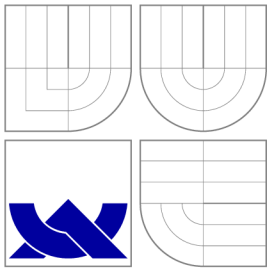
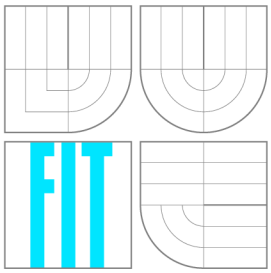


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

UML PROFIL PRO MODELOVÁNÍ KOMPONENTOVÝCH SYSTÉMŮ

AN UML PROFILE FOR MODELLING OF COMPONENT-BASED SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PAGÁČ

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. MAREK RYCHLÝ, Ph.D.

BRNO 2011

Abstrakt

Práce se zabývá modelováním softwarových komponentových systémů v jazyce UML a rozšíření jazyka UML technikami meta-modelování s využitím technologie UML Profile za účelem vytvoření podpory pro vybraný komponentový systém. Dále se věnuje komponentově orientovanému vývoji. Jádro práce tvoří vytvoření metodologie pro tvorbu UML Profilů a demonstrace tohoto postupu na vybraném komponentovém systému včetně OCL omezení a popisu problémů při vytváření profilu. V práci je také popsána podpora této technologie v dostupných UML CASE nástrojích. Pro účely demonstrace práce obsahuje případovou studii obsahující příklad, který využívá vytvořený UML Profil. Profil je vytvořen v souladu se specifikací UML v2.3, OCL v2.2. Profil a demonstrace samotná využívá software IBM Rational Software Architect ve verzi 8.0.2.

Abstract

The thesis deals with the modeling of the Component Based Software (CBS) systems in the UML language and with extension of the UML language with using of meta-modeling techniques and with using of the UML Prodile technology. Thesis also deals with Component Based Development (CBD). The main part of this study deals with specifying of methodology for creating of UML Prodiles and with demonstration of this methodology on selected Component System meta-model by creating the profile including the OCL constraints and description of problems with creating of the prodile. Thesis also describes support of the UML Profile technology in existing UML CASE tools. For demonstration purposes thesis contains the case study with example which uses in this work created UML Profile. Profile is created in accordance with specification of UML version 2.3 and OCL in version 2.2. Demonstration is performed and profile itself is created using IBM Rational Software Architect version 8.0.2.

Klíčová slova

UML, UML Profil, komponentové diagramy, komponentové architektury, komponentové systémy, vývoj založený na komponentách, CBD, OCL, metamodel, meta-modelování, Rational Software Architect

Keywords

UML, UML Profile, component diagrams, component based architectures, component based systems, component based development, CBD, OCL, metamodel, meta-modeling, Rational Software Architect

Citace

Jiří Pagáč: UML profil pro modelování komponentových systémů, diplomová práce, Brno, FIT VUT v Brně, 2011

UML profil pro modelování komponentových systémů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Mgr.Marka Rychlého PhD.

.....
Jiří Pagáč
23. května 2011

Poděkování

Rád bych poděkoval vedoucímu práce Mgr. Marku Rychlému PhD. za poskytnutou odbornou pomoc během tvorby mé práce.

Obsah

1 Úvod **3**
- 1.1 Cíl práce . . . 4
- 1.2 Struktura práce . . . 4

2 Modelování SW systémů **5**
- 2.1 Jazyk UML . . . 5
- 2.2 Architektura UML . . . 7
- 2.3 Meta-modelování . . . 8
  - 2.3.1 MOF a EMOF . . . 9
- 2.4 Možnosti rozšíření UML . . . 10
  - 2.4.1 Vytvořením nebo změnou meta-modelu . . . 11
  - 2.4.2 Rozšíření použitím UML Profile . . . 11
- 2.5 Jazyk pro definici omezení – OCL . . . 13
  - 2.5.1 Jazykové prostředky OCL . . . 14
- 2.6 Přehled UML nástrojů . . . 16
  - 2.6.1 Visual Paradigm . . . 16
  - 2.6.2 IBM Rational Rose Software Architect . . . 17
  - 2.6.3 Sparx Enterprise Architect 8 . . . 17
  - 2.6.4 Papyrus 4 UML . . . 18
  - 2.6.5 ArgoUML . . . 18
  - 2.6.6 UML2 plug-ins for Eclipse . . . 18

3 Komponentové systémy **22**
- 3.1 Principy komponentových systémů . . . 22
  - 3.1.1 Rozdíly mezi CBD a OOP . . . 23
  - 3.1.2 Znovupoužitelnost komponent . . . 23
  - 3.1.3 Komunikace komponent . . . 23
  - 3.1.4 Hierarchická dekompozice komponent . . . 23
  - 3.1.5 Propojení komponent . . . 24
- 3.2 Softwarová architektura . . . 24
- 3.3 Existující komponentové modely . . . 25
  - 3.3.1 Průmyslové komponentové modely . . . 25
  - 3.3.2 Formální komponentové systémy . . . 27
- 3.4 Modelování komponentových systémů v UML 2.0 . . . 28
  - 3.4.1 Modelování komponent a jejich hierarchická kompozice a dekompozice (Components) . . . 29
  - 3.4.2 Rozhraní (Interfaces) . . . 30
  - 3.4.3 Porty (Ports) . . . 31

3.4.4 Konektory (Connectors) . . . 31
3.4.5 Vazby (Relationships) . . . 32
3.4.6 Rozmístění (Deployment) . . . 32
3.4.7 Grafická notace . . . 32

4 Tvorba UML Profilu **35**
4.1 Meta-model komponentového systému . . . 35
4.1.1 Komponenty . . . 37
4.1.2 Porty . . . 37
4.1.3 Typ portu . . . 37
4.1.4 Rozhraní . . . 38
4.1.5 Vztahy . . . 38
4.1.6 Změny vůči původnímu meta-modelu . . . 38
4.2 Metodika a implementace UML Profilů . . . 39
4.2.1 Předpoklady . . . 39
4.2.2 Postup . . . 39
4.3 Implementace profilu komponentového systému . . . 44
4.3.1 Identifikace stereotypů a meta-třídy, které budou rozšiřovat . . . 44
4.3.2 Vztahy . . . 45
4.3.3 Tagged values a atributy . . . 45
4.3.4 Určení OCL omezení . . . 45
4.4 Problémy při implementaci . . . 50
4.5 Nasazení profilu (*Deployment*) . . . 50
4.6 Přenositelnost a podpora v CASE nástrojích . . . 51

5 Praktické použití profilu **52**
5.1 Příkladová studie (zadání) . . . 52
5.2 Ukázka použití profilu a modelování systému . . . 52
5.2.1 Instalace doplňku (plug-in) . . . 52
5.2.2 Aplikace profilu . . . 53
5.2.3 Modelování systému . . . 54
5.3 Validace a ladění modelu . . . 55
5.4 Artefakty . . . 58

6 Zhodnocení a závěr **59**
6.1 Zhodnocení . . . 59
6.2 Další rozvoj . . . 60

A UML Profil pro komponentové systémy **65**

B Úplný diagram případové studie **72**

C Obsah CD **74**

Kapitola 1

Úvod

Od doby softwarové krize v šedesátých letech dvacátého století se softwaroví inženýři a plánovači projektů snaží hledat nové postupy vývoje softwaru, technologie a nástroje, které by pomohly předcházet příčinám této krize. Stále se snažíme z umění vytvořit čistě inženýrské postupy, které by umožnily vyvíjet software, protože tradiční inženýrské postupy se nedaly na software aplikovat [24]. Výsledkem této snahy by měl být způsob, jak software udělat udržovatelný, dobře dokumentovaný a vývoj provést rychle, kvalitně, s minimem chyb a finančních zdrojů. Od strukturovaného paradigmatu došlo softwarové inženýrství k funkcionálnímu, či logickému paradigmatu a později k velmi oblíbenému objektově orientovanému paradigmatu (OOP).

Objektově orientované programování přineslo spoustu příslibů do budoucna, včetně obchodování s objekty, ale někteří kritici ([24], [25]) tvrdí, že objektově orientovaný přístup je příliš zaměřený na technickou stránku věci, vede na vytváření monolitických aplikací, nepodporují rekonfiguraci vyvíjeného systému a nenaplňují očekávání vkládaná do obchodování s třídami nebo knihovnami na trhu s objekty.

Jako řešení nabízí vývoj založený na komponentách *CBD – Component Based Development* (nebo také *CBSE – Component Based Software Engineering*) s ústřední roli komponenty, zastávající roli stavebních bloků, které mají podobu „black-boxu“. S okolím komunikují jen přes své rozhraní a tím důsledně podporují opětovné použití komponenty a svou snadnou výměnu [24]. Kromě CBD se dnes prosazuje další moderní přístup ke tvorbě softwarových systémů – SOA *Service Oriented Architecture*. Mají spolu s CBD společné rysy, ale rozdíly jsou v jiné míře abstrakce [22]: SOA zahrnuje metody návrhu, analýzy a implementace, klade důraz na způsob komunikace služeb, zatímco komponenty v podání CBD nekladou takový důraz na způsob komunikace (vždy záleží na konkrétní technologii, na které jsou komponenty postaveny). Komponenty v CBD mohou být stavebními prvky služeb v SOA [8].

Mimo čistě komerčních komponentových systémů jako je CORBA nebo Enterprise Java Beans vznikají také výzkumné komponentové systémy s formálním základem a chováním popsaným formálním jazykem. Také vznikají nové systémy s novými principy, s podporou dynamické nebo mobilní architektury.

Kromě samotných technologií a přístupů k výstavbě SW systémů se soustavně klade důraz na předchozí návrh aplikace. Dnes se k tomuto účelu často používá jazyk *UML (Unified Model Language)*, který poskytuje formálně definovaný jazyk k modelování systémů. Ačkoliv je UML jazykem s velkým záběrem a nabízí také prostředky pro modelování komponentových systémů, existují situace, a to nejen při návrhu komponentových systémů, kdy tento základ nestačí.

Při návrhu UML si tento nedostatek autoři uvědomili a na rozšíření jazyka pamatovali. Technikami meta-modelování lze UML obohatit o novou sémantiku a podpořit tak softwarové inženýry při návrhu aplikací s využitím podporovaných komponentových systémů.

Tyto technologie umožňují rychlou výrobu softwaru kompozicí z prefabrikovaných komponent (nebo služeb) a tím šetří čas i prostředky na vývoj i údržbu. Odstraňuje nutnost budování systémů od úplných základů a usnadňuje rekonfiguraci systému v případě potřeby [24]. Dalšího zlepšení můžeme dosáhnout důsledným návrhem vyvíjené aplikace a zde pomáhají CASE nástroje, které mohou využít přidané sémantiky UML.

1.1 Cíl práce

Cílem této práce je seznámit se s modelovacím jazykem UML 2.0. Prostudovat způsoby jeho rozšiřování a přizpůsobování modelů jazyka UML. Prostudovat principy komponentových softwarových systémů, vývoji založeném na komponentách – CBD a způsobem jejich modelování.

Následně navrhnout postup vytváření UML profilů, pro modelování komponentového systému na základě vybraného meta-modelu, který umožní tvorbu komponentových diagramů pro popis rozmístění, propojení a hierarchickou kompozici komponent.

S pomocí UML profilu demonstrovat jeho použití ve formě příkladové studie.

Dosažený výsledek zhodnotit a nabídnout možná rozšíření.

1.2 Struktura práce

Práce pokrývá teoretické základy jazyka UML, meta-modelování a modelování softwarových systému s využitím technologií UML 2.0. Teoreticky pokrývá základy komponentových systémů (CBD) a stručně popisuje principy existujících průmyslových, i výzkumných, komponentových systémů.

Kapitola 2 popisuje jazyk UML, jeho strukturu a způsoby rozšíření sémantiky UML pro lepší popis domény řešených problémů. V závěru této kapitoly je představeno několik UML nástrojů a jejich vlastnosti.

Ve druhé polovině práce se zabývám tvorbou UML profilu, na začátku kapitoly 4 vysvětluji sémantiku meta-modelu komponentového systému, pro který jsem v následujících podkapitolách vytvořil UML profil a pokusil se zobecnit postup jeho vytváření – nebo lépe – poskytnout oporu tvůrcům UML profilu, ve formě rad, kterými se při návrhu vlastních profilů mohou řídit.

V kapitole 5 je na zvoleném příkladě demonstrováno použití profilu.

Poslední kapitola nabízí zhodnocení výsledků práce a nabízí další možný rozvoj.

Kapitola 2

Modelování SW systémů

Při vývoji libovolného systému procházíme, kromě samotné implementace, také fází, ve které analyzujeme co bude výstupním produktem, jak to bude vypadat a jaké to bude mít funkce. U softwarových systémů a různých metodik jeho vývoje tak vždy dojdeme k tomu, že po identifikaci požadavků budeme chtít části vyvíjeného systému nějakým způsobem popsat tak, aby byl výsledný popis výstižný a srozumitelný vývojářům. Činnost návrhu systému se označuje jako modelování.

Velmi důležitým cílem při modelování software je zvládnout složitost, případně rozsáhlost, systému a v neposlední řadě také usnadnit komunikaci mezi členy týmu, kteří na něm pracují.

Produktem modelování je zjednodušení (abstrakce) reálného systému [12], u kterého zanedbáváme méně podstatné, nebo matoucí detaily.

Ve snaze docílit výše uvedených cílů, byly vymyšleny různé způsoby modelování softwaru, ale v současnosti je patrně nejznámější a nejpoužívanější jazyk UML (*Unified Modelling Language*).

Tento text se věnuje především softwarovým systémům. UML však lze využít pro modelování také jiných, než softwarových systémů (např. jej lze využít k popsání *firemních procesů*).

Tato část práce pojednává o jazyku UML a popisuje jeho základy a součásti. Následující kapitola se zaměřuje na možnosti rozšíření jazyka a v závěru této části práce jsou popsány některé nástroje, které podporují modelování systémů v jazyce UML.

2.1 Jazyk UML

Jak již bylo naznačeno v úvodu: UML (*Unified Modelling Language*) je formálně definovaným, standardizovaným modelovacím jazykem pro specifikaci, návrh a dokumentaci softwarových, ale i jiných systémů a je nezávislý na konkrétním implementačním jazyku. Používá vizuální styl zápisu a je vhodný pro modelování typických objektově orientovaných softwarových systémů. Pro tuto práci je také důležité, že se dá také dále rozšiřovat.

UML ale nepředepisuje metodiku, ani způsob použití, takže lze pracovat a jazyk používat téměř libovolným způsobem. Literatura [12] uvádí tři obvyklé přístupy: pro zběžné náčrty (*sketch*, použití většinou ručně a bez pomoci SW nástrojů), pro vytváření detailních návrhů (*blueprint*, použití s pomocí podpůrných SW nástrojů, časté je generování zdrojových kódu z diagramů, nebo generování diagramů ze zdrojových kódu) a použití UML jako programovacího jazyka (UML nástroj umožní vytvořený model přímo spustit).

V UML pracujeme s abstrakcí reálného systému a různými prostředky jazyka vytváříme abstrakci systému – *model*. V rámci modelu se snažíme zachytit vztahy mezi elementy navrhovaného systému. UML nabízí různé typy diagramů, kde každý z nich umožňuje zachytit jiný pohled (*view*) na model a tak zobrazit pouze relevantní aspekty pro daný model a ostatní ignorovat.

Jednotlivé diagramy lze rozdělit do skupin na:

- *diagramy struktury*: *diagram tříd (Class)*, *diagram objektů (Object)*, *diagram nasazení (Deployment)* a pro potřeby této práce podstatný *diagram komponent (Component)*
- *diagramy chování*: *diagram aktivit (Activity)*, *diagram případů užití (Use Case)*
- *diagramy interakce*: *sekvenční diagram (Sequence)*, *diagram komunikace (Communication)*

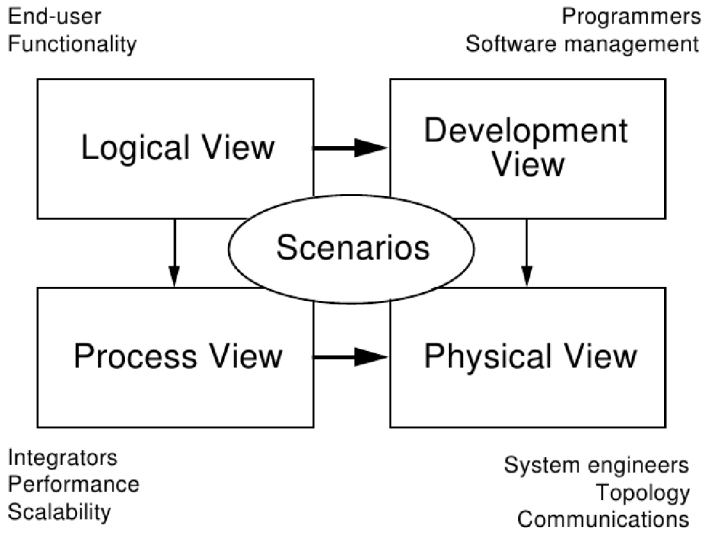
Uvedený výčet diagramů není úplný, podrobný seznam a popis diagramů UML lze najít například v [12].

Pohledy na systém můžeme rozdělit podle tzv. modelu architektury „*4+1 Architectural View Model*“ (na obrázku 2.1, dle [9]):

- *logical view* – logický pohled, obsahuje abstraktní popis části systému; používá se k modelování systému z hlediska jeho součástí a vazeb mezi nimi (diagram tříd, sekvenční diagram)
- *process view* – procesní pohled, modeluje chování systému (diagram aktivit)
- *development view* – implementační pohled; popisuje jak jsou jednotlivé části organizovány do modulů a komponent (diagram komponent)
- *physical view* – fyzický pohled popisuje vztah mezi částmi modelu a reálným systémem (diagram nasazení)
- *scenarios* – pohled případů použití (také se někdy označuje jako *use case view*, [12]) zachycuje vnější požadavky na systém a integruje předchozí pohledy (diagram případů užití)

Oproti jiným modelovacím jazykům (nebo metodám) má UML několik významných výhod, je [12]:

- *formální* – jedná se o formální jazyk s přesně definovanou sémantikou
- *standardizovaný a otevřený* – jazyk je standardizovaný a otevřený pro každého, kdo se chce podílet na vývoji, zároveň je ale vývoj řízen konsorciem společností, akademické obce a odborníků z praxe; takto je zajištěna nezávislost na jednom dodavateli, kompatibilita a spolupráce implementací UML
- *praktický* – s předchozím bodem souvisí i jeho původ v praxi a použití „best practices“ při jeho vývoji
- *škálovatelný* –je vhodný jak pro malé, jednoduché, tak i pro rozsáhlé a složité systémy
- *srozumitelný* – UML využívá snadno zvládnutelný a jednoduchý zápis (např. při srovnání s vývojovým diagramem) s dobře dokumentovanou sémantikou



Obrázek 2.1: Příklad „4+1“ Architectural View Model, podle [9]

- *prověřený časem* – historie standardu UML sahá až do roku 1995 a od té doby dospěl do robustní, léty prověřené, podoby
- *známý* – mezi vývojáři je velmi známý

2.2 Architektura UML

Standard UML je spravován konsorciem OMG (*Object Management Group*), který udržuje standard a koordinuje práci na dalším vývoji. Tato organizace také poskytuje specifikaci (nikoliv však implementaci).

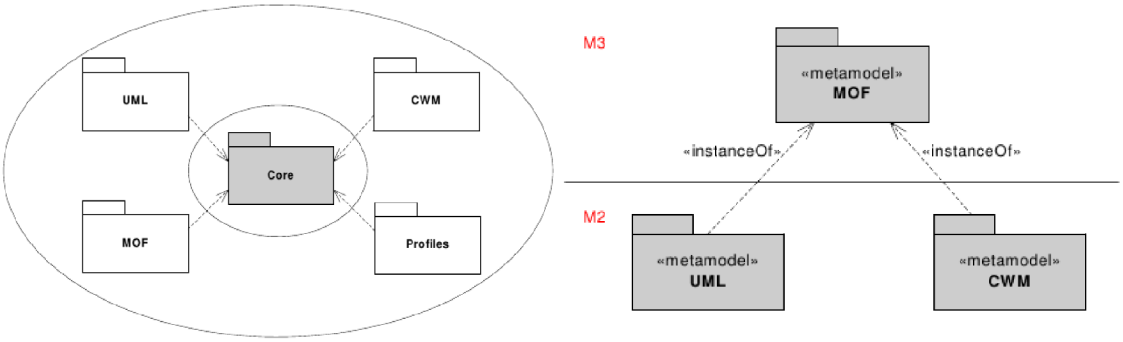
V době vzniku práce, se pro standardy související s UML, používá souhrnné označení *UML 2*. Poslední stabilní verze je UML 2.3, verze 2.4 je v beta verzi.

Specifikace se skládá ze čtyř dokumentů (lze nalézt na webových stránkách organizace OMG, [20]):

- *UML Superstructure* — popisuje diagramy statické struktury a chování, je určena pro programátory a analytiky [15]
- *UML Infrastructure* — definuje jádro (balík *Core*), obsahující meta-třídy, nad kterými je možné vystavět meta-modely; ve vztahu k UML to je meta-model UML (ze specifikace Superstructure), MOF (*Meta Object Facility*), nebo meta-model UML profilů. Kromě toho nad těmi samými meta-třídami jsou postaveny i elementy samotné knihovny *Infrastructure*; říkáme, že je *reflexivní* [14]
- *UML Object Constraint Language (OCL)* -– jazyk používaný ke specifikaci omezení nad elementy UML [19]
- *UML Diagram Interchange* — definuje formát přenosu UML struktur – XMI (*XML Metadata Interchange*)

Balík *Core* [14] v *Infrastructure Library* v sobě obsahuje další balíky, ve kterých jsou definovány znovupoužitelné „stavební bloky“: balík *Core::PrimitiveTypes* obsahuje primitivní datové typy *Boolean*, *Integer*, *String* a *UnlimitedNatural* (přirozená čísla z intervalu $\langle 0, \infty \rangle$); balík *Core::Abstraction* obsahuje většinou abstraktní meta-třídy, což znamená, že v odvozených meta-modelech se použijí pouze jejich specializace a rozšíření; balík *Constructs* obsahuje meta-třídy (ne-abstraktní) a nakonec balík *Basic*, který obsahuje podmnožinu balíku *Constructs* a tvoří základ pro meta-modely MOF, nebo UML XMI.

Kromě balíku *Core* obsahuje *Infrastructure Library* také balík *Profiles* [14], ve kterém jsou specifikovány prvky potřebné pro vytváření – převážně – UML profilů, ale obecně lze vytvořit profil rozšiřující kterýkoliv meta-model.



Obrázek 2.2: Přístupy k definici meta-modelů nad *Infrastructure Library*, převzato z [14]

Matoucí může být, že UML je použito zároveň k definici MOF, tak že oba sdílí balík *Core* a zároveň je MOF meta-metamodelem pro modely vytvořené jazyce UML (a tedy je meta-modelem pro model jazyka UML). UML tak částečně definuje samo sebe. Oba přístupy lze vidět na obrázku 2.2.

2.3 Meta-modelování

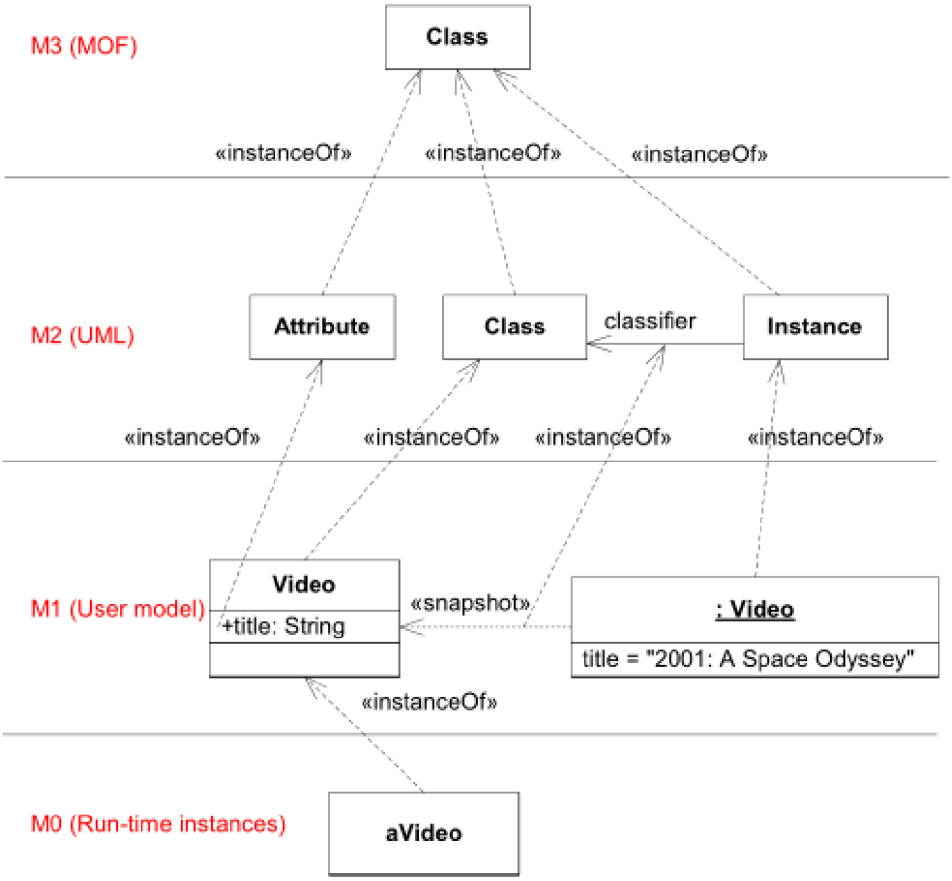
Model je instancí meta-modelu. Meta-model definuje pravidla chování elementů modelu [12]. Každý model může zároveň sloužit jako meta-model pro další „vrstvu“. Meta-modelováním rozumíme proces, ve kterém vytváříme jazyk modelu popisující doménu modelu [14]. Podobně lze také přidat další předponu „meta“ a použít výraz meta-metamodel ve smyslu instance meta-modelu.

Vztahy mezi modely a meta-modely zachycuje tzv. čtyřvrstvá hierarchie meta-modelů (*Four-layer Meta-model Hierarchy*), která definuje čtyři vrstvy označované jako *M3, ..., M0*. M3 se označuje jako vrstva meta-metamodelu, M2 je instancí vrstvy M3 a označuje se jako vrstva meta-modelu, na této vrstvě je usazen jazyk UML, vrstva M1 je vrstvou uživatelských modelů v jazyce UML (instance tříd z M2) a M0 představuje instance elementů z uživatelského modelu (vrstva M1). Tato hierarchie poskytuje potřebnou perspektivu, abychom se v předponách „meta“ neztratili.

Příklad čtyřvrstvé hierarchie je na obrázku 2.3, kde jde vidět, že elementy vrstvy s nižším indexem (model) jsou instancí elementů vrstev s vyšším indexem (meta-model), např. video v reálném světě je v uživatelském modelu reprezentováno třídou *Video*, která je opět instancí elementu *UML::Class* a ta je instancí elementu *MOF::Class*. Každý element modelu UML

je instancí právě jednoho elementu modelu MOF [6].

Vrstva označena jako M3 je tvořena MOF, což naznačuje, že meta-modelář není omezen pouze na tyto čtyři vrstvy a je možné přidávat další [14].



Obrázek 2.3: Příklad čtyřvrstvé hierarchie, převzato z [14]

2.3.1 MOF a EMOF

MOF (Meta Object Facility) je standard spravovaný organizací OMG a současně rámec (*framework*) poskytující podporu pro vývoj modelů [18]. K tomu využívá prostředky *Infrastructure Library*, které jsem zmínil v kapitole 2.2. Definuje formát XMI pro přenášení UML modelů mezi různými nástroji a také mechanizmus pro rozšiřování meta-modelu (profily jsou podmnožinou tohoto mechanizmu), který může být použit místo, nebo v kombinaci s technikou vytváření profilů [14]. Účelem MOF je místo definování nového jazyka, pro úrovně M2 a M3, použít stávající zažitý jazyk úrovně M1.

Standard MOF 2 se skládá ze dvou částí: EMOF (*Essential MOF*) a CMOF (*Complete MOF*). *EMOF* je minimální podmnožina MOF a slučuje (*package merge*) elementy z balíku *Basic* UML 2 a přidává k nim další vlastnosti z balíků *Reflection*, *Identifiers*, *Extension* a

Common.

- *Reflection* – rozšiřuje model o schopnosti důležité pro popis sebe sama (*self-describing*)
- *Identifiers* – rozšíření pro spolehlivou identifikaci objektů meta-modelu a odstraňuje nutnost spoléhat se při tom na data poskytovaná modelem
- *Extension* – přidává možnost rozšíření elementů modelu o páry „název/hodnota“

Primárním cílem EMOF je umožnit vytváření jednoduchých meta-modelů tím, že poskytuje jen nutný základ [18]. Strukturu MOF 2 zachycuje obrázek 2.4.

Package merge [14] je vazba mezi dvěma balíky *Package*, která má význam sloučení obsahu cílového balíku do zdrojového balíku. Elementy, které jsou v cílovém balíku, ale nikoliv ve zdrojovém, jsou zkopírovány do zdrojového. Pokud se element nachází v obou balících, dojde ke zkombinování jejich vlastností.

CMOF staví nad EMOF a přidává další vlastnosti, více v [18].

2.4 Možnosti rozšíření UML

UML je vhodné [12] pro modelování téměř jakéhokoliv objektově orientovaného (nebo tomuto paradigmatu blízkého) systému. UML je ale do značné míry aplikačně i jazykově nezávislé. Často bývá potřeba dodat modelům v UML další sémantiku navíc, ať už jde přímo o jazyk (např. Java, C++), nebo platformu nad kterou vyvíjíme (JEE, .NET), nebo pro specifickou doménu problému. V UML máme k dispozici různé elementy, které máme k dispozici při modelování systému: třídy, rozhraní, komponenty, objekty, atd. Při návrhu systému bychom mohli chtít podpořit jak návrháře, tak programátora a těmto standardním elementům UML dodat další význam, který můžeme odvodit od meta-modelu našeho systému, pokud ho máme, nebo např. od terminologie. Návrháři by tak mohlo přijít k užitku, kdyby pouhým opatřením třídy stereotypem získal seznam atributů, které může v modelu použít. Ideální by také bylo, kdyby ho nástroj upozornil na chyby, které udělal.

Této podpory se dá dosáhnout rozšířením UML. Důvodů, proč bychom měli chtít rozšířit UML je hodně, jsou to například:

- přidání nové sémantiky
- propojení terminologie domény problému a modelu systému
- přidání nové syntaxe pro elementy, které v UML nejsou
- vytvoření nových vlastností, nebo změna stávajících vlastností
- přidat omezení k meta-modelu tak, abychom povolili jen takový způsob nakládání s elementem modelu, který chceme

Některé z uvedených bodů lze realizovat snadno a některé hůře a to za cenu s tím spojených nevýhod.

Jestliže určíme doménu našeho problému, musíme si nejdříve ujasnit, zda prvek této domény je ve vztahu k elementům UML specializací, generalizací, nebo zda vůbec s nějakým konceptem v UML souvisí. Pokud bychom chtěli rozšířit UML o podporu „našeho nového“ komponentového systému, tak nejspíše zjistíme, že v UML pojem komponenta existuje a má takovou sémantiku, že nám stačí k němu přidat pár dalších vlastností (tj. naše komponenta

by byla speciálním případem komponenty z UML). Pokud by UML komponenta měla vlastnosti, které naopak naše nemá, tak bychom tu z UML museli omezit. Může se také stát, že v UML vůbec nenajdeme podobný koncept.

Pokud bychom netrvali na použití UML, tak můžeme vytvořit vlastní meta-model a vlastní jazyk, bez jakékoliv přítomnosti UML. Tato možnost sebou nese značná rizika, proto se nabízí využití technologií spojených s UML.

Nabízí se nám tři způsoby [14]:

1. použití MOF a vytvoření vlastních elementů meta-modelu, bez použití elementů meta-modelu UML
2. použití MOF a využití části meta-modelu UML
3. využít balíku *Profile* a postavit rozšíření nad celým UML

2.4.1 Vytvořením nebo změnou meta-modelu

Mezi bodem 1 a 2 z předchozí kapitoly není v konečném důsledku až tak velký rozdíl. V prvním případě by bylo nutné vytvořit kompletně nový software pro podporu modelování v unikátním jazyce. Ve druhém případě by bylo nutné stávající SW upravit. Při malých změnách by druhá možnost byla méně bolestná.

Ačkoliv může být velkou výhodou extrémní možnost přizpůsobení, prakticky neomezená volnost (vytváření nových typů diagramů, nebo elementů) a při použití MOF (resp. EMOF) také relativně snadná tvorba nového modelu, tak zároveň je velkou nevýhodou ztráta kompatibility s existujícími nástroji. UML je velmi dobře známé mezi vývojáři. Změněnou verzi UML však nejspíše znát nebudou a nemohou se spolehnout ani na základy, které by při použití UML profilu zůstaly. Také existuje velká pravděpodobnost zavlečení chyby již při samotném návrhu meta-modelu.

Přes všechny problémy tyto techniky přináší největší možnosti přizpůsobení a může být výhodné je použít, pokud je nutné se velmi odchýlit od UML a zároveň máme k dispozici přesný a formální popis řešené domény. Obecně však může být výsledek nejistý a v budoucnu přinést jak problémy s formálním návrhem, tak i náklady spojené s udržováním vlastních nástrojů.

2.4.2 Rozšíření použitím UML Profile

Univerzálnost jazyka UML, ale zároveň jeho – po stránce kompatibility – komplikované rozšíření popsané v 2.4.1, vedlo konsorcium OMG k zařazení i méně komplexního (tzv. „lightweight“ [12]) mechanizmu rozšíření, který by umožnil upřesnit sémantiku elementů UML k použité platformě, nebo doméně řešeného problému a zároveň by se vyhnul problémům spojeným s předefinováním meta-modelu UML při zachování jeho výhod a ekosystému (podpora nástrojů, rozšířenost).

Součástí knihovny *Infrastructure* se tak stal balík *Profile*, který poskytuje prostředky k rozšíření meta-tříd definovaných v MOF, aniž bychom vytvořili nekompatibilní verzi UML. Tato část specifikace se nazývá – stejně jako balík – *Profile* a zabývá se tvorbou profilu. Aplikujeme-li tuto metodu na jazyk UML, mluvíme o UML profilu. Obecně můžeme vytvářet profily z jakéhokoliv jazyka, který je postaven nad MOF [14].

UML profily lze vytvářet ručně, ale mnoho CASE nástrojů (*Computer Aided Software Engineering, softwarové nástroje pro podporu vývoje softwarových systémů*) poskytuje nástroje pro vizuální tvorbu profilů. Tyto profily pak lze mezi jednotlivými modely a často

i mezi jednotlivými programy přenášet pomocí XMI formátu (viz. kapitola 2.2). Profily lze také jednoduše, na rozdíl od MOF, „překrývat“ a tím kombinovat více profilů současně v jednom modelu.

Použití profilů k přizpůsobení UML konkrétní doméně je bezpečným a preferovaným způsobem, ale přesto není vhodné tyto techniky používat příliš lehkovážně. Použití je vhodné tehdy, pokud se na jeho podobě shodne většina lidí používající platformu, nebo pracující s řešenou doménou, jedině tak se takový profil bude široce používat a bude možné na jeho základě vytvořit speciální nástroje, které např. z UML generují zdrojový kód [12].

Touto technikou nelze vytvořit nový typ diagramu, nebo elementu. Pouze lze upřesnit význam již existujících elementů. Jeho atributy a vlastnosti zůstanou zachovány. Grafické přizpůsobení je také velmi omezeno, neboť CASE nástroje většinou pouze umožní ke stereotypu přiřadit grafický symbol, který doplní původní tvar.

Slibnou vlastností profilů je jejich podpora v CASE nástrojích, které mohou validovat vývojářem vytvořené modely, podle omezení, která zadal tvůrce profilu. Teoreticky také není potřeba měnit softwarové vybavení, abychom mohli použít profil, bohužel v praxi není zatím podpora profilů implementována ve všech nástrojích a pokud podpora existuje, tak má často kolísavou kvalitu a různý rozsah.

Konsorcium OMG udržuje několik často používaných UML profilů, například profily pro CORBA, EJB a další[1].

Pro definici profilů se používají standardní prvky jazyka UML:

- stereotypes (stereotypy)
- tagged values (připojený seznam atributů)
- constraints (omezení)

Jejich použití však v minulosti nebylo nijak omezeno, což vedlo na jejich špatné použití. V UML 2 je oproti dřívějším verzím specifikováno, jak se má profil vytvářet a používat v uživatelských modelech [12].

Profil může mít podobu klasické dokumentace. Dokumenty popisují modelářem vytvořené stereotypy, „tagged values“ a omezení, u kterých je popsán jejich význam, případně jejich podrobné „mapování“ na konkrétní doménu. Tyto definice může doplnit diagram UML profilu, který je oproti obyčejným diagramům tříd omezen na třídu stereotypu, meta-třídy, vazbu generalizace-specializace a na agregaci a kompozici. V některých nástrojích je však dostupná pouze kompozice (Visual Paradigm), ale ve stejném významu, jako v jiném nástroji agregace (Rational Software Architect), který pak nabízí obě dvě, ale kompozici omezenou pouze ke spojení třídy a stereotypu (nebo meta-třídy). Tuto nekonzistenci v implementaci přičítám ranému stádiu podpory profilů v těchto nástrojích a nedostatečné podpoře tohoto tématu ve specifikaci UML. Příklad profilu je na obrázku 2.5.

V následujícím textu popisuji notaci uvedených prvků pro definici UML. Vycházím z [12], pokud není uvedeno jinak. Nicméně UML nástroje se v mnoha případech nedrží oficiální syntaxe.

Stereotypy

Stereotypy lze přiřadit téměř každému prvku v UML a mají význam zvláštního případu, nebo použití ([12]) – přidávají význam elementu v UML tak, aby přesněji popisoval roli

[1] http://www.omg.org/mda/specs.htm#Profiles

elementu v rámci našeho modelu. Například v JEE (*Java Enterprise Edition* existují třídy, ale u některých můžeme užitím stereotypu «`EJB`» (deklarovaném v profilu JEE) vyjádřit, že se jedná konkrétně o *Enterprise Java Bean*.

Prvky v UML, kterým je přiřazen nějaký stereotyp, jsou často označovány názvem stereotypu, který je uzavřen mezi francouzské uvozovky (také se používají zdvojené úhlové závorky): «`stereotyp`». Počet přiřazených stereotypů není omezen, v takovém případě jsou názvy odděleny čárkou: «`stereotyp1, stereotyp2`». Některé nástroje umožňují stereotypu přiřadit grafický symbol.

V ukázce na obrázku 2.5 jsou deklarovány dva nové stereotypy, které mají samy přiřazeny stereotyp «`stereotype`»: «`WebService`» a «`Exposed`». Šipky v diagramu znamenají, že je možné tyto stereotypy přiřadit, např. třídě, nebo operaci (jsou označeny stereotypem «`metaclass`» a jedná se o prvky definované v meta-modelu).

Tagged Values

Tagged values jsou prostředkem, jak zachytit další informace spojené se stereotypem. Každý stereotyp může mít svou množinu „tagged values“, které jsou odděleny od standardních atributů elementu, tudíž nedochází k jejich překrytí. Důležité je, že se jedná o vlastnosti stereo-typovaného elementu modelu a nemají vlastní grafickou reprezentaci v UML (jako např. elementy *Port*, nebo *Interface*).

„Tagged values“ mají podobu seznamu `klíč=hodnota`. Jejich přiřazení stereotypu se provede s využitím poznámky. Např. tak, jak je to ukázáno na obrázku 2.6, nebo při vytváření UML profilu jako atributy „stereotypu“ (viz. obrázek 2.5).

Příklad vychází z existence webové stránky, která obsahuje formulář určený k autentizaci uživatele. Validace textových polí je vlastností formuláře, nikoliv samotného elementu UML, proto je vlastnost `validate=true` přiřazena stereotypu a nikoliv elementu.

V příkladu z obrázku 2.5 jsou se stereotypem «`WebService`» spojeny hodnoty `service=ServiceStyle` a `encoding=EncodingStyle`. *ServiceStyle* a *EncodingStyle* mohou nabývat hodnot svého typu, které jsou reprezentovány výčtem (třída se stereotypem «`enumeration`»).

Constraints

Omezení jsou spojena se stereotypem a definují omezení nad elementy meta-modelu. Pro zápis omezení v diagramech se často používají poznámky, protože omezení nemají v UML přiřazen žádný symbol. Omezení nemusí být aplikována pouze v rámci profilu. Pokud jsou aplikována v profilu, tak se vztahují na instance elementu meta-modelu, na které je aplikován stereotyp ke kterému jsou přiřazeny, nikoliv na stereotyp jako takový.

Omezení mohou být popsány přirozeným jazykem, nebo jazykem určeným speciálně k definici omezení – OCL *Object Constraint Language*. Pokud je použito OCL a použitý CASE nástroj takovou funkci podporuje, lze platnost podmínek v modelu ověřit.

2.5 Jazyk pro definici omezení – OCL

V celé kapitole o OCL čerpám z oficiální specifikace k tomuto jazyku. Cílem tohoto dokumentu není detailně popsat jazyk OCL, ale pouze poskytnout čtenáři základy jazyka pro pochopení omezení definovaných v pozdější části práce. Pro podrobný popis jazyka odkazuji čtenáře na dokumentaci [19].

Podle specifikace [19] je OCL (*Object Constraint Language*) „formálním jazykem používaným pro zápis výrazů nad modely v UML“. Tyto výrazy většinou specifikují integritní, nebo typová omezení, ale také mohou specifikovat, jak má vypadat výsledek nějaké operace. Často mají podobu invariantů nad modelem a objekty modelu, nebo právě podmínek určujících stav systému před a po vykonání operace. V OCL je garantováno, že vyhodnocení výrazu pouze vrátí hodnotu a nezmění stav systému, takže nedochází k vedlejším efektům, ačkoliv v OCL lze specifikovat, že se má systém změnit po provedení nějaké akce.

OCL je dalším standardem, který spravuje OMG. Byl navržen, aby vyplnil mezery v modelování UML diagramy, které nepokrývají všechny aspekty systému. Mimo jiné také doplňuje omezení specifikovaná přirozeným jazykem, která vedla k nejednoznačnosti a tím k praktické nemožnosti automaticky validovat model. Kromě přirozeného jazyka by k tomuto účelu šlo využít klasických formálních jazyků, např. predikátové logiky, ale výhodou OCL je jeho srozumitelnost i pro uživatele bez dobré znalosti matematiky.

Specifikace OCL navrhuje mnoho různých použití jazyka OCL:

- jako dotazovací jazyk
- jazyk ke specifikaci invariantů nad třídami a typy v modelu tříd
- ke specifikaci invariantů nad stereotypy
- ke specifikaci podmínek platných pro operace a metody před a po jejich provedení (tzv. *pre-conditions* a *post-conditions*)
- k popisu podmínek, které musí být splněny pro provedení přechodu (např. ve stavových diagramech)
- ke specifikaci komunikace a akcí
- ke specifikaci omezení nad operacemi
- ke specifikaci odvozovacích pravidel pro atributy všech výrazů nad UML modelem

OCL je modelovacím a specifikačním, nikoliv programovacím, jazykem s typovou kontrolou. OCL definuje, kromě primitivních typů, převzatých z UML (více v kapitole 2.2), také typy pro kolekce (*collection*, *set*, *bag*, *sequence*) a základní předdefinované typy: *OclAny* (nejobecnější typ, kterému musí vyhovovat všechny ostatní typy) a jeho specializace *OclMessage*, *OclVoid*, *OclInvalid*. Každý výraz v OCL má svůj typ.

2.5.1 Jazykové prostředky OCL

Kód OCL omezení se skládá ze specifikace kontextu (uvozený klíčovým slovem *context*), ve kterém bude výraz vyhodnocen, dále z klíčového slova *inv* (invariant), *pre* (pre-condition) a *post* (post-condition) následovaného volitelným názvem a nakonec povinným tělem výrazu (nebo těly výrazů – v rámci jednoho kontextu je možné definovat více omezení, k oddělení se používá prázdný řádek), který specifikuje omezení [19]:

context *název instance*[: *Typ*]
  {**inv**|**pre**|**post** [*název omezení*]: *tělo výrazu*}

Tělo výrazu se skládá z identifikace instance v rámci omezení a přístupu k vlastnostem nebo operacím. Kromě samotného názvu instance lze použit také klíčové slovo *self*, které je referencí na aktuální instanci kontextu.

K vlastnostem instance se přistupuje operátorem „.“ (tečka). Vlastnosti lze za sebou řetězit (podobně jako v jazyce Java). Pro provádění operací nad vlastnostmi se používá operátor „−>“ (šipka).

Pokud k některé vlastnosti přistupujeme často, můžeme definovat novou proměnnou s obsahem této vlastnosti užitím klíčového slova *let*.

Vlastností (*property*) objektu souhrnně označujeme *atribut* objektu, *konce asociace*, *operaci* a *metodu*. U operace a metody mluvíme o vlastnosti pouze tehdy, pokud má jejich atribut *isQuery* hodnotu *true*, což zaručuje, že jejich provedení nebude mít vedlejší efekt. Atribut může mít v UML modelu násobnost, pokud je násobnost větší než 1, tak mluvíme o kolekci hodnot. Kolekce jsou reprezentovány abstraktním typem *Collection* a jeho podtypy.

V OCL jsou k dispozici klasické aritmetické operátory, operátory ekvivalence a logické operátory, včetně implikace *implies*, operátoru negace (*not*) a operace podmínky (*if-then-else*).

Předdefinované vlastnosti objektů

Všechny objekty v OCL mají několik společných vlastností:

- *oclIsTypeOf(t : Classifier) : Boolean* – kontrola zda se typ objektu shoduje s parametrem
- *oclIsKindOf(t : Classifier) : Boolean* – podobně jako předchozí, ale navíc vyhovuje i nadtyp
- *oclInState(t : Classifier) : Boolean* – kontrola, zda se stavový automat, přiřazený objektu, nachází ve stavu specifikovaném v parametru
- *oclIsNew() : Boolean* – je vyhodnocena jako *true*, pokud objekt existuje po provedení operace (*postcondition*), ale neexistoval před provedením operace (*precondition*)
- *oclAsType(t : Classifier) : instance of Classifier* – umožňuje přetypování v rámci typové hierarchie

Operace nad kolekcemi

Nad kolekcemi lze provádět operace umožňující dotazovat se na její vlastnosti, nebo provést výběr podmnožiny.

Základními operátory nad kolekcemi jsou univerzální (*forAll(*podmínka*)*) a existenční (*exists(*podmínka*)*) kvantifikátory. V případě *forAll* musí podmínka platit pro všechny prvky kolekce, zatímco u *exists* postačí platnost pro jediný prvek. Dalšími operátory jsou *collect*, *select* a *reject* pro vytváření, výběr a filtrování prvků kolekce. Pro sekvenční průchod kolekcí můžeme použít iterátor kolekcí (*iterate*). Často užívaným je také operátor *isEmpty()*, který má hodnotu *true*, pokud je kolekce prázdná, *notEmpty()* s opačným významem a operátor *size()* vracející počet prvků kolekce.

```
fakulta.student->forAll( s | s.slozilPrijmaciZkousku )
fakulta.ustav.zamestnanci->exists( z | z.vedouci )
fakulta.ustav.zamestnanci->notEmpty()
```

Navigace

Mezi elementy UML diagramu se můžeme navigovat použitím asociací, které mezi nimi jsou. Vytvoření asociace mezi elementy v modelu způsobí vytvoření vlastnosti v elementu. Názvy vlastností se řídí názvem role elementu. Pokud nejsou názvy rolí specifikovány, tak se použije název dle konvence (stejně tak se použije konvence, pokud není explicitně zadán název asociace).

Podle násobnosti na koncích asociace přistupujeme buď k atributu, nebo ke kolekci.

Invarianty

OCL výraz, omezení označeného stereotypem *invariant* (zástupné klíčové slovo *inv*), musí mít vždy pravdivou návratovou hodnotu. Všechny invarianty musí být vyjádřeny OCL výrazem s hodnotou typu *Boolean*.

```
context Fakulta inv minimalniPocetUstavu:
    self.pocetUstavu > 0
```

Preconditions a Postconditions

Podobně jako u invariantu, klíčová slova *pre* a *post* reprezentují stereotypy «`precondition`», resp «`postcondition`». Používají se k definici omezení na stav systému před (pre) a po (post) provedení operace.

Preconditions specifikují stav, ve kterém musí být systém před provedením operace.

Postconditions, přes své označení „condition“ (podmínka), mohou být použity k definici operace. K navrácení hodnoty slouží klíčové slovo *result*:

```
context Fakulta::pocetStudentu() : Integer
    post result = self.pocetStudentu
```

Operace pak mohou být volány z invariantů:

```
context Fakulta inv minimalniPocetStudentu:
    self.pocetStudentu() > 30
```

2.6 Přehled UML nástrojů

Ačkoliv je nástrojů pro modelování v UML mnoho, tak ne každý nabízí podporu pro tvorbu, nebo použití profilů. Tato část práce popisuje výběr několika známých UML CASE nástrojů a popisuje stav podpory, pro tuto práci důležitých technologií.

2.6.1 Visual Paradigm

Visual Paradigm for UML[2] je multiplatformní, komerční, modelovací nástroj s podporou všech druhů diagramů UML. Tvorbu UML profilů podporuje od verze 7.2 v edicích *Enterprise, Professional* a *Standard*. V komunitní verzi podpora pro UML profily (součást tzv. *Modelling Toolset*) není dostupná.

[2] http://www.visual-paradigm.com/product/vpuml/

Nástroj podporuje vizuální tvorbu UML profilů kreslením diagramu (který ale neodpovídá diagramu profilů). Ten umožňuje definování stereotypů, používání generalizace a kompozice. V „tagged values“ je možné používat datové typy, výčtové typy, text, víceřádkový text, typ elementu modelu. Chybí typ Boolean, který je ale možné nahradit výčtem.

Program umožňuje rozšířit většinu meta-tříd meta-modelu. Ve verzi 8.0 chyběla možnost rozšířit meta-třídy vztahů (*Association, Dependency, Binding*, atd.). Ve verzi 8.1 již tato možnost byla.

Programu chybí podpora pro OCL Editor, z toho vyplývá, že nepodporuje validaci OCL. Uživatelé tento nedostatek prozatím řeší zápisem výrazu do pole pro dokumentaci. Na profil ani model to však vliv nemá.

Visual Paradigm umí vytvořený UML profil (ukázka je na obrázku 2.7) aplikovat na libovolný model v rámci projektu a podporuje export profilu ve formě přehledné dokumentace ve formátu *HTML*. Pro účely přenositelnosti lze využít formát XMI 2.1. Přenositelnost takto vytvořeného profilu ve formátu XMI jsem vyzkoušel a profil šel úspěšně importovat a používat v software *Rational Software Architect*.

Podpora UML profilů ve Visual Paradigm je nedokonalá. Program podporuje pouze základní práci s profily. Nepodporuje OCL, ani validaci modelu.

V současnosti je program ve verzi 8.1 a je spustitelný v operačních systémech *GNU/Linux*, *Microsoft Windows* a *MacOS X*.

2.6.2 IBM Rational Rose Software Architect

IBM Rational Rose Software Architect[3] je také multiplatformní, komerční, modelovací nástroj. Stejně jako předchozí nástroj poskytuje podporu pro modelování v UML. Tvorbu UML profilů podporuje od verze 8.0.

Nabízí velmi podobné možnosti tvorby UML profilů (ukázka je na obrázku 2.8) jako Visual Paradigm, včetně vizuální tvorby diagramů (minimálně u komponentových diagramů používá vlastní notaci). Také je k dispozici import a export vytvořeného profilu ve formátu XMI 2.2, UML2 a formou pluginu.

Na rozdíl od Visual Paradigm má program kompletní podporu datových typů (včetně typu Boolean), disponuje OCL editorem a umožňuje provést validaci modelu.

Program bohužel plně nepodporuje, mezi vývojáři velmi široce používanou, „lolipop“ notaci u komponentových diagramů. V externím pohledu na komponentu sice lze používat port, „lolipop“ i „socket“, ale vizuální reprezentace je problematická.

Podpora profilů je na dobré úrovni, včetně relativně dobré podpory OCL. Z tohoto důvodu jsem k vytvoření profilu použil právě tento software.

Program je založen na platformě Eclipse a v současnosti je k dispozici ve verzi 8.0 pro operační systémy GNU/Linux a Microsoft Windows. IBM nabízí také časově omezenou, ale jinak plně funkční, zkušební verzi.

2.6.3 Sparx Enterprise Architect 8

Enterprise Architect společnosti *Sparx Systems*[4] je také komerčním nástrojem s širokou podporou standardů. Poskytuje podporu pro nejnovější verzi UML, podporuje tvorbu a použití UML profilů. Pro výměnu modelů používá formát XMI 2.1.

[3] http://www.ibm.com/developerworks/wikis/display/RSA/Home

[4] http://www.sparxsystems.com

Program je ve verzi 8 a je oficiálně dostupný pouze pro operační systémy MS Windows, avšak dobře funguje i v GNU/Linux, spuštěný přes program *WINE* (opensource implementace API MS Windows). Program je dostupný také v časově omezené, plně funkční, verzi.

2.6.4 Papyrus 4 UML

Papyrus[5] je opensource nástroj založený na Eclipse a kompatibilní s Eclipse UML2, implementující standard UML2 podle specifikace OMG. Podporuje standard pro přenos diagramů DI2, umožňuje definovat UML2 profily.

Program je ve verzi 1.12 a je zatím velmi nestabilní. A i když staví na kvalitních základech platformy Eclipse, tak uživatelský komfort a kvalita diagramů není příliš dobrá. Během krátkého testování programu jsem nebyl schopen najít Port při modelování komponentových diagramů.

2.6.5 ArgoUML

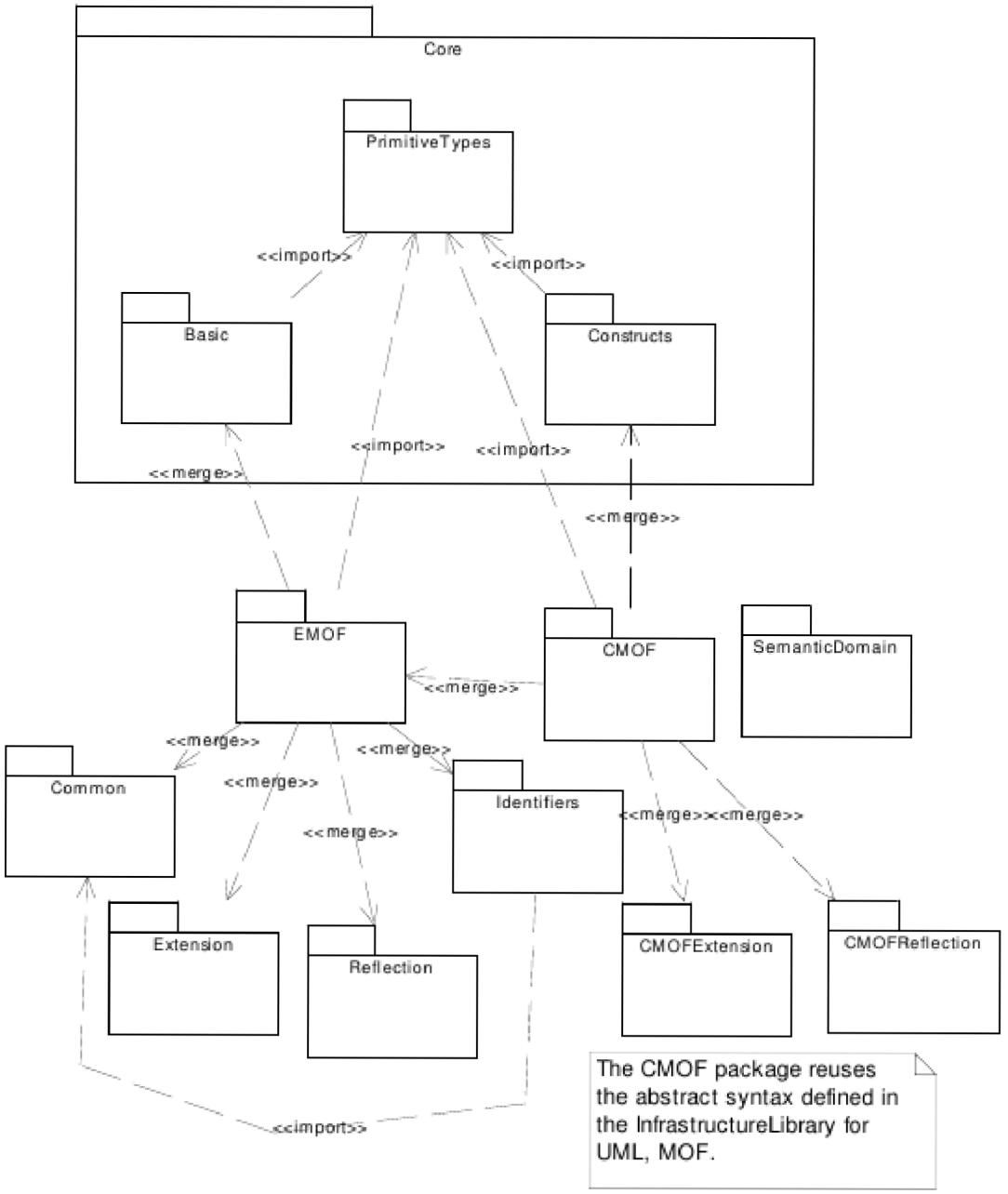
ArgoUML[6] je opensource modelovací nástroj, který sice podporuje tvorbu profilů a výměnu dat přes formát XMI, ale podporuje standard UML pouze do verze 1.4.

2.6.6 UML2 plug-ins for Eclipse

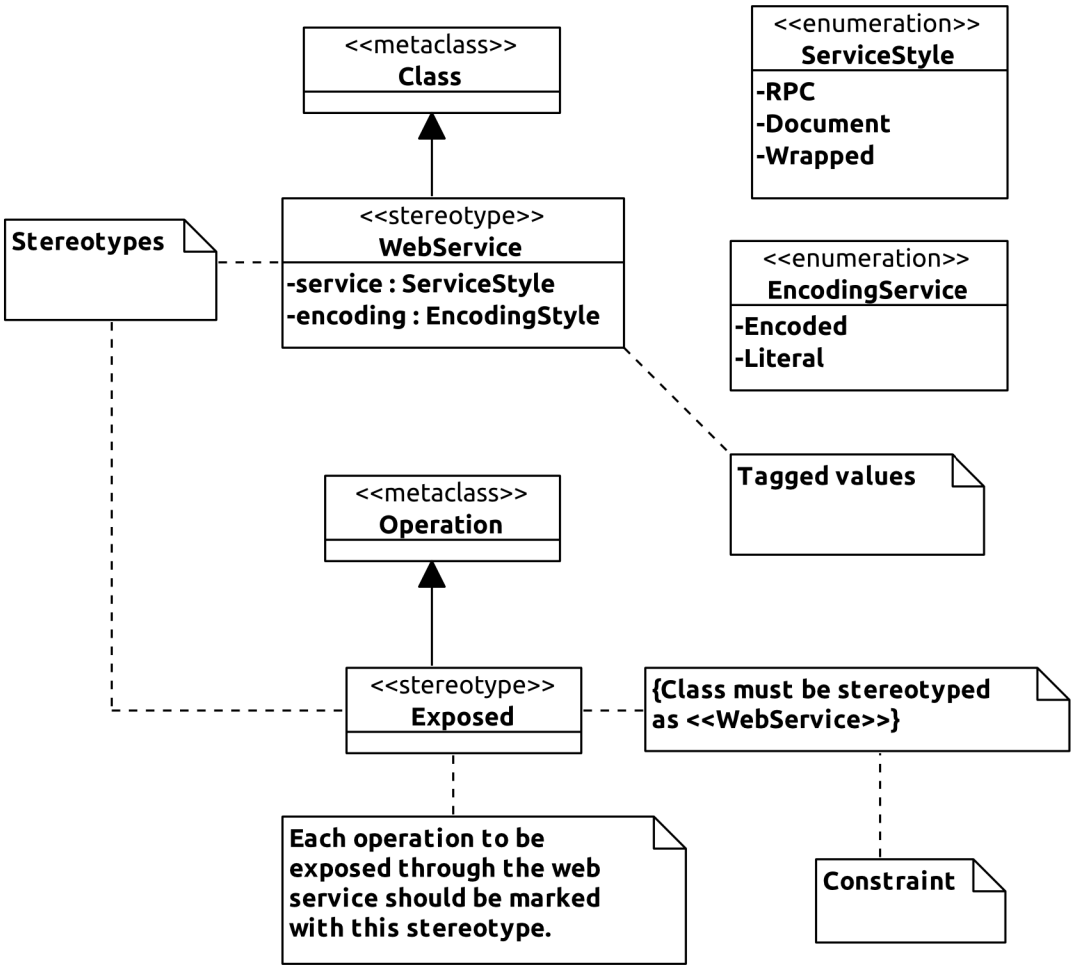
Eclipse UML2[7] je implementací UML 2.x meta-modelu pro platformu Eclipse ve formě softwarového doplňku pro IDE Eclipse.

Primárním cílem tohoto pluginu je podpora vývoje modelovacích nástrojů pro platformu Eclipse. To znamená, že poskytuje pouze meta-model, ale nikoliv modelovací nástroje samotné (k tomu ale lze použít doplněk *MDT-UML2Tools*[8]). Doplněk umožňuje vytváření a práci s UML profily. Vytváření UML profilu odpovídá psaní programu v jazyce Java, ale lze také použít průvodce a formuláře doplňku (ačkoliv se nejedná o vizuální vytváření doplňku ve smyslu programů Visual Paradigm, nebo Rational Rose Software Architect.

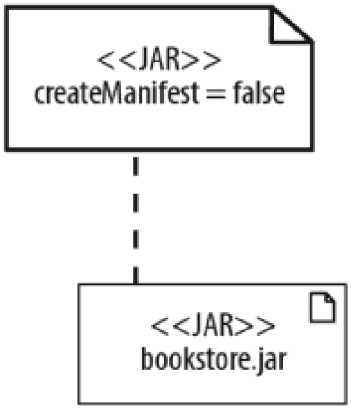
[5] http://www.papyrusuml.org/
[6] http://argouml.tigris.org/
[7] http://wiki.eclipse.org/MDT-UML2
[8] http://wiki.eclipse.org/MDT-UML2Tools



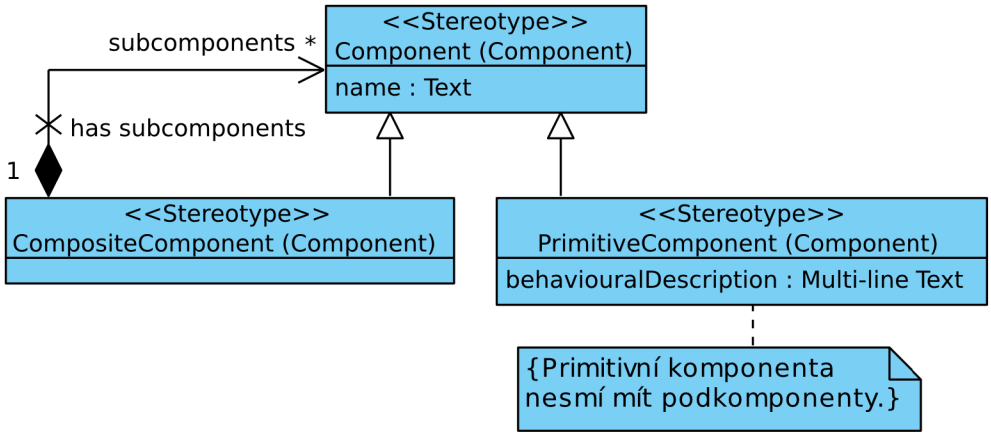
Obrázek 2.4: Vztah EMOF, CMOF a balíku *Core*, převzato z [18]



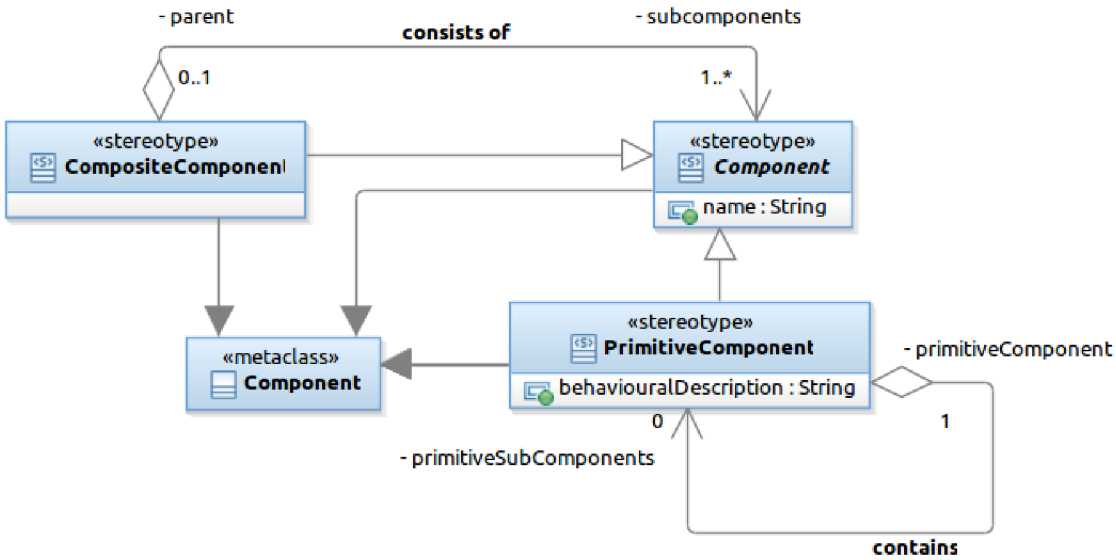
Obrázek 2.5: Ukázka vytvořeného UML profilu, převzato z [12]



Obrázek 2.6: Ukázka notace „tagged values“, převzato z [12]



Obrázek 2.7: Ukázka tvorby profilu v programu Visual Paradigm



Obrázek 2.8: Ukázka tvorby profilu v programu Rational Software Architect

Kapitola 3

Komponentové systémy

Tato část práce se věnuje jednomu z trendů současného softwarového inženýrství, kterému se předpovídá budoucnost a široké uplatnění v oboru informačních technologií. Jedná se o *Component Based Development (CBD)*, neboli vývoj založený na komponentách. Ve stručnosti se dá CBD definovat jako směr vývoje, který se zaměřuje na návrh a sestavení systému ze znovupoužitelných jednotek – komponent [23].

Někteří odborníci [24] uvádí, že komponentový vývoj uspěje tam, kde selhal vývoj založený na objektech a vypadá to, že vývoj v oboru výstavby (nejen) softwarových systémů jim dává za pravdu.

Investice (jak času, tak financí) vložené do vývoje univerzální komponenty, místo úzce specializované třídy, se vývojářům mohou vrátit v možnosti komponentu znova použít a případně také prodat na komponentových trzích. Zatímco o prodeji objektů se mluvilo, tak v praxi se tato praktika nepoužívá [24].

Důraz na snížení nákladů, tlak na vysokou produktivitu, spolehlivost a kvalitu při vývoji softwarových systémů způsobil, že CBD je stále více využíváno v komerčním prostředí [23].

V následující kapitole jsou popsány principy komponentových systémů. Kapitola 3.2 rozebírá typy softwarových architektur a kapitola následující pak některé, v průmyslu často využívané, komponentové systémy. Prostředky UML 2.0 pro modelování komponentových systémů popisuje kapitola 3.4. Poslední kapitola této části rozebírá meta-model komponentového systému, pro který bude v další části práce specifikován UML profil.

3.1 Principy komponentových systémů

Zásadním konceptem CBD je *komponenta*, která má význam [22] „prefabrikované“, znovupoužitelné jednotky s neznámou vnitřní strukturou, tzv. "black-box", komunikující zprávami přes své *přístupové body (access points)*.

Komponenty mají podle [24] následující vlastnosti:

- znovupoužitelné
- nezávislé na kontextu, tj. nemají zvenčí viditelný stav a lze je snadno vyměnit, nebo přesunout
- dají se skládat s jinými komponentami
- zapouzdřují chování

Při vývoji s využitím paradigmatu CBD procházíme třemi fázemi [8]:

1. výběr komponent (podle charakteristik výkonu, použitelnosti a spolehlivosti)
2. adaptace komponent do systému
3. složení (kompozice) komponent do funkčního systému

3.1.1 Rozdíly mezi CBD a OOP

CBD se označuje za evoluci objektově-orientovaného přístupu. Ačkoliv v některých charakteristikách jsou si tato paradigmata podobná, tak mezi nimi je několik rozdílů [22], [24]:

- objekt má vnitřní strukturu a zvenčí pozorovatelný stav, komponenta je „black-box“
- objekt je instancí konkrétní třídy, která je zasazena do hierarchie typů a to vše na úrovni jazyka, zatímco na komponentu je potřeba nahlížet z pohledu architektury a nabízí tak abstraktnější pohled
- komponenty mají vyšší kontextovou nezávislost
- znovupoužitelnost je na vyšší úrovni, než u objektů

I když by se mohlo zdát, že komponenty jsou trochu omezeným OOP a dříve, nebo později, na objekty dojde, není to pravda. Komponenty mohou být implementovány v jakémkoliv jazyce a jakoukoliv technikou. Jediná podmínka je bezstavovost [24].

3.1.2 Znovupoužitelnost komponent

Významným rysem komponent je jejich *opětovná použitelnost (reusability)*, což znamená, že je možné komponentu použít v jiném kontextu v rámci systému, nebo také v úplně jiném systému. Tato vlastnost je umožněna díky vhodnému návrhu komunikace komponent, kterým je dosažena izolovatelnost komponenty [24] od jejího okolí. Kvalitní návrh je potřebný také kvůli tomu, aby byl programátor schopen komponentu použít, aniž by znal její vnitřní strukturu. Tato vlastnost také usnadňuje udržovatelnost systému [21].

3.1.3 Komunikace komponent

Komponenty spolu komunikují zasíláním zpráv přes definovaná rozhraní, která mohou být vyžadovaná (*required*), nebo poskytovaná (*provided*) [22].

K popisu sémantiky rozhraní komponenty se používá tzv. *behavioural protocol*, k jehož specifikaci se dají použít např. regulární výrazy [21].

3.1.4 Hierarchická dekompozice komponent

Komponenty se mohou skládat do hierarchií. Kromě tzv. *jednoduché komponenty (primitive component)*, kterou tvoří pouze kód, tak v CBD existuje i *složená komponenta (composite component)* [21].

O složené komponentě říkáme, že je "grey-box". Její vnitřní strukturu tvoří jiné komponenty (a neobsahuje žádný kód „business“ logiky [21]), které se nazývají pod-komponenty a ty mohou být opět jednoduché, nebo složené [22]. Složená komponenta může obsahovat libovolný počet jiných komponent, včetně složených, ale jednoduchá komponenta již další pod-komponenty obsahovat nesmí. Komponenta také nemůže obsahovat sama sebe [21].

3.1.5 Propojení komponent

Vazby (Binding) mezi komponentami slouží k vyvolání služeb, které nabízí komponenta přes poskytované rozhraní. K odstínění [21] technických detailů komunikace komponent se používá abstrakce ve formě tzv. „connectors“. V komponentových modelech pak pro komunikaci přes konektory platí tři pravidla [21], [22]:

- vazba poskytovaného na vyžadované rozhraní pro propojení pod-komponent v rámci stejné mateřské složené komponenty
- vazba vyžadovaného rozhraní pod-komponenty a vyžadovaného rozhraní mateřské komponenty se používá k předávání požadavků pod-komponenty do vnějšího prostředí mateřské složené komponenty
- vazba požadovaného rozhraní složené komponenty a požadovaného rozhraní její pod-komponenty se používá k předání požadavků, které přijdou z prostředí mateřské složené komponenty a jsou určený její pod-komponentě

Právě aktuální vazba [22] komponent se nazývá *konfigurace (configuration)*. Možnosti propojení komponent se řídí také podle zvolené architektury.

3.2 Softwarová architektura

Architektura systému je nutným základem pro každý netriviální systém, konkrétně v případě komponentového systému nám architektura definuje vazby mezi komponentami, jak vypadá prostředí, nebo jaké role mohou komponenty zastávat [24].

Na architekturu softwarového systému můžeme nahlížet různými pohledy [22]:

- pohled popisující logickou strukturu systému; dá se dobře popsat diagramem
- pohled na chování systému popisuje komponentu jako proces, popisuje změny systému v čase, případně komunikaci komponent; k popisu je často použit formální jazyk
- pohled na fyzickou architekturu systému; k popisu fyzického rozmístění komponent je vhodný diagram rozmístění (*Deployment Diagram*) [2]

Podle schopnosti architektury změnit svou logickou strukturu, je můžeme dále rozdělit [22] na:

- *statickou* – od startu systému existuje pouze jediná konfigurace, tzn. nemění se v čase a po startu není možné komponentu přidat, ani odebrat
- *dynamickou* –umožňuje rekonfiguraci za běhu, takže komponenty mohou vznikat a zanikat podle definovaného chování, stejně tak propojení komponent
- *mobilní* – oproti dynamické architektuře mohou komponenty měnit kontext v logické struktuře systému i během provádění operace

3.3 Existující komponentové modely

Komponentový model [22] definuje syntaxi, sémantiku a pravidla kompozice komponent, konkrétní softwarové architektury podporující vývoj založený na komponentách (CBD). Podle architektury systému mohou také definovat pravidla pro chování a podporu mobility.

Kromě známých a v praxi používaných komponentových modelů, které se ujaly jako průmyslové standardy, se tato kapitola věnuje také modelům s formální bází.

3.3.1 Průmyslové komponentové modely

V této kapitole jsou stručně popsány základní principy známých komponentových systémů, které se staly průmyslovými standardy.

CORBA

CORBA (Common Object Request Broker Architecture) [16] je otevřeným standardem, spravovaným konsorciem OMG. Jak se dá poznat z názvu, CORBA poskytuje architekturou a infrastrukturu (*middleware*), pro aplikace pracující v distribuovaném prostředí nabízí služby *RPC (Remote Procedure Calling; vzdálené volání procedur)*. CORBA aplikace jsou nezávislé na konkrétní HW architektuře, operačním systému, programovacím jazyku a typu sítě. Pokud implementace různých dodavatelů CORBA řešení implementují správně standard, tak je také zaručena funkční komunikace mezi CORBA implementacemi různých dodavatelů. Jedná se o velmi robustní řešení, ale přitom velmi dobře škálovatelné a flexibilní řešení (specializované verze CORBA middleware běží na real-time a malých embedded systémech).

Aplikace jsou skládány z objektů. Pro každý takový objekt je nutné definovat rozhraní v jazyce *OMG IDL (Interface Definition Language)*. Základním konceptem architektury CORBA je oddělení implementace a rozhraní, k čemuž používá jazyk IDL a který je také klíčem k nezávislosti na platformě a implementačním jazyku. Klientský objekt použije stejné rozhraní ke komunikaci s objektem, jehož metody chce volat.

Definice rozhraní je nezávislá na programovacím jazyce a OMG poskytuje mapování z IDL do jazyků C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python a IDLscript.

Komponentový model CORBy (*CORBA Component Model, CCM*) je založen na objektech z objektového modelu CORBy a je součástí specifikace CORBA od verze 3.0 [17].

Specifikace definuje komponentový model CORBy jako plochou strukturu, takže není podporována hierarchická kompozice.

Specifikace také definuje rozšíření jazyka IDL – *Component IDL (Component Interface Definition Language, CIDL)* – pro popis rozhraní a přidává k němu vlastnosti specifické pro CBD. CIDL definuje dva základní typy komponent:

- základní *basic component* – pouze zapouzdřuje objekty původního objektového modelu, komponenty nemohou mít rozhraní
- rozšířená *extended component* – plně podporují paradigma CBD, nabízí rozhraní pro synchronní (poskytovaná rozhraní se nazývají *facets*, vyžadovaná rozhraní *receptacles*) i asynchronní *podobně* event sources *a* event sinks volání

Corba nabízí také *Component Implementation Framework (CIF)*, framework pro implementaci komponent. Sestavení probíhá vygenerováním, tzv. *skeleton* – zdrojový kód komponenty, který obsahuje vše potřebné pro navázání komunikace, programátor musí pouze implementovat těla metod. Je zachována nezávislost na jazyku, protože kód v konkrétním

programovacím jazyce se vygeneruje z popisu komponenty v CIDL (který je mapován na „objektový“ IDL a ten pak na konkrétní jazyk).

Komponentový model CORBy podporuje vytváření statické, tak i dynamické architektury komponent (kapitola 3.2).

Rodina COM a .NET

Rodina technologií *COM (Component Object Model)* [10] a .NET [11] jsou známé komponentové technologie dostupné na operačních systémech Microsoft Windows. Technologie se často používá k vytváření uživatelských desktop programů.

Microsoft COM (Component Object Model) [10] je komponentová technologie, která není omezena pouze na operační systémy Windows, ale v praxi se na jiných platformách příliš nepoužívá. Podporuje běh a komunikaci komponent v rámci jednoho počítače. Toto omezení překonává model *DCOM (Distributed COM*, který přidává možnost běhu komponent v síti.

COM podporuje mnoho programovacích jazyků, umožňuje využívat služeb systému Windows, vytvářet komponenty a sestavit na jejich základě aplikace. Další verzí COM je COM+ přidává podporu transakcí (*Microsoft Transaction Server*), asynchronní volání (*Message Queue Server*), bezpečnostní služby a další.

COM k volání metod používá *Object Remote Procedure Call (ORPC)*, k definici rozhraní je pak možné použít *Microsoft IDL (MIDL, je to rozšíření CORBA IDL)*. Komponentový model COM umožňuje komponentám implementovat pouze poskytovaná rozhraní.

Hierarchická kompozice je definována na úrovni zdrojového kódu a je dvojího typu:

- *containment* – rodičovská komponenta znova implementuje část poskytovaných rozhraní svých pod-komponent, které pak předávají volání na danou pod-komponentu
- *aggregation* – za účelem volání je klientské komponentě vrácena reference na rozhraní a volání probíhá přímo

Komponenty se musí registrovat v systémových registrech, kde dostanou přidělený unikátní identifikátor. Poté jsou komponenty sdíleny v rámci systému, což může přinášet problémy s kompatibilitou v případě nahrazení komponenty novou verzí [21].

.NET

.NET [11] je komponentovým frameworkem společnosti Microsoft. Hlavním rozdílem oproti COM je způsob vytváření komponent a jejich rozmístění v prostředí. Některé vlastnosti naopak sdílejí. Stejně jako COM, je často používána k vytváření uživatelských aplikací s „bohatým“ uživatelským rozhraním.

.Net Framework se skládá z částí [11]:

- *Common Language Runtime* — poskytuje abstrakci nad operačním systémem
- *Base Class Libraries* -– předem sestavený kód pro účely nízkoúrovňových úloh
- *Development frameworks and technologies* – poskytuje podporu pro složité a komplexní úlohy

.NET komponenty jsou překládány [21] do byte-kódu z jazyka *Microsoft Intermediate Language (MSIL)* a interpretovány virtuálním strojem *Common Language Runtime (CLR)*.

Komponenty [21] se v terminologii .NET nazývají *assemblies* a jsou složeny z kompilovaných tříd a *manifest* souboru, který popisuje závislosti komponenty a používané datové

typy a poskytovaná rozhraní, stejně jako u COM nepodporují požadovaná rozhraní a je potřeba to vyřešit v kódu. Hierarchická kompozice je frameworkem .NET podporována pouze na úrovni zdrojových kódů.

Enterprise Java Beans

Technologie *Enterprise Java Beans (EJB)* [5] komponentová architektura orientovaná na použití *client-server*. Technologie umožňuje rychlý vývoj přenositelných distribuovaných aplikací a nabízí např. prostředky usnadňující transakční zpracování, zabezpečení aplikace. Specifikace EJB lze nalézt ve zdroji [5].

Komponenty EJB vyžadují použití jazyka Java, v terminologii EJB se jim říká *Beans* a existují tři druhy:

- *Entity Bean* – komponenta reprezentuje persistentní data v databázi, nebo jiném datovém úložišti
- *Session Bean* – reprezentuje *business objekt*, volání je synchronní a komponenty tohoto typu mohou být stavové (uchovávají si svůj stav mezi jednotlivým voláním), nebo bezstavové
- Message-driven Bean – komponenty s asynchronním voláním

Komponenty spolu komunikují vlastní variantou RPC[1] – *Remote Method Invocation* a mají dva typy rozhraní: *vzdálené (remote interface)*, které konceptem odpovídá poskytovanému rozhraní a *domácí (home interface)*, což je rozhraní určeno jiným komponentám pro vytváření a vyhledávání komponent, je poskytováno aplikačním serverem, na kterém aplikace běží.

3.3.2 Formální komponentové systémy

Od průmyslových komponentových systémů se liší původem vzniku, kterým je často akademická půda, implementace vlastností, které v konzervativních komerčních systémech zatím nejsou a hlavně formálním základem.

Zástupci této kategorie komponentových systémů jsou např. Fractal [4], nebo SOFA 2 [26]. Úplný popis meta-modelu těchto systémů lze nalézt v jejich dokumentacích. Z těchto dokumentací v následujícím stručném popisu vycházím.

Fractal

Fractal je „modulární, rozšiřitelný a na programovacím jazyce nezávislý komponentový model“. Mezi významné vlastnosti Fractalu patří rekurzivní kompozice, sdílení komponent, obsahuje jediný abstraktní typ propojení komponent zapouzdřující různé typy komunikace (synchronní volání metod, RPC, apod.) a další [4].

Komponenta Fractalu má z vnějšího pohledu viditelné pouze externí rozhraní (*external interfaces*). Každé rozhraní musí mít jedinečné jméno v rámci jedné komponenty. Požadované rozhraní se v terminologii Fractalu nazývá *client*, poskytované *server*. Komponenta s viditelným obsahem je složená (*composite component*), pokud obsah není vidět a komponenta má alespoň jedno rozhraní tak mluvíme o primitivní komponentě (*primitive component*). Komponenta bez řídících rozhraní se nazývá základní komponenta (*base component*). Komponenty mohou být sdíleny v *obsahu* jiných komponent.

[1] „Remote Procedure Call – vzdálené volání procedur“

Controller komponenty může mít externí (dostupné z vnějšího prostředí komponenty) i interní rozhraní (dostupné pouze pod-komponentám). Rozhraní může být funkcionální (*functional*, odpovídá funkčním aspektům komponenty) a řídící (*control*, řídící aspekty, např. konfigurace, rekonfigurace).

Propojení komponent za účelem komunikace se nazývá *binding*. Spojení může byt *jednoduché (primitive bindings)*, což představuje spojení mezi jedním rozhraním typu *client* a jedním rozhraním typu *server*; druhým typem spojení je *složené (composite bindings)*, které spojuje obecně libovolný počet rozhraní. Na *bindings* jsou kladena další omezení, jejichž popis by překročil rozsah tohoto přehledu, omezení jsou popsána v [4].

Použití jazyka pro definici rozhraní (*Interface Definition Language*) je ve Fractalu možné (spolu s mapováním na existující programovací jazyky), ale ne nutné a rozhraní mohou být definována přímo v jakémkoliv programovacím jazyce.

SOFA

SOFA 2 je nástupcem komponentového modelu SOFA. Využívá hierarchickou kompozici komponent s podporou dynamické architektury, mnoha způsobů komunikace, verifikaci kompozice i chování, nebo oddělení řídící a výkonné (*business*) logiky komponenty a další [26].

Komponentový model rozlišuje tři základní části komponenty: *frame (rámec)*, který určuje hranici komponenty, *řídící část (control part)* a *obsah komponenty (component content)*. Řídící část je zprostředkována prostředím SOFA 2, obsah komponenty může být tvořen další komponentou (respektive komponentami), nebo přímo kódem business logiky.

Rozhraní může být *řídící (control)*, nebo *business* (to dále může být poskytované a požadované). Business rozhraní slouží pro komunikaci komponent. Komponenty mohou být propojeny pouze přes rozhraní kompatibilních typů. Přes řídící rozhraní lze přistupovat k řídící části komponenty.

Každá komponenta musí implementovat svůj rámec (*Frame*), což je kolekce poskytovaných a požadovaných business rozhraní. Veškerá komunikace musí procházet přes tato rozhraní, včetně komunikace určené pro vnitřní pod-komponenty. Komponenty, které implementují stejný rámec jsou zaměnitelné.

Kód *jednoduché (primitive component)* komponenty je uložen v centrálním repositáři (*Repository*). Pokud komponenta obsahuje pod-komponenty, jedná se o *složenou (composite component)* komponentu. Pod-komponenty mohou implementovat poskytované rozhraní rodičovské komponenty, jejich spojení je pak realizováno propojením typu *presumption* (spojení poskytovaného rozhraní rodičovské komponenty a poskytovaného rozhraní pod-komponenty), aby bylo zajištěno správné volání metod. Pokud potřebuje pod-komponenta volat metody prostředí rodičovské komponenty, tak musí realizovat spojení typu *delegation* (spojení požadovaného rozhraní pod-komponenty, stejné rozhraní musí mít i rodičovská komponenta).

3.4 Modelování komponentových systémů v UML 2.0

Komponentové diagramy v UML popisují logickou strukturu architektury (kapitola 3.2) komponentového systému a vývojářům zprostředkovávají tento pohled na systém. Jsou velmi vhodné pro popis systémů převážně se statickou architekturou: neobsahuje prostředky pro popis všech prvků softwarových architektur (dynamické, mobilní architektury) [22].

Následující podkapitoly popisují koncepty komponentových diagramů v jazyce UML 2.0, které jsou významné pro účely této práce, pro kompletní výčet vlastností odkazuji

na specifikaci *UML Superstructure* [15], ze které v následujícím textu čerpám, pokud není uvedeno jinak.

3.4.1 Modelování komponent a jejich hierarchická kompozice a dekompozice (Components)

Komponenta meta-modelu UML odpovídá konceptu komponenty, tak jak je chápán metodologií CBD (viz. předchozí kapitola). V meta-modelu UML je definována meta-třídou `Component` z balíku *Components*.

Balík *Components* nabízí podporu modelování komponent (logických i fyzických; logická komponenta může být např. komponenta business procesu, fyzickou se rozumí komponenta konkrétního systému, např. komponenta v systému CORBA) a také podporu pro manifestaci komponent (*artifact*) při jejich rozmístění ve fyzickém prostřední (*nodes*).

Různé aspekty komponenty jsou odděleny do dvou pod-balíků: *BasicComponents* a *PackagingComponents*. První z nich definuje komponentu jako spustitelnou jednotku systému a zaměřuje se na její chování, ten druhý se zabývá komponentou jako stavebním prvkem a umožňuje existenci složených komponent.

`Component` je specializací meta-třídy `StructuredClasses::Class` (zápis znamená meta-třída `Class` z balíku *StructuredClasses*). Díky tomu obsahuje všechny vlastnosti obecnější meta-třídy `Kernel::Class`.

Meta-třída `Class` z balík *StructuredClasses* přidává UML komponentě možnost mít vnitřní strukturu a porty. Pokud by meta-třída UML dědila pouze od `Classes::Class`, tak by nemohla mít vnitřní části. Tyto vnitřní části se nazývají *Parts* (specializace meta-třídy `Property`) a mají název a typ [21]. Definice vnitřní struktury je v balíku *InternalStructure*, ale její použití v modelu je nepovinné, protože vnitřní struktura je pro ostatní komponenty nepodstatná a tento koncept slouží především pro vývojáře, který tak může upřesnit chování komponenty.

Hierarchickou strukturu komponenty lze modelovat dvěma způsoby [21]:

- pomocí meta-třídy `Property` s využitím vlastností, které UML komponentě dává meta-třída `StructuredClasses::Class`
- pomocí meta-třídy `Kernel::PackageableElement`

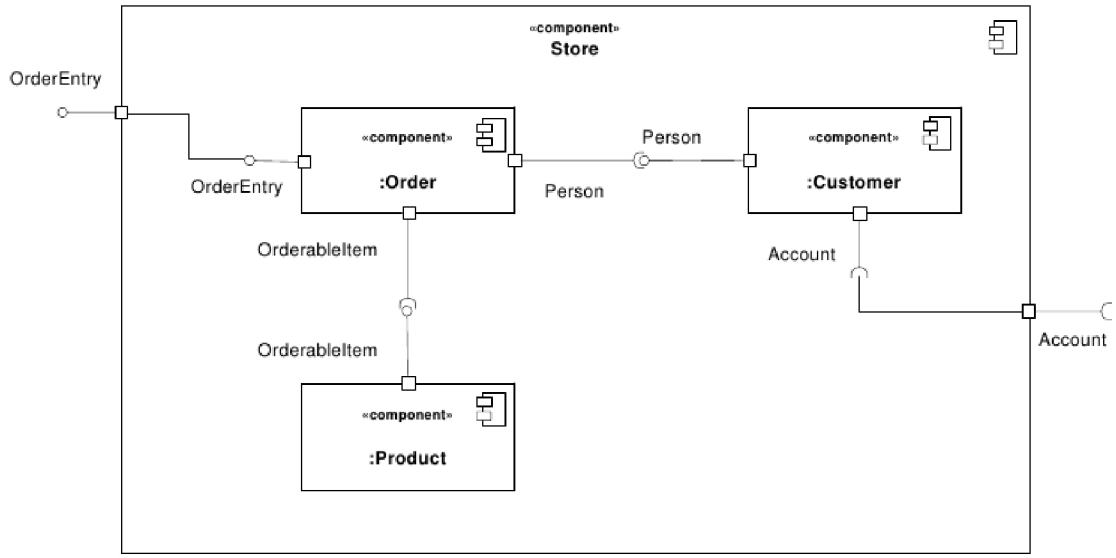
Pod-komponenty pomocí meta-třídy `Property`

Tato metoda [21] využívá faktu, že vnitřní strukturu komponenty lze popsat pomocí vnitřní struktury. Pod-komponenty jsou modelovány jako *parts*, což je specializace meta-třídy `Property`. To umožňuje přiřadit pod-komponentě násobnost. *Property* lze modelovat jako referenci na skutečnou instanci meta-třídy *Property* nebo skutečnou instanci subkomponenty určit typem elementu *part*.

Ačkoliv je tímto způsobem možné modelovat vnitřní strukturu složené komponenty, má mnoho omezení. Největším omezením je, že se komponenta jako taková musí definovat mimo a *part* obsahuje pouze referenci. Proto jsem si vybral druhou možnost, kde tyto problémy neexistují. Zájemce o modelování komponent pomocí `Property` odkazuji na podrobný popis [21].

Pod-komponenty pomocí meta-třídy `Kernel::PackageableElement`

Meta-třídy `Kernel::PackageableElement` je základní třídou pro mnoho dalších meta-tříd UML, včetně `Class`, `Component` a `Interface`. Rozšíření `PackageableElement` umožňuje elementům mít pod-elementy a velmi snadno tak vytvářet hierarchie s libovolnou hloubkou v jednom místě [21]. Ukázka takové komponenty s hloubkou 1, je na obrázku 3.1.



Obrázek 3.1: Ukázka složené komponenty v UML, převzato z [15]

3.4.2 Rozhraní (Interfaces)

Komponenty komunikují s ostatními komponentami zasíláním zpráv, které vyvolají v přijímající komponentě nějakou akci. Služby, které komponenta nabízí, jsou definovány v rozhraní, které komponenta poskytuje *provided interface*. Naopak služby, které komponenta může využít, nebo vyžaduje, definuje požadované rozhraní *required interface*.

Komponenta může poskytované rozhraní implementovat přímo („lolipop“ notace, nebo tzv. „whitebox“ notace), nebo může vztahy s rozhraním deklarovat explicitně.

Při explicitní deklaraci vztahu komponenty a poskytovaného rozhraní je tento vztah reprezentován realizací, grafická podoba je v tabulce 3.1.

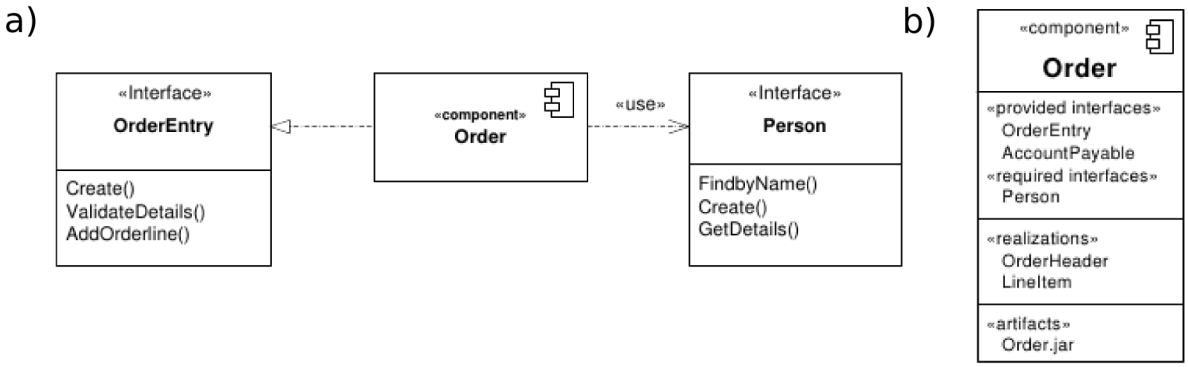
Obdobné to je i v případě požadovaného rozhraní, to může být alternativně reprezentováno vztahem závislosti (*Dependency*), který je označena stereotypem «`use`». Všechny tři možnosti jsou na obrázku

Explicitní určení vztahů umožňuje volitelně zobrazit také operace, ukázka je na obrázku 3.2.

Kromě „blackbox“ pohledu a „graybox“ pohledu na komponentu existuje také „whitebox“. Tento pohled zobrazuje, v tzv. odděleních (*Compartments*), poskytované a požadované rozhraní, artefakty, operace, vztahy, atd.; ukázka je na obrázku 3.2.

Grafická notace komponent realizujících rozhraní jsou v tabulce 3.1.

Poskytovaná a požadovaná rozhraní mohou být organizována do skupin služeb pomocí portů; viz. 3.4.3.



Obrázek 3.2: a) explicitní definice požadovaných a poskytovaných rozhraní, b) „whitebox“ pohled na komponentu, převzato z [15]

3.4.3 Porty (Ports)

Meta-třída *Port* z balíku *Ports* slouží jako bod, ve kterém se sdružuje (a prochází jím) logicky související komunikace a slouží pro komunikaci prostředí s komponentou, nebo komponenty s její vnitřní strukturou. Port [21] má jméno, typ a násobnost (určuje, kolik instancí portu bude vytvořeno uvnitř komponenty, která port vlastní).

Port dokáže předávat požadavky přicházející z prostředí do vnitřní struktury komponenty.

3.4.4 Konektory (Connectors)

Konektory (Connectors) modelují [21] komunikační linky. Propojují poskytované s požadovaným rozhraním (a obráceně).

Konektory mohou být připojeny:

- přímo mezi komponentou a rozhraním
- pokud jsou používány porty, tak mezi rozhraním a portem
- nebo mezi rodičovskou komponentou a její pod-komponentou

UML 2 specifikuje konektory: „assembly“ a „delegate“.

„Assembly“ konektor

Konektor spojuje komponentu (nebo její port) s požadovaným rozhraním a komponentu (nebo její port) s jejím poskytovaným rozhraním. Konektor je v tabulce 3.1. Grafická notace tohoto typu konektoru odpovídá stavu, kdy „socket“ (požadované rozhraní) a „lolipop“ (poskytované rozhraní) jsou v sobě těsně „zasunuty“ ([21], [22]). Pokud je nelze graficky spojit dohromady (například program Rational Software Architect to neumožňuje), je nutné použít vztah závislosti (*Dependency*, viz. tabulka 3.1 a část 3.4.5, volitelně se stereotypem «use»).

„Delegate“ konektor

Konektor *delegate* spojuje rozhraní stejného typu jedné komponenty (poskytované s poskytovaným, požadované s požadovaným). Slouží k předávání zpráv buď z prostředí k pod-komponentám složené komponenty, nebo k delegaci požadovaného rozhraní pod-komponenty směrem do prostředí její rodičovské komponenty. Stejně jako u *assembly* konektoru je možné propojit komponentu přímo, nebo přes port. K modelování této vazby se používá asociace s otevřenou šipkou na konci a se stereotypem «`delegate`» ([21], [22]).

3.4.5 Vazby (Relationships)

Používané vazby (přehled v tabulce 3.1) zahrnují:

- *asociace* (ve významu vazby «`delegate`») a *„assembly“ konektor*, viz. konektory v kapitole 3.4.4
- *generalizace* – mezi komponentami (nebo rozhraním) může existovat vztah generalizace-specializace, speciální komponenta dědí atributy, metody a poskytovaná, resp. požadovaná, rozhraní své bázové třídy
- *realizace* – užívá se k vytvoření vazby s rozhraním, které specifikuje poskytované, resp. požadované, rozhraní komponenty
- *závislost* – s různými stereotypy modelují závislostní vazby mezi komponentami, nebo mezi komponentou a rozhraním, mezi artefakty (kapitola 3.4.6) a komponentou, resp. uzlem; vazba má sémantiku pouze pro elementy modelu, nikoliv pro jejich instance a tedy i směr vazby určuje pouze sémantiku závislosti požadovaného na poskytovaném rozhraní a nikoliv směr toku zpráv [15]

3.4.6 Rozmístění (Deployment)

Rozmístění na fyzické uzly se v UML modeluje diagramem rozmístění (*Deployment diagrams*). Rozmístění je realizováno přiřazením klasifikátoru *artifact* konceptu, který se nazývá uzel (*Node*). *Node* reprezentuje fyzický uzel, například konkrétní server v síti. Pokud je potřeba přesněji specifikovat hardwarové a softwarové prostředí, je možné meta-model standardními způsoby rozšířit.

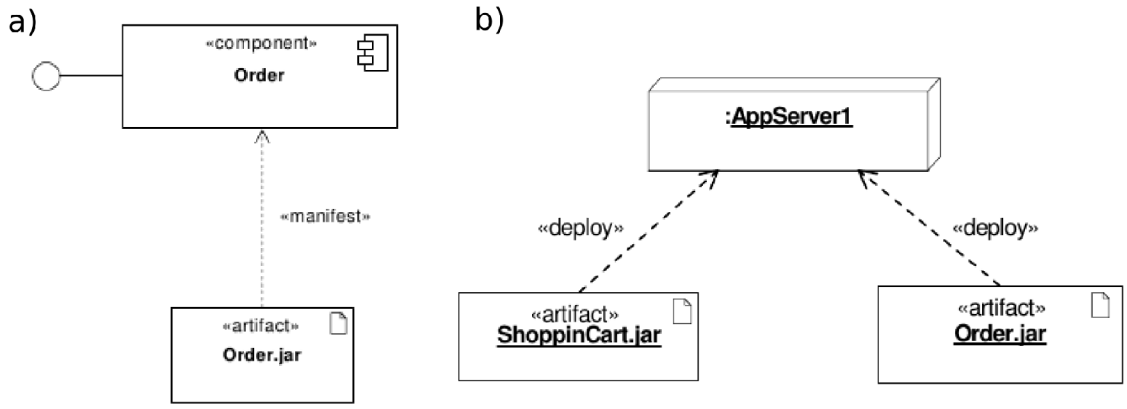
Komponenty [3] v UML 2 jsou logické jednotky návrhu a proto nelze přímo modelovat jejich umístění na fyzických zařízeních. K tomu se v UML používá koncept artefaktů (*Artifact*) a uzlů (*Nodes*).

S uzly [3] se mohou vázat pouze artefakty, ty představují fyzické jednotky (databáze, binární spustitelné soubory, skripty, atd.). Artefakt pak může „manifestovat“ závislost (vztah *dependency*) na komponentě (obrázek 3.3). V dalším kroku se vytvořením závislosti mezi artefaktem a uzlem určí, na kterém uzlu bude artefakt umístěn (*deploy*).

Na obrázku 3.3 je příklad artefaktu *Order.jar*, který má vztah závislosti (stereotyp «`manifest`») na komponentě v diagramu komponent. Přiřazení artefaktu konkrétnímu uzlu se provede v diagramu rozmístění. Relace závislosti je označena stereotypem «`deploy`».

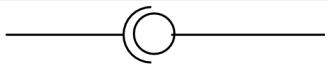
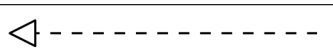
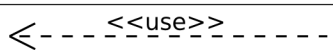
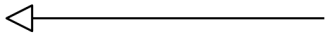
3.4.7 Grafická notace

Mezi vývojáři široce známá a používaná je tzv. „lolipop“ notace (kvůli podobnosti zápisu poskytovaného rozhraní s lízátkem). Ukázky této notace jsou v tabulce 3.1.



Obrázek 3.3: a) manifestace komponenty artefaktem, b) umístění artefaktu na uzel, převzato z [15]

Lze se setkat i s alternativní notací, která používá běžné diagramy tříd a patřičným stereotypem označené klasifikátory. Tuto alternativní notaci ve výchozím nastavení používá např. Rational Software Architect (2.6.2). Komponenty v této notaci poskytují tzv. „white-box“ pohled na komponentu. V oddílech (*compartments*) uvnitř komponenty lze zobrazit rozhraní, která komponenta realizuje a požaduje, realizace samotné komponenty, atributy, operace, vnitřní strukturu tak, jak ji lze modelovat diagramem vnitřní struktury (kapitola 3.4.1) a atributy přiřazených stereotypů. Tento pohled neumožňuje použití portů.

Typ elementu	**Notace**
Komponenta *Component*	
Komponenta implementující rozhraní (*provided*)	
Komponenta poskytující port (*Port*) s typem poskytovaného rozhraní	
Komponenta užívající («`use`») rozhraní (*requested*)	
Komponenta užívající («`use`») port s typem požadovaného rozhraní (*provided*)	
Konektor typu *assembly*	
Vazba realizace (*realization*)	
Vazba delegace (*delegate*)	
Vazba závislosti (*dependency*)	
Vazba generalizace (*generalization*)	

Tabulka 3.1: Přehled grafické notace komponentových diagramů v UML, [15]

Kapitola 4

Tvorba UML Profilu

V předchozích kapitolách byly probrány základy jazyka UML, popsány možnosti rozšíření meta-modelu UML o další sémantiku, základy komponentových systémů a na komponentách založeném vývoji.

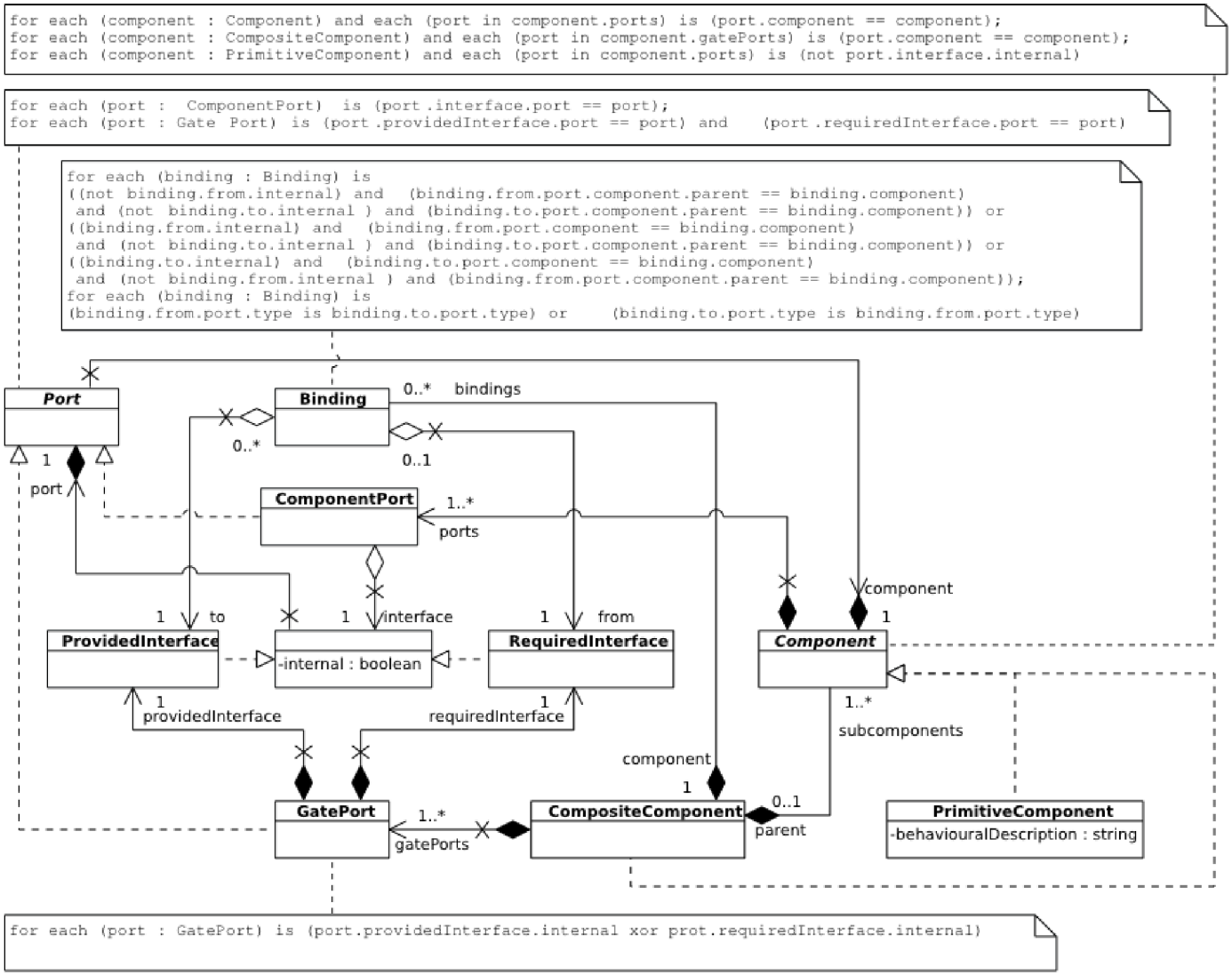
V části práce je popsán meta-model komponentového systému, pro který budu dále v této části vytvářet UML profil.

4.1 Meta-model komponentového systému

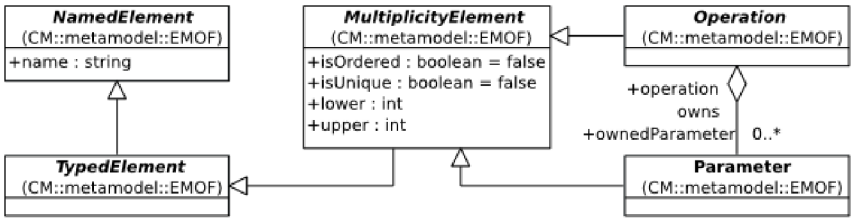
Meta-model komponentového systému, popisovaný v této kapitole, je zjednodušenou verzí meta-modelu navrženém v disertační práci [22]. Při popisu této verze částečně vycházím z původní práce.

Meta-model je popsán diagramem tříd jazyka UML 2 (kapitola 4.1) a s využitím části meta-modelu EMOF (kapitola 2.3.1), jehož relevantní část zachycuje diagram na obrázku 4.2. Od tříd EMOF: `EMOF::NamedElement`, `EMOF::TypedElement` a `EMOF::Operation`, jsou odvozeny třídy meta-modelu komponentového modelu. Sémantika těchto tříd je podrobněji popsána v dokumentu [14], zde uvádím pouze stručný popis.

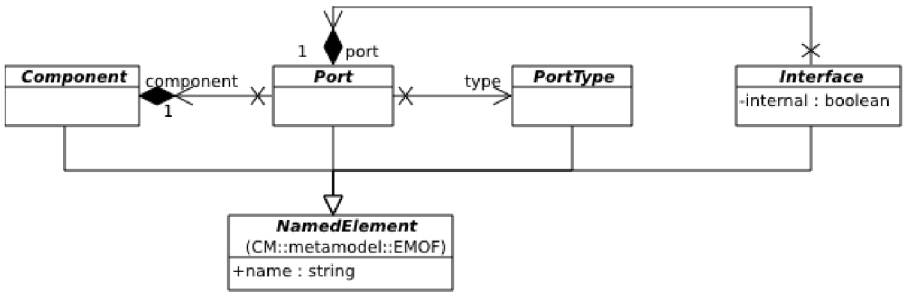
- `EMOF::NamedElement` – element, který rozšiřuje tuto meta-třídu se dá v modelu pojmenovat
- `EMOF::TypedElement` – je specializací *NamedElement*, od které získává schopnost mít jméno a sám představuje v modelu elementy, které mohou mít typ
- `EMOF::MultiplicityElement` – reprezentuje kardinalitu elementu: interval kladných celých čísel se spodní a horní mezí (může ležet v nekonečnu) intervalu
- `EMOF::Operation` – představuje operaci, kterou je možné spustit v instanci třídy, která ji obsahuje. Operace má typ a obsahuje parametry
- `EMOF::Parameter` – parametr je element s typem, reprezentuje parametr operace. Přes parametry je možné předávat operaci hodnoty



Obrázek 4.1: Komponentový meta-model – hlavní část meta-modelu komponentového systému



Obrázek 4.2: Komponentový meta-model – použité meta-třídy EMOF



Obrázek 4.3: Komponentový meta-model – třídy dědící od `EMOF::NamedElement`

4.1.1 Komponenty

Meta-model definuje abstraktní komponentu `Component` jako specializaci `EMOF::NamedElement` (obrázek 4.3) a dále dvě realizace této abstraktní komponenty: *jednoduchou komponentu* (`PrimitiveComponent`) a *složenou komponentu* (`CompositeComponent`).

Jednoduchá komponenta má, ve shodě s principy komponentových systémů, neznámou vnitřní strukturu („black-box“), je implementována přímo, nemá z venčí pozorovatelný stav, ale implementuje chování, které je popsáno v atributu `behaviouralDescription`.

Složená komponenta má vnitřní strukturu („grey-box“) tvořenou dalšími realizacemi abstraktní komponenty `Component`, které se označují jako pod-komponenty.

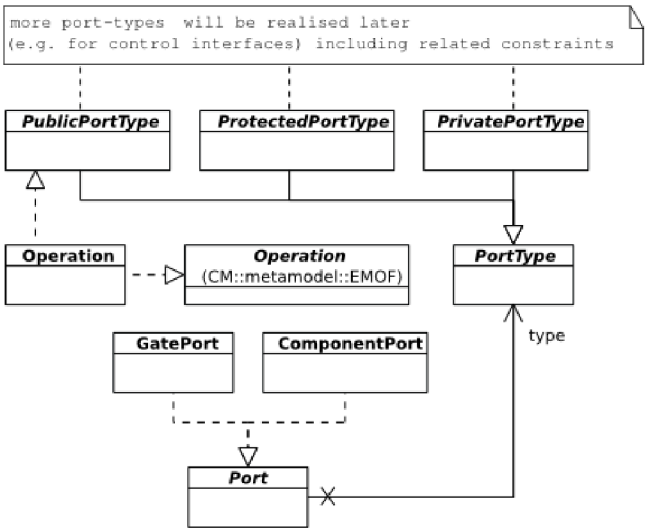
4.1.2 Porty

Abstraktní třída `Port` (specializace `EMOF::NamedElement`) má v meta-modelu význam bodu, přes který probíhá komunikace komponent. V závislosti na typu komponenty se používají dvě realizace třídy `Port`: `ComponentPort` a `GatePort` (obrázek 4.4). Každá instance třídy `Port` má svůj typ.

`ComponentPort` může používat jak jednoduchá, tak i složená komponenta (pouze jako vnitřní port), zatímco `GatePort` používá pouze složená komponenta pro předávání komunikace od svých pod-komponent do svého prostředí. K předávání těchto zpráv se využívá konceptu rozhraní (viz. podkapitola 4.1.4).

4.1.3 Typ portu

Typ portu definuje abstraktní komponenta `PortType` (specializace `EMOF::NamedElement`) se třemi abstraktními specializacemi: `PublicPortType`, `ProtectedPortType`, `PrivatePortType` (obrázek 4.4). Meta-model v této chvíli definuje pouze jedinou realizaci typu portu – typ `Operation`, která je realizací abstraktního typu `PublicPortType`.



Obrázek 4.4: Komponentový meta-model – typy portu

Třída `Operation` pak zároveň realizuje operaci podle meta-třídy `EMOF::Operation`. Představuje „business“ službu komponenty s parametry.

4.1.4 Rozhraní

Rozhraní je v meta-modelu reprezentováno abstraktní třídou `Interface` se dvěma realizacemi reprezentující poskytované (`ProvidedInterface`) a požadované (`RequiredInterface`) rozhraní. `Interface` je specializací `EMOF::NamedElement`.

Podle typu portu může port vlastnit buď jen poskytované, nebo požadované rozhraní v případě `ComponentPortu`, nebo poskytované i požadované rozhraní v případě `GatePort`, kdy jedno z nich musí být interní (atribut `internal`). Interní rozhraní je poskytováno složenou komponentou svým pod-komponentám.

4.1.5 Vztahy

Třída `Binding` reprezentuje dva specifické typy vazeb mezi rozhraním komponent:

- spojení vnitřního rozhraní portu `GatePort` složené komponenty s vnějším rozhraním jejich pod-komponent
- spojení vnějších rozhraní pod-komponent stejné složené komponenty

4.1.6 Změny vůči původnímu meta-modelu

Zde popsaný meta-model je oproti původnímu meta-modelu, definovanému v [22], jednodušší. Aby se dal komponentový systém modelovat komponentovým diagramem jazyka UML, bylo nutné meta-model opravit. Úpravy meta-modelu provedl autor původního meta-modelu ([22]) a současně vedoucí této práce. Hlavní principiální rozdíl spočívá v tom, že v původním meta-modelu informaci o typu obsahovalo rozhraní. Nyní tuto informaci obsahuje port, což koresponduje s k komponentovým diagramem v UML.

Port má v původním i tomto upraveném meta-modelu významnější roli, než v běžném komponentovém diagramu (všechna rozhraní jsou napojena na porty, žádné není realizováno komponentou přímo). Pro takový systém by šel také vytvořit UML profil, ale výsledný systém by se modeloval UML diagramem tříd. Cílem práce však bylo UML profil pro modelování komponentových diagramů.

Jednodušší verze tak, oproti meta-modelu z [22], postrádá další definované typy (původní řídící rozhraní podporující aspekt mobility komponentového systému) a také vazby `BindInward` pro spojení vnějšího poskytovaného rozhraní složené komponenty s vnitřním vyžadovaným rozhraním a `BindOutward` pro spojení vnitřního poskytovaného rozhraní složené komponenty s jejím vnějším požadovaným rozhraním, které nahradil `GatePort` a třída `Binding`.

4.2 Metodika a implementace UML Profilů

V této kapitole je popsán postup definice UML profilu vycházející z postupů a z informací v článcích [1], [7] a dokumentů [12] a [14]. Každý krok je demonstrován v následující kapitole na meta-modelu definovaném v předchozí kapitole. Technologii UML profilů jsem popsal, včetně grafické notace, v kapitole 2.4.2.

4.2.1 Předpoklady

Před začátkem tvorby profilu je vhodné zvolený meta-model vypracovat (pokud ho v této formě ještě nemáme) v podobě UML diagramu tříd, kde třídy reprezentují koncepty domény. Vytvoření diagramu velmi usnadní odvození profilu v dalších krocích. Kromě samotného diagramu se samozřejmě předpokládá také znalost řešeného problému a přehled o modelování v UML.

Také bychom měli vědět, zda a jaké knihovny (UML balík s definicemi a elementy, který se importuje použitím *závislosti* se stereotypem «`import`» do balíku s profilem) a profily (balík s jiným profilem aplikovaný použitím *závislosti* se stereotypem «`apply`») budeme při tvorbě vlastního profilu používat. Základem balíku *Profiles* jsou balíky *Constructs* a *PrimitiveTypes*. Další jsou uživatelsky definovatelné.

4.2.2 Postup

Obecně se dá postup vytvoření profilu sepsat do posloupnosti kroků, jejichž pořadí je do značné míry závazné:

1. příprava, během které vytvoříme popis meta-modelu řešeného problému v UML diagramu tříd
2. identifikujeme koncepty meta-modelu a přiřadíme jim stereotypy rozšiřující vhodnou meta-třídu
3. vytvoříme vztahy mezi stereotypy
4. případné atributy tříd přiřadíme stereotypům ve formě *tagged values*
5. definujeme omezení v přirozeném jazyce pro dokumentační účely a tam kde chceme v budoucnu použít strojové zpracování zase jazyk OCL (nebo jiný podporovaný)

V následujících podkapitolách následuje popis jednotlivých kroků.

Identifikace stereotypů

Každé třídě (nebo téměř každé) z diagramu tříd meta-modelu přiřadíme stereotyp a určíme meta-třídu z UML, kterou stereotypem rozšíříme. Meta-třídu vybíráme tak, aby co nejlépe vyhovovala sémantice konceptu, který třída meta-modelu představuje, např.u komponentového systému bude jeho komponenta, ať už se jmenuje jakkoliv (viz. popis různých komponentových systémů v kapitole 3.3), pravděpodobně rozšiřovat meta-třídu `Component`. Stereotyp je vhodné pojmenovat stejným názvem jako třídu (koncept) v meta-modelu. S volbou meta-třídy také omezujeme množinu použitelných UML diagramů.

Vhodné meta-třídy není obvykle těžké vybrat, protože používané koncepty jsou často již v UML vytvořeny, jen v obecnější podobě a někdy i včetně podobné terminologie. Technologie UML profilů umožňuje pouze upřesnit, nebo přidat, sémantiku a nelze provádět velké změny, takže pokud vybraný element má vlastnosti, které koncept z meta-modelu mít nemá, je třeba buď vlastnost omezit standardními prostředky (viz. dále), nebo vybrat meta-třídu, která je obecnější. Nejčastěji to bude UML třída.

V meta-modelu představeném v kapitole 4.1 používají komponenty pro specifikaci komunikace rozhraní *Interface*, podobným konceptem v UML je `UML::Interface`. Terminologie je zde stejná, liší se jen přidanou sémantikou. Zatímco v příkladu z [7], kde je vytvářen profil pro modelování softwarových služeb a SOA (*Service Oriented Architecture*), se ke specifikaci komunikace se službou používá koncept nazvaný *Service Specification*, ale k jeho reprezentaci je také použito `UML::Interface`.

V tom samém příkladu je také názorně vidět, že pro koncepty *Message*, *Service Partition* a *Service Provider* neexistuje v UML sémanticky blízký koncept a proto je k jejich reprezentaci použita třída (`UML::Class`). Takže pokud nedokážeme v UML najít blízké koncepty, můžeme vždy použít meta-třídu `Class` s tím, že k vytváření uživatelských modelů bude použit diagram tříd (výsledné stereotypy lze aplikovat pouze na specializace meta-třídy `Class`, tedy nikoliv např. na instance meta-třídy `Port`).

Pokud by meta-model obsahoval koncept komponenty, který by se nelišil od instance meta-třídy `Component` z UML, tak mu nemusíme stereotyp přiřazovat, leda bychom chtěli zohlednit jinou grafickou syntaxi (symbol pro komponentu), nebo terminologii v podobě pojmenování stereotypu. Z toho plyne, že počet tříd v diagramu tříd meta-modelu nesouvisí s počtem nově definovaných stereotypů.

Při vytváření stereotypu můžeme specifikovat hodnotu několika atributů ovlivňujících chování instancí stereotypu v modelu:

- `isRequired` – nastavením na *true* bude místo nové instance meta-třídy v modelu automaticky vytvořena instance stereotypu, který rozšiřuje meta-třídu
- `isAbstract` – nastavením na *true* zamezíme vytváření instancí stereotypu v modelu

Stereotypu tříd, které jsou v meta-modelu označeny jako abstraktní, nastavíme atribut `isAbstract` na hodnotu *true*. V CASE nástrojích takový stereotyp nepůjde elementu přiřadit. Abstraktní stereotyp ale lze použít jako obecný typ, od kterého mohou jiné stereotypy dědit atributy.

Pokud nějaký stereotyp v modelu aplikujeme vždy a chceme vyjádřit nutnost použití tohoto stereotypu (a tím si také ulehčit práci při vytváření modelu), můžeme nastavit atribut `isRequired` na hodnotu *true*.

Vztahy mezi stereotypy

Pro vytváření UML profilů můžeme, kromě vztahu rozšíření (*extend*), využít tři další (nepočítáme-li vztah poznámky s elementem diagramu) typy vztahů: generalizace-specializace, asociace a kompozice/agregace. Kromě zmíněné výjimky, v podobě vztahu poznámky s elementem, jsou vztahy v UML profilu vždy navigovatelné a jednosměrné. Pokud to má význam, tak můžeme mezi stejnými elementy použít jinak pojmenovaný vztah s jinými názvy rolí.

Podle názvů rolí se v UML profilu pojmenují názvy od vztahu odvozených atributů.

Generalizace-specializace zachycuje vztah „kind of“. Specializace obecného stereotypu dědí atributy obecnějšího stereotypu.

Specializací můžeme modelovat realizace abstraktních tříd z meta-modelu tak, že pro každý vztah *realizace* v meta-modelu, vytvoříme v UML profilu specializaci od abstraktního stereotypu (reprezentuje abstraktní třídu v meta-modelu) směrem ke stereotypu, který reprezentuje jednotlivé realizace tříd meta-modelu.

Dalším využitím tohoto vztahu může být vytváření stromových hierarchií, nebo jiných struktur orientovaných grafů, kde jednotlivé uzly mohou být být tvořeny abstraktními stereotypy, podle kterých pak lze v OCL kontrolovat příslušnost k danému podstromu hierarchie.

Druhým typem vztahu je asociace typu *kompozice*, resp. *agregace* modelující vztah typu „part of“. Agregace vyjadřuje silný vlastnický vztah a znamená, že odkazovaný element nemá bez majitele smysl. Agregace mezi stereotypy vytváří ve stereotypu atribut odvozený od názvu role a s typem instance druhého stereotypu se kterým je ve vztahu. Podle použité násobnosti může být atribut jednoduchý (pak se jedná o asociaci), nebo může být kolekcí instancí stejného stereotypu (nebo případně jeho specializací).

V různých nástrojích je práce s kompozicí, resp. agregací, nekonzistentní. Například v nástroji Rational Software Architect se mezi stereotypy dá použít pouze vztah agregace (značena s prázdným kosočtvercem u celku), kompozici umožňuje použít pouze mezi stereotypem a třídou nebo třídou a třídou. Nástroj Visual Paradigm neumožňuje běžnými nástroji pro tvorbu diagramu profilu třídu v diagramu vytvořit a nabízí pouze kompozici. Třída v diagramu UML profilu ovšem může vystupovat pouze ve vztahu generalizace-specializace a kompozice, resp. agregace, tzn. nedokáže rozšířit meta-třídu, ani stereotyp a tudíž třídu nelze v uživatelském modelu přiřadit elementu. Je tak vhodné nebrat použité vztahy dogmaticky a případnou kompozici v diagramu meta-modelu nahradit agregací (a případně na to upozornit v dokumentaci).

Třetí typ vztahu je jednosměrná asociace, která v elementu vytvoří atribut, nikoliv kolekci. Tento typ vztahu je dostupný v nástroji Rational Software Architect, ale není dostupný v nástroji Visual Paradigm. Nicméně ji lze nahradit vztahem agregace a násobností *0..1*, nebo *1*.

V meta-modelu mohou být použity také jiné typy vztahů, například agregace, nebo různé typy závislostí (*Dependency*). Tyto vztahy technologie UML profilů nenabízí. Ale závislost instance stereotypu na jiné instanci může být ošetřena např. poznámkou, nebo omezením. Záleží vždy na konkrétní situaci.

V některých případech není nutné přenášet všechny vztahy z meta-modelu. Jedná se většinou o vztahy mezi abstraktními stereotypy a pokud existují vazby také mezi jejich realizacemi. V meta-modelu se jedná většinou pouze o informační vazby. V profilu by pak vytvořily nechtěný atribut. Další vztahy, které není nutné v profilu reflektovat jsou ty, které se váží na atribut meta-modelu UML, např. *Type* u portu. Případnou vazbu je možné zdůraznit v poznámce, nebo v poli pro dokumentaci.

Existuje několik používaných přístupů k používání vztahů mezi stereotypy:

- omezení používání vztahů pouze na vztah *extend* mezi stereotypem a meta-třídou, příklad takového profilu je v [7]
- kromě *extend* použít i vztah generalizace-specializace k vytváření speciálních verzí stereotypů
- definovat také asociace typu agregace mezi stereotypy, např. [1]

V prvním případě modelář nedefinuje žádné asociace mezi stereotypy a vztahy jejich instancí kontroluje přes omezení definované v OCL, nebo v případě přirozeného jazyka nechá dodržování korektního použití stereotypů na uživateli. Malé rozšíření předchozího případu představuje použití generalizace-specializace. Kompozice není použita ani v případě, že se mezi třídami v meta-modelu vyskytuje.

Druhým možným přístupem je použití agregace k vyjádření vztahu celek-část (nebo části v případě kolekce) mezi stereotypy. Tímto způsobem se vytvoří atributy ve stereotypech podle názvů rolí a vztahy instancí, nebo synchronizaci vazeb mezi jednotlivými elementy pak lze kontrolovat v OCL také podle hodnot těchto atributů. V některých případech je to také jediná možnost.

Specifikace nedefinuje požadavky na způsob implementace v CASE nástrojích. V nástroji použitém pro tvorbu profilu v této práci sice vytvoření vztahu mezi stereotypy vytvoří atribut stereotypu, ale ačkoliv má nástroj veškeré informace potřebné k „naplnění“ těchto atributů v modelu, tak nástroj tyto informace nepoužívá a vyplnění hodnot je přenecháno uživateli. Tento přístup tak klade větší nároky na uživatele profilu.

Atributy

Pokud třídy v meta-modelu definují vlastní atributy, použijeme pro jejich definici *tagged values* s vhodným datovým typem. V kapitole o UML jsou popsány základní datové typy v UML. Kromě těchto datových typů můžeme použít prvky definované výčtem (*Enumeration*), nebo jako typ atributu použít stereotyp a v modelu pak do něj ukládat referenci (nebo reference) na element modelu. Takové atributy vznikají použitím agregace. Pokud prostředí nenabízí datový typ *Boolean* (například Visual Paradigm), tak je možné tento typ suplovat vytvořením výčtu s hodnotami *true* a *false*.

Mnoho atributů je již v meta-třídách předdefinováno (viz. kapitola 2.2 o architektuře UML a *MOF*). Tyto atributy lze použít k jednoznačné identifikaci komponent. Mezi tyto atributy se řadí také `isAbstract` a `isRequired`, jejichž význam je uveden v části o stereotypech. Mimo atributů nabízí meta-třídy UML také operace (např. *getAppliedStereotypes()* vrací kolekci aplikovaných stereotypů).

Meta-třídy dědí od `Element` a také dalších, takže kromě jiných obsahují i stejnou podmnožinu atributů a operací se stejným významem, jsou to především:

- `Name:`*`String`* – název elementu
- `QualifiedName:`*`String`* – kvalifikovaný název, který určuje název elementu v kontextu celého modelu, např. kvalifikovaný název pro komponentu *PodKomponenta*, jejíž rodičovská komponenta se jmenuje *HlavníKomponenta* a nachází se v balíku *HlavníBalík* by měla kvalifikovaný název skládající se z názvu všech elementů v hierarchii, spojených oddělovačem: *HlavníBalík::HlavníKomponenta::PodKomponenta*
- `Owner:`*`Element`* – vlastník elementu (například vlastníkem komponenty může být jiná komponenta, nebo balík)

- `getAppliedStereotype():Set(«metaclass»Stereotype)` – operace vrací kolekci aplikovaných stereotypů

Kompletní seznam je možné nalézt ve specifikaci *UML Infrastructure* [14].

U portu jsou, kromě výše uvedených atributů, významné operace vracející kolekce: `getProvideds():Set(«metaclass»Interface)` vrací kolekci poskytovaných rozhraní portu a obdobná operace `getRequireds:Set(«metaclass»Interface)` vracející kolekci požadovaných rozhraní portu. Dalším důležitým atributem portu je `Type:«metaclass»DataType`. Typ portu lze specifikovat třídou nebo rozhraním (meta-třídy `Class` a `Interface`).

Výčet důležitých atributů komponenty a rozhraní je stejný. Bohužel sémantika atributů `Owner` a `QualifiedName` je jiná u rozhraní patřících portu, kde vlastníkem takového rozhraní je stále komponenta, nikoliv port, což u běžného pojetí portu v UML nevadí, ale pro pojetí portu, tak jak ho chápe meta-model představený v kapitole 4.1, to představuje problém. Ani kvalifikovaný název rozhraní v sobě neobsahuje název portu. V takovém případě je možné situaci vyřešit vytvořením vztahu agregace mezi stereotypem rozšiřujícím meta-třídu `Interface`, nebo samotnou meta-třídou «`metaclass`»Interface a stereotypem rozšiřujícím meta-třídu `Port` (význam je, že třída je částí portu – „part of“). V modelu je pak třeba ručně vložit referenci na rozhraní do atributu souvisejícího portu, pokud to neudělá nástroj.

Není nutné v UML profilu definovat atribut, třída v meta-modelu pro tento atribut využívá prvků EMOF (nebo obecně MOF). V meta-modelu prezentovaném v této práci se to týká například atributu `name`, který je elementům meta-modelu zpřístupněn přes meta-třídu `EMOF::NamedElement`, v UML meta-modelu je tato vlastnost zpřístupněna přes `UML::NamedElement` (která je UML instancí meta-třídy v MOF, viz. kapitola 2.3.1).

Omezení

Omezení lze tradičně určit přirozeným jazykem a jazykem OCL (2.5). Některá prostředí umožňují omezení definovat i v jiných jazycích (např. nástroj Rational Software Architect umožňuje použití Javy). V případě přirozeného jazyka se kontrola modelu nechává výhradně na uživateli, neboť v současnosti žádný CASE nástroj neumožňuje takové použití. V mnoha CASE nástrojích (například Visual Paradigm) je to ale jediný způsob, protože nepodporují zpracování OCL. OCL však lze zapsat do poznámky, ale zpracování a vyhodnocení je opět přenecháno uživateli.

Omezení vyplývají především z omezení domény. Některá tato omezení jsou zřejmá z diagramu meta-modelu, např. násobnost vazeb, počet prvků kolekce, omezení číselných atributů nebo strukturální omezení (např. instance jednoho stereotypu lze asociovat pouze s konkrétní instancí jiného stereotypu).

Způsob realizace omezení souvisí s přístupy k vytváření vztahů uvedených v části o vztazích. V prvním uvedeném případě je lze kontrolovat díky atributům a vztahům definovaných v meta-modelu jazyka UML (předchozí podkapitola). Někdy tento přístup není možný (například zmiňovaný chybějící vztah rozhraní a portu) a je nutné kontrolovat vztahy přes hodnoty atributů. Tímto způsobem lze vytvořit omezení už na úrovni samotného atributu, například tím, že určíme maximální možnou násobnost. Pak není třeba kontrolovat maximální počet prvků kolekce přes OCL.

V praxi je vhodné kombinovat OCL s atributy meta-modelu UML a vlastními atributy stereotypu tak, aby tam, kde je to vhodné, zůstala souvislost s meta-modelem. V meta-modelu z předchozí kapitoly jsou přímo definována omezení, která vyžadují synchronizaci vazeb „rozhraní – port“, „port – komponenta“. Část z nich lze realizovat v OCL s atributy

meta-modelu UML. Synchronizace „rozhraní – port“ takto ověřit nelze, neboť zde je potřeba využít atributů stereotypu.

Nevýhodou tohoto postupu je nutnost ručně zadávat hodnoty kolekcí, pokud to neudělá CASE nástroj a z toho plynoucí nepohodlí, případně špatná údržba rozsáhlých modelů. Výhodou je odstínění od UML a terminologie UML. Pokud by se v meta-modelu poskytované a požadované rozhraní nazývalo zcela jinak, musel by si uživatel v hlavě „překládat“ terminologii.

Při definici omezení lze nastavit závažnost situace při jeho porušení, existují tři úrovně:

- *Error* – závažné porušení omezení, zobrazí se chybová hlášení
- *Warning* – chyba může nastat, nebo nemusí, jedná se pouze o varování
- *Informational* – hlášení při validaci, které má pouze informační charakter

4.3 Implementace profilu komponentového systému

Na základě metodiky z kapitoly 4.2 je v této kapitole implementován UML profile pro meta-model komponentového systému z kapitoly 4.1. Diagram tříd meta-modelu je na obrázcích 4.1, relevantní část meta-modelu EMOF na obrázku 4.2, specializace `EMOF::NamedElement` na obrázku 4.3 a hierarchie typů portu na obrázku 4.4.

4.3.1 Identifikace stereotypů a meta-třídy, které budou rozšiřovat

Z diagramu meta-modelu se dají identifikovat koncepty pro komponenty (*Component*, *PrimitiveComponent*, *CompositeComponent*), rozhraní (*Interface*, *ProvidedInterface*, *RequiredInterface*), port (*Port*, *ComponentPort*, *GatePort*), vztahy mezi rozhraními (*Binding*), typy portů (*PortType*) a pro operaci (*Operation*). Jejich význam již byl popsán v kapitole 4.1. Z těchto konceptů vytvoříme stereotypy, jim přiřazené meta-třídy určíme snadno podle popisu v meta-modelu. Mapování stereotypů na UML meta-třídy je v tabulce 4.1.

Dále jsou v diagramu meta-modelu elementy meta-modelu EMOF. Ty je potřeba zohlednit při výběru vhodných meta-tříd, ale jako takové v UML profilu nefigurují.

UML Meta-třída	**Stereotypy rozšiřující meta-třídu**
Component	«`Component`», «`PrimitiveComponent`», «`CompositeComponent`»
Interface	«`Interface`», «`ProvidedInterface`», «`RequiredInterface`»
Port	«`Port`», «`ComponentPort`», «`GatePort`»
Class	«`PortType`», «`PublicPortType`», «`ProtectedPortType`», «`PrivatePortType`», «`Operation`»
Dependency	«`Binding`»

Tabulka 4.1: Stereotypy odvozené od meta-modelu.

4.3.2 Vztahy

Meta-model definuje vztahy typu realizace mezi abstraktními třídami a jejich realizacemi. Tyto vztahy v souladu s metodikou přeneseme do UML profilu jako generalizace. Dále meta-model obsahuje generalizaci, kterou přeneseme do profilu beze změny.

Dalšími vztahy v meta-modelu jsou asociace kompozice a agregace. V UML profilu je k dispozici, pro vztahy mezi stereotypy, pouze relace agregace (také záleží na podpoře ze strany CASE nástroje). Vztahy tedy přeneseme do UML profilu jako agregace. Dodatečný význam mezi nimi můžeme popsat s pomocí poznámky, nebo jiné formy dokumentace, kterou nabízí nástroj. Násobnosti vztahů jsou z meta-modelu převzaty beze změn.

Dále zanedbáme vazby mezi abstraktním stereotypem `Component` a `Port`, protože by v realizacích `PrimitiveComponent` a `CompositeComponent` vznikl nadbytečný atribut `port`.

Proti meta-modelu jsou přidány vazby tam, kde meta-model předpokládá obousměrné vazby, konkrétně mezi *CompositeComponent* a *Binding*, dále pak mezi *CompositeComponent* a *Component*.

Kvůli přehlednosti je diagram tříd UML profilu rozdělen na tři relevantní části. Spojení diagramů je realizováno přes stereotypy portů a binding. Vztahy mezi stereotypy komponent a ostatními stereotypy jsou na obrázku 4.5. Vazby mezi stereotypy pro rozhraní zachycuje obrázek 4.6. A vztah stereotypu portu a hiearchie typů portu je na obrázku 4.7. OCL omezení jsou reprezentována pouze odkazy na jejich názvy a jejich kód lze nalézt v kapitole 4.3.4.

4.3.3 Tagged values a atributy

V souhlasu s meta-modelem nastavíme u stereotypů `Component`, `Interface`, `Port`, `PortType`, `PublicPortType`, `ProtectedPortType` a `PrivatePortType` atribut `isAbstract` na *true*.

Atribut `name` vyplývající z použití tříd meta-modelu EMOF ignorujeme, protože všechny meta-třídy UML jsou specializací `NamedElement` a atribut `name` již mají. CASE nástroje tento atribut používají (nejen) v diagramech.

Zbývají dva atributy. Atribut `internal:`*`Boolean`* ve stereotypu `Interface`, který označuje, zda rozhraní tvoří vnitřní rozhraní složené komponenty a atribut jednoduché komponenty, který obsahuje popis chování komponenty: `behaviouralDescription:`*`String`*. Tyto atributy přidáme pomocí tagged values.

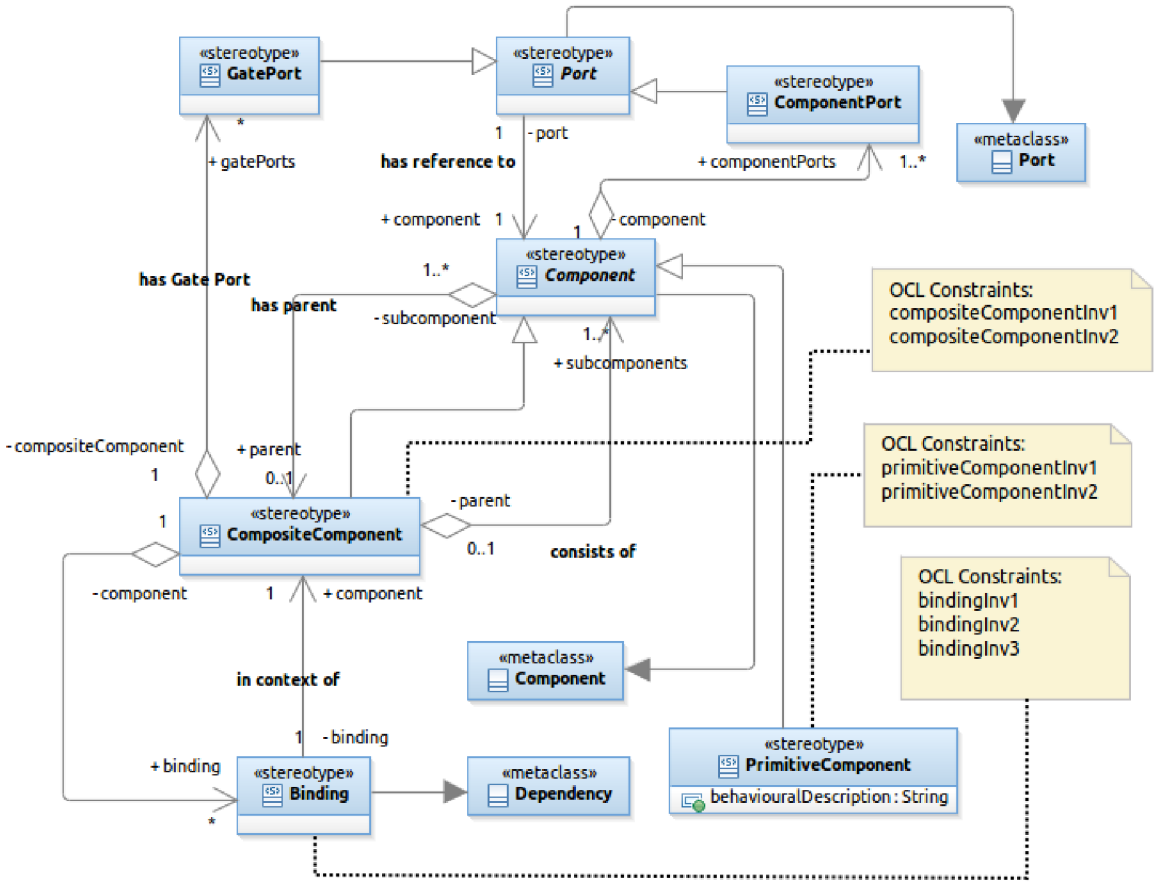
4.3.4 Určení OCL omezení

Omezení jsou určena ve dvou krocích. V prvním kroku definujeme omezení, která jsou určena přímo v meta-modelu. Ve druhém kroku definujeme omezení, která podporují vývoj, například kontrola vyplnění všech atributů.

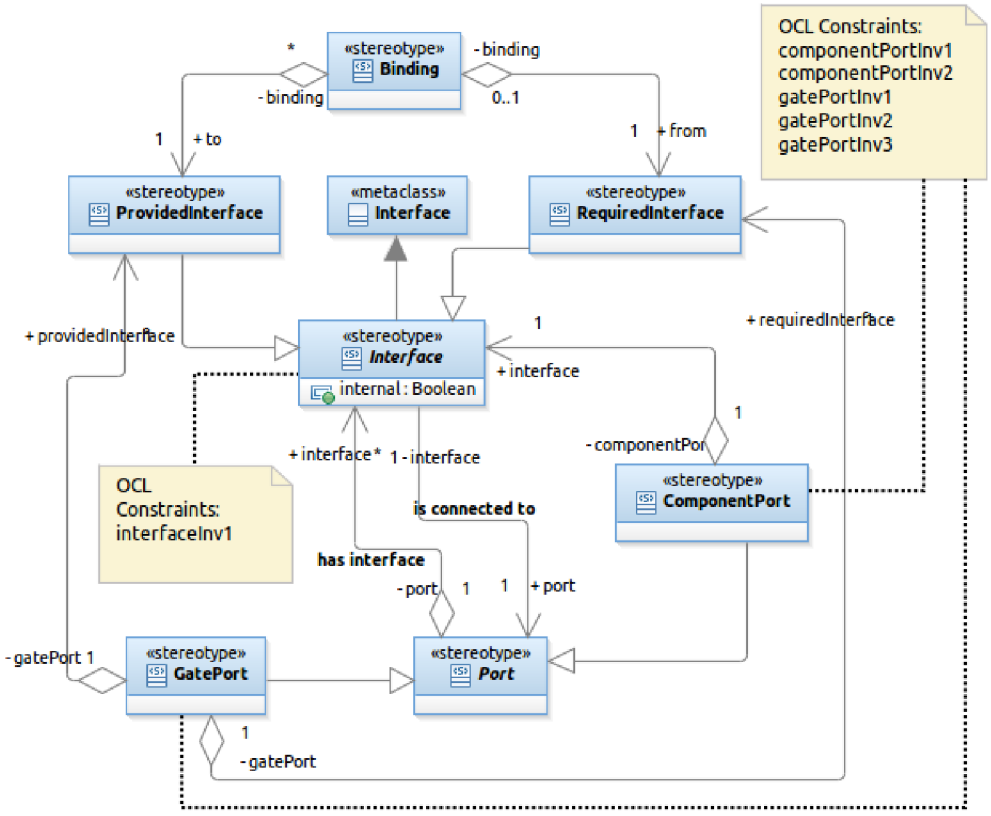
V diagramu meta-modelu jsou omezení popsána pseudokódem, v profilu je přepíšeme do OCL. V následujících přepisech OCL výrazů je použitý přesný zápis OCL dotazu. V programu RSA se zapisuje pouze tělo invariantu. Název invariantu je v implementovaném meta-modelu popsán celou větou, aby to usnadnilo ladění modelu.

Binding

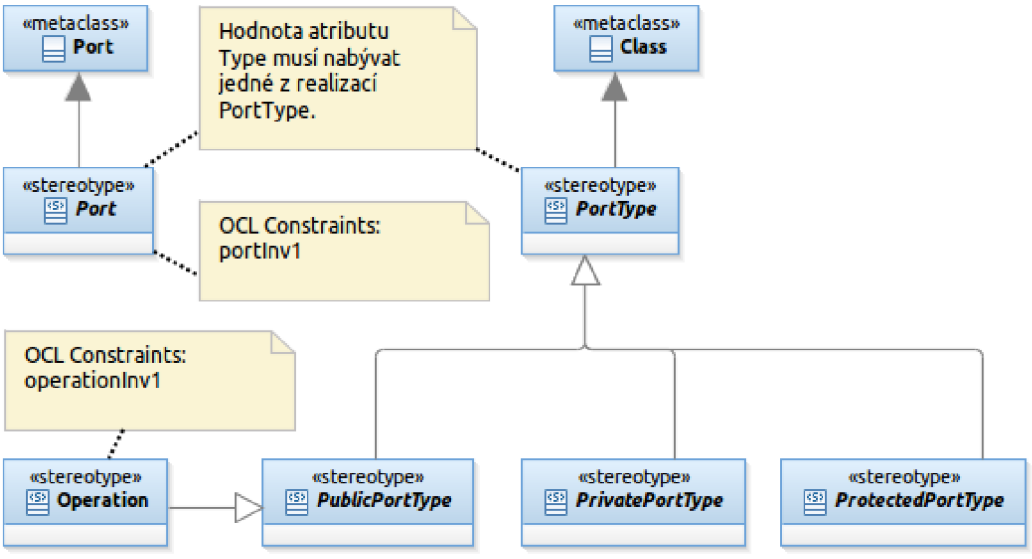
Omezení se týká propojení vnitřního rozhraní složené komponenty a jejich pod-komponent, nebo propojení vnějších rozhraní pod-komponent v rámci stejné rodičovské komponenty:



Obrázek 4.5: UML Profil: komponenty



Obrázek 4.6: UML Profil: rozhraní



Obrázek 4.7: UML Profil: typy portu

```
context Binding inv bindingInv1:
  ((not self.from.internal) and (self.from.port.component.parent = self.component)
      and (notself.to.internal) and (self.to.port.component.parent = self.component))
  or ((self.from.internal) and (self.from.port.component = self.component)
      and (notself.to.internal) and (self.to.port.component.parent = self.component))
  or ((self.to.internal) and (self.to.port.component = self.component)
     and (notself.from.internal) and (self.from.port.component.parent = self.component))
```

Lze propojit pouze rozhraní kompatibilních typů portů:

```
context Binding inv bindingInv2:
   (self.from.port.type.name = self.to.port.type.name)
     or (self.to.port.type.name = self.from.port.type.name)
```

Kontrola vyplnění atributů:

```
context Binding inv bindingInv3:
   not self.component->isEmpty() and not self.from->isEmpty()
     and not self.to->isEmpty()
```

PrimitiveComponent, CompositeComponent

Omezení abstraktní komponenty je vynecháno, protože v UML profilu není zachována vazba mezi abstraktní komponentou *Component* a abstraktním portem *Port* (z důvodů uvedených v podkapitole 4.3.2).

Rozhraní portu jednoduchých komponent musí být vždy externí:

```
context PrimitiveComponent inv primitiveComponentInv1:
   self.componentPorts->forAll( p | not p.interface.internal )
```

Vlastníkem (rodičovskou komponentou) smí být pouze složená komponenta, balík, nebo nic (v případě, že diagram není uvnitř balíku). Zde jsem k ověření použil atributy a operace meta-modelu UML, protože toto omezení nebylo v meta-modelu explicitně specifikováno. Tato konstrukce je složitější, protože nelze přímo volat operaci `isStereotypeApplied(`*`název stereotypu`*`)`, protože metoda očekává jako parametr objekt `Stereotype`. To se dá obejít metodou `getApplicableStereotype(`*`řetězec udávající kvalifikovaný název`*`)`. Funkce jazyka OCL `oclIsKindOf(`*`Element`*`)` vyhodnotí správně také generalizaci elementu, který je zadán jako parametr:

```
context PrimitiveComponent inv primitiveComponentInv2:
   (let st : Stereotype
   = self.owner.getApplicableStereotype('Component System Profile::CompositeComponent')
      in ( self.owner.isStereotypeApplied(st) ) or self.owner->isEmpty()
   or self.owner.oclIsKindOf(Package))
```

Synchronizace *GatePortu* a složené komponenty:

```
context CompositeComponent inv compositeComponentInv1:
   self.gatePorts->forAll( p | p.owner = self)
```

Stejné jako u jednoduché komponenty – rodičovskou komponentou může být pouze složená komponenta:

```
context CompositeComponent inv compositeComponentInv2:
    (let st : Stereotype
    = self.owner.getApplicableStereotype('Component System Profile::CompositeComponent')
       in ( self.owner.isStereotypeApplied(st) ) or self.owner->isEmpty()
    or self.owner.oclIsKindOf(Package))
```

ComponentPort, GatePort

Synchronizace rozhraní a portu:

```
context ComponentPort inv componentPortInv1:
    self.interface.port.qualifiedName = self.qualifiedName
```

Kontrola vyplnění atributů:

```
context ComponentPort inv componentPortInv2:
    not self.interface->isEmpty()
```

Jedno rozhraní *GatePortu* musí být interní a to druhé externí:

```
context GatePort inv gatePortInv1:
    self.providedInterface.internal xor self.requiredInterface.internal
```

Kontrola vyplnění atributů:

```
context GatePort inv gatePortInv2:
    (not self.providedInterface->isEmpty()) or (not self.requiredInterface->isEmpty())
```

Synchronizace vazeb *GatePortu* s požadovaným a poskytovaným rozhraním:

```
context GatePort inv gatePortInv3:
    (self.providedInterface.port.qualifiedName = self.qualifiedName)
       and (self.requiredInterface.port.qualifiedName = self.qualifiedName)
```

Interface

Kontrola vyplnění atributů:

```
context Interface inv interfaceInv1:
    not self.port->isEmpty()
```

Operation

Je povolena pouze jedna *operace*:

```
context Operation inv operationInv1:
    (not self.ownedOperation->isEmpty()) and (self.ownedOperation->size() = 1)
```

Port

Kontrola hierarchie typů, pokud by v budoucnu přibyly další typy portů, tak by se musely zohlednit v tomto výrazu:

```
context Port inv portInv1:
  let  st : Stereotype =
      self.type.getApplicableStereotype('Component System Profile::Operation')
        in  ( self.type.isStereotypeApplied( st ) )
```

4.4 Problémy při implementaci

Největší problémy při implementaci profilu vznikly z nekonzistence různých prostředí a implementací technologie UML profilů v CASE nástrojích.

Problémy přináší také implementace porovnání obsahu atributů v OCL, protože nelze porovnávat typy. Tento problém se dá obejít porovnáním kvalifikovaných názvů.

Svým způsobem je problém také rozhodnutí, kdy ve vyhodnocení OCL výrazu používat atributy meta-modelu UML a kdy atributy stereotypu. Většina omezení by šla v OCL vyřešit i bez vytváření vztahů mezi stereotypy. Pro uživatele by také bylo lepší, kdyby nemusel ručně vyplňovat hodnoty atributů a kolekcí. Přesto použití vztahů a odvozených atributů umožní odstínit vývojáře od velkého množství atributů a operací UML. Nakonec jsem se rozhodl v UML profilu přenést vazby mezi elementy meta-modelu také do UML profilu a použít k definici omezení takto vzniklé atributy elementů místo prostředků, které nabízí UML. Otázkou zůstává, který přístup je vhodnější?

4.5 Nasazení profilu (*Deployment*)

Nasazení profilu závisí především na použitém programovém vybavení. Rational Software Architect nabízí čtyři možnosti [13]:

- *formou uložení do souboru (file-based)* – přímá a jednoduchá metoda. Je nutné nastavit proměnou *path-map* v nastavení, pokud má být model profilu sdílen s uživatelským modelem. Jednotlivé verze RSA se často liší v nastavení, proto neuvádím konkrétní cestu k nastavení.

- *formou doplňku (plug-in)* – umožňuje lepší distribuci a správu profilu. Profil je popsán v deskriptoru *plugin.xml* a archivován do podoby balíčku, který je uživatelům distribuován. Ti jej musí importovat do svého prostředí. Použití takového profilu pak spočívá ve vybrání profilu ve výběrovém menu, plugin však bude kompatibilní jen s nástroji IBM řady Rational Software

- *formát XMI 2.2* – formát a standard spravovaný OMG

- *formát UML2* – open source formát pro export modelů UML 2. Podle konvence má výsledný soubor formát `<profile name>.profile.uml2`.

Implementovaný profil je na datové příloze ve všech formátech, včetně plug-inu pro snadnější použití v programech řady Rational Software.

4.6 Přenositelnost a podpora v CASE nástrojích

Výstup této práce je k dispozici, kromě projektových souborů, také ve formě pluginu pro nástroj IBM Rational Software Architect (alespoň verze 8.0), ve formátu UML2 pro přenos modelů v jazyce UML 2 a také ve formátu XMI verze 2.2. Nižší verzi nepodporuje.

XMI by měl být do značné míry univerzálním formátem, bohužel velmi záleží na použité verzi. Při pokusech s importováním profilu do nástrojů popsaných v kapitole 2.6 skončil import XMI souboru vždy chybou. Pokud nástroje v budoucnu implementují XMI 2.2 může import fungovat.

Jediný nástroj, který dokázal importovat a použít profil, byl program Papyrus (kapitola 2.6.4), který zvládl importovat profil ve formátu UML2.

Některé nástroje také nemají plnou implementaci profilů, nebo např. zcela chybí OCL (*Visual Paradigm*, kapitola 2.6.1), takže i kdyby se import profilu povedl, tak by nástroj mohl použít jen část profilu v podobě definovaných stereotypů a atributů.

Celkově je podpora UML profilů napříč nástroji nekonzistentní a možnosti přenositelnosti přinejmenším nejisté.

Kapitola 5

Praktické použití profilu

Tato kapitola práce demonstruje UML profil ve formě příkladové studie vytvořený v předešlé kapitole. Na vhodném příkladu demonstruje postup instalace plug-in doplňku s UML profilem a jeho použití v nástroji *IBM Rational Software Architect* (dále jen RSA), ve kterém dále demonstruje postup od definice složených a jednoduchých komponent, přes demonstraci použití portů, vytváření rozhraní, demonstraci principu generalizace-specializace komponent a také použití UML artefaktů, které se dají použít k popisu rozmístění komponent.

Vybraný příklad je zjednodušený, ale umožňuje demonstrovat všechny principy modelování komponentového systému v implementovaném UML profilu, včetně hierarchie komponent.

V podkapitole 5.3 je demonstrováno ladění modelu s využitím nástroje validace modelu.

Poslední kapitola shrnuje podporu pro modelování s využitím UML profilu v použitém CASE nástroji *Rational Software Architect* i v druhém, konkurenčním nástroji, *Visual Paradigm*.

5.1 Příkladová studie (zadání)

Předmětem příkladu je komponentový systém pro sledování objednávek. Systém ke své funkci vyžaduje zdroj datové perzistence, uživatelský účet a vstupní objednávku. Měl by po vložení a změně stavu objednávky odeslat upozornění zákazníkovi. Zaslání informací o objednávce by mělo být realizováno různými způsoby.

Systém musí vést informace o uživatelově profilu, ve kterém jsou uchována data uživatele potřebná k vyplnění fakturačních údajů a ke kontaktování zákazníka při změně stavu objednávky. Také by měl sledovat historii objednávek a umožnit získání informací o objednávce.

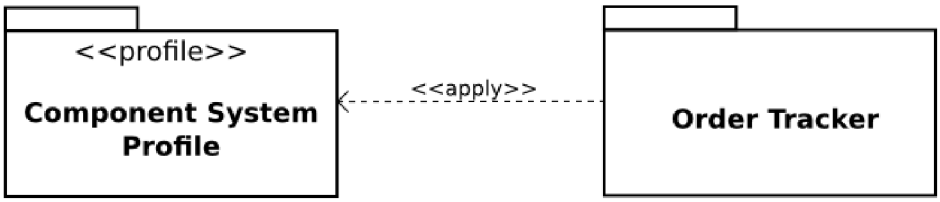
Vzhledem k tomu, že má příklad sloužit hlavně k demonstraci, je vhodné, aby rozsah příkladu nebyl příliš velký, proto je řešení příkladu zjednodušené.

5.2 Ukázka použití profilu a modelování systému

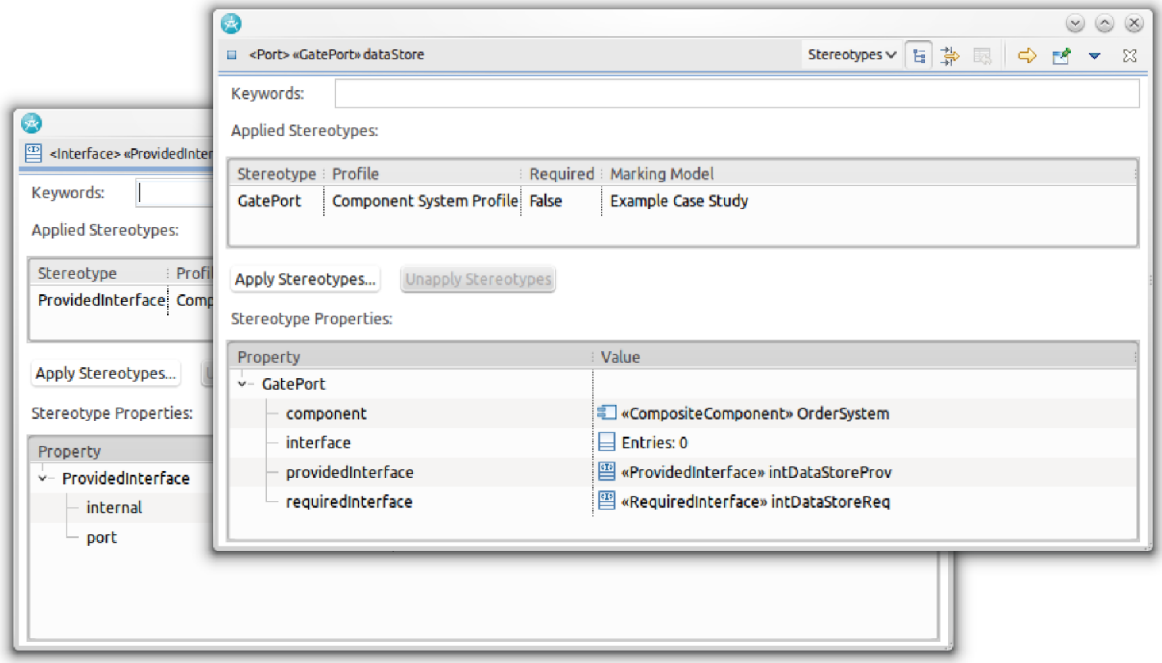
Podkapitola popisuje instalaci profilu formou pluginu a jeho aplikaci na model. Poté pokračuje realizací příkladu.

5.2.1 Instalace doplňku (plug-in)

Ruční instalace doplňku s UML profilem spočívá v „rozbalení“ distribučního zip archivu do adresáře programu Rational Software Architect (tak aby byl *jar* soubor v archivu v adresáři



Obrázek 5.1: Aplikace UML Profilu na balík s uživatelským modelem



Obrázek 5.2: Nastavení atributů instancí stereotypu

plugins). Po restartu *workspace* bude plugin dostupný.

5.2.2 Aplikace profilu

Nezávisle na použitém softwaru se profil obecně aplikuje [14] na model tak, že se mezi balíkem s uživatelským modelem a balíkem s profilem (označeným stereotypem «`profile`») vytvoří vztah závislosti (*Dependency*) označený stereotypem «`apply`».

Pokud byl profil nainstalován do prostředí formou plug-inu, je ho nyní možné aplikovat. Ve vytvořeném projektu je potřeba označit balík s modelem, pak je nutné (viz. obrázek 5.6) ve vlastnostech modelu a nabídce *Profiles* přidat profil. V dialogu, který se ukáže po kliknutí na tlačítko *Add Profile ...* je nabídka *Deployed Profiles*, ve které je nutné vybrat profil s názvem: *xpagac02 Component System Profile*. Po potvrzení dialogu je profil aplikován na model a je možné ho začít používat.

5.2.3 Modelování systému

Systém se skládá celkem z 11 komponent. Hlavní složená komponenta «CompositeComponent» OrderSystem obsahuje jednoduché komponenty:

- «PrimitiveComponent» Order – komponenta, která zpracovává objednávku
- «PrimitiveComponent» Notifier – komponenta, která se stará o zasílání zpráv o stavu objednávky

Dále «CompositeComponent» OrderSystem obsahuje další složenou komponentu «CompositeComponent» Client, která také obsahuje dvě jednoduché komponenty:

- «PrimitiveComponent» ClientProfile – komponenta spravující uživatelský profil
- «PrimitiveComponent» OrderHistory – komponenta obstarávající historii objednávek

Kromě této hlavní složené komponenty systém obsahuje také jednoduchou komponentu «PrimitiveComponent» DataStore, která reprezentuje komponentu poskytující datovou perzistenci a jednoduchou komponentu nabízející funkcionalitu k fyzickému odeslání zprávy – «PrimitiveComponent» Messenger, která je obecnější verzí tří dalších komponent, konkrétně «PrimitiveComponent» SMS, «PrimitiveComponent» E-mail a «PrimitiveComponent» DopisOnline. Tyto komponenty realizují stejné rozhraní jako komponenta (dále již budu uvádět bez stereotypu) Messenger a tedy jsou zaměnitelné. Každá z těchto komponent obstarává jinou metodu odeslání zprávy.

Komponenty v RSA a jejich uspořádání do hierarchie

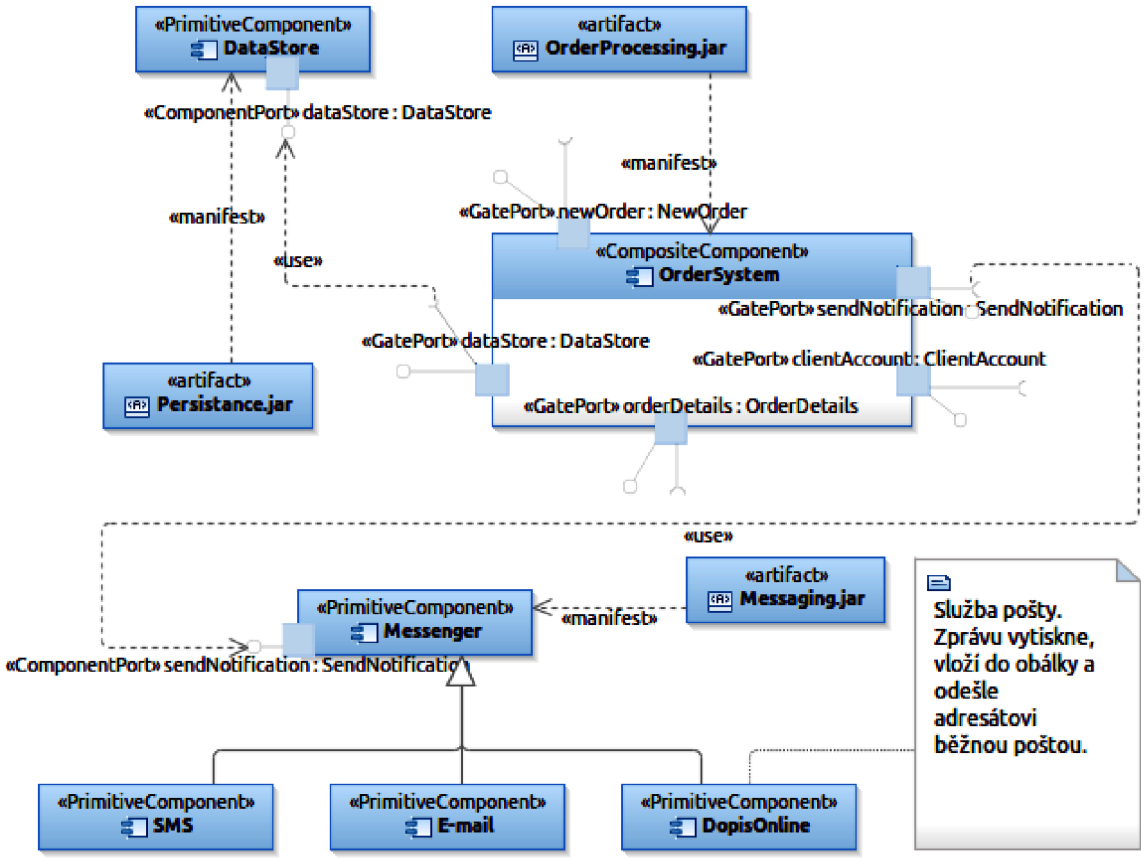
V RSA je „whitebox“ výchozím pohledem na komponentu, bohužel v tomto pohledu nelze snadno používat porty a „lolipop“ notaci. Pro přepnutí na *externí* pohled je třeba v kontextovém menu komponenty zaškrtnout položku *Filters – Show External View*.

Program RSA také neumožňuje vytváření hierarchie komponent roztažením tvaru komponenty a přetažením její pod-komponenty do vzniklého prostoru (vizuálně to jde napodobit, ale komponenta se nepřesune ve stromu v paletě *Project Explorer*), proto doporučuji komponenty vytvářet právě v *Project Exploreru*. Pro vytvoření pod-komponenty je třeba kliknout na nadřazený element a z kontextové nabídky *UML – Add UML – Component* přidat komponentu a její pod-komponenty. Takto vytvořené komponenty se automaticky nezobrazí v diagramu. Pro zobrazení v diagramu je nutné komponenty z *Project Exploreru* buď přetáhnout myší, nebo z kontextového menu vybrat *Visualize – Add to Current Diagram*. V diagramu se pod-komponenty zobrazí ve vztahu *contains* (lze vidět na obrázku 5.6).

Nyní je možné v rámci komponenty vytvořit port. Požadované a poskytované rozhraní ve formě „lolipop“ a „socket“ lze přidat z palety UML elementů *Composite Structure* (ve výchozím nastavení po pravé straně editoru).

Porty, rozhraní a vazby

Jakmile je vytvořen port v dané komponentě, můžeme mu z palety *Composite Structure* přiřadit požadované, nebo poskytované rozhraní. Na obrázku 5.2 je znázorněno nastavení atributů instancí stereotypů UML Profilu. Obdobně je třeba nastavit každý nově přidaný element.



Obrázek 5.3: Příkladová studie - hlavní složená komponenta a její vnější prostředí

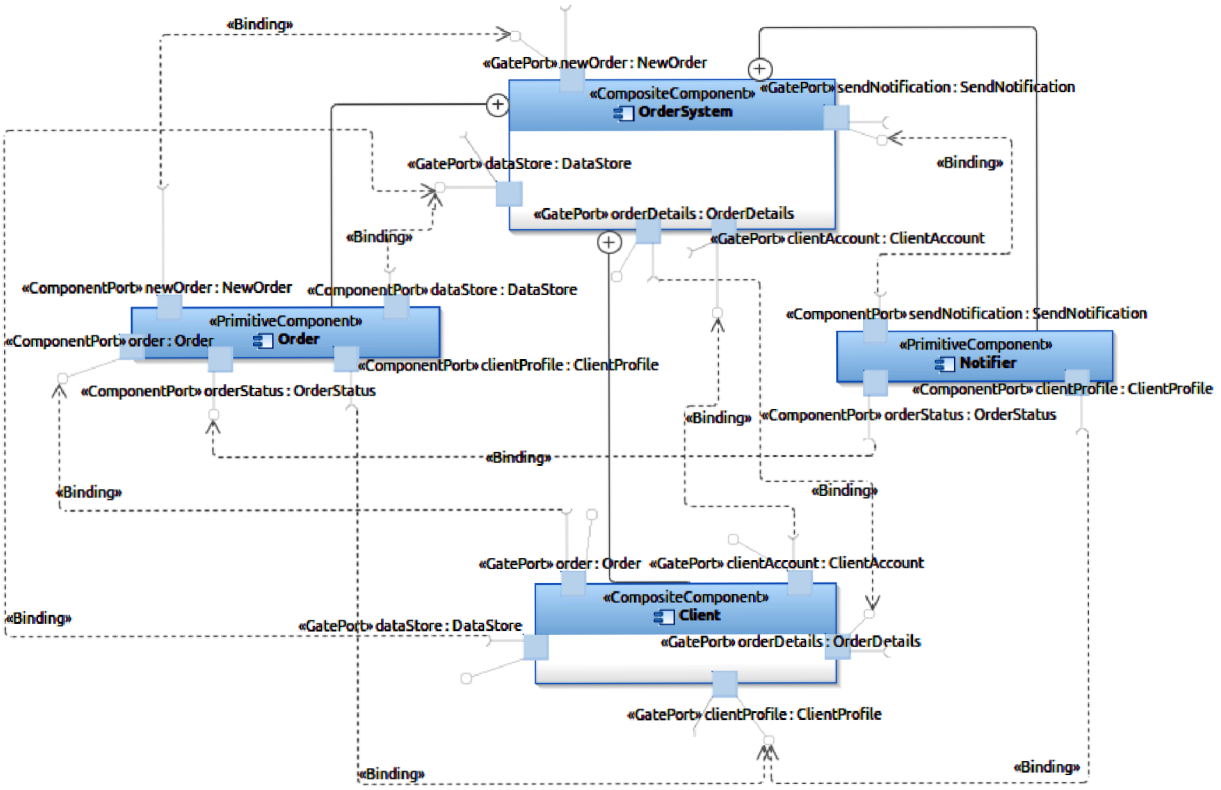
V rámci složené komponenty používáme k propojení rozhraní komponent (s kompatibilními porty) vztah závislosti (*Dependency*), označený stereotypem «`Binding`». Mimo prostředí složené komponenty pak obyčejnou závislost se stereotypem «`use`».

Kvůli přehlednosti je model rozdělen na tři samostatné diagramy. Vnější prostředí hlavní složené komponenty zobrazuje diagram na obrázku 5.3. Vnitřní prostředí této komponenty je na obrázku 5.4 a vnitřní prostředí složené komponenty *Client* prezentuje diagram na obrázku 5.5.

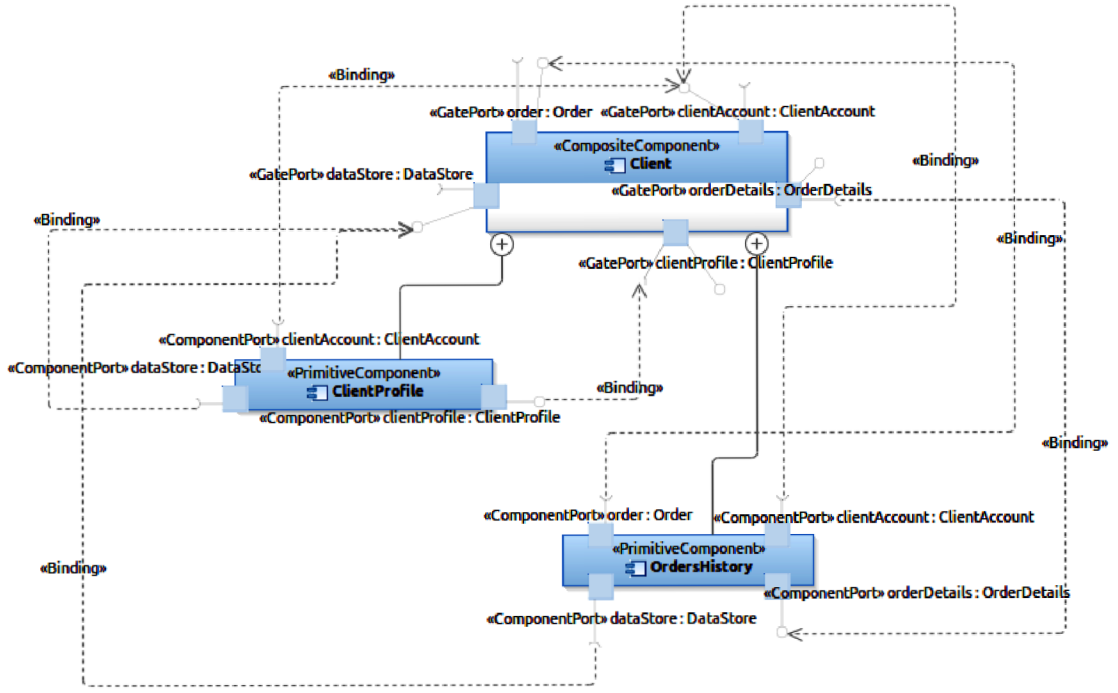
Bohužel RSA nepodporuje *assembly konektor*, proto jsou používány závislosti a nadměrně využívány vazby typu *Binding*, které by jinak jako anonymní vazby mohly být tvořeny právě *assembly konektorem* (se stereotypem *Binding*).

5.3 Validace a ladění modelu

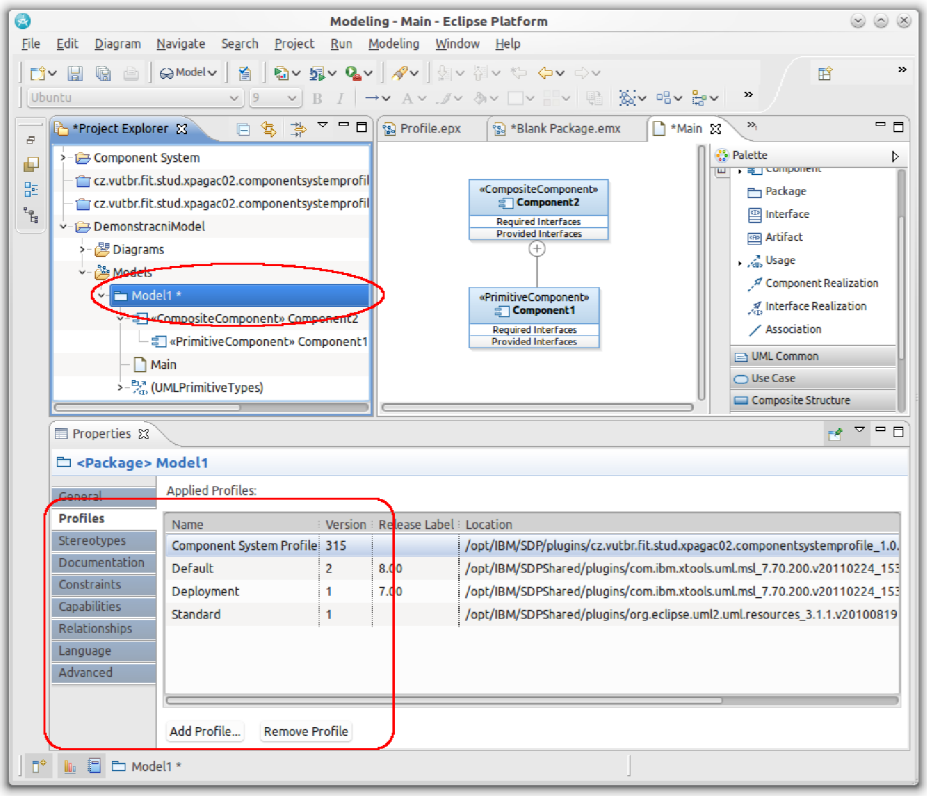
Díky přítomnosti OCL omezení a díky tomu, že RSA podporuje validaci modelu podle těchto omezení, můžeme v diagramu odhalit případné chyby. Validovat můžeme libovolnou podmnožinu diagramu, ale pokud chceme ověřit celý systém, je třeba spustit validaci nad balíkem. To se provede spuštěním funkce *Validate* z kontextového menu balíku, ve kterém máme vytvořený model.



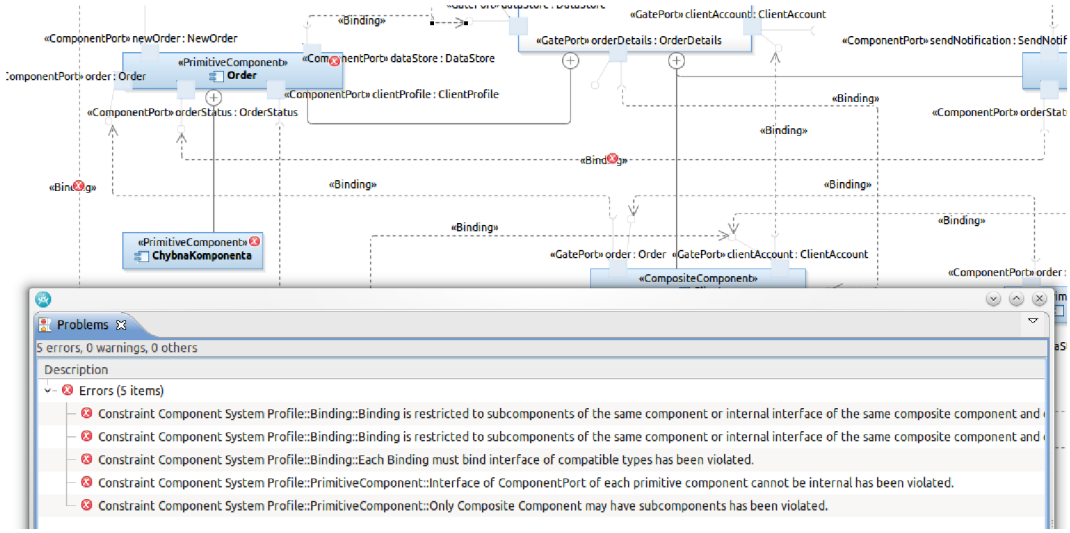
Obrázek 5.4: Příkladová studie - vnitřní prostředí hlavní složené komponenty



Obrázek 5.5: Příkladová studie - vnitřní prostředí druhé složené komponenty



Obrázek 5.6: Aplikace UML Profilu v programu Rational Software Architect



Obrázek 5.7: Validace modelu

Případná chyba se ukáže ve výpisu chyb a zároveň i formou ikonky přímo v diagramu u místa výskytu chyby.

Na obrázku 5.7 je zachycena situace při validaci modelu, ve kterém jsem udělal záměrně několik chyb:

- vytvořil jsem v modelu novou jednoduchou komponentu a jako rodičovskou zvolil opět jednoduchou komponentu
- propojil jsem rozhraní s porty, které nemají kompatibilní typy
- rozhraní *ComponentPortu* jsem nastavil jako interní

Na základě informací, které se zobrazily mohu snadno určit původ chyby a chybu napravit.

5.4 Artefakty

Artefakty (kapitola 3.4.6) jsou použity ke spárování komponent s jejich fyzickou implementací. Zde jsou použity pouze pro ukázku. Samotné rozmístění na fyzické uzly se provádí diagramem rozmístění. Pro záměry této práce to však stačí.

Kapitola 6

Zhodnocení a závěr

Kapitola shrnuje výsledky práce a hodnotí splnění jednotlivých bodů zadání.

6.1 Zhodnocení

Záměrem práce bylo vytvořit UML profil pro modelování komponentových systémů a splnit přitom dílčí úkoly. První bod zadání je splněn v druhé kapitole této práce, která popisuje a čtenáře seznamuje s jazykem UML a meta-modelováním, jako souborem technik pro rozšíření jazyka UML. Dále pokračuje třetí kapitolou, která se věnuje komponentovým systémům a problematice vývoje založené na komponentách a jejich modelování v jazyce UML.

Druhá polovina práce se zabývá metodou tvorby UML profilů, předkládá návrh jednotlivých kroků a nabízí také jednu z možností, jak se k problému postavit. Výsledný profil umožňuje tvorbu komponentových diagramů pro popis rozmístění, propojení a hierarchickou (de)kompozici komponent. Výsledkem je samotný profil, projektové soubory a doplněk (*plug-in*) vhodný k jednoduchému nasazení v podporovaných nástrojích.

Konec práce je věnován demonstraci použití profilu na komponentovém systému sledování zakázek.

Přestože existuje kvalitní specifikace konsorcia OMG ([14], [15], [18]) k samotnému UML, tak popis tvorby profilu je nepříliš obsáhlý a zaměřuje se hlavně na infrastrukturu UML. Zároveň články, které se věnují tvorbě profilů (např. [1], [7], [13]), nenabízí hlubší pohled na UML, ale nabízí jen jakýsi základní přehled a často zanechají více otázek, než odpovědí. Tato práce může sloužit jak pro úvod do meta-modelování, tak jako obšírnější návod k tvorbě profilů a jejich použití.

Protože paralelním cílem práce je implementovat profil komponentového systému, věnuje se práce také jejich principům a to s důrazem na způsoby jejich modelování v UML. Takže může sloužit i pro orientaci v oblasti vývoje založeném na komponentách.

UML profil také lze použít pro modelování architektury komponentových systémů respektujících zvolený metamodel. Jedno ze zjištění této práce je, že podpora (včetně komerčních a drahých) CASE nástrojů, v oblasti tvorby UML profilů, má prozatím značně kolísavou kvalitu, často trpí nekonzistencí nabízených prostředků. Některé programy nepodporují jazyk pro definici omezení OCL a nabízí velmi omezené možnosti přenosu UML profilu do jiných nástrojů. Vývoj však pokračuje, podpora formátů i dalších souvisejících standardů se stále zlepšuje, takže je možné, že tyto problémy budou v dalších verzích odstraněny a vytvořený profil v nich půjde použít.

6.2 Další rozvoj

Další vývoj by mohl pokračovat několika různými směry. Patrně nejzajímavější by mohlo být vytvoření, nebo přizpůsobení existujícího CASE nástroje, který by dokázal na základě diagramu generovat spustitelný kód. Jiným zajímavým pokračováním práce by mohlo být zdokonalení opensource CASE nástroje *Papyrus* (2.6.4) směrem k lepší podpoře komponentových diagramů a hlavně větší stabilitě, neboť je to jediný nástroj (z těch, které jsem zkoušel), který dokázal načíst a aplikovat profil vytvoření v *Rational Software Architect*.

Literatura

[1] Extend and customize UML with UML profile. [Online], [rev. 2010-01-07], [cit. 2011-05-13].
URL
<http://www.visual-paradigm.com/product/vpuml/tutorials/umlprofile.jsp>

[2] AMBLER, S. W.: Introduction to UML 2 Component Diagrams. [Online], [cit. 2011-04-28].
URL <http://www.agilemodeling.com/artifacts/componentDiagram.htm>

[3] BELL, D.: UML basics: The component diagram. [Online], [rev. 2004-12-15], [cit. 2011-04-28].
URL <http://www.ibm.com/developerworks/rational/library/dec04/bell/>

[4] BRUNETON, E.; COUPAYE, T.; STEFANI, J. B.: The Fractal Component Model Specification. [Online], Verze 2.0-3 (2004-02-05), [cit. 2011-05-04].
URL <http://fractal.ow2.org/specification/index.html>

[5] DeMICHIEL, L.; KEITH, M.: Enterprise Java Beans Technology. [Online], Verze 3.0 (Květen 2006), [rev. 2006-05-08], [cit. 2011-05-08].
URL <http://download.oracle.com/otndocs/jcp/ejb-3_0-fr-eval-oth-JSpec/>

[6] FAKHROUTDINOV, K.: UML, Meta Meta Models and Profiles. [Online], [cit. 2011-04-26].
URL <http://www.uml-diagrams.org/uml-meta-models.html/>

[7] JOHNSTON, S.: UML 2.0 Profile for Software Services. [Online], [rev. 2005-04-13], [cit. 2011-05-13].
URL <http://www.ibm.com/developerworks/rational/library/05/419_soa/>

[8] KOSKELA, M.; RAHIKAINEN, M.; WAN, T.: Software development methods: SOA vs. CBD, OO and AOP. 2007, [Online], [cit. 2011-05-13].
URL <http://www.soberit.hut.fi/t-86/t-86.5165/2007/final_koskela_rahikainen_wan.pdf>

[9] KRUCHTEN, P.: The 4+1 View Model of Architecture. *IEEE Software*, ročník 12, č. 6, 1995: s. 42–50.

[10] Microsoft: Microsoft COM/COM+. [Online], [cit. 2011-05-04].
URL <http://www.microsoft.com/com/default.mspx>

[11] Microsoft: .NET Framework. [Online], [cit. 2011-05-04].
URL <http://www.microsoft.com/net/>

[12] MILES, R.; HAMILTON, K.: *Learning UML 2.0*. O'Reilly Media, Inc., 2006, ISBN 0596009828.

[13] MISIC, D.: Authoring UML profiles using Rational Software Architect and Rational Software Modeler. [Online], [rev. 2005-09-06], [cit. 2011-05-10].
URL <http://www.ibm.com/developerworks/rational/library/05/0906_dusko/index.html>

[14] Object Management Group: UML 2.3 Infrastructure. Květen 2010.
URL <http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/>

[15] Object Management Group: UML 2.3 Superstructure. Květen 2010.
URL <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>

[16] OMG: Common Object Request Broker Architecture (CORBA/IIOP). [Online], Verze 3.1 (Duben 2008), [cit. 2011-05-04].
URL <http://www.omg.org/spec/CORBA/3.1/>

[17] OMG: Common Object Request Broker Architecture (CORBA/IIOP) – Components. [Online], Verze 3.1 (Leden 2008), [cit. 2011-05-04].
URL <http://www.omg.org/spec/CORBA/3.1/Components/PDF/>

[18] OMG: OMG Meta Object Facility Specifications. [Online], Verze 2.0 (Leden 2006), [cit. 2011-05-04].
URL <http://www.omg.org/spec/MOF/2.0/>

[19] OMG: OMG Object Constraint Language Specifications. [Online], Verze 2.2 (Únor 2010), [cit. 2011-05-04].
URL <http://www.omg.org/spec/OCL/2.2/>

[20] OMG: OMG Unified Modeling Language Specifications. [Online], Verze 2.3 (Květen 2010), [cit. 2011-05-04].
URL <http://www.omg.org/spec/UML/2.3/>

[21] POLÁK, M.: *UML 2.0 Components*. Diplomová práce, Karlova Univerzita PRAHA, 2010.
URL <http://d3s.mff.cuni.cz/publications/polak-masterthesis-uml20components.pdf>

[22] RYCHLÝ, M.: *Formal-based Component Model with Support of Mobile Architecture*. Dizertační práce, Brno University of Technology, 2010.
URL <http://www.fit.vutbr.cz/research/view_pub.php?id=9174>

[23] SHARMA, A.; KUMAR, R.; GROVER, P. S.: Managing Component-Based Systems With Reusable Components. *International Journal of Computer Science and Security (IJCSS)*, ročník 1, Srpen 2007: s. 52 – 57, ISSN 1985-1553.

[24] SZYPERSKI, C.; GRUNTZ, D.; MURER, S.: *Component software: beyond object-oriented programming*. Component Software Series, Addison-Wesley Professional, druhé vydání, Listopad 2002, ISBN 9780201745726, 624 s.

[25] UDELL, J.: Componentware: Component software, as exemplified by Visual Basic's custom controls, is succeeding where object-oriented computing has failed. *j-BYTE*, ročník 19, č. 5, 1994, ISSN 0360-5280.

[26] ČERNÝ, O.; HOŠEK, P.; PAPEŽ, M.; aj.: SOFA 2 Component System: User's Guide. [Online], Verze 2, [rev. 2006-11-26], [cit. 2011-05-04].
URL <http://sofa.ow2.org/docs/pdf/users_guide.pdf>

Příloha A

UML Profil pro komponentové systémy

Příloha obsahuje předmět této práce, přehledně zapsaný do tabulky a diagram tříd UML profilu v jednom celku (obrázek A.1).

«Stereotype» Component

Abstraktní komponenta systému. Má dvě realizace: *PrimitiveComponent* a *CompositeComponent*.
Extends: «metaclass»Component
isAbstract: *true*
Attributes:

- componentPorts:*«stereotype»ComponentPort* (1..*)
- parent:*«stereotype»CompositeComponent* (0..1)

«Stereotype» PrimitiveComponent

Jednoduchý typ komponenty. Nemá vnitřní strukturu tvořenou jinými komponentami. Pouze chování.
Generalization: «stereotype»Component
Attributes:

- behaviouralDescription:*String* (1)

OCL Constraints:

Rozhraní portu jednoduchých komponent musí být vždy externí:

```
context PrimitiveComponent inv primitiveComponentInv1:
    self.componentPorts->forAll( p | not  p.interface.internal )
```

Vlastníkem (rodičovskou komponentou) smí být pouze složená komponenta:
context PrimitiveComponent **inv** primitiveComponentInv2:

```
(let st : Stereotype
= self.owner.getApplicableStereotype('Component System Profile::CompositeComponent')
   in ( self.owner.isStereotypeApplied(st) ) or self.owner->isEmpty()
or self.owner.oclIsKindOf(Package))
```

«Stereotype» CompositeComponent

CompositeComponent je realizace abstraktní komponenty *Component*. Může obsahovat další komponenty, jak *CompositeComponent*, tak *PrimitiveComponent*. Předává zprávy s využitím *GatePort* a *Binding*.
Generalization: «stereotype»Component
Attributes:

- `gatePorts:` *«stereotype»GatePort* (*)
- `binding:` *«stereotype»Binding* (*)
- `subcomponents:` *Component* (1..*)

OCL Constraints:
Synchronizace *GatePortu* a složené komponenty:
context CompositeComponent **inv** compositeComponentInv1:

```
self.gatePorts->forAll( p | p.owner = self)
```

Rodičovskou komponentou může být pouze složená komponenta:
context CompositeComponent **inv** compositeComponentInv2:

```
(let st : Stereotype
= self.owner.getApplicableStereotype('Component System Profile::CompositeComponent')
   in ( self.owner.isStereotypeApplied(st) ) or self.owner->isEmpty()
or self.owner.oclIsKindOf(Package))
```

«Stereotype» Port

Port je obecnou verzí *ComponentPort* a *GatePort*. Každý port musí mít typ. **Extends:** «metaclass»Port
isAbstract: *true*
Attributes:

- `interface:` *«stereotype»Interface* (*)
- `component:` *«stereotype»Component* (1)

OCL Constraints:
Typ portu musí patřit do typové hierarchie:
context Port **inv** portInv1:

```
let st : Stereotype =
   self.type.getApplicableStereotype('Component System Profile::Operation')
      in ( self.type.isStereotypeApplied( st ) )
```

«Stereotype» ComponentPort

ComponentPort je portem pro *PrimitiveComponent*.
Generalization: «stereotype»Port
Attributes:

- `interface:`*`«stereotype»Interface`* (1)

OCL Constraints:
Synchronizace rozhraní a portu:
context ComponentPort **inv** componentPortInv1:
self.interface.port.qualifiedName = **self**.qualifiedName

Reference na rozhraní musí být nastavena:
context ComponentPort **inv** componentPortInv2:
not **self**.interface−>isEmpty()

«Stereotype» GatePort

GatePort je portem složené komponenty používaný pro komunikaci s vnitřními pod-komponentami. Komunikace viz. *Binding*.
Generalization: «stereotype»Port
Attributes:

- `providedInterface:`*`«stereotype»ProvidedInterface`* (1)
- `requiredInterface:`*`«stereotype»RequiredInterface`* (1)

OCL Constraints:
Jedno rozhraní *GatePortu* musí být interní a druhé externí:
context GatePort **inv** gatePortInv1:
self.providedInterface.internal **xor** **self**.requiredInterface.internal

Reference na obě rozhraní musí být nastavena:
context GatePort **inv** gatePortInv2:
(**not** **self**.providedInterface−>isEmpty()) **or** (**not** **self**.requiredInterface−>isEmpty())

Synchronizace vazeb *GatePortu* s požadovaným a poskytovaným rozhraním:
context GatePort **inv** gatePortInv3:
(**self**.providedInterface.port.qualifiedName = **self**.qualifiedName)
and (**self**.requiredInterface.port.qualifiedName = **self**.qualifiedName)

«Stereotype» Interface

Interface má dvě realizace: *ProvidedInterface* a *RequiredInterface* komponenty.
Extends: «metaclass»Interface
isAbstract: *true*
Attributes:

- `internal:`*`Boolean(false)`* (1)
- `port:`*`Port`* (1)

OCL Constraints:
Reference na port musí být nastavena:

```
context Interface inv interfaceInv1:
    not self.port->isEmpty()
```

«Stereotype» ProvidedInterface

Poskytované rozhraní komponenty. Musí být asociováno s *ComponentPort* nebo *GatePort*. Atribut *internal* má význam pouze pokud je rozhraní asociováno s portem *GatePort* nějaké složené komponenty (*CompositeComponent*).
Generalization: «`stereotype`»Interface

«Stereotype» RequiredInterface

Požadované rozhraní komponenty. Musí být asociováno s *ComponentPort* nebo *GatePort*. Atribut *internal* má význam pouze pokud je rozhraní asociováno s portem *GatePort* nějaké složené komponenty (*CompositeComponent*).
Generalization: «`stereotype`»Interface

«Stereotype» PortType

Reprezentuje typ „business“ služby. Může mít operaci s typovanými vstupními a výstupními parametry.
Port je s typem portu *PortType* spojen přes UML atribut *Type* portu.
Extends: «`metaclass`»Class
isAbstract: *true*

«Stereotype» PublicPortType

Veřejný (*Public*) typ portu.
Generalization: «`stereotype`»PortType
isAbstract: *true*

«Stereotype» PrivatePortType

Privátní (*Private*) typ portu.
Generalization: «`stereotype`»PortType
isAbstract: *true*

«Stereotype» ProtectedPortType

Chráněný (*Protected*) typ portu.
Generalization: «stereotype»PortType
isAbstract: *true*

«Stereotype» Operation

Realizace veřejného typu portu *PublicPortType*. Reprezentuje „business“ službu s operací s typovanými vstupními a výstupními operacemi.
Porty typu *Operation* jsou tzv. funkční operace.
Generalization: «stereotype»PublicPortType
OCL Constraints:
Je povolena pouze jedna *operace*:
context Operation **inv** operationInv1:

```
(not self.ownedOperation->isEmpty()) and (self.ownedOperation->size() = 1)
```

«Stereotype» Binding

Reprezentuje spojení mezi poskytovaným a požadovaným rozhraním subkomponent stejné složené komponenty, nebo vnitřního rozhraní mateřské komponenty a pod-komponenty. Spojení je jednosměrné od požadovaného k poskytovanému.
Typ závislosti odpovídá typu „usage“ a koresponduje se závislostí označenou stereotypem «use».
Nastavení názvu je volitelné.
Extends: «metaclass»Dependency
Attributes:

- to: *«stereotype»ProvidedInterface* (1)
- from: *«stereotype»RequiredInterface* (1)
- component: *«stereotype»CompositeComponent* (1)

OCL Constraints:
Omezení se týká propojení vnitřního rozhraní složené komponenty a jejich pod-komponent, nebo propojení vnějších rozhraní pod-komponent v rámci stejné rodičovské komponenty:
context Binding **inv** bindingInv1:

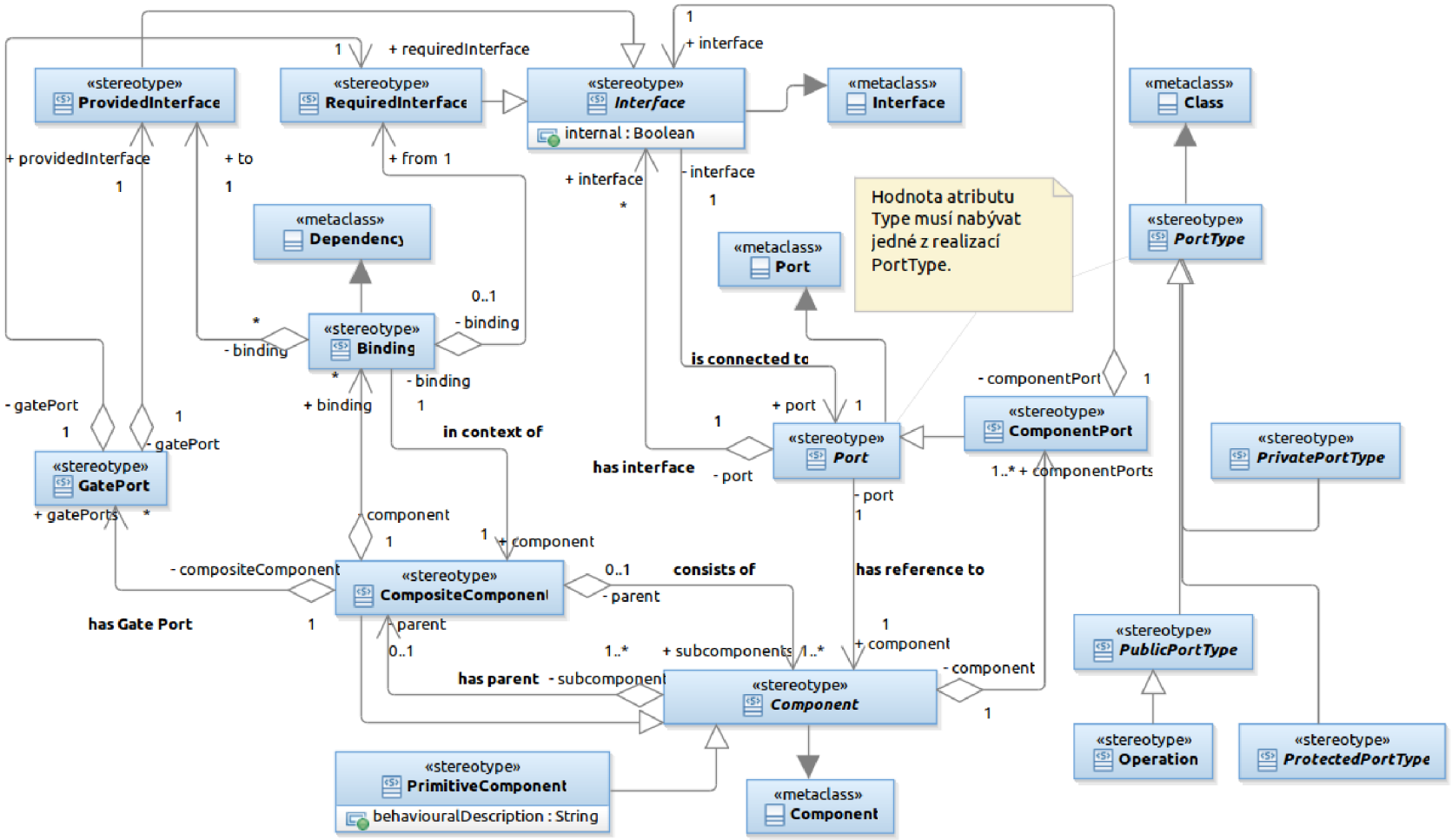
```
((not self.from.internal) and (self.from.port.component.parent = self.component)
    and (notself.to.internal) and (self.to.port.component.parent = self.component))
or ((self.from.internal) and (self.from.port.component = self.component)
    and (notself.to.internal) and (self.to.port.component.parent = self.component))
or ((self.to.internal) and (self.to.port.component = self.component)
    and (notself.from.internal) and (self.from.port.component.parent = self.component))
```

Lze propojit pouze rozhraní kompatibilních typů portů:

```
context Binding inv bindingInv2:
    (self.from.port.type.name = self.to.port.type.name)
      or (self.to.port.type.name = self.from.port.type.name)
```

Kontrola vyplnění atributů:

```
context Binding inv bindingInv3:
    not self.component->isEmpty() and not self.from->isEmpty()
      and not self.to->isEmpty()
```

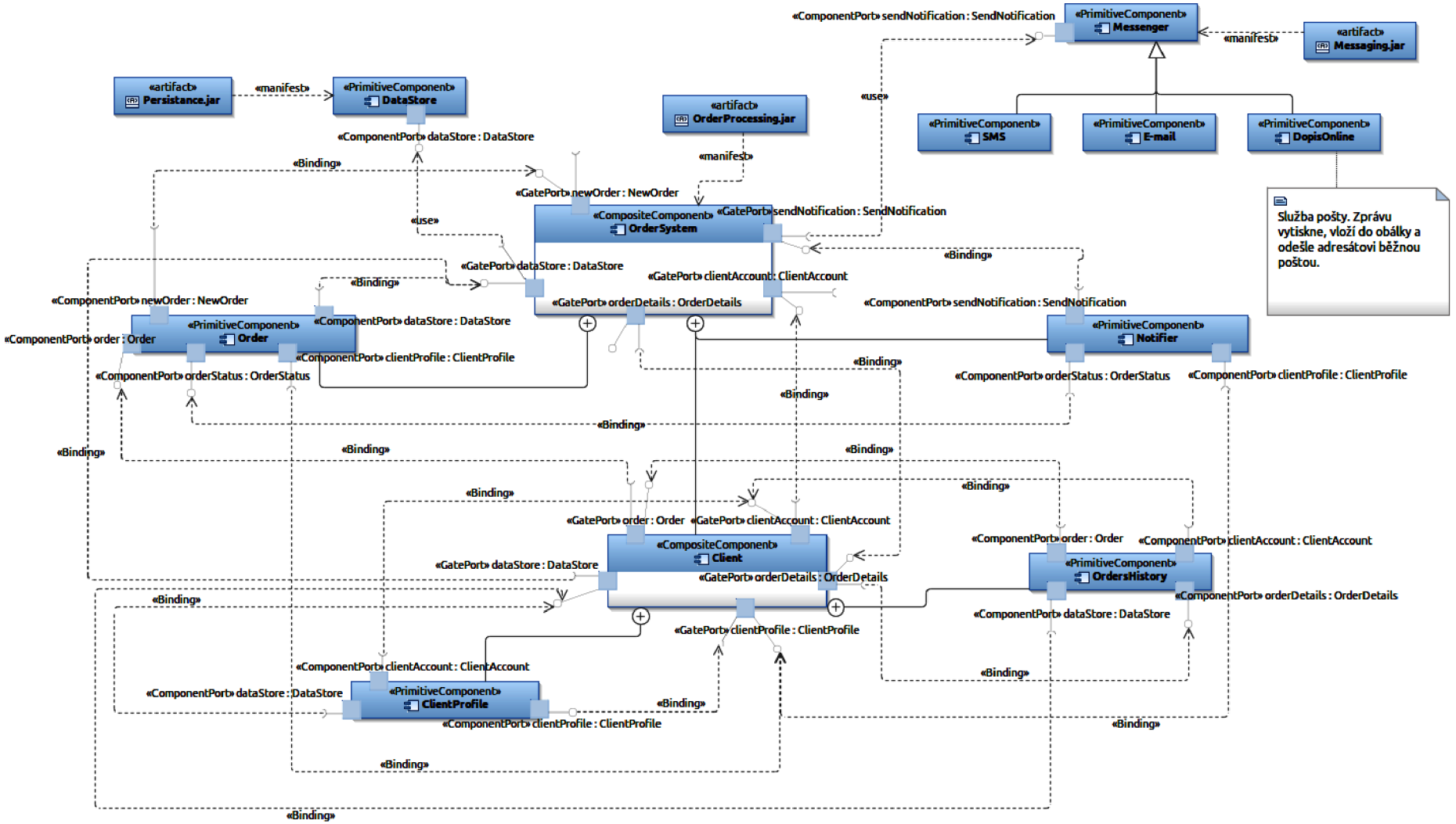


Obrázek A.1: Diagram UML Profilu

Příloha B

Úplný diagram případové studie

Diagram příkladové studie z kapitoly 5 v jednom celku je na obrázku B.1.



Obrázek B.1: Diagram komponentového systému s použitým profilem

Příloha C

Obsah CD

UML profil byl vytvořen v programu *Rational Software Architect (RSA) verze 8.0.2* (popis v kapitole 2.6.2). Tento program není volně šiřitelný. Kromě zakoupení patřičné licence je možné získat zkušební, časově omezenou, verzi programu na webu výrobce. Kromě projektových souborů pro tento nástroj, obsahuje datová příloha také plug-in usnadňující podporu profilu v tomto nástroji a soubory s profilem ve formátu XMI 2.2 (soubory **.xmi*) a UML2 (soubor **.uml*). Součástí exportu do XMI jsou také UML profily, na kterých je vytvořený profil závislý.

Na přiloženém CD jsou přítomny zdrojové soubory této práce a obrazový materiál UML profilu a příkladové studie.

Datová příloha má tuto adresářovou strukturu vzhledem ke kořenu média:

`./csp-plugin` – archiv plug-inu s vytvořeným UML profilem pro RSA
`./csp-plugin/source` – zdrojové soubory pro výrobu pluginu
`./csp-project` – projektové soubory pro RSA
`./csp-export` – soubory s profilem ve formátu XMI 2.2 a UML2
`./figures` – obrázky diagramů UML profilu a modelu
`./model-export` – soubory s modelem ve formátu XMI 2.2 a UML2 (včetně kopie profilu kvůli cestám)
`./model-project` – projektový soubor modelu příkladové studie pro RSA
`./thesis` – text této práce ve formátu PDF
`./thesis/source` – zdrojové soubory této práce