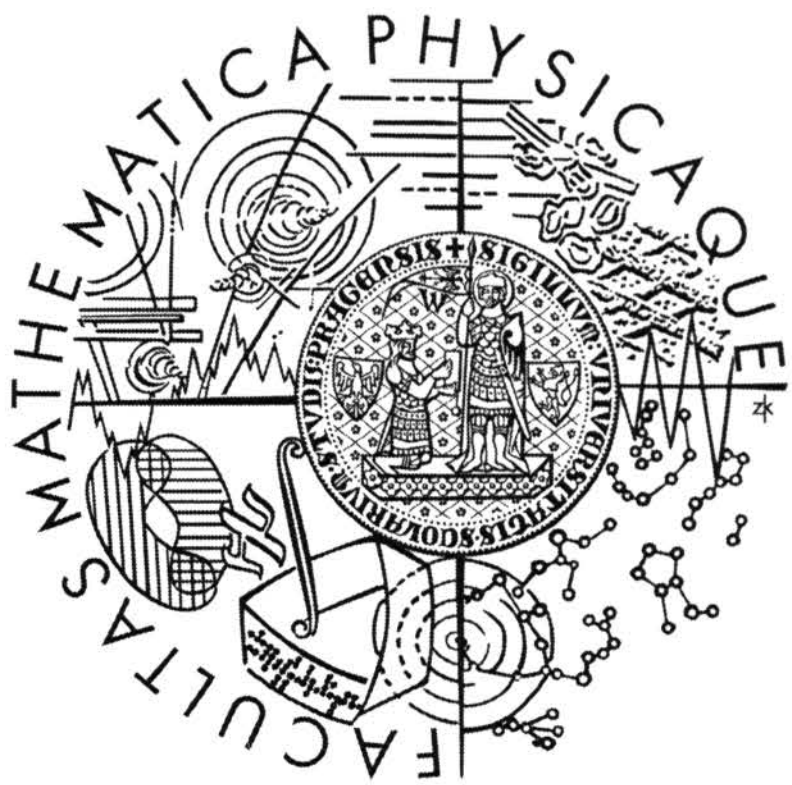
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Lukáš Bartoň

Vlastnosti MDA (Model driven architecture) a možnosti kombinace MDA s jinými způsoby specifikace požadavků

Katedra softwarového inženýrství

Vedoucí diplomové práce: *Prof. RNDr. Jaroslav Král, DrSc.*

Studijní program: *Informatika, Softwarové systémy*

Poděkování

Rád bych poděkoval prof. Jaroslavu Královi za odborné vedení práce, poskytnutí cenných rad a připomínek při zpracování práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 20.4.2006

Lukáš Bartek

.....

Obsah

Abstrakt česky

Název práce: *Vlastnosti MDA (Model driven architecture) a možnosti kombinace MDA s jinými způsoby specifikace požadavků*

Autor: *Lukáš Bartoň*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *Prof. RNDr. Jaroslav Král, DrSc.*

E-mail vedoucího: *kral@ksi.ms.mff.cuni.cz*

Abstrakt: Cílem práce je prozkoumat teoretické a praktické aspekty používání Modelem řízené architektury (Model Driven Architecture – MDA). Dále si klade za úkol prověřit její praktické přínosy při vývoji informačních systémů, možnosti využití v oblasti Architektury orientované na služby (Service Oriented Architecture -SOA) a stav nástrojů, které MDA implementují.

První část práce se zabývá strukturou MDA. Popisuje úrovně abstrakce, které MDA definuje, a teorii nasazení MDA při vývoji. Druhá část je věnována potřebným standardům, jejich vlastnostem i nedostatkům. V třetí části jsou rozebrány možnosti kombinace MDA a Architektury orientované na služby. Jsou ukázány významné problémy realizace této kombinace, ale i několik konkrétních příkladů možného využití MDA v SOA. Poslední část zkoumá stav implementace MDA v nástrojích, shrnuje jejich možnosti a společné vlastnosti. V závěru jsou zformulovány některé otevřené otázky a naznačeny možné směry dalšího rozvoje MDA.

Klíčová slova: *Modelem řízená architektura, Architektura orientovaná na služby, modelování, transformace modelů*

Abstrakt anglicky

Title: *Properties of MDA and the ways of combination of MDA with other requirement specification techniques*

Author: *Lukáš Bartoň*

Department: *Department of Software Engineering*

Supervisor: *Prof. RNDr. Jaroslav Král, DrSc.*

Supervisor's e-mail address: *kral@ksi.ms.mff.cuni.cz*

Abstract: The aim of the work is to explore theoretical and practical aspects of using Model Driven Architecture – MDA. Next goals are to examine contributions of using it for information system development in practice and possibilities of using it within Service Oriented Architecture – SOA and to investigate tools that implement it.

The first part of the work addresses MDA structure. It describes defined levels of abstraction and theory about using MDA in software development. The second part presents properties and inadequacies of utilized standards. Ways of combinations of MDA and SOA are analyzed in the third part. Main problems are exposed. Available examples are posted. The last part investigates tools that implement MDA. It summarizes their potentialities and common features. Open questions are expressed and directions of further development are denoted at the conclusion.

Keywords: *Model driven architecture, Service oriented architecture, modeling, model transformations*

1. Úvod

1.1. Důvod výběru tématu

Po celou dobu svého studia zároveň pracuji v oblasti vývoje software. Účastnil jsem se vývoje několika desítek aplikací od jednoduchých utilit až po bankovní aplikace využívající webové služby. Pracoval jsem v rolích analytika, návrháře, vývojáře i testera. Proto se zajímám o poslední trendy v oblasti vývoje a architektury software. Při hledání tématu pro svou diplomovou práci jsem vybíral mezi náměty blízkými oblasti mého zájmu. Další podmínkou bylo, že musí přinášet inovace. Nechtěl jsem analyzovat dobře známé a široce používané technologie nebo metody. Hledal jsem téma, při jehož popisu a analýze se sám něco nového naučím. Modelem řízená architektura toto bezezbytku splnila.

1.2. Stručný popis současného stavu

Modelem řízená architektura je nový, zatím nepříliš rozšířený, přístup k tvorbě software. Jejím tvůrcem je konsorcium Object Management Group (OMG), autor dobře známých a používaných standardů jako CORBA (Common Object Request Broker Architecture) nebo UML (Unified Modeling Language). Modelem řízená architektura spočívá v definování standardů a konceptů pro Modelem řízený vývoj (MDD), který se na různé úrovni používá již od 80 let. S pomocí těchto standardů můžeme vytvářet sofistikované modely, ze kterých je možné přímo generovat kód výsledné aplikace. MDA se stále rozvíjí, jsou však již k dispozici nástroje, které ji uplatňují v praxi.

„Vývojáři, kteří si osvojí Modelem řízenou architekturu, získají obrovskou volnost: mohou při změně infrastruktury vygenerovat nový zdrojový kód ze stabilních modelů. Rychlá návratnost investice je dána možností opakovaného použití aplikací a modelů po celou dobu životnosti softwaru."

(*Dr. Richard Soley, předseda a výkonný ředitel konsorcia OMG* [18])

1.3. Cíl práce, smysl cíle

Modelem řízená architektura je relativně nová a publikace o ní jsou většinou velmi teoretické. Proto se pokusím zmapovat současnou situaci nejen v teoretické oblasti, ale hlavně

v oblasti praktického použití, a zachytit nejnovější trendy v aplikaci MDA do praxe. Na praktických příkladech v různých nástrojích budu demonstrovat možnosti zahrnutých standardů a přínosy Modelem řízené architektury. Smyslem je ukázat, že již existují prakticky použitelné aplikace MDA přístupu, které naplňují vize Modelem řízené architektury, a sice, že proces tvorby informačního systému bude rychlejší, efektivnější a výsledek kvalitnější. Zároveň přitom ukážu nezralost některých standardů a vymezím oblasti, pro které standardy a modelovací nástroje zatím nejsou.

1.4. Způsob dosažení cíle

V prvních kapitolách proberu teoretické koncepty a standardy, na kterých je MDA postavena. Poté budu analyzovat možnosti kombinace MDA a Architektury orientované na služby (Service Oriented Architecture - SOA). Nakonec se zaměřím na nástroje s podporou MDA, budu na nich demonstrovat některé z výhod i problémů při nasazení MDA a kriticky zhodnotím možnosti použití těchto nástrojů.

Vzhledem k bouřlivému rozvoji v této oblasti, nedokonalostem v některých standardech, chybějícím standardům nebo jejich nedostatečné podpoře v nástrojích se budu věnovat i řešením a nástrojům, které nejsou založeny pouze na standardech OMG a spadají tak spíše do kategorie nástrojů pro MDD. Ani OMG zatím nástroje s podporou MDA necertifikuje. Na svých stránkách uvádí v seznamu nástrojů i ty, které nedodržují a nepoužívají některé základní standardy.

1.5. Omezení práce

Teoretická část samozřejmě nezachycuje všechny dostupné poznatky související s MDA a MDD. Problematika je popsána stručným a v některých případech zjednodušeným způsobem. Praktické příklady rovněž neslouží jako podrobný návod pro tvorbu aplikací v použitých nástrojích. Jejich smyslem je ukázat možnosti využití MDA.

1.6. Určení materiálu

Tato práce je určena čtenářům, kteří mají dobré znalosti v oblasti analýzy, návrhu a vývoje software. Některé části vyžadují hlubší znalosti architektury a souvisejících technologií. Předpokladem porozumění textu není znalost Modelem řízené architektury. Poznatky z této práce může využít analytik, architekt nebo vývojář hodlající přejít na MDA. Získá zde zá-

kladní přehled o nezbytné teorii, trendech dalšího vývoje v použití MDA – zejména o kombinaci MDA a SOA. Je uveden přehledový popis a kritické hodnocení nástrojů s podporou MDA. Připojeny jsou i ukázky z praktického použití Modelem řízené architektury.

Diplomová práce bude prezentována na mezinárodní konferenci uživatelů midrange systémů IBM pořádané sdružením Common Česká Republika 4. - 6. července 2006 v Hrotovicích. Vše popisované v práci lze uplatnit i mimo nástroje IBM. Firma IBM ani její obchodní partneři neposkytli žádnou přímou podporu při tvorbě této práce.

2. Vlastnosti Model Driven Architecture

V softwarových firmách se při tvorbě softwaru dnes většinou používá nějaká forma modelování pomocí jazyka UML. Např. při použití metodiky Rational Unified Process (RUP)[23] vytváříme modely během analýzy, podle nich další modely během návrhu a nakonec podle navržených modelů programujeme aplikaci. Děláme tedy stejnou činnost třikrát. Navíc když nalezneme chybu, která pochází z předchozí fáze, je zdlouhavé a nákladné opravovat ji v odpovídajícím modelu a je většinou opravena přímo v kódu. Modely potom nejsou v souladu s kódem a končí jen jako ilustrace v projektové dokumentaci. Aplikace je tak vytvářena technikami používanými již desítky let. Tento problém se pokouší řešit např. extrémní programování, které vynechává modelování a rovnou vytváří kód aplikace. Programovému kódu ale nerozumí analytik ani zákazník – není s ním možné komunikovat na dostatečně formální úrovni. Zároveň je projekt hned od počátku omezován použitými technologiemi.

Výhodnější je při specifikaci požadavků použít modely, kterým bude zákazník rozumět a které bude možné automaticky transformovat do výsledné aplikace. Transformace modelů do kódu je v CASE (Computer-aided software engineering) nástrojích podporována již velmi dlouho, ale zatím nebyla dostatečně komplexní. Nebylo možné provádět transformaci mezi modely ani vytvářet dostatečně přesné modely. Tento problém se snaží odstranit OMG pomocí souboru standardů a rámcových postupů zahrnutých pod „obchodní značku" Model Driven Architecture (MDA) – Modelem řízená architektura. Důsledkem jejího nasazení má být zvýšení rychlosti vývoje software při současném zvýšení kvality. Pro abstraktnější platformy – modely – se snáze programuje a díky automatickým transformacím budou odstraněny chyby, které vznikají při ručním převodu modelů do kódu.

2.1. Historie

Modelem řízená architektura je stará pouze několik let. Myšlenky, na kterých je založena přitom nejsou nijak nové. Už první platformově neutrální programovací jazyky (např. Cobol) umožnily zaměřit se na věcné problémy a oprostit se od problémů technologických. V 70. letech minulého století se objevilo „Software Product Line Engineering (Domain Engineering)". S rozvojem CASE nástrojů v 80. letech můžeme sledovat první pokusy o automatické generování kódu z modelu. Postupným zdokonalováním těchto metod vznikl pojem Modelem řízený vývoj (Model Driven Development – MDD), jehož je MDA podmnožinou –

jinými slovy MDA je MDD založený na standardech OMG. Do skupiny MDD spadají všechny metody, které staví vývojový cyklus na modelech. Kromě MDA to jsou např.:

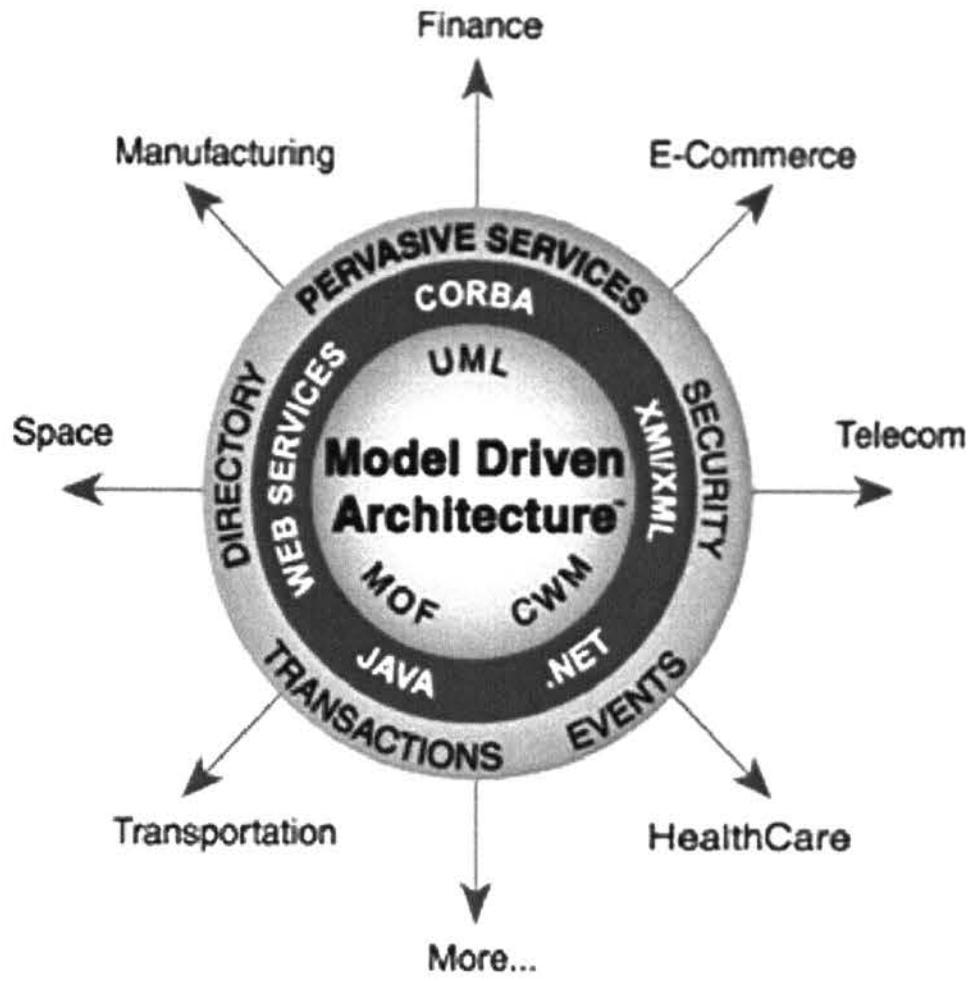
- **Rational Unified Process (RUP)** – snad nejznámější metodika pro vývoj software pomocí modelů, více v knize *The Rational Unified Process: An Introduction* [23].
- **Business Objevte Relation Modelling (BORM)** – původní česká metoda objektového návrhu systémů zaměřená zejména na úvodní fáze vývoje informačních systémů, je popsána v knize *Umění systémového návrhu, Objektově orientovaná tvorba informačních systémů pomocí původní metody BORM* [10].
- **Agile Modeling (agilní modelování)** – soubor principů, které kombinují vývojové metody extrémního programování s „nejlepšími" technikami modelování v Rational Unified Process, viz kniha *The Object Primer* [22].

Hlavním tvůrcem MDA je konsorcium OMG. V současné době do něj patří asi 800 společností – výrobců software i hardware. Hlavní úlohou OMG je tvorba a prosazování norem a specifikací pro nové technologie a softwarové platformy z oblasti informačních technologií. Neznámější a nejpoužívanější z nich jsou modelovací jazyk UML a CORBA. Nyní je hlavní pozornost konsorcia OMG směrována právě k MDA a navazujícím standardům. Architektura MDA nabízí standardy pro nový přístup k tvorbě platformově nezávislých aplikací.

2.2. Struktura

Architektura MDA je postavena na třech základních pilířích:

- **UML** (Unified Modelling Language) – jednotný jazyk pro modelování objektových systémů.
- **MOF** (Meta Object Facility) – abstraktní jazyk pro popis modelovacích jazyků (metamodelování).
- **CWM** (Common Warehouse Metamodel) – jazyk pro modelování dat.

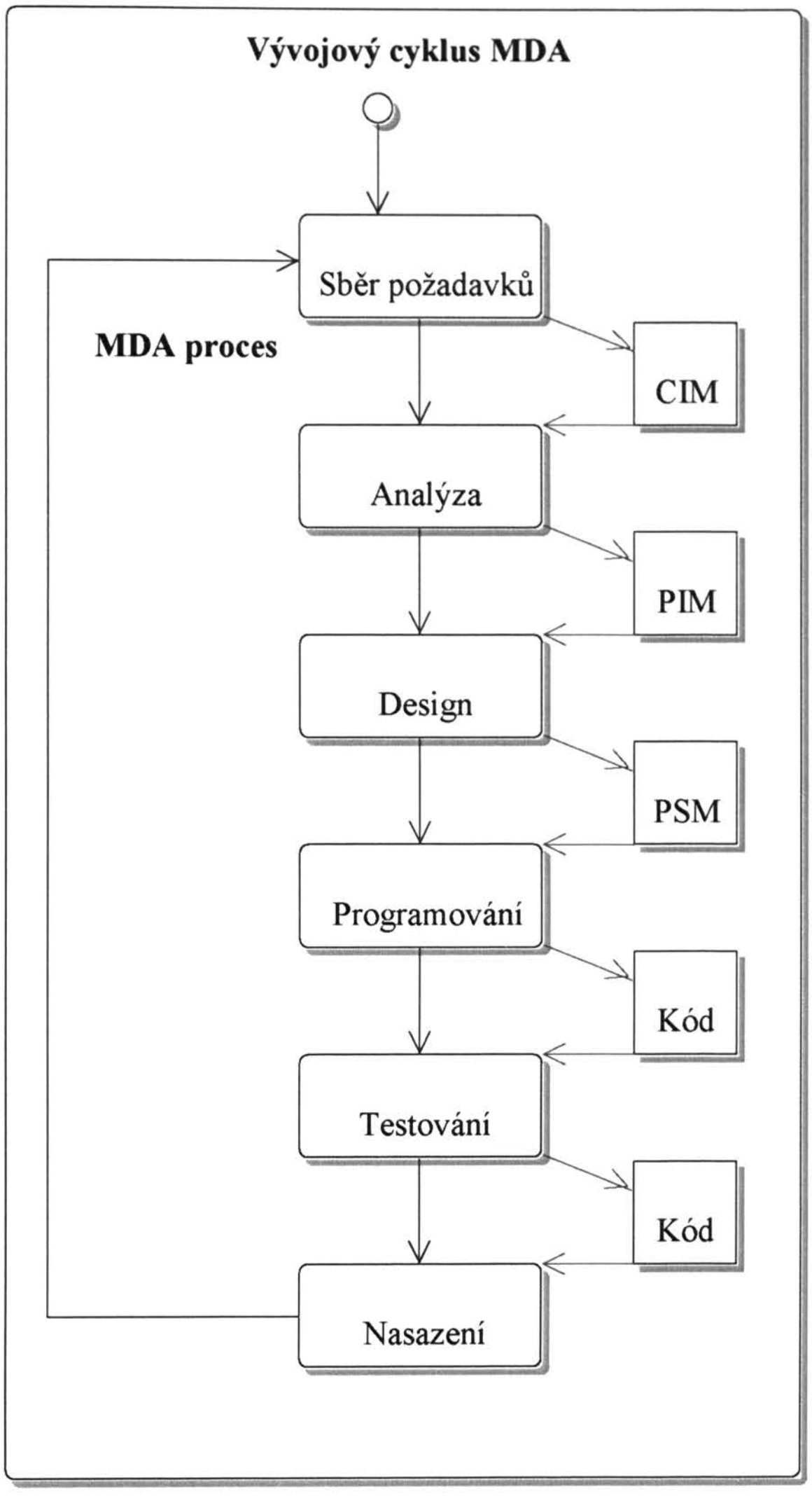


Obrázek 2-1 Struktura Modelem řízené architektury [16]

Struktura MDA je na obrázku 2-1. Na hlavní úrovni – v jádru – je samotná Modelem řízená architektura, která se „rozbaluje" ven pomocí druhé úrovně. Druhá úroveň obsahuje výše uvedené standardy UML, MOF a CWM. Na třetí úrovni jsou dnes všeobecně známé a používané softwarové technologie a platformy – CORBA, XML (Extensible Markup Language), .NET, Java. Dle OMG sem bude možné začlenit všechny budoucí platformy a technologie. Ve vnějším kruhu jsou platformově nezávislé služby, které je možné pomocí generovaného rozhraní začlenit do MDA aplikací. Šipky směřující ven ukazují možné oblasti obchodu a průmyslu, kde je možné MDA aplikovat.

2.3. *Vývojový cyklus MDA*

OMG navrhovaný vývojový cyklus MDA (viz obrázek 2-2) se příliš neliší od tradičního vodopádového modelu, případně RUP. Je možné identifikovat stejné fáze. Jedním z hlavních rozdílů je podstata artefaktů vytvářených během procesu vývoje.



Obrázek 2-2 Vývojový cyklus MDA

Artefakty jsou následující formální modely:

2.3.1. Computational Independent Model – Výpočetně nezávislý model (CIM)

Výpočetně nezávislý model zachycuje prostředí a požadavky systému. CIM popisuje systém z pohledu nezávislého na počítačovém zpracování. Technické detaily struktury a pro-

cesů v něm nejsou zachyceny. Není o nich zatím rozhodnuto, protože se jedná o nejvyšší úroveň abstrakce.

CIM je zaměřen na obchodní prostředí, ve kterém se bude systém používat. Model musí být srozumitelný uživateli systému a nemá obsahovat detaily o automatickém zpracování dat a informačních technologiích. Je zdrojem termínů, které budou použity v dalších modelech. Slouží jako pomůcka pro porozumění činnosti, kterou má systém podporovat.

CIM může zahrnovat několik UML modelů, většinou dle RM-ODP podnikových a informačních hledisek (Reference Model for Open Distributed Processing – sada ISO standardů). Při použití jazyka UML se jedná o *model případů užití* (use case model) a *model činností* (activity model).

Výpočetně nezávislý model bývá také označován jako *model domény* (domain model) nebo *obchodní model* (business model).

2.3.2. Platform Independent Model – Platformově nezávislý model (PIM)

Platformově nezávislý model zachycuje všechny automatizované procesy v systému, přičemž neobsahuje detaily nutné pro některou konkrétní platformu. Platformově nezávislý model kompletně specifikuje vytvářený systém. Model se nemění při změně platformy, tj. je přenositelný mezi různými platformami. Platformově nezávislý model má být výpočetně úplný – spustitelný. Je srozumitelný pro zadavatele systému (jedná se o prototyp) i vývojáře (testovatelná formální specifikace). Díky tomuto jsou jeho přínosy dlouhodobé.

Při vytváření platformově nezávislého modelu je možné použít obecné modelovací jazyky nebo jazyky vhodné pro oblast, ve které bude systém nasazen. Při použití jazyka UML se jedná zejména o *model tříd* (class model) a *stavový model* (state machine model), které využívají OCL (Object Constraint Language) a UML Action Semantic (aby byly výpočetně úplné).

2.3.3. Platform Specific Model – Platformově specifický model (PSM)

Platformově specifický model zachycuje propojení platformově nezávislého modelu s konkrétní platformou, na které bude systém nasazen. Tento model je úplný, vyhovuje všem omezením dané platformy. Je srozumitelný expertům na použitou platformu a technologie.

Při vytváření PSM používáme model platformy, který obsahuje technické pojmy pro popis konkrétní platformy. Model platformy poskytuje „součástky“, které můžeme použít ve specifikaci využití platformy v konkrétní aplikaci. Např. CORBA Component Model (CCM)

poskytuje pojmy EntityComponent, SessionComponent, ProcessComponent, Facet, Receptacle, EventSource apod., které používáme pro popis využití CORBA v aplikaci.

Použití modelu platformy se nejčastěji realizuje aplikací UML profilu nebo použitím modelovacího jazyka vytvořeného pro danou platformu. Používá se zejména *model tříd.*

2.3.4. Zdrojový kód

Posledním krokem ve vývoji je transformace všech platformově specifických modelů do kódu.

MDA nejen definuje čtyři základní úrovně abstrakce - CIM, PIM, PSM a kód. Zároveň říká, jak jsou propojeny. CIM je vytvářen manuálně a potom ručně nebo poloautomaticky převeden do PIM. Ten je převeden do několika PSM, které jsou transformovány do kódu. Nejsložitějším krokem je transformace z PIM do PSM. Například při vývoji J2EE aplikace je nutné provést transformaci PIM modelu tříd do PSM pro databázi, do PSM pro rozhraní a objekty v Javě, a do zdrojového kódu konfigurace a parametrů nasazení.

Požadavky na transformace, které budeme později sledovat jak na standardech, tak na jednotlivých nástrojích:

- **Trasovatelnost** transformace umožňuje dokumentaci vztahu mezi jednotlivými modely. Často je pro vývojáře důležité zjistit propojení prvků zdrojových a cílových modelů.
- **Otevřenost** – transformace mají být přístupné pro dodatečné úpravy, protože nemohou být dostatečně univerzální.
- **Přizpůsobitelnost** – transformace mohou mít nastavitelné parametry.
- **Obousměrnost** – změnou drobného detailu ve výsledném modelu a zpětným vygenerováním zdrojového modelu někdy dosáhneme požadovaného výsledku jednodušeji než přímou úpravou zdrojového modelu.
- **Inkrementálnost** – transformace by neměla ovlivnit proběhlé dodatečné úpravy v cílových modelech. Usnadňuje to přidávání nových prvků do zdrojového modelu.

Nyní popíšu jednotlivé typy transformací a vysvětlím jejich smysl:

2.3.5. Transformace CIM do PIM

Transformace z CIM do PIM je prováděna převážně ručně, protože musí proběhnout rozhodnutí, které části systému budou automatizovány. Zároveň musí byt výpočetně neúplný a neformální CIM model doplněn o detaily nutné pro další transformace.

2.3.6. Transformace PIM do PIM

Tato transformace se používá při vytváření nové verze modelu nebo při filtrování či specializaci modelu během životního cyklu systému. Transformace PIM do PIM je většinou prováděna ručně. Pro filtrování se použije jazyka QVT (Query, Views, Transformations, popis viz kapitola 3.3.1).

2.3.7. Transformace PIM do PSM

Měla by probíhat plně automaticky. Pouze volba cílové platformy je na vývojáři. Metody pro výběr vhodné platformy dle požadavků systému jsou popsány v práci *Bedir Tekinerdogan, Sevcan Bilir, Cem Abatlevi :Integratin Platform Selection Rules in the Model Driven Architecture Aproach* uvedené ve sborníku [7]. V praxi jsme navíc často omezeni požadavky zákazníka a použitými nástroji. Dále se PSM model doplní o informace, které není možné zahrnout do PIM modelu – diagramy nasazení, model uživatelského rozhraní apod.

Transformace z PIM do PSM jsou prováděny iterativně. PIM je opakovaně upravován, dokud není dosaženo požadované funkčnosti.

2.3.8. Transformace PSM do PSM

Tato transformaci slouží pro úpravy vygenerovaného PSM např. při přechodu na novou verzi platformy nebo při přenosu aplikace na jinou podobnou platformu.

2.3.9. Transformace PSM do zdrojového kódu

Je nejjednodušší a nejpřímočařejší transformace. Bývá označována také jako kompilace modelu.

Dále uvedené transformace jsou určeny hlavně pro reverzní inženýrství (RI – Reverse Engineering). RI slouží zejména k propojování nového systému s existujícími aplikacemi nebo k vytváření nového systému pomocí fragmentů existujícího.

2.3.10. Transformace zdrojového kódu do PSM

Její provedení je podobně jednoduché jako transformace opačným směrem. Často se používá už dnes pro zachycení statické struktury existujícího systému.

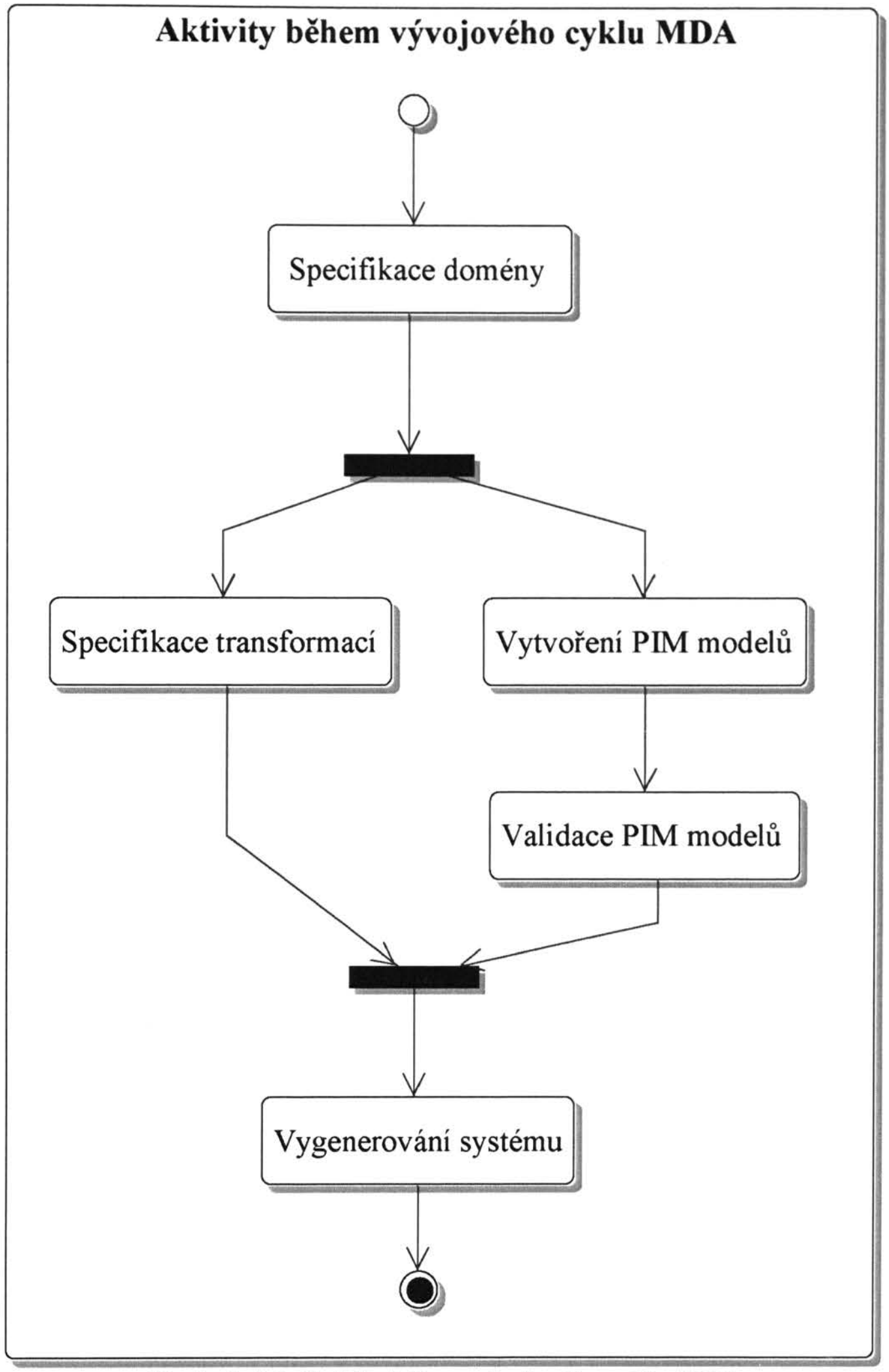
2.3.11. Transformace PSM do PIM

Automatická realizace této transformace je poměrně složitá. Výsledný PIM model musí být doplněn ručně. Výsledkem převodu je většinou pouze statická struktura systému, chybí např. invarianty, metody a akce. Ve spojení s předchozí popsanou transformací umožňuje např. vygenerování statického modelu podle databáze nebo rozhraní pro webové služby.

2.4. Metodiky pro nasazení MDA ve vývoji aplikací

Zatím není přesně specifikován způsob nasazení MDA při vývoji aplikací. Existují dva základní postupy: *generátor kódu* a *virtuální stroj interpretující model*. Podrobněji je toto popsáno v kapitole 5.1.5.

Nejrozšířenější je aplikace MDA pomocí generátoru kódu. Během vývojového cyklu se pak provádí činnosti zobrazené na obrázku 2-3. Následující odstavce popisují ideální případ, kdy jsou všechny transformace provedeny zcela automaticky. Vychází ze zkušeností autora práce s nástroji s podporou MDA, které většinou implementují metodiky pro OOAD (Object Oriented Analysis and Design - Objektová analýza a design).



Obrázek 2-3 Aktivity během vývojového cyklu MDA

2.4.1. Specifikace domény

Specifikace domény začíná vytvořením doménového modelu. Jsou vytvořeny nové nebo použity existující domény. Za pomoci těchto domén jsou vytvořeny CIM modely. Tuto činnost vykonává zejména analytik ve spolupráci se zákazníkem. Při vytváření nové domény můžeme zároveň vytvořit nový jazyk. Není to však nutné, pro většinu domén postačuje vytvořit UML profil, nástroje pro práci s UML profily jsou dnes běžné. V tabulce 2-1 je ukázka

takového profilu. Důležité je použít jazyk srozumitelný pro zákazníka. Např. při vývoji matematicky zaměřené aplikace je výhodnější používat jazyka MathML[19].

Stereotyp	Popis
<<BusinessActor>>	Účastnící byznys procesu.
<<BusinessEntity>>	Předměty byznys procesu.
<<BusinessWorker>>	Zdroje používané v byznys procesu.

Tabulka 2-1 Některé stereotypy z UML profilu pro modelování byznys procesů

2.4.2. Specifikace transformací

Specifikace transformace vyžaduje vytvoření mapování z metamodelu (domény) použitého pro PIM do metamodelů pro PSM. Následuje vytvoření transformací z PIM do PSM. Mapování a transformace nemusí být pro každou aplikaci vytvořeny znovu. V různých aplikacích lze použít stejné transformace (pokud jsou PIM i PSM v obou aplikacích ze stejné domény). Tyto transformace a jejich překladač mohou být zakoupeny jako součást nástroje, modul pro modelovací nástroj nebo samostatná aplikace.

Podobně je vytvořena transformace z PSM do zdrojového kódu. Některé z těchto transformací OMG standardizovalo, např. mapování UML do Javy. V tabulce 2-2 je uvedena část popisu transformace z UML do EJB (Enterprise JavaBeans).

Element PIM modelu – zdroj	Element(y) PSM modelu(ů) – cíl
Třída	Na EJB key class
Třída, které není součástí jiné třídy (kompozice)	Na EJB komponentu a EJB datové schéma
Třída	Na EJB datovou třídu uvnitř datového schématu, na které je mapována třída, která zdrojovou třídu zahrnuje v hierarchii kompozice
Asociace „uvnitř kompozice“	Na EJB asociaci uvnitř odpovídajícího EJB schématu (podobně jako předchozí pravidlo)
Asociace s třídou „uvnitř kompozice“	Na dvě EJB asociace a EJB datová třída
Atribut	Na EJB atribut odpovídající EJB datové třídy
Metoda	Na EJB metodu komponenty odpovídající třídě, které metoda patří (viz pravidlo výše)
Asociace „mezi kompozicemi“	Na EJB asociaci na EJB key class

Tabulka 2-2 Ukázka textového popisu transformace z UML modelu do UML modelu používajícího EJB profil

Přes vygenerovaná rozhraní jsou realizována napojení vytvářené aplikace na existující systémy. Pokud jsou tyto systémy také vytvořeny pomocí MDA, může být toto napojení generováno automaticky z PIM modelů těchto systémů.

Volba a vytváření transformací je úlohou architekta – odborníka na cílové platformy.

2.4.3. Tvorba PIM modelů

Pomocí profilů nebo jazyků pro použitou doménu se vytvoří podrobný a úplný model systému. Nejdříve se modeluje jeho statická strukturu, kterou v UML zachycuje diagram tříd. Následuje dynamický model, pro který se v UML používá stavový diagram a diagram spolupráce. Nakonec jsou do diagramu tříd a stavového diagramu doplněny těla operací a akcí. Zároveň jsou vytvořeny modely rozhraní pro napojení na existující aplikace.

2.4.4. Validace PIM modelů

Po dokončení PIM modelů mohou být tyto modely spuštěny a otestovány. Trasovatelnost případů užití z CIM modelu až na prvky vygenerované testovací aplikace umožňuje automatizaci některých testů.

2.4.5. Vygenerování systému

Nakonec jsou mapování a transformace použity pro převod PIM modelů do PSM modelů. PSM model je doplněn o informace potřebné k nasazení systému – je vytvořen diagram nasazení. Poté je pomocí PSM kompilátoru vytvořen kód systému, který je znovu otestován. Nyní může být systém nasazen.

2.4.6. Použití MDA s jinými metodikami

Teoreticky nejsou překážky pro použití MDA mimo metodiky pro OOAD. Bohužel zatím chybí vhodné nástroje a standardy. Ukazuje to přehled nástrojů v kapitole 5. Orientace OOAD metodik na tvorbu objektových softwarových systémů zatím omezuje možnosti použití MDA v některých jiných oblastech např. v SOA (viz kapitola 4.2).

2.5. Stav a budoucnost MDA (z pohledu standardů)

Vytvoření standardů pro složitou technologii jakou je MDA vyžaduje dostatek času. V červnu 2003 schválila OMG MDA Guide Version 1.0.1[20], druhou předběžnou specifikaci

MDA (první vyšla v roce 2001 pod názvem Model Driven Architecture – Technical Perspective[21]). Nová verze měla být schválena v polovině roku 2005, zatím se tak nestalo (duben 2006).

MDA není konkurencí pro stávající technologie (např. CORBA, .NET, J2EE). PIM model abstrahuje od těchto technologií, MDA tedy pracuje na vyšší úrovni abstrakce. Na technologiích jsou závislé až PSM a zdrojový kód. OMG přepokládá, že bude v budoucnu možné transformovat PIM do PSM nejen pro existující, ale i pro budoucí technologie (doba životnosti PIM modelů má být 15 až 20 let). MDA umožní propojení existujících a budoucích technologií. Pokud se podaří ideu MDA prosadit mezi výrobci vývojových nástrojů, může v budoucnu vypadat tvorba aplikací tak, jak popisuje předchozí kapitola: vytvoření platformově nezávislého modelu, použití vhodné transformace a kompilátoru modelu, a ve výsledku vznik aplikace propojené s požadovanou databází a fungující na vybrané platformě. V případě požadavku na změnu funkcionality takové aplikace bude upraven PIM a zopakována procedura vygenerování aplikace. Z PIM modelu bude také možné vygenerovat stejnou aplikaci pro nové platformy.

2.6. Možné důsledky nasazení MDA

Pojem „vývojář" získává při uplatnění principů MDA jiný význam. Už to není programátor, ale odborník na popis systému ve znalostní doméně jeho nasazení. Dojde k rozdělení na *vývojáře – programátora* (programuje v tradičních vývojových prostředích) a *vývojáře modelujícího aplikace*. Jak bude později ukázáno na příkladech v konkrétních nástrojích, je už nyní možná např. abstrakce od znalostí konkrétních SŘBD a není nutná ani znalost jazyka SQL.

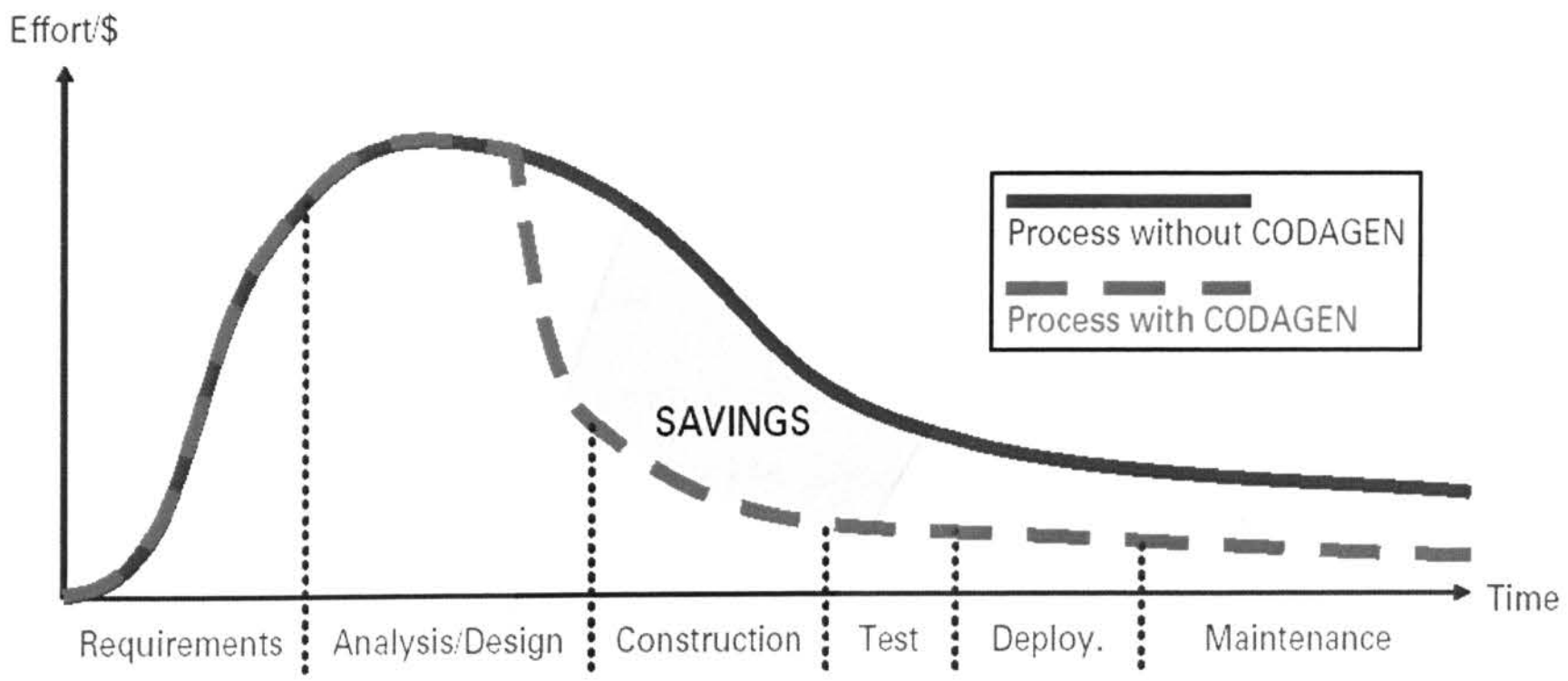
Nicméně vývojář – programátor samozřejmě nezůstane bez práce. Tvorba kvalitních nástrojů pro podporu MDA je mimořádně složitá i zajímavá. Programátor najde uplatnění i při tvorbě a optimalizaci transformací, úpravě PSM modelů a kódu vytvořeného z modelů. Zavedení MDA ho zbaví rutinní práce, jakou je ruční vytváření tříd a databázových tabulek podle modelu. Ty budou automaticky generovány MDA nástroji.

Důležitým důsledkem má být zvýšení kvality a zároveň produktivity. Dle odkazů z webu OMG vede použití MDA v reálných projektech ke zvýšení produktivity a tedy snížení nákladů až o desítky procent. Viz tabulka 2-3. Seriozní srovnávací studie na toto téma zatím neexistuje. Zveřejňovány jsou pouze úspěšné případy.

Firma – projekt	Použitý nástroj	Zvýšení produktivity
ABB – Web-based Simulation Toolbox	ArcStyler	O 45% méně programátoru a testerů
Lockheed Martin – F-16 Mission software	Kennedy Carter iCCG	Zkrácení času vývoje o 20%
Deutsche Bank Bauspar AG – správa vkladových a úvěrových účtů	ArcStyler	Úspory ve výši 40%
CGI – rozsáhlý software na zakázku	Codagen	Celkové úspory 26%, úspory se projeví ve fázích následujících po analýze (design, vývoj, testování), pro ty jsou 40 – 60%
E-SoftSys – Workflow	OptimalJ	Zvýšení produktivity o 40%
Rakouské spolkové dráhy – Rozvrhování provozu	ArcStyler	Snížení nákladů na vývoj o 34%
Credit Suisse	ArcStyler	Úspory 35 – 55%
M1 Global - Firemní informační systém	M1 Global MDE Studio	Úspory 62%
DaimlerChryler - Plánování výroby	ArcStyler	Úspory 15 – 30%

Tabulka 2-3 Úspory po nasazení nástrojů s podporou MDA[65]

Hlavní úspory lze předpokládat ve fázích tvorby platformově závislých modelů a kódu. Tj. během návrhu, vývoje a testování, které MDA částečně automatizuje pomocí transformací. MDA nijak neusnadní sběr požadavků a analýzu. Tento předpokládaný průběh úspor prezentuje i firma Codagen, viz obrázek 2-4.



Obrázek 2-4 Úspory při použití nástroje Codagen na vývoj rozsáhlého software na zakázku[65]

3. Standardy a technologie pro MDA

3.1. Čtyřvrstvá architektura modelů dle OMG

Pro lepší pochopení vztahu mezi různými standardy využívanými MDA je důležitá znalost definovaných modelovacích vrstev. OMG používá pro standardy čtyřvrstvou architekturu. Vrstvy označuje: M0, M1, M2 a M3.

3.1.1. Vrstva M0: Instance

Vrstva M0 odpovídá běžícímu systému, ve kterém existují konkrétní (reálné) instance. Příkladem instancí jsou *projekt Programování aplikace IDFÚ* a *zaměstnanci Lukáš Bartoň* a *Ivo Pospíšil* (v aplikaci Work Sequencer použité jako jeden z příkladů v této práci, viz kapitola 5.2). Instance mohou mít různá ztělesnění: řádek v databázové tabulce nebo objekt v právě běžícím programu.

3.1.2. Vrstva M1: Model systému

Vrstva M1 obsahuje modely – např. UML model softwarového systému. Ve vrstvě M1 je např. definován pojem *projekt* s atributy *název* a *popis*.

Mezi vrstvou M0 a M1 je pevný vztah. Pojmy z vrstvy M1 jsou kategorie instancí z vrstvy M0. Naopak každá položka z vrstvy M0 je instancí pojmu z vrstvy M1. *Projekt Programování aplikace IDFÚ* je instancí pojmu *projekt. Lukáš Bartoň* a *Ivo Pospíšil* jsou instancemi pojmu *zaměstnanec*.

Pojmy z vrstvy M1 přímo specifikují strukturu instancí ve vrstvě M0. UML třída *projekt* popisuje strukturu instancí projektů ve vrstvě M0. Instance nesplňující tuto specifikaci jsou nepřípustné.

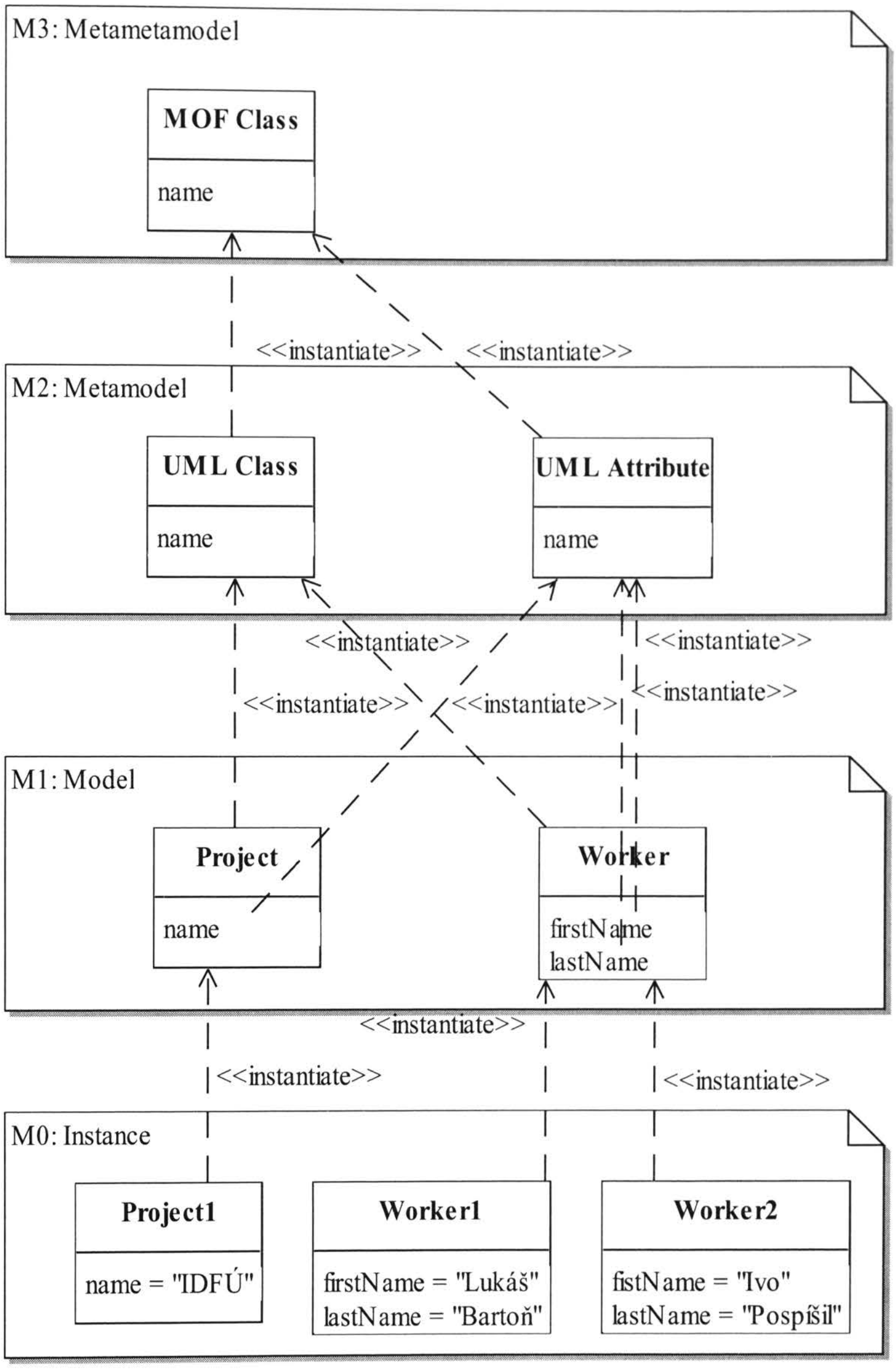
3.1.3. Vrstva M2: Model modelu

Elementy ve vrstvě M1 (třídy, atributy, apod.) jsou instancemi tříd z vrstvy M2. Element z vrstvy M2 specifikuje elementy z vrstvy M1. Mezi elementy z vrstvy M2 a M1 je stejný vztah jako mezi elementy z vrstvy M1 a M0. Model vytvořený ve vrstvě M2 nazýváme *metamodel*. Každý UML model z vrstvy M1 je instancí UML metamodelu definovaného ve spe-

cifikaci OMG. Při vytváření metamodelu vytváříme nový modelovací jazyk. Dalším příkladem takto definovaného jazyka je CWM.

3.1.4. Vrstva M3: Model M2

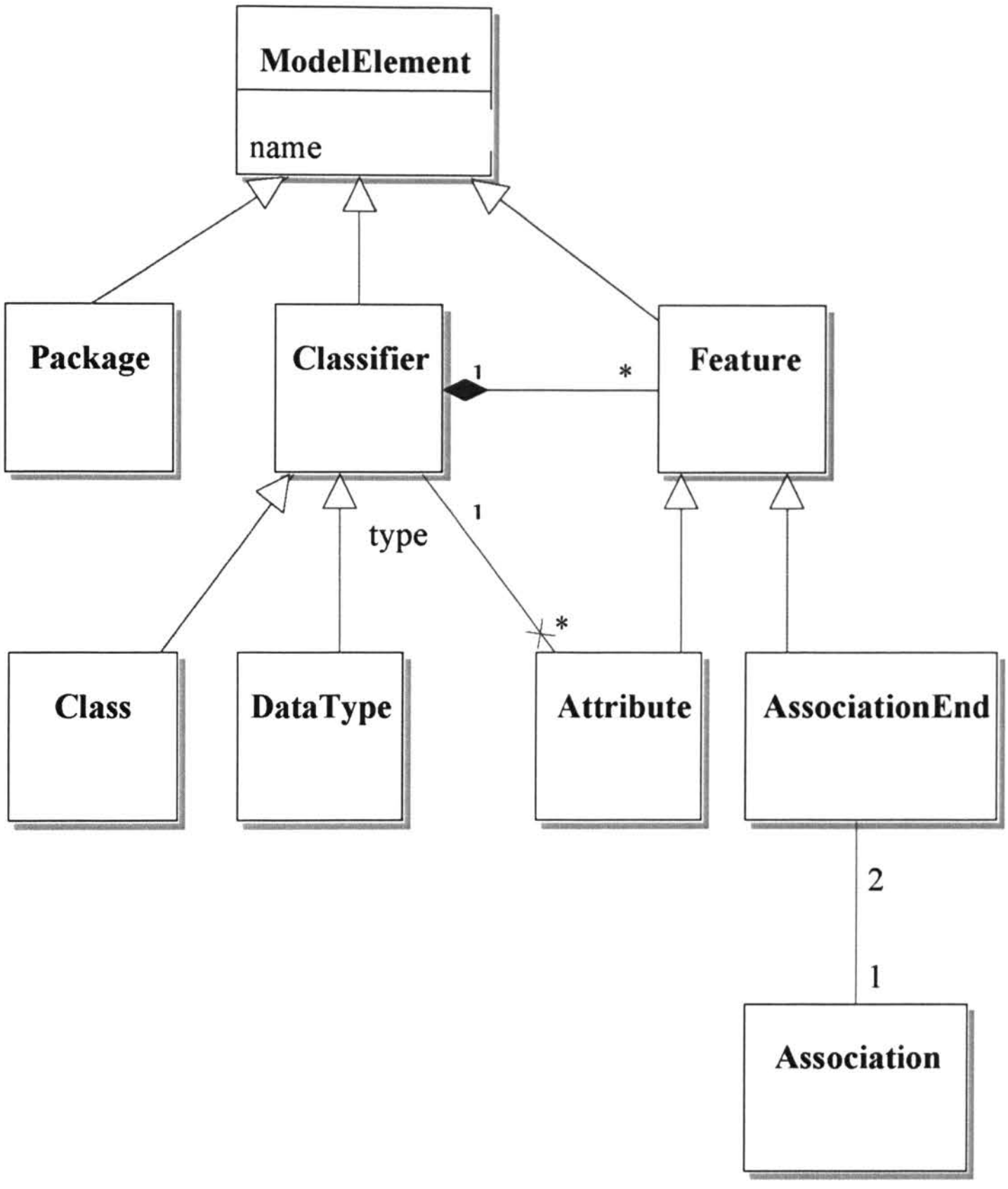
Stejně tak jsou i elementy vrstvy M2 instancemi elementů z vyšší vrstvy. Modely vytvořené ve vrstvě M3 označujeme *metametamodely.* Vrstva M3 definuje pojmy užívané při vytváření modelů (definování nových modelovacích jazyků) ve vrstvě M2. Obrázek 3-1 ukazuje vazby mezi elementy z vrstev M0 až M3:



Obrázek 3-1 Čtyřvrstvá architektura modelů dle OMG

3.2. MOF – Meta Object Facility

MOF je standardem OMG definujícím jazyk pro vytváření modelovacích jazyků. MOF odpovídá vrstvě M3, tj. nejvyšší vrstvě ze čtyř v předchozí kapitole uvedených. Jelikož neexistuje vyšší vrstva, je MOF definován sám v sobě. V MOF je vytvořen metamodel pro jazyky UML a CWM. Zjednodušená struktura MOF je zobrazena na obrázku 3-2.:



Obrázek 3-2 MOF metamodel

Metamodelování umožňuje MDA transformace, protože jeden z přístupů k MDA (viz kapitola 5.1.2.) **definuje transformace mezi metamodely vytvořenými pomocí MOF (mezi různými modelovacími jazyky nebo v rámci jednoho jazyka). Tento přístup usnadňuje znovupoužitelnost transformací a nezávislost nástrojů na jednom konkrétním modelovacím jazyku.**

OMG ve standardu MOF definuje nejen jazyk pro vytváření modelovacích jazyků, ale i další funkce užitečné zejména při vývoji modelovacích nástrojů:

3.2.1. MOF Repository Interface

Definice MOF obsahuje specifikaci rozhraní pro MOF úložiště. Toto rozhraní umožňuje získávání informací o uložených M1 modelech. Ve standardu Meta Object Facility IDL Language Mapping Specification[39] je definováno CORBA IDL pro toto rozhraní. Proto může být toto úložiště použito na různých platformách. Např. v Java existuje nativní rozhraní s touto funkcionalitou. Toto rozhraní specifikované firmou Sun se jmenuje Java Metadata Interface (JMI). Používá ho například nástroj openMDX (viz kapitola 5.9).

3.2.2. Výměna modelů

Další významnou součástí definice MOF je standard pro ukládání M1 modelů do souborů – na XML založené XMI (XML Metadata Interchange). Tento standard umožňuje výměnu modelů mezi různými nástroji. Je možné vytvářet modely v jednom nástroji a pro transformace používat jiný nástroj (např. modelovat v nástroji Poseidon a transformace provádět pomocí AndroMDA).

3.3. Metody pro popis transformací

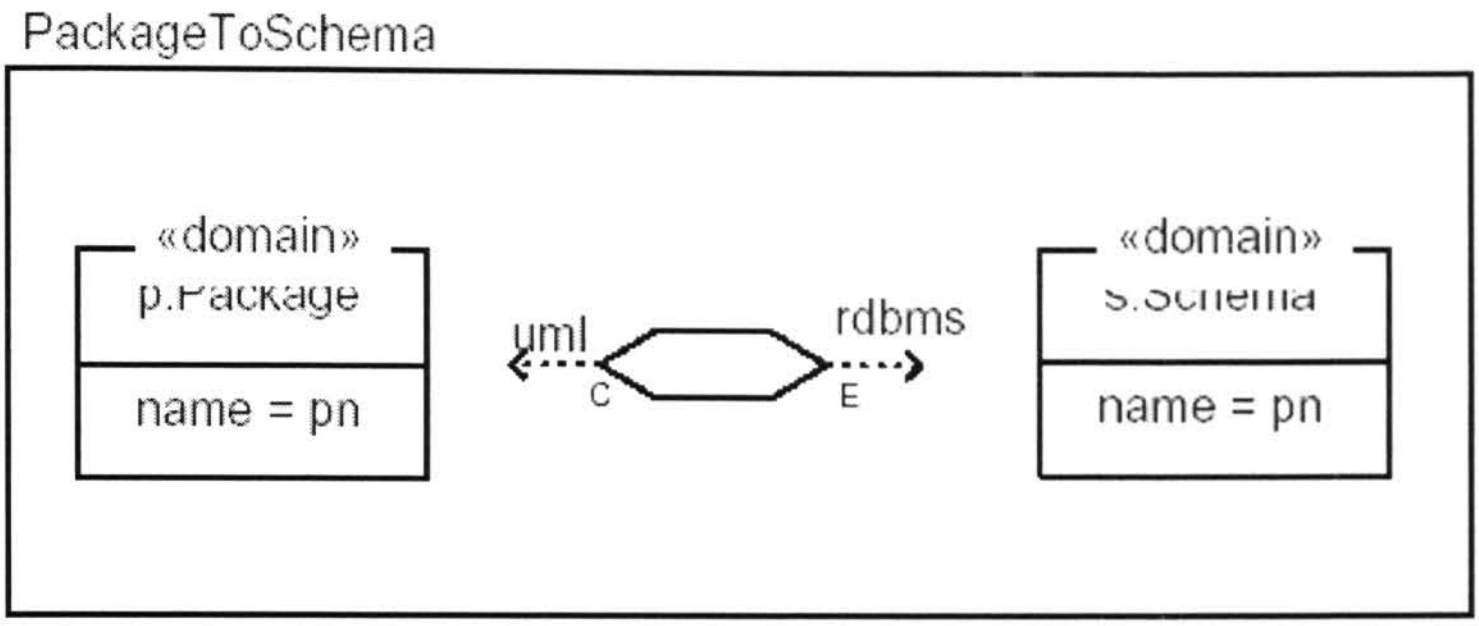
Oblast popisu transformací mezi modely a z modelu do kódu je nejsložitější a zároveň nejdůležitější v MDA. Standard QVT od OMG byl schválen teprve nedávno a není dostatečně zralý – není často podporován v nástrojích a popis transformací v něm je dosti složitý. Také chybí standardizovaný jazyk pro popis transformací z model do textu. Proto je popsán nejen tento standard, ale i další možnosti. V této kapitole jsou uvedeny spíše teoretické návrhy – prezentované na konferencích. V kapitole 5.1.10. jsou popsány další možnosti používané v praxi.

3.3.1. QVT – Query, Views and Transformations [29]

OMG schválilo jazyk QVT teprve v listopadu 2005. Je určen pro popis transformací mezi modely vytvořenými v jazycích, které jsou definovány pomocí MOF (QVT je součástí poslední verze standardu MOF). Je to deklarativní jazyk odvozený od jazyka OCL. Lze ho použít pro:

- vytváření pohledů na modely
- vytváření dotazů nad modely
- definice transformací mezi modely

Součástí standardu je textová i grafická syntaxe.



Obrázek 3-3 Ukázka grafického popisu transformace v QVT[29]

Ukázka popisu jednoduché transformace z UML 2.0 diagramu tříd do EMF diagramu tříd zapsaná v nástroji Borland Together 2006:

```
transformation Uml_To_Ecore;

metamodel 'http://www.borland.com/together/uml';
metamodel 'http://www.borland.com/together/uml20';
metamodel 'http://www.eclipse.org/emf/2002/Ecore';

mapping main(in pack: uml::kernel::packages::Package):
      ecore::EPackage {
 init{
    result := makePackage(pack);
    }
    object{
    }
}

mapping makePackage(in pack: uml::kernel::packages::Package):
ecore::EPackage {
 init {
var allClasses := pack.ownedMembers->select(c |
c.oclIsKindOf(uml20::classes::Class))->
      oclAsType(OrderedSet(uml20::classes::Class));
 }

 object {
```

```
    name := pack.name;
    eClassifiers += allClasses->collect(c | makeClass(c))
      ->asOrderedSet();
    eSubpackages := pack.nestedPackages->collect(p |
    makePackage(p))->asOrderedSet();
  }
}
mapping makeClass(in cls: uml20::classes::Class): ecore::EClass
{
    object{
      name := cls.name;
    }
}
```

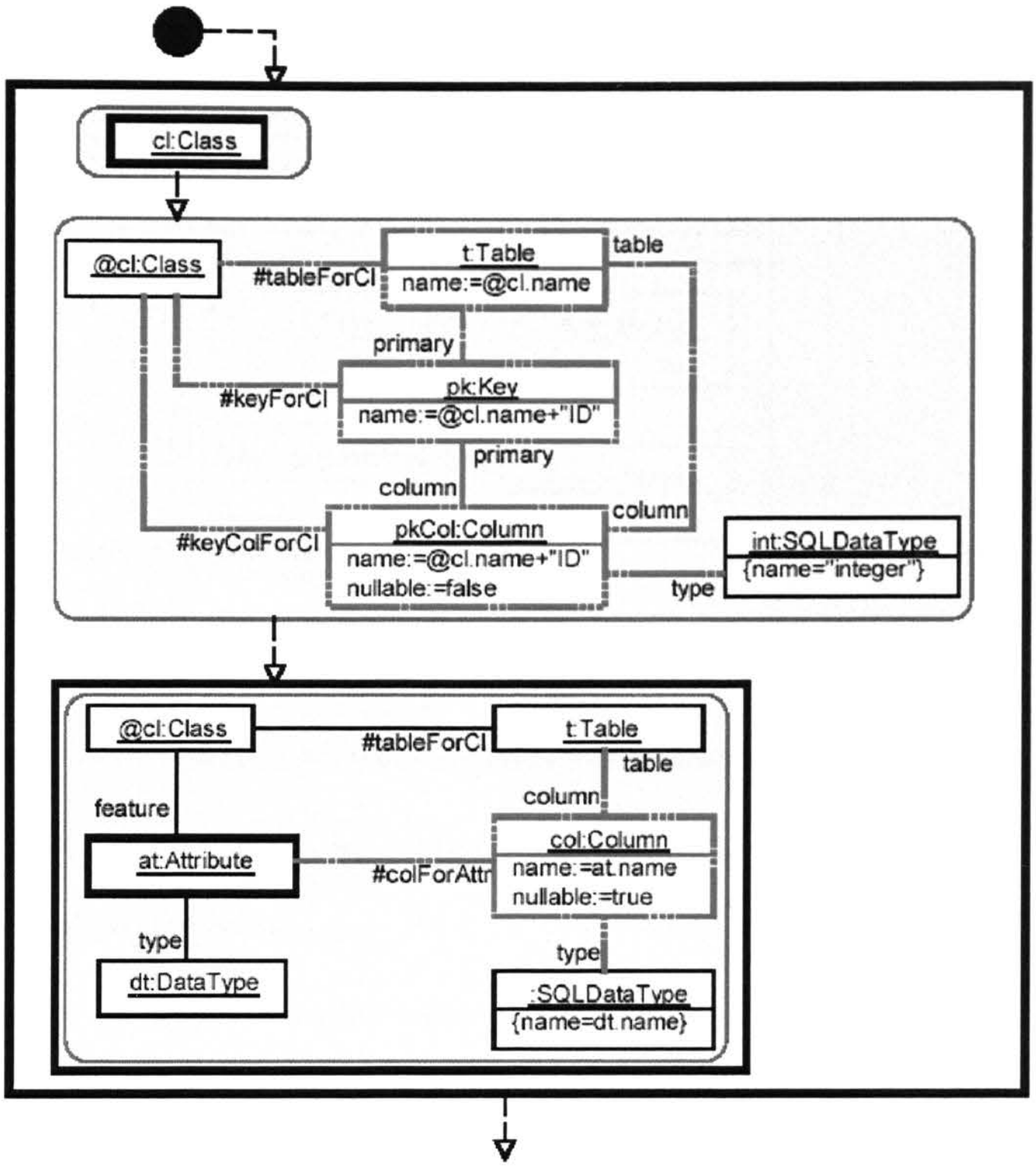
Za hlavní nedostatky QVT osobně považuji:

- Nedostatečnou podporu pro vytváření více PSM modelů z jednoho PIM, propojení výsledných PSM a podporu pro popis transformací více zdrojových modelů do více cílových modelů.
- Těžkopádnost při zápisu složitějších transformací: rekurzivní popis kopírování složitých grafů prvků je velmi nepřehledný. Jednoduchý popis transformace mezi podobnými prvky, např. z UML třídy do Java třídy, vyžaduje kompletní popis mapování všech typů atributů.
- Chybí podpora definice obousměrného mapování, která je potřebná pro dvoucestné transformace.
- Chybí podpora pro zápis omezení pro transformace.
- Deklarativnost a jednosměrnost transformací neumožňuje vytvářet dostatečné výkonné nástroje např. schopné porovnat rozdíly mezi existujícím zdrojovým a cílovým modelem.
- Deklarativní zápis transformací znesnadňuje vytváření algoritmicky složitých transformací, např. když je v cílovém modelu potřeba vytvořit strukturu prvků, která neodpovídá žádným elementům ze zdrojového modelu.
- Chybí podpora pro vytváření jmen pro různá prostředí. Např. UML, Java a C++ mají různá omezení a zvyklosti pro pojmenování tříd a atributů.
- Chybí podpora pro „dědičnost“ transformací – to znesnadňuje znovupoužitelnost.

Protože standard QVT byl schválen teprve nedávno, vzniklo velké množství dalších jazyků pro popis transformací. Důsledkem pozdní adopce standardu je i malá podpora QVT v existujících nástrojích. Grafickou syntaxi QVT v tuto chvíli (duben 2006) nepodporuje žádný nástroj. Kromě dále zmiňovaných jazyků pro popis transformací existuje mnoho dalších: YATL – Yet Another Transformation Language[26], UMLX[27], ATL[28], ...

3.3.2. Model Transformation Language MOLA

Jazyk MOLA je grafický, kombinuje tradiční strukturované programování s pravidly pro transformace založenými na vzorech. Jeho cílem je poskytnutí přirozené a vysoce čitelné prezentace pravidel pro transformace mezi modely. Jeho základem je grafický zápis cyklů FOREACH a WHILE. Na obrázku 3-4 je FOREACH znázorněn černým tučným obdélníkem – cyklus je prováděn pro každou třídu a pro každý atribut. Elementy rozpoznávané pomocí vzorů zachycuje černý obdélník, nově vytvářené elementy a asociace jsou zakresleny červeně. Vnoření dalšího cyklu FOREACH znázorňuje šipka mezi šedými rámečky. Elementy vytvořené na vyšší úrovni mohou být vstupem pro nižší úroveň (cyklus je proveden pro každou třídu a pro všechny její atributy).



Obrázek 3-4 Ukázka popisu transformace modelu tříd do "ER modelu"[7]

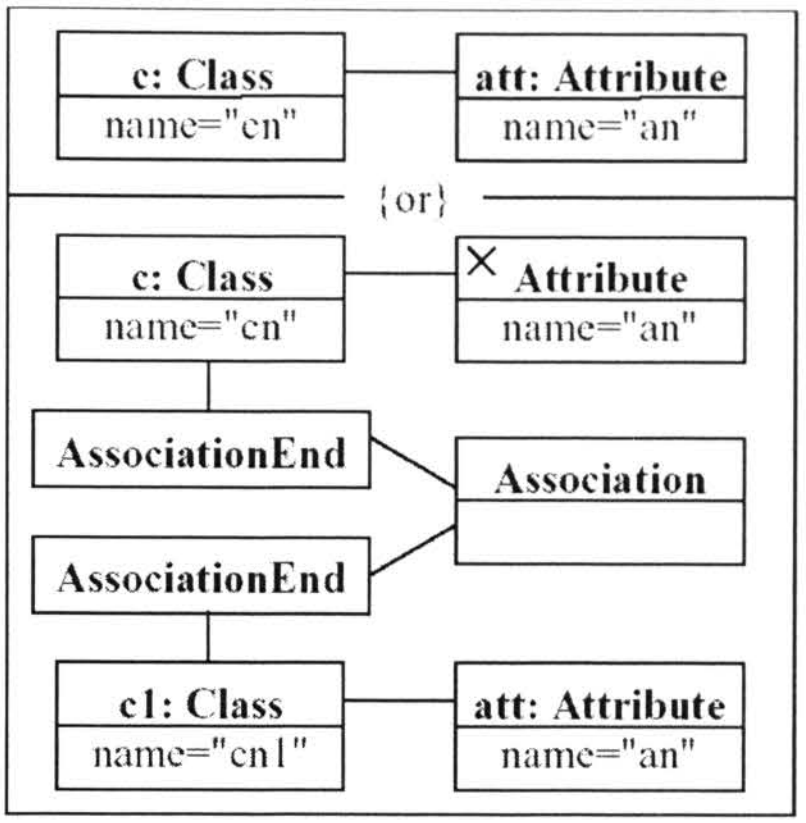
3.3.3. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models

Grafický jazyk pro popis dotazů nad modely, který by měl být základem jazyka pro popis transformací. Popisuje omezení pomocí lexikální podobnosti, struktury a chování – tj. neúplně zadaných vzorů. Součástí jazyka je i mapování dotazů do OCL.

Na obrázku 3-5 je dotaz pro výběr tříd z modelu tříd, odpovídá mu následující OCL:

```
someUmlModel.contents->select(c: Class |
      (c.name="cn" and c.feature->select(f |
      f.oclIsKindOf(Attribute))->includes(att | att.name="an"))
or (c.name="cn" and not c.feature->select(f |
      f.oclIsKindOf(Attribute))->includes(att | att.name="an")
      and c1.name="cn1" and c1.feature->select(f |
      f.oclIsKindOf(Attribute))->includes(att | att.name="an")
      and c.oppositeAssociationEnds->includes(ae |
```

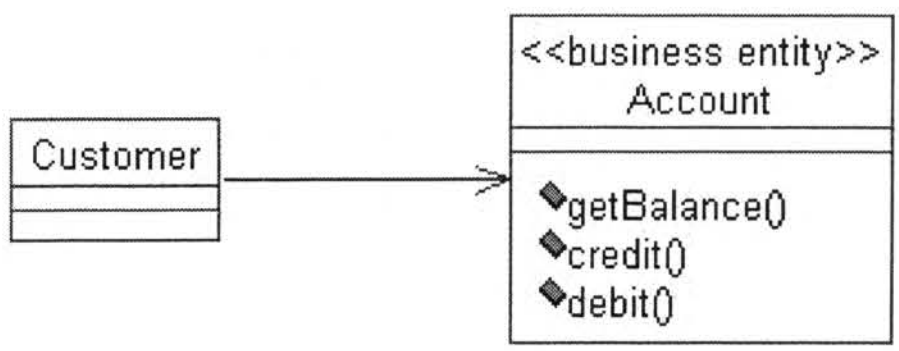
```
ae.participant = c1)))
```



Obrázek 3-5 Dotaz pro výběr tříd z modelu[7]

3.3.4. GREAT - Graphgesteuertes Entwurfs-Analyse und-Transformationssystem

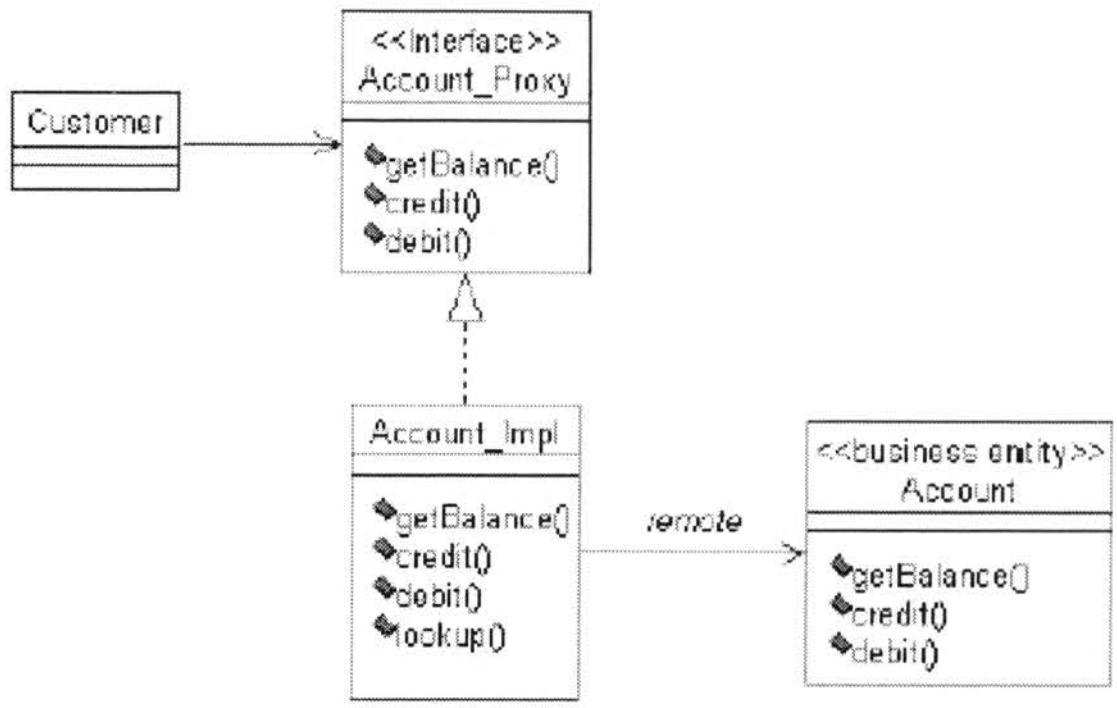
GREAT je jazyk a nástroj pro popis a provádění transformací založený na přepisovacích pravidlech pro grafy. Od UML metamodelu pro diagram tříd odvozuje strukturu grafu. Nástroj vytvořený autory jazyka GREAT umí transformovat modely uložené v XMI. Transformace jsou samostatnými moduly, které mohou používat API pro imperativní popis transformace nebo popsat transformaci deklarativně - přepisovacími pravidly. Jazyk GREAT předpokládá používání UML profilů pro zanesení dodatečných informací do PIM modelu. Následující obrázky a kód ukazují popis jednoduché transformace:



Obrázek 3-6 Transformovaný PIM model[25]

```
EARS remoteRelations(
      graph:Root,
      resultSet:EntitySet) {
```

```
        if c1 in graph.entities,
        c1 matches Clazz,
        a in c1.associations,
        c2 in a.target,
        c2 matches Clazz(stereotype => ST),
        ST == "business entity"
        then { *
        resultSet.add(new Entity[] { c1, a, c2 });
        *}
}
```



Obrázek 3-7 Výsledný PSM model[25]

3.4. UML profily

UML profily umožňují specializaci UML. Například UML profil pro CORBA definuje pojmy pro modelování CORBA rozhraní, UML profil pro Java definuje způsob modelování zdrojového kódu Java. Alternativou pro profily je definovaní nového metamodelu, tj. modelovacího jazyka. Toto je rozebráno více v kapitole 5.1.2 v kontextu existujících nástrojů.

Profil obsahuje množinou stereotypů, omezení a pojmenovaných hodnot.

Stereotyp je definován jménem a prvky z UML metamodelu, se kterými je propojen. Zachycuje určité aspekty domény daného profilu. Například stereotyp <<JavaClass>> je definován pro metatřídu Class z UML metamodelu. Při vytváření UML modelu můžeme tento stereotyp aplikovat na třídy.

Součástí definice stereotypu může být omezení. Omezení se zapisuje v jazyku OCL nad UML metamodelem. Popisuje restrikce pro prvky modelu, které používají daný stereotyp. Např. pro stereotyp <<JavaClass>> existuje omezující podmínka: „JavaClass může mít pouze jednoho předka".

Pojmenované hodnoty jsou doplňkové meta-atributy definované pro metatřídu z UML metamodelu a konkrétní stereotyp. Mají jméno a typ.

UML profil přináší do modelu dodatečné informace, které je možné následně použít při transformacích mezi modely a z modelu do kódu. Zatím jsou standardizovány UML profily pro *CORBA*, *EAI* (Enterprise Application Integration – standard pro výměnu metadat o rozhraních aplikací za účelem integrace), *EDOC* (Enterprise Distributed Object Computing – zahrnuje profily pro vývoj komponentových aplikací, profily pro Javu, EJB, vzory a další potřebné), *QoS and Fault Tolerance* (definuje obecný rámec pro popis aplikací se zaručenou kvalitou), *Schedulability, Performance, and Time* (definuje pojmy pro modelování aplikací v reálném čase), *Testování* a *Software Radio*. Podrobný popis profilů je k dispozici na webové stránce [30]. Dále existuje spousta nestandardizovaných profilů od jednotlivých dodavatelů vývojových nástrojů a výzkumných skupin.

3.5. UML Action Semantic

UML Action Semantic je standardizované rozšíření UML o procedurální jazyk. Popisuje metamodel pro akce a aktivity. Akce je základní jednotkou pro popis chování, přijímá množinu vstupů a konvertuje ji na množinu výstupů. Akce může modifikovat stav systému. UML 2.0 definuje více než 50 akcí – např. CallOperationAction, CreateObjectAction, ReadVariableAction. Akce jsou kvalifikovány do několika tříd – základní akce, akce vstupu a výstupu, akce pro manipulaci s objekty, akce pro akceptování události apod. Aktivita se skládá z více akcí. Akce a aktivity je možné použít pro vytváření přímo spustitelných modelů.

Bohužel chybí standard pro konkrétní syntaxi akcí a aktivit. Podle knihy [1] je nedostatkem Action Semantic definování akcí a aktivit na příliš nízké úrovni abstrakce. Přirovnává je k asembleru pro UML. Vhodným řešením by bylo vytvořit nový jazyk založený na UML Action Semantic operující na vyšší úrovni abstrakce. Někteří dodavatelé nástrojů proto vytvořili vlastní způsob zápisu akcí a aktivit, např. v Borland Developer Studio 2006 je jazyk pro popis akcí podobný OCL.

3.6. Executable UML

Executable UML (xUML) je v podstatě spustitelný profil pro jazyk UML. Je zaměřen prakticky a na trhu existují nástroje, které ho podporují, např. od Kenedy Carter[31](hlavní tvůrce xUML) nebo Project Technology[32].

Executable UML je standard „de facto". Používá vybranou podmnožinu konstruktů UML doplněnou o Action Specification Language (ASL). ASL umožňuje plně specifikovat chování platformově nezávislého systému. Nasazení xUML zhruba odpovídá postupu popsanému v kapitole 2.4.

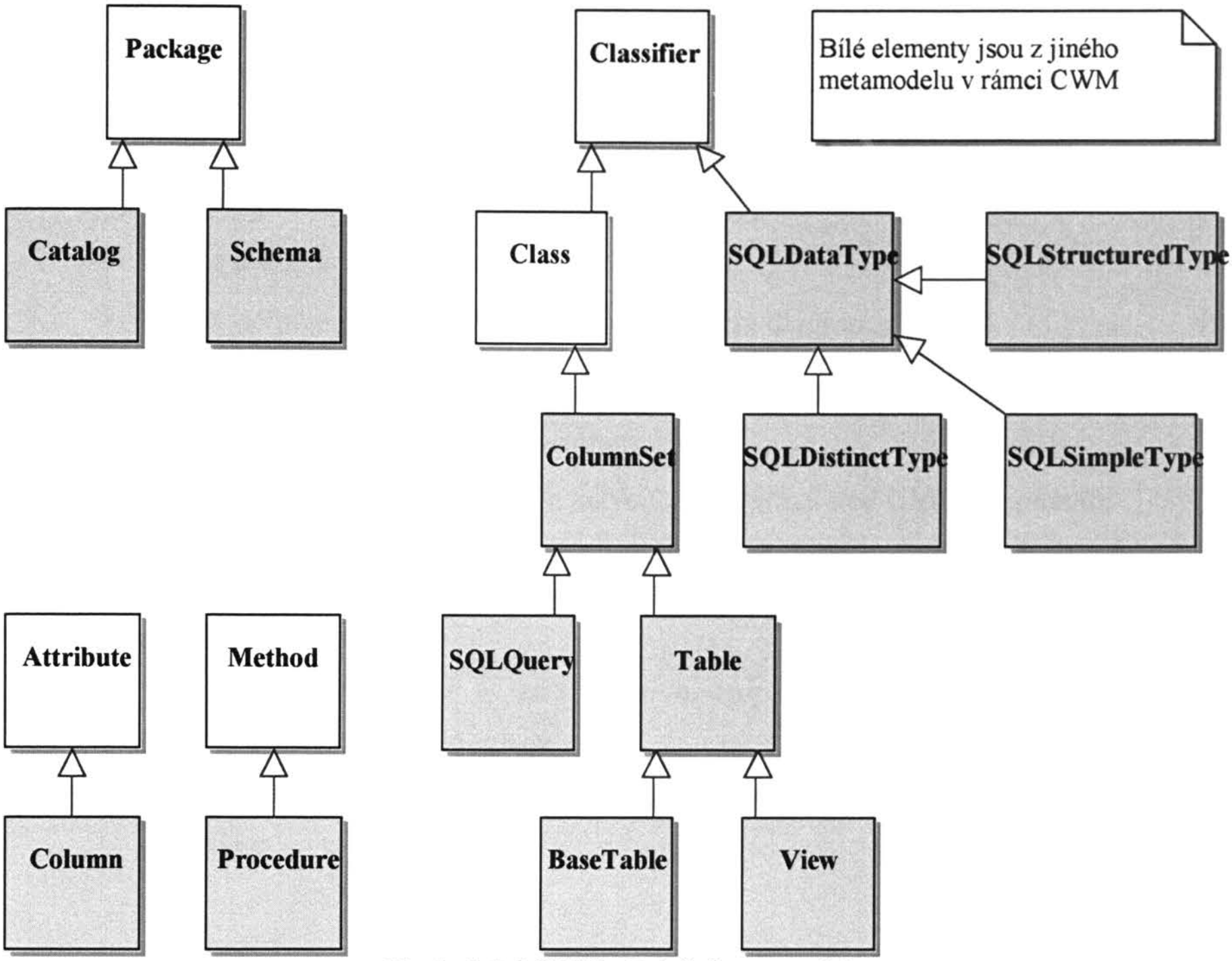
Executable UML definuje přesná kritéria pro oddělení modelů rozdělením modelů do různých domén. Jednotlivé modely se potom propojují pomocí „mostů", které se mi jeví pro rozsáhlejší modely velmi těžkopádné. Executable UML nutí vývojáře vytvářet přesné kompletní modely, protože modely musí být spustitelné.

Více informací o xUML je v knihách [3] a [4] nebo na webových stránkách výše uvedených společností.

3.7. CWM - Common Warehouse Metamodel

Jazyk CWM je stejně jako UML postaven nad MOF, proto ho můžeme používat při vývoji MDA aplikací. CWM není v dostupných modelovacích nástrojích podporován. Slouží pro modelování datových skladů. Poskytuje standardní rozhraní pro výměnu metadat mezi datovými sklady v heterogenních prostředích. Je podobný UML, protože vznikl jeho úpravou – vše nepotřebné bylo odebráno (např. modelování chování) a přidaly se pojmy potřebné, např. modelování relačních databází. V datových skladech jsou kombinovány informace z různých zdrojů, proto CWM umožňuje modelovat různé technologie:

- relační databáze
- záznamy a struktury
- OLAP - On Line Analytical Processing
- XML - Extensible Markup Language
- transformace dat
- vizualizace informací
- dolování dat
- vícerozměrné databáze
- ...



Obrázek 3-8 CWM - Relační metamodel

3.8. OCL – Object Contraints Language

Pomocí struktury UML diagramů většinou nelze vyspecifikovat všechny požadavky. Pro objekty v modelech je nutné zapsat omezení, která jsou často získána v přirozeném jazyku. Použití přirozeného jazyka vede k nejasnostem. Taková omezení nelze využít při automatických transformacích. Pro jednoznačný zápis existuje řada formálních jazyků. Tyto jazyky jsou však těžko pochopitelné pro osoby bez kvalitního matematického vzdělání. Proto vznikl jazyk OCL, který tímto nedostatkem netrpí.

OCL je čistě specifikační jazyk. Výraz v něm nemá vedlejší účinky - nemůže změnit žádný objekt v modelu. Vyhodnocení výrazu v jazyku OCL pouze vrací hodnotu. OCL není programovací jazyk. Je typovaným jazykem - každý výraz v něm má svůj typ. Vyhodnocení výrazu v jazyku OCL je okamžité, stav objektů v modelu se během vyhodnocení nemění. OCL výrazy lze snadno přeložit do programovacích jazyků. OCL lze použít pro modely v UML i v MOF (přímo v metamodelu UML je použito několik stovek omezení zapsaných v OCL). Z jazyka OCL je odvozen jazyk pro popis transformací QVT.

Použití OCL:

- Dotazovací jazyk
- Specifikace omezení a typů v diagramu tříd (v UML)
- Specifikace invariantů pro stereotypy (v MOF)
- Popis pre- a post- podmínek operací a metod
- Popis podmínek pro změnu stavu ve stavovém diagramu
- Specifikace příjemců zpráv a signálů
- Specifikace výchozí hodnoty atributů
- Specifikace odvození atributů a jakýchkoliv výrazů nad UML diagramem

Příklad omezení v OCL:

```
context Person inv:
let income : Integer = self.job.salary->sum() in
      if isUnemployed then
            income < 100
      else
            income >= 100
      endif
```

4. Kombinace modelem řízené architektury s architekturou orientovanou na služby

4.1. Architektura orientovaná na služby (Service Oriented Architecture - SOA)

SOA je IT framework kombinující jednodušší byznys funkce a procesy (tzv. služby informatiky) do byznys aplikací a složitějších procesů. SOA systémy spadají do třídy n-vrstvých aplikací. Služba informatiky je v práci [33] definována jako „ucelený definovaný výstup procesů informatiky poskytovaný jejím interním i externím zákazníkům. Služby jsou výsledkem kombinace konkrétních instancí procesů a zdrojů, které probíhají v oblasti informatiky". Jeden z mála standardů vymezujících Architekturu orientovanou na služby je SOA Reference Model[47] od organizace OASIS (Organization for the Advancement of Structured Information Standards).

Hlavním přínosem SOA je zvětšení flexibility IT. To následně umožňuje větší flexibilitu IT při podpoře byznysu. Flexibilita spočívá v možnosti reagovat na nové podmínky trhu vytvořením nového propojení existujících služeb.

Při vývoji SOA systémů se dle [13] vykonávají zejména následující činnosti:

- **Vyhledávání služeb** – objevování nových požadavků byznys prostředí a vyhledávání služeb v existujících aplikacích, rozvíjení funkcí existujících služeb.
- **Vytváření nových služeb** – obalování existujících funkcí do služeb, vytváření nových služeb, úprava existujících služeb a propojování hotových služeb.
- **Nasazení služeb** – zavedení platformy pro běh služeb, nasazení konkrétních služeb.
- **Používání a správa služeb** – udržování platformy pro běh služeb.

V dalším textu se budu zabývat možnostmi využití MDA při vytváření a správě služeb.

4.2. Modelování a metodiky při vývoji SOA aplikací

Jak bylo uvedeno v kapitole 2.4.6., současné nástroje s podporou MDA implementují metodiky pro OOAD. Část jich navíc umožňuje odděleně od „objektů" modelovat byznys procesy. Pro provádění transformací umožňují některé nástroje definovat architekturu cílové platformy (typicky pomocí profilů). To podle článku [40] i mých zkušeností pro modelování

SOA systémů nestačí. V OOAD se nepracuje s pojmy pro orchestraci služeb do byznys procesu a registry služeb ani s návrhovými vzory pro výměnu dokumentů. Základem OOAD jsou případy užití, v SOA jsou to byznys procesy. Metodiky pro OOAD vyžadují těsně spřažené objekty s dobře definovanými rozhraními (tzv. fine-grained API), objekty spolu komunikují pomocí volání metod. V SOA jsou služby spřaženy volně, základním komunikačním principem SOA je výměna dokumentů. Pevné vazby v SOA existují pouze lokálně. V článku [43] jsou popsány rozdíly v chápání dat uvnitř a vně služby, které jsou způsobené přechodem od distribuovaných objektů k SOA. V SOA je nutné při práci s daty vně služby používat některé nové postupy - význam dat je relativní, je nutné uchovávat různé verze dat atd.

Při tvorbě SOA systémů je podle článku [40] nutné používat zároveň metodiky pro *modelování byznys procesů*, *architekturu řešení, firemní architekturu* a *objektově orientovanou analýzu a design.* Modelovací jazyky a nástroje pokrývající jednotně všechny tyto oblasti zatím neexistují.V článku [42] je popsána první představa SOAD (Service-oriented modeling and architecture). SOAD je pokusem o sjednocení používaných metodik pomocí společného metamodelu, avšak s nejasným výsledkem a bez odkazů na podporu v nástrojích.

Jedním z prvních kroků na poli propojení MDA s výše uvedenými metodickými oblastmi je standardizace mapování mezi MDA a TOGAF (The Open Group Architecture Framework)[41] organizací The Open Group. TOGAF je framework pro popis architektury informačních systémů. Toto mapování popisuje korespondenci mezi architekturou informačních systémů a MDA. Cílem mapování je usnadnění společné evoluce firemní architektury a aplikací založených na modelech.

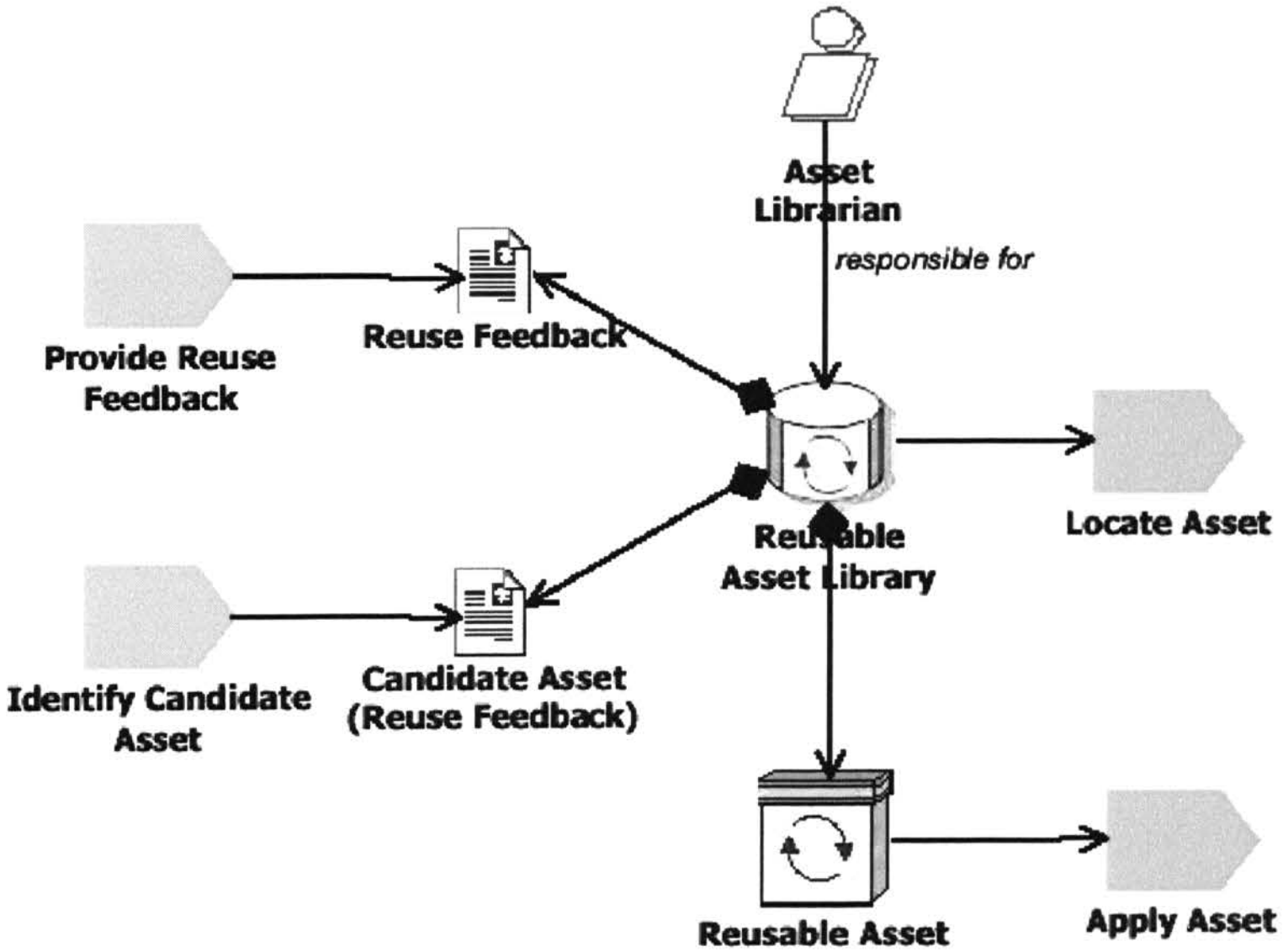
OMG si je také vědoma nedostatků v současných standardech. V kapitole 5.2 dokumentu MDA Guide[20] stojí, že standardy pro integraci aplikací, tj. i pro SOA, zatím chybí a bude nutné je vytvořit. Za tímto účelem vznikla nová pracovní skupina (SOA SIG – Special Interest Group), která má za úkol mimo jiné i zavedení MDA principů do SOA. V jejím plánu je např. vypracování SOA metamodelu propojujícího principy z výše uvedených oblastí.

Podrobná diskuse k nutnosti použití SOA místo distribuovaných objektů pro rozsáhlé systémy je nad rámec této práce. Článek [45] popisuje problémy vznikající i při přechodu od monolitických k distribuovaným systémům – distribuovanost objektů nelze transparentně skrýt, k tvorbě distribuovaných systémů nelze přistupovat stejně jako k tvorbě systémů lokálních. Výhody služeb v porovnání s distribuovanými objekty pak rozebírá článek [46].

4.3. Asset-based Development – ABD

ABD je metodika firmy IBM pro řízení procesu vývoje softwarových systémů. Je založena na znovupoužitelných softwarových artefaktech. Je doplňkem komplexnější metodiky Rational Unified Process (RUP). Artefakty se rozumí komponenty, vzory, služby, frameworky, šablony apod. ABD využívá standardy UML, MDA a RAS (Reusable Asset Specification). Pomocí zásuvného modulu ji podporují nástroje firmy IBM pro RUP. ABD definuje kvalitativní vlastnosti artefaktů důležité pro jejich znovupoužitelnost, kterými jsou: snadnost použití, vysoká soudržnost, nízká spřaženost a srozumitelnost kontextu použití artefaktu.

ABD lze použít pro vývoj SOA systémů. Činnosti jí definované pokrývají všechny oblasti uvedené v kapitole 4.1. Jsou ze služeb rozšířeny i na další artefakty (konkrétně vyhledávání, vytváření, správa a používání). Hotové artefakty jsou uloženy v knihovně spolu se záznamy o používání artefaktu.

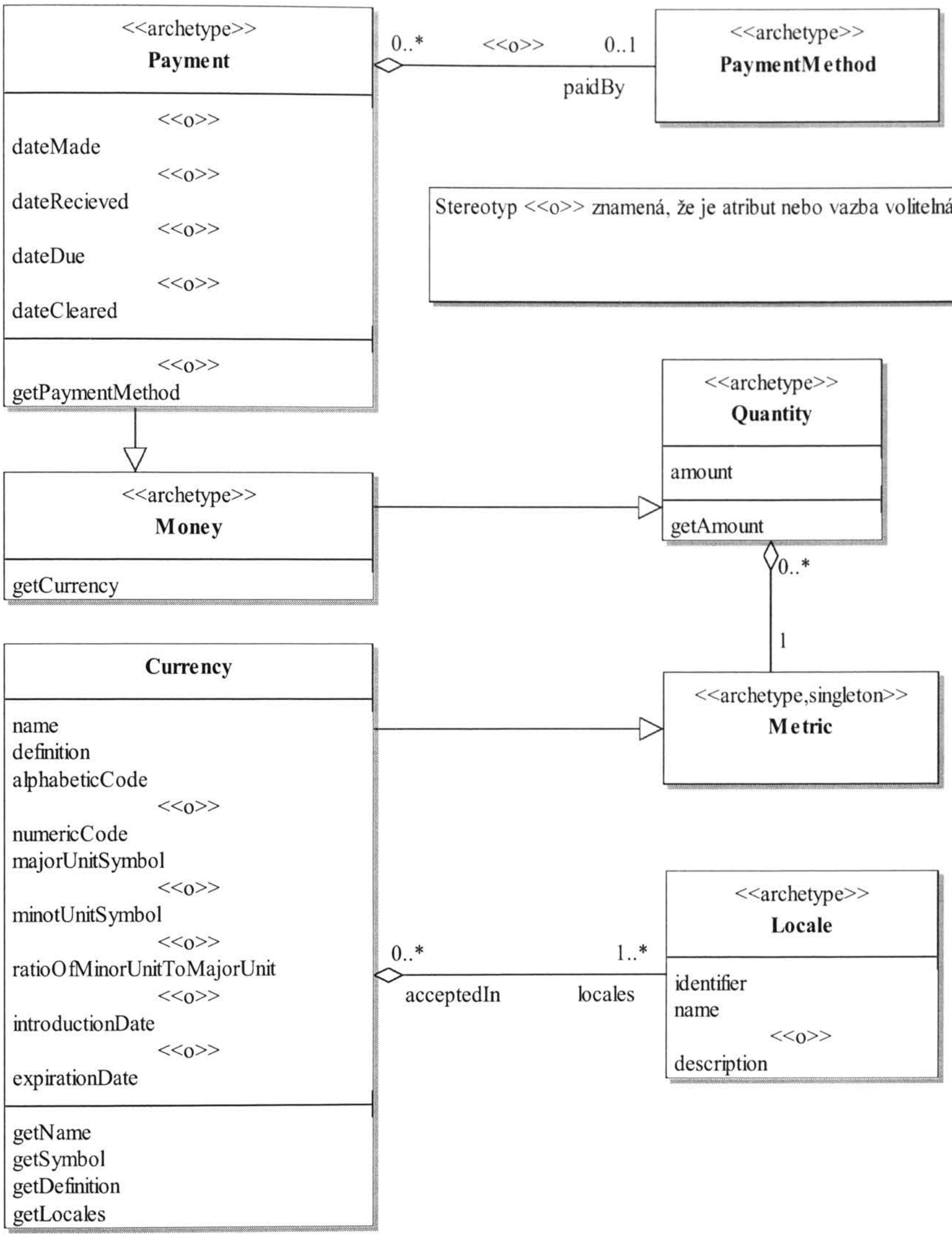


Obrázek 4-1 Správa a používání artefaktů[61]

ABD používá UML při modelování byznysu a architektury. Implementuje systémy s použitím artefaktů vyvinutých pomocí MDA technik. Pro popis artefaktů používá RAS. Pro vývoj a používání artefaktů navrhuje nasadit zejména IBM Rational Architect (následník IBM Rational XDE, viz kapitola 5.4). Tento nástroj poskytuje výchozí sadu artefaktů v podobě návrhových vzorů z knihy [34] a také vzory dle specifikace J2EE (Java 2 Platform, Enteprise Edition), tzv. BluePrints[48]. Další vzory je možné najít na stránkách firmy IBM.

ABD metodika však nedefinuje přímo způsob využití MDA pro vývoj softwarových artefaktů. Zabývá se spíše správou artefaktů a jejich využíváním.

Myšlenky znovupoužitelných softwarových artefaktů v ABD jsou velmi podobné metodě popisované v knize Enterprise Patterns and MDA: Building Better Software with Archetype Patterns[2]. Podle této knihy mají být základním stavebním prvkem při vývoji softwarových systémů archetypální návrhové vzory. Součástí knihy je i katalog návrhových vzorů pro oblast obchodu. Aplikace mají být vytvářeny na základě těchto vzorů – jejich přizpůsobením pro konkrétní použití (tj. většinou zjednodušením). Vzory jsou tedy znovupoužitelnými artefakty ve smyslu metody ABD. Praktické nasazení této metody je v knize ukázáno na nástroji ArcStyler. Myslím, že součástí těchto komplexních vzorů by mohl být i popis mapování na konkrétní platformy. Tím by odpadla nutnost modely před transformací z PIM do PSM „značkovat". Bylo by žádoucí doplnit nástroje o vzory z takového katalogu. Zatím v nástrojích převažují vzory z oblasti vytváření softwarových systémů.



Obrázek 4-2 Ukázka části archetypu Money

4.4. Příklady použití MDA v SOA

Propojení MDA a SOA bude rozebráno ve dvou rovinách:

- **Použití MDA pro vývoj jednotlivých služeb** – tato oblast spadá pod vývoj „obyčejných" MDA aplikací, kde výsledná aplikace navíc poskytuje rozhraní služby. Z této oblasti jsou zajímavé zejména přínosy „metainformací" o službách pro jejich propojování do SOA aplikací – ***vývoj zdola nahoru.***
- **Použití MDA pro vývoj SOA systémů** – zapojení modelů do vývoje SOA systémů a jeho přínosy pro vývoj aplikací ***shora dolů.***

4.4.1. Existující korespondence SOA a MDA

Jednotlivé úrovně modelů v MDA podle [37] odpovídají úrovním abstrakce v SOA následujícím způsobem:

- CIM model – popis vysokoúrovňových byznys služeb, modelování byznys procesů.
- PIM model – popis aplikací, které poskytují nebo používají služby.
- PSM model – popis služeb na konkrétní platformě – např. webových služeb.

Jelikož se v současných nástrojích s podporou MDA pracuje převážně s PIM a PSM modely, je jejich zapojení do vývoje SOA aplikací na nejvyšší úrovni (propojování služeb) problematické. To je ve shodě s metodickými nedostatky současných nástrojů popsanými v kapitole 4.2. Dále se proto zaměřím na popis možného využití, přínosů i limitů těchto nástrojů pro použití v SOA.

4.4.2. Problém iterativního vývoje v SOA

Vývoj (MDA) aplikace je většinou kombinací iterativního a inkrementálního vývoje – mění se stávající funkce a přidávají se nové. Během iterací může docházet ke změnám datových struktur, objektů a rozhraní uvnitř aplikace. Pro vývoj samotných služeb není takový iterativní vývoj problémem. Změny uvnitř služby se ale nesmí promítnout do sémantiky dokumentů zpracovávaných službou, protože by to ovlivnilo konzumenty služby.

Bude tedy nutné zachovat sémantiku zpracovávaných dokumentů během iterací. Vývoj rozhraní pro zpracování vstupů a výstupů by měl být „pouze" inkrementální. Sémantika zpracovávaných dokumentů bude hranicí, za kterou se iterativní vývoj (tj. změna vnitřních struktur služby) nemá projevit. Nemožnost dobře definovat sémantiku dokumentů znamená, že nelze dobře definovat ani byznys proces používající tyto dokumenty. Více o problémech spojených s propojováním služeb lze najít např. v [36].

Přínosem MDA je možnost definovat rozhraní služby ve chvíli, kdy přijde z byznysu požadavek na extrakci funkcí aplikace do služby (typicky ho budeme popisovat pomocí stereotypů v modelu, jak je ukázáno v kapitole 4.3.3). Pomocí transformací pak snadno vytvoříme novou verzi aplikace, která bude obsahovat i kód potřebný pro nasazení služby. Pozdní definice rozhraní služby snižuje riziko nutnosti jeho následných úprav.

Zatím není příliš dobře vyřešen způsob modelování změny struktury dokumentů při zachování sémantiky – tj. abstraktní popis transformace různých vstupních dokumentů do struktur modelované aplikace a transformace z aplikace do výstupních dokumentů (např. pro XML dokumenty se k tomu účelu v ESB (Enterprise Service Bus) používají XSLT transformace). Teoreticky je možné změnu struktury dat v dokumentu modelovat pomocí UML profilu pro EDOC nebo pomocí CWM transformací. Existuje však jediný nástroj implementující EDOC profil v dostatečném rozsahu (Component-X Studio, viz kapitola 4.5.1). Zatím je nutné popis takové transformace řešit pomocí komponenty pro zpracování dokumentů, která v modelu vystupuje jako černá skříňka. Velmi přínosné by bylo modelovat chování integračních frameworků založených na ESB, tj. modelovat např. brokering zpráv dle obsahu nebo úplnou změnu formátu zprávy (nejen její struktury). Otázkou zůstává možnost praktické realizace. ESB řešení nejsou založena na standardech ani nejsou dostatečně otevřená. Toto vše jsou příklady limitů nástrojů pro OOAD zmiňovaných v kapitole 4.2.

4.4.3. Vývoj SOA aplikací směrem od byznysu

Jelikož je SOA architektura orientována na byznys, je smysluplné vytvářet SOA aplikace směrem od byznys modelů. Pro tento účel je možné použít Business process model (BPM). V tomto modelu lze definovat služby a způsob jejich orchestrace. Nepřímo se tak popisuje rozhraní jednotlivých služeb. Modelovací nástroje umožňují exportovat BPM do BPEL a WSDL, ze kterého lze automaticky vygenerovat kód pro rozhraní služeb a třídy zasílaných zpráv. Pohodlnější a výhodnější pro MDA je export těchto artefaktů do UML. Artefakty potom mohou být zahrnuty do modelu aplikací odpovídajících jednotlivým službám. V modelu mohou být doplněna napojení rozhraní služby na aplikaci.

4.4.4. Přínosy služeb založených na MDA, jejich integrace

Při vývoji SOA aplikací lze postupovat i zdola nahoru. Aplikace vytvořené pomocí MDA mají dostatek metainformací pro dodatečnou automatickou integraci v SOA. Metain-

formace je možné uložit např. ve formátu RAS. Podle [14] je hlavním přínosem MDA to, že teoreticky umožňuje velmi těsné propojení služeb:

- Integrace služeb založených na stejné platformě
- Integrace služeb z různých technologických platforem.
- Vytváření služeb s přidanou hodnotou (příkladem jsou informace o zpoždění letů transformované do textu SMS)
- Zapojení adresářových služeb (vysoce distribuovaných databází) - zavedení služeb pracujících v určitém kontextu nebo profilu (příkladem jsou informace o počasí doručené dle lokality požadavku)

Právě uvedeným možnostem odpovídají na technologické úrovní tyto stupně integrace:

- Standardní B2B a EAI integrace
- CISB (Common Integration Service Bus) – správa sezení, různé modely propojení, transakcí a zasílání zpráv
- Ukládání stavu sezení, transakcí a zpráv, včetně transakčního zasílání zpráv
- Společný model služeb a jejich kvality
- End-to-end transakce integrovaných služeb

Základním problémem realizace těchto „vizí" jsou chybějící standardy a nedostatečné využívání (dodržování) standardů výrobci nástrojů. Propojování dvou služeb založených na MDA a vyvinutých pomocí nástrojů různých dodavatelů přináší jisté problémy. Protože integrace musí jít až na úroveň PSM modelů, jsou potřeba znalosti o transformacích z PIM do PSM pro obě služby. **Tyto transformace zatím nejsou OMG standardizovány**. Pokud by byly, museli by takové standardy dodržovat i výrobci nástrojů. Vzhledem ke stavu dodržování standardů např. pro webové služby (rozdílná serializace Microsoft a Apache Axis - problémy s datumy, null nebo choice apod.) je toto očekávání nereálné.

Proto nejsou zatím existující nástroje pro integraci hotových služeb (např. IBM Websphere Business Integration Modeler) orientovány na byznys, nýbrž na vývojáře a pracují maximálně na úrovni abstrakce PSM modelů.

4.4.5. UML pro popis orchestrace služeb do byznys procesu

I když lze modely byznys procesů snadno zapojit do vývoje MDA aplikací, není to dostačující řešení. Tento model je vhodný pro CIM a částečně PIM modely. Pro MDA je potřeba zachytit větší detaily, zejména závislosti na platformě – např. integraci na úrovni transakč-

ního zpracování. Toto umožňují až dostatečně bohaté a rozšiřitelné jazyky, např. UML. Ukážu jednu z možností využití UML při vývoji SOA aplikací.

Pro orchestraci webových služeb se používá jazyk BPEL. Nabízejícím se postupem je vytvořit model byznys procesu se zapojením služeb a z něj vygenerovat BPEL. Pro pohodlné vytváření takového modelu je možným řešením použít vhodný UML profil.

V článku Model Driven Architecture in a Web services world [35] je takový profil popsán. Dále je tam ukázáno (v nástrojích od IBM popsaných níže), jak pomocí takového profilu vytvořit model aktivit, který bude možné exportovat do BPEL4WS. Řešení z článku vzniklo v době před UML 2.0, proto je v některých ohledech „neohrabané". UML 2.0 obohatilo diagramy aktivit o možnosti zachytit tok objektů podobně jako v Petriho sítích. Mapování některých prvků z UML diagramu aktivit do BPEL je potom následující:

Prvek UML modelu	**Pojem v BPEL4WS**	**Element v BPEL4WS**
Model aktivit	Definice procesu	`process`
Vstupní parametr modelu	Přijmutí vstupu	`recieve`
Výstupní parametr modelu	Operace aktivita vydání výstupu	`reply`
Rozhodovací uzel následující po parametru	Řízení toku	`recieve - source`
Rozhodovací uzel následující po aktivitě	Řízení toku	`invoke - source`
Vstupní a výstupní parametry modelu a aktivit	Proměnné procesu	`variable`
Operace objektu se stereotypem <<BusinessEntity>> (na objektu z nějakého modelu tříd)	Volání metody služby	`invoke`
Uzel „DataStore"	Vydání odpovědi – změna parametru procesu	`assign`
Koncový stav	Ukončení procesu	`terminate`

Tabulka 4-1 Část mapování BPEL4WS a UML modelu aktivit

UML 2.0 umožňuje snadné propojení diagramu aktivit s diagramem tříd. Do diagramu aktivit lze umisťovat aktivity, které odpovídají metodám objektů z diagramu tříd. Při transformaci modelu tříd potom stačí pro takto použité metody vygenerovat kód potřebný pro nasazení služby. Je výhodné použít pro takové třídy (rozhraní) nějaký stereotyp – ve výše uvedeném příkladu je to << BusinessEntity >>. Pak je možné metody tohoto objektu volat v ně-

jakém novém byznys procesu bez nutnosti znovu generovat celou aplikaci, které je objekt součástí. To umožní pracovat s MDA aplikacemi způsobem popsaným v předchozí kapitole. Takové použití UML však vede k orientaci na propojování rozhraní místo výměny dokumentů, což není příliš vhodné pro další rozvoj SOA systému.

4.4.6. Sklad modelů a informací o službách

Firma innoQ v [44] navrhuje postupy, které umožní využít potenciál služeb vytvořených pomocí MDA při vývoji SOA systémů. Počítá s vytvořením modelu celého SOA systému. Modely jednotlivých služeb mají být uloženy ve skladu modelů. Zároveň navrhuje uložit do skladu i informace o využívání služeb (dynamické vazby, způsoby použití, ...). Dalším zdrojem informací pro sklad má být reverzní inženýrství existujících aplikací. Informace ze skladu modelů umožní automaticky generovat propojení služeb. Podle blogu [47] i vedoucího projektu z firmy innoQ má toto řešení podstatné nedostatky: zatím neexistuje společný metamodel pro všechny části nutné v SOA, cena za něj může být příliš vysoká – vyšší než náklady na používání heterogenních modelů a nástrojů. Není také jasné, jak těsně spřažený je vygenerovaný systém.

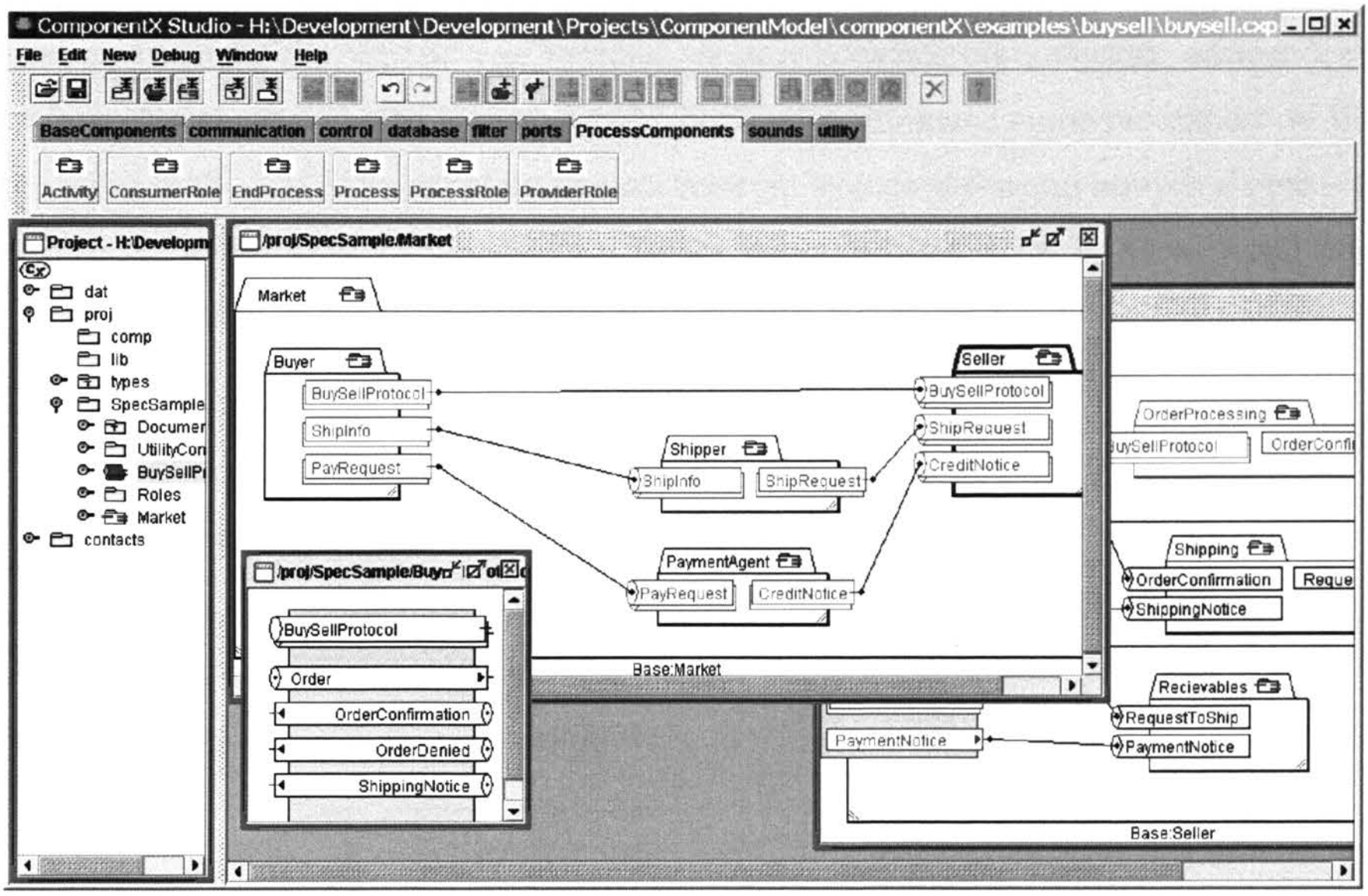
Tento postup firma innoQ zčásti implementovala v jedné z největších švýcarských bank. Podařilo se jí generovat popis rozhraní služeb, Cobolské programy a webové rozhraní pro přístup k datům, dokumentaci ke službám v HTML a infrastrukturu pro J2EE aplikace. Docílila 100% synchronizace modelů a implementace. Tento postup je podobný těm z metodiky ABD popisované v kapitole 4.3, kde se také používá společný sklad informací o artefaktech.

4.5. Vývojové nástroje a řešení kombinující MDA a SOA

4.5.1. Component-X™ Studio

Component-X™ Studio je vizuální vývojové prostředí pro vytváření aplikací založených na komponentách dle standardu EDOC (UML profil standardizovaný OMG) potažmo MDA. Podporuje komponenty založené na Java (přístup k databázím, síťová komunikace, aplikační logika), XML dokumenty a webové služby. Tyto komponenty je možné vizuálně vytvářet a „skládat" do složitějších aplikací. Komponenty odpovídají službám v SOA a komunikují spolu pomocí XML dokumentů.

V základní sadě jsou komponenty pro přístup k middlewaru (SOAP, JSM), pro transformaci dokumentů v XML, mapování mezi XML a SQL, aplikační logiku (větvení, iterace, výběry), filtry apod.



Obrázek 4-3 Vývoj aplikace v Component-X Studio[62]

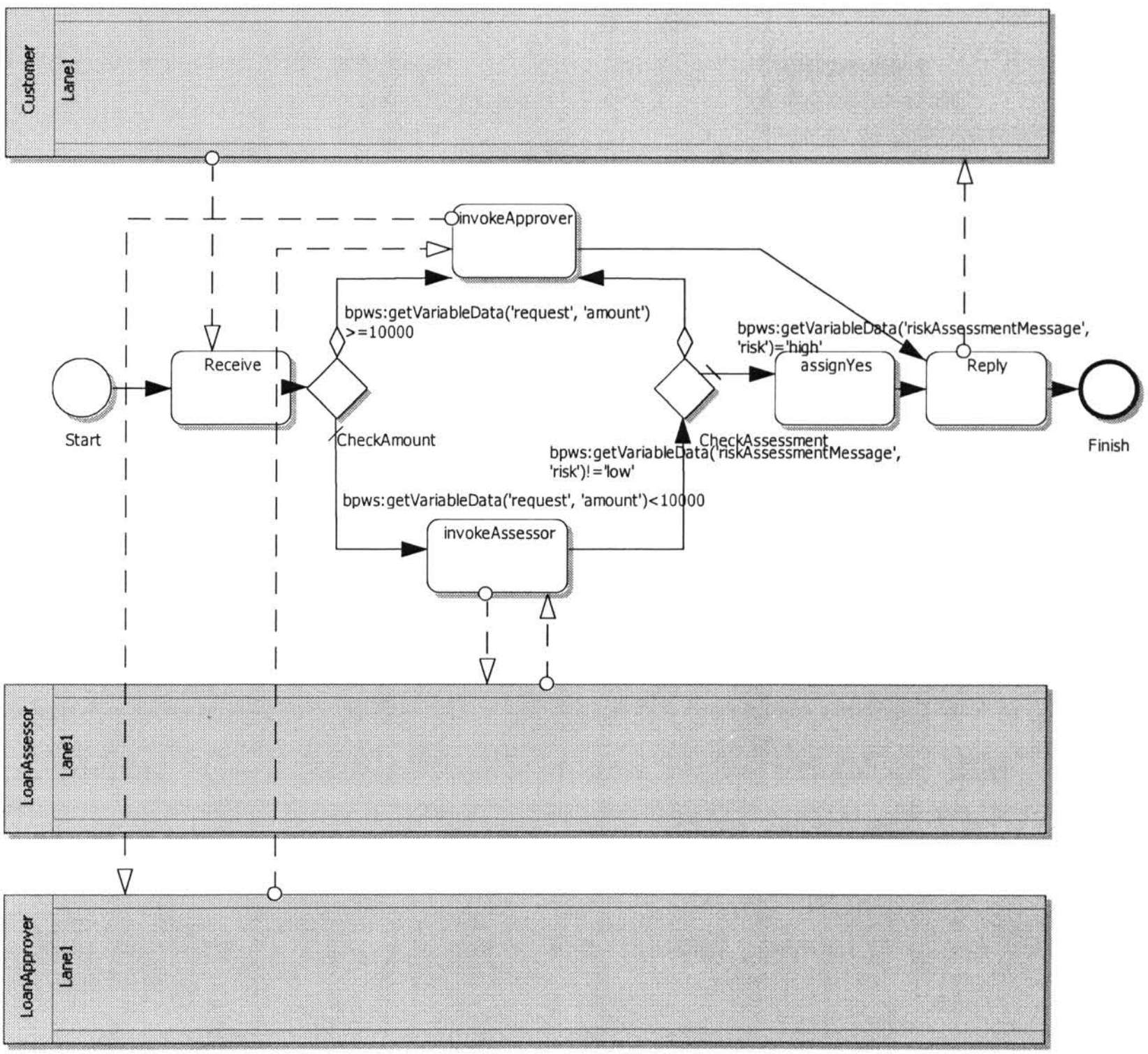
Nové komponenty je možné vytvářet hierarchickým skládáním existujících komponent, importem webových služeb z ebXML BPSS (Electronic Business using eXtensible Markup Language Business Process Specification Schema, standard United Nations Centre for Trade Facilitation and Electronic Business - UN/CEFACT a organizace OASIS), SOAP WSDL nebo UDDI, naprogramovat je v Java nebo koupit je hotové.

Komponenty spolu komunikují pomocí protokolů. Protokol určuje typ dokumentů, které si komunikující strany vyměňují. Komponenty je možné snadno testovat pomocí automaticky vygenerovaného webového uživatelského rozhraní (pro běh komponent se používá server Apache Tomcat). Nástroj k vytvořeným komponentám generuje hierarchicky strukturovanou dokumentaci.

Firma Data Access Technologies, dodavatel tohoto nástroje, připravuje za podpory americké vlády novou verzi, která bude založena na platformě Eclipse a aplikačním serveru JBoss. Tento produkt by měl implementovat řešení dle standardu Open Source eGov Reference Architecture (OSERA) od US General Services Administration's (GSA).

4.5.2. Borland Together 2006 Architect

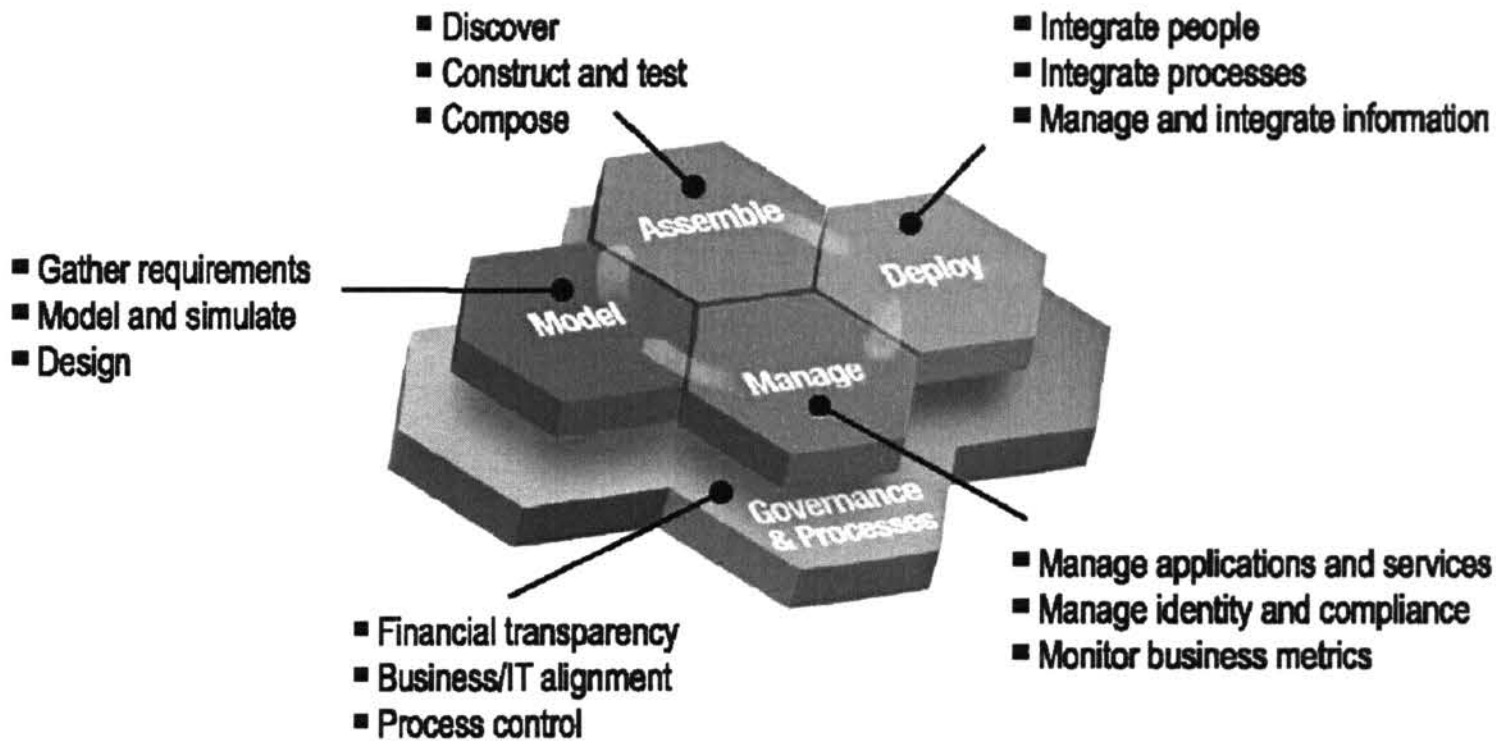
Borland Together 2006 Architect zahrnuje podporu pro vytváření modelů byznys procesů. Použitý modelovací „profil" umožňuje vytvářet modely dostatečně formální pro následný export do BPEL4WS a WSDL. Lze vytvářet všechny potřebné typy objektů, událostí i elementů pro řízení toku. U těch je možné zadat doplňující informace nutné pro export do WSDL a BPEL4WS. Z WSDL se potom pomocí nástrojů vygeneruje kostra nových služeb – jejich rozhraní a třídy, které odpovídají zasílaným zprávám (např. IBM Rational Application Developer, Apache Axis 2 nebo gSOAP).



Obrázek 4-4 BPM vytvořený v Borland Together 2006 Architect, včetně informací pro export do BPEL4WS

4.5.3. IBM

Nástroje firmy IBM pokrývají celý vývojový cyklus SOA aplikací – modelování, vytváření, nasazování a správu. IBM také dodává aplikační servery pro běh SOA aplikací. Sadu nástrojů pro vývoj SOA aplikací označuje „IBM SOA Foundation". Tyto nástroje jsou orientovány směrem k byznysu tzv. Business-Driven Development for SOA. Tyto nástroje umožňují modelovat byznys procesy, navrhovat a vytvářet služby, spojovat služby do aplikací, nasazovat vytvořené aplikace a spravovat je. Při zkoumání možností těchto nástrojů jsem se zaměřil na jejich přínos pro vývojáře, kterým jsou určeny nástroje z balíku IBM Rational Software Development Platform. Tyto nástroje jsou postaveny nad otevřenou a modulární platformou Eclipse.



Obrázek 4-5 IBM SOA Foundation[61]

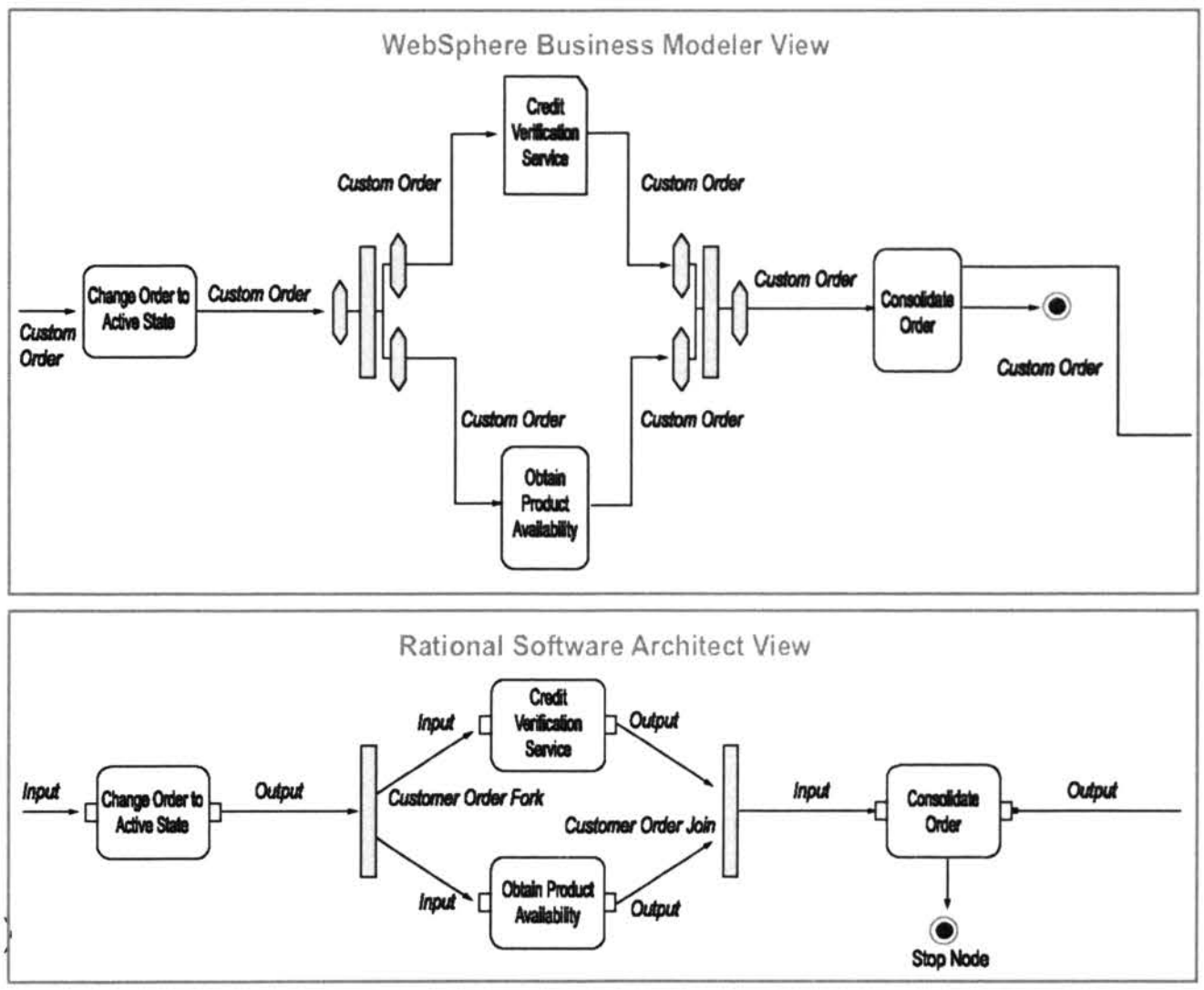
Nástroje jsou rozděleny podle rolí v projektovém týmu. Na obrázku 4-7 je uveden jejich přehled. Podpora SOA v těchto nástrojích je zaměřena na technologie IBM – Websphere (aplikační server), DB2 (databázový server), Tivoli (správa heterogenních informačních systémů) a Lotus (nástroje pro týmovou práci). Podporují standardy XML, WSDL, SOAP, BPEL. Jsou zaměřeny na vývoj aplikací v Java, zahrnují podporu pro Java Server Faces a J2EE.

- **IBM Websphere Business Modeler**

IBM Websphere Business Modeler je nástroj určený pro analytiky k modelování byznys procesů. Umí exportovat vytvořené modely do UML a BPEL. Pro tuto oblast je funkčně srovnatelný s Borland Together Architect 2006.

- **IBM Rational Software Architect**

IBM Rational Software Architect umí importovat projekty vytvořené v IBM Websphere Business Modeler. Byznys model je kompletně mapován do UML s použitím profilů pro byznys modelování. Např. procesy jsou převáděny způsobem podobným tomu, který jsem popsal v kapitole 4.4.5. Jedním z mála nedostatků je, že pro rozhodovací uzly nejsou převedeny podmínky pro výstupní hrany.



Obrázek 4-6 Import modelu byznys procesu z WBM do RSA[61]

Tento nástroj podporuje vývoj aplikací pomocí MDA. Softwarový architekt v něm transformuje byznys model (CIM) do modelu softwaru (PIM a PSM). V SOA by měl nástroj sloužit zejména pro vývoj jednotlivých služeb. Při vytváření modelů v něm lze používat hotové artefakty popsané pomocí RAS. Možnosti samotného vývoje MDA aplikací v tomto nástroji jsou podrobně popsány v kapitole 5.4.

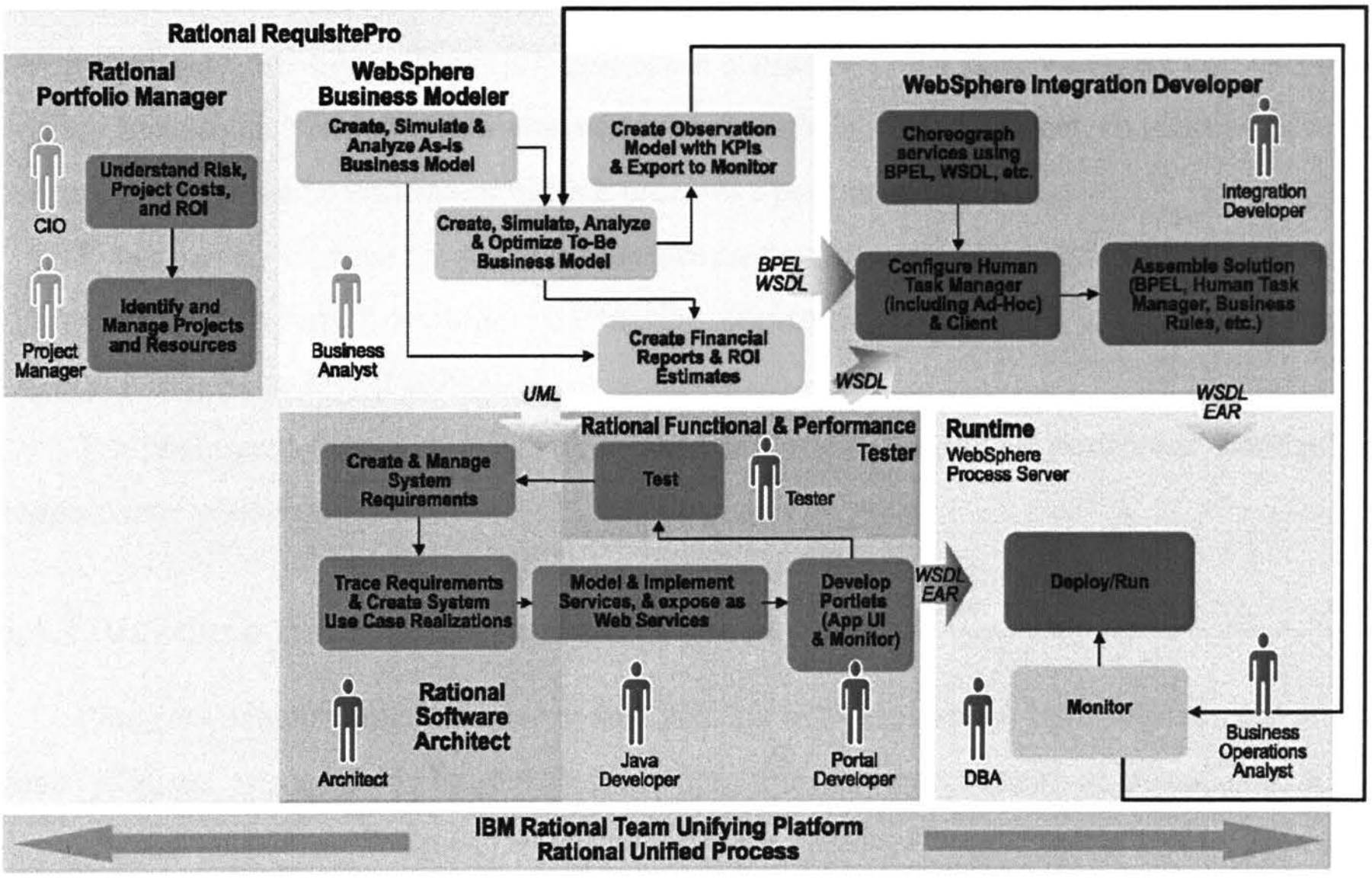
- **IBM Rational Application Developer**

IBM Rational Application Developer je integrované vývojové prostředí pro vývoj a používání služeb. Automaticky generuje z UML modelů WSDL, kostru EJB apod. Obsahuje průvodce pro vytvoření webové služby při vývoji zdola nahoru – pro třídu v Java vygeneruje WSDL rozhraní, konfiguraci pro nasazení v SOAP a webovou stránku pro testování třídy.

- **IBM Websphere Integration Developer**

IBM Websphere Integration Developer je nástroj pro vytváření SOA aplikací. Umí importovat modely z IBM Websphere Business Modeler. Vývojář pomocí tohoto nástroje zjemňuje importovaný model. Služby lze importovat i z UDDI registru. Podporuje integraci služeb na různých platformách (od různých dodavatelů – Microsoft, IBM, SAP, ...) Nástroj zahrnuje podporu pro testování a nasazení vytvořených aplikací.

Dále IBM dodává produkty pro správu vytvořených služeb (IBM Rational Portfolio Manager), řízení vývoje SOA aplikací v RUP (RUP Plug-In for SOA a RUP Plug-In for WebSphere Business Modeler), testování funkčnosti (IBM Rational Functional Tester) a výkonu (IBM Rational Performance Tester). S těmito nástroji může vývoj probíhat přesně podle metodiky ABD popisované v kapitole 4.3.



Obrázek 4-7 Vývoj SOA aplikací pomocí nástrojů IBM[61]

5. Nástroje s podporou MDA/MDD

Nástroje jsem hodnotil z pohledu vývojáře. Pro tento přístup mám dostatek zkušeností z praxe. Vyvíjel jsem monolitické, dvouvrstvé a třívrstvé aplikace pro firemní zákazníky v různých verzích Borland Delphi, vývojových prostředí pro Java, PHP a C++. Myslím, že dokážu dobře posoudit jejich přínosy pro tuto oblast nasazení. Výčet uvedených nástrojů není úplný, výběr je omezen na několik nejzajímavějších. Obsáhlý, nikoliv kompletní seznam nástrojů je na http://www.omg.org/mda/committed-products.htm.

5.1. Hodnocení a klasifikace nástrojů s podporou MDA/MDD

Při zkoumání nástrojů s podporou MDA jsem našel oblasti vlastností, ve kterých se tyto nástroje od sebe odlišují, přistupují k nim různě a mnohé z nich jsou v některé takové oblasti velmi inovativní. Přehled těchto vlastností umožní představu o možnostech těchto nástrojů a usnadní prozkoumání a hodnocení dalších nástrojů s podporou MDA.

U popisovaných nástrojů se vždy zaměřím na ty vlastnosti, které činí popisovaný nástroj výjimečným vzhledem k ostatním. Ve většině popisovaných nástrojů jsem implementoval různé jednoduché testovací aplikace.

Pro pochopení funkce nástrojů bylo nutné podrobně nastudovat používané standardy a podporované platformy.

5.1.1. Validace transformovaného modelu

Před provedením transformace modelu provádí většina nástrojů jeho validaci. Cílem validace je zjistit, zda model splňuje požadavky této transformace. Jsou to zejména: *použití odpovídajícího modelovacího jazyka a profilu, správné nastavení stereotypů a zadání pojmenovaných hodnot, pojmenování vztahů v modelu tříd* apod. Základní rozdíl v přístupu k validaci modelu je mezi nástroji, které tvoří integrované prostředí pro modelování a nástroji spouštěnými z příkazové řádky:

- Nástroj **nedovolí vytvořit nevalidní model**
- Nástroj **vede k vytváření validních modelů**
- Nástroj **umožňuje explicitní validaci modelu bez provedení transformace**
- Nástroj **při transformaci vypíše, které části modelu nejsou validní** (textově nebo graficky označením v modelu)

- Nástroj **při transformaci nevalidního modelu skončí chybou** na prvním nevalidním prvku v modelu

5.1.2. Založeno na MOF, UML, jiných jazycích

Podle [24] se mezi dodavateli nástrojů sdruženými v OMG zatím vedou nerozhodnuté spory o tom, zda MDA založit na MOF nebo pouze na UML.

Zastánci UML jsou „sdruženi" v 2U Partners. Patří mezi ně např. Hewlett-Packard, IBM, Ericsson, Unisys. Chtějí vytvořit přesnější specifikaci UML a potom UML rozšiřovat o profily pro podporu různých platforem, tzv. DSM (Domain Specific Modeling).

Zastánci MOF tvrdí, že svět není pouze objektový a jsou situace, kdy jsou vhodnější jiné pohledy na systém, např. datově orientované, hierarchické nebo servisní. Tj. chtějí vytvářet různé DSL (Domain Specific Languages). Tyto firmy se sdružují v U2, jedná se např. o Softlab nebo JP Morgan. Na straně specializovaných jazyků je i Microsoft se svým konceptem „Software Factories" a tvůrci Eclipse Modeling Framework (EMF).

V praxi je UML de facto standardem pro modelování. OMG zatím standardizovala jediný další jazyk založený na MOF – nepříliš rozšířený CWM. Ostatní oblasti se snaží pokrýt profily, např. EDOC pro oblast služeb. V nástrojích převažuje UML - nástroje implementují nějakou metodiku pro OOAD. Důsledky toho omezení pro použití MDA v SOA jsou popsány v kapitole 4.2.

Podle výše uvedeného se dělí i nástroje s podporou MDA:

- **Nástroje založené na MOF** – modely mohou být v jakémkoliv jazyku vytvořeném v MOF, transformace se potom definují na metamodelech těchto jazyků.
- **Nástroje omezené na UML** – modely mohou být pouze v UML a transformace jsou z UML do UML, používají se zejména UML profily a značkování.
- **Vlastní doménově specifický jazyk** nebo ostatní nestandardizované jazyky.

5.1.3. Podporovaná standardy

Jelikož jsou základem MDA standardy, je pro přenositelnost modelů a integraci aplikací důležité, aby nástroje tyto standardy podporovaly. Schvalování standardů je pomalé, standardy nejsou dostatečně zralé nebo je těžké je implementovat. Proto výrobci nástrojů většinou některé standardy podporují pouze částečně, v neaktuální verzi nebo používají vlastní řešení (zejména pro popis transformací a mapování). To do značné míry omezuje automatickou integraci aplikací postavených na MDA.

Nejdůležitější standardy, jejich verze a způsoby implementace v testovaných nástrojích jsou:

- **MOF 1.4** nebo **2.0** – vytváření nových modelovacích jazyků, nepřímo popis transformací mezi modely
- **UML 1.4, 1.5, 2.0** – vytváření modelů
- **OCL** dle verze UML - zápis omezujících podmínek
- **UML profily** – CORBA, CCM, EDOC, EAI, QoS and Fault Tolerance, Testing, Schedulability, performance, and time – vytváření modelů pro specifickou oblast nasazení
- **XMI 1.0, 1.1, 1.2, 1.3, 2.1** – import a export (dodavatelé mají často „vlastní" verze XMI, např. XMI 1.3 Unisys pro UML 1.3 s rozšířeními Unisys)
- **QVT** dle verze MOF – popis transformací a dotazů nad modely
- **CWM 1.1** – modelování databází
- **Transformace** z PIM dle standardizovaných mapování (profilů pro PSM) - CORBA a Java
- **Executable UML**
- **EMF** – Eclipse Modeling Framework, modely podporované pomocí modulů pro platformu Eclipse, dnes prudce se rozvíjející oblast DSL jazyků.

5.1.4. Možné transformace

V MDA jsou dva základní typy transformací: z modelu do modelu a z modelu do kódu. Velká část nástrojů podporuje pouze transformaci z modelu do kódu, jsou to tedy pouze generátory kódu. Existují také nástroje, které přímo interpretují model a nevytváří uživatelem upravovatelné modely s nižší úrovní abstrakce a kód. Možnosti jsou tedy následující:

- **Z modelu do modelu**
- **Z modelu do kódu nebo textu (dokumentace)**
- **Bez transformací**

5.1.5. Generátor kódu nebo interpretr modelu

S předchozím bodem souvisí přístup k realizaci MDA v nástrojích, které jsou integrovaným prostředím (nejen dávkovou utilitou). Jsou dva základní přístupy: **generátor** a **interpretr**. Některé nástroje tyto dva přístupy kombinují.

- **Generátor** provádí transformace PIM do PSM a PSM do kódu, vygenerovaný PSM model i kód může (musí) být doplňován. Tento přístup vychází ze znalostí a přístupů, které již dlouho existují v CASE nástrojích. Proto je v současné době nejpoužívanějším řešením. Vývojový cyklus vypadá při použití generátoru jako vzor popsaný v kapitole 2.4. Výhodou je, že vygenerované modely a kód je možné optimalizovat pro danou platformu např. za účelem výkonu.
- **Interpretr** používá model jako kód pro virtuální stroj. Model je možné ladit nebo dokonce měnit za běhu. Virtuální stroj buď provádí transparentní překlad modelu do mezikódu (což je použití generátoru bez možnosti upravovat výstup) nebo interpretuje model přímo, tj. pracuje na vyšší úrovni abstrakce. Popis transformací mezi modely je tím ukryt do virtuálního stroje, který by měl nabízet možnosti rozsáhlé parametrizace a podporovat různé platformy. Vývoj kvalitního virtuálního stroje je velmi složitý, proto není tento přístup příliš rozšířený a používá se zejména jako doplněk k předchozímu při validaci a testování PIM modelů.
- **Kombinace obou předchozích přístupů**

5.1.6. Modely a elementy, které je možné transformovat

Jednotlivé nástroje se od sebe liší také tím, jaké elementy a modely jsou schopny transformovat. Většina nástrojů je omezena pouze na statickou strukturu, všechny nástroje podporují alespoň modely tříd. I když některé nástroje podporují možnost transformovat libovolné typy modelů, jsou ukázkové (hotové) transformace v nich omezeny jen na modely tříd. Pouze menšina nástrojů zatím dokáže transformovat dynamické prvky v modelech – např. metody zapsané v ASL, kontrolovat za běhu invarianty v OCL nebo podporovat metody pro změnu stavu ve stavových diagramech. Základní rozdělní kopíruje strukturu UML:

- **Modely byznys procesů**
- **Modely nasazení**
- **Modely aktivit**
- **Stavový diagram**
  - více stavových diagramů pro jednu třídu
  - akce pro změnu stavu a při změně stavu
  - omezení pro změnu stavu v OCL
- **Modely tříd**
  - popis metod v některém jazyku pro zápis akcí (např. ASL)

- dědičnost (zejména pro ukládání dat)
- odvozené atributy zapsané např. v OCL
- omezení např. v OCL

5.1.7. Počet řádek kódu potřebných k dokončení aplikace

Jedním z kritérií hodnocení efektivity nástroje je počet řádek kódů nutných pro dokončení modelové aplikace po transformaci nejnižší úrovně modelu do kódu. Takové zkoumání by bylo časově náročné a znamenalo by dokonalé zvládnutí daných nástrojů, protože modelová aplikace nemůže být příliš jednoduchá. Tuto informaci proto uvedu pouze tam, kde takový test již existuje, ať už provedený tvůrcem nástroje nebo třetí stranou.

5.1.8. Podpora prototypování a testování pomocí automaticky generovaného uživatelského rozhraní

Pro zkrácení doby iterace mezi jednotlivými verzemi modelu je výhodné vygenerovat pro model uživatelské rozhraní, které umožní jeho testování. V některých nástrojích lze vygenerovat jednoduché uživatelské rozhraní přímo z PIM modelu. Toto se pak prezentuje zákazníkovi nebo analytikovi. Dle mých zkušeností mu budou rozumět daleko lépe než přímo modelům. Podobné rozhraní může využívat i programátor pro ladění rozhraní mezi komponentami aplikace.

5.1.9. Zapojení modelu při vytváření uživatelského rozhraní

Jelikož je obtížné modelovat a popisovat transformacemi kvalitní uživatelské rozhraní, podporují některé nástroje „propojení" modelu s uživatelským rozhraním. Umožňují např. vkládání automaticky generovaných tabulek a editorů pro jednotlivé třídy nebo zapsat v OCL omezení pro akce uživatelského rozhraní.

5.1.10. Způsob popisu transformace

Nejen z důvodů, které byly popsány v kapitole 3.3 vyvinuli tvůrci nástrojů různé způsoby popisu transformací mezi modely a z modelu do kódu. Ve všech případech je nutné použít značkování zdrojového modelu např. pomocí profilů – stereotypů a pojmenovaných hodnot:

- **Speciální imperativní API** – pro přístup k elementům modelu a vytváření nového modelu.

- **Šablony** – většinou transformace do textu, v šablonách se odkazujeme na elementy modelu, součástí šablon může být imperativní API pro přístup k elementům modelu.
- **Transformace založené na vzorech** – rozpoznávání známých vzorů v modelu a popis jak určitý vzor transformovat.
- **QVT**
- **XSLT** – transformace modelů uložených v XMI do XMI nebo textu.
- **Bez možnosti měnit transformace** – transformace není možné měnit nebo jsou transparentní.

5.1.11. Trasovatelnost a dvoucestnost, podpora iterativního vývoje

Při vývoji informačních systému je trasovatolnost velmi důležitá. MDA přináší možnost trasovat požadavky z CIM modelu až do kódů a naopak. Pro zachování těchto vazeb je důležité, aby nástroje umožňující modifikovat modely z nižších úrovní byly dvoucestné. Dvoucestnost je podstatná také pro iterativní vývoj (provádění transformací). Při opětovném spuštění transformace musí v cílovém modelu nebo kódu zůstat dodatečně provedené úpravy. Ne všechny nástroje jsou dvoucestné nebo podporují iterativní vývoj. Trasovatelnost je typicky implementována pomocí „protokolu“ o transformaci. Z pohledu dvoucestnosti a podpory iterativního vývoje existují následující přístupy:

- **Podpora iterativního vývoje přímo v jazyku nebo API pro popis transformací**
- **Promítnutí změn v nižší úrovni modelu/kódu do vyšší úrovně** (funkce nástroje pro provádění transformace)
- **Zachování změn kódu/modelu při iterativním provádění transformace**
  - Označení částí modelu v nižší úrovni, které nejsou validní vzhledem k vyšší úrovni
  - Zachování kódu zapsaného do označených oblastí
- **Přepsání upraveného kódu/modelu**

5.1.12. Podpora pro napojení na existující aplikace, reverzní inženýrství

Informační systémy dnes neexistují odděleně, ale musí být spolu navzájem propojeny. Proto je žádoucí, aby nám nástroj usnadnil vytváření takových propojení. Obdobou propojování informačních systémů je vývoj nové verze systému, v tom případě je nutné vytvářet např. napojení na existující databázi nebo je žádoucí využít hotové části systému. Pokud propojované informační systémy nebudou vyvinuty pomocí MDA, bude nutné použít reverzní inže-

nýrství, které vytvoří ze softwarových artefaktů modely. Ty potom zahrneme do vyvíjeného (modelovaného) systému. Podpora pro reverzní inženýrství je pouze v některých nástrojích:

- **Reverzní inženýrství databáze** – je nejčastěji podporovanou možností, zřejmě protože modelování databází má velkou tradici a je častou praxí začínat při vývoji aplikací návrhem databáze
- **Reverzní inženýrství kódu** – viz kapitola 2.3.10 a 2.3.11
- **Napojení na existující aplikace pomocí „proxy"** – neprovádí se reverzní inženýrství celých aplikací, ale pouze jejich rozhraní, která jsou potom v modelu transparentně obalena (např. import rozhraní webové služby)

5.1.13. Podpora pro SOA

Možnosti využití MDA při vývoji SOA aplikací byly rozebrány v kapitole 4. Při zkoumání nástrojů budu tedy zjišťovat možnosti uvedené v kapitole 4.

5.1.14. Uživatelské rozhraní nástrojů

Jelikož MDA navazuje na generování kódu a úpravy modelů v CASE nástrojích, disponuje dnes řada těchto nástrojů možností provádět transformace mezi modely nebo z modelu do kódu. Naopak různé IDE do sebe postupně integrují možnosti vytvářet modely a generovat z nich kód nebo existující kód vizualizovat. Díky standardům pro výměnu modelů (XMI) mohla vzniknout řada nástrojů, které pouze provádí transformace modelu vytvořeného v jiné aplikaci. Základní typy uživatelského rozhraní jsou:

- **Nástroje pro dávkové zpracování z příkazové řádky** – jejich základním nedostatkem je nepohodlná validace vstupních modelů (srovnatelné s překladem z příkazové řádky). Jedná se spíše o generátory kódu. Některé je možné integrovat do vývojového procesu pomocí nástrojů pro řízení překladu (např. z Ant nebo Maven skriptu).
- **Nástroje pro dávkové zpracování s uživatelským rozhraním**
- **Nástroje pro dávkové zpracování integrovatelné do existujících vývojových prostředí** – nejčastějším případem jsou nástroje integrované do platformy Eclipse, které jako zdroj používají modely z EMF. Jsou to většinou také pouze generátory kódu omezené na jazyk vývojového prostředí.
- **Modelovací nástroje s možností provádět transformace** – umí provádět transformace mezi modely a z modelů do kódu. Výsledný kód je potom nutné dále upravovat v nějakém jiném vývojovém prostředí.

- **Integrované modelovací a vývojové prostředí s možností provádět transformace** – nejrozsáhlejší a nejkvalitnější nástroje. Těsná integrace s IDE (integrované vývojové prostředí) jim umožňuje dobrou trasovatelnost z modelu až do kódu nebo výbornou podporu dvoucestnosti. Tyto nástroje můžeme dále rozdělit na:
    - Nástroj vysoce integrovaný do IDE – např. M1Global MDE Studio
    - IDE „dodatečně" rozšířené o možnosti modelování – např. Borland Developer Studio 2006
    - Modelovací a vývojový nástroj integrovaný na stejně platformě – např. IBM Rational Architect a IBM Rational Application Developer

5.1.15. Podporované platformy

Jelikož je MDA platformově nezávislá, nástroje s její podporou mohou generovat PSM modely a kód pro různé platformy. Většina nástrojů je však omezena na konkrétní platformu, která je nejčastěji založena na Java. Orientace na Java je zřejmě dána tím, že OMG zatím standardizovala pouze profily pro Java (z programovacích jazyků) a že pro Java existuje otevřená platforma Eclipse, nad kterou je vytváření modelovacích a vývojářských nástrojů snadné. Nejrozšířenější platformy jsou následující:

- **Programovací jazyky** (pro které je možné vygenerovat kód nebo i PSM)
    - Java
    - C#
    - C++
    - Object Pascal
- **Middleware** (pro který je možné vygenerovat kód nebo PSM)
    - Web Services – XML/SOAP
    - CORBA
    - J2EE - Java RMI, EJB
    - .NET
- **Podporované technologie pro ukládání dat**
    - Perzistence zahrnutá v aplikačním serveru pro J2EE
    - Databáze podporované v ADO .NET (BDP)
    - Frameworky pro ukládání dat Castor JDO a Hibernate

5.1.16. Podpora metrik a auditů pro modely

Pro dosažení kvality při vývoji MDA aplikací je důležitá možnost provádění auditů a vyhodnocování metrik nad modely. Krajní hodnoty některých metrik kontrolované automatickým auditem mohou odhalit chybný návrh (např. velká hloubka dědičnosti nebo vysoká spřaženost a malá soudržnost). Audity nemusí používat pouze metriky, mohou v modelu vyhledávat špatné návrhové vzory, označit nepřenositelné konstrukce nebo špatně pojmenované elementy.

5.1.17. Podpora pro znovupoužitelnost transformací a modelů

Pro úspěšné nasazení v praxi je důležité, aby alespoň část artefaktů vytvářených během vývojového procesu byla znovupoužitelná. Nad rámec tradičních vývojových nástrojů a postupů přináší MDA možnost znovu použít přímo modely a transformace:

- **Znovupoužitelnost modelů, návrhových vzorů a profilů** – ukládání částí modelů a profilů jako vzorů do repositáře, import vzorů, modelů a profilů z jiných nástrojů. Použití transformací založených na vzorech a profilech.
- **Univerzálnost transformací** – univerzálně definované transformace je možné použít v mnoha projektech. Transformace slouží jako další místo k „ukládání znalostí“ o vývoji aplikací pro konkrétní platformu.
- **Přenositelnost transformací** mezi různými nástroji je zatím nereálná. Většina nástrojů totiž používá nestandardní metody pro popis transformací. MDA tak paradoxně váže vývojáře na konkrétní nástroj a neumožňuje jim snadno integrovat aplikace vytvořené v různých nástrojích.

5.1.18. Odezva, kvalita a stabilita

Jedná se o subjektivní hodnocení testovaných nástrojů. Testovací konfigurace byla P4 2,4 GHz, 2 GB RAM a oddělené pevné disky 7200 ot./s pro data, programy a stránkovací soubor. Srovnávat mohu s tradičnými prostředími, ve kterých jsem napsal statisíce řádek kódu, tj. Eclipse, Borland Delphi apod. Odezva, kvalita a stabilita nástrojů jsou zásadní pro praktické nasazení, není příjemné místo ladění vlastních chyb v kódu bojovat s chybami nástroje a po každé změně modelu čekat minuty na provedení transformace. Proto je dobré hodnotit a zejména srovnávat mezi jednotlivými nástroji:

- Odezvu vývojového prostředí
- Rychlost transformací podobně složitých projektů

- Stabilitu aplikací během testování
- Složitost ovládání (vzhledem k mým zkušenostem s modelovacími nástroji a vývojem aplikací)
- Funkčnost nástrojů při vytváření vlastních testovacích aplikací
- Funkčnost příkladů dodávaných k aplikaci
- Kvalitu nápovědy, dokumentace a uživatelské podpory (pouze pokud je poskytována zdarma)

5.1.19. Licence, cena, testovací verze

Pro to, aby firma začala MDA nástroje používat, je důležitá i cena nástrojů a možnost nástroje předem vyzkoušet.

- **Open source** – zvládnout kvalitně MDA je příliš nákladné, takže Open Source řešení nejsou ve srovnání s komerčními nástroji zatím dostatečně komplexní.
- **Možnosti testovací verze** – všechny mnou uváděné nástroje nabízí testovací verze, které jsem také vyzkoušel. U některých nástrojů jsem měl přístup k plné verzi a proto jsem otestoval tuto.
- **Cena**

Následuje přehled, kde u jednotlivých nástrojů popisuji zejména vlastnosti, kterými se liší od ostatních a které jsou z pohledu MDA zajímavé. Společné hodnocení všech nástrojů dle všech právě uvedených kritérií je v tabulce na konci kapitoly 5.

5.2. Borland Developer Studio 2006

Podpora pro MDA se poprvé objevila v Delphi 6 jako rozšíření Bold for Delphi od švédské firmy Bold Soft. Borland tuto firmu posléze koupil. Od Delphi verze 8 byl proto Bold for Delphi přejmenován na ECO Framework (Enteprise Core Objects). Testoval jsem poslední verzi Delphi, které Borland sjednotil s dalšími produkty pod jednu obchodní značku Borland Developer Studio 2006 (Delphi, C++ Builder a C# Builder dohromady). ECO framework je od verze III možné používat i pro aplikace v C# (verze III je právě v Borland Developer Studio 2006).

Velmi zajímavě je v ECO frameworku vyřešena „transformace" z PIM modelu do PSM modelu a kódu. Je vygenerován PSM model tříd a kostra kódu, která pro metody zapsané ASL, vyhodnocování výrazů v OCL nebo při kontrole omezení volá interpretr PSM modelu

(rozhraní frameworku) – parametrem volání jsou názvy prvků v modelu. Toto řešení umožňuje měnit model za běhu aplikace, volat nově vytvořené metody a používat nové atributy nebo třídy. Zároveň lze vygenerovanou kostru upravovat a dopsat tak vlastní metody v C# nebo Pascalu (např. v použité verzi ASL není možné zapsat metodu pro násobení měny a času). Kostru kódu také používají standardní komponenty při návrhu uživatelského rozhraní ve VCL .NET. Kromě kódu je vygenerována i databáze. Framework se stará o ukládání objektů. Je možné pracovat s databázemi podporovanými v ADO .NET nebo použít XML soubor. Nelze nijak upravovat transformace a aplikace jsou omezeny možnostmi frameworku.

Mapování z PIM do PSM je jednoduché. Každé třídě v PIM odpovídá třída a rozhraní v PSM. První z PSM tříd přímo odpovídá třídě v PIM modelu rozšířené o metody pro potřeby frameworku. Druhá reprezentuje rozhraní pro práci se seznamy objektů této třídy.

Modelovací část je odvozena od Borland Together. Podporuje používání návrhových vzorů při vytváření modelů. Umožňuje vytvářet model tříd a pro jednotlivé třídy v modelu vytvářet stavový diagram v jazyku odvozeném od UML 2.0. Všechny modelované elementy je možné použít při další tvorbě aplikace. Z modelu lze generovat dokumentaci v HTML. Nad modelem je možné provádět audity a zjišťovat hodnoty různých metrik.

V modelu tříd odpovídajícímu statické struktuře aplikace je možné použít agregace, dědičnost tříd a rozhraní, v OCL zapisovat invarianty, odvozené atributy a omezení volání metod. V ASL, jehož syntaxe je od OCL odvozena, lze zapsat těla metod. Pro třídy je možné nastavit název tabulky v databázi (mohou existovat i dočasné třídy), pro jejich atributy potom jména sloupců (zvolit zda je atribut přechodný nebo perzistentní).

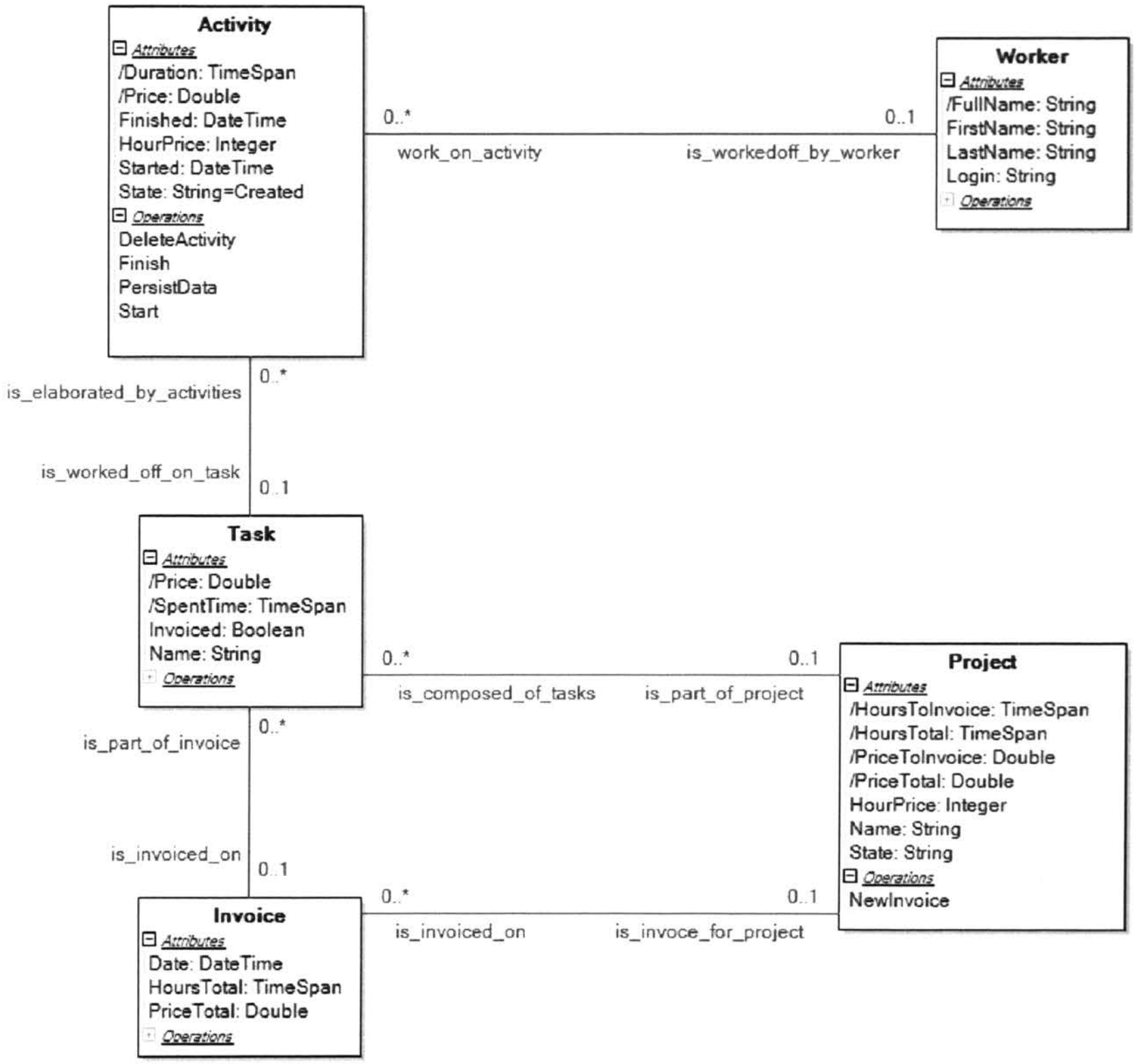
Ve stavovém diagramu je možné v OCL zapsat omezení pro změnu stavu, v ASL potom akce při změnách stavu. Pro změnu stavu se používají speciální metody z diagramu tříd, tzv. spouště.

Pro návrh uživatelského rozhraní lze používat komponenty zapouzdřující .NET 1.1. Dále jsou k dispozici komponenty pro zpracování dotazů zapsaných v OCL za běhu (např. pro zobrazování v tabulkách). V GUI lze používat omezení akcí zapsané v OCL v návaznosti na objekt vybraný v určité „tabulce“. Je možné vytvářet webové aplikace. Pro rychlé prototypování – testování modelů – lze použít „Autoforms“. To je automaticky vygenerované webové rozhraní, které umožňuje editaci, změnu stavů a volání metod nad všemi třídami z modelu.

Funkční jádro aplikace („transformovaný model“ a doplněný kód) lze exportovat jako DLL pro použití v dalších projektech. Framework podporuje reverzní inženýrství, existující databázi je možné „zabalit“ do Eco Space, tj. rozšířit ji o tabulky a sloupce používané frameworkem a vygenerovat odpovídající PIM model tříd.

Nástroj jsem testoval vytvořením aplikace Worksequencer. Tato aplikace slouží pro kontrolu času stráveného členy týmu při jednotlivých činnostech v prostředí s více projekty. Stačilo naprogramovat asi 10 řádek kódu v Pascalu. Vše ostatní bylo možné vytvořit pomocí modelů nebo návrháře uživatelského rozhraní.

Na následujících obrázcích jsou ukázky postupu vytváření aplikace. Nejdříve se vytváří PIM modely – model tříd zachycující statickou strukturu tříd v aplikaci (obrázek 5-1) a stavové diagramy pro jednotlivé třídy (obrázek 5-2).



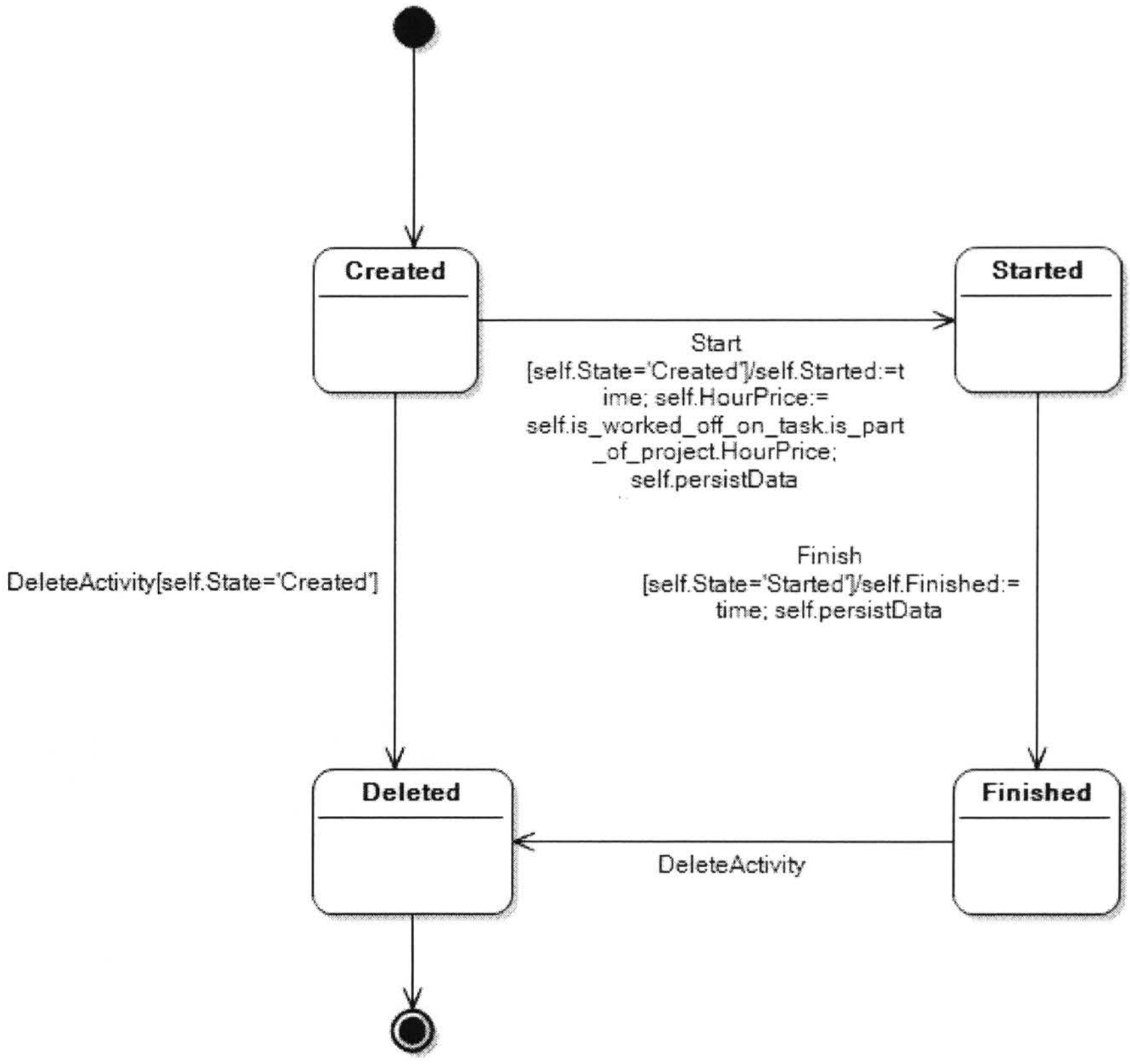
Obrázek 5-1 PIM – diagram tříd

V modelu tříd jsou několikrát použity odvozené atributy zapsané v OCL, např. atribut Project.HoursToInvoice:

```
self.is_composed_of_tasks
->select(Invoiced=false).SpentTime->sumTime
```

Metoda zapsaná v ASL má v Pascalu kód, který pouze se správnými parametry volá framework:

```
procedure Activity.Finish(time: System.DateTime);

type

  TArrayOfobject = array of &object;

begin

  Self.AsIObject.Invoke('Finish',

        TArrayOfobject.Create((System.Object(time))));

end;
```



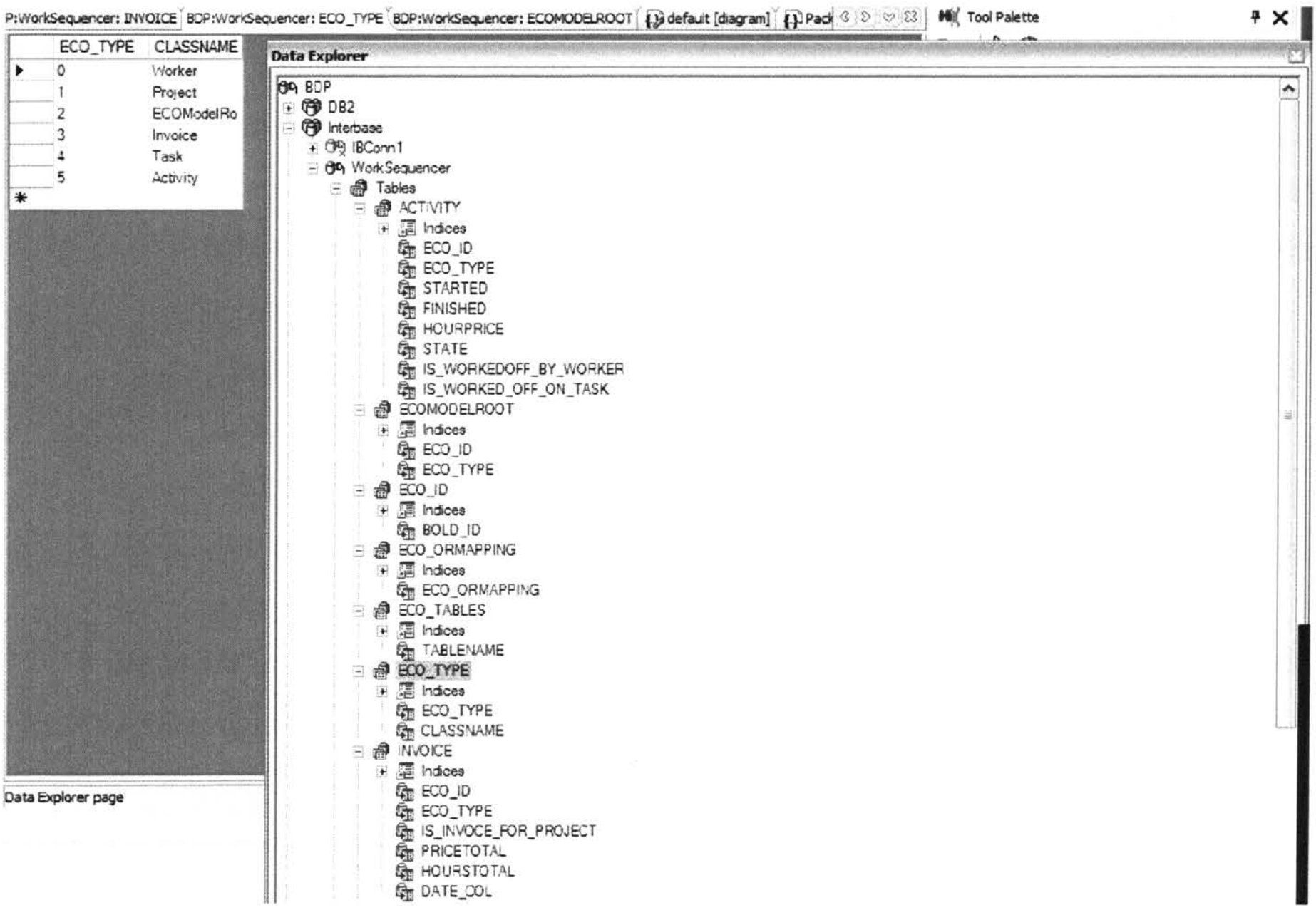
Obrázek 5-2 PIM - stavový diagram pro třídu Activity

Některé metody není možné zapsat v ASL, proto je nutný zápis přímo v kódu. Jedna taková metoda je v následující ukázce:

```
function Activ-
ty.DurationDeriveAndSubscribe(reevaluateSubscriber,
```

```
        resubscribeSubscriber: ISubscriber): system.object;
begin
  AsIObject.Properties['Started'].
     SubscribeToValue(reevaluateSubscriber);
  AsIObject.Properties['Finished'].
     SubscribeToValue(reevaluateSubscriber);
  if  Finished.CompareTo(Started)>0 then
     result := Finished.Subtract(Started)
  else
     result:=TimeSpan.Zero;
end;
```

Po vytvoření PSM modelů a kódu a nastavení přístupů přes ADO .NET je automaticky vygenerována struktura databáze. Část struktury je zobrazena na obrázku 5-3. Framework používá pomocné tabulky a sloupce pro uložení informací o jednotlivých třídách.



Obrázek 5-3 Vygenerované schéma pro databázi Interbase

Potom se s pomocí komponent frameworku a VCL.NET vytváří uživatelské rozhraní. Pro zobrazování objektů v tabulkách se užívají komponenty, které umí vyhodnotit OCL „dotazy“ nebo omezení pro akce. Např.:

- OCL jako „dotaz“ na všechny projekty: `Project.allInstances`

- OCL pro činnosti na konkrétním projektu: `self.is_composed_of_tasks`
- Omezení pro položku v menu: `self.state='Started'`

Během vytváření aplikace jsem narazil na několik nedostatků, které znesnadňují praktické použití:

- Editor modelů neumožňuje export a import XMI, takže nelze přenést modely do jiných aplikací.
- Nefunguje automatické přejmenování názvů sloupců v databázi při změně jmen atributů. Tyto změny je nutné dokumentovat a provádět odděleně.
- Jsou podporovány pouze .NET 1.1 komponenty pro uživatelské rozhraní, verze 1.1 byla už v době vydání Borland Developer Studio 2006 zastaralá (aktuální je .NET 2.0).
- Typová a syntaktická kontrola OCL výrazů a ASL metod při jejich zápisu je slabá, takže většina chyb se projeví až za běhu aplikace. V kombinaci se špatnou nápovědou to vede ke zdlouhavému ladění.
- Pro chybu v ASL/OCL editoru nelze pohodlně zapsat metody s více řádky.
- Nemožnost napsat všechny metody v ASL vede k nepřehledným kombinacím kódu rozděleného mezi ASL a Pascal – není totiž možná přímá navigace z modelu do kódu.
- Při kopírování komponent uživatelského rozhrání dochází ke smazání všech jejich atributů.
- Chybí podpora refaktoringu modelu, zejména uvnitř OCL výrazů, jejichž špatnou syntaxi zjistíme až za běhu aplikace. To vyžaduje testy s velkým funkčním pokrytím.

I přesto, že s pomocí ECO frameworku je možné v Borland Developer Studio 2006 rychle vytvářet aplikace (viz můj přiklad), není zatím v praxi moc používán. Borland ve svých prezentacích uvádí reference pouze na dvě menší firmy. Dle diskuze v konferenci uživatelů Delphi jde o začarovaný kruh – je to nová, neznámá a nepoužívaná technologie a její použití je potencionálně rizikové.

5.3. Borland Together 2006 Architect

Nejnovější verze modelovacího nástroje firmy Borland podporuje vytváření modelů v UML, EMF Ecore (Eclipse Modeling Framework) a BPMN (Business Process Modeling

Notation). Nástroj je postaven na platformě Eclipse, což umožňuje další úpravy vygenerovaného Java kódu v tomto prostředí.

Možnosti modelování jsou omezeny neúplnou podporou UML 2.0 (nástroj byl uveden na trh krátce po jeho schválení jako standardu). Při vytváření modelů je možné používat návrhové vzory. Vzory je možné ukládat do repositáře. Lze vytvářet a používat UML profily. Modely mohou být exportovány a importovány v XMI. Nad vytvořenými modely je možné vyhodnocovat metriky a provádět audity podobně jako v Borland Developer Studio.

Together umožňuje transformace mezi UML a EMF modely popsané v QVT. Ze zápisu transformace v QVT je automaticky generován Java kód modulu pro Eclipse, který potom transformaci provádí. Modul k tomu používá Java API pro práci s metamodely UML a EMF. Generování kódu modulu odpovídá převodu z deklarativního jazyka QVT do imperativní Java. Z QVT je navíc možné volat funkce napsané v Java. Pomocí téhož Java API lze vytvářet transformace z UML do textu. API je určeno pouze pro přístup k modelům, nijak neusnadňuje jejich transformace nebo převod do kódu, jak je tomu v případě IBM Rational Architect popisovaného v následující kapitole.

Funkce pro BPM byly popsány v kapitole 4.5.2 o použití tohoto nástroje při vývoji SOA aplikací.

V Borland Together 2006 Architect jsem si vyzkoušel vytváření profilů, modelů a psaní transformací v QVT. Narazil jsem přitom na většinu nedostatků QVT popsaných v kapitole 3.3.1. Dalším nedostatkem je slabá podpora všech vlastností UML 2.0, která omezuje např. modelování byznys procesů – „Activity Partition“ v modelu aktivit neumožňuje nastavení vlastnosti „Represents“, proto nelze odkazovat na interface správného procesu (viz kapitola 4.4.5).

5.4. IBM Rational Architect

Nástroje z kolekce IBM Rational jsou špičkou na trhu. Podpora MDA se poprvé objevila v produktu IBM Rational XDE v roce 2002. V první verzi byl podporován deklarativní zápis transformací pomocí vzorů. V roce 2003 byl v nové verzi IBM Rational XDE for Java zaveden imperativní (procedurální) přístup k vytváření transformací pomocí API pro manipulaci s modely. Tento způsob je použit i v poslední mnou testované verzi IBM Rational Architect 6.0.

Modelovací část plně podporuje UML 2.0. Modely mohou používat UML profily. Aplikace poskytuje kvalitní podporu pro vytváření a správu profilů. Při vytváření modelů je možné používat vzory, které lze ukládat do repozitáře.

API rozhraní pro vývoj transformací poskytuje nejen primitivní metody pro přístup k prvkům modely, ale zejména vysokoúrovňové funkce, jako je *inteligentní a hluboké kopírování prvků modelu* nebo *podpora UML uživatelských profilů*. Toto API netrpí některými nedostatky QVT – podporuje transformaci a snadné propojení z jednoho modelu do více modelů, obsahuje podporu pro vytváření jmen kompatibilních s jazyky Java a C++, umožňuje zadání omezení pro transformaci a díky imperativnímu přístupu je možné snadno vytvářet algoritmicky složité transformace. Rozhraní je přizpůsobeno iterativnímu vývoji, např. metoda pro přidávání atributu třídě v cílovém modelu kontroluje, zda třída již neobsahuje stejný atribut z předchozí iterace. V takovém případě volání metody vrátí odpovídající existující instanci atributu.

Transformace jsou vyvíjeny jako zásuvné moduly pro platformu Eclipse, nad kterou jsou vývojové nástroje od IBM postaveny. Eclipse Plugin Development Environment (PDE) poskytuje funkce pro snadnou integraci transformací do vývojového prostředí – vytvoření položky v menu, ikony v panelu nástrojů apod. Odladěné zásuvné moduly s transformací je potom možné distribuovat vývojářům v týmu nebo dále prodávat.

Kód transformace je rozdělen do tří základních částí:

- **Validace vstupních parametrů** – rozhraní pro transformace nijak neomezuje jejich vstup ani důsledky transformace. Proto je nutné před spuštěním procesu transformace otestovat integritu vstupů (tj. např. zda modely používají správné profily a jsou zadány všechny potřebné pojmenované hodnoty a nastaveny stereotypy). Jelikož složité transformace mohou trvat velmi dlouho, měla by rychlá validace ušetřit čas tím, že neumožní spuštění transformaci nad špatným vstupem.
- **Vlastní provedení transformace**
- **Verifikace správného propojení mezi prvky modelů**, které se účastnily transformace – většinou jde o kontrolu, zda vývojář svými úpravami neodstranil některé vygenerované vazby z vyšší úrovně abstrakce.

IBM Rational Architect je nejkomplexnější ze zkoušených nástrojů (umožňuje modelovat nejen objektové systémy, ale i databáze, J2EE aplikace, vytvářet XML schémata, importovat byznys procesy apod.). Transformace mezi modely i z modelu do textu fungují dobře. Základní dodané transformace (do Java, C++, CORBA a EJB) jsou velmi jednoduché a pro praktické nasazení bude nutné vytvořit transformace vlastní, složitější.

5.5. AndroMDA

AndroMDA je Open Source MDA generátor kódu z PIM modelů. Pro transformace používá tzv. „cartridges". „Cartridge" je zásuvný modul pro AndroMDA, který provádí jeden druh transformací. Součástí nástroje jsou (na rozdíl od jiných nástrojů) hotové transformace pro nejpoužívanější platformy – Java, Spring, EJB, .NET, Hibernace, Struts atd.

Vstupem generátoru jsou modely uložené v XMI používající vhodný profil. Profil odpovídá jedné „cartridge", profily jsou dobře zdokumentovány. Profily definují stereotypy pro skupiny PSM metatříd – tříd, na které se prvek označený odpovídajícím stereotypem transformuje. „Cartridge" pomocí kódu v Java parsuje model na metaobjekty, které odpovídají PSM metatřídám. Pro každou PSM metatřídu je definována Velocity šablona pro generování kódu. Vstupní model je možné omezit – validovat - podmínkami zapsanými v OCL.

Ukázka šablony pro AndroMDA – vytvoření EntityBean pro třídu v UML:

```
// ...
public abstract class ${class.name}Bean
   implements javax.ejb.EntityBean
{
   // ...
   #foreach ( $att in $class.attributes )
      #set ($atttypename = "#javaMapping($att)")
      // ...
      public abstract $atttypename
         get${str.upperCaseFirstLetter(${att.name})}();
      // ...

      public abstract void
         set${str.upperCaseFirstLetter(${att.name})}
         (${atttypename} newValue);
   #end
   // ...
   public abstract void validate();

   // ...
}
```

V nejbližší době by měl být dokončen zásuvný modul integrující AndroMDA do platformy Eclipse (v tuto chvíli je nutné transformace spouštět z příkazové řádky nebo v Maven/Ant skriptu). Díky otevřenosti je AndroMDA používáno pro generování kódu v několika projektech (nástrojích): JunoMDA - generování PHP kódu, MDARAD - modelování a generování webových aplikací založených na Java a dalších. Dle webu projektu je chystána podpora pro transformace mezi modely popsané v ATL a Java a možnost používat pro modely UML 2.0.

AndroMDA je dobře fungující generátor kódu. V porovnání s ostatními testovanými nástroji nabízí širokou paletu hotových transformací a dobrou podporu v komunitním fóru.

5.6. Interactive Object ArcStyler

ArcStyler zavádí do praxe vývoj aplikací s využitím MDA popsaný v knize [5]. Vývojář nejprve vytvoří CIM a PIM model (je možné importovat XMI). Pak je třeba vybrat cílovou platformu. ArcStyler podporuje J2EE (např. IBM WebSphere, BEA WebLogic) a Microsoft .NET. Této platformě odpovídá tzv. „MDA-cartridge". Do PIM modelu jsou dodatečné informace pro provedení transformace zaneseny pomocí značkovaní. Pro jeden PIM model je možné definovat více sad značek, můžeme ho tak transformovat na různé platformy.

V dalším kroku ArcStyler generuje kód (s PSM modely se nepracuje). Vygenerovaný kód závisí na použité „MDA-cartridge" a většinou zahrnuje celou infrastrukturu. Např. v typickém J2EE projektu ArcStyler vygeneruje zdrojový Java kód EJB tříd, testy pro JUnit, SQL skripty, EJB deskriptory, skripty pro sestavení pomocí ANT, konfigurační soubory pro nasazení aplikace a projektové soubory pro JBuilder nebo Eclipse. Všechny změny ve vygenerovaném kódu jsou zachovány při opětovném generování kódu - při jednotlivých iteracích v projektu.

Existující „MDA-cartridges" lze upravovat, funguje jejich dědičnost. Uživatel může vytvářet nové „MDA-cartridges" s pomocí průvodce, který vygeneruje jejich infrastrukturu. Pro popis transformací se používají Java API (je založeno na JMI) a šablony.

ArcStyler jako jeden z mála nástrojů umožňuje pomocí MagicDraw Teamwork serveru skutečně sdílet modely v týmu. Tento server narozdíl od textově orientovaných CVS a Subversion pracuje s jednotlivými prvky modelu. Modul pro transformace z aktuální verze ArcStyler je možné integrovat do nástrojů řady IBM Rational, které umožňují kvalitnější modelování než vlastní ArcStyler. Integrace s IBM Rational řeší i nutnost importu vygenerovaného kódu do IDE pro další úpravy.

ArcStyler je kvalitní nástroj pro provádění transformací. Součástí nástroje jsou hotové transformace na několik různých platforem. O jeho kvalitě svědčí i dostatečný počet referencí z reálných projektů. Hlavními nedostatky jsou nepříliš intuitivní ovládání modelovací části a nepřehledná dokumentace zaměřená spíše na UML než na vývojový proces dle MDA. V dokumentaci chybí kvalitní příklady. V modelovací části nástroje znatelně vázne podpora pro návrhové vzory.

5.7. Compuware OptimalJ

Compuware OptimalJ je produkt zaměřený na vývoj J2EE aplikací postavený nad platformou Eclipse. Umožňuje vytvořit platformově nezávislý model tříd, komponent a služeb. Ten v něm lze transformovat do modelu databáze, EJB a integračního modelu. Umožňuje integrovat části aplikace pomocí webových služeb, JCA (J2EE Connector Architecture – API pro integraci aplikačních serverů a informačních systémů) nebo CICS/COBOL (Customer Information Control System – transakční mainframe server). Z těchto modelů je vygenerován kód, který zahrnuje veškerou infrastrukturu aplikace. Tento kód je možné ve vývojovém prostředí dotvořit a přímo nasadit na aplikační server.

OmptimalJ je reprezentantem velké skupiny produktů, které pomocí modelování usnadňují vývoj J2EE aplikací. Podobné funkce mají např. Enterprise Architect, M1Global MDE Studio nebo MIA Generation/Transformation. Proto není těmto nástrojům věnován další prostor.

5.8. Kenedy Carter iUML

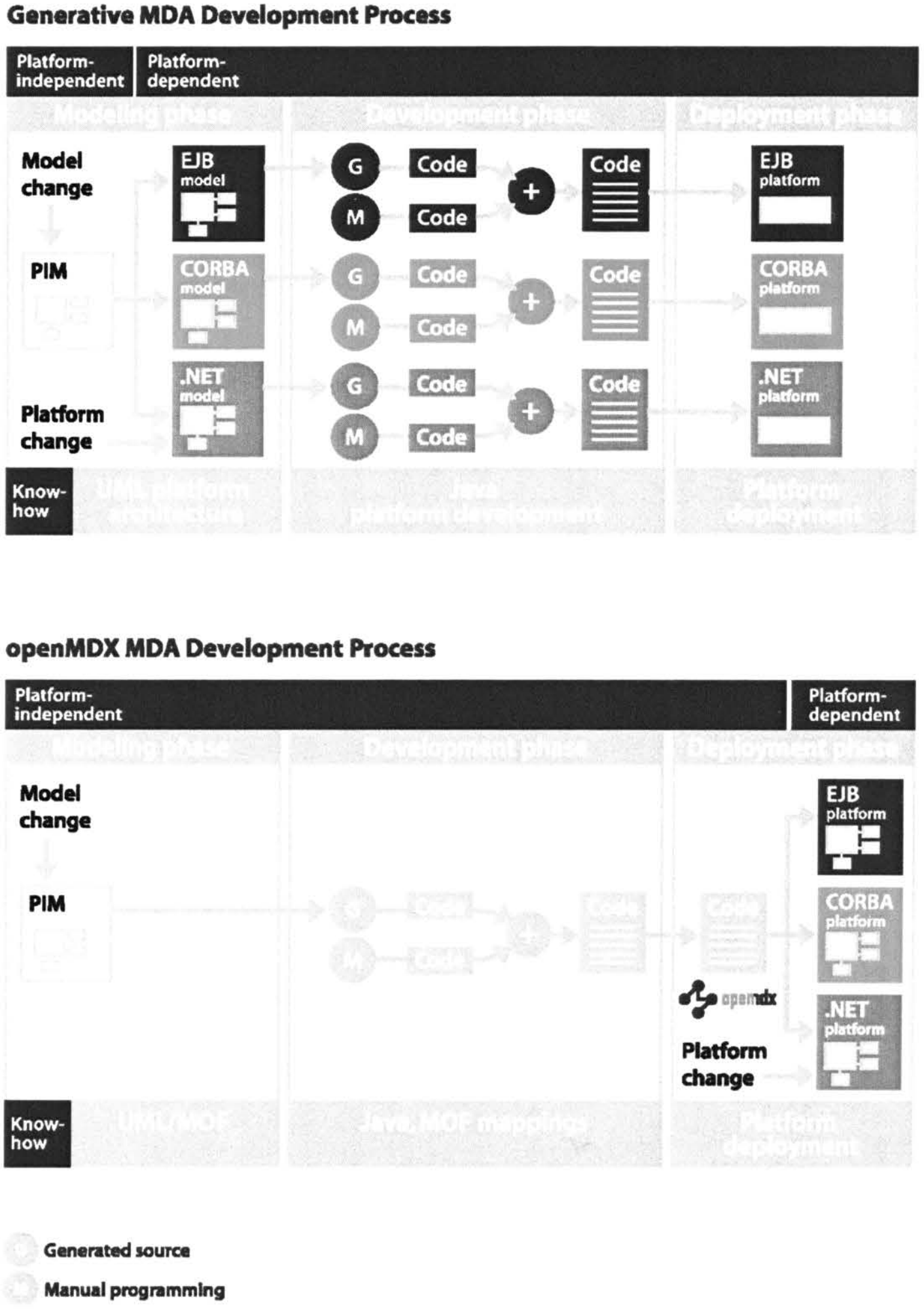
Firma Kenedy Carter je jeden z tvůrců jazyka Executable UML (viz kapitola 3.6). Produkt iUML (modelovací nástroj) spolu s iCCG (kompilátor modelu) umožňuje vygenerovat z modelu kompletní aplikaci. Tyto nástroje jsou zaměřeny na aplikace pro vestavěné systémy (embedded systems), jako referenci uvádí např. software pro bojový letoun F16. Kompilátor iCCG generuje C nebo C++ kód. Právě díky orientaci na vestavěné systémy je možné generovat 100% kódu. Jedná se totiž o přesně specifikované platformy, které dobře zapadají do rámce vývoje pomocí metodik pro OOAD.

5.9. OpenMDX

OpenMDX je framework, který zajišťuje abstraktní vrstvu mezi aplikační logikou a použitou platformou (J2EE, CORBA, .NET). Z modelu se podobně jako v Borland Developer Studio 2006 vygeneruje kostra kódu, kterou je nutné doplnit o aplikační logiku. Struktura vygenerovaného kódu je závislá na použitých platformově nezávislých stereotypech (viz služby na vnějším kruhu struktury MDA, kapitola 2.2). Stereotypy existují např. pro všechny J2EE návrhové vzory[48]. Framework pomocí zásuvných modulů zajišťuje transparentní propojení různých komponent aplikace umístitelných na rozdílných platformách. Zásuvné moduly mají rozhraní definované pro danou abstraktní funkcionalitu (např. perzistenci, uživatelské rozhra-

ní, distribuování objektů). Tento model umožňuje změnu rozmístění komponent nebo použité platformy bez nutnosti znovu generovat aplikaci. Jádro frameworku je implementováno podle standardů MOF, JMI a JDO (Java Data Objects).

Architektura openMDX je na obrázku 5-4. Je patrné, že stačí existence jednoho společného metamodelu abstraktní platformy a není nutné implementovat a spravovat různé verze PIM označených pro různé platformy, vygenerovaných PSM modelů a kódu. Tím je dosaženo snadné přenositelnosti mezi platformami. Daní za ni je nemožnost využít naplno potenciál dané platformy, protože díky abstrakci nemáme přístup ke všem jejím prostředkům.



Obrázek 5-4 Architektura openMDX[64]

5.10. Tabulka vlastností zkoumaných nástrojů

	Borland Developer Studio 2006	**Borland Together 2006**	**IBM Rational Architect**	**AndroMDA**
1. Validace transformovaného modelu	Vede k vytváření validních modelů	Při transformaci vypíše, které části modelu nejsou validní	Při transformaci vypíše, které části modelu nejsou validní	Při transformaci vypíše, které části modelu nejsou validní
2. Založeno na MOF, UML, jiných jazycích	Vlastní jazyk odvozený od UML	Částečná podpora MOF – pro práci s UML	Částečná podpora MOF – pro práci s UML	Založeno na UML 1.4, chystaná je podpora UML 2.0
3. Podporované standardy	UML, OCL nikoliv dle standardu	UML 2.0 (neúplně), MOF 2.0, XMI 2.1, QVT, profily pro EJB 2.1, OCL bez editoru s kontrolou syntaxe, EMF	UML 2.0 (úplná podpora), XMI 2.1, QVT, OCL bez editoru s kontrolou syntaxe, UML profily	UML 1.4, XMI 1.4, OCL
4. Možné transformace	Z modelu do kostry kódu a dokumentace, částečně bez transformace	Z modelu do modelu, z modelu do kódu/textu	Z modelu do modelu, z modelu do kódu a textu	Zatím z modelu do textu, chystána je podpora transformací mezi modely
5. Generátor kódu nebo interpretr modelu	Kombinace generátoru kódu a interpretru	Generátor kódu	Generátor kódu	Generátor kódu
6. Modely a elementy, které je možné transformovat	Model tříd, stavový diagram	Všechny typy modelů	Všechny typy modelů	Všechny typy modelů, zejména však modely tříd
7. Počet řádek kódu potřebných k dokončení aplikace	Subjektivní odhad - nízký	Nezměřeno	Nezměřeno	Nezměřeno
8. Podpora prototypování a testování	Automaticky generované webové rozhraní	Nutno implementovat do odpovídajících transformací	Nutno implementovat do odpovídajících transformací	Nutno implementovat do odpovídajících transformací
9. Zapojení modelu při vytváření GUI	Podporováno, např. tabulku a akce provázané s modelem	Nutno implementovat do odpovídajících transformací	Nutno implementovat do odpovídajících transformací	Implementováno v existujících transformacích
10. Způsob popisu transformace	Nelze definovat	QVT pro transformace z modelu do modelu, Java API pro transformace z modelu do textu	Vestavěné Java API, QVT pomocí rozšíření	Vestavěné Java API a Velocity šablony, pro transformaci modelů Java API a ATL
11. Trasovatelnost a dvoucestnost	Nepodporuje	Záznam z provedené transformace	Záznam z provedené transformace, podpora pro iterativní vývoj v API	Nutno implementovat do transformací
12. Podpora napojení na existující aplikace, reverzní inženýrství	Reverzní inženýrství databáze do modelu tříd	Nepodporuje	Nutno použít nástroje pro integraci aplikací	Nepodporuje
13. Podpora pro SOA	Není	Pomocí definice vhodného profilu a transformací	Import BPM, profily pro modelování SOA aplikací a navazující SOA nástroje	Pomocí definice vhodného profilu a transformací – existující transformace umožňují generovat webové služby
14. Uživatelské rozhraní nástroje	Transformace a modelář dodatečně integrované do IDE	Modelovací nástroj s podporou transformací integrovaný v Eclipse	Modelovací nástroj se při současné instalaci nástroje pro vývojáře integrují v jeden celek	Zatím dávkové zpracování, je chystaná integrace do platformy Eclipse
15. Podporované platformy	Object Pascal, C#, databáze přes ADO .NET	Java, ostatní záleží na implementovaných transformacích	Java včetně J2EE, C/C++, modelování databází	Hotové transformace pro Java, Sturts, .NET, EJB, Spring, Hibernate, …
16. Podpora metrik a auditů	Metriky, Audity	Metriky, Audity	Existuje rozšíření pro metriky i audity	Není
17. Podpora znovupoužitelnost transformací a modelů	Ukládání návrhových vzorů do repositáře	Ukládání návrhových vzorů do repositáře, QVT založené na profilech	Transformace pro Java API jsou založené na profilech a jsou tedy znovupoužitelné, při modelování lze používat vzory	Transformace jsou založeny na profilech a jsou tedy znovupoužitelné
18. Odezva , kvalita a stabilita	Rychlé provádění transformací i spouštění aplikací, spousta chyb v IDE, stabilní	Pomalé provádění transformací v QVT, stabilní	Pomalé provádění transformací, stabilní	Proti ostatním nástrojům rychlé provádění transformací, stabilní
19. Licence, cena, testovací verze	Komerční, 1100 – 3500 USD, testovací verze	Komerční, 1000 – 5000 USD, testovací verze	Komerční, 3000 – 10 000 USD, testovací verze	Zdarma – Open Source

	Interactive Object ArcStyler	**Kennedy Carter iUML**	**OpenMDX**	**Compuware OptimalJ**
1. Validace transformovaného modelu	Vede k vytváření validních modelů	Vede k vytváření validních modelů	Při transformaci vypíše, které části modelu nejsou validní	Vede k vytváření validních modelů
2. Založeno na MOF, UML, jiných jazycích	Pro modelování je podporováno UML 1.4, umí importovat modely z MOF repository	Executable UML	UML	UML
3. Podporované standardy	UML 1.4, XMI 1.4, OCL, MOF 1.4	Executable UML	UML 1.4, XMI 1.4	UML 1.3, XMI 1.2
4. Možné transformace	Z modelu do modelu, z modelu do kódu/textu	Z modelu do modelu, z modelu do kódu/textu	Z modelu do kostry kódu	Z modelu do modelu, z modelu do kódu/textu
5. Generátor kódu nebo interpretr modelu	Generátor kódu	Generátor kódu, umí vygenerovat 100% kódu	Kombinace generátoru kódu a interpretru	Generátor kódu
6. Modely a elementy, které je možné transformovat	Všechny typy modelů	Všechny typy modelů	Model tříd	Model tříd
7. Počet řádek kódu potřebných k dokončení aplikace	Nezměřeno	0	Na příkladu Sun PetStore, 2 421 místo původních 18 001	Nezměřeno
8. Podpora prototypování a testování	Vygenerování JUnit testů a testovacího webového rozhraní	Vygenerování testovací aplikace pro PIM model	Není	Není
9. Zapojení modelu při vytváření GUI	Implementováno v existujících transformacích	Pomocí propojení modelu aplikace s modelem domény GUI	Implementováno v existujících transformacích	Nutno implementovat do odpovídajících transformací
10. Způsob popisu transformace	Vlastní API založené na JMI, vygenerování základu transformace, možnost dědičnosti transformací	Rozšiřitelný kompilátor	Nelze definovat	Pomocí pravidel a šablon
11. Trasovatelnost a dvoucestnost	Záznam z provedené transformace, podpora pro iterativní vývoj v API	Záznam z provedené transformace, podpora pro iterativní vývoj	Nepodporuje	Záznam z provedené transformace, podpora pro iterativní vývoj
12. Podpora napojení na existující aplikace, reverzní inženýrství	Reverzní inženýrství kódu v Java a databázi (DDL)	Nepodporuje	Nepodporuje	Nepodporuje
13. Podpora pro SOA	Pomocí definice vhodných transformací – existující transformace umožňují generovat webové služby	Teoreticky možné, ale je mimo hlavní zaměření nástroje	Pomocí vytvoření vhodných zásuvných modulů pro požadovanou platformu, není možné, ale omezeno na OO koncepty	Podpora pro integraci pomocí webových služeb
14. Uživatelské rozhraní nástroje	Modelovací nástroj s podporou transformací, nebo integrace s nástroji IBM Rational	Modelovací nástroj s podporou transformací	Dávkové zpracování	Transformace a modelář dodatečně integrované do IDE
15. Podporované platformy	Java, J2EE, .NET, ASP .NET, C#, další dle použitých transformací	C++,C zejména vložené systémy	Java, J2EE, .NET	Java, J2EE, .NET
16. Podpora metrik a auditů	Není	Není	Není	Není
17. Podpora znovupoužitelnost transformací a modelů	Dědičnost transformací, chybí podpora pro návrhové vzory	Není	Znovupoužitelnost zásuvných modulů pro jednotlivé platformy	Návrhové vzory,
18. Odezva , kvalita a stabilita	Rychlé provádění transformací i spouštění aplikací, drobné chyby v IDE, nekvalitní nápověda	Velmi složité ovládání modelovacího nástroje, kompilátor nebylo možné otestovat	Proti ostatním nástrojům pomalejší provádění transformací, stabilní	Rychlé provádění transformací i spouštění aplikací, velmi kvalitní IDE díky integraci s Eclipse
19. Licence, cena, testovací verze	Nezjištěna, testovací verze	Nezjištěna, testovací verze modelovacího nástroje	Zdarma – Open Source	Nejméně 6 800 EUR, testovací verze

Tabulka 5-1 Vlastnosti zkoumaných nástrojů

6. Závěr

V práci jsem popsal strukturu Modelem řízené architekturu, shrnul potřebné standardy, prověřil možnosti použití v SOA a ukázal vlastnosti existujících nástrojů. Fungující a neustále se rozvíjející nástroje s podporou MDA jsou dle mého názoru důkazem, že Modelem řízená architektura je životaschopná a bude se v budoucnu dále rozvíjet. Modelem řízená architektura je tvůrci nástrojů podporována a poskytuje kvalitní strategii pro vývoj pomocí modelů. Otevřenou otázkou zůstává, zda MDA implikuje použití postupů OOAD.

6.1. Použití MDA v SOA

Na příkladech a konkrétních nástrojích jsem ukázal, že je možné použít Modelem řízenou architekturu pro modelování byznys procesů a vytváření jednotlivých služeb. Složitější dynamické chování zatím nelze popisovat pomocí modelů. Důvodem je absence vhodných metodik, modelovacích standardů a nástrojů pro tuto problémovou doménu. Není jasné, zda je bude možné vytvořit, aktivity ze strany výrobců nástrojů i standardizačních organizací v tomto směru existují.

6.2. Přínosy MDA při vývoji softwaru

Hlavními přínosy MDA a tedy důvodem pro její rozšíření v praxi jsou zvýšení kvality a produktivity díky automatickým transformacím modelů do kódu a zvýšení úrovně abstrakce, na které vývojář pracuje.

MDA také nabízí potenciál pro vývoj skutečně přenositelných aplikací. Vysoká úroveň abstrakce modelů a automatické transformace minimalizují závislost na konkrétní platformě. Při změně platformy stačí použít jinou sadu transformací. Pomocí vhodných transformací lze dobře oddělit infrastrukturní kód a byznys logiku, která je potom znovupoužitelná při změně platformy (pro případ, že ji není možné dostatečně podrobně zachytit v PIM modelu).

Dalším důležitým přínosem je kvalitní dokumentace v podobě aktualizovaných PIM a PSM modelů. Aktualizace nemusí být díky automatizaci transformací prováděny zpětně dle upraveného kódu, výhodnější je provádět je přímo v modelech.

6.3. Problémy použití MDA při vývoji softwaru

Nasazení MDA přináší i problémy a v některých oblastech MDA (zatím) aplikovat nelze.

Vývojáři musí být velmi zdatní v modelování. Bude nutné vývojáře dodatečně vyškolit a naučit novému stylu práce. Vývojáři si musí zvyknout na oddělení od kódu. Musí se naučit vytvářet a používat transformace. To přinese dodatečné náklady na zavedení MDA.

Nástroje zatím nemají dostatečnou podporu pro udržení stability datového modelu mezi verzemi aplikace. Vývoj větších aplikací je týmová práce, většina modelovacích nástrojů bohužel nedisponuje kvalitní podporou pro týmový vývoj modelů.

Pro některé oblasti nasazení zatím standardy chybí (SOA) nebo nejsou zralé (QVT). Dodavatelé nevytváří nástroje schopné vzájemné spolupráce, tj. nedodržují standardy nebo důležité standardy neimplementují. Díky tomu nenaleznou uplatnění některé výhody MDA a hrozí vznik závislosti na konkrétním dodavateli. Většina nástrojů je zatím omezena na modelování statické struktury aplikace a vygenerování infrastrukturních objektů, byznys logiku musí vývojář implementovat v programovacím jazyku.

Modelem řízenou architekturu nelze použít pro všechny typy projektů. Nevýhodná je zejména pro vývoj nízkoúrovňových komponent, pro malé triviální projekty nebo tam, kde je jádro aplikace tvořeno složitými algoritmy, nikoliv statickou strukturou.

6.4. Další směry rozvoje MDA

Zejména pro SOA a EAI (Enterprise Application Integration) chybí metodiky, standardy (jazyky a profily) a vhodné modelovací nástroje. Je potřeba, aby dozrály existující a byly vytvořeny nové standardy, metodiky a nástroje. Nástroje by mohly začít podporovat i jiné modelovací jazyky – jakékoliv jazyky definované pomocí MOF.

Dalším krokem k rozšíření MDA může být orientace nástrojů na modelování byznys artefaktů a vytváření jejich katalogů, jak je popsáno v knize [2]. Používání artefaktů by mohlo snížit potřebu „značkování" modelů pomocí profilů apod. Většina nástrojů může být také doplněna o komplexnější transformace (bez z nich jsou v praxi těžko použitelné). To otevře novou oblast trhu – dodávku komplexních transformací specialisty pro dané platformy (podobně jsou dnes dodávány softwarové komponenty a knihovny).

6.5. Přínos práce k řešené problematice

Práce uceleně popisuje strukturu MDA. Popisuje a kriticky hodnotí používané standardy. Na příkladech konkrétních nástrojů ukazuje možné aplikace, přínosy a problémy při vývoji informačních systémů podle MDA. Přehledně kategorizuje možnosti existujících nástrojů. Odhaluje oblasti, pro které je použití MDA méně vhodné, naznačuje možné směry dalšího rozvoje a klade některé fundamentální otázky.

7. Rejstřík obrázků

8. Rejstřík tabulek

9. Použitá literatura a další zdroje

[1] Anneke Kleppe, Jos Warmer, Wim Bast: MDA Explained, The Model Driven Architecture™: Practice and Promise, Addison-Wesley, 2003.

[2] Jim Arlow, Ila Neustadt: Enterprise Patterns and MDA, Bulding Better Sowftware with Archetype Patterns and UML, Addison-Wesley, 2004.

[3] Leon Starr: Executable UML, How To Build Class Models, Prentice Hall, 2002.

[4] Stephen J.Mellor, Marc J.Balcer: Executable UML, A Foundation for Model Driven Architecture, Addison-Wesley, 2002.

[5] Richard Hubert: Convergent Architecture, Building Model-Driven J2EE Systems with UML, Willey Computer Publishing, 2002.

[6] Jos Warmer, Anneke Kleppe: The Object Constraint Language Second Edition, Getting Your Models Ready for MDA, Addison-Wesley, 2003.

[7] Uwe Assman, Mehmet Aksit, Arend Rensink: Model Driven Architecture, European MDA Workshops: Foundations and Applications, Springer, 2005.

[8] UML OCL 2.0 Anapted Specification, OMG 03-10-14, 2003.

[9] Joseph Schmuller: Tlach Yourself UML in 24 Hours, Sams, 1999.

[10] Jiří Polák, Vojtěch Merunka, Antonín Carda: Umění systémového návrhu, Ob-

jektově orientovaná tvorba informačních systémů pomocí původní metody BORM, Grada Publishing, 2003.

[11] Alena Buchalcevová: Metodiky vývoje a údržby informačních systémů, Grada Publishing, 2005.

[12] Alena Buchalcevová: Model Driven Architecture jako nový přístup k vývoji i integraci aplikací, Katedra informačních technologií VŠE, 2003.

[13] Alan W.Brown, Simon K. Johnston, Grant Larsen, Jim Palistrant: SOA Developmnet Using the IBM Rational Software Development Platform: A Practical Guide, IBM, September 2005.

[14] Rakesh Radhakrishnan, Mike Wookey: Model Driven Architecture Enabling Service Oriented Architectures, Sun Micro Systems, March 2004.

[15] Zbyněk Vávra: Modelem řízená architektura, Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky, Katedra informačních technologií, 2004.

[16] Webové stránky konsorcia OMG, http://www.omg.org a zejména http://www.omg.org/mda.

[17] http://www.opengroup.org/cio/MDA-ADM/

[18] Richard Soley and the OMG Staff Strategy Group: Model Driven Architecture, OMG, 2000.

[19] MathML – Mathematical Markup Language, World Wide Web Consortium,

http://www.w3.org/TR/MathML2/, 2003.

[20] Joaquin Miller, Jishnu Mukerji: MDA Guide Version 1.0.1, OMG, 2003.

[21] Joaquin Miller, Jishnu Mukerji: Model Driven Architecture – Technical Pespective, OMG, 2001.

[22] Scott W. Ambler: The Object Primer, Third Edition, Agile Model-Driven Development with UML 2.0, Cambridge University Press, 2004.

[23] Philippe Kruchten: The Rational Unified Process: An Introduction, Addison-Wesley, 2003.

[24] Sascha Alexander: Nekonečný boj o UML 2.0, Computer World, 2003.

[25] Alexander Christoph: Graph Rewrite Systems for Software Design Transformations, Forschungszentrum Informatik (FZI), Germany.

[26] Octavian Patrascoiu: YATL: Yet Another Transformation Language, Computing Laboratory, University of Kent, United Kingdom.

[27] E.D.Willink: UMLX, A graphical transformation language for MDA, Thales Research and Technology Limited, 2003.

[28] ATL - The Atlas Transformation Language, http://www.sciences.univ-nantes.fr/lina/atl/

[29] MOF QVT final adopted specification, OMG, 2005.

[30] Webová stránka
http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[31] Webové stránky, manuály, návody a prezentace společnosti Kennedy Carter.

[32] Project Technology, Inc., Nucleus UML Suite, www.projtech.com.

[33] T. Bruckner: Řízení služeb informatiky v podmínkách outsourcingu. Disertační práce, VŠE, Praha, 2001.

[34] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Návrh programů pomocí vzorů, Stavební kameny objektově orientovaných programů, Grada Publishing, 2003.

[35] Keith Mantell: Model Driven Architecture in a Web services world, IBM developer Works, 2003.

[36] Jaroslav Král, Michal Žemlička: Implementation of Business Processes in Service-Oriented Systems, MFF UK, 2005.

[37] David S. Frankel, MDA, SOA and Technology Convergence, MDA Journal December 2003.

[38] Meta Object Facility IDL Language Mapping Specification,
version 2.0, OMG, 2006.

[39] Petr Smil: Vývoj metodik a metod objektové analýzy a designu, Softwarové noviny, leden 2000.

[40] Olaf Zimmermann, Pal Krogdahl, Clive Gee: Elements of Service-Oriented Analysis and Design, IBM Developer Works, June 2004.

[41] TOGAF/MDA Mapping, The Open Group White Paper, December 2005.

[42] Ali Arsanjani: Service-oriented modeling and architecture, IBM Developer Works, November 2004.

[43] Pat Helland: Data on the Outside Versus Data on the Inside, Microsoft Corporation, 2005.

[44] Stefan Tilkov: SOA and MDE, innoQ Deutschland GmbH, 2005.

[45] Samuel Kendall, Jim Waldo, Ann Wollrath and Geoff Wyant: Note on Distributed Computing, 1994.

[46] Werner Vogels: Web Services Are Not Distributed Objects, IEEE Internet Computing, Volume 7, Number 6, 2003.

[47] Radovan Janeček: Case Study of Model Driven Engineering, blog autora, 27th November, 2005.

[48] J2EE Patterns, Sun Java Center, January 14, 2002.

[60] Webové stránky, manuály, návody a prezentace společnosti Borland.

[61] Webové stránky, manuály, návody a prezentace společnosti IBM.

[62] Webové stránky, manuály, návody a prezentace společnosti Data Access Technologies.

[63] Webové stránky, manuály a návody k projektu AndroMDA.

[64] Webové stránky, manuály a návody k projektu openMDX.

[65] Webové stránky konsorcia OMG.

[66] Webové stránky, manuály, návody a prezentace společnosti Compuware.

[67] Webové stránky, manuály, návody a prezentace projektu Eclipse.

10. Seznam zkratek – doplnit při dalším čtení textu.

ASL	*Action Specification Language,* jazyk pro popis dynamického chování modelů v UML 2.0. Není definována jeho syntaxe, pouze metamodel.
BPEL	*Business Process Execution Language*, jazyk pro orchestraci služeb. Umožňuje ji popsat včetně rozhodovacích kroků a napojení na jednotlivé služby SOA komponent. Jeho varianta BPEL4WS (BPEL for Web Services) se používá pro orchestraci webových služeb.
BPMN	*Business Porcess Modeling Notation,* notace pro zápis byznys modelů, standard BPMI (Business Process Modeling Initiative) a OMG.
CASE	*Computer-aided design engineering,* počítačem podporovaný návrh software, CASE nástroje slouží pro podporu vývoje a údržby softwarových systémů.
CIM	*Computational Independent Model*, výpočetně nezávislý model, viz kapitola 2.3.1.
CORBA	*Common Object Request Broker Architecture*, standard OMG pro architekturu distribuovaných objetků.
CVS	*Current Version System,* software pro správu verzí, zejména zdrojových kódů.
CWM	*Common Warehouse Metamodel*, jazyk pro modelování dat a datových skladů odvozený od UML, standard OMG, viz kapitola 3.7.
DDL	*Data Definition Language,* jazyk pro popis definice dat, většinou SQL skript pro vytvoření struktury databáze.
DSL	*Domain Specific Languages,* označuje přístup, kdy je pro každou doménu vytvořen specifický jazyk. Ten může být definován v MOF.
DSM	*Domain Specific Modeling,* označuje přístup, kdy je pro každou doménu vytvořen model v existujícím jazyku, např. UML model nebo profil.
EAI	*Enterprise Application Integration,* integrace firemních informačních systémů.

ECO	*Enterprise Core Objects,* framework pro vytváření MDA aplikací od firmy Borland, viz kapitola 5.2.
EJB	*Enterprise JavaBeans,* Java API pro vytváření distribuovaných systémů.
EMF	*Eclipse Modeling Framework,* framework pro vytváření nástrojů pro práci s modely a transformace modelů do kódu.
ESB	*Enterprise Service Bus,* infrastruktura pro integraci firemních aplikací. Měla by umožňovat zasílání zpráv a jejich zabezpečení, transformace zpráv, dynamické směrování zpráv dle obsahu, asynchronní komunikaci apod.
GUI	*Graphics User Interface*, grafické uživatelské rozhraní.
J2EE	*Java 2 Platform, Enterprise Edition,* standard pro vývoj komponentových vícevrstvých aplikací, v poslední verzi označován zkratkou Java EE. Viz http://java.sun.com/javaee/.
JMI	*Java Metadata Interface*, rozhraní pro práci s MOF modely v Java, viz kapitola 3.2.1.
MDD	*Model Driven Development,* vývoj aplikací založený na modelech.
MOF	*Meta-Object Facility*, jazyk pro popis modelovacích jazyků, je v něm definováno UML a CWM, standard OMG, viz kapitola 3.2.
.NET	Framework pro vývoj aplikací od firmy Microsoft. Zahrnuje podporu pro distribuované a webové aplikace, komponenty pro uživatelského rozhraní a přístup k databázim apod. Viz http://www.microsoft.com/net/.
OASIS	*Organization for the Advancement of Structured Information Standards*, konsorcium, které vyvíjí standardy pro elektronické obchodování a webové služby.
OOAD	*Object Oriented Analysis and Design,* objektová analýza a návrh, zahrnuje objektové metodiky třetí generace - Booch, Object Modeling Technique, Shaelr/Mellor a další. Více viz např. [39].

OCL	*Object Constaint Language,* jazyk pro zápis omezení MOF modelů, součást standardu MOF, viz kapitola 3.8.
PIM	*Platform Independent Model,* platformově nezávislý model, viz kapitola 2.3.2.
PSM	*Platform Specific Model*, platformově specifický model, viz kapitola 2.3.3.
RAS	*Reuseable Asset Specification*, standard OMG definující způsob zabalení znovupoužitelných softwarových artefaktů. Je založený na UML. Hotový balíček obsahuje klasifikaci artefaktu, implementaci artefaktu včetně strukturovaných metadat, která ho definují a popisují, způsob použití artefaktu a související artefakty. Je důležitou součástí metodiky Asset-based Development (ABD) pro vývoj software.
RUP	*Rational Unified Process,* metodika pro řízení procesu vývoje softwarových systémů firmy Rational, dnes vlastněná IBM. Více informací viz např. [23].
SOA	*Service Oriented Architecture*, architektura orientovaná na služby, definice viz kapitola 4.
SOAP	*Simple Object Access Protocol*, protokol pro výměnu zpráv mezi aplikacemi založený na XML, standard W3C konsorcia.
TOGAF	*The Open Group Architecture Framework,* framework pro popis architektury firemních informačních systémů.
UDDI	*Universal Description, Discovery, and Integration*, specifikace od OASIS konsorcia pro publikování a vyhledávání informací o webových službách.
UML	*Unified Modeling Language,* jednotný jazyk pro modelování, standard OMG,
W3C	*World Wide Web Consortium*, organizace zabývající se standardy pro webové technologie, např. HTML, XML, SOAP, WSDL, URL.
WSDL	*Web Services Description Language*, jazyk pro popis rozhraní webových služeb založený na XML, standard W3C konsorcia.

XML	*Extensible Markup Language*, univerzální rozšiřitelný jazyk původně určený pro web, díky rozšiřitelnosti je používán pro výměnu dat ve většině oblastí informatiky, standard W3C.