OBJEKTOVÝ PŘÍSTUP A UML V NÁVRHU INFORMAČNÍHO SYSTÉMU

Vojtěch Merunka

Katedra informačního inženýrství, Provozně ekonomická fakulta, ČZU Praha
merunka@pef.czu.cz, http://kii.pef.czu.cz/~merunka

Motto: Analýza systému pomocí UML by neměla být jen grafické znázorňování softwarového kódu vytvářené aplikace

Abstrakt

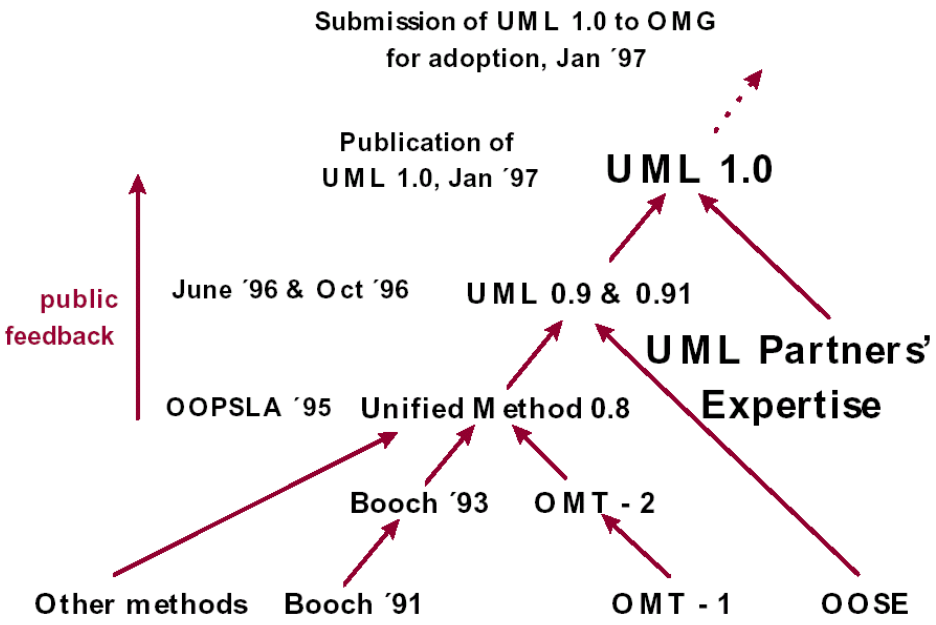
Článek podává kritický pohled na postavení nástroje UML v procesu tvorby informačního systému. Závěry, ke kterým autor článku došel, vycházejí z jeho vlastních zkušeností s analýzou a tvorbou objektově orientovaných aplikací v praxi především ve firmě Deloitte&Touche a také s výzkumem v oblasti objektového modelování.

Klíčová slova: UML, objektová analýza, objektový návrh, objektové modelování, business objekty, konceptuální objekty, softwarové objekty, hierarchie mezi objekty.

1. Objektový přístup a vznik UML

V dnešní době jsme svědky toho, jak se vyplnily předpovědi průkopníků z přelomu 80. a 90 let o budoucnosti objektově orientovaného přístupu. První oblastí, kde OOP prokázalo svoje přednosti, bylo programování. Objevily se objektově orientované a smíšené programovací jazyky. Na širší rozšíření objektově orientovaných aplikací jsme si však museli ještě čtvrt století počkat. Nejbližší budoucnost nám zřejmě přinese větší rozšíření objektově orientovaných operačních systémů a databází - pokud nás ale obratní prodejci těch současných nepřesvědčí, že jsou vlastně již teď všechny jejich produkty 100% objektově orientované.

Před vznikem UML, v první polovině 90. let, jsme měli několik mezi sebou soupeřících objektových metodologií s navzájem odlišnými notacemi. Jednalo se o tzv. objektové metodiky první generace. Mnoho softwarových firem nepoužívalo pouze jednu metodiku, ale kombinaci několika – nejčastěji objektové modely z OMT spolu s interakčními diagramy z metody Boochovy a Use-Case přístupem z Jacobsonovy metody OOSE. [5] Většina z těchto metodik se poté stala základem pro jazyk UML, jak ukazuje schéma z dokumentace o UML [1]:



UML přinesl sjednocení dosavadních notací. Notace UML nejvíce vychází z OMT[1] a stala se uznávaným standardem. UML ale do sebe zahrnul z původních metodik a notací mnoho navzájem různých prvků a stále je zahrnuje. Je to například tzv. „business extension" z původní Jacobsonovy metody, která byla přidána ve verzi 1.2 a nebo chystaná verze 2.0, která pohltí metodiku SDL pro podporu real-time procesů. [1]

2. Je UML metoda?

Samozřejmě že není. UML je „jen"grafický jazyk. To by samo o sobě nemělo vadit – je dobře, že od roku 1996 máme konečně pro objektové modelování k dispozici standard. Problém je ale v tom, že k „univerzálnímu" jazyku nemáme odpovídající metodiku[2] a tak se za metodiku mnohdy považuje samotná znalost UML.

Drtivá většina odborné literatury o UML svým čtenářům předstírá, že metodika problém není, a že se naučí modelovat v UML tím, že se naučí kreslit jednotlivé diagramy. Autor má podobné negativní zkušenosti i s náplněmi školících kurzů „moderního objektového modelování"[3].

3. Je UML jednoduchý a srozumitelný nástroj?

Zkušenosti z praxe říkají, že není. UML není rozhodně nástrojem, který laik za rozumně krátkou dobu (třeba během 15 minut na začátku schůzky s analytiky) pochopí tak, že je schopen číst a rozumět diagramům. Toto není nereálný požadavek, protože v minulosti bylo možné takto pracovat s entitně relačními a data-flow modely. Bohužel v objektově

[1] Jen pro zajímavost: První verze UML 0.8 a 0.9 používaly notaci, která byla od původní OMT odlišnější, než UML v pozdějších verzích, kdy bylo u některých objektových pojmů ustoupeno od zvláštních symbolů ve prospěch původní notace OMT.

[2] Existuje sice RUP – Rational Unified Process, ale ten se zabývá projektovým řízením a ne podrobnými postupy při tvorbě jednotlivých modelů.

[3] Někteří mlamojové dokonce svým obětem tvrdí, že začnou správně objektově modelovat, pokud si zakoupí jejich CASE nástroj, který pracuje v UML.

orientovaném modelování podobný elegantní a jednoduchý nástroj nemáme. Namísto toho zadavatele posíláme na dlouhá školení o UML, kde je nutíme umět pracovat s CASE nástroji.

Pro jedince, kteří neovládají programování, je UML příliš složitý a potom i nesprávně interpretují celý objektový přístup. [3,4,10] Je samozřejmě možné pro neprogramátory vybrat z UML přijatelnou podmnožinu pojmů, ale většina odborné literatury i výklad v kurzech se příliš a často zbytečně opírá o programátorské znalosti. Srozumitelnost a jednoduchost UML narušuje:

1. UML modely obsahují příliš mnoho pojmů. Pojmy jsou na různých úrovních abstrakce, někdy se i významem překrývají (např. vazby mezi use-cases), dokonce se jejich definice v různé literatuře liší. Proto může stejný model jinak interpretovat analytik a jinak programátor (typickým příkladem jsou asociace mezi objekty).
2. V UML diagramech je více variant pro zobrazení některých detailů v modelech (např. kvalifikátory a vazební objekty nebo stavové diagramy, které jsou mixem automatů Mealyho a Mooreova typu). Záleží na analytikovi, kterou variantu si vybere.
3. Některé pojmy jsou nedostatečně definovány (např. události ve stavových diagramech), jeden symbol UML pokrývá více odlišných pojmů (např. v diagramu sekvencí splývá datový tok mezi objekty s řídícím tokem, což kromě jiného komplikuje implementaci).
4. I když je UML po grafické stránce celkově vydařený, tak podle některých analytiků například vadí totožný symbol obdélníka pro instanci a třídu (rozlišují se jen vnitřním popisem) a také směr šipky dědění, která vede směrem k rodičovskému objektu, i když v kódech programovacích jazyků (i při laických výkladech co to je dědění) je dědičnost realizována opačným směrem – od rodičovského objektu k jeho potomku.

4. Je UML objektově orientovaný?

Podle definice samotných tvůrců UML je. UML podporuje mnoho objektově orientovaných pojmů a v současné době nemáme jiný „více“ objektově orientovaný a přitom také standardní modelovací jazyk. Úspěch UML v praxi se opírá o mnoho úspěšných projektů, kde software byl vytvářen vesměs v jazyce C++ nebo Java, což jsou jazyky, které využívají objektově orientovaný přístup. Podívejme se ale na „objektovost“ UML podrobněji:

4.1 IS-A není hierarchie typů ani dědění

Hierarchie dědění a hierarchie typů v jednom systému nemusí vždy znamenat totéž. Je tomu tak proto, že například při používání dědičnosti máme v programovacích jazycích nástroje pro skrývání metod a dat děděných objektů a polymorfismu mezi objekty můžeme dosáhnout i bez dědění. Také hierarchie objektů ve fázi zadání není zcela totožná s děděním ani s hierarchií typů, jak bude ukázáno dále. Jenomže v UML se tyto tři sice podobné, ale ne totožné, hierarchie zobrazují stejně[4]. Při modelování je však na „hierarchie objektů“ třeba nahlížet různě …

1. *Z pohledu tvůrce* – návrháře nových objektů. Tato hierarchie je **hierarchií dědičnosti**, protože dědičnost je nástrojem pro tvorbu nových tříd. Ale k tomu dojde až při návrhu systému, kdy nás zajímá programová realizace.
2. *Z pohledu uživatele.* Tento pohled, který náleží do fáze analýzy, lze ještě podrobněji rozdělit:

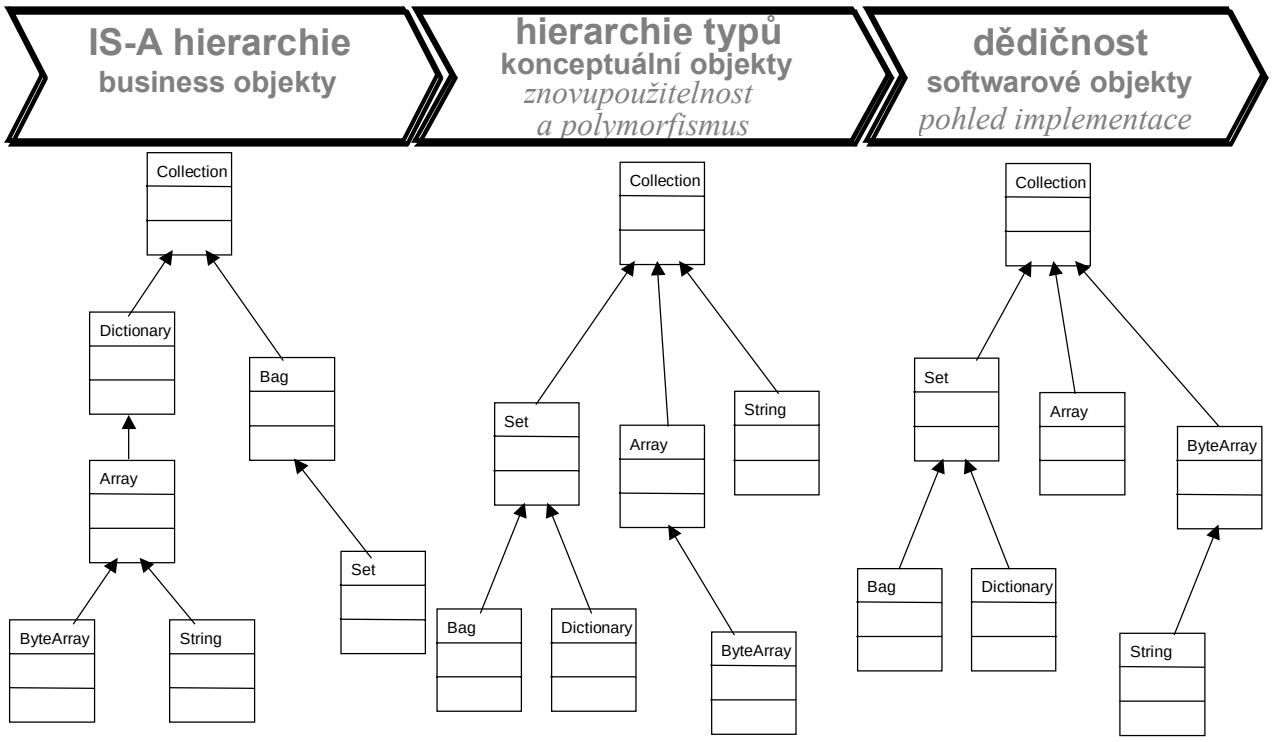
[4] K rozlišení vazeb můžeme použít uživatelsky definovaný stereotype, ale tím se dostáváme mimo definovaný standard.

2.1. *Z pohledu polymorfismu* – pohled analytika, který potřebuje objekty ve svém systému použít, ale jejich tvorbou se ještě nezabývá. Objekty na nižších úrovních hierarchie potom musejí být schopny dostávat stejné zprávy a sloužit ve stejném či podobném kontextu, jako objekty vyšších úrovní. Právě tato hierarchie je **hierarchie typů**.

2.2. *Z pohledu popisu zadání* – Objekty zde seskupujeme do „skupin“ neboli domén, u kterých nerozlišujeme ještě mezi pojmem třída objektů a množina objektů (podrobněji v kapitole 4.2). Hierarchie mezi objekty je dána hierarchií těchto domén – doména „podtypu“je totiž podmnožinou domény „nadtypu“. Tato hierarchie objektů je anglicky označována jako IS-A, česky ji můžeme přeložit „**je jako**“ (nebo „**patří k**“). Od hierarchie typů se může v konkrétních případech lišit proto, že se nezabývá jen chováním objektů na rozhraní (na takový programátorský pohled je totiž v této fázi analýzy ještě brzo), ale objektem celkově včetně jeho struktury vnitřních dat, tedy tak, jak objekty chápe zadavatel.

U jednoduchých úloh je samozřejmě pravda, že uvedené tři hierarchie jsou totožné. Proto se asi tímto faktem kuchařky na UML příliš nezabývají. U komplexnějších úloh však toto tvrzení neplatí a to například při návrhu systémových knihoven, které se opakovaně znovupoužívají při tvorbě konkrétních systémů.

Pěkným a názorným příkladem je ukázka na obrázku, kde je hierarchie IS-A, hierarchie typů a dědění pro část systémové knihovny jazyka Smalltalk týkající se sad (collections) objektů. Podobnou knihovnu najdeme v každém objektově orientovaném programovacím jazyce a tato problematika je i blízká datovým strukturám v objektových databázích. Jsou to následující třídy[5]:



[5] Skutečný počet tříd v této knihovně je mnohem větší. Např. základní verze Smalltalk/VisualWorks 5i.4 jich má 109.

- **Collection** (česky „sada“). To je abstraktní třída, ze které jsou odvozovány jednotlivé konkrétní třídy. Společnou vlastností všech těchto je objektů je schopnost obsahovat jako svoje data další objekty.
- **Dictionary** (česky „slovník“). Slovník je taková sada, kde každá hodnota v ní uložená má přiřazenou jinou hodnotu (takže dohromady tvoří dvojici), která slouží jako přístupový klíč k dané hodnotě. Slovníky můžeme použít opravdu jako slovníky pro jednoduché překlady z jednoho jazyka do druhého. Dalším často uváděným příkladem použití objektových slovníků je například jejich použití pro telefonní seznam – klíčem jsou jména osob a hodnotami s klíči spojenými jsou telefonní čísla.
- **Array** (česky „pole“). Pole lze jednoduše popsat jako slovník, kde klíče hodnot mohou nabývat pouze přirozených čísel 1 až velikost pole. Na hodnoty pole se přistupuje tedy také jakoby pomocí klíčů.
- **ByteArray** (česky „bajtové pole“). Jedná se o takové pole, kde povolený okruh hodnot je omezen na celá čísla v intervalu 0 až 255.
- **String** (česky „řetězec znaků“). Na řetězec znaků lze nahlížet jako na pole, kde povolený okruh hodnot je omezen na znaky.
- **Bag** (česky „pytel“). Bag je taková sada, do které lze objekty ukládat nebo z ní vybírat. Hodnoty uvnitř pytle ale nejsou přístupné pomocí klíčů, ale pouze skrze svoji hodnotu.
- **Set** (česky „množina“). Množina je taková sada typu „pytel“, do které lze stejnou hodnotu vložit jen jednou. Pokud množina hodnotu již obsahuje, tak další vložení stejné hodnoty je ignorováno na rozdíl od výše uvedeného pytle, který násobné výskyty stejné hodnoty dovoluje. Objekty množina svojí funkčností odpovídají matematickému chápání množin. Proto se tak jmenují.

Uvedený popis tříd na obrázku sleduje pohled analytika a zadavatele – tedy hierarchii „je jako“ (IS-A). Pokud bychom ale popis soustředili na popis funkčnosti – rozhraní objektů, které je vymezeno okruhem přípustných zpráv a přemýšleli více programátorsky, dojdeme k poněkud odlišné **hierarchii typů**. Například slovníky mohou dostávat stejné zprávy jako množiny a lze tedy na ně ve smyslu vzájemného polymorfismu nahlížet jako na podtypy množin. A naopak s řetězci znaků se pracuje značně odlišným způsobem než s poli, takže je můžeme považovat za typ, který s poli polymorfní není.

A do třetice hierarchie dědění, která je podstatná pro programovou realizaci uvedených tříd, je také trochu jiná. Řetězce znaků je výhodné implementovat děděním z bajtových polí a přidáním či pozměněním potřebných metod. Naopak pole a bajtová pole se implementačně dost liší, protože obyčejná pole se v paměti realizují jako pole odkazů na objekty kdežto bajtová pole jsou jednoduché úseky paměti. Dědění mezi poli a bajtovými poli proto užitečné není a jejich funkční podobnost není dána příbuzností v dědění.

Jak bylo na příkladu ukázáno, problematika hierarchií objektů není jednoduchá. Při návrhu systému je proto nejlepší dědění odsunout na pozdější fáze a zacházet s ním především jako s implementačním nástrojem. Nejprve je třeba na úrovni objektů reálného světa rozpoznat hierarchii „je jako“ (is-a), pořádně ji prokonzultovat se zadavatelem, potom ji upřesnit vymezením příslušných typů pro konceptuální objekty a až nakonec přemýšlet o optimální implementaci typů pomocí dědění softwarových objektů. Ve většině programovacích jazyků lze proto nalézt prostředky (které se ale bohužel velmi málo nebo nesprávně používají) pro oddělení typů a tříd.

Na vše zde diskutované lze samozřejmě UML použít, ale velmi záleží na schopnostech analytiků a manažerů projektů, aby si zajistili správnou interpretaci svých modelů, protože vývojáři mají silnou tendenci všechny modely chápat jen jako zobrazení struktur pro zdrojový kód, k čemuž ještě napomáhá již uvedený fakt, že v UML se tyto hierarchie v UML zobrazují stejně.

4.2 Třídy objektů a množiny objektů

Na pojem třídy se v objektovém modelování může nahlížet dvojím způsobem:

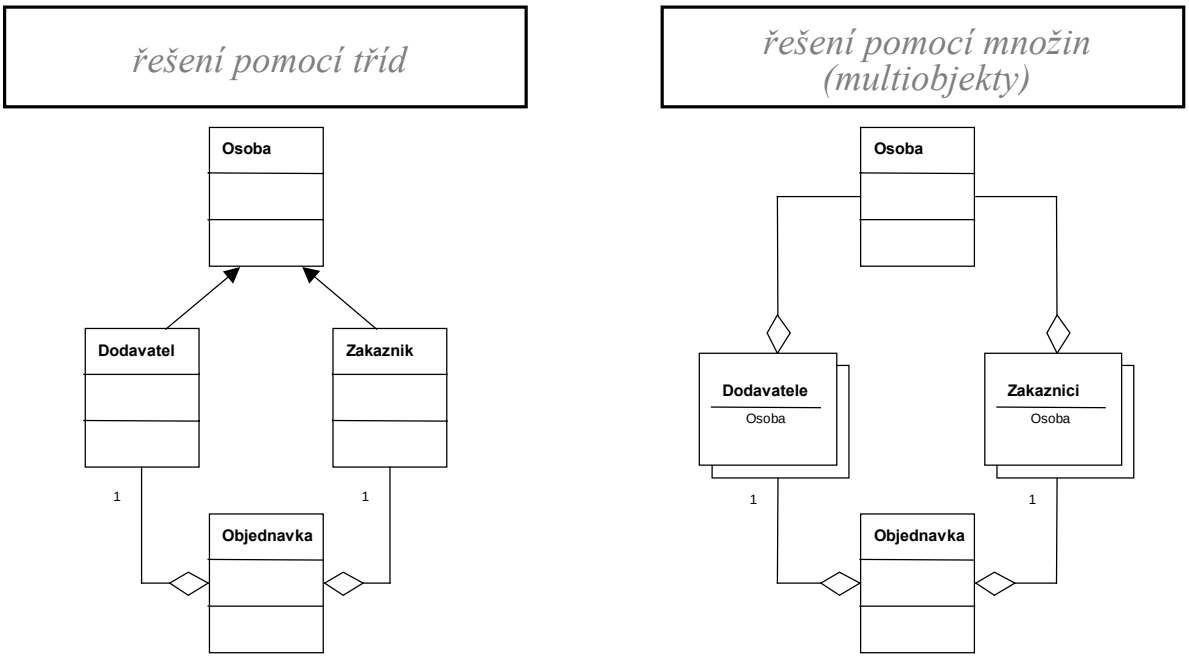
1. Třída je **realizace objektového typu**; ve třídě uchováváme popis struktury objektů tohoto typu a množinu jeho operací/metod. V čistě objektových jazycích (např. CLOS, Smalltalk) se s třídou na rozdíl od hybridních/méně objektově orientovaných jazyků (např. C++, C#, VB, Object Pascal v Delphi a Java) může pracovat jako s objektem se vším všudy, diskutovaný popis struktury spolu s operacemi jsou jeho data, se kterými lze za chodu programu pracovat, měnit je, doplňovat přidávat nebo mazat.
2. Úplně jiný pohled na třídu je chápání třídy jako množiny, která vymezuje **výskyt objektů** jednoho příslušného typu.

Naneštěstí se v UML oba způsoby nazírání na pojem třídy mísí dohromady a nerozlišuje se mezi třídou jako objektem sám o sobě a mezi třídou jako pomyslnou množinou tvořenou všemi objekty, které do dané třídy patří. UML sice ve své specifikaci obsahuje tzv. multiobjekty, které můžeme chápat jako množiny instancí a na druhou stranu také obsahuje tzv. powertypes, které je možné použít při zobrazování samotných tříd jako objektů, ale tuto možnost většina analytiků nepoužívá a ani nejsou podporovány v běžně používaných CASE nástrojích, neboť implementace UML v CASE nástrojích je vesměs jen částí celého UML.

Nesprávné splynutí diskutovaných dvou pojmů vede potom k tomu, že se všechny množiny objektů v modelovaném systému modelují jako třídy, což má nepříznivý vliv i na implementaci. Problém si ukážeme na příkladu. V systému, kde jsou objekty třídy **Osoba**, potřebujeme vymezit z nějakého důvodu dva druhy osob: **Zákazníky** a **Dodavatele**. Pro většinu analytiků se nabízí jednoduché UML řešení: sestrojit dvě nové podtřídy **Zákazník** a **Dodavatel**, které „dědí“ ze třídy **Osoba**. Toto řešení však není vždy optimální minimálně ze dvou důvodů:

1. Pokud nepotřebujeme v nových podtřídách nové atributy a operace, tak se nové třídy zavádí jen kvůli potřebě zanesení do modelu skutečnosti, že některé osoby jsou a jiné nejsou zákazníky a dodavateli.
2. Přináší potíže v případě, kdy se jedna a ta samá osoba stane zákazníkem a dříve byla dodavatelem a dokonce nebo kdy bude současně zákazníkem i dodavatelem.

Proto je mnohdy výhodnější toto zadání realizovat pomocí kombinace tříd a množin objektů. Obě varianty jsou na obrázku:



Nerozlišené pojmy třída a množina je možné z důvodu jednoduchosti a srozumitelnosti modelovat jednotně na úrovni hierarchie „je jako“ (is-a) mezi business objekty. Pro konceptuální objekty, kde se tato hierarchie převádí na hierarchii typů, jsou však především množiny tvořené objekty z různých tříd potřeba a pro objekty softwarové, kde hierarchie typů vede k dědění v programovacím jazyce je to už nezbytnou nutností.

4.3 UML a čistě objektové programovací jazyky a objektové databáze

Není pochyb o tom, že UML je silný nástroj pro modelování, které je zakončeno implementací ve smíšeném programovacím jazyce. Typický UML projekt je programován v C++ nebo v Javě a řešení využívá relační databázový server. Otázkou však je, jak UML podporuje čisté objektové programování v jazycích jako je například Smalltalk a jak pomáhá při analýze a návrhu báze dat pro nerelační objektové databáze jako je například Gemstone. [3] Vývojáři, kteří v těchto prostředích pracují, se shodují v názoru, že UML příliš vhodný není a upozorňují především na:

1. UML pracuje s typovanými datovými modely, což čistě objektově orientovaným jazykům, které jsou dynamické, nevyhovuje. Stejně je tomu i s objektovými databázemi.
2. UML přímo nepodporuje některé vazby mezi objekty (např. závislost mezi objekty nebo polymorfismus bez přítomnosti dědění), které sice smíšené jazyky neznají, ale pro čisté objektové programování jsou důležité. V takových případech UML nabízí jen možnost definovat si vlastní stereotypy.
3. Většina sofistikovaných objektově orientovaných algoritmů je založena na spolupráci více objektů ve vhodné datové struktuře. V UML se však taková řešení musejí rozkreslovat do oddělených diagramů[6] popisujících zvlášť statickou datovou strukturu, zvlášť výpočetní mechanismus a zvlášť stavy a přechody objektů a to ještě jen po jednotlivých objektech (nelze jedním diagramem znázornit vzájemné souvislosti operací, stavů a přechodů více objektů mezi sebou).

[6] Je sice pravda, že v některých publikacích – ale ne od autorů UML – se nezakazuje tvorba vlastních nových diagramů. Ale i kdyby to bylo přípustné, tak nemáme žádnou praktickou možnost s nimi pracovat.

Pokud má být UML opravdu univerzální modelovací jazyk, což by jistě bylo k užitku, tak musí do budoucna lépe podporovat jiné programovací[7] „kultury“, než je jen programování s objekty ve smíšených jazycích a s relačními databázemi.

5. Jak UML podporuje fáze vývoje projektu?

Pokud chceme/musíme používat OOP ve všech fázích vývoje životního cyklu softwarového systému, tak je třeba po celou dobu nějakým způsobem pracovat s pojmem objekt. [11,12,13]

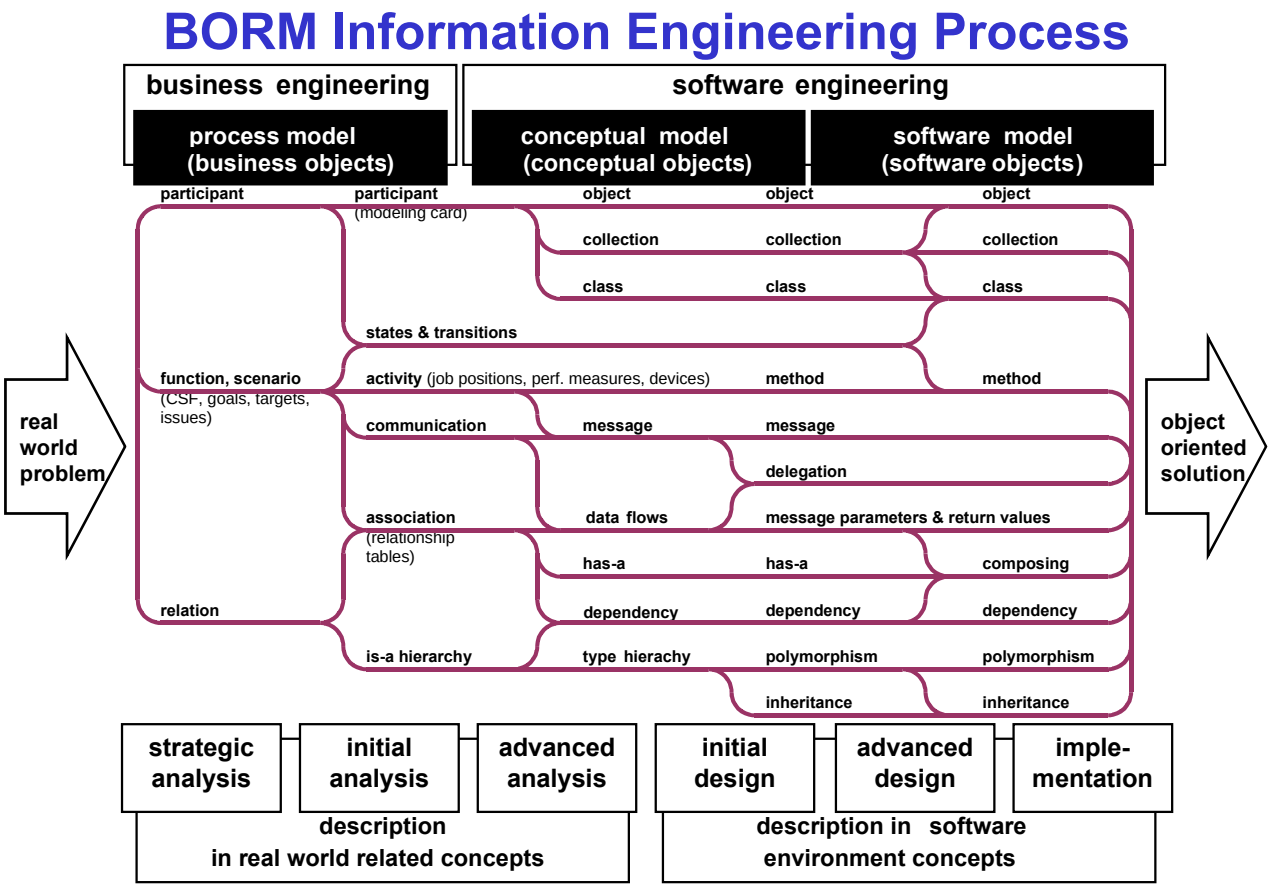
Během tvorby systému je třeba model systému postupně transformovat do takové podoby, která je nezbytná pro fyzickou realizaci systému v podobě programu v daném programovacím jazyce. Právě dotvoření modelu do takové úrovně podrobností, které již odpovídají výrazovým prostředkům použitého prostředí (programovací jazyk, databáze, …), lze považovat za okamžik ukončení objektově orientovaného návrhu a možné zahájení implementace.

Samotný pojem objektu včetně jeho vlastností se však v průběhu vývoje IS mění. Jinak chápe objekt programátor při implementaci v nějakém konkrétním programovacím jazyce a jinak chápe objekt zadavatel, protože pro něj je objekt zobrazením nějaké entity reálného světa, která je v okruhu jeho zájmu při formulaci zadání. Tato záležitost byla již v tomto článku popisována na příkladě hierarchií mezi objekty. Pokud je modelování systému zahájeno softwarově orientovanými pojmy, tak zbavujeme většinu zainteresovaných osob ze strany zadavatele schopnosti sledovat projekt již od samého počátku.

Podle našich zkušeností, které jsme předali do návrhu naši vlastní metodiky BORM [14,15,16], je výhodné objekty v průběhu projektování rozdělit na

1. **Softwarové objekty**, se kterými se pracuje v modelech závěrečných fází vývoje IS. Tyto objekty obsahují pojmy přímo odpovídající konstrukcím z objektových programovacích jazyků. Modely v této fázi projektu musí také zohledňovat refactoring kódu, znouvupoužitelnost kódu a jeho návaznost na tzv. „legacy“ systémy.
2. **Konceptuální objekty**, se kterými se pracuje v modelech prostředních fází vývoje, kdy se musíme věnovat realizaci zadání, ale ještě nejsme pod omezujícím vlivem implementačního prostředí. Tyto objekty obsahují základní pojmy objektově orientovaného paradigmatu, jako například polymorfismus objektů, zapouzdření, skládání, delegování, klasifikace objektů podle různých dimenzí, závislost objektů, třídy a množiny objektů atd. Je pravda, že některé z konceptuálních pojmů jsou shodné se softwarovými pojmy, ale značnou část je třeba při přechodu na softwarové objekty transformovat, protože současné používané programovací jazyky (*např. C++*) podporují pojmy OOP pouze omezeným způsobem.
3. **Esenciální objekty**, které také označujeme jako „**Business objects**“. Modely těchto objektů vyplňují mezeru mezi zadáním - tj. chápáním na aplikační úrovni zadavatele a mezi konceptuálním objektovým modelem.

[7] Otázka do diskuse: Má UML být modelovacím jazykem i pro logické, funkcionální, agentové a další paradigmata programování nebo ne?



Z uvedených informací lze očekávat, že jednotlivé atributy a vazby mezi objekty se budou v průběhu vývoje IS měnit, a že každý následující prvek modelu bude mít zřejmě svého předchůdce z předešlé fáze, ze kterého byl odvozen. Pojmy a vazby různých diagramů UML však postrádají jakoukoliv hierarchii. Ve specifikaci UML se nedočteme, jaké pojmy a vazby patří do jaké fáze vývoje projektu. Až na výjimky se ani nedozvíme, jak spolu mezi sebou souvisí. Výjimkou je například vazba asociace mezi objekty a skládání mezi objekty, protože skládání je definováno jako zvláštní případ asociace – čemuž lze rozumět i tak, že skládání lze z asociací odvozovat během upřesňování modelu. Jenomže obě vazby jsou součástí stejných diagramů. Asociaci tak můžeme použít i u diagramu popisujícího podrobný návrh těsně před implementací a také můžeme použít skládání už v modelech prvotní analýzy.

5.1 Business objekty

Business objekty nelze chápat pouze jako počáteční zjednodušení budoucích softwarových objektů, jak často analytici v UML chybují. [2,6,7] Business model je sice jednodušší, ale zároveň obsahuje pojmy, které současné objektové programovací jazyky přímo nepodporují. Jsou to například různé formy polymorfismu objektů[8].

Při práci na velkých projektech se analytici informačních systémů setkávají s problémem, kdy při startu projektu nejsou známy všechny požadavky na systém a zákazník očekává, že jejich nalezení a upřesnění bude až součástí projektu. Celá záležitost je ještě o to složitější, protože funkčnost budovaných rozsáhlých systémů má vliv i na vlastní organizační a řídící strukturu podniku nebo organizace, kam se systém zavádí – jsou to například nové či pozměněné

[8] Například různé reakce téhož objektu na poslanou zprávu v závislosti na jeho stavech nebo na kontextu odkud byla zpráva odeslána.

pracovní funkce, změna řízení, nové pracovní funkce, nová oddělení atp. Proto je žádoucí se při práci na informačních systémech zabývat i změnou těchto souvisejících struktur.

Právě procesní modely sestavované z business objektů jsou ověřenou a v praxi používanou metodou pro analýzu, návrh a implementaci organizačních změn za aktivní spoluúčasti zadavatelů a návazného budování informačního systému. V sadě nástrojů UML sice máme use-case diagramy, ale pokud potřebujeme modelovat větší podrobnosti, tak musíme použít buď diagramy z tzv. „business extension UML“, a nebo – což je bohužel v praxi obvyklejší – diagramy určené k zobrazování softwarových struktur jako je třeba object interaction diagram a nebo class diagram.

5.2 Konceptuální a softwarové objekty

V nástrojích UML se „konceptuální“ a „softwarové“ objekty nerozlišují a používají se pro ně také stejné diagramy. Ve fázi analýzy ale potřebujeme co nejlépe poznat zadání a tam mohou být implementační podrobnosti na obtíž. A naopak ve fázi návrhu se potřebujeme zaměřit na realizaci výstupů z analýzy, ale nepotřebujeme již znát některé aspekty modelované reality.

Podceňování rozdílností modelů v jednotlivých fázích tvorby informačního systému vede v některých případech „opravdových programátorů“ k takovému zjednodušení, kdy se analýza pomocí UML chápe pouze jako grafické znázornění budoucího softwarového kódu – typicky v C++. Analytické modely potom neslouží k upřesnění zadání s potenciálními uživateli systému, kteří jsou navíc stresováni složitostí modelů, které jsou jim předkládány. Projekty v UML velmi často trpí tímto nedostatkem. Jako reakce se proto v praxi dokonce objevily dva „opravné“ přístupy:

1. „Extrémní programování“, které v diskutované souvislosti lze charakterizovat jako přístup, kdy se využívá vlastností vyspělých vývojových prostředí objektových jazyků k programování a testování částí systému. Modelování systému se omezuje téměř k nule a v rychlých – maximálně několikadenních – iteracích se na základě softwarové implementace systému po částech upřesňuje zadání. Modely a dokumentace tak jak je známe z klasických metod, se sestavují až později na základě postupně vznikajícího kódu. [9]
2. Metodologie šité na míru konkrétní aplikační oblasti (angl. Domain Specific Methodologies). Jedná se o metodologie, které jsou rozsahem a postupem podobné klasickým metodám, ale pracují s pojmy, které jsou specifické pro danou oblast (např. telekomunikace, chemický průmysl, …). Ve své oblasti použití tyto metody dosahují výrazně lepších výsledků, protože výrazové prostředky UML jsou zde nedostatečné, nutí analytika modelovanou problematiku transformovat do podoby, která nedovoluje zachytit všechny potřebné detaily zadání. Takto sestavený konceptuální model je nedostatečný pro tvorbu kódu a proto se vývojáři musejí při kódování vracet k modelované problematice a upřesňovat detaily systému. Potom považují UML za málo užitečný a zbytečně pracný mezikrok mezi zadáním a implementací. [8]

6. Závěr

Cílem článku nebyla negativní kritika UML samotného. UML je první v praxi úspěšný pokus o zavedení rozumného objektového standardu a je vhodné ho používat. Záměrem autora bylo upozornit na některé problémy, které s sebou používání UML v praxi přináší. Diskutované problémy lze shrnout do následujících bodů:

1. UML není metoda, je to „jen“ zobrazovací prostředek.
2. UML je komplikovaný – kdo neovládá programování, tak se ho obtížně učí.
3. UML nezdůrazňuje, které pojmy se mají používat ve fázi analýzy a které až ve fázi návrhu a implementace.
4. UML málo podporuje čistě objektové programovací jazyky a objektové databáze.
5. UML málo pomáhá tam, kde na počátku projektu není ještě zcela vymezené zadání.
6. Modelování v UML nesmí být jen zobrazování budoucího zdrojového kódu aplikace.

Myšlenky popisované v tomto článku jsou syntézou vlastních zkušeností s objektovým modelováním v praxi především ve firmě Deloitte&Touche, s výzkumem, s výukou tvorby informačních systémů na ČZU a ČVUT, názorů získaných s debat s kolegy a různých dalších zdrojů.

Literatura:

[1] *UML documents*, http://www.omg.org nebo http://www.rational.com/uml

[2] Ambler, Scott W: *Be Realistic about the UML*, http://www.agilemodeling/essays/references.htm

[3] Ambler, Scott W: *Toward executable the UML*, http://www.sdmagazine.com

[4] Thomas, David: *UML – The Universal Modeling and Programming Language?,* September 2001, z knihovny na http://www.ltt.de

[5] Cook, Steve; Daniels, John: *Object-Oriented Methods and the Great Object Myth*, Object in Europe, SIGS Publications, Vol 1. Nr. 4, 1994

[6] Taylor, D., A. *Business Engineering with Object Technology*, John Wiley 1995

[7] Kotonya, G and Sommerville, Ian, *Requirements Engineering: Processes and Techniques,* 1999, J. Wiley and Sons

[8] *Domain Specific Methodology – 10 Times Faster Than UML*. Metacase Ltd., Finland, http://www.metacase.com

[9] *Extreme programming approach*, http://www.xprogramming.org a ještě také jiná stránka na http://www.extremeprogramming.org

[10] Graham, Ian; Simons, Anthony J H: *30 Things that Go Wrong in Object Modeling with UML 1.3*, University of Sheffield & IGA Ltd. http://www.iga.co.uk

[11] Goldberg A., Rubin K. S.: *Succeeding with Objects - Decision Frameworks for Project Management*, Addison Wesley, Reading Mass, 1995

[12] Ambler S.: *Process Patterns – Building Large-Scale Systems Using Object Technology*, SIGS Books 2000, ISBN 0-521-64568-9

[13] Ambler S.: *More Process Patterns – Delivering Large-Scale Systems Using Object Technology*, SIGS Books 2000, ISBN 0-521-65262-6

[14] Knott, Roger P.; Merunka, Vojtěch; Polák, Jiří: *Process Modeling for Object Oriented Analysis using BORM Object Behavioral Analysis*, in Proceedings of Fourth International Conference on Requirements Engineering, Chicago 2000

[15] Merunka, Vojtěch; Polák, Jiří; Rivas, Luis: *BORM – Business Object Relation Modeling*, in Proceedings of WOON - Fifth International Conference on Object-Oriented Programming, St. Petersburg 2001

[16] Merunka, Vojtěch; Polák, Jiří: *Úvod do metody BORM – Minikurz*, ve sborníku 5. ročníku celostátní konference Objekty 2000, Praha 2000, http://objekty.pef.czu.cz

[17] Merunka, Vojtěch; Virius Miroslav: *Jazyk modelovací, unifikovaný*, CHIP, Vogel Publishing s.r.o., série článků únor-duben 2002

[18] Molhanec, Martin: *UML – několik kritických poznámek*, ve sborníku konference Tvorba softwaru, Ostrava 2002