České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Tomáš Turek**

Studijní program: Otevřená informatika (magisterský)
Obor: Softwarové inženýrství

Název tématu: **Využití aspektově-orientovaného přístupu pro tvorbu adaptivních uživatelských rozhraní**

Pokyny pro vypracování:

Nastudujte současné frameworky a doménově specifické jazyky pro tvorbu adaptivních uživatelských rozhraní. Zaměřte se na XML User Interface Language, User Interface Markup Language, User interface protokol a dále pak Java Server Faces. Navrhněte a implementujte kontextově závislou Java EE aplikaci a s pomocí nástroje AspectFaces vytvořte adaptivní uživatelská rozhraní pro zmíněné frameworky a jazyky. Zhodnoťte možnosti nástroje AspectFaces z hlediska transformace strukturních informací aplikace do fragmentů uživatelského rozhraní reprezentujících data. Vyhodnoťte výhody, nevýhody a omezení daných frameworků a jazyků.

Seznam odborné literatury:

[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

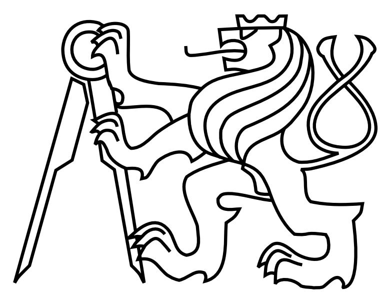
[3] Tomas Cerny and Eunjee Song. 2011. UML-based enhanced rich form generation. In Proceedings of the 2011 ACM Symposium on Research in Applied Computation (RACS '11). ACM, New York, NY, USA, 192-199. DOI=10.1145/2103380.2103420 http://doi.acm.org/10.1145/2103380.2103420

Vedoucí: Ing. Tomáš Černý, MSc.

Platnost zadání: do konce letního semestru 2014/2015

V Praze dne 25. 2. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Využití aspektově-orientovaného přístupu pro tvorbu adaptivních uživatelských rozhraní

Bc. Tomáš Turek

Vedoucí práce: Ing. Tomáš Černý, MSc.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

31. prosince 2014

Poděkování

Touto cestou bych chtěl poděkovat panu Ing. Tomáši Černému, MSc. za vedení a cenné rady při vytváření této diplomové práce. Dále bych chtěl poděkovat panu Ing. Miroslavu Macíkovi za poskytnutí materiálů a rad k jazyku UIProtocol. Také bych chtěl poděkovat své rodině, za jejich podporu během mého studia a tvorby diplomové práce.

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 29. 12. 2014 ....

Abstract

This thesis deals with the use of the aspect-oriented approach for creating adaptive user interfaces and use of this approach with the frameworks and domain-specific languages for creating user interfaces. They are XML User Interface Language, User Interface Markup Language, User interface protocol and JavaServer Faces. These languages are used with the tool AspectFaces to create adaptive user interfaces for context-aware Java EE application. Subsequently evaluated their advantages, disadvantages and limitations for creating adaptive user interfaces.

Abstrakt

Tato diplomová práce se zabývá využitím aspektově-orientovaného přístupu pro tvorbu adaptivních uživatelských rozhraní a využitím tohoto přístupu s frameworky a doménově specifickými jazyky pro tvorbu uživatelských rozhraní. Jedná se o jazyk XML User Interface Language, User Interface Markup Language, User interface protokol a JavaServer Faces. Tyto jazyky jsou použity s nástrojem AspectFaces k vytvoření adaptivních uživatelských rozhraní pro kontextově závislou Java EE aplikaci. Následně jsou zhodnoceny jejich výhody, nevýhody a omezení pro tvorbu adaptivních uživatelských rozhraní.

Obsah

1 Úvod **1**
1.1 Motivace . . . 1
1.2 Cíle práce . . . 1
1.3 Definice . . . 2
1.4 Struktura práce . . . 2

2 Rešerše **5**
2.1 Adaptivní uživatelské rozhraní . . . 5
2.1.1 Adaptace obsahu . . . 5
2.1.2 Adaptace navigace . . . 6
2.1.3 Definice . . . 6
2.1.4 Kontext . . . 6
2.1.4.1 Rozdělení kontextu . . . 7
2.2 Aspektově-orientovaný přístup . . . 7
2.2.1 Koncept . . . 8
2.3 AspectFaces . . . 8
2.3.1 Inspekce kódu . . . 9
2.3.2 Transformace . . . 10
2.3.2.1 Prezentační pravidla . . . 11
2.3.2.2 Kompozice šablon . . . 12
2.3.2.3 Rozvržení layoutu . . . 12
2.3.3 Integrace . . . 12

3 Jazyky pro tvorbu UI **13**
3.1 User Interface Markup Language . . . 13
3.1.1 Koncept . . . 13
3.1.2 Struktura dokumentu . . . 14
3.2 XML User Interface Language . . . 16
3.2.1 Overlays . . . 16
3.2.2 XBL Binding . . . 16
3.2.3 XUL Template . . . 17
3.3 JavaServer Faces . . . 18
3.3.1 Životní cyklus . . . 18
3.3.2 Component model . . . 19
3.3.3 Facelets . . . 20

3.4 User interface protocol 21
3.4.1 Vlastnosti 22
3.4.2 Modely 22
3.4.3 Uživatelská rozhraní 22
3.4.4 Akce 23
3.4.5 Události 24
3.4.6 Server 24
3.4.7 Klient 24

4 Návrh **25**
4.1 Kontextově závislá aplikace 25
4.1.1 Požadavky 25
4.1.2 Specifikace pravidel adaptace 26
4.1.3 Kontextové parametry 27
4.1.4 Architektura 28
4.2 Adaptace fragmentů UI 28

5 Realizace **31**
5.1 Kontextová aplikace 31
5.1.1 Architektura 31
5.1.2 Datový model 32
5.1.3 Kontext 33
5.1.4 Pravidla adaptace 33
5.1.5 Realizace pravidel 34
5.2 JavaServer Faces 36
5.2.1 Architektura 36
5.2.2 Generování fragmentů 36
5.2.3 Adaptivní uživatelské rozhraní 37
5.3 XUL 39
5.3.1 Architektura 39
5.3.2 Generování fragmentů 40
5.3.3 Adaptivní uživatelské rozhraní 40
5.4 UIML 43
5.4.1 Architektura 43
5.4.2 Generování fragmentů 45
5.4.3 Adaptivní uživatelské rozhraní 45
5.5 UIProtocol 46
5.5.1 Architektura 47
5.5.2 Generování fragmentů 48
5.5.3 Adaptivní uživatelské rozhraní 49

6 Testování **51**
6.1 Kritéria porovnání 51
6.2 Experiment měření kódu 52
6.2.1 Nastavení experimentu 53
6.2.2 Výsledky měření 53

6.2.3 Vyhodnocení . . . 55
6.3 Experiment měření datového přenosu . . . 56
6.3.1 Nastavení experimentu . . . 56
6.3.2 Výsledky měření . . . 57
6.3.3 Vyhodnocení . . . 57
6.4 Srovnání jazyků . . . 58

7 Závěr **61**
7.1 Budoucí vývoj . . . 62

A Seznam použitých zkratek **67**

B Instalační příručka **69**
B.0.1 Softwarové požadavky . . . 69
B.0.2 Instalace serveru . . . 69
B.0.3 Kompilace projektu . . . 69
B.0.4 Spuštění klienta . . . 70
B.0.4.1 JSF . . . 70
B.0.4.2 XUL . . . 70
B.0.4.3 UIP . . . 71

C Vlastnosti jazyků **73**

D Uživatelská rozhraní **75**

E Obsah přiloženého CD **81**

Seznam obrázků

2.1 Aspect weaving 9
2.2 Komponenty AspectFaces [13] 9
2.3 Fáze inspekce [13] 10
2.4 Fáze transformace [13] 11

3.1 Meta-Interface Model [27] 14
3.2 XBL Binding [11] 17
3.3 Životní cyklus JSF [12] 19
3.4 UIP komunikace [23] 21
3.5 Generování uživatelského rozhraní s ohledem na kontext [23] 23

4.1 Doménový model kontextové aplikace 26
4.2 Adaptace formulářových položek 28
4.3 Adaptace výpisu dat 29

5.1 Struktura aplikace 31
5.2 Sekvenční diagram kontextově závislé aplikace 33
5.3 Architektura systému s JPA aplikací, AspectFaces, UIPServerem a UiGE [24] 47

C.1 Struktura XULRunner aplikace 73

D.1 JSF - obrazovky UI 75
D.2 JSF - obrazovky mobilního UI 76
D.3 XUL - obrazovky UI 77
D.4 XUL - obrazovky mobilního UI 78
D.5 UIP - obrazovky UI 79

E.1 Seznam přiloženého CD 81

Seznam tabulek

2.1 Asociace kontextových parametrů do jednotlivých tříd . . . 7
2.2 Typy Annotation Descriptorů . . . 10

4.1 Rozdělení kontextových parametrů . . . 28

5.1 Generované UI fragmenty pro XUL aplikaci . . . 40

6.1 Výsledky měření SLOC UI. . . . 53
6.2 Výsledky měření znovupoužitelnosti zdrojového kódu. . . . 54
6.3 Výsledky měření adaptivního zdrojového kódu. . . . 54
6.4 Výsledky měření potřebného úsilí pro realizaci adaptace. . . . 55
6.5 Testovací sestava . . . 57
6.6 Výsledky měření komunikace . . . 58

C.1 Porovnání vlastností jazyků pro tvorbu UI . . . 74

Kapitola 1

Úvod

V této kapitole představím motivaci a cíle mé diplomové práce a popíši obsah jednotlivých kapitol.

1.1 Motivace

Každá aplikace potřebuje, pro interakci s uživatelem mít, nějaký druh uživatelského rozhraní. Tato uživatelská rozhraní jsou velmi důležitá a jejich hlavním úkolem je poskytovat pokud možno nejlepší komfort obsluhy. Zde vzniká problém s narůstajícím počtem různých uživatelů se specifickými požadavky.

Roste totiž počet různých elektronických zařízení a jejich používání v odlišných prostředích s rozmanitými typy uživatelů. Je potřeba vytvářet taková uživatelská rozhraní, která se dokáží adaptovat v závislosti na aktuálním kontextu použití. Ten charakterizuje situaci ve které je aplikace využívaná. Může zahrnovat aktuální prostředí, vlastnosti zařízení se kterým uživatel ovládá aplikaci a v neposlední řadě i schopnosti a preference samotného uživatele.

Pro takto kontextově závislou aplikaci je nutné vynaložit spoustu prostředků na vývoj a údržbu adaptivního uživatelského rozhraní. Jednou z hlavních příčin této náročnosti je komplexnost a obsah řady interaktivních prvků. Jedním z příkladů je vytvoření aplikace pro více různých platforem. Často musíme vytvořit několik separátních aplikací, protože existují rozdílné technologie nebo rozdílné interakce se zařízením. To má za následek zvýšení nákladů a nižší pružnost při změnách. Tyto změny je nutno udělat pro každou platformu zvlášť.

Hlavní motivací této práce je vyhodnotit, jak lze zefektivnit vývoj adaptivních uživatelských rozhraní, využijeme-li aspektově-orientovaného přístupu a zkombinujeme ho s různými doménově specifickými jazyky určenými pro tvorbu uživatelských rozhraní.

1.2 Cíle práce

Cílem práce je nastudovat a následně vyhodnotit výhody, nevýhody a omezení současných frameworků a doménově specifických jazyků pro tvorbu adaptivního uživatelského rozhraní s využitím aspektově-orientovaného přístupu. V práci se zaměřím na čtyři různé jazyky a jejich frameworky. Jsou to: XML User Interface Language (XUL), User Interface Markup

Language (UIML), User interface protokol (UIP) a JavaServer Faces (JSF). Při porovnávání se zaměřím na kritéria ohledně produktivity, znovupoužitelnosti a flexibility.

Pro porovnání těchto jazyků navrhnu a implementuji kontextově závislou Java EE aplikaci s podporou automatického generování uživatelského rozhraní pomocí aspektově orientovaného přístupu. Konkrétně použiji nástroj AspectFaces. Následně pro kontextově závislou aplikaci bude vytvořeno několik adaptivních uživatelských rozhraní postavených na zmíněných jazycích pro tvorbu uživatelských rozhraní.

Součástí práce je i zhodnocení možností nástroje AspectFaces u vytvořených adaptivních uživatelských rozhraní z hlediska transformace strukturních informací aplikace do fragmentů uživatelského rozhraní.

1.3 Definice

V této části definuji hlavní pojmy, které budu velmi často v textu této práce používat.

Kontext jsou veškeré informace, které charakterizují situaci určité entity. Entitou je osoba, místo nebo objekt, který je považován za relevantní pro interakci mezi uživatelem a aplikací, včetně uživatele a aplikace samotné.

Kontextově závislé uživatelské rozhraní je uživatelské rozhraní, které se změní, když se změní kontext. Kontextová uživatelská rozhraní jsou buď adaptivní, nebo adaptabilní. **Adaptivní uživatelské rozhraní** je to, ve kterém systém aktivuje kontextové změny. **Adaptabilní uživatelské rozhraní** je takové, ve kterém uživatel iniciuje změnu v souvislostech.

Aspektově-orientované programování (AOP) je programovací paradigma, které si klade za cíl zvýšit modularitu tím, že umožňuje oddělení částí logiky programů do tzv. aspektů.

Doménově specifický jazyk (DSL) je programovací jazyk, který je prostřednictvím vhodné abstrakce a výrazového slovníku zaměřen na omezenou, konkrétní problémovou doménu.

Widgety jsou primární stavební bloky uživatelského rozhraní, jako jsou tlačítka nebo dialogy.

1.4 Struktura práce

Tato práce se skládá z následujících částí:

Úvod o problematice a záměrech diplomové práce a její struktuře.

Rešerše představí v první části kapitoly problematiku adaptivního uživatelského rozhraní a kontextově závislé aplikace. Dále bude probráno programové paradigma pro aspektově orientované programování a nástroj AspectFaces.

Jazyky pro tvorbu UI V této kapitole se zabývám analýzou jednotlivých jazyků a frameworků pro tvorbu uživatelského rozhraní.

Návrh zde se zaobírám návrhem kontextově závislé aplikace, která bude následně sloužit pro porovnání jednotlivých jazyků.

Realizace popisuje jakými způsoby jsou implementována jednotlivá rozhraní a jejich propojení s kontextově závislou aplikací. U jednotlivě vytvořených rozhraní je popsáno i využití nástroje AspectFaces.

Testování popisuje experimenty prováděné nad vytvořenou testovací aplikací. Je zde popsáno provedení a výsledky experimentů s následným zhodnocením pro porovnávané jazyky.

Závěr shrne výsledky práce a zhodnocení jejího přínosu. Představím i možné budoucí navázání k této práci.

Kapitola 2

Rešerše

V této kapitole představím rešerši o kontextově závislých aplikacích a adaptivních uživatelských prostředí. Ze začátku definuji základní definice potřebné pro řešení problematiky adaptivních rozhraní.

2.1 Adaptivní uživatelské rozhraní

Kvalitní uživatelské rozhraní umožňuje efektivní komunikaci mezi uživatelem a aplikací. Proto dobrý design rozhraní je zásadní pro budoucí úspěch jakékoliv aplikace. Zde vzniká výzva pro vývojáře vytvořit UI takové, aby uživatelům usnadnilo efektivní vykonávání úkolů. Design je velmi náročný s ohledem na rozdílné požadavky uživatelů, kde může jít o věk, geografii, úroveň znalostí, osvětlení, a tak dále. Rozhraní, které vyhovuje jednomu člověku, nemusí totiž vyhovovat druhému. Z toho důvodu je důležité, aby se uživatelské rozhraní dokázalo adaptovat požadavkům uživatele.

Adaptivní uživatelské rozhraní (také známo jako AUI) je druh uživatelského rozhraní, které se dokáže adaptovat dle profilu uživatele a kontextu. Tato kontextově závislá uživatelská rozhraní lze rozdělit na *adaptivní*, *adaptabilní* a *kombinovaná*. Adaptivní uživatelské rozhraní je takové, ve kterém systém aktivuje kontextové změny. Adaptabilní uživatelské rozhraní je takové, ve kterém uživatel iniciuje změnu manuálně. Kombinované představuje kombinaci obou přístupů.

Existují dvě hlavní techniky používané pro adaptaci. Je to adaptace obsahu (Adaptive presentation) a adaptace navigace (Adaptive navigation). [28]

2.1.1 Adaptace obsahu

Cílem adaptace obsahu je zobrazení určité informace na základě aktuálního uživatele. Toto může znamenat, že uživateli se základní znalostí o systému se zobrazí pouze minimum informací. Na druhou stranu uživatel s pokročilou znalostí bude mít přístup k mnohem detailnějším informacím.

Způsobem jakým AUI může dosáhnout této adaptace je pomocí skrývání informací, které mají být prezentovány dle zkušeností uživatele. Další možností je kontrolovat množství odkazů k příslušným zdrojům na stránce.

2.1.2 Adaptace navigace

Adaptace navigace ve smyslu vedení uživatele k vykonání specifického cíle v rámci systému se provádí tím, že se změní způsob jakým je systém ovládán dle určitých faktorů uživatele. Tyto faktory mohou zahrnovat různou úroveň znalostí uživatele, prostředí, aktuální úkol, a tak dále.

Adaptivní navigace lze dosáhnout mnoha způsoby. Například poskytováním odkazů, které navedou uživatele k dosažení konkrétního cíle, nebo měnit dostupné zdroje konkrétnímu uživateli.

2.1.3 Definice

Multiplatformní uživatelské rozhraní je takové uživatelské rozhraní, jehož zdrojový kód je přenositelný mezi různými platformami operačních systémů.

Konkrétní uživatelské rozhraní (CUI) je uživatelské rozhraní, které je určené pro konkrétní platformu, definuje způsob interakce a konečnou podobu UI.

Abstraktní uživatelské rozhraní je popis uživatelského rozhraní v podobě nezávislé na platformě, které je možné transformovat na konkrétní uživatelské rozhraní. Abstraktní uživatelské rozhraní je nezávislé na způsobu interakce (například grafická, hlasová, hmatová, 3D) a technologických možnostech (například hardware, programovací jazyk).

2.1.4 Kontext

Pojem kontext je velmi široce používaný termín. K jeho definici ohledně adaptivních uživatelských rozhraní použiji dvě definice. První definice od Dey a Abowd [5] definovali kontext, kontextové povědomí a kontextově závislou aplikaci jako:

> „Kontext je jakákoliv informace, která může být použita k charakterizaci situace subjektu. Subjektem je osoba, místo nebo objekt, který je považován za relativní pro interakci mezi uživatelem a aplikací, včetně uživatele a aplikace samotné. Systém je závislý na kontextu (context aware), pokud využívá kontext k poskytnutí relevantních informací nebo služeb uživateli, kde relevance závisí na úkolu uživatele. Povědomí o kontextu (context awareness) je možnost vytvořit kontext. Kontextové aplikace (Context aware applications) se adaptují podle místa použití, kolekcí lidí, zařízení a dostupných prostředků, a jejich změny v čase. Aplikace se zabývá prostředím a reaguje na změny.“

Druhou definicí je od Chen a Kotz [17] definují kontext dle rozdílu, co je relevantní a co je kritické.

> „Kontext je soubor stavů v prostředí a nastavení, která buď určuje chování aplikace, nebo ve kterém dojde k události aplikace a je zajímavá pro uživatele.“

Obecně platí pro aplikace využívající kontext, že sledují různé parametry prostředí zahrnující lokaci, čas, světlo, hluk, připojení k internetu, fyzickou aktivitu, aktuální úkol, identitu, role, historie aktivit, typ zařízení a spoustu neomezených možností. Tyto parametry popisují aktuální kontext použití (context of use). Ten se skládá ze tří kategorií: uživatel, platforma a prostředí, kde uživatel přistupuje ke kontextově závislé aplikaci na platformě v určitém prostředí.

2.1.4.1 Rozdělení kontextu

Pro účely této práce rozdělím parametry kontextu do čtyř tříd dle Reena Hanumansetty [19]. Rozděluje parametry kontextu do tříd dle sad požadavků na systém pro podporu těchto kontextových parametrů. K jednotlivým třídám pak budou přiděleny požadavky pro vytvoření systému.

Klasifikace je členěna do dvou základních kategorií dle typu změn na statické a dynamické. Pokud se parametr změní během relace, tak je přidělen jako dynamický parametr. Jinak je kategorizován jako statický parametr. Příkladem statických parametrů je platforma, role a uživatelské preference. Platforma popisuje vlastnosti zařízení a způsob jak s ním uživatel komunikuje. Statické parametry můžeme vyřešit během fáze vývoje designu uživatelského rozhraní, na rozdíl od dynamických, které vyžadují adaptaci uživatelského rozhraní za běhu. Příkladem jsou parametry lokace, rychlosti internetu a času.

Další úrovní klasifikace je dělení na funkcionální a prezentační. Do první kategorie patří parametry měnící funkčnost backendu. Například parametr role, který ovlivňuje zda uživatel může použít danou funkcionalitu nebo ne. Do další kategorie patří parametry měnící front end. Příklady této kategorie jsou platforma, lokace a uživatelské preference.

Tabulka 2.1 zobrazuje příklad obecného rozložení parametrů do tříd.

	Statické	Dynamické
Funkcionální	Role, identita, platforma, preference	Připojení k internetu, čas, lokace
Prezentační	Platforma, preference	Osvětlení, aktivita uživatele, orientace zařízení

Tabulka 2.1: Asociace kontextových parametrů do jednotlivých tříd

2.2 Aspektově-orientovaný přístup

Aspektově-orientovaný přístup (AOP) [7] je programovací paradigma, které řeší oddělení zodpovědnosti (*concerns*)[1]. Tím se rozumí rozdělení počítačového programu na různé části tak, aby se z hlediska funkcionality tyto části co možná nejméně překrývaly.

Na rozdíl od OOP, které pomáhá oddělení zodpovědnosti na základě implementace jednotlivých tříd, je AOP více zaměřené na oddělování zodpovědnosti než OOP. Řeší totiž problematiku takzvaných průřezových zodpovědností (*cross-cutting concerns*).

[1] Zodpovědnost je jakákoliv funkcionalita nebo chování programu.

Průřezové zodpovědnosti jsou takové zodpovědnosti, které se nedají jednoduše zapouzdřit a vyskytují se na více místech programu. Jedná se například o řízení transakcí nebo o logování průběhu programu. AOP se snaží řešit zapouzdření těchto průřezových zodpovědností a zároveň rozdělit program na jednotlivé moduly tzv. aspekty.

2.2.1 Koncept

Klíčovým konceptem AOP, který řeší rozdělení zodpovědnosti v programu je Join point model (JPM). Tento model umožňuje definovat dynamickou strukturu průřezových zodpovědností nezávisle na objektově orientované hierarchii. JPM definuje několik důležitých pojmů. Jejich význam vysvětlím na příkladu s logováním, kdy před a po vykonání metody proběhne logování.

Join point je místo v programu, kam je možné aplikovat průřezovou zodpovědnost pomocí AOP. Může sem patřit volání metod, inicializace tříd, struktura tříd (jméno třídy, názvy a datové typy atributů a jejich anotace) apod. V našem příkladu se jedná o místa volání metod.

Advice je rozšiřující kód, který chceme použít k rozšíření stávajícího modelu. Příkladem je funkcionalita logování, která bude vykonána při vykonání nějaké metody. *Advice* můžeme použit v *Join pointech*.

Pointcut jedná se o místo v aplikaci, kde je potřeba aplikovat průřezovou zodpovědnost. Jedná se o výběr *Join pointů*, pro které je aplikován *Advice*. Pro příklad logování je *Pointcut* souborem *Join pointů* před vykonání metody a po vykonání metody.

Aspect je kombinací *Advice* a *Pointcut*, určuje tedy logiku, která je do aplikace vložena a místo, kde bude spuštěna. Přidání logovacího aspektu do aplikace uděláme tak, že definujeme *Pointcut* a vybereme *Advice* s funkcionalitou logování.

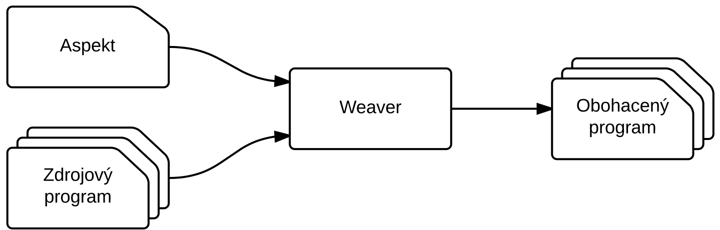
Weaving je proces (obrázek 2.1) integrace aspektů do specifických míst aplikace tak, že spojí aspekty a hierarchickou strukturu aplikace.

Velká část programovacích jazyků implementuje, nebo využívá knihovny pro aspektově orientované programování. Jednou ze známých implementací je *AspectJ*, která rozšiřuje jazyk Java o AOP. V této práci se zaměřím na jiný nástroj. Tím je *AspectFaces*, který podrobněji popíši v následující části kapitoly.

2.3 AspectFaces

AspectFaces (AF) [13, 14] je nástroj sloužící k automatickému generování fragmentů uživatelského rozhraní na základě meta-modelu získaného z inspekce rich entit[2] tak, aby se dosáhlo snížení množství ručně vytvořeného zdrojového kódu aplikace. K dosažení cíle AF

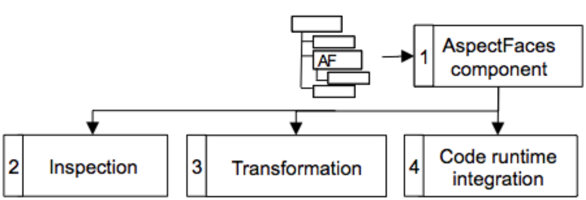
[2]Doménové entity rozšířené o informace prostřednictvím anotací.



Obrázek 2.1: Aspect weaving

implementuje techniku Rich entity aspect/audit design (READ)[15, 16], která je založená na aspektově orientovaném programování.

Hlavní problém, který AF řeší, jsou průřezové zodpovědnosti při vytváření uživatelského rozhraní. Při vytváření uživatelských formulářů narážíme na opakování kódu kvůli zodpovědnostem (concerns). Patří sem: propojení dat, prezentace vstupních polí, rozvržení, zabezpečení a validace vstupů. Všechny tyto zodpovědnosti AF dokáže separovat a spojit je dohromady do jednoho fragmentu. Obrázek 2.2 ukazuje tři základní komponenty AspectFaces, které dále v textu popíši. Jsou to komponenty: 2) inspekce, 3) transformace a 4) integraci.



Obrázek 2.2: Komponenty AspectFaces [13]

2.3.1 Inspekce kódu

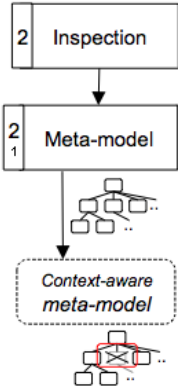
AspektFaces využívá inspekci zdrojového kódu k získání potřebných informací na sestavení meta-modelu, který bude reprezentovat strukturu, dle které bude možné později sestavit fragmenty uživatelského rozhraní.

Fáze inspekce kódu je znázorněna na obrázku 2.3. Tato fáze shromažďuje všechny dostupné informace, které jsou součástí datového modelu (entity, POJO, DTO, třídy). Z těchto modelů dokážeme získat informace o všech polích, která jsou součástí objektu a také o jejich omezeních (validace, bezpečnostní omezení, nebo vlastní anotace). V tomto kroku je sestaven meta-model reprezentující strukturu dat. V další fázi je potřeba vzít v úvahu kontext

aplikace, jako jsou uživatelské preference, role atd. a vytvořit kontextově závislý meta-model, ve kterém některá pole nejsou platná.

Momentálně existují dvě implementace inspekce. Jedna je schopná získat informace o datovém modelu z XML. Druhá implementace provádí inspekci Java tříd pomocí reflexe a anotací. Pro každou anotaci existuje tzv. Anotation Descriptor. Anotation Descriptor udává, jaké anotace jsou brány v potaz při inspekci entity a jak budou reprezentovány v meta-modelu. AF rozlišuje čtyři typy Descriptorů. Jejich výčet spolu s případy použití popisuje tabulka 2.2. Parametry vytvořené Anotation Descriptory jsou v meta-modelu svázány s jednotlivými atributy inspektované třídy. Pro každý atribut je v meta-modelu uložen jeho datový typ.

V meta-modelu mohou existovat i parametry nevázané na konkrétní atribut entity. To je například informace, zda celé vygenerované rozhraní má být pouze pro čtení. Další nezávislou informací jsou jména inspektovaných tříd.



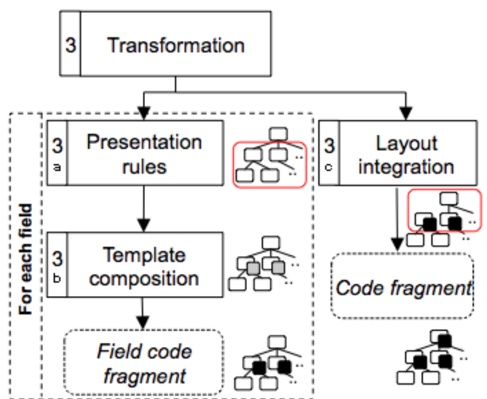
Obrázek 2.3: Fáze inspekce [13]

Typ Descriptoru	**Použití**
VarialbleJoinPoint	Přidává nové proměné do AF kontextu
OrderJoinPoint	Určuje pořadí elementů
EvaluableJoinPoint	Vyhodnocuje hodnotu anotace a pokud je hodnota vyhodnocená jako `false`, tak atribut bude ignorován.
SecurityJoinPoint	Podmíněné zobrazování na základě uživatelských rolí

Tabulka 2.2: Typy Annotation Descriptorů

2.3.2 Transformace

Fáze transformace je hlavní část celého procesu. Využívá aspektově orientovaný přístup k opětovnému použití částí programu a zároveň poskytuje konfigurovatelná a obecně transformační pravidla. Celá fáze transformace je znázorněna na obr. 2.4 a je složena ze tří fází.



Obrázek 2.4: Fáze transformace [13]

2.3.2.1 Prezentační pravidla

Prezentační pravidla nám udávají, jak budou jednotlivá pole z meta-modelu namapována v uživatelském rozhraní. Jinak řečeno, pravila určují jaká šablona bude použita pro reprezentaci datového pole v UI. Prezentační pravidla v AF korespondují s aspektem v AOP. Prezentační pravidla totiž určují, jaká šablona (představuje *Advice*) bude aplikována na vybrané místo dle pravidel (*Pointcut*).

Tato prezentační pravidla jsou v AF reprezentována pomocí konfigurace, ve které jsou jednotlivá pravidla specifikována dle datového typu atributu z meta-modelu a sady pravidel pro výběr šablony. Datovým typem se myslí jméno primitivního typu jazyka Java, nebo jméno třídy. Pravidla pro výběr šablony jsou psána pomocí Expression Language (EL) a mají přístup k meta-modelu a kontextu AF.

V příkladu kódu pro mapování 2.1 lze vidět mapování pro datový typ `String`. Ten se normálně mapuje na šablonu pro psaní textu. Když bude nad příslušným atributem v entitě umístěna anotace `@Email`, tak bude použita šablona pro vkládání emailu.

Příklad 2.1: Příklad pravidla mapování

```
<mapping>
        <type>String</type>
        <default tag="textTag.xhtml" size="20"
                javaPattern="" minLength="0" maxLength="255"/>
        <var name="Person.username" tag="emailTag.xhtml"/>
        <cond expr="${email==true}" tag="emailTag.xhtml"/>
        <cond expr="${link==true}" tag="linkTag.xhtml"/>
        <cond expr="${maxLength>255}" tag="textAreaTg.xhtml"/>
</mapping>
```

2.3.2.2 Kompozice šablon

Výsledkem prezentačního pravidla je výběr šablony. Šablona obsahuje fragment kódu pro widget, který je popsán pomocí DSL jazyka. V této fázi jsou jednotlivé šablony sestaveny dle dat z meta-modelu a kontextu. Tato data jsou pomocí EL zpracována a vložena do definovaných míst v šabloně. Tato místa jsou vyznačená znaky dolaru. Pokud například chceme vložit do šablony název pole, tak do šablony vložíme řetězec `$fieldName$`.

2.3.2.3 Rozvržení layoutu

V této poslední fázi transformace se sestaví výsledný fragment kódu. Jedná se o další aspekt v rámci AspectFaces. Jsou to šablony určené k organizaci vytvořených UI widgetů do layoutu.

2.3.3 Integrace

Poslední komponentou v AF je *Weaver*. Ten řeší způsob generování UI a integraci s aplikací. AF podporuje generování fragmentů jak *staticky*, tak i *dynamicky*. Statické generování probíhá během fáze vývoje. Na druhou stranu existuje dynamické generování během runtimu aplikace. Výhodou je přístup k aktuálnímu kontextu použití a díky tomu lze přizpůsobovat fragmenty potřebám uživatele.

Kapitola 3

Jazyky pro tvorbu UI

Účelem této kapitoly je seznámit se s jednotlivými jazyky a jejich vlastnostmi pro tvorbu uživatelských rozhraní.

3.1 User Interface Markup Language

User Interface Markup Language (UIML) [21] je jazyk k popisu uživatelského rozhraní nezávisle na cílové platformě. UIML klade důraz na oddělení zodpovědností interaktivních aplikací tak, aby migrace programu z jedné platformy na druhou vyžadovalo pouze malé nebo vůbec žádné změny. Tento jazyk je založen na základě jazyka XML. To přináší výhody v možnosti relativně snadného definování transformací z UIML do jiného jazyka. Tyto transformace přetransformují uživatelské rozhraní z abstraktní reprezentace popsané pomocí UIML na konkrétní reprezentaci cílové platformy (HTML, JSP, VoiceXML atd.). Nástroje postavené kolem UIML rozšiřují jazyk využíváním transformací tak, že poskytují vývojáři jeden jazyk k vytvoření uživatelských rozhraní, které bude fungovat na různých platformách.

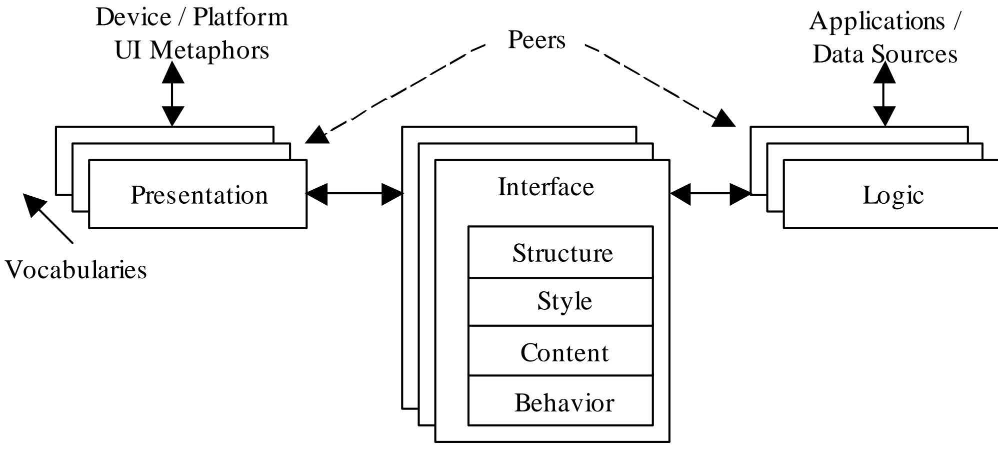
Protože jazyk je založen na XML, tak komponenty uživatelského rozhraní jsou reprezentovány pomocí tagů. Jazyk sám o sobě neobsahuje žádné specifické tagy závislé na platformě jako je `<window>` nebo `<menu>`. Místo toho se UIML spíše chová jako metajazyk tak, že sémanticky popisuje UI pro jiné jazyky. K tomu využívá pouze malý set mocných tagů jako je `<part>` popisující část UI, nebo `<property>` pro definování vlastnosti části UI. Tyto tagy jsou nezávislé na druhu UI (grafické, textové), cílovém zařízení (PC, telefon), nebo jazyku, do kterého bude UIML přetransformováno (HTML, Java Swing, VoiceXML).

Pro použití UIML je potřeba definovat slovník, ten obsahuje soubor tříd pro části UI a vlastností tříd. Slovník si definujeme dle potřeb pro vývoj. Pokud potřebujeme, lze vytvořit slovník, jehož třídy budou korespondovat 1:1 k souboru UI widgetů cílové platformy. Na druhou stranu lze definovat slovník s abstrakcí tříd (titulek, obsah, akce) tak, aby dle něho bylo možné popsat UI pro odlišné platformy.

3.1.1 Koncept

UIML je postaven na Meta-Interface modelu [27]. Tento model, který je vyobrazen na obrázku 3.1, rozděluje interface na tři hlavní části: *Presentation*, *logic* a *interface*. *Logic*

komponenta poskytuje UI jednoznačný způsob pro komunikaci s aplikací tím, že zapouzdřuje informace o protokolu, přenosu dat a názvu metod. *Presentation* komponenta poskytuje jednoznačnou cestu pro renderování UI tím, že zapouzdřuje informace o widgetech, jejich vlastnostech a zpracování událostí. Komponenta *Interface* popisuje dialog mezi uživatelem a aplikací pomocí sady abstraktních částí UI, událostí a volání metod, které jsou nezávislé na zařízení a aplikaci.



Obrázek 3.1: Meta-Interface Model [27]

3.1.2 Struktura dokumentu

Jak bylo řečeno UIML je postavené na jazyku XML. V této části popíši struktury dokumentu UIML (příklad 3.1) a jejich význam v popisu uživatelského rozhraní. Jak jde vidět z obrázku 3.1, struktura dokumentu koresponduje s Meta-Interface modelem.

Příklad 3.1: Struktura UIML dokumentu

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
"-//OASIS//DTD UIML 4.0 Draft//EN" "http://uiml.org/dtds/UIML4_0a.dtd">
<uiml xmlns='http://docs.oasis-open.org/uiml/ns/uiml4.0'>
       <head>...</head>
       <template>...</template>
       <interface>...</interface>
       <peers>...</peers>
</uiml>
```

interface Jedná se o hlavní komponentu v UIML. Slouží jako kontejner pro popis rozhraní zahrnující strukturu, obsah, styl a chování popsané pomocí čtyři elementů: *structure*, *style*, *content* a *behavior*.

- **structure** Popisuje logickou strukturu uživatelského rozhraní popsané hierarchicky pomocí elementů *part*. Každá část uživatelského rozhraní (element *part*) je definována jménem instance a jménem třídy. Jméno instance je unikátní a identifikuje část UI v dokumentu. Jméno třídy určuje třídu, do jaké patří element. Pro realizaci UI je důležitý slovník jmen tříd, protože dle jmen tříd budou jednotlivé části UI mapovány dle transformací na widgety cílové platformy.
- **style** Obsahuje seznam vlastností a hodnot, které jsou použity při sestavení UI. Tyto vlastnosti (např. font, barva, layout atd.) specifikují jakým způsobem bude rozhraní sestavené pro jednotlivá zařízení. Často je jeden *style* pro každý soubor nástrojů, na něž se UIML dokument přetransformuje.
- **content** Každá část UI může být asociována s několika typy obsahu, jako je text, zvuk, nebo obrázek. UIML separuje obsah od struktury UI. Tato separace dovoluje zobrazit různé druhy obsahu dle aktuálních okolností. Pro příklad UI může zobrazovat text v různých jazycích, nebo místo textu použít ikony.
- **behavior** Tento prvek popisuje chování interakce mezi uživatelem a uživatelským rozhraním. Pro příklad můžeme popsat, co se stane, když uživatel stiskne tlačítko. Zde také definujeme, kdy a jak UI vyvolá metodu z komponenty logic. Chování se popisuje pomocí sekvence proměnných a pravidel. Pravidla jsou složena z podmínek a seznamu akcí. Kdykoliv jsou podmínky platné, tak související akce se vykonají.

peers Popisuje transformace UIML dokumentu do cílových platforem. Transformace jsou prováděny dle jmen instancí a názvů tříd použitých v UIML dokumentu. Transformace jsou definované pomocí dvou druhů mapování. Používá k tomu tyto elementy: *presentation* a *logic*.

- **presentation** Definuje způsob mapování logických částí UI, událostí a vlastností do UI toolkitu. Toto mapování definuje slovník použitý v UIML dokumentu. Při běžném vytváření UI není potřeba pokaždé specifikovat toto mapování, ale můžeme použít již existující.
- **logic** Definuje, jakým způsobem bude UI komunikovat s aplikační logikou, která poskytuje funkcionalitu. Tento element se chová jako prostředník mezi UI a zbytkem aplikace. Popisuje způsob vyvolání metody v aplikační logice.

head Tento element obsahuje metadata o aktuálním UIML dokumentu (např. autor, datum, verze atd.). Obsah tohoto elementu nemá vliv na podobu a chování uživatelského rozhraní.

template UIML šablony umožňují rozvrhnout části uživatelského rozhraní na znovu použitelné části. Šablony dovolují:

- vložit jeden fragment do více míst v UIML dokumentu
- UIML dokument obsahuje fragment z jiného UIML dokumentu
- vnořovat do sebe styly a další prvky

3.2 XML User Interface Language

XML User Interface Language (XUL) [2] je jazyk založený na XML a slouží pro tvorbu multiplatformních grafických uživatelských rozhraní, které jsou používané v produktech Mozilla jako například Firefox či Thunderbird. Tento jazyk umožňuje vytvářet grafická uživatelská rozhraní GUI podobně jako webové stránky. XUL se totiž opírá o několik existujících webových standardů a technologií jako CSS, JavaScript a DOM, což činí XUL relativně snadný na naučení pro webové vývojáře, kteří již ovládají Dynamické HTML (DHTML). XUL není standardizovaný a proto existuje pouze jediná implementace interpreta, jedná se o vykreslovací jádro Gecko.

Cílem XULu je vytvářet multiplatformní aplikace na rozdíl od DHTML, které je zaměřené pro vývoj webových stránek. Z toho důvodu je vyjadřovací schopnost XULu orientovaná na aplikační artefakty, jako jsou okna, štítky a tlačítka, namísto stránek, nadpisů a hypertextových odkazů.

XUL využívá pro tvorbu multiplatformních aplikací několik technologií. Jsou to: XML, DTD, HTML, CSS, DOM, XBL, RDF a JavaScript.

3.2.1 Overlays

Overlays je mocný mechanizmus v XUL aplikacích. Tento nástroj umožňuje přizpůsobovat, nebo rozšiřovat stávající aplikace bez nutnosti zasahovat do originálního kódu. Tento nástroj například hojně využívají rozšíření prohlížeče Mozzila Firefox, kdy potřebujeme rozšířit UI o ovládající prvky. Overlays poskytuje obecné mechanizmy pro:

- Rozšíření stávající komponenty
- Přepsaní částí XUL souboru, bez nutnosti zasahovat do struktury UI
- Opětovné použití částí UI

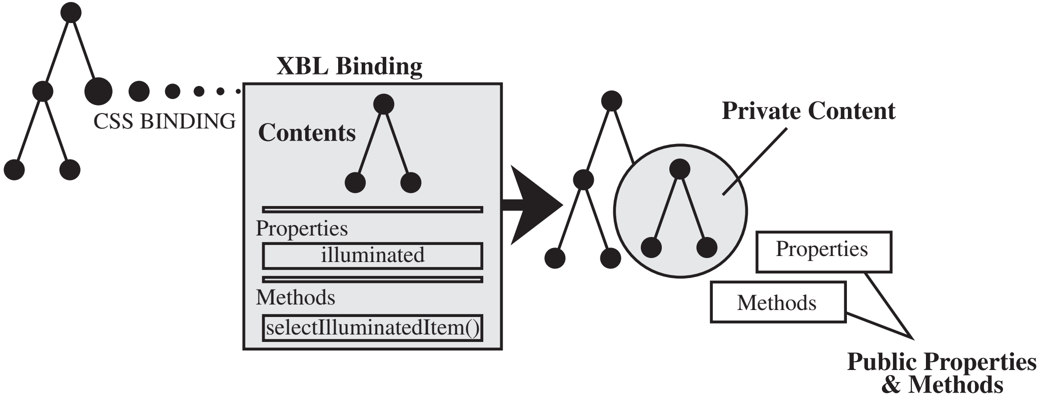
Jedná se o XUL soubory s definicí extra obsahu pro UI. Často se skládá z UI elementů, které budou přidány v důsledku aktualizace, nebo vylepšení. Overlays využívají dva způsoby pro transformaci UI. První způsob je velmi podobný jako u jiných jazyků *include*. Funguje tak, že použití Overlay je zahrnuto v XUL souboru. Druhý způsob je mocnější a umožňuje definovat Overlays externě, což dovoluje vývojáři upravovat existující aplikaci bez nutnosti zasahovat do originálního kódu.

3.2.2 XBL Binding

EXtensible Bindings Language (XBL) je XML jazyk, který umožňuje rozšiřovat XML elementy o anonymní obsah, vlastnosti a implementaci metod. XBL se používá k vytvoření znovu použitelných složených widgetů, které mohou být nadále využity v UI. K tomuto účelu tento jazyk využívá dvojici XUL a CSS, kde CSS slouží k navázání XUL elementu s XBL dokumentem.

XUL může využít XBL k zapouzdření nového obsahu, vlastností a metod. Toto je velmi užitečné při vývoji softwaru, kde vytvoříme komplexní widget tvořený XUL elementy. Díky

zapouzdření lze využívat nově vytvořené widgety bez znalosti jakým způsobem jsou widgety implementovány. Obrázek 3.2 znázorňuje příklad, jak lze využít XUL a XBL. Na obrázku je znázorněné propojení XBL dokumentu (který v sobě obsahuje nový obsah, vlastnost `illuminated` a metodu `selectedIlluminatedItem`) s konkrétním uzlem v XUL dokumentu. Zde se k výběru uzlu a k jeho napojení používá *CSS Binding*. Výsledkem je vytvoření nového složeného widgetu, který obsahuje privátní obsah a nabízí vývojářům prostřednictvím JavaScriptu přístup k vlastnostem a metodám.



Obrázek 3.2: XBL Binding [11]

3.2.3 XUL Template

XUL Template je nástroj pro generování dynamického obsahu UI, příkladem je dynamická tabulka. Dále se v textu budu o XUL Template zmiňovat jako o šablonách. Šablony produkují bloky UI z výsledků dotazu na datové zdroje. Analogií k dotazům v XUL jsou databázové dotazy. Pro každý výsledek z dotazu se vygeneruje konkrétní obsah. Syntaxe šablon dovoluje generovat odlišný obsah na základě specifikovaných pravidel, aktuálních kritérií a hodnot z výsledků dotazu. Šablony v XUL jsou účinným nástrojem pro propojení uživatelského rozhraní a business logiky aplikace.

Každá šablona získává data z datových zdrojů. Momentálně je podporováno několik typů datových zdrojů jako je RDF, XML a SQL databáze. Datové zdroje jsou v dokumentu identifikované pomocí URI. Pro příklad datový zdroj, který získává data z XML souboru, je definován pomocí URI adresy, která odkazuje na konkrétní XML soubor. Během sestavení šablony se nejdříve načtou data ze zdroje dat, dále se provede dotaz na datový zdroj a na základě výsledku v kombinaci se šablonou se vytvoří obsah.

XUL Template se skládá z dotazu a sérií pravidel. Dotaz obsahuje instrukce jak získat soubor dat z datového zdroje. Samotná syntaxe dotazů záleží na konkrétním typu datového zdroje. Série pravidel definuje generování obsahu na základě různých podmínek.

3.3 JavaServer Faces

JavaServer Faces (JSF) [12] je aplikační framework určený pro tvorbu webových uživatelských rozhraní. Je navržen tak, aby výrazně zmírnil zátěž při vývoji a údržbě aplikací, které běží na Java aplikačním serveru a vykresluje jejich uživatelská rozhraní zpět k cílovému klientu. JSF poskytuje následné způsoby pro usnadnění vývoje:

- Usnadňuje sestavit UI ze sady znovu použitelných UI komponent.
- Zjednodušuje migraci aplikačních dat mezi uživatelským rozhraním a aplikační logikou.
- Pomáhá řídit stav uživatelského rozhraní v rámci komunikace se serverem.
- Poskytuje jednoduchý model pro propojení klientsky generovaných událostí s aplikačním kódem na straně serveru.
- Umožňuje snadné postavení vlastních UI komponent a jejich znovupoužití.

Jedná se o Java standart pro tvorbu uživatelských rozhraní na bázi komponent a je součástí Java EE. V této práci se zabývám JSF verzí 2, která využívá Facelets jako základní templatovací systém. Facelets umožňuje využít jiné technologie popisující finální podobu UI, jako jsou: JSP, HTML5, XUL atd. Hlavní myšlenkou je sestavit uživatelské rozhraní na základě component-drive modelu. K tomu slouží speciální sada XML tagů, kterým jsou předávána data k zobrazení, nebo editaci ze standardních Java beanů.

3.3.1 Životní cyklus

JSF zpracovává HTTP požadavky pomocí šesti fází, ty jsou vyobrazeny na obrázku 3.3 a v textu dále je vysvětlen jejich význam. Běžný řídící tok je znázorněn plnými čarami, přerušované čáry znázorňují alternativní tok v závislosti na stavech jednotlivých fází.

Restore View fáze obnoví předchozí stav aplikace pro danou session, tj. znovu vytvoří strukturu reprezentující zobrazený pohled. Pokud požadovaný pohled není v session nalezen (např. při prvním zobrazení pohledu), přechází se rovnou na fázi render response, protože nemá smysl projít následujícími 4 kroky.

Apply Request Values v této fázi životního cyklu je každé komponentě aktualizován její aktuální stav z informací získaných z requestu (parametry, hlavičky, cookies atd.).

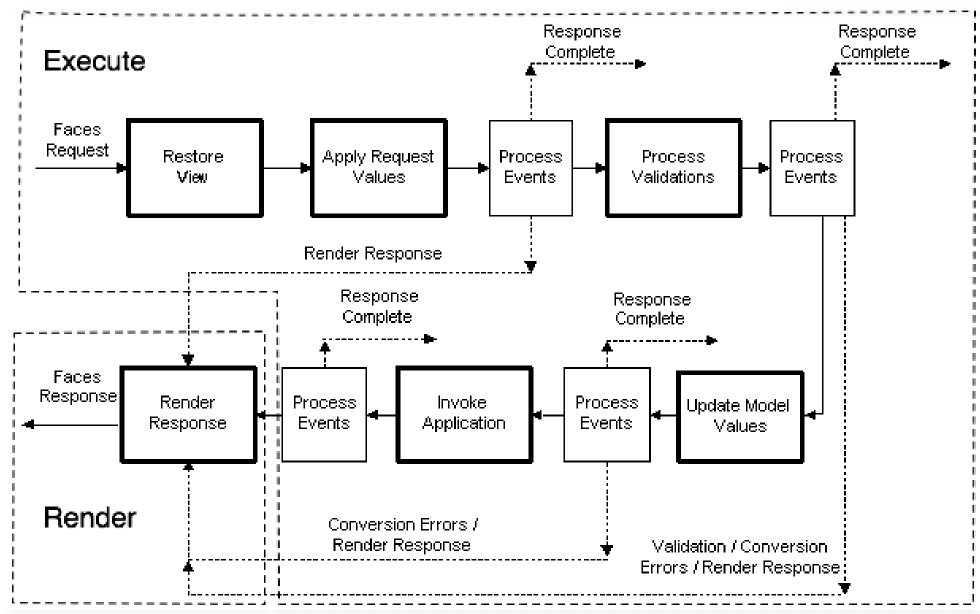
Process Validations fáze spustí registrované validátory na vstupní data. Při selhání validace se zaregistrují chybová hlášení pro jednotlivé komponenty a současný pohled se vyrenderuje znovu i s chybovými zprávami.

Update Model Values se spustí po úspěšné validaci vstupů. Tato fáze nastaví hodnoty JavaBeans, až v této fázi se dostávají vstupní data do naší aplikace. Nastavení se děje standardním mechanizmem voláním JavaBean setterů.

Invoke Application spustí požadované action metody a na základě výstupních klíčů se rozhodne o další navigaci na jiný pohled, nebo se přistoupí k znovu sestavení současného pohledu s novými daty.

Render Response zapíše do výstupního proudu HTTP Response vlastní HTML s daty z reprezentace tagu.

Celý životní cyklus zpracování požadavků je spravován pomocí servletu `FacesServlet`. Rozlišujeme dva druhy volání. První je *Initial requests*, kdy se při prvním zobrazení stránky provedou pouze fáze *Restore View* a *Render response*. Druhé je *Postbacks*, při kterém uživatel odeslal formulář na stránku a životní cyklus projde všechny fáze vývoje.



Obrázek 3.3: Životní cyklus JSF [12]

3.3.2 Component model

Komponenty uživatelského rozhraní jsou základním stavebním kamenem pro vytváření JSF uživatelských rozhraní. Jednotlivé komponenty reprezentují konfigurovatelné a znovupoužitelné elementy uživatelského rozhraní. Komponenty se mohou pohybovat v různých mezích složitosti od jednoduchých (jako je tlačítko, textové pole) až po složené (jako je tabulka). Komponenty mohou být volitelně spojeny s odpovídajícími objekty z datového modelu aplikace, pomocí EL.

JSF umožňuje komponentám uživatelského rozhraní využívat několik dodatečných API:

Konvertory jsou zásuvné třídy určené k převodu lokálních hodnot komponent na výstupní hodnoty požadovaných datových typů, a obráceně.

Eventy a listenery jedná se o API určené pro vysílání a odposlouchávání událostí pomocí listenerů. API je postaveno tak, aby podporoval JavaBeans specifikace.

Validátory jsou zásuvné třídy, které zkoumají hodnoty komponent (získané z příchozího dotazu) a ujišťují se, že tyto hodnoty jsou v souladu s business pravidly. Chybová hlášení při selhání validace mohou být vygenerována a poslána zpět uživateli během fáze vykreslení UI.

Uživatelské rozhraní konkrétní stránky je postaveno na základě spojování komponent uživatelského rozhraní v závislosti na konkrétním požadavku klienta. Pohled je tvořen stromem tříd, které implementují `UIComponent`. Komponenty ve stromu jsou ve vztahu rodič-dítě k ostatním komponentům. Kořenový prvek stromu musí být instance `UIViewRoot`.

3.3.3 Facelets

Facelets je open source webový framework, který slouží k tvorbě prezentační vrstvy ve webových aplikacích postavených na JavaServer Faces. Od verze JSF 2.0 plně nahrazuje technologii JavaServer Pages (JSP), která původně v JSF tuto funkci plnila. Facelets jsou založeny na standardu XML. Tato skutečnost znamená mimo jiné významné zkrácení doby potřebné pro zvládnutí JSF frameworku. Facelets nabízí tyto vlastnosti:

Konverze elementů ve Facelets lze zadávat elementy z knihovny značek ve dvou formách: jako kvalifikovaný XML element nebo prostřednictvím atributu `jsfc` na libovolném nekvalifikovaném prvku. V druhém případě Facelet kompilátor bude ignorovat aktuální element a zpracuje ho jako element uvedený v atributu `jsfc`.

Šablony Facelets poskytuje možnost využití šablon. Facelets soubor může odkazovat na hlavní šablonu a poskytovat ji obsah pro vyhrazená místa definovaná v šabloně. Soubor, který odkazuje na takovou šablonu, se nazývá klient šablony. Samotný klient šablony může být znovu použit jako šablona pro jiné klienty a tak může být vytvořena hierarchie šablon.

Znovupoužitelnost obsahu Kromě šablon, Facelets poskytuje podporu pro znovu použití fragmentů tím, že umožní zahrnout obsah, který se nachází v jiném souboru. Vložení takového obsahu může být provedeno třemi způsoby:

1. **Odkazováním na soubor** – jedná se o jednoduchý způsob, který přímo vloží obsah odkazovaného souboru na konkrétní místo.
2. **Vlastní značky** – lze vložit obsah nepřímo pomocí vlastních značek. Takové značky mohou být spojeny s Facelet v taglib souboru. Výskyt tohoto tagu se pak nahradí obsahem, který je definovaný ve Facelets.
3. **Kompozice komponent** – Facelets poskytuje mechanizmus kompozice komponent, který vytváří obsah skládáním komponent. Výsledný obsah je dostupný jako JSF komponenta.

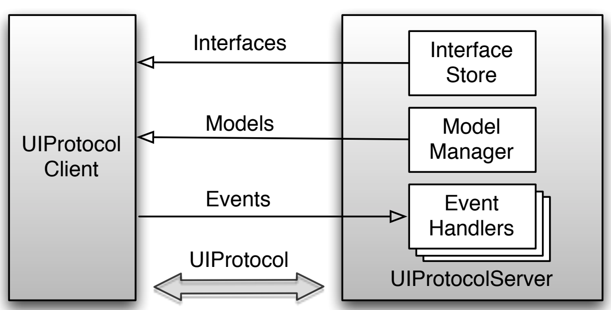
Parametrizované vkládání Facelets umožňuje upravit vkládaný obsah prostřednictvím parametrů. Zde parametry slouží jako proměnné předávající vlastnosti nebo obsah pro modifikaci vkládaného obsahu. Parametrizované vkládání také slouží pro definování rozhraní nových komponent vytvořených pomocí kompozice.

3.4 User interface protocol

Popis jazyka User interface protocol (UIP) vychází z analýzy provedené v práci Jindřicha Baška [9].

Jedná se o technologii určenou pro vytváření uživatelských rozhraní. UIP je navržen, aby poskytoval efektivní řešení pro tvorbu aplikací typu klient-server s různorodými klientskými platformami. Hlavní myšlenkou UIP je snížit nároky na vývoj uživatelského rozhraní pro více platforem tak, že poskytuje prostředky pro vývoj uživatelského rozhraní nezávisle na konkrétní platformě klienta. UIP používá architekturu client-server, kde klient komunikuje prostřednictvím uživatelské rozhraní s uživatelem a komunikuje se serverem pomocí UIP. Server obstarává logiku aplikace a poskytuje klientovi uživatelské rozhraní popsané pomocí DSL jazyka UIP, který je postavený na XML.

UIP není jen jazyk pro popis uživatelského rozhraní, ale také definuje protokol komunikace mezi klientem a serverem, jak je vidět na obrázku 3.4 zobrazující komunikaci. UIP klient odesílá na server pouze *Events* (události), které jsou na straně serveru zpracovány aplikační logikou. V opačném směru UIP server odesílá klientovi *Interfaces* (popis struktury uživatelského rozhraní) a *Models* (obvykle data, která jsou zobrazována prostřednictvím uživatelského rozhraní).



Obrázek 3.4: UIP komunikace [23]

UIP používá několik typů objektů pro komunikaci mezi serverem a klientem:

Interfaces hierarchický popis struktury uživatelského rozhraní, prvků uživatelského rozhraní a jejich vzhledu.

Models data používaná v UIP aplikaci. Data jsou oddělena od uživatelského rozhraní a mohou být dynamicky měněna a aktualizována.

Events události vyvolané uživatelem, tj. kliknutí myší na tlačítko, stisk klávesy atd., nebo klientem tj. žádost o model, uživatelské rozhraní, akci, chyba atd.

Actions modifikace modelu nebo uživatelského rozhraní, aniž by byla uživatelem vyvolaná událost zpracována aplikační logikou.

Tyto objekty jsou popsány pomocí XML elementů v UIP dokumentu, který je zasílán mezi klientem a serverem. Struktura UIP dokumentu je znázorněna v příkladu 3.2.

Příklad 3.2: Struktura UIP dokumentu

```
<?xml version="1.0" encoding="UTF-8"?>
<UIProtocol version="1.0" time="1234567890">
        <interfaces>
                <!-- interfaces definition here -->
        </interfaces>
        <models>
                <!-- models definition here -->
        </models>
        <events>
                <!-- event notifications here -->
        </events>
        <actions>
                <!-- actions definition here -->
        </actions>
</UIProtocol>
```

3.4.1 Vlastnosti

Základním kamenem UIP jsou tzv. vlastnosti *Properties*. *Properties* se používají k popisu vlastností elementů uživatelských rozhraní (např. vzhledu, pozice). Dále slouží k uložení dat v modelech, pro přenos dat v událostech a k popisu měněných dat v akcích. Vlastnosti mohou odkazovat na jiné vlastnosti pomocí klíče.

3.4.2 Modely

Data aplikace jsou v UIP ukládána v tzv. modelech (*Models*). Každý model má svůj název, který musí být specifický v rámci UIP aplikace. Modely obsahují jednotlivé vlastnosti, v nichž jsou uložena data aplikace.

Jeden model může existovat ve více variantách. Varianty jednoho modelu mají stejný název, ale odlišují se pomocí tzv. vlastností variant (*Variant Properties*). *Variant Property* je běžná vlastnost, která musí mít specifikovanou hodnotu a název a nesmí obsahovat klíč na jinou vlastnost. Varianty modelů mohou být použity např. při lokalizaci aplikace, kde máme několik variant modelu pro různé jazyky.

3.4.3 Uživatelská rozhraní

UIP je původně koncipované pro popis konkrétního uživatelského rozhraní tak, že popisuje konkrétní podobu UI pomocí komponent (tlačítka, textová pole, obrázky atd.) a jejich

vlastností jako je umístění nebo vzhled. Výsledné rozhraní lze vykreslit na konkrétním UIP klientovi. Podrobnosti o konkrétním uživatelském rozhraní jsou popsány ve specifikaci UIP [29].

UIP umožňuje definovat uživatelské rozhraní také s ohledem na kontext použití. K tomu používá proces automatického generování uživatelského rozhraní (obrázek 3.5), kde transformuje abstraktní uživatelské rozhraní na konkrétní uživatelské rozhraní dle kontextu. Abstraktní uživatelské rozhraní je funkční specifikace uživatelského rozhraní, která popisuje, co má být prezentováno uživateli. Neřeší však jakým způsobem. Proto je abstraktní uživatelské rozhraní hlavním vstupem pro automatické generování uživatelského rozhraní. Pro definování abstraktního uživatelského rozhraní vycházím z rané verze specifikace UIP pro abstraktní uživatelské rozhraní (AUIP) [23].

Abstraktní uživatelské rozhraní se skládá z těchto elementů:

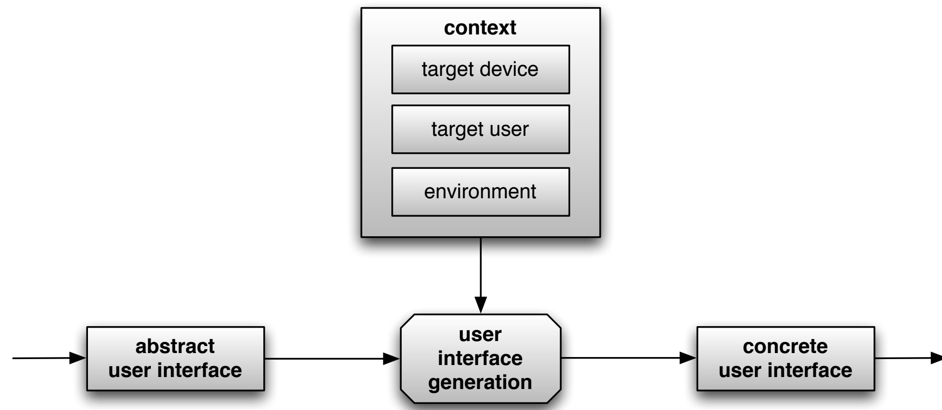
container element pro seskupení elementů do hierarchické struktury.

display element pro zobrazení informace. Není určený pro popis interakce.

trigger element pro definování interakce mezi uživatelem a aplikací.

input element pro vstupní data od uživatele.

Tyto elementy nespecifikují, jakým způsobem bude reprezentováno finální uživatelské rozhraní. Element obsahuje obecné vlastnosti jako je titulek, popis a ikona. Tyto vlastnosti mohou být použity UI generátorem pro vygenerování finální podoby komponenty (např. nemá smysl zobrazovat text pro uživatele, který neumí číst) v konkrétním uživatelském rozhraní.



Obrázek 3.5: Generování uživatelského rozhraní s ohledem na kontext [23]

3.4.4 Akce

Akce (*Actions*) slouží pro popis změn modelů, které může vykonat sám klient bez účasti aplikační logiky umístěné na serveru. Každá akce má jedinečný identifikátor.

3.4.5 Události

Události (*Events*) slouží pro zasílání informací o uživatelem vykonaných akcích. Události jsou jediným komunikačním objektem posílaným klientem. Server po přijetí události vykoná určitou akci dle typu události, která je definována pomocí `id`. Události mohou v sobě zasílat rozšiřující data ve formě vlastností, bez možnosti odkazování pomocí klíče na jinou vlastnost. Specifikace UIP definuje sadu několika standardních událostí pro obstarání základní funkcionality UIP aplikace. Jsou to např. události o stavu klienta, životní cyklus klienta, informování o chybách, připojení zařízení ke klientovi, žádosti o *Interface*, *Model* nebo *Action*.

3.4.6 Server

Na serveru jsou uložena veškerá data UIP aplikace. Klient se k serveru připojuje a tato data ze serveru získává. UIP server poskytuje pomocí TCP protokolu UIP dokumenty. Mediální soubory jako obrázky a zvuky jsou poskytovány pomocí HTTP protokolu. Server zpracovává příchozí události od klienta pomocí tzv. *Event handlers*. *Event handlers* jsou programy spouštěné na serveru při přijetí události. Tyto programy zpracují přijatou událost a odešlou klientovi zpět výsledky v podobě aktualizace modelu, nebo odesláním objektu *Model*, *Interface* nebo *Action*. *Event handlers* jsou programovány tvůrcem UIP aplikace.

3.4.7 Klient

Klient se stará o zobrazení uživatelského rozhraní a integraci s uživatelem. Se serverem komunikuje prostřednictvím zasílání událostí. Klienti se rozdělují pro UIP do čtyř tříd dle vlastností, které podporují. Jednotlivé třídy a jejich podporované vlastnosti jsou popsány ve specifikaci UIP [29]. V současnosti existuje několik implementací klientů pro různé platformy (iOS, .NET, PHP).

Kapitola 4

Návrh

V této kapitole navrhuji řešení na porovnání jednotlivých jazyků pro tvorbu uživatelských rozhraní. Za účelem testování navrhuji kontextově závislou aplikaci a pravidla pro adaptaci uživatelského rozhraní. Dle tohoto návrhu budou v následující kapitole vytvořeny adaptivní uživatelská rozhraní pomocí jednotlivých jazyků a frameworků z kapitoly 3.

4.1 Kontextově závislá aplikace

Pro potřeby této práce je potřeba navrhnout kontextově závislou aplikaci, která bude sloužit jako společný základ pro všechny porovnávané jazyky pro tvorbu adaptivních uživatelských rozhraní z kapitoly 3. Tato aplikace bude reagovat na změny v kontextu, které vyplynou ze sledování kontextových parametrů.

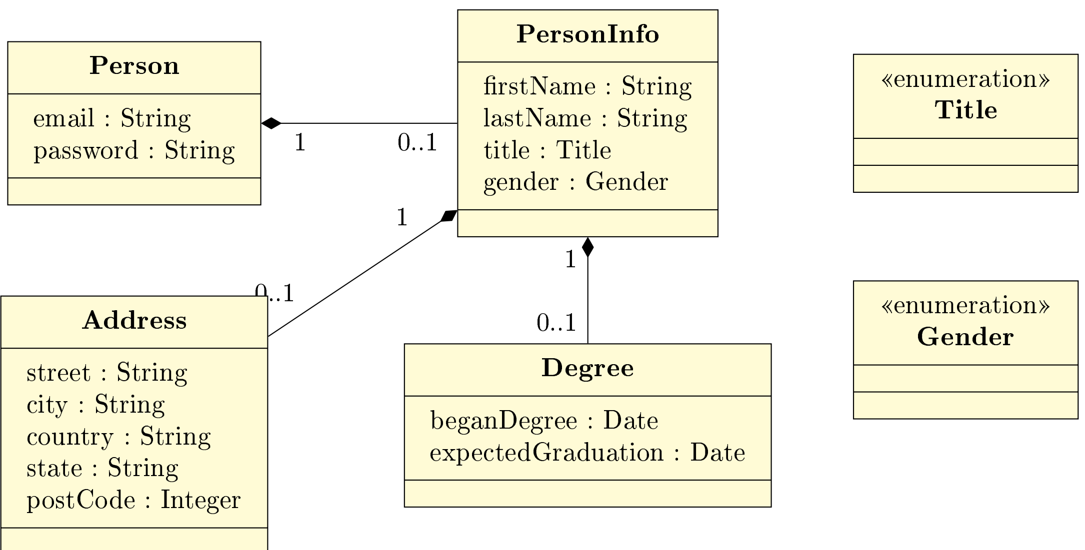
Návrh kontextové aplikace jsem rozdělil do tří částí, kde definuji požadavky a chování aplikace dle kontextu použití.

1. Definovat požadavky na aplikaci, design a doménový model.
2. Specifikovat pravidla pro adaptaci aplikace, kde pravidla jsou specifikována dle podmínek a akcí, které budou vykonány.
3. Identifikace kontextových parametrů, na které bude kontextová aplikace reagovat.

4.1.1 Požadavky

Testovací aplikace bude velmi jednoduchá ohledně množství funkcionality, ale zato bude podporovat různé druhy adaptace uživatelského rozhraní. Jedná se o aplikaci pro spravování seznamu osob a jejich informací jako je jméno, email, adresa, vzdělání atd. Aplikace umožňuje přidávat, odebírat a mazat uživatele. Dále poskytne možnost zobrazit výpis registrovaných uživatelů a detailní výpis informací o uživateli. Pro testovací účely bude aplikace obsahovat formulář umožňující měnit kontext použití za běhu aplikace. Doménový model aplikace, který je vyobrazen na obrázku 4.1, je jednoduchý a obsahuje pouze 4 entity reprezentující osobu a informace o ní.

Hlavním požadavkem na aplikaci je její adaptace dle aktuálního kontextu, tyto požadavky definuji v následujících sekcích.



Obrázek 4.1: Doménový model kontextové aplikace

4.1.2 Specifikace pravidel adaptace

Zde specifikuji pravidla, dle kterých se bude aplikace adaptovat. Pravidla jsou tvořena podmínkami a akcemi. Každá podmínka pravidla obsahuje nejméně jeden kontextový parametr, který v další fázi identifikuji v sekci 4.1.3. Tato pravidla budou v testovací aplikaci podporována:

RULE 1 Grafické UI
Podmínka: Uživatel je dítě.
Akce: Ovládací prvky a popisky formulářových polí jsou zobrazeny pomocí obrázků.

RULE 2 Nápověda
Podmínka: Uživatel je senior, nebo uživatel odeslal více jak 2x nevalidní formulář.
Akce: U všech formulářových polí se zobrazí textová nápověda.

RULE 3 Zobrazit formulář informace o studiu
Podmínka: Uživatel je student.
Akce: Zobrazí se formulář informace o studiu.

RULE 4 Adaptace formulářů dle lokace
Podmínka: Pokud aktuální lokace je v zemi USA.
Akce: Formulář pro vložení adresy bude obsahovat položku státu.

RULE 5 Předvyplnění země dle lokace

Podmínka: Lokace je známa.
Akce: Formuláře obsahují před vyplněné údaje o zemi dle lokace.

RULE 6 Mobilní UI

Podmínka: Typ zařízení je mobilní platforma, nebo velikost obrazovky zařízení je malá.
Akce: UI bude reprezentováno pomocí mobilní platformy.

RULE 7 Způsob zobrazení výpisu

Podmínka: Typ zařízení je mobilní.
Akce: Výpis bude reprezentován pomocí listu místo tabulky.

RULE 8 Lokalizace a internacionalizace

Podmínka: Preferovaný jazyk je znám.
Akce: UI bude lokalizováno dle preferovaného jazyka uživatele.

4.1.3 Kontextové parametry

Pro účely adaptace definuji několik kontextových parametrů, na které bude aplikace reagovat. Součástí návrhu specifikuji typ a povahu každého identifikovaného kontextového parametru. Kontextová aplikaci bude sledovat tyto kontextové parametry:

1. Věk uživatele (hodnoty: dítě, student, dospělý a senior)
2. Jazyk preferovaný uživatelem
3. Lokace
4. Velikost obrazovky
5. Typ zařízení (mobil, desktop)
6. Frekvence chyb uživatele

Pro integraci kontextu do aplikace je potřeba rozdělit jednotlivé kontextové parametry dle jejich vlastností. Jsou to vlastnosti: kontext, úrovně změn a povaha. Jednotlivé parametry rozděluji dle metody, kterou jsem popsal v kapitole 2.1.4.1. Důvodem proč rozděluji tyto parametry dle vlastností, je způsob jakým ovlivňují jednotlivé části aplikace. Vlastnost kontext nám říká, z jakého kontextu použití bude kontextová aplikace sbírat hodnoty parametru. Povaha parametru udává, kdy se mění hodnota parametru. Dynamické parametry se mění během relace, což je rozdílné od statických, které jsou během celé relace neměnné. Poslední vlastnost popisuje, v kterých částech aplikace se adaptace projevuje.

Výsledné rozdělení kontextových parametrů je popsané v tabulce 4.1.

Parametr	Kontext	Povaha	Úroveň
Věk	Uživatel	Dynamický	Funkční, Prezentační
Jazyk	Uživatel	Dynamický	Prezentační
Lokace	Prostředí	Dynamický	Funkční
Velikost obrazovky	Platforma	Statický	Prezentační
Typ zařízení	Platforma	Statický	Prezentační
Frekvence chyb	Uživatel	Dynamický	Funkční

Tabulka 4.1: Rozdělení kontextových parametrů

4.1.4 Architektura

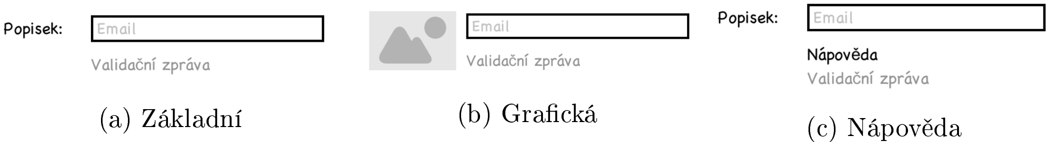
Aplikace je postavena na Java EE architektuře a využívá třívrstvý model MVC. Nejnižší vrstva aplikace se skládá z entit a kontextu použití. Z těchto entit pomocí inspekce bude nástroj AspectFaces získávat potřebné informace pro generování UI. Aplikační vrstva obsahuje kontrolory s aplikační logikou a kontextový framework. Poslední vrstvou je vrstva prezentační, ta obsahuje uživatelské rozhraní. V této vrstvě budu implementovat pro jednotlivé testované jazyky uživatelská rozhraní. Tato architektura přináší výhodu v odstranění závislosti aplikace na uživatelském rozhraní.

4.2 Adaptace fragmentů UI

V této sekci definuji, jak se bude adaptovat uživatelské rozhraní dle jednotlivých fragmentů reprezentující UI. Účelem je definovat jednotné chování adaptace uživatelského rozhraní tak, aby bylo totožné pro všechna rozhraní vytvořená pomocí jazyků z kapitoly 3. Způsob jakým uživatelská rozhraní dosáhnou těchto adaptací bude sloužit jako kritérium pro budoucí porovnání jazyků.

Jednotlivé adaptace navazují na předem specifikovaná pravidla ze sekce 4.1.2, kde jsem definoval kontextově závislou aplikaci. K jednotlivým fragmentům jsou přiloženy wireframes pro lepší představu podoby fragmentů UI.

Formulářové pole tento fragment se v základní podobě (obrázek 4.2a) skládá z popisku, vstupního pole a validační zprávy, která se zobrazuje pouze při nevalidním odeslání formuláře. Při aplikování pravidla 1 se změní způsob reprezentace popisku z textové podoby na grafickou (obrázek 4.2b). Druhá adaptace (dle pravidla 2) rozšíří formulářové pole o nápovědu (obrázek 4.2c).



(a) Základní (b) Grafická (c) Nápověda

Obrázek 4.2: Adaptace formulářových položek

Zobrazení dat tento fragment je podobný formulářovému poli, zobrazuje však data místo vstupního elementu. Jinak se tento element adaptuje stejným způsobem.

Výpis jedná se o fragment pro zobrazení výpisu dat. Tento element se adaptuje dle pravidla 7 na dva typy výpisů. První je seznam (obrázek 4.3b), kde jednotlivé položky výpisu budou zobrazeny jen jako seznam odkazů na podrobnosti o položce. Druhou formou je výpis pomocí klasické tabulky se sloupci (obrázek 4.3a).

Head 1	Head 2	Head 3
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6
Cell 7	Cell 8	Cell 9
Cell 10	Cell 11	Cell 12

(a) Tabulka

Cell 1 >
Cell 4 >
Cell 7 >
Cell 10 >

(b) Seznam

Obrázek 4.3: Adaptace výpisu dat

Formulář Struktura formuláře se bude adaptovat, podle aktuálního kontextu použití se zobrazí pouze relevantní formulářová pole. Konkrétně je tato adaptace vázána na pravidla 3 a 4. Druhá adaptace je založená na počtu položek formuláře, kde se při velkém množství položek adaptuje na formulář s průvodcem.

Navigace jedná se o fragmenty pro navigaci v aplikaci. Tyto fragmenty jsou adaptovány dle pravidla 6. Navigace se adaptuje změnou zobrazení a struktury menu a změnou tlačítek s akcemi.

Kapitola 5

Realizace

V této kapitole se zaměřím na popis samotné realizace kontextové aplikace a jednotlivých uživatelských rozhraní. Kapitola se skládá z několika částí. První část popisuje realizaci kontextové aplikace. V dalších částech se zaměřím na realizaci jednotlivých adaptivních uživatelských rozhraní.

5.1 Kontextová aplikace

Jedná se o Java EE[1] aplikaci, konkrétně Java EE 6. Tato aplikace obsahuje veškerou business logiku aplikace a nástroje pro kontextovou závislost. Při vývoji kontextové aplikace jsem se musel zabývat několika problémy. Tyto jednotlivé problémy a jejich řešení je popsáno níže.

5.1.1 Architektura

Bylo potřeba vytvořit takovou architekturu, která by nebyla závislá na uživatelském rozhraní. K tomu se velmi dobře hodí architektura MVC[2], která mi umožňuje oddělit uživatelské rozhraní od kontextové aplikace. Jak bylo výše naznačeno, jedná se o enterprise aplikaci složenou z několika modulů a zabalenou do EAR archivu (obrázek 5.1).

`EAR`
- `ejb` .... Business logika kontextově závislé aplikace a modely
- `aui-rules` .... Rule engine pro adaptivní UI
- `bundle` .... Texty a obrázky
- `aspectfaces-`{$jsf, uiml, uip, xul$} .... Uživatelské rozhraní pro konkrétní jazyk

Obrázek 5.1: Struktura aplikace

Prvním a nejdůležitějším modulem je `ejb`. Tento modul obsahuje kompletní business logiku aplikace a modely. Obsahuje také nástroje pro řešení kontextové závislosti, jejíž řešení

[1] Java platforma určená pro vývoj a provoz podnikových aplikací a informačních systémů. [22]

[2] MVC je softwarová architektura, která rozděluje datový model aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent tak, že modifikace některé z nich má jen minimální vliv na ostatní. [25]

popíši v sekci 5.1.3. Druhým důležitým modulem je modul `aui-rules`, ten obsahuje adaptivní pravidla a logiku pro jejich vyhodnocení. Další moduly aplikace obsahují implementaci jednotlivých adaptivních uživatelských rozhraní.

5.1.2 Datový model

K reprezentaci datového modelu v aplikaci slouží entity, které v sobě obsahují několik skupin metadat. Do první skupiny patří metadata pro prezsistenci dat. Konkrétně je použito Java Persistence API [18]. Další obsahují metadata specifikující validační omezení a metadata pro generování uživatelského rozhraní. Takové entity jsou označovány jako *rich entity*. Příkladem takové entity je v aplikaci entita `Person`.

Příklad 5.1: Entita Person

```
@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = "email"))
public class Person extends EntityObject {

    private String password;
    private String email;

    @OneToOne(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
    private PersonInfo personInfo;

    @NotNull
    @NotEmpty
    @UiPassword
    @UiOrder(2)
    public String getPassword() {
        return password;
    }

    @NotNull
    @NotEmpty
    @Email
    @UiType(value = "main")
    @UiOrder(1)
    public String getEmail() {
        return email;
    }

    ...
}
```

K specifikování validačních omezení je použito Bean validation API [10]. Bean validation rozšiřují JavaBeany pomocí anotací o metadata specifikující validační omezení. Pro příklad uvedu situaci, kdy entita `Person` obsahuje několik validačních omezení u atributu `email`. Jed-

ním omezením je povinnost atributu specifikovaná pomocí anotaci `@NotNull` a `@NotEmpty`. Dalším omezením je validace emailové adresy specifikovaná pomocí anotace `@Email`.

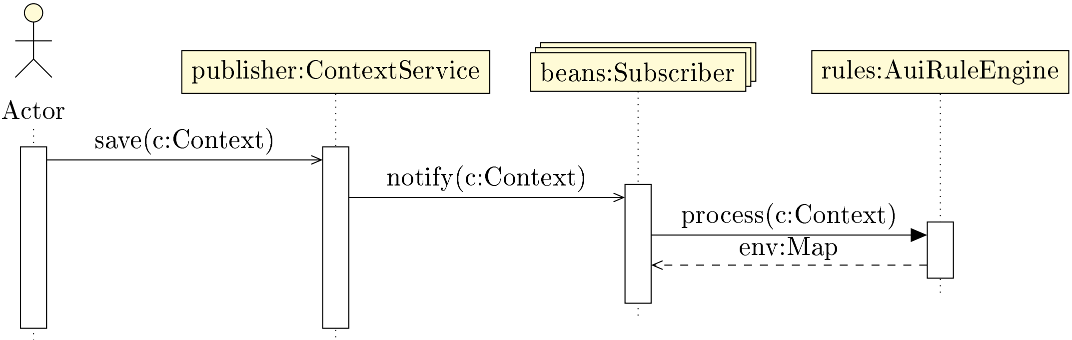
Pro účely využití aspektově-orientovaného přístupu pro generování uživatelského rozhraní, konkrétně využití nástroje AspectFaces, je potřeba rozšířit metadata entit o data potřebná pro generování UI. Patří sem data pořadí atributů `@UiOrder`, typ atributu `@UiType`, nebo podmíněné zobrazení dle uživatelské role `@UiUserRoles`. Tyto rich entity budou sloužit nástroji AspectFaces pro sestavení meta-modelu pomocí inspekce těchto entit.

5.1.3 Kontext

Jedná se o problém implementace a integrace kontextové závislosti do aplikace. Při implementování kontextové závislosti aplikace je potřeba vyřešit způsob, jak bude kontext implementován a jak bude aplikace reagovat na jeho změny.

Kontext reprezentuji pomocí bean třídy `Context`. Tato třída ukládá atributy kontextu, které jsem specifikoval v kapitole návrh kontextové aplikace 4.1.3. Pro správu aktuálního kontextu byla v aplikaci vytvořená služba `ContextService`. Tato služba má na starost spravovat aktuální kontext použití. To znamená, že uchovává aktuální kontext pro uživatele, umožňuje měnit kontext a distribuovat aktuální kontext do komponent závislých na kontextu. Jedná se o `Statefull` EJB vázanou na `SessionScope`. To umožňuje aby každý uživatel aplikace měl vlastní kontext použití.

Kontext lze distribuovat dvěma způsoby. První způsob je zažádáním komponenty o aktuální kontext. Druhý způsob je pomocí návrhového vzoru *Observer*, kde při změně kontextu informuji všechny pozorovatele o jeho změně. K tomuto účelu jsem využil technologii v Java EE a to CDI Events. Výhodou druhého způsobu je snížení provázanosti na `ContextService`. Nevýhodou druhé možnosti je její omezené použití, které jde použít jen na CDI bean. Sekvenční diagram 5.2 znázorňuje sekvenci akcí, které se vykonají při změně kontextu. Subscriber je kontextově závislá komponenta spravovaná CDI.



Obrázek 5.2: Sekvenční diagram kontextově závislé aplikace

5.1.4 Pravidla adaptace

Při realizaci adaptivních pravidel bylo zapotřebí implementovat pravidla adaptace navržené v kapitole 4.1.2. Při integraci těchto pravidel do aplikace jsem se rozhodl použít Java

Rule Engine API (JSR 94), což je API poskytující integraci Business Rule Management System (BRMS)[3]. V aplikaci využívám konkrétně implementaci Drools [1].

Důvodem proč jsem zvolil pro realizaci adaptivních pravidel technologii BRMS je to, že nabízí prostředky pro definování a vykonávání pravidel dle vstupních informací. Toto řešení také přináší několik výhod. Jednou z velkých výhod je definice pravidel pomocí DSL jazyka, konkrétně jazyk Drools Rule Language (DRL). Dále jsem docílil separace adaptivních pravidel na jedno místo a tím omezil zbytečné duplicity těchto pravidel, které by jinak vznikaly při vytváření několika uživatelských rozhraní.

V příkladu 5.2 je definované pravidlo 2 pomocí jazyka DRL. Toto pravidlo definuje, za jakých podmínek bude zobrazena nápověda v aplikaci. Pravidlo je složené ze dvou částí a jsou to podmínky a akce. Podmínky testují aktuální prostředí, které může obsahovat kontexty aplikace jako je kontext použití a AspectFaces kontext. Konkrétně v uvedeném příkladu podmínka testuje kontext použití (entita `Context`), zda uživatel je stařec, nebo zda vlastnost `invalid` je větší než 2. Pokud je tato podmínka splněná, tak se vykoná akce, která obohatí prostředí o aplikování nápovědy. Podobným způsobem jsou definována i zbylá pravidla z kapitoly 4.1.2.

Příklad 5.2: Implementace pravidla 2

```
global java.util.Map env;

rule "RULE 2: Help" when
    cz.cvut.fel.aui.model.Context(age == Age.ELDER || invalid > 2)
then
    env.put("applyHelp",true);
end
```

5.1.5 Realizace pravidel

Během návrhu kontextové aplikace byly rozděleny parametry kontextu z tabulky 4.1 dle úrovní ovlivňující chování kontextově závislé aplikace. Jedná se konkrétně o tři parametry kontextu: věk, lokace a frekvence chyb. Konkrétně se jedná o adaptivní pravidla 3, 4 a 5, která ovlivňují funkcionalitu aplikace.

K realizaci chování těchto adaptivních pravidel jsem se rozhodl použít nástroj AspectFaces, který umožňuje generovat fragmenty uživatelského rozhraní v závislosti na profilu uživatele a bezpečnostních omezení [15, 16]. Tyto vlastnosti se velmi hodí pro realizaci pravidel 3 a 4, které jsou založené na filtrování prvků formulářů. V aplikaci je toto chování realizováno pomocí obohacených *rich entit* o anotace popisující omezení na jednotlivých atributech. Anotaci `@UiUserRoles` užívám pro omezení vstupních polí dle kontextového parametru věku uživatele. V příkladu 5.3 tato anotace udává omezení pro zobrazení adresy pouze uživatelům věkových skupin: student, dospělý a senior. Anotace `@UiProfiles` udává

[3] Jedná se o softwarový systém umožňující spravovat a vykonávat rozhodovací logiky.

v jakých profilech bude konkrétní prvek zobrazen, kde je profil závislý na kontextovém parametru lokace. V příkladu je realizované pravidlo 4, které dovoluje zobrazit pole státu pouze, pokud je lokace uživatele ve Spojených státech.

Pravidlo 5 je odlišné od předchozích dvou pravidel, a to tím, že není založené na filtrování prvků formuláře, ale udává určité neobvyklé chování daného atributu. K tomu jsem opět použil nástroj AF, který tentokrát bude dle kombinace anotace **@UiType** a prezentačních pravidel mapovat atribut státu na speciální šablonu, která bude realizovat předvyplnění formulářového pole dle aktuální lokace.

Příklad 5.3: Vyžití AF anotací v rich entitách

```
@UiProfiles({"COUNTRY_US"})
public String getState();

@UiUserRoles({"student","adult","elder"})
public Address getAddress();

@UiType(value = "state")
public String getCountry();
```

Dále je potřeba vyřešit i propojení kontextu aplikace a kontextu AspectFaces. To jsem vyřešil pomocí adaptivních pravidel realizovaných pomocí rule enginu popsaném v kapitole 5.1.4. Realizace těchto pravidel je uvedena v příkladu 5.4. Vstupem pro pravidla je kontext aplikace a AF kontext, ten je dle kontextu aplikace nastaven a připraven pro budoucí využití při generování uživatelského rozhraní. Samotné využití nástroje AF je u jednotlivých UI trochu odlišné, proto budu popisovat využití tohoto nástroje až během popisu realizace jednotlivých UI.

Příklad 5.4: Pravidla pro nastavení AF kontextu

```
rule "RULE 3: AspectFaces roles" when
    cz.cvut.fel.aui.model.Context($age:age)
    ctx: com.codingcrayons.aspectfaces.configuration.Context()
then
    ctx.setRoles(new String[]{$age.name().toLowerCase()});
end

rule "RULE 4: AspectFaces profile COUNTRY_%" when
    cz.cvut.fel.aui.model.Context($country:country)
    ctx: com.codingcrayons.aspectfaces.configuration.Context()
then
    ctx.setProfiles(new String[]{"COUNTRY_"+$country});
end
```

5.2 JavaServer Faces

Tato kapitola popisuje realizaci uživatelského rozhraní pomocí JSF. Uživatelské rozhraní realizované touto technologií slouží jako referenční implementace adaptivního uživatelského rozhraní pro další realizace adaptivních UI v ostatních frameworcích. Hlavním důvodem proč je JSF jako referenční implementace je to, že je součástí Java EE specifikace, ve které je logika aplikace realizovaná.

Během realizace jsem narazil na omezení použití verzí JSF. Důvodem je kompatibilita AspectFaces pouze s verzí JSF 2.0, která umožňuje dynamické generování fragmetů UI. Z tohoto důvodu je realizace aplikace limitována pouze na Java EE 6 specifikaci. Proto nelze využít nové vlastnosti JSF 2.2, jako je podpora HTML5 nebo Resource Library Contracts.

Pro tvorbu uživatelského rozhraní využívám několik dodatečných nástrojů:

Twitter bootstrap jedná se o sadu CSS a JavaScript knihoven pro tvorbu responzivních webových stránek.

PrimeFaces jedná se o sadu UI komponent pro JSF.

5.2.1 Architektura

V této části popíši hlavní rysy realizace uživatelského rozhraní a jeho napojení na aplikační logiku. Napojení je řešené standardně pomocí backing bean, které se starají o zpracování dat z formulářů. Tyto beany pak využívají EJB obsahující logiku aplikace k vykonání akcí.

Dále bylo potřeba propojit JSF s kontextem aplikace a nástrojem pro řešení adaptivních pravidel. K tomu je v EL JSF registrovaná pod jménem `viewHelper` java beana, která zpřístupňuje aktuální konfiguraci adaptace. Využívá k tomu odchytávání eventů s aktuálním kontextem, který pak zpracuje pomocí `AuiRuleEngine` na výslednou konfiguraci adaptace.

Jedna z důležitých částí uživatelského rozhraní je hierarchie Facelets šablon tak, aby realizace konkrétní stránky byla pokud možno nejvíce nezávislá na cílovém zařízení (počítač, telefon). Celá hierarchie je složena ze tří vrstev. Nejnižší vrstva reprezentuje určitou stránku uživatelského rozhraní a ta poskytuje vyšší vrstvě obsah stránky. V druhé vrstvě je hlavní šablona, která zahrnuje obsah společný pro všechny stránky nezávisle na platformě. Hlavní šablona importuje konkrétní šablony s rozložením prvků UI dle cílové platformy z třetí vrstvy. Tento způsob částečně odstínil realizaci stránek UI od finálního rozložení prvků na cílové platformě. Neřeší však reprezentaci obsahu stránky pro různé platformy, které se mohou lišit v použití různých widgetů. Problém řeším pomocí kompozice komponent ve Facelets tak, že definuji vlastní sadu UI komponenty. Tyto komponenty mají několik realizací v separátních souborech, které jsou určené pro jednotlivé platformy. Pomocí těchto komponent definuji tlačítka, menu a výpisy.

5.2.2 Generování fragmentů

AspectFaces je integrován do JSF pomocí elementu `ui:af`, který se stará o dynamické generování fragmentů a jejich integraci do uživatelského rozhraní. Pomocí AspectFaces generuji tyto fragmenty z entit: formuláře, seznamy, tabulky a detaily. K tomu využívám tři sady AF šablon a k nim prezentační pravidla:

http - sada šablon určená pro desktopové UI. Obsahuje prvky pro generování formulářů a detailů.

mobile - sada šablon pro mobilní UI. Obsahuje prvky pro generování formulářů a detailů.

table - sada šablon pro generování seznamů a tabulek.

Pro generování fragmentů UI je potřeba napojit generátor AF tak, aby realizoval adaptaci UI popsanou v sekci 5.1.5. Veškeré nastavování AF se odehrává ve třídě[4], která je implementací handleru pro JSF tag `ui:af`. Nejdůležitější částí této třídy je metoda `hookAddToAFContext`, která nastavuje AF kontext, dle kterého se generují fragmenty UI. AF kontext nastavuji pomocí služby `AuiRuleEngine` (viz příklad 5.5), která vyžaduje kontext aplikace. Zde vzniká problém, jak přistupovat ke kontextu aplikace, který je přístupný přes CDI. Handlery tagů totiž nejsou spravovány CDI kontejnerem. Je několik možností jak předat kontext do handleru a to buď v parametru tagu, nebo existuje druhá možnost pomocí `BeanManageru`, kterou jsem zvolil.

Příklad 5.5: Nastavení AF kontextu

```
public class AdaptiveGeneratorHandler extends DefaultAFGeneratorHandler {
    protected void hookAddToAFContext(Context context) {
        cz.cvut.fel.aui.model.Context config = getContext();
        context.setLayout("desktop");

        try {
            getRuleEngine().process(context.getVariables(),config,context);
        } catch (Exception e) {
            logger.log(Level.SEVERE,e.getMessage(),e);
        }

        context.setLayout(applySettings(context.getLayout(), null));
    }
    ...
}
```

5.2.3 Adaptivní uživatelské rozhraní

V této sekci popíši jak je realizované adaptivní uživatelské rozhraní dle jednotlivých adaptací. Tyto adaptace vycházejí z kapitoly 4.2 návrhu adaptivního UI a popisuji je odděleně dle ovlivněných fragmentů UI. Hlavním účelem je popsat metodiky jednotlivých jazyků, kterými lze dosáhnout dané adaptace. Popřípadě popsat omezení jednotlivých jazyků a frameworků kvůli kterým nebylo možné docílit konkrétní adaptace.

[4] AdaptiveGeneratorHandler

Formulářová pole Popis chování adaptace formulářových polí se nachází v kapitole 4.2. Konkrétně se jedná o grafickou reprezentaci popisku formulářového pole a zobrazení nápovědy. Obě adaptace jsou řešeny pomocí skrývání obsahu. V příkladu 5.6 je znázorněna tato implementace pro zobrazování obrázku. Jeden z problémů takovéhoto řešení je v propagaci napříč uživatelským rozhraním. Zde byl tento problém vyřešen pomocí AF, kde obslužný kód pro adaptaci je obsažen v AF šablonách a tak je společně generován s formulářovými poli. Samotná logika adaptace testuje hodnoty z AF kontextu. Jsou to vlastnosti `applyImage` pro zobrazení obrázku a `applyHelp` pro zobrazení nápovědy.

Příklad 5.6: Realizace adaptace formulářového pole v JSF

```
// AF template
...
<ui:fragment rendered="$applyImage$">
    <h:graphicImage name="#{imageResources.getImage(id)}"
    alt="#{imageAlt}"/>
</ui:fragment>
...
<ui:fragment rendered="$applyHelp$">
    <p class="help">#{help}</p>
</ui:fragment>
...
```

Formuláře Adaptace struktury formulářů je řešena dynamickým generováním formulářů pomocí AF. Způsob využití AF pro adaptaci je popsán v sekci 5.1.5 a 5.2.2.

Adaptaci založenou na počtu prvků ve formuláři nebylo možné implementovat na generované formuláře. Původním předpokladem bylo využít AF, který by dle počtu prvků formuláře nebo počtu entit vybral vhodný layout. Toho bohužel nelze v AF docílit, protože chybí funkcionalita pro pokročilejší správu layoutů dle meta-modelu.

Mobilní UI Tuto adaptaci jsem se rozhodl implementovat dvěma způsoby a porovnat je. První použitá technika je pomocí responzivního designu, kde jsem použil knihovnu *Twitter bootstrap*[5]. Druhou možností je definování UI pomocí univerzální sady UI komponent. Konkrétně k tomuto účelu používám PrimeFaces, který má společnou sadu komponent, jak pro desktopovou verzi, tak i pro mobilní. O samotnou finální podobu se stará RenderKit.

Oba způsoby jsou postaveny na stejné myšlence a to definovat pouze jedno UI pro nejrůznější zařízení. Nevýhodou prvního řešení je to, že nejde použít při rozšíření aplikace o jinou platformu, jež není založena na HTML. Druhá možnost pomocí univerzální sady komponent tento problém řeší a to díky možnosti používat různých renderů v JSF. Příkladem je knihovna Xulfaces, která umožňuje vytvářet XUL uživatelská rozhraní pomocí JSF, viz v sekci 5.3.

[5] Jedná se o framework pro tvorbu responzivních webových stránek. Je založen na webových technologiích HTML, CSS a JavaScript

Výpis dat Adaptaci výpisu dat řeším pomocí vlastní speciální komponenty `aui:table`, pro kterou je generován obsah pomocí AF, viz příklad 5.7. Tato komponenta přepíná mezi implementacemi tabulky a listu dle konfigurace nastavení. Realizace tabulky je postavena na komponentě `h:dataTable`, pro kterou AF generuje sloupce. Realizace listu je řešená pomocí opakovaného volání AF generátoru pro každou položku seznamu. Definice speciální komponenty bez implementací se nachází v souboru `/webapp/resources/components/table.xml`. Samotné konkrétní implementace jsou v souborech `table.xml` a ty se nacházejí v podsložkách složky `/webapp/WEB-INF/components`, kde jsou rozdělené dle typu zařízení.

Příklad 5.7: Použití univerzální komponenty pro výpis dat

```
<aui:table value="#{personController.all}">
  <af:ui instance="#{item}" edit="false" config="table" layout="table.xhtml"/>
</aui:table>
```

Lokalizace Nástroje pro lokalizaci a internacionalizaci jsou součástí JSF. Pouze bylo potřeba pozměnit chování JSF tak, aby nenastavovalo automaticky lokalizaci, ale aby přebíral lokalizaci z kontextu. To jsem vyřešil nastavováním lokalizace na kořenovém prvku stromu.

5.3 XUL

Tato kapitola popisuje realizaci adaptivního uživatelského rozhraní pomocí jazyka XUL. Uživatelské rozhraní je vytvořeno jako XUL aplikace postavená na běhovém prostředí *XULRunner* [3]. Toto prostředí umožňuje integrovat Mozilla technologie a použít je v aplikaci. Dále aplikuji knihovnu *Xulfaces* [4], která přidává podporu XUL jazyka do Java Server Faces. Obsahuje XUL RenderKit, knihovnu s XUL tagy a odporu Facelets. Tuto knihovnu jsem se rozhodl využít na propojení uživatelského rozhraní s kontextovou aplikací.

5.3.1 Architektura

Uživatelské rozhraní je realizované jako klient, který přistupuje k serveru, na kterém běží aplikační logika kontextové aplikace. Klient v sobě obsahuje tyto části: XUL dokumenty, XBL dokumenty, javascript knihovny, CSS a obrázky ze kterých se skládá uživatelské rozhraní.

K propojení UI s kontextovou aplikací bylo možné použít několik způsobů. Jedna z možností je využít nástroje XUL Templates a RDF datové zdroje. Tuto možnost jsem nezvolil kvůli potřebě implementovat několik javascriptových knihoven určených pro komunikaci se serverem a také knihovny pro validaci dat. Druhou možností, kterou jsem nakonec zvolil je propojení pomocí knihovny Xulfaces. Jedná se o rozšíření JSF o podporu XUL uživatelských rozhraní. Toto řešení jsem se rozhodl použít, protože poskytuje už hotové knihovny řešící propojení XUL s Java EE aplikací. Zajišťuje odesílání formulářů, vykonávání akcí na straně serveru, validace hodnot formulářů a v neposlední řadě i způsob napojení nástroje AspectFaces. Veškerá komunikace probíhá odesíláním požadavků přes protokol HTTP. Na straně serveru jsou následně zpracovány FacesServletem a xulfaces, viz životní cyklus JSF v sekci

3.3.1. V samotné implementaci UI jsem využíval JSF minimálně (jen pro nezbytné účely) tak, aby použití co nejméně ovlivňovalo vytvářené UI.

Dále se zaměřím na samotnou strukturu uživatelského rozhraní. Nejprve popíši klientskou aplikaci a její architekturu. Uživatelské rozhraní je tvořené z oken a dialogů, kde existuje hlavní okno, které se stará o navigaci a zobrazení obsahu dle layoutu. Za účelem vytvoření i mobilní podoby UI jsem vytvořil dvě takováto okna aplikace (desktopová verze `root.xml` a mobilní verze `mobile.xul`). Každá řeší jinak rozložení prvků a navigaci. Do těchto oken je pak vkládán obsah z XUL dokumentů pomocí nástroje Overlays[6]. Hlavní obsah stránek se vkládá do elementu identifikovaného jako `content`.

Jak bylo výše napsáno uživatelské rozhraní je spouštěné přes XULRunner a toto prostředí má definovanou architekturu. Struktura aplikace je popsaná v příloze C.1.

5.3.2 Generování fragmentů

Generování fragmentů je prováděné dynamicky pomocí AspectFaces. AF je integrované pomocí JSF a pomocí tagu `ui:af`. Na rozdíl od JSF implementace není AspectFaces přímo integovaný s UI, ale pouze generuje Overlay fragmenty. Tyto fragmenty jsou dostupné na url adresách, které jsou vypsané v tabulce 5.1.

Adresa	Fragment
/aspectfaces-xul/overaly/context.xul	Formulář kontextu aplikace
/aspectfaces-xul/overaly/people.xul	Výpis všech osob
/aspectfaces-xul/overaly/person.xul	Detail konkrétní osoby
/aspectfaces-xul/overaly/registration.xul	Formulář registrace

Tabulka 5.1: Generované UI fragmenty pro XUL aplikaci

Tyto fragmenty jsou následně pomocí XUL Overlay integrovány do uživatelského rozhraní. Napojení generátoru AF je řešené stejným způsobem jako v implementaci JSF popsané v sekci 5.2.2. Generovány jsou tyto fragmenty: formuláře, seznamy a detaily. K tomu je potřeba použít dvě sady AF šablon a prezentačních pravidel:

xul - pro generování formulářů a detailů. Zde je rozdíl oproti implementaci pomocí JSF, kde byla separátní konfigurace pro mobilní UI.

table - sada šablon pro generování seznamů a tabulek.

5.3.3 Adaptivní uživatelské rozhraní

UI se adaptuje při změně konfigurace adaptace, která je přístupná pro jednotlivé relace klienta na adrese `/aspectfaces-xul/rdf/adaptation`. Tato konfigurace je tvořena výstupem ze služby `AuiRuleEngine` a zpracována pomocí javascript modulu `Aui.jsm`. Ten, při získání nové konfigurace, vykoná funkci `handleConfig` (př. 5.8), ve které se děje veškerá adaptace na straně klienta. Problém nastal při získávání aktuální konfigurace ze serveru.

[6] Viz v podkapitole 3.2.1

Nešlo totiž použít technologii *WebSocket*[7] kvůli omezení AF na Java EE 6 specifikaci. Proto se musí uživatelské rozhraní stále dotazovat na nastavení adaptace, místo toho, aby klientské aplikaci byla konfigurace zasílána automaticky.

Příklad 5.8: Obsluha adaptace UI

```
Aui.handleConfig = function (config) {

    // registrace XBL Bindingu dle konfigurace
    if (config["applyHelp"]) {
        Binding.regist(helpBinding);
    } else {
        Binding.unregist(helpBinding);
    }
    ...
}
```

Formulářová pole Tato adaptace je řešená pomocí technologie XBL Binding. Myšlenka řešení je taková, že obohatíme konkrétní prvky dokumentu o anonymní obsah z XBL dokumentu. Ten lze velmi jednoduše aplikovat pomocí CSS Bindingu tak, že pomocí CSS selektorů určíme, na jaké elementy dokumentu se aplikuje XBL. To má obrovské výhody v možnosti široké aplikace změn napříč UI bez minimálních zásahů do kódu. Stačí přidat jeden CSS styl a adaptace se aplikuje napříč celým UI. Dodatečná data, jako je konkrétní obrázek nebo text nápovědy, jsou předávány prostřednictvím parametrů elementu. Na příkladu 5.9 uvádím, jak jsem takovouto adaptaci implementoval napříč aplikací. Pomocí této techniky jsou implementovaná pravidla 1 a 2.

Příklad 5.9: Využití XBL Binding pro obohacení elementu o obrázek

```
// CSS binding
.field[image] {
    -moz-binding: url('chrome://aui/skin/images.xml#field');
}

// XBL dokument
<binding id="field">
        <content>
                <xul:box>
                        <xul:image xbl:inherits="src=image"/>
                </xul:box>
                <children/>
        </content>
</binding>
```

[7] Jedná se o protokol poskytující full-duplex komunikační kanál na jediném TCP spojení.

```
// XUL dokument
<hbox class="field" image="img.png" >
       <label />
       <textbox />
</hbox>
```

Formuláře Struktura formuláře se adaptuje díky AF dynamickému generování, viz 5.1.5. Adaptace formuláře dle počtu jeho prvků nebylo možné jednoduše implementovat na generované formuláře. Původním předpokladem bylo využít AF, který by dle počtu prvků formuláře nebo počtu entit vybral vhodný layout. To bohužel nelze v AF docílit, protože chybí funkcionalita pro pokročilejší správu layoutů dle meta-modelu. Pomocí XUL jsem dosáhl adaptace formuláře mezi oběma reprezentacemi: běžný formulář nebo wizard[8]. Podobu výsledného formuláře upravuji pouhou změnou typu dialogového okna.

Mobilní UI Dle pravidla 6 adaptuji rozvržení prvků uživatelského rozhraní pomocí dvou hlavních oken, viz popis architektury 5.3.1. Tato okna definují layout, do kterého bude vkládán spojený obsah pomocí Overlay. Řešil jsem i různé druhy chování UI dle typu zařízení. U desktopové verze je stále otevřené hlavní okno aplikace a při akcích se otevírají dialogová okna. Na druhou stranu mobilní verze nevyužívá dialogová okna, ale vše obstarává hlavní okno.

Způsob jakým jsou jednotlivé widgety prezentovány, určuje běhové prostředí. Toto prostředí v sobě obsahuje sadu XBL dokumentů definující konkrétní podobu a chování widgetu. Příkladem je vytváření doplňků pro aplikaci Mozilla Firefox, které jsou dostupné, jak pro mobilní, tak i desktopovou aplikaci. XUL má i předpoklady pro možnost nasazení responzivního designu, díky CSS a podpoře *Media Queries* (př. 5.10). S kombinací Media Queries a XBL Binding lze velmi flexibilně upravovat strukturu UI.

Příklad 5.10: Responzivní design XUL rozhraní

```
@media (max-width: 320px) {
    /* modifikace box layoutu */
    .field {
        -moz-box-orient: vertical;
    }
    #my-listbox listheader {
        display: none;
    }
    #my-listbox listcell:not(.main){
        display: none;
    }
}
```

[8] Jedná se o typ uživatelského rozhraní, které vede uživatele k uskutečnění komplexního úkolu pomocí sekvence přesně definovaných kroků.

Výpis dat Adaptace tabulky na seznam je v XUL velmi jednoduchá. K reprezentaci tabulky i seznamu slouží stejný widget (`listbox`). Jednotlivé reprezentace se liší dle vkládaného obsahu, který generuji pomocí AF. Pro účely seznamu se vygeneruje pouze seznam položek. V druhém případě se vygeneruje hlavička a obsah s několika sloupci. Další možností je vždy vygenerovat tabulku a její obsah redukovat jen pomocí CSS, viz příklad 5.10.

Lokalizace Nástroje pro lokalizaci a internacionalizaci jsou součástí XUL. První možnost je nejjednodušší, je to lokalizace UI pomocí DTD (příklad použití 5.11). Tyto soubory jsou automaticky importovány podle chrome manifestu[9]. Druhou možností je lokalizace pomocí javascript API. To na rozdíl od DTD umožňuje parametrizovanou lokalizaci. Internacionalizace směru psaní (př. zprava do leva) je řešená stejným způsobem jako u HTML pomocí atributu dir.

Příklad 5.11: Lokalizace v XUL

```
<!-- DTD entita lokalizace -->
<!ENTITY delete "Smazat" >

<!-- lokalizace v XUL dokumentu -->
<botton label="&delete;"/>
```

5.4 UIML

V této kapitole se zaobírám realizací adaptivního uživatelského rozhraní pomocí jazyka UIML. Při realizaci nebylo možné dosáhnout funkčního prototypu UI. Základní problém nastal v nevyhovujících interpretech. Nepodařilo se mi získat jakéhokoliv interpreta, který by dokázal realizovat adaptivní UI. Volně dostupný interpret, který existuje v době realizace této práce, je pouze interpret *jUIML* a interpret *UIML.NET*. Ani jeden z interpretů neimplementuje kompletní specifikaci UIML, která je potřebná pro vytvoření adaptivního UI. Vyřešením tohoto zásadního problému by bylo vytvoření vlastního interpreta. Což je nad rámec této práce. Z těchto důvodů je realizace UIML UI nastíněna pouze jako návrh možného řešení, které se bude opírat o dvě specifikace. První je specifikace UIML verze 4.0 [21] a druhá je specifikace generického slovníku [20].

5.4.1 Architektura

Pro vývoj pomocí UIML je nejprve potřeba si zvolit slovník, kterým bude UI definované. Rozhodl jsem se použít generický slovník [20] od společnosti *Harmonia*. Tento slovník je sestaven tak, aby byl nezávislý na platformě a jazyku. Všechny specifikované třídy tohoto slovníku jsou tvořeny prefixem `G` a jménem třídy (př. `G:Button` – obecné tlačítko, `g:title` – vlastnost titulek). Cíle generického slovníku:

[9] Více v příloze C.1

- Definuje generickou sadu widgetů, reprezentující nejběžnější používané ovládací prvky, které mohou být mapované na komplexnější ovládací prvky.
- Definuje generickou sadu `properties`, která popisuje základní aspekty generických komponent.
- Definuje základní datové typy, které lze mapovat na specifické typy cílového jazyka.
- Dovoluje zacházet s rozložením UI tak, aby podporoval různé způsoby technik layoutů.

Když jsem definoval slovník, bylo v další fázi potřeba definovat transformace UIML dokumentu, viz element `peers`. Pro mapování `presentation` jsem použil existující sadu, která je postavena na generickém slovníku používaném při definování `interface` a je jím *Generic 1.2 Harmonia*. Druhé mapování `logic` propojuje uživatelské rozhraní s kontextovou aplikací. Specifikace tohoto mapování se nachází v souboru `logic.uiml` a definuje rozhraní tří komponent: `ContextService`, `PersonService` a `AuiRuleEngine`. K definovanému rozhraní se definuje i způsob propojení, to je učeno pomocí URI adresy v parametru `location`. V návrhu předpokládám propojení pomocí vzdáleného volání EJB. Tento způsob je popsán ve specifikaci UIML [21]. Na příkladu 5.12 je znázorněno mapování EJB `ContextService`.

Příklad 5.12: Definice komponenty ContextService v UIML

```
<d-component id="ContextService" maps-to="cz.cvut.fel.aui.service.ContextService"
               location="ejb://localhost:8080/java:module/ContextService">
        <d-method id="getContext" maps-to="getContext" return-type="c:Context"/>
        <d-method id="save" maps-to="save">
               <d-param id="context" type="c:Context"/>
        </d-method>
    <event class="Event:Context"/>
</d-component>
```

Dále představím základní strukturu použitou pro vytvoření uživatelského rozhraní. Definuji ji podle částí, ze kterých se skládá UI v UIML dokumentech.

Structure Struktura uživatelského rozhraní je definována v souboru `main.uiml`. Ta je tvořena z hlavního kontejneru, který zastřešuje dvě části `menu` a `container`. Menu obsahuje strukturu pro navigaci v UI. Část `container` reprezentuje část UI, ve které se nachází obsah UI. Obsah lze měnit za běhu pomocí elementu `restructure`. Ten poskytuje způsob, jak upravovat strukturu UI při splnění určitých podmínek.

Behavior Chování uživatelského rozhraní jsem rozdělil do čtyř skupin. První skupinou je `navigation`, která obsahuje pravidla pro dynamické změny obsahu UI. Pravidla vykonají restrukturalizaci obsahu při vyvolání události `g:actionperformed`. Například definuji pravidlo, které při vyvolání akce na části `peopleMenu` nahradí obsah v `container` za strukturu, která představuje výpis uživatelů. Další skupinou je `business`. Zde definuji, při jakých akcích

uživatele se vykonají funkce nebo vzdálené metody popsané v UIML dokumentu `logic`. Třetí skupinou je `aui`, která definuje adaptivní chování UI. Toto chování popíši v sekci 5.4.3. Do poslední skupiny spadají vygenerovaná chování pomocí nástroje AspectFaces. Tato skupina definuje chování jednotlivých vygenerovaných částí UI, jako je ukládání hodnot ze vstupních polí při vložení nové hodnoty.

5.4.2 Generování fragmentů

Při návrhu UI jsem počítal pouze s využitím statického generátoru. Není totiž možné použít dynamické generování bez integrace AF do interpreta UIML. Jedinou možností, jak integrovat obsah z jiných zdrojů , je integrace pomocí `template`. Pomocí hodnoty[10] v atributu `source` se určuje, který template je integrován do UIML dokumentu. Jedno z možných řešení je vystavit dynamický generátor jako webovou službu na určité URI adrese, na které by se při požadavku vygeneroval fragment UI. Problém se nachází ve specifikaci atributu `source`, kterému nelze dynamicky parametrizovat adresu, a tak posílat potřebná data na nastavení AF kontextu.

Pro generování fragmentů UI jsem vytvořil statický generátor `AFGenerator`. Ten generuje tři fragmenty UI. Jsou to formuláře, detail entity a tabulky. Fragmenty formuláře a detailu jsou tvořeny pomocí čtyř UIML dokumentů, které lze integrovat pomocí *template*. Tyto dokumenty jsou vygenerovány pomocí těchto AF sad šablon:

structure – tato sada generuje pouze strukturu formuláře. Struktura formuláře je tvořena z formulářových polí, které v sobě obsahuje čtyři části: popisek, obrázek, vstupní pole a nápovědu.

style – tato sada generuje vlastnosti pro jednotlivá vygenerovaná formulářová pole. Generované vlastnosti jsou: text popisku, text nápovědy, validační omezení a adresa k obrázku.

layout – tato sada generuje rozložení jednotlivých prvků ve formulářovém poli. Rozložení je definováno podle vzájemné pozice jednotlivých prvků.

behavior – tato sada generuje chování pro jednotlivé prvky vygenerovaného formuláře. Příkladem vygenerovaného chování je přiřazení nové hodnoty do proměnné, pokud se změní hodnota vstupního pole.

Pro generování tabulek stačí použít jen jednu sadu AF šablon **table**. Ta současně generuje jak strukturu tabulky pomocí sloupců, tak i jejich vlastnosti: text v hlavičce tabulky a reference na obsah sloupce.

5.4.3 Adaptivní uživatelské rozhraní

K řešení adaptivního chování jsem využil popis chování UI v UIML. Toto chování popisuji ve skupině nazvané `aui`, o které jsem se již zmínil v sekci 5.4.1. Zde podrobněji popíši navrhované řešení, jakým lze v UIML dosáhnout adaptace. Celý proces je založen na pravidlech reagujících na událost `Event:Context`, která je vyvolána při změně kontextu aplikace.

[10] Hodnota je adresa k souboru, který obsahuje UIML dokument s template.

Tento event je definován v *logic* komponentě popisující službu `ContextService`. Při změně kontextu se nastaví proměnné s prefixem `AUI`. Ty reprezentují aktuální nastavení adaptace. Pro nastavení těchto hodnot použiji vzdálené volání služby `AuiRuleEngine`.

Některé adaptace jsou řešeny až pomocí interpreta. Takovým příkladem je lokalizace nebo výběr druhu uživatelského rozhraní (grafické, textové atd.). O těchto adaptacích se zmíním při popisu jejich řešení.

Formulářová pole Adaptace pro zobrazení grafické reprezentace popisku a zobrazení nápovědy jsou řešeny pomocí skrývání obsahu. Využívá se k tomu vlastnost `g:visible`, která pro hodnotu `false` nevykreslí danou část UI. Všechny obrázky mají tuto vlastnost propojenou s proměnou `AUI:applyImage`. Části UI reprezentující nápovědu jsou propojeny s proměnou `AUI:applyHelp`.

Formulář Adaptace struktury formulářů jsem nedosáhl. Důvodem je použití statického generátoru UI, viz 5.4.2.

Výpis dat Adaptace reprezentace výpisu je řešená pomocí dvou definic template pro jednu tabulku. Jedna obsahuje strukturu tabulky pomocí třídy `G:Table` s dynamicky vygenerovanými sloupci pomocí nástroje AF. Druhá obsahuje strukturu listu pomocí třídy `G:List`. Jaká definice se použije pro restrukturalizaci UI určí pravidlo, které dle hodnoty v `AUI:table` vybere jednu z template.

Mobilní UI K vytvoření mobilního rozhraní vycházím z publikovaného článku [6]. Využívá stejný generický slovník pro definování platformě nezávislého uživatelského rozhraní, které je pomocí mapování přetransformováno na platformně specifickou podobu UIML. Tuto specifickou podobu lze vykreslit UIML interpretem pro cílovou platformu.

Lokalizace je řešena pomocí různých druhů obsahů (tag `content`). Pro každý jazyk je definován jeden druh identifikovaný dle zkratky jazyka. Lokalizace textů se nacházejí v konstantách, na které se lze pomocí tagu `reference` odkazovat v UI. Samotný výběr druhu obsahu řeší interpret dle jeho implementace.

5.5 UIProtocol

V této kapitole popíši realizaci adaptivního uživatelského rozhraní pomocí UIP. Pro realizaci UI jsem využil existující práci, která tuto problematiku řeší. Jedná se o diplomovou práci Jindřicha Baška [8], která se zabývá generováním uživatelských rozhraní pomocí UIP a JPA. Konkrétně se v této práci nachází řešení pro integraci UIP do Java EE aplikačního serveru. Dále také obsahuje nástroje pro generování uživatelského rozhraní pomocí AspectFaces.

Pro vytvoření adaptivního uživatelského rozhraní jsem využil UiGE [23], který je součástí UIP Serveru. Ten generuje konkrétní uživatelské rozhraní zkombinováním abstraktního uživatelského rozhraní a kontextu použití. Jako jediný ze čtyř jazyků kterými se moje práce zabývá, obsahuje specifikaci a nástroje pro tvorbu kontextově závislých uživatelských rozhraní.

5.5.1 Architektura

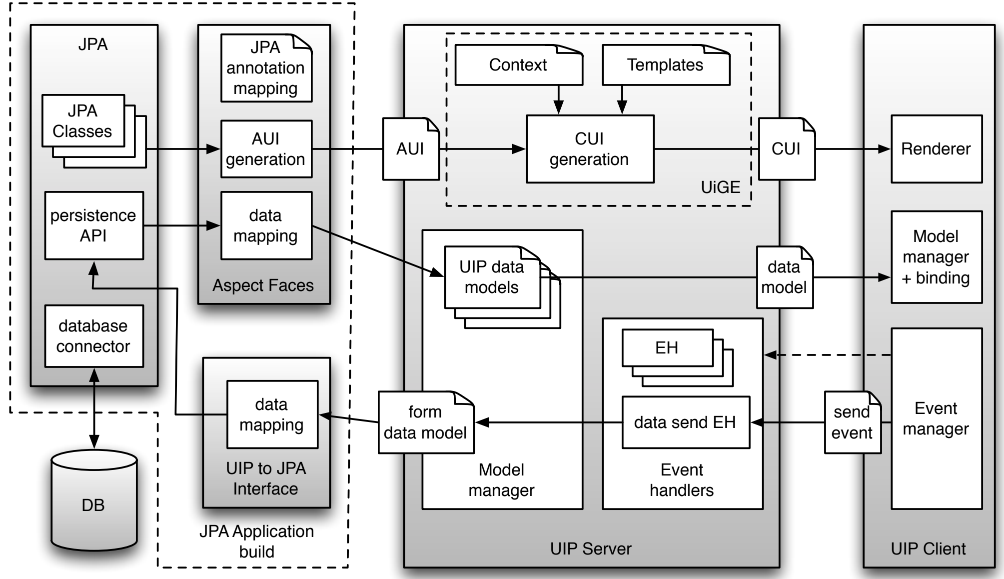
V této části popíši řešení existující architektury dle jednotlivých komponent a jejich rolí v adaptivním UI. Podrobný popis lze nalézt ve zmíněné práci [8], ze které vychází vytvořené UI. Architektura použitá pro integraci UIP je znázorněná v obrázku 5.3. Skládá se ze čtyř hlavních komponent.

AUI Generator Jedná se o komponentu obsahující nástroj AspectFaces. Účelem této komponenty je generovat části abstraktního uživatelského rozhraní z entit. Generátor AUI, kromě popisu AUI, vytváří i pomocné modely UIP pro propojení prvků AUI a ukládání dat vyplněných uživatelem.

UIP Server Jde o Java implementaci serveru pro UIP, který je integrován do Java EE prostředí. Jedná se o hlavní článek UIP architektury, který se stará o vytváření UIP rozhraní, komunikaci s klientskou aplikací a zpracování událostí. UIP Server je propojen s Java EE aplikací pomocí Java Connector Architecture (JCA).

UiGE Je to součást UIP Serveru. Jejím úkolem je poskytovat UIP klientovi konkrétní uživatelské rozhraní, které vygeneruje z abstraktního popisu UI, kontextu použití a pravidel.

UIP Client Jedná se o klientskou aplikaci podporující UIP, která vykresluje konkrétní uživatelské rozhraní a komunikuje s uživatelem aplikace. V případě této práce byl použit .NET klient.



Obrázek 5.3: Architektura systému s JPA aplikací, AspectFaces, UIPServerem a UiGE [24]

Jedním z omezení existujícího řešení, na které jsem během realizace narazil, je propojení UIP Serveru s kontextovou aplikací. Propojení je na úrovni datové vrstvy. Na této vrstvě uživatelské rozhraní přímo přistupuje k datovým zdrojům a tak obchází business vrstvu aplikace. Sám autor tento nedostatek zmínil v textu a vytvořil částečné řešení, které je založené na vzdáleném volání EJB. Zde jsem narazil na omezení pro vzdálené volání Statefull EJB, které používám pro implementaci služeb `AuiRuleEngine` a `ContextService`. Problém je v realizaci JCA konektoru, který nepodporuje JNDI se Statefull kontexty.

Zde vznikl problém při napojení kontextu aplikace s UIP. Nebylo totiž možné využít službu `ContextService`, která se stará o správu kontextu. Řešením je vytvořit bezstavovou business vrstvu a přesunout tak kontext z business vrstvy do prezentační. Tento krok je i velmi logický vzhledem k UIP, které již spravuje kontext a tak není potřeba mít duplicitní implementaci kontextu. Zde ale vzniká problém. Pokud je business logika aplikace kontextově závislá, nemá tato logika přístup ke kontextu v UI a je tak nutné pro každou operaci přidat aktuální kontext. Kontext použití je v AUIP tvořen třemi modely:

User - model popisující schopnosti a preference uživatele (např. porozumění textu, jaké jazyky ovládá atd.)

Device - model popisující vlastnosti zařízení na kterém bude vygenerované rozhraní spuštěné (např. velikost obrazovky, dotykové ovládání, podporované UI element atd.)

Environment - model popisující vlastnosti prostředí, ve kterém se nachází koncové zařízení (např. světelné podmínky, hluk atd.)

Pro propojení UI s kontextovou aplikací je potřeba vyřešit transformaci dat z UIP modulů na entity používané v EJB a zpět. Existující řešení již obsahuje částečně řešenou logiku pro transformaci entit. Ta má určitá omezení, jako je transformace pouze základních datových typů parametrů objektu. To bylo potřeba vyřešit u registrace nových osob, jelikož služba `PersonService` přijímá objekt typu `Person`, který obsahuje parametr `personInfo` neprimitivního datového typu. Řešením je vytvořit vlastní handler `RegistrationMessageHandler`, který vytvoří požadovaný objekt z dat obsažených v UIP modelech. Pro vykonání EJB metody pak použije vytvořený objekt.

5.5.2 Generování fragmentů

Generování fragmentů za pomocí AF a integraci tohoto nástroje jsem převzal z existujícího řešení [8], které umožňuje jak statické, tak i dynamické generování. Pro dynamické generování fragmentů UI je zaregistrován handler pod identifikátorem `integration.aui.generate`. Jeho nedostatkem, na který jsem narazil, je chybějící podpora pro nastavení AF profilu. Dynamicky jsou generovány dva druhy fragmentů UI a to formuláře a detaily. Pro vygenerování jednoho fragmentu je použito pět sad šablon:

Uiprotocol – sada pro generování struktury formuláře. Výstupem je popis abstraktního UI.

LocaleModels – sada pro generování UIP modelů. Výsledkem je model určený pro uchování lokalizovaných textů fragmentů UI.

ModelModels – sada pro generování UIP modelů. Výsledkem je model určený pro uchování dat z formuláře.

SupportModels - sada pro vygenerování UIP modelu. Výsledkem je model určený pro podporu zobrazení validačních zpráv.

SendValidation – sada pro vygenerování UIP handleru. Výsledkem je JS obsahující validační logiku pro formulář.

5.5.3 Adaptivní uživatelské rozhraní

Adaptace UI je řešena pomocí dvou nástrojů. První je AspectFaces, který generuje kontextově závislou strukturu formulářů. Druhým nástrojem je UiGE [23], který se stará o finální podobu CUI. Využívá k tomu různá mapování, kde z AUIP element je mapován na konkrétní UIP element dle specifikovaných pravidel, vlastností elementu a kontextu použití. Níže popsané adaptace nejsou všechna implementována ve výsledném UI. Důvodem je nemožnost konfigurovat mapování UiGE u použitého řešení. Pro změnu mapování je nutné zasáhnout do zdrojového kódu UiGE. Z toho důvodu navrhuji možné řešení pouze pro některé adaptace.

Formulářová pole Každý element v AUIP lze popsat tagem `label` se třemi vlastnostmi: *title*, *icon* a *description*. Ty popisují element textově, ikonou a podrobným popisem. Tyto vlastnosti lze využít pro adaptace 1 a 2, kde UiGE použije tyto vlastnosti pro specifikování finální podoby elementu. Na příklad, pokud uživatel nerozumí psanému textu, tak UiGE použije pro popis vstupního prvku ikonu místo textu.

UiGE umožňuje adaptaci druhu vstupního prvku formuláře, pokud nemá specifikovaný typ tohoto prvku. Ten je vybrán až při generování CUI dle jeho vlastností[11] a chování. Takovým příkladem je element třídy `public.input` s nastaveným výčtem povolených hodnot. Tento element může být reprezentován jako *select box*, *radio button* nebo jeden *checkbox*. Finální podobu pak určuje UiGE podle množství povolených hodnot. Pokud jsou pouze dvě možné hodnoty, tak finální podoba je tvořená jedním checkboxem.

Formulář Adaptace struktury formuláře je řešená pomocí AF popsané v sekci 5.5.2. Adaptace finální podoby formuláře lze řešit pomocí UiGE mapování. Kde se element `container`, který sdružuje formulářová pole, může mapovat na různé reprezentace dle počtu prvků v něm. Jedná se pouze o návrh, který nebyl realizován.

Výpis Adaptace výpisu lze vyřešit stejným způsobem a to mapováním elementu `container`. Ten může být finálně reprezentován jako tabulka, nebo generický kontejner dle podporovaných elementů UIP klienta.

Mobile UI Jelikož je celé uživatelské rozhraní definované v abstraktní podobě, tak není potřeba řešit rozložení nebo reprezentaci prvků UI. O reprezentaci finální podoby se postarají UiGE a UIP klient. UiGE vytvoří vhodnou reprezentaci UI, která je zaslána UIP klientovi.

[11] Příklady vlastností: podporované elementy UIP klienta, množství místa na obrazovce, počet vstupních možností, velikost obrazovky atd.

Ten pak vykreslí uživatelské rozhraní podle finální platformy (.NET, Windows Mobile 8, iOS atd.). Tento způsob přináší výhody při nasazení UI na různé platformy a zařízení. [24]

Lokalizace je v UIP řešena pomocí více variant jednoho modelu, viz 3.4.2. Konkrétní varianta modelu pak obsahuje data pro danou lokalizaci.

Kapitola 6

Testování

Účelem této kapitoly je porovnat jednotlivé jazyky pomocí experimentů nad AUI. Na závěr této kapitoly provedu zhodnocení všech porovnávaných jazyků a popíši jejich výhody a nevýhody. Nejprve ozřejmím kritéria, která budou použita při porovnávání jednotlivých jazyků pro tvorbu adaptivních uživatelských rozhraní.

6.1 Kritéria porovnání

Pro účely porovnání jednotlivých jazyků a frameworků pro tvorbu adaptivního uživatelského rozhraní jsem zvolil několik kritérií, která definují vlastnosti porovnávaných technologií.

Produktivita Produktivita je jednou z velmi důležitých vlastností, protože UI představuje značnou část zdrojového kódu aplikace. Průzkum [26] provedený v roce 1992 ukázal, že kód UR tvoří 48% celkového aplikačního kódu a jeho vytvoření zabírá 50% času vývoje. Pro měření produktivity provedu experiment měření kódu 6.2. Při určování produktivity se zaměřím na tyto parametry.

- Celkový počet řádků UI kódu
- Počet řádků konfigurace
- Počet řádků pro integraci UI s kontextovou aplikací

Znovupoužitelnost U jednotlivých jazyků zkoumám znovupoužitelnost fragmentů UI a k tomu určených metodik. Důvodem, proč se zabývám znovupoužitelností, je to, že toto kritérium zvyšuje produktivitu při vývoji a údržbě UI. Umožňuje použít již jednou napsanou část kódu na více místech v aplikaci. Tak odstraňuje duplicity kódu. Tímto kritériem se zabývám v experimentu měření kódu 6.2. Při porovnávání se zaměřím na tyto vlastnosti:

- Množství znovupoužitelného kódu UI
- Oddělení částí UI (Separation of Concerns)

- Vytváření vlastních widgetů
- Templates

Datový přenos Dalším kritériem je množství dat potřebných pro komunikaci mezi klientem a serverem, což má následně vliv na celkovou odezvu. Účelem tohoto testu je zjistit, jak moc UI zatěžuje komunikaci na síti. Měřím množství dat přenesených mezi klientem a server pro vygenerování uživatelského rozhraní a následný vliv na odezvu UI, kde na mobilních platformách může být díky vysokému datovému přenosu znatelná odezva, případně i finanční zátěž.

- Množství přenesených dat
- Odezva
- Četnost komunikace

Flexibilita a adaptace Jedná se o důležitá kritéria, která udávají možnost dosažení cílového vzhledu UI. Výsledkem těchto kritérií je, zda vytvořená UI dokáží dosáhnout výsledného chování UI, které jsem specifikoval v kapitole 4.2. Vzhledem k flexibilitě se budu zabývat i možnostmi jazyka pro adaptaci uživatelského rozhraní, kde se zaměřím na tyto vlastnosti:

- Adaptace navigace
- Adaptace obsahu
- Oddělení obsahu od prezentace
- Flexibilní struktura

Nezávislost na platformě Tímto kritériem se zaměřuji na závislost konkrétní technologie pro tvorbu UI na cílovou platformu klienta a serveru. Čím menší závislost bude mít daná technologie, tím lépe, jelikož lze snadno rozšířit množství podporovaných platforem bez zásahu do aplikace. Kritéria měření jsou:

- Závislost na platformě klienta
- Závislost na platformě serveru (databinding)
- Množství práce při migraci na jiné platformy

6.2 Experiment měření kódu

Smyslem tohoto experimentu je porovnat zdrojové kódy jednotlivých adaptivních uživatelských rozhraní. Jejich realizace je popsána v kapitole 5. K porovnání používám softwarovou metriku SLOC. Ta je založena na měření počtu řádku zdrojového kódu programu. SLOC se typicky používá k předpovědi množství úsilí, které je potřebné k vytvoření programu.

6.2.1 Nastavení experimentu

Měření se zabývá pouze samotnými uživatelskými rozhraními bez kontextově závislé aplikace. Vzhledem k velkému množství jazyků použitých při vývoji uživatelských rozhraní, je potřeba brát v potaz vyjadřovací schopnosti jednotlivých jazyků. Proto pro měření počtu řádků jsou použity dvě metriky. První metrikou je počet fyzický řádků LOC, ta počítá skutečný počet řádků kódu (bez prázdných řádků a komentářů). Nevýhoda LOC je v citlivosti na formátování a ve vyjadřovacích schopnostech jazyka, čímž se zkresluje skutečnost[1]. Druhou metrikou, která není citlivá na formátování, je měření logických řádků LLOC. Tato metrika měří počet proveditelných řádků. Způsob počítání je u této metodiky vázán na konkrétní programovací jazyk. Vzhledem ke kombinaci různých druhů jazyků, které mají rozdílné vyjadřovací schopnosti a formátování, jsem použil obě dvě metodiky měření LOC a LLOC. Tím lze dostat lepší představu o kódu jednotlivých UI.

Pro měření obou metrik byl použit program Unified Code Count (UCC) [30], který byl rozšířen pro podporu jazyků XUL, UIML a UIP. Do měření se nepočítají cizí knihovny potřebné pro realizaci AUI, jelikož nebylo vynaloženo úsilí na jejich realizaci.

6.2.2 Výsledky měření

V kapitole jsou prezentovány výsledky z měření experimentu. Výsledky jsou rozděleny do čtyř částí. Měření proběhlo na JSF AUI, na rozdíl od ostatních AUI, třikrát. Důvodem je použití různých způsobů adaptace platformy UI, viz 5.2.3. JSF-B je uživatelské rozhraní využívající responsivní design pro tvorbu mobilního UI. JSF-P je AUI využívající rozšíření PrimeFaces pro tvorbu mobilní a desktopové verze UI. JSF je UI, které pro realizaci desktopové a mobilní platformy používá rozdílné widgety.

Tabulka 6.1 prezentuje produktivitu jednotlivých jazyků a frameworků. K měření jsou použity tři parametry, které reprezentují množství úsilí potřebné pro vytvoření UI. *Integrace* udává množství SLOC potřebné pro integraci uživatelského rozhraní s kontextovou aplikací. *Konfigurace* udává množství SLOC nutné na konfiguraci AUI. Poslední parametr *UI* udává SLOC pro definování uživatelského rozhraní.

Parametry	JSF	JSF-B	JSF-P	XUL	UIP	UIML
LOC						
Integrace	384	384	384	477	430	119
Konfigurace	279	279	279	277	333	1286
UI	1100	742	603	708	356	449
Celkem UI	1763	1405	1266	1462	1119	1854
LLOC						
Integrace	269	269	269	338	326	82
Konfigurace	204	204	204	200	260	951
UI	547	368	282	379	255	282
Celkem UI	1020	841	755	917	841	1315

Tabulka 6.1: Výsledky měření SLOC UI.

[1] př. OOP vs. XML

Tabulka 6.2 reprezentuje množství znovupoužitelného kódu při realizaci UI. Do znovupoužitelného kódu AUI se počítají fragmenty, které lze opět použít na více místech UI. Příkladem jsou AF šablony, vlastní widgety, layouty atd. Toto měření je zaměřené pouze na kód definující uživatelské rozhraní, a proto se nepočítá integrace a konfigurace.

Parametry	**JSF**	**JSF-B**	**JSF-P**	**XUL**	**UIP**	**UIML**
LOC						
UI	1100	742	603	708	356	449
Znovupoužitelné UI	800	442	303	313	278	157
Znovupoužitelné %	72.73%	59.57%	50.25%	44.21%	78.09%	34.97%
LLOC						
UI	547	368	282	379	255	282
Znovupoužitelné UI	391	212	126	196	215	101
Znovupoužitelné %	71.48%	57.61%	44.68%	51.72%	84.31%	35.82%

Tabulka 6.2: Výsledky měření znovupoužitelnosti zdrojového kódu.

Tabulka 6.3 obsahuje údaje o celkovém množství úsilí pro realizaci adaptace. Toto měření je provedeno napříč celým UI, protože je část logiky adaptace obsažená jak v integraci, tak v konfiguraci. Hlavním výstupem tohoto měření je množství kódu, které je nutné vytvořit pro realizaci adaptací.

Parametry	**JSF**	**JSF-B**	**JSF-P**	**XUL**	**UIP**	**UIML**
LOC						
Celkové UI	1763	1405	1266	1462	1119	1854
AUI	1098	269	249	395	72	176
AUI %	62.28%	19.15%	19.67%	27.02%	6.43%	9.49%
LLOC						
Celkové UI	1020	841	755	917	841	1315
AUI	619	189	179	222	23	143
AUI %	60.69%	22.47%	23.71%	24.21%	2.73%	10.87%

Tabulka 6.3: Výsledky měření adaptivního zdrojového kódu.

Tabulka 6.4 obsahuje údaje o množství úsilí pro realizaci jednotlivých druhů adaptací. V tomto měření nejsou uvedeny realizace JSF-B a JSF-P. Obě tyto realizace jsou shodné s JSF, až na adaptaci platformy. Jelikož JSF-B je realizací desktopové platformy a JSF-P mobilní. V adaptaci platformy je znázorněné množství úsilí na vytvoření UI pro více platforem. Řádky *desktopová* a *mobilní* platforma udávají, o kolik bylo potřeba rozšířit UI pro realizaci obou platforem. Množství úsilí na vytvoření adaptace je v tabulce rozdělené na fixní, jedná se o *konstantní* množství SLOC pro realizaci adaptace a *variabilní*, v němž je úsilí závislé na počtu aplikací dané adaptace napříč UI. Hodnota určuje množství práce pro realizaci jedné adaptace. Hodnoty v závorkách udávají náročnost adaptace, pokud její implementaci lze umístit do AF šablon.

	JSF		XUL		UIP		UIML	
Adaptace	*LOC*	*LLOC*	*LOC*	*LLOC*	*LOC*	*LLOC*	*LOC*	*LLOC*
Platforma								
Desktopová	497	265	45	11	0	0	0	0
Mobilní	358	179	62	17	0	0	0	0
Celkem rozšířeno	855	444	107	28	0	0	0	0
Formulářová pole								
Fixní	2 (8)[1]	2 (6)	82	40	0	0	2 (8)	2 (6)
Variabilní[2]	6 (0)	4 (0)	0	0	0	0	6 (0)	4 (0)
Formulář								
AF generátor[3]	102	76	94	70	0	0	79	62
Generování (Var.)	1	1	1	1	13	3	-	-
Configurace	41	30	43	32	0	0	81	54
Výpis dat								
Fixní	71	29	18	9	0	0	24	12
Variabilní[4]	1	1	1	1	0	0	20	13

[1] Hodnota v závorkách udává realizaci pomocí AF. [2] Závisí na množství formulářových polí v UI.
[3] Úsilí k nasazení AF generátoru a jeho propojení s kontextem. [4] Závisí na množství tabulek v UI.

Tabulka 6.4: Výsledky měření potřebného úsilí pro realizaci adaptace.

6.2.3 Vyhodnocení

Při vyhodnocení je potřeba brát v potaz dvě rozdílné skupiny porovnávaných jazyků. Jednou skupinou jsou jazyky pro popis CUI (JSF a XUL) a druhou skupinou jsou jazyky umožňující abstraktní popis UI (UIP a UIML).

Produktivita Nejlepší výsledky ohledně produktivity dosahuje UIP s UiGE při definování UI, jeho nevýhodou je však obrovská náročnost na integraci s kontextovou aplikací. Nejhorších výsledků vzhledem k produktivitě dosahuje UIML. To je důsledkem obrovské velikosti konfigurace nutné pro definování transformací a slovníku. JSF má velmi dobré výsledky, vezmeme-li v potaz frameworky využívající stejnou sadu widgetů pro tvorbu mobilního i desktopového UI. Pokud cílové platformy používají rozdílnou sadu widgetů, je JSF neefektivní.

Znovupoužitelnost Vysoké procento znovupoužitelného kódu u všech AUI je zapříčiněno nástrojem AF, kde většinu SLOC tvoří AF šablony a templaty. Z výsledků nelze vyvodit, který z jazyků je lepší. Tyto údaje vystihují charakteristiku vytvořeného UI. UI z JSF je tvořeno převážně z vlastních widgetů a dvou sad AF šablon pro mobilní a desktopovou podobu UI. Znovupoužitelný kód v UIP není tvořen AF šablonami, jako u ostatních implementací, ale je tvořen handlery pro validaci vstupních polí.

Adaptace Ohledně vytváření adaptivních uživatelských rozhraní vítězí jazyky postavené na popisu UI v abstraktní podobě. Konkrétně se jedná o UIP a UIML. Je to zapříčiněno posunutím zodpovědnosti, ohledně finální podoby, na vyšší úroveň. Konkrétně u UIP, které

dopadlo nejlépe, se adaptace řeší při generování konkrétního uživatelského rozhraní (UiGE) a UIP klienta. Jako jediný z jazyků obsahuje nástroje pro tvorbu kontextově závislých UI. Nejhůř dopadl JSF a to vzhledem k rozšíření UI pro jiné platformy. Na rozdíl od ostatních jazyků je potřeba pro rozšíření UI na jinou platformu vynaložit několikanásobně vyšší úsilí. U XUL není potřeba vynaložit příliš velké úsilí na rozdíl od JSF. Pro každou platformu se používá stejná sada widgetů. Jediné, co bylo potřeba řešit, je rozložení prvků UI a rozdílné chování.

U adaptace formulářových polí je nejdůležitějším parametrem variabilní náročnost. Ta dopadla nejlépe u UIP a XUL. Tyto implementace mají pouze konstantní náročnost. Zato UIML a JSF mají konstantní náročnost pouze v případě, když je adaptace prováděna pomocí automatického generování fragmetů kódu. Pokud chceme přidat adaptaci mimo automaticky generovaný kód (např. menu) je pokaždé nutno ručně přidat část implementace adaptace. Úsilí je tak lineárně závislé na počtu prvků.

Adaptace struktury formuláře pomocí AF nebylo dosaženo u jazyka UIML, viz 5.4.2. Byl pouze použit statický generátor. U UIP je pro vygenerování jednoho fragmentu potřeba několikanásobně více úsilí než u JSF a XUL. Je to zapříčiněno způsobem nastavování AF kontextu. Ten je konfigurován zvlášť pro každý generovaný fragment a je tak variabilní. U JSF a XUL je nastavování AF kontextu umístěné v AF generátoru, který se stará o nastavení AF kontextu sám.

Mezi nejlepší jazyky vzhledem k adaptaci výpisu patří UIP s UiGE. Nejhůře dopadl UIML a to vzhledem k výběru slovníku, který neumožnil definovat seznam a tabulku pomocí stejné struktury. To má za následek vytvoření dvou separátních struktur.

6.3 Experiment měření datového přenosu

Smyslem tohoto experimentu je porovnat jednotlivé implementace uživatelských rozhraní vzhledem k jejich požadavkům na komunikaci mezi serverem a klientem. Toto pozorování má za úkol ukázat, jak moc tyto jazyky a frameworky zatěžují síťový přenos při jejich použití v architektuře klient-server, která je použita při realizaci kontextové aplikace. Tento experiment má ukázat jaký vliv má komunikace na odezvu UI a případně i zátěž závislou na množství přenesených dat.

6.3.1 Nastavení experimentu

Experiment je proveden na třech AUI a to JSF, XUL a UIP. Na UIML nebylo možné provést toto pozorování vzhledem k tomu, že nebyl vytvořen funkční prototyp, viz 5.4. Pro účely měření byla kontextová aplikace spuštěna na serveru, jehož konfigurace je v tabulce 6.5b. K tomuto serveru se jednotlivá uživatelská rozhraní připojují jako klienti. Tyto UI jsou spuštěné na separátním počítači, jehož konfigurace je uvedena v tabulce 6.5a. Server s klientem jsou propojeni přes WiFi síť, která je v módu 54Mbps (802.11g). Uživatelské rozhraní JSF a XUL poběží na webovém prohlížeči *Mozilla Firefox*. K otestování UIP uživatelského rozhraní je použit *.NET* klient.

Pro měření jsou použity dva nástroje. U JSF a XUL se měří síťová aktivita pomocí vývojářského nástroje pro analýzu síťového provozu, který je integrován v prohlížeči. Pro

měření síťové aktivity u UIP bylo potřeba použít nástroj *Wireshark*, jelikož UIP používá vlastní protokol pro komunikaci mezi klientem a serverem.

Samotné měření je prováděné na jednom scénáři použití aplikace, scénář je pro všechny UI stejný. Pro experiment byla povolena cache, pokud to umožnil klient. Před každým zahájením scénáře byla cache vyčištěna. Pro experimenty byl zvolen tento scénář:

1. Zahájit relaci
2. Otevřít formulář pro registraci nového uživatele.
   (a) Vložit nevalidní data
   (b) Odeslat formulář
   (c) Vložit validní data
   (d) Odeslat formulář
3. Otevřít formulář pro úpravu kontextu
   (a) Změnit kontext věku uživatele na hodnotu Dítě
   (b) Uložit kontext
4. Otevřít formulář registrace
   (a) Vyplnit validní údaje
   (b) Odeslat formulář
5. Ukončit relaci

Procesor	Intel Core i5, 2,4 GHz
Paměť	8 GB, DDR3
OS	Windows 8 64bit
Prohlížeč	Mozilla Firefox 34
UIP klient	UIProtocol.dotNET

(a) Klient

Procesor	Intel Core i7, 2.3GHz
Paměť	8 GB, DDR3
OS	OS X Yosemite
Java	Oracle JDK 7 update 45 64bit
Server	JBoss AS 7.1.1.Final

(b) Server

Tabulka 6.5: Testovací sestava

6.3.2 Výsledky měření

Výsledky měření jsou uvedeny v tabulce 6.6. Hodnoty jsou zde uvedeny sumárně po splnění celého scénáře. Parametr počet dotazů udává, kolikrát klient komunikoval se serverem. Parametr přeneseno dat udává, kolik se přeneslo dat po síti. Započítává se i multimediální obsah jako ikony, CSS atd. Parametr Celková odezva udává součet všech časů nutných pro vykonání všech dotazů. Poslední tři údaje znázorňují nejkratší, nejdelší a medián odezvy ze všech komunikací nutných pro splnění scénáře.

6.3.3 Vyhodnocení

Jak lze vyčíst z výsledků měření, tak UIP potřebuje pro vykonání stejného scénáře několikanásobně více komunikace se serverem. To je způsobeno samotnou architekturou UIP,

	JSF	**XUL**	**UIP**
Protokol	HTTP	HTTP	UIProtocol
Počet dotazů	26	18	93
Přeneseno dat (kB)	394,79	111,34	157,6
Celková odezva (s)	1,184	1,034	7,07
Min. odezva (s)	0,05	0,05	0,0004
Max. odezva (s)	0,2	0,25	1,5
Medián odezvy (s)	0.1	0.1	0.05

Tabulka 6.6: Výsledky měření komunikace

kde mezi sebou server a klient často komunikují. Velkou část komunikace tvoří eventy pro validaci vstupních polí, kterým byla změněna hodnota a ztratil se focus. Samotná odezva pro odeslání jednoho UIP dokumentu je však velmi rychlá. Je to díky pouze jednomu TCP spojení, které je po celou dobu sezení klienta otevřené a tak není potřeba pokaždé navazovat spojení se serverem, jako je tomu u HTTP protokolu použitého u JSF a XUL.

Problém u UIP je celkový čas strávený čekáním na odezvu od serveru. Tento čas je až 7x delší než u ostatních UI. To má za následek znatelnou prodlevu reakcí UI, která byla znatelná již na výše uvedené testovací konfiguraci. Na mobilních sítích by tato prodleva byla ještě znatelnější.

V množství přenesených dat dopadl nejlépe XUL. Je to dáno tím, že se jedná o tlustého klienta, který obsahuje většinu definice UI, jako jsou XUL dokumenty, obrázky, CSS a JS. Klient využívá server pro vykonání business operací a ke generování fragmentů uživatelského rozhraní. Toto generování tvoří většinu datového přenosu. JSF je, na rozdíl od XUL, tenký klient a proto musí vše stahovat ze serveru. Při prvním požadavku na stránku si prohlížeč uloží do webové cache 273 kB. Je tvořena CSS a JS soubory. Vzhledem k množství přenášených dat je nejnáročnější JSF, to může mít vliv při použití v mobilních datových sítích.

6.4 Srovnání jazyků

Na základě praktických zkušeností, získaných při implementaci adaptivních uživatelských rozhraní a analýze výsledných zdrojových kódů, shrnu výhody a nevýhody jednotlivých jazyků a frameworků. V tabulce C.1 jsou vyobrazeny rozdíly mezi nimi.

JSF Největší výhodou tohoto jazyka je to, že nabízí kompletní framework pro tvorbu webových uživatelských rozhraní pro Java EE aplikace. Na rozdíl od ostatních jazyků usnadňuje vývojářům práci při integraci aplikace, protože způsob integrace je jasně dán. Je to díky Java EE specifikaci a i následné kompatibilitě s ostatními Java EE technologiemi. Další výhodou je relativně rychlé zvládnutí tohoto frameworku, pokud již člověk má zkušenosti s webovými technologiemi. Kladem je i velké množství již existujících nástrojů pro vývoj JSF.

Mezi nevýhody patří omezení tohoto jazyka pouze pro tvorbu webových rozhraní, a tak například nelze vytvořit konzolové rozhraní. Pro tvorbu adaptivního rozhraní nepatří mezi nejefektivnější z porovnávaných jazyků. Velkou část adaptací je totiž nutno řešit ručně

pomocí podmínek v kódu. Je to zapříčiněno omezenou prací s hierarchickou strukturou komponent, kterou nelze upravovat bez zásahu do zdrojového kódu. Vývoj UI pro více druhů zařízení je bez použití rozšiřujícího frameworku velmi neefektivní. S tím je spojena i finální podoba widgetů, která je silně svázána se samotnou implementací komponenty, a proto ji nelze jednoduše nahradit. Často je nezbytně nutné vytvořit novou komponentu.

XUL Mezi největší výhody tohoto jazyka jsou nástroje pro práci se strukturou UI. Jako jediný z porovnávaných jazyků nabízí nástroje pro snadnou modifikaci struktury UI, bez nutnosti zasahovat do zdrojového kódu. Pro tvorbu adaptivního rozhraní se velmi osvědčila technologie XBL Binding, pomocí které lze velmi efektivně aplikovat adaptace struktury UI napříč celým UI. Také je tato technologie úsporná ohledně zdrojů na serveru, který nemusí skládat finální podobu UI. Tu si samo vytvoří vykreslovací jádro. Samotné zvládnutí tohoto jazyka je velmi snadné, pokud vývojář ovládá DHTML.

Nevýhodou tohoto jazyka je jeho omezení pouze na GUI. Samotný jazyk XUL není standardizovaný, a tak není podporován mimo aplikace společnosti Mozilla. Zásadním nedostatkem tohoto jazyka je způsob integrace s Java EE aplikací. Neobsahuje totiž žádné existující řešení pro integraci. Té lze dosáhnout za pomocí Xulfaces, což je použití JSF. Tento jazyk sice obsahuje nástroj pro propojení business vrstvy s prezenční, ale jen z jednoho směru a to pomocí datasource (získávání dat). Je potřeba vyvinout vlastní řešení pro odesílání dat a také nástroje pro validaci.

UIML Největší výhodou tohoto jazyka je jeho koncept, který umožňuje popisovat uživatelské rozhraní nezávisle na cílové platformě. Lze tak definovat jakékoliv UI nezávislé na zařízení, interakci i jazyku UI. Díky jeho vlastnosti metajazyka lze definovat všechna porovnávaná AUI (JSF, XUL a UIP) jen za pomocí UIML a mapování. Koncept jazyka klade důraz na rozdělení zodpovědností a to znatelně od ostatních jazyků zvyšuje znovupoužitelnost. Odděluje strukturu, vlastnosti, obsah, chování, aplikační logiku a finální reprezentaci, jež lze libovolně nahradit, a tím kompletně změnit chování UI.

Největší nevýhodou je náročnost na zvládnutí tohoto jazyka. Kvůli širokému rozdělení zodpovědností je jazyk velmi popisný a pro definování jednoduchého vstupního pole je potřeba napsat několikanásobně více kódu než u ostatních jazyků. To je zapříčiněno jeho zaměřením na strojové zpracování nástrojů generujících UI. Tento problém lze vyřešit pomocí generátorů, jako je AF. Při vytváření adaptivního rozhraní nezáleží tak moc na UIML, ale hlavně na zvoleném slovníku a transformacích pro cílovou platformu.

UIP Tento jazyk umožňuje definovat uživatelská rozhraní pro libovolnou platformu a zařízení. Na rozdíl od ostatních porovnávaných jazyků obsahuje nástroje pro definici kontextově závislých UI. S využitím AUIP a UiGE lze velmi jednoduše vytvořit adaptivní UI. Jazyk obsahuje vlastní protokol s full-duplexním komunikačním kanálem mezi klient-server.

Jedná se o čerstvě vyvíjenou technologii, a proto, na rozdíl od ostatních, není ještě ustálena. Konkrétně se jedná o specifikaci AUIP, která je v rané fázi vývoje. Současným omezením je chybějící vývojářská dokumentace s příklady a znovupoužitelnost již vyvinutých nástrojů, u kterých hlavně chybí možnost konfigurace (UiGE). S tím souvisí i počáteční náročnost na zvládnutí této technologie. Velkým omezením tohoto jazyka je nemožnost definovat dynamický obsah přímo v UI, to je pak nutné řešit programově pomocí handlerů.

Kapitola 7

Závěr

V rámci této práce jsem zkoumal využití aspektově-orientovaného přístupu pro tvorbu adaptivních uživatelských rozhraní a využití této metodiky s frameworky a doménově specifickými jazyky pro tvorbu těchto rozhraní. Za tímto účelem jsem nastudoval čtyři doménové jazyky a jejich frameworky. Jedná se o jazyky XML User Interface Language, User Interface Markup Language, User interface protocol a Java Server Faces. Tyto jazyky jsem použil k vytvoření adaptivních rozhraní pro kontextově závislou aplikaci a následně zhodnotil jejich výhody, nevýhody a omezení.

V rámci této práce byla implementována kontextová aplikace, která je založená na platformě Java EE. V návrhu aplikace jsem určil na jakých kontextových parametrech je aplikace závislá a následně specifikoval její chování při změně hodnot v těchto parametrech. Během implementace jsem využil standardní Java EE technologie pro distribuci kontextu a jeho změn do závislých komponent aplikace.

V následující fázi byla vytvořena tři funkční adaptivní rozhraní za pomocí jazyku JSF, XUL a UIP. Pro čtvrté rozhraní pomocí jazyka UIML bylo pouze navrhnuto možné řešení implementace, protože funkční uživatelské rozhraní se nepodařilo vytvořit z důvodu absence vyhovujících nástrojů.

Za účelem porovnání těchto adaptivních rozhraní bylo potřeba navrhnout jejich jednotné adaptivní chování závislé na změnách kontextu použití. Toto adaptivní chování bylo posléze definováno pomocí doménově specifického jazyka DRL, což je jazyk určený pro popis business pravidel. Tato definice byla vložena do separátní komponenty aplikace. Tato pravidla, dle aktuálních kontextů aplikace, vytvoří konfiguraci adaptace, podle níž se adaptují uživatelská rozhraní. Vytvořil jsem tak centralizovanou konfiguraci těchto pravidel a jejich distribuce do uživatelských rozhraní.

Při vývoji adaptivních UI byl využit aspektově-orientovaný přístup. Konkrétně se jedná o nástroj AspectFaces, který dle meta-modelu získaného inspekcí entit a kontextu použití generuje adaptované fragmenty uživatelského rozhraní. Tento nástroj byl použit ve všech čtyřech uživatelských rozhraní pro generování fragmentů UI. Jako jsou formuláře, detail objektů, tabulky a seznamy. Tento nástroj je velmi univerzální a šlo jej použít pro generování struktury, vzhledu, chování a obsahu UI.

Při realizaci jedné adaptace jsem narazil na omezení nástroje AspectFaces. Jednalo se o adaptaci formuláře, který se má při určitém množství vstupních prvků změnit na wizard.

AspectFaces totiž při finálním sestavení fragmentu dovoluje použít pouze předem daný layout. Toto omezení by bylo možné vyřešit rozšířením funkcionality AspectFaces o dynamické vybírání layoutu dle meta-modelu. V tomto konkrétním případě dle počtu parametrů entity.

Po porovnání jazyků navrhuji dvě efektivní cesty jak vyvíjet adaptivní uživatelská rozhraní. První je definovat rozhraní v abstraktní podobě, která definuje co prezentovat uživateli, místo toho jak. Finální podobu zajistí generátor uživatelského rozhraní, který dle definovaných pravidel a kontextu použití sestaví konkrétní uživatelské rozhraní. Tento způsob výrazně snižuje množství úsilí vývojáře potřebné na vytvoření adaptivního rozhraní. Takovým frameworkem je UIP s UiGE. Druhou možností je použít jazyk, který je velmi flexibilní a umožňuje dynamicky měnit strukturu UI bez zásahu do zdrojového kódu. Tímto jazykem je XUL a ekosystém jazyků používaných kolem něj. JSF a UIML nejsou příliš efektivní při vytváření adaptací. Nevýhodou u JSF je příliš složitá práce při dynamických úpravách již existující struktury UI. UIML je velmi složitý pro použití a špatně čitelný pro vývojáře. Je to zapříčiněno jeho hlavním zaměřením, které cílí na použití v generátorech UI.

7.1 Budoucí vývoj

Na tuto práci lze v budoucnu navázat z různých směrů. Jedním ze směru je využití zjištěných poznatků pro budoucí vývoj adaptivního uživatelského rozhraní. Zde doporučuji vydat se směrem technologií UIP nebo XUL.

Dále navrhuji analyzovat momentálně vyvíjený webový standart Web Component[1] a jeho možnosti pro vývoj adaptivních uživatelských rozhraní. Tento standart by v budoucnu měl přinést do HTML5 technologie podobné těm z XUL. Jsou to technologie Shadow DOM a vlastní elementy.

Dalším směrem je zaměření se na nástroj AspectFaces a možnost rozšíření jeho funkcionality o dynamické vybírání layoutů v závislosti na meta-modelu.

Dále lze rozvinout řešení kontextového povědomí v Java EE aplikacích, buď v podobě použití Java EE technologií, nebo integrací již existujících frameworků.

Navázat lze i v možnostech využití doménově specifických jazyků pro definování pravidel adaptivního chování.

[1] <http://www.w3.org/TR/components-intro/>

Literatura

[1] *Drools dokumentation* [online]. Red Hat JBoss Middleware, 2014. [cit. 20. 11. 2014]. Dostupné z: <http://docs.jboss.org/drools/release/6.1.0.Final/drools-docs/html_single/>.

[2] *XML User Interface Language* [online]. Mozilla Foundation, 2014. [cit. 9. 12. 2014]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL>.

[3] *XULRunner documentation* [online]. Mozilla Foundation, 2014. [cit. 9. 12. 2014]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/XULRunner>.

[4] *Xulfaces*, 2007. Dostupné z: <http://xulfaces.sourceforge.net/>.

[5] ABOWD, G. et al. Towards a Better Understanding of Context and Context-Awareness. In GELLERSEN, H.-W. (Ed.) *Handheld and Ubiquitous Computing*, 1707 / *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999. s. 304–307. doi: 10.1007/3-540-48157-5_29. Dostupné z: <http://dx.doi.org/10.1007/3-540-48157-5_29>. ISBN 978-3-540-66550-2.

[6] ALI, M. F. – PÉREZ-QUINONES, M. A. – ABRAMS, M. Building multi-platform user interfaces with UIML. *Multiple User Interfaces–Cross-Platform Applications and Context-Aware Interfaces*. 2004, s. 95–118.

[7] Aspect-oriented programming. *In: Wikipedia: the free encyclopedia* [online]. Wikimedia Foundation, 2014. [cit. 9. 12. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Aspect-oriented_programming>.

[8] BAŠEK, J. *Generování uživatelských rozhraní pomocí UIP a JPA*. Diplomová práce, ČVUT, Fakulta informačních technologií, katedra softwarového inženýrství, Praha, 2013.

[9] BAŠEK, J. *Webové řešení pro podporu vývoje v UIProtocolu*. Bakalářská práce, ČVUT, Fakulta elektrotechniská, katedra počítačů, Praha, 2011.

[10] BERNARD, E. *Bean Validation specification*, 2013. Dostupné z: <https://jcp.org/en/jsr/detail?id=349>.

[11] BULLARD, V. – SMITH, K. T. – DACONTA, M. C. *Essential XUL Programming*. New York : Wiley, July 2001. ISBN 978-0-471-41580-0.

[12] BURNS, E. – KITAIN, R. *JavaServer^{TM} Faces Specification*, 2009. Dostupné z: <https://jcp.org/aboutJava/communityprocess/final/jsr314/>.

[13] CERNY, T. – CEMUS, K. *AspectFaces documentation* [online]. Coding Crayons. [cit. 9. 12. 2014].

[14] CERNY, T. – SONG, E. UML-based Enhanced Rich Form Generation. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, s. 192–199, New York, NY, USA, 2011. ACM. doi: 10.1145/2103380.2103420. Dostupné z: <http://doi.acm.org/10.1145/2103380.2103420>. ISBN 978-1-4503-1087-1.

[15] CERNY, T. et al. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. *SIGAPP Appl. Comput. Rev.* December 2013, 13, 4, s. 53–66. ISSN 1559-6915. doi: 10.1145/2577554.2577561. Dostupné z: <http://doi.acm.org/10.1145/2577554.2577561>.

[16] CERNY, T. – DONAHOO, M. J. – SONG, E. Towards Effective Adaptive User Interfaces Design. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, s. 373–380, New York, NY, USA, 2013. ACM. doi: 10.1145/2513228.2513278. Dostupné z: <http://doi.acm.org/10.1145/2513228.2513278>. ISBN 978-1-4503-2348-2.

[17] CHEN, G. – KOTZ, D. et al. *A survey of context-aware mobile computing research.* Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, 2000.

[18] DEMICHIEL, L. *JSR 317: Java^{TM} Persistence API*, 2009. Dostupné z: <https://jcp.org/en/jsr/detail?id=317>.

[19] HANUMANSETTY, R. G. *Model based approach for context aware and adaptive user interface generation.* Master of Science, Virginia Polytechnic Institute and State University, 2004.

[20] *Generic UIML Vocabulary Specification.* Harmonia, 2011. Dostupné z: <http://www.uiml.org/toolkits/Generic_UIML_Vocabulary_Specification.pdf>.

[21] HELMS, J. et al. *User Interface Markup Language (UIML).* OASIS, 2008. Dostupné z: <http://docs.oasis-open.org/uiml/v4.0/cd01/uiml-4.0-cd01.html>.

[22] Java EE. *In: Wikipedia: the free encyclopedia* [online]. Wikimedia Foundation, 2014. [cit. 9. 12. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Java_EE>.

[23] MACÍK, M. *User Inteface Generator.* Dissertation Thesis Proposal, CTU in Prague, 2011. DCSE-DTP-2011-01.

[24] MACIK, M. et al. Platform-Aware Rich-Form Generation for Adaptive Systems through Code-Inspection. In HOLZINGER, A. et al. (Ed.) *Human Factors in Computing and Informatics*, 7946 / *Lecture Notes in Computer Science.* : Springer Berlin Heidelberg, 2013. s. 768–784. doi: 10.1007/978-3-642-39062-3_55. Dostupné z: <http://dx.doi.org/10.1007/978-3-642-39062-3_55>. ISBN 978-3-642-39061-6.

[25] Model-view-controller. *In: Wikipedia: the free encyclopedia* [online]. Wikimedia Foundation, 2014. [cit. 9. 12. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Model-view-controller>.

[26] MYERS, B. A. – ROSSON, M. B. Survey on User Interface Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, s. 195–202, New York, NY, USA, 1992. ACM. doi: 10.1145/142750.142789. Dostupné z: <http://doi.acm.org/10.1145/142750.142789>. ISBN 0-89791-513-5.

[27] PHANOURIOU, C. *UIML: A Device-Independent User Interface Markup*. PhD thesis, Virginia Polytechnic Institute and State University, 2000.

[28] RAMACHANDRAN, K. *Adaptive user interfaces for health care applications* [online]. IBM, 2014. [cit. 9. 12. 2014]. Dostupné z: <http://www.ibm.com/developerworks/library/wa-uihealth/>.

[29] SLOVÁČEK, V. *UIProtocol specification draft 8*. CTU in Prague, 2009.

[30] *Unified Code Count*. USC Center for Systems and Software Engineering, 2014. Dostupné z: <http://csse.usc.edu/ucc_wp/>.

Příloha A

Seznam použitých zkratek

AF AspectFaces

AOP Aspektově-orientované programování

AUI Adaptivní uživatelské rozhraní

AUIP Abstraktní uživatelské rozhraní UIP

BRMS Business Rule Management System

CSS Cascading Style Sheets

CUI Konkrétní uživatelské rozhraní

DHTML Dynamické HTML

DOM Document Object Model

DRL Drools Rule Language

DSL Doménově specifický jazyk

EL Expression Language

GUI Grafické uživatelské rozhraní

Java EE Java Platform, Enterprise Edition

JSF JavaServer Faces

JSP JavaServer Pages

JSR 94 Java Rule Engine API

LLOC Logical line of code

LOC Line of code

MVC Model-view-controller

READ Rich entity aspect/audit design

SLOC Source lines of code

UCC Unified Code Count

UIML User Interface Markup Language

UIP User interface protokol

URI Uniform Resource Identifier

XBL EXtensible Bindings Language

XUL XML User Interface Language

ZK Zdrojový kód

Příloha B

Instalační příručka

B.0.1 Softwarové požadavky

- Java JDK 7
- Maven
- JBoss AS 7.1.1.Final
- Mozilla Firefox

B.0.2 Instalace serveru

Stáhněte aplikační server JBoss AS 7.1.1.Final

```
http://jbossas.jboss.org/downloads
```

Po stažení rozbalte a zprovozněte podle návodu:

```
https://docs.jboss.org/author/display/AS7/Getting+Started+Guide
```

Spusťte server s konfigurací standalone-full

```
Linux, OS X:    standalone.sh -c standalone-full.xml
Windows:        standalone.bat -c standalone-full.xml
```

B.0.3 Kompilace projektu

Přejděte do složky obsahující zdrojové kódy kontextové aplikace. Na DVD se nachází v adresáři:

```
/sources/adaptive-user-interface/
```

Pro zkompilování kontextové aplikace s adaptivním uživatelským rozhraním a její nahrání na spuštěný server použijte níže uvedené Maven příkazy. Vždy se zkompiluje aplikace pouze s jedním UI, které se určí dle maven profilu.

```
jsf:     mvn clean package jboss-as:deploy -P jsf
xul:     mvn clean package jboss-as:deploy -P xul
uiml:    mvn package exec:java -P uiml
uip:     mvn clean package jboss-as:deploy -P uip
```

Sestavená aplikace v podobě EAR archívu se nachází v:

```
ear/target/AdaptiveUI.ear
```

Na DVD se nacházejí již sestavené aplikace. Jsou k nalezení v:

```
jsf:     /build/jsf/AdaptiveUI.ear
xul:     /build/xul/AdaptiveUI.ear
uip:     /build/uip/AdaptiveUI.ear
```

Pro jejich nahrání daný archiv zkopírujte do složky `standalone/deployments` ve složce se serverem.

B.0.4 Spuštění klienta

Po nahrání jednotlivých kontextových aplikací lze spustit jejich klienty dle níže popsaného návodu.

B.0.4.1 JSF

Pro spuštění JSF klienta zadejte do prohlížeče tuto url adresu:

```
http://localhost:8080/aspectfaces-jsf/
```

B.0.4.2 XUL

Požadavkem pro spuštění je nainstalovaný webový prohlížeč Mozilla Firefox. Dále je potřeba aby byl tento prohlížeč obsažen v systémové proměnné PATH. Přejděte do složky se zdrojovým kóde:

```
/sources/adaptive-user-interface/aspectfaces-xul/src/main/xulrunner
```

Uživatelské rozhraní lze spustit pomocí scriptu:

```
Linux, OS X:     ./run-firefox
Windows:         ./run-firefox.bat
```

B.0.4.3 UIP

Pro spuštění lze použít UIP klienta `UIProtocol.dotNET.GenericClient.exe`. Ten se nachází na DVD ve složce:

```
uip/client
```

Připojte se s tímto nastavením:

```
Connect to:    localhost
Elements:      Desktop: cz.ctu.uip.client.wpf.desktop
Client class:  cz.ctu.uip.client.wpf.desktop
User:          User 1
```

Příloha C

Vlastnosti jazyků

```
xulrunner/
├── chrome/ .................................Složka se soubory uživatelského rozhraní
│   ├── content/ .......................Obsahuje XUL soubory definující strukturu UI
│   │   └── root.xul ........................................Definice hlavního okna
│   ├── js/ ...........................................................Složka s JS
│   │   ├── modules/ ........................Složka obsahující JavaScript code moduly
│   │   └── bootstrap.js ........Skript poskytující společné moduly a funkce napříč UI
│   ├── locale/ ....................................Obsahuje složky s lokalizacemi
│   │   └── cs-CZ/ .............................Složka obsahující soubory s lokalizací
│   └── skin/ ..........................................Složka pro různé vzhledy
│       └── base/ ...........Složka konkrétního vzhledu, obsahuje obrázky, CSS a XBL
├── default/
│   └── preferences/
│       └── prefs.js ..............................Specifické nastavení XULRunneru[1]
├── application.ini ................................Obsahuje informace o aplikaci
└── chrome.manifest ................Definuje URI na kterých jsou definované zdroje
```

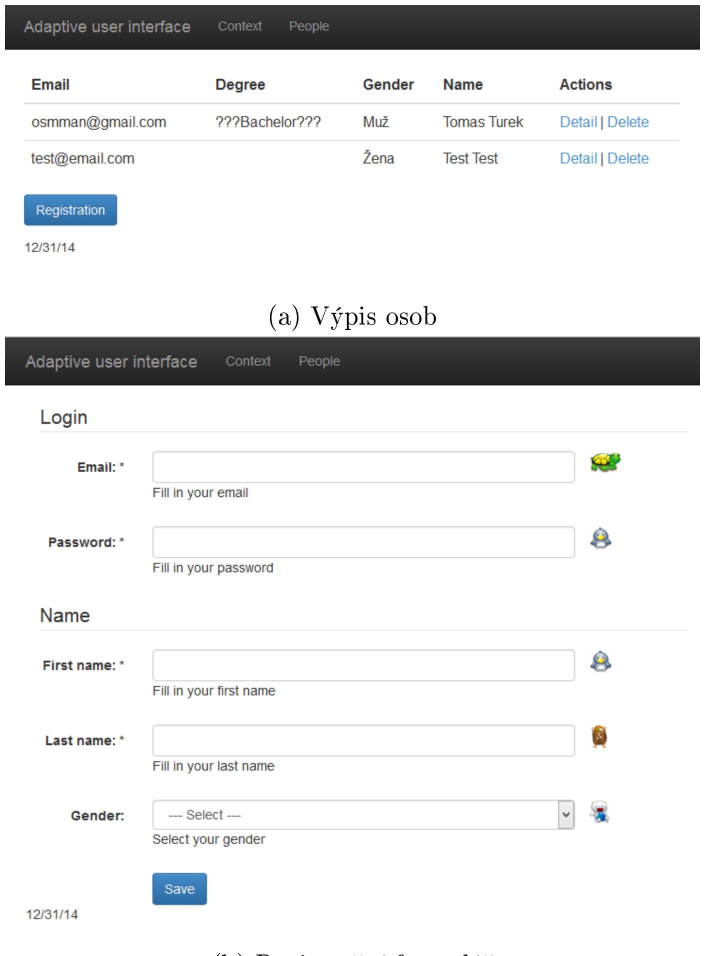
Obrázek C.1: Struktura XULRunner aplikace

Vlastnost	**JSF**	**XUL**	**UIP**	**UIML**
Standardizované	JSR 344	Ne	Ne	OASIS
XML Schema	Ano	Ano	Ano	Ano
Architektura	Component Model	Document Object Model	Client-Server	Meta-Interface Model
Nezávislost na				
Platformě	Ano	Ano	Ano	Ano
Zařízení	Ne	Ne	Ano	Ano
Jazyku UI	částečná (JSP,HTML)	Ne	Ne	Ano
Databinding	Ne	Ano	Ano	Ano
Nástroje pro znovupoužitelnost				
Skládání rozhraní	Composition	Overlays	Nested interfaces	Template
Dynamický obsah	Ano	Ano	Ne (pouze programově)	Ano
Vlastních widgetů	Ano	Ano	Ne	Ano
Oddělení struktury od				
Obsahu	Ne	Ne	Ano	Ano
Vzhledu	CSS	CSS	Ne	Ano
Chování	Ne	Ne	Ne	Ano
Integrace backendu				
Existující řešení	Ano	XUL Template,XulFaces	Ano	Java RMI, EJB
Typová kontrola	Ne	Ne	Ne	Ano (dle finálního jazyka)
Validace	Ano	Ne	Ano	Ne
Konvertory	Ano	Ne	Ne	Ne
Zamezení chyb	Ano	Ne	Ano	Ano
Zpracování chyb	Exception Handlery	—	Standardní eventy	Komponenty logic a behavior
Adaptace				
Povědomí o kontextu	Ne	Ne	Ano	Ne
Lokalizace	Ano	Ano	Ano	Ano
AF generátor	Dynamický	Dynamický	Dynamický	Statický
Techniky	Skrývání obsahu	Overlays, XBL Binding	UiGE	Restructure
Bez úprav ZK	Ne	Ano	Ano	Ne

Tabulka C.1: Porovnání vlastností jazyků pro tvorbu UI

Příloha D

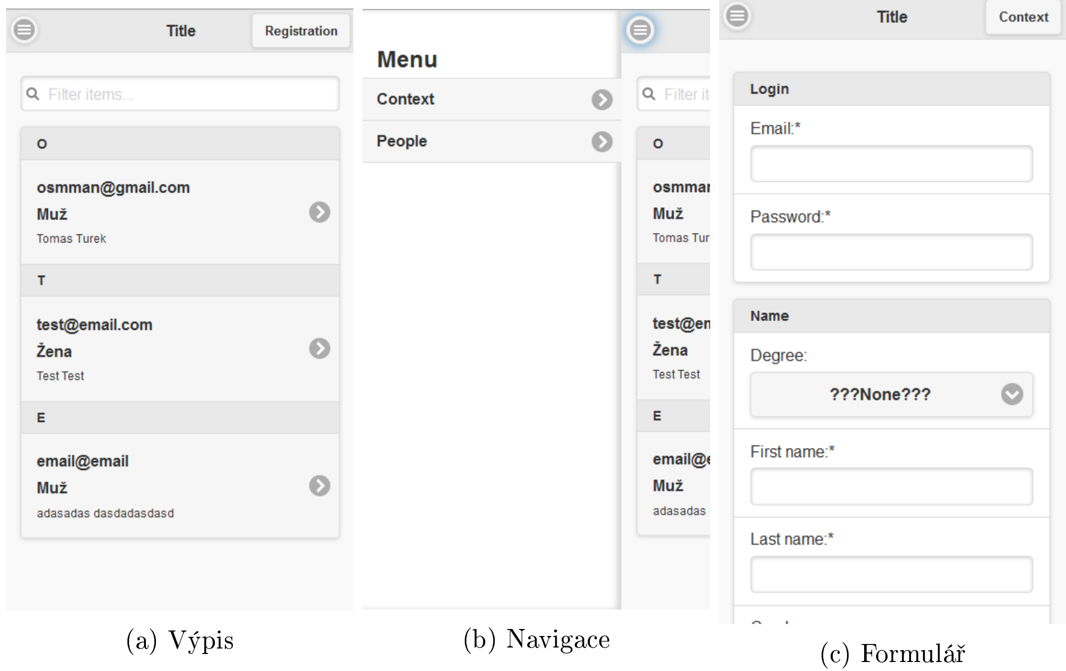
Uživatelská rozhraní



(a) Výpis osob

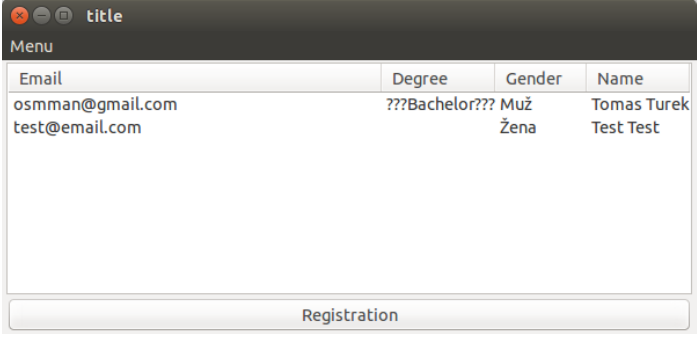
(b) Registrační formulář

Obrázek D.1: JSF - obrazovky UI

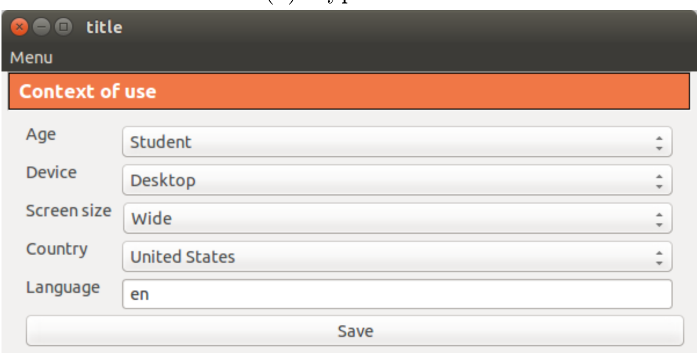


(a) Výpis (b) Navigace (c) Formulář

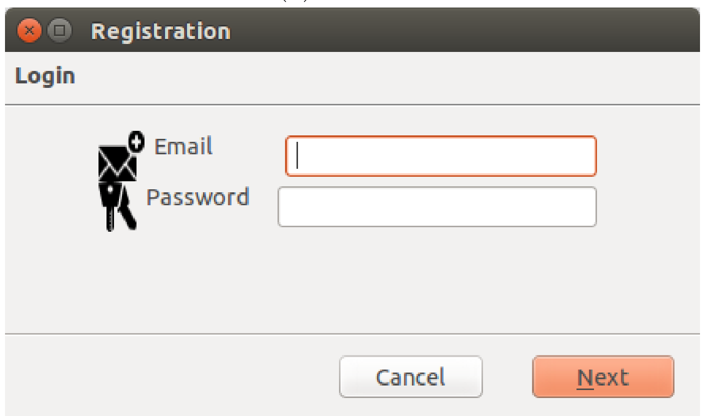
Obrázek D.2: JSF - obrazovky mobilního UI



(a) Výpis osob

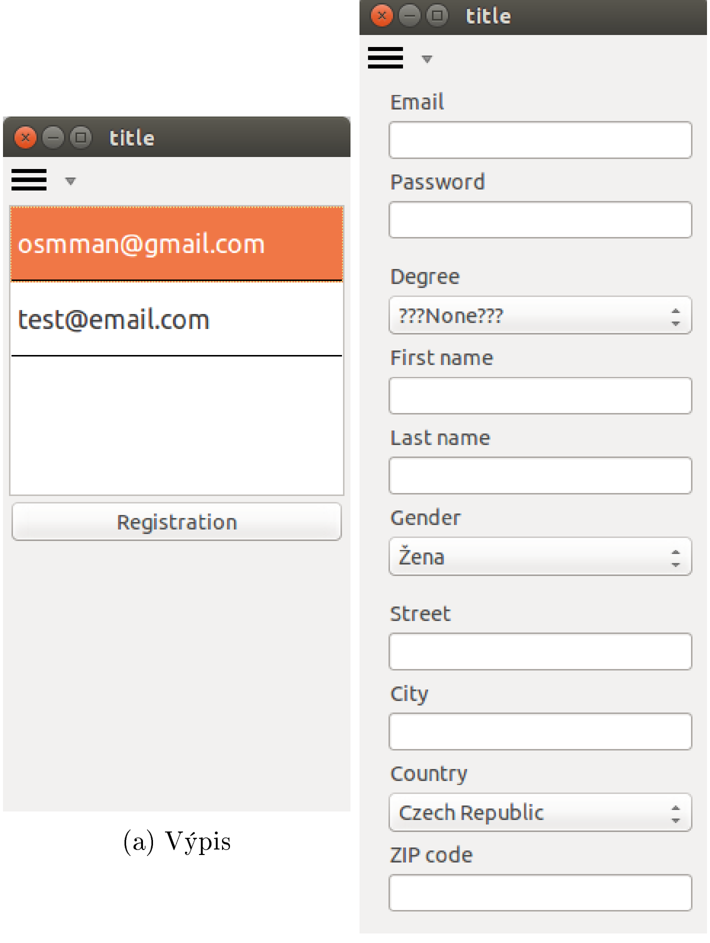


(b) Formulář



(c) Wizard

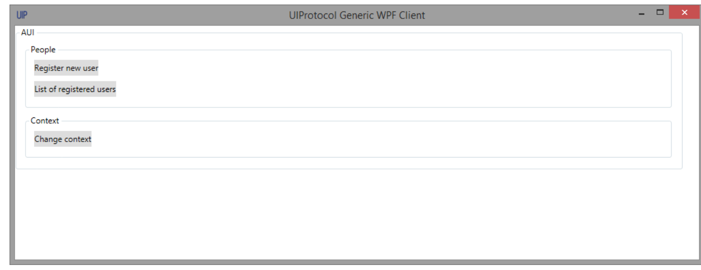
Obrázek D.3: XUL - obrazovky UI



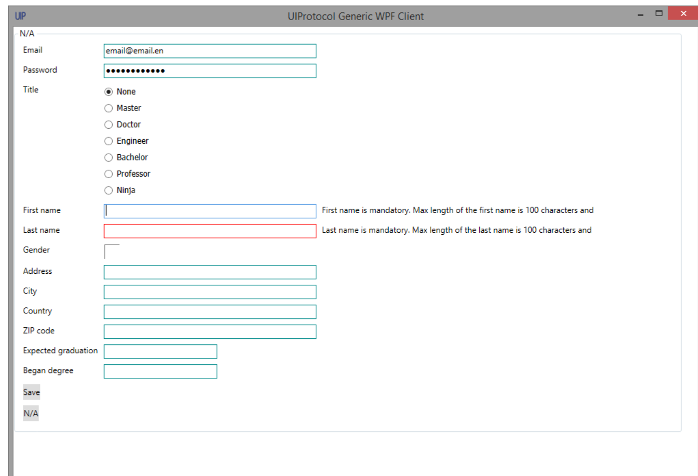
(a) Výpis

(b) Formulář

Obrázek D.4: XUL - obrazovky mobilního UI



(a) Navigace



(b) Formulář

(c) Výpis osob

Obrázek D.5: UIP - obrazovky UI

Příloha E

Obsah přiloženého CD

```
/
├── build/
│   ├── jsf/ ........................ Adresář se sestavenou JSF aplikací
│   ├── uip/ ........................ Adresář se sestavenou UIP aplikací
│   └── xul/ ........................ Adresář se sestavenou XUL aplikací
├── sources/
│   ├── adaptive-user-interface/ .... Zdrojový kód aplikace
│   └── thesis/ ..................... Zdrojový kód textu diplomové práce
├── text/
│   └── thesis.pdf .................. Text diplomové práce
├── uip/
│   ├── client/ ..................... Adresář s UIP klientem
│   └── server/ ..................... Adresář se zdrojovým kódem UIP serveru
├── install.md ...................... Instalační příručka
└── README.TXT
```

Obrázek E.1: Seznam přiloženého CD