název práce
Nástroje pro automatizaci tvorby uživatelského rozhraní
User interface design automation tools

katedra obhajoby
katedra počítačů

obor
Softwarové inženýrství

studijní program
Bakalářský

vedoucí
Černý Tomáš Ing., MSc.
katedra počítačů (13136)
cernyto3@fel.cvut.cz

oponent
Čemus Karel Ing.
katedra počítačů (13136)
cemuskar@fel.cvut.cz

student
Řežábek Michal (studuje) - zadáno
rezabmic@fel.cvut.cz
Softwarové inženýrství
studijní plán: Softwarové inženýrství (STM-A7B) (BPSTMSI)
katedra obhajoby podle studijního plánu: katedra počítačů

literatura
[1] Tomas Cerny, Michael J. Donahoo, and Eunjee Song. 2013. Towards effective adaptive user interfaces design. In Proceedings of the 2013 Research in Adaptive and Convergent Systems (RACS '13). ACM, New York, NY, USA, 373-380. DOI=10.1145/2513228.2513278
http://doi.acm.org/10.1145/2513228.2513278

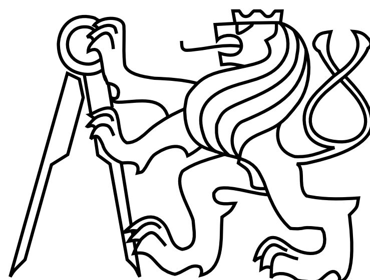
[2] Tomas Cerny, Karel Cemus, Michael J. Donahoo, and Eunjee Song. 2013. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. In Applied Computing Review, Vol. 13, Issue 4, ACM, New York, NY, USA, 53-65. ISSN 1559-6915
http://www.sigapp.org/acr/Issues/V13.4/ACR-13-4-2013.pdf

[3] Tomas Cerny and Eunjee Song. 2011. UML-based enhanced rich form generation. In Proceedings of the 2011 ACM Symposium on Research in Applied Computation (RACS '11). ACM, New York, NY, USA, 192-199. DOI=10.1145/2103380.2103420
http://doi.acm.org/10.1145/2103380.2103420

pokyny
Nastudujte existující nástroje pro automatizovanou tvorbu uživatelského rozhraní pro platformu Java EE. Zaměřte se na nástroje MetaWidget a AspectFaces a jejich možnosti adaptability pro různé typy uživatelů a zařízení. Navrhněte Java EE aplikaci, na které budou možnosti adaptability ukázány, implementujte ji a následně využijte nástroj MetaWidget a AspectFaces. Porovnejte výše zmíněné nástroje a vyhodnoťte jejich výhody a nevýhody.

zadavatel
CodingCrayons s.r.o.

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Nástroje pro automatizaci tvorby uživatelského rozhraní

Michal Řežábek

Vedoucí práce: Ing. Tomáš Černý, MSc.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

23. května 2014

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Tomáši Černému, MSc. za zajímavé téma bakalářské práce, jehož realizací jsem získal mnoho nových znalostí a zkušeností.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 23. 5. 2014 ....

Abstract

This thesis introduces and compares techniques for automatic graphic user interface generation and actively developed tools for the Java EE platform which implement them. Comparison is made on practical usability and focuses primarily on the tools AspectFaces and Metawidget. One of the comparison criterion is, due to a rising number of users and platforms using web applications, the ability of the tool to make advanced adaptations to the generated user interface. Tools AspectFaces and Metawidget are presented on a real application.

Abstrakt

V této práci jsou představeny a navzájem porovnány techniky automatizovaného vytváření grafického uživatelského rozhraní a aktivně vyvíjené nástroje pro platformu Java EE, které tyto techniky implementují v praxi. Porovnání je zaměřeno především na nástroje AspectFaces a Metawidget a možnosti jejich praktického použití. Díky vzrůstajícímu počtu uživatelů a platforem využívajících webové aplikace, je jedním z kritérií porovnání také schopnost těchto nástrojů provádět pokročilé adaptace generovaného uživatelského rozhraní. Použití nástrojů AspectFaces a Metawidget je prezentováno na reálné aplikaci.

Obsah

1 Úvod 1

2 Kritéria porovnání 3
2.1 Produktivita . . . 3
2.2 Znovupoužitelnost . . . 3
2.3 Flexibilita . . . 3
2.4 Povědomí o kontextu . . . 3
2.5 Low threshold a high ceiling . . . 4
2.6 Nezávislost na konkrétní technologii . . . 4

3 Techniky tvorby UR 5
3.1 Restate-to-extend techniky . . . 5
3.1.1 Statické generátory kódu . . . 6
3.1.2 Grafické nástroje . . . 6
3.1.3 Modelovací jazyky . . . 6
3.2 Inspection-based techniky . . . 6
3.2.1 Model driven development . . . 7
3.2.1.1 Externí modely . . . 7
3.2.1.2 Domain driven design . . . 7
3.2.2 Object/User Interface Mapping . . . 8
3.2.3 Rich entity aspect/audit design . . . 8
3.2.3.1 Klíčové pojmy AOP . . . 9
3.2.3.2 Zpracování READ komponent . . . 9
3.3 Vyhodnocení technik . . . 10

4 Adaptace UR 13
4.1 Techniky adaptace obsahu . . . 14
4.1.1 Prezentace relevantního obsahu . . . 14
4.1.2 Responsive web design . . . 14

5 Nástroje pro runtime generování UR 15
5.1 OpenXava . . . 15
5.2 Apache Isis . . . 16
5.3 Metawidget . . . 16
5.3.1 Architektura . . . 16
5.3.2 Inspectory . . . 17

5.3.3 InspectionResultProcessory . . . 17
5.3.4 WidgetBuildery . . . 18
5.3.5 WidgetProcessory . . . 18
5.3.6 Layouty . . . 18
5.4 AspectFaces . . . 18

6 Case study: Aplikace Cost Report **19**
6.1 Doménový model . . . 20
6.2 Funkční požadavky . . . 24
6.3 Nefunkční požadavky . . . 25
6.4 Případy použití . . . 26
6.5 Implemetace . . . 27
6.5.1 Responsivní tabulky . . . 27
6.6 Testování . . . 28
6.6.1 Unit testy . . . 28
6.6.2 Integrační testy . . . 28
6.6.3 Selenium testy . . . 28

7 Generování UR **29**
7.1 Rich entity . . . 29
7.2 Inspekce metadat . . . 31
7.2.1 Inspekce v AspectFaces . . . 31
7.2.2 Inspekce v Metawidgetu . . . 32
7.3 Zpracování metadat . . . 33
7.4 Mapování atributů na widgety . . . 34
7.4.1 Mapování v Metawidgetu . . . 34
7.4.2 Mapování v AspectFaces . . . 34
7.5 Vytváření widgetů . . . 35
7.5.1 Vytváření widgetů v Metawidgetu . . . 35
7.5.2 Vytváření widgetů v AspectFaces . . . 36
7.6 Layout . . . 37
7.6.1 Layout v Metawidgetu . . . 37
7.6.2 Layout v AspectFaces . . . 38
7.7 Konfigurace . . . 39
7.7.1 Konfigurace Metawidgetu . . . 39
7.7.1.1 Konfigurační třídy . . . 40
7.7.2 Konfigurace AspectFaces . . . 41
7.8 Použití v UR . . . 43
7.8.1 Použití Metawidgetu . . . 43
7.8.2 Použití AspectFaces v UR . . . 43
7.9 Možnosti adaptace UR . . . 44
7.9.1 AspectFaces . . . 44
7.9.2 Metawidget . . . 45
7.10 Výsledná podoba UR . . . 46
7.11 Vyhodnocení . . . 46

8 Závěr **49**

A Popis případů užití **55**

B Obsah přiloženého CD **57**

Seznam obrázků

3.1 READ framework [18] 10

5.1 Metawidget pipeline [22] 17

6.1 Doménový model 20
6.2 Diagram případů použití 26
6.3 Explicitní zobrazení sloupců uživatelem 28

7.1 UR aplikace Cost Report 46

A.1 CRUD operace 56

Seznam tabulek

3.1 Výsledné srovnání technik 11

6.1 Výčet atributů doménového objektu Auto 21
6.2 Výčet atributů doménového objektu Systém 21
6.3 Výčet atributů doménového objektu Komponenta 21
6.4 Výčet atributů doménového objektu Spojovací prvek 22
6.5 Výčet atributů doménového objektu Materiál 22
6.6 Výčet atributů doménového objektu Provedená práce 23
6.7 Výčet atributů doménového objektu Nástroj 23

7.1 Typy AnnotationDescriptorů 31
7.2 Vyhodnocení řádků kódu 46

Kapitola 1

Úvod

Grafické uživatelské rozhraní, které budu dále označovat pouze zkratkou UR, je již řadu let nejpoužívanějším typem uživatelských rozhraní sloužícím ke vzájemné interakci uživatele a aplikace. Nespočet výzkumů v této oblasti dal vzniknout rozmanité škále technik vytváření UR. Jednotlivé techniky popisují více či méně konkrétním způsobem postup vytváření UR. Některé z nich se soustřeďují pouze na použití ve výzkumných projektech, jiné jsou orientovány spíše prakticky.

V poslední době však došlo ve výzkumu UR k radikální změně související s nárůstem uživatelů a nových platforem. Výzkum technologií pro tvorbu UR se začal soustředit na možnost vytváření plastických UR, které mění svou podobu na základě aktuálního kontextu použití za účelem maximalizace jejich použitelnosti. Již dlouhodoběji je v hledáčku některých výzkumů také problém duplikace informací z vrstvy doménových objektů, která je nedílnou součástí postupu vytváření UR určité skupiny technik.

UR jsou vytvářena čím dál komplexnější, obsahují řadu interaktivních prvků a čas strávený jejich tvorbou tvoří značnou část vývoje softwarové aplikace. Jednou z možností, jak proces vytváření UR zefektivnit, je nevytvářet UR manuálně, ale vymyslet způsob, jak tento proces automatizovat. Správně provedená automatizace by navíc mohla představovat řešení problému duplikace informací a při vytváření UR by mohla mít přístup k runtime metadatům, na základě kterých by mohla provádět pokročilé adaptace.

Cílem této bakalářské práce je provést porovnání nástrojů pro automatizovanou tvorbu UR pro platformu Java EE (Java Platform, Enterprise Edition) se zaměřením na nástroje AspectFaces a Metawidget a jejich možnosti adaptace. Jelikož tyto nástroje představují pouze implementaci konkrétní techniky vytváření UR, první pozornost bude věnována právě porovnání jednotlivých technik vytváření UR.

Pro účely porovnání nástrojů AspectFaces a Metawidget byla vytvořena aplikace Cost Report, která realizuje skutečné požadavky na systém pro správu součástek stanovené týmem CTU Cartech. Aplikace byla vytvořena ve třech verzích. UR první verze je vytvořeno manuálně za pomocí frameworku JavaServer Faces, druhé pomocí AspectFaces a třetí pomocí Metawidgetu. Výsledné porovnání je vytvořeno na základě praktických zkušeností získaných při vytváření této aplikace a také na základě analýzy výsledného kódu UR jednotlivých verzí.

Kapitola 2

Kritéria porovnání

2.1 Produktivita

Vytváření UR představuje značnou část vývoje softwarovarové aplikace. Průzkum [26] provedený v roce 1992 ukázal, že kód UR tvoří 48% celkového aplikačního kódu a jeho vytváření tvoří 50% celkového času vývojové fáze projektu. Možná úspora času v této části vývoje SW je proto velice žádaná.

2.2 Znovupoužitelnost

Znovupoužitelnost fragmentů UR redukuje čas strávený vytvářením UI a příznivě tak ovlivňuje produktivitu dané techniky vytváření UR. Kromě již zmíněné produktivity však přináší znovupoužitelnost také další výhody. Díky oddělení definice widgetu[1] od zbytku UR je snadné dosáhnout konzistentního UR a zlepšit tak použitelnost vytvářeného UR.

2.3 Flexibilita

Důležitým kritériem je také flexibilita, tedy možnost dosažení požadovaného vzhledu UR. R. Kennard [24] zmiňuje, že „pokud není možné dosáhnout požadovaného vzhledu UR, trpí tím použitelnost UR - rozhodující faktor UR pro automatizované vytváření UR“.

2.4 Povědomí o kontextu

Povědomí o kontextu (context awareness) zastupuje myšlenku, že „aplikace může porozumět kontextu, vyhodnocovat jej a na základě těchto znalostí provádět adekvátní operace. Navíc při změně kontextu by se aplikace měla přizpůsobit novým okolnostem“ [30]. Calvary et al. [20] definuje kontext použití jako směsici tří entit: uživatelů systému, hardwarových a softwarových platforem, které mohou být použity k interakci se systémem a v neposlední řadě

[1] Widgety jsou nazývány jednotlivé elementy grafického uživatelského rozhraní [11].

také fyzického prostředí, ve kterém interakce probíhá. Přizpůsobení se kontextu je označováno jako adaptace. Problematiku adaptace je možné rozdělit na multi-targeting a plasticitu [20]. „*Multi-targeting* se zaměřuje na technické aspekty adaptace, zatímco *plasticita* poskytuje způsob, jak kvalifikovat použitelnost systému po adaptaci“ [20]. Transformace, které jsou prováděny v průběhu adaptace UR bývají označovány jako *UI remolding* [21]. Tyto transformace zahrnují odstranění widgetů, které se díky změně kontextu stanou irelevantní, přidání nových relevantních widgetu, nahrazení stávajícího widgetu jiným typem widgetu nebo reorganizaci widgetů prostřednictvím změny layoutu [21].

2.5 Low threshold a high ceiling

Myers et al. [25] popisuje ideální nástroj jako ten, který má „low threshold and high ceiling“. Threshold - v překladu práh - představuje potřebné úsilí vynaložené k naučení se daný nástroj používat. Ceiling - v překladu strop - představuje propracovanost výsledku, který pomocí nástroje dostaneme. Ideálním stavem je za co nejméně vynaložené práce dostat co největší užitek.

2.6 Nezávislost na konkrétní technologii

Technologie pro tvorbu UR mají tendenci se často měnit. Nezávislost na technologiích umožňuje využít širší škálu existujících technologií a také schopnost přizpůsobit se nově vznikajícím.

Kapitola 3

Techniky tvorby UR

UR může být vytvářeno manuálně nebo tento proces může být automatizován. Automatizované techniky patří do skupiny automatizovaného programování, které je definováno jako syntéza programu ze specifikace [5]. V tomto případě je výsledkem syntézy UR. Použitelnost automatizovaných technik závisí do značné míry na složitosti vytvoření výše zmíněné specifikace [5].

Nápad generovat UR na základě specifikace se zrodil již koncem 70 let. Netrvalo to dlouho a začaly se objevovat první model based systémy [17]. Prvotním cílem bylo zjednodušit a zrychlit proces vytváření UR, avšak následně se přidaly také další požadavky na podporu nově objevujících se platforem a schopnost adaptace na konkrétní kontext použití [17]. Nové platformy přinášejí nová runtime metadata, která otevírají v minulosti téměř nepředstavitelné možnosti pro vytváření adaptivních UR.

Jednotlivé techniky tvorby UR mohou vytvářet UR *staticky*, *dynamicky* nebo také oběma způsoby. Statické vytváření UR probíhá ve fázi vývoje. Ve fázi vývoje však chybí informace o daném kontextu použití, což má za následek nemožnost vytvářet dostatečně plastické UR, které by se dokázalo přizpůsobit na míru aktuálním potřebám uživatele. Dynamické vytváření UR probíhá v runtimu. Díky tomu mohou být k vytvoření specifikace UR využity také runtime metadata nesoucí informaci o kontextu použití.

Techniky tvorby UR lze rozdělit na *restate-to-extend* a *inspection based* [19].

3.1 Restate-to-extend techniky

Restate-to-extend přístup vyžaduje při tvorbě UR duplikaci informací z doménové vrstvy. Tato duplikace způsobuje porušení jednoho ze základních programovacích principů, a sice Dont repeat yourself (DRY). Následkem je značné prodloužení procesu vytváření UR a také jeho následné správy, čímž se zvětšuje prostor pro provedení chyb, které jsou často způsobené nepozorností programátora [23]. Další nevýhodou je, že není možné separovat opakující se koncerny UR (layout, prezentace, zabezpečení a další). Následkem toho je nepřehledný a nesnadno spravovatelný kód, který není možné opětovně používat.

Mezi zástupce restate-to-extend technik vytváření UR patří statické generátory kódu, grafické nástroje, doménově specifické modelovací jazyky a také MDD-Externí modely.

3.1.1 Statické generátory kódu

Statické generátory kódu, jako například Spring Roo, dokáží vytvořit UR velmi rychle s minimální úsilím programátora. Výsledkem je však často pouze základní UR (v případě Spring Roo CRUD aplikace).

„Vytváření UR je plné estetiky a psychologie a nelze očekávat, že se dá vygenerovat kompletní UR splňující veškeré naše představy. Automatizace by měla být používána pouze pro určité části UR a její výsledek by měl bez jakéhokoliv viditelného rozdílu zapadnout mezi manuálně vytvářené fragmenty UR“ [9].

Jedním z důvodů je, že statický generátor nemá dostatek metadat k vytvoření UR podle našich představ. V případě manuálního přizpůsobení vygenerovaného UR našim představám dokáže být regenerování velmi pracné. Kvůli výše zmíněným důvodům by nemělo být generování UR prováděno staticky, ale dynamicky (v runtimu).

3.1.2 Grafické nástroje

Grafické nástroje představují techniku tzv. vizuálního programování. Vytváření UR probíhá stylem drag-and-drop, kde uživatel vybere požadovaný widget a umístí ho na správné místo v editoru. Uživatelský vstup je poté zpracován a pomocí statického generátoru kódu transformován do výsledné podoby. Tato technika poskytuje uživateli dobrý přehled o vytvářeném UR, jelikož reprezentace UR v editoru odpovídá výsledné podobě UR. Tento fakt bývá označován zkratkou WYSIWYG - What You See Is What You Get. Ovládání těchto nástrojů je typicky velmi snadné a intuitivní, občas dokonce i zábavné. Celkový proces vytváření však dokáže být velmi pracný a k dosažení precizního výsledku je zapotřebí vynaložit značné úsilí. Problémem je především nesnadné dosažení konzistentního zobrazení widgetů v UR [24].

3.1.3 Modelovací jazyky

Většina UR webových aplikací je vytvářena pomocí domain specific modelovacího jazyka. Tento deklarativní přístup poskytuje největší kontrolu na vytvářeným UR. Můžeme přesně vyjádřit veškerá validační kritéria, upravovat vzhled pomocí kaskádových stylů (CSS) a přidávat interaktivní chování pomocí Javascriptu. Nevýhodou je požadovaná znalost hned několika domain specific jazyků a také časová náročnost. Oproti těmto nevýhodám se pyšní také mnoha výhodami. Představují časem prověřený způsob tvorby UR. Existuje široká škála implementací těchto technik, z nichž mnohé jsou součástí již fungujících, rozsáhlých a úspěšných projektů.

3.2 Inspection-based techniky

Inspection-based techniky využívají k tvorbě UR (nebo jeho části) existující metadata z vrstvy doménových objektů. K získání těchto metadat využívají sub-disciplínu reversního inženýrství zvanou software mining, konkrétně nejběžnější přístup dynamické analýzy, který představuje reflexe [24]. Řeší tak problém duplikace informací, se kterým se potýkají restate-to-extend techniky. I přes široké možnosti, které reflexe přináší, ne všechny informace lze

získat pomocí inspekce [24]. Příkladem může být pořadí atributů třídy, které v Javě není možné pomocí inspekce zjistit. Proto je třeba tyto informace explicitně specifikovat ve formě metadat. V jazyku Java k tomuto účelu slouží anotace. Nutnost explicitní specifikace metadat však zvyšuje komplexitu kódu a představuje tak nevýhodu inspection-based technik.

3.2.1 Model driven development

Model driven development (MDD) je založen na myšlence vytvoření pouze modelu aplikace, ze kterého je poté automaticky vygenerována zbylá část aplikace - tedy i UR. Model může mít grafickou či textovou podobu. Techniky, které využívají grafickou reprezentaci modelu pomocí UML tvoří podskupinu MDD technik nazývanou Model Driven Architecture (MDA). Model může být vyjádřen pomocí několika vrstev abstrakce, které umožňují popsat i složité závislosti mezi komponentami UR. Na každou vrstvu je navíc možné aplikovat různé typy adaptace.

Myers et al. [25] však upozornil na dva problémy související s automatizovaným vytvářením UR na základě modelů. Prvním problémem je, že tato technika neposkytuje dobrý přehled o vytvářeném UR. Druhý problém představuje velký threshold(viz podkapitola 2.5) způsobený nutností naučit se modelovací jazyk. Navíc generování kompletního UR se neobejde bez omezení flexibility, jelikož vytváření UR představuje rozsáhlou problematiku, kterou současné technologie neumožňují realizovat se zachováním stejné flexibility, jakou disponují manuální techniky [25] [24].

Tento přístup se dále dělí na:

- **Static modelling** – statický přístup vytváření UR
- **Generative runtime modelling** – dynamický přístup vytváření UR umožňující jednoduché adaptace. Provádí generování kódu.
- **Interpreted runtime modelling** – dynamický přístup vytváření UR, který neprovádí generování kódu. Namísto toho interpretuje UR v runtimu pomocí application enginu. Umožňuje tak pokročilejší adaptace a deploy změn nevyžaduje rekompilaci aplikace.

3.2.1.1 Externí modely

Speciálním případem MDD je využívání modelu k vytvoření pouze části aplikace, jako například právě UR. Za těchto okolností se nelze vyhnout duplikaci informací z vrstvy doménových objektů, a proto tento případ spadá pod restate-to-extend techniky.

3.2.1.2 Domain driven design

Domain driven design (DDD) spadá také do kategorie MDD technik. Jedná se spíše o lightweight MDD, jelikož model tvoří pouze jedna vrstva abstrakce, což je typicky vrstva doménových objektů vyjádřených pomocí objektového jazyka. Hlavním výhodou tohoto přístupu je možnost soustředit se na problémovou doménu. Tuto techniku hojně využívají nástroje určené pro rychlý vývoj (rapid development). V průběhu vývoje se formuje tzv. *ubiquitous*

language - jazyk, kterým spolu komunikují jak programátoři, tak i znalci problémové domény. Tento jazyk je používán také v kódu, čímž zlepšuje jeho čitelnost a srozumitelnost. Z tohoto důvodu je kód pochopitelný i pro znalce problémové domény. S přibývající znalostí problémové domény je postupně vytvářen také kód.

Raneburger [29] však zmiňuje, že „bez explicitní specifikace detailních informací jako například layoutu či designu je výsledné UR těžko vyhovující reálným požadavkům.“ Problémem je, že v doménových objektech je možné vyjádřit pouze omezené množství informací a zachycení požadavků na design či použitelnost tak není reálné. Tento přístup je proto vhodný v případě, že není vyžadováno řešení na míru (custom UR) a vystačíme si s generickým UR. Ušetříme tak spoustu času, který bychom jinak věnovali údržbě UR.

3.2.2 Object/User Interface Mapping

Object/User Interface Mapping (OIM) je technika, která na základě inspekce existující back-end architektury provádí mapování objektů na widgety UR vytvářené pomocí programátorem zvoleného front-end frameworku.

Úkolem OIM není vytvořit kompletní UR, ale pouze automatizovat vytváření těch widgetů, které by za použití restate-to-extend přístupu obsahovaly duplikaci informací z doménové vrstvy [9]. Automatizace procesu vytváření takových komponent přináší mnoho ušetřeného času a pomáhá předcházet možným chybám, které mohou vzniknout díky duplikaci kódu. Důležité však je, aby technika, která automatizaci provádí, dokázala vytvářet UR na základě reálných požadavků a zachovala tak flexibilitu manuálních technik.

OIM technika je založená na podpoře široké škály back-end a front-end technologií a jednoduché rozšířitelnosti či dokonce vytvoření vlastní implementace pro zatím nepodporované technologie. Tímto přístupem se snaží omezit závislost na konkrétní technologii a nabídnout tak možnost využití této techniky co největšímu počtu programátorů. Podpora široké škály technologií pomáhá vytvořit architekturu takovým způsobem, aby byla co nejflexibilnější [10].

OIM umožňuje generovat UR jak staticky, tak i dynamicky. K mapování mohou být využity nejen existující business objekty, ale také netradiční zdroje, jako například reprezentace objektu v JSON formátu.

3.2.3 Rich entity aspect/audit design

Rich entity aspect/audit design (READ) je inspection-based technika založená na aspektově orientovaném programování (AOP). V generalized procedure (GP) jazycích máme k dispozici pouze jednu možnost dekompozice problému - funkcionální a jeden typ kompozice - volání funkcí. AOP přináší další element - *aspekt*. Aspekt představuje systémovou vlastnost prolínající se několika systémovými komponentami, kterou nelze zapouzdřit do samostatného celku. Příkladem aspektů UR je layout, prezentace formulářových polí nebo zabezpečení. AOP umožňuje takové aspekty izolovat, skládat a opětovně využívat. O propojení aspektů a komponent, které je v AOP označováno jako *weaving*, se stará *aspect weaver*. Výsledkem může být stejná směsice obou druhů funkčních jednotek. Aspekty jsou však definovány odděleně, a tak se zlepšuje čtení, znovupoužitelnost i údržba kódu [19]. Předmětem inspekce jsou

doménové objekty (entity) rozšířené o další potřebné informace formou anotací nazývané *rich entity*. Generování UR může být provedeno jak staticky, tak i dynamicky.

3.2.3.1 Klíčové pojmy AOP

Join point – dobře definované snadno identifikovatelné místo v kódu, na které lze aplikovat aspekty. READ rozlišuje statické a dynamické join pointy. *Statické join pointy* tvoří specifické elementy v kódu data modelu, které představují jména tříd, názvy a typ jejich atributů a také anotace. Všechny tyto informace jsou dostupné v době kompilace a jejich získání probíhá při inspekci. *Dynamické join pointy* tvoří informace získané v runtimu jako například geolokace, velikost obrazovky zařízení nebo uživatelská role.

Pointcut language – jazyk využívaný ke specifikaci join pointů, na které má být aplikován určitý aspekt. READ k tomuto účelu využívá Unified Expression Language [12] (EL). Dotazování na informace z READ kontextu je realizováno díky možnosti EL kontextu přistupovat k READ kontextu.

Pointcut – specifikace konkrétních join pointů, na které má být aplikován určitý aspekt. Příklad: `${email == true}` - vybere všechny atributy, které slouží k uchovávání emailu.

Advice – vyjádření aspektu, který má být aplikován. V READ je advice použit například ke specifikaci šablony pro prezentaci fieldu, která je součástí prezentačních pravidel. Příklad: `tag="emailTag.xhtml"` - k prezentaci bude použita šablona emailTag.xhtml.

3.2.3.2 Zpracování READ komponent

UR si můžeme představit jako strom komponent. Každá komponenta má asociován handler, který je při vykreslování UR volaný rendererem. READ lze jednoduše napojit na tento proces pomocí implementace vlastního handleru. Když tedy zavolá renderer na READ komponentě její handler, předá se zodpovědnost za zpracování READ frameworku.

READ handler dostane referenci na instanci, kterou má zpracovat, a případně také další atributy blíže specifikující konkrétní způsob zpracování. Poté provede inspekci renderované instance. Cílem inspekce je získat statické join pointy a poskytnout je READ kontextu [19]. Inspekce je prováděna pomocí Reflective API a jejím výsledkem je metamodel reprezentovaný pomocí stromové struktury. Kořen stromu tvoří instance informace, jeho uzly jsou pak tvořeny atributy získanými pomocí inspekce a listy stromu reprezentují statické joinpointy.

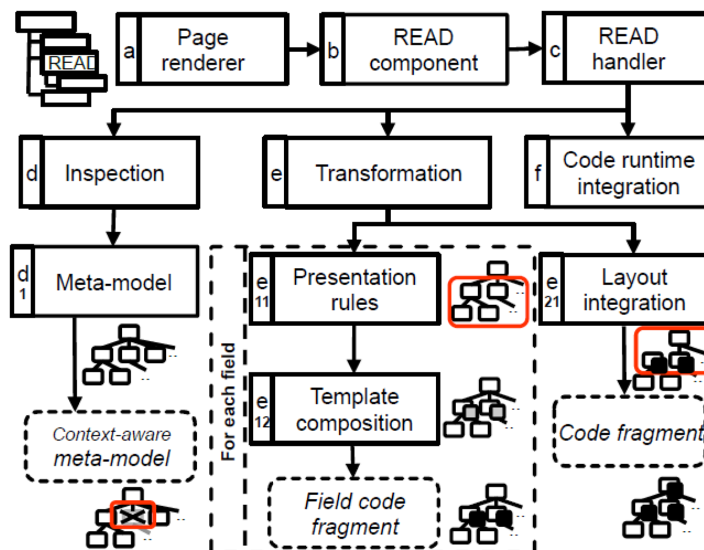
Některé elementy mohou být dále filtrovány pomocí Annotation Driver Participant Patternu (ADPP) na základě konkrétních požadavků na renderování nebo zabezpečení. Výsledný metamodel, označovaný jako *context-aware metamodel*, je poté předán READ kontextu ve formě join pointů.

Následuje dvoufázový proces transformace metamodelu na UR komponenty řízený *transformation weaverem*. Tento proces probíhá pro každý atribut zvlášť. V první fázi je pomocí *prezentačních pravidel* vybrána vhodná šablona pro prezentaci atributu. Prezentační pravidla tvoří dvojice pointcut a advice. Pointcut specifikuje typ atributů, kterých se daný způsob prezentace týká a advice specifikuje generickou šablonu pro prezentaci. Specifikace generické šablony pro renderování komponent je prováděna pomocí doménově specifického jazyka. Další

nedílnou součásti šablony je integrace jednotlivých aspektů řízená pomocí pointcutů. Pointcuty specifikují, zda má být daný aspekt integrován či nikoliv.

Po dokončení transformace všech atributů následuje integrace layoutu. Layout popisuje uspořádání widgetů ve výsledném UR. Jeho struktura je podobně jako u šablony pro prezentaci, popsána pomocí doménově specifického jazyka doplněného o pro READ specifické tagy, které označují místo v layoutu náležící konkrétnímu či anonymnímu (nijak nekonkretizovanému) widgetu a umožňují také iterovat přes anonymní widgety.

Popsaný mechanizmus zpracování READ komponent je velmi snadno škálovatelný. Přidání dalších aspektů je velice snadné. AOP v dnešní době využívá k vytváření UR pouze tato technika. Díky separace jednotlivých aspektů přináší nejen přehledný kód snadný na údržbu, ale především zvyšuje znovupoužitelnost jednotlivých fragmentů UR. Generické šablony jsou odděleny od layoutu a dalších aspektů, které jsou specifické pro konkrétní řešení UR. Stačí je tak definovat pouze jednou. Čas strávený tvorbou UR se tak významně snižuje.



Obrázek 3.1: READ framework [18]

3.3 Vyhodnocení technik

Manuální techniky

Manuální techniky (modelovací jazyky) se nevyhnou duplikaci existujících informací z vrstvy doménových objektů. Neumožňují separovat cross-cutting concerny UR, díky čemuž je možnost znovupoužití komponent UR jen velmi malá a s tím souvisí také pracné dosažení konzistence UR. Problémem jsou také pouze omezené možnosti adaptace. Výhodou manuálních technik je velká flexibilita.

Poloautomatizované techniky

Poloautomatizované techniky (grafické nástroje) nevyužívají automatizaci ke zefektivnění procesu vytváření UR, ale k jeho zjednodušení a zpřehlednění. Sdílí tak problémy manuálních technik a navíc přidávají také menší flexibilitu.

Automatizované restate-to-extend techniky

Automatizované restate-to-extend techniky (MDD-externí modely, statické generátory) jsou v porovnání s manuálními a poloautomatizovanými technikami efektivnější, ale díky duplikaci informací z doménové vrstvy stále ne dostatečně. Mezi výhody MDD-externích modelů patří možnost aplikovat adaptace na kteroukoliv vrstvu abstrakce modelu a také dobrá znovupoužitelnost jednotlivých modelů. Threshold této techniky je ale díky nutnosti naučit se nový modelovací jazyk docela velký. Největší překážku pro praktické použití MDD-externích modelů však představuje pomalé generování UR, které se pohybuje v řádu sekund. Statické generátory zase nemají přístup k runtime metadatům a navíc znesnadňují následnou údržbu UR - v případě provedení změn ve vygenerovaném UR je jeho následné přegenerování velmi náročné. Obě tyto techniky navíc generují kompletní UR a potýkají se tak s problémem nedostatečné flexibility.

Automatizované inspection-based techniky

Automatizované inspection-based techniky (DDD, OIM, READ) jsou nejefektivnější. Domain driven design neumožňuje zachytit širokou problematiku vytváření UR v jediné vrstvě. Vytváří tak pouze generické UR, což ovšem ne vždy musí představovat problém. Pokud však generické UR představuje problém, je vhodné využít OIM nebo READ k runtime generování částí UR, které se nejvíce potýkají s duplikací informací z vrstvy doménových objektů.

	Grafické nástroje	Statické generátory	Modelovací jazyky	MDD vícevrstvé modely	DDD	OIM	READ
Produktivita	5	1	5	3	1	2	2
Znovupoužití	5	5	4	3	1	1	1
Flexibilita	3	5	1	3	4	2	1
Povědomí o kontextu	5	5	3	1	5	2	2
Threshold	1	1	5	4	3	3	2
Technologická nezávislost	5	5	5	1	3	1	1

Tabulka 3.1: Výsledné srovnání technik

Hodnocení: 1-5
1 - nejlepší
5 - nejhorší

Kapitola 4

Adaptace UR

Uživatel posuzuje kvalitu aplikace především podle použitelnosti UR. Použitelnost UR může být posuzována například z hlediska designu, jednoduchosti navigace nebo relevantnosti prezentovaného obsahu. Adaptivní systémy nabízejí alternativu k tradičnímu „one design fits all“ přístupu ke tvorbě UR. Jejich na první pohled jednoduchá myšlenka spočívá v přizpůsobení se uživateli místo toho, aby se uživatel přizpůsoboval jim [27].

Cílem systémů, které dokáží přizpůsobit své UR na míru aktuálním potřebám uživatele, je maximalizovat použitelnost UR pro konkrétní kontext použití. Prvním krokem ke splnění tohoto cíle je získání ideálně co největšího množství metadat popisujících daného uživatele a kontext použití. Získaná metadata jsou poté (za pomocí data miningových metod) zpracována za účelem vytvoření modelů kontextu použití. Vytvořené modely jsou následně používány k sestavení prezentačních pravidel řídících proces adaptace. Mnohé z metadat jsou však dostupné až v runtimu, a proto vzniká požadavek pro adaptivní systémy na runtime generování UR.

Podle způsobu provádění adaptace lze systémy, které umožňují adaptovat své UR na základě kontextu použití, rozdělit na *adaptivní*, *adaptabilní* a *kombinované* [31]. Adaptivní systémy provádějí adaptaci bez zásahu uživatele, zatímco u adaptabilních systémů provádí adaptaci uživatel manuálně. Kombinované systémy představují kombinaci obou zmíněných přístupů [31].

Problém adaptace lze rozdělit na následující podproblémy:

Adaptace obsahu - zahrnuje vybrání relevantního obsahu k prezentaci a optimální techniky prezentace vybraného obsahu.

Adaptace textu - úkolem je prezentace relevantního obsahu

Adaptace prezentace - úkolem je optimalizovat způsob prezentace obsahu

Adaptace navigace - slouží k vytvoření efektivní navigace, která pomůže uživatelům lépe se orientovat na webu a snadno najít cestu k dosažení požadovaného cíle.

4.1 Techniky adaptace obsahu

Page variants – existuje několik variant stejné stránky, přičemž každá stránka je vhodná pro určitý typ uživatelů. Varianta vyhovující konkrétnímu uživateli je vybrána dynamicky.

+ nejjednodušší způsob adaptace obsahu
– malá škálovatelnost
– časová náročnost (je vyžadováno napsání mnoho variant stránky)

Fragment variants – stránka je zkonstruována jako kombinace fragmentů UR (odstavec, obrázek,...). Každý fragment UR může nebo nemusí být vykreslen konkrétnímu uživateli.

4.1.1 Prezentace relevantního obsahu

Prerequisite explanation – před prezentováním určité problematiky jsou poskytnuty veškeré informace nutné k pochopení dané problematiky.

Comparatiove explanation – využívá již získané znalosti uživatele pro lepší vysvětlení prezentované problematiky. Příkladem může být uživatel, který na našich stránkách hledá informace o jazyku PHP a my o tomto uživateli víme, že zná programovací jazyk Java. Můžeme tedy této informace využít ke komparativní prezentaci hledané problematiky, která poskytne danému uživateli pouze informace, které jsou v jazyku PHP odlišné.

Explanation Variants – systém uchovává několik prezentovatelných variant stejné problematiky(lišících se například stupněm odbornosti) a dynamicky provádí výběr nejvhodnější varianty pro konkrétního uživatele.

Sorting – informace jsou prezentovány v pořadí určeném na základě vyhodnocení relevantnosti pro určitého uživatele.

4.1.2 Responsive web design

Responsive web design (RWD) představuje jednu z možností adaptace prezentace obsahu webové aplikace, která využívá stejný HTML kód pro všechny platformy. Všechny zobrazovací zařízení přistupují na stránku ze stejné URL. Tím odpadá nutnost provádět přesměrování. Adaptaci zajišťují pravidla kaskádových stylů(CSS) pomocí media queries. Mohou tak vzniknout robustní CSS soubory nepříznivě se podílející na výkonu aplikace.

Kapitola 5

Nástroje pro runtime generování UR

Aktivně vyvíjené nástroje pro runtime generování UR pro platformu Java EE.

5.1 OpenXava

OpenXava je open source (LGPL), lightweight model driven nástroj určený pro rychlý vývoj aplikace (RAD [15]). Z modelu, který tvoří objekty doménové vrstvy, umožňuje generovat kompletní webovou aplikaci včetně AJAX UR. Poskytuje tak vyšší vrstvu abstrakce, která odstiňuje programátora od kontaktu s doménově specifickými jazyky, jako třeba HTML, Javascript, CSS nebo SQL a ten se tak může lépe zaměřit na problémovou doménu, což je základní myšlenou Domain driven designu (DDD).

OpenXava aplikace rozlišuje artefakty systému na byznys komponenty, kontrolery, moduly, validátry, editory a kalkulátory, nicméně ke generování aplikace je vyžadováno pouze vytvoření byznys komponent [28]. *Byznys komponenty* nepředstavují nic jiného, než dobře známé doménové objekty. Ty v sobě uchovávají informaci o perzistenci, datové struktuře, byznys logice, UR, validaci a podobně [28]. Jejich deklarace je prováděna pomocí anotací. Ke specifikaci generovaného UR slouží prezentační anotace. Business logika může být specifikována pomocí Automated Business Logic (ABL [4]) anotací nebo ji lze popsat v samostatné vrstvě. ABL dokáže vyjádřit až 95% logiky, a to za pomocí pouze několika řádků kódu [1]. Umožňuje tak nejen rychlý vývoj aplikace, ale i snadné provedení změn v business logice. Oba typy anotací jsou natolik samovysvětlující, že je dokáže pochopit i laik [8]. I když je možné vytvořit aplikaci pouze za pomocí informací z doménových objektů a výchozího nastavení, výsledek většinou není vystačující [28]. Akce, které může uživatel provádět, a jejich handlery je možné definovat prostřednictvím *kontrolerů*. Ty spolu s byznys komponentami tvoří *moduly*. Validátory, kalkulátory a editory představují další artefakty systému, které prostřednictvím aplikace v business objektech, přinášejí další možnosti přizpůsobení aplikace na míru. Validátory umožňují specifikovat vlastní validační logiku, kalkulátory jsou určeny ke specifikaci znovupoužitelné business logiky a pomocí editorů můžeme vytvářet za pomocí jazyků pro prezentaci (JSP, Javascript, HTML,..) vlastní komponenty UR. Výsledné zobrazení stránky v prohlížeči představuje zobrazení určitého modulu. Typicky má strukturu master-detail [13], kde master představuje list objektů a detail slouží k jejich editaci. Né

vždy je však struktura master-detail vyhovující, a tak OpenXava nabízí také možnost vytvoření vlastního zobrazení (custom view) opět přímým využitím jazyků pro prezentaci (JSP, Javascript, HTML,..).

Generovaná aplikace je kvalitní. OpenXava automatizuje optimalizaci, znovupoužitelnost, integritu i rozšířitelnost, a to bez znatelného dopadu na výkon [8]. Umožňuje také snadné debugování za použití standardních přístupů.

Nevýhody vyplývající z přístupu one-design-fits-all představují nízká flexibilita, malá kontrola nad vytvářeným UR a nemožnost adaptace UR.

5.2 Apache Isis

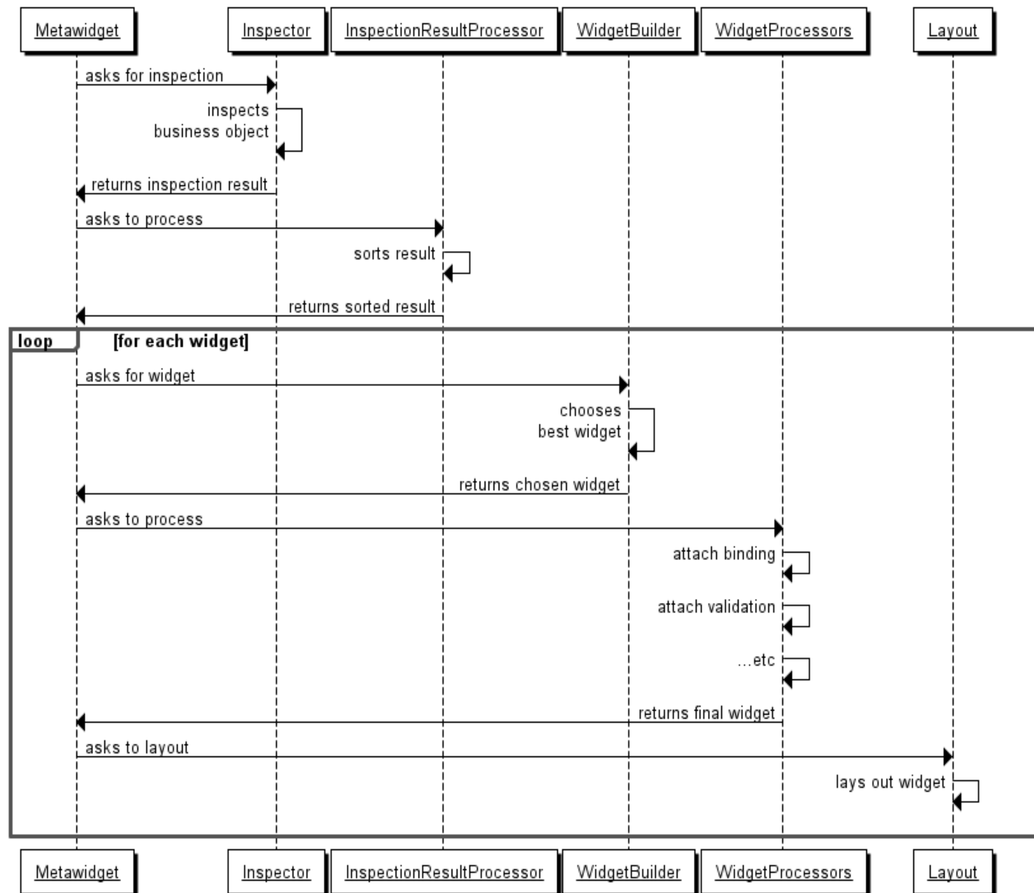
Apache Isis [2] je dalším zástupcem domain driven nástrojů určených pro rychlý vývoj aplikací v Javě. Jedná se o open source Java framework, který má mnoho společného s výše zmíněným nástrojem OpenXava. Jednou z motivací k jeho vytvoření bylo automatické generování UR pro doménové objekty. Záměrem však není generování stejného UR, které by jinak mohlo být vytvořeno manuálně, ale vytvářet generické UI, někdy označované jako Object Oriented User Interface (OOUI) [24]. K vytvoření aplikace stačí vytvořit objekty doménového modelu. Součástí Apache Isis je implementace Restful Objects specifikace [16], díky které mohou být doménové objekty zpřístupněny v JSON formátu prostřednictvím RESTových služeb a HTTP protokolu. Apache Isis používá ke tvorbě UR Apache Wicket framework a layout je reprezentován v JSON formátu.

5.3 Metawidget

Metawidget[14] je open source OIM nástroj, který za využití existujících front-end a back-end technologií dokáže automatizovat vytváření takových widgetů UR, k jejichž vytvoření lze využít informace, získané na základě inspekce existující back-end architektury. Automatické generování je použito pouze ke zlepšení procesu vytváření UR, ne k jeho nahrazení [24]. Díky stanovení tohoto omezení nemusí být Metawidget považován jako konkurence UI frameworků a navíc může generovat stejné UR, jaké by mohlo být vytvořeno manuálně [24].

5.3.1 Architektura

Architektura Metawidgetu byla navržena s cílem splnit předpoklady pro široké praktické využití, které představuje především podpora běžně používaných technologií a praktik pro tvorbu UR a také snadná rozšířitelnost. Architektura se tak skládá z několika menších objektů (Inspectory, WidgetProcessory, Layouty apod.), které jsou z výkonnostních důvodů a z důvodu bezpečného přístupu vláken imutabilní [24]. Imutabilní objekty jsou alokovány pouze jednou, nikdy nejsou kopírovány a nepotřebují být synchronizovány, jelikož nemají problém s konkurenčním přístupem vláken. Jejich problémem je však konfigurace, která musí být prováděna při instanciování. Aby se mohly imutabilní objekty vyvarovat konstruktorům s mnoha parametry, používají se ke konfiguraci konfigurační třídy [24].



Obrázek 5.1: Metawidget pipeline [22]

5.3.2 Inspectory

Inspectory provádějí inspekci back-end architektury a získávají metadata potřebná k sestavení UR. Každý Inspector je zaměřen na inspekci určitého typu metadat. Obecně lze rozdělit Inspectory na Inspectory atributů, Inspectory anotací, XML Inspectory a Composite Inspectory. Inspekci může provádět jediný Inspector nebo více Inspectorů jako součást Composite Inspectoru. Composite Inspector postupně volá jednotlivé Inspectory a výsledek inspekce postupně zpracovává pomocí kombinujícího algoritmu. Výchozí implementace kombinujícího algoritmu je vyhovující pro většinu případů, ale pokud by nám tato implementace nevyhovovala, můžeme si vytvořit vlastní Composite Inspector a vlastní implementaci kombinujícího algoritmu.

5.3.3 InspectionResultProcessory

InspectionResultProcessory slouží ke zpracování výsledku inspekce. Typickým příkladem jejich použití je explicitní seřazení atributů nebo vyhodnocení EL výrazů.

5.3.4 WidgetBuildery

WidgetBuildery se starají o instanciování widgetů. Každý z nich je zodpovědný za vytváření určité knihovny widgetů. Typ widgetu se volí na základě výsledku inspekce. V případě, že chceme zkombinovat několik knihoven widgetů, můžeme tak učinit za pomocí CompositeWidgetBuilderu.

Zpracování započne první definovaný WidgetBuilder. Pokud není zodpovědný za vytvoření daného typu widgetu, vrátí null a zodpovědnost za zpracování se přesune na následující WidgetBuilder. V opačném případě vytvoří novou instanci daného typu widgetu. Zpracování končí poté, co některý z WidgetBuilderů vrátí hodnotu, která není null. Poslední zmíněný WidgetBuilder typicky umožňuje vytváření všech typů widgetů.

WidgetBuildery nezajišťují konfiguraci widgetů (nastavení validace, bindingu, id apod.), ale tuto práci přenechávají WidgetProcessorům.

5.3.5 WidgetProcessory

Každý widget je následně po svém vytvoření zpracován množinou WidgetProcessorů. WidgetProcessory jsou postupně volány a výsledek zpracování jedním Processorem jde na vstup následujícího. Během zpracování může být daný widget nejen pozměněn, například přidáním validátoru, nastavením required atributu nebo data bindingu, ale může být také nahrazen jiným widgetem nebo dokonce hodnotou null, která způsobí zastavení dalšího zpracování widgetu a tím pádem také jeho nevykreslení v layoutu.

5.3.6 Layouty

Díky zmíněným hranicím generování, layout definuje pouze rozmístění generovaných widgetů, ale i tak zahrnuje velký stupeň variability, který musí být generátor UR schopen obsáhnout, aby si zachoval požadovanou flexibilitu [24]. Metawidget nabízí širokou škálu předdefinovaných layoutů a také možnost "jednoduše"vytvořit vlastní.

5.4 AspectFaces

AspectFaces (AF) je open source nástroj implementující techniku Rich entity aspect/audit design (READ). Slouží ke genervání částí UR, které se v případě použití restate-to-extend technik nejvíce potýkají s problémem duplikace informací. Hlavní využití tak najde při automatizaci vytváření formulářových polí a sloupců tabulek. Ke generování UR využívá princip aspektově orientovaného programování, které umožňuje separovat cross-cutting concerny UR a vytvořit generické tagy, které je možné opětovně využívat. Popis použití AspectFaces je součástí kapitoly 7.

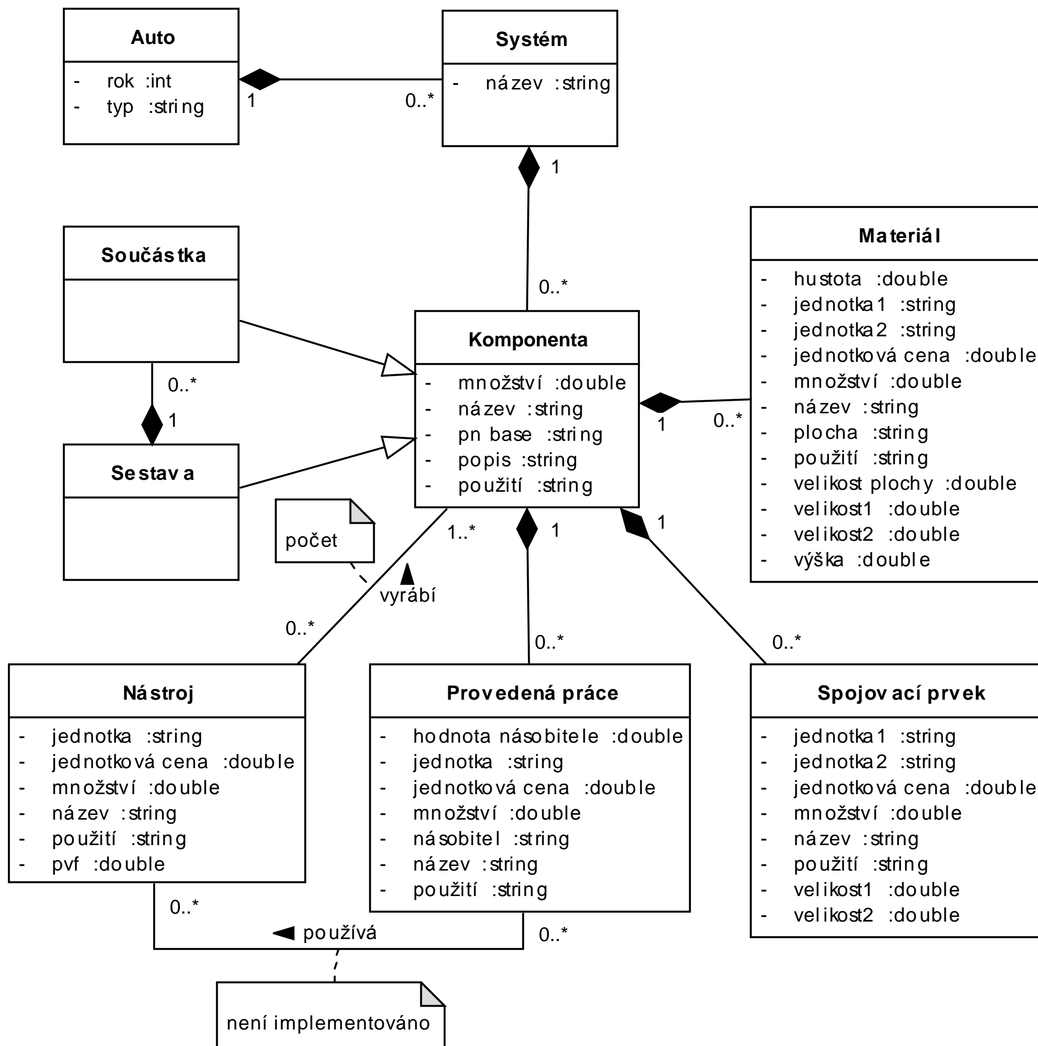
Kapitola 6

Case study: Aplikace Cost Report

Aplikace Cost Report je navržena pro studentský projekt CTU Cartech, který se účastní prestižní mezinárodní inženýrské soutěže Formula Student/SAE. Tým CTU Cartech má za úkol navrhnout a sestavit vůz formulového typu disponující velkým výkonem, dobrými jízdními vlastnostmi i oslnivým designem, jehož náklady na sestrojení budou co nejnižší. Soutěžní týmy jsou hodnoceny z dynamických disciplín, které se týkají výkonu na trati, ale i ze statických disciplín, jejichž součástí je tzv. Cost Report, kde se hodnotí správná kalkulace výrobní ceny vozu. Týmy musí vypracovat dokument ve formátu .xls nebo .xlsx, který obsahuje podrobný přehled o všech atributech tvořících celkovou cenu vozu a zveřejnit jej před konáním soutěže. Disciplína Cost Report tvoří 10% celkového hodnocení, a proto ji není vhodné zanedbat.

Původním záměrem bylo vytvářet dokument manuálně, avšak při rozsáhlém množství součástek a složitosti výpočtu celkové ceny bylo snadné dopustit se chyb ve výpočtu. Kromě toho byla se strukturou dokumentu obeznámena pouze zodpovědná osoba za tuto činnost a ostatní se tak nemohli podílet na vytváření či úpravě dokumentu. Jelikož byl dokument kvůli obtížné správě vytvářen až těsně před závodem, nemohl být v průběhu vytváření využit jako zdroj informací například pro monitoring průběžné ceny, která by mohla upozornit na možné úspory. Vznikl tak záměr proces vytváření tohoto dokumentu automatizovat a správu součástek provádět postupně prostřednictvím přehledného UR webové aplikace.

6.1 Doménový model



Obrázek 6.1: Doménový model

Auto

Auto reprezentuje konkrétní vozidlo formulového typu pro daný rok.

Atribut	Povinný	Popis
rok	ano	Rok, ve kterém je auto navrhováno.
typ	ano	Typ auta odlišuje auta se stejným rokem.

Tabulka 6.1: Výčet atributů doménového objektu Auto

Validační požadavky: Dvojice rok a typ budou tvořit unikátní záznam auta.

Systém

Systém reprezentuje část auta složenou ze součástek a sestav.

Atribut	Povinný	Popis
název	ano	Název systému.

Tabulka 6.2: Výčet atributů doménového objektu Systém

Validační požadavky: Název systému bude v kontextu auta unikátní.

Komponenta

Komponenta reprezentuje sestavu nebo konkrétní součástku sestavy(nejmenší část auta).

Atribut	Povinný	Popis
název	ano	Název systému.
popis	ano	Popis komponenty.
použití	ne	Popis použití komponenty.
množství	ano	Počet kusů komponent, použitých v daném systému či sestavě.
P/N base	ano	Identifikační označení, používané ve strojírenské praxi, unikátní pro daný systém.

Tabulka 6.3: Výčet atributů doménového objektu Komponenta

Validační požadavky: Identifikátor P/N base bude unikátní pro komponentu v kontextu systému a bude tvořen pěti znaky.

Sestava

Sestava představuje část systému, ve které jsou evidovány součástky, z nichž je složena. V případě, že sestava nemá přiřazeny žádné součástky, ze kterých by se měla skládat, je v tomto případě brána jako součástka systému - tedy dále nedělitelný celek.

Součástka

Součástka představuje část systému, u které se již nerozlišují součásti, z nichž je složena, ale považuje se jako jeden celek.

Spojovací prvek

Spojovací prvek použitý ke konstrukci komponenty.

Atribut	Povinný	Popis
název	ano	Název spojovacího prvku dle cenových tabulek.
množství	ano	Počet kusů komponent v daném systému či sestavě.
použití	ne	Popis použití spojovacího prvku.
jednotková cena	ano	Cena za jeden kus spojovacího prvku po parametrizaci.
jednotka1	ne	Slouží k definování velikosti1 spojovacího prvku.
velikost1	ne	Parametr spojovacího prvku, který udává jeho cenu.
jednotka2	ne	Slouží k definování velikosti2 spojovacího prvku.
velikost2	ne	Parametr spojovacího prvku, který udává jeho cenu.

Tabulka 6.4: Výčet atributů doménového objektu Spojovací prvek

Materiál

Materiál použitý ke konstrukci komponenty.

Atribut	Povinný	Popis
název	ano	Název spojovacího prvku dle cenových tabulek.
množství	ano	Množství materiálu.
použití	ne	Popis použití materiálu.
jednotková cena	ano	Cena za jednotku.
jednotka1	ne	Slouží k definování velikosti1 materiálu.
velikost1	ne	Parametr materiálu, který udává jeho cenu.
jednotka2	ne	Slouží k definování velikosti2 materiálu.
velikost2	ne	Parametr materiálu, který udává jeho cenu.
plocha	ne	Pojmenování plochy, volitelná textová hodnota.
velikost plochy	ne	Velikost plochy v jednotkách m2.
výška	ne	Výška prostorového objektu.
hustota	ne	Hustota materiálu v jednotkách kg/m3.

Tabulka 6.5: Výčet atributů doménového objektu Materiál

Provedená práce

Provedená práce potřebná ke konstrukci komponenty.

Atribut	Povinný	Popis
název	ano	Název provedené práce.
množství	ano	Množství jednotek.
popis	ne	Popis provedené práce.
jednotková cena	ano	Cena za jednotku.
jednotka	ano	Jednotka pro jednotkovou cenu.
násobitel	ne	Název násobitele procesu.
hodnota násobitele	ne	Číselná hodnota dle cenových tabulek.

Tabulka 6.6: Výčet atributů doménového objektu Provedená práce

Nástroj

Nastroj použitý ke konstrukci komponenty.

Atribut	Povinný	Popis
název	ano	Název provedené práce.
množství	ano	Množství jednotek.
použití	ne	Popis použití nástroje.
jednotková cena	ano	Cena za jednotku.
jednotka	ano	Jednotka pro jednotkovou cenu.
product volume factor	ano	Schopnost nástroje produkovat výrobky.

Tabulka 6.7: Výčet atributů doménového objektu Nástroj

6.2 Funkční požadavky

Funkční požadavky vyjadřují, co by měl systém dělat. Každý požadavek je specifikován tak, aby jej bylo možné otestovat, má přiřazenu prioritu a unikátní ID.

FReq1 - Základní operace s daty

Systém bude umožňovat uživateli provádět základní operace s daty (CRUD). *Vysoká priorita.*

FReq2 - Podrobné informace o ceně

Systém bude poskytovat uživateli informaci o celkové ceně vozu i jeho jednotlivých částí. *Vysoká priorita.*

FReq3 - Potvrzení důležitých akcí

Systém bude vyžadovat potvrzení všech akcí, které provádí smazání dat z databáze, prostřednictvím potvrzovacího dialogu. *Vysoká priorita.*

FReq4 - Validace uživatelského vstupu

Systém bude validovat uživatelský vstup jak na straně klienta, tak i na straně serveru. *Vysoká priorita.*

FReq5 - Zobrazení/skrytí sloupců tabulky

Systém bude umožňovat zobrazit nebo skrýt jednotlivé sloupce tabulky, které obsahují data. Sloupce, které data neobsahují (sloupec s akcemi), nebude možné skrýt. V případě, že uživateli nebudou prezentována všechna data (viz NFReq3 v podkapitole 6.3), může uživatel využít této funkce k jejich zobrazení. *Střední priorita.*

FReq6 - Řazení záznamů v tabulkách

Systém bude umožňovat řadit záznamy v tabulkách podle jednotlivých sloupců jak vzestupně, tak i sestupně. *Střední priorita.*

FReq7 - Aktivace/deaktivace nápovědy u formulářových polí

Systém bude umožňovat uživateli aktivaci a deaktivaci nápovědy u formulářových polí. *Nízká priorita.*

FReq8 - Generování Cost Reportu

Systém bude umožňovat generovat soubor s údaji o ceně konkrétního auta a jeho součástech. Výsledný soubor bude ve formátu .xls nebo .xlsx a svou strukturou i obsahem bude odpovídat standardu pro Cost Report. Implementace tohoto požadavku není součástí této bakalářské práce. *Vysoká priorita.*

6.3 Nefunkční požadavky

Nefunkční požadavky specifikují, jak má systém provádět operace. Zahrnují například požadavky na výkon, bezpečnost, rozšířitelnost, ale i spravovatelnost či adaptivitu.

NFReq1 - Relační databáze

Data budou uchovávána v relační databázi.

NFReq2 - Zobrazení nápovědy

Systém bude zobrazovat nápovědu pouze nad aktivním formulářovým polem.

NFReq3 - Zobrazení pouze relevantních informací

U tabulek s větším množstvím sloupců bude systém na základě priority automaticky zobrazovat pouze takový počet sloupců, který bude možné prezentovat na obrazovce konkrétního zařízení bez scrollování.

NFReq4 - Proměnlivý layout

Layout formulářů se bude měnit v závislosti na velikosti obrazovky.

NFReq5 - Označení povinných formulářových polí

Povinná formulářová pole budou označena znakem "*".

NFReq6 - Podpora vícejazyčnosti

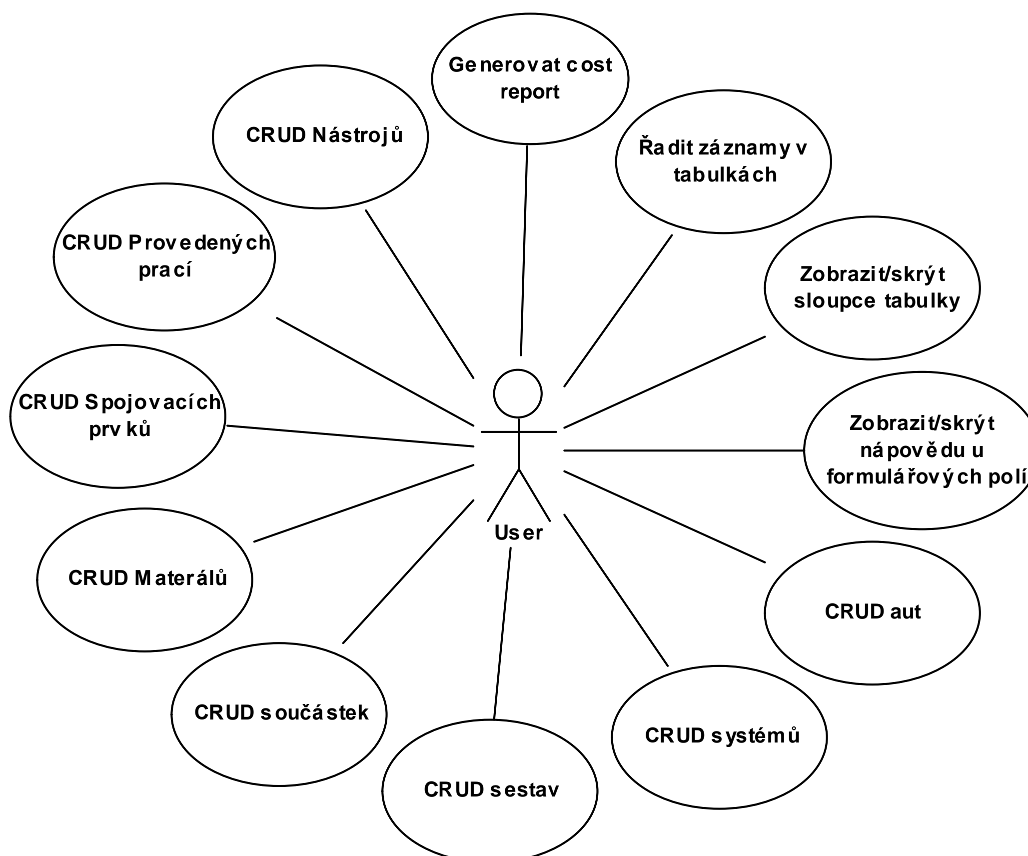
UR bude v angličtině, avšak aplikace bude připravena na možné rozšíření o další jazyky.

NFReq7 - Číselné hodnoty reprezentované jako double

Veškeré číselné hodnoty s výjimkou roku budou reprezentovány jako double.

6.4 Případy použití

Případy použití popisují sekvenci interakcí mezi systémem a aktéry nezbytných k dosažení určitého cíle. V systému je jediná skupina aktérů, označovaných jako uživatelé. Úkony, které mohou v aplikaci provádět, jsou znázorněny na obrázku 6.2. Podrobný popis jednotlivých případů užití je uveden v příloze A.



Obrázek 6.2: Diagram případů použití

6.5 Implemetace

K implementaci UR aplikace Cost Report byl použit framework JavaServer Faces (JSF) verze 2.2. Servisní vrstva aplikace, DAO (Data access object) vrstva a JSF backing beany jsou implementovány jako Spring Beany pomocí Spring frameworku. K persistenci dat je použito Java persistence API, jehož implmentaci zajišťuje Hibernate.

V průběhu vývoje UR jsem narazil na problém nekompatibility nástroje AspectFaces se zmíněnou verzí JSF. Tento problém jsem proto nahlásil a následně se také podílel na jeho nápravě. Zároveň jsem navrhl možné vylepšení AspectFaces nástroje zahrnující přidání indexu zpracovávaného atributu do AF kontextu, který byl v aplikaci Cost Report využit k identifikaci prvního zpracovávaného atributu, jemuž mohl být nastaven atribut `autofocus`. Postup generování UR aplikace Cost Report je popsán v samostatné kapitole 7. Z důvodu zestručnění bakalářské práce je v následující podkapitole popsána pouze vybraná část implementace funkcionality aplikace zahrnující vytváření responsivních tabulek.

6.5.1 Responsivní tabulky

Související požadavky: FReq5, NFReq3

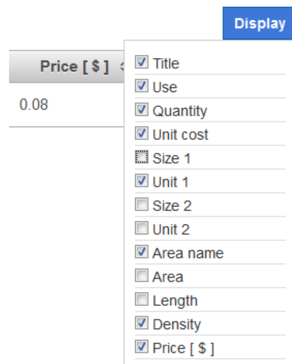
Postup řešení

Zobrazení tabulek s velkým množstvím sloupců na malých obrazovkách je častým problémem, který musí designéři aplikace řešit. Díky menšímu místu pro prezentaci dat nemohou být v tabulce zobrazena všechna data, kterými aplikace disponuje.

Nejjednodušším řešením je použít horizontální scroll bar. Uživatel však v průběhu scrollování může ztratit kontext prezentovaných dat. Alternativní variantu představuje zobrazení omezeného počtu dat určeného šířkou obrazovky zařízení s možností explicitního zobrazení zbylých dat uživatelem. K použití scroll baru v tomto případě dojde až ve chvíli, kdy uživatel provede zobrazení většího počtu sloupců, než je možné přehledně prezentovat. Tento přístup, představující implementaci techniky adaptace obsahu zvanou *responsive design* (viz podkapitola 4.1.2), je použit pro zobrazení vícesloupcových tabulek aplikace Cost Report.

Sloupce tabulky zastupují atributy entity, která slouží k uchovávání dat prezentovaných v tabulce. Omezení zobrazených sloupců je prováděno na základě přiřazení priority zobrazení. Priorita zobrazení je reprezentována jako výčtový typ, jehož každá položka má specifikovánu CSS třídu, která realizuje patřičné chování priority zobrazení. Responsivní tabulky jsou implementovány metodou *progresive enhancement*, kde k aplikaci prioritizovaného zobrazování dojde pouze v případě, že je v prohlížeči povolen Javascript a uživatel tak může provést explicitní zobrazení skrytých sloupců. V opačné případě jsou zobrazeny všechny sloupce tabulky.

V případě generování UR aplikace pomocí AspectFaces nebo Metawidgetu je možné specifikovat prioritu zobrazení jednotlivých atributů pomocí vlastní anotace a využít inspekci metadat, prováděnou v průběhu generování UR, k jejímu zpracování.



Obrázek 6.3: Explicitní zobrazení sloupců uživatelem

6.6 Testování

6.6.1 Unit testy

Unit testy slouží k validaci funkcionality samostatných jednotek aplikace v izolaci od zbytku aplikace. V objektovém jazyku představuje samostatnou jednotku třída a její metody. Izolace testované třídy lze docílit nahrazením objektů za tzv. *mock objekty*, které simulují chování izolované třídy. Díky izolaci jednotek a testování pouze malých částí funkcionality je provádění unit testů velmi rychlé. Testování aplikace proto začíná právě provedením unit testů, které dokáží odhalit základní chyby ve funkcionalitě a až poté, v případě jejich úspěšného provedení, následují rozsáhlejší testy.

6.6.2 Integrační testy

Integrační testy validují vzájemnou integraci několika komponent části aplikace. Testy probíhají v prostředí, ve kterém bude nasazena finální aplikace.

6.6.3 Selenium testy

Selenium testy umožňují simulovat interakci uživatele s aplikací v prostředí webového prohlížeče a validovat její výsledek. Automatizují zdlouhavé manuální front-endové testy, do značné míry odstraňují z testování lidský faktor a zaručují pokaždé stejný průběh testů. Nepřináší však jen samá pozitiva. Čas potřebný k vytvoření testů není zrovna zanedbatelný a ne vždy se nám tyto testy stihnou vyplatit. Také je nutné počítat s jejich velkou náchylností na změny v aplikaci. Navíc eliminace lidského faktoru z testování nemusí být vždy tou správnou cestou [6]. V případě manuálního testování se můžeme vžít do role uživatele aplikace a přemýšlet nad možnými vylepšeními z hlediska použitelnosti UR. Výsledné UR tak může být v případě manuálního testování použitelnější [6]. Po zvážení výše zmíněných kladů a záporů jsem se rozhodl využít k testování selenium testy pouze částečně a kombinovat je s manuálními testy.

Kapitola 7

Generování UR

V této kapitole je popsán postup použití nástrojů AspectFaces a Metawidget k runtime generování UR aplikace Cost Report. Předmětem generování jsou části UR, které mají největší závislost na modelu aplikace a při využití restate-to-extend technik vytváření UR vyžadují duplikaci informací z vrstvy doménových objektů. Závěr této kapitoly je věnován srovnání výsledného kódu UR při použití zmíněných nástrojů s kódem UR vytvořeným bez použití automatizace za pomoci frameworku JavaServer Faces.

7.1 Rich entity

Entity reprezentují datový model aplikace, který v sobě zahrnuje metadata sloužící k jeho mapování na tabulky relační databáze. Kromě informací o persistenci, mohou entity obsahovat také několik dalších metadat specifikujících například validační omezení či informace potřebné k jejich zobrazení v UR. Takto rozšířené entity jsou označovány jako rich entity. Příklad takovéto entity je uveden níže.

```
@Entity
@Table(uniqueConstraints =
        @UniqueConstraint(columnNames = {"year", "type"}))
public class Car extends AbstractBusinessObject {

   private Integer year;
   private String type;

   @UiPlaceholder("#{text['type']}")
   @UiOrder(2)
   @NotNull
   @Size(max = 2)
   public String getType() {
      return type;
   }

   public void setType(String type) {
      this.type = type;
   }
```

```
    @UiPlaceholder("#{text['year']}")
    @UiOrder(1)
    @NotNull
    public Integer getYear() {
        return year;
    }

    public void setYear(Integer year) {
        this.year = year;
    }
}
```

Ukázka kódu 7.1: Příklad rich entity

Validační omezení blíže specifikující uchovávanou hodnotu jednotlivých atributů jsou vyjádřeny pomocí Java API pro validaci JavaBean (Bean validation) [7]. *Bean validation* představuje rozšíření JavaBean objektového modelu o metadata pro validaci. Standardizuje definici, deklaraci a validaci omezení, která blíže specifikují formát uchovávané hodnoty pro platformu Java. Tato specifikace se snaží zkombinovat nejlepší vlastnosti a funkce dostupných validačních frameworků a promítnou je do API, které by bylo možné využívat nezávisle na konkrétní implementaci.

Rich entita Car (viz ukázka kódu 7.1) obsahuje anotace `@Size` a `@NotNull` specifikující validační omezení na get metodách(označovaných také jako gettry) atributů `type` a `year`. Anotace `@Size(max=2)` omezuje množinu hodnot, které je možné uchovávat v atributu `type` typu String pouze na textové řetězce s maximálním počtem znaků rovným 2. Anotace `@NotNull` specifikuje požadavek na povinné vyplnění hodnoty atributů `year` a `type`.

Metadata potřebná ke zobrazení entit, jako například informaci o pořadí atributů, kterou Java nezachovává, je možné specifikovat pomocí anotací příslušného nástroje, který se stará o generování UR nebo pomocí vlastních anotací. Vlastní anotace lze v Javě vytvářet pomocí speciální `@interface` pseudo-anotace, která označí danou komponentu jazyka jako anotaci.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface UiPlaceholder {
    public String value() default "";
}
```

Ukázka kódu 7.2: Vlastní anotace

Styl zápisu je velice podobný interfacům Javy. Parametry anotace jsou definovány jako jednoduché metody neobsahující parametry ani tělo. Jejich návratový typ je jakýkoliv primitivní typ, String, Class, enum, anotace nebo pole kteréhokoliv ze zmíněných typů. Každý parametr může mít deklarovánu výchozí hodnotu. Elementy jazyka, na které může být anotace aplikována, jsou specifikovány pomocí anotace `@Target`. Způsob zpracování vlastní anotace určuje anotace `@Retention`.

Vlastní anotace `@UiPlaceholder` slouží ke specifikaci hodnoty HTML5 `placeholder` atributu formulářového vstupního pole a má tedy pouze jediný atribut `value`. Tato anotace může

být aplikována na metody nebo atributy třídy a díky `@Retention(RetentionPolicy.RUNTIME)` bude dostupná pro reflexi v runtimu. U výše zmíněné rich entity Car (viz ukázka kódu 7.1) je tato anotace aplikována na gettry atributů `year` a `type`. Jelikož má anotace `@UiPlaceholder` pouze jediný atribut `value`, může být zapsána ve zkráceném tvaru.

```
@UiPlaceholder("placeholder") = @UiPlaceholder(value = "placeholder")
```

Ukázka kódu 7.3: Zkrácený zápis anotace

Pokud vytvoříme vlastní anotaci, musíme se také postarat o její zpracování příslušným nástrojem pro generování UR. Tato problematika bude popsána v následující podkapitole.

7.2 Inspekce metadat

7.2.1 Inspekce v AspectFaces

AspectFaces odstiňuje programátora od problematiky inspekce metadat. Její implementace, o kterou se stará třída JavaInspector, je nekonfigurovatelnou součástí frameworku. Úkolem programátora je tak pouze uvedení požadovaných AnnotationDescriptorů v konfiguračním souboru aspectfaces-config.xml (viz podkapitola 7.7.2) a v případě vlastních anotací také jejich vytvoření.

AnnotationDescriptory zajišťují identifikaci anotace, která má být zaregistrována k inspekci a určují způsob jejího zpracování v průběhu inspekce. Všechny Descriptory proto musí implementovat interface AnnotationDescriptor a jeho jedinou metodu `getAnnotationName`. Tato metoda je volána před samotnou inspekcí a slouží k registraci anotace ke zpracování při inspekci. Vrací úplný název anotace (tedy včetně názvu package), pro kterou je Descriptor určen.

Způsob zpracování anotace se liší v závislosti na typu anotace. Proto AspectFaces rozlišuje čtyři typy Descriptorů. Jejich výčet spolu s přídady použití popisuje tabulka 7.1. Typ Descriptoru určuje interface pro zpracování anotace, který daný Descriptor musí implementovat. Název interfacu je shodný s názvem typu Descriptoru.

Typ Descriptoru	**Použití**
VariableJoinPoint	Přidání nových proměnných do AF kontextu.
OrderJoinPoint	Zpracování anotací určujících pořadí elementů.
EvaluableJoinPoint	Zpracování anotací, jejichž hodnota má být předána k vyhodnocení *evaluatoru* [1]. Výsledkem vyhodnocení je boolean hodnota. V případě, že je rovna false, anotovaný atribut je ignorován a nebude součástí vygenerovaného UR.
SecurityJoinPoint	Zpracování anotací určujících podmíněné zobrazení na základě uživatelských rolí.

Tabulka 7.1: Typy AnnotationDescriptorů

```
public class UiPlaceholderAnnotationDescriptor implements AnnotationDescriptor,
    VariableJoinPoint {

   @Override
   public String getAnnotationName() {
      return "cz.cvut.cartech.aspectfaces.annotations.UiPlaceholder";
   }

   @Override
   public List<Variable> getVariables(AnnotationProvider annotationProvider) {
      List<Variable> variables = new ArrayList<Variable>();
      variables.add(new Variable("placeholder", annotationProvider.getValue("value"
          )));
      return variables;
   }
}
```

Ukázka kódu 7.4: Příklad AnnotationDescriptoru

Ke zpracování výše zmíněné anotace `@UiPlaceholder` (viz ukázka kódu 7.2), která přidává informaci o hodnotě `placeholder` atributu, tedy bude použit VariableJoinPoint Descriptor. Příklad jeho implementace je uveden v ukázce kódu 7.4.

K přidání proměnných do AF kontextu je třeba specifikovat jejich název a hodnotu. Název proměnné představuje klíč, pomocí kterého lze následně přistupovat k hodnotě proměnné uchovávané v AF kontextu. Hodnota proměnné může být tvořena hodnotou jednoho z atributů anotace, nebo v případě bezparametrické anotace, hodnotou specifikovanou programátorem. K získání hodnoty atributů zpracovávané anotace slouží AnnotationProvider předaný v parametru metody. Ten obsahuje metodu `getValue`, která vrací hodnotu atributu anotace, jehož název je specifikován v parametru této metody.

7.2.2 Inspekce v Metawidgetu

Metawidget využívá k získání a zpracování metadat tzv. *Inspectory* (viz podkapitola 5.3.2). Ty se na rozdíl od AspectFaces AnnotationDescriptorů starají nejen zpracování anotací, ale jsou zodpovědné za celou problematiku inspekce určitého typu metadat. Díky tomu má programátor možnost vytvářet vlastní Inspectory jakéhokoliv typu metadat, od anotací až po XML konfigurační soubory jako je hibernate.xml.config. V případě specifikace více typů Inspectorů jsou pak tyto Inspectory sloučeny a zpracovávány jako jeden CompositeInspector.

Ke zpracování anotací slouží podskupina Inspectorů zvaná *AnnotationInspectors*, tedy Inspectory anotací. Inspectory anotací nejsou typicky vyvářeny pro jednotlivé anotace, ale pro celou skupinu anotací (například JPA anotace). Jejich společným předkem je BaseObjectIspector, který poskytuje vzorovou implementaci inspekce. Úkolem jednotlivých Inspectorů anotací je pouze přepsat metodu `inspectProperty` zodpovědnou za inspekci vlastností třídy. Vlastnostmi třídy jsou míněny atributy třídy nebo její gettry/settry v závislosti na zvoleném stylu zápisu anotací. Tato metoda je volána pro každou vlastnost, a proto je nutné ověřit, zda je daná anotace na aktuální vlastnosti dostupná.

Styl zápisu anotací se může lišit, avšak v kontextu aplikace by měl být jednotný. Příklad dvou různých stylů zápisu může být umístění anotací nad private atributy nebo nad jejich gettry/settry. Ve výchozím nastavení Metawidget očekává anotace nad gettry/settry, avšak specifikace vlastního stylu je umožněna v konfiguraci Inspectoru. Rozdílné styly zápisu sdružuje společný interface Property, který umožňuje Inspectorům zpracovávat rozdílné styly zápisu jednotným způsobem.

Možná implementace Inspectoru pro vlastní anotaci `@UiPlacehoder`(viz ukázka kódu 7.2) je uvedena níže.

```
public class UiPlaceholderInspector extends BaseObjectInspector {

   public CustomAnnotationsInspector() {
      this(new BaseObjectInspectorConfig());
   }

   public CustomAnnotationsInspector(BaseObjectInspectorConfig config) {
      super(config);
   }

   @Override
   protected Map<String, String> inspectProperty(Property property)
      throws Exception {

      Map<String, String> attributes = CollectionUtils.newHashMap();
      UiPlaceholder placeholder = property.getAnnotation(UiPlaceholder.class);
      if (placeholder != null) {
         attributes.put("placeholder", placeholder.value());
      }
      return attributes;
   }
}
```

Ukázka kódu 7.5: Příklad Metawidget Inspectoru

7.3 Zpracování metadat

V Metawidget lifecyclu se provádí zpracování metadat získaných při inspekci pomocí tzv. *InspectionResultProcessorů*. InspectionResultProcessory se liší od Inspectorů tím, že obsahují závislost na UR, kterou mohou využít například pro vyhodnocení EL výrazů. Inspectory tuto závislost nemají z důvodu možného použití i pro aplikace bez UR (například s REST rozhraním). Příkladem typického zpracování metadat je seřazení atributů nebo vyhodnocení EL výrazu. V AspectFaces je naproti tomu veškeré zpracování metadat neoddělitelnou součástí frameworku.

7.4 Mapování atributů na widgety

7.4.1 Mapování v Metawidgetu

V Metawidgetu jsou mapovací pravidla (zapsaná v podobě if else bloků) součástí WidgetBuilderů (viz podkapitola 5.3.4). Z pohledu programátora je tak orientace v mapovacích pravidlech značně obtížná, jelikož nejsou součástí jednoho souboru. Navíc k jejich editaci je třeba vytvořit vlastní implementaci WidgetBuilderu. Faktem ale je, že tato pravidla jsou ve většině případech vyhovující. V ukázce kódu 7.6 je uvedena část kódu HTMLWidgetBuilderu[2], zodpovědná za mapování atributů typu Number a String na widgety. Z této ukázky je patrný další společný problém Metawidgetu, který představuje nesnadná specifikace výchozích hodnot atributů vytvářených widgetů.

```
else if ( Number.class.isAssignableFrom( clazz ) ) {
   component = application.createComponent( HtmlInputText.COMPONENT_TYPE );
} else if ( String.class.equals( clazz ) ) {
   if ( TRUE.equals( attributes.get( MASKED ) ) ) {
      component = application.createComponent( HtmlInputSecret.COMPONENT_TYPE );
   } else if ( TRUE.equals( attributes.get( LARGE ) ) ) {
      component = application.createComponent( HtmlInputTextarea.COMPONENT_TYPE );

      // XHTML requires the 'cols' and 'rows' attributes be set,
      // even though most people override them with CSS widths and
      // heights. The default is generally 20 columns by 2 rows.

      ( (HtmlInputTextarea) component ).setCols( 20 );
      ( (HtmlInputTextarea) component ).setRows( 2 );
   } else {
      component = application.createComponent( HtmlInputText.COMPONENT_TYPE );
   }
}
```

Ukázka kódu 7.6: Mapování v Metawidgetu

7.4.2 Mapování v AspectFaces

AspectFaces odděluje mapování do samostatného XML konfiguračního souboru označovaného jako *AspectFaces profile*, jehož obsah tvoří seznam prezentačních pravidel (viz podkapitola 3.2.3.2) a volitelně také seznam atributů, které nají být v kontextu daného profilu ignorovány. Jednotlivá prezentační pravidla jsou ohraničena tagem `mapping`. Povinnou částí mapovacího pravidla je definice typů pro které je určeno a také výchozího tagu[3], který se aplikuje v případě, že není stanoveno jinak. Volitelnou částí mapovacího pravidla je specifikace podmínek, při jejichž splnění dojde k použití jiného tagu, než toho, který je specifikován jako výchozí. Podmínky jsou vyjádřeny jako EL výraz, který je vyhodnocen na true nebo false.

[2]HTMLWidgetBuilder je implementace WidgetBuilderu pro JavaServer Faces prostředí, která je součástí Metawidgetu.

[3]Alternativní označení widgetu, používané v AspectFaces.

AspectFaces poskytuje vlastní expression language, kde začátek a konec výrazu je ve výchozím nastavení označen znakem `$`, ale toto nastavení je možné změnit v aspectfaces.properties. Uvnitř tohoto výrazu je možné přistupovat k proměnných z AF kontextu prostřednictvím jejich názvu. Příklad mapovacího pravidla pro atribut typu String, spolu s příkladem definice ignorovaného atributu, je uveden na ukázce kódu 7.7.

```
<mapping>
   <type>String</type>
   <default tag="html/text.xhtml" maxLength="255" size="30" required="false"/>
   <condition expression="${not empty email and email == true}" tag="html/email.
       xhtml"/>
   <condition expression="${not empty password and password == true}" tag="html/
       password.xhtml"/>
</mapping>

<ignore-fields>
   <name>id</name>
</ignore-fields>
```

Ukázka kódu 7.7: Mapování v AspectFaces

Součástí konfigurace AspectFaces může být jeden, ale i více takovýchto profilů, které mohou být využity pro specifikaci odlišných mapovacích pravidel pro jiný typ zobrazení. Příkladem může být jeden profil obsahující mapování na formulářová pole a další na sloupce tabulky.

7.5 Vytváření widgetů

7.5.1 Vytváření widgetů v Metawidgetu

V Metawidgetu je vytváření widgetů rozděleno na dvě fáze. V první fázi, kterou má na starosti tzv. *WidgetBuilder* (viz podkapitola 5.3.4), dojde k vytvoření samotného widgetu, který má přiřazeny jen nejnutnější atributy. Nejnutnějšími atributy je myšleno například id, ale i CSS třídy, jejichž deklarace je součástí konfigurace Metawidgetu. Toto řešení ovšem snižuje přehled o vytvářeném UR a představuje také omezení flexibility Metawidgetu. V případě, že WidgetBuilder není určen k vytváření zpracovávaného typu atributu, vrátí null a zodpovědnost za zpracování se přesune na následující WidgetBuilder. Většina WidgetBuilderů proto začíná podobnou podmínkou jako je na ukázce kódu 7.8, kde je uveden začátek metody `buildWidget` PrimefacesWidgetBuilderu.

```
@Override
public UIComponent buildWidget( String elementName, Map<String, String> attributes,
     UIMetawidget metawidget ) {

   // Not for PrimeFaces?
   if ( TRUE.equals( attributes.get( HIDDEN ) ) ) {
      return null;
   }
```

```
    {...}
}
```

Ukázka kódu 7.8: Metoda buildWidget

WidgetBuildery jsou typicky vytvářeny pro skupinu widgetů [4], proto poté většinou následují mapovací pravidla, na základě kterých volí WidgetBuilder vhodný typ widgetu pro reprezentaci daného atributu (viz podkapitola 7.4.1).

Ve druhé fázi je widget zpracováván sérií tzv. WidgetProcessorů (viz podkapitola 5.3.5), které widgetu doplní veškeré požadované atributy. Příklad WidgetProcessoru je uveden na ukázce kódu 7.9.

```
public class PassThroughAttributesProcessor implements WidgetProcessor<javax.faces.
    component.UIComponent, UIMetawidget> {

     @Override
     public UIComponent processWidget(UIComponent widget, String elementName, Map<
         String, String> attributes, UIMetawidget m) {
          Map passthrough = widget.getPassThroughAttributes(true);

          //set placeholder for HtmlInputText component
          Object placeholder = attributes.get(CustomInspectorResultConstants.
              PLACEHOLDER);
          if (widget instanceof HtmlInputText && placeholder != null) {
               passthrough.put("placeholder", placeholder);
          }

          // set other attributes

          return widget;
     }
}
```

Ukázka kódu 7.9: Metawidget WidgetProcessor

Vhodné je také poznamenat, že metoda `buildWidget` WidgetBuideru (viz ukázka kódu 7.8) je zodpovědná pouze za vytváření daného widgetu (návratový typ UiComponent) a metoda `processWidget` WidgetProcessoru (viz ukázka kódu 7.9) za jeho zpracování. V případě požadavku na zobrazení popisu (atribut `label`) formulářového pole je tak vytvoření tohoto popisu možné až v příslušném Layoutu.

7.5.2 Vytváření widgetů v AspectFaces

AspectFaces nepracuje s widgety jako s objekty, ale pouze jako se Stringy. Programátor vytváří generické šablony widgetů pomocí DSL jazyka, který je zvyklý používat k vytváření UR. K proměnným z AF kontextu je možné přistupovat stejně jako v podmínkách mapovacích pravidel, tedy pomocí EL výrazu. AspectFaces weaver se poté postará o jejich integraci

[4] Podobně jako Inspectory pro skupinu anotací.

do šablony a vznikne tak konkrétní widget. Výhodou tohoto přístupu oproti Metawidgetu je jednoduchost a také přehlednost. Příklad generické šablony textového formulářového pole je uveden na ukázce kódu 7.10. Popis formulářového pole (atribut `label`) je součástí této šablony, tedy na místě, kde by ho programátor očekával spíše než v definici layoutu, jako je tomu u Metawidgetu. V šabloně je také využita proměnná `placeholder`, jejíž zpracování bylo popsáno v předchozích podkapitolách.

```
<ui:param name="id" value="#{prefix}$fieldName$"/>
<ui:param name="label" value="#{text['$fieldName$']}"/>
<ui:param name="value" value="#{$ClassName$.$fieldName$}"/>
<ui:param name="rendered" value="#{empty render$FieldName$ ? 'true' :
    render$FieldName$}"/>
<ui:param name="required" value="#{empty required$FieldName$ ? '$required$' :
    required$FieldName$}"/>
<ui:param name="maxlength" value="$maxLength$"/>
<ui:param name="minlength" value="$minLength$"/>
<ui:param name="size" value="$size$"/>
<ui:param name="placeholder" value="$placeholder$"/>

<ui:fragment rendered="#{empty edit ? false : edit and rendered}">
   <div class="inputWidget">
      <label jsf:id="#{id}Label" jsf:value="#{label}:" for="#{id}">
         <ui:fragment rendered="#{required or forceRequired}">
            <span class="required">*</span>
         </ui:fragment>
      </label>

      <input type="text" jsf:value="#{value}" jsf:id="#{id}" maxlength="#{maxlength
          }" title="#{label}" jsf:rendered="#{rendered}" jsf:required="#{required}"
          >
         <f:validateLength minimum="#{minlength}" maximum="#{maxlength}" />
         <c:if test="#{not empty placeholder}">
            <f:passThroughAttribute name="placeholder" value="#{placeholder}" />
         </c:if>
      </input>
   </div>
</ui:fragment>
```

Ukázka kódu 7.10: Šablona widgetu v AspectFaces

7.6 Layout

7.6.1 Layout v Metawidgetu

Layout je v Metawidgetu reprezentován jako objekt. V případě použití JSF jakožto front-end frameworku, využívá Metawidget k vytváření layoutu koncept JSF rendererů, kde jednotlivé Layouty jsou potomky třídy Renderer. Widgety, které byly do této chvíle reprezentovány jako objekty, jsou v této fázi transformovány do formátu HTML. V závislosti na typu Layoutu

se také může přidat pro jednotlivé komponenty popis (`label`) či jiné související elementy. S problematikou implementace rendererů se však běžně programátor při vytváření UR nesetkává, a tak pokud chce vytvořit vlastní Layout, musí si nejprve tento koncept nastudovat. Příklad jednoduchého Layoutu, který je součástí Metawidgetu, je uveden na ukázce kódu 7.11.

```
public class HtmlSimpleLayoutRenderer extends Renderer {

   private static final String   AFTER_FACET = "after";

   @Override
   public void encodeBegin( FacesContext context, UIComponent component )
      throws IOException {

      ResponseWriter writer = context.getResponseWriter();

      // Important to wrap output in something with an id,
      // so that 'label for' can refer to it

      writer.startElement( "div", component );
      writer.writeAttribute( "id", component.getClientId( context ), "id" );

      // Display as 'inline' so as not to affect formatting.
      // However don't use a 'span' because we're not allowed
      // to put some tags (i.e. 'div', 'table') inside a 'span'

      writer.writeAttribute( "style", "display: inline", null );
      super.encodeBegin( context, component );
   }

   @Override
   public void encodeEnd( FacesContext context, UIComponent component )
      throws IOException {

      super.encodeEnd( context, component );
      FacesUtils.render( context, component.getFacet( AFTER_FACET ) );

      ResponseWriter writer = context.getResponseWriter();
      writer.endElement( "div" );
   }
}
```

Ukázka kódu 7.11: Jednoduchý Layuot v Metawidgetu

7.6.2 Layout v AspectFaces

V AspectFaces je layout popsán pomocí HTML obohaceného o speciální tag sloužící k iterování přes jednotlivé atributy a také o dva konkrétní EL výrazy, které označují místo v kódu, kde má být widget aplikován. Jelikož AspectFaces umožňuje obalit vstupní pole

tagem `div` již v prezentační šabloně, vytváření podobného "layoutu", jaký je uveden na ukázce kódu 7.11, není nutné. Z tohoto důvodu také AspectFaces ke generování UR definici layoutu nevyžaduje. Pokud není layout definován, jednotlivé widgety jsou naskládány za sebou v pořadí, v jakém byly zpracovány po inspekci.

Příklad definice tabulkového layoutu pomocí `div` tagů je uveden v ukázce kódu 7.12. Iterační blok označuje párový tag `af:iteration-part`. Kód uvnitř iteračního bloku bude vygenerován při každé iteraci. K označení místa, kde má být aplikován následující widget lze využít výraz `$af:next$`. Může se však stát, že budeme chtít některé konkrétní widgety zpracovávat jiným způsob než ostatní. K označení místa, kde má být aplikován konkrétní widget tak můžeme využít výraz `$af:nazevWidgetu$`.

```
<div class="tableLayout">
   <af:iteration-part maxOccurs="100">
      <div>
         <div>$af:next$</div>
         <div>$af:next$</div>
      </div>
    </af:iteration-part>
</div>
```

Ukázka kódu 7.12: Table layuot v AspectFaces

7.7 Konfigurace

7.7.1 Konfigurace Metawidgetu

Metawidget nevyžaduje ke svému fungování žádnou konfiguraci. Místo toho aplikuje přístup *convention over configuration*, který upřednostňuje vhodně zvolené výchozí nastavení oproti zdlouhavé konfiguraci. Nicméně aby se dal Metawidget použít k vytváření UR splňující reálné požadavky, tak si pouze s výchozím nastavením nevystačíme a potřebujeme jej přizpůsobit k obrazu svému. Veškerá konfigurace Metawidgetu je sdružena pouze do jednoho XML souboru. Variant konfigurace se však může v aplikaci vyskytovat hned několik (podobně jako profilů v AspectFaces).

Součástí konfigurace je registrace veškerých komponent archtektury Metawidgetu, tedy Inspectorů, WidgetuBuilderů, WidgetProcessorů a Layoutu. Registrace komponenty je vyjádřena jako tag, jehož název je tvořen názvem komponenty, kterou chceme registrovat a namespace označuje název balíčku, ve kterém se daná komponenta nachází. Příklad registrace Inspectorů je uveden v ukázce kódu 7.13.

```
<metawidget xmlns="http://metawidget.org"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget
       -1.0.xsd">

   <UIMetawidget xmlns="java:org.metawidget.faces.component">
      <inspector>
```

```
        <compositeInspector xmlns="java:org.metawidget.inspector.composite" config
            ="CompositeInspectorConfig">
          <inspectors>
            <array>
              <metawidgetAnnotationInspector xmlns="java:org.metawidget.
                  inspector.annotation"/>
              <propertyTypeInspector xmlns="java:org.metawidget.inspector.
                  propertytype"/>
              <jpaInspector xmlns="java:org.metawidget.inspector.jpa"/>
              <beanValidationInspector xmlns="java:org.metawidget.inspector.
                  beanvalidation"/>
              <customAnnotationsInspector xmlns="java:cz.cvut.cartech.
                  metawidget.inspectors"/>
            </array>
          </inspectors>
        </compositeInspector>
      </inspector>
    </UIMetawidget>

</metawidget>
```

Ukázka kódu 7.13: CompositeInspector v Metawidgetu

7.7.1.1 Konfigurační třídy

Inspectory, InspectionResultProcessory, WidgetBuildery, WidgetProcessory a Layouty jsou z výkonnostních důvodů a z důvodu bezpečného přístupu vláken imutabilní (neměnné). Někdy však bude potřeba, aby uchovávaly vnitřní stav. K tomuto účelu se proto využívají konfigurační třídy. Instance konfigurační třídy je předána jako parametr v konstruktoru komponenty (viz ukázka kódu 7.5), která si z ní vezme potřebné informace nebo jej předá ke zpracování konstruktoru předka. O zpracování konfiguračního souboru, cachování, vytváření imutabilních objektů a jejich znovuvyužití se stará ConfigReader. K tomu, aby mohly být znovuvyužívány i imutabilní objekty, které používají konfigurační třídy, je zapotřebí, aby tyto třídy implementovaly metody equals a hashcode. Příklad takové konfigurační třídy je uveden v ukázce kódu 7.14.

```
public class HtmlWidgetBuilderConfig {

   private String    mDataTableStyleClass;
   private String[]  mDataTableColumnClasses;
   private String[]  mDataTableRowClasses;
   private int       mMaximumColumnsInDataTable = 5;

   /**
    * Sets the maximum number of columns to display in a generated data table.
    * By default, Metawidget generates data tables for collections based on their
    * properties. However data tables can become unwieldy if they have too many
    * columns.
    *
```

```
     * @param maximumColumnsInDataTable
     *            the maximum number of columns in a data table, or 0 for unlimited
     * @return this, as part of a fluent interface
     */
    public HtmlWidgetBuilderConfig setMaximumColumnsInDataTable( int
       maximumColumnsInDataTable ) {
       mMaximumColumnsInDataTable = maximumColumnsInDataTable;
       return this;
    }

    {...other setters, getters, equals and hashCode...}
```

Ukázka kódu 7.14: Konfigurační třída HTMLWidgetBuilderu

Z výše uvedené ukázky kódu je patrné, že podobné konfigurační třídy pro WidgetBuldery či WidgetProcessory bude nutné používat často, jelikož se za jejich pomocí specifikují detaily zobrazení jako například CSS třídy, které mají být aplikovány na vytvářené komponenty nebo maximální počet sloupců generované tabulky. V popisu settru atributu `maximumColumnsInDataTable` je nastíněno možné řešení problému - zobrazení tabulek s mnoha sloupci, který využívá implmentace HTMLWidgetBuilderu. Počet generovaných sloupců je omezen hodnotou tohoto atributu bez možnosti zobrazení ostatních sloupců uživatelem v UR. V aplikaci Cost Report je naproti tomu využit přístup, který tento problém řeší bez omezení dat, jež jsou prezentována uživateli (viz. podkapitola 6.5.1).

7.7.2 Konfigurace AspectFaces

V AspectFaces je konfigurace rozdělena do čtyř částí. V podkapitole 7.4.2 byla zmíněna první z nich - AspectFaces profily. Další částí konfigurace představuje konfigurační soubor aspectfaces-config.xml. Tento konfigurační soubor slouží k registraci výše zmíněných profilů a také AnnotationDecriptorů. Příklad jeho obsahu je uveden v ukázce kódu 7.15.

```
<aspectfaces-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://aspectfaces.com/schema/j2ee"
   xsi:schemaLocation="http://aspectfaces.com/schema/j2ee http://aspectfaces.com/
       schema/j2ee/aspectfaces-config-1.4.0-SNAPSHOT.xsd">

   <configurations-registration>
      <configuration name="default" path="/WEB-INF/af/html.config.xml" check-
          modification="true" lazy-load="true"/>
    </configurations-registration>

    <annotations-registration>
      <!-- Custom annotations -->
      <name>cz.cvut.cartech.aspectfaces.annotationDescriptors.
          UiPlaceholderAnnotationDescriptor</name>

      <!-- AspectFaces annotations -->
      <!-- javax.validation annotations -->
      <!-- JPA annotations -->
```

```
    </annotations-registration>

</aspectfaces-config>
```

Ukázka kódu 7.15: aspectfaces.config.xml

AspectFaces rozšiřuje množinu JSF tagů o `ui` tag. Způsob jeho zpracování je konfigurován v konfiguračním souboru aspectfaces.taglib.xml. Důležitou částí konfiguračního souboru je specifikace třídy handleru, který se stará o zpracování `ui` tagu. Příklad jeho obsahu je uveden v ukázce kódu 7.16. V tomto příkladu je ke zpracování použita vlastní implementace handleru CustomAFHandler.

```
<facelet-taglib xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns
       /javaee/web-facelettaglibary_2_0.xsd"
   version="2.0">
   <namespace>http://codingcrayons.com/aspectfaces</namespace>

   <tag>
      <tag-name>ui</tag-name>
      <component>
         <component-type>ui</component-type>
         <renderer-type>uigen.renderer</renderer-type>
         <handler-class>cz.cvut.cartech.aspectfaces.handler.CustomAFHandler</
             handler-class>
      </component>
   </tag>
</facelet-taglib>
```

Ukázka kódu 7.16: aspectfaces.taglib.xml konfigurace

Poslední částí konfigurace AspectFaces je soubor aspectfaces.properties. V souboru jsou specifikovány lokace hlavičky a patičky používaných ke kompozici generovaného bloku kódu. Dále je zde možné změnit výchozí znak používaný k označení začátku a konce AspectFaces EL výrazu. Součástí tohoto souboru je také deklarace anotací, použitých k seřazení atributů, určení podmíněného zobrazení na základě uživatelské role, profilů zobrazení a také anotací určené k označení atributů, které mají být ignorovány. Volitelně je možné také specifikovat názvy atributů, které bude aspectfaces globálně ignorovat. Příklad obsahu tohoto souboru je uveden v ukázce kódu 7.17.

```
# global
template-variable-identifier=$

# header and footer for JSF composition
gallery=WEB-INF/af/profile/
header=WEB-INF/af/profile/shared/header.xhtml
footer=WEB-INF/af/profile/shared/footer.xhtml

# model annotations
```

```
order-annotation=com.codingcrayons.aspectfaces.annotations.UiOrder
profiles-annotation=com.codingcrayons.aspectfaces.annotations.UiProfiles
user-roles-annotation=com.codingcrayons.aspectfaces.annotations.UiUserRoles

# globally ignored fields
ignore-fields=parentId
ignoring-annotations=UiIgnore javax.persistence.Transient
```

Ukázka kódu 7.17: aspectfaces.properties konfigurace

7.8 Použití v UR

7.8.1 Použití Metawidgetu

Integrace Metawidgetu do UR se nijak neliší od integrace jiných widgetů. Pomocí tagu `metawidget` je označeno místo v kódu, kde má být v runtimu aplikován vygenerovaný kód. Metawidget umožňuje generovat UR pro kolekce objektů, objekty, ale i pro jednotlivé atributy pomocí `value` atributu `metawidget` tagu. Jelikož kolekce objektů je také objekt, bude se Metawidget ve výchozím nastavení snažit provést inspekci této kolekce. V případě, že požadujeme, aby se provedla inspekce třídy, ve které je tato kolekce definována, můžeme využít atribut `inspectFromParent` s hodnotou true. Příklad použití `metawidget` tagu v UR je uveden v ukázce kódu 7.18. Pomocí atributu `rendererType` je v ukázce specifikován typ Rendereru, který bude použit ke generování výsledného HTML kódu.

```
<m:metawidget value="#{indexBean.car.year}" rendererType="simple" />
```

Ukázka kódu 7.18: Použití Metawidgetu v UR

7.8.2 Použití AspectFaces v UR

AspectFaces je do UR zakomponován prostřednictvím `ui` tagu. Ke specifikaci objektu, pro který má být generováno UR, slouží atribut `instance`. Na rozdíl od Metawidgetu neumožňuje provádět generování UR pro celou kolekci objektů (tedy ne najednou) či pouze jeden atribut objektu. `UI` tag se skládá ve výchozím nastavení z mnoha dalších atributů specifikujících výslednou podobu generovaného UR. Množinu atributů `ui` tagu je navíc možné jednoduše rozšířit pomocí vlastní implementace AF handleru. Konfigurace handleru je součástí konfiguračního souboru aspectfaces.taglib.xml (viz podkapitola 7.7.2). Příklad použití `ui` tagu v UR je uveden v ukázce kódu 7.19. V tomto příkladu je použit vlastní tag `autofocusFirst`, který slouží k indikaci, že má být nastaven atribut `autofocus` na první generované formulářové pole.

```
<af:ui instance="#{indexBean.car}" edit="true" autofocusFirst="true" />
```

Ukázka kódu 7.19: Použití AspectFaces v UR

7.9 Možnosti adaptace UR

Implementovat adaptaci UR na základě metadat, dostupných ve fázi vývoje, je možné například pomocí JSF frameworku. Co ale není možné, je využití runtime metadat k provádění složitějších adaptací. Navíc adaptace manuálně vytvářeného UR je typicky realizovatelná pouze způsobem vytváření několika verzí jedné stránky, kde na základě předem specifikovaných podmínek je použita vhodná varianta pro aktuální kontext použití. Toto řešení je časově náročné a nákladné, což nejsou zrovna vhodné vlastnosti pro praktické použití. Nástroje pro automatizované vytváření UR proto otevírají nové možnosti jak tuto problematiku řešit efektivně a za vynaložení minimálního úsilí programátora.

AspectFaces i Metawidget provádějí generování UR v runtimu, čímž splňují předpoklad pro provádění pokročilejší adaptace UR na základě runtime metadat. Ne vždy je ale tento předpoklad dostačující. Příkladem mohou být domain driven nástroje OpenXava nebo Apache Isis, které neumožňují programátorovi využít runtime metadata k adaptaci UR, ačkoliv provádějí generování UR v runtimu. Metawidget i AspectFaces však mají v porovnání s těmito nástroji otevřenější architekturu a práce s nimi probíhá na nižších vrstvách abstrakce. Díky těmto vlastnostem tak mohou provádět adaptaci UR za využití runtime metadat.

7.9.1 AspectFaces

Ke zpracování runtime metadat je nutné v AspectFaces vytvořit vlastní implementaci handleru, který provádí zpracování `ui` tagu. K jejímu vytvoření stačí podědit výchozí handler DefaultAFGeneratorHandler, který již obsahuje implementaci problematiky aspect weavingu a vykreslování komponent, a poté přepsat metodu `hookAddToAFContext`, která je volaná před samotným zpracováním tagu a může být tak využita ke zpracování runtime metadat [5]. Tato metoda umožňuje manipulovat přímo s AF kontextem, který je předán jako její parametr. Příklad možné manipulace s AF kontextem, který je součástí dokumentace AspectFaces [3], je uveden v ukázce kódu 7.20.

```
public class CustomHandler extends DefaultAFGeneratorHandler {

   @Override
   protected void hookAddToAFContext(Context context) {

      // publish list of current UI profiles
      context.setProfiles(new String[]{"US"});

      // public list of current user's security roles
      context.setRoles(new String[]{"Admin"});

      // set layout to use for rendering
      context.setLayout("specialLayout.xhtml");

      // custom constant in JSF context
```

[5] Mezi další možnosti využití této metody patří také zpracování programátorem definovaných atributů `ui` tagu, rozšiřujících množinu základních atributů.

```
        context.getVariables().put("customVariable", "Value of the variable");

        // extension of EL by custom class with util functions
        context.getVariables().put("utils", new Utils());
    }
}
```

Ukázka kódu 7.20: Zpracování runtime metadat v AspectFaces

7.9.2 Metawidget

Zpracování runtime metadat je nutné provést před vytvářením widgetů. Vytváření widgetů předchází v Metawidget lifecyclu inspekce, kterou provádí Inspectory, a zpracování získaných metadat, které provádějí InspectionResultProcessory. Zpracování runtime metadat proto musí být součástí jedné z těchto komponent. Inspectory však nemají přístup k žádnému úložišti, které uchovává informace o výsledné podobě UR. Jedinou komponentou architektury Metawidgetu, která může zpracování runtime metadat provést, je tak InspectionResultProcessor.

Metawidget však nemá vlastní kontext pro uchovávání informací specifikující výslednou podobu UR, který by mohl být jednoduše pozměněn v závislosti na runtime metadatech podobně jako je tomu v AspectFaces. Místo toho jsou tyto informace roztříštěny po konfiguračních třídách, atributech komponenty metawidgetu a dalších objektech, které uchovávají informace získané při inspekci a zpracování runtime metadat je tak náročné.

7.10 Výsledná podoba UR

Obrázek 7.1: UR aplikace Cost Report

7.11 Vyhodnocení

V této kapitole je popsán konkrétní dopad, který mělo zavedení automatizace tvorby UR na výslednou podobu kódu UR. Ke zhodnocení tohoto dopadu byla použita metrika SLOC, tedy počet řádek kódu. Z počtu řádků kódu je možné odhadnout časovou náročnost jednotlivých způsobů vytváření UR.

V objektových jazycích představuje metrika uvažující počet skutečných (fyzických) řádků kódu (SLOC-P) výrazně nepřesný odhad závislý na formátování kódu. Proto pro objektové jazyky byla použita metrika SLOC-L, která uvažuje tzv. logické řádky kódu, tedy takové řádky kódu, které je možné provést. Pro ostatní jazyky byla použita metrika SLOC-P.

	JSF	**Metawidget**	**AspectFaces**
Entity	323	348	344
Konfigurace generování	0	303	383
Kód UR	1269	691	626
Celkem	**1592**	**1342** (o 15,7 % méně)	**1353** (o 15 % méně)

Tabulka 7.2: Vyhodnocení řádků kódu

Entity

Dodatečná metadata specifikující výslednou podobu UR, tvoří méně než 8% celkového kódu rich entit. Metawidget neumožňuje nastavit globálně ignorované atributy. Místo toho

je nutné v každé entitě označit atribut, který má být ignorován, anotací @UiHidden. Proto mají rich entity o 4 řádky více v porovnání s AspectFaces.

Konfigurace generování

V konfiguraci generování jsou zahrnuty veškeré programátorem vytvářené soubory, které jsou zapotřebí k automatizovanému generování UR daným nástrojem. V AspectFaces jsou součástí běžně prováděné konfigurace také mapovací pravidla(viz podkapitola 7.4.2). Naproti tomu v Metawidgetu se s těmito pravidly programátor zaobírá pouze tehdy, když chce vytvořit vlastní WidgetBuilder. AspectFaces vyžaduje ke svému fungování kromě konfiguračních souborů také vytvoření generických widgetů (tagů). Ty je však možné díky oddělení cross-cutting concernů vytvořit pouze jednou a následně je znovuvyužívat napříč projekty. Ke zjednodušení použití AspectFaces je součástí dokumentace také základní konfigurace generování, která byla využita také při vytváření AspectFaces verze aplikace Cost Report. V Metawidgetu je základní konfigurace generování standardních widgetů součástí výchozího nastavení, které je automaticky aplikováno, pokud není specifikováno jinak.

Kód UR

Kód UR se díky automatizovanému generování výrazně zredukoval. Konkrétně použitím Metawidgetu se počet řádků kódu UR snížil o 45,5% a použitím AspectFaces dokonce o 50,7%. Vyšší počet řádků kódu u Metawidgetu je způsoben komplikovaným vytvářením vlastních Layoutů, kterému je zkrátka občas výhodnější se vyhnout a použít Metawidget pouze ke generování jednotlivých atributů, k jejich vykreslení použít Simple Renderer a Layout vytvořit klasickým způsobem.

Celkový počet řádků

Celkový počet řádků, který byl snížen při použití automatizace zhruba o 15%, by se mohl zdát ne příliš obdivuhodný. Vezmeme-li však v úvahu, že kód konfigurace generování je z větší části tvořen výchozím nastavením nebo případně znovupoužitým kódem, tak je možné považovat tento výsledek za velmi dobrý. Výsledné UR je navíc jednoduše udržovatelné a pravděpodobnost chyb v UR, které jinak vznikají při duplikaci informací z vrstvy doménových objektů, se výrazně snížila.

Kapitola 8

Závěr

V rámci této bakalářské práce jsem pro účely porovnání nástrojů AspectFaces a Metawidget vytvořil webovou aplikaci Cost Report sloužící k evidenci součástek vozidla formulového typu a exportu dokumentu Cost Report, který obsahuje podrobný přehled o všech atributech tvořících celkovou cenu vozu. Aplikace byla vytvořena ve třech verzích lišících se v přístupu, jakým bylo vytvářeno UR. UR první verze aplikace bylo vytvořeno manuálně pomocí frameworku JavaServer Faces, zatímco u zbylých dvou verzí byla k vytvoření fragmentů UR, které jsou nejvíce závislé na datovém modelu aplikace, použita automatizace. K automatizaci vytváření UR byly použity nástroje Asepctfaces a Metawidget, každý z nich odděleně v samostatné verzi aplikace.

Na základě praktických zkušeností, získaných při implementaci aplikace Cost Report a analýzy výsledného kódu UR jsem vytvořil porovnání nástrojů AspectFaces a Metawidget. Generování UR pomocí Metawidgetu se díky nutnosti pracovat přímo s JSF API ukázalo v porovnání s AspectFaces, který umožnuje provádět definici widgetů i layoutu jednoduše pomocí DSL, náročnější. Další nevýhodou Metawidgetu se ukázala být absence vlastního kontextu, díky níž není možné přistupovat ke specifikaci generovaného UR z jednoho místa a provádět její modifikace. Absence kontextu Metawidgetu způsobovala problém především při zpracování runtime metadat, kde nebylo možné nalézt jednoduchý způsob, jakým by mohla být tato metadata zpracována podobně jako ve vlastní implementaci AspectFaces handleru.

AspectFaces je však stále v ranných fázích vývoje, s čímž souvisí teprve postupně vznikající dokumentace, větší množství chyb než v Metawidgetu, který již byl otestován nemalou skupinou uživatelů a také menší podpora technologií. Implementace aplikace Cost Report tak posloužila k odhalení několika chyb v AspectFaces a přinesla také jedno vylepšení, kterým je evidence indexu aktuálně zpracovávaného atributu v AF kontextu. V průběhu implementace AspectFaces verze aplikace jsem narazil na problém nekompatibility AspectFaces s JSF verze 2.2 a podílel se na opravě modulu OnDemand, který tuto nekompatibilitu způsoboval.

Použitím automatizace vytváření UR jsem dosáhl snížení počtu celkového počtu řádků kódu o 15% u AspectFaces a 15,7% u Metawidgetu. Úspora kódu UR byla dokonce 50% u AspectFaces a 45% u Metawidgetu. Nevýhoda nutnosti explicitní specifikace dodatečných metadat se projevila nárůstem LOC rich entit pouze o 8%.

Literatura

[1] *ABL technical summary* [online]. Dostupné z: <http://www.automatedbusinesslogic.com/overview/technical-summary>.

[2] *Apache Isis* [online]. Dostupné z: <http://isis.apache.org/>.

[3] *AspectFaces documentation* [online]. Dostupné z: <http://wiki.codingcrayons.com/display/af/AspectFaces>.

[4] *Automated business logic* [online]. Dostupné z: <http://www.automatedbusinesslogic.com/>.

[5] *Automatic Programming* [online]. Dostupné z: <http://www.cs.utexas.edu/users/novak/autop.html>.

[6] *Automatizace testů pomocí selenia aneb cesta tam a zase zpátky* [online]. Dostupné z: <http://www.ibacz.eu/blog/-/blogs/automatizace-testu-pomoci-selenia-aneb-cesta-tam-a-zase-zpatky>.

[7] *Bean validation Specification* [online]. Dostupné z: <http://beanvalidation.org/1.1/spec/>.

[8] *Domain driven automation* [online]. Dostupné z: <http://java.dzone.com/articles/domain-driven-automation-article>.

[9] *Elevator Pitch Cartoon - Metawidget* [online]. Dostupné z: <http://metawidget.sourceforge.net/elevator.php>.

[10] *Frequently Asked Questions - Metawidget* [online]. Dostupné z: <http://metawidget.sourceforge.net/doc/faq/core.php>.

[11] *GUI widget* [online]. Dostupné z: <http://en.wikipedia.org/wiki/GUI_widget>.

[12] *Java Unifed Expression Language* [online]. Dostupné z: <http://juel.sourceforge.net>.

[13] *Master detail interface* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Master_detail_interface>.

[14] *Metawidget* [online]. Dostupné z: <http://metawidget.org/>.

[15] *Rapid appliacation development* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Rapid_application_development>.

[16] *Restful Objects* [online]. Dostupné z: <http://restfulobjects.org/>.

[17] Pierre A Akiki, Arosha K Bandara, and Yijun Yu. Adaptive model-driven user interface development systems. *ACM Computing Surveys (CSUR)*, 47(1):9, 2014.

[18] Tomas Cerny, Karel Cemus, Michael J Donahoo, and Eunjee Song. Aspect-driven, data-reflective and context-aware user interfaces design. *ACM SIGAPP Applied Computing Review*, 13(4):53–66, 2013.

[19] Tomas Cerny, Michael J Donahoo, and Eunjee Song. Towards effective adaptive user interfaces design. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 373–380. ACM, 2013.

[20] David Thevenin Quentin Limbourg Laurent Bouillon Jean Vanderdonckt Gaëlle Calvary, Joëlle Coutaz. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS*, 15:289–308, 2003.

[21] Joëlle Coutaz Gaëlle Calvary. The problem space of ui adaptation, ui plasticity. 2012.

[22] Richard Kennard. *Metawidget User Guide and Reference Documentation*. verze: 3.8.

[23] Richard Kennard. Duplication in user interfaces. 2010.

[24] Richard Kennard. Derivation of a general purpose architecture for automatic user interface generation. 2012.

[25] Brad Myers, Scott E Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[26] Brad A Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202. ACM, 1992.

[27] AF Norcio and J Stanley. Adaptive human-computer interfaces. Technical report, DTIC Document, 1988.

[28] Javier Paniza. *Learn OpenXava by example*. CreateSpace Independent Publishing Platform, 2011. ISBN:1466369701.

[29] David Raneburger. Interactive model driven graphical user interface generation. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 321–324. ACM, 2010.

[30] Sanjin Sehic, Fei Li, Stefan Nastic, and Schahram Dustdar. A programming model for context-aware applications in large-scale pervasive systems. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on*, pages 142–149. IEEE, 2012.

[31] David Thevenin, Gaëlle Calvary, and Joëlle Coutaz. A development process for plastic user interfaces. In *Proceedings of the CHI'2001 Workshop: Transforming the UI for Anyone, Anywhere, Seattle, Washington, USA*, 2001.

Příloha A

Popis případů užití

UC1 - Generovat Cost Report

Uživatel bude moci provést export dat s údaji o ceně konkrétního auta a jeho součástech. Výsledný dokument bude svou strukturou i obsahem splňovat standard pro Cost Report. Tento případ užití reflektuje požadavek FReq8.

I když je tento případ užití jedním z nejdůležitějších na celé aplikaci, nesouvisí přímo s předmětem bakalářské práce a jeho implementaci tak nebyla věnována velká pozornost.

UC2 - Zobrazit/skrýt sloupce tabulky

Uživatel bude moci zobrazit/skrýt sloupce tabulky. Tento případ užití reflektuje požadavek FReq5.

Scénář

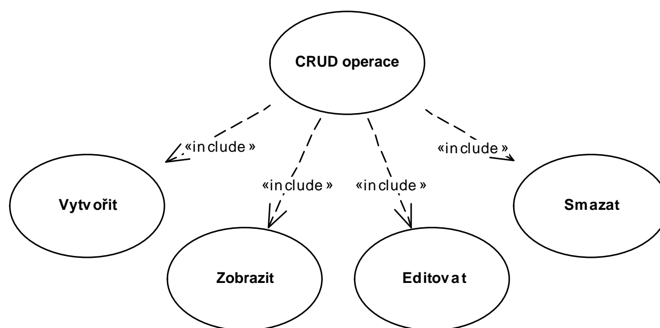
1. Uživatel klikne na tlačítko pro zobrazení seznamu sloupců tabulky.
2. Systém zobrazí seznam dvojic názvů sloupců tabulky a k němu přiřazeným checkboxům. Pro sloupce, které jsou zobrazeny bude checkbox zaškrtnutý, pro ostatní nikoliv.
3. Uživatel zaškrtne/odškrtne checkbox přiřazený k názvu sloupce, který chce zobrazit/skrýt.
4. Systém zobrazí/skryje daný sloupec tabulky.

UC3 - Zobrazit/skrýt nápovědu u formulářových polí

Uživatel bude moci zobrazit/skrýt nápovědu u formulářových polí. Tento případ užití reflektuje požadavek FReq7.

UC4 - CRUD Aut

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Aut. Tento případ užití reflektuje požadavek FReq1.



Obrázek A.1: CRUD operace

UC5 - CRUD Systémů

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Systémů. Tento případ užití reflektuje požadavek FReq1.

UC6 - CRUD komponent

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Komponent. Tento případ užití reflektuje požadavek FReq1.

UC7 - CRUD spojovacích prvků

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Spojovacích prvků. Tento případ užití reflektuje požadavek FReq1.

UC8 - CRUD materiálů

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Materiálů.

UC9 - CRUD prací

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Provedených prací. Tento případ užití reflektuje požadavek FReq1.

UC10 - CRUD nástrojů

Uživatel bude moci provádět CRUD operace(viz obrázek A.1) jednotlivých Nástrojů. Tento případ užití reflektuje požadavek FReq1.

UC11 - Řadit záznamy v tabulkách

Uživatel bude moci řadit záznamy v tabulákách podle jednitlivých sloupců a to jak sestupně, tak i vzestupně. Tento případ užití reflektuje požadavek FReq6.

Příloha B

Obsah přiloženého CD

app - adresář se zdrojovými soubory aplikace Cost Report

- **aspectFacesVersion** - AspectFaces verze aplikace
- **jsfVersion** - JSF verze aplikace
- **metawidgetVersion** - Metawidget verze aplikace

war - adresář s .war soubory

selenium - adresář se selenium testy vygenerovanými pomocí nástroje Selenium IDE

text - text této bakalářská práce