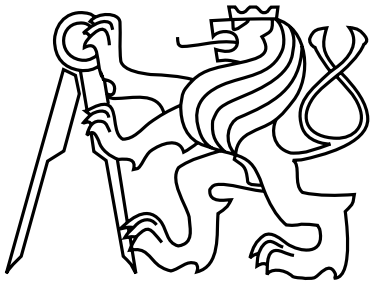
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Servisně orientovaný aspektový vývoj uživatelských rozhraní pro mobilní aplikace

Pavel Matyáš

Vedoucí práce: Ing. Martin Tomášek

Studijní program: Otevřená informatika, Bakalářský

Obor: Softwarové inženýrství

29. března 2016

Poděkování

Zde můžete napsat své poděkování, pokud chcete a máte komu děkovat.

Prohlášení

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

V Kořenovicích nad Bečvárkou dne 15. 5. 2008 .....

Abstract

Translation of Czech abstract into English.

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její obsah. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.
Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato bakalářská práce se zabývá servisně orientovaným generováním uživatelského rozhraní pro mobilní zařízení s využitím knihovny AspectFaces.

První část práce popisuje využité technologie, aktuální situaci v tvorbě UI pro mobilní zařízení, specifikuje požadavky a cíle práce a analyzuje již existující řešení. Druhá část práce ukazuje návrh řešení, které splňuje stanovené požadavky a cíle. Ve třetí části je popsána struktura a vlastní implementace řešení. Poslední část pak obsahuje otestování řešení a ukazuje vzorovou aplikaci na dvou různých mobilních prostředích.

ZMENIT PO DODELANI Práce obsahuje seznam použitých zkratek, které lze najít v příloze A, instalační a uživatelskou pří- ručku v příloze B, použité UML diagramy viz. příloha C a zdrojové kódy aplikace, které jsou přiloženy na CD, obsah tohoto cd je v příloze D.

1.1 Motivace

Nedílnou součástí většíny dnešních aplikací je uživatelské rozhraní. Uživatelské rozhraní by mělo uživateli co nejvíce usnadňovat manipulaci se softwarem a tudíž být intuitivní a použitelné, nemluvě o tom, že by mělo pěkně vypadat. Vývoj takového rozhraní je však časově velmi náročný proces, který zahrnuje nejenom samotný vývoj, ale také rozsáhlé testování, hlavně z hlediska funkčnosti a použitelnosti. Navíc se dnes většina vývojářů softwaru snaží podporovat co nejvíce platforem. Chtějí uživatelům nabídnout možnost operovat s jejich vytvořeným systémem nejen z počítače či laptopu, ale také z tabletu nebo mobilního zařízení. Mobilní verze grafického uživatelského rozhraní nebývá často moc rozdílná a vyskytují se v ní vesměs stejné grafické prvky jako v desktopové verzi a vývojáři tak musí v podstatě vytvořit kopii rozhraní pro mobilní platformu. Tím však vytváří jednu věc hned několikrát a vzniká duplicita. Problémem je, že se často uživatelské rozhraní mění, ať už se změna týká rozložení komponent, přidání nebo odebrání komponenty nebo třeba validace uživatelského vstupu. Takováto změna se pak musí provést na všech platformách a někdy dokonce na více místech v rámci aplikace, což může být v případě rozsáhlých systémů nejednoduchý úkol, který stojí vývojáře spoustu zbytečného času. Pokud bychom byli schopni nadefinovat uživatelské rozhraní jen jednou pro všechny platformy na jednom místě, tento problém bychom odstranili. To však vyžaduje vytvořit pro každou platformu framework, který takto nadefinované uživatelské rozhraní umí interpretovat.

Bylo mi nabítnuto vytvořit takovýto framework pro dvě mobilní platformy, kontrétně pro Android a Windows Phone, což mi přislo velmi užitečné a zajímavé a proto jsem se rozhodl zpracovat toto téma jako bakalářskou práci.

Kapitola 2

Popis problému a specifikace cíle

2.1 Popis problematiky

Softwarový systém má sloužit člověku k řešení nějakého problému. Uživatel přitom často problém blíže specifikuje a systém musí mít způsob, jak uživateli sdělit jeho řešení. K tomu slouží uživatelská rozhraní, která umožňují vzájemnou komunikaci systému s uživatelem. Uživatelské rozhraní přitom není jednoduchá záležitost. V první řadě by mělo být navrženo tak, aby sloužilo uživateli. Mělo by mu umožnit jednoduchou interakci se systémem, být intuitivní, funkční a hlavně použitelné.

2.1.1 Různá uživatelská rozhraní

Aby bylo uživatelské rozhraní použitelné a uživatelsky přívětivé je třeba prozkoumat, jakým způsobem člověk s aplikacemi spolupracuje. Nejen tímto se zabývá disciplína zvaná Human Computer Interaction, která zkoumá potřeby uživatelů z různorodých hledisek. Díky této disciplíně vzniklo množství uživatelských rozhraní, pomocí kterých může člověk s počítačem komunikovat. Jedním z takových rozhraní je textové uživatelské rozhraní, značené CUI, jehož typickým zástupcem je příkazová řádka. Dalším typem je Hlasové uživatelské rozhraní, které dokáže interpretovat povely zadané lidskou řečí. Nejrozšířenějším je však grafické uživatelské rozhraní, zkráceně GUI, které využívá grafické prvky. GUI se stalo velmi oblíbeným právě proto, že je jednoduché a grafické prvky v člověku vyvolávají podobnost s vnějším světem. Také není nutné znát žádně specifické příkazy, jako v případě příkazové řádky, nebo hlasové povely jako v případě hlasového rozhraní. Zmínil bych ještě multimodální rozhraní, která používají k interakci více lidských smyslů a tak jsou vhodná například i pro lidi s postižením [6].

V softwarových systémech je nejběžnějsím způsobem interakce uživatele se systémem právě grafické uživatelské rozhraní a tím se taky v této práci zabýváme. Návrh GUI je potřeba důkladně zvážit, neboť závisí na mnoha aspektech. Důležité je, pro jaké zařízení GUI tvoříme, jaký účel má aplikace, která bude rozhraním disponovat a také stav uživatele a prostředí, ve kterém se nachází. Typ zařízení je důležitý hlavně proto, protože každé zařízení používá jiné ovládací prvky. Zatímco u mobilního zařízení můžeme očekávat použití dotykového displeje, u počítače zase použití myši, klávesnice nebo dokonce jiných externích vstupních zařízení,

jako může být například grafický tablet. Také je třeba vzít v potaz jinou velikost displeje a jiná rozlišení jednotlivých zařízení. Účel aplikace ovlivňuje GUI hlavně z hlediska obsahu, tedy jaké komponenty je nutné mít, aby byla aplikace využívána k daném účelu. Těžko by asi někdo chtěl emailového klienta bez možnosti odeslat zprávu. Stav uživatele a prostředí může zase ovlivnit způsob ovládání aplikace. Příkladem může být palubní počítač v automobilu, na kterém by měl uživatel být schopen přepnout rádiovou stanici, aniž by se přestal věnovat řízení.

Co ale takové grafické rozhraní nejčastěji obsahuje? Běžně GUI disponuje ovládacími a vizuálními prvky pomocí kterých lze aplikaci ovládat. Osobně bych GUI prvky rozdělil na vstupní a výstupní. Vstupní prvky zachycují uživatelský vstup a akce, které systém ovládají a výstupní zobrazují uživateli výsledky těchto akcí, data a aktuální stav systému. Vstupními prvky jsou nejčastěji vstupní pole, do kterých může uživatel napsat text, něco zaškrtnout, vybrat mezi možnostmi atp. Takovéto skupině vstupních polí se říká formulář. Systém samozřejmě může pole ve formuláři předvyplnit a z polí tak udělat výstupní prvky, které rovněž budou uživatele informovat o stavu systému. Dalšími vstupními prvky jsou také tlačítka, které provádějí dané akce jako například odeslání formuláře. Výstupními prvky jsou například statické texty, tabulky nebo seznamy položek.

Pro pořádek, v textu budu používat zkratky GUI i UI, kterými budu bez rozdílu myslet grafické uživatelské rozhraní nikoli tedy jinou formu zmíněnou v této sekci.

2.1.2 Tvorba uživatelského rozhraní

Tvorba uživatelského rozhraní není vůbec jednoduchá záležitost. Vývojáři vkládají do tvorby uživatelského rozhraní velké úsilí a značné množství času. Bylo zjištěno, že uživatelské rozhraní zabírá přibližně 48% kódu aplikace a zhruba 50% času, který vývoji aplikace věnujeme [11]. Další čas a úsilí také zabere testování rozhraní hlavně z hlediska použitelnosti, které opět stojí spoustu času a nákladů. Například vývojář mnohdy nedokáže odhadnout chování cílové skupiny, která systém bude používat, a tak se často dělají testy s koncovým uživatelem, u kterých se zkoumá, jak daný uživatel software ovládá. Z těchto testů často odhalíme, že uživatelské rozhraní je nedostačující a neposkytuje uživateli komfort při ovládání systému, který by poskystkout mělo. Z vlastní zkušenosti s tímto testem také vím, že velkým problémem je uživatelský vstup, který musí být validován, aby uživatel nevložil data, která jsou v rozporu s modelem, na který je rozhraní namapováno [13]. Také je žádoucí zobrazovat uživateli pouze to, co by vidět měl, například na základě jeho uživatelské role v systému. V neposlední řadě je také důležité, jak rozhraní vypadá. Důležitým aspektem rozhraní je, jakým způsobem jsou v něm reprezentována data , a také jak jsou uspořádány jeho jednotlivé části. Z výše uvedeného lze vidět, že je tvorba uživatelského rozhraní opravdu náročný a rozsáhlý proces. Právě proto je poskytovat pro systém více verzí uživatelských rozhraní, například pro různé platformy nebo pro různé uživatelské role, obtížný úkol [13].

Softwarový systém se po nasazení musí také dále udržovat. Ne nadarmo je jedním z hlavních a kritických aspektů dobrého softwaru jeho udržovatelnost, anglicky Maintainability. Udržovatelnost můžeme definovat jako schopnost systému se dále měnit a vyjívet na základě požadavků zákazníka. Změny by přitom měly být lehce proveditelné a neměly by nějak výrazně ovlivnit stav systému. Požadavky na změnu lze očekávat vždy, neboť nevyhnutelně vznikají jako reakce na změny v podnikatelském prostředí (http://faculty.mu.edu.sa/public/uploads/1429431793.2

například desktopovou a mobilní aplikaci, které obě obsahují formulář namapovaný na určitou entitu v databázovém modelu. Tento model se nějak změní, například v dané entitě rozdělíme jeden sloupec na dva. Naneštěstí neexistuje žádný machanismus, který by automaticky zaručil, že je UI v souladu s modelem [13]. Z pohledu vývojáře to pak znamená, že pokud změní databázový model, musí také změnit uživatelské rozhraní v obou klientských aplikacích, aby korespondovalo s novým databázovým modelem, což je jistá forma typové kontroly. Zde nejenom, že musí vývojář udělat dvakrát stejnou věc, ale také může udělat chybu, což vyústí v nefunkčnost systému. Také pokud se takový formulář vyskytuje třeba na pěti místech v aplikaci, změna je už časově náročnější, hůře proveditelná a ještě více náchylná na chybu vývojáře, který může na nějaký výskyt formuláře zapomenout. Takovým zásahem do systému nemusí být jen změna databázového modelu, ale také změna validací uživatelského vstupu, které se týkají i bussiness modelu, nebo třeba změna rozložení či pořadí jednotlivých polí ve formuláři.

2.1.3 Využití webových služeb pro zisk a odeslání dat

Jak už bylo řečeno v grafickém uživatelském rozhraní máme výstupní grafické prvky, jako například tabulky či seznamy položek. Tyto komponenty jsou určeny k tomu, aby zobrazovaly uživateli určitá data. Otázkou je odkud se tato data berou. Je hned několik způsobů, kde mohou být data uložena. Jednou z možností je, že má aplikace vlastní databázi. Takováto aplikace není určena k tomu, aby komunikovala nebo sdílela data s dalšími instancemi této aplikace na jiných zařízeních. Pokud komunikaci chceme, je vhodná architektura klient-server. Server může mít vlastní databázi, ze které poskytuje klientům informace například prostřednictvím webových služeb. Webová služba umožňuje jednomu zařízení interakci s jiným zařízením prostřednictvím sítě[7]. V tomto případě je jedním zařízením server ,druhým klienstká aplikace a interakcí je myšlen vzájemný přenos dat. V mobilních aplikacích jsou velmi populární interpretací webových služeb RESTful Web Services využívající Representional State Transfer (REST), který byl navržen tak, aby získával data ze zdrojů pomocí jednotných identifikátorů zdrojů (URI), což jsou typicky odkazy na webu. Využívá se právě v aplikacích s klient-server architekturou a ke komunikaci používá HTTP protokol. Výhodou využití HTTP protokolu je, že jeho metody poskytují jednotné rozhraní pro manipulaci se zdroji dat. Http metoda PUT se využívá k vytvoření nového zdroje, DELETE zdroj maže, GET se používá pro získání aktuálního stavu zdroje v nějaké dané reprezentaci a POST stav zdroje upravuje [4].

Víme tedy, že klient je schopen získat ze serveru data pomocí HTTP dotazu. Aby přijatá data mohl reprezentovat v UI, musí znát jejich strukturu., což by mohl být problém. Naštěstí jsou data ve spojitosti s RESTful službami nejčastěji přenášena ve formě XML nebo alternativně ve formě JSON[14]. Zmíněné formy dat vzikají serializací objektů, jejichž definici můžeme většinou získat z dokumentace poskytovatele webové služby, stejně tak jako formát, který pro bude pro serializaci použitý. Je také nutné znát metodu, kterou lze pro využití zdroje použít, popřípadě dodatečné parametry, kterými lze webovou službu nastavit. Tato data na klientovi můžeme zpracovat více způsoby. Jednou z možností je napsat si vlastní parser. Dalším způsobem je využít nějaké knihovny, která umí data sama deserializovat do objektu. Obdobně to funguje i v případě odesílání dat. Zdroj webové služby definuje v jakém formátu data přijme a v dokumentaci opět nalezneme definici objektu, do kterého se bude

snažit data deserializovat. Z toho plyne, že klientská aplikace se i při zisku i při odesílání dat musí adaptovat na určitou, předem danou strukturu. Tedy pokud se změní struktura objektu, ze kterého serializací data vznikají a deserializuje se do něj vstup z klienta, je nutné upravit i příslušná místa v klienstké aplikaci, která zpracování a odesílání uživatelského vstupu mají nastarosti.

Představme si nyní následující problém. Mějme server a na něm model naříklad Tým, který obsahuje dva sloupce - název týmu a počet členů. Vytvoříme si klienstkou aplikaci, která tato data získá a zobrazí, například v tabulce. Nyní se rozhodneme, že by měl přibýt sloupec, obsahující zkratku týmu. Nejdříve se na to podíváme z pohledu zisku dat. Upravíme tedy model na serveru a v datech, která jsou poskytována webovou službou, tedy přibude další hodnota. Proto musíme upravit klientskou aplikaci, aby s těmito dodatečnými daty počítala a rovněž je zobrazila v tabulce. Pokud se však rozhodneme, že se nějaký sloupec odstraní, je situace o trochu složitější. Po získání dat nám na klientovi hodnota bude chybět. Pokud nad hodnotou provádíme nějaké operace a nemáme klienta správně ošetřeného, může to vyústit i v pád aplikace.

Nyní budeme data posílat. Předpokládejme, že jsme ještě neprovedli změny výše a klient je tedy v souladu s modelem na serveru. Je důležité poznamenat, že server může určovat, které hodnoty vyžaduje. Pokud tedy přidáme novou hodnotu, kterou server označí jako povinnou, bude pokus neupraveného klienta zaslat data neúspěšný, neboť je server odmítne. Musíme tedy klienta upravit tak, aby bylo možné novou hodnotu zadat, to znamená přidat nové vstupní pole a upravit parser, či objekt, ze kterého se data připravují serializací na odeslání. Nastane-li odstranění nějakého sloupce z modelu na serveru, bude to pro klienta opět problém, protože bude zasílat data obsahující hodnotu, kterou server nezná a ten data opět odmítne. Znovu je nutné klienta upravit.

Poznamenejme ještě, že pokaždé, kdy je nutné upravit klienta, se musí vydat nová verze aplikace. Bohužel v dnešní době je možnost aktualizaci neprovést, a to hlavně na mobilních zařízeních, příkladem může být Google Play na Androidu [10]. Když si novou verzi člověk nenainstaluje, hrozí tvůrcům buď uživatel s nefunkční aplikací nebo chyba na serveru, záleží na provedené změně. Spousta vývojářů tohle řeší třeba podmínkami na verzi aplikace. Znamená to, že na serveru je stále stará verze modelu, která podporuje starou strukturu dat? Nebo označili na serveru nová pole za nepovinná? Druhým používaným řešením je vynutit aktualizaci aplikace, což se mi zdá jako docela dobré řešení, ale i zde se vyskytují otázky. Co když člověk třeba nemá dostatek mobilních dat na stažení nové verze aplikace? Co když na aktualizaci právě nemá čas? Tento problém bychom eliminovali, pokud by server klienta informoval o tom, co vyžaduje a klient by se dynamicky těmto potřebám přizpůsobil.

2.1.4 Existující řešení

Snažil jsem se najít existující řešení pro mobilní aplikace, které by vytvářelo definici komponenty, například formuláře, na základě modelu a které by pro zisk těchto definic využívalo webových služeb. Bohužel jsem nenašel žádné řešení, které by přesně odpovídalo těmto specifikacím, uvádím však řešení, která řeší alespoň jejich část. Dále pak uvádím projekt AFSwinx [**?**], který sice požadavky splňuje, ale není určen pro mobilní aplikace, nýbrž pro Java SE platformu a AspectFaces [3] , z něhož AFSwinx vychází a který je v základu určen pro Java EE aplikace.

2.1.4.1 Řešení z IBM developerWorks

Článek Build dynamic user interfaces with Android and XML [9] popisuje možnost dynamického vytvoření formuláře z XML souboru pro Android aplikace. Podle návodu aplikace stáhne z URL adresy určitý XML soubor, ve kterém je nadefinována struktura formuláře. Návod dále ukazuje, jak stažené XML parsovat a dynamicky vytvořit na jeho základě v aplikaci formulář. Tento způsob tedy formulář centralizuje. Pokud se tedy formulář vyskytuje na více místech v aplikaci a je třeba ho změnit, stačí upravit daný XML soubor. Bohužel není XML dokument generován automaticky z modelu a není využito k jeho získání webových služeb, jinak by tento způsob byl pro naše účely řešením. Také je škoda, že na základě návodu nebyla vytvořena žádná knihovna, kterou by Android aplikace mohly používat.

2.1.4.2 PHP Database Form

PHP Database Form [5] je rozšíření pro PHP. Toto rozšíření dokáže automaticky z modelu v databázi vytvořit HTML kód formuláře, včetně validací jednotlivých polí. Umožňuje vybrat pro vytvoření pouze některou část tabulky a to pomocí SQL dotazu. Dále pak umožňuje dodatečná nastavení. Lze nastavit názvy polí, jejich viditelnost, dodat validace tam, kde nejsou, nastavit, jak se bude pole zobrazovat atd. Hlavními výhodami tohoto rozšíření jsou: menší množství kódu, jednoduché validování dat a možnost upravit si formulář dle libosti pomocí CSS. Využití vyžaduje PHP verzi 5.3 a Apache, Tomcat nebo Microsoft IIS web server. PHP Database Form podporuje všechny majoritně využívané databáze a webové prohlížeče. Dnes už by se i toto rozšíření dalo použít pro mobilní aplikace, neboť existují možnosti vytvářet multiplatformní mobilní aplikace pomocí HTML, CSS a JavaScriptu, které spouští aplikaci na mobilním zařízení v režimu webového prohlížeče. Takovou možností je například Apache Cordova [2].

2.1.4.3 AspectFaces

AspectFaces je framework, který se snaží o to, aby bylo UI generováno na základě modelu [12], k čemuž využívá inspekci tříd. To umožní nadefinovat UI pouze jednou a veškeré změny v modelu jsou automaticky do uživatelského rozhraní reflektovány. UI lze nadefinovat v modelu pomocí velkého množství anotací z JPA, Hibernate nebo si lze nadefinovat i anotace vlastní. Lze určit například pravidla pro dané pole, pořadí v UI nebo třeba label. Framework zatím poskytuje dynamickou integraci pouze s JavaServer Faces 2.0, ale pracuje se na integraci i s jinými technologiemi. Poslední stabilní verze frameworku je 1.4.0 a je dostupný pod licencí LGPL v3.

2.1.4.4 AFSwinx

TODO citovat bakalářku od Martina Tento framework byl vytvořen jako koncept a slouží pro generování uživatelského rozhraní v Java SE aplikacích využívajících pro tvorbu UI knihovnu Swing. Tento framework používá RESTful webové služby pro zisk definic komponent, díky kterým je schopen dynamicky postavit formulář či tabulku. Takové definice komponent vznikají za pomocí části frameworku AFRest, která ke generování dat využívá inspekce příslušného modelu na serveru, na který by měla být komponenta namapována. Jelikož se tvoří

komponenta na základě tohoto modelu, nenastane tak, že by s ním nebyla v souladu. Inspekci tříd zprostředkovává knihovna AspectFaces, které věnuji samostatný odstavec. Definice komponenty je přenášena ve formátu JSON a obsahuje informace o komponentě, například její rozložení, pole, které má obsahovat nebo pravidla, která pro jednotlivíá pole platí. Pole z definice se v případě formuláře interpretuje jako vstupní políčko, v případě tabulky jako sloupec.

2.1.5 Cíle práce

TODO citovat Martinovu bakalářku Vzorem pro tento projekt je výše zmíněný framework AFSwinx. Framework se snaží o zjednodušení tvorby uživatelských rozhraní hlavně z hlediska množství kódu a udržovatelsnosti. Framework na straně serveru využívá inspekce tříd k vytvoření definice modelu, které poskytuje klientovi pomocí webových služeb, stejně tak jako data, kterými se má budoucí komponenta naplnit. Klient tyto informace pouze získává a interpretuje je. Klient také nemá informaci o celém procesu tvorby komponenty, zná pouze nutné informace jako je formát dat, například JSON, XML a připojení. Na vytvoření komponenty stačí klientovi pouze pár řádků kódu. Cílem této práce je vytvořit obdobný framework pro mobilní platformy Android a Windows Phone. Žádoucí je také některé prvky z AFSwinx znovupoužít. Cílem práce je také přinést do stávajícího frameworku něco navíc.

Kapitola 3

Analýza

3.1 Funkční specifikace

V rámci této práce bude zpracován framework ve dvou verzích, pro mobilní platformu Android a mobilní platformu Windows Phone. Musí umožňovat jednoduše vytvářet dva typy komponent, formulář, který umožní uživatelský vstup a list, pro zobrazení většího množství dat uživateli. Kromě vytvoření komponent je nutné poskytkout další funkce, které umožní práci s vytvořenými komponentami, jako je například odeslání dat z komponenty na server. Framework musí samozřejmě disponovat funkcionalitou, která umožní správné vytvoření a nastavení komponenty z hlediska zabezpečení, získávání dat a jejich vložení do komponenty, vzhledu komponenty či její lokalizace. Všechny funkční požadavky jsou uvedeny v následujícím seznamu položek.

3.1.1 Funkční požadavky

Framework by měl splňovat následující požadavky.

- Framework bude umožňovat automaticky vytvořit formulář nebo list na základě dat získaných ze serveru.
- Framework bude umožňovat získat ze serveru data, kterými komponentu naplní.
- Framework bude umožňovat naplnit formulář i list daty.
- Framework bude umožňovat odeslat data z formuláře zpět na server.
- Framework bude umožňovat používat lokalizační texty.
- Framework bude umožňovat validaci vstupních dat na základě definice komponenty, kterou obdržel od serveru.
- Framework bude umožňovat upravit vzhled komponenty pomocí skinů.
- Framework bude umožňovat koncovému uživateli specifikovat zdroje definic komponent, dat a cíle pro jejich odeslání ve formátu XML.

- Framework bude umožňovat vytvářet následující formulářová pole - textové, číselné, pro hesla, pro datum, dropdown pole, checkboxy, option buttony.
- Framework bude umožňovat resetovat úpravy ve formuláři nebo formulář vyčistit.
- Framework bude umožňovat získat data z formuláře i listu.
- Framework bude umožňovat schovat validační chyby.
- Framework bude umožňovat jednoduše získat komponentu i na jiném místě v programu, než kde ji vytvořil.
- Framework bude umožňovat generování komponent určených pouze pro čtení.

Pro uživatele, který bude framework využívat, bude proces tvorby komponenty zapouzdřen. Nemusí vědět, jak definice dat vypadá ani jak se komponenta tvoří či naplňuje daty. Bude potřebovat znát jen kód pro vytvoření komponenty, akce, které lze nad komponentou provádět a jak specifikovat, odkud se bere definice komponenty, data pro její naplnění a kam se případně data odešlou.

3.2 Popis architektury a komunikace

3.2.1 Definice komponent

TODO citovat bakalářku Frameworky pro mobilní platformy Android a Windows Phone, které je cílem vytvořit, navazují, jak už bylo zmíněno, na projekt AFSwinx. Tento framework vytváří na straně serveru tzv. definice komponent, které komponentu popisují z hlediska vzhledu, rozložení i obsahu. Jedná se tedy o metadata [**?**], neboli data, která popisují další data. Tyto definice framework poskytuje prozatím ve formátu JSON, plánuje se i XML, ale zatím není podporováno. Proto budeme s JSON formátem počítat i v tomto projektu. Cílem autora AFSwinx bylo, aby tyto definice komponent byly nezávislé na platformě, což i jejich využitím potvrdíme. Definice typicky obsahuje informace jako

- název definice,
- celkové rozložení komponenty,
- seznam polí, které se v komponentě vyskytují.

Jednotlivá pole mají velké množství dalších vlastností, jako např. identifikátor, popisek, viditelnost, validační pravidla atp. S tvarem těchto definic bude framework počítat, na základě toho se definice bude na klientovi udržovat a z ní vytvářet uživatelské rozhraní.

Taková definice vzniká na serveru na základě inspekce modelu. Za tu vděčíme knihovně AspectFaces, která inspekci vytváří ve formátu XML a AFSwinx ji zobecňuje a převádí do formátu JSON. Na serveru zastupuje roli modelu databázová entita a vlastnosti, které má inspekce modelu zachytit a do definice promítnout, jsou určeny datovými typy atributů a pomocí anotací. Inspekce dokáže do definice na základě datového typu nebo anotace promítnout například pořadí v UI, jakým widgetem bude daný atribut reprezentován či jaký

label bude budoucí widget mít. Definici komponenty je možné získat pomocí HTTP dotazu na konkrétní zdroj na serveru, který AFSwinx používá a je schopen nám takovou definici poskytnout. Tento konkrétní zdroj, poskytující definici komponenty, samozřejmě musíme specifikovat. Také lze určit dva další zdroje, jeden dat a zdroj, na který budeme odesílat uživatelský vstup. V požadavcích jsme definovali, aby framework umožňoval uživateli tyto zdroje specifikovat ve formátu XML. Již v AFSwinx byl pro to vytvořen XML soubor a k němu XML parser. V Android frameworku bude žádoucí z hlediska efektivity tento parser a soubor využít. Jelikož AFSwinx je napsaný v Javě a Windows Phone nepodporuje Javu, nýbrž jazyk C#, a tedy ani import .jar souborů, bude nutné tento parser přepsat. Pro představu jak vypadá struktura zmíněného XML souboru, uvedeme definici všech tří zdrojů pro profilový formulář.

Listing 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="personProfile">
        <metaModel>
            <endPoint>toms-cz.com</endPoint>
            <endPointParameters>/AFServer/rest/users/profile</endPointParameters>
            <protocol>http</protocol>
            <port></port>
            <header-param>
                <param>content-type</param>
                <value>Application/Json</value>
            </header-param>
        </metaModel>
        <data>
            <endPoint>toms-cz.com</endPoint>
                    <!-- ... obdobne jako metamodel-->
            <security-params>
                <security-method>basic</security-method>
                <userName>#{username}</userName>
                <password>#{password}</password>
            </security-params>
        </data>
        <send>
            <endPoint>toms-cz.com</endPoint>
                    <!--... obdobne jako metamodel -->
            <security-params>
                    <!--... obdobne jako data -->
            </security-params>
        </send>
    </connection>
</connectionRoot>
```

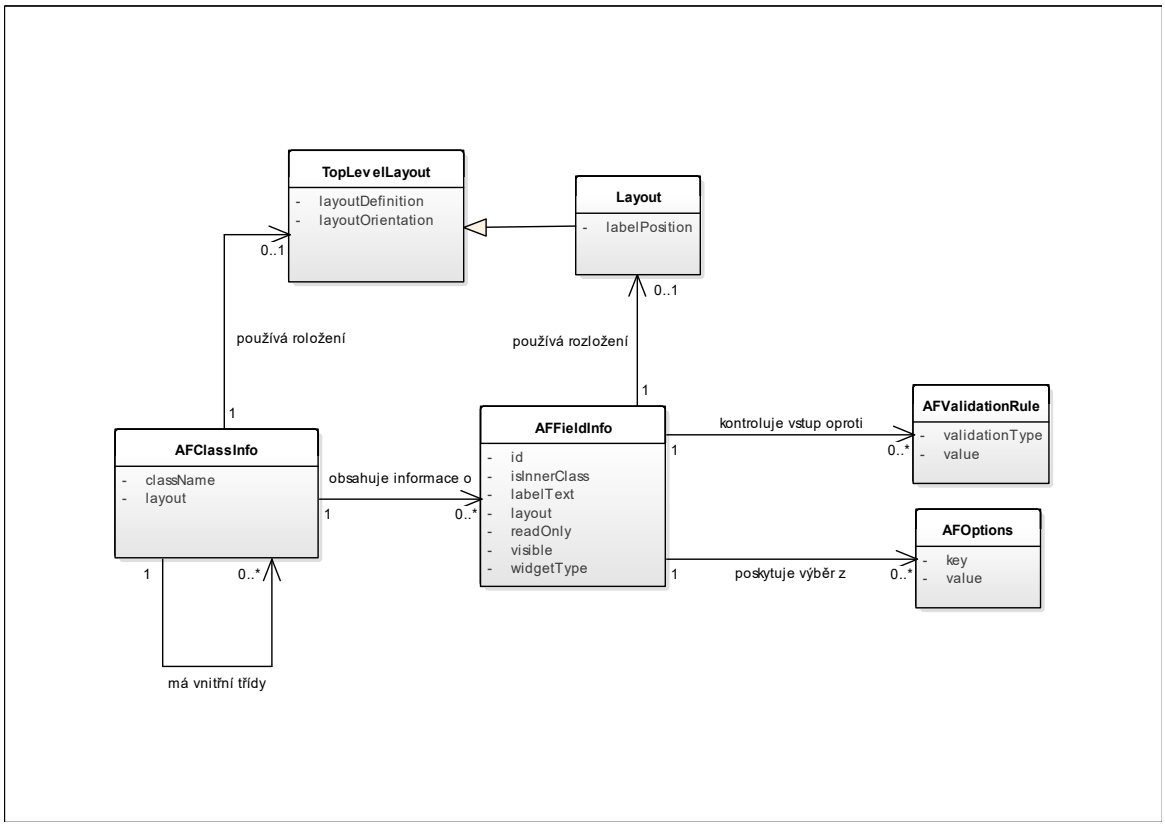
Jak lze z ukázky vidět, jsou zdroje nadefinovány vlastně URL adresou rozdělenou na části a dodatečnými parametry, jako je forma dat, která lze očekávat nebo zabezpečení. Například definice profilového formuláře se nachází na adrese <http://toms-cz.com/AFServer/rest/users/profile> a očekáváme ho ve tvaru JSON souboru. Pokud by byl specifikován port, přibude za toms-cz.com ještě dvojtečka a jeho hodnota. Zdroj dat je nadefinován v uzlu <data> a kam se odešle uživatelský vstup určuje uzel <send>. Lze také specifikovat metodu (get, post, put, delete), která se použije, pro data a meta model je v základu použita metoda

get a pro odeslání metoda post.

Za zmínku stojí výrazy ve složených závorkách označené vpředu hashtagem. Tyto výrazy jsou určeny k nahrazení. V AFSwinx se pak klíč ve složených závorkách hledá v mapě parametrů, kterou framework předává jako parametr metodě kontaktující zdroj, a nahrazuje se hodnotou v ní pod klíčem uloženou. Umožňuje to tak nadefinovat zdroj v XML souboru pouze jednou, například pro více různých uživatelů. Klíč a hodnotu si může uživatel nastavit sám, jen se musí shodovat klíče ve zmíněné mapě a v souboru. V zájmu znovupoužití XML souboru a parseru se tedy tomuto chování budou muset oba tvořené frameworky přizpůsobit.

3.2.2 Reprezentace metadat ve frameworku

Získaná metadata je potřeba ve frameworku nějak rozumně udržovat. AFSwinx už pro to určitou strukturu definuje a je tedy žádoucí ji opětovně využít. Tuto část systému, která uchová získané informace o komponentě, ukážeme na následujícím doménovém modelu, vytvořeného za pomocí UML v programu Enterprise Architect. Dle Arlowa je UML, neboli česky unifikovaný modelovací jazyk, univerzální jazyk pro vizuální modelování systémů. UML je velice silný nástroj hlavně proto, že je srozumitelný pro lidi a zároveň je navržen tak, aby byl univerzálně implementovatelný [**?**]. Doménový model definuje jaké části je potřeba v systému mít a jak se vzájemně ovlivňují. Jde tedy o model popisující strukturu i chování systému.



Obrázek 3.1: Doménový model objektů obsahující metadata o komponentě

Tento diagram z obrázku 3.1 si nyní popíšeme, neboť je pro vývoj obou frameworků důležitý. Pro Android z hlediska toho, že je nutné znát to, co budeme znovu používat a pro Windows Phone budeme tuto část muset přepsat do jazyka C#.

3.2.2.1 AFClassInfo

AFClassInfo udržuje informace o hlavním objektu metadat. Obsahuje informace o názvu objektu a rozložení komponenty. Dále drží definice 0 až N informací o polích, která se mají v komponentě vyskytnout. Součástí je také 0 až N vnitřích tříd, tedy referencí na objekt stejného typu AFClassInfo. Může se totiž stát, že v modelu, nad kterým je prováděna inspekce a ze kterého se metadata vytváří, obsahuje neprimitivní datový typ. Například v modelu Osoba to může být objekt typu Adresa, který obsahuje další atributy jako třeba název ulice či město. Tento typ je ale nutno v komponentě reprezentovat také, a tak je zevnitř provedena jeho inspekce, která je později v metadatech reprezentována jako vnitřní třída.

3.2.2.2 AFFieldInfo

Tento objekt popisuje jednu proměnnou, nad kterou byla provedena inspekce a ze které se má vytvořit pole, které se v komponentě vyskytne. Informuje jaký widget má být při vytváření pole použit, určuje jednoznačný identifikátor pole v rámci komponenty, dále pak, zda má být tvořené pole viditelné a upravitelné, repektive jen pro čtení. Definuje jak bude pole rozloženo, hlavně z hlediska pozice labelu, jehož hodnota je ve AFFieldInfo rovněž zaznamenána. V neposlední řadě jsou v tomto objektu uloženy informace o validační pravidlech, oproti kterým se má validovat uživatelský vstup. Navíc ještě v případě, že by uživatel měl mít na výběr pouze z určitých předem definovaných možností, zahrnuje AFFieldInfo i informace o těchto možnostech.

V tomto objektu je také uloženo, zda se jedná o vnitřní třídu, kterou jsme zmiňovali výše. Tento fakt je velmi důležitý, neboť záleží na pořadí polí v komponentě, ve kterém mají být vykreslovány. Inspekce modelu na serveru s tím počítá, a tak pole umístí na správné místo v metadatech a označí ho jako classType, tedy vnitřní třídu, jejíž popis můžeme nalézt v metadatech v části s vnitřními třídami. V rámci zachování správného pořadí vykreslení polí je tedy nutné, aby klienstký framework fakt, že se jedná o složený datový typ, při vytváření polí komponenty zaznamenal a na pozici, kde tuto skutečnost objeví, vložil pole, o nichž jsou informace uloženy v příslušné vnitřní třídě.

3.2.2.3 AFValidationRule

Tento objekt popisuje pravidlo, které by měl splňovat uživatelský vstup ve vytvářeném poli. Obsahuje typ validace, který určuje o jakou validaci se jedná a případně hodnotu pravidla. Referenční framework AFSwinx obsahuje výčtový typ s názvy validací, které podporuje a které se mohou tedy v metadatech objevit. Například definuje validační pravidlo typu MAX a hodnotou je nějaké číslo. Tedy říká, že hodnota v poli nesmí přesáhnout číslo určené hodnotou pravidla. V obou frameworcích na mobilních platformách tedy bude nutné tyto validace umět zpracovat, přičemž typ zpracování bude na obou platformách trochu jiný.

3.2.2.4 AFOptions

Pro určité typy widgetů, které mají být použity pro vytvoření polí, je nutné specifikovat možnosti, ze kterých si bude uživatel vybírat. Takovými widgety je například dropdown menu nebo skupina radio buttonů. Tento objekt popisuje tyto možnosti formou klíče a hodnoty. Klíč je hodnota, kterou by měl framework odesílat na server a hodnota by měla být zobrazována klientovi.

3.2.2.5 TopLevelLayout

Objekt by měl být využit k popisu rozložení celé komponenty. Objekt definuje dvě vlastnosti. Za prvé je to orientace, tedy ve směru jaké osy bude komponenta či její část vykreslována. Dále je to pak definice rozložení, která má určovat, jestli bude komponenta či její části vykreslovány v jednom či více sloupcích.

3.2.2.6 Layout

Popisuje rozložení částí komponenty, tedy vytvářených polí. Jak je vidět na obrázku **??**, dědí z TopLevelLayoutu orientaci a definici rozložení, které jsou popsány výše. Navíc má vlastnost LabelPosition, která by měla být tvořenými frameworky využita k umístění labelu vzhledem k vytvářenému poli.

3.2.3 Tvorba komponent

Z přijatých a uložených metadat budou frameworky umět vytvořit prozatím dva typy komponent. Jednou komponentou je formulář, který bude hlavně řešit uživatelský vstup, ale může být využit, v případě, že ho předvyplníme daty, i ke zobrazení dat uživateli. Druhou komponentou je list, neboli seznam položek, který bude uživateli umožňovat zobrazení většího množství informací. Ve frameworku AFSwinx tuto možnost zajišťovala tabulka, která však není pro mobilní zařízení úplně vhodným způsobem, protože vyžaduje pro zobrazení mnoho místa, které na mobilních zařízeních většinou nemáme. Když trochu pouvažujeme nad strukturou metadat, která ze serveru dostáváme, zjistíme, že by se dala využít zároveň pro formulář i list. Lišit se bude pouze grafická reprezentace metadat. Zatímco ve formuláři využijeme definic proměnných, nad kterými byla provedena inspekce, k tvorbě formulářových polí, v listu je můžeme využít k tvorbě informací o jedné z jeho položek. Rozložení, které definujeme ve výše popsaném TopLevelLayoutu zase použijeme v případě formuláře k uspořádání polí, tedy jestli budou pod sebou či vedle sebe nebo v jednom či dvou sloupcích. Stejně to můžeme udělat v listu s uspořádáním informací o položce. Některé informace sice přijdou v listu nazmar, jako například typ widgetu nebo validace , ale pořád je to výhodnější, než aby obě komponenty měly vlastní strukturu metadat.

Abychom tedy rozlišili, zda se z metadat vytvoří formulář nebo list, budeme potřebovat dva typy builderů, které komponenty postaví. Uživateli, který bude framework používat, by tedy mělo stačit specifikovat builder, který požadovanou komponentu bude tvořit. Aby bylo používání frameworků co nejvíce uživatelsky přívětivé, způsob tvoření builderů a jejich používání by se neměly lišit. Uživateli pak bude stačit naučit se pouze vytvořit jeden typ komponenty a druhý typ vytvoří pouze výměnou builderu.

Zaměřme se nyní na tvorbu formuláře, který je tou složitější komponentou, jelikož uživatelský vstup, který do něho bude zadáván se musí validovat a odesílat na server narozdíl od listu, který je pouze pro čtení. Návrh toho, jak by měl proces tvorby probíhat, lze vidět v diagramu aktivit v příloze B.1. Diagramy aktivit popsující určitý proces složený z dílčích podprocesů a můžou být použity například právě pro analýzu a popis algoritmu.

V diagramu lze vidět, že pokud bude chtít uživatel frameworku vytvořit formulář, bude muset nejdříve specifikovat, kde framework najde metadata. Ten pak o tato data požádá server. Server musí samozřejmě data dynamicky vytvořit, a tak provede inspekci, o které jsme již psal výše a vytvoří metadata, která pak klientskému frameworku předá zpět. Ten je určitým způsobem zpracuje a postaví na základě nich požadovanou komponentu. Pokud uživatel zároveň nadefinoval i zdroj dat, kterými by se měl formulář naplnit, požádá klient znovu server o tato data. Server je vygeneruje a opět zašle zpět, načež se komponenta těmito daty naplní. Poté se takto vytvořená komponenta předá uživatelovi, tedy vývojáři, který s ní může dále pracovat, například ji vložit do GUI tam, kam potřebuje. Jak je vidět, tak diagram aktivit pouze proces navrhuje a neříká tedy, jak bude daná věc implementována. K tomu se samozřejmě vrátíme v kapitole o implementaci.

Vraťme se ještě k procesu postavení formuláře. Jak jsem již psal, v metadatech se nachází pro každé pole, které má být součástí komponenty, typ widgetu, kterým má být pole reprezentováno. Například to může být textové pole, checkbox nebo skupina radio buttonů. Ke každému widgetu bude potřeba vytvořit vlastní builder, který bude pole vytvářet a také určovat, jak se z widgetu dají získat data a naopak, jak je do něj vložit.

3.2.4 Lokalizace

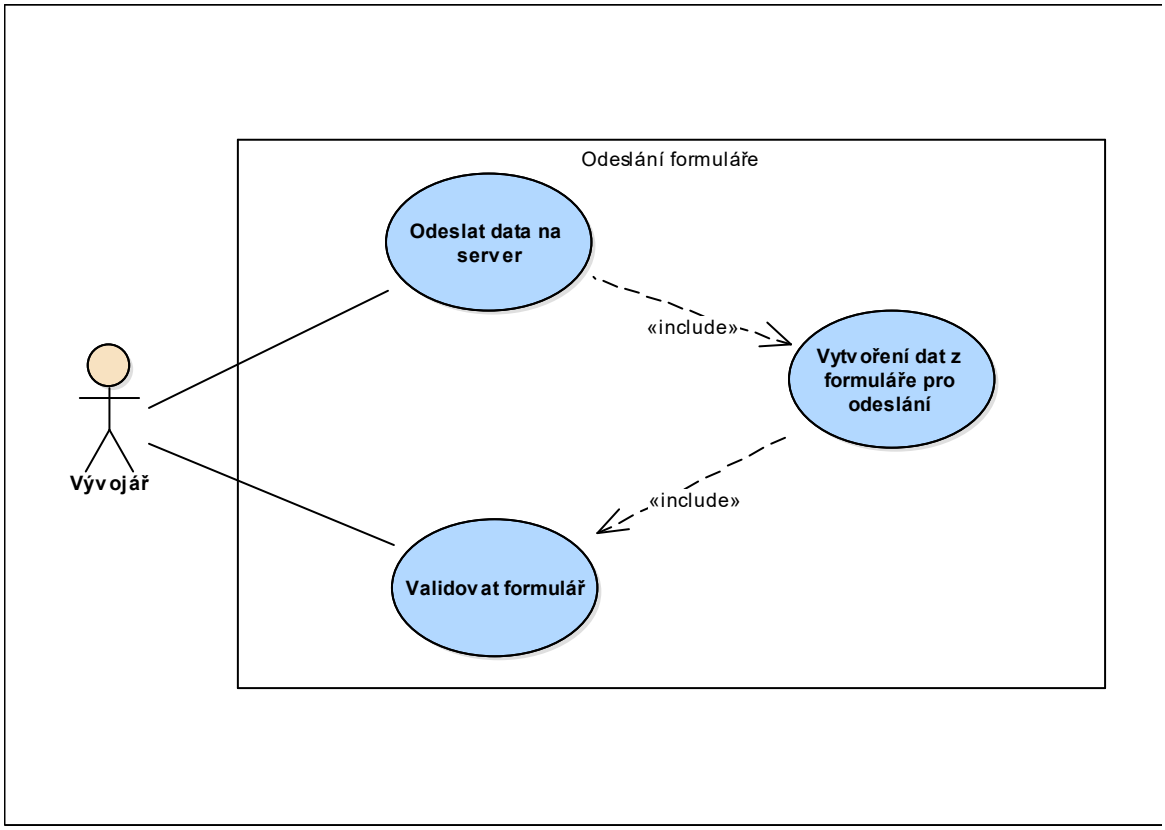
V rámci metadat, která přijdou ze serveru se mohou objevit texty určené pro lokalizaci, tedy překlad. Tyto texty jsou realizované pomocí klíčů, ke kterým lze překlad přiřadit a vyskytují se v labelech, v možnostech, ze kterých může uživatel v daných typech widgetů vybírat nebo ve validačních chybách. Frameworky tedy musí disponovat funkcionalitou, který tuto lokalizaci umožní včetně změny jazyka za běhu aplikace. V Androidu se typicky tyto lokaklizační texty umisťují do souboru strings.xml ve složce values. Tuto vlastnost je požadováno zanechat, a tak i musí být Android verze frameworku navržena. Pro Windows phone je to obdobné. Zde se texty vkládají do souborů Resources.resw, které jsou většinou umístěny ve složce Strings. Aby byl způsob lokalizace textů v aplikaci jednotný, bude možné využít lokalizační část frameworku i nad jeho rámec k překladu například textů tlačítek či položek v menu.

3.2.5 Práce s vytvořenou komponentou

Komponentu nestačí jen vytvořit, ale cílem frameworku je také umožnit s ní další práci. Vývojáři by mělo být umožněno například odeslat formulář, zkontrolovat to, co uživatel zadal nebo upravit její vzhled. Návrh možností, které by měly oba mobilní frameworky podporovat, je vidět v diagramu případu užití, který lze nálezt v příloze. Případy užití nám říkají jak lze používat systém nebo jeho dílčí části [**?**]. Případ užití je iniciován aktérem, jímž je v tomto případě hlavně vývojář, který bude framework používat. V této práci definuji některé případy užití i pro samotný framework, neboť chciv zobrazit, jaké akce musí framework provést pro

splnění úkolu, zadaným uživatelem. Definici posloupností jednotlivých akcí popisuje vždy příslušný scénář případu užití. Pro práci s formulářem byl ještě vytvořen diagram aktivit, který popisuje tři hlavní možnosti práce s formulářem a to odesílání dat, resetování a vyčištění formuláře. Tento diagram lze nalézt v příloze B.2.

Nyní ale zpět k případům užití. Jedním z hlavních případů užití je odeslání formuláře. Tuto část use case digramu systému, kterou lze vidět na následujícím obrázku, podrobněji popíšu. Pro stručnost předpokládám hladký průběh scénáře, samozřejmě všude tam, kde se vyskytne IF, tedy podmínka, by měl být uveden i alternativní scénář, který řeší to, co se stane při nesplnění podmínky.



Obrázek 3.2: Diagram případů užití pro odeslání formuláře

3.2.5.1 Případ užití: Validace formuláře

Na obrázku 3.2 se tento případ jmenuje Validovat formulář a má zachycovat validaci uživatelského vstupu ve formuláři. V diagramu jde také vidět, že by uživatel měl být schopen užívat validaci formuláře i explcitně, tedy ne jen v rámci odeslání dat. Následující scénář tohoto případu užití popisuje postup, jak by měla validace formuláře probíhat.

Vstupní podmínky:

- Framework zná formulář, který chce validovat.

- Tento formulář musí být vytovřený za pomoci metadat.

Scénář případu užití:

1. Uživatel požádá systém o validaci daného formuláře.
2. Systém získá z formuláře všechna pole
3. WHILE existuje nezvalidované pole DO
   (a) Systém získá všechna pravidla, kterým pole podléhá
   (b) WHILE existuje nenavštívené pravidlo DO
      i. Systém získá pro dané pole a pravidlo příslušný validátor.
      ii. Systém provede validaci.
      iii. IF validace selhala THEN
         A. Systém přidá k chybovým hláškám určených pro zobrazení příslušnou hlášku o chybě validace.
   (c) IF alespoň jedna z validací na poli selhala THEN
      i. Systém zobrazí k danému poli validační chyby.

3.2.5.2 Případ užití: Vygenerování odesílaných dat

Tento případ užití popisuje akci frameworku, která musí být provedena, aby šlo úspěšně na server odeslat data. Framework nebude znát přímo objekt, který chce na serveru odesláním dat vytvořit nebo upravit. Co však zná, je jeho struktura, na základě které sice objekt nevytvoří, ale je schopen poskládat pro server přijatelná data, ze kterých si server, který objekt již zná, dokáže tento objekt vytvořit. Důležitý je formát dat, ve kterém mají být serveru tyto data zaslány. Už bylo zmíněno, že AFSwinx podporuje prozatím jen JSON formát, ale počítá se s přidáním dalším. Bylo by tedy dobré, navrhnout tento případ užití pro více možnoných formátů. V diagramu 3.2 je tento use case nazván Vytvoření dat z formuláře pro odeslání a popisuje ho níže zmíněný scénář.

Vstupní podmínky:

- Framework zná formulář, ze kterého chce sestavovat data pro odeslání.
- Tento formulář musí být vytovřený za pomoci metadat.
- Framework musí znát serverem akceptovaný formát dat.

Scénář případu užití:

1. Uživatel chce poskládat z formuláře data k odeslání.
2. «include» Validovat formulář
3. IF validace proběhla úspěšně THEN

(a) Systém získá z formuláře všechna pole

(b) WHILE existuje pole, které nebylo zahrnuto v odesílaných datech DO

i. Systém získá builder, který pole postavil.

ii. Systém požádá builder o data, která se v poli nachází.

iii. Systém určí název proměnné a třídu, do které patří, a nastaví jí data.

iv. Systém podle formátu dat, které server očekává, rozhodne, v jakém formátu data zaslat a na tento formát data převede.

3.2.5.3 Případ užití: Odeslání dat na server

Posledním případem užití z obrázku 3.2 je use case Odeslat data na server. Ten zahrnuje předchozí případ, což jde ostatně vidět v níže uvedeném scénáři.

Vstupní podmínky:

- Framework zná formulář, ze kterého chce sestavovat data pro odeslání.
- Tento formulář musí být vytovřený za pomoci metadat.
- Framework musí znát zdroj, na který mají být data odeslána a všechny potřebné informace, které zdroj vyžaduje.

Scénář případu užití:

1. Uživatel chce odeslat data z formuláře na server.
2. «include» Vytvoření dat z formuláře pro odeslání
3. IF bylo vytvoření dat z formuláře úspěšné THEN

   (a) Systém odešle data na specifikovaný zdroj

   (b) Systém informuje uživatele o výsledku akce

3.2.5.4 Úprava vzhledu komponenty

Pozastavil bych se ještě nad případem užití Nastavit skin, který lze najít v příloze B.3 Vzhled je nedílnou součástí každé aplikace s grafickým uživatelským rozhraním. V Android aplikacích se vzhled definuje buď v XML šablonách pro daný view, to v případě, že je GUI vytvořeno staticky [1], nebo pomocí Javy v případě, že jej vytváříme dynamicky. Na Windows Phone platformě jsou obdobou xml šablon xaml soubory a případě dynamického vytváření, slouží k úpravě metody napsané v C#. Ve Windows Phone aplikacích se však preferuje neměnit barvy komponent. Ke změně barev slouží barevná témata, která mají výhodu v tom, že jsou konzistentní na všech zařízeních [8]

. Navíc Windows phone umožňuje změnit barevné téma zařízení v nastavení. Toto téma neovlivní jen GUI operačního systému, ale také aplikace. Může se tedy stát, že pokud vývojář nastaví například barvu textu na bílou a aktuální téma, které zařízení používá, má bílé

pozadí, nepůjdou texty jednoduše vidět. Windows Phone verze frameworku bude tedy muset zohlednit i tyto situace.

Jelikož se komponenty tvoří dynamicky v rámci frameworku a uživatel k procesu tvorby nemá explicitně přístup, je nutné poskytnou mu možnost nadefinovat vzhled komponenty předem. K tomuto účelu by měly frameworky disponovat skiny, které bude možno nastavit builderu komponenty. V případě, že by uživatel frameworku skin nenadefinovat, měl by existovat určitý základní vzhled. Od tohoto základního skinu může uživatel ve svých skinech dědit a překrýt pouze části, které mu nevyhovují a chtěl by je jinak. Po sestavení komponenty už není jednoduše možné za pomocí frameworku tento vzhled změnit. Framework sice bude poskytovat možnost získat reprezentaci komponent i jejich částí, ale použití již bude komplikovanější a bude vyžadovat znalost dynamické tvorby UI na jednotlivých platformách.

3.2.6 Práce na existujícím řešení

Cílem práce je také přidat nějakou hodnotu do stávající verze frameworku AFSwinx. Pro tento účel jsem se rozhodl přidat novou anotaci, kterou zohlední dříve zmiňovaná inspekce na serveru při tvorbě metadat. Cílem anotace bude sdělit klientovi, že se má anotací označený atribut v modelu na serveru porovnat druhým atributem, který bude určen v parametrech anotace, ve smyslu menší nebo rovno než. Anotace se bude týkat hlavně datumů a bude se tedy řešit, zda anotací označený atribut typu Datum obsahuje datum dřívejší nebo stejné než druhý atribut určený v parametrech anotace. Tato informace se v metadatech objeví prozatím pouze u informací o poli s typem widgetu CALENDAR, tedy u těch, jež mají být reprezentovány jako DatePicker. Píši prozatím, neboť takováto anotace by se dala využít i pro číselné atributy. Kontrétně bude informace přidána do sekce rules, tedy bude se jednat o validační pravidlo. AspectFaces [3] ve své dokumentaci popisuje, že pokud chce uživatel přidat novou anotaci, musí vytvořit tzv. anotační deskriptor, který pak musí zaregistrovat v konfiguračním souboru aspectfaces-config.xml uloženém ve složce WEB-INF. Tím se zajistí, že si inspekce anotace všimne a promítne ji do metadat ve formě XML. Jak jsme již ale psal, AFSwinx tyto XML soubory ještě převádí do jiné platformově nezávislé podoby a v tomto procesu tedy bude nutné taktéž provést změny. Po vytvoření anotace je nutné přidat ji na daná místa na serveru a také je žádoucí vytvořit validátory nejen v obou vytvářených frameworcích pro mobilní platformy, ale také v již existujícím řešení AFSwinx pro platformu Java SE.

Literatura

[1] [online].

[2] *Apache Cordova* [online]. [cit. 23. 03. 2016]. Dostupné z: <https://cordova.apache.org/>.

[3] *AspectFaces framework* [online]. [cit. 23. 03. 2016]. Dostupné z: <http://www.aspectfaces.com/overview>.

[4] *What Are RESTful Web Services?* [online]. [cit. 22. 03. 2016]. Dostupné z: <https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html>.

[5] [online].

[6] *Typy uživatelských rozhraní a jejich specifika/old – Wikisofia* [online]. [cit. 22. 03. 2016]. Dostupné z: <https://wikisofia.cz/index.php/Typy_uživatelských_rozhraní_a_jejich_specifika/old>.

[7] *Web service* [online]. [cit. 22. 03. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Web_service>.

[8] [online].

[9] ABLESON, F. *Build dynamic user interfaces with Android and XML* [online]. [cit. 23. 03. 2016]. Dostupné z: <http://www.ibm.com/developerworks/xml/tutorials/x-andddyntut>.

[10] BARTLETT, M. *Enable or Disable Auto-Update Apps in Google Play for Android* [online]. [cit. 23. 03. 2016]. Dostupné z: <http://www.technipages.com/auto-update-apps-setting-google-play>.

[11] 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.

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

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

[14] JENKOV, J. *Web Service Message Formats* [online]. [cit. 23. 03. 2016]. Dostupné z: <http://tutorials.jenkov.com/web-services/message-formats.html>.

Příloha A

Seznam použitých zkratek

CUI Character User Interface

GUI Graphical User Interface

UI User Interface

REST Representational State Transfer

URI Uniform Resource Identifier

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

XML Extensible Markup Language

Java SE Java Platform Standart Edition

Java EE Java Platform Enterprise Edition

URL Uniform Resource Locator

PHP Personal Home Page

HTML HyperText Markup Language

SQL Structured Query Language

CSS Cascading Style Sheets

JPA Java Persistence API

LGPL GNU Lesser General Public License

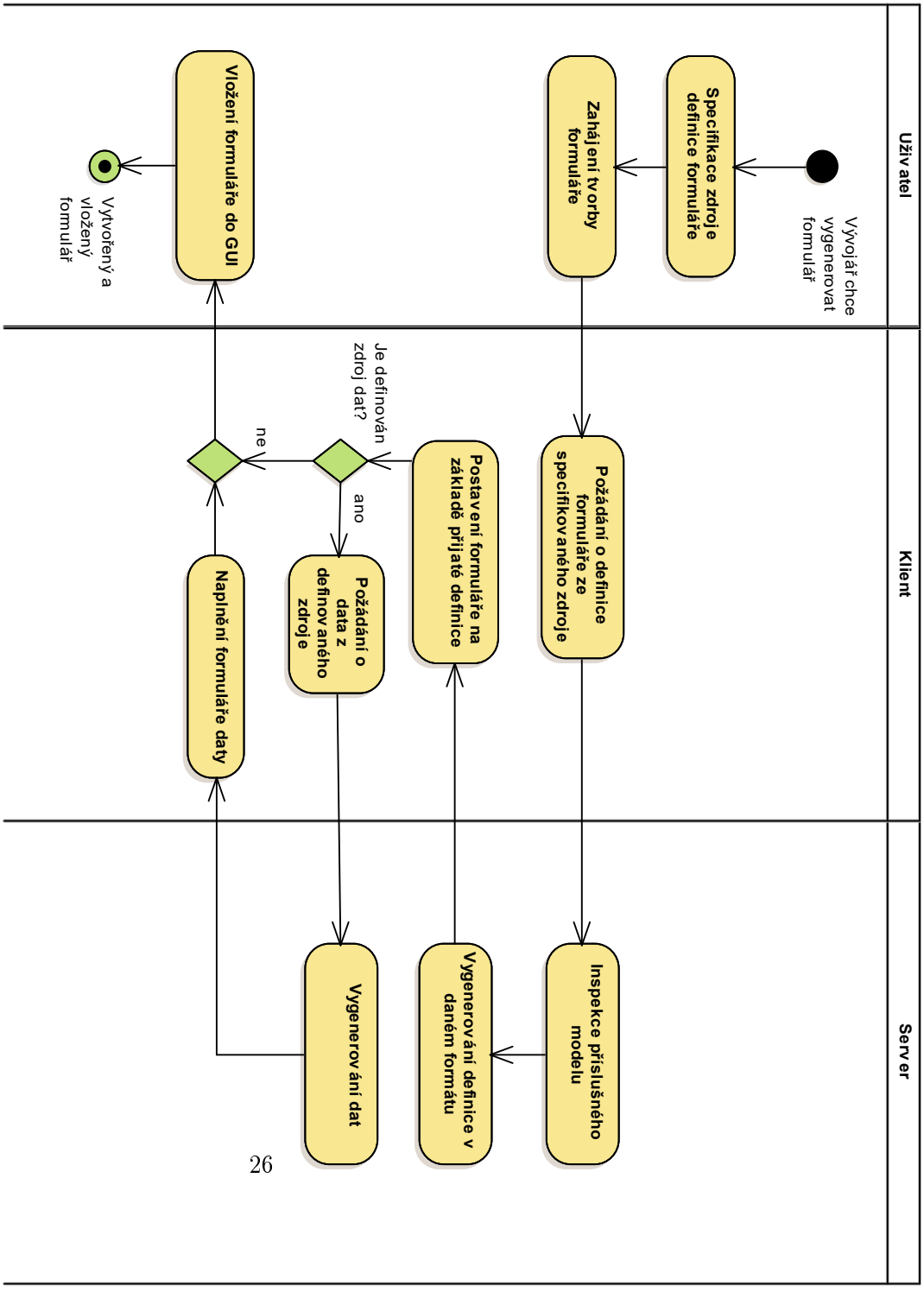
UML Unified Modeling Language

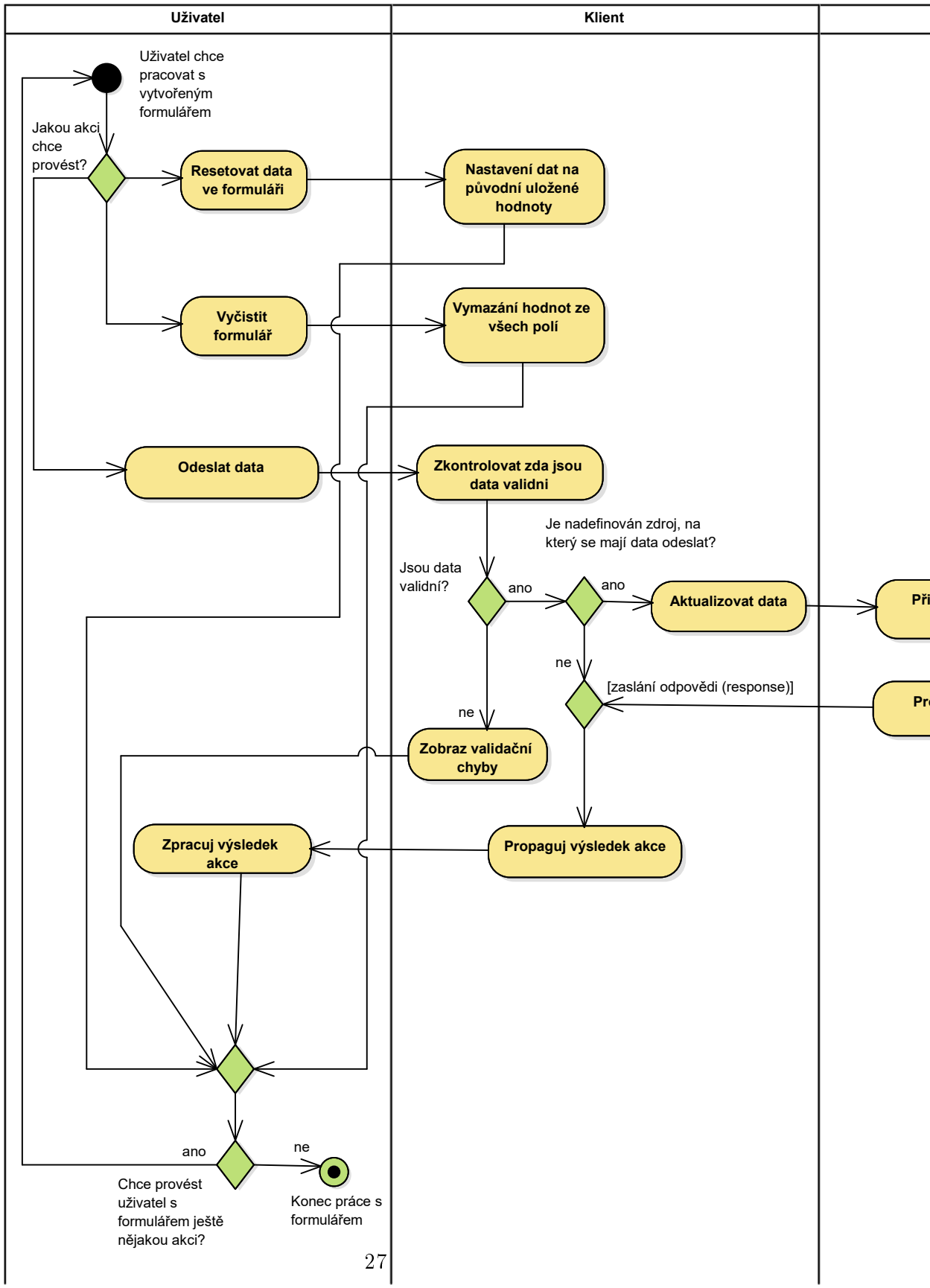
C# C Sharp

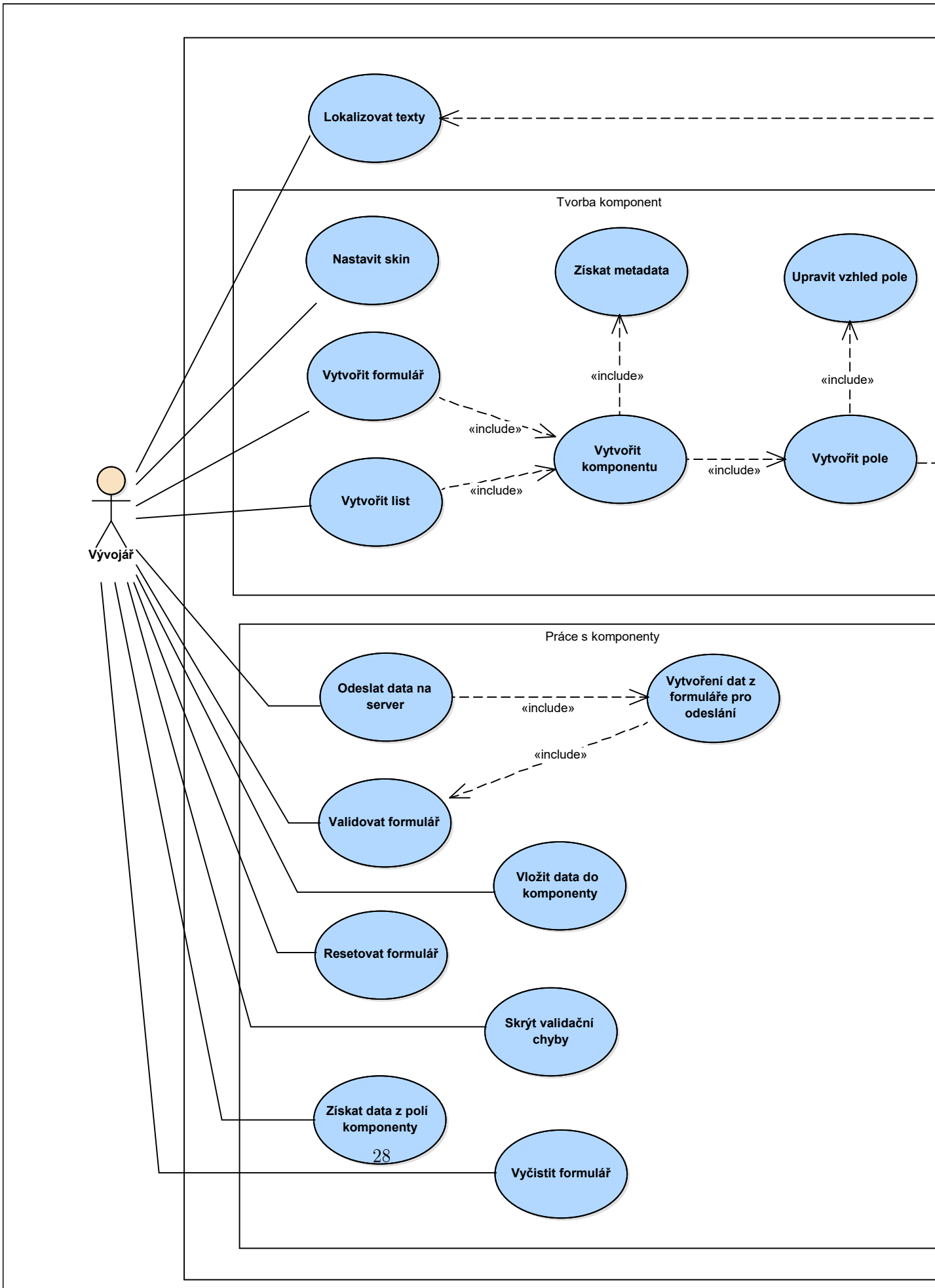
Příloha B

UML diagramy a obrázky

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


Uživatel
Klient
Server
Vývojář chce vygenerovat formulář
Specifikace zdroje definice formuláře
Zahájení tvorby formuláře
Požádání o definice formuláře ze specifikovaného zdroje
Inspekce příslušného modelu
Vygenerování definice v daném formátu
Postavení formuláře na základě přijaté definice
Je definován zdroj dat?
ano
ne
Požádání o data z definovaného zdroje
Vygenerování dat
Naplnění formuláře daty
Vložení formuláře do GUI
Vytvořený a vložený formulář


Uživatel
Klient
Uživatel chce pracovat s vytvořeným formulářem
Jakou akci chce provést?
Resetovat data ve formuláři
Nastavení dat na původní uložené hodnoty
Vyčistit formulář
Vymazání hodnot ze všech polí
Odeslat data
Zkontrolovat zda jsou data validni
Je nadefinován zdroj, na který se mají data odeslat?
Jsou data validní?
ano
ano
Aktualizovat data
ne
[zaslání odpovědi (response)]
ne
Zobraz validační chyby
Zpracuj výsledek akce
Propaguj výsledek akce
ano
ne
Chce provést uživatel s formulářem ještě nějakou akci?
Konec práce s formulářem

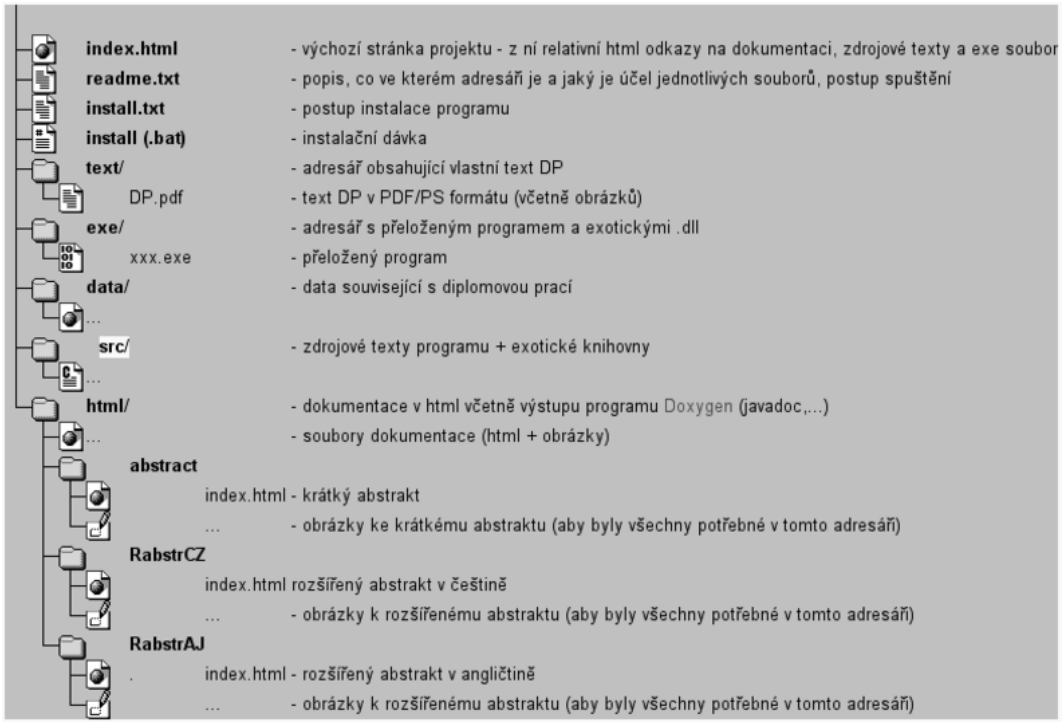

Lokalizovat texty
Tvorba komponent
Nastavit skin
Získat metadata
Upravit vzhled pole
Vytvořit formulář
«include»
«include»
«include»
Vytvořit komponentu
Vytvořit pole
«include»
Vytvořit list
«include»
Vývojář
Práce s komponenty
Odeslat data na server
«include»
Vytvoření dat z formuláře pro odeslání
«include»
Validovat formulář
Vložit data do komponenty
Resetovat formulář
Skrýt validační chyby
Získat data z polí komponenty
Vyčistit formulář

Příloha C

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [**?**]):



Obrázek C.1: Seznam přiloženého CD — příklad

Na GNU/Linuxu si strukturu přiloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případne index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.