MASARYKOVA UNIVERZITA V BRNĚ
FAKULTA INFORMATIKY

Řešení IS pomocí Model Driven Architecture

DIPLOMOVÁ PRÁCE

Bc. Miroslav Patočka **2008**

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Poděkování

Chtěl bych poděkovat RNDr. Radku Ošlejškovi, Ph.D. za odborné vedení, poskytnuté rady, připomínky a ochotu při konzultacích této diplomové práce.

Shrnutí

Práce se věnuje architektuře MDA (Model Driven Architecture). Popisuje vývojové prostředí Enterprise Architect (Sparx Systems) a zaměřuje se na klíčové vlastnosti, podporu MDA a nedostatky nástroje. Použití Enterprise Architectu je demonstrováno na případové studii, která se zabývá návrhem malého informačního systému podle MDA. V závěru práce je provedena implementace webové služby modelovaného informačního systému.

Klíčová slova

OMG, Model Driven Architecture, Computation Independent Model, Platform Independent Model, Platform Specific Model, Enterprise Architect, Web Services, WSDL, SOAP.

Obsah

1 Úvod

V oblasti softwarového inženýrství existuje celá řada metodik a přístupů, kterými se vývoj softwaru řídí. Společným jmenovatelem těchto přístupů je snaha proces vývoje urychlit a zajistit zlepšení či udržení kvality výsledného softwaru. Velkou motivací těchto metodik je také úspora vynaložených nákladů. Metodiky v sobě zahrnují komplexní postupy a návody jak software vyvíjet. Podmnožinu množiny metodik tvoří postupy založené na vývoji řízeném modely (Model Driven Development). Jde o přístup založený na myšlence postupného zpřesňování modelů: z té nejvyšší vrstvy abstrakce, která obsahuje modely zákazníkových procesů bez vztahu k jejich implementaci, až po tu nejnižší vrstvu, která obsahuje modely přímo mapovatelné do zdrojových kódu. Se snahou zformalizovat a zautomatizovat tyto přechody mezi jednotlivými modely přichází koncept MDA (Model Driven Architecture). Jedná se o přístup k vývoji software založený na tvorbě formálně popsaného modelu, který zde tvoří primární artefakt. Vývoji informačního systému podle specifikace MDA se věnuje tato diplomová práce.

V úvodní kapitole se práce zaměřuje na popis konceptu MDA. Zdůrazňuje specifika vývoje softwaru, který je podle tohoto konceptu řízený. V textu jsou popsány jednotlivé modely architektury MDA a transformace mezi těmito modely. Kapitola se věnuje také popisu kritizovaných vlastností konceptu MDA a poslední část se zabývá vztahem MDA k inkrementálnímu a iterativnímu vývoji.

Další kapitola se zabývá nástroji pro vývoj dle MDA. Definuje základní rozdělení nástrojů a uvádí výčet vývojových prostředí, která mají pro MDA podporu a která reprezentují určitou skupinu (open-source, komerční, české, ...). Dále se práce zaměřuje na nástroj Enterprise Architect od společnosti Sparx Systems Pty Ltd. Ukazuje možnosti tohoto nástroje a rozsah jeho podpory pro vývoj řízený dle MDA. Aby popis Enterprise Architectu nebyl jen pouhou kopií manuálu, zaměřuji se pomocí konkrétních příkladů na nedostatky tohoto nástroje, které jsou buď svépomoci opraveny (např. přepisem transformačních skriptů) nebo jsou nahlášeny a konzultovány s výrobcem nástroje. V kapitole jsou také na příkladech předvedeny možnosti úprav vestavěných transformačních skriptů a diskutovány jsou i používané datové typy a jejich transformace.

Čtvrtá kapitola se zabývá praktickým vývojem informačního systému školních jídelen podle MDA. Příklady jsou realizovány ve vývojovém prostředí Enterprise Architect a ukazují diagramy používané ve třech hlavních MDA modelech (CIM, PIM, PSM). Na modely jsou

aplikovány transformace vestavěné v Enterprise Architectu a také transformace dle vlastních upravených skriptů.

Pátá kapitola se věnuje implementované části systému – webové službě zajišťující poskytování jídelníčků externím systémům. Kapitola ve svém úvodu obsahuje teoretickou část zabývající se technologií webových služeb (WSDL/SOAP, REST, ...). Následuje podkapitola popisující WSDL model vytvořený v Enterprise Architectu a podkapitola s popisem aplikační logiky implementované části.

Text je doplněn ilustračními obrázky. Většinou se jedná o diagramy ukázkových příkladů nebo diagramy modelů zpracovávaného informačního systému školních jídelen. Obsaženy jsou i ukázkové fragmenty transformačních skriptů a platformově nezávislý návrh grafického uživatelského rozhraní. Diagramy, skripty a zdrojové kódy, které nejsou uvedeny v textu práce, jsou k nahlédnutí v příloze.

Všechny obrazové přílohy, zdrojové kódy, zdrojové soubory k projektu jídelny a k projektu s ukázkovými příklady, skripty a generovaná dokumentace k projektům jsou k dispozici na přiloženém CD.

2 Model Driven Architecture

2.1 Co je MDA?

Specifikace MDA pochází od sdružení OMG (Object Management Group). Tato skupina, konsorcium několika set společností realizujících se na trhu objektových technologií, se zaměřuje na vytváření standardů, které umožňují interoperabilitu a portabilitu distribuovaných objektově-orientovaných aplikací. Specifikaci MDA toto sdružení schválilo jako standard v září roku 2001. V roce 2003 pak byla členy konsorcia přijata podrobnější definice této architektury. [21]

Koncept MDA využívá, rozšiřuje a integruje většinu stávajících specifikací skupiny OMG. Jde zejména o UML (Unified Modeling Language), MOF (Meta-Object Facility), CWM (Common Warehouse Metamodel), XML (Extensible Markup Language), XMI (XML Metadata Interchange) a IDL (Interface Definition Language).

„*MDA se významně odlišuje od předchozích přístupů ve využití modelovacích jazyků jako je UML. Původně se využívaly hlavně pro jednodušší porozumění a komunikaci. Naproti tomu v MDA je modelovací jazyk klíčovou částí pro definici softwarového systému. Většina - v ideálním případě všechny - specifikací struktury a chování budoucího systému, která je uložena právě v modelu, je automaticky transformována do zdrojových kódů. Znalost platformy je zakódována do transformací, které mohou být takto použity pro více systémů.*“ [14]

2.1.1 Jiné pojetí modelu

První uvedená odlišnost souvisí se způsobem použití modelovacích jazyků. Martin Fowler definuje tři základní způsoby užití UML. [5]

- UML as sketch – použití UML k vytváření dokumentací, k zachycení a prezentování myšlenek a návrhů a jako podpůrný nástroj pro komunikaci. Modely se zaměřují na určité aspekty popisovaného systému a přitom jiné detaily systému mohou skrývat. Jedná se dnes o nejčastější způsob využití UML.
- UML as blueprint – zde UML slouží k popsání systému v detailech. Zatímco „UML as sketch“ se snaží zvýraznit důležité informace, „UML as bleuprint“ se snaží o komplexní popis. Cílem je omezit programátorskou práci a zapojit do vývoje automatizované transformace modelů do zdrojových kódů. Modely musí obsahovat

dostatek informací, aby mohly být do zdrojových kódů efektivně a kompletně transformovány.

- UML as programming language – jestliže dosáhneme bodu, kdy je systém kompletně popsán pomocí modelů, stávají se diagramy kódem a lze je kompilovat do spustitelných binárních souborů.

Někteří autoři řadí využití UML v MDA do kategorie „UML as programming language“ [8]. Nástroj Enterprise Architect, pomocí kterého je v této práci vytvořena ukázková realizace informačního systému, používá UML spíše jako „blueprint“. To poukazuje na skutečnost, že mezi vizí MDA a realizací pomocí MDA nástrojů je v dnešní době ještě značný rozdíl. Na druhou stranu nelze nedostatky grafického programování připisovat jen na vrub nedostatečné vyspělosti nástrojů. Pro zvýšení účinnosti se musí také UML kvalifikovat jako grafický programovací jazyk. UML coby programovací jazyk totiž v současné době není dostatečně sémanticky bohatý (nemá dostačující "slovní zásobu"), aby mohl adekvátně vyjádřit všechny implementační detaily. Ani nová verze UML 2.0 nepřináší zcela uspokojivé řešení. K modelovacímu jazyku je stále ještě nutné používat klasický programovací jazyk. [17]

Orientace na model jako hlavní artefakt vývoje napomáhá v propojení business oblasti s oblastí technologickou a to tím, že přímo do vývoje více začleňuje odborné specialisty a obchodníky, pro které osvojení si grafického modelovacího jazyku je podstatně jednodušší než učit se tradičním programovacím jazykům.

2.1.2 Platformově nezávislý vývoj

Další klíčovou vlastností MDA je oddělení business logiky od technologie platformy. Platformou se zde rozumí množina subsystémů a technologií zajišťujících logickou sadu rozhraní a vzorů, které může aplikace využívat, aniž by musela vědět, jak jsou platformou poskytované funkce implementovány. Díky tomu mohou být aplikace vytvořené pomocí MDA snadno realizovány v celé řadě otevřených platforem (CORBA, J2EE, .NET, ...), a to nejen současných, ale i budoucích. To dává aplikacím vytvořeným pomocí MDA šanci na mnohem větší životnost. Vytváření aplikací přesahujících jedinou platformu v sobě ukrývá také vysoký potenciál pro snížení nákladů spojených s technologickými změnami. Tím, že MDA umožňuje vývoj aplikací na vyšší úrovni abstrakce, dovoluje soustředit se více na samotnou podstatu business logiky dané domény než na jednotlivé technické problémy spojené s konkrétní platformou.

2.1.3 Automatizované transformace

Klíčovou vlastností MDA jsou také transformace mezi jednotlivými modely. O úspěšnosti konceptu MDA bude totiž z velké části rozhodovat snadnost použití těchto transformací. To vede k potřebě transformace co nejvíce automatizovat.

Transformace jsou procesy tvorby cílových modelů z modelu zdrojového a dělíme je na:

- Transformace „shora-dolů“ – cílový model po vygenerování není modifikován. Změny jsou vždy provedeny v modelu zdrojovém a transformacemi jsou propagovány do modelu cílového.
- Transformace „jednosměrná“ (*one-way, unidirectional*) – do cílového modelu mohou být zaneseny dodatečné informace, které se při opakované transformaci nepřepíší.
- Transformace „obousměrná“ (*two-way, bidirectional*) – provedené změny jsou při transformaci propagovány oběma směry. [14]

Před dalším popisem transformací je nejprve potřeba definovat jednotlivé modely používané v MDA. K transformacím se opět vrátíme v kapitole 2.2.5.

2.2 Modely MDA

MDA definuje čtyři úrovně modelu: model nezávislý na počítačovém zpracování (CIM – Computation Independent Model), platformově nezávislý model (PIM – Platform Independent Model), platformově specifický model (PSM – Platform Specific Model) a kód aplikace (Code).

2.2.1 CIM (Computation Independent Model)

Model nezávislý na počítačovém zpracování modeluje firemní procesy a definuje slovník pojmů modelované oblasti. Notace modelu procesů není v UML standardizována. Doporučuje se použít upravený diagram aktivit. Pro slovník pojmů problémové oblasti se používá konceptuální model tříd. Modelem CIM projekty v MDA začínají a vytvářejí jej business analytici nebo doménoví experti, uživatelé.

2.2.2 PIM (Platform Independent Model)

Platformově nezávislý model reprezentuje koncepci řešení dané problémové oblasti na základě konkrétních požadavků. Model PIM řeší koncepční otázky ale nezohledňuje, jak bude probíhat jejich technologická implementace. Do modelu PIM je integrováno co možná největší množství technologických faktorů architektonicky podstatných pro návrh systému,

které jsou však důsledně platformově nezávislé. Základním diagramem je digram tříd, ale používají se i všechny ostatní UML diagramy (vyjma těch, které obsahují platformově závislé elementy, jako například diagram nasazení). Model PIM vytváří IT analytici.

2.2.3 PSM (Platform Specific Model)

Platformově závislý (specifický) model reprezentuje řešení na dané platformě. PSM vzniká transformací z platformově nezávislého modelu PIM (může ale vzniknout i z PSM, jak je ukázáno později na příkladech). Pro jeden PIM může existovat více PSM modelů. PSM je podkladem pro vlastní implementaci a má stejnou strukturu jako kód vyvíjené aplikace. Model PSM je tvořen zejména diagramy tříd, může ale obsahovat i další diagramy (diagram nasazení, ...). Tento model je vytvářen vývojáři.

2.2.4 Code

Také zdrojový kód aplikace je z pohledu MDA chápán jako model. Jde o model konkrétní realizace na dané platformě. [16]

2.2.5 Transformace mezi modely CIM, PIM, PSM, Code

Transformace mezi modelem CIM a modelem PIM je zcela manuální. Obvyklým postupem je transformace firemních procesů do akcí uživatele, což je zachyceno diagramem případů užití. Specifikace MDA se ale touto transformací nezabývá.

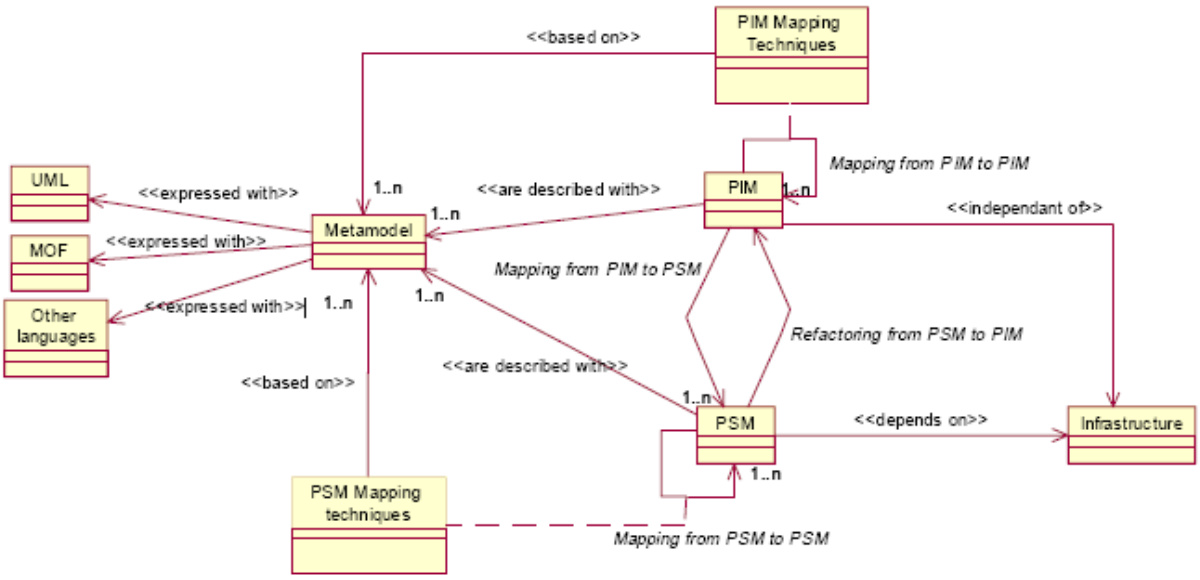
Transformace mezi modelem PSM a kódem není ničím novým. Pro tuto transformaci existovalo a existuje mnoho nástrojů. Model PSM má stejnou strukturu jako kód, takže umožňuje snadné automatizované generování. Problémem není ani opačný směr. Pomocí reverzního inženýrství lze vizualizovat kód v modelech. [11]

Přínosem MDA je hlavně automatizace transformací mezi modely PIM a PSM. Specifikace MDA rozděluje transformace mezi těmito modely do čtyř kategorií [22]:

1. Návrhář může na základě prostudování platformově nezávislého modelu vytvořit manuálně platformově specifický model.
2. Návrhář může po rozboru platformově nezávislého modelu použít ke zjednodušení práce s tvorbou platformově specifického modelu již vytvořené šablony.
3. Na platformově nezávislý model se použije algoritmus, který vytvoří kostru platformově specifického modelu. Tu pak návrhář manuálně doplní. K doplnění může použít šablony z bodu 2.

4. Algoritmus použitý na kompletní platformově nezávislý model vytvoří kompletní platformově specifický model.

Poslední z postupů je pro naplnění vizí MDA nejpřínosnější. Postup takové automatizované transformace vypadá následovně: Model PIM je doplněný mapovacími značkami určujícími, jaká obecná transformační pravidla budou použita. Vybrána je dále cílová platforma a na základě mapovacích značek jsou provedeny transformace s ohledem na zvolenou cílovou platformu. Mapovací značkou se v MDA rozumí přiřazení obecnějšího návrhového vzoru příslušnému modelovanému elementu. Konkrétní způsob transformace označeného elementu pak může být pro různé platformy jiný. Při tvorbě PSM modelu dojde k doplnění příslušných implementačních atributů, operací nebo i celých tříd a tím vznikne základ platformově specifického modelu, přičemž přidané implementační prvky jsou označeny jako modelově závislé a ve výsledném modelu lze vidět, které prvky jsou z PIM a které byly přidány až v PSM [16]. To je také důležité pro případný obrácený postup transformace z modelu PSM do modelu PIM. Reverzní postup je složitější a zatím tato problematika není uspokojivě zvládnuta. Reverzní transformace jsou užitečné, pokud například najdeme při testování chybu, která je zjevně v analýze, a chceme opravu této chyby promítnout zpět do analytických modelů. Jiným příkladem může být tvorba dokumentace pomocí reverzního inženýrství. Zde sice snadno získáme z kódu model PSM, ale ten není přímo určený pro analytiky, takže bychom z něj potřebovali vytvořit ještě model PIM. [26]



Obr. 2.1: MDA: vztah mezi modely, metamodely, jazyky, platformou, ...

2.3 Kritika MDA

Zatímco výčty výhod MDA se postupně prolínají celým textem, souvislejší popis nedostatků tohoto konceptu ještě uveden nebyl.

Blogy, fóra a diskuze věnované modelem řízenému vývoji obsahují celou řadu více či méně fundovaných názorů na MDA. Uvedl bych názory dvou významnějších odborníků z oblasti vývoje softwaru. Prvním z nich je Martin Fowler, guru v oblasti metodik vývoje softwaru a refactoringu, a druhým je Steve Cook, softwarový architekt, jeden z tvůrců jazyka OCL a propagátor doménově specifického modelování (Domain-Specific Modeling).

Martin Fowler se kriticky pouští do použití UML coby nástroje grafického programování v MDA. UML vznikl z notace vhodné pro použití coby „sketch“ (tj. jako nástroj pro podporu komunikace, zachycení myšlenek a návrhů), ale pro použití v MDA coby grafický programovací nástroj není UML ještě připravený. I přes snahu změnit tuto situaci v UML 2.0, chybí stále jasné příklady a praktické zkušenosti, které by ukázaly použitelnost této myšlenky v praxi. Martin Fowler také neshledává vytváření sekvenčních diagramů a diagramů aktivit lepším než je psaní kódu v moderním programovacím jazyku. A i když se uvedené problémy podaří vyřešit, je ještě potřeba, aby se stala technologie populární, protože pouze to, že je programovací jazyk dobrý, nezaručí jeho široké rozšíření a používání. Fowler by také uvítal možnost vývoje softwaru na vyšší úrovni abstrakce, ale není si jistý, zda tomu v MDA jazyk UML dopomůže. [4]

Steve Cook hodnotí MDA z pohledu společnosti Microsoft a srovnává jej s doménově specifickým modelováním (DSM). Domain-Specific Modeling je další z přístupů k vývoji softwaru, kde hlavním artefaktem je model. Na rozdíl od MDA se DSM nesnaží o automatické konvertování modelů, ale je založen na tvorbě modelu pro každou část systému (doménu) zvlášť a na následném ověřování či schvalování těchto modelů navzájem (pomocí nástrojů).

Cook kritizuje už samotné pojmenování MDA a tvrdí, že nejde o žádnou architekturu, ale jen o standardizovaný přístup k modelově orientovanému vývoji založený na abstrahování podobných vlastností cílových platforem. Nelíbí se mu také, že existuje málo standardních způsobů mapování mezi modely PIM a PSM. Princip MDA „napiš jednou a nasaď kdekoliv“ prý vede ke kompromisům, které mají negativní dopad na výkon a platformovou integraci. Sdružení OMG také prý při prezentování MDA nezmiňuje nevýhody spojené s využitím integrovaných modelů, vzorů, rámců a nástrojů, které je používají. A také prý fakt, že MDA je založena na používání UML a MOF, snižuje její použitelnost. Poukazuje také na konkrétní nedostatky při transformaci UML modelu do zdrojových kódů: Třídy v UML nelze přímo

použít k popisu tříd v C#, protože nepodporují „motion of properties“. A rozhraní v UML zase nelze přímo použít k popisu rozhraní Java, protože neobsahuje statická pole. [2]

2.4 MDA a iterativní vývoj

V odborných článcích se objevují i polemiky o vztahu MDA k iterativnímu vývoji softwaru. Analytická společnost Forrester Research v článku „MDA Is DOA, Partly Thanks To SOA“ od Carla Zetieho tvrdí, že životní cyklus vývoje podle MDA je neodmyslitelně typu vodopád. S tím nesouhlasí například David Frankel, autor publikací o MDA, a tvrdí, že nezná žádné přední MDA odborníky, kteří by vývoj MDA řídili dle modelu vodopád. Ve své první knize „Model Driven Architecture: Applying MDA to Enterprise Computing“ [7] uvádí, že vývoj dle MDA je bližší iterativnímu vývoji než vývoji podle modelu vodopád. Tuto skutečnost také ukazuje na několika různých příkladech z praxe. Životní cyklus vývoje dle MDA považuje za iterativní také Anneke G. Kleppe, autor knihy „MDA Explained: The Model Driven Architecture : Practice and Promise“ [10]. Stephen J. Mellor, další autor publikací o MDA [18], pak tvrdí, že vývoj dle MDA je nejen iterativní a inkrementální (přírůstkový), ale i rekurzivní.

David Frankel také vystupuje proti tvrzení Carla Zetieho, že systémy vyvíjené dle MDA mají jen monolitickou strukturu, a ukazuje příklady založené na komponentově orientované architektuře. [6]

Osobně soudím, že MDA lze bez problémů použít ve spojení s životním cyklem typu vodopád a že vyvíjený systém může mít monolitickou strukturu. Pokud inkrementální vývoj chápeme tak, že je projekt rozdělen na jednotlivé samostatně realizovatelné části a vývoj probíhá po těchto částech (přírůstcích), pak i zde lze MDA použít. Jednotlivé přírustky se však nevyvíjí zcela samostatně, musí se kontrolovat vazby na ostatní části [3]. U otázky použití iterativního vývoje společně s MDA není odpověď zcela jednoznačná a svědčí o tom výše zmiňované polemiky odborníků z oblasti vývoje softwaru.

3 MDA nástroje

3.1 Přehled a rozdělení MDA nástrojů

Ne všechny nástroje, které jsou prodávány jako „MDA nástroje“, jimi opravdu jsou. To, že umí nástroj generovat zdrojový kód z diagramu tříd neznamená, že jde o MDA nástroj. Vývojová prostředí s podporou MDA by měla rozlišovat čtyři základní modely MDA (CIM, PIM, PSM, Code) a měla by být schopna provádět transformace mezi (alespoň některými) z těchto modelů.

MDA nástroje lze rozdělit do následujících skupin [12]:

- Partial (dílčí) – nástroje typicky používající výstupy z UML modelovacích nástrojů ke generování kódu.
- Complete (kompletní) – nástroje starající se o vše od sběru požadavků až po vygenerování kódu.

Kompletní nástroje lze pak dále dělit na:

- Model-and-generate – nástroje, které z modelů generují zdrojové kódy. V současnosti většina z těchto nástrojů neumí generovat kompletní kód, který by bylo možné bez modifikací kompilovat. Je pak na vývojářích, aby kód doplnili.
- Model-and-execute – nástroje používající Executable UML (xUML, xtUML). Umožňují spustit model, na kterém mohou analytici a návrháři kontrolovat a demonstrovat funkce vytvářeného systému už v procesu návrhu.

Následující výčet MDA nástrojů není kompletním seznamem vývojových prostředí s podporou MDA, ale jde spíše o výběr několika reprezentantů zastupujících určitou skupinu (open-source, komerční, české, ...).

1. Borland Together 2006 – zástupce komerčních produktů od významné světové IT společnosti. Podporuje specifikaci Query View Transformation (QVT), která uživatelům umožňuje účelně provádět převody mezi modely. Má podporu pro OCL 2.0 se syntaktickým zvýrazněním, ověřováním správnosti a smyslu kódů.
2. OptimalJ – další zástupce komerčního produktu. Zavádí model procesů založený na diagramu aktivit v jazyce UML 2.0. Podporuje vývojovou platformu Eclipse.
3. AndroMDA – zástupce open-source produktů. Používá se v kombinaci s jinými nástroji (ArgoUML, MagicDraw, Maven). Umožňuje psát vlastní transformační

skripty i v Javě (včetně dalších jazyků používaných pro psaní transformací, např. QVT, ALT).

4. Select Architect – produkt českého výrobce (firma LBMS s.r.o.). Vývojové prostředí Select je dodáváno s metodikou LBMS Development Method, která poskytuje konkrétní návod na postup vývoje a údržby vícevrstevných aplikací s využitím principů MDA. Podporu pro MDA zajišťuje modul Select Solution for MDA.
5. Middlegen – zástupce open-source produktů. Používá databázově řízené generování kódu založené na JDBC, Antu, Velocity a XDocletu. Vývoj začíná vytvořením databáze a připojením Middlegenu k této databázi. Následuje přejmenování tabulek a sloupců, vyladění asociací a mapování. Pak je pomocí Middlegenu a XDocletu generován zdrojový kód a dodatečné soubory.

3.2 Enterprise Architect (Sparx Systems)

Vývojové prostředí Enterprise Architect pochází z dílny australské společnosti Sparx Systems Pty Ltd., kterou v roce 1996 založil Geoffrey Sparks. Vývojem Enterprise Architectu se zabývají již více než 7 let. Sparx Systems má v současné době více než 100.000 registrovaných platících uživatelů (licencí). Nástroj Enterprise Architect je poměrně levný (v ceně od 135 do 335 amerických dolarů) a má velmi nízké systémové nároky (Intel Pentium, 128 MB RAM, 70 MB HDD). I při současném spuštění více instancí (při současném porovnávání čtyř verzí nástroje) byla práce programu rychlá.

3.2.1 Základní vlastnosti EA

Enterprise Architect je komplexním vývojářským nástrojem pokrývajícím vývoj softwaru od sběru požadavků, přes analýzu, návrh, testování až po údržbu.

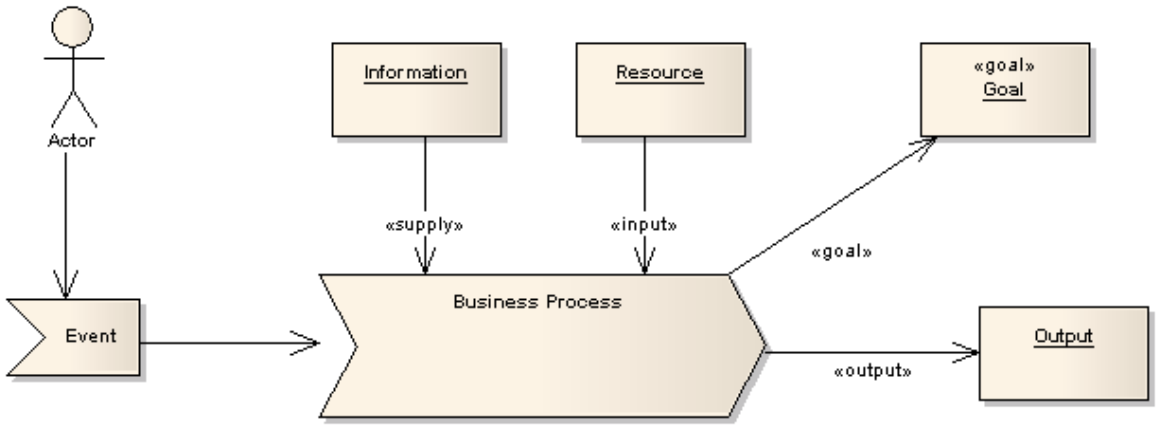
3.2.1.1 Podporované modely

Enterprise Architect podporuje všech 13 modelů UML 2.1 (Package diagram, Class diagram, Object diagram, Composite Structure, Component diagram, Deployment diagram, Use Case diagram, Activity diagram, State Machine diagram, Communication diagram, Sequence diagram, Timing diagram a Interaction Overview diagram) [24]. Popis UML diagramů není v rozsahu této práce. Je ale potřeba uvést některá rozšíření UML, která Enterprise Architect poskytuje a která jsou použita v ukázkové realizaci informačního systému jídelen.

Business Process Model. Tradičně je UML spojováno více se systémovým návrhem a programovým inženýrstvím než s analýzou a modelováním podnikových procesů. Standardy

UML 2.x ale poskytují podporu i pro modelování procesů, aktivit a informací kritických pro obchod. K standardní UML notaci poskytuje Enterprise Architect další dvě osvědčená UML rozšíření. Jde o Business Process Modeling Notation (BPMN) a nadstavbu Ericsson-Penker.

V ukázkovém řešení informačního systému jídelen je použit Business Process Model. Pro základní orientaci v tomto modelu je zde uveden popis použitých elementů a notace. Diagram ze základními elementy je uveden na obrázku 3.1.



Obr. 3.1: Business Process Model

Element v podobě tučné šipky znázorňuje Business Process. Business Process je kolekcí aktivit navržených k vytvoření specifického výstupu pro určitého zákazníka nebo trh. Každý proces má přesně definovaný cíl. Ten je na diagramu znázorněn objektem se stereotypem <<*goal*>>, který je s procesem svázán asociací se stejným stereotypem. K provedení daných aktivit používá proces vstupní zdroje. Ty se dělí na informace (*Information*) a zdroje (*Resources*). Informace proces využije k provedení aktivit, ale nijak je nemění. Zdroje jsou také použity k provedení aktivit, ale navíc jsou procesem změněny nebo doplněny. Informace jsou s procesem svázány pomocí spojení se stereotypem <<*supply*>> a zdroje pomocí spojení se stereotypem <<*input*>>. Procesy také typicky vytvářejí jeden nebo více výstupů (*Output*). Výstup jednoho procesu může být vstupem procesu jiného. Proces je spouštěn událostmi (*Events*). Událost může být v procesu zpracována a modifikována nebo může sloužit jen jako trigger ke spuštění procesu. [24]

3.2.1.2 Transformace a synchronizace

Pomocí vestavěných skriptů podporuje Enterprise Architect transformace z modelu PIM do modelu PSM, generování zdrojových kódů z modelu PSM a reverzní inženýrství. V aktuální verzi nástroje jsou k dispozici transformace pro Javu, DDL (ORM), EJB Entity, EJB Session,

C#, XSD, JUnit, NUnit a WSDL. Generování zdrojových kódů je možné pro jazyky ActionScript, C, C#, C++, Delphi, Java, PHP, Python, VB.Net a Visual Basic. Pro tyto jazyky má Enterprise Archtitect také podporu reverzního inženýrství. Z modelu databázového schématu lze generovat databázové skripty pro následující databáze: DB2, Informix, Ingres, InterBase, MSAccess, MySQL, Oracle, PostgreSQL, SQL Server 2000 a 2005, SQL Server 7, Sybase a Sybase ASE. [24]

Enterprise Architect také přináší podporu pro reverzní inženýrství některých binárních modulů. Podporovány jsou soubory Java Archive (.jar), Net PE file (.exe, .dll, bez nativních Windows DDL a Exe), Intermediate Language file (.il).

Produktivitě práce napomáhá také synchronizace mezi diagramy nad stejným modelem. Změny provedené nad třídou v jednom diagramu se projeví v modelu a tím i v ostatních diagramech, kde je tato třída (nebo její instance) zobrazena. Pokud například v sekvenčním diagramu vytváříme volání daného objektu, nabídne nám Enterprise Architect výběr z metod definovaných v odpovídající třídě. Když pro volání vytvoříme metodu novou, objeví se tato metoda i v modelu a také ve všech dalších diagramech, ve kterých je třída zobrazena.

3.2.1.3 Další vlastnosti EA

Prvním krokem v návrhu aplikace či v modelování obchodních procesů bývá typicky sběr požadavků. Správa požadavků v Enterprise Architectu umožňuje požadavky evidovat a modelovat. Namodelované požadavky lze vzájemně propojovat do hierarchií a lze je propojit i s elementy modelů, které tyto požadavky implementují.

Enterprise Architect také umožňuje pohodlně pracovat s návrhovými vzory. Vzor si návrhář může vytvořit jako běžný UML model tříd, ten pak uloží do souboru „XML Pattern file“. Vytvořený vzor pak lze po importu použít v libovolném projektu. Importované návrhové vzory jsou přístupné pomocí dialogového okna, ve kterém je zobrazen náhled diagramu, seznam elementů a stručný textový popis. Vybraný vzor lze z dialogového okna přímo vložit do diagramu, kde je možné jej podle potřeby upravovat. Výrobce Enterprise Architectu poskytuje ke stažení balík návrhových vzorů popsaných v knize „Design Patterns - Elements of Reusable Object-Oriented Software“ [9].

Enterprise Architect také podporuje návrh webových služeb. Umožňuje vytvořit WSDL model, generovat z tohoto modelu WSDL soubor a reverzně také z WSDL souboru vytvářet WSDL model.

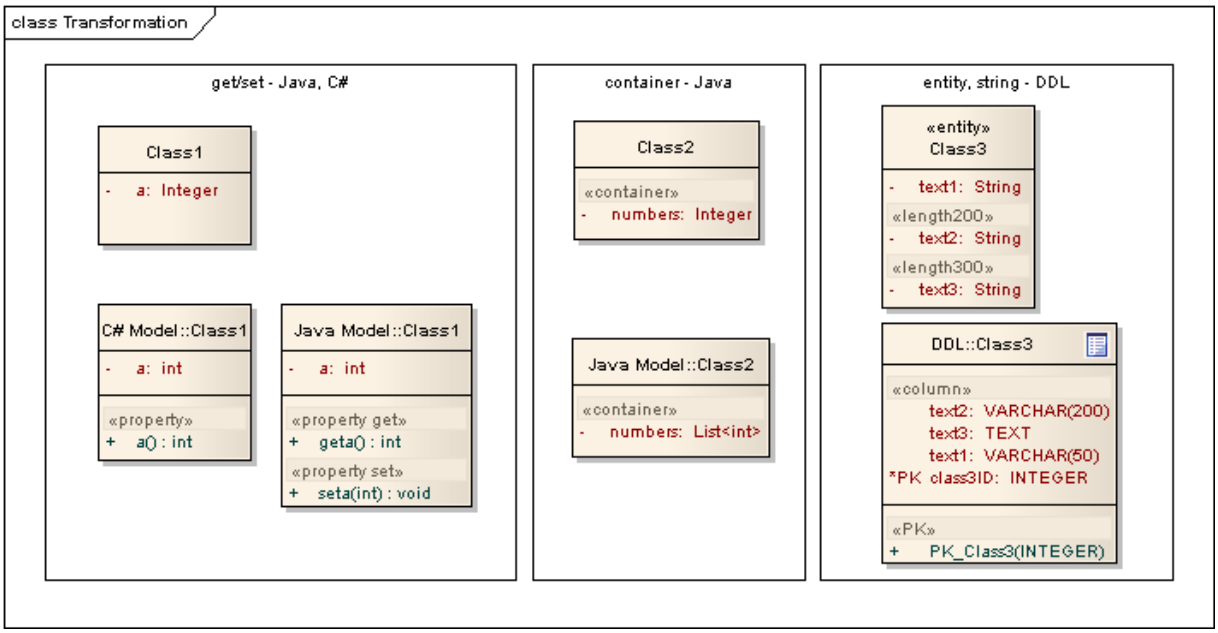
Mezi další vlastnosti patří podpora testování (Unit, Integration, System, Acceptance, Scenario), podpora tvorby dokumentace (RTF, HTML), podpora projektového managementu a

podpora týmového vývoje. [24]

3.2.2 Úprava transformačních skriptů

Enterprise Architect obsahuje vestavěné skripty pro transformace modelů. Umožňuje ale také psát své vlastní transformační skripty a nebo vestavěné skripty upravovat. Editor transformačních skriptů nalezneme pod volbou *Settings – Transformation Templates...* (nebo *Code Generation Templates...* pro skripty ke generování zdrojových kódů). Transformační skript pro danou platformu se skládá z několika dílčích skriptů, které se vztahují vždy k určitému elementu modelu (*File, Namespace, Class, Attribute, Connector*, atd.).

Při realizaci informačního systému jídelen jsem pozměnil několik skriptů pro transformaci z platformově nezávislého modelu do modelu DDL, Java a C#. Tyto upravené skripty umožňují automatické vytváření get a set metod pro atributy transformovaných tříd, generování platformově závislých kolekcí, omezení transformace do DDL modelů jen na třídy se stereotypem <<*entity*>> a specifikování datových typů v DDL modelu.



Obr. 3.2: Výsledek aplikace upravených transformací

Na obrázku 3.2 jsou třídy *Class1, Class2* a *Class3*, které jsou transformovány upravenými transformačními skripty do příslušných platformově závislých modelů. Třída *Class1* je transformována transformačními skripty uvedenými v příloze B do platformově závislých modelů Java a C#. Transformační skripty vytvoří pro všechny atributy třídy metody getter a setter. Pomocí původních vestavěných skriptů docházelo ke generování get a set metod jen pro

atributy, které byly manuálně označeny volbou *Property*. Vlastní generování metod get a set provádí dílčí skript *Properties*. Ten je volán ze skriptu pro zpracování atributů s označením *Property*. Stačilo tedy transformaci *Properties* zavolat i ze skriptu transformujícího běžné atributy:

```
$type=%CONVERT_TYPE("Java",attType)%

%Properties(attName,$type)%
```

Nejprve se konvertuje platformově nezávislý typ atributu na ekvivalentní datový typ v jazyce Java. Skript na tvorbu metod get a set je pak volán s parametry, kterými je jméno atributu a již zkonvertovaný datový typ atributu. Analogickým způsobem je upraven i transformační skript pro jazyk C# uvedený v příloze B.

Způsobem, jak v platformově nezávislém modelu použít kontejner jako typ atributu, může být jeho nahrazení polem, které v platformově závislých modelech převedeme na konkrétní kontejner daného jazyka. Můžeme ale také opatřit daný atribut značkou nebo stereotypem značícím, že se jedná o kolekci, a toto označení pak využít pro automatickou transformaci typu. Obrázek 3.3 ukazuje transformační skript, který převede typ atributu označeného stereotypem <<*container*>> na kolekci *List*<*TypAtributu*>. Tímto upraveným skriptem byla transformována třída *Class2* z obrázku 3.2. Místo konkrétního typu kolekce lze v transformačním skriptu také použít makro pro výchozí typ kolekce daného jazyka.

```
Attribute
{
  %TRANSFORM_REFERENCE()%
  %TRANSFORM_CURRENT("scope","type","stereotype")%
  scope=%qt%%attScope == "Public" ? "Private" : value%%qt%

  %if attStereotype=="container"%
    type=%qt%List<%CONVERT_TYPE("Java",attType)%>%qt%
  %else%
    type=%qt%%CONVERT_TYPE("Java",attType)%%qt%
  %endIf%

%if classStereotype=="enumeration"%
  stereotype="enum"
%else%
  stereotype=%qt%%attStereotype%%qt%
%endIf%
}
```

Obr. 3.3: Skript pro transformaci kontejnerů

Informace, která je pomocí značky nebo stereotypu <<*container*>> zanesena do platformově nezávislého modelu, je stále „platformově nezávislá“, protože i v jazycích, které kontejnery neobsahují, lze takový atribut převést na běžné pole a nebo si typ kontejneru nadefinovat.

Při transformacích ve vývojovém prostředí Borland Together hrají roli stereotypy transformovaných tříd [1]. Například pouze objekty označené stereotypem *SwDev_entity* jsou zpracovány transformačními skripty pro generování DDL modelů. Tato možnost v Enterprise Architectu nebyla a objekty k transformaci se musely manuálně vybírat a nebo se musela provést transformace nad všemi objekty z balíku. Přepsáním skriptu (viz. obrázek 3.4) jsem omezil transformace do DDL modelů jen na třídy se stereotypem <<*entity*>>. Do ukázkového DDL modelu byly tímto skriptem transformovány všechny třídy *Class1, Class2* a *Class3*, ale pouze třída *Class*3 se stereotypem <<*entity*>> prošla omezením v transformačním skriptu.

```
%if classStereotype=="enumeration"%
%endTemplate%

%if classStereotype=="entity"%

Table
{
  %TRANSFORM_REFERENCE("Table")%
  %TRANSFORM_CURRENT("language", "stereotype")%
  language=%qt%%genOptDefaultDatabase%%qt%
%list="Attribute" @separator="\n" @indent="  "%
%if elemType != "Association"%
  PrimaryKey
  {
    Column
    {
      name=%qt%%CONVERT_NAME(className, "Pascal Case","Camel Case")%ID%qt%
      type=%qt%%CONVERT_TYPE(genOptDefaultDatabase,"Integer")%%qt%
    }
  }
%endIf%
}
%list="Connector" @separator="\n"%

%endIf%
```

Obr. 3.4: Skript pro transformaci tříd <<entity>> do DDL modelu

Při transformaci platformově nezávislého modelu na model DDL se typ String převádí na typ VARCHAR(50). Pokud potřebujeme větší rozsah datového typu, musíme jej manuálně v DDL modelu změnit. Aby se potřebný rozsah datového typu generoval automaticky,

můžeme si u příslušného atributu pomocí značky nebo stereotypu poznačit požadovanou délku. Upraveným skriptem (z přílohy B) pak lze pro jednotlivé databáze generovat požadované datové typy daného rozsahu. Další upravený skript (z přílohy B) transformuje typ String na typy VARCHAR(50), VARCHAR(200) a TEXT. Výběr cílového typu se provádí na základě stereotypů, které udávají délku řetězce.

3.2.3 Datové typy

Enterprise Architect při tvorbě atributů u entit, které nemají nastaven žádný konkrétní jazyk, nabízí k výběru následující platformově nezávislé datové typy (Common Types): *Boolean, Integer, String* a *UnlimitedNatural*. Další platformově nezávislé datové typy můžeme najít v nastavení transformací datových typů (*Settings – Code Datatypes...*). Přednastaveny jsou zde například tyto platformově nezávislé typy: *Byte, Char, Float, Double, Long, Short, DateTime*, ... Pro libovolný jazyk můžeme nastavit, na jaké konkrétní typy se mají platformově nezávislé typy převádět. Pokud transformaci platformově nezávislého typu nenastavíme, pak v generovaných platformově závislých modelech zůstane typ nezměněn a je na architektovi, aby jej pro danou platformu manuálně upravil.

Pokud chceme v platformově nezávislém modelu použít kontejner jako typ atributu, parametru nebo návratové hodnoty operace, můžeme jej nahradit polem a až v platformově závislých modelech místo pole uvést konkrétní kontejner daného jazyka. V modelu pak ale není zřejmé, zda návrhář zamýšlel použití kontejneru nebo běžného pole, proto můžeme opatřit daný atribut značkou nebo stereotypem značícím, že se jedná o kolekci, a toto označení pak využít pro automatickou transformaci typu. Lze ale také definovat vlastní platformově nezávislý typ, např. *Container*, a ten pak transformovat do platformově závislých kontejnerů.

Kontejnery, které jsou modelovány příslušnou asociací, v platformově nezávislém modelu jako atribut třídy neuvádíme. Kontejner v Enterprise Architectu namodelujeme pomocí asociace s kardinalitou 0..* nebo 1..*, určením orientace vazby a pojmenováním role. Název role pak bude i názvem kontejneru. Před transformací do platformově závislého modelu si určíme typy kontejnerů pro danou platformu. Lze nastavit pro daný jazyk jeden typ kontejneru, který se aplikuje na všechny transformované kontejnery, nebo lze určit pro každou roli typ kontejneru zvlášť. Po transformaci do platformově závislého modelu se v třídě nový atribut nevytvoří a kolekce zůstane „skryta“ v asociaci. Vytvoří se ale nové get/set metody pro manipulaci s příslušnými kontejnery. Zde se Enterprise Architect liší například od vývojového prostředí Borland Together, které při transformaci z PIM do PSM vytvoří nové atributy –

kolekce, ale již nevytváří get/set metody pro manipulaci s těmito kolekcemi. Enterprise Architect vytváří atributy – kolekce až při generování kódu z platformově závislých modelů.

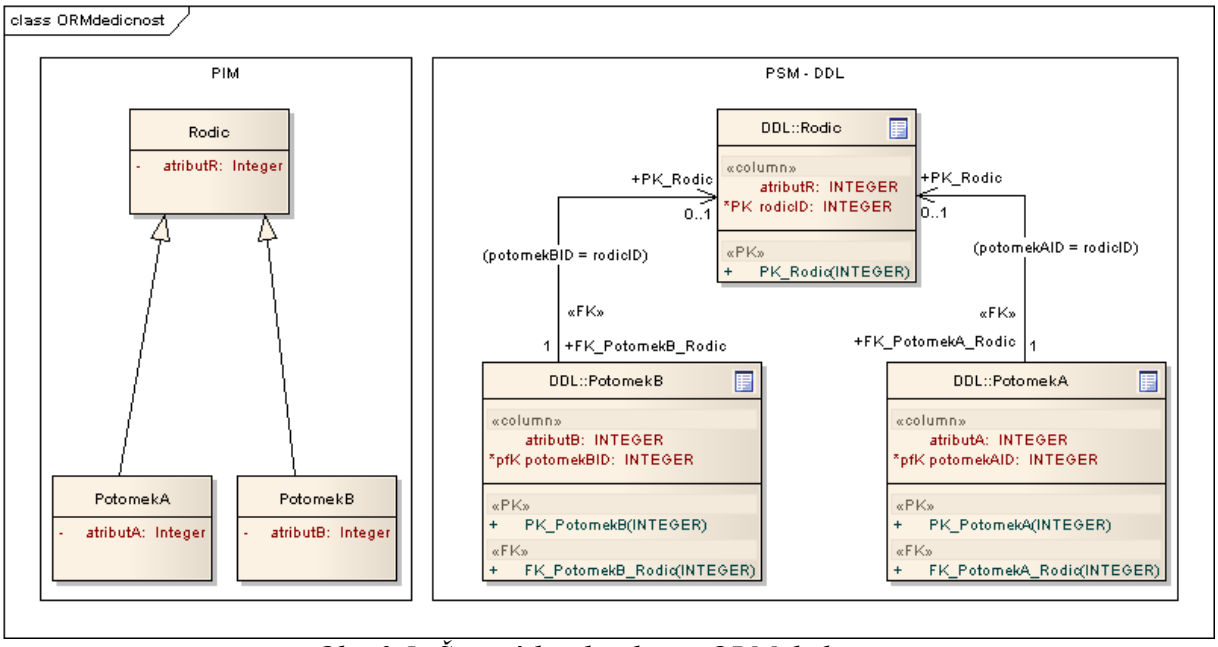
3.2.4 Nedostatky EA

V této kapitole jsou uvedeny některé nedostatky vývojového prostředí Enterprise Architect. Jde buď přímo o chyby, které způsobují například špatné transformace modelů, nebo o drobné nedostatky, které ztěžují práci s tímto nástrojem (například absence některých vlastností nebo jejich obtížné a neintuitivní dohledávání).

3.2.4.1 Špatná kardinalita u transformace dědičnosti

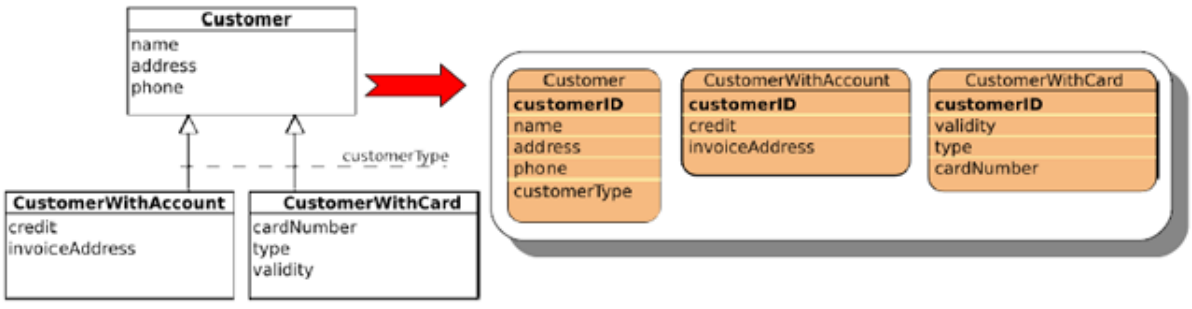
Enterprise Architect ve verzi 7.0 obsahuje chybný vestavěný skript transformace mezi platformově nezávislým modelem tříd a datovým modelem. Pokud se v modelu tříd objevuje vztah generalizace, pak po transformaci má odpovídající vazba špatnou kardinalitu (viz. obrázek 3.5). Entita reprezentující rodičovskou třídu je ve vztahu s právě jednou entitou reprezentující potomka a potomek je ve vztahu s žádným nebo jedním rodičem. Enterprise Architect používá upravené objektově-relační mapování typu „1:1“. Podle tohoto mapování by měla být výsledná kardinalita přesně obrácená. V uživatelském manuálu k Enterprise Architectu verze 7.0 můžeme najít příklad transformace vztahu generalizace, kde je výsledná kardinalita správně. Ukázkový příklad však neodpovídá výsledku transformace podle skriptů vestavěných ve verzi 7.0. Enterprise Architect ve verzi 6.5 pak kardinalitu u výsledného datového modelu vůbec neuvádí. Potřebný kód pro vytvoření kardinality v příslušném skriptu chybí a rovněž ukázkový příklad v manuálu (k verzi 6.5) kardinalitu postrádá.

Špatnou kardinalitu jsem opravil přepsáním vestavěného transformačního skriptu a o chybě jsem informoval výrobce. „Bug report“ jsem odeslal 31. března 2008 a odpověď jsem obdržel 2. dubna 2008. Za společnost Sparx Systems Pty Ltd odpověděl Simon McNeilly. Ověřili, že zmiňovaný skript opravdu obsahuje chyby, které opraví do dalšího vydání Enterprise Architectu. V době zaslání odpovědi byla k dispozici zkušební verze 7.1.829, která obsahuje stejnou chybu jako testovaná verze 7.0.813. Opravu chybného skriptu lze tedy očekávat nejspíše ve vydání Enterprise Architectu 7.1.830. „Bug report“ a odpověď naleznete v příloze A.



Obr. 3.5: Špatná kardinalita u ORM dědičnosti

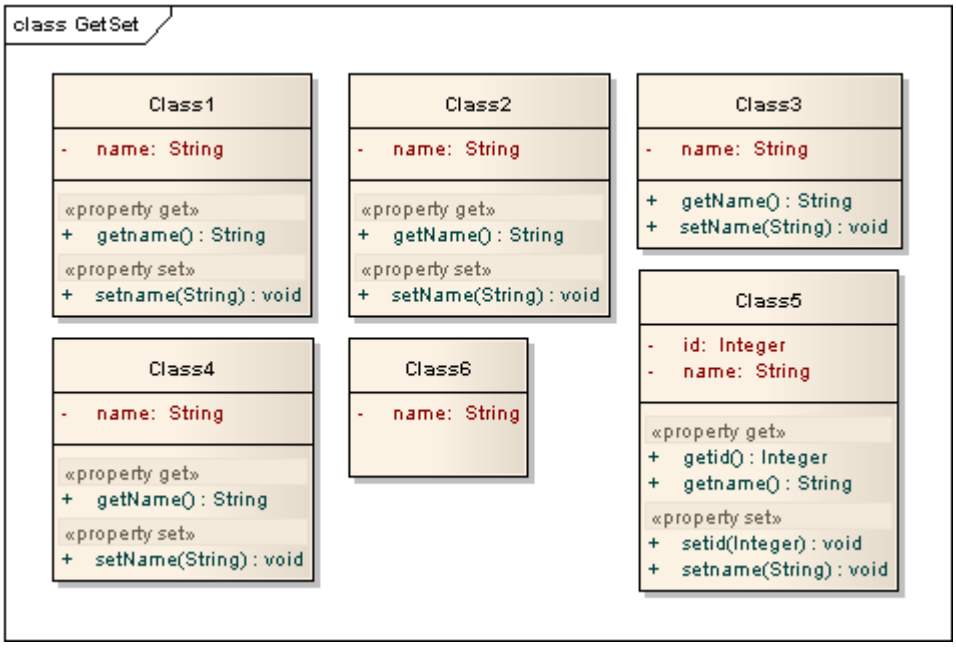
Při objektově-relačním mapování 1:1 se každá třída stává tabulkou. Tyto tabulky mají stejný primární klíč a diskriminátor se stává atributem. Tento atribut určuje, zda uložený záznam reprezentuje objekt typu rodič nebo objekt typu potomek, případně o jakého potomka se jedná. Instance obsahuje více tabulek, což vyvolává složitější přístup k datům. Enterprise Architect při objektově-relačním mapování atribut diskriminátor nevkládá. Je pak na architektovi, aby tento atribut ručně doplnil, nebo aby si dopsal příslušný transformační skript. Pokud bychom ponechali entitu bez diskriminátoru, pak musíme při čtení záznamu procházet všechny tabulky reprezentující třídy potomků.



Obr. 3.6: Diskriminátor (customerType) u mapováni 1:1 [23]

3.2.4.2 Generování get a set metod

Vývojářské nástroje pro jazyk Java zpravidla umožňují automatizované vytváření metod getter a setter. Obdobně je tomu i u nástrojů pro vývoj v C#. V prostředí Enterprise Architectu existuje několik způsobů, jak tyto metody vytvořit. Na obrázku 3.7 jsou třídy *Class1* až *Class6*, kde metody get a set vznikly rozdílným postupem. V příloze C jsou pak tyto třídy po transformaci do platformově závislého modelu (Java). Metody get a set ve třídě *Class1* vznikly automatizovaně po zatržení volby „*Property*“ ve vlastnostech daného atributu. Vytvořené metody jsou označeny stereotypem <<*properte get*>> a <<*properte set*>>. Názvy těchto metod jsou tvořeny pouze malými znaky. Podle konvence psaní názvů metod v jazyce Java by však každé vnitřní slovo názvu mělo začínat na velký znak [25]. Tuto chybu lze napravit změnou výchozího nastavení v *Tools – Options... – Source Code Engineering – Capitalized Attribute Name for Properties*. Nastavení však nelze zvolit zvlášť pro jednotlivé jazyky, je platné pro všechny platformy.



Obr. 3.7: Metody getter a setter

Metody get a set z třídy *Class2* jsou vytvořeny stejně jako metody třídy *Class1* jen s tím rozdílem, že byl po vyvolání dialogového okna u volby „*Property*“ manuálně přepsán jejich název. Docílili jsme sice správného pojmenování metod, ale tato činnost by měla být prováděna automaticky a ne manuálně.

Ve třídě *Class3* byly metody vloženy manuálně jako běžné operace. Problém tohoto postupu se projeví při generování zdrojového kódu třídy. V těle metod totiž chybí přiřazení

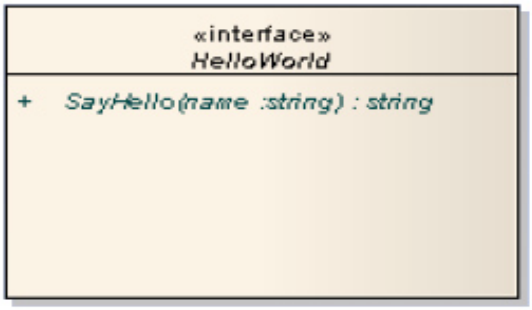
hodnoty do daného atributu třídy (pro metodu set) a vrácení hodnoty (pro metodu get), takže by bylo nutné tento kód vepisovat ručně. Ve třídě *Class4* jsou metody vytvořeny stejným postupem jako ve třídě *Class3*, ale navíc byl u těchto metod zvolen stereotyp *<<property set>>* a *<<property get>>*. Generování zdrojového kódu proběhne bez problémů, těla metod již nejsou prázdná, ale jde o postup náročný na manuální práci.

Třída *Class5* slouží k demonstraci dalšího nedostatku nástroje Enterprise Architect. Pokud transformujeme třídu s metodami get a set, které jsme vytvořili zatržením volby *„Property“* nebo výběrem stereotypů *<<property get>>* a *<<property set>>,* do platformově závislého modelu, ze kterého pak generujeme zdrojový kód, je v tomto kódu v těle metod uveden místo názvu příslušného atributu výraz *<unknown>*. Když jsem informaci o tomto problému zaslal vývojářům Enterprise Architectu, tak mi odpověděli, že se jedná o známou chybu. Transformace v jazycích Java a C# neprovádí kopírování názvů atributů do těla get a set metod. Pokud zkusíme zdrojový kód generovat přímo ze tříd v platformově nezávislém modelu, pak zjistíme, že v kódu tato chyba není. Generovat zdrojový kód z platformově nezávislého modelu sice MDA povoluje [20], ale Enterprise Architect pro takové generování kódu nemá vytvořeny transformační skripty a výsledný kód obsahuje prvky platformové nezávislosti (např. netransformované obecné datové typy apod.).

Na třídu *Class6* byl aplikován ukázkový transformační skript z kapitoly 3.2.2, který vytváří metody get a set automatizovaně až při transformaci do platformově závislých modelů. Při tomto postupu nedochází k chybám s výrazem *<unknown>* a ani k chybám s prázdným tělem metody. Generované zdrojové kódy jsou uvedeny v příloze D. Zdrojový kód `Class5.java` vznikl generováním z modelu PIM a zdrojový kód `Class5a.java` z modelu PSM.

3.2.4.3 Generování WSDL modelu v EA

WSDL je jazyk pro popis rozhraní webových služeb. Více se tomuto jazyku věnuje kapitola 5.1.1. Enterprise Architect by měl umět z třídy, která reprezentuje rozhraní, vygenerovat model WSDL. Podle manuálu k Enterprise Architectu by měl být z tohoto rozhraní, které má definovanou operaci se vstupními parametry a návratovou hodnotou (obrázek 3.8), vygenerován WSDL model s balíky *Types, Messages, Services, Bindings* a *PortTypes* (obrázek 3.9).

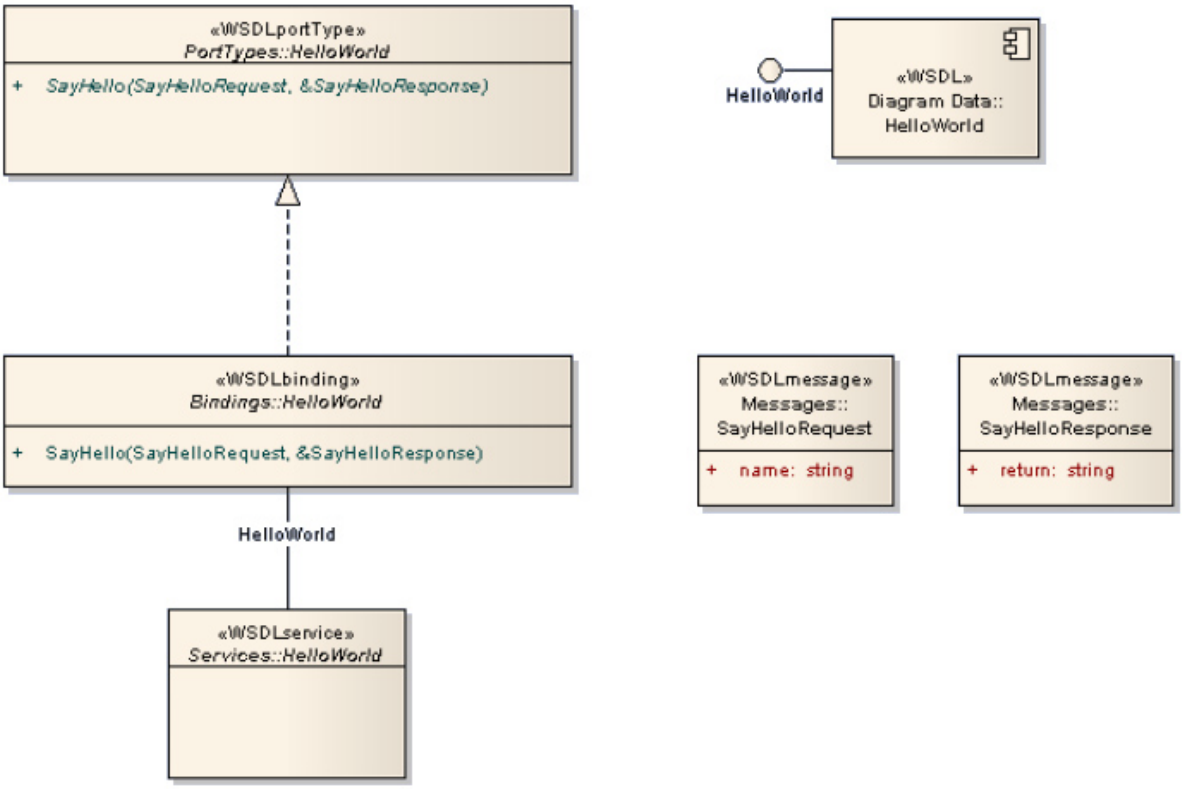


Obr. 3.8: Třída před transformací do WSDL modelu

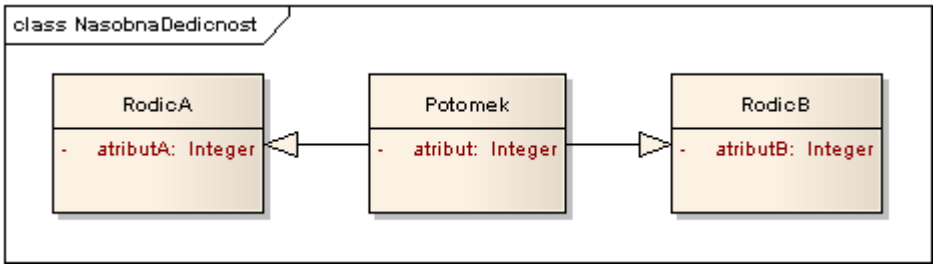
Při testu Enterprise Architectu verze 7.0 (build 813) i verze 7.1 (build 828) však byl vždy vygenerován jen jeden balík (*Types*) a ostatní balíky (*Messages, Services, Bindings* a *PortTypes*) chyběly. O chybě jsem opět informoval výrobce Enterprise Architectu. „Bug report“ jsem odeslal 20. dubna 2008 a odpověď mi přišla 22. dubna 2008. Vimal Kumar ze společnosti Sparx Systems mi poradil, abych rozhraní, které chci transformovat do WSDL modelu, nevytvářel běžným způsobem, tj. vložením třídy a zvolením stereotypu <<*interface*>>, ale abych přímo použil element rozhraní, který přetáhnu z toolboxu do diagramu. Pokud vytvořím rozhraní tímto novým způsobem, pak transformace do WSDL modelu funguje správně. Přesto tuto vlastnost Enterprise Architectu uvádím mezi chybami, protože pokud nástroj umožňuje dosažení jednoho cíle (tj. vytvoření stejného rozhraní) dvěma různými způsoby, pak by následné transformace neměly být závislé na zvoleném způsobu, který k vytvoření rozhraní vedl. „Bug report“ a odpověď naleznete v příloze A.

3.2.4.4 Násobná dědičnost

UML ve svých modelech nezabraňuje vytváření násobné dědičnosti, není k tomu ani důvod, existují jazyky, které násobnou dědičnost používají. Na druhou stranu jsou ale jazyky, kde násobná dědičnost neexistuje. Rozhodl jsem se tedy otestovat, jak se Enterprise Architect vypořádá s transformací platformově nezávislého modelu, který obsahuje násobné dědičnosti, do platformy, ve které násobná dědičnost neexistuje. Třídy určené k transformaci jsou uvedeny na obrázku 3.10. Jde o dvě rodičovské třídy a jednoho potomka. Po transformaci do platformově závislého modelu zůstala násobná dědičnost zachována (pro jazyk Java i C#). Transformované třídy jsou zobrazeny na obrázcích v příloze C.



Obr. 3.9: Generovaný WSDL model



Obr. 3.10: Násobná dědičnost

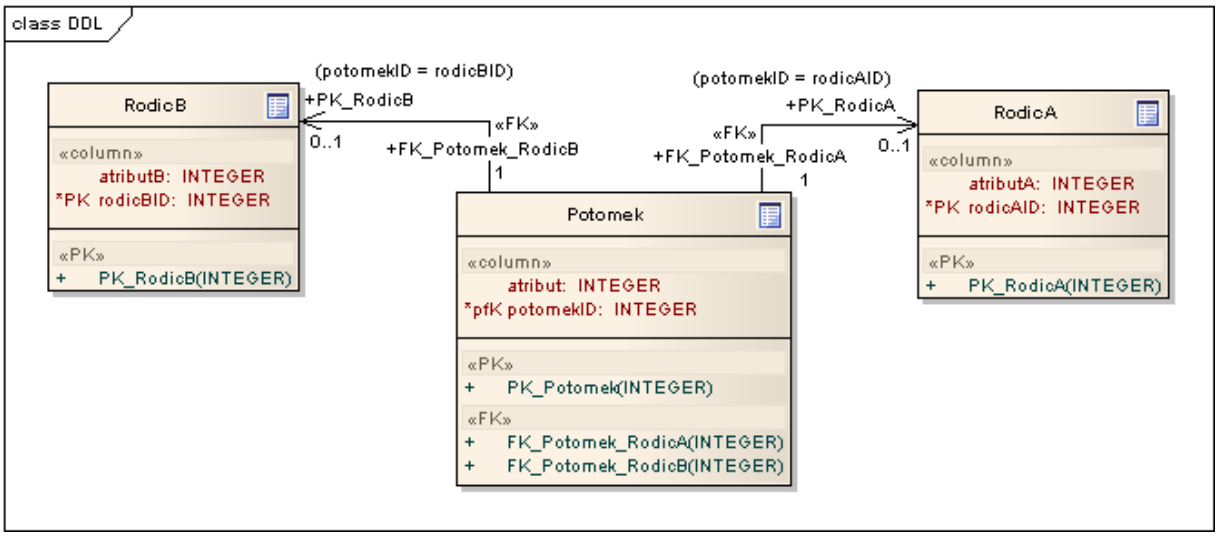
Pokud dále z těchto platformově závislých modelů generujeme zdrojový kód příslušného jazyka, dojde ve výsledném kódu k chybě. Třída potomka se snaží dědit z obou rodičovských tříd, to je však v daných jazycích nepřípustné a zdrojové kódy nepůjdou zkompilovat:

Java: `public class Potomek extends RodicA RodicB {…`

C#: `public class Potomek : RodicA, RodicB {…`

Při transformaci tříd do DDL modelu dojde také k chybě. V transformovaném modelu se objevuje špatná kardinalita, jedná se však o chybu v transformačním skriptu, která byla rozebírána v kapitole 3.2.4.1 a která nesouvisí přímo s problémem transformace násobné

dědičnosti. Pro transformaci do modelu DDL je použito upravené objektově-relační mapování typu „1:1“ [23]. Všechny entity reprezentující rodičovské třídy, ze kterých dědí třída daného potomka, mají stejný primární klíč. Tento model s transformovanou násobnou dědičností je na rozdíl od předchozích dvou modelů (Java, C#) použitelný. Model DDL je uveden na obrázku 3.11 a zdrojové kódy pro jazyk Java a C# jsou v příloze D.



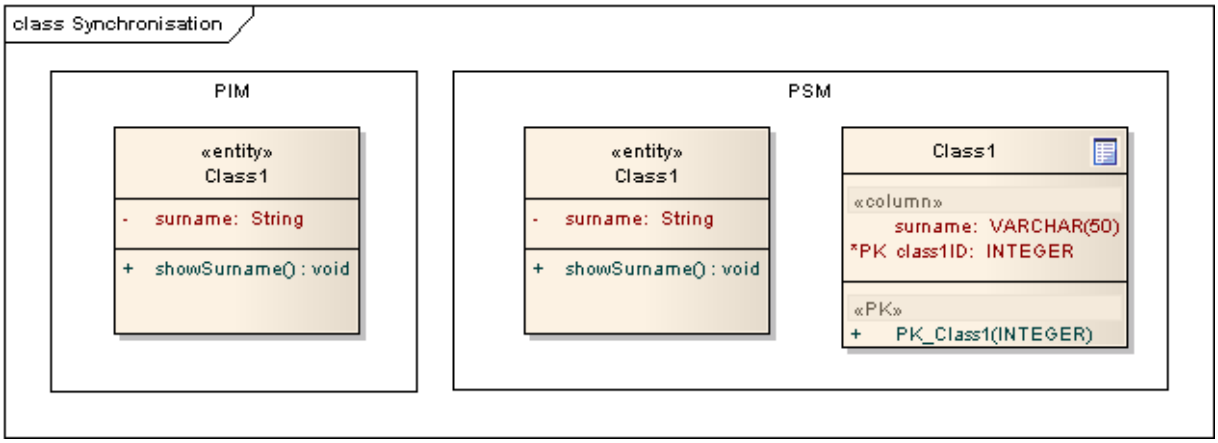
Obr. 3.11: Násobná dědičnost (DDL model)

3.2.4.5 Synchronizace modelů a kódu

Možné synchronizace mezi jednotlivými modely a zdrojovými kódy byly uvedeny v kapitole 3.2.1. Na tomto příkladu je ukázán nedostatek, ke kterému dochází při synchronizaci modelu s kódem. Na obrázku 3.12 jsou uvedeny dvě třídy a jedna entita DDL modelu. Třída *Class1* měla původně atribut *name* a operaci *showName()*. Z této třídy umístěné v modelu PIM byla vygenerována třída *Class1* (Java) a entita *Class1* (DDL) patřící do modelu PSM. Z modelu PSM byly dále vygenerovány zdrojové kódy. Následně byl u třídy *Class1* z modelu PIM změněn atribut *name* na atribut *surname* a změněna byla také metoda *showName()* na metodu *showSurname()*. Po provedení dalších transformací z modelu PIM do modelu PSM došlo ke správné změně v názvu atributu a metody i v modelu PSM. Problém však nastal při generování kódů. Zatímco vygenerovaný DDL skript byl upraven správně, u zdrojového kódu Java došlo k přidání další operace a dalšího atributu:

```
private String name;

private String surname;
```

Zamezit tomuto chování lze částečně pomocí volby *Tools – Options... – Attribute/Operations – On forward synch, prompt to delete code features not in model*. Pak je uživateli při synchronizaci nabídnuto, zda nechce smazat elementy, které jsou obsaženy v kódu a nejsou v modelu. Nedojde sice ke „zdvojení“ měněných atributů a metod, ale dojde k jejich smazání a vytvoření nových. Tím se ale ztratí například kód v těle upravovaných metod.



Obr. 3.12: Synchronizace PIM a PSM modelů

Pokud byla změna názvu provedena ve zdrojovém kódu, pak pomocí reverzního inženýrství (reverse engineering) byla správně promítnuta i do modelu PSM. Ve skutečnosti však nešlo o změnu názvu daného atributu v modelu, ale o vytvoření nového atributu a smazání atributu předchozího. Reverzní inženýrství ale vůbec nefungovalo u skriptů DDL, Enterprise Architect tuto možnost nepodporuje. Celé ukázky zdrojových kódů z tohoto příkladu jsou uvedeny v příloze D.

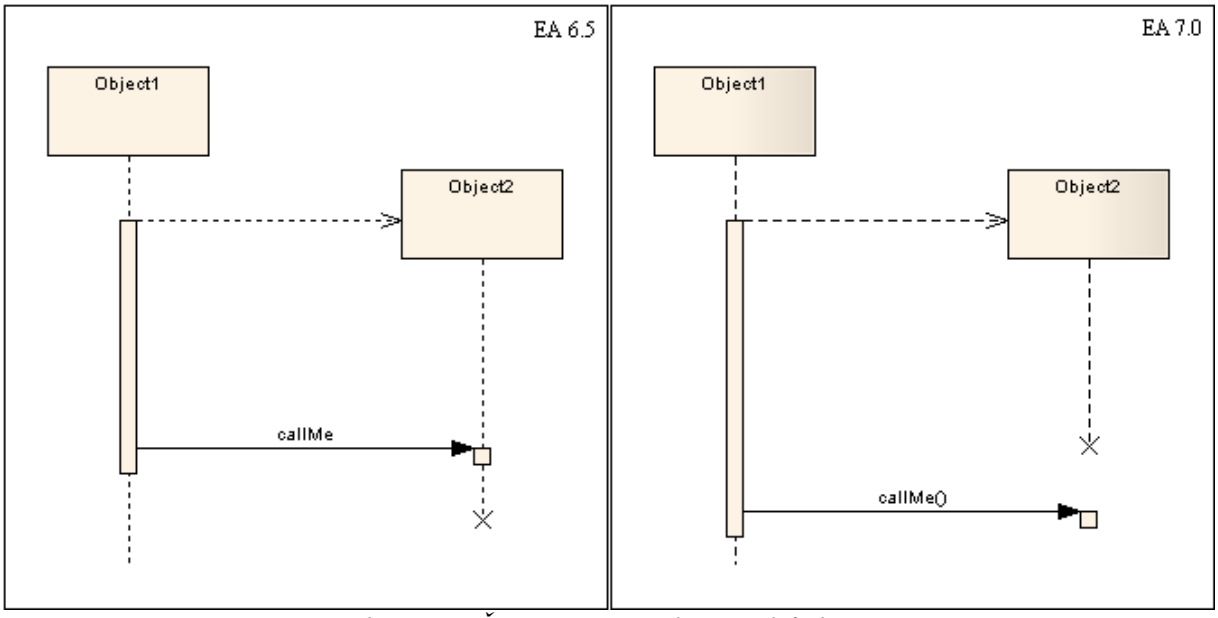
3.2.4.6 Názvy balíků

UML dává možnost pojmenování balíků velkými znaky včetně prvního znaku v názvu. Ale například v jazyce Java bývá zvykem psát balíky malými znaky [25]. Generovat v platformově závislých modelech nebo ve zdrojovém kódu názvy balíků tak, jak se na dané platformě používají, je jednoduchou záležitostí, přesto v Enterprise Architectu tato vlastnost chybí. Velikosti znaků v názvech balíků jsou po transformaci stejné jako v platformově nezávislém UML modelu, tedy i velké první písmeno v názvu balíku zůstane po transformaci opět velké. Tuto chybu jsem vývojářům oznámil ve svém třetím „bug reportu“. Odpověď přišla opět rychle (do dvou pracovních dnů). Společně jsme upravili transformační skripty tak, aby se velké znaky v názvech balíků při transformaci změnily na malé znaky. Upravený skript je v příloze B.

3.2.4.7 Další drobné nedostatky

K výčtu nedostatků nástroje Enterprise Architect ještě patří slabší podpora psaní scénářů. Scénáře lze totiž psát jen v prostém textu. Pro srovnání, například editor v nástroji Visual Paradigm umožňuje ve scénářích psát akce uživatele a odpovědi systému do oddělených sloupců, automaticky čísluje jednotlivé akce, umožňuje vkládat preconditions a post-conditions apod. Enterprise Architect umožňuje pouze zvolit typ scénáře (*Simple, Alternate, Basic Path, ...*).

Nepohodlná práce je také s „životní čárou“ (life line) u objektů v sekvenčním diagramu. Pokud nezmění uživatel výchozí nastavení, tak se každý nově vytvořený objekt automaticky ukončí značkou pro destrukci objektu. Tato značka se však nepřizpůsobuje ostatním elementům diagramu a nelze změnit její pozici. Chyba se týká verze 7.0, ale například ve verzi 6.5 k této chybě vůbec nedocházelo. Porovnání stejného diagramu z verze 7.0 a z verze 6.5 naleznete na obrázku 3.13. Výchozí nastavení lze změnit zrušením volby *Tools – Options.. – Sequence – CarbageCollect (auto delete)*. Poté již nově vytvořené objekty nejsou automaticky ukončovány. Také tuto chybu jsem nahlásil vývojářům Enterprise Architectu. Ti slíbili, že „bug report“ postoupí zodpovědným osobám a opravu vydají v dalších verzích programu.



Obr. 3.13: Špatná manipulace s "life line"

4 Případová studie: Informační systém jídelen

Tato kapitola se zabývá aplikací MDA na vývoj informačního systému školních jídelen JSSI (Jídelny Speciálních Škol Ivančice).

4.1 Popis informačního systému JSSI

Speciální školy Ivančice, pro které je systém jídelen navržen, jsou spojením několika školských zařízení: mateřská škola, základní škola, internát a dětský domov pro sluchově postižené žáky. Škola má více oddělených jídelen a kuchyní. Jídlo tedy není vydáváno pouze v jedné centrální jídelně, ale i v menších výdejnách (např. v dětském domově). Stejně i příprava jídla není centralizovaná v jedné kuchyni. Pro výdej a objednávání jídel se používají běžné papírové jídelní kupony. Díky charakteru školy má i jídelna svá specifika, například jídla se připravují i k večeri a snídani (kvůli internátu) a také o víkendech (kvůli dětskému domovu).

Informační systém školní jídelny by měl zlepšit fungování jídelny v oblasti výdeje a objednávání jídel. Také by měl poskytovat informace o množství potřebných surovin k přípravě objednaných jídel v jednotlivých výdejnách. Systém bude sloužit též k vytváření a publikování jídelníčků. Jídelní kupony by měly být nahrazeny elektronickými kartami.

Popis doménové oblasti, jednotlivých procesů a případů užití je uveden u modelů v následujících kapitolách.

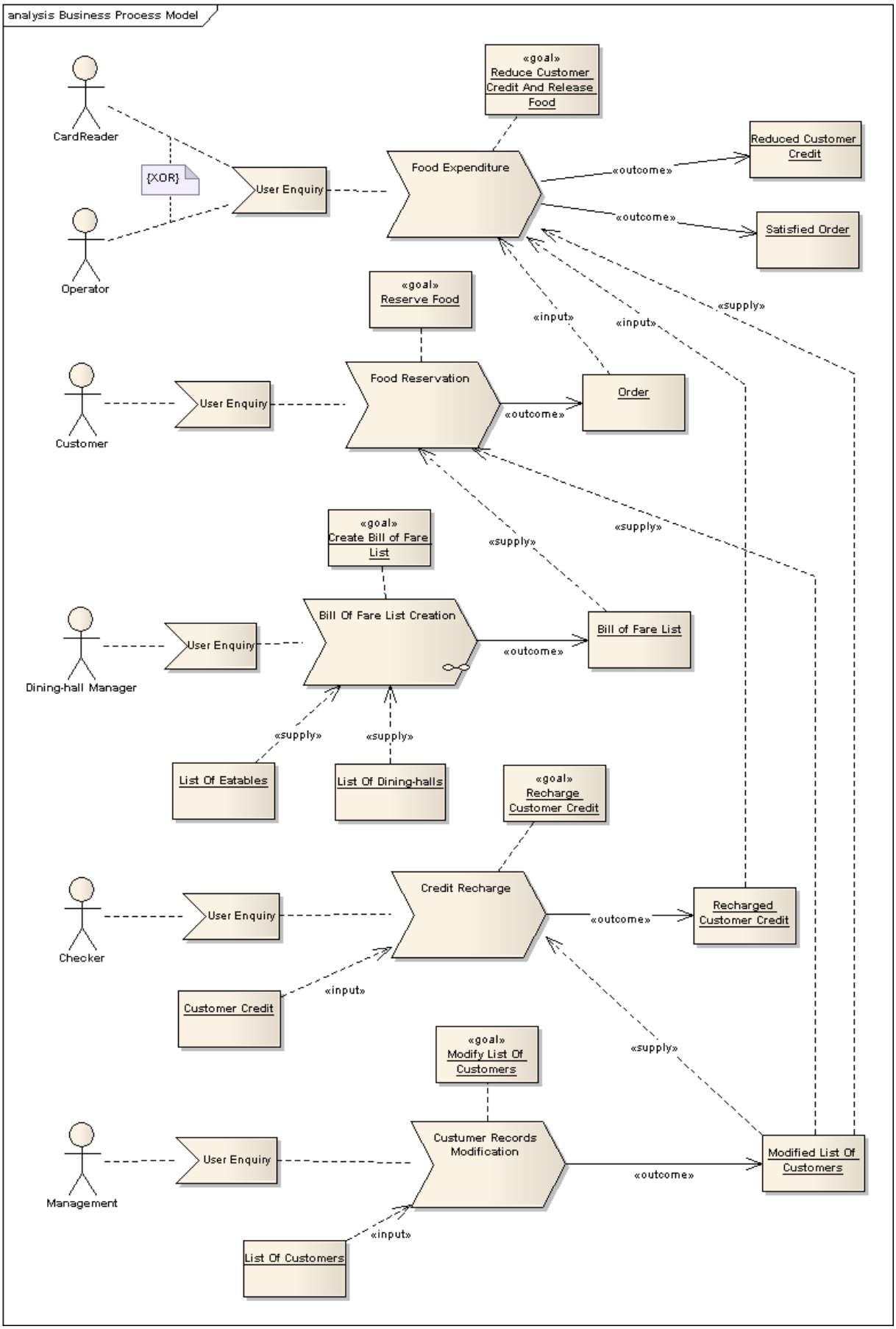
4.2 CIM (JSSI)

Počítačově nezávislý model (model nezávislý na počítačovém zpracování, Computation Independent Model) informačního systému školních jídelen obsahuje model procesů (Business Process Model) a doménový model (Domain Model).

4.2.1 Business Process Model (JSSI)

Business Process Model (BPM) znázorňuje procesy probíhající v jídejně speciálních škol Ivančice. Procesy jsou znázorněny na obrázku 4.1.

Vedení školy (*Management*) do systému vkládá zákazníky a modifikuje jejich údaje (*Customer Records Modification*). Vstupem tohoto procesu je seznam zákazníků (*List Of Customers*) a výstupem je změněný seznam zákazníků (*Modified List Of Customers*). Doplnění kreditu do účtu jednotlivých zákazníků (*Credit Recharge*) zahajuje pokladní (*Checker*).



Obr. 4.1: Business Process Model (JSSI)

Vstupem tohoto procesu je zákazníkův kredit (*Customer Credit*) a výstupem je doplněný zákazníkův kredit (*Recharged Customer Credit*). Proces doplnění kreditu také používá seznam zákazníků. Přípravu jídelníčků (*Bill Of Fare List Creation*) zahajuje vedoucí jídelen (*Dining-hall Manager*). Proces přípravy jídelníčků je složitější než ostatní procesy a tak je zvlášť modelován v kompozitním modelu, jehož diagram je uveden v příloze E. Než se vytvoří jídelníček, je nejprve potřeba pomocí základních potravin definovat jídla, která je možno v jídelně připravit. To zajišťuje proces definice jídla (*Food Definition*). Tento proces zahajuje vrchní kuchař (*Chef*). Z nadefinovaných jídel pak může vedoucí jídelny (*Dining-hall Manager*) sestavit menu (*Dishes*). Toto je zajištěno procesem definice menu (*Dishes Definition*). Nadefinovaná menu vedoucí jídelny vkládá do jídelníčku (*Bill Of Fare*). Vkládání menu do jídelníčku zajistí proces tvorba jídelníčku (*Bill Of Fare Creation*). Zákazník (strávník, *Customer*) používá jídelníčky, ze kterých si objednává jídla. To se děje v procesu rezervace jídla (*Food Reservation*). Výstupem tohoto procesu je objednávka (*Order*). Objednávka je uspokojena při výdeji jídla (*Food Expenditure*). Tento proces je zahájen čtečkou karet (*Card Reader*) a nebo operátorkou (*Operator*), která výdej jídla zadá manuálně, pokud si například strávník zapomene kartu. Výstupem procesu je vyřízená objednávka (*Satisfied Order*) a snížený kredit zákazníka (*Reduced Customer Credit*).

4.2.2 Doménový model (JSSI)

Doménový model (Domain Model) popisuje vztahy mezi entitami vystupujícími v informačním systému jídelen. Model je zachycen na diagramu z obrázku 4.2.

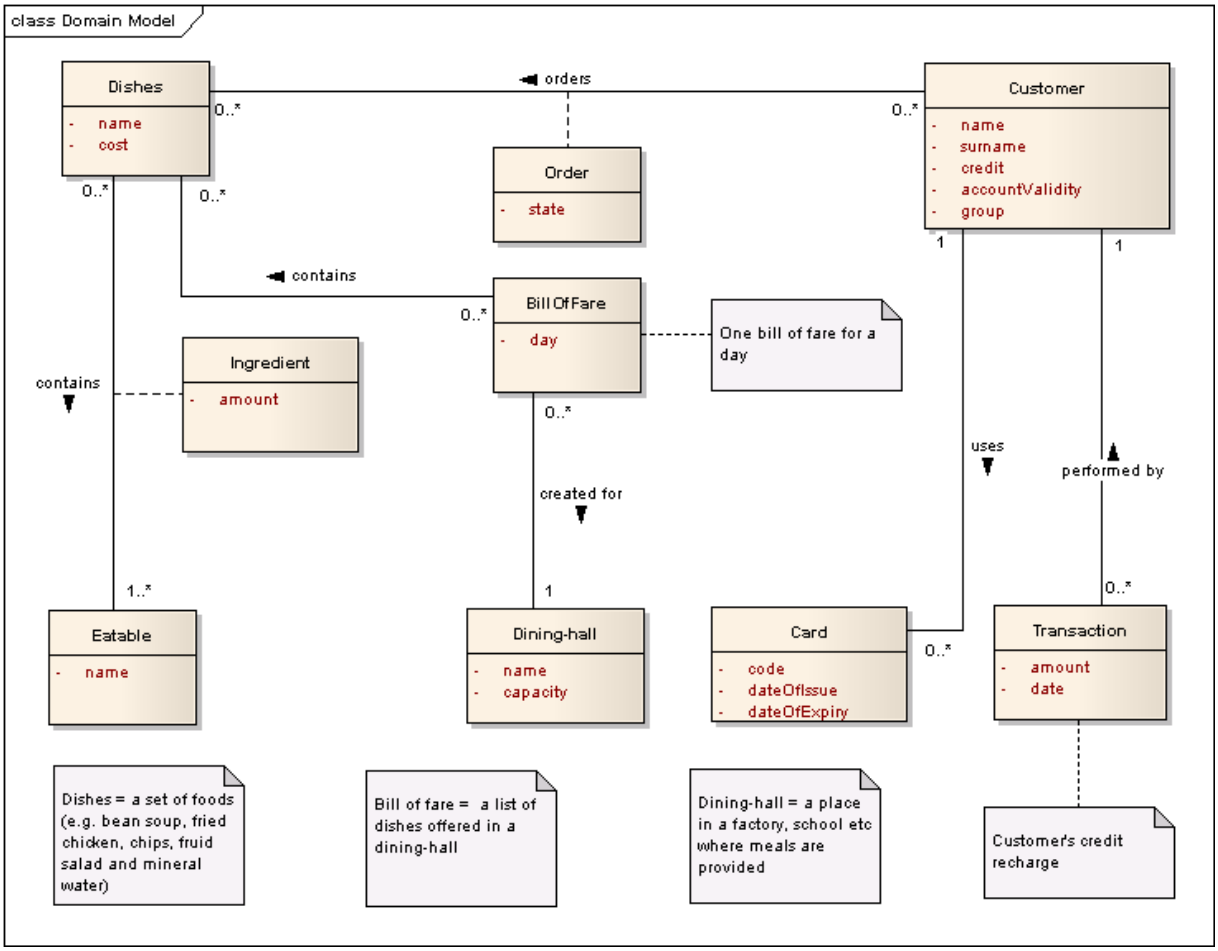
Potravina (*Eatable*) – reprezentuje libovolnou potravinu, kterou evidujeme v informačním systému a která slouží k přípravě jídla. Jedna potravina může být součástí více jídel.

Jídlo (*Dishes*) – reprezentuje jídlo, které se podává ve školní jídelně. Skládá se z jedné nebo více potravin. Jídlo je položkou jídelníčku a lze na něj vytvořit objednávku.

Ingredience (*Ingredient*) – reprezentuje vztah mezi jídlem (*Dishes*) a potravinou (*Eatable*). Obsahuje atribut množství (*amount*). Tento atribut se využije při zjišťování množství potravin potřebných k přípravě objednaných jídel.

Zákazník (*Customer*) – reprezentuje strávníka, který má nebo dříve měl právo stravovat se ve školní jídelně. Zákazník má své jméno (*name*), příjmení (*surname*) a kredit (*credit*). Atribut platnost účtu (*accountValidity*) určuje, zda má zákazník právo se v jídelně stravovat. Zákazníci se ihned po ukončení tohoto práva z informačního systému neodstraňují a po určitou dobu zde zůstávají kvůli statistickým výstupům. Zákazník také patří do určité skupiny (*group*). Podle příslušnosti ke skupině jsou zákazníkům počítány ceny za jídlo. Zákazníkovi může být vydáno

více karet (*Card*). Pokud kartu ztratí, je mu i nadále evidována a při jejím nálezu tak lze určit, komu patřila. Zákazník dále provádí transakce (*Transaction*).



Obr. 4.2: Doménový model (JSSI)

Karta (*Card*) – reprezentuje elektronickou kartu, kterou zákazník používá pro výdej jídla. Karta smí být přiřazena jen jednomu zákazníkovi. Karta má svůj kód (*code*), datum vydání (*dateOfIssue*) a datum ztráty platnosti (*dateOfExpiry*). Pokud je karta nahlášena jako ztracená nebo odcizená, je vyplněno datum ztráty platnosti.

Transakce (*Transaction*) – reprezentuje připsání kreditu zákazníkovi. Obsahuje připsanou částku (*amount*) a datum (*date*) provedení transakce.

Jídelna (*Dining-hall*) – reprezentuje výdejnu jídel, ve které jsou vydávána jídla objednaná tímto informačním systémem. Jídelna má svůj název (*name*) a kapacitu míst (*capacity*).

Jídelníček (*BillOfFare*) – reprezentuje seznam jídel (*Dishes*), která jsou k výdeji v daný den a v dané jídelně (*Dining-hall*). Den, po který jídelníček platí, je dán atributem den (*day*). Daný jídelníček platí jen pro jednu jídelnu.

4.3 PIM (JSSI)

Platformově nezávislý model (Platform Independent Model) informačního systému školních jídelen obsahuje model případu užití (Use Case Model), analytický model tříd (Analysis Class Model), návrhový model tříd (Design Class Model), diagramy aktivit (Activity Diagram), stavové diagramy (State Machine Diagram), sekvenční diagramy (Sequence Diagram), diagram nasazení (Deployment Diagram) a návrh uživatelského rozhraní (GUI).

4.3.1 Model případů užití (JSSI)

Model případů užití (Use Case Model) zachycuje potřeby uživatelů a znázorňuje interakce mezi uživatelem a systémem. Diagram je uveden na obrázku 4.3.

Zákazník (*Customer*) si vytváří objednávky (*Make Order*), ruší objednávky (*Cancel Order*), čte objednávky (*View Orders*), čte jídelníčky (*View Bill Of Fare*) a kontroluje si svůj kredit (*View Credit*).

Vedení školy (*Management*) eviduje zákazníky (*Manage Customers*).

Externí systém (*External System*) si čte jídelníčky (*View Order*).

Vrchní kuchař (*Chef*) si čte jídelníčky (*View Order*), eviduje potraviny (*Manage Eatables*) a jídla (*Manage Foods*), a čte souhrn objednávek (*Read Orders Summary*), ze kterého se dozvídá informace o množství objednaných porcí a potřebných potravin.

Vedoucí jídelny (*Dining-hall Manager*) vytváří (*Manage Bills Of Fare*) a čte jídelníčky (*View Bill Of Fare*), eviduje menu (*Manage Dishes*) a může určovat jejich ceny (*Set up Costs*).

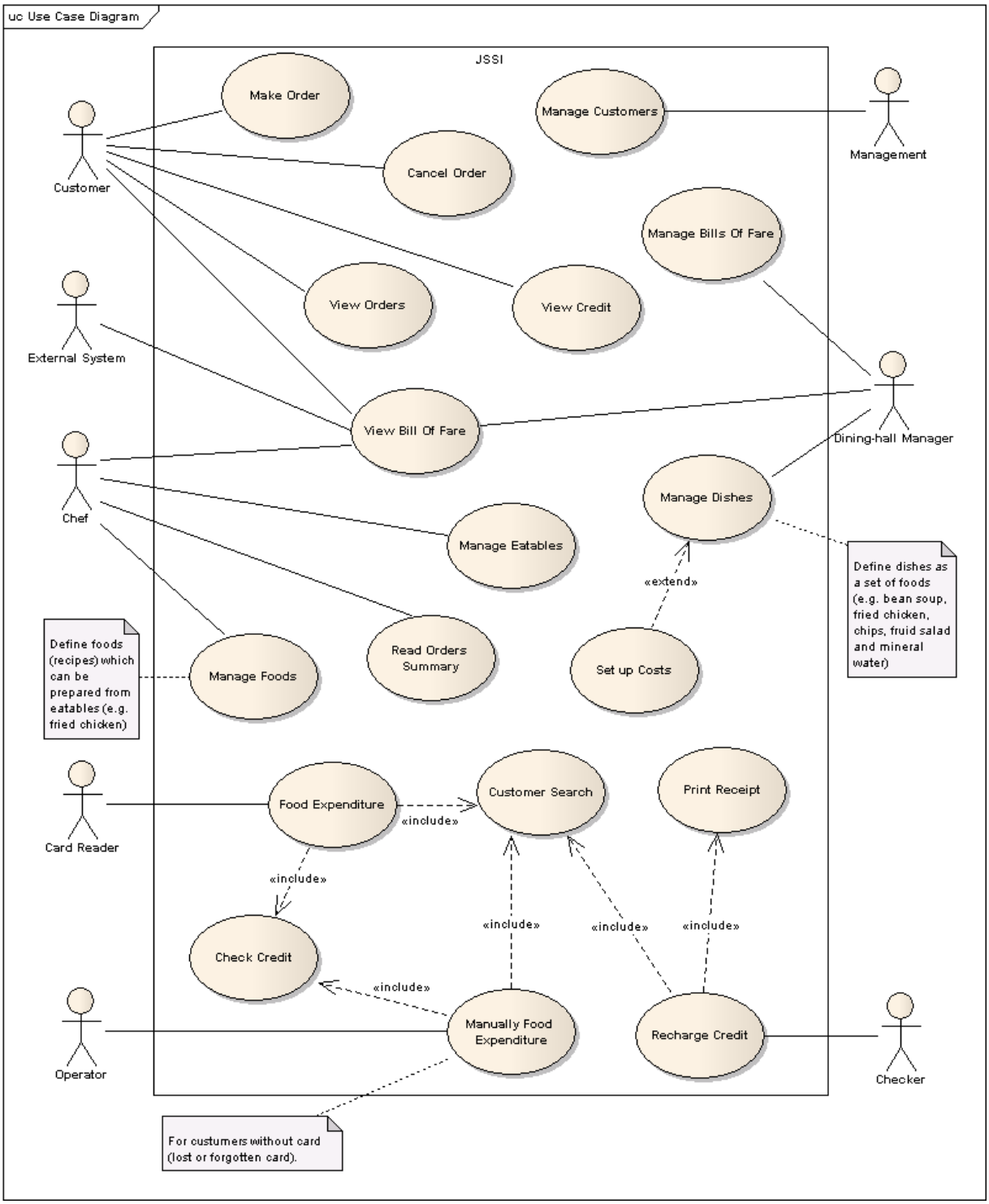
Čtečka karet (*Card Reader*) vydává jídlo zákazníkovi (*Food Expenditure*).

Výdejce (*Operator*) může vydat jídlo manuálním zadáním (*Manually Food Expenditure*). Toho se využije například v případě, že zákazník zapomněl kartu.

Pokladní (*Checker*) přijímá platby od zákazníků a doplňuje jejich kredit (*Recharge Credit*). Při příjmu platby také tiskne potvrzení (*Print Receipt*).

Případ užití výdej jídla (*Food Expenditure*) a manuální výdej jídla (*Manually Food Expenditure*) užívají další případ užití kontrola kreditu (*Check Credit*) a vyhledání zákazníka (*Customer Search*). Aby mohlo být zákazníkovi vydáno jídlo, musí mít dostatečný kredit. Kredit se odepisuje při výdeji jídla.

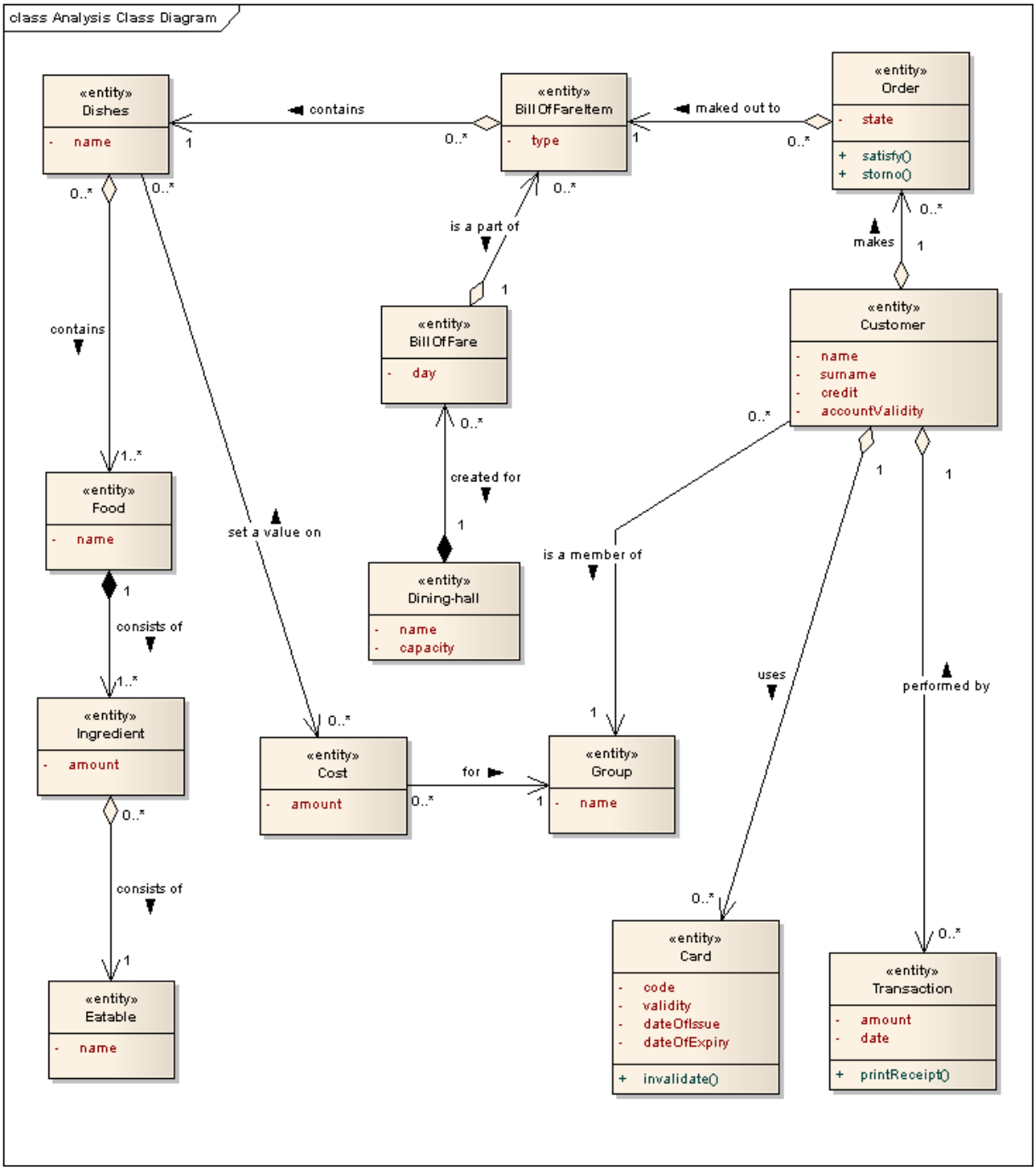
Případ užití vyhledání zákazníka (*Customer Search*) je použit také případem užití doplnění kreditu (*Recharge Credit*).



Obr. 4.3: Use Case Diagram (JSSI)

4.3.2 Analytický model tříd (JSSI)

Analytický model tříd (Analysis Class Model) zachycuje statickou strukturu systému. Model je znázorněn diagramem na obrázku 4.4.



Obr. 4.4: Analysis Class Diagram (JSSI)

Třídy v tomto modelu školních jídelen mají stereotyp <<*entity*>>, což znamená, že jde o entitní třídy. Entitní třídy představují objekty nesoucí informace a reprezentují uložená data. Analytický model je odstíněn od návrhových detailů. Neobsahuje například datové typy u atributů a ani nemá definovány typy návratových hodnot u operací. V analytickém modelu jsou ale zpřesněny vazby mezi třídami (směr, kompozice, agregace). Popis většiny z tříd by kopíroval popsané entity v doménovém modelu z balíku CIM. Proto jsou v následujícím textu uvedeny popisy tříd a elementů, které se od doménového modelu liší a nebo mají svá specifika vztažená k analytickému modelu.

K zpřesnění došlo u tříd reprezentujících menu, jídlo, potravinu a ingredienci. Třída potravina (*Eatable*) reprezentuje potravinu, tak jak byla definována v doménovém modelu. Třída ingredience (*Ingredient*) reprezentuje určité množství dané potraviny. Z ingrediencí se skládá jídlo (*Food*). Třída menu (*Dishes*) je kolekcí jídel (*Food*) a má definován svůj název. Jedno menu může mít určeno více cen (*Cost*), které jsou vždy vztaženy k určité skupině zákazníků (*Group*).

Ke změně došlo také u jídelen, jídelníčků a položek jídelníčků. Jídelny (*Dining-hall*) obsahují jídelníčky (*BillOfFare*). Jídelníček je kolekcí položek jídelníčku (*BillOfFareItem*) a je vypsán na určitý den (*day*). Položka jídelníčku obsahuje právě jedno menu. Atribut typ (*type*) u položky jídelníčku určuje o jaký druh denního jídla jde (snídaně, svačina, oběd, večeře, ...). Na položku jídelníčku mohou zákazníci (*Customer*) vytvářet objednávky (*Order*). Stav objednávky je uveden v atributu stav (*state*). Každý zákazník je členem nějaké skupiny (*Group*). Tento vztah slouží k zjištění ceny, kterou zákazník zaplatí za vydanou porci (tj. za dané menu). Různou cenu totiž platí žáci, učitelé, důchodci, apod., přitom zákazník je členem právě jedné ze skupin.

4.3.3 Návrhový model tříd (JSSI)

Návrhový model tříd (Design Class Model) stejně jako analytický model tříd znázorňuje statickou strukturu systému. Oproti analytickému modelu však obsahuje další úpravy a návrhové detaily. Návrhový model je zachycen na obrázku 4.5. Kvůli rozměru stránky byl vložen diagram, který nezobrazuje atributy a operace tříd. Plný diagram je k dispozici na přiloženém CD. V následujícím textu jsou popsány prvky návrhového modelu, které se liší od modelu analytického.

V modelu jsou použity následující stereotypy tříd:

- **<<*entity*>>** - tímto stereotypem jsou označeny třídy, které reprezentují objekty nesoucí informace, uložená data.

- ***<<control>>*** - označuje třídy, které provádějí určitou činnost. Třídy tvořící výkonnou logiku systému.
- ***<<boundary>>*** - označuje třídy, které reprezentují rozhraní systému. Tyto třídy zpřístupňují aktérům funkce systému.

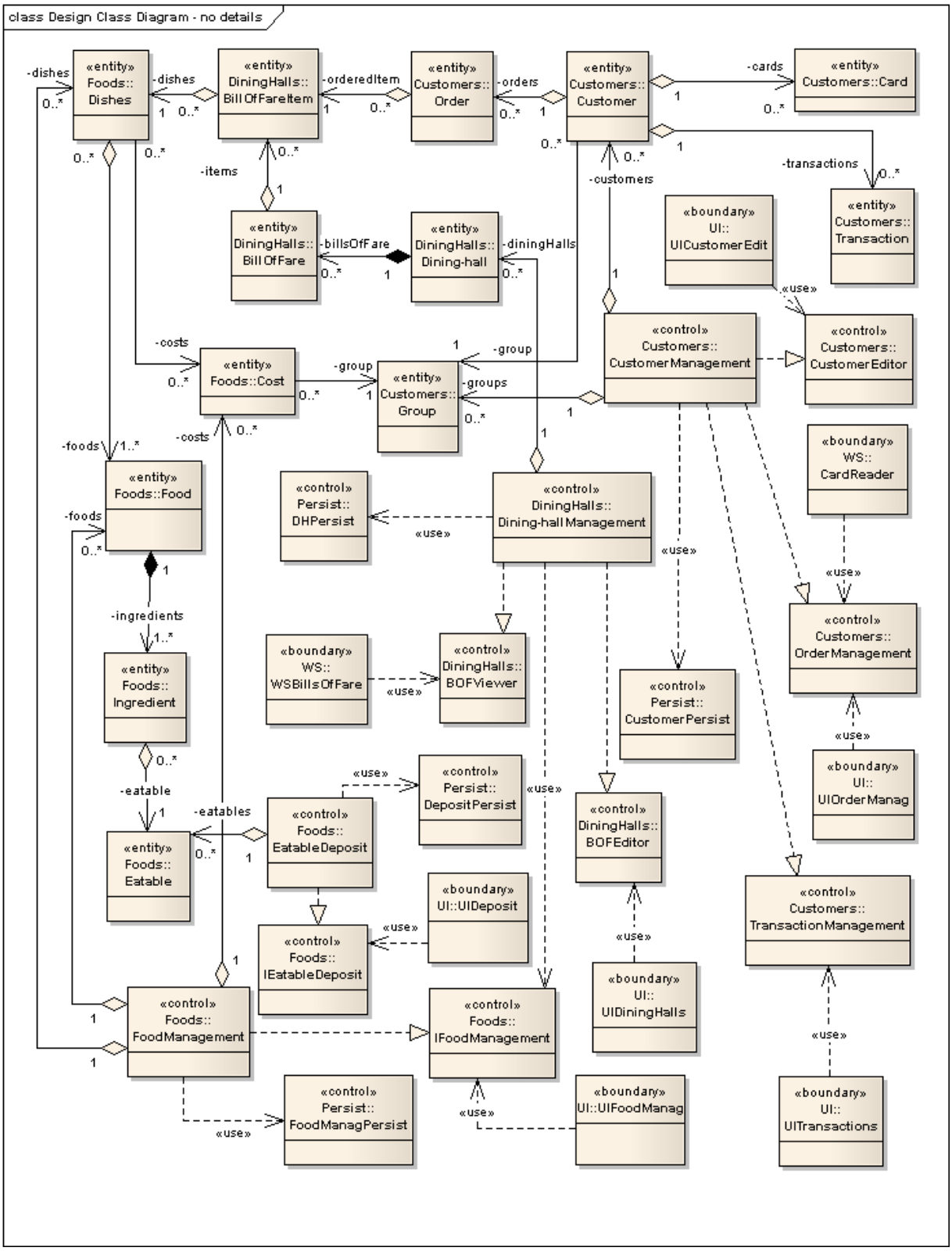
Třídy návrhového modelu jsou rozděleny do balíků. Příslušnost k danému balíku je uvedena před názvem každé třídy (např. *Customer::*). V modelu je celkem šest balíků (*Customers, Foods, DiningHalls, Persist, UI* a *WS*). Podrobněji jsou tyto balíky popsány v kapitole 4.3.7.

Třídy návrhového modelu jsou doplněny také o typy atributů a návratových hodnot operací. Model obsahuje i parametry operací a jejich datové typy. Lze však (podobně jako u jiných elementů modelu) nastavit, zda tyto elementy v diagramech zobrazovat. Protože je tento návrhový model součástí platformově nezávislého balíku, jsou i datové typy nezávislé na platformě (common types).

Model je dále doplněn i o metody getter a setter. Bylo by ale možné všechny tyto metody generovat až při transformaci tříd do platformově závislého modelu. Díky synchronizaci sekvenčních diagramů s diagramem tříd se však hodilo některé metody getter a setter použít už v návrhovém modelu. Více o možnostech synchronizace mezi modely je uvedeno v kapitole 3.2.1.2.

V třídách návrhového modelu nejsou atributy reprezentující kontejnery. Tyto kontejnery jsou modelovány pomocí asociací. V Enterprise Architectu vytvoříme asociace s kardinalitou 0..* nebo 1..*, určíme orientace vazby a pojmenujeme role. Název role pak bude i názvem kontejneru. Při generování zdrojového kódu jsou automaticky vytvořeny i modelované kontejnery.

Pokud třída implementuje nějaké rozhraní, pak metody tohoto rozhraní v implementující třídě uvedeny nejsou. Informace, že v této třídě metody mají být implementovány, je obsažena ve vazbě mezi rozhraním a implementující třídou. Při generování zdrojového kódu jsou tyto metody automaticky doplněny.

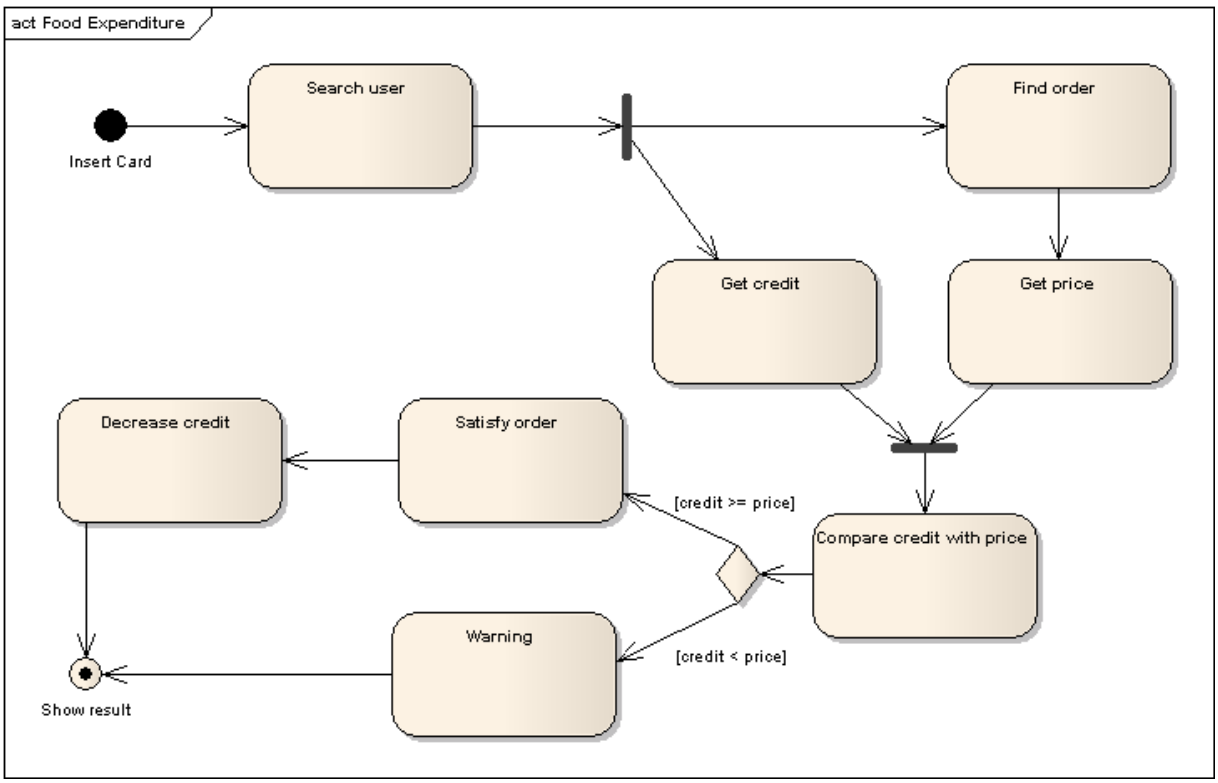


Obr. 4.5: Design Class Diagram (bez detailů)

4.3.4 Diagram aktivit (JSSI)

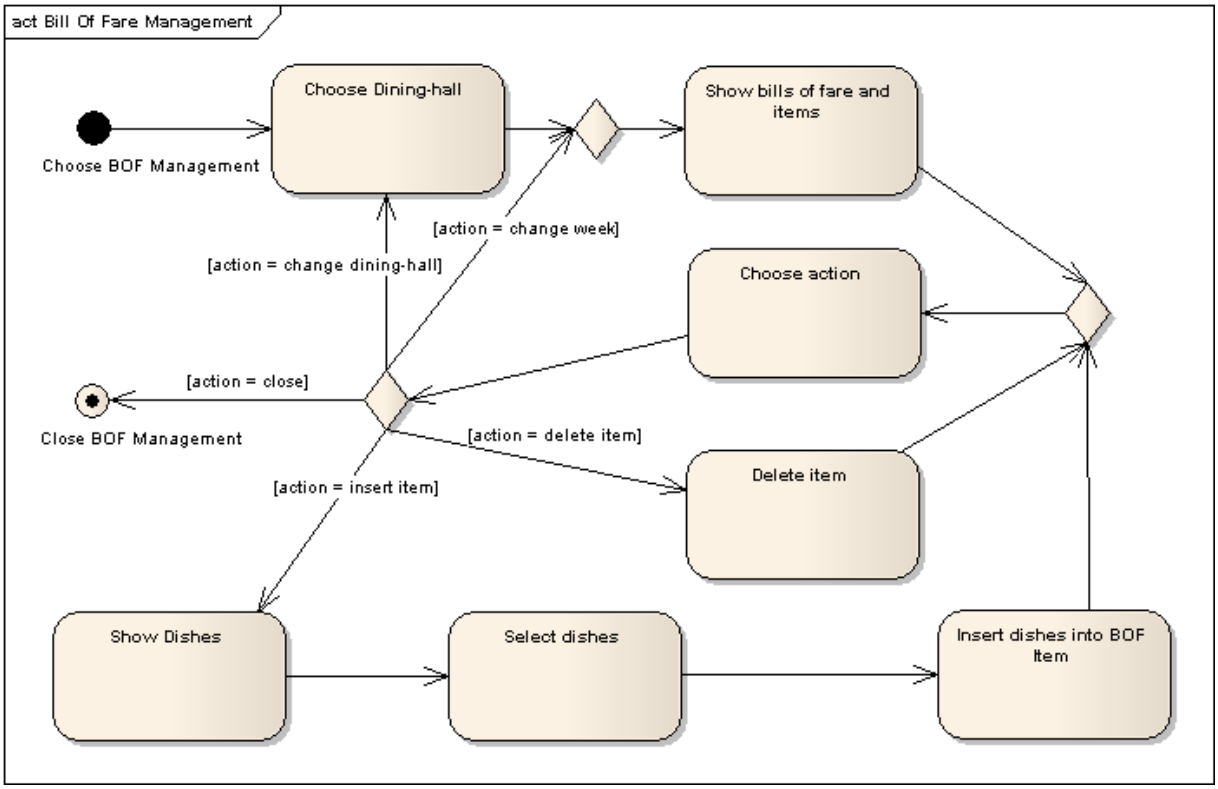
Diagramy aktivit (Activity Diagram) zobrazují toky činností v informačním systému školních jídelen. Diagramy dokumentují a popisují vybrané případy užití. Namodelovány pomocí diagramu aktivit jsou případy užití editace jídelníčků (*Manage Bills Of Fare*) a výdej jídla (*Food Expenditure*).

Diagram aktivit popisující případ užití výdej jídla je uveden na obrázku 4.6. Případ užití je zahájen vložením karty (*Insert Card*). Následuje vyhledání zákazníka (*Search user*), kde zákazník je hledán podle čísla použité karty. Diagram dále obsahuje dvě paralelní větve. V jedné větvi dojde ke zjištění výše kreditu zákazníka (*Get credit*). Ve druhé větvi je nejprve nalezena objednávka (*Find order*) a podle skupiny zákazníka je pak určena cena za danou objednávku (*Get price*). Po provedení obou paralelních větví následuje akce porovnání ceny objednávky a výše kreditu (*Compare credit with price*). Jestliže je kredit vyšší nebo roven ceně objednávky (*credit >= price*), následuje aktivita vyřízení objednávky (*Satisfy order*) a poté aktivita snížení kreditu (*Decrease credit*). Pokud je kredit nižší než cena objednávky (*credit < price*), pak dojde upozornění (*Warning*) a objednávka vyřízena není. Případ užití končí výpisem výsledku operace (*Show result*).



Obr. 4.6: Activity Diagram - výdej jídla

Diagram aktivit popisující případ užití editace jídelníčků je uveden na obrázku 4.7. Případ užití je zahájen výběrem funkce pro editaci jídelníčku (*Choose BOF Management*). Na začátku tohoto případu užití si uživatel vybere jídelnu, ve které chce jídelníčky editovat (*Choose Dining-hall*). Po výběru jídelny se uživateli objeví seznam jídelníčků a jejich položek pro aktuální týden (*Show bills of fare and items*). Uživatel si vybere akci, pomocí které chce pokračovat dále (*Choose action*). Pokud se jedná o výběr smazání položky (*action = delete item*), proběhne akce smazání vybrané položky jídelníčku (*Delete item*). Při výběru akce vložení položky (*action = insert item*) dojde k aktivitě zobrazení seznamu menu (*Show Dishes*). Následuje aktivita výběru menu pro vložení (*Select Dishes*) a pak samotné vložení vybraných menu do jídelníčku (*Insert dishes into BOF Item*). Pokud uživatel vybere akci změna vybrané jídelny (*action = change dining-hall*), může změnit jídelnu, ve které chce editovat jídelníčky. Výběr akce změna týdne (*action = change week*) pak vede k zobrazení jídelníčku pro jiný týden. Případ užití je ukončen zavřením editace jídelníčků (*Close BOF Management*).

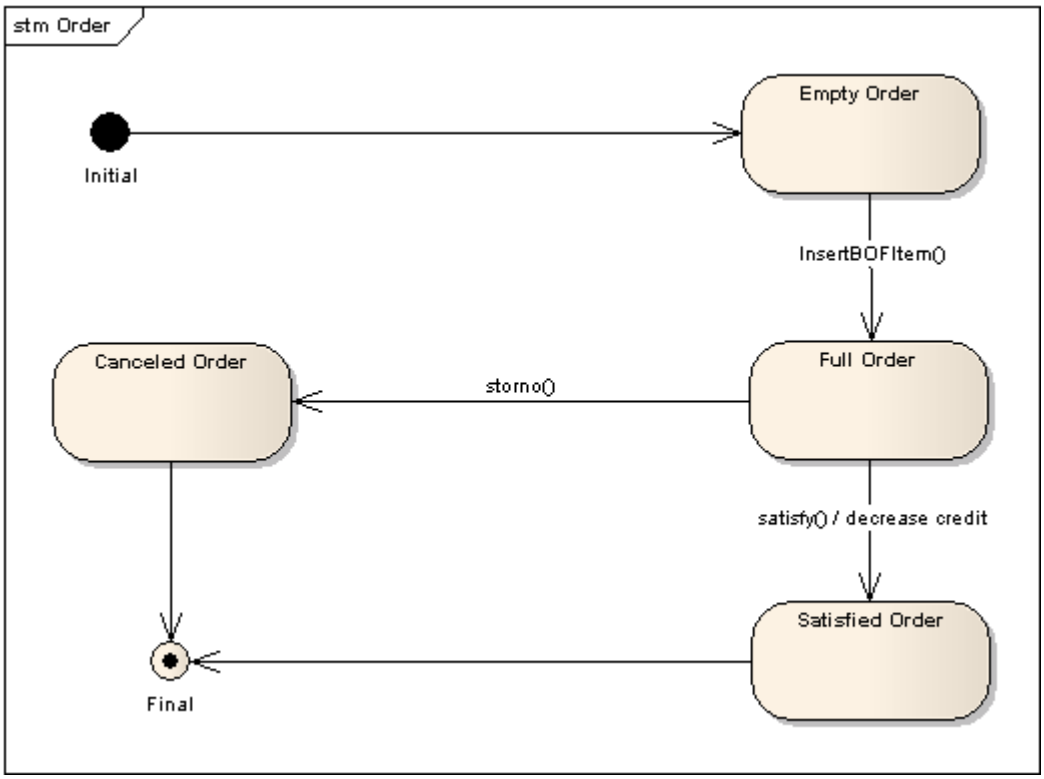


Obr. 4.7: Activity Diagram - editace jídelníčků

4.3.5 Stavový diagram (JSSI)

Stavový diagram (State Machine Diagram) představuje graf stavů a přechodů a popisuje odezvy objektů. Pomocí ukázkového stavového diagramu jsou modelovány stavy objektu objednávka. Tento diagram je uveden na obrázku 4.8.

Objekt reprezentující objednávku je po vytvoření ve stavu prázdná objednávka (*Empty Order*). Po události vložení položky jídelníčku do objednávky (*InsertBOFItem()*) přejde objednávka do stavu plné objednávky (*Full Order*). Pokud strávník zruší svoji objednávku událostí zrušení (*storno()*), přejde plná objednávka (*Full Order*) do stavu zrušená objednávka (*Canceled Order*). Pokud je plná objednávka (*Full Order*) úspěšně vyřízena událostí vyřízení (*satisfy()*), přejde do stavu vyřízená objednávka (*Satisfied Order*) a zároveň je vyvolána akce snížení kreditu u zákazníka (*decrease credit*).



Obr. 4.8: State Machine Diagram - objednávka

4.3.6 Sekvenční diagram (JSSI)

Sekvenční diagramy (Sequence Diagram) znázorňují interakce mezi objekty v informačním systému školních jídelen. Diagramy ukazují, jak spolu objekty komunikují v čase. Čas na diagramu postupuje po vertikální ose směrem dolů. Následující sekvenční diagramy pomocí

znázorněných interakcí popisují činnosti v případech užití editace jídelníčků a poskytování jídelníčků externím systémům pomocí webové služby.

Kvůli kompaktnosti je v textu na obrázku 4.9 uveden diagram znázorňující vytvoření nové položky jídelníčku (*Insert Bill of Fare Item*). Sekvenční diagramy modelující poskytování jídelníčků webovou službou jsou popsány v kapitole 5.2.2. Další sekvenční diagramy jsou uvedeny v příloze E.

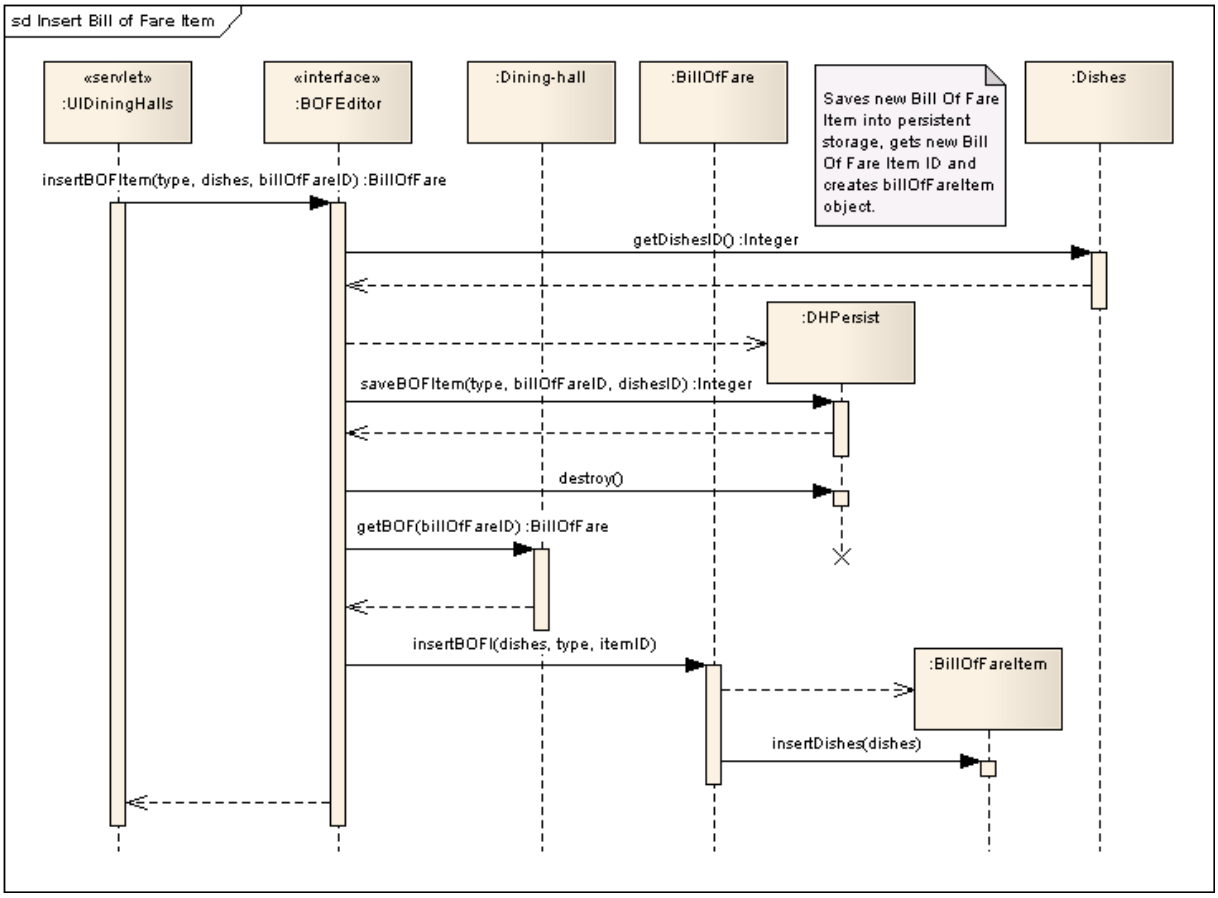
Vytvoření nové položky zahajuje instance třídy *UIDiningHalls*, která volá pomocí metody *insertBOFItem* instanci třídy *BOFEditor*. Přitom předává druh denního jídla (*type*), vkládané menu (*dishes*) a číslo jídelníčku (*billOfFareID*), pro který se nová položka vytváří. Instance třídy *BOFEditor* si zjistí id vkládaného menu zavoláním metody *getDishesID*. Následuje vytvoření instance třídy *DHPersist*, která umožní uložit data do perzistentního úložiště. Na této instanci se zavolá metoda *saveBOFItem* a předají se parametry druh denního jídla (*type*), číslo jídelníčku (*billOfFareID*) a číslo vkládaného menu (*dishesID*). Jako návratovou hodnotu dostaneme identifikační číslo nové položky jídelníčku. Instanci třídy *DHPersist* již dále nepotřebujeme, takže ji můžeme uvolnit z paměti. Instance třídy *BOFEditor* pak volá metodou *getBOF* instanci třídy *Dining-hall*, aby získala jídelníček s daným id jídelníčku. Na instanci jídelníčku *BillOfFare* pak zavolá metodu *insertBOFI*, která způsobí vytvoření instance třídy *BillOfFareItem*, do které se vloží menu (*dishes*). Nová položka jídelníčku je tedy vytvořena, vložena do jídelníčku, uložena v perzistentním úložišti a je možné s ní dále pracovat.

Sekvenční diagram *Show All Dining-halls* znázorňuje zobrazení všech jídelen. Jde o první činnost, která je provedena v modelovaném případu užití. Nejprve je vytvořena instance třídy *BOFEditor* a následně *DHPersist*, která načte data jídelen z perzistentního úložiště. Následuje vytvoření instancí jídelen, ze kterých jsou získány informace o daných jídelnách. Tyto informace pak mohou být vypsány uživateli.

Další činností, kterou znázorňuje sekvenční diagram *Show Bill Of Fare Items*, je zobrazení všech položek jídelníčku. Správce jídelníčků vytvoří instanci třídy *DHPersist*, která nahraje data z perzistentního úložiště. Následuje vytvoření instance jídelníčku. Správce jídel vytvoří instance menu a správce jídelníčků vytvoří položky jídelníčku, do kterých daná menu vloží. Jídelníček je poté vložen do jídelny a je možné s ním dále pracovat, například vypsat uživateli informace o položkách jídelníčku.

Sekvenční diagram *Show All Dishe*s znázorňuje zobrazení všech menu. Správce jídel vytvoří instanci třídy *FoodManagPersist*, která je zodpovědná za načtení dat z perzistentního úložiště. Po načtení dat jsou vytvořeny instance třídy Dishes. Z těchto instancí získáme potřebné informace, které mohou být vypsány uživateli.

Sekvenční diagram *Delete Bill of Fare Item* znázorňuje smazání položky jídelníčku. Nejprve jsou prohledány jídelny a jejich jídelníčky za účelem nalezení položky určené ke smazání. Po nalezení je tato položka (instance třídy *BillOfFareItem*) uvolněna z paměti. Následuje vytvoření instance třídy *DHPersist*, která se postará o smazání položky jídelníčku i z perzistentního úložiště dat.



Obr. 4.9: Sequence Diagram - Insert Bill of Fare Item

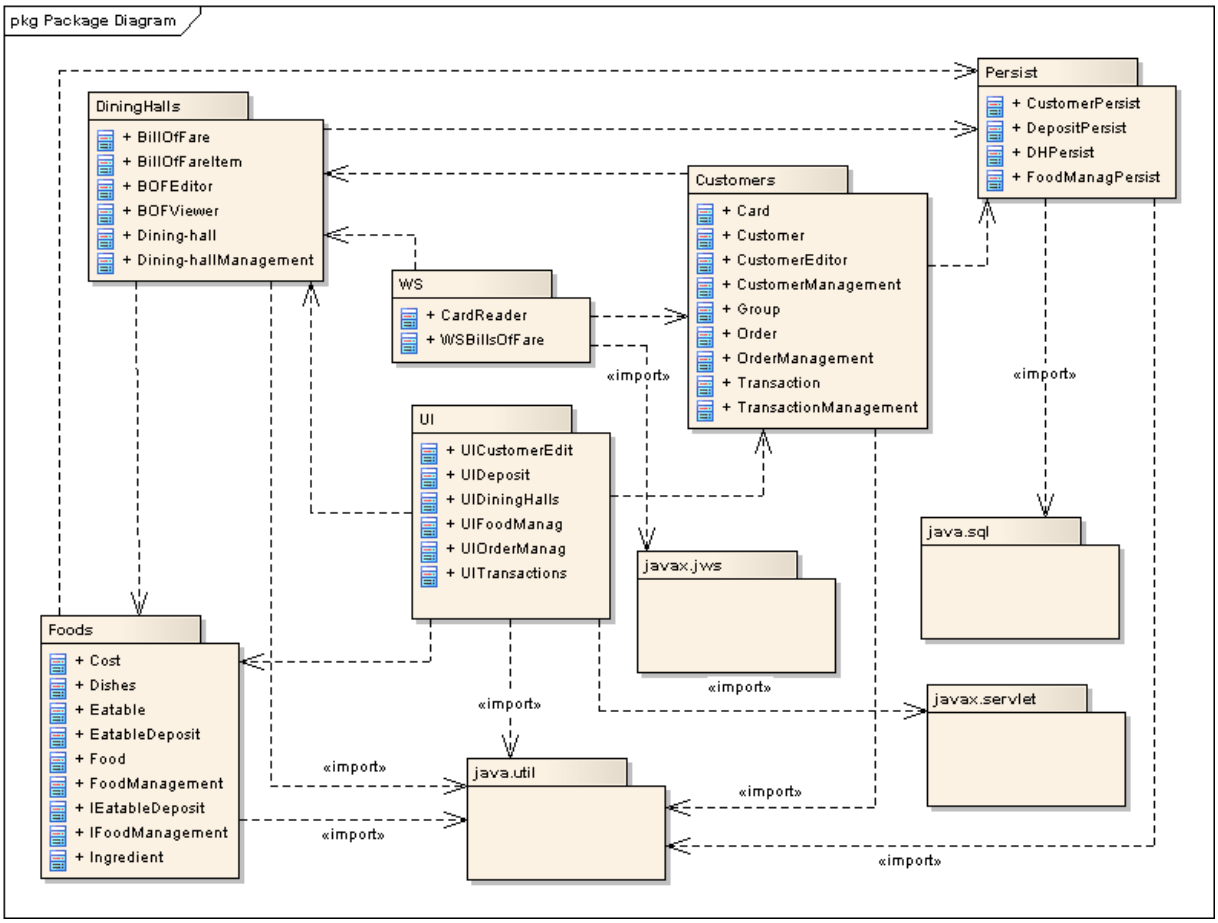
4.3.7 Diagram balíků (JSSI)

Diagram balíků (Package Diagram) znázorňuje rozdělení tříd do balíků a vzájemné závislosti mezi balíky. Diagram je uveden na obrázku 4.10. Model informačního systému školních jídelen obsahuje šest balíků:

- *DiningHalls* – sdružuje třídy, které souvisejí s jídelnami a jídelníčky.
- *Customers* – sdružuje třídy, které mají vztah k zákazníkovi.
- *Foods* – sdružuje třídy, které souvisejí s jídlem a potravinami.

- *Persist* – sdružuje třídy, které se starají o perzistentní ukládání dat.
- *UI* – sdružuje třídy uživatelských rozhraní.
- *WS* – sdružuje třídy reprezentující rozhraní webových služeb.

V diagramu jsou ještě uvedeny použité balíky Java API.

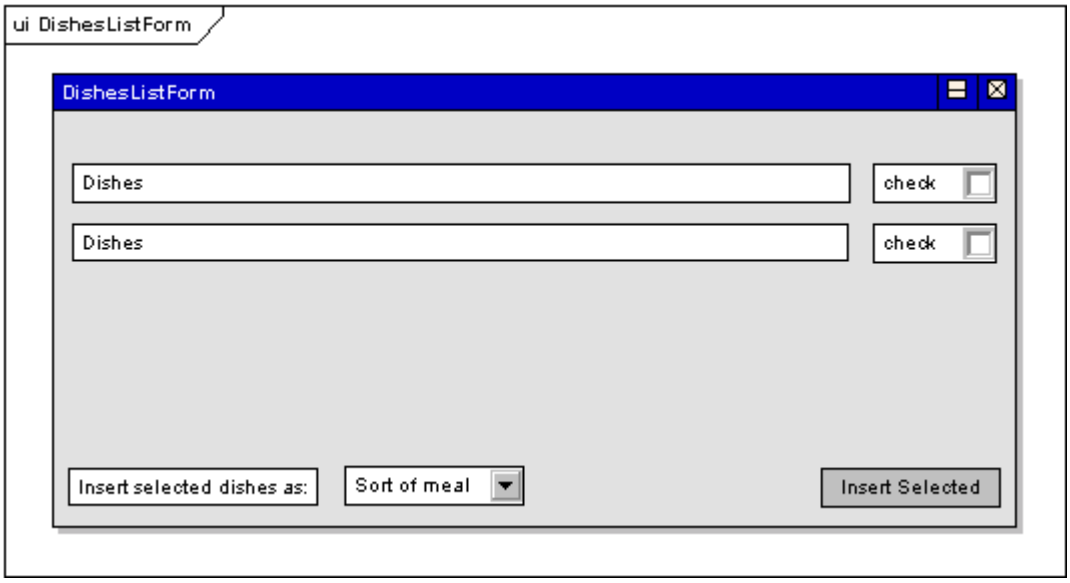


Obr. 4.10: Package Diagram (JSSI)

4.3.8 Grafické uživatelské rozhraní (JSSI)

Diagram grafického uživatelského rozhraní (GUI) zobrazuje formuláře sloužící ke komunikaci s uživateli informačního systému školních jídelen. Návrhy formulářu jsou uvedeny na obrázku 4.11 a v příloze E. Navržená rozhraní nejsou svázána s konkrétní platformou a Enterprise Architect ani neumožňuje z tohoto návrhu generovat zdrojové kódy desktopových či webových aplikací. Nejde tedy o klasický „GUI Builder“, ale pouze o vizualizaci formulářů grafického rozhraní. Tyto návrhy nám mohou pomoci například

při analýze uživatelských požadavků na systém. Formulář *DishesListForm* (z obrázku 4.11) je součástí rozhraní, které používá vedoucí jídelny k editaci jídelníčků, a slouží k vložení menu do jídelníčku. Tento formulář je vyvolán z formuláře *BOFEditorForm* (v příloze E), který vedoucímu jídelny zobrazuje jídelníčky pro vybranou jídelnu a vybraný týden. Menu lze pomocí tohoto dialogového okna do jídelníčku vkládat a z jídelníčku je mazat.



Obr. 4.11: Návrh GUI

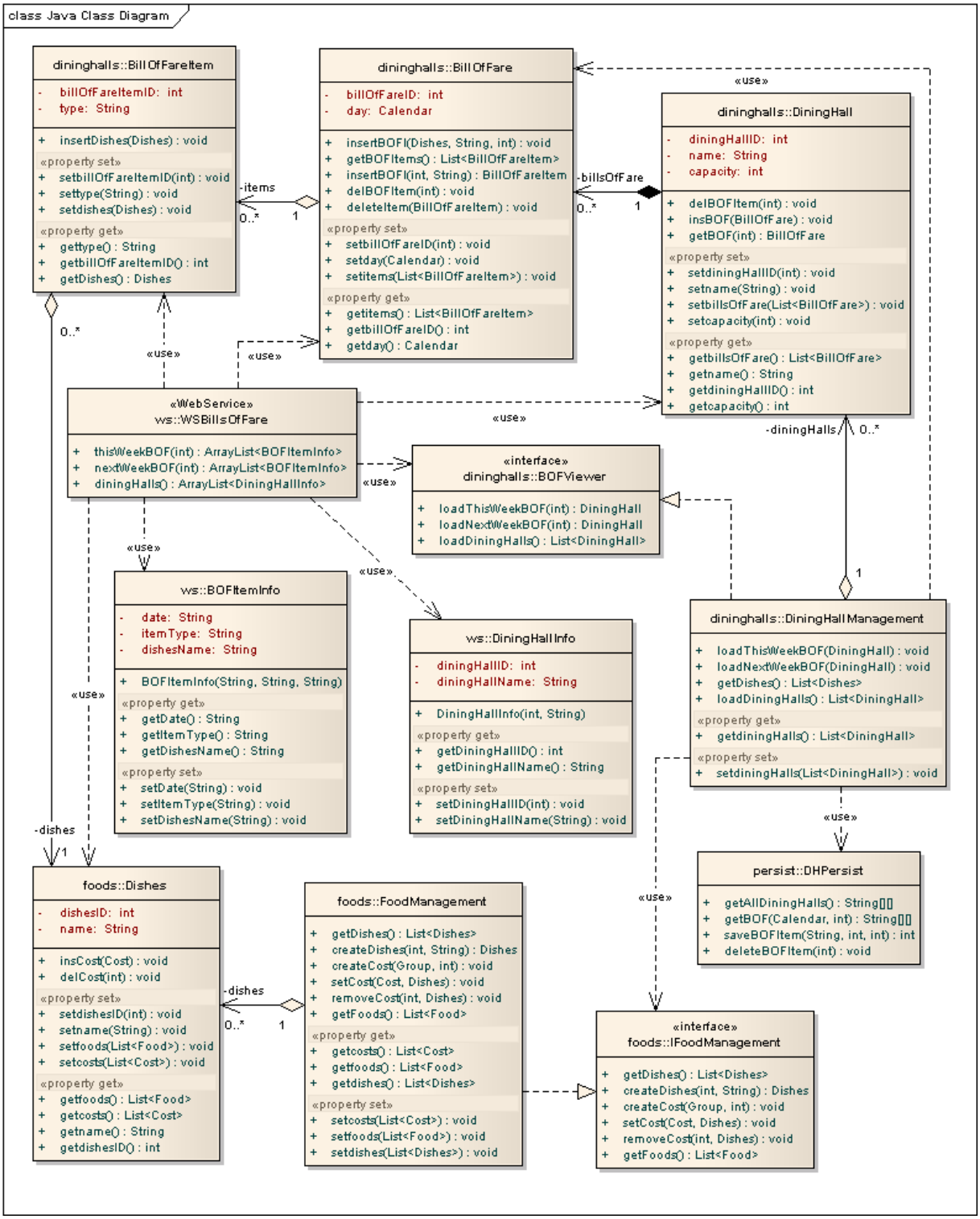
4.4 PSM (JSSI)

Platformově závislý model (Platform Specific Model) informačního systému školních jídelen obsahuje model systému specifický pro jazyk Java, diagram nasazení (Deployment Diagram), model testovacích tříd (JUnit Model), databázový model (DDL model) a WSDL model.

4.4.1 Diagram Java tříd (JSSI)

Platformově závislý model tříd znázorňuje Java třídy, které jsou použity při realizace případu užití čtení jídelníčků externím systémem pomocí webové služby. Diagram je uveden na obrázku 4.12. Model vznikl transformací z platformově nezávislého modelu tříd. Při transformaci byly generovány metody getter a setter. Transformační skript také generuje metody getter a setter pro kontejnery, které však ještě nejsou vedeny coby atributy třídy. Jsou zatím modelovány jen pomocí asociací. Jako atributy budou vytvořeny až při generování zdrojového kódu. Do modelu byly doplněny další dvě třídy *BOFItemInfo* a *DiningHallInfo*. Jde o tzv. POJO objekty (Plain Old Java Object) - třídy, kde ke každému atributu se přistupuje

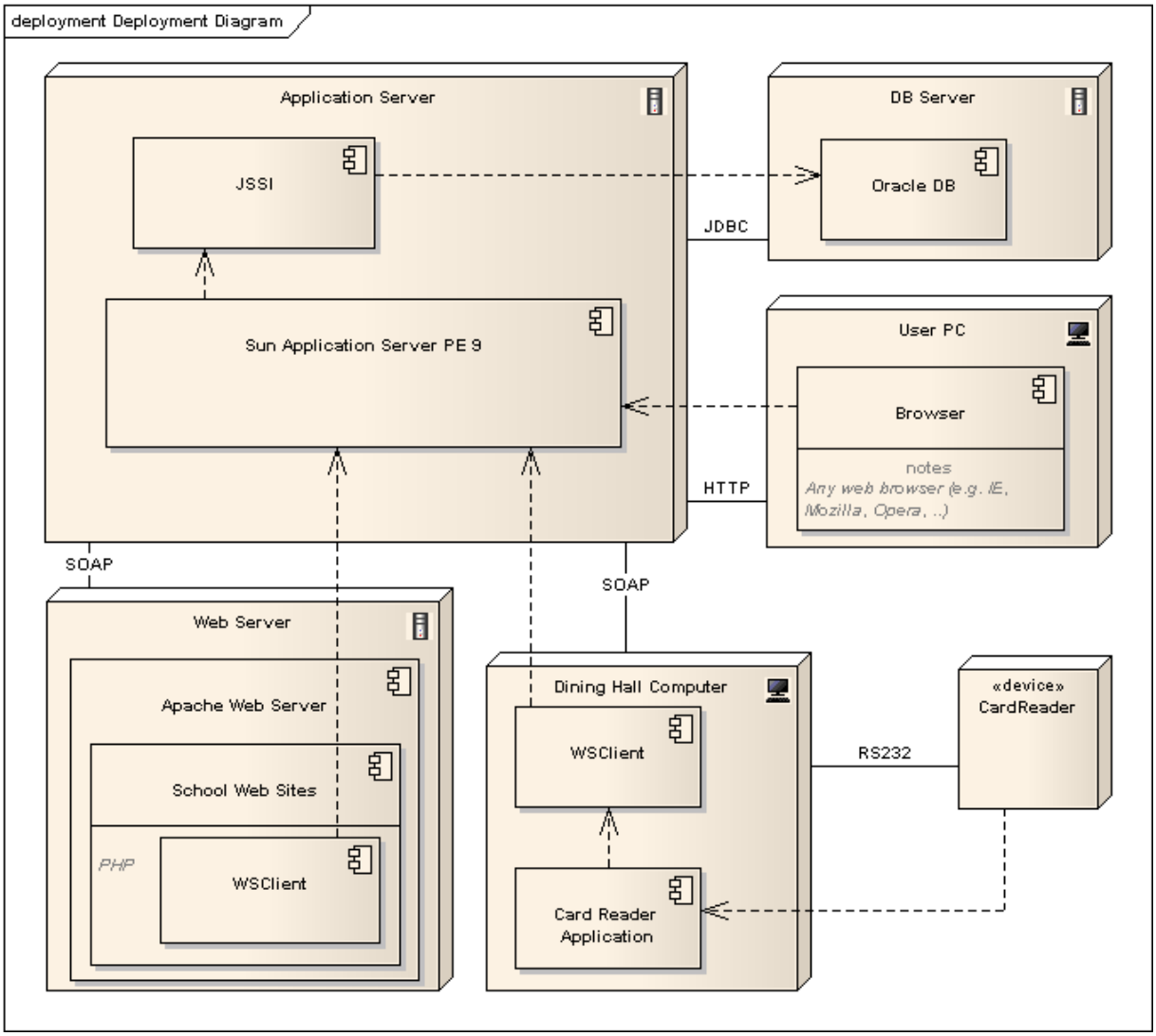
pomocí metod s prefixem get, set, případně is. Další metody třídy neobsahují. Tyto třídy slouží k vracení informací o jídelnách a jídelníčcích webovou službou.



Obr. 4.12: Diagram Java tříd (PSM)

4.4.2 Diagram nasazení (JSSI)

Diagram nasazení (Deployment Diagram) zobrazuje rozmístění softwarových komponent na hardwarových zařízeních a jejich propojení. Diagram je zobrazen na obrázku 4.13.



Obr. 4.13: Deployment Diagram (JSSI)

Hlavní hardwarovou komponentou systému jídelen je aplikační server, na kterém je nasazen Sun Application Server SE 9. Tento server zajišťuje chod samotného informačního systému JSSI. Systém dále využívá databázi Oracle, která je nasazena na databázovém serveru. Školní webové stránky, které běží na dedikovaném webovém serveru, přejímají od informačního systému jídelníčky pomocí klienta webové služby. Další webovou službu informačního systému využívají čtečky elektronických karet, které jsou připojeny k počítačům v jídelnách (*Dining Hall Computer*). Uživatelské počítače (zákazníků, vedoucího jídelny,

vedení školy, kuchaře, ...) pomocí webového prohlížeče přistupují k informačnímu systému (přes webový server na Sun Application Serveru).

4.4.3 Další modely

V balíku PSM je dále model tříd pro jednotkové testy frameworku JUnit, databázový model (DDL) a model WSDL. Model testovacích tříd JUnit vznikl transformací z platformově závislého modelu tříd. V testovacích třídách byly při transformaci automaticky vytvořeny metody *main(String[])*, *setUp(), tearDown()* a dále testovací metody *testJmenoMetody()*. Databázový model DDL vznikl transformací z tříd platformově nezávislého modelu. Díky omezení v upraveném transformačním skriptu byly při transformaci zpracovány jen třídy se stereotypem <<*entity*>>. Transformace automaticky generovala primární a cizí klíče. Vygenerovány byly také tabulky rozbíjející vazbu M:N. Název těchto tabulek je tvořen podle následujícího pravidla: *Join*`Tabulka1`*To*`Tabulka2`. Datové typy byly transformovány dle zvoleného typu cílové databáze. Kvůli ukázkové implementaci byla pro tento model vybrána databáze MySQL (tedy odlišná od databáze plánované v modelu nasazení). To nám také na praktickém příkladu ukazuje výhodu MDA. Z platformově nezávislého modelu lze snadno vygenerovat DDL model pro libovolnou databázi. Diagramy JUnit a DDL jsou uvedeny v příloze E. WSDL model je popsán v kapitole 5.2.1, která se zabývá webovou službou informačního systému jídelen.

5 Webová služba jídelny

Informační systém jídelen zveřejňuje jídelníčky také pomocí webové služby. Primárně je tato služba systému určena pro poskytování jídelníčků webovému portálu školy. Z podstaty webových služeb je ale možné poskytnuté jídelníčky přijímat i na dalších systémech a zařízeních založených na různých platformách.

5.1 Web Services

Pod pojmem webové služby (Web Services) rozumíme systém navržený pro podporu přenosu informací mezi počítači různých platforem. Jde o obdobu technologií určených pro vzdálené volání funkcí v distribuovaných systémech jako RPC, CORBA či RMI. Oproti těmto technologiím jsou však webové služby zcela nezávislé na platformě (operačním systému, programovacím jazyku, architektuře počítače, ...). Při komunikaci pomocí webových služeb hraje jeden z počítačů roli poskytovatele webové služby a druhý počítač je klientem webové služby. Poskytovatel služby specifikovaným způsobem dává data k dispozici na síti. Klient si zjistí adresu a popis služby, aby ji následně byl schopen využívat. Data se zpravidla přenášejí v XML formátu a díky tomu jsou snadno rozšiřitelná. Název „webové služby“ je daný historicky, ale dnes je vlastně nesprávný, protože nejde pouze o služby poskytované přes web. Původní záměr přenášet SOAP zprávy přes protokol HTTP (HyperText Transfer Protocol) byl později rozšířen o možnost přenosu zprávy dalšími přenosovými protokoly, jako například SMTP (Simple Mail Transport Protocol). [15]

Existuje několik modelů webových služeb. Tyto modely spadají do dvou základních kategorií: *SOAP-based Web Services* a *REST-based (RESTful) Web Services*.

5.1.1 SOAP-based Web Services

Webové služby založené na protokolu SOAP přestavují tradiční skupinu webových služeb. Technologie těchto služeb je tvořena třemi částmi:

- protokolem pro vzdálené volání procedur
- popisem poskytovaných služeb
- mechanizmy pro nalezení služeb

Pro vzdálené volání procedur se využívá protokol SOAP. Přenášené SOAP zprávy mají formát XML. Formát XML má výhodu v tom, že se předávaná data nemusí omezovat na text, je možné předávat si složité objekty nebo kolekce objektů. Formát XML umožňuje také

popsat strukturu a typy předávaných dat a pomocí XML Namespaces lze zamezit kolizím stejných jmen pro různé věci. Předávané parametry také nesou informaci o tom, kterou funkci je třeba zavolat. Zpráva v SOAPu má kořenový element *Envelope* (obálka). V této obálce jsou pak dva elementy *Header* (hlavička) a *Body* (tělo). Hlavička je nepovinná a využívá se pro přenos pomocných informací.

Abychom mohli webovou službu používat, musíme znát, jaké funkce nám webová služba poskytuje, jaké mají tyto funkce parametry a jaké hodnoty vracejí. K tomuto účelu slouží WSDL (Web Service Definition Language) dokument, který definuje služby jako kolekci koncových bodů zpracovávajících zprávy. Abstraktní definice koncového bodu a zpráv je oddělena od jejich konkrétní podoby a datových vazeb, což umožňuje opětovné použití abstraktních popisů. Web Services Description Language je jazyk založený na formátu XML. Tento jazyk vznikl jako společná iniciativa firem Microsoft a IBM, které si uvědomovaly potřebu sjednocení jazyka pro popis rozhraní webových služeb. WSDL dokument používá pro popis následujících elementů:

- **Types** - Definice datových struktur používaných ve zprávách. Lze použít libovolný typový systém, ale nejčastěji se používá XML Schema. O mapování datových typů XML schémat na nativní datové typy použitého jazyka se starají nástroje pro webové služby.
- **Message** - Abstraktní typová definice předávané zprávy. Definuje formát zpráv pomocí dříve definovaných datových typů. Každá zpráva se může skládat z několika logických částí s vlastním datovým typem.
- **Operation** - Abstraktní popis operací podporovaných službou. Každá operace definuje obvykle dvě zprávy. Jedna zpráva je vstupní a druhá výstupní. Zpráv lze ale definovat i méně. Každá zpráva obsahuje žádnou, jednu nebo více částí. Tyto části odpovídají parametrům a návratovým hodnotám. Protože zprávy mohou obsahovat více částí, mohou tak i funkce vracet více návratových hodnot než jen jednu.
- **Port Type** - Sdružuje dohromady několik operací podporovaných jedním nebo více uzly.
- **Binding** - Navázání určitého typu portu na konkrétní protokol a formát přenosu zpráv.
- **Port** - Každá brána (port) má vazbu (binding), což je způsob, jak se daná brána volá. Jde o kombinaci vazby a síťové adresy.
- **Service** - Kolekce několika koncových bran (portů) sdružených do jedné služby. Jedna služba může mít jednu nebo více bran.

WSDL soubor webovou službu plně popisuje, takže lze za pomoci automatizovaných nástrojů z WSDL popisu vygenerovat kód pro volání služby v nějakém konkrétním programovacím jazyce. Tento vygenerovaný kód pak lze volat stejným způsobem, jako by se jednalo o lokálně implementovanou funkci. [15]

WSDL dokumenty není nutné psát ručně, existují nástroje pro jejich generování přímo z kódu programovacího jazyka (např. Java2WSDL, NetBeans, …). Vývojové prostředí Enterprise Architect by mělo umět třídu navrženou v platformově nezávislém modelu transformovat do WSDL modelu a z něj následně generovat WSDL dokument. Této problematice se věnuje kapitola 3.2.4.3.

Abychom mohli webové služby používat, dozvědět se vůbec o jejich existenci, potřebujeme mechanismus pro jejich nalezení. Jedním z těchto mechanismů je UDDI (Universal Description, Discovery and Integration). UDDI slouží pro registrování, kategorizování a vyhledávání webových služeb a funguje jako velký adresář, který obsahuje informace o subjektech a službách, které poskytují. Tento registr sám rovněž pracuje jako webová služba. Registr obsahuje čtyři druhy entit:

- **Business entity –** Pro každou firmu z registru je zaznamenán její název, stručný popis a kontaktní údaje. Business entitě mohou být přiřazeny klasifikační identifikátory určující oblast podnikání a geografickou polohu.
- **Business service -** Seznamy služeb, které firma poskytuje. Každá služba obsahuje seznam šablon vazeb (binding templates).
- **Binding template -** Šablony vazeb popisují, jak je možné se službou komunikovat. Tato informace bývá popsána odkazem na WSDL soubor. Každá šablona také odkazuje na typ služby, který implementuje.
- **Service typ -** Typ služby definuje abstraktní službu. Více firem může nabízet stejný druh služby se stejným rozhraním a tedy i typem služby.

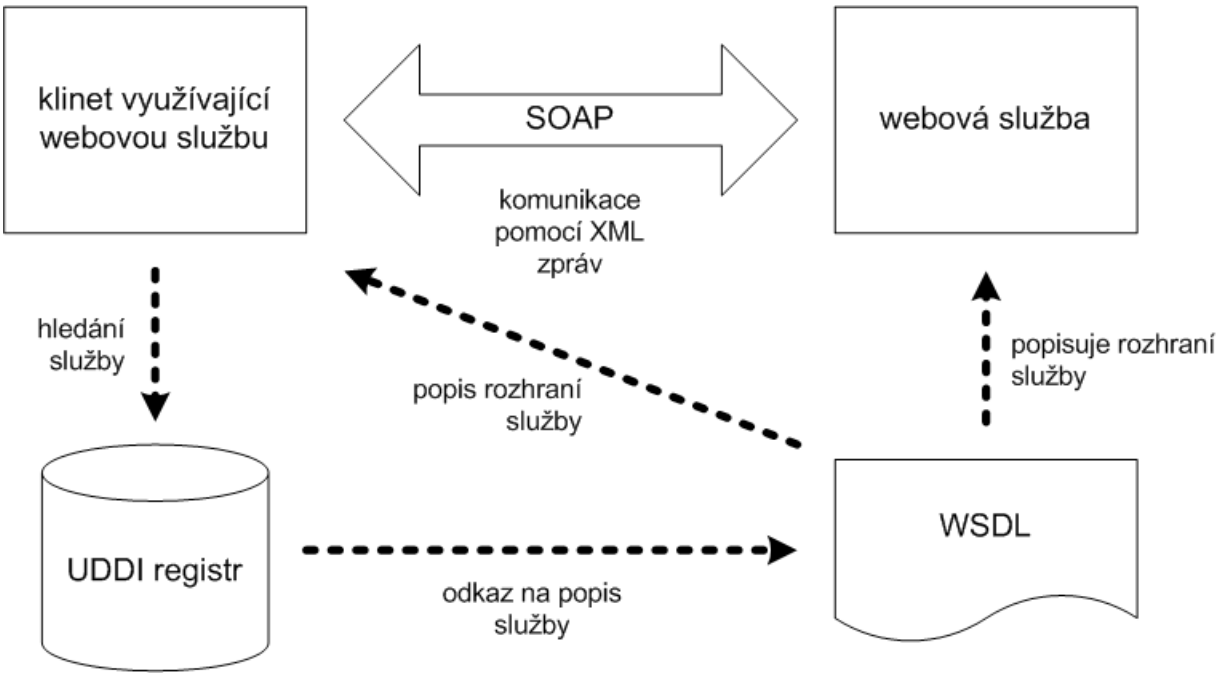
Pří hledání webové služby si vývojář projde registr a najde si službu, kterou potřebuje. Získá pro ni popis WSDL a může ji začít používat. [13]

Existují dvě centrální databáze UDDI, které spravují firmy IBM a Microsoft. Plné dvě třetiny záznamů v těchto databázích jsou však neplatné. Databáze také nijak nezaručují důvěryhodnost poskytovatelů služeb. Existuje však také konkurenční řešení WSIL (Web Services Inspection Language), kde při hledání služby nejprve najdeme důvěryhodného poskytovatele, kterého požádáme o popis rozhraní poskytovaných služeb. WSIL popisuje

služby pomocí souboru jménem `inspection.wsil`, který je umístěný vždy v hlavním adresáři webového serveru poskytovatele, tedy na všeobecně známém místě. [15]

Jako transportní protokol je zpravidla používán protokol HTTP, ale lze použít například i protokoly SMTP nebo FTP (File Transfer Protocol). Protokol HTTP je prakticky základem dnešní internetové infrastruktury a není blokován firewally, což je jedna z hlavních výhod oproti jiným protokolům (např. DCOM).

Vztah mezi technologiemi SOAP, WSDL a UDDI je znázorněn na obrázku 5.1.



Obr. 5.1: Webová služba [13]

5.1.2 REST-based Web Services

Webové služby založené na technologii REST (REpresentational State Transfer) jsou kolekcemi webových zdrojů identifikovaných svojí URI adresou. Každý dokument a proces je modelován jako webový zdroj s unikátní URI. S těmito zdroji se pracuje pomocí akcí specifikovaných v HTTP hlavičce. Pro použití těchto webových služeb není zapotřebí SOAP ani WSDL standardů. Výměna zpráv mezi komunikujícími stranami může proběhnout v libovolném formátu (XML, JSON, HTML, apod.). REST-based webové služby používají metody GET, PUT, POST a DELETE protokolu HTTP. V mnoha případech může jako klient pro REST-based webové služby sloužit webový prohlížeč. Snadno lze záložkovat požadavky a odpovědi ukládat do webové cache paměti. Také administrátor sítě může snadno prohlídkou

HTTP hlaviček kontrolovat, co se na webové službě děje. REST se jeví jako vhodná technologie pro webové služby, u kterých nevyžadujeme větší zabezpečení, než jaké je dostupné v HTTP infrastruktuře. REST-based webové služby poskytuje například Amazon a Google Maps. [19]

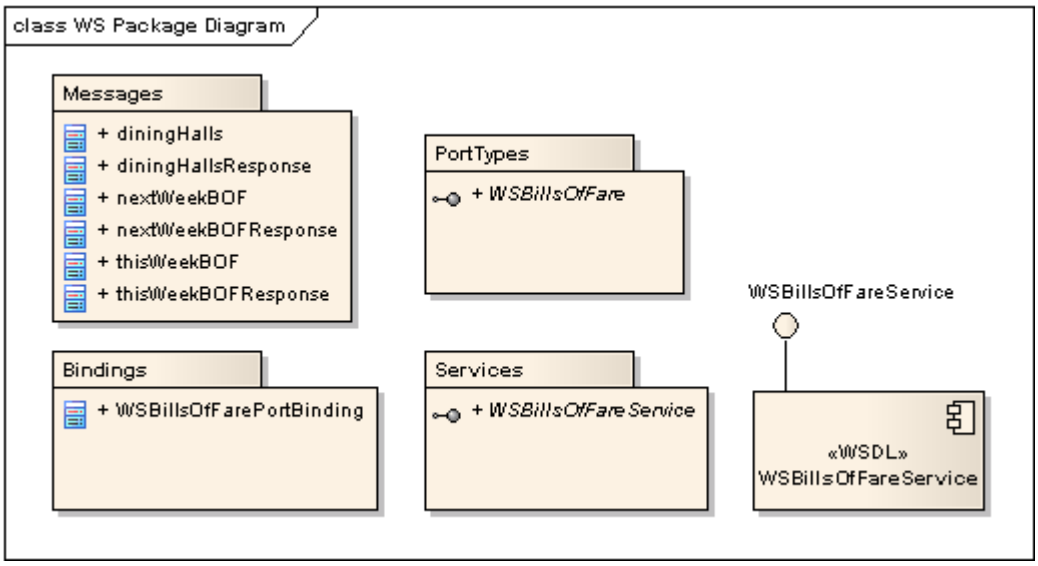
5.2 Realizace webové služby

Enterprise Architect umožňuje namodelovat WSDL model a z něj generovat WSDL soubor. WSDL model je v Enterprise Architectu také možné automaticky generovat z Java rozhraní (*interface*). Enterprise Architect ale nemá podporu pro automatické vytváření serverové nebo klientské části webové služby z WSDL popisu, jak je tomu u některých jiných nástrojů (Axis WSDL2Java, NetBeans,..). Je tedy možné napsat tyto části ručně nebo využít jiný nástroj. K realizaci serverové části webové služby jsem tedy použil vývojové prostředí NetBeans.

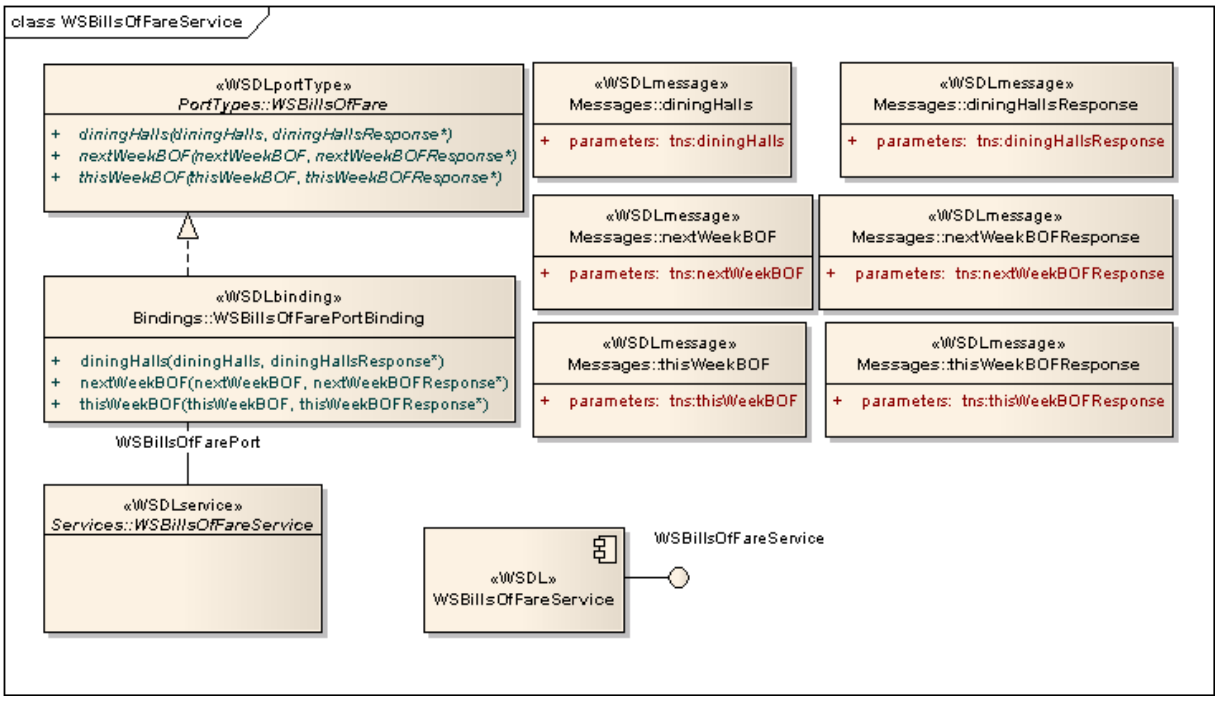
5.2.1 WSDL model

Diagramy WSDL modelu graficky znázorňují WSDL soubor. WSDL model patří do balíku PSM. Jazyk WSDL je sice velice obecný, aby zachoval platformovou nezávislost [15], přesto právě výběr konkrétní technologie (WSDL/SOAP, REST, ..) pro realizaci webové služby řadí tento model do platformově závislého balíku PSM. V platformově nezávislém balíku PIM je rozhraní webové služby reprezentováno jako platformově nezávislá třída se stereotypem *<<boundery>>*.

WSDL model pro webovou službu *WSBillsOfFare* z obrázku 5.2 obsahuje následující balíky: *Messages, Bindings, PortTypes* a *Services*. V balíku *Messages* jsou obsaženy WSDL zprávy, které Enterprise Architect reprezentuje pomocí UML třídy se stereotypem *<<WSDLmessage>>*. Webová služba *WSBillsOfFare* používá zprávy *diningHalls, thisWeekBOF* a *nextWeekBOF*. Zpráva výstupní obsahuje ve svém názvu sufix *Response*, zpráva vstupní je bez sufixu. Balík *Binding* obsahuje WSDL vazby reprezentované UML třídami se stereotypem *<<WSDLbinding>>*. Třída reprezentující WDSL vazbu implementuje operace specifikované v rozhraní *WSDLportType*. Proto je třeba před vytvářením WSDL vazeb nejprve definovat tato *WSDLportType* rozhraní. Rozhraní jsou reprezentována UML třídami se stereotypem *<<WSDLportType>>* a jsou obsažena v balíku *PortTypes*. Balík *Services* obsahuje WSDL službu reprezentovanou UML třídou se stereotypem *<<WSDLservice>>*. Služba *WSBillsOfFareService* je asociována s WSDL vazbou *WSBillsOfFarePortBinding*. Popsané třídy jsou znázorněny na obrázku 5.3.



Obr. 5.2: Diagram balíků z WSDL modelu

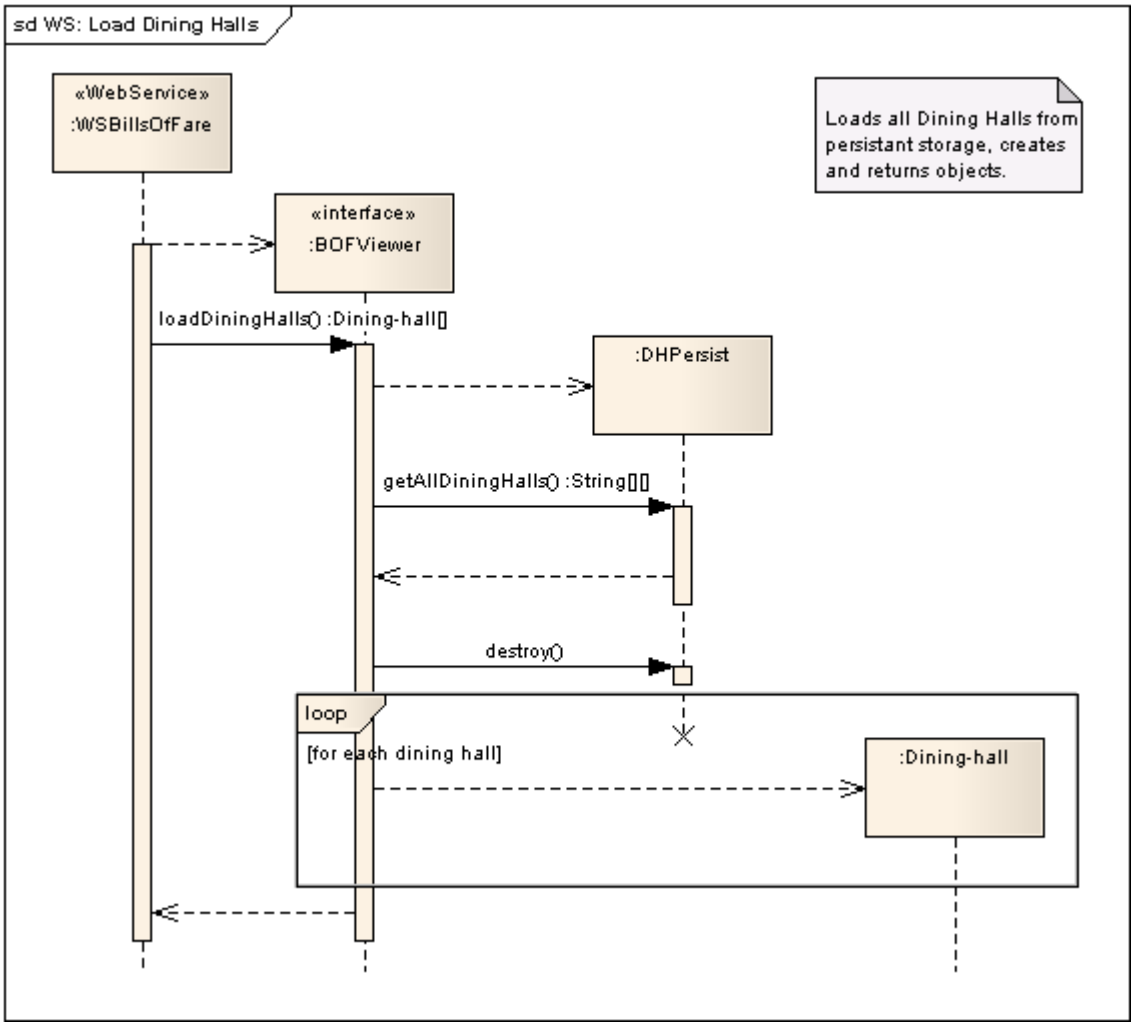


Obr. 5.3: WSDL model

5.2.2 Aplikační logika

Serverová část webové služby byla implementována pomocí nástroje NetBeans, který umožňuje klientskou i serverovou část služby vytvořit z WSDL popisu a nebo pomocí

grafického WSDL editoru. Vytvořené části obsahují kompletní infrastrukturu potřebnou k provozu služby. Zbývalo tedy jen implementovat aplikační logiku poskytovaných metod. Aplikační logika obsažená v těchto metodách je znázorněna na analytických sekvenčních diagramech WS *Load Dining Halls* (obr. 5.4) a *WS Load This Week Bill Of Fare* (příloha E).



Obr. 5.4: Aplikační logika webové služby

Webová služba zpřístupňuje celkem tři metody: *diningHalls()* (pro získání seznamu jídelen), *thisWeekBOF(int)* (pro získání jídelníčků v aktuálním týdnu v jídelně dané parametrem) a *nextWeekBOF(int)* (pro získání jídelníčků v následujícím týdnu v jídelně dané parametrem).Vnitřní implementace posledních dvou metod je podobná, takže stejný kód těchto metod jsem pomocí refaktorování („Extract Method“) vložil do nové pomocné metody.

Metoda *diningHalls()* nejprve vytvoří instanci třídy *BOFViewer*, na které pak volá metodu *loadDiningHalls()*. Instance třídy *BOFViewer* vytvoří instanci třídy *DHPersist*, která nahraje informace o jídelnách z perzistentního úložiště (tím je v tomto ukázkovém příkladě databáze

Derby). Pro databázové dotazy je použito předpřipravených SQL dotazů („PreparedStatement“). Samotné spojení s databází zajišťuje třída *DerbyConnection,* která obsahuje identifikaci databáze a přístupové údaje. Pro komunikaci s databází používá JDBC (Java Database Connectivity). Z načtených dat jsou vytvořeny objekty reprezentující jídelny (*DiningHall*) a ty jsou v kolekci vráceny instanci třídy *WSBillsOfFare*. Ta je použije k vytvoření objektů třídy *DiningHallInfo*, které vrací klientovi.

Klient získané ID jídelny může použít při volání metod *thisWeekBOF(int)* a *nextWeekBOF(int)*. Instance třídy *BOFViewer* nejprve vytvoří instanci jídelny (třída *DiningHall*), instanci třídy *DHPersist* a instanci rozhraní *IFoodManagement*, které slouží k vytváření menu (*Dishes*). Pomocí metod třídy *java.util.Calendar* (resp. *GregorianCalendar*) je zjištěno datum, které odpovídá pondělku aktuálního (nebo následujícího) týdne. Pro každý den počínaje pondělkem a konče nedělí jsou pak z databáze za pomoci instance třídy *DHPersist* získány informace o jídelníčcích v dané jídelně. Informace následně slouží k vytvoření instancí třídy *BillOfFare* (jídelníček pro daný den a danou jídelnu), k vytvoření instancí třídy *Dishes* (menu) a k vytvoření instancí třídy *BillOfFareItem* (položka jídelníčku), do kterých jsou příslušná menu vložena. Jídelníčky jsou pak vloženy do jídelny, která je vrácena instanci třídy *WSBillsOfFare*. Ta ji použije k vytvoření objektů třídy *BOFItemInfo*, které jsou následně vráceny klientovi.

Webová služba je založena na aplikačním rozhraní JAX-WS (Java API for XML-Based Web Services) a byla nasazena na serveru GlassFish, kde byla také testována pomocí vestavěné testovací stránky. Výstup jednoho z testů je možné najít v příloze F. Ve výstupu jsou nejprve uvedeny hodnoty parametrů, se kterými byla webová služba volána. Následuje seznam vrácených objektů (kolekce objektů *BOFItemInfo*) a *SOAP Request* i *SOAP Response* zprávy. Pro otestování webové služby jsem také vytvořil standardní Java aplikaci *WSBillsOfFareClientApplication*, která použije webovou službu k výpisu jídelníčků všech jídelen v aktuálním týdnu.

6 Závěr

Specifikace MDA je poměrně mladá a svými vizemi sahá za hranice současných nástrojů a modelovacích jazyků. Pokud by se konceptu MDA podařilo nakonec uspět, znamenalo by to stejný milník ve způsobu programování, jako byl přechod od programování v Assembleru k vyšším programovacím jazykům. Textové programování by bylo nahrazeno programováním grafickým. Grafické programování pak umožní povýšit tvorbu aplikací na abstraktnější úroveň. Takový vývoj by byl přístupnější doménovým expertům a business analytikům. Tvorba programů by pak mohla spočívat ve vytvoření doménových a business modelů a jejich následných automatizovaných transformacích vedoucích až vytvoření spustitelné aplikace.

Zůstaneme-li ale u výhod, které nám MDA přináší už nyní a bude přinášet v blízké budoucnosti, je třeba zmínit následující: Vytvořený implementačně nezávislý návrh zvyšuje přenositelnost aplikace, což vede k úspoře nákladů a snížení komplexnosti vývoje při přechodu na jinou platformu. Automatickým generováním je v dnešní době možno vytvořit až 80% zdrojových kódů, což vede k úspoře programátorské práce a ke zvýšení produktivity. Automatické generování kódu také přispívá ke kvalitě, protože eliminuje programátorské chyby dané lidským faktorem. V Enterprise Architectu ale tak vysokého poměru mezi automaticky generovaným a ručně psaným kódem dosaženo nebylo a z velké části musel být kód doplňován ručně.

Stále se vedou polemiky o tom, ve kterých oblastech lze MDA použít. Zda se hodí k vývoji webových služeb, zda se hodí k vývoji webových aplikací, zda je vhodná pro malé projekty, zda je použitelná pro iterativní vývoj, apod. Pro nalezení odpovědí je nutné testovat MDA v praxi, provádět případové studie a jejich analýzy. Stewen Witkop ze společnosti EDS uvádí několik příkladů, které nám mohou pomoci v rozhodování, zda k vývoji MDA použít či nikoliv: Přistup MDA funguje nejlépe na rozsáhlých projektech a ve velkých společnostech. Společnosti, které MDA používají, měli už dříve zkušenosti s modelováním pomocí UML. Pro MDA je potřeba se rozhodnout na začátku projektu, rozhodně ne v průběhu projektu nebo před jeho dokončením. [27]

Důležité pro úspěch architektury MDA je také její přijetí mezi významnými výrobci IDE. Podporu pro MDA do svých nástrojů začlenilo i IBM (Rational) a Borland (Together). Já jsem se ve své práci zabýval nástrojem Enterprise Architect firmy Sparx Systems.

Enterprise Architect se ukázal jako plnohodnotný vývojářský nástroj. Pokrývá celý vývoj softwaru od sběru požadavků, přes analýzu, návrh, testování až po údržbu. V úrovni podpory architektury MDA tento nástroj nijak významně nezaostává za ostatními leadery na trhu.

V praktických příkladech jsem ověřoval proklamované vlastnosti nástroje a až na několik výjimek nástroj uvedené vlastnosti splňoval. Jestliže většinu funkcí a možností nástroje lze nalézt v manuálech a tutoriálech, o nedostatcích se zde nepíše. Proto jsem větší část popisu nástroje věnoval právě jeho nedostatkům, zvláštním vlastnostem a ukázkovým příkladům.

Přestože fórum komunity kolem nástroje Enterprise Architect v diskuzi „Bugs and Issues“ obsahuje velké množství aktuálních příspěvků, chyby objevené v nástroji jsem zde většinou nenašel.Výrobce vývojového prostředí, firmu Sparx Systems Pty Ltd., jsem tedy upozornil na několik chyb nalezených v tomto nástroji. Celkem jsem ohlásil šest chyb. První chyba (špatná kardinalita) byla uznána a opravení bylo slíbeno v dalším vydání nástroje. Odpovědí na druhou ohlášenou chybu (špatné generování WSDL modelu) byl návod, jak postupovat, aby k chybě nedošlo. Ve své práci ale uvádím důvody, proč i nadále danou vlastnost programu shledávám jako chybnou. Třetí ohlášení se spíše týkalo špatného výchozího nastavení nástroje, které způsobovalo generování metod s názvy v rozporu s konvencemi jazyku Java. Tato chyba nejspíše nebude opravena. Čtvrtá chyba se opět týká generování špatných názvů, v tomto případě šlo o jména balíků. Zde nebylo možné změnit výchozí nastavení, takže jsem společně s vývojáři Enterprise Architectu napsal nový transformační skript, který tuto chybu řeší. Pátá chyba (špatné názvy atributů v těle metod getter a setter v generovaných zdrojových kódech) byla vývojářům už známa. Šestá chyba (špatná manipulace se zakončením „life line“ v sekvenčním diagramu) byla předána odpovědným vývojářům a uvedení oprav bylo opět slíbeno v některém z nových vydání nástroje. Odpovědní lidé ze společnosti Sparx Systems reagovali na má hlášení rychle a ochotně. První a druhý „bug report“ jsem ve zkrácené verzi vložil k nahlédnutí do přílohy.

Enterprise Architect jsem použil pro návrh informačního systému školních jídelen. Jde o typové řešení vývoje malého informačního systému podle architektury MDA v nástroji Enterprise Architect. Příklad doplňuje teoretické pasáže o praktickou složku a může sloužit jako případová studie.

Při výběru části systému, kterou jsem měl implementovat, jsem volil mezi vytvořením webové aplikace pro editaci jídelníčků a vytvořením webové služby pro poskytování jídelníčků externím systémům. Zvolil jsem webovou službu. Jedním z důvodů byla podpora Enterprise Architectu pro modelování a generování WSDL modelů a druhým důvodem byl můj osobní zájem prohloubit své znalosti v této oblasti.

Práce obsahuje také množství grafických příloh. Jestliže se v práci pojednává o modelech, které mimo jiné přinášejí výhody vizualizace informací, pak by ani práce samotná neměla být bez obrázků. Grafické schéma má mnohdy větší vypovídací schopnost než rozsáhlý textový

popis. Příkladem může být obrázek 2.1 , který zachycuje vztahy mezi modely, metamodely, jazyky a platformou. Čtenářům takové grafické přílohy pomohou v rychlém zorientování se v tématu a k snadnějšímu pochopení popisované problematiky.

Další výzkum, který by na práci navazoval, by mohl například porovnávat různá vývojová prostředí s podporou MDA. Pomocí každého z těchto nástrojů by byl vytvořen návrh stejného informačního systému (např. školních jídelen), což by vedlo k zvýraznění hledaných odlišností, které by pak bylo možné snadněji porovnat a zdokumentovat. Podobně by šlo porovnat i jednotlivé verze jednoho vývojového prostředí. Analyzováním výsledků by bylo možné určit, jakým směrem se v nástrojích podpora MDA vyvíjí.

Seznam použité literatury

[1] Borland Software Corporation, Successful Implementation of Model Driven Architecture, 2007. http://www.borland.com/resources/en/pdf/products/together/together-successful-implementation-mda.pdf

[2] Cook, S., Domain-Specific Modeling and Model Driven Architecture, MDA Journal, 2004. http://www.bptrends.com/publicationfiles/01-04%20COL%20Dom%20Spec%20Modeling%20Frankel-Cook.pdf

[3] Česká společnost pro systémovou integraci, Inkrementální vývoj IS, http://www.cssi.cz/all_terminologie.asp?kod=463&strana=9&volba=n

[4] Fowler, M., Model Driven Architecture, 2004. http://martinfowler.com/bliki/ModelDrivenArchitecture.html

[5] Fowler, M., UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley Professional, ISBN: 0321193687, 2003.

[6] Frankel, David., MDA Journal: A Response to Forrester, 3. duben 2006. http://www.bptrends.com/deliver_file.cfm?fileType=publication&fileName=04%2D06%2DCOL%2DMDA%2DResponseToForrester%2DFrankel%2Epdf

[7] Frankel, David S., Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley, 2003.

[8] Frankel, David S., Model-Driven Architecture - Reality and Implementation, IONA Technologies, Inc., s. 31. http://www.omg.org/mda/mda_files/DFrankel_MDA_v01-00_PDF.pdf

[9] Gamma, E., a kol., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, ISBN: 0201633612, 1994.

[10] Kleppe, Anneke G., a kol., MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, ISBN:032119442X, 2003.

[11] Kleppe, Anneke G., Warmer, J., What is the Model Driven Architecture?, 2005. http://www.klasse.nl/mda/mda-introduction.html

[12] Kontio, M., Architectural manifesto: Choosing MDA tools, 2005. http://www.ibm.com/developerworks/webservices/library/wi-arch18.html

[13] Kosek, J. Inteligentní podpora navigace na WWW s využitím XML, Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky, diplomová práce, 2002. http://www.kosek.cz/diplomka/dp.pdf

[14] Kožusznik, J., Dotazování, pohledy a transformace v MDA, Objekty 2003, VŠB-Technická univerzita Ostrava, s. 120 – 128, ISBN 8024802740.

[15] Kuba, M. Web Services. Zpravodaj ÚVT MU. ISSN 1212-0901, 2003, roč. XIII, č. 3, s. 9-14.

[16] LBMS, MDA (Model Driven Architecture), LBMS s.r.o. http://www.lbms.cz/Reseni/Tema/MDA.htm

[17] Maierová, M., Grafické modelování aneb Výhody i meze jazyka UML, Computerworld, IDG Czech, číslo 31, 2004. http://archiv.computerworld.cz/cwarchiv.nsf/clanky/2731C646FCEA4758C1256F5500491182?OpenDocument

[18] Mellor, Stephen. J., a kol., MDA Distilled: Principles of Model-Driven Architecture, Addison-Wesley, 2004, ISBN:0201788918.

[19] NetBeans, Sun Microsystems, Inc. Introduction to Web Services. http://www.netbeans.org/kb/60/websvc/intro-ws.html

[20] Object Management Group, MDA Guide Version 1.0.1, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01

[21] Object Management Group, MDA Specifications, 16.11.2007. http://www.omg.org/mda/specs.htm

[22] Object Management Group, Model Driven Architecture (MDA), 9.7.2001. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01

[23] Ošlejšek, R., Objektové metody návrhu IS, Masarykova univerzita, Fakulta informatiky, skripta, 2007. http://www.fi.muni.cz/~oslejsek/PA103/

[24] Sparx Systems, Enterprise Architect 7.0 User Guide, 2007. http://www.sparxsystems.com.au/bin/EAUserGuide.pdf

[25] Sun Microsystems, Inc., Code Conventions for the Java™ Programming Language, 1999. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[26] Šveřepa, J., MDA, aneb architektura řízená modelem, Softwarové noviny, 10/2004. http://www.lbms.cz/Reseni/_pdf/0410-SWN-JS-MDA-aneb-architektura-rizena-modelem.pdf

[27] Witkop, S., Driving business agility with Model Driven Architecture, 2003. http://www.compuware.com/dl/optimaljeds.pdf

Abecední rejstřík

Příloha A

Bug reports:

Hello Miroslav,
Thank you for your email.We have confirmed that this template is wrong and will fix it for a future build of EA.
Best Regards,
Simon McNeilly
Sparx Systems Pty Ltd
support@sparxsystems.com.au
http://www.sparxsystems.com.au

-----Original Message-----

ENVIRONMENT DETAILS

EA Version: 7.0
EA Add In: No Add-In
Build No: 813
Repository Type: No Repository
Operating System: Windows XP
Service Pack: SP2

ISSUE DETAILS

Subject:

Wrong multiplicity in built-in transformations

Details:

There is wrong multiplicity in built-in transformations, Source should
be "0..1" and Target should be "1"

Wrong built-in template:

%if connectorType == "Generalization"%
... (zkráceno)...
%endTemplate%

Steps to Reproduce:

Source should be "0..1" and Target should be "1"

Hello Miroslav,

Thanks for the note.

Do not create a Class with stereotype "interface". Instead, use an Interface (by dragging and dropping an "Interface" icon from the "Class" tool-box on to the diagram).

Hope this resolves your issue. If you have any further queries, feel free to write-in.

Best Regards,

Vimal Kumar
Sparx Systems Pty Ltd
support@sparxsystems.com.au
http://www.sparxsystems.com.au

-----Original Message-----

ENVIRONMENT DETAILS

EA Version: 7.1.828
EA Edition: Corporate
EA Add In: No Add-In
Repository Type: No Repository
Operating System: Windows XP
Service Pack: SP2

ISSUE DETAILS

Subject:

Bug in WSDL transformation

Details:

I tried a new version of EA 7.1 Build 828, where I made a class with stereotype <<interface>> and one operation as well as your example in EA_7_0_UserGuide.pdf - page 890. Then I applied WSDL transformation, but the result was different from your example. There was only package of <XSDSchema> Types, but none portType, binding, messages etc.

Příloha B

Upravené skripty:

```
Attribute
{
  %TRANSFORM_REFERENCE()%
  %TRANSFORM_CURRENT("scope","type")%
  scope=%qt%%attScope == "Public" ? "Private" : value%%qt%
  type=%qt%%CONVERT_TYPE("C#",attType)%%qt%
}

$type=%CONVERT_TYPE("C#",attType)%
%Property(attName,$type)%
```

Transformace - get, set metody (C#)

```
Attribute
{
  %TRANSFORM_REFERENCE()%
  %TRANSFORM_CURRENT("scope","type","stereotype")%
  scope=%qt%%attScope == "Public" ? "Private" : value%%qt%
  type=%qt%%CONVERT_TYPE("Java",attType)%%qt%
%if classStereotype=="enumeration"%
  stereotype="enum"
%else%
  stereotype=%qt%%attStereotype%%qt%
%endIf%
}

$type=%CONVERT_TYPE("Java",attType)%
%Properties(attName,$type)%
```

Transformace - get, set metody (Java)

```
%if packagePath == ""%
%list="Namespace" @separator="\n"%
%list="Class" @separator="\n"%
%endTemplate%

Package
{
 %TRANSFORM_CURRENT("name")%
 name=%qt%%TO_LOWER(packageName)%%qt%
 %list="Namespace" @separator="\n" @indent="   "%
 %list="Class" @separator="\n" @indent="   "%
}
```

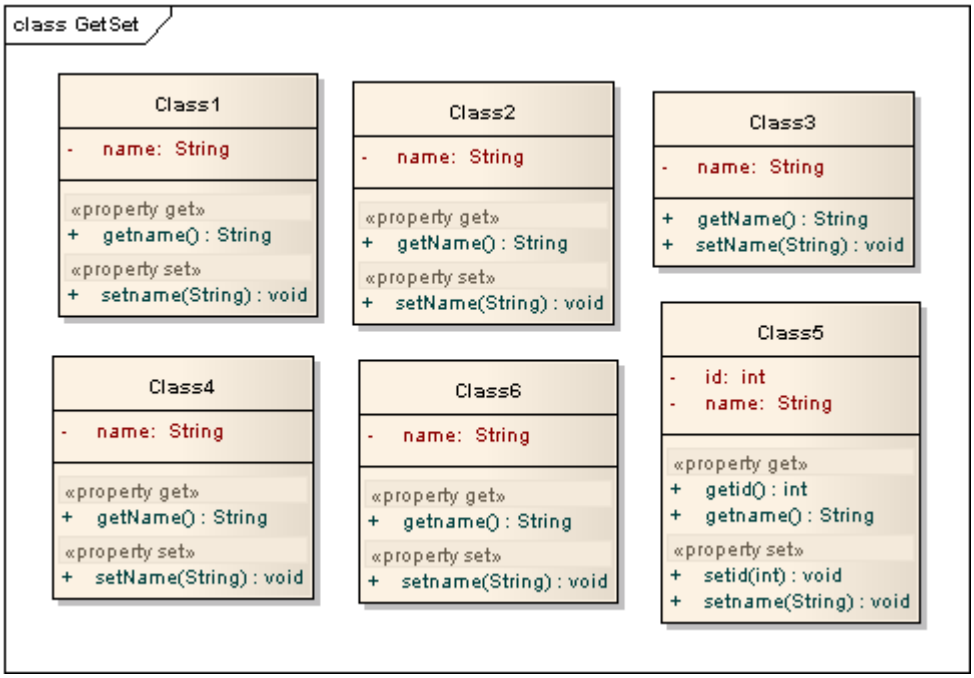
Transformace - úprava názvů balíků

```
Column
{
  %TRANSFORM_CURRENT("type", "stereotype", "collection", "constant", "containment"

  type=%qt%%CONVERT_TYPE(genOptDefaultDatabase,attType)%%qt%

  %if attStereotype=="length200"%
  type=%qt%VARCHAR(200)%qt%
  %endIf%
  %if attStereotype=="length300"%
  type=%qt%TEXT%qt%
  %endIf%
 }
```

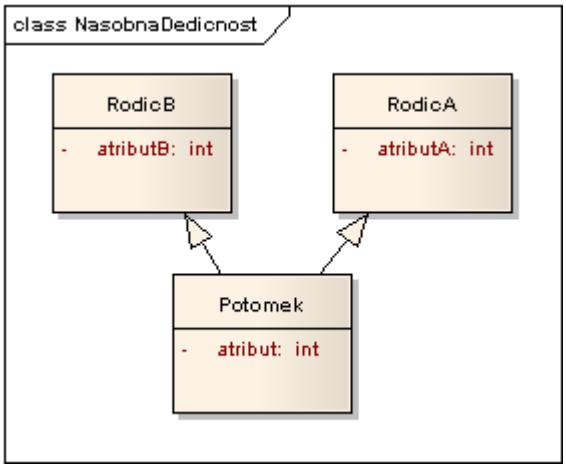
Transformace - DDL, datové typy

Příloha C

Diagramy z příkladů:



Ukázka různých způsobů tvorby get, set metod (PSM)



Transformace násobné dědičnosti (PSM)

Příloha D

Zdrojové kódy příkladů: (formátování kódu kvůli kompaktnosti upraveno)

```
public class Class5 {

        private Integer id;
        private String name;

        public Class5(){}
        public void finalize() throws Throwable {}
        public Integer getid(){return id;}
        /**
         * @param newVal
         */
        public void setid(Integer newVal){id = newVal;}
        public String getname(){return name;}
        /**
         * @param newVal
         */
        public void setname(String newVal){name = newVal;}
}
```

Generování get a set metod (Class5.java)

```
public class Class5 {

        private int id;
        private String name;

        public Class5(){}
        public void finalize() throws Throwable {}
        public int getid(){return <unknown>;}
        /**
         * @param newVal
         */
        public void setid(int newVal){<unknown> = newVal;}
        public String getname(){return <unknown>;}
        /**
         * @param newVal
         */
        public void setname(String newVal){<unknown> = newVal;}
}
```

Generování get a set metod (Class5a.java)

```
using NasobnaDedicnost;
namespace NasobnaDedicnost {
        public class Potomek : RodicA, RodicB {

                private int atribut;
                public Potomek(){}
                ~Potomek(){}
                public override void Dispose(){}
        }//end Potomek
}//end namespace NasobnaDedicnost
```

Násobná dědičnost (C#)

```
public class Potomek extends RodicA RodicB {

        private int atribut;
        public Potomek(){}
        public void finalize() throws Throwable {
                super.finalize();
        }
}
```

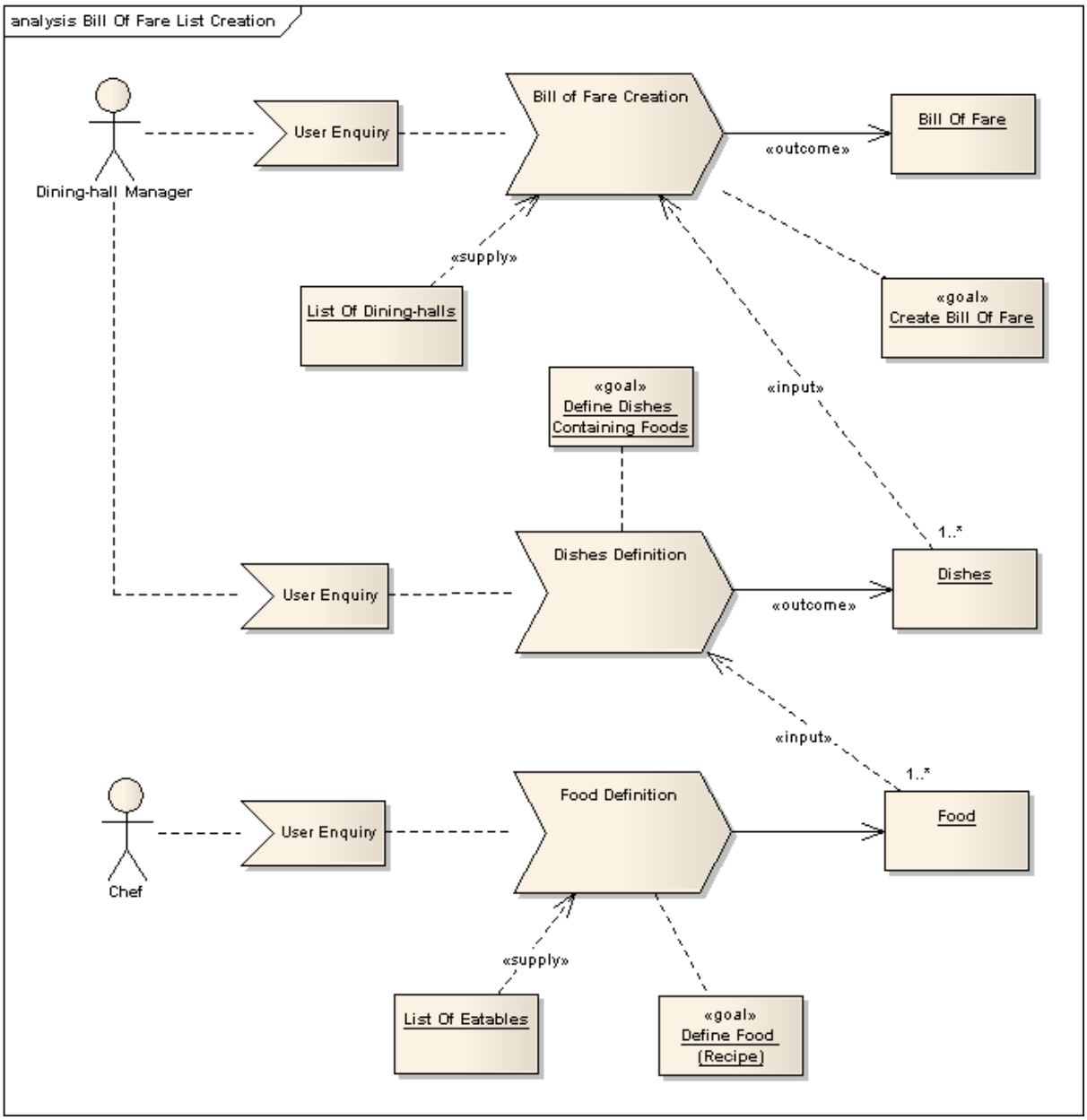
Násobná dědičnost (Java)

```
public class Class1 {

        private String name;
        private String surname;

        public Class1(){}

        public void finalize() throws Throwable {}
        public void showName(){}
        public void showSurname(){}
}
```

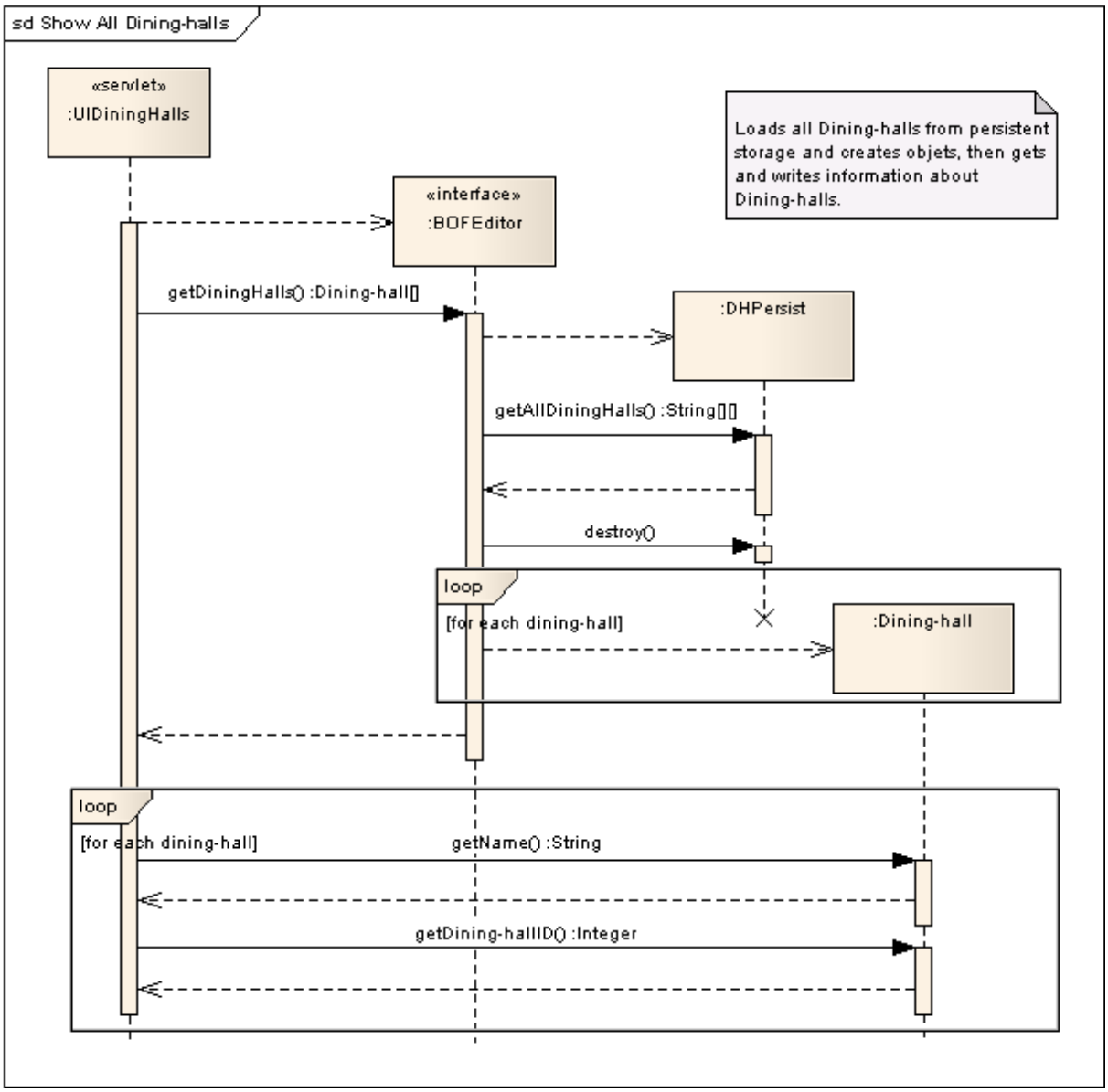
Synchronizace modelu a kódu

Příloha E

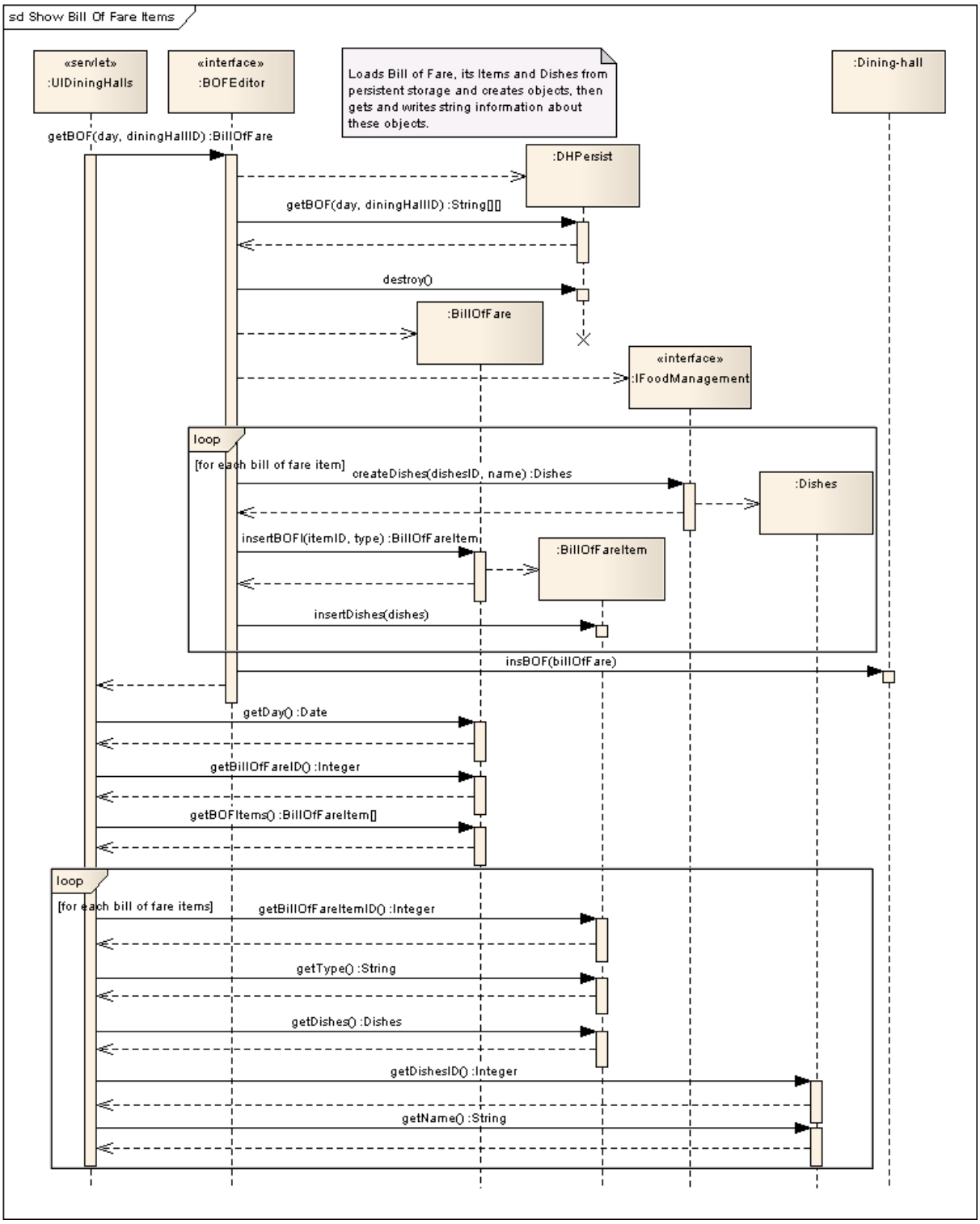
Diagramy JSSI:



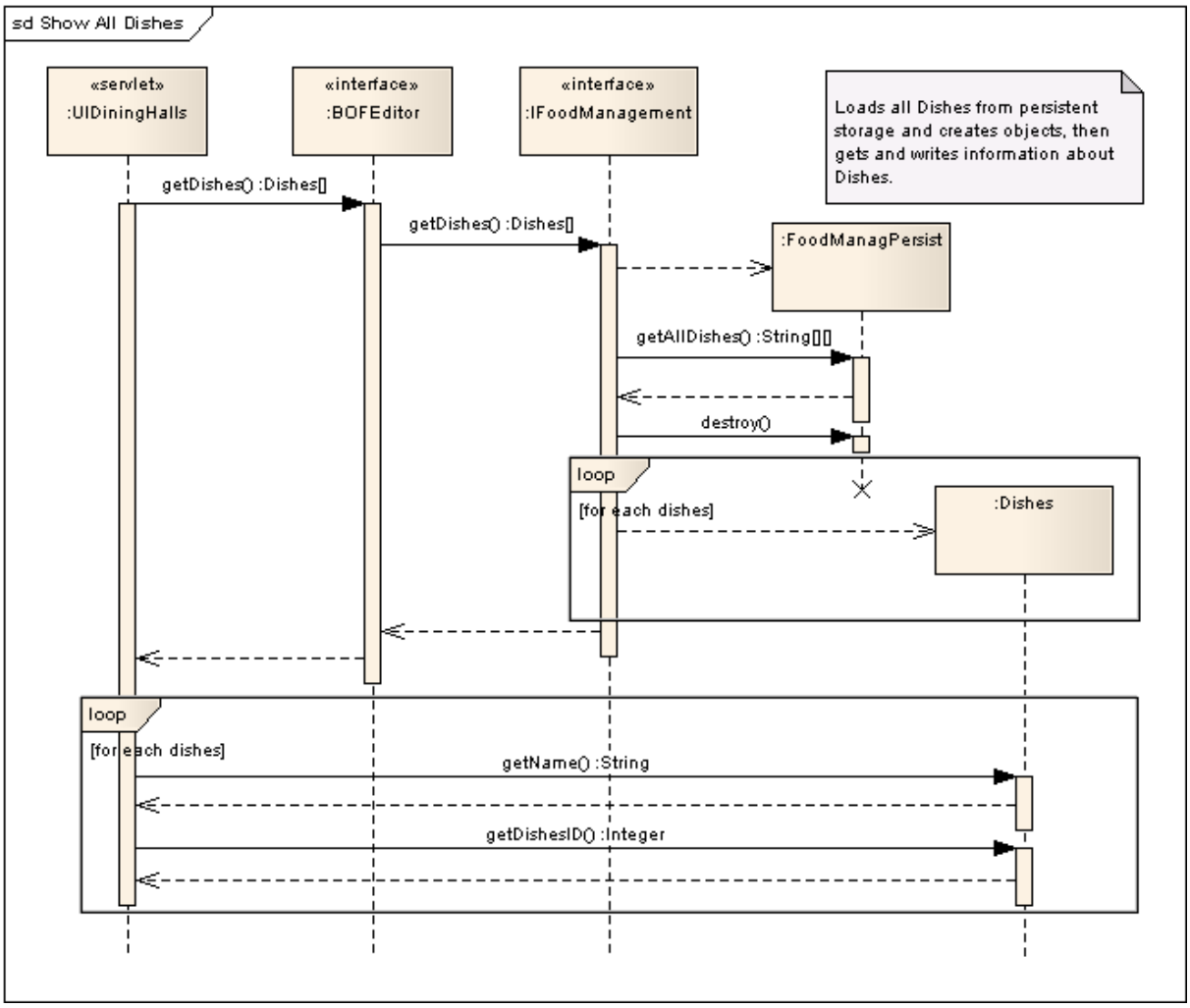
Proces Bill Of Fare List Creation (BPM)



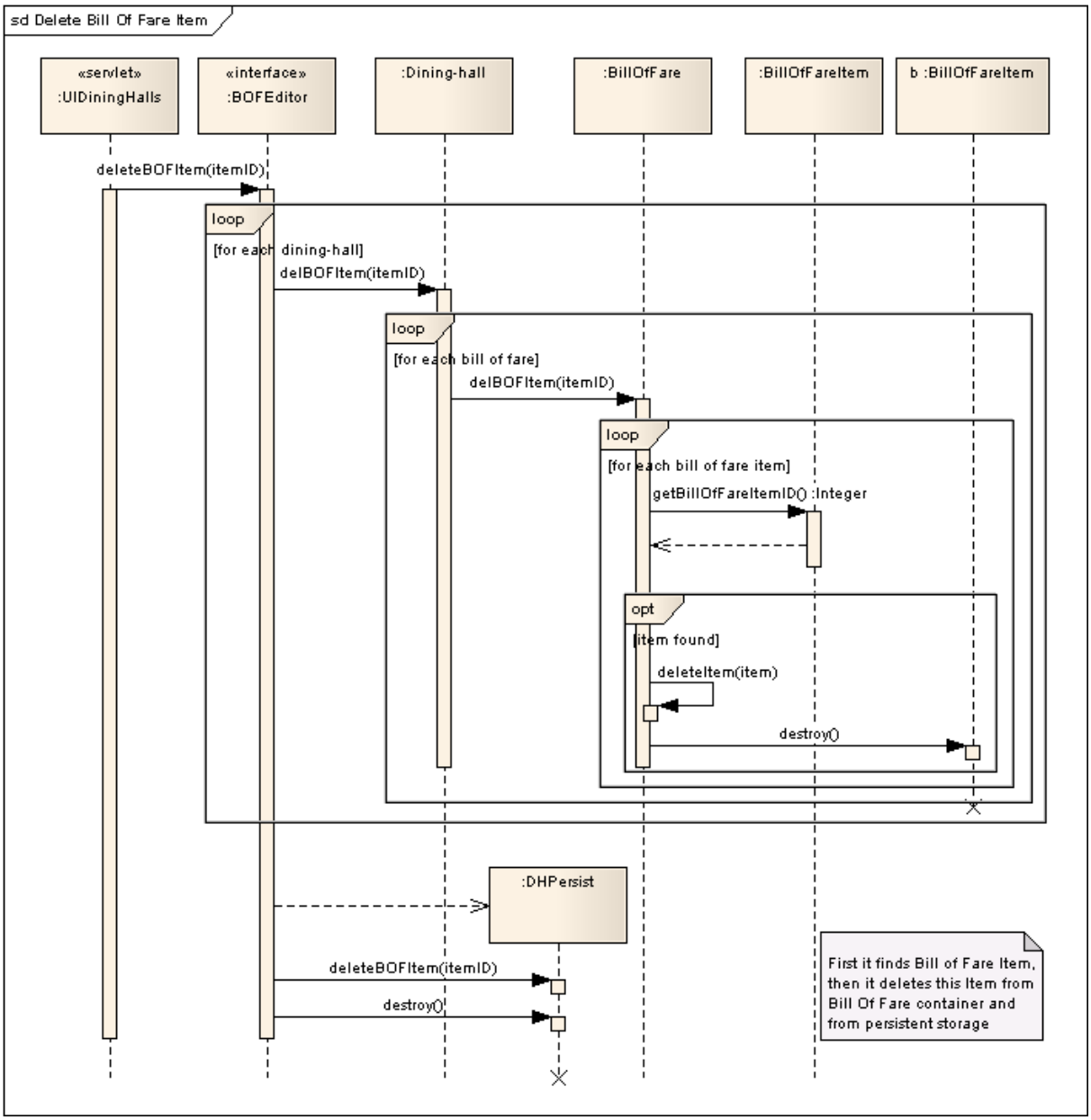
Sekvenční diagram Show All Dining-halls (JSSI)



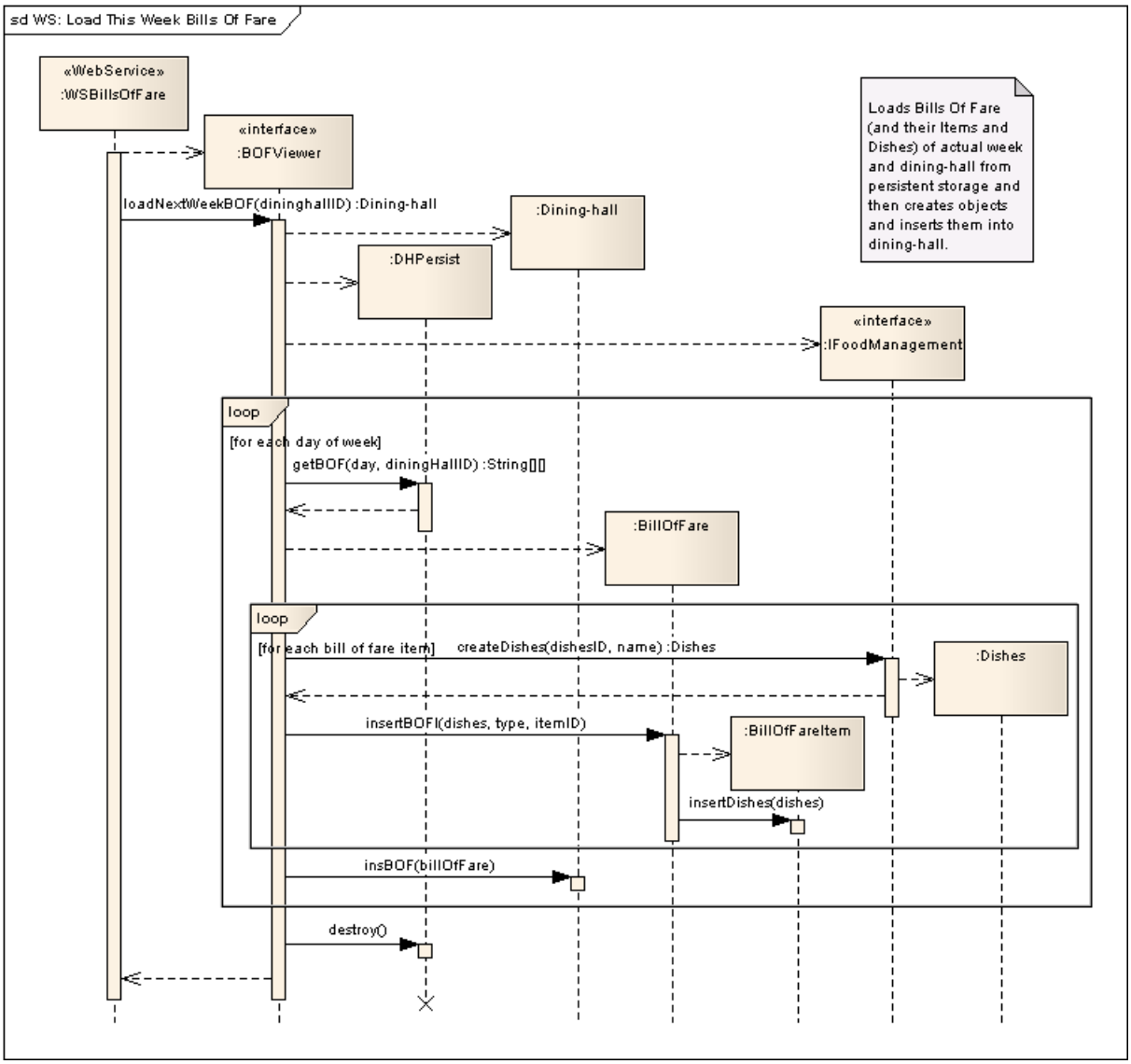
Sekvenční diagram Show Bill Of Fare Items (JSSI)



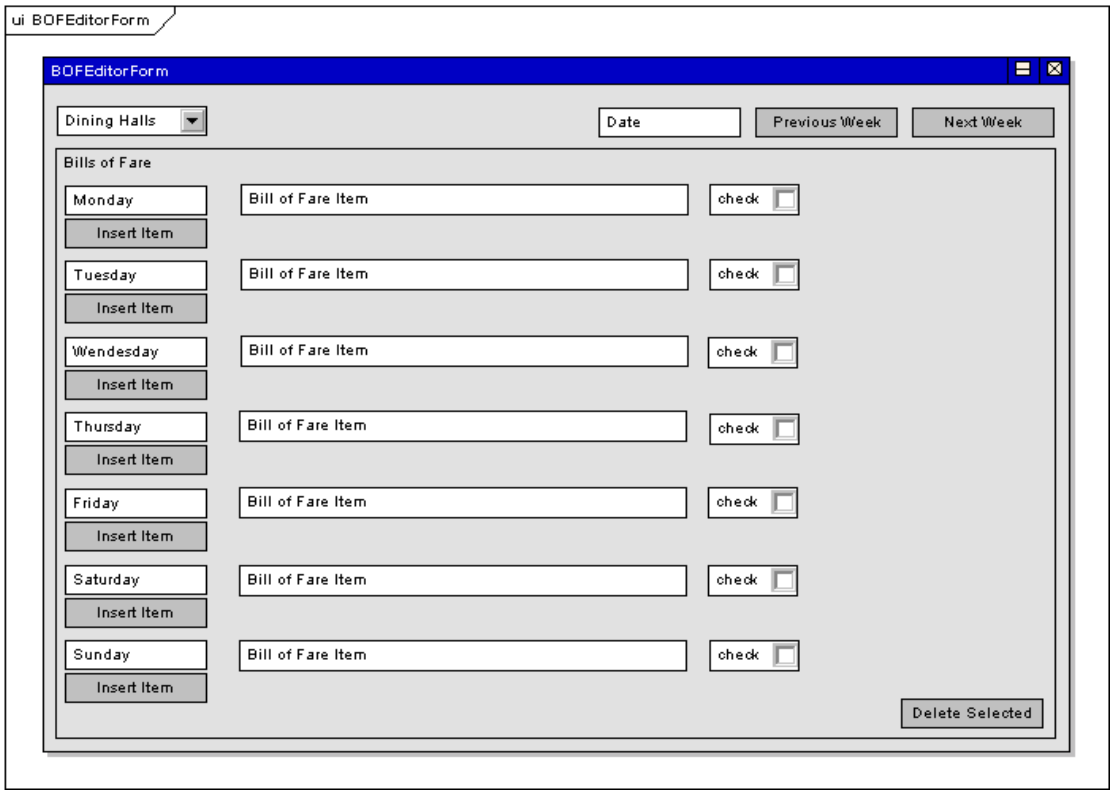
Sekvenční diagram Show All Dishes (JSSI)



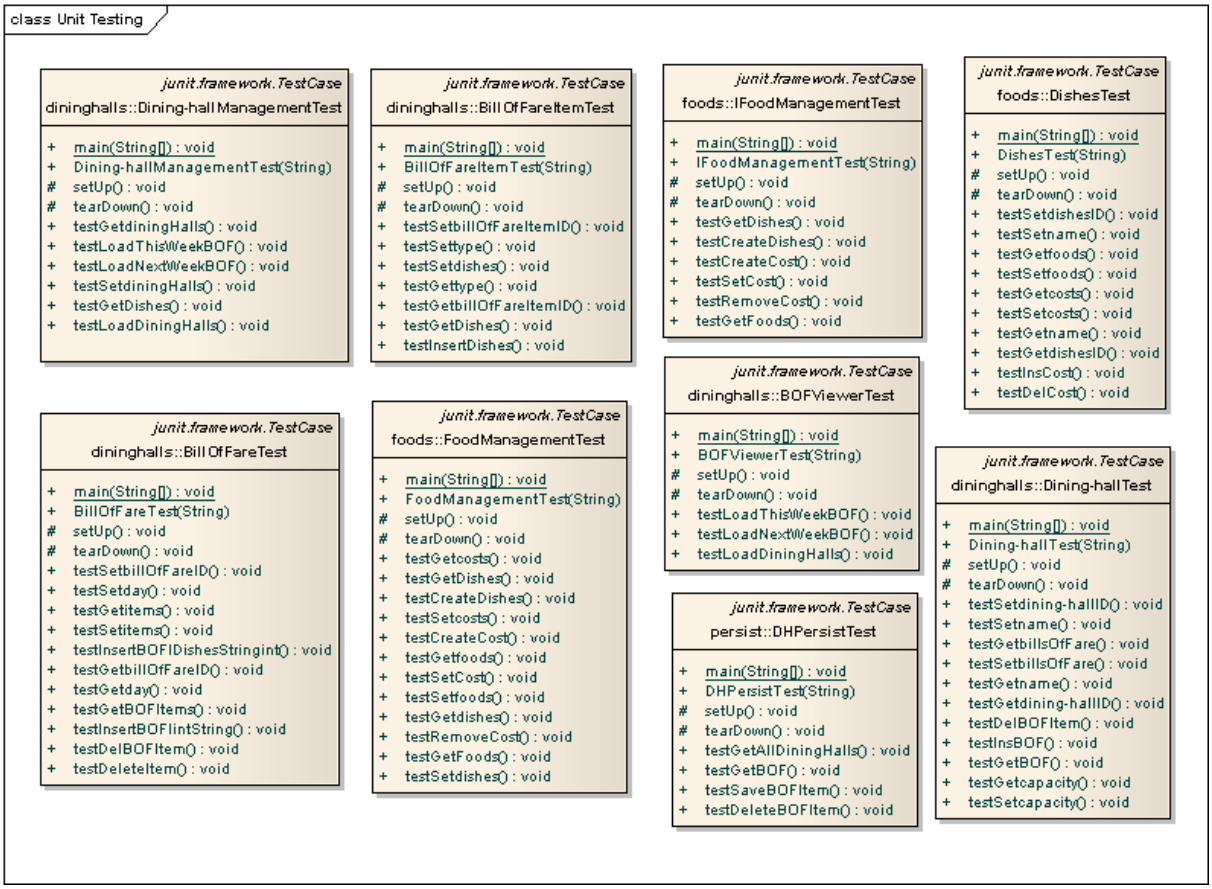
Sekvenční diagram Delete Bill Of Fare Item (JSSI)



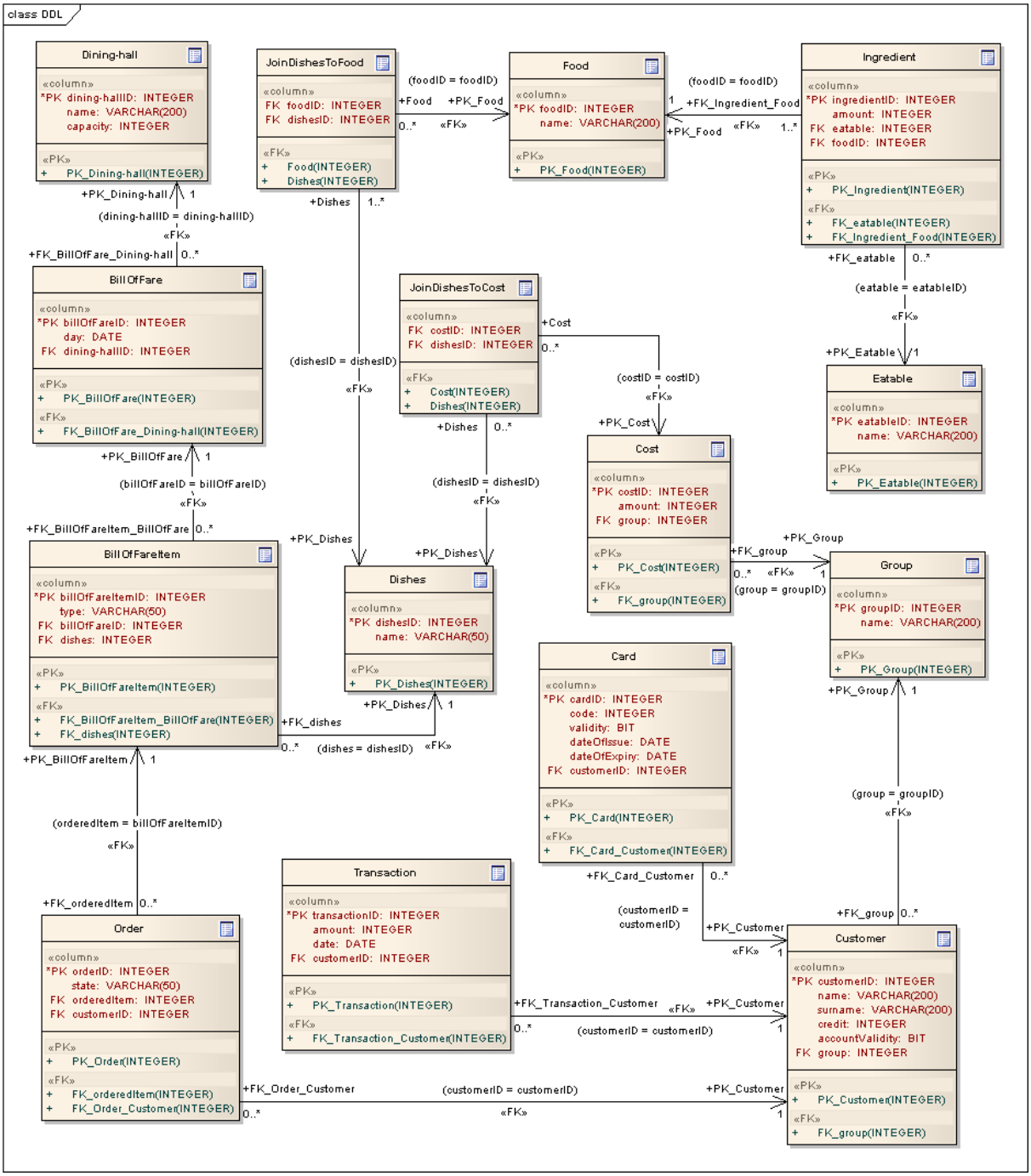
Sekvenční diagram WS: Load This Week Bills Of Fare (JSSI)



Návrh GUI: BOFEditorForm (PIM, JSSI)



Třídy JUnit (PSM, JSSI)



DDL model (PSM, JSSI)

Příloha F

Webová služba:

Method parameter(s)

Type	Value
int	1

Method returned

java.util.List : "[org.jssi.ws.BofItemInfo@119052d, org.jssi.ws.BofItemInfo@1a2fd1b, org.jssi.ws.BofItemInfo@19702b5, org.jssi.ws.BofItemInfo@132fa84, org.jssi.ws.BofItemInfo@1dcf9f4, org.jssi.ws.BofItemInfo@9e3fa6, org.jssi.ws.BofItemInfo@164d296, org.jssi.ws.BofItemInfo@3d8b15, org.jssi.ws.BofItemInfo@761d28, org.jssi.ws.BofItemInfo@744d21, org.jssi.ws.BofItemInfo@173d088, org.jssi.ws.BofItemInfo@7fcdfc, org.jssi.ws.BofItemInfo@ac86bb, org.jssi.ws.BofItemInfo@c16890, org.jssi.ws.BofItemInfo@1b805ec, org.jssi.ws.BofItemInfo@8d096f, org.jssi.ws.BofItemInfo@1667577, org.jssi.ws.BofItemInfo@1aa005a, org.jssi.ws.BofItemInfo@1fd3d92, org.jssi.ws.BofItemInfo@d54dac]"

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:thisWeekBOF xmlns:ns2="http://ws.jssi.org/">
            <diningHallID>1</diningHallID>
        </ns2:thisWeekBOF>
    </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:thisWeekBOFResponse xmlns:ns2="http://ws.jssi.org/">
            <return>
                <date>5. květen, 2008</date>
                <dishesName>Makový závin, mléko</dishesName>
                <itemType>snidane</itemType>
            </return>
            <return>
                <date>5. květen, 2008</date>
                <dishesName>Boršť, Kuře na paprice</dishesName>
                <itemType>obed</itemType>
            </return>
            <return>
                <date>5. květen, 2008</date>
                <dishesName>Cibulačka, Filé na kari</dishesName>
                <itemType>obed</itemType>
            </return>
            .
            .
            (zkráceno)
            .
            .
            <return>
                <date>9. květen, 2008</date>
                <dishesName>Chléb, med, pomeranč</dishesName>
                <itemType>vecere</itemType>
            </return>
        </ns2:thisWeekBOFResponse>
    </S:Body>
</S:Envelope>
```

Test webové služby informačního systému jídelen (JSSI). Metoda thisWeekBOF vrací jídelníčky pro aktuální týden a zvolenou jídelnu.

Příloha G

Obsah CD:

- **/JSSI/** – složka obsahuje výstupy z případové studie vytvořené v Enterprise Architectu.
- **/JSSI/DDL/** – generované SQL skripty se schématem databáze a skript s testovacími daty pro implementovanou webovou službu.
- **/JSSI/Diagrams/** – export diagramů do PNG obrázků.
- **/JSSI/Java/** – zdrojové kódy vygenerované z Java PSM modelu.
- **/WSDL/** – WSDL soubor pro popis webové služby.
- **/JSSI-dokumentace/** – HTML dokumentace případové studie generovaná pomocí Enterprise Architectu.
- **/PRIKLADY/** - složka obsahuje výstupy z ukázkových příkladů vytvořených v Enterprise Architectu (skripty, zdrojové kódy a diagramy).
- **/PRIKLADY-dokumentace/** - HTML dokumentace ukázkových příkladů generovaná pomocí Enterprise Architectu.
- **/WSBillsOfFareApplication/** - projekt webové služby vytvořený ve vývojovém prostředí NetBeans. Obsahuje zdrojové soubory, testovací soubory, atd. Implementované třídy a metody obsahují komentáře včetně JavaDoc.
- **/WSBillsOfFareClientApplication/** - klient pro otestování webové služby.
- **/JSSI.eap** – zdrojový soubor Enterprise Architectu pro projekt případové studie.
- **/priklady.eap** – zdrojový soubor Enterprise Architectu pro projekt s ukázkovými příklady.
- **/thesis.pdf** – text diplomové práce.

Upravené transformační skripty jsou součástí souborů *.eap. Pro prohlížení projektů Enterprise Architectu poskytuje firma Sparx Systems verzi s označením LITE. Tato verze je ke stažení na adrese: http://www.sparxsystems.com.au/bin/EALite.exe