VYSOKÁ ŠKOLA EKONOMICKÁ V PRAZE
Fakulta informatiky a statistiky
Katedra informačních technologií

Borek Bernard

Vývojové prostředí pro návrhové vzory v C#

Vedoucí diplomové práce:	Ing. Alena Buchalcevová, Ph.D
Recenzent:	Ing. Jan Stuchlík
Studijní program:	Aplikovaná informatika
Obor:	Informační systémy a technologie

Praha, 2006

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem čerpal.

V Praze dne 28. srpna 2006

Podpis

Poděkování

Za vlídný přístup i cenné připomínky bych chtěl poděkovat vedoucí práce, paní ing. Aleně Buchalcevové, Ph.D. Můj vděk za konzultace ohledně některých témat rovněž patří panu René Steinovi.

Abstrakt

Hlavním tématem práce je problematika podpory návrhových vzorů ve vývojových prostředích. V dnešní době existuje dostatek materiálů, které vzory popisují, kategorizují nebo analyzují, současně však chybí plnohodnotná podpora ze strany nástrojů, která by umožnila vzory používat jako reálné artefakty softwarového vývoje. Hlavním cílem práce tak bylo prozkoumat možnosti nástrojové podpory a přinést vlastní vizi „vývojového prostředí pro návrhové vzory“.

Práce je rozdělena do čtyř hlavních tematických celků. První zkoumá možnosti formálního zápisu vzorů, druhý mapuje úroveň podpory návrhových vzorů v dnešních CASE nástrojích, třetí popisuje vizi softwarových továren a doménově specifických jazyků, zatímco poslední shrnuje předchozí poznatky a přináší koncept vývojového prostředí pro návrhové vzory založeného na Visual Studiu 2005.

Představený koncept je tříúrovňový a podporu vzorů řeší jak na úrovni diagramů, tak na úrovni zdrojového kódu. V kódu jsou návrhové vzory explicitně zachyceny pomocí .NET atributů, vizualizaci kódu obstarává Class Designer a návrhové vzory jako takové přehledným způsobem zachycuje speciálně navržený doménově specifický jazyk. Práce bohužel vznikala v době, kdy nástroje pro doménově specifické jazyky v rámci Visual Studia byly v ranné a nestabilní fázi svého vývoje, takže naznačený koncept mohl být realizován pouze částečně. Kromě něj práce přináší i některé další myšlenky týkající se podpory návrhových vzorů ve Visual Studiu a jejich integrace do běžného vývojového procesu.

Klíčová slova

Návrhové vzory; Vývojové prostředí (IDE); Formální zápis vzorů; UML; MDA; Softwarové továrny; Doménově specifické jazyky (DSL); Domain-Specific Language Tools (DSL Tools); Guidance Automation Extensions (GAX); Guidance Automation Toolkit (GAT); Smart Client Software Factory (SCSF); Visual Studio 2005

Summary

The main topic of this thesis is design patterns and their support in integrated development environments (IDEs). Currently, there are many good sources that describe, categorize or analyze patterns but there is some lack of full-fledged support from tools which makes it hard to use design patterns as first-class software development artifacts. Therefore, the main goal of this thesis was to explore different approaches to patterns support in software tools and to present a new vision of "integrated development environment for design patterns".

The thesis consists of four main parts. The first one deals with design patterns formalization, the second one explores the current state of patterns support in existing CASE tools, the third one describes the vision of software factories and domain-specific languages, while the last part brings in a new concept of integrated development environment for design patterns implemented on top of Visual Studio 2005.

This concept has three layers and focuses on both diagram view and code implementation. On the code level design patterns are explicitly denoted using .NET attributes; the code visualization is provided by built-in Class Designer; and the high-level visualization of design patterns and their relationships is depicted using a new domain-specific language. Unfortunately, the thesis was written in times when the domain-specific languages tools for Visual Studio were unstable and virtually unusable so the concept could be implemented only partially. Besides this concept, the thesis brings in some other thoughts related to design patterns support in Visual Studio and their integration with standard software development process.

Keywords

Design patterns; Integrated Development Environment (IDE); Design patterns formalization; UML; MDA; Software Factories; Domain-Specific Languages (DSLs); Domain-Specific Language Tools (DSL Tools); Guidance Automation Extensions (GAX); Guidance Automation Toolkit (GAT); Smart Client Software Factory (SCSF); Visual Studio 2005

Obsah

Prohlášení i
Poděkování ii
Abstrakt iii
Klíčová slova iii
Summary iv
Keywords iv

1 Úvod 1
1.1 Téma práce 1
1.2 Hlavní cíle 2
1.3 Historický kontext (důležité upozornění) 2
1.4 Obsah práce 2
1.5 Přínosy práce 3

2 Úvod do návrhových vzorů 4
2.1 Zasazení návrhových vzorů do kontextu softwarového vývoje 4
2.2 Problémy současného stavu 5

3 Formální zápis návrhových vzorů 7
3.1 Potřeba formálního zápisu 7
3.2 Námitky proti formalizaci 7
3.3 Přístupy k formalizaci vzorů 8
3.4 (Ne)vhodnost konvenčních programovacích jazyků 8
3.5 (Ne)vhodnost objektových notací 9
3.6 (Ne)vhodnost formálních specifikačních jazyků 10

4 Přehled samostatných formalizačních technik 11
4.1 LayOM – Layered Object Model 11
4.1.1 Hlavní ideje 11
4.1.2 Příklad 12
4.1.3 Další vlastnosti LayOM 13
4.1.4 Shrnutí, výhody, nevýhody 13
4.2 Zachycení vzorů v programovacím jazyce pomocí atributů 13
4.3 DisCo 14
4.4 LePUS – Language for Patterns Uniform Specification 15
4.4.1 Hlavní ideje 15
4.4.2 Shrnutí, výhody, nevýhody 16
4.5 PDL – Patterns Detection Language 16
4.5.1 Úvod 16
4.5.2 Hlavní ideje 17
4.5.3 Detekce vzorů 18
4.5.4 Shrnutí, výhody, nevýhody 18
4.6 Diagramy omezení a třívrstvé modelování 19
4.6.1 Hlavní ideje 19
4.6.2 Příklad 20
4.6.3 Shrnutí, výhody, nevýhody 21
4.7 Shrnutí 21

5 Formalizační techniky založené na UML ... 22
5.1 Zachycení vzorů pomocí Vennových diagramů ... 22
5.2 Využití notace pro kolaboraci ... 23
5.3 Anotace pomocí zápisu „vzor:role“ ... 24
5.4 Anotace pomocí tagovaných hodnot ... 24
5.5 Shrnutí ... 26

6 Podpora návrhových vzorů v současných nástrojích ... 27
6.1 Podpora vzorů v CASE nástrojích ... 27
6.2 Speciální nástroje ... 30
6.3 Jiné přístupy ... 32
6.4 Podpora vzorů ve vývojových nástrojích ... 33
6.5 Shrnutí ... 33

7 Softwarové továrny ... 34
7.1 Diskuse o roli UML ve světě softwarového vývoje ... 34
7.2 Softwarové továrny ... 38
7.2.1 Vize ... 38
7.2.2 Konkrétní podoba softwarových továren ... 39
7.3 Doménově specifické jazyky (DSLs) ... 40
7.4 Podpora softwarových továren ve Visual Studiu 2005 ... 42
7.4.1 DSL Tools ... 42
7.4.2 Guidance Automation Extensions (GAX) a Guidance Automation Toolkit (GAT) ... 44
7.5 Příklady softwarových továren ... 45
7.5.1 HL7 Software Factory ... 45
7.5.2 Smart Client Software Factory (SCSF) ... 45
7.5.2.1 Úvod ... 45
7.5.2.2 Stručný přehled Composite UI Application Blocku (CAB) ... 46
7.5.2.3 Obsah SCSF ... 49
7.5.2.4 Práce se SCSF ... 50
7.5.3 Další softwarové továrny od skupiny patterns & practices ... 52
7.5.4 Enterprise Framework Factory ... 52
7.5.5 Project Glidepath ... 53
7.6 Problémy současných softwarových továren ... 54

8 Vývojové prostředí pro návrhové vzory ... 55
8.1 Možnosti podpory vzorů ve vývojových prostředích ... 55
8.2 Visual Studio jako vývojové prostředí pro návrhové vzory ... 56
8.3 Clipcode-GoF-DSL: doménově specifický jazyk pro návrhové vzory ... 58
8.4 Vlastní přístup k podpoře vzorů ve Visual Studiu ... 61
8.4.1 Koncept ... 61
8.4.2 Úroveň 1: Kód ... 62
8.4.3 Úroveň 2: Class Designer ... 64
8.4.4 Úroveň 3: Doménově specifický jazyk pro návrhové vzory ... 66
8.5 Shrnutí ... 67

9 Shrnutí a závěr ... 68
9.1 Stručné shrnutí jednotlivých kapitol ... 68
9.2 Shrnutí konceptu vývojového prostředí pro návrhové vzory ... 68
9.3 Možnosti další práce ... 69
9.4 Závěr ... 69

Zdroje ... 70
Webové stránky a další zdroje z internetu ... 72

Terminologický slovník 75
Seznam obrázků a tabulek 77
Příloha A – korespondence ohledně LayOM 79
Příloha B – Výpis kódu pro definici uživatelského atributu 81
Příloha C – Výpis kódu abstraktní továrny realizované pomocí abstraktních tříd 82
Příloha D – Výpis kódu abstraktní továrny realizované pomocí rozhraní 84

1 Úvod

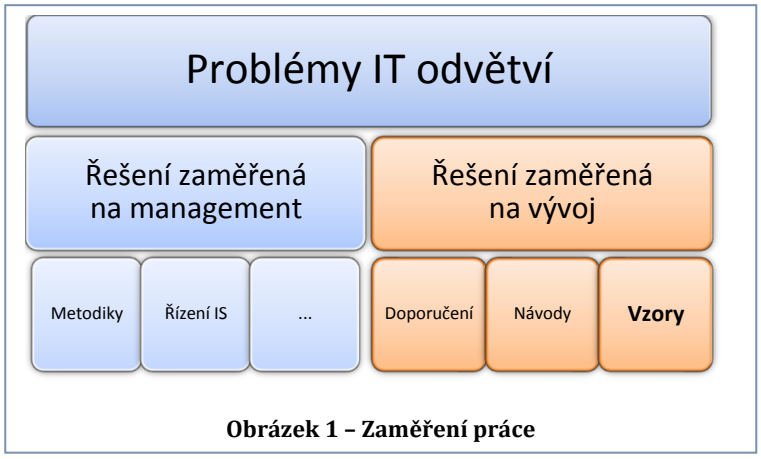
1.1 Téma práce

V posledních letech se hodně mluví o neutěšeném stavu v oblasti softwarového inženýrství. IT projekty jsou drahé, málokdy jsou hotové v termínu a není výjimkou, že výsledek není k úplné spokojenosti zákazníka.

Při snaze o řešení současného stavu lze vypozorovat dva základní směry. Jeden se zaměřuje na vedení softwarových projektů, jinak řečeno na manažerskou stránku věci. Typickými tématy tohoto směru jsou řízení a provoz informačních systémů, outsourcing, řízení podnikových a informatických procesů, rozvoj informačních systémů a podobně.

Druhý směr řešení současného stavu je zaměřen na softwarový vývoj. Má více co do činění s konkrétními technologickými záležitostmi a snaží se k řešení problému přispět tím, že zefektivní práci každého jednotlivého vývojáře.

Být dobrým programátorem je totiž obtížné. Objem různých knihoven, aplikačních rozhraní a frameworků za poslední léta výrazně narostl a není proto náhodou, že jsou odborná knihkupectví plná detailních příruček o programování v Javě, o tvorbě stránek v ASP.NET nebo o syntaxi různých programovacích jazyků.



Obrázek 1 – Zaměření práce

Skvělý programátor ale ještě nemusí být dobrým *vývojářem*. Typickým příkladem je programátor, který dokáže komplikovaný algoritmus zapsat na pár řádků svého oblíbeného programovacího jazyka, ale současně zapomene vytvořit jakoukoli nápovědu, neodsadí zdrojový kód a jediné, co ho bude zajímat, je výkonnost výsledného programu. Pokud takovýto člověk pracuje na jádru operačního systému, je pro svou firmu velmi cenný, jedná-li se však o aplikačního vývojáře, zbude po něm nepřehledný, neudržovatelný a nesrozumitelný kód, který bude dlouhodobě spíše problémem než přínosem.

Druhý naznačený směr je charakteristický tím, že se snaží dělat z dobrých programátorů také dobré vývojáře. Jaké k tomu má prostředky? Nabízí programátorům osvědčené praktiky, různá doporučení nebo *vzory*, jejichž dodržování vede k přehlednějšímu, znovupoužitelnějšímu a v konečném důsledku levnějšímu kódu.

Tento směr je dlouhodobou iniciativou, jejíž kořeny sahají hluboko do minulého století. Například vzor Model-View-Controller byl poprvé popsán už v 70. letech 20. století a komunitou kolem Smalltalku byl aktivně používán už v letech osmdesátých[1]. Velký zájem o návrhové vzory napříč celou vývojářskou komunitou vyvolala kniha *Návrh programů pomocí vzorů* od „Gang of Four“ ([GoF, 1994]), která se rychle stala živoucí legendou a vznikla řada dalších vzorů, např. pro tvorbu uživatelských rozhraní, pro vícevláknové programování nebo třeba pro řídicí procesy.

Na co naopak doposud takový důraz kladen nebyl, je praktická aplikace vzorů při každodenním programování. V různých katalozích se sice často uvádějí příklady zdrojových kódů, které daný vzor realizují, mezi ukázkou a reálně použitelných artefaktem softwarového vývoje však existuje

[1] Podrobnosti jsou uvedeny např. v článku *Model View Controller History*, http://c2.com/cgi/wiki?ModelViewControllerHistory.

podstatný rozdíl. Navíc nemusí být vývojáři známý programovací jazyk, který byl v ukázce použit, čímž se situace dále komplikuje.

Tato diplomová práce by proto chtěla přispět především ke konkrétní aplikaci vzorů v reálných programech a softwarových systémech skrze vývojová prostředí (ačkoliv je z názvu práce patrné zaměření na platformu .NET a jazyk C#, řadu poznatků lze využít obecně).

1.2 Hlavní cíle

Popsat teoretická východiska podpory návrhových vzorů ve vývojových prostředích

Zmapovat současnou situaci na poli CASE nástrojů i specializovaných aplikací

Navrhnout vlastní vývojové prostředí pro návrhové vzory

Komplexně popsat a zhodnotit problematiku podpory návrhových vzorů ve vývojových prostředích, popsat možnosti budoucího vývoje

1.3 Historický kontext (důležité upozornění)

Diplomová práce byla značně ovlivněna některými okolnostmi, které je nutno zmínit. Název, zadání i přiložená osnova pocházejí z akademického roku 2004/2005, zatímco práce byla dokončena v létě 2006. Za tu dobu se v oblasti zájmu událo velmi mnoho…

Původně měl být hlavním výstupem práce jednoduchý CASE nástroj, ve kterém by uživatel mohl vizuálně navrhnout aplikaci pomocí návrhových vzorů a strojově by mu potom byl vygenerován zdrojový kód v jazyce C# (zaměření na platformu .NET a jazyk C# vycházelo ze skutečnosti, že téma diplomové práce bylo vypsáno na námět Jana Stoklasy z české pobočky Microsoftu). V roce 2005 však Microsoft vydal Visual Studio 2005, učinil tak první krok k realizaci vize „softwarových továren“, kterým je věnována důležitá část této práce, a současně příjemným způsobem vstoupil do plánů této diplomové práce, protože vývojové prostředí pro návrhové vzory již nebylo nutno vytvářet od začátku.

Mohlo by se zdát, že tím práce ztratila svůj hlavní cíl, ten se ale ve skutečnosti pouze posunul. Visual Studio je totiž pouze *potenciálním* vývojovým prostředím pro návrhové vzory – obsahuje řadu pomůcek, které dokáží značně zjednodušit život (např. není potřeba programovat modelovací framework), zbývá ale ještě hodně práce, než bude Visual Studio plnit takovou roli, jakou si jeho autoři představují.

Celkově vzato je pro tuto práci vydání Visual Studia 2005 velkým štěstím a přínosem, protože umožnilo věnovat se zajímavějším tématům než konstrukci jednoúčelového CASE nástroje.

1.4 Obsah práce

Hlavní témata práce jsou následující:

Formální zápis návrhových vzorů – obecná východiska jsou uvedena v kapitole 3, samostatné formalizační techniky v kapitole 4 a kapitola pátá se věnuje notacím založeným na UML.

Kapitola 6 mapuje **podporu návrhových vzorů v dnes dostupných CASE nástrojích**.

Sedmá kapitola popisuje **vizi „softwarových továren“ a doménově specifických jazyků**.

Osmá kapitola představuje **koncept Visual Studia jakožto vývojového prostředí pro návrhové vzory**.

Obsahem práce naopak *není*:

Katalog vzorů. Základní orientace ve světě návrhových vzorů se předpokládá, ačkoliv není úplně nezbytná.

Popis všemožných témat souvisejících s problematikou vzorů. Za tímto účelem vzniklo mnoho jiných prací, např. [Hochmann, 2006].

Detailní pohled na všechna probíraná témata. Práce je obsahově poměrně široká a pro zachování rozumného rozsahu byly často vypuštěny detaily, které nebyly pro pochopení dané problematiky nezbytně nutné.

1.5 Přínosy práce

Tato práce přináší:

Obsáhlý přehled formálních zápisů návrhových vzorů s vyzdvihnutím hlavních myšlenek spíše než technických podrobností. Podobné přehledy byly vždy zaměřeny méně komplexně a obsahovaly možná až zbytečně mnoho detailů.

Popis vize softwarových továren a jejich návaznost na problematiku podpory návrhových vzorů ve vývojových prostředích.

Návrh doménově specifického jazyka pro vizualizaci implementovaných návrhových vzorů.

Koncept tříúrovňového vývojového prostředí pro návrhové vzory realizovaného v prostředí Visual Studia. Součástí je notace pro zachycení návrhových vzorů na úrovni zdrojového kódu a výše zmíněný doménově specifický jazyk.

2 Úvod do návrhových vzorů

Návrhovým vzorům je věnováno několik vynikajících knih (např. [GoF, 1994] nebo [Fowler, 2002]), na internetu lze nalézt řadu volně dostupných materiálů a vzory byly rovněž námětem několika diplomových prací zpracovaných na Vysoké škole ekonomické (např. [Dvořák, 2003] nebo [Hochmann, 2006]). Tato kapitola se proto zaměřuje pouze na témata, která mají přímou návaznost na hlavní cíle práce.

2.1 Zasazení návrhových vzorů do kontextu softwarového vývoje

Vývoj aplikace nebo softwarového systému se vždy řídí určitým paradigmatem, tedy jakousi sadou souvisejících konceptů. Těchto paradigmat existuje mnoho, např. funkcionální nebo strukturované programování, ale během posledních desetiletí se prosadil především objektově orientovaný přístup, který pro popis systému používá pojmy jako třída, metoda, dědičnost, zapouzdření, polymorfismus a podobně. Nezkušený vývojář začíná s velmi malou sadou konceptů, většinou je zaměřen pouze na zvládnutí základních jazykových konstruktů. Pokročilejší vývojář už rozumí širší sadě konceptů, do které se přidávají algoritmy, pochopení datových struktur a podobně. Návrhové vzory představují další stupeň, který umožňuje vývojáři chápat souvislosti mezi objekty a jejich kompozici do funkčně pokročilejších celků. (Volně převzato z [Bosch, 1998].)

O přesném vymezení návrhových vzorů se píší celé články, viz např. [Pattern Definition], základní idea však nijak komplikovaná není a dobře ji vystihuje např. definice uvedená v [Bosch, 1998]:

> „Každý vzor popisuje opakující se problém a jeho obecné řešení, které může být přizpůsobeno konkrétní situaci."

Přesné vyjádření vzoru nemá svou standardizovanou podobu, např. prapředek všech prací o vzorech, [Alexander, 1979], uvádí, že základními prvky vzoru jsou kontext, problém a řešení, zatímco populární kniha [GoF, 1994] jako čtyři hlavní prvky uvádí název, problém, řešení a důsledky. Ačkoliv se tedy jednotlivé zápisy svou strukturou liší, tabulka 1 na příkladu dvou různých katalogů demonstruje, že lze většinou určitou shodu nalézt:

[GoF, 1994]	[Dvořák, 2003]
Název a klasifikace	Název
Účel Jiné názvy Motivace Aplikovatelnost	Problém Podmínky
Struktura	Řešení
Účastníci	
Spolupráce	
Důsledky Implementace	Odůvodnění a souvislosti Výsledek
Ukázkový kód	Příklady
Známá použití	Reference
Související vzory	Související vzory

Tabulka 1 – Popis návrhových vzorů a jejich přibližné mapování

Vedle toho, že návrhové vzory nabízejí návody na řešení opakujících se problémů, je jejich další důležitou úlohou *komunikační funkce*. Obeznámený vývojář například ihned ví, co očekávat od „Singletonu". Návrhové vzory se tak stávají důležitými dokumentačními a komunikačními pro-

středky, což je hlavní důvod, proč [GoF, 1994] jako jednu z nejdůležitějších charakteristik vzoru uvádí právě jeho název.

Obrázek 2 schematicky zobrazuje svět vzorů a je na něm dobře vidět, že návrhové vzory tvoří pouze jejich podmnožinu. Slovo „návrhové“ naznačuje, že jsou primárně zaměřeny na oblast návrhu (designu) softwarového systému, tuto skutečnost je však potřeba brát spíše jako určité vodítko a nikoliv jako dogma, protože je pojem návrhových vzorů poměrně široký.

Obrázek 2 – Svět vzorů (zdroj: [Stein, 2004])

Motivace vzniku vzorů byla naznačena hned v samém úvodu – zatímco o technických detailech jednotlivých technologií vždy existovaly podrobné manuály a řady knih, chyběly postupy, jak tyto technologie co nejlépe využít k řešení reálných problémů. V lepším případě se vývojář mohl opřít o interní know-how softwarové firmy, ve které byl zaměstnán, v horším případě musel na nějaké „dobré“ řešení přijít sám.

Katalogy návrhových vzorů se snaží tuto mezeru zaplnit. Patrně nejslavnější knihou je [GoF, 1994], bylo však popsáno mnoho dalších vzorů, ať už obecných či specifických pro určitou technologii nebo platformu (existují např. vzory pro JEE, pro vícevláknové programování a podobně).

Mohlo by se zdát, že standardizací, zveřejněním a obecným přijetím návrhových vzorů se všechny problémy vyřešily. Bohužel nikoliv…

2.2 Problémy současného stavu

Prvním problémem je poměrně **slabá podpora návrhových vzorů ze strany softwarových nástrojů**. V devadesátých letech minulého století se objevil jazyk UML, modelování obecně zažilo velký boom a na trhu se objevila řada CASE nástrojů, které začaly podporovat „Model Driven Development“. Je trochu s podivem, že dnešní CASE nástroje s výbornou implementací standardu UML obvykle podporují návrhové vzory pouze okrajově nebo dokonce vůbec – přitom právě doména návrhu softwarových systémů by měla být jejich silnou stránkou. O pravděpodobných příčinách lze spekulovat a jedno z možných vysvětlení nastiňuje například kapitola 7.1, faktem však zůstává, že se v současnosti o problém jedná.

Na své vyřešení rovněž čeká problém **formálního zápisu vzorů**, který s podporou vzorů v nástrojích úzce souvisí. V současnosti je totiž zvykem popisovat vzory především slovně a tedy vágně, což znamená pro strojové zpracování vážnou komplikaci. Jak dokumentuje kapitola 4, v akademické sféře několik formálních notací vzniklo, ale žádná se nestala de-facto standardem a žádná není používána dnešními CASE nástroji.

V neposlední řadě je velkým problémem **vzdělávací proces nových vývojářů**. Ačkoliv je informatika na Vysoké škole ekonomické oproti jiným českým školám ještě hodně prakticky a „architektonicky“ zaměřena, za celé studium jsem zmínku o návrhových vzorech zaslechl pouze ve dvou předmětech, kde se navíc jednalo pouze o stručné doplnění k jinak technické látce (nutno dodat, že se prý v současnosti situace zlepšuje). Rovněž na knižním trhu existuje značná nerovnováha, porovná-li se nabídka technických titulů a knih zaměřených na modelování. Není proto divu, že mnoho vývojářů o návrhových vzorech slyšelo jen velmi letmo nebo dokonce vůbec.

Školy sice ke změně osnov nutit nelze, stejně jako není možné ovlivnit nabídku vydavatelství jinak než odpovídající poptávkou, nástroje však mohou pomoci i při řešení tohoto problému. Pokud by se podpora vzorů stala standardní součástí vývojových prostředí, každý programátor by

se s nimi setkával přirozeným a nenuceným způsobem. Pokyny v knihách by se mohly změnit z instrukcí typu „na řádek 30 doplňte následující kód …“ například na instrukce „v menu Vložit vyberte návrhový vzor XY“ a podobně.

Současný stav tedy zdaleka není ideální a zbytek práce se snaží svou troškou přispět k realizaci velkého snu – zakořenění návrhových vzorů v každodenní práci běžného vývojáře.

3 Formální zápis návrhových vzorů

Vývojové prostředí pro návrhové vzory musí pracovat se strukturovaným a formalizovaným katalogem vzorů. Tato kapitola se věnuje teoretickým i praktickým východiskům formalizace, zatímco čtvrtá a pátá kapitola přinášejí přehled jak samostatných formalizačních pokusů, tak notací založených na UML.

3.1 Potřeba formálního zápisu

Návrhový vzor je tvořen několik elementy, např. názvem, účelem, účastníky a podobně (viz tabulka 1 na straně 4, záleží na konkrétním katalogu). Kromě struktury a ukázkového kódu jsou většinou všechny další oblasti popsány v přirozeném jazyce, což má své výhody i nevýhody. Přirozený jazyk je dobře srozumitelný a umožňuje věci popsat do hloubky, na druhou stranu ale trpí některými nedostatky, především nepřesností a vágností.

Formální zápis by přinesl následující výhody:

1. Přesný popis vzorů by měl pozitivní vliv na snadnost jejich **pochopení**. Jeden diagram někdy řekne více než mnoho řádků textu.
2. Byl by **jasnější vztah mezi vzory**. V současnosti nemusí být úplně snadné na základě popisu vzoru stanovit, zda není speciálním případem, variací nebo součástí jiného vzoru.
3. Bylo by možné provést **formální validaci**, tedy zjistit, zda programový kód odpovídá danému vzoru nebo vzorům.
4. Formalizace vzorů by **usnadnila, respektive umožnila podporu v nástrojích**.

3.2 Námitky proti formalizaci

Vedle uvedených výhod se objevují i názory, že formalizace není možná nebo dokonce že není žádoucí. Zde je argumentace převzatá z [Baroni aj., 2003]:

Námitka: Vzory popisují řešení široké kategorie problémů. Základní výhodou používání vzorů je párování řešení s relevantními problémy. Zaměření pouze na řešení tuto výhodu ničí.

Odpověď: Detailní specifikace řešení nenaznačuje, že by snad zadání problému bylo nepodstatné. Jinými slovy, konkrétní specifikace řešení neznamená opomenutí dalších důležitých částí aplikace vzorů.

Námitka: Formální specifikace skoro vůbec nebo vůbec nepřispívá k pochopení, kdy a jak vzor použít.

Odpověď: Je především věcí názoru, zda striktní popis vzorů pomáhá jejich pochopení či nikoliv. Tak jako tak, nejasné a nejednoznačné formulace určitě přesnému pochopení příliš nenapomáhají.

Námitka: Vzory jsou určitou abstrakcí, zobecněním, a proto mají být vágní, nejednoznačné a nepřesné. Pokud by byly precizně vyjádřené třeba nějakou matematickou formulkou, přestaly by být vzory.

Odpověď: Předpoklad, že přesná specifikace může popisovat pouze přesné a konkrétní problémy a že proto popis obecných problémů musí být vágní, je špatný. Specifikace může být precizní a současně zaměřená na obecné problémy.

Námitka: Ve vzorech neexistují fixní elementy, každá část ve vzoru se může změnit. Jinými slovy, pokud je struktura fixní, nejedná se o vzor.

Odpověď: Pokud by toto byla pravda, potom neexistuje způsob, jak vzory dobře popsat nebo pochopit.

3.3 Přístupy k formalizaci vzorů

Existuje několik základních přístupů k formalizaci vzorů:

1. **Komplexní popis vzorů s důrazem na přirozený jazyk**. Popis je sice částečně strukturovaný a formalizovaný, např. použitím diagramů v nějaké objektové notaci, ale hlavní důraz je stále kladen na přirozený jazyk. Typickým zástupcem jsou knihy [GoF, 1994] nebo [Fowler, 2002], podobných příkladů ale existuje více, např. [Dvořák, 2003] a další.
2. **Formalizace na úrovni zdrojového kódu**. Tento přístup si nedělá ambice na kompletní a univerzální popis vzorů, ale má určité výhody, které budou zmíněny níže.
3. Vytvoření **speciálního jazyka, ať už textového nebo vizuálního**, pro formální popis vzorů. Některé tyto jazyky vycházejí z matematických notací a snaží se vzory zachytit v co nejstriktnější podobě, jiné notace mají své hlavní cíle poněkud odlišné a snažní se např. o co nejpřehlednější vyjádření vztahů mezi vzory nebo o něco podobného.

Následující kapitoly se věnují vhodnosti, resp. nevhodnosti jednotlivých formalizačních prostředků. Základ je přejat z práce [Baroni aj., 2003] s tím, že některé myšlenky byly poněkud rozvedeny.

3.4 (Ne)vhodnost konvenčních programovacích jazyků

I intuice dává tušit, že návrhové vzory nepůjdou dobře zachytit pomocí konvenčních objektově orientovaných jazyků jako jsou C++, Java nebo C#. Kdyby šly, nebylo by nutné nic řešit, klasické objektově orientované jazyky však ve vztahu k návrhovým vzorům trpí několika problémy (první čtyři jsou popsané v [Baroni aj., 2003] nebo [Bosch, 1998]):

1. **Vystopovatelnost** (*traceability*). Protože objektově orientované programovací jazyky nepodporují syntaxi pro zápis návrhových vzorů, vývojář vzor implementuje jako oddělené třídy, metody a zprávy mezi nimi, takže vzor, který je ve fázi návrhu jednotnou entitou, se ve fázi implementace roztříští. V knize [Greenfield, Short, 2004] se s problémem vystopovatelnosti pojí ještě tzv. **problém rekonstrukce** (*reconstruction problem*), kdy je prakticky nemožné z cílové implementace sestrojit původní model, který byl této implementaci předobrazem.
2. **„Self-problem“** neboli problém odkazu sama na sebe se projeví v situacích, kdy objekt *A* posílá zprávu objektu *B*, ten ale neobsahuje implementaci zpracování zprávy, a proto ji předává dál objektu *C*. Když se však zpráva dostane až k němu, odkaz na sebe ukazuje na *C* (konečného příjemce, zpracovatele zprávy) místo na *B* (původního příjemce zprávy). Poprvé byl tento problém popsán v [Lieberman, 1986].
3. **Znovupoužitelnost** (*reusability*). Návrhové vzory jsou primárně prezentovány jako návrhové struktury, v cílovém objektově orientovaném jazyce však nemají prostředky, jimiž by mohly být prezentovány. Ačkoliv tedy vzory mohou být sdíleny a znovupoužívány na úrovni návrhu, konkrétní implementace znovupoužívat nelze.
4. **Implementační režie** (*implementation overhead*). Omezení znovupoužitelnosti uvedené v minulém bodu samo o sobě znamená nutnost psát více kódu než by bylo v ideálním případě nutné, existují ale i další příčiny. Řada vzorů totiž ke své implementaci potřebuje

nějakou triviální funkčnost, např. předání zprávy dalšímu objektu. Tuto funkčnost bohužel musí psát vývojář, což je jednak trochu zbytečná práce, jednak znepřehledňuje výsledný kód.

5. I kdyby se podařilo překonat předchozí uvedené problémy, faktem zůstává, že **zdrojový kód není z principu vhodný pro zachycení návrhových vzorů**. Přítomnost objektových notací dokládá, že kód není dostatečný ani pro popis návrhu samostatného softwarového systému, natož pak návrhových vzorů, které jsou aplikovatelné na celou třídu softwarových aplikací. Návrhové vzory navíc nejsou závislé na cílovém implementačním prostředí, natož pak na konkrétním programovacím jazyce.

První čtyři problémy se snaží řešit formalizační metoda LayOM popsaná v kapitole 4.1. Z hlediska jazyka C# se jako zajímavá možnost nabízí využití atributů, ve kterých je pro podobné případy skryt velký potenciál, ale o nich bude pojednáno později. Důležitým závěrem této podkapitoly je zjištění, že programový kód díky bodu 5 pro popis návrhových vzorů příliš použít nelze.

Na tomto místě je důležité zmínit ještě jednu věc. Aby byl vzor vůbec k něčemu, musí být implementovatelný a programový kód tak v popisu vzoru hraje svou důležitou roli. Ačkoliv tedy kód sám o sobě pro úplný popis návrhových vzorů nestačí, vyřešení uvedených problémů by bylo pro formalizační pokusy velkým přínosem.

3.5 (Ne)vhodnost objektových notací

Ve většině knih o návrhových vzorech je používána nějaká objektová notace, nejčastěji UML. Nebylo by jednodušší použít zavedenou notaci a nevymýšlet novou?

Námitky proti používání stávajících objektových notací vycházejí ze skutečnosti, že mezi návrhem jednoho softwarového systému a popisem obecného vzoru existuje podstatný rozdíl. Zatímco výsledkem návrhu softwarového systému by měly být konkrétní podklady pro implementaci cílové aplikace, návrhové vzory se zaměřují na řešení obecně pojatých problémů. Jeden návrhový vzor může být použit v libovolném počtu programů.

Při použití obecně rozšířené terminologie potom mluvíme o návrhovém vzoru na straně jedné a o *instanci návrhového vzoru* na straně druhé. Notace jako UML nebo OMT (ta druhá je používána v [GoF, 1994]) jsou zaměřeny na modelování *instancí* návrhových vzorů, a proto jim chybí některé výrazové prostředky pro vzory jako takové. Příkladem může být absence symbolu pro dědičnou hierarchii objektů – lze nakreslit třídu a několik jejích potomků, v tom případě už ale nepopisujeme dědičnost v její podstatě, ale pouze její příklad, což pro obecný popis vzorů znamená problém.

Na druhou stranu by bylo špatné zamlčet evidentní výhody, které by použití UML mělo. Největší předností je obecná rozšířenost UML a tedy okamžité pochopení diagramů širokou vývojářskou komunitou. Dalším podstatným kladem je dobrá podpora UML v dnešních CASE nástrojích.

V zásadě se tedy lze setkat se třemi typy názorů:

1. UML je pro popis návrhových vzorů dostačující a vhodné
2. UML pro popis návrhových vzorů není vhodné, protože neobsahuje potřebné výrazové prostředky
3. Výhody UML lze využít, ale notaci je potřeba mírně modifikovat

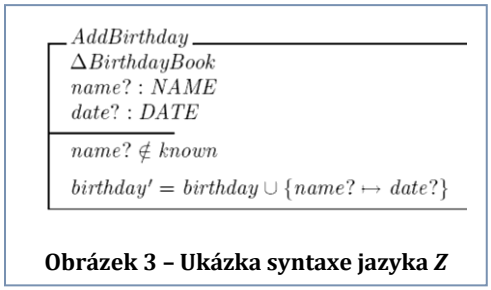
Rozšířenost prvního názoru dokumentují snad všechny dostupné knihy o návrhových vzorech, druhý názor vedl ke vniku formalizačních technik popsaných v kapitole 4 a třetí postoj byl základem notací uvedených v kapitole 5.

Osobně považuji za důležitý kontext, ve kterém se má notace používat. Pokud má sloužit k přesnému a úplnému popisu vzorů v jejich podstatě, souhlasím, že UML některé výrazové prostředky postrádá. Pokud mají být diagramy použity v knize a jejich hlavní úlohou je prezentovat vzory široké vývojářské komunitě, považuji UML za velmi vhodný komunikační prostředek. Své přednosti mají i notace od UML odvozené, ale jejich konkrétní využití už mi není tolik zřejmé – sice umožňují zkombinovat výhody obou krajních přístupů, na druhé straně ale nutně kombinují i jejich nevýhody (např. upravený class diagram bude při prvním setkání hůře srozumitelný než standardní diagram tříd).

Myslím tedy, že hlavní závěry této kapitoly jsou následující: v určitých případech má smysl hledat zcela samostatnou, na UML nezávislou notaci pro popis vzorů, zatímco jindy je rozumnější využít přednosti zažité a všem známé UML notace.

3.6 (Ne)vhodnost formálních specifikačních jazyků

Pro konstrukci softwarových systémů byly vymyšleny formální specifikační jazyky, které se snaží rigorózně definovat chování vyvíjeného softwarového systému bez ohledu na implementační detaily. Příkladem je jazyk *Z* popsaný ve volně dostupné knize [Spivey, 1992]. Nabízí se tedy otázka, jestli by třeba konkrétně jazyk *Z* nešel použít také pro formalizaci návrhových vzorů.



Obrázek 3 – Ukázka syntaxe jazyka *Z*

Při pohledu na ukázku syntaxe z obrázku 3 je hned vidět první velký problém: formální specifikační jazyky používají komplikovanou matematickou notaci (jazyk *Z* je konkrétně založen na syntaxi predikátové logiky), která je pro vývojáře obtížně srozumitelná. (Jak bylo uvedeno, jedním z hlavních cílů formalizačního úsilí je mimo jiné zlepšení komunikace návrhových vzorů mezi vývojáři.)

Kromě obtížné syntaxe mají formální specifikační jazyky ve vztahu k návrhovým vzorům i další omezení, např. popisují *vnější* vlastnosti systému, zatímco implementační záležitosti vůbec neřeší (ačkoliv jsou návrhové vzory obecnými návody hodícími se do mnoha situací, aspekt konečné implementace v nich hraje důležitou roli). Dalším podstatným omezením je skutečnost, že formální specifikační jazyky obvykle neumějí zachytit koncept obecné množiny softwarových systémů (podobně jako je tomu u objektových notací). Své určení tedy jazyky typu *Z* mají, ale pro popis návrhových vzorů nejsou vhodné a nebudeme se jim dále věnovat.

4 Přehled samostatných formalizačních technik

Tato kapitola představuje samostatné, na UML nebo jiné objektové notaci nezávislé formalizační techniky. Důraz je vždy kladen na hlavní myšlenku notace a její výhody, případně nevýhody, nikoliv na technické detaily, které lze v případě zájmu vyčíst z odkazovaných materiálů.

4.1 LayOM – Layered Object Model

V kapitole 3.4 byly uvedeny některé problémy, kvůli kterým je obtížné návrhové vzory zachytit na úrovni zdrojového kódu. LayOM (*Layered Object Model*) se snaží do tradičního objektového modelu přidat nové elementy, které by umožnily některé z těchto problémů překonat.

4.1.1 Hlavní ideje

Původně byl LayOM navržen jako jazyk se syntaxí podobnou Pascalu, který se pomocí tzv. delegujících kompilačních objektů převáděl do C++ (podrobnosti viz [Bosch, 1998, str. 8]). Objevila se však i implementace v jazyce podobném Javě, viz [LayOM Compiler] a [LayOM Specification], takže je potřeba LayOM primárně vnímat jako obecný objektový model, který je relativně nezávislý na použité syntaxi.

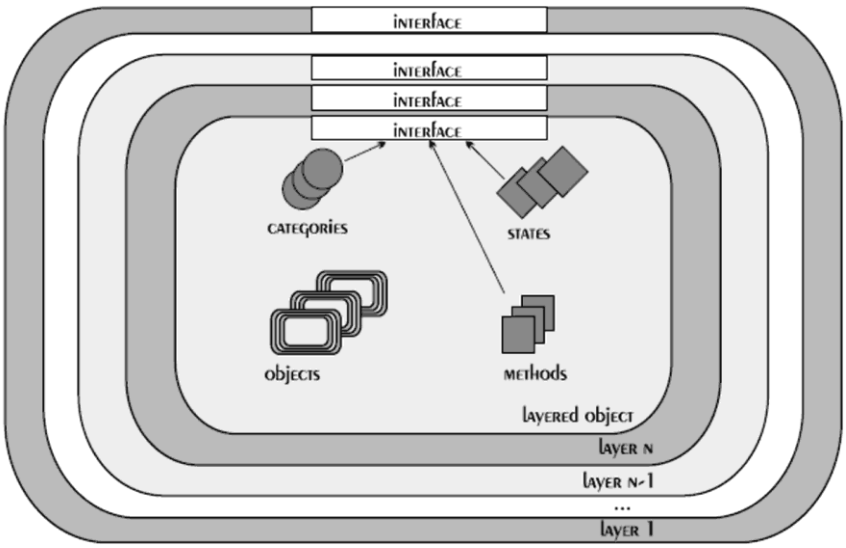
Jak bylo uvedeno v kapitole 3.4, klasické objektově orientované jazyky jako C++, Java nebo C# trpí ve vztahu k návrhovým vzorům minimálně následujícími problémy:

vystopovatelnost (*traceability*)

problém odkazu sama na sebe (*self-problem*)

znovupoužitelnost (*reusability*)

implementační režie (*implementation overhead*)



Obrázek 4 – Princip LayOM – obalení objektu vrstvami (zdroj: [Bosch, 1998])

LayOM se problémy v těchto čtyřech oblastech snaží řešit zavedením tzv. vrstev (*layers*), které obalují standardní objekty a umožňují tak jejich seskupování do logicky vyšších celků. Vrstvy přijímají zprávy z vnějšku a předávají je zapouzdřenému objektu. Hlavní princip naznačuje ob-

rázek 4 – uprostřed je objekt rozšířený o některé nestandardní komponenty (konkrétně o kategorie a stavy, viz níže) a kolem tohoto objektu jsou umístěny vrstvy, které s okolím komunikují pomocí svých rozhraní.

Objekt se v modelu LayOM může skládat z následujících komponent:

Instanční proměnné a metody (*instance variables*, *methods*) s velmi podobným významem jako v běžných objektově orientovaných jazycích. Jediným rozdílem je, že instanční proměnné mohou mít další funkčnost přidanou obalovou vrstvou.

Stavy (*states*) jsou abstrakcí interního stavu objektu.

Kategorie (*categories*) popisují charakteristiky množiny klientů, ke kterým by se objekt měl chovat stejně.

4.1.2 Příklad

Následující příklad je převzat z [Bosch, 1998] a zachycuje konkrétní podobu jazyka LayOM založeného na syntaxi Pascalu:

```
class TextEditWindow
  layers
    rs : RestrictState(Programmer,
                       accept all when distFromOrigin<100 otherwise reject);
    pin : PartialInherit(Window, *, (moveOrigin));
    po : PartOf(TextEditor);
  variables
    loc : Location;
  methods
    moveOrigin(newLoc : Location) returns Boolean
    begin
      loc := newLoc;
      self.updateWindow;
    end;
  states
    distFromOrigin returns Point
    begin
      return ((lox.x - self.origin.x).sqr + (lox.y - self.origin.y).sqr).sqrt;
    end;
  categories
    Programmer
    begin
      sender.subClassOf(Programmer);
    end;
end; // class TextEditWindow
```

Třída TextEditWindow má definovány tři vrstvy:

Vrstva RestrictState omezuje přístup k instancím třídy Programmer a jejím podtřídám, pokud je kurzor myši vzdálen od počátku okna víc než o 100 pixelů.

Vrstva PartialInherit určuje, že třída TextEditWindow dědí všechny metody třídy Window s výjimkou metody moveOrigin, a zasahuje tak do mechanismu dědění. Většina současných objektově orientovaných jazyků takovéto chování nepodporuje.

Vrstva PartOf označuje, že je třída TextEditor součástí třídy TextEditWindow.

Na ukázce je dále vidět zachycení stavů a kategorií – stav distFromOrigin vrací vzdálenost od počátku, zatímco do kategorie Programmer budou spadat všechny instance třídy Programmer nebo libovolné její podtřídy.

4.1.3 Další vlastnosti LayOM

Kromě toho, že je LayOM rozšířeným objektovým modelem, je také objektovým modelem rozšiřitelným. Softwarový inženýr může přidat např. nový typ vrstvy, ale také strukturální prvky jako např. události, což z LayOM dělá flexibilní nástroj.

4.1.4 Shrnutí, výhody, nevýhody

Na LayOM je dobrá především jeho exaktní a prakticky použitelná podoba, neboť lze vytvořený model konvertovat do rozšířených jazyků C++ nebo Java (podle použitého kompilátoru).

Další charakteristikou LayOM jsou jeho dobré vyjadřovací schopnosti. Pomocí LayOM lze vyjádřit všechny GoF vzory a díky principu rozšiřitelnosti by nebyl problém zachytit i zcela nové druhy vzorů, pro které by třeba ve standardní specifikaci LayOM nemusely být dostupné žádné jazykové konstrukty.

Pokud přehlédneme drobnou vadu na kráse, kterou je roztříštěná syntaxe LayOM (jak bylo uvedeno, existují kompilátory jak pro syntaxi podobnou Pascalu, tak pro syntaxi podobnou Javě), poměrně velkou nevýhodou LayOM zůstává přílišná specifičnost jeho vyjadřovacích prostředků. LayOM nepřináší žádnou vizuální notaci a pro přehledný zápis vzorů se tedy nehodí. Na druhou stranu ani k cílové implementaci nemá tak blízko, jak by se mohlo zdát, protože musí být do cílového jazyka teprve konvertován. V podstatě se tak jedná o určitý meta-jazyk, který se sice jako rozšířený OO jazyk tváří, k cílovému spustitelnému souboru je od něj ale ještě přeci jen poměrně daleko. LayOM tak zůstává stát kdesi napůl cesty: na jednu stranu není vhodný pro univerzální popis návrhových vzorů, na druhou stranu není ani cílovým implementačním jazykem. Pokud si uvědomíme, že to samé se dá prohlásit třeba o UML, je nutno na LayOM pohlížet jako na jeho určitou alternativu. Je ale vůbec žádoucí vytvářet další notaci, která má podobný účel jako UML a nepřináší nic extra nového?

Autor metody LayOM, Jana Bosch ([Bosch]), v e-mailové korespondenci odpověděl takto[2]: „Myslím, že tyto připomínky jsou oprávněné, a ačkoliv v některých bodech nesouhlasím, myslím, že přehlédnuta byla především časová dimenze. LayOM byl vyvinut primárně v letech 1994 až 1998, zatímco UML dosáhlo reálného rozšíření až později.“ Tato slova nepřímo naznačují, že dnes je již LayOM do značné míry překonanou záležitostí.

Výhody	Rozšíření syntaxe OO jazyka o nové elementy
	Relativní blízkost k cílové implementaci
	Rozšířený i rozšiřitelný objektový model
	Schopnost vyjádřit všechny GoF vzory, možnost rozšíření pro nové vzory
Nevýhody	Nemá ambice ani na tvorbu cílového implementačního kódu, ani na abstraktní popis vzorů
	Notace je pouze textová (pseudokód), chybí grafická notace
	V určitém smyslu nepřináší o nic víc než UML
	LayOM je do značné míry překonaná záležitost.

Tabulka 2 – Výhody a nevýhody LayOM

4.2 Zachycení vzorů v programovacím jazyce pomocí atributů

V roce 1997 napsal Görel Hedin práci *Language Support for Design Patterns using Attribute Extension* ([Hedin, 1997]), kde představil koncept nízkoúrovňové podpory návrhových vzorů na

[2] Kompletní korespondenci s Janem Boschem, tvůrce metody LayOM, a Antonem Jansenem, současným pokračovatelem výzkumu v této oblasti, je možno nalézt v příloze A.

úrovni kódu. Jestliže byl LayOM pokusem o vyjádření návrhových vzorů pomocí rozšířeného objektově orientovaného „meta-jazyka“, je technika Görela Hedina snahou zakomponovat návrhové vzory do zcela standardního objektově orientovaného jazyka, konkrétně C++, pomocí tzv. atributů. Atributy jsou doplňkové informace zapsané ve zdrojovém kódu a konkrétně v této technice určují, v jaké roli jakého návrhového vzoru třída či metoda vystupuje. Důsledky jsou velmi příjemné – především tato metoda zcela eliminuje problém vystopovatelnosti (*traceability*).

Konkrétní ukázky je v případě zájmu možné nalézt v práci [Hedin, 1997], z praktických důvodů však nebudou do tohoto textu zařazeny. V době psaní této práce už totiž poměrně dlouho existují objektově orientované jazyky, které přímou podporu pro atributy obsahují, ať už se jedná o Javu, C#, Visual Basic .NET nebo další jazyky cílené na platformu .NET (a nejen je). Z tohoto pohledu nemá příliš smysl se ke konkrétnímu zápisu v Hedinově práci vracet, což ovšem nijak nesnižuje sílu jeho myšlenky: zachytit určité artefakty návrhových vzorů přímo ve zdrojovém kódu pomocí atributů.

Jako určitá nevýhoda se může jevit nutnost explicitního zásahu do zdrojového kódu vývojářem, který musí ručně „označkovat“ odpovídající třídy, vlastnosti a metody. Nevýhoda je to však jen částečná a mnohdy zdánlivá – pokud bude kód generován nějakým CASE nástrojem, mohou být atributy do kódu přidány automatizovaně, tedy bez námahy pro vývojáře.

Důležitým výsledkem této kapitoly je poznatek, že zachycení vzorů v cílovém programovacím jazyce pomocí atributů je konečným řešením problému vystopovatelnosti. Zbylé formalizační techniky tedy zkoumají opačnou oblast, která se snaží o formalizaci vzorů na jejich nejabstraktnější úrovni.

Výhody	Zachycení návrhových vzorů explicitně v cílovém programovacím jazyce
	Zbavení se problému vystopovatelnosti
	Konečné řešení ve směru specifičnosti při formalizačních snahách
Nevýhody	Atributy nejsou schopné popsat návrhové vzory v jejich podstatě
	Při ruční tvorbě je nutno třídy, metody a atributy „označkovat“ manuálně; této nevýhody se lze zbavit použitím CASE nástroje

Tabulka 3 – Výhody a nevýhody zachycení vzorů pomocí atributů

4.3 DisCo

DisCo je metoda popsaná v [Mikkonen, 1998]. Jak bude uvedeno v zápětí, DisCo pravděpodobně není příliš perspektivní cestou, proto bude tato kapitola kratší a spíše pro úplnost.

DisCo popisuje softwarový systém pomocí *tříd*, *vztahů* a *akcí*. Definice tříd a jejich relací vypadají celkem srozumitelně – třída se zapisuje jako class C = {x} a relace jako relation (n).R.(m): C x D (tento zápis vyjadřuje vztah mezi třídami C a D s kardinalitou danou čísly *n* a *m*).

Akce je naproti tomu poměrně komplikovaná záležitost, která je definována jako atomická jednotka konání, která může být chápána také jako multi-objektová metoda. Příklad jejího formálního zápisu je následující:

$$A(c:C, i):$$
$$i \quad c.x$$
$$c.x' \quad i$$

Převedeno do běžné řeči, akce sestává ze seznamu účastníků, parametrů, podmínky pro spuštění a definice změny stavů po vykonání akce.

Bohužel má metoda DisCo řadu problémů:

Jak bylo vidět, použitá syntaxe je příliš vzdálena současným trendům, kterým vévodí notace UML. Matematický popis je nepřehledný, málo intuitivní a příliš komplikovaný.

DisCo má omezené výrazové prostředky. Popis celé reality sestává pouze ze tříd, jejich vztahů a akcí, což je chudší slovník než u jiných metod.

Model akcí příliš nezapadá do objektově orientovaného paradigmatu. Je obtížné DisCo konvertovat do nějakého konkrétního programovacího jazyka.

Tento seznam ukazuje, že metoda DisCo příliš perspektivní není.

Výhody	Snaha popsat vzory v jejich abstraktní podobě
Nevýhody	Omezené výrazové prostředky
	Špatná převoditelnost matematického modelu do reálného programovacího jazyka
	Neintuitivní notace, absence grafické notace

Tabulka 4 – Výhody a nevýhody metody DisCo

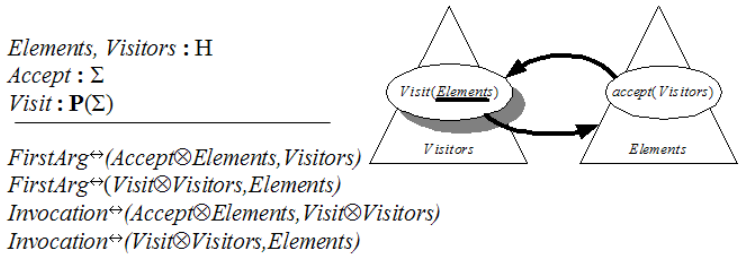
4.4 LePUS – Language for Patterns Uniform Specification

LePUS (jazyk pro jednotnou specifikaci vzorů, language for patterns uniform specification) je deklarativní jazyk, který se velkému zájmu těší především v akademické komunitě. Bylo o něm vydáno mnoho článků, viz např. [Eden's publications].

4.4.1 Hlavní ideje

LePUS je formální specifikační jazyk pro objektově orientovaný design a architekturu, který je postavený na několika málo stavebních blocích, které se vyskytují při designu prakticky všech objektově orientovaných programů. LePUS je matematickou strukturou $M = \langle U, R \rangle$, kde U je množina základních entit (atomů), z nichž každá je buďto třídou nebo metodou, a R je množina základních relací mezi atomy.

LePUS zavádí velké množství termínů, z nichž většina je definována matematicky na základě množinových operací. Mezi základní pojmy patří hierarchie, klany, kmeny, uniformní množiny prvního řádu, uniformní množiny vyšších řádů, totální relace atd. Vedle striktního matematického jazyka přichází LePUS také s grafickou notací, jejíž příklad ukazuje následující obrázek:



Obrázek 5 – Příklad návrhového vzoru Visitor (zdroj: [Eden, 03])

Nalevo je návrhový vzor Visitor popsán pomocí striktním matematickým jazykem, zatímco napravo je zobrazen jeho grafický ekvivalent. Na obrázku jsou k vidění následující elementy:

Trojúhelník představuje *hierarchii*, tedy množinu tříd ve vztahu dědičnosti.

Elipsa se stínem představuje množinu funkcí se stejnou signaturou, které se mohou v rámci dědičné hierarchie překrývat.

Šipky s nepřerušovanými čarami naznačují volání metody jiného objektu.

Matematický i vizuální slovník metody LePUS je poměrně rozsáhlý a do detailů se mu věnují výše odkázané práce, myslím však, že uvedený příklad demonstroval hlavní ideje poměrně dobře.

4.4.2 Shrnutí, výhody, nevýhody

Hlavní sílou LePUSu je hutnost zápisu při zachování velké vyjadřovací schopnosti. U této metody se poprvé setkáváme s entitami vyšší kvality, kdy např. jeden trojúhelník zachycuje libovolně složitou hierarchickou strukturu tříd – je jedno, zda má třída dva nebo dvacet potomků, stejně jako je jedno, zda jsou tito potomci přímí nebo nepřímí; jediným důležitým kriteriem je zde vztah dědičnosti. Tím se LePUS liší od běžných objektových notací typu UML, které umožňují zachytit pouze jeden konkrétní příklad dané hierarchie, a je tedy velmi zdařilým pokusem o formalizaci vzorů na té nejabstraktnější úrovni. Obsahuje dostatečný počet entit vyšších řádů, které mají zcela jinou sémantiku než objektové notace typu UML.

Je dobře, že LePUS přichází s grafickou notací, protože jazyk na bázi predikátové logiky by byl pro řadu vývojářů jen těžko srozumitelný. Přesto považuji notaci LePUSu ve srovnání s UML za poměrně obtížnou, neboť k jejímu plnému pochopení je potřeba vstřebat řadu nových pojmů, které nejsou v jiných vizuálních jazycích běžně dostupné. Situaci navíc komplikuje skutečnost, že LePUS není podporován ze strany nástrojů.

Další nevýhodou je relativně velká vzdálenost od cílového zdrojového kódu. LePUS sice kromě prostého popisu vzorů má sloužit i k jejich praktickému zakotvení v cílovém implementačním prostředí, ke kontrole shody implementace s daným vzorem (*reasoning*, *pattern conformance*) a také k detekci vzorů ve stávajících systémech, sám o sobě je však velmi vysokoúrovňovou záležitostí a konverze do konkrétního kódu není triviální. Cíle jsou jedna věc, jejich realizace věc druhá.

Celkově vzato se ale jedná o jednu z nejzajímavějších formalizačních technik.

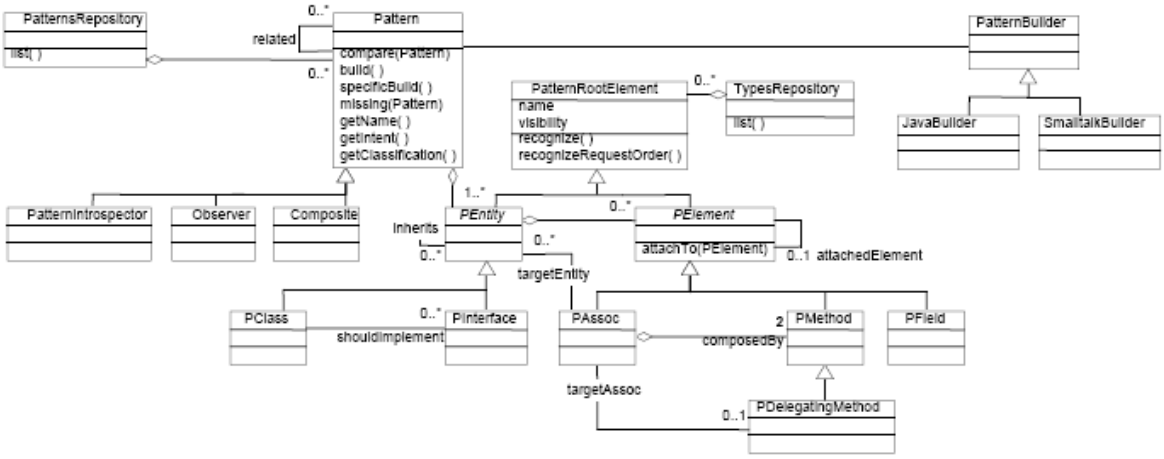
Výhody	Bohaté výrazové prostředky pro většinu myslitelných návrhových situací (bez problémů dostačuje pro popis GoF vzorů, ale i řady dalších)
	Vedle rigorózní matematické notace je připravena i výrazně intuitivnější grafická reprezentace
	Schopnost popsat vzory v jejich podstatě
Nevýhody	Nejsou zabudovány žádné prostředky pro konverzi LePUS modelu do cílového implementačního prostředí
	Grafická notace, jakkoliv intuitivní, je poměrně vzdálená UML, což snižuje její praktickou použitelnost

Tabulka 5 – Výhody a nevýhody LePUSu

4.5 PDL – Patterns Detection Language

4.5.1 Úvod

Zatímco předchozí formalizační pokusy byly zaměřeny především na popis vzorů (ne vždy a ne zcela – např. LePUS uvažuje i o dalších použitích sebe sama, ale primární úlohou přeci jen zůstává prostá deskripce vzorů), PDL, Patterns Detection Language, míří trochu jinam. Jak už samotný název napovídá, kromě popisu jde v první řadě rovněž o generování cílového kódu a především o detekci vzorů ve stávajících systémech. PDL byl poprvé důkladněji popsán v práci [Albin-Amiot, Guéhéneuc, 2001].



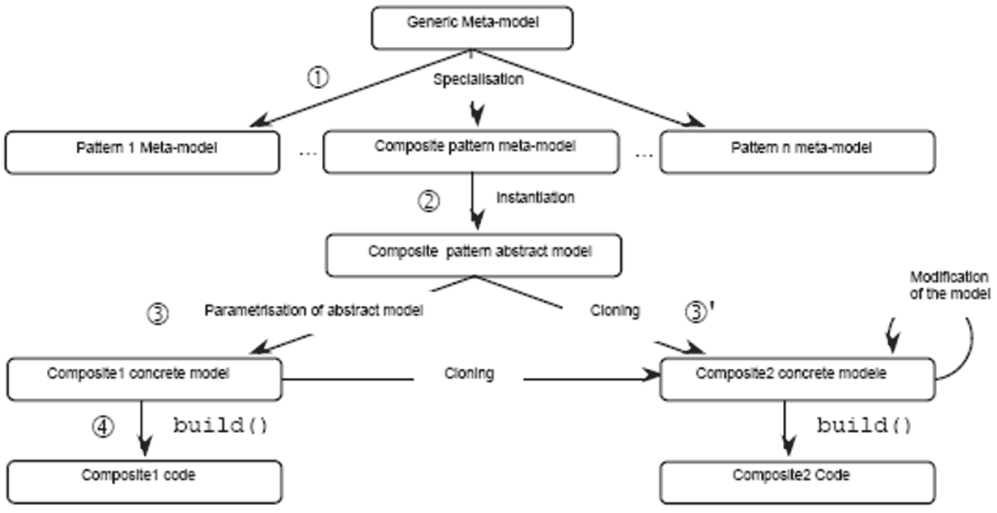
Obrázek 6 – Metamodel PDL (zdroj: [Albin-Amiot, Guéhéneuc, 2001])

Proč PDL vznikl? Autoři se domnívají, že formalizační pokusy lze zhruba zařadit do dvou hlavních kategorií. Ta první se snaží o formalizaci vzorů na úrovni zdrojového kódu (na předchozích stránkách bylo představeno několik takových pokusů, jmenovitě LayOM nebo zapsání vzorů do kódu pomocí atributů). Druhou kategorii tvoří metody stavějící nikoliv na programovacích jazycích, ale na modelovacích notacích (mezi příklady by patřily metody DisCo nebo LePUS). První kategorie nemůže mít ambici na vytvoření obecného popisného jazyka pro vzory, zástupci druhé kategorie naopak často představují problém pro praktickou implementaci vzorů v kódu.

PDL se snaží mít své základy v druhé kategorii, ale současně mít na zřeteli cílovou implementaci ve zdrojovém kódu. Konkrétně řečeno se snaží řešit problém vystopovatelnosti.

4.5.2 Hlavní ideje

Základními kameny PDL jsou tzv. entity a elementy, které jsou definovány v samém jádru meta-modelu (viz obrázek 6). Význam meta-modelování ve vztahu ke vzorům je v PDL takový, že existence meta-modelu zajišťuje existenci vzorů jako entit první třídy v generovaných modelech. PDL není prvním formalizačním pokusem založeným na meta-modelování, viz např. [Pagel, Winter, 1996], ale na rozdíl od všech předchozích uvádí konkrétní postupy pro generování programového kódu.



Obrázek 7 – Princip fungování PDL (zdroj: [Albin-Amiot, Guéhéneuc, 2001])

Jak PDL funguje, ukazuje příklad návrhového vzoru Composite:

1. V prvním kroku se *obecný metamodel* specializuje pro použití v daném konkrétním případě. Pokud by např. měl být strom objektů neměnný, bylo by nutno přidat novou entitu nazvanou třeba ImmutablePClass jako potomka třídy PClass. Pro základní vzor Composite je však obecný model sám o sobě dostatečný.
2. V druhém kroku dochází k instanciaci meta-modelu. Výsledek se v terminologii PDL nazývá *abstraktní model*, protože v sobě neobsahuje žádné informace o aplikačním kontextu. Abstraktní model drží veškeré informace vztahující se k návrhovému vzoru jako takovému.
3. Třetí krok transformuje abstraktní model do *konkrétního modelu*. Konkrétní model se od abstraktního liší tím, že je upraven pro konkrétní aplikační kontext. Diagramy uváděné v základní knize [GoF, 1994] jsou prakticky konkrétními modely v terminologii PDL.
4. Finálním krokem je konverze konkrétního modelu do *cílového implementačního jazyka*. Jedná se o analogickou situaci jako když je z UML generován zdrojový kód.

Podrobný příklad s konkrétními ukázkami grafických notací i s generovaným výstupem v jazyce Java je možno nalézt v práci [Albin-Amiot, Guéhéneuc, 2001].

4.5.3 Detekce vzorů

Důležitou součástí PDL je i detekce vzorů v programovém kódu. Přístup PDL k detekci je poněkud jiný než by bylo možno očekávat: nejedná se o obecně použitelnou detekci ve stávajících zdrojových kódech, ale o rozpoznávání vzorů *vygenerovaných pomocí PDL* samotného. PDL tedy ve zdrojovém kódu nepožaduje žádné speciální značky (tak jako např. technika zachycení vzorů v kódu pomocí atributů), ale pokud byl daný zdrojový kód vygenerován pomocí meta-modelu, abstraktního a konkrétního modelu PDL, mělo by být možné takový vzor automaticky rozpoznat.

Rozpoznávání vzorů je algoritmicky daleko složitější úlohou než jejich generování, ale ve velmi hrubých rysech se vlastně jedná o inverzní proces: na základě artefaktů zdrojového textu se vytvoří konkrétní model a každý abstraktní model pak zkoumá, zda náhodou není daný konkrétní model jeho instancí. Pokud ano, došlo k rozpoznání vzoru.

4.5.4 Shrnutí, výhody, nevýhody

Hlavním přínosem PDL je komplexní pohled na problematiku nástrojové podpory vzorů. Metamodel zajišťuje, že vzory budou v metodě PDL zcela plnoprávné objekty, abstraktní model konkretizuje meta-model pro specifické požadavky daného návrhového vzoru, konkrétní model potom představuje určitou alternativu k UML a připravuje abstraktní model na reálné požadavky a omezení cílové aplikace, zatímco poslední krok vede k vygenerování finálního zdrojového kódu.

Ačkoliv komplexní přístup PDL působí sympaticky, má také několik nevýhod. Tou první je věčný problém modelování: jak generovaný kód bezproblémově integrovat s uživatelským kódem. Autoři PDL spoléhají na JavaXL, nástroj provádějící source-to-source transformace, což ale neřeší problém v jeho základu.

Druhým problémem je obtížná detekce vzorů. Jak již bylo řečeno, PDL se snaží, aby v kódu nemusely být speciální „značky“ explicitně určující, o jaké části vzoru se jedná. Na jednu stranu se jedná o pěknou myšlenku, na druhou stranu to s sebou přináší velké praktické komplikace. Což je trochu ironií, vzpomeneme-li si, co znamená zkratka PDL…

Výhody	Komplexně pojatá problematika vzorů
	Schopnost vysokoúrovňového popisu a současně generování cílového implementačního kódu
	Rozšiřitelnost díky velmi obecnému meta-modelu
Nevýhody	Problém integrace uživatelského a generovaného kódu
	Obtížná detekce vzorů, protože ty nejsou ve zdrojovém kódu nijak explicitně zachyceny

Tabulka 6 – Výhody a nevýhody PDL

4.6 Diagramy omezení a třívrstvé modelování

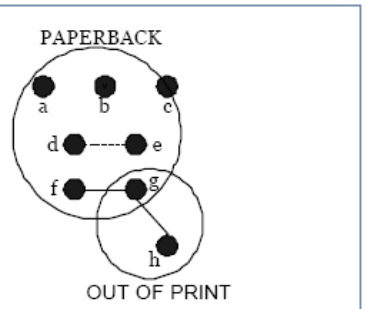
Diagramy omezení (*constraint diagrams*) popisují vzory jejich rozdělením do třech samostatných modelů: modelu *rolí*, *typů* a *tříd*. Podrobné informace lze získat v práci [Lauder, Kent, 1998].

4.6.1 Hlavní ideje

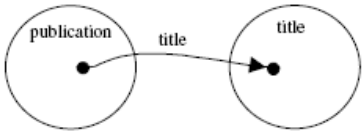
Model rolí je nejabstraktnější, zachycuje pouze podstatu vzoru a záměrně přehlíží detaily specifické pro danou aplikační doménu. Typový model upřesňuje model rolí o abstraktní stavy a rozhraní operací. Model tříd nakonec implementuje typový model.

Základem notace diagramů omezení jsou Vennovy diagramy, tedy jazyk velmi podobný tomu, který je používán v teorii množin. Na obrázku 8 např. elementy *a*, *b* a *c* znázorňují samostatné členy množiny, *d* a *e* nejsou nutně rozdílné a *f*, *g* a *h* přestavují jeden element, který může v daném časovém okamžiku existovat v jedné a právě jedné pozici. Vztah mezi elementy naznačuje šipka, např. na obrázku 10 je naznačen vztah „každá publikace má svůj titul“.

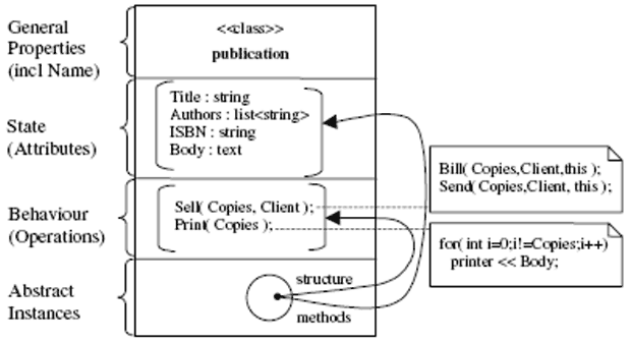
Tato grafická notace je nakonec propojena s diagramem tříd UML. Ten standardně obsahuje tři oddíly: první pro jméno a další obecné vlastnosti, druhý pro atributy a třetí pro operace. V práci [Lauder, Kent, 1998] je navržena notace, která by přidala čtvrtý oddíl obsahující právě diagramy omezení. Výsledek je zachycen na obrázku 9.



Obrázek 8 – Diagramy omezení (zdroj: [Lauder, Kent, 1998])

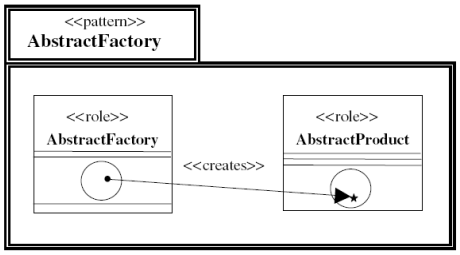


Obrázek 10 – Vztah "každá publikace má svůj titul" (zdroj: [Lauder, Kent, 1998])



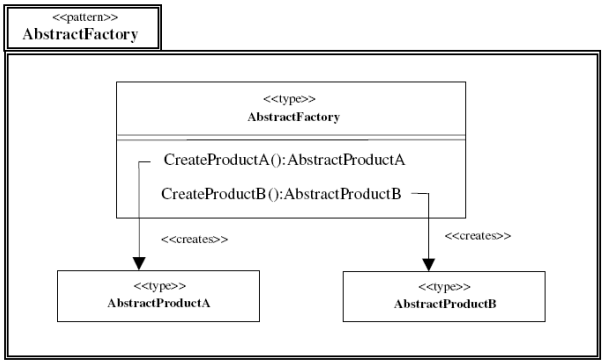
Obrázek 9 – Symbol třídy rozšířený o oddíl pro abstraktní instance (zdroj: [Lauder, Kent, 1998])

4.6.2 Příklad



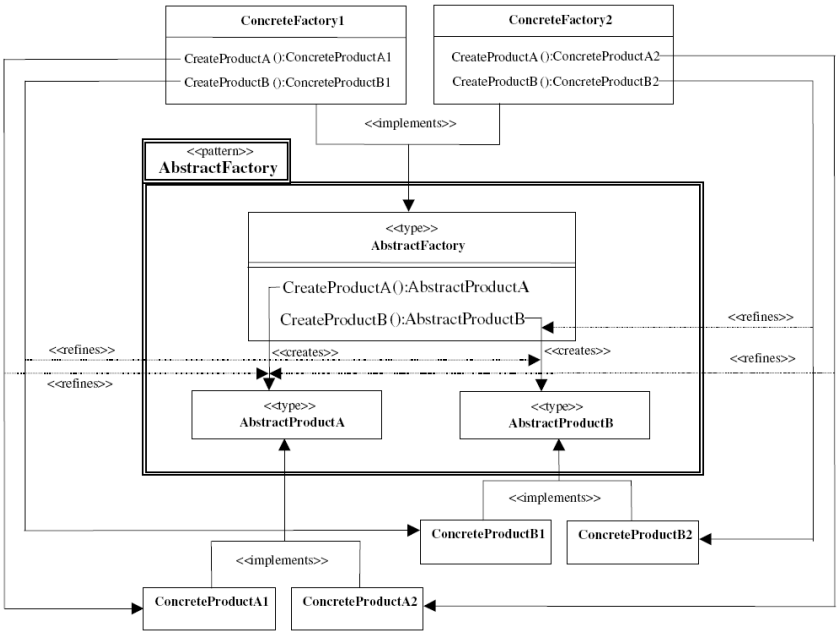
Obrázek 11 – Abstract Factory jako model rolí (zdroj: [Lauder, Kent, 1998])

Na nejvyšší úrovni je diagram zbaven veškerých detailů a zachycuje návrhový vzor v jeho podstatě. Na uvedeném příkladu je demonstrována abstraktní továrna, kde role AbstractFactory vytváří mnoho rolí AbstractProduct. Jak píší autoři, „[ve smyslu struktury a chování] tento model zachycuje vzor, celý vzor a nic než vzor“.



Obrázek 12 – Abstract Factory jako typový model (zdroj: [Lauder, Kent, 1998])

Typový model je upřesněním modelu rolí (podrobný převod je načrtnut v [Lauder, Kent, 1998]). V modelu se už neobjevuje čtvrtý oddíl zachycující odpovídající diagramy omezení, rovněž místo o rolích zde již mluvíme o typech.



Obrázek 13 – Převedení typového modelu do modelu tříd (zdroj: [Lauder, Kent, 1998])

Posledním krokem je převedení typového modelu do diagramu tříd, jak demonstruje obrázek 13. Pokud bychom ho překreslili do trochu přehlednější podoby, získali bychom diagram velmi podobný tomu v GoF katalogu.

4.6.3 Shrnutí, výhody, nevýhody

Tříúrovňové modelování umožňuje zachycení vzorů jak v jejich podstatě, tak v určitých detailech. Jednoznačným plusem této notace je částečná integrace s UML, což zvyšuje čitelnost diagramů, na druhou stranu samotné diagramy omezení připomínají spíše matematický zápis. Největší nevýhodou této notace je, že vůbec neřeší finální implementaci. Můžeme se na ni dívat jako na určitou nadstavbu nad UML, směrem od UML ke kódu však nepřináší žádné řešení.

Výhody	Komplexně pojatá problematika vzorů
	Integrace s UML
Nevýhody	Neřeší konečnou implementaci
	Diagramy omezení se příliš nepodobají zaběhnutým notacím

Tabulka 7 – Výhody a nevýhody diagramů omezení

4.7 Shrnutí

V této kapitole bylo popsáno a demonstrováno několik formalizačních technik, které často přinesly řadu originálních myšlenek. V zásadě byly notace zaměřeny na tři rozdílné oblasti: na implementaci, na vysokoúrovňový popis vzorů a na něco mezi těmato dvěma póly, přičemž se některé notace zaměřovaly i na více než jednu oblast.

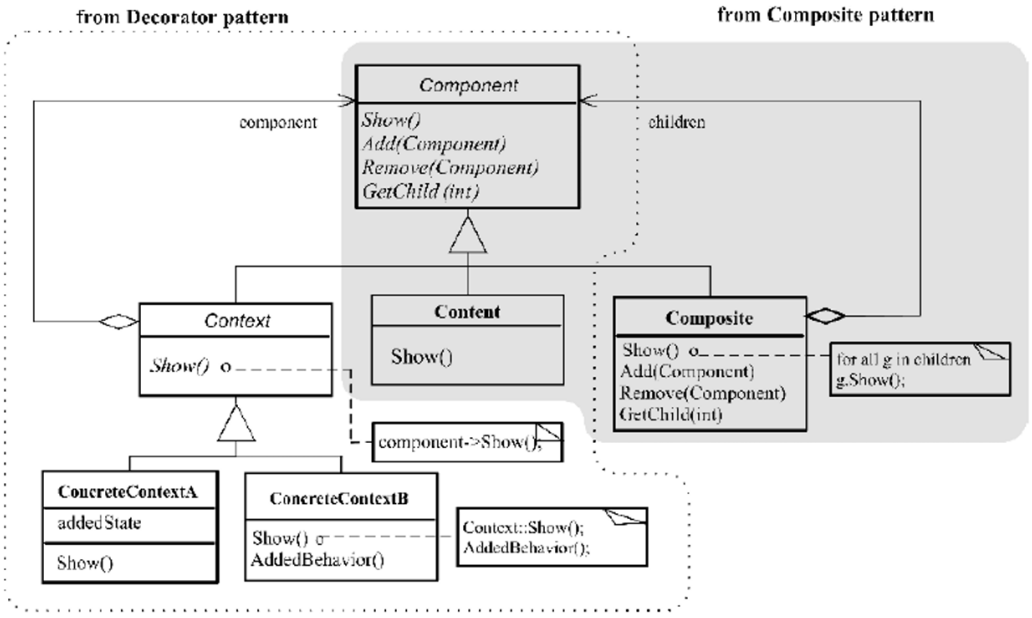
Výrazným rysem všech notací byla poměrně komplikovaná a neintuitivní syntaxe, která by před používáním vyžadovala určité studium. Druhým charakteristickým znakem byla nulová nebo velmi nízká podpora v současných softwarových nástrojích. Spíše než praktickou využitelnost tedy tyto techniky přinesly několik zajímavých myšlenek, z nichž některé budou uplatněny v dalším textu práce.

5 Formalizační techniky založené na UML

V kapitole 3.5 bylo popsáno, proč UML není vždy úplně vhodnou notací pro popis návrhových vzorů. Zkráceně řečeno, UML jako jakákoliv jiná objektová notace postrádá výrazové prostředky pro popis komplexnějších entit typu „množina tříd“ a primárně se používá pro popis *instance* návrhového vzoru, ne pro popis návrhového vzoru jako takového. Rovněž však bylo řečeno, že použití UML by mělo nezanedbatelné výhody – jeho obecná rozšířenost a snadná srozumitelnost v komunitě softwarových architektů a vývojářů by výrazně posílila komunikační funkci vzorů, navíc dnes na trhu existuje celá řada CASE nástrojů s kvalitní podporou UML. Snad právě proto se objevilo několik pokusů o integraci návrhových vzorů do této de facto standardní objektové notace. Kapitola vychází z práce [Dong, 2002].

5.1 Zachycení vzorů pomocí Vennových diagramů

První notace, navržená Johnem Vlissidesem, spoluautorem knihy [GoF, 1994], zachycuje návrhové vzory v diagramu tříd pomocí Vennových diagramů[3]. Názornou ukázku zachycuje následující obrázek:



Obrázek 14 – Zachycení vzorů v UML pomocí Vennových diagramů (zdroj: [Dong, 2002])

Diagram tříd obsahuje dva návrhové vzory: Decorator a Composite[4]. Šedým stínováním je zachycen návrhový vzor Composite, zatímco „obtečkováním“ je vyznačen vzor Decorator.

Výhodou této notace je její velká přehlednost – pouhým pohledem je velmi dobře vidět, které třídy spadají do kterého návrhového vzoru. Tím ale bohužel dobré vlastnosti této notace více méně končí.

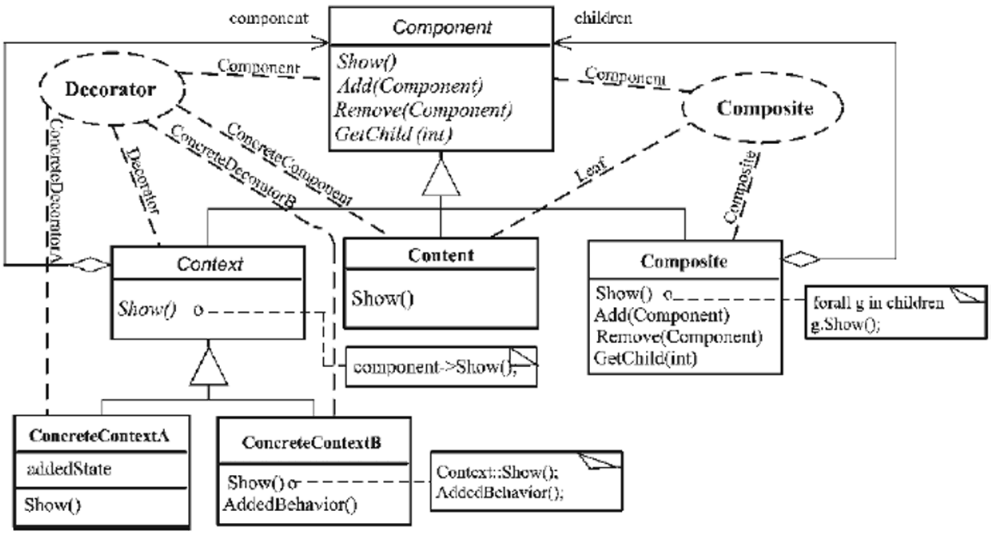
Hlavním problémem je škálovatelnost této notace. Na malých diagramech je velmi intuitivní, ale při rozsáhlejších modelech může být velmi obtížné zakreslit hranice návrhových vzorů tak, aby celé dílo zůstalo přehledné. Důležité je ještě jedno další omezení – zachycení vzorů pomocí Vennových diagramů neobsahuje prostředky pro vyjádření rolí, v jakých jednotlivé elementy vystu-

[3] Vennovy diagramy jsou grafickou notací používanou v teorii množin, viz např. http://en.wikipedia.org/wiki/Venn_diagram.
[4] Tyto vzory často vystupují společně, takže je jedná o reálný příklad.

pují. V uvedeném příkladu je celkem zřejmé, že třeba třída Composite vystupuje v roli kompozitního elementu vzoru Skladba, ale diagram tříd nemusí být takto „uhlazený“ (může být z jistých důvodů třeba nakreslen vzhůru nohama), navíc pokud budou používány termíny z cílové domény aplikace a název třídy Composite bude nahrazen třeba názvem Picture, nemusí být diagram vůbec přehledný.

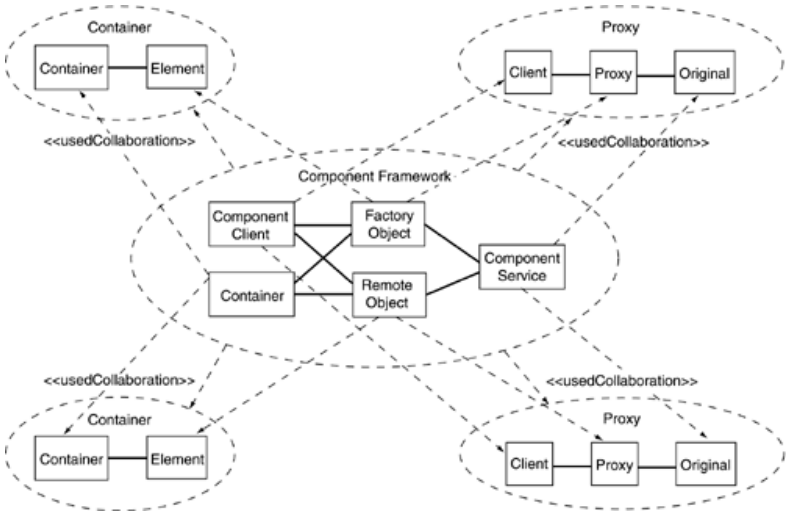
5.2 Využití notace pro kolaboraci

UML samo o sobě obsahuje prostředek pro popis rolí, v jakých třídy vystupují. Do diagramu tříd lze přidat notaci tzv. parametrizované spolupráce, jak ukazuje následující obrázek:



Obrázek 15 – Zachycení vzoru pomocí notace pro kolaboraci (zdroj: [Dong02])

Výhodou tohoto způsobu zápisu je, že explicitně zachycuje nejen názvy použitých návrhových vzorů, ale taky obsahuje propojení návrhového vzoru s konkrétní třídou, která ve vztahu ke vzoru vystupuje v určité roli. Také zde však nastávají problémy u rozsáhlejších diagramů, i když menší než v případě zápisu pomocí Vennových diagramů:

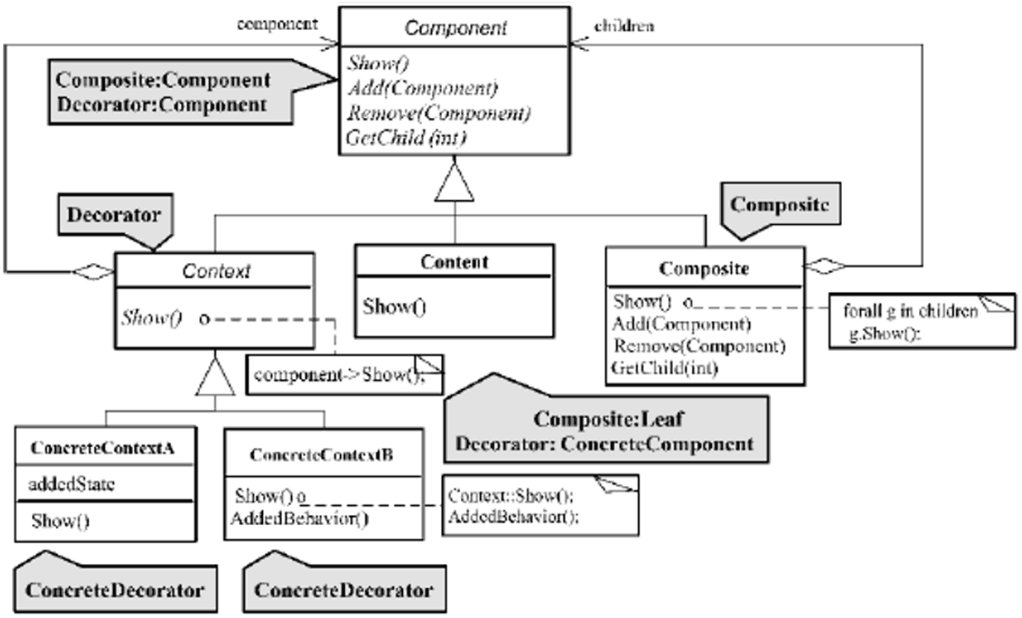


Obrázek 16 – Příklad komplexnosti diagramu při použití kolaborací (zdroj: [Yacoub, Ammar, 2003])

Přes tyto problémy je popis vzorů pomocí kolaboračních elementů v některých technikách použit, jako příklad může sloužit kniha [Yacoub, Ammar, 2003] věnující se metodice *Pattern-Oriented Analysis and Design* (POAD).

5.3 Anotace pomocí zápisu „vzor:role“

Aby se předešlo komplikovaným a málo přehledným diagramům při použití přerušovaných čar, navrhl John Vlissides notaci nazvanou „pattern:role annotation“, viz [Vlissides, 1998]. Základní myšlenou je každou třídu označit „štítkem“, který by nesl název vzoru a roli, kterou třída v daném vzoru zastává. Pokud nemůže dojít k nejasnostem, může být navíc uveden pouze název návrhového vzoru:



Obrázek 17 – Anotace "vzor:role" (zdroj: [Dong, 2002])

Vlissides tvrdí, že tento zápis je velmi dobře přehledný a řeší problémy předchozích zápisů. Vystínované štítky určitě přehlednosti napomáhají, ale jeden problém zůstává: ne vždy musí v určité roli vystupovat třída jako celek. Stává se, že svou roli hraje také metoda nebo atribut a anotace „vzor:role“ pro tento případ nenabízí žádné řešení. Proto existují ještě další pokusy.

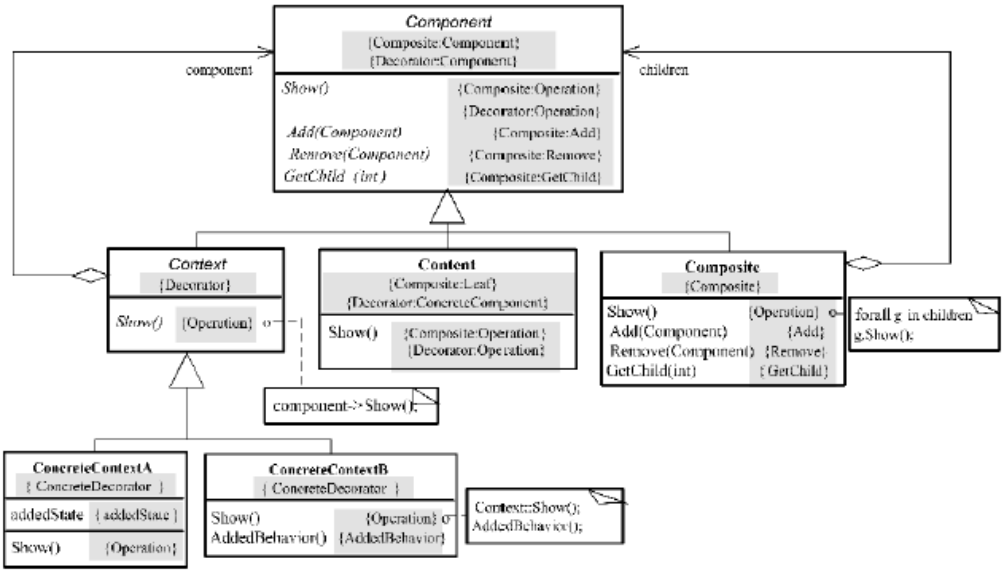
5.4 Anotace pomocí tagovaných hodnot

Vennovy diagramy a odvozené zápisy řeší, jak v UML zachytit návrhový vzor. Notace založená na kolaboračních grafických elementech a „pattern:role“ notace kromě zachycení vzoru obsahuje i prostředky pro popis toho, jakou roli v rámci daného vzoru třída zastává. Žádný z doposud jmenovaných způsobů však neřeší situaci, kdy nějakou úlohu v návrhovém vzoru hraje metoda nebo atribut. [Dong, 2002] navrhuje notaci založenou na používání tzv. tagovaných hodnot (*tagged values*), u kterých je potřeba zmínit určité historické souvislosti.

V UML specifikaci verze 1.3 byly tagované hodnoty jedním ze tří mechanismů rozšiřitelnosti – vedle nich šlo použít ještě stereotypy a omezení (*constraints*). Tagovaná hodnota byla jednoduchým párem „jméno-hodnota“, která mohla být přiřazena k libovolnému elementu bez nutnosti

definovat zvláštní stereotyp[5]. Specifikace UML 1.4 toto chování označila jako zavržené (deprecated) a podporovala ho pouze z důvodů zpětné kompatibility. V UML 2.0 se z tagované hodnoty stává atribut přiřazený stereotypu. („When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values.“ – [UML 2.0 Superstructure, str. 649]). Jinými slovy není možné, aby byla nějakému elementu přiřazena tagovaná hodnota, která by nebyla postavena na stereotypu. Dongův záznam návrhových vzorů spoléhá na tagované hodnoty tak, jak byly popsány ve verzi UML 1.3, na což je potřeba při čtení zbytku této podkapitoly brát ohled.

Dongův přístup využívá toho, že dvojici tag-hodnota lze použít pro zápis typu „vzor:role“:



Obrázek 18 – Anotace pomocí tagovaných hodnot (zdroj: [Dong, 2002])

U každého elementu, ať už se jedná o třídu, atribut nebo metodu, lze uvést tagovanou hodnotu ve tvaru {název_vzoru:role_elementu_ve_vzoru}. [Dong, 2002] dále rozebírá, jak lze zvýšit přehlednost takovýchto diagramů – použít lze stínování tagovaných hodnot (tak jako na ukázce), kombinaci s Vennovými diagramy nebo dokonce vytvoření nového oddílu v symbolu pro třídu, který by byl vyhrazen výhradně tagovaným hodnotám popisujícím vzory. Praktická aplikace je popsána v práci [Dong, Yang, 2003], kde je představen UML profil podporující zápis pomocí tagovaných hodnot.

Výhodou tohoto zápisu je poměrně dobrá expresivita při zachování rozumné přehlednosti v tom smyslu, že přes diagram nevedou desítky čar spojujících různé elementy na dvou protilehlých koncích. Na druhou stranu se o tomto způsobu zápisu ani nedá říci, že by byl na první pohled příliš „čitelný“. Je potřeba projít element po elementu, podívat se, k jakému vzoru patří a až na konci celého modelu si lze udělat komplexní obrázek o tom, z jakých vzorů se vlastně navrhovaný systém skládá.

Je na místě ještě zmínit další praktickou nevýhodu: uvedená syntaxe již není standardně v UML 2.0 podporována. Např. široce populární CASE nástroj Enterprise Architect s plnou podporou standardu UML 2.0 neobsahuje prostředky, jak Dongovu notaci zapsat.

[5] Metamodel UML 1.3 definuje vztah stereotyp-tagovaná hodnota jako kompozici, kde stereotyp může mít libovolně mnoho tagovaných hodnot, zatímco tagovaná hodnota má nepovinně jeden stereotyp (0..1). Podrobnosti viz specifikace UML 1.3 nebo [Kraval 01, str. 60].

5.5 Shrnutí

Snaha modelovat návrhové vzory v UML je opodstatněná kvůli rozšířenosti a velké „čitelnosti“ UML diagramů. Na předchozích stránkách bylo k vidění několik technik, které se snažily vzory do UML integrovat, a tyto pokusy většinou nebyly zcela marné. Vizuálně velmi přehledné jsou notace založené na Vennových diagramech a o přehlednost jde v mnoha případech především. Ostatní techniky se snaží zachytit i role, ve kterých jednotlivé elementy vystupují, vždy je to ale na úkor přehlednosti nebo rychlého pochopení diagramu.

V souvislosti s UML je ještě potřeba uvést, že některé názory považují tuto notaci za přímo vhodnou pro zápis vzorů bez nutnosti nějakých dalších úprav. Tento názor vychází z přesvědčení, že vývojář jakožto inteligentní bytost pochopí principy vzoru i z jeho instance a že tudíž není potřeba zavádět žádné nové speciální symboly. Čistě formálně by tedy obsahem této kapitoly mělo být ještě pojednání o UML jako takovém, z pochopitelných důvodů ho však nahrazuje alespoň tato krátká poznámka.

Tím je přehled formalizačních technik a notací ukončen. Předchozí dvě kapitoly ukázaly, že najít vhodnou notaci pro zachycení návrhových vzorů není vůbec jednoduché. Taková notace by měla být dostatečně obecná, aby umožnila zachytit vzory v jejich podstatě, současně by ale měla být jednoznačně převoditelná do implementačního jazyka, aby byla reálně využitelná. Dále se přidávají požadavky na srozumitelnost a přehlednost, čímž se situace ještě více komplikuje.

Některé notace se ideálu přiblížili více a některé méně, zcela uspokojivé řešení ale zatím nabídnuto nebylo. Tématu formalizace se ještě tato práce bude věnovat, nyní je ale nutno projít druhou velkou oblast, kterou je podpora návrhových vzorů v nástrojích. Následující kapitola demonstruje úroveň podpory vzorů v současných aplikacích a sedmá kapitola hledí trochu více do budoucnosti, když se zaměřuje na vizi tzv. softwarových továren, které by v nejbližší budoucnosti mohly významně ovlivnit způsob vývoje softwarových aplikací.

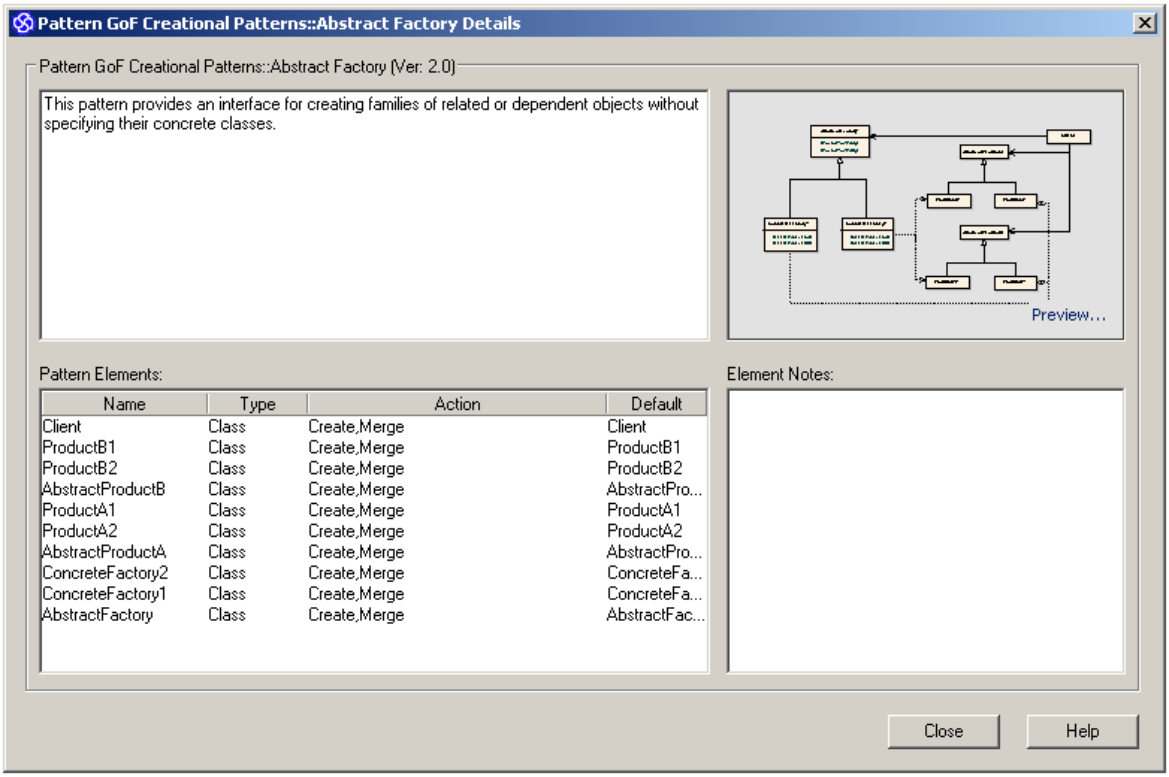
6 Podpora návrhových vzorů v současných nástrojích

Předchozí kapitoly ukázaly řadu formálních notací pro popis vzorů, které měly společné především to, že se žádná z nich při reálném vývoji příliš neujala a že žádná z nich nenalezla podporu v dnes rozšířených CASE nástrojích. Protože však tyto CASE nástroje obvykle podporují UML na velmi dobré úrovni a protože jsou často návrhové vzory v literatuře vyjádřeny právě touto notací, určitá podpora vzorů je ve většině CASE nástrojů přítomna. Kromě toho vznikly i specializované programy, které se na návrhové vzory přímo zaměřují. Tato kapitola demonstruje, na jaké úrovni lze dnes zhruba podporu ze strany nástrojů očekávat.

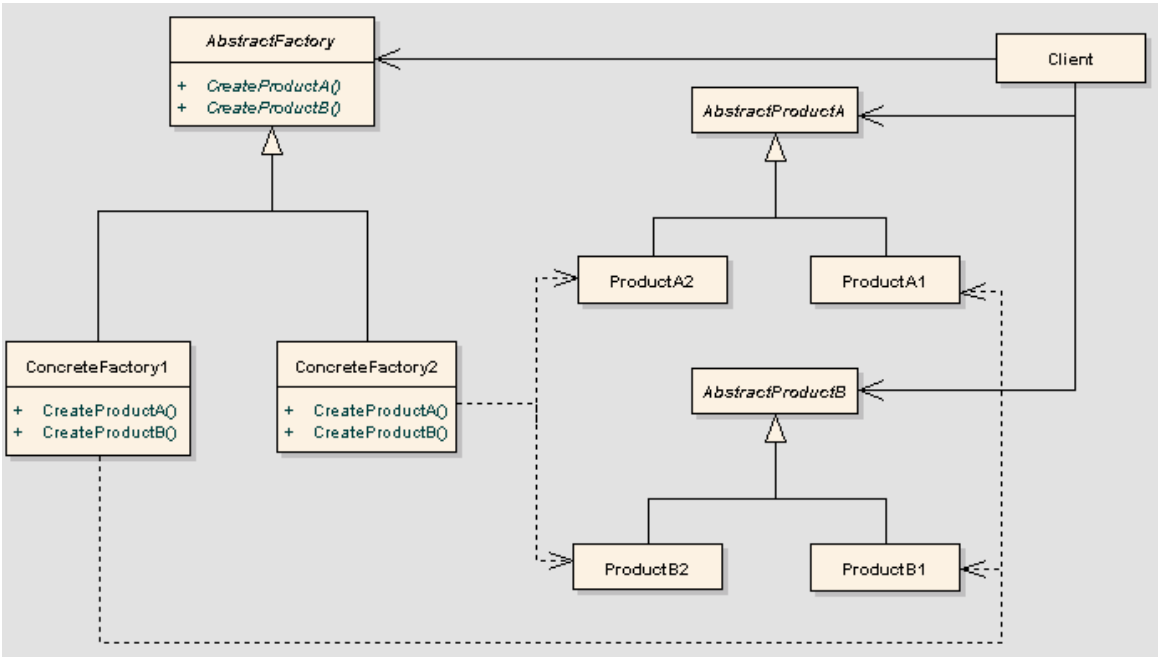
6.1 Podpora vzorů v CASE nástrojích

Na prvním místě je potřeba říci, že podpora návrhových vzorů v dnešních CASE nástrojích není samozřejmostí. Například PowerDesigner, který patří k nejrozšířenějším nástrojům své třídy, pro návrhové vzory neobsahuje vůbec žádnou podporu (testováno ve verzi 11). Některé progresivnější nástroje už je však podporují – jako příklad byl vybrán populární *Sparx Enterprise Architect*. Nakolik je tato podpora reálně použitelná, demonstruje příklad, v němž je úkolem implementovat strukturu odpovídající vzoru Abstract Factory s tím, že konkrétní továrny jsou implementovány jako Singletony a konkrétní produkty jsou vytvářeny továrními metodami.

Návrhový vzor se v nástroji Enterprise Architect vkládá ze záložky *Resource View*, kde se nachází složka *UML Patterns* a v ní zanořené složky *GoF Behavioral Patterns*, *GoF Creational Patterns* a *GoF Structural Patterns*. Prvním krokem je vložení návrhového vzoru Abstract Factory. Určitým omezením je, že nelze specifikovat počet konkrétních továren ani konkrétních produktů a musíme se spokojit s tím, co architekt vzoru připravil.

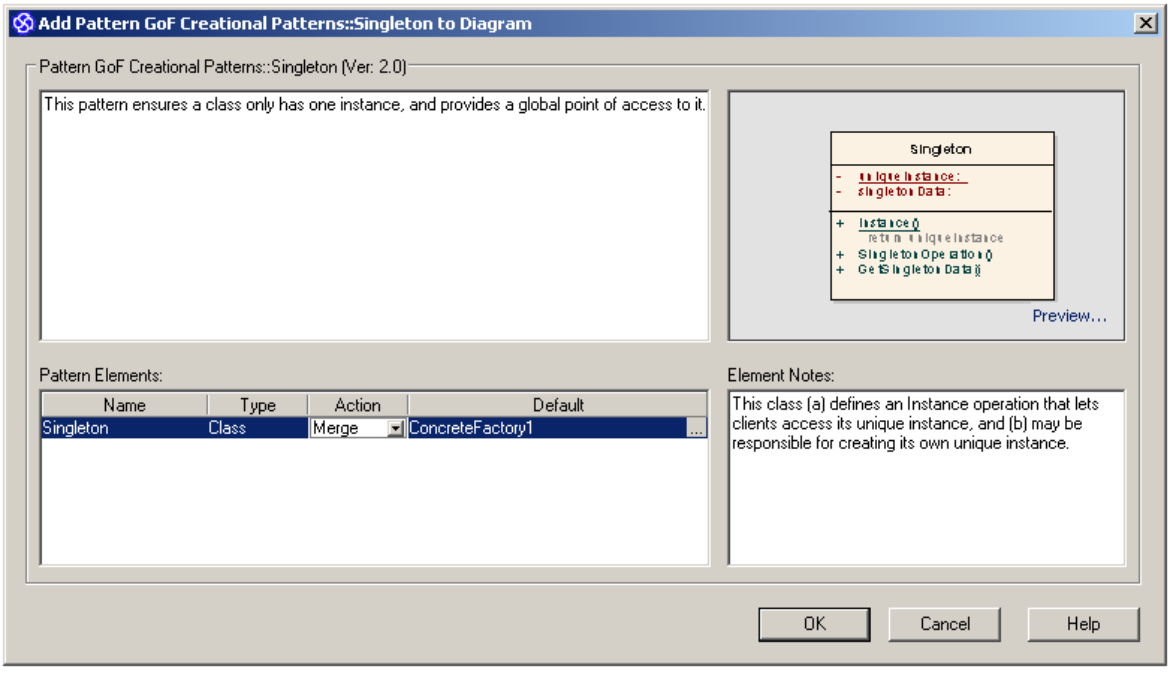


Obrázek 19 – Vložení návrhového vzoru v nástroji Enterprise Architect



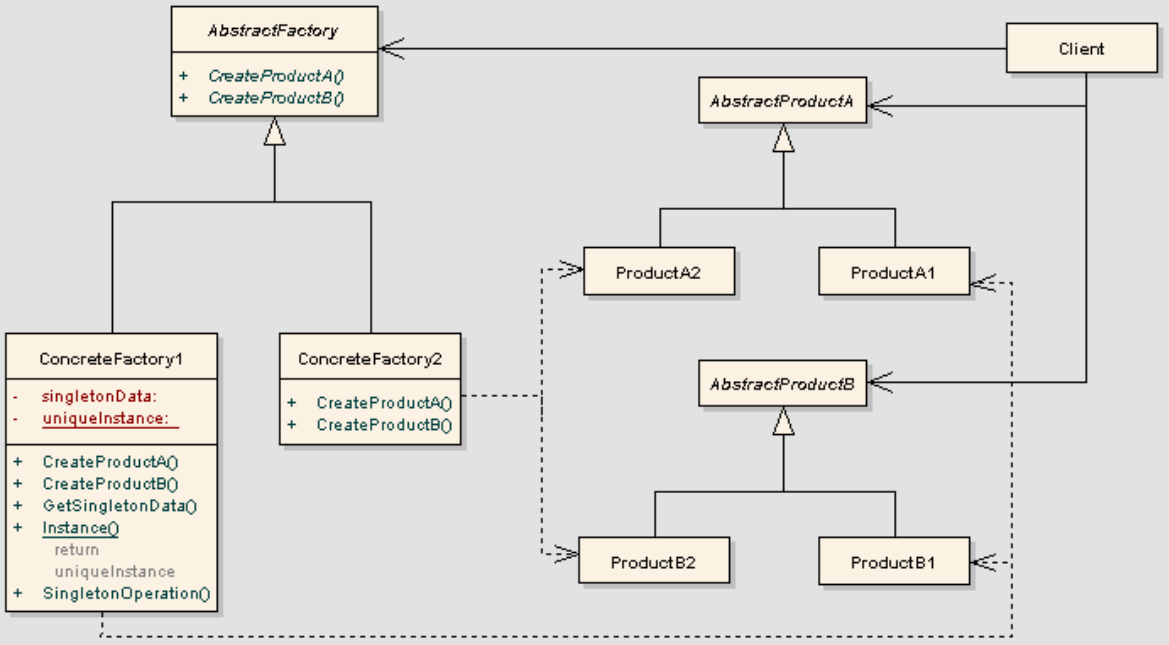
Obrázek 20 – Výsledek vložení abstraktní továrny

Druhým krokem je proměna konkrétních továren na Singletony. Enterprise Architect umožňuje vzory nejen vkládat, ale aplikovat je i na již hotové objekty diagramu. Dialog pro vložení Singletonu bude nastaven následovně:



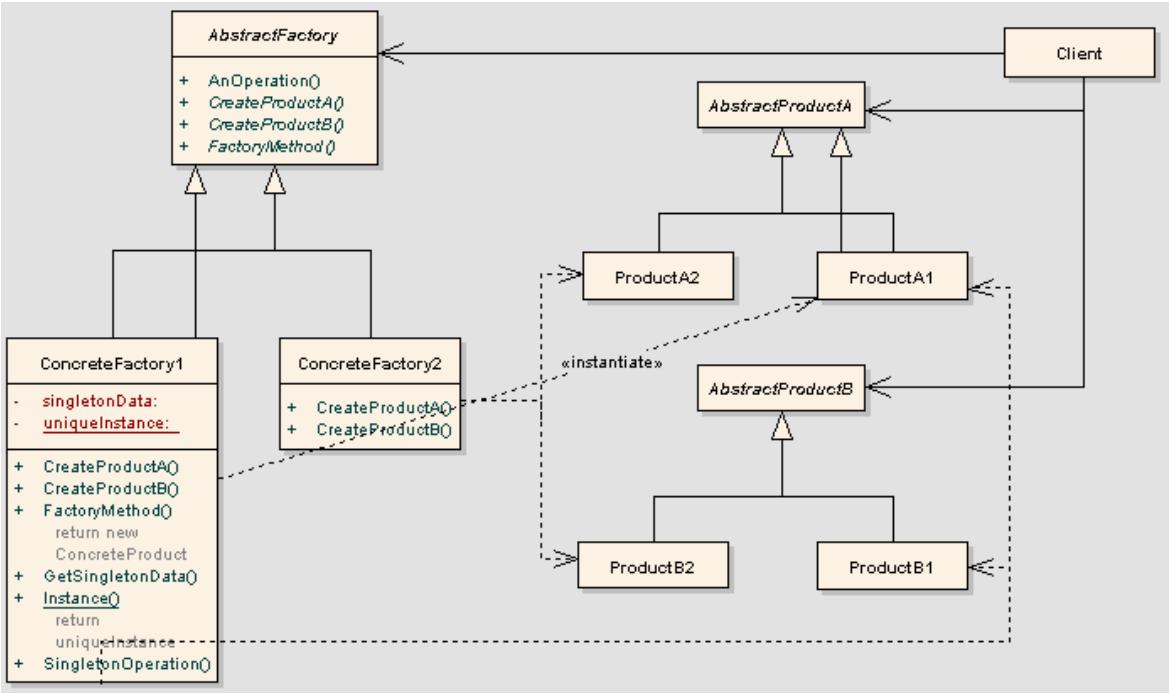
Obrázek 21 – Nastavení singletonu tak, aby ovlivnil již existující prvek diagramu

Akce byla změněna z Create na Merge a byl vybrán prvek diagramu, na který se má vzor aplikovat. Zde je výsledek:



Obrázek 22 – Výsledek aplikování vzoru Singleton

Aplikování Singletonu na druhou konkrétní továrnu by bylo zcela analogické, podívejme se proto, jak si Enterprise Architect poradí s aplikováním vzoru tovární metoda. Pokud budeme postupovat mechanicky a pomocí metody Merge přidáme vzor Factory Method, výsledek bude následující:



Obrázek 23 – Mechanické aplikování vzoru Factory Method

Ačkoliv „strojově“ dopadlo aplikování vzorů tak, jak mělo, příliš smyslu vzniklý diagram nedává. Metody CreateProductA() a CreateProductB() už totiž *byly* továrními metodami a aplikovat tento vzor znovu tak nemělo smysl. Zde jsou vidět praktické limity Enterprise Architectu: mechanická práce se vzory základním způsobem podporována je, v tomto posledním případě ale EA neměl jak poznat, že už daný vzor jednou aplikován je.

Může být otázkou, zda je vůbec potřeba, aby CASE nástroj rozuměl sémantice vnikajícího modelu. Není to nakonec vývojář, který musí vzory ovládat, aby je mohl smysluplně aplikovat? Nepochybně je jeho chyba, když se návrhový vzor pokouší aplikovat dvakrát, což ovšem neznamená, že by se nástroj neměl snažit těmto chybám předejít. U aplikace MS Word také nikdo nezpochybňuje funkci automatické kontroly pravopisu, ačkoliv je to primárně člověk, kdo by měl svůj jazyk dobře ovládat.

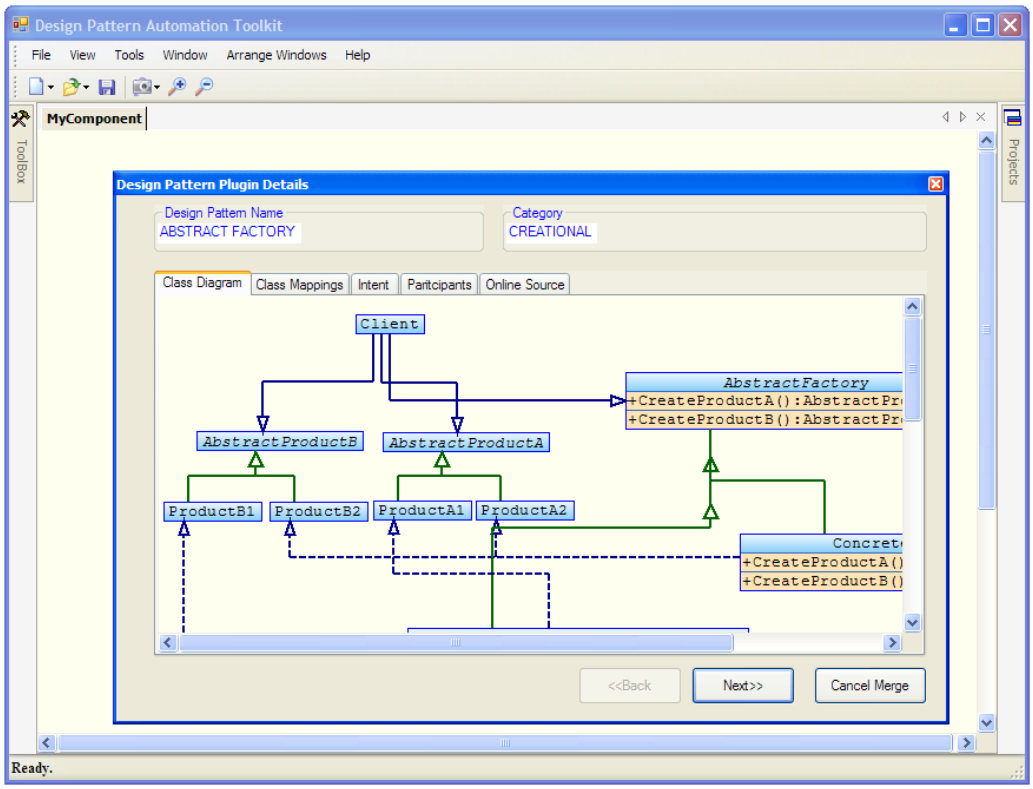
V kontrole sémantiky vytvářeného modelu tedy Enterprise Architect selhává, po zdárném vytvoření modelu však nabízí plný komfort CASE nástroje, takže lze snadno s diagramem manipulovat, je možno si nechat vygenerovat zdrojový kód, podporován je round-trip engineering a podobně. Importovat lze rovněž vzory nové, jejich vytvoření není obtížné.

Podpora vzorů v Enterprise Architectu je tedy poměrně použitelná, ovšem poněkud nepružná a „jednosměrná“ v tom smyslu, že když už je jednou návrhový vzor vložen, Enterprise Architect se dále k modelu chová jako ke standardnímu class diagramu.

6.2 Speciální nástroje

Zatímco v Enterprise Architectu byla podpora návrhových vzorů jen jednou z mnoha doplňkových funkcí, existují i nástroje, které se na vzory zaměřují primárně. Pro tuto kapitolu jako příklad poslouží DPAToolkit (Design Patterns Automation Toolkit), což je aplikace volně dostupná z adresy http://dpatoolkit.sourceforge.net/.[6] Implementovaný příklad bude stejný jako v předchozí kapitole.

Vložení návrhového vzoru se provádí z menu Tools Add Design Patterns, kde je předpřipraveno 23 vzorů z GoF katalogu, ale nástroj umožňuje importovat i vzory vlastní (interně jsou vzory uloženy v jazyce XML). Po zvolení vzoru Abstract Factory se zobrazí následující dialog:

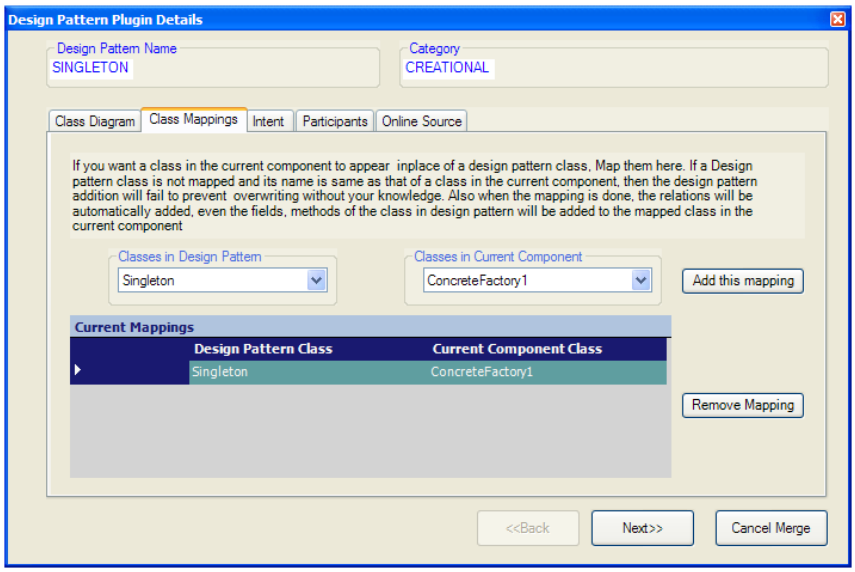


Obrázek 24 – Vložení návrhového vzoru Abstract Factory

[6] Použita byla verze 0.24.0.

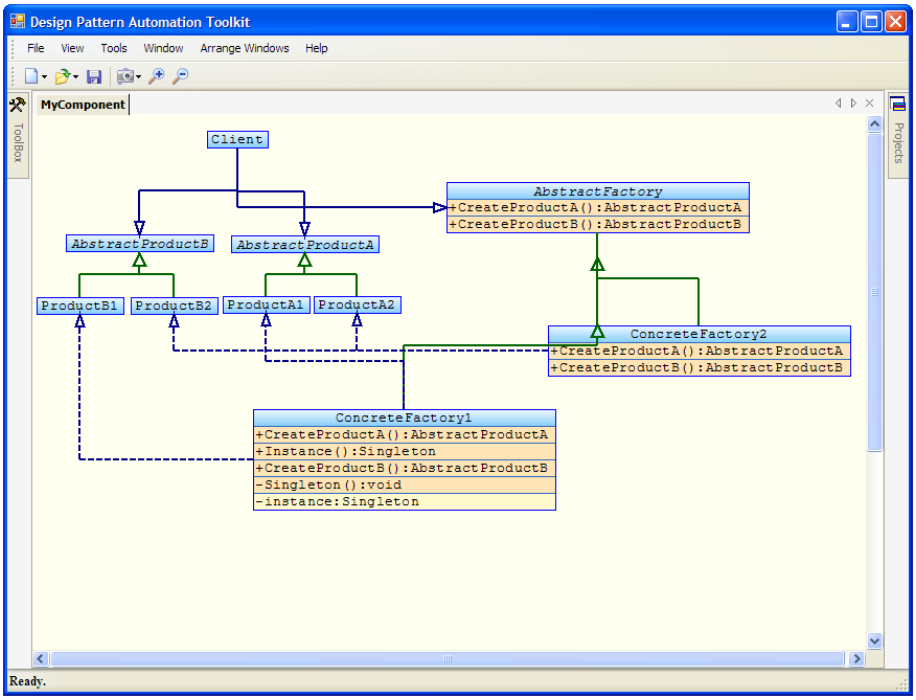
Přes jiné uživatelské rozhraní je zde vidět poměrně velká podobnost s odpovídajícím dialogem Enterprise Architectu. Také zde si není možno vybrat, kolik abstraktních nebo konkrétních produktů se má vytvořit a v podstatě dojde k pouhé kopii předpřipraveného vzoru do diagramu. Výsledkem je class diagram, který umožňuje podobnou manipulaci jako v běžném CASE nástroji.

Druhým krokem je aplikování vzoru Singleton na konkrétní továrny:



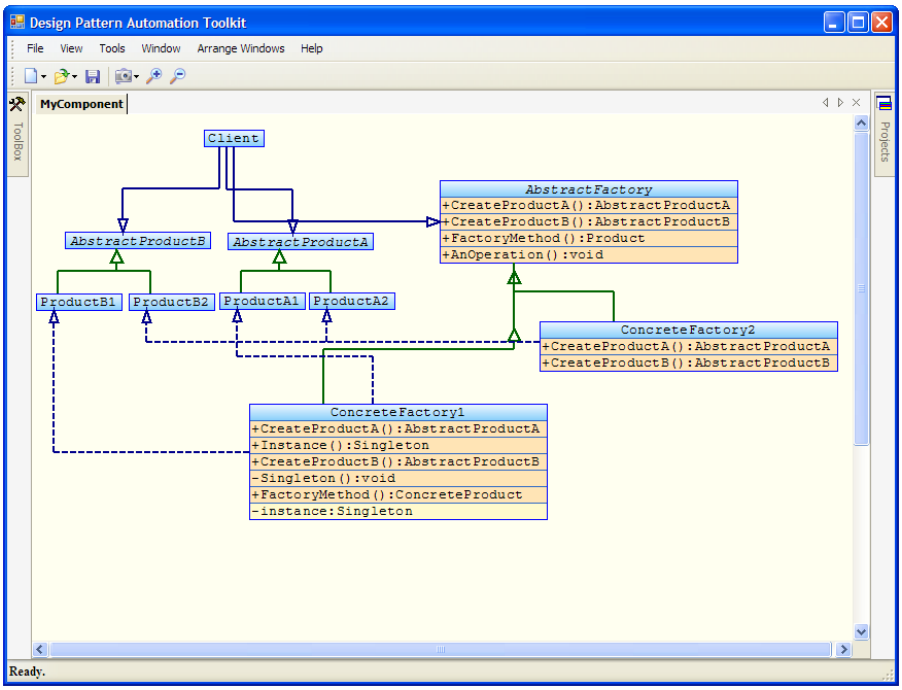
Obrázek 25 – Aplikování vzoru singleton na konkrétní továrnu

Na záložce Class Mappings je potřeba definovat, že třída označená jako Singleton z návrhového vzoru Singleton bude aplikována na třídu ConcreteFactory1. Po kliknutí na Next nástroj zkontroluje, zda není toto mapování v nějakém konfliktu – v tomto případě není – a po kliknutí na Merge je návrhový vzor Singleton vložen do diagramu. Výsledek je následující:



Obrázek 26 – Výsledek aplikování vzoru Singleton

Při snaze o duplicitní aplikování vzoru Factory Method se bohužel DPAToolkit zachová podobně špatně jako Enterprise Architect:



Obrázek 27 – Po aplikování vzoru Factory Method

Diagram sice vizuálně vypadá lépe (z ConcreteFactory1 vede do třídy ProductA1 pouze jedna šipka), ale na druhou stranu nedošlo ke korektnímu namapování vzoru na existující diagram a metoda ConcreteFactory1.FactoryMethod() chybně vrací ConcreteProduct místo správného ProductA1. Aplikace je sice zatím pouze v beta verzi, toto je však dost zásadní záležitost.

DPAToolkit přichází s podporou generování zdrojového kódu v jazycích C++, C#, Visual Basic .NET a Java a také s jejich reverzním inženýringem, kvalita a použitelnost je však značně nižší než u vyspělých CASE nástrojů.

Celkově je DPAToolkit spíše zajímavým než užitečným nástrojem. Využití by mohl najít při výuce vzorů, pro reálný vývoj ale chybí řada nástrojů běžně dostupných v plnohodnotných CASE nástrojích.

6.3 Jiné přístupy

Na trhu existují i další, těžko zařaditelné produkty, které práci s návrhovými vzory nějakým způsobem podporují. Příkladem je *Design Pattern Framework*[7] – jedná se o balíček dokumentů, projektů Visual Studia a referenční implementace.

Design Pattern Framework obsahuje:

> 23 GoF vzorů implementovaných jako projekty ve Visual Studiu 2005. Každý vzor je implementován na jednom strukturálním příkladu, na jednom příkladu z reálného světa a v třetí implementaci je vzor optimalizován pro .NET Framework 2.0 (celkově je tedy v produktu 69 implementací GoF vzorů).
>
> Přiloženy jsou UML diagramy všech GoF vzorů ve formátu Microsoft Visio.
>
> GoF vzory jsou stručně popsány v PDF dokumentech.

[7] Komerční produkt dostupný z adresy http://www.dofactory.com/Framework/Framework.aspx

Důležitou součástí je referenční implementace internetového obchodu pomocí ASP.NET 2.0 a Windows Forms 2.0. Zde jsou použity nejen GoF vzory, ale i řada vzorů z knihy *Patterns of Enterprise Application Architecture* [Fowler, 2002] a SOA vzorů.

Nakonec jsou přiloženy projekty s implementovanými vzory z populární knihy *Head First Design Patterns*.

Spíše než o nástrojovou podporu se tak jedná o studijní materiál, který ale může být díky referenční implementaci i poměrně prakticky použitelný.

6.4 Podpora vzorů ve vývojových nástrojích

Na závěr zbylo téma podpory návrhových vzorů přímo ve vývojových prostředích. Vzhledem k tomu, že ani CASE nástroje vzory příliš dobře nepodporují, dala by se ze strany vývojových prostředí zaměřených na finální implementaci očekávat pouze slabá nebo žádná podpora. Ve skutečnosti jsou na tom vývojová prostředí poměrně dobře, například IBM nabízí pro Eclipse (obecné vývojové prostředí, které se používá především pro vývoj v Javě) plugin *Design Pattern Toolkit*[8], který pomocí šablon a jejich transformací automatizuje některé vývojářské činnosti a pomáhá s implementací architektonických doporučení.

Rovněž Visual Studio, které je pro tuto práci ze všech vývojových prostředí nejzajímavější, obsahuje nástroje umožňující nebo usnadňující práci s návrhovými vzory. Zde by ale pár snímků obrazovky nestačilo, protože podpora je zaprvé daleko rozsáhlejší než např. v případě nástroje Enterprise Architect a zadruhé vychází ze zcela nové koncepce tzv. „softwarových továren“. Toto téma je tak důležité, že mu bude věnována celá příští kapitola, demonstrace podpory návrhových vzorů ve Visual Studiu bude tedy dokončena až poté.

6.5 Shrnutí

Podporu pro návrhové vzory lze dnes najít ve čtyřech základních typech produktů: v CASE nástrojích, ve specializovaných aplikacích, ve vývojových prostředích a nakonec v produktech jiného typu.

Ačkoliv současné CASE nástroje na špičkové úrovni podporují UML (dnes už běžně ve verzi 2.0), podpora pro návrhové vzory zcela chybí nebo není příliš sofistikovaná, jak demonstroval nástroj Enterprise Architect. Specializované nástroje mohou být zajímavé pro výuku, ale vzhledem k tomu, že většinou obsahují pouze malou podmnožinu funkčnosti plnohodnotných CASE nástrojů, nejsou vhodné pro reálný vývoj.

Může znít trochu překvapivě, že tam, kde selhávají CASE nástroje a specializované aplikace, by měla mít navrch běžná vývojová prostředí, ale hned následující kapitola demonstruje, že moderní IDE udělala v podpoře vzorů velký krok kupředu…

[8] http://www.alphaworks.ibm.com/tech/dptk

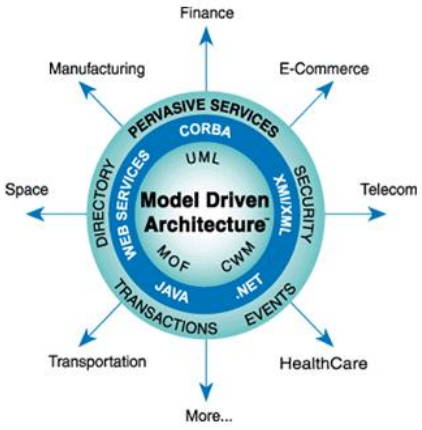
7 Softwarové továrny

Softwarové továrny jsou neobyčejně významným tématem nejen pro tuto práci, ale pro celé IT odvětví. Mají totiž ambice i potenciál změnit způsob softwarového vývoje tak, jak ho známe dnes, a který je už několik desetiletí více méně stejný. Tato kapitola prezentuje jak vizi softwarových továren, tak jejich první implementace založené na Visual Studiu 2005.

7.1 Diskuse o roli UML ve světě softwarového vývoje

Konsorcium OMG již řadu let zaštiťuje a prosazuje specifikaci *Model-Driven Architecture (MDA)*, která je založena na oddělení návrhu a konečné implementace a v jejímž středu hraje důležitou roli UML. Softwarové továrny a doménově specifické jazyky jsou do určité míry alternativou k MDA a tato kapitola se pokouší vysvětlit, proč si Microsoft vybral právě je.

Základní ideou MDA je oddělení návrhu od konkrétní implementační technologie. Jak ukazuje obrázek 28, v samém centru stojí několik technologií OMG, konkrétně UML, MOF (*Meta-Object Facility*), XMI (*XML Metadata Interchange*) a CWM (*Common Warehouse Metamodel*). Další vrstva pak naznačuje, že softwarový systém popsaný pomocí tzv. platform-independent modelu (PIM) může být implementován na mnoha různých aplikačních platformách, ať už se jedná o Javu, .NET, webové služby nebo o standard CORBA. MDA v souvislosti s konkrétní platformou mluví o tzv. platform-specific modelu, PSM.



Obrázek 28 – OMG Model-driven Architecture (zdroj: [MDA])

Architektura MDA je tak jako celek připravená na celou řadu „domén“[9], od finančních aplikací přes e-commerce systémy až třeba po zdravotní systémy. Důležité však je, že MDA primárně spoléhá na sjednocené jádro a specializace na konkrétní domény není vyjádřena nijak explicitně. Jazyk UML tak hraje v MDA naprosto zásadní roli.

Idea MDA intuitivně vypadá velmi dobře: návrh bude oddělen od cílové implementace, softwarové systémy budou vytvářeny na vyšší úrovni abstrakce (kdo by o to nestál) a jeden platform-independent model bude možné realizovat na více technologických platformách. MDA by rádo posunulo softwarový vývoj zase o trochu dál, bohužel však přehlíží některá jeho důležitá specifika.

V softwarovém vývoji se často mluví o třech úrovních abstrakce, kterými jsou architektura, design a implementace. Důležité je především zažité oddělení návrhu a implementace. Předpokládáme-li vývoj v objektově orientovaném prostředí, ve fázi designu jsou načrtnuty atributy a metody tříd, jejich asociační nebo agregační vztahy a vztahy dědičnosti, zatímco ve fázi implementace dojde k předání této *dokumentace* vývojářům a ti daný model převedou do zdrojového kódu, který lze zkompilovat a spustit.

V článcích [Reeves, 1992], [Shore, 2006], [Haack, 2006] nebo i dalších byl však prezentován do určité míry opačný názor, že *samotný zdrojový kód je návrhovým artefaktem*, a to dokonce finálním a jediným. Pěknou, i když možná trochu kontroverzní argumentaci sepsal [Shore, 2006] v článku *Quality With a Name*. Její volná citace následuje.

> Software neexistuje. Dobře, přeháním, ale ne tolik. Po spuštění aplikace dojde k jejímu nahrání z magnetických polí pevného disku do RAM paměti, procesor začne zpracovávat

[9] Přesné vymezení tohoto pojmu bude uvedeno níže.

> elektrické signály a dá instrukce grafické kartě, aby k monitoru vyslala jiné elektrické impulsy, které ve výsledku rozsvítí nebo nerozsvítí jednu z milionu barevných teček. Software nejsou jedničky a nuly, ale fascinující svět elektrických impulsů a nízkoúrovňových instrukcí. Jediný způsob, jak napsat software, je specifikovat instrukce na úrovni elektrických impulsů, což ale dnes přirozeně nikdo nedělá. Vývojář připraví specifikaci, která dohodnutým způsobem diriguje chování programu. Tuto specifikaci předá jinému programu, který se automaticky postará o vytvoření „softwaru". Touto specifikací mohou být instrukce ve strojovém kódu, v assembleru, v nějakém jazyku třetí generace nebo v čemkoliv. Podstatné je, že se pro tuto specifikaci vžil pojem *zdrojový kód*. Zdrojový kód je tedy návrhovým dokumentem.
>
> Jaký je ale potom význam UML a debat u magnetické tabule? Stejně jako u zdrojového kódu, i zde se nejedná o nic jiného než o užitečné abstrakce. Rozdíl je pouze ten, že zdrojový kód je přímo přeložitelný do elektrických impulsů, zatímco UML nikoliv … zatím.
>
> (…)
>
> Zdrojový kód je (finálním) návrhem! Prosím, pokračujte v malování UML diagramů, diskutujte nad CRC kartičkami a oblepujte si stěny diagramy tříd – abstrakce je nenahraditelná pro dobré pochopení designu. Jen nezaměňujte tyto artefakty s finálním návrhem. Pamatujte, že návrh musí fungovat, jinak je k ničemu. Pokud design nemůže být kompilátorem nebo podobným nástrojem automaticky konvertován do softwaru, design není dokončen, protože *nefunguje*.

Mnohá tvrzení jsou v uvedené citaci poměrně silná, základní pointa by však měla být zřejmá: rozdíl mezi návrhem a implementací není zdaleka tak ostrý, jak se konvenčně tvrdí. Přímo je potom navíc vyjádřena myšlenka, že UML není a nemůže být vhodné pro finální návrh, protože není automaticky převoditelné do spustitelného souboru.

Že se nejedná o extrémní pohled jednoho odborníka, dokládá citace z knihy *Software Factories* ([Greenfield, Short, 2004]):

> „Jedna z nejdůležitějších forem automatizace je generování kódu. Jak jsme uvedli dříve, část mezery [mezi fungujícím kódem a vysokou abstrakcí] může být přemostěna automatickým upřesňováním, což se děje každý den v kompilátorech.
>
> Kompilace byla jednou z nejrannějších inovací v oboru. Většina vývojářů ji považuje za tak přirozenou věc, že ani nepřemýšlejí, co všechno se při kompilaci ve skutečnosti děje, a sami sebe považují za „tvůrce softwaru", ačkoliv ve skutečnosti píší pouze specifikace, na základě kterých potom stroje vytvářejí skutečný software."

Zatímco článek Jamese Shora spíše obhajuje často podceňovanou pozici zdrojového kódu při vývoji softwarového systému, řada jiných článků už přímo napadá UML. Podle mého názoru je základní idea pěkně shrnuta v článku Philla Haacka *UML Is Chinese To Me* ([Haack, 2006]), který opět budu volně citovat.

> Softwarový vývoj je o abstrakci. Jazyk C se snažil vývojáře abstrahovat od asembleru, u Javy nebo u C# se už dokonce nemusíme starat ani o správu paměti. Při existenci mnoha jazyků však např. nemusí být vývojáři ve Visual Basicu zřejmé, co znamená zápis `class MyClass : Foo` (dvojtečka znamená v C# dědění nebo implementaci rozhraní). Není snad UML přirozeným vývojovým krokem? Bohužel nikoliv…
>
> Nejúspěšnější abstrakce jsou totiž ty, které abstrahují nějakou technickou záležitost. Např. grafická notace pro popis elektrických obvodů se těší úspěchům, kterým se UML nikdy ani nepřiblížilo. Kdy naopak abstrakce příliš nefunguje, je při aplikování na koncepty reálného světa. Vezměme si jako příklad třídu Objednávka. Strávím nad jejím vývojem nějaký čas a rozhodnu se, že na SourceForge umístím její diagram, takže kdokoliv jiný si bude moci tuto třídu stáhnout a bez námahy používat. Kdyby takto věci fungovaly, kdyby stačilo natáh-

> nout mezi dvěma třídami jednu čáru a bum, uživateli se po kliknutí na tlačítko přidá položka do nákupního košíku, bylo by všechno skvělé. Každý systém je ale (bohužel) tak jedinečný, že takto jednoduše věci prakticky nikdy nefungují.
>
> Řešení se přirozeně hledá – často se mluví o tzv. „executable UML“. To ovšem znamená přidat do jinak celkem jednoduché notace mnoho specifických symbolů, čímž se z UML prakticky stává další programovací jazyk. Svým způsobem je přechod k takto pojatému UML stejný jako přechod od angličtiny k čínštině. Tam lze také mnoho anglických písmen vyjádřit jedním znakem, což ji ale rozhodně nedělá snadněji srozumitelnou.

Citace z knihy *Software Factories* opět ukazuje, že tento názor není osamocený:

> „Přes velké množství CASE nástrojů UML příliš nepozvedlo úroveň abstrakce, na které vývojáři tvoří software. Jeho hlavní přínos je v oblasti návrhu, jelikož postrádá zaměření a sémantickou přesnost na to, aby mohlo výrazně přispět i v oblasti implementace. Většina tohoto přínosu navíc měla formu neformálních modelů načrtnutých na bílé tabuli. Největší pokrok v automatizaci vývoje byl založen na programovacích jazycích a strukturovaných značkovacích jazycích typu XML nebo HTML. Martin Fowler to komentuje výmluvně, když identifikuje tři různé cíle, ke kterým se UML hlásí: neformální design, neboli skicování, jak to nazývá, model-driven development, což nazývá „blueprintingem“, a přímá exekuce, když jsou modely interpretovány. Tvrdí, že UML uspělo v prvním cíli, když ustanovilo široce používaný vizuální slovník. Neuspělo však v dalších dvou cílech.“

O něco níže v této knize jsou uvedeny tři důvody, proč její autoři nepovažují UML za dlouhodobě perspektivní:

1. UML chybí preciznost, kterou mají standardní programovací jazyky. To nepřekvapí, protože UML bylo primárně vytvořeno pro dokumentaci a ne pro vývoj, přesto však celému IT odvětví trvalo poměrně dlouho, než si tuto skutečnost uvědomilo. Autoři za hlavní důvod považují falešný marketing, který UML doprovázel.
2. UML špatně podporuje komponentně orientovaný vývoj. Místo toho, aby ohledně komponent neposkytovalo nic (což by bylo možná lepší), nabízí UML několik vzájemně rozdílných a nekompatibilních možností. Naneštěstí navíc žádná z nich nekopíruje komponentní architektury, které se dnes běžně užívají.
3. Třetí důvod autoři vydávají za ten, který kompletně znemožňuje použít UML jako základ pro model-driven development – je jím přílišná obecnost UML kombinovaná s jeho špatnou rozšiřitelností. Doslova uvádějí „UML modeluje cokoliv a současně nic“. UML vyšlo z konceptů programovacích jazyků, které byly ve své době populární, a mechanismus rozšiřitelnosti byl pouze jednoduchý (autoři *Software Factories* ho přirovnávají k textovým makrům). Od té doby bylo pomocí těchto mechanismů rozšiřitelnosti přidáno mnoho vlastností, které se snaží přiblížit modelování reálných domén, ale vzhledem k charakteru UML byly tyto snahy ne zcela úspěšné. Novou verzi UML specifikace (2.0) navíc autoři knihy považují za ještě komplexnější a náročnější, aniž by byly vyřešeny problémy sémantické nejednoznačnosti a určité dezorganizovanosti.

Oficiální stanoviska Microsoftu jsou shrnuta v dokumentu *Visual Studio 2005 Team System Modeling Strategy and FAQ* ([Micorosft Modeling Strategy]), kde mimo jiné Microsoft svůj vztah k UML a MDA poměrně podrobně vysvětluje:[10]

Předně popírá, že by svou iniciativou doménově specifických jazyků stál proti UML. Microsoft je rád, že UML sjednotilo roztříštěný trh různých proprietárních notací. Ostatně nástroj Visio, standardně dodávaný s Visual Studiem 2003 verzí Enterprise Architect, UML notaci podporuje a svého času se jednalo o velmi populární nástroj pro kreslení UML diagramů. Anonymní prů-

[10] „Microsoft“ bude v následujících odstavcích používán kvůli jednoduchosti v personifikované podobě, ačkoliv by bylo korektnější psát „skupina v Microsoftu věnující se modelovacím technologiím“.

zkumy však ukázaly, že jen zlomek vývojářů používá jiný než diagram tříd a že pouze zlomek tohoto zlomku z diagramu generuje kód. (Na tomto místě je nutno podotknout, že přes nepochybnou popularitu Visia je jeho podpora UML na velmi základní úrovni a pro seriózní vývoj je téměř nepoužitelná. Ostatní CASE nástroje došly v podpoře round-trip engineeringu výrazně dál, ale ani dnes nefunguje vše zcela stoprocentně, a jak naznačovaly předchozí úvahy, je docela dobře možné, že ideálního stavu ani není možno dosáhnout.)

Jednoduše řečeno, masivní nástup UML v devadesátých letech nevedl k očekávanému zvýšení produktivity vývojářské práce a ke zjednodušení vývoje. Ba co víc, u mnoha architektů mohla negativní zkušenost s „model-driven development" vést k vytvoření zápornému postoji k modelování jako takovému. Microsoft proto jako svou hlavní vizi uvádí snahu změnit pohled vývojářů na hodnotu, která je v modelování ukryta. U mnoha vývojářů stojí modelování v ústraní zájmů nebo ho považují za užitečné pouze pro mítinky či dokumentaci. Microsoft chce, aby se z diagramů staly vývojové artefakty, které skutečným způsobem zefektivní vývoj a které nebudou stát „před" nebo „vedle" vývoje, ale budou jeho integrální součástí.

Proč se však Microsoft vydal vlastní cestou a nepostavil své řešení nějakým nepřímým způsobem na UML, když tato specifikace umožňuje rozšíření hned několika způsoby? Odpověď znovu pochází z dokumentu [Microsoft Modeling Strategy]:

> „Když jsme se dívali, kam směřuje specifikace UML 2.0, narazili jsme na několik důvodů, proč se stále nemůže stát základem pro nic víc než pro dokumentaci. Specifikace UML 2.0 přidala ke standardu další komplexnost s ještě víc propletenými sub-jazyky, ale stále přirozeným způsobem neřeší kritické záležitosti pro moderní vývoj, jako jsou databázový design, testování, nasazení, orientace na služby, komponentový vývoj nebo konstrukce uživatelských rozhraní. Jelikož žádný přirozený UML sub-jazyk nevyhovuje požadavkům konektivity služeb, museli bychom se uchýlit k popisu našeho Application Design DSL pomocí stereotypů a tagů na existujícím UML pod-jazyku. To by vedlo k příliš komplikovanému modelu postaveném na něčem, co je už samo o sobě mnoha lidmi z oboru považováno za příliš přeplácanou a složitou specifikaci. …"

Microsoft si proto vybral cestu softwarových továren a doménově specifických jazyků, přičemž UML doporučuje hlavně pro následující použití:

skicování
malování na tabuli
dokumentace
konceptuální schémata, která nemají přímý vztah ke kódu

Iniciativu MDA vnímá Microsoft jako trochu dvojsečnou záležitost. Na jedné straně, podobně jako u UML, tomuto konceptu vyčítá přílišnou abstraktnost a vzdálenost od reality softwarového vývoje. Pěkně postoj vystihuje věta „zatímco nezávislost na platformě může být pro některé zákazníky důležitá, my častěji slýcháme volání po vyšší produktivitě, predikovatelnosti, bezpečnosti, kontrole a efektivnějších cestách k nasazení a správě aplikací." Na druhé straně Microsoft kladně hodnotí používání modelů pro vývoj aplikací v heterogenním prostředí a důraz na dobře popsaná mapování mezi modely.

Zajímavostí je, že některé společnosti používají značku „MDA" v trochu jiném smyslu než „konkrétní standard od OMG". K proklamování „MDA" jim většinou stačí vzít nějakou technologii od OMG a postavit na ní vlastní model-driven development, aniž by byly používány PIM nebo PSM. Některé společnosti tak například za základ MDA považují specifikaci MOF, která jim umožňuje definovat vlastní jazyky tam, kde předdefinované modely UML nestačí. Tím se celkem paradoxně přístup těchto firem velmi blíží přístupu Microsoftu. Právě kvůli těmto na první pohled ne zcela zřejmým souvislostem bude Microsoft podporovat spolupráci s UML a MOF nástroji na jiných platformách.

Stručné shrnutí této kapitoly by bylo takové, že Microsoft a s ním i řada dalších odborníků nepovažují UML za prostředek, který by mohl revolučním způsobem změnit způsob softwarového vývoje. UML považují za dobrý dokumentační a skicovací nástroj, vyčítají mu však přílišnou komplexnost při současném přehlížení důležitých a v praxi často zásadních aspektů vývoje. Microsoft se proto rozhodl vydat jinou cestou…

7.2 Softwarové továrny

7.2.1 Vize

Jack Greenfield, jedna z hlavních postav iniciativy softwarových továren v Microsoftu, říká o tomto termínu následující:

> Softwarové továrny jsou o znovupoužívání vzorů, frameworků, nástrojů, šablon, požadavků, testů, instrumentací, procesních návodů a mnoha dalších aktiv používaných v životním cyklu softwarového vývoje, v systematickém spíše než v ad-hoc smyslu, a nikoliv o vytváření bezpočtu kopií stále té samé věci.
>
> (převzato z [O'Tuathail, 2006])

Trochu volněji řečeno, hlavní ideou softwarových továren je přinést do světa softwarového vývoje podobný způsob práce, jaký je obvyklý téměř ve všech ostatních odvětvích, ať už zmíníme automobilový průmysl, výrobu osobních počítačů nebo třeba stavebnictví. Všude tam dochází ke znovupoužívání již jednou vyvinutých a vyrobených součástí, které společně s jinými součástmi nakonec vytvoří funkční celek. Například v automobilovém průmyslu je zcela nemyslitelné, aby automobilka vyráběla své auto od tekutiny do ostřikovačů do posledního šroubku. Naopak, automobilky mají jedny z nejkomplexnějších dodavatelských řetězců. To má řadu výhod – firma se může věnovat výrobě aut (a ne šití sedaček), riziko i náklady jsou rozloženy po celém odvětví a nejsou akumulovány v jediné firmě, dochází k efektivní dělbě práce atd. atd.

Naproti tomu se v softwarovém odvětví setkáváme s téměř nulovým znovupoužíváním již jednou udělaného. Nejde jen o nedůvěru k externím komponentám, což řadu společností vede k vlastnímu vývoji, ale znovupoužití jakýchkoli softwarových artefaktů je obtížné i v rámci firmy. V minulosti se zrodilo několik pokusů o vyřešení tohoto problému, většinou ale bez výraznějších výsledků. Např. COM přinesl komponentové programování a umožnil tak např. používat kontrolu pravopisu ve všech aplikacích Office bez nutnosti duplicitního programování, současně však zůstal na příliš nízké úrovni abstrakce a způsob softwarového vývoje do nové dimenze příliš neposunul (nebylo to ani jeho cílem). Jako jiný příklad lze uvést UML a iniciativu MDA, která podle některých selhala zase kvůli příliš vysoké úrovni abstrakce.

Softwarové továrny chtějí dnešní situaci změnit. Nejlépe tento pojem vysvětlí uvedení analogie, kterou byl samotný termín inspirován. Autoři si budoucnost softwarového vývoje představují tak, že vývojář bude spravovat „softwarovou továrnu", která „na pás" bude dostávat standardizované softwarové artefakty (komponenty, vzory atd., viz Greenfieldova slova) a s pomocí invence vývojáře bude „vyrábět" výsledný produkt.[11] V reálu je potom onou továrnou vývojové prostředí, IDE. V konceptu softwarových továren tedy hraje podpora ze strany nástrojů zcela klíčovou roli.

Na závěr je potřeba poznamenat, že idea softwarových továren není nová. „Domain-Specific Modeling" existuje již velmi dlouho a nástroje, které s DSM pomáhají, se reálně používají již přes 10 let (jedná se např. o open source nástroj GME nebo komerční MetaEdit+). Vstup Microsoftu

[11] Zajímavostí je, že termín softwarová továrna u některých vývojářů a architektů evokuje situaci, kdy je vývojář dělníkem v továrně a veškerou práci diriguje „továrna". Ve skutečnosti, jak bylo uvedeno, je to zcela naopak. Nicméně z uvedených důvodů se lze setkat s názorem, že termín „softwarová továrna" není vhodný a že se od něj časem upustí.

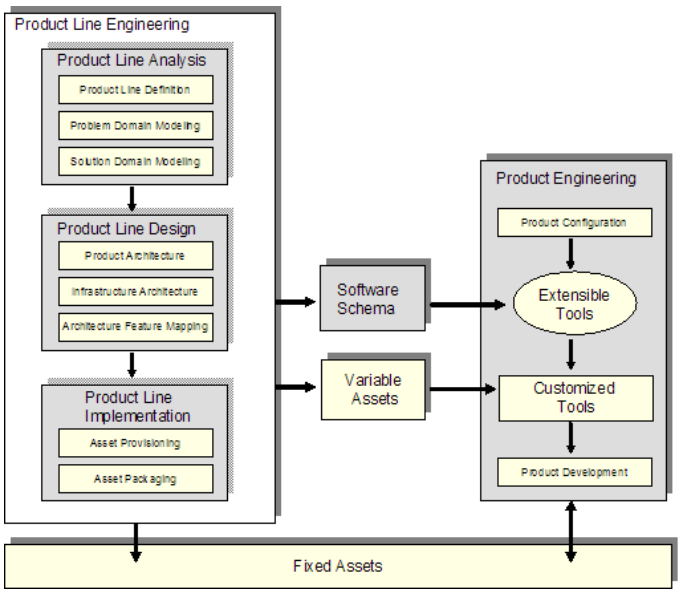
na pole modelování je však přesto významným okamžikem, protože vliv této firmy na vývojářskou komunitu je značný, a jestli má někdo „tržní sílu“ reálně prosadit novou vizi softwarového vývoje, pak je to právě Microsoft. Se svým Visual Studiem má navíc před konkurencí značnou výhodu, protože umožňuje koncept softwarových továren doručit přímo k architektům a vývojářům, aniž by bylo nutno do vývojového procesu integrovat nástroje třetích stran.

7.2.2 Konkrétní podoba softwarových továren

Předchozí řádky popisovaly softwarové továrny velmi abstraktně, spíše jako vizi nebo dlouhodobou iniciativu. Současně je ale tento koncept částečně realizován už dnes.

Obecně, ale přitom exaktně je softwarová továrna definována v knize *Software Factories* (schematicky je tato definice zachycena na obrázku 29):

> Softwarová továrna je *softwarová produkční řada* která konfiguruje rozšiřitelné nástroje, procesy a obsah pomocí *šablony softwarové továrny*, která je založena na *schématu softwarové továrny*, za účelem automatizace vývoje a údržby variant prototypového produktu pomocí adaptace, sestavování a konfigurace komponent založených na frameworku.



Obrázek 29 – Schéma softwarové továrny (zdroj: [Greenfield, 2004])

Jak tato špatně čitelná definice naznačuje, softwarové továrny jsou po teoretické stránce poměrně komplikovanou záležitostí. Kniha *Software Factories* se všem jejím aspektům věnuje na více než 600 stranách textu a pro případné zájemce se jedná o hodnotný a zajímavý zdroj informací. Na následujících stranách bude nabídnut poněkud praktičtější pohled.

Softwarová továrna se obecně skládá ze

1. Softwarových aktiv
2. Nástrojů

Do softwarových aktiv se řadí znovupoužitelné kusy kódu, dokumentace, referenční implementace a podobně. Softwarové nástroje potom zahrnují různé průvodce, generátory kódu a vizuální designéry.

Softwarová továrna obvykle obsahuje šablonu, která pomáhá rychle začít s vývojem určitého typu aplikace (příkladem je předpřipravené solution ve Visual Studiu), ale i v dalších fázích vývojového cyklu obvykle poskytuje nástroje, které vývojářům pomáhají a všelijak usnadňují práci.

Typickým znakem softwarových továren je jejich přizpůsobitelnost konkrétním podmínkám a konkrétnímu vývojovému týmu. Tuto práci obstarává softwarový architekt.

Softwarové továrny mají ve srovnání s obvyklým vývojovým procesem tyto výhody:

Konzistence. Softwarové továrny pracují s pojmem *produktová řada* (*software product line*), což je sada aplikací, které sdílejí vlastnosti a architekturu.[12] Jednotlivé instance produktových řad jsou tím pádem konzistentní, což přináší úspory pro vývoj, údržbu, školení a podobně.

Kvalita. Předem stanovená architektura a její snadné používání ve vývojovém procesu má za následek vyšší kvalitu výsledného kódu.

Produktivita. Vychází ze znovupoužívání softwarových aktiv (aplikačních bloků, frameworků, …), poskytování automatizovaných návodů a generování kódu z modelů, které poskytují abstraktnější pohled na vytvářený softwarový systém. Softwarové továrny usnadňují nebo zcela eliminují řadu rutinních činností a vývojáři se tak mohou soustředit na ty aspekty vývoje, které přinášejí skutečnou přidanou hodnotu.

Konkrétní obsah softwarových továren není normativně určen a může se v různých implementacích lišit. Softwarová továrna postavená na Visual Studio 2005 bude obvykle obsahovat následující komponenty (převzato z dokumentace k [SCSF]):

Tovární schéma ilustruje aktivity a aktiva použitelná pro tuto továrnu.

Referenční implementace ukazuje příklad hotové aplikace, kterou lze s pomocí továrny vybudovat.

Architektonická doporučení a vzory vysvětlují některá rozhodnutí, která byla v rámci softwarové továrny učiněna.

Postupy „jak něco udělat“ obsahují procedury pro splnění určitých úkolů.

Recepty automatizují procedury v postupech „jak něco udělat“ (buďto celé nebo jen určité kroky). Tím se vývojáři pomáhá od rutinních činností.

Šablony jsou předpřipravené elementy aplikace s oblastmi, které mají být vyplněny vývojářem podle aktuálních požadavků. Mohou existovat např. šablony pro výchozí solution Visual Studia nebo třeba pro jednotlivé prvky projektu.

Návrháře neboli designéry umožňují vizuální pohled na některé aspekty vyvíjené aplikace. Příkladem takového návrháře je třeba GUI designér nebo diagram tříd.

Znovupoužitelný kód je reprezentován komponentami, které implementují nějakou běžnou funkcionalitu.

Konkrétní příklady reálných softwarových továren budou uvedeny v kapitole 7.5 po vysvětlení dalších souvisejících témat.

7.3 Doménově specifické jazyky (DSLs)

Podobně jako v centru iniciativy MDA stojí jazyk UML, softwarové továrny pracují s takzvanými doménově specifickými jazyky (*domain-specific languages*, *DSLs*), respektive s jejich podmnožinou.

Obecně se doménově specifickým jazykem myslí notace, ať už textová nebo vizuální, která popisuje určitou oblast reálného světa. Mezi příklady může patřit např. notace pro popis elektrických obvodů, plán rozmístění nábytku v místnosti, notová osnova nebo třeba obyčejná mapa. Ačkoliv

[12] Více viz např. [Greenfield, Short, 2004] nebo [Software Product Lines]

mezi uvedenými příklady neexistuje prakticky žádná vizuální podobnost, mají společné to, že přehledným, srozumitelným a vysoce efektivním způsobem popisují danou problémovou oblast. Doménově specifické jazyky se už desetiletí vyskytují také v oblasti softwarového inženýrství, kde je ale jejich vymezení a použití poněkud složitější.

Historicky nacházely doménově specifické jazyky své hlavní využití jako pomocníci v úzce specializovaných oblastech, například regulární výrazy pomáhaly se zpracováním textových řetězců, syntaxe UNIXových shellů usnadňovala práci systémovým administrátorům a podobně. V té době byl doménově specifický jazyk jakýmsi makrem, skriptovacím jazykem nebo „malým programovacím jazykem“. V současnosti začíná koncept doménově specifických jazyků pronikat také do oblasti softwarového inženýrství, což je pro tuto práci nejzajímavější a nejdůležitější. Pokud tedy budou ve zbytku práce zmiňovány doménově specifické jazyky, obvykle se jimi budou rozumět DSL zaměřené na oblast softwarového inženýrství.

Přesné vymezení pojmu „doména“ ve smyslu DSL zaměřeného na softwarové inženýrství je poměrně obtížné a v dostupné literatuře se většinou pouze naznačuje. V knize *Software Factories* je doména definována jako „oblast zájmu nebo rodina souvisejících objektů“[13]. Jako příklad je uvedena e-commerce doména obsahující procesy pro online prodej a entity účastnící se těchto procesů, např. Objednávka, Produkt nebo Zákazník. Reálný svět definuje tzv. *problémovou doménu* (*problem domain*), zatímco na straně softwaru se mluví o *doméně řešení* (*solution domain*).

Existuje však i obecnější výklad tohoto pojmu. Například SQL je ukázkou doménově specifického jazyka, který lze uplatnit při budování prakticky libovolného softwarového systému využívajícího databázi. Doménou se v tomto případě myslí spíše jeden určitý pohled na dané řešení, jeden z jeho aspektů. V oblasti modelování může být příkladem diagram tříd (ať už ten z UML nebo nějaký upravený), což je doménově specifický jazyk použitelný v prakticky libovolné objektově orientované aplikaci. Z praktického hlediska se zdá být tento výklad užitečnější, protože široce použitelné doménově specifické jazyky pro oblast financí, zdravotnictví nebo e-commerce lze dnes najít jen stěží, zatímco doménově specifických jazyků zaměřených na určitý aspekt vývoje existuje celá řada, ať už se jedná o dotazovací jazyky (SQL, XQuery, LINQ, …), o jazyky pro vizuální návrh tříd nebo o další.

Pokud bychom výklad z minulého odstavce dovedli do důsledků, bylo by možné označit UML za DSL orientující se na doménu objektově orientovaného návrhu a např. C# za jazyk orientující se na doménu softwarového vývoje. Tato práce však bude respektovat převládající názor, že UML není dostatečně „specifické“ a bude jej proto řadit do kategorie obecně orientovaných (general purpose) jazyků. Tento odstavec dává nicméně tušit, že UML a DSL k sobě nemají zase tak daleko, jak by se mohlo na první pohled zdát, a že do značné míry záleží na nastavení „hranice obecnosti/specifičnosti“.

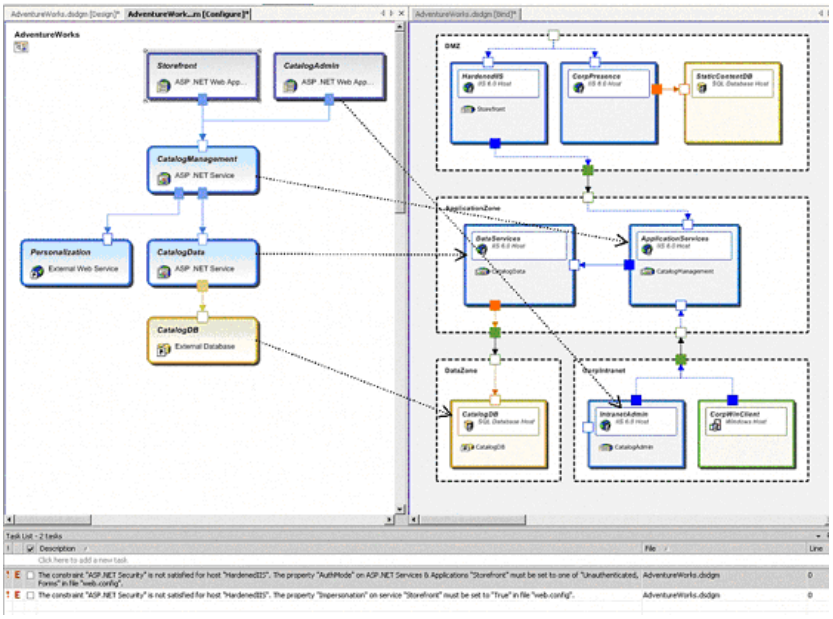
Hlavní znaky doménově specifických jazyků tak, jak jsou pojaty v této práci, jsou tedy následující:

> Bereme v úvahu pouze podmnožinu všech doménově specifických jazyků, a to podmnožinu zaměřenou na softwarové inženýrství („doménou“ není softwarové inženýrství samo, ale vždy nějaká jeho poddoména).
>
> Doménou se rozumí buďto určitá oblast reálného světa (zdravotnictví, e-commerce, …) nebo určitý aspekt vyvíjeného systému (přístup k datům, vztah tříd, …). Softwarová továrna chápající doménu v prvním smyslu se někdy nazývá jako *vertikální*, v druhém smyslu jako *horizontální*. Existuje více horizontálních doménově specifických jazyků než vertikálních.
>
> Jazyk je typicky vizuální namísto textového. DSL slouží k modelování.

Mezi hlavní cíle doménově specifických jazyků patří snaha poskytovat přehledný vizuální jazyk pro danou doménu, automatizovat některé činnosti softwarového návrhu a vývoje (např. gene-

[13] Kapitola 3, *Dealing with change*, sekce *Software as Simulation*.

rováním kódu) a někdy také vizualizovat existující systémy. Příklad doménově specifického jazyka ve Visual Studiu ukazuje následující obrázek:



Obrázek 30 – Příklad doménově specifického jazyka ve Visual Studiu (zdroj: [Regio, Greenfield, 2005])

7.4 Podpora softwarových továren ve Visual Studiu 2005

Od obecného popisu principů softwarových továren a doménově specifických jazyků nyní přecházíme k jejich realizaci v prostředí Visual Studia 2005. Toto vývojové prostředí není jediné, které softwarové továrny podporuje (už dříve byl zmíněn např. Design Pattern Toolkit pro IDE Eclipse), Visual Studio je však pro platformu .NET prostředím nejrozšířenějším a nejdůležitějším.

Podpora softwarových továren ve Visual Studiu 2005 je realizována pomocí dvou add-onů, kterým se věnují následující kapitoly.

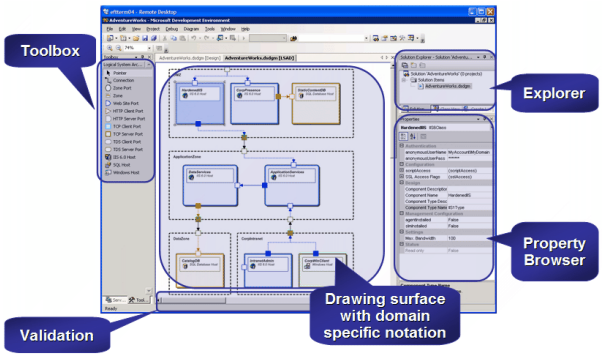
7.4.1 DSL Tools

DSL Tools (Domain-Specific Language Tools) slouží k vytváření vlastních vizuálních jazyků, ze kterých je potom možné generovat zdrojový kód nebo další artefakty softwarového vývoje. Pro vytváření nových jazyků a návrhářů je potřeba Visual Studio 2005 Professional nebo některá z týmových edic Visual Studio Team System, pro jejich používání stačí Visual Studio 2005 Standard Edition.

DSL Tools jsou nedílnou součástí vývojového balíčku Visual Studio 2005 SDK[14], který se neustále vyvíjí a je aktualizován zhruba každý měsíc. V současné době procházejí podstatnými změnami právě nástroje pro práci s doménově specifickými jazyky, když se např. mezi dubnovou a červnovou verzí SDK zcela změnil formát souborů, do kterých se definice jazyků ukládá (starší formáty .dsldd a .dsldmd byly nahrazeny novějším a sjednoceným .dsl). Všechny zde uvedené formace se vztahují k verzi August 2006 CTP (Community Technical Preview), která by již měla být svou podobou blízká finální verzi plánované na září 2006.

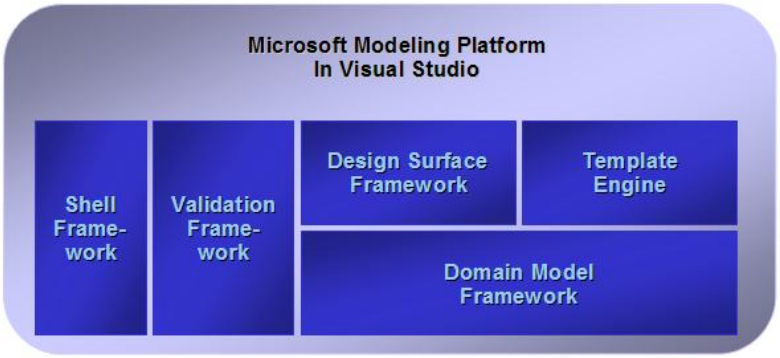
[14] VS SDK je k dispozici po registraci na adrese http://affiliate.vsipmembers.com/. Domovskou stránkou rozšiřitelnosti Visual Studia je http://msdn.microsoft.com/vstudio/extend/.

DSL Tools vycházejí z myšlenky, že prakticky všechny doménově specifické jazyky potřebují od Visual Studia stejné služby – samotný jazyk má být možné editovat ve vizuálním návrháři, komponenty se mají zobrazovat v Toolboxu, projekt má být možné procházet v Solution Exploreru a k editování vlastnosti jednotlivých prvků diagramu má sloužit mřížka vlastností. Ať už bude doménově specifický jazyk modelovat databázi, uživatelské rozhraní nebo třeba rodinný rodokmen, vždy budou potřeba všechny vyjmenované záležitosti.



Obrázek 31 – Společné prvky všech doménově specifických jazyků (zdroj: [Seemann, 2005])

DSL Tools proto přicházejí s platformou, která jednoduchým způsobem tyto služby zpřístupňuje. Její komponenty jsou následující:



Obrázek 32 – Microsoft Modeling Platform (zdroj: [Seemann, 2005])

Domain Model Framework je ústřední komponentou pro podporu vizuálních návrhářů. Při vývoji vlastního vizuálního prostředí pro doménově specifický jazyk by bylo potřeba udržovat v paměti řadu informací, např. jaké tvary jsou aktuálně v diagramu, jaká je jejich barva, pozice či velikost, ale také doménově specifické informace. Domain Model Framework všechny tyto problémy řeší pomocí in-memory databáze diagramu, ke které navíc dodává řadu důležitých služeb, například:

- podporu transakcí, takže Undo je možné implementovat jako rollback; transakce mohou být vnořené
- silně typové API pro přístup k objektům diagramu
- podpora serializace do XML, kterou je možno upravit nebo kompletně nahradit vlastní implementací
- přístup k objektům Domain Model Frameworku pomocí XPath dialektu

Design Surface Framework poskytuje prostředí pro samotnou práci s diagramy. Automaticky podporuje činnosti, které uživatel od vizuálního návrháře očekává, jako např. zoomování, scrollování, výběr více prvků diagramu a podobně. Mezi vlastnosti, které by bylo v případě vlastního vývoje poněkud obtížnější naprogramovat, patří podpora automatického propojování tvarů či jejich rozmístění do přehledného layoutu.

Template Engine slouží ke generování textových souborů, např. C# zdrojových kódů nebo XML konfiguračních souborů. Šablony jsou psány v jazyce podobném ASP.NET a mohou tedy obsahovat jak statický text, tak dynamicky generované části. Šablonovací

systém lze volat z Visual Studia nebo z příkazové řádky a DSL Tools navíc obsahují mechanismus, který umožňuje transformaci šablon z vlastního prostředí.

Shell Framework integruje vytvářený balíček do Visual Studia a lze pomocí něj ovlivnit např. položky v hlavním nebo v kontextovém menu, komponenty dostupné v Toolboxu a podobně.

Validation Framework hlídá sémantickou správnost vytvářeného modelu (tím se vizuální návrháře pro doménově specifické jazyky liší od běžných kreslicích nástrojů). Autor návrháře může např. zajistit, že ve vytvářeném rodokmenu nikdy nebude rok narození potomka nižší než rok narození jeho rodiče a podobně.

DSL Tools dále obsahují:

Průvodce pro jednoduché vytvoření nového projektu

Šablony některých často požadovaných DSL, jako např. class diagramu, stavového diagramu a některých dalších (tyto šablony mohou v budoucnu přibývat)

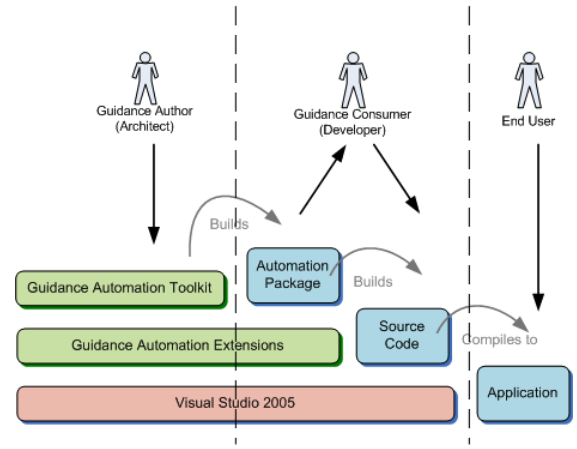
Dokumentaci

Příklady (jeden z nich je poměrně komplexní a na reálném příkladu popisuje vytvoření DSL pro návrh průvodců, bohužel v použité CTP verzi nefungoval)

7.4.2 Guidance Automation Extensions (GAX) a Guidance Automation Toolkit (GAT)

Vedle DSL Tools přichází Microsoft také s trochu odlišnou iniciativou, která má pomoci vývojářům s aplikací pokynů a pravidel, které přicházejí od softwarových architektů.

Guidance Automation Extensions (GAX) jsou běhovým prostředím pro balíčky návodů (*guidance packages*)[15], zatímco Guidance Automation Toolkit (GAT) je nástrojem určeným pro softwarové architekty, kteří pomocí něj tyto balíčky vytvářejí. Vztah mezi těmito komponentami demonstruje obrázek 33: na počítači softwarového architekta musí být nainstalován GAX i GAT, na počítači vývojáře stačí pouze GAX a odpovídající balíček návodů, zatímco ke koncovému uživateli se nakonec dostane pouze hotová aplikace.



Obrázek 33 – Vztah GAX, GAT a Visual Studia (zdroj: dokumentace k [SCSF])

Balíčky vytvořené pomocí GAT se skládají z následujících komponent ([Introduction to GAT]):

Recepty (*recipes*) automatizují aktivity, které by vývojáři obvykle prováděli ručně podle nějakých pokynů. Tím se šetří práce, která navíc může plodit „lidské chyby“. Určitou činnost lze aplikovat na jeden nebo na řadu elementů jednoho solution, které lze navíc vybrat podle určitého kriteria (úkol lze např. aplikovat na všechny projekty napsané v jazyku C#).

Akce (*actions*) jsou atomickými jednotkami práce. Recept se skládá z posloupnosti akcí.

[15] Pojem *guidance* se obtížně překládá. Jedná se o pokyny, návody, doporučení nebo směrnice, kterými softwaroví architekti specifikují práci svým vývojářům. *Guidance package* bude překládán jako *balíček návodů*, což sice není úplně přesné, ale poměrně přijatelné. Problematické pojmy jsou rovněž uvedeny a popsány v terminologickém slovníku na konci práce.

Šablony pro transformaci textových šablon (*text template transformation templates*), známé také jako **T4**, jsou šablony, které jsou tvořeny textem a skriptlety (skriptlet je výraz v jazyce C# nebo Visual Basicu, který je při zpracování šablony nahrazen svým výsledkem).

Průvodci (*wizards*) od vývojáře sbírají informace, které ke svému běhu potřebuje recept.

Typové konvertory (*type converters*) po validaci převádějí pole z UI reprezentace do plně typové reprezentace.

Šablony pro Visual Studio jsou psány v XML a mohou vytvářet nová solutions nebo přidávat projekty do těch stávajících.

7.5 Příklady softwarových továren

V době dokončování této práce (v červenci a srpnu 2006) se začaly objevovat první reálně použitelné softwarové továrny. Tato kapitola stručně popisuje jednu z nich a upozorňuje i na některé zajímavé projekty, na kterých se v současnosti pracuje a které by měly být v brzké době dostupné.

7.5.1 HL7 Software Factory

Na prvním místě je potřeba zmínit HL7 (Health Level 7) Software Factory, která se objevila již před vydáním Visual Studia 2005 a měla demonstrovat vizi softwarových továren na reálném příkladu. Jednalo se o úplně první softwarovou továrnu realizovanou Microsoftem v prostředí beta verze Visual Studia 2005.

Health Level 7 je standardizační organizace v Severní Americe zaměřující se na oblast zdravotnictví, konkrétně na klinická a administrativní data. Hlavní vizí HL7 Software Factory bylo „zjednodušit návrh a implementaci aplikací vyhovujících standardům HL7 v prostředí webových služeb pomocí nástrojů Microsoftu“.

Podrobný popis projektu je uveden v článku [Regio, Greenfield, 2005], tato továrna je však zajímavá především dvěma skutečnostmi:

1. Na rozdíl od ostatních továren demonstrovaných v této kapitole se jedná o *vertikální* softwarovou továrnu, tedy takovou, která popisuje určitou oblast reálného světa „od A do Z“. Z toho vyplývá její nutně komplexní charakter.
2. Někteří odborníci (i z Microsoftu) ji považují za neúspěch, viz např. [Osherove, 2006] nebo [Brabant, 2006] (David Brabant se navíc specializuje na oblast zdravotnictví). Důvodem je skutečnost, že i po více než roce intenzivní práce není softwarová továrna reálně použitelná.

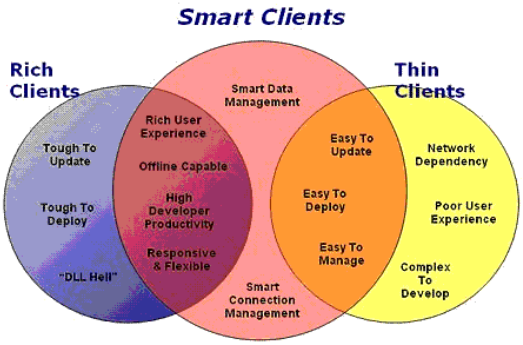
Podrobnější rozbor toho, zda je HL7 Software Factory opravdovým neúspěchem a jestli není celá vize softwarových továren nereálná, je uveden v samostatné kapitole 7.6, panuje ovšem všeobecná názorová shoda, že Microsoft začal s příliš složitým příkladem. Zbytek kapitoly proto prezentuje jednodušší softwarové továrny, které se v současnosti začínají objevovat a jejichž použitelnost je na dobré úrovni.

7.5.2 Smart Client Software Factory (SCSF)

7.5.2.1 Úvod

Jako hlavní příklad byla vybrána Smart Client Software Factory ([SCSF]), která slouží k vývoji tzv. chytrých klientů. Chytrý klient zde neznamená zařízení typu PDA nebo Smartphone, ale běžnou desktopovou aplikaci, která se snaží využít některých výhod webových aplikací, jako např.

snadného nasazení, správy verzí, podpory spolupráce s webovými službami a podobně – koncept je naznačen na obrázku 34.



Obrázek 34 – Koncept chytrých klientů (zdroj: [Building Smart Clients])

Celá továrna je postavena na *Composite UI Application Block*u (CAB[16]), což je aplikační blok vyvinutý skupinou *patterns & practices*[17]. Ta se v rámci Microsoftu věnuje návrhovým vzorům, doporučením pro vývoj aplikací, nejlepším praktikám a podobně. Všechna doporučení jsou vydávána buďto formou dokumentů nebo tzv. aplikačních bloků, které vždy nějakým způsobem řeší určitou problémovou oblast, jako třeba logování nebo kešování, a kromě pokynů poskytují i zdrojové kódy s výchozí implementací, kterou je možno v případě potřeby upravit. Nejznámějším počinem skupiny patterns & practices je Enterprise Library, která se skládá z obecně použitelných aplikačních bloků pro kešování, kryptografii, přístup k datům, ošetřování výjimek, logování a bezpečnost. Kromě těchto aplikačních bloků nabízí skupina p&p i řadu dalších, mezi nimiž je podle počtu stažení nejoblíbenější právě Composite UI Application Block.

7.5.2.2 Stručný přehled Composite UI Application Blocku (CAB)

CAB pomáhá vývojářům vytvářet aplikace, které mají mít následující charakteristiky:

Aplikace má být tvořena z nezávislých, ale kooperujících modulů.

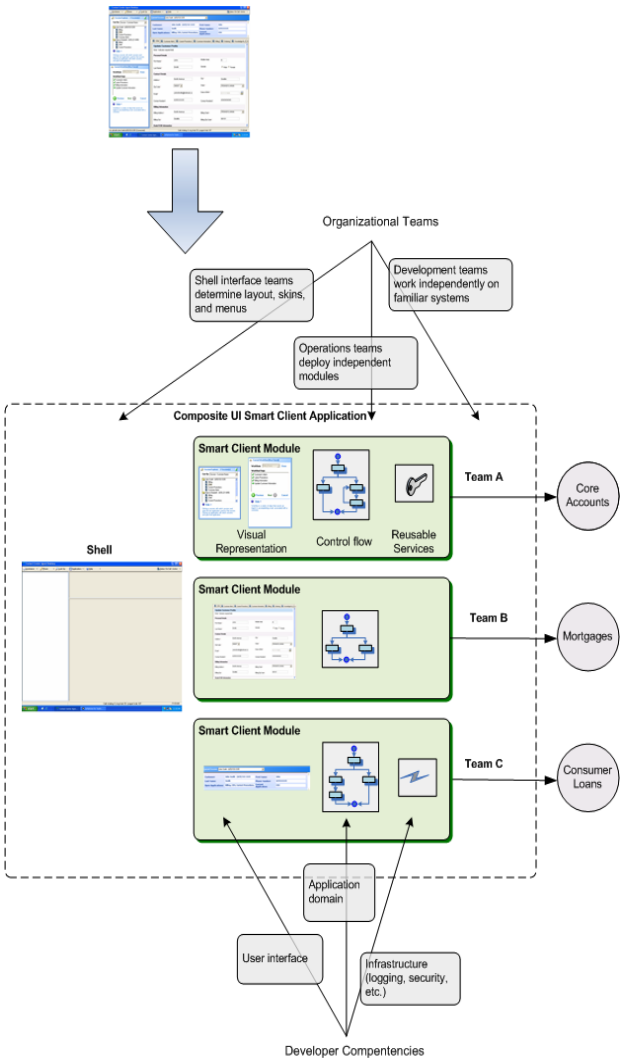
Při vývoji je potřeba oddělit vývojáře modulů od vývojářů tzv. shellu. Shell je základním rozhraním pro interakci s uživatelem a dobrým příkladem je třeba shell Visual Studia, který může hostit editor kódu, debugovací nástroje nebo i zcela jiný produkt, např. BizTalk Server.

Architekti by rádi použili framework pro konzistentní vývojový proces (na rozdíl od vlastního vývoje „od nuly“).

Ne každá aplikace tedy bude CAB potřebovat, ale od určité hranice složitosti se jeví jako vhodný a dobře použitelný prostředek.

Základní pointou CAB je oddělení shellu od jednotlivých modulů. Moduly lze vyvíjet několika týmy současně a k jejich integraci a kooperaci dojde až při společném běhu v jednom shellu. Tento koncept ukazuje obrázek 35.

[16] CAB koliduje se zkratkou pro Caching Application Block, ale jedná se o vžitou a v dokumentaci používanou zkratku také pro Composite UI Application Block. Vzhledem k tomu, že v tomto textu nemůže dojít k omylu, bude zkratka CAB používána výhradně pro Composite UI Application Block.
[17] http://msdn.microsoft.com/practices/



Obrázek 35 – Schematické znázornění CAB (zdroj: dokumentace k [SCSF])

Aplikace se skládá (může skládat) z následujících komponent:

SmartPart – vizuální element aplikace neboli *pohled*

Workspace – kontejner pro SmartPart, kontroluje jeho viditelnost, layout a podobně. Existuje několik druhů těchto kontejnerů, např. MdiWorkspace, TabWorkspace a podobně – názvy jsou dostatečně výmluvné.

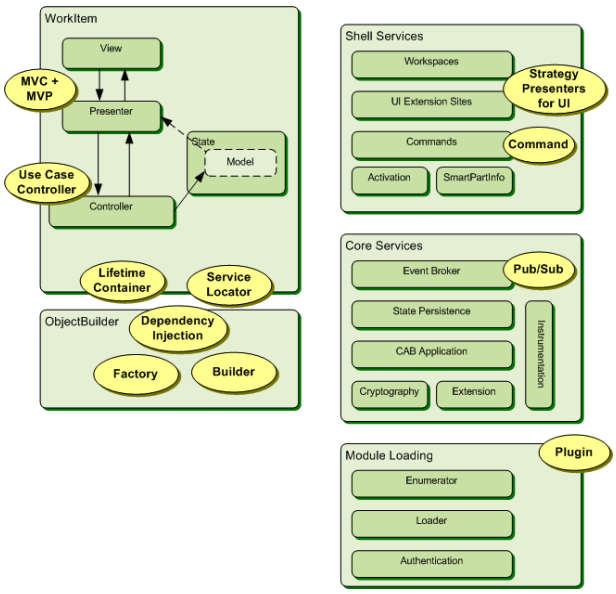
WorkItem – sdružuje komponenty, které mohou kolaborovat, aby obsloužily nějaký případ užití (use case).

Služby – k dispozici jsou základní infrastrukturní služby pro autentizaci, nahrávání modulů a podobně, ale ale vytvářet i služby vlastní, dodávané modulem.

Modul – sdružuje služby, WorkItems, SmartParts, kontrolery, business entity a inicializační třídu. V podstatě se jedná o jednotku nějaké komplexnější funkčnosti pro aplikaci, modul si lze představit jako plugin.

Stav – každá WorkItem má svůj stav.

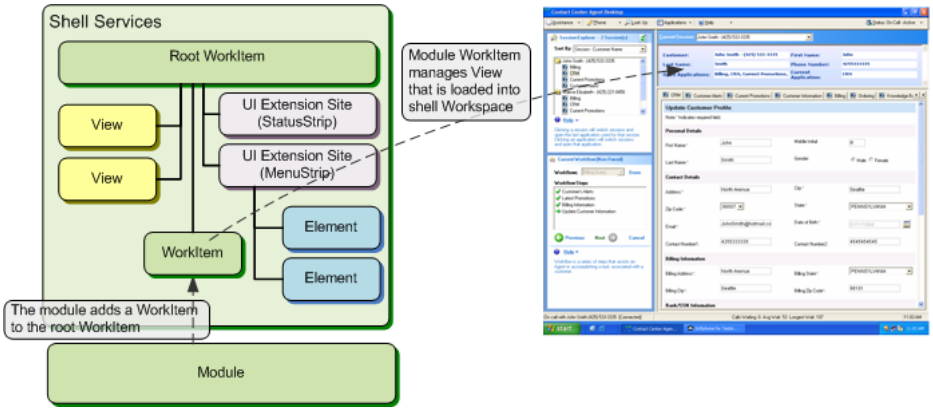
existuje ještě několik dalších komponent, viz [CAB HoL]



Obrázek 36 – Využití návrhových vzorů v CAB (zdroj: dokumentace k [CAB])

Celá architektura využívá řadu návrhových vzorů, např. MVC (alternativně MVP, Model-View-Presenter), Dependency Injection, Command, Plugin atd. Jednotlivé komponenty jsou díky použitým vzorům volně svázány a třeba implementace mechanismu Dependency Injection je tak rozsáhlá, že si vyžádala samostatný projekt nazvaný *Object Builder* (podrobnosti viz [Object Builder]). Důsledkem volného svázání je skutečnost, že lze libovolnou funkčnost v případě, že nevyhovuje, nahradit vlastní implementací. Více viz např. [CAB HoL], kapitola *06 – Extending CAB*.

Struktura aplikace vytvořené pomocí CAB je schematicky znázorněna na obrázku 37:



Obrázek 37 – Příklad aplikace vytvořené pomocí CAB

Nalevo je vidět struktura aplikace – shell obsahuje kořenovou WorkItem a nabízí modulům takzvané *UI extension sites* (místa rozšiřitelnosti), tedy elementy grafického uživatelského rozhraní, které mohou moduly změnit (např. přidat položku do menu nebo změnit text stavového řádku). Modul může přidat vlastní WorkItem a poskytnout pohled (SmartPart), který se v uživatelském rozhraní projeví nejčastěji jako určitý panel. Na snímku obrazovky vpravo je vidět příklad uživatelského rozhraní, které se skládá z několika oblastí dodaných jednotlivými moduly. Rozhraní jako celek je synchronizováno, takže pokud jsou např. editovány údaje o určitém zákazníkovi,

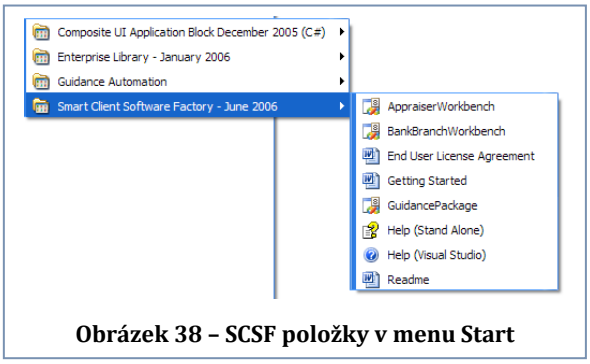
všechny pohledy se synchronně věnují právě tomuto zákazníkovi.[18] Tím je naplněna základní idea, že moduly jsou oddělené, ale spolupracující.

Další popis by již zabíhal do technických detailů, které nejsou pro pochopení fungování Smart Client Software Factory nezbytné. Podrobnější popis CAB je k dispozici přímo v dokumentaci a pro „rychlé“ seznámení jsou k dispozici Hand-on Labs ([CAB HoL]).

7.5.2.3 Obsah SCSF

Po nainstalování vytvoří SCSF v menu Start skupinu odkazů, z nichž většina vede na soubory s dokumentací (navzdory tomu, že pojem „továrna“ může vyvolávat dojem něčeho velkého a komplikovaného). Vedle dokumentace jsou připraveny odkazy na dvě referenční implementace – AppraiserWorkbench a BankBranchWorkbench.

Dokumentace je důležitou součástí SCSF. Obsahuje nejen popis různých součástí této softwarové továrny, ale i konkrétní a přímo použitelné „how-to“ témata. Dokumentace je členěna následovně:



Obrázek 38 – SCSF položky v menu Start

Úvod do SCSF
- Jak rychle začít
- Instalace
- Copyright apod.

Vysvětlení základních konceptů SCSF
- Co jsou softwarové továrny
- Koncept chytrých klientů
- Popis Composite UI Application Blocku. Je o něco podrobnější než popis uvedený v předchozí podkapitole, pro plné pochopení všech detailů však nepostačuje a vývojář by se tak měl nejprve s tímto blokem důkladně seznámit.
- Vysvětlení principu automatizovaných návodů

Vývoj s pomocí SCSF
- Vysvětlení vzorů použitých v architektuře CAB. Jako příklady jsou uváděny odkazy do referenčních implementací.
- „How-to“ témata – jednotlivé vývojářské úlohy a jejich řešení popsaná krok za krokem, vždy i s uvedením, která část je podporována automatizovaně a které kroky je nutno provést manuálně
- Další záležitosti, které si vyžadují pozornost

Přizpůsobování SCSF
- Přizpůsobení automatizovaných návodů
- Přizpůsobení dokumentace

Popis referenčních implementací

Kromě dokumentace nainstaluje SCSF do Visual Studia následující balíčky návodů (guidance packages):

Smart Client Application – šablona, podle níž se vytvoří výchozí solution

Add Business Module – šablona projektu, která vytvoří výchozí bod pro vývoj CAB modulu obsahujícího WorkItem

Add Foundation Module – šablona projektu, která vytvoří výchozí bod pro vývoj CAB modulu poskytujícího služby dalším modulům

[18] Synchronizace různých pohledů je v SCSF řešena pomocí návrhového vzoru View Navigation, viz dokumentace k SCSF, oddíl *Developing Smart Client Applications*, kapitola *Useful Patterns for Smart Client Development* (ms-help://ms.scsf.2006jun/SCSF/html/03-070-View%20Navigation.htm).

Add View (with presenter) – recept, který vytvoří pohled a odpovídající objektu Presenter z návrhového vzoru MVP

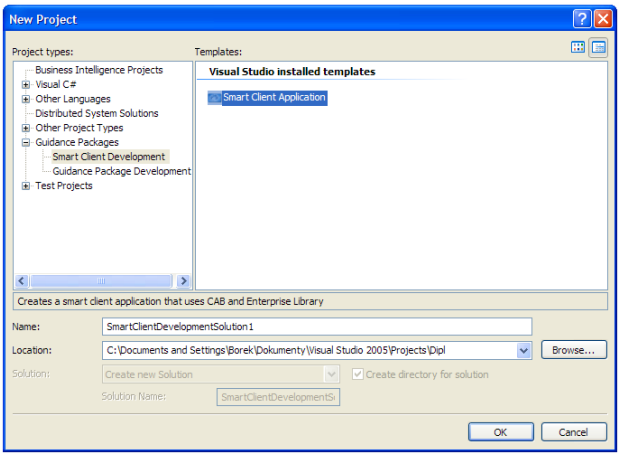
Add Smart Web Reference – recept, který vytvoří třídy a příkazy použitelné pro pro volání webových služeb podle vzoru Asynchronous Web Service Invocation with Timeout (tento vzor je popsán v dokumentaci SCSF)

Add Event Publication – recept, který do existující třídy přidá kód pro deklaraci a publikování události

Add Event Subscription – recept analogický tomu předchozímu, do existující třídy přidá metodu, která se přihlásí k odběru dané události

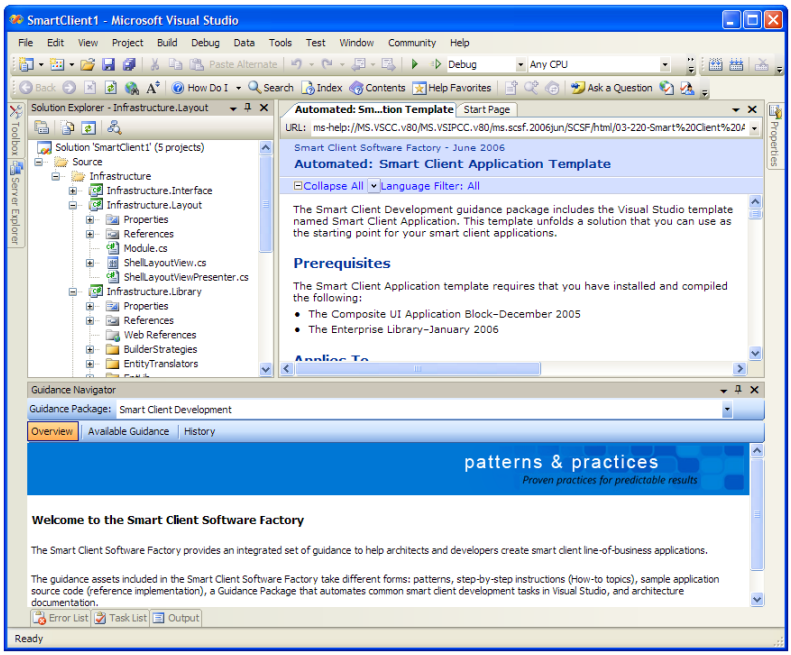
7.5.2.4 Práce se SCSF

Po instalaci SCSF se v dialogu pro vytvoření nového projektu zobrazí nová položka *Guidance Packages / Smart Client Development / Smart Client Application*.



Obrázek 39 – Nová položka v dialogu pro přidání projektu

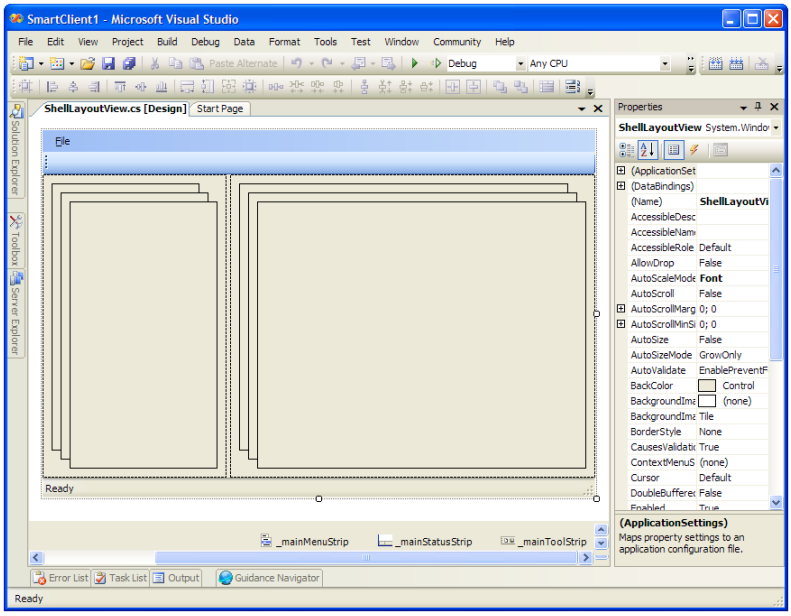
Po nastavení několika hodnot ve průvodci se otevře Visual Studia 2005 v následující konfiguraci:



Obrázek 40 – Softwarová továrna pro vývoj klientských aplikací

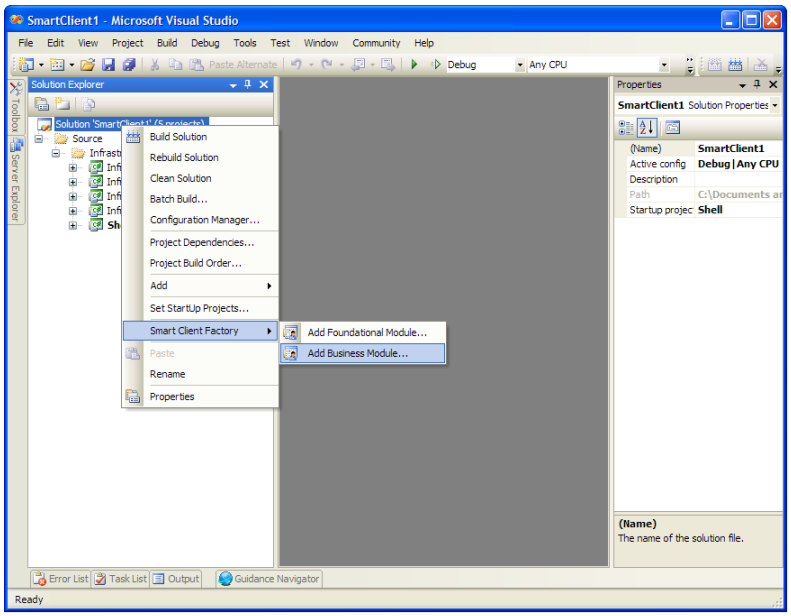
Takto vypadá softwarová továrna, respektive její část realizovaná jako vývojové prostředí. V solution exploreru (vlevo nahoře) je předpřipraveno několik projektů – tři infrastrukturní a jeden pro vývoj shellu. V pravé části je ve výchozím stavu otevřena stránka s úvodními informacemi (jedná se o jedno z „how-to“ témat) a ve spodní části je zobrazeno okno guidance navigátoru, který je standardní součástí GAX a zobrazuje dostupnou automatizovanou podporu.

Stále se přirozeně jedná o plné Visual Studio, takže po otevření souboru ShellLayoutView.cs lze editovat uživatelské rozhraní zcela stejně, jako jakýkoliv jiný Windows Forms 2.0 formulář:



Obrázek 41 – V „softwarové továrně“ zůstávají k dispozici všechny vlastnosti Visual Studia

V čem se naopak prostředí SCSF od běžného Visual Studia liší, je komfort, s jakým lze provádět určité akce. Následující obrázek ukazuje základní interakci softwarové továrny s vývojářem:



Obrázek 42 – Interakce softwarové továrny s vývojářem

Většina automatizačních pomocníků je ukryta v kontextových nabídkách solution exploreru, případně lze použít okno Guidance Navigatoru zmíněné před chvílí. Kontextové nabídky jsou skutečně kontextové a tak např. po kliknutí na celé solution je nabídnuto přidání infrastrukturního nebo business modulu (viz obrázek 42), zatímco po kliknutí na konkrétní projekt je nabídnuto přidání pohledu a „chytré webové reference".

Na příkladu receptu Add View (with presenter) je vidět, jak softwarová továrna pomáhá s návrhovými vzory. Vykonání tohoto receptu vytvoří Windows Forms formulář, který představuje prvek View návrhového vzoru Model-View-Presenter. Rovněž vytvoří prvek Presenter a odkaz na něj přidá do prvku View. Recept tak sám zařídí, aby nově přidané soubory odpovídaly návrhovému vzoru MVP.

7.5.3 Další softwarové továrny od skupiny patterns & practices

Smart Client Software Factory je jen jednou z několika softwarových továren, které skupina patterns & practices v současnosti (v létě 2006) postupně uvolňuje. K dispozici jsou nebo budou také následující továrny:

Web Service Software Factory[19] byla vydána v červenci 2006 a podporuje vytváření webových služeb na platformě ASP.NET nebo Windows Communication Foundation. Tato továrna je oproti SCSF komplexnější a názorně je demonstrována ve webcastu Dona Smithe *Introducing the Web Service Software Factory* (viz [WSSF webcast]).

Mobile Client Software Factory[20] byla rovněž uvolněna v červenci 2006 a jedná se o určitou obdobu Smart Client Software Factory s tím rozdílem, že se zaměřuje na mobilní klienty typu PDA. Rovněž stojí na Composite UI Application Blocku, v této softwarové továrně však CAB musel být přizpůsoben .NET Compact Frameworku.

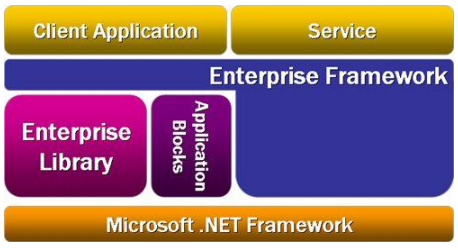
Zatím pouze chystanou továrnou je **Web Client Software Factory**, která je zmíněna v dokumentaci k SCSF a na Wikipedii (http://en.wikipedia.org/wiki/Software_factory), zatím je však známo pouze předběžné datum vydání, které by mělo být počátkem roku 2007.

Tyto softwarové továrny budou v blízké budoucnosti tvořit základ vývoje typických aplikací pro .NET Framework, už dnes se ale objevují další a další továrny, které jsou neméně zajímavé.

7.5.4 Enterprise Framework Factory

Enterprise Framework Factory je zajímavá především tím, že vedle balíčků vytvořených pomocí GAT integruje i doménově specifický jazyk pro vysokoúrovňový popis vytvářených aplikací. V současnosti zatím tato softwarová továrna není dostupná a veškeré informace tak byly čerpány z článku [Santos, 2006].

Vývoj této továrny byl podobný, jako u většiny ostatních. Prvním evolučním krokem byla integrace zkušeností, doporučených postupů a vzorů do frameworku. Jak ukazuje obrázek 43, Enterprise Framework odstiňuje cílovou aplikaci (ať už se jedná o klientský program nebo o serverovou službu) nejen od .NET Frameworku samotného, ale i od Enterprise Library a dalších aplikačních bloků. Přes poměrně zásadní architektonickou změnu vývoje aplikací pomocí Enterprise Frameworku zůstala vývojářská zkušenost podobná jako dříve: většina know-how byla uložena v dokumentaci, v referenční implementaci a v konkrétních postupech typu „krok za krokem".



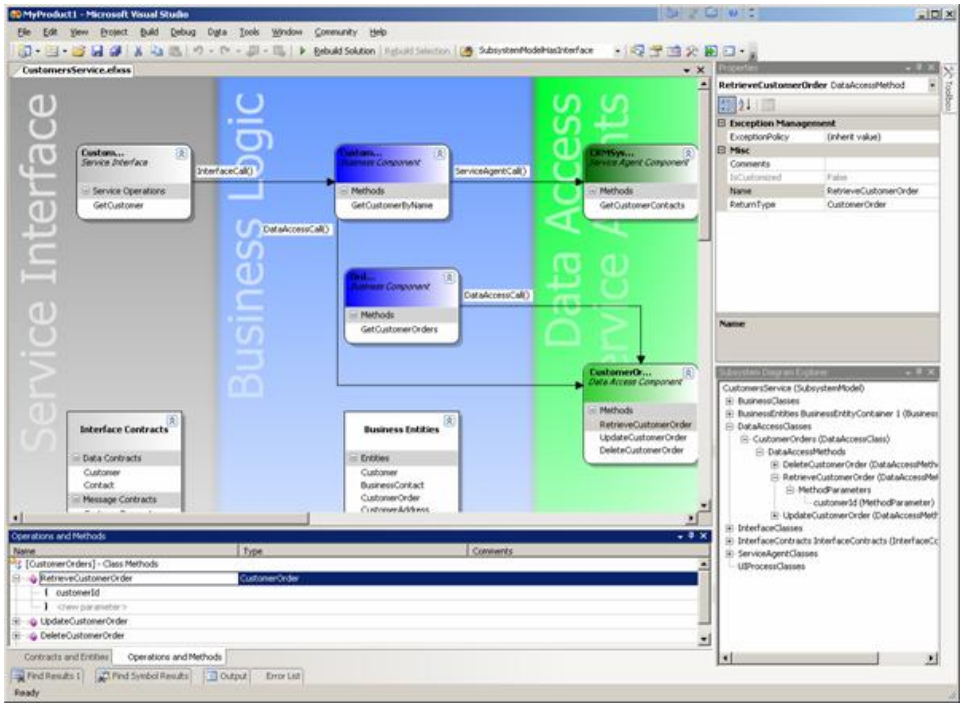
Obrázek 43 – Struktura EFx aplikačního frameworku (zdroj: [Santos, 2006])

[19] http://practices.gotdotnet.com/svcfactory
[20] http://practices.gotdotnet.com/projects/mobile

Hlavní důvodem bylo, že Visual Studio příliš dobře automatizaci vývojářských úloh nepodporovalo (jedinou cestou byly komplikované „enterprise šablony“, které obvykle dodavatele frameworku dokázaly spolehlivě odradit).

Z „frameworku“ se mohla stát „továrna“ díky uvedení Visual Studia 2005, které přineslo Guidance Automation Toolkit a DSL Tools. Některé úkoly související s Enterprise Frameworkem tak bylo možno zautomatizovat díky příslušným receptům a DSL Tools přinesly možnost vytvořit vlastní vizuální designer, jehož náhled zachycuje obrázek 44.

Cílem této továrny je poskytovat architektům a vývojářům zapouzdřené vzory, doporučení a zkušenosti, které staví na .NET Frameworku, Enterprise Library a aplikačních blocích skupiny patterns & practices, tedy na tom nejlepším, co dnes Microsoft v oblasti architektonických doporučení nabízí. Respekt a současně obavy může vyvolávat řada abstrakčních vrstev, které EFx obsahuje. Už sám .NET Framework zapouzdřuje některé vzory a doporučené postupy, Enterprise Library v tomto pokračuje více do hloubky a nyní je přidána ještě další vrstva. Přínos softwarové továrny oproti čistému frameworku se tedy může projevit především v tom, že pomůže vypořádat se s tolika vrstvami abstrakce.



Obrázek 44 – Vizuální návrhář Enterprise Framework Factory (zdroj: [Santos, 2006])

Jak bylo řečeno, tato softwarová továrna zatím není k dispozici a nemohu proto popsat vlastní zkušenosti. Návrh aplikace pomocí DSL je však do budoucna velkým příslibem.

7.5.5 Project Glidepath

S příchodem Windows Vista a .NET Frameworku 3.0 se objeví řada zcela nových aplikačních rozhraní, knihoven a technologií, které není kvůli jejich rozsahu jednoduché zvládnout. Vedle Windows Forms bude možno uživatelská rozhraní vytvářet pomocí Windows Presentation Foundation, webových služeb a nejen jich se dotkne příchod Windows Communication Foundation a úplnými novinkami budou subsystémy Windows Workflow Foundation a Windows CardSpace.

Microsoft proto připravil softwarovou továrnu nazvanou Project Glidepath[21], která by měla první kroky s vývojem pro Windows Vista a .NET Framework 3.0 usnadnit. Microsoft navíc tento projekt pojal „komunitně“, takže vedle atraktivní domovské stránky této továrny zřídil i pravidelně aktualizovaný weblog a zajímavostí je, že se obsah balíčků v továrně bude aktualizovat pomocí RSS.

Pro tuto práci sice není Project Glidepath po odborné stránce nijak významný, demonstruje ale, kde všude mohou v budoucnosti softwarové továrny najít uplatnění.

7.6 Problémy současných softwarových továren

Příklad HL7 Software Factory ukázal, že i když je k dispozici více než rok času a řada špičkových odborníků, nemusí být implementace softwarové továrny úspěšná. To u některých vyvolalo obavy, jestli je vůbec koncept softwarových továren životaschopný. Z tohoto pohledu je zajímavá odpověď Maura Regia, který stál u zrodu HL7SF a je rovněž spoluautorem základního dokumentu [Regio, Greenfield, 2005]. Říká, že:

> Softwarové továrny jsou dlouhodobou iniciativou a současný stav nástrojů a platforem je pouhým náznakem toho, jak by jednou podpora softwarových továren mohla a měla vypadat.
>
> Ačkoliv se HL7SF nepodařilo implementovat v plné šíři, jedna část je již hotova: generování webových služeb odpovídajících standardům HL7. Spíše než za produkt tuto část Regio označuje za prototyp demonstrující, že taková továrna existovat může. Druhá část je komunitní projekt, který má za cíl vytvořit HL7 knihovny na platformě .NET. Ten zatím prochází vývojem a konkrétní výsledky by měly být k dispozici ke konci roku 2006.
>
> Regio považuje názor, že by se softwarové továrny měly zaměřovat pouze na úzkou doménu, za správný. Dodává, že tyto „malé“ továrny by pak měly být komponovány do větších celků, které by tak mohly podporovat např. celé zdravotnické odvětví. Připouští komplexnost tohoto přístupu, ten je ale podle něj nevyhnutelný.

Na závěr své odpovědi, publikované v článku [Osherove, 2006a], píše Regio následující:

> „Jsou softwarové továrny komplikované? Ano, jsou velmi komplikované. Je robot používaný na automobilové lince komplikovaný? Ano, je. Je užitečný a efektivní? Zeptejte se Fordu, BMW nebo jiného výrobce automobilů.
>
> Máme dnes k dispozici nástroje, které by umožnili realizaci vize softwarových továren? Zatím nikoliv. Pokud se však podíváte na nástroje jako Guidance Automation Toolkit (GAT) nebo DSL Tools, je to cesta správným směrem.
>
> Neočekávejte, že dnes vyvinete velmi sofistikovaného „robota“. Současně ale neočekávejte, že vize softwarových továren zcela selže.“

Současný stav tedy neumožňuje budování opravdu komplexních vertikálních továren, předchozí kapitola však zároveň ukázala, že použitelné softwarové továrny pomalu vznikají a že tento koncept je živý. V současnosti je sice předčasné odhadovat, zda bude dlouhodobě a globálně úspěšný, Microsoft mu však zatím věří.

[21] http://www.projectglidepath.net/

8 Vývojové prostředí pro návrhové vzory

Předchozí kapitoly se věnovaly různým aspektům podpory návrhových vzorů ve vývojových prostředích, formalizačními technikami počínaje a vizí softwarových továren konče. Tato kapitola všechny nabyté poznatky shrnuje a rozebírá koncept „vývojového prostředí pro návrhové vzory“.

8.1 Možnosti podpory vzorů ve vývojových prostředích

Konkrétní podoba podpory návrhových vzorů se v různých vývojových prostředích může lišit a není nijak standardizovaná. Pro účely této kapitoly je úroveň podpory rozdělena do tří skupin, které nemusejí být univerzálně použitelné, na tomto místě však dovolují demonstrovat některé další záležitosti. Tyto tři úrovně jsou následující:

1. **Návrhové vzory jsou zapouzdřeny do knihoven**, ať už se těmto knihovnám říká „aplikační bloky“, „framework“ nebo i jinak. Vývojové prostředí potom práci s návrhovými vzory nepodporuje přímo, ale skrze tento znovupoužitelný kód. **Vzory jsou pro vývojáře i pro vývojové prostředí zcela transparentní**.
2. Vývojové prostředí obsahuje **automatizační prostředky, které práci s návrhovými vzory usnadňují**. Někdy je možné, že jsou vzory před vývojářem zcela skryty – ten např. v menu vybere položku „Add a View“ a IDE se samo postará o aplikování vzoru MVC, aniž by vývojář musel tento vzor do detailu ovládat.
3. Vývojové prostředí obsahuje **přímou podporu pro návrhové vzory**, před vývojářem nic neskrývá a naopak od něj očekává jejich dobrou znalost.

Podpora na prvním stupni je už z podstaty přítomna ve všech vývojových prostředích, přitom je často přehlížena. Je to proto, že větší knihovny či frameworky často proklamují „vestavěnou podporu osvědčených praktik a návrhových vzorů“, mezi vývojáři však o ní často neexistuje příliš konkrétní povědomí. Příkladem může být .NET Framework, který integruje řadu návrhových vzorů, např. Observer (události), Two Step View (ASP.NET), Front Controller (HTTP moduly), Abstract Factory (ADO.NET) a další, přesto ho lze používat, aniž by bylo nutno o návrhových vzorech cokoliv vědět. Protože se do různých frameworků integruje čím dál větší množství znalostí a vzorů, podpora naznačená v prvním bodu se bude neustále zlepšovat, a to přirozeným způsobem.

Druhý stupeň nejlépe odpovídá dnešním softwarovým továrnám skupiny *patterns & practices* tak, jak byly představeny v kapitole 7.5. Ve Visual Studiu 2005 zajišťuje takto pojatou podporu vzorů Guidance Automation Toolkit, respektive běhové prostředí Guidance Automation Extensions (viz kapitolu 7.4.2). I zde je možné, aby návrhové vzory zůstaly před vývojářem skryty, obvykle se ale jejich znalost již minimálně doporučuje, např. v popisované Smart Client Software Factory jsou všechny související vzory popsány v dokumentaci a od vývojáře se jejich ovládání očekává.

Třetí stupeň podpory je přítomen např. v nástroji Enterprise Architect, který byl představen na straně 27 (nejedná se sice o klasické vývojové prostředí, koncept však demonstruje dobře). Uživatelské rozhraní obsahuje příkazy typu „vložit návrhový vzor“, po čemž vývojář vybere vhodný vzor, nastaví mu parametry a vloží jej do aktuálního projektu. Zde hraje důležitou roli formální zápis vzorů, tedy tematika dopodrobna rozebíraná v kapitolách 3, 4 a 5. Pro Visual Studio 2005 existuje podpora vzorů i na této úrovni, jak bude podrobněji popsáno níže.

Zde naznačené rozdělní vyvolává dvě otázky. Tou první je, která úroveň podpory je „nejlepší“. Jednotlivé úrovně byly seřazeny od nulové, respektive nepřímé podpory vzorů až po jejich „plnou“ podporu, což ovšem nijak neimplikuje jejich praktickou využitelnost nebo kvalitu. V určitém smyslu může např. první úroveň vypadat nejatraktivněji, protože používání fra-

meworku vede k automatickému a samozřejmému používání vzorů. Na druhou stranu takováto prostředí zaostávají ve chvíli, kdy určité vzory nejsou ve frameworku obsaženy a je nutná jejich dodatečná implementace. Pokud vezmeme v úvahu druhý krajní případ, tedy vývojové prostředí podporující návrhové vzory jako takové, zde bude podpora vývoje založeného na návrhových vzorech výborná, na druhou stranu se ale zvýší nároky na vývojáře, větší bude rovněž náchylnost ke špatnému aplikování vzorů a podobně. Druhou úroveň je potřeba považovat za řešení prostřední, nikoliv kompromisní. V některých případech bude takováto podpora vyhovovat, v jiných ne, podobně jako tomu bylo u první a třetí úrovně. Mým osobním názorem je, že „vývojové prostředí pro návrhové vzory" by mělo být *schopno* podporovat všechny úrovně, ale ne nutně současně. Visual Studio takovým vývojovým prostředím je.

Druhá otázka zní, jak do naznačeného dělení zapadají doménově specifické jazyky. Stručnou odpovědí je „nijak". Doménově specifické jazyky jsou konceptem, který s návrhovými vzory může, ale nemusí souviset:

> Doménově specifický jazyk může být jedním z prostředků, jak vzory před vývojářem zcela skrýt. Příkladem je situace, kdy mají programátoři za úkol vytvářet uživatelská rozhraní průvodců pro různé situace, např. pro rezervaci pokoje v hotelu (průvodce od zákazníka zjistí požadovanou lokalitu, typ pokoje, osobní údaje a podobně). Z počátku programátoři vytvářejí **každého průvodce „od nuly"** pomocí Windows Forms, ale formuláře občas vypadají a chovají se nekonzistentně. Tým si proto stanoví interní **vzory**, jako např. že každá obrazovka musí vždy obsahovat tlačítka Back, Next, Cancel a Finish, že Back uživatele přivede na poslední zobrazenou stránku a nikoliv na fixně stanovenou obrazovku a podobně. Postupem času vývojáři zjistí, že stále dokola píší ten samý kód pro zobrazování základních tlačítek a rozhodnou se vytvořit malý interní **framework**. To jim značně ulehčí práci, ale kód není pro členy jiných vývojových týmů dostatečně čitelný a rovněž je potřeba zaškolovat nově příchozí programátory. Nakonec tedy dojde k vytvoření **vizuálního jazyka** pro popis průvodců, který může být podobný např. stavovému diagramu a je tím pádem dobře srozumitelný všem. Tým si tak prošel cyklem „instance vzor framework jazyk", který je pro některé scénáře typický.[22] Doménově specifický jazyk tak stojí na konci procesu, který vývojáře od používání doménově specifických vzorů prakticky zcela odstínil.[23]
>
> Doménově specifický jazyk může s podporou vzorů pomáhat. Příkladem je softwarová továrna Enterprise Framework Factory popsaná v kapitole 7.5.4.
>
> Pokud chce vývojové prostředí podporovat návrhové vzory jako objekty první třídy, i zde mohou doménově specifické jazyky pomoci. Příklady budou uvedeny níže.

8.2 Visual Studio jako vývojové prostředí pro návrhové vzory

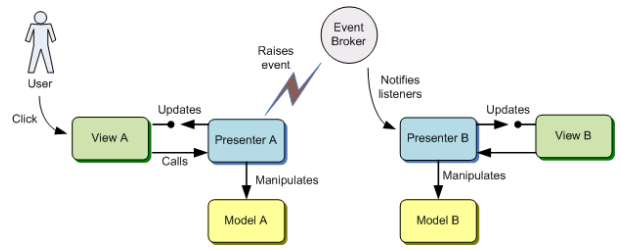
Jak bylo řečeno v předchozí kapitole, na základní úrovni podporuje návrhové vzory každé vývojové prostředí a Visual Studio v tomto ohledu není výjimkou. Nejen že je řada vzorů integrována v samotném .NET Frameworku, ale skupina patterns & practices navíc uvolňuje nové knihovny, které zapouzdřují další osvědčené praktiky a vzory. Příkladem jsou různé aplikační bloky nebo jejich balíček nazvaný Enterprise Library. I mimo Microsoft vznikají různě zaměřené frameworky, např. *Ingenious MVC* (MVC framework pro ASP.NET i Windows Forms) nebo *MonoRail* (webový framework inspirovaný Ruby on Rails).

---

[22] V literatuře, např. [Yacoub, Ammar, 2003], se někdy uvádí poněkud jednodušší životní cyklus vzoru „dolování čištění znovupoužívání". Ve světě doménově specifických jazyků je však poslední krok poměrně logický.

[23] Uvedený příklad je dopodrobna rozvinut v případové studii, která je po instalaci DSL Tools k dispozici na cestě C:\Program Files\Visual Studio 2005 SDK\<číslo verze>\VisualStudioIntegration\Samples\DSLTools\End-End.WizardUIP.

Na druhé úrovni jsou vzory podporovány primárně nástrojem Guidance Automation Toolkit, spektive běhovým prostředím Guidance Automation Extensions.[24] Na těchto dvou tách stojí většina současných softwarových továren, jak bylo demonstrováno v kapitole 7.5. ry jsou obvykle základním způsobem popsány v dokumentaci a Visual Studio pak s jejich aplikováním pomáhá pomocí kontextových bídek. Dokumentace vzorů bývá pouze základní, např. SCSF u každého vzoru uvádí „pouze" problém, řešení, nevýhody, odkaz do referenční implementace a odkaz na další zdroje. Řešení navíc není zachyceno pomocí UML diagramu, ale pouze slovním popisem někdy doplněným o náčrtek, který není v žádné standardní notaci a spoléhá na čtenářovu intuici.



Obrázek 45 – Příklad notace používané k zachycení vzorů v dokumentaci SCSF

Na trhu se v současnosti začínají objevovat nástroje třetích stran, které se rovněž snaží ulehčit aplikaci doporučených praktik, vzorů a frameworků. Příkladem je *NetAdvantage CAB Extensibility Kit* (NCEK)[25], což je add-in do Visual Studia, který pomáhá s vývojem aplikace postavené na Composite UI Application Blocku. Podporou Enterprise Library a CAB se rovněž chlubí nástroj DeKlarit[26].

Vývojové prostředí třetí úrovně by mělo umět podporovat vzory jako takové, nezapouzdřené do knihoven a frameworků. Jak ukázala kapitola 6 (*Podpora návrhových vzorů v současných nástrojích*), v oblasti modelovacích prostředí a CASE nástrojů není situace zdaleka ideální a Visual Studio v tomto ohledu také momentálně žádnou výraznou podporu nepřináší. Skutečnost, že třetí úroveň příliš dobře nepodporují dodavatelé CASE nástrojů ani Microsoft a že v současnosti nic nenasvědčuje tomu, že by se situace měla změnit, určitým způsobem naznačuje, že tito dodavatelé patrně třetí úroveň nepovažují za nejdůležitější. Bylo by takové vývojové prostředí vůbec užitečné?

Osobně vidím následující přínosy podpory vzorů jakožto objektů první třídy:

1. Pokud by vývojové prostředí podporovalo detekci vzorů, bylo by možné vizualizovat stávající systémy a lépe je tak pochopit.
2. Vizualizace návrhových vzorů by pozvedla úroveň abstrakce, skrze kterou by bylo možné nahlížet na implementovaný systém.
3. Zjednodušilo by se vkládání vzorů (eliminoval by se problém „implementační režie").
4. Bylo by možné systém validovat
5. Vývojové prostředí by bylo vhodné pro výuku vzorů

Třetí úroveň podpory má ale současně své problémy:

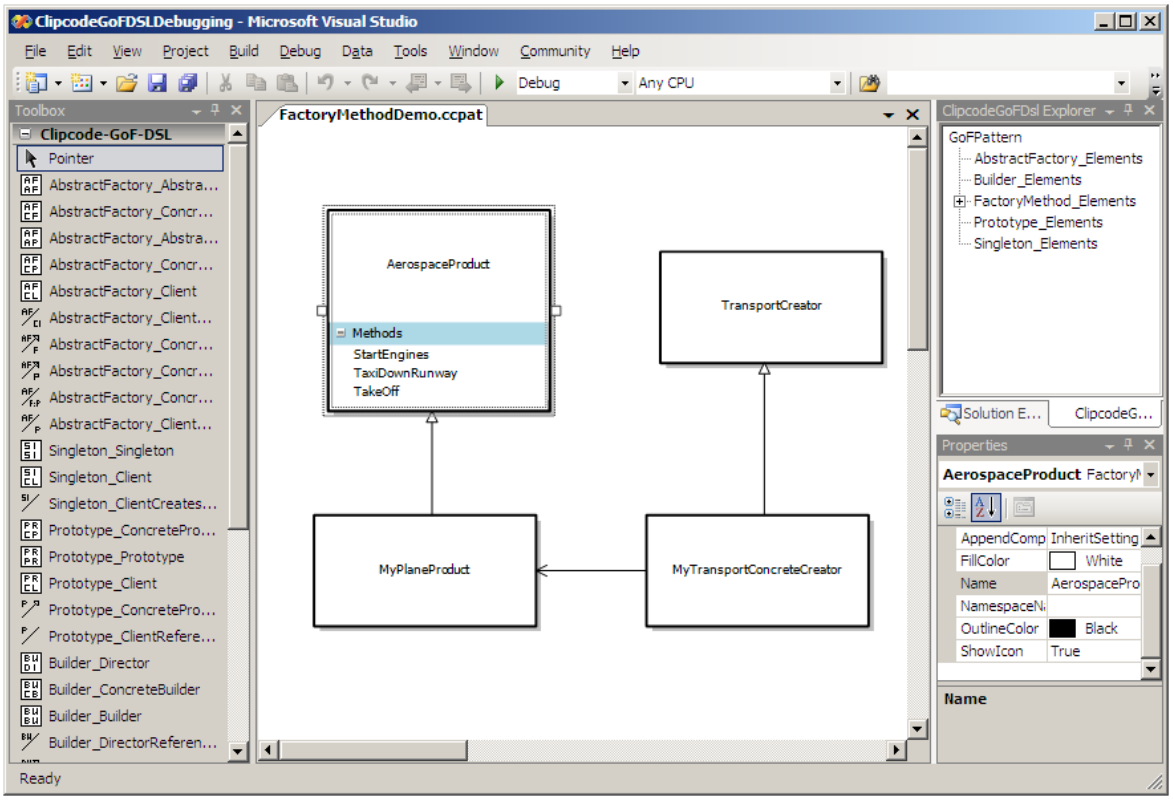
1. Je implementačně náročná. Například detekce vzorů ve stávajících systémech představuje velmi obtížný úkol, ale ani vložení vzoru tak, aby došlo k jeho plynulému zasazení do stávajícího systému, není triviální záležitostí.
2. Problémem je vhodná notace. Jak bylo ukázáno v kapitolách 4 a 5, každá notace má svá omezení. V knihách jsou v současnosti preferovány UML diagramy, ty ale ve Visual Studiu nemají přímou podporu a ve vztahu k návrhovým vzorům trpí určitými problémy, jak se podrobněji uvádí v kapitole 3.5.

Následující kapitola rozebírá tuto tematiku do detailů.

[24] V době psaní práce byla nejaktuálnější „Technology Preview" verze z června 2006, která však již vykazovala dostatečnou stabilitu.
[25] http://www.infragistics.com/resource/Technologies/CAB/Default.aspx
[26] http://www.deklarit.com/

8.3 Clipcode-GoF-DSL: doménově specifický jazyk pro návrhové vzory

V červenci 2005 dal Eamon O'Tuathail k dispozici doménově specifický jazyk pro některé GoF vzory nazvaný Clipcode-GoF-DSL ([O'Tuathail, 2005], [O'Tuathail, 2006a]). Jedná se o první a v současnosti také jediný počin svého druhu, který bude v této kapitole představen a prodiskutován. Při testování byla použita aktuálně nejnovější verze z března 2006, pro jejíž běh je vyžadováno Visual Studio 2005 SDK March CTP a po drobných úpravách je použitelná i verze SDK April 2006 V2 RTM.



Obrázek 46 – Doménově specifický jazyk pro návrhové vzory Clipcode-GoF-DSL

Používání Clipcode-GoF-DSL demonstruje obrázek 46:

Největší plochu zabírá vizuální návrhář, který používá notaci podobnou UML class diagramu.

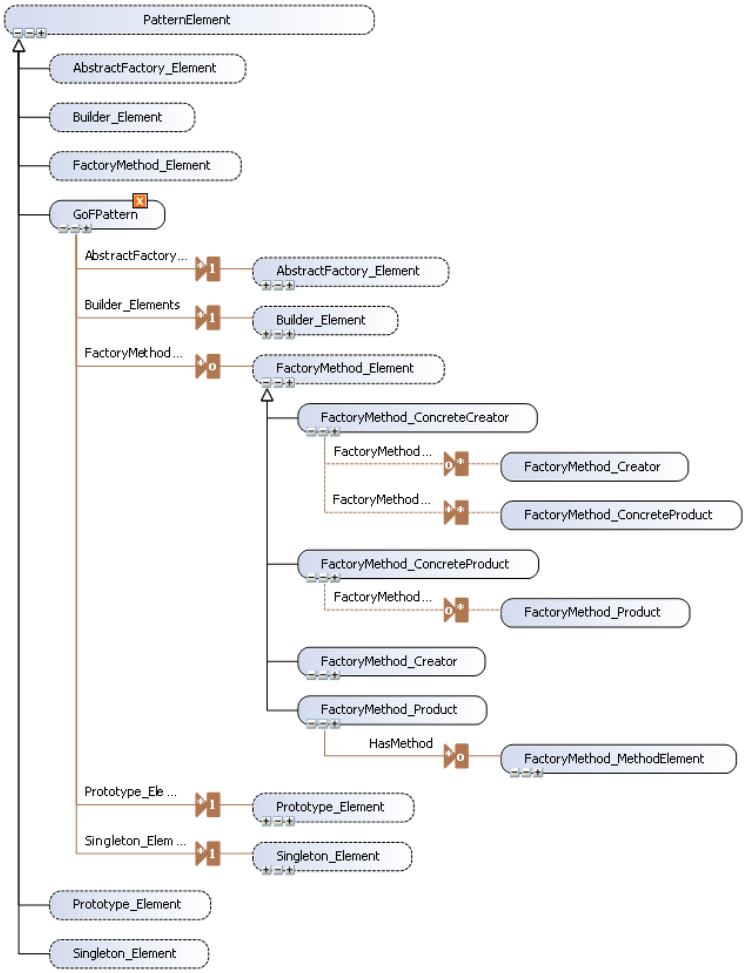
Toolbox (vlevo) obsahuje všechny podporované vzory, respektive všechny role těchto vzorů. U uvedeného příkladu vzoru Factory Method jsou dostupné tvary pro role Creator, Product, Concrete Creator a Concrete Product. Pro každý vzor jsou dále dostupné speciální tvary pro spojnice, u vzoru Factory Method se jedná o tvary ConcreteCreatorReferences (na uvedeném obrázku je tento tvar použit mezi boxy MyTransportConcreteCreator a MyPlaneProduct), ConcreteCreatorInherits (MyTransportConcreteCreator TransportCreator) a ConcreteProductInherits (MyPlaneProduct AerospaceProduct). Proč se autor tohoto DSL rozhodl pro vytvoření speciálního tvaru pro každou součást vzoru, bude uvedeno níže.

Napravo je potom zobrazen standardní dialog vlastností pro editaci povětšinou vizuálních atributů vybraného prvku.

Nad panelem vlastností je „Clipcode-GoF-DSL Explorer“, který zachycuje strukturu diagramu ve stromovém uspořádání. Po kliknutí na nějaký uzel je možné editovat jeho vlastnosti v odpovídajícím panelu, nedojde však k vybrání odpovídajícího elementu v diagramu.

Spojnice se v diagramu chovají „chytře“, např. u vzoru Factory Method nelze natáhnout spojnici ConcreteProductInherits mezi jinými elementy, než mezi abstraktním a konkrétním produktem. Základním a vestavěným způsobem je tedy ověřována validita vytvářeného modelu, validační framework popsaný v kapitole 7.4.1 jako takový však využit není.

Vedle integrace s Visual Studiem podporuje Clipcode-GoF-DSL také generování zdrojového kódu v jazyce C#. Tuto funkčnost obstarávají T4 šablony popsané v kapitole 7.4.1, v principu by tedy nebyl problém přidat podporu pro další jazyky, např. pro Visual Basic.



Obrázek 47 – Doménový model Clipcode-GoF-DSL

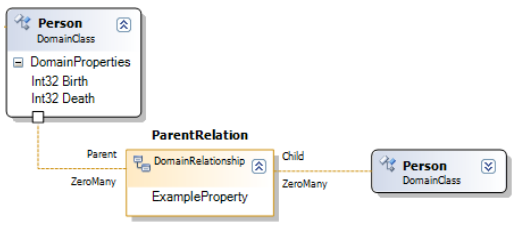
Obrázek 47 zachycuje doménový model Clipcode-GoF-DSL, tedy určitou analogii metamodelu z terminologie UML/MOF. Z úsporných důvodů je zobrazen pouze vzor Factory Method, další jsou definovány podobně. Navíc je potřeba dodat, že použitá grafická notace odpovídá DSL Tools z dubna 2006 a starších, zatímco novější návrhář (označovaný jako „v1“) již používá notaci trochu jinou. Principy se však příliš nezměnily.

V doménovém modelu je pro každou roli podporovaných návrhových vzorů definována jedna doménová třída, která je na obrázku zachycena obdélníkem se zaoblenými rohy. Mezi jednotlivými třídami mohou existovat tři druhy vztahů (stejné druhy vztahů jsou zachovány i v DSL Tools „v1“):

1. **Vkládání** (*embedding*), kdy jedna doménová třída obsahuje třídy další. V diagramu je tento vztah zachycen nepřerušovanou čarou.
2. **Odkazování** (*reference*) je slabším vztahem než vkládání a znamená, že doménová třída A *ví o* třídě B nebo že ji *využívá*. Zakresluje se přerušovanou čarou.
3. **Dědičnost** (*inheritance*) ve stejném smyslu jako v OOP.

V původní verzi byla podporována snadná konverze mezi vztahem vkládání a odkazování, takže v podstatě pouze stačilo na spojnici klepnout pravým tlačítkem myši a vybrat odpovídající příkaz. Z DSL Tools „v1“ tato možnost zmizela.

Třída má v daném vztahu vždy nějakou roli. Role jsou na diagramu zachyceny pomocí hnědých šipek nebo obdélníčků se symboly 0, 1 nebo *, které naznačují možnost či nemožnost navigace vztahu a jeho kardinalitu, případně parcialitu. V DSL Tools „v1“ doznalo zobrazení vztahů dosti podstatné změny, jak ukazuje obrázek 48. Relace tak nyní může mít své atributy jako jakákoliv jiná doménová třída.



Obrázek 48 – Zobrazení vztahu v DSL Tools „v1“

Tolik ke stručnému popisu Clipcode-GoF-DSL, který byl svého času přijat s poměrně velkým nadšením (viz [O'Tuathail, 2005]) a jedná se minimálně o zajímavou demonstraci toho, jak lze DSL Tools využít. Pozoruhodné rovněž je, že implementace trvala jednomu člověku pouze několik dní. Přesto je potřeba upozornit na některé limity, na které Clipcode-GoF-DSL naráží.

Nejvíce vadí, že jeden element diagramu může vždy zachycovat pouze jednu roli jednoho návrhového vzoru. V kapitole 6 byla jako příklad používána situace, kdy mají být konkrétní továrny ve vzoru Abstract Factory implementovány jako Singletony. Je tedy nutné, aby jedna třída v UML class diagramu vystupovala jak v roli Concrete Factory ze vzoru Abstract Factory, tak v roli Singletonu z návrhového vzoru Singleton. Tuto situaci nelze v Clipcode-GoF-DSL zachytit, což v podstatě znamená, že jej nelze využívat v reálném vývoji.

Trochu problematické je rovněž generování kódu. Clipcode-GoF-DSL umožňuje velmi snadno získat fungující kód z daného diagramu, jedná se však o jednorázový proces, po kterém již není model s kódem nadále v synchronizaci. Opačný případ, který je podle mého názoru určitým ideálem, demonstruje Class Designer dodávaný s Visual Studiem – zde je kód v programovacím jazyce nejen v synchronizaci, ale v podstatě se jedná jen o jiný pohled na stejná data. Editace v kódu se tedy okamžitě projeví v Class Diagramu a naopak.

Je rovněž poněkud nepraktické, že na každý návrhový vzor připadá několik samostatných tvarů, takže už při pěti podporovaných návrhových vzorech je Toolbox dlouhý a nepřehledný (viz obrázek 46 na začátku této podkapitoly). Eamon O'Tuathail v odpovědi k článku [Perrone, 2005] vysvětlil, proč zvolil právě tuto cestu.

Vybíral ze třech možností: jeden element na vzor, jeden element pro společné prvky napříč všemi vzory a nakonec jeden element pro každou roli každého návrhového vzoru. První přístup by znamenal, že vývojář z Toolboxu přetáhne celý návrhový vzor a ve vizuálním návrháři mu pak nastaví potřebné parametry. Tento přístup by fungoval u jednoduchých vzorů typu Singleton, ale u komplikovanějších vzorů typu Abstract Factory by nedovoloval dostatečnou pružnost (element na diagramu může obsahovat několik oddílů, které se např. v Class Designeru používají pro vnořené třídy, tyto oddíly však nemohou být zanořené a rovněž by nedošlo k optimálnímu využití plochy vizuálního návrháře).

Druhý přístup by znamenal, že jeden element diagramu bude sdílen několika rolemi z různých návrhových vzorů. Řada návrhových vzorů má totiž roli Client, Concrete<something> nebo Abstract<something>, nabízí se proto využití jednoho elementu pro spřízněné prvky. V tomto případu je však problémem sémantika – pokud by byl definovaný vztah dědičnosti mezi prvky

Abstrack<something> a Concrete<something>, znamenalo by to, že např. ConcreteProduct může dědit po AbstractFactory. Syntakticky bude vše v pořádku, sémanticky však model dávat smysl nebude. Eamon O'Tuathail připouští, že tento problém by mohl mít řešení, on si však vybral třetí cestu, kde je pro každou roli definován jeden element.

Tato kapitola tedy ukázala, že Clipcode-GoF-DSL je zajímavou demonstrací využití DSL Tools, že se ale současně jedná především o „proof of concept" spíše než o reálně použitelný nástroj. Využití Clipcode-GoF-DSL spočívá především ve vytvoření jednoho návrhového vzoru, jeho přizpůsobení ve vizuálním návrháři, vygenerování zdrojového kódu a manuální integraci do stávajícího kódu vyvíjené aplikace. Jakmile je jednou tento proces ukončen, není žádná cesta zpět. Clipcode-GoF-DSL je tak možno považovat za jakési vizualizované a modifikovatelné fragmenty kódu, větší ambice tento doménově specifický jazyk nemá.

8.4 Vlastní přístup k podpoře vzorů ve Visual Studiu

Tato kapitola nastiňuje, jak by mohla být podpora „třetí úrovně" ve Visual Studiu realizována. Je potřeba podotknout, že stav DSL Tools v srpnu 2006 byl spíše demonstrací budoucí platformy než aktuálně použitelným nástrojem. Dokumentace se často odkazovala na zastaralé SDK z dubna 2006 nebo nebyla dostupná vůbec, dodávané příklady nebyly funkční a řadu chyb obsahoval i nástroj samotný. Tato kapitola proto prezentuje výsledky snažení, které bylo za daných podmínek možné dosáhnout.

8.4.1 Koncept

Tato kapitola vyjadřuje můj osobní názor na to, jak by mohla nebo měla podpora vzorů ve Visual Studiu vypadat. Tento názor si nedělá nárok na univerzální platnost a představuje jen jeden z možných přístupů.

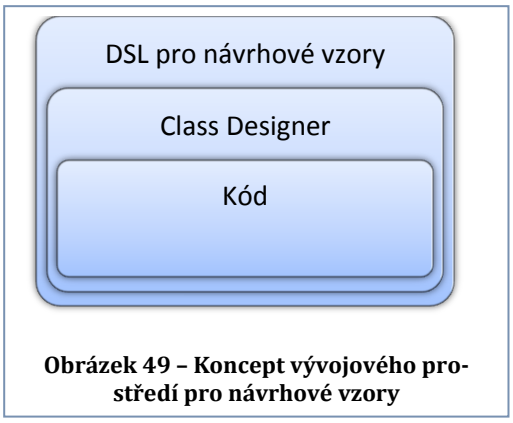
Stanovil jsem si následující základní požadavky:

1. Jazyk pro popis návrhových vzorů musí být vizuální. V kapitole 4 byly uvedeny příklady jak vizuálních, tak textových jazyků pro popis vzorů, podle mého názoru ale může skutečnou přidanou hodnotu přinést pouze grafická notace.
2. Návrh by měl být přímo převoditelný do zdrojového kódu. Některé notace uvedené v kapitolách 4 a 5 tento požadavek nepovažují za klíčový, ve vývojovém prostředí je však otázka zdrojového kódu podstatná.
3. Zdrojový kód by měl být snadno převoditelný do vizuální notace, a to nikoliv pomocí detekce, ale zcela automaticky (jedná se tedy o určitý protipól předchozímu požadavku). Tuto vlastnost řada formalizačních technik vůbec neuvažuje, přitom se jedná o logický způsob, jak eliminovat algoritmicky obtížnou detekci vzorů.
4. Model by měl být v první řadě přehledný a srozumitelný.
5. Model by mělo být možné validovat, případně kontrolovat na dodržování doporučených praktik a postupů stanovených architektem (např. že konkrétní továrny vzoru Abstract Factory by měly být v daném projektu implementovány jako Singletony). Kromě syntaxe by tedy vývojové prostředí mělo hlídat i sémantickou správnost modelu.

První bod ve Visual Studiu pomáhají realizovat nástroje pro doménově specifické jazyky. Ty současně obsahují platformu, která může pomoci i v dalších bodech, např. generování kódu usnadňují T4 šablony a pro validaci existuje validační framework. Do jisté míry unikátní je třetí požadavek, který má přitom velkou praktickou hodnotu: umožňuje deklarativním zápisem provázat svět kódu se světem diagramů.

Ideu vývojového prostředí pro návrhové vzory nastiňuje obrázek 49. Základem jsou tři hlavní úrovně. Na té první je samotný zdrojový kód v jazyce C#, případně Visual Basic .NET. Nízko-

úrovňovou manipulaci s kódem zajišťuje Class Designer, a to především proto, že obsahuje velké množství funkčnosti „zdarma“ (je standardně dodáván s Visual Studiem). Návrh vývojového prostředí se snaží tohoto faktu využít a Class Diagram do své architektury integrovat. Na nejvyšší úrovni stojí doménově specifický jazyk pro popis návrhových vzorů, který na rozdíl od diagramu tříd umí zachytit i pokročilejší vztahy mezi jednotlivými prvky vyvíjené aplikace.



Obrázek 49 – Koncept vývojového prostředí pro návrhové vzory

Hlavní charakteristikou tohoto konceptu je integrace „top-down“ i „bottom-up“ přístupu. Jak bylo uvedeno dříve, některé formalizační techniky se zaměřují na „dokonalý“ vysokoúrovňový popis vzorů, ale zcela přehlížejí implementační stránku věci, zatímco na druhé straně existují techniky, které se od textového zápisu vzorů ani neodpoutaly. Tento koncept však uvažuje jak grafický popis vzorů v jejich komplexnosti, tak záležitosti související s kódem.

Rovněž je potřeba poznamenat, že uvedená struktura je pouze schematická. Class Designer totiž sám o sobě pracuje s doménově specifickým jazykem zaměřeným na návrh tříd a jejich vztahů v prostředí .NET Frameworku, takže jeho případné rozšíření pro potřeby vzorů by eliminovalo nutnost třetí vrstvy jakožto samotné a relativně oddělené komponenty. Na druhou stranu by rovněž bylo možné prostřední vrstvu zcela vynechat a věnovat se pouze kódu a doménově specifickému jazyku pro vzory, protože kód a class designer jsou v podstatě jen jiným pohledem na stejná data. Obrázek 49 je tak skutečně především konceptuálním návrhem spíše než nějakým konkrétním ztvárněním, což se ovšem podle mého názoru nevylučuje s užitečností všech třech vrstev.

8.4.2 Úroveň 1: Kód

Ve zde popisovaném konceptu hraje kód ústřední roli – bez vyřešení této nejnižší komponenty by nebylo možné postupovat dále. Cíle byly následující:

1. Poskytnout vývojovému prostředí dostatek informací k tomu, aby mohlo z kódu snadno získat použité vzory.
2. Vyhnout se jakýmkoliv nestandardním zásahům do syntaxe používaného jazyka
3. Vyhnout se používání nástrojů nebo technik, které by byly unikátní nebo nestandardní

Druhý a třetí bod bylo snadné zajistit, protože jazyky určené pro platformu .NET přirozeným způsobem podporují atributy, tedy určité meta-informace, které lze připojit ke kódu a které lze programově získat za běhu pomocí reflexe. Platforma .NET obsahuje řadu vestavěných atributů a umožňuje definovat i atributy vlastní, čehož je s výhodou využito.

Definice vlastního atributu vypadá následovně:

```
using System;
namespace PatternAttributes
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Event |
        AttributeTargets.Field |
        AttributeTargets.Interface |
        AttributeTargets.Method |
        AttributeTargets.Property |
        AttributeTargets.Struct,
    AllowMultiple=true, Inherited=false)]
    public sealed class PatternAttribute : Attribute
    {
```

```
        private string _Name;
        public string Name
        {
            get { return _Name; }
        }
        private string _Role;
        public string Role
        {
            get { return _Role; }
        }
        private string _PatternId;
        public string PatternId
        {
            get { return _PatternId; }
            set { _PatternId = value; }
        }

        private string _Notes;
        public string Notes
        {
            get { return _Notes; }
            set { _Notes = value; }
        }

        public PatternAttribute(string name, string role)
        {
            _Name = name;
            _Role = role;
        }
    }
}
```

(Celý kód i s komentáři je uveden v příloze B).

Atribut Pattern podporuje čtyři vlastnosti, dvě povinné a dvě volitelné:

> Name (povinně) – řetězec identifikující návrhový vzor, např. „Abstract Factory“.
>
> Role (povinně) – řetězec identifikující roli v rámci návrhového vzoru, např. „Concrete Product“.
>
> PatternId (nepovinně) – řetězec jednoznačně identifikující návrhový vzor v projektu. Umožňuje řešit situaci, kdy je v projektu více návrhových vzorů stejného typu, např. více abstraktních továren, a nebylo by tedy jasné, která třída k jakému vzoru patří. Parametr je nepovinný, protože ne vždy je v jednom projektu užit určitý návrhový vzor vícekrát. Jedinečnost tohoto parametru musí být zajištěna dodatečně a je věcí konkrétního řešení, jestli se bude jednat o GUID, náhodně vygenerované číslo nebo nějaký jiný identifikátor.
>
> Notes (nepovinně) – umožňuje přiřadit k danému elementu libovolné textové poznámky, pokud je to potřeba.

Takto zvolená implementace má své výhody i nevýhody. Výhodou je jednoduchost a pružnost, protože jméno může být libovolný řetězec a množina použitelných vzorů tedy není omezená. Nevýhodou je, že vývojové prostředí nehlídá syntaktickou správnost vkládaného řetězce a bez upozornění tedy projdou i překlepy. Tato notace je však pouze jednou z částí celého konceptu a není primárně určena k tomu, aby ji psal vývojář (ačkoliv je to možné). Celý koncept naopak počítá s tím, že atributy budou přidávány vývojovým prostředím automaticky.

V prostředí .NET Frameworku lze jednomu elementu přiřadit libovolné množství atributů, třeba i stejného typu, pokud je atribut takto definován. Ve výpisu kódu je na třídu PatternAttribute aplikován atribut AttributeUsage s volitelnou vlastností AllowMultiple nastavenou na true, čímž je podporováno více atributů [Pattern(…)] pro jeden element kódu. Tím je vyřešena situace, kdy jeden element kódu vystupuje ve více různých rolích různých vzorů (např. konkrétní továrna může vystupovat v roli Concrete Factory vzoru Abstract Factory a současně v roli Singleton vzoru Singleton).

8.4.3 Úroveň 2: Class Designer

Class Designer je vizuální návrhář tříd přítomný ve Visual Studiu 2005 (v edici Standard Edition nebo vyšší). Diagramy vytvářené tímto návrhářem se nazývají Class Diagramy a v projektu Visual Studia se ukládají jako soubory s příponou .cd.

Pro koncept vývojového prostředí návrhových vzorů je Class Designer zajímavý, protože:

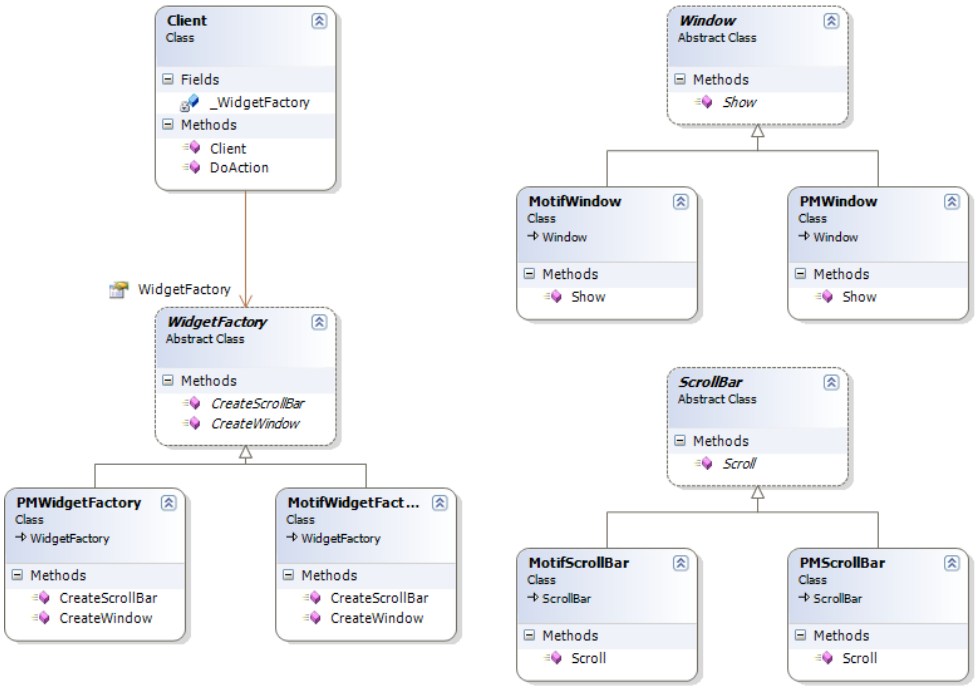
Diagram je vždy v dokonalé synchronizaci s kódem (souvisí s požadavkem generování kódu a jeho zpětným vizuálním zobrazením)

Je přehledný

Zapouzdřuje velké množství funkčnosti. Lze pomocí něj nastavit skoro všechny parametry modelovaných tříd, metod, vlastností, rozhraní atd.

Je standardně dodávaný se všemi vyššími edicemi Visual Studia (pro tyto edice je „zdarma“)

Class Designer má ve Visual Studiu dvě základní úlohy: zaprvé usnadňovat práci při vývoji kódu a zadruhé vizualizovat kód stávající. Jeho možnosti, přednosti a zápory demonstruje následující ukázka, která implementuje vzor Abstract Factory na příkladu uvedeném v GoF katalogu v sekci „motivace“ (jedná se o systém pro používání platformově nezávislých ovládacích prvků uživatelského rozhraní).

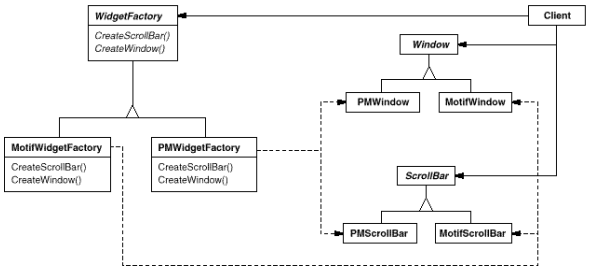


Obrázek 50 – Vizualizace kódu implementujícího vzor Abstract Factory

Obrázek 50 zobrazuje výsledek vizualizace kódu uvedeného v příloze C, který implementuje vzor Abstract Factory pomocí abstraktních tříd (další možnost bude uvedena níže). Class Diagram dobře zachycuje vztah dědičnosti a díky názvům tříd a metod lze z diagramu poměrně dobře vyčíst, jak se patrně tato ukázka kódu chová. Už nyní ale narážíme na některá omezení Class Designeru.

Především nepodporuje zobrazení vztahu závislosti (*dependency*), který se v notaci OMT nebo UML zachycuje pomocí přerušované čáry. Class Designer ve Visual Studiu podporuje pouze dva druhy vztahů: dědičnost a asociaci. Asociace vyjadřuje takový vztah tříd, kdy jedna drží referenci na druhou pomocí nějaké své vlastnosti. Class Designer umí takovou asociaci zobrazit dvěma

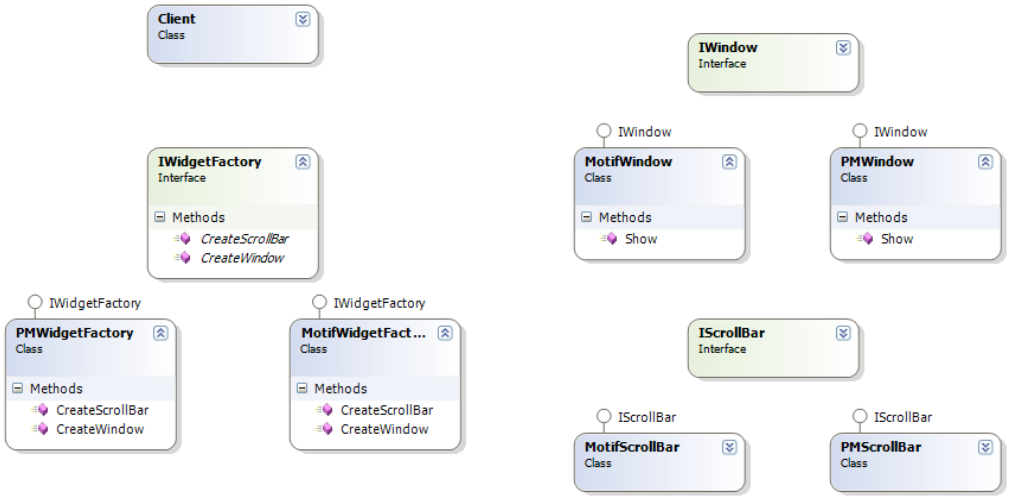
způsoby – buďto jako běžnou vlastnost, kdy mezi dvěma třídami nevede žádná čára, nebo jako „asociační vlastnost“, kdy se vlastnost nezobrazuje uvnitř boxu, ale jako šipka mezi dvěma třídami (pro vizualizaci je patrně vhodnější druhý způsob).



Obrázek 51 – Zápis vztahu závislosti pomocí přerušované čáry v notaci OMT (zdroj: [GoF, 1994])

Chybějící podpora vztahu závislosti podstatně komplikuje použitelnost Class Designeru pro zachycování komplexnějších vztahů mezi třídami. Není např. možné zachytit, že třída A volá nějakou z metod třídy B, že jako argument v nějaké své metodě přijímá instanci třídy B a podobně. Právě takovéto vztahy však často hrají v návrhových vzorech důležitou roli. Dobrou zprávou je, že se v příštích verzích Class Designeru možná podpora pro vztah závislosti objeví – příslušný feature request [Moth, 2004] zamítnut nebyl, v současnosti je označen jako „postponed“ (odložený) a díky poměrně vysokému zájmu uživatelů o tuto vlastnost Microsoft možná v příští verzi Class Designeru volání komunity vyslyší.

Další problém demonstruje obrázek 52:



Obrázek 52 – Abstract Factory implementovaná pomocí rozhraní

Zde je zobrazena vizualizace kódu uvedeného v příloze D, který implementuje stejný příklad s použitím rozhraní namísto abstraktních tříd. Class Designer implementaci rozhraní naznačuje pouze symbolem „lízátka“ a související objekty čarou vůbec nespojuje. Kdyby nebyl diagram uspořádán „logickým způsobem“, bylo by už obtížné se v něm vyznat.

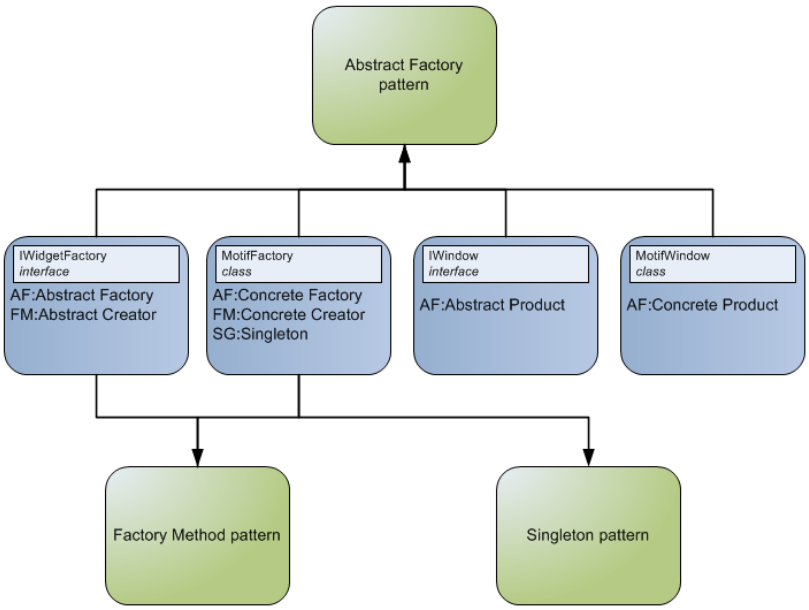
Na tomto příkladu je dobře vidět, proč by doménově specifický jazyk pro návrhové vzory pomohl. Ačkoliv kód v příloze C i D implementuje zcela stejnou situaci pomocí zcela stejného návrhového vzoru, oba Class Diagramy se od sebe výrazně liší. Doménově specifický jazyk by pomohl tyto rozdíly zapouzdřit a vývojáři prezentovat jednotný pohled na použitý návrhový vzor.

V současnosti rovněž Class Designer nepodporuje vizuální zobrazování uživatelských atributů (viz feature request [Holland, 2005]), což by se v určité formě mohlo hodit pro zobrazování atributu Pattern popsaného v předchozí kapitole. Class Designer vkládání vlastních atributů podporuje pouze přes panel vlastností, tedy nevizuálně.

Jak tedy bylo vidět, Class Designer je zajímavý nástroj pro práci s kódem, ve vztahu k návrhovým vzorům však má určité mezery.

8.4.4 Úroveň 3: Doménově specifický jazyk pro návrhové vzory

K překonání některých nevýhod Class Designeru může sloužit doménově specifický jazyk pro popis návrhových vzorů. Jeho základní strukturu zachycuje obrázek 53:



Obrázek 53 – Prototyp doménově specifického jazyka pro návrhové vzory

Modře jsou na diagramu zachyceny třídy, rozhraní a další elementy kódu, zatímco zeleně jsou zachyceny návrhové vzory samotné. Šipky naznačují, který element kódu implementuje který návrhový vzor, přičemž uvnitř modrého boxu je vždy naznačena role, ve které daný element ve vztahu k návrhovému vzoru vystupuje. Tato notace přirozeným způsobem podporuje situace, kdy element kódu implementuje více návrhových vzorů, jak demonstruje např. rozhraní IWidgetFactory nebo třída MotifFactory.

Tento doménově specifický jazyk jde trochu jinou cestou než řada notací uvedených v kapitole 4. Ty se často snažily popsat návrhové vzory v jejich principech a zákonitostech, což má své dobré opodstatnění a také využití, vždy ale za určitou cenu (nepřehlednost, přílišná vzdálenost od cílové implementace a podobně). Zde navržený doménově specifický jazyk je naproti tomu postaven na jiných prioritách – především na přehlednosti a praktické využitelnosti.

Svým přístupem se liší rovněž od Clipcode-GoF-DSL. Cílem Eamona O'Tuathaila bylo poskytnout vizuální prostředí pro vložení, úpravu a vygenerování zdrojového kódu *jednoho* vzoru, zatímco cílem této notace je podporovat více vzorů současně, jak je obvyklé v reálném vývoji.

Co se konkrétní implementace tohoto diagramu týče, v principu není problém podporovat jeho plnou synchronizaci s kódem nebo s Class Designerem, protože se vlastně jedná pouze o zúžený, specifický pohled na zdrojový kód, ze kterého je získávána pouze informace o použitých návrhových vzorech. Po označení nějakého zeleného boxu by mělo být možné z kontextové nabídky přejít do Class Diagramu, kde by byly všechny související třídy nebo rozhraní vybrány a připraveny k použití.

Celá tato kapitola však musí být zakončena jedním velkým *bohužel* – obrázek 53 je nefunkčním prototypem nakresleným ve Visiu, protože DSL Tools byly v době dokončování práce v podstatě nepoužitelné. Jak již bylo řečeno, v dubnu 2006 došlo k zavržení do té doby používaných formátů a nástrojů a Microsoft přišel s výrazně přepracovanou verzí DSL Tools, která však byla v srpnu 2006 nestabilní, nezdokumentovaná a na úrovni jakési alfa verze. Pokoušel jsem se doménově specifický jazyk pro vzory vytvořit jak v nestabilní verzi vizuálního návrháře, tak „ručně" editací příslušných (a dosti komplikovaných) XML souborů, vizuální návrhář však některé

potřebné vlastnosti zatím neobsahoval a XML formát zase nebyl zdokumentovaný (o celou práci jsem nakonec přišel tím, že po jednom zásahu do XML souboru už projekt nešel ani zkompilovat, ani otevřít ve vizuálním návrháři, kde by se dala sjednat náprava). Mrzí mě, že nebylo možné uvedené koncepty plně implementovat, stav DSL Tools to ale bohužel v termínu dokončování této práce neumožnil.

8.5 Shrnutí

Tato kapitola demonstrovala, že Visual Studio může dobře posloužit jako vývojové prostředí pro návrhové vzory. Podpora první a druhé úrovně je poskytována už dnes a představený koncept vlastního vývojového prostředí třetí úrovně ukázal, jak by mohly být návrhové vzory podporovány v blízké budoucnosti.

9 Shrnutí a závěr

9.1 Stručné shrnutí jednotlivých kapitol

Ústředním motivem celé práce bylo hledání odpovědi na otázku „jak podporovat návrhové vzory ve vývojových prostředích“. Logická stavba práce a nejdůležitější výsledky jednotlivých kapitol byly následující:

Možnostem **formálního zápisu vzorů** se věnovaly kapitoly 3 (Formální zápis návrhových vzorů), 4 (Přehled samostatných formalizačních technik) a 5 (Formalizační techniky založené na UML). Čtvrtá kapitola ukázala řadu formalizačních technik, které přinesly několik zajímavých myšlenek, na druhou stranu však často také trpěly svými problémy, nejčastěji přílišnou abstraktností, nesrozumitelností a nulovou nebo nízkou podporou v nástrojích. Pátá kapitola představila formalizační notace postavené na UML, které se snažily vylepšit zobrazování vzorů modifikací UML class diagramu. V těchto notacích se bohužel vždy projevila nepřímá úměrnost mezi přehledností diagramu a jeho schopností zachytit všechny záležitosti týkající se vzorů.

Kapitola 6 demonstrovala, na jaké úrovni lze zhruba očekávat **podporu návrhových vzorů v dnešních CASE nástrojích**. Některé nástroje návrhové vzory zcela ignorují (např. PowerDesigner), jiné je podporují poměrně dobře (např. Enterprise Architect), podpora je však často málo pružná a ne zcela plnohodnotná.

Sedmá kapitola popisovala **softwarové továrny**, které jsou pro tuto práci velmi významné. Jejich ambicí je posunout softwarový vývoj na vyšší evoluční stupeň, a to automatizací některých rutinních činností a integrací modelování do vývojového procesu. Bylo předvedeno několik příkladů softwarových továren, které se dnes začínají objevovat a které s vývojem skutečně pomáhají, rovněž ale bylo poukázáno na současné problémy, se kterými je potřeba počítat a na jejichž vyřešení bude nutno ještě nějaký čas počkat – především se jedná o stav nástrojů a platforem, které plnou realizaci softwarových továren ve Visual Studiu 2005 zatím neumožňují.

Osmá kapitola nakonec všechny poznatky zúročuje a navrhuje **koncept vývojového prostředí pro návrhové vzory**. Tomuto bodu se podrobněji věnuje následující podkapitola.

9.2 Shrnutí konceptu vývojového prostředí pro návrhové vzory

Pojem „vývojové prostředí pro návrhové vzory“ je poměrně abstraktní a kapitola 8 se ho pokoušela trochu konkretizovat. Pro účely této práce jsem úroveň podpory rozdělil do tří hlavních skupin:

1. Nepřímá podpora vzorů skrze frameworky, knihovny a jiné znovupoužitelné kusy kódu, které návrhové vzory zapouzdřují
2. Podpora vzorů skrze framework a jeho částečnou automatizaci (např. Smart Client Software Factory)
3. Plná podpora vzorů jako takových

Považuji za důležité zopakovat, že seřazení od „nejnižší“ po „nejvyšší“ podporu ještě nemusí znamenat seřazení od nejméně žádané po nejužitečnější možnost. V určitém smyslu může např. být nejužitečnější, pokud jsou návrhové vzory zapouzdřené do znovupoužitelného kódu a neví o

nich ani vývojář, ani vývojové prostředí. V jiných případech může být naopak významná třetí úroveň podpory, proto je v kapitole 8.4 představen vlastní koncept vývojového prostředí pro návrhové vzory.

Koncept vychází z technologické platformy Visual Studio 2005 + DSL Tools a staví na třech základních úrovních, kterými jsou

1. Kód
2. Class Designer
3. Doménově specifický jazyk

Hlavním rysem tohoto přístupu je integrace top-down i bottom-up přístupu, kdy hraje důležitou roli nejen vizuální notace zachycující vztahy vzorů, ale i cílová implementace. Pro zachycení vzorů na úrovni zdrojového kódu jsou použity atributy (viz příloha B), které umožňují deklarativní zachycení návaznosti kódu na jednotlivé role návrhových vzorů. Class Designer je potom do architektury řešení integrován především proto, že se jedná o poměrně zdařilý modelovací nástroj s širokou škálou funkčnosti, který je s Visual Studiem dodáván „zdarma“. Ačkoliv by tedy v principu nebyl problém se bez Class Designeru obejít, jeho použití může některé věci usnadnit. Zcela navrchu potom stojí doménově specifický jazyk navržený pro přehledné zachycení vztahu kódu k použitým návrhovým vzorům. Diagram může z principu být v dokonalé synchronizaci s kódem, protože ten informace o použitých návrhových vzorech nese v sobě.

Současná situace DSL Tools a Class Designeru bohužel zatím neumožňuje tento koncept plně realizovat. Podle informací z Microsoftu by měla první stabilní verze DSL Tools vyjít v polovině září 2006 a realizaci konceptu by také hodně napomohlo konvertování Class Designeru na tuto novou modelovací platformu, k čemuž by podle zákulisních informací také mělo *časem* dojít (v současnosti je interakce s Class Designerem možná pomocí špatně zdokumentovaného API nebo pomocí Modeling PowerToys).

9.3 Možnosti další práce

Po vydání DSL Tools „v1“ a ještě lépe i po vydání nové verze Class Designeru bude možné zde naznačený koncept realizovat. Případnému zájemci doporučuji promyslet následující otázky:

> **Je vývojové prostředí pro návrhové vzory vůbec ještě potřeba?** (Dnes je těžké odhadovat, kam se posune podpora vzorů na druhé naznačené úrovni. Jednou z možností je, že bude dostupných tolik aplikačních bloků a softwarových továren, že pro praktický vývoj bude tato úroveň podpory dostačovat. Přesto myslím, že i v budoucnu najde třetí úroveň podpory své využití, např. ve školství.)
>
> **Jak nejlépe realizovat kooperaci Class Designeru s doménově specifickým jazykem?** (Navigovatelnost z DSL na odpovídající objekt(y) Class Designeru? Rozšíření Class Designeru pro přechod na odpovídající element doménově specifického jazyka?)
>
> **Jak podporovat validaci modelu**? (Zde by měl pomoci validační framework přítomný v DSL Tools)

9.4 Závěr

Myslím, že tato práce komplexně a poctivě prozkoumala možnosti podpory návrhových vzorů ve vývojových prostředích. Konkrétně ve vztahu k platformě .NET a jazyku C# dělá velké kroky kupředu Visual Studio, které od verze 2005 podporuje doménově specifické jazyky a další aspekty vize softwarových továren. V budoucnu by se mohlo stát velmi dobrým vývojovým prostředím pro návrhové vzory.

Zdroje

U všech zdrojů je pokud možno uvedena internetová adresa, ze které lze daný materiál získat. Odkazy byly platné v den dokončení práce, tj. 21. srpna 2006.

[Albin-Amiot, Guéhéneuc, 2001] ALBIN-AMIOT, Hervé, GUÉHÉNEUC, Yann-Gaël: *Meta-Modeling Design Patterns: Application to Pattern Detection and Code Analysis*. Workshop on Adaptative Object-Models and Metamodeling Techniques. ECOOP (European Conference on Oriented Programming), 2001. http://www.yann-gael.gueheneuc.net/Work/Publications/Documents/-ECOOP01AOOSDM.doc.pdf

[Alexander, 1979] ALEXANDER, Christopher: *The Timeless Way of Building*. Oxford University Press, 1979. ISBN 0-19-502402-8

[Baroni aj., 2003] BARONI, Aline L., GUÉHÉNEUC, Yann-Gaël, ALBIN-AMIOT, Hervé: *Design Patterns Formalization*. Výzkumná zpráva. EMNantes School of Engineering, 2003. http://www.yann-gael.gueheneuc.net/Work/Publications/-Documents/Research+report+Metamodeling+June03.doc.pdf

[Bosch, 1998] BOSCH, Jan: *Design Patterns as Language Constructs*. Journal of Object-Oriented Programming, 1998. http://www.itu.dk/people/ydi/-PhD_courses/adaptability_design/bosch98design.pdf

[Brabant, 2006] BRABANT, David: *Designing and Implemeting HL7 Software Factory: story of a failure?* Článek na „David's blog", 2005. http://www.xhovemont.be/-archive/2006/05/16/1222.aspx

[Cooper, 2003] COOPER, James W.: *C# Design Patterns*. Addison-Wesley, 2003. ISBN 0-201-84453-2

[Dong, 2002] DONG, Jing: *UML Extensions for Design Pattern Compositions*, in Journal of Object Technology, vol. 1, no. 5, pages 149–161. http://www.jot.fm/-issues/issue_2002_11/article3

[Dong, Yang, 2003] DONG, Jing, YANG, Sheng: *Visualizing Design Patterns With A UML Profile*. University of Texas at Dallas, 2003. http://citeseer.ist.psu.edu/742581.html

[Dvořák, 2003] DVOŘÁK, Miloš: *Návrhové vzory (design patterns)*. Diplomová práce VŠE, 2003. http://objekty.vse.cz/Objekty/Vzory

[Eden, 02] EDEN, Amnon H.: *A Visual Formalism for Object-Oriented Architecture*. Concordia University, Canada, 2002. http://www.eden-study.org/articles/-2002/idpt02-with-corrections.pdf

[Eden, 03] EDEN, Amnon H.: *Formal Specification of Object-Oriented Design*. Research seminar, University of Essex, 2003. http://www.eden-study.org/articles/-2002/theory-ood_essex-seminar.ppt

[Fowler, 2002] FOWLER, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. ISBN 0321127420

[GoF, 1994 cz] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph, VLISSIDES, John: *Návrh programů pomocí vzorů*. Grada, 2003. ISBN 80-247-0302-5

[GoF, 1994] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph, VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2

[Greenfield, GREENFIELD, Jack: *Software Factories: Assembling Applications with*

2004] *Patterns, Models, Frameworks and Tools*. Prezentace dostupná z http://colab.cim3.net/file/work/EmergingTechnology_Conference/2004-06-03_MITRE/Greenfield_2004_06_03.ppt

[Greenfield, Short, 2004] GREENFIELD, Jack, SHORT, Keith: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004. ISBN 0471202843

[Haack, 2006] HAACK, Phil: *UML Is Chinese To Me*. Článek na blogu „You've been Haacked", 2006. http://haacked.com/archive/2006/04/15/UMLIsChineseToMe.aspx

[Hedin, 1997] HEDIN, Gorel: *Language Support for Design Patterns using Attribute Extension*. Lecture Notes in Computer Science, 1997. http://citeseer.ist.psu.edu/hedin97language.html

[Hochmann, 2006] HOCHMANN, Zbyněk: *Návrhové vzory v C#*. Diplomová práce VŠE, 2006.

[Kraval, 2001] KRAVAL, Ilja: *Úvod do sémantiky UML 1.3 podle standardu OMG s komentářem*. Objects Consulting, 2001. Dostupné na vyžádání zdarma, viz http://www.objects.cz/produkty/Produkty.html.

[Lauder, Kent, 1998] LAUDER, Anthony, KENT, Stuart: *Precise Visual Specification of Design Patterns*. Proceedings of ECOOP '98, 1998. http://citeseer.ist.psu.edu/-kent98precise.html

[Lieberman, 1986] LIEBERMAN, Henry. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*. Proceedings of OOPSLA '86, 1986. http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/-Delegation.html

[Mikkonen, 1998] MIKKONEN, Tommi: *Formalizing Design Patterns*. In Proc. ICSE'98, str. 115-124. IEEE Computer Society Press, 1998. http://citeseer.ist.psu.edu/-context/324977/0

[O'Tuathail, 2005] O'TUATHAIL, Eamon: Release of a DSL for the Gamma et al. Design Patterns. Vlákno na MSDN Forums, http://forums.microsoft.com/msdn/showpost.aspx?postid=64106&siteid=1

[Osherove, 2006] OSHEROVE, Roy: *Are Software Factories too complicated to be used in the real world?* Článek na „ISerializable - Roy Osherove's Blog", 2006. http://weblogs.asp.net/rosherove/archive/2006/05/16/446667.aspx

[Osherove, 2006a] OSHEROVE, Roy: *HL7 Software Factory Project is a success in progress, says MS Architect Mauro Regio*. Článek na „ISerializable - Roy Osherove's Blog", 2006. http://weblogs.asp.net//rosherove/archive/2006/05/18/446937.aspx

[O'Tuathail, 2006] O'TUATHAIL, Eamon: *Domain Specific Languages (and a bit about Software Product Lines, Software Factories and the Guidance Automation Toolkit)*. Clipcode, 2006. Prezentace dostupná z http://www.clipcode.biz/-workshops/dsl.html.

[O'Tuathail, 2006a] O'TUATHAIL, Eamon: *Clipcode-GoF-DSL – A Visual Studio 2005-based Domain Specific Language for the Gang of Four Design Patterns*. Clipcode, 2006. Dostupné z http://www.clipcode.biz/workshops/dsl.html.

[Pagel, Winter, 1996] PAGEL, Bernd-Uwe, WINTER, Mario: *Towards Pattern-Based Tools*. EuropLop'96, 1996.

[Reeves, 1992] REEVES, Jack W.: *What is Software Design?* C++ Journal, 1992.

http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm

[Regio, Greenfield, 2005] REGIO, Mauro, GREENFIELD, Jack: *A Software Factory Approach To HL7 Version 3 Solutions*. MSDN Library, 2005. http://msdn.microsoft.com/-library/default.asp?url=/library/en-us/dnbda/html/HL7SoftFac.asp

[Roth, 2005] ROTH, Bob: *Integrating UML 2.0 into Visual Studio Team System*. Presentation to SD Best Practices, 2005. http://training.dthomas.co.uk/-downloads/sdbp.ppt

[Santos, 2006] SANTOS, Jezz: The EFx Architectural Guidance Software Factory. Vyjde v magazínu The Architecture Journal, dostupné též z adresy http://blogs.msdn.com/jezzsa/articles/677177.aspx

[Seemann, 2005] SEEMANN, Jochen: Domain Specific Language Tools For Model-Driven Development In Microsoft Visual Studio 2005. Prezentace z konference Tech-Ed 2005. http://download.microsoft.com/download/5/a/5/-5a5790b4-f8a6-49d0-aef8-fa8c3b7f378a/ARC411_Seemann.ppt

[Shore, 2006] SHORE, James: *Quality With a Name*. James Shore „Successful Software" Articles, 2006. http://www.jamesshore.com/Articles/Quality-With-a-Name.html

[Spivey, 1992] SPIVEY, Michael J.: *The Z Notation: a reference manual*. Prentice Hall, 1992. http://spivey.oriel.ox.ac.uk/mike/zrm/index.html

[Stein, 2004] STEIN, René: *Návrhové vzory a .Net Framework*. Prezentace z „programátorských večerů" ČVUT, 2004. http://blog.renestein.net/-PermaLink,guid,641073f3-c3b7-4734-8400-84f4c8755e59.aspx

[Vlissides, 1998] VLISSIDES, John: *Notation, Notation, Notation*. C++ Report, April 1998.

[Yacoub, Ammar, 2003] YACOUB, Sherif M., AMMAR, Hany H.: *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison Wesley, 2003. ISBN 0-201-77640-5

[Perrone, 2005] PERRONE, Claudio: Clipcode-GoF-DSL and the One-To-One Mapping Anti-Pattern. Článek na blogu „Claudio Perrone's Monologues", 2005. http://developers.ie/blogs/cperrone/archive/2005/08/08/1585.aspx

[Popescu, 2006] POPESCU, Iulian: Extending Class Designer. Vlánkno na MSDN Forums, 2006. http://forums.microsoft.com/MSDN/-ShowPost.aspx?PostID=374011&SiteID=1

[Moth, 2004] MOTH, Daniel: *Class Designer - Support a dependency association*. Požadavek na novou vlastnost Class Designeru v programu Microsoft Connect, 2004. https://connect.microsoft.com/VisualStudio/feedback/-ViewFeedback.aspx?FeedbackID=115080

[Holland, 2005] HOLLAND, Doug: *Class Diagrams Should Show Custom Attributes Adorning Classes / Methods etc...* Požadavek na novou vlastnost Class Designeru v programu Microsoft Connect, 2005. http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=105498

Webové stránky a další zdroje z internetu

[Bosch] Domovská stránka Jana Bosche, tvůrce objektového modelu LayOM. http://www.janbosch.com/

[Building Smart Clients] *Building Smart Client using .NET*. http://www.codeproject.com/-dotnet/DotNetBuildSmClnts.asp

[CAB HoL] Microsoft Corporation: Composite UI Application Block - Hands On Labs. http://www.microsoft.com/downloads/-details.aspx?FamilyId=AB44F082-3ABE-4583-8844-7252FF7C9638&displaylang=en

[CAB] Microsoft patterns & practices: Smart Client – Composite UI Application Block (prosinec 2005). http://msdn.microsoft.com/-library/default.asp?url=/library/en-us/dnpag2/html/cab.asp

[Clipcode C# Design Patterns] Clipcode: Design Patterns in C#. Ukázkové implementace všech GoF vzorů. http://www.clipcode.biz/stream/gof/

[DoFactory C# Design Patterns] DoFactory: Design Patterns in C#. Ukázkové implementace všech GoF vzorů. http://dofactory.com/Patterns/Patterns.aspx

[DSL Tools] Microsoft: DSL Tools homepage. http://msdn.microsoft.com/-vstudio/DSLTools/

[Eden‘s publications] A. H. Eden – publikace. http://www.eden-study.org/-publications.html

[GA.net] Microsoft: GuidanceAutomation.net. http://guidanceautomation.net/

[Introduction to GAT] Microsoft Corporation: *Introduction to the Guidance Automation Toolkit*. http://msdn.microsoft.com/vstudio/teamsystem/-Workshop/gat/intro.aspx

[LayOM Compiler] The LayOM Complier. http://www.ipd.bth.se/rise/LayOM/

[LayOM Specification] LayOM Language Specification. http://www.ipd.bth.se/rise/-LayOM/LangSpec.html

[MDA] OMG Model Driven Architecture. http://www.omg.org/mda/

[MDA-wiki] Wikipedia: Mode-driven architecture. http://en.wikipedia.org/-wiki/Model-driven_architecture

[Micorosft Modeling Strategy] Microsoft Corporation: Visual Studio 2005 Team System Modeling Strategy and FAQ (květen 2005). http://msdn.microsoft.com/-library/default.asp?url=/library/en-us/dnvs05/html/-vstsmodel.asp

[NCEK] Infragristics: NetAdvantage CAB Extensibility Kit. http://www.infragistics.com/resource/Technologies/CAB/-Default.aspx

[Object Builder] Microsoft patterns & practices: *ObjectBuilder Dependency Injection Framework*. http://www.codeplex.com/Wiki/-View.aspx?ProjectName=ObjectBuilder

[OMG] The Object Management Group (OMG), http://omg.org/

[Pattern Definition] Coplien, J. O., Gabriel, D.: A Pattern Definition. http://hillside.net/-patterns/definition.html

[SCSF] Microsoft patterns & practices: *Smart Client Software Factory* (červenec 2006). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/scbatlp.asp

[Software Product Lines] Software Product Lines, http://www.softwareproductlines.com/

[UML 2.0 Superstructure] OMG: Unified Modeling Language: Superstructure, version 2.0. OMG, 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04

[VS SDK]

[WSSF webcast] SMITH, Don: *MSDN Architecture Webcast: Introducing the Web Service Software Factory*. Webcast dostupný z adresy http://msevents.microsoft.com/CUI/-WebCastEventDetails.aspx?EventID=1032300719&EventCategory=5&culture=en-US&CountryCode=US

Terminologický slovník

Termín	Anglický originál	Popis
Termíny přejaté z terminologického slovníku katedry informačních technologií		
CASE (Computer Aided Software / System Engineering)	CASE (Computer Aided Software / System Engineering)	Komplex počítačových nástrojů pro podporu analýzy, návrhu a implementace informačních systémů a informačních a komunikačních technologií, ale i dalších činností, souvisejících s vývojem IS/ICT. Nástroje typu CASE umožňují zachycení modelů zkoumaného světa ideálně na třech úrovních – konceptuální, technologické a implementační. Dalšími výstupy CASE systémů jsou dokumentace IS/ICT a programové kódy a skripty pro definici obsahu datové základny
UML (Unified Modeling Language)	UML (Unified Modeling Language)	Grafická notace pro zápis modelů informačních systémů, shrnuje a standardizuje dosavadní notace a je východiskem pro vznik nových. Je určen pro zápis jednotlivých pohledů na systém v objektově orientovaných metodikách, používá se například v CASE určených pro podporu tvorby informačních systémů, jeho použití však může být širší (reorganizace podniku).
Další termíny		
aplikační block	application block	Sada dokumentů, referenčních implementací a znovupoužitelného kódu, která pomáhá s určitým aspektem vývoje nebo s určitým typem aplikací. Příkladem jsou aplikační bloky pro logování, kešování, přístup k datům a podobně.
atribut	attribute	Dodatečná informace, kterou lze připojit k elementu zdrojového kódu a která může být za běhu získána díky mechanismu reflexe. Platforma .NET obsahuje vestavěnou podporu pro atributy od verze 1.
balíček návodů	guidance package	Balíček vytvořený nástrojem GAT, který lze nainstalovat do Visual Studia 2005 a který pomáhá automatizovat některé vývojářské činnosti.
Class Designer	Class Designer	Vizuální návrhář standardně dodávaný s Visual Studiem 2005 od edice Standard. Slouží k manipulaci s ⇨Class Diagramy.
Class Diagram	Class Diagram	Obecně se jedná o diagram tříd, známý např. z notace UML, pokud však v práci byl použit pojem „Class Diagram“ s velkými písmeny, byl míněn diagram vytvářený ⇨Class Designerem, který slouží k vizualizaci kód v jazyce C# nebo Visual Basic .NET.
Composite UI Application Block (CAB)	Composite UI Application Block (CAB)	⇨Aplikační blok pro vývoj desktopových aplikací sestavených z nezávislých, ale spolupracujících modulů („pluginů“).
doména	domain	Určitá oblast zájmu. Práce pojímá tento termín úžeji a zaměřuje se pouze na domény související se softwaro-

		vých inženýrstvím. Viz též ⇨horizontální doména a ⇨vertikální doména.
Doménově specifický jazyk	Domain-Specific Language (DSL)	Jazyk, ať už textový nebo vizuální, který je zaměřený na jednu konkrétní ⇨doménu.
Guidance Automation Extensions (GAX)	Guidance Automation Extensions (GAX)	Běhové prostředí pro ⇨balíčky návodů a ⇨GAT v rámci Visual Studia 2005.
Guidance Automation Toolkit (GAT)	Guidance Automation Toolkit (GAT)	Rozšíření Visual Studia 2005, které architektům umožňuje vytvářet a upravovat ⇨balíčky návodů.
horizontální doména	horizontal domain	Taková ⇨doména, která je zaměřena na jeden aspekt softwarového vývoje, např. na vztah tříd nebo na dotazování databází. Viz též ⇨vertikální doména.
MDA (Model Driven Architecture)	MDA (Model Driven Architecture)	Standard OMG pro vývoj softwarových systémů založený na modelování. Odděluje aplikační logiku od implementačních technologií. Ústřední roli v MDA hraje ⇨UML.
návrhový vzor	design pattern	⇨Vzor zaměřený na oblast návrhu softwarového systému.
Smart Client Software Factory (SCSF)	Smart Client Software Factory (SCSF)	⇨Softwarová továrna podporující vývoj desktopových aplikací založených na ⇨Composite UI Application Blocku.
softwarová továrna	software factory	Vývojové prostředí, které automatizuje některé vývojové aktivity a pomáhá s vývojem určitého typu aplikace.
vertikální doména	vertical domain	Taková ⇨doména, která je zaměřena na určitou oblast reálného světa, např. na zdravotnictví nebo na finance. Viz též ⇨horizontální doména.
vzor	pattern	Popisuje opakující se problém a jeho obecné řešení, které může být přizpůsobeno konkrétní situaci.

Seznam obrázků a tabulek

Obrázek 1 – Zaměření práce ... 1
Obrázek 2 – Svět vzorů (zdroj: [Stein, 2004]) ... 5
Obrázek 3 – Ukázka syntaxe jazyka *Z* ... 10
Obrázek 4 – Princip LayOM – obalení objektu vrstvami (zdroj: [Bosch, 1998]) ... 11
Obrázek 5 – Příklad návrhového vzoru Visitor (zdroj: [Eden, 03]) ... 15
Obrázek 6 – Metamodel PDL (zdroj: [Albin-Amiot, Guéhéneuc, 2001]) ... 17
Obrázek 7 – Princip fungování PDL (zdroj: [Albin-Amiot, Guéhéneuc, 2001]) ... 17
Obrázek 8 – Diagramy omezení (zdroj: [Lauder, Kent, 1998]) ... 19
Obrázek 9 – Symbol třídy rozšířený o oddíl pro abstraktní instance (zdroj: [Lauder, Kent, 1998]) ... 19
Obrázek 10 – Vztah "každá publikace má svůj titul" (zdroj: [Lauder, Kent, 1998]) ... 19
Obrázek 11 – Abstract Factory jako model rolí (zdroj: [Lauder, Kent, 1998]) ... 20
Obrázek 12 – Abstract Factory jako typový model (zdroj: [Lauder, Kent, 1998]) ... 20
Obrázek 13 – Převedení typového modelu do modelu tříd (zdroj: [Lauder, Kent, 1998]) ... 20
Obrázek 14 – Zachycení vzorů v UML pomocí Vennových diagramů (zdroj: [Dong, 2002]) ... 22
Obrázek 15 – Zachycení vzoru pomocí notace pro kolaboraci (zdroj: [Dong02]) ... 23
Obrázek 16 – Příklad komplexnosti diagramu při použití kolaborací (zdroj: [Yacoub, Ammar, 2003]) ... 23
Obrázek 17 – Anotace "vzor:role" (zdroj: [Dong, 2002]) ... 24
Obrázek 18 – Anotace pomocí tagovaných hodnot (zdroj: [Dong, 2002]) ... 25
Obrázek 19 – Vložení návrhového vzoru v nástroji Enterprise Architect ... 27
Obrázek 20 – Výsledek vložení abstraktní továrny ... 28
Obrázek 21 – Nastavení singletonu tak, aby ovlivnil již existující prvek diagramu ... 28
Obrázek 22 – Výsledek aplikování vzoru Singleton ... 29
Obrázek 23 – Mechanické aplikování vzoru Factory Method ... 29
Obrázek 24 – Vložení návrhového vzoru Abstract Factory ... 30
Obrázek 25 – Aplikování vzoru singleton na konkrétní továrnu ... 31
Obrázek 26 – Výsledek aplikování vzoru Singleton ... 31
Obrázek 27 – Po aplikování vzoru Factory Method ... 32
Obrázek 28 – OMG Model-driven Architecture (zdroj: [MDA]) ... 34
Obrázek 29 – Schéma softwarové továrny (zdroj: [Greenfield, 2004]) ... 39
Obrázek 30 – Příklad doménově specifického jazyka ve Visual Studiu (zdroj: [Regio, Greenfield, 2005]) ... 42
Obrázek 31 – Společné prvky všech doménově specifických jazyků (zdroj: [Seemann, 2005]) ... 43
Obrázek 32 – Microsoft Modeling Platform (zdroj: [Seemann, 2005]) ... 43
Obrázek 33 – Vztah GAX, GAT a Visual Studia (zdroj: dokumentace k [SCSF]) ... 44
Obrázek 34 – Koncept chytrých klientů (zdroj: [Building Smart Clients]) ... 46
Obrázek 35 – Schematické znázornění CAB (zdroj: dokumentace k [SCSF]) ... 47
Obrázek 36 – Využití návrhových vzorů v CAB (zdroj: dokumentace k [CAB]) ... 48
Obrázek 37 – Příklad aplikace vytvořené pomocí CAB ... 48
Obrázek 38 – SCSF položky v menu Start ... 49
Obrázek 39 – Nová položka v dialogu pro přidání projektu ... 50
Obrázek 40 – Softwarová továrna pro vývoj klientských aplikací ... 50
Obrázek 41 – V „softwarové továrně“ zůstávají k dispozici všechny vlastnosti Visual Studia ... 51
Obrázek 42 – Interakce softwarové továrny s vývojářem ... 51
Obrázek 43 – Struktura EFx aplikačního frameworku (zdroj: [Santos, 2006]) ... 52
Obrázek 44 – Vizuální návrhář Enterprise Framework Factory (zdroj: [Santos, 2006]) ... 53
Obrázek 45 – Příklad notace používané k zachycení vzorů v dokumentaci SCSF ... 57
Obrázek 46 – Doménově specifický jazyk pro návrhové vzory Clipcode-GoF-DSL ... 58
Obrázek 47 – Doménový model Clipcode-GoF-DSL ... 59

Obrázek 48 – Zobrazení vztahu v DSL Tools „v1“ 60
Obrázek 49 – Koncept vývojového prostředí pro návrhové vzory 62
Obrázek 50 – Vizualizace kódu implementujícího vzor Abstract Factory 64
Obrázek 51 – Zápis vztahu závislosti pomocí přerušované čáry v notaci OMT (zdroj: [GoF, 1994]) 65
Obrázek 52 – Abstract Factory implementovaná pomocí rozhraní 65
Obrázek 53 – Prototyp doménově specifického jazyka pro návrhové vzory 66
Tabulka 1 – Popis návrhových vzorů a jejich přibližné mapování 4
Tabulka 2 – výhody a nevýhody LayOM 13
Tabulka 3 – výhody a nevýhody atributů při popisu návrhových vzorů na úrovni zdrojového kódu 14
Tabulka 4 – výhody a nevýhody metody DisCo 15
Tabulka 5 – výhody a nevýhodu LePUSu 16
Tabulka 6 – Výhody a nevýhody PDL 19
Tabulka 7 – Výhody a nevýhody diagramů omezení 21

Příloha A – korespondence ohledně LayOM

From: Borek Bernard
To: Jan Bosch
Subject: Design Patterns and LayOM question

Dear Mr. Bosch,

I'm a student from the Czech Republic and I am currently writing a diploma thesis on design patterns, their formalization etc. I recently came across your paper called "Design Patterns as Language Constructs" and since then, I really want to ask you one question: is there any fundamental difference between LayOM and, say, UML in the relation to design patterns?

I mean that when you need some really general language to describe design patterns in their essence, neither LayOM nor UML is the option. When you need some language for final implementation, neither LayOM nor UML is the option. Both LayOM and UML are too abstract to be considered final implementation languages but too specific to describe design patterns in their essence.

So why did you create (at least, I believe you're the author of LayOM) another "middle-range" notation beside UML? Do you think that it has better expressiveness than UML (although I think that UML is pretty good at what it's doing)? Is there some point I'm missing?

I would be very graceful if you could send me a short answer.

Best regards,

Borek Bernard

From: Jan Bosch
To: Borek Bernard
Subject: RE: Design Patterns and LayOM question

Dear Bernard,

Thank you for your email. I appreciate your assessment of LayOM in the context of describing design patterns. Although I think your comments are fair, although I disagree on several points, I think the main topic that is missing is the time dimension. LayOM was primarily developed during the period 1994-1998 whereas UML became really successful in the latter period.

What I would like to draw your attention to is the more recent work on architectural design decisions that basically uses architectural styles, architectural patterns, design patterns and other structures as the basic building blocks to build architectures from. Anton Jansen is active in this area and he can send you some papers if you're interested. Finally, I have mostly left the academic world and am working these days for Nokia Research Center. However, Anton and several others stay active in the field and can help you if you have more questions.

Thanks,

Jan

From: Anton Jansen
To: Borek Bernard
Subject: RE: Design Patterns and LayOM question

Hi Bernard, You were wondering about the relationship between design patterns and LayOM and Jan pointed you to me for my work on architectural decisions, which is related to design patterns. I pondered a long time about design patterns and the ideas used within LayOM and the formalization of design patterns in general. In my research I came to the conclusion that the design patterns to a couple of interresting points:

- Design patterns are ALWAYS adapted when they are implemented to fit the specific context. The text book descriptions of the design patterns (e.g. gang of four) define sometimes tactics for these adaptations, but are nowhere and cannot be complete.

- Current implementation languages indeed lack a first class representation of design patterns. It is not explicit that a piece of code is due to one or more design patterns.

- The reasons why a design patterns are used in a design is often not clear either. Most patterns have often multiple benefits and drawbacks. However, it is often unclear which benefits are important to the architect/designer (and with which drawbacks he/she can live).

I'm currently researching the concept of architectural design decisions, which underly a design. Design patterns in this concept are collections of design decisions, which are "prepackaged" and are (widely) known and documented. With the help of additional design decisions, they can be adapted to fit the design in question. For more information about architectural design decisions and how they relate to design patters, please read the attached paper, which was published on WICSA last year.

With kind regards,

Anton Jansen

Příloha B – Výpis kódu pro definici uživatelského atributu

```
1  using System;
2
3  namespace PatternAttributes
4  {
5    [AttributeUsage(
6      AttributeTargets.Class |
7      AttributeTargets.Event |
8      AttributeTargets.Field |
9      AttributeTargets.Interface |
10     AttributeTargets.Method |
11     AttributeTargets.Property |
12     AttributeTargets.Struct,
13   AllowMultiple=true, Inherited=false)]
14   public sealed class PatternAttribute : Attribute
15   {
16     // Name and Role are positional arguments (initiated using constructor)
17     private string _Name;
18
19     /// <summary>
20     /// Gets the pattern name, e.g. "Abstract Factory".
21     /// </summary>
22     public string Name
23     {
24       get { return _Name; }
25     }
26
27
28     private string _Role;
29     /// <summary>
30     /// Gets the role name inside the pattern, e.g. "Concrete Product".
31     /// </summary>
32     public string Role
33     {
34       get { return _Role; }
35     }
36
37     // PatternId is optional named argument (used as 'PatternId=1234')
38     private string _PatternId;
39     /// <summary>
40     /// Gets or sets the pattern id. Should be unique per project.
41     /// </summary>
42     public string PatternId
43     {
44       get { return _PatternId; }
45       set { _PatternId = value; }
46     }
47
48     private string _Notes;
49     /// <summary>
50     /// Gets or sets additional notes to this pattern element.
51     /// </summary>
52     public string Notes
53     {
54       get { return _Notes; }
55       set { _Notes = value; }
56     }
57
58     /// <summary>
59     /// Annotates target with a pattern-role value pair.
60     /// </summary>
61     /// <param name="name">Pattern name, e.g. "Abstract Factory"</param>
62     /// <param name="role">Role name inside the pattern, e.g. "Concrete Product"</param>
63     public PatternAttribute(string name, string role)
64     {
65       _Name = name;
66       _Role = role;
67     }
68   }
69 }
```

Příloha C – Výpis kódu abstraktní továrny realizované pomocí abstraktních tříd

```
1  using System;
2  using PatternAttributes;
3
4  namespace AbstractFactoryClasses
5  {
6    class AbstractClassesImplementation
7    {
8      static void Main(string[] args)
9      {
10       Client client = new Client(new MotifWidgetFactory());
11       client.DoAction();
12
13       client = new Client(new PMWidgetFactory());
14       client.DoAction();
15     }
16   }
17
18   [Pattern("Abstract Factory", "Abstract Product")]
19   public abstract class Window
20   {
21     public abstract void Show();
22   }
23
24   [Pattern("Abstract Factory", "Abstract Product")]
25   public abstract class ScrollBar
26   {
27     public abstract void Scroll();
28   }
29
30   [Pattern("Abstract Factory", "Concrete Product")]
31   public class PMWindow : Window
32   {
33     public override void Show()
34     {
35       Console.WriteLine("I'm a PM window");
36     }
37   }
38
39   [Pattern("Abstract Factory", "Concrete Factory")]
40   public class PMWidgetFactory : WidgetFactory
41   {
42     public override ScrollBar CreateScrollBar()
43     {
44       return new PMScrollBar();
45     }
46
47     public override Window CreateWindow()
48     {
49       return new PMWindow();
50     }
51   }
52
53   [Pattern("Abstract Factory", "Concrete Product")]
54   public class PMScrollBar : ScrollBar
55   {
56     public override void Scroll()
57     {
58       Console.WriteLine("I'm a PM scrollbar");
59     }
60   }
61
62   [Pattern("Abstract Factory", "Concrete Product")]
63   public class MotifWindow : Window
64   {
65     public override void Show()
66     {
67       Console.WriteLine("I'm a Motif window");
68     }
69   }
```

```
70
71    [Pattern("Abstract Factory", "Concrete Factory")]
72    public class MotifWidgetFactory : WidgetFactory
73    {
74      public override ScrollBar CreateScrollBar()
75      {
76        return new MotifScrollBar();
77      }
78
79      public override Window CreateWindow()
80      {
81        return new MotifWindow();
82      }
83    }
84
85    [Pattern("Abstract Factory", "Concrete Product")]
86    public class MotifScrollBar : ScrollBar
87    {
88      public override void Scroll()
89      {
90        Console.WriteLine("I'm a Motif scrollbar");
91      }
92    }
93
94    [Pattern("Abstract Factory", "Client")]
95    public class Client
96    {
97
98      private WidgetFactory _WidgetFactory;
99
100     public WidgetFactory WidgetFactory
101     {
102       get
103       {
104         return _WidgetFactory;
105       }
106       set
107       {
108         _WidgetFactory = value;
109       }
110     }
111
112     public Client(WidgetFactory factory)
113     {
114       _WidgetFactory = factory;
115     }
116
117     public void DoAction()
118     {
119       Window myWindow = _WidgetFactory.CreateWindow();
120       ScrollBar myScrollBar = _WidgetFactory.CreateScrollBar();
121       myWindow.Show();
122       myScrollBar.Scroll();
123     }
124   }
125
126   [Pattern("Abstract Factory", "Abstract Factory")]
127   public abstract class WidgetFactory
128   {
129     public abstract ScrollBar CreateScrollBar();
130
131     public abstract Window CreateWindow();
132   }
133 }
```

Příloha D – Výpis kódu abstraktní továrny realizované pomocí rozhraní

```
1  using System;
2  using PatternAttributes;
3
4  namespace AbstractFactoryInterfaces
5  {
6    class InterfacesImplementation
7    {
8      static void Main(string[] args)
9      {
10       Client client = new Client(new MotifWidgetFactory());
11       client.DoAction();
12
13       client = new Client(new PMWidgetFactory());
14       client.DoAction();
15     }
16   }
17
18   [Pattern("Abstract Factory", "Abstract Factory")]
19   public interface IWidgetFactory
20   {
21     AbstractFactoryInterfaces.IScrollBar CreateScrollBar();
22     AbstractFactoryInterfaces.IWindow CreateWindow();
23   }
24
25   [Pattern("Abstract Factory", "Abstract Product")]
26   public interface IScrollBar
27   {
28     void Scroll();
29   }
30
31   [Pattern("Abstract Factory", "Abstract Product")]
32   public interface IWindow
33   {
34     void Show();
35   }
36
37   [Pattern("Abstract Factory", "Concrete Product")]
38   public class PMWindow : IWindow
39   {
40     public void Show()
41     {
42       Console.WriteLine("I'm a PM window");
43     }
44   }
45
46   [Pattern("Abstract Factory", "Concrete Factory")]
47   public class PMWidgetFactory : IWidgetFactory
48   {
49     public AbstractFactoryInterfaces.IScrollBar CreateScrollBar()
50     {
51       return new PMScrollBar();
52     }
53
54     public AbstractFactoryInterfaces.IWindow CreateWindow()
55     {
56       return new PMWindow();
57     }
58   }
59
60   [Pattern("Abstract Factory", "Concrete Product")]
61   public class PMScrollBar : IScrollBar
62   {
63     public void Scroll()
64     {
65       Console.WriteLine("I'm a PM scrollbar");
66     }
67   }
68
69   [Pattern("Abstract Factory", "Concrete Product")]
```

```
70   public class MotifWindow : IWindow
71   {
72     public void Show()
73     {
74       Console.WriteLine("I'm a Motif window");
75     }
76   }
77
78   [Pattern("Abstract Factory", "Concrete Factory")]
79   public class MotifWidgetFactory : IWidgetFactory
80   {
81     public AbstractFactoryInterfaces.IScrollBar CreateScrollBar()
82     {
83       return new MotifScrollBar();
84     }
85
86     public AbstractFactoryInterfaces.IWindow CreateWindow()
87     {
88       return new MotifWindow();
89     }
90   }
91
92   [Pattern("Abstract Factory", "Concrete Product")]
93   public class MotifScrollBar : IScrollBar
94   {
95     public void Scroll()
96     {
97       Console.WriteLine("I'm a Motif scrollbar");
98     }
99   }
100
101  [Pattern("Abstract Factory", "Client")]
102  public class Client
103  {
104
105    private IWidgetFactory _WidgetFactory;
106
107    public IWidgetFactory WidgetFactory
108    {
109      get
110      {
111        return _WidgetFactory;
112      }
113      set
114      {
115        _WidgetFactory = value;
116      }
117    }
118
119    public Client(IWidgetFactory factory)
120    {
121      _WidgetFactory = factory;
122    }
123
124    public void DoAction()
125    {
126      IWindow myWindow = _WidgetFactory.CreateWindow();
127      IScrollBar myScrollBar = _WidgetFactory.CreateScrollBar();
128      myWindow.Show();
129      myScrollBar.Scroll();
130    }
131  }
132 }
```