VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

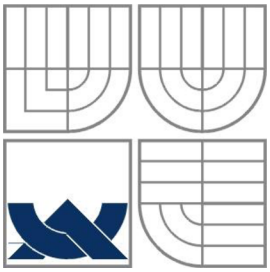
MODELOVACÍ NÁSTROJ PRO GRAFICKÝ NÁVRH KOMPONENTOVÝCH SYSTÉMŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

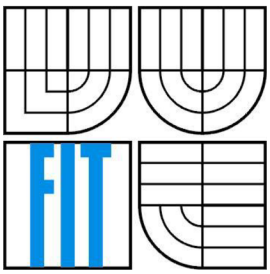
AUTOR PRÁCE
AUTHOR

Bc. ZOLTÁN ZEMKO

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER 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. ZOLTÁN ZEMKO

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. MAREK RYCHLÝ, Ph.D.

BRNO 2013

Abstrakt

Softwarové inženýrství založené na komponentách (Component-based Software Engineering) popisuje rozsáhlý informační systém jako množinu komponent. Práce se snaží poukázat na výhody tohoto přístupu. Definuje taktéž pojmy jako komponentový a standardní software a uvádí základy modelovacích technik komponentových systémů v jazyce UML. V dokumentu je popsána struktura Eclipse Modeling Project. Čtenář by z tohoto textu měl získat teoretický přehled postupů vývoje modelovacích nástrojů nad platformou Eclipse. Práce taktéž obsahuje návrh a postup implementace modelovacího nástroje podporujícího návrh komponentových systémů, který byl vyvinut použitím Eclipse Modeling Framework a Graphical Modeling Framework.

Abstract

Component-based Software Engineering describes a complex information system as a set of components. The thesis seeks to highlight the benefits of this approach. Also defines terms such as standard software, component software, and others. It provides an introduction to the modeling techniques of component-based systems in UML. The second half of the document describes the structure of the Eclipse Modeling Project. The reader by these lines should obtain a theoretical overview of the development of modeling tools under Eclipse. The document includes design and implementation process description of the tool for modeling component-based systems which has been developed using the Eclipse Modeling Framework and Graphical Modeling Framework.

Klíčová slova

EMP, EMF, GMF, GEF, komponenta, komponentový software, diagram komponent, UML, Eclipse

Keywords

EMP, EMF, GMF, GEF, component, component software, component diagram, UML, Eclipse

Citace

Zemko Zoltán: Modelovací nástroj pro grafický návrh komponentových systémů, diplomová práce, Brno, FIT VUT v Brně, 2013

Modelovací nástroj pro grafický návrh komponentových systémů

Prohlášení

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

.....
Bc. Zoltán Zemko
22. června 2013

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce panu RNDr. Markovi Rychlému, Ph.D., za odborné vedení, poskytnuté rady a připomínky a čas věnovaný konzultacím.

Obsah

1 Úvod 3

2 Vývoj založený na komponentoch 5

2.1 Komponenty a komponentový softvér 5

2.1.1 Znovu použiteľnosť komponent 5

2.1.2 Komponenty a softvérové abstrakcie 5

2.1.3 Komponent z pohľadu nasadenia 6

2.2 Softvér vlastnej výroby vs. štandardný softvér 6

3 Komponent diagramy v UML 9

3.1 Rozhranie 9

3.1 Komponent 9

3.2 Stereotypy komponentov 10

3.2.1 Pod systémy 11

3.3 Porty komponentov 11

3.4 Väzby medzi komponentmi 12

3.5 Vnútorná štruktúra komponentov 12

4 Eclipse Modeling Project 14

4.1 Doménovo-špecifický jazyk 14

4.2 Štruktúra Eclipse Modeling Project 15

4.2.1 Abstraktný syntaktický vývoj 16

4.2.2 Konkrétny syntaktický vývoj 18

4.2.3 Transformácia modelu 18

4.3 Eclipse Modeling Framework 19

4.3.1 EMF ECore model 20

4.3.2 Transformácia modelu ECore na Java kód 22

4.4 Graphical Modeling Framework 23

4.4.1 Doménového model 24

4.4.2 Model EMF generátora 24

4.4.3 Model grafických definícií 25

4.4.4 Model nástrojov 26

4.4.5 Model mapovania 26

4.4.6 Model GMF generátora 27

5 Analýza požiadaviek a návrh aplikácie 28

5.1 Špecifikácia požiadaviek 28

5.2 Návrh abstraktnej syntaxe modelovacieho nástroja 29

6 Vývoj abstraktnej syntaxe 34
6.1 ECore Model 34
6.2 Transformácia ECore modelu na zdrojový kód 43
6.3 Validácia abstraktnej syntaxe 43
6.3.1 Validačný adaptér 44
6.3.2 Validátori 44
6.3.3 Registrácia ValidationAdapter do katalógu validátorov 51
7 Vývoj konkrétnej syntaxe 52
7.1 Model grafických definícií 52
7.1.1 Galéria figúr 52
7.1.2 Uzly, spojenia, kabíny, textové štítky 54
7.2 Model nástrojov 55
7.3 Model mapovania 55
7.4 Model GMF generátora 57
7.5 Validácia na úrovni konkrétnej syntaxe 57
8 Prípad použitia 60
9 Zhodnotenie a ďalší smer vývoja 63
9.1 Zhodnotenie 63
9.2 Ďalší smer vývoja systému 64
10 Záver 65
11 Použitá literatúra 66
Príloha A – Obsah CD 68
Príloha B – Návod na inštaláciu 69
Príloha C – Import zdrojového kódu modelovacieho nástroja 72
Príloha D – Používateľská príručka 74

1 Úvod

Vývoj softvérových aplikácií sa čoraz viac prikláňa na používanie komponentov. Softvérový inžinieri komponenty vnímajú ako samostatnú časť systému poskytujúceho určité služby, ktoré sa dajú ľahšie udržovať a jednoducho vymeniť za novšie, alebo iné komponenty, ktoré pracujú na základe rovnakých rozhraní. [10]

Zrodilo sa nové odvetvie softvérového inžinierstva známe pod názvom softvérové inžinierstvo založené na komponentoch (Component-based Software Engineering), ktoré kladie dôraz na delenie funkcionality rozsiahleho softvérového systému do menších celkov. Táto disciplína umožní definovať, implementovať a nasadiť voľne viazané nezávislé komponenty do systému. [10]

V 2. kapitole diplomovej práce budeme skúmať význam komponent. Definujeme si pojmy ako komponentový softvér, štandardný softvér, softvér vlastnej výroby a určíme ich výhody, nevýhody.

Unified Modeling Language (UML) je grafickým jazykom, ktorý poskytuje veľkú množinu modelovacích techník. Umožní modelovať aj systémy založené na komponentoch. Na tento účel využíva komponent diagramy, ktoré patria medzi najpopulárnejšie prostriedky na vytváranie grafického návrhu komponentových systémov. V 3. kapitole tohto dokumentu sa oboznámime základnými modelovacími technikami využitím komponent diagramu.

Hlavným cieľom diplomového projektu bolo vytvoriť modelovací nástroj, ktorý umožní vytvoriť grafický návrh komponentového softvéru. Tento modelovací nástroj by mal byť nasadený ako plug-in nad platformou Eclipse. Z tohto dôvodu sa v 4. kapitole oboznámime štruktúrou Eclipse Modeling Project, ktorý obsahuje množinu frameworkov, ktoré nám umožnia vytvoriť grafický alebo textový modelovací nástroj pre platformu Eclipse. Taktiež sa definuje význam pojmu doménovo-špecifický jazyk a popíšeme jej prínos.

Po nazbieraní dostatok teoretických znalostí, v 5. kapitole sa oboznámime požiadavkami na riešenie modelovacieho nástroja, ktorý budeme vyvíjať ako doménovo-špecifický jazyk. Podľa získaných požiadaviek navrhneme model tried, ktorá bude reprezentovať štruktúru modelu vytvoreného modelovacím nástrojom (resp. abstraktnú syntax doménovo-špecifického jazyka).

Na základe vytvoreného návrhu, v 6. kapitole implementujeme meta-model popisujúci štruktúru modelu vytváraný. Jej sémantiku rozšírime o validačný mechanizmus, ktorý zabezpečí konzistentnosť údajov v modely modelovacieho nástroja.

V 7. kapitole sa zameriame na vývoj grafickej notácie prvkov modelovacieho nástroja (resp. abstraktnej syntaxe doménovo-špecifického jazyka). Popíšeme postup vytvorenia grafického editoru použitím Graphical Modeling Framework.

V 8. kapitole si uvedieme prípad použitia modelovacieho nástroja na konkrétnom príklade. Taktiež si povieme v akých aspektoch sa modelovací nástoj líši od komponent diagramu z UML a povieme si výhody jej použitia

V závere dokumentu zhodnotíme výsledky projektu. Zmienime sa o cieľoch, ktoré sa nám podarilo splniť. Popíšeme prínos modelovacieho nástroja a stanovíme ďalší smer vývoja.

2 Vývoj založený na komponentoch

V tejto kapitole budeme vychádzať z literatúry [1]. Definujeme základné pojmy ako je komponent, komponentový softvér. Zároveň budeme porovnávať štandardný softvér a softvér vlastnej výroby. Určíme ich výhody a nevýhody. Odôvodníme, prečo komponentový softvér získava čím viac, tým väčšiu popularitu.

2.1 Komponenty a komponentový softvér

Systém tvorený z softvérových komponentov sa nazýva komponentový softvér. Softvérové komponenty sú samostatne výkonné jednotky, nezávisle od produkcie, zisku a nasadenia, ktoré môžu tvoriť kompozíciu funkčného systému. Komponenty umožnia vytvorené „veci“ znovu použiť s menšími úpravami v existujúcich alebo v nových kompozíciach. [1]

2.1.1 Znovu použiteľnosť komponent

Znovu použiteľnosť predstavuje veľmi široký pojem, ktorý zahŕňa všeobecný koncept prínosu znovu použiteľnosti. Ako príklad si uveďme ľubovoľné popisy zachytávajúce výsledky návrhu. Tieto popisy, ako samotné obyčajne závisia na iných podrobnejších a viac špecializovaných popisoch. Aby sme dosiahli prínos znovu použiteľnosti, nestačí začať s jednoliatym návrhom celkového riešenia a potom ho postupne rozdeľovať na fragmenty.

Výhody takéhoto prístupu budú veľkou pravdepodobnosťou minimálne. Namiesto toho by tieto popisy by mali byť pozorne zovšeobecnené pre zabezpečenia znovu použiteľnosti v dostatočnom počte kontextov. Musíme však dávať pozor, aby sme sa vyhli príliš zovšeobecneným popisom a zachovali ich dostatočne odľahčené a voľné, aby aktuálna znovu použiteľnosť ostala praktická. Takéto popisy potom môžeme nazývať komponentmi (Sametinger, 1997). [1]

2.1.2 Komponenty a softvérové abstrakcie

Požiadavky na nezávislosť a samostatnú výkonnosť vylučuje veľa softvérových abstrakcií ako sú typové deklarácie (napr. makrá jazyka C, šablóny jazyka C++ alebo SmallTalk bloky). Iné abstrakcie ako procedúry, triedy, moduly alebo niekedy aj celé aplikácie môžu formulovať komponenty, kým sú v spustiteľnom stave a ostanú znovu použiteľné. Najstarším príkladom na softvérový komponent je knižnica s procedúrami. Požiadavka potenciálnej nezávislosti a spustiteľného stave je základná v poradí umožniť viacerých nezávislých dodávateľov, nezávislé nasadenia a robustnú integráciu. [1]

2.1.3 Komponent z pohľadu nasadenia

Z pohľadu nasadenia softvérová komponent je niečo, čo je aktuálne nasadené na komponentovom založenom prístupu. Môžeme povedať, že softvérové komponenty sú oddelenou, izolovanou časťou systému.

Napriek častým tvrdeniam, objekty sa nikdy nepredávajú, nekupujú alebo nenasadzujú sa. Jednotka nasadenia je niečo skôr statické ako trieda, množina tried alebo framework tried skompilovaný a zlinkovaný do nejakého balíčku. Objekty, ktoré logicky tvoria časť inštancií komponentov, podľa potreby sa zakladajú na triedach, ktoré boli nasadené s komponentom.

Hoci komponentom môže byť aj jediná trieda, radšej komponenty tvoria kolekcie tried, ktoré niekedy nazývame ako moduly. Z komponent preto často nie sú vytvorené inštancie. Taktiež komponent môže sa zakladať na celkom odlišných implementačných technológiách ako čisto funkcionálny alebo jazyk symbolických adries a nemusí vyzerať z vnútra ako objektovo orientovaný. [1]

2.2 Softvér vlastnej výroby vs. štandardný softvér

Softvérový vývoj môžeme deliť do dvoch táborov. Na jednej strane softvérov tvorí úplne od nuly čisto len použitím programovacích nástrojov a knižníc. Z iného pohľadu hovoríme o tzv. štandardnom softvéry, ktorý je kúpený a parametrizovaný tak, aby poskytoval riešenie, ktorý je dostatočný na vyhovenie požiadavkám (tzv. Outsourced).

Celý zákazkový softvér vlastnej výroby má síce výhody (samozrejme pri správnej funkčnosti), že môže byť optimálny požiadavkám zákazníka. Samostatne vyrobený softvér môže mať konkurenčnú výhodu trhu, ale za podmienok úspešnosti projektu, čo je nesmierne ťažké dosiahnuť.

Takýto zákazkový softvér má tiež vážne nevýhody aj napriek tomu, že pracuje správne. Výroba produktu od nuly je veľmi drahá. Údržba a prevádzka takéhoto systému môže byť taktiež veľkou záťažou. Ďalším problémom je, že softvér, vo väčšine prípadov, čiastočne alebo celkom nedokáže spolupracovať vnútro podnikovými aplikáciami zákazníka, či programami obchodných partnerov zákazníka. Tiež vo svete sa neustále menia obchodné požiadavky, preto tento typ zákazkového softvéru niekedy je už neskoro nasadený, aby bol produktívny. [1]

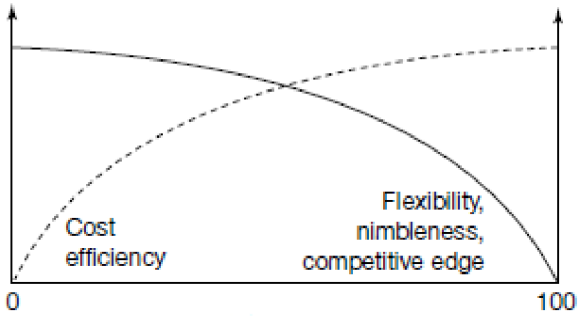
Vďaka nevýhodám zákazkového softvéru tvoreného vlastnou výrobou, pochopiteľne trendom v priemysle sa stáva tzv. Outsourced softvér. Výroba takéhoto zákazkového softvéru umožní externe obmedziť finančné riziká na základe zmlúv. Na pokrytie „Time-to-market“ rizika, svet rýchlo sa obracia k používaniu štandardizovaného softvéru. Znamená to, že softvér je ľahko prispôsobiteľný k aktuálnym potrebám. Záťaž údržby, interoperability je ponechaná na predajcu štandardného balíka.

Zostáva len vykonať parametrizácie a konfigurácie podrobností pri vytváraní novej verzie produktu. Síce to vyžaduje ešte stále veľké úsilie, avšak je nevyhnutné v dnešnej dobe zmien.

Aj štandardný softvér má svoje nevýhody. Štandardný softvér môže vyžadovať menšie či väčšie reorganizácie ovplyvnené meniacimi sa podnikovými procesmi. Aj keď prepracovanie obchodných procesov môže byť výhodné, mali by byť upravené predovšetkým pre vlastné potreby. Využitie optimálnej montáže štandardného softvéru by malo byť druhoradé. Ďalej štandardný softvéru je štandardný. Tretia nevýhoda je, že štandardný softvér nie je pod miestnou kontrolou, čiže nie je dosť „čiperný“ na rýchle prispôsobenie sa meniacim požiadavkám. [1]

Štandardné balíky vytvárajú úroveň podmienok a potrebné konkurenčné body prichádzajú z iných oblastí. Stále častejšie sa dajú vidieť softvérové služby, ako niečo, čo je treba „prežiť“. Je zrejmé, že je to ďaleko od ideálneho prípadu, keď informácie a spracovanie informácií majú veľký efekt vo väčšine obchodoch a niekedy vytvárajú ďalšie nové.

Koncept komponentového softvéru predstavuje strednú cestu, ktorá by tento problém mohla vyriešiť. Aj keď každý zakúpený komponent je jeden štandardizovaný produkt, proces nasadenia komponenty umožní významné úpravy. Je pravdepodobné, že tieto komponenty budú k dispozícií na základe kvality (úroveň výkonu, robustnosť, efektívnosť zdrojov, stupeň certifikácie, atď.) v rôznych cenách. Táto skutočnosť nám umožní si zvoliť jednotlivé priority na základe pevného rozpočtu. Navyše niektoré komponenty môžu byť vlastnoručne vyrobené tak, aby vyhovovali špecifickým požiadavkám, alebo podporovali strategické výhody. Obrázok Obr. 1 znázorňuje kompromisy, ktoré prinášajú spektrum možností, ktoré umožní komponentový softvér. [1]



Obr. 1: Spektrum medzi vlastnou výrobou a kúpenými komponentmi [1]

Graf však nie je v žiadnom prípade kvantitatívny a skutočný tvar dvoch kriviek je ľubovoľný. Avšak je jasné, že nelineárne efekty budú viditeľné, keď dosiahneme extrémy. Napríklad pri ľavom konci, keď všetko je vlastnoručne vyrobené, flexibilita je vysoká, ale efektivita nákladov bude približne nulová.

Komponentový softvér tiež rieši dlhodobý problém aktualizačných cyklov. Tradične integrované riešenia vyžadujú pravidelnú aktualizáciu. Obyčajne to bol bolestivý proces prevodu starých databáz, zabezpečenia kompatibility, preučenia zamestnancov, nákupu výkonnejšieho hardvéru, atď. V riešeniach, založených na komponentoch, sa umožní náhradu, či individuálna

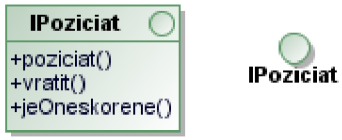
modernizácia prvkov podľa potreby. Okrem toho umožní plynulejšiu prevádzku. Samozrejme to vyžaduje rôzne spravovania jednotlivých služieb, ale získané výhody sú obrovské. [1]

3 Komponent diagramy v UML

V uvedenej kapitole sa budeme zaoberať modelovaním komponentov v UML 2. Definujeme si pojmy ako sú rozhranie, komponent, pod systém, port. V závere kapitoly preštudujeme spôsob vytvorenia väzieb medzi komponentmi a budeme modelovať vnútornú štruktúru komponenty. Obsah kapitoly bol vypracovaný na základe literatúry [2].

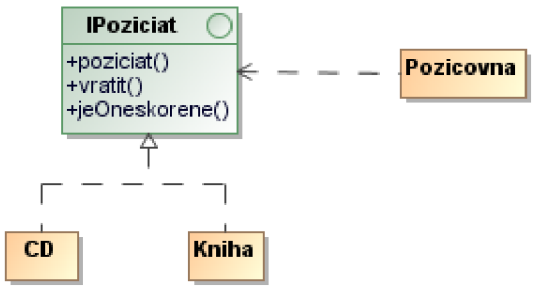
3.1 Rozhranie

Rozhranie tvorí predpis množiny verejných funkcií, ktorej úlohou je oddeliť špecifikáciu funkčnosti od fyzickej implementácie od triedy alebo pod systému. Taktiež môžeme povedať, že rozhranie tvorí kontrakt medzi odberateľom a dodávateľom služieb.



Obr. 2: Zobrazenie rozhrania v UML

Rozhrania nemôžu tvoriť inštancie. Deklarujú iba nejakú sémantiku, ktorá môže byť realizovaná nejakými klasifikátormi (triedy, pod systémy). Na nižšie uvedenom obrázku klasifikátori `CD` a `Kniha` realizujú (resp. implementujú) rozhranie `IPoziciat`, kým klasifikátor požičovňa využíva funkcie klasifikátorov `CD` a `Kniha` cez rozhranie `IPoziciat` bez toho aby vedel o implementačných podrobnostiach týchto klasifikátorov.



Obr. 3: Realizácia a používanie rozhrania klasifikátormi

3.1 Komponent

Na základe špecifikácie UML môžeme povedať, že komponent reprezentuje časť systému, ktorá ukrýva vlastný obsah pred okolitým svetom. Komponent by sme mohli prirovnať k čiernej skrinke, ktorej vnútornému chovanie je definované a sprístupnené pomocou rozhraní. Vďaka tejto skutočnosti je možné komponentu nahradiť druhou (v prípade, ak obidve sa riadia rovnakým protokolom).

Komponenty môžu zastupovať všetko, z čoho je možné vytvoriť objekt počas behu aplikácie (napr. komponenty EJB z JavaBeans), alebo môžu reprezentovať obyčajnú logickú konštrukciu (ako príklad môže byť pod systém).

Komponenty umožnia nám vytvoriť organizovanú štruktúru systému rozdelenú do zameniteľných častí a môžu sa prejavovať prostredníctvom artefaktov. Artefakt je špecifikáciou niečoho konkrétneho, ako je napr. zdrojový súbor. Príkladom môžeme uviesť opäť komponentu EJB, ktorá sa môže prejavovať cez nejaký JAR (Java ARchive) súbor.

Diagram komponentov vytvára pohľad na implementáciu, že ako sú organizované jednotlivé časti systému do modulov a taktiež umožní spracovať architektúru do vrstiev. Modeluje komponenty, závislosti medzi nimi, a spôsob akým sú klasifikátory priradzované komponentom.

Obr. 4: Notácia komponenty v UML

Komponent sa kreslí obyčajným obdĺžnikom, ktorý označíme stereotypom **<<component>>** a symbolom komponenty, ktorý sa umiestni do pravého horného rohu obdĺžnika. Každý komponent môže obsahovať požadované a sprístupnené rozhrania.

Obr. 5: Komponent s požadovaným a sprístupneným rozhraním

3.2 Stereotypy komponentov

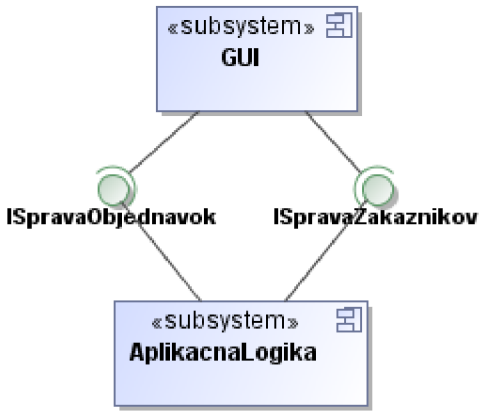
Komponenty patria medzi jedny najčastejšie stereotypné prvky v jazyku UML, nakoľko sú vhodné na definovanie ďalších podrobností o systéme. UML 2 poskytuje množinu štandardných stereotypov pre označovanie komponent.

Stereotyp	Význam
<<buildComponent>>	Definuje organizačné a vývojové účely na úrovni systému.
<<entity>>	Komponent obsahujúci perzistentnú informáciu
<<implementation>>	Definuje komponenty, ktoré ako samé neobsahujú špecifikáciu. Zastupuje implementáciu samotného prvku označeného stereotypom <<specification>>, na ktorom je závislá.
<<specification>>	Komponent špecifikujúci doménu objektov, bez toho aby definoval ich fyzickú implementáciu (napr. komponent, označená týmto stereotypom, musí mať len sprístupnené a požadované rozhrania a nesmie obsahovať realizujúce klasifikátory.
<<process>>	Transakčný komponent
<<service>>	Funkčná komponent bez súkromného stavu na výpočet hodnoty.
<<subsystem>>	Jednotka hierarchickej dekompozície rozsiahleho systému.

Tab. 1: Stereotypy komponent [2]

3.2.1 Pod systémy

Pod systém sa nazýva komponent (označovaný stereotypom **<<subsystem>>**), ktorý sa chová ako jednotka dekompozície väčšieho systému. Tvorí logickú konštrukciu k rozkladu väčšieho systému na menšie bloky. V metodike Unified Process podsystémy majú obrovskú hodnotu. Dokážu rozdeliť systém do menších, zrozumiteľnejších blokov, čo je kľúčom k úspešnému vývoju.



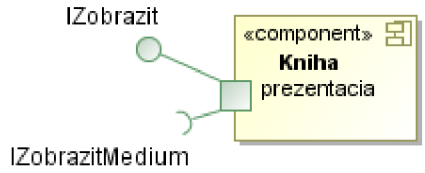
Obr. 6: Príklad IS rozdeleného na podsystémy

Ako aj z hore uvedeného obrázku vyplýva, jednotlivé podsystémy sú prepojené medzi sebou rozhraniami. Pod systém `AplikacnaLogika` realizuje rozhrania `ISpravaObjednavok` a `ISpravaZakaznikov`. Pod systém `GUI` v tomto prípade využíva funkcie pod systému `AplikacnaLogika` cez rozhrania, čiže nevie nič o implementačných detailoch pod systému `AplikacnaLogika`, čo znamená, že pod systém `AplikacnaLogika` môžeme nahradiť iným pod systémom (resp. pod systémami) za predpokladu, že budú realizovať rovnaké rozhrania. Podobne by sme mohli nahradiť pod systém `GUI` s iným pod systém, ktorý bude využívať dané rozhrania.

Na základe rozhraní vieme odstrániť nežiadané väzby medzi pod systémami. Uvedená skutočnosť vedie k architektonicky veľmi pružnému a prispôsobiteľnému systému.

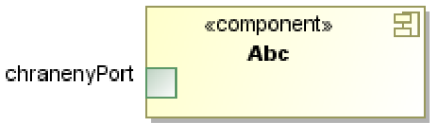
3.3 Porty komponentov

Porty signalizujú určitý bod v interakcií medzi komponentom a jeho okolím. Zoskupuje množinu prístupných a požadovaných rozhraní. Na obrázku Obr. 7 vidíme komponentu `Kniha`, ktorá obsahuje port s názvom `prezentacia`. Tento port sa skladá z sprístupneného rozhrania `IZobrazit` a požadovaného rozhrania `IZobraziť` médium.



Obr. 7: Notácia portu a jej rozhraní

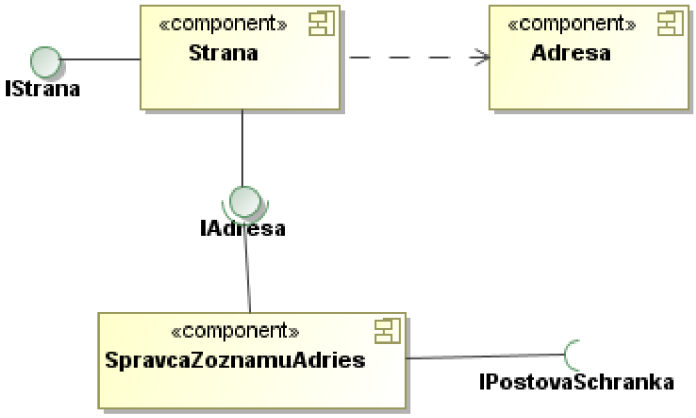
Pre porty môžeme taktiež definovať ich viditeľnosť. Ak je port nakreslený tak, že prekrýva ohraničujúci obdĺžnik komponenty, znamená to, že je verejný, čiže jej požadované a sprístupnené rozhrania sú verejné. Port môže byť však chránený alebo súkromný. V UML sa to vyznačuje tak, že v port je zanorený vo vnútri komponenty.



Obr. 8: Notácia chráneného portu

3.4 Väzby medzi komponentmi

Niektoré komponenty môžu byť navzájom závislé. Aby sa takýmto závislostiam predišlo, komunikáciu medzi dvoma komponentmi zabezpečíme cez rozhrania. Ako príklad väzieb medzi komponentmi si uvedieme príklad na obrázku Obr. 9.



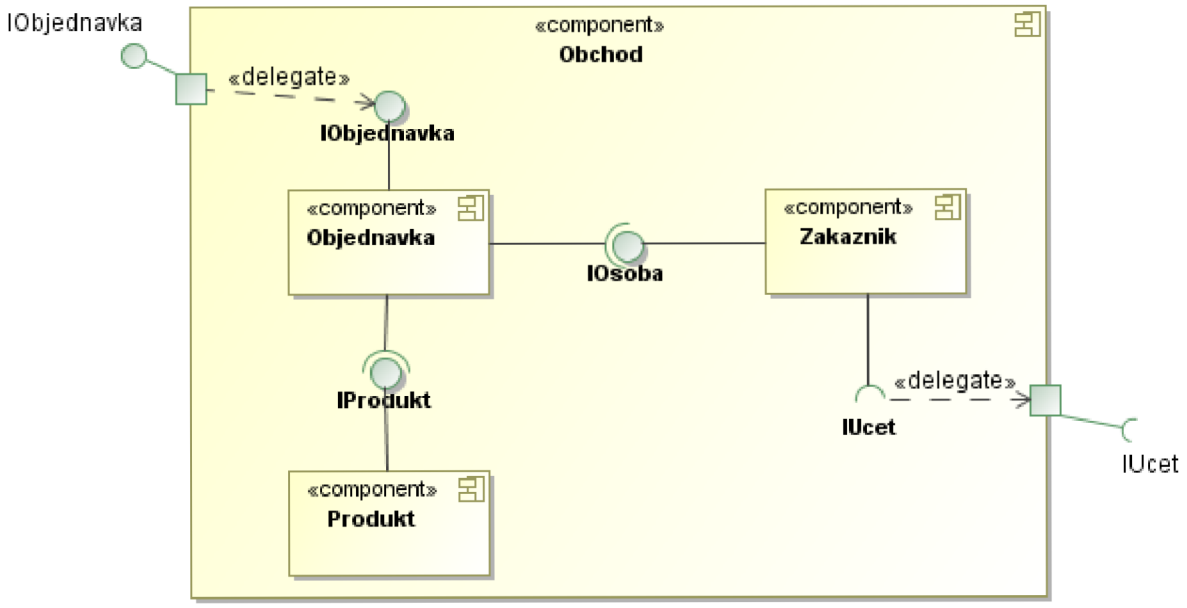
Obr. 9: Notácia väzieb medzi komponentmi

Na obrázku môžeme vidieť, že komponent `Strana` realizuje rozhrania `IStrana` a `IAdresa`, pričom je priamo závislá na komponentu `Adresa`. Komponent `SpravcaZoznamuAdries` pomocou rozhrania `IAdresa` využíva služby komponenty `Strana` a vyžaduje rozhranie `IPostovaSchranka`.

3.5 Vnútorná štruktúra komponentov

V komponent diagramoch je možné modelovať vnútornú štruktúru komponentov. Pri znázornení vnútornej štruktúry komponentu vložíme jej vnútorné súčasti alebo súčasti pripojené externe pomocou relácií závislostí.

Dôležitú roľu hrajú pri zobrazovaní vnútornej štruktúry komponenty jej porty. Komponent deleguje obyčajne zodpovednosť definovanú prostredníctvom svojich rozhraní na niektorú z vnútorných súčastí.



Obr. 10: Vnútorná štruktúra komponenty Obchod

Na obrázku Obr. 10 komponent `Obchod` poskytuje rozhranie `IObjednavka` a vyžaduje rozhranie `IUcet`. Vnútorný komponent `Objednavka` poskytuje služby okolitému svetu, ktoré sú deklarované v rozhraní `IObjednávka`, kým komponent `Zakaznik` využíva služby, ktoré prijíma cez rozhranie `IUcet`.

4 Eclipse Modeling Project

Eclipse Modeling Project je projekt zameraný na vývoj a propagáciu na modeloch založených vývojových technológií v rámci platformi Eclipse. Poskytuje jednotný súbor modelovacích frameworkov, nástrojov a implementačných štandardov.

V úvode kapitoly si definujeme, na základe literatúry [3], význam pojmu Doménovo-špecifický jazyk a popíšeme štruktúru Eclipse Modeling Project. Povieme si, že aké projekty tento celok tvoria. Definujeme pojmy ako abstraktný a konkrétny syntaktický vývoj.

V druhej časti tejto kapitoly sa zameriame na Eclipse Modeling Framework a Graphical Modeling Framework. Popíšeme význam EMF, GMF a ich životný cyklus vývoja použitím týchto technológií. Zdroje som čerpal z veľkej časti z literatúr [3], [7], [17] a [18].

4.1 Doménovo-špecifický jazyk

Doménovo-špecifické jazyky (ďalej DSL) tvoria triedu programovacích alebo špecifikačných jazykov na popis a riešenie úloh obmedzenej, konkrétnej problémovej doménovej oblasti. Zložitejšiu doménovú oblasť môže reprezentovať viac DSL jazykov, ktoré spolu tvoria rodinu DSL jazykov.

Podobný prístup ako DSL nie je novým pre špecificky účelové programovacie jazyky alebo modelovacie jazyky, ale tento názov sa stal populárnym a to vďaka aj vzostupu doménovo-špecifického modelovania.

Príklady na využitie DSL jazykov môžeme uviesť HTML, Logo pre deti, Verilog a VHDL hardvérové popisné jazyky, Mathematica a Maxima pre jazyky popisujúce matematické výrazy, SQL jazyk pre relačné databázy, YACC jazyk popisujúci gramatiku syntaktickej analýzy, jazyky na tvorbu regulárnych výrazov pre lexikálne analyzátory, Eclipse Modeling System pre tvorbu jazykov na reprezentáciu diagramov, atď.

Môžeme povedať, že DSL je meta-jazyk popisujúci údajovú reprezentáciu cielenej doménovej oblasti. Z pohľadu modelovacích nástrojov tento meta-model slúži na popis štruktúry modelu. Výsledný model potom na základe meta-modelu môže byť pretransformovaný na iný model.

Ako príklad na reprezentáciu konkrétnej oblasti, si zoberme jazyk UML ako univerzálny jazyk, ktorý sa skladá z niekoľkých doménovo-špecifických jazykov ako sú stavové automaty, štrukturálne definície, prípady použitia, atď. Každý doménovo-špecifický jazyk z UML pokrýva nejakú oblasť z vývoja softvéru. Môžeme tak povedať, že UML pokrýva doménu softvérového vývoja. [3]

Ďalej UML je modelovací jazyk, ktorý v oblasti modelom riadenom softvérovom vývoji (Model-driven Software Development) je používaný na generovanie zdrojového kódu v podobe nejakého cieľového programovacieho jazyka. Všeobecné mapovania v UML nám na tento účel obyčajne nevystačia, takže vzniká potreba na ďalšiu špecifickosť.

Riešením je umožniť vytvoriť modelovací jazyk viac špecifický cieľovému frameworku, ako napr. je UML Profile pri vývoji Java EE aplikácií. Ďalšou možnosťou je však začať vytvárať vlastné domény špecifických jazykov, ktoré obsahujú práve to, čo potrebujeme. [3]

Môžeme si položiť otázku: „Pribúdajúce definície a pohľady sa stávajú príčinou, prečo by sme mali začať vytvárať vlastný DSL?“. Väčšina ľudí však radšej začína s niečím malým a vytvára ho podľa vlastných potrieb. Aj samotný UML sa takto začal rozvíjať. Dnes ju však tvorí množina súvisiacich nástrojov, ktorá je veľká a zložitá, zatiaľ čo nástroje pre rýchly vývoj DSL produktov sú čoraz viac dostupnejšie.

To znamená toľko, že priebehu tvorby a rozširovania vlastného DSL (resp. rodiny vlastných DSL), môžeme skončiť niečím podobným k jazyku UML. Rozdiel vzniká vtom, že používame v rámci našej organizácie modely, transformačné definície, generačné zariadenia, ktoré vyhovujú naším potrebám. Idea je vtom, že nezávisle na tom, že náš DSL je definovaný pomocou UML alebo menším jazykom, ako napr. EMF Ecore, môžeme vytvoriť sadu nástrojov veľmi generatívnym spôsobom okolo nášho DSL.

Z historického hľadiska to tak vždy nebolo. Modelári museli kupovať drahé, nepružné, uzavreté modelovacie nástroje, ktoré by potrebovali nevyhnutne upraviť k svojim potrebám. Dnes môžeme vyvíjať vlastné nástroje využitím schopností veľmi silnej open source nadácie poskytovanej Eclipse Modeling Project.

Knižnice modelov a transformácií, ktoré budú taktiež k dispozícií pre opätovné použitie pre DSL aplikácie, vychádzajú z techník MDSD, sa stávajú čoraz atraktívnejšie. Ako príklad si zoberme projekt Generative Modeling Technologies (GMT) sa už stavia ako knižnica. Vďaka dostupných DSL, sa dostatočne zmenšili abstraktné medze, čo spôsobuje, že MDSD je čo raz atraktívnejší. [3].

4.2 Štruktúra Eclipse Modeling Project

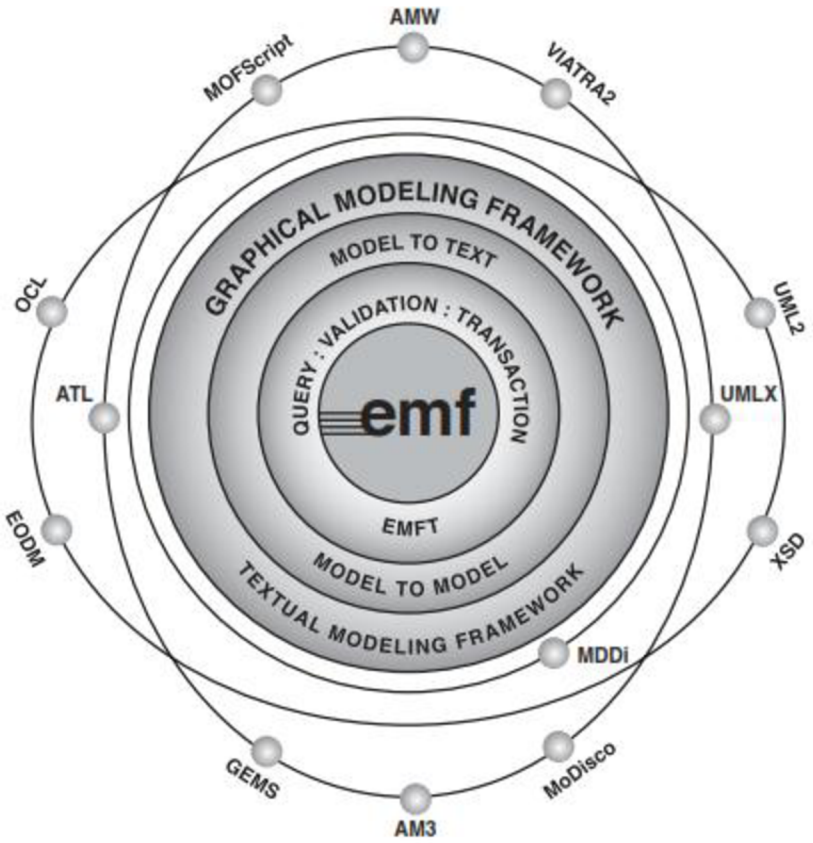
Eclipse Modeling Project je pomerne novým projektom na najvyššej úrovni Eclipse, ktorá tvorí veľkú zbierku projektov, ktoré sa zakladajú na MDSD technikách. Cieľom bolo vytvoriť zbierku pre koordináciu a zameranie sa na MDSD v Eclipse. Eclipse Modeling Project je logicky usporiadaný do množiny projektov, ktoré poskytujú nasledovné možnosti:

- Abstraktný syntaktický vývoj
- Konkrétny syntaktický vývoj
- Model-to-Model transformácie
- Model-to-Text transformácie

Na obrázku Obr. 11 vidíme, že jadro projektu Eclipse Modeling Project tvorí Eclipse Modeling Framework (EMF). Poskytuje podporu pre abstraktný syntaktický vývoj. Projektom EMF query, validácia a transformácia dopĺňajú základnú funkcionalitu ako Teneo a CDO pre databázovú

prezistenciu inštancií modelov. Okolo komponent abstraktného syntaktického vývoja sú transformačné techniky ako model na text (JET, Xpand) a model na model (QVT, ATL).

Okolo nich sa nachádza Graphical Modeling Framework (GMF), ktorý slúži na grafickú reprezentáciu modelov, a Textual Modeling Framework (TMF), ktorá sa používa na textovú reprezentáciu modelov. Nakoniec si spomenieme rad projektov (na obrázku zobrazené ako orbitály), ktoré reprezentujú modely, funkcie a výskumné iniciatívy využívajúce Eclipse Modeling Project. [3]



Obr. 11: Eclipse Modeling Project [3]

4.2.1 Abstraktný syntaktický vývoj

Jadrom každého DSL jazyka je jej abstraktná syntax, ktorá sa používa pri vývoji pre každý artefakt, vrátane grafickej konkrétnej syntaxe, model na model transformácií a model na text transformácií. Obyčajne ako prvý prvok každý jazyka DSL sa vyvíja jeho abstraktná syntax. Pre tento účel môžeme použiť Eclipse Modeling Framework (EMF). [3]

4.2.1.1 EMF Projekt

Ecore model slúži na definovanie meta-modelu DSL. Ďalej môžeme vylepšiť štruktúru a sémantiku DSL použitím jazyka Object Constraint Language (OCL). Okrem toho môžeme poskytnúť podporu na operácie transakcií, dotazov a validácie.

Ecore model poskytuje dobrý formát pre jazykové definície vhodné pre tradičné prístupy ako je napr. BNF. Model popísaný v Ecore môže mať niekoľko definícií konkrétnej syntaxe na generovanie textových a grafických editorov.

Niektoré komponenty, dostupné v Eclipse Modeling Project, rozširujú a dopĺňajú základné funkcie EMF. V rámci EMF sú komponenty, ktoré poskytujú operácie s dotazy, validáciu a transakčné funkcie. Rozširujú ich služby ako Sevice Data Objects (SDOs) a ďalšie technológie vyvinuté v rámci projektu Eclipse Modeling Framework Technology (EMFT). [3]

Transakcia modelu

Komponent transakcií modelu v EMF poskytuje transakčnú podporu pre editovanie EMF modelov. Spravovanie prístupu k transakčného editovaniu domény umožní aby viacerí klienti mohli čítať a písať modely súčasne.

Validácia modelu

Validation Framework dopĺňa transakčný framework, ktorý poskytuje integritu modelu. Síce jadro EMF poskytuje základnú podporu overovania, komponenty Validation Frameworku poskytujú väčší rozsah overovacích metód a umožnia vykonávať tzv. „živé“ overovanie inštancií modelu. Obmedzenia môžeme definovať v jazykoch Java a OCL. Audit a metrické vlastnosti GMF pre diagramy využívajú práve tento rámec.

Dotazy nad modelom

Podobne ako v databáze, obsah inštancií v modelu získavame dotazovaním. EMF modely môžu byť dotazované s využitím poskytovanej Java API, ale tzv. Model Query komponenty EMF poskytujú aj OCL a SQL alternatívy. Model Query poskytuje čisto iba programovateľné rozhranie, ale Model Search komponenty EMFT projektu umožnia integráciu s Eclipse Search UI (dialóg na hľadanie v Eclipse).

Model Search

Model Search komponent projektu EMFT ponúka rozsiahle možnosti vyhľadávania a integruje ho do dialógu hľadania Eclipse. Model Search umožní vyhľadávať na základe regulárnych výrazov a využitím OCL vyhľadávanie pre EMF a UML2 modely.

Porovnávania modelov

Podobne ako aj pri práce zdrojovým kódom, práca modelmi v rámci tímu vedie potrebe služieb na porovnanie a zlúčenie modelov. EMF Compare komponent z projektu EMFT Project poskytuje všeobecnú porovnávaciu a zlučovaciu podporu pre akýkoľvek Ecore model. EMF Compare komponenty využívajú štandardný porovnávací framework (Eclipse Comparison Framework) poskytujúce UI prostredie pre porovnanie dvoch verzií modelu.

Perzistenčné alternatívy

EMF má prostriedky poskytujúce flexibilné rozhranie, ktoré umožní základnú XMI serializáciu nahradiť alternatívami ako je napr. databázová perzistencia. Jedným takýmto rozšírením je z Teneo, ktorý je jeden EMFT projekt a využíva Java Data Objects[JDO]/Java Persistence Objects [JPOX]) na poskytovanie objektovo-relačného mapovania a perzistencie pre EMF modely.

CDO je ďalšia objektovo relačná mapovacia technológia, ktorá umožní databázovú perzistenciu. Jej nová komponent, Java Content Repository Management (JCR), umožní perzistenciu inštancií EMF modelov kompatibilného repozitárom Java Specification Request (JSR).

4.2.2 Konkrétny syntaktický vývoj

Konkrétna syntax definuje textovú alebo grafickú reprezentáciu prvkov v modelovacom nástroji definovaných v abstraktnej syntaxi DSL. Eclipse Modeling Project na vývoj konkrétnej syntaxe poskytuje Graphical Modeling Framework a Textual Modeling Framework.

Graphical Modeling Framework

Projekt GMF poskytuje grafickú konkrétnu syntax. Využitím GMF vieme vytvoriť grafický zápis pre DSL a mapovať ho do abstraktnej syntaxe. Tieto modely vytvárajú grafické diagram editory. [3]

Textual Modeling Framework

Pre tých, ktorý uprednosťňujú textovú konkrétnu syntax, bol vytvorený TMF projekt. Po vytvorení gramatiky z doménového modelu, môžeme využiť generátory, ktoré sa zameriavajú na platformu Eclipse, pre vytvorenie vysoko kvalitných textových editorov, doplnených zvýrazňovaním syntaxe. [3]

4.2.3 Transformácia modelu

Niekedy je výhodné mať k dispozícií mechanizmus, ktorý dokáže previesť model definovaný v abstraktnej syntaxe, na iný model alebo dokument. Eclipse Modeling Project ponúka komponenty slúžiace na model na model a model na text transformácie. [3]

Transformácia Model na model

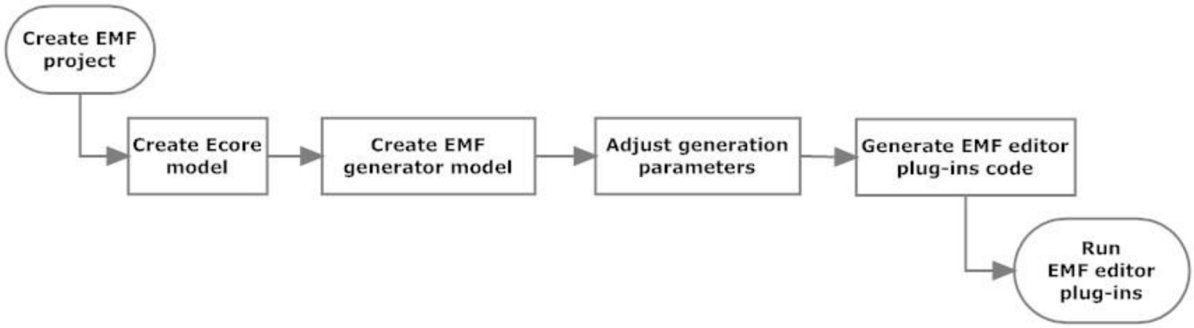
Na základe definície abstraktnej syntaxe nášho DSL, môžeme definovať modelové transformácie na generovanie nových modelov alebo textových výstupov. V prvom prípade budeme vyvíjať transformácie typu model na model využitím jazyka s názvom Operational Mapping Language (OML). Projekt Model-to-model Transformation (M2M) ponúka aj ďalšie alternatívy ako sú ATL a QVT Relations.

Transformácia Model na text

V Eclipse Modeling Project existujú viaceré alternatívy na transformácie typu model na text. Jednou najznámejšou je komponent Java Emmitter Templates (JET), ktorú EMF ako samotný používa. Avšak čím ďalej, tým viac populárnejším sa stáva Xpand šablónový motor, ktorý sa používa, ako rozšírenie, v GMF projektoch.

4.3 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) tvorí jadro Eclipse Modeling Project. Jedná sa modelovací framework a taktiež nástroj na generovanie kódu aplikácií založených na štruktúrovanom dátovom modeli.



Obr. 12: Životný cyklus vývoja EMF aplikácie [18]

Diagram na obrázku Obr. 12 poukazuje na postup tvorby aplikácie založenej na EMF modeli:

1. **Tvorba EMF modelu** – v tejto fáze EMF model môžeme definovať využitím XML schéma, UML nástroja a špecifikovaním Java rozhraní označených anotáciami.
2. **Vytvorenie modelu EMF Generátora** – na základe vytvoreného EMF modelu vývojár si môže dať vygenerovať model EMF generátora, ktorý umožní nastaviť parametre transformácie EMF modelu na Java zdrojový kód aplikácie.
3. **Generovanie EMF editoru** – transformácia EMF modelu na Java zdrojový kód na základe nastavených parametrov v modelu EMF generátora.

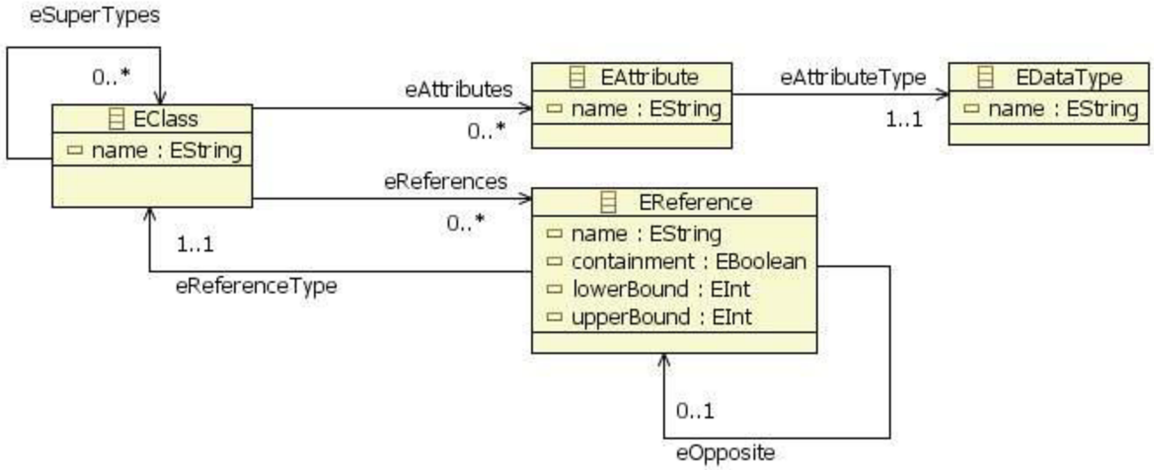
Obyčajne EMF obsahuje tri hlavné komponenty EMF.Core, EMF.Edit a EMF.Codegen [18].

- **EMF.ECore** - obsahuje dva prvky:
  - **Ecore model** – používa sa na definovanie EMF modelu
  - **Podpora počas behu** – poskytuje model perzistencie, notifikáciu zmien a reflexívne API.
- **EMF.Edit** – zabezpečuje zobrazovanie a editáciu inštancií v modeli v základnej stromovej štruktúre.

- **EMF.Codegen** – poskyuje generovanie EMF modelu na vybudovanie jednoduchého editoru vo forme stromovej štruktúry. [18]

4.3.1 EMF ECore model

Na reprezentáciu konkrétnej doménovej oblasti, EMF poskytuje mata-model nazývaný ECore model. Tento model umožní popísať štruktúru konkrétnej doménovej oblasti ako zjednodušený model tried.

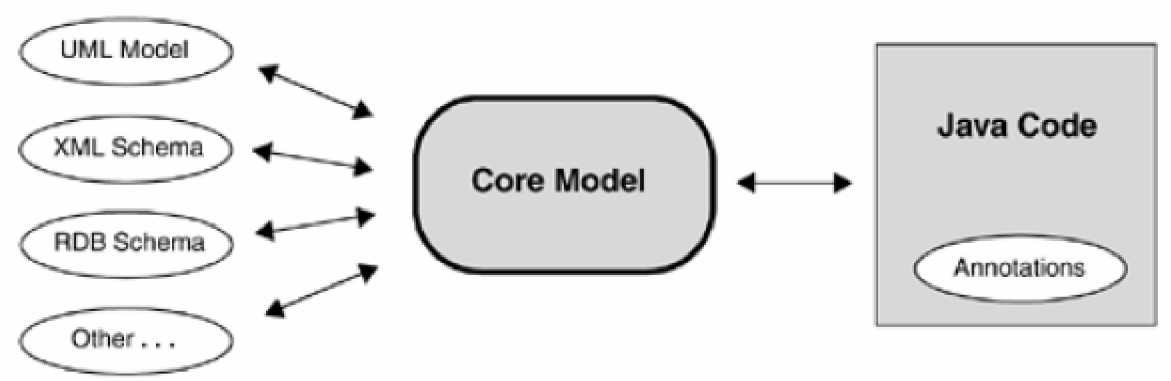


Obr. 13: Meta-model ECore modelu [18]

Meta-model Ecore modelu obsahuje nasledujúce triedy:

- **EClass** – používa sa na reprezentáciu modelovaných tried. Trieda má názov typu reťazca a môže obsahovať atribúty (`eAttributes`) a referencie (`eReferences`) na iné triedy. Na zabezpečenie viacnásobnej dedičnosti medzi triedami slúži asociácia `eSuperTypes`, ktorá zaznamenáva všetkých predkov triedy
- **EAttribute** – značí jeden atribút v modelovanej triede typu `EClass`. Každý atribút má názov a konkrétny dátový typ (`EDateType`).
- **EDataType** – reprezentuje typ atribútov typu `EAttribute`. Dátový typ musí byť pomenovaný.
- **EReference** – používa sa na vytvorenie asociácií medzi triedami `EClass`. Trieda `EReference` má názov, logickú premennú vyznačujúcu kompozíciu, dolnú a hornú hranicu a referenciu na triedu `EClass`. Asociácia `eOpposite` zabezpečuje vytvorenie obojsmernej relácie.

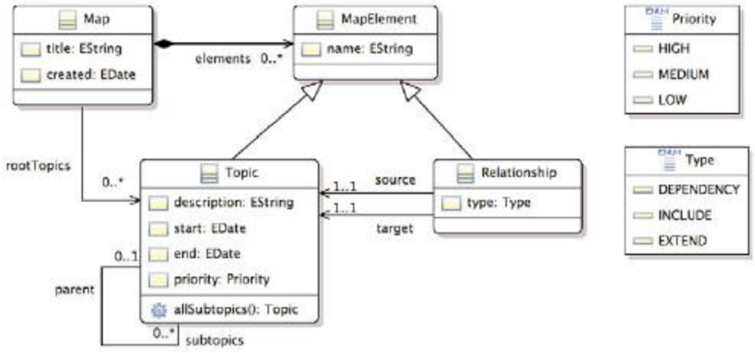
Na vytvorenie ECore modelu EMF podporuje ECore editor, UML, XML schéma, anotovaných Java rozhraní a XMI.



Obr. 14: Podporované formáty perzistencie modelov [7]

Ecore Editor

Jedná sa o editor ECore modelu založený na stromovej štruktúre. Entity modelu, ich atribúty a relácie medzi nimi sa zobrazujú v editoru formou štrukturovaného stromu. Ďalej balík nástrojov EMF Tools ponúka plug-in ECore Class Diagram Editor, ktorým môžeme ECore model editovať ako model tried využitím diagramu tried.



Obr. 15: Mindmap model v ECore Class Diagram Editor [3]

UML nástroj

Môžeme použiť UML modelovací nástroj v ktorom navrhneme model tried. EMF podporuje importovanie Rational Rose modelu alebo UML modelu na generovanie rovnocenného ECore modelu.

XML Schéma

EMF umožní transformovať XML Schéma (XSD) súbor na ECore model . Mapuje prvky XML schéma, ktoré zodpovedajú ECore prvkom.

XML Metadata Interchange

EMF používa XMI na serializáciu ECore modelov. XMI umožní interoperabilitu a zjednotenie metód definovania ECore modelu.

Anotované rozhrania v Jave

ECore môžeme modelovať použitím Java rozhraní, ktoré musia byť označené anotáciou `@model`. Model EMF generátora podľa týchto anotácií vie, že aké entity (resp. triedy) má generovať ako implementácie rozhraní entít modelu.

4.3.2 Transformácia modelu ECore na Java kód

Veľkou výhodou EMF je, že z ECore modelu dokáže vygenerovať Java zdrojový kód modelu a jednoduchý editor na jej editovanie. Transformácia ECore modelu na Java zdrojový kód sa uskutočňuje cez model EMF generátora (súbor **.genmodel*) v ktorom je možné nastaviť parametre generovania. V tomto modeli môžeme nastaviť výstup generovania pomocou vlastných šablón, pre/post podmienky generovania a implementačné detaily využitím OCL anotácií.

Využitím modelu EMF generátora z ECore modelu môžeme vygenerovať zdrojový kód pre 4 rôzne Eclipse pluginy:

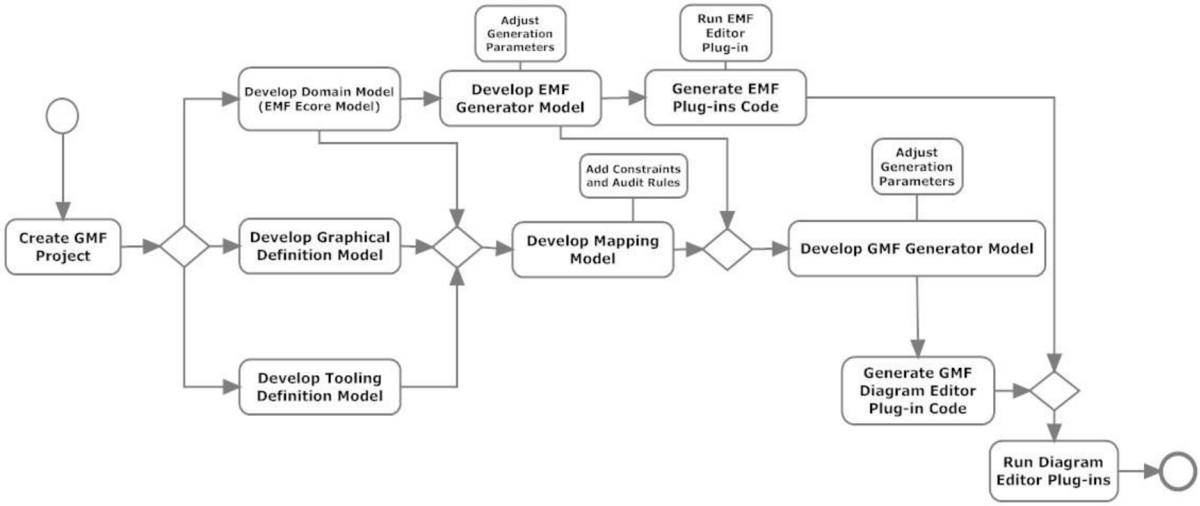
- **EMF.model** – obsahuje rozhrania a implementáciu generovaných entít z ECore modelu.
- **EMF.edit** – zabezpečuje zobrazovanie a editáciu inštancií v modeli v základnej stromovej štruktúre.
- **EMF.editor** – implementuje jednoduchý modelovací nástroj na základe stromovej štruktúry.
- **EMF.test** – jednotkové testy na testovanie funkcionality generovaného editoru.

EMF počas „znovu generovania“ už existujúceho kódu používa komponentu JMerge na zabránenie prepísania modifikácií, ktoré vykonal programátor už v existujúcom kóde. Generovaný Java kód je anotovaný `@generated` javadoc značkou na odlíšenie generovaných tried EMF generátorom.

V prípade ak túto značku vymažem alebo nahradíme značkou `@generated NOT`, JMerge komponent neprepíše označený kód.

4.4 Graphical Modeling Framework

Graphical Modeling Framework slúži na vývoj grafických modelovacích nástrojov (resp. editorov diagramov) zameraných na konkrétnu doménovú oblasť.



Obr. 16: Proces vývoja grafického modelovacieho nástroja [18]

Ako aj diagram na obrázku Obr. 16 ukazuje, vývoj modelovacieho nástroja sa delí na rôzne fázy:

- **Vývoj doménového modelu (EMF ECore Model)** – abstraktný syntaktický strom modelovacieho nástroja. Reprezentuje meta-model, ktorý určí aké údaje môžeme uložiť modelovanej domény.
- **Vývoj modelu EMF generátora (Develop EMF Generator Model) –** tento model umožní konfigurovať parametre transformácie ECore modelu do Java zdrojového kódu, ktorý vygeneruje plug-in-y doménového modelu a jednoduchého EMF editoru.
- **Vývoj modelu grafických definícií (Develop Graphical Definition Model) –** definície grafických značení uzlov a spojení v grafickom modelovacom nástroji.
- **Vývoj modelu nástrojov (Develop Tooling Definition Model)** – v tomto modeli definujeme paletu grafického modelovacieho nástroja.
- **Vývoj modelu mapovania (Develop Mapping Model)** – kombinuje modely doménový model, model grafických definícií a model nástrojov. Cieľom je mapovať, ktorá doménová entita bude reprezentovaná akým grafickým značením a akým nástrojom ju môže používateľ vytvoriť v grafickom modelovacom nástroji.
- **Vývoj Modelu GMF generátora (Develop GMF Generator Model)** – tento model umožní nastaviť dodatočné parametre transformácie Java kódu z mapujúceho modelu.

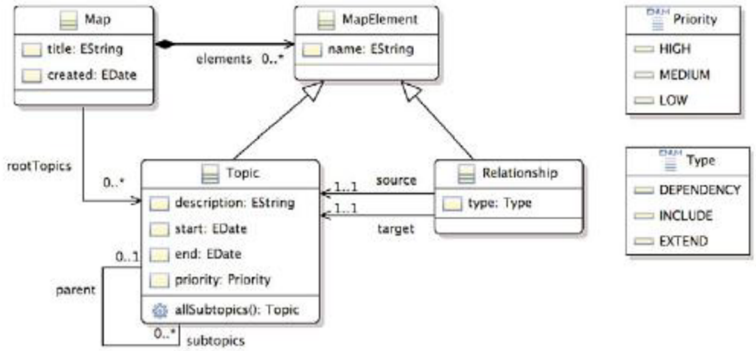
V ďalších častiach tejto podkapitoly budeme skúmať hore uvedené fázy vývoja grafického modelovacieho nástroja využitím technológie GMF. Tieto fázy predvediem na príklade na technológiu GMF z webových stránok spoločnosti Eclipse, ktorý má názov **Mindmap [15]**.

4.4.1 Doménového model

Doménový model tvorí abstraktný syntaktický strom vytváraného modelovacieho nástroja. V GMF doménový model sa obyčajne reprezentuje modelom vytvoreného v Eclipse Modeling Framework –u (EMF), ktorý vytvoriť ECore model, ktorý model slúži na:

- popis štruktúry modelov
- podporu pre udalosti pri zmene objektov počas behu
- podporu perzistencie cez základnú XMI serializáciu.
- poskytovania veľmi efektívneho API na manipuláciu s EMF objektmi.

ECore model obyčajne je možné editovať pomocou jednoduchého EMF editoru, ktorý ECore model reprezentuje ako hierarchickú stromovú štruktúru. Vývojári EMF však v balíku EMF Tools poskytujú nástroj ECore Diagram Editor, ktorý nám umožní modelovať ECore model vo forme diagramu.



Obr. 17: Mindmap ECore model [3]

Výsledný doménový model môžeme doplniť ďalšími informáciami využitím anotácií v ECore. Anotácie popisovať obsahovať OCL validačné pravidlá, definovanie funkcionalitu metód využitím jazyka OCL, atď.

4.4.2 Model EMF generátora

Transformácia ECore modelu na Java zdrojový kód sa uskutoční cez model EMF generátora (súbor **.genmodel*) v ktorom je možné nastaviť parametre generovania. Tento model umožní nastaviť výstup generovania pomocou vlastných šablón, pre/post podmienky generovania a implementačné detaily využitím OCL anotácií. Podrobnejšie informácie o transformácií ECore na zdrojový kód a modeli EMF generátora obsahuje kapitola *4.3.2 Transformácia modelu ECore na Java kód*.

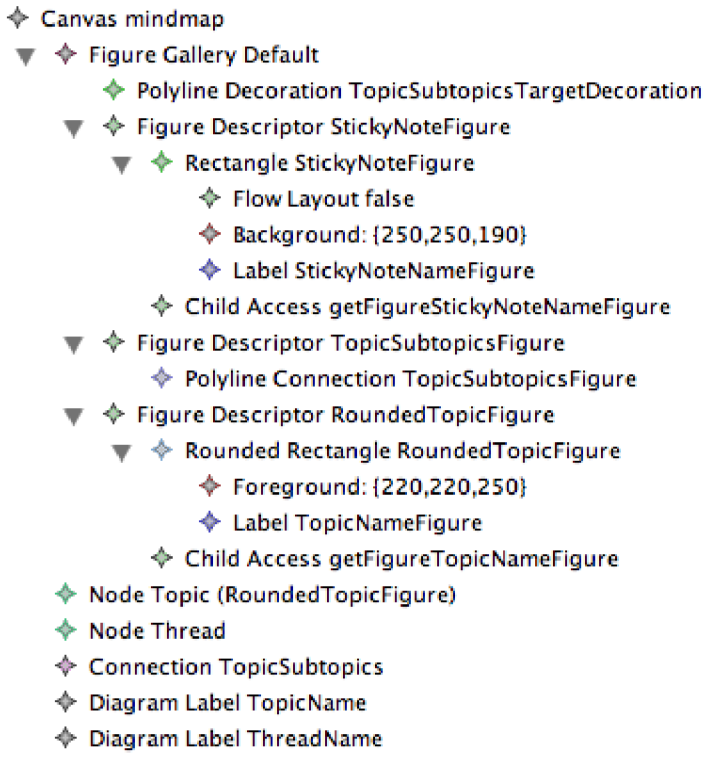
4.4.3 Model grafických definícií

Model grafických definícií je nezávislí model v GMF, ktorý slúži na definovanie vizuálnej podoby uzlov a spojení medzi uzlami. Tento model je ukladaný v súbory **.gmfgraph.*

Model grafických definícií je editovaný EMF editorom, čiže ako hierarchický strom, ktorej koreňom je prvok `Canvas`, ktorý umožní definovať dva komponenty. Prvým komponentom je `Figure Gallery`, ktorý obsahuje prvky `Figure Descriptor`. Prvok `Figure Descriptor` definuje vizuálny tvar prvku na základe primitívnych prvkov ako sú obdĺžnik, farba pozadia, typ rozloženia atď.

Druhý komponent zahrnuje v sebe prvky:

- **Uzol (`Node`)** - definuje možné uzly v diagrame. Každý uzol musí definovať, ktorá figúra bude tvoriť jej vizuálnu podobu. Zvolená figúra by mala mať tvar uzavretého priestoru.
- **Spojenie (`Connection`)** - definuje možné spojenia v diagrame. Každé spojenie musí definovať, ktorá figúra bude tvoriť jej vizuálnu podobu. Zvolená figúra by mala mať tvar čiary.
- **Kabína (`Compartment`)** – tento prvok určuje figúry, ktoré vo výslednom diagram editore budú obsahovať pod prvky umiestnené vnútri figúry.
- **Štítok (`Diagram Label`)** – popisuje textové štítky v figúrach na ktoré v mapujúcom modeli bude možné mapovať hodnoty z doménového modelu (napr. ECore).

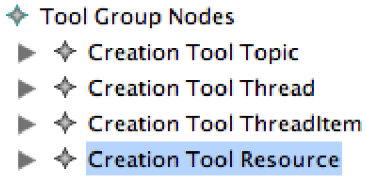


Obr. 18: Model grafických definícií [15]

4.4.4 Model nástrojov

GMF používa model nástrojov na definovanie palety, menu, kontext menu, panelu nástrojov. Tento model sa nachádza v súbore *.*gmftool*.

Model nástrojov je editovaný EMF editorom, čiže ako hierarchický strom. Koreňovým prvkom modelu je `Tool Registry`. Tento prvok môže mať po prvky `Menu Action`, `Predefined Menu`, `Context Menu`, `Popup Menu`, `Main Menu`, `Toolbar`, `Palette`. GMF zatiaľ implementuje len použitie prvku `Palette`, ktorá slúži na definovanie nástrojov, ktoré umožnia vytvoriť prvky v diagram editore.



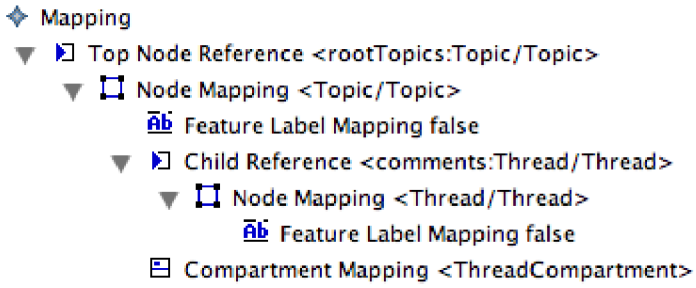
Obr. 19: Model nástrojov [15]

4.4.5 Model mapovania

V GMF mapujúci model umožní mapovať entity v doménovom modeli (ECore) k zodpovedajúcim figúram (definované v modeli grafických definícií) a k nástrojom na ich vytváranie (definovaných v modeli nástrojov).

Mapujúci model má koreňový prvok `Mapping`, ktorý môže obsahovať:

- **Canvas Mapping** – reprezentuje najvyššiu úroveň mapovania v diagram editore. Obyčajne spája koreňový prvok doménového modelu `Canvas` prvok z modelu grafických definícií a `Palette` prvkom z modelu nástrojov.
- **Top Node Reference** - mapuje na zodpovedajúce prvky z doménového modelu na figúry uzlov z modelu grafických definícií a na vytvárajúce nástroje z modelu nástrojov.
- **Link Mapping** - mapuje na zodpovedajúce prvky z doménového modelu figúry spojení z modelu grafických definícií a na vytvárajúce nástroje z modelu nástrojov.
- **Audit Container** – umožní definovať validačné pravidlá (`Audit rule`) na prvkami v doménovom modeli. Tieto validačné pravidlá sa môžu definovať využitím jazyka OCL, regulárneho výrazu alebo Java *Constraints* objektov.
- **Metric Container** – definuje opatrenia vo forme metrík (`Metric rule`). Pravidlá metrík sú definovateľné pomocou jazyka OCL, regulárneho výrazu alebo Java *Constraint* objektov.
- **Generic Style Selector** – GMF nie je ešte implementovaný.



Obr. 20: Model mapovania [15]

4.4.6 Model GMF generátora

GMF generátor v kombinácii modelu mapovania a modelu EMF generátora, dokáže vygenerovať plug-in vyvíjaného diagram editoru. Model GMF generátora umožní nastaviť parametre generovania.

GMF počas „znovu generovania“ už existujúceho kódu, podobne ako EMF, používa komponentu JMerge na zabránenie prepísania modifikácií, ktoré vykonal programátor už v existujúcom kóde. Generovaný Java kód je anotovaný `@generated` javadoc značkouna odlíšenie generovaných tried EMF generátorom.

5 Analýza požiadaviek a návrh aplikácie

5.1 Špecifikácia požiadaviek

Modelovací nástroj pre grafický návrh komponentových systémov by mal slúžiť ako plug-in pre platformu Eclipse, ktorý umožní modelovať komponent modely pomocou grafického editoru.

Používateľ bude mať možnosť si vytvoriť nový diagram na modelovanie systému založeného na komponentoch. V každom diagrame by mal vedieť uviesť názov diagramu a meno autora, ktorý diagram vytvoril.

Používateľ bude mať možnosť vložiť nové entity do modelovaného diagramu. Základné entity v diagrame tvoria komponent, rozhranie a artefakt. Komponent v systéme môže byť obyčajný alebo zložený. Obyčajný komponent umožní editovať jej názov a stereotyp. Zložený komponent je podobný obyčajnému komponentu, čiže musí mať uvedený názov a jej stereotyp, avšak tvorí jeden kontajner do ktorého používateľ bude vytvárať „dcérinné“ entity zloženého komponentu, čiže komponenty a rozhrania. Stereotypy komponent sú implicitne nastavené na stereotyp <<component>>.

Ďalšia použiteľná entita v modelovacom nástroji je rozhranie. Modelovací nástroj musí poskytovať možnosť editovať jej názov používateľom. Vytvorené rozhranie musí realizovať minimálne jeden komponent. V prípade, ak sa tak nestane, editor by ho mal upozorniť na túto skutočnosť. Poslednou entitou je artefakt, ktorá by mala umožniť meniť jej názov, stereotyp artefaktu. Artefakt implicitne bude mať nastavený stereotyp <<artifact>>.

Spojenia medzi dvoma komponentmi sa budú vytvárať cez jedno rozhranie pomocou relácií použitia a realizácie. Každá relácia použitia alebo realizácie môže byť pomenovaná.

Zložené komponenty budú poskytovať možnosť vytvorenia portu požadovaného a sprístupneného, ktorý sa napája na vonkajšie rozhranie cez reláciu použitia (v prípade portu typu požadovaného) alebo cez realizáciu (v prípade portu typu sprístupneného). Port nadväzuje spojenie vnútorného komponentu cez rozhranie, ktorú deleguje. Delegovanie rozhrania sa bude značiť reláciou <<delegate>>.

Komponenty a rozhrania môžu dediť jeden od druhého (len komponent od komponenty; alebo rozhranie od rozhrania) vlastnosti, čiže editor musí poskytovať reláciu dedičnosti pre rozhrania. Modelovací nástroj musí podporovať viacnásobnú dedičnosť. Ďalej bude overovať, či medzi entitami niekde nenastala cyklická dedičnosť.

Na vyznačenie implementácie komponenty v artefakte sa bude využívať relácia manifest medzi komponentom a artefaktom. Medzi dvoma entitami v diagrame by sa mala dať vytvoriť relácia

závislosť, ktorá bude mať editovateľný názov a stereotyp. Hodnota stereotypu bude implicitne nastavený na <<dependency>>.

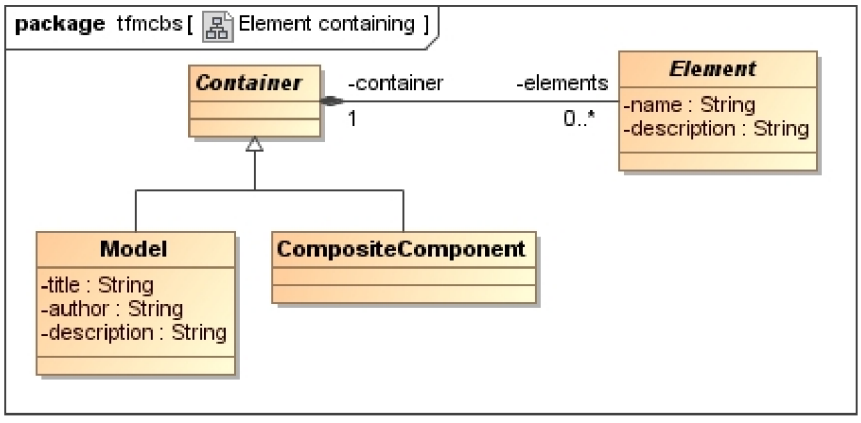
5.2 Návrh abstraktnej syntaxe modelovacieho nástroja

Jadrom každého DSL jazyka je jeho abstraktná syntax (viac. v kapitole *4.2.1 Abstraktný syntaktický vývoj*). Podľa špecifikácie požiadaviek, z predchádzajúcej kapitoly, som vytvoril návrh meta-modelu, ktorý bude neskôr implementovaný ako Ecore model v EMF.

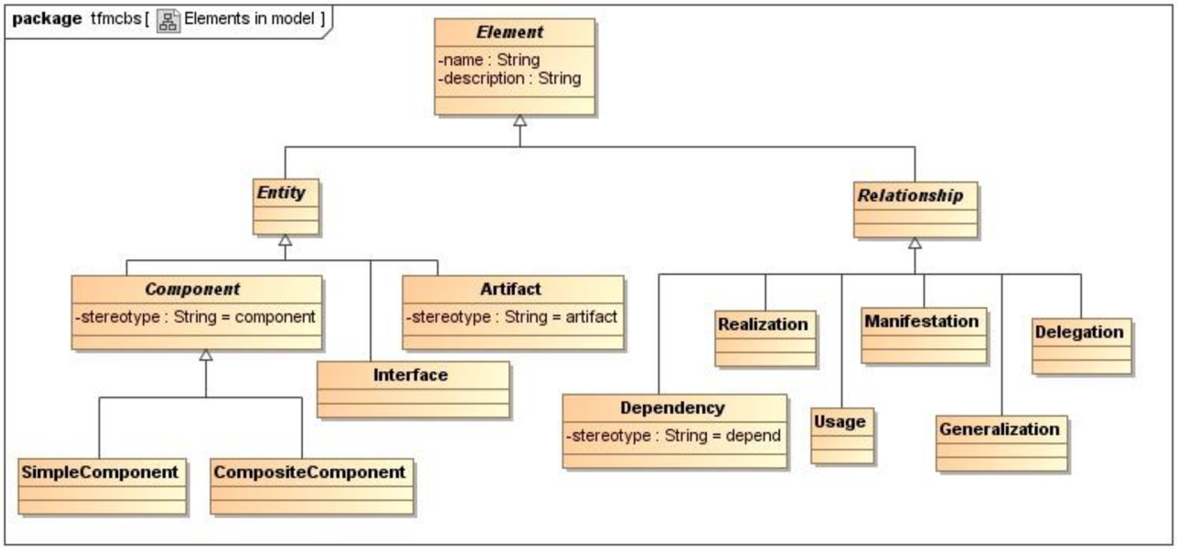
Každý model aby mohol uchovávať informácie o modelovanom svete, musí obsahovať v sebe nejaký kontajner, resp. nejaký zoznam v ktorom ukladá prvky. Modelovací nástroj pre grafický návrh komponentových systémov bude obsahovať dva typy kontajnerov:

- Jedným je špecializovaný kontajner `Model`, ktorý bude ako „koreňovým prvkom“ modelu.
- Druhým špecializovaným kontajnerom bude kontajner `CompositeComponent`, ktorý reprezentuje zloženú komponentu, tým pádom bude obsahovať prvky, ktoré sú pod prvky modelovaného zloženého komponentu.

Kontajnery sa zakladajú na abstraktnej triede `Container`, ktorý obsahuje iba zoznam na pod prvky kontajneru. Na diagrame Obr. 22 vidíme hierarchiu dedičnosti prvkov, ktoré je možné vytvoriť v modelovacom nástroji pre komponent systémy.



Obr. 21: Kontajneri prvkov v editore



Obr. 22: Hierarchia dedičnosti prvkov v editore

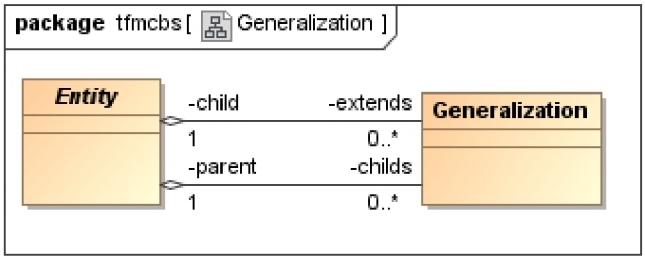
Vzhľadom na to, že kreslenie diagramov pripomína nám teóriu grafov, rozhodol som sa prvky diagramu začleniť do dvoch skupín, kde trieda `Entity` bude reprezentovať všetky uzly a trieda `Relationship` bude reprezentovať všetky relácie (resp. hrany grafu).

Entity v diagrame budú tvoriť komponenty (`Component`), rozhrania (`Interface`) a artefakty (`Artifact`). Z diagramu vyplýva, že v editore bude možné vytvoriť dva typy komponent a to sú obyčajný komponent (`SimpleComponent`) a zložený komponent (`CompositeComponent`). Obyčajný komponent reprezentuje komponent informačného systému ako všeobecný celok, kým zložený komponent umožní modelovať jej vnútornú štruktúru.

Relácie v editore budú tvoriť relácie ako závislosť (`Dependency`), realizácia (`Realization`), použitie (`Usage`), dedičnosť (`Generalization`), manifestácia (`Manifestation`). V nasledujúcich podkapitolách budeme skúmať súvislosti relácií a entít.

Relácia generalizácie

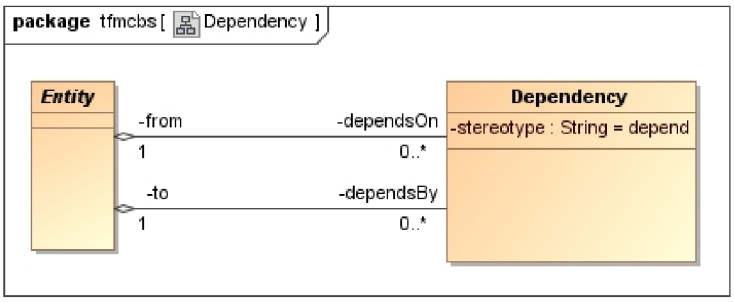
Relácia generalizácie (`Generalization`) nám umožní v modeli vytvoriť viac násobnú dedičnosť medzi entitami. Ako aj z diagramu na obrázku Obr. 23 vidíme, že entita môže dediť (atribút `extends`) a môže byť dedená (atribút `childs`) ľubovoľný krát. Do implementácií abstraktnej syntaxe bude treba pridať validačné pravidlá, aby sa v modeli mohli entity dediť jedine od entít rovnakého typu, čiže komponent od komponenty, rozhranie od rozhrania.



Obr. 23: Relácia generalizácie

Relácia závislosti

Diagram na obrázku Obr. 24 popisuje objekt reprezentujúci reláciu závislosti (`Dependency`). Atribútom `from` triedy `Dependency` značíme entitu, ktorá bude závislá, kým atribútom `to` značíme entitu, od ktorej je závislá prvá entita.

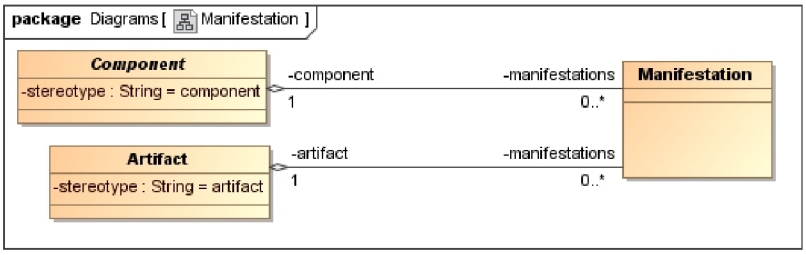


Obr. 24: Relácia závislosti

Relácia závislosti neobsahuje žiadne obmedzenia ako napr. pri relácií generalizácie (resp. dedičnosti). To znamená, že relácia závislosti môže byť aplikovaná medzi dvoma entitami ľubovoľného typu.

Relácia manifestácie

Obrázok Obr. 25 popisuje objekt reprezentujúci reláciu manifestácie (`Manifestation`). Atribútom `artifact` triedy `Manifestation` značí artefakt, ktorý obsahuje komponentu (atribút `component`).



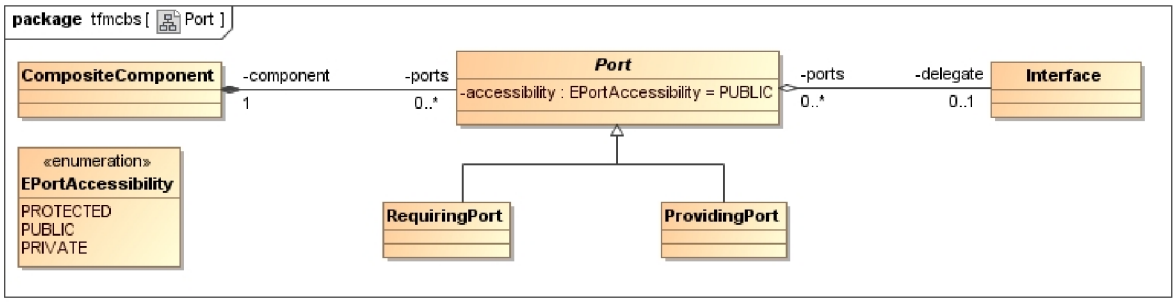
Obr. 25: Relácia manifestácie

Porty zloženého komponentu

Pred tým, ako sa pustíme do skúmania štruktúry relácií realizácie a používania, povieme si ako sú reprezentované porty v abstraktnej syntaxe modelovacieho nástroja pre zložené komponenty. Z diagramu na obrázku Obr. 26 vyplýva, že zložený komponent môže obsahovať porty typu požadovaný port (`RequiringPort`) a sprístupnený port (`ProvidingPort`).

Požadovaný port od inej komponenty z okolitého prostredia zloženého komponentu vyžaduje nejakú funkcionalitu/službu. Túto službu získa od komponenty použitím rozhrania, ktoré komponent, poskytujúci službu, implementuje. Sprístupnený port umožní poskytovať služby implementované zloženým komponentom pre okolité prostredie realizovaním rozhrania. Požadované/sprístupnené služby sú z/na port delegované cez rozhranie vnútri zloženej komponenty cez ktoré majú komponenty vnútri zloženej komponenty prístup k vonkajšiemu okoliu.

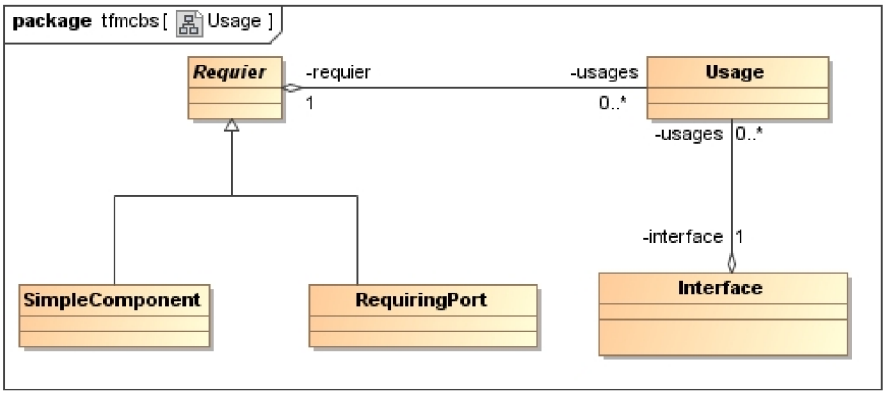
Porty majú taktiež definované viditeľnosť/prístupnosť cez atribút `accessibility` typu enumerátora `EPortAccessibility`, ktorá môže byť verejná (`PUBLIC`), chránená (`PROTECTED`), súkromná (`PRIVATE`). Tento atribút je implicitne nastavený na hodnotu `PUBLIC`.



Obr. 26: Porty v zložených komponentoch

Relácia použitia

Relácia použitia by sa mala v modelovacom nástroji vytvoriť medzi požadovaním rozhraním a obyčajným komponentom, prípadne požadujúcim portom zloženej komponenty. Na diagrame z obrázku Obr. 27 vidíme, že použitie rozhrania pomocou relácie použitia sa uskutoční referenciou objektu typu `Requier`. Práve tento abstraktný dátový typ nám zabezpečí, že relácia použitia sa bude dať vytvoriť jedine medzi požadovaným rozhraním a obyčajným komponentom (`SimpleComponent`) alebo požadujúcim portom (`RequiringPort`) zloženej komponenty, nakoľko obidve dedia vlastnosti abstraktnej triedy `Requier`.

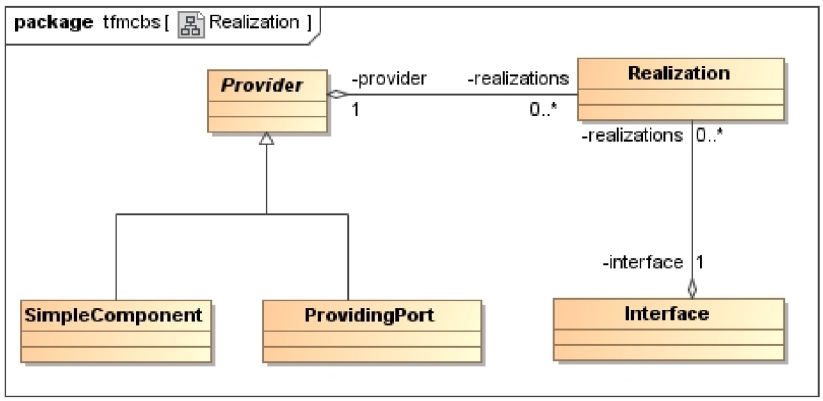


Obr. 27: Relácia použitia

Relácia realizácie

Relácia realizácie by sa v modelovacom nástroji mala dať vytvoriť sprístupneným rozhraním a obyčajným komponentom, prípadne požadujúcim portom zloženej komponenty, ktorá služby implementuje.

Na diagrame z obrázku Obr. 28 vidíme, že realizácia sprístupneného rozhrania sa uskutoční referenciou objektu typu `Provider`. Tento abstraktný dátový typ nám zabezpečí, že relácia realizácie sa bude dať vytvoriť jedine medzi sprístupneným rozhraním a obyčajným komponentom (`SimpleComponent`) alebo sprístupňujúcim portom (`ProvidingPort`) zloženej komponenty, nakoľko obidve dedia vlastnosti abstraktnej triedy `Provider`.



Obr. 28: Relácia realizácie

6 Vývoj abstraktnej syntaxe

Jadrom každého doménovo-špecifického jazyka (DSL) je jej abstraktná syntax, ktorá sa používa pri vývoji pre každý artefakt, vrátane grafickej konkrétnej syntaxe, model na model transformácií a model na text transformácií. Na vývoj abstraktnej syntaxe DSL modelovacieho nástroja pre grafický návrh komponent systémov bol použitý Eclipse Modeling Framework (EMF).

Táto kapitola sa bude zameriavať na vývoj abstraktnej syntaxe nového DSL modelovacieho nástroja vytvorením EMF ECore modelu. V druhej časti sa popíše validačný mechanizmus, ktorý rozširuje sémantiku ECore modelu, ktorý tvorí len meta-model modelovacieho nastroja.

6.1 ECore Model

Využitím EMF ECore som vytvoril meta-model modelovacieho nástroja, ktorý sa zakladá na modelu návrhu abstraktnej syntaxe DSL. Tento model je popísaný v kapitole *5.Analýza požiadaviek a návrh aplikácie*. Výsledný ECore model je v diagrame na obrázku Obr. 29.

Názov triedy:	Container
Je abstraktná trieda:	Áno
Rodičovské triedy:	Žiadne
Atribúty:	Žiadne

Referencie:

Názov	**Typ**	**Kardinalita**	**Kompozícia**
Elements	Element	0 .. *	Áno

Tab. 2: Trieda Container

Základným prvkom modelovacieho nástroja je abstraktný typ kontajner (`Container`), ktorý uchováva zoznam vnorených prvkov. Tieto prvky sú odkazované zoznamom `elements`. Kontajner má dva konkrétne špecializované typy ako model (`Model`) a zložený komponent (`CompositeComponent`). Typ `Model` sa môže vyskytnúť ako koreňový uzol hierarchického stromu modelu. Objekt typu `CompositeComponent` sa môže vyskytovať ľubovoľný krát v modeli, totiž reprezentuje špeciálny prípad komponentu, ktorý tvorí kontajner obsahujúci vnorené objekty popisujúce štruktúru zloženej komponenty.

Názov triedy:	Model
Je abstraktná trieda:	Nie
Rodičovské triedy:	Container

Atribúty:

Názov	**Typ**
Name	EString

Author	EString
Description	EString
Referencie:	Žiadne

Tab. 3: Trieda Model

Názov triedy:	CompositeComponent
Je abstraktná trieda:	Nie
Rodičovské triedy:	Container, Component

Referencie:

Názov	Typ	Kardinalita	Kompozícia
Ports	Port	0 .. *	Áno

Tab. 4: Trieda CompositeComponent

Ako už bolo naznačené, abstraktný typ `Container` obsahuje zoznam vnorených prvkov (potomkov abstraktného typu `Element`). Abstraktný typ `Element` sa špecializuje na typy `Entity` a `Relationship`.

Názov triedy:	Element
Je abstraktná trieda:	Áno
Rodičovské triedy:	Žiadne

Atribúty:

Názov	Typ
Name	EString
Description	EString

Referencie:	Žiadne

Tab. 5: Trieda Element

Objekt typu `Entity` obsahuje zoznam dedených typov (`extends`), zoznam dcériných typov (`childs`). Ďalej uchováva informácie o závislostiach medzi entitami. Zoznam `dependsBy` obsahuje závislosti v ktorých je entity závislá na inej entite, kým `depensOn` zahrňuje zoznam závislostí, v ktorých je iná entita závislá na tejto entite.

Názov triedy:	Entity
Je abstraktná trieda:	Áno
Rodičovské triedy:	Element
Atribúty:	Žiadne

Referencie:

Názov	Typ	Kardinalita	Kompozícia
extends	Generalization	0 .. *	Nie
childs	Generalization	0 .. *	Nie

dependsBy	Dependency	0 .. *	Nie
dependsOn	Generalization	0 .. *	Nie

Tab. 6: Trieda Entity

Entity v modelovacom nástroji tvoria komponenty (`Component`), rozhrania (`Interface`) a artefakty (`Artifact`). Typ `Interface` v zozname `usages` uchováva relácie použitia rozhrania a zoznamom `realizations` sú referencované relácie realizácie rozhrania. Kolekcia `delegates` slúži na skladovanie relácií delegácie v zložených komponentoch.

Názov triedy:	Interface
Je abstraktná trieda:	Nie
Rodičovské triedy:	Entity
Atribúty:	Žiadne

Referencie:

Názov	**Typ**	**Kardinalita**	**Kompozícia**
usages	Usage	0 .. *	Nie
realizations	Realization	0 .. *	Nie
delegates	Delegate	0 .. *	Nie

Tab. 7: Trieda Interface

Artefakt v modelovacom nástroji je reprezentovaný typom `Artifact`. Tento typ obsahuje zoznam `manifestations` reláciami manifestácie medzi komponentom a artefaktom.

Názov triedy:	Artifact
Je abstraktná trieda:	Nie
Rodičovské triedy:	Entity

Atribúty:

Názov	**Typ**	**Východisková hodnota**
Stereotype	EString	artifact

Referencie:

Názov	**Typ**	**Kardinalita**	**Kompozícia**
Manifestations	Manifestation	0 .. *	Nie

Tab. 8: Trieda Artifact

V modelovacom nástroji je možné ďalej vytvoriť dva typy komponent, preto sa abstraktný typ `Component` špecializuje na obyčajný komponent (`SimpleComponent`) a zložený komponent (`CompositeComponent`). Obyčajný komponent reprezentuje komponent informačného systému ako všeobecný celok, kým zložený komponent umožní modelovať jej vnútornú štruktúru.

Názov triedy:	Component
Je abstraktná trieda:	Áno

Rodičovské triedy:	Entity

Atribúty:

Názov	Typ	Východisková hodnota
Stereotype	EString	Component

Referencie:

Názov	Typ	Kardinalita	Kompozícia
manifestations	Manifestation	0 .. *	Nie

Tab. 9: Trieda Component

Typ `SimpleComponent` reprezentuje obyčajný komponent, ktorý popisuje všeobecne komponent v modeli. Komponent môže sprístupňovať a požadovať určité služby, preto typ `SimpleComponent` je potomkom typov `Provider` a `Requier`.

Názov triedy:	SimpleComponent
Je abstraktná trieda:	Nie
Rodičovské triedy:	Component, Provider, Requier
Atribúty:	Žiadne
Referencie:	Žiadne

Tab. 10: Trieda SimpleComponent

Relácie medzi entitami môžu tvoriť relácia závislosti (`Dependency`), relácia použitia (`Usage`), relácia realizácie (`Realization`), relácia delegácie (`Delegate`) a relácie dedičnosti (`Generalization`). Relácia v modeli je reprezentovaná abstraktným typom `Relationship`, na základe ktorého sú typy relácií špecializované.

Názov triedy:	Relationship
Je abstraktná trieda:	Áno
Rodičovské triedy:	Element
Atribúty:	Žiadne
Referencie:	Žiadne

Tab. 11: Trieda Relationship

Typ `Dependency` reprezentuje reláciu závislosti medzi dvoma objektmi typu `Entity`. Referenciou `from` značí entitu, ktorá bude závislá, kým referenciou `to` odkazuje na entitu, od ktorej je závislá prvá entita.

Názov triedy:	Dependency
Je abstraktná trieda:	Nie
Rodičovské triedy:	Relationship

Atribúty:

Názov	Typ	Východisková hodnota
Stereotype	EString	Depend

Referencie:

Názov	Typ	Kardinalita	Kompozícia
From	Entity	1	Nie
To	Entity	1	Nie

Tab. 12: Trieda Dependency

Typ `Generalization` umožní vytvoriť reláciu dedičnosti medzi dvoma objektmi typu `Entity`. Sémantiku typu bude treba rozšíriť o validačné pravidlá, aby sa v modeli mohli dediť entity jedine od entít rovnakého typu, čiže komponent od komponentu, rozhranie od rozhrania.

Názov triedy:	Generalization
Je abstraktná trieda:	Nie
Rodičovské triedy:	Relationship
Atribúty:	Žiadne

Referencie:

Názov	Typ	Kardinalita	Kompozícia
Parent	Entity	1	Nie
Child	Entity	1	Nie

Tab. 13: Trieda Generalization

Typ `Manifestation` umožní vytvoriť spojenie medzi artefaktom a komponentom. Referencia `artifact` reprezentuje artefakt, kým referencia `component` značí komponent medzi ktorými je vytvorená relácia manifestácie.

Názov triedy:	Manifestation
Je abstraktná trieda:	Nie
Rodičovské triedy:	Relationship
Atribúty:	Žiadne

Referencie:

Názov	Typ	Kardinalita	Kompozícia
artifact	Artifact	1	Nie
component	Component	1	Nie

Tab. 14: Trieda Manifestation

Typ `Usage` reprezentuje reláciu použitia rozhrania komponentom. Referencia `interface` značí rozhranie, ktoré je používané a referencia `requier` odkazuje na prvok, ktorý využíva implementované služby na základe použitého rozhrania.

Názov triedy:	Usage
Je abstraktná trieda:	Nie
Rodičovské triedy:	Relationship
Atribúty:	Žiadne

Referencie:

Názov	Typ	Kardinalita	Kompozícia
interface	Interface	1	Nie
requier	Requier	1	Nie

Tab. 15: Trieda Usage

Typ `Realization` značí realizáciu (resp. implementáciu) rozhrania konkrétnym komponentom. Referencia `interface` obsahuje referenciu na realizované rozhranie, kým `provider` značí prvok, ktorý rozhranie realizuje.

Názov triedy:	Realization
Je abstraktná trieda:	Nie
Rodičovské triedy:	Relationship
Atribúty:	Žiadne

Referencie:

Názov	Typ	Kardinalita	Kompozícia
interface	Interface	1	Nie
provider	Provider	1	Nie

Tab. 16: Trieda Realization

Relácia delegácie (`Delegate`) umožní delegovať vstup/výstup z portu na rozhranie v zloženom komponente. Referencia `interface` odkazuje na delegované rozhranie a referencia `port` na port zloženého komponentu.

Názov triedy:	Delegate
Je abstraktná trieda:	Nie
Rodičovské triedy:	Relationship
Atribúty:	Žiadne

Referencie:

Názov	Typ	Kardinalita	Kompozícia
interface	Interface	1	Nie
port	Port	1	Nie

Tab. 17: Trieda Delegate

Typ `Requier` je abstraktná trieda, ktorá značí prvky, ktoré využívajú služby implementované inými komponentmi na základe konkrétneho rozhrania. Zoznam `usages` obsahuje referencie na relácie použitia (`Usage`), ktoré značia rozhrania, cez ktoré sa využívajú služby.

Názov triedy:	Requier
Je abstraktná trieda:	Áno
Rodičovské triedy:	Žiadne

Atribúty:	Žiadne

Referencie:

Názov	**Typ**	**Kardinalita**	**Kompozícia**
usages	Usage	0 .. *	Nie

Tab. 18: Trieda Requier

Typ `Provider` značí prvky, ktoré implementujú rozhrania a tým poskytujú určité služby okoliu. Zoznam `realizations` obsahuje relácie realizácií (`Realization`), ktoré značia rozhrania, podľa ktorých sú služby implementované.

Názov triedy:	Provider
Je abstraktná trieda:	Áno
Rodičovské triedy:	Žiadne
Atribúty:	Žiadne

Referencie:

Názov	**Typ**	**Kardinalita**	**Kompozícia**
realizations	Realization	0 .. *	Nie

Tab. 19: Trieda Provider

Prvky, popisujúce štruktúru zloženého komponentu, komunikujú okolím cez porty. Abstraktný typ `Port` reprezentuje porty zloženého komponentu. Referencia `component` odkazuje na zložený komponent, ktorému port patrí. Referencia `delegate` obsahuje jednu reláciu delegácie na rozhranie v kontajnery komponentu.

Názov triedy:	Port
Je abstraktná trieda:	Áno
Rodičovské triedy:	Žiadne

Atribúty:

Názov	**Typ**
name	EString
accessibility	EPortAccessibility

Referencie:

Názov	**Typ**	**Kardinalita**	**Kompozícia**
component	CompositeComponent	1	Nie
delegate	Delegate	0 .. 1	Nie

Tab. 20: trieda Port

Názov zoznamu:	EPortAccessibility

Hodnoty

- PUBLIC

• PROTECTED • PRIVATE

Tab. 21: Číselník EPortAccessibility

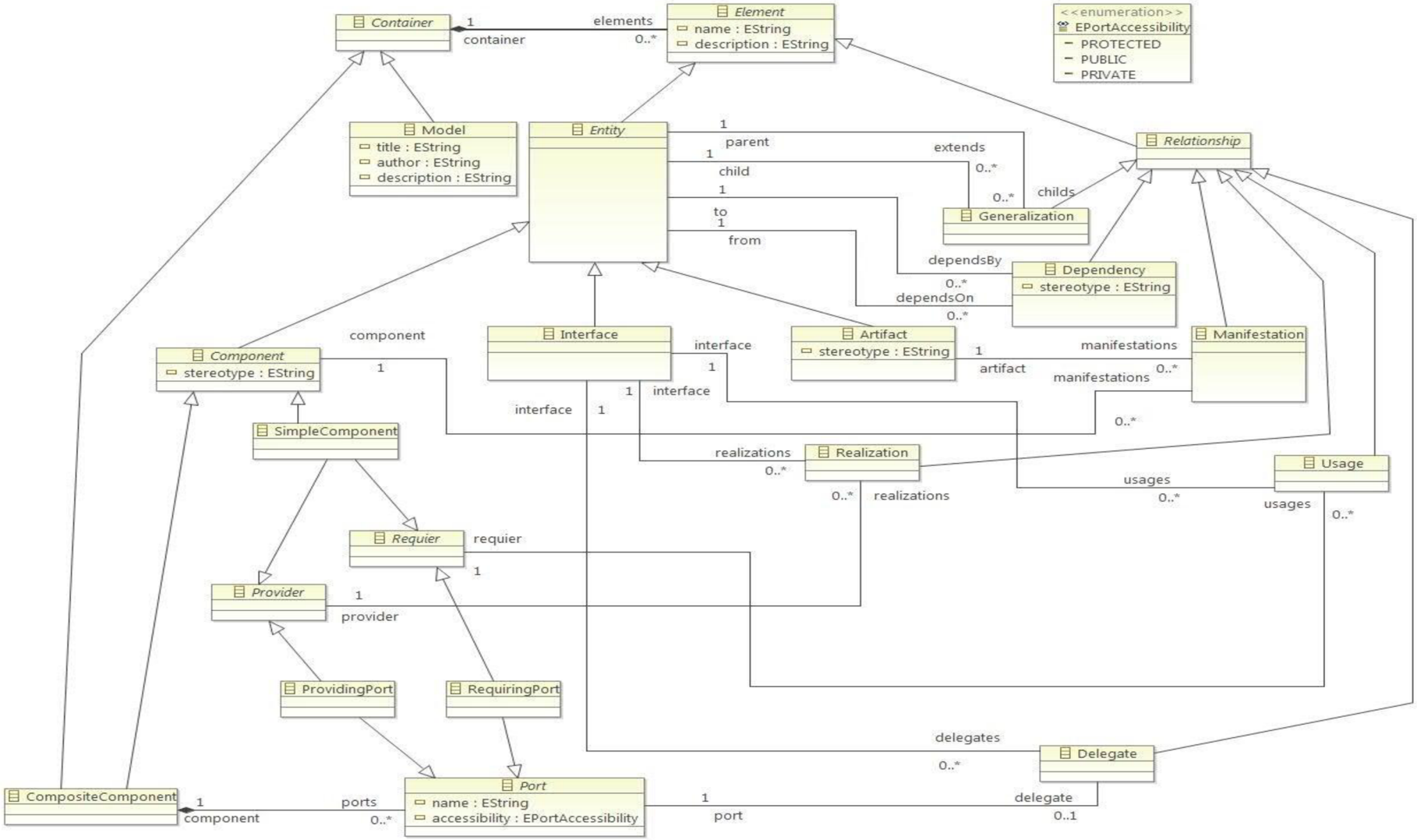
Abstraktný typ `Port` sa ďalej špecializuje na typy `ProvingPort` a `RequiringPort`. Typ `ProvidingPort` reprezentuje port sprístupňujúceho nejaké služby zloženým komponentom. Typ `RequiringPort` tvorí port požadujúce určité služby zloženým komponentom od okolia.

Názov triedy:	ProvidingPort
Je abstraktná trieda:	Nie
Rodičovské triedy:	Provider, Port
Atribúty:	Žiadne
Referencie:	Žiadne

Tab. 22: Trieda ProvidingPort

Názov triedy:	RequiringPort
Je abstraktná trieda:	Nie
Rodičovské triedy:	Requier, Port
Atribúty:	Žiadne
Referencie:	Žiadne

Tab. 23: Trieda RequiringPort



Obr. 29: Abstraktná syntax DSL modelovacieho nástroja

6.2 Transformácia ECore modelu na zdrojový kód

Po vytvorení meta-modelu modelovacieho nástroja pre podporu grafického návrhu komponent systémov, som vytvoril model EMF generátora (*EMF Generation Model*), ktorý umožní vygenerovať plug-in abstraktnej syntaxe z ECore modelu pre Eclipse platformu. V tomto modeli som nastavoval nasledovné pravidlá na generovanie:

Vlastnosť:	**Vlastnosť po anglicky:**	**Hodnota**
Názov modelu	Model name	Component Based System
Úroveň zhody	Compliance Level	6.0
Základný balík	Base Package	cz.vutbr.fit.xzemko01
Prefix	Prefix	tfmcbs
Prípona súboru	File Extensions	cbs

Tab. 24: Nastavenie vlastností v modeli EMF generátora

V ďalšom kroku som nastavil cesty pre generované plug-in-y a vygeneroval som plug-in-y **Model, Edit, Editor**. Zdrojový kód abstraktnej syntaxe sa generoval pod rovnaký projekt v ktorom je umiestnený model EMF generátora, čiže **cz.vutbr.fit.xzemko01.tool-for-modelling-of-component-based-systems**. Zdrojové kódy sú v balíkoch `cz.vutbr.fit.xzemko01.tfmcbs.model`.

Na zabezpečenie editácie prvkov v modeli vytvoril plug-in **Edit** s názvom **cz.vutbr.fit.xzemko01.tool-for-modelling-of-component-based-systems.edit**. Zdrojový kód týchto tried sa uchováva v balíku `cz.vutbr.fit.xzemko01.tfmcbs.model.provider`.

Plug-in jednoduchého editoru, ktorý zabezpečuje modelovanie abstraktnej syntaxe modelovacieho nástroja v Eclipse pomocou hierarchického stromu, sa nazýva c**z.vutbr.fit.xzemko01.tool-for-modelling-of-component-based-systems.editor** a triedy implementujúce tento balík sa nachádzajú v balíku `cz.vutbr.fit.xzemko01.tfmcbs.model.presentation`.

6.3 Validácia abstraktnej syntaxe

EMF ECore model popisuje štruktúru modelu tvoreného modelovacím nástrojom a takto dokáže zabrániť vloženiu nekonzistentných prvkov podľa pravidiel štruktúry modelu. Nedokáže však odhaliť porušenie komplexnejších pravidiel (napr. cyklická dedičnosť), preto základnú sémantiku abstraktnej syntaxe bolo treba rozšíriť o validačný mechanizmus.

Na overovanie konzistentného stavu modelu som vytvoril plug-in pre Eclipse, ktorý obsahuje validátori definujúce pravidlá, ktoré musia byť splnené, pre každý prvok v modeli. Tento plug-in je

implementovaný v projekte názvom **cz.vutbr.fit.xzemko01.tool-for-modelling-of-component-based-systems.validation**.

6.3.1 Validačný adaptér

Na implementáciu validácie v EMF sa používa Eclipse Validation Framework. EMF počas validácie volá, pre každý prvok zvlášť v overovanom modeli, metódu `validate(..)` z registrovaného objektu typu `EObjectValidator`. Vzhľadom na túto skutočnosť som sa rozhodol vytvoriť triedu `ValidationAdapter`, ktorá je potomkom typu `EObjectValidator`. Trieda `ValidatorAdapter` sa nachádza v balíku `cz.vutbr.fit.xzemko01.tfmcbs.validation`.

```
public class ValidatorAdapter extends EObjectValidator {
      @Override
      public boolean validate(EClass eClass, EObject eObject,
                  DiagnosticChain diagnostics, Map<Object, Object> context) {
            // ...
      }
}
```

Zdroj. kód 1: Trieda ValidatorAdapter

`ValidatorAdapter` je implementovaná na základe návrhového vzoru **adaptér** a po volaní metódy `Validator<E>.validate(..)` deleguje, podľa typu vstupného parametru `eObject`, validáciu na príslušný validátor. O validátoroch sa píše v nasledujúcej kapitole.

```
boolean result = true;

if(eObject instanceof Model) {
      result = ModelValidator.getInstance().validate((Model)eObject,
                                                      diagnostics, context);
} else if(eObject instanceof Interface) {
      result = InterfaceValidator.getInstance().validate((Interface)eObject,
                                                      diagnostics, context);
} // Ďalšie validátory
```

Zdroj. kód 2: Delegovanie validácie z ValidatorAdapter.validate(..) na príslušný validátor

6.3.2 Validátori

Validátori sú umiestnené v balíku `cz.vutbr.fit.xzemko01.tfmcbs.validation.validators`. Pre každý typ, definovaného v ECore modeli modelovacieho nástroja, som vytvoril príslušný validátor. Validátori implementujú rozhranie `Validator`, ktorý obsahuje verejnú metódu `validate(..)`. `ValidatorAdapter` potom využitím tohto rozhrania deleguje validáciu na validátor.

```
public interface Validator<E> {
	boolean validate(E item, DiagnosticChain diagnostics,
                                                    Map<Object, Object> context);
}
```

Zdroj. kód 3: Rozhranie Validator

Metóda `Validator.validate(..)` obsahuje 3 parametre:

- **item** – referencia na overovaný prvok v modeli
- **diagnostics** – diagnostické správy, ktoré sa zobrazia v Eclipse pre používateľa po dokončení validácie. Na zjednodušené vytvorenie diagnostických správ som vytvoril staviteľa `DiagnosticBuilder`.
- **context** – mapa na uchovávanie dodatočných informácií počas validácie. V implementáciách validátorov sa momentálne nepoužíva.

Ďalším Spoločným rysom validátorov je, že implementujú návrhový vzor **Singleton**, čiže počas behu aplikácie bude vytvorená iba jediná inštancia každého validátora. Na získanie inštancie validátora slúži metóda `getInstance()`, ktorá vracia referenciu validátora ako typ `Validator<E>`, kde generický typ `E` je typ z ECore modelu modelovacieho nástroja.

```
public final class ComponentValidator implements Validator<Component> {

	private static ComponentValidator componentValidator;

	private ComponentValidator() {
		entityValidator = EntityValidator.getInstance();
	}

	public static Validator<Component> getInstance() {
		if(componentValidator == null) {
			componentValidator = new ComponentValidator();
		}
		return componentValidator;
	}

	// Pravidla validacie …
}
```

Zdroj. kód 4: Implementácia návrhového vzoru Singleton v validátori ComponentValidator

Diagram na obrázku Obr. 30 zobrazuje implementované validátori. Keď porovnáme ECore model a týmto diagramom, vidíme, že validátori využívajú vlastnosť dedičnosti presne ako prvky implementované v ECore modeli. Avšak v prípade validátorov dedičnosť je nahradená využitím asociácie. V ďalších častiach kapitoly, popíšeme jednotlivé validátori.

ElementValidator

Najvšeobecnejším validátorom je `ElementValidator`, ktorý slúži na validáciu prvkov typu `Element` v modeli. Validátor dosial neobsahuje žiadne validačné pravidlá, totiž pre prvok `Element` nebolo treba zaviesť, žiadne obmedzenia.

EntityValidator

`EntityValidator` definuje pravidlá, ktoré slúžia na overenie prvkov typu `Entity` v modeli. Nakoľko v ECore modeli typ *Entity* je potomkom typu `Element`, `EntityValidator` používa validačné pravidlá z validátora `ElementValidator`.
`EntityValidator` implementuje ďalšie validačné pravidlá v metódach:

- `validateEntityHasName(..)` – overí, či prvok typu `Entity` má vyplnený atribút `name`, čiže má používateľ vyplnil názov entity.
- `validateEntityHasUniqueName(..)` – overí, či entita má jedinečný názov v modeli.
- `validateEntityHasNonCyclicGeneralization(..)` – overí, či entita nie je súčasťou cyklickej dedičnosti.
- `validateEntityHasUniqueExtends(..)` – zisťuje, či entita má vytvorenú každým rodičovským prvkom len jednu reláciu dedičnosti.

RelationshipValidator

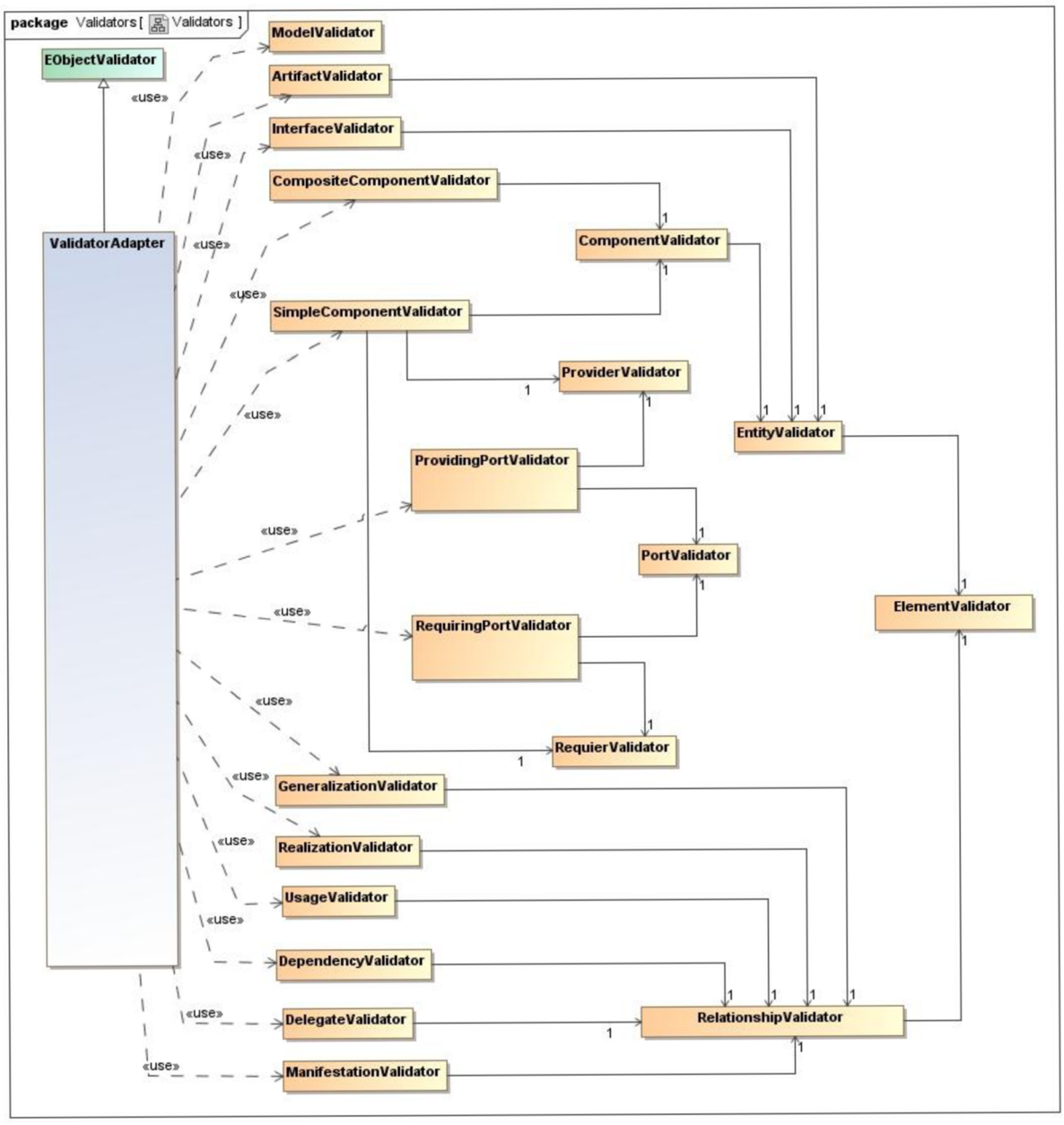
`RelationshipValidator` obsahuje validačné pravidlá definované na overovanie správnosti prvkov typu `Relationship`. `RelationshipValidator` využíva validačné pravidlá validátora `ElementValidator`, nakoľko typ `Relationship`, v ECore modeli, je potomkom typu `Element`. Validátor nedefinuje ďalšie validačné pravidlá.

PortValidator

`PortValidator` definuje validačné pravidlá pre prvky typu `Port`. Dosial implementuje jedno pravidlo v metóde `validatePortHasName(..)`, ktoré overí, či port má vyplnený názov, resp. atribút `name`.

ProviderValidator

`ProviderValidator` implementuje validačné pravidlá pre typ `Provider` z ECore modelu. Tento validátor obsahuje iba jedno validačné pravidlo implementované v metóde `validateUniqueRealization(..)`, ktoré overuje, že nie sú vytvorené dve relácie realizácie medzi tým istým rozhraním a poskytujúcim prvkom (resp. prvok typu `Provider`).



Obr. 30: ValidatorAdapter a validátori

RequierValidator

`RequierValidator` obsahuje validačné pravidlá na overovanie prvkov typu `Requier`. Tento validátor zatiaľ obsahuje jedno validačné pravidlo, implementované v metóde `validateUniqueUsage(..)`, ktorá overuje, že nie sú v modeli dve relácie použitia medzi tým istým rozhraním a požadujúcim prvkom (resp. prvkom typu `Requier`).

ProvidingPortValidator

`ProvidingPortValidator` definuje validačné pravidlá pre typ `ProvidingPort`. Tento validátor počas validácie využíva validátori `ProviderValidator` a `PortValidator`, totiž `ProvidingPort` je potomkom typov `Port` a `Provider`. `ProvidingPortValidator` neimplementuje ďalšie validačné pravidlá.

RequiringPortValidator

`RequiringPort` implementuje validačné pravidlá pre prvky typu `RequiringPort`. Nakoľko `RequiringPort` je potomkov typov `Requeir` a `Port`, `RequiringPortValidator` používa validačné pravidlá definované v validátoroch `RequierValidator` a `PortValidator`. Tento validátor však neimplementuje ďalšie validačné pravidlá.

ModelValidator

`ModelValidator` slúži na overenie správnosti koreňového prvku modelu, ktorý má typ `Model`. Tento validátor implementuje nasledujúce metódy:

- `validateModelHasTitle(..)` – overí, či model má vyplnený titul (atribút `title`).
- `validateModelHasAuthor(..)` – overí, či model má vyplnené meno autora (atribút `author`).

ArtifactValidator

`ArtifactValidator` slúži na overovanie správnosti prvkov typu `Artifact`. Nakoľko typ `Artifact` je potomkom typu `Entity`, `ArtifactValidator` používa validačné pravidlá definované v `EntityValidator` a definuje vlastné pravidlá:

- `validateArtifactStereotypeNotEmpty(..)` – overí, či artefakt má vyplnený atribút `stereotype`.
- `validateArtifactHasManifest(..)` – zisťuje, či artefakt v modeli obsahuje relácie manifestácie na komponent. V prípade, že nie, vráti správu upozornením.

InterfaceValidator

`InterfaceValidator` zahrnuje validačné pravidlá pre typ `Interface` z ECore modelu. Používa validačné pravidlá `EntityValidator` a implementuje metódu `validateInterfaceRealized(..)`, ktorá overí, či rozhranie je realizované komponentom.

ComponentValidator

`ComponentValidator` definuje validačné pravidlá pre typ `Component`. Nakoľko `Component` je špecializovaným typom `Entity`, `ComponentValidator` používa validačné pravidlá implementované v `EntityValidator`. Vlastné pravidlá implementuje v metódach:

- `validateComponentStereotypeNotEmpty(..)` - overí, či komponent má vyplnený atribút `stereotype`.
- `validateComponentNameFirstLetterIsUpperCase(..)` – overí, či názov komponentu (resp. atribút `name`) začína veľkým písmenom.

SimpleComponentValidator

`SimpleComponentValidator` obsahuje validačné pravidlá pre prvky typu `SimpleComponent`. Tento validátor využíva validačné pravidlá implementované v `ProviderValidator`, `RequierValidator` a `ComponentValidator`. `SimpleComponentValidator` neimplementuje žiadne ďalšie pravidlá.

CompositeComponentValidator

`CompositeComponent` definuje validačné pravidlá pre typ `CompositeComponent`. Využíva validačné pravidlá implementované v `ComponentValidator` a implementuje metódu `validateComponentHasPort(..)`, ktorá overí, či zložený komponent má port.

GeneralizationValidator

`GeneralizationValidator` slúži na overovanie správnosti prvkov typu `Generalization`. Nakoľko typ `Generalization` je potomkom typu `Relationship`, `GeneralizationValidator` používa validačné pravidlá definované v validátore `RelationshipValidator`. Ďalej implementuje metódu pod názvom `validateGeneralizationEnds(..)`, ktorá overí, či oba strany (resp. atribúty `child` a `parent`) odkazujú na nejakú entitu.

RealizationValidator

`RealizationValidator` implementuje validačné pravidlá na overenie správnosti prvku typu `Realization`. `RealizationValidator` používa validačné pravidlá implementované

v `RelationshipValidator`, totiž typ `Realization` je potomkom typu `Relationship`. Ďalšie validačné pravidlá implementuje v metódach:

- `validateRealizationHasSetInterface(..)` – overí, či atribút `interface` prvku typu `Realization` je nastavený, čiže relácia realizácie má určený, ktoré rozhranie bude realizované.
- `validateRealizationHasSetProvider(..)` - overí, či atribút `provider` prvku typu `Realization` je nastavený, čiže relácia realizácie má nastavené, ktorý poskytovateľ bude implementovať zvolené rozhranie.
- `validateRealizationHasProviderAndInterfaceInSameContainer(..)` – overí, či rozhranie a prvok poskytujúci služby sú v rovnakom kontajnery (napr. v rovnakom zloženom komponente).

UsageValidator

`UsageValidator` definuje validačné pravidlá pre prvky typu `Usage`. Tento validátor používa validačné pravidlá `RelationshipValidator` a implementuje ďalšie pravidlá v nasledujúcich metódach:

- `validateUsageHasInterface(..)` - overí, či atribút `interface` prvku typu `Usage` je nastavený, čiže relácia použitia má určený, ktoré rozhranie bude používané požadujúcim prvkom.
- `validateUsageHasSetRequier(..)` - overí, či atribút `requier` prvku typu `Usage` je nastavený, čiže relácia použitia má nastavený prvok, ktorý bude požadovať službu popísanú kontraktom rozhrania.
- `validateUsageHasRequierAndInterfaceInSameContainer(..)` - overí, či rozhranie a prvok požadujúci služby sú v rovnakom kontajnery (napr. v rovnakom zloženom komponente).

DependencyValidator

`DependencyValidator` slúži na overovanie správnosti prvkov typu `Dependency`. Používa validačné pravidlá definované v validátori `RelationshipValidator`. `DependencyValidator` implementuje ďalšie validačné pravidlá v metódach:

- `validateStereotypeIsNotEmpty(..)` – overí, či prvok typu `Dependency` má vyplnený stereotyp (resp. atribút `stereotype`)
- `validateDependencyEnds(..)` – overí, či oba strany (resp. atribúty `dependsFrom` a `dependsTo`) odkazujú na nejakú entitu.

DelegateValidator

`DelegateValidator` definuje validačné pravidlá pre prvky typu `Delegate`. Tento validátor používa validačné pravidlá definované v `RelationshipValidator` a implementuje ďalšie v metódach:

- `validateDelegateHasSetInterface(..)` – overí, či atribút `interface` prvku typu `Delegate` je vyplnený, čiže relácia delegácie odkazuje na rozhranie, na ktoré sa deleguje.
- `validateDelegateHasSetPort(..)` – overí, či atribút `port` prvku typu `Delegate` je vyplnený, čiže relácia delegácie má nastavený port, ktorý deleguje vstup na rozhranie.
- `validateInterfaceAndPortInSameCompositeComponent(..)` – overí, či rozhranie a port, medzi ktorými je relácia delegácie, patria rovnakému zloženému komponentu.

ManifestationValidator

`ManifestationValidator` implementuje validačné pravidlá na overenie správnosti prvku typu `Manifestation`. Počas validácie používa validačné pravidlá `RelationshipValidator` a implementuje metódu `validateManifestationEnds(..)`, ktorá overí, či prvok typu `Manifestation` má vyplnené atribúty `artifact` a `component`.

6.3.3 Registrácia ValidationAdapter do katalógu validátorov

EMF Validation Framework aby mohol používať `ValidationAdapter` na validáciu modelu vytvoreného modelovacím nástrojom, treba ho zaregistrovať do katalógu validátorov (`EValidator.Registry`). Na tento účel slúži trieda `Startup`, ktorá implementuje metódu `earlyStartup(..)` rozhrania `IStartup`. Táto metóda sa volá po inicializácií plochy Eclipse.

Túto triedu treba ešte zaregistrovať na úrovni plug-in. Do súboru **plugin.xml** pridal som rozšírenie `org.eclipse.ui.startup`, kde som určil presné umiestnenie triedy `Startup`.

7 Vývoj konkrétnej syntaxe

Konkrétna syntax definuje textovú alebo grafickú reprezentáciu prvkov definovaných v abstraktnej syntaxi DSL. Nakoľko cieľom tohto projektu bolo vytvoriť modelovací nástroj, ktorý umožní grafický návrh komponent systémov, vývoj konkrétnej syntaxe DSL bol realizovaný pomocou **Graphical Modeling Framework** (viď. kapitolu *4.4 Graphical Modeling Framework*), ktorá umožní vytvoriť grafickú konkrétnu syntax.

V tejto kapitole sa popíšu postupy vývoja modelovacieho nástroja. Ukáže sa spôsob vytvorenia modelu grafických definícií, ktorý obsahuje popis grafickej štruktúry útvarov v modelovacom nástroji. Následne sa definuje obsah palety grafického editoru, ktorá je definovaná v modeli nástrojov.

Grafické útvary sa mapujú na prvky v ECore modeli, ktoré je popísané v modeli mapovania, a vygeneruje sa model GMF generátora. Tento generátor umožní generovať plug-in pre platformu Eclipse, ktorý bude slúžiť ako grafický editor komponent systémov.

V závere kapitoly sa zameriava na vývoj validácie na úrovni konkrétnej syntaxe. Popíše sa jej využitie v grafickom editore a spôsob vývoja.

7.1 Model grafických definícií

Model grafických definícií konkrétnej syntaxe modelovacieho nástroja sa nachádza v súbore **tfmcbs.gmfgraph**. Obsah modelu by sa dal rozdelil na dve komponenty, kde prvý komponent tvorí galéria figúr a druhý obsahuje množinu uzlov, spojení, textových štítkov.

7.1.1 Galéria figúr

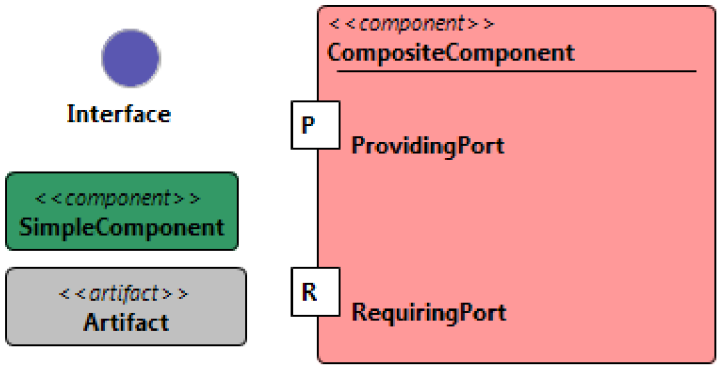
Galéria figúr obsahuje kolekciu popisov figúr (`Figure Descriptor`), ktoré popisujú aké tvary sa budú vyskytovať v grafickej konkrétnej syntaxe modelovacieho nástroja. Model grafických definícií obsahuje dva galérie figúr názvami **Nodes** a **Links**.

7.1.1.1 Galéria figúr „Nodes“

V tejto galérií sú definované tvary reprezentujúce uzly v grafickej konkrétnej syntaxi modelovacieho nástroja:

- **ArtifactFigure** – reprezentuje tvar artefaktu
- **SimpleComponentFigure** - reprezentuje tvar obyčajného komponentu
- **CompositeComponentFigure** - reprezentuje tvar zloženého komponentu
- **InterfaceFigure** - reprezentuje tvar rozhrania
- **InterfaceLabelFigure** - reprezentuje tvar textového štítku s názvom rozhrania

- **ProvidingPortFigure** – reprezentuje sprístupňujúci port
- **RequiringPortFigure** – reprezentuje sprístupňujúci port
- **PortLabelFigure** - reprezentuje tvar textového štítku portu

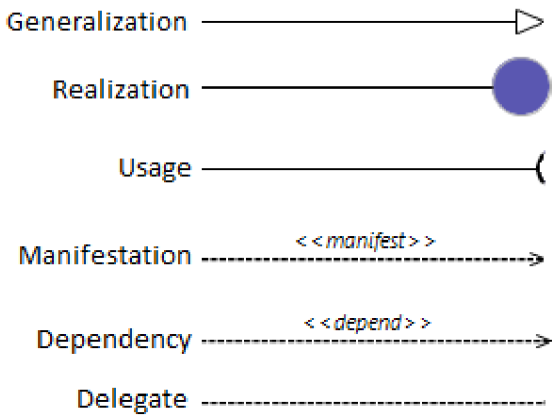


Obr. 31: Galéria figúr „Nodes"

7.1.1.2 Galéria figúr „Links“

Galéria figúr názvom Links obsahuje tvary reprezentujúce spojenia medzi uzlami v grafickej konkrétnej syntaxi modelovacieho nástroja:

- **GeneralizationFigure** - reprezentuje tvar relácie dedičnosti
- **RealizationFigure** - reprezentuje tvar relácie realizácie
- **UsageFigure** - reprezentuje tvar relácie použitia
- **ManifestationFigure** - reprezentuje tvar relácie manifestácie
- **DependencyFigure** - reprezentuje tvar relácie závislosti
- **DelegateFigure** - reprezentuje tvar relácie delegácie



Obr. 32: Galéria figúr „Links"

7.1.2 Uzly, spojenia, kabíny, textové štítky

Modelovací nástroj pre grafický návrh komponent systémov môžeme brať ako 2D modelovací nástroj formou diagramov. Modelovanie pomocou diagramov sa zakladá na teórií grafov, čiže je to zoskupenie uzlov a spojení medzi nimi.
Model grafických definícií, mimo galérií figúr, definuje, aké uzly a spojenia sa v modelovacom nástroji budú vyskytovať. Pre tento modelovací nástroj sú to nasledujúce:

Uzly

- **Artifact** – reprezentuje uzol artefaktu (resp. prvok typu `Artifact`). Tvar uzlu popisuje figúra `ArtifactFigure`.
- **SimpleComponent** - reprezentuje uzol obyčajného komponentu (resp. prvok typu `SimpleComponent`). Tvar uzlu popisuje figúra `SimpleComponentFigure`.
- **CompositeComponent** - reprezentuje uzol zloženého komponentu (resp. prvok typu `CompositeComponent`). Tvar uzlu popisuje figúra `SimpleComponentFigure`.
- **Interface** - reprezentuje uzol rozhrania (resp. prvok typu `Interface`). Tvar uzlu popisuje figúra `InterfaceFigure`.
- **ProvidingPort** - reprezentuje uzol sprístupňujúceho portu (resp. prvok typu `ProvidingPort`). Tvar uzlu popisuje figúra `ProvidingPortFigure`.
- **RequiringPort** - reprezentuje uzol požadujúceho portu (resp. prvok typu `RequiringPort`). Tvar uzlu popisuje figúra `RequiringPortFigure`.

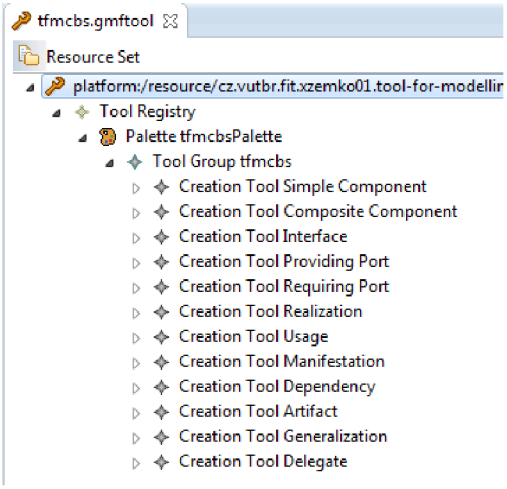
Spojenia

- **Dependency** - reprezentuje reláciu závislosti (resp. prvok typu `Dependency`). Tvar spojenia popisuje figúra `DependencyFigure`.
- **Manifestation** - reprezentuje reláciu manifestácie (resp. prvok typu `Manifestation`). Tvar spojenia popisuje figúra `ManifestationFigure`.
- **Realization** - reprezentuje reláciu realizácie (resp. prvok typu `Realization`). Tvar spojenia popisuje figúra `RealizationFigure`.
- **Usage** - reprezentuje reláciu použitia (resp. prvok typu `Usage`). Tvar spojenia popisuje figúra `UsageFigure`.
- **Generalization** - reprezentuje reláciu dedičnosti (resp. prvok typu `Generalization`). Tvar spojenia popisuje figúra `GeneralizationFigure`.
- **Delegate** - reprezentuje reláciu delegácie (resp. prvok typu `Delegate`). Tvar spojenia popisuje figúra `DelegateFigure`.

Model grafických definícií obsahuje jednu kabínu názvom `CompositeComponentCompartment`. Tento prvok vytvára kontajner pre zloženú komponentu, čiže vo výslednom diagram editore používateľ môže vložiť ďalšie prvky do tohto kontajneru. V modeli grafických definícií sú definované taktiež aj textové štítky, ktoré slúžia na zobrazenie hodnôt atribútov pre konkrétny uzol/spojenie.

7.2 Model nástrojov

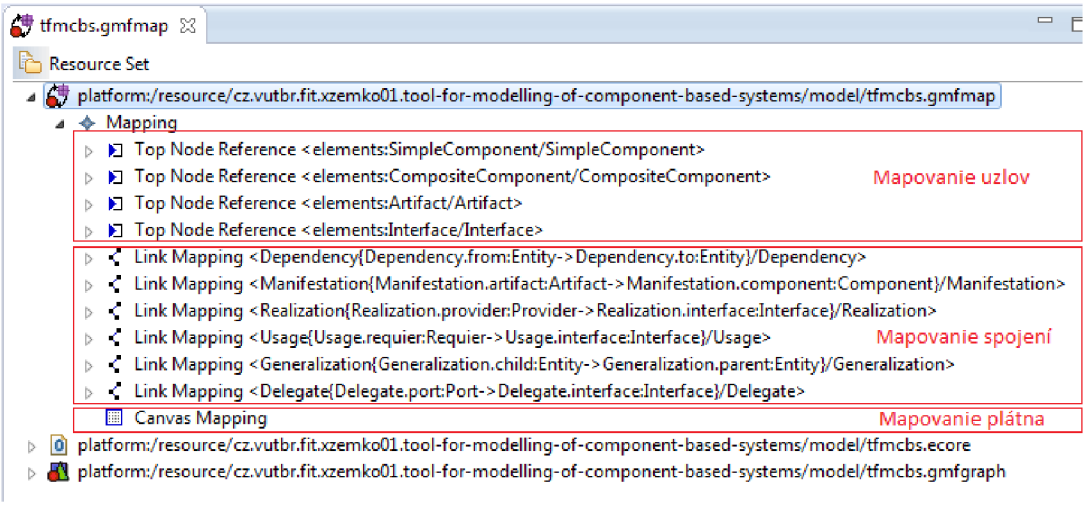
V modeli nástrojov je definovaný obsah palety modelovacieho nástroja pre grafický návrh komponent systémov. Prvky na tejto palete umožnia vytvoriť nové prvky počas modelovania grafickým modelovacím nástrojom. Tento model sa nachádza v súbory **Tfmcbs.gmftool**. Na obrázku Obr. 33 je možné vidieť obsah palety modelovacieho nástroja.



Obr. 33: Model nástrojov

7.3 Model mapovania

V GMF mapujúci model umožní mapovať entity v doménovom modeli (ECore) k zodpovedajúcim figúram, definované v modeli grafických definícií, a k prvkom na palete, definovaných v modeli nástrojov. Tento model sa nachádza v súbory **tfmcbs.gmfmap**.



Obr. 34: Model mapovania

V nasledujúcej tabuľka obsahuje prepojenie EMF ECore meta-modelu, grafických definícií a nástrojmi na ich vytvorenie v diagrame. Z modelu mapovania bol vygenerovaný model GMF generátora, ktorý umožní vygenerovať plug-in diagram editorom pre platformu Eclipse.

Komponent modelu mapovania	Typ z ECore modelu	Prvok z modelu grafických definícií	Prvok z modelu nástrojov
Node Mapping	SimpleComponent	Node SimpleComponent	Creation Tool SimpleComponent
Node Mapping	CompositeComponent	Node CompositeComponent	Creation Tool CompositeComponent
Node Mapping	Artifact	Node Artifact	Creation Tool Artifact
Node Mapping	Interface	Node Interface	Creation Tool Interface
Node Mapping	ProvidingPort	Node ProvidingPort	Creation Tool ProvidingPort
Node Mapping	RequiringPort	Node RequiringPort	Creation Tool RequiringPort
Link Mapping	Dependency	Connection Dependency	Creation Tool Dependency
Link Mapping	Manifestation	Connection Manifestation	Creation Tool Manifestation
Link Mapping	Realization	Connection Realization	Creation Tool Realization
Link Mapping	Usage	Connection Usage	Creation Tool Usage
Link Mapping	Generalization	Connection Generalization	Creation Tool Generalization

Link Mapping	Delegate	Connection Delegate	Creation Tool Delegate

Tab. 25: Mapovanie v modeli mapovania

7.4 Model GMF generátora

Model GMF generátora tvorí hierarchický strom popisujúci štruktúru implementácie diagram editoru. Je vygenerovaný z modelu mapovania. Tento model programátorovi umožní nakonfigurovať podrobnosti generovania kódu výsledného diagram editoru na nízkej úrovni.

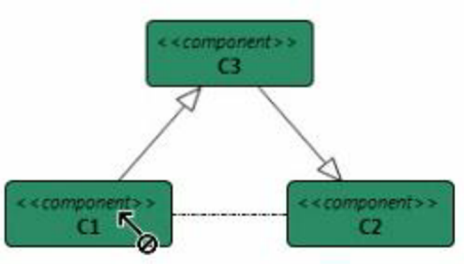
Model GMF generátora modelovacieho nástroja sa nachádza v súbore **tfmcbs.gmfgen**. Pre generovanie zdrojového kódu diagram editoru bolo treba vykonať taktiež určité nastavenia. Vykonané konfigurácie sú popísané v nasledujúcich riadkoch:

1. V prvku **Gen Editor** boli nastavené vlastnosti
   - **Diagram File Extension** na hodnotu **dcbs** (Diagram of Component-Based System), ktorý tvorí príponu diagramu modelovaného komponent systému.
   - **Domain File Extension** na hodnotu **cbs** (Component-Based System), ktorý reprezentuje príponu súboru obsahujúci model komponent systému.
2. V **Gen Diagram ModelEditPart** sú konfigurované vlastnosti:
   - **Validation Enabled** na hodnotu **true**, čím sa povolí validácia modelu v diagram editore.
   - **Validation Decorators** na hodnotu **true**, ktorý umožní GMF zobraziť štandardné značky signalizujúce chyby v diagram editore.
   - **Validation Decorator Provider Priority** na hodnotu **Medium**, čo zabezpečí, že značka signalizujúca chybu nebude zakrytý iným prvkom v diagram editore.
3. V prvku **Gen Compartment CompositeComponentCompositeComponentCompartmentEditPart** je nastavená vlastnosť:
   - **ListLayout** na hodnotu **false**, čo umožní do kontajneru zloženého komponentu vkladať prvky podľa **XY Layout**.

Po vykonaných nastaveniach z modelu GMF generátora sa vygeneroval nový plug-in pre platformu Eclipse, ktorý tvorí grafický diagram editor slúžiaci na modelovanie komponent systému. Zdrojový kód plug-in sa nachádza v projekte názvom **cz.vutbr.fit.xzemko01.tool-for-modelling-of-component-based-systems.diagram**.

7.5 Validácia na úrovni konkrétnej syntaxe

Validácia na úrovni konkrétnej syntaxe sa chová preventívne. Jej úlohou je zachytiť a zabrániť vytvoreniu relácií medzi entitami, ktoré by porušovali konzistentnosť modelu vytvoreného modelovacím nástrojom. V tejto podkapitole sa popíše spôsob implementácie tejto validácie.



Obr. 35: Zabránenie vytvorenia cyklickej dedičnosti

Bol vytvorený nový plug-in **cz.vutbr.fit.xzemko01.tool-for-modelling-of-component-based-systems.diagram.extensions**. V balíku `cz.vutbr.fit.xzemko01.tfmcbs.diagram.extensions.validators` implementuje validátori, ktoré budú overovať správnosť relácie medzi dvoma entitami pred vytvorením relácie v diagram editore. Z tohto dôvodu validátori boli mapované v modeli mapovania na relácie. Výsledok mapovania môžeme vidieť v nasledujúcej tabuľke.

Link Mapping podľa ECore typu	Validátor
Delegate	DelegateLinkTargetEndValidator
Generalization	GeneralizationLinkTargetEndValidator
Realization	RealizationLinkTargetEndValidator
Usage	UsageLinkTargetEndValidator

Tab. 26: Mapovanie validátorov na relácie

Každý z hore uvedených validátorov implementuje statickú verejnú metódu `validate(..)` ktorá vracia hodnotu príznaku (t.j. typu `boolean`). V prípade, že návratová hodnota je `true`, validácia prebehla úspešne a relácia môže byť vytvorená.

DelegateLinkTargetEndValidator

Metóda `validate(..)` sa volá pred vytvorením relácie delegácie (typ `Delegate`) medzi portom a rozhraním. Validátor overí, či port a rozhranie, medzi ktorými sa má vytvoriť relácia delegácie, je v jednom zloženom komponente.

GeneralizationLinkTargetEndValidator

Tento validátor slúži na overenie správnosti relácie dedičnosti medzi dvoma entitami (typ `Entity`). Implementuje nasledujúce podmienky, ktoré vytváraná relácia dedičnosti musí spĺňať:

- **Dedené entity musia mať rovnaký typ** – reláciu dedičnosti je možné vytvoriť jedine medzi dvoma rozhraniami alebo komponentmi. Validáciu overuje súkromná statická metóda `validateGeneralizedEntitiesHasSameType(..)`.
- **Medzi dvoma entitami neexistuje relácia dedičnosti** – medzi dvoma entitami nemôže existovať relácia dedičnosti, ktorá je rovnaká ako relácia dedičnosti, ktorú chceme práve

vytvoriť. Validáciu zabezpečí súkromná statická metóda pod názvom `validateEntityHasUniqueExtends(..)`.

- **Nevytvorí sa cyklická dedičnosť** – sleduje sa, či nová relácia dedičnosti nevytvorí cyklickú dedičnosť v modeli. Validáciu overuje súkromná statická metóda `validateEntityIsNotInCyclicGeneralization(..)`.

RealizationLinkTargetEndValidator

Metóda `validate(..)` sa volá pred vytvorením relácie realizácie (typ `Realization`).
Validátor overí, či:

- **Obidva prvky sú v rovnakom kontajnery** - prvok, poskytujúci služby, a rozhranie, medzi ktorými sa má vytvoriť relácia realizácie, sú umiestnené do rovnakého kontajneru. Metóda názvom `validateProviderAndInterfaceAreInSameContainer(..)` overuje túto vlastnosť.
- **Rozhranie nie je realizované poskytovateľom služieb** – komponent dosial neimplementuje rozhranie. Validáciu vykonáva súkromná statická metóda názvom `validateProviderNotRealizateInterface(..)`.

UsageLinkTargetValidator

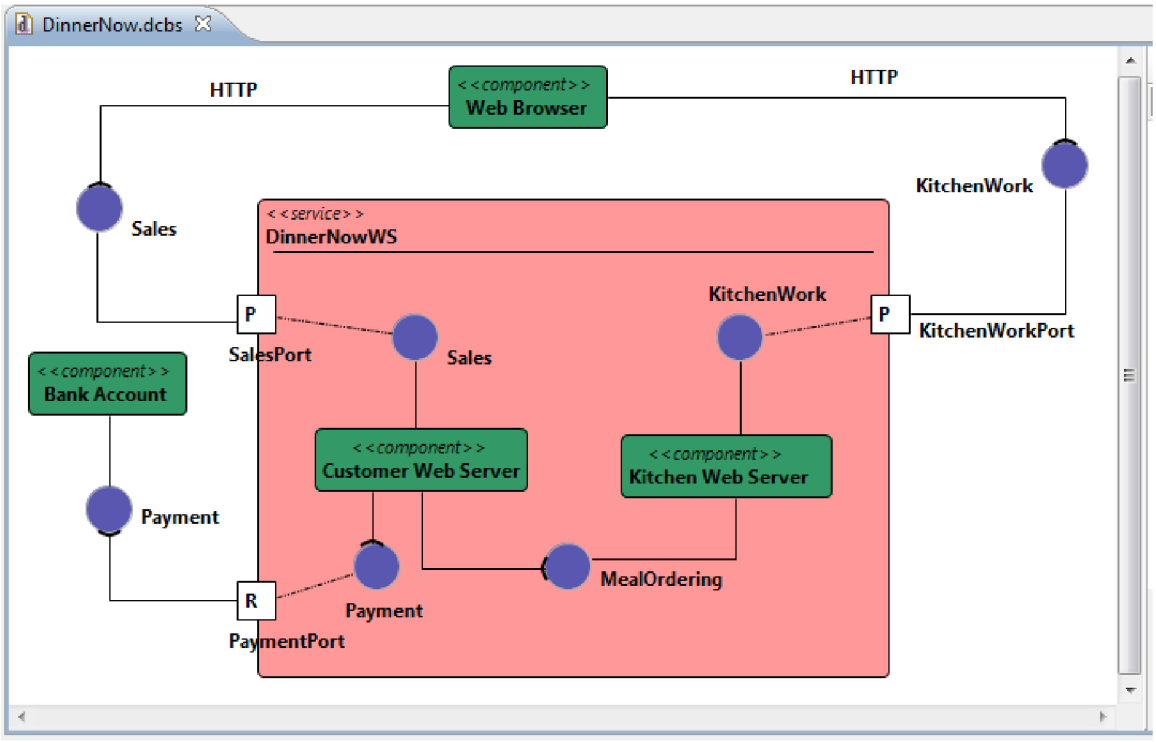
Tento validátor slúži na overenie správnosti relácie použitia (typ `Usage`). Implementuje nasledujúce podmienky, ktoré vytváraná relácia použitia musí spĺňať:

- **Obidva prvky sú v rovnakom kontajnery** - prvok, požadujúci služby, a rozhranie, medzi ktorými sa má vytvoriť relácia použitia, sú umiestnené do rovnakého kontajneru. Metóda názvom `validateRequierAndInterfaceAreInSameContainer(..)` overuje túto vlastnosť.
- **Rozhranie nie je používané prvkom požadujúcim služby -** komponent dosial nepoužíva rozhranie. Validáciu vykonáva súkromná statická metóda názvom `validateRequierNotUseInterface(..)`.

8 Prípad použitia

V tejto kapitole si ukážeme praktický príklad grafického návrhu komponent softvéru vytvoreného modelovacím nástrojom. Budeme modelovať webovú službu fiktívnej spoločnosti názvom **DinnerNow**, ktorá poskytuje služby objednávania večery pre zákazníkov cez internet. Následne budeme hodnotiť vizuálne rozdiely oproti komponent diagramov z jazyka Unified Modeling Language.

Na nasledujúcom obrázku vidíme výsledok návrhu webovej služby spoločnosti **DinnerNow** použitím komponent. Postup vytvorenia tejto webovej služby, použitím modelovacieho nástroja, je popísaný v *Príloha D – Používateľská príručka*. Webovú službu reprezentuje zložený komponent **DinnerNowWS**, ktorý obsahuje dva sprístupňujúce porty, názvom **SalesPort** a **KitchenWorkPort**, a jeden požadujúci port **PaymentPort**. Jej vnútornú štruktúru tvoria komponenty **Customer Web Server** a **Kitchen Web Server**. Komponenty medzi sebou komunikujú cez rozhrania **Sales**, **Payment**, **MealOrdering** a **KitchenWork**.



Obr. 36: Grafický návrh webovej služby DinnerNow

Webová služba **DinnerNowWS** využíva služby komponentu **Bank Account** použitím rozhrania **Payment**. K službám webovej služby pristupujeme cez webový prehliadač (komponent **Web Browser**) cez HTTP protokol využitím rozhraní **Sales** a **KitchenWork**.

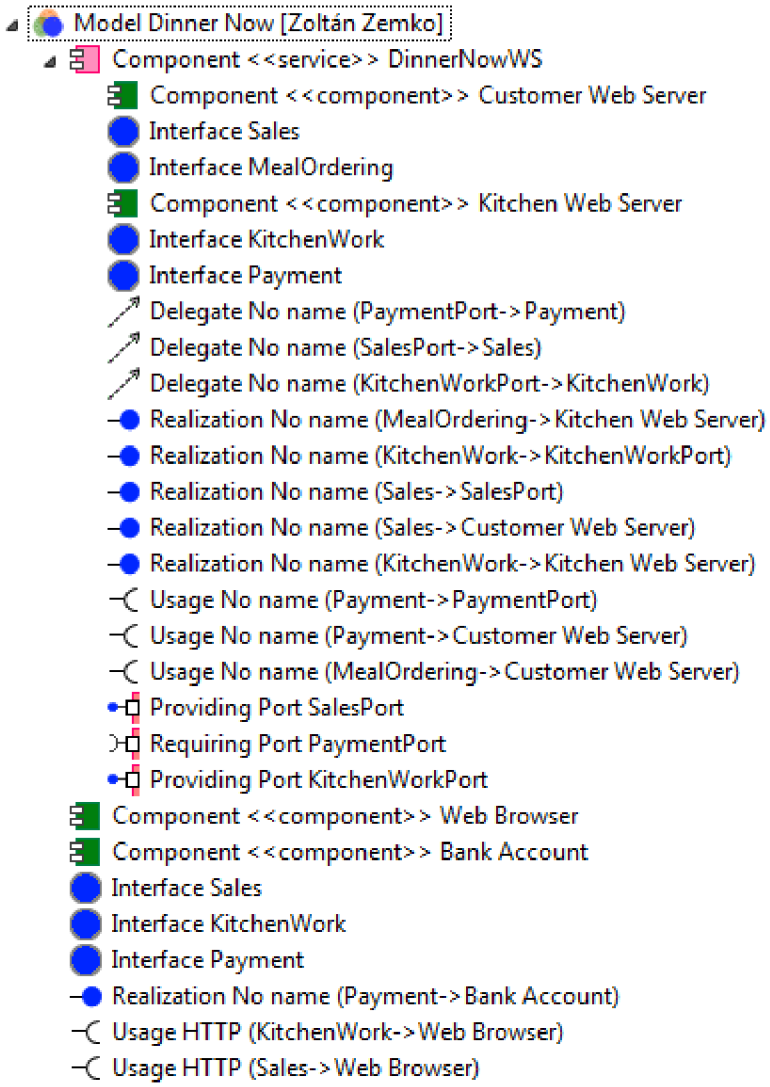
Ako aj na obrázku Obr. 36 vidíme, modelovací nástroj umožní vytvoriť jednoduchý, prehľadný návrh komponentového softvéru. Jej grafická syntax sa zakladá na syntaxe komponent diagramov z jazyka Unified Modeling Language.

V modelovacom nástroji grafická notácia rozhrania a artefaktu je rovnaká ako v UML komponent diagrame. To znamená, že rozhranie je značené kruhom, kým artefakt obdĺžnikom. Modelovací nástroj oproti UML komponent diagramom rozlišuje dva typy komponent: obyčajný komponent a zložený komponent.

Obyčajný komponent je komponent, ktorej vnútorná štruktúra nás nezaujíma, čiže je braná ako „čierna skrinka“ sprístupňujúca/požadujúca nejaké služby od okolia. Je označená obdĺžnikom. Zložený komponent obsahuje kontajner, v ktorom popisuje vnútornú štruktúru komponentu. Služby sprístupňuje/požaduje od okolitého sveta cez porty.

Porty modelovací nástroj, oproti UML komponent diagramov, rozlišuje na sprístupňujúci a požadujúci port. Sprístupňujúci port je značený obdĺžnikom a v jej strede písmenom „P“, ktorý značí, že zložený komponent cez tento port bude poskytovať službu okoliu. Požadujúci port je značený obdĺžnikom a v jej strede písmenom „R“, ktorý značí, že zložený komponent cez tento port bude využívať službu od okolia.

Model, tvorený modelovacím nástrojom, tvorí hierarchický strom zanorenia komponent. To znamená, že napr. komponent z štruktúry zloženého komponentu v tomto strome bude znázornený ako potomok rodiča, kým rodičom bude zložený komponent. Hlavnou výhodou tohto prístupu je, že modelovací nástroj umožní zobrazenie ľubovoľného počtu úrovni zanorenia pri zachovaní a jednoduchosti práce. Na obrázku Obr. 37 vidíme príklad na hierarchické stromové zobrazenie modelu, vytvoreného modelovacím nástrojom z predchádzajúceho príkladu prípadu použitia.



Obr. 37: Model DinnerNow

9 Zhodnotenie a ďalší smer vývoja

V uvedenej kapitole vyhodnotíme výsledky diplomového projektu. Popíšeme do akej miery sa podarilo splniť stanovené ciele v zadaní diplomového projektu. V závere určíme možnosti vývoja modelovacieho nástroja v budúcnosti.

9.1 Zhodnotenie

Cieľom diplomového projektu bolo oboznámiť sa modelovacími technikami komponentových systémov a na základe získaných poznatkov vytvoriť jednoduchý, efektívne použiteľný modelovací nástroj pre grafický návrh komponentových systémov pre platformu Eclipse.

Na začiatku projektu som sa oboznámil pojmom komponentového softvéru. V rámci neho som skúmal význam komponentu z pohľadu znovu použiteľnosti, abstrakcie a nasadenia. Ďalej som sa oboznámil výhodami a nevýhodami vývoja softvéru vlastnej výroby a štandardného softvéru. Následne som poukázal na výhody vývoja softvéru ako komponentový softvér.

Oboznámil som sa syntaxou komponent diagramu, ktorý je jedným najpoužívanejším prostriedkom na modelovanie komponent softvéru. V dokumente popisujem pojmy rozhranie, komponent a aké väzby môžu byť vytvorené medzi nimi.

Vzhľadom na to, že grafický modelovací nástroj pre návrh komponentových systémov mal byť cielený na platformu Eclipse, oboznámil som sa štruktúrou Eclipse Modeling Project a princípmi vývoja doménovo-špecifických jazykov v rámci neho. Podrobnejšie som sa zaoberal vývojom abstraktnej syntaxe použitím Eclipse Modeling Framework, ktorý slúži na definovanie štruktúry doméno-špecifického jazyka (resp. abstraktnej syntaxe DSL). Následne som študoval životný cyklus vývoja grafickej konkrétnej syntaxe doménovo-špecifického jazyka využitím Graphical Modeling Framework.

Po nazbieraní dostatok teoretických znalostí som modelovací nástroj rozhodol vyvíjať ako doménovo-špecifický jazyk. Po konzultácií vedúcim projektu, som na základe požiadaviek vytvoril návrh štruktúry modelovacieho nástroja v UML, použitím modelu tried. Tento model slúžil ako základ pri implementácií abstraktnej syntaxe doménovo-špecifického jazyka modelovacieho nástroja.

V rámci Eclipse Modeling Framework som, podľa návrhu, implementoval meta-model modelovacieho nástroja použitím EMF ECore modelom. Sémantiku tohto meta-modelu som rozšíril o validačný mechanizmus, ktorý dokáže zachytiť nekonzistentné údaje v modeli vytvoreného modelovacím nástrojom.

Grafickú konkrétnu syntax som implementoval využitím Graphical Modeling Framework. Nadefinoval som tvary uzlov a spojení v diagram editore a mapoval som ich na prvkov vytvorených v abstraktnej syntaxe DSL modelovacieho nástroja.

Na úrovni konkrétnej syntaxe taktiež bola vytvorená validácia. Jej úlohou je zabrániť vytvoreniu nekonzistentných prvkov v modeli.

V hore uvedeného postupu riešenia projektu vyplýva, že všetky body zadania diplomového projektu som **úspešne** vyriešil. V závere tejto podkapitoly budem hodnotiť modelovací nástroj ako výsledný produkt projektu.

Podarilo sa mi vytvoriť grafický modelovací nástroj pre návrh komponent systémov, ktorej grafická syntax sa zakladá na syntaxe komponent diagramu z UML, ale miestami sa od nej odlišuje. Grafické notácie, v ktorých sa modelovací nástroj odlišuje od komponent diagramu, slúžia na zvyšovanie prehľadnosti grafického návrhu tvoreného modelovacím nástrojom. O podrobnom porovnávaní modelovacieho nástroja a komponent diagramu, sa môžete dočítať v kapitole 8 *Prípad použitia*.

Modelovací nástroj obsahuje komplexný validačný mechanizmus. Oproti existujúcim riešeniam, vyniká svojou inteligenciou nielenže zachytiť chyby v modeli, vytvoreného modelovacím nástrojom, ale vo väčšine prípadov dokáže aj zabrániť ich vzniku.

Grafický modelovací nástroj pre návrh komponent systémov bol vytvorený ako doménovo-špecifický jazyk. Jej funkcionalita sa môže ľahko rozšíriť definovaním nového doménovo-špecifického jazyka. Viac si o tom povieme v nasledujúca podkapitole.

9.2 Ďalší smer vývoja systému

Modelovací nástroj v súčasnom stave umožní používateľovi vytvoriť model komponent systému a modelovať ho pomocou grafického diagram editoru. Nepodporuje však transformácie modelu na iný dokument.

V budúcnosti by bolo výhodné rozšíriť modelovací nástroj o generovanie dokumentácie z modelu (napr. HTML dokumentácia). Bolo by taktiež užitočné, aby modelovací nástroj podporoval import/export modelu na iné formáty iných modelovacích nástrojov. Tieto transformácie môžeme implementovať vytvorením nového doménovo-špecifického jazyka zakladajúceho na technológiách podporujúcich transformácie model na model a model na text nad platformou Eclipse.

10 Záver

Diplomový projekt sa zameriaval na vývoj modelovacieho nástroja, ktorý umožní používateľovi vytvoriť grafický návrh komponentových systémov. Cieľom bolo oboznámiť sa vývojom komponent softvéru a podľa získaných poznatkov navrhnúť a implementovať modelovací nástroj, ktorý bude nasadený na platformu Eclipse.

V teoretickej časti projektu som sa oboznámil pojmom komponentového softvéru a skúmal som výhody/nevýhody vytvorenia softvéru týmto prístupom. Ďalej som sa oboznámil grafickou syntaxou komponent diagramu, ktorý je jedným najpoužívanejším prostriedkom na modelovanie komponent softvéru.

Modelovací nástroj bol implementovaný ako doménovo-špecifický jazyk využitím Eclipse Modeling Framework a Graphical Modeling Framework. Podarilo sa vytvoriť modelovací nástroj jednoduchou, prehľadnou grafickou syntaxou a komplexným validačným mechanizmom, ktorý robí tento modelovací nástroj jedinečným.

11 Použitá literatúra

[1] SZYPERSKI, Clemens, Dominik GRUNTZ a Stephan MURER. *Component Software: Beyond Object-Oriented Programming*. 2nd ed. London: Addison-Wesley, 2002, xxxii, 589 s. ISBN 02-017-4572-0.

[2] ARLOW, Jim a Ila NEUSTADT. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. 2nd ed. Boston: Addison-Wesley, 2005, 592 s. ISBN 03-213-2127-8.

[3] GRONBACK, Richard C. *Eclipse Modeling Project: A Domain-Specific Language Toolkit*. Upper Saddle River,: Addison-Wesley, 2009, xxv, 706 s. ISBN 978-0-321-53407-1.

[4] AMBLER, Scott W. *The Object Primer: Agile Modeling-Driven Development with UML 2.0*. 3rd ed. New York: Cambridge University Press, 2004, xxvi, 545 p. ISBN 05-215-4018-6.

[5] STEINBERG, Dave. *EMF: Eclipse Modeling Framework*. 2nd ed. Upper Saddle River: Addison-Wesley, 2009, xxix, 704 s,. ISBN 978-0-321-33188-5.

[6] MOORE, Bill. *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. 1st ed. Research Triangle Park, NC: IBM, International Technical Support Organization, 2004, xii, 238 p. ISBN 07-384-5316-1. Dostupné z: http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf

[7] GÁL, Ivan. *Modelovací nástroj pro grafický návrh komponentových systémů: A Tool for Modelling of Component-Based Systems*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2009. 1 elektronický optický disk [CD-ROM / DVD]. Diplomová práca.

[8] BELL, Donald. *UML basics: Component diagrams.* [online]. 2004 [cit. 2012-11-15]. Dostupné z: http://www.ibm.com/developerworks/rational/library/dec04/bell/

[9] AMBLER, Scott. *Agile Modeling: UML 2 Component Diagrams*. [online]. 2009 [cit. 2012-11-17]. Dostupné z: http://www.agilemodeling.com/artifacts/componentDiagram.htm

[10] *Component-based software engineering.* [online]. 2013 [cit. 2013-10-12]. Dostupné z: http://en.wikipedia.org/wiki/Component-based_software_engineering

[11] GRIFFIN, Catherine. *Using EMF*. [online]. 2002 [cit. 2013-01-05]. Dostupné z: http://www.eclipse.org/articles/Article-Using EMF/using-emf.html

[12] VOGEL, Lars. *Eclipse Modeling Framework - Tutorial*. [online]. 2012 [cit. 2013-01-06]. Dostupné z: http://www.vogella.com/articles/EclipseEMF/article.html

[13] *Tutorial: EMF Validation Adapter*. IBM CORPORATION. [online]. 2005 [cit. 2013-03-28]. Dostupné z: http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.emf.validation.doc/tutorials/validationAdapterTutorial.html

[14] *GMF Samples and Tutorials*. [online]. 2011 [cit. 2013-01-10]. Dostupné z: http://gmfsamples.tuxfamily.org/wiki/doku.php

[15] *Graphical Modeling Framework: Get started*. ECLIPSE FOUNDATION. [online]. [cit. 2013-01-10]. Dostupné z: http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial#All

[16] ANISZCZYK, Chis. *Learn Eclipse GMF in 15 minutes*. [online]. 2006 [cit. 2013-01-20]. Dostupné z: http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/

[17] ELOUMRI, Miloud. *Model-Driven Software Development: Graphical Modeling Framework*. [online]. 2010 [cit. 2013-02-17]. Dostupné z: http://www.miloud.webs.com/gmf.htm

[18] ELOUMRI, Miloud. *Model-Driven Software Development: Eclipse Modeling Framework*. [online]. 2010 [cit. 2013-02-05]. Dostupné z: http://www.miloud.webs.com/emf.htm

Príloha A – Obsah CD

Súčasťou diplomovej práci je CD, ktoré obsahuje:

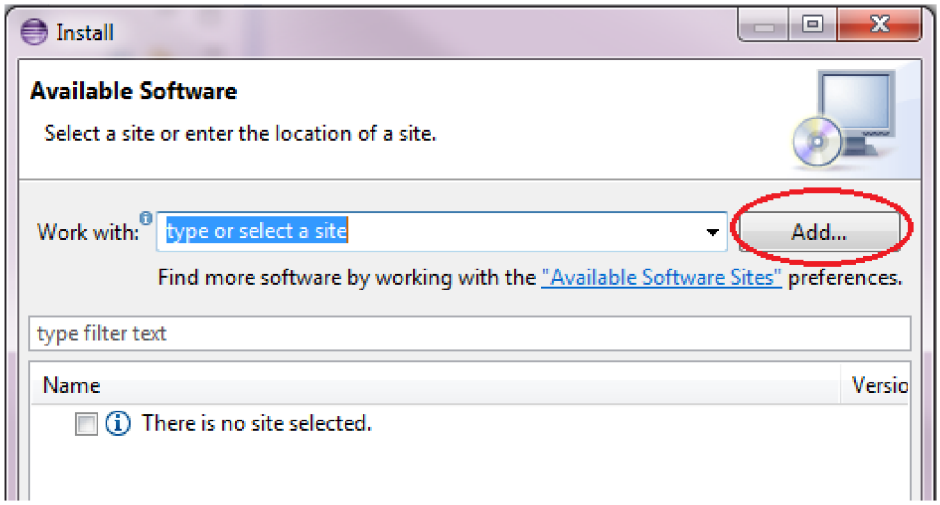
- **Plug-in do Eclipse Modelovacieho nástroja**
  Disk CD:\Tool for Modelling Component Based Systems\Binary
- **Zdrojový kód modelovacieho nástroja**
  Disk CD:\Tool for Modelling Component Based Systems\Source
- **Diplomovú prácu v elektronickej podobe**
  Disk CD:\diplomova_praca.pdf

Príloha B – Návod na inštaláciu

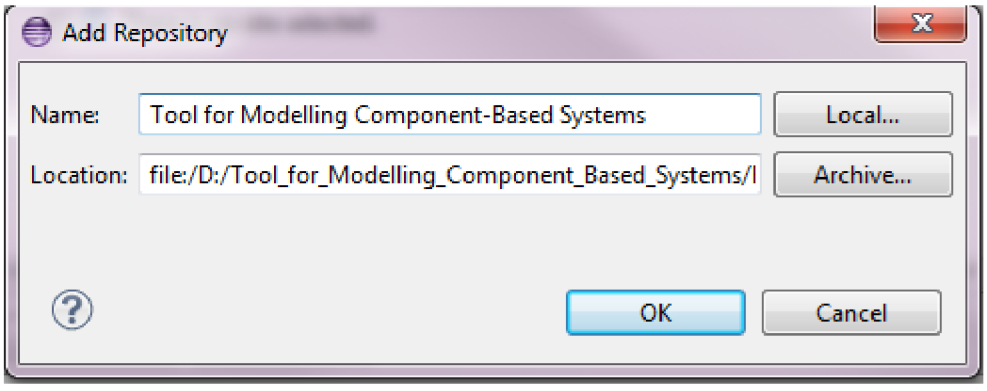
Pred inštalovaním modelovacieho nástroja pre grafický návrh komponentových systémov je potrebné nainštalovať softvér:

- **Java SE Development Kit 7**
  URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- **Eclipse Juno (4.2)**
  URL: http://www.eclipse.org/downloads/

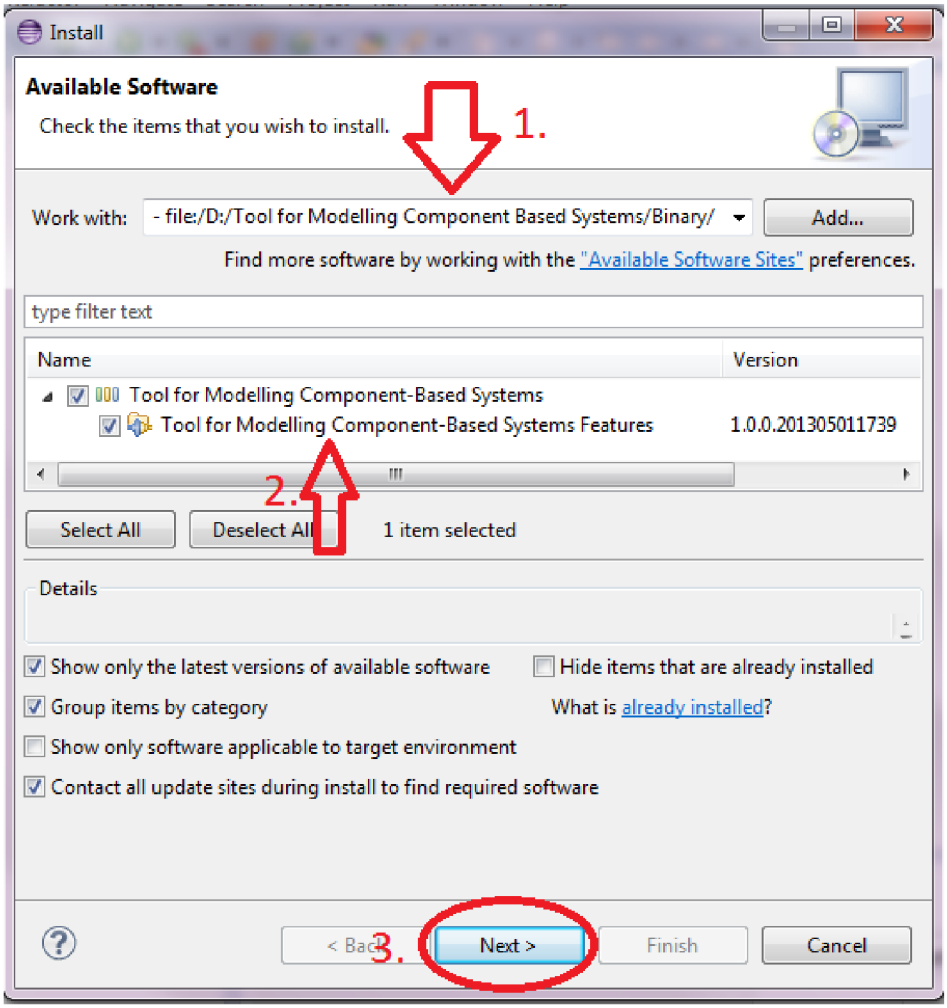
Spustíme vývojové prostredie Eclipse Juno. Z menu vyberieme **Help → Install New Software...** . Otvorí sa dialóg nadpisom **Avaible Software**. Klikneme na tlačítko **Add**.



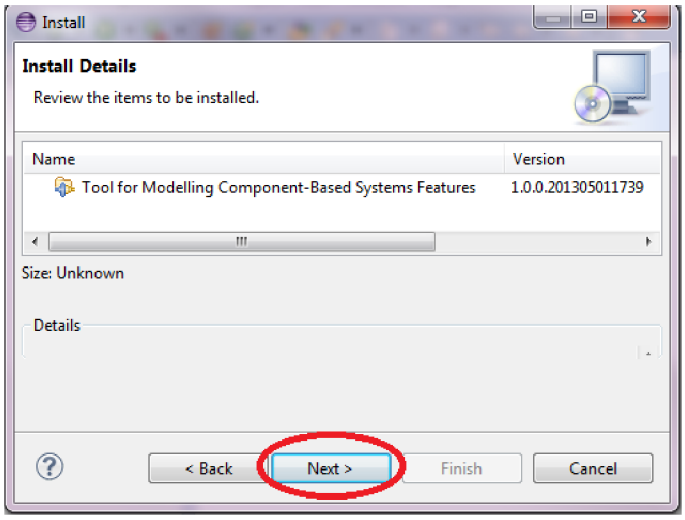
Otvorí sa dialóg nadpisom **Add Repository**. Políčko **name** vyplníme textom **Tool for Component-Based Systems** a klikkneme na tlačítko **Local**. V novom dialógu vyberieme koreňový adresár, v ktorom sa umiestňuje plug-in modelovacieho nástroja. Na CD, odovzdaného záverečnou správou diplomového projektu, sa nachádza v adresári pod uvedenou cestou **Disk:\Tool for Modelling Component Based Systems\Binary**. Dialóg Add Repository potvrdíme tlačítkom **OK**.



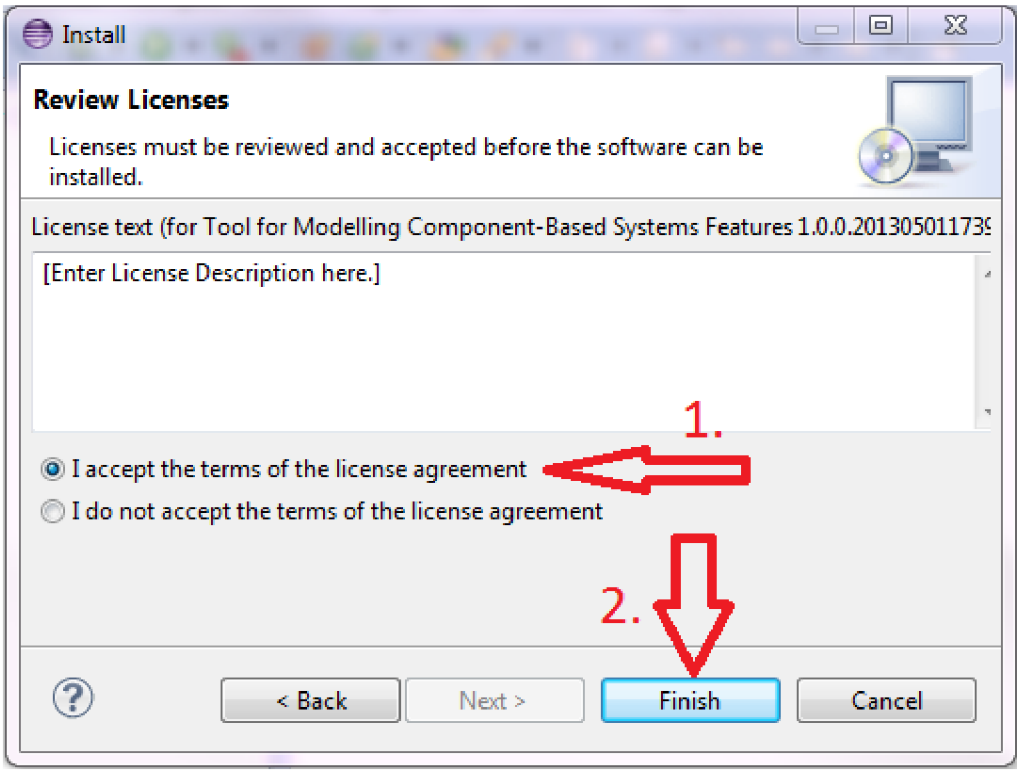
Dostávame sa opäť do dialógu nadpisom **Avaible Software**. W políčku **Work with…** vyberieme novo pridaný repozitár názvom **Tool for Component-Based Systems**. V kategórii **Tool for Component-Based Systems** označíme položku **Tool for Component-Based Systems Features** a stlačíme tlačítko **Next**.



Otvorí sa okno nadpisom **Install Details**. Pre pokračovanie stlačíme tlačítko **Next**. Otvorí sa nový dialóg oknom **Review Licenses**.

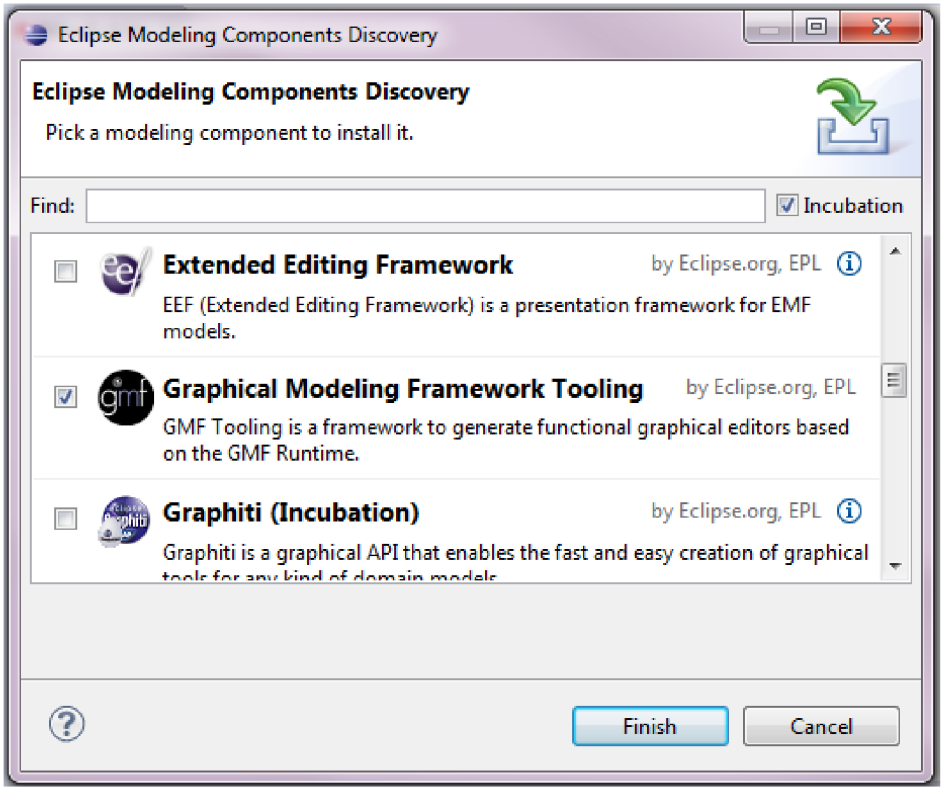


V dialógu označíme **I accept the terms of the license agreement** a stlačíme tlačítko **Finish**. Inštalátor nainštaluje plug-in modelovacím nástrojom do Eclipse. Po reštartovaní Eclipse modelovací nástroj je pripravený na použitie.



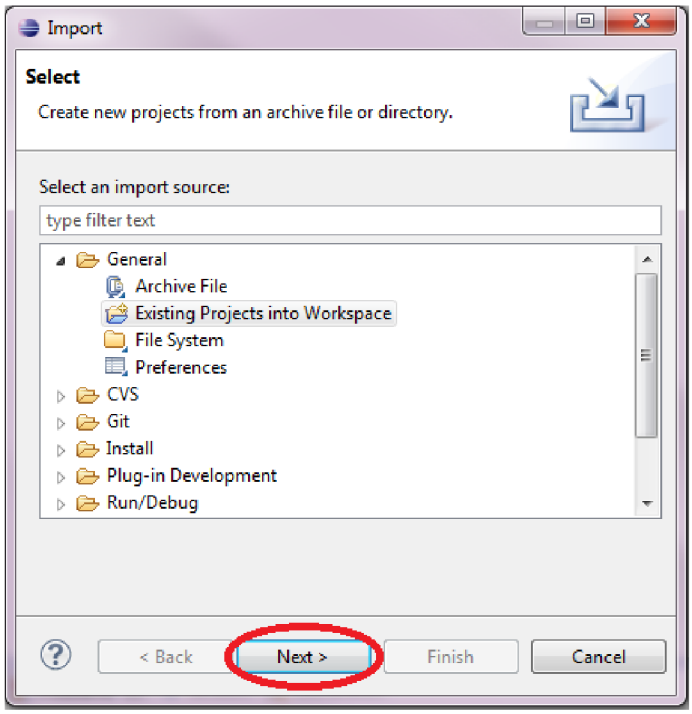
Príloha C – Import zdrojového kódu modelovacieho nástroja

Na vývoj modelovacieho nástroja budeme používať **Eclipse IDE Modeling Tools** (http://www.eclipse.org/downloads/). Spustíme vývojové prostredie. Zvolením **Help → Install Modeling Components** si vyberieme **Graphical Modeling Framework Tools**.



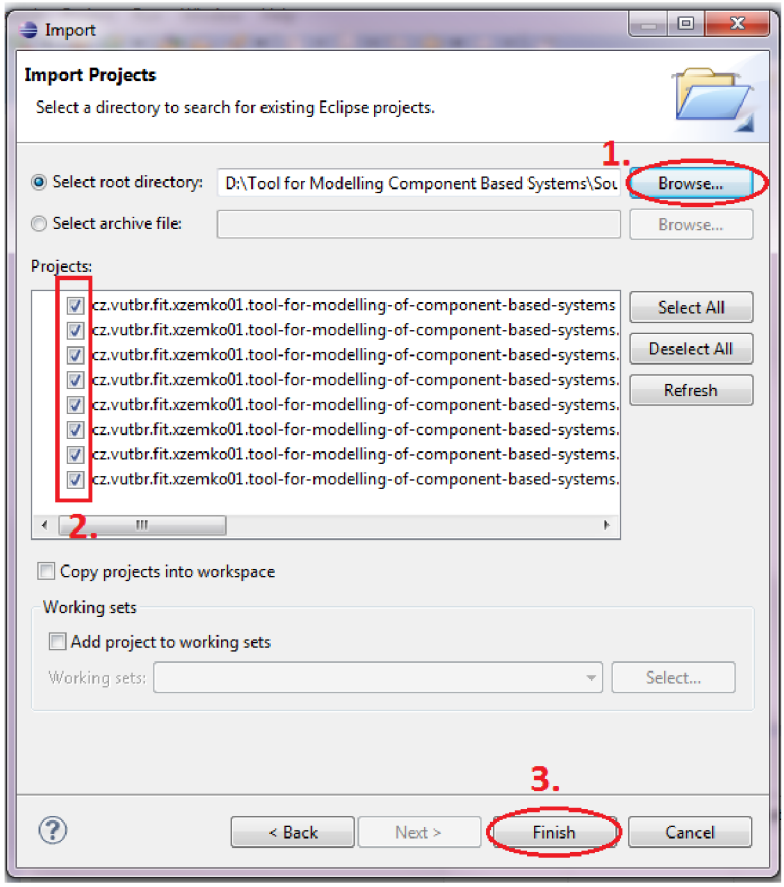
Na CD, odovzdaného záverečnou správou diplomového projektu, sa nachádza v adresári **Disk:\Tool for Modelling Component Based Systems\Source**. Obsah tohto adresára prekopírujeme na HDD. Napr. do adresára **C:\Tool for Modelling Component Based Systems\Source**.

Zdrojový kód modelovacieho nástroja do Eclipse importujeme zvolením **File → Import**. Otvorí sa dialóg importu nadpisom **Select**. Vyberieme položku **General\Existing Project Into Workspace** a stlačíme tlačítko **Next**.



Z dialógu nadpisom **Import Projects** stlačítko **Browse**. Otvorí sa dialóg nadpisom **Browse For Folder**. Zvolíme adresár **C:\Tool for Modelling Component Based Systems\Source**, v ktorom sú súbory zdrojovým kódom a potvrdíme výber tlačítkom OK.

Opäť sa vraciame do dialógu nadpisom **Import Projects**. V zozname **Projects** vyznačíme všetky načítané projekty a stlačíme tlačítko **Finish** a Eclipse načíta projekty do pracovnej plochy.



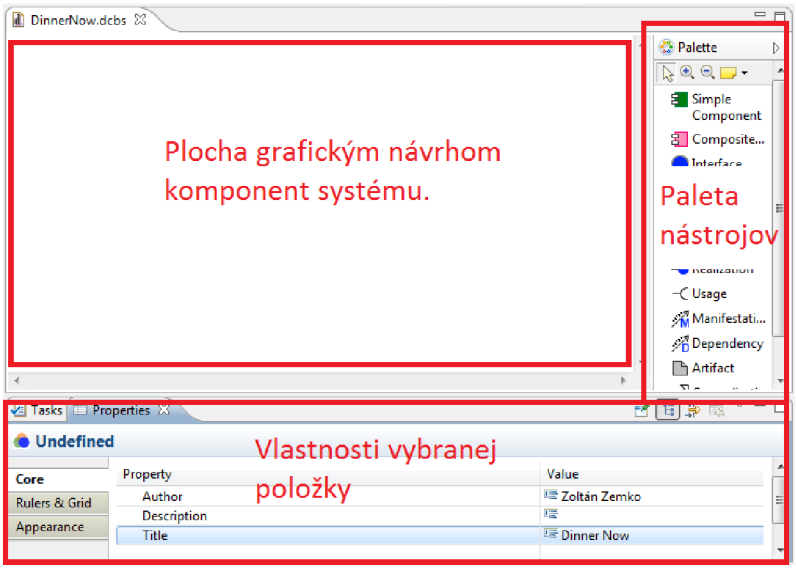
Príloha D – Používateľská príručka

V prílohe si ukážeme praktický príklad na použitie modelovacieho nástroja. Vytvoríme grafický návrh webovej služby fiktívnej spoločnosti názvom **DinnerNow**, ktorá poskytuje služby objednávania večery pre zákazníkov cez internet.

V prvom kroku si musíme vytvoriť model komponentového softvéru a diagram podporujúci jej grafický návrh. Túto činnosť vykonáme v nasledujúcich krokoch:

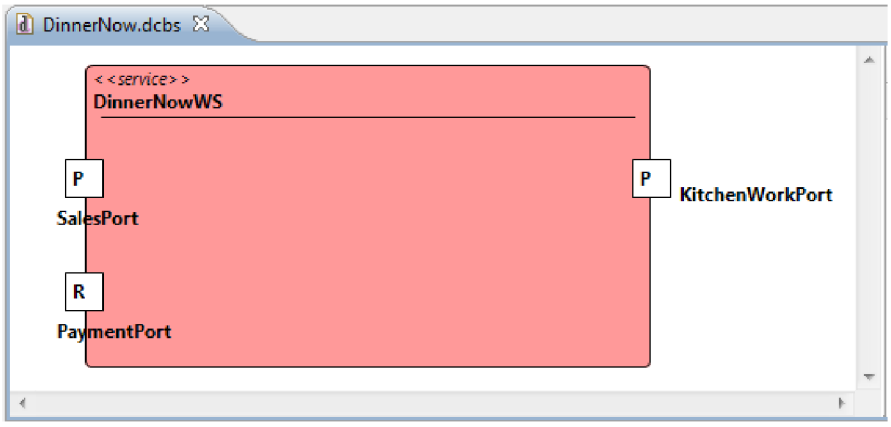
1. Vytvoríme nový projekt použitím menu **File → New → Project**, ktorý nazveme **DinnerNow**.
2. Vytvoríme model popisujúci komponentového softvér a jej diagram podporujúci grafický návrh. Použitím menu **File → New → Other...** otvoríme dialóg nadpisom **Select a Wizard** a vyberieme si položku **Component Based System Diagram** z kategórie **Tool for Modelling Component-Based Systems** a stlačíme tlačítko **Next**.
3. Otvorí sa dialóg nadpisom **Create Component Based System Diagram**. Do políčka **File name** zadáme text **DinnerNow.dcbs**, ktorý bude tvoriť názov súboru popisujúci diagram modelu komponent systému. Pre umiestnenie súboru, vyberieme projekt **DinnerNow**. Opäť stlačíme tlačítko **Next**.
4. Otvorí sa dialóg nadpisom **Create Component Based System Domain Model**. Do políčka **File name** zadáme text **DinnerNow.cbs**, ktorý bude tvoriť názov súboru obsahujúci model komponent systému. Pre umiestnenie súboru, vyberieme projekt **DinnerNow**. Stlačíme tlačítko **Finish**.

Eclipse vytvorí súbory **DinnerNow.dcbs** a **DinnerNow.cbs** pod projektom **DinnerNow**. Otvorí sa diagram editor, slúžiaci na grafický návrh komponent systémov. Popis jej používateľského rozhrania je na nasledujúcom obrázku.



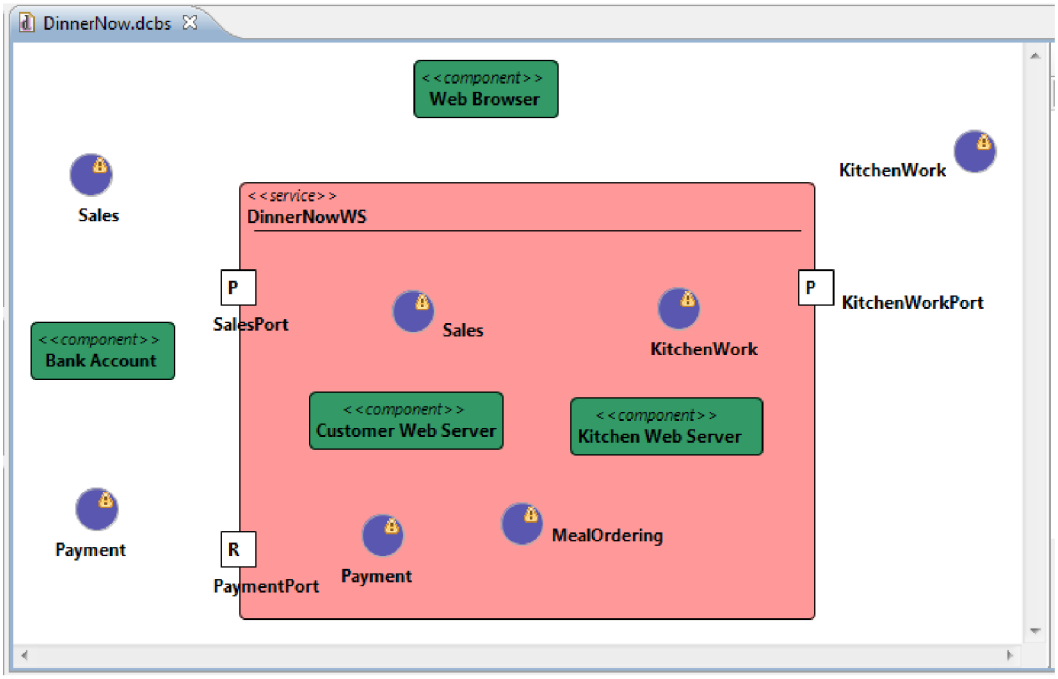
V diagram editore vytvoríme zložený komponent, ktorého nazveme **DinnerNowWS** a nastavíme jeho stereotyp na hodnotu **service**. Zložený komponent vytvoríme pomocou položky **Composite Component** z palety modelovacieho nástroja.

Zložený komponent **DinnerNowWS** bude obsahovať tri porty. Vytvoríme dva sprístupňujúce porty (`Providing Port`), pod názvom **SalesPort** a **KitchenWorkPort**, a jeden požadujúci port (`Requiring Port`) nazývaný **PaymentPort**.



V kontajneri zloženého komponentu **DinnerNowWS**, vytvoríme obyčajné komponenty (`Simple Component`) názvami **Customer Web Server** a **Kitchen Web Server**. Ďalej pridáme rozhrani (`Interface`) názvami **Sales**, **Payment**, **MealOrdering** a **KitchenWork**.

V okolí zloženého komponentu **DinnerNowWS** vytvoríme obyčajné komponenty názvami **Web Browser** a **Bank Account**. Pridáme rozhrania nazývané **Sales**, **Payment** a **KitchenWork**.

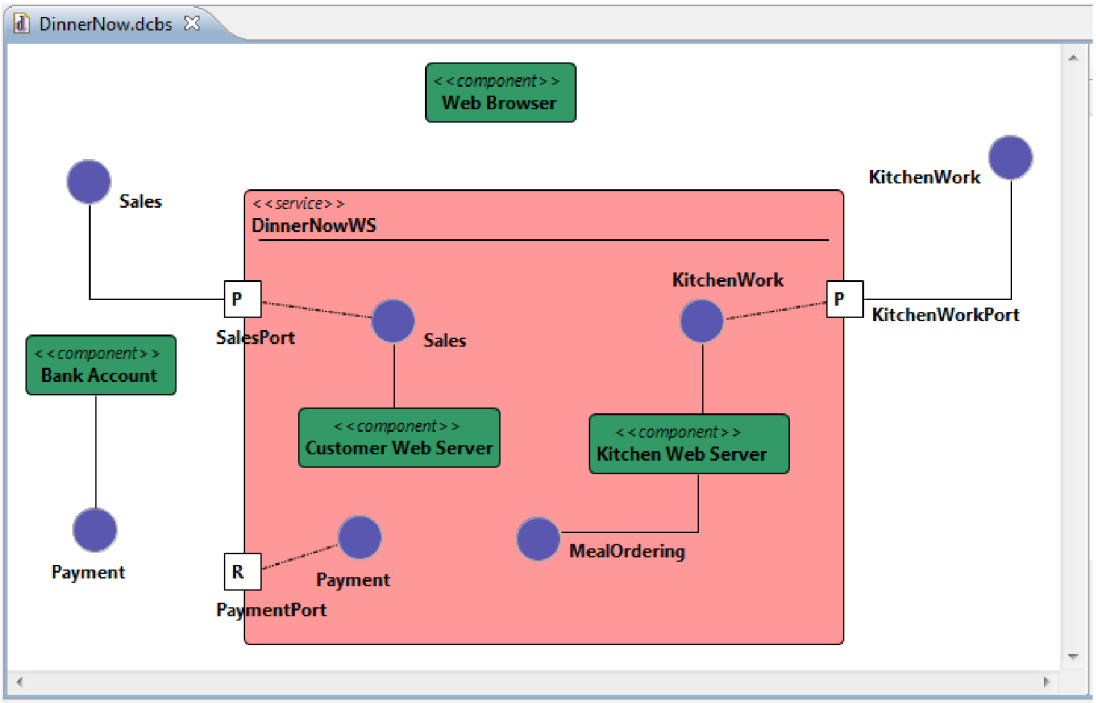


Po uložení modelu, pri každom rozhraní vidíme žltý trojuholník výkričníkom. Validačný mechanizmus upozorňuje používateľa, že rozhrania nie sú nikým implementované. Tieto upozornenia budú odstránené pri tvorbe relácií realizácií. Pred tým vytvoríme relácie delegácií medzi:

- Sprístupňujúci port **SalesPort** → rozhranie **Sales** umiestené v zloženom komponente **DinnerNowWS**.
- Požadujúci port **PaymentPort** → rozhranie **Payment** umiestené v zloženom komponente **DinnerNowWS**.
- Sprístupňujúci port **KitchenWorkPort** → rozhranie **KitchenWork** umiestené v zloženom komponente **DinnerNowWS**.

Po vytvorení relácií delegácie medzi portami a rozhraniami, vytvoríme relácie realizácií:

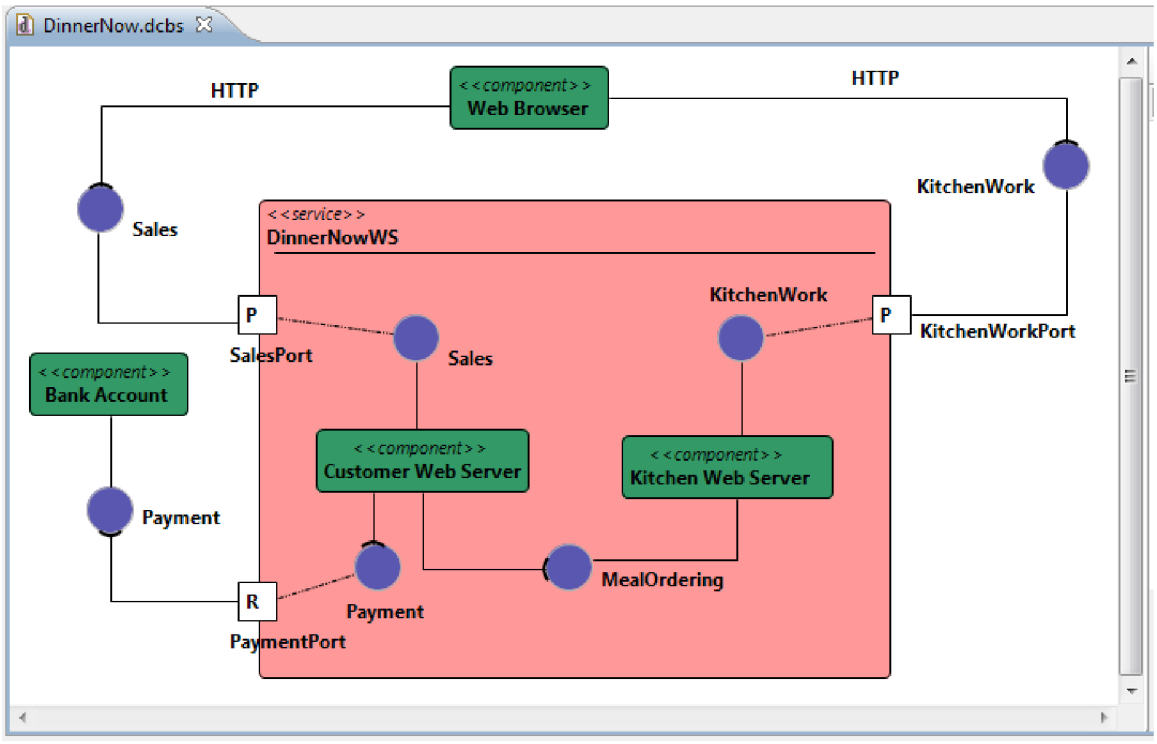
- Obyčajný komponent **Customer Web Server** → rozhranie **Sales** umiestené v zloženom komponente **DinnerNowWS**.
- Obyčajný komponent **Kitchen Web Server** → rozhranie **KitchenWork** umiestené v zloženom komponente **DinnerNowWS**.
- Obyčajný komponent **Kitchen Web Server** → rozhranie **MealOrdering**
- Sprístupňujúci port **SalesPort** → rozhranie **Sales**
- Sprístupňujúci port **KitchenWorkPort** → rozhranie **KitchenWork**



Na záver vytvoríme relácie použitia medzi:

- Obyčajný komponent **Customer Web Server** → rozhranie **Payment** umiestené v zloženom komponente **DinnerNowWS**.
- Obyčajný komponent **Customer Web Server** → rozhranie **MealOrdering**
- Požadujúci port **PaymentPort** → rozhranie **Payment**
- Obyčajný komponent **Web Browser** → rozhranie **Sales**. Atribút **name** tejto relácie vyplníme hodnotou **HTTP**.
- Obyčajný komponent **Web Browser** → rozhranie **KitchenWork**. Atribút **name** tejto relácie vyplníme hodnotou **HTTP**.

Na nasledujúcom obrázku vidíme výsledok návrhu webovej služby spoločnosti **DinnerNow** použitím komponent.


DinnerNow.dcbs
HTTP
<<component>>
Web Browser
HTTP
KitchenWork
Sales
<<service>>
DinnerNowWS
KitchenWork
P
P
KitchenWorkPort
SalesPort
Sales
<<component>>
Bank Account
<<component>>
Customer Web Server
<<component>>
Kitchen Web Server
Payment
MealOrdering
R
Payment
PaymentPort