VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

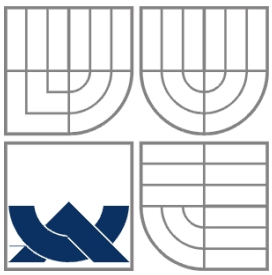
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MODELOVACÍ NÁSTROJ PRO GRAFICKÝ NÁVRH KOMPONENTOVÝCH SYSTÉMŮ

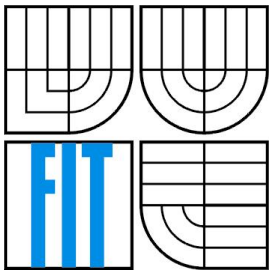
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. IVAN GÁL
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MODELOVACÍ NÁSTROJ PRO GRAFICKÝ NÁVRH KOMPONENTOVÝCH SYSTÉMŮ

A TOOL FOR MODELLING OF COMPONENT-BASED SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR
Bc. IVAN GÁL

VEDOUCÍ PRÁCE
SUPERVISOR
Mgr. MAREK RYCHLÝ

BRNO 2009

Abstrakt

Diplomová práce se zabývá komponentovým software, softwarovými rámci platformy Eclipse a vytvořením grafického editoru pro návrh komponentových systémů s využitím platformy Eclipse. Po úvodním představení koncepce UML diagramu komponent se věnuje přehledu problematiky komponentového software, pojmům jako je komponenta, modul, rozhraní a objekt. Dále se práce věnuje komponentovým technologiím od tří hlavním dodavatelů na poli komponentového software: OMG, Sun, Microsoft. Podstatná část práce se zabývá softwarovými rámci platformy Eclipse pro práci s meta-modely. Podrobně jsou popsány modely EMF, dále GEF a GMF. Hlavní část práce ukazuje návrh, implementaci a zhodnocení grafického editoru pro návrh komponentových systémů, přičemž je kladen důraz na srozumitelnost a přehlednost.

Klíčová slova

EMF, GMF, GEF, komponentový software, komponenta, diagram komponent, UML, Eclipse, rozhraní, meta-model, model, CCM, EJB, COM

Abstract

This thesis deals with component software, software frameworks for the Eclipse platform and the creation of a graphical editor for designing component systems with the usage of the Eclipse platform. After introducing the conception of UML component diagram, it describes the overview of component software, components and component technologies of major players on ground of component software: OMG, Sun, Microsoft. A significant part is dedicated to software frameworks for the Eclipse platform for manipulating with meta models. EMF, GEF and GMF are described in more detail. The main part presents the design, implementation and evaluation of a graphical editor for designing component systems with emphasis on understandability and good arrangement.

Keywords

EMF, GMF, GEF, component software, component, component diagram, UML, Eclipse, interface, meta model, model, CCM, EJB, COM

Citace

Gál Ivan: Modelovací nástroj pro grafický návrh komponentových systémů. Brno, 2009, diplomová práce, FIT VUT v Brně.

Modelovací nástroj pro grafický návrh komponentových systémů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Mgr. Marka Rychlého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ivan Gál
22. května 2009

Poděkování

Za pomoc, podněty, trpělivé konzultace a odborné rady děkuji Mgr. Markovi Rychlému.

Obsah

1 Úvod ... 3
1.1 Členenie práce ... 4
2 UML diagram komponentov ... 5
2.1 Účel ... 5
2.2 Notácia ... 5
2.2.1 Modelovanie vzťahov medzi komponentmi ... 7
2.2.2 Podsystémy ... 7
2.2.3 Znázornenie vnútornej štruktúry komponentu ... 7
3 Vývoj založený na komponentoch ... 9
3.1 Komponentový softvér ... 9
3.1.1 História ... 9
3.1.2 Komponenty, znovu použitie a komponentové riešenie ... 9
3.1.3 Vlastnosti komponentov a objektov ... 10
3.1.4 Rozhrania verzus kontextové závislosti ... 12
3.2 Komponentové technológie ... 13
3.2.1 OMG ... 13
3.2.2 Sun ... 15
3.2.3 Microsoft ... 17
3.2.4 Zhrnutie komponentových technológií ... 20
4 Softvérové rámce platformy Eclipse ... 21
4.1 EMF ... 21
4.1.1 Úvod ... 21
4.1.2 Definovanie modelu ... 22
4.1.3 Generovanie kódu ... 24
4.1.4 EMF framework ... 26
4.1.5 EMF a modelovacie štandardy ... 28
4.1.6 Editovanie modelu s EMF.Edit ... 28
4.2 GEF ... 31
4.2.1 Úvod do Draw2D ... 32
4.2.2 GEF framework ... 32
4.3 GMF ... 34
4.3.1 Komponent vykonávania (runtime component) ... 34
4.3.2 Odľahčené prostredie vykonávania ... 34
5 Špecifikácia požiadaviek ... 35
6 Návrh aplikácie ... 36
6.1 Životný cyklus vývoja založeného na GMF ... 36
6.1.1 Definícia modelu domény ... 36
6.1.2 Grafická definícia ... 36
6.1.3 Definícia nástrojov ... 38
6.1.4 Definícia mapovania ... 38
6.1.5 Generovanie kódu ... 38

6.1.6 Spustenie diagramu 38
6.2 Odľahčené prostredie vykonávania 38
6.3 Návrh meta-modelu 40
7 Implementácia 41
7.1 Ecore meta-model 41
7.2 Grafická definícia 42
7.3 Definícia nástrojov 44
7.4 Definícia mapovania 45
7.5 Model generovania a vygenerovanie editora 47
7.6 Pokročilá úprava editora 48
7.6.1 Obmedzenia väzieb 48
7.6.2 Automatizované vytváranie zástupcov (proxy) 50
8 Prípadová štúdia 53
8.1 Nový prístup k diagramu komponentov 54
9 Zhodnotenie a ďalší smer vývoja systému 55
9.1 Zhodnotenie 55
9.2 Ďalší smer vývoja systému 55
10 Záver 57
Literatúra 58
Zoznam príloh 60
Príloha 1. Návod na inštaláciu 61
Príloha 2. Používateľská príručka 62
Príloha 3. Serializovaný diagram 64
Príloha 4. Obsah CD 65

1 Úvod

Softvérové inžinierstvo založené na komponentoch (CBSE – Component based software engineering) sa týka vývoja systémov zo softvérových komponentov, vývoja komponentov a správu a zlepšovanie systému prostriedkami výmeny alebo úpravy komponentov.

Budovanie systémov z komponentov a vytváranie komponentov pre rôzne systémy vyžaduje ustálené metodológie a procesy nielen vo vzťahu k fázam vývoja a údržby, ale taktiež vo vzťahu k celkovému životnému cyklu komponentov a systému vrátane organizačných, marketingových, legálnych a iných aspektov. Okrem cieľov ako špecifikácia komponentov, kompozícia a vývoj komponentovej technológie, ktoré sú špecifické pre CBSE, existuje množstvo disciplín softvérového inžinierstva a procesov, ktoré vyžadujú, aby boli metodológie špecializované pre aplikáciu vo vývoji založenom na komponentoch. [6]

Pokrok softvéru a vývoja systémov v dnešnej dobe veľmi záleží na úspešnom nasadení CBSE. Rastúci záujem o CBSE sa odráža vo veľkom počte seminárov a konferencií týkajúcich sa CBSE.

CBSE rozlišuje proces „vývoja komponentov“ a „vývoja systémov založených na komponentoch“. Komponenty sú vyvíjané za účelom použitia a znovu použitia v mnohých aplikáciách. Vývoj systémov s komponentmi sa zameriava na identifikáciu znovu použiteľných entít a vzťahov medzi nimi počínajúc od systémových požiadaviek a dostupnosti už existujúcich komponentov.

V súčasnosti je mnoho otvorených problémov v oblasti CBSE. Otázkou je, ako môžu byť tieto problémy vyriešené? Riešenia sa môžu vzťahovať k otázke, koľko skúseností zo softvérového inžinierstva a systémového inžinierstva založeného na počítačoch môže byť využité v CBSE? Môžeme použiť rovnaké alebo podobné metódy? Do akej miery môžeme nájsť vzťahy medzi systémovými komponentmi a softvérovými komponentmi? [6]

Za posledné roky nastali významné pokroky v softvéri, ktorý sľubuje spôsobiť prevrat v tom, akým spôsobom budú softvérové aplikácie a nástroje koncipované, navrhované, modelované, konštruované, integrované, monitorované a riadené prostredníctvom použitia „open source“ technológií (ako napríklad Apache a Eclipse) a otvorených štandardov (ako napríklad Java, Java EE, UML, XML a MDA).[17] Hoci individuálne nástroje (kompilátor, modelovací nástroj, testovací nástroj, biznis integračný nástroj) sú stále dôležité, dôležitejšie však bude, ako budú tieto nástroje navzájom spolupracovať. Mali by umožniť rýchly návrh, vývoj, správu a monitorovanie aplikácií s pokračujúcim tlakom na dosahovanie lepších výsledkov za použitia nižších prostriedkov. V tomto prostredí je potrebná integračná technológia, ktorá umožňuje spoločné použitie týchto nástrojov a aplikácií pre zlepšenie procesu vývoja softvéru. V tejto práci sa budeme venovať viacerým takýmto integračným technológiám, ktoré unifikujú „open source“ a otvorené štandardy: Eclipse Modeling Framework, Graphical Editing Framework, Eclipse Graphical Modeling Framework, ktoré unifikujú Java, XML a UML technológie, takým spôsobom, že môžu byť spoločne použité na vybudovanie lepších integrovaných softvérových nástrojov.

1.1 Členenie práce

Práca sa skladá z dvoch základných častí: prvou je teoretický základ potrebný na dostatočné oboznámenie sa s problematikou komponentového softvéru a možnosťami UML, ktoré pre túto oblasť ponúka, pričom druhá časť je zameraná prakticky, venuje sa konkrétnym rámcom platformy Eclipse, na ktorých bude stavať výsledný grafický editor a popisuje návrh, implementáciu a zhodnotenie aplikácie pre návrh komponentových systémov.

Práca je rozčlenená do deviatich kapitol. Prvá kapitola sa venuje uvedeniu do problematiky softvérového inžinierstva založeného na komponentoch, snaží sa načrtnúť, aké problémy sú v súčasnosti spojené s týmto prístupom. Obsahuje aj úvod do Eclipse platformy - ako „open source“ technológie - spolu s kombináciou s otvorenými štandardmi. Zároveň predstavuje jednotlivé kapitoly práce. Druhá kapitola predstavuje komponentové diagramy, ich štruktúru a modelovacie možnosti, ktoré poskytujú. Tretia kapitola sa venuje opisu komponentového softvéru a oblastí s ním spojených, viacerými spôsobmi definuje pojem komponent a opisuje komponentové technológie troch hlavných hráčov na poli komponentového softvéru: OMG so svojím CCM, Sun s technológiou EJB a Microsoft s COM modelom. Štvrtá kapitola opisuje tri softvérové rámce platformy Eclipse pre prácu s meta-modelmi: EMF, GEF a GMF. Piata kapitola obsahuje špecifikáciu požiadaviek kladených na výsledný editor a šiesta kapitola obsahuje popis životného cyklu vývoja založeného na GMF, je v nej navrhnutý editor modelov a jeho podstatná časť: EMF Ecore meta-model. Siedma kapitola predstavuje opis procesu implementácie výsledného editora. Ôsma kapitola predstavuje prípadovú štúdiu použitia editora a diskutuje rozdiel medzi klasickým prístupom UML komponentového diagramu a našim navrhovaným prístupom. Deviata kapitola popisuje dosiahnuté výsledky a rovnako diskutuje možné rozšírenia editora do budúcnosti. Deviata kapitola je uzavretím diplomovej práce, rekapituluje jej kľúčové časti a popisuje prínos tejto práce.

2 UML diagram komponentov

V tejto kapitole predstavíme UML diagram komponentov, jeho hlavný účel, používanú notáciu a modelovanie zložitejších vzťahov.

2.1 Účel

Hlavným účelom diagramu komponentov je ukázať štrukturálne vzťahy medzi komponentmi systému. V komponentovom návrhu poskytujú diagramy komponentov prirodzený nástroj pre počiatočné modelovanie riešenia.

V UML 1.1 komponent reprezentoval implementačné prvky, ako napríklad súbory a spustiteľné súbory. Avšak táto vlastnosť je v konflikte s bežným použitím pojmu komponent, ktorým máme na mysli prvky, ako sú napríklad COM komponenty (viď kapitola 3.2.3.2). UML 2 oficiálne mení význam komponentového návrhu, v UML 2 sa za komponenty považujú samostatné, zapuzdrené jednotky v rámci systému alebo podsystému, ktorý poskytuje jedno alebo viacej rozhraní. I keď to UML 2 špecifikácia prísne nešpecifikuje, komponenty sú prísne logické konštrukcie vo fáze návrhu systému. Cieľom je možnosť jednoduchého znovu použitia alebo nahradenia rôznej implementácie komponentu v našom návrhu, pretože komponent zapuzdruje správanie a implementuje dané rozhrania. [2]

Komponenty sa používajú na organizáciu systému do znovu použiteľných a zameniteľných častí. Diagram komponentov modeluje komponenty v systéme a tak vytvára časť pohľadu vývoja. Pohľad vývoja opisuje, ako sú organizované časti systému do modulov a komponentov a pomáha pri spracovávaní architektúry systému do vrstiev.

Komponent je zapuzdrená, znovu použiteľná a zameniteľná časť softvéru. Komponenty si môžeme predstaviť ako stavebné bloky, ktoré skombinujeme, aby navzájom sedeli a vytvorili softvér. Komponenty sú voľne previazané, aby zmena komponentu neovplyvnila zvyšok systému. Na podporu voľných väzieb a zapuzdrenia sú komponenty sprístupňované cez rozhrania. Komponenty vzájomne spolupracujú cez poskytované a požadované rozhrania, aby bola zabezpečená kontrola závislostí medzi komponentmi a rovnako aby bola umožnená ich zameniteľnosť. [1]

Poskytované rozhranie komponentu je rozhranie, ktoré komponent realizuje. Iné komponenty a triedy spolupracujú s komponentom cez jeho poskytované rozhranie. Poskytované rozhranie komponentu opisuje služby poskytované komponentom.

Požadované rozhranie komponentu je rozhranie, ktoré komponent potrebuje k tomu, aby mohol fungovať. Presnejšie, komponent potrebuje inú triedu alebo komponent, ktorý realizuje rozhranie. Avšak, aby bola zachovaná voľná väzba, pristupuje k triede alebo komponentu cez požadované rozhranie. Požadované rozhranie definuje služby, ktoré bude komponent potrebovať.

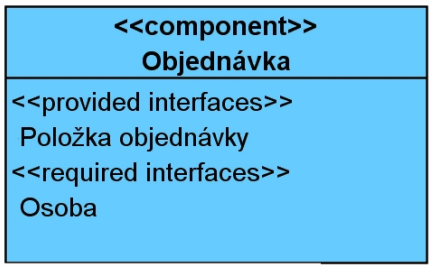
2.2 Notácia

Notácia komponentu je veľmi podobná triede v diagrame tried. Komponent je v podstate len špecializovaná verzia triedy, čo znamená, že pravidlá notácie pre triedy platia aj pre komponent.

Komponent sa znázorňuje ako obdĺžnik s názvom komponentu a stereotypom v podobe textu a/alebo ikony, ako je znázornené na obrázku 1.

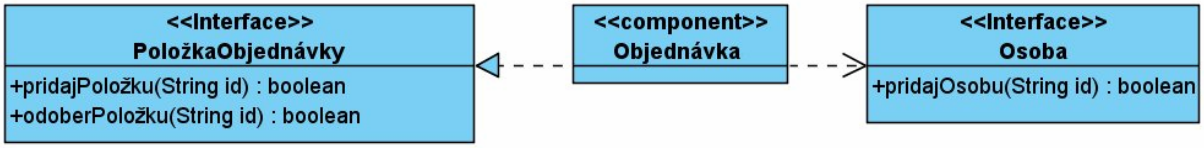
Obrázok 1. Znázornenie komponentu v UML 2.0

Poskytované rozhrania reprezentujú formálny kontrakt služieb, ktoré komponent poskytuje svojim klientom. Rozhrania môže znázorniť tromi spôsobmi.



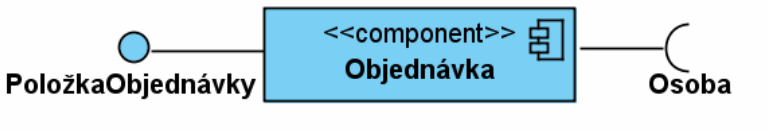
Obrázok 2. Prvý spôsob znázornenia rozhraní

Prvou možnosťou znázornenou na obrázku 2 je zápis rozhraní priamo do komponentu, pričom rozhrania sú oddelené zvislou čiarou.



Obrázok 3. Druhý spôsob znázornenia rozhraní

Druhou možnosťou znázornenou na obrázku 3 je notácia triedy so stereotypom. Ak komponent realizuje rozhranie, znázorní sa to pomocou šípky realizácie od komponentu k rozhraniu. Ak komponent požaduje rozhranie, zakreslí sa to pomocou šípky závislosti od komponentu k rozhraniu.



Obrázok 4. Tretí spôsob znázornenia rozhraní

Poslednou možnosťou znázornenou na obrázku 4 je pripojenie symbolov rozhraní ku komponentu. Symbol rozhrania s kruhom na konci reprezentuje poskytované rozhranie. Symbol rozhrania s polkruhom na konci reprezentuje požadované rozhranie.

2.2.1 Modelovanie vzťahov medzi komponentmi

Keď znázorňujeme vzťah komponentu s ostatnými komponentmi, musíme symboly rozhrania komponentov prepojiť pomocou šípky závislosti, pričom šípka smeruje od požadovaného rozhrania k poskytovanému rozhraniu. Avšak UML nástroj nám môže poskytovať zjednodušené prepojenie týchto symbolov rozhraní. Aj keď sa to zdá byť zbytočné, všetky rozhrania musia byť pomenované. Rozdielne pomenovanie požadovaného a poskytovaného rozhrania nastane napríklad, keď jeden komponent poskytuje rozhranie, ktoré je podtriedou menšieho požadovaného rozhrania.

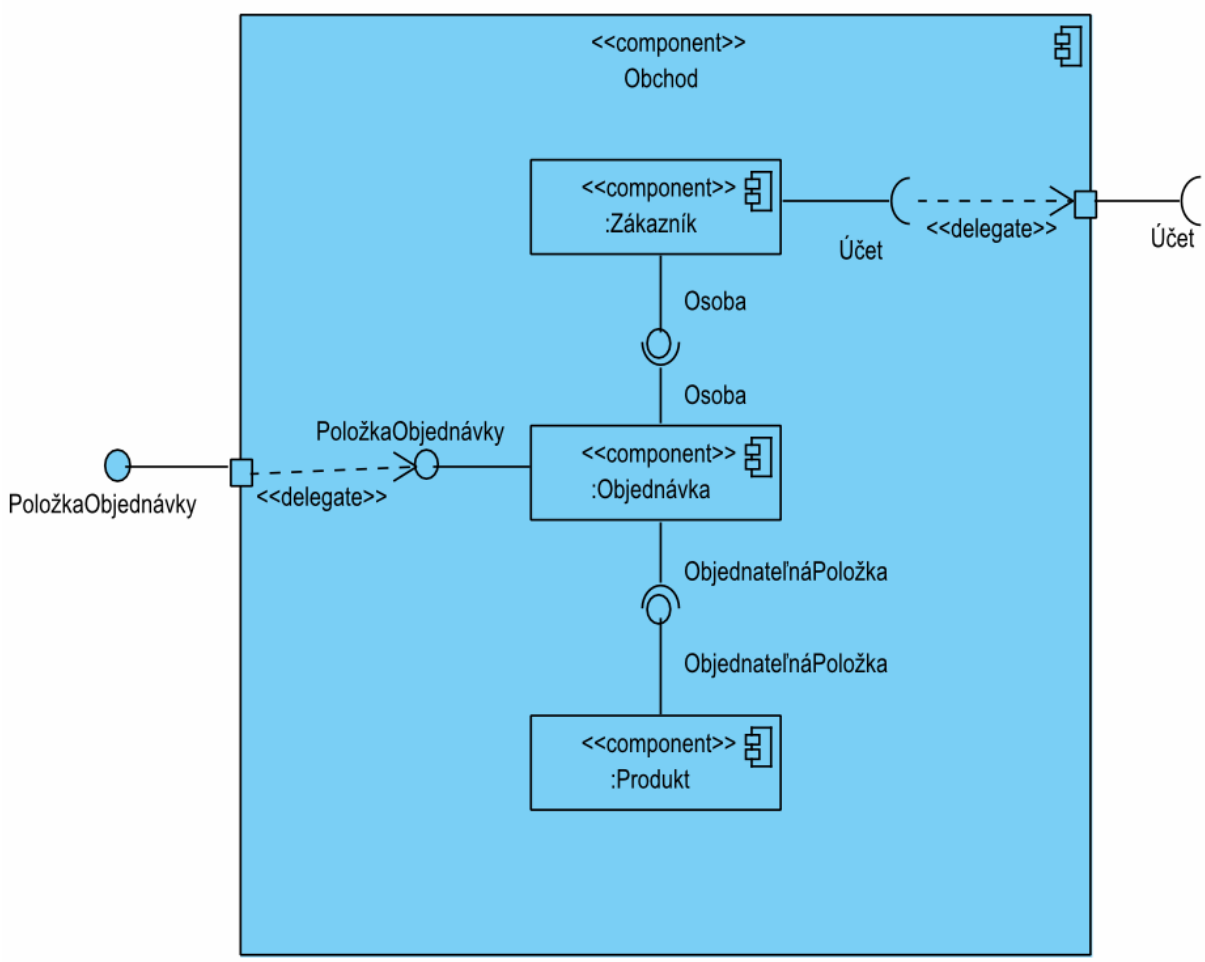
2.2.2 Podsystémy

UML 2 špecifikuje podsystém ako špecializovaný prípad komponentu s označením stereotypu <<subsystem>>. UML 2 špecifikácia definuje rozdiel medzi podsystémom a komponentom dosť vágne. V UML 1.x bol podsystém považovaný za balík. Táto notácia bola pre mnohých používateľov zavádzajúca, preto UML 2 zaviedla podsystém ako špecializovaný komponent. Rozdiel v použití týchto dvoch konštrukcií závisí na použitej metodológii. Použitím stereotypu <<subsystem>> môžeme ukázať, že komponent je podsystém veľmi veľkého systému. Je dobré si rezervovať tento stereotyp pre najväčšie časti celkového systému.

2.2.3 Znázornenie vnútornej štruktúry komponentu

Podobne ako triedy i komponenty môžu obsahovať porty a vnútornú štruktúru. Na znázornenie vnútornej štruktúry komponentu umiestnime do komponentu jeho vnútorné entity do časti s menom komponentu. Dôležitou súčasťou zloženého komponentu je port, ktorý je znázornený ako štvorec na okraji komponentu. Port poskytuje spôsob, akým sa vzťahujú poskytované a požadované rozhrania komponentu k jeho vnútorným častiam. Použitím portu je možné oddeliť vnútorné časti komponentu od vonkajších entít. Na obrázku 5 port PoložkaObjednávky deleguje rozhranie PoložkaObjednávky komponentu Objednávka pre spracovanie. Podobne vnútorné požadované rozhranie Účet komponentu Zákazník je delegované na port požadovaného rozhrania Účet komponentu Obchod. Pripojením na port Účet si môžu vnútorné časti komponentu Obchod vytvoriť lokálneho zástupcu nejakej vonkajšej entity, ktorá implementuje rozhranie portu. Keď modelujeme vzťah závislosti medzi portom a rozhraním, požadujúce rozhranie sa bude starať o všetku logiku spracovávanie v čase vykonávania. Avšak pre zložený komponent je rovnako prípustné, aby obsahoval svoju vlastnú logiku spracovávania namiesto výlučného delegovania spracovávania na požadujúce rozhranie. Požadované rozhranie Účet bude implementované komponentom mimo komponentu Obchod.

Komponenty majú vlastné konštrukcie pri znázornení portov a vnútornej štruktúry. Nazývajú sa prepojenia delegácie a prepojenia zostavovania. Používajú sa na znázornenie toho, akým spôsobom zodpovedajú rozhrania komponentu vnútorným častiam a ako vnútorné časti navzájom spolupracujú. [1]



Obrázok 5. Znázornenie vnútornej štruktúry komponentu

Poskytované rozhranie komponentu môže byť realizované jednou z vnútorných častí, rovnako ako poskytované rozhranie komponentu môže byť požadované jednou z jeho častí. V týchto prípadoch sa používa prepojenie delegácie na znázornenie, že vnútorné časti realizujú alebo používajú rozhrania komponentu. Prepojenia delegácie sa znázorňujú šípkou v smere toku komunikácie a spájajú port pripojený k rozhraniu vnútornej entity. Na obrázku 5 sú to šípky označené stereotypom <<delegate>>. Prepojenia delegácie si môžeme predstaviť tak, že port reprezentuje otvor do komponenty, skrze ktorého prechádza komunikácia a prepojenie delegácie znázorňujú smer tejto komunikácie. Takže prepojenie delegácie smerujúce od portu k internej entite reprezentuje správu, ktorá bude zaslaná tejto entite na spracovanie.

Prepojenia zostavovania znázorňujú, že komponent požaduje rozhranie, ktoré iný komponent poskytuje. Prepojenia zostavovania spájajú navzájom kruh a polkruh, ktoré reprezentujú poskytované a požadované rozhranie. Na obrázku 5 je toto prepojenie použité medzi komponentmi Objednávka a Zákazník a medzi komponentmi Objednávka a Produkt. Prepojenia zostavovania sú špeciálnym druhom prepojenia, ktoré sú definované pre použitie v prípade, keď znázorňujeme zloženú štruktúru komponentu. Je dôležité poznamenať, že používajú notáciu menoRole:menoTriedy, ktorá sa používa v zložených štruktúrach. Avšak prepojenia zostavovania sa vo všeobecnosti používajú aj ako možnosť prezentácie pre znázornenie závislosti komponentov cez rozhrania, ako už bolo uvedené.

3 Vývoj založený na komponentoch

Táto kapitola sa venuje popisu komponentového softvéru a komponentových systémov, pojmom ako komponent, modul, rozhranie, objekt a komponentovým technológiám od troch hlavných dodávateľov na poli komponentového softvéru: OMG, Sun, Microsoft.

3.1 Komponentový softvér

V tejto kapitole sa podrobnejšie pozrieme na problematiku komponentového softvéru, základné problémy pri návrhu komponentového softvéru a kľúčové vlastnosti komponentov. Popíšeme hlavné rozdiely medzi objektmi a komponentmi a rovnako sa budeme venovať problematike kontextových závislostí.

3.1.1 História

Je zrejmé, že súhra technológie a stratégií trhu pomáha komponentom dosiahnuť ich dlho očakávanú rolu. Prvý prístup, ktorý úspešne vytvoril významné trhové prostredie, prišiel v roku 1992 s programovacím jazykom Visual Basic od firmy Microsoft a jeho komponentmi (VBX). V podnikovej oblasti nasledovala CORBA 2.0 od OMG v roku 1995. Rastúca popularita distribúcie systémov a internetu viedli k ďalšiemu vývoju, ako napríklad DCOM a ActiveX od Microsoftu, prípadne Java a JavaBeans (Java komponentový štandard) od Sunu.

Dnes existujú tri hlavné skupiny v oblasti komponentového softvéru. OMG (Object Management Group) s jej štandardmi založenými na CORBA, ktorá vstúpila na trh z podnikovej perspektívy. Microsoft s jeho štandardmi založenými na COM, ktorý vstúpil na trh s perspektívy stolných počítačov. Sun s jeho štandardmi založenými na Jave, ktorý vstúpil na trh z internetovej perspektívy. Avšak všetky tri spoločnosti sa snažia tieto tri možnosti spojiť a poskytovať vo svojich produktoch.

3.1.2 Komponenty, znovu použitie a komponentové riešenie

Jedna vec, ktorú môžete s istotou tvrdiť, je, že komponenty slúžia na kompozíciu.

Softvérové komponenty umožňujú praktické znovu použitie softvérových častí a rozloženie investícií do viacerých aplikácií. Presnejšie, softvérové komponenty sú spustiteľné, nezávisle vyrobené a nasadené jednotky, ktoré vzájomne spolupracujú za účelom vytvorenia funkčného systému. Nezávislosť a spustiteľná forma je nevyhnutná, aby sa umožnila robustná integrácia a viacero nezávislých výrobcov. [5]

Znovu použitie je veľmi široký pojem pokrývajúci všeobecný koncept prínosu znovu použiteľnosti. Takým prínosom môžu byť ľubovoľné popisy zachytávajúce výsledky návrhu. Popisy bežne závisia na iných, detailnejších a špecializovanejších popisoch. Na to, aby sme vytvorili prínos znovu použiteľnosti, nestačí začať s jednoliatym návrhom celkového riešenia a potom ho rozdeliť na menšie časti. Pravdepodobné výhody tohto prístupu sú minimálne. Namiesto toho musia byť popisy starostlivo zovšeobecnené, aby bolo umožnené znovu použitie v dostatočnom počte odlišných kontextov. Avšak musíme sa vyhnúť prílišnému zovšeobecneniu, aby sme zachovali popisy

dostatočne voľné a odľahčené, aby zostala aktuálna znovu použiteľnosť praktická. Popisy v tomto zmysle sa niekedy nazývajú komponentmi. [5]

Softvérový komponent je to, čo je aktuálne nasadené – ako izolovateľná časť systému – v komponentovom prístupe. Na rozdiel od častých tvrdení sa objekty takmer nikdy nepredávajú, nenakupujú ani nenasadzujú. Jednotka nasadenia je skôr niečo statické, ako napríklad trieda alebo skôr skupina tried, ktoré sú skompilované a zlinkované do balíčku. Objekty, ktoré logicky formujú časti inštancií komponentu, sú vytvárané podľa potreby na základe tried, ktoré boli nasadené s komponentom. Takisto i komponent môže byť jediná trieda, avšak je pravdepodobnejšie, že sa skladá z kolekcií tried niekedy nazývaných modul. Z komponentov ako celku preto nie sú bežne vytvárané inštancie. Takisto môže komponent používať úplne odlišnú technológiu implementácie, ako napríklad čisté funkcie alebo jazyk symbolických adries a vôbec nemusí zvnútra vyzerať ako objektovo-orientovaný. [5]

Aby mohli byť komponenty nezávisle nasadzované, musia byť ich vzájomné závislosti a granularita od počiatku kontrolované. Je zrejmé, že pre obrovské systémy sú komponenty významným krokom vpred oproti objektom, prípadne triedam. Avšak to neznamená, že objektom sa vyhýbame. Naopak, ak je objektová technológia použitá pozorne, tak je pravdepodobne najlepším spôsobom, ako realizovať komponentovú technológiu.

Výstavba nových riešení pomocou kombinácie zakúpených a vyrobených komponentov zlepšuje kvalitu, podporuje rýchly vývoj a vedie ku kratšiemu času uvedenia na trh. Rovnako je možné dosiahnuť rýchlu adaptáciu na meniace sa požiadavky prostredníctvom investícií len do kľúčových zmien riešenia založeného na komponentoch a zároveň sa vyhneme potrebe významnej zmeny celého riešenia.

Komponentový softvér takisto ukončuje starý problém masívnych cyklov aktualizácií. Tradičné plne integrované riešenia vyžadovali periodickú aktualizáciu. Typicky to bol bolestivý proces migrácie starej databázy, zaručenia budúcej kompatibility, znovu zaškolenia zamestnancov, zakúpenia výkonnejšieho hardvéru a pod. V komponentovo založenom riešení revolúciu nahradzuje evolúcia a individuálna aktualizácia komponentov podľa potreby dovoľuje hladšie operácie. Očividne to vyžaduje iný prístup manažovania služieb, ale potenciálne zisky sú obrovské.

3.1.3 Vlastnosti komponentov a objektov

Táto časť definuje základné vlastnosti objektov a komponentov a popisuje podstatné rozdiely medzi nimi.

3.1.3.1 Objekt

Pojmy, ako možnosť vytvárania inštancií, identita a zapuzdrenie vedú k pojmu objekt. Na rozdiel od vlastností charakterizujúcich komponenty, charakteristické vlastnosti objektov sú nasledovné:

- je jednotkou možnosti vytvárania inštancie, má jedinečnú identitu
- môže mať stav, ktorý môže byť zvonku pozorovateľný
- zapuzdruje svoj stav a správanie. [5]

Pretože je objekt jednotkou možnosti vytvárania inštancie, nemôže byť z objektu čiastočne vytvorená inštancia. Keďže objekt má individuálny stav, rovnako má jednoznačnú identitu, ktorá postačuje na identifikáciu objektu i napriek zmenám stavu počas celého života objektu. Uveďme si

príklad o sekere George Washingtona. Mala päť nových porísk a štyri nové hlavy, ale stále to bola sekera George Washingtona. Toto je dobrým príkladom objektu z reálneho života, z ktorého nezostalo nič len abstraktná identita v čase.

3.1.3.2 Komponent

Charakteristické vlastnosti komponentu sú nasledovné:

- je jednotkou nezávislého nasadenia
- je jednotkou kompozície treťou stranou
- nemá zvonku pozorovateľný stav. [5]

Tieto vlastnosti majú viacero dôsledkov. Aby mohol byť komponent nezávisle nasadzovaný, je rovnako potrebné, aby bol oddelený od prostredia a od ostatných komponentov. Komponent preto zapuzdruje svoje jednotlivé vlastnosti. Takisto, keďže to je jednotka nasadenia, komponent nikdy nebude nasadzovaný čiastočne.

Aby bolo možné komponent skladať s ostatnými komponentmi treťou stranou, je potrebné, aby bol dostatočne samostatný. Takisto musí byť poskytovaný s jasnými špecifikáciami, čo požaduje a čo poskytuje. Inými slovami komponent potrebuje zapuzdrovať svoju implementáciu a spolupracovať s prostredím prostredníctvom dobre nadefinovaných rozhraní.

Komponent nemôže mať rovnako žiadny zvonku pozorovateľný stav. Požaduje sa, že komponent nemôže byť rozpoznateľný od kópií seba samého. Možnými výnimkami tohto pravidla sú atribúty, ktoré sa nepodieľajú na funkcionalite komponentu. Špecifické umožnenie pozorovateľného stavu dovoľuje v prípustných situáciách použitie tohto stavu, čo môže byť kľúčové pre výkon, pričom pozorovateľné správanie komponentu sa neovplyvní. Komponent môže použiť stav najmä na účely odkladacieho priestoru – cache. Cache je úložný priestor, ktorý môže byť odstránený bez akýchkoľvek dôsledkov okrem možného zníženia výkonu.

Komponent bude pravdepodobne pracovať cez objekty a preto bude obyčajne pozostávať z jednej alebo viacerých tried alebo nemenných prototypových objektov. Okrem toho môže obsahovať množinu nemenných objektov, ktoré zachytávajú vopred nastavený počiatočný stav a iné zdroje komponentu.

Avšak nie je potrebné, aby komponent obsahoval len triedy alebo dokonca neobsahoval triedy vôbec. Namiesto toho môže komponent obsahovať tradičné procedúry a dokonca mať globálne premenné (pokiaľ výsledný stav zostane nepozorovateľný) alebo môže byť realizovaný za použitia funkcionálneho programovacieho prístupu alebo za použitia jazyka symbolických adries alebo akéhokoľvek iného prístupu. Objekty vytvorené v komponente – presnejšie referencie na také objekty – môžu opustiť komponent a stať sa viditeľnými pre klientov komponentu, t.j. typicky pre ďalšie komponenty.

Z vyššie uvedených charakteristík môžeme sformulovať nasledujúcu definíciu: Softvérový komponent je jednotkou kompozície so zmluvne špecifikovanými rozhraniami a s iba explicitne kontextovými závislosťami. Softvérový komponent môže byť nasadzovaný nezávisle a je podstatou kompozície tretími stranami. [5]

Definícia pokrýva charakteristické vlastnosti komponentov diskutovaných doteraz. Skladá sa z technickej časti s aspektmi ako nezávislosť, zmluvné rozhrania a kompozícia. Takisto obsahuje časť

týkajúcu sa trhu s aspektmi ako tretie strany a nasadenie. Pre komponenty je nevyhnutné kombinácia technických a trhových aspektov.

Z modernejšieho uhlu pohľadu potrebuje táto definícia objasnenie. Popis nasadenia komponentu špecifikuje viac ako len závislosti a rozhrania, špecifikuje takisto, ako má byť komponent nasadený, ako majú byť z neho vytvárané inštancie a ako sa majú inštancie správať prostredníctvom zverejnených rozhraní. [5]

Ako príklad si uveďme komponent Queue, ktorý požaduje stabilný úložný priestor cez jedno rozhranie a poskytuje pridania a odobratie z radu cez ďalšie dve rozhrania. Popis komponentu vyjadruje, že keď je niečo pridané do radu prostredníctvom jedného rozhrania, môže byť cez druhé rozhranie z radu odobrané – vzájomný vzťah, ktorý nemôže individuálne špecifikácia rozhrania poskytovať. Komponentový popis takisto vyjadruje, že komponent, len čo je z neho vytvorená inštancia, môže byť používaný jedine pripojením na poskytovateľa, ktorý implementuje rozhranie stabilného úložného priestoru. Tento druhý pohľad prepájania komponentov, ktoré majú vzájomne si zodpovedajúce poskytované a požadované rozhrania, musí byť ovplyvnený pravidlami kompozície podporovaného komponentového modelu. Detaily nasadenia a inštalácie by mali byť podporované špecifickou komponentovou platformou. [5]

3.1.4 Rozhrania verzus kontextové závislosti

Na rozhrania komponentu sa môžeme pozrieť tak, že definujú prístupové body komponentu. Tieto body dovoľujú klientom komponentu, obyčajne komponentom samotným, pristupovať k službám poskytovaných komponentom. Komponent bude mať obyčajne viacero rozhraní vzťahujúcich sa k rôznym prístupovým bodom. Každý prístupový bod môže poskytovať rôznu službu, zaisťujúcu rôzne potreby klienta. Zdôraznením zmluvnej podstaty špecifikácií rozhrania je dôležité, pretože komponent a jeho klienti sú vyvíjaní vo vzájomnej neznalosti, takže je to práve zmluva, ktorá formuje spoločný priestor pre úspešnú spoluprácu. [5]

Okrem špecifikácie poskytovaných rozhraní vyššie uvedená definícia komponentov rovnako požaduje špecifikáciu ich potrieb. Inými slovami definícia vyžaduje špecifikáciu toho, čo prostredia nasadenia bude musieť poskytovať, aby mohli komponenty fungovať. Tieto potreby sú nazývané kontextové závislosti vzťahujúce sa na kontext kompozície a nasadenia. Zahŕňajú komponentový model, ktorý definuje pravidlá kompozície a komponentovú platformu, ktorá definuje pravidlá nasadzovania, inštalácie a aktivácie komponentov.

Je zrejmé, že komponent je najužitočnejší, keď poskytuje potrebnú množinu rozhraní a nemá žiadne obmedzenia kontextových závislostí – inými slovami, že dokáže pracovať vo všetkých komponentových prostrediach a pritom nevyžaduje rozhranie nad rámec tých, ktorých dostupnosť je zaručená komponentovými prostrediami. Technicky by mohol komponent byť dodaný s pripojeným všetkým potrebným softvérom, ale to by bolo v konflikte s účelom použitia komponentov. [5]

Namiesto vytvárania sebestačných komponentov so všetkými potrebnými časťami vstavanými, je možné sa rozhodnúť pre maximálne znovu použitie. Aby sme sa vyhli viacnásobne sa opakujúcej implementácii druhotných služieb vnútri komponentu, všetko, okrem hlavnej funkcionality, ktorú poskytuje komponent sám, môžeme delegovať na iný komponent. Objektovo-orientovaný návrh má k tomuto tendenciu a mnohé objektovo-orientované metodológie podporujú túto maximalizáciu znovu použitia.

Hoci má maximalizácia znovu použiteľnosti často uvádzané výhody, má takisto aj jednu podstatnú nevýhodu – prudký nárast kontextovej závislosti. Keby boli návrhy komponentov po uvoľnení naveky zmrazené a keby všetky prostredia nasadenia boli rovnaké, tak by to nespôsobovalo problémy. Avšak komponenty sa vyvíjajú a rôzne prostredia poskytujú rôzne konfigurácie a verzie, kontextové závislosti sa stávajú podstatnými. S každou pridanou kontextovou závislosťou sa stáva menej pravdepodobné, že komponent nájde klientov, ktorí dokážu uspokojiť požiadavky komponentu na prostredie. Takže, keď to zhrnieme: Maximalizácia znovu použiteľnosti minimalizuje použiteľnosť.

3.2 Komponentové technológie

V tejto kapitole diskutujeme komponentové technológie on troch najväčších poskytovateľov komponentových riešení na trhu, OMG, Sun a Microsoft. U každého z nich popíšeme riešenia, ktoré poskytuje v oblasti komponentového softvéru a základné rozdiely medzi nimi.

3.2.1 OMG

Táto časť popisuje prvého giganta na trhu komponentových systémov, konzorcium OMG a jeho komponentový model CCM.

3.2.1.1 História a súčasnosť

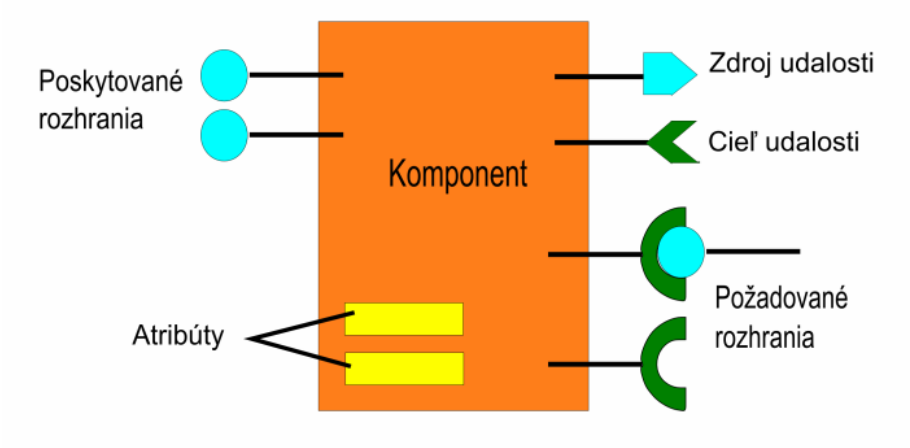
OMG (Object Management Group), založená v 1989, je doteraz najväčším konzorciom vo výpočtovom priemysle. OMG pracuje ako nezisková organizácia usilujúca sa o štandardizáciu širokej škály problémov. Pôvodné úsilia OMG sa koncentrovali na riešenia jedného základného problému – ako môžu distribuované objektovo-orientované systémy implementované v rôznych jazykoch a pracujúce na rôznych platformách vzájomne spolupracovať. Výsledkom bola Common Object Request Broker Architecture (CORBA) v jej počiatočnej verzii 1.1, uvoľnená v roku 1991, nasledovaná miernymi zlepšeniami vo verzii 1.2. Súčasná verzia je CORBA 3. Najväčším príspevkom v tejto verzii je CORBA Component Model (CCM).

3.2.1.2 CCM

CCM je logické rozšírenie Enterprise JavaBeans (EJB). Predstavuje niekoľko nových vlastností, zaručuje plne kompatibilné začlenenie exitujúcich EJB riešení a usiluje sa o udržanie pôvodného CORBA princípu, že zostane jazykovo a platformovo nezávislé. CCM je všeobecnejší ako EJB, pričom poskytuje štyri typy komponentov (komponenty služieb, komponenty sedenia, komponenty procesov a komponenty entít) na rozdiel od EJB, ktoré definujú len dva typy (Session Beans, Message Driven Beans).

3.2.1.2.1 CCM Kompozícia

CCM poskytuje abstrakciu entít, ktoré môžu poskytovať a prijímať služby cez dobre definované pomenované rozhrania nazývané porty.



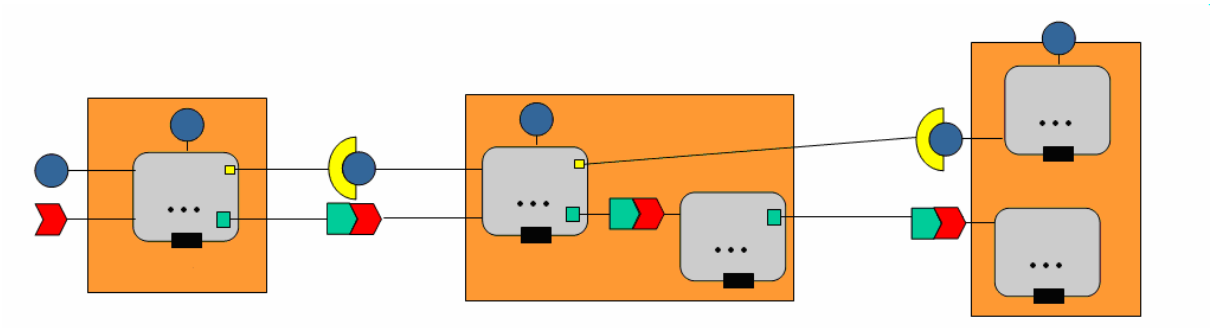
Obrázok 6. CCM komponent

Porty sa delia na päť podskupín: poskytované rozhranie, požadované rozhranie, zdroj udalosti, konzument udalosti a atribút. Poskytované a požadované rozhrania sú navzájom prepájané. Zdroje a ciele udalostí sú podobné, avšak namiesto vzájomného prepájania sú obe pripojené na kanály udalostí. Atribúty sú konfigurovateľné vlastnosti. Štruktúra komponentu je znázornená na obrázku 6.

Zdroje a konzumenti udalostí dovoľujú komponentom vzájomne spolupracovať bez tesného vzájomného previazania. Je to voľné spájanie podobne ako poskytuje návrhový vzor Observer [18]. Keď komponent deklaruje svoj záujem publikovať alebo vysielať udalosť, je to zdroj udalosti. Publikujúci zdroj je výlučný poskytovateľ udalosti pričom vysielajúci zdroj zdieľa kanál udalosti s ostatnými zdrojmi udalostí. Ostatné komponenty sa stávajú odberateľmi alebo konzumentmi týchto udalostí prostredníctvom deklarácie konzumenta udalostí. [7]

Na rozdiel od tradičného poňatia atribútov CORBA rozhraní, ktoré dovoľuje konfiguráciu hodnôt komponentov, CCM verzia atribútov dovoľuje operácie, ktoré pristupujú a modifikujú hodnoty za účelom vyvolania výnimiek. Toto je vhodná vlastnosť pre vyvolanie konfiguračnej výnimky, potom čo sa dokončila konfigurácia a k atribútu sa pristupovalo alebo bol pozmenený. [7]

CCM vytvára štandardné virtuálne hranice okolo implementácií aplikačných komponentov, ktoré vzájomne spolupracujú výlučne cez dobre definované rozhrania. Táto spolupráca kontajnerov a komponentov je znázornená na obrázku 7. CCM ďalej definuje štandardný mechanizmus kontajnerov potrebný pre vykonávanie komponentov na všeobecných komponentových serveroch. Špecifikuje infraštruktúru potrebnú na konfiguráciu a nasadenie komponentov v rámci distribuovaného systému. [8]



Obrázok 7. Prepojenie komponentov a komponentových kontajnerov

CCM obsahuje komponentový kontajner, kde môžu byť nasadzované softvérové komponenty. Komponenty zapuzdrujú aplikačnú biznis logiku. Kontajner ponúka skupinu služieb, ktoré môžu komponenty používať. Tieto služby okrem iného zahŕňajú notifikáciu, autentizáciu, perzistenciu a transakcie. Toto sú najpoužívanejšie služby, ktoré bežný distribuovaný systém požaduje. Tým, že presunieme implementáciu týchto služieb zo softvérových komponentov do komponentového kontajneru, sa zložitosť komponentov výrazne zníži.

Súčasťou CCM špecifikácie je aj CORBA kontajner pre umiestnenie EJB. Toto poskytuje mechanizmus pre spoluprácu medzi EJB a CCM komponentmi, ako bolo spomínané vyššie.

3.2.2 Sun

V tejto časti sa venujeme popisu komponentových technológií firmy Sun. Je tu spomenutých všetkých päť komponentových modelov, a to aplety, JavaBeans, EJB, servlety a aplikačné komponenty klienta.

3.2.2.1 Java EE

Java EE je považovaná za špecifikáciu, ktorá má byť implementovaná mnohými dodávateľmi. Avšak Sun poskytuje referenčnú implementáciu na demonštráciu a na objasnenie špecifikácie. Java EE architektúra je vo svojej podstate triedou komponentových modelov. Na klientskej strane sú to aplikačné komponenty, JavaBeans a aplety, na vrstve webového serveru sú to servlety a JSP stránky a na vrstve aplikačného serveru sú to EJB.

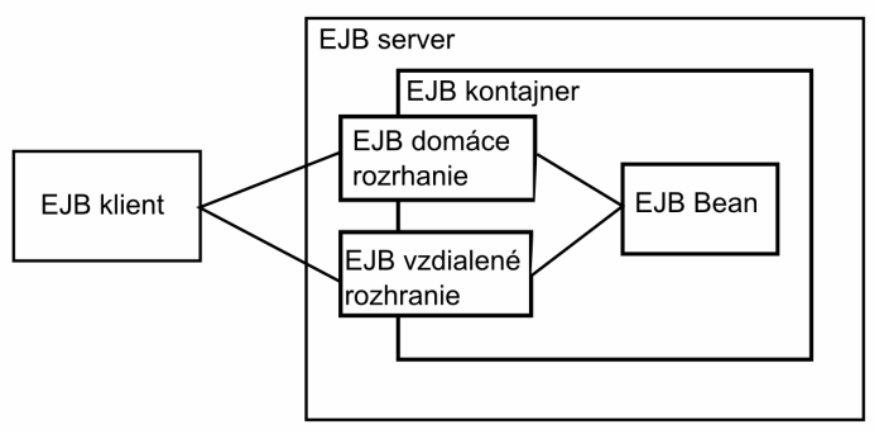
3.2.2.2 Java komponentové modely

Java definuje päť rôznych komponentových modelov. Sú to aplety, JavaBeans, EJB, servlety a aplikačné komponenty klienta.

Aplety boli prvým Java komponentovým modelom usilujúcim sa o sťahovateľné odľahčené komponenty, ktoré by rozšírili možnosti stránok zobrazených v prehliadači.

JavaBeans sa zamerali na podporu programovania orientovaného na spojenie a sú použiteľné na klientoch aj serveroch .

EJB sa zameriava na služby integrované v kontajneroch podporujúcich EJB komponenty. Kontajnerový model bol neskôr pridaný i do JavaBeans. JavaBeans kontajnery sú však len mechanizmus pre zaobalenie, pričom EJB kontajnery sú čiastočne riadené deklaratívnymi konštrukciami. Štruktúra EJB je znázornená na obrázku 8.



Obrázok 8. Štruktúra EJB komponentu

Servlety sú podobné apletom, ale pracujú na strane servera a sú to väčšinou odľahčené komponenty, z ktorých vytvára inštancie webový server, ktorý typicky spracováva webové stránky. Prislúchajúca technológia, JSP, sa môže použiť na deklaratívne definovanie webových stránok. JSP sú následne väčšinou skompilované na servlety.

Aplikačné komponenty klienta sú v podstate neobmedzené Java aplikácie, ktoré sídlia na klientovi. Komponent klienta používa JNDI menný kontext na prístup k vlastnostiam prostredia, EJB a zdrojom na Java EE serveroch. Zdroje môžu zahŕňať prístup k emailom alebo databázam.

3.2.2.2.1 Kompozícia u JavaBeans a EJB

Prístup JavaBeans ku kompozícii je skrze programovanie orientované na spojenie, čo sa niekedy nazýva nadväzovanie. JavaBeans komponenty môžu definovať zdroje a poslucháčov udalostí. Prepojením jedného poslucháča udalostí na iný zdroj udalostí môžu udalosti prúdiť. Ďalej je zahrnutá podpora hierarchických štruktúr kontajnerov. Takže namiesto umiestenia všetkých inštancií komponentov do jedného priestoru a spoliehaniu sa na priame spojenia jednotlivých komponentov kontajnery dovoľujú hierarchické vytváranie podsystémov. Ďalší prídavok, InfoBus, umožňuje flexibilné rozpájanie zdrojov a poslucháčov udalostí. Dosahuje to presmerovaním časti alebo celej komunikácie cez zbernicu, ktorá dovoľuje zachytenie a prácu so správami bez nutnosti spolupráce medzi zdrojom a poslucháčom udalosti a bez potreby znovu prepojenia. S JavaBeans infraštruktúrou zapuzdrenia a služieb a InfoBus sa môžu JavaBeans presunúť nad rámec kompozície orientovanej na spojenie, a to na sformovanie kontextovej kompozície a kompozície riadenej údajmi. [5]

EJB neposkytuje prostriedky pre programovanie orientované na spojenie (toto je jednou zo základných výhod CCM oproti EJB). Namiesto toho EJB komponenty nasledujú bežný model objektovo-orientovanej kompozície. [5] Napríklad, keď inštancia komponentu potrebuje čerstvú inštanciu iného komponentu, tak ju jednoducho vytvorí. Ak potrebuje komunikovať s inou inštanciu komponentu, jednoducho zavolá metódu. EJB nie je o systematickom zlepšovaní schopnosti kompozície komponentov prostredníctvom vytvárania spojení. Komponenty sú iba do tej miery schopné kompozície, do akej ich špecifický návrh vytvoril a všeobecná EJB architektúra ponúka len málo na jej zlepšenie alebo jej zabránenie.

3.2.3 Microsoft

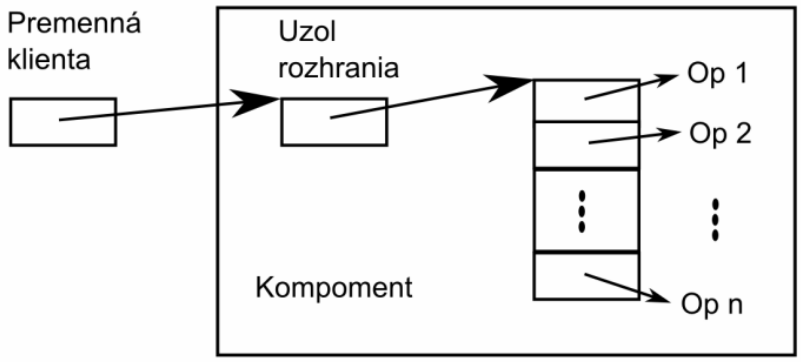
Posledným z troch poskytovateľov komponentových riešení je firma Microsoft. Táto kapitola podrobne diskutuje komponentový model COM a spôsoby kompozície týchto komponentov, t.j. obsiahnutie a agregácia.

3.2.3.1 Komponentová technológia

Komponentová technológia je u Microsoftu veľmi široká, patria sem Visual Basic Extension (VBX – nie objektovo-orientované komponenty), Object Linking and Embedding (OLE), OLE databázová konektivita (ODBC), ActiveX, Microsoft Transaction Server (MTS) a Active Server Pages (ASP). Kontextová kompozícia bola prvýkrát načrtnutá v COM modeli.

3.2.3.2 COM – prvý základný prepojovací model

COM je základom, na ktorom stavia celý komponentový softvér Microsoftu na svojich platformách. COM je binárny štandard, t.j. nijako nešpecifikuje, ako naň môže byť určitý programovací jazyk naviazaný. COM dokonca nešpecifikuje, čo je objekt alebo komponent. Rovnako nevyžaduje ani nebráni použitiu objektov na implementáciu komponentov. Jednou základnou entitou, ktorú COM špecifikuje, je rozhranie. Na binárnej úrovni je rozhranie reprezentované ako ukazovateľ na uzol rozhrania. Jedinou špecifikovanou časťou uzlu rozhrania je ďalší ukazovateľ uchovávaný v prvom políčku uzlu rozhrania. Tento druhý ukazovateľ je definovaný na ukazovanie na tabuľku premenných procedúr (funkčné ukazovatele). Obrázok 9 ukazuje COM rozhranie na binárnej úrovni.



Obrázok 9. COM rozhranie na binárnej úrovni

3.2.3.2.1 QueryInterface a IUnknown

V súvislosti s COM komponentmi je dôležité uviesť, akým spôsobom sa klient dozvie o ostatných rozhraniach a akým spôsobom porovná identitu COM objektov.

Každé COM rozhranie má prvú metódu nazývanú QueryInterface. Prvý slot tabuľky funkcií každého COM rozhrania teda ukazuje na QueryInterface operáciu. QueryInterface prevezme meno rozhrania, skontroluje, či ho daný COM objekt podporuje a ak hej, vráti zodpovedajúcu referenciu na rozhranie. Indikácia chyby sa vráti v prípade, keď overované rozhranie nie je podporované. Na úrovni QueryInterface sú rozhrania pomenované prostredníctvom identifikátorov rozhraní, ktoré sú globálne jedinečné. Keďže každé rozhranie má QueryInterface operáciu, klient sa môže dostať z akéhokoľvek poskytovaného rozhranie do akéhokoľvek iného. Ako náhle má klient referenciu na

najmenej jedno rozhranie, môže získať prístup ku všetkým ostatným rozhraniam poskytovaných rovnakým COM objektom. [5]

Uzly rozhraní sú oddelené a preto nemôžu slúžiť na jedinečnú identifikáciu COM objektu. COM definuje jedno špeciálne rozhranie, IUnknown, na implementáciu nevyhnutnej funkcionality. Všetky COM komponenty musia implementovať rozhranie IUnknown a všetky ostatné COM rozhrania sú zdedené od rozhrania IUnknown. IUnknown obsahuje tri metódy: QueryInterface, AddRef, and Release.

COM požaduje, že daný COM objekt vráti rovnaký ukazovateľ na uzol rozhrania vždy, keď je požiadaný o rozhranie IUnknown. Keďže všetky COM komponenty musia mať rozhranie IUnknown, identita uzlu rozhrania IUnknown môže slúžiť na identifikáciu celého COM objektu.

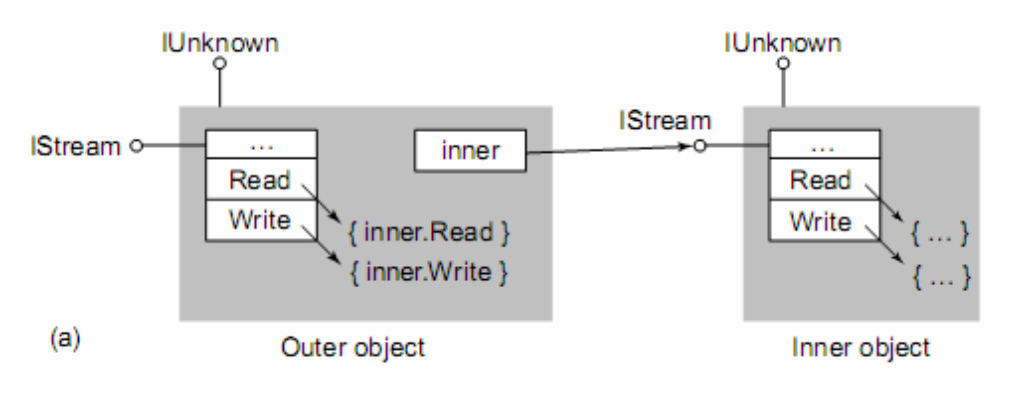
QueryInterface kontrakt vyžaduje, že akákoľvek požiadavka vráti rozhranie, ktoré je v rovnakom COM objekte, to znamená preukáže rovnakú identitu prostredníctvom požiadaviek na IUnknown. Z toho vyplýva, že množina rozhraní preskúmavaných prostredníctvom požiadaviek musí byť trieda ekvivalencie. [5]

Primárne použitie IUnknown je na identifikáciu COM objektu na najabstraktnejšej úrovni, t.j. bez nutnosti akejkoľvek špecifickej funkcionality. Odkazovanie sa na rozhranie IUnknown môžeme prirovnať na odkazovanie sa na Object v objektovo-orientovaných jazykoch.

3.2.3.2.2 COM kompozícia – obsiahnutie (containment) a agregácia

COM nepodporuje žiadnu formu implementačnej dedičnosti. Môžeme to považovať skôr za silnú stránku ako za slabosť. Poznamenajme však, že COM nedefinuje alebo sa nestará, akým spôsobom je individuálny komponent vnútorne realizovaný. Komponent sa môže rovnako skladať z tried, ktoré v rámci komponentu používajú implementačnú dedičnosť. V každom prípade neprítomnosť implementačnej dedičnosti neznamená nedostatočnú podporu pre znovu použitie.

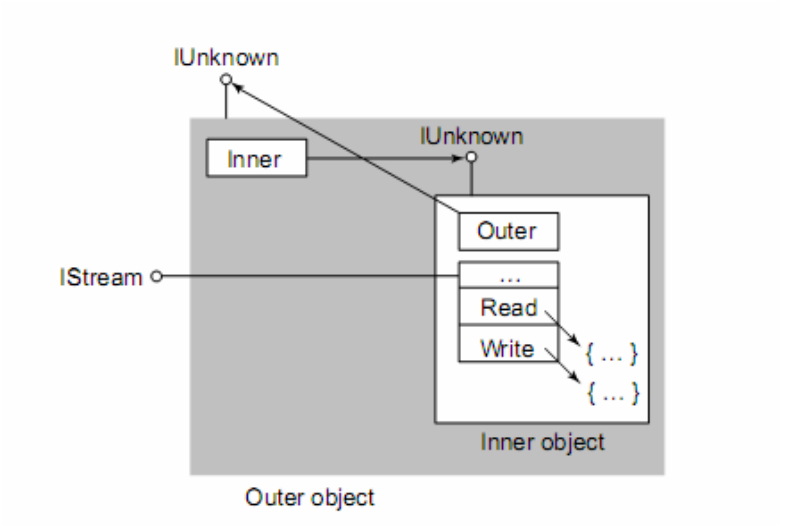
COM podporuje dva spôsoby objektovej kompozície s cieľom umožniť znovu použitie objektov. Tieto dva spôsoby sa nazývajú obsiahnutie a agregácia. Obsiahnutie je jednoduchá technika kompozície objektov – jeden objekt udržiava referenciu na druhý. Prvý objekt, nazývaný ako vonkajší objekt, tak konceptuálne obsahuje druhý, vnútorný objekt. Ak je požiadavky na vonkajší objekt potrebné vyriešiť vnútorným objektom, vonkajší objekt jednoducho presmeruje požiadavku na vnútorný objekt. Presmerovanie nie je nič iné ako volanie metódy vnútorného objektu, aby sa zabezpečila implementácia metódy vonkajšieho objektu. Obsiahnutie je znázornené na obrázku 10. [5]



Obrázok 10. Obsiahnutie [5]

Obsiahnutie postačuje na znovu použitie implementácií obsiahnutých v ostatných komponentoch. Obsiahnutie je predovšetkým úplne neviditeľné klientovi vonkajšieho objektu. Klient volajúci funkciu rozhrania nemôže zistiť, či objekt poskytujúci rozhranie spracováva volanie alebo je volanie presmerované a spracované iným objektom.

Ak sa objavia hlboké hierarchie obsiahnutia alebo ak sú presmerované metódy relatívne nákladné operácie, potom sa môže obsiahnutie stať výkonovým problémom. Pre tento dôvod COM definuje svoju druhú formu znovu použitia, ktorou je agregácia. Základný princíp agregácie je jednoduchý. Namiesto presmerovania požiadaviek môže byť referencia rozhrania vnútorného objektu dodaná priamo klientovi vonkajšieho objektu. Volania na rozhranie potom pôjdu priamo na vnútorný objekt, pričom sa ušetrí cena presmerovania. Agregácia je však užitočná len tam, kde vonkajší objekt nepotrebuje zachytávať volania, napríklad na vykonanie nejakého filtrovania alebo ďalšieho spracovávania. Rovnako je dôležité, aby klient vonkajšieho objektu nemal možnosť rozpoznať, či konkrétne rozhranie bolo agregované z vnútorného objektu.



Obrázok 11. Agregácia [5]

Pri obsiahnutí si nie je vnútorný objekt vedomý, že je obsiahnutý vnútri iného objektu. Pri agregácii je však vnútorný objekt potrebný pre spoluprácu. COM objekt má na výber, či bude alebo nebude podporovať agregáciu. Ak podporuje, tak sa môže stať agregovaným vnútorným objektom.

Prečo je potrebné toto úsilie spolupráce? Pripomeňme si, že COM rozhrania podporujú QueryInterface. Ak je rozhranie vnútorného objektu odhalené klientom vonkajšieho objektu, potom QueryInterface rozhrania vnútorného objektu musí stále pokrývať rozhrania podporované vonkajším objektom. Riešenie je jednoduché. Vnútorný objekt sa oboznámi o rozhraní IUnknown vonkajšieho objektu v čase, keď je agregované. Volanie na jeho QueryInterface sú potom presmerované na QueryInterface vonkajšieho objektu. Relácia agregácie sa prejavuje v ustanovení vzájomných objektových referencií medzi vnútorným a vonkajším objektom. [5]

Obrázok 11 ukazuje scenár použitia agregácie. Znázornenie jedného objektu vnútri druhého má podobne ako u obsiahnutie výlučne ilustratívne účely. Vnútorný objekt je plne samostatný a najpravdepodobnejšie implementovaný iným komponentom ako je vonkajší objekt.

Agregácia môže pokračovať akýkoľvek počet úrovní hlboko. Vnútorné objekty na ktorejkoľvek úrovni vždy odkazujú na rozhranie IUnknown najviac vonkajšieho objektu. Pre vnútorné potreby si vonkajší objekt ponecháva priamy odkaz na rozhranie IUnknown vnútorného objektu. Takýmto spôsobom môže vonkajší objekt požadovať rozhrania vnútorného objektu bez odkázania sa na vlastné rozhranie IUnknown. Ako je zrejmé z obrázku 11, vnútorné a vonkajšie objekty si v agregácii udržujú vzájomné odkazy. [5]

Agregácia, ako čistý nástroj pre výkon v porovnaní s obsiahnutím, je pravdepodobne zmysluplná len pre hlboko zanorené konštrukcie. Toto je jeden z dôvodov, prečo je agregácia v praxi menej dôležitá pre COM ako obsiahnutie. Ďalším dôvodom je nárast zložitosti. Agregácia avšak môže byť použitá v prípadoch, kde je potrebné efektívne znovu použitie komponentovej funkcionality. Agregácia sa rovnako môže použiť na pridanie podpory nových rozhraní pre inak nezmenené objekty. Avšak je potrebné byť pri tom opatrný, keďže nové rozhrania nemôžu byť v kolízii s akýmikoľvek (všeobecne neznámymi) rozhraniami daného objektu.

3.2.4 Zhrnutie komponentových technológií

Všetky prístupy sa spoliehajú na neskorú väzbu, zapuzdrenie a dynamický polymorfizmus. Všetky prístupy podporujú dedičnosť rozhraní, hoci v prípade COM je to takmer irelevantné. Inými slovami všetky prístupy sa spoliehajú na nejakú časť objektového modelu.

4 Softvérové rámce platformy Eclipse

V tejto kapitole sa venujeme popisu softvérových rámcov platformy Eclipse pre prácu s meta-modelmi, podrobne je popísaný EMF, ďalej GEF a GMF.

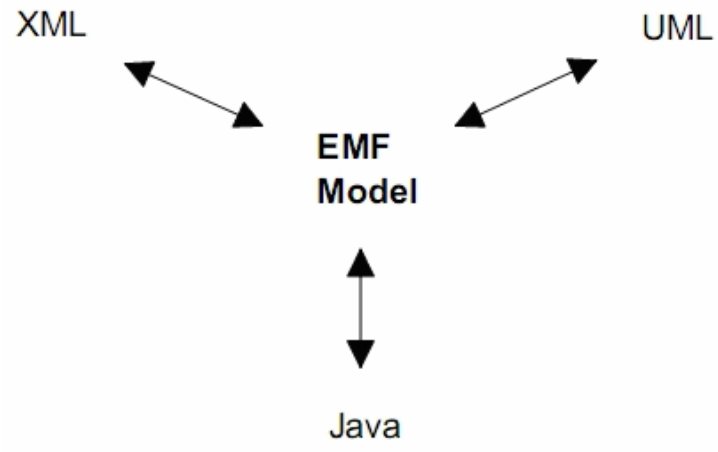
4.1 EMF

EMF sa skladá z troch základných častí: EMF, EMF.Edit a EMF.Codegen. EMF je kľúčový framework, ktorý zahŕňa meta-model Ecore pre opis modelov a behovú podporu pre modely. EMF.Edit framework zahŕňa všeobecné znovu použiteľné triedy pre výstavbu editorov pre EMF modely s podporou „undo“ a „redo“. EMF.Codegen je nástroj na generovanie kódu potrebného na výstavbu kompletného editoru pre model. Zahŕňa GUI, z ktorého môžu byť špecifikované parametre generovania.

K dispozícii sú tri stupne generovania kódu: model, adaptéry a editor. Model poskytuje Java rozhrania a implementačné triedy pre všetky triedy modelu. Adaptéry sú implementačné triedy nazývané ItemProviders (viď kapitola 4.1.6.2), ktoré prispôsobujú triedy modelu pre editovanie a zobrazovanie. Editor vytvorí vhodne štruktúrovaný editor, ktorý zodpovedá odporúčanému štýlu pre Eclipse EMF editory modelov a slúži ako počiatočný bod pre ďalšie prispôsobovanie.

4.1.1 Úvod

EMF je silný framework a nástroj na generovanie kódu pre výstavbu aplikácií založených na definícii modelu. Je navrhnutý pre praktické využitie modelovania a použiteľný pre bežného Java programátora. EMF zjednocuje tri dôležité technológie: Java, XML a UML, ako je to znázornené na obrázku 12.



Obrázok 12. EMF verzus XML, Java, UML

Modely môžu byť definované prostredníctvom UML nástroja, XML Schema alebo špecifikovaním jednoduchých anotovaných Java rozhraní, pomocou ktorých programátor napíše abstraktné rozhrania (malú podmnožinu toho, čo by inak musel napísať) a zvyšok sa vygeneruje

automaticky a pripojí sa k existujúcemu kódu. Prepojením konceptov modelovania s jednoduchou reprezentáciou týchto konceptov v Jave EMF prepojila medzeru medzi vývojármi modelov a Java programátormi.

4.1.2 Definovanie modelu

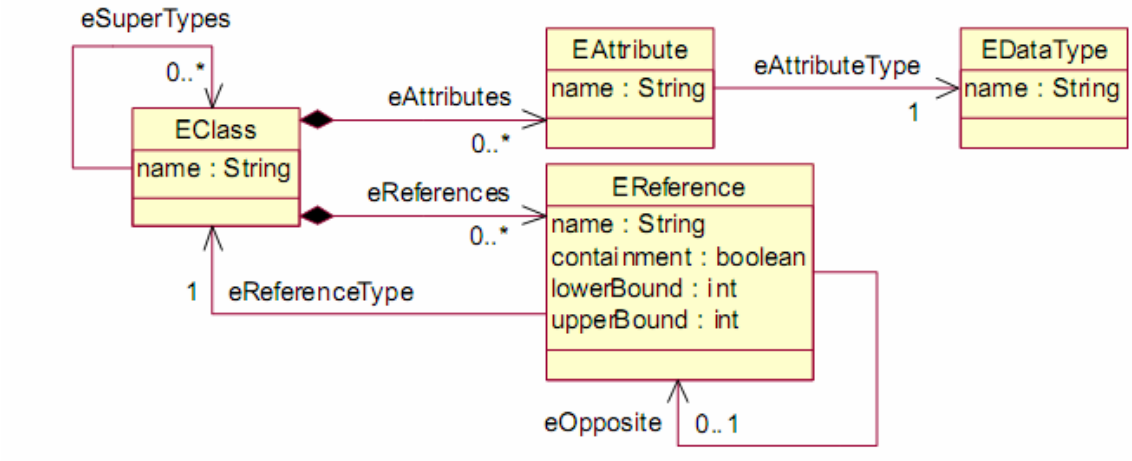
EMF prepája koncepty modelovania priamo na ich implementáciu. Prostredníctvom EMF môžeme považovať modelovanie a programovanie za totožné. Namiesto oddelenia vysoko úrovňového modelovania od nízko úrovňovej implementácie tieto dva koncepty EMF spája ako dve dobre integrované časti jedného systému.

Modelovanie nám umožňuje jednoduchším spôsobom popísať to, čo má aplikácia vykonávať, na rozdiel od popisu prostredníctvom kódu. To nám dáva možnosť, ako vysoko úrovňovým spôsobom komunikovať dizajn a zároveň vygenerovať časť, poprípade celú implementáciu. Na EMF teda môžeme nazerať ako na jemný úvod do modelovania, ak sme primárne programátori alebo ako na technológiu, ktorá sa uberá smerom MDA (Model Driven Architecture), ak sa primárne venujeme modelovaniu. Z predchádzajúceho textu nám vyplýva, že EMF stojí medzi dvoma extrémami, preferovanie implementačného programovania a preferovanie modelovania. [17]

EMF model je v podstate podmnožina diagramu tried v UML. Je to jednoduchý model tried alebo dát aplikácie. S jeho použitím môžeme dosiahnuť obrovské množstvo výhod modelovania v Java vývojovom prostredí. Mapovanie medzi EMF modelom a Javou je prirodzené a jednoduché pre pochopenie. Na druhej strane je to dostatočné pre dosiahnutie dátovej integrácie medzi aplikáciami a zvýšenie produktivity použitím generovania kódu.

4.1.2.1 Ecore meta-model

Model používaný na reprezentáciu modelov v EMF sa nazýva Ecore. Samotný Ecore je EMF model a teda svoj vlastný meta-model. Rovnako môžeme povedať, že je to meta-metamodel. Meta-model je jednoducho model modelu a keď je model sám meta-modelom, potom je meta-model v skutočnosti meta-metamodelom.



Obrázok 13. Ecore meta-model [19]

Na obrázku 13 je znázornený zjednodušený Ecore meta-model. Trieda EClass sa používa na reprezentáciu modelovaných tried. Obsahuje svoje meno a nula alebo viacero atribútov a nula alebo viacero referencií. EAttribute je použitý na reprezentáciu modelovaných atribútov. Atribúty majú meno a typ. EReference sa používa na reprezentáciu jedného konca asociácie medzi triedami. Obsahuje meno, príznak na indikáciu, či reprezentuje obsiahnutie a referenčný typ, ktorým je iná trieda. EDataType sa používa na reprezentáciu typu atribútu. Dátový typ môže byť primitívny (napríklad int, float) alebo objektového typu (napríklad java.util.Date).

Z predchádzajúceho diagramu je zrejmé, že mená tried blízko korešpondujú s UML pojmami. UML však nemôže byť Ecore model, pretože Ecore je len malá zjednodušená podmnožina UML. UML podporuje okrem modelovania štruktúry tried aplikácie i modelovanie správania aplikácie.

Na opis štruktúry tried aplikačných modelov používame inštancie tried definovaných v Ecore. Uveďme si ako príklad Objednávku tovaru, ktorá môže obsahovať nula alebo viacero položiek.

Obrázok 14. Model objednávky tovaru [17]

Triedu objednávka tovaru popíšeme ako inštanciu triedy EClass pomenovanú „PurchaseOrder“. Obsahuje dva atribúty (inštancie EAtribute, ku ktorým sa pristupuje prostredníctvom eAttributes) pomenované „shipTo“ a „billTo“ a jednu referenciu pomenovanú „items“, ktorej cieľový typ (EReferenceType) je ďalšia inštancia triedy Eclass pomenovaná „Item“.

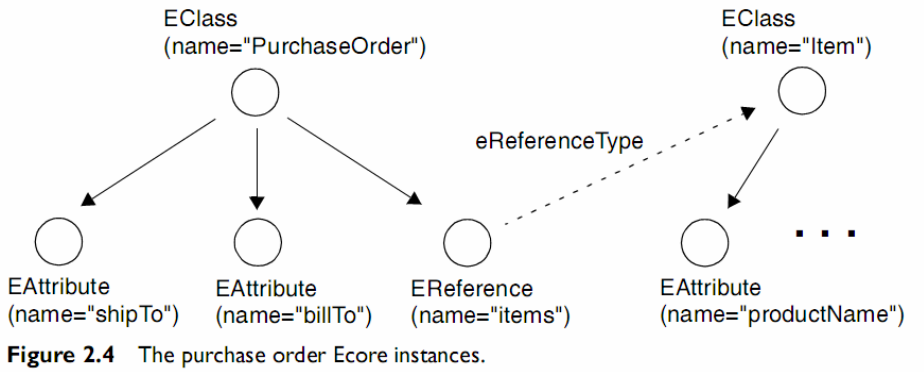


Figure 2.4 The purchase order Ecore instances.

Obrázok 15. Inštancie tried modelu [17]

Keď vytvárame inštancie tried definovaných v Ecore pre definovanie modelu našej aplikácie, vytvárame takzvaný hlavný model (core model).

4.1.2.2 Vytváranie a editovanie modelu

Model môžeme vytvoriť rôznymi spôsobmi: prostredníctvom anotovaných Java rozhraní, XML Schema alebo UML. XML Schema a UML sú voliteľným rozšírením. Pri UML máme tri možnosti: priame editovanie v Ecore (napríklad jednoduchý stromový Ecore editor), importovanie z UML (jedine pre Rational Rose), exportovanie z UML.

4.1.2.3 XMI serializácia

Na serializáciu modelov je použité XMI (XML Metadata Interchange). XMI je štandardom pre serializáciu metadát. Ecore XMI je štandardná XML serializácia metadát, ktoré používa EMF.

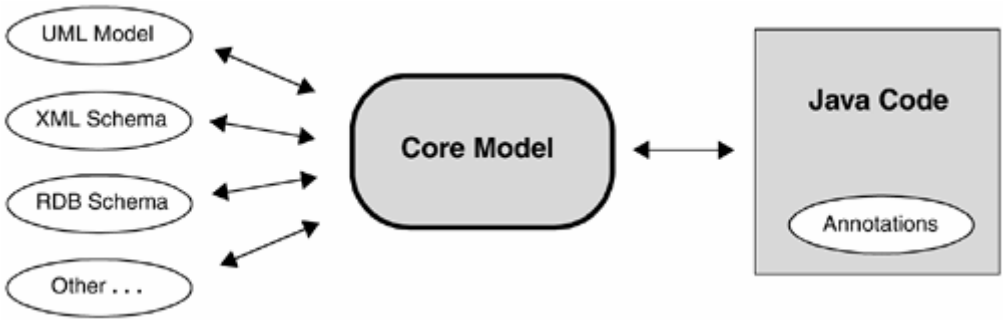
4.1.2.4 Java anotácie

EMF generátor predpokladá, že nie každé rozhranie a metóda sú časťou modelu. Generuje kód, ktorý je spájaný s kódom napísaným používateľom. Preto je potrebné označiť časti rozhrania, ktoré zodpovedajú elementom modelu a ktoré sa teda majú vygenerovať. Používajú sa na to Java anotácie „@model“ v Javadoc komentári.

4.1.2.5 Ecore zhrnutie

Výhodou použitia XML Schema pre definovanie modelu je, že inštancie modelu môžu byť serializované, aby zodpovedali tejto schéme. XML Schema okrem definovania modelu hovorí aj o perzistentnej forme modelu.

Jednou s funkcií EMF je podpora viacerých formátov perzistencie modelov, ako je znázornené na obrázku 16.



Obrázok 16. Podpora rôznych formátov perzistencie modelov [17]

4.1.3 Generovanie kódu

Najvýznamnejšou výhodou GMF je nárast produktivity, ktorá je dosiahnutá pomocou automatického generovania kódu.

4.1.3.1 Vygenerované triedy modelu

Je dôležite podotknúť, že Ecore trieda (t.j. EClass) zodpovedá rozhraniu a príslušnej implementačnej triede v Jave. Napríklad EClasss pre triedu ObjednavkaTovaru sa mapuje na Java rozhranie (public interface ObjednavkaTovaru) a zodpovedajúcu implementačnú triedu (public class ObjednavkaTovaruImpl extends ... implements ObjednavkaTovaru). Dôvodom oddelenia rozhrania od implementačnej triedy je umožnenie viacnásobnej dedičnosti v Jave.

Každé vygenerované rozhranie priamo alebo nepriamo dedí od základného rozhrania EObject, ktoré je EMF ekvivalentom triedy java.lang.Object, takže je základom všetkých modelovaných objektov. EObject prináša tri základné skupiny správania. Prvou je metóda eClass(), ktorá vracia metaobjekt objektu (t.j. EClass). Druhou sú metódy eContainer() a eResource(), ktoré vracajú nadradený objekt a zdroj. Sú súčasťou API perzistencie. Treťou skupinou sú metódy eGet(), eSet(), eIsSet() a eUnSet(), ktoré poskytujú API pre reflektívny prístup k objektom. EObject ďalej dedí od rozhrania Notifier, ktoré prináša dôležitú charakteristiku každému objektu, a tou je notifikácia o

zmene modelu v podobe návrhového vzoru Observer [18]. Podobne ako perzistencia objektov, tak i notifikácia je dôležitou súčasťou EMF. [17]

Pri generovaní kódu sa pre všetky atribúty a referencie vygenerujú get() a set() metódy. Metóda set() okrem nastavenia príslušnej premennej rovnako pošle notifikáciu všetkým pozorovateľom (observers), ktorý majú záujem byť informovaní o zmene stavu. V prípade, ak objekt nemá pozorovateľov, je dobré sa vyhnúť volaniu nákladnej metódy eNotify() prostredníctvom testu eNotificationRequired(). [17]

EMF ďalej vygeneruje ďalšie dve triedy: továreň (factory) a balík (package). Vygenerovaná továrenská trieda poskytuje metódy vytvárania pre každú triedu v modeli. Vygenerovaný balík poskytuje vhodné prístupové metódy ku všetkým Ecore metadátam. [17]

Vygenerovaný kód je prehľadný, jednoduchý a efektívny. Cieľom je, aby vygenerovaný kód vyzeral rovnako, akoby bol vytvorený ručne používateľom. Jediný rozdiel je v tom, že keďže je generovaný, tak je určite správny a zároveň vývojárom ušetrí mnoho času.

4.1.3.2 Ostatné vygenerované časti

Voliteľne je možné vygenerovať manifest súbor zásuvného modulu (editoru), takže model môže byť použitý ako Eclipse zásuvný modul, ďalej XML Schema pre model a pod. Generátor môže s použitím EMF.Edit rozšírení vygenerovať triedy, ktoré umožnia náhľady a nevratné editovanie modelu založené na príkazoch. Môže dokonca vygenerovať funkčný editor pre model.

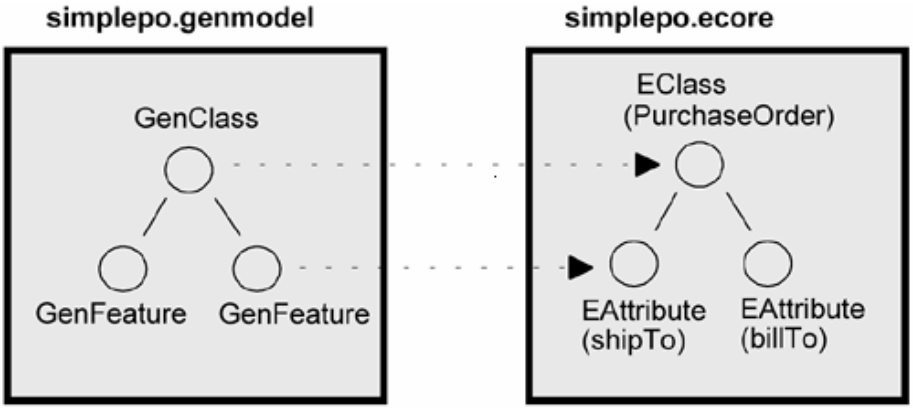
4.1.3.3 Pregenerovanie a spojenie

EMF generátor vytvorí súbory, ktoré sú určené pre kombináciu generovaných a ručne napísaných častí. Umožňuje nám pridávať metódy a inštančné premenné, ktoré pri pregenerovaní zostanú zachované. EMF používa značky „@generated“ v Javadoc komentároch na identifikáciu vygenerovaných častí. V prípade, ak vytvoríme metódu, ktorej názov koliduje s vygenerovanou metódou, naša metóda sa zachová a vygenerovaná bude odstránená. Je rovnako možné presmerovať generovanú metódu, ak ju chceme preťažiť, ale napriek tomu volať generovanú verziu.

4.1.3.4 Model generovania

Väčšina dát (t.j. triedy, ich mená, atribúty a referencie) potrebných EMF generátorom je uložená v hlavnom modeli. Avšak ku generovaniu je potrebných viacero informácií, ako napríklad kde uložiť vygenerovaný kód alebo aký prefix použiť pre vygenerovanú továreň a mená tried balíka. Na uloženie týchto informácií je použitý model generovania, aby boli tieto informácie k dispozícii v čase pregenerovania modelu. Podobne ako Ecore, tak i model generovania je EMF model. Model generovania poskytuje prístup ku všetkým dátam potrebných pre generovanie vrátane Ecore časti prostredníctvom obalenia zodpovedajúceho hlavného modelu. To znamená, že triedy modelu generovania sú „dekorátormi“ (od návrhového vzoru Decorator [18]) Ecore tried.

Má to nasledovný význam. Generátor vychádza z modelu generovania a nie z hlavného modelu. Keď sa použije generátor, tak sa edituje model generovania, ktorý striedavo nepriamo pristupuje k hlavnému modelu, z ktorého sa generuje. V systéme sa teda nachádzajú dva súbory: *.ecore, čo je XMI serializácia hlavného modelu a *.genmodel, čo je serializovaný model generovania s odkazmi na *.ecore súbor. Vzťah medzi týmito súbormi je znázornený na obrázku 17.



Obrázok 17. Vzťah medzi .ecore a .genmodel [17]

Týmto oddelením modelu generovania od hlavného modelu dosiahneme, že Ecore meta-model môže zostať čistý a nezávislý na akýchkoľvek informáciách, ktoré sú relevantné len pre generovanie kódu. Je potrebné udržovať synchronizáciu medzi týmito modelmi, ktorá sa však vykonáva automaticky.

4.1.4 EMF framework

EMF poskytuje okrem nárastu produktivity aj iné výhody, ako notifikácia o zmene modelu, podpora perzistencie zahŕňajúca XMI serializáciu a efektívne reflektívne API pre generickú manipuláciu s EMF objektmi. Poskytuje i základ pre spoluprácu s ostatnými nástrojmi a aplikáciami založenými na EMF.

4.1.4.1 Notifikácia a adaptéry

Ako bolo spomínané v kapitole 4.1.3.1, každá EMF vygenerovaná trieda implementuje funkčnosť notifikácie, t.j. môže poslať notifikáciu, kedykoľvek sa zmení jej atribút alebo referencia. Pozorovatelia notifikácie v EMF sa nazývajú adaptéry, pretože okrem pozorovania sa často používajú na rozšírenie správania (t.j. podpora doplnkových rozhraní bez použitia dedičnosti) objektu, ku ktorému sú pripojené. Adaptér môže byť pripojený ku ktorémukoľvek objektu EObject.

Na rozdiel od jednoduchých pozorovateľov sa pripojenie adaptérov pre rozšírenie správania vykoná prostredníctvom použitia továrne adaptérov (adapter factory). Továreň adaptérov slúži na pripojenie adaptéru požadovaného typu k určitému objektu. Ak je adaptér požadovaného typu pripojený k danému objektu, tak sa vráti exitujúci adaptér, inak sa vytvorí nový. V EMF je továreň adaptérov zodpovedná za vytváranie adaptérov, EMF objekt nevie o tom, že môže byť pozmenená jeho funkčnosť prostredníctvom adaptéru. Tento prístup nám dovoľuje väčšiu flexibilitu implementovať rovnaké rozšírenie správania viac ako jedným spôsobom. [17]

EMF umožňuje rovnako informovať o zmenách nielen jednotlivé objekty, ale aj hierarchické štruktúry do seba zanorených objektov. Tieto objekty vkladáme do zdrojov, ako je popísané v závere kapitoly 4.1.4.2.

Adaptéry sú základom pre používateľské rozhranie a podporu príkazov poskytovaných prostredníctvom EMF.

4.1.4.2 Perzistencia objektov

Schopnosť perzistencie a odkazovania sa na iné perzistentné objekty modelu je jednou z najdôležitejších výhod EMF modelovania. Je základom pre jemno-zrnnú (fine-grain) integráciu dát

medzi aplikáciami. EMF podporuje jednoduché ale výkonné mechanizmy pre správu objektovej perzistencie. [17]

Ako bolo uvedené v predchádzajúcich kapitolách, hlavné modely sú serializované prostredníctvom použitia XMI. EMF zahŕňa XMI serializátor, ktorý sa môže použiť pre perzistenciu objektov akýchkoľvek modelov, nielen Ecore. Ak je model definovaný prostredníctvom použitia XML Schema, EMF umožňuje perzistenciu objektov v podobe XML inštančného dokumentu zodpovedajúceho danej schéme.

EMF dovoľuje ukladať objekty v akejkoľvek perzistentnej forme, ale je potrebné napísať daný kód serializácie explicitne. Ale keď je už raz vytvorený, tak sa model môže odkazovať (a môže byť naň odkazované) na objekty v iných modeloch a dokumentoch nezávisle na tom, aká perzistentná forma sa na ne použila.

V tejto časti sa vrátime k popisu metód eContainer() a eResource(), ktoré boli spomenuté v kapitole 4.1.3.1. Kedykoľvek je nejaký objekt pridaný do relácie obsiahnutie (containment) s nejakým iným objektom, tak sa nastaví aj kontajner pridaného objektu. Takže, ak sa zavolá eContainer() na objekt, ktorý je obsiahnutý v nejakom inom objekte, vráti sa zahŕňajúci objekt.

Aby bola možná perzistencia tejto dvojice objektov, potrebujeme ich vložiť do zdroja (resource). Rozhranie Resource sa používa na reprezentáciu fyzického úložného priestoru, napríklad súboru. Na perzistenciu týchto objektov potrebujeme pridať koreňový objekt do zdroja. Po pridaní týchto objektov do zdroja a zavolaní metódy eResource() na ktorýkoľvek z nich nám táto metóda vráti daný zdroj. Ďalším dôležitým rozhraním je ResourceSet, čo je množina zdrojov, ku ktorým je pristupované spoločne za účelom umožnenia referencií medzi viacerými dokumentmi. Toto rozhranie je rovnako továreň pre svoje zdroje. [17]

4.1.4.3 Reflektívne EObject API

Toto reflektívne API môžeme použiť namiesto vygenerovaných metód pre čítanie a zápis modelu. Ako bolo spomenuté v kapitole 4.1.3.1, EObject definuje generické reflektívne API pre manipuláciu s inštanciami.

4.1.4.4 Dynamické EMF

Doteraz sme za EMF prínos považovali použitie ako nástroja na generovanie implementácií modelov. Niekedy je však potrebné zdieľanie objektov bez nutnosti dostupnosti vygenerovaných implementačných tried. Dosahuje sa to prostredníctvom použitia reflektívneho API. Zaujímavá charakteristika reflektívneho API je, že môže byť takisto použité na manipuláciu inštancií dynamických negenerovaných tried.

Uveďme si nasledovný príklad. Predstavme si, že sme nevytvorili model alebo nespustili EMF generátor na vytvorenie Java implementačných tried bežným spôsobom. Namiesto toho sme jednoducho vytvorili hlavný model za behu. Máme hlavný model nachádzajúci sa v pamäti, pre ktorý sme negenerovali žiadne Java triedy. Môžeme vytvoriť inštancie objektov modelu a inicializovať ich použitím reflektívnych volaní, ako bolo uvedené v kapitole 4.1.3.1. Táto implementácia je pomalšia ako vygenerovaná implementácia, ale správanie je presne rovnaké.

Dynamické EMF umožňuje použitie generovaných a dynamických tried dokopy. Jediné, čo je potrebné, je hlavný model v pamäti. V prípade, ak sa generované implementačné triedy neskôr

dodajú, tak sa potom použijú namiesto dynamických tried. Z pohľadu klienta je jedinou zmenou zmena rýchlosti kódu.

4.1.5 EMF a modelovacie štandardy

V tejto kapitole stručne popíšeme vzťah s dôležitými modelovacími štandardmi združenia Object Management Group (OMG), a to konkrétne UML, MOF, XMI, MDA.

4.1.5.1 UML

EMF sa týka len jedného aspektu UML – modelovania tried.

4.1.5.2 MOF

MOF a Ecore majú mnoho podobného v schopnosti špecifikovať triedy a ich štrukturálne vlastnosti a správanie, dedičnosť, balíčky a podporu reflektivity. Líšia sa v oblasti životného cyklu, štruktúry dátových typov, vo vzťahoch medzi balíčkami a komplexnými aspektmi asociácií.

4.1.5.3 XMI

Hoci je štandardne XMI používané ako serializačný formát pre EMF modely, najvhodnejšie je pre serializáciu modelov reprezentujúcich metadáta, t.j. meta-modelov ako samotné Ecore.

4.1.5.4 MDA

EMF podporuje kľúčové koncepty MDA používania modelov ako vstup pre vývoj a integráciu nástrojov: v EMF je model použitý na riadenie generovania kódu a serializáciu pre výmenu dát.

4.1.6 Editovanie modelu s EMF.Edit

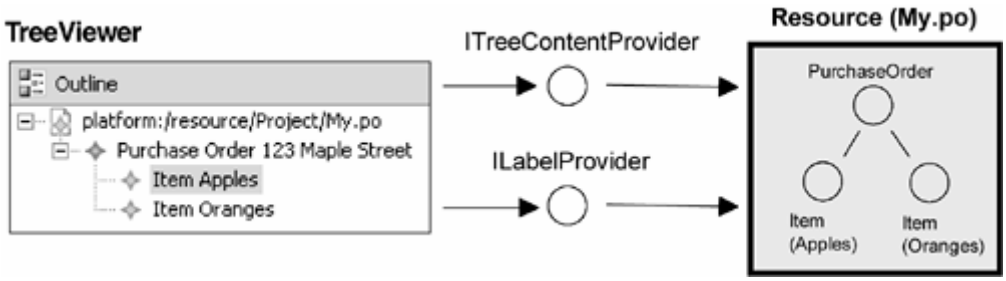
EMF.Edit framework sa používa na výstavbu funkčných prehliadačov a editorov pre model. Je možné vygenerovať editor, ktorý zobrazí a umožní editovať (t.j. kopírovať, vložiť, „drag and drop“) inštancie modelu použitím štandardných JFace prehliadačov a panelu vlastností s možnosťou neobmedzenej funkcie „undo/redo“. Alebo je možné využiť reflektívnu podporu v EMF.Edit a mať k dispozícii rovnaký druh editovania reflektívne dokonca s dynamickým EMF modelom, ktorý nebol vygenerovaný.

4.1.6.1 Zobrazovanie a editovanie EMF modelov

V tejto kapitole si popíšeme základy zobrazovania EMF modelov a možnosti ich editovania.

4.1.6.1.1 Základy Eclipse používateľského rozhrania

Používateľské rozhranie, zahrnuté v JFace, ktoré je časťou Eclipse frameworku používateľského rozhrania, je množina znovu použiteľných náhľadových tried pre zobrazovanie štruktúrovaných modelov. Namiesto priamej práce s objektmi modelu, JFace náhľady používajú poskytovateľa obsahu (content provider) pre riadenie obsahu a poskytovateľa grafických popisov (label provider) pre dodanie textu a ikon pre zobrazované objekty. Každá náhľadová trieda používa poskytovateľa obsahu, ktorý implementuje určité rozhranie poskytovateľa. Napríklad TreeViewer používa poskytovateľa obsahu, ktorý implementuje rozhranie ITreeContentProvider, ako je znázornené na obrázku 18. [17]



Obrázok 18. Poskytovatelia obsahu a grafických popisov [17]

Uveďme si príklad. Chceme zobraziť zdroj (viď kapitola 4.1.4.2) v TreeViewer. Poskytneme mu koreňový objekt. Odpovie volaním metód getText() a getImage() na svojho poskytovateľa grafického popisu, aby získal text a ikony popisov pre zobrazované objekty. TreeViewer zavolá metódu getChildren() na svojho poskytovateľa obsahu, aby získal ďalšiu úroveň objektov na zobrazenie v strome. Podobne sa to bude opakovať pre zvyšok stromu. Ďalej sa pre jednotlivé vlastnosti objektov modelu vygeneruje panel vlastností.

JFace okrem TreeViewer obsahuje i triedy TableViewer a ListViewer, ktoré pracujú rovnakým spôsobom. Na získanie obsahu používajú odlišného poskytovateľa obsahu - IStructuredContentProvider. Rozhranie ITreeContentProvider dedí od rozhrania IStructuredContentProvider, takže akákoľvek implementačná trieda stromového poskytovateľa obsahu môže rovnako byť použitá pre podporu ostatných náhľadov. [17]

Ďalšou dôležitou časťou Eclipse UI rozhrania je mechanizmus akcií. Akcie implementujú rozhranie IAction a reprezentujú príkazy, ktoré môžu byť spúšťané z položiek menu a tlačidiel panelu nástrojov. Keď sa vyberie položka menu alebo tlačidlo panelu nástrojov, framework vyvolá pridruženú akciu prostredníctvom volania metódy run().

4.1.6.1.2 EMF.Edit podpora

V tejto kapitole si popíšeme, ako pomáha EMF.Edit implementovať používateľské rozhranie pre model založený na EMF.

Na implementáciu stromového náhľadu potrebujeme implementáciu rozhrania ITreeContentProvider, ktoré je zodpovedné za vracanie potomkov objektov modelu. Rovnako potrebujeme implementáciu rozhrania ILabelProvider, ktorá nám vráti vhodný textový a grafický popis grafického náhľadu objektov. Pre panel vlastností potrebujeme vytvoriť množinu IPropertyDescriptors pre Ecore atribúty a referencie. EMF.Edit podporuje dva spôsoby, ako to implementovať. Prvým je použitie reflektívneho EObject API alebo prostredníctvom použitia vygenerovaných tried. Reflektívny prístup komunikuje s hlavným modelom za behu a poskytuje implementáciu na základe odhadov, ako by malo GUI pre daný hlavný model vyzerať. Druhý spôsob je použitie EMF.Edit generátora na vygenerovanie implementácie. Vygenerované správanie je opäť rovnaké, ako pri reflektívnom prístupe, avšak pri druhom spôsobe je prispôsobenie riešenia oveľa jednoduchšie.

EMF.Edit rovnako obsahuje podporu Eclipse akcií a modifikáciu modelu. Zmena stavu objektu modelu prostredníctvom spustenia akcie je v EMF implementovaná pomocou delegovania príkazu (návrhový vzor Command [18]). EMF.Edit obsahuje všeobecné implementácie mnohých bežných príkazov rovnako ako podporu pre prispôsobenie ich správania alebo pre implementáciu vlastných príkazov.

Ako znázorňuje obrázok 19, EMF.Edit je premostením medzi Eclipse používateľským rozhraním (UI – user interface) a EMF.Core frameworkom.

Obrázok 19. Prepojenie Eclipse UI a EMF Core

Veľké množstvo EMF.Edit funkcií je nezávislých na UI. Na podporu znovu použitia častí nezávislých na UI je EMF.Edit framework rozdelený na dve oddelené časti. Prvou je org.eclipse.emf.edit, ktorá je nízko úrovňová časť nezávislá na UI. Druhou je org.eclipse.emf.edit.ui, ktorá obsahuje Eclipse implementačné triedy závislé na UI. [17] Väčšina editovania je v skutočnosti vykonávaná prostredníctvom časti nezávislej na UI. Časť UI rieši aktuálne napojenie na obrazovku spolu s implementáciou previazanou na Eclipse UI framework. Ako si popíšeme v nasledovných kapitolách, veľké množstvo práce je vykonávané prostredníctvom delegovania dvom dôležitým mechanizmom nezávislých na UI: sú to poskytovatelia funkčnosti a príkazový framework.

4.1.6.2 Poskytovatelia funkčnosti (item providers)

Poskytovatelia funkčnosti sú najdôležitejšími objektmi v EMF.Edit. Používajú sa na prispôsobenie objektov modelu, aby objekt mohol poskytovať všetky rozhrania, ktoré potrebuje na to, aby mohli byť z neho vytvárané náhľady a mohol byť editovaný. Ako bolo popísané v kapitole 4.1.4.1, adaptéry môžu byť použité ako rozšírenia správania a zároveň ako pozorovatelia zmien. Adaptéry sú preto presne vhodné na implementáciu poskytovateľov funkčnosti. Ako rozšírenia správania môžu prispôsobiť objekty modelu, aby implementovali akékoľvek rozhrania, ktoré editory a náhľady potrebujú a zároveň ako pozorovatelia (observers) budú upozornení na zmeny stavu, ktorý môžu potom poslať načúvajúcim náhľadom.

Hoci poskytovatelia funkčnosti sú obyčajne EMF adaptéry, nie je to tak vždy. Poskytovateľ funkčnosti, ktorý niečo poskytuje EMF objektu, bude adaptér, ale iní poskytovatelia funkčnosti môžu reprezentovať nemodelované objekty skombinované do náhľadu spolu s modelovanými položkami. EMF.Edit framework bolo navrhnuté na podporu vytvárania náhľadov EMF modelov, ktoré môžu byť štrukturálne odlišné od modelu samotného (t.j. pohľady, ktoré obmedzujú objekty alebo obsahujú prídavné nemodelované objekty). [17]

EMF.Edit framework implementuje schému delegovania, pomocou ktorej väčšina funkcií obsiahnutých v objektoch modelu je v konečnom dôsledku implementovaná v ich pridružených poskytovateľoch funkčnosti. Poskytovatelia funkčnosti potrebujú vykonávať štyri základné úlohy: implementovať poskytovateľov obsahu a grafických popisov, poskytovať popisy vlastností pre EMF objekty, vystupovať v roli príkazovej továrne (command factory) pre príkazy na ich pridružených objektov modelu, posielať notifikácie zmien EMF modelu náhľadom.

4.1.6.3 Príkazový framework

Ďalšou významnou vlastnosťou EMF.Edit frameworku je podpora editovania EMF modelu založeného na príkazoch, ktorá zahŕňa automatické „undo“ a „redo“.

Príkazový framework je rozdelený na dve časti: obyčajný príkazový framework a EMF.Edit príkazy. Obyčajný framework definuje základné rozhrania príkazov a poskytuje niektoré implementačné triedy ako napríklad základný zásobník príkazov, zložený príkaz pre skladanie príkazov z iných príkazov a iné užitočné implementácie príkazov. Príkazy v tomto obyčajnom frameworku sú veľmi všeobecné a môžu byť používané nezávisle na EMF.Edit. V skutočnosti dokonca nezávisia ani na EMF objektoch modelu (t.j. EObject), sú úplne nezávislé na EMF.

EMF.Edit príkazy sú implementačné triedy najmä na editovanie objektov modelu (t.j. EObject). EMF.Edit obsahuje celú množinu všeobecných implementačných tried príkazov, ktoré prostredníctvom využitia reflektívneho EObject API poskytujú podporu pre nastavovanie atribútov, pridávanie alebo odoberanie referencií, kopírovanie objektov a iné druhy modifikácií modelu. EMF.Edit príkazy dedia od a stavajú na rozhraniach definovaných v obyčajnom príkazovom frameworku, ale zavádzajú závislosť na Ecore modeli.

4.1.6.4 Generovanie EMF.Edit kódu

EMF.Edit podpora generovania môže byť využitá na vygenerovanie poskytovateľov funkčnosti a iných tried potrebných na editovanie modelu. EMF.Edit generátor kódu nie je samostatným nástrojom, je to len ďalšia vlastnosť generátora modelu. Ako sme videli v predchádzajúcich kapitolách, kód modelu je štandardne vygenerovaný do existujúceho projektu, ktorý obsahuje hlavný model (.ecore) a model generovania (.genmodel). Ako sme opísali v kapitole 4.3.1.2, EMF.Edit framework je rozdelený na dve oddelené časti: časť nezávislá na UI a časť závislá na Eclipse UI. Štandardne sa týmto rozdelením riadi aj kód vygenerovaný prostredníctvom EMF.Edit. Funkcia vygenerovania kódu editovania vygeneruje zásuvný modul obsahujúci triedy podporujúce editovanie, ktoré sú nezávislé na UI, pričom funkcia vygenerovania kódu editoru vygeneruje zvyšok do samostatného zásuvného modulu, ktorý závisí na Eclipse UI. Je však možné nastaviť generátor, aby všetko umiestnil do jedného zásuvného modelu.

4.1.6.4.1 Regenerovanie EMF.Edit zásuvných modulov

Keď regenerujeme existujúce projekty, EMF generátor podporuje rovnaký druh spájania EMF.Edit kódu ako to je pri kóde modelu, ktorý je opísaný v kapitole 4.1.2.4. Vygenerované triedy je možné editovať za účelom pridania metód a inštančných premenných alebo modifikácie vygenerovaných. Keď sa odstráni označenie „@generated“ z vygenerovanej metódy, ktorá sa následne zmení, tak tieto zmeny budú zachované počas regenerovania.

4.2 GEF

GEF je framework založený na MVC pre vytváranie grafických editorov. Umožňuje jednoduchý vývoj grafických reprezentácií pre exitujúce modely. Celá grafické zobrazovanie sa vykonáva prostredníctvom Draw2D frameworku, čo je štandardný 2D kresliaci framework založený na SWT od Eclipse. Možnosti editovania umožňujú budovanie grafických editorov takmer pre akýkoľvek model. S týmito editormi je možné jednoduchým spôsobom modifikovať modely, ako napríklad zmena vlastností elementov alebo zmena štruktúry modelu rôznymi spôsobmi naraz. Všetky tieto

modifikácie modelu môžu byť vykonávané v grafickom editore použitím bežných funkcií ako „drag and drop“, „copy and paste“ a akcií vyvolaných z menu alebo panelu nástrojov.

4.2.1 Úvod do Draw2D

Draw2D poskytuje odľahčený grafický systém, na ktorom stojí GEF zobrazovanie. Draw2D je umiestnené v SWT plátne a riadi vykresľovanie a udalosti myši, ktoré sa objavia na plátne a sú delegované Draw2D prvkom. Prvky sú analogické k oknám v používateľských grafických systémoch. Môžu mať ľubovoľné nepravidelné tvary a môžu byť zanorené za účelom vytvorenia komplexných scén a vlastných prvkov riadenia. Prvky môžu byť priehľadné alebo nepriehľadné, môžu byť usporiadané do vrstiev a tak umožniť, aby boli časti diagramu skryté alebo vylúčené z určitých operácií.

4.2.1.1 Odľahčený systém

Odľahčený systém je grafický systém, ktorý je umiestený vnútri jedinej riadiacej komponente. S grafickými objektmi v odľahčenom systéme, v Draw2D nazývaných zobrazenie (figures), sa zaobchádza, akoby to boli normálne okná. Môže podporovať výber okna prostredníctvom ukázania myšou (bez kliknutia) a výber okna po kliknutí myšiu, dostávať udalosti myši, mať svoju vlastnú súradnicovú sústavu a mať kurzor. Každá z nich má k dispozícii grafický kontext pre renderovanie. [20] Výhoda odľahčeného systému je, že je výrazne flexibilnejší na rozdiel od natívneho oknového systému, ktorý sa obyčajne skladá zo štvorcových komponentov. Dovoľujú vytváranie a manipuláciu s ľubovoľne tvarovanými grafickými objektmi. Pretože simulujú plnohodnotný grafický systém vnútri jediného okna, umožňujú vytváranie graficky komplexného zobrazenie bez spotrebovania množstva systémových zdrojov.

4.2.2 GEF framework

Táto kapitola sa venuje základným častiam rámca GEF, ktorými sú EditParts, požiadavky, EditPolicies a príkazy.

4.2.2.1 EditParts

EditParts sú ústrednými prvkami v GEF aplikácii. Sú kontrolórmi, ktorí špecifikujú, ako sú mapované prvky modelu na zobrazenia a ako sa tieto prvky správajú v rôznych situáciách. Obyčajne je potrebné vytvoriť EditPart triedu pre všetky triedy prvkov modelu, takže hierarchia tried pre EditParts bude rovnaká ako pre model.

V skutočnosti exitujú tri rôzne typy EditParts: GraphicalEditParts, ConnectionEditParts a TreeEditParts. GraphicalEditParts sú tie EditParts, ktoré poskytujú grafickú reprezentáciu pre model. Tieto grafické reprezentácie sú zobrazenia. ConnectionEditParts reprezentujú spojenia medzi GraphicalEditParts. TreeEditParts sa používa len na budovanie stromov z modelu za použitia SWT nástrojov pre prácu so stromami. Nie je to primárny účel grafického editoru, ale sú užitočné pre náhľad. [20]

4.2.2.1.1 Zobrazenia (Figures)

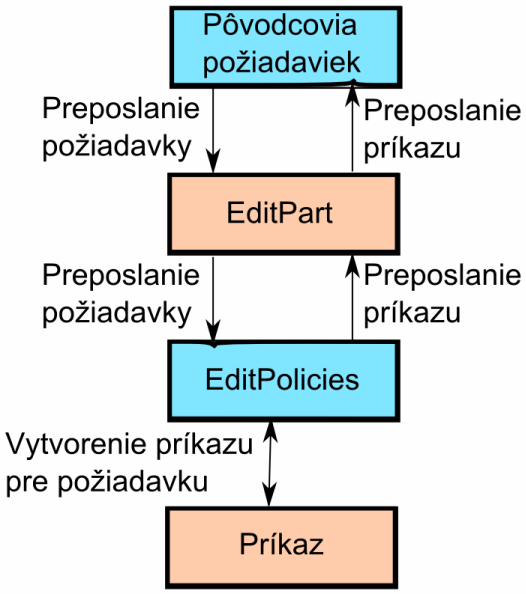
GraphicalEditParts majú podobu, ktorá je grafickou časťou modelu. GraphicalEditParts potrebujú vytvoriť zobrazenia, aktualizovať ich na základe zmien modelu a odstrániť ich, ak je to nutné pri deaktivácii EditPart.

4.2.2.1.2 Spojenia

ConnectionEditPart nie je nič iné ako GraphicalEditPart, ktoré má zdrojové a cieľové EditPart.

4.2.2.2 Požiadavky

Požiadavky sú komunikačné objekty používane v GEF. Obsahujú informáciu, ktorá môže byť nevyhnutná pre neskoršie vykonanie požiadavky. Existuje niekoľko typov požiadaviek.



Obrázok 20. Komunikačný reťazec

Obrázok 20 ukazuje typický priebeh komunikácie požiadavky a zahrnuté objekty. Nejaký objekt (typicky nástroj, akcia alebo obsluha „drag and drop“) vytvorí požiadavku. Táto požiadavka je zaslaná EditPart. EditPart nevykoná priamo požiadavku. Namiesto toho ju deleguje na EditPolicy, ktorá požiadavke rozumie. EditPolicy vytvorí príkaz pre požiadavku, ktorý sa vykoná na splnenie požiadavky.

4.2.2.3 EditPolicies

EditPolicies sú tie časti v GEF, ktoré prinášajú funkcionalitu editovania EditParts. Tento návrh je dobrým príkladom objektovo-orientovaného návrhu. EditPolicies definujú, čo je možné robiť s EditPart. EditParts bez EditPolicies by nevykonali nič. Neboli by dokonca ani schopné výberu používateľom. EditPolicies sú rovnako zodpovedné za správu odozvy (napríklad, čo sa má zobraziť, keď sa EditPart pohne alebo zmení jeho veľkosť) a majú schopnosť delegovať prácu (preposlať požiadavky) iným EditParts (napríklad svojim potomkom).

4.2.2.4 Príkazy

Príkaz je tá časť, ktorá aktuálne modifikuje model. Príkazy zjednodušujú spôsob modifikácie modelu, pretože poskytujú podporu pre obmedzenia vykonávania, „undo“ a „redo“ a kombinácie a reťazenie.

4.3 GMF

V tejto kapitole popíšeme základný princíp GMF. Životnému cyklu vývoja na GMF sa venujeme podrobne v kapitole 6.1.

Eclipse Graphical Modeling Framework (GMF) je nástroj pre vývoj grafických editorov založených na EMF a GEF. Výstavba editoru diagramov prostredníctvom GMF zahŕňa nasledovné fázy:

- Návrh EMF meta-modelu domény
- Vytvorenie grafických reprezentácií pre uzly a prepojenia diagramu
- Definíciu štruktúry diagramu a generovanie kódu

GMF môžeme rozdeliť na dva komponenty: komponent nástrojov a komponent vykonávania. Komponent nástrojov pozostáva z editorov na vytváranie a editovanie modelov popisujúcich grafický editor a generátoru na vytváranie implementácie grafických editorov. Vygenerovaný nástroj závisí na GMF komponente vykonávania a jeho účelom je vytvorenie rozšíriteľného grafického editoru.

4.3.1 Komponent vykonávania (runtime component)

Komponent vykonávania je knižnica uľahčujúca vytváranie diagramov a poskytuje množinu rozšírení používaných vygenerovaným kódom s cieľom rozšíriť správanie diagramu v priebehu vykonávania. Môžeme ho charakterizovať nasledovne:

- znovu použiteľné komponenty pre grafické editory
- štandardný model na opis vizuálnych vlastností elementov diagramu
- množina služieb rozšíriteľných pomocou deklarovaných bodov rozšírenia v čase vykonávania
- infraštruktúra na prepojenie EMF a GMF [23]

4.3.2 Odľahčené prostredie vykonávania

Je to odľahčená alternatíva k úplnému prostrediu vykonávania. Zameriava sa na čisté GEF a je veľmi jednoduchá, neobsahuje žiadne pokročilé funkcie okrem EMF a GEF.

5 Špecifikácia požiadaviek

Pri vytváraní meta-modelu sme postupovali nasledovným spôsobom. Definovali sme si požiadavky, ktoré musí editor poskytovať. V diagrame môžu byť znázornené komponenty, ktoré môžu byť navzájom poprepájané. Každý komponent je reprezentovaný abstrakciu a svojou implementáciou. Abstrakcia definuje špecifikáciu a správanie komponentu, pričom implementácia definuje konkrétnu implementáciu daného správania. Komponenty môžu byť jednoduché alebo zložené. Zložené komponenty obsahujú v sebe ďalšie komponenty, ktoré opäť môžu byť zložené alebo jednoduché. Každý komponent je charakterizovaný svojimi rozhraniami, ktoré sú buď poskytované alebo požadované. Komponenty môžu byť navzájom poprepájané prostredníctvom väzieb, t.j. požadované rozhranie jedného komponentu bude napojené na poskytované rozhranie druhého komponentu. Dôležitou funkciou je takisto prepojenie rozhraní vonkajšieho komponentu s rozhraniami svojich vnútorných subkomponentov. Dosiahneme to prostredníctvom nadefinovania špeciálnych vnútorných rozhraní, ktoré poskytujú zložené komponenty. Tieto vnútorné rozhrania sú viditeľné len pre vnútorné komponenty zloženej komponenty. Pre správne pochopenie si uveďme príklad. Poskytované rozhranie zloženého komponentu môže byť realizované jednou z jeho vnútorných častí. To znamená, že je potrebné namapovať poskytované rozhranie zloženého komponentu na poskytované rozhranie jedného z jeho vnútorných komponentov. Dosiahneme to prostredníctvom vytvorenia portu zloženého komponentu. Na port z vonkajšej strany umiestnime príslušné poskytované rozhranie a na vnútornej strane portu vytvoríme špeciálne vnútorné požadované rozhranie, na ktoré potom pripojíme poskytované rozhranie vnútorného komponentu. Port nám teda prepojí tieto dve požadované rozhrania. V našom modeli toto špeciálne vnútorné rozhranie realizujeme pomocou zástupcu (proxy).

Editor bude postavený na platforme Eclipse, konkrétne na EMF a GMF frameworkoch. Meta-model bude nadefinovaný pomocou EMF, ostatné časti editora budú vytvorené v GMF. Bude zahrnutá podpora hierarchickej dekompozície komponentov a špecifikácia jednoduchých vlastností komponentov. Editor ďalej bude podporovať XMI serializáciu a deserializáciu. Editor zabezpečí kontrolu vytváraných diagramov, aby nemohli vzniknúť nezmyselné alebo chybné diagramy.

6 Návrh aplikácie

V tejto kapitole si popíšeme návrh editora – grafického modelovacieho nástroja. Editor bude slúžiť pre návrh komponentových systémov. Editor umožní používateľovi vytvárať diagramy rozmiestnenia a prepojenia komponentov, ich hierarchickú kompozíciu a dekompozíciu a špecifikáciu jednoduchých vlastností komponentov a ich prepojení. Pri návrhu editora budeme vychádzať zo špecifikácie požiadaviek uvedenej v kapitole 5.2.

Pri vytváraní editora budeme postupovať podľa diagramu znázorneného na obrázku 21. Najprv si vytvoríme nový GMF projekt, nasledovne vytvoríme EMF Ecore meta-model, ktorého podstata je popísaná v kapitole 6.3. Nadefinujeme si grafickú podobu prvkov zobrazovaných v editore a namapujeme jednotlivé objekty modelu na ich grafickú podobu. Nadefinujeme aj definíciu nástrojov pre editovanie grafických prvkov. Tieto tri modely (t.j. model domény, model grafickej definície a model definície nástrojov) skombinujeme pomocou EMF. Na základe vytvoreného spojeného modelu vygenerujeme kód potrebný na vytvorenie editora. Posledným krokom je samotné vygenerovanie editora do jeho finálnej podoby. Všetky tieto kroky môžeme v požadovanej miere editovať a prispôsobiť tým vzhľad a funkčnosť výsledného editora.

6.1 Životný cyklus vývoja založeného na GMF

V tejto kapitole sa budeme venovať jednotlivým krokom životného cyklu vývoja založeného na GMF.

Diagram znázornený na obrázku 21 ilustruje hlavné komponenty a modely použité počas vývoja založeného na GMF. Jadrom GMF je model grafickej definície. Tento model obsahuje informácie vzťahujúce sa ku grafickým elementom, ktoré sa zobrazia počas behu v prostredí založenom na GEF. Grafické elementy nemajú priame prepojenie na modely domény. Budú pre ne poskytovať grafickú reprezentáciu a editovanie. Definícia nástrojov je voliteľná a používa sa na návrh palety, menu, panelu nástrojov a pod. Cieľom GMF je umožniť použitie grafickej definície pre rôzne domény. Dosahuje sa to použitím samostatného modelu mapovania, ktorý spája grafickú definíciu a definíciu nástrojov s vybraným modelom domény. Následne, ako sú definované príslušné mapovania, GMF poskytuje model generovania, ktorý definuje detaily implementácie pre fázu generovania. Na základe modelu generovania sa vytvorí model prostredia vykonávania diagramov (alebo model notácie). Počas práce s diagramom prostredie vykonávania spája model notácie a model domény a rovnako poskytuje perzistenciu a synchronizáciu týchto modelov.

V ďalšom texte bližšie popíšeme jednotlivé kroky životného cyklu vývoja založeného na GMF.

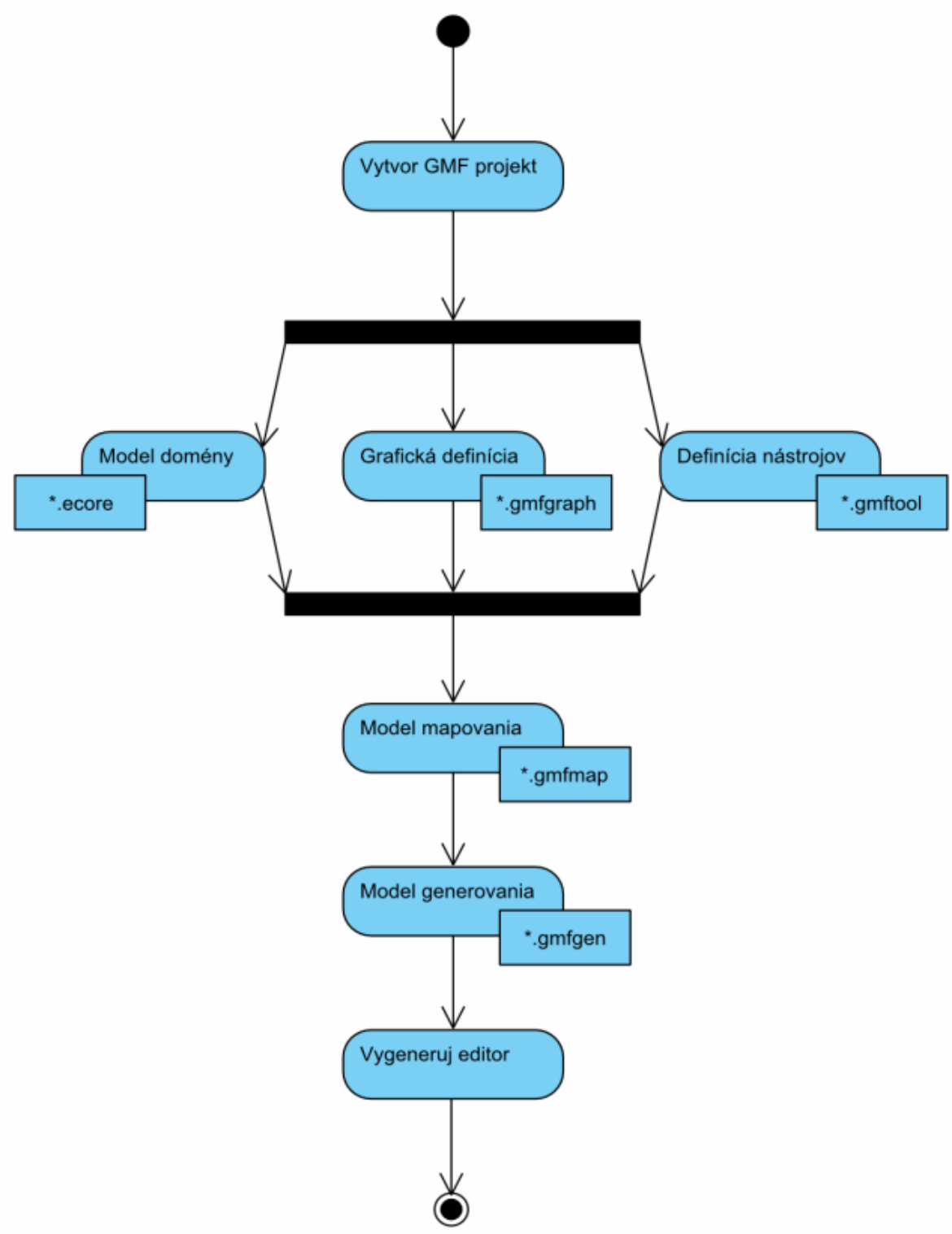
6.1.1 Definícia modelu domény

Ako model domény sa používa Ecore model z EMF (viď kapitola 5.3).

6.1.2 Grafická definícia

Model grafickej definície sa používa na definovanie prvkov, uzlov, prepojení a pod., ktoré budú zobrazené v diagrame. V tomto kroku na základe Ecore modelu domény nadefinujeme elementy diagramu, ich vlastnosti a možnosti vzájomného prepojenia (obmedzenia prepojenia niektorých entít navzájom, typy prepojení medzi entitami a pod.). Tento model definuje vizuálne aspekty

vygenerovaného editoru. Pri jeho vývoji postupujeme nasledovne. Vytvoríme grafické prvky, ktoré chceme zobrazovať v diagrame. Vytvoríme uzly, ktoré chceme znázorňovať v diagrame. Vytvoríme prepojenia, aby každý uzol zodpovedal nejakému grafickému prvku.



Obrázok 21. Životný cyklus vývoja založeného na GMF

6.1.3 Definícia nástrojov

Používa sa na špecifikovanie palety, vytvorenie nástrojov, akcií, menu a pod. pre grafické elementy.

6.1.4 Definícia mapovania

Prostredníctvom tohto modelu navzájom previažeme predchádzajúce tri modely, t.j. model domény, grafický model a model nástrojov. Je to kľúčový model a použije sa ako vstup transformačného kroku, ktorý vytvorí finálny model, model generovania. Definuje mapovanie medzi biznis logikou (EMF model domény) a vizuálnym modelom (grafickou a nástrojovou definíciou). GMF automaticky určí, ktoré elementy modelu majú byť mapované na ktoré vizuálne elementy. Avšak GMF súbor definície mapovania môžeme editovať a pridať vlastné úpravy.

6.1.5 Generovanie kódu

Keď sú už definované grafické elementy a mapovanie, môžeme vygenerovať kód potrebný na vytvorenie editora. Najprv vytvoríme model generovania, aby sme mohli nastaviť vlastnosti generovania kódu. Tento model je podobný modelu EMF „genmodel“, ktorý je popísaný v kapitole 4.1.3.4. Tento model môžeme upraviť pre naše potreby. Potom vygenerujeme kód diagramu, t.j. grafický editor.

6.1.6 Spustenie diagramu

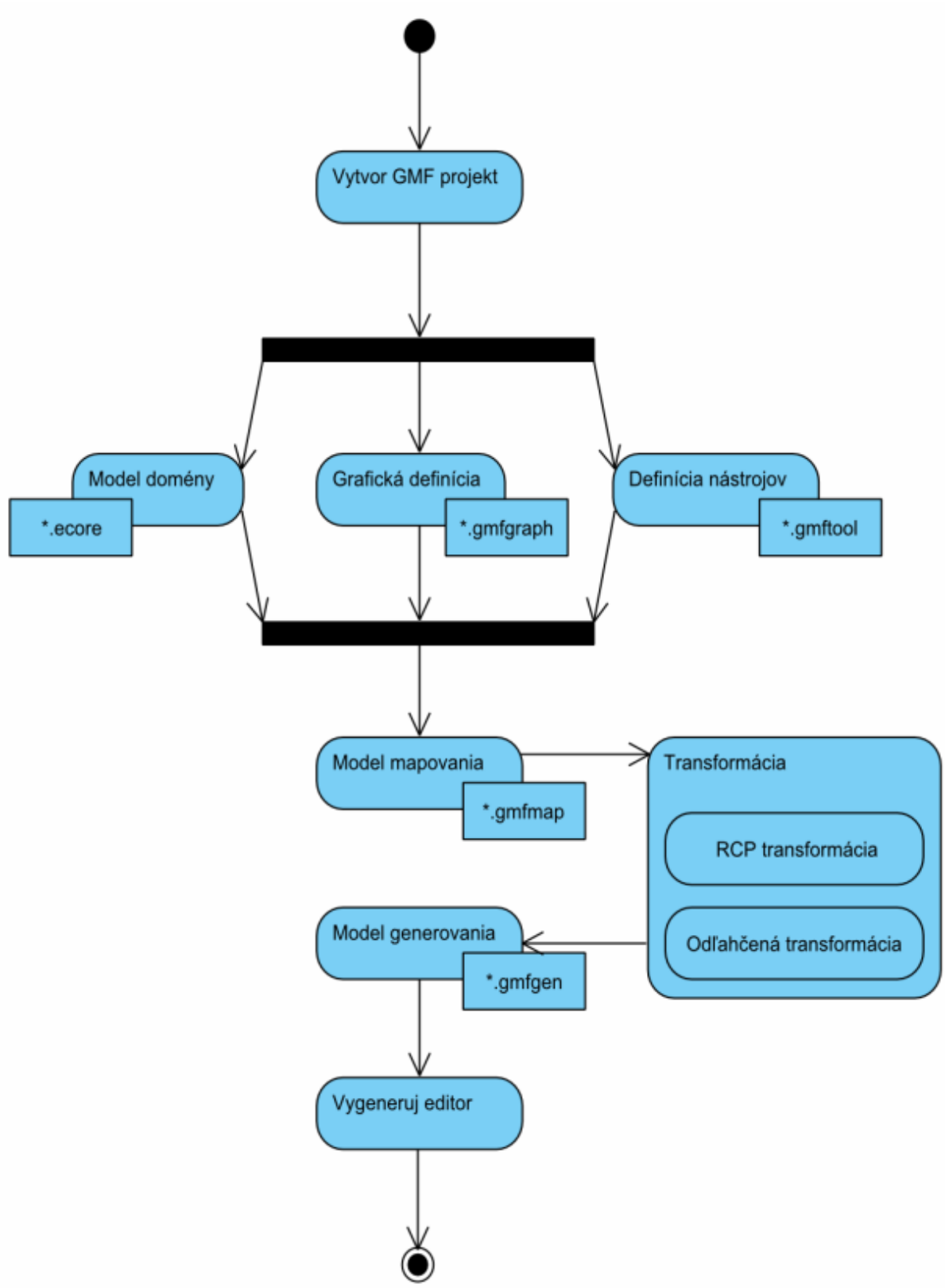
Prostredie vykonávania používa na zobrazovanie vlastný model notácie a uchováva grafické komponenty diagramu. Potom, čo sme vygenerovali zásuvný modul (editor) potrebný pre náš diagram, tak vytvoríme novú pracovnú plochu a editor spustíme. Konfiguráciu spustenia môžeme zmeniť na minimálnu konfiguráciu, ktorá zahŕňa len vygenerovaný zásuvný modul a jeho závislosti pre správne fungovanie.

GMF nám umožňuje väčšinu činností automatizovať. Takže v prípade, ak sa rozhodneme nepoužiť pokročilé možnosti (napríklad validácia), ktoré nám GMF ponúka, vytvorenie grafickej definície a definície mapovania na vybranú doménu môže byť použité na automatické vygenerovanie očakávanej základnej funkcionality v editore postavenom na kombinácii EMF a GEF.

6.2 Odľahčené prostredie vykonávania

V tejto kapitole popíšeme alternatívnu možnosť generovania – podporu odľahčeného prostredia vykonávania a Rich Client Platform (RCP). Eclipse RCP je platforma pre budovanie a nasadzovanie „tlstých klientskych“ (rich client) aplikácií.

Postup pri vývoji takýchto aplikácií je podobný štandardnému postupu. Najprv nadefinujeme všetky potrebné modely a potom môžeme vytvoriť model generovania. Nastavíme, že chceme vytvoriť RCP a potom použijeme odľahčenú transformáciu. Cieľom odľahčeného prostredia vykonávania je možnosť spolupráce s diagramami používanými plným prostredím vykonávania, čo znamená, že oba používajú rovnaký model notácie. Zároveň poskytuje minimálne závislosti a malú veľkosť pre spustenie takýchto editorov diagramov založených na RCP. V prípadoch, kde nie sú potrebné viaceré náhľady na jeden model domény, je pre zjednodušenie aplikácie možné spojiť diagram a model do jedného súboru.

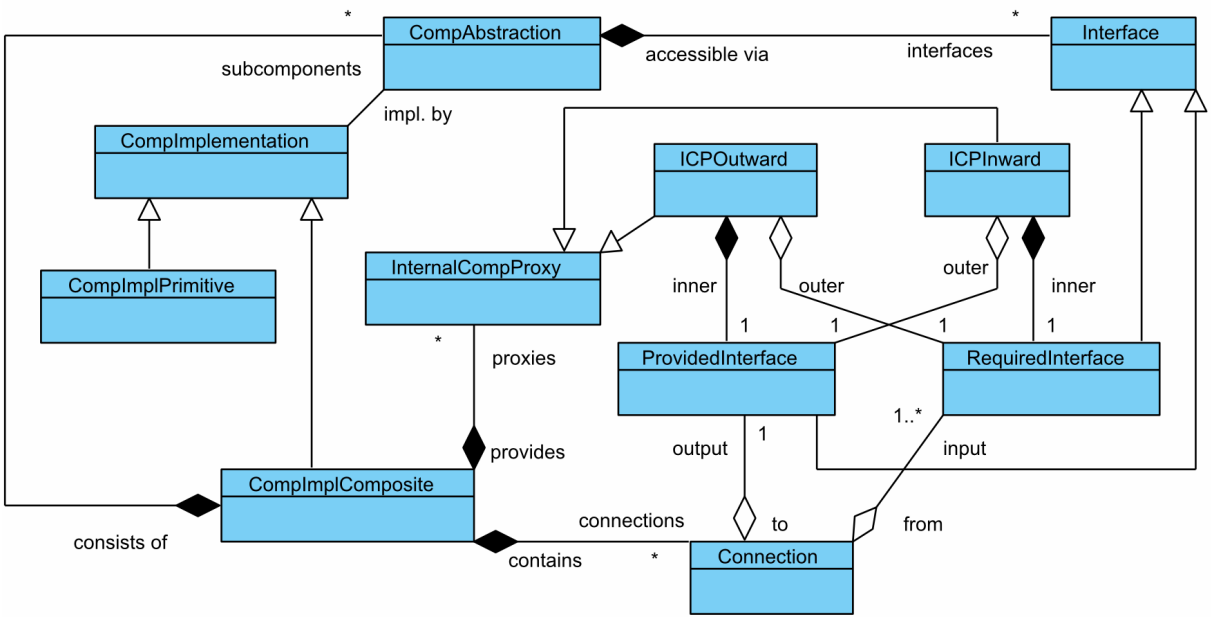


Obrázok 22. Životný cyklus vývoja založeného na GMF – odľahčené prostredie vykonávania

6.3 Návrh meta-modelu

Na Obrázok 23 je znázornený meta-model popisovanej aplikácie. Trieda CompAbstraction reprezentuje abstrakciu komponentu, trieda CompImplementation reprezentuje implementáciu komponentu, trieda CompImplPrimitive reprezentuje jednoduchý komponent, trieda CompImplComposite reprezentuje zložený komponent, trieda Interface reprezentuje rozhranie, trieda ProvidedInterface reprezentuje poskytované rozhranie, trieda RequiredInterface reprezentuje požadované rozhranie, trieda Connection reprezentuje prepojenie rozhraní, trieda InternalCompProxy reprezentuje zástupcu (proxy), trieda ICPOutward reprezentuje prepojenie externého poskytovaného špeciálneho rozhrania zloženého komponentu s vnútorným požadovaným rozhraním subkomponentu a nakoniec trieda ICPOutward reprezentuje prepojenie externého požadovaného špeciálneho rozhrania zloženého komponentu s vnútorným poskytovaným rozhraním subkomponentu. Pri návrhu tohto meta-modelu sme čerpali z [28].

Tento model nezahŕňa obmedzenie prepojenia rozhraní subkomponentov len v rámci jedného zloženého komponentu ani správne prepojenia ICPInward a ICPOutward. Umožňuje prepojenie rozhraní subkomponentov aj v rámci susedných komponentov a taktiež umožňuje prepájanie akýchkoľvek rozhraní pomocou väzieb ICPInward a ICPOutward. Túto skutočnosť musíme overovať pri prepájaní jednotlivých subkomponentov a jednotlivých rozhraní. EMF nám k tomu poskytuje adaptéry, ako bolo spomínané v kapitole 4.1.4.1.



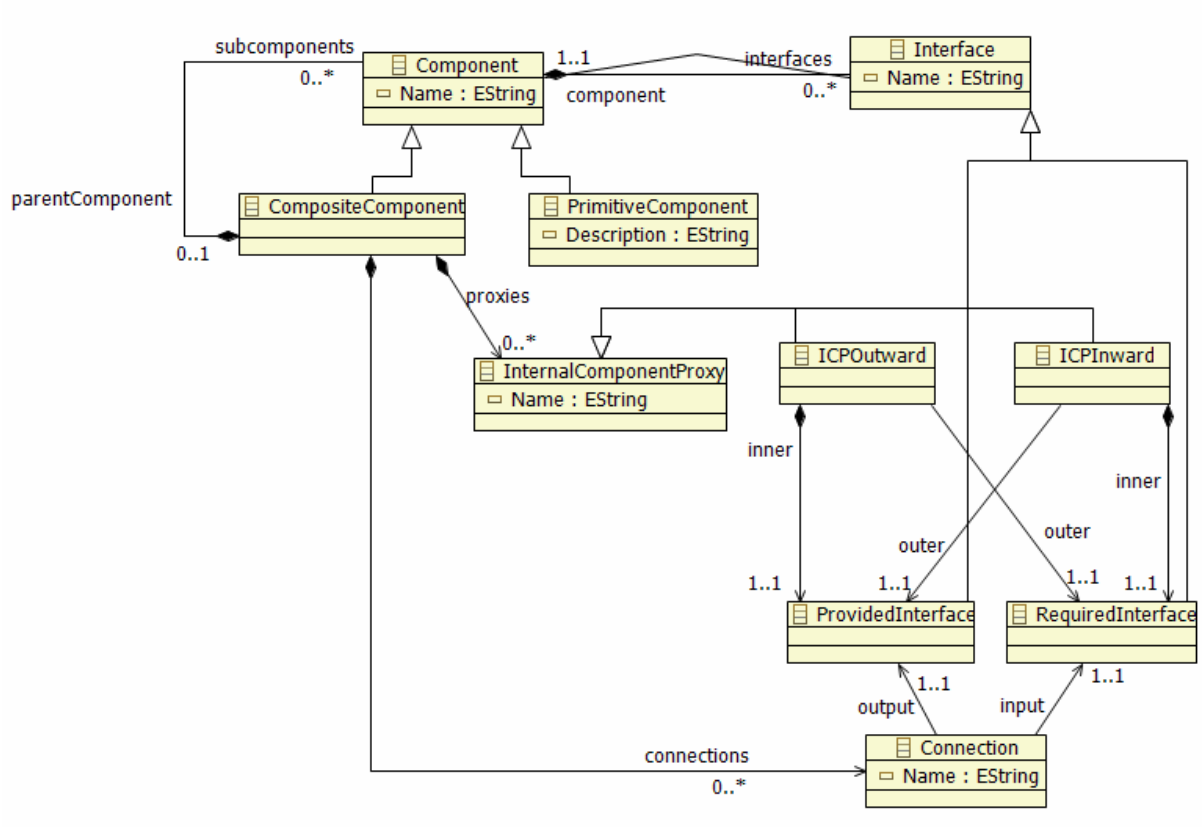
Obrázok 23. Návrh meta-modelu

7 Implementácia

V tejto kapitole popíšeme kroky, ktorými sme postupovali pri vytváraní editora. Po základnom oboznámení sa s frameworkami EMF a GMF sme pristúpili k zostrojeniu Ecore meta-modelu. Ecore meta-model môžeme vytvárať rôznymi spôsobmi, ako bolo popísané v kapitole 4.1.1. Na tento účel sa v GMF nachádza grafický editor, ktorý nám umožňuje grafický návrh modelu. Potom sme postupne prešli všetkými fázami životného cyklu vývoja založeného na GMF a zostrojili sme funkčnú základnú verziu editora. Následne sme postupne s nadobúdajúcimi skúsenosťami a vedomosťami pridávali pokročilé vlastnosti výsledného editora.

7.1 Ecore meta-model

V prvej časti sme vytvorili Ecore meta-model prostredníctvom grafického editora, ktorý je súčasťou balíka GMF. Tento meta-model je znázornený na obrázku 24. Oproti meta-modelu navrhnutému v kapitole 5.3 sme tento model upravili pre potreby a obmedzenia, ktoré nám EMF a GMF poskytujú. Z modelu bola odstránená entita ComponentAbstraction, pretože pre potreby editora sa ukázala ako zbytočná. Bola pridaná väzba parentComponent medzi CompositeComponent a Component, ktorá slúži k tomu, aby sme boli schopní pohybovať sa po hierarchickej štruktúre komponentov. Ďalšou z pridaných väzieb je väzba component medzi Interface a Component, ktorú sme pridali, aby rozhranie obsahovalo informáciu o tom, ku ktorému komponentu prislúcha. Tieto väzby sa využívajú pri kontrole vytvárania väzieb medzi entitami, aby editor podporoval vytváranie len správnych editorov, ako bude popísané nižšie. Boli pridané atribúty Name, ktoré, ako už z názvu vyplýva, hovoria o názvoch vyskytujúcich sa entít. Ďalším pridaným atribútom je atribút Description, ktorý nesie informáciu o popise primitívneho komponentu. Poslednou dôležitou zmenou je násobnosť väzby input medzi Connection a RequiredInterface, ktorá sa zmenila z 1..* na 1..1. V tomto prípade bude pre každé spojenie RequiredInterface a ProvidedInterface použitá nová väzba.

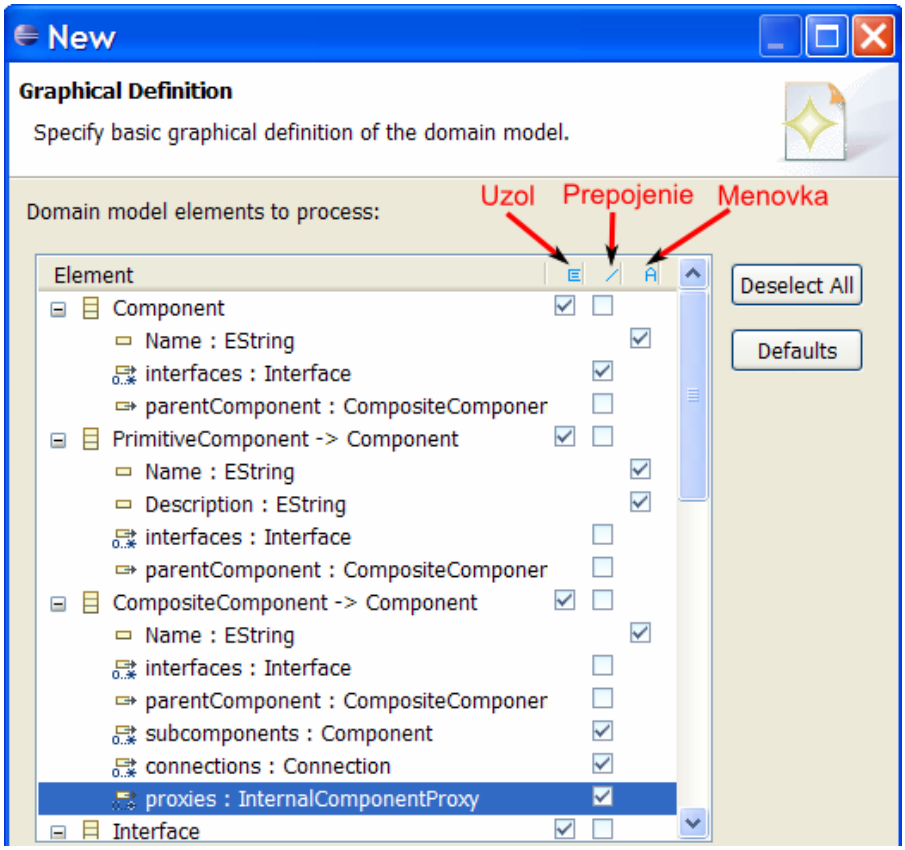


Obrázok 24. Ecore meta-model

Na základe Ecore meta-modelu sme vygenerovali model generovania. Z modelu generovania sme vygenerovali triedy modelu, ako bolo popísané v kapitole 4.1.3.1. Následne sme vygenerovali triedy poskytujúce podporu pre editovanie modelu, čomu sa podrobne venujeme v kapitole 4.1.6.

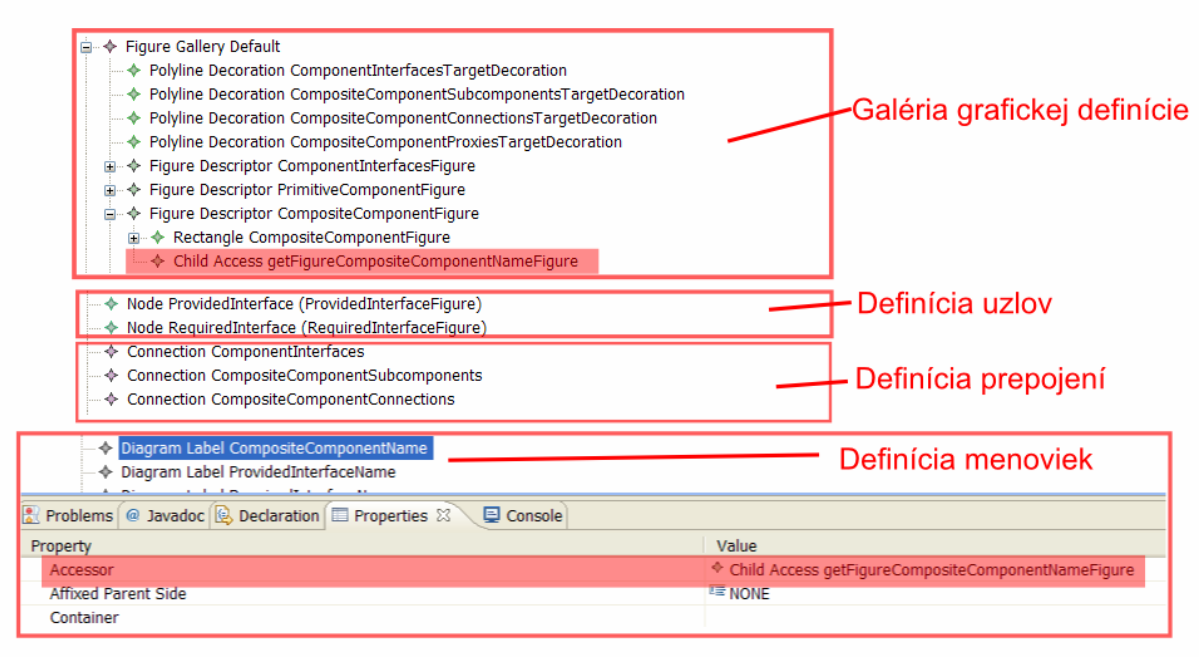
7.2 Grafická definícia

Ďalším krokom bolo nadefinovanie grafickej definície, t.j. ako bude vyzerať vizuálna stránka editora, ako budú vyzerať jednotlivé triedy a prepojenia medzi nimi. Na vytvorenie modelu grafickej definície je vhodné použiť sprievodcu, kde určíme, aké grafické prvky zodpovedajú jednotlivým triedam, prepojeniam a atribútom v Ecore meta-modeli. Ako vidno z obrázku 25, trieda Component bude reprezentovaná ako uzol, jej atribút Name bude reprezentovaný ako menovka a väzba interfaces bude reprezentovaná ako prepojenie. Je možné reprezentovať triedu z Ecore meta-modelu ako prepojenie, takýmto spôsobom sme reprezentovali triedy Connection, ICPInward a ICPOutward. Tento sprievodca nám vygeneruje základné nastavenia, ktoré je potrebné upraviť pre našu potrebu.



Obrázok 25. Sprievodca grafickou definíciou

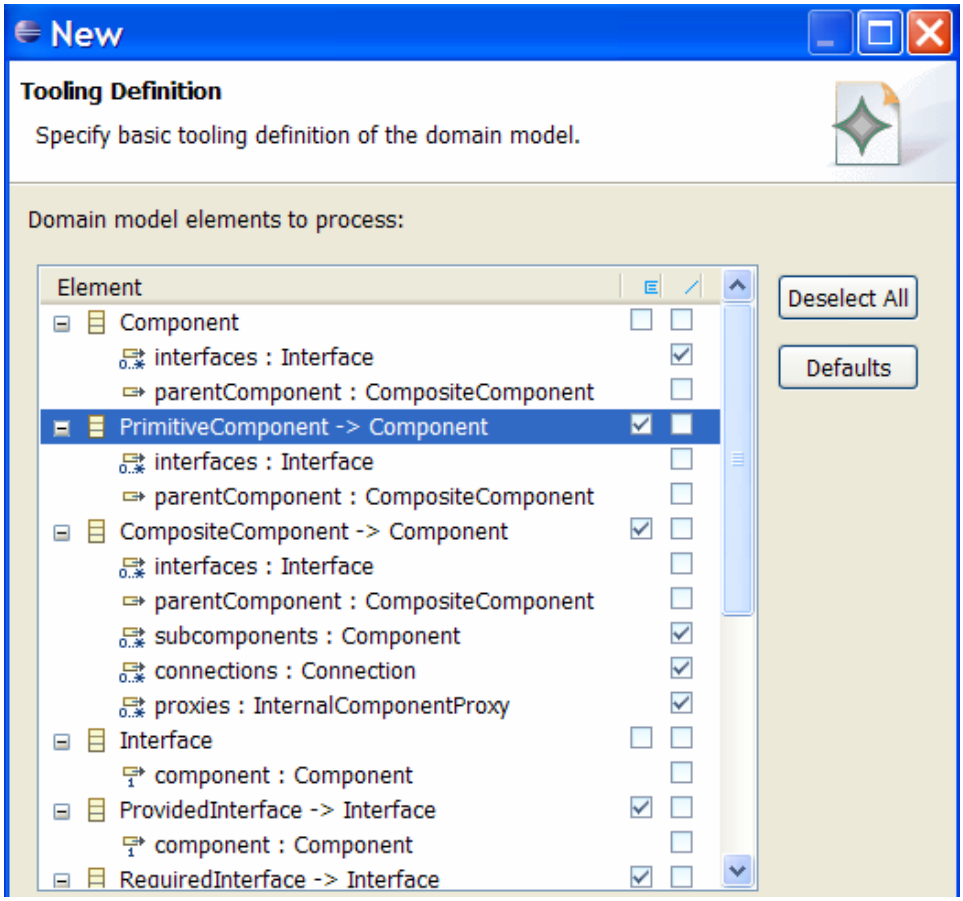
Grafická definícia sa skladá z piatich častí, a to galéria grafickej definície (Figure gallery), nastavenie uzlov, prepojení, menoviek a oddelení, ako je znázornené na obrázku 26. V galérii nastavíme výzor jednotlivých uzlov, prepojení a menoviek, t.j. napríklad uzol Connection bude reprezentovaný ako obdĺžnik so zaoblenými rohmi a bude obsahovať menovku Name, ktorá sa zobrazí vnútri tohto obdĺžnika. V nastavení uzlov priradíme k jednotlivým uzlom ich grafickú reprezentáciu z galérie grafickej definície, podobne postupujeme i pre nastavenie prepojení, menoviek a oddelení. Pri nastavení menoviek bolo potrebné zabezpečiť príslušnosť danej menovky k uzlu a prístup k menovke, na obrázku 26 to je znázornené červenými vyplnenými obdĺžnikmi. V našom editore sme nevyužili oddelenia, tento prístup by sa využil pri klasickom komponentovom diagrame, kde subkomponenty zloženého komponentu sú zobrazené vnútri tohto zloženého komponentu. Zdôvodnenie, prečo sme si vybrali druhý prístup, je uvedené v kapitole 6.4.



Obrázok 26. Vlastnosti grafickej definície

7.3 Definícia nástrojov

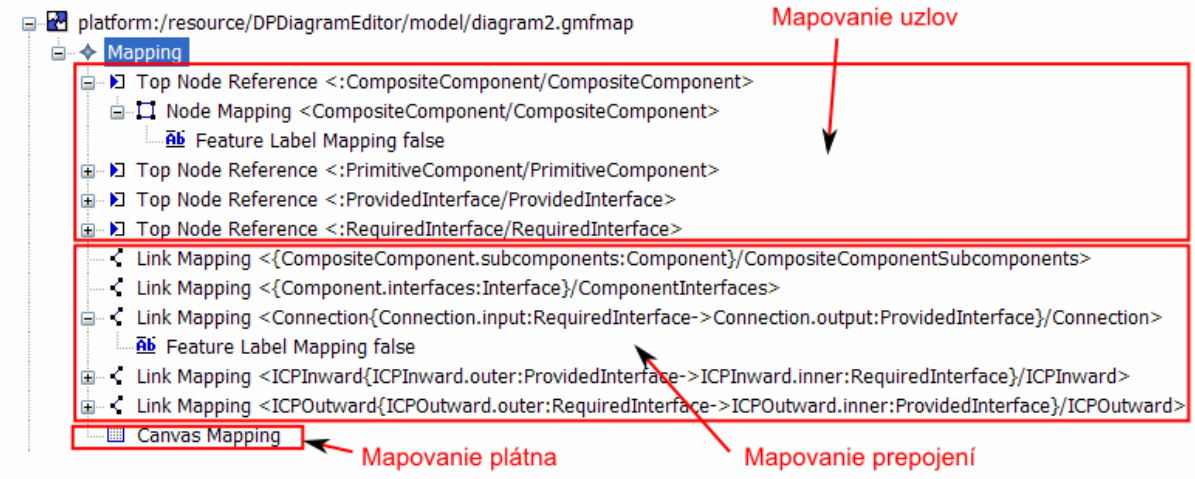
Podobne ako u grafickej definície je i u definície nástrojov vhodné použiť sprievodcu. Ako je znázornené na obrázku 27, nastavíme, ktoré triedy sa majú zobraziť v paneli nástrojov ako uzly a ktoré ako prepojenia. Panel nástrojov si rozdelíme na dve časti kvôli prehľadnosti, časť obsahujúca vytváranie uzlov a časť obsahujúca vytváranie liniek.



Obrázok 27. Sprievodca definíciou nástrojov

7.4 Definícia mapovania

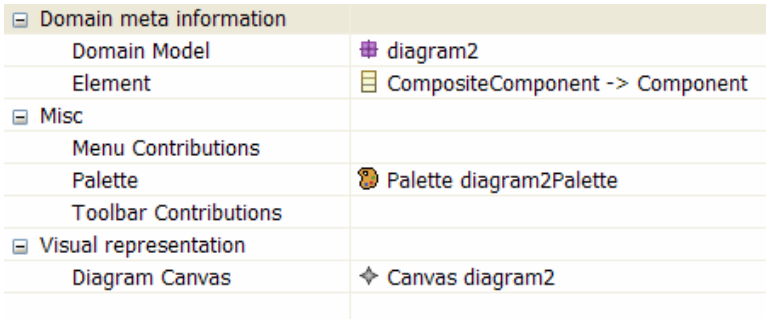
V tomto modeli navzájom prepojíme Ecore meta-model, grafickú definíciu a definíciu nástrojov.



Obrázok 28. Definícia mapovania

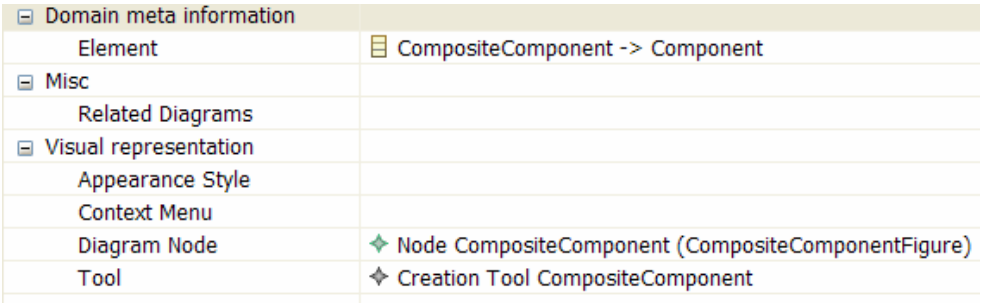
V prípade, ak použijeme sprievodcu, toto prepojenie sa nastaví automaticky. Ak sprievodcu nevyužijeme, nadefinujeme to v časti mapovania plátna, ako je znázornené na obrázku 29. Model domény nastavíme na Ecore meta-model, ako element nastavíme CompositeComponent, t.j. uzol,

ktorý bude reprezentovať samotný diagram. K palete priradíme paletu, ktorú sme vytvorili v definícii nástrojov a k plátnu diagramu priradíme plátno, ktoré sme vytvorili v grafickej definícii.



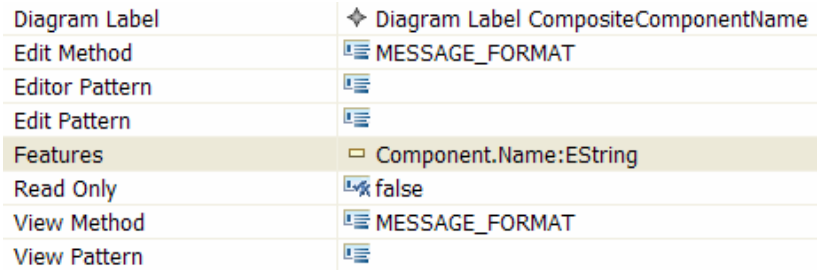
Obrázok 29. Vlastnosti mapovania plátna

Mapovanie uzlov sa nastavuje prostredníctvom Top Node Reference a vnoreného Node Mapping, ako ukazuje obrázok 28. U každého uzlu nastavíme, ktorý element z Ecore meta-modelu k nemu prislúcha (atribút Element), grafickú reprezentáciu uzlu z grafickej definície (atribút Diagram Node) a zodpovedajúci nástroj v paneli nástrojov z definície nástrojov (atribút Tool). Tieto vlastnosti mapovania uzlu znázorňuje obrázok 30.



Obrázok 30. Vlastnosti mapovania uzlu

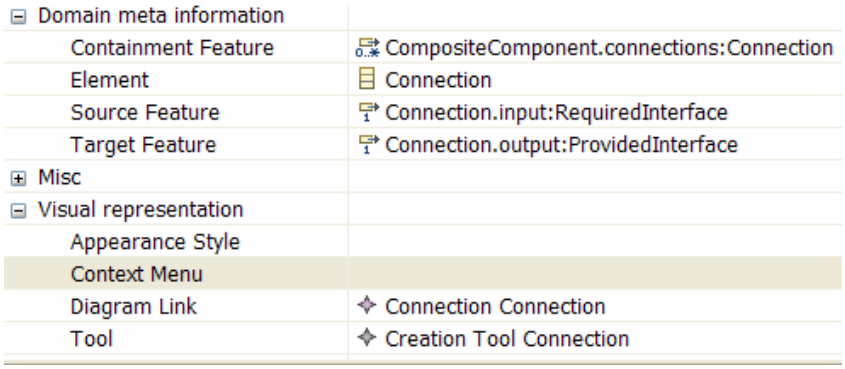
K daným uzlom a prepojeniam, u ktorých sme v grafickej definície nadefinovali menovky, je potrebné nastaviť mapovanie týchto menoviek. Napríklad pre uzol CompositeComponent pridáme Feature Label Mapping, ako ukazuje obrázok 28 a nastavíme vlastnosti tohto mapovania, t.j. ako menovku (atribút Diagram Label) vyberieme príslušnú menovku nadefinovanú v grafickej definícii a ako vlastnosť (atribút Feature) vyberieme atribút príslušného uzlu z Ecore meta-modelu, v tomto prípade to je atribút Name, ako ukazuje obrázok 31.



Obrázok 31. Vlastnosti mapovanie menovky

Pri nastaveniach prepojení sme postupovali nasledovne. Vytvorili sme si Link Mapping a nastavili sme vlastnosti tohto mapovania, t.j. v ktorej triede z Ecore meta-modelu je táto väzba

obsiahnutá (atribút Containment Feature), ktorá trieda zodpovedá tomuto prepojeniu (atribút Element), zdroj (atribút Source Feature) a cieľ (atribút Target Feature) tejto väzby, t.j. ktoré entity väzba prepája. Podobne, ako u uzlov, tak aj u prepojení nastavíme príslušnú grafickú definíciu (atribút Diagram Link) a príslušný nástroj v palete nástrojov (atribút Tool). Pre prepojenia, ktoré sú v Ecore meta-modeli reprezentované ako prepojenia (napríklad prepojenie interfaces), t.j. nie sú reprezentované ako triedy (napríklad trieda Connection), stačí nastaviť cieľ tejto väzby (atribút Target Feature) a atribúty Containment Feature, Element a Source Feature nastavovať nemusíme. Tieto nastavenia mapovania prepojenia sú znázornené na obrázku 32.

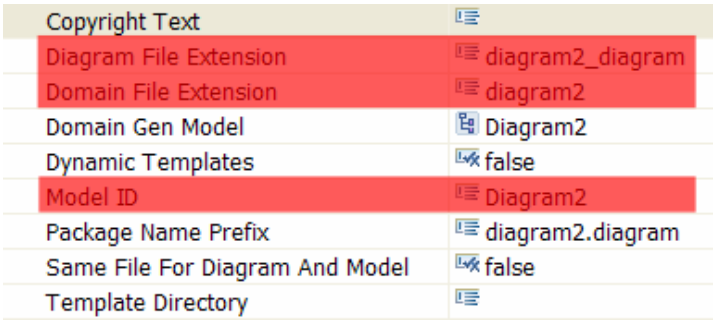


Obrázok 32. Vlastnosti mapovania prepojenia

V tejto časti sme sa podrobnejšie venovali definícii mapovania. Prínosom je, že je táto definícia popísaná podrobne na jednom mieste. Ako bude spomínané v kapitole 8.1, tak dokumentácia ku GMF je veľmi slabá a všetky tieto informácie sme museli získavať z rozličných zdrojov [20],[26],[29],[30],[31] a následne experimentovať, ako namapovať tieto nastavenia pre náš konkrétny prípad.

7.5 Model generovania a vygenerovanie editora

V tomto kroku sme vygenerovali model generovania z modelu mapovania a EMF modelu generovania, ktorý sme vytvorili z Ecore meta-modelu, ako je popísané v kapitole 6.1.



Obrázok 33. Základné nastavenia GMF modelu generovania

V modeli generovania je možné vykonávať pokročilé úpravy a nastavenia budúceho editora, spomeňme aspoň základné nastavenia, ktoré sú zvýraznené červeným obdĺžnikom na obrázku 33. Atribút Diagram File Extension slúži na nastavenie koncovky súboru, v ktorom bude uložený diagram vytvorený v našom novom editore, atribút Domain File Extension slúži na nastavenie koncovky

súboru, v ktorom bude uložený model domény diagramu a atribút Model ID nám určuje, ako sa bude menovať náš diagram v menu Eclipse (t.j. keď vyberieme v Eclipse položky z menu File -> New). Z modelu generovania vygenerujeme výsledný editor, ktorému zodpovedá vygenerovaný projekt s koncovkou *.diagram. V tomto kroku máme hotovú prvú funkčnú verziu nášho editora.

7.6 Pokročilá úprava editora

V tejto časti popíšeme ďalšie kroky, ktoré bolo potrebné vykonať, aby sme editor upravili do finálnej verzie. Problematickými časťami v Ecore meta-modeli sú InternalComponentProxy a Connection.

7.6.1 Obmedzenia väzieb

Základná vygenerovaná verzia editora dovoľuje vytvárať prepojenie i medzi uzlami, medzi ktorými by finálny editor z logického hľadiska väzby povoľovať nemal. Editor povoľuje všetko, čo mu umožňuje definovaný Ecore meta-model. Umožňuje napríklad prepájať pomocou väzby Connection akékoľvek RequiredInterface s akýmkoľvek ProvidedInterface. Avšak editor by mal podporovať prepojenie len tých požadovaných a poskytovaných rozhraní, ktoré sú na jednej úrovni zanorenia a zároveň patria komponentom, ktoré majú spoločný rodičovský komponent, t.j. zložený komponent by mal obsahovať len väzby, ktoré prepájajú len rozhrania subkomponentov zloženého komponentu a nie rozhrania susedných komponentov. Podobne je potrebné obmedziť aj väzby ICPInward a ICPOutward, aby editor povoľoval len prípustné prepojenia vnútorných a vonkajších rozhraní. V tejto kapitole je dôležité uviesť podstatný poznatok, na ktorý sme prišli počas vytvárania editora. Tieto obmedzenia sa dajú riešiť dvoma spôsobmi a tieto spôsoby sú samostatné a navzájom nezávislé.

7.6.1.1 EMF spôsob

Prvým spôsobom je kontrolovať správne vytváranie väzieb v EMF. Tento spôsob si uvedieme len stručne, pretože nemá pre náš editor príliš praktický význam, ale bol súčasťou procesu implementácie editora. Súčasťou EMF.Edit je príkazový framework, ako opisuje kapitola 4.1.6.3. EMF.Edit príkazy slúžia ne editovanie objektov modelu (t.j. EObject). Existuje viacero druhov príkazov, pre náš prípad je vhodný SetCommand, ktorý nastaví hodnotu atribútu alebo referencie objektu. Vytvoríme si vlastný príkaz, ktorý dedí od triedy SetCommand a preťažíme metódu doExecute(), do ktorej napíšeme obmedzenia, ktoré musia byť splnené, aby editor povolil nastavenie referencie, t.j. aby povolil príslušné prepojenie.

V prípade väzby Connection kontrolujeme, či komponenty prepájaných rozhraní patria spoločnému zloženému komponentu. V ďalšom kroku musíme zabezpečiť, aby sa pri nastavovaní atribútov alebo referencií daného komponentu zavolal i náš príkaz. V prípade väzby Connection musíme preťažiť metódu createSetCommand() v príslušnom poskytovateľovi funkčnosti, t.j. trieda ConnectionItemProvider. Problematiku poskytovateľov funkčnosti opisuje kapitola 4.1.6.2. Takýmto spôsobom by sme postupovali i pre väzby ICPInward a ICPOutward. Avšak pri testovaní tohto riešenia sme dospeli k záveru, že tento prístup je funkčný len pre EMF editor a nefunguje pre GMF editor, resp. vo vygenerovanom GMF editore je táto funkčnosť zabezpečená len v paneli vlastností (Properties) a nie v grafickom okne editora. Keďže sa túto cesta ukázala ako nepoužiteľná, bolo

potrebné prísť na iný spôsob, ako zabezpečiť obmedzenia pri vytváraní väzieb. Tento spôsob popíšeme v nasledujúcej kapitole.

7.6.1.2 GMF spôsob

Ako sme spomínali v predchádzajúcej kapitole, bolo treba nájsť spôsob, ktorý by fungoval i pre GMF. V tomto prípade budeme kontrolovať obmedzenia väzby pri jej vytváraní. Vo vygenerovanom editore je potrebné upraviť triedy, ktoré zabezpečujú vznik väzieb a úpravu väzieb. Jedná sa konkrétne o triedy AAACreateCommand a AAAReorientCommand, pričom AAA reprezentuje konkrétny názov väzby. Ukážme si tento postup na konkrétnom príklade pre väzbu Connection. Zameriame sa na triedy ConnectionCreateCommand a ConnectionReorientCommand v balíčku diagram2.diagram.edit.commands. Trieda ConnectionCreateCommand obsahuje metódu `boolean canExecute()`, ktorá kontroluje, či je možné túto väzbu vytvoriť alebo nie. A práve túto metódu potrebujeme upraviť.

Je potrebné skontrolovať, či prepájané RequiredInterface a ProvidedInterface patria k nejakým komponentom, t.j. či existuje väzba interfaces medzi nejakým komponentom a príslušným rozhraním. Dosiahneme to pomocou testu (`getSource().getComponent() == null`) pre počiatok väzby a (`getTarget().getComponent() == null`) pre koniec väzby. Týmto sme skontrolovali, že obe rozhrania patria nejakým komponentom, t.j. že sa nejedná o samostatne stojace rozhrania. Následne je potrebné skontrolovať, či prepájame rozhrania komponentov, ktoré patria spoločnému zloženému komponentu. Dosiahneme to podobným spôsobom, a to testom (`getSource().getComponent().getParentComponent() != getTarget().getComponent().getParentComponent()`).

Tento prístup umožňuje prepájať pomocou väzby Connection i rozhrania, ktoré patria tomu istému komponentu. Ako príklad využitia tejto možnosti si uveďme nasledovnú situáciu. Komponent poskytuje nejaké triedenie, napríklad triedenie BubbleSort. Komponent zároveň na svoju funkčnosť požaduje nejaké triedenie. Štandardne sú teda tieto dve rozhrania prepojené, komponent je schopný existovať aj samostatne. Ak sa však v systéme vyskytuje nejaký komponent, ktorý poskytuje efektívnejšie triedenie, napríklad QuickSort, tak prvý komponent môže využiť poskytované rozhranie tohto druhého komponentu.

Ďalej musíme umožniť prepájanie špeciálnych vnútorných rozhraní zloženého komponentu s rozhraniami jeho subkomponentov, ako je popísané v kapitole 6.3. V našom prípade máme väzbu Connection nadefinovanú tak, že prepája RequiredInterface s ProvidedInterface a nie naopak. Je teda potrebné skontrolovať, či príslušné poskytované, resp. požadované rozhranie nie je súčasťou nejakého zástupcu (proxy) zloženého komponentu. Rozlišujeme teda dva prípady: prepojenie vnútorného požadovaného rozhrania rodičovského komponentu s poskytovaným rozhraním subkomponentu a prepojenie požadovaného rozhrania subkomponentu s vnútorným poskytovanyým rozhraním rodičovského komponentu. Postupne prechádzame jednotlivých zástupcov rodičovského komponentu prostredníctvom volania `for (InternalComponentProxy i : c.getProxies())`. Pre každého zástupcu skontrolujeme jeho vnútorné rozhranie. Ak sa toto rozhranie zhoduje s rozhraním, pre ktoré kontrolujeme možnosť vytvorenia väzby Connection, tak povolíme vytvorenie tejto väzby.

Ďalej je potrebné obmedziť vytváranie väzieb ICPInward a ICPOutward. Postupujeme podobne. Otestujeme, či príslušné prepájané rozhrania patria nejakému komponentu, zároveň

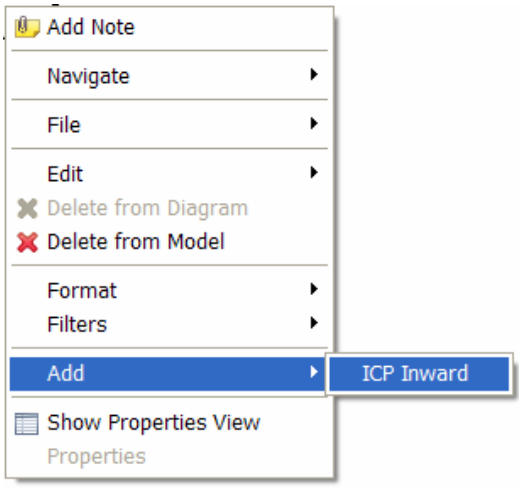
musia tieto rozhrania patriť rovnakému komponentu a tento komponent musí byť zložený komponent.

7.6.2 Automatizované vytváranie zástupcov (proxy)

V prvej časti kapitoly popíšeme mechanizmus bodov rozšírenia pre platformu Eclipse. Je to potrebné z dôvodu, že chceme pridať položku do kontextového menu Eclipse a to sa rieši práve mechanizmom bodov rozšírenia. Ďalej bude nasledovať popis konkrétnych krokov, ako sme nadefinovali bod rozšírenia pre náš konkrétny prípad. Nakoniec sa budeme venovať popisu akcie, ktorá sa vykoná po vybratí príslušnej položky z kontextového menu.

Základným pravidlom pre výstavbu modulárnych softvérových systémov je vyhnúť sa pevným väzbám (tight coupling) medzi komponentmi. Eclipse umožňuje použitie voľných väzieb (loose coupling) prostredníctvom mechanizmu rozšírení (extensions) a bodov rozšírení (extension points). Body rozšírenia sú rôzneho typu a jedine rozšírenia, ktoré sú navrhnuté pre konkrétny bod rozšírenia, budú pasovať. Ak chce zásuvný modul umožniť iným zásuvným modulom rozšíriť alebo upraviť časti jeho funkcionality, tak deklaruje bod rozšírenia. Bod rozšírenia deklaruje kontrakt, ktorý musí rozšírenie splniť.

Takýmto spôsobom dosiahneme aj pridanie novej položky do kontextového menu. Do súboru plugin.xml vo vygenerovanom editore (t.j. projekt *.diagram) pridáme bod rozšírenia kontextového menu, t.j. `<extension point="org.eclipse.ui.popupMenus">`. Existujú dva prístupy pridania položky do kontextového menu: viewerContribution a objectContribution. ObjectContribution zabezpečí zobrazenie položky menu v kontextovom menu len pre konkrétny objekt. ViewerContribution na druhú stranu zabezpečí zobrazenie položky menu v kontextovom menu náhľadu alebo editora určeného definovaným id. Podrobnejšie o bode rozšírenia kontextového menu je možné sa dočítať v [32] a [33], v tejto časti uvedieme len podstatné časti pre náš editor.



Obrázok 34. Kontextové menu pre poskytované rozhranie

V našom prípade využijeme objectContribution, pretože potrebujeme zobraziť kontextové menu len pre požadované a poskytované rozhrania (ako je znázornené na obrázku 34), pričom pre každý typ rozhrania nadefinujeme jeden objectContribution. Atribút objectClass nadefinujeme ako ProvidedInterfaceEditPart (resp. RequiredInterfaceEditPart). Tento atribút reprezentuje objekt, pre

ktorý sa má daná položka kontextového menu zobraziť. Pridáme element menu, ktorý reprezentuje položku menu, ďalej element separator, ktorý definuje submenu daného menu. Nakoniec nadefinujeme akciu, ktorá sa má pri výbere položky menu vykonať, pričom element class ukazuje na danú akciu. Ukážka tohto bodu rozšírenia pre poskytované rozhranie je uvedená nižšie.

```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution
    id="ICP_IN"
    objectClass="diagram2.diagram.edit.parts.ProvidedInterfaceEditPart">
  <menu
    id="addMenu1"
    path="additions"
    label="Add">
  <separator name="group1"/>
  </menu>
  <action
    id="action1"
    label="ICP Inward"
    style="push"
    menubarPath="addMenu1/group1"
    class="dpdiagrameditor.diagram.Action1"/>
  </objectContribution>
</extension>
```

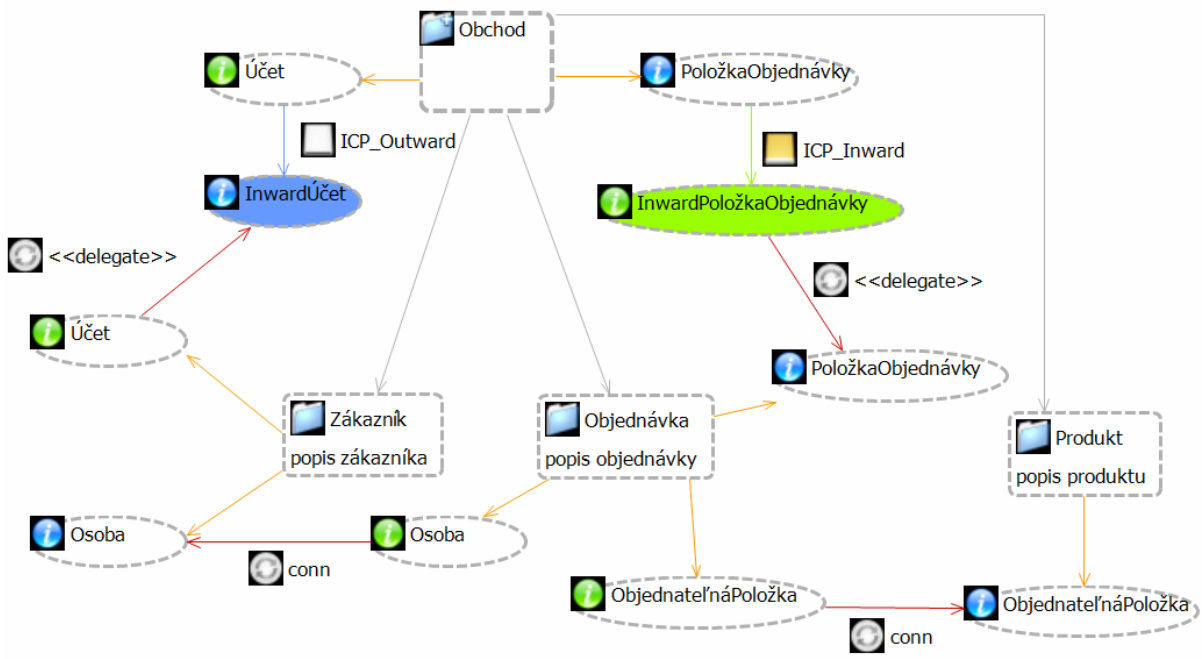
V nasledujúcej časti sa budeme venovať definovaniu akcie. Keďže je kontextové menu definované ako objectContribution, trieda reprezentujúca akciu (t.j. trieda Action1) musí implementovať rozhranie IObjectActionDelegate. V tejto triede je potrebné implementovať viaceré metódy. Metóda `void selectionChanged(IAction action, ISelection selection)` zabezpečí zobrazenie príslušnej položky menu v prípade, ak sa vyberie správny objekt (t.j. poskytované alebo požadované rozhranie). Metóda `void setActivePart(IAction action, IWorkbenchPart targetPart)`nastaví aktívnu časť editora, ktorá sa obyčajne používa na získanie pracovného kontextu pre akciu. V našom prípade nie je potrebné telo tejto metódy vypĺňať.

Poslednou a najdôležitejšou metódou je `void run(IAction action)`, ktorá vykonáva danú akciu. Princíp je nasledovný. Vytvoríme nové požadované rozhranie, nadefinujeme jeho pozíciu, vytvoríme prepojenie ICPInward medzi vybraným poskytovaným rozhraním a novo vytvoreným požadovaným rozhraním a kurzor nastavím na atribút Name novo vytvoreného požadovaného rozhrania, aby mohol používateľ zadať meno tohto rozhrania. V prvom rade vytvoríme zložený príkaz (CompoundCommand), do ktorého postupne budeme pridávať jednotlivé príkazy a nakoniec ho vykonáme. Problematike príkazov sa venujeme v kapitole 4.2.2.4. Potom vytvoríme požiadavku (Request), t.j. v našom prípade CreateViewRequest, ktorá reprezentuje vytvorenie nového požadovaného rozhrania. Požiadavka sa používa na komunikáciu s EditPart, ako znázorňuje obrázok 20 v kapitole 4.2.2. Požiadavka zapuzdruje informácie, ktoré potrebuje EditPart na vykonanie rozličných funkcií. Požiadavky sa používajú na získanie príkazov, zobrazovanie spätnej väzby a na vykonávanie všeobecných operácií. K tejto požiadavke priradíme pozíciu nového objektu, na ktorej sa má zobraziť. Z tejto požiadavky následne vytvoríme príkaz (Command) a pridáme ho do zloženého príkazu. V ďalšom kroku vytvoríme príkaz na vytvorenie väzby ICPInward a opäť ho

pridáme do zloženého príkazu. Tento zložený príkaz vykonáme. V poslednom kroku vyberieme novo vytvorené požadované rozhranie (RequiredInterfaceEditPart), vytvoríme požiadavku na editovanie tohto rozhrania (new `Request(RequestConstants.REQ_DIRECT_EDIT)`) a túto požiadavku vykonáme. Na úplné pochopenie zdrojového kódu odporúčame [20] a [26] a nahliadnuť do okomentovaného zdrojového kódu na priloženom CD.

8 Prípadová štúdia

Budeme postupovať na základe diagramu komponentov uvedeného v kapitole 2.2.3. Rozdiel medzi prístupom klasického UML diagramu komponentov a našim prístupom je popísaný v kapitole 8.1, vizuálny rozdiel môžeme zistiť porovnaním obrázkov 5 a 35. Serializovaná verzia tohto diagramu sa nachádza v prílohe 3.



Obrázok 35. Diagram prípadovej štúdie

Postupujeme nasledovne. Zostrojíme zložený komponent Obchod a poskytované rozhranie PoložkaObjednávky a požadované rozhranie Účet. Tieto rozhrania pripojíme k zloženému komponentu prostredníctvom väzby ComponentInterfaces. Prostredníctvom kontextového menu pridáme k poskytovanému rozhraniu PoložkaObjednávky vnútorné požadované rozhranie InternalPoložkaObjednávky, podobne pre požadované rozhranie Účet pridáme vnútorné poskytované rozhranie InternalÚčet. Vzniknutých zástupcov pomenujeme ICPI_PO (InternalComponentProxyInward_PoložkaObjednávky) a ICPO_U (InternalComponentProxy Outward_Účet). Vytvoríme tri primitívne komponenty Zákazník, Objednávka a Produkt a pripojíme ich k zloženému komponentu Obchod pomocou väzby CompositeComponentSubcomponents. Na úrovni subkomponentov vytvoríme požadované rozhranie Účet a poskytované rozhranie Osoba a pripojíme ich pomocou prepojenia ComponentInterfaces k primitívnemu komponentu Zákazník, ďalej vytvoríme požadované rozhranie Osoba a ObjednateľnáPoložka a poskytované rozhranie PoložkaObjednávky a pripojíme ich ku komponentu Objednávka a nakoniec vytvoríme poskytované rozhranie ObjednateľnáPoložka a pripojíme ho ku komponente Produkt. Teraz nasleduje prepájanie jednotlivých rozhraní pomocou väzby Connection. Prepojíme rozhrania InternalÚčet a Účet, rozhrania InternalPoložkaObjednávky a PoložkaObjednávky, poskytované rozhranie osoba komponentu Zákazník s požadovaným rozhraním Osoba komponentu Objednávka, a požadované

rozhranie ObjednateľnáPoložka komponentu Objednávka s poskytovaným rozhraním ObjednateľnáPoložka komponentu Produkt.

8.1 Nový prístup k diagramu komponentov

V tejto kapitole zanalyzujeme problematiku klasického diagramu komponentov a prístupu, ktorý sme zvolili pre náš editor. Zameriame sa na podstatné rozdiely týchto dvoch prístupov a zdôvodníme, prečo sme sa rozhodli postupovať daným spôsobom.

Náš navrhovaný editor sa líši od klasického prístupu UML komponentového diagramu. V klasickom komponentovom diagrame sú rozhrania znázornené prostredníctvom kruhu a polkruhu, ktoré sa na seba priamo napájajú, ako je popísané v kapitole 2.2. V našom prípade sú rozhrania reprezentované prostredníctvom uzlov a budú navzájom prepájané prostredníctvom väzieb. Ďalším rozdielom oproti klasickému komponentovému diagramu je znázornenie vnútornej štruktúry zloženého komponentu. Prístup klasického komponentového diagramu je popísaný v kapitole 2.2.2. Náš diagram komponenty znázorňuje do stromovej štruktúry, t.j. subkomponent nebude znázornený priamo vnútri rodičovského zloženého komponentu, ale bude znázornený ako potomok rodiča v stromovej štruktúre. Takto bude celý diagram komponentov reprezentovaný prostredníctvom stromovej štruktúry. Zásadná výhoda tohto prístupu je, že povoľuje zobrazenie akéhokoľvek počtu úrovní zanorenia pri zachovaní prehľadnosti a jednoduchosti práce. U klasického diagramu je zobrazenie akéhokoľvek počtu úrovní zanorenia síce rovnako možné, ale skôr len z teoretického hľadiska, pretože každé ďalšie zanorenie silne uberá na prehľadnosti diagramu a pridáva zložitosť pohybovania sa medzi rôznymi úrovňami komponentov. Poslednou podstatnou zmenou je spôsob zobrazenia portu, na klasický diagram sa opäť odkazujeme na kapitolu 2.2.2. V našom prípade je port reprezentovaný prostredníctvom zástupcu (proxy). Pri vytváraní zástupcu sa vytvorí špeciálne vnútorné rozhranie zloženého komponentu a pomocou zástupcu sa prepojí s príslušným vonkajším komponentom. Tejto problematike sa bližšie venuje kapitola 6.3. Týmto spôsobom sme dosiahli nový prístup k diagramu komponentov.

9 Zhodnotenie a ďalší smer vývoja systému

V tejto kapitole popíšeme, do akej miery sa nám podarilo splniť jednotlivé body zadania diplomovej práce a možnosti budúceho smerovania editora.

9.1 Zhodnotenie

V práci sa nám podarilo splniť zadanie vo všetkých bodoch. Problematike UML sa venuje kapitola 2, problematike komponentového softvéru kapitola 3. Softvérové rámce platformy Eclipse sú popísané v kapitole 4 a vývojový cyklus softvéru na platforme Eclipse je popísaný v kapitole 6.1. Aplikácia je navrhnutá v kapitole 6 na základe rozpracovaných požiadaviek vychádzajúcich zo zadanie, ktorá sú spracované v kapitole 5. Editor je implementovaný pomocou EMF a GMF a nachádza sa na priloženom CD. Prípadovej štúdii použitia aplikácie sa venuje príloha 3. Dosiahnuté výsledky sú diskutované v tejto kapitole a rovnako je navrhnutý ďalší smer vývoja systému.

Práca môže slúžiť aj ako ucelený úvod do problematiky komponentových systémov, ale najmä do problematiky frameworkov EMF a GMF. K EMF existuje kvalitná literatúra [17]. Ku GMF neexistuje žiadna kniha, všetky informácie sme čerpali z internetu. Na domovských stránkach projektu je síce spomínaná pripravovaná kniha, ktorá mala vyjsť v roku 2008, ale doteraz nevyšla. Dokumentácia ku GMF je nedostatočná a nesystematická. Ide hlavne o to, že dokumentácia sa nachádza na rôznych miestach, rôzne princípy sú vysvetľované v rôznych článkoch a takto vzniká neprehľadná zmes hoci i niektorých kvalitných článkov. Na stránke je viacero tutoriálov pre základnú prácu s frameworkom, ale tie nie sú podľa nášho názoru dostatočne vhodne napísané pre začiatočníkov a preto nespĺňajú svoj účel. V tutoriáloch chýbajú komentáre zdrojových kódov a tak používateľ musí hľadať ďalšie materiály, ktoré mu umožnia pochopiť dané zdrojové kódy. Absencia kvalitnej knihy, kde by bolo na jednom mieste popísané všetky podstatné rysy GMF, je veľmi citeľná.

Tým pádom zostáva potenciál tohto frameworku čiastočne nevyužitý a väčšinu používateľov rýchlo odradí, keď majú riešiť pokročilejšie funkcie. Pokročilejšie problémy sa väčšinou riešia len na fóre komunity, takže riešenie pokročilých funkcií spočíva v prehľadávaní tohto fóra, či už niekto podobný problém riešil alebo v pridaní vlastného problému s očakávaním, že nám členovia komunity budí schopní prípadne ochotní odpovedať. Táto skutočnosť veľmi spomaľuje vývoj editora. Z vlastných skúseností preto neodporúčame používať GMF na väčšie a zložitejšie projekty a to minimálne do doby, kým nevyjde kvalitná kniha, v ktorej budú popísané jednak princípy a na druhú stranu sa v nej zhrnú prínosy používateľov, ktorí sa o svoje skúsenosti podelili vo fóre komunity.

9.2 Ďalší smer vývoja systému

V kapitole 6.1.2. sme popísali grafickú definíciu editora. V tejto časti by bolo možné vylepšiť výzor výsledného editora, dajú sa tu definovať vlastné tvary uzlov a prepojení, priradiť k nim vlastné ikony a podobne. Cieľom práce však nebola prepracovaná vizualizácia výsledného editora, ale jeho formálna správnosť.

V kapitole 6.1.3. sme popísali definíciu nástrojov. Opäť v tejto časti by bolo možné editor rozšíriť, a to napríklad o vlastné kontextové menu, prípadne je možné nadefinovať aj vlastné ikony pre jednotlivé prvky v paneli nástrojov.

Ďalšiu možnosťou je pridanie niektorých automatizovaných akcií, aby používateľ mohol prostredníctvom jedného kliknutia spraviť niekoľko akcií naraz, prípadne priradenie klávesových skratiek k týmto akciám, čo umožňuje ešte väčšie zrýchlenie práce s editorom. Pokročilou funkciou by bolo pridanie validácie, či zostrojený editor je správny a spĺňa všetky definované obmedzenia. Editor môže byť vygenerovaný vo forme odľahčeného prostredia vykonávania, ako opisuje kapitole 6.2. Z palety a kontextového menu je možné odstrániť neželané položky. Veľký priestor na upravovanie editora poskytuje práve model generovania popísaný v kapitole 7.5. Funkčný základ pre ďalší vývoj aplikácie je hotový. Možností úprav je mnoho, ako sme spomínali vyššie, záleží len na používateľovi, akým spôsobom si editor upraví pre vlastné potreby.

10 Záver

Prvá časť diplomovej práce predstavila UML diagram komponentov a jeho možnosti modelovania, ktoré poskytuje. Druhá časť sa podrobne venovala komponentovému softvéru, ukázala nám viacero pohľadov na túto problematiku, sú v nej uvedené výhody tohto prístupu voči ostatným prístupom. Sú tu podrobnejšie popísané aj komponentové technológie, konkrétne CORBA Component Model od OMG, Enterprise JavaBeans od Sunu (stručne i ostatné štyri komponentové technológie od Sunu) a nakoniec Component Object Model od Microsoftu. Štvrtá kapitola sa venovala predstaveniu softvérových rámcov platformy Eclipse pre prácu s meta-modelmi. Podrobne tam sú popísané možnosti rozsiahleho EMF rámca, ktorý tvorí základnú časť navrhovanej aplikácie. Popis ostatných rámcov zahŕňa rámec GEF, ktorý slúži na vývoj grafických reprezentácií pre existujúce modely. Stručne je popísaný aj rámec GMF, ktorý tieto dva predchádzajúce rámce integruje a umožňuje nám vytvárať plnohodnotné editory s pokročilými funkciami na základe štruktúrovaného modelu. Piata kapitola špecifikovala požiadavky na editor. V šiestej kapitole je popísaný návrh grafického editoru, konkrétne životný cyklus vývoja tohto editora a jeho podstatná časť – meta-model. Nasleduje samotná implementácia editora. Kapitola sa venuje celému životnému cyklu vývoja editora až do jeho spustiteľnej podoby a nasledujú pokročilejšie úpravy editora – kontrola vytvárania väzieb a pridanie automatizovaných akcií ako položky do kontextového menu. Na záver v ôsmej kapitole popisujeme prípadovú štúdiu použitia editora a v deviatej kapitole diskutujeme možné rozšírenia editora a prípadné pokračovanie vo vývoji.

Práca vychádza z pokladov, ktorými sú knihy a zdroje na internete venujúce sa UML, komponentovému softvéru a Eclipse rámcom pre prácu s meta-modelmi.

Práca nielenže podrobne rozoberá vývoj založený na komponentoch, mnohé aspekty komponentového softvéru a popis možností softvérových rámcov platformy Eclipse pre prácu s meta-modelmi, ale rovnako sa venuje návrhu grafického editoru, jeho životného cyklu od počiatočného meta-modelu, ktorý je základom vyvíjanej aplikácie, až po finálny krok, ktorým je vygenerovanie funkčného editoru. Je v nej rovnako podrobne popísaná fáza implementácie výsledného editora, od základných častí až po pokročilú úpravu a diskutuje možné rozšírenia a ďalší smer vývoja editora. Preto je práca vhodným vstupom do sveta komponentového softvéru a do problematiky Eclipse frameworkov pre prácu s meta-modelmi, t.j. EMF a GMF. Je vhodná a dostatočne zrozumiteľná pre začiatočníkov, no venuje sa i pokročilejším aspektom vývoja editorov prostredníctvom GMF frameworku.

Literatúra

[1] Hamilton, K., Miles, R.: Learning UML 2.0. O'Reily 2006.

[2] Bell, D.: UML basics: The component diagram. 2004. [Online, navštívené 14.11.2008]. URL: http://www.ibm.com/developerworks/rational/library/dec04/bell/

[3] Ambler, S.: The Object Primer 3rd Edition: Agile Model Driven Development with UML 2.Cambridge University Press 2004.

[4] UML 2 Diagrams, Component Diagram. [Online, navštívené 16.11.2008]. URL: http://www.visual-paradigm.com/VPGallery/diagrams/Component.html

[5] Szyperski, K.: Component Software, Beyond Object-Oriented Programming. Second Edition. Addison-Wesley 2002.

[6] Crnkovic, I.: Component-based Software Engineering: Building Systems form Software Components. [Online, navštívené 16.11.2008]. URL: http://www.mrtc.mdh.se/publications/0405.pdf

[7] Barlett, D.: CORBA Component Model (CCM). 2001. [Online, navštívené 8.12.2008]. URL: http://www.ibm.com/developerworks/webservices/library/co-cjct6/

[8] Schmidt, D.C.: Tutorial on the Lightweight CCM. [Online, navštívené 8.12.2008]. URL: http://www.cs.wustl.edu/~schmidt/OMG-CCM-Tutorial.ppt

[9] Raj, G.S.: The CORBA Component Model (CCM). 1999. [Online, navštívené 6.12.2008]. URL: http://gsraj.tripod.com/corba/ccm.html

[10] Enterprise JavaBeans Overview. 2001. [Online, navštívené 8.12.2008]. URL: http://manuals.sybase.com/onlinebooks/group-as/asg1250e/ejbserv/@Generic__BookTextView/1985;pt=1979

[11] The Java EE 5Tutorial. 2008. [Online, navštívené 9.12.2008]. URL: http://java.sun.com/javaee/5/docs/tutorial/doc/index.html

[12] Nordby, K.: What are Enterprise JavaBeans components?: Part 1: The history and goals of EJB architecture. [Online, navštívené 12.12.2008]. URL: http://www.ibm.com/developerworks/java/library/j-what-are-ejbs/part1/

[13] Nordby, K.: What are Enterprise JavaBeans components?: Part 2: EJB programming model. [Online, navštívené 12.12.2008]. URL: http://www.ibm.com/developerworks/java/library/j-what-are-ejbs/part2/

[14] Nordby, K.: What are Enterprise JavaBeans components?: Part 3: Deploying and using Enterprise JavaBeans components. [Online, navštívené 12.12.2008]. URL: http://www.ibm.com/developerworks/java/library/j-what-are-ejbs/part3/

[15] The Component Object Model: Technical Overview. 1996. [Online, navštívené 10.12.2008]. URL: http://www.cs.umd.edu/~pugh/com/

[16] Eclipse Modeling Framework Project (EMF). Naposledy aktualizované 2009. [Online, navštívené 22.12.2008]. URL: http://www.eclipse.org/modeling/emf/?project=emf

[17] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T. J.: Eclipse Modeling Framework: A Developer's Guide. Addison-Wesley 2003.

[18] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1995.

[19] Merks, E.,Steinberg, D.: From Models to Code with the Eclipse Modeling Framework. 2005. [Online, navštívené 26.12.2008].
URL: http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial11 final.pdf

[20] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P.: Eclipse Developmentusing the Graphical Editing Framework and the Eclipse Modeling Framework. 2004. [Online, navštívené 11.5.2009].
URL: http://www.redbooks.ibm.com/abstracts/sg246302.html

[21] Hudson, R.: Create an Eclipse-based application using the Graphical Editing Framework. Naposledy aktualizované 2006. [Online, navštívené 1.1.2009].
URL: http://www.ibm.com/developerworks/opensource/library/os-gef/

[22] The Graphical Editing Framework. 2004. [Online, navštívené 2.1.2009].
URL: http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/ 47_Hudson.pdf

[23] Shatalin, A., Tikhomirov, A.: Introduction to the Graphical Modeling Framework. 2008. [Online, navštívené 3.1.2009].
URL: http://www.eclipsecon.org/2008/index.php?page=sub/&id=337

[24] GMF Tutorial. Naposledy aktualizované 2008. [Online, navštívené 3.1.2009].
URL: http://wiki.eclipse.org/index.php/GMF_Tutorial

[25] GMF Tutorial Part 4. Naposledy aktualizované 2007. [Online, navštívené 3.1.2009].
URL: http://wiki.eclipse.org/GMF_Tutorial_Part_4

[26] GMF Tutorial Part 3. Naposledy aktualizované 2007. [Online, navštívené 13.5.2009]
URL: http://wiki.eclipse.org/GMF_Tutorial_Part_3

[27] Learn Eclipse GMF in 15 minutes. 2006. [Online, navštívené 4.1.2009].
URL: http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/

[28] Rychlý, M.: A Component Model with Support of Mobile Architectures and Formal Description. e-Informatica Software Engineering Journal, vol. 2, no. 2, 2009, ISSN 1897-7979, (in print)

[29] Eclipse NewsPortal - eclipse.modeling.gmf. Naposledy aktualizované 2009. [Online, navštívené 16.5.2009].
URL: http://www.eclipse.org/newsportal/thread.php?group=eclipse.modeling.gmf

[30] GMF Resources – tutorials, troubleshooting, references. Naposledy aktualizované 2009. [Online, navštívené 17.5.2009].
URL: http://www.jevon.org/wiki/GMF

[31] Developer Guide to the GMF Runtime Framework. 2006. [Online, navštívené 1.5.2009].
URL: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.gmf.doc/prog-guide/runtime/index.html

[32] Vogel, L.: Eclipse Extension Points and Extensions. 2009. [Online, navštívené 11.5.2009].
URL: http://www.vogella.de/articles/EclipseExtensionPoint/article.html

[33] Eclipse Popup Menus. [Online, navštívené 11.5.2009].
URL: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/ workbench_basicext_popupMenus.htm

Zoznam príloh

1. Návod na inštaláciu
2. Používateľská príručka
3. Serializovaný diagram
4. Obsah CD
5. CD

Príloha 1. Návod na inštaláciu

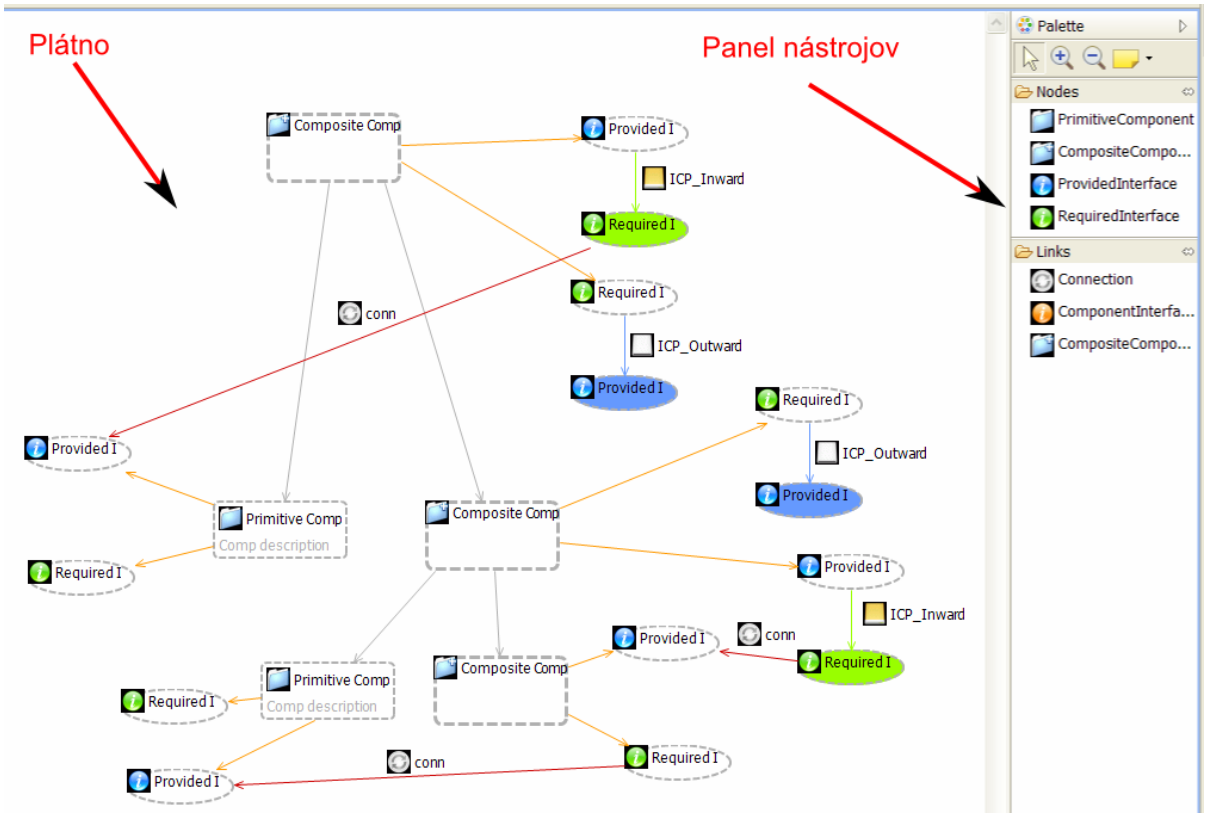
Pre spustenie editora je nutné nainštalovať nasledovný softvér v tomto poradí.

1. Java Runtime Environment 6 Update 13
   URL: http://www.java.com/en/download/index.jsp
2. Eclipse Classic 3.4.2
   URL: http://www.eclipse.org/downloads/
3. EMF 2.4.2 (2009/02/17)
   URL: http://www.eclipse.org/modeling/emf/downloads/?project=
4. GMF 2.1.3 (2009/02/18)
   URL: http://www.eclipse.org/modeling/gmf/downloads/?project=

EMF a GMF inštalujeme prostredníctvom softvérových aktualizácií Eclipse. V menu vyberieme Help > Software Updates > Available Software > Ganymede Update Site > Models and Model Development. Vyberieme Ecore Tools, Ecore Tools SDK, EMF – Eclipse Modeling Framework Runtime and Tools, EMF SDK - – Eclipse Modeling Framework SDK, Graphical Modeling Framework Runtime a Graphical Modeling Framework SDK. Nainštalujeme ich a reštartujeme Eclipse. Vytvoríme tri nové projekty prostredníctvom File > New > Java Project a nazveme ich DPDiagramEditor, DPDiagramEditor.diagram a DPDiagramEditor.edit. Do nich skopírujeme príslušné adresáre s rovnakým názvom, ktoré sa nachádzajú na priloženom CD. Prepíšeme všetky súbory súbormi z CD. Následne musíme aktualizovať projekty v Eclipse, označíme si projekty a klikneme na ne pravým tlačítkom a vyberieme Refresh. Označíme projekt DPDiagramEditor.diagram a spustíme ho ako Eclipse Application, t.j. v menu Run > Run As > Eclipse Application. Spustí sa náš editor.

Príloha 2. Používateľská príručka

Nový diagram vytvoríme nasledovne. Označíme projekt DPDiagramEditor.diagram a spustíme ho ako Eclipse Application, t.j. v menu Run > Run As > Eclipse Application. Spustí sa náš editor ako samostatná Eclipse aplikácia. V menu vyberieme New > Diagram2 Diagram. Editor sa skladá z dvoch základných častí – plátna a panela nástrojov, ako je znázornené na obrázku 36. Na plátno sa vykresľuje vytváraný diagram a panel nástrojov zobrazuje množinu uzlov a prepojení medzi nimi. Panel nástrojov obsahuje dve skupiny nástrojov, uzly (CompositeComponent, PrimitiveComponent, ProvidedInterdace, RequiredInterface) a prepojenia (ICPInward, ICPOutward, Connection, ComponentInterfaces, CompositeComponentSubcomponents).



Obrázok 36. Ukážka diagramu v editore

V diagrame môžu byť znázornené komponenty, ktoré môžu byť navzájom poprepájané. Komponenty môžu byť jednoduché (PrimitiveComponent) alebo zložené (CompositeComponent). Zložené komponenty obsahujú v sebe ďalšie komponenty, ktoré opäť môžu byť zložené alebo jednoduché. Subkomponenty zloženého komponentu k nemu pripájame prostredníctvom väzby subcomponents (CompositeComponentsSubcomponents). Každý komponent je charakterizovaný svojimi rozhraniami, ktoré sú buď poskytované (ProvidedInterface) alebo požadované (RequiredInterface). Rozhrania ku komponentom pripájame prostredníctvom väzby interfaces (ComponentInterfaces). Komponenty môžu byť navzájom poprepájané prostredníctvom väzieb (Connection), t.j. požadované rozhranie jedného komponentu bude napojené na poskytované rozhranie druhého komponentu. Dôležitou funkciou je takisto prepojenie rozhraní vonkajšieho komponentu s rozhraniami svojich vnútorných subkomponentov. Ako je popísané v kapitole 5,

realizujeme to prostredníctvom zástupcov (ICPInward a ICPOutward). Týchto zástupcov vyvoláme z kontextového menu (Add > ICPInward, resp. Add > ICPOutward), ako popisuje kapitola 7.6.2.

Pri vytváraní uzlov a prepojení ich môžeme pomenovať, prípadne neskôr tieto názvy editovať prostredníctvom vlastností (panel Properties). U primitívneho komponentu môžeme pridať aj popis. Editor podporuje operácie undo (klávesová skratka CTRL+Z) a redo (klávesová skratka CTRL+Y). Ďalej je možná úprava vzhľadu jednotlivých uzlov a prepojení prostredníctvom kontextového menu (Format > Font, Fill Color, Line Color, ….). Editor podporuje pridávanie poznámok (kontextové menu > Add Note), export diagramu do obrázku (kontextové menu > File > Save As Image File, podporované formáty sú GIF, BMP, JPEG, JPG, SVG, PNG, PDF) a funkciu Zoom (kontextové menu > Zoom)

Príloha 3. Serializovaný diagram

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:diagram2="http://diagram/1.0">
 <diagram2:CompositeComponent/>
 <diagram2:CompositeComponent Name="Obchod">
  <interfaces xsi:type="diagram2:ProvidedInterface" Name="PolozkaObjednavky"/>
  <interfaces xsi:type="diagram2:RequiredInterface" Name="Ucet"/>
  <subcomponents xsi:type="diagram2:PrimitiveComponent" Name="Zakaznik">
   <interfaces xsi:type="diagram2:RequiredInterface" Name="Ucet"/>
   <interfaces xsi:type="diagram2:ProvidedInterface" Name="Osoba"/>
  </subcomponents>
  <subcomponents xsi:type="diagram2:PrimitiveComponent" Name="Objednavka">
   <interfaces xsi:type="diagram2:RequiredInterface" Name="Osoba"/>
   <interfaces xsi:type="diagram2:RequiredInterface" Name="ObjednatelnaPolozka"/>
   <interfaces xsi:type="diagram2:ProvidedInterface" Name="PolozkaObjednavky"/>
  </subcomponents>
  <subcomponents xsi:type="diagram2:PrimitiveComponent" Name="Produkt" Description="">
   <interfaces xsi:type="diagram2:ProvidedInterface" Name="ObjednatelnaPolozka"/>
  </subcomponents>
  <connections output="/1/@subcomponents.2/@interfaces.0"
input="/1/@subcomponents.1/@interfaces.1"/>
  <connections output="/1/@subcomponents.1/@interfaces.2" input="/1/@proxies.0/@inner"/>
  <connections output="/1/@subcomponents.0/@interfaces.1"
input="/1/@subcomponents.1/@interfaces.0"/>
  <connections output="/1/@proxies.1/@inner" input="/1/@subcomponents.0/@interfaces.0"/>
  <proxies xsi:type="diagram2:ICPInward" Name="ICPI_PO" outer="/1/@interfaces.0">
   <inner Name="InternalPolozkaObjednavky"/>
  </proxies>
  <proxies xsi:type="diagram2:ICPOutward" Name="ICPO_U" outer="/1/@interfaces.1">
   <inner Name="InternalUcet"/>
  </proxies>
 </diagram2:CompositeComponent>
</xmi:XMI>
```

Príloha 4. Obsah CD

Adresárová štruktúra CD je nasledovná:

- doc/ – technická správa diplomovej práce
- project/diagram/ – serializovaný diagram z prípadovej štúdie
- project/model/ – súbory GMF modelu editora
- project/workspace/ – zdrojové kódy aplikácie
- project/plugin.xml – súbor zásuvných modulov rozšírený o vlastné položky kontextového menu
- readme.txt - návod na inštaláciu