nutnosti zavést asociativní třídu

Tato skutečnost vede ke vzniku nového pojmu a tedy nové třídy, tzv. asociativní třídy

Příklad:

Nalezneme v prvním přiblížení, že Zaměstnanec má vztah k firmě a naopak. Existuje seznam Zaměstnanců a seznam Firem. Chceme evidovat „odkdy dokdy“ je Zaměstnanec v dané Firmě zaměstnán. Pokud se podíváme na předešlý obrázek, tak levý objekt můžeme vidět jako Zaměstnance a vlevo jsou tři firmy. Na každém propojení „visí“ jedna informace s údajem odkdy dokdy. Tato informace vede k novému pojmu, k nové třídě, například nazveme tento pojem jako Karta zaměstnance. Vztah je symetrický, je možný i opačný pohled: Můžeme si představit, že vlevo je jedna Firma a vpravo její Zaměstnanci. Na každém linku opět visí informace „odkdy dokdy“. Tato symetrie pohledu je velmi důležitá a ještě se k ní vrátíme

Jiný příklad:

Existuje Seznam Aut a Seznam Majitelů aut. Jedná se o dva nezávislé seznamy, které obsahují jako své agregace objekty Aut a objekty Majitelé aut. Je zřejmé, že mezi Majiteli a Auty je určitý vztah, v tomto případě se jedná o klasický příklad vztahu M ku N. Pro jednoho majitele existuje N aut ze seznamu aut a pro jedno Auto existuje N majitelů aut (bráno z hlediska celé historie).

Toto je první krok analýzy. Provedeme druhý krok: Podle pravidla o vztahu M ku N bychom měli zavést automaticky asociativní třídu, která k tomuto vztahu patří a současně uvážit, jaké (a zda) budou v této třídě další informace. V tomto případě tuto třídu nazveme Majitelství. Je třeba si uvědomit, že od této chvíle existují tři seznamy: Seznam aut, Seznam majitelů a Seznam Majitelství. Každé Majitelství obsahuje jednu instanci Auta a jednu instanci Majitele. Důležité je to, že Seznam Majitelství musí povinně umět (mít metodu) vydat ze sebe pouze Seznam majitelství pro dané Auto anebo Seznam Majitelství pro daného Majitele. Tímto lze vždy získat pro Auto jeho Majitele a pro Majitele jeho Auto.

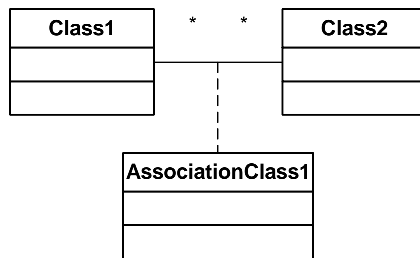
Pro daný příklad by bylo vhodné, aby se ještě asociativní třída doplnila o údaje od do (interval).

Otázka pro čtenáře: Musí být kolekce Majitelství unikátní vzhledem ke kombinaci Auta a Majitele? Pokud ne, kdy nastane případ ne-unikátnosti?

Visual prvek pro asociativní třídu

Asociativní třída je třídou, která je asociací a v UML se značí takto:

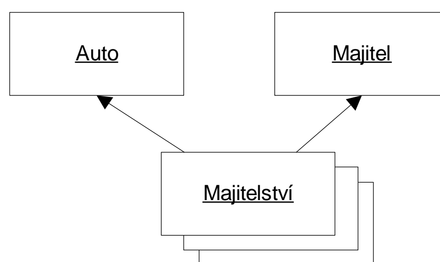
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 60 Asociativní třída

Hlavní význam asociativní třídy

Podívejme se z na zavedení asociativní třídy z hlediska konceptuálního diagramu, který ukazuje objektové reference:



obrázek 61 Konceptuální diagram pro majitelství

V předešlém obrázku je znázorněno, že existuje seznam majitelství (několik majitelství), z nichž každé má jednu objektovou referenci na jedno auto a jednu objektovou referenci na jednoho majitele. Co je pro zavedenou asociativní třídu velmi důležité, a dokonce podstatné, je ta skutečnost, že seznam majitelství musí umět nějakým způsobem vydat ze sebe anebo se musí umět zúžit na seznam podle jednoho auta a z druhé strany také tak podle jednoho majitele. Tedy tento seznam musí „umět být“ anebo „vydat ze sebe“ seznam majitelství pouze pro jedno auto anebo seznam pro jednoho majitele. Tímto „uměním“ seznamu majitelství se může získat omezený seznam majitelství pro majitele a tím seznam aut pro daného majitele. Anebo naopak můžeme získat omezený seznam majitelství podle auta a tím seznam majitelů pro dané auto. Takto seznam majitelství obsluhuje vazbu mezi jednotlivými auty a jednotlivými majiteli.

Zobecněním těchto úvah dojdeme k závěru, že seznam objektů z asociativní třídy musí být obdařen tímto uměním zúžit se anebo vydat ze sebe seznam pouze pro prvek u levého nebo pravého konce asociace.

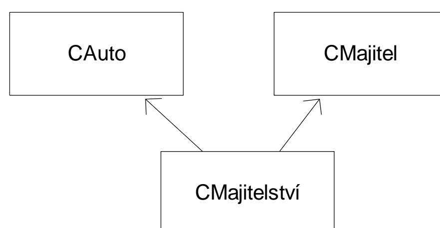
Příklad v pseudokódu. Výpis všech SPZ aut daného majitele, majitel zde „zná“ svá majitelství:

```
For each itMajitelstvi in MyMajitel.ColMajitelstvi
    lstbox_SPZ.Add itMajitelstvi.Auto.SPZ
Next
```

Jiná nepřesnější, ale možná syntaxe asociativní třídy

V souvislosti se zavedením asociativní třídy je třeba poznamenat, že asociativní třída sice má své pevné místo mezi možnými elementy UML, avšak z praktického hlediska je možná ještě druhá syntaxe pro řešení nalezení asociativní třídy, která sice není až tak přesná podle syntaxe UML, ale je v každém případě v praxi vyhovující.

Představme si, že pro příklad Aut, Majitelů a Majitelství nalezneme v modelu tříd (pozor již není řeč o instancích, ale o třídách) následující vztah:



obrázek 62 Majitelství a jeho běžná asociace

Tímto tvrdíme, že jedno majitelství má jednu běžnou asociaci na auto a také běžnou asociaci na majitele. Samozřejmě jedná se o jinou syntaxi než pomocí asociativní třídy. Pokud ale k této nepřesné syntaxi zavedeme seznam majitelství a přidáme mu vlastnost „umění“ vydat seznam majitelství pro jedno auto a naopak také pro jednoho majitele, v konečném výsledku můžeme tento zápis chápat velmi podobně jako zavedení asociativní třídy. Podmínka umění se zúžit pro jeden prvek vlevo nebo jeden prvek vpravo je velmi důležitá, bez ní totiž třída majitelství není schopna obsloužit vazbu mezi autem a majitelem. Musí totiž existovat možnost získat z pohledu jednoho majitele „pouze moje auta“ anebo z pohledu auta „pouze moje majitele“, což získáme pomocí seznamu majitelství zúžením na jedno auto anebo z druhé strany na jednoho majitele.

Poznámka: Mnohdy nastane situace, kdy nalezneme nejprve vztah podle předešlého odstavce, tj. pomocí běžné asociace a teprve po určité době se nahradí syntaxí odpovídající asociativní třídě. Nejedná se o fatální chybu v modelu. K tomuto poznatku mne dovedla praxe: Znám dobře fungující modely dovedené až k realizaci, kde se asociativní třída nezavedla a místo ní byla zavedena předešlá syntaxe.

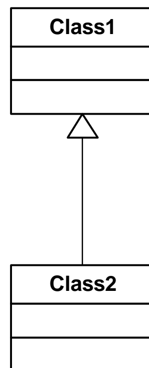
Příklad: Vraťme se k příkladu řešení systému Optimalizace svozu odpadu, kde se zavedly třídy Uzel, Úsek (jako spojnice mezi uzly). Dovedli byste zde určit asociativní třídu? Dovedli byste určit i druhou syntaxi podobnou asociativní třídě?

Model element „Generalizace a specializace“ mezi třídami

Kromě vztahu asociace existuje ještě další důležitý vztah, vztah generalizace a specializace. V OOP se zavádí pojem dědění, který byl vysvětlen v úvodu této knihy. Vztah generalizace a specializace je vztahem obecnějším. Dědění znamená v daném vývojovém prostředí, resp. jazyce zavést v implementaci operaci mezi třídami typu ***Inherits*** (například v C++, Delphi, Javě apod., nikoliv VB 6.0). Na rozdíl od toho vztah generalizace a specializace je vztahem vypozorovaným v analýze **nezávisle na tom, v jakém prostředí bude programováno**. V analýze tedy nalézáme vztah mezi třídami generalizace a specializace, který již v této fázi vyjadřuje re-use mezi nalezenými třídami. Třída ve směru specializace (tj. ta speciálnější) se „odvolává“ na třídu ve směru generalizace a používá její deklaraci k tomu, aby se nemusely společné prvky opakovat.

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Visual elementem vztahu generalizace specializace (též ve zkratce gen-spec) je spojnice mezi třídami s trojúhelníkem ve vrcholu k třídě obecnější:



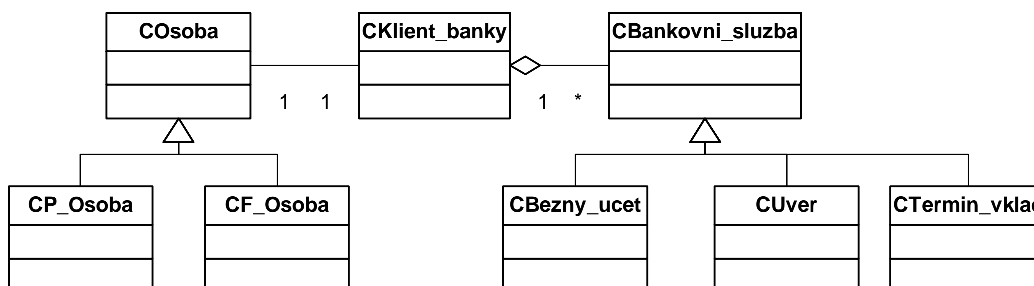
obrázek 63 Vztah gen - spec

Rozdíl mezi vztahem generalizace specializace a děděním lze jednoduše vystihnout slovy: Vypozorovaný vztah generalizace specializace v analýze (tj. který existuje reálně v pozorovaném analytickém modelu a nezávisle na prostředí, v jakém programujeme) lze implementovat různými způsoby, z nichž nejjednodušší je použít dědičnost, pokud je dědičnost daným prostředí podporována.

Z hlediska modelování je důležité znát i tu skutečnost, že zavedení vztahu generalizace specializace umožňuje používat tzv. úrovně abstrakce v řešení tvorby analytických modelů. Pomocí generalizace a specializace lze mnohdy namodelovat vztahy na abstraktnější úrovni. Poté se zavedou třídy, které jsou specializací, získáme tak různé případy téhož obecného řešení.

Příklad:

Všimněme si blíže následujícího obrázku:



obrázek 64 Gen spec a dvě úrovně abstrakce

V tomto návrhu existují dvě roviny řešení. Ta abstraktnější vyjadřuje, že „obecný“ klient může mít „obecné“ bankovní služby, že je „vidí“ svou osobu. Druhá, nižší a speciálnější úroveň nám říká, že Osoba může být buď právnickou osobou (zkratka P), anebo fyzickou osobou (zkratka F) a že bankovní služba může být buď běžný účet nebo úvěr nebo termínovaný vklad.

Všimněme si možnosti přidat další možné typy osob anebo typy služeb do spodní úrovně.

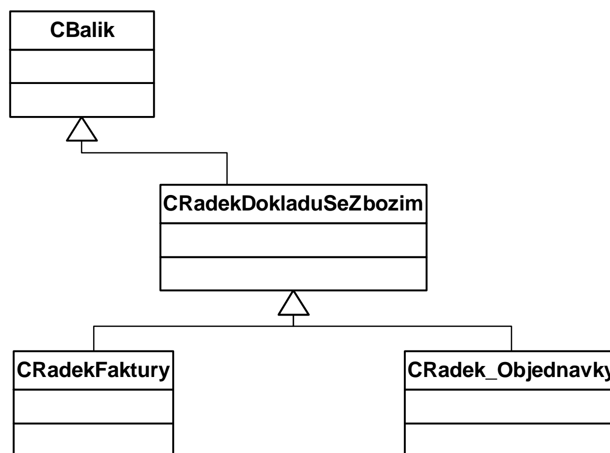
Poznámka:

Syntaxe UML 1.3 povoluje zavést vztah generalizace mnohem obecněji, nikoliv pouze pro třídy. Každý model element, který potřebuje zavést tento vztah, se zavede jako dědic Generalizable elementu. Nejedná se tedy v UML pouze o vztah mezi třídami, ale všech elementů, které lze do tohoto vztahu umístit (například Use Case, Actor apod.)

Použít pro řešení re-use asociaci nebo generalizaci - specializaci?

Na první pohled se může jevit otázka v nadpisu jako nesmyslná. Ale ukážeme si, že se opravdu mohou nabízet pro jeden a tentýž analytický problém několik možností. Tedy dotaz v nadpisu není až tak nesmyslný!

Vraťme se k našemu příkladu s re-use, ve kterém jsme zavedli Balík zboží a připomeňme si tuto konstrukci. Nalezli jsme, že se mnohé údaje v Řádku faktury a Řádku objednávky a podobných řádcích opakují. Zavedli jsme jeden univerzálně pojatý Balík zboží a vložili jej jako agregaci do Řádku faktury. Všimněme si, že jsme re-use pro Balík zboží vyřešili pomocí vložení instance typu Balík do dané instance. Podobně bychom však mohli tentýž problém vyřešit pomocí vztahu generalizace specializace a zavést různé typy řádků se společným předkem, Balíkem zboží. Tím pádem jak řádek faktury, tak řádek objednávky získá umění balíku zboží:



obrázek 65 Radek faktury a radek objednavky "umí" být balíkem zboží

Pokud se ptáme, které z těchto řešení je výhodnější, tak odpověď není až tak jednoznačná. Oba dva přístupy mají své výhody a nevýhody.

Postup při tvorbě modelu tříd

V praktickém přístupu se při tvorbě objektové aplikace v prvním kroku objektové analýzy vyhledávají třídy, které mají analytický význam **abstraktní entity z problémové domény**. Analytickou abstraktní entitou problémové domény máme na mysli abstrakci pojmu, který vystupuje a je objeven v předešlé fázi analýzy, zejména v Use Case modelu. Současně se hledají všechny entity odvozené, jako

seznamy původních entit, což jsou kandidáti na třídy seznamů (kolekce). Namísto množného čísla použijeme raději ustálený výraz *Seznam* (*Seznam Osob apod.*). Tyto kandidáty zatím nenazývejme třídami, objekty apod. Nejvhodnější výraz pro ně je ***informace*** nebo ***entita*** (např. informace *Osoba* apod.)

Nejefektivnější postup je projít Use Case model „slovo od slova“ a vyhledávat podstatná jména mající význam abstraktní entity. Tímto získáváme všechny kandidáty třídy. Můžete tyto kandidáty zdůraznit například tak, že jim dáme velké počáteční písmeno a napíšete je proloženým tiskem. Avšak pozor, ne každý kandidát na třídu bude opravdu třídou a záleží na další postupu v analýze!

Důležité pro další vyhledávání tříd je také současné rozpoznání relací, tj. vztahů mezi kandidáty (vztahy mezi informacemi). Hledáme a posuzujme současně ty kandidáty, kteří jsou ve vzájemném vztahu z problémové domény.

Tento vztah mezi kandidáty se vyhledává poměrně jednoduše, a to tak, že každý pojem (tj. kandidát), na který narazíme v problémové doméně, vyslovíme v plném znění takovém, aby měl konkrétní a hlavně jednoznačný význam. Pokud některé ze slovního spojení vyjádření pojmu vynecháme, potom také kandidát svou jednoznačnost v textu ztrácí.

Při hledání vztahů mezi kandidáty použijeme zápis *skládání informace*. Vztahy mezi kandidáty vyjádřené jako vztah ve druhém pádu (koho čeho), například *Rodné číslo Majitele* apod., vyjádříme pomocí skládání informací. Skládání informace **A** pomocí **B, C, D** vyjádříme jednoduše syntaxí pomocí závorky:

A = (B, C, D, ...)

Tento seznam skládání nepovažujeme za uzavřený a při další analýze jej doplňujeme. Je třeba si uvědomit, že hovoříme stále o kandidátech (informacích) a nikoliv objektech nebo třídách (zatím nevíme, jak se role **A, B, C, D. atd** projeví v modelu tříd). Postupně probíráme jednotlivé kandidáty z problémové domény. Začínáme nejprve vyhledávat ty kandidáty, kteří mohou existovat sami o sobě, aniž by jako pojmy skládaly jiné kandidáty, tedy vyhledáváme ty, o kterých má smysl hovořit bez ohledu na ostatní kandidáty. Tito kandidáti jsou „nejžhavějšími“ pro to, aby se stali třídami.

První možnost je, že pojem jako kandidát se stane skutečně třídou. V důsledku to znamená, že entita z problémové domény (například *Majitel*) je v objektové analýze zavede jako třída. Z ní pak mohou vznikat objekty dané třídy jako její instance, podle „kopyta“ třídy (několik instancí *Majitelů* podle potřeby). Jednotliví kandidáti, kteří skládají tohoto sledovaného kandidáta (o skládání viz předešlý odstavec), dávají vzniknout buď atributům objektů z dané třídy, anebo dávají vzniknout objektům skládajícím objekty z dané třídy.

Ve vztazích také vyhledáváme kardinalitu, tj. zda se jedná o vztah 1:N, M:N atd.

Atributy, které vzniknou z pojmů, mohou mít sice podle UML 1.3 násobnou kardinalitu (tj. může vzniknout tzv. násobný atribut s multiplicitou N), ale vede to k datovým strukturám uvnitř tříd typu „array“, které nic jiného neumí, než „držet pole hodnot“. Doporučuji se této konstrukci vyhnout (používat ji pouze výjimečně) a nezavádět násobné atributy, protože většinou signalizují vznik nového pojmu (nové třídy a z nich seznam).

Příklad: Nezavedeme násobný atribut Řádky faktury s údaji jako array hodnoty, ale zavedeme Seznam řádků faktury jako seznam instancí objektů z třídy Řádek faktury.

Zkoumáme vztahy ve smyslu „vidí anebo obsahuje“, tj. agregace a běžná asociace. Pokud se daný pojem dosazuje, potom se jedná o objekt z určité třídy.

Další možností je, že kandidáta navrhneme jako ***atribut objektu***. Takovýmto atributem se může stát pouze ten kandidát, který vystupuje v nalezené vazbě jako skládající nějakou entitu v kompozici. Je třeba si uvědomit, že atribut určitého datového typu musí být umístěn do určité třídy (ze které pocházejí objekty s daným atributem), tedy určení „tento kandidát nebude třídou, ale atributem“ znamená identifikovat také třídu, do níž atribut patří. Atribut nemůže stát samostatně bez této třídy. V

předešlém příkladě můžeme tedy psát: Kandidát *Majitel* dá vzniknout třídě budoucích majitelů, kandidáti *Rodné číslo*, *Jméno*, *Příjmení,* budou atributy.

Důležitá je jedna věc: Naše rozhodnutí nemusí být konečné. Každý atribut je stále kandidátem na vznik třídy, která vystupuje v kompozici vůči nadřízené třídě. Může se stát, že tento náš první návrh (kandidát bude atribut) nakonec opustíme a například navrhneme existenci třídy *Rodných čísel* a *Majitel* bude obsahovat jako kompozici objekt *Rodné číslo* z třídy *Rodných čísel* a nikoliv atribut! Základní otázkou pro rozhodnutí zda „vznik atributu“ anebo „vznik objektu z třídy“ je to, zda daná entita má být vybavena nějakým uměním a zda toto umění má ona provádět. U *Rodného čísla* existuje jedno takové umění, například kontrola na modulo 11, nebo dej Rodné číslo v různém formátu. Máme dvě varianty, buď *Rodné číslo* bude umět samo modulo 11, bude umět vydávat různé formáty, nebo to bude umět *Majitel Rodného čísla* jako jeho správce.

Další možností je, že některé informace můžeme chápat jako informace téhož typu lišící se pouze svou rolí. Tuto situaci rozpoznáme podle toho, že dva a více kandidáti jsou složeni ze stejných informací. V tom případě navrhneme pouze jednu třídu, ale několik instancí této třídy. Někdy se stane, že ve vztahu pojmů nastane situace známá jako vztah generalizace a specializace. Tento vztah odhalíme jednoduše tak, že pozorováním skladeb zjistíme, že některé pojmy jsou si podobné průnikem svých skládajících informací. V tom případě se snažíme průnik kandidátů vyjádřit jako jednu entitu a oba kandidáty vyjádříme jako dědice (přesněji specializace) kandidáta z průniku pojmů.

Zvláštní kapitolou je vyhledávání asociativních tříd, ale to už je nám známo: Připomeňme, že zásady pro její zavedení jsou uvedeny v příslušné kapitole o asociativní třídě.

Objekty, třídy a persistence

V objektovém programování a modelování mají procesy spojené s persistencí informace odlišnou polohu, než v datových modelech a v Coad-Yourdonově škole. Tam byly datové entity a následně tabulky „tím hlavním“ nositelem informace. V OOP jsou těmito prvky objekty. Jaká je tedy poloha persistentních dat v OOP?

Základní problém spočívá v tom, že takřka v každém objektovém prostředí **objekty neumějí samy od sebe persistenci a práci s ní**. Znamená to, že tyto objekty jsou sice nositeli informace, avšak tato informace apriori není v těchto prostředích persistentní.

Poznámka: Čest výjimkám, jako je například prostředí CACHÉ.

Znamená to, že tvůrci systémů v tomto prostředí musejí jako součást své práce nutně zahrnout i své řešení persistence objektů, tj. „ručně“ objektům přidat toto umění. Nejčastěji bývá toto umění opřeno o relační databázi umístěnou v pozadí za těmito objekty.

Obecně je scénář následující:

Objekt, který potřebuje odložit svoje data, o toto požádá objekt datové vrstvy s tím, že mu předá údaje, které chce odložit. Tento pomocný objekt údaje převezme a odloží je do databáze a vrátí objektu klíč neboli „stvrzenku“ k těmto údajům. Objekt musí mít k dispozici nějaký algoritmus, klíč, (uvedenou „stvrzenku“), která vede k opětnému vytěžení odložených dat.

Poznámka: Postup je velmi podobný, jako v úschovně zavazadel. Do úschovny odložíme přes „jedno okénko“ zavazadla a dostaneme stvrzenku, která udává klíč, jak se k těmto zavazadlům opět dostat. Na požádání můžeme opět zavazadla vyzvednout.

Mezi údaji odloženými v databázi a strukturou objektů musí být pochopitelně nějaký logický vztah. Podle toho, jak „objekt vypadá“, taková bude potřebná datová struktura v databázi. Proces vytváření požadovaných datových struktur podle struktur tříd se nazývá mapování do databáze.

Existuje několik možných databází, které můžeme použít jako „úschovnu dat“. Jedna z nich je samozřejmě relační databáze, jiné jsou například stromové databáze, vlastní souborové systémy apod.

Zajímavé jsou taková prostředí, v nichž objekty umějí již persistenci „samy od sebe“. V tomto prostředí vytvořené objekty „automaticky“ umějí persistenci díky tomu, že toto prostředí „nějak objekty persistenci naučilo“. Jako klasický příklad uvedu databázi CACHÉ (viz například stránky www.intersystems.cz resp. www.intersystems.com). Tato databáze funguje tak, že při kompilaci se v pozadí automaticky podle nějakého algoritmu vygenerují stromové struktury M-databáze. Jinak řečeno prostředí samo provede automatické mapování tříd do databáze (které je skryto a není pro uživatele prostředí patrné). Objekt tímto umí persistenci „sám od sebe“.

Zvláštním a často používaným případem je ta situace, kdy v pozadí objektů existuje relační databáze. Potom je třeba provést mapování tříd speciálně do relační databáze, tj. vytvořit datové struktury v relační databázi, které jsou schopny „uspokojit“ objekt stojící u „přepážky úschovny dat“.

Mapování objektů do relační databáze

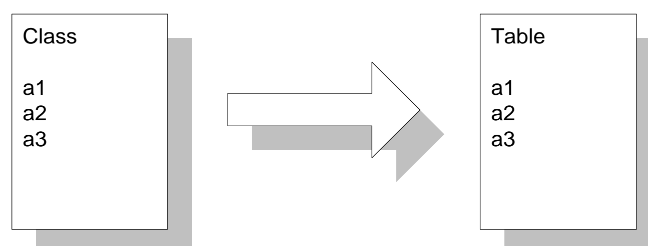
Nejprve připomenu, že tak zvané mapování objektů do relační databáze je proces, který spadá do fáze designu, tj. do fáze po zhotovení analytického dokumentu (viz kapitola o **I+I** metodě vývoje). Jeho cílem je nalézt takové datové struktury a takové funkcionality v relační databázi, které umožní objektům provádět persistenci svých informací speciálně v relační databázi.

Pro toto mapován tříd do relační databáze doporučuji dodržovat tyto zásady

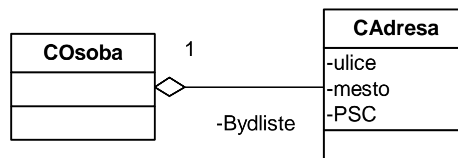
- Používejte k vazbám tzv. systémové identifikátory (autoincrement, identity apod.), slangově zvaná „idečka“ pro svou zkratku resp. předponu `id`. Vazby přes tyto systémové identifikátory mají velmi blízko k odpovídajícím objektovým referencím. Mapování je poté jednotným procesem a obsluha vazeb je rychlejší.
- V prvním kroku proveďte mapování tříd do tabulek ve tvaru 1 : 1. V druhém kroku proveďte optimalizaci, pokud je jí třeba

Mapování jedné třídy a jejich atributů

Vytvoří se tabulka s danými sloupci odpovídající atributům objektu dané třídy, tj. 1 : 1:

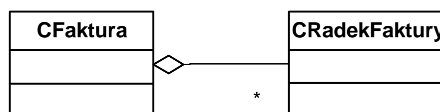


Mapování vztahu agregace 1 : 1



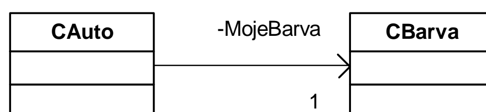
V tabulce klíč tabulky odpovídající agregovanému objektu putuje do tabulky odpovídající nadřízenému objektu. Tabulka osoba má cizí klíč id_adresa

Mapování vztahu agregace 1 : N



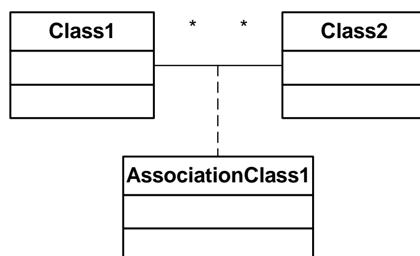
V tabulce cizí klíč tabulky odpovídající nadřízenému objektu putuje do tabulky podřízeného objektu. Tabulka řádek faktury má cizí klíč id_faktura.

Mapování běžné asociace na jeden objekt (vazba na item číselníku)



V tabulce cizí klíč odpovídající asociovanému objektu (který je viděn, tj. item číselníku) putuje do tabulky, odpovídající objektu, který asociovaný objekt vidí. Tabulka auto má cizí klíč id_barva

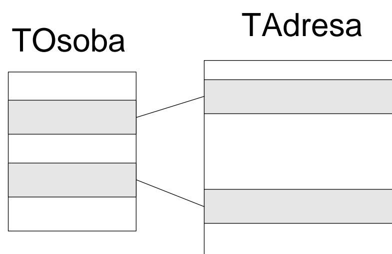
Mapování vazby M : N – asociativní třída



Mapováním vzniká tzv. asociativní tabulka. Do tabulky odpovídající asociativní třídě putují cizí klíče jak z tabulky odpovídající z jedné strany asociace, tak cizí klíč z tabulky odpovídající třídě z druhé strany asociace. Zde do tabulky associationclass1 putuje cizí klíč z tabulky class1 a cizí klíč z tabulky class2.

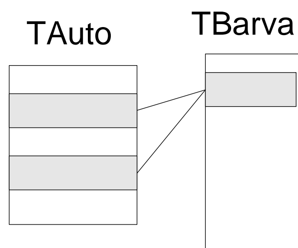
Rozdílné vztahy mezi daty při běžné asociaci a agregaci

Není bez zajímavosti, že vztah agregace a běžné asociace mezi objekty se projeví také ve vztahu mezi daty, a to nikoliv přes klíče, ale vztahem mezi záznamy jako takovými. Představme si vztah mezi záznamy mezi dvěma tabulkami při agregaci, kdy osoba má adresu jako agregaci:



S novou osobou se také zakládá nová adresa nezávisle na tom, zda dané údaje adresy již v systému existují anebo nikoliv. Tedy záznamy se ze zásady při agregaci nesdílejí! To platí i pro agregaci 1:N, například u řádků faktury, kdy faktura vlastní svoje řádky a při založení nové faktury se zakládají nové řádky (i kdyby měl tento řádek úplně stejné údaje jako již nějaký jiný existující řádek faktury)

Při běžné asociaci se naopak **shoda vyžaduje, pokud nastane.** Je povinností shodu vyhledávat.



Dvě auta mající stejnou barvu sdílejí stejný záznam v tabulce barev.

Pro zajímavost můžeme si představit, jak by vypadala situace, kdybychom u adresy přešli od modelu agregace na běžnou asociaci. Při zadání nové adresy by se muselo pomocí údajů města, ulice a PSČ nalézt odpovídající již existující adresa a pokud existuje, tak by se na ni provedl odkaz. Pokud by neexistovala, musela by se založit a poté by se provedl odkaz. V tomto modelu by se daly najít osoby na téže adrese přes shodu cizích klíčů id_adresa, v modelu s agregací se musí zásadně hledat shoda údajů a nikoliv id (id je vždy různé).

Poznámka: Podobně pracují tzv. Registry obyvatel, kde jsou odkazy na města, ulice a dokonce na čísla popisné (tj. vede se evidence všech měst, evidence všech ulic a všech možných bydlišť).

Mapování dědičnosti do datového modelu

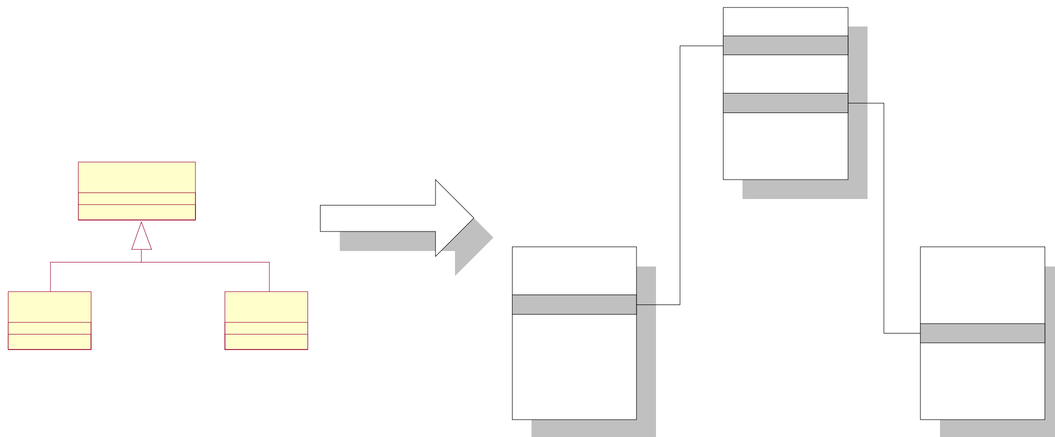
Pokud třídy v modelu tříd mají vztah dědičnosti, mapování do databáze opět vede k několika možným technikám a způsobům návrhu datového modelu.

Existují celkem tři možné způsoby, jak provést mapování dědičnosti do tabulek. Začínáme vždy tím prvním, který odpovídá vztahu 1:1 a poté můžeme provést optimalizaci, což jsou další dva možné způsoby.

I. způsob (1 : 1)

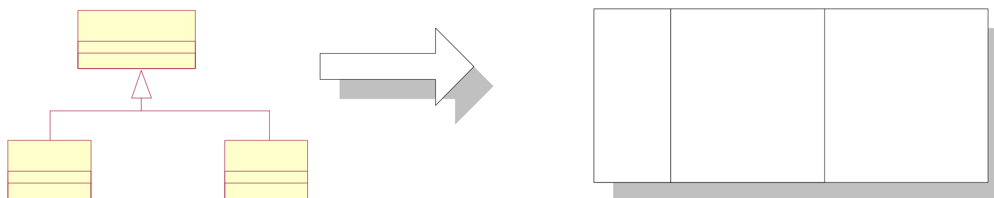
V tomto případě se každá třída (jak předek, tak její potomci) mapují do databáze jako tabulky 1 : 1, přičemž tabulka předka má vztah vůči ostatním jako silný vztah 1 : 1. Chová se tak jako „root“

nadřízená tabulka vůči svým podřízeným tabulkám 1:1. Jeden záznam v nadřízené tabulce odpovídá jednomu záznamu v některé z podřízených a naopak. Tento obrázek „mapuje přesně" vztah dědičnosti bez dodatečných úprav:



II. způsob

V tomto případě se data pro atributy objektů jak z třídy předka, tak potomka navrhují do jedné tabulky. Z hlediska optimalizace struktury dat se jedná o „schválně neoptimalizovanou strukturu", protože v jednom řádku se vyskytují data jak předka, tak od „všech potomků", přičemž platí data pouze od jednoho potomka a ostatní sloupce nemají žádný význam.



Pokud se tedy provede výběr jednoho záznamu z takovéto tabulky, potom některé ze sloupců nemají smysluplnou hodnotu a význam mají pouze sloupce pro daného potomka. Samozřejmě takovýto návrh není flexibilní, protože přidání dalšího potomka na stejné úrovni znamená změnu struktury této tabulky (přidání sloupců), kdežto v předešlém případě se jednalo o přidání celé tabulky (což je z hlediska údržby databáze flexibilnější). Tento návrh se provádí v případech optimalizace na rychlost podle vyhledávání u dat v předkovi, kdy další vazby do tabulek kriticky zpomalují operace.

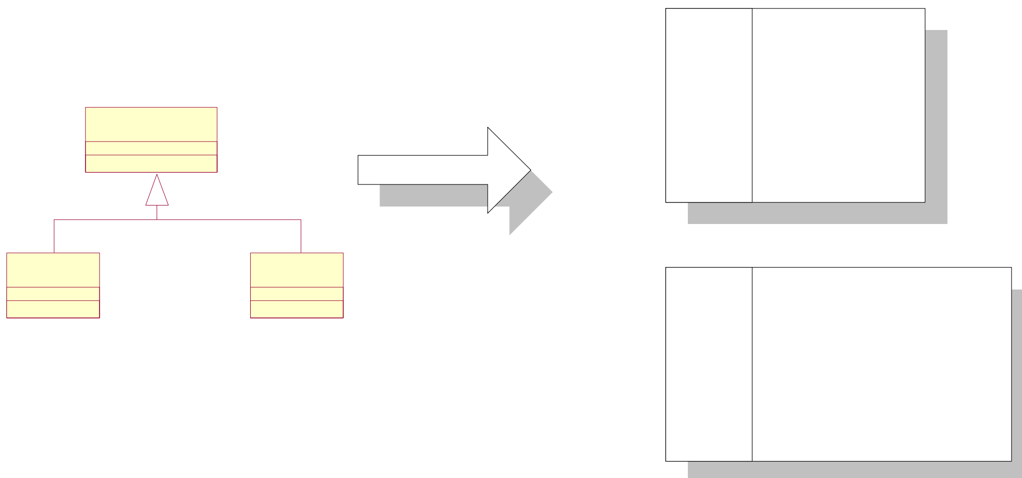
Nevýhody tohoto přístupu jsou důsledkem zavedené „nečistoty v řešení":

- Při změnách v tabulce musíme být opatrní na zavlečení chyb do jiné oblasti řešení (pracujeme nad stejnou tabulkou pro několik entit). Chybný zásah se může projevit v jiné části řešení a někdo „může být překvapen, co se to děje"
- Problém implicitních hodnot a constraintů. Při vkládání údajů do „našich" sloupců musíme obsloužit constrainty v jiných sloupcích
- Délka záznamu může být kritická

III. způsob

V tomto případě se každý potomek mapuje do své tabulky, ale současně se do této tabulky přidají sloupce data předka (předřadí se sloupce odpovídající předkovi).

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



Jedná se o „úmyslně nenormalizovanou strukturu“, kde řešení je někde mezi I a II.

Object model

Object model (objektový model) nebo také instance model vyjadřuje vztah mezi instancemi a nikoliv mezi třídami. Z podstaty věci vyplývá, že tento model je odvoditelný z Class modelu (je na něm „matematicky závislý“).

Platí jednoduché pravidlo ve vztahu instancí a jejich tříd:

- Z tříd vznikají instance objektů procesem „instanciování“, tj. realizací (konkretizací) třídy do instance
- Z asociací (včetně agregací) takto vznikají tzv. **linky** mezi objekty, tj. link je vazbou mezi instancemi
- Společným procesem přechodu tříd do instancí a přechodem asociací do linků z Class modelu vznikne Object model, který se tak stává jedním (z nekonečně mnoha možných) instancí z Class modelu

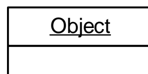
Naskýtá se pochopitelně otázka, nač tedy používat Object model, když informace, kterou nám podává, je odvoditelná a „nejedná se tedy o nic nového“.

Object model má oproti Class modelu jednu velkou výhodu: Je o mnoho snáze pochopitelnější a čitelnější, než Class model. Class model má totiž abstraktní podobu, nehovoří o konkrétních instancích, ale pohybuje se v abstraktnější rovině tříd (které tímto zastupují abstraktní pojmy jako Osoba, Rodné číslo atd.)

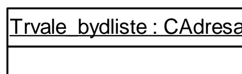
Object model stává velmi dobrým východiskem pro tvorbu pojmů a následně tříd pro svou srozumitelnost. Jako příklad mohou posloužit úvodní diagramy zavádějící Balík zboží (viz úvodní kapitoly této knihy).

Model element „Object“ v UML

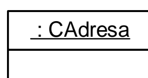
Model element Object vyjadřuje jednu instanci, tj. jeden objekt a již byl v této knize použit jako model element v sekvenčním diagramu. Jeho visual elementem je obdélník podobně jako u třídy s tím rozdílem, že název instance je povinně **podtržen**:



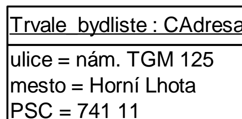
Pokud je známo, z jaké třídy pochází daná instance, můžeme označit tuto skutečnost za názvem instance, oddělovačem je dvojtečka:



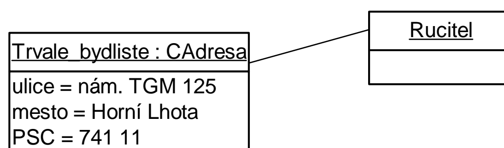
Anebo můžeme název instance vynechat a ponechat pouze název třídy, v tom případě hovoříme o **anonymním objektu:**



Protože objekt je konkrétní instancí třídy, v jeho atributech se mohou vyskytovat konkrétní hodnoty. Například třída CAdresa zavádí atributy ulice, město a PSC. Konkrétní objekt trvalého bydliště má konkrétní hodnoty, což lze v UML znázornit přiřazením hodnot do atributů:



Link mezi objekty se značí jednoduchou čarou v UML bez dalších informací:



Použití Object modelu v praxi

Pokud bychom zavedli nějaký Object diagram a v něm objekty a linky mezi nimi, tak tyto objekty by měly mít nějaké názvy (pokud to nejsou anonymní objekty). Zavedeme tedy tyto názvy objektů a z hlediska „čisté teorie“ modelování by se tyto objekty se svými názvy měly objevit v naprogramovaném systému. Ale to je pouze teorie, která se velmi těžko dá dodržet…

V praxi se většinou Object diagramy chápou jako **příklady vztahu mezi instancemi** ve smyslu „představme si objekt jako nějakou Fakturu, která má nějaké Řádky faktury atd.“. V systému se pak budou vyskytovat nikoliv objekty Faktura a Řádek faktury zavedené v Object diagramu, ale například EditovanaFaktura apod.

Poznámka: Uvedená slova berte jako osobní doporučení. Mohli byste se rozhodnout, že budete „pedanti“ a celý systém navrhnete pomocí Class modelu a instancí Object modelu. To je vaše samostatné rozhodnutí a má tu výhodu, že naprogramovaný systém bude úplně přesným obrazem jak modelu tříd (to musí být vždy), tak modelu instancí a to ***bez žádných odchylek v názvech instancí****.*

Osobně to považuji za velmi nepraktické, protože názvy instancí nejsou na rozdíl od tříd až tak pro systém rozhodující. Samozřejmě u tříd a příslušností objektů do tříd nemáme žádnou libovůli pro změny názvů!

Jak urychlit modelování? … Object diagramem

Osobně se domnívám, že význam Object diagramu se v modelování velmi podceňuje. Ostatně i mne teprve až dlouhodobá praxe v komunikaci s uživateli při konzultacích naučila „ocenit“ význam Object diagramu. Po několikeré konzultaci jsem zjistil, že rozhovor s uživatelem neznalým programování vždy vedl k formulacím ve stylu: „Představme si Fakturu, nějakou libovolnou Fakturu, a co ta obsahuje?“ apod. a přitom se pochopitelně maloval Object diagram ve smyslu příkladu vztahu objektů (nikoliv Class diagram).

Je třeba podotknout, že při rozhovoru s uživatelem činilo vždy problémy zapsat danou situaci v Class diagramu, protože se příliš času věnovalo vysvětlování nějaké abstrakce. Přitom v Object diagramu se vždy velmi rychle dobralo k výsledkům.

Tato zkušenost mne přivedla k vlastním vyjadřovacím prostředkům (čistě praktickým), které již vybočují mimo rámec UML. Znamená to, že to, co nyní budu popisovat, je sice doporučení, avšak toto doporučení **neodpovídá standardnímu používání UML!** Zavádím v komunikaci s uživatelem nový typ modelu velmi blízký objektovému diagramu. Navrhované změny v syntaxi oproti objektovému modelu, které se mi osvědčily v praxi, jsou následující:

Vyznačení směru linku v diagramu

Osvědčilo se mi doplnit k linku případně směr tohoto linku stejně jako u asociace. Tento směr odpovídá vyznačenému směru v odpovídající asociaci. Syntaxi doporučuji stejnou, jako v případě asociace (šipka na konci u třídy, v tomto případě šipka na konci u „vložené“ objektové reference na objekt).

Vyznačení agregace

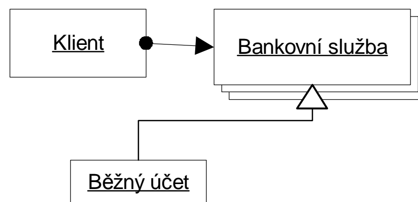
Dalším doplněním je vyznačení agregace pomocí černého puntíku v linku na straně objektu, který agreguje podřízený objekt (na stejné straně jako je kosočtverec u asociace typu agregace). Toto vyjádření označuje, že asociace, ze které tento link vznikl, je agregací a to v odpovídajícím směru.

Vztah generalizace a specializace

I když to může znít jako protimluv a nesmysl, osvědčilo se mi v konceptuálním diagramu, který je vlastně obdobou objektového, zavést obdobu vztahu generalizace specializace. Ovšem je třeba hned vysvětlit, jak v tomto diagramu vztah vlastně chápat. Problém je v tom, že ve vztahu mezi objekty nemá smysl hovořit o vztahu generalizace a specializace v tom pojetí, jako mezi třídami. Instance jsou totiž konkrétní objekty bez vzájemného vztahu re-use, který se v tomto případě projeví až použitím vztahu generalizace a specializace mezi třídami.

Zavedený vztah gen - spec v tomto konceptuálním diagramu vyjadřuje **možnost použít danou instanci nižšího kontextu specializace ve vyšším kontextu generalizace**. V konkrétním programování (například C++ apod.) tomuto vztahu odpovídá kompatibilita typů a možnost dosazení jedné proměnné (objektové reference) za druhou. Vztah mezi dvěma koncepty (pojmy), tj. „objekty“ ve vztahu generalizace a specializace označujme v konceptuálním diagramu stejným způsobem, jako ve vztahu tříd, tj. spojnicí s trojúhelníkem:

Příklad:



V tomto příkladu jsme vyjádřili, že za klient drží N bankovních služeb a za tuto službu „lze dosadit“ běžný účet.

Zavádějící v tomto diagramu může být to, že vrchní obecnější kontext (de facto objektová reference) nemusí být konkrétní instancí, ale pouze prázdnou objektovou referencí, do které lze nižší instanci tj. tuto objektovou referenci dosadit. V odpovídajícím Class modelu tomu odpovídá situace, kdy vyšší třída je abstraktní třídou.

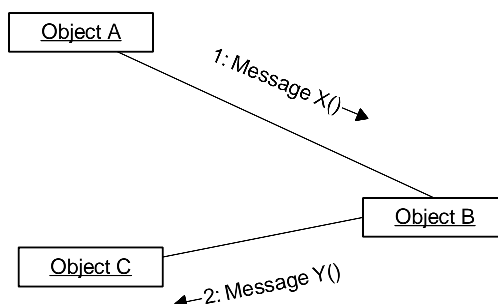
Na závěr ještě připomenu, že konceptuální diagram není součástí UML, které povoluje v objektovém diagramu pouze zavádět objekty a jednoduché linky.

Collaboration model

Dalším modelem UML je tzv. Collaboration model, často překládaný jako model spolupráce objektů. Tento model vychází z již zde uvedených dvou modelů a to modelu objektového a modelu sekvenčního. Lze jej chápat jako „hybrid dvou modelů“, modelu sekvenčního a objektového.

Základem Collaboration modelu je objektový model. Vytvoří se diagram objektového modelu (tj. objekty jako obdélníky s podtrženými názvy a spojnice mezi nimi jako linky). Tento diagram se doplní o zaslání zpráv podél těchto linků, přičemž syntaxe těchto zpráv je stejná, jako u diagramu sekvenčního. Pro znázornění pořadí zpráv se používá číslování zpráv s oddělovačem dvojtečka.

Příklad:



obrázek 66 Model spolupráce objektů

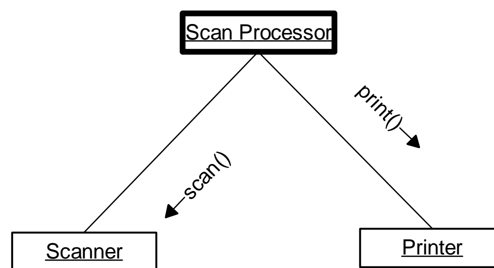
Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Je pochopitelné, že pokud namalujeme Collaboration diagram bez zpráv, tak jej můžeme považovat za znázorněný Object diagram. Coplaboration diagram nese velmi podobnou informaci jako sekvenční diagram, oproti sekvenčnímu diagramu má Collaboration diagram výhodu v možnosti znázornit současně i statickou strukturu spolupracujících objektů. Na druhou stranu sledování sekvence zpráv může někdy připomínat hádanky z nedělní přílohy novin, kde se hledají podle čísel další sekvence bodů v obrázku.

Poznámka: Osobně Collaboration diagram nepoužívám. Doporučuji používat sekvenční a objektový diagram.

Při čtení Collaboration diagramu se můžete setkat ještě se syntaxí tzv. ***aktivních objektů***. Existují scénáře spolupráce objektů, ve kterých určité objekty vystupují jako nositelé scénáře a „drží“ dělení práce mezi objekty a řídí je. Tyto objekty se nazývají aktivní objekty (active objects). Ne vždy však musí takovýto objekt existovat. Aktivní objekty se vyznačují zesíleným obdélníkem.

Příklad:



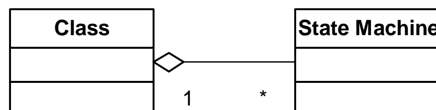
obrázek 67 Scan processor jako active object

Statechart diagram – stavový diagram

Stavový model je znám již ze strukturálního programování, kde se systém chápe jako stavový stroj. Na rozdíl od tohoto pojetí se stavový diagram v OOP chápe vždy v souvislosti s objekty a jejich stavy. Jeden z rozdílů strukturálního programování a OOP je právě v tomto pohledu. Protože ve strukturálním programování neexistují objekty, tak se těžko stavy vztahují „k něčemu“. V objektovém programování sám pojem objekt s sebou přináší svojí podstatou odpověď na otázku, kdo je vlastně nositelem stavů, tj. objekt sám.

Protože objekty pocházející ze stejné třídy musí mít také stejné stavy a jejich přechody, tak se zavádí jednoznačný vztah mezi třídami a stavovými stroji:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 68 Jedna třída obsahuje N stavových strojů

Stavový stroj přiřazený k třídě se vyjadřuje pomocí stavového diagramu.

Význam stavového diagramu tříd

Je vcelku pochopitelné, že přímo ze stavového diagramu „nelze kódovat", jinak řečeno stavový diagram nepatří k nezbytně nutným pro vytvoření systému a také se na něj pochopitelně nevztahuje požadavek úplnosti modelu. Stavový diagram však může velmi napomoci jako doplňková informace při tvorbě jiných modelů, při verifikaci mezi modely atd. Stavový model tak může urychlit tvorbu systému.

Příklad: V jedné bance při tvorbě informačního systému bylo úkolem zpracovat analyticky chod směnek různého typu, o kterých jsme do té doby nevěděli vůbec nic. Konzultant z banky nám nabídl „plachtu", na které bylo namalováno „jak to v této bance chodí se směnkami". Při bližším studiu této „plachty" jsme zjistili, že se vlastně jedná o stavový model směnky a nejlepší by bylo zapsat tuto informaci do stavového modelu a tak tuto informaci přenést do prvního modelu UML.

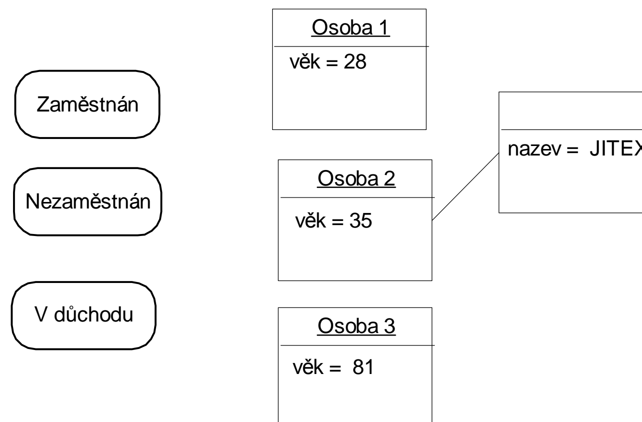
Model element State

Jeden stav objektu se jednoznačně identifikuje svým názvem a vyznačuje se v diagramu jako obdélník se zaoblenými rohy (obdélník styl „nálepka")

Jiný stav je odlišen jiným názvem. Stav objektu nemusí být dán pouze hodnotou nějakého (jednoho) atributu, ale může být odvozen od hodnot (například „hodnota atributu větší než …"), nebo celé kombinace podmínek pro atributy anebo dokonce také naplněním resp. nenaplněním nějaké vazby.

Příklad:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



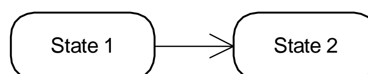
Zde je osoba buď ve stavu zaměstnána, nezaměstnána, nebo v důchodu. Rozhodující je

- věk osoby
- naplnění vazby na Zaměstnavatele (zde je návrh modelu nikoliv přes asociativní třídu, ale jednodušší model

Model element Transition, Přechod mezi stavy

Díky dynamice v objektu a díky přijímání zpráv a spouštění metod se mění stavy objektů. Připomeňme si při té příležitosti pravidlo konzistence vnitřních stavů objektů: Objekt nemění svůj stav „jen tak“, ale vždy díky své vlastní aktivitě (vyvolanou zasláním zprávy).

Změna stavu z jednoho do druhého se ve stavovém diagramu zavádí jako transition, tj. přechod mezi stavy. Přechod je jednoznačně určen svým názvem a zavádí se mezi dvojici stavů, mezi výchozí a konečný stav. Jeho visual elementem je šipka spojující výchozí stav a konečný stav:



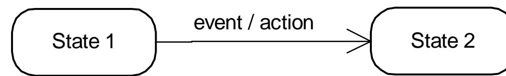
obrázek 69 Transition, přechod mezi stavy

Zde v tomto diagramu bychom četli informaci „objekt může za určitých podmínek přejít ze stavu A do stavu B“, avšak informace není úplná, protože neznáme, jaké to jsou podmínky tohoto přechodu.

K přechodům se vyznačuje také událost (trigger), který je pro tento přechodu příznačná a spouští přechod. Pro vyjádření podmínky, za jakých k přechodu dojde a za jakých nikoliv, se zavádí tzv. guard, což je výraz typu boolean, který když nabude hodnoty true, tak k danému přechodu dojde (v opačném případě nikoliv). Guard nelze zaměňovat s událostí, která přechod vyvolává, guard má význam omezující podmínky typu boolean.

Také lze vyznačit aktivitu, která změnu stavu doprovází:

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!



Pomocí této informace se již plně projevuje determinismus: Pokud zadáme také událost E a podmínku pro guard, což vede k překlopení ze stavu X do stavu Y, tak pokud se objekt nachází v X a nastane podmínka tak **nutně** dojde ke změně stavu z X do Z za projevené aktivity A. Předešlý obrázek se tedy již nečte „objekt se může ze stavu X dostat do stavu Y", ale „objekt ve stavu A při události E za splnění podmínky guardu přejde do stavu Y" (což reprezentuje úplný determinismus).

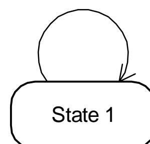
Pro studium samotných přechodů je třeba uvést, že stavový diagram nebere přechody a aktivity s nimi spojené za předmět svého zájmu a studia. Tyto přechody chápe jakoby nastávaly okamžitě, tedy čas strávený překlápěním a co se přitom děje, není pro stavový diagram zajímavým. Je zřejmé, že v některých případech i toto překlápění může být pro řešení systému podstatné. Tomuto problému se věnuje jiný typ diagramu, tzv. Activity diagram.

Stavové stroje zavedené v UML (ve vazbě na třídy) jsou deterministické. V diagramu se vyznačuje také počáteční a konečný stav tohoto diagramu. Visual prvkem počátečního stavu v diagramu (tj. stavu, odkud se na diagramu začíná spouštět stavový stroj) je plné kolečko. Visual prvkem konečného stavu v diagramu (může jich být několik) je plné kolečko s kružnicí:

obrázek 70 Počáteční stav diagramu (odkud začít číst diagram v přechodech) a konečný stav v diagramu (za kterým již nepokračuje žádný přechod)

Přechod do téhož stavu

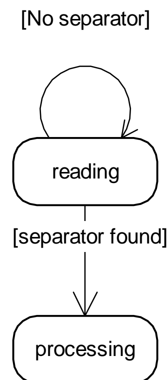
V UML je povoleno zavést transition jako přechod mezi dvěma stavy, kdy výchozí stav a konečný stav jsou shodnými:



obrázek 71 Přechod do téhož stavu

Otázka je, kdy se tento zápis vlastně použije a hlavně proč by se měl vlastně použít? Prvek přechodu do téhož stavu použijeme tehdy, když chceme zdůraznit tu skutečnost, že sice dojde k události, že sice dojde k aktivitě, ale stav objektu se nezmění.

Příklad: Word procesor vyhledává v syntaxi slova. Pokud nenalezne celé slovo (oddělovač) pokračuje ve stavu „čteme dále". Pokud bude nalezen separátor, překlopí se do stavu zpracování slova:



Dotaz pro čtenáře

V příkladu se zaměstnanci a firmou (viz objekty osoba 1, osoba 2, osoba 3) existuje také jedna situace, která by stála za to zapsat ji do diagramu a vyjadřuje přechod do téhož stavu. Která to je?

O jedné zvláštní chybě při určování stavů objektů

Setkal jsem při modelování s jednou zvláštní chybou, která vyplývala z použití modelu tzv. business procesů (což jsou modely, které nejsou součástí UML) a z chybného pochopení smyslu analýzy.

Podle tohoto přístupu se součástí analýzy stává i analýza problémové domény, tedy jako analýza se chápe i to, „jak to v daném podniku vlastně chodí“.

Z hlediska modelování samotného informačního systému pomocí UML je třeba upozornit na jednu sice triviální, avšak mnohdy zanedbávanou skutečnost:

Jakýkoliv model informačního systému v UML budeme chápat jako model daného řešeného informačního systému a ničeho jiného!

Problém je v tom, že UML nám umožňuje modelovat i jiné systémy, než pouze informační systém. Například takovýmto předmětem modelování může být i sám podnik, do kterého je informační sytém dosazen.

Při záměně předmětu modelování můžeme narazit na určité problémy. Samozřejmě tato zásada neznamená, že nebudeme vytvářet dokumenty analyzující problematiku i mimo systém. Tato analýza však není přímo součástí informačního systému jako takového, i když k ní má přímý vztah.

Příklad: Uvedli jsme situaci, kdy bylo třeba zpracovat v bankovním informačním systému směnky a byl získán stavový diagram směnek v bance, tedy „jak to chodí se směnkami v bance“. To však určitě není stavový diagram „jak budou chodit směnky v informačním systému“.

Analýza samotné logisitiky problému a podniku je velmi prospěšná pro tvorbu analytických modelů informačního systému. Následující formulace vystihuje povahu těchto analytických dokumentů:

analytické dokumenty problémové domény jsou dobrým zdrojem pro analytické modely samotného systému

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Doporučuji, aby tyto dokumenty analýzy problémové domény vytvořily ve firmě svou zvláštní oblast zdroje informací. Modely systému je používají například jako přílohy, odkazují se na ně apod. Je však rozdíl mezi analytickým modelem informačního systému a těmito dokumenty. Uvedená chyba souvisí s nepochopením rozdílné pozice těchto dokumentů a modelů. Může vést k zajímavým chybám:

Příklad: Setkal jsem se ve stavovém modelu daného informačního systému s počátečním stavem Faktury, která „byla vystavena, avšak nikoliv ještě zavedena do systému". Je pochopitelné, že si tvůrci tohoto modelu nedovedli dále s tímto zvláštním analytickým stavem Faktury poradit, protože z hlediska chodu uvnitř systému (a tedy jeho modelu) se jedná o protimluv. Faktura v systému začne žít tím, že ji do informačního systému zavedeme a tehdy se nachází v nějakém výchozím stavu. Stavy entit v informačním systému jsou platné v rámci tohoto systému a nemusí to být obrazem jedna ku jedné vůči všem stavům v oblasti dané reality.

Generalizace stavů a kompozitní stavy

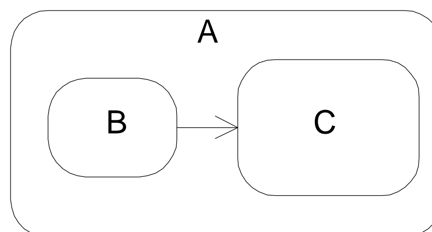
V mnoha případech potřebujeme ve stavovém diagramu vyjádřit tu skutečnost, že daný stav, který znázorňujeme, je jakýmsi „superstavem" pro jiné stavy, které jsou pod tímto stavem „uschovány". Například faktura se v prvním přiblížení nachází pouze ve dvou „superstavech":

- ve stavu *„Připravovaná k odeslání"*
- a poté ve stavu *„Odeslaná"*.

Další stavy jsou uschovány uvnitř těchto „superstavů". Například „připravovaná k odeslání" znamená

- editovaná,
- ukončena editace,
- k odsouhlasení,
- odsouhlasena,

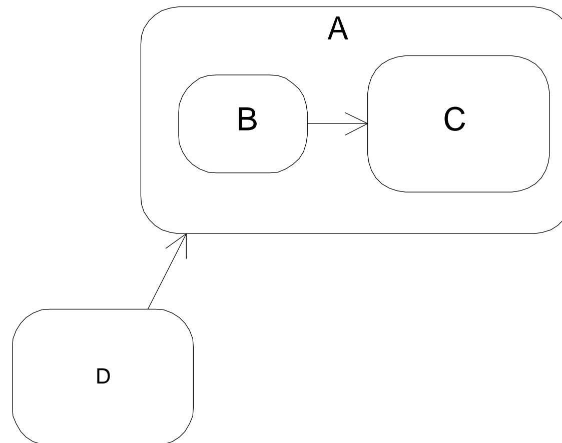
přitom do druhého superstavu se může dostat až ze stavu odsouhlasena. Ze stavu odsouhlasena může také (jinou událostí) přejít například zpět do předešlých stavů.



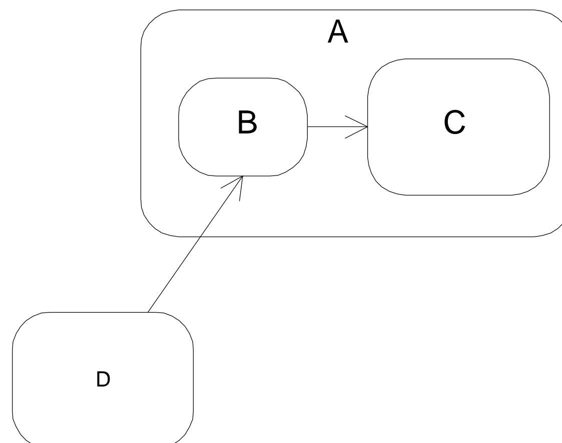
obrázek 72 Superstav A

Podle tohoto diagramu, pokud se objekt nachází ve stavu B nebo C, je to chápáno tak, že se současně nachází ve stavu A, který je jejich zobecněním.

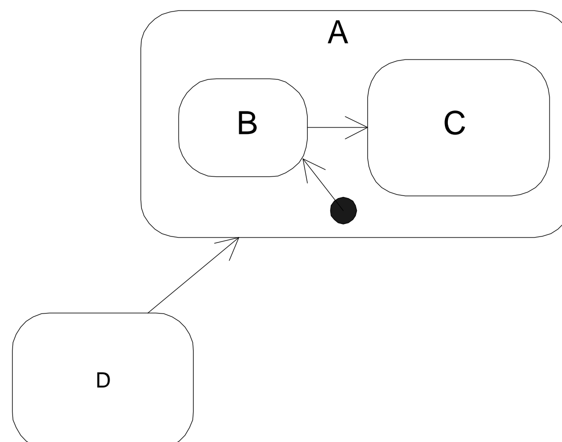
Následující syntaxe není přesná:



protože není zřejmé, zda se ze stavu D objekt překlopí do stavu B nebo C. Správně buď takto:

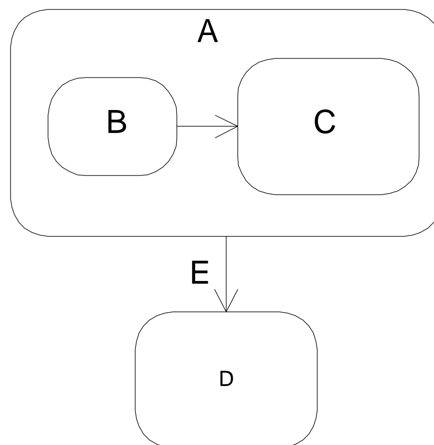


anebo pomocí počátečního stavu takto:



Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Avšak opačně je dovolena tato syntaxe:



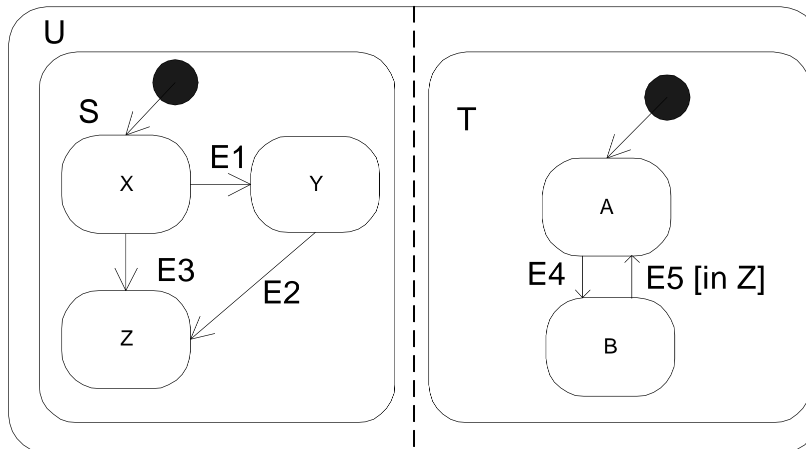
která znamená, že pokud se objekt nachází ve stavu A, tj. v jednom z podstavů B nebo C, a nastane událost E, objekt se překlopí do stavu D.

Agregace stavů

Při modelování stavů může nastat situace, kdy se jinak jednoduché modely stávají stále více složitějšími díky tomu, že dochází ke kombinaci různých jinak takřka nezávislých stavových strojů.

Jeden objekt viděný v různém kontextu může svými stavy nechat vzniknout několika různým stavovým strojům a celkové výsledné stavy, ve kterých se objekt může nacházet, se potom stávají kombinacemi těchto základních stavů z různých stavových strojů. Na těchto kombinacích různých stavových strojů by nebylo nic složitého, kdyby tyto dva okruhy stavů objektu byly vždy úplně nezávislé. Potom by byl výsledný stav chápán jako prostý kartézský součin (A x B) těchto dvou stavových strojů podle pravidla „každý stav z jednoho stroje s každým stavem z druhého stroje“. Tím by vznikla úplná množina dvojic všech možných stavů objektu. V mnoha případech však dochází k interakci díky omezujícím podmínkám mezi těmito dvěma stroji, například jsou zakázané přechody v jednom stroji, pokud druhý stroj je v určitém stavu apod. Již nelze hovořit o dvou úplně nezávislých strojích. Pro vyjádření kombinací stavů a současné interakci dvou strojů je dobrou pomůckou použít agregaci stavů.

Agregace stavů znamená použít dva stavy „vedle sebe“, které se agregují do jednoho výsledného stavu chápaného jako kombinace těchto dvou stavů a současně se mohou vyznačit omezení v obou jednotlivých stavech. Agregace se vyznačuje vložením agregovaných stavů do stavu při vyznačení přerušované čáry mezi nimi:



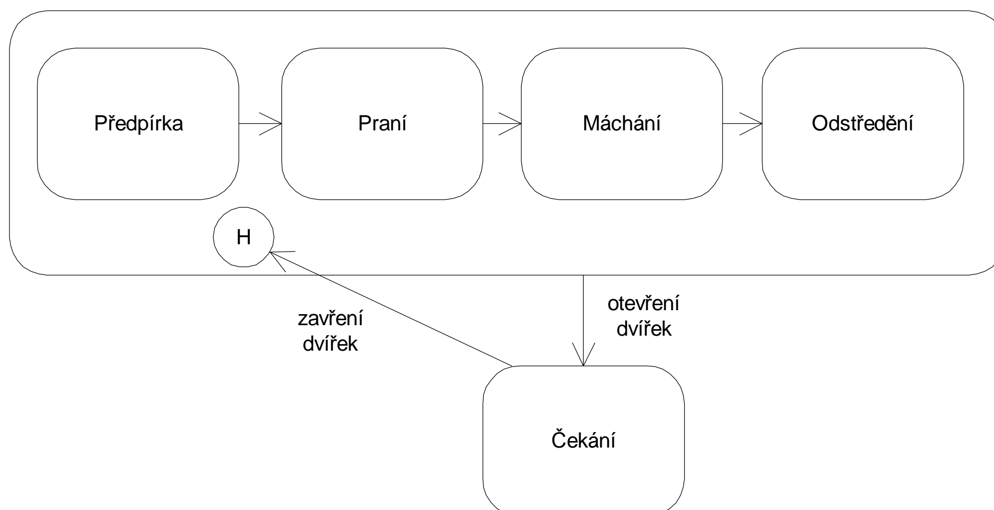
Na obrázku je nejvyšším stavem stav U. Ten je agregován ze dvou stavů označených jako S a T. Stavy S a T jsou zobecněním stavů po řadě X, Y, Z a A, B. Uvnitř stavu S je stavový stroj s počátkem ve stavu X, ve stavu T je počátečním stavem A.

Všimněte si podmínky u přechodu s událostí E5 mezi B a A, kde je omezení ve tvaru guardu *[in Z]*. Toto omezení značí, že k tomuto přechodu dojde pouze tehdy, jestliže ve vedlejším stavu S se stroj nachází ve stavu Z.

Model element History

Při opuštění generalizujícího stavu je mnohdy žádoucí znát ten podstav, ze kterého byl tento stav opuštěn. K tomu slouží element History, jehož visual prvkem je kolečko. History tak slouží jako „buffer stavu", tj. jako dočasná proměnná pro udržení historie, odkud se stav opustil.

Klasickým příkladem uváděným v literatuře je stavový diagram pračky:



V tomto typu pračky lze v kterémkoliv stavu otevřít dvířka,čímž se pračka zastaví. Pokud dvířka opět zavřeme, tak by pračka měla pokračovat tím stavem, který byl při otevření dvířek opuštěn. Bez použití elementu History bychom tuto informaci nemohli do diagramu zapsat.

Activity model, model aktivit

Jedním z dalších modelů UML je tzv. Activity model neboli model aktivit. Tento model se nejčastěji používá v těchto situacích:

- pro modelování systémů pracujících v reálném čase, například řídící systémy, systémy pro řízení technologických procesů (různé zahřívání, spouštění, napouštění cisteren atd.). Velmi často je activity diagram použit pro vyjádření problematiky paralelních procesů (multithreadové aplikace apod.), vyžadující synchronizaci procesů mezi sebou.
- pro vyjádření běhu a procesů, aktivit v analýze v situacích, kdy není ještě zřejmé, jaké jsou kompetence objektů v systému, ale znají se procesy v systému bez kompetencí.

Pokud se vrátíme ke stavovému diagramu, tak u něj samotné přechody nejsou předmětem jeho zájmu, tedy považují se za okamžité, jako by se proces sledování systému v době přechodu nesledoval a „zapnul se" až při překlopení do stavu.

Oproti tomu Activity diagram se zaměřuje právě na tyto přechody mezi stavy. Základním prvkem modelu je tzv. aktivita, angl. activity.

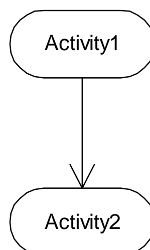
Model element Activity

Jedna aktivita se povinně nevztahuje k objektu, ale obecně k systému (bez nutného určení kompetence), tedy jedna vyznačená aktivita systému může být v konečném důsledku realizována několika objekty. To je velká výhoda použití tohoto diagramu zejména v raném stadiu analýzy.

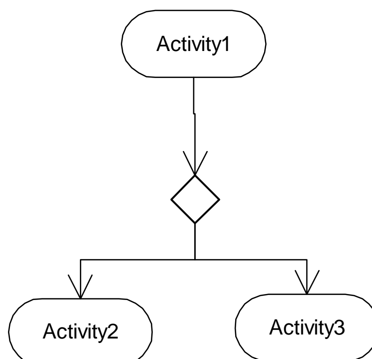
Visual elementem aktivity je ovál (dvě vodorovné rovnoběžné úsečky stejné velikosti spojené půlkružnicemi na koncích):

Model element Přechod mezi aktivitami

Přechod z jedné aktivity do druhé se vyznačuje šipkou:



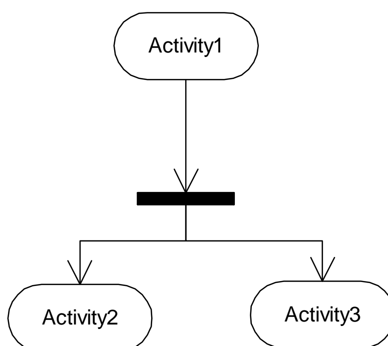
Při rozhodování je možné použít větvení a rozhodování:



obrázek 73 Element rozhodování vede k větvení podle podmínky se vykoná jedna z aktivit

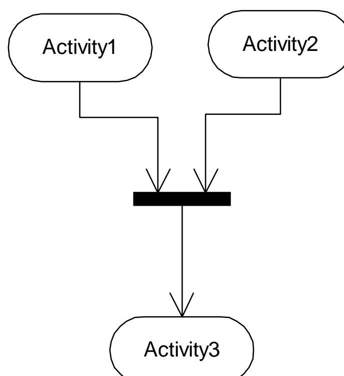
Model element horizontální čára v modelu aktivit

Pro vytvoření a zobrazení paralelních aktivit se používá horizontální čára a to jak pro spuštění paralelních aktivit:



Poznámka: Oproti předchozímu obrázku s větvením zde Activity 2 a Activity 3 běží paralelně a nikoliv exkluzivně.

Tak pro jejich synchronizaci:



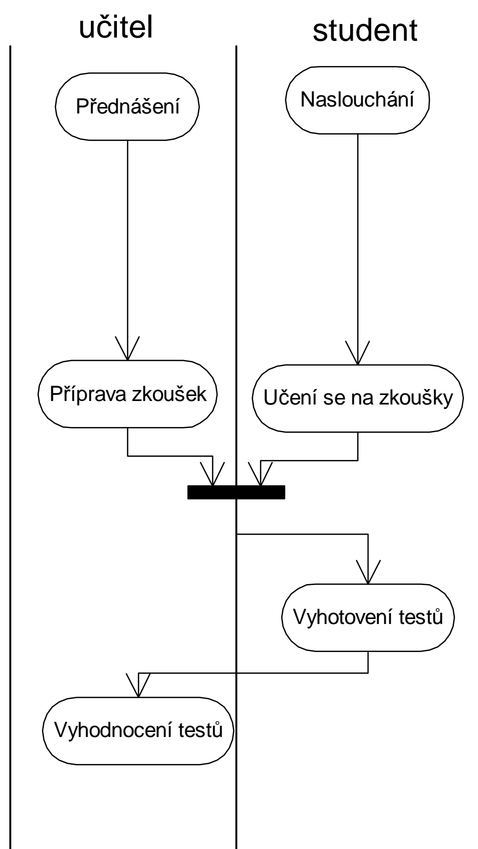
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Model element swim-lines v modelu aktivit

Pro vyjádření určitých kompetencí, které ještě nemusí být reprezentovány objekty, ale pouze „oblastmi a doménami kompetencí“, lze použít elementy zvané jako swim-lines. V překladu doslova „plavecké dráhy v bazénu“. Každá aktivita se vymezí do určité takové „plavecké dráhy“ a tím se vyznačí, pod jakou kompetenci aktivita spadá. Visual elementem swim-line je svislá čára.

Příklad

Swim-lines vymezují kompetence učitele a studenta. Navíc všimněte si, že se nejedná o model informačního systému, ale o analýzu chodu aktivit v procesech v daném „podniku“ (např. na univerzitě)



Pokud tyto „plavecké dráhy“ přejdou v konkrétní objekty, tak jednotlivé aktivity přecházejí v operace objektů. Tímto model aktivit může v limitním případě přejít až do sekvenčního modelu, potom jednotlivé swim-lines jsou reprezentovány přímo objekty a jejich aktivitami .

Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

Komponentní model

Co je to komponenta obecně?

Komponenta je uzavřená znovupoužitelná část systému, pomocí které se provádí fyzické členění systému. V UML se zavádí pojem komponenta ve dvojím významu:

- binary component (binární komponenta)
- source component (komponenta zdrojového kódu)

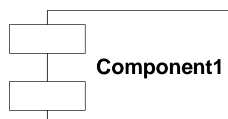
V obou případech se jedná z hlediska UML o komponentu, i když většinou se v literatuře chápe komponenta pouze ve smyslu binary component. Ve většině případech budeme mít také v této knize na mysli komponentu ve smyslu binary component, pokud nebude uvedeno jinak. Některá pravidla však platí pro oba typy komponent, jak pro binary, tak pro source komponentu.

Proč komponenty zavádět?

Základním požadavkem, proč se komponentní technologie zavádí je logické a následně i fyzické rozčlenění systému na menší celky provázané zpětně v systému. Toto logické a následně i fyzické rozdělení má několik velmi příznivých důsledků, z nichž mezi hlavní patří zvýšení přehlednosti systému, jeho snazší vývoj, vyšší stupeň re-use, lepší možnost dokumentace, u binary komponent jazyková nezávislost komponent, re-use na vyšších úrovních mezi projekty, firmami atd.

Model element Komponenta

Vyjádření samotného modelu je ve své podstatě velmi jednoduché, avšak navrhnout správný model je mnohem složitější. Komponenta má následující visual element



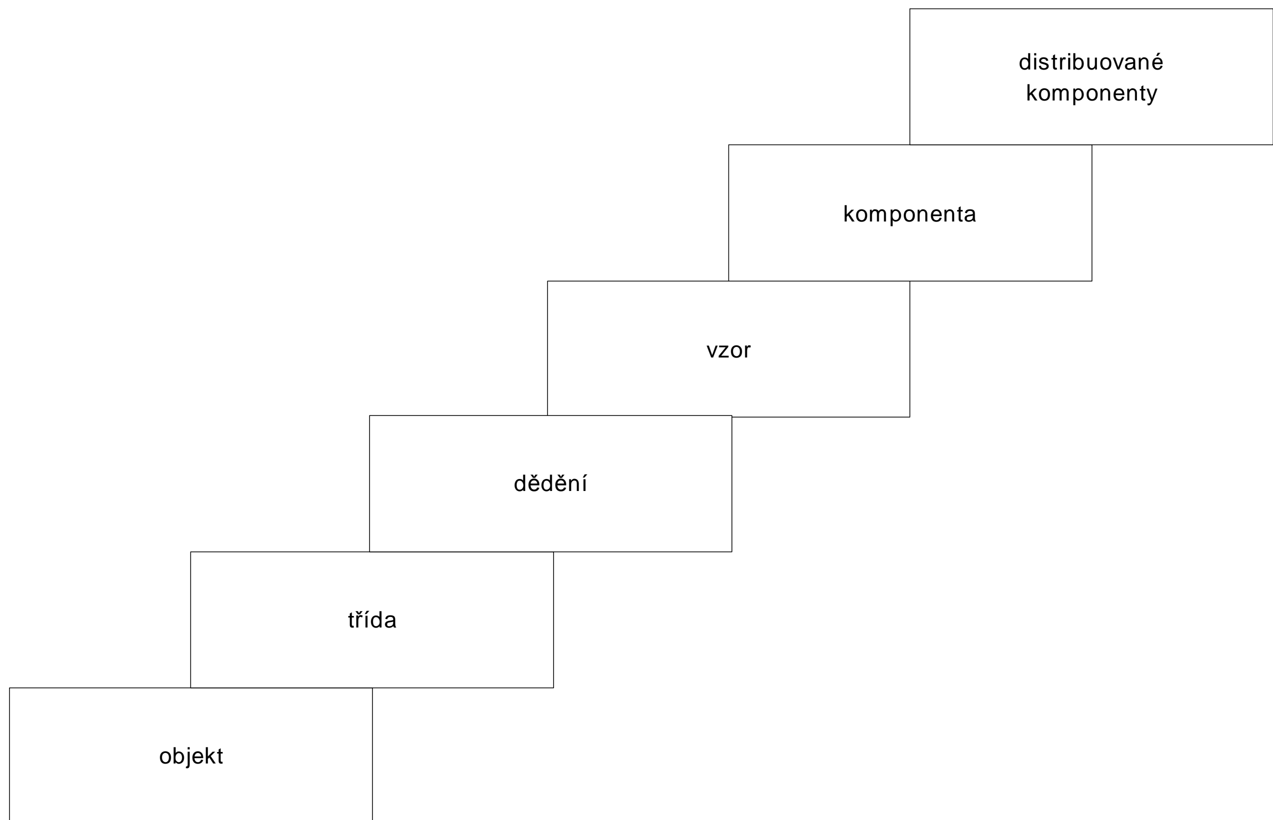
Vysvětlení komponent pomocí stupně re-use v softwaru

Kromě re-use zdrojového kódu existuje i jiný typ re-use kódu. Moderní technologie DLL knihoven obecně ukázala možnost vytvořit nový typ re-use kódu a to na binární úrovni. Existuje možnost přilinkovat k hotovému softwaru jiný hotový kus binárního balíku a přidat tak k systému pomocí DLL funkcí nové procedury na binární úrovni. Avšak přístup DLL knihoven přilinkováním funkcí neodpovídá OOP, protože linkováním můžeme přidávat pouze funkce.

Obecně však možnost přilinkovat binární balíky k systému vede k vytvoření pomyslného žebříčku re-use ve tvorbě softwaru. U objektově orientovaného programování v komponentním prostředí můžeme rozlišit následující stupně re-use kódu:

Stupně re-use v nejmodernější tvorbě SW

Následující obrázek ukazuje jednotlivé stupně re-use v softwaru:



obrázek 74 Stupně re-use v sw

Na obrázku jsou znázorněny jednotlivé stupně re-use ve tvorbě software, které vedou až k nejvyššímu stupni, ke komponentám a distribuovaným komponentám. Každá z těchto úrovní prolamuje pomyslnou hranici bránící re-use

První schod: re-use pomocí instancí objektů znovu volaných

Jedná se o nejtriviálnější re-use, tak triviální, že si jej ani neuvědomujeme. Představme si, že máme v systému zavedenu instanci objektu a tento objekt má definovánu a korektně zavedenu nějakou metodu. Můžeme opakovaně posílat zprávu objektu a tím metodu vyvolat. Objekt je tedy nositelem re-use v tom smyslu, že jedna a táž metoda nemusí být znovu a znovu definována pro nové volání objektu, jakmile je objekt v systému vytvořen.

Poznámka: Musím v této souvislosti připomenout jednu zásadní věc z OOP: opakované volání objektu stejnou zprávou nemusí dát stejný výsledek, protože objekt je nositelem svých vnitřních stavů. Průběh jedné a téže metody může být odlišný od předešlého průběhu téže metody, protože může dojít ke změně stavu objektu (např. změna hodnoty atributu daného objektu anebo změna stavu vnitřního objektu).

Druhý schod: re-use pomocí třídy

Třída umožňuje sloučit opakovanou definici instancí objektů do jedné definice, do definice třídy. Pomocí definice třídy se definují objekty, které z této třídy pocházejí. Znamená to, že pokud zavádíme nové a nové instance stejných vlastností, nemusíme každou z nich definovat znovu a znovu a postačí jedna definice třídy.

Použitím třídy je prolomena hranice mezi objekty jako instancemi. Pro druhou instanci nemusíme „samostatně“ zavést novou další definici.

Třetí schod: re-use pomocí dědění

Při definici tříd může nastat situace, kdy v definici jedné třídy se opakují prvky definice z jiné třídy. Tato redundance definice ve třídách je nežádoucí. V tom případě se zavádí re-use pomocí dědění. Společné definice ve třídách se vyjmou do třídy předka a zpětně navážou do původních definic tříd pomocí dědění.

Bariéra, která je zde prolomena, je mezi třídami. Pro novou třídu nemusíme již „samostatně“ vytvořit novou další definici, ale může znovupoužít již existujících definic tříd.

Mezanin na schodech: re-use pomocí vzoru

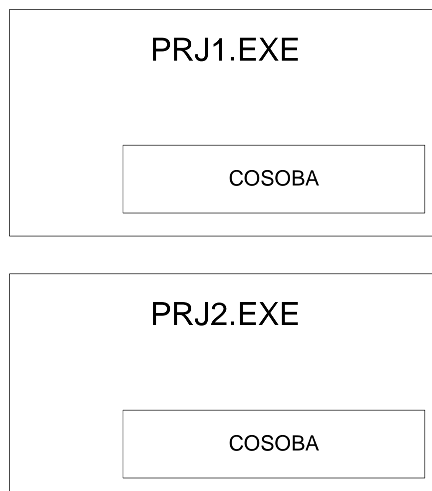
Dalším typem re-use je tzv. vzor. Vzor lze chápat jako použití řešení v různých kontextech podle určitých pravidel, tj. vzoru. Teorie vzoru je ve tvorbě SW poměrně dost složitá, protože se díky abstrakci tvorby SW mnohdy těžko vyjadřuje. Samotná problematika vzorů není předmětem této knihy a vydá na celou knihu.

Čtvrtý schod: binární re-use na lokále

Vraťme se k třetímu schodu, kde jsme zavedli re-use pomocí dědění. Je nyní otázkou, jaká je další bariéra, která brání re-use? Jaký je ten další schod? Zkusme nyní tuto bariéru nalézt.

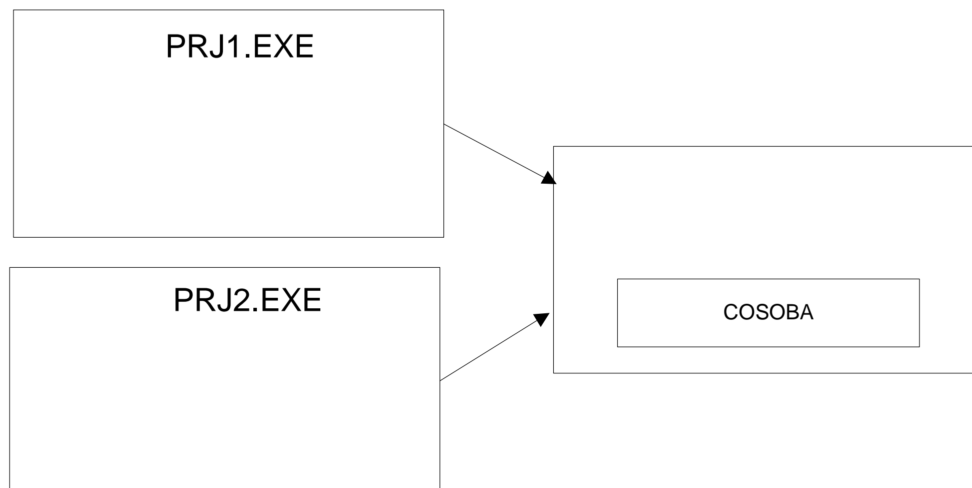
Představme si, že vytváříme projekt a v něm zavedeme třídu osob a nazvěme ji COsoba. V daném projektu tuto třídu použijeme. Zkompilujeme projekt a vznikne spustitelný soubor PRJ1.EXE. Můžeme si představit, že uvnitř zkompilovaného kódu existuje nějak zabudovaná zkompilovaná třída COsoba, kterou samozřejmě po kompilaci již nevidíme, ale její stopy bychom ve zkompilovaném kódu za pomoci hackera určitě nalezli.

Pokračujme v úvaze: Budeme vyvíjet druhý projekt a zjistíme, že v něm budeme potřebovat také třídu COsoba. Proto zavedeme řízenou knihovnu zdrojového kódu a zdrojový kód třídy COsoba budeme sdílet u obou projektů. Po kompilaci druhého projektu vznikne druhý soubor, nazvěme jej PRJ2.EXE. Také uvnitř tohoto souboru si představujeme existenci „zkompilované třídy COsoba“. Znamená to, že zdrojový kód je sice sdílen, ale po kompilaci již nikoliv, třída se ve svých stopách objevuje v obou souborech:



Uvedený obrázek nám připomíná již známou situaci z obecného principu re-use. Něco se opakuje a chtěli bychom to sdílet. Tímto požadavkem na sdílení je také současně odhalena ta bariéra, kterou bychom u tohoto schodu re-use chtěli prorazit: Požadujeme po dané třídě, aby byla „sdílena“ nikoliv zdrojem, ale „nějak“ mezi útvary PRJ1.EXE a PRJ2.EXE. Hranicí jsou tedy hranice binárního souboru.

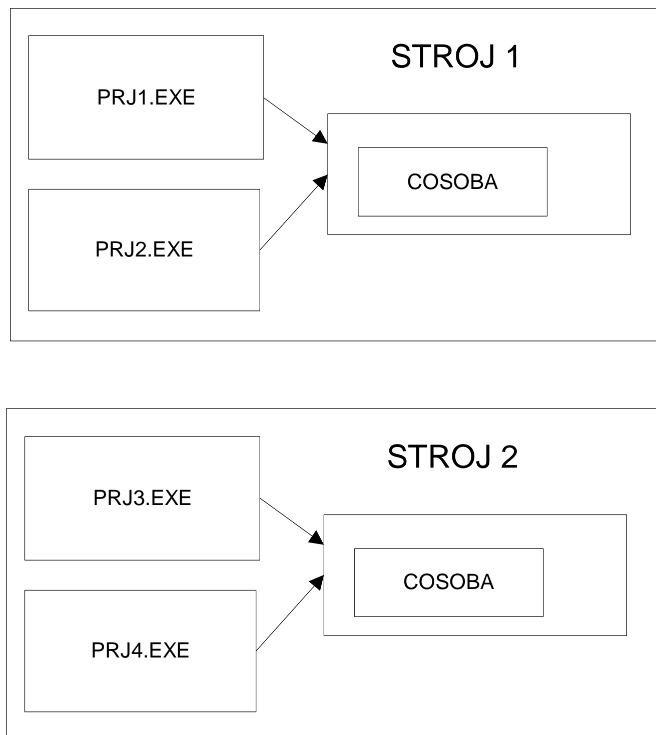
Proražením této hranice vznikne re-use na binární úrovni, tak zvaný binary re-use:



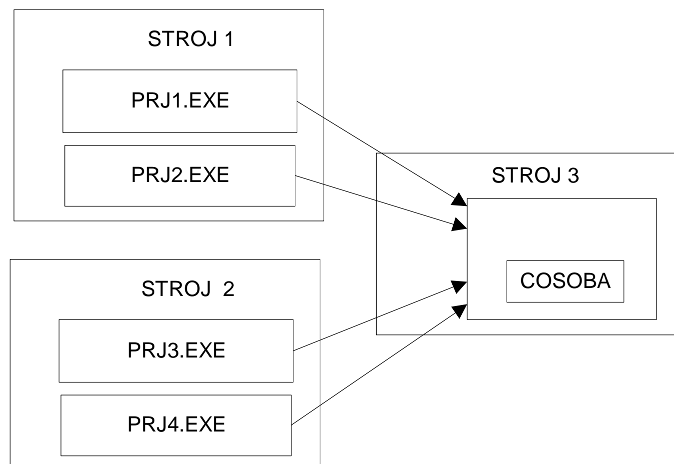
Připojení již hotových binárních balíků objektů je základem tzv. komponentní technologie. Systém se již neskládá z monolitických velkých „souborů“, ale z komponent.

Pátý schod: binární re-use mezi stroji

Stojíme na čtvrtém schodu re-use. Je otázkou, jaká je další bariéra pro re-use? Jak napovídá název odstavce, další bariérou jsou hranice mezi stroji. Ukažme si opět obrázek té situace, která vyžaduje re-use:



Všimněme si, že na předešlém obrázku je čtvrtého schodu re-use, tj. binary re-use, úspěšně dosaženo: Na každém stroji dochází k binárnímu re-use, avšak mezi stroji již re-use není, „něco se mezi stroji opakuje“. Pokud nám to technologie dovolí, mohli bychom od tohoto stavu bez re-use přejít do stavu vyššího, ale musíme prorazit hranici mezi stroji, například takto:



Komponentní technologie COM a CORBA

Existují dvě základní komponentní technologie běžící přímo jako „doplněk operačních systémů“, a to CORBA a COM. CORBA se většinou používá na platformách UNIX a platformách příbuzných, zatímco COM je implementován pod technologiemi MS Windows. Kromě toho nezapomeňme na komponenty tvořené v prostředí Java a běžící speciálně v tomto prostředí.

V dalších kapitolách si budeme vysvětlovat základy komponentní technologie obecně a současně jako ukázky zvolíme technologii COM.

Poznámka: Důvodem této volby není ani tak vyjádření „lepší nebo horší“, ale jedná se o historické důvody. Jako pracovník jsem měl možnost pracovat pouze ve firmách, které používaly MS NT technologie a poté COM

Interface objektu a jeho binární podoba

Každý objekt je schopen přijmout zprávu a reagovat na ně spouštěním odpovídající metody. Existuje tedy převodník mezi zprávou a metodou a nazývá se protokol zpráv. Protokol zpráv se skládá ze dvou částí: Na jedné straně existuje množina zpráv a na straně druhé množina metod, z nichž každá je přiřazena k dané zprávě. Právě množina zpráv, tj. vstupní brána do objektu, se nazývá *interface objektu.* Tato definice je obecná v OOP (tedy zde ještě není řeč o komponentách, ale o obecném OOP).

Problém propojení binárních objektů vede k problému binárního interfacu objektu, tj. k otázce, jak obecný interface implementovat binárně.

Nejjednodušší způsob, jak binárně implementovat interface, je vytvořit jej jako sadu ukazatelů na metody. Metody se nevolají přímo, ale přes ukazatel na ni a teprve hodnota na místě ukazatele udává, která metoda se má volat. Získat interface znamená získat ukazatel na řadu ukazatelů na metody.

Takto je vcelku elegantně odizolováno přímé volání metod.

Princip fungování komponenty jako knihovny

Komponentní technologii si vysvětlíme velmi jednoduše na určitých základních pojmech. Je třeba poznamenat, že jsem se mnohdy setkal s velkými chybami při chápání komponentní technologie a z toho důvodu použiji za účelem srozumitelnosti k vysvětlení některé „v literatuře méně obvyklé pojmy“ a poté je dáme do kontextu všeobecného používání. Tento postup se mi velmi osvědčil.

Mnohdy dochází k nedorozumění v komponentní technologii při chápání obecného vztahu „typ něčeho a instance tohoto typu“. Určité úskalí v této technologii spočívá v tom, že v ní existuje hned několik takovýchto vztahů mezi typem a instancí a zanedbání tohoto vztahu vede k chybným úvahám.

Jako první takovýto vztah „typu a instance“ je dvojice „komponenta a instance komponenty“. Instance komponenty je již žijící útvar, který plní určitou funkcionalitu. Na druhé straně komponenta je to, z čeho tato instance vznikla. Pokud chcete používat nějakou instanci komponenty, musíte si nejprve na stroj resp. na síť nainstalovat komponentu, která vám teprve tuto instanci poskytne.

Druhý takovýto vztah je třída a objekt. Tento vztah již známe, zde jej musíme uvést do kontextu komponenty. Jedna komponenta (tj. nikoliv instance) obsahuje v sobě několik tříd. Můžeme tedy požádat komponentu, aby vydala (zrodila) objekt z jedné ze svých tříd, kterou tato komponenta obsahuje a která má toto rození objektů povoleno (class je „creatable“). Získáme tak daný objekt z dané třídy. Tento objekt není ničím jiným, než již uvedenou instancí komponenty. Tento vztah je zajímavý v tom, že přilinkováním jedné komponenty získáme „na výběr“ N tříd a můžeme tedy z jedné komponenty nechat zrodit objekty jako instance komponent různých vlastností. Jedna komponenta může tedy vydávat několikero různých objektů jako své instance.

Další vztah, který se velmi často opomíjí, je vztah typ interfacu a instance interfacu. (Jak víme, interface je u objektu chápán jako množina zpráv v jeho protokolu zpráv) Většinou se tyto dva pojmy nerozlišují. Jak pracuje mechanismus vztahu typ interfacu a instance interfacu?

Představme si, že máme k dispozici komponentu a ta obsahuje N tříd, ze kterých můžeme nechat zrodit objekty neboli instance komponenty. Ke každé této třídě v komponentě je přiřazen jeden (nebo více) typů interfacu. Pokud necháme zrodit objekt z dané třídy, tak při tomto zrodu komponenta

tomuto objektu vytvoří instanci interfacu (nebo několik) daného typu interfacu a přes tuto instanci interfacu může klient objektu daný objekt volat, protože klient dostane ukazatel na tuto instanci interfacu.

Typ interfacu se tak stává samostatnou identifikovatelnou entitou, která se provazuje na třídu ve vztahu „třída versus typ interfacu 1 : N“. Znamená to, že jeden objekt zrozený z dané třídy může mít N instancí interfacu, tedy je klientovi poskytnuto N ukazatelů na tyto instance interfacu.

Otázkou je proč by měl mít objekt N interfaců? Tato skutečnost souvisí s pojmem násobnost interfacu.

Násobnost interfaců v komponentní technologii

Věnujme se teď problému dědění ve zdrojovém kódu: Představme si třídu a její zavedený interface, tj. množinu zpráv. Provedeme dědění do druhé třídy. Tato druhá třída pomocí dědění zdědí všechny vlastnosti třídy předka a tedy zdědí také interface definovaný v předkovi. Tím také objekty z třídy potomka vlastní interface definovaný třídou předka. Dědění tedy jednoduchým způsobem zavádí re-use interfaců jako odvozený re-use od dědění, protože dědění provádí „totální“ re-use.

U binárních objektů neexistuje takováto obdoba dědění, protože dědění spadá do oblasti zdrojového kódu a pro dědění musí být zdrojový kód k dispozici. Zatím není uspokojivě řešen problém dědění na binární úrovni ani teoreticky a existuje pouze dědění na úrovni zdrojového kódu.

U binárních objektů se proto re-use interfaců vyřešil nikoliv děděním, ale tzv.násobností interfaců. Existuje operace, která umožňuje přiřadit interface k danému objektu, což se nazývá implementací interfacu. Toto přiřazení vlastně nahrazuje podobnou operaci, jako je dědění, čímž umožňuje provést re-use interfaců. Typ interfacu může být definován pouze jednou a pokud se má vyskytovat u různých typů objektů, lze jej implementovat do těchto objektů.

Jaký je rozdíl mezi děděním a implementací interfaců

Mezi implementací a děděním existuje určitá podobnost. Pokud provedeme dědění mezi třídami, provádíme přiřazení interfaců z vyšších tříd předků k nižším třídám potomka a tedy objekty z třídy potomka se chovají, jako by měly násobné interfacy (zděděné od několika předků).

Existují však některé základní rozdíly, které je třeba si uvědomit.

1. U dědění se „zdědí vše“, tj. nikoliv pouze externí chování objektu (interface), ale také vnitřní struktura. To je možné díky přístupnosti zdrojového kódu. V této souvislosti můžeme hovořit o tzv. *„white-box“ přístupu* (vše viditelné). U implementace interfaců se provede re-use pouze externích vlastností objektu, tj. množiny zpráv. Vnitřní chování objektu se nepřevede a zůstává pro objekty specifické (je třeba zavést metody a vnitřní struktury zvlášť). Hovoříme o „black-box“ přístupu (vnitřek pro re-use není viditelný). Tím je tento „black-box“ re-use pouze částečný.
2. Dědění na rozdíl od implementace je tranzitivní operací. Pokud A dědí z B a B dědí z C, potom A dědí z C. U implementace toto neplatí, operace je čistě binární a nepřenáší se dále. To se může projevit jako nevýhoda u hlubokých stromů generalizace a specializace, kdy je třeba implementovat hodně interfaců z celého stromu odshora až dolů. Při dědění se naopak situace velmi zjednoduší (např. stačí mít jednoho předka a tím se dědí celý strom)

Návrh komponent v informačním systému

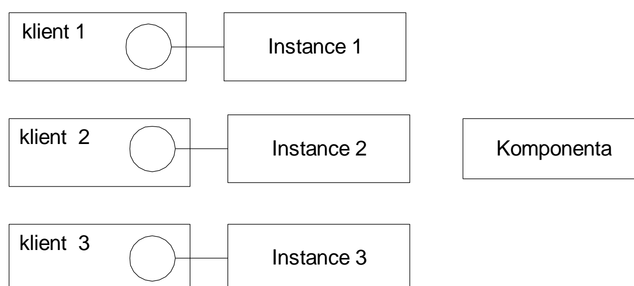
Z hlediska vztahu instance komponenty a jejího klienta existují v libovolné komponentní technologii dva základní typy komponent:

Instance komponent s izolovanými klienty

Pro tento typ komponenty příznačné, že každý klient, který chce používat instanci komponenty, si vybudí novou „svou a pouze svou izolovanou“ instanci ve svém vlastní viditelnosti a nikde jinde. Komponenta se chová jako objektová přilinkovaná knihovna ke svému klientovi. Z podstaty věci je vyloučeno, aby její instance byla viděna jiným klientem, než tím, který o ni požádal. Sama

komponenta (nikoliv instance) může být nainstalována v systému „jednou“ a je tedy sdílená, nikoliv však její instance, která je pro každého klienta „zvlášť“ v jeho prostoru viditelnosti. Sdílení tohoto druhu znamená re-use kódu v tom smyslu, že nemusíme komponentu opakovaně instalovat.

Přilinkovat tuto komponentu znamená následující: Díky komponentně jsou poskytnuty třídy, ze kterých můžeme vytvořit objekty, které jsou pouze ve viditelnosti daného klienta tvořeny jako instance komponenty. Komponenta se tímto chová stejně, jako bychom si přilinkovali klasický zdrojový kód objektového modulu, pouze je v binárním tvaru.

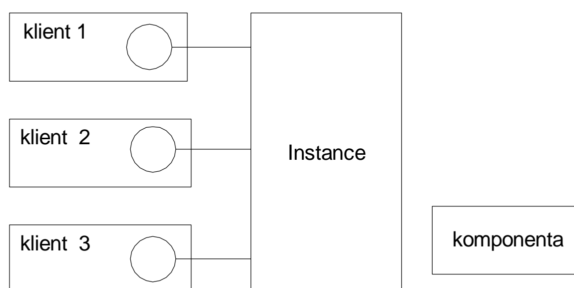


obrázek 75 Komponenta vytvořená v módu izolovaných klientů

Klasickým příkladem takovéto komponenty je například komponenta technologie ADO s poskytnutými třídami a odpovídajícími interfacy.

Instance sdílená klienty

Existuje druhý mód, ve kterém lze komponentu vytvořit a to v módu sdíleném. Pokud klient požádá o instanci komponenty, je tomuto klientovi poskytnut interface na již existující instanci komponenty. Dva a více klientů tak sdílí tutéž instanci.



obrázek 76 Komponenta vytvořená v módu sdílené instance

Sdílená instance komponenta funguje jako skutečný objektový server. „Stojí“ v systému jako objektový stroj s objekty připravenými k použití a každý nový klient se může k tomuto stroji připojit a používat jej. Přitom tento server může být instalován na síti a sdílen klienty z různých strojů. Objektové struktury v instanci komponenty se jednou naplněné stávají všeobecně sdílenými a to „přímo a naživo“. Klienti mají přístup k objektům, které již někde v systému žijí a mohou tak pro klienty přímo pracovat.

Klasickým příkladem sdílené komponenty je například Queue Server.

Vytvořit aplikační server jako sdílenou komponentu, tj. objektový server, je velmi lákavé, ***ale může se jednat o velmi nebezpečný krok***. Pokud zvolíte sdílenou komponentu, potom musíte očekávat tyto úkoly, které jste dosud neřešili:

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

- Řešíte zamykání objektů přímo v aplikační úrovni. Protože klienti instance komponenty přistupují k této komponentě kdykoliv pro editaci a změny, musíte vyřešit analytický problém zamykání objektů proti možné kolizi. Toto řešení nesmí způsobit deadlock komponenty. Pracuje se nad kopiemi objektů apod.
- Řešíte transakce na úrovni objektů (konzistence stavů při změnách v objektech). Při změnách v objektech může kdokoliv jiný vstoupit do vámi připraveného scénáře a provést úpravy, které radikálně mění situaci
- Řešíte přístupová práva na úrovni objektů: Ve volání metod objektů se musíte odvolat na část systému, která analyticky řeší přístupová práva (agenda uživatelů apod.)
- Řešíte rychlost zpracování požadavků při růstu počtu klientů, tedy multithreading komponenty (pro ActiveX EXE pod VB 6.0 velmi obtížné, spíše nemožné vůbec řešit)
- Řešíte problém ztráty identity klientů databáze, protože konekt do databáze se zúžil na jeden konekt komponenty pod jejím účtem a pokud v databázi používáte k něčemu uživatele (user connection), nastává kolize, protože existuje pouze jeden uživatel a tím je sama komponenta .

Samozřejmě tyto problémy neznamenají, že nemáte přistoupit k řešení pomocí sdílené komponenty ze zásady, jsou pouze upozorněním a varováním. Řeč teď není o tom, zda je výhodnější použít sdílenou komponentu nebo komponentu s izolovanými klienty obecně, ale kterou vybrat.

Poznámka

Většinou se business objekty navrhují v módu izolovaných klientů a technologické komponenty (queue servery, object brokery pro pooling instancí, connection brokery pro pooling konektů atd.) jako sdílené instance komponent.

Postup návrhu komponent

Při návrhu komponent se vychází z modelu tříd a tvorby packages v tomto modelu. Tedy již v analytické fázi se začnou tvořit package ze skupin tříd. Tyto package se stávají „stavebními kameny“ pro návrh komponent. Připomeňme, že přitom platí tyto zásady:

- mezi package nesmí nastat circular refeence
- komponenta se skládá z 1 až N package (které obsahují třídy)
- třída by se měla vyskytovat pouze v jedné komponentě

Poznámka:

Tyto zásady nejsou syntaktickými zásadami UML, ale zásadami pro efektivní postup návrhu komponent!

Z uvedených zásad přímo vyplývá:

- že mezi komponentami nenastává circular reference,
- že hranice komponent nikdy neprocházejí napříč package ze tříd,
- že daný package by se měl vyskytovat vždy v jedné a pouze v jedné komponentě

Nejčastějším problémem při návrhu komponent je „odstranění“ circular reference. Mnohdy totiž nastane situace, kdy „logicky vzato“ by k oddělení mělo dojít, ale „ať modeluji jak modeluji“, vzniká mi

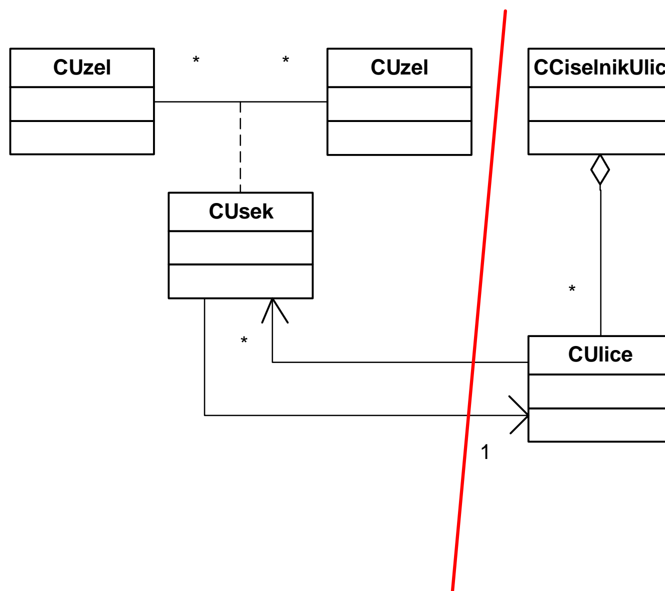
Objektové modelování a UML v praxi 2000, © Kraval Ilja, leden, červen 2001
Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

circular reference. Tento problém jsem nazval „chybou siamských dvojčat“, tj. existuje něco, co spojuje jinak vcelku logicky oddělitelné celky. Otázkou je, jak odstranit toto chybné spojení.

Základní chybou, která vede k circular reference, je nedodržení čistoty pojmů v modelu. Uvažujme logicky: Pokud jsou pojmy zavedené mimo model od sebe logicky oddělitelnými a v modelu nikoliv, tak při přechodu od pojmů k modelu existuje nějaká chyba a protimluv, tj. jeden nebo více pojmů z modelu neodpovídá pojmu z logiky věci. Pokud by byl obraz jedna ku jedné, mělo by dojít k oddělení entit také v modelu. Tedy hledejme, který pojem neodpovídá logice věci. Jeho odhalením vznikne jeden nebo více přesnějších pojmů, model se rozšíří o další prvky (třídy) a najednou lze od sebe pojmy oddělit.

Příklad:

Vraťme se k našemu řešení optimalizace svozu odpadu. Existují Uzly, které jsou spojeny Úseky. To, že Uzel může být Křižovatkou nebo Skládkou nebo Garáží není nyní zajímavé. Důležité pro náš příklad je to, že se nyní zavádí nový pojem a tím je Ulice. Existuje seznam Ulic v městě, tj. číselník Ulic. Sama Ulice nese pouze svůj název a nic víc. Ulice procházejí jednotlivými Úseky a to tak, že žádná Ulice nekončí uprostřed Úseku. Pokud by tak měla končit, umístí se na tento konec Ulice nový Uzel. Seznam Ulic jako číselník agreguje své itemy, tj. Ulice. Na druhou stranu jeden Úsek „vidí“ jednu svou Ulici z tohoto seznamu a naopak jedna Ulice, protože prochází více Úseky, „vidí“ N svých Úseků, kterými prochází. Z jedné strany je tedy vztah jedna a z druhé N. Namalujme odpovídající diagram tříd v prvním přiblížení. Současně byl vznesen požadavek, aby číselník ulic bylo možné jako komponentu dodávat i do jiných systémů, jiným firmám atd., protože je o něj zájem. V diagramu je také naznačen „řez“ kudy by měla hranice komponenty procházet:



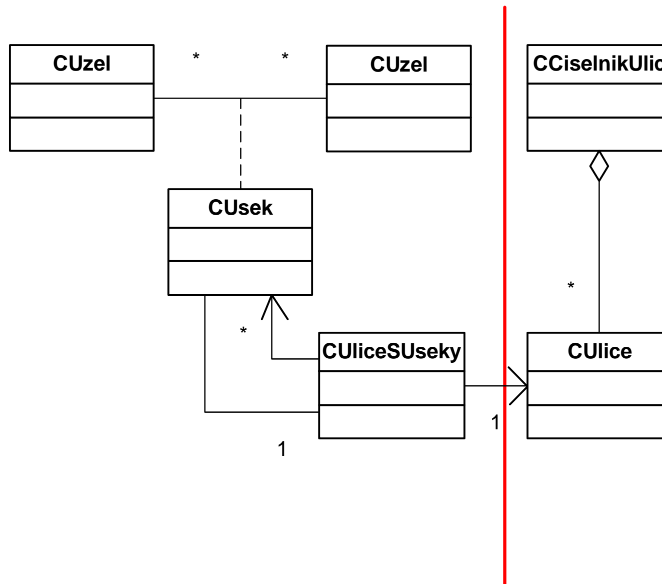
obrázek 77 Nezdařený pokus o odstřihnutí komponenty číselníku ulic

Poznámka: V diagramu jsou vyznačeny dvě jednosměrné asociace mezi úsekem a ulicí. Je možný i zkrácený zápis pomocí jedné obousměrné asociace.

Všimněme si, že takto provedený střih není možný a jasně vidíme proč: existuje zde circular reference. Na jednu stranu sice Úsek potřebuje Ulici (to je OK), ale také Ulice potřebuje vidět Úsek. Nemůžeme tedy dodat samostatnou komponentu ulic. Pokud bychom chtěli dodat komponentu ulic v této podobě, jak ukazuje diagram, museli bychom dodat také třídu Úsek, a s ním také třídu Uzel, tedy dodali bychom (vcelku zbytečně) celý namalovaný model.

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

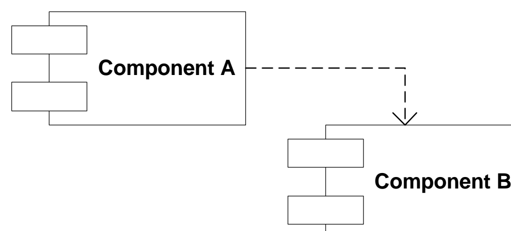
Kde je chyba? Samozřejmě v předešlých odstavcích je nesouhlas. Podle textu je Ulice chápána jako entita, která nese pouze název. Pokud se však podíváme do diagramu, tak to, co jsme v diagramu nazvali ulicí, nemá tuto vlastnost, ale **navíc** ještě vidí Úseky. To, co je na diagramu, není tou „čistou" ulicí, jak ji chápeme v číselníku ulic. Náprava je samozřejmě v nápravě této chyby. Pokud chceme zavést „čistou" ulici, tak musí být čistá i v diagramu. Kromě této „čisté" ulice bude existovat ještě „nečistá" ulice, nazvěme ji „Ulice s úseky":



obrázek 78 Zdařilý střih komponenty

Dependency v komponentním modelu

Hlavním posláním komponentního modelu je zobrazit vztah mezi komponentami v jejich závislostech, tj. ve vztahu dependency. Připomenu pouze, že vztah dependency není výsadou tohoto modelu, ale patří k obecným mechanismům UML. Zde má však velmi velký význam, protože ukazuje, která komponenta se musí ke které linkovat. Na následujícím obrázku komponenta nazvaná A potřebuje ke svému životu komponentu B.



Takto ztvárněný model celého systému jej zobrazí rozvrstven do komponent a znázorní jejich závislosti. Systém přestává být nedělitelným molochem. Systém je přehledný, fyzicky členěný, rozpadající se na menší řešitelné části.

Třívrstvová architektura

Text kapitola viz přiložená skripta „Třívrstvová architektura v praxi“ zabalená do zipu spolu s touto e-knihou jako její součást.

Deployment model

Jako poslední model uvedu tzv. deployment model. Jedná se o model implementační, který zobrazuje rozmístění zdrojů, částí HW apod. v rámci systému. Zde můžeme znázornit, jaké a kolik bude použito strojů, procesorů, jaké mechaniky atd. Jedná se tedy o model nejvzdálenější od analýzy, o model typu „železo“ spolu s údaji „co kde poběží“. Vzhledem k tomu, že tyto modely bývají součástí obchodních dokumentů, nedoporučuji jej tvořit přímo v UML, protože výrazové prostředky UML jsou dost omezené a strohé. Rozmístění zdrojů, znázornění sítě atd. doporučuji vytvořit v některém z prostředků pro tyto účely vhodném, který obsahuje spoustu elementů pro namalování rozložení zdrojů, například VISIO apod.

Nejčastější chyby objektového modelování

V této kapitole jsou shrnuty nejčastější chyby OOP a objektového modelování. Vzhledem ke shrnujícímu charakteru této kapitoly se mohu některé její pasáže oproti předešlým kapitolám opakovat.

Zaměňování třídy a instance a chybné vyhledávání tříd

Pokud se správně nevysvětlí pojem třída, většinou se začínající pracovníci v objektovém prostředí dopouštějí „fatálních chyb“. Zaměňují pojmy objekty a třídy, většinou se setkávají s problémem vůbec formulovat problém, protože touto záměnou jednou o něčem hovoří jako o objektu, potom jako o třídě. Výsledkem je totální zmatení v pojmech a nedorozumění.

Polymorfismus za každou cenu

Setkal jsem se dost často s mylným názorem a mýtem, že správný objektový program musí obsahovat polymorfismus a tak se zavádí i tam, kde není třeba. Pokud se polymorfismus takto nadbytečně zavede, potom se objektové struktury stanou velmi nepřehledné, protože se dávají „do jedné rodiny” objekty, které spolu vůbec nesouvisejí.

Objektová reference a její chybné chápání

Vztah mezi objekty je zdánlivě podobný datovému vztahu v ERD, kdy přes klíč jsou provázána data z různých tabulek. Avšak existuje tu jeden velmi podstatný rozdíl vyplývající z vlastnosti zapouzdření objektu. V ERD existují dvě datové struktury stojící **vedle sebe** a ty jsou provázané přes klíč. U objektů tomuto odpovídá nikoliv spojení, ale **vždy pohled vnořování do sebe**. Nedocenění této vlastnosti vede k silnému narušování objektových modelů.

Porušení anonymity klienta

Když budeme objekt navrhovat (tj. psát pro něj třídu), tak sice jsme vyšli z kontextu jeho celkového použití v systému, ale navrhujeme jej tak, aby jeho funkcionalita nezávisela na tom, kdo ji bude používat. Objekt pouze plní svoje funkce a nestará se to, kdo a kdy bude jeho funkcionality používat.

Dodržování tohoto principu je pro začátečníky v OOP poměrně dost náročné.

Narušení anonymity klienta znamená silné „znepřehlednění programu" a následně vznikají nestability v systému.

Narušení zapouzdření objektu

Narušení zapouzdření vede k narušení dvou základních „kladných" vlastností v OOP a těmi jsou ***konzistence vnitřních stavů objektu*** a ***úplnost informace objektu***. Důsledky:

- Systém je silně nepřehledný, nestabilní a současně neodolný vůči změnám. Jakákoliv změna vede k následným změnám (bůhví kde) mimo změněnou oblast
- Systém vytváří lepence a molochy dále již nedělitelné. Pokud chcete dodat pouze část systému, zjistíte, že musíte k zachování funkcionality dodat další a další části, které s tímto problémem zdánlivě nesouvisejí, ale musejí být také dodány. Díky rozbití pojmů a jejich nevyhranění se funkcionalita „rozprskla" přes celý systém. Velmi obtížně se v takovém prostředí zavádějí komponenty (problém „siamských dvojčat", nelze „odoperovat" od sebe části systému propojené jako siamská dvojčata)

Pokud používáme OOP, potom díky zapouzdření mají pojmy „ostré" kontury. Je pouze otázkou kvality objektové analýzy, zda vytvářené pojmy (budoucí objekty) obsahují to, co obsahovat mají a neobsahují to, co obsahovat nemají.

Chyba tříštění objektů

Tato chyba souvisí s předešlými chybnými postupy. Analytik, který se snaží vystihnout daný problém, uvažuje pouze v pojmech nižší úrovně a není schopen z těchto pojmů vytvářet další jinak logicky chápané složené pojmy vyšší abstrakce. Jedná se anachronismus převzatý ze strukturálního programování, kde každá proměnná bojuje sama za sebe a pojmy a třídy jako takové neexistují.

Například analytik uvažuje o Ceně zboží, o Druhu zboží, atd. a přitom jej nenapadne, že hlavním pojmem je Zboží samo o osobě. Výsledkem je snaha „nacpat" Cenu zboží do Řádku faktury a nikoliv objektovou referenci na celé Zboží.

Chyba nadbytečného ID

V mnoha případech se při použití relační databáze jako uložiště dat pro objekty zavádí přebytečné ID v tom smyslu, že kromě objektové reference na daný objekt se navíc uvede mezi atributy „cizí klíč ID".

Příklad: Řádek faktury „vidí" svou Fakturu, tedy má referenci na objekt Moje_Faktura a navíc se do Řádku faktury přidá ID Faktury. Jedná se vlastně o modifikaci chyby tříštění objektu, protože ID Faktury patří do Faktury a nikam jinam!

Chyba siamských dvojčat při návrhu komponent

Hlavní příčinou této chyby je podcenění použití jinak velmi doporučovaného model elementu **Package** v návrhu pomocí UML anebo jeho žádné použití (…a přitom zavedení Package patří k základním mechanismům UML!). To vede nakonec k velkým problémům při návrhu komponent. Chybný analytický návrh s „nečistými" pojmy v analýze vede nakonec k tzv. circular reference mezi třídami a výsledkem je nemožnost oddělit pojmy od sebe do komponent. Pomyslné části systému „namalované tužkou" nelze od sebe oddělit jako nelze oddělit siamská dvojčat. Vznikají tak nedělitelné systémy jako molochy, což s s sebou přináší nemožnost dělení systému na menší celky. Nejenom že takovýto

Tento dokument byl vydán jako single licence.
Není povoleno šíření dokumentu mimo prostor licence!

systém se musí vždy dodávat jako jeden celek (což je vzhledem ke vztahu k zákazníkům velmi nepříjemné), ale řízení projektu v takovém prostředí je velmi obtížné. V nedělitelném molochovi, který touto chybou vznikne, opravdu platí, že všechno souvisí se vším a žádná z kapitol vývoje není nikdy uzavřena.

Sdílená komponenta

Jak bylo řečeno v kapitole o komponentním modelu, pokud zavedete komponentu, ať už na síti, nebo na lokále, máte z hlediska jejího postavení vůči jejím klientům dvě základní možnosti, jak se instance této komponenty bude chovat (samozřejmě tuto volbu provádíte před jejím vytvořením ve fázi designu).

- Instance komponenty je sdílená klienty
- Instance komponenty není sdílená klienty a pro každého klienta existuje nová instance komponenty.

Pokud použijete sdílenou komponentu, potom vytvoříte spoustu sekundárních problémů k řešení, které máte momentálně ve 2-vrstvové architektuře již vyřešeny.

Konec dokumentu

Připomínky prosím adresujte na mail adresu objects@objects.cz

Autor RNDr. Ilja Kraval