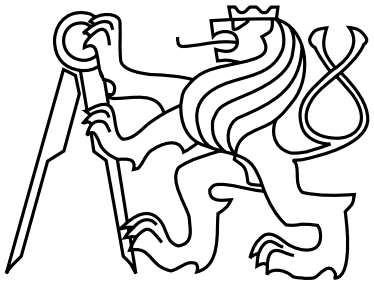
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Aspektově orientovaný vývoj uživatelských rozhraní pro Java SE aplikace

Bc. Martin Tomášek

Vedoucí práce: Ing. Tomáš Černý M.S.C.S.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

22. prosince 2014

Poděkování

Tímto bych rád poděkoval celé svojí rodině za podporu během studia. Dále bych rád poděkoval vedoucíme mé diplomové práce panu Ing. Tomáši Černému za ochotu, pomoc, čas, zkušenosti a příležitosti, které mi poskytoval během celého mého studia.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

Ve Strakonicích 1. 12. 2014 .....

Abstract

Abstrakt

Dnešní aplikace využívají grafické uživatelské rozhraní k interakci s uživatelem. V každé aplikaci můžeme najít vstupní pole, výstupní pole a další komponenty, které slouží k ovládání aplikace. Generování formulářů, která obsahují data, jenž chceme vkládat, editovat nebo prohlížet je časově náročné. Kromě generování správných vstupních polí je také potřeba nahlížet na tvorbu uživatelského rozhraní z pohledu validací, rozložení komponent a bezpečnosti. V těchto případech by bylo vhodné, aby se části rozhraní generovala na základě modelu. V případě aplikací typu klient server může model a data poskytovat server. Toto řešení zajistí centralizovanou správu dat a jejich definicí a umožní klientovi pružně reagovat na změnu datového modelu.

Obsah

Seznam obrázků

Seznam tabulek

Seznam částí zdrojových kódů

Kapitola 1

Úvod

Kapitola 2

Popis problému a specifikace cíle

2.1 Popis problematiky

Softwarové systémy jsou určeny k tomu, aby méně či více úspěšně poskytovali uživateli nástroj, který mu pomůže s řešením problémů. Systém tedy musí komunikovat s uživatelem. K tomuto účelu se využívá uživatelské rozhraní. Vývoj uživatelského rozhraní zabere přibližně 50 % času [5], který je určen na vývoj konkrétního systému. Tento údaj se samozřejmě může lišit v závislosti na účelu a velikosti systému. Při tvorbě uživatelského rozhraní se obvykle zaměřujeme na použitelnost. V tomto případě provádíme testy použitelnosti na cílové skupině, na jejichž základě jsme schopni určit, zdali je návrh použitelný či nikoliv. Důvodem tohoto testování je fakt, že obvykle systém vyvtváříme pro uživatele a ne obráceně. Z výše uvedených skutečností vyplívá, že je potřeba uživatelské rozhraní důkladně testovat, aby bylo pro cílovou skupinu správně použitelné. Bohužel, když se hovoří o uživatelském rozhraní, tak se často zapomíná na to, že toto rozhraní se musí nejen vytvořit, ale také udržovat. Softwarový systém tráví většinu svého života v udržovacím režimu, kterému se říká support nebo li podpora. V této fázi přichází na systém mnoho požadavků, které musí být proveditelné a to za přijatelné náklady. Nedílnou součástí jsou změny, které se týkají databázového modelu a obvykle tyto změny musí reflektovat UI. Podívejme se proto na systém z pohledu vývojáře. Systém pro něj musí být snadno udržovatelný, změny lehce proveditelné a bez větších dopadů na systém. V tomto případě by bylo vhodné reflektovat tyto změny v UI. Nedílnou součástí každého uživatelského rozhraní jsou validace, které by měli reflektovat například změny v databázovém modelu ale i změny v business modelu.

2.1.1 Typy uživatelských rozhraní

Jak již bylo zmíněno, tak uživatelské rozhraní se testuje na základě typu aplikace a jejím použití. Je také důležité vzít v potaz zařízení, na kterém je aplikace provozována. Může se jednat o desktopovou, mobilní či serverou aplikaci. V každém z výše uvedených případů bude návrh uživatelského rohraní podíměn jinými faktory, které jsou specifické pro dané zařízení. Těmito faktory jsou způsoby, jakými se aplikace ovládá, prostředí, v kterém se uživatel právě nachází a účel, ke kterému je aplikace určena. Například aplikace na mobilních zařízení nemusí podporovat klávesové zkratky, ale mohla by podporovat gesta. Obdobně aplikace použitá na desktopu může počítat s použití myše, touchpadu, klávesnice - jak standardní

tak dotykové, či jiného externího zařízení. Je tedy zřejmé, že uživatelské rozhraní je kromě jeho účelu podmíněno i zařízením, na kterém je používáno.

Základnímy ovládacími a vizuálními prvky téměř každé aplikace prvky jsou tlačítka, vstupní pole, přepínače, tabulky, menu a statické texty. Vstupní pole můžeme shrnout do jedné kategorie, která se nazývá formulář. Formulář obvykle osahuje 1 až N prvků, s tím, že každý prvek má zde svojí funkčnost a účel. Účelem je poskytnout uživateli možnost vložení dat, či možnost volby chování aplikace. Funkčností je tato data správně interpretovat a na jejich základě provést specifické akce. Ve formuláři také mohou být pouze statická data, která slouží k reprezentaci aktuálního stavu, který slouží uživateli k tomu, aby pochopil aktuální stav ve kterém se aplikace či jeho část nachází a na základě tohoto stavu mohl rozhodnout o další akci, pokud je toto rozhodnutí vyžadování a umožněno.

2.1.2 Získávání a vkládání dat

Aplikace použí vizuální prvky k tomu, aby uživateli reprezentovala data či umožňila uživateli tato data vytvořit. Moderním způsobem je dnes využívat k získávání a vkládání dat webové služby. Výhodou je, že se klient může připojit na různé zdroje a z těchto dat si vytvářet tzv: mashup. Obvyklé použití je takové, že server získá data z více zdrojů a ty pak interpretuje klientům. Klient tedy nezná orignální původ informací. Jedním z dalších způsobů je vlastní databáze na klientovi. V tomto případě již ale nemůžeme hovořit o klientovi, neboť se jedná o soběstačnou aplikaci, v případě, že nezískává data z jiných dalších zdrojů. Samozřejmě existují i kombinace těchto možností. Volba závisí vždy na konkrétním zadání a účelu, pro který je aplikace navržena.

V případě reprezentace je potřeba data získat. Jak již zbylo zmíněno výše existuje mnoho způsobů kde a jak data získat. Zaměříme se nyní na získání dat z jiných zdrojů. Pokud žádáme o data jiný zdroj, tak jsme obvykle schopni zjistit formát a způsob, jakým o data požádat, ale formát dat neznáme. Nejčastěji jsou data přenásena jako JSON [3] nebo XML. To nám umožní data serializovat do objektu, pokud známe definici objektu. Definice objektu lze získat obvykle v dokumentaci k dané službě, takže námi navržená aplikace očekává data specifického typu. Uvažujme nyní příklad, kdy jsme vytvořili klienta, který zobrazuje jména a příjmení uživatelů v systému. Po určité době, je však potřeba kromě jména i zobrazovat jejich uživatelské jméno. Do dat, která získáváme od služby tedy přibude sloupeček s uživatelským jménem. Nyní musíme naší klientskou aplikaci upravit, tak aby byla schopná zobrazovat i tyto informace. Provedli jsme tady poměrně triviální úpravu. Přidali jsme pole k zobrazení uživatelského jména, upravili jsme objekt, do kterého se data serializovala a v další verzi vydání aplikace se tato změna projeví. Změna tedy není klientům dostupná ihned. Po uričté době se rozhodlo, že se kolonka uživatelského jména odstraní. Tento případ je tedy mnohem horší. Neboť po serializace bude v poli reprezentující uživatelské jméno hodnota null. Pokud jsme jméno pouze vypisovali tak je vše v pořádku, avašak pokud jsme nad ním volali nějaké operace můžeme obdržet vyjímku a aplikace nebude schopna pokračovat v běhu.

V případě vkládání dat do jiného zdroje platí chování a nastavení z odstavce uvedeného výše. Je tedy potřeba znát zdroj, na který data odeslat, formát dat, metodu a popřípadě další dodatečná nastavení služby. Budeme uvažovat stejný příklad jako byl již rozebrán v předchozím odstavci s tím rozdílem, že nyní data vkládáme. Na server tedy nejprve odesíláme pouze jméno a přijímení. Předpokládejme, že tyto dvě hodny jsou serverem vyžadovány. Je

tedy potřeba mít validaci, která hlídá zdali jsou data před odesláním v pořádku, či správně interpretovat odpověď serveru, který sdělí, že data nejsou validní, pokud touto validací disponuje. Při přidání nového pole, u kterého server vyžaduje jeho vyplnění nyní klient přestane správně fungovat a server by měl data odmítat. Musíme tedy udělat změnu na klientovi. Přidat vstupní pole, přidat proměnou, která bude zastupovat uživatelské jméno a novou verzi nasadit a distribuovat. Po určité době, stejně jako v prvním případě se definice dat změní a uživatelské jméno již nebude klientům zasíláno a nebude ani možnost ho vyplňovat. Klient se opět stane nevalidním, neboť při serializaci na straně serveru dojde k chybě, protože klient posílá proměnou, kterou již nyní server neznám. Formulář se tedy opět stane nefunkčním.

2.1.3 Aspektový přístup

Z dosavadního testu je zřejmé, že aplikace obsahuje vizuální prvky, na které lze nahlížet z několika aspektů. Jedním z důležitých aspektů je bezpečnost. Funkce, které může konkrétní uživatel využívat se přidělují na základě uživatelských rolí. V této souvislosti je mnoho přístupů jak role přidělovat a spravovat, ale všechny způsoby mají jedno společné. Ověřeřují, zdali má uživatel právo akci provádět. Souvislost mezi rolí a uživatelským rozhraním je patrá. Mějmě uživatele v role administrátora. Tato role bude mít práva na úpravu uživatelských dat jiných uživatelů. Dále mějme roli hosta, která si může data uživatelů pouze zobrazit. Při detailnějším prozkoumání zjistíme, že existuje množina zobrazovaných dat, která je pro obě role stejná, nicméně pro roli hosta by měla být všechna tato data needitovatelná. Dosáhnout této možnosti lze několika způsoby a záleží na platformě a volbě řešení. Nicméně v Java SE aplikaci, která využívá SWING to znamená, že buď musí být data zobrazována jakou label nebo musí být komponenty vypnuty. V obou případech musí vývojář na základě uživatelské role zvolit jeden z přístupů a nastavovat data na konkrétní komponentě. Pokud zvolí způsob labelů, tak vytváří duplicitní formulář pouze s jiným aktivním prvkem. Pokud jsou data získávána ze serveru, tak na něm jsou již bezpečnostní politiky implementované a postačilo by jejich výsledky propagovat do klientské aplikace. Server by tedy sám rozhodl, jaká data zobrazit.

2.2 Existující řešení

V současné době existuje několik řešení, které se snaží zjednodušit tvorbu uživatelského rozrahní. Níže zmíněné technologie se zaměřují pouze na konkrétní framework. Například JSF, JSP, Swing, Android, Struts, Vaadin. Tyto řešení se zaměřují na inspekci objektů, která již nesou informace o daném objektu. Výhodou inspekce za běhu programu je to, že dokáží na základě typu dat postavit přesné uživatelské rozhraní. Mezi hlavní nevýhody patří samotné generování uživatelského rozhraní, které se dále velmi špatně donastavuje a samozřejmě také fakt, že pro inspekci je obvykle využita reflekce. Níže jsou uvedeny současné frameworky, které umožňují vytvářet generovaná uživatelská rozhraní.

2.2.1 SwinXml

Tento framework se zaměřuje pouze na generování uživatelského rozhraní ve Swingu. Základem je specifikace rozhraní pomocí XML, což je velmi velkou výhodou. neboť specifikace v

tomot formátu má jasně dané možnosti a je na první pohled zřejmé jak bude dáná komponenta fungovat. Knihovna implementuje téměř všechny možnosti, které lze nastavit standardní cestou vývoje Swing aplikace. ActionListenery a dodatečné nastavení si vývojář může specifkovat po vygenerování komponent. Hlavní nevýhodou je, že všechny komponenty, které se generují je potřeba mít specifikované i ve výsledné aplikaci. Není zde žádné zapouzdření komponent a při detailnější specifikaci se stává XML definice až příliš rozsáhlá.

2.2.2 Metawidget

Projekt Metawidget se zaměřuje na vytváření uživateského rozhraní na základě inspekce tříd. Použít ji lze z mnoha populárními frameworky jako jsou Spring, Struts, JSF, JSP a další. Generování uživatelského rozhraní probíhá na základě inspekce již existující třídy a konfigurace konkrétní aplikace. Aktuální verze je 4.0 a vyšla 1.listopadu 2014. Jsou k dispozici pod licencí LGPL/EPL. Co se týče použitelnosti frameworku tak je Mezi hlavní výhody této knihovny patří zejména široká škála podporovaných frameworků a hlavně velká škála podporovaných validátorů. Data lze získat i pomocí REST, nicméně nativní a jednoduchá podpora zde chybí. Framework bohužel neumožňuje generování tabulek.

2.2.3 AspectFaces

Jedná se o framework, který umožňuje inspekci na základě tříd. Framework umožňuje použití různých layoutů a inspekčních pravidel. Výsledné vygenerované uživatelské rozhraní se může pro stejné obejkty lišit na základě specifického nastavení. V současné době je stabilní verze 1.4.0 a na verzi 1.5.0 se pracuje. Vývojář si může své vlastní nastavení generování tříd upravit. Tato nastavení jsou v XML formátu a lze je tedy snadno modifikovat. Framework podporuje velkou škálu anotací z JPA, Hibernate a uživatel si v případě nutnosti může vytvořit vlastní antoaci, která se promítne do inspekce. Framework je prozatím bohužel jednostranně orientovaný na JSF. Tomu odpovídá i způsob generování dat a způsob, jakým je prováděna složitější inspekce.

2.3 Cíle projektu

Existující řešení poskytují mnoho různorodých funkcí. Jejich hlavními výhody jsou zkrácení času, který je potřeba k vývoji a úpravě uživatelského rozhraní. V trendem současné doby jsou webové služby, proto se i v této práci bude soustředit na získávání definice dat z webových služeb a jejích interpretaci na klienta stejně tak jako na plnění této reprezentace skutečnými daty. Inspekce tedy bude prováděna na straně serveru, který zná objekty, s kterými pracuje a klient pouze obdrží jejich definici. Tento přístup umožní klientovi pružně a ihned reagovat na změny v datovém formátu, který diktuje server. Dalším pozitivním vlivem, bude to, že server bude klientovi poskytovat i seznam validací, jimiž musí jednotlivá komponenta vyhovět, aby bylo možné data odeslat zpět na server a ten je správně zpracoval. Celý tento proces by měl být pro kleinta zapouzdřen, aby aplikace nevyžadovala od klienta více informací než je nutné. Mezi nutnou informaci patří specifikace připojení a formát dat. Například JSON, XML. Použití frameworku by mělo být velmi jednoduché a v případě, že bude chtít klient postavit

formulář, tak by mu mělo stačit pouze několik řádků kódu. Dalším důležitým aspektem jsou již existující zdroje na serveru, které by měli po přidání frameworku zůstat stejné.

Kapitola 3

Analýza

3.1 Funkční speficikace

Framework musí umožňovat uživateli vytvářet a dále pracovat s vytvořenými komponentami. Komponenty budou procházet určitým životním cyklem. Kromě samotných komponent musí framework poskytovat i dodatečné funkcionality, které jsou spojeny se získáváním dat, jejich propagací na klienta, zabezpečení, organizací komponent, lokalizací a skinováním komponent.

3.1.1 Funkční požadavky

Z dosavadního popisu problému byly vytvořeny následující požadavky na systém.

- Framework bude umožňovat generovat metadata objektů, na základě kterých budou generovány komponenty.
- Framework bude umožňovat vygenerovat formulář nebo tabulku na základě dat získaných ze serveru.
- Framework bude umožňovat získat data ze serveru.
- Framework bude umožňovat naplnit formulář i tabulku daty.
- Framework bude umožňovat odeslat data z formuláře zpět na server.
- Framework bude umožňovat používat lokalizační resource bundly.
- Framework bude umožňovat validaci dat na základě metadat, která obdržel od serveru.
- Framework bude umožňovat klientovi překrýt chybové validační hlášky.
- Framework bude umožňovat skinovatelnost.
- Framework bude umožňovat specifikovat zdroje ve formátu xml.
- Framework bude umožňovat vytvářet vstupní pole, combo boxy, výstupní pole, textarea, checkboxy, option buttony.

- Framework bude umožňovat vkládat do formulářových polí texty, čísla a datumy.
- Framework bude umožňovat generování readonly komponent.

Z požadavků vyplívá, že framework bude umožňovat získání definici dat na serveru a poté je distribuuje koncovému uživateli. Koncový uživatel tedy nebude potřebovat znát obejkty, s kterými pracuje. Toto zaručí pružnou reakci na změnu dat a generování aktuálních formuláří či tabulek.

3.1.2 Nefunkční požadavky

- Framework generovat metadata, která nebudou závislá na platformě.

3.2 Popis architektury a komunikace

Jak již bylo uvedeno, tak je definice objektů, na základě kterých se budou vytvářet komponenty, generována na serverové straně. Klient tedy komunikuje se serverem a vyžádá si tyto definice. Dále je potřeba definice na klientovi zpracovat. Definice budou platformově nezávislé. Budou tedy popisovat data v obecné formě, aby bylo možné generovat formuláře a tabulky nezávisle na platformě, jazyku a technologii. Referenční implementace bude napsána v jazyce Java s využitím komponentového frameworku Swing. Data, která jsou zasílány ze serveru na klientovi by neměli ovlivňovat ostatní klienty, kteří framework nepoužívají.

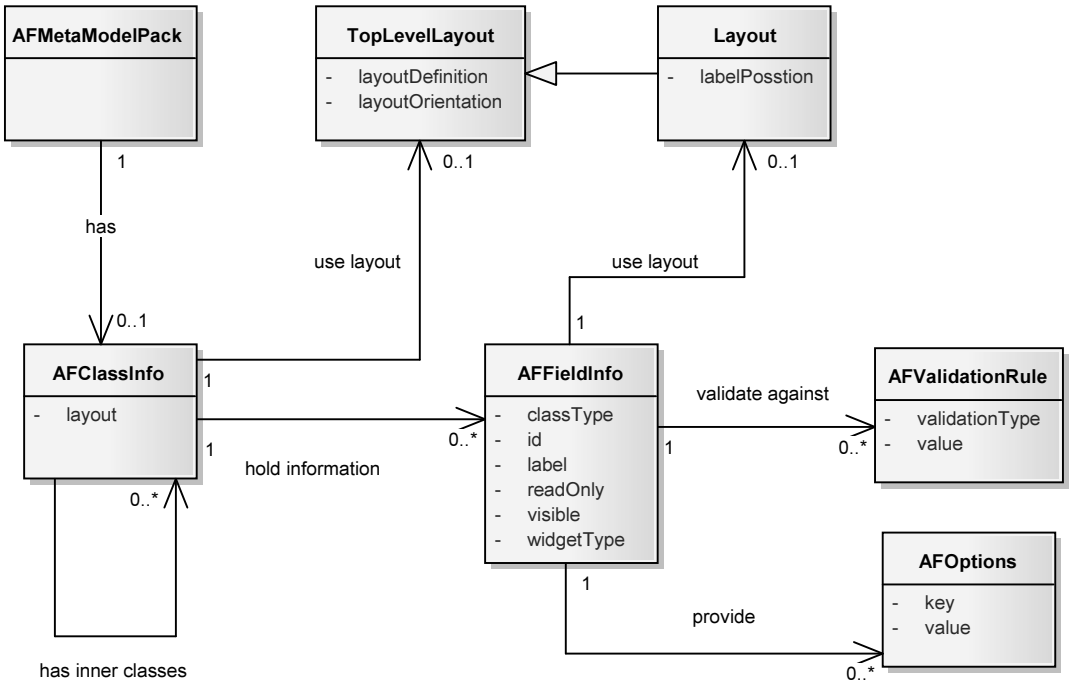
Business proces, který zachycuje generování formuláře včetně validace a odeslání je zachycen na obrázku B.2. Uživatel nejprve specifikuje zdroje, které bude klient využívat. Rozeznáváme následující zdroje:

1. Zdroj s metadaty, které definují komponentu.
2. Zdroj s daty, které budou v komponentech zobrazeny.
3. Zdroj, na který budou data odeslána.

Následně je vygenerována komponenta na základě metadat. V případě, že byl specifikován zdroj s daty, tak klientská část aplikace požádá server o tato data a vloží je do předpřipravené komponenty. Pokud datový zdroj specifikován není, tak zůstane komponenta bez konkrétního obsahu. Komponenta je nyní připravena a uživatel s ní může pracovat. V případě, že uživatel chce odeslat data na server a specifikoval zdroj, na který budou data odeslána, pak framework provede validaci dat. Pokud je validace dat úspěšná, tak se na základě metadat sestaví objekt, který je naplněn daty z aktuálního formuláře a odeslán na server. Server zpracuje request a vrátí klientovi odpověď. Na základě této odpovědi může uživatel dále upravovat formulář či s ním pracovat.

3.2.1 Metadata

Metadata jsou data, která popisují jiná data. Framework, který je popsán v této práci generuje na serverové straně metadata a zasílá je na klienta, který na jejich základě sestaví komponenty a poté je s nimi schopný pracovat. Klient musí být schopný zpětně sestavit data a odeslat je na server. Klient využívá klientskou část frameworku a server serverovou část. Z hlediska implementace je důležité, aby objekty nesoucí informace o metadatech stejné a bylo možné provést na klientovi generování na základě těchto dat. Následující doménový model znázorňuje popis definic objektu, který je vytvořen po inspekci zadaného objektu. Inspekce vytvoří XML popis, který je převeden na obecný popis, jenž lze využít ke generování dat na klientovi. Tento obecný popis je zaslán klientovi, který využívá klientskou část frameworku. Klientská část frameworku očekává tyto objekty a na jejich základě je schopná vygenerovat uživatelské rozhraní. Na obrázku 3.1 je zobrazen doménový model [4], který je použit při popisu metadat objektu. Nejedná se o doménový model frameworku, ale pouze jeho části, která je zodpovědná za reprezentaci metadat.



Obrázek 3.1: Doménový model obejktů obsahující metadat o objektu, nad kterým byla provedena inspekce

3.2.1.1 AFMetaModelPack

Tato třída zapouzdřuje informace, které popisují objekt, nad kterým byla prováděna inspekce. Třída rovněž slouží jako fasáda a nabízí programátorovi upravení metadat po gene-

rování. V případě serveru je toto návratový typ zdroje, na který klient přistoupí chce-li znát metadata, která zdroj poskytuje.

3.2.1.2 AFClassInfo

Tato třída udržuje informace o hlavním objektu, z kterého je vytvářena definice. Třída má dále reference na své vnitřní proměnné a své potomky. Na základě generování dat je potřeba udržet pořadí, v kterém byla nad jednotlivými komponenty prováděna inspekce. V případě inspekce dat je i reference na neprimitivní datový typ jeho proměnná. Nicméně je potřeba, aby klient byl schopný určit, že se jedná o složitý datový typ a určit jeho pořadí v aktuálním objektu, aby mohl rozhodnout na jaké pozici objekt zobrazit. Z tohoto důvodu je v této třídě proměnná classType, která určuje zdali se jedná o vnitřní třídu či primitivní datový typ.

3.2.1.3 AFFieldInfo

Tato třída je zodpovědná za poskytování detailních informací o proměnné objektu, nad kterým byla provedena inspekce. Třída udržuje název proměnné, widget, na který bude proměnná převedena, pravidla, která musí být splněna a název pod kterým, bude prezentována uživateli. Kromě těchto vlastností nese objekt také informace o tom, zdali je komponenta viditelná a zdali je pouze pro čtení.

3.2.1.4 AFRule

Každá proměnná má souhrn vlastností, které musí být splněny. Typ widgetu ještě vždy nemusí určovat datový typ komponenty. Dále neurčuje, zdali je pole povinné či nikoliv. Tento soubor vlastností je popisován v této třídě. Třída využívá ENUM, který specifikuje podporované validace. Důvodem je to, že klient musí být schopný vytvořit tyto validační pravidla a interpretovat je na komponentě svým vlastním způsobem, který je specifický pro technologii, kterou používá. Jednou z dalších výhod je validace XML, z kterého jsou pravidla vytvářena. Framework vytvoří pouze ty validační pravidla, která podporuje. Klient poté musí podporovaná pravidla interpretovat.

3.2.1.5 AFOptions

Některé widgety umožňují, aby si uživatel vybral z několika předem připravených možností. Tyto možnosti musí být klientovi prezentovány. Tato třída udržuje informace o možnostech výběru v dané komponentě. Proměnná klíč je hodnota, která bude odeslána zpět na server a proměnná value je hodnota, která bude zobrazena klientovi. Tímto způsobem lze klientovi zobrazit jakýkoliv text, který bude zpětně mapován na jeho skutečnou hodnotu. Kromě textu lze samozřejmě zobrazit čísla či hodnoty výčtových typů.

3.2.2 Server

Server je zařízení či software, který umožňuje zpracovat požadavky od klientů a na jejich základě vytvořit odpověď. Server tedy poskytuje svým klientům určitý typ obsahu. Způsob a původ obsahu, který server poskytuje je pro klienta povětšinou neznámý. V současné době

je velmi populární přístup, při kterém server získá data z vice zdrojů a poskytne je klientovi. Hovoříme o tzv mashup [9]. Mashup nemusí být pouze z veřejných zdrojů, lze využít i zdroje z privátních zdrojů, či lze k sestavení odpovědi využít další služby. Klient napojený na server tohoto typu nemusí mít o těchto dalších zdrojích vůbec žádnou povědomost a dotazuje se pouze vůči tomuto serveru, který zpracovává jeho požadavky. Klienti, kteří získávají data z veřejných, či privátních zdrojů serveru může být spousta. Mohou to být vlastní privátní aplikace, mobilní aplikace, javascriptoví klienti či další server, který pouze využívá veřejné zdroje serveru k sestavení odpovědi svým vlastním klientům. V těchto případech je potřeba zvážit způsob generování definic dat, které by mohly způsobit stávajícím klientům problémy. V ideálním případě, je tedy potřeba framework integrovat takovým způsobem aby byla zachována stávající funkcionalita a framework ji pouze rozšířil. Ke generování definic objektů jsou potřeba následující věci:

1. Objekt, jehož definice budou generovány.
2. Mapování, na základě, kterého bude rozhodnuto, o jaký typ komponenty půjde.
3. Definice komponenty včetně jejich vlastností jako jsou validace, layout a popis chování komponenty.
4. Layout, v kterém budou komponenty sestaveny.
5. Framework, který provede inspekci.
6. Framework, který bude inspekci řídit a bude interpretovat vygenerovaná data. Tento framework musí zároveň ověřit validitu jednotlivých komponent.

Výše uvedené vlastnosti, nebudou mít vliv na změnu funkcionality. K inspekci a mapování bude využit framework AspectFaces [1], který umožňuje na základě datových typů rozhodnout jakou komponentu využít. Definice komponent a jejich vlastností bude již v plné kompetenci vývojáře, nicméně základní komponenty a jejich chování bude předpřipraveno ve vzorovém projektu aby se vývojář mohl inspirovat. Server využívá serverovou část.

3.2.3 Klient

Klientská část frameworku bude vytvářet komponenty na základě metadat, která obdržela od serveru. Klient nebude mít žádnou znalost o objektech, které mu server poskytuje předtím než obdrží jejich definice. Klient nicméně musí vědět, který zdroj mu poskytne relevantní definice, a který ze zdrojů mu poskytne data odpovídající těmto definiciím. Zároveň také musí vědět, na který zdroj data zpětně odeslat. Zdroj je obvykle specifikován následujícími parametry:

1. Adresa serveru
2. Port
3. Protokol
4. Metoda (get, post, put, delete)

5. Dodateční hlavičkové parametry například content-type

Klient tedy bude muset vždy specifikovat tyto parametry. Z hlediska použitelnosti je vhodné mít tyto specifikace v XML souboru, který bude umět klient jednoduše načíst. Pro usnadnění bude načítání provádět framework. Ukázka je na obrázku 3.1. V ukázkovém příkladu je specifikován zdroj z metadaty, který je vždy povinný. Zdroj se nachází na adrese http://localhost:8080/AFServer/rest/users/loginForm. Zdroj s daty není specifikován, což způsobí, že ve formuláři nebudou žádná data. Formulář bude možné odeslat na adresu http://localhost:8080/AFServer/rest/users/login. Zdroj má identifikátor loginForm. V jednom XML dokumentu lze mít více zdrojů. Data to konkrétního zdroje lze vkládat pomocí EL. V hlavičce může být 0 až N parametrů, přičemž každý parameter musí být uveden ve stejném formátu jako je znázorněno na obrázku 3.1. Obdobný způsob se využívá v JavaEE aplikací v deskriptoru web.xml. Klient umí sestavit request na základě tohoto popisu a interpretovat odpověď od serveru. Není tedy nutné aby uživatel implementoval třídy, které se zvládnout připojit na server a získat data.

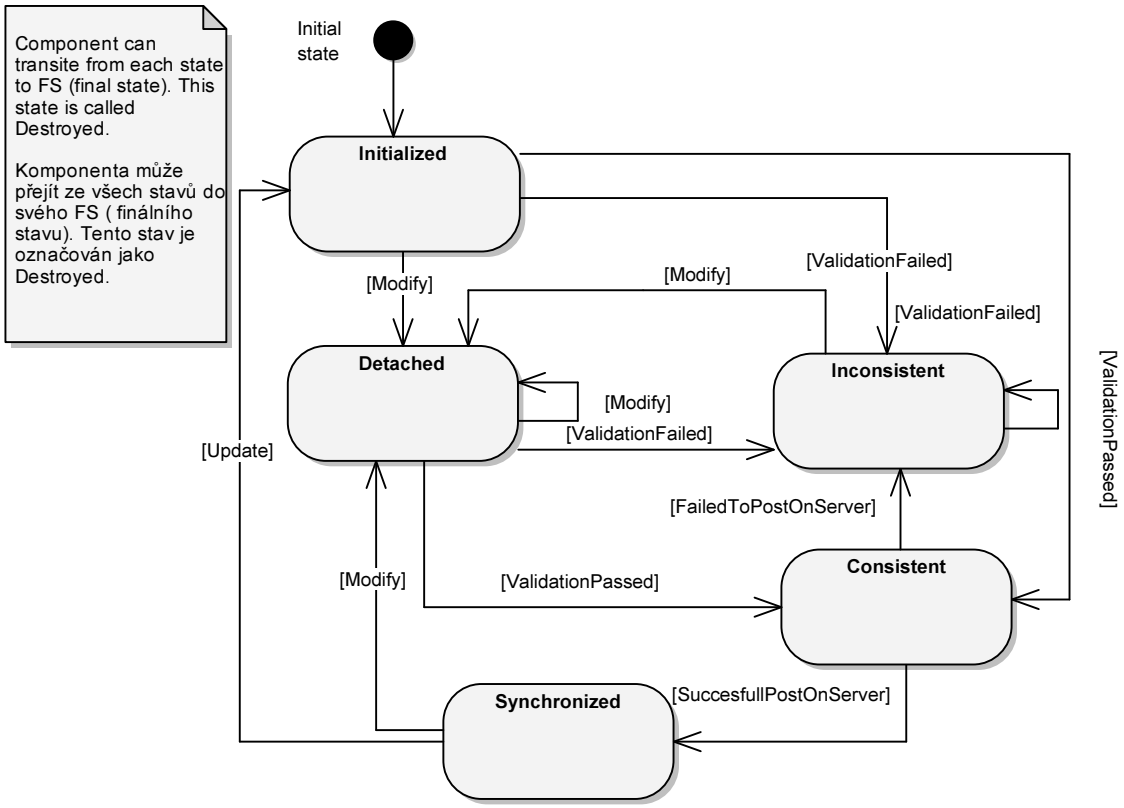
Část zdrojového kódu 3.1: Ukázka XML specifikace zdrojů

```
<?xml version="1.0" encoding="UTF-8"?>
<connectionRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <connection id="loginForm">
                <metaModel>
                        <endPoint>localhost</endPoint>
                        <endPointParameters>/AFServer/rest/users/loginForm</endPointParameters>
                        <protocol>http</protocol>
                        <port>8080</port>
                        <header-param>
                                <param>content-type</param>
                                <value>Application/Json</value>
                        </header-param>
                </metaModel>
                <send>
                        <endPoint>localhost</endPoint>
                        <endPointParameters>/AFServer/rest/users/login</endPointParameters>
                        <protocol>http</protocol>
                        <port>8080</port>
                        <method>post</method>
                        <header-param>
                                <param>content-type</param>
                                <value>Application/Json</value>
                        </header-param>
                </send>
        </connection>
</connectionRoot>
```

Je patrné, že klient je schopný získat definice formulářů či tabulek, naplnit je daty a poté zpět odeslat na server. Důvodem, proč je klient schopný generovat formuláře na základě definice ze serveru je ten, že klient pracuje se stejnými objekty, které popisují metadata, jako server. Prozatím je k dispozici pouze strohý popis dat. Klientská část musí nyní rozhodnout jak data interpretovat, jak s nimi pracovat, jakým způsobem je validovat a jak je znovu

sestavit a odeslat na server. Důležitým prvkem je i způsob uspořádání jednotlivých prvků, jejich velikosti, barvy a texty.

Využití frameworku by obdobně jako v případě serveru nemělo mít vliv na stávající použití aplikace. V případě referenčního řešení ve Swingu generuje klient JPanel, který lze vkládat do dalších panelů a vývojář tak není nikterak omezen co se týče stávající aplikace.



Obrázek 3.2: Životní cyklus formuláře

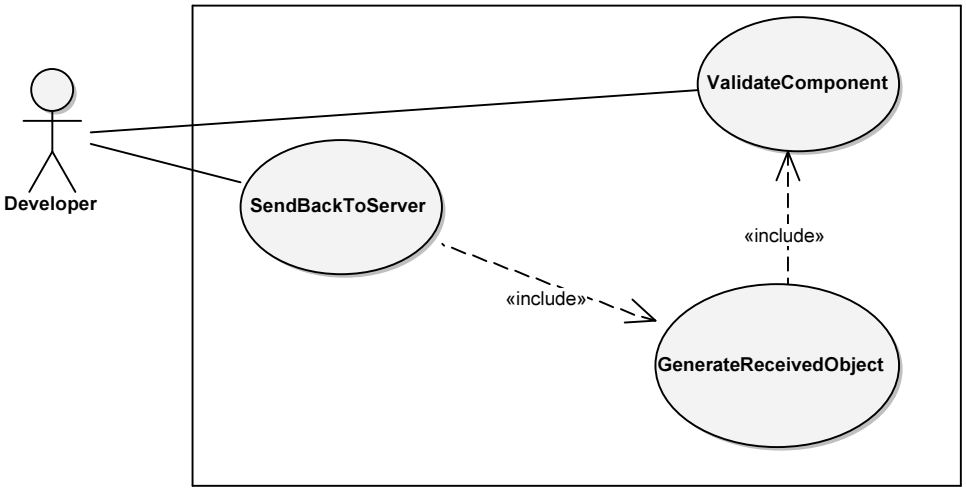
3.2.4 Životní cyklus formuláře

Formulář, jako každá komponenta, má svůj životní cyklus. Jeho stavy jsou znázorněny na obrázku 3.2. Formulář je po vygenerování a po naplnění daty v inicializačním stavu. V tomto stavu může být komponenta nevalidní, neboť data, která obdržela nemusí splňovat požadované validace. K této situaci může dojít, je-li například přidána nová proměná do datového modelu. Toto přidání obvykle probíhá tak, že se v koncové databázy, pokud jí software má, přidá políčko a nastaví se mu defaultní hodnota pro již existující data, či je hodnota null. Nicméně v definici může být pole označeno jako povinné, avšak ve formuláři nemusí být pole vyplněno, což způsobí, že jsou data nevalidní, byť uživatel ve formuláři data nezměnil. Komponenta se pak přepne do stavu Inconsistent. V případě modifikace dat se komponenta dostává do stavu Detached. Tento stav značí, že byla data změněná. Pokud uživatel data stále mění, pak komponenta zůstává v tomto stavu. Ze stavu Detached

přechází komponenta v případě úspěšné validace do stavu Consistent. V případě neúspěšné validace se komponenta dostane do stavu Inconsistent. Z tohoto stavu lze přejít pouze do dvou stavů jedním z ních je stav Detached. Do tohoto stavu lze přejít pokud uživatel změní data ve formuláři. Komponenta může také zůstat ve stavu Inconsistent, pokud uživatel data nezmění a zkusí provést validaci znovu a tato validace opět selže. Konzistentní stav značí, že je komponenta připravená k odeslání dat na server. Pokud odeslání dat selže je komponenta přepnuta do stavu Inconsistent, v případě úspěšného odeslání dat, je komponenta přepnuta do stavu Synchronized. Ze stavu synchronized lze přejít do stavu Initialized, pokud jsou data znovu načtena ze serveru a do stavu Detached, pokud jsou data upravena uživatelem.

3.3 Případy užití

Případy užití a jejich scénáře [4] specifikují chování systému. V této práci lze nahlížet na případy užití ze dvou stran. První z nich je koncový uživatel nebo-li vývojář, který framework využívá. Druhým z nich je samotný framework, který provádí akce, aby splnil úkol, který mu uživatel uložil. V této sekci se zaměříme na případy užití koncového uživatel, které specifikují použití frameworku. Na obrázku B.1 jsou zachyceny všechny tyto případy užití. Pro ukázku detailně rozebereme případ užití na obrázku 3.3. Na tomto případu je znázorněno odeslání dat z vygenerovaného formuláře zpět na server. Součástí je samozřejmě validace zadaných dat a jejich zpětné sestavení, neboť formulář byl vytvořen na základě metadat a klient tedy zná pouze strukturu objektu popsanou těmito metadaty.



Obrázek 3.3: Část případů užití znázorňující odeslání dat na server z vygenerovaného formuláře

3.3.0.1 Validace komponenty

Tento případ užití je znázorněn na obrázku 3.3 a jmenuje se ValidateComponent.

Případ užití: Validace komponenty
ID: 1
Popis: Uživatel využívá framework ke generování formulářů. Hlavním úkolem formuláře je možnost vkládat či upravovat data a odesílat je zpět na server. Před odesláním dat na server musí být provedena validace, aby se zajistilo, že bude formát dat serveru vyhovovat a bude umět s daty pracovat.
Aktér: Uživatel
Vstupní podmínky:

1. Formulář musí být sestaven na základě metadat a framework musí znát formulář s kterým chce pracovat.

Scénář:

1. Případ užití začíná obdržením požadavku od uživatele žadající validaci dat.
2. Systém vyhledá pole k validaci.
3. Dokud existují pole k validaci pak:
   (a) Systém získá konkrétní pole k validaci.
   (b) Systém určí typ komponenty a požádá builder, který ji sestavil o data.
   (c) Dokud existuje validace, která zatím nebylo na poli vykonána pak:
      i. Systém požádá validátor o validaci.
      ii. Pokud validace selže pak:
         A. Systém ukončí validování tohoto pole a zobrazí u něj chybovou validační hlášku.

Případ užití: Sestavení dat
ID: 2
Popis: Uživatel využívá framework ke generování formulářů. Hlavním úkolem formuláře je možnost vkládat či upravovat data a odesílat je zpět na server. Před odesláním dat na server musí být tyto data zpětně sestaveny, aby s nimi server dokázal pracovat.
Aktér: Uživatel
Vstupní podmínky:

1. Formulář musí být sestaven na základě metadat a framework musí znát formulář s kterým chce pracovat. Framework musí znát formát dat, které server očekává.

Scénář:

1. Případ užití začíná obdržením požadavku od uživatele žadající sestavení dat z formuláře.
2. Zahrnout(Validace komponenty).
3. Pokud validace dopadla úspěšně pak:

(a) Systém získá pole, která budou odeslána.

(b) Pokud exsituje pole, které ještě nebylo transformováno pak:

i. Systém určí typ komponenty a požádá builder, který ji sestavil o data.

ii. Systém určí název proměnné a třídu, do které patří a nastaví ji data.

(c) Systém na základě formátu dat, které server očekává rozhodne v jakém formátu data zaslat a převede je na daný formát.

Výstupní podmínka:

1. Data ve formuláři byla převedena na objekt, s kterým umí server pracovat.

Případ užití: Odeslání dat
ID: 3
Popis: Uživatel využívá framework ke generování formulářů. Hlavním úkolem formuláře je možnost vkládat či upravovat data a odesílat je zpět na server. Před odesláním dat na server musí být tyto data zpětně sestaveny, aby s nimi server dokázal pracovat a musí splnit validační kritería
Aktér: Uživatel
Vstupní podmínky:

1. Formulář musí být sestaven na základě metadat a framework musí znát formulář s kterým chce pracovat. Framework musí znát zdroj, na který mají být data odeslána a všechny potřebné informace, které zdroj vyžaduje.

Scénář:

1. Případ užití začíná obdržením požadavku od uživatele žadající odeslání formuláře na server.
2. Zahrnout(Validace komponenty).
3. Zahrnout(Sestavení dat)
4. Dokud je validace či sestavení dat neúspěšné pak:
   (a) Systém zobrazí chybové hlášky a určí, u kterých poli nastala chyba
   (b) Uživatel chybu opraví a požádá systém o opětovnou validaci a sestavení dat.
5. Systém vytvoří připojení na specifikovaný zdroj a odešle data.
6. Pokud odeslání selhalo pak:
   (a) Systém zobrazí chybovou hlášku, že nebylo možné data odeslat včetně odpovědi od serveru, je-li nějaká.

Výstupní podmínka:

1. Data byla odeslána.

3.4 Omezení frameworku

Existují určité možnosti, které nebudou ve frameworku podporovány. V následujícím přehledu budou představeny nepodporované vlastnosti frameworku v aktuální verzi.

1. Inspekce datových proměnných typu List a Array
2. Získávání dat ve formátu XML. Framework plně podporuje JSON.
3. Customizace jednotlivých polí. Framework podporuje customizaci formuláře a všech jeho polí, nicméně nedisponuje možností přizpůsobovat jednotlivá pole.
4. Odesílání dat a validace dat v tabulce. Tabulka je v této verzi pouze readonly, data tedy nemá smyls odesílat zpět na server.
5. Framework vyžaduje ke své funkčnosti jak serverovou tak klientskou stranu. V případě, že klientské straně chybí serverová strana, tak je framework nefunkční. V případě, že serverové straně chybí klientská strana, tak je serverová strana stále schopná vytvářet definice dat.
6. Klientská strana zobrazuje pouze ta data, která obdržela od serveru, nelze vytvářet automatický Mashup na klientovi. Nicméně klient může generovaný formulář umístit mezi jiné komponenty.
7. Klientská strana nedokáže sestavit objekt, který získala z metadat do takové míry aby z něj byla schopná vytvořit instanci. Nebo-li klientská strana si neudržuje konkrétní objekt, který obdržela, pouze jeho popis.
8. Serverová část využívá k automatické inspekci framework. Bez tohoto frameworku není možné inspekci provést, nicméně uživatel si může definovat svou vlastní definici bez nutnosti inspekce dat.

3.5 Uživatelé a zabezpečení

Téměř každá aplikace využívá způsob, při kterém se uživatel autentizuje a aplikace mu na základě jeho rolí přidělí oprávnění. V případě využití bezestavového protokolu, jakým REST je lze posílat informace o uživateli v hlavičce requestu. Tyto informace mohou být samozřejmě zašifrované. Framework podporuje vkládání libovolných informací do hlavičky requestu. Klient si také může zvolit zdali bude využívat http či https protokol. Velmi rozšířenou možností je využitý Oauth. Jednou z možností, je vložení parametrů do hlavičky či do adresy. Vkládání dynamických adres či proměných do hlavičky requestu framework podporuje.

Druhou části jsou uživatelské role, na základě kterých se generuje uživatelské rozhranní. Serverová část využívá framework AspectFaces [1], který podporuje uživatelské role v systému. Je jen na programátorovi, jaký framework na autentizaci a autorizace na straně serveru použije. Jednou z možností je například napsat si vlastni interceptor, který určí o jakého uživatele se jedná, přiřadí mu roli v systému, na základě které se mu zobrazí konkrétní obsah. Server při generování metadat může využít různé mapovací soubory na základě uživatelské role. Specifikace tohoto chování je opět v plné kompetenci uživatele.

3.6 Použité technologie

V následující sekci jsou rozebrány použité technologie. Kromě samotného frameworku je součástí práce i ukázkový projekt na platformě JavaEE.

3.6.1 Java SE - Swing

Klientská část frameworku je schopná vygenerovat formuláře či tabulky a naplnit je daty a data odeslat. Klientská část je přizpůsobena frameworku Swing. Důvodem je, že vývoj Swingové aplikace je rychlý a Swing zná velké množství vývojářů, kteří si framework mohou jednoduše otestovat. Nicméně metadata, která server generuje lze interpretovat v jakékoliv technologii. Swing je knihovna grafických a uživatelských prvků. Poskytuje komponenty, layouty, actionListenery, okna, dialogová okna a další prvky, pomocí kterých lze vytvářet interaktvní aplikace.

3.6.2 Java EE

Java EE je platforma sloužící k vývoji enterprise aplikací. V současné době je oblíbená jak u velkých tak u menších korporací. Java EE přináší podporu pro Restové služby, JFS, JSP, EJB, databázové frameworky, anotace a další komponenty. Aplikace v Java EE se obvykle nasazuje na aplikační server. Aplikační servery mohou být v cloudu a podílet se společně na zpracování requestů. Důvodem využití této platformy je fakt, že klient vytváří requesty vůči serveru a server tato data zpracovává. Je vhodné mít na serveru platformu, která je ověřená a má potenciál ke zpracování těchto requestů. V této práci generujeme data pomocí restového rozhranní a Java EE splňuje specifikaci, které se tohoto rozhranní týká.

3.6.3 AspectFaces

AspectFaces je framework, který umí provádět inspekci nad zadanými obejkty a na základě datových typů a dalších parametrů rozhodovat o to, jaká komponenta se použije pro konkrétní datový typ. Framework využívá AspectFaces k tomu, aby provedl toto mapování a sestavil popis uživatelského rozhranní. Hlavním důvodem využití tohoto frameworku je fakt, že je distribuován jako open source pod licencí LGPL v3 a že lze využít jeho funkcionalitu ke statické inspekci dat. Tato inspekce je již odladěna a není tedy důvod psát znovu již vynalezenou věc.

3.6.4 Ukázkový projekt

Ukázkový projekt demonstruje použití frameworku. Skládá se ze dvou částí. Klientské a serverové. Klientská část využívá pouze Swing. Serverová část je mnohem sofistikovanější a využívá aktuální technologie. Ukázkový projekt zde znázorňuje použití frameworku a jeho omezení. Ukázkový projekt je koncipován, tak aby ho bylo možné nasadit bez nutnosti dodatečného nastavení.

3.6.4.1 Glassfish

GlassFish [2] je open source aplikační server, jedná se o certifikovaný server JavaEE. Umožňuje clustering, monitoring, podporu EJB, REST a JDBC. Architektura jádra je založena na frameworku OSGI, který umožňuje vzdáleně přidávat, startovat či ukončovat komponenty bez nutnosti restartování celého serveru. Důvodem proč je Glassfish využit na tomto projektu je čistě demonstrativní. Nicméně, každý aplikační server má svá specifika, a proto je ukázková aplikace odladěna právě pro GlassFish. Jak již bylo zmíněno Glassfish distribuje vestavěnou podporou pro Rest a umožňuje použití Derby DB v režimu in memory bez nutnosti speciálního nastavení.

3.6.4.2 RestEasy

Jedná se o framework, pomocí kterého lze vytvářet RESTful aplikace. Tento framework může běžet v libovolném servletovém kontajneru. Framework podporuje například JSON, XML serializace objektů, EJB a je splňuje JAX-RS implementaci. Tvorba aplikací s restovým rozhranním je tak díky tomuto frameworku mnohem jednodušší a vývoj je rychlejší.

3.6.4.3 EJB

Enterprise JavaBeans [6] jsou serverově orientované komponenty, které zapouzdřují business logiku a přístup do databáze. Jsou spravovány v rámci serverového kontejneru, který zajišťuje jejich vytvoření i odstranění z paměti. EJB mohou být různých typů.

- Stateless
- Statefull
- Singleton

Jak již bylo zmíněno, o jejich správu se stará serverový kontejner, nemusíme tedy řešit problémy spojené s vytvořením a destrukcí singletonu[7]. Mezi hlavní výhody EJB patří transakční zpracování, zajištění systémových služeb a bezpečnostní autorizace. Abychom definovali či získali přístup k těmto třídám, používáme anotace, které jsou velmi dobře čitelné a srozumitelné.

3.6.5 Derby DB

V ukázkovém projektu je potřeba data ukládat do databáze. Při vytváření byl kladen důraz na to, aby noví uživatelé nemuseli v konfiguračních souborech specifikovat nastavení a mohli ukázkový projekt ihned nasadit a vyzkoušet. Z tohoto důvode je využita light databáze Derby. Ukázkový projekt ji využívá v in memory módu, což znamená, že budou data po zastavení serveru ztracena. Spolu s Derby DB využívá ukázkový projekt ORM s defaultním nastavením na create jsou v čisté databázy vytvořeny požadované tabulky, které odráží definice objektů, jenž jsou anotované jako entity. Další výhodou je možnost využít anotací k nastavení validací přímo na databázy. Toto validace pak mohou být využity při inspekci dat a na jejich základě mohou být vytvořeny validace, či konkrétní komponenty.

Kapitola 4

Implementace

4.1 Architektura

V tomto frameworku rozlišujeme klientskou a serverou část. Serverová část generuje data pro klienta a tímto způsobem ovlivňuje ovládací prvky, které klientská část aplikace zobrazuje uživateli. Diagram nasazení na obrázku B.3 zachycuje použití frameworku. Serverová část frameworku je nasazena na klientovi a je schopná generovat definice formulářů s použitím frameworku AspectFaces [1]. Tyto definice jsou převedeny na model, který je možné upravit a odeslat klientovi. Aby byla serverová část plně funkční je potřeba nasadit aplikaci, v které je využívána na Java EE aplikační server. Nicméně v případě využití pouze staticky generovaných definic, lze aplikaci nasadit na libovolný aplikační server, který bude poskytovat klientům definice kompatibilní s definici poskytovanými při dynamickém generování. Specifikace formuláře, je poté zaslána na klienta, který ji interpretuje za použití klientské části zvané AFSwinx. Tato část využívá i serverovou část a to z důvodu kompatibilnosti objektů a jejich vlastností. Přidání do projektu lze provést tak, že se do adresáři s knihovnami vloží přeložený jar soubor, či se přidá projekt jako Maven závislost. V současné době není framework k dispozici v centrálním repositáři, je tedy potřeba stáhnout aktuální verzi a zkompilovat ji do lokálního repositáře.

4.1.1 Server

Jak již bylo zmíněno, tak server využívá ke generování serverou část frameworku nazvanou AFRest. Na obrázku B.4 jsou zobrazeny třídy a balíčky, které tato část využívá. Jsou zde výčtové typy, které určují podporované komponenty a jejich vlastnosti, dále objekty zodpovědné za informace o volbě layoutu a objekty nesoucí informace o definicích, na základě kterých budou sestaveny formuláře či tabulky klientem a samozřejmě třídy zodpovědné za inspekci dat. Framework doplňuje do AspectFaces několik anotací, které lze využít při generování definic. Jsou to následující anotace:

1. @UIWidgeType - tato anotace určuje typ widgetu, který se použije do xml šablon, které se používají při generování definic je propagován jako proměnná s názvem widgetType

2. @UILayout - tato anotace definuje layout na dané proměnné. Lze specifikovat typ layoutu, jeho orientace a pozice popisu prvku. Do xml šablon jsou tyto hodnoty propagovány jako layout, layoutOrientation a labelPossition.

Výše zmíněné anotace akceptují pouze hodnoty z výčtových typů v balíčku common. V případě typu komponenty nebo-li widgetType přijímá anotace hodnoty ze třídy SupportedWidgets a v případě anotace určující layout lze vložit pouze hodnoty z výčtových typů LayoutDefinitions, LayoutOrientation a LabelPosition. Hlavní výhodou tohoto řešení, je typová kontrola a jistota, že klient obdrží od serveru pouze takové hodnoty, s kterými je schopný pracovat. Stejným principem jsou řešeny validace a proměnné, které definují vlastnosti jednotlových komponent.

4.1.1.1 Generování modelu

Výsledkem inspekce objektu je model, který nese informace potřebné k tomu, aby klient mohl sestavit formulář či tabulku a byl do těchto komponent schopný vložit data získaná ze serveru. Na obrázku B.5 je konečná podoba modelu, který je vytvořen k tomuto účelu. Model je výsledkem hledání analytických tříd z doménového modelu na obrázku **??**. Tento model již byl posán v analytické části, nicméně v této části je již model kompletní a proto zde budou uvedeny pouze změny oproti původnímu modelu. Proměnné, které nesou informace o layoutu, typu komponenty validátorech a jejich typech jsou výčtové typy. Jak již bylo zmíněno výhodou je typová bezpečnost a jednoznačnost vlastností, které framework podporuje. Model slouží také jako fásada, k nastavení dodatečných atributů. Jedním z těchto atributů je proměnná options ve třídě AFFieldInfo. Tato proměnná drží informace o možných hodnotách, které může komponenta nabývat. V současné verzi je tento atribut využit u komponent výběrového typu, mezi které patří například zaškrtávací políčka, či výběrová menu. Programátor specifikuje množinu těchto hodnot, v které klíč určuje hodnotu, jenž bude odeslána na server a text, který bude zobrazen uživateli je určen proměnnou value. Tyto možnosti nejsou generovány automaticky a v případě potřeby je musí programátor specifikovat ručně a to tak, že určí množinu dat a pole, ke kterému je přiřazeno. Třída AFMetaModelPack poskytuje zapouzdřuje způsob jakým se množina dat nastaví na konkrétní políčko a nabízí uživateli funkci, která je schopná nastavení provést na základě dat, zadaných uživatelem.

K dynamickému generování definic se využívá framework AspectFaces [1], který umožňuje na základě mapování rozhodnout jaká komponenta bude použita pro konkrétní proměnnou dané třídy. Dále nabízí určení layoutu, který bude použit a samozřejmě určení mapovacího souboru. Tímto lze docílit mnoha různých transformací. Tento framework je potřeba nejprve nastavit, nicméně toto nastavení provede za vývojáře serverová část frameworku AFRest. Rozhranní AFRest z obrázku B.5 a jeho implementace AFRestGenerator provedou kompletní nastavení a spustí generování dat. Rozhranní umožňuje uživateli určit mapovací soubor a template, která bude použita. Mapování lze použít na všechny promměné obejktu, či může vývojář určit, které mapování se použije na konkrétní proměnnou. Ukázka mapování z frameworku AspectFaces je na znázorněna v ukázce zdrojových kódu 4.1. Proměnná typu String se bude mapovat na vstupní textové pole, kteryé je definováno v structure/inputField.xml, v případě že se bude jednat o typ password, tak se bude proměnná typu String mapovat na vstupní textové pole typu, které místo vepsaných znaků zobrazuje zástupné znaky, kompo-

nenta je definována v structure/inputPassword.xml. Typ Address, což je neprimitivní datový typ se bude mapovat na entitní typ, jehož definice je v structure/entity.xml.

Část zdrojového kódu 4.1: Ukázka mapování proměnných na komponenty

```
<mapping>
        <type>String</type>
        <default tag="structure/inputField.xml" maxLength="255"/>
        <condition expression="${type == 'password'}" tag="structure/inputPassword.xml" />
</mapping>
<mapping>
        <type>Address</type>
        <default tag="structure/entity.xml" />
</mapping>
```

Mapování tedy určí soubor s komponentou, který bude reprezentovat aktuální proměnnou. Soubor s definicí komponenty, je pak dále využit k finálnímu definici proměnné. Ukázka vstupního textového pole je v ukázce zdrojových kódu 4.2. Komponenta je v kořenovém elementu widget. Jelikož se jedná pouze o fragment xml, který je použit ke složení celé definice, jenž je uvedena v příloze v ukázce zdrojových kódů C.1, tak zde není uvedena deklarace XML [8]. Ve výsledném XML již však deklarace již uvedena je. Popis jednotlivých uzlů je v tabulce 4.1.

Část zdrojového kódu 4.2: Ukázka definice komponenty

```
<widget>
        <widgetType>textField</widgetType>
        <fieldName>$field$</fieldName>
        <label>$label$</label>
        <validations>
                <required>$required$</required>
                <minLength>$minLength$</minLength>
                <maxLength>$maxLength$</maxLength>
        </validations>
        <fieldLayout>
                <layoutOrientation>$layoutOrientation$</layoutOrientation>
                <labelPossition>$labelPossition$</labelPossition>
                <layout>$layout$</layout>
        </fieldLayout>
</widget>
```

Knihovna AspectFaces umožňuje určovat způsob jakým bude prováděna inspekce. Tento způsob se určuje v šablonách. K optimálnímu využití je nejvýhodnější použít způsob, při kterém je provedena inspekce všech proměnných, které mají definováno mapování. V případě jednoduchých datových typů je vše v pořádku, nicméně knihovna neobsahovala nativní podporu pro neprimitivní datové typu, v případě že byla použita inspekce, která by nevyužívala JSF. Z tohoto důvodu je důležité, aby se všechny neprimitivní datové typy mapovali na entity.xml, která je znázorněna v části zdrojového kódu 4.3. Framework totiž pro všechny tyto entity provede inspekci znovu a následně části sestaví sestaví a vznikne tak kompletní definice. V tomto bodě, lze určit mapování a šablony, které se mají při rekurzivní inspekci použít. Framework AspectFaces byl proto doplněn o proměnné, které umí vrátit kanoický název třídy a na základě tohoto názvu lze provést nad touto třídou inspekci.

Část zdrojového kódu 4.3: Ukázka definice neprimitvního datového typu

```
<entityClass>
        <entityFieldType>$DataTypeFullClassName$</entityFieldType>
        <fieldName>$fieldName$</fieldName>
</entityClass>
```

Jak je patrné z výsledné definice, tak každý uzel má svého rodiče. Na základě rodiče lze určit jednoznačně určit kam uzel patří. Tato vlastnost umožňuje provádět inspekci i nad třídami, které mají více proměnných stejného datového typu. Klient totiž potřebuje znát strukturu objektu, aby ho mohl zpětně sestavit a odeslat zpět na server, který objekt přijme. Znalost struktury klient taktéž vyžaduje v případě získávání dat.

Tabulka 4.1: Uzly XML, které definuje strukturu dat

Uzel	**Popis**
widget	Typ komponenty. Určuje jak komponentu bude klient interpretovat.
fieldName	Název aktuální proměnné, kterou komponenta zastupuje.
Label	Popis komponenty, který bude zobrazen uživateli.
validations	Validace, které bude umět komponenta ověřit.
fieldLayout	Popis layoutu, který bude na komponentě použit.

4.1.2 Klient

Kapitola 5

Testování

Kapitola 6

Závěr

Literatura

[1] Framework AspectFaces. Dostupné z: <http://www.aspectfaces.com/overview>.

[2] *What is GlassFish v3?* [online]. [cit. 8. 4. 2012]. Dostupné z: <https://wikis.oracle.com/display/GlassFish/PlanForGlassFishV3>.

[3] *Java EE at a Glance* [online]. [cit. 8. 4. 2012]. Dostupné z: <http://www.oracle.com/technetwork/java/javaee/overview/index.html>.

[4] ARLOW, J. – NEUSTADT, I. *UML 2 a unifikovaný proces vývoje aplikací: Objektově orientovaná analýza a návrh prakticky*. Computer Press a.s., Brno, 2nd edition. ISBN 978-90-251-1503-9.

[5] CERNY, T. – CHALUPA, V. – DONAHOO, M. Towards Smart User Interface Design. In *Information Science and Applications (ICISA), 2012 International Conference on*, s. 1–6, May 2012. doi: 10.1109/ICISA.2012.6220929.

[6] ERIC JENDROCK, R. C.-N.-K. H. W. M. I. E. *The Java EE 7 Tutorial* [online]. [cit. 1. 9. 2014]. Dostupné z: <http://docs.oracle.com/javaee/7/tutorial/doc/>.

[7] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA : Addison-Wesley, 1995. ISBN 978-0-201-63361-0.

[8] TIM BRAY, C. M. S.-M. E. M. F. Y. J. P. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. [cit. 20. 12. 2014]. Dostupné z: <http://www.w3.org/TR/REC-xml/1>.

[9] TUCHINDA, R. et al. Building Mashups by example. *Proceedings of the 13th international conference on Intelligent user interfaces - IUI '08*. 2008, s. 15–28. Dostupné z: <http://www.isi.edu/integration/papers/tuchinda08-iui.pdf>.

Příloha A

Seznam použitých zkratek

GNU GPL GNU General Public License

OCL Object Constraint Language

UML Unified Modeling Language

MVC Model-view-controller

AJAX Asynchronous JavaScript and XML

JAAS Java Authentication and Authorization Service

SSH Secure Shell

REST Representational State Transfer

JDBC Java Database Connectivity

OSGI Open Services Gateway Initiative

HTTP Hypertext Transfer Protocol

ORM Object relational mapping

EJB Enterprise JavaBean

JPA Java Persistence API

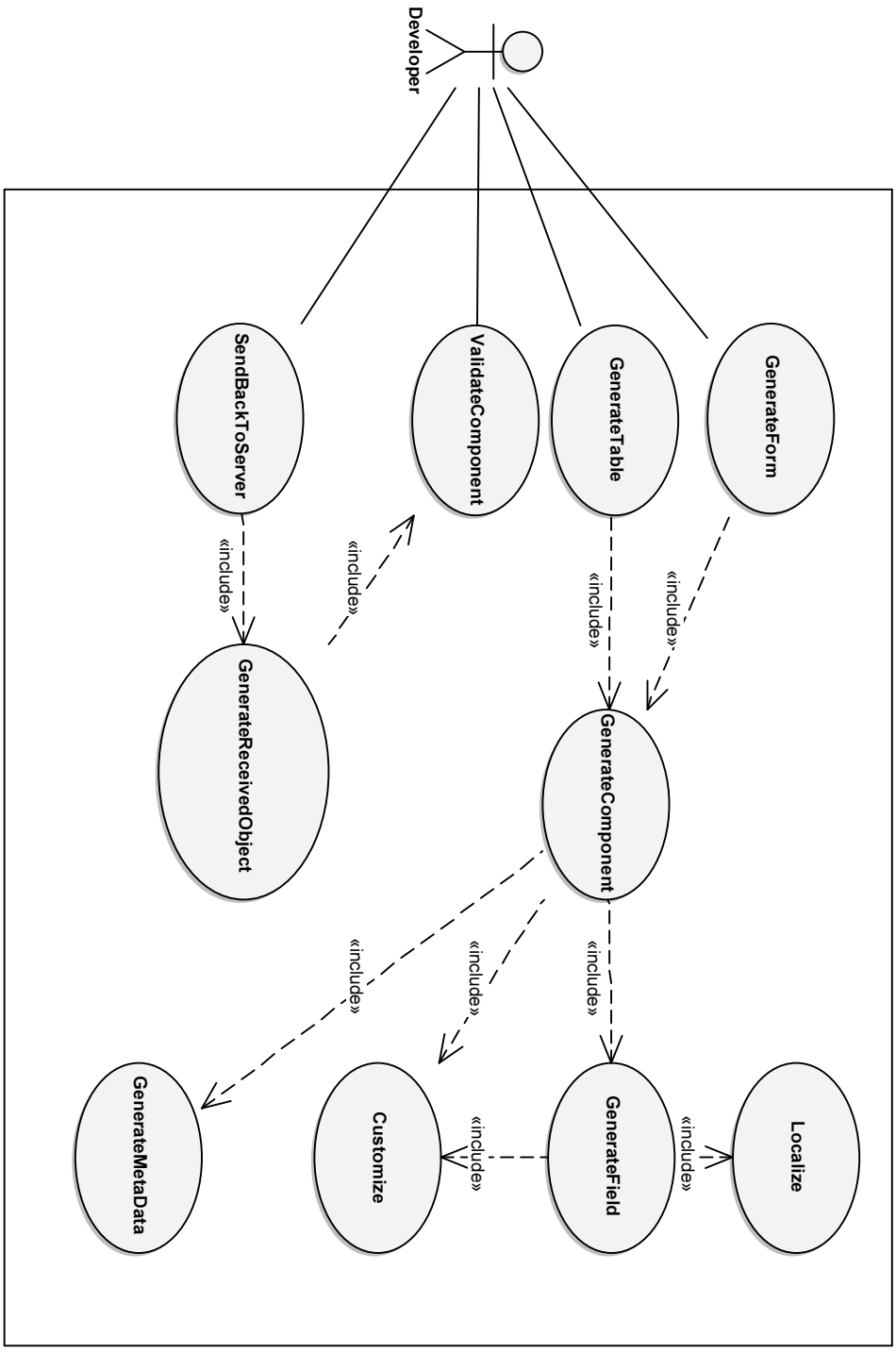
API Application programming interface

EL Expression language

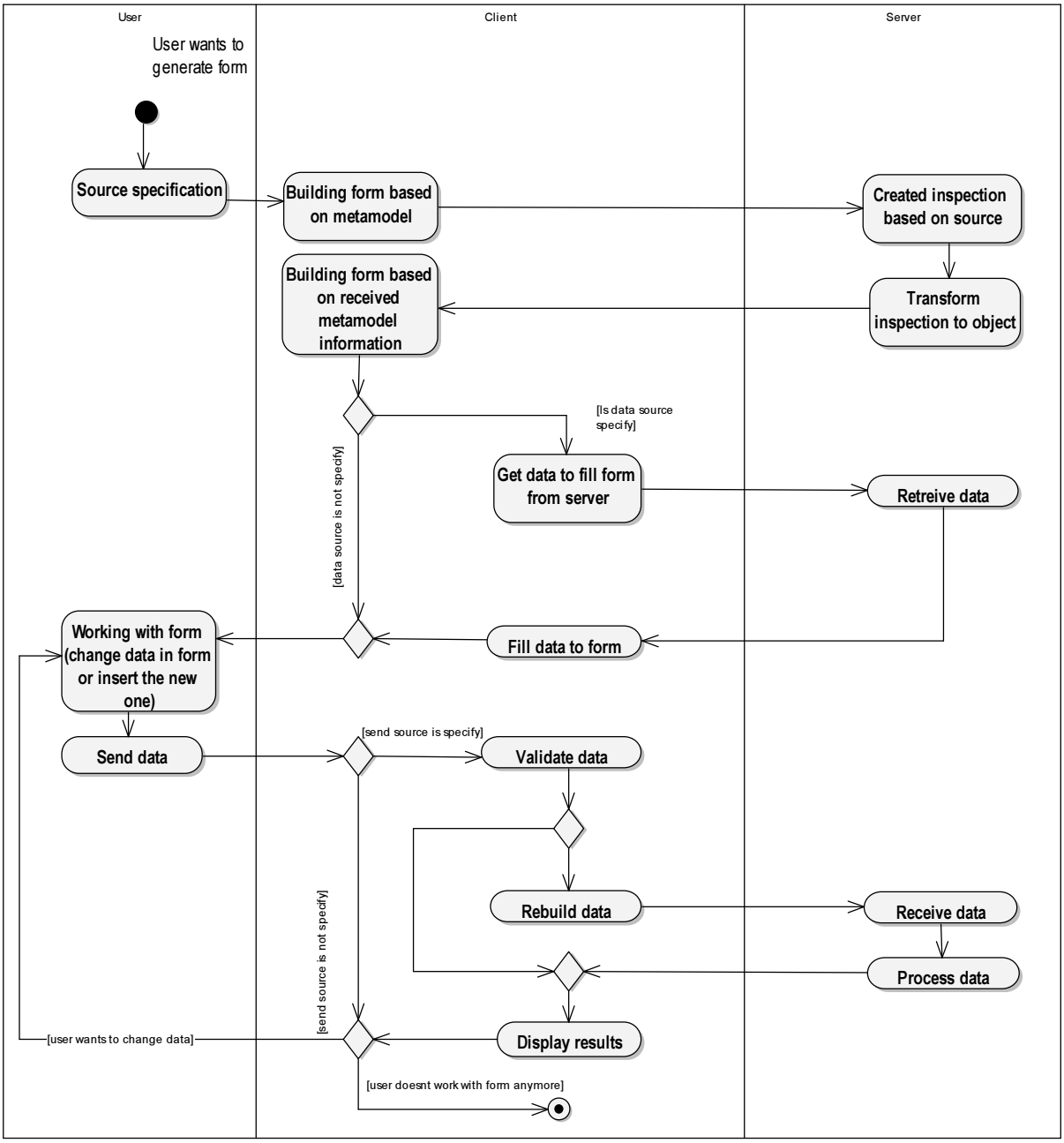
Příloha B

UML diagramy

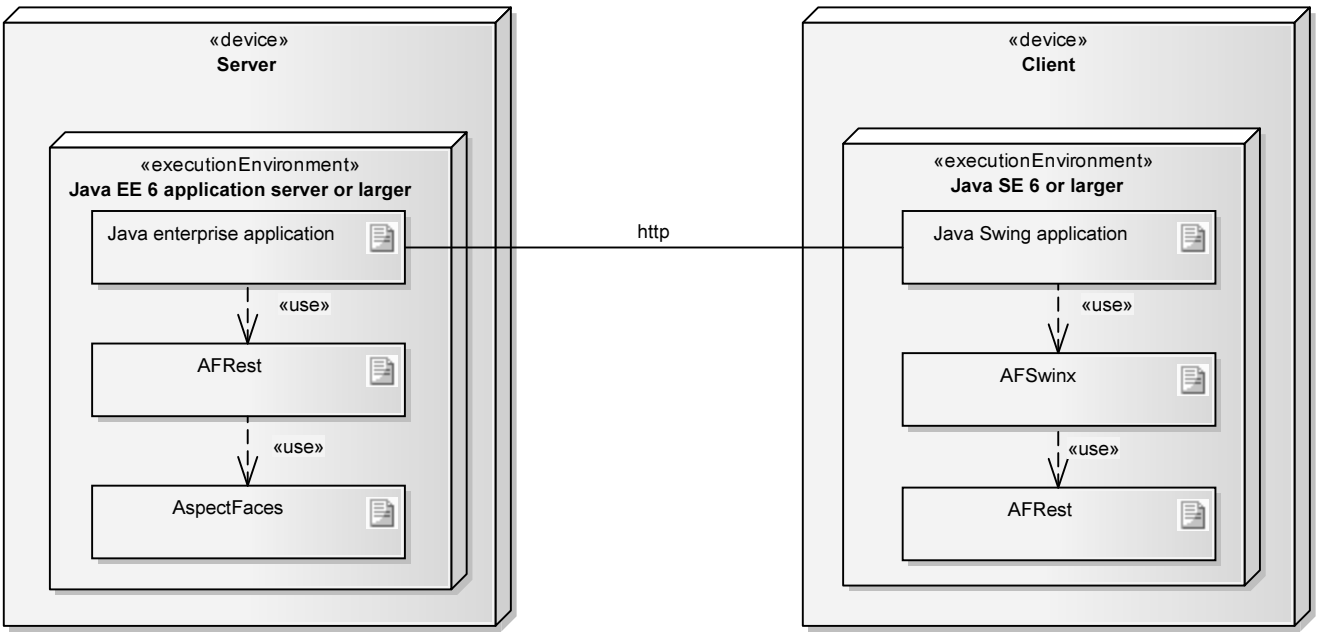
V této sekci naleznete použité UML diagramy, na které bylo v textu odkazováno.



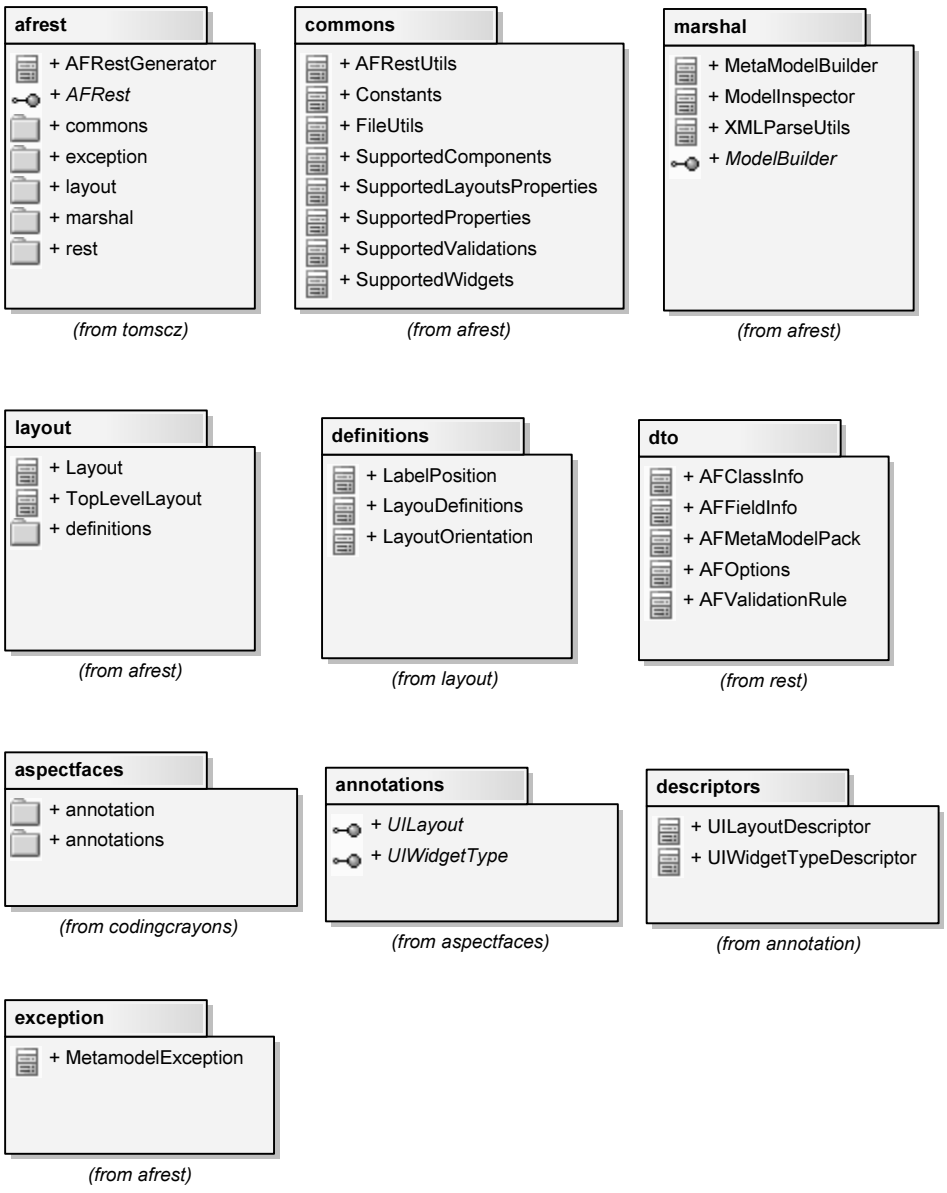
Obrázek B.1: Případy užití frameworku



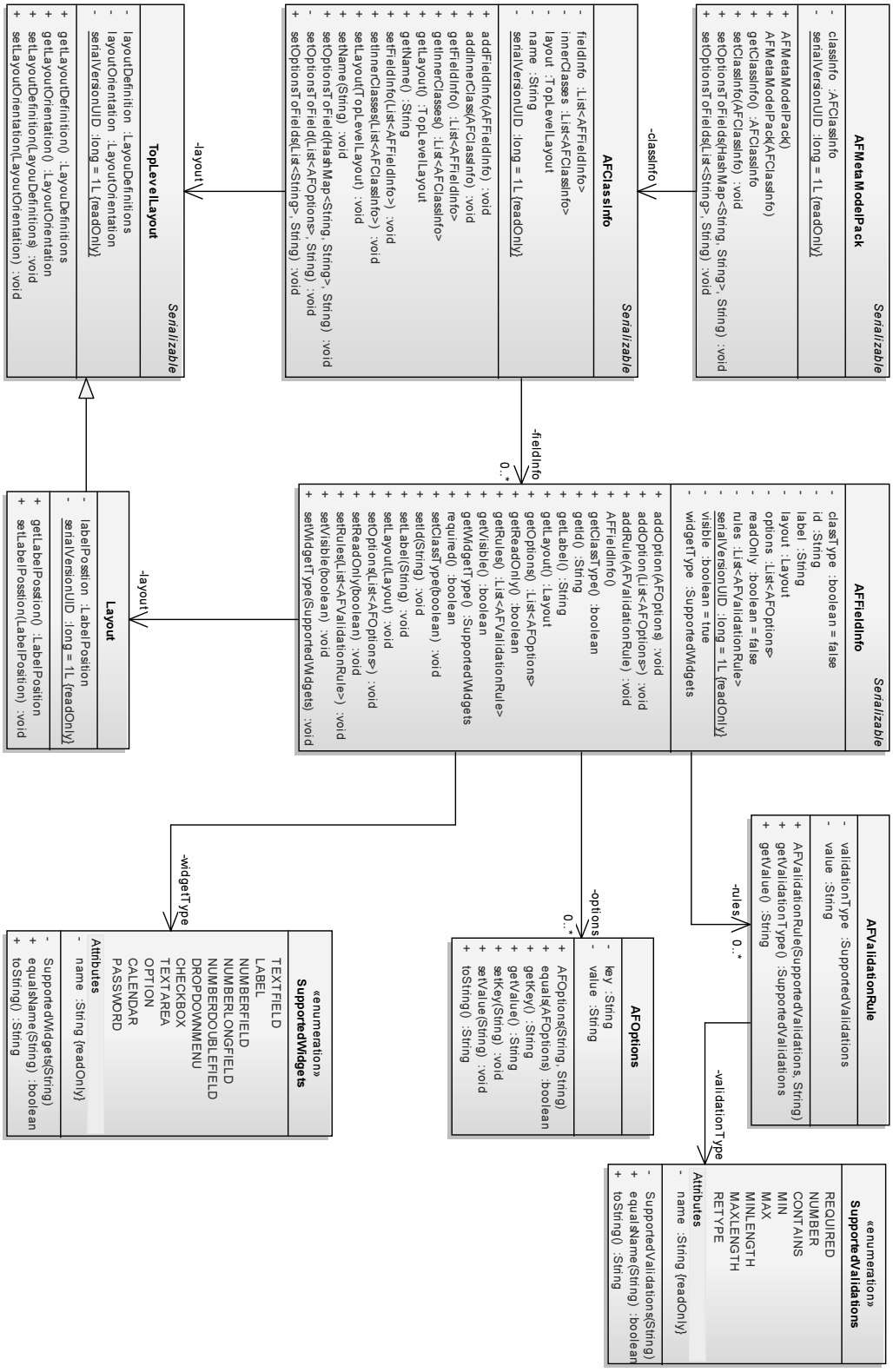
Obrázek B.2: Business model



Obrázek B.3: Diagram nasazení



Obrázek B.4: Diagram balíčků a jejich tříd z AFRest



Obrázek B.5: Doménový model obecných definic komponent

Příloha C

Ukázky zdrojového kódu a XML souborů

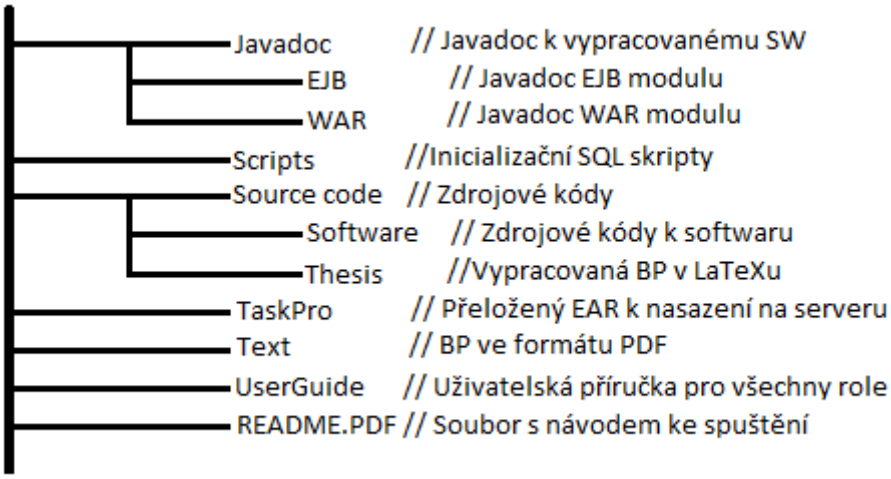
Část zdrojového kódu C.1: Ukázka definice komponenty

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<afRestEntity xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <entity>
                <entityName>person</entityName>
                <entity>
                        <entityName>address</entityName>
                        <widget>
                                <widgetType />
                                <fieldName>street</fieldName>
                                <label>Street</label>
                                <validations>
                                        <required />
                                </validations>
                                <fieldLayout>
                                        <layoutOrientation />
                                        <labelPossition />
                                        <layout />
                                </fieldLayout>
                        </widget>
                        <fieldName>myAdress</fieldName>
                </entity>
                <widget>
                        <widgetType />
                        <fieldName>firstName</fieldName>
                        <label>person.firstName</label>
                        <validations>
                                <required>true</required>
                        </validations>
                        <fieldLayout>
                                <layoutOrientation />
                                <labelPossition />
                                <layout />
```

```
                </fieldLayout>
            </widget>
            <widget>
                <widgetType>textArea</widgetType>
                <fieldName>lastName</fieldName>
                <label>person.lastName</label>
                <validations>
                    <required />
                </validations>
                <fieldLayout>
                    <layoutOrientation>AxisX</layoutOrientation>
                    <labelPossition>before</labelPossition>
                    <layout>TwoColumnsLayout</layout>
                </fieldLayout>
            </widget>
            <widget>
                <widgetType>checkBox</widgetType>
                <fieldName>confidentialAgreement</fieldName>
                <label>Confidential Agreement</label>
                <validations>
                    <required />
                </validations>
                <fieldLayout>
                    <layoutOrientation />
                    <labelPossition />
                    <layout />
                </fieldLayout>
            </widget>
        </entity>
</afRestEntity>
```

Příloha D

Obsah přiloženého CD



Obrázek D.1: Obsah přiloženého CD