VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Modelom riadená architektúra
Model driven architecture

2009 Stanislav Toman

Zadanie

Prehlásenie

„Prehlasujem, že som tuto bakalársku prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.“

_____	_____
Dátum	Podpis

Poďakovanie

Týmto by som chcel poďakovať svojmu vedúcemu Bakalárskej prace, ktorým bol Ing. Jan Kožuszník Ph.D. za vzorne vedenie a trpezlivosť.

Abstrakt

Cieľom prace je analýza a popis metodológie MDA. Práca popisuje základne črty metodológie, podrobne zobrazuje jednotlivé fázy vývojového procesu. Podrobnejšie sa pozastavuje aj pri problematike tvorby modelov v jazyku UML a rozoberá ich druhy. Zobrazuje technológie použite pri tvorbe transformácii. Časť prace je venovaná aj popisu dostupných vývojových nástrojov. Podrobne rozoberá voľne dostupný vývojový nastroj AndroMDA. Vývojový proces demonštruje na vývoji demonštračnej aplikácie, vyvinutej pomocou metodiky MDA v prostredí AndroMDA. V závere prace je zhrnutý aktuálny stav používania metodiky na svetovom trhu IT.

Kľúčové slova

MDA, Modelovo orientovaná architektúra, OMG, PIM, PSM, Model, Transformácie, UML, XMI, QVT, AndroMDA

Abstract

Object of the thesis is a describtion and an analysis of the MDA methodology. Thesis describes main features of the methodology and depicts the phases of the development process. The description of creating models with UML language and its diagrams is also included. Thesis also describes some of the common used modeling tools and a deeper analysis of an open-source tool AndroMDA. The development process is described on the demo application developed in an environment based on the AndroMDA tool.

Keywords

MDA, Model driven architecture, OMG, PIM, PSM, Model, Transformation, UML, XMI, QVT, AndroMDA

Zoznam použitých skratiek

API – Application programming interface
UML – Unified modelling language
XML – Extensible markup language
JSP – Java server pages
IIS - Internet information services
VS – Microsoft Visual Studio 2005

Obsah

Úvod

Pravdepodobne nikdy neboli informačné systémy tak v popredí záujmu, ako sú teraz. Spoločnosti už nevyužívajú svoje systémy iba pre správu rôznych dát a dokumentov, ale kladu na ne ďaleko väčšie požiadavky. S rozvojom informačnej infraštruktúry značne pribudlo funkcionalít, ktoré nám systémy ponúkajú. Môžeme pozorovať trend, pri ktorom sa systémy starajú o automatizáciu niektorých procesov, dohliadajú na logiku fungovania spoločnosti, uľahčujú správu ľubovoľných dát či poskytujú rozhranie pre komunikáciu s inými externými systémami. Toto všetko robí zložitejším nielen samotný software, ale i vývoj a predovšetkým návrh systému.

Súčasné prístupy k návrhu systémov poskytujú metódy, ako vyvíjať kvalitné systémy za použitia moderných technológii. Jednými z posledných sú napríklad metódy vývoja po komponentoch alebo servisne orientovaný prístup. Techniky návrhu sa vyvíjali spoločne s technológiami a rastúcimi nárokmi na software. V súčasnosti existuje veľa tzv. CASE nástrojov, ktoré dizajnérom uľahčujú prácu. Tieto nástroje už väčšinou neposkytujú len grafický editor pre znázornenie navrhovaného systému, ale ponúkajú podporu profilovania vzhľadom na metodiky a technológie, generovanie systémovej dokumentácie a ďalšie.

Nové technológie prinášajú aj negatívne stránky. Ich rýchly rozvoj stavia spoločnosti pred zásadnú otázku, či adaptovať existujúce systémy na nové technológie a platformy, alebo zachovať súčasnú konfiguráciu a riskovať, že konkurencia získa kvôli tejto voľbe určitú výhodu.

Preto môžeme pozorovať, že stále väčšie percento vývojárov systémov kladie dôraz na ľahkú nahraditeľnosť jednotlivých komponentov a využívaných frameworkov. Tím sa systém stáva jednoduchšie udržiavateľným a ľahšie sa dajú riešiť problémy spojené s požiadavkami na jeho neustály rozvoj.

Na druhej strane môžeme samozrejme pozorovať zvyšujúcu sa uniformnosť a opakovanie implementačných prístupov prevažne k zvolenej platforme. Medzi ne patrí napríklad využitie návrhových vzorov alebo rôznych frameworkov pre jednotlivé časti systému.

Združenie OMG (Object Management Group) navrhlo modelom riadenú architektúru (Model Driven Architecture) práve ako štandard pre vývoj software, ktorý by mal riešiť predovšetkým otázky prenositeľnosti systému, jeho udržiavatelnosti a produktivity vývojového procesu.

Myšlienka presunutia ťažiska vývoja, nachádzajúceho sa prevažne na úrovni kódu, do úrovne modelovania nie je nová, ale až združenie OMG ustanovilo MDA ako štandard.

KAPITOLA 1
ČO JE TO MDA

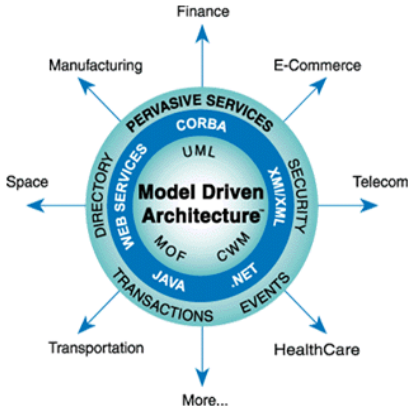
Prvá špecifikácia MDA sa objavuje na konci roku 2000, kedy sa OMG rozhodlo vytvoriť tým, ktorý by celú metódu popísal formálnejšie. S využitím ostatných štandardov ako je UML, MOF nebo XMI vznikla v roku 2001 formálna, ale stále nekompletná špecifikácia. Po tomto kroku bolo MDA odhlasované novým štandardom. Poslednou platnou špecifikáciou je dokument MDA Guide Version 1.0.1 z roku 2003.

Od začiatku bolo MDA veľkým sľubom pre budúcnosť a stalo sa témou niekoľkých štúdií a kníh. Napriek tomu nebolo medzi vývojármi veľmi rozšírené. Väčšina nebola spokojná s výsledkami, ktorých sa pomocou MDA dosahovalo. Bola to predovšetkým nekompatibilita modelovacích nástrojov, chýbali praktické skúsenosti a referenčné implementácie.

V súčasnej dobe sa MDA začína viac presadzovať aj v komerčnej sfére. Kvôli stále väčšiemu dopytu začali poprední výrobcovia modelovacích a vývojových nástrojov implementovať funkcionalitu MDA. Zavedenie vývoja pomocou MDA však zatiaľ nie je jednoduché, pretože stále chýbajú referenčné implementácie. Preto sa vývoj pomocou MDA značne odlišuje vo väčšine nástrojov a kompatibilita je stále veľmi slabá.

Ako už bolo povedané, MDA je štandard, ktorý definuje metódy pre nový spôsob návrhu systémov pomocou modelov. Najdôležitejším prvkom tohto štandardu je model, ktorý už neslúži výlučne ako prostriedok pre analýzu a návrh, ale využíva sa takmer vo všetkých fázach vývoja. Takéto modely sú potom priamou súčasťou vývoja a nie iba nutným rozšírením výsledného kódu. Už neslúži len pre zachytenie požiadavkov pri analýze, sprehľadnení riešenia daného problému, lepšiu orientáciu v systéme a poskytnutie jednotnej komunikácie medzi vývojármi. Modelovanie na báze MDA je tak v podstate programovanie, ako ho poznáme z dnešnej doby, ale na inej úrovni.

Ďalšími veľmi dôležitými prvkami sú platforma, platformovo nezávislý model (Platform Independent Model - PIM) a platformovo závislý model (Platform Specific Model - PSM). Platforma špecifikuje konkrétne prostredie alebo technológiu, PIM je model na vysokej úrovni abstrakcie neobsahujúci žiadnu väzbu na platformu a PSM je model rozšírený o informácie vzťahujúce sa k určitej platforme.



obr. 1: OMG

1.1 Ciele MDA

Použitie MDA pri vývoji systémov by malo priniesť predovšetkým nasledujúce štyri zlepšenia vývoja softwaru.

1.1.1 Portabilita

Prenositeľnosť produktu je zaistená tím, že vychádzame z modelov na vysokej úrovni abstrakcie. Tieto modely sú podľa definície MDA nezávislé na cieľovej platforme. To znamená že ten istý PIM model môžeme transformovať na viacero PSM modelov pre rôzne platformy.
Jedinou otázkou miery prenositeľnosti potom zostávajú predpisy transformácií pre rôzne platformy. Pre obľúbené a veľmi rozšírené platformy ako je Java EE či .NET existuje dostatočné množstvo transformačných predpisov. Naopak pri použití menej rozšírených platforiem je väčšinou potreba tieto predpisy najskôr vytvoriť.
Postup pri prenose existujúceho systému vyvíjaného pomocou MDA na novu technológiu či platformu je potom jednoduchý – nie je potreba nič viac len vytvoriť potrebné transformácie.

1.1.2 Znovupoužiteľnosť

Opakované využitie existujúcich komponentov je jednou zo základných otázok súčasného vývoja softwaru. So znovupoužiteľnosťou sa v informatike stretávame na každom kroku. Už sa nejedná len o tvorbu a využívanie rôznych knihovien, ale sústredíme sa tiež na komponenty, frameworky a rozhranie.
V prípade MDA môžeme hovoriť o dvoch znovupoužiteľných častiach. Jednou z nich je model. Znovupoužiteľnosť modelov je závislá na ich úrovni abstrakcie a na detailnosti ich prevedenia. Ak teda vytvárame podobné aplikácie, môžeme využiť už existujúce PIM modely. Druhou znovupoužiteľnou časťou sú transformácie. MDA technológia priamo počíta s tím, že transformačné pravidlá pre zvolenú platformu budú medzi jednotlivými modelmi rovnaké. Ak vytvoríme teda sadu transformačných pravidiel pre nejakú platformu, môžeme z tejto práce ťažiť aj v budúcnosti.

1.1.3 Interoperabilita

V dnešnej dobe stále menej systémov beží ako celok izolovaný od ostatných existujúcich systémov. Väčšinou medzi nimi nájdeme spoločné väzby. Typickým príkladom je situácia, kedy vyvíjaný systém používa komponentu iného (väčšinou existujúceho) systému. Pri vývoji sa teda musí brať ohľad nie len na funkcionalitu týchto komponent, ale aj na technológiu, pomocou ktorej je implementovaná, umiestnenie stroja, brehového prostredia a ďalšie.
MDA definuje pre prípady, kedy niektoré súčasti systému spolupracujú s externými komponentmi na inej platforme mechanizmus, ktorý ich umožňuje spojovať bez ohľadu na ich interpretáciu. Tento mechanizmus, nazývaný “bridge”, si môžeme predstaviť ako premostenie medzi dvoma modelmi, z ktorých každý je určený pre inú platformu.

Premostenie je v skutočnosti súbor mapovacích pravidiel medzi platformami. Značnou výhodou je, že sú tieto závislosti definované vnútri modelu. Tím je zaistená aspoň čiastočná interoperabilita medzi rôznymi platformami a vrstvami.

1.1.4 Produktivita

Zvýšenie produktivity je najdôležitejším faktorom požadovaným spoločnosťami pri adaptovaní technológie MDA pri vývoji aplikácií. Prvým spôsobom, ako zvýšiť produktivitu pri vývoji, je automatizácia niektorých procesov. To je v MDA zaistené automatizáciou transformácie modelov. Vďaka nej je možné rapídne znížiť množstvo vynaloženej práce pri kódovaní alebo testovaní.

Ďalší aspekt, na ktorý sa MDA zameriava, je lepšie využitie modelov. Pri vývoji pomocou tradičných technológii sa postupne stávajú modely nedôležité. Veľakrát sa totiž zmeny prevedené pri realizácii produktu prejavia iba v kóde (prípadne v dokumentácii), ale nie v samotných modeloch.

Pri využití MDA je model pevne viazaný na danú aplikáciu a musí byť udržiavaný (tak ako kód pri tradičnom prístupe). Úsilie vynaložené pri modelovaní aplikácie sa teda priamo premieta do pokroku vo vývoji a nie je len nejakou činnosťou, ktorá prináša užitočné artefakty.

1.2 Princíp vývoja pomocou MDA

Zjednodušene môžeme povedať, že základom návrhu systémov pomocou MDA je model systému na určitej úrovni abstrakcie spolu so súborom transformačných pravidiel. Tento model je potom transformovaný na modely s nižšou úrovňou abstrakcie, až postupne k samotnému kódu aplikácie. Podmienkou pre úspešné využívanie tejto metódy je to, že tak ako modely, aj transformačné pravidlá musia byť vytvorené pomocou dobre definovaného formálneho jazyka. V opačnom prípade by nešlo využiť automatizácie pri transformáciách či konečnom generovaní kódu.

KAPITOLA 2
ARCHITEKTÚRA MDA

2.1 Platforma

Platforma je súbor systémov obsahujúcich technológie, ktoré poskytujú funkcionalitu prostredníctvom rozhrania.
V rámci MDA platforma predstavuje behove prostredie, na ktorom bude náš systém bežať. Teda databáza, aplikačný server alebo technológie ktoré použijeme Java, .NET, PHP.

2.2 Model

Model je vzor, návrh alebo reprezentácia, zobrazujúca objekt alebo činnosti objektu. Podstatou modelového zobrazenia je zachovať dôležité vlastnosti modelovaných objektov a abstrahovať menej dôležité vlastnosti a tým skrývať ich zložitosť a komplexnosť. Vďaka takémuto zjednodušeniu nám modely umožňujú jednoduchšie manipulovať s objektmi ktoré zobrazujú.
Exituje veľa typov modelových zobrazení. Odlišujú sa hlavne v zobrazovanom objekte. Či sa teda jedna o skutočný materiál, zvuk, obraz alebo slovný popis, formálneho alebo neformálneho charakteru.
V IT sa stretávame najmä s modelmi objektov, systémov a procesov. Snažíme sa teda graficky, pripadne pomocou doprovodnych textov, reprezentovať všetky objekty vývoja aplikácie.
V rámci MDA, keďže samotne Modelovo Orientovane Programovanie je z veľkej časti o modeloch ako aj z názvu vyplýva, používame niekoľko druhov modelov. Keďže sú však modely v MDA veľmi dôležitou súčasťou, architektúra si vyžaduje silný, dobre použiteľný modelovací jazyk. V tomto prípade hovoríme o Unified Modeling Language.

2.2.1 Computing Independent Model CIM

Je výpočetne nezávislý model, ktorý neberie ohľad nato ako budú jednotlivé časti implementovane a či vôbec budú implementovane. Zobrazuje systém ako celok, zachytáva prostredie v ktorom sa bude systém používať, aké služby bude používať, aké výstupy bude generovať, s akými entitami bude spolupracovať. Pri tvorbe takéhoto modelu sa používajú termíny z odpovedajúceho doménového slovníka, čím sa odstránia problémy komunikácie analytikov IT a expertov z oblasti danej domény nie z oblasti IT. Tento model by mal byt ako jediný z využívaných modelov pochopiteľný a zrozumintelny aj pre užívateľa. Ak splňuje model požiadavku potrebnej detailnosti, môže byt použitý v neskorších fázach vývoja aj ako podklad pre kontrolu. Bohužiaľ nesprávne je dôležitosť CIM často podceňovaná najmä z dôvodu jeho abstraktnosti, a býva nahradený podrobnejším tzv. PIM modelom.

2.2.2 Platform Independent Model PIM

Je platformne nezávislý model, teda neobsahuje žiadne informácie spojene s cieľovou platformou systému. Popisuje podrobnejšie časti subsystému CIM ktoré budú realizovane pomocou výpočtovej techniky. Tento model vyjadruje jednotlivé entity, ich atribúty, relácie, životné cykly objektov a procesov. Vlastnosť nezávislosti na platforme je veľmi dôležitá pre znovupouzitelnost. Vďaka nej môžeme model použiť pri každej s iterácii alebo pri vývoji podobných aplikácii.

2.2.3 Platform Specific Model PSM

Je platformne závislý model, teda model ktorý zobrazuje aplikáciu alebo jej vývoj na danej platforme. Tato fáza vývoja je prakticky najnáročnejšia a najkomplexnejšia. Model vychádza z PIM modelu. Pri transformácii sa používajú návrhové vzory. Obsahuje rysy danej cieľovej platformy a potrebne črty pre implementáciu. Pri komplexných projektoch sa často využíva viac PSM modelov, kde každý popisuje určitú časť aplikácie, alebo je pomocou viacerých modelov popisovaná jedna čas v rôznych vrstvách abstrakcie. Alebo v prípade že jednou z požiadavok aplikácie je jej multi-platformnost tiež je potrebne vytvoriť model pre každú platformu.

2.2.4 Code

Zdrojový kód predstavuje samotnú naimplementovanu spustiteľnú aplikáciu. Z hľadiska MDA je kód chápaný ako ďalší model, ktorý realizuje danu aplikáciu na cieľovej platforme.

2.3 Transformácie

Ako vyplýva zo základného konceptu MDA kľúčovým procesom je transformácia abstraktných modelov do platformne špecifických modelov, ktoré môžu byt použite na generovanie cieľových zdrojových kódov aplikácie. Podľa obrázka transformácia znamená že zdrojový model sa transformuje na cieľový model na základe určitých transformačných pravidiel.

obr. 2: Transformácia

Transformačné definície sú jedným z nástrojov MDA, ktoré najviac zrýchľujú generovanie výsledného kódu a tým celkový vývoj aplikácie. Zabezpečujú celkovú automatizáciu a sú bez väčších uprav znovupouzitelne. Dôležitosť transformácii by nemala byt nikdy podceňovaná, pretože majú zásadný vplyv na kvalitu softwaru.

KAPITOLA 3
PROCES VÝVOJA

Teraz môžeme podrobnejšie rozobrať jednotlivé časti vývoja systémov v MDA tak ako nasledujú chronologicky počas vývoja.

3.1 Fázy vývoja

Fáza 1

Je potrebne vytvoriť CIM, ktorý ponesie globálne informácie o systéme, jeho funkcionalite a prostredí, v ktorom bude systém pracovať. Súčasťou je aj špecifikácia doménového slovníka.

Fáza 2

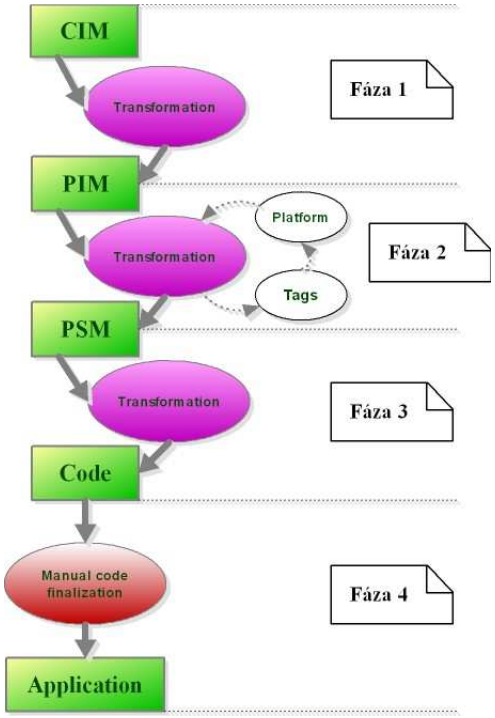
Pomocou transformačných pravidiel sa model CIM transformuje na model PIM.

Fáza 3

Podľa poziadavkov aplikácie vyberieme konkrétnu platformu na ktorej bude systém implementovaný. Pomocou transformačných pravidiel transformujeme model PIM na model PSM. V určitých prípadoch môže byt PSM model vstupom pre ďalšie transformácie detailnejšieho charakteru. Takýmto spôsobom môžeme prevádzať cyklus opakovane až pokiaľ sa nedopracujeme ku konečnému modelu PSM vhodnému pre generovanie kódu aplikácie.

Fáza 4

Na základe konečného PSM modelu generujeme výsledný kód aplikácie. Je potrebne si však uvedomiť že vygenerovaný kód nie je hotová aplikácia. Vygenerovaný výstup predstavuje základňu štruktúru systému. Posledným krokom je doprogramovanie jednotlivých časti aplikácii, ako napríklad aplikačná logika atd. Tento krok nie je zaradený do špecifikácie MDA.



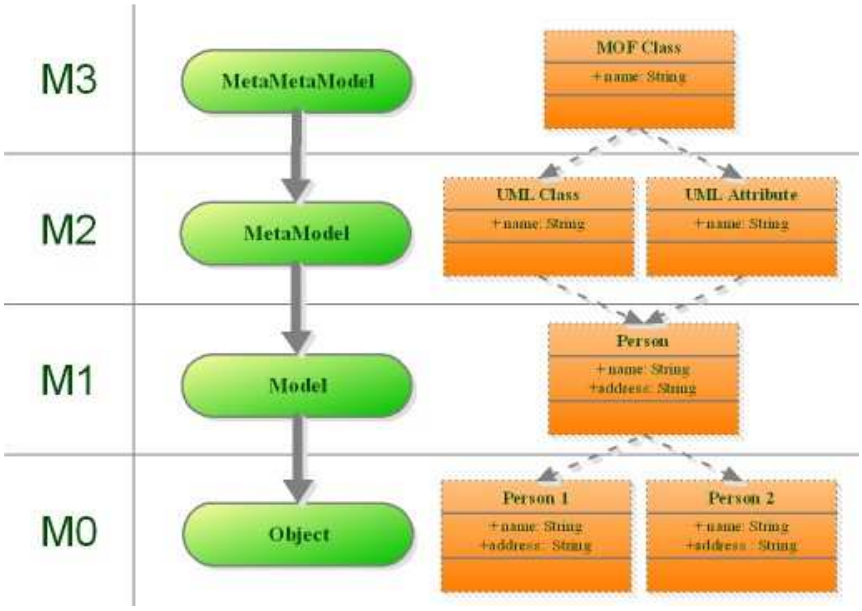
obr. 3: Proces vývoja

3.2 Modely

Pri vytváraní aplikácii pomocou MDA je potrebne vytvárať niekoľko typov modelov. Pri tomto procese sa tiež používajú rôzne techniky na značkovanie, vytváranie schém a podobne. Rozoberme si teraz tieto technológie podrobnejšie.

3.2.1 MOF

Pri definovaní jednotlivých krokov modelovo orientovanej architektúry bola spoločnosť OMG postavená pred prekážku. Bolo potrebne vytvoriť silný jazyk, ktorým by sa jednotlivé modely dali definovať. Výsledkom tejto iniciatívy je MOF. Je to štandard, ktorý definuje samotný jazyk UML. Je navrhnutý ako 4-vrstva architektúra. Na vrchnej vrstve, zvanej M3 je definovaný meta-metamodel. Tento meta-metamodel je jazykom pre tvorbu metamodelov, zvaných M2 modely. Do tejto vrstvy spadá okrem iných modelov aj nám dobre známy UML model, ktorý definuje samotný jazyk UML. Na ďalšej vrstve sú M1 modely, čo sú vlastne modely definovane jazykom UML. Posledná vrstva je M0 vrstva alebo tzv. Dátová vrstva. Ta definuje objekty skutočného sveta.



obr. 4: MOF úrovne

V máji 2006 OMG definovala dve varianty MOF:

Essential MOF

Complete MOF

A neskôr v júny 2006 bol predložený návrh na definíciu tretej varianty.

Semantic MOF

Samotná špecifikácia MOF definuje tieto prvky:

Classes – Triedy sú objekty, ktoré majú určitú identitu, stav a rozhranie, Ich stav je reprezentovaný atribútmi a konštantami, rozhranie je definovane metódami.

Associations – Popisujú vzťahy medzi triedami.

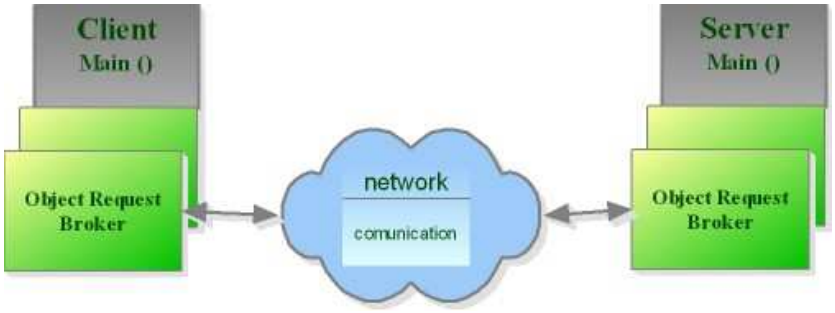
Data types – Neinstaciovatelne typy, ako primitívne typy, kolekcie a enumerácie.

Packages – Balíky pre rozdeľovanie metamodelov do logických podjednotiek

Constraints – Pravidla, ktoré obmedzujú množinu metamodelov iba na validne hodnoty a prvky

3.2.2 CORBA

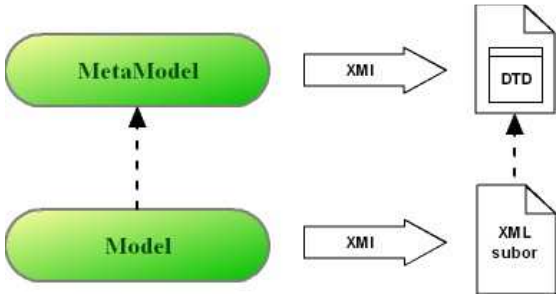
Je štandard, ktorý umožňuje softwarovým komponentom napísaným v rôznych jazykoch a bežiacich na rôznych platformách pracovať spoločné. Predstavuje mechanizmus, ktorý normalizuje sémantiku volania metód medzi objektmi, ktoré sú umiestnene, implementovane, v rôznych aplikačných doménach. CORBA využíva IDL na definíciu rozhrania ORB, prostrednicvom ktorého objekt komunikuje s ostatnými objektmi z iných domén. V súčasnosti existujú mapovania IDL na hlavne jazyky ako C++ , Java, Ruby, Smalltalk, atd.



obr. 5: CORBA

3.2.3 XMI

Je štandard na výmenu metadat medzi M3 až M1 vrstvami využívajúci technológiu XML. Predstavuje spojenie medzi metamodelom a schémou XML (XML Schema, DTD). V špecifikácii sú definovane jednotlivé elementy XML tak ako predstavujú jednotlivé prvky modelov. V súčasnosti sa špecifikácia XMI každou verziou obohacuje o nove štruktúry a elementy aby bolo čo najpodrobnejšie možné zachytiť štruktúru modelu. Avšak to zo sebou prinášalo nevýhodu v tom, že medzi nástrojmi na modelovanie panuje značná nesúrodosť. Momentálne je možné modely UML 2.0 prenášať iba jednou verziou XMI a to 2.1Schema.



obr. 6: JMI

3.2.4 JMI

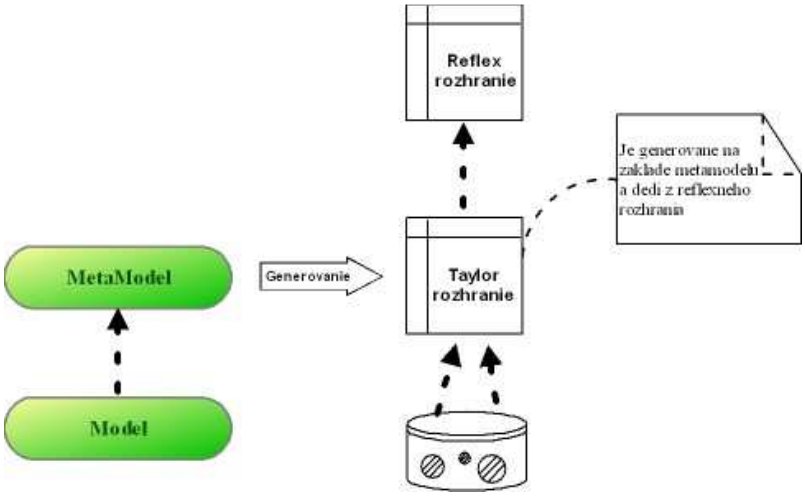
Väzba výmeny metadat medzi modelmi implementovaná pre Java framework ako API. Bola vytvorená združením JCP.

Je to platformne neutrálna špecifikácia, ktorá definuje vytvorenie, prístup a výmenu metadat v Jave. JMI definuje dva druhy rozhraní.

Taylorovske rozhranie – je adaptovane ne jediný metamodel, teda metamodely v UML 2.0 môžu komunikovať jedine cez prislusne rozhranie vytvorene pre UML 2.0

Refelxivne rozhranie – na rozdiel od Taylorovskeho sa snaží komunikovať s akýmkoľvek modelom bez ohľadu na jeho metamodel. Je možné napríklad na základe elementu pristupovať k jeho metadatam, ako funkciám, atribútom , ktoré sú definovane v jeho metamodely

V skutočnosti sú oba typy rozhrania spolu zviazane. Taylorovske rozhranie dedí svoje vlastnosti z reflexívneho.



obr. 7: JMI

3.2.5 EMF

Je to framework pre vývojové prostredie Eclipse, ktorý obsahuje nástroje na vytváranie modelov a generovanie kódu na základe štruktúrovaného dátového modelu. Zo špecifikácii modelu v XMI, EMF poskytuje nástroje pre vytvorenie sady Java tried spolu s triedami adaptéru, ktorý umožňuje zobrazovanie a príkazovo orientovane upravovanie modelu.

Základom frameworku však nie je meta-metamodel MOF ale vlastný meta-metamodel definovaný EMF s názvom Ecore. Je však veľmi podobný základnému meta-metamodelu Essential MOF. EMF je veľmi podobne a pracuje na rovnakom princípe ako JMI avšak stretáva

sa so širším využitím a to hlavne preto, že JMI ako také, bolo implementovane iba pre verziu MOF 1.4 z čoho vyplýva že je pre vývoj modelov na základe UML 2.0 nepoužiteľné. EMF podporuje najnovšiu verziu MOF 2.0 preto sa častejšie stretávame s meta-metamodelmi definovanimy v Ecore.

3.2.6 UML

Samozrejme že modelovo orientovaný vývoj aplikácii, ktorého gro je v tvorbe modelov nie je obmedzene na jeden modelovací jazyk. Avšak ten najrozšírenejší je UML. Možno aj preto, že splňuje dve najzákladnejšie podmienky pre vývoj modelov:

Čitateľnosť – Jednoduchosť, pochopiteľnosť je vždy prvým krokom k čí už upravovaniu alebo vytváraniu akéhokoľvek systému. To zahrnuje znalosti o tom čo systém obsahuje, ako sa systém sprava v rôznych situáciách.

Znovupouzitelnost – Môžeme rozumieť aj ako vedľajší produkt čitateľnosti, pretože po tom ako bol systém namodelovaný , mame tendenciu a oveľa ľahšie sa nám odstraňujú redundancie a podobnosti v systéme a tým sa model stáva prehľadným a tým aj jednoducho použiteľným opakovane

Profil

UML diagramy sú dostatočné generické nato, aby umožňovali modelovať rôzne domény. Pomocou diagramov môžeme vytvárať modely databáze, alebo dokonca samotne metamodely. K rozlíšeniu o akú doménu sa v danom modely jedna nám slúžia profily. Tvoria ucelený mechanizmus, ktorý však nezasahuje do metamodelov UML.

Stereotyp

Základnou jednotkou profilu je stereotyp. Môžeme si ho predstaviť ako značku, ktorú vlastni každý element modelu. Vďaka stereotypom definujeme čo predstavuje daný element, či sa jedna o dátový objekt, entitu, rozhranie atď...

Jazyk UML poskytuje nástroje pre vytváranie rôznych druhov diagramov, ktoré rozdeľujeme do dvoch kategórii:

Statické

-Use case diagram

-Class diagram

Dynamické

- Object diagram

- State diagram

- Activity diagram

- Sequence diagram

- Collaboration diagram

Implementačné

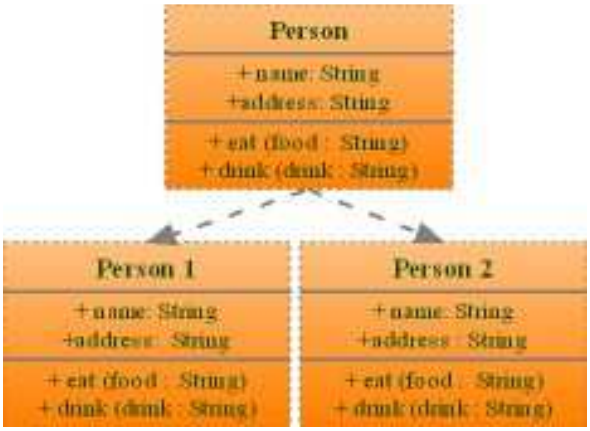
- Component diagram

- Deployment diagram

Use case diagram - Používa sa na identifikáciu primárnych elementov a procesov, ktoré formujú systém. Základne elementy sa nazývajú aktéri a procesy sa nazývajú prípady použitia.

obr. 8: Use case diagram

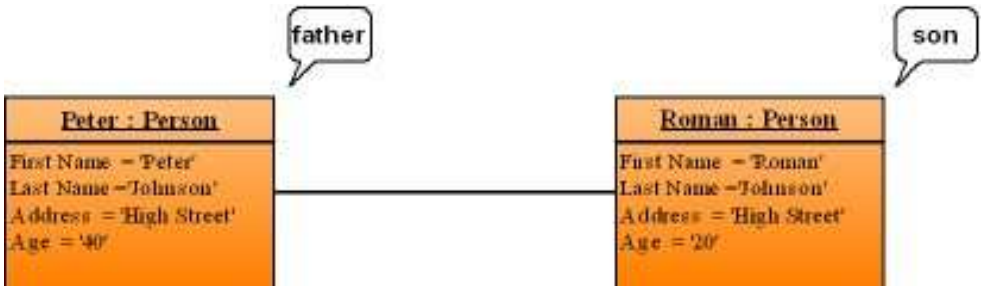
Class diagram – Používa sa na detailnejšiu definíciu návrhu systému. Diagram zobrazuje aktérov use case diagramu ako sieť vzájomné prepojených tried. Jednotlivé triedy sú medzi sebou vo vzťahoch, ktoré môžu byt typu „je“ alebo „ma“. Každá trieda ma určitú funkcionalitu, ktorá sa zobrazuje pomocou metód a určíte vlastnosti, ktoré sa zobrazujú pomocou atribútov.



obr. 9: Class diagram

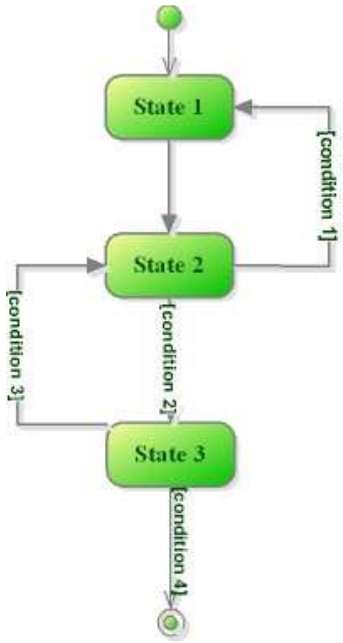
Object diagram – Je špeciálny druh triedneho diagramu. Vychádza s informácie „Objekt je inštancia triedy“. To v podstate znamená, že objekt zobrazuje stav triedy v určitom časovom

bode, počas behu systému. Objektový diagram zachytáva vzťahy asociácie a stav rôznych tried v systéme v čaše.



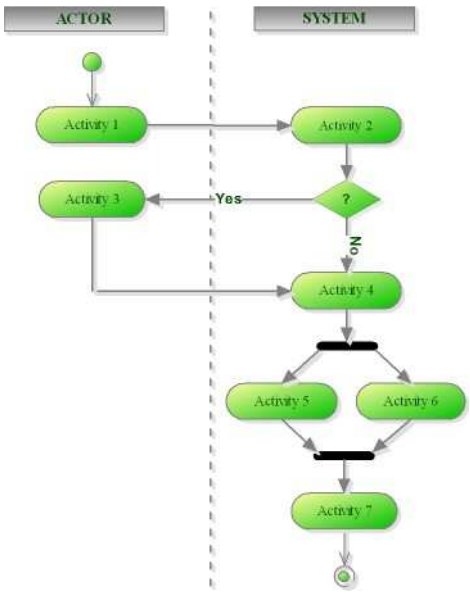
obr. 10: Object diagram

State diagram – Stavový zobrazuje rôzne stavy, v ktorých sa systém počas svojho životného cyklu. Objekty menia svoj stav ako reakcie na udalosti. Diagram zachytáva prechod od počiatočného stavu ku konečnému stavu.



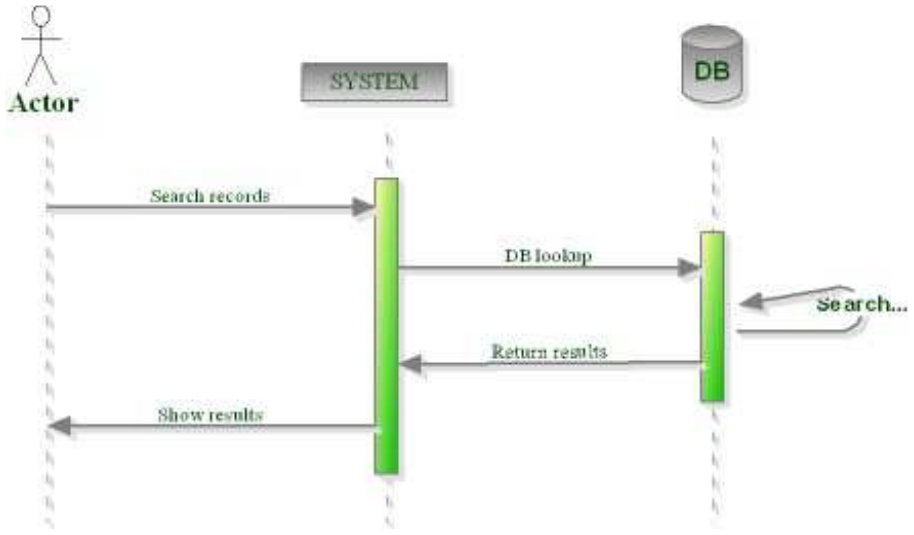
obr. 11: State diagram

Activity diagram – Prúdy procesov sú zachytene v diagrame aktivít. Podobne ako stavový diagram, activity diagram pozostáva z aktivít, akcii, prechodov, počiatočných, konečných stavov a podmienok.



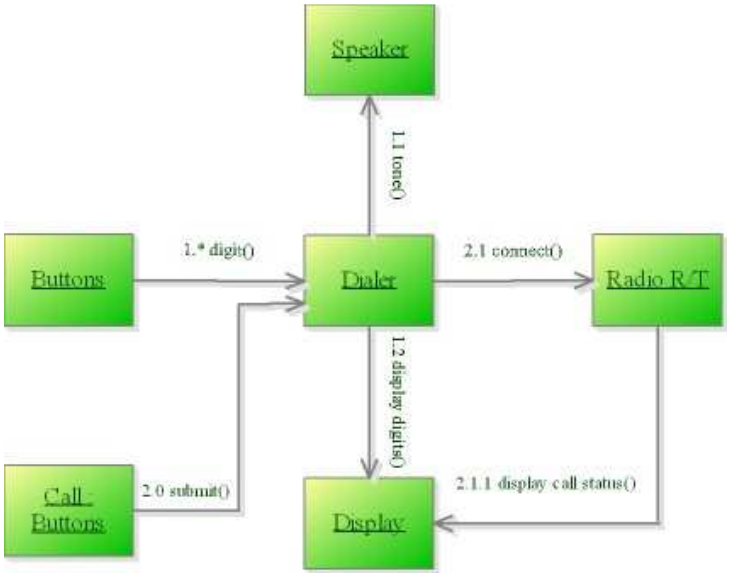
obr. 12: Activity diagram

Sequence diagram - Sekvenčný diagram reprezentuje interakcie medzi objektmi v systéme. Užitočný aspekt sekvenčného diagramu je že je chronologicky. To znamená že presná sekvencia príkazov je reprezentovaná krok za krokom, tak ako počas behu systému nasledujú. Rozdielne objekty medzi sebou komunikujú predávaním si sprav.



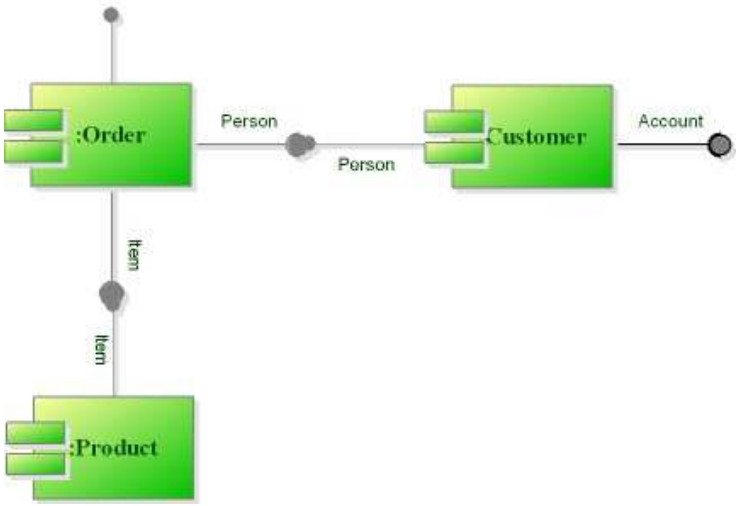
obr. 13: Sequence diagram

Collaboration diagram – Diagram zoskupuje spolu interakcie medzi objektmi, ktoré sú zoradene v číselnom zozname, čo pomáha v prípade potreby jednoducho vyhľadať potrebnú sekvenciu. Diagram zaznamenáva všetky interakcie, všetkých objektov v systéme.



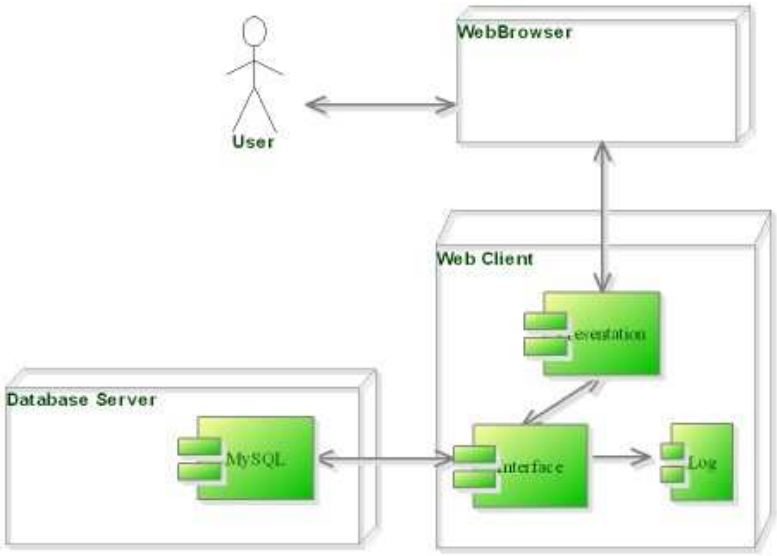
obr. 14: Collaboration diagram

Component diagram – Diagram reprezentuje časti vrchnej vrstvy systému, ktoré vytvárajú vzhľad systému. Zobrazuje komponenty, ktoré boli počas vývoja odstránené, a ktoré boli do vývoja zhasnute.



obr. 15: Component diagram

Deployment diagram – Zachytáva konfiguráciu behovych elementov aplikácie. Tento diagram sa využíva vo fáze, kedy je už daný systém implementovaný a riadi sa nim samotne nasadzovanie aplikácie.



obr. 16: Deployment diagram

3.3 Transformácie

Ako vyplýva zo základného konceptu MDA, kľúčovým procesom vývoja sú transformácie abstraktných modelov na špecifické modely, ktoré môžu byt zdrojom generovania kódu. Transformovanie znamená premena zdrojového modelu na cieľový model založená na určitých transformačných pravidlách. Na definovanie transformačných pravidiel môžeme využiť viacero metód.

3.3.1 Vlastnosti transformácii

Pozrime sa na rozličné prístupy k tvorbe transformácii a aké sú ich základne odlišnosti.

3.3.2 Transformačné pravidla

Transformačné pravidla opisujú ako majú byt elementy zdrojových modelov prelozene do elementov cieľového modelu. Transformačné pravidlo pozostáva z dvoch časti:

Časť ľavej ruky (LHS)

Časť pravej ruky (RHS)

LHS pristupuje k zdrojovému modelu , kým RHS sa spája s cieľovým modelom. Obe časti môžu byt definovane premennými, vzormi, logikou, atd. Jednotlivé postupy sa odlišujú v tom ako definujú pravidla

3.3.3 Rozsah aplikovania pravidiel

Dovoľuje definovať transformácii cieľový rozsah. Podľa potreby je možné urobiť reštrikcie na častiach modelov, ktoré sú súčasťou transformácie. Je možné definovať rozsah transformácie zdrojového, rovnako ako aj cieľového modelu. Definovanie rozsahu je dôležité pre rýchlosť a výkon transformačného procesu alebo pre prípad zložitých transformačných štruktúr.

3.3.4 Vzťah medzi zdrojovým a cieľovým modelom

Samotná definícia transformácie implikuje proces, v ktorom je na základe zdrojového modelu vytvorený cieľový model. Transformácia však môže definovať aj proces, výsledkom ktorého nie je novy model, ale úprava existujúcich cieľových modelov alebo daného zdrojového modelu. Takýto proces môže byt deštruktívneho charakteru, teda ak z modelu odstraňuje elementy, alebo môže model rozširovať o nove elementy.

3.3.5 Stratégia aplikovania pravidiel

V určitých prípadoch vyžadujeme aby sa dane pravidlo uplatnilo iba na špecifický určenom mieste. Keďže takýchto miest môže byt v modely viac je potrebne definovať stratégiu rozhodovania. Tato stratégia môže byt deterministická alebo nedeterministická.

3.3.6 Plánovanie aplikovania pravidiel

V komplexných modelových transformáciách môže byt počet transformačných pravidiel veľmi veľký. Mechanizmus plánovania sa preto využíva na určenie poradia, v ktorom budú pravidla aplikovane. Pri niektorých postupoch užívateľ nemá kontrolu nad mechanizmom plánovania. Postupy sa tiež môžu odlišovať v spôsoboch akým sú pravidla vyberane, ako sa prevádza iterácia a ako sa pravidla rozdeľujú do rôznych transformačných fáz.

3.3.7 Organizácia pravidiel

Pravidla môžu byt štruktúrované rôznymi spôsobmi. Modelové transformácie sa väčšinou odlišujú v troch oblastiach. Modularita, teda rozčlenenie pravidiel do balíčkov a modulov. Zabezpečenie znovupouzitelnosti, teda previazanie viacerých pravidiel. A organizačná štruktúra pravidiel.

3.3.8 Stopovatelne linky medzi modelmi

Transformácie môžu zachovávať linky medzi ich zdrojovými a cieľovými elementmi. Zabezpečuje sa tak možnosť prevádzania spätných analýz, synchronizácie a debugging.

3.3.9 Smerovateľnosť

Poslednou vlastnosťou v ktorej sa transformácie môžu odlišovať je smerovatelnost. Transformácia môže byt jednosmerná, obojsmerná. Nesmerová transformácia môže byt prevedená iba v jedinom smere**.** Takže zdrojový model môže byt transformovaný na cieľový model ale nie naopak. Obojsmerná transformácia môže byt spustená oboma smermi.

3.3.10 Postupy tvorby transformácii

OMG ponuka niekoľko postupov ako transformovať modely vo svojej špecifikácii (OMG, 2003). Poďme si ich podrobnejšie rozobrať a klasifikovať podľa ich vlastnosti rozoberaných v predchádzajúcich riadkoch. Aj keď klasifikácia nie je úplné kompletná, pretože OMG prezentuje transformačné postupy ako všeobecný návrh, dávajú dostatočné dobrý ucelený prehľad v rozdielnych technikách používaných pri tvorbe transformácii.

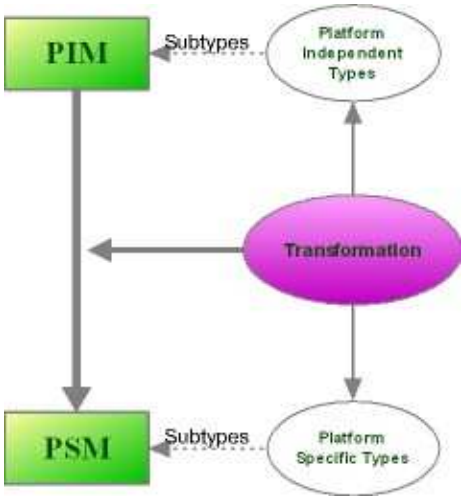
3.3.11 Značkovanie

Pri procese transformácie PIM na PSM je prvým krokom vyber cieľovej platformy. Mapovanie na danú platformu môže byt už zostrojene, ak nie je potrebne ho tak isto vytvoriť. Takéto mapovanie definuje súbor značiek, použitých na označenie elementov PIM. Tieto značkované elementy potom zavadzajú mapovanie elementov PIM na elementy PSM.

obr. 17: Proces značkovania

Tento postup transformácie používa transformačné pravidla s LHS, ktorý definuje zdrojový element so špecifickou značkou, a RHS, ktorý definuje daný zdrojový element. Tiež muší definovať pravidla vytvárania elementov. Toto môže byt prevedené ručné alebo automaticky. V neskoršom priebehu môžu byt zvolene akékoľvek ďalšie postupy, OMG nezavádza nutnosť použitia určitých špecifických postupov. Aplikovanie pravidiel je organizovane v dvoch fázach.

Značkovanie a mapovanie. Počas značkovacej fázy je rozsahom značkovania cely model. V druhej fáze je rozsah určený podľa značiek. Vo väčšine prípadov výsledkom prvej fázy je iba upravenie už stavajúceho zdrojového modelu, pri druhej fáze dochádza už k vytvoreniu nového cieľového modelu.



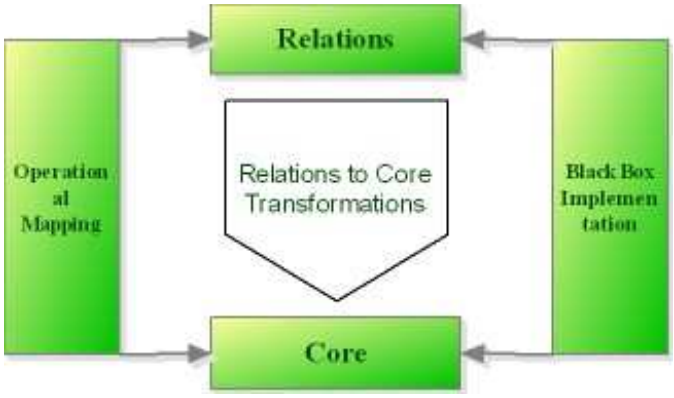
obr. 18: RHS a LHS

Na obrázku vidíme schému transformačného procesu modelu, založeného na mapovaní platformne nezávislých typov na platformne špecifické typy. Elementy zdrojového modelu sú subtypmi typov PIM. Elementy cieľového modelu sú subtypmi typov PSM.
Transformačné pravidla definovane v transformácii priamo zmapujú PIM typy na PSM typy. LHS transformácie vyberie platformne nezávislé typy a RHS vyberá platformne špecifické. Určovanie rozsahu aplikácie pravidiel nie je momentálne potrebne, mapovanie sa odohráva na úrovni celkového modelu. Vo väčšine prípadov bude výsledkom vytvorenie nového cieľového modelu. V prípade že transformácie je jednoduchá a nevyžadujeme komplexne mapovanie modelov s výsledkom vytvorenia viacerých cieľových modelov pre viacero platforiem, je stratégia deterministická. Ak je teda mapovanie iba takpovediac jedna-k-jednej nie je problém vytvoriť obojsmernú transformáciu.
Male odlišnosti sa prejavujú aj medzi postupom mapovania subtypov a postupom mapovania metamodelov. Základná štruktúra oboch procesov je rovnaká.

3.3.12 QVT

Je štandard definujúci metamodel transformácie modelov. Splňuje niekoľko základných myšlienok. Jednou s nich je že zdrojový a cieľový model sú konformne metamodelu MOF. To prakticky znamená že syntax QVT je odvodená zo syntaxe MOF. QVT v sebe integruje základy štandardu OCL. Sú definovane tri tzv. domenovo špecifické jazyky, *Relations, Core, Operational mapping,* ktoré sú vo vrstvovej architektúre. Relations a Core sú deklaratívne

jazyky na rozdielnych vrstvách abstrakcie. *Operational mapping* je imperatívny jazyk, ktorý rozširuje a normuje obe vyššie uvedené vrstvy. Syntax *Operational mapping* nie je ničím odlišná od klasického imperatívneho jazyka, teda obsahuje cykly, podmienky atď. Súčasťou architektúry QVT je aj mechanizmus *BlackBox*, ktorý funguje ako interface pre non-QVT knižnice a transformácie implementovane v iných jazykoch (XSLT, XQuery). Momentálne QVT štandard umožňuje definovať transformácie model-model. V budúcnosti sa plánuje štandardizovať aj tvorba mode-text a text-model transformácii.



obr. 19: QVT

3.3.13 Šablóny

Základom je vytvorenie šablóny cieľového modelu, v ktorej sú definovane parametre. Takúto šablónu nazývame parametricky model. Transformácia sa potom prevádza tak, že zoberieme parametricky model a nahradíme parametre informáciami zo zdrojového modelu. K tomu aby sme mohli k transformáciám využiť parametrické modely, potrebujeme vhodný jazyk na ich vyjadrenie. Existuje viac takých jazykov, zatiaľ ale neposkytujú takú funkcionalitu a efektivitu ako objektovo orientovane jazyky.Napriek tomu je tento spôsob postupu pri transformáciách značné obľúbený, možno aj preto že sa čiastočné podoba vývoju webových aplikácii v jazykoch ako PHP alebo JSP.Tento postup je výhodný hlavne v prípadoch, keď logika spracovania jednotlivých elementov nie je príliš zložitá.Nevýhodou takéhoto postupu je absencia všeobecné uznávaného a štandardizovaného jazyka. Vo väčšine prípadov sa stretávame s tým, že šablóny sú veľmi úzko zviazane s daným vývojovým prostredím a s jazykom, v ktorom boli implementovane.

3.3.14 Manuálne programovanie

Základom tohto prístupu je naprogramovanie transformácie rovnako ako ktorejkoľvek inej aplikácie. Takáto transformácia je teda program špeciálne vytvorený na manipuláciu s modelmi. Pri vývoji takýchto programov sa využívajú generovane rozhrania. Taylorovske a reflexívne ako

sme už rozoberali vyššie. Prvým krokom je teda vygenerovanie API zo zdrojového a cieľového metamodelu pomocou JMI alebo EMF. Vygenerovane rozhrania poskytujú metódy pre prístup a manipuláciu so všetkými elementmi modelov. Napríklad ak budeme chcieť transformovať triedny diagram na diagram databázový, môžem po vygenerovaní dostať rozhranie *Class*, *Property* a *Datatype* a na druhej strane *Table*, *Column* a *Type.* Definícia transformačného procesu potom odpovedá implementácii vybraných metód.

KAPITOLA 4
MDA NÁSTROJE

Prakticke využitie akejkoľvek metódy návrhu je závislé na tom, aká je jej podpora vo vývojových nástrojoch. Dynamickom a rýchlo sa rozvyjajucom svete informačných technológii, veľakrát nezáleží na kvalite myšlienok prezentovaných v rôznych teóriách a štandardoch, ale skôr na ich okamžitej a jednoduchej aplikovateľnosti. O MDA, ktorý je založený na štandardoch a automatizovaných procesoch to platí dvojnasobne.

Finálna voľba vyhovujúceho nastroja na vývoj nie je vôbec jednoduchá, keďže väčšina nástrojov je momentálne vo fáze vývoja alebo prípravy. Keďže neexistuje ani žiadny certifikačný proces pre takéto vývojové nástroje, ich implementácia je preto veľmi odlišná.

4.1 Požiadavky na vývojový nastroj

Je potrebne si ujasniť že špecifikácia OMG, ktorá sa nejakým spôsobom snaží štandardizovať proces vývoja aplikácii na základe MDA je predovšetkým teoreticky popis. Nekladie veľké obmedzenia v rámci podoby modelov, ani v rámci implementačného jazyka. UML je štandardom pre tvorbu modelov , ale špecifikácia počíta aj s podporou iných jazykov. Tato relatívne voľna špecifikácia ma za následok to, že v súčasnosti existuje veľké množstvo na stojov, ktoré tvrdia že podporujú MDA. Avšak autori týchto nástrojov majú často odlišný pohľad na metodiku MDA, čo sa jednoducho prejavuje v ponuke nástrojov. Súbor týchto funkcii určuje potom metódy akými je systém vyvíjaný.

Nastroj podporujúci vývoj systémov metodikou MDA musí poskytovať tieto funkcie:

- Tvorba modelov na základe metamodelov (MOF)
- Rozširovanie a upravovanie modelov o dodatočné informácie (typy, hodnoty, značky)
- Tvorba model-model transformácii (QVT)
- Manipulácia s transformáciami
- Podpora štandardných dátových formátov (XMI)
- Validácia modelov

4.2 Súčasné najpoužívanejšie vývojové nástroje

ArcStyler
Nastroj neposkytuje podporu samotných PSM modelov. PIM modely sú reprezentovane pomocou diagramov tried, a tie sú základom pre ďalšie transformácie. Kódový model je reprezentovaný ako systém rôznych balíkov zdrojových kódov, tvoriacich jednotlivé komponenty. Transformácie z PIM do kódu sú riadene pomocou zásuvných modulov vytvorených v jazyku JPython. Nastroj obsahuje preddefinovane transformačné moduly pre Javu, J2EE a .NET. Umožňuje pracú s UML profilmi a podporuje štandardy JMI,MOF,XMI a OCL. Nastroj poskytuje aj možnosť použitia v oblasti reverzného inžinierstva na kód tretích strán. Z aplikačných serverov podporuje BEA Weblogic, JBoss, IBM Websphere a databázy Oracle, Cloudspace a Hypersonic.

IQgen
MDA generátor implementovaný v Jave. Dôkaze generovať akýkoľvek typ textových artefaktov založených na XMI štandardizovanom formáte alebo JSP. Ako vstupný formát používa nastroj UML. Vďaka štandardizácii XMI dôkaze nastroj spolupracovať s množstvom ďalších nástrojov. IQgen patri do rodiny tzv. *tier* generatorov, cize nestara sa iba o generovanie čiastkových artefaktov, tried ale výsledkom jeho procesu je jedna, zvolená vrstva systému. Napríklad v prípade že požadovaný výstup je webová aplikácia, nastroj je schopný vygenerovať databázu, logiku a samozrejme aj užívateľské rozhranie.

IBM Telelogic Rhapsody
Tento komerčný nastroj je vyvyjany pod taktovkou spoločnosti Telelogic , ktorá je momentálne jednou z menších firiem patriacich IBM. Ako základne jazyku na tvorbu modelov sa uplatňujú okrem UML 2.1 aj SysML 1.0, MODAF a DoDAF. Nastroj umožňuje užívateľom rozšíriť tuto jazykovú výbavu až na schopnosti domenovo špecifického jazyka (DSL). To znamená že užívatelia si môžu vytvoriť svoje vlastne unikátne diagrami a elementy. Ďalšou peknou vlastnosťou ktorou sa Rhapsody odlišuje od ostatných nástrojov na trhu je plna podpora profilov, čo tiež prispia k plnej prispôsobivosti nastroja danému vývojovému softwarovému procesu. Keďže nastroj je navrhnutý naozaj robustne, jeho samozrejmosťou je aj podpora tímovej synchronizácie a verzovacieho systému. Výstupom nastroja je kompletne softwarové riešenie, to znamená zostrojenie všetkých potrebných artefaktov, ako napriklad aj build a make suborov, potrebných na spustenie systému. Flexibilitu zaručuje aj vlastnosť kompatibility uprav medzi vrstvou kódu a vrstvou modelu, teda inými slovami akákoľvek zmena na úrovni kódu sa automaticky premietne v úrovni modelu a naopak. Výhodou nástroju je aj plna podpora štandardu CORBA.

CodeGenie
Tento vývojový nastroj vytvorila anglicka skupina vývojárov známa ako Domain Solutions. Spomedzi dosial opišaných nástrojov sa CodeGenie môže pochváliť najmä prehľadnou štruktúrou modelovania. Ako modelovací jazyk používa nastroj UML a dátový formát je XMI.

IBM Rational Software Architect
Je vývojové prostredie propagovane firmou IBM. Je postavene na Eclipse open-source software frameworku. Podporuje model-code a code-model transformácie. Pracuje s UML 2.1. Kódové artefakty je možné vytvárať v jazykoch ako Java, C#, C++, EJB, WSDL, XSD. Tiež podporuje vytváranie rozhraní na komponenty tretích strán CORBA. A dôkaze vytvárať aj SQL artefakty. Čo sa tyká reverzného inžinierstva RSA poskytuje možnosť spätnej generácie modelu z kódu v Jave a C++. Vývojové prostredie v sebe integruje aj nastroj na správu verzii ClearCase a nastroj ClearQuest umožňujúci manažovanie konfigurácie.

Rational Rose XDE Developer for Java
Umožňuje automatickú alebo vlastnú synchronizáciu medzi modelom a kódom, V rámci MDA podporuje viacero modelov a dovoľuje navrhovať logičku štruktúru databázy. Zaujímavou možnosťou je automatické zachytávanie chovania sa spustenej aplikácie formou UML diagramu. Pre prístup k modelom, diagramom a príkazom sa používa skriptovací jazyk. Podporuje jazyky C++, VB, Ada95, CORBA.

AndroMDA
Tento nastroj je zrejme najvýkonnejší spomedzi open-source MDA nástrojmi. Základom je mechanizmus transformácie PIM modelu do PSM modelu reprezentovaného v pamäti. Z toho je potom pomocou šablón vygenerovaný kod.Transformacie sú riadene pomocou zásuvných modulov nazývaných cartridge. Navrhnutá architektúra aplikácie rešpektuje všeobecné doporucenia pre aplikácie Java EE. Vďaka univerzálnemu prednastaveniu modulov umožňuje nastroj vygenerovanie celej aplikácie priamo z PIM modelu. A veľmi veľkým pozitívom tohto nastroja je aj rozšírená komunita , ktorá sa podieľa na vývoji.

Eclipse Modeling Project
Projekt Eclipse zastrešuje niektoré z predchádzajúcich menších projektov, zaoberajúcich sa tvorbou a spracovaním modelov na platforme Eclipse. Zahrnuje významné pluginy pre Eclipse ako je už skôr zmieňované EMF, ale aj GMF, GMT a MDDi. Vďaka grafickému frameworku GMF je možné dostatočné prehľadné vytvoriť celu štruktúru modelov. Nastroj umožňuje transformácie typu model-model a model-text a tak isto aj reverzne text-model.

OpenMDX
Tento nastroj ma vlastnú implementáciu štandardu MDA. Je založený na architektúre zásuvných modulov. Výsledná aplikácia je touto architektúrou značné ovplyvnená. Bohužiaľ OpenMDX nepodporuje transformácie medzi PIM a PSM modelmi. Jedna sa predovšetkým o framework umožňujúci spustenie aplikácie na základe modelu postaveného na MOF štandarde.

KAPITOLA 5
ANDROMDA

Pre detailnejší rozbor som si dovolil vybrať open-source vývojový nastroj AndroMDA („andromeda“). Dôvodov je hneď niekoľko. Nastroj splňuje takmer všetky moje požiadavky na vývojový nastroj MDA . Nastroj je voľne šíriteľný. Momentálne však nie je plne implementovaná jeho funkcionalita, pretože projekt je stále vo fáze vývoja. Aj keď AndroMDA je skvelý nastroj na vývoj systémov a dôkaze vývojárom ušetriť naozaj veľké množstvo prace, nie je to univerzálny nastroj a nie vždy je vhodne ho na implementáciu použiť.

Nastroj je skvelou voľbou v prípade že :

- Začíname novy projekt
- Chceme maximalizovať množstvo generovaného kódu a tým ušetriť čas
- Vytvárame aplikáciu, ktorá ukladá svoje dáta v databáze

A naopak nie je vhodne ho použiť v prípadoch:

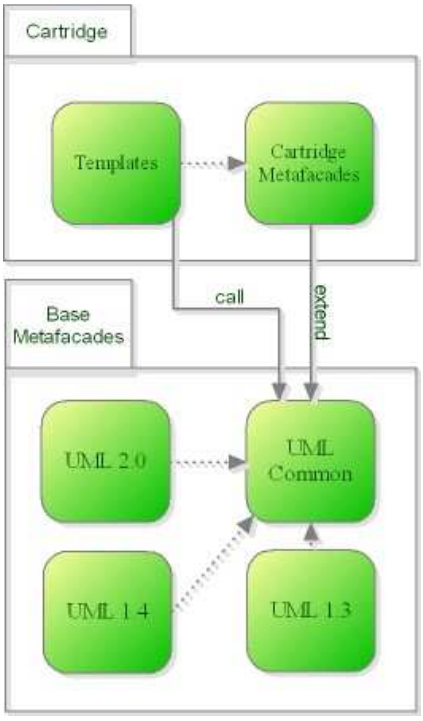
- Naša aplikácia využíva už exitujúcu databázu, ktorá nemôže byt jednoducho zmapovaná objektovým modelom
- Chceme upraviť alebo vytvoriť nove komponenty v aplikácii, ktorých konverzia do modelov by trvala príliš dlho na úkor efektivity

5.1 Metafasady

Poskytujú prístup k modelom reprezentovaným v pamäti. Vytvárajú tak uniformne rozhranie pre rôzne metamodely ako UML 1.4 alebo 2.0. Vlastne metafasady sú tiež Java API a sú využívané pri spracovaní šablón a generovaní kódu. Vďaka metafasadam sa zjednodušujú šablóny, pretože všetka inteligencia je sústredená v Java objektoch. Šablóny používajú pre prístup k objektom getter.

Ideálne by bolo keby každá trieda v metafasadach tvorila rozhranie k odpovedajúcej triede metamodelu. To však nie je pravdou, pretože neexistujú žiadne knižnice obsahujúce jednotlivé triedy metamodelov. Tie sú generovane až počas behu pomocou JMI. Z tohto dôvodu sa miesto rozhraní používajú návrhové vzory typu *Facade.* To umožňuje skryt komplexitu rozhrania JMI a udržať tak metafasady zrozumitelnejsie.

AndroMDA využíva jednu základnú metafasadu a potom po jednej špecifickej pre každý zásuvný modul. Prostredníctvom jazyka Velocity je umožnené užívateľove metafasady upravovať.



obr. 20: Cartridge a metafasády

5.2 Cartridge

Cartridge v AndroMDA sú zásuvné moduly, pomocou ktorých sa spúšťajú transformácie na elementoch modelov, ktoré sú označené odpovedajúcim stereotypom. Z týchto elementov je následné generovaný výsledný kód, pomocou šablón definovaných v deskriptore cartridgu. Každá cartridge obsahuje okrem hlavného deskriptora aj súbory zo šablónami, špecifické metafasdy, namespace deskriptor a deskriptor UML profilu.

Deskriptor cartridge definuje mapovanie medzi objektmi z metafasad a šablónami. Aby sme mohli generovať zdrojový kód do rôznych zložiek, definuje tiež tzv. outlety, v ktorých užívateľ špecifikuje umiestňovanie jednotlivých generovaných súborov vramci koreňového adresára.

Špecifikácia parametrov používaných v celej cartridge sa nachádza v name space deskriptore. Tento súbor obsahuje informácie o možných konfiguráciách cartridge. Súbor tiež deklaruje používané premenne v cartridge a ich defaultne hodnoty.

5.3 Profil

Deskriptor UML profilu obsahuje všetky informácie o profile UML používaného pre tvorbu elementov spracovaných touto cartridge. Tak isto v ňom nájdeme popis stereotypov. XML štruktúra profilu vyzerá nasledovne:

```
…
<mdElement elementClass='PropertyManager'>
        <name>PROJECT_INVISIBLE_PROPERTIES</name>
        <propertyManagerID>_9_5_1_12ab03bf_1166891767265_495897_2</propertyManagerID>
<mdElement elementClass='NumberProperty'>
          <propertyID>TOOL_TIP_STYLE</propertyID>
          <propertyDescriptionID>TOOL_TIP_STYLE_DESCRIPTION</propertyDescriptionID>
          <value xmi.value='0.0'/>
          <type xmi.value='0'/>
          <lowRange xmi.value='0.0'/>
          <highRange xmi.value='2.0'/>
</mdElement>
<mdElement elementClass='NumberProperty'>
          <propertyID>LAST_INTERFACE_STYLE</propertyID>
          <propertyDescriptionID>LAST_INTERFACE_STYLE_DESCRIPTION</propertyDescri
ptionID>
          <value xmi.value='2.0'/>
          <type xmi.value='0'/>
          <lowRange xmi.value='0.0'/>
          <highRange xmi.value='1.0'/>
</mdElement>
<mdElement elementClass='BooleanProperty'>
          <propertyID>IS_BROWSER_VISIBLE</propertyID>
          <propertyDescriptionID>IS_BROWSER_VISIBLE_DESCRIPTION</propertyDescriptio
nID>
          <value xmi.value='true'/>
</mdElement>
…
```

5.4 Šablóny

Sú základným kameňom pre generovanie kódu. V súčasnej verzii AndroMDA nenájdeme žiadne šablóny pre transformácie modelov. Ako vstup pre transformácie vždy slúži PSM model, ktorý je reprezentovaný štruktúrou v pamäti, a výstupom je potom samotný zdrojový kód aplikácie.

Pre spracovanie šablón využíva AndroMDA jazyk VTL (Velocity Template Language) z projektu Apache Jakarta. Povodne bol tento jazyk využívaný k pridávaniu dynamického obsahu na webových strankach. Autori sa rozhodli používať VTL pre jeho relatívnu jednoduchosť a prehľadnosť. VTL používa referencie na vkladanie dynamického obsahu do výstupu. Premenne tvoria buď refenciu na určitú informaciu reprezentovanú v Java kóde, alebo svoju hodnotu získajú z inštrukcie (napr. priradenim). Sú zapisovane s prefixom „$“.

5.5 Generátor

Generátor je spúšťaný pomocou nastroja Maven. Alternatívne sa dá na tuto úlohu použiť aj Akt. Základom pre tento proces je súbor s konfiguráciou *pom.xml* a doprovodne súbory obsahujúce jednotlivé property. Mechanizmus transformácie a generovania sa skladá z nasledujúcich etáp:

1. Prevod modelu PIM do modelu PSM reprezentovaného v pamäti za pouzitia metafasad.
2. Načítanie cartridge a mapovanie medzi PSM a časťami našej aplikácie v závislosti na konfigurácii v súbore pom.xml
3. Asociovanie objektov v metafasadach s odpovedajúcimi šablónami
4. Generovanie kódu aplikácie
5. Ručná implementácia tried reprezentujúcich vnútornú logiku aplikácie
6. Úprava vzhľadu JSP alebo JSF (v prípade prítomnosti webového rozhrania)
7. Kompilácia a nasadenie aplikácie

Hlavným zdrojom informácii pre koordináciu celého mechanizmu je súbor pom.xml, ktorý obsahuje informácie o umiestnení zdrojov (modelov, profilov, archívov s cartridgami), a ich parametre a konfiguráciu.

5.6 Prispôsobenie nastroja

Aj keď nám AndroMDA vytvorí celu architektúru aplikácie, vždy sa nájdu špecifické požiadavky pre vyvíjaný systém. Konfiguračné súbory poskytujú iba limitovane množstvo premenných učených pre špecifické prispôsobenie. Naviac je potrebne uvážlivo voliť počet týchto premenných. Veľké množstvo umožňuje väčšiu obecnost a prispôsobivosť, ale samotná konfigurácia sa potom na úkor toho stáva zložitou. Naopak ak je premenných malo, nie je možné nastroj prispôsobiť požiadavkám aplikácie.
Pre prekonanie týchto obmedzení zavádza AndroMDA okrem konfiguračných súborov ďalšie mechanizmy. Tie nám umožňujú robiť zmeny v šablónach alebo priamo meniť cartrdige.

5.6.1 Preťažovanie šablón

Najčastejšou úpravou je preťažovanie šablón. Pomocou preťaženia sa riešia prípady, v ktorých chceme zmeniť určitú časť šablóny alebo vytvoriť vlastnú šablónu. Často sa menia predovšetkým šablóny pre JSP alebo JSF stránky, alebo časti užívateľského rozhrania.
Najjednoduchším riešením by bola priama zmena šablón. To sa však nedoporucuje pretože šablóna je zbalená v balíku jar spolu s ďalšími súbormi cartridge. Preto môžeme v nastaveniach každej cartridge špecifikovať miesto s našimi vlastnými šablónami. Generátor potom najskôr hľadá požadované šablóny v tomto umiestnení a až potom použije preddefinovane.

5.6.2 Zlučovanie cartridge

Preťažovanie šablón sa nijako nepremieta do štruktúry cartridge. Preto aby sme mohli zmeniť nejakú časť cartridge musíme využiť mechanizmus zlučovania. Pomocou tohto mechanizmu môžeme meniť elementy v XML súboroch deskriptorov cartridgu. Podobne ako pri preťažovaní šablón zavedieme do štruktúry súboru pom.xml premennú odkazujúcu na súbor s predpismi pre zlúčenie. Tieto predpis sú vždy vo forme kľuč a k nemu textová hodnota. Pri načítaní cartridge behom inicializácie transformácii sú najskôr vo všetkých XML súboroch a šablónach ktoré obsahuje cartridge nájdené všetky reťazce odpovedajúce určitému kľúču. Tie sú potom nahradzovane odpovednou hodnotou.

Cartridge a šablóny majú preddefinovane niektoré kľúče slúžiace pravé pre vkladanie nových elementov. Možnosti mechanizmu zlučovania sa však neobmedzujú iba na pracú s prednastavenými kľúčmi. Pretože sú takto spracovávané všetky textové súbory z cartrideg môžeme ručné upraviť ktorúkoľvek ich súčasť. To znamená že môžeme zadefinovať nove outlety, premenne, zmeniť mapovanie medzi objektmi metafasad a šablónami a podobne.

KAPITOLA 6
DEMONŠTRAČNÁ APLIKÁCIA TIME TRACKER

Za účelom dokonalého predvedenia celého procesu vývoja aplikácie pomocou metodiky MDA si podrobne popíšeme proces vývoja jednoduchej aplikácie Time Tracker. Účelom aplikácie bude evidencia zamestnancov a ich odpracovaných hodín. Aplikácia bude implementovaná ako webová aplikácia s databázou.
K vývoju bude použitý nastroj AndroMDA. Aplikácia bude využívať framework C#.NET. Ako databázoví server použijeme MS SQL 2005 s využitím technológie Hibernate. Výsledkom bude webová aplikácia bežiaca na IIS.

6.1 Zostavenie vývojového prostredia

Java JDK 1.6
Ako sme si už skôr vysvetľovali androMDA je napišaná v Jave. Preto ak chceme využívať tento nastroj musíme mat v prostredí nainštalovanú Javu. Pri tomto kroku je dobre k inštalácii zvoliť priamo JDK, pretože maven, projektový manažér, ktorý si neskôr predstavíme, k svojej činnosti potrebuje JDK.

Maven 2.1.0
Prvým krokom bude inštalácia projektového manažéra Apache Maven, ktorý bude slúžiť na celkovú organizáciu zostavovania aplikácie. Jedna sa o jednoduchý konzolový nastroj fungujúci na podobnom princípe ako veľmi dobre známy Ant.

AndroMDA 3.3
Ďalej je potrebne prepojiť generátor androMDA s projektovým manažérom maven. Toto urobíme veľmi jednoducho tým že správne nakonfigurujeme vzdialený repozitár mavenu. Následné pri spustení generačného procesu maven zabezpečí že všetky potrebne knižnice androMDA budú stiahnuté do nášho lokálneho repozitára.

Microsoft Visual Studio 2005
V súčasnosti je na trhu samozrejme dostupná aj vyššia verzia VS, avšak k nasej prací s androMDA je potrebný plugin, ktorý je momentálne kompatibilný iba s verziou 2005.

Microsoft SQL Server 2005
Využijeme ho ako databázový server. Je dodávaný spoločné s inštaláciou VS 2005.

Android/VS
Plugin do visual studia 2005, ktorý je prispôsobený priamo na pracú s adnroMDA. Je voľne dostupný na serveroch androMDA.

Magic Draw UML
Je nástroj na tvorbu UML diagramov. V súčasnosti je dostupná verzia 16.0, ktorá dôkaze generovať UML2.0. Pre náš konkrétny projekt použijeme verziu 9.5, ktorá bola bohužiaľ ako posledná schopná generovať UML modely verzie 1.4, ktoré potrebujeme ako vstup do androMDA. Na webe výrobcu je možné stiahnuť tzv. community verziu, ktorú je možné používať bezplatne a na náš projekt bude bohato stačiť.

Po nainštalovaní všetkých súčasti vývojového prostredia by ich umiestnenie malo vyzerať asi takto:

C:\Program Files\Java\jre1.6.0	Java Runtime Engine
C:\Program Files\Java\jdk1.6.0	Java Development Kit
C:\Documents and Settings\< prihlasovacie meno>\.m2	Maven local repository
C:\Program Files\Apache Software Foundation\Maven 2.1.0	Maven
C:\Program Files\Microsoft SQL Server	Microsoft SQL Server
C:\Program Files\Microsoft Visual Studio 8	Microsoft Visual Studio
C:\Program Files\Android VS	Android/VS
C:\Program Files\MagicDraw UML	MagicDraw Community Edition

Pre korektne spúšťanie projektového manažéra z konzole musíme nastaviť systémové premenne takto:

M2_HOME	C:\Program Files\Apache Software Foundation\apache-maven-2.1.0
JAVA_HOME	C:\Program Files\Java\jdk1.6.0
PATH	%JAVA_HOME%\bin; %M2_HOME%\bin

6.2 Vývoj

6.2.1 Fáza 1 : Generovanie nezávislého modelu

Spustime VS a pomocou pluginu Android/VS špecifikujeme, ktoré súčasti chceme aby aplikácia obsahovala. Je možné do aplikácie zakomponovať aj unit testy, ktoré sú nevyhnutnou súčasťou každej aplikácie. Pre náš demostracny účel však testy do aplikácie nezakomponujeme. Po špecifikovaní všetkých dôležitých údajov nám plugin vygeneruje nezávislý model aplikácie, ktorý sa skladá z týchto projektových zložiek:

```
TimeTracker.Web

TimeTracker.Web.Common

TimeTracker.Core

TimeTracker.Common

TimeTracker.SchemaExport
```

Najdôležitejším výstupom však je samotný model ktorý sa nachádza v súbore:

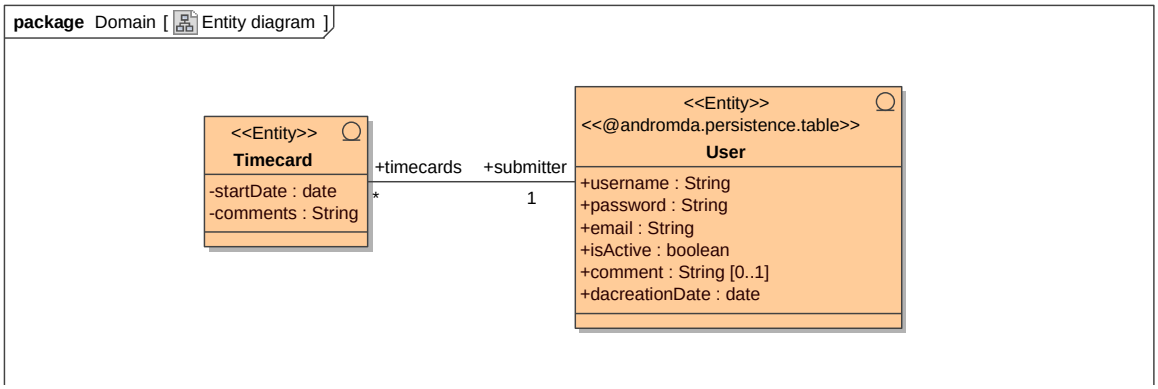
```
<solution folder> \mda\src\uml\TimeTrackerModel.xmi
```

6.2.2 Fáza 2 : Tvorba UML modelu aplikácie

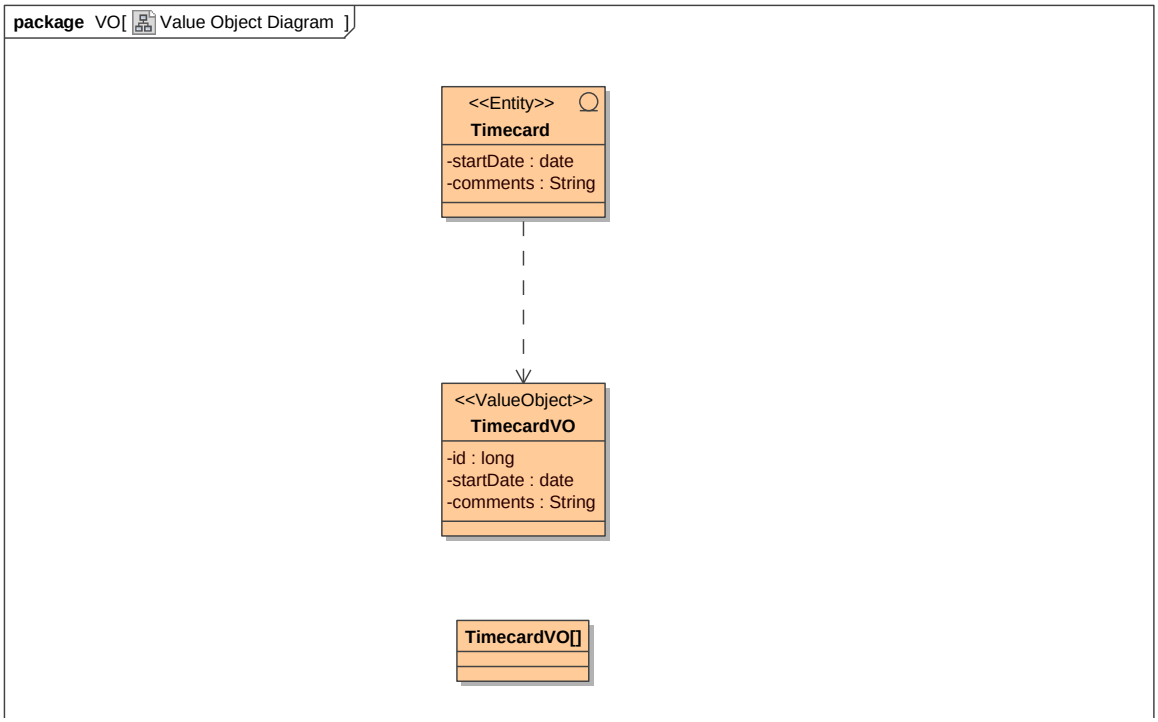
Súbor TimeTrackerModel.xmi teraz otvoríme v UML editore Magic Draw. Model obsahuje jednotlivé vrstvy aplikácie :

```
-Data
 -TimeTracker
  -Domain
   -Relations
   -Role
   -Role[]
   -User
   -UserRole
  -Service
   -Relations
   -MembershipService
  -VO
   -UserVO
```

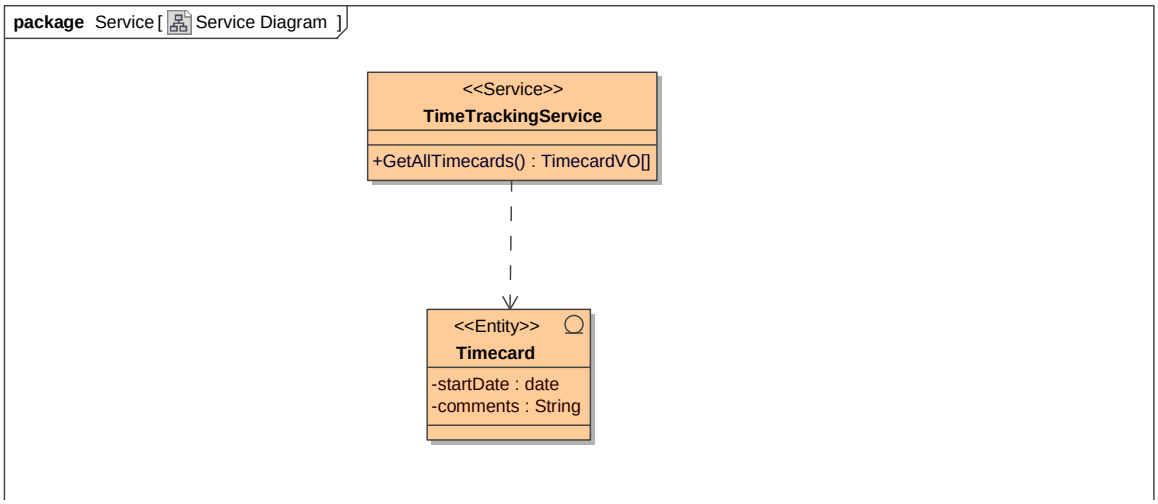
Našou úlohou je teraz vytvoriť UML diagramy pre jednotlivé vrstvy aplikácie tak aby spĺňali vnútornú logiku aplikácie:



obr. 21: Entity diagram



obr. 22: Value object diagram



obr. 23: Service diagram

Takto upravený model uložíme do súboru TimeTrackerModel.xmi vo formáte XMI 1.2. Následné vygenerujeme z modelu nove objekty do aplikácie pomocou generátora androMDA. To urobíme tak ako vždy, kliknutím na ‚Generate' v prostredí androidVS alebo jednoducho z konzole prostredníctvom manažéra Maven príkazom:

```
mvn install
```

6.2.3 Fáza 3 : Finálna manuálna úprava kódu

Po vygenerovaní môžeme vidieť že jednotlivé projektové zložky a aj koreňová zložka obsahujú priečinky ‚/src' a ‚/target'. Do týchto priečinkov sú generovane výstupy z androMDA. V zložkách ‚/src ' sú objekty, ktoré môžeme upravovať ručné a prispôsobovať ich k naším potrebám. U týchto zubrov dochádza ku generovaniu iba raz, tým sa zabezpečí perzistencia našich zmien v súbore, teda aj pri opätovných procesoch tieto súbory nebudú zmenene. V zložkách ‚/target' sú naopak objekty, ktoré by sme nemali ručné upravovať. Poslednou fázou vývoja je manuálna úprava aplikácie. Pre náš projekt to teda znamená implementácia ASP.NET formuláru, ktorý bude slúžiť na zobrazovanie výstupu aplikácie a vytvorenie databázy.

V projektovej zložke TimeTracker.Web teda vytvoríme formulár s prvkom DataGridView:

```
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
 <form id="form1" runat="server">
    <div style="text-align: center">
       <asp:Label ID="Label1" runat="server"
      Text="GetAllTimecards()"></asp:Label>
      <br />
      <asp:GridView ID="GridView1" runat="server"
      AutoGenerateColumns="false">
      <Columns>
      <asp:BoundField DataField="ID" HeaderText="ID" />
      <asp:BoundField DataField="StartDate" HeaderText="StartDate" />
      <asp:BoundField DataField="Comments" HeaderText="Comments" />
      </Columns>
      </asp:GridView>
      <br />
       
     </div>
  </form>
```

A vložíme kód do pozadia formuláru:

```
using TimeTracker.Service;
using TimeTracker.VO;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            ITimeTrackingService service = new TimeTrackingService();
            TimecardVO[] timecards = service.GetAllTimecards();
            GridView1.DataSource = timecards;
            GridView1.DataBind();
        }

    }
}
```

Implementujeme metódy pre konverziu objektov a entít v triede `TimeTracker.Core\src\TimeTracker\Domain\TimecardDao.cs`

```
public partial class TimecardDao
    {
        /// <summary>
        ///                                                                 @see
TimeTracker.Domain.ITimecardDao#ToTimecardVO(TimeTracker.Domain.Timecard)
        /// </summary>
        public                                  TimeTracker.VO.TimecardVO
ToTimecardVO(TimeTracker.Domain.Timecard entity)
        {
            TimecardVO valueObject = new TimecardVO();

            valueObject.StartDate = entity.StartDate;
            valueObject.Comments = entity.Comments;
            valueObject.Id = entity.Id;

            return valueObject;
        }

        /// <summary>
        ///                                                                 @see
TimeTracker.Domain.ITimecardDao#TimecardVOToEntity(TimeTracker.VO.Timecar
dVO)
        /// </summary>
        public                                  TimeTracker.Domain.Timecard
TimecardVOToEntity(TimeTracker.VO.TimecardVO timecardVO)
        {
            Timecard entity = Timecard.Factory.NewInstance();

            entity.StartDate = timecardVO.StartDate;
            entity.Comments = timecardVO.Comments;

            return entity;
        }

    }
```

Implementujeme metódu služby, ktorá nám vráti všetky objekty z tabuľky TIMECARD v triede `TimeTracker.Core\src\TimeTracker\Service\TimeTrackingService.cs`

```
public partial class TimeTrackingService
    {
        /// <summary>
        ///                                                                 @see
TimeTracker.Service.TimeTrackingService#GetAllTimecards()
        /// </summary>
        protected TimeTracker.VO.TimecardVO[] HandleGetAllTimecards()
        {
            IList timecards = this.TimecardDao.LoadAll();
            IList                       timecardVOs                     =
this.TimecardDao.ToTimecardVOList(timecards);
            TimecardVO[] voarray = new TimecardVO[timecardVOs.Count];
            timecardVOs.CopyTo(voarray, 0);
            return voarray;
        }
```

Poslednou úpravou bude vytvorenie databáze. Manuálne vytvoríme databázu s názvom TrackerDB napriklad v prostredi SQL Studia.

Skompilujeme projektovú zložku TimeTracker.SchemaExport. Výsledkom procesu bude vytvorenie tabuliek a väzieb v databáze TrackerDB.

Podmienkou pre tento proces je nastavenie property connectionString v konfiguračnom súbore nhibernate.config :

```
<property
name="hibernate.connection.connection_string">server=.\SQLEXPRESS;databas
e=TrackerDB;Integrated Security=SSPI;
</property>
```

Výsledkom úspešnej kompilácie projektu SchemaExport je databáza `TrackerDB`. Pre demonštračne učely vložíme do databáze demonštračne záznamy. Možzeme nato využit napríklad tento skript:

```
DELETE FROM TIMECARD
DELETE FROM USERS

SET IDENTITY_INSERT USERS ON
INSERT INTO USERS
(ID, UserName, Password, Email, Is_Active, Comment,Creation_Date)
VALUES
(1, 'bob', 'n/a', 'bob@bob.net', 1, 'comment', getdate())
SET IDENTITY_INSERT USERS OFF

SET IDENTITY_INSERT TIMECARD ON
INSERT INTO TIMECARD
ID, START_DATE, COMMENTS, SUBMITTER_FK)
VALUES
(1, getdate(), 'This is the first timecard', 1)
INSERT INTO TIMECARD
(ID, START_DATE, COMMENTS, SUBMITTER_FK)
VALUES
(2, getdate(), 'This is another timecard', 1)
SET IDENTITY_INSERT TIMECARD OFF
```

Takto vytvorenú aplikáciu vo finále skompilujeme a spustime. Výsledkom je formulár zobrazujúci výstup metódy GetAllTimecards().

ZÁVER

Tak ako počiatočným rozborom, aj konečnou demonštračnou aplikáciou sme sa presvedčili že MDA je skutočné stále veľmi agilná metodika vývoja softwarových produktov. Jej hlavnými prínosmi sú hlavne prenositeľnosť a relatívne rýchly vývojoví proces.
Je však dôležité si uvedomiť že technológia nie je vhodná pre všetky typy aplikácii. Presvedčili sme sa že napríklad na vývoj klasických vrstvovanych webových aplikačných riešení je technológia použiteľná veľmi rýchlo a dobre. Najmä preto že pri takých aplikáciách sa návrhové vzory malo odlišujú a objekty sa viac-menej iba obmieňajú. Metodika naopak nie je vhodná pre veľmi špecifické aplikácie, alebo v prípadoch, kedy chceme zasahovať už do vytvorenej aplikácie, z toho dôvodu, že jej prechod do samotného modelu by vzhľadom na zmeny ktoré chcem uskutočniť nebol dostatočné časovo efektívny.
V súčasnosti je metodika rozšírená najmä v open-source komunitách, čo tak isto tejto technológii podľa môjho názoru pridáva na hodnote. Komerčné vývojové prostredia sú tiež v určitom množstve zainteresovane do tejto metodiky, avšak nie až v tak veľkom objeme. Všeobecné ale môžeme konštatovať že tato metodika sa v súčasnosti stáva čoraz viac populárnou a žiadanou.
K faktorom, ktoré negatívne vplývajú na rozvoj MDA patri hlavne nekompatibilita modelových formátov. Tato situácia je však z časti zapríčinená aj príchodom UML 2.0 a neustálim vývojom a pridávaním elementov XMI formátu. Rozhodne veľkým prínosom pre technológiu by bola štandardizácia týchto formátov, čím by sa zabezpečila do určitej miery aj spätná kompatibilita. Verím že taký krok by sa pozitívne prejavil aj na vstupe komerčnej sféry do metodiky vo väčšom množstve.
MDA je agilná metodika, ktorá si určite zaslúži väčšiu pozornosť , najmä vďaka veľkej efektivite, ktorá je v súčasnosti dôležitým faktorom na trhu vývoja softwarových riešení.

Zdroje použitých informácii

MDA (Model Driven Architecture), LBMS, 2002-2009
http://www.lbms.cz/Reseni/Tema/MDA.htm

Simple and Practical Model Driven Architecture (MDA), Ibrahim Levent, 2008
http://fromapitosolution.blogspot.com/2008/11/simple-and-practical-model-driven.html

MDA and Model Transformation, Johan den Haan, 2008
http://www.theenterprisearchitect.eu/archive/2008/02/18/mda_and_model_transformation

MDA Specifications, Object Management Group Inc, 1997-2009
http://www.omg.org/mda/specs.htm

MDA Guide Version 1.0.1, OMG, 2003
http://www.omg.org/docs/omg/03-06-01.pdf

Meta Object Facility (MOF) 2.0 Core Specification, OMG, 2003
http://www.omg.org/docs/ptc/03-10-04.pdf

UML 2.0 Infrastructure Specification, OMG, 2003
http://www.omg.org/docs/ptc/03-09-15.pdf

MOF 2.0/XMI 2.1.1 Mapping, OMG, 2007
http://www.omg.org/docs/formal/07-12-01.pdf

Eclipse Modeling Framework, Wikipedia
http://en.wikipedia.org/wiki/Eclipse_Modeling_Framework

Unified Modeling Language, Wikipedia
http://en.wikipedia.org/wiki/Unified_Modeling_Language

Common Object Request Broker Architecture, Wikipedia
http://en.wikipedia.org/wiki/CORBA

Getting Started .NET, AndroMDA.org, 2009
http://galaxy.andromda.org/index.php?option=com_content&task=category&sectionid=12&id=43&Itemid=90

A Discussion of UML and Model-Driven Development, Mariangela Orme, 2008

Lessons Learned From the Model-Driven Architecture Applied to Critical Systems Reliability, Tristan Cordier, 2006

Návrh systému pomocí MDA, Petr Klemšinský, 2006

Model Driven Architecture Approach for Software Development in Embedded System, Gutema Jira, 2007

Prílohy

Obsah priloženého CD:

/text – text tejto bakalárskej prace v elektronickej forme

/ demo_application – zdrojové kódy demonštračnej aplikácie