Zásady pro řízení projektů tvorby IS v objektově orientovaném prostředí za použití UML

Motto : Každá složitá věc se skládá ze spousty malých jednoduchých věcí

© **Ilja Kraval**, vydáno březen roku 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 (údaj viz záhlaví), je oprávněn ke čtení a studiu 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.

Úvod

Úvodní slovo autora

E-kniha, kterou právě čtete, navazuje na elektronickou publikaci „Objektové modelování a UML v praxi 2000“, která také vyšla na Serveru objektových technologií a setkala se s velmi příznivým ohlasem. Předešlá e-kniha o objektovém modelování pojednává o vývoji a tvorbě IS objektově orientovaným způsobem za pomoci UML. Předcházející kniha je určena všem pracovníkům činným v oblasti tvorby SW, tj. analytikům, vedoucím projektů, programátorům, testerům atd. Zvládnutí předešlé knihy o objektovém modelování podává čtenáři ucelený obraz o objektově orientovaném přístupu při tvorbě SW, o modelovacím jazyce UML a o tvorbě SW pomocí komponent.

Po napsání této knihy se mi přihodila jedna zajímavá příhoda. V jedné „blíže nejmenované softwarové firmě“ se po každém ukončení školení podle vcelku dobrého zvyku provádí dotazníkovým způsobem průzkum úspěšnosti školení. Pomocí klasických otázek se zjišťují ohlasy a reakce posluchačů na průběh a výsledek kurzu. Také se tímto způsobem získají náměty a návrhy pro další semináře. Měl jsem možnost v této firmě provést vcelku obsáhlé 5-denní in-house školení OOP – UML – COM i s následnými konzultacemi. Tak trochu s napětím jsem poté očekával výsledky této dotazníkové metody. Nejzajímavějším na tomto průzkumu byl poznatek, že takřka ve všech dotaznících se objevil návrh, aby toto školení absolvovali také vedoucí pracovníci. Vzpomněl jsem si při té příležitosti, že také v jiných diskusích u jiných firem častokrát zazněl v průběhu školení tak trochu „realisticko – pesimistický“ názor, že pokud se vedoucí neseznámí s touto nejmodernější technologií tvorby IS za pomoci OOP a UML při použití komponent, tak se tento způsob tvorby SW nikdy ve firmě nezavede, a nutno dodat, že s tímto názorem lze jen souhlasit.

Jinak řečeno, mezi účastníky školení mnohdy převládl tento názor:

„Tvorba IS za pomoci OOP, UML a komponent je velmi přínosná, pochopili jsme její základy, líbí se nám a cítíme potřebu ji zavést v naší firmě, ale toto školení by také měli absolvovat ti, kteří o tomto nějak rozhodují, tj. vedoucí. Bez toho nevidíme moc velké šance na zavedení těchto postupů v naší firmě…“

Návrh je to sice pěkný, ale kdo se alespoň trochu seznámil se zvyklostmi a zaběhanými mechanismy ve firmách, tak si je velmi dobře vědom toho, že předešlou knihu o objektovém modelování v takovém rozsahu a s takovým zaměřením na technologii v tom duchu, jak je napsána, ji nebude většina vedoucích nikdy číst. Anebo v to lepším případě pokud alespoň knihu začnou číst, tak ji určitě většina z nich nedočte až do konce. Kniha je totiž více zaměřena na technologii jako takovou a soustřeďuje se na její podrobné vysvětlení. Z toho důvodu je elektronická publikace dost podrobná a obsáhlá. Tato její přednost je pro vedoucí pracovníky spíše na škodu než ku prospěchu. Přitom je zřejmé, že by každý vedoucí měl být seznámen s moderními objektovými technologiemi alespoň v takovém rozsahu, v jakém jsou znalosti pro vedoucího vyžadovány.

Z toho důvodu vznikla tato e-kniha pojednávající o zvláštnostech řízení projektu v objektově orientovaném prostředí za pomoci UML a komponent.

Je vcelku pochopitelné, že tato kniha bude v mnohém navazovat na předešlou knihu o objektovém modelování, což vede k určitému nepříjemnému dilematu: Do jaké míry se mám jako autor obou knih v této nové knize vůči knize prvé opakovat? Na jednu stranu nechci snižovat úroveň této nové knihy tím, že převezmu celé pasáže z prvé knihy, na druhé straně se nechci odkazovat na předešlou knihu, protože nemohu vyžadovat, aby si čtenář nutně prvou knihu koupil, natož aby ji podrobně studoval, čemuž jsem se chtěl právě vyhnout.

Toto dilema jsem vyřešil v této knize zavedením zvláštních kapitol a příloh. V těchto kapitolách je stručně podán výklad podrobný, ale s větším důrazem na pohled vedoucího pracovníka. Doporučuji také jejich studium, protože jinak nelze pochopit podstatné myšlenky této knihy o řízení projektů v objektově orientovaném prostředí. Koncepce této knihy je tedy taková , že je zde podán hlavně výklad týkající se řízení projektů v objektově orientovaném prostředí, přičemž sama podstata věcí v OOP, v UML je vysvětlena v základech nutných pro pochopení kontextu řízení projektů.

Není povoleno šíření dokumentu mimo prostor licence!

O čem je a komu je určena tato kniha

Tato kniha je určena především vedoucím pracovníkům činným v oblasti tvorby SW, tj. vedoucím projektů, hlavním analytikům, hlavním designérům, ale také ředitelům softwarových firem apod.

Zvládnutí této knihy podá čtenáři ucelený obraz o hlavních zásadách řízení projektů v objektově orientovaném prostředí při tvorbě SW za pomoci modelovacího jazyka UML a při tvorbě SW pomocí komponent.

V žádném případě tato kniha nepopisuje celou rozsáhlou problematiku řízení projektů jako takovou, tj. nepojednává o problematice řízení projektů vcelku a úplně. To je samozřejmě obsahem jiných a také obecnějších a také velmi rozsáhlých knih o různých metodách řízení projektů pojatých ve své obecnosti vůči libovolnému projektu v libovolné oblasti projektového řízení. Tato kniha popisuje hlavně zvláštnosti řízení projektů tvorby IS v objektově orientovaném prostředí a také v prostředí komponent a za použití UML. Chci zde upozornit na ty skutečnosti, na které je třeba se zaměřit, na to, čemu je třeba věnovat zvláštní pozornost, čeho se vyvarovat, apod., a to pouze vzhledem k tvorbě IS pomocí OOP a UML.

Přijmutí hlavní koncepce tvorby SW

Jaký je vlastně cíl tvorby informačního systému?

Samozřejmě „dobře chodící informační systém“, můžeme odpovědět stručně. Takto „vývojářsky“ byl projekt tvorby informačního systému (dále také IS) chápán až do přibližně 80.let minulého století. Pojetí IS se však již výrazně změnilo. Základním požadavkem, který se týká jeho tvorby, je nejenom požadavek na vývoj „dobře chodícího systému“, ale také na jeho „ekonomický zisk“. Poloha tvorby SW se dostala do ekonomických dimenzí, tedy jedná se již o „výrobu SW“ a nikoliv pouze o jakousi kreativní tvorbu.

V posledních dvou desetiletích se v souvislosti s informačními technologiemi hovoří o tzv. krizi softwaru. Tato krize se projevuje tak, že firmám dodávají IS rostou enormně náklady na údržbu informačního systému, tvorba IS je chaotická a mnohdy příliš nákladná. V mnoha případech dokonce firem není schopno projekt ani dokončit, natož jej dovést k požadovanému zisku.

Oproti tomu úspěšnými SW firmami jsou v období této krize SW ty firmy, které výrazně minimalizují náklady. Zejména jsou to takové firmy, které minimalizují ty náklady, které vznikají **úplně zbytečně**. Krize SW je totiž spojena právě se vznikem **zbytečných nákladů**. Naopak odstraňování zbytečných nákladů je právě důsledkem zavádění správných moderních postupů tvorby IS. Hlavní otázkou tedy je: Kde a jak zbytečné náklady při tvorbě SW vznikají a jak je odstranit?

Symptomy projektů se zbytečnými náklady

Odpovězme si na tyto otázky: Je stav ve firmě, resp. v projektech, stručně charakterizován následujícími slovy?

- tvoříme software těžkopádně
- vyrábíme software 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...

To je jenom malý výčet důsledků tvorby SW zastaralým způsobem a tedy tvorby SW s velmi vysokými a přitom zbytečnými náklady. Pokud pociťujete ve firmě některé z těchto symptomů, je to signál pro změnu, protože se jedná pro vaši firmu o **existenční otázku**. Firma v tomto stavu určitě nepřežije delší časové období.

Poznámka: Na začátku 90. let po sametové revoluci nastal velký „boom“ výroby softwaru v ČR. V té době byly nároky uživatelů na kvalitu velmi nízké a uvedený postup se zbytečnými náklady byl ve své době možný a vzhledem k vysoké poptávce po softwaru s okamžitou rychlou dodávkou možná jediný možný. Avšak všimněme si, jak nyní vzrůstají požadavky na kvalitu softwaru ze strany uživatelů. Nyní se již stává problémem uplatnit na trhu nekvalitní software, protože existují technologie, které se takovýmto rychlým a chybným postupům výroby SW brání. Trh 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 v obchodní oblasti. Požadavky zákazníků se totiž začínají také v ČR rapidně měnit a to směrem nahoru k požadavku na vyšší kvalitu SW.

Jaká má být zavedena koncepce řízení

Odpověď na otázku „jak vznikají zbytečné náklady“ a „jak je odstranit“, není až tak složitá: Sofisticky řečeno, zbytečně náklady vznikají zbytečně a dají se odstranit zavedením takových procesů ve firmě, které vedou k odstranění zbytečných prací, opakujících se prací, chybových prací apod. Otázkou je, co to znamená pro firmu konkrétně.

Existuje jeden „globální recept“, ztvárněný do koncepce, která vede k minimalizaci zbytečných nákladů. Nazval bych tuto koncepci **koncepcí zavádění maximálního re-use ve firmě**. Pojem „re-use“ zde používáme jako synonymum pro „znovupoužití“. Opakem „znovupoužití“ je „na jedno použití“.

Tady je třeba se pozastavit, protože pojem re-use musíme nyní blíže vysvětlit a dát jej do souvislosti s koncepcí řízení projektů. Co se vlastně chápe pod pojmem re-use? Většinou se re-use spojuje pouze se znovupoužitím zdrojového kódu. Zdrojový kód se namísto opakování „re-usne“, tj. znovupoužije. Avšak re-use je třeba chápat jako obecnější přístup než pouze pro znovupoužití zdrojového kódu. Re-use budeme chápat jako něco, co obecně souvisí s opakováním se ve vývoji „něčeho“ a může to být „cokoliv“, například i části různých modelů, celé pasáže nějaké dokumentace apod. Základní smysl re-use je „zabránit opakování ve vývoji, zabránit opakování v tvorbě“, tj. zabránit zdvojení informace a tím zdvojení práce, která se nemusí týkat pouze kódování (ale i modelování, dokumentace, analýzy atd.). A je vcelku pochopitelné, že zabránit opakování zavedením re-use není nic jiného, než zabránit vzniku zbytečných nákladů.

Platí základní obecné pravidlo při zavedení libovolného re-use:

Pokud se vyžaduje shoda něčeho (co by se jinak opakovalo) v různých částech systému, vede to osamostatnění tohoto určitého něčeho, a vytvoření nové samostatné entity onoho něčeho, a k vytvoření odkazů na tento nový pojem, což vede ke sdílení nového pojmu.

Uveďme si klasické příklady:

Případ volání funkcí

Pokud se opakuje kód procedury, můžeme buď přenášet kód pomocí schránky (clipboardem), což není v žádném případě doporučováno. Při použití re-use zavedeme „mechanismus volání procedury“, která je takto sdílena (volána z několika bodů). 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 a na jednom místě a její definice se pokaždé neopakuje.

Případ normalizace databáze

Další příklad se vyskytuje v teorii datových modelů, kdy se ve dvou tabulkách vyskytnou stejné sloupce stejného významu. Jedná se opět o duplicitu kódu, přesněji řečeno 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.

Případ třídy v OOP

Dalším příkladem je třída v OOP - zavádí se 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, je z ní generován, resp. pomocí ní definován. Pomocí třídy je dosaženo shody definicí objektů stejných vlastností.

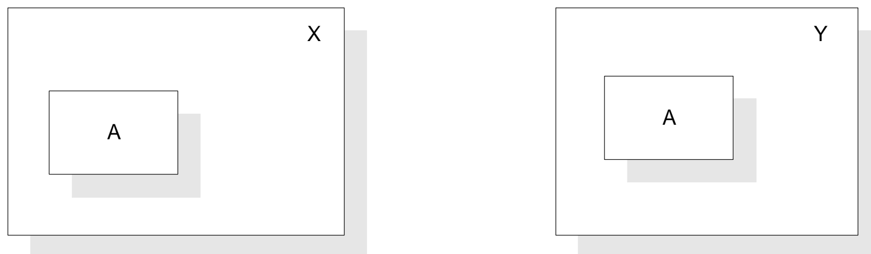
Případ dědění v OOP

Jiným příkladem je zavedení pojmu dědění: Po zjištění , že mohou existovat stejné vlastnosti ve dvou nebo více třídách, se zavede nový pojem dědění z třídy předka do třídy potomka. Všechny společné vlastnosti se spojují do společné jedné třídy. Kód je sdílen pomocí dědění.

Případ analýzy IS a designu IS v projektu

Pokud oddělíme modely analýzy IS od následných modelů designu IS (bude vysvětleno ve fázování projektu), a zpětně modely provážeme odkazem, potom dostaneme „samostatnou“ dokumentaci analýzy IS, kterou můžeme sdílet do těch projektů, které mají stejnou analýzu, ale různý design (například stejný analytický problém, ale jiná databáze, jiná platforma atd.). Analýza se provádí a dokumentuje pouze jednou.

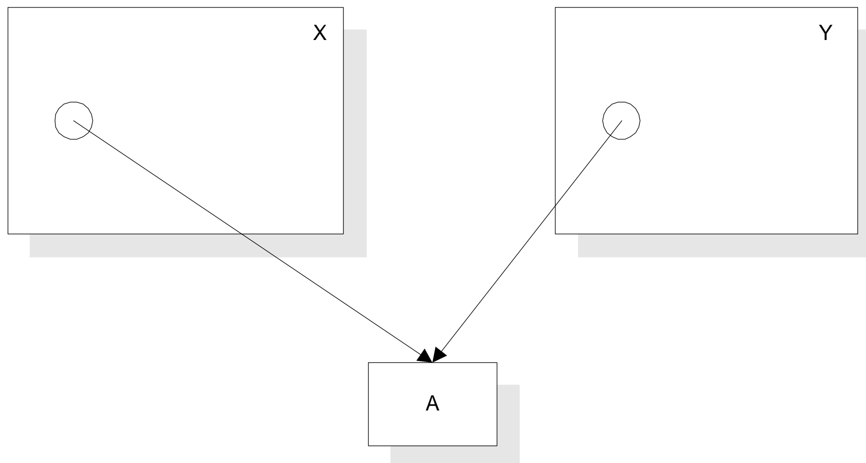
Přitom je zřejmé, že postup pro vytvoření re-use je vždy ve všech případech stejný: Najdi „to co se opakuje“, vytrhni „to co se opakuje“ do samostatného pojmu, a provaž tuto novou entitu zpět do místa, odkud byla vytržena. Vznikne tak sdílení dané entity. Tuto operaci re-use znázorněme graficky.



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 provést, musí být zavedena nějaká zvláštní operace tohoto provázání (přesněji interakce) mezi prvky typů entit A, X, a Y. V předešlém výčtu pro použití re-use mezi třídami, mezi tabulkami, mezi funkcemi, musí být nutně zavedena interakce dědění mezi třídami, join mezi tabulkami, volání mezi funkcemi, provázání analýzy a designu v projektech atd.

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

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 „neopakování již provedených prací" a tedy urychlení vývoje a tím snížení nákladů. Je někdy až překvapivé, jak je tato jednoduchá poučka ve firmách s lehkým srdcem porušována. Přitom re-use patří k prvním zásadním krokům, které drasticky snižují zbytečné 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 musím samozřejmě projít všechny opakující se části a tyto části pokaždé znovu opravovat. Údržba systému se potom stává opakujícími se stejnými úkony.
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, tedy chaosu v systému. Například v předešlém odstavci se při změnách a nepoužití re-use nejenom tyto změny opakují, ale (a to je ještě horší!) musíme si vést evidenci, kde všude tyto opakující se změny musíme provést. V některých případech se při ztrátě anebo nezavedení 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. Při ztrátě re-use se údržba systému stává naopak napínavou detektivkou, což je pro řízení projektu velmi nepříjemné

Základní koncepce řízení

Z uvedeného vyplývá, že jako první „postulát" pro zavádění moderních způsobů řízení můžeme zavést „koncepci maximálního re-use". Tato koncepce řízení znamená, že změny pracovních procesů ve firmě obecně vedou ke zvyšování re-use ve firmě a jsou také z tohoto hlediska vždy posuzovány. Firma, která se řídí dokonalejšími předpisy postupy prací, má také dokonalejší re-use jak svých činností, tak re-use v modelech a následně i v kódu IS. Zavedení vyššího stupně re-use je právě oním efektem, který vede k neustálému snižování zbytečných nákladů. Tato rovnice je velmi jednoduchá, lze ji stále posuzovat a lze podle ní v kritických situacích rozhodovat.

Objektově orientovaný přístup (OOP) z hlediska re-use

V podstatě existují dva možné přístupy k tvorbě IS

- moderní přístup k tvorbě IS pomocí objektů (OOP)
- zastaralý přístup ne-objektový, zvaný též strukturální nebo procedurální.

Mezi těmito dvěma přístupy existuje zásadní filosofický rozdíl, který vede k jinému pojímání informačního systému jako takového. Tento rozdíl vede také k jinému pojetí re-use, který je v OOP mnohem důslednější a efektivnější.

Poznámka: Úvod do teorie objektového programování viz Příloha I

Hlavní výhody OOP versus se projeví ve svých důsledcích zejména v těchto bodech:

Výhoda nových typů re-use v OOP

V systémech napsaných pomocí OOP se mnohem více uplatňují typy re-use, které nemají ve strukturálním přístupu obdobu: zavádí se třída, dědičnost, komponenty a binární objekty, distribuované objekty aj., které oproti strukturálnímu přístupu výrazně zvyšují re-use systému

Výhoda zachování stability systémů při re-use mezi částmi systému

OOP zavádí svým pojetím uzavřených objektů také re-use mezi částmi systému při zachování stability systému, i když dojde ke změnám v těchto částech systému. Strukturální systémy se na rozdíl od objektových chovají jako systémy nestabilní, jsou náchylné ke ztrátě stability po změnách v jiných částech systému, tj. re-use mezi částmi systému vede k následným nestabilitám systému.

Výhoda příbuznosti analýzy systému k realitě, srozumitelnost systému, transparence systému

OOP zavádí jiný a nový pohled na systém složený ze znovupoužitelných částí – objektů pocházejících ze tříd, které jsou v analýze reprezentovány analytickými a srozumitelnými pojmy (faktura, odběratel apod.). To vede k jinému, mnohem efektivnějšímu pohledu na analýzu IS, která se na jedné straně stává přímo „abstraktně naprogramovanou“ součástí systému, přitom pojmy a následně třídy pocházející z analýzy jsou obsaženy v naprogramovaném systému. Současně také dochází k re-use mezi různými systémy se stejnými třídami pocházejícími ze stejných oblastí analýzy. Díky fázování dokumentů projektu na analýzu a design je tak umožněno sdílení analytických prvků modelů mezi systémy jako znovupoužitelných již existujících částí systému.

Výhoda specializace analytiků a designérů

Oddělení analytické části systému a design (návrh) části systému umožňuje velmi efektivně rozdělit specialisty na analytiky IS a na designéry IS.

- analytici tvoří modely IS ve fázi analýzy, mají znalosti z problémové domény, ale nejen to, umějí také modelovat IS pomocí UML. Vytvářejí model IS v analytické rovině.
- designéři (návrháři) jsou experty na dané technologické prostředí, jsou znalí technologie IS, jako například databázista, expert na MS NT, expert na ASP apod. Pro designéry se výsledek analytického modelu (viz předešlý odstavec) stává zadáním pro implementaci v daném prostředí, tj. pro tvorbu designu.

Není povoleno šíření dokumentu mimo prostor licence!

Dochází tak k zajímavému re-use ve znalostech mezi projekty, kdy jeden designér resp. jeden analytik pracuje pro vícero projektů jako expert ve svých oblastech (designér jako expert na technologii, analytik jako expert na problémovou doménu).

Existuje opačný přístup spočívající v příčném řezu přes všechny expertní oblasti. Je charakterizován zadáním prací takto: „Sám proveď analýzu, sám proveď design a sám proveď kódování". Uvedený chybný postup vede k „de-specializaci" pracovníků se sice rozsáhlými, ale nutně pouze povrchními znalostmi bez možností sdílet tyto znalosti přes projekty (tj provést re-use mezi projekty).

Výhoda snazšího, efektivnějšího a také standardního modelování a re-use modelů

Při zavedení OOP je mnohem snazší zavést modely IS pomocí standardu UML, jejichž části lze také sdílet mezi projekty a tedy provádět re-use na úrovni modelů.

Výhoda OOP spočívající ve stabilitě systému při jeho vývoji

Objektové programování přináší do systému dvě nové kvalitativně odlišné vlastnosti, které strukturální programování nezná. Jsou to

- ***konzistence stavů objektu***
- ***úplnost informace objektu***

Tyto dvě vlastnosti přinášejí systému jednu velmi výhodnou vlastnost – stabilitu systému při vývoji systému a při jeho nutných změnách.

Podrobně o těchto pojmech viz Příloha II

Objekty jsou v systému velmi uzavřené struktury, se kterými lze komunikovat pouze tak, že se jim posílají zprávy (o objektech viz Příloha I). Tato uzavřenost objektů se nazývá zapouzdření objektů. Objekt je podobně jako „součástka systému" (například podobně jako kalkulačka v realitě) nositelem svých vnitřních stavů.

Každé narušení zapouzdření objektu v konečném důsledku znamená ztrátu konzistence stavů objektu (jakoby v realitě „vykuchaná kalkulačka"). Objekt začne měnit svoje stavy, aniž by v něm proběhly jakékoliv procesy (aniž bychom stiskli tlačítko na kalkulačce). 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 proveď následně také testy a hledej, co začne padat resp. chovat se jinak, a tam to oprav!". Slovy klasika takovýto způsob oprav zdá se mi poněkud nešťastným.

V OOP je problém změn 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. A přitom jak často se lze setkat s programy s bohužel zázračným a tedy nevypočitatelným chováním!

Není povoleno šíření dokumentu mimo prostor licence!

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“ - 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.

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

Re-use vzhledem k týmové práci a požadavky na procesy ve firmě

Použití re-use může mít několik úrovní vzhledem k týmové práci:

1. žádný re-use
2. re-use na úrovni pracovníka – pracovník sdílí své vlastní výsledky a „neopakuje se“
3. re-use na úrovni projektového týmu – pracovníci sdílejí výsledky práce v rámci týmu
4. re-use na úrovni divizí, odštěpných závodů apod. u velkých firem– pracovníci v rámci jednoho závodu sdílejí výsledky
5. re-use na úrovni celé firmy – uvnitř celé firmy se „nic zbytečně neopakuje“
6. re-use mezi firmami – trh s komponentami apod.

Proč modelování IS pomocí UML a re-use specializací

Na jednu stranu se vyžaduje dosáhnout z hlediska týmového re-use maximálního stavu spočívajícího v tom, že re-use funguje na úrovni celé firmy. Znamená to, že se nutně zavádějí mechanismy a procesy pracujícími nad tzv. firemními knihovnami, které jednak „uskladňují“ výsledky práce, ale také tyto výsledky nabízejí pro re-use a sdílení v jiných situacích znovupoužití.

Na druhou stranu projekty tvorby IS již dnes nejsou jakýmisi jednoduchými malými prográmky, kdy jeden „specialista programátor“ vytvoří „celý systém“. I vcelku nepříliš rozsáhlé informační systémy jsou příliš složité na to, aby tyto systémy bylo možné rovnou naprogramovat jedním člověkem.

Vzrůstající složitost systému je dána jednak

- rozrůstajícími se analytickými požadavky na systém
- neustále rostoucími požadavky na znalost moderních technologií zaváděnými v systému
- neustálým zdokonalováním syntaxe jednotlivých programovacích jazyků, zaváděním nových jazyků

Vyplývá z toho, že při vývoji složitého informačního systému se vyžadují z hlediska „vývojářského“ minimálně tyto znalosti:

- znalosti z dané problematiky, kde má být IS nasazen (znalosti z problémové domény)

- znalosti technologií použitých prostředků (znalosti prostředí, databáze, síť, Internet / Intranet, bezpečnost atd.)
- znalosti programování a syntaxe jazyků (C++, Delphi, Smalltalk, Java, C# apod.)

Je zřejmé, že zavedení re-use již na úrovni projektového týmu znamená zavést dohody, jinak řečeno zavést nějaké metodiky a procesy ve firmě, které povedou ke sjednocení a možnosti vzájemné předávky prací (resp. pouze v týmu). Není třeba zdůrazňovat, že například zavedení re-use na úrovni celé firmy je mnohem obtížnější, než zavést re-use na úrovni jednoho týmu.

V dnešní době již není reálné, aby jeden člověk získal expertní znalosti ze všech tří oblastí najednou.

Příklad: Má být vyvinut systém bankovní agendy úvěrů všech typů (krátkodobé, střednědobé, dlouhodobé, splátkové kalendáře, jištění, účetnictví úvěrů atd.), kde v pozadí poběží MS SQL 7.0, další požadavek je použít Intranet technologii ASP, komponenty budou psány v C++. Je zřejmé, že nelze být současně expertem na problematiku úvěrů, expertem na použitou technologii ASP a MS SQL a také vynikajícím programátor v C++ s perfektní znalostí MFC knihoven apod. Kromě těchto znalostí existují samozřejmě ještě další oblasti vědomostí, jako například pro vedoucího projektu je nutná znalost teorie řízení projektů apod. Tento výčet zde uvádím proto, že se jedná výčet minimálně nutný pro vývojový tým

Podle tohoto hlediska specializace na tři oblasti můžeme tedy projektové řízení tvorby IS rozdělit na základní skupiny:

Projekty řízené metodou příčného řezu

V tomto velmi běžně zavedeném systému řízení se postupuje takto

- pracovníkovi se zadá modul
- pracovník provede analýzu „sám pro sebe“
- pracovník provede design „sám pro sebe“
- a také sám nakóduje (mnohdy i otestuje) a odevzdává.

Samozřejmě pracovník tyto fáze nijak nepociťuje jako oddělené, protože dokumentace jakýchkoliv fází chybí. Pracovník přece nebude sám sobě analýzu a design předávat. Pokud budete po pracovníkovi žádat vyhotovení těchto dokumentů, odpadá v tomto systému předávky „sám sobě“ nejzákladnější ověření správnosti těchto dokumentů spočívající v předání dokumentace (modelů) od osoby k osobě. Předání od osoby k osobě zaručuje, že dokumentace musí být úplná, protože ten, komu je předáváno, samozřejmě musí rozumět dokumentaci předávané.

Zajímavé je, že pojem analýza IS jako model IS V tomto systému metody příčného řezu ani neexistuje jako nějaká reálná fáze, protože neexistují žádné odpovídající typy dokumentu analýzy, které by se v analýze měly vyhotovit. Metoda příčného řezu tedy schovává do sebe analýzu, design a kódování daného modulu jako do jednoho uzavřeného balíku.

Takovýto systém práce je z hlediska řízení nejjednodušší, ale má v důsledku velmi nepříjemné následující nevýhody:

- zavádí nezastupitelnost pracovníků. Je zřejmé, že daný pracovník musí být u všeho, co se týče „jeho agendy“ a to vždy, protože není nikoho jiného, kdo by jej zastoupil. Řízení lidských zdrojů je silně omezeno (pokud se dá vůbec hovořit o nějakém řízení lidských zdrojů). To má nepříjemné důsledky jak pro firmu, tak pro samotného pracovníka (nemoc, dovolená apod.)

- nelze efektivně zavést re-use ve firmě, mnohdy se některé části řešení opakují apod. (Například v jedné firmě byla u jednoho velmi rozsáhlého systému jedna funkce převodu čísla jako stringu na BCD tvar napsána a naprogramována celkem sedmkrát)
- nelze zavést specializaci pracovníků, z pracovníků se stávají „vševědi“, ovládající jak problémovou doménu, tak ovládající problematiku technologie, jakož i programování (například jeden pracovník „umí jak úvěry, tak umí MS NT a MS SQL a umí také C++“)
- zaučení nových pracovníků je zdlouhavé, pro ně samotné namáhavé a velmi těžkopádné

K těmto nevýhodám bych přidal jednu zajímavou osobní zkušenost. Mnohdy se ve firmách můžeme setkat s tzv. „umělou nezastupitelností“. Někteří pracovníci se velmi brání zavádění jakýmkoliv procesům ve firmách, které by narušily jejich nezastupitelnost v dané agendě. Brání se provést nějakou dokumentaci s argumentem: „...je třeba programovat a nikoliv něco spisovat“. Záměr těchto pracovníků je zřejmý: Pokud totiž zdokumentují svoje znalosti, chápou to jako reálné nebezpečí, že je firma již nebude potřebovat, anebo přinejmenším jejich „zásluhovost poklesne“. Takto vytvořená nezastupitelnost je umělá, doplácí na ni samozřejmě firma, ale paradoxně mnohem více na ni doplatí samotný pracovník, protože tento způsob vede k jeho silnému vnitřnímu uspokojení a zmenšené snaze „být o krok vpředu“. Osobně jsem znal jeden případ, kdy velmi dobrá expertka na bankovní účetnictví byla v určité SW firmě díky tomuto přístupu nezastupitelná, po třech letech své umělé nezastupitelnosti zjistila, že v bankách již účetní znají některé oblasti mnohem lépe. Jako expertka na bankovní účetnictví již skončila.

Projekty řízené metodou specializace na analýzu, design a kódování

Pro firmy je samozřejmě výhodné, pokud existují skuteční experti v uvedených oblastech. Důvody pro takovéto rozdělení jsou minimálně tyto:

- toto rozdělení poskytuje možnost vzniku a odborného růstu specialistů, kteří se více věnují pouze své oblasti činností a nemusí se stávat „všeznalci“. Existuje mnoho firem, které uznávají a používají opačný přístup - metodu příčného řezu. Takovýto styl práce nejenom že je velmi neefektivní z hlediska re-use (každý bojuje sám za sebe), ale navíc nutí pracovníky, aby se stávali povrchními experty pro všechno a nikoliv specialisty. Důvodem použití této metody je primitivní způsob řízení pracovníků nevyžadující žádné další znalosti, protože řízení degraduje na pouhé přerozdělování požadavků.
- pokud se znalosti o systému z problémové domény „nějak zapíší“, tj. dojde k nějak k zápisu znalostí pouze pomocí pojmů z problémové domény, lze tyto znalosti použít i v jiných systémech, které se liší pouze použitou technologií (přechod na jinou databázi, přechod na Internet apod.). Tento typ re-use je v OOP a komponentní technologii velmi silný díky své objektové povaze, nejedná se totiž pouze o „re-use znalostí“, ale v konečném důsledku o re-use již naprogramovaných a zkompilovaných částí systému, například tříd a zdrojového kódu, případně komponent apod.
- z vlastní zkušenosti mohu k tomuto výčtu důvodů přidat ještě jeden zajímavý důvod pro zavedení tohoto členění rolí na experta problémové domény a experta technologa. Tento důvod je však tak trochu skrytý a souvisí s psychologickým charakterem činností těchto pracovníků. Způsob práce v uvedených oblastech, tj. v analýze a designu, se totiž dost podstatně liší svou povahou. Při řešení analýzy, tj. při řešení otázek „co systém bude umět“, se vyžaduje získávat (a nějak zapisovat) znalosti o daném systému na dost abstraktní úrovni. Jedná se o popis systému na úrovni problémové domény, což znamená, že řeč není o technice, ale o samotném problému, například „jak má systém řešit úvěry“. Dobré předpoklady pro tuto činnost bude mít pracovník, který rád řeší problémy pouze abstraktně, bez nutné implementace, bez konkrétních technologií jako takových. Takovýto pracovník dovede systém navrhnout pouze jako „představu systému“, což vyžaduje schopnosti myslet abstraktně a nutně také vyžaduje mít velmi dobrou představivost o systému zbavenou jakýchkoliv implementačních podrobností, ale současně model systému je velmi přesný. Naopak po

Není povoleno šíření dokumentu mimo prostor licence!

designérovi, který řešící problém technologicky, se žádá něco úplně odlišného: Musí navrhnout systém podle předložené abstraktní představy převzaté od analytika do konkrétní podoby v dané technologii. Tento pracovník má rád výsledek své práce nikoliv v abstraktní, ale výrazně v konkrétní, hmatatelné a „chodící“ podobě a nikoliv v nějaké abstrakci. Většinou se jedná se o technokrata, tzv. guru na danou technologii, který zná danou technologii do největších podrobností a proto je systém schopen navrhnout lépe, než jiný pracovník. Pokud má daný pracovník pracovat v obou oblastech, bývá tento rozdíl mnohdy pro daného pracovníka, který musí měnit styl práce, vcelku pochopitelně velkým problémem. Toto do jisté míry psychologické rozdělení činností je podstatou onoho zoufalého volání programátorů „dejte nám zadání“. Programátor si rád hraje s pointry, nikoliv s úvěry.

Rozdělení prací na tyto oblasti se jeví jako vcelku logické, naskýtá se však otázka: Jak vlastně mají tito experti spolupracovat? Ptáme se:

- jaký mají mít „styl práce“ (metodiky, procesy ve vývoji)
- jaké mají používat nástroje
- jak se mají mezi sebou dorozumět, tj. jakou notaci mají používat, aby si mohli výsledky práce předat, diskutovat nad nimi, oponovat, dále zpracovávat atd.

Na obecnější úrovni lze odpovědět jednou větou: *„Všichni musí pracovat nad modely systému“*. Pod modelováním systému zde máme na mysli poněkud širší pojem: Modelování je nejenom dohoda jak pracovat s modely systému (synonymum pro metodiky), ale také dohoda čím modelovat (tj. nástroj) a dohoda syntaxe modelování (tj. UML). Tím máme také odpověď na tři otázky v předešlém výčtu.

Modelování a speciálně v OOP objektové modelování pomocí UML je řešením onoho problému, kdy z několika expertů - specialistů v různých oblastech činností nad systémem (znalec problémové domény, znalec technologie, znalec programátor) potřebujeme vytvořit spolupracující tým.

UML – „Unified modeling language“ vznikl jako ověřený a stále se zdokonalující **standardní** modelovací jazyk pro ty systémy, které vykazují znaky objektově orientovaných systémů, tedy je vhodný pro modelování informačních systémů, které jsou tvořeny objektově.

Analýza, design (návrh) a kódování systému z hlediska řízení projektů

UML poskytuje celkem 9 modelů, pomocí kterých lze systém popsat. Některé z těchto modelů se používají pro popis systému pouze v oblasti problémové domény, další modely rozšiřují tyto modely o technologické a implementační podrobnosti, tj. vytváří se modely ve fázi návrhu. Posledním „modelem IS“, který již není součástí UML, je samotný kód.

Z předešlého výkladu vyplývá vcelku logický požadavek, aby v projektech tvorby SW byly zavedeny minimálně tři fáze vývoje. Ze zkušeností v konzultacích a také podle doporučení doporučuji jako nezbytné oddělit a tím dodržet určitou posloupnost od sebe oddělených prací minimálně ve třech fázích.

- Analýza IS
- Design (též návrh) IS
- Kódování IS

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“ ve vztahu k modelům IS.

Model popisující systém pouze v oblasti problémové domény (tj. pouze „co to má umět“) se někdy nazývají esenciální model, někdy také konceptuální model nebo analytický model. Prvky tohoto modelu se nazývají prvky analýzy. Proces tvorby konceptuálních modelů IS, ve kterých se vyskytují pouze prvky business (daného „podniku“), se nazývá **analýza informačního systému**.

Konceptuální modely jsou dále rozšířeny o technologické podrobnosti („jak se bude analýza realizovat“), což je proces **návrhu systému** (také design). 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 (nejlépe v UML notaci). 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 na něj 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 analytické, v designu a jako samotný kód. Každý z těchto pohledů má jiné hledisko. Rozdělení tak vysoce zprůhlední výsledný software, protože jej vidíme v různých abstrakcích - od nejvyšší analytické až po nejhlubší „implicitní“ v kódu. Výhodou použití UML je to, že velmi jednoduše můžeme zavést odpovídající formalismy pro zápis dokumentů. Stačí pouze určit, které modely UML budou vytvořeny v analýze a v designu - samo UML tedy nabízí notaci pro analytické a design dokumenty.
- 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 zavedení 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.
- 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 existovat analytik, designér, programátor. Současně musí být určen způsob předávání dokumentů mezi těmito rolemi.

Rozdělení na fáze tedy zavádí automaticky tři základní výhody:

- přehlednost systému s pohledem na vrstvu analytickou, na vrstvu designu a na samotný kód
- zavedení prvních závazných postupů ve firmě
- zavedení rolí v projektu s povinnostmi těchto rolí

Náplň analýzy IS

Smyslem a posláním analýzy IS je vyhotovit dokumenty pomocí UML, tj. modely v UML, které:

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

Platí důležité pravidlo, že ve fázi analýzy IS 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.) a objevují se pouze pojmy z problémové domény (faktura, řádek faktury). Přitom však dodržujeme pravidlo, že analýza informačního systému je modelem systému, i když jenom abstraktním modelem.

Není povoleno šíření dokumentu mimo prostor licence!

Výsledkem analýzy IS jsou dokumenty jako modely informačního systému zapsané v 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 popisující IS jsou nezávislé na implementačním prostředí a jsou platné pro každé prostředí, protože popisují pouze podstatu problému (zodpovídají otázku pouze „o co v systému jde" a nikoliv „jak to jde"). S trochou nadsázky můžeme analytický model IS považovat za „abstraktně pojatý informační systém" naprogramovaný a fungující pouze v abstraktní rovině (v pojmech).

O jednom nebezpečí při tvorbě analýzy IS – co je vlastně předmětem analýzy?

V této souvislosti je třeba ještě podotknout na časté zaměňování předmětu analýzy IS. Tato záměna, jak mohu potvrdit z konzultací, vede k následným velkým chybám v analýze informačního systému!

Předmětem analýzy informačního systému je samotný informační sytém. Protože je analýza informačního systému pouze synonymem pro model informačního systému a to pro model určitých vlastností (konceptuální model), popisuje analýza IS vlastnosti samotného informačního systému jako takového. Pokud bychom chtěli použít slova „programování", tak analýza IS má blízko k „programování" v tom smyslu, že v analýze informačního systému „programujeme" IS pouze na abstraktní úrovni bez implementačních podrobností, ale programujeme, i když abstraktně, informační systém.

Existuje ještě druhý pojem – tzv. analýza problémové domény. Tato analýza nepopisuje samotný informační systém, ale problémovou doménu a procesy v ní. V této analýze se popisuje „co se děje v samotném podniku" a nikoliv, co se děje v informačním systému.

Poznámka: Podnikem máme na mysli dané prostředí, do něhož je informační systém implementován a jemuž má sloužit

Na první pohled se může zdát, že provést a zapsat analýzu podniku je velmi výhodné, protože logicky vzato, bez znalosti toho, co se děje v podniku, nelze odvodit to, co má provádět informační systém. To je samozřejmě pravda, jenom bych rád upozornil na určitá nebezpečí vznikající při snaze provádět analýzu problémové domény maximalistickým způsobem.

Je zřejmé, že pro zápis analýzy problémové domény lze použít některé z modelů UML (například model aktivit, model sekvenční, model tříd, objektový atd.), protože předmět (realita - podnik) je objektově orientovaný, pouze s tím rozdílem, že předmětem modelu již není systém, ale samotný podnik.

Příklad: Máte za úkol dodat systém pro úřady práce. Pomocí modelů v UML zapíšete, jak to chodí na úřadu práce a teprve na základě těchto modelů vytvoříte modely informačního systému

Poznámka: Existují i další, z hlediska UML „nestandardní modely", které se pro tuto analýzu používají, například Business Process Models, Process Hierarchy, Process Threads apod.

Pro tvorbu analýzy problémové domény lze rozlišit dva základní přístupy, které můžete použít. První z nich bych nazval *maximalistický* a druhý jako *ryze pragmatický*.

Maximalistický přístup analýzy podniku

Ve spolupráci s uživatelem systému a pomocí modelů UML resp. jiných typů modelu zapíšete model podniku. Pokud je třeba, tak na základě tohoto modelu optimalizujete procesy v podniku, tj. navrhnete v této fázi „lepší variantu chodu podniku". Vytvoříte tak nový optimalizovaný chod podniku. Na základě

tohoto modelu provedete následný výběr optimální varianty funkcionality informačního systému a vznikne tak model informačního systému (konceptuální model).

Ryze pragmatický přístup analýzy podniku

Na základě konzultací s uživatelem modelujete přímo informační systém, tj. bavíte se přímo o informačním systému. Samozřejmě při těchto konzultacích se vysvětluje také, o co v dané problematice jde, ale rovnou a přímo se hned nato hledá kontext informačního systému. Zpětně z modelu IS je zřejmá a čitelná podstata chodu v podniku. Pokud je třeba, tak analýza podniku se ve složitých situacích zapisuje pouze „bokem“ jako několik málo diagramů UML (například stavový diagram, diagram aktivit apod.).

Samozřejmě oba přístupy mají své výhody a nevýhody a je věcí názoru, ke které variantě se více přiklonit. Mě osobně z konzultací vyplynul jednoznačný závěr, že ryze pragmatický přístup vede při správném použití (viz dále) efektivněji k cíli, než maximalistický přístup, který v sobě skrývá některá možná nebezpečí, s jejímiž projevy jsem se v konzultacích bohužel velmi často setkával. Těmito nebezpečími u maximalistického přístupu jsou:

- při analýze podniku nejsou striktně definovány hranice předmětu analýzy. Může se stát, že budete analyzovat spoustu věcí, které se nakonec nestanou předmětem zkoumání funkcionalit informačního systému. Tato skutečnost ještě více vynikne při snaze optimalizovat procesy podniku, což může vést k nekonečným diskusím „jak by to mohlo být ještě lepší“. Tam již hranice a tedy „stop diskusím“ neexistují ani v rámci daného podniku. Hrozí riziko, že se celý proces analýzy podniku protáhne na velmi dlouhé časové období a například tato analýza zabere mnohem více času, než dovolují náklady a časový horizont projektu tvorby IS.
- mnohdy představa, že v daném podniku namodelujeme správně procesy, se stává až nereálnou a idealistickou. Uživatel často není schopen podat ucelený obraz o procesech v podniku, protože v podniku panuje příliš velká nejednotnost, není v něm ucelený řád apod. Modelovat tyto procesy ve smyslu „jak by měly být“ může být pro uživatele příliš abstraktní. Naopak řeč o informačním systému je konkrétní a v tomto případě vede rychleji k cíli. Přitom nastává paradoxní a velmi častá situace, kdy nutnost používat informační systém zpětně ovlivní procesy podniku příznivým způsobem, protože „není zbytí“, než se informačnímu systému přizpůsobit. Zatímco není té síly, která by pracovníky donutila pracovat podle nějakých stanovených modelů podniku (metodik), zavedením informačního systému a nutností jeho použití se touto sílou stává paradoxně sám informační systém. Mnohem snáze se totiž hovoří o zavedení informačního systému, než o optimálních procesech v podniku.
- uživatel není mnohdy schopen rozlišit hranice, co je analýzou podniku a co je analýzou informačního systému. Důsledkem jsou nedorozumění v tom, co vlastně systém má umět a co již ne, když, jak tvrdí uživatel, je to přece v analýze. Navíc pokud s uživatelem spolupracujete na modelech podniku, vzniká mnohdy velmi nepříjemná situace charakterizovaná jako „příliš velký diktát uživatele“. Aniž by uživatel posoudil ekonomický přínos některých funkcionalit informačního systému a aniž by posoudil možnosti realizace těchto funkcionalit v daném projektu, snaží se o jejich prosazení z důvodů spíše osobních, než pro firmu prospěšných. Je to důsledek neznalosti náročnosti tvorby informačních systémů.
- mnohdy bývá odůvodněna tvorba podrobné analýzy podniku tvrzením, že „procesy informačního systému jsou podmnožinou procesů podniku“. Při hotové analýze podniku stačí tedy vzít její podmnožinu a máme hotovu analýzu systému. Bohužel, takto zjednodušený pohled není správný, i když to tak na první pohled vypadá. Existuje opravdu vztah mezi analýzou podniku a analýzou informačního systému velmi podobný podmnožině, ale není to přímý vztah podmnožiny. Jedná se spíše o vztah zrcadlení. Problém je v tom, že ke vztahu přistupuje mezi oběma analýzami navíc samotný kontext informačního systému, který může daný proces výrazně ovlivnit a dokonce některé procesy i přidat, některé změnit. Znamená to, že existuje zpětná vazba – zavedení IS zpětně ovlivní procesy podniku. V důsledku proces v analýze podniku vypadá jinak v kontextu samotného podniku, a jinak v kontextu po zavedení informačního systému. Tento rozdíl může vést k zavedení dalších procesů v informačním systému, které se v původní analýze nevyskytovaly a dodatečně tak ovlivnit procesy podniku.

Jednoduše řečeno – pokud víme, jak „to chodí v podniku“, nestačí toto vzít, vyškrtat přebytečné (co nebude systém podporovat) a ostatní přenést do informačního systému. Mnohé věci musíme ještě v kontextu informačního systému přidat a opravit.

Příklad: Aby byl jasný smysl předešlého odstavce, uveďme si tento příklad. Z konzultací s uživatelem vyplynulo, že dokument daného typu před odesláním musí odsouhlasit a podepsat vedoucí oddělení. Zatím se to řešilo tak, že na písemném dokumentu existuje kolonka Datum a podpis vedoucího. A nyní jak toto bude řešeno v kontextu informačního systému? Existuje několik variant:

1. obsluha pouze zaznamená do dokumentu datum podpisu, vybere ze seznamu vedoucího, který se dosadí do dokumentu. V tomto případě se jedná o pouhé zanesení informace do systému nějakou obsluhou.
2. vedoucímu se zobrazí jeho seznam dokumentů, který je zúžen pouze na „dokumenty, které mohu podepsat“. User v roli vedoucího odklepne dokument a odsouhlasí jej, čímž se zapíše k danému dokumentu jako podepsaný.
3. Podobně jako v předešlém odstavci, avšak dokument je „dovybaven“ nezaměnitelným elektronickým podpisem dané osoby, tj. podpisem vedoucího

Otázkou nyní je, který z těchto algoritmů bude vlastně zaveden v systému? Důležité je si uvědomit, že volba z těchto algoritmů možných řešení vede nejenom k jinému procesu v systému (jiná analýza informačního systému), ale navíc vede k dalším jiným dodatečným procesům, které nesouvisí s původním řešení: budeme například řešit agendu přístupových práv resp. celou agendu elektronického podpisu, což jsou zcela nové procesy jak v analýze podniku, tak v analýze systému. Znamená to, že naše volba řešení v informačním systému se logicky zpětně přenesla do analýzy procesů podniku, která se tímto také změnila. Takže představa pouhé podmnožiny procesů takto bere za své, protože existuje zpětná vazba – existuje zpětná závislost procesů podniku na kontextu řešení v informačním systému .

Na druhé straně při použití pragmatického přístupu existuje jedno nebezpečí, která je třeba mít na paměti: Protože se snažíme hovořit o konkrétním systému a nikoliv podniku (ptáme se vlastně „co by to mělo umět a přitom se ptáme proč“, můžeme chybně některé aspekty řešení zanedbat nebo dokonce opominout. Toto nebezpečí lze minimalizovat následujícím postupem:

- vytváříme úplný a konzistentní Use Case model informačního systému (viz další kapitola o Use Case modelu). Tento požadavek nám zabezpečí úplnost našeho řešení z hlediska konzistence projektu.
- tento úplný a konzistentní Use Case model se převede do formy dokumentu „Funkční specifikace produktu“ a tento dokument zákazník odsouhlasí buď jako přílohu smlouvy o dodání díla resp. jako dodatek, resp. se stává jiným odsouhlaseným dokumentem zákazníkem.
- zavádíme postup verzování již zde: Současně při tvorbě modelu IS rozhodujeme a také zapisujeme, co bude patřit do této aktuální verze informačního systému, co do příští verze apod. K tomu, aby byl tento proces efektivní, vyžaduje se zavést tzv. evidenci požadavků. Tato evidence požadavků samozřejmě nenahrazuje plně analytický model podniku a jeho optimalizaci, ale vede k vcelku efektivní práci na dalším příštím vývoji informačního systému.

Z uvedených důvodů upřednostňuji, pokud to lze, pragmatický přístup vyjádřený slovy: „Hovořme o informačním systému jako takovém“, přičemž doporučuji dodržovat předešlý výčet minimalizující určitá nebezpečí. Cílem rozhovoru analytika je potom model samotného informačního systému a nikoliv model podniku. Samozřejmě přitom se mimo jiné také hovoří o procesech podniku, zapisují se bokem

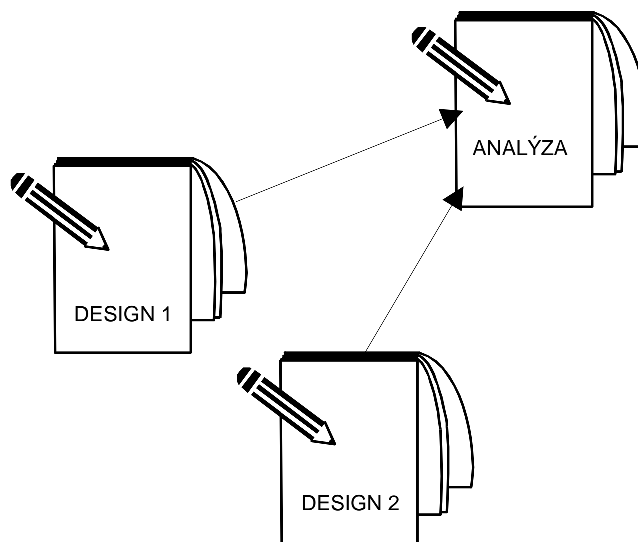
složitější případy analýzy podniku a také se evidují požadavky na další verze systému. Výsledkem jsou však vždy modely informačního systému (nikoliv podniku).

Poznámka: Při modelování procesů podniku jsem se většinou setkal s velkou rozvláčností prací nad projektem, s řešením podružných problémů nesouvisejících s informačním systémem jako takovým s následnou ztrátou koncepce vedoucí k malému „tahu" ke konečnému cíli – k tvorbě informačního systému.

Náplň designu

Práce v designu navazuje na práci v analýze. Analytický dokument se stává výchozím pro tvorbu dokumentů ve fázi designu. Designér na základě dodaných analytických modelů řeší otázku, jak bude analýza realizována konkrétně v daném prostředí, tj. jak bude systém řešen konkrétně v dané technologii. Designér převezme analytický dokument a na základě něho vytvoří dokument designu, který na analýzu navazuje, čímž vytvoří „realizaci analýzy" v technologickém detailu. Výsledkem je nový dokument, také model UML, který obsahuje analýzu v sobě uvnitř a tento nově vzniklý dokument popisuje realizaci systému.

V jiném prostředí a v jiné technologii je možné převzít tutéž analýzu a provést její „druhou implementaci do jiného prostředí". Analytická podstata problému je však stejná.



obrázek 3 : Stejná analýza, dva různé designy,kde šipky označují závislost dokumentů mezi sebou.

Jak vidět, rozdělení na analytické a design dokumenty odpovídá již zmíněnému postupu re-use sdílení. V tomto případě se operace re-use týká dokumentů analýzy - nebudeme opakovaně vytvářet analýzu při řešení stejného problému pro různá prostředí.

Důležité je (a to uvidíme při modelování v UML), že tvorba ve fázi designu navazuje na fázi analýzy bezprostředně „převzetím" analytického dokumentu, přičemž podstatným je to, že dochází k rozšíření již existujícího dokumentu o prvky designu. 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.

Není povoleno šíření dokumentu mimo prostor licence!

Příklad: V analýze se zavede třída pro objekty jako CKlient. Jedná se o klasickou konceptuální třídu. Tato třída bude existovat „stejně" i v designu, pouze designér 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, kdy design „obaluje" analýzu funkcionalitou daného prostředí.

Modelování a tvorba IS pomocí UML

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.

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 modelů v tvorbě informačních systémů.

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 systém napsaný objektově standardními prostředky, což má za následek, že pokud UML použijeme, potom našemu vyjádření modelu porozumí každý, kdo zná UML.

Z uvedené definice že „UML je pouze jazyk a nic víc" by se mohlo vcelku logicky zdát, že samotné zavedení UML tedy samo o sobě neřeší problematiku řízení projektů ve firmě resp. zavádění metodik, pracovních postupů apod. To je sice pravda - 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 nové moderního notace (což UML určitě je) vede automaticky k zavedení určitých lepších pracovních postupů.

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 podáváme přehled těchto modelů z hlediska vývojového procesu, tj. vysvětlíme si jejich

poslání a smysl ve fázích tvorby analýzy a designu. Jednotlivé modely jsou velmi detailně popsány v Příloze III.

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 - 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. 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. Někdy se tento diagram neuvádí jako zvláštní diagram, ale jako součást class diagramu (v tom případě se za počet diagramů v UML považuje 8).

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, popis sítě 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. 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.

Které jsou nezastupitelné modely

Ne všechny modely jsou si z hlediska důležitosti pro projekt rovnocenné. Některé z nich jsou dokonce mezi sebou zastupitelné. Uveďme si, které modely lze považovat za stěžejní, nezbytné a nezastupitelné, a to z jakých důvodů:

Use Case model

Je nezbytným, protože poskytuje abstraktní popis systému bez implementačních podrobností. Jeho prvky v něm a použité formulace jsou srozumitelné jak pro vývojáře („programátory"), tak pro uživatele a konzultanty neznalé vůbec nějaké teorie informačních systémů a programování (ať strukturálního nebo objektového). Tímto se stává tento model skutečným „pojítkem" celého vývoje v projektu. Navíc výstupy z něj se znovupoužijí i v jiných činnostech projektu, například pro řízení projektu, pro vyhotovení uživatelské dokumentace, pro obchodní oddělení, pro testování, atd. Tato možnost znovu použít Use Case model z projektu vývoje i pro jiné činnosti, než pouze pro samotný vývoj, je dána jeho povahou – Use Case model je totiž abstraktním a srozumitelným popisem systému, a proto jej lze použít i v jiných oblastech na první pohled vzdálených modelování IS.

Class model

Je nezbytným, protože je jako hotový model výchozím „zdrojem pro kód". Zjednodušeně řečeno, výsledný zdrojový kód je obrazem class modelu v daném programovacím jazyce, anebo obráceně řečeno, hotový celý a úplný class model je grafickou obdobou zdrojového kódu.

Vývoj v class modelu se děje po částech tzv. iterativní a inkrementální metodou (bude pospáno dále). Třídy a jejich vztahy se vždy vyvíjejí ve dvou fázích: v první fázi se vytváří model pouze analytický, tj. obsahuje pouze třídy, atributy a metody tříd jenom v rámci chápání abstraktního systému v problémové doméně, v druhé fázi se tento model doplní o podrobnosti designu, tj. služební třídy, metody a atributy poplatné danému prostředí apod.

Komponentní model

Je nezbytným, protože ukazuje rozmístění tříd do jednotlivých modulů resp. komponent.

V teorii komponent existují dva základní možné přístupy: Rozlišují se komponenty buď jako

- binární komponenty (binary component) anebo jako tzv.
- zdrojové komponenty (source component).

V prvém případě se celý systém chápe jako složen se ze „zkompilovaných propojených balíků" (což jsou binární komponenty), v druhém, případě v tzv. modulárním přístupu se části systému linkují pouze na úrovni zdrojového kódu a systém se skládá z modulů zdrojových knihoven před kompilací (zde komponenta hraje roli modulu, což oddělitelná části zdrojového kódu). Komponentní model tedy ukazuje jednak rozmístění tříd v modulech a komponentách, jednak závislosti komponent (modulů) mezi sebou.

Model je nezbytným proto, že ukazuje rozložení systému do komponent buď binárních nebo zdrojových a to ve vztahu ke třídám, které tyto komponenty před kompilací tvoří (skládají). Současně ukazuje vztah závislosti (dependency) mezi komponentami a moduly, a tím vyjadřuje nutné konfigurace skládaných systémů a určuje tak pravidla linkování částí systému.

Deployment model

Je nezbytným, protože celý již naprogramovaný systém se musí v konečném důsledku „na něčem zprovoznit". V tomto modelu se popisuje jaké budou použity stroje, jak budou zapojeny do sítě, jaké budou zdroje (procesory, apod.), disky, atd. Součástí modelu je také informace, kde bude který naprogramovaný SW nainstalován, jak bude řízen jeho chod apod.

Tento model je odpovídá k návrhu HW a rozmístění SW na něm, což je již velmi implementační záležitost a nebudeme jej zde podrobně rozebírat.

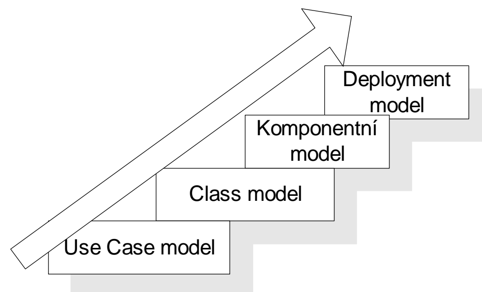
Které jsou doprovodné modely

Ostatní modely UML (sekvenční model, model aktivit, stavový model atd.) buď tvorbu těchto základních modelů podporují, tj. navedou ke správnému řešení, nebo je pomáhají upřesnit, anebo se pomocí těchto dalších modelů vytvářejí obecné vzory apod. Tyto modely bych tedy nazval spíše „doprovodnými modely" a dokonce je nemusíme považovat za povinné (i když jsou samozřejmě pro tvorbu IS přínosem).

Poznámka: Dovedu si představit projekt, ve kterém nebude ani jeden stavový diagram, ale nedovedu si představit projekt bez ani jednoho class modelu anebo bez deployment modelu

V dalších kapitolách budeme také hovořit o poloze každého z doprovodných modelů vzhledem k řízení projektu.

Důležité je, že čtyři základní modely vyjmenované v předešlé zavádějí v konečném důsledku také určitý systém postupu prací vývoje. I když každý z nich se řídí jinými pravidly pro řízení prací, lze znázornit jejich časový vztah tvorby následujícím obrázkem:



obrázek 4 Posloupnost prací na základních modelech

Lze říci, že ostatní doprovodné modely UML podporují tuto posloupnost tvorby základních modelů IS.

Kvalita SW a řízení projektů

Změny pohledů na kvalitu SW na přelomu století z hlediska řízení projektů

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“, znovu „nastartovala“ a jede se dál. Dokonce by se tento postup považoval za běžný.

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 studenti, astronomové, apod., většinou se jedná o software pro vlastní potřebu. Protože pole působnosti takového softwaru je malé, protože 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. na zakázku, podle smlouvy mezi dodavatelem a odběratelem (ekonomické systémy podniků, banky, pošty, podniky, dopravní systémy, spedice apod.). Nároky na kvalitu jsou 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 a jeho „zneprovoznění“ má pro uživatele ekonomicky katastrofické následky včetně poškození dobrého jména odběratele (například banka 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, burza apod. Selhání takovéhoto systému samozřejmě nezpůsobí pouze zvýšení nákladů, ale může mít za následek lidskou nebo společenskou katastrofu.

Je zajímavé, že v tomto žebříčku se zvyšujícími nároky se stále více uplatňuje OOP – čím větší nároky na kvalitu, tím je OOP více zastoupeno.

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 dodavatelé business systémů pro organizace nebo 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 kvalitě nebo 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 výsledku těchto postupů jedná o jednoduše pojatý a průhledný software. Problém nynějších softwarových inženýrů, kteří

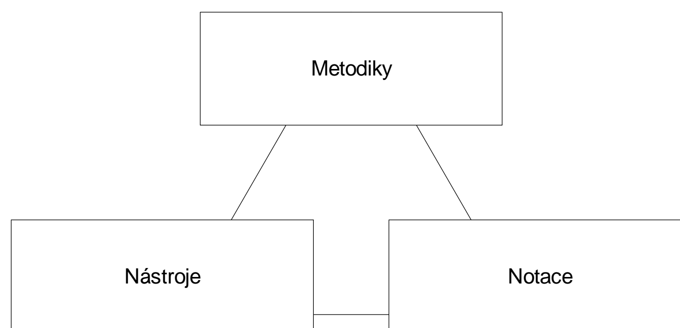
používají rychlé postupy tvorby SW, je v tom, že neumějí dělat věci jednoduchými, dokonce se věci stávají mnohem více složitějšími, než jak to tyto věci vyžadují. Technologie, která sama o sobě vyžaduje postup „tvořit SW jako jednoduchý SW" je ukryta v objektově orientovaném přístupu. Jak ukazuje praxe, prvním krokem, jak se vyhnout nebo odstranit krizi, je přechod firmy na technologii OOP s použitím komponentní technologie a samozřejmě současný přechod firmy na modelování v UML.

Ovšem v tomto kroku přechodu je ukryto velké nebezpečí a úskalí. Každá změna je riziko a dokonce platí, že každá změna i k lepšímu je vždy nejprve změnou k horšímu. Přechod na OOP, COM a UML ve firmě není totiž pouze 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 řízeným projektem.

Otázkou je, jaké kroky musí vedení firmy minimálně učinit?

Základní trojúhelník tvorby SW

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



obrázek 5 : 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 – 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, CASE nástroje apod.

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, ani pouze 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 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, 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, jaká základní pravidla při řízení projektů dodržovat. 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

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“. 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:

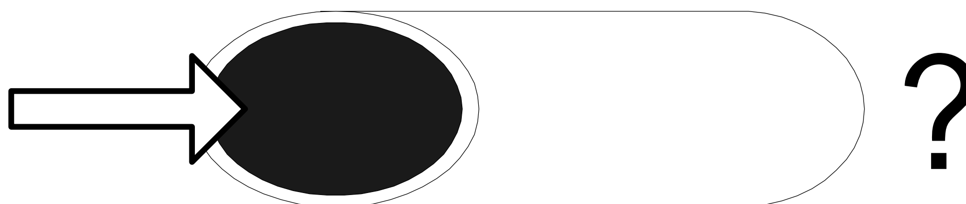
- systém, i když je rozsáhlý, je jednoduchý, protože to, co je složité a rozdělitelné na menší celky, se stává jednoduchým
- systém je srozumitelný, protože co je jednoduché, je srozumitelné
- systém je elegantní, protože co je nepřehledné, je škaredé a neelegantní
- systém má dobře definované hladiny abstrakce, uvedený pohled ve vrstvách abstrakce odpovídá systematickému pohledu přirozeného lidského chápání
- systém má jasně definovaná rozhraní - interfacy - tj. zřetelné 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 „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, lepší a horší. Uveďme si nejdůležitější z nich.

První z nich se nazývá příznačně „Metoda tunel“

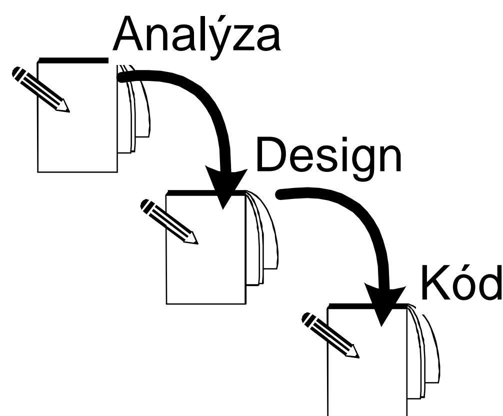


obrázek 6 : Metoda řízení projektu tunel

Není povoleno šíření dokumentu mimo prostor licence!

Její princip je velmi primitivní a bohužel také dost rozšířený - na počátku projektu se vstupuje do „tunelu“, tj. do neznáma, a podle momentální situace vedení vydává pokyny a vytváří 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. Hlavním způsobem rozložení prací je použita metoda příčného řezu, kdy se zadává pracovníkovi práce na nějakém modulu od A až do Z.

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).



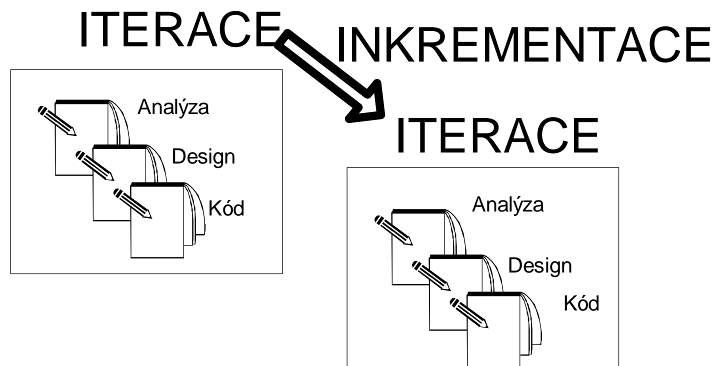
obrázek 7 Metoda řízení projektu typu vodopád

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 „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 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 věnoval mimořádnou pozornost a zakomponoval ji 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 analýzy, designu a 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 několika vodopádů částí systémů:



obrázek 8 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 hodnot 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 struktur dat). 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 iterací. 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 musíme 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 jeho konečný bod 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 hned na začátku projektu. O této výjimce budeme podrobně pojednávat v další kapitole

Jednoduché zásady řízení projektu tvorby IS v OOP a hlavní milníky 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.

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, takže kdokoliv, kdo jej vlastní, je schopen ***pouze na základě tohoto modelu*** navrhovat další modely a nakonec systém naprogramovat

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 (viz Příloha III).

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 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

Existují dva základní okruhy neznalostí při tvorbě Use Case modelu: Jako první jsou neznalosti zásadní, protože to neznalosti z problémové domény jako takové. Prostě „nevíme, co se má vlastně v dané situaci udělat“. Musíme samozřejmě tyto neznalosti odstranit například pomocí konzultanta – experta na danou problémovou doménu.

Druhý typ neznalosti se netýkají 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 i 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 staví Use Case model oproti ostatním modelům do výrazně jiného světla. 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 nejenom že tato skutečnost plně ověřuje, ale navíc můžeme 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 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 problém 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 mor všech projektů, je „bobtnání projektu“. Znáte tu situaci: Projekt jede „na plné obrátky“, ale stále přibývají další a další agendy, další útvary, a tedy další práce. Neustálé rozšiřování projektu - 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 se 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.

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 z 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".

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 nenechejte 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 co nejdříve od počátku 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. Díky vyhotovení Use Case modelu hned na začátku 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.

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 část se vynechá jenom proto, že při tvorbě modelu nejsou známy analytické informace související s daným informačním systémem. Jedná se o jednu z největších chyb v řízení projektu – problém se pouze odloží a bohužel se tento postup později mnohonásobně vymstí. Namísto toho, abychom informace o tom „jak to 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 vlastně má systém dělat". 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.

Pokud narazíme na nějaké nevyjasněnosti analýzy v Use Case modelu, tak buďme rádi, že jsme je odhalili v této fázi a ne až v programování. Zde se musí naše 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 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 se tyto argumenty vcelku logicky obrátily proti těmto námitkám: Když to dobře znáte, tak to nebude problém zapsat a to stojí za to. Doporučení tedy nakonec 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 v úvodních fázích projektu. 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.

Není povoleno šíření dokumentu mimo prostor licence!

Jediný způsob, jak zamezit tomuto velmi nepříjemnému procesu (přece jen „zákazník je zákazník“) spočívá v zavedení Use Case modelu do projektu. 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 jako funkční specifikace 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.

Představte si, že je prosazeno vytvoření Use Case modelu, který se zdárně dokončí a rozsah projektu je odsouhlasen uživatelem. Přesto se rozhodneme, že se systém v touze po dokonalosti „vylepší“. Dobře míněná snaha vede k velmi nepříjemnému důsledku: 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 již 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 - 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 příliš měnit v těchto pasážích 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 pro tvorbu uživatelské dokumentace

Je zajímavé, že na všech školeních reagovali účastníci na otázku: „Jaká je nejnepříjemnější činnost programátora?“ stejnou odpovědí: „Tvorba uživatelské dokumentace.“ Sám jsem to v některých softwarových firmách zažil osobně: Uživatelská dokumentace se tvoří jako úplně poslední (dokonce se mnohdy tiskne v noci těsně před odevzdáním produktu) a patří k jako doslova nejodpornějším činnostem programátora.

Přitom tato otázka „kdo tvoří uživatelskou dokumentaci a jak se tvoří“ je velmi vypovídající o tom, jak se firmě tvoří software. Pokud firma používá klasickou metodu příčného řezu, kdy jeden pracovník zastává roli analytika, designéra a programátora, tak musí následně psát i uživatelskou dokumentaci, protože není nikdo jiný, kdo by tuto problematiku znal. Přitom tvorba uživatelské dokumentace by neměla spadat pod roli programátora! Programátor nechť programuje!

Přitom pokud je napsán Use Case model, může uživatelskou dokumentaci psát kdokoliv, a to dokonce již v průběhu vývoje systému, pouze musí dodatečně dostat obrazovky.

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. 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. Pro tvorbu testů analytické funkcionality lze s výhodou použít Use Case model.

Použití Use Case modelu v obchodním oddělení

Podobně, jako na otázku: „Jaká je nejnepříjemnější činnost programátora“ všichni do jednoho odpovídají: „Tvorba uživatelské dokumentace“, tak na otázku: „Které oddělení nespadající pod vývoj je nejotravnějším oddělením ve vaší firmě?“ odpovídají všichni shodně: „Obchodní oddělení a to hlavně před Invexem.“ Jsou to pracovníci obchodního oddělení, kteří 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 obchodníci 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 již zaměří na to, o čem má skutečně být, tj. jaké jsou hlavní výhody, co v materiálech zdůraznit (a co naopak nezdůrazňovat), apod. Samozřejmě obchodník nevezme Use Case model, nezkopíruje jej a nepoloží jej jako nabídkový dokument na stolek na stánku veletrhu. O určitém stupni utajení Use Case modelu byla již řeč.

Jednoduchá tvorba Use Case modelu bez použití CASE nástroje

Zde si 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 Command baru)
- Vyberte šablonu pro Use Case model.
- Editujte nový dokument.

Při editaci dokumentu se řídíte těmito zásadami:

- Každá úroveň nadpisu odpovídá úrovni hierarchii kompozice Use Casů
- 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 COM, 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.

Zásady řízení projektu a jeho milníky

Pokud shrneme hlavní závěry z předešlých kapitol, potom bychom je mohli zapsat do následujících hlavních bodů:

- dokumenty vývoje IS jsou tvořeny jako modely UML (až na určité drobné výjimky v designu).

Není povoleno šíření dokumentu mimo prostor licence!

- modely jsou rozděleny na analytické modely a modely designu. Oba typy modelů na sebe navazují – design modely „obsahují“ analytické modely a „obalují“ analytické modely pomocí prvků funkcionalit dané SW technologie
- v návaznosti na předešlý bod existují specialisté analytici a specialisté designéři. Obě skupiny však pracují nad modely téhož IS a to v notaci UML. Analytici tvoří modely IS z hlediska problémové domény a jsou tedy experty jak na modelování IS, tak na danou problémovou doménu. Designéři tvoří další modely z hlediska technologie, jsou tedy také experty na jak na modelování IS, tak na použitou technologii
- existují základní čtyři nezastupitelné modely: Use Case model, model tříd, komponentní model, model rozmístění zdrojů. Ostatní modely nejsou pro projekt absolutně nezbytné, ale bývá velmi prospěšné je pro vývoj použít.
- s výjimkou Use Case modelu se analýza, design a kódování provádí iterativním a inkrementálním postupem

Milník I – Use Case model a stabilita projektu

Shrnutí předešlých bodů vede k důležitému závěru udávajícímu, jak vlastně vypadají cíle projektu ve svých fázích. Podle hlediska stability projektu (stabilita vzhledem ke změnám v projektu, jeho bobtnání atd.) se projekt vývoje IS dělí na dvě základní fáze:

- projekt před uzavřením Use Case modelu
- projekt po uzavření Use Case modelu

Projekt ve fázi před uzavřením Use Case modelu lze považovat za projekt ve stavu nestabilním, jehož podstatné části se mohou ještě měnit, projekt může v nestabilním stavu nečekaně a velmi nepříjemně nabobtnat atd. Nestabilní projekt nemá své kontury, nelze u něj odhadovat náklady, nelze určit časové proporce projektu, nelze projekt řídit atd. Kromě toho projekt před uzavřením Use Case modelu v sobě skrývá „časované bomby“. To jsou důvody, proč se vedoucí projektu snaží ve spolupráci s hlavním analytikem tuto fázi nestability pokud možno zkrátit intenzivními pracemi na Use Case modelu.

Proces uzavření Use Case modelu znamená:

- vyhotovení Use Case modelu ve své úplnosti jako konzistentní model (nechybí žádný Use Case, popisy jsou pokud možno úplné)
- odsouhlasení modelu vedoucím projektu a vedením firmy
- pokud je to možné, také odsouhlasení odběratelem informačního systému, tj. zákazníkem, pokud se jedná o systém dodávaný na zakázku

Uzavřenost Use Case modelu s sebou přináší i určitý rys projektové disciplíny: pokud je Use Case model v pořádku a odsouhlasen, a to dokonce i zákazníkem, potom by změny v tomto modelu měly probíhat podle velmi přísných pravidel a podle zásady, že každá změna je rizikem a pokud není třeba, tak změny raději neprovádět (na to jsou vyhrazeny intenzivní práce ve fázi *před uzavřením* Use Case modelu). Existují však vynucené změny, kdy není zbytí a změny provést. Jsou to případy chyb v analýze anebo neočekávaných změn samotné reality v problémové doméně (například v průběhu vývoje je vydán nový zákon apod.).

Z vlastních zkušeností jako vedoucí projektu jsem dospěl k přesvědčení, že jedním z hlavních úkolů vedoucího projektu (a následně hlavního analytika) je „udržet opratě“ projektu ve smyslu získání stability projektu, tj. zabránit bobtnání, náhlým a přitom neustálým změnám v projektu atd. Dobrým nástrojem ke splnění tohoto cíle je právě Use Case model.

Při tvorbě Use Case modelu se používají jako další podpůrné tyto modely (jejich specifikace viz příloha):

Není povoleno šíření dokumentu mimo prostor licence!

- instance (object) model, resp. konceptuální model (viz příloha, konceptuální model není součástí UML)
- stavový model
- model aktivit

Tyto modely včetně Use Case modelu se týkají pouze pojmů zavedených v analýze (reprezentanti budoucích tříd), doprovodné modely uvedené v předešlém výčtu napomáhají tvorbě Use Case modelu.

Milník II – Tvorba zbývajících nezastupitelných modelů iterativním a inkrementálním postupem

Jako další nezastupitelné modely jsou

- model tříd (Class Model)
- komponentní model (Component model)
- model rozmístění zdrojů (Deployment model)

Na rozdíl od postupu tvorby Use Case modelu tyto modely již spadají pod iterativní a inkrementální vývoj (viz předešlé kapitoly o způsobech vývoje).

Je třeba upozornit na tu skutečnost, že požadavek na „urychlenou tvorbu" Use Case modelu a jeho uzavření v žádném případě neznamená, že se má začít s těmito dalšími modely nutně až po jeho dokončení a odsouhlasení. Je možné a doporučuje se zahájit práce na dalších modelech ještě před uzavřením Use Case modelu, je však třeba počítat s tím, že je možné provést ještě následné výrazné změny v již hotových modelech (není záruka uzavřenosti problému k řešení).

Při iterativním a inkrementálním postupu se určují oblasti částečného řešení informačního systému a provede se jejich realizace následujícím způsobem:

1. vymezení oblasti k řešení pro jednu iteraci
2. vytvoří analytický class model, návrh tříd a vztahů a jejich umístění do package
3. vytvoří se odpovídající design class model (tento rozšiřuje předešlý model), návrh obrazovek, doplnění datového modelu
4. provedou se úpravy a doplnění komponentního modelu, přiřazení tříd do komponent
5. následuje tvorba prototypu v kódu
6. vymezení dalších oblastí k řešení (přírůstek)

Není povoleno šíření dokumentu mimo prostor licence!

Projekt přechodu firmy na OOP a UML

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.

Přechod na OOP má sám o sobě povahu projektu jako každého jiného projektu a to i se svými riziky, s plánováním, s termíny atd.

V tomto projektu přechodu na OOP a UML musí být bez výjimky provedeny tyto kroky:

- zaškolení pracovníků na novou technologii - na objektově orientovaný přístup a na syntaxi UML
- zavedení dvou nových stabilních rolí ve firmě
- zavedení způsobu zavedení rolí v projektech
- postupné a praktické zavádění metodických příruček pro postupy prací v projektech
- založení a spuštění pilotního projektu a poté založení dalších projektů

O těchto bodech bude pojednáno dále.

Jeden z nepříjemných faktorů 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 pouze statistický a s odchylkami. Znám případy, kdy přechod na OOP proběhl u pracovníka řádově v týdnech a znám případy, kdy neproběhl vůbec.

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 je 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 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 jeden projekt za druhým. 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 až pod 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 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)

Zaškolení pracovníků při přechodu na OOP

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

- je třeba používat externích zkušeností jako školení, literatura atd.
- je třeba nastartovat pilotní projekt v OOP (praxe je praxe...)
- 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. Díky tomuto tlaku se pracovníci prostě zaškolit musí...

Například v jedné firmě, kde chtěli zavést OOP a UML, se rozhodli, že založí souběžně dva projekty řešící tentýž problém, tedy dvě varianty - 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 objektových entit. 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 pro strukturální přístup velmi komplikovaná (pro objektovou aplikaci není toto problém díky zapouzdření) a proto se aplikace pojatá strukturálně „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 žádoucí a nezastupitelné jsou samozřejmě konzultace externistů, použití externích zdrojů, školení, knihy apod. Musím však upozornit na jedno velké svým způsobem paradoxní úskalí, 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“. 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.

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).

Není povoleno šíření dokumentu mimo prostor licence!

Role pracovníků v projektu

Jedním z kroků zavádění OOP a UML je i zavedení rolí v projektu a také ve firmě.

Základní role v projektu

Zavedení tří fází - analýzy, 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 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.

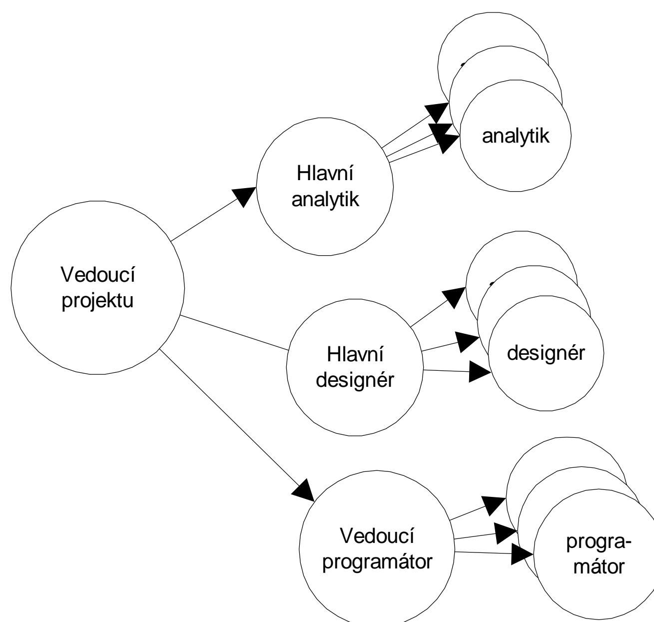
Programátor pro tvorbu svých dokumentů (zdrojový kód) využívá syntaxi daného programovacího jazyka, což neznamená, že programátor nepotřebuje znát UML. Minimálně by měl být velmi dobrým čtenářem UML, protože 90% všech dokumentů, které obdrží jako podklady ke své práci, jsou psány v UML.

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 z obou stran – jak z analýzy, tak z designu.

Týmy

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 9 Role v projektu s více pracovníky

Velmi důležité je dodržet zásadu, že:

- za analytické dokumenty je zodpovědný hlavní analytik (a nikdo jiný)
- za dokumenty designu je zodpovědný hlavní designér (a nikdo jiný)
- a za kód vedoucí programátor (a nikdo jiný)

čemuž odpovídá pohled na vrstvy v řízení projektu na předešlém obrázku. Pokud bychom tuto strukturu přirovnali ke kompetencím objektů, tak například hlavní analytik (podobně hlavní designér, vedoucí programátor) je pro vedoucího projektu „interfecem“ zodpovědným za dodání dokumentů a je záležitostí „vnitřního uspořádání práce uvnitř jeho vrstvy“, jak toho docílí. Samozřejmě hlavní analytik musí být vybaven patřičnými pravomocemi vůči svým podřízeným - tj. analytikům. Má právo práci jak zadávat, kontrolovat, 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 modely. 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. 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 duši“. 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 metodou I+I se projekt realizuje, 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 pro analýzu, jindy pro design a poté pro kódování a 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 obejití předávání dokumentů mezi dvěmi osobami. Jedna daná osoba sdílí svou vlastní „paměť a znalosti“ z jiné oblasti. Vyplývá z toho, že existuje možnost, že teoreticky se nemusí 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, neexistuje. Při sdílení osob v rolích je třeba docílit úplného „schizofrenního chování“ pracovníka 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.

Jako velmi dobrá kombinace se mi z praktických zkušeností jeví 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 s výhodou 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 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 hlavní analytik je zaměřen na abstraktní roviny projektu a designér a hlavní programátor bývá technokratem.

Stabilní role ve firmě (tj. role napříč přes projekty)

Doporučuje se, aby za účelem dobrého a efektivního řízení projektů ve firmě byli stanoveni a pracovali dlouhodobě dva pracovníci v již stabilních rolích ve firmě. Jsou jimi:

- hlavní analytik firmy
- hlavní designér firmy.

Je třeba upozornit, že zde není zde řeč přímo o rolích v jednom z projektů, ale o rolích ve firmě jako celku a to nezávisle na projektech. Charakteristika náplně činnosti těchto rolí ve firmě je následující:

- ve většině případů jsou tito pracovníci dosazování do role hlavního analytika a designéra v jednotlivých projektech, tj. slouží jako stabilní „rezervoár" těchto rolí. Pouze pokud nemohou z kapacitních důvodů tyto role v projektu obsadit, jsou do rolí hlavního analytika a hlavního designéra projektu dosazováni jiní pracovníci, než by byli tito dva.
- na základě svých zkušeností a školení vytvářejí hlavní analytik a hlavní designér firmy metodiky, sjednocují postupy tvorby analýzy a designu v celé firmě. Podle těchto metodik pracují ostatní analytici a designéři.
- kontrolují a oponují výstupy dokumentů hlavních analytiků a designérů u těch projektů, kde oni sami nevystupují jako hlavní analytici a hlavní designéři a to hlavně vzhledem k již přijatým metodikám.

Hlavním smyslem jejich činnosti je sjednocení všech prací a forem výstupů ve firmě v oblasti modelů analýzy a v oblasti modelů designu. Pokud tyto dvě stabilní role ve firmě neexistují, potom:

- těžko se zakládají nové projekty,
- obtížně se sjednocují pracovní postupy
- nesnadno se kontroluje formální správnost práce analytiků a designérů.

Zásadní priority projektu při zavádění OOP a UML ve firmě

Lze doporučit následující postup prioritních činností při zavádění OOP a UML ve firmě. Jedná se o využití iterativního a inkrementálního postupu pro tento proces zavádění OOP a UML ve firmě:

I. fáze - inicializace:

- stanovte způsob školení a kurzů OOP a UML pro pracovníky
- stanovte časový harmonogram tohoto zaškolení
- stanovte role hlavního analytika a hlavního designéra firmy, předejte jim kompetence pro zavádění metodik a pro zavádění postupů prací (procesy) v projektech

II. fáze - zavádění OOP a UML

- založte pilotní projekt středního rozsahu (odhadem cca 40 – 80 entit)
- do rolí hlavního analytika a hlavního designéra tohoto projektu dosaďte hlavního analytika a hlavního designéra firmy.
- podle principů uvedených v této knize resp. podle přílohy v této knize se pod vedením hlavního analytika a hlavního designéra vyvíjí projekt v této posloupnosti:
    - zahajte práce na Use Case modelu. Dbejte na to, aby Use Case model byl co nejdířve hotov ve své úplnosti (vyviňte tlak na vyhotovení Use Case modelu). Tvorbu Use Case modelu doprovází tvorba i jiných podpůrných modelů (viz předešlé kapitoly)
    - přibližně v 50 – 75 % hotového Use Case modelu zahajte práce na class modelu iterativním a inkrementálním způsobem od analýzy přes design až po kód. Stále však dodržte tlak na vyhotovení Use Case modelu ve své úplnosti. Při vývoji class modelu:
        - vymezte část k řešení

Není povoleno šíření dokumentu mimo prostor licence!

- hlavní analytik vyhotoví a předá řešení v class modelu pouze v analytické části doplněné o podpůrné modely v analýze – viz předešlé kapitoly
- hlavní designér doplní model o návrh, o návrh obrazovek a o datový model
- vymezená část se podle designu naprogramuje
- vymezená část se v programu otestuje jako prototyp
- takto se postupně přidáváním dalších částí vyvíjí celý projekt

- současně při vývoji projektu hlavní analytik firmy zhotoví první metodiky pro tvorbu analýzy. Zapracuje svoje praktické poznatky z prací na projektu do návodů k postupům, vypracuje návodky pro jednotné ovládání CASE nástroje a pro jednotné vyhotovení analytických modelů v UML
- podobně hlavní designér firmy provede totéž pro postupy prací v designu
- hlavní designér zahajuje práce na vyhotovení standardních vzorů řešení používaných ve firmě, se kterými se již setkal v prvním projektu

III. Fáze - další projekty

- stanovte další projekt vyvíjený v OOP a UML
- pokud nelze do rolí hlavního analytika projektu a hlavního designéra projektu dosadit pracovníky v rolích hlavního analytika a designéra firmy, určete a dosaďte jiné pracovníky. V tom případě však:
  - zaškolte tyto pracovníky
  - seznamte je s již hotovými metodikami a postupy prací
- při vývoji se postupuje stejně jako u prvního projektu, přitom výsledky prací posuzuje navíc hlavní analytik firmy a hlavní designér firmy
- pokud je třeba, zapracují na základě dalších poznatků hlavní analytik a hlavní designér další metodiky, případně opraví stávající. Metodiky se tímto nejenom rozšiřují, ale také zdokonalují. Nedoporučuje se zavádět metodiky pouze „na blind“ bez praktických zkušeností.

Výsledkem tohoto postupu je iterativní a inkrementální proces zdokonalující stav firmy k přechodu na OOP a UML. Tento proces nemá charakter ukončeného vývoje, ale vede k neustálému zdokonalování stavu výrobních procesů ve firmě, což je hlavní cíl tohoto procesu.

Příloha I – Základy OOP

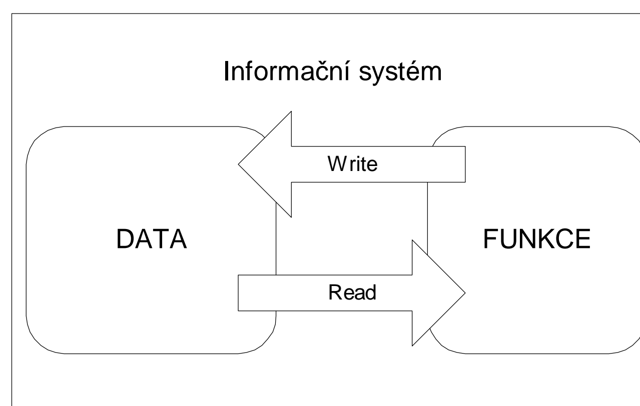
Zastaralý strukturální přístup k tvorbě SW

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, 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 pouze v době běhu systému. Pokud by teoreticky nemohl být systém nikdy vypnut, 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 obsah první oblasti - dat. Funkce pracují s daty dvěma směry do dat buď provádějí zápis anebo z dat čtení.

Uvedené rozdělení ukazuje následující obrázek:



obrázek 10: Strukturálně “ne-objektově” pojatý informační systém

Co je velmi důležité pro pochopení strukturálního pojetí a jeho nedostatků je to, že toto dělení je 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. Pokud navrhuji informační systém pomocí strukturálního pojetí, musím navrhovat pouze funkce a data, což vede k nutnosti převést model okolního 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 a to považujeme za programátorské řešení. Tento proces 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 a UML úplně zbytečně****.* Ve strukturálním pojetí se jí však nevyhneme - je vyžadována samou podstatou strukturálního pojetí, kde nic jiného než funkce a data neexistují. Tato skutečnost - 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 dat. Základní problém spočívá v tom, že naše uvažování je de facto objektově orientované a tak vidíme realitu.

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 od objektové reality ke strukturálnímu programování“ 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.

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

Pojetí informačních systémů je v objektově orientovaných systémech zapsaných pomocí UML mnohem bližší normálnímu lidskému uvažování, než jaká je klasická řeč programátora ve strukturálním programování. Pomocí OOP a UML se díky vztahu objektů reality a objektů systému získává velmi vysoká přehlednost i v informačním systému (zrcadlení se skutečností), získává se vysoká transparence systému a v neposlední řadě také vysoká logická elegance navrženého systému.

Tato skutečnost se výrazně projeví i v řízení projektů: V objektovém prostředí je díky transparenci systému a jeho logické přehlednosti odpovídající objektům pojmům zaváděn „určitý řád“ již díky podstatě věci - objektům.

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 dat 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 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.

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.).

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

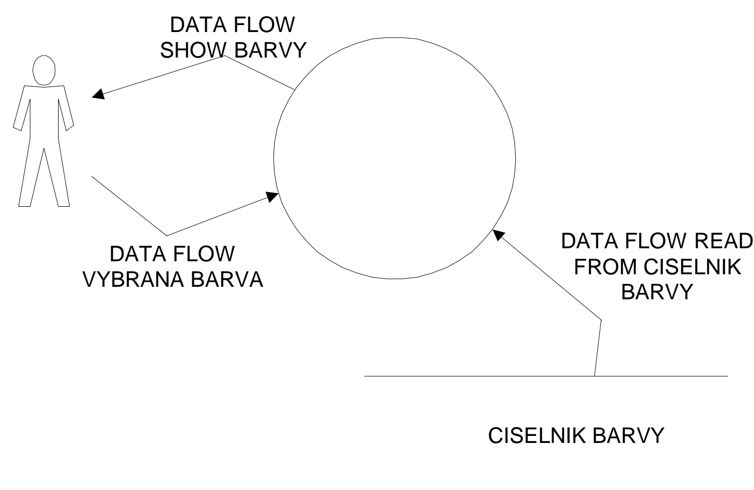
Diagramy DFD

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 (funkce) se v této škole zobrazují kružnicí , 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 11 : Ukázka Data Flow Diagramu

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

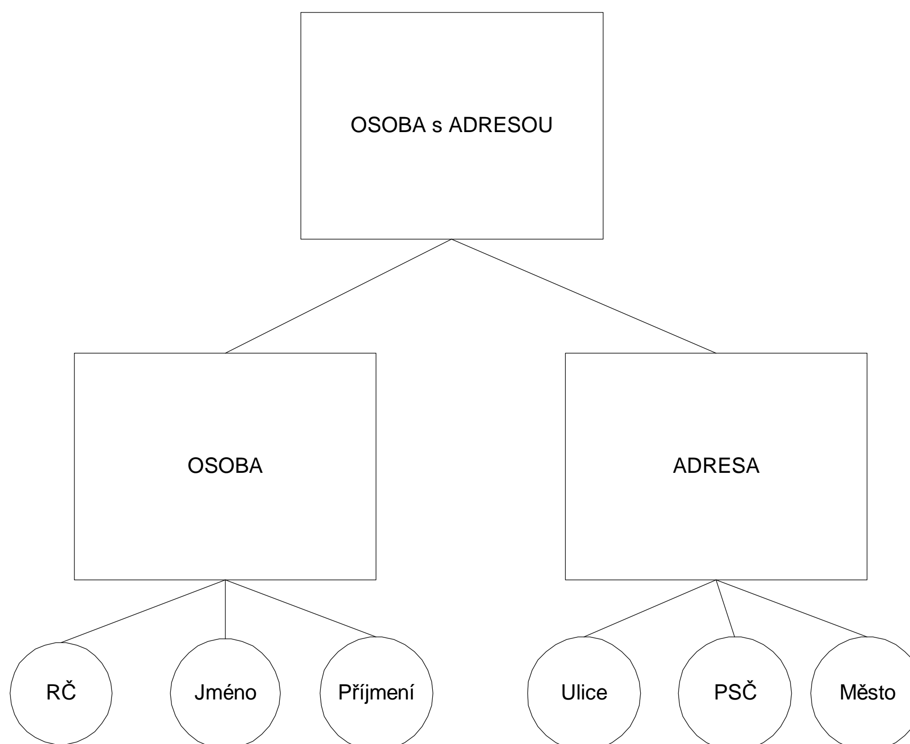
Třetí tok znamená výběr a reprezentuje tok konkrétního identifikátoru barvy (například tento datový tok obsahuje položku `id_barva`). Obsluha vybrala podle položky `nazev`, ale zpět putuje tok `id_barva`. Tento identifikátor se poté dosazuje do nějaké jiné struktury, například do dat `Auta` jako hodnota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

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 12 : Ukázka DSD diagramu. Skalár (dále nedělitelná infomrace) se v diagramu označuje kolečkem

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.

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

Objektově orientovaný přístup

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

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

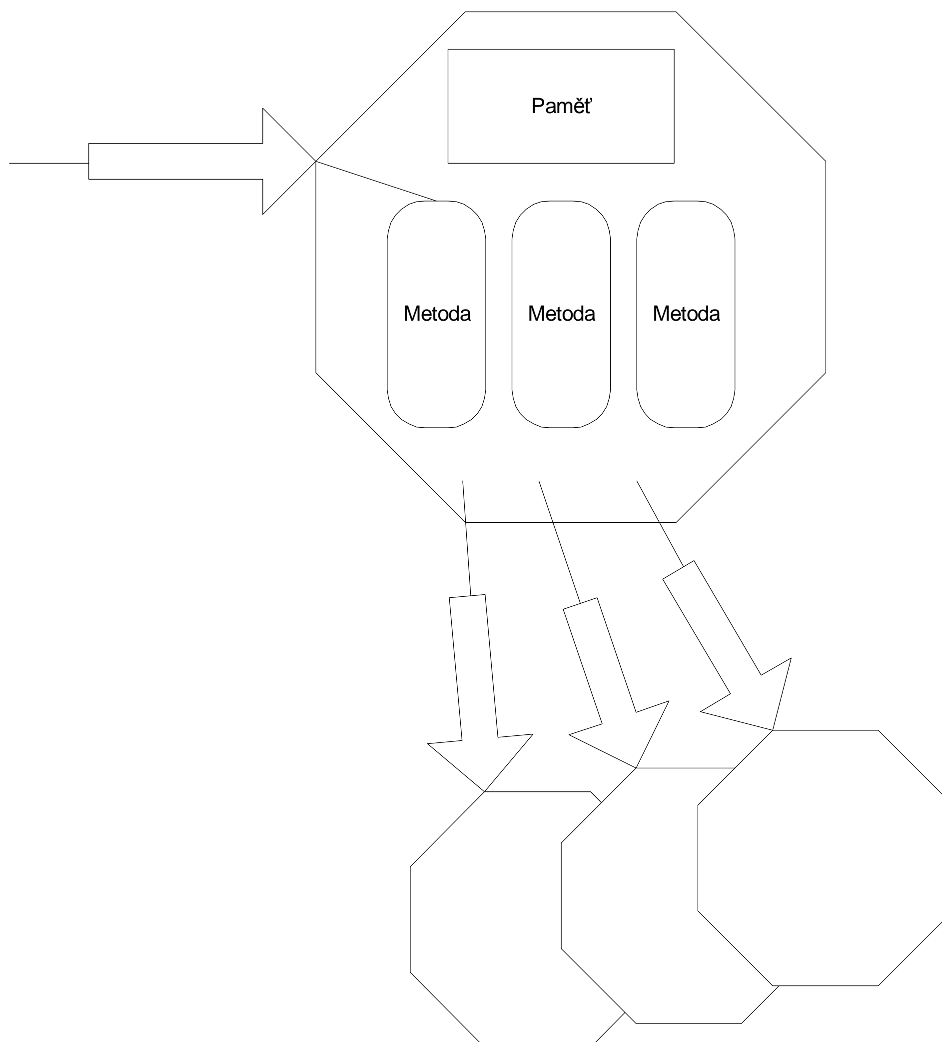
Namísto prvků DATA a FUNKCE se zavádí oproti těmto dvěma pojmům úplně odlišný prvek - objekt.

Základní vlastností objektového prostředí je existence struktur v programu (kódu anebo v binárním tvaru) majících vlastnosti *objektu*.

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 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 13 : 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).

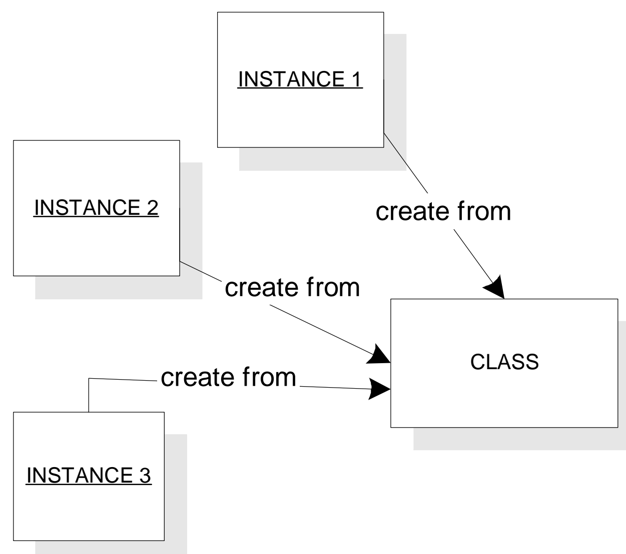
Dynamika systému je tvořena nikoliv posloupností volání a vykonání funkcí, ale postupným posíláním zpráv mezi objekty a následném vykonávání metod uvnitř objektů. Základní rozdíl oproti strukturálnímu programování je v **existenci izolovaných struktur – objektů,** přičemž každý objekt se chová jako **služebník pro jiný objekt.** Sama izolace se projevuje mechanismem zaslání zprávy objektu – objekt pro okolí neumí nic jiného, než přijmout zprávu a podle ní se zachovat. Objekty se chovají jako součástky v systému skládající systém.

Například takto se chová kalkulačka: Posílání zpráv objektu jsou „stisky kláves“ a vnitřní funkcionalita kalkulačky je nám skryta.

Třída jako objekt

Pokud budeme deklarovat dva objekty vedle sebe a bude se jednat o „dva objekty úplně stejných vlastností“ (například dvě osoby) lišící se pouze názvem (osoba1 a osoba2), potom by bylo výhodné deklarovat jejich vlastnosti (osoba má rodné číslo) v jedné definici. 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*, tedy definovat *jednou* kopyto pro budoucí objekty a můžeme jich poté definovat kolik chceme. Nebudeme se v definici opakovat, pouze se odkážeme, že objekt je z této třídy, čímž je dáno, jak je definován (z jaké šablony).

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 14 : 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“.

Třída jako vyšší forma abstrakce

Zavedením třídy vzniká nová vyšší abstrakce. Pokud zavedeme třídu jako kopyto pro budoucí instance třídy, tak vlastně definujeme vlastnosti pro **každou instanci, a to i těch, které zatím v systému neexistují**. A to je již jiná situace, než když mluvíme o jedné jediné instanci.

Příklad

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

Není povoleno šíření dokumentu mimo prostor licence!

Při modelování se musí 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

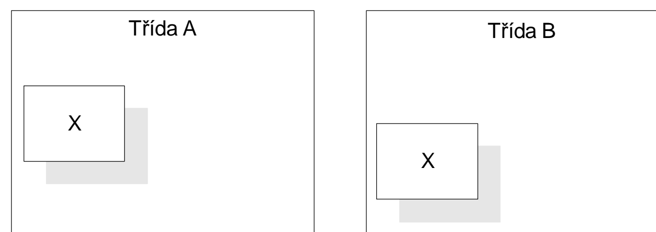
Představme si tu situaci, kdy dva různé objekty mají ve svém protokolu stejnou zprávu, ale každý z nich na ni reaguje jinou metodou. Znamená to, že oba objekty mají ve svém protokolu zpráv stejnou zprávu, ale každý z nich má k nim přiřazenu jinou metodu. Pokud je možné oběma objektům poslat tutéž zprávu, 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 (například zavolá se „hoří!“).

Dědění v OOP

Při definici tříd může nastat situace, kdy se některé části definic z různých tříd budou opakovat. 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 a dojde ke sdílení definice.

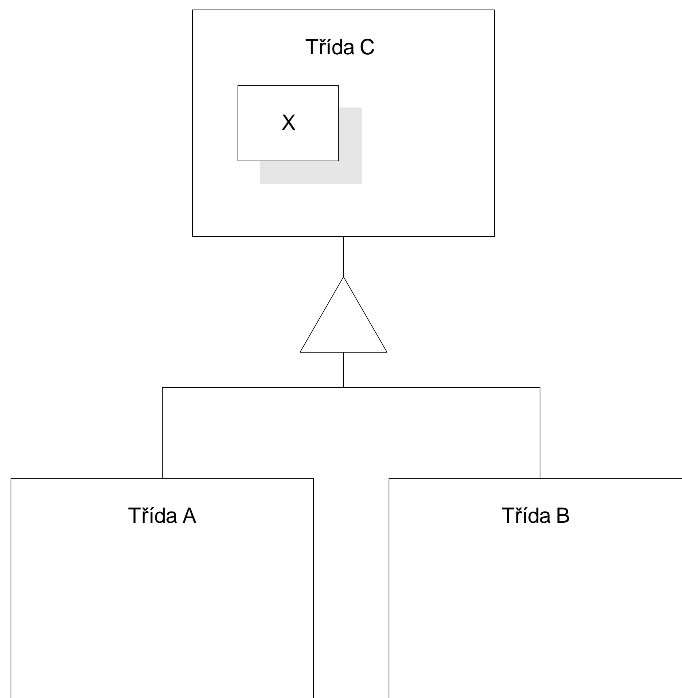
Tato interakce se nazývá dědění. Je zřejmé, že tato interakce je jednosměrná – jedna třída (tzv. potomek) se odvolává na jinou třídu (tzv. předka) a tu použije ke své definici jako předlohu.

Zavedení vztahu dědění ukazují následující obrázky:



obrázek 15 Redundance X při definicích tříd A a B s některými společnými vlastnostmi

Redundance v definicích tříd se odstraní děděním:



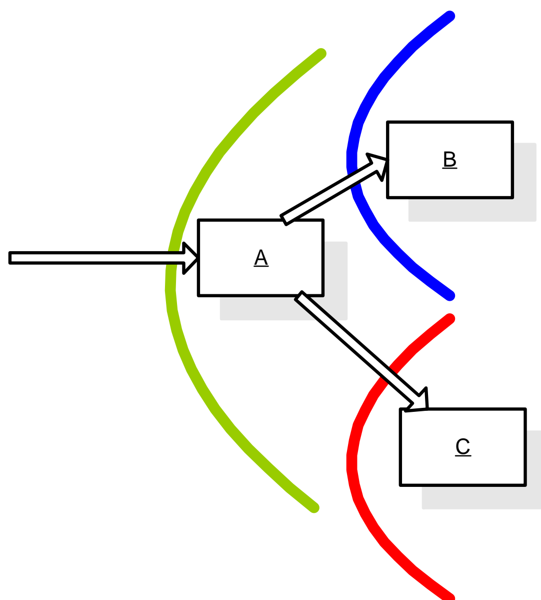
obrázek 16 Dědění zavádí re-use mezi třídami sdílením předka, třída A i třída B sdílí definici X

Objekty skládající systém

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. Pokud držím určitý objekt, označme jej jako A, (mám objektovou referenci), mohu 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**.

Princip skládání objektů jako zapouzdřených vnořených struktur do sebe s sebou 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 17 : Rozvrstvení systému podle viditelnosti v rámci objektů

Na obrázku je patrné vymezení informační oblasti *A* „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 důsledně izolovány, tj. „dovnitř vrstev“ uživatelem objektu vidět. Z tohoto důvodu se na každé této vrstvě se 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 vystižení „všech objektů v systému se vztahy vnoření spolu se zprávami“ můžeme považovat za celkový a úplný popis dynamické funkcionalitu systému.

Příloha II

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 - kdoví, 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, 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 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 typ autoincrement (IDENTITY, 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 obraz v databázi, například těsně po svém zrodu. Učiňme dohodu, že objekt, který nemá svůj obraz v DB, bude mít OID rovno -1 a ten objekt, který bude mít svůj obraz v DB, bude mít OID rovno přímo hodnotě ID v tabulce (tj. kladné hodnoty).

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

- Zrod objektu. OID implicitně rovno -1
- Vyplnění formuláře uživatelem a poté objektu
- Zavolání metody INSERT objektu, požádání objektu datové vrstvy o INSERT s návratovou hodnotou IDENTITY

- 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 svůj obraz vůči odloženým datům.

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 rušit tak 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 naruší jinak logické chování objektu a 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 a proto je o mnoho stabilnější a mnohem snadněji se vyvíjí.

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 - 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“. Jak 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 několika-násobnou 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 Excelu, dostanete tak k dispozici **úplný** Excel objekt, tj. úplný ve své funkcionalitě. Není nic mimo něj, co byste ještě museli linkovat. Vytvoříte jednu instanci a máte k dispozici celý Excel. Pokud pracujete s jedním listem spreadsheetu Excelu, potom dostanete k dispozici všechnu jeho informaci 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.

Příloha III - Modely UML

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 - 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).

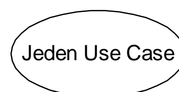
Jiný příklad: Naopak může nastat situace, kdy něco je obsaženo v systému a není dostupné v okolí (- / +). Obsluha potřebuje získat přehled o bonitě klienta banky (-) a údaj je v systému (+). Po vyžádání informace se deficit vyrovná (0 / 0).

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 - systém.

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 18 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 z technického hlediska přijatelný (teoreticky obsahuje celý popis systému z hlediska užitných činností).

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

Postup je 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

Není povoleno šíření dokumentu mimo prostor licence!

logicky na několik úrovní dalších podřízených souhrnů užitných činností o úroveň níže. A tak postupujeme stále dolů rozpadem 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.

Druhý způsob nepoužívá mechanismus package. Tento přístup rozlišuje Use Casy, které jsou „hierarchickou skladbou" jiných Use Casů. Existuje tedy rozklad samotných Use Casů směrem dolů. Rozeznáváme Use Casy, které mají povahu souhrnu užitných činností (tj. Use Casy, které se dále člení na další) 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:

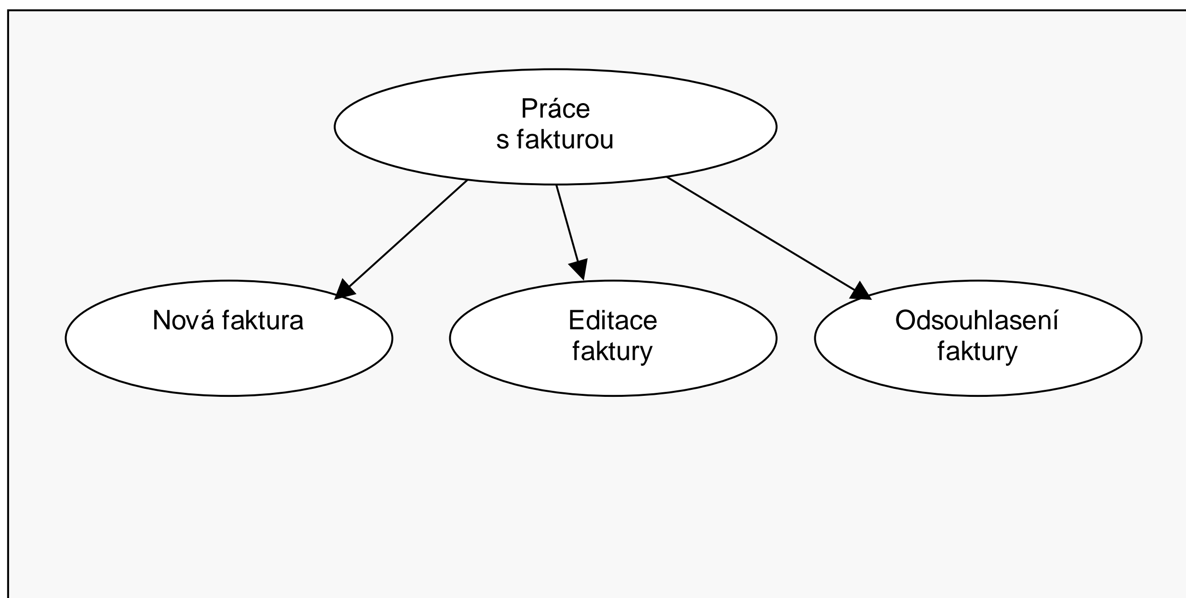
- Jaké jsou úkoly aplikace?
- Bude se z okolí přidávat, ubírat nebo měnit informaci v systému a jaké?
- Bude okolí potřebovat informace o změnách v systému a jaké?
- Jaké změny v okolí systému povedou k přísunu informací do systému?
- Jaké činnosti jsou třeba k administraci systému?
- apod.

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

Mezi jednotlivými Use Casy – užitnými činnostmi se mohou vyskytovat jednosměrné interakce – vztahy mezi Use Casy. Tyto vztahy jsou v UML chápány jako další model elementy a jejich visual elementem je šipka.

Jednou z těchto interakcí je již zavedená hierarchie od elementu use case „nahoře“ k elementu „dole“. 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:



obrázek 19 : 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.

Důležité zásady pro tvorbu Use Case modelu

Mějte vždy na paměti, že hlavním cílem Use Case modelu je nalézt všechny (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. 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 důelžité 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 dobře fungoval, 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 začnou tvořit model u nejnepodstatnějších a vedlůejších Use Casu.

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. Hierarchie je určitým pohledem analytika na daný systém a tne hodně subjektivní. Je třeba si uvědomit, že pokud daný systém bude zkoumat jiný analytik, s největší pravděpodobností u téhož systému navrhne jinou hierarchii rozkladu užitných činností. Výsledek rozkladu totiž závisí na pohledu každého z nás. Avšak úplná libovůle zde neplatí: Oba Use Case modely téhož systému se musí shodnout v celkovém výčtu seznamu koncových listů a ten musí být vždy stejný. 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 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 Use Case modelů 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 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 - list 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 - 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.

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 „Include“ mezi Use Case a první vyhledávání re-use v systému

Mezi dvěma Use Casy může nastat z analýzy vypozorovaná jednosměrná interakce, která se nazývá **include.** V předešlých verzích UML se tato interakce nazývala *uses*. Include je interakcí, která již v počátečních fázích analýzy odhaluje re-use v systému.

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 v C++ nebo v ASP).

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

Visual elementem include je šipka spolu se stereotypem označeným takto: <<include>>, ve starší verzi UML jako <<uses>>.

Příklad

Jako klasický příklad použití <<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 include. Znamená to, že užitná činnost Nová osoba je užita (tj. includována) do užitné činnosti Nový úvěr, což si můžeme představit podobně jako u volání funkce.

Je třeba hned na začátku zdůraznit, že vztah include je „zapouzdřen“ dovnitř užitné činnosti, která používá jiný Use Case. Znamená to, že když Nový úvěr má 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 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 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 include, 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**.

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 include je z hlediska teorie modelování sice „podstatný“, ale z praktického hlediska (lépe řečeno pragmatického hlediska), záměnu include 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...

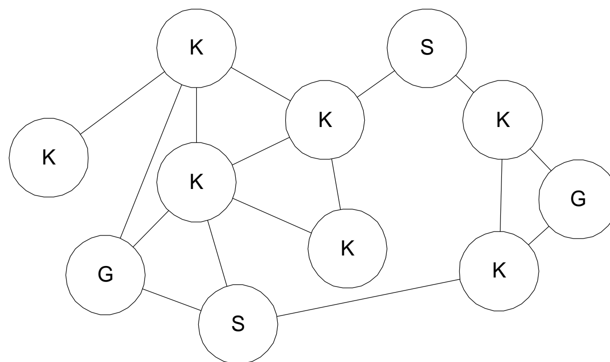
Jinak řečeno, tvrdím, že záměna extends za include při tvorbě Use Case modelu je sice chybou, ale domnívám se, že to není fatální chyba modelu. Z čistě praktického hlediska doporučuji následující postup šetřící čas a náklady:

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

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

Vztah extends 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 20 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.

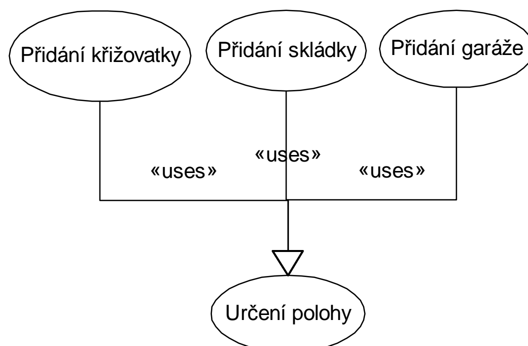
Užitná činnost přidání nové skládky

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 include. Každá z těchto tří užitných činností by měla „odskok“ include na nějakou sdílenou užitnou činnost. Nazvěme tuto společnou činnost v tomto prvním přiblížení k řešení (pozor - které není úplně přesné) jako *Určení polohy*.



obrázek 21 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)

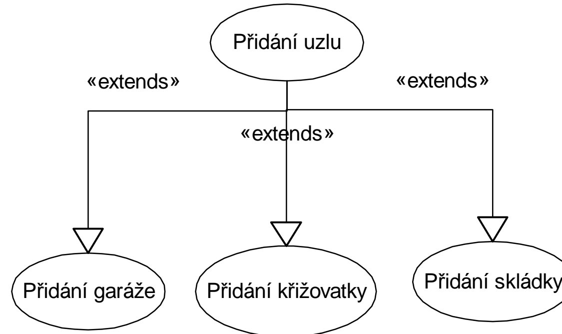
můžeme nyní vidět nikoliv jako *Udání polohy*, ale jako samotné *Přidání uzlu* (bez ohledu na to, zda se jedná o Křižovatku, Garáž nebo Skládku). Zavedeme tedy užitnou činnost *Přidání uzlu* a změníme pohled:

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



obrázek 22 Přesnější vyjádření re-use v předešlém příkladu

Všimněme si, že extends má ve smyslu re-use opačný směr, než include.

Zdůrazněme, že celá užitná činnost *Přidání křižovatky* se musí číst i spolu se všemi užitnými činnostmi, které ona sama extenduje (obráceně od šipky zpět k „obecnějšímu“ Use Case).

Rozdíl mezi extends a include si nejlépe uvědomíme na uvedeném příkladu při této úvaze: Pokud bychom vztah extends nezavedli a ponechali pouze 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.

Osobně doporučuji nehloubat nad rozdíly extends a 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 rozlišováním extends a include 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 include na 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, vlastně se 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, 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 se autor snaží neopakovat pojmy a používá 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 systémů 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 interaguje s ním:

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 a 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 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, jak 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!

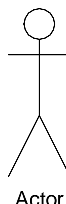
Pokud vytvoříte Use Case model a určíte v něm nepřesně actory a určíte přesně užitné činnosti, což je velmi častý (a mnohdy výhodný praktický případ), potom výsledný systém naprogramovaný podle tohoto modelu bude správný. Pokud dobře určíte actory a nepřesně určíte užitné činnosti, potom systém bude 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), 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. 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 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 – 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 23 Visual element actora

Současně s označením se do modelu také uvádí stručný a výstižný popis actoru.

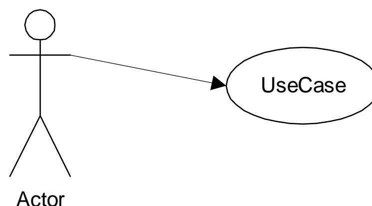
Jak se vyhledává actor

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



obrázek 24 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 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 chyba tohoto problému byla, 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 logisitiky 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 – 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 onen 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.*

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 a 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.

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 modelu? Co je opravdu nezbytné zhotovit v Class modelu a to v každém případě:

- zavést všechny třídy
- zavést všechny vztahy mezi třídami
- zavést všechny atributy
- 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).

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 – class v modelu je symbolu 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.

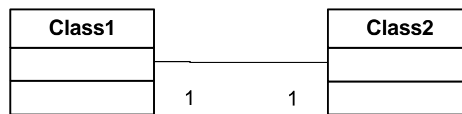
Class
-attributes
+operations()

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 25 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.

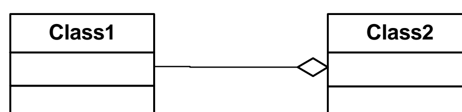
Agregace a kompozice

Daná asociace buď agregací je anebo nikoliv, což znamená, že je vztahem celek – část anebo nikoliv Pokud je agregací, může se jednat podle syntaxe UML buďto o tzv. „shared agregaci“ anebo o kompozici (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“ (majitelství). Navíc kompozice je natolik silná agregace, že nadřízený prvek nemá smysl bez svého podřízeného prvku.

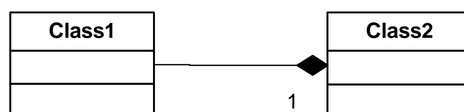
Poznámka: Z předešlého výkladu objektového modelování jsme zavedli pouze běžnou asociaci a agregaci. Zavedení kompozice toto dělení „zjemňuje v tom smyslu“, že jsme v předešlých úvahách uvažovali o agregaci jako vždy kompozici a to ostatní bylo běžnou asociací. UML dovoluje ještě něco mezi tím – shared 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 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 26 Agregace zvaná v UML jako shared

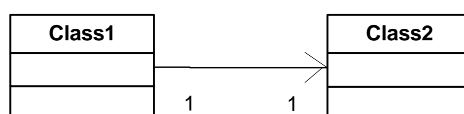
a pokud vyplněn je, jedná se o kompozici:



obrázek 27 Agregace zvaná v UML jako kompozice - unshared

Směr asociace

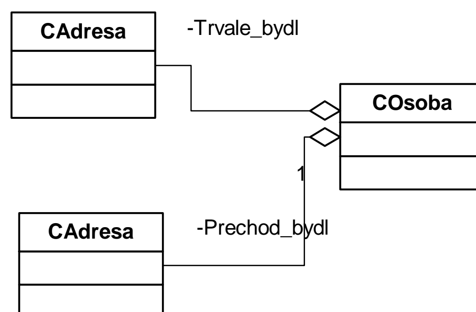
U asociací lze rozlišovat směr, který se v diagramu může také vyznačit šipkou:



Pokud šipku nezavedeme, je asociace chápána jako obousměrný vztah (jako dva jednosměrné vztahy vůči sobě). U agregace není třeba směr vyznačovat – vztah je automaticky asymetrický.

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:



obrázek 28 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

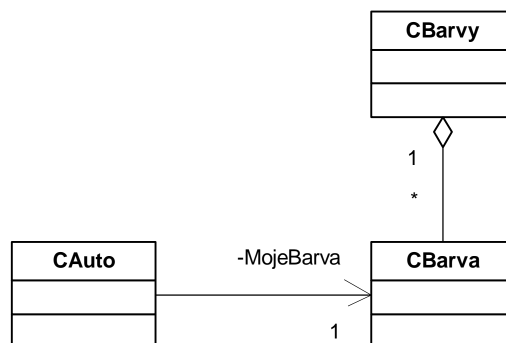
jedna nebo žádná	0..1	
od určitého čísla po určité číslo	M..N	(například od 2..4)
od nuly po libovolné kladné číslo	*	
od nuly po libovolné kladné číslo	0..*	
od jedné po libovolné kladné číslo	1..*	
apod.		

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 a natožpak 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 a jiné objekty, které je používají pro vyjádření svých vlastností, jej asociují:



obrázek 29 Použití číselníku jako klasická běžná asociace

V tomto příkladu si Auto ukazuje na svou Barvu.

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.

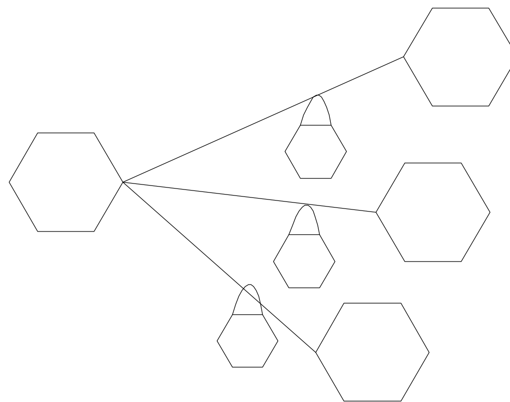
Kdy je bezpodmínečně nutné zavést asociativní třídu?

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 30 Analytická situace vedoucí k nutnosti zavést asociativní třídu

Tato skutečnost vede ke vzniku nového pojmu a tedy nové třídy – 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).

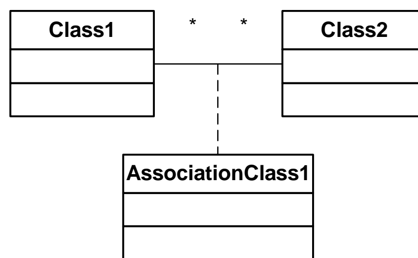
Toto je první krok analýzy. Provedeme druhý krok: Podle pravidla o vztahu M ku N bychom měli zavést automaticky asociativní třídu, která k tomuto vztahu patří a současně uvážit, jaké (a zda) budou v této třídě další informace. V tomto případě tuto třídu nazveme Majitelství. Je třeba si uvědomit, že od 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?

Visual prvek pro asociativní třídu

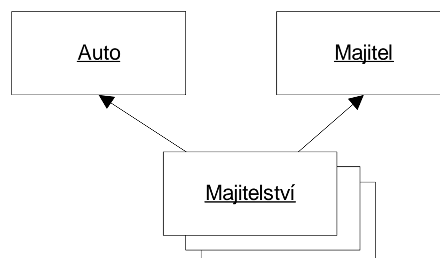
Asociativní třída je třídou, která má je asociací a v UML se značí takto:



obrázek 31 Asociativní třída

Hlavní význam asociativní třídy

Podívejme se z na zavedení asociativní třídy z hlediska konceptuálního diagramu, který ukazuje objektové reference:



obrázek 32 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 zúžit na seznam podle jednoho auta a také tak podle jednoho majitele. Tedy tento seznam musí „umět být“ anebo „vydat ze sebe“

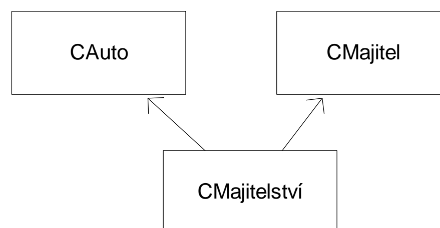
seznam majitelství pouze pro jedno auto anebo seznam pro jednoho majitele. Tímto „uměním“ seznamu majitelství se může získat omezený seznam majitelství pro majitele a tím seznam aut pro daného majitele. Anebo naopak můžeme získat omezený seznam majitelství podle auta a tím seznam majitelů pro dané auto. 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.

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 33 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 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 jednoho majitele.

Poznámka: Mnohdy nastane situace, kdy nalezneme nejprve vztah podle předešlého odstavce a obrázku pomocí běžné asociace a teprve po určité době se nahradí syntaxí asociativní třídy. 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.

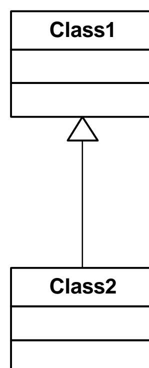
Příklad: Vraťme se k příkladu řešení systému Optimalizace svozu odpadu, kde se zavedly třídy Uzel, Úsek (jako spojnice mezi uzly). Dovedli byste zde určit asociativní třídu? Dovedli byste určit i druhou syntaxi podobnou asociativní třídě? O odpověď, pokud bude třeba, si budete moci požádat v diskusním fóru na stránkách http://www.objects.cz

Není povoleno šíření dokumentu mimo prostor licence!

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 – 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.

Visual elementem vztahu generalizace specializace (též ve zkratce gen-spec) je spojnice mezi třídami s trojúhelníkem ve vrcholu k třídě obecnější:



obrázek 34 Vztah gen - spec

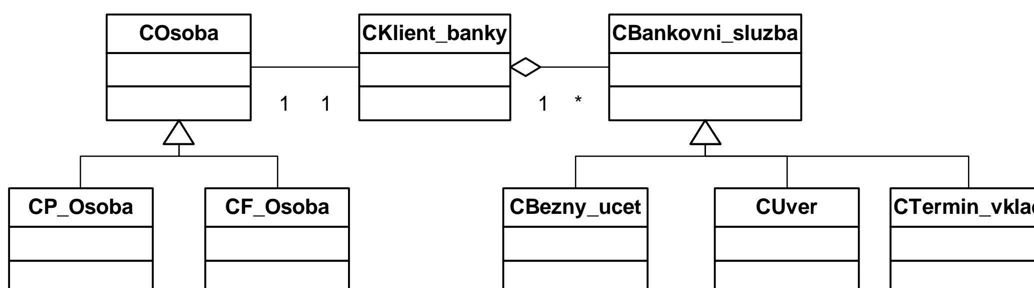
Rozdíl mezi vztahem generalizace - specializace a děděním lze jednoduše vystihnout slovy: Vypozorovaný vztah generalizace specializace v analýze (tj. který existuje reálně v pozorovaném analytickém modelu a nezávisle na prostředí, v jakém programujeme) lze implementovat různými způsoby, z nichž nejjednodušší 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 různé případy téhož obecného řešení.

Příklad:

Všimněme si blíže následujícího obrázku:

Není povoleno šíření dokumentu mimo prostor licence!



obrázek 35 Gen spec a dvě úrovně abstrakce

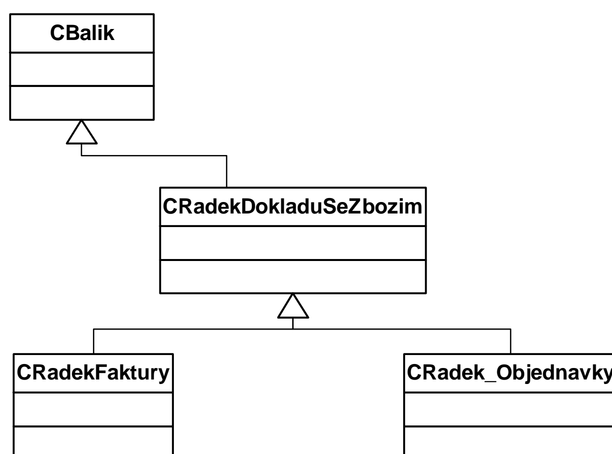
V tomto návrhu existují dvě roviny řešení. Ta abstraktnější vyjadřuje, že „obecný“ klient může mít „obecné“ bankovní služby, že je „vidí“ svou osobu. Druhá, nižší a speciálnější úroveň nám říká, že Osoba může být buď právnickou osobou (zkratka P), anebo fyzickou osobou (zkratka F) a že bankovní služba může být buď bežný účet nebo úvěr nebo termínovaný vklad.

Všimněme si možnosti přidat další možné typy osob anebo typy služeb do spodní úrovně.

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 jsem, že se mnohé údaje v Řádku faktury a Řádku objednávky a podobných řádcích se opakují. Zavedli jsme jeden univerzálně pojatý Balík zboží a vložili jej do Řádku faktury. Všimněme si, že jsme re-use pro Balík zboží vyřešili pomocí vložení instance téhož typu 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 36 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 - kolekce. Namísto množného čísla použijeme raději ustálený výraz *Seznam* (*Seznam Osoba 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, příp. odvozené entity jako seznamy apod. 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 analýzy!

Důležité pro další vyhledávání tříd je také současné rozpoznání relací - 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. kandidáta, informaci), 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 druhého 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, jaké role budou mít informace **A, B, C, D...**) . Postupně probíráme jednotlivé kandidáty z problémové domény a dáváme jim v objektové analýze jednotlivé role. Začínáme nejprve vyhledávat ty kandidáty, kteří mohou existovat sami o sobě, aniž by jako pojmy skládaly jiné kandidáty. Má smysl o nich hovořit bez ohledu na ostatní kandidáty. Tito kandidáti jsou „nejžhavějšími“ pro to, aby se stali třídami.

První možnost je, že pojem - 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 - třídu (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. Jedna důležitá zásada – atributy tříd, které mají vzniknout z pojmů, nemohou mít násobnou kardinalitu, tj. násobná vazba automaticky vede ke vzniku alespoň jedné třídy (někdy i více, kolekce, asociativní třída atd.). Zkoumáme vztahy ve smyslu „vidí – dosazuje se 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. Je třeba si uvědomit, že atribut při své definici musí vždy patřit do určité třídy (ze které pocházejí objekty s daným atributem), tedy určení „tento kandidát nebude třídou, ale atributem" znamená identifikovat také třídu, do níž 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 objektů z této třídy. Důležitá je jedna věc: Naše rozhodnutí nemusí být konečné. Každý atribut je stále kandidátem na vznik třídy. Může se stát, že tento náš první návrh (kandidát bude atribut) nakonec opustíme a například navrhneme existenci třídy *Rodných čísel* a *Majitel* bude obsahovat 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 - kontrola na modulo 11, nebo výdej Rodného čísla v různém formátu. Máme dvě varianty - buď bude *Rodné číslo* umět samo modulo 11 vydávat různé formáty, nebo to bude umět *Majitel Rodného čísla*.

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 (kandidátů). 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 - 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.

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)

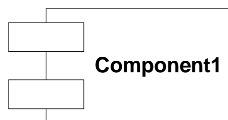
V obou případech se jedná z hlediska UML o komponentu, i když většinou se v literatuře chápe komponenta pouze ve smyslu *binary component*. Ve většině případech budeme mít také v této knize na mysli komponentu ve smyslu binary component, pokud nebude uvedeno jinak. Některá pravidla však platí pro oba 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



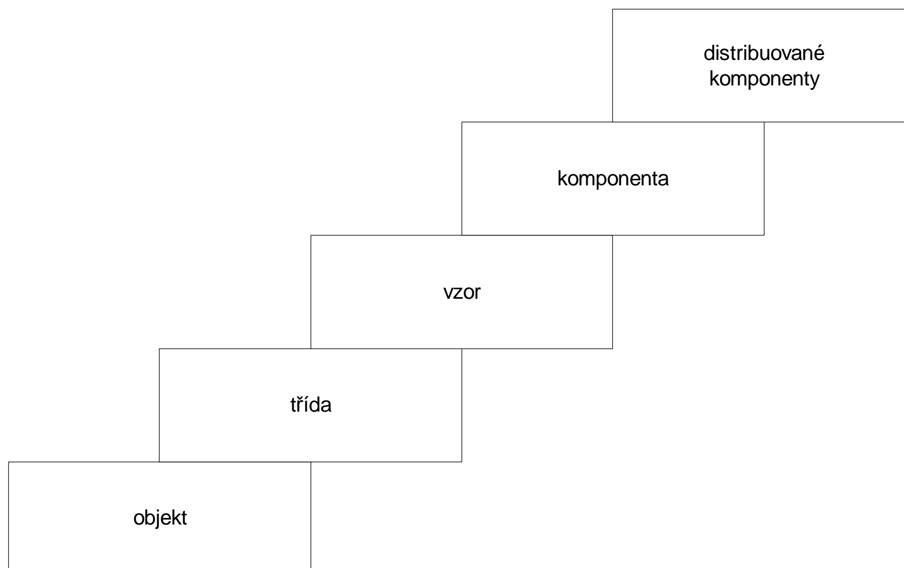
Vysvětlení komponent pomocí stupně re-use v softwaru

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 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 37 Stupně re-use v sw

Na obrázku jsou znázorněny jednotlivé stupně re-use ve tvorbě software, které vedou až k nejvyšším – 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 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 zaveden.

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 - 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 - stačí jedna definice třídy.

Použitím třídy je prolomena hranice mezi objekty – instancemi. Pro druhou instanci nemusíme „samostatně“ zavést novou další definici.

Třetí schod: re-use pomocí dědění

Při definici tříd však 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

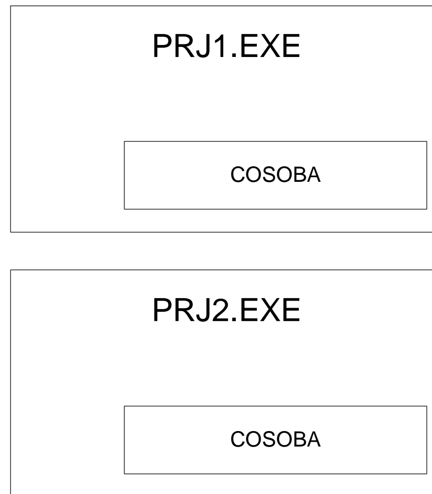
Dalším typem re-use je tzv. vzor. Vzor lze chápat jako použití řešení v různých kontextech podle určitých pravidel - tj. vzoru. Teorie vzoru je ve tvorbě SW poměrně dost složitá, protože se díky abstrakci tvorby SW mnohdy těžko vyjadřuje. Samotná problematika vzorů není předmětem této knihy a vydá na celou novou 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.

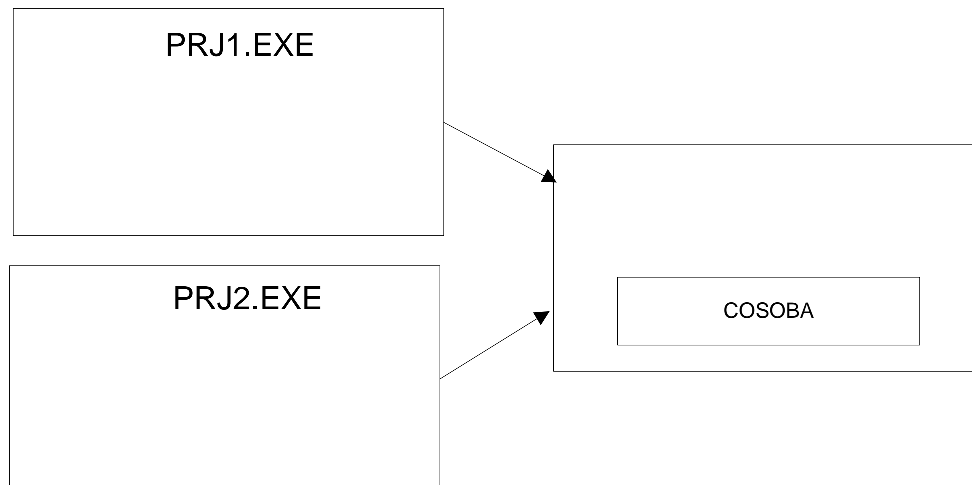
Představme si, že vytváříme projekt a v něm zavedeme třídu osob a nazvěme ji COsoba. V daném projektu tuto třídu použijeme. Zkompilujeme projekt a vznikne spustitelný soubor PRJ1.EXE. Můžeme si představit, že uvnitř zkompilovaného kódu existuje nějak zabudovaná zkompilovaná třída COsoba, kterou samozřejmě 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.

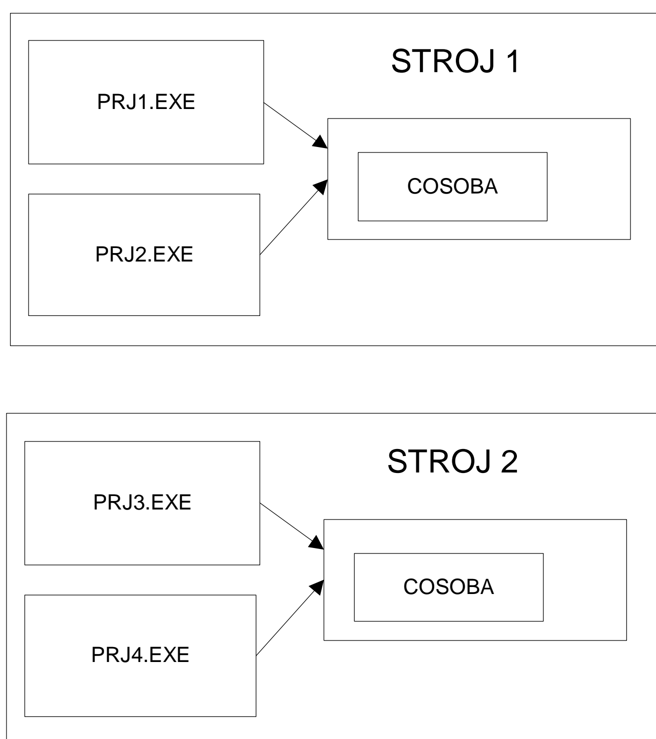
Proražením této hranice vznikne re-use na binární úrovni, tak zvaný binary re-use:



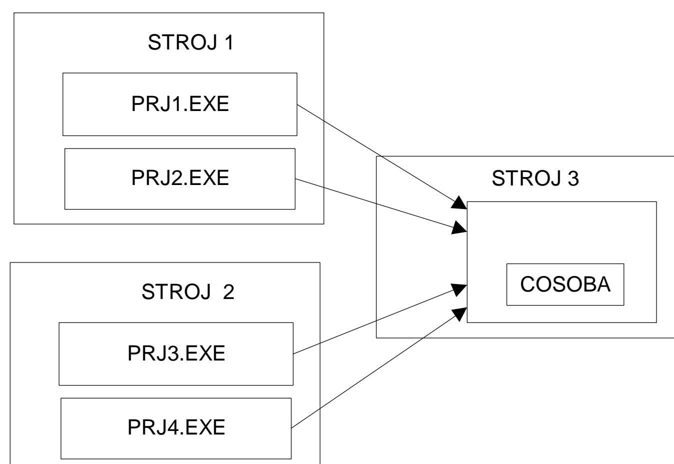
Připojení již hotových binárních balíků objektů je základem tzv. komponentní technologie. Systém se již neskládá z monolitických velkých „souborů“, ale z komponent.

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 typ – 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 – instance komponent různých vlastností. Jedna komponenta může tedy vydávat několikero různých objektů – svých instancí.

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 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, a umožňuje tak 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 „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). Tak je tento 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, v dědění se situace velmi zjednoduší (např. stačí mít jednoho předka a tím se dědí celý strom)

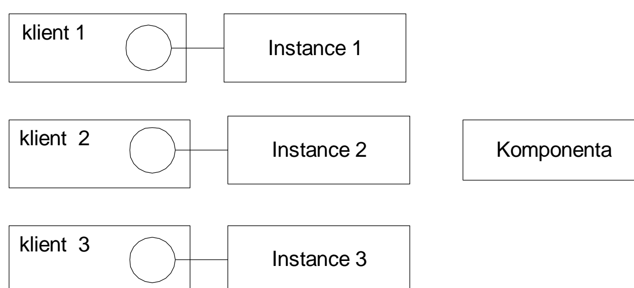
Návrh komponent v informačním systému

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 instance komponenty. Komponenta se tímto chová stejně, jako bychom si přilinkovali klasický zdrojový kód objektového modulu, pouze je v binárním tvaru.

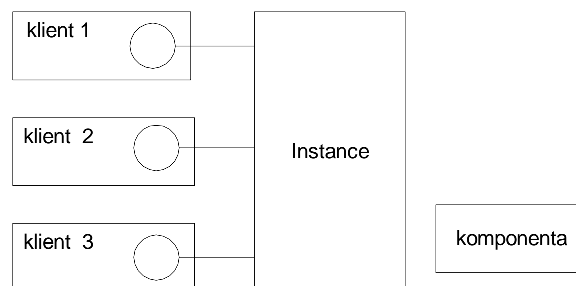


obrázek 38 Komponenta vytvořená v módu izolovaných klientů

Klasickým příkladem takovéto komponenty je například komponenta technologie ADO s poskytnutými třídami a 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 39 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:

- 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 - multithreading komponenty (pro ActiveX EXE pod VB 6.0 velmi obtížné, spíše nemožné vůbec řešit)
- Řešíte problém ztráty identity klientů databáze - connections se zúžil na jeden connection komponenty a pokud v databázi používáte k něčemu uživatele (user connection), nastává kolize – existuje pouze jeden uživatel.

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.

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

Z uvedených zásad přímo vyplývá:

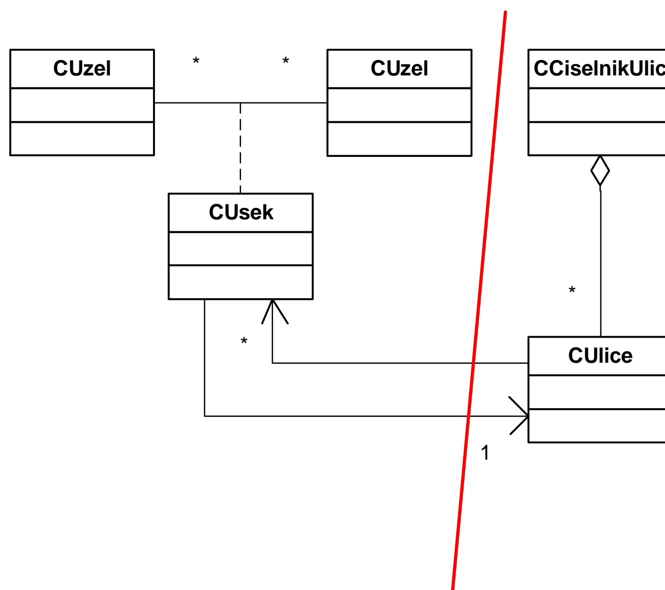
- že mezi komponentami nenastává circular reference,
- že hranice komponent nikdy neprocházejí napříč package ze tříd,
- že daný package by se měl vyskytovat vždy v jedné a pouze v jedné komponentě

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 circular reference. Tento problém jsem nazval „chybou siamských dvojčat“, tj. existuje něco, co spojuje jinak vcelku logicky oddělitelné celky. Otázkou je, jak odstranit to 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 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.

Klasický učebnicový 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:

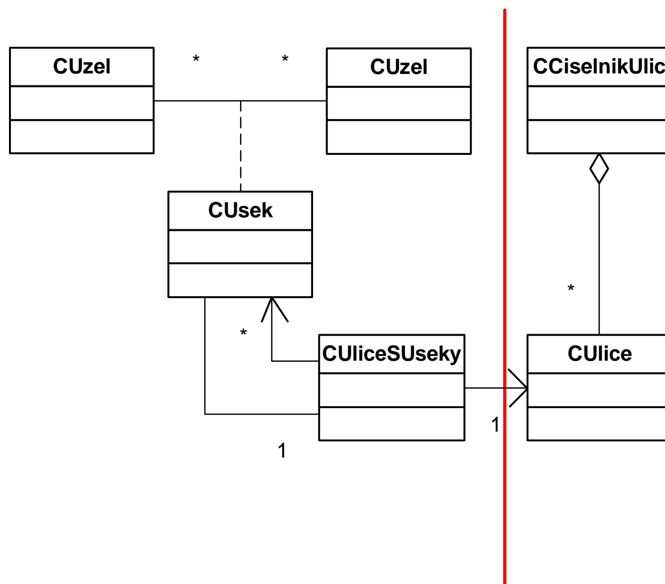


obrázek 40 Nezdařený pokus o odstřihnutí komponenty číselníku ulic

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 pro úsek, a s ním také třídu pro uzel, tedy celý namalovaný model.

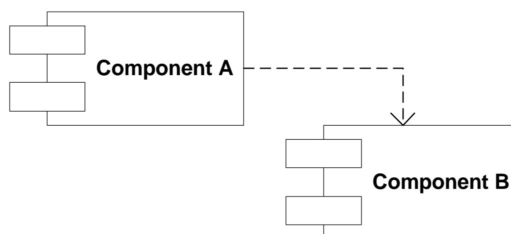
Kde je chyba? Samozřejmě v předešlých odstavcích je nesouhlas. Podle textu je Ulice chápána jako entita, která nese pouze název. Pokud se však podíváme do diagramu, tak to, co jsme v diagramu nazvali ulicí, nemá tuto vlastnost, ale **navíc** ještě vidí Úseky. To, co je na diagramu, není tou „čistou" 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 Useky:



obrázek 41 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 – 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“, je přehledný, fyzicky členěný, rozpadající se na menší řešitelné části.

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 je 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, například v prostředku VISIO apod.

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 zprostředkované zaslání zprávy přes message server**: objekt posílá zprávu druhému objektu přes nějakého prostředníka – objekt Message Serveru, který zprávu převezme a poté ji předá adresátovi. Poslání může být v tomto případě tzv. asynchronní – posílající objekt nečeká na odpověď, odešle zprávu (podobně jako poštu) do Message Server 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 mez objekty: od A k Message Serveru, od Message Serveru k objektu B. V konkrétních technologiích (například firmy Microsoft) se Mesagge Server nazývá MS Message Queue Server.

Není povoleno šíření dokumentu mimo prostor licence!

- **callback**: odložené zaslání zprávy objektu. Tohoto způsobu zasílání zpráv se používá v případech, kdy je žádoucí zaslat zprávu nikoliv dopředu známým konkrétním objektům, ale pouze těm, které to v běhu programu budou potřebovat. V tom případě se zaslání zprávy rozpadne na sekvenci dvou fází: Nejprve se objekt, který má dostat zprávu, zaregistruje k objektu, který je schopen zprávu zaslat, a poté, když nastane situace, že je třeba zprávu poslat, posílá se zpráva všem objektům, které se zaregistrovaly. Vysílající objekt tedy nezná konkrétně, koho má v seznamu, ale zprávy jim pošle, protože o to v předchozím kroku registrace požádaly. Na stejném principu pracuje událostní programování (viz skripta o událostním programování). Výhoda je ta, že dochází k izolaci vysílajícího objektu a objektu, který zprávu přijímá.

Sequence model vyjadřuje sekvence zasílání zpráv mezi objekty a používá k tomu následující syntaxi.

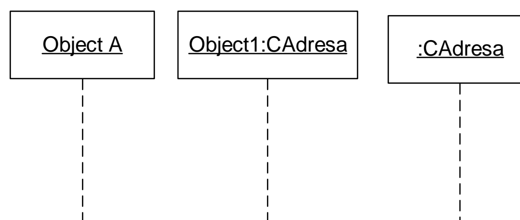
Element „Objekt" v Sequence modelu

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:

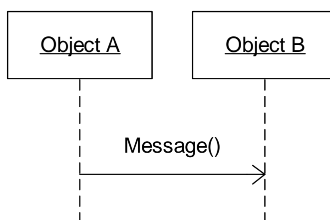


obrázek 42 Objekt bez vyznačení třídy, objekt s vyznačenou třídou a anonymní objekt

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.

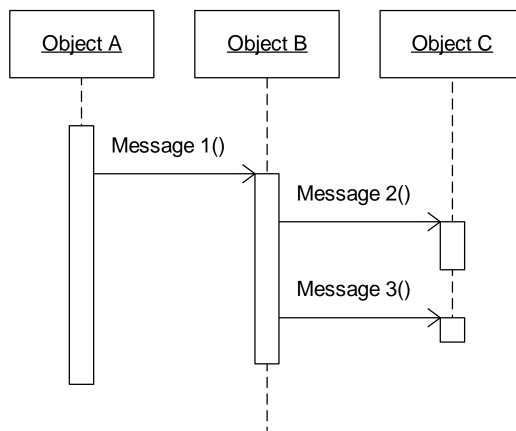
Není povoleno šíření dokumentu mimo prostor licence!



obrázek 43 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 44 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.

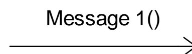
Jednoduchá a asynchronní zpráva

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 45 Synchronní 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:

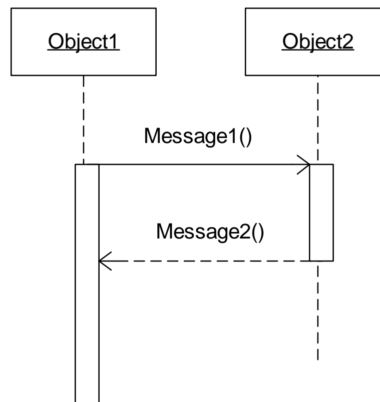
obrázek 46 Asynchronní zpráva

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.

Tato metoda 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, jak bude ukázáno na dalším obrázku. Vstupní argumenty se umisťují vždy do závorky za název metody.V syntaxi výstupních (návratových) parametrů jsem setkal se dvěma způsoby zápisu, jak se znázorňují vracejí 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. Používá se také i syntaxe tzv. návratové zprávy, která „vrací hodnoty". Návrat se tedy označí pomocí tečkované šipky:



obrázek 47 Návratová zpráva - return message

Poznámka: Osobně se této syntaxi vyhýbám a používám pouze syntaxi se vstupně-výstupními parametry zprávy přímo napsanými v závorce názvu zprávy.

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 je u středních a rozsáhlejších systémů utopií a není 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.

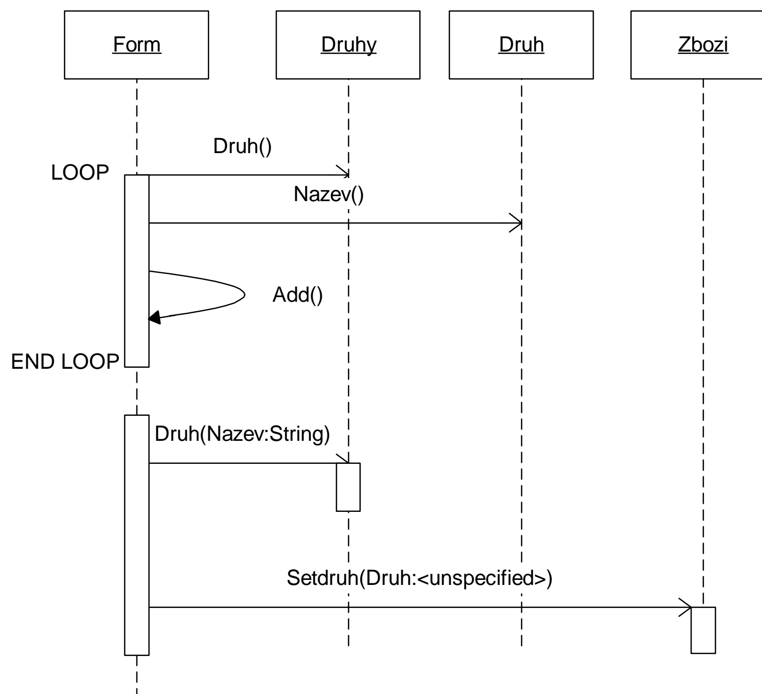
Pro sekvenční diagram je velmi důležitá ta okolnost, že zřetelně popisuje chování aplikace a současně vymezuje kompetence mezi objekty. Vymezení všech sekvenčních diagramů v systému vede vlastně k odhalení všech metod objektů a jejich posloupnosti volání. To, že jeden objekt posílá zprávu druhému objektu 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 48Možný 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.

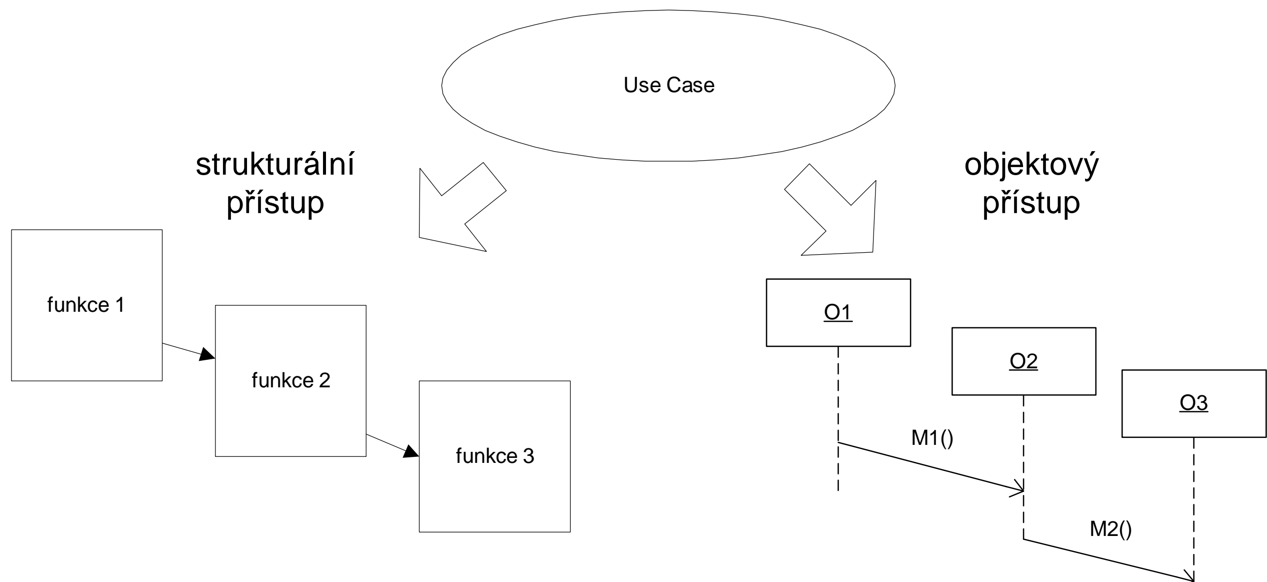
Druhá činnost vznikla po výběru daného prvku. Název je v tomto případě unikátní pro vyhledání Druhu v seznamu Druhů. Formulář si vyžádá daný druh (který byl vybrán) a ten se dosadí do zboží.

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.

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 49 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 volání funkcí a na straně druhé posílání zpráv mezi objekty jsou velmi podobné. Postupné „volání objektů mechanismem zpráv“ je na rozdíl od volání funkcí doplněno o kompetence objektů, které nahrazují kompetence funkcí.

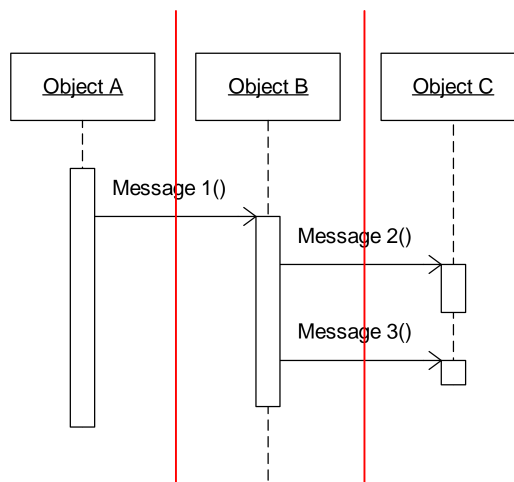
Základní rozdíl v chápání funkcí a objektů zde není znázorněn: Každý objekt O1, O2 a O3 si sebou nese své atributy, které jsou „při něm“ a se kterými může tento objekt pracovat.

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 diagramu 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 každý sekvenční diagram rozdělený svisle na oblasti dané svislými čarami životních cyklů objektů a vstup do této vrstvy se děje pouze šipkami - 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 50 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 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. Z hotového Use Case modelu se vyberou ty, které nejsou hierarchicky dále členěny a vyjadřují tak jednu činnost vyrovnání nerovnováhy mezi okolím a systémem. Slovní popis Use Casu se pomocí objektů a zpráv „překlopí“ do sekvenčního diagramu. Znamená to nalézt z popisu objekty (není povinné nyní hledat třídy) a odpovídající zprávy objektů, které ve své sekvenci realizují 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***. První z takovýchto situací je nutnost vytvoření těch scénářů 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 3.

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 3 kanálu atd.

Použití sekvenčních diagramů pro zavedení vzorů scénářů

V praxi se velmi osvědčilo použít sekvenční diagram pro vyjádření vzorů spolupráce mezi objekty. 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 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.

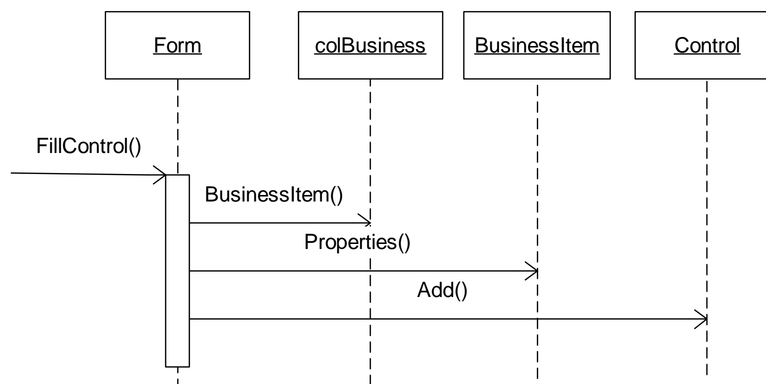
Příklad na použití sekvenčního diagramu jako vzoru

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 maá svoje Properties, které je třeba zobrazit. K zobrazení se použije Control přidávající prvky jako stringy (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:



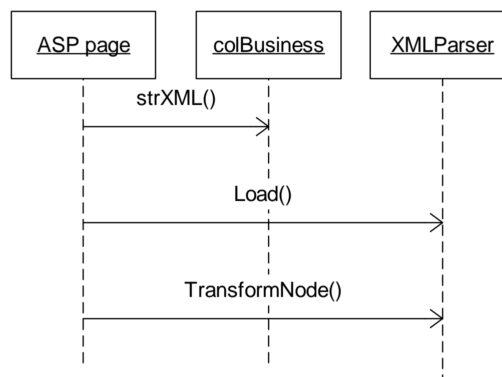
Poznámka: Scénář není vhodný pro „hluboké“ kolekce

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:



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.

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" - realizace (konkretizace) 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

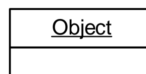
Naskýtá se pochopitelně otázka, nač tedy používat Object model, když informace, kterou nám podává, je odvoditelná a „nejedná se tedy o nic nového".

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).

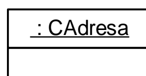
Model element „Object" v UML

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**:

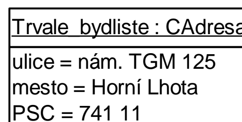


Pokud je známo, z jaké třídy pochází daná instance, můžeme označit tuto skutečnost za názvem instance, oddělovačem je dvojtečka:

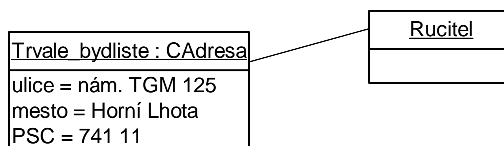
Anebo můžeme název instance vynechat a ponechat pouze název třídy, v tom případě hovoříme o **anonymním objektu:**



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 abstraktní 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

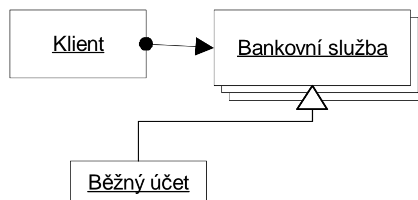
Dalším doplněním je vyznačení agregace pomocí černého puntíku v linku na straně objektu, který agreguje podřízený objekt (na stejné straně jako je kosočtverec u asociace typu agregace). Toto vyjádření označuje, že asociace, ze které tento link vznikl, je agregací a to v odpovídajícím směru.

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) - 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 „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 „prázdným" kontextem (prázdnou objektovou referencí), do kterého lze nižší instanci (objektovou referenci) dosadit. V odpovídajícím Class modelu tomu odpovídá situace, kdy vyšší třída je abstraktní třídou.

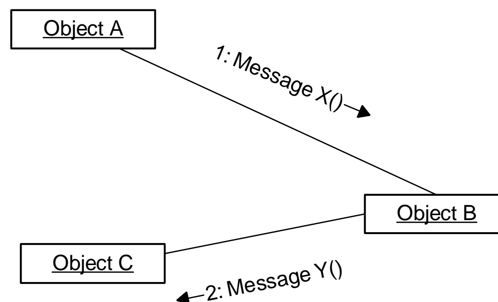
Na závěr ještě připomenu, že kontextový diagram není součástí UML, které povoluje v objektovém diagramu pouze zavádět objekty a jednoduché linky.

Collaboration model

Dalším modelem UML je tzv. Collaboration model, často překládaný jako model spolupráce objektů. Tento model vychází z již zde uvedených dvou modelů - modelu objektového a modelu sekvenčního a lze jej chápat jako „hybrid dvou modelů" – 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:



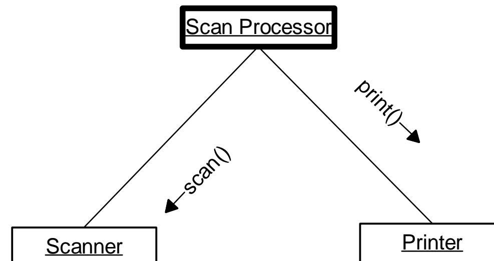
obrázek 51 Model spolupráce objektů

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 (řídí je). Tyto objekty se nazývají aktivní objekty (active objects). Ne vždy však musí takovýto objekt existovat. Aktivní objekty se vyznačují zesíleným obdélníkem.

Příklad:

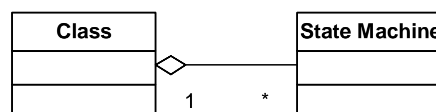


obrázek 52 Scan processor jako active object

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 – 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:



obrázek 53 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 (check) mezi modely atd. Stavový model tak může urychlit tvorbu systému.

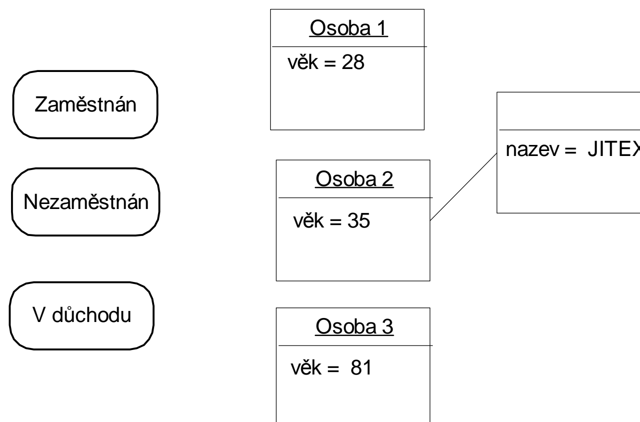
Příklad: V jedné bance při tvorbě informačního systému bylo úkolem zpracovat analyticky chod směnek různého typu, o kterých jsme do té doby nevěděli vůbec nic. Konzultant z banky nám nabídl „plachtu", na které bylo namalováno „jak to v této bance chodí se směnkami". Při bližším studiu této „plachty" 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.

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:



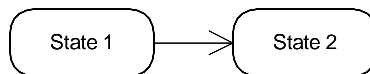
Zde je osoba buď ve stavu zaměstnána, nezaměstnána, nebo v důchodu. Rozhodující je

- 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).

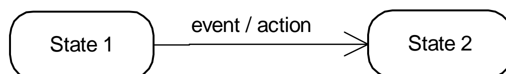
Změna stavu z jednoho do druhého se ve stavovém diagramu zavádí jako transition – přechod mezi stavy. Je jednoznačně určen svým názvem a zavádí se mezi dvojici stavů – mezi výchozí a konečný stav. Jeho visual elementem je šipka spojující výchozí stav a konečný stav:



obrázek 54 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, se zavádí tzv. guard, což je výraz typu boolean, který když nabude hodnoty true, tak k danému přechodu dojde. Také lze vyznačit aktivitu, která změnu stavu doprovází:



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).

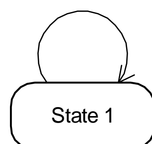
Pro studium samotných přechodů je třeba uvést, že stavový diagram nebere přechody a aktivity s nimi spojené za předmět svého zájmu a studia. Tyto přechody chápe jakoby nastávaly okamžitě, tedy čas strávený překlápěním a co se přitom děje, není pro stavový diagram zajímavým. Je zřejmé, že v některých 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 55 Počáteční stav diagramu (odkud začít číst diagram v přechodech) a konečný stav v diagramu (za kterým již nepokračuje žádný přechod)

Přechod do téhož stavu

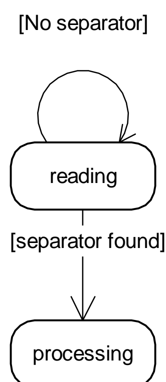
V UML je povoleno zavést transition – přechod mezi dvěma stavy, kdy výchozí stav a konečný stav je shodný:



obrázek 56 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 z 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 „jak to v daném podniku“ vlastně chodí. Z hlediska UML je třeba upozornit na jednu sice triviální, avšak mnohdy zanedbávanou skutečnost:

Jakýkoliv model v UML budeme chápat jako model daného řešeného informačního systému a ničeho jiného!

Pokud opustíme tuto zásadu, můžeme narazit na určité problémy. Samozřejmě tato zásada neznamená, že nebudeme vytvářet dokumenty analyzující problematiku i mimo systém. Tato „analýza“ však není přímo součástí systému, 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 to budou chodit směnky v informačním systému“.

Analýza samotné logisitiky problému je velmi prospěšná pro tvorbu analytických modelů samotného informačního systému. Následující formulace vystihuje povahu těchto analytických dokumentů:

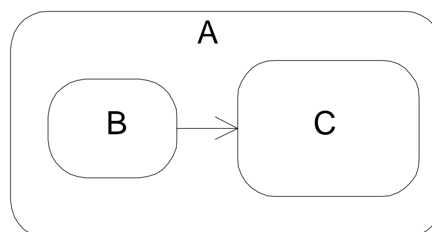
analytické dokumenty problémové domény jsou dobrým zdrojem pro analytické modely samotného systému

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 a může vést k zajímavým chybám:

Jako jeden 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 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 dané reality.

Generalizace stavů a kompozitní stavy

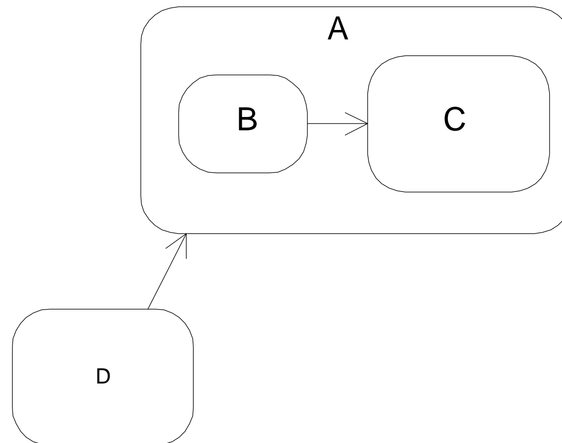
V mnoha případech potřebujeme ve stavovém diagramu vyjádřit tu skutečnost, že daný stav, který znázorňujeme, je jakýmsi „superstavem“ pro jiné stavy, které jsou pod tímto stavem „uschovány“. Například faktura se v prvním přiblížení nachází pouze ve dvou „superstavech“: ve stavu „Připravovaná k odeslání“ a 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.



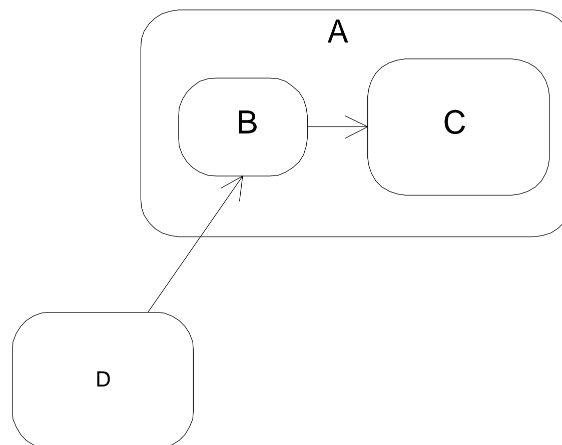
obrázek 57 Superstav A

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.

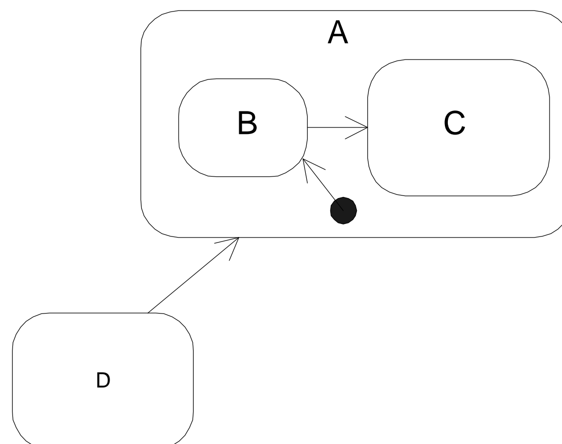
Následující syntaxe není přesná:



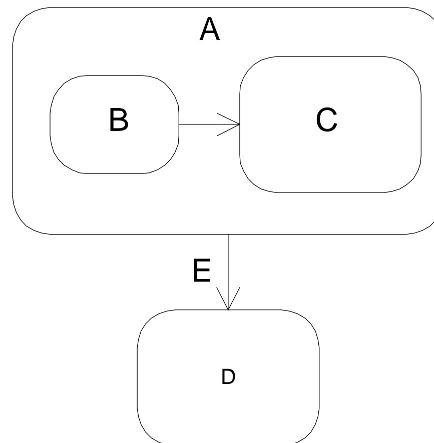
Protože není zřejmé, zda se ze stavu D objekt překlopí do stavu B nebo C. Správně buď takto:



anebo pomocí počátečního stavu takto:



Avšak opačně je dovolena tato syntaxe:



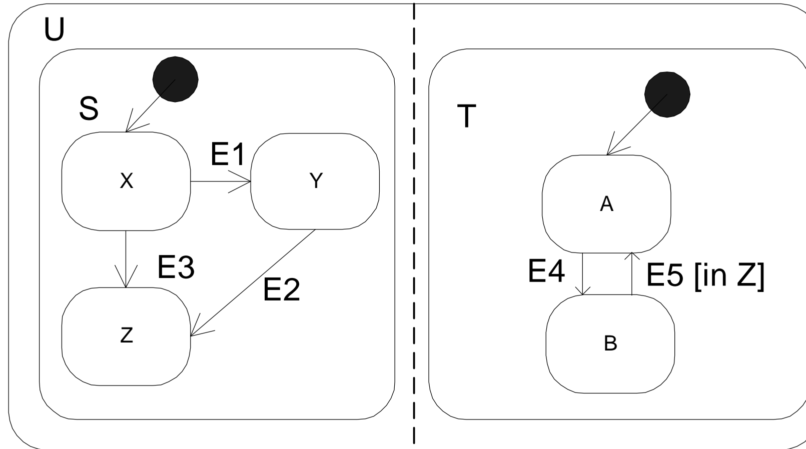
která znamená, že pokud se objekt nachází ve stavu A, tj. v jednom z podstavů B nebo C, a nastane událost E, objekt se překlopí do stavu D.

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“ a 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 úplně dvou 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:



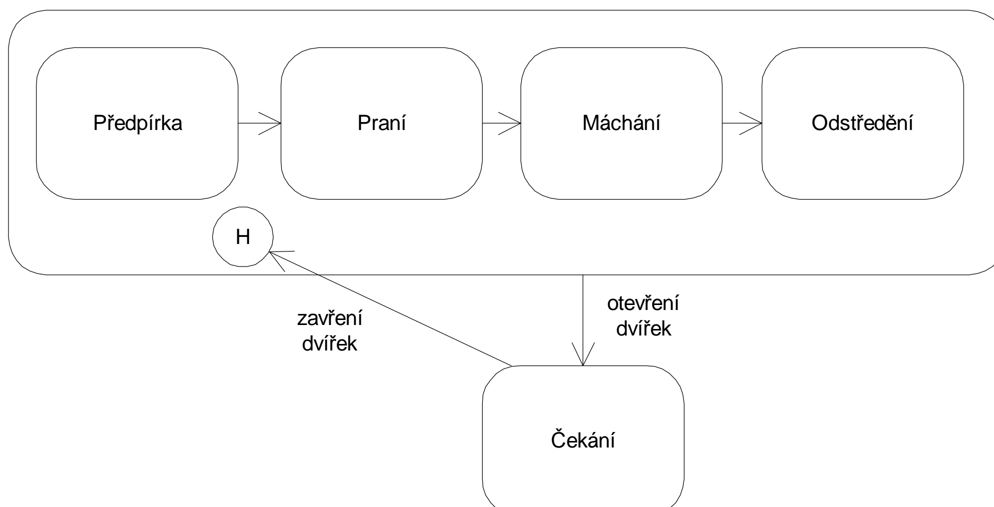
Na obrázku je nejvyšším stavem stav U. Ten je agregován ze dvou stavů - S a T. Stavy S a T jsou zobecněním stavů po řadě X, Y, Z a A, B. Uvnitř stavu S je stavový stroj s počátkem ve stavu X, ve stavu T je počátečním stavem A.

Všimněte si podmínky u přechodu s událostí E5 mezi B a A, kde je omezení ve tvaru [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" – 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.

Není povoleno šíření dokumentu mimo prostor licence!

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.

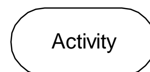
Pokud se vrátíme ke stavovému diagramu, tak u něj samotné přechody nejsou předmětem jeho zájmu, tedy považují se za okamžité, jako by se proces sledování systému v době přechodu nesledoval a „zapnul se" až při překlopení do stavu.

Oproti tomu Activity diagram se zaměřuje právě na tyto přechody mezi stavy. Základním prvkem modelu je tzv. aktivita – activity.

Model element Activity

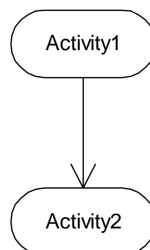
Jedna aktivita se povinně nevztahuje k objektu, ale obecně k systému (bez nutného určení kompetence), tedy jedna vyznačená aktivita systému může být v konečném důsledku realizována několika objekty. To je velká výhoda použití tohoto diagramu zejména v raném stadiu analýzy.

Visual elementem aktivity je ovál (dvě vodorovné rovnoběžné úsečky stejné velikosti spojené půlkružnicemi na koncích):

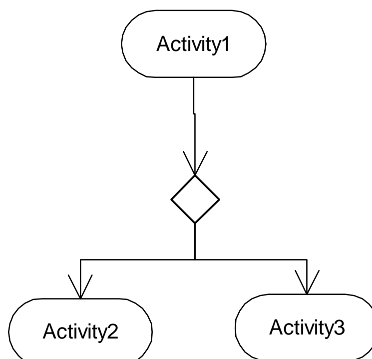


Model element Přechod mezi aktivitami

Přechod z jedné aktivity do druhé se vyznačuje šipkou:



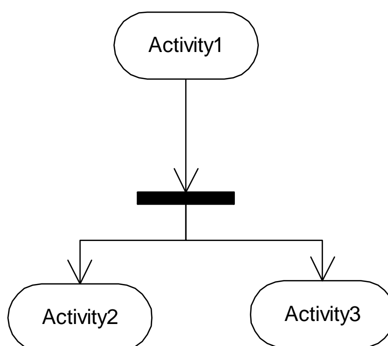
Při rozhodování je možné použít větvení a rozhodování:



obrázek 58 Element rozhodování vede k větvení podle podmínky se vykoná jedna z aktivit

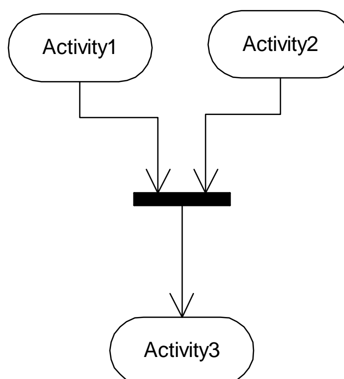
Model element horizontální čára v modelu aktivit

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:

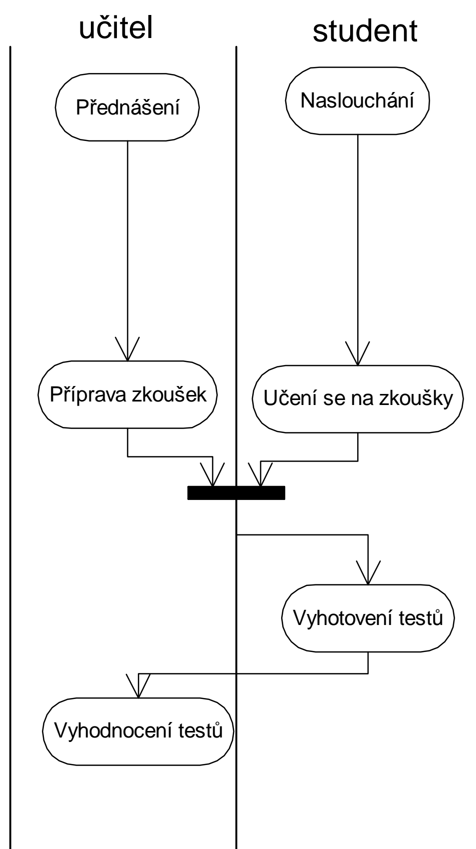


Model element swim-lines v modelu aktivit

Pro vyjádření určitých kompetencí, které ještě nemusí být reprezentovány objekty, ale pouze „oblastmi, 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 (metody) objektů. Tímto model aktivit může v limitním případě přejít až do sekvenčního modelu – jednotlivé swim-lines jsou reprezentovány přímo objekty.

KONEC DOKUMENTU