Modelom riadený vývoj softvéru

BAKALÁRSKA PRÁCA

Gabriel Ščerbák

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

9.2.1 Informatika

Vedúci bakalárskej práce:
Ing. Peter Grec

Bratislava 2009

Abstrakt

V tejto práci sa zaoberám technikami modelom riadeného vývoja softvéru a možnosťami ich využitia. Najprv sa pozrieme na motiváciu, základné definície a princípy, ale i vývoj týchto techník. Ďalej sa budeme podrobnejšie zaoberať modelmi a modelovaním tak ako sú chápené vrámci modelom riadeného vývoja softvéru. Statické modely ako také by slúžili len ako dokumentácia a preto, aby sme mohli využiť ich potenciál, budeme sa zaoberať ich transformáciami. Dnes často skloňovaným pojmom sú doménovo špecifické jazyky. Vysvetlíme si aké je ich miesto vrámci modelov a akú hodnotu nám môžu priniesť. Skutočnú hodnotu vrámci vývoja a teda zdrojový kód nám umožňujú vytvárať techniky generatívneho programovania, ktoré priblížim v samostatnej kapitole. Aspektovo-orientované programovanie je paradigma, ktorá je pre modelom riadený vývoj softvéru prirodzená a tak si vysvetlíme jej princípy a možnosti implementácie pomocou modelov. S rastom softvérových produktov súvisí i potreba správy ich rodín. Touto tematikou sa zaoberá inžinierstvo produktových línií, ktorého princípy sú vhodne realizovateľné pomocou modelom riadeného vývoja softvéru. Napokon sa zameriame na najlepšie postupy a vzory z tejto oblasti, ktoré boli publikované a neopomenieme ani otvorené problémy.

Kľúčové slová: **softvérové inžinierstvo**, **modelom riadený vývoj softvéru**, **doménovo špecifické jazyky**, **aspektovo orientované programovanie**, **inžinierstvo produktových línií**

Čestné prehlásenie

Vyhlasujem, že som bakalársku prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Bratislava 12. júna 2009 .....

Vlastnoručný podpis

Poďakovanie

Ďakujem týmto vedúcemu svojej bakalárskej práce, ktorý bol vždy ochotný pomôcť, poradiť a prispieť cennými radami a vlastnými skúsenosťami. Taktiež mu ďakujem i zato, že napriek svojim pracovným povinnostiam neváhal venovať svoj čas spoločným konzultáciám.

Obsah

Úvod

Informačné systémy nás obklopujú takmer vo všetkých oblastiach nášho života a priamo či nepriamo od nich závisia majetok, financie ale aj zdravie a životy ľudí. To kladie značnú mieru zodpovednosti na vývojárov takýchto systémov. Zároveň sa na softvér kladú neustále vyššie nároky, vyvíjajú sa stále komplexnejšie systémy a naviac sa skracujú časy vývoja pričom kvôli zvyšovaniu konkurencieschopnosti cena vývoja neustále klesá. Čím sú systémy a technológie zložitejšie, tým sú väčšie i nároky kladené na vývojárov a tým stúpa i cena ich práce a je ťažšie vyškoliť nových odborníkov. Toto sú sily, ktoré pôsobia proti sebe a požadujú od softvéru kvalitu za dosť obmedzujúcich podmienok.

V minulosti pokrok prinieslo zvýšenie úrovne abstrakcie na ktorej pracovali vývojári. Najprv to bol prechod od strojového kódu k assembleru, neskôr k procedurálnemu programovaniu, začiatkom deväťdesiatych rokov sa rozšírilo objektovo-orientované programovanie, napokon dnes možno považovať za aktuálny stav aplikačné frameworky. Ďalším krokom vyššie po tomto akoby schodisku, kde každý stupienok prináša náhle zvýšenie efektivity vývoja, no zároveň najbližšie roky postupné ustálenie, ktorý je ako sa ja i mnohí ďalší domnievajú modelom riadený vývoj softvéru.

Mojim cieľom vrámci tejto bakalárskej práce je zmapovať oblasť modelom riadeného vývoja softvéru, príslušné techniky, praktické skúsenosti a najnovšie trendy v tejto oblasti. Nadobudnuté vedomosti budem konfrontovať a konzultovať s vedúcim práce a jeho vlastnými skúsenosťami v tejto oblasti.

Kapitola 1

Modelom riadený vývoj softvéru

Tento spôsob vývoja využíva modely, ktoré sú zachytávané pomocou doménovo špecifických jazykov. Z nich sa prostredníctvom série transformácií generuje zdrojový kód aplikácie. Modely umožňujú pracovať na rôznych úrovniach abstrakcie a tak dávajú možnosť odpútať sa od konceptov všeobecného programovacieho jazyka. Reprezentované môžu byť rôznymi spôsobmi, v podobe textových doménovo špecifických jazykov, graficky napríklad v podobe diagramov jazyka UML alebo tabuľkovo či kombináciou týchto spôsobov. Vrámci modelovania je možné využiť techniky aspektovo-orientovaného programovania a oddeliť tak od seba rôzne technické či biznis domény. Transformácie umožňujú skryť zanedbateľné detaily implementácie, programovacieho jazyka alebo frameworku či prevádzať na modeloch rôzne optimalizácie. Modelovacie nástroje, editory s príjemným užívateľským prostredím zasa môžu priblížiť vývoj aplikácií doménovým špecialistom.

Obzvlášť priblíženie vývojových nástrojov problémovej doméne daného projektu je veľmi prínosné. Umožňuje to totiž zachytávať vedomosti a požiadavky doménových expertov v jazyku, ktorému títo odborníci rozumejú a teda aj pracovať s konceptami, ktoré sú im blízke pre nich prirodzeným spôsobom. Zároveň však takto dochádza k formalizácií týchto konceptov, vedomostí a aj požiadaviek čo umožní zjednodušiť prácu vývojárov, ktorí v spolupráci s doménovými expertmi budú môcť zanedbať platformové detaily, o ktoré sa postarajú nástroje, ktoré je možné vyvíjať oddelene.

1.1 História

Modelom riadený vývoj softvéru nie je myšlienkou, ktorá prišla z čista jasna, predchádzali jej mnohé iné, ktorých vývojom a kombináciou sa táto technika rozvinula do dnešného stavu, no vývoj sa tuto nezastavuje a táto oblasť sa neustále rozvíja.

1.1.1 CASE

Za prvého predchodcu by som označil nástroje CASE (Computer Aided Software Engineering) [1]. Tie často umožňovali pomocou diagramov, napríklad v jazyku UML, namodelovať systém a potom z modelov vygenerovať štruktúru aplikácie. Samozrejme táto funkčnosť nebola samozrejmosťou a častokrát bola veľmi obmedzená, trebárs bolo možné vytvárať z diagramov tried súborovú štruktúru s hlavičkami s názvom triedy. Aj tieto nástroje sa však rozvíjali a neskôr umožnili zadefinovať i šablóny, podľa ktorých sa generoval obsah súborov. Keďže podobnosť týchto postupov so základnými myšlienkami modelom riadeného vývoja softvéru je zrejmá, v počiatkoch zvykla byť táto technika posmešne označovaná ako CASE 2.0 . Toto označenie je možné považovať do istej miery za pravdivé, lebo minimálne pre dodávateľov CASE nástrojov bolo vydanie štandardu MDA, viac ďalej, akýmsi míľnikom a svoje nástroje často prispôsobili, aby viac či menej tomuto štandardu alebo nejakým jeho častiam vyhovovali.

1.1.2 Spustiteľné UML

Myšlienka spustiteľného UML (Executable UML) [MSUW04] spočívala v možnostiach využiť model systému, ktorý by mal byť presný na overovanie vlastností systému a následne jeho priamy preklad do binárnej podoby. Tento postup však naráža na viacero problémov. Prvým z nich je použitie samotného UML, ktoré ako sa ukázalo je sémanticky nejednoznačné čo je neprípustné. Riešením má byť teda akási podmnožina UML, teda lepšie povedané nejaký UML profil, ktorý je sémanticky jednoznačný a na opis systému bude postačujúci. Návrh na tento profil je už dlhšiu

[1] `http://en.wikipedia.org/w/index.php?title=Computer-aided_software_engineering&oldid=282740868`

dobu v procese štandardizácie [2], ale prednedávnom sa dočkal i prvej implementácie pod názvom fUML (Foundational UML) [3]. Ďalším problémom je fakt, že UML, hoci ako sa ukázalo je obzvlášť vhodné na opis štruktúry, nie je najvhodnejšie na opis akčnej sémantiky. Tento problém rieši táto technika zavedením nejakého vlastného jazyka akčnej sémantiky, no týchto jazykov je viacero, každý autor používa v podstate vlastný. Okrem týchto problémov sa spustiteľné UML stretáva aj s kritikou či takéto obmedzené UML bude dostatočne výrazové a či takýto postup nebude v podstate rovnako komplikovaný ako implementácia v niektorom zo všeobecných programovacích jazykov.

1.1.3 MDA

MDA je skratka označujúca štandard Model Driven Architecture (Modelom riadená architektúra), ktorý priajalo konzorcium OMG v roku 2001. Celý tento postup je publikovaný [4] a bolo o ňom napísaných niekoľko kníh, napríklad [KWB03, MSUW04].

OMG (Object Managment Group) je konzorcium firiem, ktoré si dáva za cieľ interoperabilitu a v minulosti sa to podarilo pomocou štandardov CORBA a UML. MDA mal byť ďalším takýmto štandardom, ktorý by sa zameriaval na prístup využivajúci modely, ktoré sa následne transformujú do kódu.

Špecifikom MDA je prístup vrámci ktorého sa vytvárajú modely na viacerých úrovniach abstrakcie. Na najvyššej úrovni je CIM (Computer independent Model), ktorý opisuje systém zvonku. Ďalšou úrovňou je PIM (Platform Independent Model), kde sú modely, ktoré opisujú systém všeobecne bez ohľadu na technologické detaily. Ďalšou vrstvou je PSM (Platform Specific Model), ktorý už obsahuje i konkrétne detaily technológie. Na poslednej úrovni je samotný zdrojový kód. Veľmi špecifický je tu opis prechodu medzi jednotlivými úrovňami, kedy sa odporúča k modelu vytvoriť takzvaný MM (Marking Model), teda označkovaný model, ktorý v sebe obsahuje detaily potrebné pre transformáciu do nižšej úrovne a priraďuje ich jednotlivým prvkom modelu na vyššej úrovni. Transformácia má teda k dispo-

[2] `http://www.omg.org/docs/ad/06-06-16.pdf`
[3] `http://portal.modeldriven.org/content/foundational-uml-reference-implementation`
[4] `http://www.omg.org/docs/omg/03-06-01.pdf`

zicií model a značky, z ktorých sa vytvára nový model. Ďalším špecifikom je dôraz na elaboráciu modelu, čo znamená, že sa odporúča potrebné časti do modelu či zdrojového kódu dorábať priamo ručne. Napokon treba spomenúť silné previazanie MDA so štandardom UML a taktiež sériou ďalších štandardov zadefinovaných konzorciom OMG.

Tento prístup bol prvým štandardom a preto je často považovaný za referenčný. Mnohé štandardy, ktoré využíva sú pevným základom modelom riadeného vývoja softvéru ako takého. V prvom rade MDA kladie veľký dôraz na UML (Unified Modeling Language) [Fow04] ako meta model čo znamená, že sa orientuje prevažne na grafické doménovo špecifické jazyky, hoci treba poznamenať, že vrámci MDA sa termín doménovo špecifického jazyka nepoužíva. Meta modely sa v MDA prispôsobujú pomocou mechanizmu profilov v UML. Profil pozostáva z označených hodnôt (tagged values), stereotypov a obmedzení (constraints) v jazyku OCL (Object Constraint Language) [AN05]. Transformácie umožňuje jazyk QVT (Query Views Transformations). OMG vrámci meta úrovní neskôr zašlo ešte ďalej a zadefinovalo meta meta model v podobe štandardu MOF (MetaObject Facility) [5]. MOF vo verzii 2.0 bol dôležitým miľníkom, lebo rozdelením na dva podštandardy SMOF (Standard MOF) a EMOF (Essential MOF) umožnil veľkej rodine nástrojov na prácu s modelmi založenej na technológií EMF (Eclipse Modeling Framework) [6] a meta modeli Ecore zaradiť sa medzi štandardné. V rámci štandardizácie vzniklo i mnoho hotových UML profilov ako napríklad SysML, CWM, UML4CORBA, EDOC, MARTE a mnohé iné [7]. Okrem toho bol zavedený aj štandard pre prenos modelov XMI (XML Metadata Interchange) [8] a taktiež i diagramov DI (Diagram Interchange) [9]. Za zmienku hádam ešte stojí i štandard HUTN (Human Usable Textual Notation) [10], ktorý vznikol v čase, keď sa začali ukazovať výhody textových doménovo špecifických jazykov. Okrem týchto OMG prišlo i s množstvom ďalších štandardov týkajúcich sa MDA.

[5] `http://www.omg.org/mof/`
[6] `http://www.eclipse.org/modeling/emf/`
[7] `http://www.omg.org/technology/documents/profile_catalog.htm`
[8] `http://www.omg.org/technology/documents/formal/xmi.htm`
[9] `http://www.omg.org/technology/documents/formal/diagram.htm`
[10] `http://www.omg.org/technology/documents/formal/hutn.htm`

MDA sa však veľmi ako štandard neuchytilo a to z viacerých dôvodov, no najzávažnejším bol asi fakt, že proces štandardizácie bol veľmi rýchly a ako je zrejme z množnostva štandardov aj rozsiahly. Mnohé štandardy dlho nemali svoju implementáciu ako napríklad QVT, ktoré je i dodnes implementované len čiastočne, iné zasa prešli mnohými revíziami a množstvo existujúcich verzií interoperabilite nenapomáha, napríklad v prípade XMI no a niektoré sa ukázali ako zbytočné, respektíve neužitočné ako napríklad HUTN, ktoré neumožňuje prispôsobenie syntaxe a teda namiesto textového doménovo špecifického jazyka ide ozaj len o notáciu namiesto UML. Ine štandardy si však našli svoje miesto. Napríklad UML nie je závislé od MDA a ako štandard pre notáciu modelov objektovo orientovaných systémov sa neustále používa. MOF zasa zmenami, ktoré umožnili zaradiť Ecore do štandardu EMOF si vyslúžil medzi nástrojmi založenými na EMF trvalú podporu. Mnohé UML profily sa osvedčili v praxi a sú využívané i keď nie vždy v kontexte MDA, ale skôr na modelovanie pomocou UML. Nakoniec OCL možno nájsť v nástroji oAW (openArchitectureWare) [11] v upravenej podobe pod názvom Check. Okrem problémov so štandardami MDA zlyháva i navrhnutými praktikami. Rozdelenie modelov na PIM a PSM je dosť vágne a ťažko povedať, ktoré kam patria. Označkovanie modelov a ich dorábanie spôsobuje nemalé problémy so správou verzií, ktoré je len veľmi ťažko možné odstrániť, viac o tejto problematike uvediem v kapitole o transformáciách. Monopol UML je taktiež problémom, lebo špecifikácia tochto jazyka je dosť zložitá a je pre dodávateľov problematické vytvárať nástroje, ktoré by ju kompletne spĺňali. V neposlednom rade sa ukázalo, že v mnohých prípadoch je využitie textových doménovo špecifických jazykov pohodolnejšie než grafická notácia pomocou UML.

1.1.4 MDSD

MDSD (Model Driven Software Development - Modelom riadený vývoj softvéru) [12] je súborom techník, ktoré naväzujú na predchádzajúce techniky, no je všeobecnejší. Tieto techniky opisujú viacerí autori a vyvinuli sa zväčša z praktických skúseností, no čerpajú aj z myšlienok štandardu MDA, no sleduje ho zväčša len na úrovni

[11] `http://www.openarchitectureware.org/`
[12] `http://www.voelter.de/mdsd-book/`

kompatibilného meta meta modelu (MOF). V mnohom sa však od neho odlišujú a to najmä v bodoch, ktoré si vrámci MDA vyslúžili značnú kritiku, kde MDSD prináša pragmatickejšie riešenia. Keďže tieto techniky nemajú jednoznačný štandard, ktorý by ich zastrešoval, častokrát sa spôsoby ich nasadenia od seba veľmi odlišujú a taktiež i podpora v podobe nástrojov je malá, keďže takmer každý nástroj je akoby ostrovom sám pre seba. Každopádne modelom riadený vývoj softvéru sa v praxi osvedčil a i dnes v tejto oblasti prebieha vývoj a výskum a v praxi sa presadzuje stále intenzívnejšie.

1.1.5 DSL

Doménovo špecifické jazyky (DSL - Domain Specific Languages) si v tejto dobe zasluhujú stále viac pozornosti. Zrejme je to spôsobené prevažne potrebou abstrahovať od technických detailov a priblížiť vývoj doménovým špecialistom. Taktiež značným problémom je zložitosť dnešných technológií, ktoré sa často zakladajú na frameworkoch. Framework však svojim aplikačným programovacím rozhraním sám vytvára akýsi jazyk, ktorý je preň špecifický a teda návrh frameworku možno náročnosťou prirovnať k návrhu jazyka, pričom frameworky ako také explicitne nedbajú na notáciu. Tým pádom sa vytvára ideálny priestor na nasadenie doménovo špecifických jazykov. Dobrými príkladmi sú napríklad jazyk Ruby, ktorý umožňuje vytvárať interné doménovo špecifické jazyky, viac o nich v príslušnej kapitole, nástroje MPS (meta Programming System) [13], Microsoft OSLO [14] a ďalšie, ktoré umožňujú vytvárať externé doménovo špecifické jazyky. Tieto nástroje sú však stále vo vývoji a teda nemožno hovoriť o rozsiahlom nasadení, isto však predstavujú blízku budúcnosť. Doménovo špecifické jazyky a techniky ich použitia súvisia s modelom riadeným vývojom softvéru, hoci kvôli častej absencii modelov to nemusí byť zrejmé. Všetky nástroje, ktoré umožňujú vytvárať externé doménovo špecifické jazyky totižto pracujú so štruktúrou, ktorá predstavuje strom abstraktnej syntaxe vytváraného doménovo špecifického jazyka. Práca s týmto stromom je častokrát ekvivalentná práci s modelmi a taktiež je možné v týchto nástrojoch nájsť i trans-

[13] `http://www.jetbrains.com/mps/`
[14] `http://www.microsoft.com/soa/products/oslo.aspx`

formácie, ktoré majú rovnakú úlohu ako v modelom riadenom vývoji softvéru, no zvyknú byť označované inak, napríklad ako projekcie.

1.2 Využitie

Modelom riadený vývoj softvéru našiel využitie v rôznorodých oblastiach vývoja. Spôsob akým možno z týchto techník ťažiť delí ich využitie do troch kategórií, ktoré sa neraz prekrývajú a kombinujú.

1.2.1 Vývoj integrovaných systémov

Integrované (embedded) systémy predstavujú z rôznych dôvodov výzvu. Na tieto systémy sú kladené veľké požiadavky, hlavne pokiaľ ide o spoľahlivosť a efektívnosť, neraz to bývajú i systémy reálneho času čo si vyžaduje vysokú mieru kvality kódu. Zároveň však integrované systémy poskytujú dostatočný priestor na generovanie kódu. Ich funkčnosť je častokrát možné jednoducho opísať konečno stavovým automatom. Nezriedka sú tieto systémy tvorené z komponentov, ktoré bývajú kombinované v konečnom produkte a teda možno hovoriť, že rôzna paleta produktov tvorí spolu akúsi rodinu produktov. Toto všetko sú vhodné predpoklady na nasadenie modelom riadeného vývoja softvéru. Komponenty a ich kombináciu je možné popísať vhodnými modelmi a generovať veľké časti kódu, v niektorých prípadoch i všetok kód.

1.2.2 Nástroj pre doménových špecialistov

Techniky modelom riadeného vývoja softvéru umožňujú sústrediť vedomosti doménovoých špecialistov do modelov, ktorými sa môžu riadiť aj vytvorené aplikácie. Toto mužmoňuje vytvárať aplikácie, ktoré sú vysoko prispôsobiteľné bez nutnosti zásahu vývojárov vďaka riadeniu metadáta zacyteným v modeloch. Kombináciou doménovo špecifických jazykov pre cieľovú biznis doménu a spracovania modelov je možné vytvoriť nástroje pre doménových špecialistov, ktoré abstrahujú od technických detailov a umožnia expertom prispôsobovať aplikáciu svojim potrebám.

Príklady takéhoto využitia existujú, napríklad v doméne poisťovníctva boli vytvorené viaceré nástroje, ktoré umožňujú poistným matematikom vhodným spôsobom navrhnúť poistnú zmluvu v doménovo špecifickom jazyku, trebárs pomocou matematického zápisu vzorca pre vyhodnotenie poistnej udalosti a podobne. Vytvorené modely môžu byť interpretované systémom na spracovanie zmlúv a je možné na nich vyvolať napríklad kontroly a integrovať nový typ zmluvy do biznis procesov.

1.2.3 Softvérová architektúra a produktové línie

Vďaka technikám modelom riadeného vývoja softvéru je možné softvérovú architektúru zachytiť pomocou doménovo špecifického jazyka a z neho vygenerovať základné architektonické bloky systému. Doménovo špecifický jazyk je možné vyvíjať paralelne s rozhodovaním o tom aký architektornický štýl využijeme a teda opis systému pomocou tohto jazyka tvorí do veľkej miery dokumentáciu architektúry. Ak naviac opíšeme spôsob akým sme tento jazyk vytvárali, máme k dispozícií i dokumentáciu k analýze architektúry. Transformácie a generátory následne v sebe ponesú informáciu o zvolenej technologickej platforme. Takto vygenerovaný kód nám následne môže slúžiť ako kostra aplikácie, ktorá jednak poskytuje priestor na dopracovanie, no zároveň zabezpečí vynútenie architektúry.

Ak obsahuje naše portfólio nejakú produktovú líniu, ktorej produkty sa od seba líšia len málo, môžeme tieto zmeny zachytiť prostredníctvom modelov. Tie následne budú slúžiť na konfiguráciu konkrétneho produktu a využitím tohto modelu môžeme generovať požadovaný produkt z dostupných komponentov.

Kapitola 2

Modely a modelovanie

Modely a modelovanie sú ústrednými a kľúčovými faktormi pre modelom riadený vývoj softvéru, preto je nevyhnutné si tieto pojmy ozrejmiť a vysvetliť ich význam a ich chápanie v tomto kontexte.

2.1 Modely

Slovo model je možné definovať ako reprezentáciu reálneho sveta a jeho objektov, ktorá zanedbáva detaily, od ktorých môžeme pre naše potreby abstrahovať. V takomto širšom kontexte sú modelmi trebárs i sochy, modely áut či železníc. Vždy je podstatné si uvedomiť od čoho je možné abstrahovať. Socha prirodzene neberie v úvahu ľudské tkanivá a využíva iné materiály, model auta je zmenšený v istej mierke a model železnice nemusí nevyhnutne obsahovať všetky elektronické systémy železníc. Rovnako je tomu tak i v oblasti vývoja softvéru a softvérového modelovania. Mnohé tradičné metodiky využívajú nejakú formu modelovania, ktorá umožňuje vytvárať modely predstavujúce vyvíjaný systém. Pre tieto potreby v minulosti vzniklo mnoho notácií, no dnes ich takmer úplne nahradil jazyk UML.

V kontexte modelom riadeného vývoja softvéru je podstatné, aby bol model formálny. To znamená, že model je definovaný tak, že je možné ho vytvoriť a spracúvať automatizovane pomocou nástrojov. To znamená, že modely musia mať dobre opísanú vnútornú štruktúru, musia mať presne definovanú sémantiku, teda význam, a že musí existovať dobrý spôsob ako takéto modely popísať a znázorniť.

To čo teda potrebujeme je modelovací jazyk, ktorý by bol dostatočne formálny. Ako prvé riešenie sa núka štandard UML, ktorý je v praxi rozšírený. Komplikáciou však je, že UML neposkytuje dostatočne formálny či lepšie povedané sémanticky jednoznačný model, respektíve nie všetky jeho vlastnosti sú podporované i nástrojmi. Problémom pri vývoji nástrojov podporujúcich tento UML totižto často býva komplikovaná štruktúra tohto jazyka. UML vzniklo spojením niekoľkých notácií, teda grafických značení, a zámerom autorov bolo poskytnúť v prvom rade jednotnú notáciu, nie však modelovací jazyk. Štandard UML vo verzii 2.0 však viedol k výrazným zlepšeniam, hoci problém s nástrojmi pretrváva.

Treba podotknúť, že je podstatné rozlišovať medzi pojmami model a diagram, ktoré sa často zamieňajú. Diagram predstavuje notáciu, spôsob zakreslenia modelu, ktorý je naopak len abstraktnou entitou. Toto rozdelenie je zadefinované i v špecifikácií jazyka UML, kde modelovaný systém je predstavovaný jedným modelom, no ten môže byť reprezentovaný až trinástimi rôznymi typmi diagramov, ktoré predstavujú náhľady na daný systém na základe rôznych kritérií, ktoré sú pre daný typ diagramu špecifické. Diagram spolupráce sa napríklad sústreďuje výlučne na interakciu medzi objektami a triedami. Teda pre model nie je dôležité akým spôsobom je znázornený, ale jeho štruktúra, naopak diagram je istým zobrazením modelu.

Vráťme sa teda späť k nášmu problému modelovacieho jazyka. Ako sme si vysvetlili, UML nie je často najvhodnejším modelovacím jazykom. To je dôležité, keďže modelom riadený vývoj softvéru je širšia oblasť zahŕňajúca aj modelom riadenú architektúru, víziu, ktorú prinieslo konzorcium OMG a ktorá nepriamo podporuje ako základ UML a modelovacie jazyky založené na UML.

Riešením sú doménovo špecifické jazyky. Tieto jazyky sa vytvárajú na základe analýzy danej domény, vyhľadania základných konceptov, ich vlastností a vzťahov medzi nimi. Niekedy sa opisuje i sémantika doménovo špecifického jazyka, no najčastejším riešením je generovanie zdrojového kódu nejakého všeobecného programovacieho jazyka, ktorý sémantiku už má definovanú.

Na popísanie doménovo špecifického jazyka pomocou ktorého budeme vytvárať modely je potrebné definovať jeho abstraktnú syntax, teda vytvoriť meta model, teda model modelov, ktorý bude hovoriť o tom aké typy konceptov môžeme pri mo-

delovaní použiť a aká je ich štruktúra. Napríklad v prípade modelov vytváraných pomocou jazyka UML, meta model opisuje koncepty ako je trieda, operácia, asociácia, generalizácia a podobne. Meta model formálne popisuje obsah modelu, teda vytváraním inštancií meta modelu, čiže modelov, vytvárame inštancie konceptov, ktoré sú inštanciami konceptov zadefinovaných v meta modeli. Napríklad v UML ak chceme vytvoriť triedu firma s atribútom názov, vytvoríme najprv triedu ako inštanciu triedy v meta modeli a potom jej atribút, ktorý je inštanciou atribútu v meta modeli, pričom ich pomenujeme firma a názov.

Ďalšou požiadavkou na doménovo špecifický jazyk je konkrétna syntax. Konkrétna syntax je v tomto kontexte to isté ako v predchádzajúcom notácia, teda spôsob ako vyjadriť konkrétne abstraktné koncepty modelu. Konkrétna syntax doménovo špecifického jazyka môže byť grafická ale aj textová či tabuľková prípadne iná.

2.2 Metamodelovanie

Metamodelovanie ako predpona meta napovedá umožňuje modelovanie modelov. To vlastne znamená, že môžeme vytvoriť model pre naše modely. Takýto model nazývame meta modelom, pričom príslušný model jeho inštanciou. Metamodelovanie je dôležité preto, aby sme vedeli s modelmi pracovať pomocou nástrojov a na to potrebujeme zabezpečiť ich jednotnú štruktúru, ktorú popisuje ich príslušný meta model. Teda meta model nám umožňuje kontrolovať štruktúru modelu a následne jednotne pracovať s viacerými modelmi na základe ich spoločnej štruktúry, teda ich meta modelu.

Vďaka tomuto princípu môžeme napríklad vytvárať transformácie medzi modelmi. Stačí ak modely majú spoločný meta model a teda štruktúru, ktorú môžeme automatizovane transformovať nezávisle od konkrétneho modelu. V prípade, že chceme transformovať medzi sebou dva modely, ktoré sú inštanciou rôznych meta modelov, naša úloha sa komplikuje. Aby sme vedeli transformovať inštanciu jedného meta modelu na inštanciu iného meta modelu, potrebujeme vedieť transformovať štruktúru jedného meta modelu na štruktúru toho druhého a teda potrebujeme, aby mali tieto meta modely spoločnú štruktúru. Teda to čo potrebujeme je meta meta

model, respektíve model modelov modelov.

Takto môžeme úroveň abstrakcie, s ktorou chceme pracovať, ľubovoľne zvyšovať, dôležité je však uvedomiť si dokedy to má praktický význam.

Vrámci štandardu MDA (Model Driven Architecture) [KWB03, MSUW04] boli na základe potrieb, ktoré som opísal vyššie identifikované štyri úrovne abstrakcie, ktoré sú potrebné pre prácu s modelmi.

Najnižšiu úroveň označujeme M0, tá predstavuje reálny objekt. Ak sa snažíme pomocou modelovania zachytiť napríklad štruktúru tried v programovacom jazyku, na tejto úrovni sa nachádza kód programovacieho jazyka deklarujúci túto triedu.

M1 je úrovňou modelu, teda prvý stupeň abstrakcie. Model ako sme si vysvetlili zanedbáva určité detaily. V našom prípade, trebárs v diagrame tried v UML, vrámci uvažovanej triedy môžeme mať verejné polia, ktoré však budú v zdrojovom kóde programovacieho jazyka reprezentované súkromnými poľami s metódami na ich nastavenie a získanie ich hodnoty.

Úroveň meta modelu sa označuje M2 a predstavuje model pre modely. Aby sme mohli vytvárať modely tried, potrebujeme najprv na tejto úrovni definovať čo je to trieda, teda že môže obsahovať polia, metódy a má svoj názov a podobne. UML poskytuje takúto definíciu vrámci diagramu tried.

Poslednou definovanou úrovňou, M3, je meta meta model. V našom príklade meta meta model bude obsahovať informácie o tom ako vytvárať štruktúry, ktoré môžu byť pomenované a mať nejaké atribúty ako napríklad triedy s názvom, poľami a metódami.

Štandard MDA mal medzi svojimi cieľmi interoperabilitu. Ak ju chceme dosiahnúť potrebujeme spoločný model na najvyššej úrovni, teda meta meta model. Vrámci MDA ním bol štandard MOF (MetaObject Facility). V dobe kedy bol vytvorený sa ním riadilo prakticky len UML a keďže pole modelovania bolo široké a projekt EMF (Eclipse Modeling Framework) poskytoval mnoho nástrojov na modelovanie, no vlastný meta meta model v podobe Ecore, ktorý nebol s MOF-om kompatibilný, rozhodlo sa konzorcium OMG vo verzii 2.0 MOF rozdeliť na dva podštandardy. SMOF (Standard MOF) predstavuje MOF v predchádzajúcich verziách a EMOF (Essential MOF), ktorý je zjednodušený a predstavuje v podstate

Ecore. Teda interoperabilita bola týmto zabezpečená. Neskôr sa štandardizovalo na základe MOF-u viacero meta modelov, napríklad i BPMN, a mnéhé sú stále v procese štandardizácie.

Tento proces štandardizácie nebol jednoduchý, lebo v tej dobe vyvstala otázka, ktorú sa doteraz nepodarilo zodpovedať. Existuje jednotný, spoločný meta meta model, ktorý by bol taký pružný, že by umožňoval vytvoriť meta model pre akúkoľvek doménu a poskytoval konštrukcie, ktoré by dokázali opísať rovnako dobre štruktúru i správanie konštrukcií v nej? Ak by taký meta meta model existoval, ostáva otázne ako by vyzeral, či by bol stále použiteľný a či by nebol príliš zložitý. S týmto problémom sa stretli vývojári nepriamo už skôr pri používaní UML. UML sa vynikajúco hodí na opis štruktúry, najmä vďaka diagramu tried, objektov, kompozitnej štruktúry, ale i ďalšími. Naproti tomu diagramy, ktoré opisujú správanie, teda diagram interakcií, sekvenčný, diagram aktivít a podobne sú síce dostatočne názorné, no na opis predovšetkým algoritmov, nie sú také efektívne ako samotný zdrojový kód či pseudokód. I toto vyústilo do presadenia sa textových doménovo špecifických jazykov. Problémom však je, že táto otázka ostala nezodpovedaná, dokonca neexistuje ani teória, ktorá by ponúkala jednoznačnú odpoveď. V poslednom čase je táto problematika trocha zanedbávaná, predovšetkým kvôli tomu, že sa môže ukázať, že ak by taký meta meta model existoval, jeho výrazové prostriedky by boli dosť slabé a vytváranie meta modelov by bolo príliš náročné. Momentálne sa teda spoliehame na to, že meta meta model, ktorý máme k dispozícií, čiže MOF 2.0, bude vo väčšine prípadov postačujúci.

2.3 Obmedzenia

Obmedzenia sú spôsobom ako bližšie špecifikovať vlastnosti prvkov modelov. Ich potreba vyplýva z faktu, že hoci meta model opisuje štruktúru modelov a prvky v nich, neumožňuje vyjadriť obmedzenia na atribúty prvkov a vzťahy medzi prvkami v dostatočnej miere. Práve na to slúžia obmedzenia. Vrámci štandardu UML sa využíva jazyk OCL (Object Constraint Language) [AN05], ktorý si však našiel miesto aj v iných meta modeloch vďaka svojmu dobrému návrhu a štandardizácií.

Tento jazyk bol navrhnutý zvlášť pre špecialistov biznis domén [1] a teda nemohol byť len nejakým rozhraním nad implementáciou modelu vrámci nejakého programovacieho jazyka. Tento jazyk sa inšpiroval teóriou množín, predikátovou logikou prvého rádu a jazykom Eiffel, od ktorého prevzal vstupné a výstupné podmienky v zmysle princípov Design by Contract [2]. Podpora množín umožňuje využitie množinových operácií, ktoré sú dostatočne prirodzené a zrozumiteľné veľkému množstvu doménových špecialistov. Predikátová logika umožňuje vytvárať požiadavky na prvky modelu na základe definície logických výrazov, ktoré majú platiť pre tieto prvky. Pre pohodlie tento jazyk umožňuje aj zreťazenie funkcií, teda výsledkom každej funkcie je vybraná množina, na ktorú je možné priamo aplikovať ďalšiu funkciu. Toto umožňuje vytvárať krátke zápisy, ktoré môžu predstavovať komplexné funkcie.

Základ jazyka OCL tvoria rôzne funkcie na vyhľadávanie a vyberanie prvkov modelov, prácu s výbermi v podobe množinových operácií, no taktiež i logické výrazy na vytváranie podmienok.

Tento jazyk však nenašiel silnú podporu v nástrojoch a tak sa ani v praxi príliš nepresadil, hoci zriedka sa používa. Jeho používanejšou inkarnáciou je jazyk Check nástroja openArchitectureWare, ktorý tento jazyk rozširuje.

2.4 Označkované modely

Označkovanie modelov je technika, ktorá umožňuje rozšíriť informácie, ktoré môže model v sebe niesť. Model je inštanciou nejakého meta modelu, ktorý má nejaké obmedzenia a štruktúru. Ak však chceme nejaký hotový model opätovne využiť a potrebujeme doň pridať informáciu, ktorá nemôže byť jeho prirodzenou súčasťou, napríklad týkajúcu sa konkrétnej platformy či technológie, môžeme využiť možnosti označkovania. Buď môžeme pridať ľubovoľný text ako komentár, čo môže byť trebárs zdrojový kód, ktorý sa prenesie do výsledku transformácie, alebo môžeme pridať označené hodnoty (tagged values), ktoré predstavujú páry názov a hodnota. Tie môžu slúžiť napríklad ako parametre pre transformáciu.

Tento mechanizmus je súčasťou štandardu UML a využíva ho najmä prístup

[1] `http://www.se-radio.net/podcast/2008-12/episode-120-ocl-anneke-kleppe`
[2] `http://www.se-radio.net/podcast/2007-03/episode-51-design-contract`

MDA. V praxi je však lepšie sa mu vyhnúť, lebo predstavuje viaceré problémy. V prvom rade informácia, ktorá sa nedá vyjadriť pomocou prostriedkov daného meta modelu zrejme nemá v uvažovanom modeli miesto a často môže patriť na úplne inú úroveň abstrakcie. Toto nazývame znečistením modelu. Čiastočným riešením je vytvorenie modelu, ktorý priradzuje prvkom uvažovaného modelu značky. Toto je však dosť náročné, lebo to znamená, že je potrebné skopírovať pôvodný model a pridať k nemu značky. I toto by sa dalo vyriešiť pomocou techník aspektovo-orientovaného programovania, no je potom otázne či to ozaj prináša zjednodušenie. Netreba zabúdať, že externý model pre označkovanie nerieši problém znečistenia modelu, len ho presúva, lebo v istom momente bude potrebné značky k modelu pripojiť.

Vo všeobecnosti, ak je značiek pár, je možné model rozšíriť, ak to nebude predstavovať prílišné znečistenie. Ak je v značkách kód, je lepšie vytvoriť podmienky na jeho ručné dopísanie do generovaného kódu, respektíve podtriedy vygenerovanej abstraktnej triedy. Ak je značiek naopak mnoho, no ich variabilita je malá, teda častokrát používame rovnaké značky, riešením môžu byť parametre pre transformáciu v externom konfiguračnom súbore. V prípade, že je značiek mnoho a sú dosť rôzne, treba sa zamyslieť či nehovoríme o aspekte, teda pohľade, a či by nebolo efektívne vytvoriť nový model a prepliesť ho s pôvodným pomocou techník aspektovo-orientovaného programovania.

2.5 Znovupoužiteľnosť modelov

Keďže modely sú prostriedkom na zachytenie informácií a vedomostí doménových špecialistov, pre ich efektívne využitie by sme chceli zabezpečiť možnosť ich opätovného využitia. V istom zmysle možno modely prirovnať k vzorom, lebo taktiež ide o riešenie nejakého konkrétneho problému, ktoré však nie je v ich všeobecnosti možné zachytiť na úrovni zdrojového kódu, ale potrebujeme vyššiu úroveň abstrakcie.

Modely sami o sebe prinášajú do vývoja softvéru istú hodnotu, no ich ozajstný prínos pocítime prostredníctvom transformácií a najmä generátorov, ktoré nám z týchto modelov vytvoria komponenty potrebné pre systém, či už zdrojové kódy alebo

konfiguračné súbory. Modely spolu s príslušnými meta modelmi, transformáciami a generátormi nazývame metaware, čiže softvér, ktorý opisuje softvér. Metaware by mal byť modulárny, atomický a prispôsobiteľný, aby sme boli schopní využiť ho znova i v inom projekte. Na toto treba pamätať už pri jeho vytváraní. O niektorých vhodných postupoch si povieme v kapitole o najlepších postupoch a vzoroch.

V neposlednom rade treba zabezpečiť interoperabilitu medzi nástrojmi, ktoré pracujú s modelmi a to prostredníctvom štandardného meta meta modelu. Tým pádom zabezpečíme nielen jednotnú štruktúru meta modelov a modelov, ale taktiež výhodou je i sústredenie sa na abstraktnú syntax, respektíve vnútornú štruktúru, čo sa vyplatí pri vytváraní a opätovnom používaní transformácií. Takto môžeme úplne odignorovať konkrétnu syntax a nezávisle na tom, či použijeme textovú alebo grafickú notáciu, prípadne inú, sme schopní metaware znova použiť.

Kapitola 3

Transformácie

Na to, aby sme mohli s modelmi automatizovane pracovať potrebujeme transformácie. Transformácia je funkcia, ktorá pracuje s modelom či modelmi a jej výsledkom môže byť zmenený model, nový model alebo vygenerovaný kód či ľubovoľný iný artefakt, ktorý môžeme vďaka informáciám zachyteným v modeloch automatizovane vygenerovať.

Aby mohli byť transformácie opätovne použiteľné, je potrebné, aby nezávisely od konkrétnych modelov, ale od meta modelov a umožňovali tak transformovať ľubovoľný model, ktorý je inštanciou požadovaného meta modelu. Toto je rovnaký princíp ako využívajú kompilátory, ktoré transformujú ľubovoľný korektný program v danom programovacom jazyku.

3.1 Vlastnosti transformácií

Pre ich efektívne využitie očakávame od transformácií v zmysle literatúry [KWB03] viaceré vlastnosti, ktoré si opíšeme.

3.1.1 Prispôsobiteľnosť

Jednou z nich je prispôsobiteľnosť. Hoci transformácie obsahujú pravidlá na prácu s prvkami modelu, často sa môže stať, že výsledok transformácie závisí od externe dodaných parametrov a preto by mali transformácie poskytovať vhodný spôsob na ich parametrizáciu. Príkladom potreby parametrizácie môže byť voľba vhodného

typu implementácie pre transformáciu generujúcu implementáciu pre stavové automaty, trebárs zo stavových diagramov UML. Jednou z možností je využiť návrhový vzor State [1], ktorý ponúka elegantné riešenie z pohľadu návrhu, inou je vygenerovať podmienkovú logiku prostredníctvom if konštrukcií, ktorá je ťažšie spravovateľná, no efektívnejšia alebo využiť tabuľku prechodovej funkcie, čo je najefektívnejšie riešenie, obzvlášť vhodné pre integrované (embedded) systémy. Túto vlastnosť je možné dosiahnúť viacerými spôsobmi. Požadovanú informáciu môžeme umiestniť do modelu či už ručne alebo ako výsledok predchádzajúcej transformácie, trebárs v podobe značky. Tento postup je jednoduchý, prináša však so sebou nevýhodu a to tú, že do modelu sa dostávajú informácie, ktoré nie sú preň priamo relevantné a nie je možné ich vhodne overiť, teda dôjde k už spomínanému znečisteniu modelu. Inou možnosťou je vytvorenie externého značkovacieho modelu, ktorý popisuje ako popísať značkami model, ktorý máme k dispozícií, čím sa vlastne odstráni závislosť medzi modelom a značkami. To si však vyžaduje vytvorenie nového modelu, ktorý musí odrážať štruktúru modelu, ktorý chceme označkovať. Asi najschodnejšou cestou je vytvorenie akéhosi rozhrania pre transformáciu, vhodne v podobe externého konfiguračného súboru, pomocou ktorého je možné definovať argumenty pre transformáciu, odhliadnúc od vstupného modelu a naopak zamerajúc sa na možnosti voľby, ktoré poskytuje daná transformácia. Ak zavedieme štandardný spôsob na definovanie rozhraní, môžeme jednoducho predávať argumenty pomocou generovania nových konfiguračných súborov. Toto môže umožniť aj prenášanie argumentov, ktoré transformácia nevie spracovať, lebo pri definícií transformačného prechodu, teda postupu transformácií, je možné overiť či parameter vyhovuje rozhraniu niektorej z použitých transformácií. Takýto postup je vhodné kombinovať s kaskádovaným zapojením transformácií.

3.1.2 Trasovateľnosť

Požiadavka na túto vlastnosť pre transformácie plynie z prístupu MDA (Model Driven Architecture, [KWB03, MSUW04]). Ide o to, aby bolo možné v modeli

[1] Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, ISBN 0-201-63361-2

alebo zdrojovom kóde, ktorý je výsledkom transformácie, identifikovať prvok zo vstupného modelu, ktorého transformáciou vznikol uvažovaný prvok na výstupe. Takéto prepojenie je potrebné v prípade, že chceme výsledok transformácie ručne upravovať. Prečo by sme niečo také chceli si vysvetlíme nižšie. V praxi je takýto postup občas nevyhnutný, no je lepšie sa mu vyhnúť, lebo je ťažké takúto informáciu udržiavať synchronizovanú. Je síce možné vytvárať takzvané inštancie transformácií, čo sú externé súbory opisujúce priebeh konkrétneho behu transformácie na konkrétnom modeli, no okrem toho, že takýto postup je dosť náročný, nerieši ďalšie problémy prístupu MDA. To isté platí i pre využitie identifikátorov na synchronizáciu. Využitím vhodných praktík opísaných v kapitole o najlepších postupoch a vzoroch a príslušných článkov, je možné sa týmto problémom vyhnúť a obísť sa bez tejto vlastnosti.

3.1.3 Prírastková konzistencia

Toto súvisí s vyššie uvedenou vlastnosťou. Od transformácií požadujeme, aby bolo možné do ich výsledkov pridávať nové informácie a aby sa tie po ďalšom behu transformácie neprepísali. Práve na to môže byť dobré mať uchovanú informáciu o tom či bol daný výsledný prvok generovaný a z akého prvku vstupného modelu, pre prípady, kedy prvok vo vstupnom modeli zanikne. Táto požiadavka dala vznik chráneným oblastiam kódu, ktoré majú zabezpečiť správu kódu, ktorý bol ručne dopísaný. Chceme zabezpečiť, aby nebol pregenerovaný, respektíve, aby po ďalšom generovaní sa ocitol na správnom mieste. Rovnako ako v predchádzajúcom prípade je táto vlastnosť zväčša nepotrebná, ak využijeme vhodné prostriedky na pridávanie ručne písaného kódu alebo ručne pridaného prvku modelu. Jednou z možností je generovať abstraktné triedy, ktoré môžeme špecializovať v inom súbore a rozšírenie tohto princípu i na modely.

3.1.4 Obojsmernosť

Úprava výsledkov transformácií vedie, okrem nekonzistencie s transformáciou, i k nekonzistencii so vstupným modelom, na ktorý môžu mať zmeny výsledku vplyv. Napríklad pri zmazaní všetkých prvkov vo výslednom modeli, ktoré vznikli zo

spoločného prvku vstupného modelu, stráca tento prvok vo vstupnom modeli svoj význam a očakávali by sme jeho odstránenie. Toto je obzvlášť nevhodné, lebo to znamená, že požadujeme, aby boli transformácie obojsmerné a teda bijektívne. To však nemá zmysel, lebo ak by boli čo i len surjektívne zmysel modelov na zvýšenie úrovne abstrakcie sa stratí. Injektívnosť taktiež nemusí byť samozrejmosťou. Riešením by bolo teda vytvárať ku každej transformácií i spätnú, čo je ozaj netriviálne a nemožno tento postup automatizovať. Našťastie však už spomenuté praktiky z poslednej kapitoly umožňujú vyhnúť sa tomuto problému. V spomenutom príklade môžeme odstránenie prvku jednoducho vykonať ručne na úrovni vstupného modelu a neupravovať žiaden model, ktorý by bol medzivýsledkom série transformácií alebo v prípade, že by to nebolo možné, implementovať odstránenie prvku ako transformáciu medzi modelmi, ktorá sa stane súčasťou prechodu medzi transformáciami.

3.1.5 Aké teda by mali byť transformácie?

Ak upustíme od prístupu MDA, ktorý nám prináša niekoľko komplikácií a využijeme praktiky, ktoré navrhuje Markus Völter v [Vö08, Vö09, VB04] ako napríklad kaskádovanie, externé označkovanie a podobne, sme schopný sa týmto problémom vyhnúť.

To čo od transformácií skutočne treba požadovať je v prvom rade modulárnosť. Modulárnosť umožní transformácie kombinovať a opätovne využívať hotové časti alebo i celé transformácie. Ďalej je potrebné zabezpečiť možnosť zmeny správania transformácií pre konkrétne potreby, napríklad zavedením nového parametra. Na to, aby sme to mohli spraviť neintruzívne, možno využiť preťaženie transformácií alebo techniky aspektovo-orientovaného programovania.

Transformácie by mali mať možnosť medzi sebou komunikovať, respektíve si predávať parametre v prípade, že transformácia disponuje vstupnou informáciou, ktorú sama nedokáže využiť a predpokladáme, že pre inú transformáciu bude potrebná.

Transformácie by mali byť samozrejme jednoducho vytvárateľné a upravovateľné a teda je vhodné mať k dispozícií jazyk, ktorý by bol na ich písanie vhodný, jednoduchý a dostatočné výrazový, samozrejme aj s podporou nástrojov. Tento jazyk by podľa toho čo bolo na začiatku tejto kapitoly spomenuté by nemal byť závislý od

konkrétneho modelu, no ako si ukážeme v časti o spájajúcich transformáciách, ani od konkrétneho meta modelu.

3.2 Transformácie medzi modelmi

Transformácie medzi modelmi sú, ako z ich názvu vyplýva, transformácie, ktorých výsledkom je opäť nejaký model. Takéto transformácie teda nie sú generujúce, keďže o generovaní hovoríme v prípade kedy výsledkom transformácie je zdrojový kód alebo konfiguračný súbor či iný artefakt predstavujúci reálny objekt.

Tieto transformácie majú špeciálne postavenie, pretože ich význam môže byť diskutabilný. Totižto, ak sme schopní zo vstupného modelu transformáciami dospieť do stavu, kedy nasledujúca transformácia bude schopná generovať kód, aký je význam v prechádzaní a hlavne vo vytváraní viacerých transformácií, ak by stačila jedna generujúca? Na túto otázku existuje viacero odpovedí, ktoré rozvedieme v nasledujúcich podsekciách.

3.2.1 Transformácia modelov na zjednodušenie generátora

Prvým zjavným dôvodom na využitie transformácií je prevrátenie idey, ktorá nás viedla k zamysleniu sa nad ich potrebou. Generátor, teda transformácia, ktorej výsledkom je zväčša kód, prípadne niečo iné ako model, môže byť dosť komplikovaný a náročný na údržbu. Naviac takýto postup využitia jednej generujúcej transformácie znemožňuje efektívne opätovné využitie transformácií. Netreba zabúdať ani na jednu z najväčších výhod modelom riadeného vývoja softvéru, ktorou je nepochybne prechod na vyššiu úroveň abstrakcie. Použitím transformácií medzi modelmi môžeme prirodzene prechádzať medzi jednotlivých úrovňami abstrakcie a umožniť tak generátoru pracovať s konceptami, ktoré budú na úrovni, kedy je pre generátor jednoduché s nimi pracovať a je možné sa pri návrhu generátoru viac sústrediť na optimálne využitie šablón prípadne iného mechanizmu na generovanie kódu.

3.2.2 Transformácie v roli optimalizácie

Transformácie môžu v roli optimalizácie vystupovať dvojako. Prvým prípadom je prispôsobenie meta modelu vstupného modelu, ak máme k dispozícií hotové zásobníky, teda moduly generátorov, ktoré potrebujú informáciu, ktorá sa v modeli síce nachádza, no v inej forme, v inom meta modeli. Rovnako môžeme využiť i hotové interpretátory modelov. Takáta transformácia potom plní úlohu akéhosi adaptéru pre dané dva meta modely. Optimalizácia však môže prebiehať aj na úrovni rovnakého meta modelu. Toto je možné dobre vysvetliť na príklade fungovania kompilátorov pri optimalizovaní stromu abstraktnej syntaxe [2], kedy je napríklad možné uzly predstavujúce matematické výrazy, ktoré sa dajú vyhodnotiť, nahradiť uzlom s konštantou. Podobne môžeme nahradzovať prvky modelu, ale taktiež ich aj v prípade potreby pridávať, napríklad ak náš model opisuje stavový automat a my chceme na ňom spraviť zmenu, ktorá by si vyžadovala zbytočnú ručnú námahu alebo zbytočne zaťažovala zložitosťou generátor. Táto možnosť využitia transformácií medzi modelmi bola prediskutovaná aj Markusom Völterom, konzultantom, ktorý často využíva techniky modelom riadeného vývoja softvéru a Juha-Pekka Tolvanenom z firmy MetaCase, ktorá sa zaoberá vývojom nátrojov pre doménovo špecifické jazyky [3]. Transformácia v roli optimalizácie môže využívať techniky statickej analýzy kódu na to, aby mohla zoptimalizovať model a tým trebárs rovnako ako kompilátor zefektívniť výsledný zdrojový kód.

3.2.3 Simulácia pomocou transformácií modelov

Posledným prípadom, v ktorom je vhodné využiť transformácie medzi modelmi, je simulácia. Všeobecný pojem modelu rozhodne nie je nový a tak je pochopiteľné, že existuje veľa nástrojov, ktoré umožňujú rôzne formálne modely simulovať, overovať a vyhodnocovať. Spomenúť možno nástroje ako SMC [4], ktoré umožňujú generovať kód pre stavové automaty z vlastného modelu alebo rôzne pravidlové automaty (rule engines), ktoré môžu napríklad na základe modelu BPMN [5] riadiť biznis procesy.

[2] `http://www.se-radio.net/podcast/2007-07/episode-61-internals-gcc`
[3] `http://www.se-radio.net/podcast/2008-12/episode-119-dsls-practice-jp-tolvanen`
[4] `http://smc.sourceforge.net/`
[5] `http://www.omg.org/bpmn/`

3.3 Spájajúce a rozdeľujúce transformácie

Hoci to už bolo nepriamo spomenuté, treba zdôrazniť, že transformácie môžu mať viacero vstupných i výstupných modelov.

Transformácie môžu mať viacero vstupov z niekoľkých dôvodov. Jedným z nich je, že každý model na vstupe predstavuje nejaký aspekt a to čo transformácia robí je prepletanie modelov (model weaving). O tomto postupe budem písať viac v kapitole o aspektovo-orientovanom programovaní. Podobnou príčinou môžu byť modely, ktoré slúžia ako rôzne pohľady na daný problém alebo systém. Tieto pohľady môžu mať rôznu reprezentáciu a zväčša minimálne prekrytie, no často je ich treba skombinovať. O pohľadoch sa možno dočítať viac v kapitole o najlepších postupoch a vzoroch. Poslednou príčinou môžu byť partície, teda fyzicky oddelené časti modelov. Tie je treba v istom momente spojiť, trebárs pred generovaním alebo overovaním obmedzení, a práve na to slúžia tieto transformácie. Partície budú taktiež spomenuté v kapitole o najlepších postupoch.

Transformácie majú viacero výstupných modelov v prípadoch, keď je treba oddeliť od seba časti modelu kvôli generátorom alebo vytvoriť nové modely, ktoré prenášajú informácie do iných modelov, ktoré sú špecifické pre nejaké technológie, ktoré chceme pri generovaní využiť. S týmto sa stretneme pri vývoji trojvrstvových aplikácií, kedy je potrebné preniesť tie isté informácie zvlášť do modelu pre objektovo-relačný mapovač, modelu biznis logiky a modelu prezentačnej vrstvy.

3.4 Vizualizačné transformácie

Niekedy je vhodné modely vizualizovať, čiže graficky znázorniť. Obzvlášť žiadúci je tento postup v prípade, keď na vytváranie modelov využívame textové doménovo špecifické jazyky, vtedy je vizualizácia jedinou možnosťou ako vytvoriť vhodný prehľad o systéme. V takej situácií je možné využiť nástroje, ktoré uľahčujú vizualizáciu ako napríklad GraphViz [6] a Prefuse [7]. Tie ponúkajú napríklad vytváranie grafovej reprezentácie a jej uloženia v podobe obrázkov. Môžeme využiť náš mo-

[6] `http://www.graphviz.org/`
[7] `http://prefuse.org/`

del a vytvoriť transformáciu, z ktorej je jednoduché generovať vizualizáciu pomocou generátora potrebných konfiguračných súborov konkrétneho nástroja. Túto možnosť implementuje napríklad nástroj Xtext [8], ktorý poskytuje hotový generátor a doménovo špecifický jazyk dot na využitie nástroja GraphViz.

[8] `http://wiki.eclipse.org/Xtext`

Kapitola 4

Doménovo špecifické jazyky

Už v minulosti boli snahy vytvoriť programovacie jazyky, ktoré by boli prístupné rôznym doménovým špecialistom. Takýto jazyk bol napríklad COBOL (COmmon Business Oriented Language), ktorý bol určený najmä pre doménu finančníctva a administratívy a hoci je to jeden z najstarších jazykov vôbec, používa sa dodnes.

V dnešnej dobe sú v oblasti matematiky známe Matlab a Octave, ktoré dali v podstate vzniknúť novému priemyslu a umožnili matematikom jednoducho využiť pri svojich výpočtoch a simuláciách programovanie. Rovnako tak rozšírenie jazyka XML je spôsobené predovšetkým možnosťami prispôsobenia syntaxe pomocou schémy, pre konkrétne potreby.

Teda jazyky, ktoré sú zamerané na určitú oblasť, doménu, umožňujú priblížiť programovanie širokej vrstve doménových špecialistov v danej doméne a nepodliehajú takým zmenám ako všeobecné programovacie jazyky, ktoré bývajú často previazané i s inými faktormi ako napríklad technologickou platformou. Domény by sme mohli rozdeliť do dvoch oblastí. Biznis domény sú domény, ktoré nesúvisia priamo s technológiou, sú to domény, ktorými sa zaoberajú špecialisti mimo vývojový tím, respektíve analytici pre danú oblasť. Bankovníctvo, poisťovníctvo a grafický návrh sú dobrými zástupcami biznis domén. Druhou skupinou sú technické domény. Sem patria domény, ktorými sa zaoberajú softvéroví inžinieri a častokrát súvisia s technickými detailmi systému. Komponenty, softvérová architektúra vo všeobecnosti a perzistencia dát sú niektoré z technických domén.

Čo to je teda doménovo špecifický jazyk? Je to jazyk, ktorý umožňuje špecialistom

určitej domény uľahčiť prácu a priniesť im reálnu hodnotu. Do istej miery môžeme takýto jazyk chápať ako žargón istej domény. Treba si všimnúť, že nehovoríme o programovacom jazyku, ale len o jazyku. Dôvodom je skutočnosť, že od programovacích jazykov sa zväčša očakáva sila Turingových strojov, čo umožňuje v nich opísať akýkoľvek algoritmus. To však vôbec nemusí byť pre danú doménu rozhodujúce, ba čo viac, častokrát to môže byť zbytočnou komplikáciou doménovo špecifického jazyka. Je mnoho domén, kde sa užívatelia doménovo špecifického jazyka uspokoja s možnosťou deklaratívneho konfigurovania a to pre ich potreby stačí. Ďalším kritériom je, že doménovo špecifický jazyk má priniesť špecialistom hodnotu. To znamená, že taký jazyk má byť zameraný na doménových špecialistov, čiže má byť pre nich používateľsky príjemný a vhodný na riešenie ich problémov. To má mnohé dôsledky na návrh jazyka, povieme si o nich nižšie.

V dnešnej dobe doménovo špecifické jazyky naberajú na dôležitosti a v tejto oblasti sa robí i výskum. Vhodným zdrojom informácií môže byť i kniha [1] Martina Fowlera, ktorá je síce len v procese písania, no už teraz sa stáva referenčným materiálom.

4.1 Notácia

Pre špecialistov, najmä z oblasti biznis domén, je veľmi dôležitá notácia, teda konkrétna syntax doménovo špecifického jazyka. Existuje viacero druhov notácie, ktoré by sme mohli využiť.

Textová notácia je istotne jednou z najpreferovanejších. Jej veľkými výhodami sú integrácia s existujúcimi nástrojmi, napríklad systémami na správu verzií a porovnávanie, taktiež pomerne jednoduché vytváranie editorov či možnosť použitia hotových editorov. Taktiež existujú nástroje na vytváranie parserov z gramatík vo vhodnej forme. Príkladom môže byť nástroj Antlr [2], ktorý na vstupe očakáva gramatiku v EBNF (Extended Backus-Naur Form) [3] forme a na výstupe je zdrojový kód parseru. Ďalšou výhodou textovej notácie je možnosť jednoducho vyjadriť riadiace

[1] `http://martinfowler.com/dslwip/`
[2] `http://www.antlr.org/`
[3] `http://en.wikipedia.org/wiki/EBNF`

konštrukcie a správanie. V neposlednom rade textové formálne jazyky sú dostatočne známe a existuje dostatok literatúry ohľadom ich návrhu a spracovania.

Veľmi dôležitou je i grafická notácia, ktorá sa veľmi rozšírila vďaka štandardu UML, ktorý ponúka v aktuálnej verzií trinásť rôznych diagramov. Najvyužívanejšími sú diagramy tried, ktoré umožňujú znázorniť štruktúru, diagramy aktivít, ktoré sú vhodné na pracovné postupy a diagramy stavových automatov, ktoré umožňujú sledovať zmeny stavu. Na tieto účely, ale predovšetkým na opis vzťahov je grafická notácia obzvlášť vhodná, lebo je možné vidieť všetky vzťahy a prepojenia na jednom mieste čo textová notácia neumožňuje. Grafická notácia má však svoje slabiny v možnostiach implementácie, kde si vyžaduje veľa práce vytvorenie nástrojov, ktoré by umožňovali príjemnú prácu. Problémom je i definícia sémantiky, významu, grafickej notácie.

Okrem týchto existujú i iné spôsoby notácie, trebárs formulárová či tabuľková. Dôležitá je však možnosť kombinovať notácie s plnou podporou editorov a funkciami ako dopĺňanie kódu a vytváranie prvkov kombinovane. Toto však bohužiaľ dnešné nástroje väčšinou neumožňujú.

4.2 Spracovanie

Doménovo špecifické jazyky delíme podľa spôsobu definovania na interné a externé.

Interné doménovo špecifické jazyky sú definované pomocou nejakého všeobecného programovacieho jazyka. Napríklad jazyk Ruby podporuje tvorbu interných doménovo špecifických jazykov priamo, C++ [C++01] to umožňuje prostredníctvom šablónového metaprogramovania, príkladom využitia je knižnica Proto [4] vrámci projektu Boost a taktiež aj UML profily možno chápať ako interné doménovo špecifické jazyky. Tieto jazyky majú zväčša jednoduchú implementáciu, lebo ich návrhár má k dispozícií konštrukcie všeobecného jazyka. Nevýhodami je však problém s ošetrovaním chýb, ktoré sú analyzované zväčša na úrovni všeobecného programovacieho jazyka, čo môže byť nežiadúce najmä pre doménových špecialistov z biznis domén. Týmito problémami je obzvlášť známy jazyk C++, ktorý nebol navrhovaný s ohľadom na

[4] `http://www.boost.org/doc/libs/1_38_0/doc/html/proto.html`

možnosti šablónového metaprogramovania a tak bývajú výsledkami chýb pridlhé chybové hlásenia, ktoré sú ťažko zrozumiteľné i pre vývojárov. Tieto jazyky sú ešte obmedzené i syntaxou hosťujúceho jazyka a rovnako to platí i o ich sémantike. Vrámci spomínaného C++, je možné predefinovať význam operátorov, no nie je to možné so všetkými operátormi jazyka. Nezriedka treba robiť rozhodnutia či zlepšiť syntax navrhovaného doménovo špecifického jazyka za cenu nutnosti využitia zložitých konštrukcií všeobecného programovacieho jazyka, a využitím hotových konštrukcií všeobecného jazyka za cenu, že do doménovo špecifického jazyka prenesieme časť syntaxe, ktorá nemusí byť pre danú doménu vhodná, čitateľná ani žiadúca.

Externé doménovo špecifické jazyky naproti tomu sú zväčša vytvorené pomocou nástrojov na parsovanie, transformácie a generovanie. Transformácie a generátory sú súčasťou toho, čo nazývame metaware vrámci modelom riadeného vývoja softvéru a tak tou najpodstatnejšou časťou je vytvorenie parseru. Samozrejme to znamená dodatočné úsilie, no unožňuje to i oveľa väčšiu flexibilitu pri definovaní konkrétnej syntaxe a sémantiky doménovo špecifického jazyka. Keďže vývoj programovacích jazykov je podložený teóriou a táto oblasť sa skúma, vzhľadom na iné oblasti informatiky, už dlhšiu dobu, existuje i viacero nástrojov umožňujúcich vytvoriť zdrojový kód parseru. Známe sú yacc, lex, bison či Antlr. Tieto sú však pomerne náročné a vyžadujú si znalosť parsovacích algoritmov, no k dispozícií sú už i nástroje na vyššej úrovni abstrakcie ako napríklad Xtext [5], ktorý umožňuje veľmi jednoducho definovať gramatiku a vytvorí nielen parser, ale i meta model daného jazyka, v podstate by sa dalo povedať, že ide o strom abstraktnej syntaxe, a taktiež pripraví i editor so zvýrazňovaním, navigáciou, dopĺňaním kódu, prehľadom a tak ďalej.

Okrem tohto delenia existujú i mnohé ďalšie. Za zmienku stojí rozdelenie editorov doménovo špecifických jazykov, ktoré vplýva výrazne na spôsob ich spracovania. Externý doménovo špecifický jazyk môžeme definovať najprv pomocou gramatiky, respektíve parseru, a následne pracovať s ním prostredníctvom stromu abstraktnej syntaxe alebo vygenerovaného meta modelu. Pre tento postup sú dôležité znalosti návrhu jazykov. Na takomto princípe funguje už spomínaný Xtext. Iný prístup

[5] `http://wiki.eclipse.org/Xtext`

ponúkajú takzvané projekčné editory, ktoré najprv vytvárajú strom abstraktnej syntaxe či meta model a potom je nutné vytvoriť projekcie, čo sú transformácie z modelu do doménovo špecifického jazyka a projekcie do cieľového všeobecného programovacieho jazyka, čo sú vlastne generátory. Projekcie môžu byť tesne previazané s meta modelom, môžu umožňovať jeho úpravu pomocou doménovo špecifického jazyka priamo, bez nutnosti parsovania. Tento prístup si vyžaduje vytvárať koncepty pomocou úpravy stromovej štruktúry, čo nemusí byť také prirodzené a pre biznis špecialistov prijateľné. Príkladom takéhoto nástroja je Jetbrains Meta Programming System [6] alebo Intentional Domain Workbench [7].

4.3 Návrh

Návrh programovacích jazykov vo všeobecnosti je dosť komplexný problém a hoci sú doménovo špecifické jazyky svojou štruktúrou a rozsahom zväčša jednoduchšie, je potrebné pri ich návrhu dodržiavať tie isté pravidlá.

Prvým by mohlo byť obmedzenie výrazovosti. Treba si uvedomiť, že doménovo špecifický jazyk nemusí byť programovací a že nemusí byť všeobecne použiteľný, stačí ak spĺňa požiadavky pre konkrétnu doménu.

Ďalej je podstatné zvoliť vhodnú notáciu pre danú doménu či konkrétnych užívateľov, ktorí už i sami môžu nejakú notáciu používať.

Veľmi dôležité je v prvom rade zadefinovať koncepty, ktoré jazyk priamo podporuje a s ktorými by mali doménoví špecialisti pracovať. Tu sa núkajú v zásade dve možnosti. Buď navrhneme jazyk, ktorý bude podporovať mnoho konceptov alebo navrhneme taký, ktorý ich bude podporovať len nevyhnutné množstvo. Ak sa rozhodneme pre viacero konceptov, môžeme dosiahnúť stav, kedy je na riešenie problémov vždy po ruke vhodný koncept. Toto je veľmi vhodné pre špecialistov biznis domén, najmä ak vytvoríme vhodnú formu dokumentácie, kde bude jednoduché si vyhovujúci koncept vyhľadať. Minimalizácia počtu konceptov naproti tomu umožňuje udržať konzistenciu jazyka a zabezpečiť jeho jednoduchosť a priaznivú krivku učenia sa. Zároveň takýto jazyk si vyžaduje abstraktné myslenie, aby užívateľ

[6] `http://www.jetbrains.com/mps/`
[7] `http://msdn.microsoft.com/en-us/oslo/dd727740.aspx`

jazyka bol schopný svoj problém vyjadriť pomocou konceptov, ktoré jazyk ponúka.

Ďalším dôležitým faktorom, ktorý vplýva na návrh doménovo špecifického jazyka je jeho evolúcia. Ak nerátame s vývojom jazyka vopred, môže sa to stať neskôr problémom. Ak jazyk spracúvame prostredníctvom parseru, nie je možné rátať s tým, že bude možné do gramatiky niečo pridať alebo viaceré spojiť, lebo výsledná gramatika nemusí byť jednoznačná a teda nebude parsovateľná. Tento problém nastáva aj keď sa pokúšame kombinovať rôzne jazyky alebo ich časti, teda ak chceme mať k dispozícií opätovne použiteľné časti jazykov, ktorými by mohli byť napríklad typové systémy, matematické výrazy a podobne. Tento problém je však možné čiastočne obísť pomocou scannerless parsovania [8] alebo obchádzaním parsovania prostredníctvom projekčných editorov doménovo špecifických jazykov.

Ďalšou požiadavkou je umožniť vytvoriť partície, teda fyzicky oddeliť od seba časti kódu a to si vyžaduje externé odkazy a vloženie súborov. Rovnako tak, ak chceme vytvoriť pohľady, viac v kapitole o najlepších postupoch a vzoroch, ale i v časti o aspektovo orientovanom programovaní, potrebujeme vytvoriť v jazyku rôzne sekcie respektíve body spojenia, vrámci ktorých môžeme potom oddelené pohľady kombinovať a spájať.

Hoci sú doménovo špecifické jazyky od tých všeobecných programovacích dosť odlišné, je dôležité poučiť sa pri ich návrhu zo skúsenosti s programovacími jazykmi. Menné priestory, rozsahy platnosti a mnohé ďalšie princípy môžu byť pre doménovo špecifické jazyky veľmi výhodné. Taktiež je dobré poznať rôzne programovacie paradigmy, aby sme boli schopní navrhnúť čo najvhodnejší doménovo špecifický jazyk, ktorý bude ponúkať konštrukcie, ktoré sú na riešenie problémov konkrétnej domény najlepšie. O návrhu programovacích jazykov existuje množstvo literatúry, ktorá môže byť dobrým zdrojom inšpirácie pri návrhu doménovo špecifických jazykov.

[8] `http://www.se-radio.net/podcast/2008-11/episode-118-eelco-visser-parsers`

Kapitola 5

Generatívne programovanie

Generatívne programovanie (GP, Generative programming) [1] je súborom techník, ktoré pri tvorbe softvéru využívajú rôzne možnosti generovania zdrojového kódu, ktoré umožňujú zefektívniť vývoj a zaručiť kvalitu či konzistenciu.

Je obtiažne povedať v akom vzťahu je generatívne programovanie voči modelom riadenému vývoju softvéru. Generatívne programovanie ponúka viacero možností ako vytvárať kód a neviaže sa na žiadne štandardy, preto by sme sa mohli domnievať, že modelom riadený vývoj softvéru, je v podstate len zvláštna forma generatívneho programovania, ktorá sa sústreďuje na využitie modelov. Na druhú stranu, modelom riadený vývoj softvéru ponúka komplexnejší pohľad na vývoj a nezaoberá sa len produkovaním kódu, ale hlavným cieľom je ako sme si už povedali, zvýšenie úrovne abstrakcie, čo sa dosahuje prostredníctvom využitia modelov. Tým pádom je zrejmé, že tieto techniky nie sú disjunktné, prelínajú sa, no zároveň sa zameriavajú na trocha iné oblasti softvérového inžinierstva.

Z pohľadu modelom riadeného vývoja softvéru je význam generatívneho programovania predovšetkým v konkrétnych postupoch a skúsenostiach, ktoré ponúka na dosiahnutie automatizácie tvorby zdrojového kódu. V nasledujúcich sekciách si opíšeme niektoré techniky generatívneho programovania a ich súvis s modelom riadeným vývojom softvéru.

[1] `http://www.se-radio.net/podcast/2008-05/episode-96-interview-krzysztof-czarnecki`

5.1 Metaprogramovanie

Metaprogramovanie je programovacia paradigma, ktorá využíva možnosti programovania pri tvorbe programov. To zväčša znamená, že pri programovaní vytvárame program, ktorý programuje, teda má možnosť využiť ľubovoľný algoritmus na vytvorenie algoritmu iného. Toto je prirodzená technika najmä pre interpretované jazyky, ktoré môžu vytvoriť zdrojový kód a vykonať ho prostredníctvom interpreteru.

Pre ilustráciu podstaty metaprogramovania uvediem príklad. V jazyku C je možné používať makrá. Makrá sú v zásade funkcie, ktoré pracujú len s textom zdrojového kódu. Ak chceme vytvoriť multiplatformový program v jazyku C, zväčša musíme brať v úvahu rozdiely v platformách. Vtedy sa kód pre špecifické platformy zabalí do podmienky v makre, ktorá na základe premennej určí, pre ktorú platformu treba vybrať kód. Pred kompiláciou programu v jazyku C sa najprv vyvolá makroprocesor, ktorému sa priradí hodnota príslušnej premennej podľa toho, pre akú platformu chceme práve kompilovať. Makroprocesor upraví zdrojový kód tak, že v ňom ostane už len zdrojový kód v jazyku C bez makier, v našom prípade vyhodnotí podmienku a na príslušné miesto dosadí kód, ktorý sa má vykonať, pričom ostatný sa zahodí. Tým pádom sme prostredníctvom podmienky v makre upravili program pre konkrétnu platformu.

Metaprogramovanie nie je žiadnou novinkou, umožňoval ho už napríklad jazyk Lisp začiatkom šesťdesiatych rokov prostredníctvom makier [2] . Metaprogramovanie je stále využívané i dnes, napríklad v jazyku C++ [C++01] sa zistilo až po zavedení mechanizmu šablón do jazyka, že programátor má k dispozícií metaprogramovanie. Šablóny (templates) v C++ umožňujú naproti makrám z jazyka C vytvárať metaprogramy, ktoré nepracujú so zdrojovým kódom len ako s textom, ale sú typovo bezpečné. Nové jazyky ako napríklad Ruby, podporujú metaprogramovanie ako prirodzenú techniku. Okrem toho existujú jazyky ako je jazyk Converge [3], ktoré sa na metaprogramovanie zameriavajú a rôznymi metódami ho podporujú.

Možnosť metaprogramovania je základnou požiadavkou na všeobecný programovací jazyk, ktorý by sme chceli využiť ako hosťujúci pre vytvorenie interného

[2] `http://www.se-radio.net/podcast/2008-01/episode-84-dick-gabriel-lisp`
[3] `http://www.se-radio.net/podcast/2007-05/episode-57-compiletime-metaprogramming`

doménovo špecifického jazyka.

5.2 Šablóny

Tuto nehovoríme o takých šablónach aké môžeme nájsť v jazyku C++, hovoríme skutočne o predpripravených textových formách, ktoré je možné doplniť. Šablóny sú často využívané vrámci generatívneho programovania, lebo sú pomerne jednoduché a efektívne. V tomto kontexte hovoríme o šablónach ako o hotových dokumentoch alebo ich častiach, ktoré poskytujú syntax na označenie miesta, kde bude do šablóny doplnený prvok neskôr, trebárs pomocou parametra.

Príkladom šablóny môže byť trebárs dokument predstavujúci vizuálnu prezentáciu webovej stránky, ktorý naplní webová aplikácia údajmi z databázy. Takto funguje mnoho frameworkov, lebo takýto prístup umožňuje efektívne oddeliť prezentačnú vrstvu od logickej vrstvy v zmysle architektonického vzoru MVC [4].

Ako bolo spomenuté so šablónovými systémami sa môžme stretnúť hlavne vrámci frameworkov pre webové aplikácie. Napríklad pre jazyk PHP je to Smarty Template Engine [5], Microsoft ponúka vrámci platformy .Net jazyk ASP.NET (Active Server Pages) a napokon platforma Java disponuje jazykom JSP (Java Server Pages).

V kontexte modelom riadeného vývoja softvéru sa so šablónovým systémom môžeme stretnúť v mnohých systémoch. Jedným zo zástupcov môže byť jazyk Xpand vrámci nástroja openArchitectureWare.

5.3 Doménovo špecifické jazyky

Keďže doménovo špecifické jazyky si poslednú dobu vyslúžili značnú pozornosť, presadili sa i v tejto oblasti a to najmä vďaka tomu, že je možné ich jednoducho vytvoriť, vďaka dnešným nástrojom, a zároveň umožňujú vytvárať ľubovoľné konštrukcie pre prácu s frameworkami, knižnicami a inými formami opätovne využiteľného kódu.

Zaujímavým príkladom je jazyk Ruby, ktorý umožňuje metaprogramovanie a

[4] Model View Controller `http://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=294756843`

[5] http://www.smarty.net/

ponúka i vhodné možnosti na vytváranie interných textových doménovo špecifických jazykov vďaka flexibilnej syntaxi, ktorá napríklad nevyžaduje na prístup k poliam a metódam objektu použitie bodkovej notácie, ktorá je tu nepovinná, a taktiež nevyžaduje použitie zátvoriek pri metódach s jedným argumentom. Toto umožňuje spolu s reťazením funkcií vytvárať skoro vetné konštrukcie, ktoré pripomínajú prirodzený jazyk a nie sú zaťažené syntaktickými symbolmi [6].

Úlohu doménovo špecifických jazykov ako techniky v modelom riadenom vývoji softvéru sme si už opísali v samostatnej kapitole.

[6] `http://www.se-radio.net/podcast/2007-04/episode-52-dsl-development-ruby`

Kapitola 6

Aspektovo-orientované programovanie

Aspektovo-orientované programovanie je zaujímavou paradigmou, ktorá ponúka nový pohľad na návrh systémov. Niektoré princípy tejto paradigmy je možné implementovať pomocou techník modelom riadeného vývoja softvéru, no zároveň je veľkým prínosom využitie techník aspektovo-orientovaného programovania vrámci modelom riadeného vývoja softvéru.

6.1 Záujmy

Pri návrhu systémov sa často stretávame s rôznymi záujmami (concerns), ktoré chceme vrámci navrhovaného systému uspokojiť. Týmito záujmami môžu byť funkčné i nefunkčné požiadavky, ale i konkrétnejšie záujmy technických a biznis oblastí. Záujmy možno rozdeliť na primárne a prierezové (cross-cutting), pričom primárne sú dekomponované pomocou štruktúr ponúkaných paradigmou objektovo-orientovaného programovania a konkrétnym jazykom. Naproti tomu, prierezové záujmy sú iné. Tieto svojim logickým rozsahom často ovplyvňujú viaceré objekty a rôznym spôsobom ovplyvňujú aj ich správanie. Takýmito záujmami môžu byť napríklad trasovanie, logovanie, bezpečnosť, autentifikácia, aplikácia roly vrámci návrhového vzoru na triedu a podobne.

6.2 Prierezové záujmy

Prierezové záujmy zvyknú napĺňať takzvané aspekty. Ide o jazykovú konštrukciu, ktorá je veľmi podobná triedam. Rovnako ako triedy obsahuje metódy i polia a je možné ich inštancovať. Okrem toho však obsahujú ešte i takzvané body rezu (pointcuts). Bod rezu je výraz, ktorý označuje body spojenia (join points), čo sú miesta, kde treba aplikovať nejaký kód. To samozrejme znamená, že aspekty okrem týchto výrazov obsahujú i takzvané rady (advices), ktoré opisujú kód, ktorý sa má napojiť na body spojenia.

6.3 Body rezu

Body rezu môžu byť rôzne výrazy, ktoré špecifikujú rôzne miesta, napríklad pred či po zavolaní nejakej funkcie. Okrem toho sa môžu odkazovať i na stav objektu alebo vlastnosti stacku. Taktiež rady môžu byť rôznorodé. Môžu to byť kúsky kódu, ktoré majú prístup k aktuálnemu kontextu v bode spojenia, ale môžu to byť trebárs aj nové polia či metódy pre objekt, na ktorý aspekt aplikujeme.

Aspekty nie sú úplnou novinkou, niečo podobné bolo možné dosiahnúť prostredníctvom takzvaných meta objektových protokolov. Taktiež i vrámci objektovo-orientovaného programovania bolo možné využiť návrhové vzory Zástupca (Proxy), Továreň (Factory) a Odchytenie (Interceptor) [1] na dosiahnutie podobného efektu. Aspektovo-orientované programovanie v dnešnej dobe však s pomocou dnešných jazykov umožňuje toho omnoho viac, no opis všetkých týchto techník nie je primárnym účelom tejto práce.

6.4 Aspekty pomocou modelom riadeného vývoja softvéru

Jednou z možností implementácie aspektovej orientácie v statickom ponímaní, teda na úrovni zdrojového kódu, môže byť využitie techník modelom riadeného vývoja softvéru. Modely samotné môžu predstavovať rôzne aspekty systému a spojením,

[1] Proxy `http://en.wikipedia.org/wiki/Proxy_pattern`, Factory `http://en.wikipedia.org/wiki/Abstract_factory_pattern` a Interceptor `http://bosy.dailydev.org/2007/04/interceptor-design-pattern.html`

označovaným ako prepletanie (model weaving) je možné vytvoriť model s aplikáciou modelovaných aspektov. Dobrým príkladom môže byť už len samotné UML, kde diagramy tried, interakcií a stavových automatov predstavujú rôzne aspekty triedy, v tomto prípade jej statickú štruktúru, interakciu s inými triedami a zmenu jej stavu.

To čo potrebujeme na implementáciu aspektovo-orientovaných techník pomocou modelovania je vytvoriť podobné artefakty ako vystupujú v klasickom chápaní aspektovo-orientovaného programovania. Dôležité je poskytnúť výrazový jazyk, ktorým by bolo možné opísať body rezu a teda tým určiť body spojenia vrámci modelu. Na tento účel môžu dobre poslúžiť jazyky, ktoré umožňujú vytvárať požiadavky na modely, akým je napríklad OCL, respektíve jeho implementácia Check. Rady môžu byť predstavované samotnými modelmi, pričom však treba umožniť bližšie špecifikovať, aký typ rady bude aplikovaný. Poslednou súčasťou by mal byť nástroj na vykonanie aplikácie rád na jednotlivých bodoch spojenia. Implementácia takéhoto nástroja je v podstate jednoduchou transformáciou medzi modelmi.

6.5 Prepletanie modelov

Tieto postupy umožňuje využívať napríklad nástroj Xweave [GV07] z openArchitectureWare. Ten je však čiastočne limitovaný a to na homogénne aspekty. To znamená, že pri prepletaní musí byť vždy známy hlavný model, na ktorý sa budú aplikovať rady. Heterogénne aspektovo-orientované programovanie, kde je možné spojiť viacero modelov predstavujúcich rady do modelu predstavujúceho objekt, je možné vykonávať pomocou jazyka ATL, no jeho aplikácia je v podstate ručná, čo neprináša veľký prínos. Xweave má ešte i ďalšie obmedzenie, ktorým je jediný typ rady a to aditívne pridávanie prvkov na bod spojenia. Toto však v skutočnosti nie je obmedzujúce, lebo vďaka štandardnému meta meta modelu v podobe štandardu MOF, je možné s nástrojom Xweave umiestniť na body spojenia špecifikované bodmi rezu ľubovoľné prvky vlastného meta modelu. Takže stačí vytvoriť meta model rád, namodelovať konkrétnu radu a pomocou nástroja Xweave aplikovať a v ďalšom kroku, pomocou transformácie medzi modelmi, vykonať operáciu na mieste, kde bude pr-

vok predstavujúci radu podľa jej typu.

Prepletanie modelov je možné implementovať aj bez potreby zvláštneho nástroja, ak máme k dispozícií transformácie medzi modelmi. To čo potrebujeme vykonať je totižto v podstate transformácia medzi modelmi na dvoch úrovniach. Najprv potrebujeme rozšíriť cieľový meta model o prvky meta modelu, ktorý predstavuje aspekty, respektíve rady, a následne vytvoríme inštanciu takéhoto meta modelu pomocou transformácie využivajúcej informácie z hlavného modelu a modelu aspektu.

6.6 Aspekty vrámci modelom riadeného vývoja softvéru

Techniky modelom riadeného vývoja softvéru nie sú len vhodným spôsobom ako implementovať aspektovo-orientovaný vývoj softvéru, ale taktiež je možné naopak využiť techniky aspektovo-orientovaného programovania v modelom riadenom vývoji softvéru a to hneď na viacerých úrovniach.

Prvou úrovňou je využitie aspektov v modelovaní, čo sme si v podstate opísali vyššie, viac je možné sa o tom dozvedieť v časti o najlepších postupoch a vzoroch, kde sa táto technika nazýva pohľady. Toto je veľmi všeobecný a efektívny spôsob, ktorý si vyžaduje len vytvorenie modelov aspektov, definovanie výrazov pre určenie bodov spojenia, prípadné transformácie medzi modelmi a následne prácu s možno upraveným meta modelom.

Aspekty sa môžu stať i súčasťou doménovo špecifického jazyka, ktorý vytvárame a tak môžeme ťažiť z výhod aspektovo-orientovaného programovania priamo na úrovni jazyka, bez nutnosti prepletania modelov. Tento spôsob kladie nároky na návrh doménovo špecifických jazykov, definovanie vhodných bodov rezu a vhodnú aplikáciu rád. Hoci sa môže zdať, že nepotrebujeme toľko artefaktov ako v predchádzajúcom postupe, nároky sú prinajmenšom zhodné, keďže návrh jazyka je komplikovaný, obzvlášť, ak má podporovať i aspekty.

Na ďalšej úrovni sú transformácie, presnejšie povedané výber transformácie vrámci pracovného postupu (workflow), ktorý určuje ich aplikáciu. Jednou z možností implementácie je popísať pracovný postup doménovo špecifickým jazykom, ktorý umožní aplikáciu rád, čo môže znamenať v praxi výber vhodnej transformácie.

Takto je možné napríklad vytvoriť sadu generátorov, niekedy označovanú ako zásobník (cartridge), ktorá sa aplikuje na pracovný postup a v podstate rozhodne o cieľovej platforme.

Napokon najnižšou úrovňou je samotný kód, ktorý môže byť v programovacom jazyku podporujúcom aspektovo-orientované programovanie a my môžeme prostredníctvom generátorov vytvárať aspekty a nechať ich aplikáciu na cieľovú platformu.

Takéto postupy umožňujú využívať techniky aspektovo-orientovaného programovania a vývoja softvéru, na vyššej úrovni abstrakcie, nezávisle od konkrétneho jazyka či paradigmy. Všetko čo na to potrebujeme sú nástroje, ktoré bývajú štandardnou súčasťou modelom riadeného vývoja softvéru a v prípade, že to tak nie je, je možné ich pomerne jednoducho dostupnými prostriedkami implementovať.

Aspektovo-orientované techniky umožňujú pracovať s prierezovými záujmami prirodzene v prostredí modelom riadeného vývoja softvéru, čím je možné efektívne vytvárať metaware predstavujúci knižnice aspektov.

Jednou z veľmi dôležitých oblastí, kde sa aspekty vrámci modelom riadeného vývoja softvéru dostávajú k slovu je inžinierstvo produktových línií, kde sú aspektovo-orientované programovanie a modelom riadený vývoj softvéru vhodnými implementačnými technikami.

Kapitola 7

Inžinierstvo produktových línií

O inžinierstve produktových línií sa často hovorí ako o softvérovom inžinierstve robenom poriadne. Hlavným dôvodom je zrejme fakt, že problémy, s ktorými sa v tejto oblasti stretávame, sú do značnej miery problémami softvérového inžinierstva ako takého a teda v mnohom je možné využiť metódy inžinierstva produktových línií v softvérovom inžinierstve ako takom a naopak metódy softvérového inžinierstva v inžinierstve produktových línií.

Je to pomerne mladá oblasť, ktorá sa zaoberá produktovými líniami, ich vytváraním, správou, rozširovaním, variabilitou a efektívnym opätovným využívaním súčastí naprieč produktovými líniami.

Požiadavky na vznik tejto oblasti nevychádzajú priamo len od softvérových inžinierov, ale i z produktového manažmentu. Častou požiadavkou produktových manažérov totiž býva umožniť produkt upraviť na mieru zákazníkovi tak, aby bolo potenciálnych zákazníkov čo možno najviac. To zároveň znižuje náklady na vývoj, lebo veľká časť produktu ostáva nemenná. Takto je možné nielen osloviť veľkú množinu zákazníkov, ale taktiež prispôsobiť produkt pre potreby práve jedného konkrétneho zákazníka. Produktová línia sa naviac môže v čase vyvíjať a poskytovať tak vhodný priestor i pre marketing.

Potreba produktových línií a následne i ich správy a teda inžinierstva produktových línií prišla predovšetkým z technických oblastí vývoja softvéru. Neraz je vhodné, potrebné či priam nevyhnutné mať na riešenie často sa opakujúcich problémov k dispozícií platformu, ktorá je opätovne použiteľná a teda jej použitie

šetrí prostriedky potrebné na vývoj a poskytuje dostatočnú flexibilitu na prispôsobenie pre nasadenie v konkrétnom projekte. Takáto platforma je prvým krokom k vytvoreniu produktovej línie.

7.1 Produktová línia

Pod pojmom produktová línia rozumieme viacero produktov, ktoré majú mnohé svoje súčasti podobné a odlišujú sa v zásade len v istých aspektoch. Ako jednoduché kritérium pre produktovú líniu by sme mohli povedať, že dva produkty sú súčasťou línie ak zdieľajú približne osemdesiat percent zdrojového kódu. Príkladom produktovej línie môže byť trebárs softvér pre automobily, ktorý môže fungovať stále rovnako, no variabilita spočíva v dostupnom hardvéri ako napríklad klimatizácia, GPS navigácia, elektrické okná, komponenty rôznych výrobcov a podobne, pre ktoré je nutné zaradiť naviac do produktu špecifický komponent.

Pre produktové línie je dôležité zabezpečiť možnosť uchovania konfigurácie konkrétneho produktu a taktiež možnosť aktualizovať produkt na základe nových verzií jednotlivých komponentov, ktoré tvoria danú konfiguráciu.

7.2 Model vlastností

Model vlastností predstavuje body variability v produktovej línií a ich vlastnosti. Bod variability je miesto, ktoré môže byť v danom systéme pre rôzne produkty nejakým spôsobom odlišné. Model vlastností tieto miesta a ich variabilitu opisuje. Body variability môžu určovať povinné položky, voliteľné položky a tiež položky s výlučným výberom. Body variability môžu predstavovať napríklad rôznu funkčnosť. Príkladom povinnej položky v bode variability môže byť napríklad jadro systému, voliteľná môže byť napríklad podpora importu rôznych formátov súborov a napokon výlučný výber môže predstavovať napríklad voľbu technológie pre prezentačnú vrstvu webovej aplikácie.

Okrem takýchto bodov variability by mal model vlastností podporovať aj definovanie zložitejších vzťahov medzi nimi ako napríklad závislosti, ktoré môžu byť

zachytené vrámci tohto modelu alebo v externom modeli. Príkladom môže byť voľba technológie, ktorá si zároveň bude vyžadovať i voľbu istej systémovej platformy.

7.3 Konfiguračný model

Konfiguračný model je inštanciou modelu vlastností, teda ten považujeme za jeho meta model, ktorý definuje konkrétny produkt a teda konfiguráciu produktu z produktovej línie na základe výberu konfigurácie z modelu vlastností. Konfiguračný model by mal podporovať nielen typ bodu variability, ale aj závislosti. Nástroje umožňujúce vytvárať konfiguračné modely by mali zabezpečiť konzistentnosť modelu vzhľadom na závislosti definované vrámci modelu vlastností. Teda napríklad ak výber istej voľby zamedzí výber inej voľby, nemala by tá byť viac k dispozícií.

7.4 Základné položky

Pri vývoji viacerých produktov sa často stáva, že narazíme na to, že programujeme znova to isté, respektíve niečo veľmi podobné ako sme už programovali v inom projekte. Vtedy má zmysel uvažovať o vytvorení knižnice či frameworku. Častokrát sa však použitie frameworku v rôznych produktoch líši a variabilita narastá a s tým sa komplikuje i návrh frameworku. Komplikovaný návrh na oplátku môže spôsobiť i náročnejšie použitie. Vtedy má zmysel vytvoriť základné položky produktov, teda akési komponenty, ktoré by neboli len knižnicami či frameworkami, ale v ideálnom prípade by umožňovali väčšiu variabilitu a flexibilitu. Základné položky je náročné vytvárať a je nutné ich spravovať a neustále vyvíjať. Takéto základné položky si vyžadujú aj prácu na rôznych produktoch tak, aby položky boli použiteľné vo viacerých produktoch a taktiež konzistentné s predchádzajúcimi verziami. Z tohto vyplýva aj to, že je potrebné spravovať verzie základných položiek vrámci konfigurácií.

7.5 Produktové línie pomocou modelom riadeného vývoja softvéru

Keďže vrámci inžinierstva produktových línií sa využívajú modely, ktoré nie je ťažké implementovať pomocou štandardného meta meta modelu, je taktiež možné spracovávať model vlastností ako meta model a konfiguračný model ako jeho inštanciu a využiť nástroje, ktoré podporujú modelom riadený vývoj softvéru. Správu konfigurácií môžeme teda zabezpečiť pomocou modelov, základné položky a ich zostavenie podľa konfigurácie následne pomocou transformácií a generátorov. Závislostí je možné namodelovať prostredníctvom obmedzení rovnako ako v prípade iného meta modelu, prípadne môžeme vytvoriť pre ne samostatný meta model. Pre správnu aplikáciu základných položiek, ktoré môžu byť predstavované ako metaware je potrebné využívať techniky aspektovo-orientovaného programovania. Tuto nám pomôžu techniky z predchádzajúcej kapitoly.

Kapitola 8

Najlepšie postupy a vzory

Posledných päťnásť rokov sa do popredia pozornosti softvérových inžinierov dostali vzory. Tie v oblasti vývoja softvéru chýbali už dlhšiu dobu a boli jedným z faktorov, ktoré odlišovali softvérové inžinierstvo od ostatných inžinierských disciplín. To sa však v uplynulých rokoch rapídne zmenilo a tento trend sa ukázal ako dôležitý a zásadný pre rozvoj softvérového inžinierstva. Preto by som chcel uviesť niekoľko vzorov a najlepších postupov, respektíve ich návrhy, ktoré čerpám z externých zdrojov [Vö08, VB04, Vö09] a reakcie na ne [1], kde ich nájdete viac a podrobnejšie opísané. Preložené názvy vzorov sú pre prehľadnosť zvýraznené tučným písmom.

8.1 Proces a organizácia

Modelom riadený vývoj softvéru má významný dopad na vývojový proces a i vplyv na organizáciu, v ktorej vývoj prebieha. Dôvodom je fakt, že v súlade s inžinierskými princípmi je softvér vytváraný pomocou nástrojov výrobnej linky, ktorú však treba tiež pripraviť. V kontexte modelom riadeného vývoja softvéru a vývoja softvéru ako takého, je výrobnou linkou metaware, teda sada generátorov, transformácií, šablón, editorov a tak ďalej, ktoré sú využívané na vytvorenie konkrétnej aplikácie.

Je vhodné riadiť sa pravidlom **nechaj ľudí robiť to v čom sú dobrí**. Toto pravidlo hovorí o tom, že treba využiť znalosti technologických špecialistov a zachytiť ich pomocou šablón, rovnako tak využiť aj znalosti softvérových architektov,

[1] `http://www.infoq.com/articles/model-driven-dev-best-practices`

analytikov a doménových špecialistov a tie zachytiť do doménovo špecifického jazyka a do modelov, ktoré pomocou neho budú vytvárať aplikačný vývojári. Veľmi vhodné môže byť tieto úlohy čas od času obmeniť, aby autori generátorov rozumeli požiadavkám aplikačných vývojárov, ktorí ich generátori využívajú a podobne.

To si vyžaduje oddelený vývoj jednotlivých komponentov, no zároveň je potrebné previazať metaware s požiadavkami aplikačného vývoja, ktorý ho využíva. Preto je riešením **iteratívny vývoj s dvoma stopami**. Myšlienkou je teda iteratívny vývoj, čo umožňuje okrem iného vyvíjať metaware i aplikáciu agilnými technikami. Každý krok iteraácie by mal pozostávať z paralelného vývoja, ktorého výsledkom je metaware a aplikácie, keďže nie je potrebné, aby tí istí ľudia pracovali na oboch súčastiach. Zároveň je však treba definovať fázu integrácie, aby mohol byť metaware s novou funkcionalitou poskytnutý aplikačným vývojárom.

Aby bol takýto paralelný vývoj možný, stopa sústreďujúca sa na metaware musí byť jednu iteráciu popredu a to sa dosiahne **vyňatím infraštruktúry**, čo môže znamenať napríklad, že sa vytvoria šablóny na generovanie z existujúcich aplikácií alebo vytvorených prototypov.

Na samotné začatie však potrebujeme ešte najprv pochopiť biznis zákazníka a jeho koncepty, ktoré sú však zároveň i dôležitými konceptami, s ktorými by mali pracovať doménovo špecifické jazyky, ktoré na zachytenie modelov využijeme. Tento postup sa nazýva **vytváranie jazyka paralelne s konceptami**.

S takýmto iteratívnym postupom následne súvisía i ďalšie vzory. Napríklad **nákupný košík s pevným rozpočtom** hovorí o tom, že iterácie by mali byť krátke a na každú by mal byť vyhradený zákazníkom istý rozpočet. Pravidelne by mali byť zostavované čiastkové aplikácie, trebárs každé tri týždne.

Obchodovanie s rozsahom je odporúčanie pre fázu začatia iterácie, ktoré hovorí, že je potrebné pevne stanoviť ciele iterácie na základe dohody vývojárov pre metaware, aplikáciu a zákazníka. Toto umožňuje zákazníkovi riadiť postup vývoja a kontrolovať cenu každej iterácie i funkcionalitu, ktorú sled iterácií prinesie.

Zakončením každej takejto iterácie by malo byť **schválenie iterácie**. To by malo zaručiť, že zákazník je oboznámený s postupom vývoja. V ideálnom prípade, kedy by bol zástupca zákazníka súčasťou vývojového tímu, by bolo vhodné, ak by

výsledky vývoja vrámci danej iterácie prezentoval zákazníkovi on sám. Následne by architekt a zákazník mohli prezentovať novo vzniknuté požiadavky.

Netreba zabúdať ani na priebežné **hodnotenia** zdrojových kódov písaných v doménovo špecifických jazykoch. To umožní nájsť časté chyby, lepšie špecifikovať obmedzenia a prípadnú úpravu jazyka či editoru.

Metaware ako produkt má v modelom riadenom vývoji softvéru svoje miesto. To si však vyžaduje plány vydaní, spravovanie zdrojových kódov, riešenie problémov a odstraňovanie chýb.

Ak má metaware slúžiť ako produkt, je nevyhnutná i **kompatibilná organizácia**, ktorá by umožnila medziprojektovú prácu. To je nevyhnutné, ak chceme využiť plný potenciál, ktorý takýto metaware môže priniesť, a využívať ho vo viacerých projektoch, kde môžu byť požiadavky a problémy trocha iné čo si môže vyžadovať zásahy, úpravy a opravy a teda i ľudí, ktorí vyvíjajú metaware s ohľadom na viaceré projekty.

8.2 Doménové modelovanie a doménovo špecifické jazyky

Pre nasadenie modelom riadeného vývoja softvéru je potrebné navrhnúť i vhodný doménovo špecifický jazyk. Návrh jazyka sa skladá z dvoch dôležitých aspektov. Prvým z nich je analýza domény, konceptov a abstrakcií, ktoré je treba pomocou doménovo špecifického jazyka zachytiť. Druhým krokom je návrh konštrukcií jazyka čo si vyžaduje netriviálne úsilie.

Formálny meta model umožňuje vytvoriť abstraktné koncepty, ktoré je možné použiť na opis problémovej oblasti a taktiež umožňuje formálne zachytiť a ďalej spracovávať modely. Meta model teda prináša dve veľké výhody a to v podobe konkrétnej formálnej podoby konceptov, ktoré sú pre nejakú doménu dôležité a taktiež umožňuje vytvoriť metaware na spracuvánie modelov.

Vhodné **zdroje pre jazyk** je možné nájsť na viacerých miestach. Pokiaľ ide o doménovo špecifické jazyky, môže to byť napríklad použitá architektúra, terminológia architektonických vzorov, ale tiež i frameworky a knižnice. Niekedy i biznis domény ponúkajú dobre formalizovateľné koncepty, najmä ak má daná doména

formálne základy. V ostatných prípadoch je potrebné vykonať doménovú analýzu, opísať požiadavky a využiť skúsenosti či existujúcu aplikačnú infraštruktúru.

Nielen abstraktný model je podstatný, treba si uvedomiť, že najmä pri využití doménovo špecifických jazykov v rôznych biznis doménach je podstatné pamätať na slová **notácia, notácia, notácia**, lebo častokrát využívanie jazyka nie je úplne prirodzené a častokrát v danej doméne už môže nejaká forma notácie existovať.

Jednou z prvých otázok, ktoré si pri návrhu doménovo špecifického jazyka budete klásť môže byť, **grafická alebo textová notácia**? Obe majú svoje výhody i nevýhody, ktoré treba vziať v úvahu. Dobrým riešenim môže byť i elegantná kombinácia v podobe textového doménovo špecifického jazyka na vytváranie modelov a grafickej reprezentácie na ich efektívnu vizualizáciu.

Na efektívny opis systémov treba umožniť naň rôzne **pohľady**. Pohľad môže predstavovať nejaký aspekt, teda nejaký uhol náhľadu na systém, ktorý je pre konkrétne potreby najvhodnejší. Na vytvorenie pohľadov je možné vytvoriť rôzne doménovo špecifické jazyky alebo vytvoriť v jednom jazyku akési sekcie predstavujúce daný pohľad. Na pamäti treba však mať, akým spôsobom vytvárame medzi pohľadmi závislosti, akým spôsobom sa budú pohľady kombinovať a treba tiež eliminovať redudanciu zmenšovaním prekryvov pohľadov a tým aj odstranením nutnosti synchronizácie.

Nástroje nie sú ľubovoľne škálovateľné a preto je potrebné vytvárať **partície**. Partície sú jednotky kódu doménovo špecifických jazykov, ktoré umožňujú dosiahnúť potrebnú škálovateľnosť a štruktúrovateľnosť. Partície by mali umožňovať referencovať časti inej partície, čo si samozrejme vyžaduje vytvorenie mechanizmu referencií a vkladania jednotiek v jazyku.

Dobrým spôsobom vytvárania doménovo špecifických jazykov je **učiť sa od jazykov tretej generácie**. Obzvlášť užitočnými konceptami sú priestory mien, rozsah platnosti, viditeľnosť, typové systémy a inštancovanie, špecializácia a pojem abstraktnosti (v zmysle abstraktných tried). Okrem iného je odporúčané poznať celkovo zásady návrhu programovacieho jazyka a rôzne programovacie paradigmy.

Odpovedaním na otázku **kto sú občania prvej triedy** je možné nájsť základné koncepty navrhovaného jazyka. Prv sa však treba zamyslieť nad tým, či chceme

vytvoriť jazyk s mnohými špecifickými konceptami alebo jazyk s menším počtom konceptov, ktoré sú o to výrazovejšie. Mnohé koncepty sú obzvlášť vhodné pre biznis domény, lebo umožňujú poskytnúť ľahko pochopiteľné koncepty, naproti tomu, málo konceptov síce znamená architektúru mikrokernelu [2], no koncepty sú náročnejšie a preto sa skôr takýto prístup hodí pri návrhu technicky orientovaných jazykov.

8.3 Architektúra nástrojov

Architektúra nástrojov pre modelom riadený vývoj softvéru je pomerne komplikovaná a nie je úplne jasné, aké prístupy sú najlepšie, lebo mnohí výrobcovia pristupujú k problematike výrazne odlišne. Následujúce praktiky sú však široko aplikovateľné.

V prvom rade je potrebné poskytnúť podporu pre **implementovanie meta modelu**. To znamená, že pre modely vytvárané na základe konkrétneho meta modelu treba ponúknuť možnosť kontroly obmedzení a korektnosti modelu vzhľadom na meta model.

Nástroje by mali **ignorovať konkrétnu syntax** doménovo špecifického jazyka a pracovať len s modelmi. Takto je možné využiť transformácie, ktoré nemusia rátať s komplexnosťou konkrétnej syntaxe. Konkrétna syntax je najprv parsovaná a je vytvorený model v meta objektovej reprezentácií. Takýto model je možné transformovať do iných meta objektových reprezentácií a následné je možné z meta objektovej reprezentácie transformovať do konkrétneho jazyka. Transformácie sú potom nezávislé od konkrétnej syntaxe, čo zvyšuje ich opätovnú využiteľnosť.

Pre reálny vývoj je nevyhnutná **podpora tímovej spolupráce**. To obnáša správu verzií, vytváranie vetiev, zamykanie, porovnávanie a spájanie. V tomto majú značnú výhodu textové doménovo špecifické jazyky, ktoré je možné integrovať s existujúcimi nástrojmi na správu zdrojových súborov.

Nie je možné predpokladať, že doménový špecialisti budú zvyknutí pracovať s vývojovým prostredím a ani to, že by vývojári oželeli pohodlie, ktoré im poskytujú moderné vývojové prostredia. **Na nástrojoch záleží** a to znamená, že treba

[2] `http://www.vico.org/pages/PatronsDisseny/Pattern%20MicroKernel/`

poskytnúť pre doménovo špecifické jazyky editory, ktoré ponúkajú funkcie ako je automatické dopĺňanie kódu, zvýraznenie syntaxe, navigáciu a prehľadávanie, refaktoring, prehľady a aj už vyššie spomenutú tímovú spoluprácu. Samozrejme toto neplatí len o editoroch pre doménovo špecifické jazyky, ale i pre nástroje pomocou ktorých vyvíjame metaware.

8.4 Vývoj aplikačnej platformy

Aplikačná platforma predstavuje prostriedky na vývoj aplikácií v kontexte modelom riadeného vývoja softvéru, čo zahŕňa predovšetkým metaware.

V prvom rade je vhodné fyzicky **oddeliť generovaný a ručne písaný kód**. Tým sa zabráni nežiadúcemu pregenerovaniu a problémom so správou verzií a chránených oblastí. To možno dosiahnúť využitím vhodného návrhu a návrhových vzorov. Rozhrania, abstraktné triedy, Továreň, Stratégia, Most či Šablónová metóda môžu byť obzvlášť užitočné návrhové vzory pre tento účel.

Bohatá doménovo špecifická platforma, v podobe doménovo špecifických frameworkov, môže značne uľahčiť vývoj transformácií v extrémnom prípade eliminovať ich význam. Vhodným riešením môže byť využitie konceptov a funkcionality poskytnutých frameworkom v doménovo špecifickom jazyku, ktorý ich bude využívať a svojim spôsobom riadiť framework.

Oddelením **technických subdomén**, respektíve technických aspektov, je možné vytvoriť jednak vhodné doménovo špecifické jazyky na ich opis, ktoré môžu predstavovať čiastočne sa prekrývajúce pohľady, ktoré je vďaka ignorovaniu konkrétnej syntaxe možné skombinovať, a zároveň tieto modely technických subdomén efektívne opakovane využívať pri vývoji ďalších aplikácií.

Generovaný kód nie je vhodné upravovať kvôli rôznym problémom, ktoré to spôsobuje. Sú však tri zaužívané možnosti riešenia. Využitie chránených oblastí v kóde, ktoré môžu označovať ručne dopísaný kód, ktorý nemá byť nahradený generovaným, býva obtiažne kvôli zložitým pravidlám ochrany. Ak nie sú pravidlá aplikované rozumným spôsobom môžu vyústiť do vzniku takzvaných sedimentov, teda oblastí, ktoré sú chránené, nebudú z kódu odstránené, no nie sú v ňom ani

využívané. Ďalšou možnosťou je využiť body rozšírenia, ktorými môžu byť rôzne konštrukcie kompozície v cieľovom jazyku, rôzne návrhové vzory, spätné volania alebo využitie aspektovo-orientovaného programovania. Treťou možnosťou je oddeliť ručne písaný kód fyzicky, buď do iného súboru alebo ho zadávať priamo ako súčasť modelu.

O generovaný kód sa je treba starať, lebo s najväčšou pravdepodobnosťou je možné, že v istom momente ho bude treba čítať či odstraňovať z neho chyby. Preto je podstatné generovať ku kódu i komentáre, ak je to možné, dodržiavať pravidlá pomenúvania a generovať dobre vyzerajúci kód, trebárs pomocou formátovacích nástrojov ako posledného kroku behu transformácie. V prípade, že všetok kód aplikácie je generovaný a generátory sú dostatočne otestované, tento postup nemusí byť nutný.

Jednotlivé pohľady na systém si môžu vyžadovať rôzne spôsoby spracovania, preto je dôležité, aby proces **spracovania si uvedomoval jednotlivé pohľady**. Niekedy je výhodnejšie z pohľadu generovať, no niekedy je lepšie niektoré pohľady nechať spracovať inak. Príkladom môže byť využitie frameworkov na interpretáciu stavových automatov a využitie generovania na iné pohľady. Niekedy môže záležať aj na poradí generovania, napríklad ak jedným z pohľadov opisuje typový systém pre ostatné.

Na generovanie kódu sa často využívajú **šablóny, o ktoré sa je treba starať**. Časom sú totiž šablóny zložitejšie a komplexnejšie. Preto je vhodné šablóny rozdeliť na viaceré časti, ktoré predstavujú funkčné celky, ktoré sa navzájom volajú. Zložitejšie, opakujúce sa časti šablón je možné vyčleniť ako knižničné funkcie. V neposlednom rade tu je ešte možnosť, využiť techniky aspektovo-orientovaného programovania na vhodnú modularizáciu šablón. Jednou z možností zjednodušenia je aj už skôr spomínaná bohatá doménovo špecifická platforma. Prehľadnosť šablón je možné zvýšiť aj ich prispôsobením jazyku transformácií, pričom pri generovaní využijeme pre generovaný kód nástroj na automatizované formátovanie.

Zložitosti generátorov sa môžeme vyhnúť použitím **transformácií modelov na zjednodušenie generátorov**. Napríklad ak sa nám podarí vytvoriť niektoré koncepty doménovo špecifického jazyka prostredníctvom iných, základných elementov

toho istého jazyka, je možné najprv model transformovať do takého, ktorý bude obsahovať len základné koncepty a tým odbremeniť generátor.

Výhody modelom riadeného vývoja softvéru sú tým väčšie, čím viac je možné metaware znova použiť. Preto je dôležité **umožniť jeho prispôsobenie**, predovšetkým pre doménovo špecifické jazyky, generátory a modely. Tu sa zasa priam ponúkajú techniky aspektovo orientovaného programovania a ich možné ekvivalenty vytvorené pomocou polymorfizmu a návrhových vzorov Factory, Interceptor a podobne.

Pri modelom riadenom vývoji softvéru **netreba zabúdať na testovanie**. To sa deje na viacerých úrovniach. Kontrola obmedzení meta modelu na modeloch je prvou možnosťou. Je zbytočné kontrolovať syntax vygenerovaného kódu, generátory lepšie preveria testy jednotiek, ktoré overia sémantiku kódu. Transformácie modelov je možné testovať prostredníctvom obmedzení, ktoré môžu slúžiť ako testy výsledkov transformácií. Na testovanie generátorov počas ich vývoja môže dobre slúžiť testovací model, ale treba si dať pozor na pokrytie.

8.5 Otvorené problémy

Vrámci modelom riadeného vývoja softvéru existuje viacero nezrovnalostí a nejednoznačností, niektoré sú však obzvlášť dôležité a hodné spomenutia, aby bolo možné tieto techniky využiť čo možno najlepšie.

Kombinácia notácií dnes nie je možná, pretože nástroje to dostatočne neumožňujú. Napríklad by mohlo byť žiadúce vytvárať grafické modely a umožniť pridávať k ním vo forme komentárov časti kódu, pričom by pri jeho písaní užívatelia mali k dispozícií plnohodnotný editor. Naopak pri využívaní textovej notácie by sa často zišli grafické symboly, klasickým príkladom je azda editovanie matematických vzorcov.

Veľké výhody by mohla priniesť modulárnosť a možnosť kompozície jazykov. Bolo by možné prostredníctvom knižníc komponentov vyskladať doménovo špecifický jazyk s konštrukciami a sémantikou, ktoré by boli jednak dobre definované a jednak by boli znovupoužiteľné. Problémom však je, že dnešné parsovacie algoritmy kompozíciu jazykov neumožňujú. Jedným riešením môže byť ukladanie textových in-

formácií v podobe štruktúrovaných metadát, tento postup využíva napríklad systém MPS [3].

Hoci na to neexistuje zrejmý dôvod, nástroje zväčša nepodporujú refaktorovanie metaware-u, čo by značne uľahčilo prácu s ním.

Pri kombinácií generovaného a ručne písaného kódu nie je úplne jasné čo sa má stať, ak sa model, z ktorého generujeme zmení a ručne písaný kód je závislý od predchádzajúceho modelu. Toto spôsobuje vznik už skôr spomenutých kódových sedimentov. Ideálne by bolo zabezpečiť s refaktorovaním modelu aj automatické refaktorovanie ručne písaného kódu tak, aby ho nebolo treba meniť.

Podobný problém nastáva vo vzťahu medzi doménovo špecifickým jazykom a modelom. Akým spôsobom by mala byť vykonaná automatizovaná migrácia modelov? Akým spôsobom zmeniť model keď sa zmení jazyk na základe ktorého vznikol? Je možné nekompatibilné časti modelu označiť, no to spôsobí vznik modelových sedimentov. V každom prípade zahodenie vytvoreného modelu kvôli zmene jazyka je neakceptovateľné.

Veľmi žiadúce by bolo debuggovanie na úrovni modelu, no to je vo všeobecnosti problém. Na to, aby to bolo možné je okrem iného treba, aby model mal jednoznačnú sémantiku a to súčasné štandardy neposkytujú. V tejto oblasti existujú napríklad snahy vytvoriť podmnožinu sémanticky jednoznačného UML [4], no štandard nie je ešte hotový a je otázne či by takáto úprava nebola príliš reštriktívna.

Ako bolo skôr spomenuté, možnosti spracovania modelov sú v zásade dvojaké, buď generovanie alebo interpretovanie. Otázne však ostáva či by nebolo možné selektívne z modelov generovať potrebné súčasti, ktoré by bolo interpretovať náročné, a ostatné interpretovať. Existuje vo svete modelov paralela ku kompilácií v pravom čase (just in time compilation)?

Dnešné nástroje na prácu s modelmi nie sú dostatočne škálovateľné. Spracovanie veľkých alebo mnohých modelov je dosť náročné. Treba teda vyriešiť problém s navigáciou po veľkých modeloch a vyhľadávanie. Prínosom by hádam mohol byť

[3] Meta Programming System, `http://www.jetbrains.com/mps/`

[4] Executable UML: Foundation alebo tiež fUML, `http://www.omg.org/docs/ad/06-06-16.pdf` a `http://portal.modeldriven.org/content/foundational-uml-reference-implementation`

projekt EMF Index [5]. Problémom však ešte stále ostáva vizualizácia veľkých modelov.

Mnohé nástroje disponujú takzvanými zásobníkmi, čo sú akési moduly generátorov. Problémom je, že neexistuje štandard, ktorý by opisoval ako majú fungovať, ako ich medzi sebou kombinovať, ako vytvárať medzi nimi rozhrania a ani ako spravovať implicitné závislosti, ktoré medzi nimi vznikajú prostredníctvom generovaného kódu. Pokus o praktické riešenie tohto problému pre rodinu nástrojov postavených na technológií EMF predstavuje projekt MWE [6].

[5] EMF Index `http://www.eclipse.org/proposals/emf-index/`
[6] Modeling Workflow Engine `http://www.eclipse.org/modeling/emft/?project=mwe`

Záver

Modelom riadený vývoj softvéru je súborom techník, ktoré vychádzajú z potrieb a princípov, ktoré sú už dostatočne dlho známe, no poskytuje pre ne vhodný rámec a umožňuje ich efektívne kombinovanie a pridáva k ním viacero praktík osvedčených vrámci reálnych projektov.

Tieto postupy prinášajú mnohé markantné výhody, ktoré stále ešte čakajú na objavenie a uplatnenie v širšom merítku, preto je potrebné túto oblasť rozvíjať.

V mnohých oblastiach vývoja softvéru možno dnes pozorovať snahu o automatizáciu a tak je vhodné v rozumnej miere automatizovať i úkony programovanie, ktoré sa často opakujú. Modelom riadený vývoj softvéru toto umožňuje.

Cieľom tejto práce bolo vytvoriť prehľad v danej problematike a nových trendoch, čo sa mi verím na týchto stranách podarilo. Čerpal som z odporúčanej literatúry, štandardov, mnohých článkov, prednášok a prezentácií i rozhovorom s odborníkmi zachytenými prostredníctvom podcastov. V priebehu tochto štúdia som si rozšíril okruh vedomostí z oblasti softvérového inžinierstva, čo mi pomohlo lepšie rozumieť potrebám a požiadavkám softvérových vývojárov. Taktiež som nadobudol materiály k tejto problematike i z akademickej obce či už ide o prednášky alebo diplomové práce z nemeckých a rakúskych univerzít. Všetky zistenia som sa snažil predať i vedúcemu bakalárskej práce a konfrontovať ich s jeho skúsenosťami z praxe. Zoznámil som sa i s viacerými nástrojmi a princípmi, ktorými sa riadili ich autori pri implementovaní techník modelom riadeného vývoja softvéru. Nadobudnuté vedomosti som napokon zhrnul do prezentácie, ktorú som formou prednášky predviedol pred publikom zloženým zo softvérových vývojárov.

Literatúra

[AN05] Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, chapter 25. Introduction to OCL. Addison-Wesley, 2nd edition, 2005.

[C++01] *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ In-Depth Series. Addison-Wesley, 2001.

[Fow04] Martin Fowler. *UML Distilled: A brief guide to the standard modeling language*. Object Technology Series. Addison-Wesley, third edition edition, 2004.

[GV07] Iris Groher and Markus Völter. Xweave: Models and aspects in concert. *voelter.de*, 2007.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architectrure: Practice and Promise*. Object Technology. Addison-Wesley, 2003.

[MSUW04] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Object Technology. Addison-Wesley, 2004.

[VB04] Markus Vőlter and Jorn Bettin. Patterns for model-driven development, 2004.

[Vö08] Markus Völter. Md* best practices. *voelter.de*, 2008.

[Vö09] Markus Völter. Dsl/mdsd best practices. Conference: JAX 2009, April 2009.