ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Generování uživatelských rozhraní pomocí UIP a JPA
Student:	Jindřich Bašek
Vedoucí:	Ing. Miroslav Macík
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství (magisterský)
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2013/14

Pokyny pro vypracování

Nastudujte problematiku automatického generování uživatelských rozhraní z abstraktních modelů, zaměřte se na platformu UIP a generátor UiGE. Analyzujte nástroj JFormBuilder, který umožňuje generování uživatelských rozhraní na základě inspekce kódu.

1) Navrhněte a implementujte překladač z vnitřního formátu nástroje JFormBuilder do abstraktního uživatelského rozhraní platformy UIP (UIP AUI).

2) Na základě provedené analýzy navrhněte možná vylepšení generátoru UiGE a implementuje jeho ekvivalent v jazyce Java.

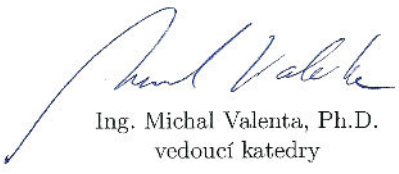
3) Navrhněte a implementujte integraci vzniklého generátoru s implementací UIP serveru v jazyce Java.

4) Navrhněte a implementuje propojení Java UIP serveru a Java aplikací využívajících JPA na datové úrovni.

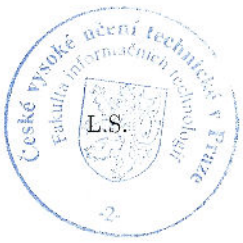
Funkčnost implementovaného řešení prokažte na příkladu vstupní JPA aplikace, která obsahuje alespoň pět entit. Vzniklé řešení otestujte pomocí standardních metod, zejména pomocí testování jednotek (unit testing) a optimalizujte výkon pomoci profilingu.

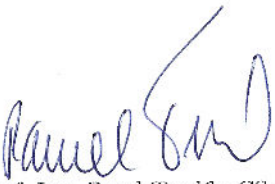
Seznam odborné literatury

Dodá vedoucí práce



Ing. Michal Valenta, Ph.D.
vedoucí katedry





prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 14. ledna 2013

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Generování uživatelských rozhraní pomocí UIP a JPA

Bc. Jindřich Bašek

Vedoucí práce: Ing. Miroslav Macík

8. května 2013

Poděkování

Touto cestou bych chtěl poděkovat panu Ing. Miroslavu Macíkovi za vedení a cenné rady při přípravě diplomové práce a panu Ing. Tomáši Černému za pomoc s porozuměním nástroji AspectFaces. Také bych chtěl poděkovat svým nejbližším, za jejich podporu a pochopení 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 etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Podbrdech dne 8. května 2013 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Bašek, Jindřich. *Generování uživatelských rozhraní pomocí UIP a JPA*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2013.

Abstract

This diploma thesis deals with combination of two approaches to the automatic user interfaces (UIs) generation. The first approach is to generate UIs on the basis of code inspection (using JPA entities) represented by tool AspectFaces (formerly JFormBuilder). The second approach is to generate concrete UIs (CUIs) from abstract UIs (AUIs). An AUI describes UI in a platform independent matter. The CUI generation process reflects context, in particular user capabilities and characteristics of target devices. It is represented by UI generator (UiGE) that is part of the UIProtocol platform. There is a synergy of combination of UiGE and AspectFaces that has potential of eliminating their disadvantages as individuals. Using this approach AspectFaces can generate context sensitive UI for various heterogeneous platforms. On the other hand, UiGE with UIProtocol is enhanced by a powerful input model - enhanced data persistence model. The solution is integrated into the Java version of the UIProtocol platform.

Keywords automatic UI generation, JPA, code inspection, UIProtocol, AspectFaces, UiGE, UIPServer, integration, Java EE

Abstrakt

Tato diplomová práce se zabývá kombinací dvou přístupů k automatickému generování uživatelských rozhraní (UI). První přístup ke generování UI je založen na inspekci kódu (inspekci JPA entit). Reprezentovaný je nástrojem AspectFaces (dříve JFormBuilder). Druhý přístup je generování konkrétních UI (CUI) z abstraktních UI (AUI). AUI popisuje UI ve své platformně nezávislé podstatě. Proces generování CUI reflektuje kontext - zejména schopnosti uživatele a vlastnosti cílového zařízení. Druhý přístup je reprezentovaný UI generátorem (UiGE), který je součástí platformy UIProtocol. Je možná kombinace UiGE a AspectFaces, která má poteciál odstranit jejich nevýhody jako samostatných nástrojů. Díky tomuto přístupu může AspectFaces generovat kontextově citlivá UI pro různorodé platformy. Na druhé straně UiGE s UIProtocolem je vylepšeno silným vstupním modelem - rozšířeným persistentním datovým modelem. Řešení je integrováno do Java verze platformy UIProtocol.

Klíčová slova automatické generování UI, JPA, inspekce kódu, UIProtocol, AspectFaces, UiGE, UIPServer, integrace, Java EE

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Nástroj ApspectFaces [8] (dříve JFormBuilder) umožňuje generovat uživatelská rozhraní na základě vstupního modelu, který vzniká inspekcí rozšířeného datového modelu aplikace. Datovým modelem je myšlen datový model Java aplikace realizovaný za využití technologie Java Persistence API (JPA[1]).

Nástroj UiGE UI Generator[23] (UiGE), který je součástí platformy UIP založené na UIProtocolu, umožňuje generovat konkrétní uživatelská rozhraní (CUI) z abstraktních uživatelských rozhraní (AUI) v závislosti na schopnostech a potřebách uživatele, který bude s CUI interagovat a na možnostech koncového zařízení, na kterém bude CUI zobrazeno. popisuje uživatelské rozhraní ve své platformě nezávislé podstatě. Udává, co bude součástí uživatelského rozhraní, ale již nespecifikuje, jak to bude prezentováno.

V [21] je představena možnost integrace těchto dvou nástrojů a generování automaticky přizpůsobitelných uživatelských rozhraní z datového modelu. AspectFaces generuje AUI z datového modelu Java aplikace a UiGE z něj vytváří CUI, které je doručeno uživateli.

Diplomová práce Generování uživatelských rozhraní pomocí UIP a JPA se zabývá návrhem a implementací integrace těchto dvou nástrojů a jejich začleněním do Java verze UIP serveru (pojmenovaného UIPServer), jak byl představen v [2].

[1] Java Persistence API je API programovacího jazyka Java, které umožňuje popis datových entit pomocí anotací a specifikuje způsob uložení těchto entit v relační databázi - objektově relační mapování [31]. Konkrétní realizací tohoto API je například nástroj Hibernate [17].

Motivace

AspectFaces umožňuje významně zkrátit dobu vývoje aplikace a finanční prostředky vynaložené na vývoj aplikace [6]. Z informací zachycených v datovém modelu aplikace generuje uživatelská rozhraní aplikace jako formuláře a tabulky. Vývojář tak nemusí ručně vytvářet podstatnou část uživatelského rozhraní, zejména formuláře a tabulky. Vyhne se při vytváření uživatelského rozhraní i opakovanému zadávání informací obsažených v datovém modelu, jako je například validace dat. AspectFaces však neumožňuje dynamicky generovat uživatelské rozhraní v závislosti na schopnostech a preferencích uživatele a možnostech cílové platformy.

Toto však umožňuje nástroj UiGE. Na druhou stranu specifikaci AUI, ze které UiGE generuje CUI, není tak snadné a pohodlné vytvořit a mohou se v ní opakovat informace již obsažené v datovém modelu aplikace.

Nástroj AspectFaces proto bude využit ke generování AUI z datového modelu aplikace a z něj UiGE vygeneruje CUI. Takto by mělo dojít ke spojení výhod obou nástrojů, které používají rozdílný přístup ke generování uživatelských rozhraní, a odstranění nevýhod při jejich použití.

Motivací pro spojení nástrojů AspectFaces a UiGE může být například vývoj aplikace, která bude sloužit pro vyplnění skupiny dotazníků. Aplikace bude přístupná pro velké množství různých platforem a zařízení. Očekává se, že uživatelé aplikace budou různého věku a různých schopností, co se týče interakce s výpočetní technikou. Někteří uživatelé mohou být i tělesně postižení. Každý dotazník bude vždy odpovídat jedné nebo několika málo entitám. Pro vyplnění každého dotazníku bude sloužit jeden formulář. Spojení nástrojů AspectFaces a UiGE umožní výrazně zkrátit a zjednodušit vývoj takové aplikace. AspectFaces vygeneruje pro každý dotazník, z entit, které jej reprezentují, jedno AUI. Není tak nutné opisovat informace obsažené v entitách dotazníků při tvorbě UI. Vygenerované AUI představuje abstraktní popis rozhraní sloužícího k vyplnění formuláře. UiGE z AUI vygeneruje CUI podle cílového zařízení a možností uživatele. Není tak nutné složitě vytvářet a optimalizovat UI pro každé zařízení a zohledňovat v nich schopnosti jednotlivých uživatelů.

Díky plánovanému propojení aplikačního serveru pro platformu UIProtocol - UIPServeru a Java aplikace využívající JPA získá platforma UIProtocol doposud citelně chybějící komplexní podporu persistence dat, se kterou je možné ukládat data do relačních i jiných databází.

Cíle práce

Cílem práce je nastudovat problematiku generování uživatelských rozhraní z abstraktních modelů a analyzovat nástroj UiGE implementující tento princip, který je součástí .NET verze platformy UIProtocol. Dále analyzovat nástroj AspectFaces, který umožňuje generování uživatelských rozhraní na základě inspekce kódu, a navrhnout integraci těchto dvou nástrojů do Java verze platformy UIProtocol. V rámci práce také vznikne ukázková aplikace, která přiblíží možnosti automatického generování uživatelských rozhraní s pomocí nástrojů UiGE a AspectFaces.

Po analýze zadání je cíle práce možné shrnout do následujících bodů:

- Navrhnout a implementovat generátor AUI využívající nástroj AspectFaces.
- Navrhnout a implementovat ekvivalent .NET verze UiGE v jazyce Java.
- Navrhnout a implementovat možná vylepšení Java verze UiGE, a to zejména v oblasti zrychlení generování Concrete User Interface - konkrétní uživatelská rozhraní (CUI).
- Upravit UIPServer tak, aby byl připraven na rozšiřování funkčnosti pomocí externích modulů.
- Integrovat Java verzi UiGE do UIPServeru jako externí modul.
- Navrhnout a implementovat propojení mezi UIPServerem a Java aplikací využívajících JPA. JPA aplikace bude v tomto kontextu Java EE aplikace nasazená na Java EE aplikačním serveru, který splňuje specifikaci Java EE 6 Full profile [29].
- Vytvoření ukázkové JPA aplikace, na které bude možné prezentovat funkčnost a schopnosti vytvořeného řešení.
- Profiling naimplementovaného řešení a návrh možných vylepšení na základě jeho výsledků.

V rámci práce bude také provedeno experimentální vyhodnocení výkonu UiGE a částí, které budou upraveny na základě provedeného profilování výkonu. Bude srovnán výkon před úpravami a po úpravách.

Struktura práce

Tato práce se skládá z následujících částí:

Kapitola 1 - Analýza představí nástroje AspectFaces a UiGE a principy a technologie, které využívají při generování uživatelských rozhraní. Bude analyzována stávající implementace UIPServeru. V kapitole je diskutováno možné řešení jednotlivých částí systému a použití technologií vhodných pro implementaci. Na základě diskuze je zvoleno řešení.

Kapitola 2 - Návrh popisuje návrh jednotlivých částí systému a jejich propojení. Návrh bere v potaz zvolené řešení a technologie z kapitoly Analýza.

Kapitola 3 - Realizace popisuje implementaci zajímavých či obtížně implementovatelných částí systému vzhledem k použitým technologiím. V závěru kapitoly je popsána ukázková testovací aplikace.

Kapitola 4 - Testování popisuje metody testování použité při vývoji systému. Pomocí profilování je vyhodnocen výkon částí systému. V kapitole je popsáno provedení a výsledky experimentálního vyhodnocení výkonu UiGE a částí upravených na základě profilování výkonu.

V závěru budou shrnuty výsledky práce a zhodnoceno splnění zadání této práce. Bude představeno budoucí směřování práce.

Kapitola 1

Analýza

Kapitola se zabývá analýzou nástrojů AspectFaces a UiGE určených k automatickému generování uživatelských rozhraní. U UiGE bude analyzována nynější implementace v jazyce C#. Nedílnou součástí kapitoly je také analýza technologie UIProtocol a její části sloužící k popisu Abstract User Interface - abstraktní popis uživatelských rozhraní (AUI). V kapitole jsou diskutovány možnosti integrace Java EE aplikace využívající AUI spolu s UIPServer. Jsou zde také diskutovány technologie vhodné pro realizaci integrace.

1.1 Opakování informace v kódu

Jak je uvedeno v [6], i když se v dnešní době dbá při vývoji složitých informačních systému na oddělení prezentační vrstvy aplikace, vrstvy business logiky a vrstvy datového modelu aplikace a persistence dat a využívá se objektového principu návrhu aplikace, přesto dochází k opakování informací, které jsou zaneseny v jedné vrstvě aplikace ve vrstvách ostatních.

K této duplikaci dochází zejména při tvorbě uživatelského rozhraní aplikace. Velká část entit v datovém modelu aplikace má svůj obraz i v uživatelském rozhraní aplikace v podobě formulářů, tabulek - seznamů entit a výpisech jednotlivých entit. Vývojář uživatelského rozhraní musí znovu uvádět informace, jako jsou podmínky validace dat, které jsou však již ve většině případů uvedeny v nejnižší vrstvě aplikace - v datovém modelu v jednotlivých entitách. Pokud je upravena nižší vrstva aplikace, například je přidán nový atribut entity nebo upraveny podmínky validace dat, musí se tato informace propagovat i do vyšších vrstev aplikace. Tím se stává údržba aplikace i při malé změně datového modelu časově náročnou zále-

žitostí. Vzniká při ni také větší pravděpodobnost zanesení chyby a vzniku nekonzistencí mezi jednotlivými vrstvami aplikace.

1.1.1 Odstranění duplicit

Nabízí se několik možností, jak toto opakování informace při návrhu a vývoji odstranit [6]. Jednou z možností je využití principu aspektově orientované programování (AOP) [1]. Zjednodušeně, vývojář vytváří nezávislé fragmenty kódu, aspekty. Aspekty se integrují do základního programu v době sestavování kódu pomocí tzv. přípojných bodů. Tím se dá předejít opakování informace v jednotlivých vrstvách aplikace a také elegantněji začlenit do aplikace částí, které prostupují všemi vrstvami aplikace, např. bezpečnost nebo logování.

Další možností je využití principu Model driven architecture - Modelem řízená architektura (MDA). Základem vývoje aplikace jsou v tomto případě modely, například UML[2], které zachycují informace o aplikaci. Z těchto modelů je poté generován kód aplikace specifický pro určitou platformu. V [12] jsou využity UML profily pro zachycení informací nutných pro správné vygenerování uživatelského rozhraní. Tímto způsobem je možné generovat s využitím UML modelů i uživatelské rozhraní aplikace.

Oba tyto principy mají velkou nevýhodu v tom, že je není možné jednoduše aplikovat na existující aplikaci. Aplikaci by bylo nutné složitě refaktorovat, případně provádět pro MDA i reverse engineering aplikace, pro zjištění její kompletní funkčnosti.

Těžit z výhod předchozích přístupů a odstranit nutnost rozsáhlé změny architektury aplikace se snaží třetí možný přístup - strojová inspekce zdrojového kódu aplikace [6]. Na základě informací zjištěných při inspekci je možné poté automaticky generovat části uživatelského rozhraní nebo kód aplikace. Tento princip používá i nástroj AspectFaces.

1.2 AspectFaces

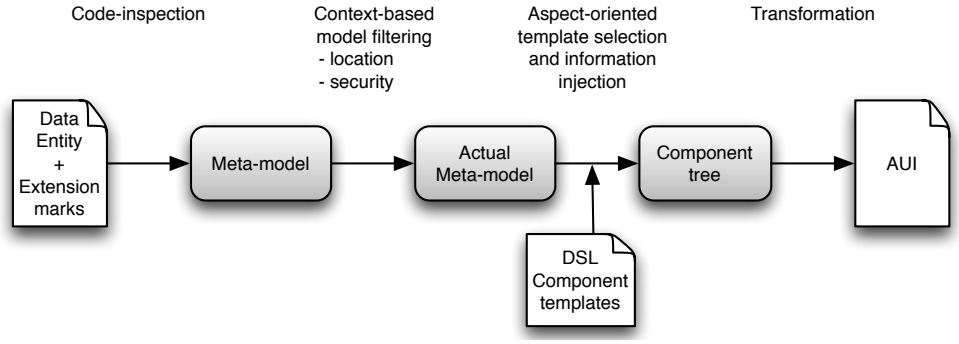
Vstupní část AspectFaces provádí inspekci kódu aplikace. Momentálně existují dvě implementace vstupní části [6]. Jedna je schopná získat informace o datovém modelu z XML. Tato je používána pro inspekci UML modelů, jak

[2]Unified Modeling Language (UML) je grafický jazyk určený k vizualizaci, návrhu a specifikaci programových systémů [26]

bylo popsáno v předchozí kapitole. Druhá implementace provádí inspekci Java tříd pomocí reflexe[3].

1.2.1 Inspekce JPA entit

AspectFaces využívá entity datového modelu, nejnižší vrstvu aplikace, jako místo, kde může být uložena většina informací potřebných pro vygenerování výsledného uživatelského rozhraní. Tyto informace jsou zachyceny v entitách převážně pomocí anotací nad jednotlivými atributy entit, jménech atributů a jejich datových typech.



Obrázek 1.1: Zjednodušený pohled na průběh inspekce kódu JPA entity pomocí AspectFaces (převzato z [21])

Nejnovější verze JPA zachycují informace a omezení datového modelu v entitách - Java třídách, které představují entity, pomocí anotací. Pomocí JPA anotací je možné vyjádřit například vztahy mezi entitami nebo vyjádřit, které atributy entity budou primárním klíčem. Mimo JPA anotací dokáže aktuální verze AspectFaces provádět inspekci anotací typu Hibernate Validation a Javax Validation. Pomocí těchto anotací je možné určit například, jestli atribut entity může nabývat hodnoty null nebo z jakého rozsahu může být celočíselný atribut [14] [33].

Pomocí těchto anotací však není možné zachytit všechny důležité vztahy, které jsou nutné pro vygenerování smysluplného uživatelského rozhraní. AspectFaces proto definuje vlastní sadu anotací. Pro odpovídající funkčnost AspectFaces musejí být JPA entity dekorovány příslušnými rozšiřují-

[3]Reflexe je schopnost počítačového programu, potažmo programovacího jazyka, procházet vlastnosti a meta-data a modifikovat strukturu a chování tříd a z nich vzniklých objektů za běhu programu [39].

cími anotacemi. Jednou z vlastností, kterou není možné bez anotací zachytit, je pořadí, v jakém se mají do výsledného vygenerovaného uživatelského rozhraní promítat jednotlivé atributy entity. Atributy Java třídy jsou totiž v meta-modelu Javy, popisujícího jednotlivé třídy, uloženy v množině (setu), která nedefinuje pořadí prvků v ní obsažených [28]. Pro určení pořadí atributů v entitě AspectFaces proto používá anotaci `@UiOrder`. Další z anotací, které AspectFaces poskytuje jsou například:

- `@UiIgnore` - atribut s touto anotací bude ignorován a jemu odpovídající tag se neobjeví ve výsledném uživatelském rozhraní.
- `@UiParam` - umožňuje přidat do meta-modelu vzniklého inspekcí JPA entit parametr pro atribut entity, který může ovlivnit výběr, případně vzhled tagů, na který bude atribut namapován.
- `@UiUserRoles` - umožňuje určit, jakou roli musí mít uživatel, pro kterého se generuje rozhraní, aby se uživateli zobrazila část rozhraní příslušející atributu, u kterého se anotace nachází.

Seznam všech podporovaných anotací je k dispozici [9].

1.2.2 Metamodel

Inspekcí entity nebo XML souboru vzniká meta-model, který je zpracováván v dalších fázích transformace modelu na uživatelské rozhraní pomocí AspectFaces.

Pro každou anotaci, musí existovat tzv. Anotation Descriptor. Tento musí být před použitím registrován v nástroji AspectFaces. Anotation Descriptor udává, jaké atributy anotace se budou uvažovat při inspekci entity a do jakého parametru v meta-modelu se hodnoty atributů anotace uloží.

Parametry vytvořené Anotation Descriptory jsou v meta-modelu svázány s jednotlivými atributy inspektované entity. Pro každý inspektovaný 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 - nebude obsahovat editační ovládací prvky. Další nezávislou informací jsou jména inspektovaných tříd.

1.2.3 Mapování a tagy

Výsledné uživatelské rozhraní se skládá z jednotlivých fragmentů - tagů. Tag je textový soubor, který může obsahovat příkazy zapsané pomocí Java Unified Expression Language (EL)[4], které jsou v průběhu transformace vyhodnoceny a případně nahrazeny parametry z meta-modelu. Tag představuje například jedno tlačítko ve webovém formuláři - tag v syntaxi JavaServer Faces nebo definice abstraktního ovládacího prvku v XML syntaxi UIProtocolu pro popis AUI.

To, jaké tagy se použijí pro jednotlivé atributy inspektované entity, se určuje pomocí tzv. mapování. Mapování je XML konfigurační soubor, ve kterém se specifikuje datový typ atributu a jemu příslušející tag, na který bude namapován. Datovým typem se myslí jméno primitivního typu jazyka Java nebo simple name[5] třídy jazyka Java. Uvedené jméno třídy, musí být konkrétním jménem třídy - inspektované JPA entity. AspectFaces momentálně nepodporuje mapování podle jména rozhraní, které entita implementuje nebo předka, z kterého entita dědí. Toto platí s jednou výjimkou. Všechny výčtové typy (enumy) jazyka Java jsou vždy mapovány podle simple name předka všech enumů, třídy `java.lang.Enum`.

Pro každý datový typ je vždy uvedeno jedno výchozí mapování. Pro každý datový typ mohou být specifikována i vedlejší mapování. Použití vedlejšího mapování je podmíněno splněním podmínky zapsané v konfiguračním souboru mapování pomocí EL. Toto umožňuje mapovat jeden datový typ na více druhů ovládacích prvků. Např. `String` se normálně mapuje na obyčejné pole pro vkládání textu, když je ale nad příslušným atributem v entitě umístěna anotace `@Email`, což je možné zjistit pomocí EL v konfiguraci mapování, mapuje se `String` na pole pro vkládání emailu.

K jednomu atributu entity je možné do výsledného uživatelského rozhraní vložit až tři tagy. K jednomu hlavnímu, určenému podle mapování podle datového typu atributu a dalších podmínek, je možné pomocí anotací `@UiBefore` a `@UiAfter` nad atributem přiřadit další dva tagy. Tyto se vloží ve výsledném uživatelském rozhraní před a za hlavní. `@UiBefore` a `@UiAfter` se chovají vzhledem ke konfiguraci mapování jako další dva atributy entity s datovým typem `Insert`. Díky tomu je možné v konfiguraci

[4] Java Unified Expression Language (EL) je speciální jazyk, který je součástí specifikace Java Server Pages. Používá se k vyhodnocování výrazu v textových souborech - nejčastěji JSP stránkách. Je však univerzální a může být použit i v kombinaci s jinými technologiemi než JSP [19]

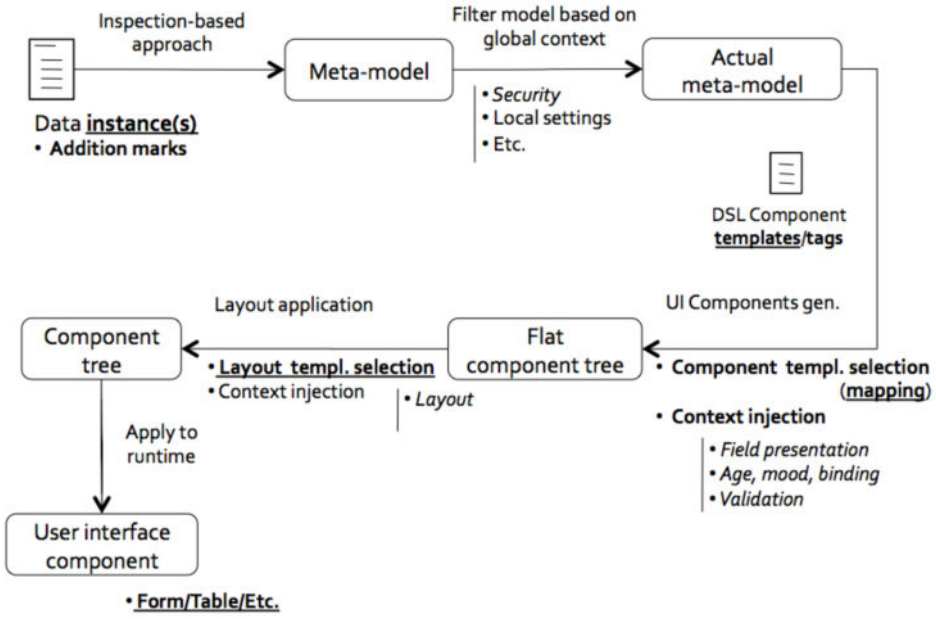
[5] Simple name je jméno třídy v jazyce Java bez specifikace balíčku a dalších modifikátorů. Např. simple name třídy java.lang.Enum je Enum.

mapování určit, na jaké tagy se namapují.

Může být vytvořeno více konfiguračních souborů s mapováními a více sad tagů. Při spouštění transformace v AspectFaces je nutné specifikovat, jaké mapování a sada tagů se při transformaci použijí. Díky tomuto je možné v jedné aplikaci, která používá AspectFaces, generovat uživatelská rozhraní pro více cílových technologií pro jednu a tu samou entitu. Např. jedno webové rozhraní v syntaxi Facelets[6] a druhé abstraktní uživatelské rozhraní pro UiGE v syntaxi UIProtocolu pro popis AUI.

1.2.4 Průběh transformace

Průběh transformace je znázorněn na obrázku 1.2. Bude popsán průběh transformace pro jednu JPA entitu. Proces transformace ale může být spuštěn i pro více entit najednou. Ze všech těchto entit je vygenerováno jedno uživatelské rozhraní.



Obrázek 1.2: Průběh transformace inspektovaných informací na uživatelské rozhraní (převzato z [9])

[6]Facelets je template framework určený k vytváření webových stránek na platformě Java EE, je součástí frameworku JSF [18].

V první fázi je analyzována entita a všechny její anotace, jak je popsáno v kapitole 1.2.1. Všechny získané informace jsou uloženy do meta-modelu popsaného v kapitole 1.2.2.

V další fázi je meta-model filtrován. Při spouštění transformace mohou být některé atributy entity označeny jako ignorované. Tyto se v této fázi odfiltrují a nebudou se již dále při transformaci uvažovat. Jednotlivé atributy entity mohou mít pomocí anotace `@UiUserRoles` nastaveno, jakou roli musí mít uživatel, pro kterého se uživatelské rozhraní generuje, aby se mu část rozhraní představovaná tímto atributem zobrazila. Při spouštění transformace je možné specifikovat, k jakým rolím uživatel patří. Pokud uživatel nepatří k požadované roli, je atribut odfiltrován.

Nastavení, jako jsou role uživatele, ignorované atributy a meta-model, jsou uloženy v tzv. kontextu. Přes kontext přistupují k jednotlivým parametrům meta-modelu skripty zapsané pomocí EL v jednotlivých tazích. Do kontextu mohou být vloženy nové parametry přístupné z EL. Těmito parametry mohou být instance jakékoliv třídy. Díky tomu je možné z EL volat jakékoliv programové konstrukce naprogramované v Javě. Takto je možné například získávat texty z resource bundle[7], které se mají zobrazit ve výsledném uživatelském rozhraní.

Tímto způsobem by bylo možné použít AspectFaces ke generování nejen textové reprezentace uživatelského rozhraní, ale i jeho objektové formy. Jako parametr by se do kontextu vložila instance třídy implementující návrhový vzor builder[8]. Tato třída by sloužila ke stavbě objektové reprezentace uživatelského rozhraní. V jednotlivých tazích by se poté pomocí EL volala příslušná metoda, která by vytvořila objektovou reprezentaci části uživatelského rozhraní.

Nyní je meta-model připraven na mapování, jak je popsané v kapitole 1.2.3. Ke každému atributu entity v meta-modelu je podle mapování přiřazen příslušný tag. Následně se vyhodnotí všechny EL skripty v namapovaných tazích.

Vzniklé fragmenty budoucího uživatelského rozhraní se uloží do stromu komponent v pořadí, jaké bylo uvedeno u každého mapovaného atributu entity pomocí anotace `@UiOrder` (viz. 1.2.1).

Nyní se mohou komponenty umístit ze stromu do layoutu. Layout je tex-

[7]Resource bundle je objekt nebo textový soubor, který obsahuje objekty specifické pro určitou lokalizaci, např. lokalizované textové řetězce [27].

[8]Builder je objektový návrhový vzor. Účelem tohoto návrhového vzoru je abstrahovat proces konstrukce složitého objektu nebo skupiny objektů, tak aby bylo možné konstruovat různé implementace těchto objektů [11].

tový soubor s libovolným obsahem, například kostra HTML dokumentu, který obsahuje speciální značky. Tyto značky jsou definované nástrojem AspectFaces. Pomocí značek je možné určovat umístění fragmentů budoucího uživatelského rozhraní v layoutu. Layout je možné specifikovat při spuštění transformace. Pokud není layout specifikován, fragmenty se umístí do uživatelského rozhraní v tom pořadí, v jakém jsou uloženy ve stromu komponent. Nakonec se na začátek a konec vygenerovaného rozhraní umístí tzv. hlavička a patička, pokud toto bylo povoleno při spuštění transformace. Hlavičkou a patičkou mohou být např. uvozující a zakončující tagy HTML dokumentu.

Již hotové uživatelské rozhraní je možné uložit do souboru nebo rovnou postoupit k dalšímu zpracování již mimo nástroj AspectFaces a zobrazit rozhraní uživateli.

1.2.5 Strategie generování

Pro jednu entitu může existovat více formulářů, uživatelských rozhraní. Uživatelé s rozdílnými oprávněními vidí různé položky formuláře, může existovat varianta formuláře pro editaci entity, pro její prohlížení. [12] uvádí způsoby, jak různé varianty uživatelského rozhraní generovat.

První možností je vygenerovat formuláře staticky, dopředu, při sestavování aplikace. Je možné buď vygenerovat zvlášť formulář pro každou variantu možného zobrazení, nebo vygenerovat jeden formulář pro entitu, který bude obsahovat části, které se budou zobrazovat podmíněně v závislosti např. na oprávněních uživatele.

Nevýhoda tohoto přístupu je zřejmá. Kombinací oprávnění, která mohou uživatelé vlastnit, může být velké množství, viz. [12]. Zobrazení jednoho podmíněného formuláře s velkým množstvím podmínek zase může trvat dlouhou dobu kvůli vyhodnocování všech podmínek.

AspectFaces proto přichází s možností dynamického generování rozhraní za běhu programu. Díky tomuto přístupu je možné generovat jednoduchá uživatelská rozhraní bez podmínek a není nutné dopředu generovat velké množství formulářů. [12] ukazuje, že tento přístup nepředstavuje ani výkonnostní problém. Při využití vyrovnávací paměti pro tagy v operační paměti, ve které se ukládají tagy načtené z disku je, doba zobrazení jednoho formuláře asymptoticky stejně dlouhá jako při použití staticky vygenerovaného formuláře bez podmínek.

1.2.6 JSF plugin

AspectFaces poskytuje plugin pro JSF[9]. Díky tomuto plugin je možné v Java aplikaci s uživatelským rozhraním založeným na JSF generovat dynamicky uživatelská rozhraní pomocí AspectFaces. Součásti pluginu, který slouží pro vyvolání automatického generování uživatelského rozhraní a vložení vygenerovaného rozhraní do stránky, viz. [9], je definice tagů pro Facelets.

```
1 <util:ui edit="true"
2     instance="#{selectedPerson},
3     #{person.personInfo}"
4     ignore="password,enabled,dateOfBirth"
5     currentUsername="#{person.username}"
6     renderMiddleName="false"
7     columnLayout="personInfo-layout.xhtml"
8     prefix="pi" />
```

Výpis kódu 1.1: Příklad vygenerování části uživatelského rozhraní pomocí AspectFaces a JSF pluginu na stránce vytvořené v Facelets. Převzato z [9].

1.3 UIProtocol

Analýza UIProtocolu vychází z analýzy provedené v [2]. Navíc je popsána varianta UIProtocolu pro popis AUI. Popis UIProtocolu z [38].

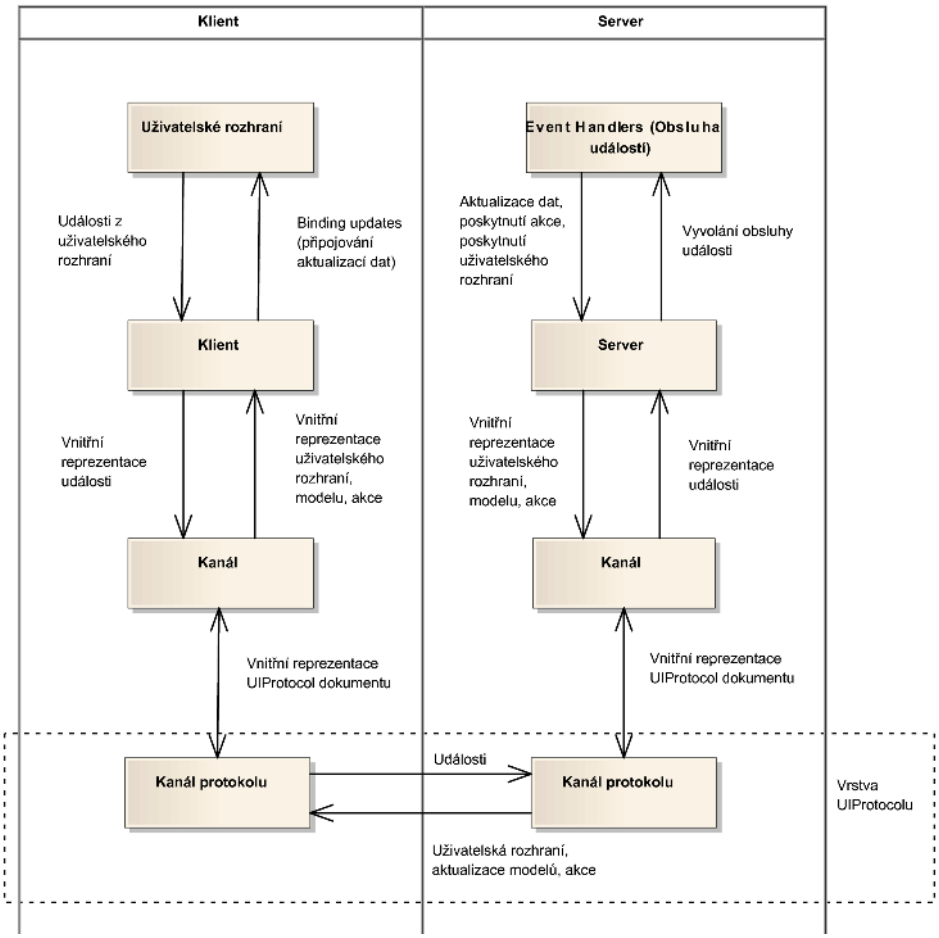
UIProtocol je technologie navržená pro popis uživatelských rozhraní (UI[10]) a pro přenášení tohoto popisu a dat souvisejících s interakcí uživatele s rozhraním mezi uživatelským rozhraním a logikou aplikace, tj. mezi klientem pro UIProtocol a serverem pro UIProtocol. Pro popis rozhraní je používáno primárně XML[11]. UIProtocol je navržen zejména pro aplikační model klient - server. Klient zobrazuje uživatelská rozhraní a komunikuje pomocí UIProtocolu se serverem, který klientovi poskytuje uživatelská rozhraní popsaná v UIProtocolu a obstarává logiku aplikace. UIProtocol je

[9] Java Server Faces (JSF) je webový aplikační framework patřící do platformy Java EE, umožňující vytvářet webová uživatelská rozhraní aplikace [32]

[10] User Interface, uživatelské rozhraní. Uživatelské rozhraní je souhrn způsobů, jakými lidé (uživatelé) ovlivňují chování strojů, zařízení, počítačových programů či komplexních systémů [10].

[11] Extensible Markup Language (XML) je obecný značkovací jazyk. Umožňuje snadné vytváření konkrétních značkovacích jazyků (tzv. aplikací) pro různé účely a různé typy dat [10]

natolik univerzální protokol, že může být použit i k jiným účelům, než jen k doručování uživatelských rozhraní. V UIProtocol serveru implementovaném v jazyce Java je např. použit pro řízení serveru, tj. pro jeho restartování, zastavení atd.



Obrázek 1.3: Referenční architektura systému s UIProtocolem

V základní verzi UIProtocolu jsou mezi klientem a serverem přenášeny XML dokumenty UIProtocolu v textové podobě. Do budoucna jsou plánovány další varianty, například UIProtocol binary, který přenáší data mezi klientem a serverem v binární podobě, nebo UIProtocol ZIP, který data přenáší komprimovaná. Dokumenty jsou přenášeny oběma směry bez potvrzování. Server může odeslat dokument připojenému klientovi kdykoliv bez toho, aby klient před tím o dokument žádal. Klient tak může být kdykoliv notifikován o změně stavu serveru. Nejdůležitější vlastnosti UIProtocolu

XML syntaxe	Mezi serverem a klientem jsou zasílány standardní XML dokumenty, které dodržují danou specifikaci [38].
MVC	Oddělení uživatelského rozhraní, dat aplikace a aplikační logiky.
Data binding (připojování dat)	Když jsou změněny údaje, které jsou zobrazeny na nějakém UI, celé rozhraní nemusí být nahrazeno novým se změněnými daty. Změněné údaje jsou automaticky nahrazeny na všech UI, na kterých jsou zobrazeny bez toho, aby aplikační logika musela vědět, kde se tyto údaje na UI nacházejí.
Snadná rozšiřitelnost	Specifikace UIProtocolu dovoluje přidávání nových vlastností do UIProtocolu bez větších změn specifikace. Také je možné vytvářet vlastní komponenty a vlastnostiUI bez toho, aby tyto kolidovaly se specifikací UIProtocolu.
Platformní nezávislost	UIProtocol není vázán na konkrétní programovací jazyk nebo platformu. Klient i server mohou být implementovány v libovolném programovacím jazyce při dodržení specifikace UIProtocolu.

Tabulka 1.1: Nejdůležitější vlastnosti UIProtocolu

jsou uvedeny v tabulce 1.1. Referenční architektura systému s UIProtocolem je zobrazena na obrázku 1.3.

UIProtocol rozlišuje několik typů objektů:

- concerete interfaces - hierarchický popis struktury CUI, prvků uživatelského rozhraní, jejich vzhledu a umístění.
- abstract interfaces - hierarchický popis struktury AUI a prvků AUI. AUI může existovat pouze na serveru, nikdy nesmí být zasláno klientovi.
- models - data používaná v UIProtocol aplikaci a zobrazovaná na

uživatelském rozhraní. 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 (aplikací), tj. žádost o model, uživatelské rozhraní, akci, chyba atd.

- actions - modifikace modelu nebo uživatelského rozhraní bez toho, aby byla uživatelem vyvolaná událost zpracována aplikační logikou. Tj. modifikace modelu uskutečněná bez součinnosti serveru.

Tyto objekty jsou popsány pomocí XML značek, vloženy do XML dokumentů UIProtocolu a zasílány mezi klientem a serverem. V případě využití UIProtocolu pro doručování popisu uživatelských rozhraní a pro přenášení tohoto popisu a dat souvisejících s interakcí uživatele s rozhraním, klient může serveru zasílat pouze události a v žádném případě nesmí zaslat model, akci nebo uživatelské rozhraní. Server může zasílat modely, akce nebo uživatelská rozhraní a nesmí zasílat události.

```
1  <?xml version="1.0" encoding="UTF-8"?> 2
2  <UIProtocol version="1.0" time="1234567890">
3    <interfaces>
4      <!-- concrete or abstract interfaces
5           definition here -->
6    </interfaces>
7    <models>
8      <!-- models definition here -->
9    </models>
10   <events>
11     <!-- event notifications here -->
12   </events>
13   <actions>
14     <!-- actions definition here -->
15   </actions>
16 </UIProtocol>
```

Výpis kódu 1.2: Ukázka UIProtocol XML dokumentu

1.3.1 Properties

Základním kamenem UIProtocolu jsou tzv. Properties. Properties se používají k popisu vlastností elementů UI (např. vzhledu, pozice), uložení dat v modelech, pro přenos dat v událostech a popis měněných dat v akcích.

Každá Property má svoje jméno, které je jedinečné v rámci elementu, který Property obsahuje. Property může, ale nemusí mít definovánu hodnotu. Pokud není hodnota definována, je považována za nulovou (null nebo nil v některých programovacích jazycích). Property může mít definován tzv. klíč. Klíč odkazuje na jiné Properties uložené v modelech. Pokud je změněna nějaká hodnota Property, na kterou ukazuje klíč, je automaticky aktualizována i hodnota proměnné, která klíč obsahuje. Toto je jedna z klíčových vlastností UIProtocolu umožňující oddělení dat a jejich prezentace. Pokud je Property v modelu změněna, tato změna se projeví automaticky na všech uživatelských rozhraních, která na tuto Property odkazovala.

1.3.2 Modely

Data aplikace jsou v UIProtocolu ukládána v tzv. modelech (Models). Každý model má svůj identifikátor, který musí být specifický v rámci UIProtocol aplikace. Modely obsahují jednotlivé Properties, v nichž jsou uložena data aplikace. Properties mohou odkazovat na jiné Properties pomocí klíče (viz. kapitola 1.3.1).

Jeden model může existovat ve více variantách. Varianty jednoho modelu mají stejný název. Varianty jednoho modelu jsou odlišeny pomocí tzv. Variant Properties. Variant Property je běžná Property, která musí mít specifikován název a hodnotu a nesmí mít uveden klíč. Každá varianta jednoho modelu má poté jednu nebo více Variant Properties. Pokud model nemá uvedenu žádnou Variant Property, jedná se o základní variantu modelu. Každá varianta jednoho modelu může obsahovat jednu nebo více běžných Properties. Tyto Properties mohou mít v jednotlivých variantách modelu stejná jména a většinou se liší jejich hodnoty nebo klíče. Varianty modelů mohou být například použity při lokalizaci aplikace. Pro každou podporovanou lokalizaci některého modelu poté existuje jedna varianta modelu.

Server posílá klientovi aktualizace modelů, pokud jsou na serveru změněna data modelu nebo pokud si klient vyžádá model. Existuje několik variant aktualizací modelu. Varianty aktualizací jsou rozlišeny podle toho, jak se naloží s daty v existujícím modelu, pokud již je tento model existuje.

- complete - všechna existující data v modelu jsou vymazána a jsou

public.models	Seznam modelů známých klientovi
public.application	Informace o UIProtocol aplikaci, ke které je klient připojen (jméno aplikace, autor, uživatelské rozhraní zobrazené při startu UIProtocol aplikace atd.)
public.client	Model udávající klientovi, jak by se měl chovat (např. jakým způsobem vykreslovat uživatelská rozhraní)
public.connection	Stav spojení mezi klientem a serverem
public.error	Informace o chybách zaslaných serverem
public.interfaces	Seznam uživatelských rozhraní (interfaces) známých klientovi
public.server	Informace o serveru, ke kterému je klient připojen

Tabulka 1.2: Nejdůležitější standardní modely

nahrazena nově přijatými daty.

- partial - všechna existující data v modelu jsou ponechána, jsou přidána nová data (Properties) a data (Properties), která již v modelu existují, jsou nahrazena nově přijatými.
- invalidate - všechna data v modelu jsou smazána.
- persistent - stejné jako partial update. Přijatá data jsou klientem zapamatována a jsou poslána na server při příštím připojení k serveru.

Data aktualizovaná v modelu mohou být interpolována. Interpolace probíhá na klientovi. Interpolace umožňuje na klientovi vytvářet animace.

UIProtocol definuje sadu standardních modelů. Standardní modely jsou používány přímo serverem a klientem pro definování vlastností UIProtocol aplikace, klienta, serveru, zařízení připojeným ke klientu atd.

1.3.3 Uživatelská rozhraní

Uživatelská rozhraní (interface) se v UIProtocolu skládají z kontejnerů (container) a elementů (element), společně komponenty. Interface může popisovat celé uživatelské rozhraní nebo jeho část. Jednotlivé Interfaces jsou jednoznačně identifikovány pomocí svojí třídy (class). Interface do sebe mohou

být vnořovány a mohou tak být vytvářeny uživatelské komponenty, které je možné použít na více místech aplikace.

Do interface se vkládají komponenty pomocí tagu element. Každý element má svůj jedinečný identifikátor. Třída elementu určuje typ zobrazeného ovládacího prvku. Třída může odkazovat na uživatelem vytvořenou komponentu (interface) nebo na standardní komponentu. Komponenty jsou vkládány do kontejnerů.

U každé komponenty je možné definovat reakci na různé události vyvolané uživatelem (např. stisk tlačítka, přejetí myší nad ovládacím prvkem, stisk klávesy atd.). Reakce se definují pomocí tagu behavior. To, na jakou událost bude reagováno, se udává pomocí atributu trigger. To, jaká akce bude při události vyvolána, je udáno pomocí atributu action, do kterého se zapisuje identifikátor akce. Potom, co uživatel například stiskl tlačítko, je vygenerována událost (Event) obsahující identifikátor akce zadané v atributu action. Event je odeslána na server ke zpracování. Pokud je klientovi známa action se stejným identifikátorem, jako je zadaný v atributu action, je tato akce na klientovi spuštěna. V případě, že je action označena pro spuštění na straně klienta, neodesílá se Event na server.

Jednotlivé vlastnosti elementů a kontejnerů se určují pomocí Properties. Interfaces existují ve dvou verzích: pro popis CUI a pro popis AUI.

1.3.3.1 Konkrétní uživatelská rozhraní

Pro každý kontejner je možné určit layout, pomocí kterého se budou komponenty uvnitř kontejneru umísťovat. Layout definuje způsob umístění komponent v kontejneru (přesně na dané pozici, seřazené za sebe, v mřížce atd.).

Pro každý element a kontejner je možné určit pozici elementu. Způsob určení pozice závisí na tom, jaký layout používá kontejner, ve kterém je komponenta vložena.

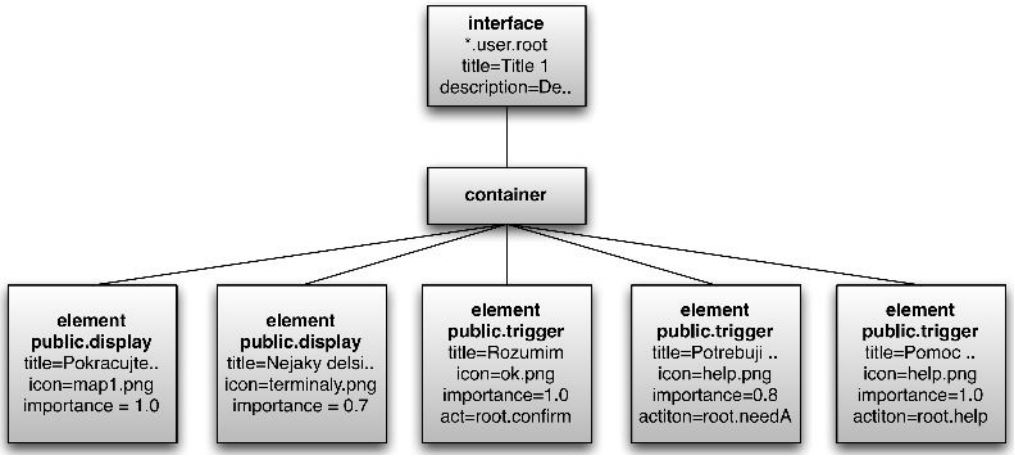
Pro každý element a kontejner je možné definovat styl. Styl určuje vzhled komponenty, např. barvu pozadí, velikost textu, šířku a typ ohraničení.

Konkretní uživatelská rozhraní jsou přímo doručována klientovi, který je vykresluje na obrazovku.

1.3.3.2 Abstraktní uživatelská rozhraní

Abstraktní uživatelská rozhraní (AUI) slouží jako vstup pro UiGE. Jsou z nich generována CUI, viz kapitola 1.5. Popisují, co má být součástí uži-

vatelského rozhraní a jak to má být seskupeno, neudávají však, kde přesně a jakým způsobem to bude zobrazeno. AUI popisují uživatelské rozhraní ve své platformně nezávislé podstatě.



Obrázek 1.4: Příklad stromové struktury AUI (převzato z [23])

AUI má podobný formát jako CUI. Nejsou zde však layouty, styly ani pozice. Místo toho AUI navíc definuje restrictions. To jsou různá omezení, která mohou ovlivnit generování konkrétního uživatelského rozhraní, např. důležitost prvku. Další odlišností proti CUI jsou tzv. labely. Definují popisky na ovládacím prvku. Tj. text, který se zobrazí uživateli jako jméno ovládacího prvku nebo kontextová nápověda, nebo se přehraje, pokud generátor vygeneruje nevizuální uživatelské rozhraní.

Třídy elementů a kontejnerů neudávají konkrétní ovládací prvky. Udávají abstraktní ovládací prvky jako zobrazení (output), vstup (input), akci (trigger), kontejner (container). Některé z nich lze blíže specifikovat - input.text. Není to ale vždy nutné nebo dokonce žádoucí.

Příklad stromové struktury AUI s jedním abstraktním kontejnerem, ve kterém je umístěno několik elementů je na obrázku 1.4. Kontejner může mimo elementů obsahovat kontejnery, ve kterých jsou umístěny další kontejnery a elementy.

Aktuální specifikaci AUI pro UIProtocol je možné nalézt v [24].

1.3.4 Události

Události (Events) slouží pro zasílání informací o uživatelem vykonané akci z klienta na server. Events jsou základem komunikace klienta se serverem. K jaké akci Event přísluší a jaká akce bude vyvolána na serveru po přijetí Event, je definováno pomocí atributu id. Změněná data je možné zasílat v Event uvnitř Properties. Properties v Events nemohou obsahovat atribut klíč (key). V obyčejných Properties v Events mohou být data posílána jednorázově.

Specifikace UIProtocolu definuje sadu standardních Events, používaných k informování serveru o stavu klienta, životním cyklu aplikace na klientu, chybách vzniklých při běhu klienta, připojení zařízení ke klientovi, při žádosti o Interface, Action nebo Model atd.

Nejdůležitějšími jsou Events zasílané při navazování a ukončování komunikace a při žádostech o Interface, Action nebo Model.

Connect Event zasílá klient při připojování k serveru ihned poté, co je otevřen kanál pro komunikaci se serverem. V Event jsou zasílány informace o klientovi, jakou verzi a vlastnosti UIProtocolu klient podporuje a přihlašovací údaje uživatele (uživatelské jméno a heslo). Server po přijetí této události zašle klientovi model public.application s informacemi o UIProtocol aplikaci poskytované serverem (viz. tabulka1.2).

Disconnect Event zasílá klient při odpojování od serveru. Ihned poté, co je Event odeslána, je uzavřen kanál pro komunikaci mezi klientem a serverem.

Request Events zasílá klient při žádosti o Interface, Action nebo Model, jejichž definici klient ještě od serveru nepřijal. Server po přijetí této Event zašle požadovaný objekt klientovi. Pokud požadovaný objekt na serveru neexistuje, zašle server aktualizaci příslušného standardního modelu se seznamem objektů daného typu (public.actions, public.interfaces) a nastaví příslušnou Property v tomto modelu na null.

Events nemusejí být použity jen pro zasílání zpráv o interakci uživatele s uživatelským rozhraním. Existuje varianta UIProtocolu, tzv. Event Protocol [25], který může být použit pro libovolnou komunikaci a řízení součástí platformy UIProtocol. Např. UiGE pomocí něj získává od klienta definice vzhledu konkrétních ovládacích prvků při rozhodování, jakým způsobem vygeneruje CUI. Event Protocol je také využit pro řízení běhu Java verze UIP Serveru, jeho zastavení, restart, atd.

1.3.5 Server

Na serveru jsou uložena veškerá data UIProtocol aplikace. Klient se k serveru připojuje a tato data ze serveru získává. XML dokumenty UIProtocolu jsou serverem poskytovány pomocí TCP protokolu. Mediální soubory jako obrázky a zvuky jsou serverem poskytovány pomocí HTTP[12] protokolu. Server zpracovává příchozí Events od klienta pomocí tzv. Event handlers. Event handlers jsou programy spouštěné na serveru při přijetí Event, zpracovávají přijatou Event a odesílají klientovi zpět výsledky zpracování v podobě aktualizace modelu nebo požadovaný Model, Interface nebo Action. Event handlers jsou programovány vývojářem UIProtocol aplikace.

1.3.6 Klient

Klient zobrazuje uživatelské rozhraní UIProtocol aplikace, stará se o interakci s uživatelem a zasílá Events serveru. Klienti pro UIProtocol se rozdělují do čtyřech tříd podle toho, jaké vlastnosti UIProtocolu podporují. Klient nižší třídy může volitelně podporovat vlastnosti vyšší třídy. Třídy klientů viz. [38].

V současnosti existuje několik klientů implementovaných v různých programovacích jazycích a určených pro různé platformy - klient pro Apple iPhone [15], klient postavený na platformě PHP [37], klient postavený na technologii Flash [42] a referenční klient pro UIProtocol implementovaný v jazyce C# pro platformu .NET.

1.3.7 Komunikace mezi klientem a serverem

Server před připojením klienta naslouchá na TCP portu. Klient se připojuje k tomuto portu. Po navázání spojení klient posílá serveru Connect event (viz. kapitola 1.3.4). Pokud je Connect event v pořádku a uživatel je autentizován a autorizován serverem, odešle server standardní model public.application s popisem UIProtocol aplikace (viz. tabulka 1.2), pokud ne, je klient odpojen.

Poté, co je spojení úspěšně navázáno, může probíhat komunikace mezi klientem a serverem. Klient zasílá serveru požadavky, zobrazuje uživatelská rozhraní a server na požadavky odpovídá a zasílá data. TCP komunikační kanál je udržován otevřený po celou dobu spojení klienta se serverem.

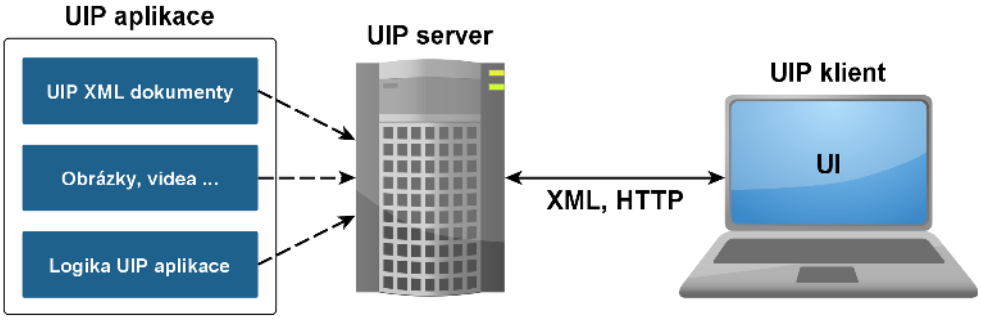
[12] Hypertext Transfer Protocol (HTTP) je internetový protokol určený pro výměnu hypertextových dokumentů ve formátu HTML [10]

Ukončit komunikaci může jak klient, tak server. Pokud chce spojení ukončit klient, zašle serveru Disconnect event (viz. kapitola 1.3.4) a následně uzavře spojení. Pokud chce spojení ukončit server, zašle klientovi aktualizaci standardního modelu public.connection (viz. tabulka 1.2), kde nastaví Property connected na hodnotu false a následně uzavře spojení.

1.4 UIPServer

Java verze serveru pro UIProtocol pojmenovaná UIPServer byla představena v [2]. Tato verze přinesla podporu souběžného provozu více UIProtocol aplikací. Aplikace mohou na serveru existovat bez toho, aby se navzájem ovlivňovaly.

Na UIPServeru může být v jednu chvíli spuštěno více UIProtocol aplikací. Aby mohla být aplikace využívána klientem, musí se z ní vytvořit instance. Toto je inspirováno aplikacemi a procesy v běžných operačních systémech. Aplikace může existovat na disku počítače, stejně jako UIProtocol aplikace na UIPServeru, neznamená to ale, že je aplikace spuštěna. Po spuštění je aplikace nahrána do operační paměti a je tak vytvořen proces. Takto se chová i UIPServer, kdy po spuštění UIProtocol aplikace je tato nahrána do paměti a vytvořena její instance. K jedné instanci může být připojen jeden a více klientů. Pokud je z jedné UIProtocol aplikace vytvořeno více instancí a ke každé instanci jsou připojeni klienti a v jedné instanci je upraven model pro všechny klienty připojené k instanci, projeví se tato změna pouze v této instanci.



Obrázek 1.5: UIProtocol klient komunikující s UIPServerem

Instance UIProtocol aplikací vytváří uživatel pomocí UIPPortalu, kde si také zvolí, k jaké instanci se chce připojit pomocí svého klienta pro UI-

Protocol. UIPPortal je komunitní portal a deployment managemet server pro UIPServer též představený v [2].

Aby mohl být uživatel připojen k instanci, kterou si vybral pomocí UIPPortalu, musí být při připojení k UIPServeru identifikován. Klient zasílá při svém připojování k serveru Connect event, která mimo jiné obsahuje Properties username a password. Tyto Properties jsou UIPServerem použity k identifikaci uživatelů a toho, k jaké instanci se chce uživatel připojit.

1.4.1 Zavádění UIPServeru

Po spuštění UIPServeru je vytvořena instance třídy `UipServer` starající se o načtení nastavení serveru (třída `Settings`), otevření spojení s databází přihlašovacími údaji uživatelů a seznamem UIProtocol aplikací a instancí (třída `DatabaAccess`), otevření spojení s poskytovatelem souborů UIProtocol aplikací z FTP serveru nebo ze souborového systému (třída `UipFilesAccess`), inicializaci logu (třída `Log`), spuštění XML a HTTP serveru naslouchajících požadavkům klientů, signálového serveru, který přijímá požadavky například na restart nebo ukončení UIPServeru a v neposlední řadě proběhne inicializace manažera instancí UIProtocol aplikací přestavovaného třídou `InstancesManager`.

1.4.2 Instance UIP aplikací

Instance třídy `XmlServerListener` otevírá TCP socket naslouchající nově se připojujícím klientům. Po připojení klienta je vytvořena instance třídy `XmlClient` spouštějící vlákno, které přijímá požadavky od klienta.

Po připojení musí klient zaslat jako první Connect Event. V této Event jsou uloženy uživatelské jméno a heslo, pomocí kterých jsou klienti serverem rozlišováni. Instance třídy `XmlClient` předá po připojení klienta přijatou Connect Event ke zpracování `InstancesManageru`, který ověří uživatelské jméno a heslo v databázi, a pokud je klient úspěšně autentizován, zjistí v databázi instanci zvolenou uživatelem, ke které bude klient přiřazen. Pokud instance na UIPServeru není spuštěna, je nejdříve instance vytvořena. Pokud není uživatel úspěšně autentizován nebo nezašle jako první connection event, je mu odeslána chybová zpráva a je od serveru odpojen.

Instanci UIProtocol aplikace představuje instance třídy `Instance`. Seznam všech spuštěných instancí si udržuje `InstanceManager`. Každá instance třídy `Instance` má svoje manažery UIProtocol dokumentů a Event handlers. Při vytváření nové instance UIProtocol aplikace se vytváří instance třídy `UipDocumentManager`, která se stará o načtení všech UIPro-

tocol XML souborů příslušejících UIProtocol aplikaci, ze které je instance vytvořena. V XML souborech jsou hledány jednotlivé Actions, Interfaces a Models.

- Models jsou předávány instanci třídy `ModelManager`, která modely parsuje, vytváří z nich vnitřní reprezentaci používanou UIPServerem a ukládá si jejich seznam.
- Actions jsou předávány instanci třídy `ActionManager`, která si ukládá jejich objektový popis a jejich seznam.
- Interfaces jsou předávány instanci třídy `InterfaceManager`, která si ukládá jejich objektový popis a jejich seznam.

Každá instance UIP aplikace má jednu svojí instanci každé z těchto tříd. Při vytváření instance UIP aplikace je také vytvořena instance třídy `UIEventManager`, která načte Event handlers z jejich souborů. Každá instance si také udržuje seznam `XmlClient` instancí, tj. k ní připojených klientů.

Jakmile je klient úspěšné přiřazen k běžící instanci, další příchozí požadavky od klienta již zpracovává instance třídy `Instance` představující danou instanci UIProtocol aplikace.

1.4.3 Zpracování požadavků klienta

Jakmile je přijata nová zpráva od klienta, je předána instance třídy `Instance` představující instanci UIProtocol aplikace, ke které je klient připojen, která zprávu zkontroluje, a pokud je v pořádku, vytvoří z ní vnitřní reprezentaci používanou dále UIPServerem. Podle obsahu zprávy je rozhodován další postup.

Pokud zprávou je žádost o Interface, Action nebo Model (Request event), je zpráva předána příslušnému manažeru, který hledá ve svém seznamu příslušnou Action, Interface nebo Model. Pokud je požadovaná položka nalezena, je odeslána zpět klientovi.

Pokud je zprávou něco jiného, je přijatá zpráva (Event) předána instanci třídy `UIEventManager`, která spustí podle identifikátoru přijaté Event Event handler se stejným identifikátorem, pokud takový existuje, a předá přijatou Event jemu.

1.4.4 Event handlers

Event handlers jsou programy, které vykonávají veškerou logiku UIProtocol aplikace na serveru. Event handlers jsou součástí UIProtocol aplikace. UIPServer podporuje Event handlers naprogramované v jazyce Java a JavaScript. API, které používají Event handlers, je popsáno v [3]. Zde je také popsáno, co musí splňovat Event Handlers zapsané v Javě a JavaScriptu.

Každému Event handler jsou při jeho spuštění předány instance objektů implementující rozhraní `IServer`, `IClient` a `IEvent`. Instance `IEvent` představuje Event přijatou od klienta a obsahuje seznam Properties přijatých v této Event. Pomocí instance `IServer` je možné manipulovat s modely pro všechny klienty připojené k instanci a získávat data z modelů společných všech klientům připojeným k instanci. Pomocí instance `IClient` je možné manipulovat s modely pro konkrétního klienta, který zaslal Event, a získávat data ze soukromých modelů tohoto klienta.

Jakmile je upraven model pro jednoho klienta, je tento upravený model uložen do seznamu soukromých modelů klienta. Tento seznam je udržován v instanci třídy `XmlClient` příslušející klientovi.

Všechny požadavky na modely vyřizuje a aktualizace modelů provádí instance třídy `ModelManager`. Pokud klient požádá o model, je nejdřív tento model hledán mezi jeho soukromými modely. Když není nalezen, tak je hledán mezi modely společnými pro všechny klienty.

1.4.5 Mediální soubory

Mediální soubory jsou klientům poskytovány pomocí HTTP serveru, který UIPServer obsahuje. Server naslouchající příchozím připojením představuje třída `HttpServerListener`. Po připojení nového klienta je vytvořena instance třídy `HttpClient`, která spouští vlákno přijímající požadavky od klienta.

1.4.6 Načítání souborů UIProtocol aplikací

Soubory UIProtocol aplikací jsou načítány pomocí statických metod třídy `UipFileAccess`. Tyto metody volají instanci třídy implementující rozhraní `IUipFileStorageAccess`. Toto rozhraní implementují ovladače pro konkrétní typ úložného prostoru, ze kterého jsou načítány soubory UIProtocol aplikací. Rozhraní obsahuje metodu `getFile`, která bude vracet instanci třídy implementující rozhraní `IUipFile`, které bude představovat jeden soubor UIProtocol aplikace uložený v daném úložišti. Parametrem této metody

je cesta k souboru v úložišti. Tato cesta je relativní a jednotná pro všechny typy úložišť.

Aplikace se v úložišti načítají ze složky nastavené v konfiguraci UIPServeru. Každé aplikace je ve vlastní složce pojmenované jednoznačných identifikátorem aplikace. Struktura složky jedné aplikace je stejná jako struktura deploy souboru UIProtocol aplikace popsaného v [3].

1.5 UiGE UI Generator

Generátor uživatelských rozhraní UiGE UI Generator [23] vytváří CUI z AUI. Generátor je součástí platformy UIProtocol. Generátor spolupracuje při generování CUI se serverem pro UIProtocol. Generátor představený v [22] a v [23] je součástí .NET verze UIProtocol serveru. Tento generátor generuje CUI pro platformu UIProtocol v syntaxi UIProtocolu popsané v kapitole 1.3.3.1.

Vstupem pro generování CUI je popis AUI v syntaxi UIProtocolu popsané v kapitole 1.3.3.2, model určující vlastnosti zařízení, na kterém bude vygenerované rozhraní zobrazeno, model popisující schopnosti a preference uživatele, pro kterého se rozhraní generuje, model popisující vlastnosti prostředí, ve kterém se nachází koncové zařízení, pro které se rozhraní generuje a model nastavení asistenčních technologií.

1.5.1 Vstupní modely

Čtyři výše zmíněné vstupní modely ovlivňují zásadním způsobem podobu výsledného CUI. Souhrnně se tyto čtyři modely nazývají model kontextu [20].

V modelu určujícím vlastnosti zařízení jsou uloženy absolutní hodnoty, např. velikost písma v bodech pro průměrného uživatele bez zrakové vady. Jsou zde také vyjmenovány třídy všech konkrétních elementů, ovládacích prvků, které zařízení podporuje. Kompletní popis modelu zařízení je možné nalézt v příloze k [23].

V modelu prostředí jsou uloženy relativní hodnoty udávající vliv prostředí na vlastnosti zařízení. Tj. potřeba zmenšit nebo zvětšit písmo a jas obrazovky v závislosti na okolním osvětlení aj.

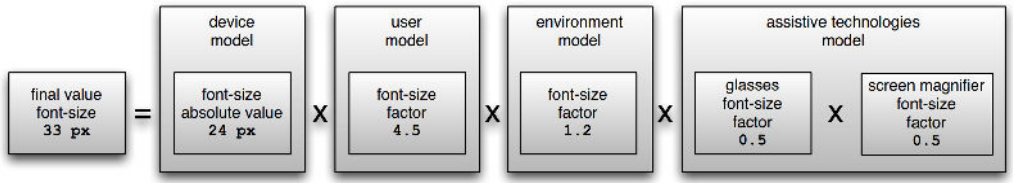
V modelu uživatele jsou podobně jako v modelu prostředí uloženy relativní hodnoty. Tyto hodnoty popisují schopnosti a preference uživatele relativně vzhledem k absolutním hodnotám uvedeným v modelu zařízení,

které představují potřeby průměrného uživatele používajícího toto zařízení. Kompletní popis modelu uživatele je možné nalézt v příloze k [23].

V modelu nastavení asistenčních technologií [20] se nacházejí relativní hodnoty udávající vylepšení schopností uživatele pomocí různých asistenčních technologií. Asistenční technologií mohou být například brýle, díky kterým bude uživatel schopný přečíst menší text.

Výsledná hodnota vlastnosti ovládacího prvku CUI, např. velikost textu na tlačítku, se potom vypočítá jako součin hodnot ze všech čtyř modelů. Pro tento případ v modelu zařízení se uvažuje velikost písma (např. 20 px). Uživatel má zhoršený zrak. V modelu uživatele je toto specifikováno příslušnou relativní hodnotou 1,5. Model prostředí udává, že obrazovka zařízení je vystavena prudkému protisvětlu. Je tedy třeba zvětšit velikost písma a zvýšit jas obrazovky. V modelu prostředí je toto specifikováno příslušnou relativní hodnotou 1,6. Model asistenčních technologií neudává žádnou hodnotu. Výsledná velikost textu tlačítka je: $20px \times 1,5 \times 1,6 = 48px$.

Na obrázku 1.6 je znázorněn obdobný výpočet včetně vlivu modelu asistenčních technologií.



Obrázek 1.6: Výpočet hodnoty vlastnosti elementu CUI ze vstupních modelů (převzato z [20])

1.5.2 Mapování

Pro vygenerování konkrétního ovládacího prvku z abstraktního je třeba určit, jaké prvky mohou být z abstraktního prvku vygenerovány. K tomu slouží tzv. mapování. Mapování určuje jeden nebo množinu konkrétních ovládacích prvků, které se z abstraktního elementu nebo kontejneru vygenerují. Každé mapování dokáže určit svoji cenu pomocí cenové funkce. Čím nižší je cena mapování, tím vhodnější je užití mapování v daném kontextu a tím snadnější by mělo být použití konkrétních ovládacích prvku reprezentovaných mapováním pro uživatele. Cenové funkce není prozatím blíže specifikována a je předmětem budoucího výzkumu [23]. Momentálně je cena každého mapování udána konstantní číselnou hodnotou.

Každé mapování o sobě může prohlásit, zda je legitimní v daném kontextu situace. Mapování má přístup ke vstupním modelům. Pokud je tak uživatel, pro kterého se rozhraní generuje, slepý, jistě bude v tomto případě neplatné mapování, které mapuje abstraktní tlačítko na konkrétní grafické tlačítko s textovým popisek a místo toho bude upřednostněno mapování na zvukovou volbu s hlasovým nebo jiným potvrzením. Mapování může být neplatné také v případě, když cílové zařízení, pro které se uživatelské rozhraní generuje, nepodporuje konkrétní ovládací prvek specifikovaný mapováním.

1.5.3 Proces generování CUI

Proces generování CUI je popsán z pohledu zdrojového kódu .NET verze UiGE naprogramovaného v jazyce C#. Analýza .NET verze UIProtocol serveru, jejíž je .NET verze UiGE součástí viz. [2].

Generátor, třída `UIGenerator`, se inicializuje v main metodě třídy `UIProtocolServer`. V tomto místě se načítá i seznam všech AUI v aplikaci. O načítání definic AUI, jejich parsování a vytvoření objektové reprezentace AUI se stará `AbstractInterfaceProvider`. Načítání souborů se spouští zavoláním metody `LoadAbstractInterfacesFromFileInfos`. `AbstractInterfaceProvider` slouží i jako manager AUI.

AUI představuje třída `AbstractInterface`. Tato třída slouží nejen k zachycení struktury AUI, ale uchovává i informace o podobě konkrétního uživatelského rozhraní, jako pozice a velikost elementů, zvolená mapování aj. Zavoláním metody `GetConcreteInterface` po ukončení procesu optimalizace dojde v vygenerování XML struktury výsledného CUI (5).

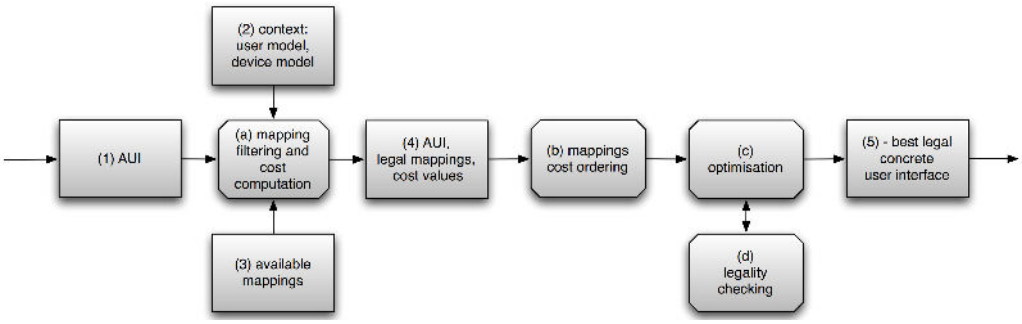
Proces generování CUI je spuštěn ve třídě `InterfaceManager` v metodě `GetInterface` v případě, že metoda nenajde požadovaný interface mezi existujícími CUI načtenými při startu serveru. Nejdříve se načte a aktualizuje model zařízení popsaný v kapitole 1.5.1. Model je pro lepší manipulaci obalen třídou `ClientProperties`. Následně je načten model uživatele a obalen třídou `UserProperties`. Model prostředí není v kódu načítán. Model zařízení a model uživatele spolu tvoří kontext generování představovaný třídou `Context`. Na obrázku 1.7 toto odpovídá (2).

Nyní je řízení předáno postupně třídě `UIGenerator`, metodě `GetInterace`, která provolá metodu `GetInterace` třídy `ConcreteInterfaceBuilder` zodpovědnou za vlastní generování CUI.

Vytvoření instance třídy `MappingsProvider` zajistí načtení všech dostupných mapování pomocí reflexe. Konkrétní implementace mapování musí implementovat rozhraní `IMapping`. Na obrázku 1.7 (3). Prostřednictvím

instance třídy `AbstractInterfaceProvider` je načtena objektová reprezentace AUI, reprezentovaná třídou `AbstractInterface`, ze které se bude CUI generovat (1). Zavoláním metody `ComputeAvailableMappings` třídy `AbstractInterface` se projdou rekurzivně všechny elementy a kontejnery abstraktního rozhraní a pro každý z nich se určí v daném kontextu legitimní mapování (a, 4). Legitimnost mapování udává metoda `CheckAndInit` z rozhraní `IMapping`.

Následně se spustí optimalizace rozmístění a výběr konkrétních elementů navržených v mapování, metoda `Optimize` třídy `Optimizer` (c).



Obrázek 1.7: Schematické znázornění procesu generování CUI pomocí UiGE (převzato z [23])

Všem elementům a kontejnerům se přiřadí lokálně optimální mapování. Pro každý element a kontejner je mezi jeho legálními mapováními vybráno to s nejnižší vahou, bez ohledu na výběr mapování u ostatních elementů a kontejnerů. Pokud element nemá žádné mapování, nebude umístěn ve výsledném CUI. Cena mapování se zjišťuje pomocí metody `EstimatedEffort` z rozhraní `IMapping`. Pro konkrétní ovládací prvky představované vybranými mapováními se pokusně vypočítá plocha, kterou by zabralo uživatelské rozhraní z nich sestavené. Pokud je tato plocha menší nebo rovna ploše pro vykreslování uživatelských rozhraní, kterou má klient k dispozici, bylo nalezeno globálně optimální rozložení uživatelského rozhraní (5) a není nutné spouštět kombinatorický optimalizační algoritmus, který jinak globálně optimální řešení hledá.

Velikost dostupné plochy pro vykreslování uživatelských rozhraní zasílá klient při připojení k UIProtocol serveru a při každé změně velikosti této plochy (změna velikosti okna, otočení obrazovky telefonu). Výpočet plochy zabraný konkrétními ovládacími prvky se spouští pomocí metody

`SimpleLayout` třídy `AbstractInterface`. Ta rekurzivně volá metodu `Layout` zvoleného mapování pro každý abstraktní kontejner a element AUI.

Pokud je vypočítaná plocha větší než plocha klienta pro vykreslování uživatelských rozhraní, je nutné spustit kombinatorickou optimalizaci představovanou metodou `OptimumSearch` třídy `Optimizer`.

1.5.4 Optimalizační algoritmus

Algoritmus hledá přiřazení mapování ke všem abstraktním elementům a kontejnerům takové, že součet ceny všech přiřazených mapování je nejmenší možný a řešení je legální. Legální řešení znamená, že plocha zabraná všemi konkrétními ovládacími prvky představovanými zvolenými mapováními je menší než plocha, kterou má k dispozici klient pro vykreslování CUI.

Algoritmus je úplný, tj. vždy najde řešení a pokud jej nenajde, řešení neexistuje, a optimální, tj. nalezené řešení je zároveň nejlepší možné řešení. Algoritmus prochází rekurzivně do šířky strom stavového prostoru problému.

Na začátku každého průchodu se zjišťuje legálnost aktuálně zkoumaného prvku stavového prostoru. Je-li nelegální, tj. jestli už byla přiřazena mapování všem abstraktním elementům a kontejnerům a spočtená plocha výsledného CUI je větší, než plocha dostupná klientovi, daný průchod končí.

Pokud je řešení legální, srovnává se cena aktuálního přiřazení mapování všech abstraktním elementům a kontejnerům (cena je součtem cen všech přiřazených mapování, je to cena výsledného CUI) se zatím nejnižší dosaženou cenou. Pokud je cena vyšší, průchod končí. Je zde použita metoda větví a hranic [5] pro ořezávání stavového prostoru problému. Počítá se i cena nekompletně namapovaného řešení. Pokud je tato cena vyšší než zatím nejnižší zjištěná, nemá cenu pokračovat v procházení této větve stavového prostoru. Cena by se již nezlepšila.

Pokud byly přiřazeny mapování všem abstraktním elementům a kontejnerům, aktualizuje se nejnižší dosažená cena.

Pokud není mapování kompletní, vybere se první nenamapovaný element nebo kontejner. Postupně se procházejí všechna mapování komponenty a přiřazují se jim jako aktuální. Každé přiřazení mapování znamená vygenerování nového nenavštíveného stavu stavového prostoru. Na tomto stavu se spustí další rekurzivní průchod.

Takto se pokračuje tak dlouho, než se projdou všechny stavy, které nebyly označeny jako nelegální nebo suboptimální metodou větví a hranic.

1.6 Vize řešení

Kapitola přibližuje vizi řešení diplomové práce podrobně rozvedenou v kapitole 2.

1.6.1 Úprava UIPServeru

Bude třeba upravit architekturu UIPServeru tak, aby šel snadno rozšiřovat pomocí nových modulů. Jednou oblastí kterou je nutné upravit, jsou součásti starající se o síťovou komunikaci a parsování příchozích zpráv. Tyto součásti momentálně existují jako jeden modul. To je ale neflexibilní přístup. Při požadavku přidat podporu pro novou variantu UIProtocolu, např. binární variantu, by bylo nutné znova programovat i část starající se o síťovou komunikaci. Tím by docházelo k velkým duplicitám kódu. Tyto části budou proto odděleny do samostatných modulů.

Další částí, kterou bude nutné vyřešit univerzálnějším způsobem, jsou document managery, které schraňují seznamy dokumentů UIProtocolu, jako interfaců, modelů a akcí. Spolu s Java verzí UiGE přibude nový document manager určený pro správu AUI. Bude nutné vytvořit jednotný interface, který budou implementovat všechny document managery. Tento interface zajistí u document managerů implementaci metod pro načtení příslušného UIProtocol dokumentu, nalezení dokumentu podle identifikátoru aj. Instance document managerů jsou součástí každé instance třídy `Instance`. Budou odstraněny gettry pro jednotlivé managery a místo toho bude jedna generická metoda vracející žádaný document manager podle jeho identifikátoru - nejspíše podle rozhraní, které identifikuje. Tato změna je nutná kvůli tomu, že UiGE včetně příslušného document manageru pro AUI nebude pevnou součástí UIPServeru. Je nutné, mimo jiné z licenčních důvodů, aby UIPServer dokázal fungovat bez UiGE jako předtím. Pro co nejjednodušší začlenění funkcionality UiGE do existujícího binárního sestavení UIPServeru bude možné přidávat do UIPServeru moduly s novou funkčností bez nutnosti jej znovu kompilovat.

V současnosti již existuje v UIProtocolu architektura umožňující spouštět pro obsluhu events tzv. event handlery. Event handlery jsou momentálně specifické pro každou UIProtocol aplikaci spuštěnou na UIPServeru. Tato architektura bude rozšířena tak, aby bylo možné přidávat do UIPServeru event handlery, které budou společné pro všechny aplikace běžící na UIPServeru. Tyto event handlery, nazvěme je **autonomní event handlery**, budou mít přístup k API celého UIProtocol serveru, nebudou tedy omezené jako event handlery specifické pro jednotlivé UIProtocol aplikace. Tyto

event handlery budou vždy považovány za důvěrné části programu. Tyto event handlery budou načítány vždy při spuštění celého UIPServeru v závislosti na konfiguraci UIPServeru. Díky tomu bude možné pomocí těchto event handlerů zařazovat do procesu zpracování požadavků klienta celé nové funkční bloky. Jedním z těchto funkčních bloků, který bude implementovat vlastní event handler bude i UiGE.

Event handler z UiGE bude reagovat na žádost o zaslání interface. Bude žádoucí tuto žádost do event handleru UiGE doručit pouze v případě, že nebyla předtím úspěšně zpracována event handlerem subsystému, který poskytuje klientovi CUI. Bude potřeba architekturu event handlerů upravit tak, aby mohlo na jednu event zaslanou od klienta reagovat více event handlerů v definovaném pořadí a aby tento řetěz zpracování mohl být kdykoliv jedním z event handlerů přerušen. Tato představa odpovídá objektovému návrhovému vzoru chain [11]. Pořadí spouštění event handlerů reagujících na stejnou event bude možné určit v konfiguraci UIPServeru.

Též bude nutné navrhnout subsystém, který by umožňoval těmto důvěryhodným event handlerům připojovat se na síťová rozhraní a posílat a přijímat zprávy ve formátu UIProtocolu. Při generování CUI totiž UiGE komunikuje s klientem po zvláštním kanálu pomocí event protocolu, což je podmnožina UIProtocolu. UiGE pomocí tohoto kanálu zjišťuje od klienta vzhled a velikost konkrétních ovládacích prvků, jak by je klient vygeneroval.

1.6.2 UiGE a jeho vylepšení

Architektura UiGE bude upravena tak, aby mohl fungovat jako modul napojený na event handler, jak bylo pospáno v předchozí kapitole. UiGE bude přepsán do jazyka Java.

Jednou z možností jak zrychlit generování uživatelského rozhraní je použít místo optimalizačního algoritmu popsaného v kapitole 1.5.4 heuristickou funkci, která by dokázala najít dostatečně kvalitní řešení blížící se optimálnímu řešení v co nejkratším čase. Jednou z možností je nehledat pro každý element mapování samostatně. Generovanými uživatelskými rozhraními jsou často formuláře, ve kterých se velká část ovládacích prvků, jako např. políček pro vkládání textu, opakuje. Všechny abstraktní elementy, které je možné namapovat na stejný konkrétní ovládací prvek pomocí stejného mapování by tak byly podle tohoto namapovány. Úkolem bude navrhnout strategii tohoto přidělování.

Tímto způsobem nebude možné mapovat kontejnery. Je nežádoucí, aby se kontejnery automaticky mapovaly na stejný konkrétní ovládací prvek.

Vnořené kontejnery by se tak například mohly všechny namapovat na kontejner s posuvníky podle prvního kontejneru, který je obaluje, i když by se jejich obsah do nich vešel bez posouvání. Toto chování pro kontejnery tak jistě není žádoucí.

Složitost algoritmu tak sice zůstane teoreticky exponenciální kvůli úplnému průchodu stromu stavového prostoru s kontejnery, ve skutečnosti ale bude mnohem lepší. V běžném rozhraní se vyskytuje více elementů než kontejnerů.

Maximální asymptotická složitost výpočtu bude: $O(n, m) = 2^n + m$, kde n je počet abstraktních kontejnerů a m počet abstraktních elementů v AUI.

1.6.3 Generátor AUI z JPA modelu

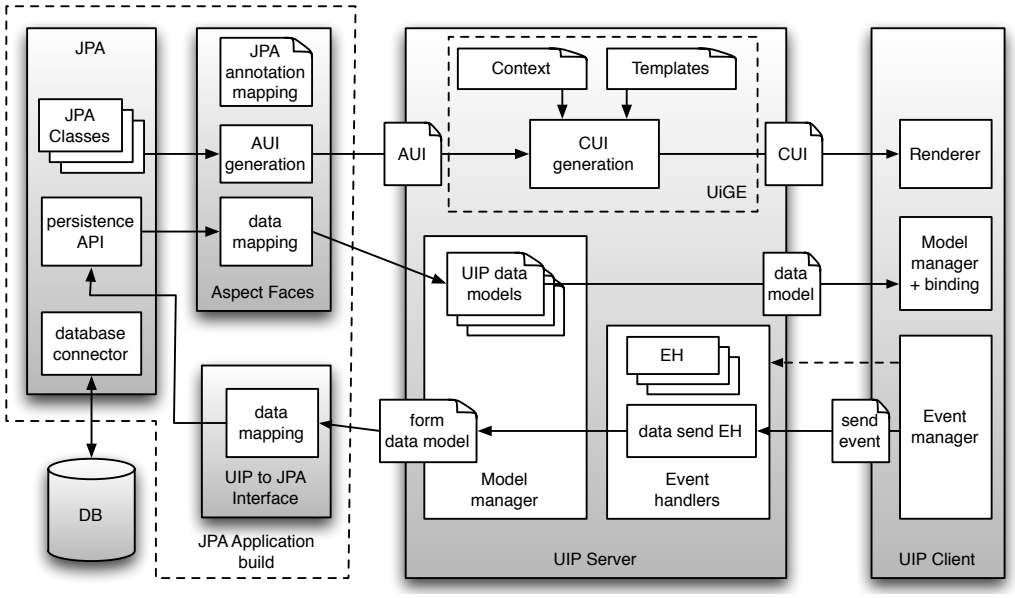
AUI ve formátu UIProtocol budou generována z datového modelu JPA pomocí nástroje AspectFaces. Vznikne mapování a příslušné tagy pro primitivní datové typy Javy, jejich objektové verze, Enum a obecnou třídu JPA modelu, která bude implementovat určité rozhraní. Generátor bude samostatnou knihovnou jazyka Java. Mapování a tagy budou její součástí. AspectFaces momentálně neumí mimo servlet načítat mapování, která jsou součástí knihovny jazyka Java, JAR souboru. Bude navržen způsob doplnění podpory pro toto do AspectFaces.

Generátor AUI bude uzpůsoben tak, aby podporoval statický i dynamický přístup ke generování AUI, jak bylo popsáno v kapitole 1.2.5.

1.6.4 Architektura systému a integrace

Bude se uvažovat JPA aplikace nasazená na aplikačním serveru pro platformu Java EE a to na aplikačním serveru, který splňuje specifikaci Java EE 6 Full profile [29]. Generátor AUI nastíněný v kapitole 1.6.3 bude součástí JPA Java EE aplikace. Generátor AUI bude mimo popisu AUI v syntaxi UIProtocolu vytvářet i pomocné modely UIProtocolu (viz. kapitola 1.3.2), na které budou přibindované některé prvky AUI. Také bude vytvářet modely, do kterých se budou ukládat data vyplněná uživatelem v CUI, které se vygeneruje z AUI. Tyto modely bude nutné jedinečným způsobem identifikovat, aby data v nich obsažená mohla být uložena do databáze v podobě správné entity pomocí modulu pro spojení JPA a UIProtocol aplikace.

Jak bylo zmíněno v kapitolách 1.6.1 a 1.6.2, UiGE bude součástí UIP-Serveru v podobě modulu napojeného na zpracování event handlerů. Bude poskytovat CUI vygenerovaná z AUI klientovi.



Obrázek 1.8: Architektura systému s JPA aplikací, AspectFaces, UIPServerem a UiGE (převzato z [21])

Součástí JPA Java EE aplikace bude modul, interface, spojující UIProtocol aplikaci a Java EE aplikaci, nazvěme jej **UIP to Java EE Connector (UTJEEC)**. Na tento modul připadá několik úkolů:

- Přijímání žádostí na vygenerování AUI z JPA modelu a spuštění procesu generování AUI.
- Přijetí dat v podobě UIProtocol modelu a převedení tohoto modelu na JPA entitu (instanci třídy jazyka Java), ze které model vznikl.
- Persistence přijatých dat do databáze.
- Přijetí žádosti na načtení dat z databáze od UIProtocol aplikace a odeslání načtených dat v podobě UIProtocol modelu.

Propojení JPA a UIProtocol aplikace nebude omezené jen na datovou úroveň, jak je zmíněno v zadání této práce. Pomocí modulu pro integraci JPA a UIP bude možné spouštět i metody EJB[13]. Modul pro integraci tak

[13]Enterprise Java Beans (EJB) jsou komponenty aplikace nasazené na serveru platformy Java EE umožňující tvorbu modulárních aplikací [34].

bude dovolovat integrovat UIP a JPA (tímto již Java EE aplikaci s JPA) aplikaci nejen na datové úrovni, ale i na úrovni business logiky aplikace. Není totiž vždy žádoucí, aby k databázi aplikace (JPA modelu) nasazené na Java EE aplikačním serveru bylo přistupováno jinak, než prostřednictvím business logiky aplikace implementované za využití EJB.

Není pochyb o tom, že jako komunikační protokol mezí Java EE JPA aplikací a UIProtocol aplikací nasazenou na UIPServeru by měl být využit UIProtocol. Event Protocol, podmnožina UIProtocolu, je napříč UIProtocol platformou používán jako standard pro komunikaci a zasílání zpráv týkajících se koordinace činnosti platformy. Jak název Event Protocol napovídá, jsou pomocí něj zasílány eventy jazyka UIProtocol (viz. kapitola 1.3.4). V tomto případě ale nebude samotný Event Protocol dostačující. Bude totiž nutné přenášet modely i interface jazyka UIProtocol. Pro komunikaci bude proto použit plnohodnotný UIProtocol, který toto dovoluje. Specifikem komunikace bude, že na rozdíl od komunikace mezi klientem a serverem UIProtocolu, kde se plnohodnotný UIProtocol také využívá, bude možné posílat eventy, modely i interface oběma směry.

UIProtocol aplikace bude k modulu pro integraci na straně Java EE JPA aplikace přistupovat prostřednictvím sady autonomních event handlerů (kapitola 1.6.1) nasazených na UIPServeru. Každý z těchto event handlerů bude obsluhovat jednu funkci modulu pro integraci jako spuštění generování AUI, persistenci dat, vyvolání EJB metody a další. Každá tato funkce tak bude identifikována jednoznačným id, aby mohla být vyvolána pomocí eventy s tímto id. Zprávu modulu pro integraci na straně Java EE JPA aplikace odešle autonomní event handler pomocí nově vytvořeného subsystému UIPServeru pro zasílání síťových zpráv zmiňovaného na konci kapitoly 1.6.1.

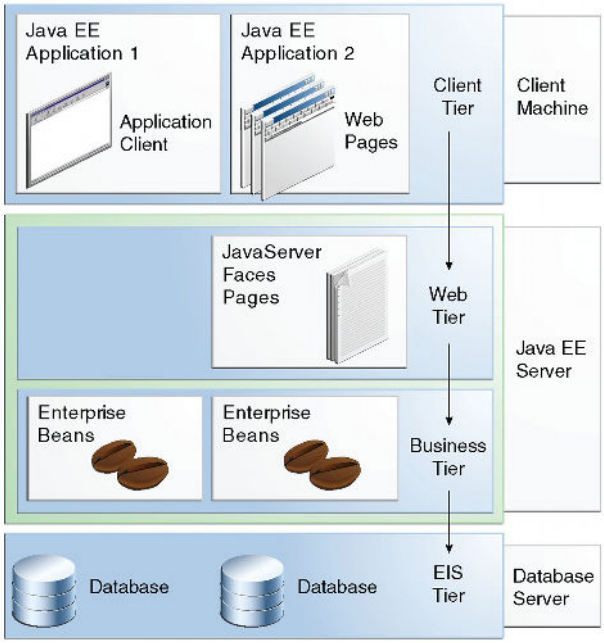
Pro implementaci modulu pro integraci na straně Java EE JPA aplikace bude nutné integrovat do Java EE aplikace potažmo Java EE serveru podporu pro UIProtocol. To znamená mimo jiné zajistit naslouchání příchozím síťovým spojením na zvláštním TCP portu. Standard EJB neumožňuje otevírání portů, které naslouchají příchozím spojením, ani vytváření vláken, která jsou pro obsluhu příchozích spojení nezbytná. Nabízí se však několik jiných možností, jak podporu pro UIProtocol do Java EE aplikace integrovat. Tyto možnosti, včetně výběru jedné preferované, budou nastíněny v kapitole 1.8.

Již v rámci [2] bylo uvažováno o integraci UIProtocol serveru do Java EE serveru, respektive zajištění nasaditelnosti UIPServeru jako Java EE aplikace na Java EE aplikační server. Modul pro integraci UIP a Java EE,

integrace UIProtocolu do Java EE a vylepšení samotného UIPServeru nastíněné v předchozích kapitolách bude navrženo tak, aby byla zajištěna snadná integrace UIPserveru do Java EE serveru jako Java EE aplikace.

1.7 Java a Java EE

Java je plně objektový vysokoúrovňový programovací jazyk vyvinutý firmou Sun Microsystems. Programy napsané v jazyce Java jsou překládány do tzv. byte kódu a spouštěny na virtuálním stroji Java Virtual Machine, který interpretuje byte kód a provádí program. Java Virtual Machine existuje pro většinu dnes široce používaných operačních systémů (Windows, Linux, Unix). Díky tomu jsou programy napsané v jazyce Java nezávislé na operačním systému a přenositelné mezi různými operačními systémy většinou bez potřeby úpravy kódu nebo rekompilace. V roce 2007 byla Java uvolněna a specifikace jazyka je nadále vyvíjena jako open source [34].



Obrázek 1.9: Architektura aplikace postavené na platformě Java Enterprise Edition (převzato z [34])

Java Enterprise Edition je platforma postavená nad programovacím jazykem Java a platformou Java Standard Edition, určená pro vývoj robustních podnikových systémů a aplikací a webových aplikací. Pro vývoj

webových aplikací jsou určeny technologie Java Servrlets - technologie pro zpracovávání požadavků klienta na zobrazení webové stránky, generování stránky, odesílání stránky klientovi, Java Server Pages - technologie rozšiřující Java Servrlets, kde Java kód je zapisován přímo do struktury HTML stránky podobně jako jazyk PHP, Java Server Faces a Facelets - framework pro vytváření webových aplikací nad technologií Java Enterprise Edition a značkovací jazyk určení pro tvorbu prezentační vrstvy (uživatelského rozhraní) webové aplikace. Pro tvorbu business logiky aplikace je určena technologie Enterprise Java Beans, dovolující snadné modelování procesů v aplikaci a business logiky, která implementuje robustní model transakčního a asynchronního zpracování požadavků a dat. Aplikace pro platformu Java Enterprise Edition jsou spouštěny na speciálních aplikačních serverech určených pro tuto platformu. Na těchto serverech jsou spuštěny jednotlivé části aplikace v tzv. kontejnerech. Existují kontejnery starající se o spouštění webové části aplikace, které vyřizují požadavky klientů a odpovídají na ně, a kontejnery pro zajištění běhu business logiky aplikace (viz. 1.9) [34].

1.8 Integrace UIProtocolu do Java EE

Podporu pro nový aplikační protokol je možné přidat do aplikace pro Java EE, aplikačního serveru pro Java EE, několika způsoby.

První možností je využít servlet[14] kontejneru a servletů, které jsou součástí každého aplikačního serveru pro Java EE. Pro přenos zpráv nového aplikačního protokolu by bylo využito HTTP[15] protokolu, který servlet kontejnery podporují již v základu. Tato možnost jistě nebude zvolena. Bylo by nutné měnit specifikaci UIProtocolu a upravit jej tak, aby bylo možné používat HTTP pro přenos zpráv v UIProtocolu. Tímto by se ztratila univerzalita využití UIProtocolu pro přenos zpráv v rámci UIProtocol platformy. Každý klient, který by chtěl komunikovat s UIPServerem nebo s UIP to Java EE Connector na platformě Java EE by tak musel implementovat nově variantu UIProtocolu přenášenou po HTTP. Kvůli množství již existujících implementovaných klientů pro UIProtocol toto není žádoucí.

Druhou možností je vytvoření Java SE klienta pro Java EE aplikační server, který by poskytoval podporu UIProtocolu. Tento klient by s Java EE

[14]Servlet je programová komponenta využívaná k rozšíření možností serveru. I když je serverem v tomto kontextu míněn server podporující libovolný aplikační protokol, většinou je pojem servlet spojen s implementací dovolující rozšiřování možností webového serveru s HTTP protokolem. Specifikace servlet je součástí platformy Java EE. [34].

[15]Hypertext Transfer Protocol (HTTP) je internetový protokol určený pro výměnu hypertextových dokumentů ve formátu HTML [10].

serverem komunikoval pomocí protokolu RMI[16] a přímo tak spouštěl EJB metody. Tato možnost také nebude zvolena. Separování podpory UIProtocolu do vlastní aplikace by neposkytovalo chtěné výhody, které nabízí Java EE server jako podpora škálování, centralizovaná správa vláken a jiné. Pokud by byla podpora UIProtocolu takto oddělena od Java EE aplikace, bylo by ji nutné nasazovat samostatně nestandardním způsobem. Pokud bude integrována přímo do Java EE aplikace, odpadne nutnost starat se o provoz dalších externích částí Java EE aplikace.

Třetí možností je využít technologie Java Connector Architecture, která je součástí Java EE Full Profile. Tato technologie slouží k integraci existujících enterprise aplikací a mimo jiné umožňuje standardizovaným způsobem vytvářet na Java EE serveru nová vlákna a otevírat nová síťová rozhraní naslouchající příchozím spojením.

Čtvrtou možností je rozšíření existujícího servlet kontejneru o podporu UIProtocolu. Servlet API bylo navrženo tak, aby bylo univerzální a aby bylo možné na jeho základě vyvinout servlet kontejner s podporou pro libovolný aplikační protokol. Jako vhodný existující servlet kontejner, který by bylo možné rozšířit o podporu nového aplikačního protokolu, se jeví v literatuře [7] dobře zdokumentovaný Apache Tomcat.

Poslední dvě možnosti budou podrobněji popsány v následující dvou kapitolách a v závěru bude jedna z nich zvolena. Na základě vybrané technologie poté bude navržena konkrétní podoba integrace UIProtocolu do Java EE serveru.

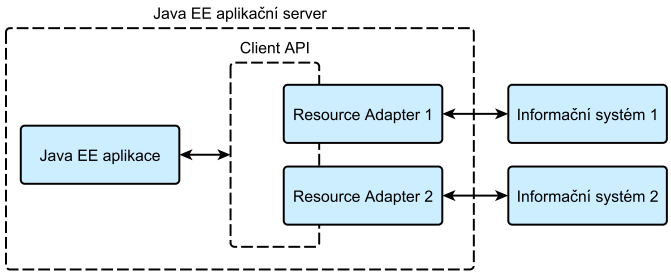
1.8.1 Java Connector Architecture

Java Connector Architecture (JCA) dovoluje standardizovaným způsobem propojit informační systémy postavené na platformě Java [13]. JCA se skládá ze dvou hlavních komponent, Client API a Resource adapter.

Resource adapter je komponenta, která definuje kontrakt, způsob propojení mezi Java EE aplikací a informačním systémem. V této komponentě je možné implementovat libovolný komunikační protokol použitý ke spojení s informačním systémem. Tato komponenta skrývá implementační specifika nutná pro vytvoření spojení a komunikaci s informačním systémem.

[16] Remote Method Invocation (RMI) je technologie meziprocesové komunikace pro platformu Java. Umožňuje vzdáleně volat metody tříd aplikace běžící na jednom virtuálním stroji jazyka Java z jiného virtuálního stroje. Takto je možné komunikovat mezi procesy i po síťovém spojení mezi různými fyzickými stroji [34].

Client API je API pro Java EE aplikaci, které definuje způsob, jakým bude aplikace interagovat s Resource adapterem a komunikovat tak s informačním systémem. JCA definuje standardizovanou podobu tohoto API, tzv. Common Client Interface. Je však možné vytvořit a používat i vlastní Client API.

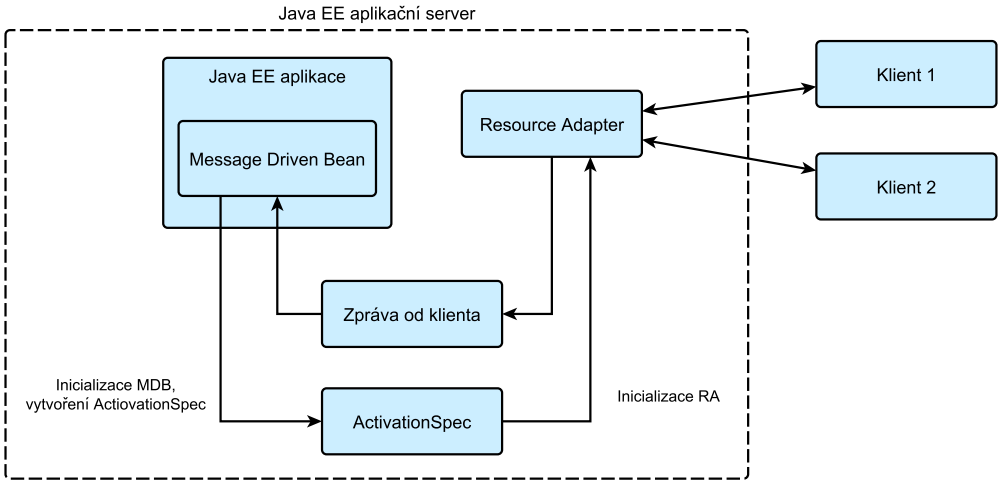


Obrázek 1.10: Resource Adapter připojující se k informačnímu systému

Ve standardním modelu použití JCA komponenta aplikace nasazené na Java EE serveru otevře spojení k informačnímu systému pomocí Client API a pomocí tohoto spojení s informačním systémem komunikuje, zasílá zprávy a přijímá odpovědi. Spojení v tomto smyslu není fyzickým spojením s informačním systémem, je to zástupný objekt skrývající specifika fyzického spojení. O fyzické připojení k informačnímu systému se stará Resource adapter.

JCA je také možné využít v druhém módu, kdy Resource adapter v sobě implementuje prostředky pro naslouchání příchozím síťovým spojením [35]. V tomto módu se nevyužívá Client API. Komunikaci nyní zahajuje informační systém, který se připojuje k síťovému portu, který otevřel Resource adapter. Resource adapter po připojení nového klienta, informačního systému předává práci vláknu, které se postará o zpracování požadavku klienta.

Resource adapter je jediná část Java EE aplikačního serveru, kde je umožněno vytvářet nová vlákna. Vytváření vláken neprobíhá standardním způsobem, tj. vytvořením instance třídy `Thread` a zavoláním její metody `start`. Ke spouštění práce v nových vláknech slouží API `WorkManager`. `WorkManager` umožňuje spouštět vykonávání kódu v nových vláknech. Kód, práce, která má být vykonána, je představována implementací rozhraní `Work`. Toto rozhraní dědí z rozhraní `Runnable`, které představuje práci, kterou má vykonat třída `Thread`. Díky tomu, že se vlákna nevytváří v Resource adapteru přímo, ale jejich vytváření zajišťuje `WorkManager`, může i zde Java EE aplikační server řídit běh a život vláken tak, jak to dělá v ostatních svých částech a kontejnerech.



Obrázek 1.11: Resource Adapter naslouchající příchozím spojením

V implementaci `Work` je o přijetí zprávy od klienta notifikovaná Message Driven Bean[17], která již obsahuje logiku pro zpracování zprávy případně zprávu deleguje do jiné části Java EE aplikace.

Součástí Resource adapteru je také implementace rozhraní `ActivationSpec`. Tato třída slouží ke konfiguraci Resource adapteru. Programátor zde může pomocí anotací `@ConfigProperty` uvést všechny konfigurovatelné vlastnosti Resource adapteru. Tyto vlastnosti jsou zde i ukládány. Instance této třídy je při inicializaci předána Resource adapteru.

Spuštění Resource adapteru je podmíněno vytvořením instance Message Driven Bean. V implementaci rozhraní `ActivationSpec` musí být uvedena anotace `@Activation` udávající rozhraní, které implementuje Message Driven Bean, jejíž inicializace zapříčiní i inicializaci a spuštění Resource adapteru. Pomocí tohoto rozhraní se tak párují Resource adapter a Message Driven Bean, která bude přijímat zprávy z Resource adapteru, které odeslal klient. V `ActivationSpec` mohou být uvedena, pokud to Resource adapter vyžaduje, pomocí anotace `@ActivationConfigProperty` všechna nastavení Resource adapteru, která se uloží do implementace rozhraní `ActivationSpec`.

Je přirozené implementovat naslouchání novým spojením pomocí neblokujícího síťového API - Non blocking I/O [16]. V běžnějším modelu realizace

[17]Message Driven Bean je komponenta Java EE aplikace dovolující asynchronní zpracování zpráv [34].

síťové komunikace za pomocí blokujícího API čeká hlavní vlákno na připojení klienta. Po připojení klienta je vytvořeno nové vlákno, které čeká na zaslání zprávy pro klienta. Pro každého připojeného klienta tak musí být vytvořeno minimálně jedno vlákno.

V neblokujícím API se nečeká na přijetí nové zprávy od klienta nebo připojení klienta blokujícím způsobem. Přesněji existuje jedno hlavní vlákno, které čeká na jakoukoliv událost, která nastane na síťovém rozhraní. V jednu chvíli může nastat více událostí, vlákno je odblokováno a o všech těchto událostech se dozví. Nemusí tak existovat jedno vlákno pro každého klienta. V základu stačí pro obsluhu síťového rozhraní jedno vlákno. Přijaté zprávy jsou často zpracovány pomocí jiného vlákna než pomocí toho, které zprávu přijalo. Nemusí se ale vytvářet vlákna nová. Nejčastější postup je takový, že se udržuje pool několika vláken, která zpracovávají příchozí zprávy a podle potřeby se příchozím zprávám přidělují. Ušetří se tak nezanedbatelné zdroje na vytváření nových vláken a jejich udržování.

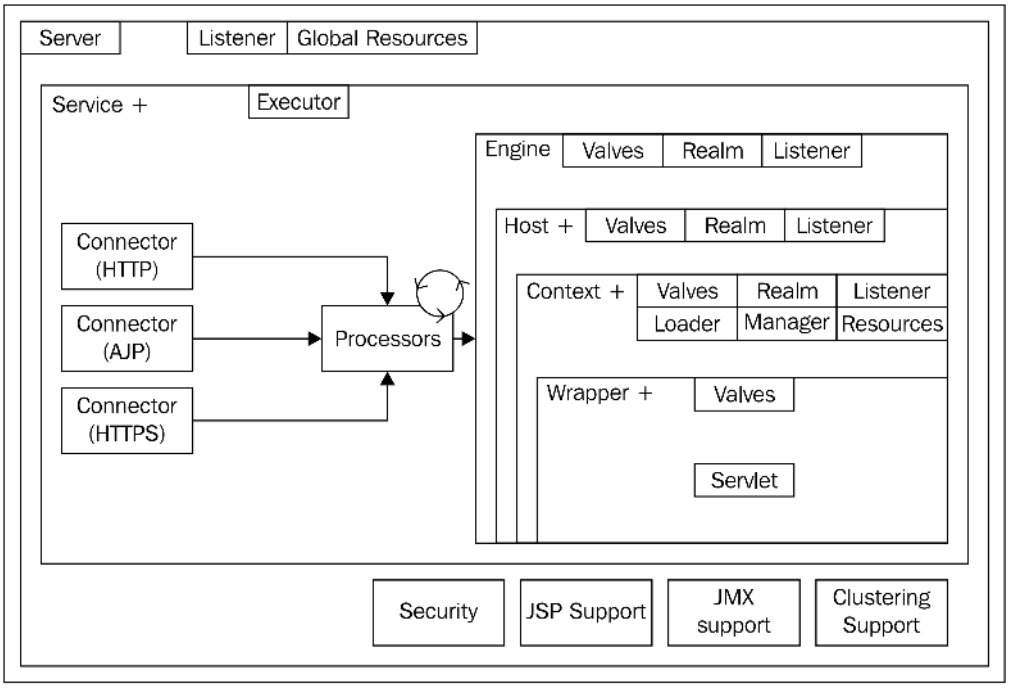
Proto je neblokující I/O vhodné pro použití v Resource adapteru. Příchozí zprávy jsou předávány ke zpracování `WorkManager` v podobě implementace rozhraní `Work` a ten si již sám řídí vytváření potřebných vláken.

1.8.2 Apache Tomcat

Apache Tomcat je servlet kontejner implementující servlet API, který poskytuje běhové prostředí pro servlety [7]. Apache Tomcat implementuje podporu pro HTTP servlety. Apache Tomcat tak funguje jako webový server rozšiřitelný pomocí servletů vývojářem webové aplikace. Díky dobře zdokumentované modulární architektuře a otevřeným zdrojovým kódům je teoreticky možné rozšířit Apache Tomcat o podporu nového aplikačního protokolu a nový typ servletu.

1.8.2.1 Architektura Apache Tomcat

Na obrázku 1.12 je schématicky znázorněna architektura Apache Tomcat. Názvy objektů na obrázku představují názvy rozhraní nebo abstraktních tříd. Toto nebude v textu dále zdůrazňováno. Proto když se v textu objeví např. zmínka o vytvoření instance třídy `Server` nebo jiné, myslí se tím vytvoření instance konkrétní implementace takové třídy nebo rozhraní. Tam, kde je ve schématu u názvu komponenty plus, může existovat více instancí takové komponenty uvnitř ve schématu nadřazené komponenty. Ve stručnosti budou popsány jen ty nejdůležitější komponenty.



Obrázek 1.12: Architektura servlet kontejneru Apache Tomcat (převzato z [7])

Po spuštění Tomcatu je vytvořena instance třídy `Server`, singleton [11], která se stará o načtení konfigurace Tomcatu, otevření lokálního síťového rozhraní, na které je možné zaslat příkaz k ukončení Tomcatu a inicializaci a spuštění ostatních částí serveru.

`Server` vytvoří podle konfigurace jednu nebo více instancí `Service`. `Service` je kontejner, který sdružuje jeden nebo více `Connector`ů a `Processor`ů spolu s jednou instancí `Engine`.

`Connector` je subsystém, který implementuje logiku pro zpracování specifik aplikačního protokolu, např. HTTP a otevírá síťové rozhraní, na kterém naslouchá příchozím spojením klientů aplikačního protokolu. Příchozí zprávy jsou předány ke zpracováním jednomu `Processor`u z poolu, který zprávy dále deleguje komponentě `Engine`.

`Engine` je zodpovědný za routování přijaté zprávy do příslušné instance `Host`. `Engine` tak umožňuje virtuální hosting, tj. provozovat na jednom serveru více domén. `Host` představuje jednu samostatnou doménu.

Na jedné doméně může být provozováno více webových aplikací. Webová

aplikace je reprezentována třídou `Context`. `Context` sdružuje všechny statické a dynamické zdroje jako webové stránky, obrázky a servlety načtené z archivu webové aplikace Web Application Archive (WAR) nebo ze složky s webovou aplikací. Každé instanci `Context` tak odpovídá jeden WAR archive. Z každého servletu se vždy vytváří jen jedna instance, která je obalená třídou `Wrapper`. `Context`, webová aplikace se dle specifikace HTTP servletu konfiguruje pomocí XML souboru `web.xml`. Zde se určuje, při jakém požadavku klienta na zdroj webové aplikace se spustí logika naprogramovaná v konkrétním servletu.

Důležitou komponentou `Context`u je `Manager`. Je to session manager, který se stará o udržování stavu komunikace mezi klientem a serverem a stavu webové aplikace. Umožňuje tak vytvářet stavové aplikace nad bezstavovým HTTP protokolem.

1.8.2.2 Integrace podpory UIProtocolu

Při integraci UIProtocolu do Apache Tomcat by byla vytvořena nová implementace `Engine`, která by přímo obsahovala seznam instancí `Context`. UIProtocol nepodporuje technologii viruálního hostingu ani jí podobnou, proto by byla vynechána vrstva v podobě instancí tříd `Host`. Bylo by možné použít existující implementaci `Service`.

Byl by vytvořen nový `Connector`, který by implementoval specifika komunikace prostřednictvím UIProtocolu. Nová implementace `Processor` by parsovala příchozí zprávy a vytvářela z nich objektovou reprezentaci. Tato by byla předána enginu spolu s údaji o klientovi, který ji zaslal a ten by podobně jako `InstanceManager` (viz. kapitola 1.4.2) v UIPServeru rozhodl, ke které instanci UIProtocol zpráva patří a která jí zpracuje.

`Context` by tak představoval jednu instanci UIProtocol aplikace reprezentované archivem UIProtocol aplikace [3]. Vlastní implementace `Manager`u by se starala o udržování stavu instance UIProtocol aplikace.

Byla by vytvořena nová specifikace servletu, podobně jako existující specifikace HTTP servletu, která by byla rozšířením generické specifikace servletu. Nazvěme ji UIP servlet. UIP servlety by tak odpovídaly nynějším event handlerům UIProtocol aplikace (viz. kapitola 1.4.4).

1.8.3 Zvolený způsob integrace UIProtocolu

Pro integraci UIProtocolu do Java EE serveru byla zvolena technologie JCA. Ta představuje standardizovaný způsob, jak integrovat do Java EE serveru podporu pro nový aplikační protokol. UIP to Java EE Connector

nebo UIPServer tak bude možné nasadit spolu s Java EE aplikací v jednom Enterprise ARchive (EAR) archivu, který bude teoreticky nasaditelný na všechny Java EE aplikační servery splňující specifikaci Java EE 6 Full Profile.

Přidání podpory pro UIProtocol do Apache Tomcat a vytvoření UIProtocol servletu by sice přineslo takovému řešení větší robustnost a dobře zdokumentovanou architekturu. Na druhou stranu, takové řešení by nebylo univerzální a kompatibilní s platformou Java EE. Podpora pro UIProtocol by musela být včleněna do Apache Tomcat manuálně. Nebylo by možné nasadit UIP to Java EE Connector nebo UIPServer společně s Java EE aplikací, případně by taková aplikace fungovala vždy jen na upraveném Apache Tomcat.

Kapitola 2

Návrh

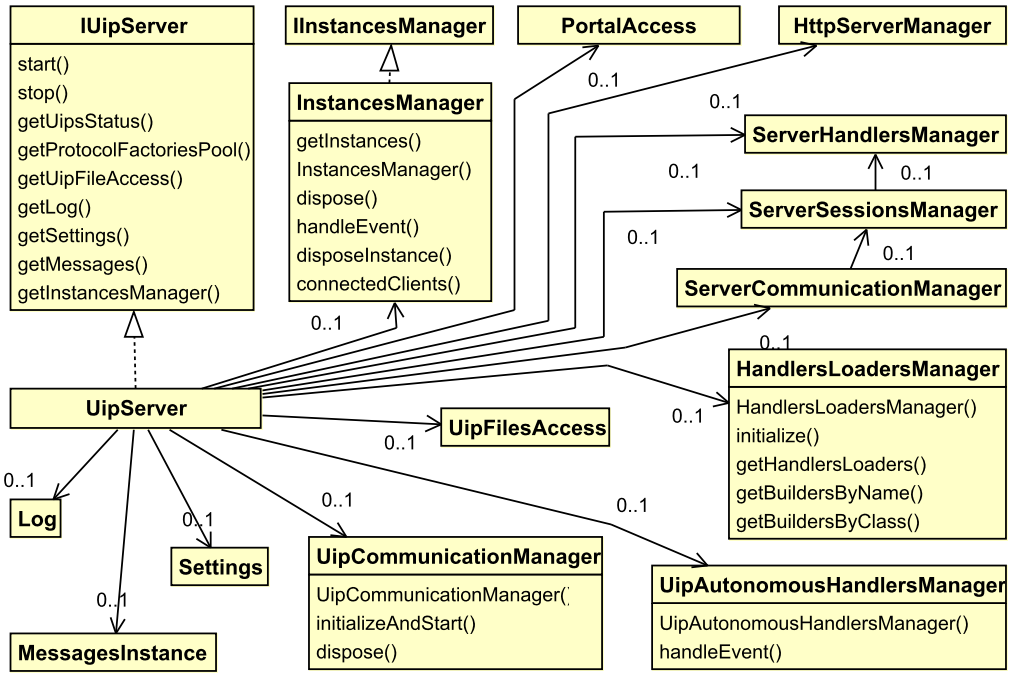
Kapitola popisuje návrh všech částí systému představených v kapitole 1.6. UML diagramy v kapitole Návrh, ani popis jednotlivých částí systému si nekladou za cíl být kompletní. Kompletní popis jednotlivých částí je nad rámec rozsahu této diplomové práce. V diagramech jsou vyznačeny jen důležité vztahy nezbytné pro pochopení návrhu jednotlivých částí systému.

2.1 UIPServer

Hlavní součásti, subsystémy UIPServeru, jsou zobrazeny na obrázku 2.1. O inicializaci a spuštění všech subsystémů se stará třída `UipServer`, která implementuje rozhraní `IUipServer`. Instance této třídy je vytvořena jako první při spouštění UIPServeru. Referenci `IUipServer` mají k dispozici všechny subsystémy UIPServeru. Tak jsou jednotlivé subsystémy propojeny. Díky rozhraní bude možné vytvořit různé implementace `IUipServer`. Implementace `UipServer` je určena pro zavádění UIPServeru v prostředí Java SE. Jiná implementace bude poskytnuta pro zavádění UIPServeru z prostředí Java EE aplikačního serveru.

Mezi subsystémy UIPServeru patří:

- `MessagesInstance` - podpora pro lokalizaci
- `Log` - logovací subsystém umožňující logování do souboru, na konzoli a do připojené databáze
- `Settings` - načítání a správa nastavení UIPServer z XML souborů



Obrázek 2.1: UIPServer a jeho hlavní součásti

- `UipFileAccess` - načítání dat UIProtocol aplikací, generický subsystém nezávislý na konkrétním úložišti, kde jsou data uložena
- `PortalAccess` - přístup k databázi s informacemi o UIProtocol aplikacích, instancích a přihlašovacími údaji uživatelů
- `HttpServerManager` - poskytování mediálních souborů UIProtocol aplikací připojeným klientům prostřednictvím HTTP protokolu
- `ServerCommunicationManager`, `ServerSessionsManager`, `ServerHandlersManager` - řízení činnosti UIPserveru externími nástroji
- `UipCommunicationManager` - řízení komunikace s UIProtocol klienty, přijímání a odesílání zpráv po síťovém rozhraní, odbavování připojujících se klientů na naslouchající síťové rozhraní
- `InstanceManager` - správa spuštěných instancí UIProtocol aplikací, udržování seznamů instancí a připojených klientů

- `UipAutonomousHandlersManager` - načítání, inicializace a spouštění autonomních event handlerů

- `HandlersLoadersManager` - inicializace běhových prostředí pro event handlery specifické pro UIProtocol aplikace psané v různých programovacích jazycích.

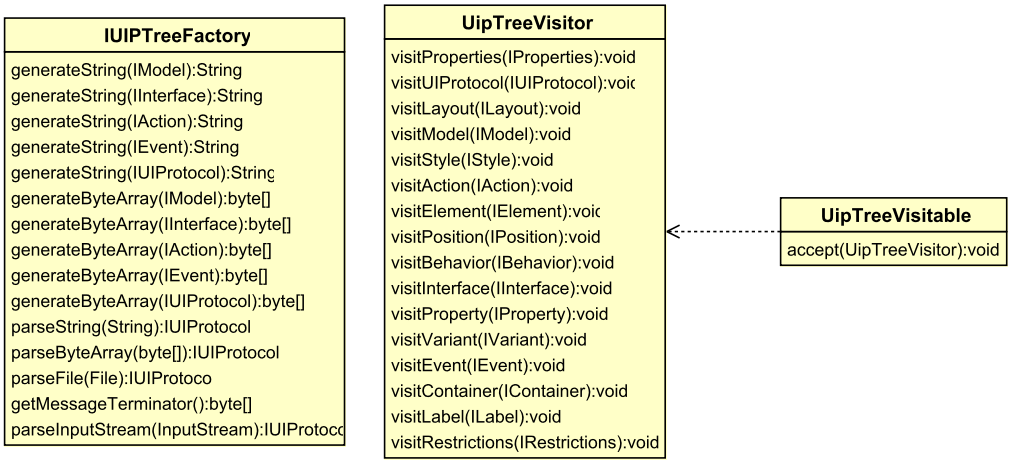
V kapitole budou diskutovány jen změněné a nové části UIPServeru důležité pro tuto práci - komunikační subsystém, event handlery a document managery. Ostatní části nebudou podrobněji zmiňovány.

Jako nedostatečný se ukázal při návrhu nynější systém pro načítání nastavení UIPServeru. Konfigurační subsystém a konfigurační XML byly změněny tak, aby vystihovaly jednotlivé konfigurovatelné subsystémy UIPServeru. Část XML souboru tak bude reprezentována v objektové podobě, která bude dostupná jednotlivým subsystémům. Každý subsystém tak bude mít v konfiguračním XML pevně daný element, který bude představovat jeho nastavení. Aby byla XML konfigurace dostatečně flexibilní, bude pro jednotlivé komponenty, které mohou být do UIPServeru začleněny bez rekompilace, jako nové autonomní event handlery, parsery UIProtocolu, protokoly pro přenos dat po síti a síťová rozhraní a další, možné definovat další nastavení nemající svůj obraz v objektové reprezentaci nastavení. Tato nastavení budou reprezentována dvojicemi klíč - hodnota. Fragment konfiguračního souboru je k nahlédnutí v příloze D.

2.1.1 Generování a parsování dokumentů UIProtocolu

Pro interní zpracování v rámci celého UIPServeru se UIProtocol dokumenty překládají do objektové podoby. UIProtocol nemusí být jen v podobě XML dokumentů, jsou navrženy i binární nebo komprimované textové formy UIProtocolu. V systému tak může existovat více generátorů a parserů různých forem UIProtocolu. Jako fasáda (návrhový vzor facade [11]) pro vygenerování dokumentu UIProtocolu z objektové formy a parsování dokumentu do objektové formy slouží implementace rozhraní `IUIPTreeFactory`. Pro podporu snadnějšího generování dokumentů UIProtocolu ze stromové objektové reprezentace implementují objekty představující části UIProtocolu návrhový vzor visitor [11], rozhraní `UipTreeVisitable`. Vlastní generátor, implementace visitoru, musí implementovat rozhraní `UipTreeVisitor`. Využití visitoru implementací `IUIPTreeFactory` je volitelné. Implementace `IUIPTreeFactory` určená v konfiguraci UIPServeru je poté využívána pro

parsování příchozí a generování odchozí komunikace mezi klientem a UIPServerem, mezi UiGE a klientem, při načítání dokumentů UIProtocolu z úložiště UIProtocol aplikací a v dalších částech systému.



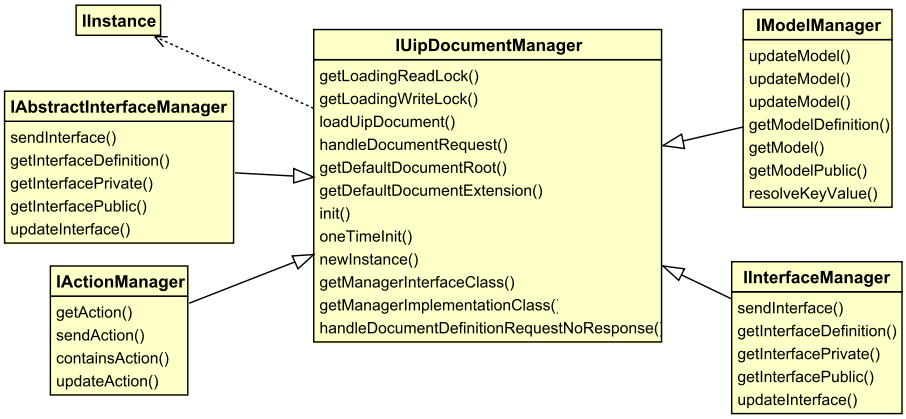
Obrázek 2.2: Generování zpráv a parsování zpráv do objektové reprezentace

2.1.2 Managery dokumentů UIProtocolu

Managery dokumentů slouží ke správě dokumentů UIProtocolu (modelů, akcí, interface, abstraktních interface). Každá instance UIProtocol aplikace představovaná třídou `Instance` implementující rozhraní `IInstance` má svoji sadu managerů dokumentů pro každý typ UIProtocol dokumentu. Jelikož UiGE, jehož součástí bude i manager abstraktních interface, nebude pevnou součástí UIPServeru, ale modulem, bez kterého je schopný UIPServer plnohodnotně fungovat, není možné prostě vložit do `Instance` referenci na dokument manager pro AUI, tak jak je to u ostatních.

Každý z managerů dokumentů tak bude implementovat rozhraní `IUipDocumentMamager` sdružující základní služby, které musí každý document manager poskytovat. Patří mezi ně zejména načtení spravovaného typu UIProtocol dokumentu z objektové reprezentace nebo inicializace document manageru při spouštění nové instance UIProtocol aplikace.

`Instance` bude udržovat místo jednotlivých referencí na konkrétní implementace jednotlivých document managerů seznam document managerů, tříd implementujících rozhraní `IUipDocumentMamager`. `Instance` bude poskytovat přístup k jednotlivým document managerům prostřednictvím generické metody s generickým parametrem. Metodě se bude předávat jako



Obrázek 2.3: Architektura managerů pro dokumenty UIProtocolu

parametr definice třídy (instance třídy `Class` v jazyce Java) rozhraní, které implementuje chtěný document manager. Každý document manager tak musí mimo rozhraní `IUipDocumentMamager` implementovat ještě jedno, jemu specifické rozhraní. Každý z managerů musí prostřednictvím metod `getManagerInterfaceClass` a `getManagerImplementationClass` rozhraní `IUipDocumentMamager` vracet definice třídy rozhraní, které manager implementuje a definice samotné konkrétní třídy manageru. Metoda `getDocumentManager` třídy `Instance` bude díky tomu schopná nalézt a vrátit požadovaný document manager. Díky tomu, že je metoda generická, bude návratový typ metody odpovídat rozhraní document manageru předanému metodě jako parametr.

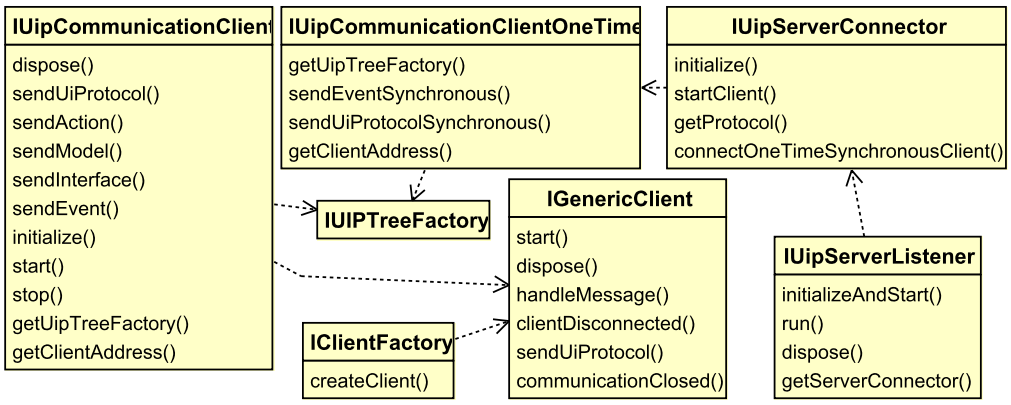
V nastavení UIPServeru bude možné specifikovat jaké document managery bude mít `Instance` k dispozici.

2.1.3 Komunikační subsystém a reprezentace klientů

Klienta, který se připojuje k serveru, komunikujícímu pomocí UIProtocolu, reprezentují dvě hlavní třídy. Jedna implementující rozhraní `IUipCommunicationClient` a druhá `IGenericClient`.

Instance implementující rozhraní `IUipCommunicationClient` má na starosti komunikaci po síťovém komunikačním kanále. Prostřednictvím instance `IUipTreeFactory` parsuje přijaté zprávy do objektové podoby a generuje data pro odeslání po síťovém spojení z objektové podoby UIProtocolu. Implementace `IUipCommunicationClient` tak může realizovat komunikaci prostřednictvím Transmission Control Protocol (TCP), User Da-

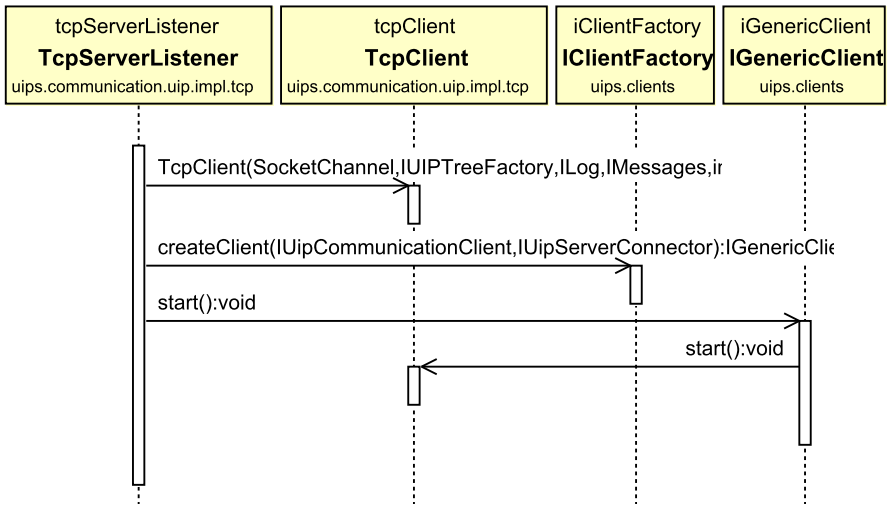
tagram Protocol (UDP) nebo šifrovaného Transport Layer Security (TLS) spojení za využití různých technologií. Hlavní rozhraní komunikačního subsystému jsou znázorněna na obrázku 2.4.



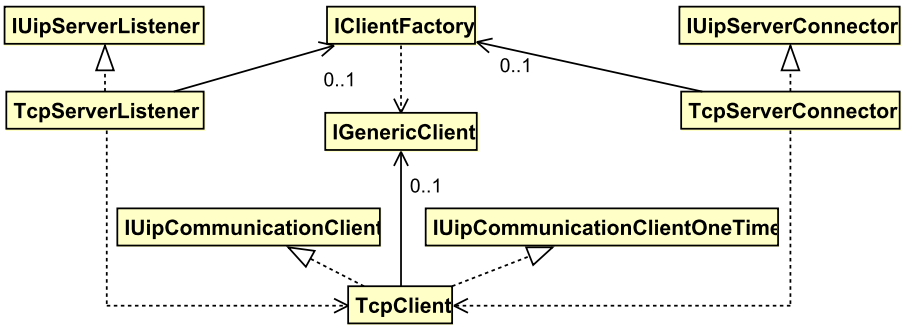
Obrázek 2.4: Rozhraní komunikačního subsystému UIPServeru

Odbavování nově se připojujících klientů a často i implementaci otevření portu pro naslouchání příchozím spojením má na starosti implementace třídy `IUipServerListener`. Při inicializaci instance implementace třídy `IUipServerListener` je instanci předána instance třídy implementující rozhraní `IClientFactory`. Tato třída vytváří objekty implementující rozhraní `IGenericClient`, která má na starosti vlastní zpracování přijaté UIProtocol zprávy, její objektové reprezentace. `IGenericClient` tak představuje implementaci vlastní aplikační logiky serveru, který komunikuje prostřednictvím UIProtocolu. Průběh připojení klienta je znázorněn na obrázku 2.5. Pro předpokládanou funkčnost musí `IGenericClient` a `IUipCommunicationClient` udržovat vzájemně referenci jeden na druhého.

Třída implementující rozhraní `IUipServerConnector` dovoluje systémům UIPServeru připojovat se k síťovému rozhraní jiné aplikace, která využívá ke komunikaci UIProtocol. Tak je možné s touto aplikací komunikovat. Jako `IUipServerListener` vytváří obdobným způsobem při připojení k síťovému rozhraní instance tříd `IUipCommunicationClientOneTime` a `IGenericClient`. Jak je vidět na obrázku 2.6, kde je vyobrazena implementace komunikačního subsystému pro TCP, rozhraní `IUipCommunicationClientOneTime` a `IUipCommunicationClient` bude často implementovat jedna třída. Nemusí to však být pravidlem.



Obrázek 2.5: Přijetí příchozího spojení a vytvoření nového klienta pomocí komunikačního subsystému

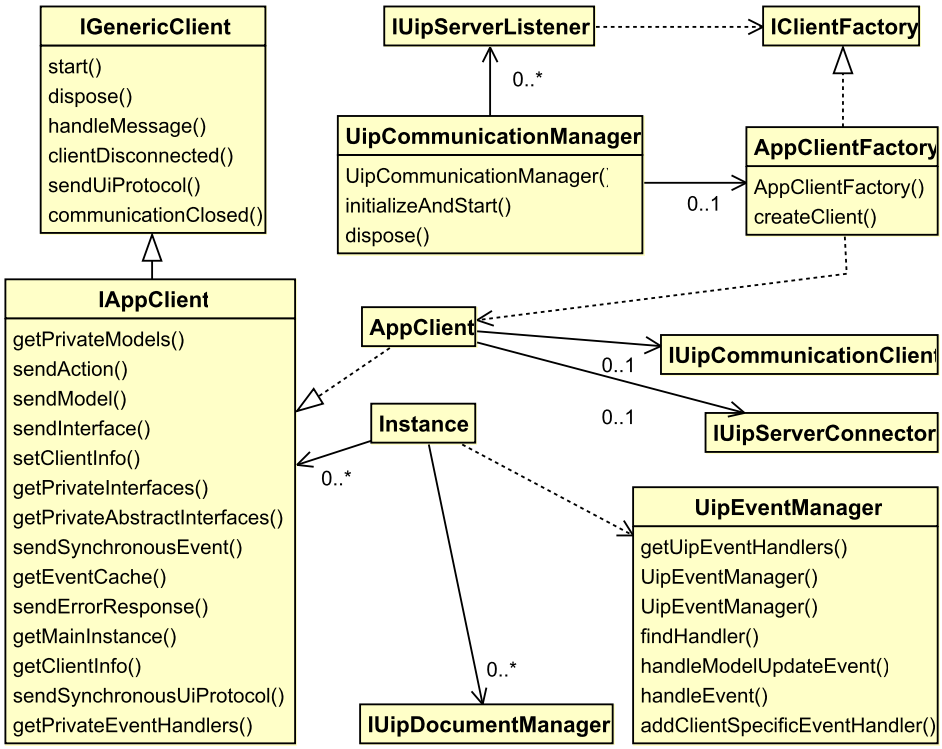


Obrázek 2.6: Komunikační subsystém pro TCP spojení

2.1.4 Instance aplikace a klienti

Obrázek 2.7 zobrazuje vlastní jádro UIPServeru - instance UIProtocol aplikací a správu klientů UIProtocolu k nim připojeným. Jádro využívá komunikačního subsystému s podporou generických klientů popsaného v předchozí kapitole.

V konfiguraci UIPServeru se uvede název třídy implementace `IUipServerConnector`, která bude naslouchat žádostem klientů o navázání spojení s UIPServerem. Té se při inicializaci předá implementace rozhraní `IClientFactory`, třída `AppClientFactory`. Instance této třídy je zodpo-



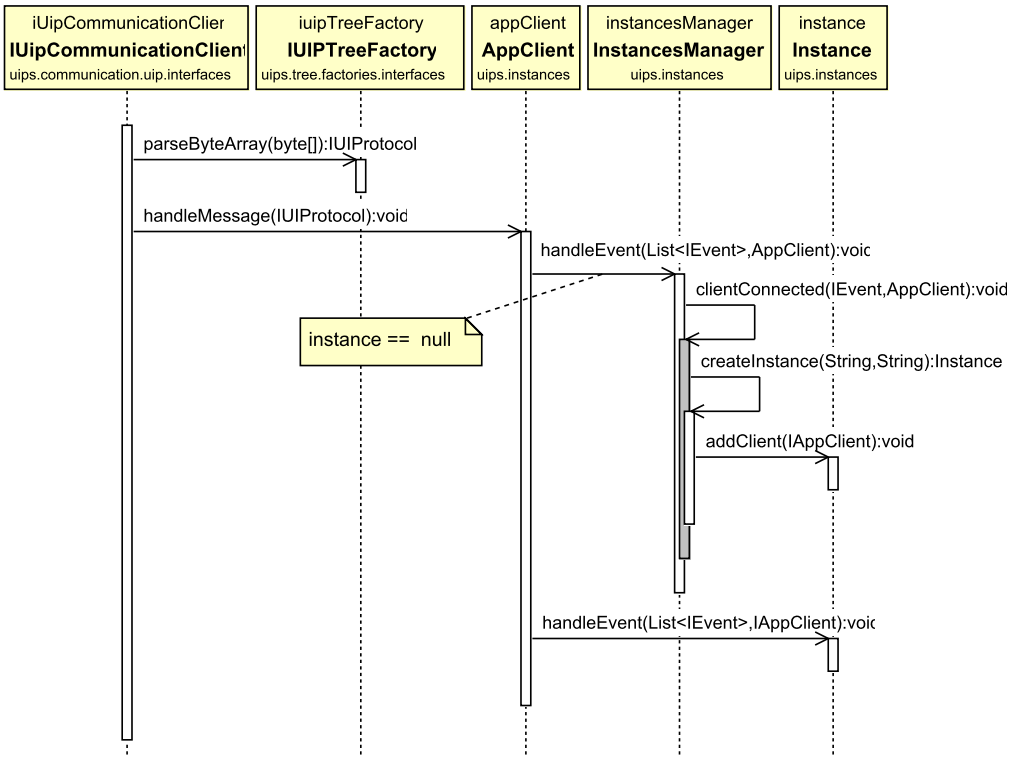
Obrázek 2.7: Klient využívající pro posílání UIProtocol zpráv komunikační subsystém

vědná za vytváření instancí třídy `AppClient`, která implementuje rozhraní `IGenericClient`. Připojení klienta a vytvoření instance `IGenericClient` je popsáno v kapitole 2.1.3.

Všechny instance `IUipServerConnector`, které naslouchají klientům na různých síťových rozhraních sdružuje třída `UipCommunicationManager`. Ta se také stará o inicializaci instancí `IUipServerConnector` podle konfigurace UIPServeru.

Třída `AppClient` představuje klienta UIProtocolu připojeného k UIPServeru. Sdružuje veškeré informace o klientovi a uchovává dokumenty UIProtocolu specifické pro klienta. `AppClient` udržuje referenci na instanci UIProtocol aplikace, ke které je klient připojen - instanci třídy `Instance` a referenci na managera běžících instancí UIProtocol aplikací `InstanceManager`. Třída `AppClient` v sobě také udržuje informace o fyzickém síťovém spojení s klientem UIProtocolu představované instancí `IUipCommunicationClient`. Pomocí ní je možné zasílat klientovi zprávy. Díky referenci na

`IUipServerConnector` může `AppClient` komunikovat pomocí síťového spojení s jinými aplikacemi, což je nezbytné pro UiGE (viz. kapitola 1.6.1).



Obrázek 2.8: Přijetí zprávy od klienta pro UIProtocol a vytvoření instance UIProtocol aplikace

Na obrázku 2.8 je znázorněn postup přijetí zprávy od klienta. `IUipCommunicationClient` klient přijme zprávu v podobě pole bajtů. Jelikož nemusí najednou přijmout celou zprávu nebo jich může naráz přijmout více, musí si udržovat buffer přijatých dat. Když je v přijatých datech identifikována kompletní zpráva, dokument UIProtocolu, je zpráva rozparsována a je z ní vytvořena objektová reprezentace UIProtocolu. O identifikaci zprávy a vytvoření objektové reprezentace se postará `IUIPTreeFactory`.

Objektová reprezentace zprávy je předána instanci `AppClient`, která z ní extrahuje event. Jak je popsáno v kapitole 1.3, klient může posílat jen eventy.

Pokud se klient zrovna připojil k serveru a zaslal svou první zprávu, není ještě spárován s žádnou instancí UIProtocol aplikace. Dokud není připojen k instanci, zpracovává všechny příchozí eventy klienta `InstancesManager`.

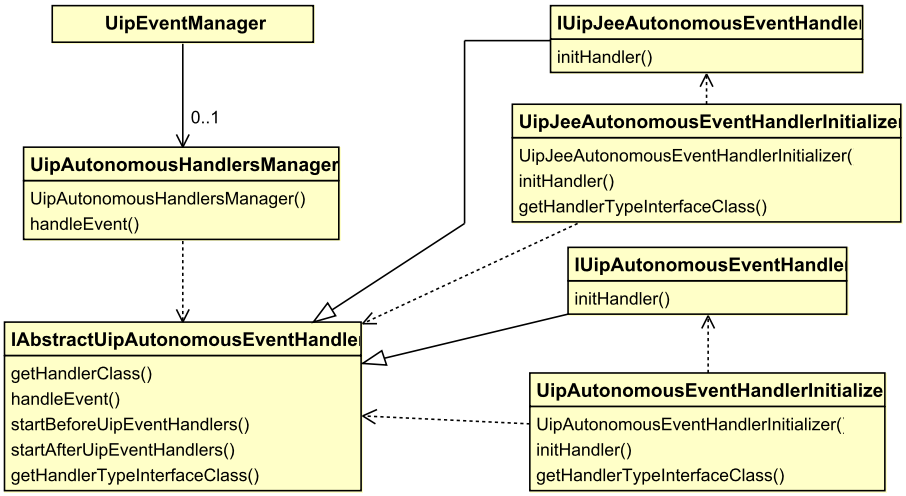
Ten páruje klienta s jím požadovanou instancí, případně instanci vytvoří, pokud tato ještě není spuštěna, viz. kapitola 1.4.2. Pokud již je klient připojen k instanci, zpracuje přijatou event tato instance.

2.1.5 Moduly UIPServeru a event handlery

Zprávy, eventy, přijaté od klienta jsou zpracovávány pomocí event handlerů. Jak je řečeno v kapitole 1.6.1, mimo již existujícího systému pro spouštění event handlerů specifických pro UIProtocol aplikaci, bude vytvořen nový systém pro spouštění autonomních event handlerů. Tyto event handlery budou společné všem aplikacím běžícím na UIPServeru. Autonomní handlery budou načítány při spouštění UIPServeru v závislosti na konfiguraci UIPServeru. Díky nim bude možné zařazovat do procesu zpracování požadavků klienta celé nové funkční bloky a rozšiřovat tak UIPServer o nové moduly.

Každá instance UIProtocol aplikace reprezentovaná třídou `Instance` obsahuje svoji instanci `UipEventManager`, která se stará o výběr odpovídajícího event handleru k event zaslané klientem a o její spuštění. Nový systém event handlerů umožňuje pro jednu event spustit více event handlerů.

Nový systém pro obsluhu autonomních event handlerů je vyobrazen na obrázku 2.9.



Obrázek 2.9: Architektura subsystému pro spouštění autonomních event handlerů

Při spuštění UIPServeru je v `UIPServer` vytvořena instance třídy

`UipAutonomousHandlersManager`, která načítá a inicializuje autonomní event handlery. V systému se budou zjišťovat event handlery za pomoci reflexe. Naleznou se všechny třídy, které implementují rozhraní `IAbstractUipAutonomousEventHandler`, které představují autonomní event handlery.

Může existovat více druhů autonomních event handlerů. Předpokládá se, že každý druh bude mít specifické potřeby inicializace - bude potřebovat inicializovat jiné vnitřní proměnné. Tak bude vytvořen jeden druh autonomních event handlerů spustitelný v prostředí Java SE a další druh, spustitelný pouze v prostředí Java EE. Ten bude mít přístup ke kontextu Java EE serveru a k `EntityManager`u JPA. Autonomní event handlery pro prostředí Java EE budou implementovat rozhraní `IUipJeeAutonomousEventHandler`, pro prostředí Java SE `IUipAutonomousEventHandler`.

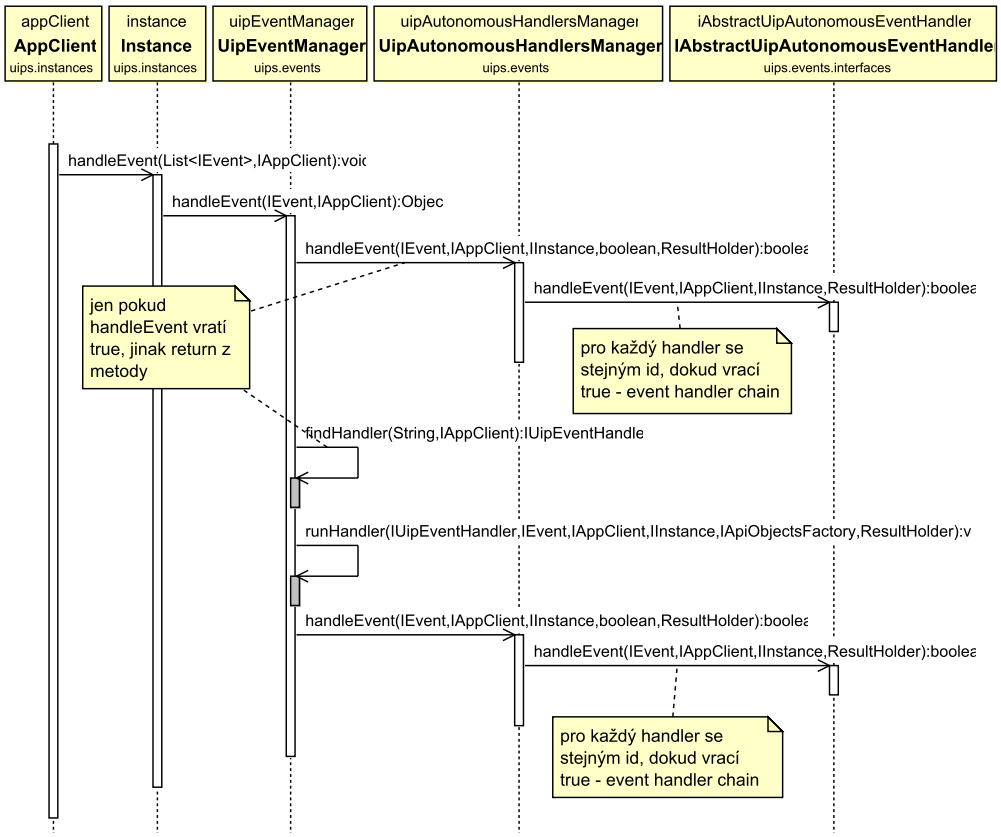
Při vytváření instance `UipAutonomousHandlersManager` se konstruktoru předá seznam instancí tříd implementujících rozhraní `IUipAutonomousEventHandlerInitializer`. Ty slouží k inicializaci event handlerů. Každý druh autonomních event handlerů má svůj inicializátor. Systém inicializuje a zpřístupní jen autonomní event handlery, pro které byl `UipAutonomousHandlersManager`u poskytnut patřičný inicializátor.

V konfiguraci UIPServeru je možné potlačit načtení některého autonomního event handleru a nastavit, v jakém pořadí se budou spouštět event handlery, které odpovídají stejné event. Event handlery se přiřazují k event podle id eventy.

Na obrázku 2.10 je znázorněn průběh spouštění event handlerů při přijetí event od klienta. Proces spouštění event handlerů vychází z principů návrhového vzoru chain [11]. Řetěz event handlerů se uplatní v případě, že pro event existuje více event handlerů, které ji mohou zpracovat. Event handlery se spouštějí až ve třech fázích. Nejdříve se spouští autonomní event handler, poté event handler z UIProtocol aplikace a nakonec opět autonomní event handlery.

Autonomní event handlery mohou specifikovat pomocí metod v rozhraní `IAbstractUipAutonomousEventHandler`, zda mají být spouštěni před spuštěním handleru z UIProtocol aplikace, po spuštění nebo v obou případech. Autonomní event handlery jsou spouštěny v pořadí, jaké je nastaveno v konfiguraci UIPServeru. Pokud pořadí nastaveno není, spouštějí se v nespecifikovaném pořadí.

První fáze spouštění event handlerů se provede pouze v případě, že existuje nějaký autonomní event handler, který se má spustit před spuštěním



Obrázek 2.10: Postup zpracování event přijaté od klienta

event handleru z UIProtocol aplikace. Jakýkoliv event handler může v první fázi přerušit řetěz event handlerů. Žádné další event handlery se nespustí a zpracování přijaté event končí.

Pokud nebyly nalezeny žádné autonomní event handlery nebo nebyl přerušen řetěz zpracování, spustí se v druhé fázi event handler z UIProtocol aplikace. Ten nemůže přerušit řetěz zpracování eventy.

Bezprostředně poté se spouští třetí fáze s autonomními event handlery, které mají být spuštěny po, obdobně jako první fáze.

Každý event handler zapojený v řetězu má k dispozici výsledky spuštění předchozího event handleru. Ten může tento výsledek změnit, vymazat nebo nahradit jiným. Celý řetěz handlerů vrací na konci pouze jeden výsledek.

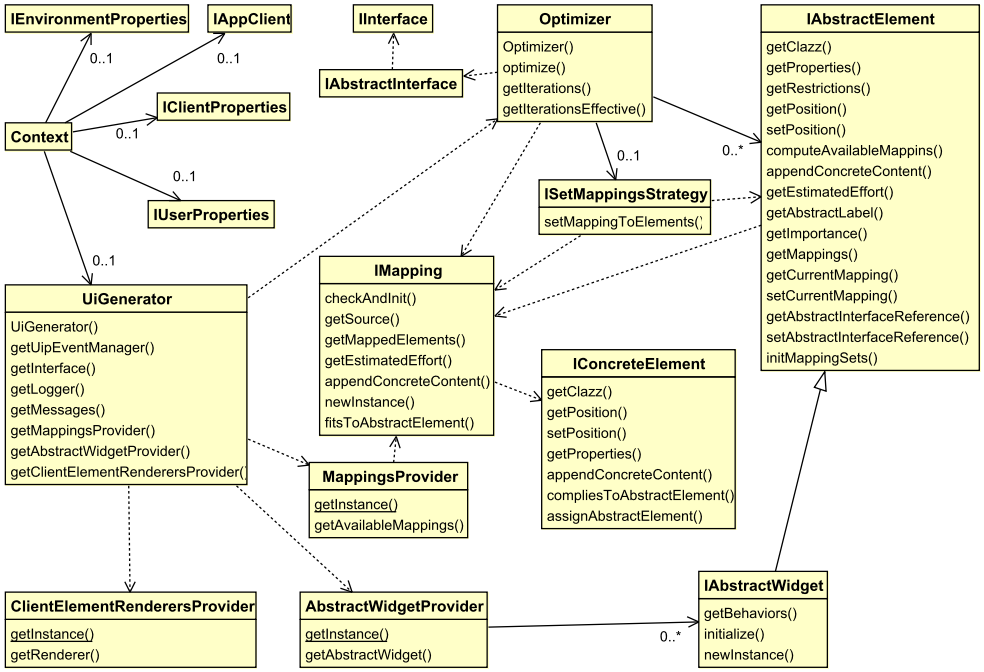
Do API pro programování event handlerů specifických pro UIProtocol aplikaci [3] přibude nová metoda, které umožní spouštění určeného event

handleru. Metodě se předá id event a seznam parametrů. Systém se chová, jako kdyby přišla nová event od klienta a spustí řetěz zpracování eventy popsaný v předchozích odstavcích. Metoda vrátí výsledek, který vrátil řetěz zpracování event handlerů. Díky tomuto bude možné spouštět jeden event handler z druhého podobně, jako jedna metoda může spustit jinou.

2.2 Java UiGE

Architektura Java verze UiGE zobrazená na obrázku 2.11 vychází z architektury .NET verze popsané v kapitole 1.5. Každá instance UIProtocol aplikace představovaná instancí třídy `Instance` bude mít k dipozici document manager `AbstractInterfaceManager` pro správu AUI.

Každá instance má svoji instanci třídy `UiGenerator`, která představuje hlavní rozhraní celého UiGE, kde se spouští generování CUI z AUI.



Obrázek 2.11: Architektura Java verze UiGE

Generátor má k dispozici kontext představovaný třídou `Context`, který sdružuje model prostředí, model zařízení a model uživatele. Součástí kontextu je také reference na instanci třídy `AppClient`, která představuje klienta žádajícího o vygenerování uživatelského rozhraní. Díky této referenci a

novému subsystému sloužícímu autonomním event handlerům k připojování k síťovým rozhraním jiných aplikací může UiGE komunikovat s klientem UIProtocolu. Při procesu generování CUI se UiGE dotazuje klienta, jak by vypadal konkrétní ovládací prvek, který se UiGE v procesu optimalizace snaží umístit na CUI. Komunikace probíhá pomocí event protokolu. Klient odesílá mimo jiné informace o velikosti ovládacího prvku, kdyby byl zobrazený na obrazovce. Pomocí této informace poté UiGE rozhoduje, zda bude ovládací prvek v této podobě umístěn na CUI.

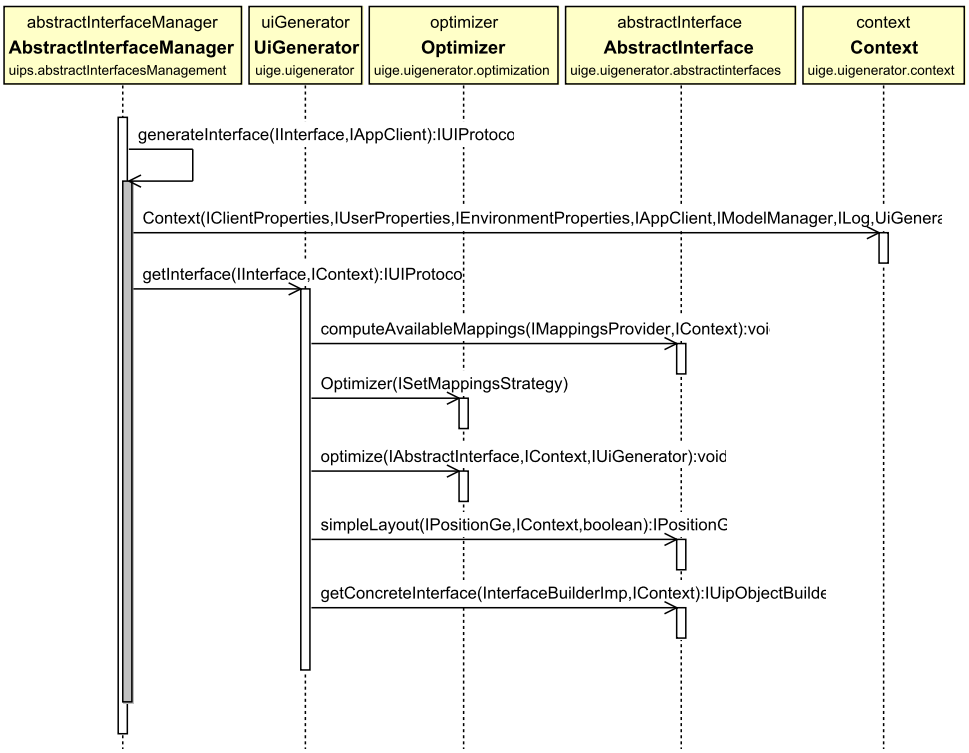
Jak je popsáno v kapitole 4.1, dotazování klienta na podobu ovládacích prvků přes síťové rozhraní je úzkým hrdlem UiGE výrazně prodlužující dobu generování CUI. Proto je součástí generátoru nová třída `ClientElementRenderersProvider`. Ta podle identifikátoru, třídy klienta, vrací instanci třídy `IClientElementRenderer`. Instance této třídy bude představovat jádro klienta zodpovědné za renderování ovládacích prvků na obrazovce. Díky tomu bude možné renderovací jádro klienta integrovat přímo do UiGE a při dotazu UiGE na vzhled konkrétního ovládacího prvku tak odpadne zdlouhavé dotazování po síti.

`IClientElementRenderer` bude při dotazu na renderování ovládacího prvku přijímat stejnou event, jaká by se posílala klientovi přes síťové rozhraní. Ve třídě `Context` bude nová metoda, pomocí které se bude žádat o renderování ovládacího prvku. Pokud je UiGE k dispozici renderovací jádro klienta, který žádá o vygenerování CUI, je použito k vyrenderování, jinak je o vyrenderování požádán klient přes síťové rozhraní.

2.2.1 Spuštění generování

Proces generování CUI zobrazený na obrázku 2.12 se spustí při přijetí požadavku klienta na odeslání uživatelského rozhraní. Součástí UiGE tak bude autonomní event handler, reagující na tuto event. Na event se stejným id bude reagovat i autonomní event handler, který hledá požadované rozhraní v manageru CUI. V nastavení UIPServer bude možné specifikovat pomocí určení pořadí event handlerů v řetězu autonomních event handlerů, zda se požadovaný interface nejdříve hledá mezi existujícími CUI nebo zda se vždy nejdřív generuje z AUI. Výchozím chováním bude, že se nejdříve hledá mezi existujícími CUI.

Pro spuštění generování slouží metoda `getInterface` třídy `UiGenerator`. Tato metoda očekává jako parametr instanci třídy `Context` a objektovou reprezentaci AUI. Kontext se sestavuje těsně před zavoláním `getInterface` třídy `UiGenerator`.



Obrázek 2.12: Spuštění procesu generování AUI vyvolané autonomním event handlerem reagujícím na event zaslanou klientem

Po zavolání `getInterface` generátor vytvoří z objektové reprezentace AUI vnitřní formu reprezentace AUI představovanou třídou `AbstractInterface`. Do té se ukládají při procesu optimalizace zvolená mapování a vlastnosti vygenerovaných konkrétních ovládacích prvků.

Před vlastním spuštěním optimalizace metodou `optimize` třídy `Optimizer` se pro každý abstraktní kontejner a element, který je součástí AUI, vyberou na základě kontextu všechna možná mapování. Tato mapování budou uvažována v procesu optimalizace. Pro každý element a kontejner se zvolí maximálně jedno mapování. Proces optimalizace a optimalizační algoritmus je popsán v následující kapitole.

Po skončení procesu optimalizace jsou pomocí mapování přiřazených jednotlivým abstraktním elementům a kontejnerům vygenerovány vlastnosti konkrétních ovládacích prvků, na které mapují, jako například velikost, aj.

Nakonec je ze všech informací sestaveno výsledné CUI pomocí třídy

`InterfaceBuilder`, která je součástí API event handlerů [3]. Třída je implementací návrhového vzoru builder [11]. Slouží k vytvoření objektové struktury CUI. Třída udržuje informace o aktuálně vytvářeném elementu CUI. Tuto informaci tak není nutné propagovat mezi jednotlivými elementy AUI, které pověřují přidáváním konkrétních ovládacích prvků do CUI mapování, která jim byla přiřazena v procesu optimalizace.

2.2.2 Optimalizační algoritmus

Struktura optimalizačního algoritmu je k nahlédnutí ve výpisu kódu 2.1. Algoritmus popsaný v kapitole 1.5.4 je upraven v části, kde se přiřazují mapování zatím nenamapovaným elementům a kontejnerům AUI (řádek 20 a 21 ve výpisu kódu 2.1). Struktura AUI je zobrazena na obrázku 1.4 v kapitole 1.3.3.2. Strategii (návrhový vzor strategy [11]) přiřazení mapování udává implementace rozhraní `ISetMappingStrategy`. Strategie přiřazení může být implementována tak, že optimalizační algoritmus nebude exaktním algoritmem, který najde optimální řešení, ale heuristikou, která nalezne řešení suboptimální (viz. kapitola 1.6.2).

```
1  optimalizace (řešení) {
2      if (nelegální(řešení)) {
3          konec
4      }
5
6      var odhadnutá_cena = cena(řešení)
7      if (odhadnutá_cena > cena(nejlepší_řešení)) {
8          konec
9      }
10
11     if (validní(řešení)) {
12         nejlepší_řešení = řešení
13         konec
14     }
15
16     var element_bez_mapování = vyber(řešení)
17     for (každé mapování z~možných mapování
18             elementu_bez_mapování) {
19         řešení = kopie řešení
20         přiřaď mapování pomocí strategie přiřazení (
21             element_bez_mapování, mapování, řešení)
```

```
22          optimalizace(řešení)
23      }
24  }
```

Výpis kódu 2.1: Optimalizační algoritmus UiGE

Nabízí se několik možných implementací strategií přiřazení mapování. Na čísla strategií bude odkazováno dále v textu:

1. Jako doposud se přiřadí mapování pouze jedné nenamapované komponentě. Algoritmus s touto strategií je úplný a nalezne optimální řešení.

2. Vybere se jedno možné mapování nenamapovaného elementu a stejně se namapují všechny elementy, kterým je možné toto mapování přiřadit a ještě nebyly namapovány. Každý element AUI si udržuje seznam mapování, pomocí kterých může být namapován. Možná mapování se vybírají postupně (řádek 17 a 18 ve výpisu kódu 2.1). Srovnává se vybrané mapování se seznamem mapování elementu. V případě, že mapování mapují na stejný konkrétní ovládací prvek se stejnými vlastnostmi, znamená to, že je možné elementu vybrané mapování přiřadit. Kontejnery a vnořené AUI se mapují jako v prvním případě.

3. Vybere se jedno možné mapování nenamapovaného elementu a stejně se namapují všechny elementy, kterým je možné toto mapování přiřadit, včetně těch, které již namapovány byly, ale toto mapování má lepší cenu, než jim přiřazené. Kontejnery a vnořené AUI se mapují jako v prvním případě.

4. Vybere se jedno možné mapování nenamapovaného elementu a stejně se namapují všechny elementy, kterým je možné toto mapování přiřadit. Mapování se přiřazuje vždy. Kontejnery a vnořené AUI se mapují jako v prvním případě.

5. Vybere se jedno možné mapování nenamapovaného elementu a stejně se namapují všechny elementy, pro které je to nejlepší možné mapování z jim přístupných mapování. Přiřazují se pouze ty, které ještě nebyly namapovány. Kontejnery a vnořené AUI se mapují jako v prvním případě.

S posledními čtyřmi strategiemi mapování není optimalizační algoritmus úplným algoritmem průchodu stavového prostoru. Je heuristickým algoritmem. Všechny strategie jsou porovnány z hlediska rychlosti optimalizace a kvality vygenerované CUI v kapitole 4.2.

2.3 Generátor AUI z JPA modelu

Generátor AUI využívá nástroje AspectFaces popsaného v kapitole 1.2.

Díky flexibilitě, kterou AspectFaces nabízí, co se týče výstupu vygenerovaného z JPA entity, bude AspectFaces využit pro vygenerování nejenom požadovaného AUI, ale i modelů UIProtocolu, nezbytných pro funkčnost AUI a event handlerů v jazyce JavaScript, které budou zajišťovat validaci formuláře vygenerovaného z AUI před jeho odesláním na server. Součástí generátoru budou čtyři mapování a příslušné tagy.

Myšlenka vyslovená v kapitole 1.2.4, že díky objektům vložených do kontextu by bylo možné přes textová mapování a Java Unified Expression Language generovat výstup v objektové podobě, bude využita při generování modelu UIProtocolu s lokalizovanými textovými řetězci z uživatelského rozhraní. Lokalizované textové řetězce se budou hledat podle určitého klíče ve skupině Resource bundlů předaných generátoru při spuštění generování AUI.

Generátor bude možné využít jak k dynamickému vygenerování jednoho AUI, tak celé UIProtocol aplikace ve formátu [3] s několika AUI.

2.3.1 Inicializace generátoru

Při prvním vytvoření instance generátoru představovaného třídou `UipAuiAppGenerator` dojde k inicializaci nástroje AspectFaces, zaregistrování anotací, které bude AspectFaces inspektovat v JPA entitách, k inicializaci mapování a s nimi sdružených tagů.

AspectFaces momentálně neumí mimo případu, kdy je spuštěn ze servletu, načítat mapování a tagy, která jsou součástí knihovny jazyka Java, JAR souboru. tj. načítat mapování a tagy z class path. Generátor AUI ale bude potřebovat načítat soubory z class path, protože ponese všechna mapování a tagy v sobě a nebude spouštěn ze servletu. Nepředpokládá se, že by vývojář využívající generátor AUI jakkoliv měnil připravená mapování a tagy.

O načítání souborů s tagy se stará implementace abstraktní třídy

`Configuration`. Bude implementována třída `ClassPathConfiguration`, která bude umožňovat načítání tagů z libovolného místa na class path, tj. z JAR, WAR, EAR archivů a ze složek souborového systému.

O načítání mapování se stará potomek třídy `ConfigurationHolder`. Neexistuje způsob, jak vytvořit a začlenit do AspectFaces nového potomka této třídy bez úpravy kódu AspectFaces. Existuje však implementace `ConfigurationHolder` nazvaná `UrlConfigurationHolder`, která načítá soubory s mapováním podle zadané Unified Resource Locator (URL). Pomocí URL ale není možné v Javě vyjádřit cestu na class path. Součásti adresy URL je specifikace protokolu, který se použije pro načtení zdroje, který URL představuje. Pro načítání souborů z class path žádný protokol neexistuje. Toto omezení lze však obejít. V Javě je možné třídě `URL`, která představuje adresu URL, předat implementaci třídy `URLStreamHandler`, která dokáže přidat podporu nového protokolu pro načítání souborů z URL. Bude vytvořena implementace pro načítání souborů z class path, viz. [40].

2.3.2 Architektura generátoru

Hlavní třídou, která představuje generátor AUI, je `UipAuiAppGenerator`. Metodami této třídy se spouští generování AUI z tříd JPA datového modelu.

Jako vstup pro generování AUI slouží třída `FormGeneratorSource`, která obsahuje všechny nezbytné údaje pro vygenerování jednoho AUI.

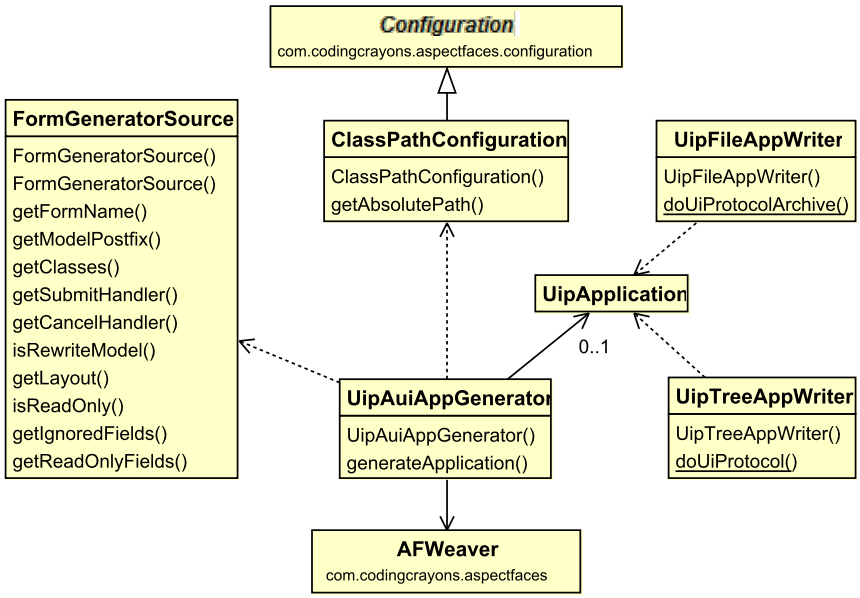
Výstupem generátoru je instance třídy `UipApplication`, která sdružuje všechny modely, interface a event handlery vytvořené generátorem.

Z instance třídy `UipApplication` dokáže třída `UipFileAppWriter` vytvořit balíček s UIProtocol aplikací podle [3] a uložit jej na disk. Třída `UipTreeAppWriter` dokáže z `UipApplication` vytvořit objektovou reprezentaci jednoho UIProtocol dokumentu, který bude obsahovat všechny vygenerované modely, interface a event handlery uložené jako property v jedné event.

2.3.3 Vstup generátoru a generování

Při generování jednoho AUI je AspectFaces spouštěn celkem čtyřikrát se čtyřmi mapováními a příslušejícími tagy.

Budou vytvořeny tagy pro mapování primitivních datových typů Javy, jejich objektových protějšků, Enum a pro mapování JPA entit, výčtových typů, enumů, které se ukládají do databáze. Taková entita musí implementovat rozhraní `NameableIdentifiable`, které bude součástí generátoru



Obrázek 2.13: Architektura AUI generátoru z JPA datového modelu

AUI. Tagy budou vytvořeny tak, aby braly v potaz všechny možné informace z JPA entit specifikované pomocí anotací. U abstraktních elementů pro zadávání čísel se tak bude brát v potaz při validaci nastavení číselného rozsahu, u elementů pro zadávání textu délka textu a regulární výraz a jiné. Enumy a výčtové databázové typy se budou mapovat na stejný abstraktní ovládací prvek umožňující výběr jedné možnosti z několika specifikovaných.

Budou vytvořeny tagy jak pro vstupní ovládací prvky, tak pro ovládací prvky sloužící pro zobrazování hodnot atributů JPA entit. V mapování se vybere příslušný tag podle podmínky zapsané v Java Unified Expression Language, která bude kontrolovat, zda je atribut entity označen jako pouze pro čtení.

Pro validaci dat zadaných ve vstupních ovládacích prvcích bude sloužit skupina univerzálních event handlerů v jazyce JavaScript. V tazích představujících abstaktní ovládací prvky AUI bude specifikováno spouštění event handleru pro kontrolu vstupu pomocí syntaxe UIProtocolu - behavior.

Jednotlivá mapování a skupiny tagů mají následující účel:

- Vytvoření vlastního popisu AUI v XML syntaxi UIProtocolu pro popis AUI. Do výsledného AUI bude možné volitelně vložit tlačítka pro uzavření AUI. Jedno tlačítko bude vyvolávat dále popsaný event handler

pro kontrolu všech vstupních polí a následně zavolá event handler definovaný v `FormGeneratorSource`. Druhé tlačítko pouze zavolá event hadler definovaný v `FormGeneratorSource`. První tlačítko je primárně určeno pro odeslání, uložení dat vyplněných v AUI, druhé pro opuštění AUI, zobrazení jiného interface. Zda se má některé z tlačítek do AUI umístit, bude možné určit použitím anotací `@UiAfter` nebo `@UiBefore` a parametrem `type` nastaveným na `submitButton` nebo `cancelButton`. Pokud nechce vývojář tlačítka umísťovat do entity, může vygenerované AUI dekorovat pomocí layoutu, ve kterém budou tlačítka umístěna (layout viz. kapitola 1.2.4).

- Vytvoření podpůrného modelu, ve kterém budou pro každý abstraktní ovládací prvek AUI uloženy výsledky validace vstupní hodnoty, tj. jestli se validace zdařila nebo ne. Tento model je spojen s popisem AUI. Abstraktní elementy AUI sloužící pro zadávání dat mohou mít property, která udává, zda se má zobrazit zpráva upozorňující na nevalidní data. Zpráva, která se zobrazí, je zadaná v property v tagu XML popisu AUI `label`.

- Vytvoření modelu, do kterého se budou ukládat data zadaná do vstupních ovládacích prvků.

- Vytvoření event handleru v jazyce JavaScript, který validuje všechna vstupní pole při stisknutí tlačítka pro odeslání formuláře vygenerovaného z AUI.

Do kontextu generátoru je přidána instance třídy `Util`. Ta umožňuje do tagů pomocí Java Unified Expression Language vkládat lokalizované řetězce. Přesněji, vkládají se pouze property UIProtocolu s key nastaveným na model s lokalizacemi a property tohoto modelu, kde je lokalizovaný řetězec uložen. Zároveň se při každém požadavku na vložený lokalizovaného řetězce generuje model UIProtocolu s lokalizací. Vkládají se do něj property s lokalizacemi, které byly z tagu požadovány poprvé.

Generování UIProtocol aplikace a AUI se spouští zavoláním metody `generateApplication` třídy `UipAuiAppGenerator`. Generátor AUI vyžaduje při spuštění generování zadání několika parametrů specifikujících podobu vygenerované UIProtocol aplikace a AUI:

- Seznam instancí tříd `FormGeneratorSource`. Každá z těchto tříd sdružuje informace potřebné pro vygenerování jednoho AUI:

  - Název vygenerovaného AUI (pro UIProtocol to je třída, class, interfacu).
  - Koncovka id modelu - pro každou entitu, ze které se vytváří AUI, je vytvořen jeden model UIProtocolu, do kterého se ukládají data zadaná do AUI. Id modelu začíná celým názvem třídy entity včetně balíčku a končí koncovkou. Díky tomuto je možné najednou zobrazit více AUI vygenerovaných ze stejné entity. Součástí každého tohoto modelu je property s názvem `class`, ve které je uložen název třídy entity, ze které byl model vygenerován.
  - Seznam tříd entit, ze kterých se bude AUI generovat.
  - Id event handleru, který se zavolá po stisknutí volitelného odesílacího tlačítka popsaného dříve.
  - Id event handleru, který se zavolá po stisknutí volitelného uzavíracího tlačítka popsaného dříve.
  - Volba, zda se má do spolu s AUI vygenerovat i model pro uložení dat z AUI. Generování modelu se zakazuje v případě, že chce vývojář zobrazit ve vygenerovaném AUI data z již existujícího UIProtocol modelu.
  - Layout, kterým má být vygenerované AUI generováno. Zadává se cesta v class path k souboru s layoutem.
  - Volba, zda se má vygenerovat AUI určené pro zadávání dat v entitě nebo pro zobrazování dat.
  - Seznam atributů entity, které se nebudou brát v potaz při generování AUI.
  - Seznam atributů entity, které budou vždy jen pro čtení.
  - Seznam rolí uživatelů, pro které se má AUI vygenerovat.

- Volba, zda se mají do výsledné UIProtocol aplikace zařadit předpřipravené modely a event handlery pro validaci vstupu jednotlivých abstraktních elementů. Tyto modely a event handlery, šablona UIProtocol aplikace podle [3], budou součástí generátoru AUI.

- Třída UIProtocol interfacu, který se zobrazí jako první klientovi po připojení k instanci UIProtocol aplikace.

- Volba, zda se mají načíst další dokumenty UIProtocolu z class path, a cesta k nim. Vývojář může specifikovat další dokumenty UIProtocolu, které se stanou součástí aplikace.

- Volba, zda se mají načíst další dokumenty UIProtocolu ze souborového systému, a cesta k nim.

Generátor nemusí generovat celou aplikaci. Při uvedení jediného `FormGeneratorSource` a zákazu přidávání předpřipravených dokumentů UIProtocolu generuje jediný AUI a jeho podpůrné modely a event handlery.

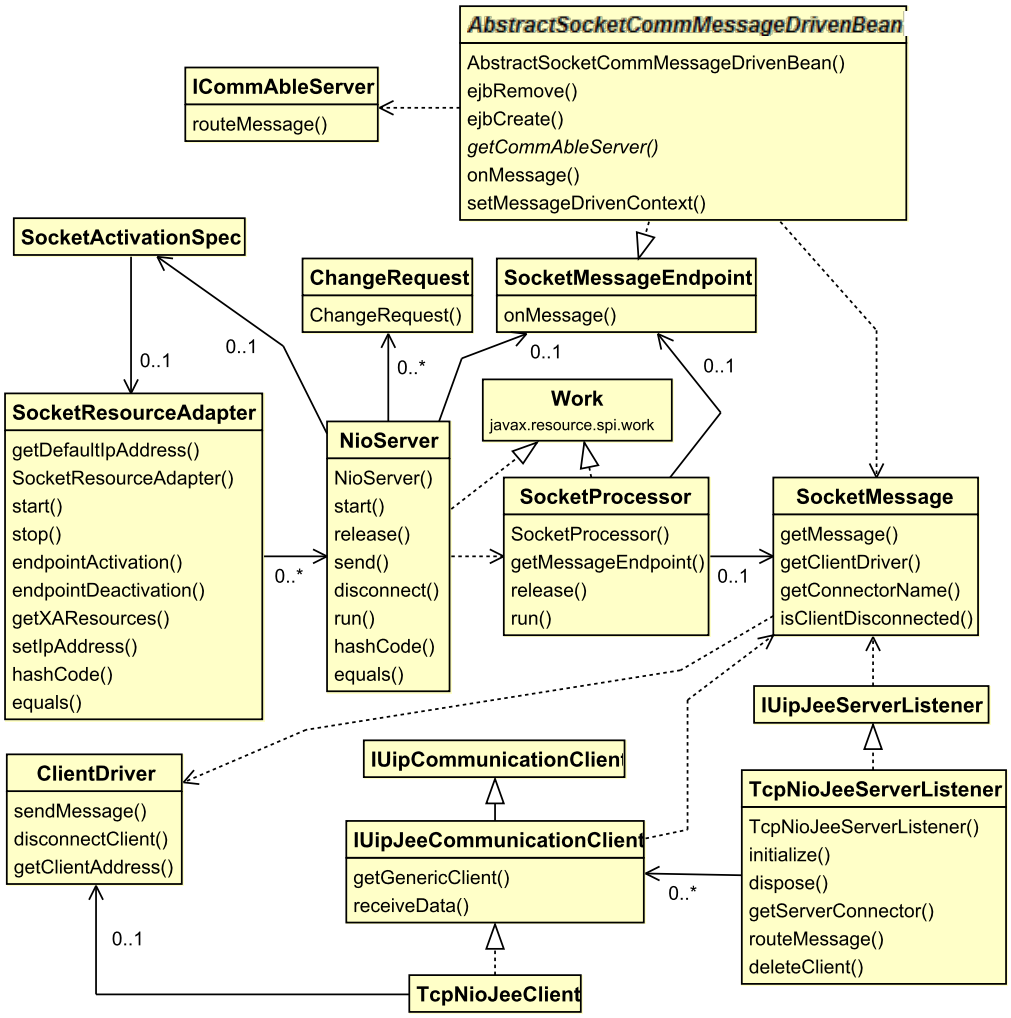
2.4 Integrace UIProtocolu do JEE pomocí JCA

Návrh integrace UIProtocolu do Java EE pomocí JCA vychází z analýzy provedené v kapitole 1.8.1. Architektura systému integrace je zobrazena na obrázku 2.14. Nazývejme dále systém integrace UIProtocolu do JEE pomocí JCA **Java EE UIP Socket Connector (JEEUSC)**.

Inicializace JEEUSC je znázorněna na obrázku 2.15. Spuštění Resource adapteru je podmíněno vytvořením instance implementace abstraktní třídy `AbstractSocketCommMessageDrivenBean`. Tu vytváří Java EE aplikační server. Tato instance musí být Message Driven Beanou. Jedna Message Driven Bean může inicializovat jeden JEEUSC. `AbstractSocketCommMessageDrivenBean` implementuje rozhraní `SocketMessageEndpoint`, jež páruje Resource adapter se všemi potomky `AbstractSocketCommMessageDrivenBean`. Inicializace Message Driven Bean zapříčiní i inicializaci a spuštění Resource adapteru `SocketResourceAdapter`. V `SocketActivationSpec` jsou obsažena nastavení portu a rozhraní, na kterém bude JEEUSC naslouchat příchozím spojením a jednoznačný identifikátor JEEUSC resource adapteru. Každý JEEUSC resource adapter inicializovaný pomocí Message Driven Bean musí mít svůj jednoznačný identifikátor.

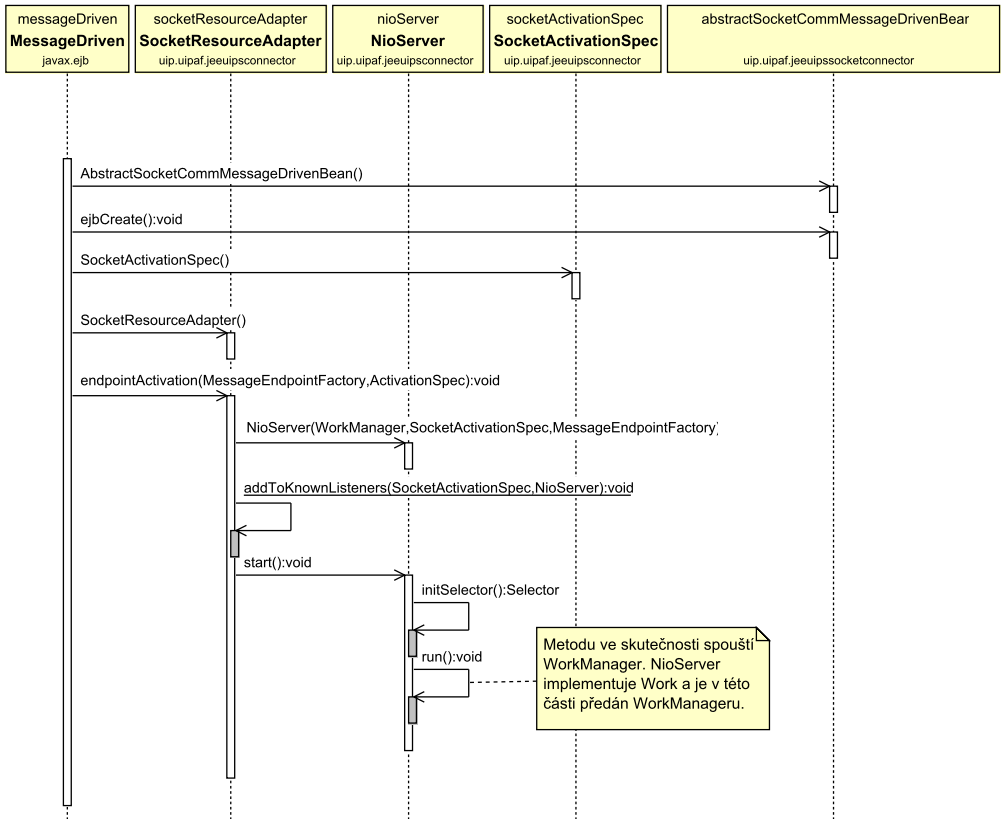
Při inicializaci `SocketResourceAdapter` se v metodě `endpointActivation` vytvoří nová instance třídy `NioServer`. Ta bude implementovat síťovou komunikaci pomocí neblokujícího síťového API - Non blocking I/O, podle článku [16]. Veškerá práce se síťovým rozhraním, tj. připojování a odpojování klientů, přijímání a odesílání zpráv, probíhá v jednom vlákně. `NioServer` přijatá surová data nijak nezpracovává, uloží je do instance třídy `SocketMessage` a předá ke zpracování `WorkManageru` v implementaci rozhraní `Work SocketProcessor`. Zpracování `SocketProcessor`, a tak i samotné přijaté zprávy je spuštěno `WorkManagerem` v jiném vlákně.

V `SocketMessage` je uložen identifikátor resource adapteru JEEUSC.



Obrázek 2.14: Návrh integrace UIProtocolu do Java EE pomocí JCA (Java EE UIP Socket Connector)

Tak je možné zjistit, pomocí kterého resource adapteru byla zpráva přijata. V `SocketMessage` je také uložena instance třídy `ClientDriver`. Ta jednoznačně identifikuje síťové spojení vytvořené pro komunikaci s klientem, a tím i samotného připojeného klienta. Prostřednictvím třídy `ClientDriver` je možné zařazovat do fronty `NioServeru` požadavky na odeslání dat klientovi nebo odpojení klienta od serveru. Jeden požadavek představuje třída `ChangeRequest`. `NioServer` požadavky zpracovává ve vlákně, ve kterém je spuštěn.



Obrázek 2.15: Inicializace (Java EE UIP Socket Connector)

`AbstractSocketCommMessageDrivenBean` bude zprávy přijaté od klienta předávat instanci třídy implementující rozhraní `ICommAbleServer`.

Modelovým návrhem je, že třída implementující rozhraní `ICommAbleServer` je singleton EJB inicializovaná při startu Java EE aplikačního serveru. Tato EJB bude obsahovat mírně upravený komunikační subsystém představený v kapitole 2.1.3, nazvěme jej **komunikační subsystém pro JEE**. Bude se lišit od komunikačního systému v kapitole 2.1.3 ve třech detailech. Ostatní rozhraní i jejich implementace sdílí s komunikačním systémem z kapitoly 2.1.3.

Místo rozhraní `IUipServerListener` bude rozhraní `IUipJeeServerListener`. Implementace `IUipJeeServerListener` nebude na rozdíl od `IUipServerListener` otevírat síťové rozhraní a naslouchat příchozí komunikaci. K tomu slouží instance třídy `NioServer`.

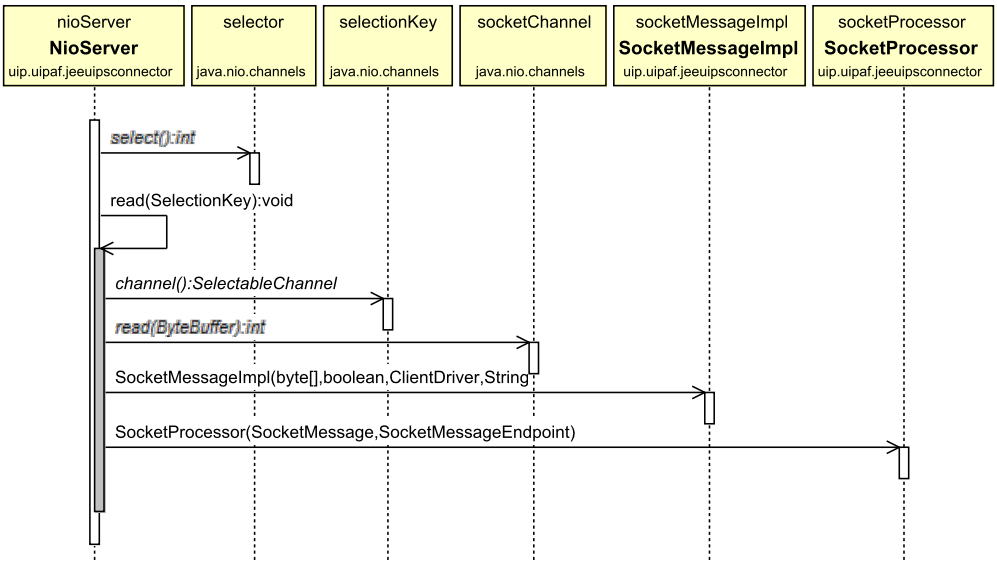
Místo managera `UipCommunicationManager` bude v komunikačním sub-

systému pro JEE nový manager schopný spravovat `IUipJeeServerListener`. Manager bude implementovat rozhraní `ICommManager`. Singleton EJB `ICommAbleServer` obsahuje managera `ICommManager`.

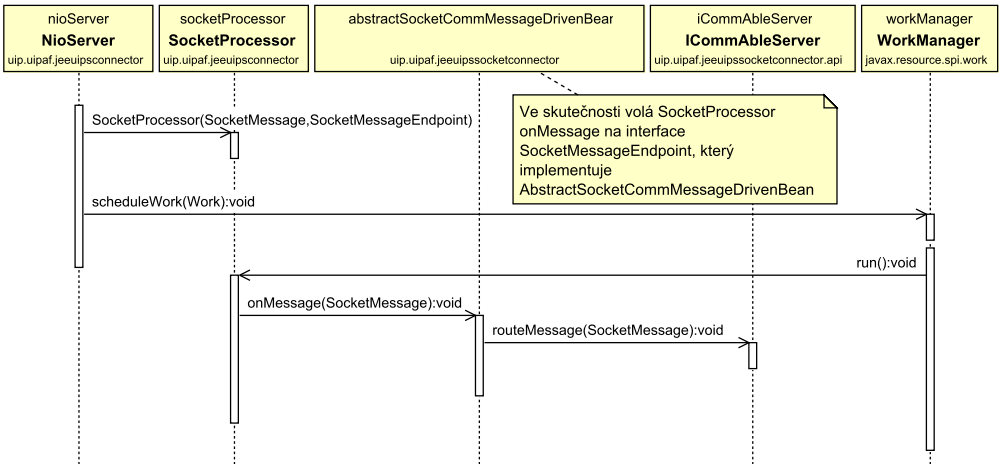
2.4.1 Přijetí zprávy

`ICommAbleServer` předá instanci `ICommManager` přijatou zprávu v instanci `SocketMessage`. `ICommManager` zjistí ze zprávy identifikátor resource adapteru JEEUSC. Podle identifikátoru hledá v seznamu, který udržuje, `IUipJeeServerListener`. Pokud ho nenajde, může se chovat implementace rozdílně. Buď vytvoří novou instanci `IUipJeeServerListener` nebo zprávu bez upozornění ignoruje. Každý resource adapter JEEUSC je spárován s jedním `IUipJeeServerListener`.

`ICommManager` předává zprávu `IUipJeeServerListener`. Ten udržuje seznam připojených klientů `IUipJeeCommunicationClient`. Toto rozhraní je rozšířením rozhraní `IUipCommunicationClient`. Navíc přidává metodu dovolující přijmutí zprávy `SocketMessage`. `IUipJeeServerListener` extrahuje z přijaté zprávy `ClientDriver` a hledá podle něj ve svém seznamu instanci `IUipJeeCommunicationClient`. Pokud ji nenajde, vytvoří novou. Vytvoří také podobně jako `IUipServerListener` pomocí `IClientFactory` instanci třídy implementující rozhraní `IGenericClient`.



Obrázek 2.16: Přijetí zprávy, část 1 (Java EE UIP Socket Connector)



Obrázek 2.17: Přijetí zprávy, část 2 (Java EE UIP Socket Connector)

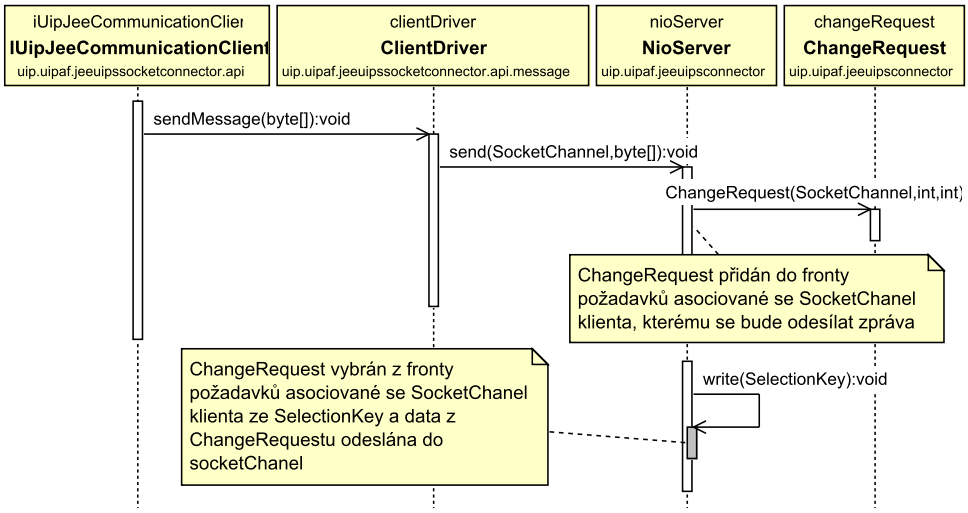
Každému klientovi připojenému ke komunikačnímu kanálu odpovídá jedna instance `ClientDriver` spárovaná s instancí `IUipJeeCommunicationClient`.

`SocketMessage` je předána `IUipJeeCommunicationClient`. Ten získá ze zprávy surová data odeslaná klientem - pole bajtů. Jelikož pole bajtů nemusí představovat celou zprávu ve formátu UIProtocol nebo jich může naráz přijmout více, musí si `IUipJeeCommunicationClient` udržovat buffer přijatých dat. Přijatá data jsou vložena do bufferu. Když je v bufferu identifikována kompletní zpráva, dokument UIProtocolu, je zpráva rozparsována a je z ní vytvořena objektová reprezentace UIProtocolu. O identifikaci zprávy a vytvoření objektové reprezentace se stará `IUIPTreeFactory`. Rozparsovaná zpráva je nakonec předána `IGenericClient`.

2.4.2 Odeslání zprávy

Odeslání zprávy iniciuje `IGenericClient`. Předá zprávu v objektové reprezentaci UIProtocolu `IUipJeeCommunicationClient`. Ten ji pomocí `IUIPTreeFactory` převede do reprezentace, která se posílá přes síťové spojení, např. do XML, a vytvoří z ní pole bajtů. Odeslání zprávy je možné naplánovat `NioServeru` zavoláním metody `sendMessage` třídy `ClientDriver`. V `NioServer` je vytvořena instance třídy `ChangeRequest` - požadavek na odeslání dat a je zařazen do fronty požadavků. Podobně se do této fronty vkládá i požadavek na odpojení klienta od serveru. Při zařazení požadavku do fronty je odblokováno hlavní vlákno, ve kterém běží `NioServer`. Ten

postupně vybírá požadavky z fronty a odesílá data po síťovém spojení nebo odpojí klienta.



Obrázek 2.18: Odeslání zprávy (Java EE UIP Socket Connector)

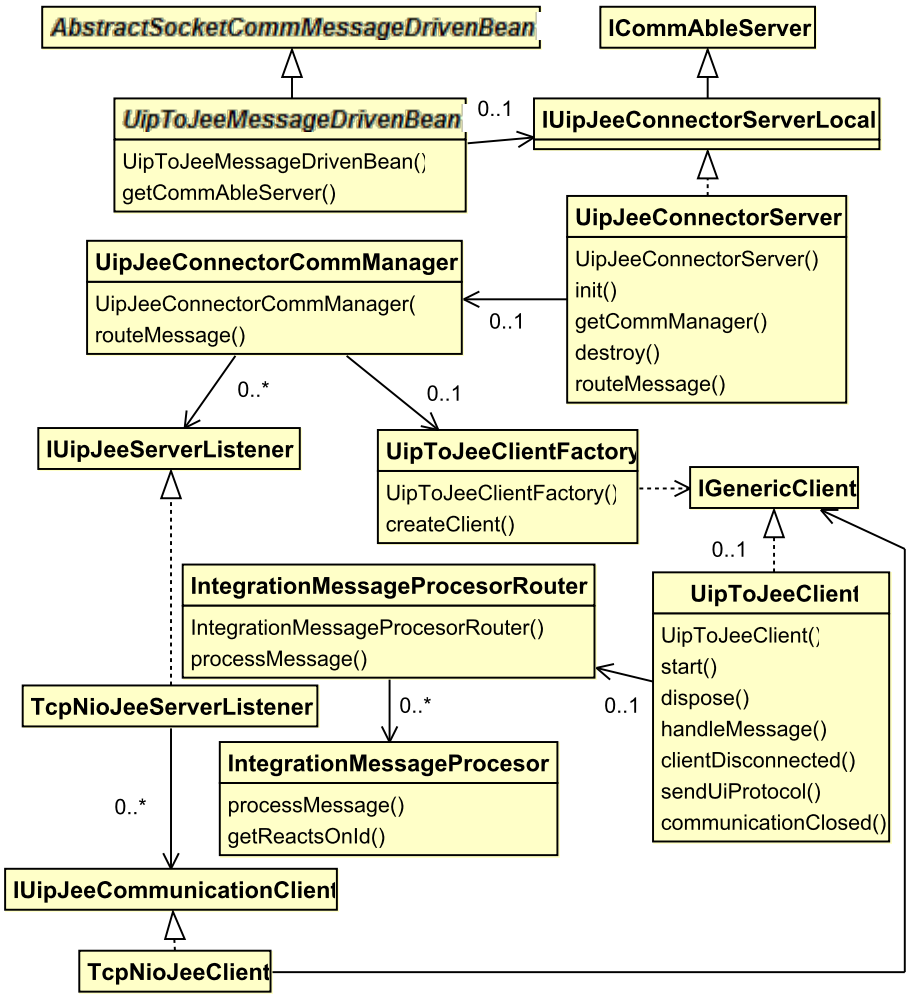
2.5 UIP to Java EE Connector

UIP to Java EE Connector zcela využívá JEEUSC a upraveného komunikačního subsystému popsaného v kapitole 2.4.

2.5.1 Strana Java EE aplikace

Implementace `AbstractSocketCommMessageDrivenBean` Message Driven Bean (MDB) `UipToJeeMessageDrivenBean` bude čekat na předání přijaté zprávy od resource adapteru JEEUSC. Zprávy budou předávány singleton EJB `UipJeeConnectorServer`, implementaci rozhraní `ICommAbleServer`. Součástí `UipJeeConnectorServer` je manager `UipJeeConnectorCommManager`, který schraňuje všechny resource adaptery JEEUSC naslouchající příchozím spojením spárované s `IUipJeeServerListener`.

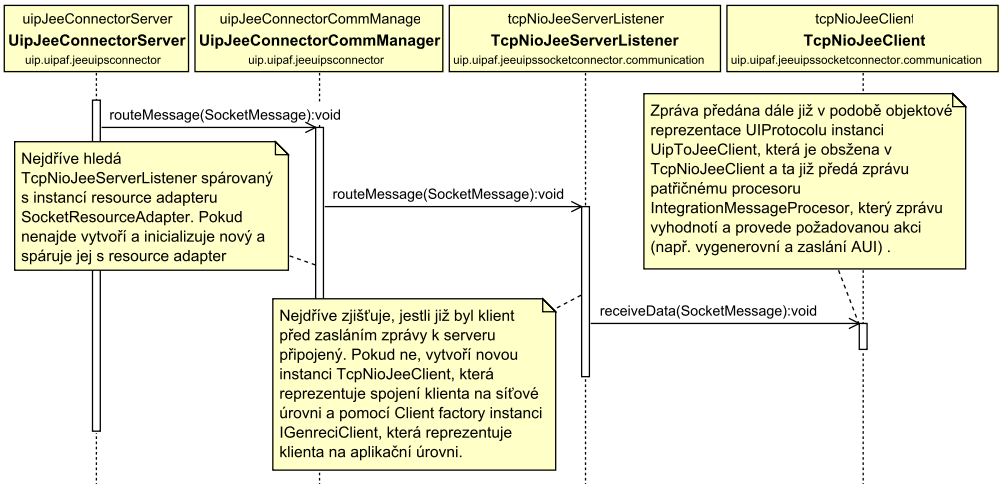
Při připojení klienta je pomocí implementace `IClientFactory` třídy `UipToJeeClientFactory` vytvořena instance třídy `UipToJeeClient`. Ta se stará o vlastní zpracování přijaté integrační zprávy. `UipToJeeClient` pracuje se zprávou v objektové podobě UIProtocolu.



Obrázek 2.19: Architektura UIP to Java EE Connector

Zpráva má podobu plnohodnotného dokumentu UIProtocolu. Může obsahovat modely, zřídka interface. Každá zpráva musí povinně obsahovat event. Podle id event UIP to Java EE Connector identifikuje, jaká integrační akce se má vykonat. Property eventy jsou parametry integrační akce. Modely zaslané spolu s event mohou představovat data, která se mají uložit do databáze nebo předat v podobě Java objektu některé metodě EJB.

Zprávy třídí podle id eventy třída `IntegrationMessageProcesorRouter`. Ta udržuje seznam implementací `IntegrationMessageProcessor`. Každá z těchto implementací umí obsloužit jednu integrační zprávu se specifickým identifikátorem.



Obrázek 2.20: Přijetí integrační zprávy v UIP to Java EE Connector

2.5.1.1 Vytváření a zpracování zpráv

Jak již bylo řečeno, klient může integrátoru zaslat v integrační zprávě i model UIProtocolu. Každý model reprezentuje instanci jedné třídy v jazyce Java. Každý zaslaný model musí obsahovat property se jménem `class`. Hodnotou property je jméno Java třídy včetně balíčku, kterou model představuje. Property modelu, které mají být uloženy jako atributy třídy, musejí mít stejný název jako tyto atributy. Při přijetí modelu se tento převádí na instanci Java třídy. Podle property `class` se vytvoří za využití reflexe instance třídy. Pro každou property, která představuje atribut třídy, se ve třídě hledá setter, metoda sloužící k nastavení hodnoty atributu. Pokud je setter nalezen, zjistí se pomocí reflexe, jakého typu je parametr settru, tj. jakého typu je atribut třídy, který nastavuje. Na tento datový typ se převede hodnota property modelu představující atribut a zavolá se setter. Bude podporován převod primitivních datových typů jazyka Java a jejích datových protějšků, Enumů a Date.

Pokud nebude datovým typem atributu ani jeden z předchozích, předpokládá se, že instance požadovaného datového typu byla zaslána jako jeden z modelů integrační zprávy nebo že se má načíst z databáze. Pokud je hodnotou property modelu představující atribut textový řetězec, předpokládá se, že se jedná o identifikátor jiného modelu zaslaného v integrační zprávě. Tento model se rekurzivně převede pomocí stejné logiky na instanci třídy jazyka Java. Pokud je hodnotou property celé číslo a navíc atribut třídy je potomkem rozhraní `Identifiable` distribuovaného spolu s generátorem

AUI, předpokládá se, že hodnotou property je identifikátor JPA datové entity. Entita se podle id pokusí načíst pomocí JPA z databáze.

Díky tomuto systému převodu UIProtocol modelů na objekty jazyka Java bude možné posílat v integračních zprávach složité struktury, ukládat do databáze několik vnořených entit najednou. V případě, kdy klient integrace nepotřebuje znát celé databázové entity, stačí mu zaslat jen id entity a integrátor je bude schopný identifikovat a načíst z databáze v případě potřeby.

Obdobným způsobem funguje opačný převod z Java objektu, JPA entity, na modely. Navíc je možné převádět na modely seznamy objektů. Z každého objektu v seznamu se vygenerují modely. Identifikátory modelů jsou očíslovány tak, aby měl každý model jedinečné id. Navíc vznikne model, ve kterém je uložen počet entit v seznamu a identifikátory všech vzniklých modelů.

2.5.1.2 Služby integrace

Bude implementováno celkem pět `IntegrationMessageProcessor`ů. Každý bude představovat jinou službu integrace. Každý bude identifikován jedinečným id. Některé budou předpokládat v event, která je součástí přijaté integrační zprávy, parametry nutné pro konfiguraci integrace, jiné pouze modely a některé oboje. UIPServer bude moci vyžadovat od integrátoru následující služby:

- Vygenerování AUI z JPA modelu. Generuje se jeden AUI. Integrátor očekává v event property udávající, z jakých JPA tříd se AUI vygeneruje, jestli bude AUI pouze pro čtení a další parametry popsané v kapitole 2.3.3. AUI se generuje pomocí Java UiGE, viz. kapitola 2.3. Vygenerovaný AUI v podobě třídy `UipApplication` se převede pomocí `UipTreeAppWriter` do jednoho UIProtocol dokumentu, jeho objektové podoby. Tento dokument se následně odešle pomocí `UipToJeeClient`, na který má každý `IntegrationMessageProcessor`, zpět integračnímu klientovi.

- Přijetí dat v podobě UIProtocol modelu a převedení tohoto modelu na JPA entitu. Persistence entity do databáze pomocí JPA třídy `EntityManager`. V eventě v integrační zprávě nejsou očekávány žádné property. Zpět je klientovi odeslána entita, jak byla změněna po uložení do databáze - v entitě se při uložení může vygenerovat identifikátor.

- Spuštění uloženého dotazu JPA, který načítá data z databáze. Jako property event se předpokládají jméno uloženého dotazu a jeho parametry. Zpět jsou klientovi odeslána načtená data v podobě UIProtocol modelů.

- Spuštění uloženého dotazu JPA, který mění data v databázi nebo ukládá data do databáze. Jako property event se předpokládají jméno uloženého dotazu a jeho parametry. Zpět je klientovi odeslána pouze informace o úspěšném vykonání dotazu.

- Spuštění metody EJB. Jako property event se předpokládá název metody, seznam parametrů metody a jejich hodnot, s kterými se má metoda zavolat, a JNDI název a cesta k požadované EJB. Instance EJB je nalezena pomocí třídy `InitialContext`. Kde přesně bude JNDI hledat EJB tak závisí na nastavení JNDI v Java EE aplikaci. Když je EJB nalezena, hledá se pomocí reflexe metoda, která se má spustit. Hledá se podle jména a seznamu parametrů. Když je metoda nalezena, spustí se se všemi dodanými parametry. Návratová hodnota metody je převedena na UIProtocol modely a zaslána zpět klientovi integrace.

2.5.2 Strana UIProtocol serveru

UIProtocol aplikace bude k modulu pro integraci na straně Java EE JPA aplikace přistupovat prostřednictvím sady autonomních event handlerů (kapitola 1.6.1) nasazených na UIPServeru. Každý z těchto event handlerů bude obsluhovat jednu integrační službu. Každá tato funkce bude identifikována jednoznačným id, aby mohla být vyvolána pomocí eventy s tímto id. Budou vytvořeny dvě sady autonomních event handlerů.

Jedna sada bude určena pro Java SE i EE. Event handlery budou implementovat rozhraní `IUipAutonomousEventHandler`. Integrační zpráva se odešle integračnímu serveru v Java EE JPA aplikaci pomocí subsystému UIPServeru pro zasílání síťových zpráv zmiňovaného na konci kapitoly 1.6.1. Pomocí stejného subsystému také přijme odpověď. Adresa integračního serveru Java EE aplikace, se kterou má UIProtocol aplikace integrovat, bude součástí distribučního balíčku UIProtocol aplikace [3] a bude uložena jako property v modelu `public.application`, viz. tabulka 1.2.

Druhá sada bude určena pro Java EE. Event handlery budou implementovat rozhraní `IUipJeeAutonomousEventHandler`. Tato sada bude určena pro UIProtocol server integrovaný do Java EE aplikace popsaný v další kapitole. Event handlery nebudou posílat integrační zprávy po síti. Budou rovnou volat implementaci služeb integrace představovaných rozhraním

`IntegrationMessageProcessor`. Každý event handler v sobě bude obsahovat instanci `IntegrationMessageProcessor` příslušné služby integrace, kterou má volat.

V UIPServeru pro Java EE budou moci být využity obě sady naráz. Jedna UIProtocol aplikace nasazená spolu s UIPServerem na Java EE bude tak moci integrovat s aplikací na jiném Java EE aplikačním serveru.

Obě sady event handlerů budou, až na způsob kontaktování integrační služby, sdílet veškerou funkcionalitu. Event handler tak po zavolání sestaví integrační zprávu a předá ji integrační službě. Po přijetí odpovědi uloží všechny přijaté modely, interface a event handlery do soukromého úložiště UIProtocol dokumentů klienta UIProtocolu (viz. [2]), který zaslal požadavek na integraci. Nyní může UIProtocol klient požádat UIPServer o dokumenty UIProtocolu vzniklé při integraci.

2.6 Integrace UIPServeru do Java EE

Díky nové modulární architektuře UIPServeru (kapitola 2.1) a integraci UIProtocolu do Java EE (kapitola 2.4) je možné integrovat UIPServer do Java EE s minimálním úsilím.

Hlavní třídou UIPServeru bude třída `UipJeeServer`, ta bude implementovat rozhraní `IUipServer` (viz. kapitola 2.1) a zároveň rozhraní `ICommAbleServer` (kapitola 2.4). `UipJeeServer` bude singleton EJB spouštěnou při startu Java EE aplikačního serveru.

MDB pro příjem zpráv od resource adapteru JEEUSC bude představovat abstraktní třída `UipServerMessageDrivenBean`. Instance této třídy má k dispozici referenci na singleton EJB `UipJeeServer`. Pro příjem zpráv je využíván upravený komunikační subsystém z kapitoly 2.4. Pro správu síťových rozhraní naslouchajících klientům bude sloužit implementace rozhraní `ICommManager CommInterfaceCommunicationManager`. Ta podle konfigurace UIPServeru vytvoří instance `IUipJeeServerListener`.

Pro vytvoření síťového rozhraní naslouchajícímu příchozím spojením je nutné vytvořit v aplikaci, do které se bude UIPServer integrovat, implementaci `UipServerMessageDrivenBean`. Je nutné nastavit pomocí anotací konfiguraci pro `SocketActivationSpec`, adresu a port, na kterém bude rozhraní naslouchat, a jednoznačný identifikátor resource adapteru JEEUSC. Tento identifikátor se uvede v konfiguraci UIPServeru u příslušného `IUipJeeServerListener`, a tak budou spárováni s resource adapterem. Tak může `IUipJeeServerListener` reagovat jen na zprávy, které jsou mu ur-

čeny.

Jelikož v EJB není možné vytvářet vlákna, je zakázáno v UIPServeru vytváření veškerých vláken. Zpracování jednoho požadavku klienta UIProtocolu tak vždy proběhne celé v jednom vlákně.

Navíc bude vytvořena implementace ovladače pro načítání souborů UIProtocol aplikací. Ta bude umožňovat načítat soubory UIProtocol aplikací z class path. Díky tomu bude možné do jednoho EAR archivu Java EE aplikace integrovat jak samotnou Java EE aplikaci, UIPServer, tak i UIProtocol aplikaci, která se bude na UIPServeru spouštět.

Bude ale možné vytvořit pouhou obálku pro UIProtocol server v podobě Java EE aplikace, která nebude dělat nic jiného, než že spustí integrovaný UIProtocol server. UIPServer tak bude fungovat stejně jako verze pro Java SE, tj. bude moci komunikovat s UIPPortalem, načítat soubory UIProtocol aplikací pomocí FTP, ale navíc přinese podporu vlastností a technologií platformy Java EE jako podporu persistence dat, transakční zpracování a mnohé další.

Kapitola 3

Realizace

Veškerá funkcionalita popsaná v kapitole 2 byla implementována. Tato kapitola se zabývá popisem implementace zajímavých částí systému a těch částí, kde bylo nutné řešit implementační problémy. V kapitole bude představena struktura knihoven systému a v závěru bude popsána implementace testovací aplikace.

3.1 Projektová infrastruktura a Maven

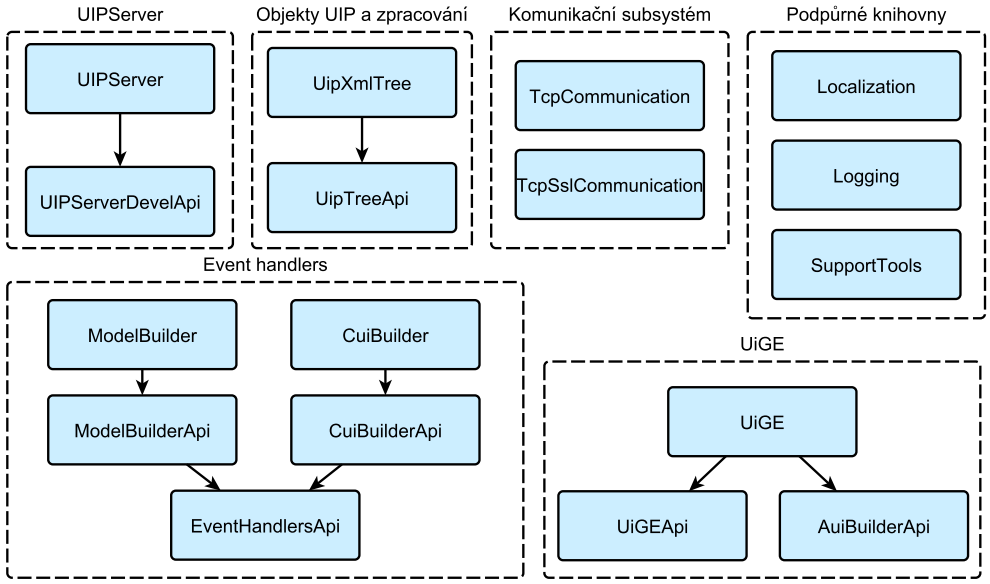
Veškeré zdrojové kódy projektu byly verzovány pomocí systému Subversion (SVN) pro správu a verzování zdrojových kódů. Zdrojové kódy byly ukládány do dvou repositářů. Část kódu, u které je jisté, že se bude šířit pod open source licencí Apache License Version 2.0, byla ukládána do repositáře hostovaného na **Sourceforge.net**. Zde bude uložen UIPServer. Projektové stránky jsou k dispozici na adrese [4]. Ostatní části systému jsou verzovány v neveřejném repositáři Katedry počítačové grafiky a interakce Fakulty elektrotechnické.

Zdrojové kódy projektu jsou sestavovány pomocí nástroje pro řízení a automatizaci sestavení aplikací Apache Maven. Maven má proti druhému nejčastěji používanému nástroji pro sestavování aplikací Apache ANT několik výhod. Konfigurace Apache Maven je jednodušší a jeho konfigurační soubory jsou snáze čitelné. Hlavní výhodou je ale systém repositářů umístěných na Internetu s aplikačními knihovnami. V konfiguraci Apache Maven je definováno, jaké knihovny bude program používat. Při sestavování aplikace jsou tyto knihovny automaticky staženy z Internetu. Projekty pro Apache Maven jsou podporovány mnoha vývojovými prostředími pro jazyk Java. Programátor tak není nucen používat konkrétní vývojové prostředí. Každý

projekt v Maven je knihovnou, která může být nahrána do repositáře knihoven. Na počítači, kde je Maven nainstalován, se vytváří lokální repositář knihoven, do kterého se ukládají vyvíjené knihovny. Vývojář tak nemusí nahrávat knihovny do veřejné repositáře jen proto, aby mohl sestavovat, spouštět a testovat aplikaci během vývoje.

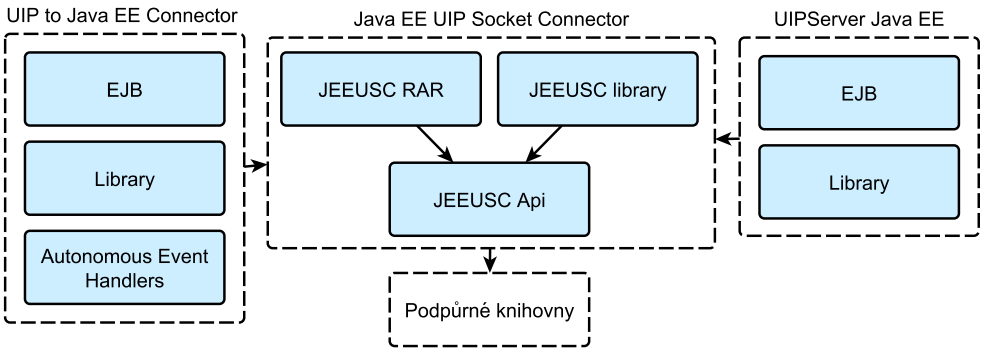
3.2 Struktura knihoven

Systém je rozdělen do sady knihoven tak, aby bylo možné využívat knihovny se sdílenou funkcionalitou ve více projektech. Každé knihovně odpovídá jeden Maven projekt. Pro každou funkcionalitu, subsystém, většinou existuje jedna knihovna s Application Programming Interface (API) subsystému a poté skupina knihoven, různé implementace API.



Obrázek 3.1: Struktura knihoven UIPServeru a UiGE

UIPServer exponuje veškeré důležité API v knihovně `UIPServerDevelApi`. Součástí této knihovny je i API komunikačního subsystému. Objektová reprezentace UIProtocolu a knihovna pro parsování a generování XML verze UIProtocolu jsou na ostatních částech systému zcela nezávislé. Používají se i v jiných projektech. Nezávislá na ostatních projektech je také sada knihoven a API, které jsou využívány event handlery specifickými pro UIProtocol aplikaci.



Obrázek 3.2: Struktura knihoven JEEUSC, UIPServeru pro Java EE a UTJEEC

Téměř všechny projekty využívají sady podpůrných knihoven pro logování, načítání lokalizace a sadu utilit pro práci s řetězci, objekty, kolekcemi a další.

U aplikací, které se nasazují na Java EE aplikační server, je vždy zvlášť knihovna pro umístění všech EJB a zvlášť knihovna pro umístění ostatní logiky aplikace.

3.3 Parsování a generování XML

Vytváření objektové reprezentace UIProtocolu z XML a generování XML bylo nejprve realizováno za využití Java technologie JAXB. Ta umožňuje mapovat pomocí anotací třídy na XML dokumenty. Objektová reprezentace UIProtocolu byla opatřena potřebnými anotacemi. JAXB parsuje XML a vytváří pomocí reflexe objektovou reprezentaci. Při generování XML opět využívá reflexe k procházení objektové reprezentace. Není tak nutné vytvářet generátory XML a parsery. JAXB vše obslouží sám. Dokáže dokonce vygenerovat třídy pro objektovou reprezentaci XML ze schématu XML.

Jak se ukázalo během profilingu (viz. kapitola 4.1), velká část procesorového času spotřebovaného při běhu systému připadne na generování a parsování XML UIProtocol dokumentů. Proto byla hledána cesta, jak generování a parsování urychlit.

UIProtocol je navržen tak, aby při jeho rozšiřování nebylo nutné měnit schéma XML dokumentů. Výhody JAXB v podobě automatického generování tříd pro objektovou reprezentaci nebo univerzálního parseru a ge-

nerátoru se tak stávají nepodstatné. Mnohem lepší výkon při čtení XML by měl podávat SAX parser. Ten prochází XML a při navštívení každého tagu vygeneruje událost, zavolá metodu template třídy. SAX nevytváří sám objektovou reprezentaci. Vývojář musí vytvořit parser sám. Vytvoření parseru je jistě složitější než využití JAXB. Parser však může být významně optimalizován pro čtení XML dokumentů s jedním konkrétním schématem. Dle [41] je SAX v průměru nejvýkonnější ze všech technologií, které JAVA nabízí pro čtení XML. Tyto výsledky se potvrdily i v provedeném měření, viz. kapitola 4.3.

Hlavní součástí SAX parseru pro UIProtocol XML dokumenty je třída `UipXmlTreeReader`, ta je potomkem třídy `DefaultHandler` z Java SAX API. `DefaultHandler` vychází z návrhového vzoru template method [11]. Tato třída je předána SAX parseru. Ten volá při čtení XML sadu template metod při navštívení nového tagu, opuštění tagu, na začátku a na konci dokumentu a dalších událostech.

`UipXmlTreeReader` při zavolání metody značící navštívení nebo opuštění tagu vybere podle jména tagu strategii (podle [11]) pro zpracování tagu. Strategie pro zpracování konkrétního tagu implementuje rozhraní `IElementParser`. Strategie vytvoří objekt z objektové reprezentace UIProtocolu a naplní jej hodnotami z atributů tagu XML. Vytvořené objekty jsou ukládány na vrchol zásobníku, který udržuje povědomí o stromové struktuře UIProtocol dokumentu. Při opouštění tagu je zavolána patřičná metoda strategie, která odstraní objekt reprezentující tag z vrcholu zásobníku. Tak se parser posune o úroveň výše ve stromu UIProtocol dokumentu. Před tím, než je objekt uložen na vrchol zásobníku, získá se reference na objekt, který je na vrcholu zásobníku. Do tohoto objektu se musí uložit reference na právě vzniklý objekt. Tak se vytváří stromová struktura objektové reprezentace. Kontroluje se typ objektu z vrcholu zásobníku. Pokud tento v sobě nemůže uložit nový objekt, je parsovaný XML dokument nevalidní a parsování končí s chybou.

Pro generování XML UIProtocol dokumentu se používá třída `XMLStreamWriter`, která je součástí standardní knihovny jazyka Java. Ta umožňuje voláním metod pro vytvoření tagu, ukončení tagu a vytvoření atributu jednoduše vytvářet XML. Sama přitom udržuje informaci, kam do stromu XML dokumentu se má nový tag zapsat. Pro vygenerování XML z objektové reprezentace se využívá visitoru popsaného v kapitole 2.1.1. Implementace `UipTreeVisitor` třída `UipXmlTreeWriter` prochází rekurzivně stromovou strukturu objektové reprezentace UIProtocolu a vytváří XML.

Výsledky měření srovnávající výkon s generováním XML pomocí JAXB

jsou též uvedeny v kapitole 4.3.

3.4 Propojení Java a .NET

Z výkonnostních důvodů bylo při návrhu Java verze UiGE představeno API umožňující integrovat vykreslovací jádro klienta, pro kterého se CUI generují, přímo do UiGE (kapitola 2.2). Odpadá tak nutnost zdlouhavé síťové komunikace mezi UiGE a klientem, když potřebuje UiGE zjistit vzhled konkrétního ovládacího prvku, jak by se vykreslil na klientovi.

Klient, se kterým byl systém testován, není ale naprogramován v Java. Klient je určen pro platformu .NET a je naprogramován v jazyce C#. Není tak možné jednoduše integrovat vykreslovací jádro do Java verze UiGE.

Jistě zde není dobré, jako v případě UiGE, jít cestou návrhu a implementace Java verze klienta. Klient využívá specifická API platformy .NET pro vykreslování uživatelských rozhraní. Bylo by tak téměř, ne-li úplně nemožné zachovat funkcionalitu klienta při vykreslování uživatelského rozhraní. Jako jediné schůdné řešení je tak nějakým způsobem zajistit integraci .NET kódu, knihoven klienta, přímo do Java verze UiGE. Přesněji do jedné knihovny, která bude implementovat API pro integraci vykreslovacího jádra klienta.

Ukázalo se, že přímá integrace knihovny platformy .NET do Java aplikace je možná. Java může volat funkce nativních knihoven pomocí technologie Java Native Interface. Pro přístup k .NET by bylo nutné napsat sadu proxy v jazyce C nebo C++, které by umožňovali přistupovat ke knihovně aplikace platformy .NET. Toto však představuje psaní velkého množství těžko odladitelného a spravovatelného kódu.

Existuje však framework **jni4net** [36], který umožňuje elegantně obejít nutnost psaní kódu pro propojení Java a .NET knihovny. jni4net sám generuje potřebný integrační kód. Integrační kód je vygenerován z knihovny platformy .NET. Jsou vygenerovány proxy pro Javu, které umožňují přímo přistupovat ke všem třídám a jejím metodám v .NET knihovně. Ty je tak v Java aplikaci možné používat jako normální třídy a metody jazyka Java. Veškerý integrační kód je zkompilován do podoby Java a .NET knihoven. Zkompilované knihovny poté stačí spolu s Java a nativní knihovnou frameworku jni4net a knihovnami .NET, ke kterým bude přistupováno, umístit na class path Java aplikace. V Java kódu aplikace je nutné jednou zavolat statickou metodu inicializující jni4net. Poté je již možné vytvářet .NET třídy a volat jejich metody.

Podmínkou funkčnosti jni4net je provozování Java aplikace v operačním

systému Windows, instalace oficiálního běhového prostředí pro .NET od Microsoftu a instalace oficiální verze Java Development Kit od Oracle.

Integrace vykreslovacího jádra .NET klienta není napevno včleněna do UiGE. Bude se distribuovat jako volitelný modul. Modul stačí umístit na class path UIPServeru s UiGE a bude automaticky použit pro zjišťování vzhledu konkrétních ovládacích prvků. Momentálně se neřeší, zda klient, pro kterého se uživatelské rozhraní generuje, používá stejnou verzi vykreslovacího jádra, jako je to integrované. Nyní by při rozdílných verzích mohlo docházet k nekonzistencím. Vygenerované CUI by se mohlo zobrazovat na klientovi nesprávně.

Integrace vykreslovacího jádra tak pouze převede event, ve které se předávají instrukce pro vykreslování, z objektové reprezentace UIProtocolu, kterou využívá Java do objektové reprezentace, kterou využívá .NET klient a zavolá vykreslovací jádro .NET klienta. Jádro vrátí podobu vykresleného prvku opět v event. Ta je převedena do objektové reprezentace Java UIProtocolu a předána UiGE.

3.5 JBoss 7 a JCA

UIP to Java EE Connector (UTJEEC) a UIPServer pro Java EE byl testován spolu s testovací aplikací popsanou v následující kapitole nasazeny na aplikačním server JBoss AS 7.1. JBoss AS 7.1 je aplikační server pro platformu Java EE splňující specifikaci Java EE 6 Full Profile. Je tak na něj možné nasadit aplikaci zabalenou v archivu EAR, obsahující EJB a resource adapter.

Archiv EAR je soubor s aplikací určenou k nasazení na aplikační server. EAR obsahuje archivy (knihovny):

- Java ARchive (JAR), ve kterých se nacházejí zkompilované třídy aplikace aplikace a EJB, textové soubory, obrázky a jiné zdroje aplikace
- WAR, ve kterých se nacházejí soubory webové aplikace jako třídy servletů, stránky JavaServer Pages (JSP) a Facelets, mediální soubory a další
- Resource Adapter Archive (RAR), ve kterém se nacházejí zkompilované třídy a ostatní soubory resource adapteru využívající API JCA

JEEUSC využívá API JCA, proto je jedna z knihoven, z kterých se skládá, typu RAR. Na obrázku 3.2 je to **JEEUSC RAR**.

Na JBoss AS 7.1 není možné v základu nasadit aplikaci s RAR archivem. Kontejner, který se stará o spouštění resource adapteru není ve výchozí konfiguraci povolen. Ve složce `bin` aplikačního serveru v konfiguračním souboru `standalone`, je třeba upravit cestu ke konfiguračnímu souboru `jboss.server.default.config` z `standalone.xml` na `standalone-full.xml`. Druhý konfigurační soubor povoluje spuštění JCA kontejneru. Konfigurační soubor `standalone` ve složce `bin` má příponu `conf.bat` nebo `conf`, v závislosti na tom, zda je JBoss spouštěn na Windows nebo na Linuxu / Unixu.

Aby byl inicializován resource adapter JEEUSC obsažený v Java EE verzi UIPServeru a v UTJEEC, je třeba vytvořit konkrétní implementace abstraktních tříd, MDB, dědících z abstraktní třídy `AbstractSocketCommMessageDrivenBean`. Tyto MDB notifikuje resource adapter o přijetí nové zprávy od klienta. Implementace musejí být vytvořeny v archivu, ve kterém jsou umístěny EJB aplikace, která bude Java EE verzi UIPServeru nebo UTJEEC využívat.

Pro UTJEEC musí být vytvořena minimálně jedna implementace třídy `UipToJeeMessageDrivenBean` (viz. kapitola 2.5.1). Pro UIPServer musí být vytvořena minimálně jedna implementace `UipServerMessageDrivenBean` (viz. kapitola 2.6).

Vytvořené implementace musejí být opatřeny anotací s nastavením pro `SocketActivationSpec` resource adapteru. Zde se nastavuje číslo portu a adresa, na které se otevře síťové spojení a identifikátor resource adapteru (viz. kapitola 2.4).

JBoss 7.1 nedokáže automaticky provést spárování Resource adapter a Message Driven Bean popsané v kapitole 1.8.1. JBoss 7.1 potřebuje mít určený RAR archiv, ve kterém se má resource adapter hledat. RAR může být nakonfigurován v XML konfiguračním souboru JBossu. Tento soubor ale není součástí distribučního archivu EAR, ve kterém bude umístěn resource adapter. Již při výběru způsobu integrace UIProtocolu v kapitole 1.8.3 bylo rozhodnuto, že aplikaci s integrovaným UIProtocolem bude možné nasadit pomocí jednoho souboru. Proto je tento způsob nevhodný.

Druhým způsobem, jak spárovat RAR archiv obsahující Resource adapter s Message Driven Bean, je využít anotaci `@ResourceAdapter`. Tato anotace není součástí Java EE 6 API, je specifická pro JBoss. Proto musí být součástí EAR archivu aplikace knihovna JAR s touto anotací. Tato anotace a knihovna by neměly bránit nasazení aplikace na jiný Java EE aplikační server. Potřebná knihovna se nazývá `jboss-ejb3-ext-api`.

Anotace `@ResourceAdapter` očekává jako parametr název RAR archivu s resource adapterem včetně relativní cesty k němu. Jako výchozí složka se bere ta, ve které je umístěn EAR s celou aplikací. Pokud je RAR umístěn uvnitř EAR, musí být v relativní cestě uveden i název EAR archivu. Pro cestu uvnitř archivu se používá oddělovač `#`. Použití anotace tak bude vypadat například následovně:

```
@ResourceAdapter(value = "UipAfDemoAppUserRegSeparate.ear#" +
    "UipsJavaEeSocketConnector-rar-1.0.0.rar")
```

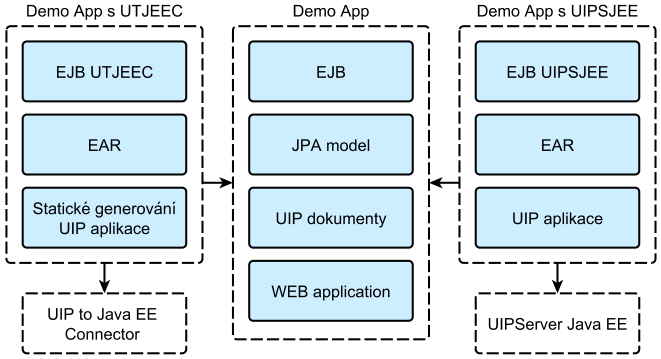
Příklad implementace EJB včetně všech potřebných anotací je k nahlédnutí ve výpisu kódu 3.1.

```
1  @MessageDriven(
2      name = "UipsToJavaConnector",
3      messageListenerInterface = uip.uipaf.
4          jeeuipssocketconnector.api.message.
5          SocketMessageEndpoint.class,
6      activationConfig = {
7         @ActivationConfigProperty(propertyName = "port",
8             propertyValue ="9876"),
9         @ActivationConfigProperty(propertyName = "connectorName",
10            propertyValue ="UipToJeeConnectorXml"),
11        @ActivationConfigProperty(propertyName = "ipAddress",
12            propertyValue ="localhost")
13     }
14 )
15 @ResourceAdapter(value = "UipAfDemoAppUserRegSeparate.ear#" +
16     "UipsJavaEeSocketConnector-rar-1.0.0.rar")
17 public class UipToJeeConnectorEndPointMessageDrivenBean
18     extends UipToJeeMessageDrivenBean {}
```

Výpis kódu 3.1: Příklad implementace EJB včetně konfiguračních anotací

3.6 Testovací aplikace

Funkčnost implementace byla ověřována s pomocí testovací aplikace. Testovací aplikace vychází z ukázkové aplikace AspectFaces [9]. Struktura testovací aplikace je zobrazena na obrázku 3.3. Testovací aplikace představuje hrubou kostru systému sloužícího jako katalog elektronických firemních formulářů. Umožňuje registraci uživatelů a vytváření záznamů o elektronických formulářích.



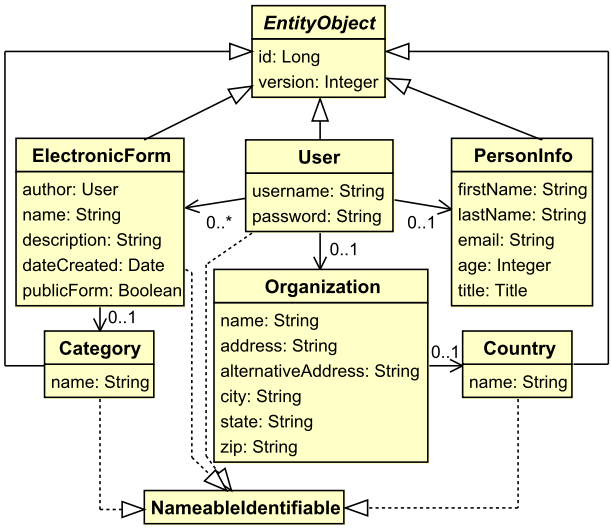
Obrázek 3.3: Struktura knihoven testovací aplikace

V testovací aplikaci je implementována sada EJB. Pro každou entitu datového modelu existuje EJB, která se stará o její načítání, ukládání a ostatní business funkcionalitu s ní související.

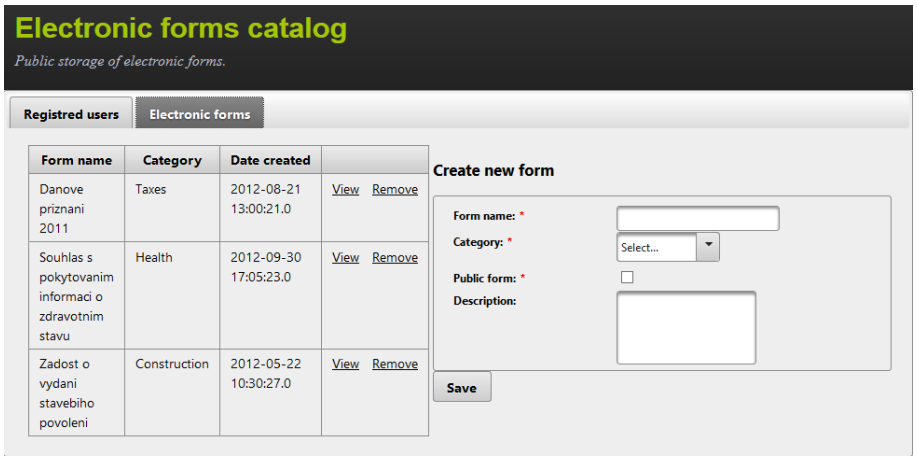
Uživatele představuje entita `User`. Každá entita uživatele je vazbou jedna ku jedné spojená s entitou s informacemi o uživateli `PersonInfo`. Organizaci, ve které uživatel pracuje, představuje třída `Organization`. `Organization` je z demonstračních důvodů embeded entitou vloženou do entity `User`. Entita `Country` představuje stát, ve kterém se nachází organizace uživatele. Entita `ElectronicForm` je elektronický formulář. Jeden uživatel může vytvořit více formulářů. Každý formulář patří do určité kategorie reprezentované entitou `Category`.

Pro provoz testovací aplikace není třeba vytvářet databázi a konfigurovat databázový stroj. Využívá se vestavěné Java implementace databázového stroje, který ukládá databázi do operační paměti. Pro testovací aplikaci to znamená, že po vypnutí aplikačního serveru s testovací aplikací jsou ztracena všechna data vytvořená v aplikaci.

K aplikaci je možné přistupovat pomocí webového rozhraní (ukázka



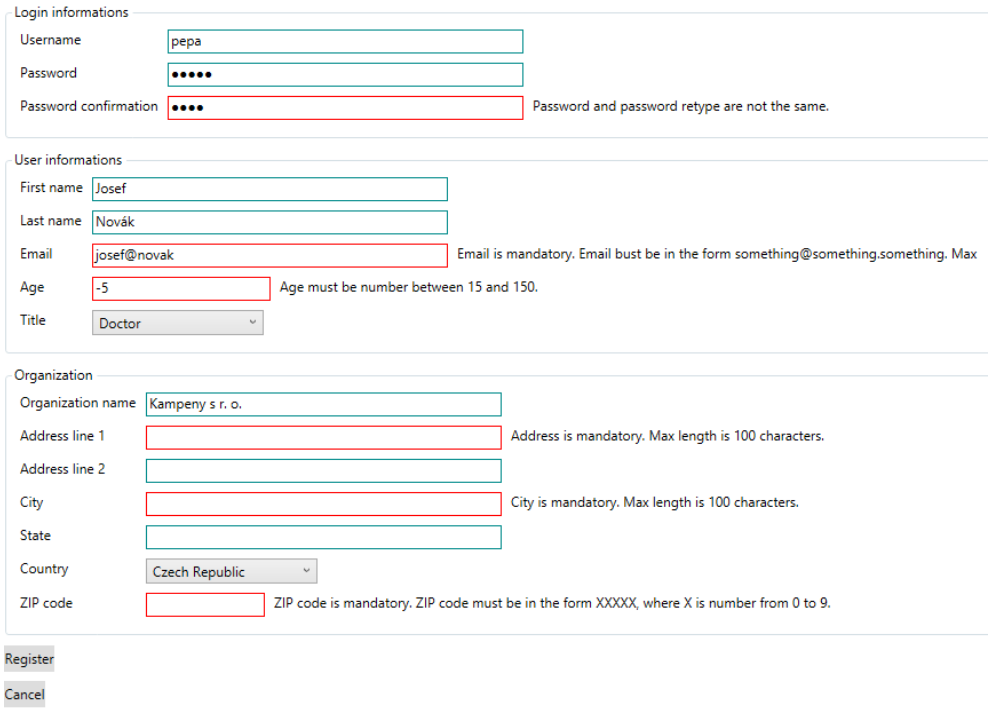
Obrázek 3.4: Datový model testovací aplikace



Obrázek 3.5: Ukázka webového rozhraní testovací aplikace

viz. obrázek 3.5) a pomocí rozhraní pro klienta UIProtocolu, UIProtocol aplikace. Součástí testovací aplikace je UIProtocol aplikace - uživatelské rozhraní aplikace pro UIProtocol klienta. V UIProtocol aplikaci je předpřipravené jediné AUI sloužící jako menu aplikace (obrázek 3.8). Ostatní uživatelská rozhraní jsou generována pomocí generátoru AUI a UiGE.

UIProtocol aplikace obsahuje sadu event handlerů v JavaScriptu, které



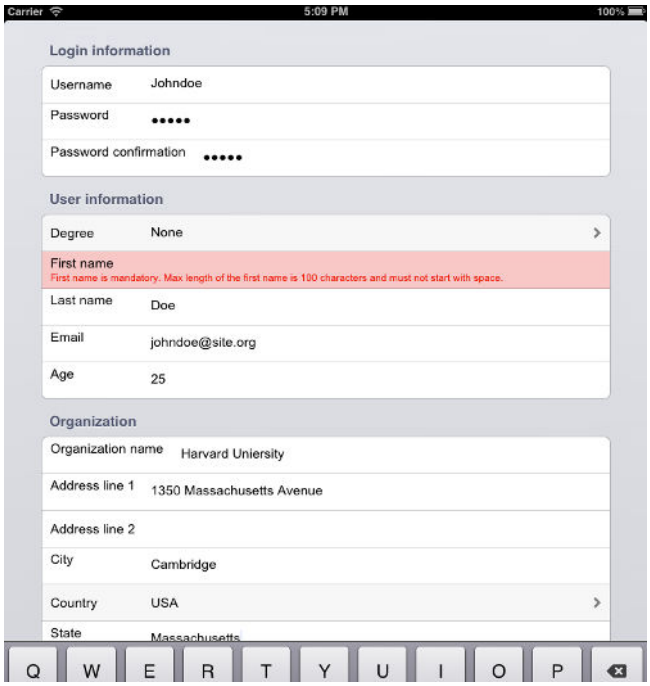
Obrázek 3.6: Ukázka formuláře testovací aplikace vygenerovaného pomocí generátoru AUI z JPA modelu a UiGE pro platformu Windows a .NET klienta

volají autonomní event handlery určené k integraci s Java EE aplikačním serverem (viz. kapitola 2.5.2). Jsou volány služby pro generování AUI, spuštění EJB metody a persistence JPA entity.

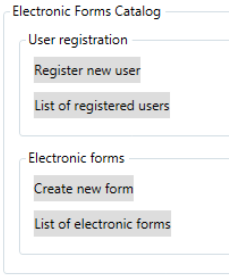
Všechny prozatím popsané části testovací aplikace jsou společné pro dvě její implementace.

První implementace obsahuje UTJEEC pro integraci s UIPServerem a UIProtocol aplikací. Součástí této implementace je samostatně spustitelná aplikace, která vygeneruje UIProtocol aplikaci, uživatelské rozhraní testovací aplikace v UIProtocolu, určenou pro nasazení na UIPServer s UiGE a autonomní event handlery určenými k integraci s Java EE aplikačním serverem (viz. kapitola 2.5.2). Tato aplikace demonstruje propojení samostatného UIPServeru s aplikací nasazenou na Java EE aplikačním serveru.

Druhá implementace obsahuje zároveň integrovaný UIPServer pro Java EE a UTJEEC. Tato aplikace demonstruje integraci UIPServeru do Java EE aplikačního serveru. Požadavky na integrační služby odeslané UI-



Obrázek 3.7: Ukázka formuláře testovací aplikace vygenerovaného pomocí generátoru AUI z JPA modelu a UiGE pro platformu iOS a zařízení iPad



Obrázek 3.8: Ukázka uživatelského rozhraní testovací aplikace vygenerovaného pomocí UiGE

Protocol částí aplikace jsou zpracovány přímo integrovaným UIPServerem, který má přímý přístup k Java části aplikace (viz. kapitola 2.5.2).

Z pohledu uživatele se obě implementace chovají zcela stejně a neliší se svou funkcionalitou. Data zadaná ve webovém rozhraní testovací aplikace je možné zobrazit v UIProtocol rozhraní a naopak.

Kapitola 4

Testování

Testování je nedílnou součástí vývoje každého software. Systém byl testován pomocí sady unit testů. Většina z těchto testů je spouštěna při sestavování jednotlivých knihoven systému pomocí nástroje Maven. Integrace mezi jednotlivými částmi systému probíhající přes síťové spojení a příjem a zasílání zpráv po síti bylo testováno pomocí k tomu implementované jednoúčelové aplikace a pomocí nástroje Apache Jmeter.

Apache Jmeter je nástroj určený pro testování síťových aplikací. Dokáže simulovat klienty připojující se k TCP nebo UDP portům a zasílat na tyto porty požadavky, například HTTP požadavky. Pomocí Jmeter je také možné analyzovat odpovědi zaslané testovaným serverem.

UIPServer pro Java SE by měl být funkční na jakémkoliv operačním systému s běhovým prostředím jazyka Java - JavaJRE nebo OpenJDK. Server byl testován na operačních systémech Mandriva Linux 2010.2 a Microsoft Windows 8. UiGE bez integrovaného vykreslovací jádra .NET klienta byl testován a je funkční na operačních systémech Mandriva Linux 2010.2 a Microsoft Windows 8. UiGE s integrovaným vykreslovacím jádrem .NET klienta je funkční pouze na operačním systému Windows (viz. kapitola 3.4).

UIPServer pro Java EE a UTJEEC byl testován nasazený spolu s testovací aplikací (kapitola 3.6) na Java EE aplikačním serveru JBoss AS 7.1. Testování probíhala na operačních systémech Mandriva Linux 2010.2 a Microsoft Windows 8.

Bylo provedeno profilování výkonu UiGE v UIPServeru pro Java SE a experimentální vyhodnocení výkonu a srovnání strategií přiřazení mapování pro UiGE a čtení a zápisu XML pomocí JAXB a SAX. Profiling a experimentální vyhodnocení výkonu probíhalo na operačním systému Windows

Procesor	Intel Core 2 Duo T7500 2,2 GHz, 4 MB L2 Cache
Čipová sada	Intel PM965 Express
Operační paměť	4 GB, DDR2, frekvence 667 MHz
Grafická karta	NVIDIA GeForce 8600M GS s 256 MB paměti
Pevný disk	Seagate ST9500420ASG 7200 otáček/min
Operační systém	Windows 8 64bit
Java	Oracle JDK 7 update 17 32bit

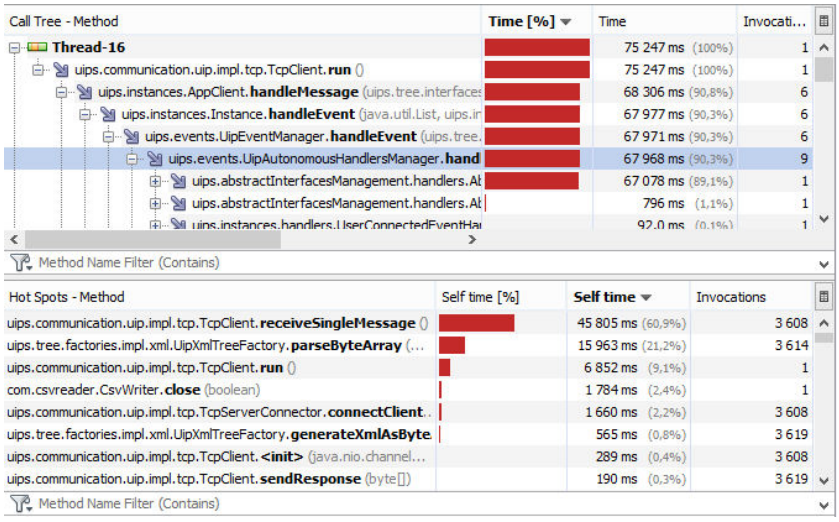
Tabulka 4.1: Konfigurace testovacího počítače

na sestavě popsané v tabulce 4.1.

Úkolem diplomové práce nebylo provádět uživatelskou studii hodnotící kvalitu a použitelnost CUI vygenerovaných pomocí UiGE. Tato studie byla provedena v [21].

4.1 Profiling

Profilování výkonu bylo provedeno pomocí profilovacího nástroje integrovaného do vývojového prostředí NetBeans. Ukázka výsledků profilování je na obrázku 4.1.



Obrázek 4.1: Vývojové prostředí NetBeans s otevřenými výsledky profilování

Byl profilován průběh vyřízení požadavku klienta a vygenerování uživatelského rozhraní na obrázku 3.6 pomocí UiGE se strategií mapování pro exaktní algoritmus na UIPServeru pro Javu SE.

Z výsledků profilování je patrných několik částí systému, které by bylo velice vhodné optimalizovat pro zvýšení výkonu generování CUI.

Generování nejvíce zdržuje dotazování klienta UIProtocolu na vzhled konkrétních ovládacích prvků. Z tohoto důvodu byla navrhnuta a implementovaná integrace vykreslovacího jádra klienta do UiGE, viz. kapitoly 2.2 a 3.4. Při pěti opakovaných měřeních trval průměrně běh optimalizačního algoritmu při generování CUI devět sekund. Díky využití integrovaného vykreslovacího jádra se doba běhu optimalizačního algoritmu zkrátila na 200 ms.

Nízký výkon generátor při dotazování klienta na ovládací prvky přes síťové připojení je způsobený zejména nutností parsovat a generovat zprávy, které se posílají po síti. Parsování a generování probíhalo pomocí JAXB. To se však ukázalo jako málo efektivní. Proto bylo nahrazeno parsováním pomocí SAX a generováním pomocí `XMLStreamWriter`. Výsledky měření srovnávající výkon JAXB a SAX, včetně výkonu UiGE při použití obou, jsou uvedeny v kapitole 4.3.

Poslední částí, ve které profilování prokázalo možné zpomalení běhu UiGE, je vlastní optimalizační algoritmus UiGE. Proto byly navrženy a implementovány heuristické optimalizační algoritmy, které by toto měly řešit (viz. kapitola 2.2.2). Experimentální srovnání těchto algoritmů viz. kapitola 4.2.

4.2 Experimentální měření výkonu UiGE

Byly srovnávány jednotlivé strategie přiřazování mapování z kapitoly 2.2.2 z hlediska výkonu, tj. doby, za kterou proběhne optimalizace výběru mapování, a z hlediska kvality vygenerovaného uživatelského rozhraní. Se strategií mapování číslo 1 je optimalizační algoritmus exaktním algoritmem. Pokud najde řešení, je toto řešení nejlepší možné. Pokud řešení nenajde, potom řešení neexistuje. S ostatními strategiemi je optimalizační algoritmus heuristikou.

Kvalita CUI, představovaná celkovou cenou přiřazených mapování, vygenerovaných heuristickými algoritmy byla srovnávána s cenou CUI vygenerovaného pomocí exaktního algoritmu. Byla vypočítána relativní chyba heuristické metody pro každé měření. Ta udává, jak se výsledné CUI liší

kvalitou, cenou, od ideálního CUI vygenerovaného pomocí exaktního optimalizačního algoritmu. Čím menší relativní chyba, tím více se vygenerované CUI blíží kvalitou k ideálnímu. Nula znamená, že je vygenerované CUI optimální.

Relativní chyba se spočítá jako $\varepsilon = (C - C_{opt})/C_{opt}$, kde C je cena řešení spočítaného heuristikou a C_{opt} je cena optimálního řešení pro stejnou velikost problému.

4.2.1 Nastavení experimentu

Pomocí k tomu určenému nástroje byla vygenerována sada testovacích AUI v XML syntaxi UIProtocolu. Z těchto AUI nástroj vytvořil aplikaci ve formátu [3]. Hlavní rozhraní aplikace, které se zobrazí po připojení klienta UIProtocolu k aplikaci, obsahuje tlačítko, které spustí test. Pro účely testování byla vytvořena upravená implementace třídy Optimizer z UiGE. Tato třída vygeneruje každý AUI s pomocí všech strategií přiřazení mapování. Pro zpřesnění výsledků je generování s každou strategií spuštěno vícekrát. Měří se čas, po který běží optimalizační algoritmus, včetně výpočtu validních mapování pro abstraktní elementy. Při optimalizaci se využívá integrovaného vykreslovacího jádra .NET UIProtocol klienta. Je tak odfiltrován vliv výkonu parsování a generování XML a síťového subsystému na měření. Výsledný čas pro strategii je zprůměrován počtem opakování měření.

Čas měření se zjišťoval jako rozdíl časů na začátku a konci měření udaných metodou `getCurrentThreadCpuTime` třídy `ThreadMXBean`.

Bylo vygenerováno šest testovacích sad AUI. V každé sadě bylo vygenerováno několik AUI se zvětšujícím se počtem elementů nebo kontejnerů. Jedna sada obsahovala pouze abstraktní kontejnery. Další sady obsahují pouze elementy. V sadách se stupňuje počet druhů abstraktních elementů, které se jistě namapují na různé konkrétní ovládací prvky, od jednoho až k pěti různým elementům.

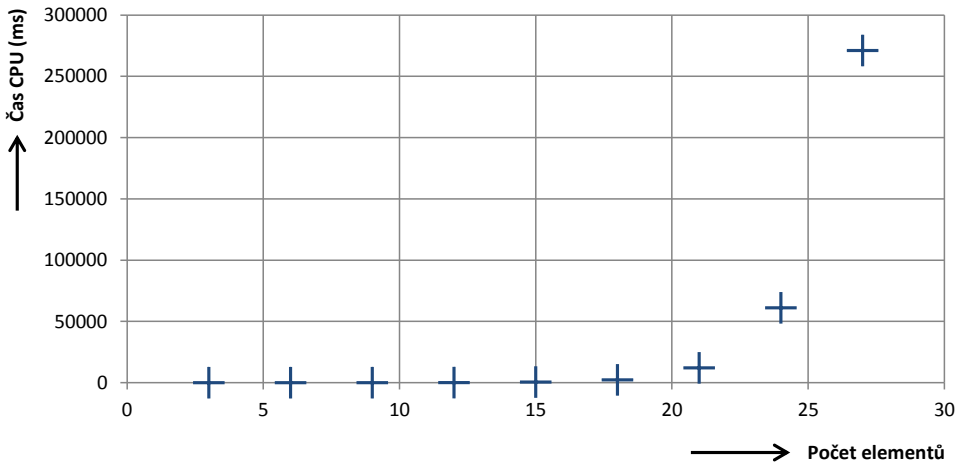
4.2.2 Výsledky měření

V kapitole jsou prezentovány výsledky z testovací sady s kontejnery a třemi různými elementy. Tyto výsledky ilustrují celé měření. Výsledky měření s ostatními sadami se od těchto v principu neliší. Kompletní výsledky testování včetně všech sad testovacích AUI je možné nalézt na DVD přiloženém k diplomové práci. Počet kombinací v tabulkách, které musí exaktní algoritmus v nejhorším případě prověřit, se spočítá jako součin počtu legálních mapování elementů a kontejnerů přes všechny elementy a kontejnery AUI.

Každý kontejner a element může mít různá validní mapování. Čas výpočtu je tedy závislá nejen na počtu elementů AUI, ale i na počtu jim validních mapování.

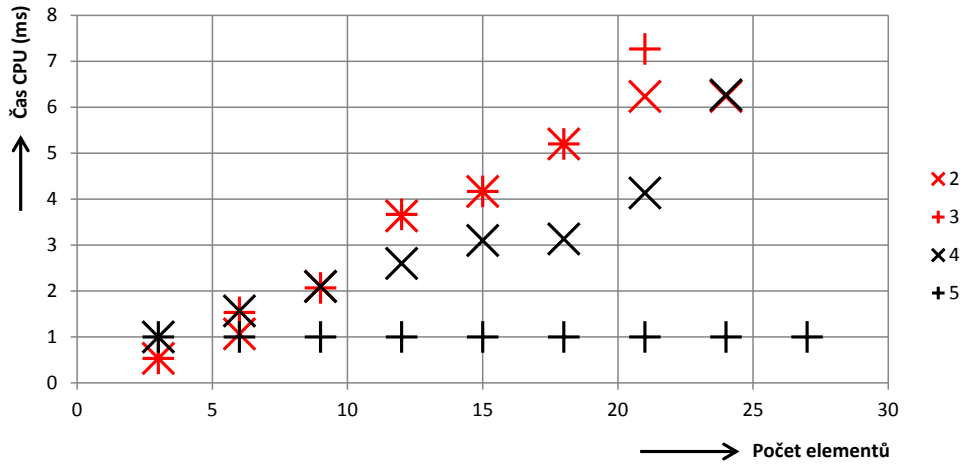
# elementů	Číslo strategie # kombinací	1	2	3	4	5
		t (ms)				
3	12	1,03	0,53	0,53	15	1
6	48	3,13	1,07	1,53	1,57	1
9	192	15	2,1	2,07	2,1	1
12	768	78	3,63	3,67	2,6	1
15	3072	530	4,17	4,17	3,1	1
18	12288	2309	5,2	5,2	3,13	1
21	49152	12106	6,23	16	4,13	1
24	196608	61106	6,23	15	6,27	1
27	786432	271067	16	7,27	16	1

Tabulka 4.2: Doba generování CUI pro různé strategie mapování v závislosti na počtu elementů AUI pro tři různé elementy



Obrázek 4.2: Závislost doby generování CUI pomocí exaktního algoritmu (strategie 1) na počtu elementů AUI pro tři různé elementy

V grafu 4.2 se jasně potvrzuje exponenciální závislost doby generování na počtu elementů AUI pro exaktní algoritmus, jak byla popsána v kapitole 1.6.2.



Obrázek 4.3: Závislost doby generování CUI pomocí heuristiky (strategie 2 - 5) na počtu elementů AUI pro tři různé elementy

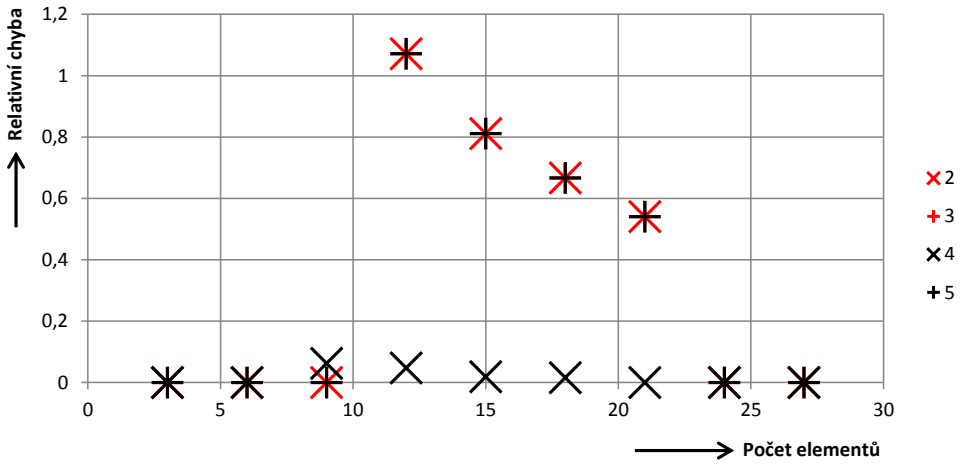
Nejméně výpočetně náročný byl podle grafu 4.3 optimalizační algoritmus se strategií 5. Doba výpočtu byla kolem 1 ms, což je rozlišovací schopnost metody použité pro měření času výpočtu. Strategie 2 a 3 byly podobně náročné. Mírně méně časově náročná bylo strategie 4. I když to není z grafu pro strategii 5 patrné, optimalizační algoritmus se všemi strategiemi vykazoval z výsledků měření lineární závislost doby výpočtu na počtu elementů AUI. Měření probíhalo i pro složitější rozhraní, kde bylo lineární prodloužení doby výpočtu znatelné.

Nejkvalitnější CUI s nejmenší relativní chybou generoval podle grafu 4.4 optimalizační algoritmus se strategií 4. Chyba nebyla ani u jedné sady nikdy větší než jedna desetina. Vygenerované rozhraní s touto strategií bylo i vizuálně často nerozpoznatelné od rozhraní vygenerovaného exaktním algoritmem. Ostatní strategie vykazovaly zcela stejnou relativní chybu pro všechna měření.

Všechny heuristiky vždy do určitého počtu elementů generovaly ideální řešení. V této oblasti nebylo nutné optimalizovat. Všechny elementy mohly být namapovány svými nejlepšími mapováními. Od určitého prahu, kdy nebylo možné přiřadit elementům lokálně optimální mapování, se kvalita vygenerovaného řešení skokově zhoršila a dále se pozvolna snižovala. Nejhorší výsledky podává heuristika, když je největší poměr počtu jedinečných abstraktních elementů, které se namapují každý na jiný konkrétní ovládací

# elementů	Číslo strategie # kombinací	2	3	4	5
		Relativní chyba			
3	12	0	0	0	0
6	48	0	0	0	0
9	192	0	0	0,0625	0
12	768	1,071	1,07	0,0476	1,07
15	3072	0,811	0,811	0,0189	0,811
18	12288	0,667	0,667	0,0159	0,667
21	49152	0,541	0,541	0	0,541
24	196608	0	0	0	0
27	786432	0	0	0	0

Tabulka 4.3: Relativní chyba vygenerovaného CUI pro různé heuristické strategie mapování v závislosti na počtu elementů AUI pro tři různé elementy



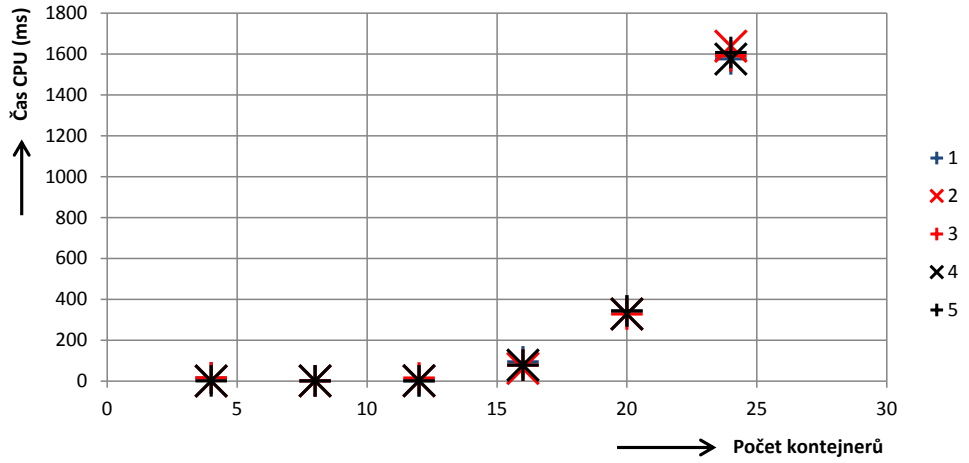
Obrázek 4.4: Závislost relativní chyby heuristiky pro generování CUI (strategie 2 - 5) na počtu elementů AUI pro tři různé elementy

prvek, ku počtu všech elementů v AUI. Se snižujícím se poměrem klesá složitost problému a heuristika tak dokáže vygenerovat kvalitnější uživatelské rozhraní.

Graf 4.5 ilustruje tvrzení z kapitoly 1.6.2, a to, že mapování kontejnerů se vždy optimalizuje exaktním algoritmem s exponenciální složitostí.

# kontejnerů	Číslo strategie # kombinací	1	2	3 t (ms)	4	5
4	24	15	0,03	16	0,53	1
8	96	1,03	0,5	1,07	1,03	1
12	384	1,57	1,03	15	1,57	1
16	1536	94	62	78	78	78
20	6144	343	328	328	328	343
24	24576	1576	1638	1591	1575	1607

Tabulka 4.4: Doba generování CUI pro různé strategie mapování v závislosti na počtu kontejnerů AUI



Obrázek 4.5: Závislost doby generování CUI pro různé strategie mapování na počtu kontejnerů AUI

4.2.3 Vyhodnocení

Nejlepší výsledky podával heuristický optimalizační algoritmus se strategií přiřazení mapování číslo čtyři. Tato strategie přiřazuje mapování nenamapovaného elementu všem elementům, kterým je možné toto mapování přiřadit, a to i těm, které jsou již namapované. Implementačně je tato strategie ze všech heuristických strategií nejjednodušší. I když nebyl se strategií optimalizační algoritmus nejrychlejší, podával nejlepší výsledky v poměru cena - výkon. Při vizuální kontrole vygenerovaného CUI bylo toto často neodlišitelné od CUI vygenerovaného exaktním algoritmem, i když byla rela-

tivní chyba řešení nenulová. Tato heuristika byla proto zvolena jako výchozí pro UiGE.

4.3 Experimentální měření výkonu parsování a generování XML

V kapitole je experimentálně vyhodnocena rychlost parsování XML při využití JAXB a SAX a rychlost generování XML při využití JAXB a `XMLStreamWriter`.

4.3.1 Nastavení experimentu

Pro měření byly využity testovací AUI v XML syntaxi UIProtocolu z předchozí kapitoly. XML jsou vygenerované tak, že velikost souboru je přímo úměrná počtu elementů v XML. Pro měření byla naimplementována zvláštní aplikace. Ta postupně načítá obsah XML souborů do textového řetězce. Parsuje se text v textovém řetězci. Měření tak není ovlivněné rychlostí načítání souboru z disku. Každý soubor se parsuje pomocí JAXB a SAX. Z XML je generována objektová reprezentace UIProtocolu. Pro zpřesnění výsledků je každé XML parsováno tisíckrát. Výsledný čas parsování jednoho souboru je tímto počtem zprůměrován.

Čas měření se zjišťoval jako rozdíl časů na začátku a konci měření udaných metodou `getCurrentThreadCpuTime` třídy `ThreadMXBean`.

Z objektové reprezentace bylo vzápětí vygenerováno XML pomocí JAXB a `XMLStreamWriter`. Měření bylo opět opakováno tisíckrát. XML bylo vygenerováno jako textový řetěz a nezapisovalo se na disk.

4.3.2 Výsledky měření

Jak je zřejmé z grafu 4.6, parsování XML pomocí SAX bylo několikrát rychlejší něž pomocí JAXB. Se zvětšující se velikostí souboru navíc doba parsování pomocí SAX rostla pomaleji něž s JAXB.

Generování XML pomocí `XMLStreamWriter` již o tolik rychlejší než JAXB nebylo. Přesto se doba generování zkrátila. Navíc se zvyšujícím se počtem prvků v objektové reprezentaci, tudíž se zvětšujícím se počtem elementů, rostla doba potřebná pro vygenerování XML s `XMLStreamWriter` pomaleji než s JAXB.

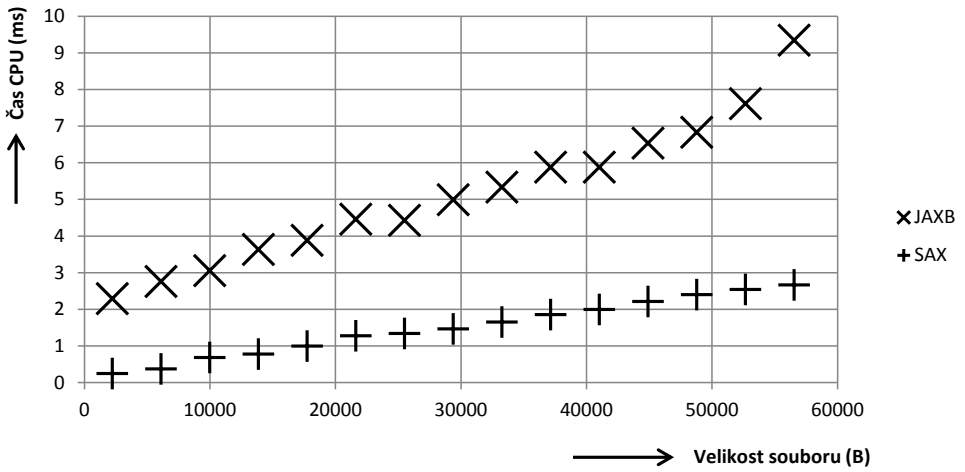
	Čtení		Zápis	
	JAXB	SAX	JAXB	XSW
Velikost souboru (B)	Čas CPU (ms)			
2217	2,29	0,249	0,078	0,078
6097	2,76	0,375	0,265	0,202
9977	3,06	0,686	0,437	0,344
13857	3,64	0,780	0,577	0,406
17737	3,88	0,998	0,765	0,593
21617	4,46	1,28	0,796	0,671
25497	4,43	1,34	1,01	0,749
29377	4,99	1,47	1,19	0,998
33257	5,34	1,65	1,45	1,03
37137	5,88	1,86	1,55	1,06
41017	5,88	2,00	1,79	1,28
44897	6,54	2,22	1,87	1,44
48777	6,83	2,40	2,03	1,51
52657	7,61	2,54	2,57	1,59
56537	9,35	2,67	2,53	1,76

Tabulka 4.5: Srovnání doby čtení a zápisu XML souboru pomocí JAXB a SAX v závislosti na velikosti XML souboru

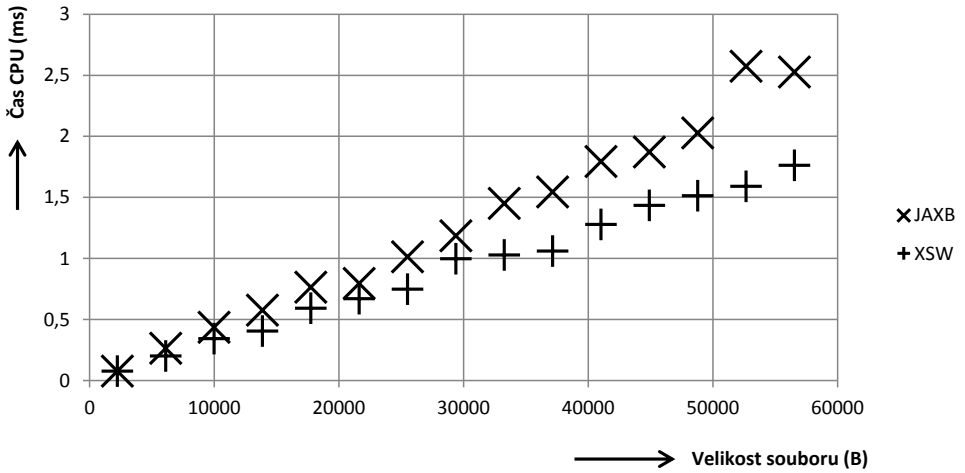
Pomocí stejné metody, jako byla popsána v kapitole 4.2, byla měřena doba generování jednoho CUI pomocí UiGE. Nyní UiGE nepoužívalo integrované vykreslovací jádro klienta, ale komunikovalo s klientem po síti. Byla změřena doba vygenerování CUI při použití JAXB a při použití SAX. SAX dokázalo zkrátit dobu generování jednoho CUI v průměru téměř pětkrát.

4.3.3 Vyhodnocení

Měření prokázalo větší výkon SAX při parsování XML. SAX dokázalo také významně zkrátit dobu generování CUI v UiGE. Z těchto důvodů je parsování XML v UIPServeru a ostatních souvisejících systémech implementováno pomocí SAX.



Obrázek 4.6: Závislost doby čtení XML souboru pomocí JAXB a SAX na velikosti XML souboru



Obrázek 4.7: Závislost doby zápisu XML souboru pomocí JAXB a XMLStreamWriter na velikosti XML souboru

Závěr

V úvodu práce byly stanoveny cíle navrhnout a implementovat generátor AUI využívající nástroje AspectFaces, navrhnout a implementovat .NET verzi UiGE v jazyce Java a jeho vylepšení. Cílem bylo, aby se generátor stal modulem, který je možné začlenit do existujícího Java UIProtocol aplikačního serveru nazvaného UIPServer. Bylo tak třeba upravit architekturu UIPServeru, aby jej bylo možné rozšiřovat pomocí modulů. Dalším cílem bylo navrhnout a implementovat propojení mezi UIProtocol aplikací běžící na UIProtocol serveru a JPA aplikací nasazenou na Java EE aplikačním serveru. Posledním z cílů práce bylo vytvořit ukázkovou aplikaci, která by demonstrovala možnosti generování uživatelských rozhraní a propojení UIProtocol a JPA aplikace. Naplnění těchto cílů vychází z provedené analýzy generátoru UiGE, který je součástí .NET verze platformy UIProtocol, nástroje AspectFaces, který umožňuje generování uživatelských rozhraní na základě inspekce kódu a dalších technologií. Vytýčené cíle byly splněny následovně.

Byla pozměněna architektura UIProtocol serveru UIPServeru. Funkcionalitu UIPServeru je možné nově rozšířit pomocí modulů. Rozšiřující moduly je možné do UIPServeru přidávat bez nutnosti rekompilace kódu. Podle nastavení UIPServeru a pomocí reflexe jazyka Java jsou moduly načteny při startu UIPServeru. UIPServer je tak nyní možné snadno rozšířit o podporu nových variant UIProtocolu, jako je binární varianta. Nejdůležitější změnou je přepracování subsystému UIPServeru, který zachycuje události představující interakci uživatele s uživatelským prostředím a žádosti klienta pro UIProtocol, tzv. eventy. Event může být zpracována pomocí programu, event handleru, běžícího v bezpečném omezeném prostředí. Dosud bylo možné spouštět pouze event handlery, které jsou součástí UIProtocol aplikace na-

sazené na UIPServeru. Nová architektura umožňuje vytvářet a jako moduly přidávat do UIPServeru tzv. autonomní event handlery. Ty jsou společné pro všechny UIProtocol aplikace. Tyto event handlery mají neomezený přístup k API UIPServeru. Dovolují neomezeným způsobem rozšiřovat proces zpracování událostí (event). Tak je možné nyní odchytit eventu znamenající žádost o poskytnutí uživatelského rozhraní a předat ji Java verzi UiGE, který je v podobě modulu integrován do UIPServeru.

Byla navržena a implementována Java verze generátoru uživatelských rozhraní UiGE. Byla navržena a implementována skupina heuristických optimalizačních algoritmů s cílem zrychlit optimalizaci uživatelského rozhraní během jeho generování. Proces optimalizace uživatelského rozhraní je NPC optimalizační problém s exponenciální složitostí. Bylo provedeno experimentální hodnocení navržených algoritmů. Byla hodnocena kvalita vygenerovaných uživatelských rozhraní - relativní chyba ceny uživatelského rozhraní proti rozhraní vygenerovanému exaktním algoritmem a doba generování. Z algoritmů byl vybrán jeden, který nabízí nejlepší poměr cena - doba generování. Tento vybraný algoritmus ve většině případů generoval téměř stejně kvalitní rozhraní jako exaktní algoritmus, ale za řádově kratší dobu.

Na základě provedeného profilování výkonu byla navržena a implementována úprava generování a parsování XML dokumentů UIProtocolu. Místo technologie JAXB byla zvolena v případě implementace této práce několikanásobně výkonnější technologie SAX. Též byl vylepšen UiGE na základě profilování. UiGE potřebuje při generování rozhraní znát podobu komponent uživatelského rozhraní, jak se zobrazují v klientovi UIProtocolu. Proto s klientem, který žádá o vygenerování rozhraní, komunikuje po síti. Toto je však zdlouhavé. Proto je nyní možné volitelně integrovat do UiGE modul s vykreslovacím jádrem klienta. Odpadá tak nutnost komunikace po síti. Generování rozhraní se díky tomu citelně zrychlilo.

Byl navržen a implementován generátor abstraktních uživatelských rozhraní UIProtocolu pro UiGE. Generátor využívá nástroje AspectFaces k inspekci JPA datového modelu aplikace. Na základě informací zjištěných při inspekci je nástrojem AspectFaces sestaveno abstraktní uživatelské rozhraní. Generátor abstraktních rozhraní má podobu nezávislého modulu, který je možné využít jak ke statickému vygenerováni několika rozhraní, tak k dynamickému generování rozhraní za běhu aplikace.

Byla navržena a implementována integrace aplikačního protokolu UIProtocolu do Java EE platformy pomocí technologie Java Connector Architecture. S pomocí této integrace vzniklo propojení mezi UIProtocol aplikací

nasazenou na UIPServeru a Java EE aplikací nasazenou na Java EE aplikačním serveru. Pro UIPServer byl vytvořen modul umožňující komunikaci s Java EE. Propojení aplikací je možné realizovat nejen na úrovni datové, ale i na úrovni business logiky. Z UIProtocol aplikace je možné volat metody Enteprise Java Bean, žádat o vygenerování abstraktního uživatelského rozhraní, odesílat na Java EE server data, která se zde uloží do databáze, a data z databáze načítat. UIProtocol platforma tak získává podporu chybějící komplexní persistence dat. Díky propojení je možné vytvořit aplikaci, která implementuje veškerou business logiku v Javě a nabízí více uživatelských rozhraní pro různé platformy. Například jedno webové a jedno pro UIProtocol klienty.

Díky modulární architektuře UIPServeru mohla být s minimálním úsilím vytvořena verze UIPServeru nasaditelná na Java EE aplikační server. UIPServer tak může těžit z robustního běhového prostředí aplikačního serveru a využívat všech jeho výhod. UIPServer může být součástí jiné Java EE aplikace.

Byla vytvořena ukázková aplikace demonstrující možnosti generování abstraktních uživatelských rozhraní z datového modelu aplikace, generování konkrétních uživatelských rozhraní pro UIProtocol, možnosti propojení UIProtocol a Java EE JPA aplikace a v neposlední řadě i možnost integrace UIPServeru do Java EE aplikace. Aplikace demonstruje propojení principů generování uživatelských rozhraní na základě inspekce datového modelu a generování uživatelského rozhraní na míru uživateli. Části řešení byly testovány pomocí unit testů.

Tímto byly úspěšně splněny všechny požadavky ze zadání diplomové práce a cíle stanovené v úvodu práce.

Touto diplomovou prací vývoj představeného řešení nekončí.

Budoucí vývoj

Je plánováno nadále vyvíjet představené řešení. Hlavní úkoly do budoucna jsou:

- Vytvořit podrobnou vývojářskou dokumentaci pro použití generátoru abstraktních uživatelských rozhraní.
- Rozšiřovat možnosti generátoru abstraktních rozhraní, zejména schopnosti mapovat nové datové typy na ovládací prvky.

- AspectFaces neumožňuje mapování abstraktních typů, s čímž bylo při návrhu počítáno. Pro zprovoznění testovací ukázkové aplikace tak bylo třeba v knihovně generátoru AUI přímo nakonfigurovat, jak se mají mapovat některé datové typy, entity, z datového modelu ukázkové aplikace. Jakmile bude požadovaná funkcionalita do AspectFaces přidána, je třeba odstranit popsanou konfiguraci z generátoru AUI.
- Dále rozšiřovat možnosti propojení Java EE a UIProtocol aplikace.
- Navrhnout a implementovat jednotné řešení popisu mapování abstraktních ovládacích prvků na konkrétní v UiGE tak, aby nevznikaly nekonsistence mezi Java a .NET implementací UiGE.
- Vytvořit novou verzi dokumentace UIProtocol aplikace [3] podle změn a rozšíření navržených v diplomové práci.
- Navrhnout a implementovat rozšíření propojení Java EE a UIProtocol aplikace umožňující notifikaci UIProtocol aplikace o změně entity v databázi.
- Řešením spojující AspectFaces a UiGE neumožňuje generovat rozhraní na základě popisu průběhu určité činnosti. Generování uživatelských rozhraní na základě popisu činnosti (workflow) je tak zajímavou oblastí výzkumu a přístupem, který by významně rozšířil možnosti stávající platformy.

Předchozí výčet není úplný. Budoucí vývoj navrženého systému bude pružně reagovat na výsledky výzkumu v oblasti automatického generování uživatelských rozhraní a na vývoj nástrojů AspectFaces a UiGE.

Literatura

[1] Aspect Research Associates: Aspect programming. 2013. Dostupné z: http://www.aspectprogramming.com/home/aosd

[2] Bašek, J.: *Webové řešení pro podporu vývoje v UIProtocolu*. ČVUT, 2011, bakalářská práce.

[3] Bašek, J.: *UIProtocol Application Specification*. ČVUT, 2012, draft 3.

[4] Bašek, J.: Web solution for UIProtocol dev. support. 2013. Dostupné z: http://sourceforge.net/projects/uipwds

[5] Benli, O. S.: THE BRANCH-AND-BOUND APPROACH. 2013. Dostupné z: http://www.csulb.edu/~obenli/Research/IE-encyc/bb.html

[6] Cerny, T.; Chalupa, V.; Rychtecky, L.; aj.: Machine-driven Code Inspection to Reduce Restated Information. *Lecture Notes in Information Technology (LNIT). Newark, Delaware: Information Engineering Research Institute*, ročník 1, 2012: s. 213–218, ISBN 978-1-61275-009-5.

[7] Chetty, D.: *Tomcat 6 Developer's Guide*. Packt publishing, 2009, ISBN 978-1-847197-28-3.

[8] Coding Crayons team: AspectFaces. 2013. Dostupné z: http://www.aspectfaces.com/

[9] Coding Crayons team: AspectFaces Wiki. 2013. Dostupné z: http://wiki.codingcrayons.com/display/af/AspectFaces

[10] Dictionary.com team: Dictionary.com. 2013. Dostupné z: http://dictionary.reference.com

[11] Erich, G.; Richard, H.; Ralph, J.; aj.: Design patterns: elements of reusable object-oriented software. *Reading: Addison Wesley Publishing Company*, 1995.

[12] Černý, T.; Song, E.: Model-driven Rich Form Generation. *INFORMATION-AN INTERNATIONAL INTERDISCIPLINARY JOURNAL*, ročník 15, č. 7, 2012: s. 2695–2714, ISSN 1343-4500.

[13] Farrell, W.: Introduction to the J2EE Connector Architecture. 2013. Dostupné z: http://www.ibm.com/developerworks/java/tutorials/j-jca/section2.html

[14] Ferentschik, H.; Morling, G.: Hibernate Validator Reference Guide. 2013. Dostupné z: http://docs.jboss.org/hibernate/validator/4.2/reference/en-US/html_single/

[15] Gerhat, P.: *UIProtocol Client for Apple iPhone*. ČVUT, 2010, bakalářská práce.

[16] Greenfield, J.: The Rox Java NIO Tutorial. 2013. Dostupné z: http://rox-xmlrpc.sourceforge.net/niotut/

[17] Hibernate contributors: Hibernate. 2013. Dostupné z: http://www.hibernate.org/

[18] Hookom, J.: Facelets - JavaServer Faces View Definition Framework. 2013. Dostupné z: https://facelets.java.net/nonav/docs/dev/docbook.html

[19] Java comunity contributors: JavaServer Pages 2.1 Expression Language Specification. 2013. Dostupné z: http://download.oracle.com/otndocs/jcp/jsp-2.1-fr-eval-spec-oth-JSpec/

[20] Macik, M.: Context model for ability-based automatic UI generation. In *Cognitive Infocommunications (CogInfoCom), 2012 IEEE 3rd International Conference on*, IEEE, 2012, s. 727–732.

[21] Macik, M.; Cerny, T.; Basek, J.; aj.: Platform-aware rich-form generation for adaptive systems through code-inspection. In *SouthCHI 2013 - Int. Conference on Human Factors in Cumputing & Informatics*, SouthCHI, 2013.

[22] Macík, M.: *User Inteface Generator*. CTU in Prague, 2009, diploma thesis.

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

[24] Macík, M.: *UIPA - Abstract User Interface Description using UIP, draft 3*. CTU in Prague, 2012, format specification.

[25] Macík, M.; Slováček, V.: *UIProtocol Extension Kit, draft 5*. ČVUT.

[26] Object Management Group: Introduction to OMG's Unified Modeling Language. 2005. Dostupné z: http://www.omg.org/gettingstarted/what_is_uml.htm

[27] Oracle Corporation team: Class ResourceBundle. 2013. Dostupné z: http://docs.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html

[28] Oracle Corporation team: Java Class Reference. 2013. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getFields()

[29] Oracle Corporation team: Java EE Compatibility. 2013. Dostupné z: http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

[30] Oracle Corporation team: Java Lesson: Annotations. 2013. Dostupné z: http://docs.oracle.com/javase/tutorial/java/annotations/

[31] Oracle Corporation team: Java Persistence API. 2013. Dostupné z: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[32] Oracle Corporation team: Java Server Faces. 2013. Dostupné z: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[33] Oracle Corporation team: Javax Validation Reference. 2013. Dostupné z: http://docs.oracle.com/javaee/6/api/javax/validation/constraints/package-summary.html

[34] Oracle Corporation team: *The Java EE 6 Tutorial*. Oracle Corporation, 2013, Part No: 821–1841–12.

[35] Poziombka, W.: Serve It Up with J2EE 1.4 - Use JCA 1.5 and EJB 2.1 to Extend your App Server. 2013. Dostupné z: http://www.theserverside.com/news/1364656/Serve-It-Up-with-J2EE-14-Use-JCA-15-and-EJB-21-to-Extend-your-App-Server

[36] Savara, P.: jni4net. 2013. Dostupné z: http://jni4net.sourceforge.net/

[37] Sedláček, V.: *Tenký klient podporující UIProtocol na platformě PHP*. ČVUT, 2008, bakalářská práce.

[38] Slováček, V.: *UIProtocol specification draft 8*. ČVUT, 2009, specifikace protokolu.

[39] Sobel, J. M.; Friedma, D. P.: An Introduction to Reflection-Oriented Programming. 2013. Dostupné z: http://www.cs.indiana.edu/~jsobel/rop.html

[40] Stackoverflow contributors: URL to load resources from the classpath in Java. 2013. Dostupné z: http://stackoverflow.com/questions/861500/url-to-load-resources-from-the-classpath-in-java

[41] Staveley, A.: JAXB, SAX, DOM Performance. 2013. Dostupné z: http://dublintech.blogspot.co.uk/2011/12/jaxb-sax-dom-performance.html

[42] Vavroušek, D.: *Implementace klienta UIProtokolu ve Flash*. ČVUT, 2009, bakalářská práce.

Příloha A

Slovník

Glossary

anotace Forma metadat, poskytujících informace o programu, která není součástí samotného programu. Anotace nemá žádný přímý vliv na kód, který anotuje [30].

builder je objektový návrhový vzor. Účelem tohoto návrhového vzoru je abstrahovat proces konstrukce složitého objektu nebo skupiny objektů, tak aby bylo možné konstruovat různé implementace těchto objektů [11].

EJB Enterprise Java Beans jsou komponenty aplikace nasazené na serveru platformy Java EE umožňující tvorbu modulárních aplikací [34].

Facelets Facelets je template framework určený k vytváření webových stránek na platformě Java EE, je součástí frameworku JSF [18].

FTP File Transfer Protocol je protokol pro přenos souborů mezi počítači pomocí počítačové sítě [10].

HTTP Hypertext Transfer Protocol je internetový protokol určený pro výměnu hypertextových dokumentů ve formátu HTML [10].

Java Unified Expression Language (EL) je speciální jazyk, který je součástí specifikace Java Server Pages. Používá se k vyhodnocování výrazu v textových souborech - nejčastěji JSP stránkách. Je však univerzální a může být použit i v kombinaci s jinými technologiemi než JSP [19].

JNDI Java Naming and Directory Interface je API jazyka Java pro adresářové služby, které umožňuje vyhledávat Java aplikaci objekty a služby pomocí jména [34].

JPA Java Persistence API je API programovacího jazyka Java, které umožňuje popis datových entit pomocí anotací a specifikuje způsob uložení těchto entit v relační databázi - objektově relační mapování [31]. Konkrétní realizací tohoto API je například nástroj Hibernate [17].

JSF Java Server Faces je webový aplikační framework patřící do platformy Java EE, umožňující vytvářet webová uživatelská rozhraní aplikace [32].

Message Driven Bean je komponenta Java EE aplikace dovolující asynchronní zpracování zpráv [34].

reflexe je schopnost počítačového programu, potažmo programovacího jazyka, procházet vlastnosti a meta-data a modifikovat strukturu a chování tříd a z nich vzniklých objektů za běhu programu [39].

resource bundle je objekt nebo textový soubor, který obsahuje objekty specifické pro určitou lokalizaci, např. lokalizované textové řetězce [27].

RMI Remote Method Invocation je technologie meziprocesové komunikace pro platformu Java. Umožňuje vzdáleně volat metody tříd aplikace běžící na jednom virtuálním stroji jazyka Java z jiného virtuálního stroje. Takto je možné komunikovat mezi procesy i po síťovém spojení mezi různými fyzickými stroji [34].

servlet je programová komponenta využívaná k rozšíření možností serveru. I když je serverem v tomto kontextu míněn server podporující libovolný aplikační protokol, většinou je pojem servlet spojen s implementací dovolující rozšiřování možností webového serveru s HTTP protokolem. Specifikace servlet je součástí platformy Java EE. [34].

simple name Jméno třídy v jazyce Java bez specifikace balíčku a dalších modifikátorů. Např. simple name třídy java.lang.Enum je Enum.

tag Fragment uživatelského rozhraní nebo kódu, na který se mapuje atribut inspektované entity v nástroji AspectFaces.

UI User Interface, uživatelské rozhraní. Uživatelské rozhraní je souhrn způsobů, jakými lidé (uživatelé) ovlivňují chování strojů, zařízení, počítačových programů či komplexních systémů [10].

UIProtocol je technologie, vyvíjená na ČVUT, určená k popisu uživatelských rozhraní a doručování těchto rozhraní uživateli.

UIPServer je aplikační server pro platformu UIProtocol, který vznikl v rámci [2].

UML Unified Modeling Language je grafický jazyk určený k vizualizaci, návrhu a specifikaci programových systémů [26].

XML Extensible Markup Language je obecný značkovací jazyk. Umožňuje snadné vytváření konkrétních značkovacích jazyků (tzv. aplikací) pro různé účely a různé typy dat [10].

Příloha B

Seznam použitých zkratek

Acronyms

AOP aspektově orientované programování.

API Application Programming Interface.

AUI Abstract User Interface - abstraktní popis uživatelských rozhraní.

CUI Concrete User Interface - konkrétní uživatelská rozhraní.

EAR Enterprise ARchive.

JAR Java ARchive.

JCA Java Connector Architecture.

JEE Java Enterprise Architecture.

JEEUSC Java EE UIP Socket Connector.

JSP JavaServer Pages.

MDA Model driven architecture - Modelem řízená architektura.

MDB Message Driven Bean.

RAR Resource Adapter Archive.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

UiGE UiGE UI Generator[23].

URL Unified Resource Locator.

UTJEEC UIP to Java EE Connector.

WAR Web Application Archive.

Příloha C

Instalační příručka

Stáhněte z webových stránek JBoss nejnovější verzi aplikačního serveru JBoss Application Server 7:

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

Stažený balíček s aplikačním serverem rozbalte a zprovozněte podle:

`https://docs.jboss.org/author/display/AS71/Getting+Started+Guide`

Je třeba upravit spouštěcí konfiguraci aplikačního serveru JBoss. Postup viz. kapitola 3.5.

C.1 Java EE, UTJEEC a samostatný UIPServer

Ve složce `prepaired_apps` na DVD se nachází soubor `UipAfDemoAppUserRegSeparate.ear`. Zkopírujte jej do složky `standalone/deployments` ve složce s JBoss 7.1.

Spusťte JBoss ze složky `bin` ve složce s JBoss 7.1. pomocí `standalone.sh` pro Linux nebo `standalone.bat` pro Windows.

Ve složce `prepaired_apps` na DVD se v souboru `UIPServer-1.0.zip` nachází UIPServer s připravenou testovací aplikací. Rozbalte tento soubor na pevný disk. Vznikne složka `UIPServer-1.0`.

UIPServer spusťte ze složky `UIPServer-1.0/bin` pomocí `UIPServer.sh` pro Linux nebo `UIPServer.bat` pro Windows.

Poté, co se spustí jak UIPServer, tak JBoss, je možné se pomocí klienta pro UIProtocol pro .NET v základní konfiguraci připojit k UIPServeru. Zobrazí se rozhraní testovací aplikace. Webové rozhraní testovací aplikace je možné zobrazit zadáním adresy `http://localhost:8080/uip` do internetového prohlížeče.

C.2 Java EE s integrovaným UIPServerem

Nejdříve se ujistěte, že není spuštěný samostatný UIPServer a JBoss 7.1, případně je ukončete.

Ve složce `prepaired_apps` na DVD se nachází soubor `UipAfDemoAppUserRegIntegrated.ear`. Zkopírujte jej do složky `standalone/deployments` ve složce s JBoss 7.1.

Spusťte JBoss ze složky `bin` ve složce s JBoss 7.1. pomocí `standalone.sh` pro Linux nebo `standalone.bat` pro Windows.

Poté, co se spustí JBoss, je možné se pomocí klienta pro UIProtocol pro .NET v základní konfiguraci připojit k UIPServeru integrovanému v testovací aplikaci. Zobrazí se rozhraní testovací aplikace. Webové rozhraní je možné zobrazit zadáním adresy `http://localhost:8080/uip` do internetového prohlížeče.

Příloha D

Ukázka konfiguračního souboru UIProtocolu

Fragment konfiguračního souboru UIProtocolu v novém formátu.

```
<?xml version="1.0" encoding="UTF-8"?>
<uipsProperties>
    <uipsId>server1</uipsId>
    <sendWelcomeObjects>false</sendWelcomeObjects>
    <logging> ... </logging>
    <portalCommunication>
        <class>...</class>
        <properties>
            <entry key="user.0.passwordserver">i2home</entry>
            ...
        </properties>
    </portalCommunication>

    <uipCommunication>
        <!--Maximum count of simultaneously connected XML UIP
        clients per one instance (0 - without limitation)-->
        <maxClientsPerInstance>40</maxClientsPerInstance>
        <commInterface>
            <serverListenerClass>...</serverListenerClass>
            <serverConnectorClass>...r</serverConnectorClass>
            <uipTypeClass>....UipXmlTreeFactory</uipTypeClass>
            <properties>
                <!--IP adress of computer network interface
```

```
24                where XML UIP server will listen for new
25                connections (0.0.0.0 - all interfaces)-->
26                <entry key="ServerAddress">0.0.0.0</entry>
27                <!--TCP port where XML UIP server will listen
28                for new connections-->
29                <entry key="ServerPort">5678</entry>
30            </properties>
31        </commInterface>
32    </uipCommunication>
33
34    <httpServer> ... </httpServer>
35    <appInstance> ... </appInstance>
36    <serverCommunication> ... </serverCommunication>
37
38    <uipHandlers>
39        <loadHandlersDynamically>true
40         </loadHandlersDynamically>
41        <runInSeparateThread>true</runInSeparateThread>
42        <handlersLoader>
43            <class>....JavaHandlersLoader</class>
44            <properties></properties>
45        </handlersLoader>
46
47        <!-- If there is more handlers for the same event
48        class it depends on their order. Handlers that are
49        prior to others are executed earlier. Thanks to
50        this behavior it is possible to make chain of
51        handlers for the same event. Handlers for the same
52        event that are not listed inside uipHandlers are
53        inserted to the end of chain in random order -->
54        <autonomousHandler>
55            <class>....EventHandler</class>
56            <disabled>false</disabled>
57            <properties></properties>
58        </autonomousHandler>
59        ...
60    </uipHandlers>
61    <fileStorage> ... </fileStorage>
62    <applicationFormat> ... </applicationFormat>
63 </uipsProperties>
```

Příloha E

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `prepared_aps` ..... balíček s UIPServerem s připravenou ukázkovou aplikací a EAR archivy testovací aplikace s integrovaným UIPServerem a s UTJEEC určené pro nasazení na aplikační server JBoss 7.1
- `source` ..... zdrojové kódy implementace
- `thesis` ... text diplomové práce a některých materiálů ve formátu PDF
  - `source` ..... zdrojová forma práce ve formátu LaTeX
- `measure` ..... výsledky měření a testovací soubory z kapitoly 4