Ilja Kraval

Analytické modelování informačních systémů pomocí UML v praxi

Object Consulting
2010

Sledujte aktuální novinky na http://www.objects.cz

Anotace: V knize je popsán obecný přístup k analýze informačního systému, respektive vytvoření analytického modelu informačního systému pomocí UML, respektujícího zásady objektově orientovaného přístupu. Detailně je popsáno použití dvou základních diagramů UML: Class Diagramu a Use Case Diagramu. V demonstračních příkladech jsou kromě již známých analytických vzorů a vzorů GOF využívány také původní vzory autora.

KATALOGIZACE V KNIZE – NÁRODNÍ KNIHOVNA ČR

Kraval, Ilja
Analytické modelování informačních systémů pomocí UML
v praxi / Ilja Kraval. – 1. vyd. – Lipina : Object Consulting, 2010. –
140 s.
(978-80-254-6986-6 : brož.)

007+004 * 004.414.2 * 004.42UML
- informační systémy
- analytické modelování
- UML
- příručky

004.4/.6 - Programové vybavení. Programové prostředky [23]

1. vydání
© RNDr. Ilja Kraval, 2010
ISBN 978-80-254-6986-6

Sledujte aktuální novinky na http://www.objects.cz

Kapitola 1

Objektové paradigma

Jako bývalý fyzik mohu potvrdit, že pro jakoukoli teorii je vždy přínosem, pokud je postavena na solidních základech, kterým se říká základní tvrzení neboli axiomy. Ve fyzice jsou jimi například Newtonovy zákony, Einsteinovy rovnice gravitačního pole aj.

Totéž platí u návrhu informačních systémů (dále také „IS“). I když se tvorba softwaru jeví jako praktická věc s praktickými výstupy, také zde lze jako v každé jiné teorii vypozorovat základní axiom neboli paradigma. Podobně jako ve fyzice je možné i v teorii návrhu IS z tohoto axiomu poměrně jednoduše odvodit další důležitá pravidla. Existence paradigmatu neboli axiomu v teorii návrhu IS napovídá, že je nutné jej dobře zvládnout a že je třeba jej dobře pochopit, protože to nám poskytuje dobrý předpoklad k osvojení všech dalších správných postupů při návrhu informačního systému. Pochopitelně podobně platí i opačné tvrzení: Zanedbání tohoto principu vede k neustálým problémům při tvorbě softwaru (dále také „SW“) a je příčinou velmi hrubých chyb v modelech IS. Tímto axiomem návrhu IS je tzv. *objektové paradigma*.

> *Axiom a tedy „zákon všech zákonů“ návrhu informačních systémů se nazývá objektové paradigma nebo také objektový princip, resp. objektová filosofie.*

Hned v úvodu musím zdůraznit, že zde nemám na mysli postuláty tzv. objektově orientovaného programování (angl. Object Oriented Programming, tj. OOP), které se zavádějí v literatuře jako trojice vlastností objektů „zapouzdření + dědičnost + polymorfismus“. Hovořím nyní o principu ještě vyšším, obecnějším a základnějším, z něhož již známé vlastnosti OOP vyplývají teprve jako důsledek.

Nejlépe pochopíme objektové paradigma paradoxně pomocí úvah nad důsledky špatného postupu tvorby softwaru, protože díky nim velmi názorně vyniknou katastrofické efekty vývoje při jeho nedodržení. Rozebereme si proto nejprve chybné a nesprávné postupy tvorby IS a dospějeme tak logicky k nutnosti jak pochopit a jak při vývoji zohlednit princip zvaný objektové paradigma.

1.1 Metoda tvorby informačních systémů zvaná *Tunel*

Jeden ze způsobů tvorby IS se příznačně nazývá *Tunel*. Tato metodika tvorby SW je díky svým vlastnostem obecně velmi nedoporučována, protože se spíše jedná o „anti-metodiku“ než o správné zavedení dobrého a kvalitního postupu tvorby softwaru.

Podstata této „anti-metodiky“ je jednoduchá: Na počátku projektu tým vstupuje do tmavého tunelu, tedy do „černé tmy“. Členové vývojového týmu tvoří software podobně, jako když se tápe poslepu tmou. Je vidět opravdu jenom na jeden, dva kroky dopředu. Vývojáři

Sledujte aktuální novinky na http://www.objects.cz

pomocí nárazů do stěn zjišťují, kudy cesta nevede. Neblahými zkušenostmi z neúspěšných cest, což reprezentují nešťastné zprávy od uživatele nad odevzdaným kusem kódu: „Takto jsme to přece vůbec nechtěli…“, se zjišťuje, co se vlastně nemělo a co se tedy mělo dělat. Metodou „pokus–omyl“ se odhaluje správná cesta. Vedoucí projektu se snaží neustálými operativními zásahy uřídit projekt a najít tak nadějné světýlko na konci tunelu. Většinou se díky hektičnosti prací projekt opravdu „s odřenýma ušima“ a s úlevou dokončí, a „jakýs takýs“ informační systém se zákazníkovi nakonec odevzdá. Bohužel někdy může nastat katastrofická situace, kdy se nadějné světélko na konci tunelu promění ve světla protijedoucího vlaku a celý projekt skončí opravdovým krachem.

> *Základní charakteristikou metody* Tunel *je použití pouze operativního řízení projektu. Projekt je postaven jen na řešení problémů vznikajících ad hoc v dané chvíli a většinou se jedná o hašení požárů. Neexistuje jednotný postup, tj. neexistuje řízení ve smyslu predikce, není a ani nemůže být zavedeno plánování a rozvržení prací dopředu na delší časová údobí, a pokud se někdy nějaký plán navrhne, hned v dalších dnech se stejně změní. Chybí relevantní a efektivní kontrola dokumentů projektu. Většina dokumentů projektu je obsahově i formálně velmi slabá. Dokonce mnohdy z celé dokumentace projektu existuje pouze a jenom zdrojový kód.*

Metoda *Tunel* je bohužel natolik známá a tak často používaná, že osobně neznám vývojáře softwaru, který by během své kariéry alespoň jednou tunelem neprošel a nezakusil si tak jeho opravdové strasti a záludnosti. Musím uvést ještě jeden smutný poznatek z dlouholeté praxe: Stále existují firmy, které se ještě potácejí ve tmě tunelu, a není těchto firem opravdu málo! Je to vcelku pochopitelné, protože pokud v SW firmě nezavedeme žádnou metodiku, automaticky se nasadí jako implicitní, tedy jako defaultní metodika *Tunel*. Asi již tušíme, že hlavní a nosná myšlenka všech správných a doporučovaných metodik výroby softwaru spočívá v doporučení „jak opustit tunel“, anebo přesněji „jak rozsvítit světlo v tunelu“.
Poznámka: *V žádné SW firmě nemůže být zavedeno stoprocentní světlo v tunelu. Vždy bude existovat část řízení projektu postavená na operativním řízení, protože chyby se dělají, postupy nejsou dodržovány a také mohou nastat nepředvídatelné situace. Avšak metoda* Tunel *je postavena jen a pouze na principu operativního řízení a neexistuje žádný rámec, který by vymezoval jakákoliv pravidla prací. S nadsázkou lze říci, že smyslem opuštění tmy tunelu je obrátit nepříznivý poměr „tma versus světlo“ z 90 % tmy ku 10 % světla na poměr mnohem příznivější, například 20 % tmy ku 80 % světla.*

Když si rozebereme důsledky použití metodiky *Tunel*, zjistíme, že je lze rozdělit do tří velkých skupin:

- důsledky pro firmu a její ekonomiku;
- důsledky pro vývojový tým a atmosféru v něm;
- důsledky projevující se ve vlastnostech samotného softwaru.

1.1.1 Ekonomické důsledky metody *Tunel*

V učebnicích řízení softwarových projektů se uvádějí tyto tři ekonomické důsledky působení metodiky *Tunel* a následně nezvládnutých projektů IS:

Zvýšené náklady

Je zřejmé, že zbytečná práce vydaná v tunelu vniveč, k tomu časté předělávky a neustálé opravy chyb, to vše vždy něco stojí. Proto se náklady projektu díky chaosu v metodě *Tunel* enormně zvyšují.

Sledujte aktuální novinky na http://www.objects.cz

Pozdní dodávka systému
Při průchodu *Tunelem* nastávají mimo jiné také problémy s dohodnutými termíny v projektu. Objevují se neustále další a další problémy k řešení, systém se neustále předělává a navíc se vynořují nové oblasti řešení, se kterými se sice původně nepočítalo, ale musí být do projektu logicky zahrnuty, jinak by systém nefungoval korektně. To vše stojí samozřejmě další čas. Několikeré odklady termínu mají pochopitelně následný nepříznivý vliv na obchodní vztahy se zákazníkem, tj. na vztahy s odběratelem systému.

Nesoulad naprogramovaných funkcionalit IS s původními požadavky zákazníka
Vzhledem k tomu, že v *Tunelu* většinou vládne velký chaos a všechny požadavky na systém se zjišťují ad hoc a operativně, velmi těžce se pak odhalují systematicky potřebné funkcionality. Výsledkem tohoto chaotického postupu bývá nepříjemná situace, kdy zákazník odebírající systém je nespokojený, protože podle něj systém sice běhá rychle, ale dělá něco úplně jiného, než se žádalo. Uživatel mnohdy odmítá systém navržený v *Tunelu* převzít jako funkčně nepřijatelný a to se všemi nepříznivými obchodními důsledky.

1.1.2 Důsledky pro atmosféru ve vývojovém týmu

Dalšími důsledky metody *Tunel* jsou nepříznivé jevy v oblasti týmové práce, což pociťují vývojáři přímo na své kůži.

Neznalost postupů prací v projektu
V průchodu *Tunelem* platí obecná neznalost „kdy má co kdo dělat“. Vše je pouze v kompetenci vedoucího projektu, který řídí projekt spíše ze zkušeností a pouze pomocí vlastní intuice. Vedoucí projektu nemá žádnou příručku, ani žádné rady a pokyny, které by mu napověděly, jak v dané chvíli postupovat a co a kdy v týmu po pracovnících vyžadovat. Proto vedoucí projektu volí své vlastní, ať už lepší nebo horší řešení přímo v dané chvíli a na daném místě. Řízení se pro něj stává velmi náročnou prací plnou lokálních operativních zásahů bez možnosti jakékoliv kontroly správnosti rozhodnutí, přičemž z hlediska metod řízení chybí kvalitativní porovnání čehokoliv s čímkoliv.

Jobovy zvěsti v projektu
Každá porada v projektu začíná tzv. „jobovými zvěstmi“, co vše nechodí, co chybí, co je třeba ještě udělat. Pro vedoucího projektu je řízení velmi obtížné a tento pracovník mívá díky tomu mnohdy sklon k žaludečním vředům.

Bobtnání projektu
Jedním z nejhorších jevů, které doprovázejí tvorbu softwaru v tunelu, je tzv. bobtnání projektu. Tato opravdu úděsná choroba projektu se projevuje tak trochu paradoxními syndromem: Vývojáři pracují se stále větším úsilím a odvádějí stále více práce, avšak paradoxně čím více toho v hektické atmosféře vykonají a odevzdají, tím více další práce přibývá (namísto toho, aby práce logicky ubývala). V tunelu nejsou a ani nemohou být známy přesné hranice řešení systému, to má za následek jejich neustálé změny a posuny.

Žádná predikce v projektu a vývoj se jeví jako „dobrodružná hra“
Při metodě *Tunel* není díky jeho podstatě možná jakákoliv predikce v projektu. Ani na začátku, ani v průběhu projektu, a dokonce ani v jeho závěrečných fázích nelze odhadovat další pracnost, tj. jaké všechny práce bude třeba ještě udělat. Účastníci projektu nevědí, co je čeká na jejich cestě a co na ně „vybafne za rohem“. Každý

Sledujte aktuální novinky na http://www.objects.cz

projekt se tak stává sázkou do loterie a připomíná spíše dobrodružnou výpravu za pokladem pirátů než cílenou a efektivní výrobu softwaru.

Hektičnost prací
Chaotický vývoj vede i ke zvláštní atmosféře ve firmě, která je charakterizována vysokou hektičností prací, přílišným shonem a všeobecnou nervozitou. Vyrobený software se neustále předělává doslova „jako za trest“. Provádí se zbytečná a tedy úmorně nepříjemná práce. Výsledkem jsou extrémně náročné vztahy na pracovišti. Firma si v konečném důsledku díky neustálým úpravám a opravám uzurpuje vysoké nároky na volný čas pracovníků. Problém nespočívá ani tak v tom, že by se pracovníci bránili příliš vysokému vypětí sil a přesčasům (pokud mají slušný výdělek), ale jako vysoce inteligentním pracovníkům a expertům jim velmi vadí zbytečnost vykonané práce. Nakonec to mohou pociťovat i jako určitou křivdu, že díky neuvěřitelnému chaosu ve vývoji musejí zůstat přesčas a dané chyby stále dokola opravovat s povzdechem: „Po kolikáté už v této agendě, to už opravdu nikdo neřekne, jak to má být správně?!“. Ve firmě, která pracuje metodou *Tunel*, je úplnou samozřejmostí pracovat hodně přesčas, dokonce až tak, že se to považuje za obvyklý standard chování.

Špatné vztahy na pracovišti
Protože v projektu vládne chaos, není jasné, kde vlastně vznikají chyby. Většinou se proto za chyby postihují úplně jiní pracovníci, než ti, kteří chyby tvoří. Mnohdy bývají paradoxně za chyby postihováni nejvýkonnější zaměstnanci, protože kdo udělá nejvíce práce, pochopitelně udělá i nejvíce chyb, uplatňuje se tu známé rčení: „Kdo nic nedělá, nic nepokazí“. Metoda *Tunel* tedy v žádném případě neprospívá dobrým vztahům na pracovišti.

1.1.3 Důsledky ovlivňující vlastnosti samotného softwaru

Třetí oblastí jsou důsledky pro samotnou podobu kódu. To nás zajímá nejvíc, protože díky této okolnosti nakonec dospějeme ke slíbenému důležitému axiomu.

Pokud se podíváme na kód vytvořený v tunelu nezaujatýma očima, ihned rozpoznáme, že byl tvořen v tunelu, bohužel. Jeho rysy a charakter tomu totiž odpovídají. Na kódu tvořeném v *Tunelu* se velmi dobře pozná, že byl tvořen v *Tunelu*, protože má spoustu negativních vlastností a není vůbec pěkný. Otázkou však je, co dělá některý kód pěkným a jiný škaredým? Mezi vývojáři se pro velmi „nepěkný“ kód ustálil výstižný a přitom hanlivý výraz „kód paskvil“. Na druhé straně existuje označení pro software jako „elegantní software“. Následující seznam dobře vystihuje charakteristiku kódu-paskvilu:

Software bez myšlenky
Software nevykazuje žádné nosné myšlenky a vypadá jako slepenec nesourodých ad hoc přilepených řešení. Výtvor postrádá jakoukoliv elegantní myšlenku a je slepen z mnoha dílčích řešení, která jsou těžkopádná a která jsou viditelně velmi lopotně vypocena.

Přidání atributu, resp. sloupce v tabulce metodou „padni kam padni“
Pokud je třeba přidat atribut, tak díky chaotickému slepenci se „někam“ nešťastně přidá do míst, kam vůbec nepatří. Systém sice funguje, ale pro informace se chodí do neobvyklých a překvapivých míst.

Nízká transparence řešení
Software tvořený v tunelu vykazuje silnou nepřehlednost. Jednotlivé prvky softwaru

v projektech se obtížně identifikují a nacházejí. Opravdu velmi pracně a někdy dokonce až za hranicí možných postupů se vyhledávají všechna relevantní místa, kde se nalezená chyba nebo požadavek na změnu nachází a jaké jsou všechny důsledky změn. U hodně složitých a rozsáhlých systémů se dokonce při opravě chyby ani nenajdou všechna místa, kde se chyba vyskytuje, protože se o všech místech prostě neví. Systém se jeví jako zbytečně složitý, vypadá jako slepenec plný těžko identifikovatelných a tedy obtížně opravitelných chyb. Odhalení chyb (například v testování) ještě není zárukou, že se tato chyba skutečně odstraní ve všech místech svého výskytu.

Režie údržby

Díky nízké transparenci softwaru enormně vzrůstá režie na údržbu systému. Poměr času pro úkony „nalézt chybu, opravit chybu a nalézt relevantní důsledky opravy“ jde výrazně v neprospěch efektivních zásahů do systémů. Vývojáři tráví zbytečně moc času režií samotného rozchození systému než samotným vývojem.

Nízká stabilita systému vůči změnám

Při metodě *Tunel* je každý i malý a jednoduchý analytický požadavek na změnu v konečném důsledku raději odmítán i za cenu obchodních ztrát. Systém tak vykazuje vysokou „synergetickou nestabilitu“: I malé změny v systému vedou k jeho nečekaným kolapsům paradoxně v odlehlých částech systému. Díky nízké transparenci, nestabilitě a chybovosti se systém vyznačuje velkou pracností i v případě jednoduché údržby i po drobném zásahu. Po malé změně v systému se vyžaduje obrovský díl práce na opětovném bezchybném stabilním zprovoznění systému. Proto bývá mnohdy zákazník přemlouván, aby upustil od svého jinak rozumného požadavku jen proto, že drobná změna, z pohledu zákazníka opravdu malicherná, může mít katastrofální následky.

Nízká flexibilita systému

Flexibilitu systému můžeme chápat jednoduše jako vlastnost lehkého a elegantního přizpůsobení se systému novým požadavkům. Měřítkem flexibility je odpověď na otázku: „Co se stane, když v systému změníme to nebo ono. Bude to bolet?“

Uveďme si příklady takovýchto dotazů: „A co se stane, když bude jiný klient chtít použít jiný typ databáze?“, anebo: „a co když se přidá nový možný algoritmus výpočtu?“, anebo: „co se stane, když na tuto událost má být spuštěna funkcionalita v nově přidaném modulu?“ atd. Pokud nás budou tyto změny hodně bolet, systém není flexibilní.

Společný jmenovatel všech vlastností: Nízká opětovná použitelnost

Dostáváme se k důležitému bodu výkladu: Společným jmenovatelem všech uvedených nectností „anti-metodiky“ *Tunel* je nemožnost efektivně a spolehlivě zavést tzv. opětovnou použitelnost neboli znovupoužitelnost, slangově zvanou re-use. V tunelu se některé části agend vyvíjejí několikrát a to aniž by se vědělo, že se na daném problému již ve firmě pracuje. V systému se následně vyskytují opakující se části kódu jako znovu naprogramované funkcionality. To pochopitelně vede k dalším nečekaným problémům díky existenci několikanásobných řešení.

Poslední bod je natolik závažný, že se mu budeme věnovat blíže. Je totiž těsně spjat s tím, co se nazývá jako již zmíněné objektové paradigma neboli objektový princip.

Sledujte aktuální novinky na http://www.objects.cz

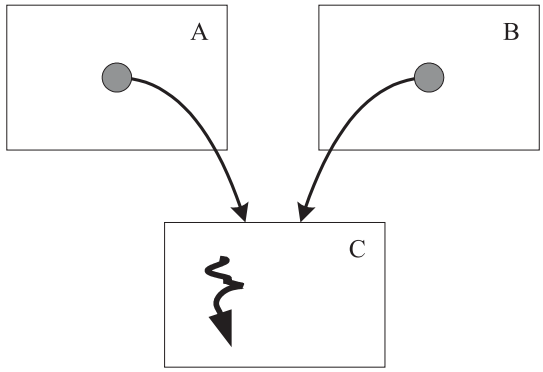
1.2 Opětovná použitelnost

Objektové paradigma úzce souvisí se znovupoužitelností neboli opětovnou použitelností, slangově zvanou re-use. Proto si nejprve vysvětlíme, jak opětovná použitelnost funguje.

Představme si, že existuje nějaký prvek A, což může být libovolný element v analýze, návrhu nebo kódu, a nechť existuje nějaký jiný prvek B. Zjistíme, že v prvku A se vyskytuje určitá jeho část, která se opakuje také v prvku B, viz obrázek 1.1.

Obrázek 1.1: Část opakující se v prvku A i v prvku B.

Kromě této první situace umožňuje princip opětovné použitelnosti zavést také situaci druhou, z hlediska opětovné použitelnosti na vyšší úrovni: Zavede se nový prvek, nazvěme jej například C, a do tohoto prvku se umístí opakující se část z A i z B. V původních místech se obě opakující se části zruší a nahradí se interakcí tak, že z bodů vytknutí se vyvede interakce použití do prvku C. Daná opakující se část se tedy „vytkne“ do prvku C, jak ukazuje obrázek 1.2.



Obrázek 1.2: Vytknutí společné části.

Takto obecně funguje opětovná použitelnost. Situaci první budeme dále nazývat „situace před zavedením interakce“ a situaci druhou budeme nazývat „situace po zavedení interakce“. Jako příklady takovýchto interakcí při tvorbě IS lze konkrétně uvést:

- zavádí se volání funkce, což je použití jedné funkce druhou funkcí;
- zavádí se dědičnost v objektovém programování jako použití jedné třídy druhou třídou;
- zavádí se vztah mezi tabulkami přes tzv. JOIN mezi nimi (obecně relace mezi entitami), což je použití jedné datové struktury druhou datovou strukturou;
- zavádí se vztah «include» v *Use Case Modelu*, což je použití jednoho případu užití druhým případem užití;

- v dokumentaci se autor odkazuje v textu na jiný očíslovaný odstavec (například viz bod ten a ten apod.);
- aj.

K tomu, aby bylo možné zavést re-use, neboli opětovnou použitelnost, musí být zavedena již uvedená interakce použití, která má specifické vlastnosti. Vyjmenujme si je:

Směrovost interakce
Interakce použití uvedená v předešlé kapitole je evidentně směrová. Je zřejmé, že pokud prvek A používá prvek C, tak z toho v žádném případě neplyne, že obráceně prvek C používá prvek A. Pokud je interakce identifikována jako obousměrná, jedná se podle tohoto náhledu o dvě interakce: Jedna má směr „tam“ a druhá má směr „zpět“.

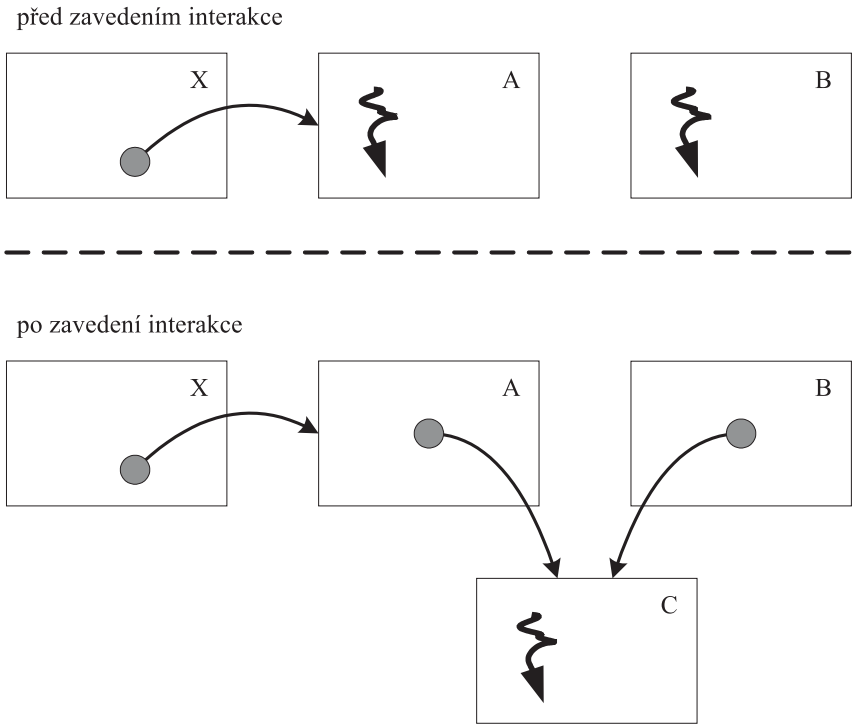
POZNÁMKA: *Je třeba při této příležitosti upozornit na záludnost ukrytou v technologii relační databáze. V relační databázi při propojení záznamů přes shodu hodnot ve sloupcích tato interakce* není jednosměrná *automaticky díky povaze vztahu v relační databázi. Původní jednosměrnost vztahu se v relační databázi svou povahou „znehodnotí“. Mnohdy právě tato skutečnost bývá pro „databázisty“ vážným problémem v chápání směrovosti vazby, pokud se dívají pouze na datové struktury. Naproti tomu objektové programování směrovost vazby výrazně dodržuje: Pokud nějaký objekt používá interface druhého objektu, tak z toho vůbec neplyne, že druhý objekt používá interface prvého objektu (k tomu je třeba zavést další interakci „zpět“). Z tohoto titulu jsou relační databázisté v určité nevýhodě při chápání objektového principu.*

Klient prvku
Prvek, který ve směrové interakci používá jiný prvek, se nazývá klientem prvku. Používaný prvek budeme také nazývat „objekt“ nebo „element“, resp. jenom „prvek“ (POZN.: *jedná se o obecnější pojem, než tzv. objekt konkrétně v objektovém programování*). Prvkem může být i modelovací prvek v návrhu IS, například prvek typu *Use Case*, typu *Actor*, typu *Class*, typu *Component* atd.

Identifikace prvku a ukazatel na něj
Obecný objekt neboli element, tj. prvek, který je použit, musí být v systému identifikován, aby mohl být použit. Pro tu situaci, kdy klient používá konkrétní identifikovaný prvek, se také používá slangová terminologie, že „klient si drží ukazatel na používaný prvek“, přičemž pod ukazatelem se nemyslí pouze „paměťový“ ukazatel v OOP (kde tato věta platí samozřejmě také), ale obecně jakýkoliv ukazatel (třeba i datový, například cizí klíč apod.).

Vnější a vnitřní pohled na prvky, služby a jejich implementace
Nechť nový další prvek X používá prvek A z obrázku 1.2, pro přehlednost je tato situace zobrazena na obrázku 1.3. Obrázek zobrazuje obě situace, tj. situaci jak před zavedením interakce, tak i situaci po zavedení interakce, nyní však i s prvkem X jako klientem prvku A.

Pro další vysvětlení je nyní třeba zdůraznit na první pohled zřejmý fakt: Obě situace na obrázku 1.3, tj. jak situace před zavedením interakce, tak i situace po jejím zavedení, si musí být z hlediska pohledu prvku X rovnocenné. Pokud by toto tvrzení neplatilo, nedala by se první situace nahradit situací druhou, tj. nedala by se zavést opětovná použitelnost. Jinak řečeno obě situace jsou si z pozice klienta X vůči A rovnocenné. Logicky z toho vyplývá



Obrázek 1.3: Porovnání situací před a po zavedení interakce použití.

velmi důležitý závěr: *Interakce použití prvku končí na hraně daného prvku, tedy s nadsázkou řečeno interakce již „nevidí“ další vnitřní interakce tohoto prvku s dalšími prvky.*

Konkrétně na obrázku 1.3 nesmí pro uvedenou interakci „prvek X používá prvek A“ hrát žádnou roli, zda byl prvek C z prvku A „vytknut“ anebo nikoliv. Jinak řečeno pro interakci „prvek X používá prvek A“ je irelevantní, zda byla nebo nebyla zavedena interakce za hranicí prvku A. Vyplývá z toho, že interakce „prvek X používá prvek A“ zřetelně a jasně končí na vnější hranici A, dále již její viditelnost nepokračuje (tj. platí, že „prvek X nevidí interakci, že prvek A používá prvek C“).

Z tohoto důvodu je třeba větu „daný nějaký prvek používá nějaký jiný prvek“ přeformulovat na přesnější vyjádření, které mnohem lépe odpovídá tomuto vnějšímu pohledu interakce, jenž končí na hranici prvku. Nebudeme tedy od této chvíle říkat, že prvek X používá prvek Y, jak jsme byli zvyklí, ale tuto větu přeformulujme takto: *Prvek Y nabízí ven službu použití a prvek X zvenku tuto službu použije.* Pod službou použití máme na mysli nabízenou možnost být použit v interakci použití. Příklady nově zavedené formulace této věty:

1. Jsme zvyklí říkat, že „jedna funkce zavolá druhou funkci“. Přesnější formulace však zní: Daná funkce nabízí svou službu „být zavolána“ a jiná funkce tuto funkci zavolá tak, že tuto službu použije.

2. Podobně jsme zvyklí říkat, že jedna třída podědí z druhé třídy. Přesnější formulace však zní: Jedna třída nabízí službu „být poděděna“ a jiná třída tuto službu využije a tak z ní podědí.

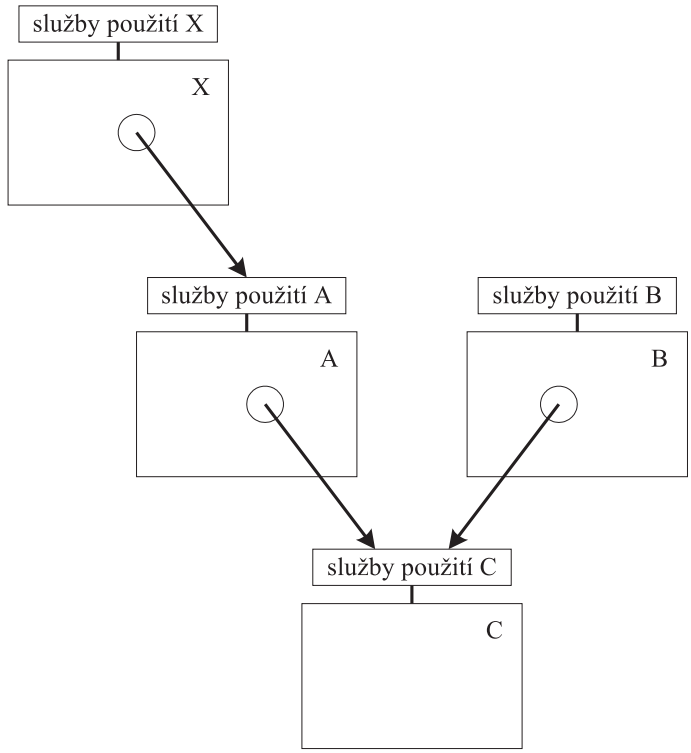
3. Velmi podobně říkáme, že jeden objekt zavolá metodu druhého objektu. Přesnější formulace však zní: Objekt nabízí službu metody interfacu „být zavolána“ a druhý objekt zavolá metodu objektu a tím tuto službu použije.

4. Podobně v relační databázi platí, že tabulka má zaveden primární klíč a nabízí tak svou službu použití pomocí SQL vazby JOIN přes cizí klíč jako odkaz z jiné tabulky.

Zobecnění této věty pro libovolný prvek tedy zní takto:

> *Interakce v návrhu IS obecně funguje tak, že prvek poskytuje službu možného použití jiným prvkům a jiný prvek tuto službu použije a tím de facto použije daný prvek.*

Pohled na interakci jako větu „prvek nabízí službu a klient ji použije“ znázorňuje obrázek 1.4.



Obrázek 1.4: Použití prvku znázorněno jako služba.

1.3 Formulace objektového paradigmatu

Obecný objektový princip neboli objektové paradigma (dále také „OP“) lze formulovat takto:

> *Prvek v systému nabízí jiným prvkům svou službu možného použití. Jiný prvek (nazývá se klient) přistupuje k tomuto prvku a použije tuto jeho nabízenou službu*

Sledujte aktuální novinky na http://www.objects.cz

použití a tím tento prvek použije. Vzniká tak směrová interakce použití prvku prvkem. Klient má takto zpřístupněnou službu prvku a přitom pro něj není viditelná vnitřní struktura použitého prvku neboli není viditelná implementace nabízené služby.

Upozornění: *Toto paradigma je obecné. Je na něm postaven návrh IS pomocí modelování v UML v libovolném diagramu, což je pro další výklad obzvlášť důležité!*

Přirovnání služby k tlačítku
Použití jednoho prvku druhým prvkem v objektovém modelování je podobné, jako kdybychom si představili, že prvek nabízející službu by měl na sobě „tlačítko služby ke stisknutí". Na tomto tlačítku je nápis, který vystihuje službu, kterou může druhý prvek použít. K použití dojde tak, že druhý prvek (klient) stiskne tlačítko a tím dostane službu. Ten, kdo stiskne tlačítko, je na jednom konci interakce (začátek „šipky" v diagramech), a naopak ten, kdo tlačítko nabízí, je na druhém konci této interakce (konec „šipky" v diagramech).

1.3.1 Relativita pohledu na prvek v IS a nejčastější chyba návrhu IS

Objektové paradigma zavedené v předešlém odstavci má za následek doposud neobvyklý poznatek: Je třeba přijmout jako fakt tu skutečnost, že *pohled na libovolný prvek v IS je vždy relativní*! Buď se jedná o popis daného prvku „zvně", tedy buď se popisují jeho služby, anebo se zkoumá daný prvek „zevnitř", tj. jak jsou tyto služby vnitřně strukturovány (také se říká ekvivalentně „jak jsou implementovány"). Tato relativita pohledu na prvek má důležitý a dokonce velmi praktický důsledek:

Pokud se hovoří o daném prvku, musí se vždy ještě navíc přesně zadat (pokud to není jasné z kontextu), zda se hovoří o vnějším pohledu na služby nebo o vnitřním pohledu na implementaci služeb prvku. Nemá proto smysl hovořit pouze o „jednom absolutním pohledu" na prvek, protože pohled na prvek je relativní – buď je vnější nebo vnitřní.

Závěr je tedy jasný: Existují dva možné pohledy, které mají různý výsledek obrazu! Pokud autor popisu neuvede, který z těchto pohledů má na mysli, dopouští se tak hrubé chyby.
Poznámka: *Samozřejmě analytik při rozhovoru s laikem nebude vyžadovat dodržování tohoto pravidla. Ale musí mu být z kontextu řeči vždy jasné, zda se hovoří o vnějším nebo o vnitřním pohledu na prvek.*

Velkým problémem návrhu IS a hlavně obrovským zdrojem chyb a to již ve fázi analytického modelování je skutečnost, že tyto dva pohledy (tj. „vnější pohled na prvek" a „vnitřní pohled na prvek") jsou sice dva pohledy na jeden a tentýž prvek, ale oba mají svůj specifický a tedy jiný výsledek svého obrazu. Bohužel, mohu potvrdit z mnohaletých zkušeností konzultanta a školitele, že neznalost anebo dokonce pouze mírné podcenění tohoto dvojího pohledu jako důsledku objektového paradigmatu vede k hrubým chybám v návrhu IS a to již na analytické úrovni. Uveďme si nyní takové klasické příklady hrubých chyb.

1.3.2 Příklady chyb při návrhu IS vzniklé zanedbáním objektového paradigmatu

Představme si rozhovor analytika s budoucím zákazníkem systému. Analytik se chce od zákazníka dozvědět, jaké osoby se budou evidovat v systému. Zákazník se v rozhovoru vyjádří

v tomto smyslu: „Chceme v systému evidovat studenty.“ Poté následuje dotaz od analytika: „Jak od sebe rozeznáme dva evidované studenty (tj. nikoliv v realitě, ale evidované studenty v systému) například při výpisu anebo při zobrazení seznamu studentů na obrazovce?“ Zákazník odpoví: „Každý evidovaný student obsahuje v sobě rodné číslo, tj. máme povoleno evidovat rodná čísla u studentů. Dva studenty rozeznáme od sebe právě podle rozdílného rodného čísla. Jiný evidovaný student bude mít jinou hodnotu evidovaného rodného čísla.“ (POZN.: *pro tento příklad pomineme možné zdvojení rodných čísel a neuvažujeme zahraniční studenty*). Podobně se analytik dozví totéž o zaměstnancích, o nich zákazník řekne: „Potřebujeme evidovat zaměstnance a u nich budeme také evidovat rodné číslo a opět platí: Jiný zaměstnanec v evidenci, jiné rodné číslo.“ (POZN.: *také pro tento případ pomineme možné zdvojení rodných čísel a neuvažujeme zahraniční zaměstnance*). Navíc z dalšího rozhovoru vyplyne, že vcelku pochopitelně student může být zaměstnancem. Nyní si analytik může položit tuto otázku: *Obsahuje evidovaný student (resp. zaměstnanec) rodné číslo nebo neobsahuje?* Důsledným logickým rozborem se dojde k určitému myšlenkovému rozporu, který zní takto:

1. Na jedné straně je zřejmé, že každý evidovaný student a každý evidovaný zaměstnanec musí mít v sobě uvedeno rodné číslo se svou hodnotou rodného čísla, protože tak to přesně znělo v zadání a tak se budou od sebe tyto prvky v evidenci odlišovat.

2. Na straně druhé, pokud má zaměstnanec i student v sobě rodné číslo a pokud je student současně i zaměstnancem, tak dojde ke zdvojení rodného čísla, protože totéž číslo se vyskytuje i se svou hodnotou jak v evidovaném studentovi, tak v evidovaném zaměstnanci. Pokud by tedy obě evidence, jak evidence studentů, tak evidence zaměstnanců, nesly v sobě rodné číslo, pak by se rodná čísla evidovala dvakrát, což je samozřejmě nežádoucí.

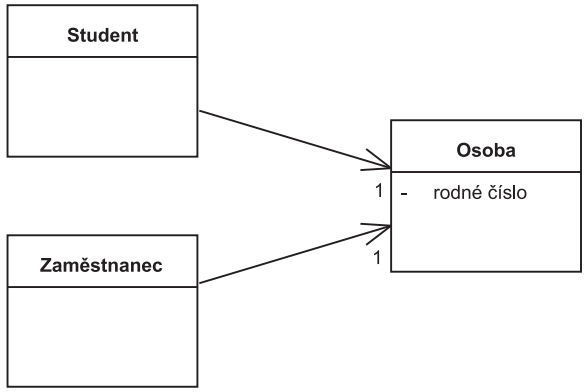
Jaká je tedy správná odpověď? Obsahuje student rodné číslo nebo jej neobsahuje?

Paradoxně, právě díky objektovému paradigmatu a *díky relativitě pohledu na prvek jsou obě dvě předešlé odpovědi správné* a nejsou v rozporu! Platí tedy „ano i ne“. Pokud se nám jeví tato odpověď nesmyslná, pak stačí dodat: *Záleží na úhlu pohledu.*
POZNÁMKA: *Sama „relativita“ by se dala stručně definovat slovním spojením „... záleží na úhlu pohledu“.*

Onu dvojakost šalamounské odpovědi chytré horákyně umožňuje právě důsledné dodržení objektového paradigmatu a jeho správné pochopení. Jak bylo řečeno, je třeba díky objektovému paradigmatu rozlišovat úhel pohledu na dané evidované prvky, tj. rozlišujeme pohled na evidované prvky zvně jako pohled na prvky poskytující služby a krom toho rozeznáváme pohled na prvky zevnitř, tj. pohled na implementaci těchto služeb. Právě zohlednění relativity těchto pohledů, „jeden zvně a druhý zevnitř“, vede ke správné odpovědi na zmíněnou otázku. Pokud se drží v evidenci prvek reprezentující evidovaného studenta, tak je zřejmé, že tento prvek lze zvně požádat o rodné číslo. Tuto informaci bude evidovaný prvek určitě schopen nějak vydat. Proto lze z hlediska vnějšího pohledu odpovědět: „Ano, prvek evidovaný student má v sobě určitě rodné číslo, když jej vydává.“ Ale je třeba být opatrný a správně chápat tuto větu! Řeč je o tom, že student „umí vydat rodné číslo“ a to je vnější pohled na evidovaný prvek a nikoliv pohled vnitřní! Z vnějšího pohledu se tento prvek opravdu jeví tak, že „umí rodné číslo“, a zde je slovo „umí“ oním synonymem ve významu „obsahuje“. Stejně tak totéž platí o daném prvku evidovaného zaměstnance, i tento prvek lze zvně požádat o rodné číslo a tento prvek je schopen jej nějak vydat. Zvně se tedy jeví, jako by oba prvky rodné číslo obsahovaly, protože jej umí. Avšak tyto dvě věty vůbec neznamenají, že jak prvek

Sledujte aktuální novinky na http://www.objects.cz

evidovaný student, tak evidovaný zaměstnanec, mají přímo v sobě implementováno rodné číslo jako atribut a že ho tedy „obsahují“ ve vnitřním pohledu! Při pohledu dovnitř, tj. do vnitřních struktur těchto prvků, se zjistí, že rodné číslo není ani v jednom z nich, tedy není přímo v těchto prvcích, ale je implementováno do dalšího prvku evidovaná osoba, který tyto dva prvky používají tak, jak je uvedeno na obrázku 1.5.



Obrázek 1.5: Použití prvku evidovaná Osoba.

Sice jsme malinko předběhli v syntaxi jazyka UML, ale je zřejmé, o co tu jde: Obrázek 1.5 vyjadřuje jednoduchou skutečnost, že každý evidovaný prvek Student používá jeden evidovaný prvek Osoba a stejně tak každý evidovaný prvek Zaměstnanec používá také jeden evidovaný prvek Osoba, které mohou a nemusí být stejnými evidovanými prvky (tj. tatáž evidovaná Osoba). Při pohledu na tento obrázek se doporučuje tak trochu „rozdvojit osobnost“, protože obrázek obsahuje oba pohledy současně, jeden „zvně“ a jeden „zevnitř“ (a ty se nesmí míchat!). Nechť existuje nějaký další prvek informačního systému, který bude používat prvek Student (například naprogramovaný formulář označený jako „Formulář detail studenta“. Tento formulář má zpřístupněn evidovaný prvek Student a používá jej). Otázka zní: „Může tento prvek (například Formulář detailu studenta) nějak zobrazit i údaje osoby, jako je například rodné číslo?“.

Odpověď zní ano, protože evidovaný student používá evidovanou osobu, takže může nějak v dané technologii převzít údaje osoby a předat je ven, například formuláři. Z pohledu Formuláře detailu studenta tedy platí tvrzení: „Student má rodné číslo – přesněji řečeno umí jej nějak vydat“. Na druhé straně ve vnitřní struktuře Studenta je patrné, že on sám jako prvek nemá přímo rodné číslo jako svůj atribut, ale má jej Osoba, kterou evidovaný prvek Student používá v interakci. V tomto návrhu má rodné číslo jako atribut až prvek Osoba, kterou oba evidované prvky Student i Zaměstnanec používají. Důvod „vytknutí osoby“ v předešlém modelu se studenty a zaměstnanci je zřejmý: Tímto řešením se zabránilo zdvojení osob a rodných čísel v evidenci informačního systému.

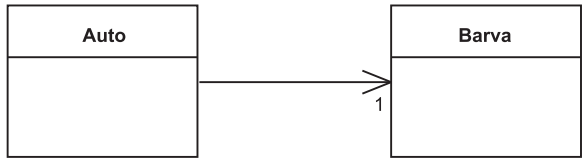
Velkou chybou by byla konstrukce, kdy by jak Student, tak Zaměstnanec měli oba přímo v sobě implementovány atributy rodné číslo, viz obrázek 1.6.

Je zřejmé, že sice v tomto modelu „umí“ jak evidovaný Student, tak evidovaný Zaměstnanec vydat rodné číslo, protože je vlastní přímo jako atribut, ale bohužel, jak plyne z jednoduché logiky, pokud bude zaměstnanec současně i studentem, nastává problém dvojí evidence téhož rodného čísla v systému, což je nežádoucí.

Obrázek 1.6: Chybný model s redundancí rodných čísel.

Poznámka: *Je třeba upozornit na to, jak úzce souvisí tento příklad s opětovnou použitelností neboli unikátností neopakovaných informací! Díky „vytknutí“ (viz operace opětovné použitelnosti) se zabraňuje zdvojení evidence osob v systému.*

Jako další příklad lze uvést evidenci aut a jejich možných barev. Je zřejmé, že pokud existuje v evidenci nějaké zaevidované auto, žádá se znát i jeho barvu. Opět lze položit obdobnou otázku: Obsahuje evidované auto barvu nebo ne? Odpověď zní „šalamounsky“: „Ano i ne, záleží na úhlu pohledu!“. Pokud se drží v programu evidovaný prvek Auto a chceme znát jeho Barvu, potom tento požadavek sice Auto umí splnit (pohled zvně), ale pokud se podíváme, jak jej umí splnit (pohled zevnitř), tak zjistíme, že evidovaný prvek Auto má odkaz na evidovaný prvek Barva a proto je schopen jej splnit, viz obrázek 1.7.



Obrázek 1.7: Vztah Auto má Barvu.

Opět vnější pohled umožňuje požádat prvek Auto o informace z Barvy, ale vnitřně tyto informace nejsou umístěny přímo v Autě jako jeho atributy, ale v prvku Barva a od něj je lze získat interakcí.

1.3.3 Důležité pravidlo návrhu IS pro analytika

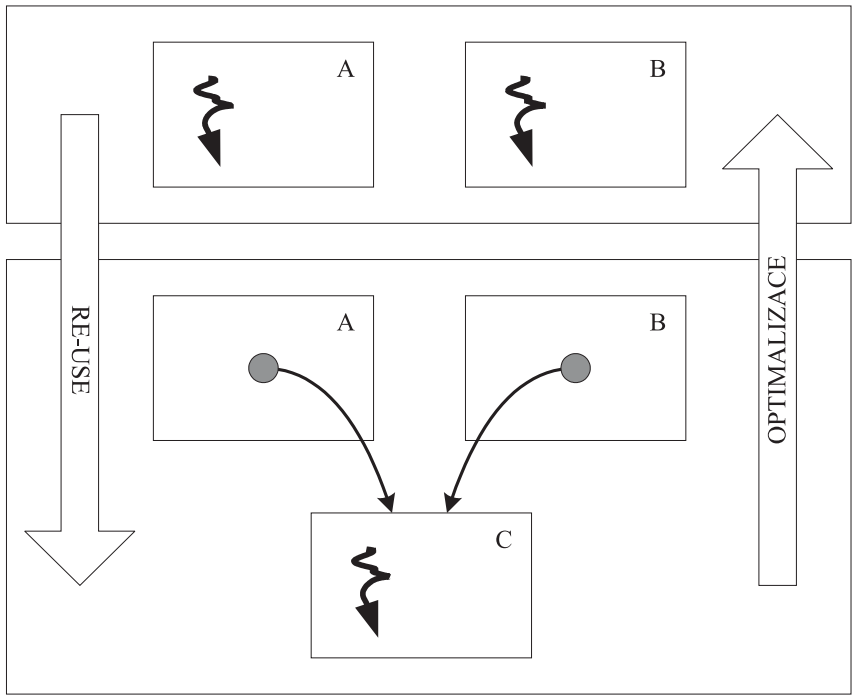
Je třeba si do všech důsledků uvědomit, že právě objektové paradigma je hlavní pomůckou při řešení základní otázky, kde vlastně informace v IS leží, kde ji hledat, kam ji umístit, tj. který prvek ji v IS vlastní. Pokud je objektové paradigma pochopeno do důsledků, potom se analytik nenechá zmást vnějším pohledem objektového paradigmatu. Vnější pohled je totiž svým způsobem bez pochopení objektového paradigmatu velmi matoucí! Při pohledu na prvek zvně platí, že díky interakcím s jinými prvky v IS lze daný prvek de facto požádat o vše, s čím tento prvek interaguje. To znamená, že „vnější pohled na prvek bere vše“, tj. pokud se drží evidovaný prvek, je možné jej požádat o libovolnou informaci, kterou má díky interakcím přístupnou. Navíc tato vlastnost je zřetelně tranzitivní, tj. pokud můžeme požádat prvek o libovolnou informaci, kterou má zpřístupněnu díky interakcím s dalšími prvky, tak samozřejmě platí, že i tyto prvky na koncích interakcí (které daný prvek používá), mají opět tuto schopnost vnějšího pohledu, tj. i ony jsou schopny vydat libovolnou informaci, kterou mají zpřístupněnu dalšími interakcemi atd. Uvedené pravidlo lze vyjádřit jako následující pomůcku analytika:

> *Pokud se hovoří o vlastnostech prvku (a tedy jeho „umění“), jedná se o vnější pohled a ten v sobě zahrnuje v implementaci i případné další použití interakcí.*

Z toho důvodu může být tato služba implementována pomocí interakcí, což vede k delegování tohoto umění na jiné prvky. Analytik se nesmí nechat zmást slovy laika, že „daný prvek obsahuje tuto informaci“. V duchu si tuto větu přeloží do jiné, korektnější podoby, která přesně odpovídá objektovému paradigmatu: „daný prvek je schopen vydat tuto informaci“. Tím si ponechává otevřená dvířka pro možnou implementaci této služby interakcemi s jinými prvky a nebude ji chápat přímo jako „obsah právě tohoto prvku“.

1.3.4 Proces zvaný optimalizace

Optimalizace je technologický postup, při kterém se úmyslně nedodrží opětovná použitelnost. Díky tomuto postupu se získává určitá technologická výhoda, většinou rychlost zpracování, případně velikost paměti apod. Ztrátou je však vždy snížená opětovná použitelnost. Optimalizace je tedy cíleným technologickým krokem (UPOZORNĚNÍ: *pouze technologickým krokem!*) ve směru proti re-use, jak ukazuje obrázek 1.8.

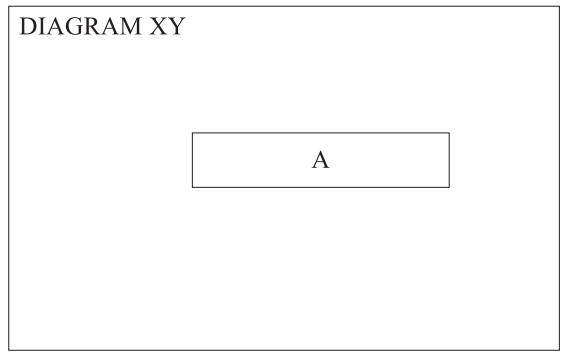


Obrázek 1.8: Optimalizace jako krok proti re-use.

Cílem optimalizace je technologický „handl“ neboli „obchod“ – v technologickém návrhu získáme nějakou technologickou výhodu (například rychlost), ale ztrácíme výsadu opětovné použitelnosti. Každá optimalizace by měla být podložena testy o její nutnosti. Obecně se v literatuře (a praxe tomu nasvědčuje) varuje před předčasnou a zbytečnou optimalizací, protože ztráta opětovné použitelnosti nejenom zbytečně „zesložití“ systém, ale hlavně vede ke snížení transparence systému. Proto by každý krok optimalizace měl být podložen výsledky testů, že systém tuto optimalizaci potřebuje.

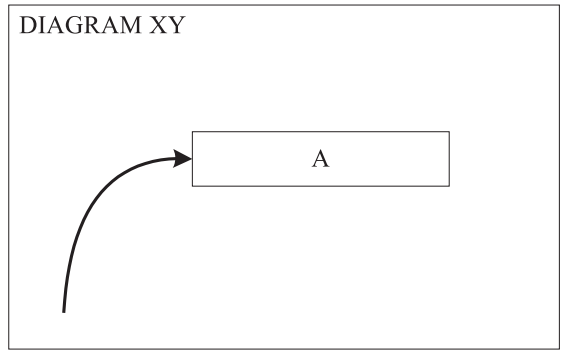
1.3.5 Pohled analytika na diagramy

Jako pomůcku lze uvést následující představu, jak vidí vývojář diagramy v okamžiku, když je navrhuje, resp. poté čte. Na začátku se zavede prvek A a neuvažují se jeho interakce, viz obrázek 1.9.



Obrázek 1.9: První krok – zavedení prvku.

Tento prvek se zavádí v diagramu proto, aby byl nějak použit (POZN.: *nemá smysl zavádět prvky, které nikdo nikdy nepoužije*). Tuto interakci použití znázorníme jako šipku na tento prvek, viz obrázek 1.10.

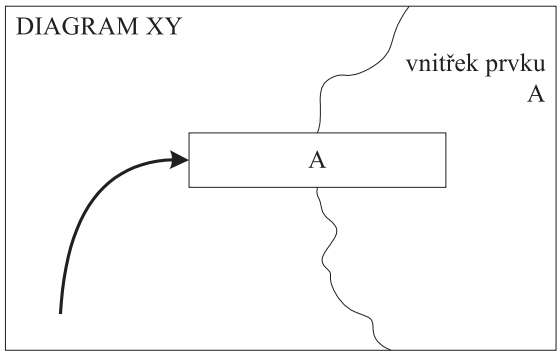


Obrázek 1.10: Použití prvku nějakým jiným prvkem.

Šipka na obrázku 1.10 tedy dává vědět, že někdo tento prvek použije. U tohoto obrázku je dobré si v duchu představit, že prvek je rozdělen na „vnějšek“ a „vnitřek“. Vnějšek je konec šipky u prvku, vnitřek je třeba si představit jako na obrázku 1.11 (POZN.: *tuto představu samozřejmě nekreslíme, pouze si ji v duchu vybavujeme!*).

Představa znázorněná na obrázku 1.11 (v ostrém diagramu projektu se samozřejmě nekreslí) je velmi názorná: Existuje prostor venku mimo prvek A, v tomto prostoru se vyskytují klienti prvku A, kteří jej používají. Na obrázku je tento prostor znázorněn vlevo od prvku A jako prostor se šipkou. Tito klienti něco potřebují a prvek A je schopen jim odpovídat.

Existuje také druhý prostor, to je prostor označený jako vnitřek A. Tam jsou implementovány služby těchto odpovědí pro klienty prvku A. Tyto implementace však mohou využívat další interakce jiných prvků. Analytik v prvé chvíli samozřejmě neví, jak budou tyto služby implementovány. Na základě požadavku „co bude prvek umět“ (vnější pohled na

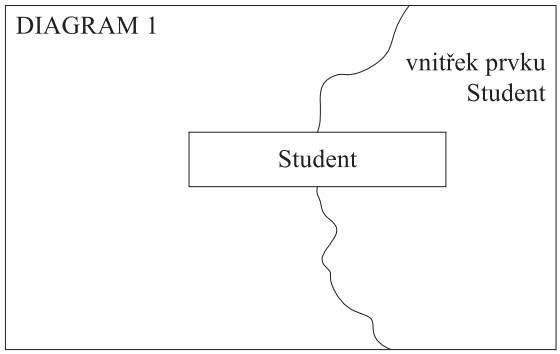


Obrázek 1.11: Rozdělení prostoru na dvě části.

prvek zleva) se snaží najít takovou implementaci této odpovědi, aby byla logicky co nejlepší (vnitřní pohled zprava).

To je v podstatě základní myšlenka analytického návrhu.

Jako příklad tohoto postupu lze uvést právě již zmíněný případ s osobami. První krok – žádá se evidovat studenty, proto vznikne představa znázorněná na obrázku 1.12.



Obrázek 1.12: Rozdělení prostoru na dvě části u prvku Student.

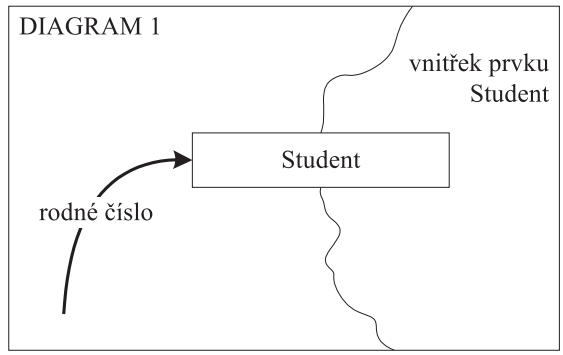
Pokračuje se dále v úvahách: Zákazník se vyjádřil, že „student obsahuje rodné číslo a studenty budeme podle této informace rozlišovat“. Pozor, nyní si analytik musí v duchu přeložit tuto větu do správné analytické podoby: „Student bude umět vydat rodné číslo“, viz obrázek 1.13.

Analytik se dále dozví, že totéž platí také o zaměstnancích, tj. analogicky nakreslí představu znázorněnou na obrázku 1.14.

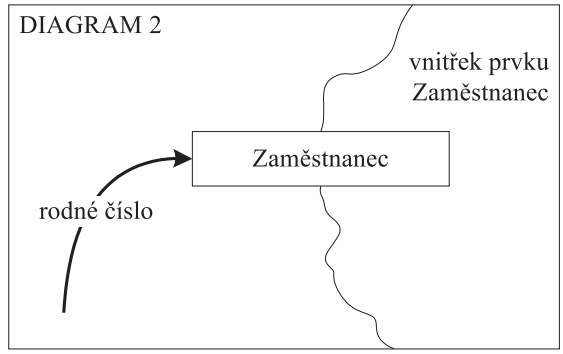
Nyní dá obě představy dohromady. Protože se žádá, aby rodná čísla byla v systému unikátní a oba prvky, jak zaměstnanec, tak student by jej přitom uměli vydat, navrhnou se vnitřky prvků Zaměstnanec a Student pomocí interakce s prvkem Osoba, která bude mít rodné číslo, viz obrázek 1.15.

Nesmíme zapomenout na to (a z obrázku 1.15 je to mimochodem patrné), že i na prvek Osoba je uplatněno objektové paradigma. Znamená to, že i prvek Osoba může být požádán o služby a ví se, že jeho vnitřek lze v dalších úvahách (případně a pokud třeba) opět „překopat“.

Sledujte aktuální novinky na http://www.objects.cz



Obrázek 1.13: Žádost o rodné číslo po prvku Student.



Obrázek 1.14: Žádost o rodné číslo po prvku Zaměstnanec.

1.3.6 Základní dva důsledky objektového paradigmatu

Objektové paradigma zavedené v předešlých dvou odstavcích má dva základní důsledky:

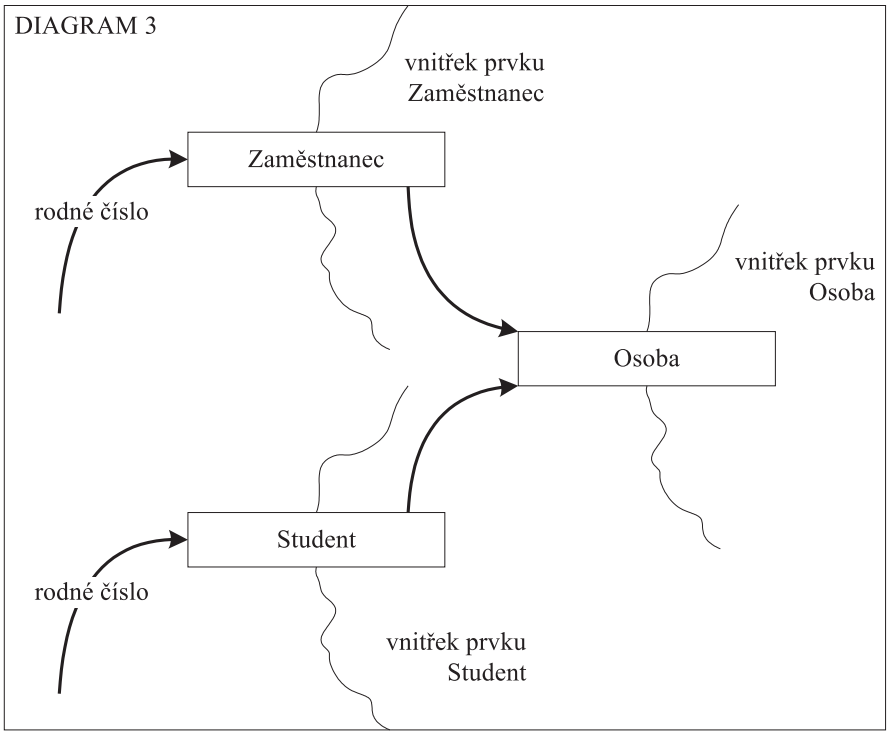
1. Klient prvku vidí u prvku pouze služby a nevidí jejich implementaci. Znamená to, že klient má u prvku zpřístupněny pouze služby (tlačítka s nápisy) a nevidí, jak jsou tyto služby implementovány (nevidí „dráty“ za tlačítky).

2. Pro implementaci služby (tj. vnitřek) platí tzv. anonymita klienta. To znamená, že když se navrhuje logika implementace služby, klient je v té chvíli anonymní a je mimo jakoukoliv kontrolu prvku.

První důsledek je zřejmý, souvisí se zavedením objektového paradigmatu a byl již vysvětlen v předešlé kapitole. Nyní si vysvětlíme druhý důsledek objektového paradigmatu – anonymitu klienta.

1.3.7 Anonymita klienta

Anonymitu klienta bychom mohli jednoduše vystihnout slovy: „Implementace služby (tj. vnitřek služby objektu) nikdy neví, kdo stiskne tlačítko, a nemá v moci, co se děje venku mimo prvek“. Vyplývá z toho důležitý a praktický závěr: Při návrhu objektu se nelze spoléhat na chování klienta.



Obrázek 1.15: Delegování služby v objektovém paradigmatu.

Důsledek objektového paradigmatu „anonymita klienta“ vyplývá přímo z principu opětovné použitelnosti: Jestliže se nabízí prvek k tomu, aby byl opětovně použit (například vystavením do knihovny kódu), tak se nabízí komukoliv, kdo jej může použít. Slovo „komukoliv“ je zde synonymum pro anonymního klienta.

Vysvětlující příklad
V jedné firmě v Čechách, která byla dceřinou společností německé firmy, jsem při školení vysvětloval anonymitu klienta. Jeden z účastníků při výkladu tak trochu žertem poznamenal, že zná vyšší princip, než je anonymita klienta a že je to tzv. „německý ordnung“. Hned to také vysvětlil: Jejich německý šéf se po příjezdu do české firmy hodně divil, proč se u editačního políčka, které nesmí pustit nulovou hodnotu, při testování pečlivě ověřuje, že systém skutečně tuto nulovou hodnotu nepustí dál. Německý šéf argumentoval tím, že v manuálu přece stojí, že „obsluha nesmí zadat nulu“, takže tyto testy jsou zbytečné. Chvíli trvalo, než čeští pracovníci pochopili jeho zvláštní způsob uvažování: Dotyčný německý šéf si totiž nedovedl představit, že by obsluha zadala do políčka nulu, když to má zakázáno. Po vysvětlení, že německá obsluha by to asi opravdu neudělala, ale česká první, co by zkusila, by zadala nulu, německý šéf připustil, že toto testování je nutné.

Tento příklad o „německém ordnungu“ přesně vystihuje podstatu problému anonymity klienta. Zmíněný německý pracovník se spoléhal na to, že obsluha nezadá nulu, a to je ignorace anonymity klienta.

1.3.8 Nepovolené metodiky pro chování klienta jako důsledek porušení anonymity klienta

Princip správného návrhu prvku spolu se zohledněním anonymity klienta znamená držet se pravidla „žádný německý ordnung“. Klient je „nespolehlivý element“ a nesmíme se na jeho chování spoléhat. Existuje jediná možnost, jak ignorovat důsledky anonymity klienta: Vydá se metodický pokyn pro klienta objektu. Jinak řečeno, porušování anonymity klienta předpokládá vydávat metodické pokyny pro prostor mimo objekt, které objekt nemůže mít pod kontrolou. Praktický důsledek anonymity klienta znamená nutnost přemýšlet nad tzv. „blbovzdorností objektu“. Anonymita klienta pro každý objekt nebo prvek (tedy nejenom pro celý systém jako v předešlém příkladu) znamená, že při návrhu objektu není čisté opírat se o „německý ordnung“ mimo prostor objektu a dodržení anonymity klienta vede ke stabilitě objektu vůči jakémukoliv chování klienta (to je ona požadovaná „blbovzdornost“).

Stejné pravidlo platí například i v objektovém programování: Jestliže autor odevzdává do knihovny objektovou třídu k použití, tak by se neměl spoléhat na to, co bude muset dělat naprogramovaný klient, aby tyto objekty z dané třídy správně a logicky fungovaly. V důsledku to znamená, že při návrhu objektu nejde jen a pouze o rozdělení kódu na dvě části: vnitřek a vnějšek objektu, tj. nejde jen o rozmístění kódu „co je venku a co uvnitř“. Objektové paradigma a následná anonymita klienta vedou k tomu, že vnitřek objektu je „uzavřený stabilní svět s konzistentními stavy“ odpovídajícími tomu, co u objektu postupně voláme. Tento svět je schopen relevantně reagovat na jakékoliv chování okolí a nespoléhá se na logiku tohoto okolí. Samozřejmě, může se stát, že objekt bude na některá chování klienta reagovat dost podrážděně, například vyhazovat výjimky. Porušení anonymity klienta může v tomto případě například znamenat, že objekt tyto výjimky nebude vyhazovat a spoléhá se na to, že klient nepoužije daný objekt zrovna takto.

1.3.9 Dokumentace objektu a anonymita klienta

Jak už bylo řečeno, pokud se poruší anonymita klienta, musí se vydat metodický pokyn pro chování mimo objekt, aby daný objekt fungoval správně. V předešlém příkladu s „německým ordnungem“ musel být vydán metodický pokyn, že obsluha nesmí zadat nulu a obsluha to musí dodržet.

Poznámka: *Pokud by se v příkladu dbalo na anonymitu klienta, tento pokyn obsluze bude sice vydán také, ale pokud jej obsluha poruší a zadá nulu, tak to systém ošetří, reaguje na to a nespoléhá se na klienta!*

Pokud se však objekt navrhne „čistě“ s ohledem na anonymitu klienta, tak veškerá nutná dokumentace, kterou potřebuje tvůrce klienta (programátor používající objekt), spočívá pouze v popisu služeb a jejich implementací. Jinou dokumentaci již vývojář, který programuje okolí objektu, nepotřebuje. Pokud se však objekt (obecně prvek) navrhne „nečistě“, potom kromě popisu služeb a jeho implementací musíme dodat ještě dokumentaci pro nutné chování klienta, jinak se dvojice „klient-prvek“ nebudou chovat vždy korektně.

1.4 Vzor *Dichotomie třída-instance*

Dichotomie (z řeckého dicha = nadvakrát a tome = řez) je rozlišení dvou kvalitativně odlišných stavů jevu nebo vlastnosti. Často se jedná o odpověď na otázku zjišťovací (ano/ne), není to však podmínkou (dichotomický bývá např. znak pohlaví s hodnotami žena/muž). V podstatě se jedná o rozdělení množiny na dvě

disjunktní části podle nějakého znaku (buď je prvek v jedné množině anebo je v druhé).

Vzor *Dichotomie třída-instance* patří k základním vzorům objektového modelování a vyplývá přímo z axiomu opětovné použitelnosti neboli re-use (viz předešlá kapitola).

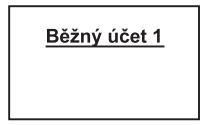
Nechť potřebujeme definovat vlastnosti u nějakého prvku v systému. Vzor Dichotomie třída-instance *předepisuje postup, kdy se tyto vlastnosti prvku nedefinují přímo v tomto prvku, ale mimo něj v tzv. třídě, v ní se tyto vlastnosti definují. Následně se daný prvek prohlásí za prvek pocházející z této třídy a proto má tyto vlastnosti definované v této třídě. Vlastnosti prvku jsou dány příslušností do třídy, kde jsou tyto vlastnosti definovány.*

V systému existuje několik tříd a z nich vznikají celé skupiny prvků stejných vlastností (s různými vlastnostmi mezi skupinami). Prvky ze stejné třídy mají stejné vlastnosti, liší se však hodnotami těchto vlastností (tj. vypadají jako klony), prvky z různých tříd mají odlišné vlastnosti.

Znamená to, že pokud se zavádějí prvky v systému a žádá se definovat vlastnosti těchto prvků, musí se vlastnosti prvků zavádět „jakoby oklikou“: Zavede se nejprve jejich třída a v ní se definují „bokem“ tyto vlastnosti. Teprve poté lze zavádět prvky s vlastnostmi tak, že se musí zadat, ze které třídy daný prvek pochází, a proto má tyto vlastnosti.

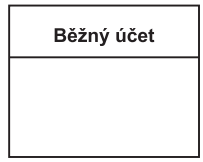
Stejně jako v OOP se v UML daný prvek nazývá instance třídy nebo také objekt z dané třídy. Instance se v modelování značí pomocí obdélníku, uvnitř je název instance, který je podtržen. Třída se značí jako obdélník a název uvnitř není podtržen.

Příklad instance neboli objektu zapsaného v UML (může jich být více) je na obrázku 1.16.



Obrázek 1.16: Evidovaná instance s názvem prvku Běžný účet 1.

Příklad třídy je na obrázku 1.17.



Obrázek 1.17: Příklad třídy.

Synonymem pro třídu jsou v lidské řeči také druh, resp. typ. Synonymem pro prvek ze třídy jsou také instance třídy, objekt ze třídy nebo také výskyt ze třídy.

Jako příklad vztahu třída-instance lze uvést: Daný konkrétní prvek theMravenec, kterého lze spatřit na zemi, má vlastnosti mravence, protože pochází ze třídy Mravenec. V jednom

mraveništi je možné vidět spoustu prvků z téže třídy Mravenec. Roli třídy Mravenec zde představuje „Matka královna Mravenec“, která je rodí. Včely mají jiné vlastnosti, protože je rodí třída „Matka královna Včela“.

Díky tomuto vzoru je celý prostor informačního systému pomyslně rozdělen na dva disjunktní podprostory (zde je patrná zmíněná dichotomie = rozdělení na dvě části). V jednom podprostoru jsou třídy a v druhém instance, které z nich vznikají. Části prostoru s třídami se také říká statická a část prostoru s instancemi se nazývá analogicky část dynamická. Důvod těchto názvů „statická“ a „dynamická“ je ten, že počet tříd neboli typů v již nasazeném informačním systému je neměnný, ale počet instancí se může v běhu programu měnit. Z tohoto důvodu jsou takto rozděleny i modely v UML: Existují modely statické povahy a existují modely dynamické povahy.

Je zřejmé, že jsou-li dány třídy, pak musí platit, že instance, které z nich vznikají, musejí mít vlastnosti definované v těchto třídách. Pro určení vlastností instancí je tedy třeba vždy nalézat třídy, instance jsou z nich poté odvozeny. Z tohoto pohledu můžeme chápat modely instancí jako podružné a pouze jako příklady, protože jsou vždy odvoditelné z modelu tříd.

Pomůcka pro rozlišení třídy a instance je následující: Je-li řeč o vlastnostech evidovaných prvků, je řeč o třídě, a pokud je řeč o příkladu evidence a chování aplikace, pak je řeč o instancích.

> *Model, který zavádí třídy a jejich vztahy, se v UML nazývá model tříd. Jedním z úkolů analytika je nalezení modelu tříd na analytické úrovni (*POZN.*: bude ještě podrobně popsáno).*

Laik tomuto rozdělení na třídy a instance rozumí zdravým selským rozumem intuitivně velmi dobře, jenom si tyto rozdíly neuvědomuje.

Příklad
Analytik hovoří se zákazníkem a ten mu řekne: „Přejeme si evidovat běžné účty. Každý běžný účet bude mít tyto vlastnosti: Bude mít datum založení, klienta, stranu obrat debet, kredit atd.“ V té chvíli je řeč o třídě, tedy v UML by se zavedla třída, definující vlastnosti všech běžných účtů, které z ní vzniknou. Analytik se poté zeptá: „A kolik takových běžných účtů v systému očekáváte, že bude, až se systém naplno rozběhne?“ Odpověď: „Okolo 100 000“. Nyní je řeč o počtu instancí vzniklých ze třídy v předešlé větě. Všimněme si, že údaj o počtu instancí „není udán přesně“, protože počet instancí je dynamická veličina (na začátku je počet velmi malý a postupně roste).

1.4.1 Tipy a triky analytika

Český jazyk nerozlišuje mezi třídou a instancí. Pokud zákazník použije nějaké podstatné jméno, tak teprve až z kontextu celé věty je zřejmé, jestli má na mysli třídu nebo instanci. Některé jazyky se snaží třídu a instanci tak trochu od sebe odlišit a to pomocí určitého a neurčitého členu. Například „the car“ znamená konkrétní auto. Toto odlišení používají některé CASE nástroje, které v některých případech dávají do názvu instance (kromě podtržení názvu) předponu „the“, například theClient apod. Analytik musí velice dobře sledovat, zda se v rozhovoru se zákazníkem pohybuje v prostoru instancí (evidované výskyty) anebo ve třídách (definované vlastnosti těchto instancí).

1.4.2 Použití vzoru *Dichotomie třída-instance* v „papírové evidenci“, tj. „v systému kartoték“

Evidenci postavenou na dichotomii tříd si můžeme výstižně vysvětlit pomocí „papírové evidence“, kterou reprezentuje například obdoba klasické papírové kartotéky na úřadě. Výhodou této představy je to, že základní myšlenky „papírové evidence“ můžeme následně velmi jednoduše ztvárnit za pomocí programování do podoby informačního systému v dané technologii IT.

Představme si, že bychom si potřebovali „něco zapsat neboli zaevidovat“, například chceme evidovat občany, firmy, auta, resp. další prvky. Máme přitom k dispozici jednoduché listy papíru a tužku. Úplně prvním krokem by mohlo být zapsání nestrukturovaných poznámek na velký prázdný list tak říkajíc „bez ladu a skladu“. Takové poznámky jsou sice vhodné na zápis „brainstormingových myšlenek“, ale pro evidenci například zmíněných občanů, firem, aut nebo jiných prvků jsou velmi nepřehledné a tedy absolutně nevhodné. Rozhodneme se proto zavést obdobu papírové kartotéky pomocí „evidenčních kartiček“. Můžeme si jednoduše představit, že pro evidenci prvků zavedeme mechanismus „evidenčních šuplíků“. Například pro evidované osoby se zavede jeden šuplík a v něm se budou vyskytovat kartičky evidovaných osob. V každém daném šuplíku mají všechny kartičky stejný formát. Vypadají jako stejný dotazník vyplněný různými lidmi, tj. mají stejná políčka, například rodné číslo, jméno, příjmení apod., ale každá kartička reprezentuje jeden „zápis“ o jedné evidované osobě a obsahuje svoje vlastní hodnoty v políčkách. Je to podobné, jako když jsou u lékaře evidovány papírové zdravotní karty anebo jako když se evidují čtenáři a knihy pomocí papírové kartotéky v knihovně. Podobně bychom zavedli šuplík pro evidovaná auta a šuplík pro evidované firmy. V každém takovém šuplíku by existovaly kartičky, které mají pochopitelně vždy v daném šuplíku stejný formát, ale šuplík od šuplíku se formát kartiček pochopitelně liší (tj. formát kartičky pro evidovanou osobu je jiný než formát pro evidované auto).

Takováto představa evidenčních šuplíků s kartičkami je názorná a dále nám jasně vysvětlí vzor *Dichotomie třída-instance*, který je pro návrh IS důležitý. Položme si otázku, jakým mechanismem budou vlastně v takovéto kartotékové evidenci vznikat nové evidované kartičky (například nová kartička evidované osoby)? Navrhněme pro naše šuplíky následující obecný mechanismus zaevidování nové informace, tj. vzniku nové evidenční kartičky pro libovolný šuplík:

U každého šuplíku vytvoříme jednu ještě nevyplněnou kartičku daného formátu, která má sice stejný formát, jako ostatní kartičky, avšak nebude se nikdy vyplňovat. Tato jedna dopředu připravená nevyplněná kartička bude přiřazena k šuplíku a slouží pouze jako šablona, tedy jako předloha pro vznik nové evidenční kartičky. Postup tvorby nové evidenční kartičky (například zaevidování nové osoby) je pak velmi jednoduchý: Když přijde požadavek na zaevidování nové osoby, vezmeme nejprve tuto nevyplněnou kartičku přiřazenou k danému šuplíku jako předlohu, tuto kartičku vložíme do kopírky, vytvoříme její kopii a teprve tuto kopii jako novou kartičku vyplníme hodnotami a zasuneme mezi již založené evidenční kartičky do šuplíku. Všimněme si, že ona „původní jedna nevyplněná kartička“ má jiné postavení, než ostatní evidenční kartičky. Kartička vzor slouží pouze jako šablona pro kopírování nových kartiček a pro nic jiného, a proto nebude nikdy vyplněna.

Obecný princip tvorby nových kartiček pro libovolný šuplík můžeme tedy chápat velmi jednoduše: Ke každému šuplíku přiřadíme jednu jedinou nevyplněnou kartičku jako vzor, ta slouží pouze jako předloha ke kopírování a vzniku spousty nových již vyplněných evidenčních kartiček. Znamená to, že každý šuplík obsahuje jednu „kartičku vzor“ a spoustu

„N evidenčních kartiček“, které vznikly kopírováním (klonováním) z této jedné nevyplněné kartičky. Proto mají všechny kartičky v šuplíku stejný formát (vznikly jako klony dané kartičky vzoru), ale šuplík od šuplíku se formáty liší (každý šuplík má svou kartičku vzor).

Je zřejmé, že je třeba rozlišovat mezi „kartičkou vzorem“ (která nebude nikdy vyplněna a slouží pouze ke kopírování) a evidenčními kartičkami v šuplíku (které jsou vyplněny, tj. jejich políčka mají své konkrétní vyplněné hodnoty, vznikly procesem kopírování, proto takových kartiček bude v šuplíku obecně N).

Takto zavedený evidenční papírový kartotékový systém je evidentním a názorným příkladem použití vzoru *Dichotomie třída-instance*:

- Jedna nevyplněná kartička jako vzor u šuplíku (která je vždy pro daný šuplík jen jedna a ze které se jako ze vzoru vykopírují nové kartičky v šuplíku) reprezentuje pro daný šuplík jeho třídu. Evidenční kartičky v šuplíku (kterých je obecně N a které vznikly kopírováním z kartičky vzoru) reprezentují instance této třídy.

- Zatímco kartička vzor je nevyplněna, její kopie tedy instance mají již vyplněné hodnoty a reprezentují konkrétní evidovanou informaci. Každý šuplík má svou „kartičku vzor“, tj. třídu. Třídy (tj. „kartičky vzory“) se šuplík od šuplíku liší.

Naprogramovaný informační systém funguje na stejném principu, pouze „papírové evidenční kartičky“ jsou nahrazeny „naprogramovanými virtuálními kartičkami“, tj. strukturami v databázích, objektovými strukturami, recordy aj.

1.4.3 Co je to vztah META

Primárně je vzor *Dichotomie třída-instance* chápán jako vztah od tříd k instancím, tj. je definována třída, která zavádí vlastnosti instancím. Při vývoji systému a odhalování informací a vztahů mezi nimi však nastává situace přesně opačná: Nejprve jsou patrné evidované instance jako příklady evidence (evidované kartičky jako příklady evidence) a zatím není známo, z jakých tříd pocházejí. Teprve po nalezení vícero příkladů zobecňujeme tyto instance do tříd. Tomuto přechodu se také říká přechod do úrovně meta. Dá se také říci, že pro budoucí instance je model tříd meta pravidlem.

> *Vztah zpětný od instancí ke třídám se nazývá vztah meta.*

Vztah meta není obecně směrem nahoru tranzitivně uzavřen. To znamená, že lze opětovně aplikovat vztah meta na třídy, tj. lze vytvořit vztah, kde prvky jsou třídy a třídami jsou třídy tříd neboli „metatřídy“. To je již vysoká abstrakce (je například použita v definici jazyka UML). Prakticky na problém meta narazíme při analytickém modelování v samotném úvodu *Class Diagramu* při vyhledávání tříd.

1.4.4 Častá chyba záměny názvu instance za její obsah

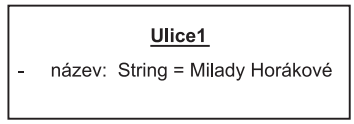
Při vyhledávání instancí jako příkladů evidence bychom se neměli dopustit laické chyby, která spočívá v záměně názvu instance s jejím obsahem. Měli bychom proto dodržovat následující pravidlo:

> *Při volbě odpovídajícího názvu pro evidovanou instanci je třeba vybrat takový název, který je maximálně neutrální a nezávislý na tom, jaké hodnoty bude mít daná instance ve svých atributech. Pokud víme, z jaké třídy pochází instance, zvolíme za název instance nejlépe název třídy ve spojení s číslem, například Osoba1,*

Osoba2, Auto1 apod. Pokud ještě nevíme, z jaké třídy pochází daná instance, zvolíme pro název instance úplně neutrální výraz, například Element1, Element2, Prvek1, resp. neutrální písmena s čísly, například A1, A2 atd.

Podstatné pro zavedení názvu instance totiž není „vystihnout obsah“, ale pouze odlišit jednu instanci od druhé. Například je jasné, že Osoba1 je jinou instancí než Osoba2, stejně tak Prvek1 je jinou instancí než Prvek2. K tomu slouží názvy instancí a k ničemu jinému.

Tato zásada se jeví jako jasná a srozumitelná, ale mnohdy můžeme mít silné nutkání ji porušit. Důvodem je laické nazývání prvků v běžném životě, kde instancím dáváme názvy podle jejich obsahu. Například běžně říkáme „ulice Milady Horákové“ a každý tomu rozumí. V analytickém modelu jako příklad evidence bychom však neměli zavést evidovanou instanci s názvem Ulice Milady Horákové, ale název bychom měli zvolit neutrálně, například Ulice1. Tato instance má atribut název typu string a v něm je hodnota řetězce „Milady Horákové“. Graficky v UML bychom mohli evidovanou instanci jako příklad evidence znázornit obrázkem 1.18.



Obrázek 1.18: Typicky neutrální název instance.

Kapitola 2

Úrovně abstrakce informačního systému

V předešlých kapitolách jsme dospěli k poznatku, že charakteristickou vlastností softwaru, který je tvořen v *Tunelu*, je nedodržování opětovné použitelnosti se všemi katastrofálními důsledky. Logicky z toho plyne, že pokud chceme při výrobě softwaru „opustit *Tunel*“, měli bychom nasadit při vývoji IS takové mechanismy, které povedou v maximální míře k dodržování principu opětovné použitelnosti. Toho však nelze dosáhnout bez zohlednění principu úrovní abstrakce dokumentace, jak si dále ukážeme. Nejprve si však musíme dobře vysvětlit, co se chápe pod pojmem úroveň abstrakce vývoje a dokumentace IS.

Pokud se vyvíjí anebo popisuje informační systém, vždy se tak děje na určité úrovni abstrakce. Každá úroveň abstrakce je určena mírou detailu implementace, tedy tím, jak je popis systému implementačně podrobný. Pro posouzení míry detailu implementace dokumentace může dobře posloužit následující postup:

Otevřeme dokumentaci a začneme ji číst. Čím více se v dokumentaci začnou objevovat prvky z dané technologie, tím více je dokument tzv. implementačně podrobnější, tj. má vyšší míru detailu implementace. Samozřejmě se vzrůstající složitostí podrobností technologie bude dokumentu méně rozumět laik, který není s podrobnostmi technologie seznámen a neumí programovat. Takovému dokumentu bude rozumět například pouze zkušený programátor nebo technolog. Naopak, pokud bude daná část dokumentace projektu napsána nezávisle na technologii, tj. bez jakýchkoliv technologických podrobností a specialit, bude jí rozumět i laik.

Jako příklad ukázky takovéto „technologicky nezávislé“ věty z dokumentace, tj. věty s nulovou mírou detailu implementace, můžeme uvést následující příklad: *Každá evidovaná Osoba má atributy Rodné číslo, Jméno a Příjmení.* Všimněme si, že z této věty nepoznáme, zda je informační systém napsán v databázi ORACLE nebo MS SQL, zda je použito vývojové prostředí C# anebo JAVA. Tato věta má míru detailu implementace 0 %.

Na druhé straně existuje jiná část dokumentace, kde se již vyskytuje přímo kód, například v relační databázi příkaz ve tvaru:

```
CREATE TABLE TOsoba
(
 id_osoba int IDENTITY(1,1),
 rc VARCHAR(10),
 jm VARCHAR(16),
 pm VARCHAR(30)
)
```

Sledujte aktuální novinky na http://www.objects.cz

V tomto případě má tato „věta“ velmi vysokou míru detailu implementace. Když ji čteme, musíme rozumět dané technologii a danému prostředí databáze. Zatímco první větě laik rozumí, danému příkazu rozumí jen vývojář seznámený s danou technologií.

> *Každá úroveň abstrakce je dána mírou detailu implementace. Míra detailu implementace určuje, jak moc jsme implementačně podrobní a tedy do jaké míry je vyjádření (syntaxe) v dokumentech vývoje poplatné dané technologii.*

O tom, jak důležité je pro opuštění metodiky *Tunel* zohledňovat při vývoji úrovně abstrakce, svědčí následující příklad ze života softwarové firmy tápající *Tunelem*:

Představme si rozhovor dvou přátel programátorů z firmy vyrábějící bankovní systémy. Oba vývojáři jsou z různých divizí a náhodou se potkají například v jídelně, kde se dají do hovoru. První povídá: „Tak dělám na agendě Pokladna, právě programuji výběr peněz z pokladny a následné převody na vnitřních účtech banky...“ a druhý překvapeně zvolá: „Ale já taky dělám na agendě Pokladny a taky dělám převody na vnitřních účtech banky!“ Dalším rozhovorem oba zjistí, že vlastně pracují na tomtéž. Samozřejmě to svědčí o podivném stavu organizace práce ve firmě, ale nejedná se o až tak výjimečnou situaci.

V softwarové firmě není vůbec jednoduché zavést takové postupy prací, při nichž by se vývojáři vyvarovali situacím s opakujícím se řešením, jak se může na první pohled zdát. Oba programátoři v té chvíli, když mluvili o své práci, vůbec nehovořili jako programátoři. Vyjadřovali se pouze v pojmech a funkcionalitách bez implementačních podrobností a nikoliv v programátorských pojmech. Všimněme si, že teprve tímto rozhovorem „o co tam jde“ zjistili, že pracují na tomtéž. Dokonce kdyby si navzájem ukázali pouze kód, tak by se nad ním nejprve zamysleli, převedli by tento kód do úrovně vyšší abstrakce (lidově řečeno přelouskali by jej do podoby „o co tam jde“) a potom by jeden z nich zvolal: „Ale to je přece totéž, co dělám já, to je také agenda Pokladny!“

Každý kód má svou logiku vyjádřitelnou pouze pojmy bez implementačních podrobností a na této úrovni se vyjadřujeme, když o daném kódu hovoříme ve smyslu „o co tam jde“.

Je zde zřetelně vidět, že vývoj informačního systému a jeho následná dokumentace musí podléhat úrovním abstrakce. Prakticky se doporučuje zavést tři úrovně abstrakce, tj. tři okruhy dokumentace. Doporučené úrovně abstrakce IS jsou následující:

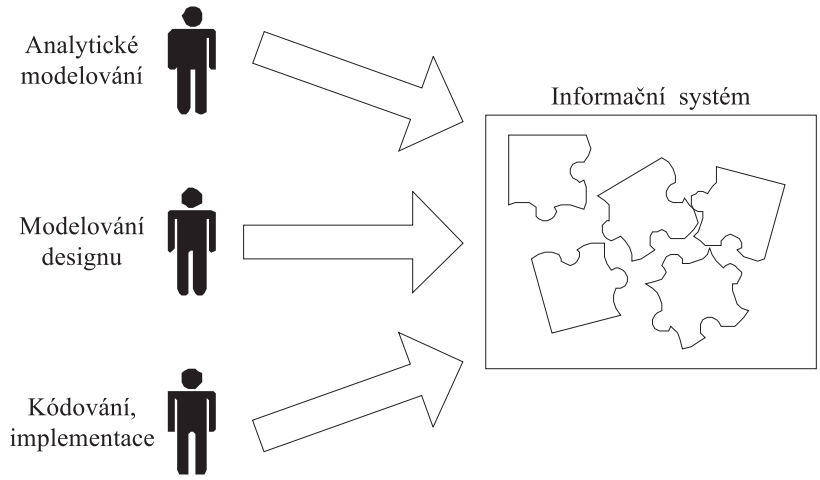
- *Analytické modelování (míra detailu implementace nula), dále také „AM“;*
- *Návrh designu (sice model, ale poplatný výrazově danému prostředí), dále také „D“;*
- *Kódování (nejnižší úroveň abstrakce, maximální detail implementace), dále také „K“.*

Vidíme, že na jednom pólu abstrakce je *kódování* (má nejvyšší míru detailu implementace, tj. vyšší už není) a na straně druhé *analytické modelování* (míra detailu implementace je nula). Střední úroveň mezi nimi se nazývá *modelování designu* (někdy také jen design, návrh technologie aj.).

Doporučuje se, aby takto byly rozděleny práce, role a dokumentace v projektu, abychom neskončili jako zmínění dva programátoři v rozhovoru nad agendou Pokladna.

Je třeba mít na paměti, že všechny tři úrovně abstrakce mají jako předmět popisu stejný informační systém, tj. obsahově musí být shodné, liší se způsobem a syntaxí popisu. Představu o úrovních abstrakce ukazuje obrázek 2.1.

Sledujte aktuální novinky na http://www.objects.cz



Obrázek 2.1: Úrovně abstrakce.

2.1 Nejnižší úroveň abstrakce zvaná Kódování

Nejnižší úroveň abstrakce, tedy úroveň s nejvyšší mírou detailu implementace, se nazývá Kódování. Vyšší míra detailu implementace už neexistuje. V tomto pohledu je na informační systém nahlíženo pouze jako na kód a jeho realizaci. Jedná se o pohled programátora, který tento kód tvoří. Míní se tím nejenom veškerý kód v daném programovacím jazyce (např. JAVA, C#, Delphi apod.), ale i SQL příkazy, skripty apod.

Tato nejnižší úroveň abstrakce se také nazývá *implementační úroveň* nebo *úroveň realizace* anebo prostě *kódování*, což budeme nadále jako název této úrovně používat. Výslednými dokumenty, tj. artefakty, které ztvárňují myšlenky z této úrovně abstrakce, jsou zdrojové kódy, zkompilované balíky souborů, SQL skripty apod., tedy to, co v konečném důsledku opravdu fyzicky realizuje a fyzicky instaluje informační systém. Úroveň abstrakce Kódování představuje samu fyzickou podstatu IS. U jednoduchých „prográmků“ lze mnohdy přímo napsat kód bez potřeby dokumentace dalších úrovní abstrakce, protože zde „vše může být v hlavě“ jednoho nebo několika málo vývojářů. Jenomže problém je v tom, že každý větší systém mnohdy začínal jako malý a postupně se rozrostl do podoby, kdy už „nelze mít vše v hlavách“.

> *Nejnižší úroveň abstrakce, tedy s nejvyšší úrovní detailu implementace, je kódování. Jazyk úrovně abstrakce kódování je složen z rezervovaných slov daného prostředí. Jedná se o náhled programátora.*

2.2 Nejvyšší úroveň abstrakce zvaná Analytické modelování

Protipólem proti nejnižší úrovni abstrakce informačního systému je nejvyšší úroveň abstrakce, která se nazývá *analytické modelování*. Míra detailu implementace na této úrovni je nula, tj. dokumenty z této úrovně jsou výrazově implementačně nezávislé. Na této úrovni se popisuje podrobně informační systém v nejvyšší možné abstrakci pouze pomocí evidovaných pojmů bez jakýchkoliv implementačních podrobností.

Sledujte aktuální novinky na http://www.objects.cz

Nejvyšší úrovní abstrakce popisu IS je analytické modelování, které má míru detailu implementace nula. Na této úrovni abstrakce se popisují evidované pojmy s jejich vztahy a popisuje se chování výskytů z těchto pojmů.

Na úrovni abstrakce analytického modelování se pohybují všichni účastníci projektu včetně „laiků“, kteří nazírají na informační systém jako na souhrn evidovaných pojmů a na chování výskytů těchto pojmů. Z pohledu této úrovně abstrakce systém „něco provádí“, „něco umí“, „něco eviduje“, „je k nějakému užitku“ apod. Aniž by se na této úrovni zaváděly jednotlivé elementy z programovacího jazyka, popisuje se podrobně, co systém provádí, jaké informace se evidují a jakou mají tyto informace skladbu. Používají se pouze pojmy jako synonymum pro „evidované informace“ (například faktura, běžný účet apod.). Popisuje se také chování výskytů z těchto pojmů.

Popis systému na úrovni abstrakce analytického modelování je oproštěn od prvků a syntaxe daného programovacího jazyka a je ve svém vyjádření implementačně nezávislý.

Myšlenky ztvárněné na této úrovni abstrakce mají povahu modelů informačního systému. Efektivním a standardním jazykem pro jejich zápis je modelovací jazyk UML (Unified Modeling Language). Velmi přesná syntaxe jazyka UML vede k tomu, že nevznikají „podivné“ dokumenty analytického modelování, které nemají hlavu ani patu, ale tvoří se technické dokumenty na vysoké úrovni abstrakce pomocí určitých pravidel a to technicky přesné syntaxe jazyka UML. Možnost použití jazyka UML pro tvorbu dokumentů na úrovni analytického modelování je dána využitím vzoru *Dichotomie třída-instance*:

Na úrovni analytického modelování jsou uživatelem zavedené evidované pojmy převedeny do modelů v UML jako takzvané analytické třídy a výskyty z těchto pojmů jsou zavedeny jako objekty neboli instance z těchto tříd.

Již zmíněný vzor *Dichotomie třída-instance* vede k zásadní možnosti použít UML pro modelování na úrovni analytického modelování. Pojmy zavedené uživatelem se namapují na tzv. analytické třídy. Analytická třída zavedená v UML jako prvek modelu pak reprezentuje evidovaný pojem zavedený uživatelem.

Nyní je důležité upozornit na tu skutečnost, že předmětem analytického modelování je samotný informační systém a nic jiného. Znamená to, že se jedná o nejvyšší abstrakci napsaného programu, přičemž samotný program je realizován na nejnižší úrovni – kódování. Modely analytického modelování nejsou nějakými „obecnými“ větami, ale konkrétním popisem programu, i když úroveň abstrakce je vysoká.

Pomocí modelů v UML se na úrovni abstrakce analytického modelování odpovídá na základní otázku „*Co?*“, tj. odpovídá se na otázku „o co v informačním systému jde, co se eviduje a jak se výskyty informací chovají“. Na této úrovni se pohybují všichni účastníci projektu, včetně těch, kteří neumějí programovat, ale také těch, kteří programovat umějí. V okamžiku, kdy se vyžaduje takzvaně „vysvětlit“, k čemu systém slouží, co eviduje a jak eviduje apod., automaticky se pohybujeme na této nejvyšší úrovni abstrakce.

Úroveň abstrakce analytického modelování je používána všemi účastníky projektu včetně uživatelů, resp. expertů na danou problémovou doménu, tj. z hlediska tvorby informačních systémů „laiků“. Při doporučeném modelování v jazyce UML je však třeba provést pro laika výstupy z této úrovně, neboť jazyk UML má své vyjadřovací způsoby odborné povahy, které i přes vysokou abstrakci vyjadřování nemusejí být zcela srozumitelné úplně každému. Stačí však jen určité drobné úpravy, aby se staly snadno srozumitelnými i laikovi. Díky této

Sledujte aktuální novinky na http://www.objects.cz

skutečnosti se mohou k informačnímu systému vyjadřovat další účastníci projektu, kteří jinak netvoří „programátorský“ tým, což jsou například externí konzultanti, uživatelé, obchodníci apod.

2.3 Střední úroveň abstrakce zvaná Modelování designu

Mezi úrovní abstrakce analytického modelování a kódování se zavádí ještě jedna úroveň, která se nazývá *modelování designu* (dále také zkratka „D“). Synonymem pro tuto úroveň je také název *modelování návrhu* nebo také *technologický návrh*, resp. *design*. Jedná se o úroveň abstrakce, která je ještě stále hodně abstraktní v tom smyslu, že se nejedná o samotný realizovaný a fyzicky viditelně fungující systém. Stále se ještě jedná o modely, ale tyto modely jsou již technologickým návrhem v daném vývojovém prostředí. Jinak řečeno, jsou to modely poplatné danému prostředí, které navrhují neboli zadávají realizaci v daném vývojovém prostředí. Patří sem například návrh relační databáze, návrh obrazovek, model tříd pro daný OOP jazyk, komponentní model v daném prostředí atd. Dalo by se také říci, že tato úroveň „překlápí“ vysoce abstraktní popis IS z úrovně „pouze“ evidovaných pojmů s nulovou mírou implementace (AM) do podoby techničtějšího modelu, blížícímu se danému prostředí, v němž bude kód napsán.

Zatímco základní otázkou analytického modelování je „Co?“, tj. logicky o co tam jde, tak základní otázkou modelování designu je „*Jak?*“ (avšak nikoliv „jak“ ve smyslu algoritmů, ale ve smyslu jak bude navržená myšlenka z analytického modelování ztvárněna do konkrétní technologie). Dokumenty z této úrovně tvoří experti na technologie.

Pro modely designu je možné opět použít jazyk UML. Rozdíl oproti úrovni abstrakce analytického modelování je však ve významu pojmu třída a instance:

> *Na rozdíl od úrovně abstrakce analytického modelování, kde jsou třídy evidovanými pojmy v IS, na úrovni modelování designu se již jedná o třídy (neboli typy) z konkrétního vývojového prostředí (například objektové třídy v jazyce C# nebo v JAVA jazyce, nebo tabulky v MS SQL, v ORACLE atd.)*

Modely designu jsou tedy na nižší, implementačně a technologicky konkrétnější úrovni abstrakce, ale jsou stále ještě abstrakcí (nedají se zkompilovat).

Je třeba zdůraznit, že se stále jedná o návrh, tj. o model. Prvky designu tedy ještě nejsou konkrétními již realizovanými prvky (což je až kód, tedy ještě nižší úroveň abstrakce), ale jsou pouze technologickým návrhem.

> *Modelování návrhu je zadáním pro nejnižší úroveň abstrakce, tj. pro kódování.*

Jedním z charakteristických průvodních jevů systému vyvíjeného bez ohledu na existenci úrovní abstrakce, (tj. chybí-li při vývoji analytický model a jeho technologické řešení – návrh designu), je stále opakující se požadavek programátorů: „Dejte nám dobré zadání, potom jsme schopni systém dobře realizovat!“ Jedná se vlastně o důsledek zanedbání existence úrovní abstrakce. Každá vyšší úroveň abstrakce se stává zadáním pro nižší úroveň abstrakce. Artefakty úrovně abstrakce analytického modelování jsou zadáním pro tvorbu modelů návrhu technologie a následně modely návrhu jsou zadáním pro kódování.

2.4 Fázování projektu

Vždy, tedy i při tvorbě toho nejjednoduššího softwaru, je třeba se nejprve zamyslet „o co tam jde“, potom je třeba se zamyslet „jak to navrhnout v daném prostředí“ a následně se

vytváří kód. Při tvorbě softwaru tedy vždy existují uvedené úrovně abstrakce analytického modelování, designu a kódování. Vždy se při vývoji informačního systému těmito úrovněmi prochází, jedinou otázkou je, zda jsou tyto „průchody“ dobře zdokumentovány. Je nutno podotknout, že většinou (s výjimkou kódu, který se musí vždy odevzdat) velmi špatně.

> *Zavedení úrovní abstrakce vede k fázování prací projektu a tedy k přechodu z jedné úrovně do druhé. Přechod z úrovně analytického modelování do úrovně designu se nazývá mapování do designu, přechod z designu do kódování se nazývá realizace nebo také implementace.*

Uvedené fázování prací však neznamená, že se musí celý systém nejprve namodelovat na analytické úrovni, pak provést celý design a poté vše kódovat. Existuje několik možných škol projektového řízení, které zavádějí postupy, jakým způsobem procházet úrovněmi abstrakce.

Tento průchod může být například dost hektický a bouřlivý. Při samotném procesu vývoje díky zpětným vazbám se může analytický model stále opravovat, tj. v okamžiku vývoje nemá analytické modelování v dané chvíli stoprocentně formalizovanou podobu. Mělo by však vždy platit, že když se dokončuje určitá stabilní verze části SW, tak by měly být pokud možno co nejlépe zdokumentovány všechny jeho tři úrovně abstrakce a také postup mapování z analytického modelu do designu.

2.4.1 Požadavek na oddělení dokumentace úrovní abstrakce a požadavek na čistotu analytických dokumentů

Ať už probíhá fázování projektu (tj. průchod úrovněmi abstrakce) jakkoliv, jeden z důležitých požadavků na dokumentaci projektu spočívá v nutnosti vždy od sebe oddělit dokumentaci z různých úrovní abstrakce. Tento problém se ani tak netýká samotného kódování, přece jen povaha tvorby kódu (tj. psaní kódu v textovém tvaru) je natolik odlišná od modelování, že se technicky ani nedá „promíchat“ s dokumenty modelování v UML. Tento požadavek na „neprolínání dokumentace“ se týká dvou vyšších úrovní abstrakce, tj. analytického modelování a technologického návrhu. Neměly by se tedy vytvářet dokumenty typu „miš-maš“, ve kterých se vyskytují prvky jak z úrovně analytického modelování, tak prvky z technologického návrhu (tedy designu). Jako příklad klasické chyby „zamíchání pojmů z technologického návrhu do analytického dokumentu“ bych uvedl často používanou definici tabulek zavedenou již v analytickém dokumentu.

Měl jsem osobně možnost provádět řadu auditů a oponentur u analytických dokumentů různých firem (v ČR a v SR) a mohu díky těmto bohatým zkušenostem potvrdit, že právě „míšení“ pojmů z technologického návrhu do dokumentů analytického modelování je jedním z nejčastějších prohřešků analytika.

Poznámka: *Je vcelku pochopitelné, že této chyby se většinou dopouštějí vynikající technologové, kteří jsou „převeleni“ do role analytika a mají za úkol vytvořit analytický dokument.*

2.4.2 Úrovně abstrakce a jejich nezbytnost dokumentace

V předešlých kapitolách byly popsány tři úrovně abstrakce, tj. úroveň analytického modelování, technologického návrhu (designu) a kódování (realizace).

Pokud se podíváme na dokumentaci projektu tvořenou v *Tunelu*, zjistíme, že v ní nejsou zastoupeny dokumentace všech těchto tří úrovní. Pochopitelně nalezneme určitě poslední realizační část (tj. dokumenty kódování a implementace), ta tam nechybí, protože se žádá fyzicky vytvořit a odevzdat zákazníkovi jako existující fungující systém a bez něj by byl

Sledujte aktuální novinky na http://www.objects.cz

zákazník velmi nespokojen. Ale kromě kódu tvořeného v *Tunelu* již další dokumentaci buď nenalezneme anebo ji nalezneme velmi slabou. Mohlo by se proto zdát, že tyto tři zmíněné úrovně abstrakce neexistují vždy a nejsou tedy povinně zavedené.

Problém je však v tom, že všechny tři úrovně abstrakce (tj. analytické modelování, technologický návrh a kódování) existují vždy, ať chceme nebo ne, pouze nebývají vždy zdokumentovány. Myšlenky z těchto vyšších úrovní abstrakce totiž mnohdy zůstávají pouze v hlavách vývojářů a nejsou tedy odevzdány ve formě dokumentace.

Pro návrh informačního systému a tvorbu softwaru platí jednoduché pravidlo: Vždy, když tvoříme nějaký software (například část informačního systému), vždy se musíme nejprve zamyslet „o co tam jde logicky“, následně musíme vymyslet „jak toto logické realizovat v daném vývojovém prostředí“ a poté napíšeme samotný kód, tj. systém vytvoříme a nasadíme u zákazníka. Všimněme si, že předešlé výrazy „o co tam jde logicky“ a následně „jak toto realizovat v daném vývojovém prostředí“ nejsou ničím jiným, než synonymem pro úrovně abstrakce analytického modelování („o co tam jde logicky“) a technologického návrhu neboli designu („jak toto realizovat v daném vývojovém prostředí“).

Zřetelně vidíme, že myšlenky poplatné těmto úrovním abstrakce existují zákonitě vždy a proto se jim nikdy nevyhneme. Je pouze otázkou, zda jsou myšlenky z vyšších úrovní abstrakce někde nějak rozumně zapsány nebo (jako v případě vývoje v *Tunelu*) nikoliv. Je třeba vzít také v úvahu, že zapsaná myšlenka je i myšlenkou více či méně promyšlenou. Z běžného života dobře víme, jak je někdy „myšlenka v hlavě“ záludná. Teprve když si začneme svůj nápad rozepisovat na papír, vezmeme metr a kalkulačku, mnohdy musíme svou myšlenku upravit nebo zcela přehodnotit. Pokud bychom ji začali realizovat ihned, mohli bychom se snadno dostat do problémů, protože v samotné realizaci to nemusí vycházet tak, jak jsme si původně představovali. Navíc zapsání dané myšlenky do zdokumentované podoby může vést k posouzení její správnosti i jinými účastníky projektu, než je sám autor.

2.5 Úrovně abstrakce a diagramy UML

Jazyk UML umožňuje navrhovat modely informačního systému jak v analytickém modelování, tak ve fázi návrhu technologie. Nabízí sadu diagramů, jejichž význam a účel by měl vývojář dobře znát, aby tyto diagramy mohl používat správně a korektně. Jednotlivé diagramy UML lze přirovnat ke specializovaným nástrojům modelování. Podobně když otevřeme skříňku s nástroji a bereme do ruky kladivo, šroubovák apod., tak i diagramy se používají za určitým účelem a v určité fázi vývoje. Cílem této přehledové kapitoly je seznámit čtenáře s výčtem diagramů, které poskytuje jazyk UML, a současně vysvětlit, k čemu a kdy se používají v projektech a to i s ohledem na úrovně abstrakce. Na konci kapitoly si uvedeme důležitost a nezastupitelnost těchto diagramů z hlediska návrhu IS.
Poznámka: *Pro označení diagramů budeme uvádět zásadně anglické názvy, protože se jedná o rezervovaná slova z UML. Teprve až sekundárně budeme používat pokud možno co nejbližší český překlad.*

V nadpisu názvu diagramu také uvedeme zkratkou, na které úrovni abstrakce se daný diagram používá. Dále budeme pro potřebnost diagramů rozlišovat dva protipóly tvořeného softwaru:

- *systémy evidenční* nebo také *podnikové* neboli *obchodní* (anglicky *enterprise*, resp. *business*). Jsou to systémy, které jsou určeny pro podporu lidské činnosti, evidenci apod. Spadají sem systémy, jako jsou bankovní informační systémy, účetnictví, evidence na poště atd.

Sledujte aktuální novinky na http://www.objects.cz

- *systémy* nazývané jako *technologické*. Nepodporují přímo lidskou činnost a nemají povahu evidence, ale podporují činnost nějakého technického zařízení, přístroje anebo jiného softwaru. Například řešený software je vložen do jiného softwaru (komunikační modul, ovladač, firewall apod.), nebo software ovládá stroje, čidla, atd. (tzv. „embedded systems"). Většinou bývá napsán „na hodně nízké úrovni" (např. v C, Assembleru apod.). Tyto systémy nebývají složité svou rozsáhlostí, ale samotným návrhem a programovacími technikami, jako je například alokace při nedostatku paměti, paralelní programování, zpracování přerušení atd.

Samozřejmě existují systémy napůl evidenční a napůl technologické. Obsahují svou část technologickou a současně i něco evidují. Například řídicí systém celého mrakodrapu, který ovládá všechna čidla, výtahy, klimatizaci, protipožární systém atd. a současně vede i evidenci těchto prvků (pokoje, chodby, panely, uživatele, hesla atd.), je takovou ukázkou současně složitého evidenčního systému a také přitom složitého systému technologického.

***Use Case Diagram* (AM)**

Nejlepší český překlad je asi *Diagram případů užití*. Jeden případ užití (tj. *Use Case*) se chápe přesně ve významu tohoto slovního spojení: Venku se stane událost potřeby použití systému, tj. někdo nebo něco z okolí potřebuje něco od systému, poté přistupuje k systému a provede se (říká se také instanciuje se) „jeden případ užití". Synonymum pro laika je také „funkcionalita systému" viděná v kontextu laika budoucího uživatele takto: „něco/někdo venku" – „potřebuje" – „použije", a to, co se v navrhovaném systému vyvolá použitím, je jeden případ užití. Diagram se tvoří ve fázi analytického modelování jako jeden z prvních diagramů, tj. začíná se jím. Některé školy modelování jej díky tomu zařazují přímo do fáze „Requirements", tj. fáze požadavků na systém. Není to úplně přesné, protože modelování pomocí případů užití je již návrhem IS z pohledu funkcionalit, které tento systém nabízí. Před tím však proběhla ještě skutečná fáze sběru požadavků, tj. vznik a správa „haldy požadavků", jejich uspořádání, schválení apod. To ještě není modelování a není tedy obrazem systému. Z pohledu úrovní abstrakce tvorba *Use Case Diagramu* již spadá do modelování systému na nejvyšší úrovni abstrakce. Tomuto diagramu bude v knize věnována poměrně velká pozornost.

***Class Diagram* (AM, D)**

Nejlepší český překlad je nejspíš *Diagram tříd*. Vyjadřuje model typů neboli tříd (význam třídy viz předešlá kapitola *Dichotomie třída-instance*). Používá se jak na úrovni analytického modelování, tak na úrovni technologického designu. Jazyk UML je totiž nejenom unifikovaný jazyk, ale i jazyk univerzální, pomocí něhož lze modelovat jak úroveň abstrakce analytického modelování, tak úroveň abstrakce designu neboli technologického návrhu.

Class Diagram se používá pro modelování všech situací v návrhu informačního systému, které podléhají dichotomii třída-instance, tj. obecně jako vztah typ a výskyt z tohoto typu. *Class Diagram* vyjadřuje zavedené třídy a vztahy mezi nimi v daném prostředí. Na úrovni analytického modelování se jedná o analytické třídy, na úrovni technologického návrhu o dané typy z prostředí, což samozřejmě záleží na daném prostředí. Dané technologické třídy mohou být například tabulky se sloupci v relační databázi, objektové třídy z OOP, XML struktury, formuláře atd. To vše se v jazyce UML na úrovni technologického návrhu chápe jako třída s dodatkem, co vyjadřuje (pomocí tzv. stereotypů). Pro modelování těchto typů z prostředí se využije definovaných vlastností třídy z UML. Rozdíl *Class Diagramů* na různých úrovních abstrakce vyjadřuje věta v předešlé kapitole o úrovních abstrakce:

Sledujte aktuální novinky na http://www.objects.cz

Zatímco na úrovni abstrakce analytického modelování jsou třídy evidovanými pojmy v návrhu informačního systému, tak na úrovni modelování designu se již jedná o třídy (neboli typy) z konkrétního vývojového prostředí (například objektové třídy v C# nebo v Javě, nebo tabulky v MS SQL, v ORACLE atd.)

Jinak řečeno jazyk UML je použitelný jazyk jak pro modelování analýzy (fáze analytického modelování), tak tabulek v RDB, rovněž pro objektové třídy v OOP, stejně tak jako pro jiné další struktury (XML apod.), metadata apod. Rozdíl je pouze v tom, jaký význam mají třídy v daných modelech (tj. jaký typ vyjadřují). Přesně takto toho využívají CASE nástroje, např. Enterprise Architect (dále jen „EA“): Všimněme si například, že v EA je modelovací typ prvku tabulka zaveden jako „zvláštní třída“, která pouze „přejmenuje“ třídu na tabulku, pojem atributy na sloupce apod. K tomu účelu se využívá mechanismus „stereotype“, který je zaveden jako jeden z mechanismů extenze v jazyce UML.

Model analytických tříd je následně „namapován“ v technologickém postupu do tabulek nebo do objektových tříd, avšak ne vždy 1:1, například jedna analytická třída dá vzniknout dvěma třídám v OOP a v tabulkách dojde například naopak ke sloučení tříd neboli tabulek apod.

Jedná se tedy o transformaci jednoho modelu tříd do druhého modelu tříd, který není vždy v poměru tříd 1:1, tj. nemusí být počet tříd z jednoho modelu roven počtu tříd v druhém modelu. Znamená to, že jazyk UML svým mechanismem modelu tříd může popsat všechny modely tříd ve všech prostředích, včetně analytického (tj. implementačně nezávislého).

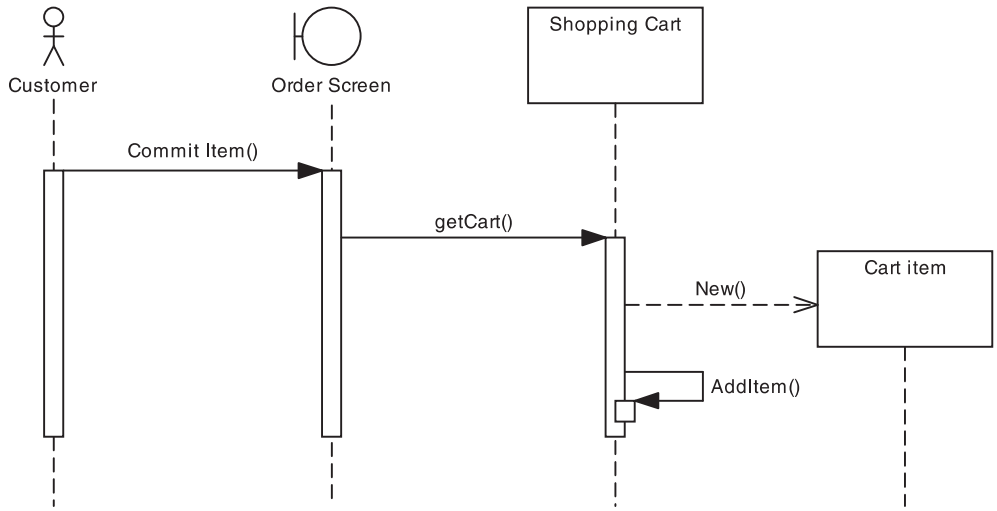
***Sequence Diagram* (D)**

Neboli *Sekvenční diagram*. Jedná se o diagram instanční, tj. nosným prvkem jsou instance, nikoliv třídy. *Sekvenční diagram* vyjadřuje detailně posloupnost zpráv mezi objekty (instancemi) v čase. Těmito objekty mohou být buď objekty z programování (JAVA, C# apod.) anebo vyšší implementační celky jako jsou komponenty, servery, subsystémy apod. Hlavním smyslem je popsat časový scénář posloupnosti zpráv mezi takovými objekty v technologiích. Tento diagram se nejčastěji používá v technologickém návrhu, v technologických systémech (tj. nikoliv čistě evidenčních, například hlásiče požáru, ovládání výtahů, strojů apod.). Další časté použití je ve vzorech objektového programování, tj. pokud vzor obsahuje dynamiku a je žádoucí ji popsat, použije se *Sekvenční diagram*. V analytickém modelování se jej nedoporučuje používat. Příklad *Sekvenčního diagramu* je uveden na obrázku 2.2. (Jednotlivé diagramy na obrázcích 2.2 a 2.4 jsou části diagramů z Helpu nástroje Enterprise Architect.)

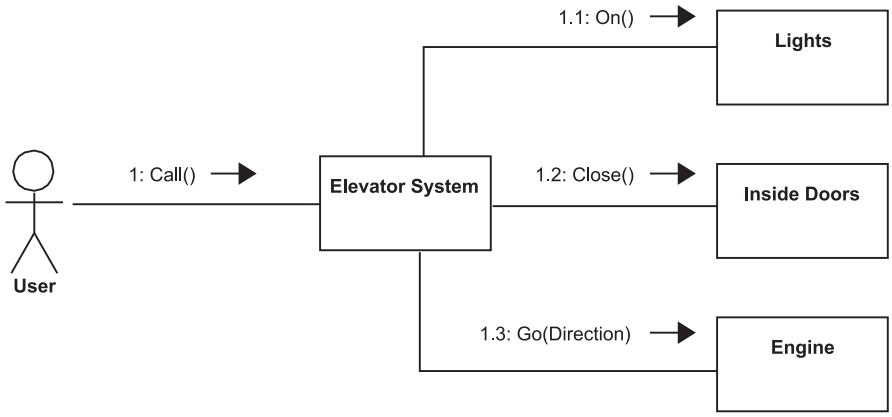
***Communication Diagram* (D)**

Tedy *Diagram komunikace*. Tento diagram je svou podstatou velice podobný předešlému diagramu, tj. vše, co bylo řečeno o *Sekvenčním diagramu*, platí také pro *Diagram komunikace*, až na to, že *Diagram komunikace* neukazuje zřetelně časovou posloupnost, ale zřetelněji zobrazuje strukturu objektů. Tyto dva diagramy jsou si tak podobné, že jednu a tu samou situaci popíšeme pouze jedním z nich, chceme-li raději zobrazit časovou posloupnost zpráv, použijeme *Sekvenční diagram*, pokud chceme spíše znázornit strukturu objektů, použijeme *Diagram komunikace*.

Sledujte aktuální novinky na http://www.objects.cz



Obrázek 2.2: Příklad *Sekvenčního diagramu*. (zdroj: [1])



Obrázek 2.3: Příklad *Diagramu komunikace*.

***Activity Diagram* (AM, D, *BPM*)**

Diagram aktivit neboli *činností*. Používá se tam, kde u sledovaného systému známe aktivity neboli činnosti systému a chceme je znázornit pomocí diagramu. Dá se využít:

- v analytickém modelování (AM) pro vyjádření posloupností činností při popisu algoritmů probíhajících uvnitř případů užití;
- pro tzv. modelování procesů podniku, tzv. *Business Process Modeling*. Předmětem modelování je v tomto případě okolí informačního systému. Smyslem tohoto modelování je nalézt systematicky případy užití;
- v designu pro znázornění chodu programu v technologii, dříve známý jako „vývojový diagram programu“.

Tento diagram budeme podrobně probírat v kapitole o nalézání případů užití.

***Object* (také *Instance*) *diagram* (AM, D)**

Jedná se o *Diagram objektových struktur*, tedy *instancí*. Instancemi mohou být buď

analytické výskyty informací nebo objekty z programování, resp. vyšší implementační celky. Matematicky vzato se jedná o limitní případ *Diagramu komunikace*, kdy se v tomto diagramu nezobrazí žádné zprávy a zůstane zobrazena jen struktura objektů. Přesto se uvádí ve výčtu diagramů UML jako zvláštní diagram. Používá se v raných fázích analýzy pro znázornění struktury informace jako příklad evidence a v technologickém návrhu pro zobrazení struktur vyšších implementačních celků.

***State Diagram* (AM, D)**

Neboli *Stavový diagram*. Používá se pro vyjádření stavů a přechodů prvků systému. Při zavedení *Stavového diagramu* musíme vždy uvést, k čemu se diagram vztahuje, tj. čeho je *Stavovým diagramem*. Těmito prvky, které mohou nést stavy a provádět přechody mezi nimi, mohou být buď analytické instance, resp. prvky mimo systém (například úvěr v bance apod.), objekty z OOP tříd anebo vyšší implementační celky (komponenty, servery, subsystémy apod.). *Stavový diagram* se velmi často používá při návrhu technologických systémů, které se chovají jako stavové stroje.

***Component Diagram* (D)**

Neboli *Komponentní diagram*. Jedná se o diagram zavedený pouze na úrovni technologického návrhu. UML zavádí pojem komponenty jako část implementace, která je ohraničená, opětovně použitelná a vyměnitelná. V programování se jedná o synonymum pro „modul“. Každé prostředí zavádí své konkrétní implementační celky a nějak je nazývá (např. library, unit, script, package, assembly, ActiveX DLL atd.). Cílem *Komponentního diagramu* je znázornit rozmístění kódu ve tvaru tříd z designu do těchto částí implementace a následně nalézt odpovídající vztah závislostí mezi komponentami, tj. vyjádřit, která komponenta si kterou musí přilinkovat (v kódu uses, using, import, include, apod.).

***Deployment Diagram* (D)**

Diagram nasazení, také *Diagram instalace* nebo také *Diagram implementace*. Zobrazuje systém z hlediska jeho instalování, tj. zobrazuje prvky typu hardware (stroje, servery, disky, tiskárny atd.), dále externí a podpůrný software (operační systém, frameworky, virtuální stroje apod.) a v neposlední řadě i artefakty neboli výsledky „vývojářské činnosti“ projektu na úrovni instalace (instalované soubory apod.) a zavádí i vztahy mezi nimi. Tento diagram je v podstatě nejvíce vzdálen úrovni abstrakce analytického modelování („fyzično nejfyzičnější“).

***Package Diagram* (AM, D)**

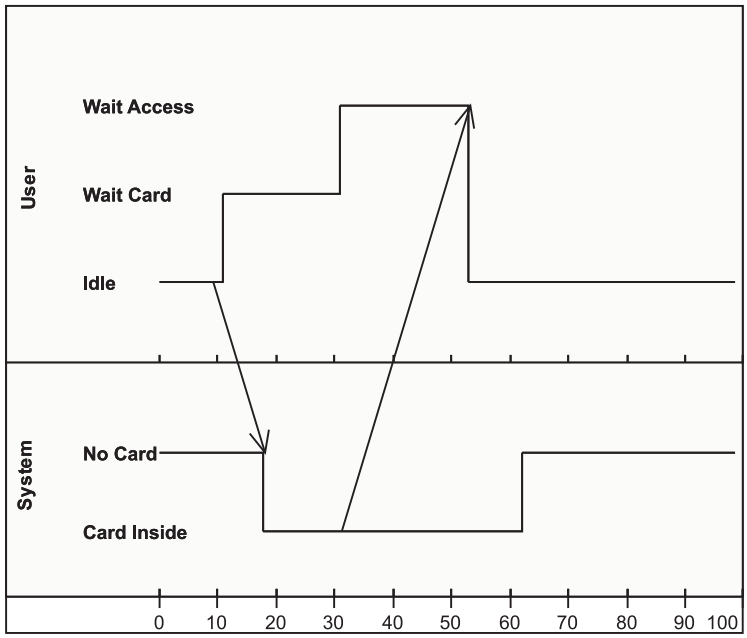
Diagram balíčků. Prvek typu *Package* slouží ke správě modelu v UML (tzv. Model Managment). Pomocí prvků typu *Package* se organizuje model (podobně jako je zavedena organizace souborů v adresářích na stroji), pomocí něj se provádí exporty, importy, distribuce modelu apod. Prvek typu *Package* je základním kamenem organizace modelu. Zvláštní postavení má prvek *Package* v *Class Diagramu*, kde se pomocí tohoto prvku určuje tzv. vrstvení aplikace ještě před návrhem *Komponentního diagramu*.

Poznámka: *Další diagramy byly dodány do verze UML 2.0, předešlé verze je neznaly.*

***Timing* (D)**

Diagram časování. Je určen výhradně pro technologické systémy, které pracují v reálném čase. Zobrazuje velmi přehledně změny stavů hned několika objektů v čase

souběžně pomocí lomených čar. Vodorovnost značí jeden stav objektu, změna vodorovnosti je změna stavu objektu. Příklad použití: Průběh změn stavů čtečky (jeden objekt) a do ní vsunuté karty (druhý objekt), viz obrázek 2.4.



Obrázek 2.4: Příklad *Diagramu časování*. (zdroj: [1])

***Composite Structure Diagram* (AM, D)**

Diagram kompozitních struktur. Je kombinací *Instančního diagramu* a *Diagramu tříd*, kdy jeden prvek obsahuje jiné prvky, což se znázorní vnořením skládajících prvků do třídy tohoto prvku. Jedná se o doplňkový diagram, který vzniká kombinací dvou diagramů, je tedy alternací již existujících zápisů. Nemusíte jej používat, stačí znát *Class Diagram* a *Object Diagram*, tento diagram je v podstatě jejich kombinací.

***Interaction Overview Diagram* (AM, D)**

Nejlepší překlad asi jako *Diagram přehledu interakcí* nebo *Diagram interakcí*. Jedná se o diagram, který je používán ve velmi specifické situaci. Diagramy v tomto výčtu

- *Sequence Diagram*
- *Activity Diagram*
- *Timing Diagram*
- *Communication Diagram*

lze, pokud chceme, vzájemně propojit tak, že můžeme vyjádřit jejich následnost. Například chceme vyjádřit, že na tuto sekvenci zpráv (vyjádřeno *Sekvenčním diagramem*) navazuje tento průběh aktivit (tj. *Diagram aktivit*) apod. K tomu můžeme použít zmíněný *Diagram přehledu interakcí*. V jednom z těchto diagramů vymezíme oblast neboli rámeček, může to být i celý diagram, v druhém diagramu vymezíme další oblast ve tvaru rámečku. Takovéto rámečky z různých diagramů můžeme použít v jednom diagramu typu *Interaction Overview Diagram*.

Pokud tedy chceme vyjádřit propojenost těchto čtyř diagramů (například „po této sekvenci následuje tato aktivita“), můžeme tak učinit pomocí diagramu *Interaction Overview*.

***Custom Diagram* (není součástí jazyka UML)**

Mimo oblast syntaxe jazyka UML existuje ještě jeden diagram, který je podporován mnoha CASE nástroji, ale paradoxně v UML není zaveden. Protože tento diagram není v UML, nemá název. Pracovně ho nazvěme jako *Custom Diagram* neboli *Uživatelský diagram*. Důvodem jeho použití je to, že existují určité myšlenky, které bychom chtěli vyjádřit v diagramu, ale tzv. striktní UML je neumožňuje zapsat, protože v diagramu určitého typu je povoleno umístit pouze prvky určitého typu.

Například bychom chtěli znázornit, že tento prvek *Use Case* používá ve svém scénáři prvky z těchto tříd. Bohužel, v striktním UML nelze dát do jednoho diagramu prvek typu *Use Case* a prvek typu *Class*. Pokud totiž založíme diagram typu *Use Case*, nepodaří se nám do něj umístit prvek typu *Class* a naopak. Vypneme proto v CASE nástroji striktnost a poté můžeme do diagramu typu *Use Case* vložit třídu anebo do diagramu typu *Class* vložit prvek typu *Use Case*. Vznikne tak sice podle UML „nestriktní diagram“, ale mnohdy je takový postup účelný, proto doporučuji používat takový nástroj, který umí *Custom Diagramy*.

2.6 Použití diagramů v projektu

V předcházející kapitole jsme si uvedli výčet všech diagramů, které UML nabízí. Z hlediska použití v projektu jsou však některé z nich více důležité a některé méně. To znamená, že u některých typů diagramů bychom měli požadovat maximální úplnost, tj. vyžaduje se, aby popisovaly celý systém, u jiných toto požadovat nemusíme (dokonce některé diagramy se nemusejí v projektu vyskytovat vůbec). Nejprve se věnujme požadavkům na diagramy u evidenčních (tedy podnikových) systémů. Poté se budeme zabývat systémy technologickými.

2.6.1 Nezastupitelnost diagramů a jejich postavení v projektech

Abychom pochopili důležitost diagramů, představme si následující ideální situaci:

> *Každá činnost v projektu bude zadána a zaznamenána pomocí diagramu UML. Tedy diagram UML s textovým doprovodem se stane v projektu zadáním a podle něj se provedou konkrétní práce v projektu a poté se diagram s textem stane i částí dokumentace projektu.*

Je třeba podotknout, že až takto ideálně modelování a následné práce nikdy neprobíhají. Jedná se o neustálé „kolečko“ ve smyslu vyhotovení diagramu s textovým doprovodem, následuje fyzická realizace podle tohoto zadání a poté se provede kontrola shody „navržený model versus realita“. Velmi často výsledek „nesedí“ s diagramem, přičemž mohou nastat při těchto neshodách následující dvě situace:

1. Zrealizovaný artefakt je špatně, tj. nedodrželo se zadání, v tom případě je třeba artefakt opravit.

Sledujte aktuální novinky na http://www.objects.cz

2. Zrealizovaný výsledek je dobře, ale zadání bylo vytvořeno špatně nebo se zadání změnilo a nestihlo se zaznamenat (vývoj předběhl změny v zadání pomocí UML). Je povinností autora diagramu provést jeho opravu a dát diagramy a texty do souladu s realitou.

Pro nalezení těch diagramů, které musí být úplné, použijeme postup, kdy jdeme zpětně ve vývoji systému, tedy stejně, jako když se pustí film pozpátku.

Na konci návrhu vývoje je systém nasazen, tedy je implementován, tj. je instalován fyzicky. Tyto práce měly proběhnout podle *Diagramu nasazení*, tedy prvním nezbytným diagramem je *Deployment Diagram*.

Představme si, že v kroku před tím programátor měl již zadán kód, tj. věděl přesně, jak budou vypadat třídy v OOP, atributy a metody, ale nevěděl, do kterého celku má tento kód napsat. Například nechceme, aby vše tzv. „napráskal“ do jednoho `.exe` programu. Rozmístění kódu mu zadává *Komponentní diagram*. Dalším nezbytným diagramem je tedy *Component Diagram*.

Ještě před tím se programátor musel dozvědět, jak má kód vypadat. Tuto informaci získá jako zadání v *Diagramu tříd*, který by měl být vyhotoven jak na úrovni analytického modelování, tak na úrovni technologického návrhu. Dalším nezbytným diagramem je tedy *Class Diagram*.

Když analytik vyhledával třídy a strukturu informace, musel se zajímat o to, k čemu systém slouží, jak se bude používat a co bude umět. Dalším nezbytným diagramem je tedy *Use Case Diagram*.

K tomu, aby analytik určil, co systém bude umět, musel se zajímat o kontext použití systému, tedy o to, co se děje v okolí, že se bude používat systém. Dalším nezbytným diagramem je tedy *Business Process Model*, čemuž v UML odpovídá *Activity Diagram*.

2.6.2 Shrnutí přehledu diagramů UML nezbytných pro vývoj IS

U následujících diagramů je třeba dbát na to, aby byly nejen správné, ale také úplné:

- *BPM* (jako zvláštní případ *Activity Diagramu*);
- *Use Case Diagram*;
- *Class Diagram* (AM + D);
- *Component Diagram*;
- *Deployment Diagram*.

Další diagramy se samozřejmě mohou vyskytovat v projektu také, ale nemusejí být úplné a nemusí pokrývat celý systém. Například *Communication Diagram* anebo *Sequence Diagram* se použije pro komunikaci mezi servery a jinde se již nepoužije.

U technologických systémů se do popředí dostává navíc *Stavový diagram* (*State Diagram*), který se používá v technologických systémech ve své úplnosti. Výhodou je totiž popis technologického systému jako stavového stroje. Navíc se u těchto systémů používají ostatní diagramy mnohem častěji, i když nemusí být také úplné.

Sledujte aktuální novinky na http://www.objects.cz

Kapitola 3

Class Diagram

Tento diagram je postaven na základním vzoru *Dichotomie třída-instance* (viz předešlé kapitoly), tj. *Class Diagram* zavádí třídy a vztahy mezi nimi. Používá se jak na úrovni analytického modelování, tak na úrovni technologického návrhu. Na úrovni analytického modelování se jedná o tzv. analytické třídy, které reprezentují evidované pojmy, na úrovni technologického návrhu se jedná o typy z daného prostředí (tabulky, OOP třídy, funkce a proměnné atd.).

Diagram tříd si budeme v této knize vysvětlovat nikoliv jako ve škole, tj. z čeho je složen, jaké má prvky atd., ale použijeme techniku vysvětlování za pomoci praxe: Budeme postupovat krok za krokem v posloupnosti tak, jak se tento diagram tvoří, jak je navrhován a jak se s ním pracuje. Spolu s těmito kroky budeme také objasňovat syntaxi tohoto diagramu. Do výkladu zahrneme nejčastěji používané vzory v analytickém modelování a vzory mapování do technologie OOP a RDB.

Při výkladu se budeme snažit co nejvíce používat vzory, neboli opakující se řešení, a ty budeme nějak nazývat. Na konci kapitoly *Class Diagram* provedeme souhrn syntaxe v kapitole ve stylu „get it together".

Samozřejmě budeme dodržovat i úrovně abstrakce, tj. nejprve provedeme návrh tříd v analytickém modelování a poté mapování do technologií (OOP a RDB).

3.1 První krok – vyhledávání konkrétních tříd

V prvním kroku analytik vyhledává tzv. konkrétní třídy. Konkrétní třídy jsou takové třídy, které dávají vznik instancím, tj. které mají svůj přímý obraz ve zrozených instancích. Z řečeného vyplývá, že existují také třídy, které nemají svůj obraz ve zrozených instancích, ty se nazývají abstraktní, vyhledávají se až nakonec, proto je budeme probírat až na konci výkladu.

Hlavním důvodem, proč analytik při tvorbě *Class Diagramu* začíná právě konkrétními třídami, které „rodí evidované instance", je vcelku logický: Pokud si totiž v duchu představujeme budoucí aplikaci jako souhrn evidovaných pojmů, tedy jako souhrn toho, „co se eviduje", tak vždy v prvním kroku vidíme instance jako příklady evidence a teprve z těchto příkladů evidence zobecňujeme vzorec do modelu tříd. Znamená to, že si vždy, ať chceme nebo nechceme, nejprve představujeme evidované instance jako možné situace konkrétní evidence. Takovýchto několik příkladů evidovaných situací si v duchu vybavíme včetně možných hodnot a teprve poté nalézáme třídy.

Nikdy v naší úvaze nenalezneme třídy jako první „ještě před představou o instancích", i když to tak někdy může zdánlivě vypadat. V takových případech se totiž jedná o velmi

jednoduché situace evidence, kdy nám představa instancí velmi rychle probleskne hlavou a hned vidíme třídy.

Důvod, proč při tvorbě modelu tříd vždy nejprve vidíme instance a teprve poté třídy, je jednoduchý: Evidované instance jsou konkrétními příklady evidence s konkrétními hodnotami, jak by evidence mohla vypadat, kdežto třídy jsou jejich meta-pravidlem, ze kterého tyto instance vznikají. Toto meta-pravidlo na úrovni tříd na první pohled nikdy nevidíme. Spatřujeme až jeho projevy, tj. při hledání meta-pravidla vždy nejprve nalezneme příklady, které z tohoto meta-pravidla vypadnou.

Dá se říci, že model tříd je zvně neviditelné meta-pravidlo pro budoucí instance a z toho důvodu, chceme-li takovýto zvně neviditelný vzorec nalézt, musíme si nejprve v duchu představit několik příkladů jeho aplikace a tyto příklady pak zobecníme do meta-pravidla neboli do modelu tříd.

Podobně platí i věta obráceného pohledu: Dostane-li čtenář do ruky již hotový model tříd ve tvaru *Class Diagramu* a neumí si v dané chvíli představit instance, je mu tento diagram na nic. Jinak řečeno, protože *Diagram tříd* se chová jako vzorec neboli pravidlo pro budoucí instance, musíme si umět velmi dobře představit aplikaci tohoto vzorce na instanční úrovni. Pokud to neumíme, je nám diagram k ničemu. Z již uvedeného o pohledu na instance a třídy v *Class Diagramu* vyplývá zajímavý a tak trochu paradoxní závěr:

> *Primárním pohledem na* Class Diagram *je (paradoxně) pohled na instance, které z něj vznikají, a to jak při jeho tvorbě, tak při jeho následném studiu!*

Pro postupy tvorby *Class Diagramu* proto musíme dodržovat toto velmi důležité pravidlo:

> *Každou evidenci nejdříve vidíme jako příklady na úrovni evidovaných instancí a teprve poté zobecňujeme tyto příklady do meta-pravidla neboli do modelu tříd. Podle této zásady si nejprve představujeme evidované instance (například i s hodnotami) a následně zobecňujeme příklady instancí do tříd.*

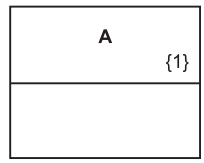
Je třeba zdůraznit, že není povinností malovat v projektu instance, pokud odevzdáváme model tříd. Je totiž zřejmé, že odevzdaný *Diagram instancí* musí zákonitě vyplývat z odevzdaného nalezeného *Diagramu tříd*. U velmi složitých modelů je však možné jako dodatek připojit i příklad evidence na úrovni instancí pro dobrou představu příkladů evidence.

Jak již bylo řečeno v předešlých kapitolách, třída se značí obdélníkem s názvem nepodtrženým, instance se značí obdélníkem s názvem instance podtrženým. Jazyk UML sice umožňuje přiřadit k instanci třídu, ze které pochází, ale protože vycházíme z instancí při neznalosti tříd (ty hledáme poté), nebudeme toto přiřazení provádět ihned.

3.1.1 Multiinstanční třídy a jednoinstanční třídy, vzor *Singleton*

U tříd uvádí analytik tzv. multiplicitu, tj. hodnotu, kolik instancí z dané třídy je dovoleno zrodit, přičemž valná většina konkrétních tříd jsou tzv. multiinstanční. Znamená to, že z této třídy může pocházet „dopředu neznámý počet instancí“. Pokud není třída multiinstanční, je většinou jednoinstanční. Taková třída bývá v OOP mapována do technologie a programování podle vzoru *Singleton*. Výjimečně bývá třída zavedena s přesným počtem několika instancí, to jsou většinou číselníky (neboli code list) s pevným počtem prvků.

Protože je valná většina tříd multiinstančních, je zvykem a nepsanou dohodou, že pokud neoznačíme u třídy multiplicitu, je automaticky chápána jako multiinstanční. V případě, že není multiinstanční, měli bychom tuto multiplicitu zadat. Příklad jednoinstanční třídy zapsané v EA je na obrázku 3.1.



Obrázek 3.1: Jednoinstanční třída v EA.

Poznámka: *Z uvedeného vyplývá, že třídám dáváme názvy v jednotném čísle, protože je to název „meta vzorce“ neboli třídy, ale uvádíme k ní multiplicitu.*

Příklad Flexibilní test neboli vzor *Problém META*
Již při nalézání konkrétních tříd může nastat problém, který se nazývá *Problém META*. Vysvětlíme si ho na ukázkovém příkladu:

Jedna česká firma dodává na trh aplikaci, kterou bychom mohli nazvat „Flexibilní test“. Aplikace je napsaná pomocí PHP, JAVA a MySQL. Po zakoupení se aplikace nainstaluje ve firmě na její vnitřní síť, tj. na intranet. Když si tuto aplikaci firma koupí a spustí, tak v té chvíli uživatel nenalezne v aplikaci ani jeden test. Firma však může pověřit své pracovníky, aby v aplikaci zakládali nové testy s otázkami a odpověďmi. Tuto aplikaci si může koupit například Vysoká škola ekonomická (VŠE) a založit si např. testy k přijímacím zkouškám. Podobně si aplikaci může zakoupit třeba Armáda České Republiky (AČR), připraví si testy pro příjem vojáků z povolání, podobně firma XY si připraví testy pro příjem nových zaměstnanců.

Nejprve jako první příklad uveďme chybu spojenou s problémem META, kterou nikdo určitě neudělá. Zeptejme se:

Bude se v systému vyskytovat třída VŠE? Nebo třída AČR? Odpověď: Určitě ne.

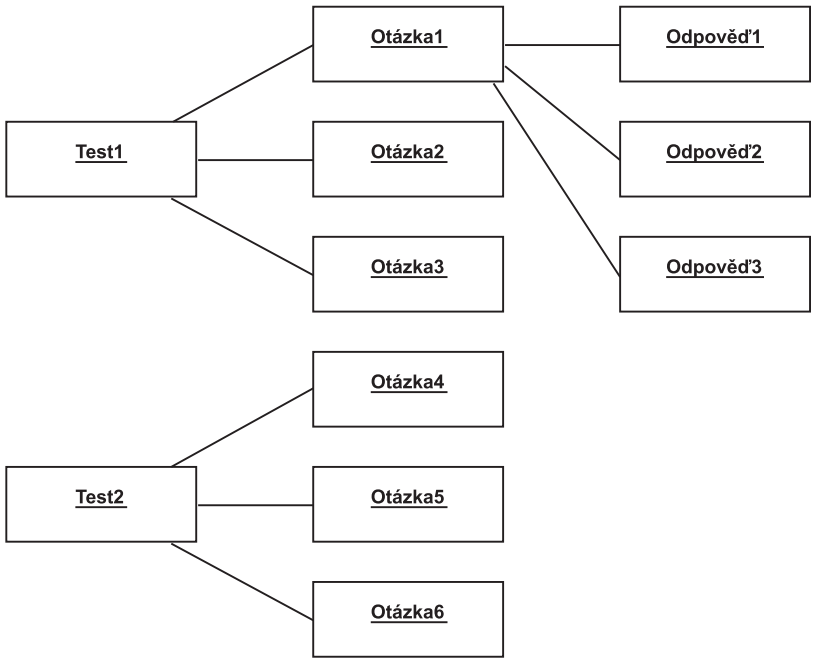
Další otázka zní: Jaké třídy můžeme tedy očekávat v aplikaci „Flexibilní test“? Zkusme jich několik nalézt a hned také současně uveďme jejich multiplicitu:

třída Test
Ta bude v budoucnu reprezentovat založené nové testy. Navrhneme ji jako třídu multiinstanční (tj. instancí z této třídy Test bude více). Každá instance z této třídy bude mimo jiné obsahovat například datum založení testu, poté údaj, kdo ho založil apod.

třída Otázka
Také bude multiinstanční (otázek bude určitě více). Prvky z třídy Otázka obsahují text otázky.

třída Odpověď
Také multiinstanční třída. Pro jednoduchost, protože nemůžeme nyní zadávat složité příklady, předpokládejme jednoduchou konstrukci testu, kdy máme otázky typu výběr z odpovědí, tj. jedna otázka má několik odpovědí, z toho je jedna nebo více správných. Každá odpověď bude tedy obsahovat příznak „jsem správná“ = ano/ne (typ boolean). Naše představa instancí v evidenci může být nyní jako na obrázku 3.2.

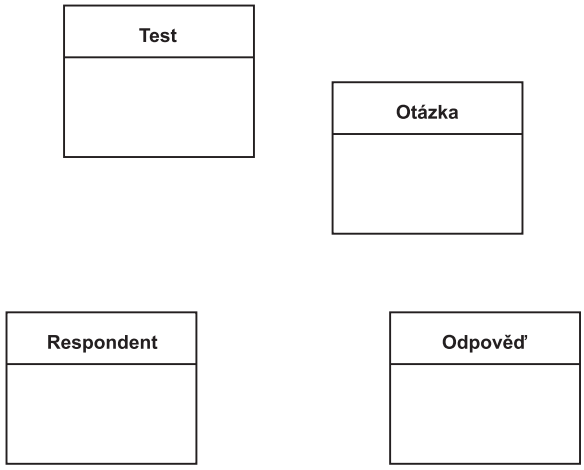
Obrázek 3.2 zobrazuje náčrt představy části evidence testů. V diagramu není zobrazeno, že každá otázka má svůj text (s nějakou hodnotou) a že odpověď má také text a navíc příznak správná = ano/ne. Budeme také evidovat ty, kteří odpovídají na testy, tj. zavedeme třídu Respondent.



Obrázek 3.2: Několik instancí v aplikaci Flexibilní test.

třída Respondent
Tato třída bude také multiinstanční.

Pokusme se předešlé věty ztvárnit do *Diagramu tříd*, například v prvním přiblížení bez vztahů na obrázku 3.3.



Obrázek 3.3: Návrh tříd v aplikaci Flexibilní test.

Důležitá poznámka: *Všimněme si, že bychom již nyní potřebovali určit vztahy mezi třídami, které jsou vidět v* Instančním diagramu, *a totiž, že otázka spadá do testu a má své odpovědi atd. To bude předmětem dalších kapitol.*

Příklad je sice asi jasný, ale stále v něm není vidět uvedený vzor nazvaný *Problém META*. K němu dojdeme následující úvahou: Představme si, že jsme v úplně jiné firmě, než v této uvedené, která vyvíjí aplikaci flexibilní test. O aplikaci flexibilní test jsme nikdy neuvažovali. Přijde k nám nějaká firma XY, s. r. o. jako zákazník a řekne: Máme tady tento konkrétní test s těmito konkrétními otázkami a konkrétními možnými odpověďmi, a podá nám disk s „Word“ souborem. Nabízí nám přitom vysokou odměnu, pokud jim tento jediný test s těmito otázkami a odpověďmi umístíme na jejich intranet v HTML podobě do tří dnů.

Nyní má „naše softwarová firma“ na výběr ze dvou možností: Buď vezme uvedený „Word“ soubor, převezme otázky a odpovědi, a přímo v PHP (anebo jiném prostředí) doslova tyto otázky a odpovědi „napráská natvrdo“ v kódu do tohoto prostředí. Vznikne tak aplikace, kde přímo v kódu budou texty otázek a texty odpovědí.

Mohla by se však ukázat i druhá varianta řešení, která nás může napadnout právě díky znalosti příkladu z předešlé firmy: Co kdybychom tuto aplikaci „nenapráskali natvrdo“, ale vyvinuli flexibilní variantu, poté vyplnili jeden dotazník, tedy ten, který si přejí, a tuto aplikaci s jedním vyplněným testem jim pak odevzdali?

V čem je výhoda druhého postupu? Kdyby zákazník přišel znovu, můžeme mu velmi rychle dodat další dotazník pouhou konfigurací (tj. vyplněním hodnot instancí a nikoliv napsaným programem). Vidíme, že problém META se tu projevil v této podobě: Máme na výběr navrhnout aplikaci buď tak, jak je původně zamýšlena a zadána, anebo namísto toho vytvořit šablonu pro libovolný dotazník, tedy provést posun do meta-úrovně a řešit „flexibilní aplikaci“ o jednu úroveň meta výše. Ve flexibilní variantě to, co bylo v předešlé variantě na úrovni tříd, je nyní posunuto do instancí vyšších tříd a to je posun do META.

Která z těchto variant má právo na život? Odpověď není až tak jednoduchá, protože se jedná nejen o analytický, ale také o obchodní problém. Přímo v zadání pro trh je totiž přesně dáno, co se vlastně žádá vyvinout.

Tento příklad byl schválně zvolen tak, aby odpověď na otázku „napráskat natvrdo“ anebo „navrhnout a napsat aplikaci flexibilní dotazník“ byla jasná ihned na první pohled: Pro firmu se zadáním „jeden dotazník za tři dny s vysokou odměnou“ bychom určitě vyvinuli aplikaci „natvrdo“, abychom neriskovali, že zakázku nestihneme. Všimněme si, že oproti tomu v první firmě bylo pojetí aplikace jako flexibilní test přímo a přesně dáno obchodním zadáním s tím, že se taková aplikace bude na trh dodávat a bude sloužit právě k tomu, aby si zákazník sám mohl vyhotovit nový test.

Uvedený posun META se promítne samozřejmě až do úrovně abstrakce kódování. Představme si, že nějaký prvek GUI, například *Label*, by získal svoji hodnotu pro zobrazení uživateli například v pseudokódu podobném C# takto:

```
MyLabel.Text = "Vysoká škola ekonomická";
```

a přitom se jedná o aplikaci Flexibilní dotazník. Zarazí nás to? Určitě. V této aplikaci by se dalo spíše očekávat, že tato hodnota se získá z nějaké proměnné (měnitelná hodnota v instanci), například takto:

```
MyLabel.Text = gMajitelAplikace.GetNazev();
```

Naopak, pokud bychom vytvářeli aplikaci přímo „natvrdo“, tak by nás konstrukce kódu s uvedením řetězce nepřekvapila.

Poznámka: *Samozřejmě předešlý příklad kódu v sobě skrývá jeden nedostatek. Uvedená řetězcová konstanta by měla být deklarována bokem, aby se dala případně změnit zásahem do kódu na jednom místě. V každém případě se však jedná o zásah do kódu a nikoliv o změnu hodnoty v proměnné.*

Sledujte aktuální novinky na http://www.objects.cz

Problém posunu meta je asi zřejmý, navíc však osobně varuji před velmi nebezpečnou situací, nazval bych ji „velké oči s touhou po dokonalosti“. Jedná se o případ, kdy se žádá aplikace na míru (tj. „natvrdo“) a z touhy po dokonalosti se zvolí posun do META úrovně dokonce se zdůvodněním, že to bude jednodušší. Aplikace posunutá do META v žádném případě není jednodušší, naopak, je pro nalezení řešení o mnoho složitější, nejen o řád, ale dokonce o řád řádů! Při snaze vyvinout flexibilní aplikaci na META úrovni reálně hrozí, že nebudeme mít dost zkušeností a dostatek konzultantů pro obsažení všech variant, které mohou ve flexibilním řešení nastat.

Mám osobně velmi špatné zkušenosti se snahou posunout aplikaci do meta-úrovně, protože na první pohled jednoduché problémy se nakonec v meta úrovni „rozmnožily“ do nepřekonatelných problémů. V konečném důsledku, pokud se něco podařilo dodat, vznikla jakási podivná aplikace, která se konfigurovala šíleným způsobem a doprogramovávala se ve specifických situacích velmi pracně v silně nepřehledném prostředí.

3.2 Vztahy mezi třídami

V předešlé kapitole jsme si vysvětlili, že prvním krokem při návrhu *Class Diagramu* v analytickém modelování je nalézání tzv. konkrétních tříd, tj. tříd, které nejsou abstraktní a které „rodí“ analytické instance. Důvod tohoto doporučení spočívá v tom, že každou aplikaci nejprve vidíme jako příklad evidence v instancích a teprve poté ji zobecňujeme do „meta pravidla“, tj. „meta vzorce“ neboli do modelu tříd platného pro všechny příklady instancí (i dosud neznázorněných). V předešlé kapitole jsme také upozornili na možný *Problém META* při návrhu konkrétních tříd.

Dalším krokem je nalézání interakcí neboli vztahů mezi třídami v modelu tříd. Všimněme si, že v předešlém příkladu s „Flexibilním testem“ jsme již takové vztahy potřebovali zavést. Například zřetelně vidíme na úrovni instancí, že konkrétní instance testu obsahuje konkrétní instance otázek, konkrétní otázka má možné konkrétní odpovědi pro výběr atd.

Jazyk UML zavádí v modelu tříd dva vztahy:

1. Association *neboli česky* Asociace*;*
2. Generalization *neboli česky* Generalizace *(slangově zvaná také jako* dědičnost*).*

Mezi těmito dvěma vztahy je principiální rozdíl, mají jinou povahu, působí v jiné oblasti, chovají se jinak a v modelu implikují rozdílné působení na instance rodící se ze tříd.
Poznámka: *Jednou z velkých a častých chyb návrhu IS je záměna těchto dvou vztahů, bude pojednáno v kapitole Vzor Bridging neboli Přemostění na straně 85.*

Při vyhledávání vztahů mezi třídami v *Class Diagramu* zásadně začínáme vyhledáváním vztahů *Asociace*. Tento vztah je totiž jednodušší, je velmi dobře viditelný a navíc vztah *Generalizace* nalezneme až poté, když zřetelně vidíme vztahy *Asociace*. Důležité pravidlo zní: Nehledejme nejprve vztahy *Generalizace*, ale *Asociace*. Z tohoto důvodu budeme vztah *Asociace* brát jako první a také si ho vysvětlíme dříve než vztah *Generalizace*, který probereme až nakonec. Nejprve definujme, jak je interakce asociace v jazyce UML obecně zavedena:

> *Vztah* Asociace *v* Class Diagramu *je vztahem znázorněným na úrovni tříd, ale vyjadřuje budoucí vztah mezi instancemi (tj. vztah mezi budoucími instancemi, až se narodí). Vztah mezi instancemi vzniklý ze vztahu* Association *se v jazyce UML nazývá* Link *a je chápán jako instance vztahu* Association.

Sledujte aktuální novinky na http://www.objects.cz

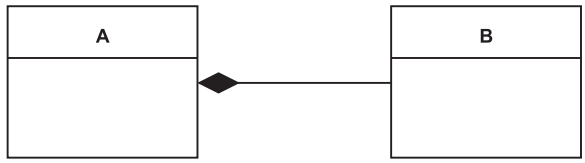
To znamená, že zatímco instanciováním tříd vznikají instance ze tříd (tj. výskyty ze třídy, viz předešlá kapitola), instanciováním vztahů *Association* vznikají vztahy *Link* mezi instancemi ze tříd. Tyto vztahy mezi instancemi jsou velmi dobře patrné, dobře viditelné a proto se vztah *Asociace* nalézá lépe a snadněji než *Generalizace*. Platí také obrácené pravidlo, které mimochodem přesně vysvětluje, jak vlastně funguje asociace v modelu tříd: Pokud najdete vztah mezi instancemi v modelu instancí, buďte si na sto procent jisti, že tento vztah mezi instancemi vznikl z asociace v modelu tříd.

Protože vždy navrhujeme podobu *Class Diagramu* postupem zobecňování od instancí ke třídám, tak nejprve vidíme *Linky* mezi instancemi a následně tyto *Linky* zobecňujeme do vztahů *Asociace* v modelu tříd.

V jazyce UML sice existuje pouze jeden vztah *Asociace*, ale liší se z hlediska analytického významu. Analytické rozdíly v různém použití asociace se v jazyce UML vyjadřují různým označením konců asociace. Podle toho se pak asociace chápe jako vztah s jiným analytickým významem (ale stále se jedná o jeden typ vztahu *Asociace*). Probereme nyní všechna tato použití vztahů *Asociace* v analytickém modelování pomocí jazyka UML postupně tak, jak je budete vyhledávat.

3.2.1 Vztah *Kompozice*

Vztah *Kompozice* je asociací, která se znázorní spojnicí a jeden z konců asociace se označí černým kosočtvercem, viz obrázek 3.4.



Obrázek 3.4: Vztah *Kompozice*.

Poznámka: *V EA můžete tento vztah vytvořit dvěma možnými postupy, buď zvolíte přímo v Toolboxu vztah* Kompozice *anebo zavedete asociaci a poté jeden z konců asociace zvolíte jako kompozit.*

Protože vztah *Kompozice* je asociací, jedná se o budoucí vztah mezi instancemi ze tříd A a B. Jinak řečeno, až se narodí instance ze třídy A, bude ve vztahu linku pocházejícím z kompozice vůči instanci (resp. instancím) ze třídy B. Vztah *Kompozice* označuje velmi zřetelný vztah „celek/část“ v „silném majitelství“, kde instance ze třídy A (tj. instance ze třídy označené na konci asociace kosočtvercem) jsou chápány jako celek (tj. majitel).

Kompozice má následující analytický význam: Instance ze třídy B jako „část“ v analytickém vztahu „celek/část“ mohou vznikat, zanikat a žít pouze v kontextu (tj. majitelství) instance ze třídy A a nijak jinak. Znamená to, že když se rodí instance ze třídy B, tak je dán prostor jejího zrodu, života a smrti pouze a jenom v kontextu instance ze třídy A, tj. tam také bude instance ze třídy B žít.

Díky „silné vlastnosti vlastnictví součástky“ jsou v kompozici zakázány následující scénáře:

1. *Instance ze třídy B se nesmí narodit bez svého majitele (mimo jeho kontext) ještě před zrodem instance ze třídy A anebo instance ze třídy A (majitel)*

nesmí být vymazána ze systému s tím, že současně podřízená instance ze třídy B v systému zůstala.

2. *Instance ze třídy B nesmí během svého života přeputovat od jednoho majitele (jedné instance) k jinému majiteli (druhé instance).*
3. *Instance ze třídy B nesmí být v typu vztahu* Kompozice *vlastněna dvěma různými instancemi.*

Instance ze třídy B se analyticky chápou jako nedílné součástky instance ze třídy A. Navíc informace obsažené v instancích B nemají samy o sobě samostatně použitelný kontext, tj. pokud je řeč o instanci z B, její význam je pouze v kontextu daného majitele instance ze třídy A. Logickým důsledkem těchto vlastností je to, že instance ze třídy B nemůže být sdílena ve dvojím vztahu linku kompozice vůči dvěma instancím majitelů. Jinak řečeno, jedna instance ze třídy A si rodí svoji, resp. svoje instance ze třídy B a druhá instance ze třídy A si rodí také svoji jinou, resp. svoje instance ze třídy B. V kompozici tedy laicky řečeno platí „svůj k svému“, tj. každý majitel si rodí své vlastní součástky bez jejich sdílení.

Na konci asociace u třídy B je třeba v analytickém modelování uvádět ještě 3 vlastnosti konce asociace, z nichž nyní si vysvětlíme dvě a třetí vlastnost si objasníme později.

Multiplicita na konci asociace

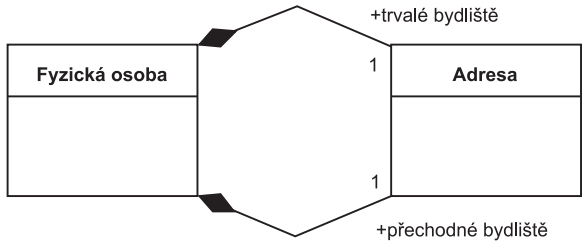
Na konci kompozice u třídy B se uvádí tzv. multiplicita, což je povolený počet instancí, které mohou vznikat, žít a zanikat na straně „část“. V jazyce UML se uvádí počet instancí buď celým číslem (tj. právě tolik instancí, kolik uvádí číslo) anebo rozsahem, kde se používají dvě tečky jako oddělovač spodní/horní hranice. Pro multiplicitu N, tj. otevřený seznam s libovolným počtem instancí, se používá znak * (hvězdička). Většinou možné hodnoty multiplicity zavádějí CASE nástroje (například EA) přímo v nabídce editace konce asociace.

Existuje i druhý, nikoliv podle UML čistý a striktní výklad, ale je prakticky často používán: Mnohdy se uvádí pouze jedna hodnota multiplicity jako horní hranice a dolní se vynechává. Musíme proto přesně vědět, který z těchto dvou způsobů autor používá, zda striktní a čistý zápis anebo ten, kdy autor uvádí pouze horní hranici. Uvedený „nečistý zápis“ s vynechanou spodní hranicí se mnohdy používá proto, aby se předešlo kolizi při chápání spodní hranice laikem nebo zákazníkem. Mnohdy je totiž spodní hranice pro laika překvapivá. Tento jev souvisí s častou chybou záměny pojmu z analytického modelování a pojmu z modelování podniku. Zatímco analytický model popisuje IS (je modelem softwaru), předmětem modelu podniku je oproti tomu okolí, tedy podnik. Pojmy zavedené jak v modelování IS tak v modelu podniku však znějí stejně, ale mají jiný význam: Například pojem faktura existuje jak v modelu podniku (to je skutečná faktura mimo systém), tak na druhé straně existuje faktura jako evidovaný pojem v analytickém modelování, což je budoucí kus programu (naprogramované objektové třídy a tabulky). Tyto dva pojmy faktura (mimo systém a uvnitř systému) jsou sice ve vztahu, ale chovají se jinak a je třeba je přesně od sebe odlišovat. Jedním z typických rozdílů je určení spodní hranice multiplicity ve vztazích *Asociace*. Jako příklad uveďme větu: „Faktura došlá do podniku musí mít alespoň jeden řádek“. Tato věta nebude „otrocky přenesena“ do návrhu systému ve smyslu multiplicity 1..*, protože by to znamenalo, že při založení faktury v systému se musí ihned založit první řádek. Konstrukce v systému je však jiná: Evidovaná faktura se po založení bude nacházet ve stavu otevřená k editaci (není ještě připravena k odsouhlasení) a v té chvíli může mít 0 řádků. Protože spodní a horní hranice multiplicity se určují přes všechny stavy evidovaných entit, je tak zřejmé, že správně má být u evidované faktury spodní hranice počtu řádků 0. Uvedenou

podmínku, zda má evidovaná faktura alespoň jeden evidovaný řádek se bude kontrolovat v případu užití „Uzavření editace faktury“. Všimněme si, že dolní hranice multiplicity je u pojmu „evidovaná faktura“ jiná, než u „faktura v podniku“.

Role u konce asociace

Druhou vlastností, kterou bychom měli u konce kompozice na straně část uvést, je tzv. role. Z hlediska teorie jazyka UML se jedná o určení, jakou roli hraje daná třída v tomto vztahu *Asociace* vůči druhé třídě. To sice zní složitě, ale prakticky se za roli volí název instance ze třídy na straně část, tedy tak, jak ji vidí majitel – celek. Dobře zvolený název instance totiž reprezentuje roli třídy. Typický příklad role znázorňuje příklad na obrázku 3.5.



Obrázek 3.5: Použití role v kompozici.

Na obrázku 3.5 si všimněme, že třída Adresa je použita ve dvou kompozicích, tedy dvakrát, jednou v roli trvalé bydliště a podruhé v roli přechodné bydliště. Jinak řečeno, v majitelství (kontextu) instance Fyzická osoba se vyskytují dvě instance ze třídy Adresa, jedna se nazývá trvalé bydliště a druhá přechodné bydliště.
Poznámka: *Tento model zavádí dvě instance adresy a nikoliv seznam adres, toto řešení bude ještě předmětem dalšího výkladu.*

Třetí vlastností konce asociace je tak zvaná „Směrovost“ a tu si vysvětlíme v jiné další kapitole.

Při návrhu IS se vztah *Kompozice* objevuje ve dvou základních vzorech, jeden objasňuje vztah *Kompozice ku jedné* a druhý *Kompozice ku N*:

1. Vzor *Osoba má adresu*;
2. Vzor *Faktura má řádky faktury.*

Nyní si oba vzory vysvětlíme.

Vzor *Osoba má adresu* (jako v e-shopu)

Tento vzor vyjadřuje vztah *Asociace* „kompozit ku jedné“ jako již uvedený příklad na obrázku 3.5. Takovéto řešení adresy pomocí kompozitu by bylo vhodné použít například v aplikaci obdoby e-shopu, tj. tam, kde se adresy nevyžadují jako jednoznačné, ale adres je přitom několik druhů (fakturační, dodací atd.). Existuje jiné řešení adres pomocí seznamu adres, jak bude ukázáno v dalších kapitolách.

Pro pochopení vztahu mezi třídami musíme mít velmi dobrou představu o tom, v jakém vztahu budou instance vzniklé z těchto tříd. V kompozici platí, že každý majitel jako „celek“ vlastní své instance na konci kompozice jako „část“. Protože každá instance-majitel má své vlastní instance-části (tj. rodí si svoje instance-části), pak i kdyby dvě osoby bydlely na

stejné adrese, tak by se v daném řešení tyto dvě adresy zadaly dvakrát pro každou osobu zvlášť.

Situace vlastněných instancí v kompozici je podobná, jako když vám někdo omylem pošle dvakrát stejný mail s přílohou. V tom případě dostanete dva e-maily a každý z nich má svou vlastní přílohu, teprve porovnáním obsahu zjistíte, že tyto e-maily i přílohy jsou shodné. V kompozici platí pravidlo obdobné rčení „já mám svoje, ty máš svoje“.

Připomenu ještě jinou možnost zadání multiplicity, než jako je na obrázku 3.5, může být také 0 až 1.

Mapování kompozice ku jedné do relační databáze

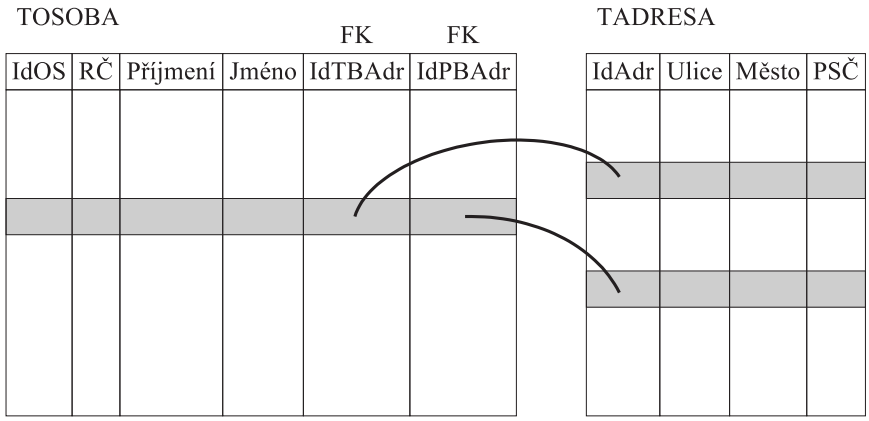
Jak bylo vysvětleno v předešlých kapitolách, po předání analytického modelu do technologie se provede realizace technologického návrhu, nejlépe pomocí vzorů mapování. Nyní si ukážeme dva známé základní vzory mapování kompozice ku jedné do relační databáze.

Poznámka: *Pro vazby mezi tabulkami v relační databázi budeme používat zásadně primární klíč ve tvaru tzv. Id, slangově ídečko, tedy autoinkrement (tj. číslování řádků automatem), jako jsou například sloupce typu IDENTITY, SERIAL, SEQUENCE atd. v různých databázích. U názvů tabulek budeme dávat předponu před název velké T, čímž odlišíme analytickou třídu od tabulky.*

Mapování kompozice ku jedné do RDB jako „1:1“, tedy tzv. čisté mapování

Můžeme si představit situaci, že analytik odevzdal do technologie analytický návrh v podobě modelu tříd jako kompozice ku jedné, například ve tvaru podle obrázku 3.5. Technolog se tento analytický model rozhodne namapovat do relační databáze s použitím vzoru 1:1, jde tedy o tzv. čisté mapování.

V tom případě se zavede pro každou analytickou třídu po jedné tabulce (tj. vztah mezi analytickou třídou a tabulkou je 1:1). Podle tohoto mapování vzniknou z analytického modelu mapováním dvě tabulky TOSOBA a TADRESA. Atributy zavedené v obou třídách, například rodné číslo, příjmení a jméno, se zavedou jako sloupce v těchto tabulkách. Nyní je třeba ještě zavést relaci mezi tabulkami, která bude realizovat vazbu kompozitu ku jedné. Za tím účelem se zavede pro každý vztah *Kompozice* do tabulky TOSOBA cizí klíč s vazbou na primární klíč z tabulky TADRESA, cizí klíč bude mít název odvozený z „role + název třídy“. Vznikne tak konstrukce vztahu mezi řádky znázorněná na obrázku 3.6.



Obrázek 3.6: Mapování kompozice ku jedné do RDB 1:1.

Je vidět, že všechny adresy se tak vyskytují v jedné tabulce a všechny funkcionality nad nimi mohou být opětovně použity.

Mapování kompozice ku jedné do RDB rozpuštěním kompozitu
Další způsob mapování do relační databáze odpovídá takzvanému postupu optimalizace.

Jak bylo řečeno, cílem optimalizace je „obchod“ v technologickém návrhu – v návrhu designu získáme nějakou technologickou výhodu (například rychlost), ale ztrácíme výsadu opětovné použitelnosti. Každá optimalizace by proto měla být podložena testy o její nutnosti. Obecně se správně v literatuře (a praxe tomu nasvědčuje) varuje před předčasnou a zbytečnou optimalizací, protože ztráta opětovné použitelnosti nejenom zbytečně „zesložití“ systém, ale hlavně vede ke snížení transparence systému.

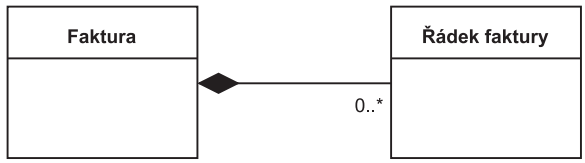
Mapování rozpuštěním kompozitu ku jedné je klasickým příkladem optimalizačního postupu. Představme si situaci, že technolog zjistí, že vazba mezi tabulkami přes cizí klíč při mapování 1:1 (předešlý případ) v některém z procesů zpracování zpomaluje systém takovým způsobem, že nelze toto zpomalení akceptovat a přitom všechny jiné technologické finty (indexace apod.) selhaly. Technolog se proto rozhodne pro optimalizaci mapováním rozpuštěním kompozitu do majitele. Tabulka TADRESA se tedy neobjeví, struktura třídy Adresa se namapuje přímo do tabulky majitele – namísto ní se tedy objeví nové atributy v tabulce TOSOBA, které budou mít název role plus název atributu (například „ulice trvalého bydliště“, „město trvalého bydliště“ atd.). Adresa jako entita se v RDB „rozpustí“ do svého majitele, tj. v tabulce TOSOBA se objeví v tomto případě dalších 2×3 sloupců, viz obrázek 3.7.

IdOS	RČ	Příjmení	Jméno	UliceTB	MěstoTB	PSČTB	UlicePB	MěstoPB	PSČPB

Obrázek 3.7: Mapování kompozice ku jedné do RDB rozpuštěním kompozitu.

Je zřejmé, že oproti předešlému mapování se sice získala rychlost, protože načtením jednoho záznamu dostaneme také i adresy, ale ztratili jsme opětovnou použitelnost. Každý sloupec již nyní bojuje sám za sebe, protože adresa v tabulkách neexistuje. Současně dochází také ke snížení transparence systému, protože pokud chceme opravit nebo změnit adresy, musíme dobře znát, kam jsme tyto adresy rozpustili. K tomu nám slouží nezastupitelný analytický model, a pokud jej nemáme, tak vzniká vážný problém s vyhledáváním rozpuštěné entity.

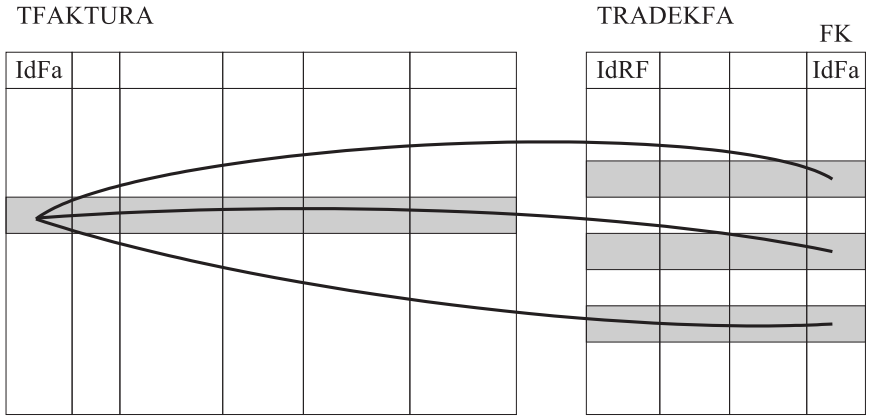
Vzor *Faktura má řádky faktury*
Vzor vyjadřuje kompozici ku N, prvek majitel obsahuje jako své součástky seznam prvků ze stejné třídy, viz obrázek 3.8.



Obrázek 3.8: Vzor *Faktura má řádky faktury.*

Prvek faktura v sobě obsahuje „dynamický“ seznam s dopředu neznámým počtem prvků řádků faktury.

Mapování kompozice ku N do RDB jako „1:1“, tedy tzv. čisté mapování
V tomto případě se zavedou dvě tabulky, jedna pro fakturu (např. TFAKTURA) a druhá pro její řádky (např. TRADEKFA) a objeví se cizí klíč na straně tabulky TRADEKFA, tj. každý řádek, který má v tomto sloupci tuto hodnotu, patří k danému řádku faktury s daným `IdFa`, viz obrázek 3.9.

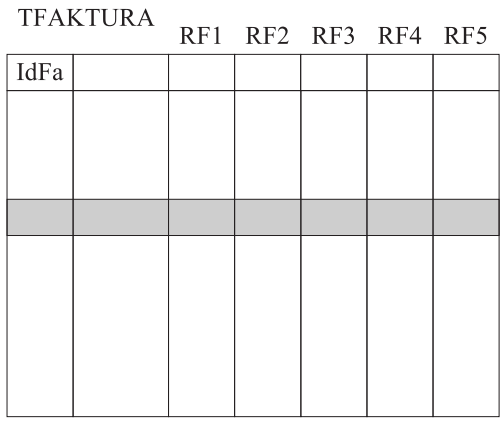


Obrázek 3.9: Klasické mapování kompozice ku N.

Mapování kompozice ku N do RDB rozpuštěním kompozitu
Jedná se (naštěstí) o velmi málo používané mapování. V tomto případě se nejprve omezí počet řádků faktury na určitý maximální počet (například 5) a provede rozpuštění kompozitu ku N do majitele opakováním struktury řádků v majiteli, viz obrázek 3.10.

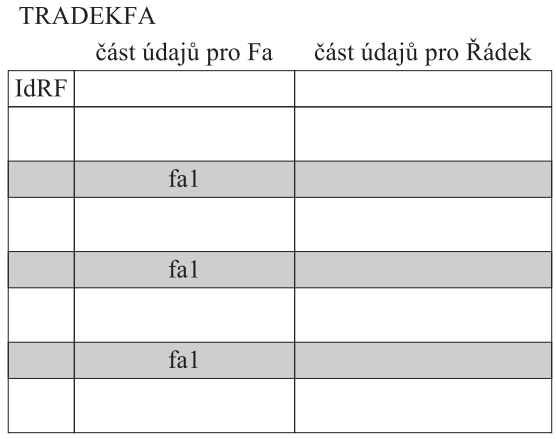
Řádky faktury se tak opakují vedle sebe přímo ve struktuře faktury. Toto mapování má nevýhodu nejenom v tom, že přidání nového řádku nad 5 by znamenalo změnu struktury, ale navíc každá případná změna se musí provést 5 krát. Nejhorší důsledek tohoto mapování spočívá v tom, že jakékoliv vyhledávání údajů přes řádky se musí provést 5 krát a poté výsledek sloučit, tedy v tomto případě se jedná o výrazné zpomalení. Pokud bychom například chtěli vyhledat všechny řádky (a k nim faktury), které mají odkaz do daného zboží, musíme tento dotaz provést 5 krát „vedle sebe“.

Mapování kompozice ku N do RDB rozpuštěním majitele do jeho částí
Existuje ještě jedno mapování, naštěstí je také velmi výjimečné. Jedná se o případ, kdy se



Obrázek 3.10: Nepříliš obvyklé mapování kompozice ku N rozpuštěním kompozitu.

nezavede tabulka majitele, ale tento majitel se opakuje pokaždé znovu a znovu s každým řádkem (tj. „rozpustí se“ ve svých částech), viz obrázek 3.11.



Obrázek 3.11: Nepříliš vhodné mapování kompozice ku N rozpuštěním majitele.

Při tomto mapování se jednak údaje opakují a navíc se ztrácí identita vrcholu – majitele.

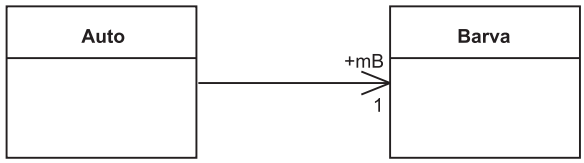
3.2.2 Vztah *Odkaz do seznamu* (slangově číselníková vazba)

V analytickém modelování v *Class Diagramu* se jedná spolu s kompozicí o nejčastěji používaný vztah. *Odkaz do seznamu* nebo také slangově *Odkaz do číselníku* nebo také *Číselníková vazba* je rovněž asociací, znamená to tedy, že se jedná o vztah sice znázorněný mezi třídami, ale jeho realizace se projeví jako vztah mezi instancemi (až vzniknou). Vztah *Odkaz do seznamu* je reprezentován jediným vzorem *Auto má barvu*.

Vzor *Auto má barvu*

Představme si systém pro evidenci aut, u kterých se eviduje také jejich barva. Tyto barvy jsou tvořeny seznamem a každé auto má vybránu jednu barvu z tohoto seznamu. Situaci

zapsanou pomocí modelu UML vyjadřuje vzor *Auto má barvu*, což lze znázornit diagramem na obrázku 3.12.



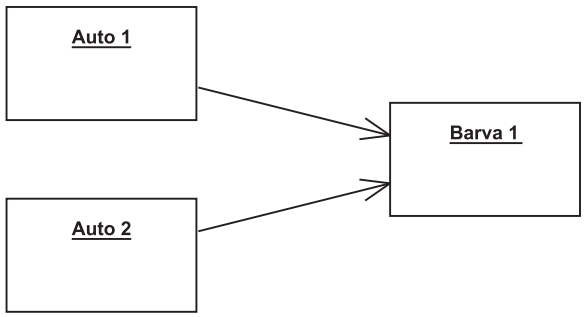
Obrázek 3.12: *Odkaz do seznamu – Auto má barvu.*

Vztah je vyjádřen spojnicí mezi třídami, na jednom konci je šipka (její význam bude vysvětlen), u ní je multiplicita 1 a je zde také uvedena role (zde mB).

Vztah s názvem *Odkaz do seznamu* reprezentovaný vzorem *Auto má barvu* je (stejně jako kompozice) asociací. Jedná se tedy opět o vztah mezi instancemi ze tříd A a B, tj. až se narodí instance ze třídy A, tak tato instance bude ve vztahu *Link* k instanci ze třídy B. To znamená, že pokud držíme instanci ze třídy Auto, potom tato instance používá (říká se také slangově „vidí“) instanci ze třídy Barva. Avšak oproti kompozici se v tomto případě nejedná o vztah „celek/část“, resp. vztah majitelství. Dokonce se dá říci, že je to přesný protipól majitelství, tj. instance ze třídy Auto sice používá instanci ze třídy Barva (tj. vidí ji a používá ji jako svoje mB), ale instance ze třídy Barva není jejím majetkem, tedy instance ze třídy A ji nerodí, neničí ji. Instance ze třídy Auto tedy instanci ze třídy Barva sice používá, ale neovládá ji jako majitel. Takovou situaci bychom mohli chápat jako „zápůjčku ukazatele na druhou instanci“.

Instance ze třídy Auto a Barva jsou tedy na sobě nezávislé v tomto smyslu: Existuje nějaká evidovaná instance ze třídy Auto na straně jedné a také existuje nějaká evidovaná instance ze třídy Barva na straně druhé, tj. v této chvíli jsou obě instance nezávislé na sobě. Poté se provede provázání vztahem mezi nimi tak, že daná instance ze třídy Auto „si ukáže“, tedy dostane odkaz, resp. vyplní si odkaz na instanci ze třídy Barva. V našem příkladě tedy existuje seznam instancí aut a druhý zcela nezávislý seznam barev. Konkrétní auta mají konkrétní barvu, tj. ukazují si na barvu, je tu odkaz auta na určitou barvu.

Důležitou a analyticky vyžadovanou vlastností vztahu *Odkaz do seznamu* je sdílení instancí. Pokud se v evidenci vyskytnou dvě instance aut se stejnou evidovanou instancí barvy, pak linky tohoto vztahu vedou k téže instanci, tedy sdílejí ji, viz obrázek 3.13.



Obrázek 3.13: Shoda odkazu na instanci barvy ze dvou instancí aut.

Všimněme si, že právě v této vlastnosti je výrazný analytický rozdíl oproti kompozici. Připomeňme, že v kompozici platilo „svůj k svému“, kdežto zde vztah vede ke sdílení instance.

Nejčastější scénáře naplnění tohoto vztahu je výběrem obsluhou podle tohoto vzoru scénáře: *„Obsluze se zobrazí seznam barev, obsluha vybere barvu a na ni se prováže editovaná instance auta. . . “.*

Protože nejobvyklejším případem použití tohoto vztahu je vztah do číselníku (tj. vztah na entitu vyjadřující jednoduchou vlastnost a mající jednoduchý obsah, například „kód + text“ apod.), vznikl slangový název tohoto vztahu také *Číselníková vazba*. Není to však přesné vyjádření, protože na místě instance, na níž se ukazuje, může být i velmi složitý prvek ve složitých vazbách, než pouze prvek z číselníku.

Vlastnost směrovost na konci asociace

V předchozích kapitolách jsme si uvedli, že na konci asociace u třídy B je třeba v analytickém modelování uvádět tři vlastnosti konce asociace. Dvě z nich (multiplicitu a roli) jsme si již objasnili. Nyní si vysvětlíme třetí vlastnost, která se nazývá směrovost a je reprezentována šipkou v asociaci. V UML je reprezentována vlastností konce asociace „isNavigable“.

Směrovost vazby vyjadřuje směr použití mezi instancemi a následně z toho plynoucí směr použití mezi třídami. Bohužel tuto vlastnost mnozí autoři analytických modelů zanedbávají a neuvádí ji. Důvodem je zřejmě ta skutečnost, že autory modelů jsou v tomto případě bývalí „databázisté“. Relační databáze totiž uvedenou směrovost svou technologií zruší a proto se směrovost vazby jeví v RDB jako neviditelná. Příčinou tohoto skrytí směrovosti vazby v relační databázi je fakt, že vztah podmínky vazby mezi tabulkami ve smyslu „where něco = něco“ je symetrický. Z toho důvodu mnohdy autoři směrovosti nevěnují náležitou pozornost. Jedná se však o chybu, jak si ukážeme v kapitole Vzor *Modulární nůžky* na straně 93.

U vzoru *Auto má barvu* lze směr šipky interpretovat tak, že instance ze třídy Auto ve svém vnitřním pohledu používá instanci ze třídy Barva ze seznamu instancí ze třídy Barva (tj. vnitřně si ukazuje na seznam barev) a proto šipka asociace směřuje „od auta k barvě“. Obráceně to však neplatí, šipka udává směr, pokud bychom mohli personifikovat, řekli bychom, že „Barva nepoužívá Auto a Barva o Autě nic neví“. V důsledku to znamená, že by bylo chybou, přesněji řečeno „zašpinění optimalizací“, tj. technologickým ústupkem, pokud by na jedné straně analytik odevzdal model ve tvaru *Auto má barvu*, ale v designu a následně v kódu by se na straně u evidovaných Barev (například ve třídě Barva, resp. jiné obslužné třídě) objevila závislost na kódu obsluhujícím evidovaná Auta, tj. pokud by se číselník Barev „zašpinil“ kódem obsluhující prvky třídy Auto.

Z toho vyplývá, že pokud se tato směrovost dodrží až do technologie, pak lze následně číselník barev „odstřihnout“ do zvláštního subsystému (tj. skupiny package v prostředí JAVA anebo skupiny assembly v .NET, knihovny v C++ apod.) a subsystém spravující auta by si jej linkoval (uses, using, import, include atd.). Jak vidno, určování směrovosti vazeb (slangově „šipkovitosti“) je důležitým krokem pro návrh modularity systému a tedy i komponentního modelu (bude uvedeno v dalších kapitolách).

Důležitá poznámka: *Pro vyjádření oboustranného vztahu se již prakticky vžil usus, že obousměrná vazba (tj. směr v použití tam i zpátky) se znázorňuje tak, že se šipky prostě neuvedou. Znamená to, že „šipka ano zapíná směrovost“, kdežto její neuvedení znamená „obousměrnost“. I když syntaxe v UML je trochu odlišná, tak takto se v modelech „šipkovitost“ používá v praxi. Pokud však studujete model tříd, je třeba dobře vědět, zda autor v modelu používá směrovost anebo tuto vlastnost prostě ignoruje a vůbec nikde neuvádí.*

Sledujte aktuální novinky na http://www.objects.cz

Jak analytik rozezná vztah *Kompozice* od vztahu *Odkaz do seznamu*?

V minulých kapitolách byla zdůrazněna zásada, že při vyhledávání tříd se nejprve vyhledávají instance jako příklady evidence a z nich dochází k zobecnění do meta-pravidla, tj. modelu tříd. Tuto radu nyní doplníme o další bod:

> *Při návrhu modelu tříd vycházíme z návrhu instancí a současně vyhledáváme nejjednodušší scénáře, které s těmito instancemi pracují. Na základě těchto scénářů určujeme povahu vazeb asociace.*

Představme si jako příklad, že bychom dopředu nevěděli, že se žádá evidovat vztah aut a barev jako vztah *Odkaz do seznamu*. Zákazník totiž v mluvě nerozlišuje povahu vztahu a všude říká jednoduché slovní spojení „obsahuje“: Faktura obsahuje řádky, Auto obsahuje barvu atd. Jak bychom tedy určili správný vztah z hlediska UML?

Pokládali bychom zákazníkovi otázky v takové podobě, abychom odhalili jednoduché scénáře, jak se bude v systému s instancemi pracovat, například takto: „Když se založí v evidenci nové evidované auto, budeme spolu s ním pokaždé znovu a znovu zakládat barvu anebo se bude barva tohoto nového auta vybírat ze seznamu?“ Odpověď: „Bude se vybírat ze seznamu“. Odhalili jsme tedy, že se jedná o vztah *Odkaz do seznamu* a nikoliv o vztah *Kompozice*. Vyplývá z toho další důležité pravidlo postupu vyhledávání tříd:

> *Při návrhu modelu tříd nejprve hledáme instance, k tomu navíc musíme nalézt také nějaký scénář, který s těmito instancemi pracuje, poté díky znalosti chování ve scénářích zobecňujeme do modelu tříd také odpovídající typy asociací.*

Scénáře s instancemi se zapisují v jiném modelu – v *Use Case Modelu*. Z toho plyne, že z pohledu posloupnosti vývoje se tedy nejprve vyhledává model případů užití, ve kterém jsou tyto scénáře obsaženy. Zde však v této knize vysvětlujeme model tříd jako první z toho důvodu, protože chceme dobře napsat scénáře, k čemuž musíme velmi dobře vědět, jaké vztahy jsou pro instance a následně třídy možné a platné. Díky této znalosti potom také víme, na co se tedy zákazníka efektivně ptát a co z něho tak říkajíc dostat. Pro vztahy mezi třídami a scénáři mezi instancemi platí navíc důležité a jednoduché logické pravidlo:

> *Pokud díky jednomu scénáři mezi instancemi nalezneme vztah mezi třídami (tedy vzorec pro instance), potom každý jiný scénář, který pracuje s instancemi z těchto tříd, musí tento vzorec dodržet a nemůže se chovat podle jiného vzorce.*

Musí proto platit, že pokud se v jednom scénáři instance auta a barvy chovají jako vztah *Odkaz do seznamu*, v jiném dalším (tj. ještě nenalezeném) scénáři to nemůže být kompozice a naopak. Doporučuji proto nalézt a využít vztahy v meta pravidlu, tj. *Class Diagramu*, protože to nám ušetří spoustu času při psaní scénářů: Pokud totiž co nejdřív odhalíme model tříd a typy vztahů *Asociace*, tak se nám osvětlí, v jaké základní logice musejí další scénáře fungovat.

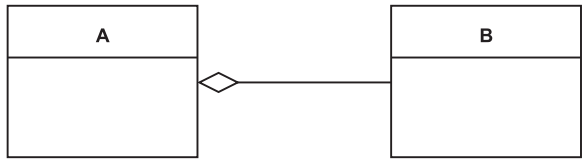
Jako první ještě neznámý scénář, který má odhalit povahu vztahu mezi instancemi, je třeba zvolit takový, který je pro zákazníka co nejjednodušší, který zákazník dobře zná a umí si jej představit a popsat. Většinou se jedná o scénář nějaké „práce s instancemi na obrazovce“. Doporučuji se zaměřit zejména na scénáře vzniku instancí a vzniku vztahů mezi nimi.

Mapování vztahu *Odkaz do seznamu* do relační databáze

Mapování do relační databáze je v tomto případě jednoduché a nezná optimalizaci: Vyjděme ze vzoru *Auto má barvu*. Technolog-databázista zavede dvě tabulky, pro každou analytickou třídu po jedné, nazvěme je například TAUTO a TBARVA. Relaci mezi tabulkami, která bude realizovat uvedenou analytickou vazbu ku jedné, uskutečníme pomocí cizího klíče, který putuje proti směru šipky. Ve vzoru *Auto má barvu* bude putovat do tabulky TAUTO cizí klíč s vazbou na primární klíč z tabulky TBARVA. Tímto klíčem může být buď tzv. ídéčko (IDENTITY, SERIAL, SEQUENCE atd.) anebo u číselníků již dále zaručeně neměnný jednoznačný kód (pozor na Murphyho zákony!).

3.2.3 Vztah *Sdílená agregace*

Vztah *Sdílená agregace* (angl. *Shared Aggregation*) je oproti předešlým vztahům poměrně výjimečně se vyskytující vztah, který se svým způsobem podobá kompozici, protože se také jedná o vztah „celek/část“ neboli majitelství. Vztah *Sdílená agregace* se zavede podobně jako kompozice – spojnicí mezi třídami a jeden z konců asociace se označí kosočtvercem, který však není černý, ale je prázdný, viz obrázek 3.14.



Obrázek 3.14: Vztah *Sdílená agregace*.

Vše, co bylo z hlediska zápisu řečeno u kompozice, platí i zde – na koncích asociace se opět uvádějí tři vlastnosti:

- role;
- multiplicita;
- směrovost.

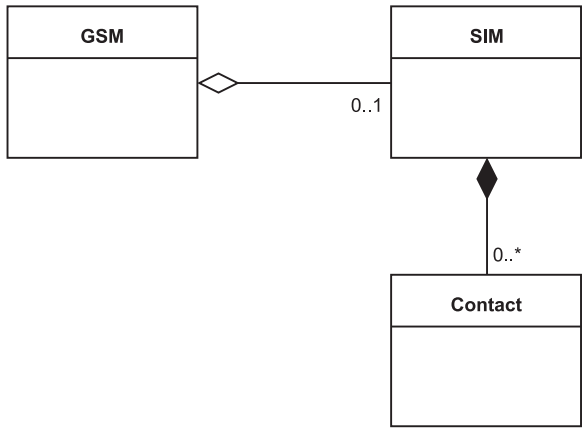
Vztah *Sdílená agregace* tedy podobně jako *Kompozice* také vyjadřuje viditelný vztah „celek/část“, kde budoucí instance majitel je označena kosočtvercem, ale majitelství je ve sdílené agregaci oproti kompozici relativně volnější. Na rozdíl od kompozice je povolena existence alespoň jednoho z těchto scénářů (POZN.: *musíme připomenout, že ani jeden z nich není v kompozici dovolen!*):

1. Instance ze třídy B může žít dočasně sama, může se narodit (vzniknout) dříve než její majitel (instance ze třídy A) anebo naopak může svého majitele přežít.
2. Instance ze třídy B může přeputovat k jinému majiteli, tj. nejprve žije pod jedním majitelem, poté se „zaživa“ přestěhuje k jinému majiteli.
3. Instance ze třídy B může mít současně jako instance vícero majitelů.

Sledujte aktuální novinky na http://www.objects.cz

Vzor *GSM telefon má SIM-kartu*

Tento vzor vystihuje velmi výstižně vlastnosti vztahu *Sdílená agregace* a současně názorně vysvětluje rozdíl sdílené agregace oproti kompozici. Na obrázku 3.15 byl zvolen vztah „GSM telefon drží SIM-kartu“ jako *Sdílená agregace*. Telefon i s kartou chápeme jako celek, ale je dovoleno SIM-kartu vyjmout a pak bude žít samostatně mimo telefon. Na druhé straně kontakty na SIM-kartě (nemyslím měděné, ale adresář telefonů) putují nerozlučně se SIM-kartou a proto je v tomto vztahu zvolena kompozice. Například rozšlapat telefon neznamená, že bude určitě rozšlapána i SIM-karta, protože ta v té chvíli mohla být na stole anebo v jiném telefonu. Oproti tomu rozšlapat SIM-kartu znamená rozšlapat vždy i kontakty na ní.



Obrázek 3.15: Vzor *GSM telefon má SIM-kartu*.

V evidenčních systémech nastává situace sdílené agregace sice výjimečně, ale je třeba s ní počítat.

Vzor *Agregace položek ex post*

V některých případech se může stát, že nějaký prvek sice obsahuje svoje položky podobně jako faktura má řádky, ale nejprve se ve scénáři vytvářejí položky a teprve poté jejich majitel. Nejprve se tedy zakládají samostatné položky a ty se poté v určitém procesu agregují podle nějaké podmínky do následně vzniklého majitele. V tomto případě bychom neměli vztah mezi majitelem a položkami malovat jako kompozici, protože majitel položek vznikl až následně po vzniku a agregaci položek. Stejně tak, pokud se s položkami dodatečně „hýbe“ a přesouvají se z jednoho agregátu do druhého, neměl by se tento vztah malovat jako kompozice.

Jako příklad bych uvedl evidenční systém webhostingu, ve kterém si zákazník během určitého období vyžádal určité služby. S každou realizovanou službou se mu založil řádek faktury, avšak faktura zatím neexistovala. V procesu fakturace se vytvořila „hlavička“ faktury agregací všech již existujících řádků daného zákazníka. V tomto případě by se vztah majitelství faktury a řádků neměl znázorňovat jako kompozice, protože majitel vznikl ex post po položkách.

3.2.4 Vztah *Asociační třída*

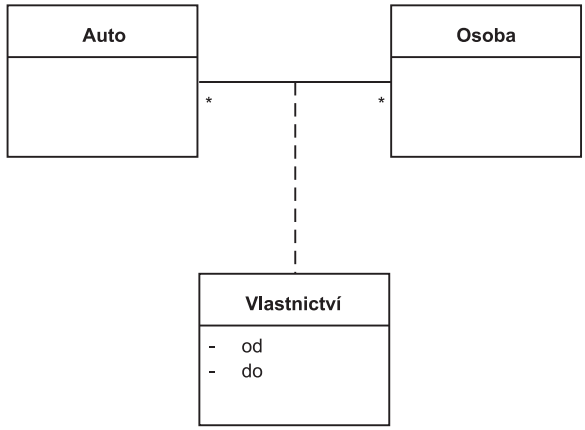
Asociační třída je v UML definována poměrně stručně: *Asociační třída* je třídou, která je současně i asociací.

Jinak a srozumitelněji řečeno to znamená, že instance asociační třídy mají schopnost dát jiné instance z jiných tříd do vztahu mezi sebou. Zní to sice složitě, ale po vysvětlení „jak to funguje“, se vyjasní tato definice a také smysl použití tohoto vztahu.

Vzor *Auto, Osoba, Vlastnictví*

Představme si, že máme v systému seznam evidovaných aut a seznam evidovaných osob. Jedná se o dva nezávislé seznamy, které o sobě „nevědí“ a žijí svými „nezávislými životy“, tj. není mezi nimi žádný vztah například vyjádřený vzory *Faktura má řádky* anebo *Auto má barvu*. Slangově řečeno, jedná se o dva vedle sebe žijící „číselníky“.

Přijde nový požadavek, že se žádá evidovat, které auto bude vlastněné kterou osobou a naopak, která osoba bude vlastnit které auto. Navíc se žádá evidovat také „odkdy dokdy“ toto vlastnictví platilo. Všimněme si, že díky tomu je počet instancí mezi osobami a auty v tomto vztahu „kdo co vlastní, resp. co je kým vlastněno“ v multiplicitě M:N. Možných řešení tohoto požadavku je několik, my zvolíme řešení pomocí asociační třídy. Nejprve si ukážeme, jak asociační třídu zavedeme a poté jak funguje. Zaveďme tedy třídu Vlastnictví jako asociační třídu tak, jak je uvedeno na obrázku 3.16.

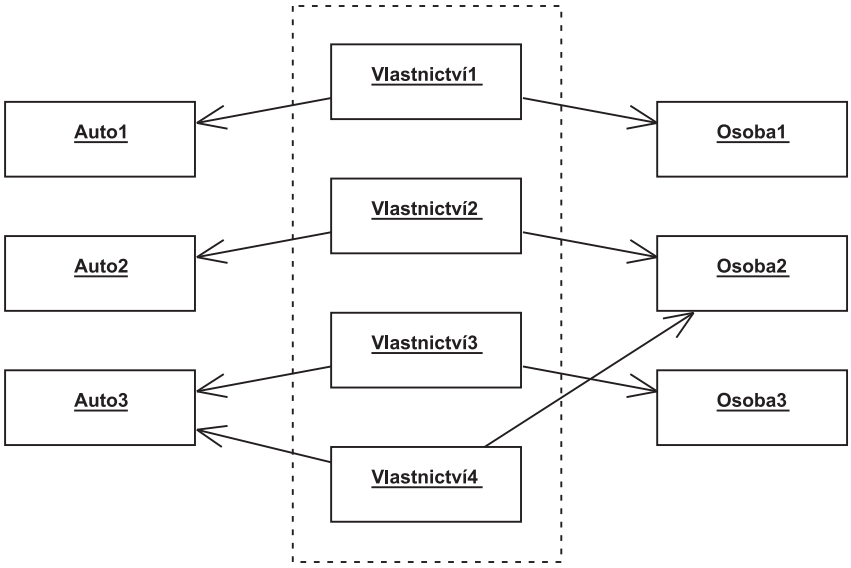


Obrázek 3.16: Evidence vlastnictví aut osobami zprostředkovaná asociační třídou.

Abychom poznali, jak funguje v modelu tříd jakákoliv asociace, musíme pochopit, jak funguje tento vztah mezi instancemi. Z toho důvodu si vysvětlíme, jak funguje asociační třída na úrovni instancí.

Základní pravidlo pro instance asociační třídy zní: Každá instance z asociační třídy má vždy po jednom odkazu na jednu instanci z každé třídy, které propojuje. Jinak řečeno, v instanční rovině má odkaz na jednu instanci z jedné třídy a jeden odkaz na jednu instanci z druhé třídy. Jako příklad evidence instancí předešlého modelu *Auto, Osoba, Vlastnictví* si uveďme následující obraz instancí, viz obrázek 3.17.

Vlevo vidíme několik evidovaných instancí aut, vpravo několik evidovaných instancí osob, tedy dva opravdu nezávislé seznamy. Uprostřed jsou znázorněny instance z asociační třídy vlastnictví. Všimněme si, že každá instance Vlastnictví (čtěte je po řadě shora dolů) má po jednom odkazu vlevo a vpravo, tj. Vlastnictví1 má odkaz na Auto1 a Osoba1, Vlastnictví2 má odkaz na Auto2 a Osoba2, Vlastnictví3 má odkaz na Auto3 a Osoba3, Vlastnictví4 má odkaz na Auto3 a Osoba2. To je první důležitá vlastnost instancí z asociační třídy:



Obrázek 3.17: Příklad instancí v evidenci Aut, Osob a Vlastnictví.

Každá instance asociační třídy má po jednom odkazu na instance z každého seznamu, které provazuje.

Úkolem prvků asociační třídy je propojit instance zleva a zprava a naopak. K tomu slouží druhá vlastnost instancí (současně viz obrázek 3.17):

Seznam instancí z asociační třídy bude mít povinně implementovánu funkcionalitu průchodu zprava doleva a naopak, což si můžeme prakticky představit jako implementaci dvou filtrů. Jeden funguje nad polem ohraničeným čárkovanou čarou „zprava“ a druhý „zleva“.

Filtr zleva nazvěme „Filtr pro auta“ a bude mít jako vstupní parametr Auto. Výstupem budou takové instance Vlastnictví (tj. seznam Vlastnictví) ze všech Vlastnictví, která si ukazují na tento vstupní parametr Auto. Pokud tedy bude vstupním parametrem Auto1 (viz obrázek 3.17), výstupem bude seznam s jedním prvkem Vlastnictví1, pokud bude vstupním parametrem Auto3, výstupem bude seznam o dvou prvcích – Vlastnictví3 a Vlastnictví4. Shrňme tedy, co se díky vlastnostem asociační třídy v systému děje v tomto příkladu:

1. Máme v ruce evidovanou instanci, například Auto3.
2. Pustíme filtr nad instancemi asociační třídy a vyfiltrujeme ty instance Vlastnictví, které si na něho ukazují. (V příkladu s instancí Auto3 držíme nyní v ruce Vlastnictví3 a Vlastnictví4).
3. Díky tomu, že držíme seznam instancí Vlastnictví, můžeme každou z nich požádat o další údaje (viz kapitola vnější a vnitřní pohled v objektovém paradigmatu), například údaje odkdy dokdy, a o instanci Osoby, ze které získáváme vnějším pohledem další údaje.

Asociační třída nám svými prvky provázala instanci ze třídy Auto (zde Auto3) s prvky ze třídy Osoba (zde Osoba3 a Osoba2) a navíc nám ještě poskytla informaci odkdy dokdy

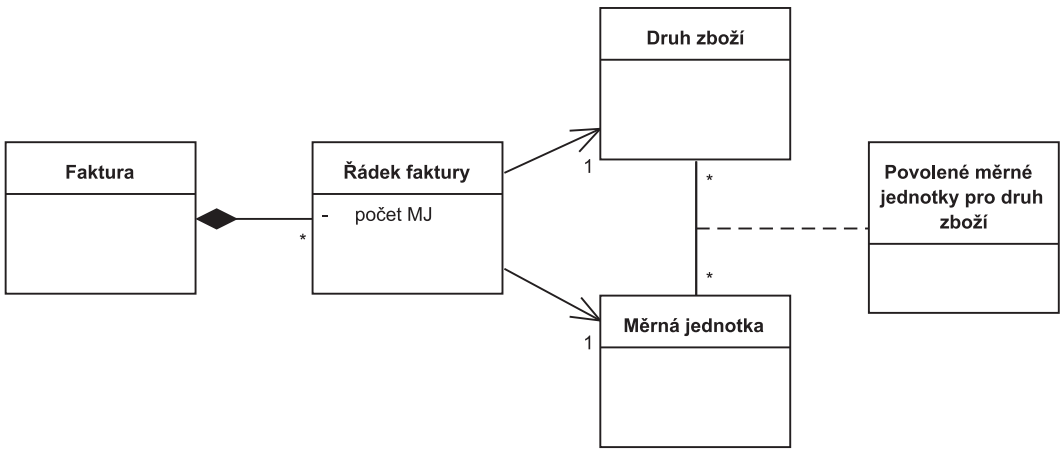
vlastnictví trvalo. Je zřejmé, že úplně stejně můžeme zavést obrácený postup, filtr zprava pro Osoby a jistě by byl takový průchod zprava doleva žádoucí. Asociační třída tedy zavádí instance, které vytvářejí vztahy mezi jinými instancemi.

> *Instance asociační třídy propojují instance ze tříd (tj. tvoří asociaci) tak, že každá instance asociační třídy má po jedné instanci z každého seznamu vztahu a díky filtrům nad nimi se realizuje průchod z jedné strany vztahu na jeho druhou stranu a to včetně možnosti získání informace uprostřed umístěné v prvku asociační třídy.*

Vzor *Povolené kombinace*

Velmi častým použitím vztahu *Asociační třídy* je vzor *Povolené kombinace*. Asociační třída v rámci vytváření vztahů mezi dvěma nezávislými seznamy umožňuje omezit, resp. povolit určité kombinace, což má v analytické praxi velmi časté použití.

Představme si evidenci faktur. Faktura obsahuje řádky faktury a ty obsahují údaje, co se fakturuje. Těmito údaji jsou: Odkaz do seznamu Druhu zboží (co se fakturuje), počet měrných jednotek (tj. kolik) se fakturuje a odkaz do seznamu Měrné jednotky. Je zřejmé, že určitý Druh zboží se může fakturovat jen v určitých Měrných jednotkách, vzniknou tak povolené kombinace zboží a měrných jednotek. Obrázek 3.18 je příkladem takového využití asociační třídy.



Obrázek 3.18: Povolené kombinace Druh zboží versus Měrné jednotky.

Instanční obsahy tříd Druh Zboží, Měrné jednotky a Povolené měrné jednotky pro druh zboží mohou být zapsány v textovém tvaru například tak, jak je uvedeno v tabulce 3.1.

Tabulka 3.1: Instanční obsahy tříd.

Druh zboží	Měrné jednotky	Povolené MJ pro druh zboží
	1 kus	1 [1,1]
1 rohlík	2 litr	2 [2,2]
2 mléko	3 hektolitr	3 [2,3]
3 mouka	4 kg	4 [3,4]
	5 cent	5 [3,5]

Sledujte aktuální novinky na http://www.objects.cz

Význam instancí ze třídy Povolené MJ pro Druh zboží je nyní zřejmý, jedná se vlastně o „dvojkombinace“ odkazů propojujících oba dva seznamy. Díky tomu se v dynamice scénářů omezují povolené kombinace z druhé strany, například bude vybrán Druh zboží číslo 2 (mléko), tím se zúží výběr instancí na ty, které mají v pravém sloupci 2 a nabídnou se měrné jednotky pouze 2 a 3, tj. litr a hektolitr.

Syntaxe vztahu M:N s chybějící asociační třídou

Pokud se podíváme na předešlý výčet instancí ze třídy Povolené měrné jednotky pro Druh zboží (viz předešlý odstavec), potom je zřejmé, že nemá smysl v tomto výčtu podruhé zavést již existující kombinaci (tj. takovou, která tam už je), například v seznamu dvojkombinací znovu zopakovat kombinaci [1,1]. Takováto opakující se kombinace nepřináší žádnou novou informaci a dokonce by bylo nežádoucí, aby se v seznamu znovu vyskytla, protože by se mohla opakovat několikrát, tedy „bůhví kolikrát“. Z toho důvodu bychom měli zakázat toto zdvojení a nasadit podmínku unikátnosti na dvojici instančních odkazů do seznamu. Všimněme si, že současně v instancích tříd není žádná jiná informace, než pouze unikátní dvojkombinace odkazů.

Pokud takováto situace při návrhu IS v analytickém modelu nastane, tj. asociační třída nenese žádnou další informaci a dvojkombinace je unikátní, pak jazyk UML umožňuje znázornit vztah M:N bez asociační třídy ve zkrácené podobě tak, jak je uvedeno na obrázku 3.19.

Obrázek 3.19: Vztah M:N bez asociační třídy.

Tento zápis sice jazyk UML dovoluje, ale pokud provádíte analytické modelování informačních systémů, doporučuji jej doplnit o asociační třídu hned a na začátku analytického modelování.

Poznámka k technologickým systémům

V technologických systémech, u kterých není v pozadí databáze, je snaha nahradit asociační třídu některým jiným z konkurenčních vzorů, které budeme brát v seznamu vzorů. Důvodem tohoto postupu speciálně u technologických systémů je skutečnost, že v těchto systémech může být technologickým problémem implementovat ručně v programu bez podpory databáze „rychlý filtr“ uvedený jako nutná implementační podmínka funkčnosti asociační třídy. Proto se autoři technologických systémů asociační třídě většinou vyhýbají a používají velmi blízké náhradní vzory.

Vztah *Asociační třída s vícero konci*

Až do verze 1.7 jazyk UML rozlišoval dva typy prvků, které se týkaly asociační třídy:

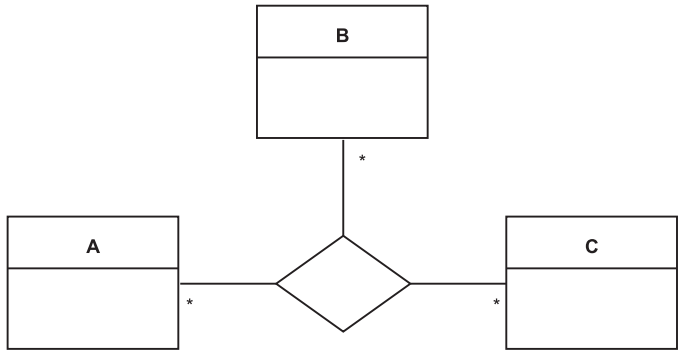
1. *Asociační třídu se dvěma konci* (tak, jsme ji brali v předešlé kapitole);
2. *Asociační třídu s vícero konci* (tzv. N-ary Association Class).

Od verze 2.0 zavádí jazyk UML pouze jeden pojem *Asociační třída*, která může mít dva anebo více konců. Tato změna je vcelku pochopitelná, protože vlastnosti asociační třídy s vícero konci než dva se od „dvojné“ asociační třídy liší skutečně pouze počtem konců.

V předešlých kapitolách jsme zavedli a vysvětlili asociační třídu, která propojovala dva nezávislé seznamy pomocí svých instancí obsahujících dvojici odkazů do dvou seznamů, což v předešlém výčtu reprezentuje první typ prvku. Nyní si probereme druhý bod předešlého výčtu.

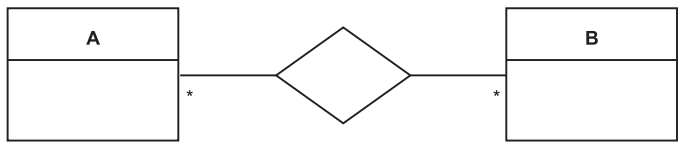
Asociační třída s vícero konci se liší od předešlé s dvěma konci počtem prvků (seznamů), které propojuje. Prvky asociační třídy s vícero konci tedy mohou propojit například tři nezávislé seznamy (případně více seznamů, ale to je opravdu výjimečná situace). Znamená to, že namísto o evidované dvojici odkazů propojujících dva seznamy nyní hovoříme o evidované trojici případně čtveřici odkazů (POZN.: *v praxi jsem se s více jak třemi nesetkal*). Je zřejmé, že asociační třída s vícero konci funguje na stejném principu jako dvojná, tedy ta, kterou jsme dosud probírali. Rozdíl je pouze v počtu seznamů, které se asociační třídou propojují. Syntaxe asociační třídy s vícero konci je však z historických důvodů trochu odlišná, než je syntaxe dvojné asociační třídy.

Pro zavedení vícenásobné asociační třídy se využívá jiné znázornění vztahu *Asociace*. Zatím jsme pro asociaci používali pouze spojnici. Prvek *Asociace* lze v *Class Diagramu* zavést i druhým způsobem, jako velký kosočtverec. Propojení mezi tímto prvkem *Asociace* a třídami pomocí plných čar znázorňuje, které třídy jsou asociací propojeny, například jako na obrázku 3.20.



Obrázek 3.20: Trojná asociace.

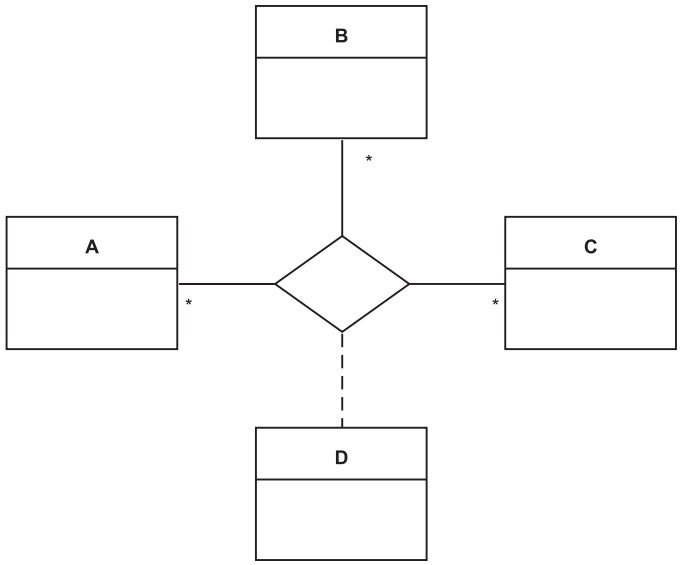
Plyne z toho, že asociace mezi dvěma třídami lze tedy namalovat buď jako spojnice (jak jsme ji brali) anebo touto ekvivalentní syntaxí i s vloženým kosočtvercem. Obrázky 3.19 a 3.21 jsou si tedy ekvivalentní.



Obrázek 3.21: Druhá varianta zápisu asociace.

Podle definice UML je asociační třída třídou a současně asociací. Pokud tedy k dané asociaci ve tvaru kosočtverce připojíme čárkovanou čarou ještě další třídu, znázornili jsme tak

asociační třídu reprezentující současně i tuto asociaci. Asociační třída je tedy ke kosočtverci připojena čárkovanou čarou a ostatní třídy, které propojuje, jsou připojeny plnými čarami, viz obrázek 3.22.



Obrázek 3.22: Asociační třída D s třemi konci propojující prvky z A, B a C.

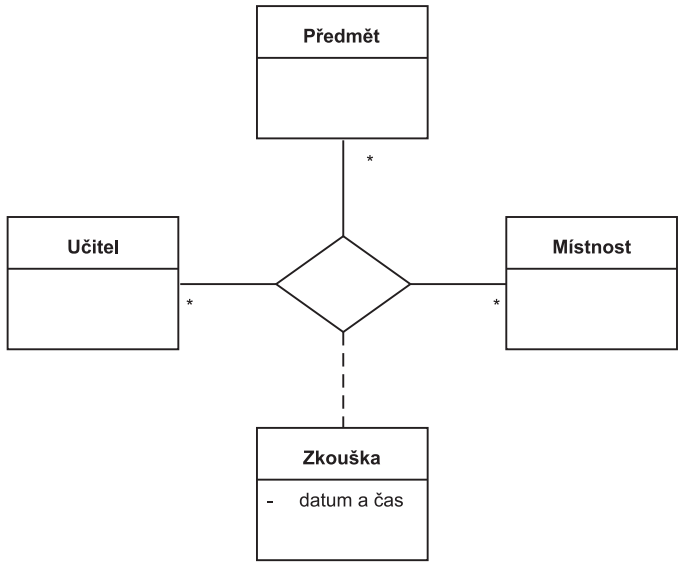
Je třeba zdůraznit, že u trojné asociační třídy v instanční rovině existují také odkazy, stejně jako u dvojné, ale nikoliv dva, ale tři. Představme si, jako by v obrázku 3.22 byly schovány tři šipky z D: Jedna do A, jedna do B, jedna do C. Na tyto šipky působí filtrace a díky ní dostáváme pro daný prvek prvky z druhé strany, avšak díky většímu počtu propojených seznamů mohou být kombinace bohatší. Například chytneme jeden prvek A, dostaneme na druhé straně prvky B a C, pak ještě z nich chytneme jeden prvek B a dostaneme již pouze prvky C atd.

Vzor *Učitel, Předmět, Místnost, Zkouška*

Nechť máme za úkol vytvořit evidenci zkoušek na vysoké škole, na něž se mají studenti hlásit přes internet. Nebudeme se zabývat tou částí evidence, která řeší to, že pokud je student v daném semestru a daném oboru, tak potom má omezen výběr zkoušek (různé systémy kreditů za zkoušky apod.). Takové algoritmy má každá škola jiné, a proto bychom to řešili specificky podle zadání dané školy. Věnujme se té části, která zavádí zkoušku, kterou si student poté vybírá a na kterou se hlásí.

Pro odhalení vztahů zvolme scénář založení nové zkoušky v systému, nechť zní takto: „Obsluha zakládá novou zkoušku. Vybere jednoho učitele ze seznamu učitelů, který bude garantem zkoušky, vybere jeden předmět ze seznamu předmětů (udává, z čeho zkouška je) a vybere místnost, kde se zkouška koná, obsluha zadá datum a čas.“ Jedna instance zkoušky tedy obsahuje tři odkazy na tři instance ze tří seznamů: jeden odkaz na instanci učitele, jeden na instanci předmět a jeden na instanci místnost. Zobecníme tedy řešení do modelu tříd ve tvaru asociační třídy, viz obrázek 3.23.

Důvod, proč je třída Zkouška zvolena právě jako asociační třída a nikoliv pomocí tří vztahů typu *Odkaz do seznamu* (tj. vzor *Auto má barvu*), je v tom, že je jasný evidentní



Obrázek 3.23: Příklad asociační třídy s vícero konci.

„průskok“, tedy vazba mezi prvky seznamů. Student při výběru zkoušky například postupuje takto: Zobrazí se mu seznam všech předmětů (jak uvedeno, neřešíme omezení), student vybere předmět, v té chvíli se zúží seznam všech zkoušek na ty, které mají tento předmět. Zobrazí se již tímto omezené seznamy učitelů a místností, pak třebas vybere učitele a dojde k dalšímu zúžení (filtraci). Nakonec se student dobere až k výběru zkoušky.

Souhrn vztahů v *Class diagramu*

Probrali jsme vztahy, které analytik potřebuje k tomu, aby analyticky vyjádřil asociaci různého významu.

Proveďme malou inventuru všech doposud známých vztahů, které používá analytik v *Class Diagramu* při analytickém modelování.

Interakce v analytickém *Class Diagramu* se dělí takto:

A) *Asociace* (vede ke vztahu mezi instancemi)

1) *Kompozice*

Vzor *Osoba má adresu* (jako v e-shopu)

Vzor *Faktura má řádky faktury*

2) *Odkaz do seznamu*

Vzor *Auto má barvu*

3) *Sdílená agregace*

Vzor *GSM telefon má SIM-kartu*

Vzor *Agregace položek ex post*

4) *Asociační třída*

Vzor *Auto, Osoba, Vlastnictví*

Vzor *Povolené kombinace*

Vzor *Učitel, Předmět, Místnost, Zkouška*

B) *Generalizace* (zatím jsme nebrali!)

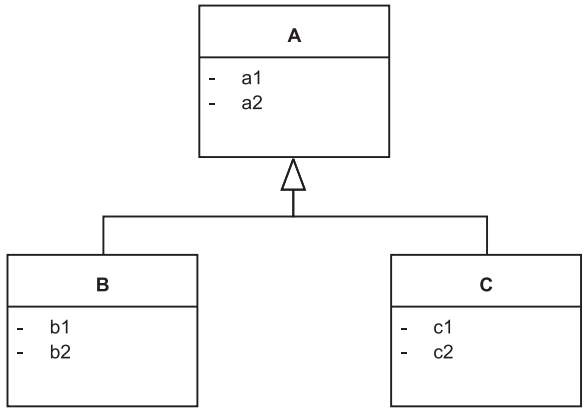
Posledně uvedený vztah *Generalizace* je „z úplně z jiného soudku“, než jak jsme zatím vztahy *Asociace* chápali. Je „fyzikálně řečeno“ z jiného prostoru – budeme ho brát v další kapitole.

3.2.5 Vztah *Generalizace*

Vztah *Asociace*, který jsme brali doposud, vyjadřuje budoucí vztah mezi instancemi, i když je znázorněn na úrovni *Diagramu tříd*. Asociace je pravidlem pro budoucí instance ve znění: Až se narodí tato instance z této třídy, bude ve vztahu link k jiné (nebo jiným) instancím. Jedná se tedy o pravidlo, jak budou instance mezi sebou „sešněrovány“, až se narodí. To je jeden z důvodů, proč je vztah *Asociace* dobře viditelný a odhalitelný a proto se hledá vždy jako první. Oproti tomu vztah *Generalizace* je interakce přímo mezi třídami a nevede ke vztahu mezi instancemi.

Připomeňme si, že z hlediska analytického významu se asociace ještě dále dělí podle označení konců asociace (černý kosočtverec, bílý kosočtverec, žádný kosočtverec), podle multiplicity a vlastnosti „směrovost“, jak bylo probráno v předešlých kapitolách pomocí analytických vzorů.

Generalizace má slangový výraz *dědičnost*, protože do technologie OOP se mapuje právě pomocí inheritance neboli dědičnosti. V *Class Diagramu* se znázorňuje pomocí spojnice mezi třídami, přičemž na jednom konci je trojúhelník a směr šipky trojúhelníku vyjadřuje směr použití, tj. která třída kterou třídu v této interakci používá. Většinou se maluje jako strom se šipkami zespodu nahoru, viz obrázek 3.24.



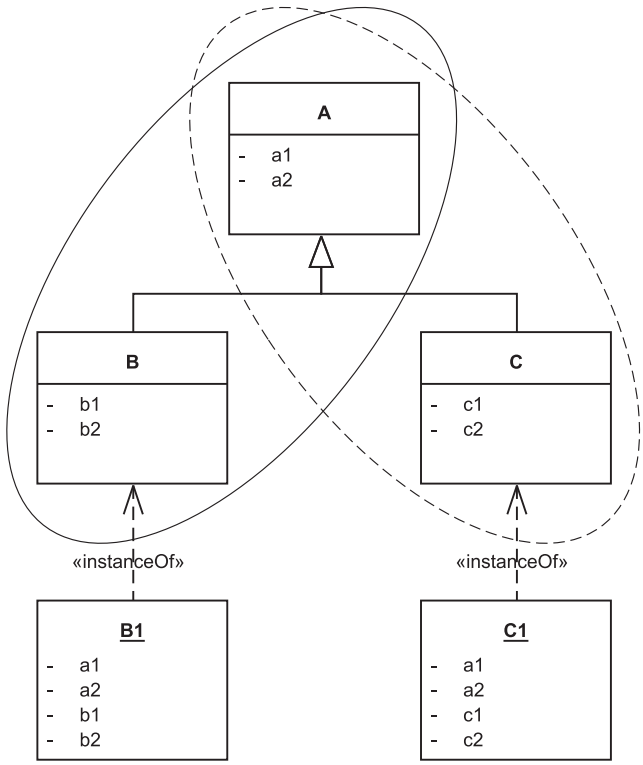
Obrázek 3.24: Vztah *Generalizace*.

Z obrázku 3.24 je patrné, že třída B používá třídu A a stejně tak třída C používá třídu A. Jsou zde zavedeny tři třídy A, B, C. Každá z nich se chápe jako „předpis“ pro instance (zatím ale dopředu neříkáme, jak instance vznikají, pouze víme, že třídy A, B, C jsou předpisem, tedy „kuchařkou“). Třída A definuje atributy a1, a2 pro budoucí instance. Podobně i třída B zavádí atributy pro budoucí instance b1, b2 a třída C zavádí atributy c1, c2.

Pokud by tedy generalizace nebyla na obrázku 3.24 zavedena, tak by z třídy A vznikaly instance s atributy a1, a2, z třídy B instance s atributy b1, b2 a ze třídy C instance s atributy c1, c2. Jenomže navíc je na diagramu znázorněn i vztah *Generalizace* a to při

zrodu instancí výrazně mění situaci. Obrázek 3.24 ukazuje, že třída B používá třídu A ve vztahu *Generalizace* a stejně tak třída C také používá třídu A ve vztahu *Generalizace*. Díky generalizaci tyto třídy spolu interagují jako dva předpisy, tedy třída B (jako předpis) použije třídu A (jako předpis – tedy pozor, zde není řeč o vztahu mezi instancemi!). Třídě, která je v generalizaci použita (zde A), se říká předek, a třídě, která používá (zde B a C), se říká potomek. Vrchní třída je třídou více obecnou a třída potomek je třídou více speciální.

Zavedení vztahu *Generalizace* v tomto případě znamená, že třída B použila třídu A ještě před vznikem instancí. Nejedná se tedy o interakci mezi instancemi, ale o interakci přímo mezi třídami, tedy o interakci, kdy jeden předpis použije rovnou a přímo druhý předpis, když rodí svou instanci. Důsledkem zavedení vztahu *Generalizace* na obrázku 3.24 je, že když požádáme třídu B o vznik instance, tak třída B se kromě svého předpisu „podívá“ přes interakci generalizace také na předpis třídy A a vydá instanci, která bude mít vlastnosti podobné situaci, jako kdybychom obě třídy (tedy předpisy obou tříd) slili dohromady. Totéž platí samozřejmě i pro třídu C. Tuto skutečnost bychom mohli znázornit pomocí obrázku 3.25, kde si namalujeme také instance vzniklé z těchto tříd.



Obrázek 3.25: Struktura instancí B1 a C1 z tříd potomků.

Všimněte si na obrázku 3.25 těchto důležitých okolností: O vznik instance žádáme třídu B (resp. C). Vzniká instance, na obrázku je označena jako B1 (resp. C1). Tato instance (když ji „rozpitváme“) vypadá, jako by byla složena ze dvou částí, první část instance pochází z definice dané třídou B a druhá část pochází z definice dané třídou A. Přitom je důležité si uvědomit, že jsme při žádosti oslovili pouze třídu B a interakci generalizace při této žádosti nevidíme (viz vlastnost Vnější a vnitřní pohled na prvky, služby a jejich implementace na straně 9).

Generalizace jako obdoba skládání předpisů

Vztah *Generalizace* je interakcí mezi třídami, která dává možnost, aby jeden předpis neboli třída potomek mohl použít druhý předpis neboli druhou třídu, tedy předka. Vznikají tak ze třídy potomka instance, které vypadají tak, jako by se oba předpisy sloučily dohromady. Ale pozor, neděláme sami toto „slití“, žádáme o instanci vždy pouze jednu třídu potomka, a ta za svou hranicí díky generalizaci interaguje s jinou třídou, předkem. Tato interakce je tranzitivní v tom smyslu, že pokud máme další interakci generalizace výše (například nad třídou A by byla ještě třída předek X), tak i tato žádost se přenese nahoru a tím vznikne instance slitá ze všech předků od potomka směrem nahoru.

> *Doporučuje se číst vztah* Generalizace *ve směru šipky trojúhelníku, tedy od potomků k předkům, jak určuje směr vztahu, a nikoliv naopak.*

Jak fungují vztahy *Generalizace* a *Asociace* v šuplíkové kartotékové evidenci

Vzpomeňme na příklad papírové evidence, když jsme si pomocí kartiček vysvětlovali, jak funguje vzor *Dichotomie třída-instance*. Nyní je vhodná příležitost si pomocí tohoto názorného příkladu vysvětlit rozdíl mezi vztahem *Asociace* a *Generalizace*.

Zaveďme tedy papírovou kartotékovou evidenci (viz kapitola 1.4.2 na straně 24), kde ke každému šuplíku je přiřazena jedna prázdná kartička „vzor“, neboli třída, která slouží pouze jako předloha pro vznik nových evidenčních kartiček kopírováním z této šablony. Takovýchto evidenčních šuplíků máme několik.

Vztahy *Asociace* zavádějí pravidla, jak budou provázány jednotlivé evidované kartičky mezi sebou. Například vzor *Odkaz do seznamu* funguje tak, že na jedné evidované kartičce v šuplíku je uveden „odkaz“ na jinou kartičku v jiném (nebo stejném) šuplíku. Jako příklad vzoru *Auto má barvu* zaveďme šuplík Barev a šuplík Aut. Každá evidenční kartička Auta bude mít nějak v sobě udělaný odkaz na nějakou evidenční kartičku Barvy (například očíslováním kartiček). Například konkrétně evidenční kartička číslo 145 v šuplíku aut má barvu číslo 13, což je „červená“. Jiná evidenční kartička aut má například tutéž barvu „červená“, tedy má také odkaz na barvu číslo 13. U kartiček tříd neboli „šablony“ to znamená, že šablona auta má v sobě zaveden budoucí „odkaz“ (například „zde vyplňte číslo barvy“) do šuplíku barev, následně konkrétní evidenční kartička auta má konkrétní odkaz (link) na konkrétní kartičku barvy. Podobně kompozice umožňuje „slepovat“ neboli skládat kartičky do sebe.

Asociační třídu si lehce vysvětlíme na příkladu evidence sňatků na matrice. Zaveďme šuplík Muž, šuplík Žena (anebo jeden šuplík Osoba a odlišíme muže a ženy evidencí pohlaví) a třetí šuplík Sňatek. Každá kartička sňatku obsahuje jeden odkaz do šuplíku žen, jeden odkaz do šuplíku mužů a datum a čas, případně další údaje.

To vše jsou asociace, které vedou ke vztahům mezi evidenčními kartičkami v šuplících. Vztah *Generalizace* však oproti asociaci funguje „úplně jinak a jinde“. Jedná se o vztah již přímo mezi kartičkami „předpisy“, tedy o interakci již mezi kartičkami-vzory. Zaveďme podle předešlého příkladu šuplíky A, B a C a vztah *Generalizace* od třídy B k třídě A a od třídy C k třídě A, viz obrázek 3.25. Pokud budeme chtít vytvořit novou evidenční kartičku v šuplíku B, tak musíme postupovat takto: Vezmeme kartičku vzor (třídu) B, zjistíme, že má vztah *Generalizace* k třídě A, proto k ní přidáme kartičku vzor třídu A, *obě dvě* vsuneme do kopírky a vznikne nová evidenční kartička do šuplíku B, která má vlastnosti jak B, tak A.

K interakci generalizace tedy dochází před vznikem instancí a je to interakce „přímo a rovnou mezi předpisy“.

Nezávislost instancí potomků

Všimněme si druhé důležité okolnosti: Co mají společné instance B1 a C1 ve smyslu, zda se mohou navzájem použít? Jsou v nějakém vztahu mezi sebou, resp. vidí se, či snad ovládají (viz vztahy v předešlých kapitolách)? Odpověď zní nikoliv, jsou to nezávislé instance.

Nedejme se zmást tím, že na obrázku 3.25 vidíme prolínající se plnou a čárkovanou elipsu. Ty jsou namalovány v prostoru tříd a tento průnik elips říká: Až se narodí instance z B, bude mít v sobě část své struktury stejně definovanou (ze třídy A) jako v instancích C a nic víc. Část definice struktury se sdílí, ale to není sdílení instancí, avšak sdílení definice vlastností, tedy společná definice struktur (viz „společná“ políčka a1, a2, ale každá instance má svoje políčka a1, a2 se svými konkrétními hodnotami!).

Abstraktní třída

Dokud jsme nezavedli vztah *Generalizace*, nemělo smysl zavádět pojem abstraktní třída. Doposud totiž měla třída jediné možné použití, a to rodit instance. Nyní však ještě může třída figurovat jako předek v generalizaci, což mění situaci.

V některých případech se může stát, že třída již nemá poslání rodit instance (jak tomu bylo doposud), ale stává se pouze předlohou pro dědičnost a rodit z ní instance ani nemá smysl. Takové třídě budeme říkat abstraktní. Její označení v diagramu je skloněným řezem písma (kurzíva, Italic). Třídě, která není abstraktní, tj. té, ze které má smysl rodit instance, se říká konkrétní.

Poznámka: *V UML však takový pojem není zaveden, třída má v UML vlastnost „IsAbstract“ typu boolean, přičemž true znamená „je abstraktní“.*

Chybný postup „Metoda generalizačního buldozeru“

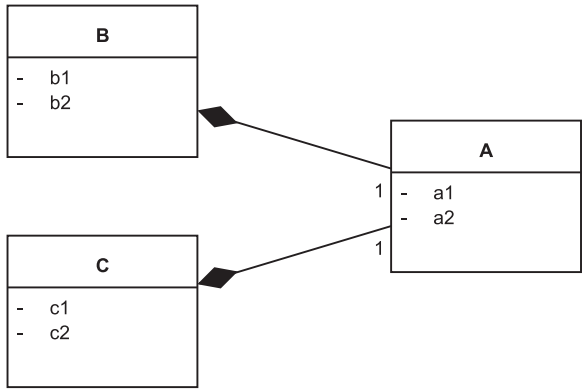
Když si na obrázku 3.25 prohlédneme, jak vlastně vypadají struktury instancí ze třídy C a B, mohlo by se zdát, že vodítkem pro vyhledávání generalizace je právě shoda stejných struktur. Bohužel to je častý omyl a je to nejběžnější chyba při vyhledávání generalizace.

Tuto chybnou metodu vyhledávání generalizace bych nazval „Metodou generalizačního buldozeru“ a vřele ji nedoporučuji. Uvedená nedoporučovaná metoda spočívá v tomto postupu:

Hledáme třídy a jdeme podle doporučení přes instance. Nalezneme pomocí rozhovoru se zákazníkem strukturu instance B1 jako (a1, a2, b1, b2). Třídy nyní ještě neznáme a zkusíme jako řešení zavést třídu B, která bude definovat tyto atributy. Poté hovoříme o instanci C1 a zjistíme, že instance má strukturu (a1, a2, c1, c2). Zavedeme tedy třídu C, která zavádí tyto atributy, a přitom si ověříme, že analyticky je a1 a a2 v obou případech totéž. To samozřejmě musíme vyřešit. Rozhodneme se tedy o vytknutí společného a zavedeme proto společného předka A, do nějž dáme společnou strukturu. Dostaneme tak zmíněný obrázek 3.25 s generalizací, kde A je předek. Vypadá to sice jako postup pěkně, když jsme dostali stejný obrázek. Ale v úvaze je bohužel hrubá chyba!

Z obrázku 3.25 znázorňujícím vztah *Generalizace* správně vyplývá, že pokud vznikne instance ze třídy B, potom má vlastnosti (a1, a2, b1, b2) a stejně tak relevantně instance z třídy C má vlastnosti (a1, a2, c1, c2). Z toho ale vůbec neplyne opačná implikace: Jestli totiž nalezneme, že instance B1 „umí“ atributy (a1, a2, b1, b2) a instance C1 „umí“ (a1, a2, c1, c2), tak z toho ještě neplyne, že musíme použít generalizaci, abychom dosáhli tohoto efektu. V mnoha případech (a dokonce ve většině!) se „to společné“ vyřeší pomocí skládání

instancí, tedy to společné se vloží do instance, která se kompozicí vloží do prvku B i C. Vznikne tak jiné a mnohdy lepší řešení, které řeší „to společné (a1, a2)“, viz obrázek 3.26.



Obrázek 3.26: Možné a mnohdy lepší řešení společné části informace A.

V této chvíli je dobré vzpomenout si na kapitolu objektového paradigmatu a na „vnější a vnitřní pohled“ na prvky. Představme si, že držíme v ruce instanci ze třídy B anebo instanci ze třídy C, viz obrázek 3.26. Všimněme si, že z vnějšího pohledu můžeme instanci ze třídy B nebo instanci ze třídy C požádat o údaje (a1, a2), přičemž tyto instance budou tyto údaje umět vydat díky kompozici ku jedné. Z vnějšího pohledu tyto instance „umějí“ (tedy zvně obsahují) tuto informaci, vnitřně to dělají pomocí „vloženiny“ instance ze třídy A. Zapsáno ve struktuře už B1 není (a1, a2, b1, b2), ale B1 = (b1, b2, A1 = (a1, a2)).

Jako příklad bych uvedl následující úvahu: Pokud evidovaný prvek typu X obsahuje IČ a evidovaný prvek jiného typu Y také obsahuje IČ (a jsou to dva prvky z různých tříd), tak to vůbec neznamená, že musí mít třídy pro X a pro Y společného předka. IČ lze totiž do obou prvků vložit jako kompozit ku jedné.

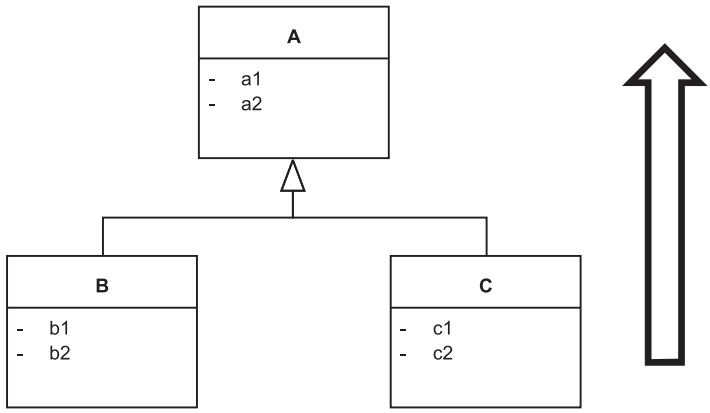
Z uvedeného vyplývá důležité doporučení: Nalezená shoda struktury uvnitř instancí nemusí vést ke vztahu *Generalizace* a většinou také nevede! Otázkou tedy je, co vede k nalezení generalizace a jaký je postup?

Zástupnost pojmů v generalizaci aneb jak se správně hledá *Generalizace*

Ke správnému nalezení generalizace slouží jiný postup založený na hlavní vlastnosti generalizace a tou je tak zvaná „zástupnost rolí neboli pojmů zespodu nahoru“. Namalujeme si ji jako pomyslnou šipku, viz obrázek 3.27.

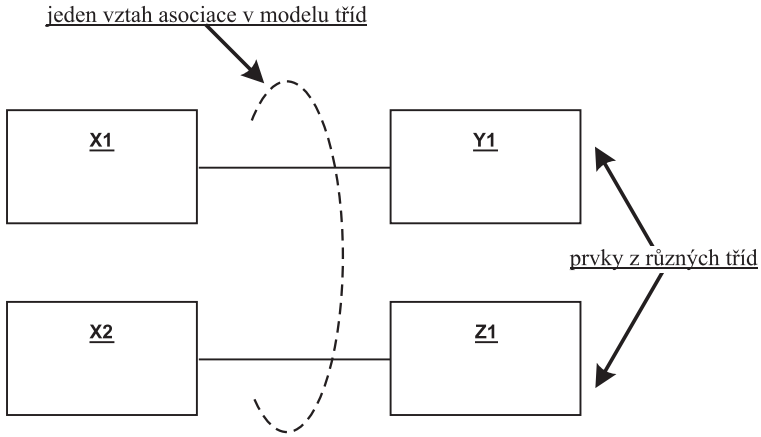
> *Základní vlastností generalizace je skutečnost, že všude, kde se na konci asociace objeví třída předka, může v této pozici v instancích sehrát roli libovolná instance z konkrétního potomka, protože umí tuto roli díky dědičnosti sehrát (tj. umí se vložit do konce asociace na místě svého předka). Obecnější pojem předka se tak stává zástupným pojmem pro libovolného konkrétního potomka, který může tuto roli díky generalizaci sehrát.*

Prakticky se tato vlastnost projeví v *Instančním diagramu* v efektu, který bychom mohli nazvat *rodina heterogenních linků*. Když tato situace nastane, tak v instanční rovině zřetelně vidíme několik linků mezi instancemi, přitom všechny tyto linky patří pod jednu asociaci (mají totiž analyticky stejný význam), ale přitom zjistíme, že tyto linky mají na svých



Obrázek 3.27: Zástupnost rolí a pojmů zespodu nahoru.

koncích instance z různých tříd. Tuto situaci si můžeme názorně představit například na obrázku 3.28.



Obrázek 3.28: Rodina heterogenních linků.

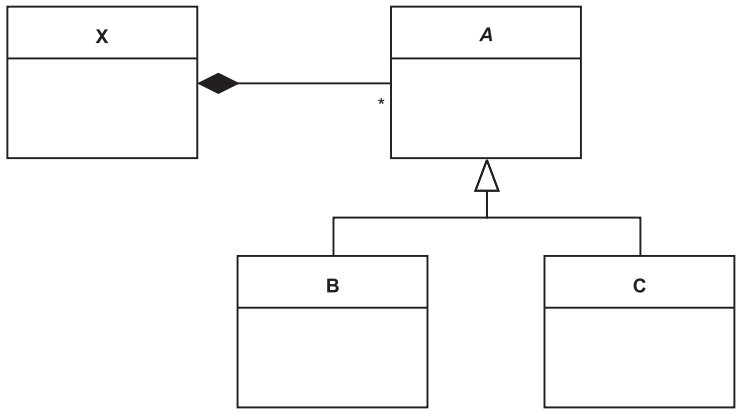
Obrázek 3.28 znázorňuje situaci, kdy jsme zřetelně identifikovali dva linky mezi evidovanými instancemi, jeden mezi instancemi X1 a Y1, druhý mezi instancemi X2 a Z1. Přitom však zjišťujeme, že oba linky by měly spadat v modelu tříd pod jednu asociaci, tj. jedná se o jeden vztah (například úvěr má ručitele apod.), ale prvky Y1 a Z1 patří do různých tříd.

Je zřejmé, že takovýchto situací, kdy na jedné straně linků jsou instance z různých tříd a přitom tyto linky patří pod jeden vztah, může být obecně nepřeberné množství. Prakticky se však heterogenní link vyskytuje zejména ve dvou základních nejčastěji používaných analytických vzorech, které si nyní vysvětlíme.

Vzor *Heterogenní seznam*

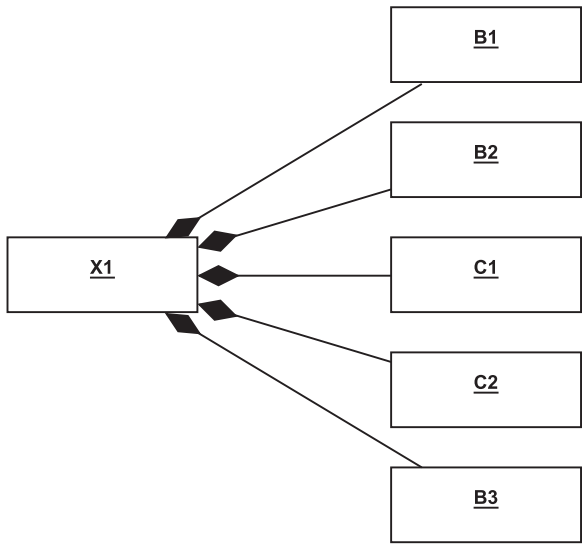
Vzor si vyjádříme pomocí obrázku 3.29.

Jedná se vlastně o kombinaci vzoru *Faktura má řádky* (kompozice ku N) a vztahu *Generalizace*. Všimněte si, že na straně kompozice ku N je jako třída „vlastněného“ uvedena abstraktní třída A. Z ní nikdy nebudou vznikat instance. Třídy B a C již nejsou abstraktní,



Obrázek 3.29: Vzor *Heterogenní seznam*.

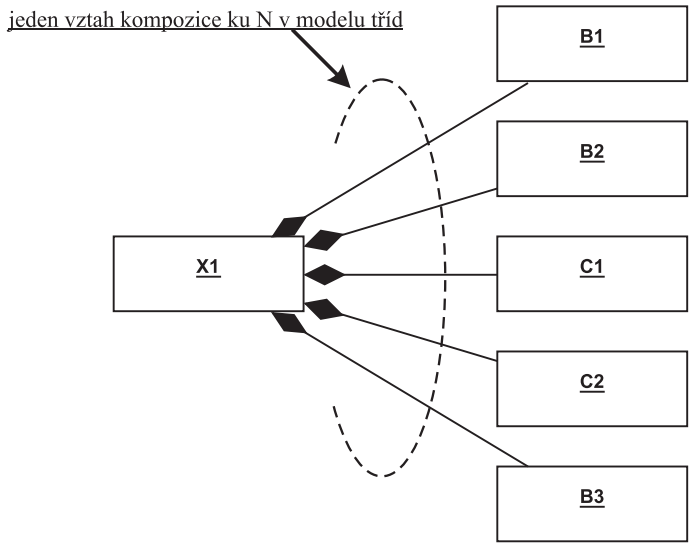
jsou tedy konkrétní. Nyní je dobré představit si namalovanou velkou šipku ve vlastnosti zástupnosti rolí zespodu nahoru. Ta funguje tak, že instance ze tříd B a C vstupují do vztahů linků z kompozice na straně vlastněných (jako by se vtělovaly do pozice v třídě A na konci kompozice) a jsou to tedy ony, kdo tento vztah v instancích realizují. Možný instanční model by mohl tedy vypadat například jako na obrázku 3.30.



Obrázek 3.30: Možné prvky vzoru *Heterogenní seznam* v instancích.

Všimněte si, že prvek X1 drží v seznamu instance z různých tříd (tedy z B i z C), ale přitom všechny linky v tomto vztahu jsou chápány jako instanční realizace jednoho jediného vztahu *Kompozice ku N* v modelu tříd. Tuto situaci bychom pro vysvětlení mohli znázornit pomocí obrázku 3.31.

Uvedená situace je umožněna zástupností rolí zespodu nahoru. Díky ní mohou být v jednom seznamu prvky různých typů, a proto takové situaci říkáme *rodina heterogenních linků*.



Obrázek 3.31: Linky do prvků z různých tříd a přitom jeden vztah v modelu tříd.

Mimochodem laik předešlou situaci chápe dobře, jenom ji vyjadřuje protimluvem. Diagram vzoru *Heterogenní seznam* (viz obrázek 3.29) má toto laické slovní vyjádření: „X drží v kompozici N prvků typu A, což mohou být prvky typu B nebo C.“.

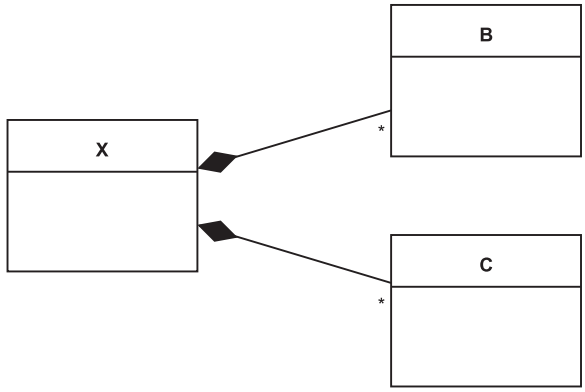
Příklady: Klient má N bankovních služeb, což mohou být buď úvěry anebo termínované vklady, anebo: Zpracuje se seznam dokladů, což jsou faktury nebo objednávky. Všimněme si, že v předešlých „laických“ větách je řeč vlastně o jednom typu, což je „nadtyp“ (například bankovní služba anebo doklad), ale přitom máme na mysli konkrétní podtypy, což jsou konkrétní potomci.

Pro odhalení vztahu *Generalizace* obecně (například konkrétně ve vzoru *Heterogenní seznam*) je právě vrchní zástupný pojem reprezentovaný pojmem nadtypu velmi důležitý a právě tato vlastnost tvoří podstatu vztahu *Generalizace*, což si nyní vysvětlíme podrobněji.

Podle doporučení z předešlých kapitol vyjděme z modelu instancí. Nechť tedy neznáme řešení v modelu tříd a teprve jej hledáme. Nalezli jsme vztah mezi instancemi, například tak, jak je uvedeno na obrázku 3.30. Tento model instancí má v modelu tříd dvě možná řešení, tedy do modelu tříd jej lze zobecnit dvěma možnými způsoby. Je na analytikovi, aby zvolil to řešení, které je pro danou situaci lepší.

První způsob řešení nevyužije generalizaci. Uvedené linky kompozitů od X1 k prvkům z B a C se chápou jako linky ke *dvěma seznamům*, jeden je seznamem prvků ze třídy B a druhý je seznamem prvků ze třídy C. Znamená to, že linky z X1 do B1, B2 a B3 bychom chápali jako linky jednoho vztahu *Kompozice ku N* a ostatní linky z X1 do C1 a C2 bychom chápali jako druhou skupinu linků, které realizují druhý vztah *Kompozice ku N*. Pak by zobecnění do modelu tříd a tedy navržený model tříd vypadal jako na obrázku 3.32. V tomto „klasickém“ řešení jsme nepoužili generalizaci.

Díky vztahu *Generalizace* máme i druhou možnost, jak zobecnit předešlý příklad s instancemi do modelu tříd. Musíme k tomu ale zavést „zástupný“, tedy vyšší, neboli obecnější pojem, který je „nadtypem“ jak pro B, tak pro C. Jinak řečeno, zvolíme řešení pomocí vzoru *Heterogenní seznam*, viz obrázek 3.29. Všimněte si, že instanciováním tohoto modelu dostaneme stejný obrázek vztahu instancí, ale v tomto případě se nejedná o dva vedle sebe stojící seznamy, ale o jeden seznam z A, do nějž se mohou „schovat“ (vstupovat do něj) instance



Obrázek 3.32: Dva seznamy bez použití vztahu *Generalizace*.

jak ze třídy B, tak ze třídy C. Výsledek je strukturou instancí stejný, ale nehovoříme o dvou, ale o pouze o jednom seznamu.

Otázkou tedy je, kdy zvolit řešení „dva seznamy“ a kdy „jeden heterogenní se zástupným pojmem“? Jedná se o rozhodnutí již na analytické úrovni! Pro volbu řešení „s generalizací nebo bez ní“ musíme posoudit následující skutečnosti:

Pokud se u pojmů B a C jedná skutečně o dva pojmově nezávislé seznamy a nikdy se nehovoří o nich jako o prvcích se společným názvem, tak zavedeme opravdu dva nezávislé seznamy.

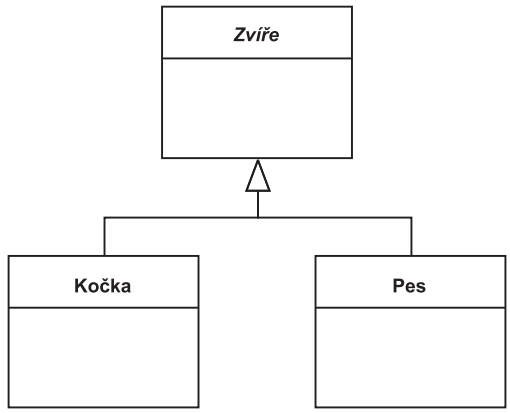
Jakmile však zjistíme, že dochází pojmově k zobecnění, tedy ke generalizaci, a vzniká nový zástupný pojem pro všechny pojmy obou seznamů (tedy začneme hovořit o společném pojmu pro oba subtypy), a tedy hovoříme o jednom seznamu, potom bychom měli přijmout řešení s generalizací. To znamená, že pokud se začne hovořit o „všech A“ anebo o „seznamu z A“ anebo „pro každé A“ apod., ale přitom se má na mysli konkrétně B nebo C (případně další pojmy-potomci), potom má třída A právo na život a měli bychom využít vztah *Generalizace*.

Jestli se nyní domníváme, že to možná nepoznáme, pak se určitě mýlíme: Poznáme to, protože to uslyšíme a je to normální logický postup. Samotný laický pohled na věc používá generalizaci, jenom se k vyjádření nepoužívají třídy a zápis v jazyce UML, ale slovní vyjádření.

Uveďme si klasický příklad ze života. Poslouchejme dobře tyto věty: Na dvorku pobíhají různá zvířata, což jsou kočky a psi. Každé zvíře umí na povel vydat zvuk, ale každý to dělá podle svého typu. Vyhnali jsme všechna zvířata ze dvorku, atd. Existují dvě možná řešení v modelu tříd pro tento „laický příklad“: Buď na dvorku zavedeme dva seznamy, tj. oddělené dva seznamy, jeden seznam koček a druhý seznam psů. To by odpovídalo řešení bez vztahu *Generalizace*. Druhou možností je přijmout řešení zavedením vztahu *Generalizace* (a zde zřetelně lepší) podle vzoru *Heterogenní seznam*. Zavedeme tedy generalizaci, viz obrázek 3.33.

Toto řešení s generalizací je určitě lepší, protože analytické vyjádření „zvíře“ je zde zřetelně patrné a hovoří se o něm v předešlých větách.

Jenom musím upozornit na jednu záludnou chybu, které se laik může dopustit. Pokud totiž v generalizaci v UML pracujeme s pojmem nadtypu (zde Zvíře), tak je to pouze zástupný pojem pro konkrétní instance některého z podtypů. Třída Zvíře není sama o sobě instanciovatelná, i když to tak ve větách na první pohled vypadá. Když tedy řeknu: „Na dvorku jsou zvířata“, tak to neznamená, že vidím instance ze třídy Zvíře (ta je abstraktní a nemá tedy instance). Vzhledem k modelu na obrázku 3.33 to znamená, že konkrétně vidím



Obrázek 3.33: Generalizace Zvíře, Kočka, Pes.

nějaké kočky a nějaké psy a nazývám je společným pojmem zvířata. Je dobré si představit, že pokud se řekne „zvíře“, tak je to instančně konkrétně buď kočka nebo pes (nikoliv samo zvíře), resp. do budoucna něco dalšího (potomek), co umí být zvířetem. Jedná se o typově speciálnější pojmy, tedy o třídy potomků v generalizaci. Pokud tedy na dvorku vidíme například konkrétně instanci psa, máme pro něj pojmově dvě vyjádření: Je to Pes a současně i Zvíře. Podobně bychom takovéto dvojí pojmové vyjádření vyslovili i pro instanci kočky, je to Kočka a současně i Zvíře.

Na tomto vysvětlení je zřetelně vidět, že řešení s generalizací má jednu velkou výhodu: Pokud analytik řeší problém popisu systému a hovoří v ní o zvířeti anebo zvířatech, pak přidání podtypu nemá na tyto analytické věty žádný vliv, protože jsou platné pro všechny podtypy. Pokud je tedy vhodné použít pojem „zvíře“, potom je jedno, jaký konkrétní podtyp máme v té chvíli na mysli, protože řeč je o obecnějším zvířeti. Plyne z toho, že přidání dalšího podtypu zvířete na dvorek (třebas typ Kachna) nemá již na tyto věty takto zapsané vliv a nebudou se tedy měnit.

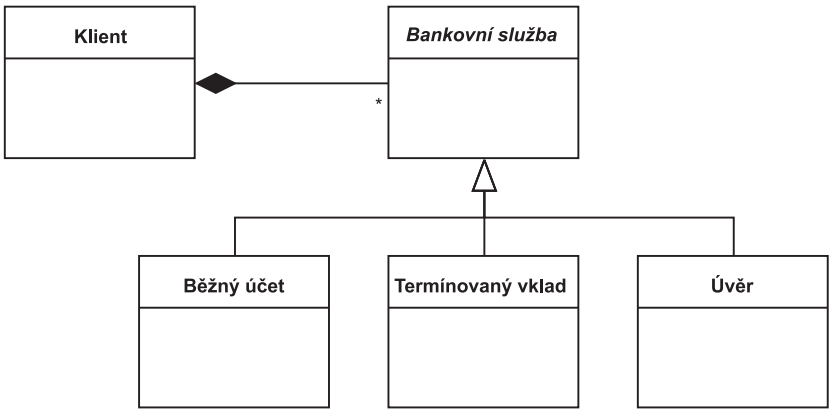
Jiná situace by nastala, pokud bychom použili řešení klasické a tedy bez generalizace. V tom případě by každý typ, zde kočka a pes, vytvářel svůj samostatný seznam. Vznikly by tak dva vedle sebe stojící seznamy, jeden seznam z koček a druhý seznam ze psů. To, co jsme v prvním případě jednoduše vyjádřili hromadným a obecnějším zpracováním přes pojem zvíře, je již nepoužitelné, protože jsme zavedli konkrétní seznam z koček a vedle něj stojící druhý konkrétní seznam ze psů. Proto nyní musíme pokaždé obsloužit oba seznamy zvlášť. Navíc je zřejmé, že přidání nového podtypu znamená přidat nový seznam a řešit jej jako nový seznam na dvorku.

Jako další příklad použití vzoru *Heterogenní seznam* bychom mohli uvést část systému evidujícího bankovní služby (obrázek 3.34).

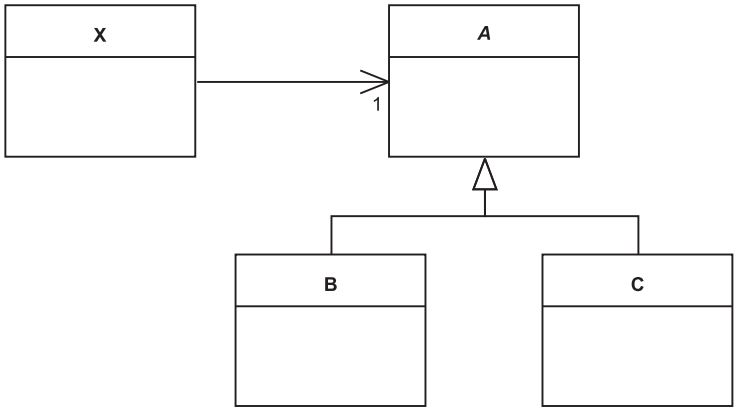
Vzor *Odkaz na vrchol stromu*

Druhou velmi často se opakující situací, kde je vhodné použití generalizace, je řešení pomocí vzoru *Odkaz na vrchol stromu*, viz obrázek 3.35. Jedná se o kombinaci vztahů *Odkaz do seznamu* (číselníkové vazby) a *Generalizace*.

V tomto případě se jedná o analytickou situaci, kdy z X vede odkaz na prvek, který se sice pojmově nazývá A, ale může se v něm konkrétně v instancích vyskytovat buď prvek



Obrázek 3.34: Klient má N bankovních služeb.



Obrázek 3.35: Vzor *Odkaz na vrchol stromu*.

typu B anebo prvek typu C. Například instance X1 se odkazuje na prvek B1, ale prvek X2 se odkazuje na prvek C1, přičemž oba tyto odkazy se chápou jako realizace téhož vztahu.

Jako příklad použití vzoru *Odkaz na vrchol stromu* si můžeme uvést následující větu z analytického modelování: Ručitel úvěru může být buď fyzická osoba anebo právnická osoba. Všimněme si, že také zde vzniká rodina heterogenních linků: V jedné instanci úvěru vede například odkaz na instanci typu fyzická osoba a v druhé instanci úvěru vede odkaz téhož typu na instanci právnická osoba, tedy na instanci jiného typu, než u předešlé instance, a důležité je, že přitom oba tyto odkazy v obou úvěrech reprezentují totéž, tj. odkaz na ručitele úvěru.

Je zajímavé uvést si obdobu řešení, kde se vyhneme použití vztahu *Generalizace*. V tom případě by v úvěru byly umístěny dva disjunktní odkazy do dvou seznamů, jeden do seznamu fyzických osob a druhý do seznamu právnických osob a mezi nimi by platil vztah XOR – je vyplněn právě jeden z nich.

Vyjmenovali jsme si sice dva základní a opravdu nejčastěji používané vzory s využitím generalizace, *Heterogenní seznam* a *Odkaz na vrchol stromu*, ale je zřejmé, že další všemožné situace použití generalizace můžeme dostat libovolnou kombinací asociace z předešlých kapitol s generalizací. Tyto situace se vyskytují již méně často, ale musíme s nimi počítat.

Vzor *Diskriminátor*

V souvislosti se vztahem *Generalizace* je vhodné si uvést další doporučený postup zvaný vzor *Diskriminátor*. Jedná se o doporučení (tj. nikoliv povinný postup), avšak považuji jej za opravdu velmi dobré doporučení. Vzor *Diskriminátor* má několik stupňů, záleží na autorovi, na kterém z nich se zastaví a jak tedy vzor použije.

1. stupeň vzoru *Diskriminátor*

Doporučuje se, aby byla ke každému stromu dědičnosti přidána navíc dohoda, podle které se jednoznačně „označí“ konkrétní třídy nějakým kódem. Tato dohoda je součástí analytické i technologické dokumentace a je zásadně již dále neměnná. Každá konkrétní třída ve stromu dědičnosti tak získává nejenom svůj název, ale také má v rámci daného stromu dědičnosti přiřazen navíc jednoznačný a dále již neměnný kód (může to být neměnné číslo nebo řetězec).

Laicky řečeno, tento vzor doporučuje „očíslovat si“ ve stromu dědičnosti jednoznačně a již neměnným způsobem konkrétní třídy. Pokud se například změní název třídy (což se může promítnout až do technologie), kód třídy se již nezmění. Tato dohoda je dobrá například pro případ, kdy se řeší převodník „výběr typu“ algoritmem (třebas při načítání dat) a spuštění scénáře pro zrod instance tohoto typu apod.

Poznámka: *Toto „okódování“ tříd je používáno také v Design Patterns (GOF), například ve vzoru* Prototype, Strategy, Interpreter *aj. To však není předmětem této knihy.*

2. stupeň vzoru *Diskriminátor*

Dále se doporučuje, aby se tato dohoda „okódování konkrétních tříd“ umístila do jednoduchého číselníku, který obsahuje tento kód a text. Text se zobrazuje uživateli například při výběru scénáře zrodu instancí (založení nové bankovní služby apod.). V našem příkladu Zvíře, Kočka, Pes by takový číselník mohl vypadat takto:

```
Kód Text
---------
1   Kočka
2   Pes
```

Zavedení takového „číselníku typů“ řeší například následující analytický problém scénáře: *Obsluze se zobrazí seznam typů zvířat, obsluha vybere typ zvířete a dále se provede podle typu...*

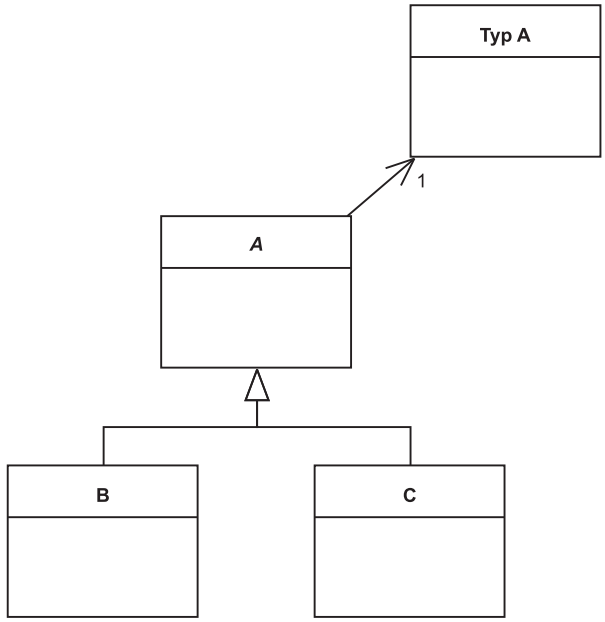
Číselník typů je zde synonymum pro zavedený *Diskriminátor* a u konkrétního stromu mu dáme název *Typ <A>*, kde A je název horní třídy (například Typ zvířete apod.). *Diskriminátor* tedy chápeme jako jednoduchý číselník, který „okóduje“ konkrétní třídy ve stromu dědičnosti, má své vyjádření jako zavedený implementovaný číselník (tzv. code list).

3. stupeň vzoru *Diskriminátor*

U evidenčních systémů (nikoliv technologických) se doporučuje, aby se strom dědičnosti nejenom dovybavil číselníkem *Diskriminátor* (viz 2. stupeň vzoru), ale také aby každá instance měla odkaz na odpovídající item číselníku, ze které třídy pochází, tedy řešení je navrženo tak, jak ukazuje obrázek 3.36.

Z uvedeného vyplývá, že každá instance nejenže pochází z nějaké třídy, ale současně její odkaz do číselníku udává „hodnotově“ odkazem do seznamu, co je vlastně zač, z jaké třídy pochází.

Sledujte aktuální novinky na http://www.objects.cz



Obrázek 3.36: Vzor *Diskriminátor* u evidenčních systémů.

Mapování generalizace do relační databáze

Existují tři základní způsoby mapování do RDB, které mají ještě svoje další varianty.

Mapování generalizace do RDB jako „1:1“

Vyjděme z analytického modelu, který je odevzdán jako zadání do technologie RDB (obrázek 3.24). Na úrovni designu relační databáze se zavedou 3 tabulky, pro každou analytickou třídu po jedné. Podle analytického modelu vzniknou mapováním tři tabulky TA, TB a TC. Primární klíč z tabulky TA (například `IdA`) se objeví jako cizí klíč v tabulkách potomků a přes něj se provede vazba, viz obrázek 3.37.

Jedna vyšrafovaná dvojice záznamů z tabulky předka a potomka provázaná přes klíč je jedna instance, první dvojice reprezentuje instanci ze třídy B, a druhá dvojice záznamů znázorňuje instanci ze třídy C. Atributy zavedené ve všech třídách se samozřejmě objeví jako sloupce těchto tabulek.

Existuje druhá varianta tohoto mapování, kdy se jako cizí klíč neobjeví dole nový sloupec, ale jako cizí klíč vystupuje přímo primární klíč `IdA` převzatý shora. Jinak řečeno, celá struktura všech tří tabulek by měla jediný společný klíč `IdA`, který se ve spodních tabulkách chová současně jako primární klíč a také jako klíč cizí (obrázek 3.38).

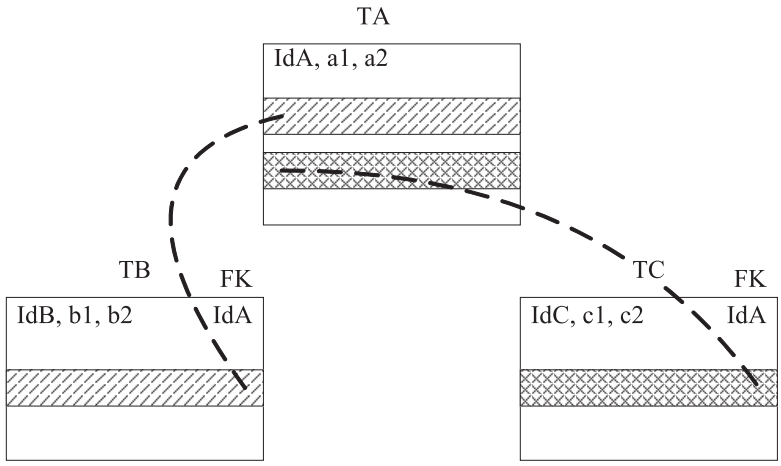
Poznámka: *V technologii Hibernate, která zavádí mapování z OOP do RDB, je obdobou tohoto mapování postup zvaný jako „table-per-subclass“.*

Mapování generalizace do RDB Kočkopsem

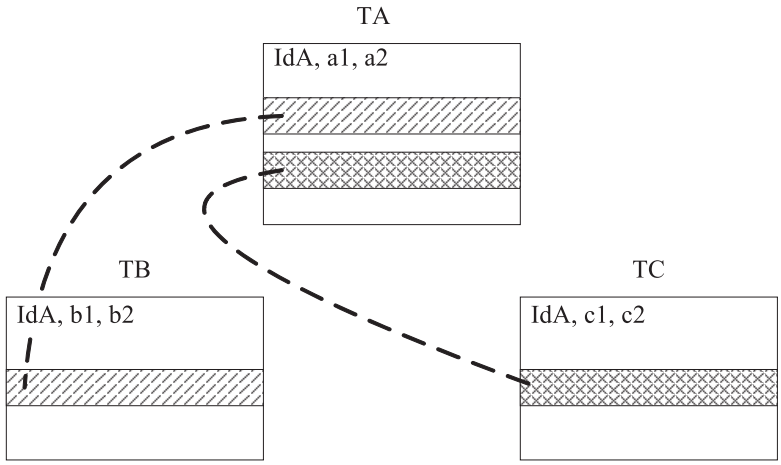
Poznámka: *V technologii Hibernate, kde se provádí mapování z OOP do RDB, je obdobou tohoto mapování postup zvaný jako „table-per-class hierachy“.*

Existují dvě varianty tohoto vzoru, z nichž druhou zásadně nedoporučuji.

Sledujte aktuální novinky na http://www.objects.cz



Obrázek 3.37: Mapování generalizace do relační databáze 1:1.

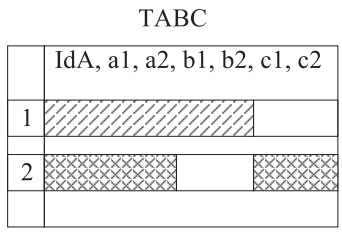


Obrázek 3.38: Mapování čisté 1:1 se společným klíčem ve všech tabulkách.

1. Kočkopes klasický

Při této variantě mapování nevzniknou tři tabulky, ale jen jedna, do které se umístí všechny tři namapované analytické třídy, viz obrázek 3.39. Namísto tabulek pro entity Zvíře, Kočka, Pes vznikne doslova a bez nadsázky „Kočkopes“, tedy „superdlouhá“ tabulka, do které se umístí všechny záznamy najednou vedle sebe (na obrázku 3.38 provedeme myšlenkové sloučení všech tří tabulek).

Sloučením záznamů do jedné tabulky vzniknou v každém záznamu „zajímavé“ a „nezajímavé“ (tedy irelevantní) oblasti sloupců: Například pro záznam instance z B jsou relevantní sloupce a1, a2, b1, b2, ale c1 a c2 nejsou relevantní. Stejně tak symetricky pro instance z C jsou zajímavé a1, a2, c1, c2, ale b1 a b2 nejsou zajímavé. K tomu, abychom věděli, se kterými sloupci máme vlastně pracovat, slouží jeden sloupec umístěný někde na začátku tabulky, což není nic jiného, než již známý kód diskriminátoru (význam viz předešlé kapitoly). Hodnota v tomto sloupci udává, se kterými sloupci se má pracovat (například 1 s B sloupci a 2 s C sloupci).



Obrázek 3.39: Mapování generalizace sloučením tabulek.

2. Kočkopes s přetížením sloupců
Jedná se o velmi nedoporučené mapování. V tomto případě se vychází z mapování kočkopsem, ale pokračuje se s myšlenkou slučování také u atributů. Uvedu kočkopsa s přetížením sloupců na příkladu, který jsem viděl v praxi a který vedl k totálnímu kolapsu vývoje tohoto systému.

Zaveďme abstraktní třídu Bankovní služba, která má (jako příklad) dva potomky Termínovaný vklad a Úvěr. Při mapování do kočkopsa vznikne jediná tabulka obsahující jak sloupce pro termínový vklad, tak pro úvěr. Autor databáze zjistil, že sloupec jistina (kolik se půjčilo v úvěru) je stejného typu MONEY jako počáteční vklad (kolik se vložilo na termínovaný vklad) a s potěšením zjistil, že tyto dva sloupce jsou disjunktní (což je samozřejmé, když se jedná o kočkopsa vzniklého původně generalizací!). Proto se rozhodl, že tyto dva sloupce sloučí do jednoho a hodnota diskriminátoru bude udávat, o co v sloupci jde. Tento sloučený sloupec se už nemůže jmenovat ani jistina a ani vklad a proto jej nazval PARMONEY1. Číslo na konci názvu není index, ale součást názvu, protože zjistil, že takovýchto sloupců typu MONEY potřebuje více. Vznikly tak sloupce PARMONEY1, PARMONEY2, PARMONEY3 až PARMONEY10. Význam sloupců je v tomto mapování dán hodnotou diskriminátoru. S potěšením zjistil, že totéž může provést i se sloupci typu DATE (datum). Zavedl tedy sloupce PARDATE1, PARDATE2, PARDATE3 až PARDATE10, a pak pokračoval dále s čísly a varchary atd.

Toto mapování vřele nedoporučuji. Ohodnocení slovy „nepřehledné a málo transparentní“ je ještě slabý výraz, systém je vlastně v názvech sloupců zašifrován. Možná je to jediná drobná (úsměvná) výhoda: Pokud databázi dostane nějakým způsobem do ruky konkurence, tak vůbec neví, která bije. Bohužel jsem si ověřil, že po několika měsících vývoje to neví ani sami autoři systému. Navíc zde nastává ještě jeden hrůzný jev: Pokud dojde v nějakém procesu (například zpracování splátek úvěrů) k chybě a na začátku procesu se zvolí jiná hodnota diskriminátoru, tak proces proběhne nad „špatnými“ instancemi (například termínovanými vklady). Tato chyba samozřejmě u čistého mapování anebo klasickým „Kočkopsem“ bez přetížení sloupců není možná, protože sloupce mají své specifické názvy a jsou proto od sebe odlišeny.

Mapování generalizace do RDB rozpuštěním předka
Poznámka: *V technologii Hibernate, kde se provádí mapování z OOP do RDB, je obdobou tohoto mapování postup zvaný jako „table-per-concrete class“.*

Toto mapování je velmi jednoduché a stačí jej proto popsat slovy. Tři analytické třídy A, B, C se namapují tak, že tabulka předka se rozpustí do svých potomků a jeho struktura se zopakuje ve všech potomcích. Vzniknou tak tabulky TAB a TAC, kde pole a1, a2 se zopakují v obou tabulkách.

3.3 Další vzory v *Class diagramu*

Připomeňme si nejprve vzory, které jsme již v analytickém *Class Diagramu* probrali.

Vzor *Dichotomie třída-instance*
Informační systém je rozdělen do dvou prostorů, prostor tříd a prostor instancí. Třídy jsou meta-pravidlem pro budoucí instance.

Vzor *Flexibilní test*
U aplikace je třeba dobře určit úroveň meta, tedy co bude třídou a co bude instancí. Řešení by nemělo „uhnout“ do jiné úrovně meta, než je obchodně a analyticky žádoucí.

Vzor *Osoba má adresu* (jako v e-shopu)
Vyjadřuje kompozici ku jedné. Každá osoba má adresu v kompozici dané role.

Vzor *Faktura má řádky faktury*
Vyjadřuje kompozici ku N. Prvek má svoje položky.

Vzor *Auto má barvu*
Prvek si odkazuje na jiný prvek ze seznamu prvků, slangově číselníková vazba.

Vzor *GSM telefon má SIM-kartu*
Neboli *Sdílená agregace*. Prvek je sice vlastněn, ale vlastnictví není až tak otrocké. Je narušena alespoň jedna z nutných podmínek *Kompozice*, kdy vlastněná instance může žít dočasně sama anebo přeputovat anebo má vícero majitelů.

Vzor *Auto, Osoba, Vlastnictví*
Asociační třída se dvěma konci. Její prvky propojují dva nezávislé seznamy, které jinak o sobě neví.

Vzor *Učitel, Předmět, Místnost, Zkouška*
Trojná asociační třída, která namísto dvou propojuje svými instancemi tři nezávislé seznamy.

Vzor *Generalizace*
Generalizace je interakce mezi třídami, kdy jedna třída jako předpis (potomek) používá druhou třídu (předka). Uplatní se důležitá vlastnost zástupnost rolí zespodu nahoru.

Vzor *Heterogenní seznam*
Kombinace vzorů *Generalizace* a *Faktura má řádky faktury*. Prvek drží seznam prvků typu A, ale díky generalizaci do této pozice mohou vstupovat instance konkrétních potomků.

Vzor *Odkaz na vrchol stromu*
Kombinace vzorů *Generalizace* a *Odkaz do seznamu*. Prvek si odkazuje na jiný prvek typu A, ale díky generalizaci do této pozice mohou vstupovat instance z potomků.

Probereme si nyní další vzory.

Sledujte aktuální novinky na http://www.objects.cz

Vzor *Vlak má svoje vagóny*

Tento vzor je konkurenční k asociační třídě, tj. ke vzoru *Auto, Osoba, Vlastnictví*. Poprvé jsem se setkal s použitím tohoto vzoru v jedné české firmě, která vyvíjela systém pro evidenci vagónů a vlaků. Jednalo se o klasický systém dopravy pro velké firmy (např. SETUZA, SLOVNAFT, CEMENTÁRNY apod.). Tyto firmy si kromě nákladních automobilů vedou také evidenci vagónů a vlaků, jak svých, tak vagónů jiných firem. Každý vlak a stejně tak každý vagón, který projel přes jejich firemní nádraží, tzv. vlečku, se zaevidoval v informačním systému (POZN.: *vlečka je něco jako „soukromé malé vlakové nádraží" určené pro nakládku a vykládku z vagónů přímo v závodě*).

V této evidenci vznikl zdánlivý rozpor: Na jedné straně je zřejmé, že by bylo dobré zaevidovat každý vagón sám o sobě jako takový, tj. bylo by dobré, aby vznikl seznam všech evidovaných vagónů, které kdy projely vlečkou. Měl by tak vzniknout seznam zaevidovaných vagónů (slangově „číselník vagónů"), přičemž každý prvek tohoto seznamu (tj. evidovaný vagón) by měl nést své základní údaje jako své atributy, resp. násobné kompozity, například číslo vagónu (podle toho jej najdeme při znalosti z reálu v evidenci), jaký je to typ vagónu, nosnost, hmotnost, počet náprav atd. Měl by tak vzniknout seznam všech evidovaných vagónů, které se kdy objevily na vlečce firmy. Slangově programátorskou hantýrkou řečeno, vznikl by tak „číselník vagónů", ze kterého by se evidovaný vagón vybral a následně by se použil odkaz na něj (tj. byl by použit vzor *Odkaz do seznamu*).

Na straně druhé se žádá v systému evidovat také vlaky. Každý evidovaný vlak má evidované vagóny, tj. evidovaný vlak by měl „držet" evidované vagóny v agregaci. Tato úvaha je však v protimluvu s předešlou úvahou, kdy evidované vagóny by se měly vyskytovat v „číselníku vagónů" a mají tedy stát samostatně a nikoliv v nějaké agregaci vůči vlakům, když se nabízejí ze seznamu evidovaných vagónů (tj. z „číselníku vagónů") vzorem *Odkaz do seznamu*.

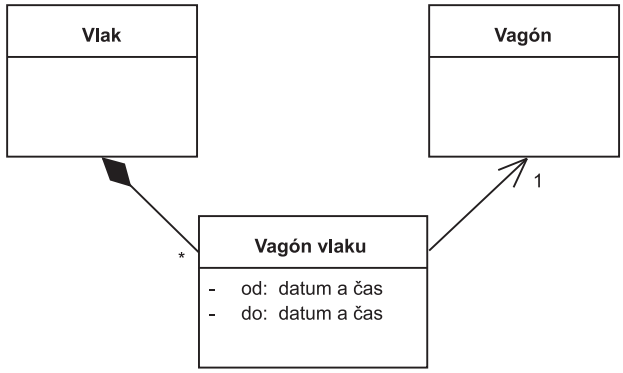
Zdánlivý rozpor spočívá ve velmi časté chybě analytika, kterou bychom mohli nazvat „chybou scházejícího pojmu". Kromě pojmu „evidovaný vagón" (což bylo zavedeno v předešlých odstavcích jako prvek z „číselníku vagónů") existuje ještě další pojem a tím je „evidovaný vagón vlaku". Jedná se o dva rozdílné pojmy: Evidovaný vagón je opravdu „čistý vagón" (chápáno nikoliv jako vagón čistý v realitě, ale jako evidovaný prvek, který nese údaje pouze o daném vagónu a nic víc), kdežto pojem evidovaný vagón vlaku odpovídá v představě zařazení vagónu do vlaku. Tento pojem může kromě toho, o který vagón z číselníku se jedná, nést ještě informaci datum a čas zařazení vagónu vlaku do vlaku, datum a čas vyřazení vagónu vlaku z vlaku apod.

Správná analytická věta tedy nezní „Vlak má vagóny", ale „Vlak má svoje vagóny". Znázorněme nyní tento pojmový rozdíl pomocí UML diagramu na obrázku 3.40. Dostaneme tak mimochodem i vyjádření tohoto vzoru.

Pro názornost jsme úmyslně umístili do prvku Vagón vlaku také atributy (zde dva atributy od, do typu „datum a čas"), aby vynikla povaha evidence, co vlastně nese. Všimněme si, že se jedná o kombinaci dvou analytických vztahů (základních analytických vzorů) *Kompozice ku N* a *Odkaz do seznamu* (slangově zvaný *Číselníková vazba*).

Signálem k nalezení této analytické situace jsou věty typu: „Něco má něco z něčeho", tj. „X má XY z Y". Typické situace, které by mohly být tímto vzorem řešeny, jsou například: „Student má zkoušky studenta ze seznamu zkoušek", „Osoba má adresy z číselníku adres" (i s historií stěhování!), „Třída má žáky třídy ze seznamu žáků školy" aj.

Sledujte aktuální novinky na http://www.objects.cz



Obrázek 3.40: Vzor *Vlak má svoje vagóny.*

Konkurenční řešení ke vzoru *Vlak má svoje vagóny*

Našli jsme řešení pomocí vzoru *Vlak má svoje vagóny.* Je třeba upozornit na to, že uvedená analytická situace umožňuje navrhnout hned několik řešení, tedy nabízí se použít také jiná „konkurenční řešení". Pokud mezi možná řešení zahrneme také „Vlak má svoje vagóny", potom bychom je mohli vyjmenovat takto:

1. Vlak má svoje vagóny;
2. Obrácený vztah téhož vzoru;
3. Použití asociační třídy;
4. Dvě sdílené agregace.

Vysvětleme si rozdíly v použití jak tohoto vzoru, tak i jeho konkurenčních řešení.

Vlak má svoje vagóny

V této variantě volíme v evidenci řešení, kdy ve vnitřní struktuře (tj. vnitřním pohledu) je v instanci vlaku implementována kolekce vagónů vlaku, tj. seznam z vagónů vlaku. Vyjádřeno pomocí objektového přístupu to znamená, že prvek instance vlaku má v sobě (tj. ve vnitřním pohledu) implementovánu svou kolekci a můžeme tedy přímo použít službu instance vlaku v evidenci například takto:

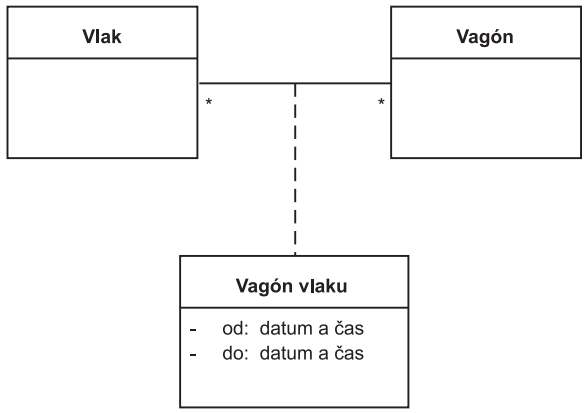
```
myVlak.GetVagonyVlaku(i).GetVagon();
```

Vzhledem k tomu, že podstatnou vlastností vlaku jsou jeho vagóny, není důvod toto řešení nepoužít. Jinak řečeno nevadí nám, že instance vlaku zná vagóny a „musí s nimi v kódu žít a ony s ním".

Asociační třída

V tomto případě volíme řešení „dva nezávislé seznamy" – slangově dva číselníky, jeden jako seznam Vlaků a druhý jako seznam Vagónů, ale ty jsou „bokem propojeny" asociační třídou tak, jak je uvedeno na obrázku 3.41.

Oba dva seznamy pro vlaky a vagóny jsou nezávislé v tom smyslu, že ve vnitřní struktuře ani vlaku a ani vagónu nenajdeme „nic o tom druhém". Můžeme také říci, že tyto dva seznamy jsou na sobě typově nezávislé. Vyplývá z toho, že by se dokonce daly oba seznamy



Obrázek 3.41: Použití asociační třídy.

implementovat do dvou nezávislých částí systému, tj. do dvou modulů jako skupina package v JAVA, assembly v .NET, unit v Pascalu, resp. library v C++. Třetí modul by tvořil kód obsluhující prvky z asociační třídy. V tomto řešení pomocí asociační třídy již nemůžeme požádat prvek `Myvlak` o kolekci `VagónVlaku`, protože ji nezná. Pro získání tohoto seznamu musíme jít do „centrální evidence“ například takto:

```
mojeVagony = gSeznamVlakVagon.GetVagonyVlaku(mujVlak);
```

Je dobré si uvědomit, že pokud je v pozadí relační databáze, ze které se plní objekty v aplikaci, potom SQL příkaz vyjde úplně stejně, pouze konstrukce v kódu je umístěna do jiných částí (do jiných tříd).

U vlaků a vagónů však není toto řešení nejvhodnější, protože vlak a vagóny nechápeme jako dva nezávislé seznamy propojené třetími prvky (vysvětlení viz předešlé řešení). Oproti tomu pro evidenci sňatků bychom zavedli asociační třídu, která by propojila dva seznamy mužů a žen a nepoužili bychom vzor *Vlak má svoje vagóny*, protože bychom tím „diskriminovali“ buď muže nebo ženy.

Obrácený vztah téhož vzoru

Je možné i další řešení vzoru *Vlak má svoje vagóny* v obráceném garde, ale předem upozorňuji, že bychom se k němu určitě nepřiklonili. V tom případě by se řešení znázornilo tak, jako na obrázku 3.42.

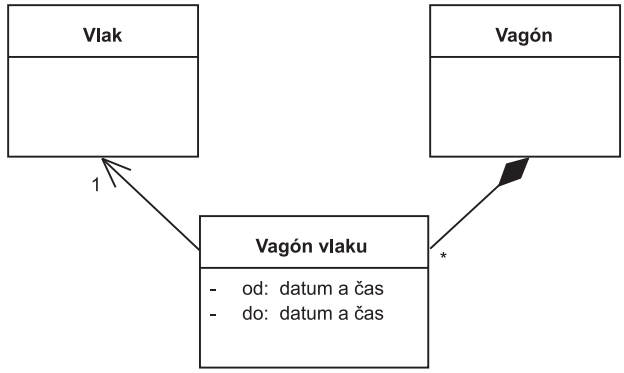
Každý vagón obsahuje kolekci z vlaků. V tomto řešení je to seznam vlaků, který se chová jako „čistý číselník“. Vagón již není „čistý číselník“, protože ve své vnitřní struktuře obsahuje seznam z Vagónů vlaků. K tomu řešení bychom se asi nepřiklonili, protože jako „čistý číselník“ chápeme seznam vagónů a nikoliv vlaků (vagón zde není v tomto smyslu „čistý číselník“, protože obsahuje svoje vlaky).

Řešení pomocí dvou sdílených agregací

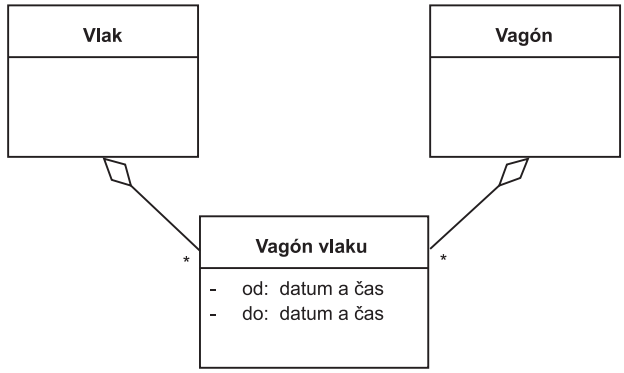
V tomto řešení jsou prvky chápány jako členy dvou kolekcí současně, viz obrázek 3.43.

V tomto případě jsou do obou prvků vlak a vagón umístěny sdílené prvky. Všechny třídy jsou spolu provázány a budou žít v jednom modulu.

Otázkou vždy je, které z řešení zvolit v daném konkrétním případě. Je třeba podotknout, že v mnoha případech není na tuto otázku zcela jednoznačná odpověď a případné spory mezi analytiky řeší hlavní analytik.



Obrázek 3.42: Obrácené použití vzoru.



Obrázek 3.43: Řešení pomocí dvou sdílených agregací.

Vzor *Bridging* neboli *Přemostění*

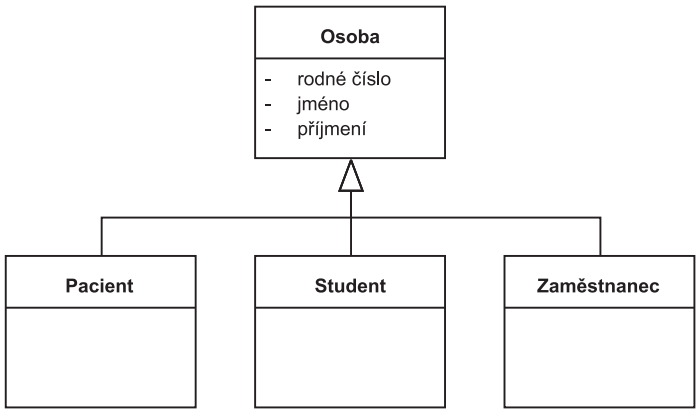
Tento vzor řeší chybné postupy při špatném použití dědičnosti. Vysvětlení vzoru si uvedeme příkladem:

Máme za úkol řešit evidenci osob ve fakultní nemocnici. Pro jednoduchost tohoto příkladu nejprve nebudeme uvažovat cizince (tato varianta se dá pomocí generalizace vyřešit také, ale není předmětem našeho příkladu, možnost této varianty bude uvedena na konci příkladu). Z rozhovorů se zákazníkem záhy zjistíme, že v evidenci mají figurovat různé typy osob. Jsou jimi Pacient, Zaměstnanec a Student. Ptáme se proto sami sebe: „Je žádoucí, aby Pacient, Zaměstnanec a Student uměl to, co Osoba, například rodné číslo, jméno a příjmení?“. Odpověď zní ano, určitě.

Rozhodneme se proto „naučit“ tyto typy (a z nich následně instance) tomu, co umí Osoba tak, že tyto typy budou potomky typu Osoba, jak je uvedeno na obrázku 3.44.

Pomocí tohoto řešení bude instance ze třídy Pacient „umět“ atributy rodné číslo, jméno a příjmení, protože pochází ze třídy, která je podědila. I když řešení na první pohled vypadá dobře (a dokonce se podobné varianty vyskytují v literatuře jako doporučená řešení), jedná se o řešení chybné. V tom právě spočívá největší záludnost tohoto chybného řešení.

Tato chyba je natolik záludná, že ji mnohdy nepoznáme přímo, ale až na základě projevů jejího důsledku. Ten je naštěstí velmi signifikantní a příznačný – najednou při návrhu pocítíme silné nutkání „křížit třídy“, tedy snažíme se o tvorbu tříd ve tvaru kříženců. Jednoduše zapamatovatelná poučka v tomto případě zní: Pokud při návrhu *Class Diagramu* začneme

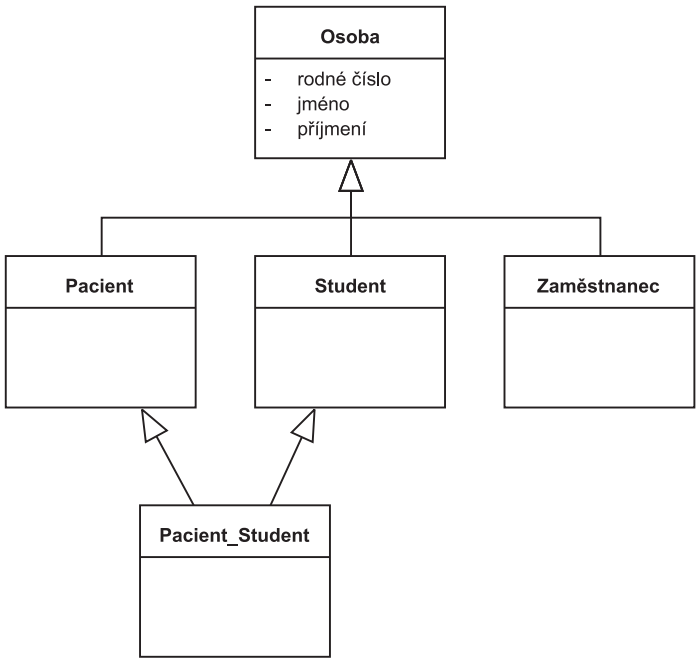


Obrázek 3.44: Chybně použitá dědičnost.

uvažovat o třídách-kříženců, zastavme úvahy a model přehodnoťme, protože s největší pravděpodobností obsahuje chyby.

V našem příkladu s evidovanými osobami v evidenčním systému fakultní nemocnice vede ke snaze o tvorbu kříženců jednoduchá otázka: A co když bude Pacient současně i Zaměstnancem? Anebo Student současně Zaměstnancem? Co s tím? Budeme nějak tvořit dvojkombinace anebo dokonce trojkombinace?

Ještě sice existuje jedna možnost dalšího postupu, použít tzv. vícenásobnou dědičnost, tedy podědit ze dvou předků (mimochodem jazyk UML to nezakazuje) způsobem znázorněným na obrázku 3.45.

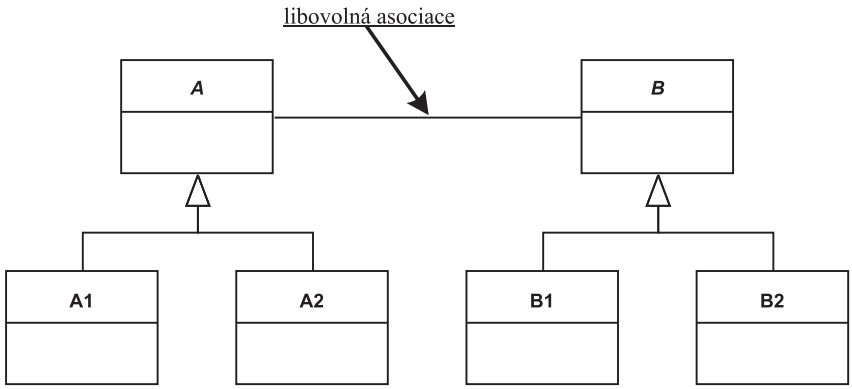


Obrázek 3.45: Pokus o křížení jako příznak chyby.

To by však znamenalo prokřížit všechny dědice. Všimněme si, že takovýchto kombinací je tam „nějak moc“. A to ještě netušíme, jak explozivní charakter má právě snaha o tvorbu kříženců! Představme si, že na druhý den přijde zákazník s dalším požadavkem: „Včera jsme o tom nehovořili, my tam máme i knihovnu a chceme evidovat Čtenáře...“. Napadne nás hned otázka: „A kdo všechno může být čtenářem?“. A tušíme nepříjemnou odpověď: „Každý, jak Student, tak Zaměstnanec, tak i Pacient...“. Ale pozor, navíc musíme do úvahy s křížením započítat i již existující kombinace, tj. prokřížit třídu Čtenáře s třídami již prokříženými, tedy s třídou Student_Zaměstnanec, Student_Pacient atd. Vidíme, že snaha o křížení tříd vede k efektu kombinatorního nárůstu dalších kříženců. Nyní je zřejmé, že toto řešení je špatné. Velmi silným příznakem chybně použité dědičnosti je právě efekt kombinatorní exploze subtříd v dědičnosti. Předešlou větu lze zformulovat i takto:

> *Při špatně použité dědičnosti kombinatorika hraje proti nám s efektem exploze subtříd, při správně použité dědičnosti hraje kombinatorika pro nás. V návrhu tříd s dědičností je třeba docílit takového stavu, aby kombinatorika hrála pro nás a nikoliv proti nám. Řešením je vzor* Přemostění.

Podstatou vzoru *Bridging* je řešení, ve kterém jsou zavedeny dva nebo více disjunktních stromů (které nevedou ke kombinatorice) a tyto stromy jsou propojené přes asociaci mezi vrcholovými třídami. Příklad použití tohoto vzoru pro dva stromy dědičnosti je uveden na obrázku 3.46.



Obrázek 3.46: Přemostění dvou stromů dědičnosti.

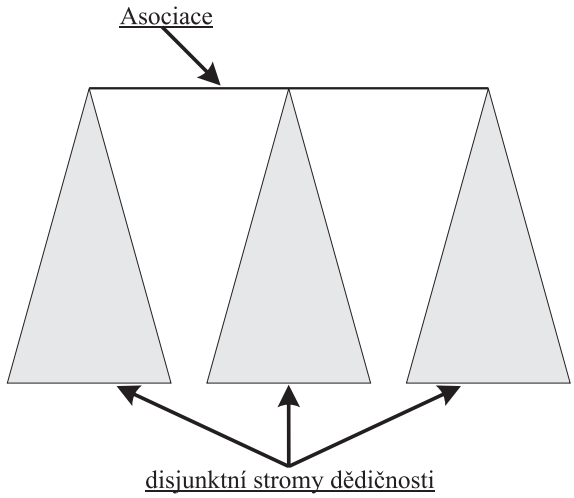
Všimněme si jednoho příznivého jevu: V tomto modelu kombinatorika již „hraje pro nás“, tj. v pozicích A nebo B se mohou nezávisle na druhém stromu vyskytovat dědicové z druhého stromu. Je to způsobeno zástupností rolí zespodu nahoru ve vztahu *Generalizace*. Právě díky tomu mohou vznikat všechny možné požadované kombinace automaticky bez zavádění dalších tříd-kříženců. K obrázku 3.46 je třeba dodat ještě upřesňující informace:

1. Uvedené stromy dědičnosti mohou být samozřejmě hlubší a širší, zde jsou uvedeny pouze s jednou úrovní dolů a s jednou úrovní šíře. Také stromů může být více než dva (viz dále).

2. Pod libovolnou asociací máme nyní na mysli asociace ve vzorech, jak jsme je brali v minulých kapitolách (*Odkaz do seznamu*, *Faktura má řádky* atd.).

3. Vzor jsme nazvali *Přemostění*, protože asociace připomíná most přes údolí, které je tvořeno dvěma vrcholy stromů dědičnosti.

Na obrázku 3.46 jsme pro přehlednost zvolili dva stromy dědičnosti, obecně však ve vzoru *Přemostění* může být stromů dědičnosti i více než dva, což můžeme znázornit obrázkem 3.47.

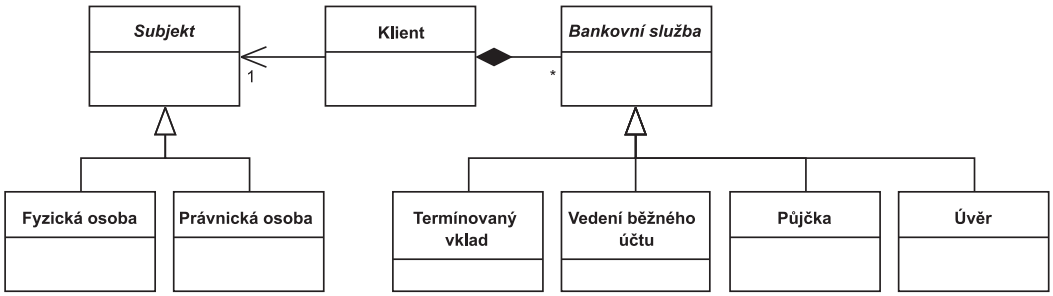


Obrázek 3.47: Zobecnění vzoru *Bridging* pro více stromů.

Na obrázku 3.47 šedé trojúhelníky symbolicky znázorňují stromy dědičnosti.

> *Při problému špatně použité generalizace se snažíme nalézt řešení pomocí několika disjunktních stromů, které propojíme přes vrcholové třídy některou z asociací.*

Navíc je třeba upozornit na zajímavou variantu tohoto vzoru, kdy některý ze stromů může mít triviální podobu „bez dědiců“, tedy existuje pouze vrchní „neabstraktní“ třída. Jako příklad si na obrázku 3.48 uveďme řešení s klienty v bankovním informačním systému.

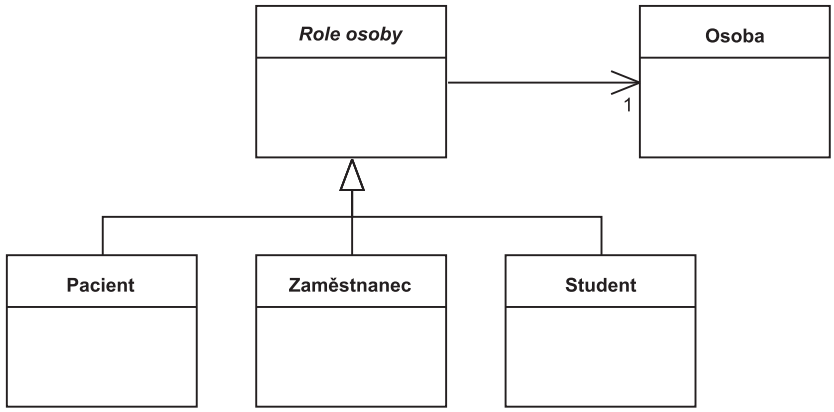


Obrázek 3.48: Tři propojené stromy s jedním stromem triviálním (Klient).

Tento model bychom mohli přečíst takto: *Klient má N Bankovních služeb různého typu a Klient je Subjektem různého typu.* Všimněme si, že v tomto případě hraje kombinatorika s námi, technicky tedy přicházejí v úvahu všechny možné kombinace všech dědiců podle pravidla „každý s každým“. Pokud chceme některou z kombinací zakázat, měli bychom se obrátit na povolené kombinace z *Diskriminátorů* (vzor *Diskriminátor* viz předešlé kapitoly).

Jaké je tedy řešení v našem příkladu s fakultní nemocnicí? V uvedeném špatném modelu, který vede ke křížencům, zřetelně chybí asociace, která by přemostila stromy. Pacient (a podobně Student, resp. další role osoby) však musí „umět Osobu“ (tj. držíme-li například

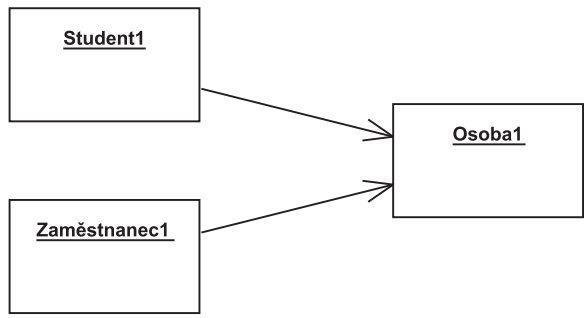
evidovaného Pacienta, měl by nám umět vydat „rodné číslo, jméno a příjmení“), ale nemůže ji umět díky generalizaci. Znamená to, že když tedy nebude vztah *Generalizací*, musí být v tom případě *Asociací*. V další úvaze musíme vybrat jednu z možných a tedy povolených asociací analytického modelu, které jsme brali. Je zřejmé, že kompozice nám nepomůže, protože to by byla cesta z deště pod okap: V kompozici platí „svůj k svému“ a my potřebujeme docílit shody osoby, pokud se jedná o tutéž osobu. Je třeba tedy zavést vztah, který vede ke sdílení prvků, zvolíme tedy vztah *Odkaz do seznamu*. Prvky ze tříd Pacient, Student atd., tedy budou „umět“ Osobu, ale nikoliv díky generalizaci, ale proto, že si na ni každý prvek bude odkazovat podle vzoru *Odkaz do seznamu*. Protože tuto vlastnost má mít každý prvek, můžeme zavést generalizaci pro společný odkaz do seznamu způsobem uvedeným na obrázku 3.49.



Obrázek 3.49: Řešení příkladu s přemostěním.

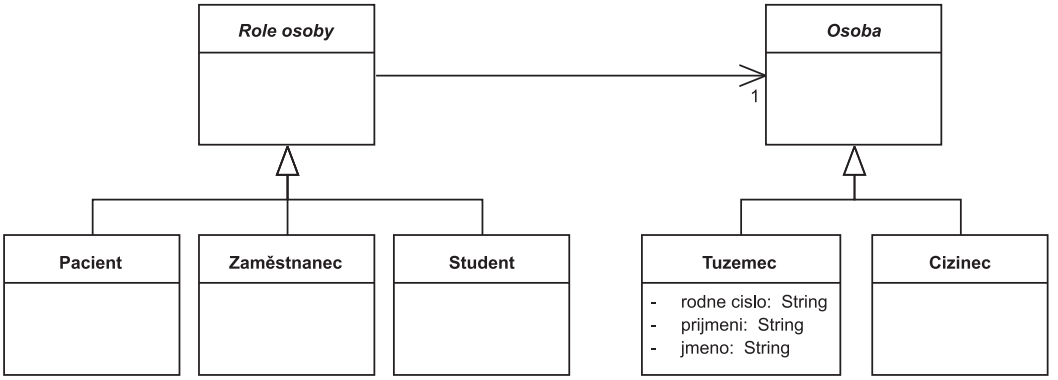
Je zřejmé, že přidání nového dědice do stromu dědičnosti již nepřináší žádný problém. Když zákazník přijde ještě se Čtenářem, tak žádný studený pot s dalšími kříženci, ale s úsměvem přidáme dalšího dědice.

Jak bylo řečeno, vždy je nutné v *Class Diagramu* vidět instance. V instanční rovině se budou rodit instance z dolních konkrétních tříd, nikoliv z třídy Role osoby, která je abstraktní. Každá instance z potomka má odkaz do seznamu osob. Pokud mají dvě instance odkaz na tutéž osobu, reprezentují tutéž osobu, což v instancích můžeme znázornit například obrázkem 3.50.



Obrázek 3.50: Příklad instancí z předešlého modelu tříd.

Když se podíváme na model tříd správného řešení s přemostěním (viz obrázek 3.49), vidíme, že vpravo sice není strom (jedná se o triviální strom s jednou třídou), ale dovedeme si představit situaci, kdy se strom rozroste a triviální již nebude. Mohlo by tomu být při požadavku evidovat i cizince. Potom by řešení vypadalo tak jako na obrázku 3.51.



Obrázek 3.51: Řešení doplněné o subtypy osob.

Poznámka: *Uvedený vzor* Bridging *je zobecněním vzoru* Bridge *známého z Design Patterns OOP (GOF). Ten řeší specifickou situaci v technologických systémech, kde podle stejného principu musí být oddělen pohled na systém v logické části aplikace a v technologické části aplikace, což vede ke dvěma disjunktním stromům propojeným asociací.*

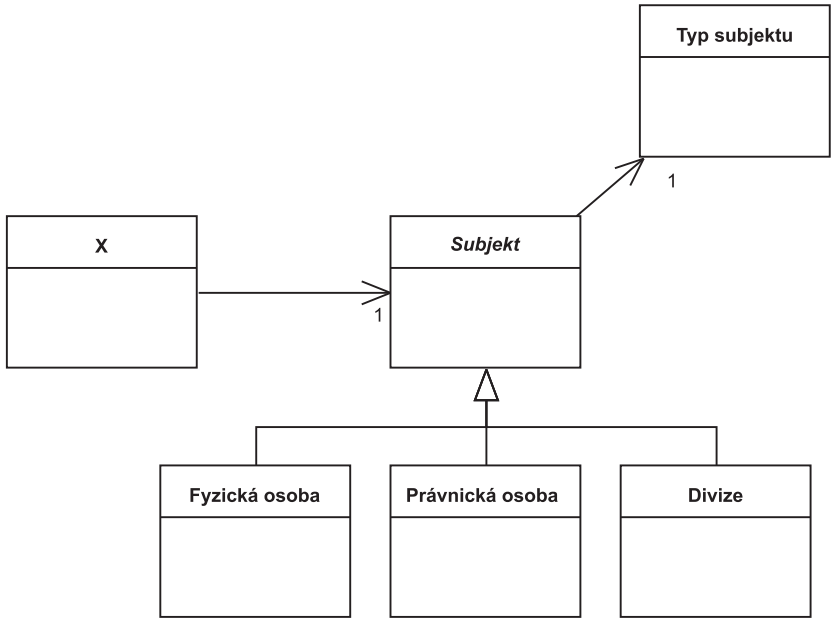
Vzor *Party*

Vzor *Party* zavedl pan Fowler (viz [6]). Jedná se vlastně o návod řešení, jak by měla vypadat agenda Subjektů (právnické osoby, fyzické osoby, divize atd.). Aniž by to pan Fowler uváděl, tak jeho řešení vychází ze základního vzoru *Odkaz na vrchol stromu* následujícím způsobem: Protože je třeba, aby Subjekty figurovaly v různých pozicích jako různé subtypy (například ručitelem může být právnická osoba nebo fyzická osoba apod.), je vhodné všechny subjekty umístit pod jednoho společného abstraktního předka třídy Subjekt. Odkázat si na Subjekt znamená díky zástupnosti rolí odkázat se instančně na libovolného potomka, viz obrázek 3.52.

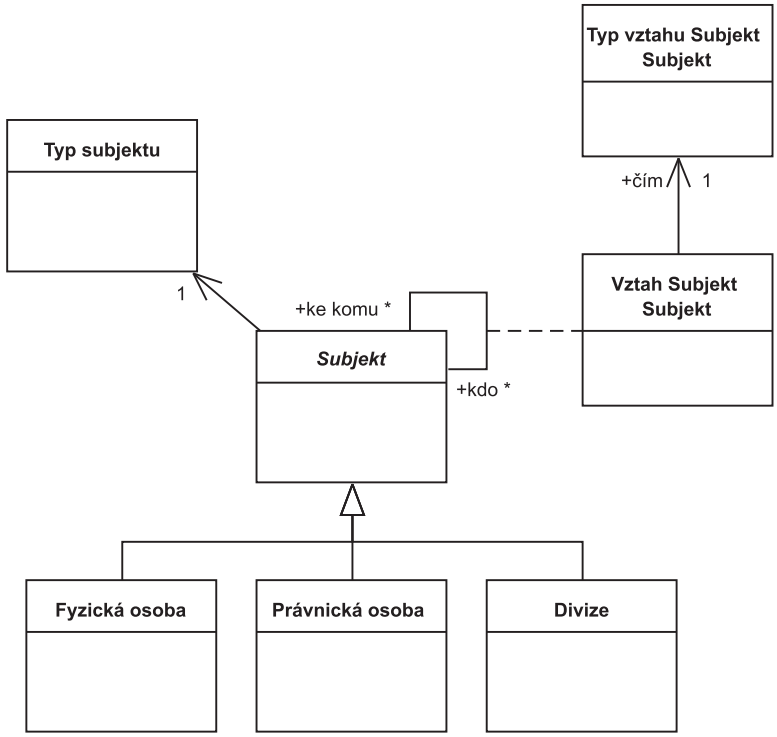
Díky této konstrukci společného předka může být (viz vzor *Odkaz na vrchol stromu*) na konci ukazatele instance libovolného potomka. Všimněme si této paradoxní vlastnosti vrchní třídy: Pokud pomineme odkaz do typu (diskriminátor), tak vrchní abstraktní třída Subjekt neobsahuje žádné společné údaje. Její existence je dána požadavkem kompatibility, aby libovolný subtyp mohl vystupovat v roli subjektu. Můžeme kdykoliv přidávat další typ subjektu – tedy dědice, kteří jsou začleněni do dané „party“.

Dalším krokem je zavedení vztahu mezi Subjekty, jako je například „člen dozorčí rady, člen představenstva, je dceřinou společností, je divizí, atd.“. Vzor *Party* doporučuje tyto vztahy zavést na úrovni mezi Subjekty, tedy jako asociační třídu mezi třídou Subjekt a znovu na třídu Subjekt. Krom toho je třeba ještě určit, o jaký vztah se jedná, což bude reprezentovat číselník obsahující předešlý výčet, viz obrázek 3.53.

Výhodou umístění vztahu na úrovni horní abstraktní třídy je v tom, že můžeme nyní zavést libovolný vztah mezi libovolnými instancemi dědiců subjektu. Je zřejmé, že nemá smysl zavádět libovolné vztahy mezi libovolnými dědici. Například vztah „Právnická osoba versus Právnická osoba jako člen dozorčí rady“ nemá smysl, stejně tak „Fyzická osoba versus Fyzická osoba jako člen představenstva“ atd. Zavedeme proto vzor *Povolené kombinace*. Když



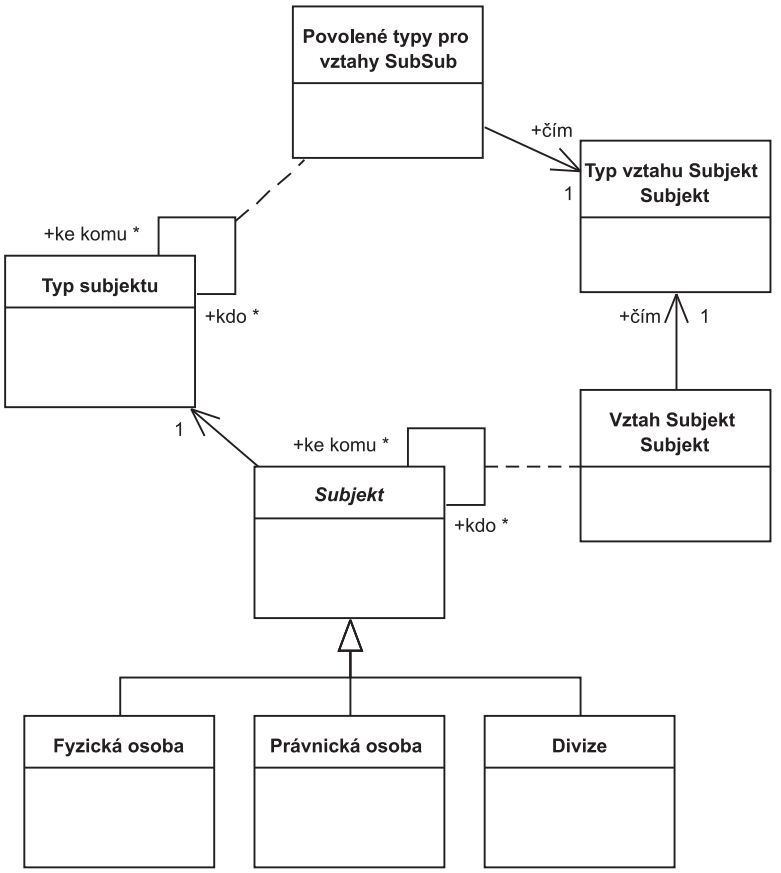
Obrázek 3.52: Vzor *Party* jako aplikace vzoru *Odkaz na vrchol stromu*.



Obrázek 3.53: Přidání vztahu Subjekt versus Subjekt.

se blíže podíváme, o co v těchto povolených kombinacích jde, jsou to vlastně trojkombinace. Daly by se sice řešit pomocí trojné asociační třídy, my však zvolíme řešení rozdílné (i když

podobné), které také obsahuje „tři konce“, ale má trochu jiný tvar, než trojná asociační třída, viz obrázek 3.54.



Obrázek 3.54: Přidání povolených kombinací pro typy vztahů.

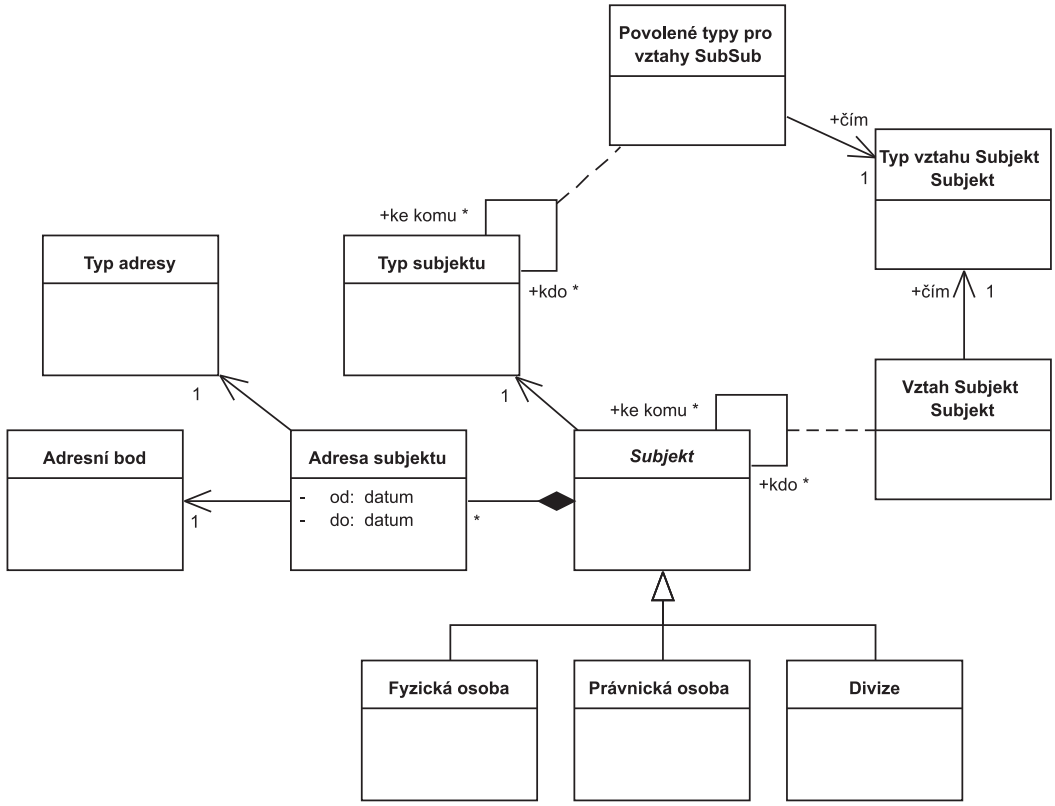
Například povolené kombinace by instančně mohly vypadat tak, jak je uvedeno v tabulce 3.2.

Tabulka 3.2: Povolené kombinace instancí.

Typ subjektu	Typ vztahu	Povolené kombinace		
		kdo	ke komu	čím
1 fyzická osoba 2 právnická osoba 3 divize	1 člen dozorčí rady 2 člen představenstva 3 je dceřinou společností 4 je divizí	1	2	1
		1	2	2
		2	2	3
		3	2	4

Pokud máme ještě implementovány adresy pomocí adresních bodů, potom by dalším krokem bylo zavedení adres subjektů pomocí vzoru *Vlak má svoje vagóny*, viz obrázek 3.55.

Navíc by bylo vhodné zavést ještě povolené kombinace mezi typem adresy a typem subjektu (například trvalé bydliště nemá smysl pro právnickou osobu). Výsledkem je potom model na obrázku 3.56, doplněný o tyto povolené kombinace.



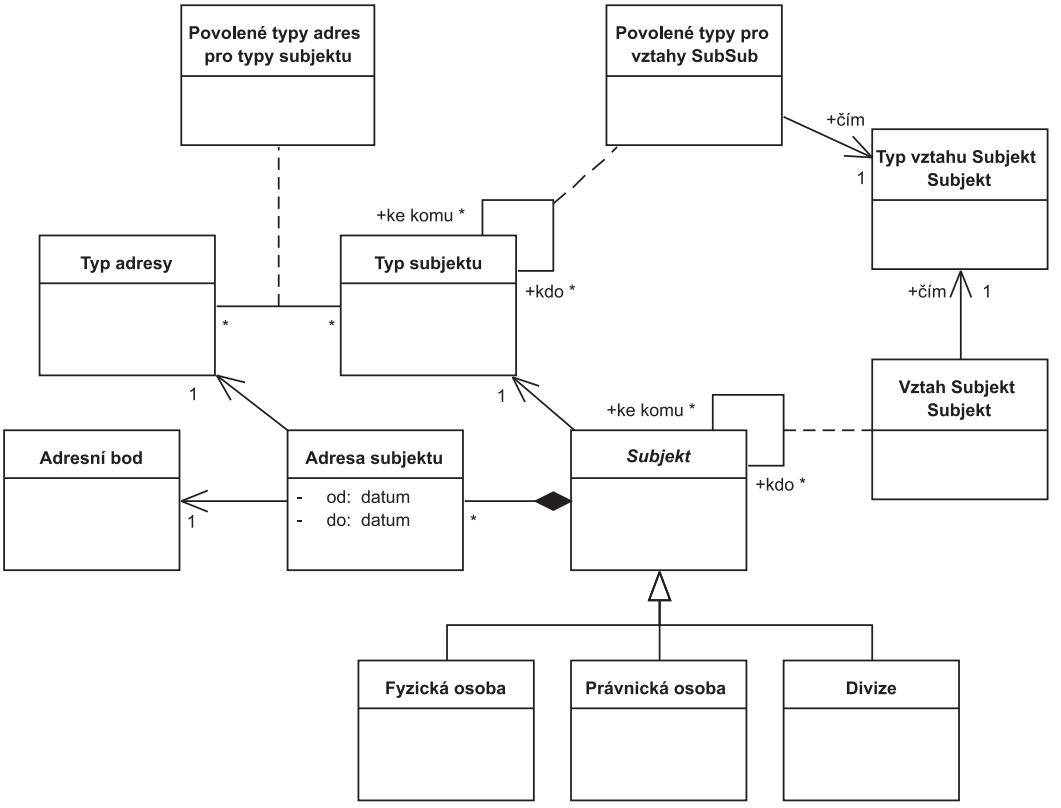
Obrázek 3.55: Přidány adresy subjektu podle vzoru *Vlak má svoje vagóny.*

Vzor *Modulární nůžky*

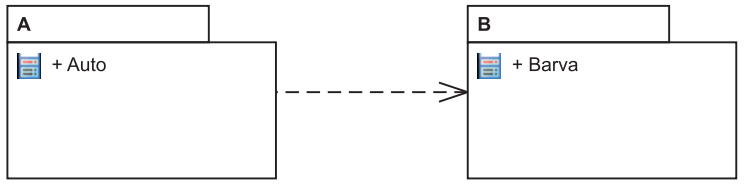
U *Class Diagramu* je mimo jiné důležité sledovat i vztahy závislosti mezi třídami, které vyplývají z interakcí. Tyto závislosti mezi třídami potřebujeme pro určení následného rozdělení kódu do modulů, neboli v modelu UML pro rozmístění tříd do prvků typu *Package* a následně pro tvorbu tzv. komponentního modelu. Při rozmísťování tříd do prvků typu *Package* totiž musíme dbát na určitá pravidla. Správné a nikoliv chybné umístění tříd do prvků typu *Package* totiž umožňuje rozdělit aplikaci do správných vrstev.

Představme si, že jsme umístili skupinu tříd do prvku typu *Package*, který jsme nazvali jako A a druhou jinou skupinu tříd jsme umístili do druhého prvku typu *Package*, který jsme nazvali jako B. Nechť v našem příkladu do prvku Package A byla umístěna třída Auto a do prvku Package B byla umístěna třída Barva ze vzoru *Auto má barvu* (viz obrázek 3.12 na straně 54). Protože Auto je umístěno do prvku Package A a Barva do prvku Package B a přitom „Auto potřebuje Barvu“, tak tímto vzniká závislost také mezi prvky Package A a B v tom smyslu, že Package A potřebuje Package B. Pokud taková situace nastane, měli bychom ji v modelu tříd znázornit pomocí vztahu Dependency mezi prvky typu *Package* tak, jako na obrázku 3.57.

Vztahy závislosti mezi prvky typu *Package* udávají, který *Package* musíte mít k dispozici, pokud používáte jiný Package. Zde například, pokud používáte prvek Package A, měli byste mít současně k dispozici prvek Package B, jinak budete mít nekonzistentní model (tj. „Auto bez Barvy“).



Obrázek 3.56: Přidání povolených typů adres pro typy subjektu.

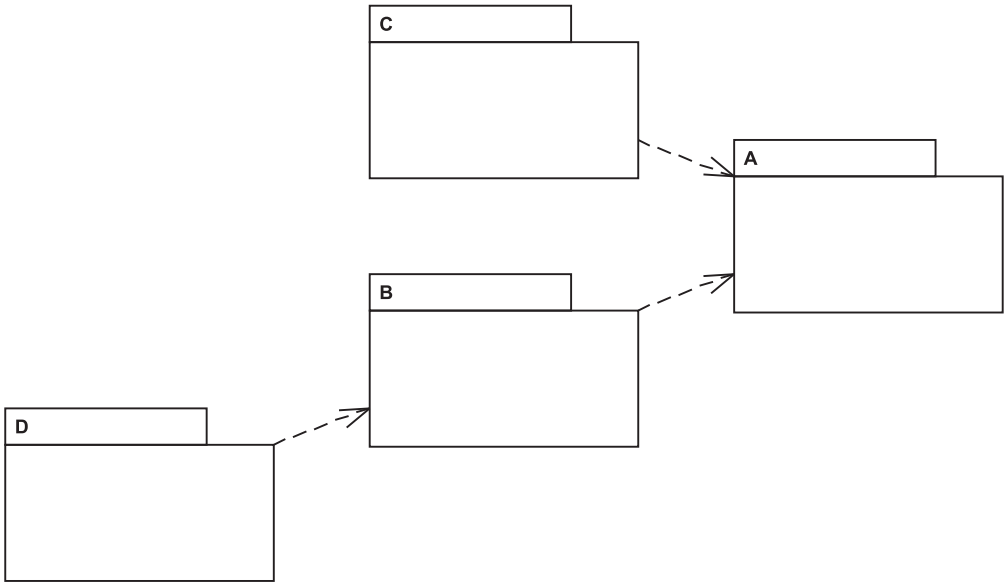


Obrázek 3.57: Závislost prvku Package A na prvku Package B.

Při návrhu IS je velmi důležité zabránit vzniku tzv. molochálních systémů, tj. systémů s velkými „kusisky“ kódu, proto je třeba řídit se tímto doporučením: Správné rozmístění tříd do *Package* by mělo být provedeno bez cirkulárních závislostí. V tom případě totiž samotné prvky *Package* velmi jasně a přehledně udávají vrstevnatost systému. Pokud nastane cirkulární reference, pak se tato informace stává velmi nepřehlednou a nakonec vede k návrhu již zmiňovaných molochálních systémů, tj. systémů s velkými „kusy“ kódu podléhajících změnovému řízení.

Uveďme si příklad. Nechť jsme nalezli třídy a rozmístili je do prvků typu *Package* tak, že vznikly vztahy Dependency, znázorněné na obrázku 3.58.

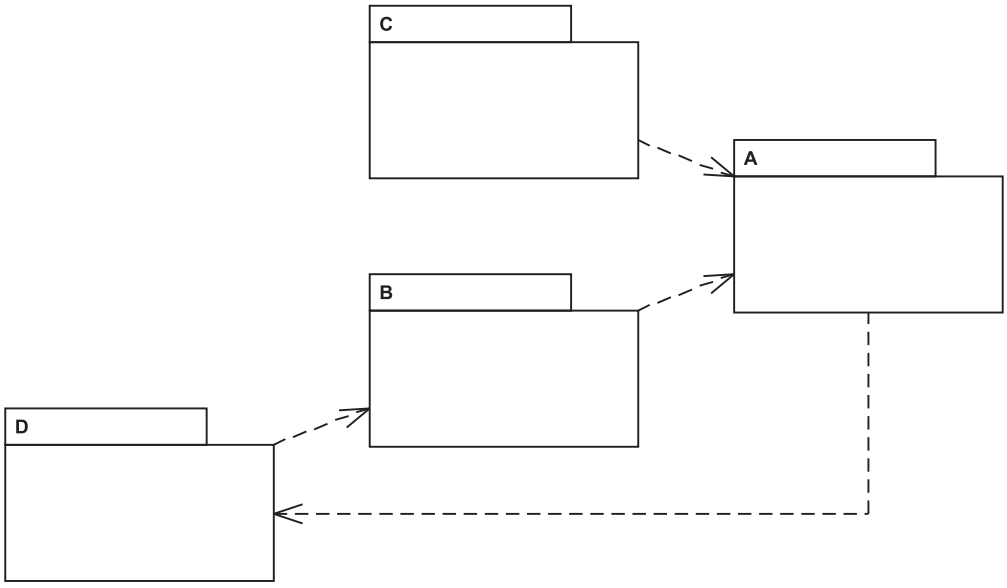
Z obrázku 3.58 je patrné, jaká skupina tříd (tj. *Package*) je nejvíce centrální, která je nad ní a který Package je více satelitem. V tomto případě je Package A centrální skupinou tříd, nad ní jsou prvky Package B a C a nad B je ještě D. Všimněme si, že rozmístění do vrstev má velkou vypovídací schopnost i pro vedoucího projektu, který vidí, co reprezentuje



Obrázek 3.58: Prvky *Package* s přehlednými vrstvami v *Class Diagramu*.

„centrální práce“, na kterých jsou závislí ostatní. Ideální by bylo implementovat systém v pořadí balíčků tříd A, B, C a nakonec D.

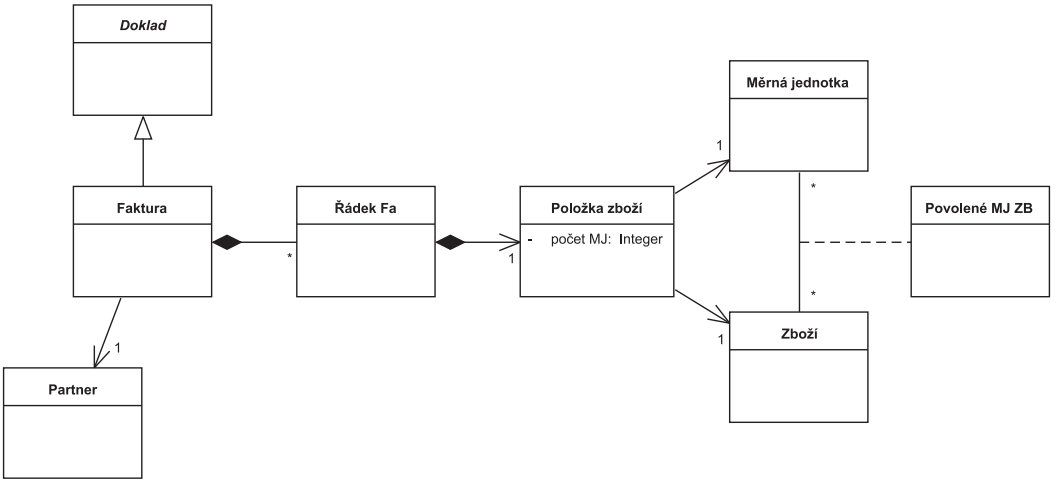
Problém spočívá v tom, že původně přehledný obrázek vrstev se najednou stane silně nepřehledným, pokud se v něm vyskytne cirkulární závislost. Nechť se v našem příkladu dodatečně zjistí, že nějaká třída v prvku Package A je závislá na nějaké třídě v prvku Package D, viz obrázek 3.59.



Obrázek 3.59: Nepřehledné vrstvy s cirkulární referencí.

Vidíme, že pohled na vrstvy se najednou úplně změnil: Nyní existují dvě vrstvy, jedna vnitřní vrstva A + B + D a nad ní C. Z uvedeného příkladu je zřejmé, že pro správné rozmístění tříd do vrstev, tj. do nepříliš velkých prvků *Package* bez cirkulárních závislostí, je třeba znát, jak fungují závislosti mezi třídami a jak správně „stříhat“ model tříd. Právě k tomu slouží vzor *Modulární nůžky*, který si nyní vysvětlíme.

Vyjděme z ukázkového diagramu na obrázku 3.60, ve kterém se vyskytují všechny vztahy potřebné v *Class Diagramu* s výjimkou násobné asociační třídy, která se od dvojné liší pouze počtem konců.



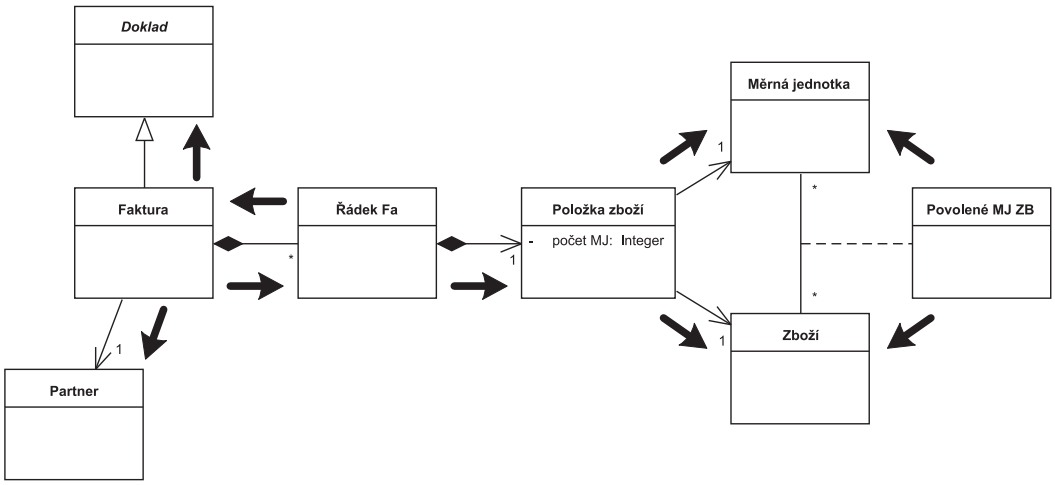
Obrázek 3.60: Příklad pro zavedení vzoru *Modulární nůžky*.

Všimněme si, že oproti již dříve uvedenému příkladu s povolenými kombinacemi se zde objevuje navíc nová třída Položka zboží, která je vložena do řádku faktury v kompozici ku 1. Důvod její existence přenechávám nyní jako vhodnou otázku k úvaze pro čtenáře nebo do případného školení.

Jako další krok si namalujme vztahy závislosti, viz obrázek 3.61. Ty vyjadřují, která třída potřebuje kterou jinou třídu. Tato závislost by se projevila tak, že pokud bychom jednu třídu z modelu vyjmuli (například takzvaně „nepřilinkovali“ do systému v některém z modulů), pak by závislá třída tak říkajíc „zařvala“, že jí daná „nepřilinkovaná třída“ chybí.

Vztahy závislosti jsou, jak je vidět, následující:

- Faktura potřebuje Doklad díky *Generalizaci*;
- Faktura potřebuje Partnera díky *Odkazu do seznamu*;
- Faktura potřebuje Řádek faktury (*Faktura má řádky faktury*);
- Řádek faktury potřebuje Fakturu (zná svého majitele, na konci není šipka);
- Řádek Faktury potřebuje Položku zboží (*Osoba má adresu* jako v e-shopu);
- Položka zboží potřebuje Měrnou jednotku (*Odkaz do seznamu*);
- Položka zboží potřebuje Zboží (*Odkaz do seznamu*);



Obrázek 3.61: Závislosti mezi třídami díky vztahům v *Class Diagramu*.

- Třída Povolené MJ ZB potřebuje jak třídu Měrná jednotka, tak třídu Zboží (asociační třída propojující instance z těchto tříd).

Je třeba zdůraznit, že tyto „tlusté černé šipky“ závislostí mezi třídami se do modelu nikdy nemalují. Vyplývají totiž z již existujících vztahů, a pokud bychom je do diagramu umístili, pouze bychom čtenáře zmátli. My si je nyní uvádíme proto, abychom si mohli uvést postup modulárních nůžek.

Postup střihu ve vzoru *Modulární nůžky*

Obrázek 3.61 odpovídá v matematické teorii (teorie grafů) tzv. orientovanému grafu, který obsahuje vrcholy (to jsou nyní naše třídy) a orientované hrany (to jsou naše tlusté černé šipky). Pokud bychom se dotázali matematika, jak se v orientovaném grafu zjistí, zda je v něm kruh (tj. situace, kdy jdeme podél šipek a dojdeme opět do vrcholu, ze kterého jsme vyšli), odpověděl by následovně:

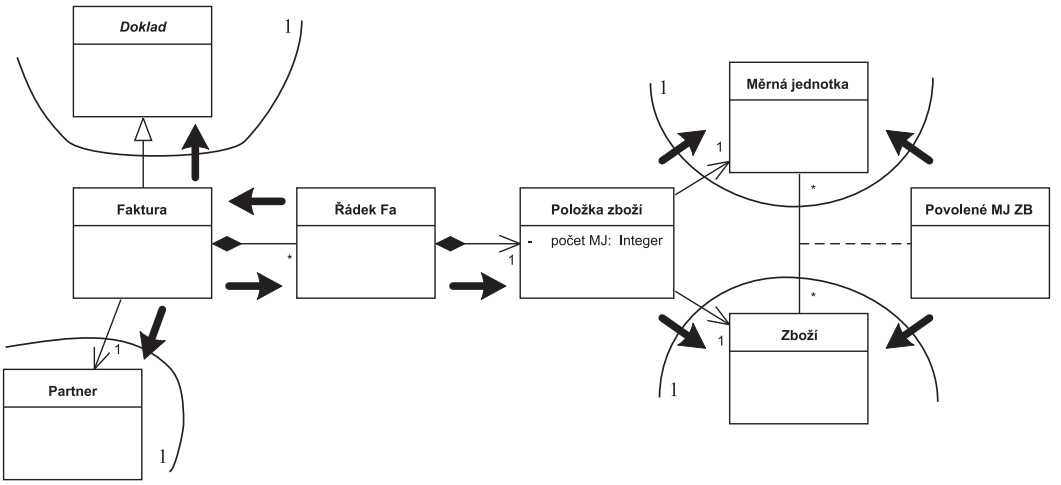
> *Najděme nejprve tak zvané listy grafu. To jsou v našem případě ty třídy, do kterých šipky vedou a z nichž už šipky nevedou. Označme si je a z algoritmu je vylučme. Opět najděme listy (tj. již bez prvních „vyhozených“ listů) a opět je označme a z algoritmu vylučme. Takto pokračujme cyklem dál a dál. Pokud vyloučíme všechny prvky, graf nemá kruh. Pokud se někde zarazíme a nemůžeme pokračovat dále, protože již neexistují prvky, do kterých šipky vedou a z kterých nevedou, zůstal nám v ruce jeden nebo vícero kruhů.*

Aplikujme tento postup na obrázek 3.61. Dostaneme tyto výsledky:

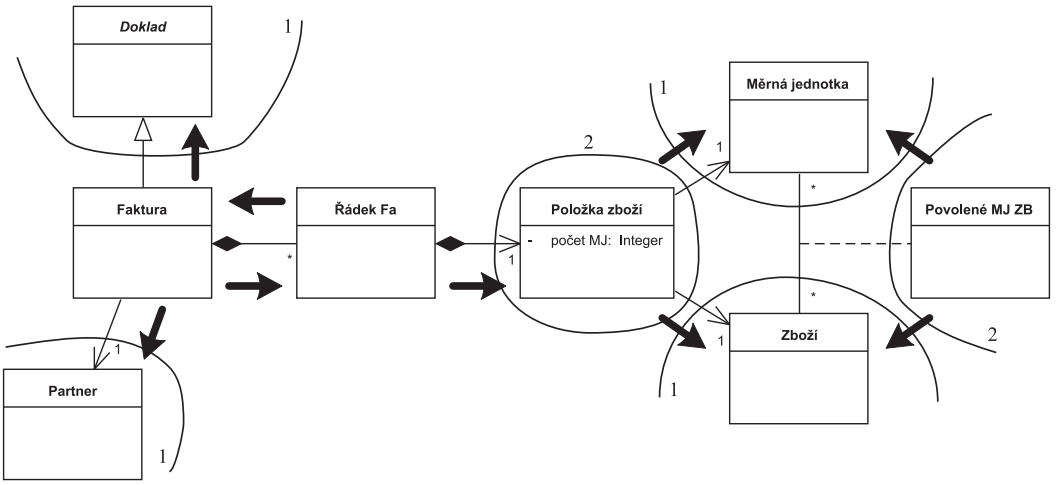
V prvním kroku zjistíme, že listy jsou třídy Doklad, Partner, Měrná jednotka, Zboží. Do nich totiž šipky vedou a z nich nevedou. Označme je tedy a vylučme z algoritmu jako krok číslo 1, viz obrázek 3.62.

Nyní pokračujme v postupu, ale již označené třídy se jej neúčastní, tj. nyní hledáme třídy, do nichž šipky vedou, ale z nich vedou šipky pouze do již označených oblastí. V druhém kroku vypadnou třídy Povolené MJ ZB a Položka zboží, viz obrázek 3.63.

Nyní nám již zůstaly třídy, u kterých nenajdeme ani jednu, do které by šipky vedly a šipky z ní by vedly pouze do vymezené oblasti. Identifikovali jsme kruh a provedeme poslední označení oblastí, viz obrázek 3.64.



Obrázek 3.62: Výsledek prvního kroku.



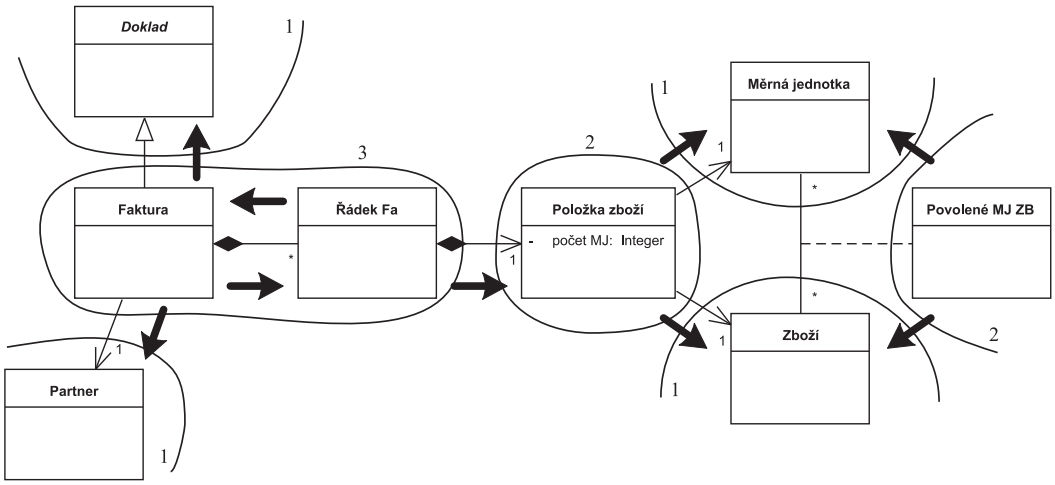
Obrázek 3.63: Druhá iterace při vyhledávání kruhu.

Postup modulárních nůžek je nyní velmi jednoduchý: Třídy rozvrstvíme do prvků typu *Package* tak, že je povoleno „stříhat“ pouze mezi jednotlivými oblastmi, tj. nesmíme přetnout vyznačenou oblast. Můžeme si uvést sice laickou, ale velmi výmluvnou představu: Vyznačené oblasti jsou kameny, mezi nimi je voda a nůžky mohou stříhat systém pouze po vodě, nikoliv napříč kameny.

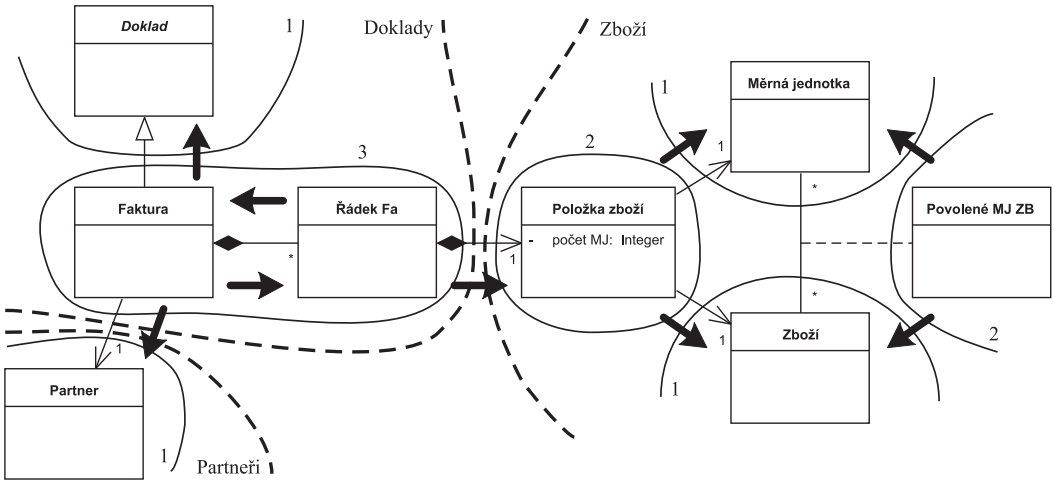
Tuto činnost „střihu systému“ provádějí většinou hlavní analytik spolu s hlavním designérem za účasti vedoucího projektu. Jedná se totiž o vrstvení systému na části, které mají mezi sebou přesně definované závislosti, což má vliv na další návrh modulů a současně na rozdělování prací. V uvedeném příkladu by se například přijalo řešení „střihu“ tak, jak je uvedeno na obrázku 3.65.

Systém byl takto rozdělen na tyto části: Package Zboží, Package Doklady a Package Partneři. Tlusté černé šipky mezi těmito oblastmi se projeví jako následné závislosti mezi prvky *Package*, znázorněné na obrázku 3.66.

Uvedený postup vzoru *Modulární nůžky* je třeba ještě doplnit o určitá nutná doporučení:



Obrázek 3.64: Poslední iterace – kruh Faktura/Řádek Faktury.

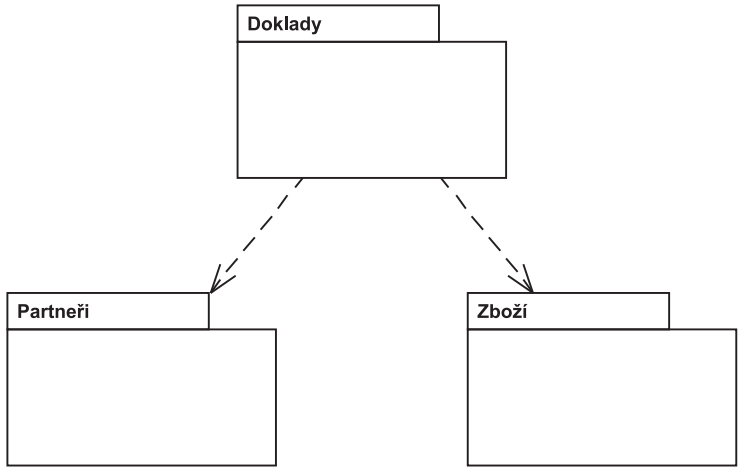


Obrázek 3.65: Možné rozmístění tříd do prvků typu *Package*.

1. Aby uvedený postup rozdělení tříd do modulů s necirkulárními závislostmi korektně fungoval, musí v některých případech technolog nasadit vzor *Observer*, resp. nějakou jeho technologickou obdobu (LISTENER v jazyce JAVA, delegáty v C#, události ve Visual Basicu, call back v C apod.). Jedná se o technologický problém odstranění zpětné funkcionální vazby, který sice nemusí zajímat analytika, ale považuji za vhodné si uvedenou problematiku vzoru *Observer* prostudovat.

2. Při analytickém návrhu je třeba dbát na „čistotu“ řešení. Pokud totiž zavedeme do daného řešení prvky, které patří jinam, narušíme tak rozmístění prvků a nastávají problémy. Daný modul má sloužit příslušnému účelu a neměl by do něj být umístěn jiný prvek, který tomuto účelu neslouží.

3. V některých případech lze vyřešit nežádoucí cirkulární referenci mezi třídami vložením asociační třídy mezi tyto třídy, která tuto vazbu nahradí.



Obrázek 3.66: Nalezené závislosti mezi prvky typu *Package*.

Kapitola 4

Use Case Diagram

V předešlých kapitolách jsme se seznámili s návrhem modelu tříd a to jak na úrovni analytického modelování, tak designu. Návrh modelu tříd ale není prvním krokem analytika, ten nejprve vyhledává případy užití a teprve poté třídy. Důvod, proč jsme si vysvětlili *Class Model* jako první, je ten, že bez správného pochopení vztahů v *Class Diagramu* nelze dobře tvořit model případů užití (tzv. prvky typu *Use Case*) a jejich zpracování. K tomu, aby analytik správně zpracoval model případů užití, musí znát a chápat, jak fungují vztahy mezi informacemi v informačním systému, což vyjadřuje právě model tříd. Vztah mezi informacemi je omezen syntaxí vztahů mezi třídami, resp. instancemi (tj. kompozice, číselníková vazba atd.), a proto při zpracování scénářů případů užití je třeba tyto vztahy dobře ovládat.

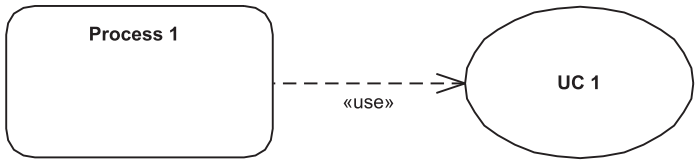
4.1 Prvek typu *Use Case* neboli případ užití

Jeden případ užití lze chápat tak, jak je nazýván: „Je to jeden případ užití informačního systému.“ Tvorba *Use Case Modelu* je tedy modelování IS z náhledu jeho možných použití. Zde je důležitý pohled na „užitek“, který případ užití daného IS přináší, a to nejenom analyticky, ale i obchodně. Někdy se zjednodušeně případ užití chápe jako „funkcionalita systému“, tedy výčet případů užití vlastně v tomto pojetí znamená „co systém bude umět“. Je to sice názorné a pro laika srozumitelné, ale zde bychom měli být velmi opatrní. Jak si totiž ukážeme dále, z hlediska použití systému existují svou povahou a pozicí v systému vlastně dva druhy případů užití, které se sice syntaxí neodlišují, ale analytik je musí bezpodmínečně rozlišovat.

První z nich jsou ty případy užití, které skutečně reprezentují situaci, kterou chápeme jako získání užitku při použití systému. U těchto případů užití si můžeme představit, že fungují v časové posloupnosti takto: Nejprve jsou okolí a systém ve stavu „čekání“ (něco jako „Idle“ stav na obou stranách). V této chvíli nikdo nic od systému nepotřebuje a nikdo nic nechce. Najednou se venku mimo systém začne odehrávat nějaký děj, například do firmy přijde objednávka, kterou je třeba zpracovat apod. Takovému ději mimo systém budeme říkat *Business Process* neboli česky *proces podniku*, zkráceně *proces*. Děj v okolí (tedy proces) probíhá mimo informační systém a v určitém okamžiku někdo (nebo něco) „pocítí potřebu použít“ informační systém. V rámci tohoto děje se tedy přistoupí k IS a spustí se (neboli „instanciuje“) daný případ užití a použije se.

Poznámka: *Spouštěcím elementem ve zvláštním případě nemusí být prvek z okolí, ale čas. Jedná se o případy užití spouštěné vnitřním časovačem, například noční uzávěrka, zpracování apod.*

Pokud označíme daný proces podniku jako „Process 1“ a daný případ užití jako „UC 1“, můžeme si danou situaci znázornit na obrázku 4.1.



Obrázek 4.1: Spuštění (resp. použití) případu užití procesem podniku.

Zde je třeba upozornit na tuto důležitou okolnost: Pokud čtenář studuje scénář popisující děj procesu „Process 1“ na obrázku 4.1, potom by měl při čtení tohoto scénáře dospět až do bodu, kde bude explicitně vypsáno použití případu užití „UC 1“ například takto:

```
...
a dále se použije případ užití „UC 1"...
...
```

Kromě těchto případů užití existují i další případy užití, které nemají tuto povahu „spouštěných případů užití z procesů“. Tyto druhé jsme zatím nebrali a v prvním kroku se nevyhledávají.

> *V prvním kroku se vyhledávají případy užití, které mají povahu spouštěných případů užití podle obrázku 4.1 na základě požadavku použití z procesu.*

Zde musím zdůraznit jednu opravdu velmi důležitou skutečnost, se kterou mají problémy zejména programátoři, kteří se podílejí na analytických pracích. Případ užití není totéž, co spuštěná funkce. Základním prvkem pro identifikaci případu užití je požadavek jako nutkání použít informační systém a z něj se odvíjí odhalení a definice daného případu užití. Případ užití odhalujeme na základě sekvence děje: Nikdo nic nepotřebuje, venku se něco odehraje, nastane událost a požadavek na to, aby se použil systém. Tento požadavek neboli nutkání použít systém je obrazem nalezeného případu užití, který vznikl právě proto, aby tento požadavek užitku (požadavku na užitek) splnil.

4.2 Vyhledávání případů užití

Běžné informační systémy obsahují řádově stovky až tisíce případů užití. Z toho důvodu při tvorbě *Use Case Diagramu* musí analytik řešit tyto zásadní otázky: „Jak případy užití nalézat systematicky? Jak postupovat, abychom nějaký neopomněli, nějaký nezapomněli zavést a zpracovat?“

Jedna z klasických metodik vyhledávání případů užití byla preferovaná hlavně v období zrodu jazyka UML, tj. ve druhé polovině 90. let minulého století. Nazvěme ji pracovně „aktérová škola“. Tato škola přistupuje k řešení problému nalézání případů užití pomocí jednoduchého pravidla: „Najděme nejprve prvky z okolí systému (tzv. aktéry neboli prvky typu *Actor*), které budou potřebovat a používat systém, případně najděme ty prvky z okolí, které budou komunikovat se systémem, resp. ty prvky z okolí, kterým bude systém předávat informace. Až nějaký takový prvek (neboli prvek typu *Actor*) z okolí najdeme, zeptejme se, proč a nač aktér tento systém potřebuje, resp. jak jej použije, resp. jak komunikuje. Tímto

najdeme všechny případy užití.“ Tato dnes již klasická metodika se ukázala jako nikoliv stoprocentně účinná. Mnohdy je navíc zdrojem nových problémů: vede ke zdlouhavým a zbytečným diskusím nad názvy aktérů nebo dojde k chybnému určení počtu aktérů, kteří komunikují s daným případem užití, tj. nastává chybný jev, kdy počet logických rozhraní nesedí s počtem aktérů u daného případu užití (bude blíže rozvedeno ve zvláštní kapitole).

Je třeba podotknout, že „aktérová škola“ se jakžtakž dobře osvědčuje u technologických systémů (jako je například hlásič požáru, řídicí modul výtahů v mrakodrapu, komunikační modul apod.), protože se jedná o systémy s nikoliv velmi početnými procesy v okolí a s relativně malým počtem případů užití (řádově desítky). Tyto systémy nebývají většinou složité procesně, ale po technologické stránce, například vyskytne se problém, že je k dispozici málo fyzické paměti nebo aplikace musí pracovat ve vícero vláknech, zpracovávat signály apod. U těchto systémů odpadá i nepříjemná diskuse ohledně pojmenování aktérů, protože s výjimkou „nějaké nepodstatné živé obsluhy“ jsou aktéry u technologických systémů většinou externí softwary nebo hardware se specifickým rozhraním, které lze snadno pojmenovat bez kolizních názvů. U informačních systémů, které jsou charakteristické právě vysokým stupněm složitosti chodu procesů okolo systému, však „aktérová škola“ výrazně selhává.

Je třeba podotknout, že i přes toto selhání stoprocentnosti není tato škola úplně zatracena jako nesmyslná. Stává se rozumným podpůrným (a tedy dodatečným doplňkovým) postupem u jiné školy, kterou budeme nazývat jako „procesní škola“ (viz dále). Nedokonalá „aktérová škola“ doplňuje lepší a korektnější „procesní školu“ tak, že pokládá dodatečně kontrolní otázky ve smyslu „kdo použije systém?“. Navíc mnohdy získáme od zákazníka dokumenty, ve kterých se to hemží „rolemi v podniku“, jako jsou ředitel, manažer, obchodník, realitní makléř apod. Tyto role se stávají vstupními parametry zmíněných otázek „kdo a proč používá systém?“ Musíme však být u tohoto postupu s aktéry opatrní na jednu záludnost: Některé případy užití bývají shodné pro několik rolí v podniku, jinak řečeno mnohdy dva nalezené případy užití nakonec splývají v jeden případ užití pro dvě a více různých rolí v podniku. I když se jedná o jeden případ užití, je pak tendence hovořit o dvou případech pro dva aktéry. Anebo se v tomto případě do modelu zavleče další hrubá chyba: Určí se pro několik rolí v podniku správně jeden „splývající“ případ užití, ale namalují se k němu dva a více aktérů, i když případ užití má pouze jednoho aktéra (řešení rolí bude podrobněji vysvětleno ve zvláštní kapitole).

Nakonec ještě jedna poznámka: Existuje typ softwaru, kde nemusíme používat žádnou zvláštní školu pro vyhledávání případů užití a vystačíme si pouze s dobrým logickým členěním případů užití do prvků typu *Package*, což u tohoto software vede k systematičnosti. Jedná se o software typu „desktop aplikace“, jako je například textový editor, tabulkový procesor (Word, Excel apod.) s jedním uživatelem.

> *Pro systematické vyhledávání případů užití se v posledních letech dostala více do popředí škola tzv. procesního modelování (*Business Process Modeling*, neboli* BPM*).*

Podstata „procesní školy“ pro vyhledávání případů užití je sice podobná, ale trochu jiná, než u „aktérové školy“. Její návod zní také velmi podobně: „Najděme všechny procesy podniku v okolí, které vedou k použití systému, následně tak najdeme případy užití vyvolané těmito procesy podniku“. Pokud se podíváme na obrázek 4.1, potom tato škola přesně respektuje úvodní myšlenku „jak chápat případy užití a jejich vztah k okolí“: Nalezené procesy vyvolávají požadavek na použití systému a spouštějí nalezené případy užití. Základní otázkou procesní školy tedy není „kdo použije systém“, ale „co se děje okolo systému, že se použije systém“. Tato škola je výhodná také pro zákazníka: Nemusí se pracně přemýšlet nad

Sledujte aktuální novinky na http://www.objects.cz

tím „kdo bude používat systém“, ale otázka zní jednodušeji: „jak to děláte“, resp. „co se bude u vás dít, že použijete systém“.

Pokud se například budeme věnovat agendě zpracování úvěrů v bance, budeme si odpovídat na dotaz „Jak chodí úvěry“ a nikoliv „Kdo s nimi pracuje“. (Jak však bylo řečeno, procesní škola může být doplněna také navíc otázkou z „aktérové školy“ ve smyslu „kdo potřebuje systém“, ale tato otázka slouží spíše pro kontrolu a revizi již nalezených případů užití při již nalezených procesech). Procesní škola nalézání případů užití má několik nesporných výhod:

- umožňuje popsat chod procesů (například „jak chodí úvěry“);
- umožňuje systematický rozklad procesů a tím dát řád do haldy případů užití;
- zákazníkovi je tato škola dobře srozumitelná, modely procesů podniku s případy užití bývají zákazníkem přímo vyžadovány jako součást projektové dokumentace.

Jak vyplývá z účelu zavedení procesní školy, tedy k čemu vlastně slouží (nezapomeňme, že hledáme případy užití informačního systému!), je zřejmé, že procesní model popisuje chod podniku v návaznosti na nový vyvíjený systém a nikoliv chod podniku „dnešní“, tj. ještě bez navrhovaného systému. Velmi častou otázkou je, zda máme modelovat i současný stav nebo nikoliv. Odpověď je čistě praktická: Pokud se po vás současný model podniku vyžaduje (a zákazník jej zaplatí), pak jej musíte vyhotovit, tj. model současného chodu podniku se stává součástí dokumentace projektu stejně jako budoucí chod procesů s novým systémem. Pokud se výstupy současného stavu podniku přímo nevyžadují a nemusíte odevzdávat formalizovaný model současného stavu, nemusíme jej modelovat, ale to ještě neznamená, že se mu nebudeme věnovat. Je vhodné si se zákazníkem pohovořit i o současném stavu, tedy o současném chodu podniku se současným informačním systémem, a tyto poznatky si nějak zapsat jako cenný zdroj informace. Pokud se však nevyžaduje výstup, nemusíme být při zápisu až tak formální (například nemusíte malovat diagramy), stačí pouze textový tvar zápisu. Pokud se i tak vedoucí projektu rozhodne, že součástí projektu bude procesní model stávajícího stavu, aniž by to zákazník požadoval, jedná se v tom případě o ekonomickou stránku a možnosti zdrojů projektu, tj. o otázku, kdo to zaplatí a kdo a za jak dlouho tuto práci provede.

4.3 Jaké jsou nutné výstupy analytika při vyhledávání případů užití pomocí procesů podniku (*BPM*)

Analytik odevzdává tři základní druhy analytických artefaktů při vyhledávání případů užití v procesní škole vyhledávání případů užití:

1. *Diagramy Rozkladu procesů* (tzv. dekompozice procesů);
2. *Diagramy Podpory IS*;
3. *Diagramy Chodu procesu*.

Je třeba upozornit, že sice tyto tři body musíme vysvětlit postupně neboli sekvenčně za sebou (jinak to nejde), ale analytik na nich pracuje souběžně, tedy současně. Neměl by z tohoto výkladu vzniknout dojem, že nejprve se dělá bod 1, pak bod 2 a pak bod 3. Pracujeme na té části, o které máme momentálně nejvíc informací, a poznatky o ostatních bodech začnou z těchto prací postupně vyplývat.

Sledujte aktuální novinky na http://www.objects.cz

4.3.1 Diagram Rozkladu procesů

Velkou výhodou „procesní školy“ je systematičnost daná možností rozkladu procesů a to nám umožňuje následně uspořádat systematicky i případy užití. Nejprve si vysvětlíme princip tohoto postupu a poté i syntaxi v UML. Základní myšlenka je jednoduchá: Procesy umějí takzvaný rozklad procesů. V rozkladu procesů existují „vyšší“ procesy, které se skládají z „nižších“ procesů, resp. obráceně, vyšší procesy se rozkládají na nižší procesy. Laik rozumí tomuto skládání procesů (nebo naopak rozkladu procesů směrem dolů) a intuitivně jej chápe opravdu správně. Například když se bavíme se zákazníkem o tom, jak vypadá „proces podniku ‚Fakturace v podniku‘ “, bude pro něj srozumitelná tato jednoduchá věta: *Proces „Fakturace“ se skládá z procesů „Založení faktury“, „Editace faktury“, „Odsouhlasení faktury“ atd.*

Všimněme si, že vyšší proces „Fakturace“ je v tomto případě složen z nižších procesů „Založení faktury“, „Editace faktury“, „Odsouhlasení faktury“ atd., respektive naopak, proces „Fakturace“ se rozkládá na nižší podnikové procesy. Díky této vlastnosti procesů „skládat se“ můžeme zavést systematičnost rozkladu všech procesů podniku. Zavedeme vrchní „top“ proces a dáme mu název IS. Chápeme jej jako „proces podniku zahrnující všechny procesy podniku podporované IS“. Následuje rozklad odshora dolů – na první úrovni rozložíme proces ještě dost nahrubo (například „Evidence subjektů“, „Účetnictví“, „Skladové hospodářství“ atd.). Další rozklad „zjemní“ další procesy atd. Takovouto dekompozici odshora dolů můžeme chápat jako rozdělení na oblasti řešení daného IS podobně jako zoom u lupy: Nejprve vidíme oblasti řešení jako celkový obraz a postupně pohled zaostřujeme na větší detaily. Tento rozklad procesů nebo také oblastí řešení znázorníme graficky, přičemž pro zákazníka je nejlépe odevzdat jej ve formě stromu a nazvat jej „oblasti řešení IS“. Asi není třeba zdůrazňovat, že zákazník tento strom vidí rád, protože mu dává dobrý přehled o funkcionalitách IS, a to opravdu velmi přehledně a názorně.

Tipy a triky při tvorbě rozkladu procesů

1. Dobrou pomůckou pro lepší komunikaci se zákazníkem je raději používat namísto „rozklad procesů“ synonymum „rozklad oblastí řešení“.

2. Představme si, že bychom měli dopředu napsat uživatelskou příručku evidenčního systému. Názvy kapitol odshora dolů mohou poté napomoci nalezení rozkladu, protože samy reprezentují možný rozklad oblastí řešení.

3. Představme si, že bychom měli u evidenčního systému navrhnout celou aplikaci se vším všudy přes jedno velké „super menu“ aplikace. Rozklad menu (horní lišta, submenu, submenu submenu atd.) reprezentuje pomůcku pro možný rozklad.

4. Rozklad na horních úrovních není jednoznačný. Dva analytici by mohli provést rozklad jinak: Jeden by například rozložil danou úroveň na šest nižších úrovní, druhý na tři a tři a dal by ještě úroveň nad nimi apod. Pokud by však oba pracovali na stejném IS, museli by se dole sejít u stejných procesů a případů užití. K posouzení, která varianta rozkladu je lepší (když obě vedou ke stejnému cíli), platí jednoduché pravidlo: Lepší je ta varianta, která je bližší pohledu zákazníka.

5. Rozklad sice po jeho dokončení vypadá jako strom, ale při jeho tvorbě se skládá jako mozaika. V dané okamžiku se například ví hodně o spodních úrovních, něco známe pro změnu nahoře, ale ještě není jasné, kam co bude umístěno. *Diagram Rozkladu procesů*

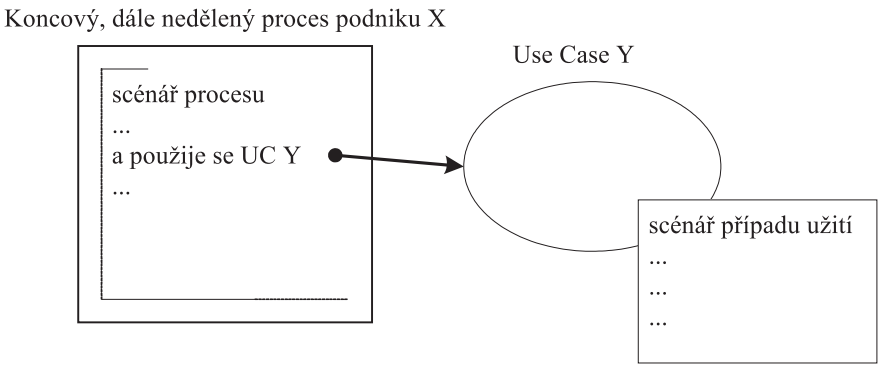
se tedy postupně skládá zespodu nahoru a shora dolů, dokud není celý hotov. Potom již vypadá jako hotový strom.

Ukončení rozkladu procesů

Abychom správně pochopili, kde nastal bod ukončení rozkladu procesů podniku, musíme si nejprve odpovědět na otázku, proč vlastně hledáme procesy a proč provádíme rozklad procesů. Cíl je jasný: Hledáme případy užití a na to musíme mít fokus. To znamená, že v okamžiku, kdy nalezneme proces podniku spouštějící případ užití, rozklad povinně končíme. Ukončovací proces rozkladu je ten, který spouští případ užití, tj. přímo v něm, v jeho scénáři se přímo hovoří o tom, že se tento případ užití použije, nebo jinak řečeno „případ užití se zavolá a spustí“. Jedná se tedy o již popsaný případ, viz obrázek 4.1, kdy na jedné straně je proces a ten spouští případ užití.

Pokud najdeme v jednom procesu více zavolání případů užití, pak lze tento proces ještě jednou rozložit tak, aby každý proces spouštěl právě jeden případ užití. Je to doporučený postup pro lepší systematický přehled. Mnohdy pak bývají procesy venku pouze „malou obálkou“ zavolání případu užití, například „Obsluha potřebuje zaevidovat novou fyzickou osobu v systému a použije se UC ‚Založení nové fyzické osoby‘ “ apod.

Je dobré si představit, že případ užití vypadá jako „tlačítko na systému“, které se nabízí ven k použití. Vnější proces venku provádí nějakou činnost, poté vnější děj přistoupí k tomuto tlačítku a to se stiskne. Tam končí popis procesů, protože další rozklad by vedl k popisu chování systému a to je vnitřek případu užití, neboli implementace jeho služby. V tomto bodě tedy rozklad končíme. Konečný proces, který se již dále nerozkládá, se nazývá Action (Akce). Této představě odpovídá obrázek 4.2.



Obrázek 4.2: Schéma konce rozkladu procesů.

Daná situace je velmi podobná popisu chodu funkcí. Představme si, že máme popsat chod funkce F1, která uvnitř sebe volá funkci F2. Začneme popisovat chod funkce F1, až dospějeme k bodu zavolání funkce F2. Popis chodu funkce F1 zde zastavíme (uvnitř funkce F1 nebudeme přece popisovat chod funkce F2, která je volána) a prostě napíšeme „a zavolá se F2“. Další popis chodu umístíme do funkce F2. Nyní si tento názorný příklad převedeme na situaci ukončení rozkladu u posledního procesu rozkladu (viz obrázek 4.2): Funkce F1 odpovídá procesu podniku „X“, funkce F2 odpovídá případu užití „Y“. Pokud bychom popisovali další chod děje na straně procesu (tj. pokračovali bychom v rozkladu procesů „venku“), dopustili bychom se stejné chyby, jako kdybychom v chodu funkce F1 popisovali chod funkce F2.

Tok děje u koncového procesu je tedy tento: Něco se děje venku, což popíšeme. V rámci tohoto děje se spustí případ užití, tam musíme popis procesu vně ukončit. Dále popíšeme děj uvnitř případu užití (program). Vnější popis tedy končí na stisknutí (zavolání, použití) případu užití informačního systému.

Chyba přílišné granulity rozkladu

Jako příklad této chyby bych uvedl následující nesprávný rozklad: Autor nejprve rozkladem procesů dospěl až k procesu „Příjem došlé faktury do podniku“ (úmyslně jsem tento proces podniku nazval dlouhým názvem, aby bylo patrné, co se v něm děje). Pokud by autor postupoval správně, měl by v této chvíli rozklad ukončit, zvolit tento proces jako Action a ve scénáři tohoto procesu správně vyvolat případ užití s názvem „Založení nové došlé konečné faktury v systému“. Správným řešením je tedy zde ukončit rozklad, protože jsme nalezli proces Action vyvolávající případ užití. Scénář tohoto procesu by v té chvíli vypadal velmi jednoduše:

```
Scénář BP „Příjem došlé faktury do podniku":

Do podniku přišla nová konečná faktura k proplacení od obchodního
partnera (například poštou). Obsluha tuto fakturu zadá do systému,
viz případ užití „Založení nové došlé konečné faktury v systému".
```

Tímto správným postupem rozkladu byl nalezen náležitý konečný prvek rozkladu a je dobře napsán scénář včetně vyvolání daného případu užití. Tak by to bylo správně. Představme si však, že namísto toho autor chybně pokračuje v rozkladu procesů ještě dál. Vychází mu proto rozklad daného procesu „Příjem došlé faktury do podniku“ (tj. ještě na straně *BPM*!) na další subprocesy. Rozkládá tento proces (kde měl již správně skončit) dále na subprocesy „Založení a editace hlavičky faktury“, „Založení řádků“ atd.

Je zřejmé, že se jedná o chybu, protože těmto subprocesům již neodpovídají případy užití, jak jsme si je vysvětlili a zavedli na počátku této kapitoly tímto mechanismem: *Nejprve se nic neděje, poté dojde k spuštění procesu vně, mimo systém, tedy v podniku, a vyvolá se případ užití.* Právě pro programátory bývá tato situace matoucí, protože nerozlišují mezi případem užití a funkcí. Rozdíl je právě v pojmu „požadavek, užitek a nutkání k použití systému“, což definuje jako v zrcadle případ užití ve smyslu „něco chci a něco dostanu“. Právě případy užití ve své podstatě definují ono „co chci a co proto dostanu“. V prvém správném řešení nastalo nutkání, požadavek k použití systému „musím založit novou fakturu v systému“. Vyvolaný užitek má svůj srozumitelný počátek a konec požadavku, jakousi obdobu „transakce užitku“. Tato „transakce užitku“ je však dána vnějším požadavkem „co chci“. Jednoduše řečeno, případ užití definuje odpověď na otázku „proč přistupujeme k systému a použijeme jej, jaký byl požadavek a jaký případ užití jej plní“. Například „založení hlavičky faktury“ není v tomto případě případem užití vyvolaným vnějším procesem, ale je to již součást scénáře již spuštěného případu užití, protože vnější požadavek zněl „založit fakturu“ a nikoliv „založit hlavičku“. Jinak řečeno, obsluha vyvolá užitek systému, protože potřebuje zaevidovat novou fakturu a nikoliv pouze hlavičku. Nejdůležitější na předešlé větě je slůvko „potřebuje“, s čímž mnohdy programátoři mají trochu problém, protože funkce nezná slovo „potřebnosti“.

Samozřejmě, pokud by existoval takový proces vně, který by vyvolal požadavek *pouze a jenom* na založení hlavičky (neumím si jej představit), tak by měl takový případ užití jako případ užití spouštěný zvně nárok na život. Znamenalo by to však, že by nastal tento zajímavý děj: Nejprve nikdo nic nechce. Venku se spustí nějaký proces „XY“, začne plynout jeho scénář (venku mimo systém), z jeho toku vznikne požadavek na zaevidování pouze

hlavičky faktury a spustí se případ užití „Založení hlavičky“ (a nic víc!). Je zřejmé, že právě vyhledávání takovýchto případů užití spouštěných požadavkem užitku zvně procesu, tedy vyhledávání událostí požadavku vně s následným nalezením transakce užitku neboli případu užití, dávají nejenom dobrou představu o všech nabízených případech užití, ale také vytvoří velmi dobrý obraz o *hranicích systému*. V laické představě můžeme tyto případy užití chápat jako „všechna nabízená tlačítka“, která náš naprogramovaný stroj nabízí. Vidíme, že chybný rozklad procesu dále na subprocesy například „Založení hlavičky a editace hlavičky faktury“ popisuje již součásti spuštěného případu užití, tj. dostali jsme se omylem za jeho hranici. V tomto případě je oprava chyby jednoduchá: Uvedený popis rozkladu za požadovaným bodem se přemístí z prostoru procesů „venku mimo systém“ do prostoru za hranice případu užití, tj. dovnitř systému, protože se jedná o jeho řešení (co dělá systém) a problém je takto vyřešen.

Jedná se sice o chybu, ale je třeba upozornit na tu skutečnost, že na začátku prací na rozkladu se skutečně mnohdy „rozklad přežene“ a začnou se popisovat děje, které již patří za tlačítko „dovnitř systému“, a neměly by proto být součástí rozkladu procesů. Důvodem je totiž to, že v té chvíli ještě není patrna hranice systému. Pro úplně úvodní práce na projektu to při komunikaci se zákazníkem nevadí, protože zákazníkovi se popisuje souběžně s rozkladem i část funkcionality systému, což je ochoten přijmout a na počátku prací to může být i výhodou. Při dalším kroku zavedení případů užití však musí již být tyto procesy přebytečného rozkladu přemístěny do popisu případu užití, tj. přesunuty z „venku“ „dovnitř“.

4.3.2 Diagram Podpory IS

Dalším artefaktem, který analytik odevzdává, je vztah mezi nalezenými procesy na konci rozkladu a zavedenými případy užití. Cílem je tedy znázornit, který daný proces typu Action spouští který daný případ užití. Některé CASE nástroje (například SCA) mají možnosti tuto vazbu zavést pomocí vnitřní evidence, EA tuto evidenci přímo nepodporuje. Za tím účelem vám jazyk UML umožňuje použít vztah Dependency, viz obrázek 4.1. U každého listu rozkladu se tedy zavede tato vazba mezi procesem a případem užití.
Poznámka: *Ve white-papers nástroje EA se také zavádí vztah Dependency mezi procesem podniku Action a případem užití pro popis toho, který případ užití souvisí s kterým procesem podniku. Avšak autoři EA navrhují zobrazit tento vztah obráceně ve směru od případu užití k procesu s odůvodněním, že „případ užití tzv. realizuje proces“. Osobně se domnívám, že přesnější a hlavně logičtější je namalovat tento vztah ve směru od procesu k případu užití, protože proces je ten prvek, který spouští a tedy používá případ užití a nikoliv naopak.*

Doporučené uspořádání případů užití a procesů pomocí prvků typu *Package*

Je vhodné, aby se případy užití včetně *Use Case diagramů* a prvků typu *Actor* zakládaly do zvláštního prvku *Package* a nebyly tak založeny do prvku *Package*, kde jsou umístěny procesy. Důvodem je skutečnost, že vývojáři (technologové a programátoři) čekají na případy užití. Samotné procesy je nezajímají, považují je za omáčku a balast. Prvky *Use Case* se tedy zakládají bokem ve svém vlastním prvku *Package* a přenášejí se do diagramů v prvku *Package BPM* metodou „drag and drop“.

Doporučuji také uvnitř tohoto prvku typu *Package* případy užití uspořádat do menších prvků typu *Package* a nedávat je všechny na jednu hromadu. Za tím účelem je ideální vycházet přesně ze struktury rozkladu procesů v *BPM*, tj. rozložit prvky *Package* případů užití přesně podle rozkladu procesů *BPM*, aby rozklad prvků *Package*, který uspořádává případy užití, odpovídal přesně rozkladu *BPM* až po poslední prvek *Package* obsahující

pouze případy užití. Vzniknou tak dva souběžné rozklady: Rozklad *BPM* a rozklad *Package* v případech užití, oba mají až po poslední prvek *Package* v rozkladu stejnou strukturu.
Poznámka: *V balíčku případu užití je ještě navíc jeden nebo vícero balíčků pro prvky typu* Actor *a pro interní případy užití, které vznikají až v dalším kroku zpracování, nikoliv jako případy užití, které jsou spouštěné procesy – ty jsme ještě nebrali.*

4.3.3 Diagram Chodu procesu

V mnohých případech je třeba doplnit procesní model o chod procesu neboli slangově o tzv. „vývojový diagram procesu".

Myšlenka tvorby *Diagramu Chodu procesu* je jednoduchá: Vyšší proces se nejenže rozkládá do nižších procesů, ale současně mezi těmito nižšími procesy je patrná i logika chodu procesů, tj. jak jdou po sobě, jejich chod a větvení, rozhodování atd. Tento diagram je zapotřebí vyhotovit zejména u složité logiky chodu procesů v podniku a samozřejmě také tam, kde se tento diagram přímo vyžaduje od zákazníka.

Pro další výklad je třeba si prostudovat syntaxi procesního modelování. Nástroj EA podporuje dvě školy procesního modelování, klasickou Eriksson-Penkerovu školu a tzv. *Business Process Modeling Notation* neboli *BPMN*.

I když jsem dlouho používal klasickou Eriksson-Penkerovu školu, doporučuji raději používat novější školu *BPMN*. Syntaxe *BPMN* je jednak celosvětovým standardem a kromě toho není složitá a je srozumitelná i laikovi. Navíc přechod od jedné školy ke druhé není vůbec násilný, protože obě dvě vycházejí z jednoho diagramu UML – z *Activity diagramu* a liší se pouze použitými stereotypy.

Celou syntaxi *BPMN* najdete na oficiálních stránkách `http://www.bpmn.org`, kde naleznete také poslední verzi specifikace [3]. V době psaní této knihy byla poslední oficiální verzí *BPMN 1.2* a Beta verze *BPMN 2.0*. Doporučuji mít specifikaci *BPMN* po ruce, tedy stáhnout si ji z uvedené pozice. Samozřejmě ji můžete prostudovat celou, pro praktickou potřebu analytika doporučuji dobrou znalost kapitol 8 a 12, která by měla být od analytika vyžadována.

Pokud používáte EA, prostudujte si také kapitolu *BPMN* v Helpu tohoto nástroje. Sice ve srovnání se specifikací *BPMN* nástroj EA nepodporuje všechny prvky, ale pro techniku nalézání případů užití je výčet prvků pro tvorbu *BPMN* diagramů v EA dostatečný. Vyjmenujme si a popišme si ty prvky z *BPMN*, které budete určitě potřebovat.

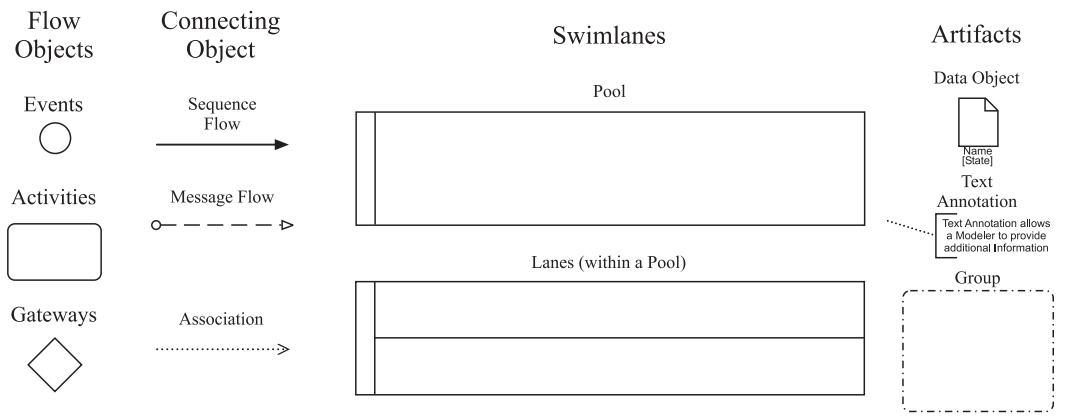
Základní elementy *BPMN*

Základní prvky *BPMN* znázorňuje obrázek 4.3.

Activity
: *Activity* je činnost, která je prováděna v rámci podnikových procesů, a je reprezentována obdélníkem se zaoblenými rohy.

Event
: *Event* je jev, který se stane v průběhu obchodního procesu a má vliv na pořadí a načasování činností procesu. Události jsou reprezentovány jako malé kruhy s různými hranicemi, lze rozlišit povahu události, například událost zahájení akce jako tenká černá linka, akce uprostřed jako dvojitá čára a akce na konci silnou černou čarou.

Gateway
: *Gateway* se používá k ovládání sekvence toků, tj. sloučení, větvení atd. v rámci chodu



Obrázek 4.3: Základní typy prvků *BPMN*. (zdroj: [2])

procesu. *Gateway* se dále ještě dělí podle povahy co vyjadřuje, může například představovat rozhodnutí, paralelní spuštění („vidlička“), spojení, resp. synchronizaci atd. Pro *Gateway* typu větvení je třeba u každého *Sequence flow* z něj vycházejícího definovat také podmínku typu boolean, která podobně jako switch slouží k určení, která větev chodu procesů ve větvení nastane.

Sequence flow

Sequence flow se používá pro zobrazení pořadí, v jakém jsou prováděny činnosti v rámci procesu. Zobrazuje se jako šipka s plnou čarou.

Message flow

Message flow se používá pro zobrazení toku zprávy mezi dvěma subjekty. Prvek je reprezentován přerušovanou čarou, u zdroje zprávy se umístí kruh, u příjemce zprávy šipka.

Group

Group se používá ke sdružování informací a artefaktů, maluje se čerchovanou čarou.

Pool

Pool reprezentuje entitu nebo roli, která je spjata s procesy a „pod níž“ nebo v „jejíž kompetenci“ se činnosti odehrávají.

Lane

Lane se používá pro specifické vymezení oblastí chodu procesů v rámci daného prvku *Pool*. Prvky *Pool* a *Lane* vycházejí z přirovnání plaveckých drah v bazénu. Umožňují umísťovat činnosti do „plaveckých drah“ – „pruhů“ podle nějakého kritéria.

Data object

Data object nemá přímý vliv na proces, zobrazuje pouze informaci, která se nějak týká procesu (například artefakt, formulář, oběžník apod.). Je prezentován jako obdélník s přeloženým horním rohem jako dokument.

Text Annotation

Text Annotation se používá jako volný text a obsahuje nějaké vysvětlení k diagramu, resp. jeho částí nebo k prvkům.

Sledujte aktuální novinky na http://www.objects.cz

Tipy a triky pro nalezení chodu procesu

Pro odhalení chodu procesu doporučuji sice jednoduchý, ale velmi účinný trik. Protože nižší procesy, které jdou ve své logice vývojového diagramu nějak po sobě, skládají horní proces, je zřejmé, že případný slovní popis horního procesu musí sedět s diagramem chodu procesů, které jej skládají. Tuto myšlenku lze otočit. Nejprve popišme scénář chodu horního procesu jednoduše slovy. Poté tento scénář zpracujeme tak, že jej dále rozdělíme na odstavce, resp. části odstavců („parsujeme“ jej) a z těchto částí necháme vytvořit nižší procesy v *Diagramu chodu procesu*. Všude, kde se objeví slovo „pokud“ nebo „jestliže“ apod., nasadíme prvek větvení.

Druhým tipem je rada „soustřeďte se na scénáře“. Celá logika slovních popisů scénářů procesů a následně logika chodu případů užití musí vcelku sedět. Diagramy jsou pouze následným grafickým vyjádřením dobře zpracovaných scénářů.

Závěrečné shrnutí postupu procesního modelování

Shrneme si poznatky do praktického postupu:

1. Hledejte rozklad procesů jako oblasti řešení.

2. Snažte se ukončit rozklad tam, kde jste narazili na případy užití, tj. kde se viditelně v popisu nalezeného procesu spouští „něco“ v našem systému. Dál už procesy zásadně nerozkládejte!

3. K těmto koncovým dále neděleným procesům přiřaďte případy užití. Cílem je získat obrázek 4.1, kde nalevo v procesu je popis chodu venku, napravo v případu užití „je program“. Nezapomeňte na možnou chybu „přehnaného rozkladu“, v tom případě rozklad upravte, protože má končit o jednu nebo více úrovní výše.

4. Chod procesu získáte nejjednodušeji tak, že slovně popíšete horní proces a poté jej „rozparsujete“ na části do *Diagramu chodu procesu*. Tato metoda vypadá sice pracně, ale je spolehlivá a mnohdy rychlejší, než zdlouhavé hloubání přímo nad modelem a diagramy v *BPMN* s následnými opravami logických chyb.

5. Pokud je vaším cílem návrh IS (tj. jste analytici IS), mějte na paměti, že cílem vaší práce je najít případy užití, na které čekají technolog a programátoři. Pokud tedy není zákazníkem vyžadován detailní *BPM* model s dalšími doprovodnými prvky, „nechoďte do přílišných detailů“, zobrazení procesů vám slouží pouze a jen k tomu, aby případy užití odpovídaly chodu podniku.

4.4 Postupy popisu vnitřků případů užití

V předešlé kapitole jsme si ukázali, jak se hledají případy užití, které jsou vyžadovány od okolí. Současně s tím však začínáme mít také představu, co se děje uvnitř případu užití, tedy jaký děj v programu proběhne a tím případ užití přinese pro okolí užitek. Nyní tedy přistoupíme k druhému kroku a tím je popis vnitřku případu užití. Mohli bychom také tento postup popisu vnitřku případu užití nazvat „popisem implementace případu užití“ stejně, jako když popisujeme funkci, resp. metodu objektu a její implementaci, tj. vnitřní průběh kódu. Existuje několik škol, jak popisovat vnitřek případu užití. Uveďme jen ty nejčastěji používané.

Sledujte aktuální novinky na http://www.objects.cz

Žádný popis vnitřku případů užití

Tato škola vychází z následující myšlenky: „Nic – nepopisujte nic, vaše práce již skončila“. Byla preferovaná v počátcích vzniku jazyka UML a dnes se již nedoporučuje. Jednak se vynecháním popisu vnitřku případu užití připravujeme o skvělou příležitost, jak předat z analytického modelování do technologie algoritmy programu a také hrozí vážné nebezpečí, že v logice celého chodu procesů podniku a programů, které jej podporují, budou uschovány nebezpečné logické protimluvy.

Popis vnitřků případů užití pomocí *Activity Diagramu*

Chod programu uvnitř případu užití lze dobře popsat pomocí *Activity Diagramu*. Nyní však již nejde o popis chodu procesu jako v předešlé kapitole, ale jde o popis chodu programu, tedy o popis činností uvnitř systému. Jedná se již o aktivity samotného programu a nikoliv okolí, tedy jde již o „čistý“ *Activity Diagram*. Prvky typu *Activity* se proto neoznačují stereotypem «business process».

Je třeba upozornit na tu skutečnost, že díky možnosti předešlého postupu existuje i druhá technika zpracování analytických dokumentů, která není založena na případech užití, ale je celá postavena pouze na prvcích typu *Activity*. Osobně ji nedoporučuji, protože většinou vede k nepřehledným diagramům. Její princip je následující: Aktivity uvnitř systému navazují na aktivity venku mimo systém a oboje lze znázornit pomocí jednoho *Activity Diagramu*. Vnějšek a vnitřek se oddělí technikou *Poolu* a *Lane*. To však znamená, že bychom namalovali jeden *Diagram aktivit* „promíchaný“ se všemi aktivitami vnějšími i vnitřními a pomocí prvků typu *Pool* a *Lane* je pak členili na „*activity* venku“ a na „*activity* systému“. Dá se tak vyjádřit ta skutečnost, že „vnější aktivita“ ve „vnějším *Poolu*“ spouští „vnitřní aktivitu“ uvnitř systému. Nepoužívají se tedy případy užití a vše je vyjádřeno jako činnosti, pouze se pomocí *Poolu* a *Lane* odděluje „činnost venku“ a „činnost uvnitř“. Tato technika je samozřejmě možná, ale nemám s ní dobré zkušenosti.

Ověřil jsem si, že z hlediska náhledu na systém a následného předání do technologie je lepší oddělit od sebe vnější chování a chování systému pomocí tohoto schématu: Vnější aktivita, tedy proces podniku, spouští případ užití a teprve případ užití má implementaci jako spuštěnou sekvenci „vnitřních aktivit“. Chování vnitřku případu užití je na rozdíl od techniky *Poolu* „zabaleno do případu užití“ a tak odevzdáno do technologie. V tomto trochu jiném pohledu vše funguje tak, že z vnějšího pohledu systém nabízí případy užití a ty mají vnitřní chování vyjádřené pomocí *Activity Diagramu*. Popis vnitřku případu užití pomocí *Activity Diagramu* je samozřejmě z hlediska přístupu UML úplně čistý, má však velkou praktickou nevýhodu oproti následující škole slovních scénářů: Máme k dispozici sice diagramy v UML, ale nikoliv texty scénářů, které jsou použitelné v projektu nejenom pro vývoj, ale i v jiných částech dokumentace projektu. To se jeví jako velká nevýhoda, protože texty jsou pak použitelné „všude možně v projektu“. Není však na škodu zkombinovat následující školu slovních scénářů s technikou *Activity Diagramu* a dodat jak texty, tak i *Diagram aktivit*.

Popis vnitřku případů užití pomocí slovních scénářů

Dnes velmi preferovaným postupem je popsat vnitřky případů užití slovním scénářem. Jak bylo řečeno u předešlé školy, scénáře mohou být doplněny navíc o *Activity Diagramy* znázorňující scénáře graficky. Budeme se věnovat blíže této škole slovních scénářů.

4.4.1 Zásady psaní scénářů případů užití

V této kapitole si uvedeme doporučené postupy pro psaní scénářů, tipy a triky.

Jednoduchost a srozumitelnost, žádné vysvětlování

Scénáře popisují jednoduše, jednoznačně a srozumitelně chod programu. Je třeba se vystříhat nějakého vysvětlování. Pokud chceme něco vysvětlit, umístíme toto vysvětlení do přílohy. Programátor chce dostat do ruky zadání, co se žádá programovat a nic víc. Například pokud scénář začne slovy: „V podstatě jde o to, že...“, tak to už je špatně.

Žádná synonyma

V textu scénáře jsou zakázána synonyma. Jestliže nějakému prvku dáte jednou určitý název, potom tento prvek musíte vždy takto nazvat. Příklad: „Obsluze se zobrazí seznam klientů, obsluha vybere zákazníka...“. To je špatně, protože pokud měl autor na mysli totéž, potom je čtenář zmaten.

Ustálená slovní spojení – *Scenario Patterns*

V textu scénářů byste měli používat ustálená slovní spojení, neboli *Scenario Patterns*. Jedná se o opakující se analytické situace, kdy se opakuje scénář, ale mění se účastníci scénáře. V evidenčních systémech se jedná o klasické situace práce obsluhy, například zobrazení a výběr ze seznamu, editace apod. Firmy si většinou tyto vzory tvoří samy podle svých potřeb. Příklad jednoduchého vzoru scénáře:

```
Část editace <A>:

Obsluze se zobrazí seznam <B> (<B1>, <B2>, ...), obsluha vybere <B>
a vybrané <B> se prováže do editovaného <A>.
```

Příklad použití tohoto vzoru pro scénář „výběr Barvy do Auta“:

```
...

Obsluze se zobrazí seznam Barev (kód, text), obsluha vybere Barvu a
vybraná Barva se prováže do editovaného Auta.

...
```

Popis chování na straně systému

Při popisu scénáře evidenčního systému se většinou píše, co provádí v ovládání programu obsluha (obsluha zadá, obsluha vybere atd.) a na straně druhé se popisuje, co se odehraje na straně systému (kontroly vstupu apod.). Existují dvě možnosti, jak popisovat chování na straně systému:

- *Klasický způsob popisu, co dělá systém.*

  V tomto případě se chování na straně systému popisuje větou s podmětem = „systém“, například: „Obsluha zadá částku, systém zkontroluje modulo 11...“.

Sledujte aktuální novinky na http://www.objects.cz

- *Použití zvratného slovesa pro popis, co dělá systém.*

  V tomto případě se chování na straně systému popisuje větou se zvratným slovesem, například: „Obsluha zadá částku, zkontroluje se modulo 11...“. Tento způsob má blíže k objektovému náhledu, první klasický způsob implikuje představu volání systémových funkcí. Pokud však použijete zvratná slovesa, musíte mít jistotu, že čtenář scénářů (může být i laik) musí dobře chápat, že zvratné sloveso znamená „uvnitř systému“ a že tak nečiní obsluha mimo systém například na kalkulačce.

Vzor pro ošetření chybových stavů – *Happy Scenario* alias *Exception Flow*

Analytik musí ve scénáři popsat i to, co se odehraje, když nastane chybový stav například na vstupu. Doporučuje se použít vzor *Happy Scenario* nazývaný také jako vzor *Exception Flow*. Ošetření mezního chybového stavu se popíše jako odskok podobný odskoku na návěstí podle následujícího schématu:

```
...
<algoritmus podmínky, která se vyhodnotí>
Pokud není splněno, tak <návěstí scénáře>
```

Návěstí scénáře je označení textu mimo scénář (například na konci pod čarou) s dvojtečkou, například takto:

```
CHYBA1: text...
```

V nástroji EA lze tento text umístit do vedlejšího scénáře a označit ho jako typ scénáře *Exception Flow*. Příklad použití tohoto vzoru:

```
...
Obsluha zadá číslo účtu a vybere kód banky.
Zkontroluje se modulo 11.
Pokud modulo 11 nesouhlasí, pak CHYBA1.
...
-----------
CHYBA1: Obsluha se upozorní a neumožní se pokračovat dále.
```

V předešlém příkladu byla použita technika „čára na konci scénáře“. Pokud používáte EA, můžete zavést tento scénář jako jiný scénář tohoto případu užití s názvem CHYBA1, jako typ scénáře mu zadejte *Exception Flow* (POZN.: *v nástroji EA lze typy scénářů administrací měnit, resp. přidávat*). Smyslem vzoru *Happy Scenario* je umístit ošetření chybových stavů mimo hlavní scénář, kterému se také říká výstižně *Happy Scenario*, protože končí „happy endem“.

Doporučení: nepoužívat techniku Alternate flow

Jednou z technik psaní scénářů, kterou nedoporučuji, je technika Alternate Flow. Jedná se o osobní zkušenost získanou na základě konzultací v různých firmách, ale samozřejmě můžete a nemusíte toto doporučení zohlednit. Technika se nazývá Alternate Flow, umožňuje zavést větvení ve scénáři, které funguje takto: V hlavním scénáři se označí odstavce

čísly, například 1., 2. atd. případně i strukturovaně 1.1, 1.2, 2.1 atd. Následně se definuje tzv. alternativní scénář pro některý z odstavců tak, že se definuje podmínka, za které tento alternativní scénář platí. Pokud je tato podmínka splněna, tak tento alternativní scénář „nahrazuje“ původní část scénáře s daným číslem stejně, jako by ho „vytlačil“ z původního scénáře a nahradil jej. Považuji tento způsob psaní scénářů za poměrně nepřehledný, zejména pokud je takovýchto „vytlačení“ více.

Přehledné větvení – technika switch

Doporučuji také nepoužívat klasickou konstrukci větvení „Pokud ... jinak ...“, ale každou větev z větvení uvést vlastní podmínkou podobně, jako to dělá switch anebo větvení v *Activity Diagramu*. Čtenáři usnadníte pochopení logiky větvení. Snažte se vždy o srozumitelnost větvení. Příklad:

```
Pokud X > 0, pak:
  ...
  ...
Pokud X ≤ 0, pak:
  ...
```

Trik s «include»

Při velmi nepřehledných větveních s dlouhými scénáři mezi větvemi si můžete vypomoci interakcí «include», kterou si vysvětlíme v další kapitole.

4.5 Interakce mezi případy užití

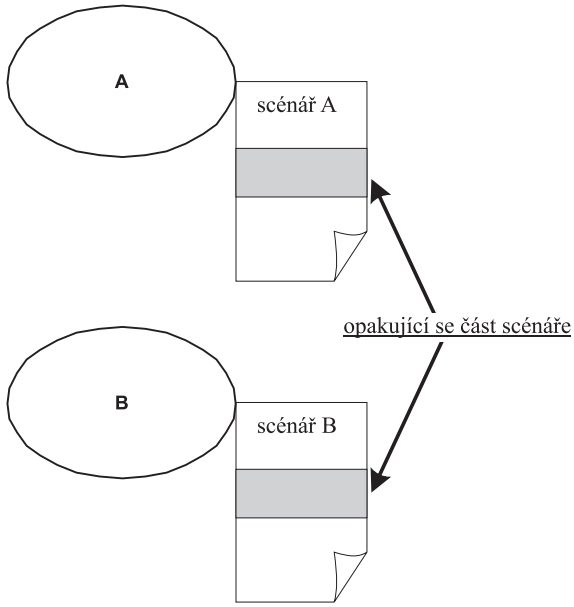
Základní princip, proč existují interakce mezi případy užití, je nám dobře znám: Opětovná použitelnost neboli re-use. Existují však i jiné „podružnější“ důvody použití interakcí mezi případy užití. Jazyk UML zavádí celkem tři druhy interakcí mezi případy užití, nyní si je popíšeme.

4.5.1 Interakce «include»

Představme si situaci, že jsme předešlými postupy nalezli případ užití a nazvěme jej „A“. Napíšeme k němu scénář podle pravidel uvedených v předešlé kapitole, a zjistíme, že scénář je v pořádku a jsme s ním spokojeni. Následně nalezneme druhý případ užití podle předešlých postupů a pojmenujme jej například „B“. Začneme psát jeho scénář podle doporučení z předešlé kapitoly a najednou dostaneme silný pocit „deja vu“. Tuto část scénáře jsme už někde psali. A opravdu, rozborem zjistíme, že část scénáře v případu užití „A“ a část v případu užití „B“ se opakuje, například: „Obsluze se zobrazí seznam fyzických osob, obsluha vybere osobu“. Zobrazíme si tuto situaci schématicky obrázkem 4.4.

Tuto situaci dobře známe z prvních kapitol: Je to situace „před vytknutím“ při aplikaci principu opětovné použitelnosti neboli re-use. Pokud tato situace nastane, jsme povinni část opakujícího se scénáře vytknout do nového případu užití pomocí interakce zvané «include». Poznámka: *Samozřejmě tuto situaci odhalíme pouze v případě, když dodržujeme zásady psaní scénářů: žádná synonyma a používat pouze ustálená slovní spojení.*

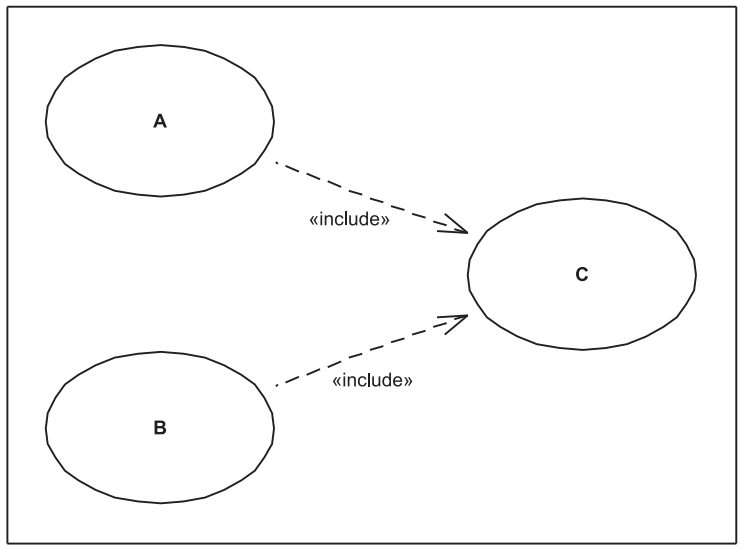
Při zavedení interakce «include» postupujeme takto:



Obrázek 4.4: Oba scénáře obsahují opakující se část.

1. V diagramu zavedeme nový případ užití „C“ a provážeme dvakrát interakcí «include» ve směru od „A“ k „C“ a od „B“ k „C“, viz obrázek 4.5.

2. Současně však provedeme i úpravy v textech scénářů: Opakující se část scénáře umístíme do případu užití „C“ a na původních místech ve scénářích A a B použijeme větu vyjadřující „volání“ případu užití „C“, například „použije se případ užití ‚C‘ “, nebo „zavolá se případ užití ‚C‘ “ apod.



Obrázek 4.5: Zavedení operace «include» v diagramu.

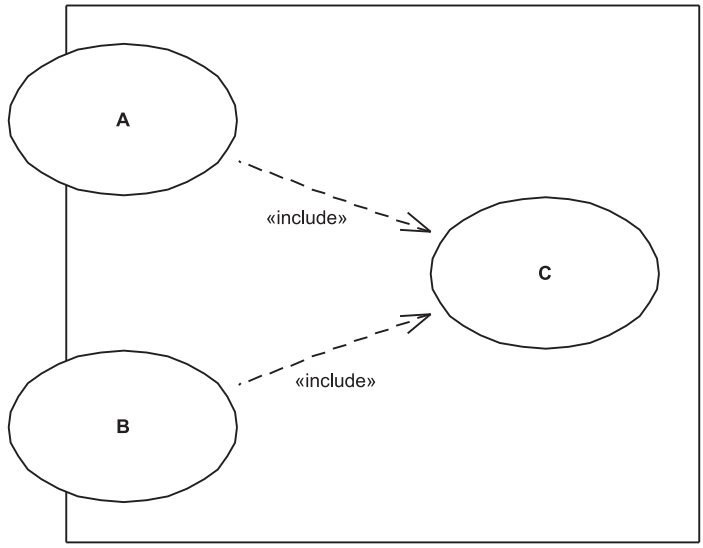
Jak je vidět, interakce «include» je obdobou volání funkcí v klasickém programování a přesně tak je i přirovnávána v dokumentu specifikace jazyka UML.

Interakci «include» můžeme zavést i z jiného důvodu než z důvodu řešení opětovné použitelnosti. Pomocí této interakce můžeme zpřehlednit dlouhý scénář, resp. složité větvení tím, že části scénáře umístíme mimo tento případ užití a „zavoláme“ je pomocí interakce «include» podobně jako u dlouhých funkcí v programování (viz kapitola 4.4.1 na straně 113).

Úskalí při nalézání případů užití a při interakci «include»

Použití interakce je myslím zřejmé, ale skrývá v sobě úskalí: Všimněme si, že před vytknutím jsme měli 2 případy užití, po vytknutí máme 3 případy užití. Počet případů užití se evidentně zvýšil. Zeptejme se však: Zvedla se užitkovost systému? Odpověď zní samozřejmě, že ne. Pouze došlo k „přeskupení scénářů“, tedy k restrukturalizaci případů užití, k jakési obdobě „refaktoringu“. V této skutečnosti se skrývá často rozšířená chyba, která se týká nalézání případů užití a vztahu «include». Je patrné, že v prvním kroku, když pracujeme s procesy podniku, hledáme ty „užitkové“ případy užití, které jsou spouštěné z procesů, teprve v druhém kroku nalézáme „restrukturalizační“ případy užití, které vznikají vytknutím scénářů. Podle jejich vzniku tedy rozlišujeme dva typy případů užití: Ty, které hledáme v prvním kroku jako „splněné požadavky procesů = případy užití spouštěné z procesů“, a oproti tomu existují druhé případy užití, které vznikají „vytknutím“ z jiných případů užití.

Případy užití nalezené v prvním kroku jsou charakteristické tím, že jsou využity zvenku z procesů podniku, druhé jsou charakteristické tím, že jsou spouštěny zevnitř, tj. uvnitř systému jinými případy užití. Mnohdy má zejména bývalý programátor problém s tímto náhledem na systém, protože programátor syntetizuje program a „skládá“ jej zespodu, kdežto analytik analyzuje, tedy rozkládá jej dovnitř.



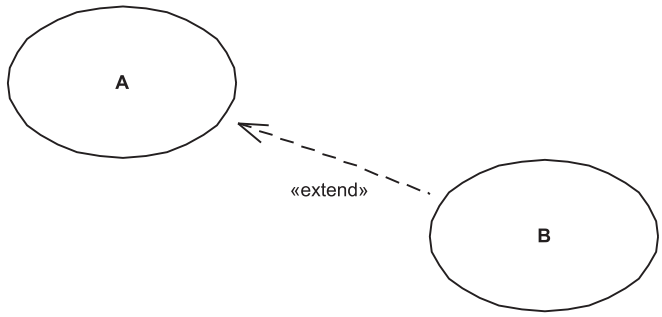
Obrázek 4.6: Z vnějšího pohledu pouze dva případy užití.

Na obrázku 4.6 jsou rámečkem vyznačeny hranice systému. Případy užití „A“ a „B“ jsme symbolicky vyznačili jako „ven mimo rámeček vyčuhující případy užití“, tedy jako „tlačítka na systému“. Ten, kdo je venku, vidí tyto dva případy užití. Tak to také navrhneme v počátečních krocích analýzy. To, že uvnitř je nějaký třetí případ užití „C“, nás v prvním kroku

vůbec nezajímá. K vytknutí dojde až ve druhém kroku. Je proto dobré rozlišovat případy užití nalezené v prvním kroku (zvyšují užitkovost systému) a případy užití nalezené v druhém kroku vytknutím pomocí interakce «include» (provádějí pouze přeskupení, nezvyšuji užitkovost systému).

4.5.2 Interakce «extend»

Interakce «extend» je tak trochu zvláštní a v použití dost výjimečná. Na obrázku 4.7 je směr interakce úmyslně namalován od „B“ k „A“.



Obrázek 4.7: Interakce «extend».

Popis vlastností a použití interakce «extend» podle obrázku 4.7:

1. Klient těchto dvou případů užití přistupuje a spouští „A“, nikoliv „B“. Takže zvně se volá „A“, nikoliv „B“.

2. Případ užití „B“ tzv. „extenduje“ neboli rozšiřuje případ užití „A“. V každém případu užití lze zavést jeho vlastnost tzv. Extension Point (může jich být několik). Extension Point se zavádí mimo scénář jako zvláštní prvek u případu užití. Pod tímto bodem si představte něco jako „vystavený háček“, ke kterému se může přivěsit jiný rozšiřující případ užití. Daná interakce „extend“ se přiřadí k danému prvku Extension Point a může být za určité podmínky vyvolána. Vyjadřujeme tak tu situaci, kdy rozšiřující případ užití („B“) rozšiřuje rozšiřovaný případ užití („A“) o svou funkcionalitu zavěšením k danému bodu extenze a spustí se za určité podmínky.

Ale buďme opatrní na následující chybnou záměnu: Rozšíření pomocí vztahu «extend» není podmíněné «include», jak by se mohlo na první pohled jevit. Pokud se ve scénáři objeví volání jiného případu užití za určité podmínky v daném bodě scénáře (tj. obdoba `if`), nepoužijeme v tom případě interakci «extend», ale interakci «include» podle tohoto schématu:

```
...pokud <podmínka>, potom se použije případ užití „B"...
```

Tato situace volání za podmínky se maluje jako «include», nikoliv jako «extend», protože případ užití volající má ve své implementaci (ve scénáři) zmínku o případu užití volaném, tj. ve scénáři případu užití „A“ se vyskytuje volání případu užití „B“. Oproti tomu interakce «extend» nám umožňuje řešit situaci, kdy rozšiřovaný případ užití (zde případ užití „A“) „neví“ o rozšiřujícím (zde případ užití „B“) a o rozšiřujícím případu užití není tedy ve scénáři žádná zmínka. Případ užití „A“ nabízí pouze bod extenze a k němu lze připojit jeden nebo více rozšiřujících případů užití. Rozšiřující případ užití nabízí „něco navíc“, zatímco interakce «include» představuje přímé zavolání.

Příklady použití interakce «extend»:

1. Bonusové případy užití vyvolané nastalou situací, například Nabití karty + nabití s bonusem. Případ užití „Nabití s bonusem“ se však připojil navíc jakoby dodatečně, tj. v původním scénáři nabití karty není o nabití bonusu žádná zmínka.

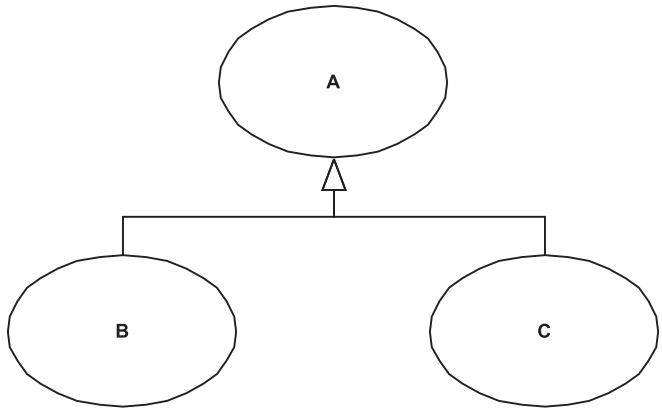
2. Vyvolání Helpu aplikace – případ užití běží s podporou Helpu nebo bez ní.

3. Vyvolání jiné akce uživatele „přiháčkované“ k danému případu užití (obdoba akcí na pravé tlačítko myši nad prvkem nebo textem apod.).

Při rozlišování interakcí «include» a «extend» bychom se měli držet této zásady: Pokud v daném scénáři „natvrdo“ oslovujeme v daném bodě jiný případ užití a voláme jej (i v případě, že se jedná o větvení typu `if`), potom použijeme interakci «include». Pokud je rozšiřující případ užití takzvaně „přivěšen“ a původní případ užití o přívěsku „neví“ (tj. není o něm zmínka ve scénáři), použijeme interakci «extend». Druhá situace „přívěsku“ je o mnoho méně častá než prvá.

4.5.3 Interakce generalizace mezi případy užití

Třetím typem interakce mezi případy užití je generalizace, kterou již známe z předešlých kapitol z modelu tříd. Jedná se o úplně stejnou interakci, protože tato interakce je v jazyce UML zavedena jak pro třídy, tak pro případy užití. Tato interakce byla již popsána v předešlých kapitolách. Vše, co bylo řečeno o vztahu *Generalizace* mezi třídami, platí i zde pro případy užití.



Obrázek 4.8: Vztah *Generalizace* mezi případy užití.

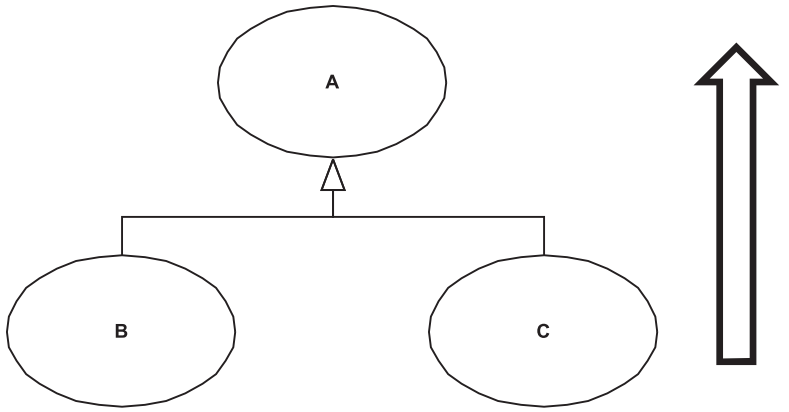
Připomeňme si vlastnosti vztahu *Generalizace* a aplikujme je na případy užití:

1. Pokud zavoláme případ užití „B“, pak díky dědičnosti bude mít vlastnosti případu užití „B“ + „A“.

2. Má smysl hovořit o tzv. abstraktním případu užití „A“, který slouží jen jako předloha pro generalizaci a jako případ užití nebude nikdy použit (přesněji nebude nikdy přímo instancován).

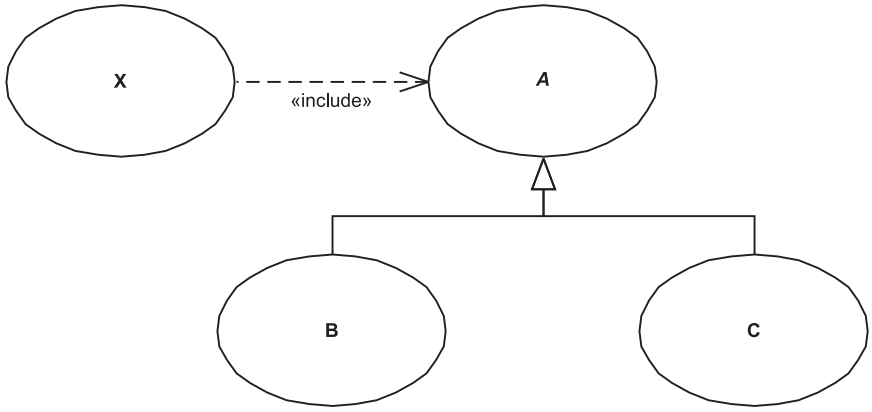
Ujasněme si nyní, k čemu se dá generalizace mezi případy užití použít. K tomu slouží další (dokonce nejdůležitější) vlastnost vztahu *Generalizace*. Nejdůležitějším rysem generalizace totiž nejsou první dvě vyjmenované vlastnosti, ale tzv. zástupnost rolí zespodu nahoru.

To znamená, že všude, kde se odvoláváme na případ užití „A“ (tj. „voláme jej ze scénáře“), může tuto roli sehrát libovolný z dědiců, zde případ užití „B“ nebo případ užití „C“. Tuto vlastnost zástupnosti (neboli kompatibility zespodu nahoru) si, podobně jako v modelu tříd, vyjádříme na obrázku 4.9 pomocí šipky zespodu nahoru.



Obrázek 4.9: Zástupnost rolí zespodu nahoru v generalizaci u případů užití.

Otázkou je, jak se uvedená vlastnost projeví v praxi. Všimněme si tohoto jevu: Ve scénáři se odvoláme na případ užití „horní“, tj. „A“, ale konkrétně se tam provede buď „B“ nebo „C“. Tento jev programátoři znají velice dobře a v návrhu IS vyjadřuje tzv. *polymorfní chování*. Uveďme si příklad kombinace interakce «include» s generalizací, viz obrázek 4.10.



Obrázek 4.10: Kombinace vztahů «include» a *Generalizace*.

Ve scénáři uvnitř X se napíše „provede se případ užití ‚A‘ “, ale konkrétně se vyvolá jeden z dědiců „B“ nebo „C“, kteří se v tomto bodě díky generalizaci mohou „zespodu“ dosadit. Ukážeme si dále, že laik tomuto efektu rozumí dobře, pokud scénáře napíšeme korektně a správně. Jak bylo řečeno, obrázek 4.10 je vyjádřením známého jevu z objektového programování, který se nazývá polymorfismus. Hovoříme o funkcionalitě A a přitom se konkrétně v tomto bodě vyvolá buď případ užití „B“ anebo případ užití „C“, oba totiž mají schopnost do bodu zavolání „zespodu vlézt“ díky velké šipce.

Generalizace se používá v modelu případů užití ve dvou základních vzorech (je možné, že v praxi narazíte na další). Tyto vzory také dobře vysvětlují, jak generalizace mezi případy

užití vlastně funguje, jaký má smysl a mimochodem ukazují, jak je jednoduchá a pochopitelná „laickým selským rozumem“.

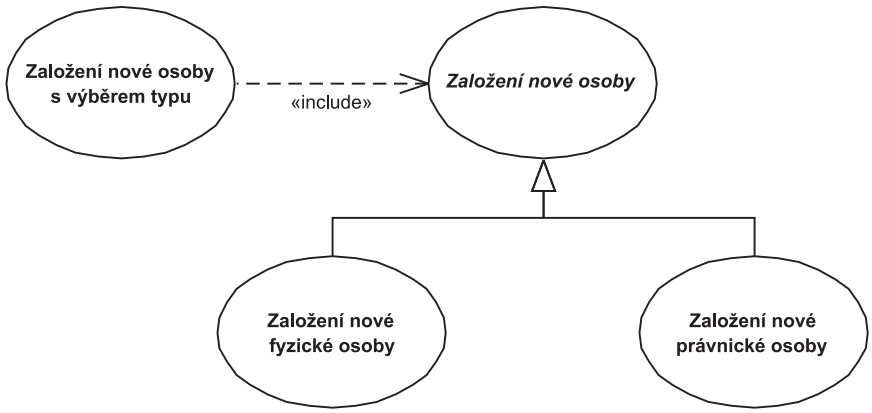
Vzor *Výběr algoritmu*

Jedná se o analogii vzoru *Strategy*, který je převeden do modelu případů užití. Jako úvodní příklad pro vysvětlení tohoto vzoru zaveďme případ užití (zatím vynecháme jeho název) s tímto scénářem:

> `Obsluze se zobrazí seznam typů osob. Obsluha vybere typ osoby. Dále se provede` ***`Založení nové osoby`*** `podle vybraného typu.`

Tento scénář je laikovi srozumitelný, dokonce on sám se takto vyjadřuje. Odvoláváme se na případ užití „Založení nové osoby“, ale ten nebude nikdy vyvolán konkrétně, budou vyvolány konkrétně případy užití podle vybraného typu, jako jsou „Založení nové fyzické osoby“, „Založení nové právnické osoby“ atd. Daný případ užití „Založení nové osoby“ slouží pouze jako zástupný pojem, tedy jako abstraktní případ užití, tj. jako předek v generalizaci. Zde je právě ukázkově uplatněna interakce generalizace, kdy jeden pojem „nahoře“ (případ užití předek) slouží pouze jako zástupný pojem pro jiné případy užití „dole“ (konkrétní dědicové). Mluvíme sice o případu užití „nahoře“, ale konkrétně máme na mysli jeden ze „spodních“ případů užití.

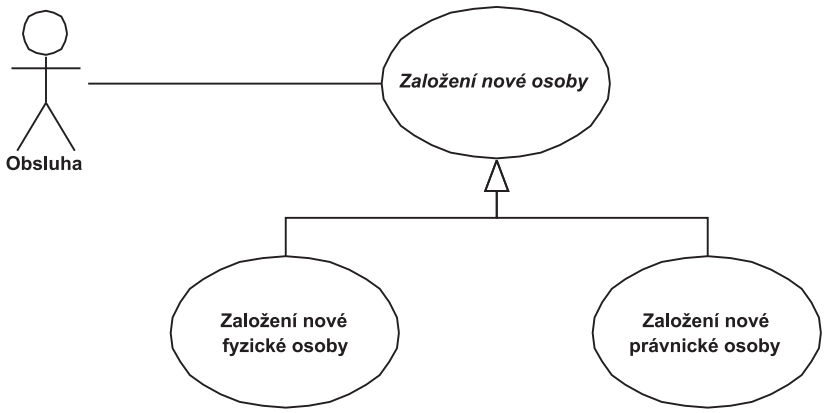
Nazvěme případ užití obsahující tento scénář jako „Založení nové osoby s výběrem typu osoby“ a můžeme potom namalovat diagram, který uvedenou myšlenku ve scénářích znázorňuje pomocí diagramu UML, viz obrázek 4.11.



Obrázek 4.11: Použití vzoru *Výběr algoritmu* pro založení nové osoby.

U řešení bez generalizace by se ve scénáři případu užití „Založení nové osoby s výběrem typu osoby“ objevila obdoba přepínače s interakcemi «include» takto:

> `Pokud obsluha vybere typ osoby Fyzická osoba, pak se zavolá „Založení nové fyzické osoby",`
>
> `Pokud obsluha vybere typ osoby Právnická osoba, pak se zavolá „Založení nové právnické osoby"`
>
> `atd.`



Obrázek 4.12: Zkrácená varianta zápisu téhož vzoru.

Někteří autoři diagram na obrázku 4.11 zjednodušují zkratkou a malují přímo interakci okolí s abstraktním případem užití tak, jak je uvedeno na obrázku 4.12.

V podstatě touto zkratkou činí dohodu, že vynechávají scénář výběru typu. I tento zápis je možný, je však nutné si uvědomit, že se bude programovat také výběr typu, což je třeba zohlednit například při odhadech pracnosti. Existuje druhá varianta tohoto vzoru, kdy výběr typu neprovádí obsluha, ale sám program, například při načítání (importu) údajů. Příklad scénáře pro zpracování zpráv s využitím vzoru *Výběr algoritmu*:

```
Postupně se načítají zprávy, u každé se zjistí typ a provede se
„Zpracování zprávy" podle zjištěného typu.
```

Princip je stejný, jako když výběr činí obsluha, pouze zde výběr činí sám automat, tedy program. Zavedou se dědicové případu užití „Zpracování zprávy“, podle schématu: „Zpracování zprávy typu X“, „Zpracování zprávy typu Y“, ... a další postup je již zřejmý.

Vzor *Reakce*

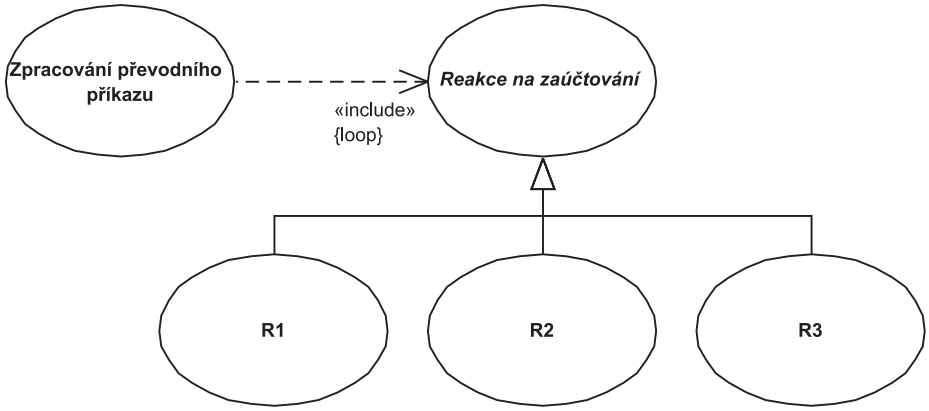
Jedná se o obdobu vzoru *Observer* (resp. *Listener*). V některých případech je třeba v určitém bodě scénáře upozornit na to, že v dané chvíli se na tento bod scénáře provedou reakce v okolních částech systému. Například při zaúčtování převodního příkazu v bance se vyvolají další funkcionality, které se při zaúčtování mají spustit. Klasickou (neobjektovou) možností je vyjmenovat tyto funkcionality v daném bodě scénáře přímo a zavolat je pomocí «include». Označme tyto funkcionality, tj. případy užití, jako „R1“, „R2“, „R3“ (R = reakce). Klasický scénář pak může vypadat takto:

```
...
//bod zaúčtování
provede se „R1"
provede se „R2"
provede se „R3"
...
```

Jinou možností je použít vztah *Generalizace*. Scénář se změní takto:

```
...
//bod zaúčtování
A dále se provedou všechny „Reakce na zaúčtování"
...
```

Takové vyjádření ve scénáři úplně stačí, nyní zbývá už jen specifkovat, co vše může být chápáno jako „Reakce na zaúčtování“ (které se zavolají v daném bodě) a co tedy bude zavoláno konkrétně. To vyjádříme pomocí vztahu *Generalizace* obrázkem 4.13.



Obrázek 4.13: Využití generalizace ve vzoru *Reakce*.

Na obrázku 4.13 jsme pomocí prvku „constraint“ vyznačili, že daná interakce «include» se volá v cyklu. Do pozice „nahoře“ postupně vstupují v cyklu jeden případ užití za druhým (dědicové „R1“, „R2“ a „R3“). Výhodou použití tohoto vzoru je to, že pokud se při dalším vývoji nebo skládání subsystému objeví další nová nutná reakce, jednoduše ji přidáme pouze jako dalšího dědice. V předešlém klasickém příkladu bez generalizace by to však znamenalo otevřít původní scénář volající reakce a přidat další reakci jako další řádek scénáře. Další výhodou tohoto vzoru je také to, že text scénáře je srozumitelný i laikovi. Slovnímu spojení „provedou se všechny reakce na zaúčtování...“ rozumí dobře, pouze potřebuje znát výčet, jaké konkrétní reakce mohou být v dané roli reakce. To mu ukážeme jako dědice dané reakce v diagramu.

4.6 Prvek typu *Actor*

V předešlých kapitolách jsme si ukázali, jak se hledají případy užití, které jsou využívány okolím. Ve výkladu jsme poté přistoupili i k popisu vnitřků případu užití, tj. k „popisu implementace případu užití“. Jako poslední prvek *Use Case Diagramu* nám zbývá vysvětlit prvek typu *Actor*.

Je třeba hned v úvodu připomenout, že pokud bychom postupovali při vyhledávání případů užití podle metodiky „aktérové školy“ (viz úvodní kapitola o tvorbě *Use Case Diagramu*), tak bychom se tímto typem prvku zabývali hned na počátku jako prvním. My jsme však použili pro vyhledávání případů užití procesní školu, a proto se jím zabýváme jako až posledním, doposud jsme jej totiž nepotřebovali.

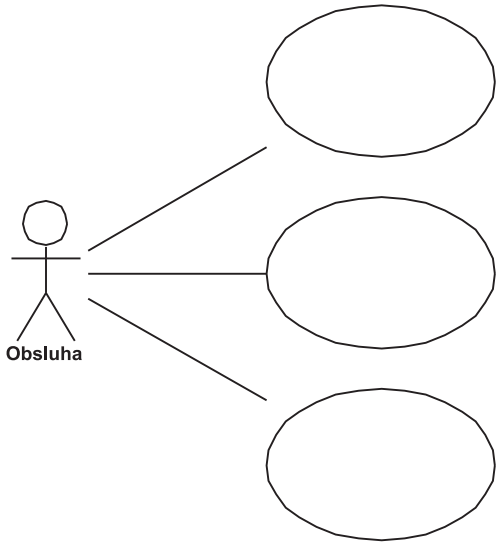
> *Doporučení: Pro vyhledávání případů užití jsou primárními procesy podniku a nikoliv prvky typu* Actor.

4.6.1 Definice prvku typu *Actor*

Prvek typu *Actor* reprezentuje prvek z okolí systému, který komunikuje se systémem, tj. konkrétně v *Use Case Diagramu* interaguje s některým z případů užití. Jeho implicitním obrázkem je „panáček s názvem“. Doporučuje se, aby se pro neživé okolní prvky (externí systémy) zavedl jiný obrázek než panáček (například obrázek PC) třebas mechanismem „zavést stereotype + výměna prezentačního prvku“, viz obrázek 4.14.

Obrázek 4.14: Zobrazení „živých“/„neživých“ prvků typu *Actor*.

Dalším prvkem *Use Case Diagramu* je spojnice mezi prvkem typu *Actor* a prvkem typu *Use Case*, nazývá se *Use*. Do jednoho diagramu se umístí prvky typu *Use Case*, se kterým daný prvek typu *Actor* komunikuje, například jako na obrázku 4.15.



Obrázek 4.15: Prvek typu *Actor* a několik prvků typu *Use Case*.

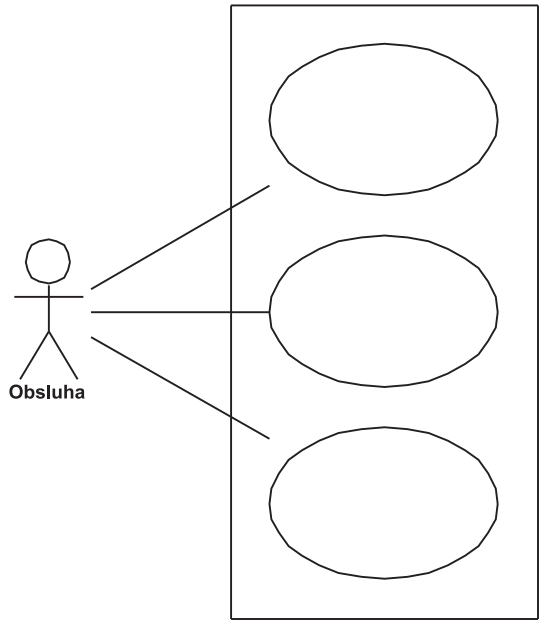
Doporučuje se použít i prvek *Boundary*, který vymezuje vnitřek a okolí systému, tedy jako na obrázku 4.16.

V některých firmách malují obrázek obráceně s prvkem *Actor* uprostřed, tak jak je uvedeno na obrázku 4.17.

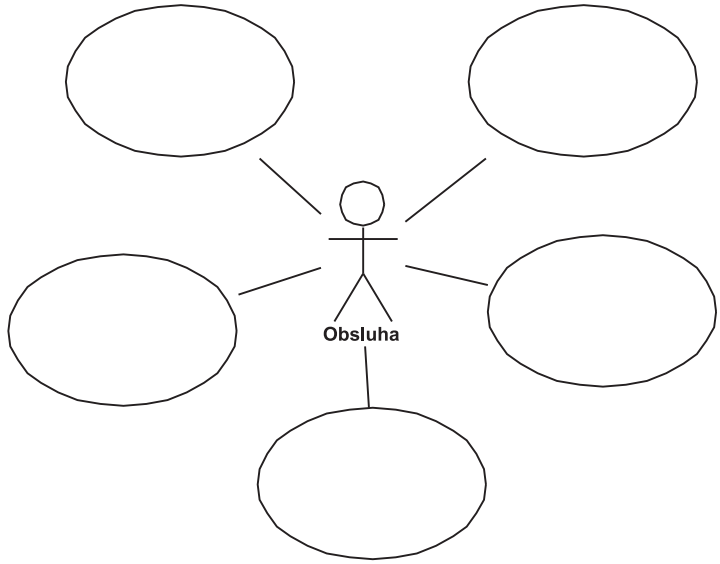
Tomuto způsobu se raději vyhněte, protože nezdůrazňuje opticky, co je venku a co uvnitř a nepůsobí příliš technicky a nakonec ani esteticky.

4.6.2 K čemu slouží a proč se vyhledávají prvky typu *Actor*

Zde je třeba opět zopakovat, že preferujeme procesní školu pro vyhledávání případů užití. Je zřejmé, že pokud bychom použili „aktorovou školu“, odpověď by zněla: „Prvky typu *Actor*



Obrázek 4.16: Použití prvku *Boundary*.



Obrázek 4.17: Chybné rozložení diagramu.

hledáme pro nalezení případů užití.“. Nyní jsou však již případy užití již dobře nalezeny pomocí *BPM*, scénáře jsou popsány. K čemu tedy ještě prvky typu *Actor*? Existují dva základní okruhy důvodů pro vyhledávání prvků typu *Actor*:

1. vyhledání prvků typu *Actor* pro samotný vývoj systému (tj. hledisko vývojáře);
2. vyhledávání prvků typu *Actor* pro obchodní účely (tj. hledisko obchodníka a zákazníka).

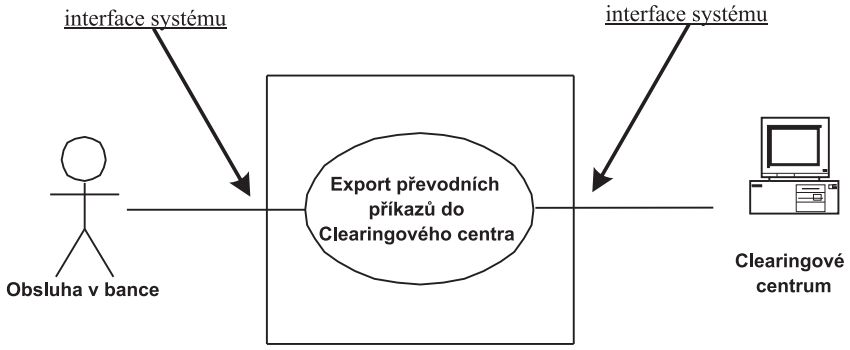
Sledujte aktuální novinky na http://www.objects.cz

Vyhledávání prvků typu *Actor* z hlediska vývojáře

Jedná se o primární důvod vyhledání prvků typu *Actor* a souvisí s problematikou rozhraní systému:

> *Prvky typu* Actor *spolu se spojnicí* Use *vůči prvku* Use Case *odhalují rozhraní systému a vzniká tak dotaz na jejich následné technologické řešení již v brzké fázi analýzy.*

Znamená to, že pokud se podíváme na znázorněný *Use Case Diagram* i s prvky typu *Actor* spolu se spojnicemi *Use* a s prvkem *Boundary*, tak nás z hlediska návrhu IS zajímají právě průsečíky mezi spojnicemi *Use* a prvkem *Boundary*. Tyto průsečíky reprezentují rozhraní neboli interface systému, viz například obrázek 4.18.



Obrázek 4.18: Identifikace budoucích interfaců systému již v raných fázích analýzy.

Pro další návrh systému tak vzniká velmi příznivá situace, kdy již v analýze, tj. hned v počátcích vývoje projektu, vznikají otázky technologické povahy o komunikaci systému vůči okolí a úvahy, co tuto komunikaci bude následně konkrétně v technologii reprezentovat. Například na obrázku 4.18 jsou patrná dvě rozhraní a je otázkou, co je reprezentuje technologicky. Z jedné strany je vidět dotaz na povahu „obrazovek“ (bude to ASP, JSP, lokální aplikace s okny apod.?), na straně druhé je otázkou, jak se odešlou převodní příkazy do Clearingového centra (dávka přes internet, datová linka?). Odhalení interfaců je důležité i pro analytika a to zejména pokud se řeší komunikace s externími systémy. Systém totiž nemůže od okolního prvku požadovat něco, co okolní prvek neumí a analytik musí tedy při návrhu funkcionalit samotného systému počítat pouze s tím, co daný prvek z okolí umí a podporuje.

Pokud budeme například navrhovat utilitu pro EA, musíme velmi dobře prostudovat interface tohoto nástroje, tj. co EA poskytuje ven, jinak nemůžeme dobře navrhnout samotné funkcionality, tedy případy užití. Do analýzy případů užití tedy spadá i analýza interfaců prvků typu *Actor* – externích systémů.

Vedlejší důvody pro vyhledávání prvků typu *Actor*

Vedlejší důvody bychom mohli nazvat také jako „obchodně-prezentační“:

> *Prvek typu* Actor *je výrazným okrasným prvkem* Use Case Diagramu.

Z hlediska obchodního úspěchu projektu bychom neměli zanedbat ani vedlejší důvody pro uplatnění prvků typu *Actor*. Nezanedbatelným faktorem se totiž pro zákazníka stává také

osobní dojem. Obrázky s panáčky a s externími systémy jako PC vypadají pro zákazníka lépe a zajímavěji než chudé obrázky bez panáčků. Při prezentaci u zákazníka proto ukazujeme *Use Case Diagramy* zásadně i s prvky *Actor*. V některých případech u evidenčních systémů chce zákazník vidět návaznost funkcí neboli rolí v podniku (např. ředitel, manažer, účetní, obsluha na přepážce atd.) s použitím systému, resp. s četností použití případů užití u těchto funkcí. V tomto případě se opět použije technika prvků *Actor*. Dříve, než si však podrobně vysvětlíme použití prvků typu *Actor* v těchto případech, musíme se nejprve věnovat dvěma vážným problémům, které s prvky typu *Actor* souvisejí. Po vysvětlení těchto problémů se pak k tomuto tématu vrátíme.

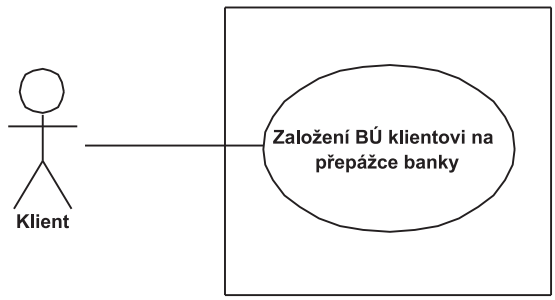
Problém zbytečné hádky nad kontextem prvku *Actor*

Prvním problémem, na který můžeme narazit při práci s prvky typu *Actor*, je tzv. zbytečná hádka nad kontextem prvku *Actor*.

Znázornění prvků *Actor* narušuje základní princip anonymity klienta systému. Pokud navrhujeme systém, nevíme nikdy, kdo je na druhé straně, tj. mimo systém. Snaha o přesnou specifikaci prvků typu *Actor* mohou vést k neplodným diskusím „kdo je vlastně za klávesnicí a jak ho nazvat“.

Příklad z praxe na hádku nad kontextem prvku typu *Actor*

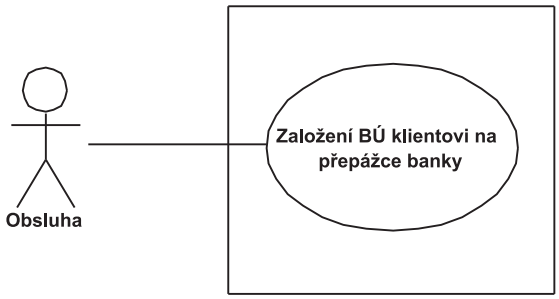
Tento příklad se skutečně stal a rád na něj (dnes již s úsměvem) vzpomínám. Ve firmě, kde jsem měl možnost svého času pracovat, byl zaveden vcelku dobrý zvyk. Když analytik dokončil větší kus své práce, byl povinen uspořádat veřejnou prezentaci analytického modelu i s promítáním před celým týmem vývojářů, včetně technologů a programátorů. Tento postup měl hned několik výhod: Jednak si analytik musel prezentaci velmi dobře připravit, aby se neztrapnil, také tým se rychleji seznámil s dokumenty analytické povahy, a krom toho povinností diváků-členů týmu bylo „rozcupovat“ analýzu v případě nalezených logických chyb. Dodnes rád vzpomínám na opravdu bouřlivé diskuse nad analytickými modely, a musím říci, že kvalitě analýzy to jenom prospělo. Dovedete si asi představit, co se v týmu odehrálo, když se při takové prezentaci promítl na stěnu model s nalezeným případem užití „Založení běžného účtu klientovi na přepážce banky“ i s prvkem typu *Actor* s názvem „Klient“ tak, jak je uvedeno na obrázku 4.19.



Obrázek 4.19: Název prvku typu *Actor* Klient.

Okamžitě se zvedla ruka a kolega oponoval: „Tam přece nepatří Klient! Já tam vidím obsluhu.“. A navrhl model na obrázku 4.20.

A kdo měl pravdu? Ptáme se, kdo je vlastně prvkem typu *Actor*? Klient nebo Obsluha? Jinak řečeno, který z těchto dvou obrázků je správně? Tým se rozdělil na dva nesmiřitelné



Obrázek 4.20: Název prvku typu *Actor* Obsluha.

tábory. Jedni argumentovali rozumně: „Přece nebudete tvrdit, že klient přeskočí přepážku, odstrčí obsluhu a naťuká si to sám? Já tam vidím Obsluhu!“. Druzí naopak tvrdili také vcelku logicky: „Prvkem *Actor* je Klient a Obsluha je jenom prostředník. Ta přece nic do systému nepřináší, sama obsluha pouze zprostředkuje klientovi založení účtu. To už tam můžete namalovat taky klávesnici a monitor, protože ona ta Obsluha jinou funkci nemá. A navíc, přece nebudete tvrdit, že tento případ užití je tu pro Obsluhu. Případ užití je pro Klienta, jak je již patrné v názvu případu užití. Jemu, tedy Klientovi přináší užitek, nikoliv Obsluze.“.

Největší chybou, které jsme se tehdy dopustili, bylo to, že jsme se o tomto hádali 2 týdny a práce stála. Výrobní porady začínaly otázkou „jak nazvat panáčka“. Přitom scénáře byly napsány a vědělo se, co se má naprogramovat. Hlavním závěrem je následující praktické doporučení:

> *Pokud je identifikována povaha rozhraní, tak další diskuse nad názvem prvku* Actor *je z hlediska samotného návrhu informačního systému neplodná a zbytečná. Názvy prvků* Actor *se pak stávají problémem pouze obchodně-prezentačním.*

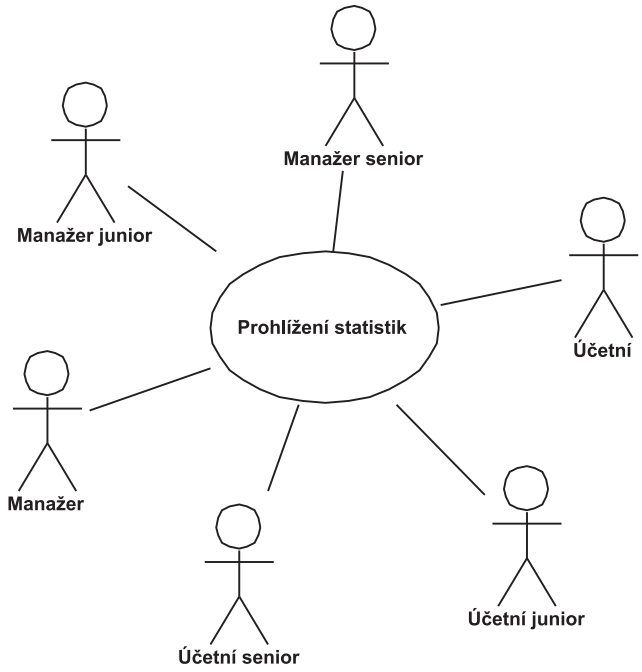
Z pohledu vývojáře (a zdůrazňuji z pohledu vývojáře) jsme identifikovali interface ven typu „obrazovka pro člověka“. Samotný název již nyní hraje roli okrasného prvku, což je problém obchodně-prezentační.

Jak by tedy bylo správné řešení právě tohoto příkladu? Důležité je, že z hlediska technologie byl problém interfacu již v dané chvíli vyřešen (zvolena daná technologie obrazovek). Přichází tedy na řadu druhé hledisko, obchodně-prezentační. Osobně jsem na 90 % přesvědčen, že až obrázek uvidí pracovník banky, bude u případu užití raději vidět prvek typu *Actor* s názvem Obsluha než s názvem Klient, a proto bych tam z dnešního pohledu umístil tento vůči zákazníkovi méně kolizní název.

Druhý problém s prvky typu *Actor* – snaha o jejich množení

Situaci si osvětlíme nejprve jedním příkladem z praxe. Firma, která dodávala ekonomické systémy, chtěla přecházet ze zastaralejší technologie na technologii modernější. Vývojáři měli současně v úmyslu také vytvořit *Use Case Model*. V té době byla populární „aktorová škola“ a tak tím hodlali začít. V průběhu nalézání prvků typu *Actor* však narazili na vskutku zajímavý problém a kvůli tomu se na mne obrátili s dotazem. Jejich problém spočíval v tom, že jim z neznámých důvodů počet prvků typu *Actor* neustále rostl a stále další prvky typu *Actor* jim přibývaly. Přiložili také diagramy.

Uprostřed diagramu byl umístěn jednoduchý případ užití a ten byl obklopen skupinou prvků typů *Actor* přibližně tak, jak uvádí obrázek 4.21.



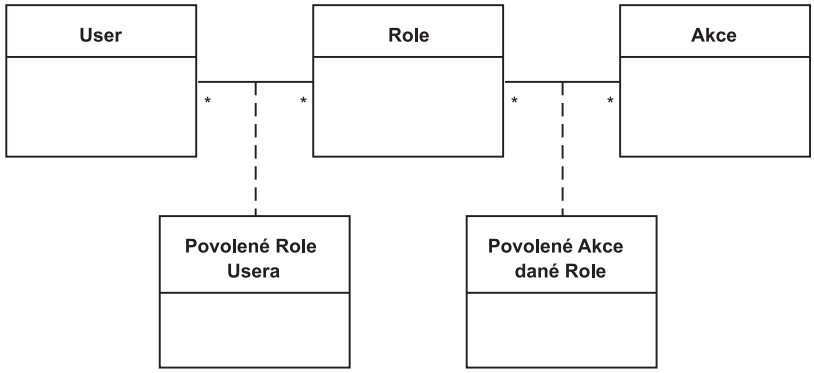
Obrázek 4.21: Typická ukázka rozmnožení prvků typu *Actor*.

Otázkou je, proč namalovali tolik prvků typu *Actor* kolem jednoho případu užití? Chyba spočívala ve špatném chápání agendy přístupových práv. V agendě přístupových práv se vyskytují tzv. Role a lze odhadnout, jaké Role se v systému vyskytují (viz obrázek 4.21) – Účetní, Účetní junior, ... atd. Na druhé straně v systému existují Akce, jako je „Založení faktury“, „Odsouhlasení faktury“ atd. a jedna z nich je (opět viz obrázek 4.21) „Prohlížení statistik“. Obrázek 4.21 tedy chybně ukazuje, která Role může spustit danou akci „Prohlížení statistik“. Odhalení chyby není až tak složité: Počet prvků typu *Actor* musí odpovídat počtu interfaců, proto je tato úvaha mylná. Jednoduchou pomůckou je namalovat *Boundary*, poté spočítat interfacy a zeptat se: „Kolik lidí sedí u systému při použití systému tímto případem užití?“ Odpověď zní: „Jeden“.

Podstatou jsou dvě již známé chyby: První je „záměna vnitřku a vnějšku“ (tj. zda hovoříme o pojmu venku nebo v systému) a druhá je „nazývání instancí jejím obsahem, nikoliv neutrálně“ (tj. chyba názvu instance „ulice Milady Horákové“). Tyto chyby nyní odhalíme a problém vyřešíme. Položme si nejprve otázku: Když se zavede prvek Role v agendě přístupových práv, o jaký typ prvku se z pohledu specifikace jazyka UML jedná? Ptáme se, je to prvek typu *Use Case*, *Class*, *Association*, *Generalization*, *Actor*, nebo jiný typ prvku v UML? Prvek Role je součástí řešení v agendě přístupových práv, tedy uvnitř systému a je to analytická třída. Jedná se o budoucí program (vzniknou z ní prvky systému jako tabulka, třída v OOP, funkce atd.). Stejně tak Akce je také třídou: Je třeba si „okódovat“ a pojmenovat spustitelné akce (instance akcí) a přiřadit právo ke spuštění k dané Roli. Příklad řešení takové agendy přístupových práv může vypadat tak, jako na obrázku 4.22.

Scénář přihlášení i s výběrem role bude vypadat například takto (vynechávám *Exception Flow*):

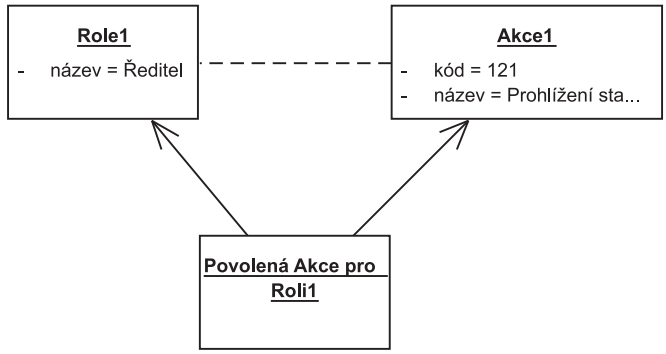
„Někdo" (synonymum pro anonym v objektovém paradigmatu) zadá username a password. V seznamu Userů se najde daný User s username a ověří se



Obrázek 4.22: Jednoduchá agenda přístupových práv.

```
password. Zobrazí se seznam Povolených rolí daného Usera, obsluha vybere
jednu Roli (může žít v daném okamžiku v jedné roli). Prvky User i Role
jsou stále dostupné až do ukončení přihlášení obsluhy.
```

Na jiném místě aplikace se poté vyhodnocuje právo k dané akci v dané situaci, což se děje tak, že pro danou roli se buď vysvítí nebo zešedí nabídka pro spuštění dané akce. Vyhodnotí se nejprve, zda existuje instance Povolené akce dané role pro danou dvojkombinaci „Role + Akce“. Takže prvky „Účetní“, „Manažer“ atd. tedy nejsou „aktoři“, ale instance ze třídy Role, navíc nazvané obsahem (vzpomeňme na chybu „ulice Milady Horákové“), stejně tak Akce nejsou případy užití, ale instance ze třídy Akce. Připomeňme, že chyba názvu instance spočívá v tom, že bychom správně měli nazývat instance pokud možno neutrálně (odvozujeme je například z názvu třídy). Situaci nám nejlépe ozřejmí příklad evidence instancí (jak je v této knize stále doporučováno), viz obrázek 4.23.



Obrázek 4.23: Příklad evidence přístupových práv v instancích.

Diagram na obrázku 4.23 ukazuje, jak vlastně funguje „povolení prohlížení statistik pro ředitele“. Existuje instance Role1, která má v atributu název text „Ředitel“ a existuje Akce1 která má kód = 121 a název = „Prohlížení statistik“. K nim existuje instance práva z asociační třídy, která je provazuje. Pokud tato instance existuje (zde ano), tak ten anonym, který se přihlásil do role ředitele podle předešlého scénáře, má právo k této akci.

Uvedená chyba „množení aktorů“ je vážná, pokud vede ke špatnému určení počtu rozhraní daného případu užití. Počet interfaců musí souhlasit s počtem aktorů a je třeba prvek *Actor* vybavit vhodným názvem – pro zákazníka nejméně kolizním.

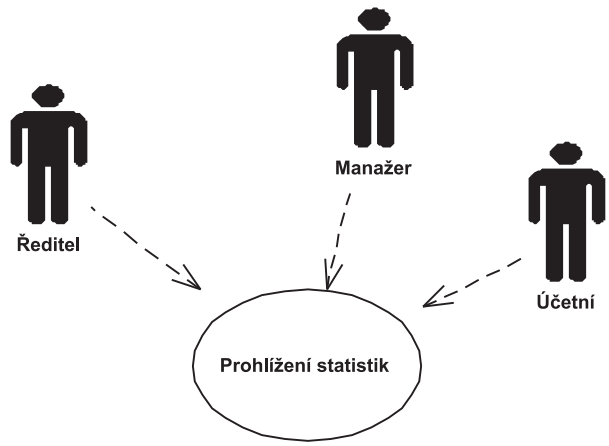
Dokončení kapitoly obchodní problém s rolemi v podniku a případy užití

Vrátíme se nyní k řešení požadavku, kdy zákazník chce vidět, které role ve smyslu funkce v podniku mimo systém (tj. opravdu ředitel venku, účetní venku atd.) používají které případy užití, navíc s udáním četností (např. za den). Některé systémy to potřebují z obchodního hlediska. Jedná se hlavně o případ, kdy je systém navržen jako „hostovaná služba“, která se u zákazníka neinstaluje, ale je používána jako množina služeb na internetu a je fakturována podle četnosti použití. Nedoporučoval bych dělat ústupek v tom smyslu, že vytvoříme chybně více prvků typu *Actor*, než kolik je interfaců, což je, jak ukazuje předešlý příklad, hrubou chybou. Vhodnější je diagramy pro účely zákazníka tvořit „jakoby bokem“ navíc, mimo vývoj a tím problém obejít.

Doporučuji v tomto případě postupovat takto: Zavedeme další stereotyp pro typ *Actor*, tento stereotyp chápeme jako „role v podniku“. Název může být zvolen například jako „business role“. Není to prvek *Actor* v tom významu, jak jej známe, ale je to role neboli funkce v podniku. Proto bychom mu měli dát i jiný obrázek, například „jiného panáčka“, viz obrázek 4.24.

Obrázek 4.24: Role v podniku s jinou grafikou.

Poté vytvoříme diagram vyjadřující, která role v podniku může používat který případ užití pomocí vztahu Dependency, například jako na obrázku 4.25.



Obrázek 4.25: Použití případu užití rolemi v podniku jako obchodní dokument.

Máme tak zachovánu čistou podobu analytického dokumentu do vývoje se správným počtem interfaců a současně poskytneme zákazníkovi jiný obchodní dokument s rolemi ve smyslu funkce v podniku mimo systém. V tomto druhém, obchodním dokumentu lze také navíc „ocenit“ dané Dependency číselnou hodnotou, například „frekvence použití za den“. Velmi dobře k tomu poslouží mechanismus tagových hodnot.

Kapitola 5

Epilog – závěrečná kapitola a doporučení k dalšímu vzdělávání

V této knize jsem se snažil logickou formou předat čtenáři poznatky z tvorby analytického modelování od základních tvrzení až po velmi speciální vzory. Z vlastní zkušenosti mohu potvrdit, že neznalosti v samotných základech tvorby analytického modelování vedou k těm nejhrubším chybám v analytických modelech s katastrofickými důsledky pro celý projekt. Chyby se dělají, ale je-li analytický model od základu špatně, ani kvalitní programátoři to už nevytrhnou. Otázkou je, jak dál pokračovat ve studiu a co doporučit.

Zájemcům o praktické příklady a procvičení základů získaných díky právě dočteným řádkům této knihy můžeme nabídnout internetový kurz profesního růstu analytiků, který přímo a konkrétně navazuje na tuto knihu. Je možné si tak zde probranou látku procvičit na mnoha příkladech a to pod vedením zkušeného lektora, autora této knihy. Vzhledem k individuálnímu přístupu ke každému jednotlivému účastníkovi kurzu je počet účastníků kurzu omezen. Bližší informace najdete na `http://www.objects.cz`.

Samozřejmě lze čtenáři poradit, aby získal co nejvíce zkušeností praxí. Ale získávání praktických zkušeností v sobě obsahuje jeden háček: Mnohdy praxe sice ukáže, že „něco je špatně“, ale nemusí se najít to místo „kde je chyba“ a jak se z ní poučit. Krom toho při učení se vlastními chybami bývá školné mnohdy příliš drahé.

Na našich domovských stránkách `http://www.objects.cz` najdete také řadu dalších školení, ať už jako konzultace nebo kurzy, a to jak přímo ve firmě, tak formou pobytových či prezenčních kurzů. Aktuální a podrobné informace najdete na zmíněné adrese.

Vaše připomínky ke knize, názory a další náměty uvítám. Pište je, prosím, na mou e-mailovou adresu `objects@objects.cz`.

Těším se nashledanou snad někdy na některém z našich školení nebo u příští knihy z naší edice.

RNDr. Ilja Kraval

Sledujte aktuální novinky na http://www.objects.cz

Server objektových technologií

http://www.objects.cz

- UML
- OOP
- Moderní metodiky vývoje IS
- Dokumenty a články zdarma
- Zajímavá školení

Novinky na serveru jednou asi za 14 dní

blíže viz http://www.objects.cz

Soupis bibliografických citací

[1] *Enterprise Architect : User Guide* [nápověda programu na disku]. Sparx Systems, 2009.

[2] *Object Management Group / Business Process Management Initiative* [online]. [cit. 5. ledna 2010]. Dostupné na internetu: <http://www.bpmn.org>.

[3] *Business Process Model and Notation (BPMN) : Version 1.2* [online]. document No. formal/2009-01-03. Object Management Group, 2009. [cit. 5. ledna 2010]. Dostupné na internetu: <http://www.omg.org/spec/BPMN/1.2>.

[4] *UML® Resource Page* [online]. [cit. 5. ledna 2010]. Dostupné na internetu: <http://www.uml.org>.

[5] *OMG Unified Modeling Language™(OMG UML), Superstructure : Version 2.2* [online]. document No. formal/2009-02-02. Object Management Group, 2009. [cit. 5. ledna 2010]. Dostupné na internetu: <http://www.omg.org/spec/UML/2.2/Superstructure>.

[6] Fowler, M. *Analysis Patterns : Reusable Object Models*. Addison-Wesley, 1997. Object Technology Series. ISBN 0-201-89542-0.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. Professional Computing Series. ISBN 0-201-63361-2.

[8] Kraval, I. *Design Patterns v OOP : se zaměřením na JAVU, C# a Delphi*. Valašské Klobouky : Object Consulting, 2002.

Sledujte aktuální novinky na http://www.objects.cz

Obsah

Sledujte aktuální novinky na http://www.objects.cz

Sledujte aktuální novinky na http://www.objects.cz

RNDr. Ilja Kraval

Analytické modelování informačních systémů pomocí UML v praxi

1. vydání
vydal Object Consulting s. r. o., Lipina 100, 766 01 Valašské Klobouky
v roce 2010
počet stran 140

sazba a zlom Ing. Viktor Patras

vytiskl Tribun EU s. r. o., Gorkého 41, 602 00 Brno (www.knihovnicka.cz)
vysázeno písmem Computer Modern v LaTeX 2_ε
formát B5
vazba V2

Sledujte aktuální novinky na http://www.objects.cz