Objektově orientované programování a modelování v diskusi a v příkladech

Autor	RNDr. Ilja Kraval, výhradní autorské právo
Adresa	firma Object Consulting, Lipina 100, 766 01 Valašské Klobouky
Mail	objects@objects.cz
URL	http://www.objects.cz
datum vydání	10.12.2001

Úvodní slovo

Dostává se vám do rukou první publikace shrnující nejčastější otázky s odpověďmi a příklady z objektově orientovaného programování a objektového modelování.

Tyto dotazy, dopovědi a příklady vznikly na základě diskusí, konzultací a řešení různých situací při konzultacích nad objektovým modelováním a OOP.

Pokud máte zájem poslat svůj vlastní dotaz nebo připomínku do dalšího vydání této publikace, pošlete ji ve tvaru přílohy zazipovaného WORD dokumentu na mail adresu mailto:objects@objects.cz . Pokud se tento dotaz nebo připomínka bude týkat některé ze zde uvedených kapitol, uveďte její číslo a název, na kterou se odvoláváte.

Těším se na Vaše maily,

autor RNDr. Ilja Kraval, mailto:objects@objects.cz

1. Strukturální programování versus OOP - úvaha

Autoři zajímavého článku z KIT VŠE mi nabídli zajímavý námět k zamyšlení. V uvedeném článku se uvádí příklad s židličkami v jídelně sdílenými mezi žáky. Jedná se o příklad klasického optimalizačního postupu, šetřícího v tomto případě zdroje. Ušetříme židličky: Každý žák si nebude přece nosit svou vlastní židličku. Budou se používat ty, které jsou v jídelně a pouze ty, které jsou třeba. Autor tímot poukazuje na šetření zdorjů ve strukturálním programování, ovšem toto je klasický příklad podporující přístup pomocí OOP a nikoliv jako příklad proti OOP. Autor článku totiž musel vysvětlit podstatu věci pomocí objektů, jako jsou Jídelna, Židle, Kolekce Židlí, Žák, Kolekce žáků, Obsazení židle žákem atd.

Tak to hned vidí "objektový analytik". To, o čem je řeč v článku a zdánlivě to podporuje strukturální programování, není nic jiného, než v teorii OOP tzv. pooling instancí objektů (prováděných Object Brokeru), které výrazně šetří zdroje. Přesně tak totiž pracuje Object Broker jako správce objektů.

V příkladu podobně "Správce jídelny" přiřazuje "volné a připravené" židličky žákům, a pokud žáci nemají již žádné židle k dispozici, tak buď (neoptimalizovaně) vytvoří židle nové nadpočetné (nepříjemná režie zrodu objektu, u židliček ještě horší než v programu, to by se žáci načekali, než by se nová vytvořila...) anebo pošle žáky do jiné Jídelny, tedy na jiný zdroj - tzv. metoda load balancingu (přesměrování na jiný procesor, na jiný stroj, kde jsou židličky volné). Anebo se žáci "frontují" a čekají, až se židličky uvolní a správce posílá žáky tam, kde je nejmenší fronta. To vše se v OOP a hlavně v třívrstvové architektuře modeluje a tím optimalizuje aplikace. Navíc nikde není řečeno, že objekty musí být "plněny" hned jak vzniknou. Dokonce mohou být "prázdné", kolekce cachované apod. Rozdíl mezi strukturálním programováním a OOP je v podstatě věci - strukturální nezná objekty židle, jídelnu, atd. ale pouze proměnné a funkce. A v takovém prostředí se velmi špatně optimalizuje (představme si neobjektový load balancing!). Takže příklad s židličkami v článku není nic jiného,než příklad na možnost volby různých OO modelů.

Rozpor v tom, že OOP je pomalejší se nestává kritickým. Zatímco před patnácti - dvaceti lety bylo hlavním problémem "jak se vměstnat do 64 Kb paměti a ušetřit každý byte", tak dnes je krize tvorby někde jinde - ve způsobu tvorby SW, v jeho údržbě, v jeho efektivním a rychlém návrhu, v řízení projektu atd. Tedy dnes není situace kritická v HW, ale v SW.

Na druhé straně "dnešní rychlosti a kapacity" se dosahuje velmi dobrými návrhy optimalizací, které jsou možné díky dobrému návrhu v OOP. Některé představy pomalosti OOP souvisí s tím, že se jako jediný scénář spolupráce objektů považuje ten "klasický" a tedy ten nejpomalejší (přesyp itemy z jedné kolekce do druhé apod.). Velkými zastánci strukturálního programování jsou uživatelé tzv. RAD, kde se program naklikává. Přitom ani největšího zastánce tzv. RAD "klikaček" nenapadne, že toto prostředí, kde se naklikává, je navrženo a napsáno OBJEKTOVĚ - provazují se objekty jako DBGrid, Datasource, Connection atd. mezi sebou a tím se získává funkcionalita. Přitom sám DBGrid v sobě nemá nataženy všechny údaje pro zobrazení, ale jako každý správný zapouzdřený objekt "toto pouze simuluje...". Tedy zastánci strukturálního programování používají program napsaný objektově a přitom paradoxně popírají výhody OOP. Problém pomalých OOP programů je v neznalosti tvůrců v možnosti modifikovat scénáře spolupráce objektů.

Přitom se vždy doporučuje nejprve navrhnout jako analytický model nejjednodušší klasický a tedy ten "pomalý" a teprve poté, až se dobrým rozborem zjistí kritické místo, tak toto optimalizovat. Jinak provedete předčasnou a možná chybnou optimalizaci.

Uvedu klasický příklad pomalosti a rychlosti a předčasné chybné optimalizace:

V jednom systému v daném scénáři:

- v ASP se zrodila business kolekce pomocí ActiveX DLL komponenty

- naplnila se cyklem itemy a data převzala z DB (zrod tolika objektů, kolik je záznamů v DB)

- poté se z kolekce přebíraly itemy cyklem a tato property

  se jako stringy v ASP skriptu doplnily do patřičných míst do tabulky mezi tagy TR a TD.

- ASP stránka se zobrazila

A teď jednalo se o kolekci vlastnící vícero členů a ukázalo se, že je to pomalé. Můžete tipovat, kde je kritické místo... Mohu říci, že osobně jsem tipoval špatně: Hádal jsem, že pomalé je to proto, že se kolekce musí zrodit a naplnit cyklem z DB, protože tam dochází cyklem ke zrodu objektů.

Není pravda, pomalé bylo převzetí itemů do ASP kódu, což si dotyční ověřili několika málo zkušebními testy. Uvedené naplnění proběhlo rychle, protože se odehrálo ve zkompilovaném kódu za hranicí komponenty a nikoliv ve skriptu.

Řešení? Kolekce vydala "XML string" a ten se pomocí XSL šablony přetransformoval do tabulky, čímž došlo ke zrychlení sekvence o několik řádů... (Samozřejmě existuje spousta jiných řešení...) Dobré je tedy nejprve navrhnout "klasiku", poté najít kritické místo a na něj se "fokusovat". Pokud bych předčasně hádal (v tomto případě špatně) a optimalizoval, tak bych začal dumat nad tím, jak tuto kolekci rychleji naplnit anebo jak "simulovat" její naplnění...

2. Public proměnná v Class modulu

2.1 Dotaz

Před časem vyšel na serveru článek jasně vysvětlující vhodnost zapouzdřovat jakýkoliv vnitřní stav objektu pomoci Let a Get Property a že není vhodné vůbec používat deklaraci Public v class modulu. S článkem lze určitě souhlasit, ale zároveň pro vývojáře vyplývá jakési dogma - Nikdy nepoužij Public v class modulu !! Měl bych dva případy, kdy podle mého názoru je třeba se nad tímto dogmatem zamyslet a zvážit použití Public:

Případ první, polymorfní interface: Když máme deklaraci class modulu, který je použit čistě jen jako interface, tedy se implementuje v jiném objektu a není v něm žádný kód. Pak bych v tomto interfacu použil Public. V objektu, kde se implementuje, totiž Visual Basic automaticky rozdělí tuto deklaraci do LET a GET metod. Z tohoto důvodu se jeví použití Public v interfacu jako důvodné.

2.2 Odpověď

K Vašim myšlenkám jenom jednu poznámku: Podle definice objektu (viz knihy a skripta o OOP) je objekt uzavřenou strukturou, která má vnitřní uzavřenou paměť (atributy), vnitřní metody (operace), obsahuje vnitřní objekty (asociace a agregace) a objekt má schopnost přijmout zprávu přes interface a na základě převodníku zpráva - metoda spustit metodu. Množina zpráv, které může objekt přijmout, se nazývá v OOP obecně interfacem. COM interface není nic jiného, než binární implementace této vlastnosti, tj. interfacu objektu obecně. Každý interface (nejenom COM interface) je vstupní branou do objektu umožňující (přes ni) zaslat zprávu objektu a tím si vyžádat jeho službu. Tedy úvahy o interfacu jako Private postrádají automaticky smysl, protože jsou to dveře (a dokonce jediné dveře) do objektu. Udělat je Private znamená tyto dveře zazdít. Interface je již sám svou povahou zpřístupněn pro okolí.

3. Zpětná reference - owner

3.1 Dotaz

Narazili jsme na následující problém ve VB: Máme objekt z třídy CA, který agreguje objekt CB. Objekt B má zpětnou referenci na objekt A (owner). Jestliže chci zrušit instanci objektu A použitím

```
Set A=Nothing
```

tak se tento objekt nezruší - neprovede se Terminate, protože objekt B má na něho referenci. Otázka zní, kde zrušit referenci na objekt A v objektu B, když to nelze provést v metodě Terminate objektu A?

3.2 Odpověď

V prvé řadě se domnívám, že se nejedná o problém Visual Basicu v tom smyslu,že by neuměl správně počítat reference a "nesprávně uvolňoval nadbytečné objekty". To Visual Basic umí velmi dobře. Problém je v tom, že se jaksi všeobecně podává doporučení ohledně uvolňování podřízených objektů ve znění: *Umístěte do události Terminate objektu uvolnění podřízených objektů*. A ono to nemusí fungovat vždy. Celý zádrhel je v tom, že událost Terminate se spustí skutečně až při uvolnění objektu a nikoliv při uvolnění nějaké reference, která zatím ještě nevede k uvolnění objektu. Tím při kruhové referenci (circular reference) objektů nastane to, že událost se nemůže spustit! Vysvětlení je prosté: Nadřízený objekt má ve své události Terminate uvolnění podřízeného objektu. Ale zavolání

```
Set Nadrizeny = Nothing
```

nezpůsobí vyvolání požadované události, protože existuje ještě zpětná reference od podřízeného objektu a počet referencí není nula. Pokud by tato reference od podřízeného objektu neexistovala, tak by se tímto příkazem `Set Nadrizeny = Nothing` nadřízený objekt uvolnil, událost se vyvolala a tím by se uvolnil i podřízený objekt. Tedy podřízený objekt svou referencí na nadřízený jej drží při životě a tím drží při životě sám sebe sám sebe, což je důsledek kruhové reference.

Hlavní rozdíl oproti ostatním klasickým jazykům je v tom, že VB nezná klasický destructor, ale oproti tomu umožňuje použít pro vložení kódu při uvolnění událost Terminate objektu. Ale jak se ukazuje, ono to není totéž!

Správné řešení spočívá v tom, že nadřízený objekt musí svým členům poslat *jinou* zprávu (tj. vyvolat metodu), ve které se nastaví tato reference uvnitř podřízeného objektu na ownera na Nothing. Je to časově zrcadlově opačný postup, než jak tato reference vznikla. Když konstrukce vznikala, vznikl owner, potom podřízený objekt a tomuto objektu se přiřadila reference na ownera. Nyní při destrukci konstrukce tuto větu musíte číst přesně obráceně. Nejprve se musí nastavit reference na ownera na Nothing, potom se uvolní podřízený objekt a potom se uvolní owner.

3.3 Dotaz 2 k témuž tématu

Tímto problémem „zpětné reference“ jsem se zabýval také a jako funkční se jeví řešení jdoucí úplně opačným směrem než je rozebráno v dotazu. Nehledá způsob "jak počítat reference", ale "jak získat referenci bez zvednutí počitadla odkazů". Když to nejde tak, půjdeme na to jinak. Sprostě (chytře) obejdeme počítání referencí. VB nám "ulehčuje" život a volá za nás "na pozadí" AddRef pro zvýšení RefCount (počitadla) referencí a Release pro jeho snížení. Získáme proto ukazatel pomocí funkce ObjPtr. Ten však nelze ihned použít ve VB pro práci s objektem, ale musíme ještě pro VB připravit použitelný odkaz.

Dosáhneme toho kombinací vlastností Get/Set v našem objektu Slave. Tyto budou sdílet interní proměnnou:

```
Private mpPointerToMaster As Long
```

Do ní uložíme ukazatel na Master ve vlastnosti:

```
Friend Property Set Master(ByVal ioMaster As CMaster)
    mpPointerToMaster = ObjPtr(ioMaster)
End Property
```

Nedokumentovaná funkce ObjPtr nám místo objektu vrátí pouze ukazatel na objekt. Důležité je, že pouhým získáním ukazatele nedojde ke zvýšení počitadla referencí.

Kdykoliv budeme chtít v objektu Slave volat Master, obrátíme se na vlastnost:

```
Friend Property Get Master() As CMaster
    Dim MasterI As CMaster
```

```
    ' Skopírujeme ukazatel a vytvoříme ilegální referenci (bez počítání
referencí)
    ' Nesmíme se pokusit kopírovat mimo oblast viditelnosti!
    Call CopyMemory(MasterI, mpPointerToMaster, 4)

    ' Vrátíme použitelnou referenci
    Set Master = MasterI

    ' Zničíme ilegální referenci nastavením na Null pointer
    ' Obdoba nastavení na Nothing
    Call CopyMemory(MasterI, 0&, 4)
End Property
```

CopyMemory nám skopíruje ukazatel na objekt do objektu a tím obejdeme

standardní schéma VB.

3.4 Odpověď

Musím vyslovit důrazné varování. Zní to sice elegantně obejít VB, ale pozor na počítání referencí v komponentní technologii. Bude Vaše konstrukce fungovat i přes rozhraní komponent a přes rozhraní procesů? Zkuste například udělat tuto konstrukci pomocí ActiveX EXE anebo přes rozhraní procesů do jiného threadu, jestli bude funkční. A navíc - nenastanou problémy s komponentami - s hlídáním "kdy je uvolnit"? Například u sdílené instance apod.?

Samozřejmě u kruhové reference nemá vůbec smysl o tomto problému uvažovat, protože oba objekty musí být v jedné komponentě. Ale co když někdo tuto konstrukci použije přes rozhraní komponent nebo přes rozhraní procesů, kdy jeden objekt je v jednom procesu a druhý v druhém? Bude funkční? Podle všeho nikoliv.

Můj názor k řešení: Osobně se domnívám, že řešení problému je opravdu v dohodě zániku asociací. Tato dohoda musí v projektu znít: V jakém kontextu asociace vznikla, v takovém musí zaniknout (ale v opačném garde). Pokud Faktura má Řádky a Řádek vidí svou fakturu jako MojeFaktura, tak při zrodu Řádku se naplní asociace

```
Set NewRadek.MojeFaktura =Me
```

a stejně tak, když se Faktura likviduje (resp. likviduje pouze jeden svůj Řádek) tak musí říci Řádku, aby si zrušil asociace. Nemůže k tomu použít událost Terminate, ale služební metodu, nazvěme ji například `DeleteAllAsoc` . Takovýchto metod je více - například vymazání z perzistence, tj,. odeslání příkazu do DB vrstvy se svým ID. Tedy nadřízený objekt nemůže použít událost Terminate, ale volá ve scénáři uvolnění nejprve metody (resp. metodu, která všechny dá dohromady) podřízeného objektu.

Osobně bych byl velmi opatrný ke konstrukcím, které nejsou standardní v dané technologii, zejména pak k těm, které obcházejí samotnou vrstvu prostředí a jdou příliš hluboko do implementace a tak ji obcházejí. Je to podobné, jako ve „standardním autě“ použít nestandardní součástku. Takováto řešení jsou náchylná k nestabilitám vůči dalším verzím prostředí apod.

4. Flexibilita datové vrstvy

4.1 Dotaz

... mám na Vás ještě jeden dotaz, jak to děláte Vy. Zřejmě by tento problém zajímal také více lidí.

Mám package (*pozn. autora: tazatel zřejmě pracuje Javě*)

bussines	
GUIaplikace1	
GUIaplikace2	(fakt je to nádhera oddělit bussines od GUI)
DBfilesystem	pro úklid objektu do souboru txt
DBCache	pro úklid do Cache
DBSybase	atd.

Dotaz zní:

1. Má existovat nějaká obecná Package DBobjekty, ktera bude mít implementaci jinou v každém z balíčku DB* a pak se podstrčí jen např. DBfiles.JAR.

nebo

2. DBobjekt balíček je obecný a podle nastaveni (parametr, konfigurace) se vytvoří instance DBobjektu z konkrétního package a pracuje se jen s rozhraním?

4.2 Odpověď

Oba přístupy jsou možné a záleží spíše případ od případu, hlavně obchodně a následně konfiguračně, který je výhodnější (hlavně obchodně výhodnější). Nic jiného není rozhodující. Ve většině případů se volí varianta konfigurace „natvrdo“, kterou si uživatel nemůže měnit, ale musí požádat dodavatele o změnu. Technicky jsou však možné obě varianty.

Teoreticky „objektově nejčistší“ se mi jeví varianta definovat zvlášť interface odpovídající funkcionalitě volání DBObjektu a vyměňovat za tímto interfacem implementace DBObjektu „natvrdo“ pomocí verzí a tedy konfigurací při dodávce uživateli. Jedinou výjimku by tvořil systém, kdy již přímo v zadání je požadavek spolupráce systému s heterogenními databázemi a požadavek na jejich výměnu za běhu programu. Jedná se vlastně o obchodní problém: Pokud dodáte flexibilní program s možnou výměnou databáze přímo uživatelem, měl by tuto funkcionalitu také ocenit, nejlépe vyšší cenou...

5. CACHÉ

5.1 Dotaz

Zajímal by mne Vás názor na Caché, a to jednak:

- do jaké míry naplňuje teorii objektově orientovaného programovaní
- srovnání databáze Caché s relační databází

5.2 Odpověď

CACHÉ databáze patří k perspektivním a velmi silným databázovým prostředkům. Jazyk této databáze je z hlediska OOP velmi čistý srovnatelný se SmallTalkem. Jedná se o objektové prostředí, které provádí automatické mapování objektového modelu tříd do stromové struktury dat. Vývojáři v rámci designu tedy vypouštějí náročný proces designu pro získání persistence objektů mapováním do relační databáze (jednoduše řečeno - objekty umějí persistenci „automaticky“).

Navíc tento prostředek podporuje připojení přes ODBC a také podporuje SQL syntaxi tam, kde je třeba a umí poskytnout ActiveX interface, takž klienta lze napsat třeba například ve VB. Podporuje také tvorbu WEB aplikace samostatnými prostředky.

Co se týče rychlosti, mám zprostředkované informace. Podle zkušeností z konzultací s firmami, které zavádějí CACHÉ přímo v praxi, a kde jsem zaváděl OOP a UML, jedná se podle zkoušek těchto pracovníků o velmi rychlou databázi, přibližně o dva řády rychlejší, než jsou relační databáze.

Rychlosti je docíleno jiným typem vazby mezi daty, než jaký používají relační databáze: V relační databázi je vazba tvořena shodou hodnot ve sloupci, tedy databázový stroj se sice optimalizovaně, ale přece jen neefektivně, "hrabe" v celé tabulce. V těchto firmách jsou vždy překvapeni "pomalostí" relačních databází. Ve stromové databázi, která je schována za prostředím objektů, je vazba vytvořena přímou vazbou, tj. každý prvek dat vidí pouze díky stromu pouze své prvky a nikoliv všechny.

Příklad:

"Zaměstnanec má N dětí". V relační databázi je tabulka dětí a každé dítě má cizí klíč zaměstnance a teprve shodou této hodnoty (což provádí databázový stroj) se získají pro konkrétního zaměstnance jeho děti. Ve stromové databázi zaměstnanec jako uzel stromu má rovnou ve stromu "ty svoje děti" a při dotazu na jeho děti se jde rovnou do jeho dětí. Většinou na CACHÉ přecházejí ty firmy, které používaly M-databázi (strukturální předchůdce CACHÉ).

Pro ty, kteří se zatím nesetkali s touto databází s objektovým prostředím zde uvádím odkaz:

www.intersystems.cz

kde si můžete také stáhnout zdarma databázi pro jednoho uživatele.

6. Hierarchické struktury objektů

6.1 Dotaz

... rád bych s Vámi konzultoval jeden problém, který řeším. Jde o hierarchické struktury. Problém si můžete představit tak, že máme souborový systém,ve kterém jsou složky a soubory. Moje první úvaha byla taková, že složka obsahuje buď další složky, nebo soubory. Proto jsem založil dvě třídy:

```
class TAdresar {
    private colAdresare as collection
    private colSoubory as collection
} + protokol zpráv a metody add, rem, count, clear

class TSoubor {
    private sNazevSouboru as string
```

```
}
```

Ručně(plnit jednotlivé objekty například v události stisknutí tlačítka) bych opravdu byl schopen, a také jsem to zkoušel, daný systém vytvořit. Takže jsem si vytvořil instanci třídy TAdresar a do ní jsem vložil další adresáře a soubory. Ale jak si asi dokážete představit, tak je těžké plnit tuto strukturu pomocí smyček for..next, když není předem jasné, kolik úrovní bude daná hierarchie mít, natož tuto strukturu procházet a tvořit z ní strom, například v TreeView.

Rozhodl jsem se tedy, že vazbu mezi TAdresar a TSoubor vyřeším pomocí asociativní třídy. To mi navíc dalo možnost definovat do některého adresáře odkaz na daný soubor. Tím se ale pořád neřeší problém adresářové struktury. Proto jsem založil ještě další asociativní třídu TVazbaAdresarAdresar, která mi definovala vazby mezi jednotlivými adresáři. A abych se konečně dostal ke konečnému řešení:

Vytvořil jsem třídu adresář(TAdresar), která obsahuje proměnnou + protokol zpráv o názvu adresáře.

```
class TAdresar {
    private sNazevAdresare as string
}
```

A kolekci těchto adresářů. Realizoval jsem vše v prostředí visual basic, i když příklady spíše odpovídají Javě.

```
class TAdresare inherit Collection {
}
```

Dále jsem vytvořil třídu TSoubor, a jeho kolekci, která jej opět umí omezit na konkrétní adresář.

```
class TSoubor {
    private sNazevSouboru as string
}
class TSoubory inherit Collection {
}
```

A pak vazbu adresář-adresář (TVazbaAdresarAdresar) a soubor-andresář(TVazbaAdresarSoubor) a jejich kolekce TVazbyAdresarAdresar, TVazbyAdresarSoubor,

```
class TVazbaAdresarAdresar{
    private oNadrizevyAdresar as TAdresar
    private oPodrizenyAdresar TAdresar
}

class TVazbyAdresarAdresar inherit Collection{
}

class TVazbaAdresarSoubor{
```

```
    private oAdresar as TAdresar
    private oSoubor as TSoubor
}

class TVazbyAdresarSoubor inherit Collection{
}
```

Pokud se nepletu, vazba TVazbaAdresarSoubor je asociativní třídou a TVazbaAdresarAdresar je také asociativní třídou. Jestliže jde opravdu o asociativní třídu, tak je otázkou, jak zakreslit v Class Modelu asociativní třídu TAdresarAdresar.

Výhoda druhého řešení je také ta, že jsem schopen vytvořit i zástupce pro daný file v jiném adresáři a také link mezi adresáři ve stylu UNIXu. Jde mi tedy o to, jestli jsem správně pochopil použití asociativní třídy a jestli je toto řešení opravdu objektově čisté.

6.2 Odpověď

Nejprve co se týče "objektové čistoty", na kterou se ptáte. Z tohoto hlediska je vaše řešení tzv. "čisté", protože zavádíte objekty a třídy "technologicky správně", ale bohužel samo řešení není příliš šťastné, jeví se mi jako zbytečně složité.

Přístup k řešení je velmi podobný jako v design pattern Composite a zní jednoduše:

V této struktuře existují Uzly - Nodes hierarchie, které mají tu schopnost, že lze po nich "chodit ve stromu hierarchie" nahoru dolů. Prvek Node má tedy tyto vlastnosti:

```
'-------------------
'Class CNode
'-------------------

Private mNazev As String
Private mcolNodes As CColNodes
Private asocMyParentNode As CNode
```

Pokud je to root, potom asocMyParent je Nothing.

Dále existují dva dědicové:

```
CFile (inherited from CNode)
CDirectory (inherited from CNode)
```

Můžeme přidat ještě jeden atribut na úroveň CNode, která rozlišuje typ:

```
'-------------------
'Class CNode
'-------------------
```

```
Private mNazev As String
Private mcolNodes As CColNodes
Private asocMyParentNode As CNode
Private mTyp As Integer '0 = File, 1 Directory
```

anebo tuto informaci číst nějak pomocí classname(záleží na syntaxi OOP jazyka).

Navigace je snadná: Průchod dolů se děje v kolekci CColNodes (kolekce vracející item typu CNode). Pro průchod stromem je možné zavést rekurzivní mechanismus. Průchod nahoru se děje přes asocMyParents objekt - asociovaného vlastníka kolekce.Nedoporučuji provádět konstrukci Node jako dědice kolekce (jak jsem pochopil ve vašem návrhu). Myslím, že věta: "Node obsahuje kolekci svých Node" je názornější a srozumitelnější, než že "Node je kolekcí".

CDirectory umí přidat do kolekce nový prvek, CFile má kolekci vždy prázdnou

(CColNodes.Count je nula pro CFile prvek).

Poznámka: Existuje ještě více perfekcionalistická konstrukce, že kolekce colNodes je až v dědici CDirectory, když ho vlastně CFile nepotřebuje, ale to je již drobnost.

Vaše varianta a asocitavní třídou je také po malé úpravě možná:

Zavedete asociaci na úrovni mezi CDirectory a CNode takto:

```
Left Association End - CDirectory (může být jen Directory)
Right Association End - CNode (může být jak File, tak Directory)
```

Tento vztah by měl význam, pokud dané adresáře a fily nemají čistě kompositní charakter.

V kódu by to vypadalo nějak takto:

```
'--------------------
'class CAsocDirHasNode
'--------------------

Private mDirectory as CDirectory 'left
Private mNode as CNode 'right
```

a musí existovat kolekce z těchto itemů, která se musí umět zúžit pro levý nebo pravý prvek. Jinak rozdíl obou přístupů ukáže názorně namapování do relační DB: V prvém případě bude tabulka CNode obsahovat (kromě atributů): - svůj klíč , - cizí klíč parentNode, tj, (ID, Parent_ID, ...)

v druhém bude existovat asociativní tabulka, obsahující - svůj klíč, - cizí klíč z Node, - druhý cizí klíč z Node, tj. (ID, Dir_ID, Node_ID, ...)

7. N-vrstvová architektura a nezávislost komponenty na kontextu použití

7.1 Dotaz

Následující mail, který jsem nedávno obdržel, mne inspiroval k napsání tohoto článku:

```
... V soucasne dobe vytvarim diky Vam svuj prvni jednoduchy program (na
evidenci mereni pacientu) a tvorim ho zatim jako monolit - standard.exe
- spise si zkousim zavadeni trid podle nalezenych entit z analyzy,
jejich kodovani, vytvareni seznamu objektu (agregacnich i asociacnich),
jejich plneni z databaze, mapovani objektu atd. Uz ted ale tusim, ze
bych mel trochu uvazovat o komponentním reseni. Precetl jsem si clanek o
chybe v knize o OOP, ale zatim jsem stejne vsechny podobne pripady resil
podle teto knihy (u monolitu to nevadi). Samozrejme tusim, ze takova
aplikace je potom bez opravy teto chyby nepouzitelna pro prechod na
komponentni technologii, ale zatim i po precteni knihy o COM technologii
moc netusim jak toto resit. Doufam, ze taky naleznu nejake dalsi
informace ve skriptech "COM a DCOM v praxi".
Jak jsem predeslal jsem uplny zacatecnik v pouzivani COM technologie a
pri studiu jsem narazil uz hned na zacatku na zasadni nejasnosti, bez
jejichz vysvetleni se snad ani neda pokracovat. Tech dotazu by bylo vic,
ale samozrejme si uvedomuji, ze Vas nemohu takto obtezovat (uz tak je
ten e-mail hrozne dlouhy). Presto by mi velice pomohlo, kdybyste mi dva
dotazy zodpovedel (nebo treba reagoval nejakym clankem na webu, ikdyz
vim, ze se tam spise zabyvate problemy OOP nez COM - mimochodem dekuji
za upozorneni na novy clanek; jsem rad za kazdy novy clanek, ktery tam
objevim; uz se moc tesim na Vasi novou knihu o UML).

1. Prvni problem souvisi s tou chybou v knize o OOP. Kdyz zadny objekt
business logiky nesmi volat formulare a msgboxy, jak jsou pak predavany
informace ze serveru na klienta. Opravdu je to vzdycky tak, jak to
```

popisujete v knize o COM, ze server po prijeti zpravy od klienta vyvola udalost a tuto udalost zachyti "obalujici" klient? Opravdu to znamena, ze kdyz chci programovat pro COM, tak budu psat jednu udalost za druhou?

2. Druha moje nejvetsi nejasnost spociva v nepochopeni znovupouzitelnosti komponenty business logiky. Chapu, ze komponenty ActiveX Control mohu pouzit i v jinych podobnych aplikacich. Ale to jsou prvky GUI. Vy ale naopak zduraznujete, ze nejvetsi vyhoda je v re-use komponent obsahujicich aplikacni logiku, a ze klienty je stejne treba vzdy vytvorit znovu. Uvadite, ze pri rozhodovani o komponentach, je treba porad myslet na to, zda pujde komponentu pouzit i v jinych aplikacich. Ale jak muzu v uplne jine aplikaci pouzit komponentu obsahujici entity "Majitel", "Vozidlo" atd., kdyz tato aplikace resi jiny problem? Snad by se daly pouzit malinke komponenty typu kazdy objekt(trida) zvlast, tedy komponenta "Majitel", komponenta "Vozidlo" atd.

Samozrejme Vam nebudu zazlivat, kdyz kvuli nedostatku casu neodpovite ani na tyto dva dotazy. Uz tak jste mi poskytl ve svych dvou knihach, ve skriptech a clancich na webu zaplavu novych informaci a znalosti.
Dekuji. Me podekovani take smeruje panu Radovanu Novakovi za napsani skript o technologii ADO...
R.S.

Rozhodl jsem se na tento mail odpovědět článkem, protože odpověď a může být užitečná i dalším čtenářům. Obě otázky zdánlivě spolu nesouvisí, ale jak si ukážeme, mají velmi úzký vztah.

Odpověď 1

Začneme první otázkou. Opravdu žádná komponenta business logiky nesmí volat GUI prvky. To je dobré brát jako poučku, tedy rada zní: Jakmile se o něco podobného pokusíme, pak “něco” v nás zablikne a řekne “ne”. Ale poučka sama o sobě nestačí, pokud nechápeme její smysl, tedy odpovězme si - proč tomu tak je?

Teprve technologie distribuovaných objektů (tj. případ, kdy je klient od objektu vzdálen na jiném stroji resp. v jiném procesu) umožnila zavést takzvanou n - vrstvovou architekturu.

Zavedení distribuovaných objektů a následně n-vrstvové architektury je v rámci MS technologií nyní (k datu vydání této e-knihy) možné těmito dvěma základními způsoby:

- v případě klasické technologie COM komponent:
    1. pomocí „klasického“ DCOMu
    2. pomocí MTS resp. COM+
    3. nepřímo pomocí technologie ASP stránek, za nimiž je COM objekt přilinkován
- v případě .NET technologie pomocí využití assembly na vzdáleném stroji
    1. voláním Web Services
    2. přes ASP.NET

Zavedení n-vrstvové architektury není samoúčelné, tj. n -vrstvová architektura není zavedena proto, aby zavedena byla, ale proto, že je výhodné ji zavádět. Uveďme si v čem tato výhoda spočívá.

Představme následující příklad: Řešíme část systému pracující s osobami. Zavedeme proto dvě třídy pro dva typy objektů, třídu `COsoba` pro samotnou osobu a třídu `CSeznamOsob` pro seznam osob. V těchto třídách nechť se nevolá žádný prvek GUI. Obě třídy umístíme do komponenty (COM anebo class assembly), která je součástí komponent business logiky, zvolme bud ActiveX DLL nebo v .NET class library.

Odpovězme si na první otázku, “jak vlastně volá business logika prvek GUI”. Odpověď zní jednoduše - nijak. Naopak, prvky GUI volají naši komponentu a dostávají odpovědi (tj. co se odehraje v business logice) jako výsledky tohoto volání. Na základě tohoto výsledku prvek GUI zavolá jiný prvek GUI.

Například nechť námi zvolený seznam osob má zavedenu metodu `LoadFromDB`, která způsobí natažení seznamu z databáze do živých objektů seznamu. Jedná se o metodu business vrstvy, která požádá datovou vrstvu o data a naplní jimi objekty seznamu. Pokud tato operace selže (například spadne konekt do DB), potom je chybou v návrhu, aby **uvnitř** metody `LoadFromDB` v jeho error větvi existovalo přímo volání chybového `MsgBox`. Správným řešením je, aby buď `LoadFromDB` vrátilo error hodnotu jako výsledek, anebo se nastavil nějaký atribut objektu (vynesené ven jako property), které indikuje chybu. Takže nikoliv takto:

Třída `CSeznamOsob` , její metoda `LoadFromDB:`

```
...
On error goto ErrTrap
      ... výkonná část metody
      Exit Sub
ErrTrap:
      MsgBox “Error...
End Sub
```

ale (například) takto:

```
On error goto ErrTrap
      ... výkonná část metody
      LoadFromDB = cdbOK
      Exit Function
ErrTrap:
      If Err.Number = .... Then
            LoadFromDB = cdbConnectFailure ‘vlastní konstanta apod.
      End If
End Sub
```

Na straně klienta objektu potom může vypadat kód takto

```
RetValue = MySeznam.LoadFromDB
Select Case RetValue
...
Case cdbCOnnectionFailure
      MsgBox “
End Select
```

Důležité je dodržet princip - komponenta business logiky je pouze “služebníkem” (serverem) pro své okolí, v tomto případě je služebníkem GUI prvku. Jinými slovy okolí - v tomto případě například GUI, volá funkcionalitu této komponenty.

Všimněme si, že pokud použijeme tento postup, potom dodržíme jednu důležitou zásadu: Komponenta sama o sobě je nezávislá na kontextu, kde a v jakém prostředí bude použita, tj. zda se jedná o formulář Windows aplikace, nebo ASP formulář, anebo dokonce když se jedná o součást jiné komponenty. Takže takto navrženou komponentu napíšeme pouze jednou a můžeme ji použít v různých situacích (aniž bychom ji znovu kompilovali), které jsme popsali.

Co se stane, pokud dovolíme, aby komponenta zavolala uvnitř sebe GUI prvek? Pokud zůstaneme na úrovni jednoho stroje, potom nemusíme narazit na problém. Ale jakmile začneme pracovat s distribuovanými objekty (viz předešlý případ), nastane problém. U ASP stránek je nasnadě, že objevení se MsgBoxu na serveru vzdáleném stovky kilometrů nemá smysl. Podobně pokud pro tvorbu instance na lokální síti použijeme vzdálený objekt, tak MsgBox na straně serveru je opět nesmysl, kdo by běhal do jiné místnosti dívat se na nějaké hlášení. Takže volání GUI prvku by výrazně omezilo možnost použití naší komponenty a to pouze na desktop aplikaci (tj. přesněji na jeden stroj). Takovéto omezení je pochopitelně v rozporu s požadavky na rozsáhlé síťové systémy anebo na aplikace typu Internet - Intranet, což je v rozporu s pojetím n-vrstvové architektury.

Za zmínku stojí ještě problém vložení komponenty do komponenty. Tam totiž výrazně vyniká analytický nesmysl volání GUI prvku: Jak má vnitřní komponenta oznámit vnější komponentě, co se odehrálo? Má toto oznámení nést v sobě jako MsgBox? Proč, když není nikdo, kdo to bude číst, a

bůhví, kdo a kdy to bude číst. Vnější komponenta musí umět sama zareagovat na výsledek služby vnitřní komponenty a ta ji musí předat výsledek, jinak vnitřní komponenta dopředu předjímá, co se má dělat.

Odpověď 2

Druhá otázka s první otázkou úzce souvisí. Jedná se o opětovnou použitelnost komponenty business logiky. Samozřejmě, pokud budete vyvíjet komponenty jako jedna osoba s jednou vyvíjenou aplikací, tak nebudete spatřovat žádný re-use, protože všechno je použito pouze jednou, tj. vše co vyvinete je na jedno použití (ale jste si jist, že tomu tak bude za půl roku?).

Představme si však větší firmu, například řešící IS s řádově několika projekty a několika programátory. Všechna řešení firmy se rozpadají do obchodních projektů a v rámci projektů se používají oblasti (moduly, komponenty apod.), v rámci nich třídy, ale ty se mohou opakovat. A nyní je otázkou, jak použít re-use? Uveďme příklad ze systému Doprava, Náklady - v modulu nazvaném Mzdy se pracuje s Osobami a také v modulu Řidiči se používají Osoby. Nebudou se Osoby přece vyvíjet dvakrát! V obou je třeba použít komponentu a tou je komponenta Osob. Podotkněme, že UML povoluje použít jak tzv. source component, tj. knihovnu zdrojů, tak binary component, tj. komponentu v pojetí binárního útvaru. Re-use znamená, že komponenta je sdílená oběma moduly (komponentami).

Omyl

Jako poslední bych rád poopravil jeden omyl čitelný z mailu a vzniklý asi nepochopením a nesprávnou formulací v knize o COM. Je možné v technologii DCOM na lokální síti poslat událost ze serveru na klienta, ale není to v žádném případě znakem třívrstvové architektury!

Tomuto způsobu oznámení klientovi, že se něco v komponentně odehrálo, je třeba věnovat zvýšenou pozornost, protože nemusí vždy vést k žádoucímu cíli, zejména u velkého počtu klientů resp. u těch operací, které vyžadují delší čas. Navíc u Internetové technologie není toto řešení schůdné a je použitelné pouze u lokální sítě.

Při použití DCOMu a vyvolání události komponentou na serveru a odchycením klientem se totiž (ve VB 6.0) sekvenčně a za sebou (!) vyvolají všechny odpovídající metody klientů a server čeká až do chvíle, než operaci skončí poslední klient (pokud použijete ActiveX EXE komponentu). Pokud například použijete tento způsob u vyvolání Refresh GUI prvku podle scénáře: jeden klient změní nějaký seznam, tím se vyvolá událost, ostatní klienti to odchytí a provedou refresh obrazovky pro nové zobrazení seznamu, tak tento N-násobný refresh při tomto řešení bude probíhat nikoliv vedle sebe, ale za sebou (pokud aplikaci píšete ve VB 6.0 a nikoliv multithreadově). První klient se refreshne, druhý se refreshne atd. Po celou tuto dobu bude server odmlčený (vyplývá z povahy sdílené komponetny ActiveX EXE), což může vést k chybám a dlouhým prodlevám. Samozřejmě problém je v jednom threadu zpracování na serveru, který musí obsloužit všechny klienty (doslova obvolat je všechny).

Podobné krizové situace se řeší jednoduše pomocí Message Queue Serveru (viz skripta o třívrstvové architektuře). Jeden klient způsobí uložení pouze Message do MSQ Serveru a ostatní si je vyberou a paralelně zpracují.

Podotkněme, že tento nástroj je v rámci .NET již ve vašem systému zabudován.

Závěr

V článku je poukázáno na vlastnost n-vrstvové architektury vyplývající z povahy komponentní technologie - nezávislost komponenty na kontextu použití. Nedodržení této zásady vede k narušení principů n- vrstvové architektury a problémům u síťových systémů. V článku je také poukázáno na možnosti re-use komponent v systému a upozornění na možnost chyby možného způsobu vyvolání události na serveru a odchycení u klientů komponent.

8. Interface, třída, objekty a příkazy ve VB 6.0

8.1 Dotaz

V tomto článku budu reagovat na časté dotazy, ohledně násobných interfaců, komponent, tříd atd. ve speciálně VB 6.0

8.2 Odpověď

Nejprve je třeba zdůraznit, že sama syntaxe Visual Basicu 6.0 je postavena tak, aby použití komponentní technologie nečinilo syntaktické rozdíly oproti jeho původním částem - tedy jinak řečeno, zabudování objektů z "nové" komponenty podléhá ve Visual Basicu stejným syntaktickým pravidlům, jako zabudování objektů z již existujících tříd. Visual Basic tedy svou syntaxí přímo odpovídá pravidlům komponentní technologie (přesněji řečeno speciálně technologii ActiveX) a to činí Visual Basic 6.0 velmi přívětivým pro použití komponentní technologie ve své jednodušší podobě ActiveX.

Tato vlastnost Visual Basicu 6.0 činí tvorbu komponent velmi jednoduchou (například tvorba komponenty s určitými vlastnostmi odpovídá "klikání" ve výběru typu projektu, nastavení podmínek kompilace atd.) a použití instance komponenty odpovídá některým z "obvyklých" příkazů Visual Basicu. Proto je třeba znát určité základy komponentní technologie, jinak nám z této činnosti zůstane pouze klikání samo o sobě.

Z hlediska sémantického je dobré rozlišovat dva pojmy

- Komponenta
- Instance komponenty

Pod komponentou budeme mít na mysli nainstalován nějaký SW balík, tj. nainstalovány binární soubory na svém resp. vzdáleném stroji, a tato instalace nám teprve poté umožní vytvořit instanci komponenty, tj. vytvořit a spustit již existující objekty z této komponenty.

Je vidět, že vztah mezi komponentou a instancí komponenty je podobný, jako vztah mezi třídou a objektem, kde třída je kopytem pro objekty. Nainstalovat si komponentu ještě nemusí znamenat, že na našem stroji "žije" některá z jeho instancí, znamená pouze možnost takovouto instanci nechat vytvořit.

Z hlediska Visual Basicu vytvořit komponentu znamená na začátku projektu zvolit ActiveX komponentu (nejčastěji použitelné typy DLL, EXE, OCX), naprogramovat tento projekt a poté z něj vytvořit instalační balík. Nainstalovat tuto komponentu znamená vzít tento instalační balík a nainstalovat si tento SW produkt buď u sebe anebo zpřístupnit si tvorbu na síti v celém systému (například pomocí MTS).

Vytvořit instanci je vlastně nějaké zavolání této komponenty a vytvoření instance s určitými vlastnostmi. K tomu slouží pojem třída.

8.2.1. Komponenta, třída a instance komponenty

Nyní je otázkou, jaký vlastně vztah mezi komponentou a třídou?

Každá komponenta může obsahovat ***několik tříd***. Ve Visual Basicu tomu odpovídá zavedení několika class modulů projektu komponenty. Pomocí těchto tříd vzniknou instance.

Z hlediska tvorby instancí komponent je důležité od sebe odlišit dva základní typy komponent (viz ostatní články tohoto dokumentu) - zda se jedná o komponentu sdílenou mezi klienty anebo s izolovanými klienty. Nejprve vyřešme případ komponenty s izolovanými klienty. Ve VB se jedná doporučený typ komponenty a odpovídá jí typ ActiveX DLL.

Vytvoříme tedy komponentu ActiveX DLL a v ní několik tříd - class modulů. Po kompilaci je každé z těchto tříd přiřazen na světě jednoznačný identifikátor CLSID (class identifier). Pro představu nechť jsme vytvořili nový projekt AcitveX DLL a v něm zavedli tři class moduly (multinstance public creatable) `CA, CB` a `CC`. Vznikly tři nové CLSID "obsažené" v této komponentě a spolu s nimi se přiřazují tři názvy tříd.

Při instalaci komponenty na jiném stroji si tato komponenta "s sebou" přinese tato CLSID a zapíše si je na stroji do registry. Současně je k těmto CLSID přiřazen i jejich uživatelský název, tj. "čitelné" názvy tříd `CA, CB, CC.` Po instalaci tedy existuje v našem systému převodník "název třídy" a jeho CLSID

Po přilinkování projektu do VB máme poté možnost vytvořit novou instanci komponenty, a to v našem příkladu dokonce tři možnosti - buď z třídy `CA`, nebo z `CB` nebo `CC`.

Příkaz pro vytvoření nové instance komponenty je ve Visual Basicu pravá strana příkazu `SET ... NEW` a za ním požadovaná třída, tj. pravá část příkazu:

```
Set A = New CA
```

znamená pro komponentu následující: Nejprve se pomocí převodníku nalezne pomocí názvu třídy (v tomto případě `CA`) odpovídající CLSID a toto CLSID se stává vstupním parametrem pro tvorbu nové instance komponenty. Komponenta vytvoří novou instanci z této třídy. Pokud zvolíme jiný název třídy, bude tomu odpovídat jiné CLSID a tedy požadavek na instanci komponenty jiných vlastností.

Máme tedy v našem případě možnost vytvořit tři různé objekty (instance komponenty) ze tří různých tříd od téže komponenty. Neznamená to však násobnost interfacu! Každý z těchto objektů je jiný a má vždy jeden interface, jak si ukážeme v následující kapitole.

8.2.2. COM Interface

V předešlém příkladu jsme vytvořili v projektu ActiveX DLL tři class moduly a tedy pomocí této komponenty máme možnost vytvořit tři různé instance ze tří tříd. To však neznamená, že máme k dispozici trojnásobný interface. Každý ze vzniklých objektů má pouze jeden interface, a trojnásobná možnost tvorby pouze znamená, že každý objekt z jiné třídy má jiné vlastnosti.

Spolupráce objektů v COM se děje pomocí tzv. COM interfacu (viz kniha o COM). Interface je obecně v OOP chápán jako jedna část protokolu zpráv objektu - ona vnější množina zpráv, které objekt může přijmout a na základě nich spustit metodu.

Připomeňme, že protokol zpráv patří k základním pojmům OOP. Nejprimárnější vlastností objektů je možnost přijmout zprávu a na základě této zprávy spouštět vnitřní metody. Tento převodník se nazývá protokol zpráv a množina zpráv, kterou může objekt přijmout, se nazývá interface objektu.

V COM se tento pojem interface dále specifikuje na binární technologii takto:

- Jedná se o COM interface v binární podobě (reprezentován v binární podobě jako řada adres-ukazatelů na metody)
- Každý COM interface obsahuje první tři zprávy vymezené na služební účely
- K objektu lze přiřadit několik COM interfaců

Zdůrazníme, že v poslední vlastnosti se hovoří o objektech, tedy instancích komponent nikoliv o komponentách. Podrobněji jsou vlastnosti COM interfacu popsány ve zmíněné knize, pouze stručně se zmíníme, že ony tři služební metody jsou *QueryInterface, Add a Release*. První slouží k "výměně" interfacu, druhé dvě obsluhují počítání referencí na instanci komponenty.

Důležité pro další pochopení je, že Visual Basic při zavedení class modulu v našem projektu komponenty automaticky s class modulem zavede nejenom novou třídu (CLSID), ale i nový interface odpovídající svými zprávami public metodám deklarovaným této třídě. Každý interface získá také svůj identifikátor, tzv. IID (interface identifier) a také existuje převodník mezi názvem interfacu a číslem IID.

Podobně jako existuje vztah mezi komponentou a instancí komponenty, jako mezi třídou a objektem, je vhodné rozlišovat mezi interfacem a instancí interfacu. Rozlišujeme tedy pojmy:

- interface
- instance interfacu

podobně jako ve vztahu třída-objekt, komponenta -instance komponenty. V konkrétní instanci komponenty existují konkrétní instance interfaců. V komponentě existují třídy a interfacy jako "kopyto pro možné budoucí objekty a jejich instance interfaců".

V našem případě v projektu se třemi class moduly tedy po kompilaci vzniknou tři třídy se svými CLSID a tři interfacy se svými IID, přičemž (a to je tak trochu matoucí) názvy tříd a interfaců jsou shodné (uschovaná čísla CLSID a IID však shodná nejsou). V našem případě tedy v komponentě existují "schovány" tři třídy `CA, CB, CC` a tři interfacy `CA, CB, CC,` přičemž platí, že pokud necháme zrodit instanci z třídy `CA`, může mít pouze jednu instanci interfacu a to z interfacu `CA` , podobně pro `CB` a podobně pro `CC`.

Pokud deklarujeme ve Visual Basicu proměnnou `Dim` resp. atribut třídy `Private (Public) As` název class modulu, potom tento název reprezentuje interface a nikoliv třídu!

```
Dim A as CA
```

znamená deklaraci proměnné `A` typu interface `CA`. Za touto deklarací je tedy schováno IID a nikoliv CLSID. Protože zpřístupnění konkrétní instance interfacu z instance komponenty se děje přes ukazatele, můžeme si tuto proměnnou představit jako proměnnou typu ukazatel na interface. Za deklarací je uschován převodník mezi IID interfacem a názvem interfacu CA. Znamená to, že proměnná `A` je typu interface `CA`, tedy jinak řečeno je to proměnná typu IID.

Příkaz `Set` znamená dosazení do proměnné typu ukazatel na interface konkrétní hodnoty ukazatele na interface (tedy instanci interfacu reprezentuje ukazatel).

Vrátíme se zpět k dvojici příkazů:

```
Dim A as CA

Set A = New CA
```

Můžeme nyní doplnit celý jeho význam z hlediska komponentní technologie. Již víme, že pravá strana příkazu `Set A = New CA` znamená "komponento, nechej zrodit jednu instanci komponenty z třídy `CA`". Když tato komponenta (jako jakési pozadí) nechá za pomocí COM a operačního systému zrodit novou instanci z této třídy, tak při této činnosti zjistí, že instance má vlastnit jednu instanci interfacu typu interfacu `CA`. Proto tuto instanci interfacu vytvoří (je jenom na komponentě, kde skutečně bude alokována a kde vznikne tato instance interfacu) a drží si ukazatel na ni. Levá část příkazu není nic jiného, než oslovení komponenty: Máš-li instanci interfacu typu `CA`, přesněji máš-li instanci interfacu typu IID (IID odpovídající typu interfacu `CA` - dotaz totiž vznikl na základě převodníku název interface IID), tak mi dej tuto instanci - ukazatel na ni a dosadím si ji do `A`.

Pokud bychom se v tomto případě dotázali na jiný interface, tak komponenta odpoví error, například

```
Dim A as CB

Set B = New CA
```

znamená: Nechej zrodit komponentu z třídy `CA` a poté se zeptáme - máš-li instanci interfacu typu `CB`, dej mi na něj ukazatel a dosadím si jej do `B`. V tomto případě samozřejmě program havaruje (komponenta nemá tento typ interfacu a proto nemůže poskytnout ukazatel), což se projeví ve VB již při psaní ve vývojovém prostředí (prostředí provádí tuto kontrolu "dopředu").

Poznámka: Pokud použijeme `CreateObject` namísto `New`, tak k dotazu na získání ukazatele na instanci interfacu dochází skutečně až při běhu programu, což se označuje jako tzv. late binding - k

vazbě na ukazatel na instanci interfacu nedochází "dopředu", ale až v okamžiku interpretace příkazu, tedy pokud se ptáme špatně, chyba se projeví až při běhu programu

8.2.3. Násobný interface

Komponentní technologie umožňuje přiřadit k jedné třídě vícero interfaců, tj. instance komponenty z dané třídy může mít dvě a více instancí interfaců. Ve Visual Basicu se tato vlastnost zavádí pomocí příkazu `Implements` a označuje se jako implementace interfacu do již existující třídy.

Platí jednoduché pravidlo, že dokud tento příkaz ve VB nepoužijeme, nemá smysl hovořit o násobném interfacu.

Princip tvorby násobného interfacu je velmi jednoduchý. Nechť v systému již existuje interface `CX`. Například jsme si do našeho projektu přilinkovali komponentu, která obsahovala class modul `CX` a díky tomu se v systému objevil interface `CX`.

Zaveďme class modul `CA` a do něj zaveďme příkaz Implements `CX`:

```
Implements CX
```

Pokud toto provedeme, potom jsme k našemu budoucímu objektu přidali nový interface, tedy množinu zpráv (jedna stran protokolu zpráv objektu) a musíme vyplnit druhou stranu, tj. jaké metody budou spouštěny na danou množinu zpráv. Tato množina metod dodržuje ve VB podtržítkovou syntaxi a musíme všechny tyto metody vyplnit.

Od této chvíle pokud necháme zrodit instanci komponenty ze třídy `CA`, bude mít tato instance komponenty dvě instance interfaců, jeden "implicitní" typu `CA` a druhý typu `CX`.

Dvojice řádků kódu

```
Dim A as CA

Set A = New CA
```

má nyní již hlubší význam: Opět znamená nechej zrodit objekt ze třídy `CA`, ale tento objekt má dvě instance interfaců díky zavedenému implements příkazu. Proto pokud se zeptáme přes příkaz Set, ptáme se na jeden z nich a tím je ten, který je typu `CA` (v tomto případě).

Ještě více vynikne tato situace na tomto postupu, kdy se neptáme na interface stejného názvu:

```
Dim X As CX

Set X = New CA
```

Tato dvojice příkazů znamená: Nechej zrodit instanci komponenty ze třídy `CA` (tato instance má dvě instance interfaců, jeden typu `CA` a druhý typu `CX`) a druhá část příkazu znamená - pokud máš ukazatel na instanci interfacu typu `CX`, dej mi jej. Instance komponenty jej má a vrátí jej. Vše tedy proběhne v pořádku.

Přiřazování `Set` se nepoužívá pouze při zrodu instancí komponent spolu s příkazem `New`, ale také mezi dvěma proměnnými typu interface. Význam je však úplně stejný: `Set` znamená oslovení instance "máš-li instanci interfacu daného typu, dej mi ji (ukazatel)", přičemž interface (tj. typ) je dán typem proměnné na levé straně příkazu:

Ukažme si to na spojení obou příkladů dohromady:

Do class modulu `CA` byl implementován interface `CX`. Zaveďme proměnnou typu interface a nechejme zrodit instanci komponenty

```
Dim A as CA

Set A = New CA
```

Poté zaveďme druhou proměnnou typu interface:

```
Dim X As CX
```

A proveďme přiřazení

```
Set X = A
```

Čteno zprava doleva, první část příkazu je dotaz do instance komponenty přes proměnnou A. Proměnná A je instance interfacu přiřazená k instanci komponenty přes příkaz New z třídy CA, tedy instance komponenty je z třídy CA. Přes tuto proměnnou A se ptáme instance komponenty: "máš-li instanci interfacu typu CX (X je typu CX), dej mi jej jako ukazatel a já si je dosadím do X". Protože instance komponenty podporuje tento interface, vrátí na něj ukazatel.

Z uvedeného je zřejmé, že obě instance interfaců patří k témuž objektu , o čemž se můžeme přesvědčit pomocí operátoru Is.

Dokončíme příklad v tom smyslu, že se dotážeme čtenářů - malý kvíz:

Do class modulu CA byl implementován interface CX. Prohlédněte si tyto tři posloupnosti kódu:

```
1.

Dim A as CA

Dim X As CX

Set X = New CX

Set A = X
```

```
2.

Dim A as CA

Dim X As CX

Set X = New CA

Set A = X
```

```
3.

Dim A as CA

Dim X As CX

Set A = New CA

Set X = A
```

Otázky zní:

1. Dvě z těchto posloupností jsou správně, jedna špatně. Která je chybná a kde bude havarovat?

2. Jsou dvě z těchto posloupností kódu, které jsou správné, plně zaměnitelné nebo nikoliv?

Pokud jste dobře četli, nebudou pro vás tyto otázky žádnou hádankou.

9. O jedné chybě v knize Základy objektově orientovaného programování

9.1 Úvod

Po napsání knihy ***Základy objektově orientovaného programování*** jsem podle ohlasu s potěšením zjistil, že se tato kniha stala spolu s její následovnicí ***Základy komponentní technologie COM*** jednou z velmi oblíbených knih mezi čtenáři a příznivci objektového programování a komponentní technologie.

Jako autor se však musím vrátit k jedné pasáži v této knize. Když jsem tuto knihu v roce 1998 psal, neměl jsem v té době ještě takové zkušenosti se zaváděním třívrstvové architektury a s komponentní technologií jako připsaní druhé knihy a jako dnes. Z toho důvodu jsem se v této první knize dopustil jednoho prohřešku proti koncepci n-vrstvové architektury a proti zásadám komponentního přístupu. O této chybě bude nyní tento článek. Ukážeme si také, jaká je podstata této chyby a také jak se jí vyvarovat.

Nutno podotknout, že z hlediska samotných principů OOP se o nějakou podstatnou chybu ani tak nejedná. Avšak pokud by někdo chtěl přijmout tuto chybnou pasáž jako východisko pro tvorbu systému pomocí komponent a jako východisko pro tvorbu n-vrstvové architektury, narazil by na problémy.

Povězme si o nich.

9.2 Popis chybného modelu v knize

Začneme ve zmíněné knize ***Základy objektově orientovaného programování*** na straně 197. Zde se zavádí v příkladu pro editaci osob následující myšlenková konstrukce, která není přesná z hlediska modelování v komponentním prostředí. Model je následující:

Osobu je třeba editovat. Proto se zavedla metoda pro osobu - nazval jsem ji v knize **`EditujSe`**. Lze tedy zavolat objekt osoby například takto:

```
Osoba.EditujSe
```

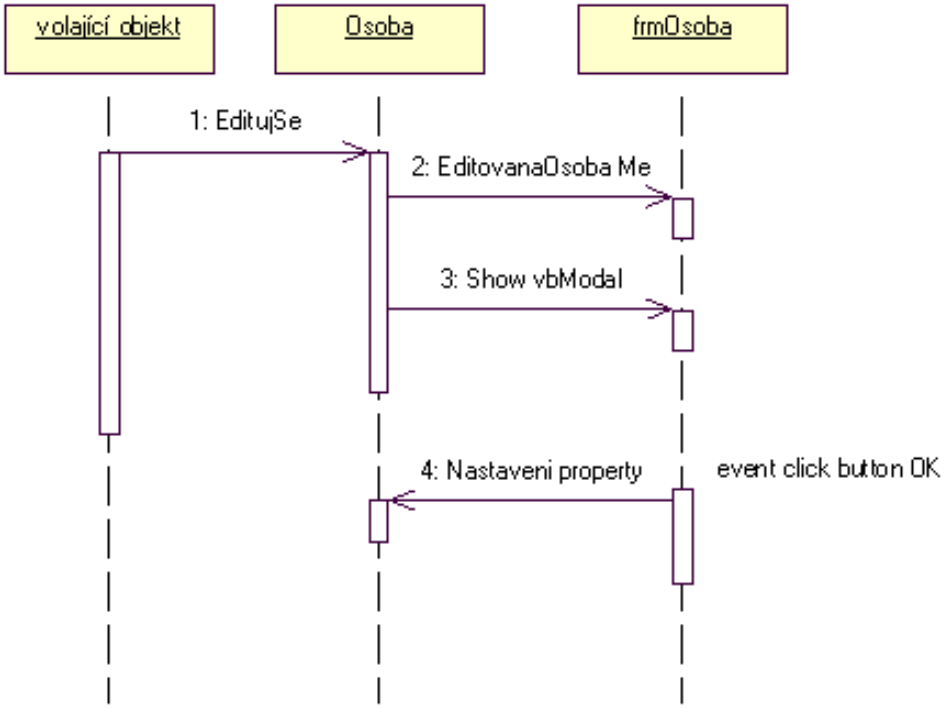
Tato metoda podle modelu v knize způsobí vyvolání scénáře:

```
Public Sub EditujSe()
            Set frmOsoba.EditovanaOsoba = Me
            frmOsoba.Show vbModal
End Sub
```

Kde **frmOsoba** je formulářem pro editaci osoby (obsahuje tři editační pole - pro rodné číslo, pro jméno a příjmení). Podle této konstrukce lze tedy volat objekt osoby, aby se editovala. Po stisku OK tlačítka ve formuláři **frmOsoba** dojde k "přesypání" informace z formuláře do objektu osoba (problém kontroly jsme vynechali).

Uvedená konstrukce na první pohled vypadá jako velmi elegantní - osoba má mezi svými "možnými uměními" také svou editaci. Uživateli objektu stačí zavolat toto "umění po editaci" a vše ostatní obslouží sama osoba.

Pokud bychom tuto konstrukci chtěli zapsat pomocí UML, potom by scénář posílání zpráv mezi objekty vypadal následovně:



Nejprve volající objekt (nějaký blíže nespecifikovaný objekt - uživatel objektu osoby, který je anonymní - nezajímá nás v tomto scénáři, kdo to je) zavolá osobu, aby se editovala. Ta pošle zprávu formuláři, která nastaví property editovaná osoba ve formuláři na sebe. Poté zavolá formulář, aby se zobrazil.

Nakonec nám sekvenční diagram umožňuje zapsat ještě pokračování tohoto scénáře dále - zapsali jsme symbolicky, že po stisku tlačítka OK dojde k volání všech property osoby a předají se jí informace z formuláře. Znázorněná zpráva "Nastaveni property" číslo 4 v předešlém obrázku má tak povahu celé nedekomponované skupiny zpráv.

9.2.1. V čem je chyba

Uvedený scénář bude funkční a dokonce se vám může i líbit, ale... má jeden háček. Nejprve si tohoto háčku všimněme v teoretické rovině.

Při návrhu v komponentním prostředí a při přechodu na třívrstvovou architekturu je business vrstva oddělena od GUI vrstvy a není zásadně problémem business vrstvy, jak se má daný objekt zobrazit. Bohužel v našem případě toto již neplatí. Zavolání objektu osoba EditujSe vyvolává přímo formulář a ten je obsažen v kódu metody objektu osoba. Znamená to, že objekt osoba obsahuje ve své působnosti formulář a podle pravidel "čisté teorie" jsme porušili jednu ze zásad komponentní technologie. Obrazně řečeno, objekt osoba je "zašpiněn" a není čistý - lze v něm vyhledat GUI prvek.

Poznámka: Například podle stejného pravidla nesmí objekt business vrstvy obsahovat hlášení - okno typu `MsgBox`.

Tolik teorie, ale samozřejmě nás zajímá, k jakým důsledkům vede porušení této zásady v praxi. Samozřejmě každá teoretická poučka má svůj obraz v praxi a tedy ptáme se, co tedy konkrétně chybného nastane.

Pro ilustraci představme si, že objekt osoba existuje na jednom stroji a formulář má existovat na druhém stroji. A navíc přidejme ještě více klientů s formuláři na více různých strojích. Z hlediska třívrstvové architektury se jedná o "klasickou situaci", která je požadována jako výsledek našeho snažení při návrhu třívrstvové architektury. V té chvíli zjišťujeme, že uvedený model nemá řešení.

Zamíchání formuláře do objektu osoba "předjímá" architekturu, která bude zvolena - a totiž formulář se objeví v rámci dané aplikace EXE pouze tam, kde je objekt osoba. Oddělit skutečně fyzicky formulář od osoby již teď není možné z prostého důvodu - jsou spolu spjaty jako siamská dvojčata.

Ještě více problém vynikne, když budeme chtít tuto aplikaci přenést na Internet anebo na Intranet. V té chvíli plně vynikne nesmyslnost zamíchání formuláře do objektu osoby. Vždyť k čemu nám je vyvolání formuláře na serveru, kde je objekt osoby, v místnosti tisíc kilometrů daleko...

Připomeňme, že z hlediska objektového modelování nemá `MsgBox` z úplně stejného důvodu co dělat v objektu business logiky.

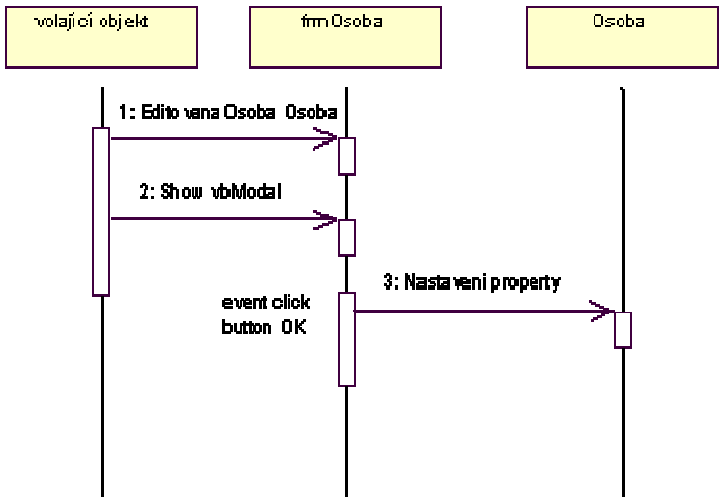
Problém zmíněného příkladu je v tom, že pokud bychom zůstali u Visual Basicu a ponechali řešení takm jak je pouze v jeho působnosti (jako desktop aplikace), mohli bychom tuto konstrukci přijmout (s určitým tušením problémů - kdo ví, kdy bude třeba změny?). Charakteristickým znakem řešení v komponentním prostředí je však nezávislost na způsobu implementace a ta je zde porušena - tedy nelze tento přístup obecně použít.

A jaké je tedy správné řešení?

9.2.2. Správné řešení

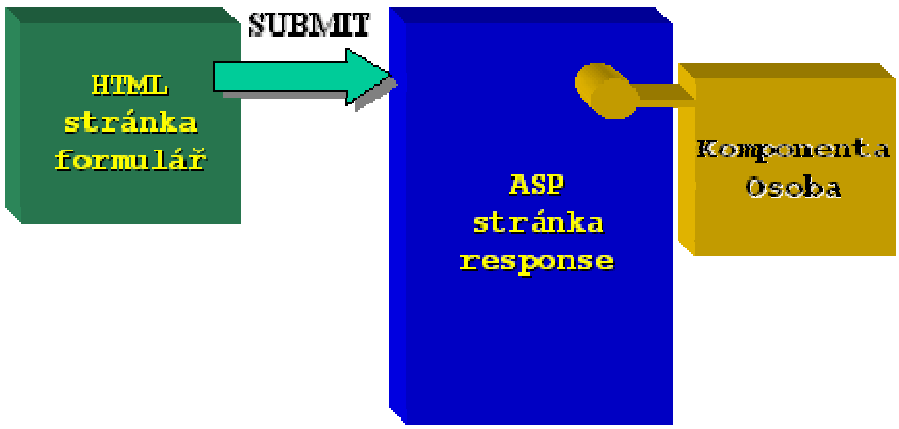
Fyzické oddělení prvku formuláře od business logiky vyžaduje, aby scénář editace v našem příkladu zahajoval formulář. Tedy volající objekt nevolá objekt osoby, ale formulář a ten teprve prvek business logiky, tj. osobu.

Jak si ukážeme, tento model umožňuje objekt osoby použít libovolně v jakémkoliv scénáři podle toho, jak poskládáme v architektuře prvky - komponenty:



Na objektu osoby se v tomto případě vyžaduje jediné - přijmout property hodnoty.

Pokud bychom přešli se komponentou osoby na Internet resp. Intranet, potom můžeme znázornit zapojení objektu osoby do aplikace například takto:



Scénář zůstává i v této technologii stejný. Nejprve se vyplní formulář (v tomto případě HTML klient) a objekt osoba přebírá hodnoty property přes ASP stránku, do které je jako komponenta zapojen.

Tímto způsobem lze jednoduše komponentu osoby - díky své nezávislosti na prostředí zapojit do libovolné aplikace, která může být tvořena komponentním způsobem (například zapojení do skriptu apod.).

9.2.3. Závěr

V uvedeném článku je poukázáno na jednu chybu v knize ***Základy objektově orientovaného programování***. Tato chyba souvisí se špatně zvoleným modelem spolupráce objektů z hlediska rozložení objektů do vrstev - objekt business vrstvy se stal v tomto chybném modelu závislým na GUI prvku, což by z hlediska zavedení komponentního programování vedlo určitě k problémům. Je třeba striktně oddělovat business prvky od GUI prvků.

Pravidlo “jak se vyvarovat této chybě” je v tomto případě naštěstí velmi jednoduché - v prvku business logiky nezavádějte volání prvků formuláře a dodržujte scénář “od actora ke GUI a od GUI k business prvku”.

10. Nejčastější chybné postupy v objektové analýze a návrhu vypozorované v praxi

Příspěvek z konference, viz adresa:

http://www.pef.czu.cz/aktivity/KonferenceObjekty/pozvanka_Objekty2000.html

Abstrakt: V tomto příspěvku jsou popsány nejčastější chyby v modelování podle OOP a UML v komponentním prostředí, se kterými jsem se setkal v praxi při konzultacích ve firmách při zavádění OOP, UML a komponentní technologie. Bližší podrobnosti naleznete také v [1].

Klíčová slova: UML, OOP, COM, OA & OD

10.1 Úvod

Poslední dva roky jsem se intenzivně věnoval konzultační činnosti při zavádění OOP a UML u mnoha softwarových firem a také jsem pracoval jako externí spolupracovník při tvorbě analytických modelů informačních systémů tvořených v UML. Oboje mi nabídlo novou a nezaměnitelnou zkušenost: Ukázalo se mi:

- jak vlastně přemýšlejí ti, kteří potřebují poradit v OOP a UML
- jakých se dopouštějí chyb ti, kteří začínají s OOP a UML
- kde a jak se vlastně tvoří omyly a mýty o OOP a UML
- jak postupovat (a jak ne) při zavádění OOP a UML
- kde začínat s OOP a UML.

Musím přiznat - také jsem se setkal s nezdary se zavedením OOP a UML ve firmách (naštěstí ojediněle) a z těchto jsem se poučil nejvíce. Celkově právě tato zkušenost získaná z mnoha konzultací je pro mne velmi poučná a nezastupitelná.

Zajímavé na všech konzultacích v těchto firmách, které chtěly zavádět OOP a UML (až na několik výjimek) bylo to, že na začátku ve všech firmách panovalo velké rozladění ze způsobu tvorby SW strukturálním a neobjektovým způsobem. Současně toto rozladění doprovázela obrovská touha po

změně technologie tvorby SW k “něčemu lepšímu”, k “něčemu systematičtějšímu”, k “něčemu průhlednějšímu” a k “něčemu stabilnějšímu”. Tito pracovníci tedy většinou snili o lepším způsobu tvorby SW a při seznámení se s OOP a UML najednou zjistili, že tento přístup je právě splněním onoho snu.

Trýznivý stav ve firmách před zaváděním OOP a UML mohu stručně charakterizovat takto:

- tvoříme software těžkopádně
- vyrábíme SW s chybami a výsledný informační systém je nestabilní
- obtížně dokumentujeme výsledky práce
- nevíme, jak zapsat analýzu a design
- pokud dojde ke změnám v zadání, obtížně tyto změny zavádíme anebo nejsme schopni tyto změny uhlídat
- tvoříme SW bez nějakého “chytrého systému ve firmě”
- tým pracuje bez koncepčního řízení a každý v týmu a ve větších firmách týmy mezi sebou “bojují každý sám za sebe”
- opakujeme práci s minimálním re-use
- a jiné negativní projevy, vedoucí ke špatným vztahům a mnohdy i k znechucením pracovníků...

Všechny tyto symptomy a ještě mnoho dalších “negativních” se shrnuje ve tvorbě SW pod jeden stručný pojem: **Nekvalita**.

Opravdu přechod na objektové programování a poté na objektové modelování pomocí UML, to je řeč o kvalitě tvorby SW se všemi důsledky z toho plynoucími. Z hlediska zaměstnanců se jedná o příjemnější způsob tvorby SW a z hlediska firmy se jedná o celkové drastické snížení nákladů díky zvýšení kvality SW, rychlejší a kvalitnější tvorbě SW a díky snížení nákladů na údržbu.

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

Na druhé straně jsem ve firmách působil jako externí spolupracovník a podílel se přímo na tvorbě modelů v UML několika informačních systémů také jako analytik v OA. Dá se říci, že pod mýma rukama prošla spousta modelů UML a to i modely s chybami, tedy měl jsem možnost pracovat s objektovými modely chápanými jako modely horší a modely lepší. Obrovské kvantum příkladů z OOP a UML mne naučilo rychlému náhledu na systém a k rychlé tvorbě modelů v UML a také mi ukázalo, jak se vyvarovat nejčastějších chyb a omylů, a tedy jak se vyhnout složitým a neschůdným řešením a to hned na začátku návrhu systému.

Právě o nejčastějších chybách při objektovém modelování, tj. v čem nejvíce “tápou” začínající s OOP, je tento příspěvek

10.2 Zaměňování třídy a instance a chybné vyhledávání tříd

Pokud se správně nevysvětlí pojem třída, většinou se začínající pracovníci v objektovém prostředí dopouštějí “fatálních chyb”.

Pro vysvětlení pojmu třída je třeba těmto chybujícím zdůraznit, že by bylo možné v “obecné teorii OOP” hovořit o objektu bez třídy, tj. lze deklarovat a zavést jeden objekt jakoby “přímou definicí”. Stačilo by definovat jeho vlastnosti, tj. jaké má nový objekt atributy, jaké má metody, jaký má protokol zpráv a z jakých objektů je složen. Tento způsob tvorby objektů “deklaruj jeden objekt po druhém” je podle “čisté” teorie OOP možný, má však jednu nevýhodu: Pokud budeme deklarovat “objekt úplně stejných vlastností” podruhé a lišící se pouze názvem, potom se v definici budeme opakovat.

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

Znamená to, že pojem třída je pouze něčím, co nám napomáhá definovat nové objekty jako jejich forma, jejich kopyto, jejich šablona. Jenom pomocí třídy nic nenaprogramujeme, stejně jako jenom pomocí šablony na boty (kopyta) nikoho neobujeme. Musíte pomocí šablony objekt nechat zrodit (Create, Set New, volání constructoru ... apod.).

Protože třída je v (čistém) OOP objektem, má smysl hovořit o metodách a atributech třídy jako u každého jiného objektu. V praxi jsem se setkal velmi často s chybným chápáním třídy, kdy se zaměňovala třída s instancí. Hlavním příčinou tohoto nepochopení je to, že existují jazyky sice objektové, ale “méně čisté” a v nich je třída zavedena jako typ proměnné a nikoliv jako objekt. V tom případě funkcionalita třídy jakoby teoreticky postrádala smysl (…jak chápat metodu a atributy u něčeho takového, jak je typ proměnné?), i když je metoda třídy v těchto jazycích zavedena.

Hlavní chybou v chápání třídy je očekávání od třídy “více a tedy něčeho jiného” než to, co tu bylo řečeno.

10.3 Polymorfismus za každou cenu

Setkal jsem se s mnohými způsoby vysvětlení polymorfismu - některé velmi nepřesné a velmi složité. Přitom vysvětlení polymorfismu je jednoduché.

Představme si tu situaci, kdy dva různé objekty mají ve svém protokolu stejnou zprávu, ale každý z nich na ni reaguje jinou metodou. Logicky vzato, pokud je možné oběma objektům poslat tutéž zprávu, u každého z nich to vyvolá jinou metodu.

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

Stručná definice polymorfismusu: jedna stejná zpráva, dva různé objekty, dvě různé metody.

Setkal jsem se dost často s mylným názorem a mýtem, že správný objektový program má obsahovat polymorfismus a tak se zavádí i tam, kde není třeba. Pokud se polymorfismus takto nadbytečně zavede, potom se objektové struktury stanou velmi nepřehledné, protože se dávají ”do jedné rodiny” objekty, které spolu vůbec nesouvisejí.

10.4 Objektová reference a její chybné chápání

Každý objekt musí být jednoznačně ***identifikován,*** aby mohl být nadřízeným objektem osloven při zaslání zprávy. Pochopitelně sama identifikace přímo implikuje možnost oslovit “ten správný objekt” (tj. programátor ví, co píše) a současně identifikace implikuje také možnost s daným objektem opakovaně pracovat (je to tentýž objekt jako před tím oslovený).

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

oslovení pomocí názvu objektu realizováno v 90% pomocí proměnné typu ukazatel na alokovanou paměť, kde se oslovený objekt nachází. Pomocí tohoto přístupu “držet” objekt znamená “držet” v proměnné s názvem objektu hodnotu ukazatele na objekt, tj. “ukazovat si” na konkrétní místo v paměti a tam se nachází oslovovaný objekt. To je však implementační pohled - z hlediska modelování hovoříme na abstraktnější úrovni pouze o identifikaci objektu pomocí jednoznačného názvu objektu a nestaráme se o to, jak konkrétně je toto oslovení pomocí názvu provedeno.

Vztah mezi objekty je zdánlivě podobný datovému vztahu v ERD, kdy přes klíč jsou provázána data z různých tabulek. Avšak existuje tu jeden velmi podstatný rozdíl vyplývající z vlastnosti zapouzdření objektu.

Pokud někdo používá objekt A, potom tento objekt má v sobě další objektové reference ukryty jako svoje vnitřní záležitosti a dotyčný uživatel objektu A o těchto referencích nic neví. Tedy tento pohled odpovídá doslova přirovnání uzavřené krabičky obsažené v jiné uzavřené krabičce (krabičky v krabičkách).

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

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

Nedocenění této vlastnosti vede k silnému narušování objektových modelů.

10.5 Porušení anonymity klienta

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

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

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

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

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

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

Narušení anonymity klienta znamená silné znepřehlednění programu a následně vznikají nestability v systému.

Jako zvláštní případ narušení anonymity klienta bych uvedl přecenění významu model elementu Actor v Use Case modelu. Setkal jsem se s případem, kdy tato chyba vedla až k fatálnímu zádrhelu tvorby modelu, protože se začaly míchat "actoři" do tvorby uvnitř systému ve funkcích rolí u přístupových práv a s každou novou rolí (nikoliv funkcionalitou) se musel měnit model a přidávat nový Actor…

10.6 Narušení zapouzdření ano či ne

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

```
Public MyAtribut as String
```

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

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

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

Opravdu tato konstrukce `Public atribut` není v OOP povolena a dokonce je přímo zakázána. Narušení zapouzdření objektu vede narušení dvou základních "kladných" vlastností v OOP a těmi jsou ***konzistence vnitřních stavů objektu*** a ***úplnost informace objektu***. Rozeberme si tento problém podrobněji, protože je to problém podstatný. Ukážeme si také, že se nejedná o nějaké pouhé plané filosování. Porušení principu zapouzdření má pro vývoj informačního systému katastrofální a tedy velmi praktické a (bohužel) velmi negativní důsledky.

10.6.1. Porovnání strukturálního programování a OOP

Ve strukturálním (tj. "neobjektovém") programování existují dva základní okruhy prvků v systému - prvním okruhem prvků jsou "data" neboli proměnné a druhým okruhem jsou "funkce" neboli procedury. Přitom u pojmu "data" není podstatné, zda se jedná o perzistentně uchovanou informaci na disku anebo zda se jedná pouze o proměnnou dočasně umístěnou v paměti. V obou případech se pod pojme data schovává tentýž typ prvku v systému, který má základní vlastnost - ***možnost něco si pamatovat*** tj. "podržet si" nějakou informaci (a podotkněme, že "nic víc"). Tento první okruh prvků ve strukturálním programování reprezentuje tedy něco jako paměť informačního systému.

Druhým okruhem prvků informačního systému jsou funkce (procedury apod.), což jsou posloupnosti činností systému reprezentované konkrétním kódem. Tyto prvky systému jsou výkonnými prvky a jejich úkolem je pracovat nad prvním okruhem - nad daty. Jsou to funkce, které mění obsahy dat. Z toho také vyplývá zavedení archaického názvu pro informační systémy jako "systémů pro zpracování dat". Pomocí funkce lze tedy s informací uchovanou v datech pracovat. Až teprve kombinace "funkce-proměnná" dává ve strukturálním programování části systému relevantní smysl.

V objektovém programování tomuto vztahu "výkonná část a data" odpovídá vztah metody objektu a atributu objektu a posloupnosti volání funkcí odpovídá posílání zpráv mezi objekty. Hodnoty atributů daného objektu reprezentují stavy tohoto objektu. Důležité je (a dokonce nejdůležitější), že pokud dodržíme princip zapouzdření, potom jedinou možností, jak změnit hodnotu atributu objektu, je poslat tomuto objektu zprávu a vyvolat tak jeho metodu. Pouze metoda objektu mění hodnotu atributu objektu. V této poslední větě je také uschován princip zapouzdření - jediná možnost změny hodnoty atributu je změnit jej pomocí běhu některé z metod objektu.

Konzistence vnitřních stavů objektu

Protože možné hodnoty atributů reprezentují možné stavy objektu a protože jedinou možností, jak změnit hodnotu atributu, je vyvolat metodu objektu, tak z toho plyne, že jedinou možností ***jak změnit vnitřní stav objektu je změnit jej přes jeho metodu a nijak jinak***. Tento závěr je velmi důležitý. Ve svém důsledku tento závěr znamená, že výčtem všech metod objektu také dostaneme výčet všech možností, jak měnit atribut a tedy jak měnit stav objektu. Tímto je sám objekt (a nikdo jiný) plně odpovědný za změny svých stavů. Změny stavů objektů jsou determinovány pouze jeho metodami (což neznamená, že je stav objektu determinován plně - kdoví, kdo a kdy bude tyto metody volat – to je princip anonymity klienta!).

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

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

10.6.2. Klasický příklad narušení konzistence vnitřních stavů

Většina současných systémů založených na OOP patří z hlediska perzistence dat do tzv. hybridních systémů. Znamená to, že o perzistenci dat se v pozadí stará relační databáze. V OOP má relační databáze mnohem podřadnější význam než ve strukturálním programování, kde je ERD ústředním diagramem. V OOP se databáze chová vůči žijícím objektům business vrstvy pouze jako úschovna zavazadel. V určitém okamžiku jsou objekty datové vrstvy požádány od objektů business vrstvy o “odložení” dat podle nějakého algoritmu. V jiném určitém okamžiku jsou objekty datové vrstvy požádány o “vydání těchto dat” podle téhož algoritmu, jak byly do databáze vloženy. Podotkněme, že v relační databázi je tímto algoritmem shoda hodnot klíčů mezi tabulkami. Existují i jiné efektivnější algoritmy “uložení a vyložení souvisejících zavazadel”, které zaručují mnohem vyšší rychlost a flexibilitu - například algoritmus tzv. post-relační databází založený na rychlých stromových strukturách.

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

Objekt má data odložena v databázi, avšak to je pouze jeden z možných stavů objektu. Objekt také nemusí mít svůj obraz v databázi, například těsně po svém zrodu. Učiňme dohodu, že objekt, který nemá svůj obraz v DB, bude mít OID rovno -1 a ten objekt, který bude mít svůj obraz v DB, bude mít OID rovno přímo hodnotě ID v tabulce (tj. kladné hodnoty).

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

- Zrod objektu. OID implicitně rovno -1
- Vyplnění formuláře uživatelem a poté převzato do objektu
- Zavolání metody INSERT objektu, požádání objektu datové vrstvy o INSERT s návratovou hodnotou IDENTITY
- Dosazení navráceného IDENTITY do ID a pokud se operace nezdařila, je ID rovno -2

Všimněme si, že v tomto scénáři se atribut ID objektu mění podle stavu objektu - buď je ID rovno -1 (objekt ještě není uložen), nebo je ID = -2 (nezdar) anebo je rovno ID v tabulce databáze a v tom případě objekt má svůj obraz vůči odloženým datům (ví, jak svá zavazadla z úschovny opět vybrat).

A nyní se zeptejme - má smysl toto ID povolit jako `Public`? Pokud tak učiníme, povolíme dosud neznámým funkcím zasáhnout do jinak uzavřených scénářů nabytí hodnoty ID a narušit tak konzistenci stavů objektu. Každý analytik by samozřejmě "vyskočil až ke stropu", kdyby někdo chtěl toto ID učinit `Public` - vždyť mechanismus chodu ID musí být determinován uzavřenými scénáři chování objektu a ničím jiným. Narušení zapouzdření a tedy povolení změny atributu naruší jinak logické chování objektu a silně destabilizuje systém. Pokud povolíme atributy jako Public, potom ztrácíme kontrolu nad objekty a také nad vývojem systému. Zapouzdření vede k logickému chování objektů a ke konzistenci stavů, přitom porušení zapouzdření vede k nedeterminovanému chování objektů a v konečném důsledku ke ztrátě stability systému. Systém se zapouzdřenými objekty drží ve svých objektech konzistenci a proto je o mnoho stabilnější a mnohem snadněji se vyvíjí.

10.6.3. Zapouzdření a úplnost informace objektu

V předešlém odstavci jsme popsali jeden z důsledků zapouzdření z hlediska "logického chování" objektu. Vnitřní konzistence stavů objektu znamená, že jedinou možností jak ***změnit vnitřní stav objektu je změnit jej přes jeho metodu a nijak jinak***. Narušení zapouzdření objektu v konečném důsledku znamená ztrátu konzistence stavů objektu. Objekt začne měnit svoje stavy, aniž by v něm proběhly jakékoliv procesy. Takovéto chování objektu není ničím jiným, než změnou stavu bez vnitřního chování, tj. nelogické chování – například v SW je možní učinit chybnou konstrukci: "jablko je shnilé, aniž by shnilo".

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

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

Existuje ještě druhý velmi podstatný závěr plynoucí z vlastnosti zapouzdření a tím je **úplnost informace objektu**. Tato vlastnost souvisí s analytickou přesností a úplností pojmů, které skládají další pojmy a tak tvoří systém. Zajímavé je, že samotné úvahy o úplnosti informace objektu se budou jevit (jak si ukážeme) jako velmi jednoduché triviální, ale přitom tyto jednoduché zásady nejsou již z principu ve strukturálním programování dodržovány a málokdo si to uvědomuje. Ve strukturálním programování neexistuje ekvivalent této vlastnosti a tím dochází k zajímavým efektům zalepování a přilepování IS, k tvorbě "nepřehledných lepenců systému", k nedělitelným "molochům" a na druhé straně k rozbíjení pojmů, jejich tříštění a jejich nejednotnosti.

Představme se, že jako analytik popisujete nějaký pojem, který bude figurovat jako entita v informačním systému. Jako klasický příklad zvolme pojem Faktura. Můžeme provést následující stručný popis:

Faktura "vidí" Partnera (tj. Dodavatele resp. Odběratele), obsahuje Datum vydání, Datum dodání, obsahuje Řádky faktury atd.

Takto bychom popsali informaci faktury. Je zřejmé, že tímto popisem vymezujeme oblast pojmu Faktury, tj. to co do ní patří a naopak také pochopitelně tímto popisem automaticky vymezujeme, co do Faktury nepatří. Všimněme si například, že sama Faktura nevidí Zboží faktury, ale Zboží je

přístupné a viděno (například) až Řádkem faktury. Podobný popis proběhne v myšlenkách analytika. Konkrétním výrazem těchto na první pohled abstraktních úvah je potom některý z diagramů objektového modelování zapsaný pomocí UML (například objektový model a model tříd).

Přitom se může zdát jako velmi triviální následující myšlenka:

Když buduji pojem Faktura, který má v OOP díky povaze objektů poté i svůj přesný obraz v objektech, tak tímto pojmem mám na mysli vše, co má tento pojem obsahovat. Samozřejmě současně s tímto pojmem nemám na mysli to, co tento pojem obsahovat nemá*.

Celá Faktura jako objekt je opět složena z jiných pojmů - objektů (do kterých v této úvaze nyní nevcházím, například Řádky faktury apod.) a tím tento pojem Faktury vytváří určitou oblast svého existenčního vymezení. Jednoduše řečeno - “Faktura je jako objekt toto a toto” a tím automaticky není “vše ostatní”. Jak triviální úvaha, ale jak často není takováto jednoduchá zásada v objektovém programování (natož strukturálním) dodržována! Tato myšlenka je vyjádřením úplnosti pojmu Faktury.

Připomeňme, že ve strukturálním programování složeném z dat a funkcí nemá tato myšlenka úplnosti resp. neúplnosti pojmu smysl. Program je totiž složen z funkcí a dat a nikoliv z objektů. Určitou obdobou se může jevit zavedení modulárního programování, tj. zavést moduly (někdy nazývané knihovny), v kterých jsou něco jako “informace” daného pojmu ve tvaru funkcí a dat zavedeny a odloučeny od ostatních částí systému. Avšak strukturální modulární programování má svá omezení daná zejména tím, že velmi těžko vytvoříte několika-násobnou instanci jednoho modulu (několik instancí od sebe identifikovaných). Pokud například vytvoříte modul “Řádek faktury”, velmi obtížně se v tomto modulárním prostředí tvoří “Řádky faktury”, tj. několik instancí modulu vedle sebe. Modul je totiž zaveden pouze jako jeden, čímž vznikají problémy s multi-instancemi. Souběžně s tím vzniká problém identifikace těchto entit ze stejného modulu (jeden řádek, druhý řádek, faktura číslo X, faktura číslo Y atd.). Tento problém se musí řešit pomocí datových vztahů. Podotkněme, že pokud zavedete modul s možností několika instancí, přecházíte tím do oblasti OOP a tento modul odpovídá možnému implementačnímu zavedení třídy. Klasickým příkladem je rozdíl mezi BAS a CLS modulem ve Visual Basicu. Můžeme BAS modul chápat jako případ třídy s možností vytvořit pouze jednu instanci, kterou tímto nemusíme identifikovat od jiné instance a tedy nemusíte tuto instanci oslovovat názvem (vznikají tak systémové funkce jako metody objektu systému).

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

10.6.4. Narušení zapouzdření a neúplnost informace objektu

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

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

Uveďme si klasický příklad na úplnost a neúplnost informace objektu:

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

10.6.5. Nepříznivé důsledky narušení úplnosti informace objektu pro vývoj IS

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

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

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

10.7 Chyba tříštění objektů

Tato chyba souvisí s předešlými chybnými postupy. Analytik, který se snaží vystihnout daný problém, uvažuje pouze v pojmech nižší úrovně a není schopen z těchto pojmů vytvářet další (jinak logicky chápané) pojmy vyšší abstrakce. Jedná se anachronismus převzatý ze strukturálního programování, kde každá proměnná bojuje sama za sebe a pojmy jako takové neexistují.

Například analytik uvažuje o Ceně zboží, o Druhu zboží, atd. a přitom jej nenapadne, že hlavním pojmem je Zboží samo o osobě. Výsledkem je snaha "nacpat" Cenu zboží do Řádku faktury a nikoliv objektovou referenci na celé Zboží.

10.8 Chyba přebytečného ID

V mnoha případech se při použití relační databáze jako uložiště dat pro objekty zavádí přebytečné ID v tom smyslu, že kromě objektové reference na daný objekt se navíc uvede mezi atributy "cizí klíč ID".

Příklad: Řádek faktury "vidí" svou Fakturu, tedy má referenci na objekt Moje_Faktura a navíc se do Řádku faktury přidá ID Faktury. Jedná se vlastně o modifikaci chyby tříštění objektu, protože ID Faktury patří do Faktury a nikam jinam!

10.9 Chyba siamských dvojčat při návrhu komponent

Hlavní příčinou této chyby je podcenění použití jinak velmi doporučovaného model elementu **Package** v návrhu pomocí UML anebo jeho žádné použití (…a přitom zavedení Package patří k základním mechanismům UML!). To vede nakonec k velkým problémům při návrhu komponent. Chybný analytický návrh s "nečistými" pojmy v analýze vede nakonec k tzv. circular reference mezi třídami a výsledkem je nemožnost oddělit pojmy od sebe do komponent, pomyslné části systému "namalované tužkou" nelze od sebe oddělit jako siamská dvojčat. Vznikají tak nedělitelné systémy jako "molochy", což s s sebou přináší nemožnost dělení systému na menší celky. Nejenom že takovýto systém se musí vždy dodávat jako jeden celek (což je vzhledem ke vztahu k zákazníkům velmi nepříjemné), ale řízení projektu v takovém prostředí je velmi obtížné. V nedělitelném molochovi, který touto chybou vznikne, opravdu platí, že všechno souvisí se vším a žádná z kapitol vývoje není nikdy uzavřena.

10.10 Úskalí tvorby sdílených komponent

Pokud zavedete komponentu, ať už na síti, nebo na lokále, máte z hlediska jejího postavení vůči jejím klientům dvě základní možnosti, jak se instance této komponenty bude chovat (samozřejmě tuto volbu provádíte před jejím vytvořením ve fázi designu).

- Instance komponenty je sdílená klienty

- Instance komponenty není sdílená klienty a pro každého klienta existuje nová instance komponenty. Dále takovou komponentu budeme nazývat jako komponentu s izolovanými klienty

Zde je třeba se trochu se zastavit, protože jsem se při konzultacích setkal s mnoha nepochopeními právě v této oblasti.

V prvé řadě všimněme si, že rozlišujeme instanci komponenty a komponentu. (Většinou se toto rozlišení neprovádí a směšují se oba pojmy). Pod instancí komponenty máme na mysli jednu existující žijící strukturu binárního útvaru - binárního objektu. Obecně z jedné komponenty můžeme vytvořit N instancí komponenty (pokud to model komponenty dovoluje). Pod samotnou komponentou naopak chápeme to, co máme v systému nainstalováno jako možnost tvořit instance z této komponenty. Vztah mezi instancí komponenty a komponentou je podobný, jako je mezi objektem a jeho třídou.

Samozřejmě oba typy komponent - jak sdílená, tak s izolovanými klienty - mají diametrálně odlišné chování.

Objektový sdílený server

Sdílená instance komponenty funguje jako skutečný objektový server. “Stojí” v systému jako objektový stroj s objekty připravenými k použití a každý nový klient se může k tomuto stroji připojit a používat jej. Přitom tento server může být instalován na síti a sdílen klienty z různých strojů. Objektové struktury v instanci komponenty se jednou naplněné stávají všeobecně sdílenými a to “přímo a naživo”. Klienti mají přístup k objektům, které již někde v systému žijí (jednou nainstalované) a mohou tak pro klienty přímo pracovat. Pojem aplikační server má takto velmi konkrétní význam - spousta věcí z databáze odpadá, například někde již existuje seznam barev, seznam aut, stačí šáhnout do seznamu objektů...

Objektové izolované knihovny

Druhý typ komponenty, tj. komponenta s izolovanými klienty, je velmi odlišná. Každý klient, který chce používat instanci komponenty, si vybudí novou “svou a pouze svou pro sebe izolovanou” instanci ve svém vlastní viditelnosti a nikde jinde. Instance se nechová jako sdílený stroj, ale spíše jako přilinkovaná knihovna ke svému klientovi. Z podstaty věci je vyloučeno, aby tato instance byla viděna jiným klientem. Sama komponenta (nikoliv instance) může být nainstalována v systému “jednou” a je tedy sdílená, nikoliv však její instance. Sdílení znamená re-use kódu v tom smyslu, že nemusíme aplikaci (komponentu) distribuovat na klienty.

Který typ volit?

Vytvořit aplikační server jako sdílenou komponentu, tj. objektový server, je velmi lákavé, ale při přechodu od 2-vrstvové architektury k 3-vrstvové ***se může jednat o velmi nebezpečný krok***.

Pokud zvolíte typ sdílené instance komponenty, potom musíte očekávat tyto úkoly, které jste dosud neřešili:

- Řešíte zamykání objektů přímo v aplikační úrovni. Protože klienti instance komponenty přistupují k této komponentě kdykoliv pro editaci a změny, musíte vyřešit analytický problém zamykání objektů proti možné kolizi. Toto řešení nesmí způsobit deadlock komponenty (něco na způsob obsluha uzamkne a odejde na kafe).
- Řešíte transakce na úrovni objektů (konzistence stavů při změnách v objektech). Při změnách v objektech může kdokoliv jiný vstoupit do vámi připraveného scénáře a provést úpravy, které radikálně mění situaci.
- Řešíte přístupová práva na úrovni objektů - ve volání metod objektů se musíte odvolat na část systému, která analyticky řeší přístupová práva (agenda uživatelů apod.)
- Řešíte rychlost zpracování požadavků při růstu počtu klientů - multithreading komponenty (pro ActiveX EXE pod VB 6.0 velmi obtížné, spíše nemožné vůbec řešit pro STA model a frontování požadavků)

- Řešíte problém ztráty identity klientů databáze - connections se zúžil na jeden connection komponenty a pokud v databázi používáte k něčemu uživatele (user connection), nastává kolize, existuje pouze jeden user databáze, komponenta.
- Odpadá nejjednodušší možnost optimalizace aplikace, která spočívá v naplnění objektů pouze těch, které potřebuje klient a žádných jiných navíc (kolekce simulanti apod.)

Samozřejmě tyto problémy neznamenají, že nemáte přistoupit k řešení pomocí sdílené komponenty ze zásady, jsou pouze upozorněním a varováním. Řeč teď není o tom, zda je výhodnější použít sdílenou komponentu nebo komponentu s izolovanými klienty obecně, ale kterou vybrat při přechodu od dvouvrstvové architektury na třívrstvovou architekturu.

Odpověď je jasná - začněte vždy s izolovanými klienty. Pokud totiž měníte svoji technologii, tak zásadně platí “nejlepší je dosažení cíle při minimalizaci změn”. Pokud použijete sdílenou komponentu, potom vytvoříte spoustu sekundárních problémů k řešení, které máte momentálně ve 2-vrstvové architektuře již vyřešeny.

10.11 Závěr

Dvouletá praxe konzultací v několika desítkách firem u mne jednoznačně vede k jednomu důležitému závěru:

Teoretické poznatky shrnuté v OOP, specifikaci a doporučení UML a v návrhu komponent nejsou žádnou šedou teorií a mají své praktické a mnohdy až pragmatické důsledky.

11. Kam patří kontrolní mechanismy

11.1 Dotaz

Obdržel jsem jako reakci na předešlý článek, ve kterém je zdůrazněna nezbytnost zachovat business logiku čistou od GUI prvků, několik mailů.

Začněme s prvním:

```
Dobry den, se zajmem jsem si precetl dalsi clanky a tak jako onen pisatel
Vam taky dekuji za Vasi publikacni cinnost. Je videt, ze jste pro nas
autorita. Uz dlouho se tesim na knizku o objektove analyze. Potesilo mne,
ze problem z posledniho clanku jsem resil (asi intuitivne, vlastne to
vyplyvalo z logiky) dobre. Datovy objekt poslal misto dat stav zpracovani a
volajici si zpravu uzivateli zaridi sam. Mam ale podobnou otazku:

Kdyz logiku aplikace obsahuje bussines vrstva, proc je spousta logiky a
kontrol ve formularich? Nemel by formular rict
”objekte.zkontrolujSiTytoUdaje()"?

Mejte se fajn a at Vas bavi prace!
```

11.2 Odpověď

Takže samozřejmě díky za slova uznání, to potěší vždy. Nyní k samotnému dotazu. Jak je to s kontrolními mechanismy v OOP? Patří do formulářů nebo do objektů?

11.2.1. Kontrolní mechanismy

Dobrá teorie se pozná podle toho, že její závěry jsou velmi praktické. V předešlých článcích jsme se zabývali zdůvodněním zapouzdření objektů. Existují dva základní závěry hovořící pro zapouzdření objektů (viz předešlé články):

1. úplnost informace objektu, tj. daná entita obsahuje vše, co potřebuje a žádná informace spojená s ní není mimo ni.

2. konzistence vnitřních stavů objektu, tj. vnitřní hodnoty objektu se mohou měnit pouze přes metody daného objektu

Když se podíváme na oba tyto důležité závěry OOP, potom z čistě teoretického hlediska je zřejmé a jasné, že kontrolní mechanismy (tj. kontrola rodného čísla na modulo 11, kontrola čísla účtu apod.) by ***teoreticky měly patřit pouze do samotného objektu business vrstvy*** a tedy nikoliv do formuláře. Mechanismus kontroly je poté spojen s daným objektem a tím pádem se tento mechanismus s objektem nese. Algoritmus kontroly a její průběh se stává také vlastností "vnitřku" objektu nezávisle na kontextu použití objektu. Pokud tento objekt dosadíme do jiného prostředí (například zavoláme ActiveX DLL objekt ze skriptu Wordu nebo z jiného skriptu resp. z VB aplikace apod., v .NET assembly jinam linkujeme apod.) , potom si tento objekt s sebou nese tento kontrolní mechanismus sám a není třeba volat "něco mimo objekt". Většinou se mechanismus kontroly použije například při volání privátního veririfikačního algoritmu umístěného do volání nastavení Property objektu apod..

Dodržení této zásady, tj. aby objekt nesl vše to, co potřebuje s sebou, včetně kontrolních mechanismů, má za následek, že objekt je úplný a kontrolu můžeme vždy použít při volání jeho metod.

Představme si například, že objekt bude plněn z formuláře, tj. obsahy některých polí z formuláře jsou poté objektem přijímány do Property a uvnitř se volá mechanismus kontroly. Pokud "něco nehraje", potom sám objekt samozřejmě nevolá GUI, ale předá "dohodnutou" hodnotou zpět tomu, kdo jej používá, tj. předá odpovídající výsledek operace (např. neúspěch apod.), nebo vyvolá událost, výjimku apod..

V jiném kontextu stejný objekt (myšleno ze stejné třídy) však může přijímat například údaje z textového souboru, tj. z nějaké dávky, a představme si, že i tam bude vyžadováno použít takovýto verifikační mechanismus. Opět se použije "úplně stejný kód" již jednou psaný, přičemž teď už není přijímanou hodnotou textové pole z formuláře, ale textové pole načtené ze souboru. Podobně v jiném kontextu může objekt přijímat údaje z pole ADO z databáze a provést kontrolu, nebo v heterogenním prostředí přijímá zprávu z Message Queue Serveru apod.

Znamená to, že kontrolní mechanismus by měl být spojen s objektem. Důsledkem toho je maximální re-use objektu z dané třídy, protože mechanismus kontroly je vždy a všude přenášen spolu s objektem. Tedy první závěr je ten, že vzato z hlediska čistého OOP by kontrolní mechanismus měl v objektu být obsažen.

Znamená to, že návrh pisatele ve smyslu:

```
objekte.zkontrolujSiTytoUdaje()
```

jez hlediska OOP správný a přesný. Je třeba pouze podotknout, že mnohdy volání této metody bývá schováno uvnitř jiných metod, například v property apod., a nevolá ji přímo uživatel objektu. Scénář je například takový, že uživatel se pokusí nastavit nějaké property a uvnitř je volána tato metoda, kterou sám uživatel nepoužívá.

11.2.2. Ale existují ústupky...

Zdůvodnili jsme si, že objekt by měl nést kontrolní mechanismus v sobě, ale existují ústupky z důvodu optimalizace.

Připomeňme si, že každá optimalizace nějakého systému, tj. odstranění nějaké technologicky krizové situace, je ústupkem a handlem typu "něco za něco", např. rychlost za paměť, rychlost za zdvojení informace při indexech apod. I v našem jinak "teoreticky čistém příkladu" může nastat určitý ústupek od deklarovaných zásad.

Představme si, že jsme napsali aplikaci, ve které je objekt business logiky nějak ***fyzicky vzdálen*** od uživatele a prvek GUI komunikuje s objektem business vrstvy po síti, což může být lokální síť nebo Internet. Pokud existuje takováto hranice, potom se komunikace při vyplňování formuláře může

zkomplikovat a zpomalit, protože se kontrolní mechanismus přenáší po síti “tam a zpět”, což zatěžuje síť a zpomaluje aplikaci (velmi výrazné u Internet aplikací). Navíc pokud použijete ASP technologii resp. jinou Internetovou skript technologii bez remote scriptingu (vzdáleného volání skriptu na serveru “bez reloadu stránky”), tj. použijete pouze technologii Request - Response stránek (přenos informace je pouze pomocí FORM tagů), potom nastane i problém verifikace údajů v rámci daného otevřeného formuláře a tato verifikace se bohužel provádí až dodatečně po odeslání FORM údajů.

Tyto důvody vedou k ústupku od zásady, aby pouze business objekt obsahoval verifikační mechanismy a tyto mechanismy se přenášejí také na klienta objektu (což je z hlediska OOP samozřejmě nečisté, protože klient se o problémy objektu sám nemá co starat a vede to k nabývání redundantního řešení).

Tak se například ošetří JavaScriptem vstupní údaje ASP stránky, aniž by se volal server. Je pochopitelné, že pokud verifikační mechanismus přenesete mimo objekt, potom se dopouštíme rozbití objektu a nepoužíváme důsledně re-use a tedy opakujete práci.

Uvedený ústupek však neznamená, že daný objekt by neměl již tento verifikační mechanismus sám obsahovat! Pokud nastane podobná situace vzdáleného použití objektu, a chcete daný objekt používat také v jiném (například dosud neznámém kontextu), potom by měl být poskytnut výběr - objekt buď může použít danou verifikaci anebo nikoliv. Pokud tento objekt dosadíme do prostředí, které kontrolu obsahuje, potom můžeme volat metody bez verifikace, pokud jej dosadíme do prostředí, které neprovádí kontroly, budeme volat metody s kontrolami.

11.3 Závěr

V článku je zodpovězena otázka, kam umístit kontrolní mechanismy metod a správná odpověď zní - z hlediska OOP pouze do metod objektu a tím provést re-use těchto mechanismů kontrol. V opačném případě dochází k opakovanému řešení téhož problému (ztráta re-use).

Může však nastat ústupek při vzdáleném volání objektu, kde je mnohdy výhodné umístit kontroly do klienta objektu. V tom případě samozřejmě sice získáme lepší průchodnost aplikace, ale ztrácíme re-use.

Osobně se domnívám, že protože se jedná o ústupek vzniklý pouze kvůli problémům technologickým, určitě v budoucnu při zlepšení technologických podmínek nastane postupné opouštění tohoto ústupku pro získání lepšího re-use (rychlejší sítě, technologie event-driven ASP jako je v .NET apod.)

12. Objekty s historií v čase

12.1 Problém k řešení

Při pobytovém semináři ve Valašských Kloboukách byl vysloven následující dotaz:

V našich informačních systémech se vyskytuje spousta entit, které vyžadují sledovat historické údaje všech svých změn v čase. Například osoby vystupující ve smlouvách při změnách svých údajů sice tyto údaje mění v čase, ale jedná se stále o tutéž osobu. Například osoba jako slečna se provdá a změní tak svoje příjmení. Ve smlouvě potom tato osoba musí vystupovat sice jednou, ale musí umět nějak vydat jak svůj původní obraz (v čase sepsání smlouvy), tak současný stav. Oba stavy jsou důležité - jeden pro kontrolu správnosti vydané smlouvy (obraz smlouvy v čase sepsání), druhý je důležitý pro kontrolu osoby v současnosti.

Například existuje podobný požadavek u tzv. číselníků (někdy nazývaných kódovníky), což jsou seznamy nějakých jednoduchých entit typu kód + název:

- země (změna názvu země)

- ulice (přejmenování ulice),
- města (přejmenování města)
- barvy (změna názvu barvy v systému)
- a další

Změna názvu (přejmenování apod.) nesmí vést ke ztrátě reference a současně informace musí být úplná. Jak by se tento problém řešil pomocí OA & OD?

12.2 Odpověď

Předcházející dotaz je předložen analytikovi znalému OA. Je zřejmé, že na jeden problém existuje vždy několik možných řešení, uveďme si jedno z nich.

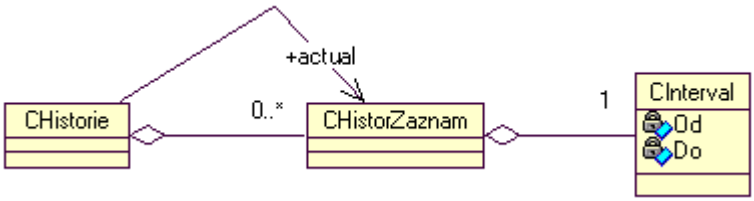
Musí existovat dvě roviny řešení, jedna abstraktnější a druhá konkrétnější. Tím se dosáhne maximální flexibility řešení problému při současném vytvoření vzoru typu generalizace - specializace pro další řešení pro další entity, které vyžadují historii v čase. Současně se požaduje, aby problémové okruhy jednotlivých entit byly od sebe izolovány a mohlo se tak přistoupit i k řešení pomocí komponent. Oba požadavky dané dohromady vedou k příznivé situaci, kdy další řešení pro další entitu je již na abstraktnější úrovni řešeno, tedy přidání kódu je minimalizováno jak je jen možné a současně toto přidání znamená pouze přidání komponenty bez zásahu do původního již zkompilovaného kódu. Jedná se pak pouze binární přilinkování nové binární knihovny k existujícím binárním souborům (možnost použití komponentní technologie).

Poznámka: Pokud vám tyto věty připadají příliš abstraktní, věžte, že jsou naopak velmi praktické pro řešení, jak si ukážeme dále. Řešení minimalizuje jak přidávání kódu, tak zásahy do původního kódu (nemusí se znovu kompilovat).

12.2.1. Abstraktnější úroveň

Na abstraktnější úrovni se navrhnou dvě základní třídy, nazvěme je `CHistorie` a `CHistorZaznam` (poznámka: Každá třída začíná předponou C jako Class)

Význam `CHistorie` je takový, že drží jako svou agregaci seznam objektů z `ChistorZaznam`, historické záznamy. Historický záznam `CHistorZaznam` obsahuje `CInterval` mající časový údaj údaj `Od Do`:



Jeden z objektů z třídy `CHistorZaznam` v seznamu `Historie` je platným - aktuálním a na ten si historie ukazuje asociací. Třída `CInterval` je vyvedena mimo historický záznam pro možnost zavedení dalších funkcionalit nad intervalem.

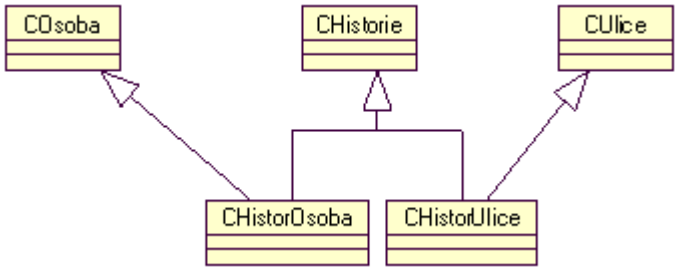
Na této úrovni je tento model tříd dostačující.

12.2.2. Druhá implementační úroveň

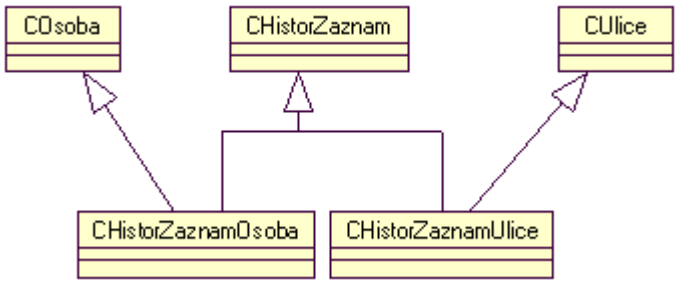
Pro každou entitu vyžadující sledovat svou historii v čase je třeba založit **tři** třídy:

- Třída dané entity (například osoba, barva, země, ulice). Jedná se o “čistou entitu” neznající žádnou historii (například u osoby se jedná o třídu obsahující atributy rodné číslo, jméno a příjmení). Název třídy se volí jako `C<název entity>` (např. `COsoba`)
- Třída entita s historií, název se zvolí jako `CHistor<název entity>` (např. `CHistorOsoba`, `CHistorZeme`, `CHistorUlice` apod.). Tato třída je současně specializací třídy `Cistorie` a současně je specializací třídy z bodu 1, tj. je specializací třídy `C<název entity>`.
- Třída historický záznam entity, název se volí `CHistorZaznam<název entity>`. Třída je specializací jak třídy bodu 1 (`C<název entity>`), tak `CHistorZaznam`.

Diagramy pro osobu a ulici vypadají takto:



a současně



Vysvětlivky diagramu:

Např. pro osobu:

Každá historická osoba je potomkem v gen-spec. třídy historie, která drží historické záznamy, takže i ona díky tomu drží své historické záznamy. V případě historické osoby jsou historické záznamy typu historických záznamů osoby. Historický záznam osoby je gen-spec potomkem historického záznamu, tj. má vlastnosti historického záznamu, např. obsahuje interval, a je a potomkem gen-spec u osoby, tj. zná současně také rodné číslo, jméno a příjmení. To že, sama historická osoba je také gen-spec potomkem osoby, je dobrou vlastností - jedná se o možnost obrátit se na historickou osobu přímo jako na osobu, což reprezentuje aktuální osobu. Můžeme s historickou osobou pracovat jako s osobou přímo bez kolizí typu.

Podotkněme, že každý historický objekt jako potomek historie musí umět nějak vydat objekt pro daný časový okamžik v minulosti (podle intervalů). Tuto metodu “dědí” ve vztahu gen-spec od historie.

Historická osoba má svoje property osoby delegováno na objekt aktuální historický záznam. Historická osoba tedy při dotazu na rodné číslo přesměruje tento dotaz na asociovaný objekt aktuální historický záznam osoby.

12.2.3. Případ užití pro přidání nového historického záznamu osoby, například u přejmenování

Nechť v okamžiku zahájení této činnosti je vybrána konkrétní historická osoba. Zobrazí se formulář pro změnu osoby (rč, jméno, příjmení). Protože je tato historická osoba současně aktuální osobou, z tohoto objektu historické osoby se převezmou property rodné číslo, jméno a příjmení a zobrazí se jako implicitní hodnoty formuláře.

Po odsouhlasení OK obsluhou se provede verifikace formuláře a současně se verifikuje, zda došlo ke změně aktuální osoby. Pokud ano, potom se v kolekci historických záznamů u aktuálního historického záznamu osoby změní v intervalu Do na hodnotu “teď”. Současně se vytvoří nový historický záznam osoby s novými údaji a s Od hodnotu ”teď” a přidá do seznamu historických záznamů. Tento objekt se dosadí do asociace aktuálního historického záznamu osoby. Nakonec se zavolá u objektů metoda pro persistenci do datové vrstvy. Podotkněme, že tento scénář je platný již na vyšší abstraktní úrovni, programátor pouze vyplní některé speciální kódy pro svůj případ (například porovnání dvou objektů apod.).

12.2.4. Postup programátora

1. Založí dvě komponenty...
2. Založí tři třídy...
3. Provede implementaci gen-spec (ve VB pomocí implementace interfaců) a vyplní svůj “speciální kód” ve svých třídách...
4. Založí další tři tabulky podle mapování do relační databáze...
5. Tímto je naprogramováno “vše” ohledně historie objektů...
6. Vytvoří formuláře atd...

12.2.5. Výhody tohoto postupu

- Část problému - jádro historie, je již dopředu naprogramováno a připraveno pro další použití. To, co je historie, již existuje a dokonce jako zkompilovaný zdroj pro binary re-use.
- Postup má menší zdroj chyb - jádro historie je jedno, je centralizováno a historie se neprogramuje znovu a znovu
- U nové entity je jasný postup - existuje vzor na abstraktní úrovni. Řešení je hotové a nemusí se provádět znovu analýza, diskuse atd.
- Programátor je nucen (kompilátor jej nepustí dále) vyplnit vzor při implementacích přesněji, než při řešení znovu a znovu novém, kde má velkou libovůli a tedy může sám zanést chyby
- Nemusí se linkovat žádná zdrojová knihovna a nic již hotového se znovu nekompiluje (komponentní technologie)

13. Objekty s historií v čase, agregace a asociace

13.1 Úvodní slovo

Uvedené příklady v této sekci mají posloužit jako ilustrace možného řešení některých obecně platných problémů pomocí OOP v zápise UML. V žádném případě si nevyhrazuji právo tvrdit, že nabízené řešení je nesporně to nejlepší a dokonce že nemůže obsahovat chyby. Každý navržený model lze vylepšit a v každém modelu se chyby mohou vyskytnout. Pokud bychom brali ohled pouze na bezchybnost a stoprocentní dokonalost, nikdy bychom nenaprogramovali ani jeden řádek kódu.

Následující příklady jsou spíše ilustrací "jak se dělá OA a OD pomocí UML" a nejsou to návody pro konkrétní hotová řešení. Budu jenom vděčný, pokud se najdou solidní oponenti některých mých analytických návrhů.

13.2 Návaznost na předešlý článek

V článku ***Objekty s historií*** je řešen případ takových objektů, které mají znát historii svých změn. Řešení uvedené v článku je však zjednodušeno na případ pouze "osamoceného objektu". Co když objekt s historií vstupuje do vazeb k ostatním objektům? Jak potom vypadá problém historie záznamů? Uveďme několik příkladů, kdy je naše řešení nedostatečné:

- Osoba má adresu, což je objekt z třídy adres. Pokud se změní adresa, tj. osoba se přestěhuje, jak se to projeví v osobě s historií? Mění se i ona se svou historií?
- Jak se projeví změna názvu ulice (přejmenování ulice) u dané osoby?
- Evidované auto má asociaci na jednu z barev ze seznamu barev, tj. ukazuje si na objekt barva z existujícího číselníku barev. Auto bylo k určitému datu přelakováno...
- Evidované auto má stejně jako v 3. odstavci asociaci na jednu z barev a tato barva změní díky přechodu na jiné evropské normy svůj název.

Jinými slovy - zobecněme tedy řešení z předešlého článku nejenom na osamocený objekt, ale i na vazby mezi nimi...

13.3 Agregace

Jako modelový příklad zvolme klasickou agregaci, kdy osoba má svou adresu, tj. pokud se nezabýváme historií objektů, potom můžeme namalovat takovýto diagram tříd:

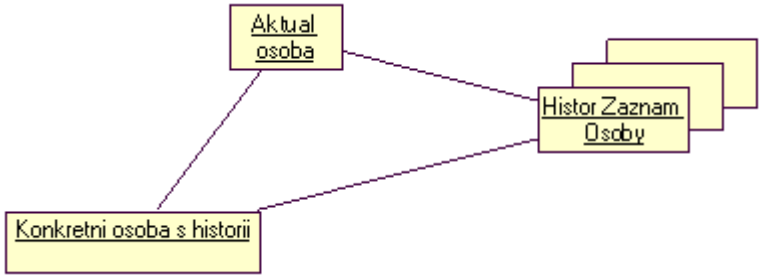
Avšak co se stane, když jak osoba, tak adresa se stanou objekty vyžadující vést svou historii? Je zřejmé, že oba dva se stanou potomky ve vztahu gen-spec vůči třídě historii a že vzniknou historické záznamy u každého z nich - jak u osoby, tak u adresy. Znamená to, že počítáme s tím, že osoba může změnit své iniciály a také se může přestěhovat. Stále však bude řeč o jedné osobě (paní se vdá, přejmenuje a ještě se přestěhuje a je to stále tatáž paní).

Chceme, aby osoba s historií vybavená adresou s historií znala nejenom svou historii, ale také aby byla možnost získat historické adresy. Pro samostatné objekty je situace zřejmá (viz předešlý článek),

jak však budou tyto objekty spolu propojeny a jak budou v otázkách historie spolupracovat mezi sebou?

Ukažme si nejprve, jak celý model našeho analytického návrhu objektů s historií funguje. K vysvětlení použijeme tzv. objektový diagram, který lépe osvětlí základní myšlenky a je v podobných situacích pro vysvětlování velmi vhodný.

Vztah mezi objekty (nikoliv třídami) ukazuje následující diagram:



Na diagramu je znázorněna jedna “jakási” instance konkrétní osoby s historii (tato instance vznikla z třídy `CHistorOsoba`). Díky gen-spec tento objekt drží historické záznamy osoby. Množina - kolekce objektů je znázorněna symbolicky třemi objekty nad sebou. Objekt také obsahuje jednu instanci aktuálního záznamu Aktual osoba, což je asociací jedna instance ze seznamu historických záznamů osoby.

V tomto našem analytickém modelu má objekt Aktual osoba význam nikoliv pouze aktuální osoby pro dnešek, ale obecně pro jakékoliv datum, které zvolíme. Tedy jinak řečeno chápeme význam aktual osoby jako aktuální pro nějaké zadané datum. Význam Aktual osoby je zřejmý při nastavování osoby na daný historický moment. Objekt Konkrétní osoby s historií nechť má metodu (opět díky gen-spec od Historie), kterou nazvěme například

```
SetAktualForDate (aDate as Date)
```

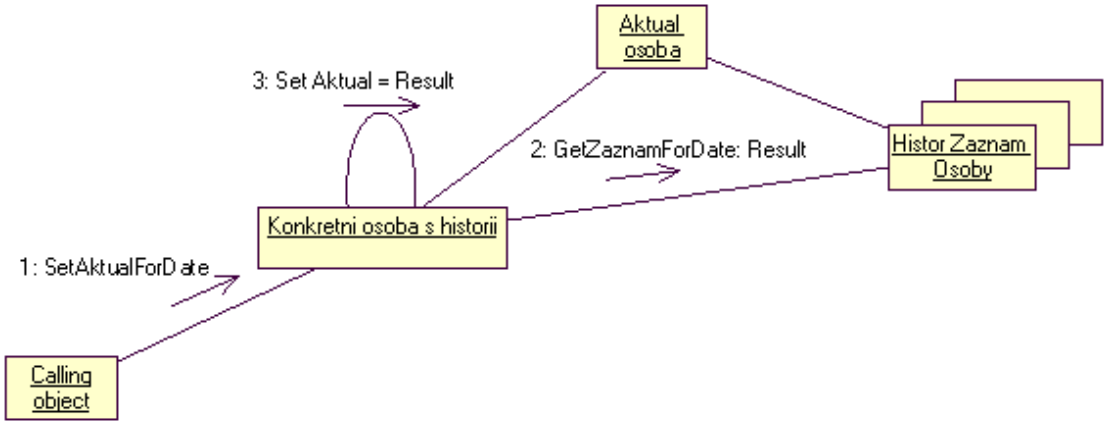
Metoda nastaví osobu do stavu k danému datumu.

Tato metoda provede v objektu následující:

1. Převezme jako vstupní parametr datum `aDate`
2. Požádá kolekci historických záznamů (v tomto případě historické záznamy osoby) o vydání záznamu s intervalem “od do” obsahující dané datum (záznam aktuální pro požadované datum)
3. Dosadí tuto instanci do Aktual osoby.

Poznámka: není zde řešena logická větev, pokud se nenajde záznam. Je otázkou dohody, jak se tento chybový stav uživatel objektu doví (je možné property, návratová hodnota error apod.)

V collaboration diagramu bychom mohli tento diagram namalovat takto:

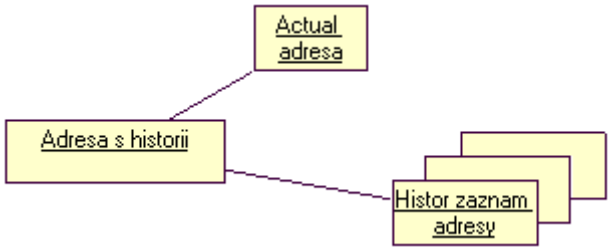


První zpráva jde od vnějšího uživatele objektu Konkrétní osoba s historií a nazývá se `SetAktualForDate`. Druhá zpráva - Konkrétní osoba s historií zavolá svou kolekci záznamů - pošle zprávu `GetZaznamForDate` (vstup je datum), která vrací `Result` - jeden ze záznamů, platný pro vstupní datum. V třetím kroku si Konkrétní osoba s historií dosadí tento výsledek do Aktual Osoby.

Poznámka: Nastavení Aktual osoby na aktuální "dnes" je zvláštním případem pro `aDate = Now`.

Protože konkrétní osoba s historií má vztah gen-spec vůči osobě, můžeme se na ni obrátit jako na osobu (například má interface a v něm rč, jméno, příjmení). Uvnitř se však všechny požadavky delegují - přesměrují na Aktual osobu. Například property rodné číslo, které Konkretní osoba s historií má, nemá svůj obraz uvnitř objektu jako vnitřní member `mRC`, ale deleguje se na objekt Aktual osoba. Uživatel objektu tedy může nastavit Konkrétní osobu na určité datum pomocí metody `SetAktualForDate` a poté s ní pracuje jako s osobou v tomto daném nastaveném čase.

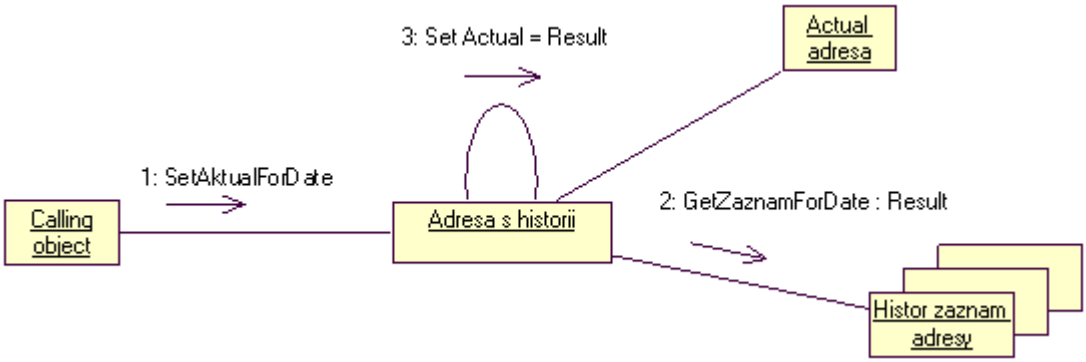
Protože adresa s historií je ze "stejné rodiny hisotrie", situace je pro ji samu o sobě úplně stejná:



a podobně pro zaslání zprávy

```
SetAktualForDate (aDate as Date)
```

která nastaví aktuální adresu

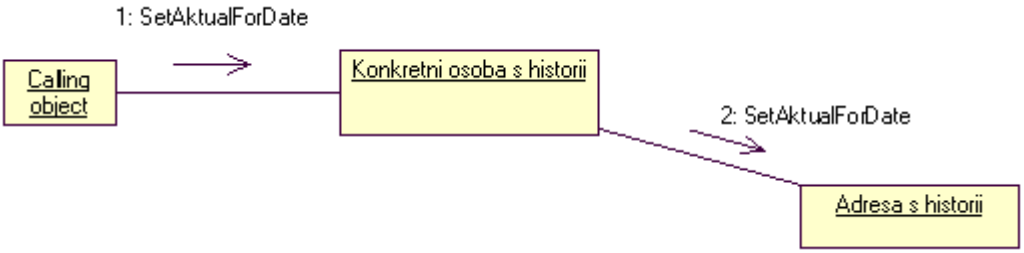


Podobnost obou obrázků není náhodná, je způsobena právě zavedením abstraktnější úrovně v gen-spec (viz předešlý článek).

Oba případy jsou tedy zřejmé samy o sobě, pokud stojí samostatně. Jak je tomu v případě, že osoba má adresu?

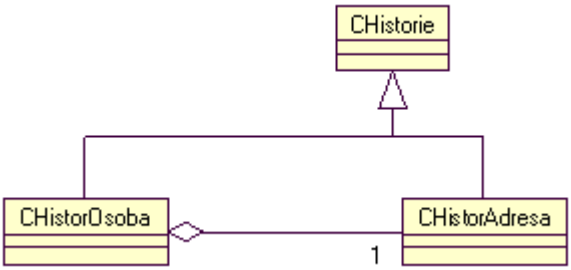
Nyní se již nabízí řešení, jak spojit oba diagramy dohromady:

Objekt konkrétní historické osoby bude obsahovat svou adresu s historii a pokud on dostane zprávu `SetAktualForDate`, potom tuto zprávu pošle také svému podřízenému objektu adresy s historií:



Sama osoba se tedy nastaví do odpovídajícího datumu a současně i adresa se také nastaví do daného datumu. Pro naše řešení je toto dostačující závěr. Protože i adresa s historií se chová jako adresa, může s ní osoba pracovat jako a s nastavenou adresou.

Z uvedeného modelu vyplývá také návrh v diagramu tříd: Objekt s historií obsahuje druhý objekt s historií (s tím, že je nutno na něj delegovat požadavek pro aktuální nastavení záznamů, což nám statický obraz modelu tříd neposkytne):

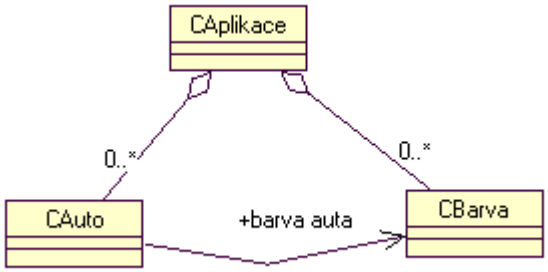


Tedy problém agregace není složitý. A co běžná asociace?

13.4 Běžná asociace

Jako klasický příklad asociace si zvolme následující:

Existují evidovaná auta, existuje číselník barev a každé auto má jednu z barev tohoto číselníku. V diagramu tříd bychom tento model namalovali takto:



Z uvedeného diagramu vyplývá, že v systému se budou nacházet dva seznamy - seznam aut a seznam barev. Každé auto si “ukazuje” na jednu již existující barvu. Znamená to, že auto tuto barvu nevlastní, ale dostane ji a dosadí si ji jako objekt na své určité místo - “moje barva” (rozdíl oproti agregaci, kdy majitel nechává rodit a ničit svou část sám).

K dosazení objektu barvy do objektu auta může dojít například v takovémto scénáři:

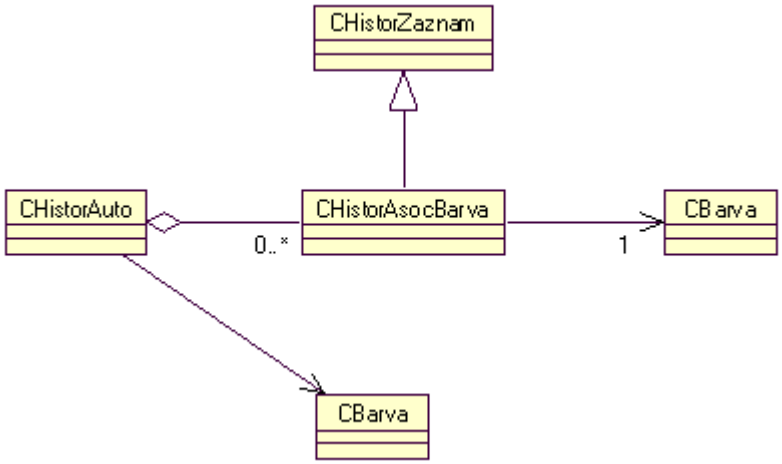
1. Obsluha má zvoleno dané auto
2. Obsluze se zobrazí seznam barev
3. Obsluha vybere barvu
4. Daná barva se dosadí do zvoleného auta.

Od této chvíle si dané auto ukazuje na danou barvu.

A nyní se vynořuje otázka - a co otázka historie? Co to vlastně znamená, “držet historii barev auta”?

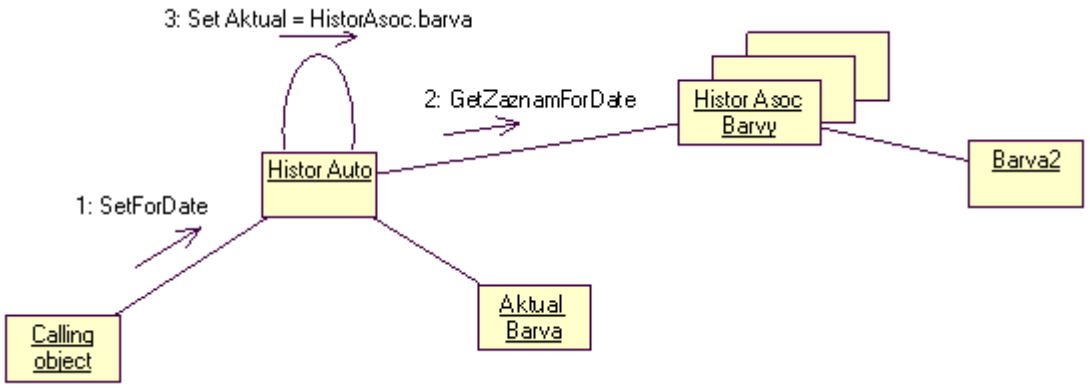
Auto si ukazuje na určitou barvu, tj. můžeme si to představit tak, že v daném místě auta je dosazen určitý již existující objekt ze seznamu barev. V určitém okamžiku se však vezme jiný objekt barvy ze seznamu objektů barev a “přepíše se” vazba na již existující objekt na daném místě a auto si ukazuje na jiný objekt barvu (jako když se přepíše ukazatel apod.). Link (vazba mezi objekty) se tak přepíše a historická informace zmizí.

I když je situace odlišná, než jaká byla u agregace, uvedený popis co se vlastně odehraje, nabízí řešení - musí existovat kolekce linků, tj. musí existovat něco, co drží linky (ukazuje na barvy) a přitom udává odkdy dokdy tento link platil. Zavedeme tedy “mezikus” držící asociaci:



Mezikus - objekt z třídy CHistorAsocBarva je potomkem v gen-spec historického záznamu a proto má také od - do interval. Tento mezikus převzal asociaci na sebe a ukazuje si na odpovídající barvu. Sám nese informaci odkdy dokdy tato asociace na barvu platila.

Princip práce s tímto modelem je nyní už stejný, jako u agregace. Ukažme si na to na odpovídajícím collaboration diagramu.

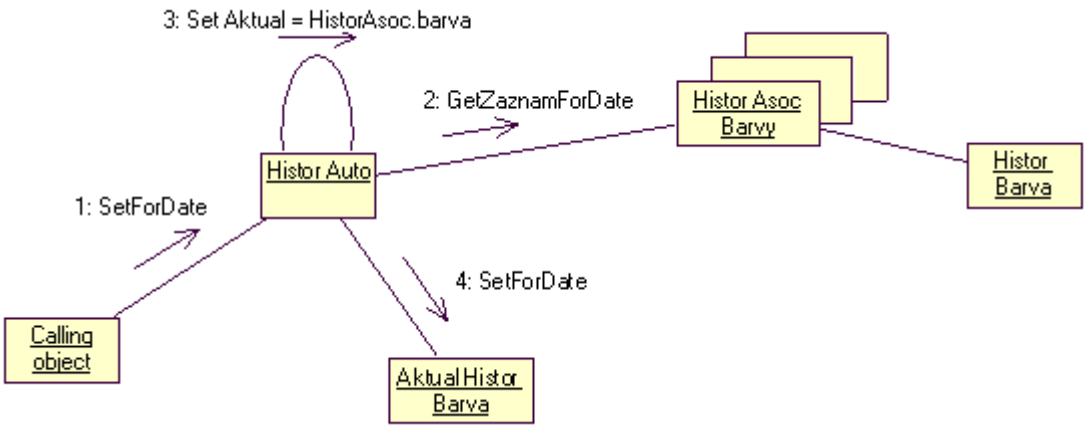


Objekt Histor Auto když dostane požadavek, aby se nastavil na určité datum, požádá záznamy asociací o vydání toho záznamu s asociací, který odpovídá zadanému datumu. Z tohoto záznamu převezme link na barvu a dosadí ji do aktual barvy.

13.4.1. Asociace na objekty s historií

A nyní zbývá řešit poslední kombinaci - co když i barva je historické povahy? Co když se číselníky mění a tady názvy a kódy barev také? A my chceme tyto změny zaznamenat i s historií....

Je zřejmé, že řešení je pouze dalším malým krůčkem - mezikus asociace si nebude ukazovat na "čistou" barvu, ale na barvu s historií.:



Oproti předešlému řešení se změnilo:

1. Kam ukazuje link mezikusu, není barva, ale barva s historií (tj. tento číselník barva má odpovídající žádané vlastnosti historie)
2. Aktuální barva není typu barva, ale barva s historií.
3. Musíme provést ještě jednu synchronizaci `SetForDate` - musíme pro toto datum nastavit odpovídající barvu z historie barvy. Důležité je, že se na toto ptáme až "na konec sekvence zpráv", tj. toto nastavení je až 4. zprávou v pořadí (viz diagram). Pořadí je důležité - nejprve nalezneme, která to vlastně byla barva (bez ohledu na její historii) a potom se teprve ptáme na její historii.

14. "Plnost modelu" v OOP

14.1 Dotaz

V tomto článku budu reagovat na následující mail, který navazuje na předešlou diskusi:

```
Byla provedena analýza a připraven objektový model. Objektový model byl
odladěn. Data aplikace ukládá prostřednictvím datového objektu. Při návrhu
aplikace jsme vycházeli z předpokládaného objemu dat cca 10.000 záznamů.
```

Při startu aplikace je datový model naplněn všemi daty. Až do tohoto okamžiku vše proběhlo podle představ a vcelku jsme nanarazili na obtíže. Aplikace byla nasazena.

V praxi se ukázalo, že množství dat bude podstatně větší než se předpokládalo a počet záznamů se začal blížit 300.000. Aplikace je funkční, ale její start je neúnosně dlouhý. Je to pochopitelné, při spuštění se plní model všemi daty.

Řekli jsme si, že tedy musíme aplikaci optimalizovat, uživatel všechna data najednou nepotřebuje. A tady jsme narazili. Celý model je postaven tak, že jsou všechna data k dispozici a během činnosti aplikace s datovou vrstvou komunikuje pouze jednosměrně - ukládají se nově pořízená data. Pokud uživatel na data aplikuje výběrová kritéria, mělo by dojít k vytěžení odpovídajících dat a jejich zobrazení, žádný z objektů však není vybaven metodami, které by toto umožňovaly. Úpravy znamenají, že se musíme vrátit na začátek a udělat analýzu který objekt bude vybaven potřebnými metodami pro přístup k datům atd. Příčina je zřejmá, od aplikace se chce něco jiného (alespoň co se týče množství) než na co byla navržena. V praxi se velmi často setkávám s tím, že požadavky na aplikaci se během programování ze strany zákazníka mění, a je nutno na tyto změny reagovat.

Jakým způsobem přistupujete k analýze Vy? Děláte na začátku objektový model v "ideální" podobě a potom řešíte "úzká" místa realizované aplikace i za cenu dost hrubých zásahů do modelu, nebo již při analýze máte na paměti optimalizaci modelu?

Tomáš P.

14.2 Odpověď

Problém, který je popsán v mailu, odpovídá již popsanému problému v jednom z článků této e-knihy, konkrétně v ”Úskalí přechodu na n-vrstvovou architekturu”. Hlavním problémem je použití “plného modelu”, který se musí na začátku spuštění aplikace naplnit a teprve poté používat. Je zřejmé, že takováto zjednodušená představa může narazit na technologické problémy, jinak přesně a výstižně popsané v mailu.

V prvé řadě musím odpovědět na jeden závažný dotaz v mailu: Analýza jako taková je nezávislá na problematice popsané v mailu. To, co popisujete jako problém, je již analyticky nezávislé. Takže situace není až tak kritická, jak se na první pohled zdá.

Vysvětlíme tuto skutečnost blíže.

Ve Vašem mailu se nezmiňujete, zda se jedná o řešení pomocí komponent, nebo zda se jedná o skupinu EXE souborů (tzv. samostatné monolity). V každém případě, ať už se jedná o použití komponentní technologie anebo nikoliv, lze možná řešení z hlediska návrhu rozdělit na dvě základní skupiny:

1. Dochází ke sdílení “živých” objektů od různých klientů v aplikaci
2. Nedochází ke sdílení “živých” objektů od různých klientů v aplikaci

Nejprve si tedy určete, zda vaše aplikace je navržena jako bod 1. nebo jako bod 2 (podle toho se některé kroky změní). Pokud je vaše aplikace napsána ve VB, tak jedinou možností, jak zavést sdílení objektů, je použít ActiveX EXE komponentu sdílenou mezi klienty a to buď pomocí DCOMu anebo pomocí ASP, kde zavedete tuto komponentu ve viditelnosti APLICATION SCOPE viditelnou všemi SESSION (což nedoporučuji vzhledem k serializaci požadavků volání komponenty a tedy dokud neobslouží komponenta jedno volání klienta, druhý se k ní nedostane).

Pokud je Vaše aplikace navržena pomocí ActiveX DLL (např. pod MTS a COM+) anebo jako pouze skupina samostatných EXE souborů, potom se živé objekty v aplikaci od různých klientů nesdílejí. V těchto případech každý klient má “svůj živý model”, do kterého druhý klient nevidí. Jeden živý objekt je používán pouze jednou aplikací (klientem).

Připomeňme, že v případě sdílené komponenty se tato komponenta chová jako “samostatně stojící objektový server” podobně jako databázový server. Řešení optimalizace takovéhoto serveru není vůbec jednoduché (například zrovna nyní studuji zajímavou práci na toto téma, určitě se k tomuto problému dostaneme později). Navíc musíte řešit i problémy s transakcemi, uzamykáním atd.

Jednodušší situace nastává pro optimalizaci v případě nesdílených komponent. V tom případě lze použít metodu “optimalizace v kontextu daného scénáře”. Zde totiž každý scénář použití objektu klientem vymezuje striktně toto použití objektu a nemusí se brát v úvahu v tomto scénáři použití tohoto objektu v jiném scénáři. Důsledkem je možnost provést optimalizační úpravu scénáře použití objektu. Samotná podstata zapsaná v analýze se však nemění a je třeba doplnit (nikoliv změnit!) funkcionalitu objektu, většinou dané kolekce.

Uveďme klasický příklad. V analytickém dokumentu v reportu z Use Case modelu se dočteme:

Název Use Case: “Editace firmy”

Popis: Obsluze se zobrazí seznam firem (IČO a název). Obsluha vybere firmu. Zobrazí se údaje firmy pro editaci (následuje popis údajů pro editaci...). Obsluha zvolí OK a změněné údaje se z formuláře dosadí do vybrané firmy.

Pokud toto řešíme “klasickým scénářem”, tak na počátku aplikace

1. Necháme provést načtení dat
2. Naplníme kolekci firem firmami s vytěženými daty

   Uvedená kolekce “stojí jako server” pro použití. Poté pokračujeme v tomto scénáři:
3. Přesypeme property itemů - firem do GUI
4. Necháme obsluhu vybrat firmu
5. Provedeme editaci firmy
6. Zavoláme firma.update

Je pochopitelné, že takovýto scénář může narazit na problémy popsané v mailu. Úzké místo spočívá v naplnění kolekce na začátku aplikace. Tu však obsluha potřebuje pouze pro zobrazení. Pokud tedy “zobrazení kolekce” vyřešíme jinak (náhradní optimalizační scénář), můžeme poté pracovat pouze s jednou instancí firmy. Tuto úvahu si můžeme dovolit, protože více firem v celém daném scénáři editace netřeba. Optimalizovaný scénář potom vypadá takto:

Kolekce se na začátku neplní. Naučíme ji však vydávat „data“ ICO a Název. Toto kolekce umí “něco navíc”, protože to potřebuje optimalizační scénář. Pomocí tohoto umění se nakonec zobrazí řada stringů ICO a Název.

Obsluha vybere ICO a “jinak prázdná kolekce bez itemů” převezme toto ICO, vytvoří pouze jeden objekt firmy, naplní jej daty (ICO je unikát) a vydá jej ven pomocí této nové metody. Poté se pokračuje dále - po editaci formuláře a převzetí hodnot do firmy zavoláme firma.update.

Samozřejmě takovýto postup se činí v rámci scénáře, kde se pracuje pouze s jedním objektem firmy. Podobně můžeme řešit problematiku vazby objektů z číselníků - ve scénářích pracujících s jednou instancí můžeme tento objekt vytvořit pouze jeden (a nikoliv plnit celý číselník všemi itemy) a naplnit jej přes klíč v kolekci a vydat ven.

V těchto metodách optimalizace během daného scénáře musíme být pouze opatrní na možnou ztrátu reference mezi objekty. Naštěstí v rámci daného scénáře izolovaného klienta je tato chyba velmi dobře viditelná. Uveďme příklad: v předešlém postupu bude objekt Firma pracovat s jedním objektem Země s číselníku Zemí (objekt Země umístěn v Adrese firmy). V tom případě nám ztráta reference na Zemi nehrozí, protože se jedná pouze o jednu instanci a ztráta reference připadá v úvahu až při násobném použití objektů.

Je pochopitelné, že pokud budeme pracovat s objekty sdílenými různými klienty, potom tato možnost optimalizace jednoduše odpadá, protože musíme počítat s možností použití objektu libovolným klientem v libovolném scénáři. V tom případě pokud nechceme plnit celou kolekci, potom musíme "nějak zjišťovat", kterých sdílených objektů třeba a kterých netřeba (například počítáním referencí apod.). To samozřejmě velmi zesložiťuje tento problém.

Pokud v této optimalizační metodě potřebujeme "výběrově omezené kolekce", potom samozřejmě využijeme s výhodou omezená "plnění kolekcí". Například kolekce má property Filter odpovídající property Filter pro Recordset apod. Protože má každý klient svou vlastní kolekci, každý z nich si ji může omezit jak chce bez ohledu na jiné klienty. Podle scénáře buď kolekci naplníte anebo nemusíte ji plnit hned na začátku, ale provedete podobné optimalizační kroky jako v předešlém případě.

V případě, že používáte sdílené objekty, potom doporučuji převést systém na systém s izolovanými klienty. Můžete se přesvědčit, že tento postup není až tak bolestný, jak se na první pohled zdá: Model tříd je totiž shodný v obou případech! Na vysvětlenou, proč jsou modely tříd shodné: Když se řekne kolekce, nepředstavujte si automaticky, že se jedná o N objektů alokovaných na začátku aplikace! Kolekce je objekt jako každý jiný, a může mít tedy "různě plný obsah" podle metod, které voláte a podle stavu, v jakém se kolekce nachází. Například v této souvislosti podotkněme, že všechny objekty DB GUI apod., které zobrazují přímo hluboké tabulky a obsahy zdrojů dat (přes datasource), jako například DBGRID, pouze "imitují" svou plnost! Ony pouze velmi chytře cachují svůj obsah.

Pokud si dobře pamatuji na zmínky o analýze systému uvedeného v mailu, jedná se o řešení hierarchických struktur ve skladě. Zde se nabízí například řešení změnit scénář "celého plnění" na částečné plnění pouze daného Node (jeden aktuální Node apod.). Pokud obsluha požádá o další Node, potom teprve dochází k dalšímu plnění. Znamená to přidat metody plnění "bez rekurzivního volání podřízených prvků" a například pracovat pouze na dané úrovni. To odpovídá omezení "jedna firma" v předešlém příkladu.

14.3 A nakonec jedno doporučení...

Nenechejte se v těchto optimalizačních metodách svést na scestí strukturálního programování. V této souvislosti uvedu příklad dotazu, který byl skutečně vznesen při konzultaci v jedné firmě: A co tak provést zobrazení tak, že dostaneme z dat dané stringy a ty zobrazíme, tj. provedeme toto mimo objekty? Odpověď zní nikoliv - musíte dodržet kompetenci objektů a žádat o novou funkcionalitu danou kolekci a nikoho jiného. Například kolekce umí vydat postupně stringy pro zobrazení, umí vydat "jeden izolovaný item" apod. Nežádejte datovou vrstvu přímo. Například za kolekcí v datové vrstvě může být schován recordset, ale žádejte si funkcionalitu od kolekce a teprve z ní se převádí dotaz dovnitř do datové vrstvy. Když už chcete takovéto řešení přijmout, obalte data danou kolekcí (jinak řečeno vložte datový objekt do ní)! V opačném případě se systém rozpadne na nedefinované části a bude těžko udržitelný.

Také z toho vyplývá důležitá odpověď na otázku: Jaké změny tyto optimalizační kroky vlastně pro nás znamenají?

Odpověď: Pro scénář v daném Use case se jedná o změnu oné technologicky kritické části scénáře, například jiný scénář zobrazení apod. Pro samotné objekty business jsou to doplnění (nikoliv změny!) jejich funkcionalit, které jako nově přidané funkcionality mohou splnit požadavky těchto nových scénářů.

A nakonec nejdůležitější doporučení: Neprovádějte optimalizaci předčasně. Uvedený postup "optimalizace v kontextu daného scénáře" lze totiž vždy provést pouhým doplněním funkcionalit objektů. Navíc pokud dodržíte kompetence objektů, potom tyto změny provádíte za neustále dobře

funkčního sytému. Velmi dobře znám z praxe, jak předčasné optimalizační metody dovedly “rozvrtat” systém ještě před jeho zprovozněním.

15. Objekty, synergetika a stabilita systému

15.1 Synergetické efekty

Pro jednoduchost si můžeme vysvětlit synergetické efekty jednoduše takto:

V synergetice neplatí operace lineárního sčítání, tj. 1 plus 1 rovná se 2, protože přidání jedničky k jedničce nejsou pouze dvě jedničky, ale připočítá se také „nějak“ také možná interakce mezi nimi. Tedy součet efektů je dán jako jedna plus jedna plus nějaká interakce jedna-jedna. Klasickým příkladem je smíchání dvou různých systémů, kde prvky mezi sebou interagují - výsledkem není pouze směs, ale také průběh interakce. Například efekt smečky u šelem patří mezi synergetické efekty - dva rozzuření psi napadající jedince ve smečce nejsou pouze “dva psi”, ale také jejich vzájemná interakce vzájemného působení smečky (psi se chovají jinak než dva jedinci postavení jenom vedle sebe, jsou “dva spolu”).

Na první pohled se jedná o tak trochu teoretické úvahy, ale pro objektové programování má zamezení synergetických efektů velmi zajímavé a prakticky příznivé důsledky.

15.2 Synergetické efekty ve strukturálním programování

Jak známo, RAD zkratkou pro tzv. *“Rapid Aplication Development”*. Jedná se zvláštní druh vývoj SW aplikace, kdy pomocí poměrně dost sofistikovaných nástrojů se jednoduchým způsobem velmi rychle vyvine jednoduchá aplikace naklikáním GUI prvků na formuláře, přičemž tyto GUI prvky mají přímou vazbu do dat (většinou přes nějaký datasource). Vývojové prostředí tak provádí automaticky určité kroky, které se jinak provádějí ručně. Nevýhodou je však mnohem menší takřka minimální re-use mezi již hotovými prvky (například z projektu do projektu).

Představme si, že by nám někdo položil otázku:

Máme na výběr ze dvou možných způsobů tvorby SW aplikace. První z nich je charakterizován velmi velkou rychlostí tvorby prvků pomocí jakýchsi automatizovaných mechanismů vývojového prostředí (RAD), avšak pokud vytvoříme nějaký celek, nemůžeme jej znovu použít. Pokud se něco opakuje s malými odlišnostmi, musíme znovu tento prvek vyvinout, i když nutno podotknout, že pomocí automatu vcelku rychle. V aplikaci tedy není zaveden takřka žádný re-use, ale tato nevýhoda je vyvážena možností poměrně rychle vytvořit opakující se část aplikace. Samozřejmě, pokud se cokoliv v aplikaci dodatečně změní resp. nalezne chyba, musíme tuto chybu anebo změnu opravovat a zavádět tolikrát, kolikrát jsme (i když rychle) část aplikace znovu tvořili.

Druhý způsob spočívá v možnosti tvorby systému ve svých menších částech mnohem pracněji, protože nepoužíváme automat a musíme proto mnohem více práce provádět “ručně”. Ale zase máme k dispozici tak maximální re-use, proto žádnou práci neprovádíme nikdy dvakrát (nikdy, pokud jsme důslední). Systém je z tohoto hlediska čistý, vše je jen jednou a také je systém přehlednější, průhlednější a stabilnější, flexibilnější na opravy atd. Je zřejmé, že maximální re-use později urychlí vývoj aplikace, protože čím bohatší máme již hotovou knihovnu, tím rychleji se nám tvoří další prvky z knihovny odvozené.

Otázka zní - který z těchto dvou přístupů je lepší?

Samozřejmě neexistuje jednoznačná odpověď. Základním kriteriem jsou koncepce firmy, koncepce projektů a náklady projektů. Podívejme se na tuto otázku z hlediska nákladů:

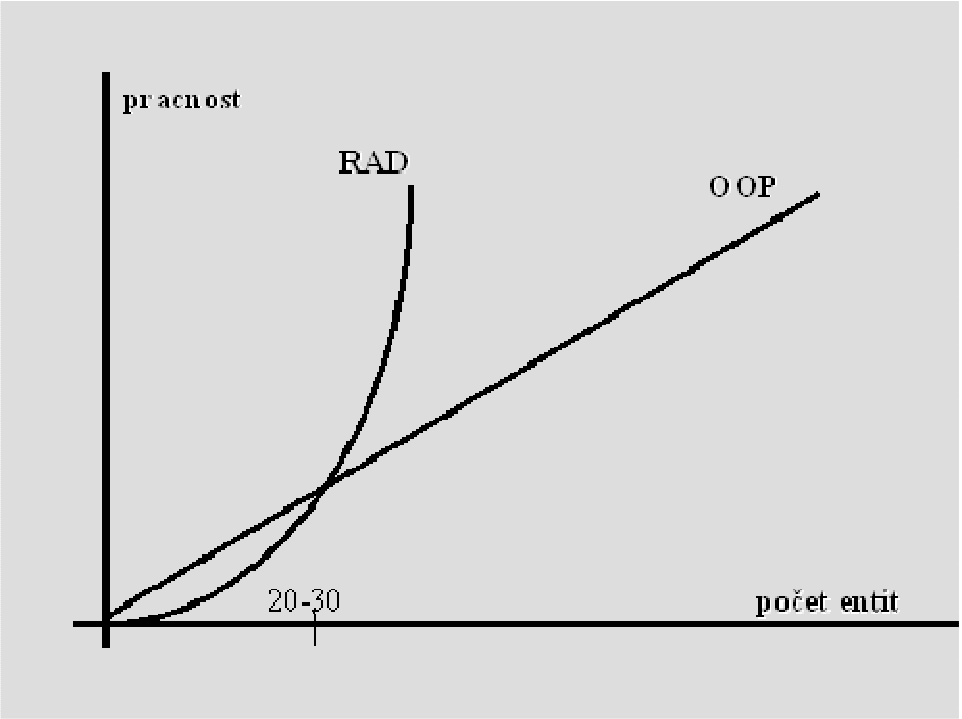
Postup pomocí RAD je výhodný tam, kde vyvíjíte jednoduché a co do počtu entit malé aplikace (například do 10-30 tabulek apod.). Jedná se tedy o aplikace a projekty firmy vystačující s malým re-use. Pokud si sami pro sebe vytvoříte malou aplikaci s pěti tabulkami a třemi formuláři, tak bude plně dostačující vytvořit tuto aplikaci pomocí RAD. Z hlediska nákladů není zanedbatelné také zaučení a

kvalifikace pracovníků - na vývoj RAD vystačí průměrný programátor, který se rychle orientuje v daném automatizovaném vývojovém prostředí.

Oproti tomu druhý (OOP) přístup je nezbytným tam, kde narůstá složitost problému. Pokud budete vyvíjet aplikaci (ať už desktopovou, na lokální síti nebo Internetu) s větším počtem entit (například 40 a více), začnete mít s použitím RAD problémy. Začne se projevovat synergetický efekt nárůstu entit. Z hlediska RAD platí zajímavé synergetické pravidlo: S větším počtem entit neroste pracnost systému lineárně, ale pracnost roste mnohem rychleji, protože entity nejsou zapouzdřeny a důsledně odděleny od sebe, ale naopak neustále více a více provázány. Znamená to, že nemůžete u RAD použít jednoduchou metodu lineárního nárůstu entit, tj.přidávání stále více entit bez následných problémů. Pokud máte v systému 20 tabulek a rozšíříte je na 30, nepřibude vám v RAD 1/3 práce, ale mnohem více. Dokonce od určitého momentu synergetický efekt vztahu mezi entitami přeroste u stovek tabulek do velmi silných rozměrů. Od té chvíle je velmi obtížné systém udržovat v stabilním chodu (například představme si dobu celého jednoho měsíce chodu systému po změně v něm bez dalších zásahů, což by mělo být běžné)

Nutno podotknout, že nevýhodou přístupu pomocí OOP (které odstraňuje tento nežádoucí efekt), je určitá “pomalost” při začátku vývoje. Další brzdou se může stát průměrnost programátorů, kteří z vlastní neznalosti nejsou schopni využít všech výhod re-use a tedy ztrácejí výhody tohoto přístupu. Právě velmi častým jádrem diskusí o výhodách OOP jsou neznalosti, které jsou způsobeny nevyužitím výhod re-use v OOP. To nakonec bohužel vede k vyšší pracnosti v OOP než u RAD a tedy pro tyto pracovníky zůstane RAD výhodnější.

Uvedený efekt synergetického zvyšování pracnosti aplikace u strukturálního programování lze vyjádřit pomocí grafu:



Tento graf jsem si ověřil v praxi. Linearita nárůstu pracnosti v OOP je dána izolací jednotlivých pojmů od sebe. Znamená to, že nový pojem vzniká jako samostatný pojem izolovaný od ostatních a tedy

pracnost roste takřka lineárně. V OOP neexistují “velké” a “malé” objekty. Díky zapouzdření existují pouze “stejné” objekty obsahující další objekty - a to je vyjádřením linearity v grafu.

Jako příklad - představme si, že řešíte výstupní sestavy a přehledy ve vašem systému jako výstupy pro manažera a rozhodnete se použít výstup do Excelu. Zamontujete si objekt Excelu do své aplikace sestav. Sama vaše aplikace není příliš složitá, pouze naplňuje objekt Excelu a ovládá jej. Navenek však vypadá vaše aplikace jako “veliká a složitá”, protože toho umí moc (díky Excelu). Ale vámi přidaná skutečná hodnota pracnosti není příliš rozsáhlá. Udělá to za vás Excel. Ale sám Excel uvnitř také není nějak složitý, pokud bychom jej rozbalili, opět bychom zjistili, že obsahuje několik málo objektů a tato první vrstva rozbalení Excelu je jednoduchá. Rozbalíme další objekty a tak dále - žádná vrstva není složitá a to je podstatou oné linearity na grafu.

A teď si představme, že tyto vrstvy odstraníme a uvidíme “vše odshora až dolů až do nehlubších střev Excelu a to naráz”. Jinak řečeno odstraníme zapouzdření na všech úrovních objektů. Dovedete si představit tu nepřehlednost?

16. O jednom záludném úskalí při přechodu firmy na n-vrstvovou architekturu

16.1 Dvojí chápání přechodu na technologii OOP

Přechod na OOP a komponentní technologii se děje ve dvou základních rovinách.

Na straně jedné se jedná o problematiku zvládání této technologie samotnými jedinci, kteří se vzdělávají, kteří se učí a s menším nebo větším úspěchem tuto technologii vstřebávají.

Na straně druhé hovoříme o samotném přechodu firmy na technologii OOP a komponentní technologii. V tom případě se již nejedná pouze o zúžený problém něčemu porozumět a umět něco používat, ale o řízený firemní projekt se stanoveným cílem, určenými zdroji a vytvořeným harmonogramem. Problematika přechodu na novou technologii je pro firmu totiž mnohem širší, než pouhé “zvládnutí a pochopení” této technologie. Samozřejmě, vyžaduje se, aby pracovníci tuto technologii zvládli jak teoreticky, tak prakticky, to je však podmínka nutná a nikoliv postačující.

Přechod firmy na novou technologii musí být řízeným a plánovaným procesem a nikoliv pouze jakési ad-hoc vyhlášení přechodu na novou technologii. I kdyby se pracovníkům během krátké doby podařilo jak teoreticky, tak prakticky zvládnout novou technologii OOP (což není příliš pravděpodobné), bez řízeného přechodu na novou technologii skončí tento proces naprostým chaosem a destabilizací projektů.

Samotný proces přechodu SW firem na OOP má svoje přesně dané zásady. Jejich zanedbání nebo dokonce nedodržení může vést k nezvládnutí vytýčeného cíle a firma zůstane na úrovni technologie na začátku procesu bez zvládnutí OOP a komponent.

V tomto článku se seznámíme s jednou z hlavních zásad přechodu na OOP a komponentní technologii a k čemu vede její nedodržení.

16.2 Postupné kroky

Pokud stoupáme po schodech, je bezpečné mít druhou nohu vždy na stupni nižším. Podobně je tomu v SW firmách při změnách technologie. Každá změna technologie, i když k lepšímu, je pro SW firmu velkým rizikem. Platí jednoduché varování:

každá změna je nebezpečné riziko !

Pokud se vedení SW firmy rozhodne k přechodu na jiné technologie, mělo by uvážit toto varování a naplánovat změny ve firmě jako postupné a nikoliv jako revoluční metodou čínských skoků. Líbivé projekty rychlého přechodu na nové technologie v řádově dnech a týdnech a bez plánu jsou zaručenou cestou do pekel. Většinou se ve velkém nadšení z moderních technologií zamění cíl s přáním a výsledkem je nakonec nezvládnutý proces.

16.3 V čem tkví záludnost zavádění moderních technologií

O dvou rovinách přechodu na OOP - tj. přechod na OOP u jednotlivce a u firmy - bylo pojednáno v úvodu tohoto článku. Záludnost moderních technologií spočívá v tom, že při prostudování těchto technologií a jejich náležitém pochopení se jedinec může dostat do situace, kdy se mu tato technologie již jeví jako zvládnutelná a použitelná, protože jí rozumí a chápe ji, dokonce vidí všechny její výhody. Avšak neuvědomuje si v té chvíli, že toto není postačující podmínka pro to, aby se tato technologie dala ve firmě používat. Ve chvíli tohoto “osvícení” se lehce propadne pocitu, že se jedná pouze o technologický problém. Není to pravda. Jedná se o celý problém změn ve firmě a v tom je ta záludnost.

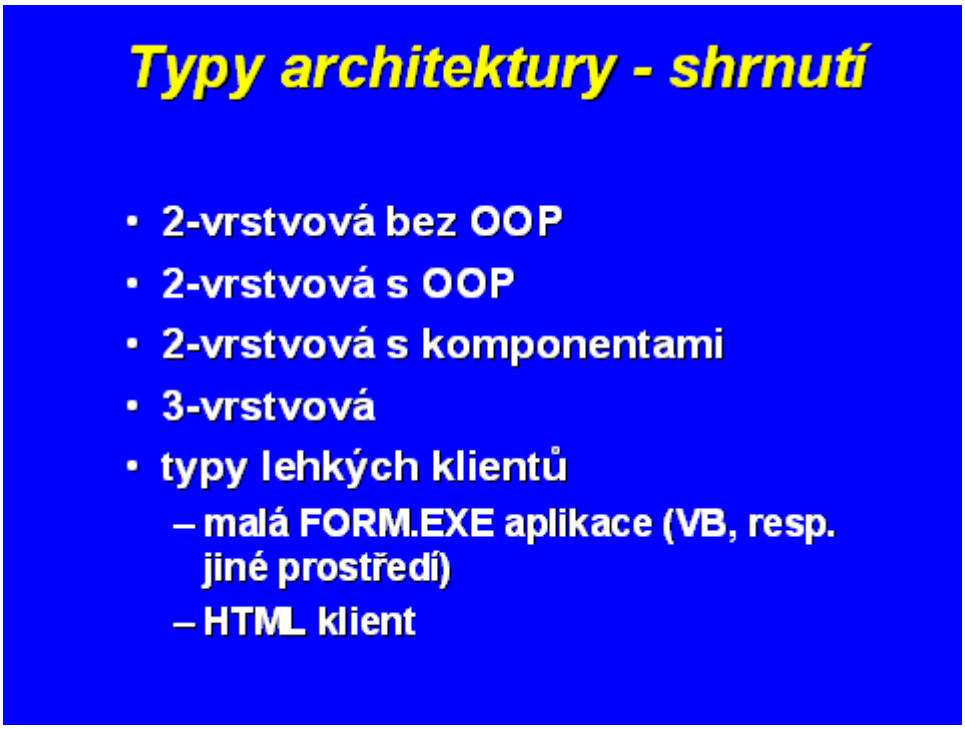
16.3.1. Odkud a kam se jde: Od 2-vrstvové přes OOP až k n-vrstvové architektuře

Přibližně 90% firem u nás pracuje v tzv. 2-vrstvové architektuře. Znamená to, že podstatou jejich systémů jsou tzv. databázové aplikace. Klienty databáze představují spustitelné programy, které nejsou napsány ani objektově a ani komponentně, které pomocí SQL těží data, zobrazují a opět pomocí SQL ukládají.

Základním informačním prvkem vývoje systému je tzv. ERD diagram.

Na druhém konci pomyslného žebříku (schodů), kam chceme dospět, je n-vrstvová architektura.

Pokud si shrneme všechny stupně možné architektury, dostaneme následující žebříček



16.3.2. Doporučení pro postupné změny architektury v ideálním případě

Pro ideální případ (který jak víme nikdy nenastává) by bylo vhodné doporučit následující postup změn, který kopíruje předcházející stupně architektur. Posloupnost takto navržených změn totiž není násilnou cestou a je maximálně “splavnou”:

- Fáze 0 - systém je napsán dvouvrstvově a neobjektově
- Fáze I - přechod na OOP v klientech. Cílem je vytvořit klienty databáze pomocí OOP (architektura je stejná jako ve fázi nula). Vytvářejí se modely tříd apod., ale programy nejsou napsány pomocí komponent
- Fáze II - přechod na komponenty, avšak pouze na lokále. Na strojích, kde jsou klienti, se program již netvoří pouze spustitelnými soubory, ale skládá se z komponent. Tyto komponenty se však na různých strojích opakují.
- Fáze III - přechod na n-vrstvovou architekturu - komponenty volané přes sít resp. Internet / Intranet. Komponenty se již v systému neopakují

Ne vždy nám však život dovolí takový luxus a plánovat si postupné změny podle našeho přání. V mnoha případech by pomalost přechodu sice znamenala stabilitu, ale také ztrátu zakázky.

Jak postupovat v tomto případě?

16.4 Doporučení pro přímý přechod na třívrstvovou architekturu při zachování postupných změn

Obchodní situace vyžaduje rychle zvládnout fázi nazvanou v předešlé kapitole jako Fáze III. Samozřejmě přistupujeme k problému s vědomím rizika “přeskoku” dvou fází, které měly zabezpečit stabilitu projektu. Ale nedá se, vyžaduje se “zrychlený přesun”.

I zde se můžeme dopustit jedné zásadní chyby, a o této chybě je tento článek. Tato chyba “velkého skoku” souvisí s typem komponent, které zvolíme pro naše řešení. Nyní si tento problém náležitě vysvětlíme.

16.4.1. Instance komponent sdílené a instance s izolovanými klienty

Pokud zavedete komponentu, ať už na síti, nebo na lokále, máte z hlediska jejího postavení vůči jejím klientům dvě základní možnosti, jak se instance této komponenty bude chovat (samozřejmě tuto volbu provádíte před jejím vytvořením ve fázi designu).

1. Instance komponenty je sdílená klienty
2. Instance komponenty není sdílená klienty a pro každého klienta existuje nová instance komponenty. Dále takovou komponentu budeme nazývat jako komponentu s izolovanými klienty

Zde je třeba se trochu se zastavit, protože jsem se při konzultacích setkal s mnoha nepochopeními právě v této oblasti.

V prvé řadě všimněme si, že rozlišujeme instanci komponenty a komponentu. (Většinou se toto rozlišení neprovádí a směšují se oba pojmy). Pod instancí komponenty máme na mysli jednu existující žijící strukturu binárního útvaru - binárního objektu. Obecně z jedné komponenty můžeme vytvořit N

instancí komponenty (pokud to model komponenty dovoluje). Pod samotnou komponentou naopak chápeme to, co máme v systému nainstalováno jako možnost tvořit instance z této komponenty. Vztah mezi instancí komponenty a komponentou je podobný, jako je mezi objektem a jeho třídou.

Samozřejmě oba typy komponent - jak sdílená, tak s izolovanými klienty - mají diametrálně odlišné chování už z hlediska analytického.

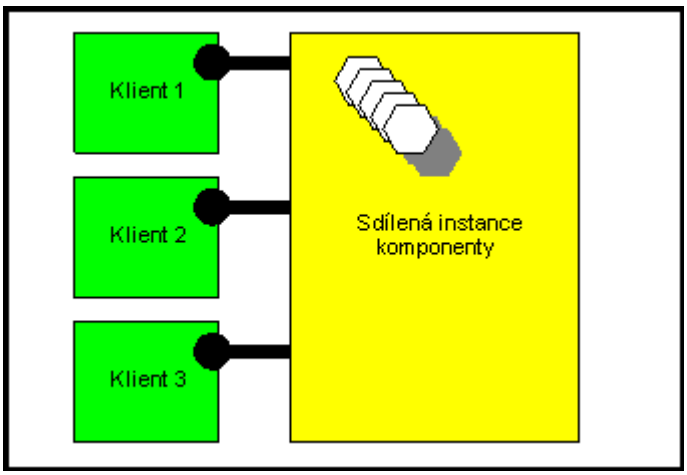
16.4.2. Objektový sdílený server

Sdílená instance komponenta funguje jako skutečný objektový server. “Stojí” v systému jako objektový stroj s objekty připravenými k použití a každý nový klient se může k tomuto stroji připojit a používat jej. Přitom tento server může být instalován na síti a sdílen klienty z různých strojů. Objektové struktury v instanci komponenty se jednou naplněné stávají všeobecně sdílenými a to “přímo a naživo”. Klienti mají přístup k objektům, které již někde v systému žijí a mohou tak pro klienty přímo pracovat. Pojem aplikační server má takto velmi konkrétní význam - spousta věcí z databáze odpadá, například někde již existuje seznam barev, seznam aut, stačí šáhnout do seznamu objektů...

16.4.3. Objektové izolované knihovny

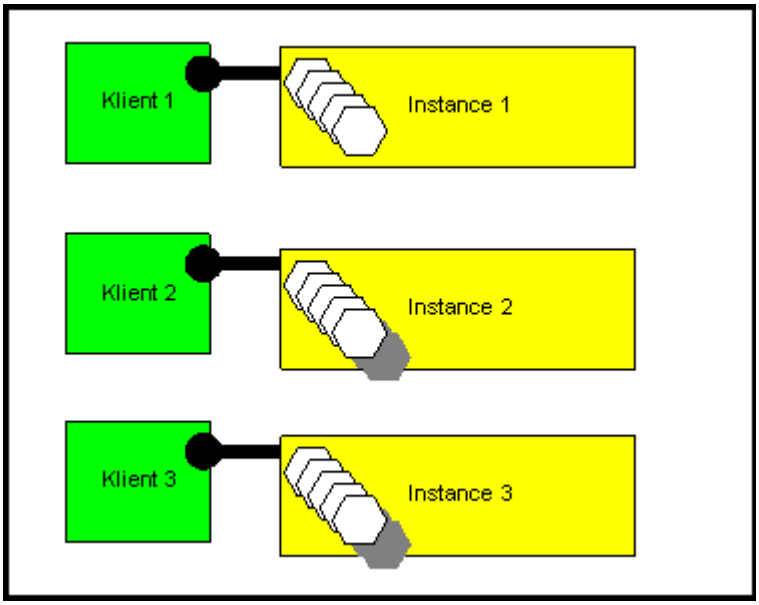
Druhý typ komponenty, tj. komponenta s izolovanými klienty, je velmi odlišná. Každý klient, který chce používat instanci komponenty, si vybudí novou “svou a pouze svou izolovanou” instanci ve svém vlastní viditelnosti a nikde jinde. Instance se nechová jako sdílený stroj, ale spíše jako přilinkovaná knihovna ke svému klientovi. Z podstaty věci je vyloučeno, aby tato instance byla viděna jiným klientem. Sama komponenta (nikoliv instance) může být nainstalována v systému “jednou” a je tedy sdílená, nikoliv však její instance. Sdílení tohoto druhu znamená re-use kódu v tom smyslu, že nemusíme aplikaci (komponentu) distribuovat na klienty.

Následující obrázek ukazuje sdílenou instanci komponenty:



Jak Klient 1, tak Klient 2, tak Klient 3 “vidí” tutéž objektovou strukturu znázorněnou na obrázku symbolicky několika šestihrany (objekty).

U komponenty s izolovanými klienty je situace odlišná, každý klient vidí jenom tu “svou instanci komponenty”:



Je zřejmé, že u druhého typu komponent dochází ke sdílení informace mezi klienty až na úrovni databáze.

16.5 ActiveX EXE, ActiveX DLL a jedna zavádějící úvaha

Jak jsem si všiml v konzultacích, v souvislosti s komponentami typu ActiveX EXE a ActiveX DLL (viz skripta COM a DOM) se mnohdy provádí jedna zavádějící úvaha.

V našem dělení komponent na dva typy (sdílená a s izolovanými klienty) jsme se zajímali pouze o jednu vlastnost - jaký je vztah komponent vůči klientům a žádná jiná vlastnost nás nezajímá.

Ze základů COM technologie je patrné, že můžeme komponentu ActiveX EXE považovat podle našeho dělení za komponentu sdílenou. Na druhé straně ActiveX DLL podle téhož dělení považujeme za komponentu s izolovanými klienty. Tato úvaha je jasná a zřejmá.

Mnohdy však začátečníci tuto úvahu otáčejí a tvrdí, že chci-li vytvořit sdílenou komponentu, musím vytvořit ActiveX EXE komponentu a pokud chci vytvořit komponentu s izolovanými klienty, musím vytvořit ActiveX DLL. Chyba této úvahy spočívá v tom, že ne každá komponenta, která je sdílená, musí být ActiveX EXE a podobně pro ActiveX DLL - ne každá komponenta s izolovanými klienty musí být ActiveX DLL. Tato úvaha je většinou důsledkem omezeného rozhledu Basicářů, kteří nevědí, že pojem komponenta je obecnější, než pouze ActiveX komponenta, která je velkým omezením obecnější a mnohem složitější komponenty.

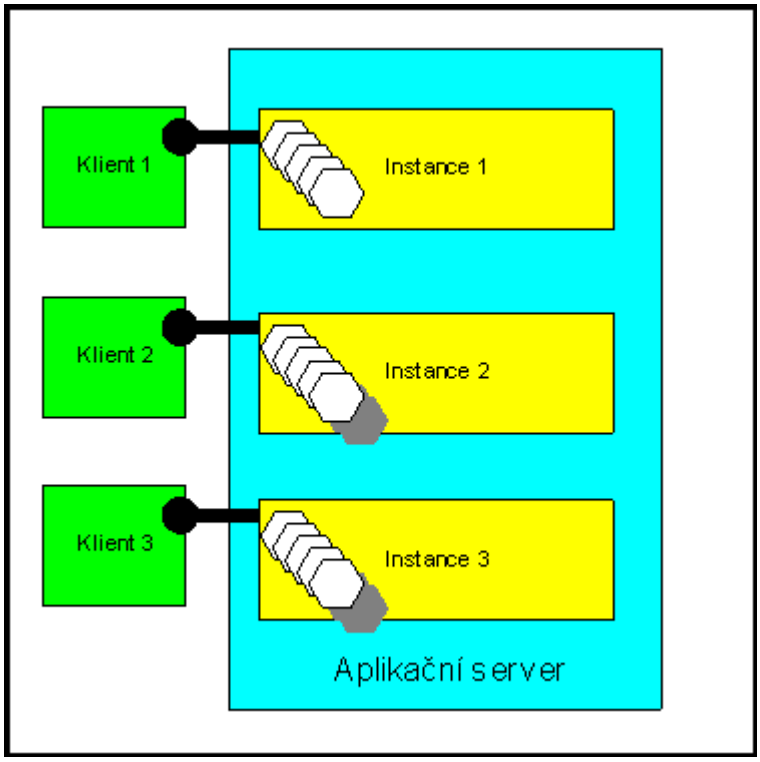
Takže platí následující tvrzení - obecně sdílená komponenta nemusí být totožná s ActiveX EXE komponentou, ale ActiveX EXE je sdílenou komponentou.

Ve VB máme možnost tvořit pouze ActiveX komponenty a z toho titulu pro Basicáře oba pojmy splývají... V C++ při tvorbě COM komponentu (nepoměrně složitěji) bychom mohli vytvořit komponentu “šitou na míru” podle potřeby.

16.6 Komponenta s izolovanými klienty a n-vrstvová architektura

U modelu komponenty s izolovanými klienty jsem se setkal s mylným názorem, že v tomto případě - tj. u komponenty s izolovanými klienty - se nejedná o třívrstvovou architekturu. Dotyční vycházeli z mylného přesvědčení, že existence business vrstvy znamená sdílený objektový server, tj. komponentu sdílenou. Není to pravda. Pro třívrstvovou architekturu je rozhodující, zda klienti musí nebo nemusí provádět „redistribuci aplikace“ a zda dochází k využití funkcionality na vzdáleném stroji. Z hlediska sdílení je pro n-vrstvovou architekturu důležité sdílení komponent a nikoliv sdílení instancí komponent (rozdíl viz předešlé kapitoly).

Podívejme se na obrázek komponent s izolovanými klienty znovu:



ale trochu jinak - barevně jsme označili Aplikační server jako stroj, na kterém se budí instance komponent. I takováto architektura odpovídá třívrstvové architektuře.

16.7 Při přechodu na OOP - sdílení komponent nebo ne?

Vytvořit aplikační server jako sdílenou komponentu, tj. objektový server, je velmi lákavé, ale při přechodu od 2-vrstvové architektury k 3-vrstvové ***se může jednat o velmi nebezpečný krok***.

Pokud zvolíte sdílenou komponentu, potom musíte očekávat tyto úkoly, které jste dosud neřešili:

- Řešíte zamykání objektů přímo v aplikační úrovni. Protože klienti instance komponenty přistupují k této komponentě kdykoliv pro editaci a změny, musíte vyřešit analytický problém zamykání objektů proti možné kolizi. Toto řešení nesmí způsobit deadlock komponenty (něco na způsob obsluha uzamkne a odejde na kafe). Pracuje se nad kopiemi objektů apod.

- Řešíte transakce na úrovni objektů (konzistence stavů při změnách v objektech). Při změnách v objektech může kdokoliv jiný vstoupit do vámi připraveného scénáře a provést úpravy, které radikálně mění situaci

- Řešíte přístupová práva na úrovni objektů - ve volání metod objektů se musíte odvolat na část systému, která analyticky řeší přístupová práva (agenda uživatelů apod.)

- Řešíte rychlost zpracování požadavků při růstu počtu klientů - multithreading komponenty (pro ActiveX EXE pod VB 6.0 velmi obtížné, spíše nemožné vůbec řešit)

- Řešíte problém ztráty identity klientů databáze - connections se zúžil na jeden connection komponenty a pokud v databázi používáte k něčemu uživatele (user connection), nastává kolize.

Samozřejmě tyto problémy neznamenají, že nemáte přistoupit k řešení pomocí sdílené komponenty ze zásady, jsou pouze upozorněním a varováním. Řeč teď není o tom, zda je výhodnější použít sdílenou komponentu nebo komponentu s izolovanými klienty obecně, ale kterou vybrat při přechodu od dvouvrstvové architektury na třívrstvovou architekturu.

Odpověď je jasná - začněte vždy s izolovanými klienty. Pokud totiž měníte svoji technologii, tak zásadně platí “nejlepší je dosažení cíle při minimalizaci změn”. Pokud použijete sdílenou komponentu, potom vytvoříte spoustu sekundárních problémů k řešení, které máte momentálně ve 2-vrstvové architektuře již vyřešeny.

16.8 Závěr

Článek měl upozornit na jedno možné úskalí pří přechodu od neobjektového IS k systému psanému objektově ve třívrstvové architektuře. Toto úskalí spočívá při tomto přechodu ve snaze příliš brzo použít sdílenou komponentu. Sdílená komponenta může přinést spoustu problémů, které v té chvíli vůbec nemusely vyvstat, a to je bezesporu chybný postup přechodu.

Doporučený postup pro přechod k OOP a třívrstvové architektuře je tedy následující:

V ideálním případě:

- Fáze 0 - systém je napsán dvouvrstvově a neobjektově

- Fáze I - přechod na OOP v klientech.

- Fáze II - přechod na komponenty, avšak pouze na lokále.

- Fáze III - přechod na třívrstvovou architekturu - komponenty s izolovanými klienty

- Fáze IV - případná tvorba objektových serverů - sdílených komponent

Fáze I a II lze “urychlit přeskočením”, Fázi III nedoporučuji přeskočit v žádném případě

17. Zobrazení výsledků operace

17.1 Dotaz

V tomto článku budu reagovat na následující mail, který navazuje na předešlou diskusi:

```
Dobrý den pane Kravale,

děkuji mnohokrát za informace ohledně článků. Ještě bych si Vás dovolil
otravovat jednou maličkostí. Řeším teď objekt pro zálohování a obnovu
databáze Sybase (nevím, zda není chybou použít pouze jeden objekt). Ale
myslím, že se jedná o problém celkový v rámci OOP. Bussines objekt nesmí
zobrazovat hlášky - dalo by se to použít pouze na jednom stroji. Co ale
udělat v případě, že chci zobrazit např. informační žádost o vložení
diskety? V některé knize jsem četl, že by objekt měl obsahovat elementární
metody a ty pak dle požadavků uživatelů objektu spojit do větších celků. S
tím myslím souhlasíte.

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

Děkuji mnohokrát za odpověď.

S pozdravem

Kamil K.
```

Odpověď

Nejprve několik poznámek:

Určitě není fatální chybou “použít jeden objekt”. Většinou řešení takovéhoto problému (komprimace, šifrování, zálohování atd.) bývá problémem skutečně několika málo instancí objektů a navíc většinou není potřeba instalovat dvě a více instancí ze stejné třídy vedle sebe. Zda použít jeden “velký” objekt nebo několik malých v tomto případě bývá pouze na rozhodnutí nikoliv analytické povahy, ale jedná se o otázku rozložení funkcionalit podle potřeby vašeho systému. Je to důsledek “jedno-instančních” objektů, kde neexistuje potřeba několika instancí ze stejné třídy. Určitá funkcionalita programu (například komprimace) je zapouzdřena do jedné instance objektu a není potřeba více instancí stojících vedle sebe. Je zřejmé, že velmi podobného efektu dosáhneme i ve strukturálním programování pomocí knihoven, tedy jinak řečeno tato problematika nemusí dát vyniknout výhodám OOP do všech důsledků. Například matematické funkce jako je SINUS nebo COSINUS, funkce šifrování, komprimace apod. Můžeme v tomto případě zavést třídy například C_MATH, C_CRYPTOR apod. a zavést tak instanci - objekt s odpovídajícími metodami. Výhoda je ta, že nemusíme zavádět jiné proměnné, se kterými by jinak funkce pracovaly a zavedeme je jako atributy objektu. Podobného efektu však dosáhneme knihovnou-modulem s proměnnými “schovanými” pouze pro tento modul (zapouzdření samozřejmě dodržíme!), protože většinou nepotřebujeme “dvě instance pro sinus nebo dvě instance pro cosinus, dvě instance pro kryptování, pro komprimaci” apod.

Druhá připomínka:

Je pochopitelné, že pokud dojde k výzvě k vložení diskety, potom se nejedná o problém n-vrstvové architektury - disk se nachází na stroji u klienta a kód, který provádí tuto operaci, by měl být u tohoto klienta a nikoliv na vzdáleném stroji. To však nic nemění na požadavku “dodržet zásadu mlčenlivosti komponenty vzhledem ke GUI”. Jak se autor mailu zmiňuje, používá správně metodu skládání metod do sebe. Samozřejmě tyto metody se navzájem volají. Uvedený požadavek nevolat GUI v komponentě znamená, že vnější metoda interpretuje výsledek, který získala od vnitřní metody přes návratové hodnoty těchto metod. Tedy každá komponenta ve svých metodách uvnitř nevolá GUI, ale vždy vrací interpretující výsledek.

Příklad (pozor, je to momentální výmysl pouze pro vysvětlenou):

Nechť objekt má metodu

```
Function SetDisk(strNazevDisku As String) As Integer
```

která má nastavit disk podle názvu (například název je vybrán pomocí TreeView zobrazující seznam disků). Je to funkce, která vrací konstantu. Například

```
ccmpDiskOK = 1

ccmpDiskNotReady = -1

ccmpDiskNotExists = -2
```

atd. Současně mohlo dojít k nastavení vnitřního stavu objektu.

Zavoláním této metody se dovíme také stav operace, který nastal. Uvnitř objektu však nevoláme nic z GUI. Teprve obalující například GUI aplikace, která obsahuje tuto komponentu, zavolá:

```
IntResult = MyComprimator.SetDisk(strDiskName)

Select Case IntResult
```

a zavolá GUI.

Důležité pro pochopení toho, jakou má mít z tohoto hlediska povahu komponenta, je v oné interpretaci výsledku metody. Pouze hodnota (buď návratová hodnota metody, nebo nastavený stav objektu, nebo oboje) interpretuje výsledek a teprve vnější prostředí podle tohoto stavu - hodnoty reaguje v komunikaci s uživatelem.

Touto čistotou (komunikujeme s komponentou interpretujeme její stavy hodnotami) dosáhnete toho, že komponenta se může vkládat kamkoliv bez obav z omezení prostředí, kam bude dosazena. Nemusíte se potom bát, jak se bude volat GUI

18. Ztráta identity klienta databáze při použití sdílené komponenty

18.1 Úvodní slovo

Uvedený příklad v tomto článku má posloužit jako ilustrace možného řešení v OOP a v zápise UML. V žádném případě si nevyhrazuji právo tvrdit, že nabízené řešení je nesporně to nejlepší a dokonce že nemůže obsahovat chyby. Každý navržený model lze vylepšit a v každém modelu se chyby mohou vyskytnout. Pokud bychom brali ohled pouze na bezchybnost a stoprocentní dokonalost, nikdy bychom nenaprogramovali ani jeden řádek kódu.

Následující příklady jsou spíše ilustrací “jak se dělá OA a OD pomocí UML” a nejsou to návody pro konkrétní hotová řešení. Budu jenom vděčný, pokud se najdou solidní oponenti některých mých analytických návrhů.

18.2 Konkrétní problém z praxe

V článku ***O jednom záludném úskalí při přechodu firmy na n-vrstvovou architekturu*** se doporučuje při přechodu na tuto architekturu používat nesdílené komponenty s izolovanými klienty. V článku se uvádí, že může vyvstat jako jeden z možných nových problémů “ztráta identity klientů databáze”. `Connections` do databáze ve dvouvrstvové architektuře se mohou zúžit na jediný `Connection` komponenty. Pokud se v databázi používá k něčemu uživatel databáze, potom může z tohoto důvodu nastat kolize. V předešlém článku není tento problém blíže popsán a činíme tak zde.

Představme si následující příklad z praxe, který skutečně nastal. Zde je podán mírně modifikovaný pro účely lepšího osvětlení podstaty věci:

Při řešení přechodu od dvouvrstvové na n-vrstvovou architekturu se zvolil model se sdílenou komponentou, tj. model s instancí komponenty vyskytující se pouze jednou v celém systému a klienti se k ní připojují jako k objektovému serveru. K tomuto připojení může dojít například pomocí technologie DCOM anebo pomocí MS Internet / Intranet technologie (Internet Information Server) přes ASP stránky. V tomto druhém případě je komponenta zavedena do aplikace na úrovni viditelnosti `APPLICATION` ve společném `global.asa` souboru (viz skripta Základy ASP).

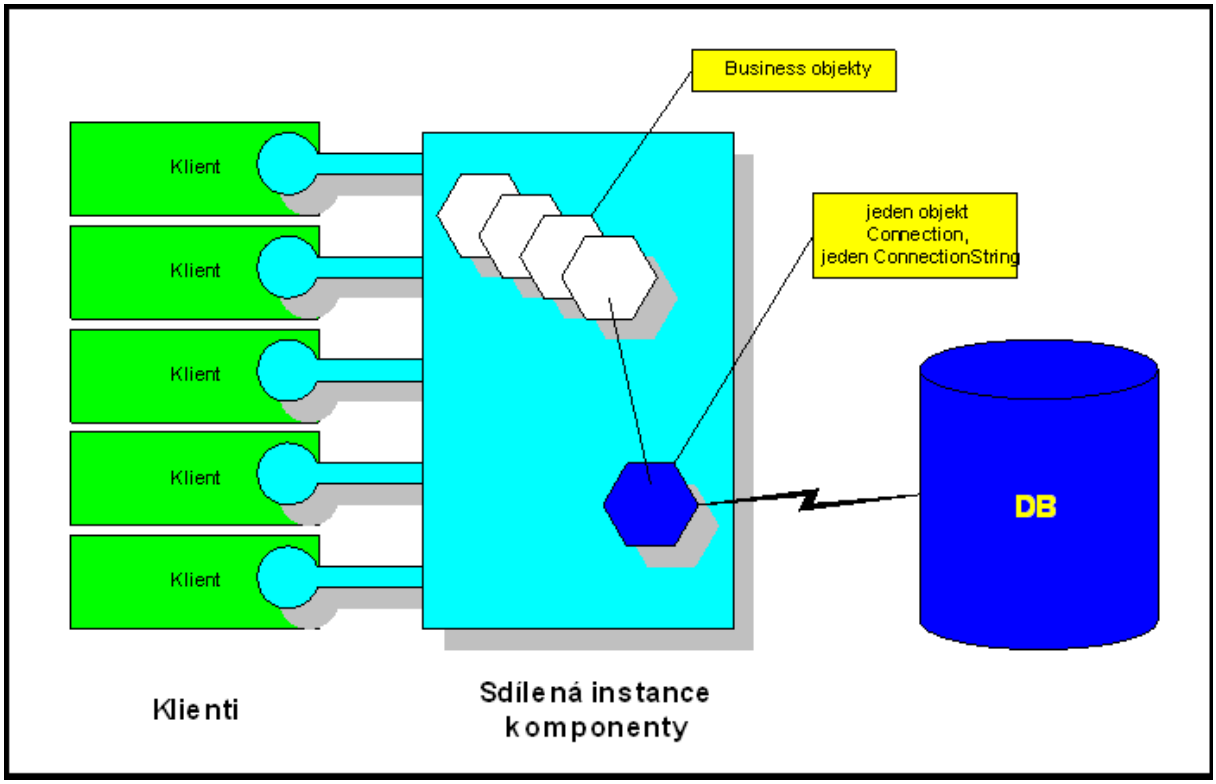
Je třeba připomenout, že v případě sdílené komponenty lze očekávat nové úkoly k řešení spojené s obsluhou takovéto komponenty vícero klienty. Jedná se o zamykání objektů, o problém transakcí nad objekty, o řešení přístupových práv atd. Tyto problémy skutečně v uvedeném případě nastaly a jedním z nich byla ztráta identity klientů databáze.

18.3 Logování operací

V uvedeném případě v praxi byl v původní dvouvrstvové architektuře řešen problém tzv. logu, tj. záznamu o operacích provedených uživatelem databáze velmi jednoduše pomocí triggerů. Nad každou tabulkou, která vyžadovala speciální zápis do logu (tj. zápis toho “kdo, co a kdy udělal”), byl postaven trigger, který zapsal do zvláštní tabulky uživatele databáze, uvedenou operaci a čas jejího provedení. Trigger měl pochopitelně v daný okamžik k dispozici všechny potřebné údaje včetně uživatele databáze, který tuto operaci prováděl. Tímto jednoduchým mechanismem byla každá kritická operace zapsána centrálně spolu s uživatelem a časem.

Existují samozřejmě jiné způsoby logu, důležité pro tento příklad je to, že tento způsob byl již vyřešen ve dvouvrstvové architektuře a to vcelku uspokojivě.

Při přechodu na n-vrstvovou architekturu se sdílenou komponentou se v prvé chvíli nedocenil nově vzniklý problém spojený se sdílením klientů. Instance komponenty byla navržena s jedním konektem do databáze (spolupráce s databází byla prováděna pomocí technologie ADO, viz skripta o ADO technologii):



Na první pohled se připojení do DB pomocí jednoho konektu, tj. pomocí jednoho `Connection` objektu komponenty a jeho jednoho `Connectionstring` property, jeví jako velmi výhodné. Za prvé cena za konetky je minimální. Navíc spousta informací je ve sdílené komponentě přístupná “přímo a naživo” v business objektech a tedy datová vrstva nemá za úkol provádět sdílení informace (to se děje už v aplikaci), ale pouze informace zperzistentnit (tj. učinit stálými i v případě vypnutí a zapnutí systému). Čistě teoreticky - pokud by systém nikdy nemohl “spadnout” a nikdy “se nevypnul”, potom by datová vrstva v tomto modelu neměla význam. Posláním datové vrstvy v tomto modelu je pouze “znovuoživení” objektů v okamžiku restartu systému a proto je třeba v určitých okamžicích objekty zperzistentnit a uložit jejich obsah. V tomto modelu je pochopitelně přenos dat mezi databází a objekty mnohem omezenější, než u dvouvrstvové architektury a u n-vrstvové s izolovanými klienty. Na druhé straně spoustu věcí se musí řešit na aplikační úrovni.

Nelze provést kombinaci tohoto modelu se sdílenou komponentou s jedním konektem (podle obrázku) a modelu logu s triggery uvedeným v předešlém odstavci. Je pochopitelné, že v modelu sdílené komponenty s jedním konektem existuje pouze jeden uživatel databáze a tím je sama instance komponenty. Do logu se tedy bude zapisovat pouze jeden uživatel (odvozený z `Connectionstring` komponenty), ať už o operaci požádal kterýkoliv z klientů komponenty. Vzhledem k databázi se identifikace klienta ztratila a to díky odstínění klienta od databáze přes prostřední vrstvu business objektů. Z logu nelze určit, který klient komponenty požádal o zápis.

Problém je tedy nastolen, pokusme se o řešení.

18.4 Úplné a “čisté” řešení u sdílené komponenty

Pokud zůstaneme u modelu se sdílenou komponentou a přitom budeme chtít řešit logování, tj. zápis operací, potom musíme toto logování řešit již na úrovni objektů v samotné komponentě. Součástí analytického řešení se tedy stává agenda uživatelů, což vede k dalšímu rozšíření systému o novou část aplikace. Z hlediska UML bychom tuto situaci mohli charakterizovat jako rozšíření Use Case modelu o další činnosti spojené se administrací uživatelů a o zápisy do Logu.

V tomto modelu přímo na úrovni objektů se žádá o zapsání do Logu, což nyní reprezentuje objekt se svými metodami. Je pochopitelné, že když máme OOP aplikaci plně pod kontrolou (v daném okamžiku máme k dispozici všechny potřebné objekty), můžeme o zápis požádat v libovolném okamžiku ve scénáři spolupráce objektů a nikoliv pouze při operaci nad tabulkou. Ve složitějších scénářích se jedná o nespornou výhodu.

Požádání o službu zápisu do logu se děje přes objekt - nazvěme jej jako `Log`. Objekt `Log` má v našem návrhu “hlavní” metodu pro zápis informace o provedené operaci. Za touto metodou je již díky encapsulaci pro uživatele objektu uschován způsob, jakým je tato informace uložena a kam se ukládá. Může se dokonce jednat o řešení s volitelným způsobem zápisu. Zaveďme například property `Mode` tohoto objektu `Log` s hodnotami

- do tabulky
- do text filu anebo
- do systémového logu daného operačního systému

Před použitím `Logu` se musí tento mód nastavit. Spolupráce s tímto objektem pak vypadá například symbolicky zapsána v pseudo-kódu VB takto:

```
...
Log.Mode = logTextfile ‘konstanta s hodnotou např. 1
StrIn = User.Name & ”;” & Time & ”;” & ”přihlášen”
Log.Write StrIn
...
```

Toto řešení má velkou výhodu v tom, že je od OO analýzy až po kódování plně pod naší kontrolou, jak se bude vlastně do `Logu` zapisovat. Navíc použití OOP nám umožňuje schovat do objektu `Log` implementační podrobnosti tohoto zápisu v konkrétní podobě. Pro uživatele objektu `Log` stačí zavolat jeho metodu pro zápis (např. `Write`) s předchozím nastavením módu zápisu (viz předchozí ukázka).

Další rozšíření analytického modelu může například využít jednoznačných identifikátorů objektů (tzv. `OID` - object identifier). Hodnoty `OID` se mohou do `Logu` zapisovat spolu s dalšími informacemi a tím identifikovat objekt, nad kterým probíhá operace. Je také třeba zapsat operaci a uživatele (v předešlém případě byl zapsán `Name` uživatele). Je analytickou otázkou, odkud se při volání sdílené komponenty dostane do dané operace samotná identifikace uživatele. Jedná se o problém identifikace klienta komponenty, který volá metodu komponenty.

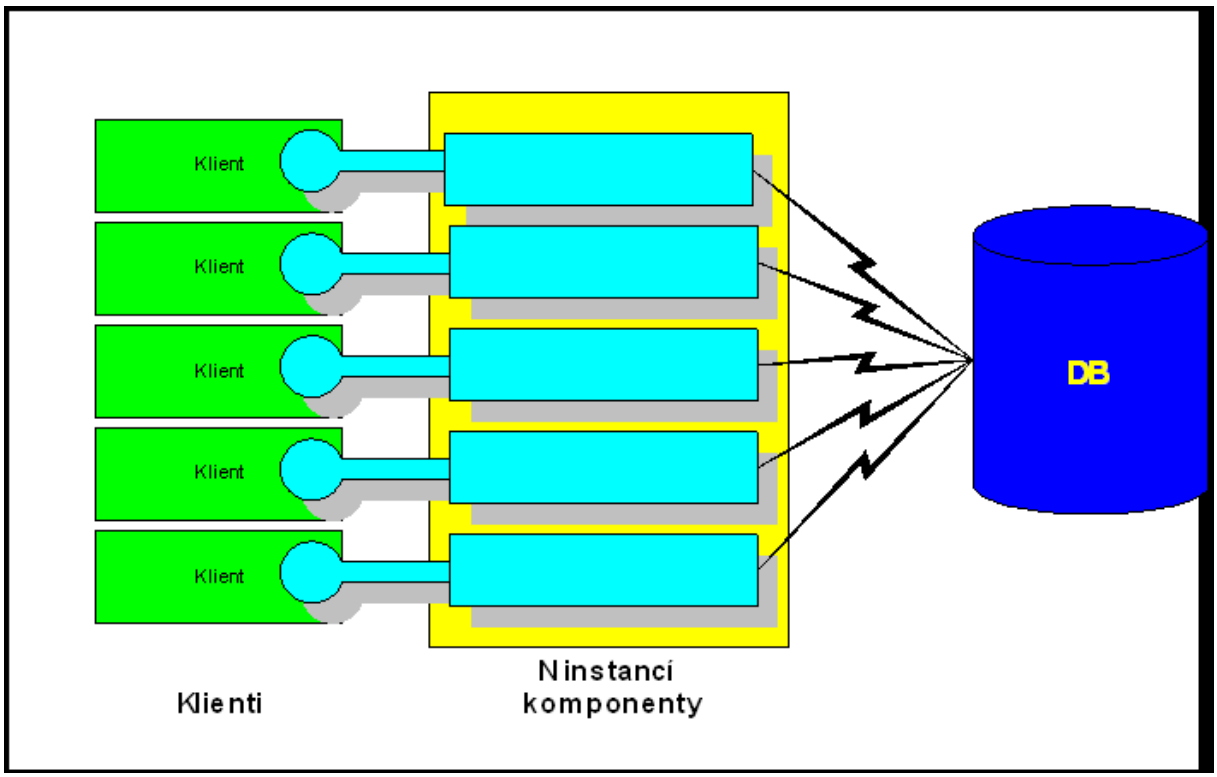
Nevýhodou tohoto úplného přístupu je “objevení se” celé nové agendy uživatelů k dalšímu řešení, což může z hlediska plnění obchodní zakázky firmy a přechodu na třívrstvovou architekturu zkomplikovat

situaci. Pokud se vyžaduje “rychlé řešení”, potom mohou nastat problémy v projektu kapacitního a časového charakteru.

Navíc při přechodu od dvouvrstvové architektury se může pro první fázi přechodu využít již stávajícího mechanismu bez jeho narušení (viz další kapitola) a tím ušetřit čas a kapacity při přechodu na třívrstvovou architekturu.

18.5 Řešení pro komponentu s izolovanými klienty

Pokud bychom hned na začátku nepřistupovali k řešení pomocí sdílené komponenty, ale pomocí komponent s izolovanými klienty, potom se situace v našem příkladu výrazně zjednoduší:



Každý klient “vlastní” svou instanci komponenty a každá tato instance má svůj konekt do databáze. Z tohoto důvodu je klient vůči databázi jednoznačně identifikován (jeho identita není odstíněna) a dokonce v našem příkladu se nemusí v prvém kroku vůbec nic měnit - triggery obsluhující zápisy do logovací tabulky budou v tomto modelu fungovat ihned a beze změn.

Existují samozřejmě nevýhody takto zvoleného modelu - v prvé řadě informace se sdílejí až na úrovni databáze, tj. naplněné objekty v instancích komponent se musejí častěji synchronizovat vůči datům (refresh objektů apod.). Další nevýhodou je růst konektů do databáze podle počtu klientů, což se dá řešit pomocí tzv. *connection poolingu* (bude pojednáno dále).

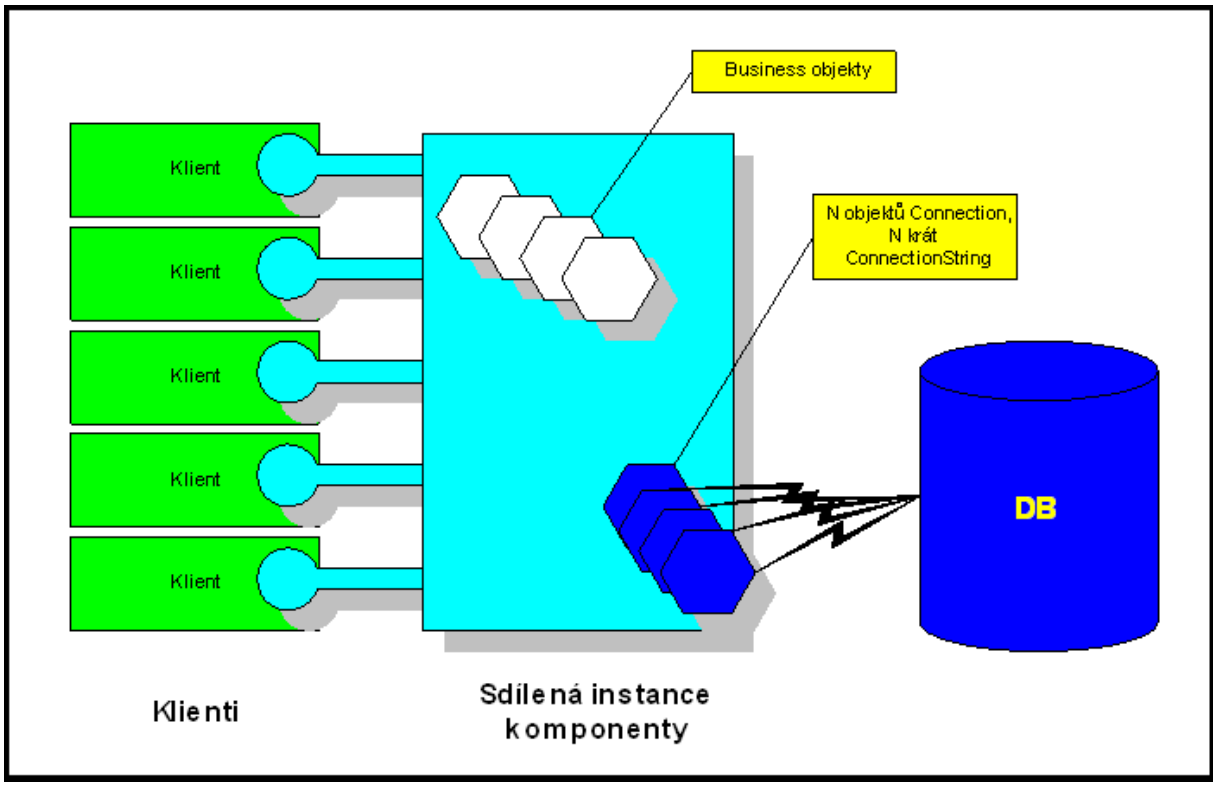
18.6 Řešení pro sdílenou komponentu a zavedené triggery

Třetí řešení může vyplývat ze situace, kdy je sdílená komponenta již připravena k použití, ale nemá zavedenu agendu uživatelů a přitom se žádá použít modelu zápisu s triggery. V tomto případě se jedná o jakousi kombinaci sdílené komponenty a modelu zápisu do logu s triggery nad tabulkami.

Řešení tohoto problému můžeme navrhnout pomocí zavedení kolekce `Connection` objektů v dané komponentě. Každý klient je sice odstíněn od databáze, ale komponenta má k dispozici N otevřených `Connection` objektů a je na klientovi, aby před operací zvolil ten pravý `Connection` (ten, který je jeho). Klíčem této kolekce konektů bude samotný `Connectionstring`. Spolupráce klienta se sdílenou komponentou může být následující:

Klient zavolá komponentu a předá jí `ConnectionString`. Pokud tento konekt ještě neexistuje, komponenta jej založí a otevře jej vůči databázi. V druhé fázi klient bude žádat v nějakém scénáři o operaci nad komponentou (například založení nového objektu apod.). Klient zkontroluje, zda není komponenta uzamknuta. Pokud není, uzamkne ji pro sebe a předá jí svůj `ConnectionString`, který je od té chvíle považován za aktivní. Komponenta nalezne odpovídající `Connection` podle `ConnectionStringu` a dosadí jej za aktivní (`Set ActiveConection =...`). Při dané volané operaci použije tento aktivní konekt pro danou datovou operaci. V operaci se poté spouští trigger pro Log se správným uživatelem databáze podle aktivního konektu. Po ukončení operace klient komponentu odemkne pro další použití.

Uvedený model ukazuje následující obrázek:



Klient při volání nějaké operace komponenty si nejprve “vybírá” z kolekce konektů ten svůj a přes tento konekt se provádí datová operace nad databází. Uzamčení komponenty je nutné, protože se jedná o sled volání metod komponenty, do kterého by mohl vstoupit a nežádoucím způsobem zasáhnout druhý klient. Uzamčení však nevede k uživatelskému deadlocku (odchod na kafe apod.), protože se jedná o sled volání komponenty v jedné sekvenci kódu - kontrola uzamčení, samotné uzamčení, výběr konektu, volání operace a odemčení se děje v jedné sekvenci například po spuštění OK buttonu ve formuláři.

Tímto způsobem - tj. výběrem konektu klientem - lze také vyřešit problém ztráty identity klientů v databázi.

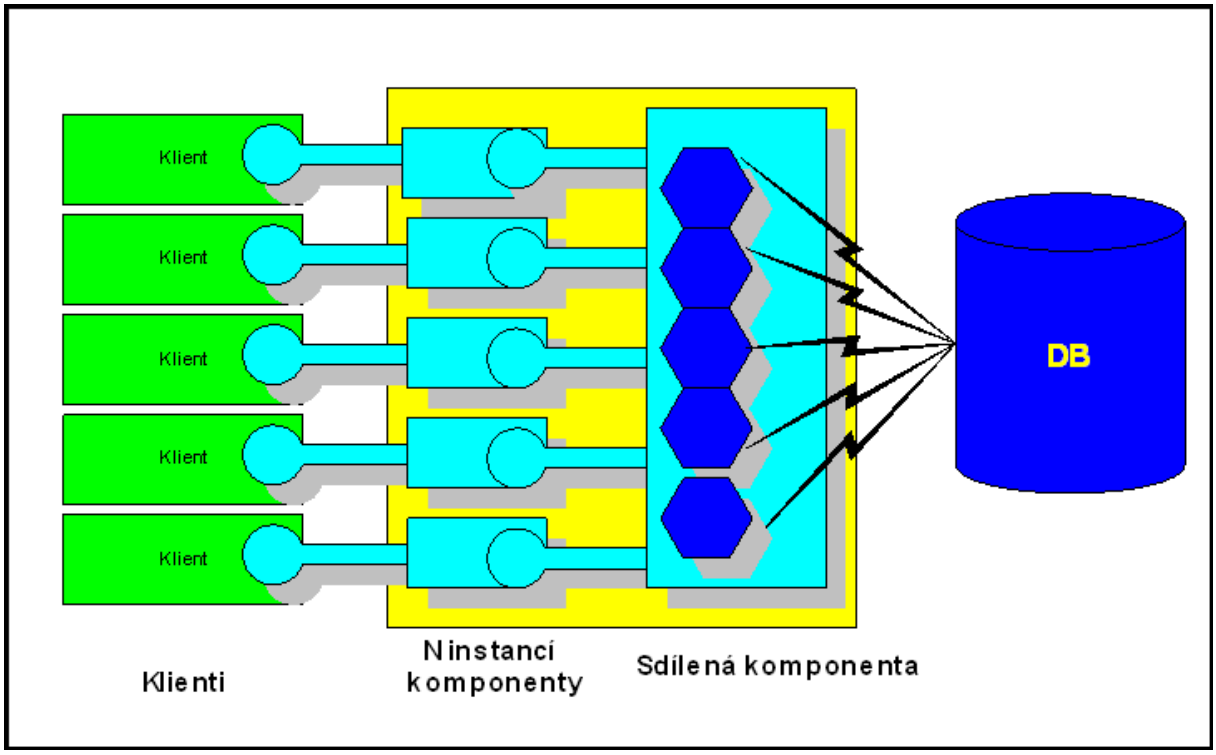
Je zřejmé, že komponenta by měla nějakým způsobem hospodařit s `Connection` objekty v tom smyslu, že by měla uvolnit ty, které nikdo nepoužívá resp. nebude v dohledné době používat. Při žádosti o nový konekt (dáno obsahem `Connectionstringu`) by se nemusel tento nový konekt zakládat, pokud již existuje nějaký volný. Měl by se použít stávající nepoužitý konekt. Dá se to zabezpečit například tím, že se konekty nebudou uvolňovat ihned po jejich použití, ale “zůstanou na nějaký čas viset” a bude se sledovat ve vedlejším procesu nepoužité konekty. Až po určitém časovém intervalu se budou tyto visící konekty uvolňovat (například za 5 minut apod.), čímž se optimalizuje čas pro tvorbu konektů. Protože klient pouze žádá komponentu o konekt a sám jej nezakládá, je úplně v moci komponenty, jaký konekt vydá a kdy jej zruší.

Tento model je sice “nečistým hybridem”, ale ukazuje již na možnost manipulace s konekty a optimalizace práce nad nimi. Tato optimalizace při práci nad kolekcí konektů se obecně nazývá *connection pooling* (pool - zásoba, rezerva).

18.7 Izolovaní klienti a connection pooling

V předešlé kapitole se popsala možnost řídit práci s konekty do databáze u sdílené komponenty. Bez dodatečných úprav nelze obecně u “čistého” modelu s izolovanými klienty zavést tento model práce nad connection poolem, protože u modelu s izolovanými klienty jeden klient neví nic o druhém klientovi a každý z nich má svůj nezávislý konkekt.

Neznamená to však, že nelze úpravami u izolovaných klientů connection pooling použít. Zaveďme “v pozadí” za izolovanými klienty společnou sdílenou komponentu určenou pro connection pooling. Její postavení ukazuje následující obrázek:



V uvedeném modelu jsou sice instance komponenty izolovány (nejedná se o model sdílené komponenty), tj. každý klient "vlastní" svou instanci komponenty, ale pro samotný pooling konektů (na obrázku modré šestihrany) je použita sdílená komponenta. Scénář spolupráce s touto komponentou při optimalizaci konektů do databáze je stejný, jako byl uveden v předešlém odstavci, pouze klienty sdílené komponenty se stávají instance komponenty s izolovanými klienty. Protože sdílená komponenta pouze "hospodaří" s konekty (s `Connection` objekty), je relativně málo dotazována svými klienty - volání nastává pouze při jejich zrodu a zániku.

18.8 Závěr

Na jednom z příkladů z praxe bylo ukázán možný problém ztráty identity klienta databáze u sdílené komponenty. Bylo ukázáno, v čem je v tomto případě výhoda postupného přechodu ke třívrstvové architektuře pomocí komponent s izolovanými klienty. Nakonec se ukázalo, že lze pomocí sdílených komponent optimalizovat zrod a zánik konektů do databáze i u komponent s izolovanými klienty při zavedení sdílené komponenty, která řídí zrod a zánik těchto objektů `Connection`.

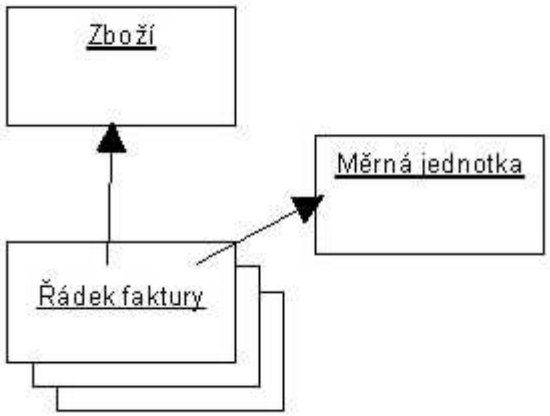
19. Prvek ze seznamu

19.1 Dotaz

V knize Objektového modelování v praxi 2000 se často připomíná přesnost. Z toho vyplývá, že by textový popis měl souhlasit s grafickou reprezentací téhož. Jinými slovy. Pokud je z textového popisu vytvořen diagram, mělo by jít i z diagramu odvodit textový popis, který by z analytického (!) hlediska odpovídal popisu, podle kterého byl diagram vytvořen.

Na str. 45 je tento příklad konceptuálního diagramu s popisem (uvádím jen spornou část):

…Řádek Faktury si ukazuje na jedno konkrétní Zboží ze Seznamu Zboží. Dále Řádek faktury si ukazuje na jeden objekt Měrná jednotka ze Seznamu měrných jednotek…

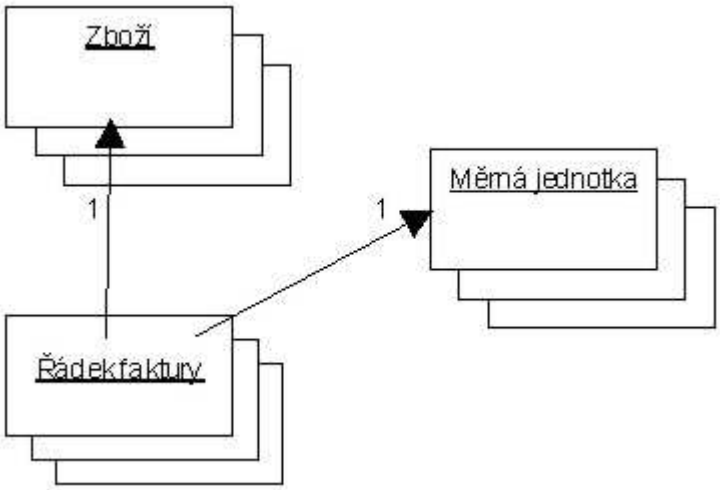


Pokud by měl někdo odvodit popis pouze z diagramu, **nedospěl** by ke stejnému analytickému výsledku, ale k něčemu takovému:

…Řádek Faktury si ukazuje na jedno konkrétní Zboží. Dále Řádek faktury si ukazuje na jeden objekt Měrná jednotka… Nemohl by totiž z diagramu nic vědět o Seznamu Zboží a Seznamu měrných jednotek. V diagramu se nám tento pojem vytratil.

Mám dojem, že jsme konverzí textu do diagramu přišli o důležitou informaci. Podle diagramu to vypadá, jako by šlo o specializovaný systém v němž figuruje pouze **jediné** zboží a **jedna** měrná jednotka. Myslím si, že následující diagramy jsou přesnějším (přesným) obrazem původního popisu.

V prvním jsem pro zachování informace „1 ze seznamu“ použil označení číslicí jako u násobnosti v UML, v druhém jsem zavedl nový element „1 ze seznamu“ za pomoci zvýraznění jedné položky (více se mi líbí).



Jaký je Váš názor na ztrátu informace v diagramu?

19.2 Odpověď

Nemáte až tak úplně pravdu s tím, že je třeba v diagramu, který znázorňujete, namalovat také to, že daný objekt pochází z nějakého seznamu. To je právě v tomto vztahu mezi Řádkem faktury a Měrnou jednotkou resp Druhem zboží úplně vedlejší! Jakmile je objekt jednou dosazen, nezajímá mne, jak se tam dostal a že má nějaké "soudruhy itemy" vedle sebe.

Někde na jiném diagramu tuto skutečnost nezávisle na tomto diagramu vyjádřím.

Tato úvaha oddělitelnosti problému řádku faktury a měrné jednotky zní takto: Řádek faktury vždy ví jenom o jedné měrné jednotce! Tato úvaha mimo jiné vede v důsledku k možnosti nejjednodušší optimalizace, jaká existuje a tou je optimalizace v omezeném scénáři klienta kolekce. Klient při takovéto optimalizaci nepracuje s celou kolekcí, ale pouze s několika málo členy této kolekce. Ostatní itemy se jednoduše neoživí. Vím, že budu pracovat například jenom s jedním objektem.

20. Dotaz k diagramu na předešlém obrázku v kapitole

20.1 Dotaz:

K předešlému diagramu v kapitole 18: Nejsem zvyklý používat nákresy objektů pro vyjadřování souvislostí mezi objekty. Za tímto účelem používám diagramy tříd (class diagrams). Pokud někde zmiňuji konkrétní objekty, nebo dokonce sady objektů, tak jedině v Collaboration Diagram. Navíc, v class diagramu můžete vyjádřit úplně pohodlně a zcela přehledně vztah jedna-ku-en, atd.

Další věc, která mě zarazila, je terminologie "ukazuje si". Pokud vím, objekty mohou mít následující relace:

- asociace (obousměrná, jednosměrná)
- agregace
- specializace
- realizace

Nikde nic o ukazování!

Šipky, které jsou použity v diagramu mají černou výplň. Něco tahového neznám: vždycky jsem si myslel, že výplň má být bílá a pak to oznamuje specializaci. Bohužel teď nemám po ruce UML Guide od Booche, s níž jsem chtěl toto konzultovat. Podle obrázku mi připadá, jako by byl Řádek faktury poděděn po Zboží a Měrné jednotce.

Pokud má autor na mysli asociaci, tedy to, že z objektu Řádek faktury se můžeme dostat na Zboží či Měrnou jednotku, měla by být šipka jednoduchá, s žádnou výplní.

Nechci tu být za pana chytrého, nicméně s UML pracuji už nějaký ten pátek a používám jej celkem rutinně ke své práci. Možná jsem moc deformován Rational Rose. Zajímal by mě hlavně názor pana Kravala na tuto problematiku. Myslím, že UML by mělo být jednotné a jestli jednotné není, tak jako mnoho jiných "standardů" dnes, co si počít? Kde se učit? Jestli u Grady Booche (můj případ) nebo někde jinde? Rád bych totiž rozuměl i jiným diagramům, než těm, které sám vytvořím, a ostatně, chtěl bych aby i jiní lidé rozuměli mým diagramům. Je více odnoží UML? Co je toto za způsob vyjadřování (myslím samozřejmě UML :)?

20.2 Odpověď

Pochopitelně mate pravdu, ze předešlý obrázek není z UML. Jedná se skutečně o obdobu diagramu instancí a nejblíže k němu má "Collaboration diagram bez zprav". Uvedený diagram (samozřejmě s upozorněním, ze není z UML) uvádím ve svých e-knihách jako velmi výhodný pro úvodní analytické úvahy a konzultace s uživatelem a nazývam jej konceptuální resp. pojmový diagram a je analytickym zdrojem pro tvorbu Class diagramu (což je již UML). Chápejte jej tedy jako "PRE-UML diagram" a je nestandardní a stojí mimo UML. To také ve skriptech velmi zdůrazňuji!

Diagram tříd je o jednu abstrakci "meta" výše a mnohdy bývá obtížné jej rychle vytvořit. Osvědčilo se mi v praxi u tvorby IS vyjít z nižší úrovně abstrakce a to na úrovni instancí. Tedy s uživatelem při diskusích mluvíme (a malujeme) nikoliv třídy, ale nejprve instance ve smyslu: "Představte si jednu fakturu, jednu konkrétní fakturu, co ta obsahuje?" atd. Přechod od instanci k třídám jako zobecnění je poté proveden až v druhém kroku. Osobně se totiž domnívám, ze takto vždy uvažujeme, i když mluvíme rovnou o třídách: Nejprve několik případů, poté zobecníme.

Rozdíl oproti diagramu instancí v UML (který vznikne jako zvláštní případ Collaboration diagramu) je v tom, ze na rozdíl od UML ve vztahu objektu se v pojmovém diagramu u linku mezi objekty rozlišuje, zda se jedna o agregaci nebo asociaci a v jakém směru. Samozřejmě pojem "ukazuje si", který použil autor předešlého dotazu,. je ve Vašem výčtu možných vztahů synonymem pro jednosměrnou asociaci od A k B (A "vidi" B). Šipky mají být pochopitelně nikoliv plné (dělal to asi ve WORDU).

Tento "instanční model" má jednu velkou výhodu - dává do vztahu role bez tříd. Teprve v druhém kroku se jedná o hledání tříd. Uvedu klasický příklad.

Věta z konzultace: Na letišti se u letu letadla budou evidovat piloti a pasažéři pro případ havárie letadla. Nyní jsou jasné ROLE, a volba rešení tohoto diagramu muže být :

LET asociace na N PILOTu,

LET agregace N PASAZERu.

Teď však v této chvíli nevím, jestli Pilot a Pasažér jsou ze stejné třídy anebo pouze z jedné rodiny tříd anebo nemaji žadný vztah. Takto mohu "iterativne" zpřesňováním pojmů dospět až k modelu tříd.

21. User jako globálně viditelný objekt

21.1 Dotaz

Zúčastnil jsem se semináře "Nekteré obecne známé agendy IS rešené objektove v UML a pomocí komponent", kde se probírala mimo jiné agenda přístupových práv. Svůj dotaz se mně nepodařilo na semináři srozumitelně zformulovat, takže zůstal nezodpovězen. Zkusím to nyní takto. Předpokládalo se, že někdy při přihlašování vnikne objekt USER reprezentující přihlášeného uživatele i s jeho právy. Na začátku každé metody aplikačních objektů (označme ho APLIK) je volána metoda objektu User za účelem ověření přístupu. Ptám se: Jak objekt APLIK, získá objektovou referenci na objekt USER? Z analytického pohledu existují v OOP pouze tři možnosti, jak může objekt A získat referenci na jiný objekt B.

1. Objekt A sám stvoří objekt B.

2. Objektu A je může být objekt B předán jiným objektem v parametr poslané zprávy, čili jako parametr volání metody.

3. Objekt A může objekt B získat jako návratovou hodnotu zprávy, kterou A poslal nějakému jinému objektu C. Referenci na tento objekt C však musel získat jedním z těchto tří způsobů.

V našem případě pro objekty APLI u USER není způsob 1 možný z principu. Způsob 3 problém jen odsouvá na získání objektu C, takže zbývá možnost 2. Vidím tato řešení:

1. Objekt APLIK a každý aplikační objekt má metodu setUser(USER), která nastaví referenci na objekt uživatele do svých instančních proměnných. Na každém nově zřízeném objektu pak musí objekt volat setUser a předat mu referenci na objekt uživatele, jenž si drží.
2. Každá metoda má povinný parametr typu USER, refernece na uživatele je předávána do všech volaných metod. Každá metoda má uživatele k dispozici, reference se nemusí ukládat do vnitřního stavu objektu.
3. Upustí se od objektově čistého řešení. Objekt USER se uloží do globální proměnné nebo se magicky získá voláním globální funkce nebo statické metody. (Pohybujeme se v jazycích, kde třída není objekt)

Mám tyto otázky:

- Existuje nějaké lepší řešení, než jaká jsem nastínil?

- Pokud ne, které řešení považujete za nejvhodnější pro praktické uplatnění?

21.2 Odpověď

Umístit objekt do obdoby „globální“ viditelnosti, tj. do viditelnosti všech zúčastněných tříd, které jej potřebují, v žádném případě neodporuje OOP. Jedná se o situaci velmi běžnou a existuje na to dokonce návrhový vzor zvaný SINGLETON

Namísto toho, aby se pracně předávala reference do lokální viditelnosti, umístí se objekt (který má tuto povahu) do obdoby „globální viditelnosti“. Dokonce lze tuto skutečnost zapsat v modelu, kdy se sice napíše asociace na třídu, ale ke konci asociace se přidá značka g (jako global).

Je zřejmé, že objekt User bude potřebovat "takřka každý", proto je dobré jej umístit do této obdoby „globální viditelnosti“. Podobně se například umísťují do globální viditelnosti vlastní datové objekty, přes které se volá databáze.

V případě, že jazyk podporuje objektové proměnné v globální viditelnosti (Objektový Pascal, Visual Basic apod.), umístí se přímo tam.

Pozor: Neznamená to, že objekt v globální viditelnosti je sdílen "přes celou aplikaci". Sdílený objekt by musel být umístěn "nějak" mezi všechny instance komponent tak, aby byl viděn všemi klienty (například v ASP jako Application scope apod.). Tato zde zmiňovaná globální viditelnost je míněna jako "globální viditelnost v rámci daného klienta (komponenty)". Např. pro představu ve VB 6.0 je klasicky zavedeným globálním objektem jedna instance objektu v modulu BAS a je umístěna jako Public. Je globální v rámci tohoto projektu a nikoliv přes celý informační systém. K tomu, aby byla sdílená přes celý IS, musela by být vyvinuta jako sdílená instance komponenty.

V čistých OOP jazycích, které neznají pojem „globální proměnné“ (například Java a C# a pod.) se reference na tyto objekty získávají pomocí „static“ metody abstraktní třídy, v našem případě bychom ji nazvali například

```
static CUser GetUser()
```

Pokud již požadovaná instance existuje, vrací se tento jeden sdílený objekt. Právě takto je navržen vzor SINGLETON (jedno-instanční objekt)

22. Kniha Objektové modelování a UML“ v praxi 2000

22.1 Připomínky k knize Objektové modelování a UML“ v praxi 2000

Dočetl jsem knihu a chtěl bych v této diskusi představit několik pohledů na ni.

22.1.1. Začátečník v OA, OM a UML

Jsem začátečník a proto mohu e-knihu hodnotit pouze z tohoto pohledu – co se znalostí OA, OM a UML týče. Tady by stačilo, možná, jen jedno slovo: **Nepostradatelná** Na našem trhu nemá obdoby. Každý, kdo si chce udělat obrázek o OA a OM by si ji měl přečíst. V češtině nenalezne lepší volbu. Hloubka, do níž kniha zasahuje, je zvolena velmi dobře a proto se nesetkáte s povrchním popisem ani s popisem, který do úplných detailů (pro začátečníka nepoužitelných) rozebírá určité téma.

22.1.2. Programátor

Tento pohled se mně jeví jako nejvíce rozporuplný. Na jednu stranu se stává pro každého programátora téměř nutností se v dané problematice orientovat a na druhou stranu vám hrozí jedno „nebezpečí“. Doposud jsem byl spíš programátor „fanatik“. Pocit z dobře napsaného kódu zde nemusím popisovat. Programátoři jej znají, ostatní nepochopí. Snad je výstižné slovo euforie. Nyní začínám podobné pociťovat i při analýze a modelování. Dříve jsem se snažil tuto, pro mě nudnou část „zmáknout“ co nejdřív a hurá do kódování. Teď vidím i v těchto ranných etapách dobrodružství a každý správný krok jen umocňuje mé nadšení. Mám však pocit jisté „nevěry“. Oddaný „kodér“ pošilhává stále více k analýze a modelování. Uvidíme, jak to dopadne. Každopádně ji programátorům vřele doporučuji.

22.1.3. Zájemce

Každý zájemce získá s e-knihou i skripta „Třívrstvová architektura v praxi“ i s doprovodnými příklady. Je to jakási „přidaná hodnota“ v .ZIP souboru distribuce. Skripta jsou též psána čtivým a srozumitelným jazykem. ***Vynikající***. Navíc je v e-knize avizováno zveřejnění modelů a zdrojových kódů. Při spojení e-knihy a již vypracovaných příkladů zkušených odborníků vznikne komplet dalece přesahující její současnou hodnotu.

22.1.4. Typograf a korektor

V knize je několik „prohřešků“, které by neměly projít do hotového díla. Nemají sice vliv na kvalitu obsahu, ale i kvalita zpracování se cení. Jako jeden z největších prohřešků je používání „spojovníku“ ve všech případech. Kde má být pomlčka – je spojovník. Lze nalézt i znak palce místo uvozovek apod. Toto nejsou klasické překlepy, ale typografické hrubky. Dále by bylo vhodné dodržet jednotu ve výrazech. Oblíbený „moloch“ je psán jednou v uvozovkách a jindy, ve stejné situaci, bez. **Korektura nebyla tak důsledná, jak by měla být**. Je štěstí, že se jedná o e-knihu a vše lze snadno napravit v aktualizovaném vydání.

22.2 Odpověď

Díky za podnětné připomínky. Samozřejmě průběžně bude kniha opravována, chyby korektury vznikly tím, že

- jedná se o e-knihu, kde autor a korektor je jednou osobou. Pro korekturu jsou dobré vždy druhé oči
- samozřejmě dodržení termínu při současném vytížení ve školeních, in-house konzultacích, správě tohoto serveru, psaní článků atd. se projevila v těchto nedostatcích, které budou průběžně odstraňovány.

23. Role osoby

23.1 Dotaz

Mějme třídu COsoba. To bude "osoba o sobě", takže podporuje zprávy typu Vydej/Nastav Jméno, Příjmení, DatumNarození ...Nechť osoby vystupují v našem informačním systému v různých rolích. Jedna z nich bude třeba Zaměstnanec, druhá bude OsobaPobírajícíDůchod, třetí bude třeba Uživatel. Každá z těchto rolí má u sebe dodatečné atributy a dodatečné zprávy na rozhraní. Plni elánu začneme modelovat třídy.

CZamestnanec extends COsoba,

COsobaPobirajiciDuchod extends COsoba,

CUzivatel extends COsoba.

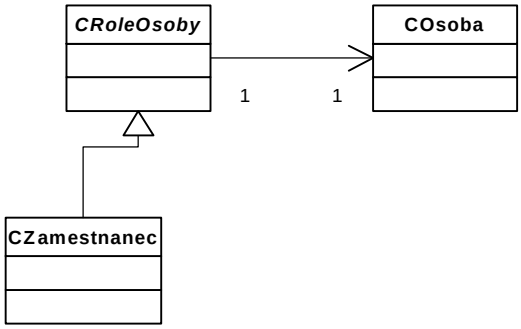
Nojo, jenže co když někdo je jak zaměstnanec, tak uživatel. Nebo je to zaměstnaná osoba pobírající důchod, která může figurovat jako uživatel. Jinými slovy, musíme použít vícenásobnou dědičnost (pokud ji prostředí nabízí), a definovat CZamestnanecPobirajiciDuchod, CUzivatelPobirajiciDuchod, CZamestnanecUzivatel a nakonec ještě CZamestnanecUzivatelPobirajiciDuchod.

Uf. A teď si představte, že po nějaké době života systému přijde zákazník s tím, že by chtěl ještě přidat roli KontaktníPracovníkObchodníhoPartnera. Soustava osmi tříd tak má vzrůst na šestnáct. Já jsem slabší povaha, já bych s tím asi praštil. Generalizace-specializace asi není nejlepším prostředkem k řešení tohoto problému.Co tedy použít?

23.2 Odpověď

Opravdu není nejlepším řešením. Jedná se o klasický problém hierarchie stromu, který vede k násobnosti typu „každý s každým“. Uvedený problém řeší návrhový vzor BRIDGE.

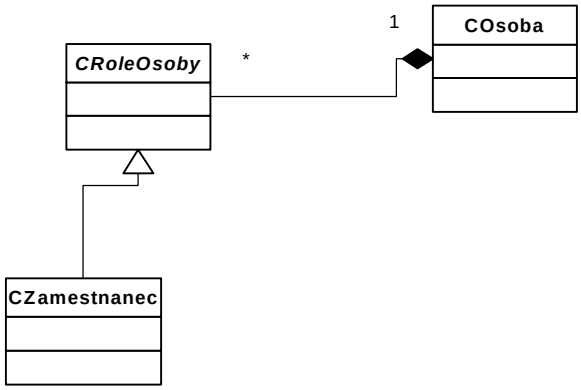
Jedno z řešení je použít třídu Cosoba nikoliv jako generalizaci, ale jako asociaci. V prvním přiblížení lze do každé třídy role, která „to potřebuje“ (máte je vyjmenovány) vložit asociaci na třídu COsoba. V druhém kroku lze tento přístup ještě vylepšit a provést vztah gen spec v uvedených rolích a na vrchní úrovni vložit tuto asociaci pouze do nejvyššího předka všech rolí, vznikne tak realizace vzoru BRIDGE:



CRoleOsoby je abstraktní třídou a dole se rozvíjí strom všech rolí osoby, každá z rolí má díky podědění od této nejvyšší třídy za sebou osobu.

Pokud potřebujeme k roli osoby přistoupit jako k osobě, potom můžeme ještě do třídy CRoleOsoby implementovat interface osoby a delegovat metody tohoto interfacu na asociovaný objekt osoby.

Také jsem se setkal v diskusi s návrhem, který je podobný pouze s tím rozdílem, že role je „schována“ až v agregaci za osobou, tedy takto:



Je zřejmé, že třída CRoleOsoby má i v tomto případě vztah k třídě COsoba jako konec asociace, tj. referenci ve směru opačném, než je směr agregace. Stejně jako v předešlému modelu existuje tedy reference od role osoby k osobě a tady rozdíl oproti předešlému modelu není (role „vidí“ svou osobu) . Tento druhý model s agregací však nepracuje s rolemi jako se „samostatnými entitami“, ale s rolemi jako se součástmi osoby.

Na rozdíl od toho první model s asociací chápe entity role jako něco, za kým je schována nějaká osoba (role jako „maska“), druhý model s agregací chápe osoby jako něco, co role obsahuje.

Máme tedy tři možné pohledy:

1. Každá role osoby je dědicem osoby, tedy je vlastně také osobou (vede k divokému stromu dědičnosti, jak uvádí autor dotazu, není zde nikde namalováno v diagramu)
2. Za každou rolí je schována osoba, tedy stačí, aby existovala jedna role obecná, která toto činí a ostatní role toto podědí (první obrázek této kapitoly)
3. Osoba obsahuje role a ty jsou různého typu (druhý obrázek této kapitoly), každá role „vidí“ svého majitele, osobu.

Mě osobně se nejvíce líbí varianta 2.

24. XML a datová vrstva

24.1 Dotaz

Dobrý den,

oddělení databázové vrstvy od obchodní logiky je určitě správné. Kromě výhod, které už byly v tomto fóru uvedené, bych doplnil ještě jednu: Izolace vývoje schématu databáze do databázové vrstvy. Nemyslím zrovna přidání atributu, to se promítne napříč všemi vrstvami, ale například přidáte sledování historie v některé třídě a ejhle, databáze se zesložiťuje a dotazy se mění, i když rozhraní databázové vrstvy se navenek měnit nemusí.

Dovedeno do důsledků, objekty databázové vrstvy by se při připojení k databázi měly "porozhlédnout" a zjistit, s jakou verzí databázového schématu "mají tu čest" a podle toho své chování přizpůsobit. To může znít podivně ve světě relačních databází, kde je verzování přece jenom pod kontrolou, ale je to určitě aktuální v případech, kdy se data ukládají v XML, která se pak různě posílají, vracejí,...

Schopnost načíst objekty z různých XML slovníků podle mě bude čím dál častějším zákaznickým požadavkem.

To mě ale přivádí k bodu, který mi není moc jasný. V příkladech, které tu byly uvedeny, se objektu databázové vrstvy posílá zpráva UlozOsobu, která jako své parametry předává hodnoty atributů a cizích klíčů.

Druhý možný přístup je ten, že se objektu databázové vrstvy pošle zpráva Uloz(), kde jediným parametrem bude objekt třídy COsoba, tedy business objekt. Metoda, která obslouží zprávu Uloz() se nejprve tohoto objektu zeptá na jeho třídu, a potom se ho podrobně "vyptá" na jeho atributy a případné asociované objekty (pomocí metod get...()), a tyto údaje pak do databáze uloží na "správné místo".

Výhody: V případě podpory vyvíjejícího se schématu databáze či XML se můžete chovat podle zjištěné verze: vyptáte se na ty atributy, které jsou potřeba, a způsobem, který je specifický pro zjištěnou verzi, je uložíte.

Nevýhody: Vzniká kruhová závislost mezi vrstvou obchodní logiky a vrstvou databázovou. Takže se vlastně jedná o tutéž vrstvu.

Tato nevýhoda by šla odstranit. Objekty business vrstvy nemusí podporovat zprávu UlozSe() - činný slovesný způsob - která je stejně obratem hned převedena na zprávu pro databázový objekt, aby uložil ty a ty hodnoty. Místo toho by všichni zájemci o ukládání objektů business vrstvy toto své přání sdělili přímo databázové vrstvě, nikoliv zprostředkovaně přes business objekt. Databázový objekt "zařídí uložení" nějakého business objektu, který se mu předloží. Konajícím zde však není objekt business vrstvy, ale databázový objekt.

Co si o tom myslíte?

24.2 Odpověď

Myslím, že by se tím narušil základní princip existence business vrstvy a bylo by to zbytečně složité.

Možná by bylo jednodušší získat flexibilitu tím, k čemu je XML vlastně určeno: Zpráva Uloz() bude mít vstupní parametry ve formátu XML a nikoliv ve formátu proměnných daného prostředí. Tím se dostáváme blízko k moderním technologiím na Internetu (SOAP apod.). Potom opravdu volání těchto metod bude flexibilní vzhledem k měnícím se strukturám dat.

25. Značky a sady

25.1 Dotaz

... V objektovém modelovani nejsem (zatím) příliš zběhlý a chtěl bych prodiskutovat následující problém:

Chci udělat objektový model, ve které se bude pracovat s různými sadami entit. Uvedu příklad:

Budu chtít udělat aplikaci na kreslení elektrotechnických schémat. Jde o to, ze bych chtěl mít různé sady schématických značek (základní, rozšířenou aj.). V aplikaci by uživatel mohl používat různé sady těchto značek tak, že by tyto sady značek nějakým způsobem k aplikaci připojil (jakým to je to co právě nevím). Dovedu si zhruba představit tento kousek objektového modelu - budu mít třídu Značka, která bude mít vlastnosti a metody společné všem schématickým značkám (např. název, popis, zkratka).

Pomocí metody Vykreslit by se mohla např. vykreslit na pracovní plochu. Její rozhraní implementují všechny schématické značky. Potom budu moci pracovat s kolekcí značek a bude mi jedno se kterou značkou ve skutečnosti pracuju. Co nevím je to, jak navléknout ono propojení různých sad dodaných k aplikaci. Dále si myslím, že by každá schématická značka měla mít asi svůj formulář pro vyplnění jejích specifických parametrů (vlastností). Nevím jestli je tato úvaha správná. Připadá mi, že je to dost obtížné naprogramovat.

25.2 Odpověď

V každém případě musíte od sebe oddělit grafické prvky a prvky jednotlivých schemat. Nazvěme grafické jako ObrazecNeco a prvky schématu jako PrvekNeco.

ObrazecNeco se umí vymalovat podle svého typu a má za sebou v asociaci PrvekNeco, od něhož si bere údaje pro zobrazení. Edituje se vždy PrvekNeco.

Příklad:

Existuje PrvekOdpor, obsahující pouze hodnotu velikosti odporu a existuje ObrazecOdpor, slovně popsán takto: „Obdélnik se dvěma vrcholy pro připojení (vertexy) s vyobrazením hodnoty odporu“

Sama hodnota velikosti odporu je v objektu PrvkuOdpor, ale zobrazuje se pomocí ObrazecOdpor.

Tedy jinak řečeno, objekt ze třídy CObrazecOdpor jako grafický prvek obsahuje objekt obdélník, obsahuje objekt nápis (uprostřed) a dva vrcholy pro připojení a vidí svůj PrvekOdpor (nápis převezme svůj obsah od property velikost odporu PrvkuOdporu)

Až rozepíšete všechny možné dvojice, zjistite, že můžete řešení vylepšit ještě pomocí návrhového vzoru BRIDGE (viz kapitola Role osoby)

26. Rychlost kolekce

26.1 Dotaz

Při aplikování OO analýzy a návrhu jsme narazili na velký problém s rychlostí filtrů nad kolekcí:

Chceme-li filtrovat kolekci objektů, zavoláme v DB objektu SQL příkaz, který vrátí seznam ID objektů vyhovujících dané podmínce (SELECT ID FROM <table> WHERE <podmínka>) a tento seznam si v kolekci zapamatujeme. Při procházení prvky kolekce pak pro načtení každého prvku voláme znovu SQL příkaz (v DB objektu) na jeho nahrání (SELECT * FROM <table> WHERE ID=<object_id>). Toto sebou nese velikou režii v inicializaci dotazu na SQL serveru a jeho vyřízení. Při standardním postupu (bez použití OOP) by jsme použili pro filtrování seznamu SQL příkaz (SELECT * FROM <table>), který do recordsetu natáhne všechna (nebo vybraná) pole a při procházení seznamem by se již SQL příkaz nevolal - což je mnohem rychlejší. Dá se tato situace nějak elegantně vyřešit bez toho, aby kolekce byla zároveň databázovým recordsetem?

26.2 Odpověď

Máte dvě řešení A a B. Jedno je s kolekcí, která si "zapamatuje" ID, druhé s Recordsetem.

Z hlediska "objektovosti" není vůbec důležité, které z nich zvolíte, ale podstata je v něčem jiném.

První řešení, které popisujete (se zapamatovanými ID) je čisté proto, že toto zapamatování je schováno v kolekci uvnitř ní (ona si to pamatuje). Jedná se o nějaký datový útvar uvnitř kolekce, tedy její vnitřnosti. Nikdo se neptá přímo tohoto útvaru, ale kolekce, protože ona je správcem seznamu svých prvků a daná ID jsou pouze pomocníkem ("jakoby index")

Pokud dovnitř kolekce neschováte tyto zapamatované údaje ID, ale celý Recordset, pak máte stejné řešení a je stejně objektově čisté, přičemž je to vámi zmiňovaná varianta B. Znamená to, že není nic proti ničemu z hlediska OOP, pokud uvnitř kolekce "žije" Recordset. Kolekce jej "obaluje" a nikdo recordset přímo nepoužije ve smyslu, že by obešel kolekci. Kolekce má svůj příkaz "Refresh", který není nic jiného, než obnovení Recordsetu. Důležité je dodržet zásadu: Pokud chci pracovat se seznamem a jejími prvky, musím jít přes kolekci a ne přímo do recordsetu, to jsou pouze "data". Tedy uvnitř kolekce se kolekce obrací na data z tohoto RS.

V projektech s ADO často používáme tu variantu, že uvnitř kolekce žije RS, který je po naplnění (Mycol.Refresh) odpojen od DB a používáme na něj property Filter a Sort. Je to překvapivě rychlé.

Navíc pokud může dané prostředí GUI pracovat s recordsetem jako Datasourcem, potom můžete dokonce pro GUI prvek, který umí zobrazit recordset, požádat kolekci o tato data pro zobrazení (dej data jako rs):

```
...
Mycol.Refresh 'obnov data
Set MyDB_GUI_Prvek.Datasource   = Mycol.rs 'dej data, zobrazim ve VB 6.0
...
```

Důležité je však vždy požádat kolekci. Chybou z hlediska objektového návrhu je, aby přímo formulář provedl konstrukci SQL, provedl dotaz, naplnil tak RS a dosadil jej do GUI prvku. To se funkcionality aplikace rozprsknou do formulářů a aplikace nebude vývojově stabilní.

27. Volání vlastního Property

27.1 Dotaz

Je deklarována třída, třída má vlastnost jméno. Fragment kódu vypadá následovně.

```
==================================================
Private mvarJmeno As String

Public Property Let Jmeno(ByVal vData As String)
      mvarJmeno = vData
End Property

Public Property Get Jmeno() As String
      Jmeno = mvarJmeno
End Property
==================================================
```

Dále je deklarována metoda poskytující normalizované jméno (např. jméno převedená na velká písmena). V podstatě lze kód napsat buď

```
==================================================
Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(Jmeno)
End Function

NEBO

Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(mvarJmeno)
End Function
==================================================
```

Co je "správné"? Oba postupy dají stejný výsledek za předpokladu, že Property Get je napsána jako výše. První způsob má jeden krok navíc (musí se vyzdvednout mvarJmeno pres property Jmeno). Na druhou stranu je tu jistota, že pokud později někdo udělá zásah do kódu Property Get např. takový, že změní kód takto:

```
==================================================
Public Property Get Jmeno() As String
      Jmeno = NějakáOperace(mvarJmeno )
End Property
==================================================
```

tak už oba způsoby povedou k jiným výsledkům. To je logické.

Otázka zní:
Je tento kód

```
??????????????????????????????????????????
Public Property Get Jmeno() As String
      Jmeno = NějakáOperace(mvarJmeno )
End Property
??????????????????????????????????????????
```

formálně přípustný?

```
Důsledky jsou již jasné.
Pokud tento kód není formálně přípustný, tak je sice horší ReUse (funkci
NějakáOperace budu aplikovat na mvarJmeno ze 2 míst, jednou při vyzdvednutí
Jmeno, jednou při vyzdvednutí JmenoNormalizovane = při opravě kódu je to
potenciální zroj nestability, protože se to určitě opraví jenom na jednom
místě), ale nemá valný smysl zabývat se rozdílem v kódu
================================================
Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(Jmeno)
End Function

VERSUS

Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(mvarJmeno)
End Function
================================================

Pokud tento kód je formálně přípustný, tak je jasné, že potom by se mělo
používat

Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(Jmeno)
End Function
```

27.2 Odpověď

Váš dotaz směřuje k základní obecné otázce:

V kódu metody je použit atribut, má se tento atribut „číst“ přímo nebo přes property? Druhý případ odpovídá tomu, že objekt volá „sám sebe“ (volá své property) při běhu dané metody.

Pro vysvětlení je třeba připomenout, že volání property je volání metody. Nic víc totiž v syntaxi property nehledejme. Tedy odpověď na tuto obecnou otázku je zřejmá: potřebuji funkcionalitu schovanou za property anebo ne? Pokud ano, potom property použiji, pokud ne potom nikoliv. Toto rozhodování je úplně stejné jako u každé jiné metody!

Příklad:

Představme si, že v objektu je nějaká metoda, která něco umí, například verifikaci něčeho a tato se dá zavolat „zvenku“. Tedy můžeme zavolat objekt Verifikuj_Něco().

V běhu jiné metody potřebujeme tutéž verifikaci. Samozřejmě uvnitř objekt metodu Verifikuj_Něco() v běhu této nové metody použije. Nad tím váhat nebudete. Ale s property je to úplně stejné, protože property je také metoda jako každá jiná. Například v Javě by vám tento dotaz vůbec nevznikl.

To, co vás tak trochu zmátlo, je „objekt volá sám sebe“ a chcete uvnitř něj porvádět znovu „druhou“ encapsulaci. Pokud se nacházíte uvnitř objektu (navrhujete jej, viz e-kniha OM a UML v praxi 2000), potom je máte jako tvůrce vnitřně rozkuchán (zvednutá kapota auta). Jediné kritérium pro rozhodnutí mezi variantami:

```
Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(Jmeno)
End Function

VERSUS
```

```
Public Function JmenoNormalizovane() as String
      JmenoNormalizovane= UCase(mvarJmeno)
End Function
```

je: „... a potřebuji v `JmenoNormalizovane` funkcionalitu metody `Jmeno` nebo ne“? Zdůrazňuji metody `Jmeno`, i když je to property. O tomto se mohu zde plnohodnotně rozhodovat, protože jak metoda `Jmeno`, tak metoda `JmenoNormalizovane` jsou nyní ve stejném informačním prostoru téhož objektu, jsou současně kódovány a rozbaleny k tvoření a můžete kombinovat jak chcete.

28. Agregace, běžná asociace

čtu Vasi knihu "O. mod. a UML v praxi 2000" a zejména kapitolu "Model trid, Class model". Jsem úplný začátečník v OOP i UML. Možná proto mi Váš popis tvorby tohoto modelu přijde pro mne příliš stručný. Předpokládám, že nemůžete odpovídat všem, kteří četli Vaši knihu na jejich dotazy ke "konkrétním problémům s OOP a UML". Hlavně mi není úplně jasný způsob rozlišení asociací mezi objekty na "běžnou asociaci" a "agregaci". Např. příklad s barvou auta. Podle mého je barva přímo fyzickou součástí auta, tedy vztah celek - část, a proto agregace. Myslim, ze zmatek je způsoben tím, ze používáme objekty v kombinaci s relační databází. Jinak by přece každé auto mělo svou vlastní barvu...tak jak je to mezi TOsoba a TAdresa na str. 131.

Mám pocit, že z hlediska OOP je nesmysl, aby za barvu určitého objektu byl odpovědný jiný objekt (v tomto případě CBarvy = číselník). To je jako kdybychom se podívali na auto, ale jeho barvu bychom nebyli schopni vidět přímo, ale našli bychom na aute odkaz na někoho, kdo nám barvu tohoto auta sdělí ... to ale není popis současné reality.

Za odpověď předem děkuji,

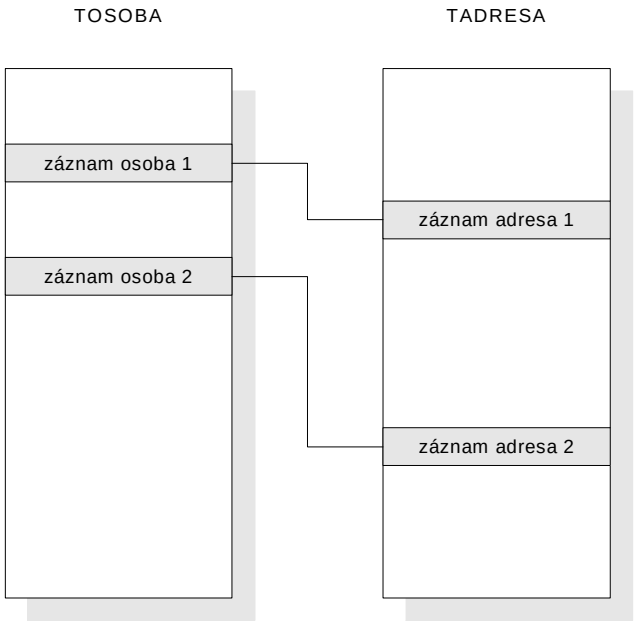
28.1 Odpověď

Jak píšete, jste zvyklý na relační databáze a proto zahájím vysvětlení tam. Protože mezi modelem tříd a relační databází existuje přímý vztah (viz ekniha – mapování tříd do ERD), můžeme si ukázat, jak se projeví běžná asociace a agregace přímo v datech a potom přejdeme k objektům.

Takže v datech jako první věc, co uděláme, přijmeme požadavek, že k vazbě mezi tabulkami dochází zásadně pomocí cizího klíče tvořeného systémovým klíčem (například automatické číslo jako autoincrement, IDENTITY apod. anebo jako generované unikátní číslo jako GUID) Toto číslo zprostředkující vazbu jako prim,ary a foreign klíč budeme slangově nazývat, jak bývá zvykem, jako ID resp. IDečko. Takto realizovaná vazba je totiž analyticky mnohem čistší (odděluje problém vazby od samotné analytické reality) a má velmi blízko k objektovému náhledu, jak si ukážeme.

Vezměme si jako příklad nejprve Adresy. Dlouhou dobu jsem se domníval, že Adresy jsou klasickým příkladem „agregace“, protože tak jsem se s nimi setkal v několika systémech.

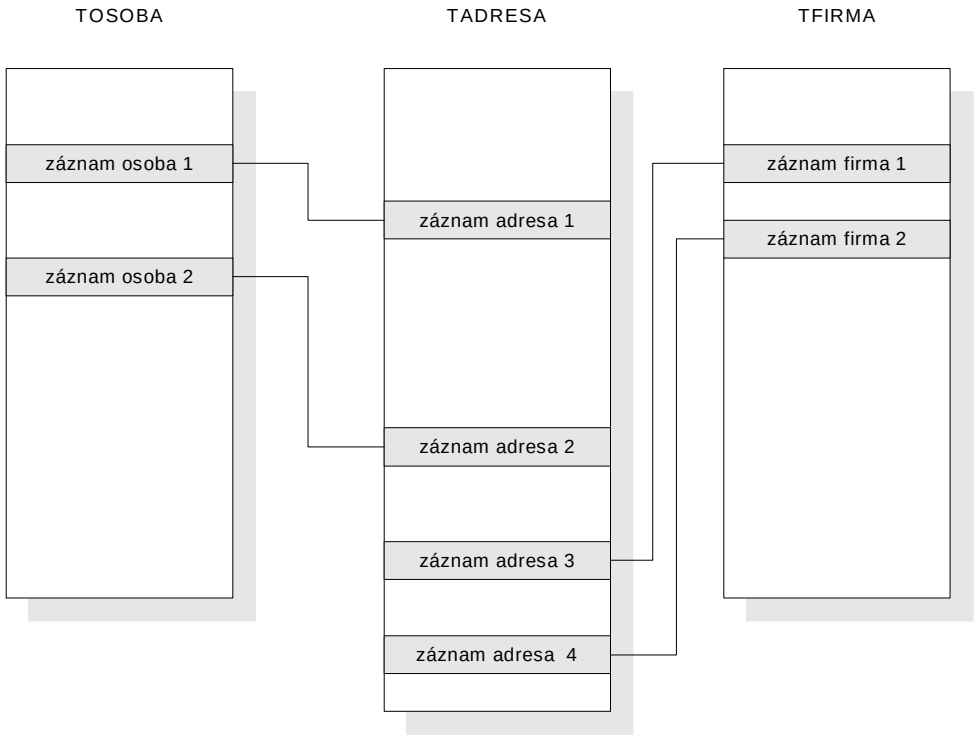
Takže agregace Adresy. Mějme tabulku TOsoba a Tadresa (zapomeňme teď na objekty). Jsou provázány tak, že cizí klíč ID_ADRESA je v tabulce Tosoba (pozor, nikoliv naopak!). Tosoba má tedy cizí klíč adresy. Pro agregaci je charakteristické chování záznamů (nikoliv struktury!). Pokud někdo založí adresu jako nový záznam, tak spolu s ním se vloží „fungl“ nový záznam adresy. Pokud někdo vloží další záznam osoby, tak i ona bude mít nový záznam adresy a to nezávisle na tom, zda obě osoby bydlí na téže adrese. Pokud bydlí na stejné adrese, dojde ke shodě hodnot v těchto dvou záznamech. Situaci osvětluje tento obrázek:



Můžeme si představit, že každý nový záznam osoby vyžaduje nový záznam adresy. To je charakteristické pro agregaci.

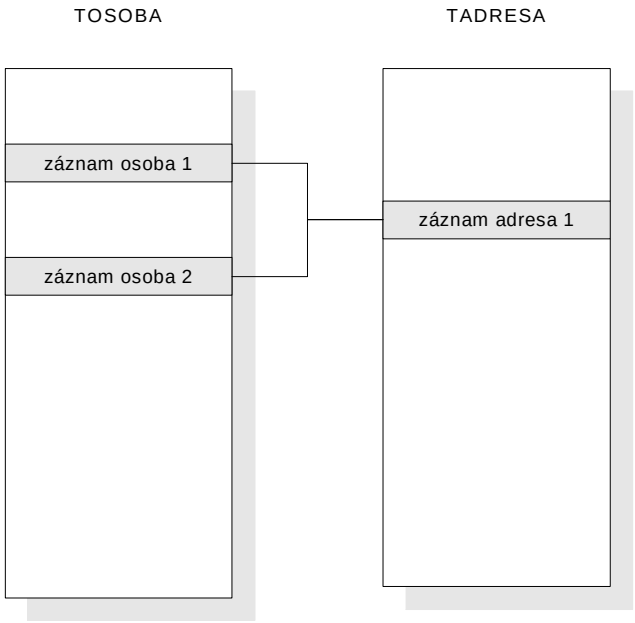
Všimněme si, že v tabulce osoby zůstalo jenom ID adresy a nic víc. Sama osoba není vůbec zodpovědna za to, jak vlastně adresa vypadá. Osoba jenom ví, že adresa existuje a která je ta právě její (díky ID).

Pokud vám není jasné, proč ID putuje právě tímto směrem od adresy do osoby, tak je to proto, že adresa je použita i v jiných částech systému, například pro sídlo firmy:



Takto zvolené řešení má své výhody a nevýhody: Výhodou je snadné naprogramování a snadné zadávání nové adresy (není na nic vázáno), nevýhodou je datový objem (redundance adres), ale také analytická nepřesnost vyplývající z možného zadávání adres libovolně. Například někdo napíše náměstí TGM, druhý napíše nám. TGM a máme dvě různé adresy.

Druhá možnost je zavést běžnou asociaci. Znamená to, že existují adresy „zvlášť“ jako číselník. Pokud dvě osoby mají stejnou adresu, potom je vyžadována shoda jejich ID adresy, tedy musí dojít ke shodě záznamů, tedy poud dvě osoby bydlí na stejné adrese, nastane shod ajejich cizích klíčů a tedy shoda záznamů v adrese:



Všimněme si, že z hlediska datových vztahů se nic nemění (cizí klíče jsou stejně položené). Rozdíl je ve funkcionalitě, v druhém případě musí obsluha při zadávání adresy „nějak“ (rozumně) vybírat, například nejprve město, potom ulice atd.

V tom je základní rozdíl mezi běžnou asociací a agregací. Podobně u aut můžeme zvolit agregaci, což by znamenalo každému autu dle libovůle zadat barvu (někdo napíše dozelena, někdo zelenkavá apod.). Pokud však existuje číselník barev, musí být vybrána barva.

Pokud datové vazby přes ID zaměníte za vazby mezi objekty (objektové reference) a přidáte každému záznamu objektovou funkcionalitu (interface a metody), převedete problém do roviny OOP.

K vaší závěrečné poznámce o tom, jak je to s tou zodpovědností: V obou případech jak v agregaci, tak v běžné asociaci auto ví jenom to, že má barvu a to buď dodanou z číselníku, anebo agregovanou. V datech to znamená, že má ID barvy a to je vše. Auto „neví nic“ o struktuře barvy (zda má kód, text nebo co vlastně má...). Auto není za barvu vůbec odpovědno, za barvu je odpovědna barva (nikoliv číselník, to je správce všech barev jako svých prvků – itemů, ví jejich počet apod.) .

29. Problémy s přechodem na OOP a UML ve firmě

29.1 Dotaz

Mám v naší firmě dost velké problémy, jsem velkým zastáncem OOP a UML a objektového modelování obecně, ale vedení není ochotno přejít na tyto nové technologie V jednom projektu mi šéf dokonce jako vedoucímu projektu dal možnost použít tyto technologie, ale termíny byly natolik našponované, že projekt skončil neúspěchem. Teď někteří mí „kolegové“ škodolibě argumentují tím, že strukturálně (a navíc bez analýzy atd.) by tento projekt byli schopni zvládnout rychleji než pomocí OOP a UML a nezdar přisuzují mě. Máte podobné zkušenosti a jak takovou situaci řešit? Sám mám pochybnosti, že strukturálně by se zvládl rychleji, ale jak to dokázat? Rád bych vedení nějak dokázal, že tento projekt svůj čas potřebuje, ale nejsou ochotni mne poslouchat (vidí jenom svoje obchody...). Co byste mi poradil?

29.2 Odpověď

Nejprve k dotazu, zda jsem se již setkal s takovou situací: Ano a dokonce velmi často, není to žádný ojedinělý jev. „Šťastná firma“, která tento problém nemá! Největší problém přechodu na OOP a UML ve firmách není v pracovnících nižší úrovně, ale ve vedení. Z toho důvodu vznikla také doprovodná kniha o řízení projektů určená právě těmto vedoucím pracovníkům (viz náš server). Mnoho vedoucích se domnívá, že pro přechod na OOP je postačující vyškolit pracovníky, ale že oni sami žádné školení nepotřebují. Omyl.

Přechod na OOP má jedno riziko: OOP není jednodušší a „méně pracné“ než strukturální programování a dokonce v počátcích je dokonce pracnější (viz článek o synergii v projektech). Protože je OOP postaveno na maximálním re-use, tak se jeho výhody neprojeví, pokud se firma nepřevede do stavu, kdy je schopna tento re-use využít. Jste ve firmě sám, kdo pracuje objektově? Pak můžete sice vytvořit skupinu tříd, dát je dokonce do komponenty, například v .NET do assembly ... ale ruku na srdce, kdo jiný ji použije, než vy? Vaše práce byla v tomto případě zbytečná, protože ji nikdo jiný nepoužije. Pokud neexistuje centrální knihovna, tak jste sám voják v poli a vaše OOP zůstalo tam někde trčet s vámi...

Ke srovnání rychlosti tvorby dvou projektů: Pokud má někdo před sebou dva projekty téhož problému, z nichž

- první nemá žádnou dokumentaci, žádnou analýzu, má pouze nepřehledný strukturální a narychlo slepený chybový kód
- druhý je zdokumentovaný, má oddělené abstraktní úrovně podle normy ISO v modelech UML, je čitelný a transparentní psaný v OOP

tak mu v prvním (a hloupém) přiblížení vyjde, že druhý projekt je „pracnější a tedy nákladnější“.

Je to však tentýž přístup, jako kdybychom zkoumali kostku, zda je falešná a přitom když by padlo jiné číslo než 6, tak bychom tento hod prostě nezapočítali. To se potom dojde k zajímavým závěrům: „Protože na kostce padají pouze šestky, kostka je falešná“, ale o selekci hodů ani zmínka. Připadá vám toto přirovnání přitažené za vlasy?

Takže co jsme u porovnání nákladů u projektů nezapočítali: Kde se u prvého projektu započítaly náklady na nutnou mnohem vyšší údržbu programu (který je nečitelný a chybový), kde je započítána chybovost díky chybějící analýze a sporům s uživatelem, že „takto to nechci“ a kde jsou náklady na následné předělávky, kde jsou započítány ztráty způsobené nízkou kvalitou a tedy rizikem ztrát zakázek, kde je započítáno riziko, že když pracovníci tvořící tento program odejdou z firmy nebo prostě odejdou na jiný projekt, tak není jak a co předat jiným pracovníkům, kde jsou náklady na tvorbu uživatelské dokumentace, která se v případě modelování provádí mnohem snadněji, kde jsou náklady spojené s malou flexibilitou programu (malá změna = obrovské úsilí), kde jsou započítány náklady spojené s velkým nutným úsilím při nesystematickém řízení chaotického projektu (přesčasy apod.), kde je započítáno srovnání nákladů spojených s minimální opětovnou použitelností již hotových částí systémů atd.

Pochopitelně, pokud zanedbáme všechny tyto faktory a zůstane nám pouze holá skutečnost, že se něco odevzdalo (a že se to možná i podobá se tomu, co chceme), tak nám vyjde výhodnější postup jako prvý, bez dokumentace, bez modelování a bez abstraktních úrovní, bez systematického řízení projektu.

Současný moderní trend tvorby SW však používá druhý přístup (vedoucí ke kvalitě SW), přitom v tomto druhém postupu se provádí výroba SW velmi rychle (pokud je firma na tento postup nastavena). Při tomto druhém kvalitním postupu je projekt zdokumentovaný, má oddělené abstraktní úrovně podle normy ISO v modelech UML, projekt je v každé fázi v modelech a v kódu čitelný a transparentní, napsaný přehledně v OOP. To je myslím jasné a probrané dost podrobně v e-knihách o objektovém modelování a o řízení projektů v OOP prostředí.

Problém je v tom, že mnohde vedení těmto celosvětově již přijatým argumentům nenaslouchá (viz například normy ISO pro tvorbu SW).

Jak z toho ven? Takže velmi praktické rady vyplývající z mých zkušeností z dlouholeté konzultační praxe jsou tyto:

První rada, kterou bych doporučoval, je přímo změna vašeho postoje. V prvé řadě musíte odosobnit problém. Vůbec není smyslem někomu něco dokazovat. Doporučuji na dokazování zapomenout, protože to vede automaticky k atmosféře vzájemných argumentů typu: „Vy to děláte blbě...“ Tento přístup je třeba vypustit a to v každém případě, tedy i kdyby to bylo tisíckrát pravda. Jinak řečeno myšlenku „proboha, jak vy to ale děláte špatně“ si před vaším okolím ani nepřipouštějte! Změna vašeho postoje musí být taková, že vaše okolí a tedy i vedení postupně dojde samo k závěrům, které vy potřebujete a hlavně, tyto závěry nebudou už potom od vás, ale od nich (Sokratova zásada diskuse). Smiřte se DOPŘEDU s tím, že vavříny budou na jiné hlavě. Sledujte pouze svůj cíl (aby to ve firmě bylo takto, protože to chcete, a to je cíl, nikoliv to, že dokážete, že máte pravdu). Nakonec výsledný stav ve firmě má na starosti vedení a nikoliv vy, takže toto smíření se s osudem „vařiny jinému“ není až tak bolestivé. Chce to hlavně hodně asertivity. Tedy změňte postoj na pozitivní směr: Dělejte vše k tomu, aby oni sami přijali závěry, které vy potřebujete, nesmíte však přitom „poučovat“. Dopředu se smiřte s tím, že zásluhy nebudou v tomto případě vaše, to je asi jasné. Cíl a motiv této rady: Zaveďte ve svém konání positivismus. Rozbijte a zničte atmosféru dělící tým na „vy a my“, kde „vy to děláte blbě“ a „my víme, jak to dělat dobře“, přičemž obráceně si totéž myslí i ta druhá skupina. Jakmile pocítíte atmosféru „my a vy“, musíte hrany takovýchto diskusí ihned zaoblit a přejít na pozitivní atmosféru.

Druhá rada: Více poslouchejte než hovořte. Vaše kroky musí být hodně promyšlené a mít povahu spíše dobře připravených návnad než předložených a rozbalených „vašich“ informací. Na vámi předložených a rozbalených „postupech, jak to dělat“ se budou hledat jenom mouchy a nikdo je nepřijme za své. Nechte také něco rozbalit svého šéfa a nasměrujte ho tak, aby „to rozbalil dobře“. Přejděte tedy do polohy „technologického vezíra šéfa“ a nikoliv „revolucionáře bortícího jeho iluze“. Pokud vám to zní jako podlézání, tak to je omyl a ještě jednou: Odosobněte se! Nebavíme se teď o vašich odměnách, ale o nové technologii ve firmě! Nezapomeňte přitom, že zavádění nové technologie je jeho parketa a nikoliv vaše, a přitom vy mu do toho chcete „kafrat“ (a to je práce pro pyrotechnika). Najděte v jeho výkladu styčné body, které potřebujete a tam podvrhněte správnou myšlenku. Napovězte, aby pokračoval vaším směrem. Správná věta: „To je dobrý nápad, taky jsem o tom četl, to by bylo opravdu zajímavé, ...“ a předložit kudy pokračování. V každém případě proces jeho poznání musí skončit tím, že „je to jeho nápad“. Vás si bude vážit nikoli proto, že předkládáte konečná řešení, ale že vás má vždy po ruce. Cíl této rady: Myšlenky, jak řídit firmu a zavádět OOP technologii, nejsou vaše, ale odpovědného vedoucího, protože on je vedoucí. Poznámka: Pokud chcete tuto technologii zavádět ve firmě sám osobně, tak v tom případě se staňte sám šéfem, :)

Třetí rada: „Odstraňujte“ opozici tím, že z nich uděláte spojence. Totéž, co platí vůči šéfovi, platí i vůči vašim kolegům. Zapomeňte na pozici „světlonoše nesoucího vždycky pravdu“. Takoví pracovníci jsou ve firmách vždy odsouzeni na odstřel. Postupně infikujte ostatní myšlenkami, rady dávejte nikoliv mentorské, ale nenápadně, příkladem apod.

Čtvrtá rada: Jaká je potom vlastně moje pozice v týmu? Nyní je vaše posice „já jsem ten, kdo umí, nebo se intenzivně věnuje OOP a UML a řízení projektů a ostatní to neumí a neznají“. Paradoxně nastává situace, kdy se snahou toto dokázat tento stav pouze více a více utvrzuje a jste tak sám proti sobě. To je chyba. Vžijte se do role, že vaše pozice bude v budoucnu tato: „Všichni umíme OOP a UML, někdo víc a někdo míň, a přitom postupně ostatní uznávají, že já ve firmě patřím ke špičce těch druhých, tj. těch, kteří to umějí...“ S touto myšlenkou zahajujte každou diskusi, i kdyby váš kolega nikdy o OOP a UML neslyšel. I on má právo být pochválen...

Pátá rada: Snažte se vytvořit atmosféru společně všeobecně pozitivních zážitků při zavádění OOP a UML technologie. Například pokud existuje nějaké školení apod., jeďte tam vy, kolegové, ale i váš šéf. Samozřejmě jako konzultant doporučuji své školení OOP a UML, viz server http://www.objects.cz . Ale právě v tomto vašem případě bych považoval za velmi účinné využít možnosti rekreačně – školícího dvoudenního pobytu pro zaměstnance SW firmy. Jedná se o dvoudenní pobyt zaměstnanců (nejlépe včetně šéfů) ve středisku na rekreační chatě s ubytováním v chatkách i s možnostmi rehabilitačního zařízení, takže je v tom schována i odměna pro zaměstnance. Dopoledne se provádí školení OOP a UML také se zaměřením na zásady řízení projektů (včetně diskuse). Tam dochází ke zmíněnému „prvnímu infikování OOP a UML“ v příjemné atmosféře a co je důležité – je tam možnost, aby vdení

přijalo tyto myšlenky jako své. V pozdní odpoledne a večer následuje rekreace a rehabilitace. Velkou výhodou tohoto školení je neformálnost a dobré prostředí vhodné k nastartování správné tvůrčí atmosféry při přechodu na OOP a UML ve firmě a to i mezi vedoucími. Samozřejmě buďte opatrní, jak přivést šéfa k této myšlence. Chybná je formulace: „Šéfe, teda vy byste fakt potřeboval školení jako sůl!“ Jak vyplývá z předešlých rad, nejlepší by bylo, kdyby s touto myšlenkou přišel on sám. Lze například šéfovi poslat nabídku „školení s rekreačním pobytem“ a vás přitom nezmiňovat (viz http://www.objects.cz , je možné požádat o nabídku přes mail mailto:objects@objects.cz). Při těchto školeních samozřejmě nesmí vzniknout atmosféra typu: „Tak vidíte, a kdo teda měl pravdu, já nebo vy...?“ Porušíte tak zásady uvedené v předešlých odstavcích (zmíněná věta je konfrontační) a vaše úsilí je zmařeno.

Šestá rada: Dodržujte zásadu ***„little start, fast grow“***. Začít s málem, ale stále přidávat více a více. Tato zásada by vás měla vyvarovat megalomanských představ, že v krátké době bude firma vypadat podle vašich představ. Přerod firmy je vždy postupný proces a neděje se pomocí „čínských skoků“. Velké změny jsou mnohem riskantnější a mají větší pravděpodobnost, že povedou k fiasku. Zavádějte postupně jeden konkrétní krok za druhým. Podařilo se v analýze zavést zatím povinně pouze USE CASE model? Výborně, už to je úspěch! Tato zásada vám umožňuje zavádět věci „po malých soustech“ a to dokonce velmi konkrétně. Pokud máte před sebou příliš „velké sousto“, nevíte ani odkud „se zakousnout“.

Uvedená doporučení samozřejmě nemusí vést vždy k pozitivnímu výsledku, ale jejich porušení je určitě velmi dobrou brzdou k jejich dosažení, to mohu potvrdit z vlastní zkušenosti.

Tedy dobře, řekente si, doporučení přijata, ale odkud tedy začít? Vzpomeňte na šestou radu, tedy *„malý start, rychlý růst“*: Lze například zahájit uvedeným dvoudenním firemním školením OOP a UML s rekreačním pobytem.

Nakonec abych nezapomněl, jedna rada poslední, ale je nejdůležitější, která by vás měla naplnit optimismem: V celém tomto procesu popsaném v předešlých odstavcích se můžete mnohému naučit a vzdělat se, například v oblasti OOP, UML, řízení projektů apod. Ať se stane cokoliv, třeba s firmou, s kolegy a s vaším zaměstnáním a místem, tak to co máte v hlavě, vám nikdo nikdy nesebere...

KONEC DOKUMENTU