Objektově orientovaná analýza a design

Vojtěch Merunka a Jiří Polák

Úvod

V posledních letech se neustále dočítáme o stále nových a stále výkonnějších typech počítačů a o stále důmyslnějším programovém vybavení. Informatika se stává nedílnou součástí života každého z nás. Pojem počítač a program je pro většinu z nás stejně samozřejmou reálií jako např. automobil, televize, pračka a nebo lokomotiva. S tím však přímo souvisí pro některé z nás i problematika tvorby software, na který se zaměřujeme. Na rozdíl od klasických disciplín, jakými je třeba stavitelství nebo automobilový průmysl, je softwarové inženýrství stále velmi mladou (což by vadit nemělo) a nevyspělou (což už vadí) disciplínou. Dnes téměř nikoho nenapadne postavit most nebo automobil "na koleně" bez pomoci standardů, metodologických technik, znalostí atp. Software je však stále ve velké míře vytvářen řemeslným způsobem. Statistiky nám ukazují velmi varující důsledky takovéhoto počínání; pouze necelých deset procent prostředků vynakládaných na tvorbu softwaru se vydává na produkty, které se potom opravdu používají. Vše ostatní se v lepším případě přepracovává a v tom horším vyhazuje.

Situace na poli tvorby software je v mnoha případech kvalitativně velmi blízká dřevním dobám počítačové éry. Tu a tam se něco povede, například i patnáctiletý mladík naprogramuje věc, před kterou zkušení počítačoví orli padnou na zadek, ale mnohem častěji se něco nepovede či se chyba projeví až po určité době. Tuto nám všem známou situaci lze "zahrát do autu" řečmi o kreativním a uměleckém charakteru programování počítačů apod. Samozřejmě je třeba respektovat genialitu počítačových umělců - např. tvůrců operačních systémů nebo speciálních programů např. pro řízení raketoplánu apod., ale je naprogramování personální a účetní agendy pro malou soukromou firmu stejný případ? Právě že není. Drtivá většina vytvářených programů všude po světě má mnohem více charakter průmyslového výrobku než uměleckého či řemeslného díla. Takovéto programy by měly být stejné produkty inženýrství, jako třeba domy nebo stroje, jenž jsou produkty jiných inženýrských specializací. Umělecký a řemeslný způsob tvorby software má sice svoje nezastupitelné místo, ale není vhodný pro masové použití, protože se v této oblasti nemůže rovnat výhodám průmyslové produkce. Dokonce i pro "uměleckým způsobem" vytvářená díla jsou nezbytně nutné "průmyslově vyrobené" komponenty (moduly, subsystémy atp.).

Úkolem softwarového inženýrství je od 60. let právě hledání teoretických a praktických prostředků vedoucích k významnému zvýšení spolehlivosti a efektivity programátorské práce v kombinaci s novými architekturami výpočetních systémů.

Klasické techniky návrhu

Dnešní doba klade stále rostoucí požadavky na tvorbu stále dokonalejšího a spolehlivějšího softwaru v co nejkratším čase. Nabízí se tu srovnání s průmyslovou revolucí z přelomu 18. a 19. století, která měla za důsledek přesun objemu výroby od drobných řemeslníků k prvním velkovýrobcům. Již více než 20. let k tomuto napomáhají prostředky tzv. "**klasické strukturované**

analýzy a designu informačních systémů", které byly během 80. let vypracovány téměř k dokonalosti. Jedná se především o **Yourdonovu strukturovanou metodu**, která se postupně stala součástí mnoha veřejných i firemních metodologií. Z jiných používaných prostředků jmenujme například **Jacksonovu programovací techniku**. Vzhledem k omezenému rozsahu tohoto článku nemůžeme dopodrobna popisovat vlastnosti uvedených technik a budeme proto u čtenáře předpokládat jejich částečnou znalost, i když není bezpodmínečně nutná pro přečtení našeho článku. Zaměřme se nyní na silné a slabé stránky klasických technik:

Přednosti klasických technik

Nespornou předností klasických technik je jednak jejich vyzrálost daná dlouholetým a relativně četným používáním. K dalším přednostem patří existence příslušného podpůrného matematického aparátu, jakým je například relační algebra, techniky datové normalizace apod.

Slabá místa klasických technik

Slabá místa klasických technik návrhu spočívají v jimi podporované architektuře výpočetních systémů a s tím souvisejících konkrétních kroků a etap návrhu. Základem a cílem systémů je totiž klasický von Neumanův počítač a s ním související nevyhovující imperativní model výpočtu a oddělení procesní stránky výpočtu (algoritmy) od datové stránky (data v paměti), což má negativní vliv na podobu jednotlivých technik a nástrojů používaných v jednotlivých etapách vývoje softwarové aplikace. Klasická analýza vesměs spočívá ve vytvoření dvou modelů: datového modelu (nástroj: ERD = entity-relationship diagram) a funkčního (DFD = data-flow diagram) modelu, které mají spolu formálně velmi málo společného, a dokonce mohou být na počátku analýzy vytvářeny nezávisle na sobě. V dalších etapách od analýzy k designu dochází k rozpracovávání datového modelu a funkčního modelu a jejich doplnování o další modely, které dále popisují různými způsoby řešený systém: o modely datové struktury (DSD = data structure diagram) a modely programové struktury (SCH = structure chart). Design tedy končí dopracováním relativně velkého množství slabě spolu provázaných modelů, které slouží jako podklad ke kódování v příslušném programovacím jazyce. (viz. obrázek) Samozřejmě, že klasická analýza a návrh spěje vždy k jedinému celistvému a provázanému systému, ale činí tak relativně komplikovaným způsobem přes řadu modelů spojených často jen disciplínou tvůrců, která může být podpořena i nástroji CASE.

Objektově orientovaný přístup

Objektově orientovaný přístup (OOP) je založen na teorii objektového datového a výpočetního modelu. (Vzhledem k neustálenosti českého názvosloví budeme zde používat pojmy "objektový" i "objektově orientovaný", neboť je chápeme tak, že v příslušném kontextu vyjadřují totéž). Kořeny OOP lze vysledovat přibližně od konce 60. let především v konstrukci simulačních systémů (systém Simula-67) a dále v experimentech souvisejících s výzkumem nových architektur počítačů, operačních systémů a uživatelských rozhraní (Smalltalk) a s výzkumem v oblasti reprezentace znalostí.

V klasickém pojetí, jak již bylo popsáno, se pozornost při tvorbě programu zaměřuje na rozpoznání a návrh datového a funkčního modelu. V OOP se na rozdíl od klasického pojetí operuje s objekty, které popisují jak datovou, tak i procesní stránku řešené problematiky. Objekt je, chceme-li jej tak vidět, určitá jednotka, která popisuje (modeluje) nějakou část reálného světa. Datová povaha objektu je dána tím, že objekty se skládají z příslušných **vnitřních dat** (složky objektu), což jsou v rozumném případě opět nějaké jiné objekty. Funkční povaha každého objektu je dána tím, že každý objekt má jakoby okolo svých vnitřních dat obal či zeď, která je tvořena množinou malých samostatných částí kódu, jenž jsou nazývány **metodami**. Metody slouží k tomu, aby popisovaly, co daný objekt **dokáže dělat** např. se svými vnitřními daty (objekty). S vnitřními daty nějakého objektu lze manipulovat (číst, nastavovat, měnit) pouze pomocí kódu nějaké metody tohoto objektu. Jinými slovy to znamená, že objekt dovoluje provádět jen ty operace, které povoluje jeho množina metod. Data uvnitř objektu jsou v tomto obalu z metod **zapouzdřena** a navenek nejsou přímo dostupná. Objekt je navenek **nedělitelná entita**. Všechno, co lze do počítače uložit, je možné transformovat do podoby nějakého objektu. Objektem mohou být části operačního systému počítače, např. ovladač tiskárny (data: údaje potřebné pro tisk a nastavení stránky, metody: vytiskni znak, vytiskni řádek, vytiskni obrázek, odstránkuj atd.) nebo okno na obrazovce (data: obsah okna, poloha okna, barvy a metody: (vytiskni text, vytiskni obrázek, přesuň, zavři, zvětši atd.). Objektovou podobu mohou mít i "objekty" z reálného světa, který vkládáme do počítače, jako například osoby (data: jméno, adresa, číslo OP, datum narození a metody: ukaž jméno, změň adresu, prozraď svůj věk, přihlaš se na adresu xy, atd).

Pro objekty jsou definovány pouze dvě přípustné akce. První z nich je **pojmenování** nějakého objektu a druhou je tzv. **poslání zprávy**. Zpráva představuje vlastně žádost o provedení nějaké operace - metody nějakého objektu. Součástí zprávy mohou být tzv. parametry zprávy, což jsou vlastně data - objekty, které představují **datový tok** směrem k objektu přijímajícímu danou zprávu. Například ve zprávě "přihlaš se na adresu: 'Jilemnická 35' ", kterou posíláme objektu pojmenovanému třeba "Jan Novák", je tímto parametrem údaj 'Jilemnická 35'. Pokud objekt "Jan Novák" neví, jak na zprávu reagovat, tj. nemá k této zprávě odpovídající metodu, tak v systému dochází v závislosti na jeho realizaci k chybovému hlášení, anebo k poslání nějaké náhradní zprávy.

Protože poslání zprávy má mít za následek provedení kódu jedné z metod objektu, který zprávu přijal, tak tento zmíněný kód také vesměs dává nějaký výsledek v podobě nějakých dat - objektů, které představují **datový tok** ve směru od objektu - příjemce zprávy k objektu - vysílači zprávy. Pokud použijeme příklad s objektem "Jan Novák", tak po poslání zprávy "sděl svůj věk" u tohoto objektu dojde ke spuštění metody, jejíž kód s pomocí údaje o roku narození (vnitřní data objektu) a letošním roce (údaj získaný třeba z objektu "operační systém") vypočítá stáří a vrátí ho jako výsledek volání této zprávy.

V popsaném mechanismu je patrná určitá podobnost mezi voláním funkce (procedury) v klasickém smyslu s případnými parametry a posláním zprávy, mezi provedením funkce a provedením metody a mezi výslednou hodnotou po provedení funkce a po provedení metody. Na rozdíl od volání funkcí je však posílání zpráv mnohem silnějším nástrojem. V objektovém modelu výpočtu totiž poslaná zpráva, na rozdíl od volání funkce, nemusí trvale identifikovat příslušnou metodu, která se bude provádět. O výběru a spuštění metody totiž rozhodne obvykle až objekt, který je příjemcem zprávy, což se označuje jako tzv. **pozdní vazba**. Je-li příslušný objekt a metoda se zprávou svázána již v době překladu kódu programu, jedná se o tzv. **brzkou vazbu**. V případě brzké vazby však není žádného výpočetního rozdílu mezi voláním klasických funkcí a posíláním zpráv.

S využitím pozdní vazby tedy zprávu "vytiskni text: 'Dobrý den!' ", která má jeden parametr, můžeme poslat objektu "okno" na obrazovce, nebo i objektu "tiskárna". Oba objekty sice mají svoje navzájem vnitřně velmi odlišné metody pro tisknutí textu, ale dané objekty si tyto odlišnosti nechávají pro sebe, což je pro tvůrce programu přínosné v tom, že se o ně nemusí starat a není nucen je zohledňovat při psaní kódu programu.

Je-li tedy stejná zpráva poslána různým objektům, může mít za následek spuštění různých metod, což znamená, že v algoritmu úlohy není třeba brát v úvahu vnitřní detaily jednotlivých objektů, protože objekty samy mají schopnost **sebeidentifikace** (samy vědí, co jsou a jak mají reagovat na zprávy). Je samozřejmé, že jednotlivé kódy metod se skládají z dalších posílání zpráv dalším objektům.

Objektová analýza

Modelujeme-li pomocí nějaké objektové techniky, tak se vždy snažíme najít množinu - systém objektů, které popisují jednotlivé části z modelované problémové oblasti. Tyto objekty spolu komunikují pomocí posílání zpráv. Není tu nějaký souvislý kód programu. Místo něj se aplikace skládá z malých částí kódu popisujících jednotlivé objekty. Celý systém nepotřebuje mít v ideálním případě hlavní program, který by popisoval průběh aplikace od začátku do konce, neboť objektový program se více než klasicky strukturovanému programu podobá asynchronnímu simulačnímu systému řízenému sledem událostí přicházejících z vnějšího prostředí. O výhodách tohoto modelu bylo již napsáno velmi mnoho. Na tomto místě se proto spokojíme s upozorněním na možnost tvorby a ladění takovýchto programů za jejich chodu a na možnost inkrementálního programování.

Základem objektového modelu je tedy objekt, který se skládá z jiných objektů a který s ostatními objekty komunikuje pomocí posílání zpráv. **Skládání objektů** do sebe je sice velmi důležitá vlastnost objektů, ale není jediná.

Další vlastností je takzvané **dědění mezi objekty**, které nám umožňuje definovat vnitřní strukturu a metody nových objektů pomocí jiných již existujících objektů. To, že se při popisu a programování nových objektů odkazujeme na již existující objekty, nám umožňuje tyto nové objekty definovat pouze tím, jak se od stávajících objektů liší, což přináší kromě obrovských úspor při psaní programů také možnost vytvářet nadtypy a podtypy mezi objekty, vyčleňovat společné vlastnosti různých objektů atp. Umožňuje-li systém přímo (ne zprostředkovaně přes jiné objekty) dědit od více než jednoho objektu, potom podporuje tzv. **vícenásobné dědění**.

Velmi důležitým pojmem v oblasti OOP je také pojem **třídy objektů**. Máme-li v systému velké množství objektů, které mají sice různá data, ale shodnou vnitřní strukturu a shodné metody, potom je výhodné pro tyto objekty zavést jeden speciální objekt, který je nazýván **třída**, a který pro všechny tyto objekty dohromady shromažďuje popis jejich vnitřní struktury a jejich metody. Takto popsané objekty jedné třídy jsou nazývány **instancemi** této třídy. V některých systémech jsou třídy implementovány pouze jako abstraktní datové typy a ne jako objekty, což mj. znamená, že nemohou být vytvářeny či modifikovány za chodu programu.

V objektových systémech se můžeme setkat ještě s jedním důležitým druhem hierarchie mezi objekty - tzv. **závislostí** mezi objekty. V závislosti objektů rozeznáváme dva typy objektů: řídicí objekty - tzv. **klienty** a řízené objekty - tzv. **servery**. Žádá-li nějaký klient provedení nějaké operace od serverů, tak posílá zprávu, neboť zpráva je i zde jedinou možností, jak spustit nějakou operaci. Na rozdíl od standardního poslání zprávy, kdy je třeba znát příjemce zprávy, v případě závislých objektů klient nepotřebuje znát svoje servery, protože oni sami jsou povinni svého klienta sledovat a zprávy od něj zachycovat. Zprávu, která je signálem pro servery, tedy objekt klient posílá bez udání příjemce. Tato hierarchie bývá úspěšně využívána při tvorbě grafických uživatelských rozhraní a při tvorbě komplikovanějších simulačních modelů.

Úloha konceptuálního modelování

Techniky objektově orientované alalýzy a designu tedy spočívají v postupném modelování daného problému pomocí prostředků OOP. Ve srovnání s klasickými technikami je objektový model bližší svými funkcemi realitě i lidskému myšlení a nedochází v něm mj. ke vzájemnému odtržení datové a funkční stránky popisovaného systému. O to více tímto roste důležitost konceptuálního modelu konkrétního řešeného problému. Úloha konceptuálního modelování v OOP je vlivem odlišných vlastností objektových modelů minimálně stejně tak důležitá, jako v klasických technikách návrhu. Vyjmenujme nejdůležitější funkce konceptuálního modelování:

- Jedná se o nástroj, který tvůrcům systému umožňuje formálním způsobem modelovat řešenou realitu a najít tzv. **esenciální model** řešeného systému, což umožňuje formálně reprezentovat požadavky zadavatele.
- Konceptuální model je základním nástrojem pro návrh (design) programu - programové struktury.
- Slouží jako dorozumívací a jednotící prostředek pro členy vývojového týmu, pomáhá při ladění a testování.
- Je základem pro tvorbu dokumentace.
- Slouží jako dostatečně formální a přesný dorozumívací prostředek mezi zadavateli systému a řešiteli systému. Zadavatele nezatěžuje zbytečnými detaily konkrétní počítačové implementace a použitého programovacího jazyka a řešitele nezatěžuje různými fakty z problémové reality řešeného systému, která nejsou podstatná pro vytvářenou softwarovou aplikaci.

Vlastnosti objektově orientované analýzy a designu

Vlastnosti OOP se promítají nejen do podoby programu, ale i do procesu jeho tvorby. O celém životním cyklu budeme hovořit v následující části tohoto článku. Nyní se zaměřme na některé jeho části, a to konkrétně na objektově orientovanou analýzu a návrh (design).

Jak již bylo naznačeno, v objektově orientované analýze nedochází při tvorbě modelu ke vzájemnému odtržení datového a funkčního pohledu na modelovaný systém. V ideálním případě je dokonce možné vystačit s jedním nástrojem (jeden typ diagramu) a s jednou technikou. V případech, kdy to možné není, si však stále jednotlivé modely, i když jsou zaměřeny na různé části problému, zachovávají mnohem větší stupeň vzájemných souvislostí, než je tomu u klasické analýzy.

Lze říci, že analýza v OOP končí tehdy, je-li řešený problém popsán mj. v příslušném konceptuálním modelu, který úplně a formálním způsobem popisuje zadání daného problému. Design je v OOP přímým pokračováním analýzy bez nějakého extrémně výrazného přechodu. Na rozdíl od klasického designu se zde v maximální míře používají stejné nástroje, jaké sloužily i v analýze. Konceptuální modely, které ve fázi analýzy popisovaly řešený problém (tj. zadání), se ve fázi designu rozpracovávají do podoby použitelné pro jejich počítačovou implementaci.

Vzhledem k povaze kódu objektového programu OOP nezná ani ostrý přechod mezi fází designu a fází psaní kódu programu. V případě použití vyspělých objektových programovacích nástrojů (vývojových prostředí) a programovacích jazyků lze považovat psaní kódu programu za ukončení designu na nejnižší úrovni granularity. Takto viděný objektový program je potom vlastně stále model, který je však rozpracován do takové podrobnosti, že jeho jednotlivé elementy jsou totožné s výrazovými prostředky použitého programovacího prostředí. V klasickém pojetí je mezi designem a mezi kódováním programu velmi ostrý přechod, který je dán na jedné straně počtem a vlastnostmi jednotlivých používaných modelů a na straně druhé používanými programovými prostředky a požadavky na podobu, kódování a ladění klasického programu.

Fáze životního cyklu v OOP

V klasických metodách návrhu se rozeznávají dva základní modely pro popis životního cyklu programového díla od zadání, přes analýzu, design (návrh), k implementaci, ladění, testování, provozu a údržbě. Jedná se o tzv. "vodopádový model" nebo o tzv. "spirální model". Vzhledem k vlastnostem OOP, je pro tvorbu objektových aplikací vhodnější po určité úpravě spirální model, jak ho uvádí obrázek.

Jeden cyklus spirálního modelu v OOP má fáze zadání, analýzy, designu (návrhu), implementace, testování a provozu. Fáze zadání a analýzy se dohromady označují jako **stadium expanze**, protože při nich dochází ke hromadění informací potřebných pro vytvoření aplikace. Stadium expanze končí s dokončením analytického konceptuálného modelu, který na logické úrovni reprezentuje požadované zadání a formálně popisuje řešený problém.

Zbývající fáze od designu přes implementaci k testování a provozu se označují jako **stadium konsolidace**. Je tomu tak proto, že v těchto etapách se model, který je produktem předchozí expanze, postupně stává fungujícím programem, což znamená, že na nějakou myšlenkovou "expanzi" zadání zde již není prostor ani čas. V tomto stadiu je také počítáno s tím, že od některých idejí z expanzního stadia bude třeba upustit vzhledem k časovým, kapacitním, implementačním nebo i intelektuálním schopnostem. Odtud tedy název tohoto stadia.

Na životním cyklu v OOP je také zajímavý samotný programový produkt, který je v OOP vždy považován za jakýsi prototyp, neboť může kdykoliv posloužit jako součást nového zadání při nastartování dalšího vývojového cyklu. V OOP tedy neznáme prototypy v klasickém pojetí, protože každý produkt (v počítačové angličtině se označuje jako "deliverable") je vzhledem k zadání, ze kterého vyšel, dostatečně funkční a může být proto používán. Stejně však také může sloužit pro tvorbu nového produktu - a to nejen skrze zkušenosti s ním, ale přímo i svým programovým kódem jako kostra či výchozí model expanze vývoje nového produktu. Vzhledem ke vztahu předchozího a následného produktu vesměs nebývá velkým problémem přechod od užívání jednoho produktu k druhému za jejich plného provozu.

I když objektový spirální model přináší více volnosti do životního cyklu aplikace než klasické modely, tak přesto i zde platí některá omezení. Nejdůležitějším z nich je skutečnost, že není možné jednu část cyklu řešit pomocí klasické techniky a nástroje a jinou pomocí objektové techniky a nástroje. Střídání klasických a objektových technik podle našich zkušeností přináší více škody než užitku a je dokonce horší, než střídání objektové (resp. klasické) techniky s metodou "na koleně". Méně problematické (v některých případech dokonce prospěšné) je provedení celé analýzy a designu v duchu objektových technik s konečnou programovou implementací v neobjektovém programovacím prostředí. V tomto případě přináší největší problémy pouze nedostatečný výrazový aparát daného programovacího jazyka.

Vzhledem ke vlastnostem OOP je kdykoli možné (jak dokazují mnohé CASE systémy) použít objektové programovací prostředky při implementaci modelů vzniklých na základě klasické analýzy a designu. Lze totiž dokázat, že klasický model výpočtu může (samozřejmě na určité úrovni abstrakce) představovat zvláštní případ objektově orientovaného modelu. Jak je vidět, s mísením klasických technik a nástrojů s objektovými je to velmi podobné jako s krevními skupinami. Některé lze kombinovat a jiné ne. A stejně jako v medicíně i zde platí, že nejlepší je zachování jednoho přístupu pro všechny fáze životního cyklu.

Jakou metodologii použít

Přibližně od roku 1990 bylo doposud vyvinuto několik metodologií pro objektově orientovanou analýzu a design. Všechny tyto metody mají společný cíl, kterým je nalezení co nejlepšího objektového modelu řešené aplikace a jeho následného využití pro tvorbu programu. Mezi dnes nejpoužívanější metodologie patří:

- **OMT - Object Modelling Technique**, jejímž autory jsou J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy a W. Lorensen. Metoda byla poprvé publikována nakladatelstvím Prentice Hall v knize "Object-oriented Modelling and Design" v roce 1991. Technika je zajímavá tím, že je v podstatě hybridní technikou zahrnující jak objektové, tak i klasické nástroje. Nejčastěji se používá pro návrh databázově orientovaných aplikací.
- **OOATool - Object-Oriented Analysis Tool**, jejímž autory jsou P. Coad a E. Yourdon (autor již zmiňované klasické strukturované metody). Metoda byla publikována v časopise American Programmer v roce 1989 a posléze v knihách obou autorů "OOA" a "OOD" nakladatelství Yourdon Press. Od počátku je k dispozici jako stejnojmenný CASE prostředek podporující jazyky Smalltalk a C++. Z uvedených tří technik je nejsnažší pro pochopení a zvládnutí, a proto se poměrně často používá ve školství.
- **OOSD - Object-Oriented Software Development**, jejímž autorem je G. Booch. Metoda byla poprvé publikována v roce 1991 v knize nakladatelství Benjamin/Cummings "Object-Oriented Design with Applications". Tato poměrně velmi složitá metodologie je určena pro týmový vývoj rozsáhlých aplikací v jazyce C++, kde se také velmi úspěšně používá.

Pro výše uvedené tři metodologie jsou dnes k dispozici i CASE prostředky podporující především jazyky C++ a Smalltalk. Jejich společnou nevýhodou je však skutečnost, že ani jedna z není není uznávaným standardem. Podle statistických výzkumů v USA a Velké Británii ani jedna z nich není používána tvůrci objektových programů z více jak 20%. Domníváme se, že to způsobuje jednak nedostatek zkušeností s nimi, protože se jedná vlastně o první generaci objektových metodologií, a také proto, že ani jedna z nich nepodporuje v dostatečné míře všechny možnosti, které nabízí OOP. Oba dva důvody zřejmě zmíněným metodám zabraňují stát se všeobecně uznávaným standardem.

Kromě tří výše uvedených metodologií existuje ještě několik vesměs velmi kvalitních technik nebo i jen nástrojů, jejichž rozsah je však omezen na několik publikací nebo na výuku softwarového inženýrství na vysokých školách a nebo se jedná o firemní know-how. Jedná se především o:

- **OOSE - Object-Oriented Software Engineering** autora Ivara Jacobsona. Metoda je velmi kvalitně popsána ve stejnojmenné knize, která je považována spolu s knihami P.Coda a E.Yourdona za bible OOP. Vzhledem k tomu, že metoda má původ ve skandinávské škole, tak je velmi zajímavá pro aplikace z oblasti simulace a řízení.
- **Object-Oriented Structured Design notation** autorů A.I. Wassermanna, P.A. Pitchera a R.J. Mullera publikovaná poprvé v časopise IEEE Computer č. 23(3) 1990. Metoda je zajímavá především svojí notací.
- **OOER notation** autorů K. Gormana a J. Choobineha poprvé publikovaná na 23. mezinárodní konferenci o informatice (computer science) na Havaji v létě 1991. Metoda používá notaci ER diagramů v klasické Chenově syntaxi rozšířené o objektové vlastnosti. Je výhodná pro použití v objektově orientovaných databázích.

Podle našeho názoru dosud není vyvinuta taková objektová metodologie, která by podporovala všechny možnosti, které přináší OOP a přitom byla minimálně tak elegantní, jako dnes již klasická Yourdonova strukturovaná metoda. Domníváme se, že tomuto ideálu se nejvíce blíží

Coad-Yourdonova metodologie, kterou jsme si pro vlastní potřebu částečně modifikovali o vlastní zkušenosti s OOP, mezi které patří kromě obsahových změn také úprava původního objektového diagramu. Úpravy měly za cíl grafická a obsahová zjednodušení, větší podporu teoretických prostředků OOP a větší podobnost s diagramy používanými v klasické strukturované analýze. Tyto úpravy nakonec vyústily v nový nástroj námi pojmenovaný jako ORD (Object-Relationship Diagram), který používáme i v tomto článku.

Jak objektově modelovat

Všechny metodologie mají společné hledání a využití objektového modelu (či modelů), který co nejlépe vystihuje řešenou problematiku. Vzhledem k tomu, že se jedná o velmi složitý a zároveň velmi zodpovědný úkol, tak bylo vypracováno několik postupů, jak si tuto práci ulehčit. Coad a Yourdon proto vymysleli tzv. **vícevrstevný model** (multi-layer), jehož podobu lze v nejrůznějších formách najít i v jiných metodologiích. Vícevrstevný model je založen na tvorbě objektového modelu, metodou postupného doplňování od vrstvy **subjektů**, přes vrstvu **objektů**, vrstvu **struktury** až k vrstvě **služeb**, které se skládají na sebe tak, jakoby byly nakreslené na průhledných fóliích. Cílový objektový model je tvořen souhrnem struktur z jednotlivých vrstev.

Zmíněný vícevrstevný model Coad a Yourdon ještě doplňují o tzv. komponenty (components), kterými jsou relativně samostatné objektové modely (vytvořené metodou více vrstev). Každá z komponent popisuje řešený problém z jiné stránky. Zpravidla i v té nejjednodušší aplikaci můžeme vždy rozeznat komponentu **vlastního problému** (problem domain) a komponentu **uživatelského rozhraní** (human interaction). Tyto komponenty a především komponenta vlastního problému by měly být nedílnou součástí analytického konceptuálního modelu. Právě důslednost v rozlišení těchto dvou komponent problému je téměř vždy přímo úměrná celkové kvalitě výsledného programu. Jen na okraj si připomeňme dobře známé odstrašující příklady různých programů, které a) odhalují vnitřní struktury dat a algoritmy na svoje rozhraní a otravují tak uživatele zbytečnými hláškami o indexování, různými teploměry ukazujícími průběh všech možných procesů apod. anebo b) průběh výpočtu podřizují sledu obrazovek, menu či oken. Kromě diskutovaných komponent je možné podle typu řešené úlohy ještě rozeznat komponentu **zpracování dat** (data management) a **řízení úloh** (task management).

I když je tvorba objektového modelu usnadněna jeho rozdělením do oddělených vrstev a komponent, tak se stále jedná o velmi obtížný a přitom klíčový problém. Zatím není k dispozici (a asi nikdy nebude) nějaký jednoznačný návod, který by posloužil k efektivní a optimální transformaci slovního zadání do formální podoby modelu. Velkou roli zde hraje zkušenost návrháře a jeho schopnost komunikovat se zadavateli. Mnoho problémů spojených s výběrem vhodné struktury např. ve vrstvě subjektů, se dá vyřešit metodou přímého napodobení problémové reality. V OOP je totiž ve větší míře, než tomu bylo doposud u počítačů zvykem, možné vytvářet struktury přímo podle obrazu skutečnosti.

Velkým problémem však bývá rozpoznání objektů a metod. Některé části problému lze totiž modelovat nejen jako objekty, ale i jako metody. Dokonce je jich více, než by se mohlo na první pohled zdát. K řešení těchto problémů byly a jsou vyvíjeny různé různě složité techniky. Tou nejjednodušší a také nejméně účinnou je následující návod: "Popište problém slovy. Podtrhejte si podstatná jména a máte objekty. Podtrhejte si slovesa a máte metody". Mezi ty složitější patří např. metoda tzv. analýzy chování (behavioral analysis), kterou vyvinul v Dortmundu v roce 1993 G. Heeg (zakladatel stejnojmenné firmy zabývající se objektově orientovanými technologiemi založenými na Smalltalku).

Transformace problému do objektového modelu (a použití OOP vůbec) totiž do určité míry souvisí i s filosofií v programování. Považujeme-li objekty za materii, potom je možné OOP prohlásit za určitou formu "materialismu" v programování. Neexistuje zde totiž žádný proces (metoda), který by nepříslušel k nějakému objektu, což je v příkrém kontrastu s "idealisticky" pojatým klasickým programem, který má hlavní tělo a podprogramy řízené jakoby samostatně (bez materie) a manipulují s jednotlivými proměnnými až na výjimky bez ohledu na jejich obsah. V OOP musí být každá činnost spojena s objektem (konečnou podobu každé činnosti vždy určuje nějaký objekt). Kromě toho mohou být objektem i události, jejichž informační hodnota nekončí s jejich provedením, ale je třeba ji později opět využívat či zpracovávat. Takové události nemohou být metodami, protože si je systém musí nějakým způsobem zapamatovávat, a proto je "materializována" na objekty - samozřejmě s příslušnými prováděcími metodami. Vidíme tedy, že objektů v objektově orientovaném systému je víc, než je těch objektů z reálného světa, na které si lze přímo "sáhnout". Dalšími kandidáty na objekty jsou kupříkladu generalizující pojmy (např. savec či osoba) nebo agregované pojmy (např. plánovací oddělení nebo živočišná výroba), které také v reálném světě jako individua nepotkáváme.

Samostatnou kapitolou je problém stanovení vhodné struktury objektů. Není to lehký úkol, i když již víme, se kterými objekty se bude pracovat. Největší chybou v modelování vhodné struktury objektů je podle naší zkušenosti podcenění možností, které přináší skládání objektů. Je to způsobeno zřejmě tím, že ostatní hierarchie (dědění, třídně-instanční vazba, nebo závislost) jsou v OOP nové, a proto se na ně v literatuře zabývající se výkladem OOP klade větší důraz, než na skládání, které již dříve bylo více méně známým pojmem. Vede to k mylnému přesvědčení, že v objektovém systému se musí na jedné straně všechno mezi sebou dědit (i to, co dědit nelze) a na druhé straně, že se objekty skládají pouze z atomických složek (čísla, znaky, texty) a ne z jiných objektů.

Nejpádnějším důvodem pro dědění mezi objekty by mělo být především využívání vlastností (metod) objektů, od kterých se dědí. Je však třeba mít na paměti, že zděděná vlastnost (vnitřní struktura či metoda) však slouží a náleží jen tomu objektu, který ji zdědil. Má-li tedy objekt od kterého se dědí nějaká svoje konkrétní vnitřní data, tak tyto data jsou dědícímu objektu utajena (zapouzdřena) úplně stejně, jako jakémukoliv jinému objektu. Využívat dědění pro přenos nějakých dat od objektu ze kterého se dědí k objektu, ke kterému se dědí, je proto většinou dost nevhodné (i když je možné takovou strukturu naprogramovat).

Pokud jeden objekt potřebuje od druhého objektu získávat jeho data nebo dokonce jeho data zpracovávat jeho metodami (bude mu posílat zprávy), tak jedinou rozumnou možností je realizovat druhý objekt jako složku objektu prvního. Složení druhého objektu do prvního totiž zaručí v metodách prvního objektu jeho snadnou dostupnost pro posílání zpráv, protože se složením druhý objekt stává součástí vnitřních dat prvního objektu. Hierarchie skládání v objektových modelech také plní stejnou úlohu, jakou má v klasickém ER modelu jednosměrně interpretovaná relace mezi entitami nebo atribut.

Hierarchii závislosti volíme tehdy, když jeden objekt potřebuje využívat schopnosti (zpracovávat jeho data jeho metodami) druhého objektu, ale nepotřebuje ho přímo znát. Pokud používáme programovací prostředek, který závislost nepodporuje, můžeme zvolit skládání, které zaručí podobnou funkčnost. Na rozdíl od využití závislosti je však toto řešení méně elegantní, neboť objekty, které by mohly být typu server, jsou pasivní, a proto objekty, které by mohly být typu klient, musí samy spouštět příslušné akce (musí samy jednotlivým "serverům" posílat příslušné zprávy).

Příklad úlohy - lékařská ordinace

Na závěr jsme vybrali příklad objektového návrhu jednoduché aplikace pro potřeby lékařské ordinace. Program má za úkol evidovat pacienty, u kterých je třeba zaznamenávat základní osobní údaje, dále jednotlivé lékaře, u kterých kromě osobních údajů se navíc eviduje odbornost každého z lékařů. Pacienti chodí k různým lékařům na vyšetření, přičemž na jednom vyšetření je pouze jeden lékař a jeden pacient a u každého vyšetření se zaznamenává druh vyšetření, diagnóza, datum vyšetření a datum další plánované návštěvy. Kromě obvyklých akcí musí systém jednotlivým lékařům pomáhat při posílání pozvánek pacientům k vyšetřením (prohlídkám).

Vzhledem k jednoduchosti úlohy byly rozpoznány pouze dva subjekty: "ordinace" a "uživatelské rozhraní". V subjektu uživatelské rozhraní se nachází objekt "Okno" a objekt "Tiskárna", u kterého předpokládáme, že již existuje jako součást operačního systému. V subjektu "ordinace" jsou rozpoznány čtyři typy objektů: terminátor "uživatel" a tři množiny instancí tříd "Osoba", "Doktor" a "Vyšetření". O třídě objektů "Osoba" předpokládáme, že je součástí nějaké knihovny.

Třída "Doktor" dědí od třídy "Osoba" všechny vlastnosti (strukturu i metody), k nimž přidává dvě metody "pozvi k vyšetření" a "vyber záznamy vyšetření" a také složku "odbornost" do struktury svých instancí. Ze zásad dědění objektů v OOP tedy vyplývá, že doktor je osoba jako každá jiná s některými vlastnostmi navíc, což mj. znamená, že každý doktor může být v navrhovaném systému pacientem v nějakém vyšetření.

Na rozdíl od dědičnosti, která je v našem příkladu použita pouze jednou, je hierarchie skládání objektů použita dvakrát. Poprvé mezi objekty doktor a vyšetření, kde každý doktor má složku, která je množinou objektů vyšetření (tak jako v reálném životě má každý doktor svoji schránku s kartičkami). Pro každé vyšetření platí, že smí "náležet" právě jednomu doktorovi. Není tedy žádné vyšetření bez doktora ani více jak jeden doktor na jednom vyšetření. Druhé použití skládání objektů je mezi vyšetřením a osobou, kde každý objekt vyšetření má složku pojmenovanou pacient s jedním objektem osoba (nebo doktor). V diagramu je vyznačeno, že tato složka nesmí být nikdy prázdná, protože není žádné vyšetření bez pacienta. Pro úplnost je třeba dodat, že zbývající složky o objektů doktor, vyšetření a osoba (odbornost, druh, datum, adresa, jméno, ...) jsou také nějakými složenými objekty (pokud je v našem systému objektem i text a číslo). Závislost je použita u objektů doktor (klient) a "Okno" s "Tiskárnou" (servery).

Kromě objektů, které mají přímý obraz v reálném světě, jsou v našem modelu ještě objekty vyšetření, které jsou ve skutečnosti procesy. Protože však jejich informace jsou používány i po jejich skončení, což je také důležitou součástí zadání tohoto příkladu, tak nám v modelu "zmaterializovaly" na objekty. Jako nepřímý důkaz správnosti tohoto počínání nám může posloužit fakt, že i v reálném světě má naše vyšetření nějakou hmatatelnou podobu, a to v příslušném papírovém formuláři o vyšetření. Na rozdíl od našeho objektového vyšetření však papírové vyšetření neumí ukazovat a vybírat údaje.

Na prvním ukázce ORD ordinace je zobrazen pouze subjekt "ordinace" s vrstvami objektů a struktury. Druhá ukázka ORD ordinace již přináší komplexnější pohled, ve kterém je kromě druhého subjektu "uživatelské rozhraní" spolu s vrstvami objektů a struktury také část vrstvy služeb, která se týká pozvání pacientů na plánovaná vyšetření k nějakému datu. Celý proces je rozpracován na úrovni komponenty datového zpracování. Na úrovni komponenty uživatelského rozhraní je naznačen pouze řídícími zprávami mezi závislými objekty.

Pro zpracování pozvání pacientů slouží čtyři metody. Objekty doktor mají dvě metody "pozvi k vyšetření" a "vyber záznamy vyšetření", objekty vyšetření mají jednu metodu "ukaž údaje" a objekty osoba (tedy i doktor) mají k tomuto účelu jednu metodu "ukaž osobní údaje".

- Metoda "ukaž osobní údaje" vrací jako výsledek jméno, příjmení a adresu osoby a je spouštěna zprávou z metody "ukaž údaje", která náleží objektům vyšetření. Objekty vyšetření tuto zprávu mohou posílat, protože objekt pacient (osoba) patří mezi vnitřní data vyšetření. Jinak řečeno metoda "ukaž údaje" objektu vyšetření manipuluje s jedním z vnitřních dat tohoto vyšetření.
- Metoda "ukaž údaje" vrací jako výsledek seznam údajů o vyšetření, které zahrnují diagnózu, druh a osobní údaje o pacientovi tohoto vyšetření (proto sama posílá zprávu). Metoda "ukaž údaje" je spouštěna zprávou z metody "vyber záznamy vyšetření" objektů doktor. Je to možné proto, že objekty doktor mají mezi svými vnitřními daty složku s množinou vyšetření. Tedy podobně jako v předchozím případě metoda "vyber záznamy vyšetření" manipuluje z jedním z vnitřních dat objektu doktor, kterým je množina provedených vyšetření tohoto doktora.
- Metoda "vyber záznamy vyšetření" je spouštěna zprávou s jedním parametrem "datum" a vypočítává a vrací množinu údajů o těch vyšetřeních, které jsou plánovány k danému datu. Tato metoda je spouštěna zprávou z metody "pozvi k vyšetření".
- Metoda "pozvi k vyšetření" je spouštěna uživatelem a nejprve posílá zprávu "vyber záznamy vyšetření k datu xy". Po obdržení výsledku vysílá signální zprávu pro závislé objekty (servery), že mohou začít tisk pozvánek (komentář do okna a celé pozvánky na papír v tiskárně). Protože předpokládáme náležité objektové vlastnosti i u objektů "Okno" a "Tiskárna", tak nám stačí jen jeden stejný signál pro oba tyto servery, neboť objekt "Tiskárna" si jej bude interpretovat jako tisk kompletní pozvánky na papír a objekt "Okno" jako zobrazení menšího kontrolního textu v nějaké části svého okna na obrazovce.

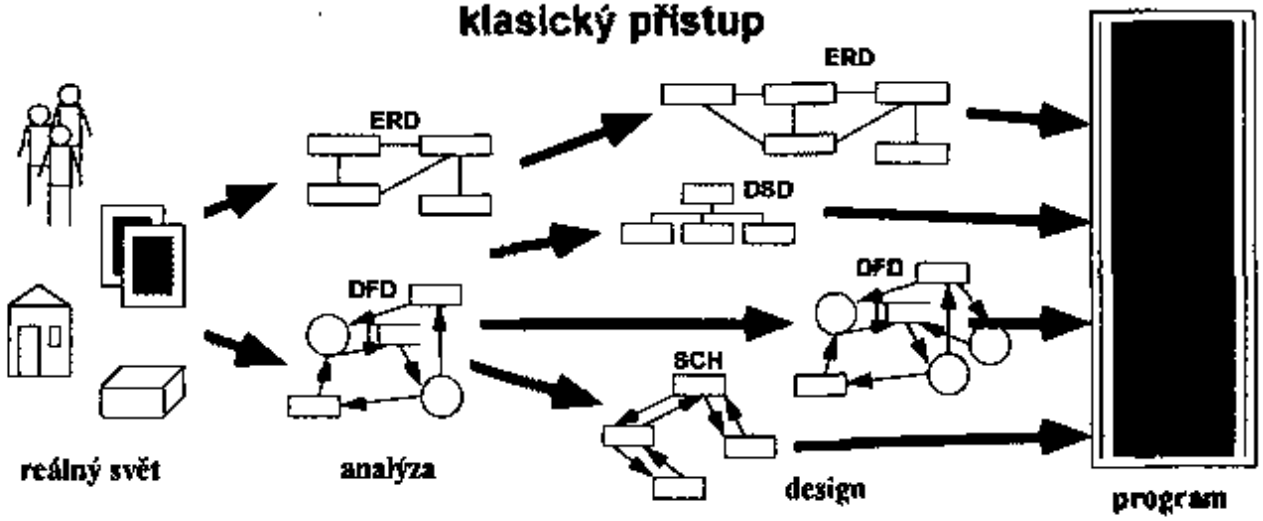
Závěr

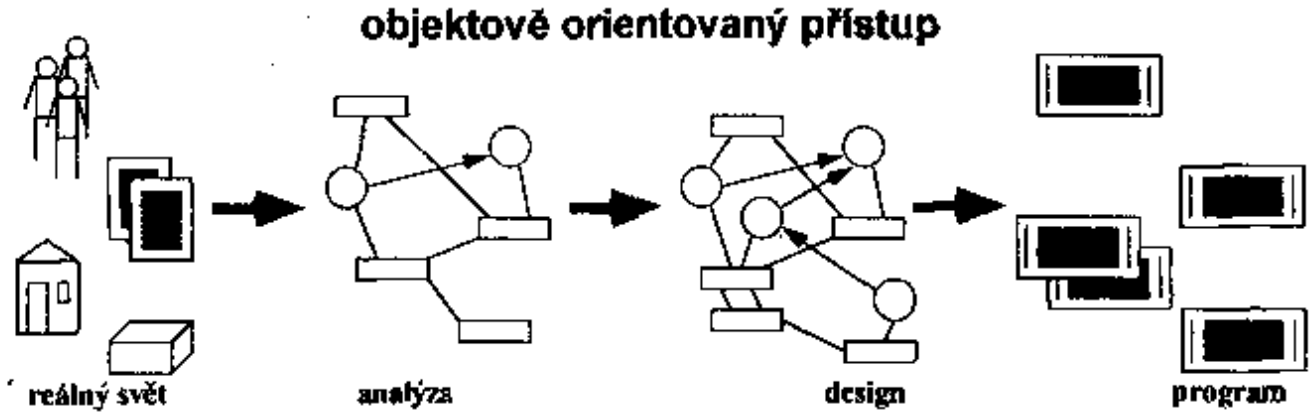
Objektově orientované techniky analýzy a designu se staly během několika málo let trvalou součástí softwarového inženýrství. Podstatnou měrou přibližují charakter tvorby programů k ostatním druhům inženýrské práce. Zkušenosti s využíváním OOP vypovídají o průměrně trojnásobném zvýšení produktivity programátorské práce (i té řemeslné a umělecké). Při tvorbě objektového softwaru šitého na míru zákazníkovi (např. evidenční systémy, účetní systémy, systémy pro podporu rozhodování) se běžně dosahuje 80% znovuvyužitelnosti kódu. Techniky OOA a OOD zkracují v některých případech tvorbu či úpravy objektových programových aplikací z týdnů a měsíců na dny až hodiny.

Doba, kdy se tvůrci programů obešli bez OOA a OOD (a všech dalších OO PTP - pojmů na tři písmena) a kdy znalosti o nich byly na úrovni narychlo prováděných překladů z nějaké víceméně náhodně získané (a samozřejmě velmi drahé) zahraniční publikace, by měla v našem zájmu (tím myslíme ty jedince, kteří to s tvorbou programů i v dnešní době stále myslí vážně) co nejdříve skončit. Věřte nám, že se vám to určitě vyplatí.

přehled vybraných vlastností nejpoužívanějších objektových systémů

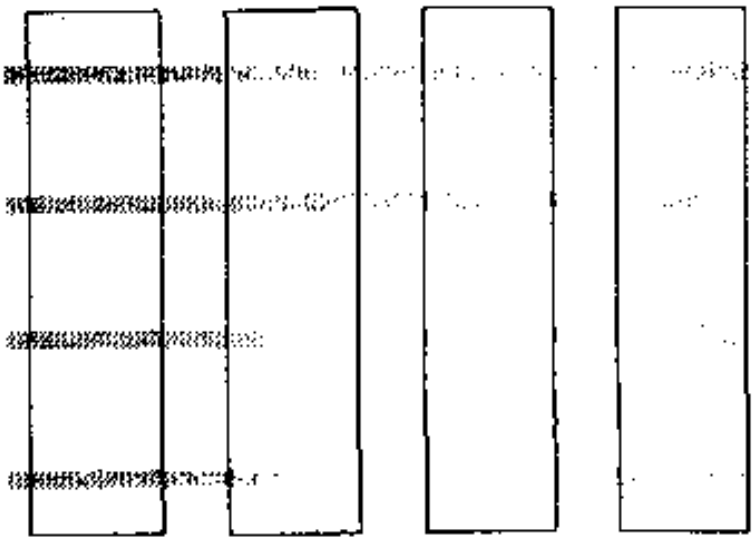
systém	podporuje závislost	vícenásobné dědění	neobjektové programování	druh vazby zpráva-metoda	třídy jsou objekty	programovatelnost za chodu
Smalltalk	ano	ne	ne	pozdní	ano	ano
Objective-C	ano	ne	ano	pozdní	ano	ne
C++	ne	ano	ano	brzká i pozdní	ne	ne
Turbo Pascal	ne	ne	ano	brzká i pozdní	ne	ne





MULTI-LAYER & MULTI COMPONENT MODEL

P.Coad, E.Yourdon 1989



problémová doména | zpracování dat | zpracování úloh (tasks) | uživatelské rozhraní

KOMPONENTY

VRSTVA SUBJEKTŮ
identifikace částí problému

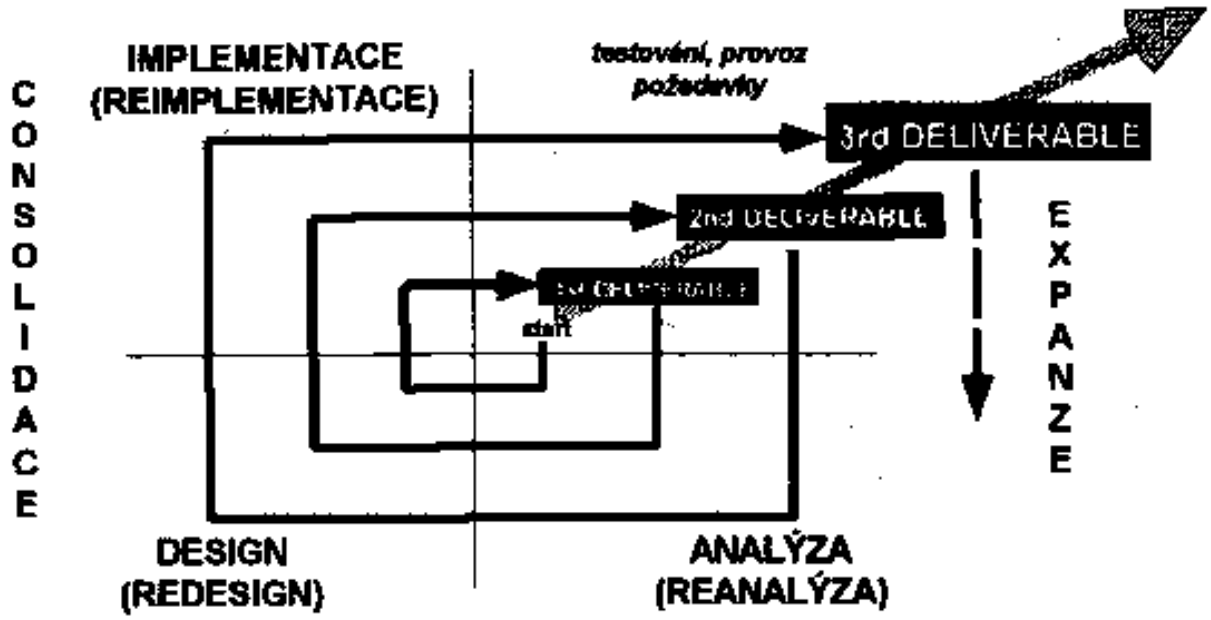
VRSTVA OBJEKTŮ
identifikace objektů, tříd, terminátorů ...

VRSTVA STRUKTUR
skládání, dědičnost a závislost mezi objekty

VRSTVA SLUŽEB
metody, zprávy, datové toky ...

SPIRÁLNÍ MODEL PRO OOA a OOD

J. Pugh 1993



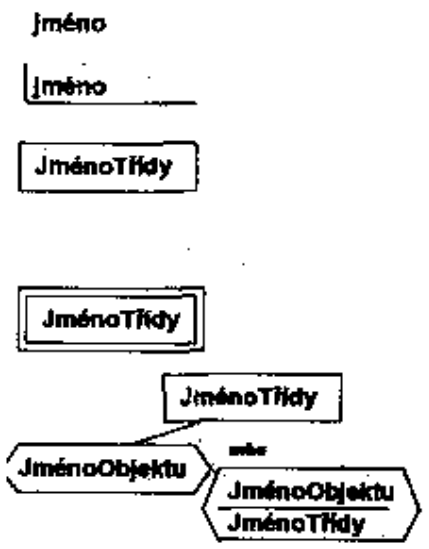
Na rozdíl od klasických prototypů jsou "deliverables" maximálně funkční. Není žádný podstatný rozdíl mezi prototypem a finálním produktem. S každým produktem je vždy možné pokračovat dalším cyklem.

VRSTVA SUBJEKTŮ

umožňuje

1) rozdělit projekt do menších částí
2) rozčlenit relativně samostatné části problému
3) vyznačit závislosti mezi částmi

VRSTVA OBJEKTŮ



Objekty bez udání třídy.

Terminátor - element rozhraní systému (např. uživatel nebo spolupracující program)

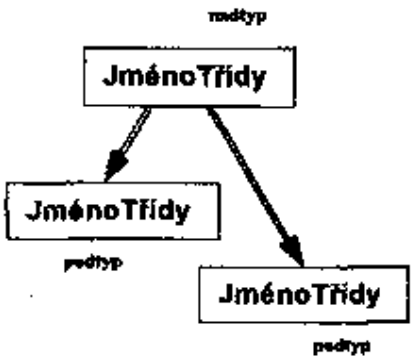
Objekty s udáním třídy. Reprezentuje množinu všech instancí dané třídy.

Objekty dané třídy (jako v předchozím případě) spolu s třídou samotnou jako objekt (vnější obdélník)

Jeden konkrétní objekt (konkrétní instance dané třídy).

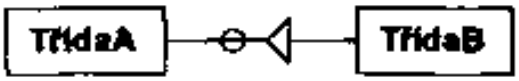
Znovupoužité objekty (z jiných částí systému nebo z knihoven), se od nově vytvářených objektů liší hvězdičkou u svého označení.

VRSTVA STRUKTURY
DĚDIČNOST

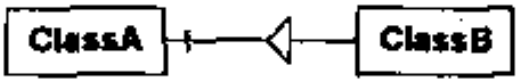


VRSTVA STRUKTURY
SKLÁDÁNÍ

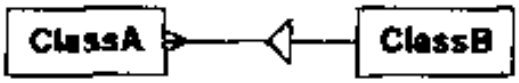
Objekty - instance třídy A obsahují pojmenovanou část, která je objektem - instancí třídy B.



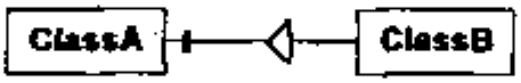
Objekty - instance třídy A obsahují pojmenovanou část, která je množinou objektů - instancí třídy B.



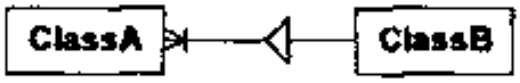
Každý objekt třídy B je součástí jednoho objektu třídy A a nebo není součástí žádného objektu třídy A.



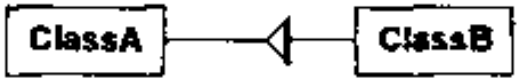
Každý objekt třídy B může být součástí více objektů třídy A a nebo není součástí žádného objektu třídy A.



Každý objekt třídy B je součástí právě jednoho objektu třídy A.

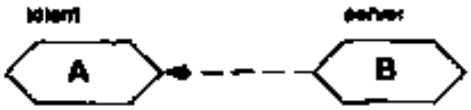


Každý objekt třídy B je vždy součástí alespoň jednoho objektu třídy A.

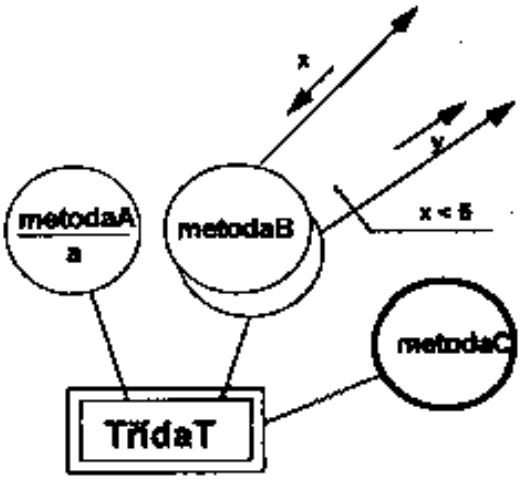


Třída A nesmí obsahovat takové objekty, které by měly pojmenovanou část prázdnou (tj. bez objektu třídy B).

VRSTVA STRUKTURY
ZÁVISLOST MEZI OBJEKTY



VRSTVA SLUŽEB

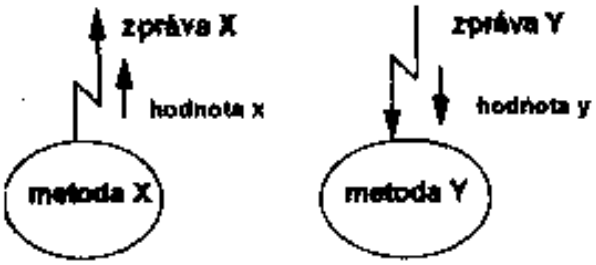


Třída T má dvě instanční zmetody "metodaA" a "metodaB" a jednu třídní metodu "metodaC".

U "metody A" je vyznačena hodnota "a", která představuje výsledek po provedení této metody.

U "metody B" jsou vyznačeny v časovém sledu za sebou dvě zprávy. "metoda B" nejprve pošle první zprávu, a po přijetí hodnoty "x" (výsledek volání zprávy) volá v případě, že platí výraz "x < 5" druhou zprávu, která obsahuje parametr "y".

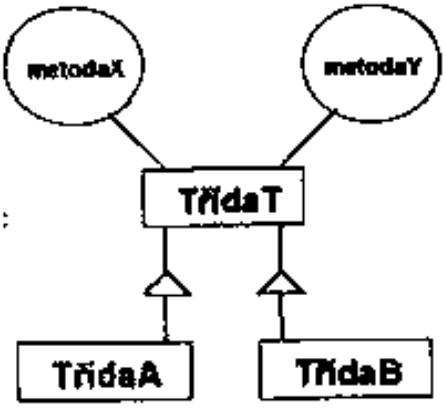
"Metoda C" je vyznačena dvojitou čárou, protože generuje nové objekty - instance třídy T.



"Zpráva X" je signálem řídícího objektu (klienta) a "zpráva Y" je signálem řízeného objektu (serveru) v závislosti objektů.

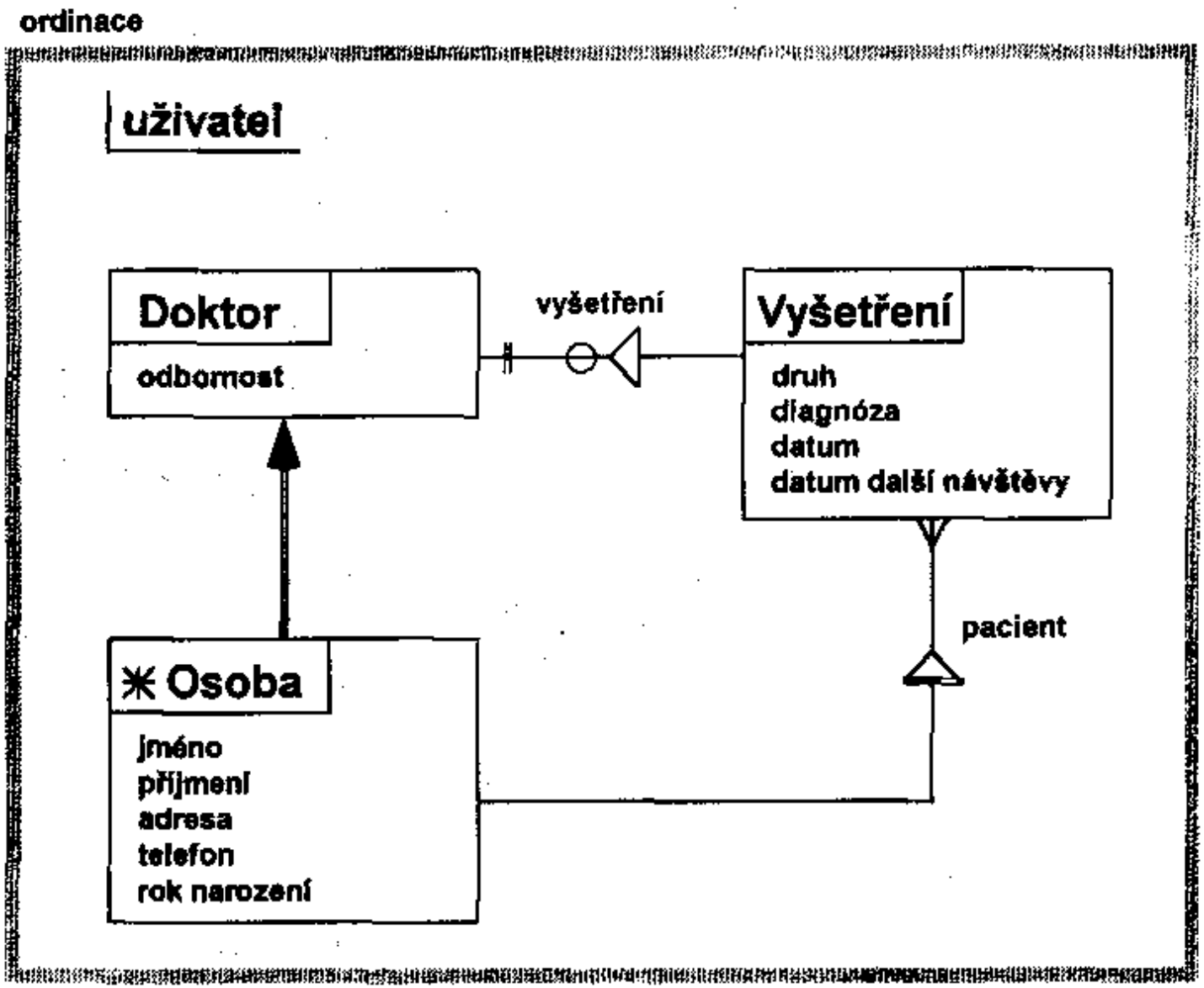
ÚSPORNÝ ZÁPIS

je totéž jako:



ORD - lékařská ordinace

subjekt, objekty, struktura



ORD - lékařská ordinace

subjekty, objekty, struktura, služby