ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Matyáš** Jméno: **Pavel** Osobní číslo: **420122**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Automatické generování uživatelského rozhraní na mobilních platformách s ohledem na kontext využívající klasifikaci UI

Název diplomové práce anglicky:

Pokyny pro vypracování:

Analyzujte a rozšiřte možnosti kontextově generovaných uživatelských rozhraních na mobilních platformách, které reflektují způsob a význam jednotlivých komponent UI v různých situacích [1] [3]. Ověřte možnosti automatické detekce kontextu na základě údajů od zařízeních, která se v blízkosti uživatele (nearby devices) a spadají do kategorie IoT včetně využití těchto principů v oblasti bezpečnosti. Základní myšlenky jsou v přiložené literatuře [2]. Situaci demonstrujte na vhodných prototypech, které budou nasazeny v reálném produkčním prostředí.

Seznam doporučené literatury:

"[1] M. Tomasek and T. Cerny. On web services ui in user interface generation in standalone applications. In Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems, RACS, pages 363{368, New York, NY, USA, 2015. ACM.
[2] M. Trnka, M. Tomasek, and T. Cerny. Context-Aware Security Using Internet of Things Devices, pages 706{713. Springer Singapore, Singapore, 2017.
[3] M. Tomasek and T. Cerny. Automated User Interface Generation Involving Field Classification. In Software Networking Journal, pages 54{76, January 2017}

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Martin Tomášek, katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **27.11.2017** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2019**

[redacted]
Ing. Martin Tomášek
podpis vedoucí(ho) práce

[redacted]
podpis vedoucí(ho) ústavu/katedry

[redacted]
prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

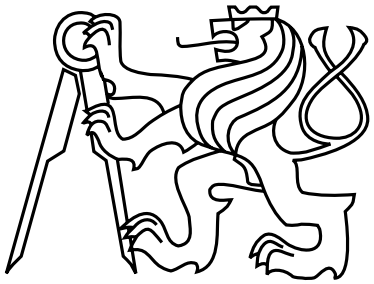
III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

3.5.2018
Datum převzetí zadání

[redacted]
Podpis studenta

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Automatické generování uživatelského rozhraní na mobilních platformách s ohledem na kontext využívající klasifikaci UI

Bc. Pavel Matyáš

Vedoucí práce: Ing. Martin Tomášek

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

20. května 2018

Poděkování

Především bych rád poděkoval vedoucímu mé diplomové práce Ing. Martinu Tomáškovi za jeho čas, zkušenosti a dobré rady, které mi při psaní práce poskytl. Poděkování patří také celé mé rodině a všem, co mě podporovali během studia.

Prohlášení

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

V Praze dne 21. 5. 2018 .....

Abstract

The goal of this work is to extend the process of automatic user interface generation for mobile platforms to use information about user context. UI components consists of parts where each part is less or more important based on the actual situation. For example in forms, each field has its purpose and severity. Based on these properties, each field can be evaluated and classified what parameters it should have in UI component in terms of visibility, validation rules, layout and so on. During this classification process the user context should be considered. In case of mobile devices informations like battery capacity, type and strength of Internet connection, location or nearby devices can be used. The work focuses primarily on the possibilities of collectiong context data its usage in component classification and its integration in process of automatic user interface generation for mobile platforms.

Abstrakt

Cílem této práce je rozšířit způsob automatického generování uživatelského rozhraní pro mobilní platformy o využití infomací o kontextu uživatele. Komponenty UI se skládají z částí, přičemž každá část je v různých situacích jinak důležitá. Například ve formulářích mají jednotlivá pole svůj účel a informace z nich jsou více nebo méně potřebné. Každé pole je na základě těchto vlastností možné ohodnotit a klasifkovat, jaké parametry by mělo v komponentě mít z hlediska viditelnosti, validačních pravidel, rozložení a podobně. Při procesu klasifikace by měl být zohledněn kontext uživatele. V případě mobilních zařízení lze využít informací jako je množství baterie, typ a síla internetového připojení, poloha nebo blízká okolní zařízení. Práce se primárně zaměřuje na možnosti sběru kontextových dat, jejich využití pro klasifikaci komponent a její integraci do procesu automatického generování komponent uživatelského rozhraní pro mobilní platformy.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato diplomová práce se zabývá analýzou, návrhem a implementací způsobu využití kontextu v oblasti automatického generování uživatelských rozhraní pro mobilní platformy. Součástí řešení je framework pro sběr kontextových dat z vlastního i blízkých okolních zařízení, serverová aplikace, která tato data ukládá, a aplikace umožňující správu rozhraní klientských aplikací a nastavení procesu využití kontextových dat. Serverové aplikace jsou pro účely testování řešení na reálném mobilním zařízení nasazeny v produkčním prostředí. Všechny vytvořené aplikace jsou prototypy.

První část práce se zabývá kontextovými daty, jejich popisem, možnostmi jejich sběru, dostupností a způsobem jejich využití v oblasti generování uživatelských rozhraní. Ve druhé části jsou analyzovány požadavky, které je potřeba pro dosažení cíle splnit, a na jejich základě je navrženo řešení. V následující kapitole je popsána vlastní implementace navrženého řešení. Po implementaci následuje testování, jehož součástí je i demonstrace funkcionality v praxi. Poslední část se věnuje procesu nasazení řešení.

V příloze A lze najít seznam použitých zkratek. Použité UML diagramy a obrázky jsou umístěny v příloze B, ukázky zdrojových kódů v příloze C. Nakonec lze v příloze D nalézt obsah CD, které obsahuje zdrojové kódy diplomové práce a vytvořených řešení.

1.1 Motivace

V dnešní době je uživatelům dostupný nespočet různorodých mobilních aplikací [48]. Na konci roku 2017 jich bylo v obchodě s aplikacemi Google Play na Androidu více než tři miliony, na Apple App Store více než dva miliony. Každá z těchto aplikací se snaží být svým způsobem unikátní a poskytnout svému uživateli co nejlepší zážitek. S takovým počtem aplikací mají navíc uživatelé opravdu z čeho vybírat a i díky tomu se zvyšují jejich požadavky [58]. Jedním z nejdůležitějších vlastností aplikace je uživatelské rozhraní, které by mělo být navrženo tak, aby co nejvíce usnadňovalo práci se softwarem [47]. Pokud je navíc snadno ovladatelné, navrženo podle aktuálních trendů, s příjemným designem či jiným způsobem atraktivní, zvyšuje to šanci aplikace uspět na trhu.

Jeden ze způsobů, jak udělat uživatelské rozhraní aplikace atraktivní a lépe použitelné, je jeho personalizace, neboli schopnost rozhraní se co nejvíce přizpůsobit svému uživateli [43]. Čím více si bude aplikace s uživatelem rozumět, tím relevantnější formu rozhraní mu dokáže

nabídnout. Nástrojem personalizace může být například zohlednění kontextu uživatele. Zde by se mohly aplikace inspirovat interakcí mezi lidmi. Při komunikaci dokáží lidé zohledňovat situační informace, neboli kontext, a tím zvýšit vzájemné porozumění, usnadnit akci a podobně [31]. Například pokud člověk vidí, že je jeho protějšek ve stresu a chystá se vykonat neuváženou akci, může mu to zkusit rozmluvit. Lidé také přizpůsobují styl svého hovoru prostředí, ve kterém se nachází. Pokud by tedy aplikace měla informaci o tom, že je člověk ve stresu, mohla by například zobrazit při důležité akci varovaní, zda si je uživatel provedením skutečně jistý. Ve druhém případě se uživatel může například nacházet v jedoucím automobilu a aplikace mu proto nabídne ne příliš rozptylující a snadno ovladatelnou verzi svého uživatelské rozhraní.

Využití kontextu v oblasti personalizace uživatelského rozhraní mi přislo jako velmi zajímavé téma. Ve své bakalářské práci [47] jsem se navíc věnoval interpretaci automaticky generovaného popisu komponent uživatelského rozhraní na mobilních platformách. Nabízí se tedy možnost do tohoto procesu generování popisu komponent integrovat mechanismus, který využije kontextová data k úpravě těchto definic. V návaznosti na úpravu procesu je vhodné rozšířit i prototypy, které vznikly v rámci bakalářské práce [47]. Tyto prototypy navíc poslouží pro demonstraci aktuální funkcionality.

Kapitola 2

Popis problému a specifikace cíle

2.1 Popis problematiky

Nejprve je nutné vymezit, jaké informace pojem kontext zahrnuje. Definicí kontextu je mnoho, většina z nich popisuje kontext jako soubor aktuálních informací o poloze, prostředí, objektech a lidech v okolí a jejich změn. Některé definice jsou ještě obohaceny o informace jako je datum a čas, teplota, emoční stav nebo pozornost uživatele. Obecně by se tedy dalo říct, že kontext je všechno, co přímo či nepřímo ovlivňuje uživatele během jeho interakce s aplikací. V případě mobilních aplikací lze vytyčit informace jako je množství baterie, poloha, rychlost či orientace zařízení, okolní světelné či zvukové podmínky, kvalita a typ internetového připojení, ostatní okolní zařízení a obdobně. Aplikaci, která tyto informace zohledňuje, například ve svém uživatelském rozhraní, nazýváme context-aware[31].

2.1.1 Zdroje kontextových informací

Aby aplikace mohla kontext využít ve svůj prospěch, musí informace nejdříve získat. Naivním řešením by bylo požádat uživatele, aby informace manuálně poskytnul. To však už na první pohled není správné řešení, neboť by se tím zbytečně zvýšila náročnost úkolu a tím by rozhraní v podstatě již nesplňovalo svůj primární účel, a to usnadňovat práci se systémem. Navíc by uživatelé nejspíše nevěděli, jaké informace jsou pro aplikaci relevantní. Sběr informací by tedy měl být automatický bez akce ze strany uživatele [31]. Naštěstí jsou mobilní zařízení dnes vybavena různými druhy vestavěných senzorů a technologií, které zprostředkovávají přístup k mnoha ze zmiňovaných použitelných informací.

Senzory primárně slouží pro měření okolních podmínek, v jakých se zařízení nachází. Seznam senzorů a poskytovaných dat se může lišit platforma od platformy, pro ilustraci je uvedena platforma Android. Zde můžeme senzory rozdělit do tří hlavních kategorií [26]:

1. **Senzory pohybu** měří například akceleraci zařízení. Patří zde akcelerometr, gyroskop či senzor gravitace.
2. **Senzory prostředí** měří parametry prostředí jako je množství osvětlení, teplotu vzduchu nebo tlak.
3. **Senzory polohy** zjišťují fyzickou polohu zařízení, tedy například jeho orientaci.

Kromě senzorů mají vývojáři možnost zjistit například stav baterie mobilního zařízení, tj. aktuální kapacitu, zdraví, teplotu, napětí nebo jestli se aktuálně baterie nabíjí a jakým způsobem. Také lze získat stav připojení k internetu, tedy zda je zařízení připojeno na internet, a pokud ano, tak o jaký typ a kvalitu připojení se jedná - 3G, LTE, Wi-Fi. V neposlední řadě jsou vývojářům zpřístupněny informace o hardwaru a softwaru zařízení, jako je například verze systému či označení modelu. Výjimkou nejsou ani data týkající se rozlišení a velikosti displaye [3].

Pokud však vývojáře spíše zajímá oblast kontextových informací, která se týká okolí, lze získat například seznam okolních dostupných Wi-Fi sítí nebo může využít Bluetooth k vyhledání blízkých zařízení. Informace o poloze zařízení, rychlosti či nadmořské výšce lze obdržet z GPS [3].

V dnešní době jsou také stále častejší různé chytré hodinky, fitness náramky a další tzv. wearables neboli nositelná elektronika, která je dalším potenciálním zdrojem kontextových dat. Například srdeční tep uživatele může napovědět o emočním stavu či pozornosti uživatele. Wearables jsou často propojeny s mobilním telefonem, se kterým sdílejí nasbíraná data. Ta se většinou uchovávají v cloudových uložištích a je možné je využít k různým statistikám.

Všechna tato zmíněná data lze využít při personalizaci uživatelkého rozhraní aplikace. Tato práce se zabývá právě možnostmi sběru a využitím těchto dat, zejména dat o stavu zařízení a blízkých okolních zařízeních, v oblasti automatické tvorby uživatelského rozhraní a bezpečnosti aplikace. Způsoby získávání těchto dat budou popsány v analýze.

2.1.2 Dostupnost informací

Kontextových informací, které lze získat, je opravdu mnoho, nejsou však dostupné vždy a za všech okolností. Tato sekce obsahuje popis problémů a omezení týkajících se sběru těchto informací.

2.1.2.1 Dostupnost informací o svém zařízení

Nejprve se zaměříme na dostupnost infomací o zařízení, na kterém aplikace běží. Například v případě senzorů může nastat problém u starších zařízení, které nemusí obsahovat některý ze senzorů nebo používají zastaralou verzi SDK, která vývojářům přístup k datům neumožňuje [26]. Jelikož mnohé z dostupných informací jsou považovány za citlivé, nelze je jen tak poskystovat každé aplikaci. Z tohoto důvodu je na většině mobilních platformách zavedeno řízení přístupu k těmto informacím [45]. Konkrétně Android platforma podmiňuje přístup k datům právy, které uživatel potvrzuje při instalaci aplikace. Navíc od Androidu 6.0 je přístup k některým informacím považován za natolik citlivý či nebezpečný, že musí být souhlas uživatele udělen explicitně při startu aplikace, přičemž uživatel může přístup i zamítnout [4]. Příkladem může být poloha zařízení nebo data z tělesných senzorů.

Další problém nastává s daty spojenými s využitím sítě, bluetooth nebo GPS. Obecně nelze předpokládat, že je uživatel připojený k internetu, má zapnutý Bluetooth nebo určování polohy. Aplikace mohou zapnutí vyžadovat, ale volba je opět na uživateli.

Tato fakta komplikují automatický sběr kontextových dat. Velmi často jsou aplikace jednoduché a vyžadují například jen připojení k internetu. V momentě, kdy ale bude aplikace požadovat kromě toho, co skutečně potřebuje, ještě další přístupová práva či zapnutí sítí,

může být narušena uživatelova důvěra. Nebo to může na první pohled uživateli připadat nepřínosné. V případě, že většinu požadavků odmítne, jakýkoliv sběr a využití kontextových dat se stává téměř neuskutečnitelným.

2.1.2.2 Dostupnost informací o okolních zařízeních

Jak je vidět, problémy mohou nastat i v případě, že se chce aplikace dozvědět informace o zařízení, na kterém běží. Co se týče informací o dalších zařízeních v okolí, zde je situace ještě o něco neúprosnější. O zařízeních lze získat údaje například pomocí Bluetooth nebo Wi-Fi, dat je však velmi málo. Nejčastěji lze zjistit identifikátor zařízení, typicky jeho mac adresu a název zařízení. V případě Bluetooth vystavují některá zařízení pro ostatní svůj typ a v některých případech i jaké služby poskytují.

Například fitness náramky v některých případech poskytují služby umožňující přístup k počtu kroků, ušlé vzálenosti či k poslednímu měření srdečního tepu. Takových náramků je však na světě mnoho a jedna z mála věcí, čím se od sebe odlišují, je mobilní aplikace, se kterou jsou náramky propojené. Bohužel výrobci náramků neprodávají pouze náramek, ale i aplikaci, takže se velmi často stává, že služby poskytující užitečné informace jsou zpřístupněny pouze jejich aplikaci a neexistuje k nim žádné veřejně přístupné rozhraní.

2.1.3 Metody zpřístupnění informací z okolních zařízení

Získat data je tedy opět o něco více komplikované. Nicméně ne nemožné. Tato část práce se věnuje různým způsobům, jak data získat.

2.1.3.1 Reverzní inženýrství

Někteří vývojáři jsou schopni do zařízení vniknout a získat data přes Bluetooth metodou reverzního inženýrství [44]. Vývojář se nejdříve připojí k náramku přes Bluetooth, získá adresy služeb, které náramek poskytuje a poté experimentálně zjišťuje, jaká data a v jakém formátu do zařízení poslat, aby získal, co potřebuje. To ale není obecně aplikovatelné. Vývojáři, jež chtějí získávat kontextová data, se nebudou chtít zaměřovat pouze na jeden konrétní náramek a ani zřejmě nebudou provádět tuto proceduru pro každý náramek na trhu, nemluvě o tom, že to u některých zařízení nebude možné.

2.1.3.2 Google Fit API

Daleko slibnější řešení je Fit API od společnosti Google [14]. To se dělí na několik částí. V následujícím seznamu jsou popsány nejdůležitejší z nich.

Sensors API slibuje přístup k tzv. raw datům jak ze senzorů zařízení, na kterém aplikace běží, tak z jeho spárovaných wearable zařízení. Toto API slouží spíše k manuálnímu jednorázovému přístupu k datům. Pokud má uživatel zájem o automatické získávání dat v intervalech, měl by využít následující API.

Recording API umožňuje vývojáři definovat posluchače pro konkrétní typ dat. Data se poté sbírají na pozadí s definovanou periodicitou.

Bluetooth LE API poskytuje možnost hledat a získávat data z ostatních zařízení. Tato zařízení musí podporovat Bluetooth Low Energy (zkráceně BLE), které API využívá k vyhledání zdrojů dat zvoleného typu v okolí zařízení, na kterém aplikace běží. Aby bylo zařízení identifikovatelné jako zdroj dat, musí se takto i prezentovat, a to podle standardního GATT profilu [12]. V praxi to znamená, že pokud chce vývojář získat srdeční tep, BLE API nalezne pouze zařízení, která se prezentují jako měřič srdečního tepu. Může se tedy stát, že periferní zařízení umí měřit danou hodnotu, ale nepředstavují se jako zdroje této hodnoty. V takovém případě jej BLE API nenalezne. Pro takové situace umožňuje Fit API definovat vlastní zdroj dat tím, že se na straně periferního zařízení zaregistruje služba sloužící jako softwarový senzor, přes který lze data zpřístupnit [15]. To je však možné jen na wearables s operačním systémem Android.

Fit REST API obsahuje historická či dlouhodobější data. API je zabezpečené, stejně jak je tomu u ostatních API od Google pomocí OAuth 2.0 protokolu. Toto RESTové API umožňuje vývojářům získávat, přidávat nebo měnit data v tzv. fitness storu, což je centrální repozitář, do kterého se ukládají data z různých aplikací a zařízení.

Nevýhodou FIT Api je, že zatím spolupracuje jen s omezeným množstvím výrobců [14]. Aplikací, které umožňují spárování s aplikací Google Fit, a tedy jsou jejich data přinejměnším dostupná skrze zmiňované REST API, ale stále přibývá. Dalším problémem, který už byl naznačen u BLE API je, že mnoho periferních zařízení nezpřístupňuje své datové služby pod standardními označeními specifikovanými v GATT profilu [12]. Nepochybnou komplikací jsou také přístupová práva k datům, která jsou v případě dat o zdravotním stavu a návycích uživatele velmi přísná, některá dokonce Google zprostředkovává jen vybraným vývojářům na požádání [16].

2.1.3.3 Webová API

Někteří z výrobců umožňují, podobně jako dříve popsané Fit REST API, získat již dříve nasbíraná data z webového API. Takovým výrobcem je například Fitbit [11]. Rozhraní bývají zabezpečeny protokolem OAuth 2.0 a mají často omezený počet volání za danou dobu. Zmiňované Fitbit Web API poskytuje navíc vývojářům možnost implementovat tzv. Subscribtions API, které aplikaci notifikuje vždy, když mají uživatelé přístupná nová data, čímž by se teoreticky dala u těchto konkrétních náramků snížit doba mezi skutečným měřením a získáním dat v aplikaci.

2.1.3.4 Další možnosti zpřístupnění dat

Dalším řešením, jak dostat data z přidružených zařízení je mít na straně zařízení službu, či aplikaci, která bude data vysílat. Tento způsob byl již nastíněn u Google Fit BLE API, který

ale předpokládá, že na obou stranách beží operační systém Android. To ale není pravidlem, například Samsung na svých zařízeních provozuje mobilní operační systém Tizen [29]. Princip však zůstává stejný, na blízkém zařízení běží Tizen aplikace, která data zpřístupňuje.

Neexistuje tedy zatím univerzální způsob, kterým by se dala data ze zařízení získat, a to hlavně kvůli touze po exkluzivitě výrobců. Jelikož uživatelé vlastní různé typy chytrých hodinek a jiných zařízení, není lehké využít data z těchto zařízení k personalizaci uživatelského rozhraní. V mnoha případech by si navíc musel uživatel nainstalovat do svého periferního zařízení další aplikaci, což se opět může rozhodnout neudělat. Ačkoliv by byly informace z periferních zařízení k procesu personalizace rozhraní velmi užitečné, pravděpodobnost, že se k nim vývojáři bezproblémově dostanou, je velmi malá.

2.1.4 Možnosti využití dat

Hlavním cílem této práce je analyzovat způsoby personalizace uživatelského rozhraní za pomoci kontextových dat, díky čemuž by rozhraní mělo být pro uživatele více přívětivé a použitelné. Použitelnost označuje kvalitu interakce uživatele s aplikací, přičemž kvalita interakce se hodnotí podle různých kritérií, jako je například doba, kterou uživatel potřebuje k dokončení úkolu, nebo počet chyb, které přitom udělal. Jinými slovy použitelnost je kvalitativním atributem rozhraní udávající, jak jednoduché je rozhraní používat [38]. Použitelnost je jedno z témat, kterými se zabývá disciplína Human-computer interaction (HCI) neboli interakce člověka s počítačem [17]. Další oblasti, které tento obor zkoumá, jsou efektivita, design a implementace systémů z hlediska uživatele. Také z hlediska uživatele vyhodnocuje jednotlivé aspekty sytémového rozhraní. Cílem HCI je zlepšení interakce mezi uživatelem a systémem tím, že se systém přizpůsobí uživatelským potřebám.

Jedním z hlavních předmětů rozhraní by tedy měl být uživatel a jeho požadavky. Návrh rozhraní, který se zaměřuje primárně na uživatele, se nazývá User Centered Desing (zkráceně UCD). Tento způsob návrhu uživatelského rozhraní, kdy jsou do středu stavěny potřeby uživatele, dal prostor ke vzniku různých heuristik či pravidel, které by rozhraní měly splňovat, pokud chtějí být pro uživatele atraktivní, přístupné a uživatelsky přívětivé [17]. Asi nejznámější heuristikou je heuristika Jakoba Nielsena obsahující 10 základních pravidel, které by každé UI mělo splňovat [60]. Pro tyto účely jsou důležitá následující tři pravidla:

1. **Flexibilita a efektivita používání.** Uživateli by mělo být umožněno u častějších akcí v aplikaci urychlit proces tím, že od nich bude vyžadováno méně interakce. To znamená, že by měl systém poskytovat klávesové zkratky, makra, funkční klávesy apod.

2. **Estetický a minimalistický design.** Rozhraní by mělo obsahovat jen to, co je skutečně nutné k dokončení aktuálního úkolu.

3. **Prevence chyb.** Rozhraní by mělo, pokud je to možné, varovat uživatele před potenciální chybou.

A právě k dosažení těchto třech cílů lze využít kontextová data. V případě prvního pravidla bychom mohli využít kontext následujícím způsobem. Mějme příklad, uživatel dělá běžnou každodenní akci ve svém internetovém bankovnictví. Jak už bývá u takových systémů zvykem, při provedení akce potřebují ověřit uživatele například autorizační SMS zprávou. S

využitím kontextu bychom mohli uživateli umožnit tento autorizační krok přeskočit a tím mu umožnit efektivnější použití systému. Systém by například mohl ověřit, zda se uživatel nachází ve stejném kontextu, jako když prováděl akci naposledy, to znamená, má kolem sebe stejná zařízení, provádí to ve stejnou denní dobu, je připojen na stejné Wi-Fi síti atp. Pokud by se uživatel nacházel ve stejném či podobném kontextu, mohl by systém autorizační část vynechat [58].

Využití kontextových informací v případě druhého pravidla opět demonstrujeme na příkladě. Uživatel si potřebuje objednat jízdenku. Dopravní společnost ale aktuálně provádí sběr informací o cestujících, a proto ve formuláři pro objednání jízdenky vyžaduje pohlaví, věk a vzdělání uživatele. Jelikož má ale uživatel na svém mobilním zařízení baterii téměř v kritickém stavu, nemá příliš času tato pole vyplňovat a dopravní společnost tak může potenciálně přijít o zákazníka. Z tohoto důvodu využije dopravní společnost právě této kontextové informace získané ze senzoru baterie a tato pole v případě tohoto konkrétního uživatele nebude vyžadovat, či je úplně vynechá, čímž se zajistí minimalistický design nutný ke splnění úkolu.

V případě prevence chyb se může jednat o člověka, který se chystá koupit velmi drahou položku v eshopu. Systém však detekuje díky biometrickému senzoru z chytrých hodinek, které má uživatel na zápěstí, že má uživatel vysoký srdeční tep, z čehož usoudí, že může být ve stresu. Z tohoto důvodu mu aplikace při kliknutí na tlačítko koupit nabídne varovný dialog obsahující ujištění, zda si je uživatel svou akcí jistý, čímž ho může ochránit od případné chyby.

2.1.5 Použití kontextových dat při generování UI

Tato sekce popisuje způsoby, jak zakomponovat nasbíraná kontextová data do již existujícího procesu generování užívatelského rozhraní, který byl předmětem mé bakalářské práce. Tento způsob je stručně popsán v následují podsekci.

2.1.5.1 Dosavadní způsob generování UI

Tato práce je založená na mé bakalářské práci, která se zabývala vytvořením prototypů frameworků pro mobilní platformy Android a Windows Phone, jež interpretují JSON data popisující uživatelské rozhraní a vytváří z těchto dat UI komponenty, konkrétně formuláře, tabulky a listy. Definice komponenty obsahuje její název, rozložení (jednosloupcové či dvousloupcové) a seznam polí, které se v ní vyskytují. U každého pole může být uveden jeho label, widget, kterým by pole mělo být reprezentováno, validační pravidla, která musí pole splňovat a případně i výčet možností, kterých může pole nabývat. Tato data jsou automaticky generována z modelu na serveru a zpřístupněna klientům pomocí RESTful webových služeb [57].

K vygenerování definice z modelu je využit framework AspectFaces [7], který používá reflexi k inspekci statických modelových tříd. Základním předmětem inspekce jsou atributy třídy a jejich datové typy. Lze dodat i další vlastnosti pomocí XML konfiguračních souborů nebo široké škály JPA i Hibernate anotací. V případě, že standardní anotace nestačí, framework umožňuje vytvořit anotaci vlastní [55].

Díky tomuto přístupu jsou mobilní aplikace, které používají zmíněné mobilní frameworky, schopny se dynamicky adaptovat na změny v modelu na serveru bez toho, aby se do nich muselo jakkoliv zasahovat. Změnou v modelu může být změna datového typu, přidání či odebrání polí nebo změna anotace, například maximální délky řetězce. Frameworky tak usnadňují udržovatelnost klientských aplikací. Zároveň je jednodušší i jejich implementace, neboť je většina vlastností komponent popsána již na serveru, jmenovitě layouty polí ve formuláři, pořadí polí nebo také jejich validace. Vývojář se tedy stará pouze o specifikaci URL adresy, odkud se popisná data získají, a potřebných parametrů. Poté komponentu může nastylovat a umístí ji tam, kam potřebuje [47]. Klientské aplikace je tedy nutné upravovat jen například při přidání či odebrání celých komponent nebo při změně designu komponent. To je výhodné i z hlediska aktualizací aplikací v obchodech s aplikacemi. Nová verze aplikace musí projít často důkladnou manuální kontrolou kvality, což celý proces aktualizace zpomaluje a v některých případech bývá nová verze i zamítnuta [40].

Menší nevýhodou implementace klientské strany je, že vývojář musí specifikovat pro každou komponentu její napojení na server, tedy endpointy webových služeb, skrze které je zpřístupněn popis komponenty ve formátu JSON. U komponent se vždy definuje endpoint modelu, který poskytuje samotný popis komponenty se specifikací polí (sloupců), validacemi či možnostmi, kterých může dané pole (sloupec) nabývat. Tento popis je jednotný pro formulář, tabulku i list. Druhý endpoint zprostředkovává data, kterými se má komponenta naplnit, přičemž formát dat je opět pro všechny tři typy stejný. U formulářů není třeba tento endpoint specifikovat, pokud není žádoucí formulář předvyplnit daty. Třetí endpoint se definuje pouze u formulářů a určuje, kam se budou data z formuláře odesílat. Dále musí vývojář specifikovat případné bezpečnostní či hlavičkové parametry, které se budou posílat při požadavku na server [57]. Pokud všechno vývojář naspecifikuje, výsledkem je nemalý XML soubor, který musí být přítomen v každé klientské aplikaci, která chce tento přístup generování komponent využívat. Ideálnějším řešením by bylo klientské strany od tototo souboru osvobodit a informace z něj centralizovat.

2.1.5.2 Integrace kontextových dat

Zmiňovaný koncept generování by měl být upraven a rozšířen tak, aby zohledňoval kontextová data a na základě nich vygeneroval relevantnější reprezentace komponent grafického rozhraní. Formuláře a tabulky (respektive listy v případě mobilních aplikací) se nachází v téměř každé aplikaci a slouží ke sběru informací od uživatele nebo k jejich zobrazení. Tyto komponenty se skládájí z menších celků - polí v případě formuláře, sloupců v případě tabulky. Pole mají svůj specifický účel a svou důležitost, což je určeno zpravidla danými business požadavky. Více důležitá pole jsou taková, bez kterých by nebylo možné dokončit požadovaný business úkol. Například při objednání letenky je nutné vyplnit odletové místo a cílovou destinaci. Méně důležitá pole často slouží pouze pro sběr informací o uživateli a nejsou pro splnění cíle podstatná. V příkladu s letenkami by to byl například checkbox k odběru novinek.

Méně důležitá pole by se za určitých podmínek, například když má uživatel málo baterie nebo malý display, teoreticky nemusela zobrazovat. Nebo by alespoň nemusela být vyžadována či validována. To by však znamenalo, že by muselo existovat mnoho různých verzí uživatelských rozhraní pro každou situaci, což by bylo implementačně velmi náročné a pravděpodobně by docházelo k mnoha duplicitám v kódu [43, 34]. Nicméně díky zmiňovaným

frameworkům je možné integrovat proces rozhodování, které pole má či nemá být vykresleno, vyžadováno či validováno přímo do procesu generování komponent. Data se vyfiltrují a upraví podle specifikovaných podmínek. Takto upravená data zpřístupní server klientským frameworkům opět skrze webové služby, kde se z dat vyrobí hotová, upravená verze komponenty a klientská strana se nemusí měnit [56].

Procesu generování UI musí být v průběhu známo, jak jsou pole důležitá a k jakému účelu slouží. Jelikož se jedná spíše o business záležitost, bylo by přívětivé, kdyby šlo toto nastavení provádět v uživatelském rozhraní, které by zvládl obsluhovat i člověk bez programátorských znalostí. Samotná filtrace polí vyžaduje popis podmínek od programátora a soubor kontextových informací, na základě kterých se podmínky vyhodnotí. Výhodné by tedy bylo, kdyby mezi serverem poskytujícím popis komponent a klientskou stranou existoval mezičlánek, který by zodpovídal za správu komponent v aplikaci a jejich propojení s klientem i serverem, čímž by se vyřešil i problém s XML souborem na klientské straně zmiňovaný dříve v popisu dosavadního generování UI. Mezičlánek by také spravoval informace o důležitosti jednotlivých polí komponent, strategie a podmínky jejich zobrazení.

2.2 Existující řešení

V této sekci jsou popsána existující řešení, která využívají podobných myšlenek, jako jsou popsány výše. Jedná se o řešení, která generují uživatelská rozhraní, využívají kontextu k jeho personalizaci nebo se snaží jakkoliv ulehčit tvorbu uživatelského rozhraní. Zároveň jsou zde uvedeny již zmiňované frameworky, které zajišťují aktuální proces dynamického generování uživatelského rozhraní na mobilních platformách, ze kterého tato práce vychází.

2.2.1 AspectFaces

Jak již bylo popsáno v dřívějších odstavcích této kapitoly, AspectFaces [7] provádí inspekci modelových tříd a vyrábí z nich popisy komponent uživatelského rozhraní, tedy na základě datového typu a anotací atributů v modelu určí, jaké se pro ně použijí widgety a jaké vlastnosti budou mít. Díky tomu, že je UI definované v modelu, dějí se změny pouze na jednom místě a jakákoliv úprava se automaticky promítne do zmiňovaného popisu komponenty. Poslední verze distribuována pod open source licencí LGPL v3 je 1.5.0.

2.2.2 AFRest

AFRest slouží primárně pro převod popisu UI komponent, který byl vygenerovaný na základě inspekce tříd knihovnou AspectFaces, do definic komponent nezávislých na platformě. Tyto definice jsou poskytovány ve formátu JSON klientským stranám pomocí RESTful webových služeb. Klientskou stranou může být jakákoliv platforma, která umožňuje zpracování JSONu a dynamické vytváření grafických prvků UI při běhu aplikace.

2.2.3 AFAndroid, AFWinPhone a AFSwinx

Mobilní frameworky AFAndroid a AFWinPhone jsou prototypy, které byly vytvořeny v rámci mé bakalářské práce. Hlavním cílem bylo rozšířit rodinu interpreterů popisů komponent

uživatelského rozhraní, které vznikají z modelů na serveru. Tato rodina v té době obsahovala framework AFSwinx, který zajišťoval generování UI v Java SE aplikacích využívajících grafickou knihovnu Swing. Všichni interpreteři využívají knihovnu AFRest uvedenou výše [57] s výjimkou AFWinPhone. Ten využívá verzi AFRest přepsanou do C#, neboť Windows Phone nepodporuje Javu, ve které je AFRest napsán, a ani import JAR souborů. AFAndroid je napsán jako nativní Android knihovna v Javě v kombinaci s Android SDK verze minimálně 11, která je využitelná na zařízeních s Androidem verze 2.3.4 Gingerbread a výše. AFWinPhone je jeho klon přepsaný v C# na platformu Windows Phone[47], primárně zaměřený a otestovaný na verzi 8.1. Bohužel Windows Phone v průběhu let ze zařízení téměř zcela vymizel. Podle statistik společnosti Statcounter je celosvětově už jen 0.55 % zařízení běžících na tomto operačním sytému, zatímco v roce 2015 jej vlastnilo zhruba pětkrát více zařízení a jeho tehdejší nová verze Windows Phone 10 vypadala, že bude populárnější a toto procento navýší [27]. Z důvodu nízkého zastoupení na trhu zařízení se tedy neplánuje Windows Phone verze nadále rozširovat.

2.2.4 Easy Admin

Easy Admin je nástoj, který dokáže vygenerovat vcelku atraktivní administraci k PHP backendu. Nástroji stačí dát seznam entit, ze kterých má administraci vytvořit. Podmínkou je, aby entity používaly Doctrine ORM, neobsahovaly složené primární klíče a nepoužívaly databázovou dědičnost. Další podmínkou je PHP framework Symfony verze 2.3 a výše, který musí aplikace používat. Vygenerovaná administrace umí základní CRUD operace nad entitami, full-textové vyhledávání, stránkování a jednoduché řazení. Navíc má plně responzivní design. Dnes je již standardním balíčkem PHP frameworku Symfony pod názvem EasyAdminBundle [9].

2.2.5 Vaadin

Java framework Vaadin od verze 7 obsahuje tzv. `FieldFactory`, což je implementace rozhraní, které dokáže z entit vygenerovat formulářová pole. Ve Vaadinu se obecně prvky uživatelského rozhraní definují na backendu v Javě. `FieldFactory` využívá pro generování JPA anotací. Proces je rekurzivní, dokáže tedy zpracovávat i vnořené entity. Pokud je v modelu definovaná anotace `@ManyToOne`, je pole defaultně reprezentováno jako select widget, `@OneToOne` nebo `@Embedded` vytvoří vnořený formulář atp. Widgety polí jsou určeny podle datových typů v entitě [30].

2.2.6 Článek - Context-aware Generation of User Interface Containers for Mobile Devices

Tento článek pojednává o využití kontextu při generování uživatelských rozhraní. Autoři dělí uživatelské rozhraní na části - kontejnery, které na sebe navazují a definují business úkol. Prvním úkolem je rozčlenit uživatelské rozhraní na tzv. task tree (strom úkolů) podle toho, v jakém úkolu hraje část UI svou roli. Úkoly se řadí do několika vrstev, které jsou ohodnoceny podle toho, kolik úkolů se v nich nachází, jakého typu úkoly jsou a jak na sebe úkoly navazují. Vznikne tedy několik různých konfigurací, na které jsou aplikovány heuristická pravidla na základě kontextu, což vybere nejlepší konfiguraci. Tato konfigurace je předložena

tzv. designeru a výsledkem je abstraktní popis kontejneru uživatelského rozhraní, kterému se nastaví v posledním kroku uživatelsky přívětivá vizuální stránka [46].

2.2.7 Článek - Context-aware User Interface Framework for Mobile Applications

Článek informuje čtenáře o možnosti využití jazyku XUL (XML User Interface Language) k upravení uživatelského rozhraní na základě kontextu. XUL je upravená verze XML vytvořená nadací Mozilla Foundation speciálně pro popis uživatelského rozhraní. Aby se tento popis proměnil v uživatelské rozhraní, musí být interpretován na platformě. Autoři článku využili XUL k definici vlastních tagů popisující rozhraní v různých situacích. Mezi kontextové informace, které autoři využívají, patří pozice a aktivita uživatele. Například pokud uživatel používá aplikaci při chůzi, nadefinují UI určité CSS vlastnosti, v tomto případě například větší písmo. Další vlastnosti mohou být viditelnost či velikost ovládacích prvků aplikace [34].

2.2.8 AWARE

Aware [8] je Android a iOS framework umožňující získávat, zaznamenávat nebo sdílet kontextové informace ze senzorů mobilního zařízení. Mezi podporované senzory patří akcelerometr, baterie, senzor gravitace, gyroskop, magnetometr, rotace zařízení, měřič okolního osvětlení, velikost obrazovky atd. Také umí získat informace o nalezených zařízeních z Wi-Fi či Bluetooth. Pokud by uživateli nestačila základní funkcionalita, lze přidat některý z předpřipravených pluginů. Jeden z těchto pluginů je také schopen získávat data z Fitbit náramků. Bohužel nejedná se o data v reálném čase, nýbrž o již předem nasbíraná data ze serverů Fitbitu dostupná přes Fitbit API [11]. Aktuální verze vyžaduje minimálně Android 4.4 Kitkat a výše, v případě iOS verzi 8.0 a výše. Velkou výhodou tohoto frameworku je, že je open-source.

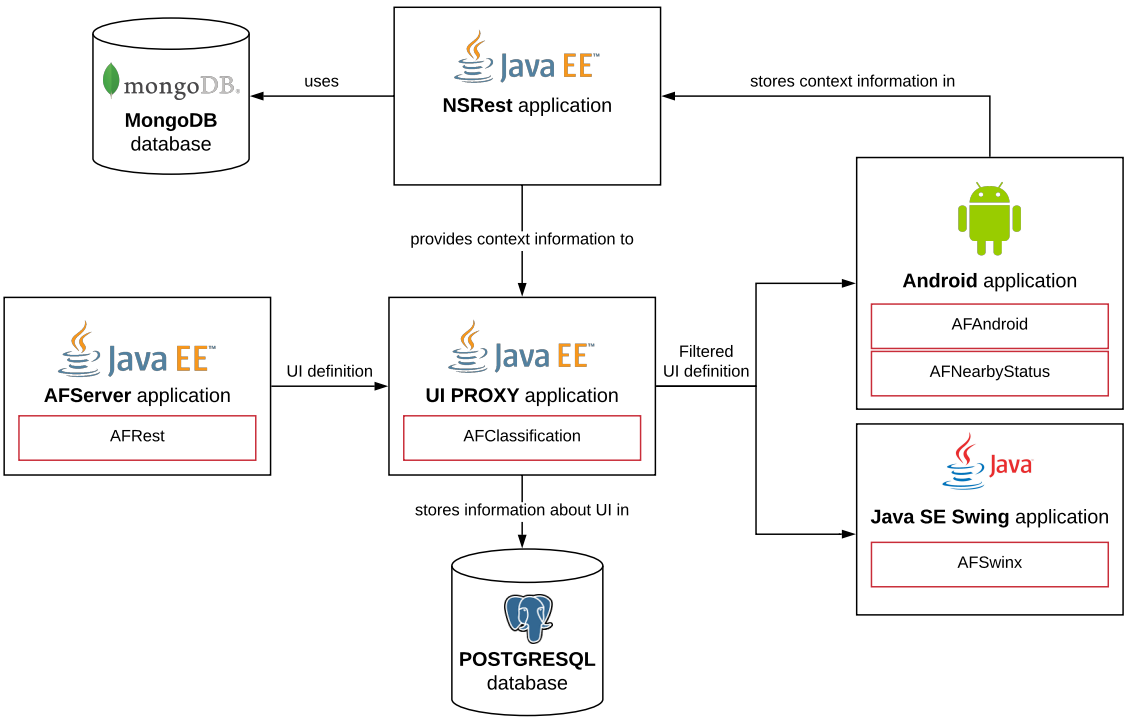
2.3 Cíle práce

Cílem této práce je rozšířit aktuální koncept generování uživatelského rozhraní na mobilních zařízeních z modelu na serveru [47], který je zprostředkován za pomoci AFAndroid, AFRest a AspectFaces, o využití kontextu, zejména informací o blízkých okolních zařízeních, pro vytvoření přívětivějšího uživatelského rozhraní. Tyto kontextové informace je potřeba nejdříve získat, součástí řešení tedy musí být i knihovna, která data ze zařízení sbírá. Nasbíraná data je třeba ukládat a zpřístupnit procesu generování UI, je tedy nutné vytvořit i aplikaci pro tyto účely. Vzhledem k charakteru již existujícího řešení, které generuje formuláře, tabulky a listy, je nejvhodnější způsob, jak UI upravit na základě zmiňovaných kontextových dat, ovlivnit viditelnost či validační pravidla jednotlivých částí generovaných komponent.

Proces úpravy je řešen až v momentě, kdy se vygeneruje popis komponenty ve formátu JSON [55] v aplikaci , která slouží jako mezičlánek mezi klientem a serverem. Úprava komponenty se řídí důležitostí a účelem jejích částí, přičemž správa těchto business vlastností

je v mezičlánku zpřístupněna pomocí uživatelského rozhraní. Tento mezičlánek také zodpovídá za správu podmínek a pravidel, které se při procesu úpravy využijí. Dalším cílem je centralizovat dříve popisovaný XML soubor s popisem endpointů webových služeb, které zprostředkovávají popisy komponent, její data nebo místo, kam se mají odesílat formuláře. Pro lepší představu architektury a komunikace navrhovaných aplikací byl vytvořen obrázek 2.1.

Je zřejmé, že celý proces od modelu na serveru ke komponentě UI na klientovi se změní, nicméně je žádoucí zachovat stávající funkcionalitu. Aby bylo možné tohoto dosáhnout, je potřeba aktuální řešení upravit, zejména klientské frameworky AFAndroid a AFSwinx a příslušné ukázkové aplikace. Z důvodů zmiňovaných výše nebude Windows Phone verze již podporována, proto se jí tato práce již dále nezabývá.



Obrázek 2.1: Hrubý návrh architektury a komunikace aplikací

Kapitola 3

Analýza

3.1 Funkční specifikace

V rámci této práce je rozšířen uvedený způsob generování uživatelského rozhraní o sběr a využití kontextových informací v oblasti personalizace jeho komponent. Toto rozšíření se týká hlavně frameworku AFAndroid, nicméně je potřeba upravit i stávající řešení pro Java SE aplikace.

Jak již bylo zmíněno, pro dosažení požadované funkcionality je potřeba vyvinout hned několik řešení, které spolupracují. Je nutné podoknout, že aplikace a frameworky, které jsou v rámci této práce vyvinuty, nejsou určeny k vydání, nýbrž se jedná o proof of concept. Prvním důležitým bodem v procesu je sběr kontextových informací. Ten musí probíhat automaticky a jelikož sběr dat probíhá současně s prací uživatele v aplikaci, je nutné, aby běžel na pozadí a neblokoval uživatelské rozhraní [49]. Musí být vytvořen Android framework, který toto umožní. Nasbíraná data je vhodné ukládat, aby se dala později využít v procesu generování komponent. Je nutné tedy vytvořit aplikaci, která kontextové informace přijme, uloží a poté je zprostředkuje dalším aplikacím. V neposlední řadě je potřeba vyvinout aplikaci, která slouží primárně jako správce komponent a korespondujících business požadavků, které se týkají důležitosti a účelu jednotlivých polí komponent. K účelu nadefinování těchto vlastností polí musí mít systém uživatelské rozhraní. Aplikace má zastávat roli proxy mezi klientem a serverem a musí disponovat funkcionalitou, která zpracuje požadavek na získání komponenty z klientské aplikace následujícím způsobem. Získá definice komponent ze serveru, aplikuje na ně zmiňovaná pravidla podle nadefinovaných podmínek, což vyústí v upravenou definici komponenty, kterou v této podobě naservíruje klientské aplikaci. Všechny funkční požadavky rozčleněny do skupin podle aplikace, které se týkají, jsou uvedeny v následujících seznamech položek.

3.1.1 Funkční požadavky frameworku na sběr dat

V následujícím seznamu jsou popsány funkční požadavky frameworku, který slouží ke sběru kontextových dat na Android zařízení. Jelikož tento framework má získávat hlavně informace o stavu zařízení a data o okolních zařízeních, je na něj dále v tomto textu odkazováno jako na **AFNearbyStatus**.

- Framework bude umožňovat sbírat následující data o zařízení a jeho stavu
  - informace o stavu baterie,
  - informace o poloze (tj. zeměpisná šířka, délka a nadmořská výška) a rychlosti pohybu zařízení,
  - informace o síťovém připojení (např. zda je uživatel připojený, typ sítě či její název),
  - hardwarové a softwarové informace o zařízení (např. mac adresa, verze systému nebo název zařízení).
- Framework bude umožňovat získat co nejvíce možných dat o blízkých zařízeních.
- Framework bude umožňovat získat informace o aktuálním stavu uživatele v aplikaci (např. na které obrazovce se nachází či jaký uživatel je nyní přihlášen).
- Framework bude umožňovat určit, jaká data se budou sbírat.
- Framework bude umožňovat z důvodu náročnosti sběru dat na baterii vypnout některé ze sběračů dat na základě kapacity baterie.
- Framework bude umožňovat spustit proces sběru kontextových informací na pozadí.
- Framework bude umožňovat spustit proces sběru dat, který se bude provádět automaticky s možností samostatného opakování.
- Framework bude umožňovat nadefinovat maximální dobu jednoho sběru informací.
- Framework bude umožňovat odeslat nasbíraná data na server.
- Framework by neměl zbytečně zasahovat do aplikace tj. neměl by například vyžadovat od uživatele zapnutí Bluetooth, Wi-Fi a podobně.

Důvody některých požadavků nemusí být na první pohled zřejmé. Jako první je zde požadavek týkající se možnosti vypnout sběr některých dat na základě kapacity baterie. Jelikož má získávání dat probíhat na pozadí současně s aplikací, do které je integrováno, zvýší se i náročnost aplikace na baterii. V momentě, kdy se bude aplikace současně dotazovat senzorů a využívat Bluetooth, GPS a Wi-Fi ke zjišťování informací, může baterie radikálně snižovat svou kapacitu. Největšími spotřebiteli baterie jsou podle [53] využití sítě, GPS a čtení ze senzorů, přičemž nejvíce baterie trpí, pokud se provádí operace na pozadí v době, kdy má zařízení spát.

Dalším požadavkem je možnost frameworku nadefinovat maximální dobu sběru informací. Sběr informací, zejména hledání okolních zařízení, není instatní, ale trvá určitou dobu. Například objevování Bluetooth zařízení podle Android dokumentace [2] trvá od svého startu dvanáct vteřin. Uživateli frameworku by však měla být nabídnuta možnost tento proces ukončit i dříve. Z toho důvodu je možnost definice maximální doby sběru dat požadována.

Jelikož se kontext uživatele bude časem s velkou pravděpodobností měnit, je třeba kontextová data při používání aplikace zjišťovat opakovaně. Aby uživatel nemusel opakované měření spouštět vždy manuálně, měla by existovat možnost spustit proces s periodicitou, se kterou se bude měření automaticky opakovat.

Poslední požadavek, který potřebuje upřesnění, je, že framework má být schopen nasbírat co nejvíce informací o blízkých zařízeních. K tomuto účelu je využit Bluetooth a Wi-Fi adaptér. Využití Bluetooth je přímočaré, jednoduše se provede sken prostředí. Wifi adaptér je schopen získat například okolní Wi-Fi sítě, jejich názvy, mac adresy routerů a podobně. V případě, že je uživatel připojen na Wi-Fi síti, lze určit jeho IP adresu, ze které by teoreticky měl jít zjistit rozsah IP adres sítě. Tyto adresy by se daly kontaktovat a zjistit informace o zařízeních, které je mají přiděleny.

3.1.1.1 Způsob výpočtu rozsahu IP adres

IP adresa se skládá ze dvou částí - síťové adresy (network address) a hostitelské adresy (host address). Aby bylo možné tyto dvě části rozlišit, existuje maska podsítě, která určuje dělení. Pokud je na IP adresu a masku podsítě použita operace bitový AND, je výsledkem právě síťová adresa [28]. Síťová adresa je první IP adresou v dané síti, poslední adresou je broadcast. S použitím masky podsítě lze zjistit, kolik IP adres se mezi síťovou a broadcast adresou nachází. Výpočet je velmi jednoduchý, pokud je maska podsítě například 255.255.254.0 tak je rozsah ip adres roven $(2*256)-2$, neboť existuje 256 možností v rozsahu 255.255.254.0 až 255.255.254.255 a 256 možností v rozsahu 255.255.255.0 až 255.255.255.255. První a poslední adresa je však síťová adresa a broadcast a není možné, aby byly přiřazeny zařízení, proto mínus 2.

3.1.2 Funkční požadavky aplikace uchovávající data

- Aplikace bude přijímat kontextová data nasbíraná frameworkem AFNearbyStatus.
- Aplikace bude data uchovávat v databázi.
- Aplikace bude poskytovat data ostatním aplikacím ve formátu JSON.
- Aplikace bude k přijmu a zpřístupnění dat využívat RESTful webové služby.

Jelikož má aplikace využívat RESTful webových služeb a má výhradně sloužit k uchovávání dat o stavech zařízení a jejich blízkých zařízeních, získala označení **NSRest**.

Aby bylo možno identifikovat, která kontextová data patří k jakému zařízení, je potřeba ukládat k jednotlivým měřením i unikátní identifikátor zařízení. Nabízí se mac adresa, kterou má každé zařízení v síti. Je přiřazována výrobcem zařízení (respektive jeho síťového rozhraní) a měla by být unikátní. Existují však způsoby [37], kterými lze mac adresu zařízení změnit a tudíž unikátnost narušit. V případě Android zařízení, je mac adresa závislá na Wi-Fi nebo Bluetooth. Pokud není ani jedna služba zapnutá, mac adresu zařízení není možné zjistit. Využití mac adresy jako identifikátoru se tedy nedoporučuje. Jako alternativy Android nabízí sériové číslo nebo ANDROID_ID, což je 64-bitové číslo, které je vygenrováno při prvním zapnutí zařízení. Ani ANDROID_ID však není 100% spolehlivé, zejména na starých zařízeních, které toto číslo ještě nemají [33].

Označení zařízení unikátním identifikátorem je tedy problém. Pokud je žádoucí označit všechna zařízení pokud možno stejným typem identifikátoru, je nejvhodnější použít mac adresu. U tohoto typu aplikace, která vyžaduje minimálně připojení k internetu, aby si

mohla stáhnout potřebné definice komponent UI, se není třeba obávat, že bychom mac adresu nezískali. Samozřejmě to není ideální řešení, neboť jak bylo zmíněno, mac adresy zařízení lze manuálně změnit, nicméně tato práce se zabývá pouze proof of conceptem řešení a prozatím tedy identifikace zařízení pomocí mac adresy postačí.

3.1.3 Funkční požadavky proxy aplikace

Primárním cílem proxy aplikace je, aby klientské aplikace nekomunikovaly přímo se serverem, ale právě skrz tuto aplikaci. Důvodem je vytvořit možnost přenášená data upravit, aniž by se na klientské či serverové straně muselo něco zásadně měnit. Sekundárním cílem proxy aplikace je převzetí a centralizace specifikací napojení komponent na server, která je nyní řešena zmiňovaným XML souborem. Níže jsou vypsány všechny funkční požadavky na proxy aplikaci.

- Aplikace bude umožňovat správu komponent klientských aplikací, u kterých půjde nadefinovat napojení na server poskytující popisy komponent.
- Aplikace bude umožňovat získat definici či data komponenty ze serveru, který tato data poskytuje nebo zde odeslat data z formuláře.
- Aplikace bude umožňovat správu obrazovek, které se nachází v klientské aplikaci.
- Aplikace bude umožňovat přiřadit k obrazovkám komponenty, které se v nich mají nacházet.
- Aplikace bude umožňovat správu business případů a jejich fází, které mohou v klientské aplikaci nastat.
- Aplikace bude umožňovat přiřadit business fázím jednotlivé obrazovky, které jsou pro danou situaci relevantní.
- Aplikace bude umožňovat nadefinovat polím komponenty, které se nachází v obrazovce, důležitost a účel ve vztahu ke konkrétnímu business případu a fázi.
- Aplikace bude umožňovat rozpoznat podle obrazovky a komponenty z ní, kterou klientská aplikace žádá, jakého business případu a fáze se požadavek týká.
- Aplikace bude umožňovat určit dané business fázi podmínky a algoritmus, na základě kterých se vyhodnotí a provedou úpravy v komponentě.
- Aplikace bude umožňovat upravit vlastnosti komponenty na základě zmiňovaných podmínek a algoritmu business fáze.
- Aplikace bude poskytovat definice obrazovek včetně komponent, které se v nich nachází, klientským aplikacím skrze RESTful webové služby.
- Aplikace bude poskytovat klientské aplikaci definici menu, která obsahuje URL adresy endpointů, na kterých se nachází definice jednotlivých obrazovek skrze RESTful webové služby.

Téměř všechny body, vyjma bodů týkajících se samotné úpravy komponenty včetně definice podmínek a algoritmu, na základě kterých se úprava provede, by mělo být možné provádět v uživatelském rozhraní.

3.2 Popis architektury a komunikace

V původním řešení probíhala komunikace pouze mezi klientem a serverem. Klient byl jeden z interpreterů definic uživatelského rozhraní, tedy Android aplikace využívající AFAndroid framework nebo Java SE aplikace používající AFSwinx [47].

K ilustraci popisu architektury a komunikace jsou použity UML diagramy. Unified Modeling Language (UML) je standard, podle kterého se kreslí diagramy systémů pomáhající při jejich návrhu, implementaci i údržbě. Výhodou UML je, že je srozumitelný i napříč jazyky. [35]. Diagramů, které UML nabízí, je mnoho, každý z nich slouží k jinému účelu. Podle něj lze diagramy rozdělit do dvou (resp. tří) skupin: [51]:

- **Diagramy struktury** popisují strukturu systému, z čeho a jak je systém sestaven.
- **Diagramy chování** popisují, jak systém funguje a jak se chová v různých situacích.
- **Diagramy interakce** jsou podskupinou diagramu chování a jsou specializovány na interakce mezi jednotlivými částmi systému.

K objasnění, jaké softwarové komponenty byly použity v původním řešení a jak byly propojeny, je použit diagram nasazení B.1. Jsou zde vidět dvě komponenty, klient a server. Serverová Java EE aplikace, která používá framework AFRest a Aspectfaces, je schopna vygenerovat definice komponent. Klientem je Android aplikace využívající mobilní framework AFAndroid, jež je schopna definice interpretovat a vytvořit z nich komponenty uživatelského rozhraní. K přenosu definic uživatelského rozhraní směrem od serveru ke klientovi, jak je v diagramu také naznačeno, je použit HTTP protokol.

Nyní je cílem mezi klienta a server vložit proxy aplikaci, která zprostředkuje komunikaci mezi klientem a serverem. Veškerá data, která si původně vyměňovali klient a server přímo, nyní půjdou skrz tuto aplikaci, což umožní v proxy data předtím, než se dostanou ke klientovi, modifikovat. Aktuální rozložení lze vidět na novém obecném diagramu nasazení B.2.

U Android klienta je vidět nová komponenta AFNearbyStatus, která má sbírat kontextová data ze zařízení a jeho okolí. Také je zde zobrazena aplikace NSRest sbírající kontextová data, které Android aplikace získá. Tato data poskytuje proxy aplikaci, která je využije při úpravě popisu komponent. Veškerá komunikace mezi systémy je zprostředkovaná pomocí HTTP protokolu. Je nutno podotknout, že diagram B.2 je použit pouze pro ilustraci problému a nemusí nutně odpovídat reálnému nasazení. Například se mohou všechny tři serverové aplikace nacházet na jednom serveru.

3.2.1 Proces sběru dat

Nejprve bude popsán způsob, jakým jsou získávána a ukládána kontextová data. Jak již bylo dříve zmíněno sběr kontextových dat probíhá na straně klienta a je zprostředkován pomocí frameworku AFNearbyStatus. Veškeré požadavky na sběr dat jsou uvedeny v sekci Funkční požadavky frameworku pro sběr dat.

3.2.1.1 Struktura kontextových dat

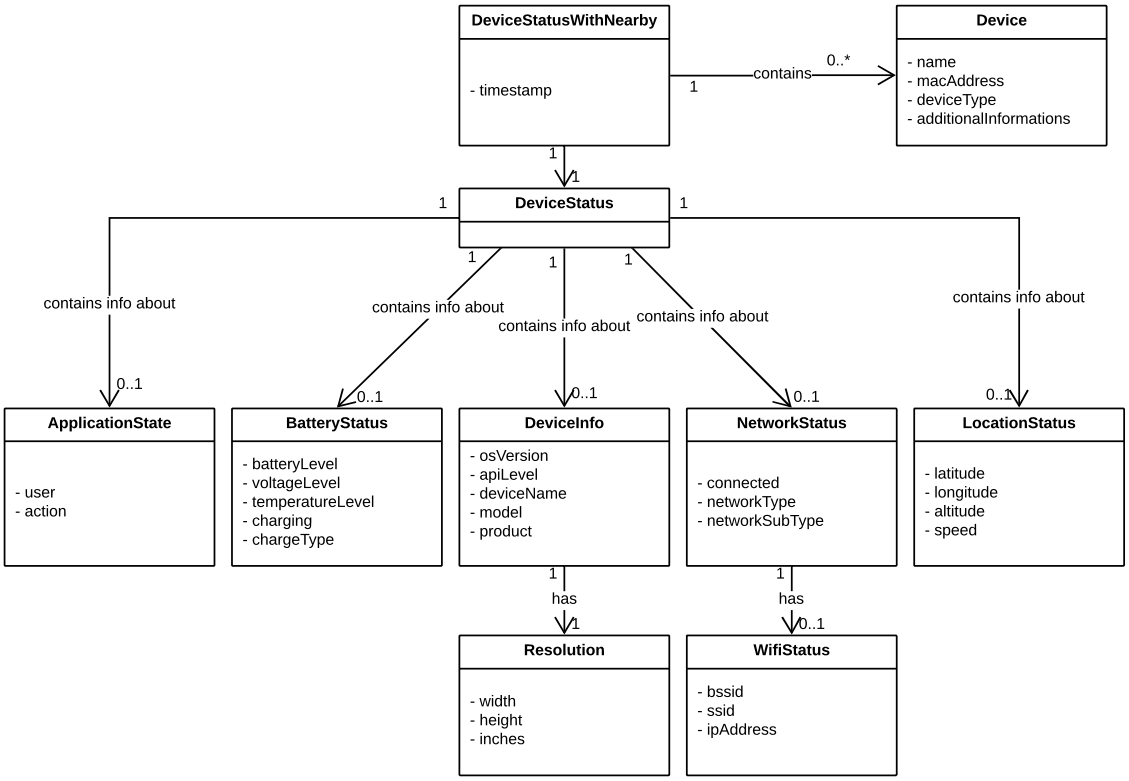
Nasbíraná kontextová data musí mít definovanou strukturu, kterou musí aplikace NSRest, která je uchovává, očekávat. Tu lze ukázat na doménovém modelu. Doménový model je konceptuální model, který popisuje jak strukturu, tak částěčně i chování systému. Definuje totiž, jaké části se v sytému náchází, jaké mohou mít atributy a jak se tyto části ovlivňují [35]. Z doménového modelu 3.1 je patrné, že výsledný model vzniklý z měření (tj. `DeviceStatusWithNearby`) obsahuje čas měření a získaná data. Ta mohou být rozdělena do dvou skupin - na data týkající se zařízení a na data o blízkých okolních zařízeních. Uživatel frameworku AFNearbyStatus má možnost definovat, která data o zařízení se budou a nebudou sbírat. Podle toho, zda se data sbírají, jsou vyplněny tyto informace do modelu `DeviceStatus`. U modelu s informacemi o stavu připojení k síti (v diagramu jako `NetworkStatus`) je `WifiStatus` obsažen pouze tehdy, když je zařízení připojeno během měření k Wi-Fi síti. Co se týká blízkých zařízení, pro všechny typy existuje pouze jeden obecný model obsahující informace, které lze zjistit u všech typů zařízení, nezávisle na použité technologii k jejich získání (Bluetooth, Wi-Fi, 4G). Takovými informacemi jsou mac adresa zařízení, způsob jeho získání a název. Pokud lze zjistit další informace, měly by být uvedeny v poli s dodatečnými informacemi. Například u Bluetooth zařízení je často zjistitelná třída zařízení, tj. například telefon, počítač, nositelné elektronické zařízení, audio-video atd. [2].

Kontextová data mohou být objemná, jejich velikost se odvíjí hlavně od počtu nalezených zařízení v okolí. Je pravděpodobné, že v některých frekventovaných prostředích, jako například ve školách či na studentských kolejích, bude počet okolních zařízení vyšší. Ačkoliv pro každé zařízení existuje množina vlastností, která je u všech zařízení stejná, dodatečné informace, které lze zjistit o každém jednotlivém zařízení, se mohou velmi lišit. Kontext uživatele může být velmi proměnlivý. Změna může být způsobena pohybem uživatele, ale také se může měnit jeho okolí. S nárustem počtu aplikací, které budou tento framework využívat, poroste také velmi rychle množství vyprodukovaných dat. Vývojaři používající framework mají možnost spoustět proces sběru opakovaně a to i v krátkých intervalech, což také ovlivní rychlost vzniku těchto dat. Objem dat, rychlost jejich vzniku či změny a jejich různorodost jsou tři hlavní charakteristiky Big dat [41].

3.2.1.2 Přenos a uchování kontextových informací

Tato data jsou přenesena do aplikace NSRest pomocí webových služeb. Endpointy webových služeb očekávají data v serializovaném formátu, například ve formátu JSON. Data se na straně NSRest deserializují a uloží do databáze.

Na data tohoto typu lze využít jak klasickou relační databázi, tak i NoSQL databázi. Je však žádoucí přenášet a ukládat data jednoho měření spíše jako celek. U relačních databází by se data zbytečně rozdělila do menších celků (tabulek) a v případě vyžádání dat by se opět musela složit dohromady. Navíc by v případě, že se data nedají zjistit nebo uživatel frameworku nadefinoval pouze sběr některých dat, vznikaly by v databázi záznamy se spoustou prázdných hodnot. Dále u těchto dat není předpokládáno, že by se někdy upravovala, spíše budou nahrazena daty z nových měření. Také by se dalo předpokládat, že čtení záznamů z databáze bude častější než zápis. Data se v průběhu používání aplikace totiž sbírají pouze jednou za daný časový úsek, zatímco vyžadována jsou častěji, neboť si uživatel mezitím několikrát zobrazí komponenty UI, které jsou s pomocí těchto dat upravovány [54].



Obrázek 3.1: Doménový model zobrazující strukturu kontextových dat

Z těchto důvodů je výhodnější použít NoSQL databázi. Tomuto typu databází tolik nezáleží na struktuře dat, která může být velmi proměnlivá. Jsou také dobře horizontálně škálovatelné, data mohou být velmi jednoduše replikována či distribuována. K tomuto účelu existují ve většině NoSQL databází již implementované mechanismy. Tento typ databází může být také vhodný při prototypování či agilním vývoji, kdy se může schéma (pokud je zavedeno) či struktura dat často měnit [22].

Typů NoSQL databází je více. Mezi nejznámější z nich patří dokumentové databáze, databáze klíč-hodnota, grafové databáze nebo například XML databáze. Databáze klíč-hodnota jsou nejjednoduším typem NoSQL databází, který pracuje podobně jako hashovací tabulka. Ukládají se páry klíč-hodnota, přičemž klíč slouží jako identifikátor, pod kterým je uložena libovolná hodnota. Co je v hodnotě uloženo, databázový systém nezajímá. Grafové databáze mají strukturu orientovaného či neorientovaného grafu, kde uzly grafu reprezentují data a hrany reprezentují vztahy mezi nimi a jejich vlastnosti.

Dokumentové databáze ukládají informace jako nezávislé jednotky - dokumenty, což umožňuje jednodušeji ukládat nestrukturovaná data. U dokumentů nemusí být definováno žádné schéma, nicméně je možné ho určit. Dokumenty se ukládají v určitém formátu, velmi populárním je formát JSON [22]. Nejznámější dokumentovou databází je MongoDB, dalšími

zástupci jsou například CouchDB, MarkLogic, OrientDB nebo Couchbase [54]. MongoDB používá pro serializaci formát JSON a BSON. JSON je klasická Javascriptová objektová notace, ve které je uživateli dokument prezentován. Uvnitř však MongoDB používá pro zvýšení efektivity, poskytnutí více datových typů nebo lepšímu řazení formát BSON (Binary JSON), což je binární reprezentace JSON formátu [19]. Jelikož jsou kontextová data z AFNearbyStatus v aplikaci NSRest přijímána a poskytována ve formátu JSON, je vhodné použít pro jejich uchovávání právě dokumentovou databázi MongoDB.

3.2.2 Struktura proxy aplikace

Nyní bude popsána proxy aplikace. Ve funkčních požadavcích proxy aplikace bylo zmíněno, že je nutné zajistit správu komponent, jež se mají nacházet v klientských aplikacích. Vlastnosti komponent by mělo jít nastavit skrz uživatelské rozhraní. Komponenta by měla mít název, pod kterým bude identifikovatelná na straně klienta. Dále by měla obsahovat typ, který určuje, jestli se jedná o formulář, tabulku nebo list. Komponenty se vážou ke klientské aplikaci, ve které se nachází, přičemž proxy takových aplikací může spravovat více. Aplikace by měla mít možnost získat pouze komponenty, které ji patří, takže při získání komponenty by se měla aplikace, ze které je požadavek odeslán, identifikovat.

Proxy má také spravovat obrazovky, na kterých se komponenty nacházejí. Jedním z důvodů je, že se stejná komponenta může vyskytovat na více obrazovkách, nemusí však mít na všech obrazovkách stejnou důležitost a význam. Tím, že se spravovuje kombinace obrazovky a komponenty, je možné stejné komponenty na více obrazovkách od sebe odlišit. Dalším z důvodů je zjednodušení implementace klientské části z hlediska připojení na endpointy, na kterých se nachází metadata komponent. Pokud by proxy spravovala pouze komponenty, na klientské části by se musely místo připojení na původní server [55] definovat připojení na proxy, a to včetně všech paramentrů (např. content-type), které jsou pro získání definice komponenty potřeba. Díky tomu, že proxy spravuje i obrazovky klientské aplikace, je možné klientovi poskytnout definici menu. Tento popis menu by obsahoval definici připojení na proxy, ze kterých lze získat definice obrazovek. Tyto definice obrazovek by obsahovaly připojení na proxy, ze kterých lze získat komponenty, jež se na obrazovkách nachází. Klientská část tak musí jen z této definice menu vytvořit tlačítka. Když uživatel na tlačítko z menu klikne, stáhne se definice příslušné obrazovky a vytvoří se buildery komponent, kterým se přípojení nastaví [55]. Buildery již budou automaticky při vyrobení komponenty zasílat požadavek na získání definice komponenty na proxy aplikaci (místo na původní server), a to včetně všech parametrů, které byly na proxy definovány jako potřebné. Buildery však jsou jen předpřipravené, v definici připojení se mohou nacházet proměnné, které je třeba doplnit. Klientský framework si tedy musí zachovat možnost nastavit builderu parametry připojení, ze kterých se nahradí proměnné v popisu připojení, což bylo popsáno dříve v sekci *Získání komponenty skrz proxy aplikaci.*

Jak již bylo zmíněno dříve, proxy aplikace spravuje i business případy, které mohou v klientské aplikaci nastat. Takovým business případem může být například objednání letenky. Proces je často rozdělen do několika kroků (fází). V případě objednávky letenky to můžou být například tyto čtyři fáze: výběr letu, vyplnění informací o zavazadlech či pojištění, vyplnění osobních informací a platba. Každá z fází může být reprezentována jednou čí více obrazovkami, na kterých se mohou nacházet komponenty. Například ve fázi výběru letu může

uživatel vidět tabulku s dostupnými lety, při vyplňování osobních informací formulář. Tuto strukturu zachycuje doménový model proxy aplikace B.3.

Doménový model B.3 lze rozdělit na část datovou, business a část klasifikační. Business a klasifikační část je inspirována a z části převzata z článku *Automated User Interface Generation Involving Field Classification* [56]. Mezi datovou část patří entity, které ukládají data o obrazovkách, komponentách a jejich napojení na server. Ty byly popsány výše stejně tak, jako části týkající se business případů a fází, které lze zařadit do business části. Do ní také patří entity v levé dolní části modelu. Cílem business části je držet infomace o důležitosti jednotlivých komponent, respektive jejich částí, v rámci konktrétní business fáze. Z tohoto důvodu je přiřazeno v business fázi 0 až N business polí (`BCField`) a každé z nich má definovanou svou důležitost (v modelu jako `BCFieldSeverity`). Důležitost je určena účelem a mírou potřeby dané informace, tedy jak důležité je, aby uživatel danou informaci systému poskytl. V tabulce 3.1 je popsán význam jednotlivých účelů polí a v tabulce 3.2 úrovně potřeby informací.

Tabulka 3.1: Typy účelů polí v komponentě

Účel	**Popis**
SYSTEM_IDENTIFICATION	Značí, že hodnota v poli je nutná k dokončení business úkolu v systému.
SYSTEM_INFORMATION	Značí, že hodnoty z tohoto pole se ukládají a jsou používány systémem.
INFORMATION_MINING	Značí, že hodnoty z pole jsou používány pro data mining a mohou být užitečné později.
FUTURE_INTERATION	Značí, že hodnoty z pole mohou být využity systémem ke zlepšení budoucí interakce uživatele se systémem.

Tabulka 3.2: Úrovně potřeby informací polí v komponentě

Úroveň potřeby informace	**Popis**
CRITICAL	Značí, že bez tohoto pole není možné dokončit daný business úkol.
REQUIRED	Značí, že pole není kritické pro dokončení business úkolu, ale jeho vyplnění může zlepšit interakci uživatele se systémem, což může zjednodušit i dokončení business úkolu.
NEEDED	Pole slouží ke sběru informací, které jsou určeny k úpravě uživatelského rozhraní.
NICE_TO_HAVE	Informace z tototo pole mohou být později užitečné, ale nejsou povinné.

Tyto infomace je nutno specifikovat v uživatelském rozhraní proxy všem polím v komponentách, které se nachází na obrazovkách přiřazených k fázi business případu. Je vhodné

podotknout, že stejná komponenta se může nacházet na více obrazovkách, které jsou všechny v rámci jedné business fáze. V takovém případě se musí definovat důležitost polí vícekrát, pro každou obrazovku zvlášť, neboť význam polí může být rozdílný pro rozdílné obrazovky.

Klasifikační část slouží k ohodnocení polí podle jejich důležitosti v dané fázi a následné klasifikaci pole, která určí, jak se má pole v komponentě po její úpravě chovat. V doménovém modelu B.3 je popsána entitami `Client`, `DeviceType`, `ClientProperty`, `Configuration`, `GeneratedField` a `Behavior`. `Client` by měl obsahovat zjištěné kontextové informace, které se použijí v procesu úpravy komponenty, který je popsán v sekci *Úprava komponenty s využitím kontextu*. Základní informace se drží přímo v této entitě a patří mezi ně například identifikátor zařízení nebo jméno uživatele, který zařízení právě používá. Typ zařízení je definován pomocí `DeviceType`. Další informace, například typ připojení, kapacita baterie, či okolní zařízení, jsou uloženy v entitě `ClientProperty` v párech klíč-hodnota. Význam entit `GeneratedField`, `Configuration` a `Behavior` bude vysvětlen později při popisu úpravy komponenty.

3.2.3 Získání komponenty skrz proxy aplikaci

Zatímco v původním řešení [55, 47] získávala klientská aplikace informace o komponentách přímo ze serveru, který je poskytoval, nyní k tomu využívá proxy. Průběh je jednoduchý. Aplikace pošle požadavek na proxy, kterou komponentu chce, proxy z definic komponenty, kterou má uloženou v databázi, zjistí endpoint serveru, který data poskytuje. Tato data ze serveru získá a předá je klientské aplikaci.

Avšak i v tomto jednoduchém procesu existují jistá úskalí. Endpoint na serveru, jež poskytuje informace o komponentě, může být zabezpečený nebo vyžadovat, aby měl HTTP požadavek na tato data různé parametry. Takovými parametry může být content-type nebo autorizace uživatele. Některé tyto informace, jako zmiňované údaje sloužící k autorizaci uživatele, má však k dispozici pouze klient, a to jen při běhu aplikace. V původním řešení [55] byl tento problém řešen tak, že se do definicí endpointu, které byly původně v XML souboru, vložily proměnné, jež se před použitím definice endpointu doplnily z mapy parametrů, kterou uživatel frameworku specifikoval v kódu před postavením komponenty. Proměné začínaly znakem # a jejich název byl uzavřen ve složených závorkách. V úryvku původního XML souboru 3.1 lze vidět příklad proměnných na řádcích 8 a 9. Název proměnné musel být stejný jako název klíče v mapě parametrů, ze které se proměnné nahrazovaly. Jelikož nyní má být správa těchto připojení ke zdrojům s informacemi o komponentách v rámci proxy, je nutné implementovat mechanismus, který tyto parametry přenese na klienta, kde se při získávání komponenty doplní a odešlou zpět na proxy.

3.2.3.1 Úprava komponenty s využitím kontextu

Před odesláním definice komponenty z proxy na klienta je možné využít kontext uživatele a vytvořit tak osobitější formu uživatelského rozhraní. Nejprve je třeba kontext uživatele získat z aplikace NSRest, která jej poskytuje pomocí webových služeb. Aby byl získaný kontext co nejrelevantnější, měla by aplikace NSRest poskytovat endpoint, který vrátí záznam měření s časem co nejbližším k času aktuálnímu. Bohužel se nelze vyhnout situaci, že bude kontext zastaralý. Záleží na tom, kdy se měření kontextu provede. Ke zvýšení pravděpodobnosti,

Listing 3.1: Ukázka použití proměnných v XML specifikaci zdrojů definic komponent

```
<?xml version="1.0" encoding="UTF-8"?>
<connectionRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    ↪ instance">
    <connection id="personProfile">
        <metaModel>
            <!--endpoint specification, header parameters
                ↪ specification-->
            <security-param>
                <securityMethod>basic</securityMethod>
                <username>#{usernameVariable}</username>
                <password>#{passwordVariable}</password>
            </security-param>
        </metaModel>
        <!--specification of send and data source-->
    </connection>
</connectionRoot>
```

že uživatel při rozkliknutí obrazovky v klientské aplikaci bude mít k dispozici co nejbližší kontextové informace, je dobré proces měření provádět buď při jakémkoliv probuzení aplikace nebo periodicky i v době, kdy je aplikace na pozadí nebo zařízení spí. Druhá možnost však může výrazně ovlivňit výdrž baterie, jak už bylo zmíněno ve funkčních požadavcích na framework pro sběr kontextových dat [53].

V této práci jsou kontextové informace využity dvěma způsoby. Buď se vyhodnotí aktuální kontext nezávisle na předchozích nebo se využijí i předchozí kontexty. V prvním případě se může jednat o vyhodnocení na základě kapacity baterie zařízení nebo typu připojení. Například při nízké kapacitě baterie nebo špatném připojení bude komponenta UI obsahovat pouze nejdůležitější části nutné k dokončení úkolu.

Ve druhém případě je zkoumána hlavně podobnost kontextů. Aby bylo možné porovnávat více kontextů, musí mít něco společného. Například denní dobu či akci, kterou uživatel v aplikaci prováděl v momentě, kdy byl kontext naměřen. Na základě toho lze uvažovat následovně. Uživatel nyní provádí stejnou akci jako včera ve stejnou denní dobu, pokud má stejný či podobný kontext, lze mu nabídnout lepší verzi uživatelského rozhraní. Z tohoto důvodu je vhodné u jednotlivých měřeních ukládát i aktuální akci uživatele a čas měření, které je možné najít v sekci o struktuře kontextových dat nebo přímo v diagramu 3.1.

Podobnost kontextu lze určovat například u blízkých okolních zařízení. Pokud uživatel provádí operaci v internetovém bankovnictví ze stejného zařízení, účtu, ve stejnou dobu a ve stejném kontextu jako obvykle, lze usoudit, že je to standardní situace a uživateli například odpustit dodatečnou verifikaci. Naopak pokud by se kontext uživatele od minulé operace změnil, tj. má například kolem sebe úplně nová zařízení, je vhodnější verifikaci ponechat, neboť se může stát, že aplikaci používá pod tímto uživatelským účtem někdo jiný [58].

Proxy aplikace využívá kontextová data k úpravě zmiňovaných komponent UI, tudíž se mohou upravovat pouze vlastnosti jednotlivých komponent a jejich částí. V sekci o strukutře

proxy aplikace bylo slíbeno vysvětlení tří zbývajících entit `GeneratedField`, `Configuration` a `Behavior` nacházejících se v doménovém modelu B.3, nejprve je však nutné si přiblížit proces úpravy komponenty, který bude probíhat následovně. Proxy si vyžádá informace o komponentě ze serveru, který je poskytuje. Z klientského požadavku musí vědět, na které obrazovce se komponenta nachází. Díky kombinaci komponenty a obrazovky je systém schopen určit business fázi. Ta obsahuje definice důležitosti polí komponenty - účel a míru potřeby informace. Dále se získají kontextové informace o uživateli ze systému NSRest a uloží se v modelu `Client`.

Množina business polí s definicemi důležitosti a kontextové informace o klientovi slouží jako vstupy pro výpočetní modul. Ten vypočítá na základě těchto dat pro každé pole ohodnocení, na jehož základě se může určit chování (`Behavior`) pole ve výsledné komponentě. Pole se může chovat celkem pěti způsoby, které jsou popsány v tabulce 3.3.

Tabulka 3.3: Způsoby chování polí

Chování	**Projeví se v komponentě**	**Popis**
REQUIRED	Formulář	Značí, že je pole viditelné a je vyžadováno. Uživatel tedy pro odeslání formuláře bude muset tuto hodnotu vyplnit.
VALIDATION	Formulář	Pole je viditelné, ale není vyžadováno. Jsou však na něm zkontrolovány validace, které má pole nastavené. V případě, že pole již mělo validaci required, je tato validace zrušena.
ONLY_DISPLAY	Formulář	Pole je viditelné, ale není vůbec validováno.
HIDDEN	Formulář, Tabulka, List	Pole je v komponentách přítomno, ale je pro uživatele zneviditelněno.
NOT_PRESENT	Formulář, Tabulka, List	Pole není v komponentě vůbec přítomno.

Pro každé z těchto chování jsou v `Configuration` nadefinovány spodní a horní limity ohodnocení, které jsou specifické pro danou business fázi. Chování se tedy vybere podle toho, do kterého intervalu v konfiguracích spadá jeho ohodnocení. Následně vzniká model `GeneratedField` s tímto chováním, což značí, že pole prošlo procesem ohodnocení a klasifikace a je možné jej na základě určené skupiny chování upravit v definici komponenty. Úprava pole je přímočará, systém pole nalezne v definici komponenty a upraví jeho vlastnosti tak, jak je to popsáno v tabulce 3.3.

3.2.3.2 Výpočet ohodnocení polí komponenty

V řešení je nutné definovat, jakým způsobem se pole ohodnotí. K tomu je nutné vytvořit algoritmus, který zpracuje definované důležitosti pole a kontextové infomace. Algoritmus může pole ohodnotit libovolně. Existuje jen jedno omezení. V případě, že má pole definovaný účel jako `SYSTEM_IDENTIFICATION` a zároveň je jeho severita `CRITICAL`, mělo by získat

maximální možné ohodnocení, čímž je zajištěno, že má pole vždy chování typu `REQUIRED` a je tedy ve výsledné komponentě vždy vyžadováno [56]. Při návrhu algoritmu je nutné velmi dobře zvážit význam jednolivých kontextových informací, stejně tak jako důležitosti polí. Také je třeba mít na paměti, že výsledné skóre určuje chování podle definované konfigurace, kterou je tedy nutné při ohodnocování polí vzít v potaz.

V této práci jsou vytvořeny dva algoritmy na ohodnocení polí. První by měl pracovat jen s aktuálním kontextem, konkrétně s aktuální kapacitou baterie a typem připojení k síti. Cílem tohoto algoritmu je, aby pole, která nejsou kritická pro dokončení business úkolu, nebyla v komponentě přítomna nebo alespoň nebyla vyžadována.

Druhý algoritmus by měl pracovat s blízkými okolními zařízeními. Okolní zařízení v aktuálním kontextu se porovnají s okolními zařízeními, které měl uživatel v blízkosti v době, kdy prováděl stejnou akci naposledy. Výsledkem porovnání je procentuální podobnost mezi oběma skupinami okolních zařízení. Pokud je podobnost vysoká, nejsou některá pole v komponentě vyžadována. Možností, jak algoritmus navrhnout, je samozřejmě více. Například aby bylo porovnání relevantnější, podobnost zařízení může být zkoumána jen na těch, která jsou se zařízením uživatele spárována přes Bluetooth. Dalším způsobem je neporovnávat pouze dvě měření (tj. aktuální a poslední), ale vytvořit z minulých měření standardní kontext uživatele, se kterým se aktuální kontext porovná. V předchozím textu bylo zmíněno, že množství informací, které je aktuálně možné o okolních zařízeních získat, je značně omezené. S postupem času se však může tato situace změnit. Čím více informací půjde zjistit, tím nápaditější rozhodovací algoritmy mohou vznikat. V této práci jde ale primárně o proof of concept a jako ukázka jednoduchý algoritmus postačí.

3.2.3.3 Stavy komponenty

Jak lze odvodit z předchozího popisu, komponenta prochází během jejího vytvoření několika stavy, kterých si lze povšimnout v diagramu B.5. Jak již bylo popsáno, komponenty se na proxy ukládají ve formě popisu zdrojů, ze kterých je lze pak získat. Pokud tato definice zdroje obsahuje proměnné, které je potřeba nahradit, nachází se komponenta ve stavu *Partly defined component source*, neboli má definován zdroj pouze částečně. Pokud žádné proměnné neobsahuje nebo již byly doplněny, je komponenta ve stavu *Fully defined component source*, tedy má dostupné všechny informace k tomu, aby mohl být získán její popis z backendové aplikace, která popisy komponent poskytuje. Poté, co se tato definice získá, přechází komponenta do stavu *Defined*. Z tohoto stavu lze komponentu rovnou postavit a přejít do stavu *Built* nebo lze definici komponenty před postavením upravit zmiňovaným způsobem na základě její klasifikace, což odpovídá přechodu do stavu *Build* přes stav *Filtered*. Postavenou komponentu lze ještě vložit do uživatelského rozhraní, čímž komponenta přejde do stavu *Displayed*. Stav *Built* potažmo *Displayed* je v procesu tvorby komponenty konečný. Po sestavení komponenty začíná její životní cyklus, jehož popis lze nalézt v mé bakalářské práci [47].

3.2.3.4 Rekapitulace spolupráce aplikací

V předchozím textu již bylo naznačeno, jak spolu jednotlivé aplikace komunikují a co vše je potřeba udělat, aby klientská aplikace obdržela upravenou komponentu. Pro přehlednost

je zde spolupráce aplikací zrekapitulována a popsána zjednodušeným activity diagramem, který lze nalézt v příloze B.4. Jak lze v digramu vidět, klientská aplikace (tj. Frontend) zasílá požadavek na získání definice celé obrazovky. Definice obrazovky obsahuje pro všechny komponenty, které by se v ní měly nacházet, popisy připojení na proxy včetně parametrů, které je potřeba s požadavkem poslat. Pro každý z těchto popisů připojení se vytvoří buildery komponent a nastaví se jim toto připojení. Poté, když bude klientská aplikace chtít pomocí builderu vytvořit komponentu, zašle požadavek na proxy podle přípojení, které bylo builderu nastaveno. Proxy poté získá definici komponenty ze serveru, který ji poskytuje (tj. AFServer). Pokud je obrazovka, kterou klient požadoval spárovaná s business případem, měly by být na proxy pro každé pole komponenty definovány již zmiňované business vlastnosti - účel a důležitost informace. Z NSRest se získají kontextová data pro zařízení, které požadavek poslalo, a na základě těchto dat a business vlastností se definice komponenty upraví a v tomto stavu se navrátí klientské aplikaci. Ten upravenou definici interpretuje, čímž vznikne požadovaná komponenta. Takto se to provede pro všechny komponenty, které mají na obrazovce být, a uživateli se zobrazí sestavená obrazovka.

3.3 Úprava existujícího řešení

Proces generování komponenty se od původního řešení [55, 47] změnil. Aby byla zajištěna původní funkcionalita, tj. aby projekty z původního řešení fungovaly tak jako předtím, je potřeba je upravit. První změnou je, že by se nyní neměly získávat přímo definice komponenty. Aplikace by nyní měla získat z proxy definici celého menu. Z menu by se měla vytvořit tlačítka, po jejichž rozkliknutí se získá definice obrazovky a na základě této definice se připraví buildery komponent, které se na obrazovce nacházejí. Programátorovi klientské aplikace by měly být tyto buildery předány, aby je mohl využít k případnému sestavení komponenty a vložení do uživatelského rozhraní. Vytvoření grafické reprezentace menu z definice, kterou zašle proxy, by měla být zodpovědností frameworků AFAndroid a AFSwinx, které mají tvorbu UI prvků z definic na starosti. Kromě převodu definice do grafické reprezentace by tlačítka v menu měla mít nastavena posluchač na kliknutí, který provede požadavek na proxy k získání definice obrazovky a zajistí předpřípravu builderů komponent, které zpřístupní vývojáři klientské aplikace. Dále by měly frameworky umožňovat získání a zpracování definice obrazovky i manuálně, tedy ne pouze přes sestavené menu, pro případ, že by bylo potřeba definici obrazovky získat i bez toho, aby uživatel na něco kliknul. Takovým případem může být například Login obrazovka, která se automaticky zobrazuje bez interakce uživatele po startu aplikace.

Klientské aplikace se změnám také nevyhnou. Je potřeba nahradit menu, které je nyní vytvořeno manuálně, za menu generované, což s sebou přináší nemalé množství úprav v logice aplikace. Také se změní způsob inicializace builderů komponent. V původním řešení se builderům předával XML soubor s konfigurací připojení na server, který poskytoval definice komponent. Tento XML soubor již není potřeba, neboť jsou tato připojení včetně parametrů ve správě proxy, která je poskytuje právě s definicí obrazovky, a připojení by se mělo automaticky builderu nastavit při jejím zpracování, které je popsáno výše.

3.4 Použité technologie

V této sekci jsou popsány technologie, které byly použity při výrobě či úpravě aplikací nebo frameworků. Konkrétně se jedná o Android framework pro sběr dat AFNearbyStatus, proxy aplikaci, aplikaci pro uchovávání dat NSRest, frameworky pro interpretaci generovaného UI AFAndroid a AFSwinx a ukázkové klientské aplikace.

3.4.1 Původní řešení

V první řadě jsou uvedeny aplikace a frameworky původního řešení, které jsou v této práci využívány nebo rozšiřovány.

3.4.1.1 AFRest

Tento framework slouží k vytvoření definice komponenty, které jsou poskytovány pomocí RESTových webových služeb ve formátu JSON. Popis komponenty se vytváří na základě inspekce modelových tříd na serveru, kterou zajišťuje knihovna AspectFaces [7]. V popisu komponenty jsou obsaženy informace jako její rozložení v rozhraní, popisy polí, které má obsahovat, jejich labely, typy widgetů, které by je měly reprezentovat nebo například validační pravidla, která by měla pole splňovat.

3.4.1.2 AFServer

Tato serverová aplikace byla vytvořena pro demostraci funkcionality frameworku AFRest. Aplikace slouží jako backend zpracovávající data z klientských aplikací a zároveň je vystaveno REST API, ze kterého lze získat popisy komponent, které jsou vytvořené pomocí zmiňovaného frameworku AFRest. K definici těchto endpointů je použita technologie RestEasy. Aplikace je nasazena na aplikačním serveru Glassfish. Data jsou držena v jednorázové (v tom smyslu, že po vypnutí aplikace data nepřetrvávají) in-memory databázi DerbyDB [55].

3.4.1.3 AFAndroid a AFSwinx

Data, která AFServer poskytuje, jsou ve formátu JSON a lze je interpretovat v jakékoliv technologii, která umožňuje parsování JSONu a dynamické vytváření prvků uživatelského rozhraní za běhu aplikace. V původním řešení byli tito interpreteři tři - AFSwinx pro Java SE aplikace, AFAndroid pro Android aplikace a AFWinPhone pro Windows Phone aplikace. Již dříve v této práci bylo rozhodnuto, že se mají rozšiřovat pouze první dva, tedy AFSwinx a AFAndroid. Pro každý z interpreterů zmíněných výše existuje ukázková aplikace, která demostruje možnosti frameworku.

3.4.2 Java EE

Java EE je platforma založená na programovacím jazyce Java, která se využívá ke tvorbě robustnějších aplikací (např. webové aplikace). Skládá se z několika kontejnerů, přičemž každý z nich zajišťuje podporu určité funkcionality. Například Web kontejner zajišťuje podporu JSP

stránek a servletů, sloužících pro prezentaci obsahu uživateli, EJB kontejner obsahuje vše, co je spojeno s business logikou aplikace [36]. Dále Java EE podporuje například RESTful služby, databázové frameworky, JPA anotace atp. V této práci byla Java EE využita při implementaci proxy aplikace a aplikace NSRest. U obou aplikací je využito podpory Restových služeb ke zpřístupnění rozhraní pro komunikaci mezi aplikacemi. V případě UI proxy je využito Servlet API a JSP stránky z Web kontejneru k vytvoření uživatelského rozhraní pro správu obrazovek, komponent a business případů. Java EE aplikace jsou obvykle nasazeny na aplikační server. V této práci byl využit aplikační server Glassfish, jemuž je později v této podkapitole věnována vlastní sekce.

3.4.3 Java SE - Swing

Jak bylo zmíněno, je potřeba upravit i framework AFSwinx vytvářející uživatelské rozhraní z jeho definice pro Java SE aplikace, konkrétně v knihovně pro tvorbu UI Swing. Swing je jeden ze tří frameworků sloužících pro tvorbu UI v desktopových Java SE aplikacích. Dalšími dvěma jsou AWT a JavaFX. JavaFX je nejnovější a měla by nahrazovat Swing [18], stejně tak jako Swing nahradil AWT. Nicméně Swing je stále používán velkým množstvím aplikací. Vývojáři navíc mají možnost Swingové GUI pouze rozšířit o JavaFX prvky. Díky tomu, že je Swing stále podporován, má smysl tento framework rozšiřovat.

3.4.4 Android SDK

Android SDK je využito v mobilní alternativě AFSwinx - frameworku AFAndroid, frameworku AFNearbyStatus sloužícím ke sběru kontextových dat a také v ukázkové Androidí aplikaci. Toto SDK nabízí nástroje pro tvorbu Android aplikací a knihoven. Zdojové kódy aplikace lze psát v Javě, Kotlinu nebo C#, zde byla zvolena Java. Android SDK poskytuje kromě knihoven využitelných při tvorbě aplikací také nástroje pro build aplikací, ADB (Android Debug Bridge), umožňující instalaci aplikací do zařízení a sledování jejich průběhu, nebo Android emulátor [5]. Grafické uživatelské rozhraní může být definováno staticky v XML šablonách nebo dynamicky tvořeno pomocí kódu, kterého využívá právě AFAndroid. Android jako operační systém má mnoho verzí. Dle statistik [6] má k únoru 2018 nejvíce zařízení verzi 6.0 Marshmallow. Nejstarší verze, která je ještě relativně rozšířená, je Android 4.4 Kitkat, což odpovídá SDK verze 19. Původní verze AFAndroid frameworku podporovala SDK verze 11, z důvodů implementace nové funkcionality musí být minimální verze SDK zvýšena na 18, což stále pokryje většinu aktuálních verzí na trhu.

3.4.5 Maven a Gradle

. Pro správu závislostí a build aplikací je v případě Java EE aplikací použit Apache Maven, u Android aplikací Gradle. Gradle je oficiálním buildovacím nástrojem pro Android. Byl zvolen Googlem pro jeho flexibilitu, Maven je podle něj lépe pochopitelný, avšak hůře customizovatelný v případě, že je potřeba build proces speciálně upravit. Gradle je navíc rychlejší, a to hlavně díky tomu, že provádí vždy pouze to, co je nutné. Občas se také stane, že jsou v projektu závislosti, které jsou nechtěné. Zatímco Maven umožňuje tyto definice pouze přetížit jinou verzí, Gradle umožňuje vybírat, které závislosti se použijí nebo definovat pravidla substituce této závislosti. Maven se na druhou stranu používá již delší

dobu, má prozatím větší podporu ve vývojových prostředích a je často spojený právě s Java EE aplikacemi. Gradle navíc umožňuje automatickou konverzi z Mavenu pro případ, že by bylo nutné přejít z jednoho systému na druhý.

3.4.6 MongoDB

V sekci *Přenos a uchování kontextových informací* bylo uvedeno, že je pro uchování kontextových dat využita MongoDB. MongoDB je jedna z nejznámějších NoSQL dokumentových databází. Aktuální verze je 3.6.3. Pro interakci s tímto databázovým systémem pomocí jazyka Java je nutné použít MongoDB Java driver [21]. Ten poskytuje podporu pro ukládání Javovských POJO tříd. Driveru je nutné jen říct, kde třídy nalezne. Pro převod POJO do BSON reprezentace jsou použity tzv. kodeky. V případě, že některý z datových typů není podporovaný, lze si dopsat vlastní kodek, který jej do BSON reprezentace převede. Java driver také poskytuje spoustu anotací, kterými se dá převod do BSON reprezentace instruhovat, například anotace `@BsonIgnore` atribut třídy při serializaci vynechá. Pro automatickou serializaci a deserializaci je nutné mít na atributy POJO třídy vytvořeny settery a gettery.

3.4.7 PostgreSQL

Pro správu dat na proxy je použita klasická databáze, kontrétně PostgreSQL. PostgreSQL je open-source objektově-relační databázový systém vydávaný pod licencí MIT, což je jedna z věcí, kterou se odlišuje od svého konkurenta MySQL. Stejně jako u MongoDB pro interakci s databázovým systémem pomocí Javy, je potřeba mít v projektu PostreSQL JDBC driver. Ten je stejně jako celý databázový systém také open-source. Jeho aktuální verze podporují všechny dominantní verze Javy tj. Javu 6,7 i 8 [24].

3.4.8 Glassfish

Glassfish [13] je aplikační server vyvinutý pro platformu Java EE. Glassifish je open-source, existuje však i komerční verze od společnosti Oracle. Původní řešení generování UI bylo odladěno na právě tento aplikační server [55], konkrétně na verzi 3 a 4. Aby nemusel být instalován nový aplikační server pro proxy aplikaci a NSRest, bylo rozhodnuto odladit je na tento aplikační server také. Glassfish je v základu pro tyto dvě aplikace dostačující, má integrovanou podporu pro REST, což je stěžejní pro komunikaci aplikací. Jediným nedostatkem je, že v základu používá DerbyDB, což je in-memory databáze, a disponuje JDBC driverem pouze pro ni. Aby mohla proxy a aplikace NSRest využívat svých PostreSQL a Mongo databází, bylo nutné tyto drivery dodat externě a poté je v administraci Glassfishe nakonfigurovat. Aktuální verze Glassfishe je 5.0, v této práci je ale použita předchozí verze 4.1.2, jelikož převod a odladění původní aplikace by byl zbytečnou komplikací nesouvisející s tématem práce.

Kapitola 4

Implementace

Jak bylo zmíněno v analýze, je potřeba pro dosažení cílů práce vytvořit jeden nový framework zajišťující sběr kontextových dat v Android aplikacích a dvě serverové aplikace, jednu pro ukládání kontextových dat a druhou sloužící jako proxy mezi klientem a serverem posyktujícím metadata o komponentách, které se klientem využívají k automatické tvorbě prvků uživatelského rozhraní. Poté je třeba upravit stávající proces generování uživatelského rozhraní tak, aby nasbíraná data o kontextu uživatele využíval k tvorbě přívětivějšího uživatelského rozhraní. Vložení proxy mezi klienta a server a potřeba využití kontextových dat vyžaduje úpravu frameworků pro interpretaci informací o komponentách UI AFAndroid [47] a AFSwinx [55]. Změna těchto frameworků s sebou přináší i změnu ukázkových klientských aplikací. Popis, jak spolu aplikace komunikují, byl již popsán v analýze a je zobrazen na diagramu nasazení B.2.

4.1 AFNearbyStatus

AFNearbyStatus je framework umožňující sběr dat o kontextu, ve kterém se zařízení nachází. Skládá se ze dvou částí, první část sbírá informace o zařízení samotném a druhá část sbírá informace o blízkých okolních zařízeních.

4.1.1 Modul DeviceStatus

Tato část frameworku se zabývá sběrem informací o zařízení, na kterém aplikace běží. Je schopen získávat data o

- baterii,
- hardwaru a softwaru zařízení,
- poloze a rychlosti zařízení,
- připojení zařízení k síti,
- aktuálním uživateli aplikace,

- aktuální akci uživatele aplikace.

Každá z těchto oblastí má svého vlastního *minera*, který tato data ze zařízení *vytěží*. Všichni z těchto minerů mají abstraktního předka, který u minerů vynucuje implementaci metody `mineAndFillStatus(DeviceStatus deviceStatus)`, ve které by se měla data získat a nasetovat do modelu předávaného v parametru metody.

4.1.1.1 BatteryStatusMiner - informace o baterii

Stará se o získání informací o baterii. Detaily o baterii se nacházejí v tzv. sticky `Intentu`, do kterého data broadcastuje `BatteryManager`. `Intent` je asynchronní machanismus, který umožňuje předávání zpráv mezi jednotlivými komponentami v Androidu. Pokud je `Intent` sticky, znamená to, že přežívá v systému a je připraven zpracovat budoucí zprávy, u kterých není známo, kdy nastanou. Takovou zprávou může být například informace o tom, že je baterie v kritickém stavu [3].

Z totoho `Intentu` miner získává aktuální kapacitu baterie, teplotu baterie, její napětí a jestli je baterie právě nabíjena, popřípadě jakým způsobem. Mezi způsoby, kterými může být baterie nabíjena, patří nabíjení přes USB, nabíjení přes zásuvku a bezdrátové nabíjení.

4.1.1.2 NetworkStatusMiner - informace o síti

Informace o připojení zařízení k síti lze získat ze systémové služby `CONNECTIVITY_SERVICE`. Tato služba monitoruje stav připojení k síti a jeho změny. NetworkStatusMiner z ní zjišťuje, zda je zařízení připojeno, v případě, že ano, tak typ (WI-FI, MOBILE) a podtyp (3G, 4G) připojení. V případě, že je připojen přes Wi-Fi, tak je schopen zjistit název sítě, její BSSID a SSID identifikátory a IP adresu, která byla zařízení v rámci této sítě přidělena.

4.1.1.3 LocationStatusMiner - informace o poloze

V případě polohy zařízení je situace o něco komplikovanější. Poloha zařízení se řadí mezi velmi citlivá data a uživatel musí aplikaci při běhu explicitně udělit práva k jejímu získávání [4]. Konkrétně se jedná o práva `ACCESS_COARSE_LOCATION`, které umožní přístup k hrubé poloze zařízení, a `ACCESS_FINE_LOCATION`, které umožní přesné získávání polohy. Miner tedy kontroluje, zda tato práva aplikace má. Vzhledem k tomu, že ve funkčních požadavcích bylo určeno, že by aplikace měla běžet na pozadí, pokud možno bez zbytečných zásahů do aplikace, nebude miner toto přístupové právo vyžadovat. Pokud ho však vývojář aplikace vyžádal z jiných důvodů a aplikace toto právo má, lze zjistit informace o poloze zařízení.

Poloha zařízení může být zjištěna pomocí GPS nebo pomocí internetu. Použití GPS je samozřejmě přesnější, pokud však nemá uživatel zapnuté určování polohy, je možné získat alespoň méně přesnou polohu pomocí sítě [3].

Polohu lze získat opět ze systémové služby. Mechanismus na získání polohy v Androidu nenabízí jednorázové získání polohy, lze pouze registrovat posluchače na její aktualizace, což zavolá funkci jednou na začátku a poté vždy, když se poloha změní. Lze ale zaregistrovat posluchače a při iniciálním získání polohy polohu uložit a posluchače zrušit.

4.1.1.4 DeviceInfoMiner - informace o zařízení

Tento miner získává ze zařízení hardwarové informace a softwarové informace. Mezi tyto informace patří verze operačního systému, verze Android SDK, které systém používá, název a model zařízení, rozměry displaye a mac adresu. Většina z těchto informací je dostupná jako veřejné konstanty, složitější je jen získání mac adresy zařízení a rozměrů displaye.

Mac adresa zařízení se dá získat z Bluetooth či Wi-Fi rozhraní. Od Androidu 6.0 [1] se již z důvodu ochrany dat přes tato dvě rozhraní nelze k mac adrese dostat. Metody, které původně vracely mac adresu zařízení, nyní vrací konstratní hodnotu `02:00:00:00:00:00`. K mac adrese se ale stále dá dostat přes `NetworkInterface` s označením wlan0.

Rozměry obrazovky mobilního zařízení lze získat ze systémové služby `WINDOW_SERVICE`. Ta nabízí šířku a výšku displaye v pixelech. Velikost displayů je však často udávána i v palcích, což je také zajímavý údaj. Naštěstí lze přibližné rozměry v palcích ze šířky a výšky displaye v pixelech poměrně jednoduše vypočítat, jelikož Android nabízí hodnotu, která určuje hustotu pixelů na jeden palec [3].

4.1.1.5 ApplicationStateMiner - informace o stavu uživatele

V analýze bylo popsáno, že pro porovnávání více kontextů, musí mít něco společného, například akci, kterou uživatel provádí, nebo uživatele, který právě aplikaci používá. Pro tyto účely byl vytvořen tento miner, který deklaruje dvě metody pro získání právě těchto dvou údajů. Metody jsou abstraktní, neboť tyto informace mohou být zjišťovány a uloženy v různých klientských aplikacích jiným způsobem. Jejich implementace je tedy ponechána na uživateli frameworku.

4.1.2 Modul Nearby

Nearby modul se zabývá hledáním a sběrem informací o blízkých okolních zařízeních. Výsledkem hledání je seznam zařízení. Všechna zařízení nazávisle na způsobu získání nebo typu zařízení mají společný model `Device`, jak je patrné z diagramu 3.1. Pokud lze o zařízení zjistit informací více, jsou uloženy v hashmapě v párech klíč-hodnota. Byly implementovány tři způsoby hledání a sběru informací o blízkých zařízeních, které jsou v této práci nazvány jako *findery* a budou popsány níže. Každý finder dědí od abstraktního předka a implementuje dvě metody, jedna hledání a sběr informací nastartuje, druhá jej ukončí a nastaví do nadřazeného modelu `DeviceStatusWithNearby` (viz 3.1).

4.1.2.1 BTDevicesFinder - hledání přes Bluetooth

Hledání Bluetooth zařízení poskytuje Android skrz službu `BLUETOOTH_SERVICE`. Uživatel musí mít samozřejmě zapnutý Bluetooth. Pokud zapnutý není, lze jeho zapnutí vyžádat, ale ze stejných důvodů jako u získání informací o poloze zařízení nebude zapnutí frameworkem vyžadováno. Služba poskytuje začít hledání metodou `startDiscovery()`, která automaticky v jiném vlákně vyhledává zařízení. Před začátkem vyhledávání je nutné službě nadefinovat callback, který se zavolá vždy, když bude nějaké zařízení nalezeno. V tomto callbacku se do modelu zařízení uloží základní info, tedy název zařízení, mac adresa a způsob získání. U

tohoto typu zařízení lze ještě říci, zda je s mobilním zařízením přes bluetooth spárováno a jakou má třídu (tj. počítač, mobil, audio, video). Tyto informace jsou uloženy do mapy s dodatečnými informacemi [2]. Hledání zařízení trvá klasicky podle dokumentace Androidu 12 vteřin. Pokud je nutné ukončit hledání dříve, lze tak učinit metodou `cancelDiscovery()`.

4.1.2.2 NearbyNetworksFinder - okolní Wi-Fi sítě

Přes systémovou službu `WIFI_SERVICE` lze v případě, že má uživatel zapnutou Wi-Fi, získat seznam okolních Wi-Fi sítí, na které se lze potenciálně připojit. O těchto sítích lze zjistit jejich identifikátory SSID a BSSID. Získání těchto informací je téměř instatní, tedy není nutné implementovat přerušující mechanismus.

4.1.2.3 SubnetDevicesFinder - zařízení na stejné Wi-Fi síti

V analýze byl rozebrán způsob, jakým lze zjistit rozsah IP adres sítě z IP adresy, která byla Wi-Fi sítí zařízení přidělena, a masky podsítě. Na těchto zařízeních se potencíálně nachází blízká okolní zařízení. Pokud je adresa v síti dostupná, lze zjistit mac adresu zařízení, které se na ni nachází, stejně tak i jeho název.

Tento finder si nejprve zjistí první adresu v síti, kterou je adresa sítě (network adress) a poté určí její rozsah. Tyto IP adresy si přidá do listu adres, u kterých bude zkoušet, zda jsou dostupné. V první řadě si však přidá adresy z ARP cache [39], ve které se nacházejí poslední dostupné IP adresy. Jejich dostupnost je vhodné prozkoumat jako první, protože pokud byly nedávno dostupné, budou pravděpodobně dostupné i nyní. Zařízení, u kterých je třeba zjistit dostupnost, může být mnoho, proto je vhodné problém paralelizovat. K tomuto účelu je vytvořen exekutor nad fixním thread poolem. Vlákna jsou tedy vytvořena dopředu a exekutor jim pouze postupně přiřazuje IP adresy, které se mají prověřit.

4.1.2.4 Další způsoby sběru dat o blízkých zařízeních

Při implementaci Nearby modulu byly vyzkoušeny i další přístupy sběru dat o blízkých zařízeních, při kterých byly využity Google FIT SDK [14] a Bluetooth LE API. Cílem bylo s využitím těchto technologií získat biometrická data o uživateli, například jeho srdeční tep. Framework měl tato data sbírat na pozadí bez dodatečné interakce uživatele ať s aplikací, nebo s blízkým zařízením. Také je žádoucí, aby implementace byla pro všechna blízká zařízení jednoho typu (např. fitness náramek) pokud možno stejná, neboť implementovat sběr dat pro každé zařízení zvlášť by bylo velmi náročné. Jak již bylo vysvětleno v popisu problematiky, tohoto bohužel není v současné době možno dosáhnout. Google FIT SDK však slibuje jednotné API alespoň pro podmnožinu náramků, proto bylo rozhodnuto jej zkusit použít.

Jelikož mají být kontextová data co nejaktuálnější, je vhodné použít Sensors API, které umožňuje čtení dat daného typu ze senzorů zařízení v reálném čase. Aby bylo možné Google FIT SDK v aplikaci použít, je nutné, aby vlastnila OAuth 2.0 klientský identifikátor. Ten lze vygenerovat v Google API konzoli. Aby konzole mohla přiřadit identifkátor ke konkrétní aplikaci, je třeba v procesu generování specifikovat tzv. SHA-1 (Secure Hash Algorithm 1)

otisk aplikace, který lze vygenerovat například v IDE Android Studio, a název hlavního balíčku aplikace. Kromě identifikátoru je také nutné aplikaci definovat práva na získání dat z tělesných senzorů, v případě srdečního tepu je to `android.permission.BODY_SENSORS`. Nyní je možné Google FIT SDK používat. V první řadě musí aplikace najít vhodné zdroje dat v okolním prostředí pro specifikovaný typ dat. V ukázce kódu C.2 je požadován datový typ `TYPE_BPM`, který odpovídá datům o srdečním tepu za minutu. Pokud jsou nějaké zdroje nalezeny, je potřeba zaregistrovat posluchače, který kontroluje, zda nejsou nějaká data dostupná s určitou frekvencí, konkrétně v kódu C.2 je tato frekvence jednou za deset vteřin. Pokud se vše podaří, jsou vývojáři data předána v `OnDataPointListeneru`.

Tento přístup byl vyzkoušen celkem se čtyřmi fitness náramky různých cenových kategorií. Jedná se o náramky TomTom Sports Touch, Cube 1, VeryFit 2.0 a Samsung Gear Fit2 Pro. Ačkoliv všechny tyto náramky umožňují měření srdečního tepu, bohužel ani jeden z náramků SensorsAPI nevyhodnotilo jako zdroj těchto dat. Po analýze problému bylo zjištěno, že Goolge FIT SDK při hledání dostupných zdrojů dat hledá zařízení, která se prezentují podle standardního GATT profilu [12] jako *Hearth rate monitor*, přičemž zmiňovaná zařízení se tímto způsobem pravděpodobně neprezentují.

Z tohoto důvodu bylo rozhodnuto nespoléhat na logiku implementovanou v Google FIT SDK a zkusit se na zařízení připojit ručně pomocí Bluetooth LE API [2]. Každé zařízení, které podporuje BLE hostuje GATT server, na který je možné se připojit a získat informace o službách, které zařízení poskytuje. Implementace byla provedena podle návodu v Android dokumentaci[1]. Opět bylo nutné aplikaci přidělit práva, v tomto případě se jednalo o práva `android.permission.BLUETOOTH` a `android.permission.BLUETOOTH_ADMIN`. Poté je provedeno vyhledání zařízení. Při nalezení zařízení je provolána callback funkce, ve které je třeba se připojit na GATT server zařízení. Po připojení by měl být dostupný seznam služeb, kterými zařízení disponuje. Služby jsou uloženy pod unikátními identifikátory, srdeční tep je uložen pod konstantou `SampleGattAttributes.HEART_RATE_MEASUREMENT`.

Bohužel ani v tomto případě zařízení nereagovala, respektive sken zařízení je neregistroval. Důvod tohoto problému nebyl ani po zdlouhavé analýze odhalen, je však zřejmé, že se jedná o problém s připojením k zařízením přes Bluetooth Low Energy. Jelikož získání srdečního tepu pro vytvoření prototypů není stěžejní, bude tento problém vyřešen v rámci budoucího vývoje.

4.1.3 Průběh sběru dat

Data se mají sbírat na pozadí. Všechny zmíněné minery a findery proto běží v AsyncTasku, což je Androidí mechanismus pro spouštění kódu na pozadí, díky čemuž je zajištěno, že hlavní vlákno, které má na statrosti UI, nebude blokováno. To je podmíněno tím, že kód spouštěný na pozadí nezasahuje žádným způsobem do UI. AsyncTasky jsou dva, jeden pro sběr informací o zařízení a druhý pro sběr dat o blízkých zařízeních.

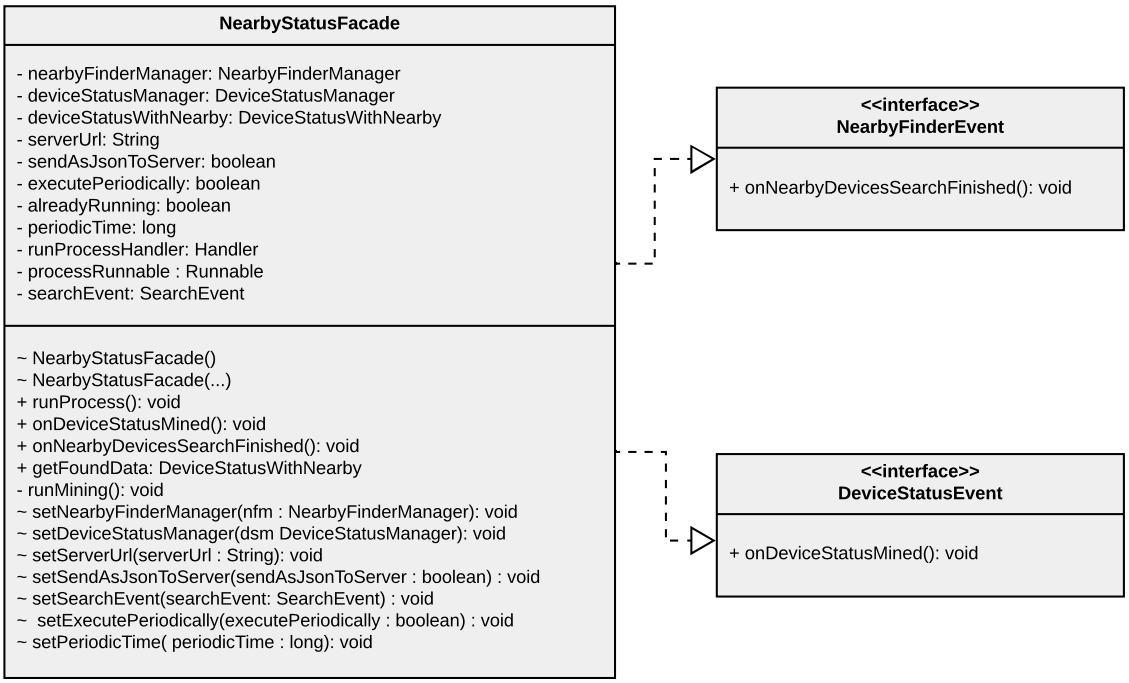
V požadavcích byla zmíněna možnost frameworku vypnout některé findery na základě kapacity baterie. Kapacita baterie se zjišťuje ve fázi sběru dat ze zařízení, je tedy nutné, aby sběr informací o samotném zařízení proběhl před hledáním blízkých zařízení. Podle kapacity baterie se findery vyfiltrují a zbydou jen ty, které se mají skutečně použít. Až poté by se mělo spustit vyhledávání blízkých zařízení.

[1] <`https://developer.android.com/guide/topics/connectivity/bluetooth-le`>

Jelikož musí sběry na sebe navazovat a oba probíhají ve zvláštní instanci AsyncTasku, bylo nutné tuto návaznost explicitně zajistit. AsyncTask sbírající stav zařízení po ukončení procesu volá abstraktní metodu `onDeviceStatusMined()`, v jejíž implementaci je spuštěn druhý AsyncTask, který hledá blízká zařízení. I na konci tohoto tasku se volá metoda `onNearbyDevicesSearchFinished()`, která značí, že proces sběru kontextových informací byl ukončen a je možné s daty pracovat. V případě, že uživatel definoval URL adresu endpointu, na který by se měla data poslat, po skončení tohoto procesu se data pošlou v dalším asynchronním tasku na server, kde se uloží.

4.1.4 Interakce s frameworkem

S frameworkem může vývojář interagovat pomocí fasády. Fasáda (Facade) je návrhový vzor, který poskytuje rozhraní pro ovládání složitějšího systému, čímž lze vývojáře odstínit od vnitřní implementace systému. Vytvořením takového rozhraní lze také omezit přístup k jednotlivým komponentám frameworku a zamezit tak, aby byly části frameworku používány jiným způsobem, než bylo zamýšleno[42]. Fasádu pro ovládání frameworku AFNearbyStatus lze nalézt v následujícím diagramu tříd 4.1.



Obrázek 4.1: Diagram tříd zachycující fasádu pro interakci s frameworkem AFNearbyStatus

V diagramu tříd 4.1 si lze povšimnout, že uživatel může spustit z fasády pouze proces

těžení dat a získat vytěžená data. Fasáda implementuje dvě rozhraní. První rozhraní je `DeviceStatusEvent` deklarující metodu `onDeviceStatusMined()`, která se volá při ukončení procesu získávání informací o zařízení. Fasáda v její implementaci začne proces získávání blízkých zařízení tak, jak je to popsáno výše. Druhé rozhraní funguje obdobně, tj. deklaruje metodu, která se zavolá po dokončení sběru informací o blízkých zařízeních. To značí, že jsou data připravena a jsou poslána na server, pokud vývojář specifikoval jeho adresu.

Fasáda je poměrně komplexní třída se spoustou vlastností, které se procesu sběru dat dají nastavit. Z 4.1 si lze povšimnout, že se tyto vlastnosti dají nastavit jen ze stejného balíčku, tj. konstruktor třídy a všechny settery jsou package-private. Třída fasády by totiž neměla být zvenčí nikdy přímo instanciována, ale vytvořena pomocí builderu. Builderu lze říci, jestli má být proces spoušten periodicky (popřípadě jak často), zda mají být data odeslána po skončení na server, jaké *minery* a *findery* se mají použít a také horní hranici doby, po které proces skončí. Také lze nastavit `SearchEvent`, který deklaruje dvě abstraktní metody, které se volají na začátku vyhledávání a na konci vyhledávání pro případ, že by chtěl uživatel frameworku v těchto dobách provést nějakou akci. Builder fasády je implementován jako singleton [42], což znamená, že existuje pouze jedna instance v systému. Důvodem použití singletonu je, že se proces sběru dat bude pravděpodobně spoustět z různých míst v aplikaci, přičemž nastavené vlastnosti se zřejmě nebudou příliš měnit. Nicméně možnost alternovat vlastnosti je ponechána, což má za následek to, že se musí vytvořit vždy nová instance fasády s případnými aktualizovanými vlastnostmi. Příklad použití je popsán v úryvku kódu C.1.

4.2 NSRest

Jak již bylo vysvětleno, aplikace NSRest uchovává nasbíraná kontextová data, které přijímá a poskytuje dále pomocí RESTových služeb. Příjímací endpoint je pouze jeden, který slouží pro uložení kontextových dat do systému. Očekává data serializovaná do formátu JSON ve stejné struktuře v jaké se ve frameworku AFNearbyStatus vytváří. Endpointů poskytujících data dalším aplikací je více. Jsou implementovány endpointy poskytující tato data:

1. Všechny záznamy pocházející od libovolného zařízení.
2. Všechny záznamy pocházející od konkrétního zařízení, které je určeno mac adresou.
3. Záznam, jehož doba získání je nejblíže dané časové značce od libovolného zařízení.
4. Záznam, jehož doba získání je nejblíže dané časové značce od konkrétního zařízení , které je určeno mac adresou.
5. Záznam, jehož doba získání je nejblíže dané časové značce od konktrétního zařízení, které je určeno mac adresou a s konktrétní akcí uživatele.
6. Poslední záznam od konkrétního zařízení, které je určeno mac adresou a s konkrétní akcí uživatele.

K posledním dvěma endpointům (tj. 5. a 6.) jsou vytvořeny ještě endpoint, který vrací pouze blízká okolní zařízení z tohoto záznamu, a endpoint, který vrací jméno uživatele, který aplikaci používal v momentě, kdy byl záznam pořízen. Všechny endpointy poskytují data ve

formátu JSON, kromě enpointu, který získává jméno uživatele, ten je vrácen v plain textu. Endpointy jsou veřejně dostupné, není tedy implementováno žádné zabezpečení. V případě, že kritériím nevyhovuje ani jeden ze záznamů kontextových dat, je vyhozena vlastní výjimka `NotFoundException`.

Aplikace je připojena na MongoDB, jak již bylo zmíněno dříve v textu. Pro připojení je třeba vytvořit mongo klienta, kterému je naspecifikována adresa a port, na kterém je databáze uložena. Standardní port pro MongoDB je 27017. V případě, že je databáze zabezpečená, je nutné ještě přidat uživatelské jméno, heslo a jméno autorizační databáze, vzhledem ke které se budou přístupová práva uživatele ověřovat. Aby správně fungoval převod Java POJO třídy do BSONu, což je vnitřní reprezentace dat v MongoDB, je nutné specifikovat sadu kodeků, podle kterých se třídy serializují. Java driver pro MongoDB disponuje funkcí pro automatické určení potřebné sady kodeků na základě cesty k balíčkům, kde se nachází POJO třídy. Toto připojení k MongoDB je v projektu implementováno jako singleton. Připojení tedy existuje pouze jednou a lze k němu staticky přistupovat z libovolného místa v aplikaci.

MongoDB používá pro identifikaci dokumentů `ObjectId`, které se skládá z důvodu zajištění unikátnosti ze čtyř částí - čas, identifikátor stroje, na kterém databáze beží, idetifikátor procesu a počítadla, které začíná na náhodné hodnotě [23]. Pokud `ObjectId` není explicitně určeno, je vygenerováno automaticky právě tímto způsobem. Všechny POJO třídy, které je žádoucí do databáze uložit, by měly toto id obsahovat, což lze konkrétně v tomto projektu zajistit děděním od třídy `MongoDocumentEntitiy`.

Aplikace má třívrstvou architekturu [52], tzn. je rozdělena do datové, servisní a prezentační vrstvy. Datová vrstva se stará pouze o data, tedy provádí dotazy nad databází a data nijak dále neupravuje. Implementuje se pomocí tzv. DAO (Data Access Object). Servisní vrstva data upravuje pro prezentační vrstvu, aplikuje na data business logiku a podobně. Prezentační vrstva se obvykle stará o zobrazení dat uživateli aplikace pomocí UI. Tato aplikace však slouží jen jako RESTové API a uživatelské rozhraní nemá, v tomto případě je prezentační vrstvou poskytování dat ve formátu JSON skrze endpointy.

4.3 Proxy aplikace

Tato sekce se zabývá implementací proxy aplikace. Nejprve bude popsána správa entit tj. aplikací, komponent, obrazovek a business případů v uživatelském rozhraní, poté vše, co se týče klasifikace jednotlivých komponent a využití kontextových dat k úpravě definic komponent. Proxy aplikace je podobně jako NSRest rozdělena do tří vrstev - datová, servisní a prezentační.

4.3.1 Správa entit v aplikaci

Proxy spravuje aplikace, obrazovky, které se v aplikacích vyskytují, komponenty na obrazovkách a business případy, které mohou v aplikacích nastat. Vše lze zobrazit a upravovat v grafickém uživatelském rozhraní. Toto rozhraní je implementováno pomocí Servlet API. Pro každou obrazovku UI byl vytvořen servlet, který disponuje metodami `doGet()` a `doPost()`. První metoda se používá ke zobrazení obrazovky, druhá při modifikaci obsahu, tj. například při odeslání formuláře. V aplikaci jsou tyto servlety:

- Servlet pro vytvoření a editaci aplikací (obrazovek, komponent, business případů a business fází).
- Servlet pro výpis seznamu aplikací (obrazovek, komponent, business případů a business fází), které se nacházejí v systému.
- Servlet pro konfiguraci důležitostí polí komponent v rámci business fáze.

4.3.1.1 Správa aplikací

U aplikací se ukládá název aplikace, její unikátní identifikátor a obrazovky, které se v ní vyskytují. Dále je nutné specifikovat URL adresy, na kterých jsou nasazeny aplikace NSRest poskytující kontextová data a backendová aplikace poskytující definice komponent (v tomto případě AFServer). Tyto URL adresy se poté používají ke správnému směrování HTTP požadavků. U komponent a obrazovek se z těchto URL adres skládájí endpointy, na kterých lze definice obrazovek a komponent nalézt. V případě, že uživatel změní adresy v modelu aplikace, je změna propagována do všech komponent a obrazovek, které tyto adresy používají.

Unikátní identifikátor se generuje automaticky při vytvoření aplikace a slouží k určení, ze které klientské aplikace jsou posílány požadavky. Klientská aplikace zašle tento identifikátor s požadavkem v hlavičce pod názvem `Application`. Na proxy je implementován filtr požadavků, který UUID z hlavičky získá, zjistí, zda takovou aplikaci spravuje, a pokud ano, nastaví tuto aplikaci do kontextu požadavku. Pokud hlavička Application chybí nebo aplikace nebyla nalezena, filtr požadavek zruší a zašle klientovi odpověď s kódem 400 - Bad request. UUID slouží pro zabezpečení definic obrazovek a komponent. Pokud totiž udělá klient požadavek na komponentu či obrazovku, která této aplikaci nepatří, je požadavek odmítnut.

4.3.1.2 Správa obrazovek a jejich komponent

U obrazovek se ukládá identifikátor, název obrazovky, který se bude na klientovi zobrazovat, její pořadí v menu a komponenty, které obsahuje. Při vytvoření obrazovky se jí přiřadí URL adresa, na které lze získat její definici. Obrazovka nemůže obsahovat dvakrát stejnou komponentu. Z jedné definice komponenty si vývojář může vytvořit komponent kolik chce, opakovat její definici by tedy nemělo smysl.

U komponent se definuje název komponenty, její typ (formulář, tabulka, list), endpoint, na kterém lze najít informace o tom, jaká pole obsahuje, a poté definice připojení na zdroje popisu komponent (v tomto konkrétním případě definice připojení na AFServer). Každý z popisů připojení (model, data, send) lze zapout či vypnout. Pokud je zapnutý, je nutné specifikovat zbytek URL adresy, na které se zdroj nachází, zbytek proto, že URL samotné aplikace se získá z modelu aplikace, do níž komponenta patří. Pokud zdroj vyžaduje parametry (např.content-type), lze je specifikovat v hlavičkových parametrech, autorizaci lze uvést v bezpečnostních parametrech.

Tato struktura duplikuje strukturu XML souboru, který sloužil k definicím zdrojů v původním řešení [55]. Díky specifikaci těchto připojení v modelu komponenty již není tento XML soubor potřeba. V analýze bylo zmíněno, že specifikace připojení na zdroje komponentových definic občas vyžadují doplnění některých parametrů či části URL klientskou aplikací.

V původním řešení existuje v klientských frameworcích interpretujících definice komponent mechanismus, který tyto proměnné umí nahradit. Jelikož klient disponuje tímto mechanismem a také jako jediný hodnotami, které mají proměnné nahradit, dává smysl, že nahrazování proměnných zůstane záležitostí klienta. Specifikace připojení pro každou z komponent se posílá na klienta s definicí obrazovky. Pokud se tedy ve specifikaci objeví proměnná (např. #{username}), klient ji doplní během vytváření builderu komponenty. Při sestavení komponenty tímto builderem se pošlou na proxy v hlavičce požadavku na získání komponenty už doplněné informace. V případě, že bylo nutné doplnit část URL, posílá se z klienta extra hlavička `real_endpoint` obsahující kompletní URL adresu, kterou proxy použije při získávání komponenty z backendové aplikace místo té, kterou má uloženou.

4.3.1.3 Správa business případů a jejich fází

Pro definici business případů je nutný název a popis. V business případech je možné definovat fáze, u kterých se definuje název fáze, obrazovky, které se jí týkají, algoritmus a konfiguraci, podle kterých se definice komponent na obrazovkách upraví. Po vytvoření fáze se pro všechny obrazovky, které byly připojeny, proiterují komponenty, které obsahují, a pošle se požadavek na endpoint, ze kterého lze získat informace o tom, jaká pole obsahují. Tento endpoint v aplikaci AFServer nebyl, musel být tedy doimplementován. Z těchto informací se vytvoří v systému seznam polí business případu (`BCField`). Uživatel je poté přesměrován na stránku, kde si lze nadefinovat pro každé pole jejich severitu, tj. účel a míru potřeby informace.

Když uživatel zobrazuje obrazovku, je nutné určit business případ a fázi, kterým obrazovka náleží, aby mohly být komponenty upraveny podle zvoleného algoritmu a konfigurace, které fáze obsahuje. K tomu je třeba vědět, v jakém stavu se klient nachází. Jelikož je však klient bezestavový, lze pouze určit, kterou obrazovku právě zobrazuje. To určení fáze mírně komplikuje, neboť v aplikacích se běžně vyskytuje stejná obrazovka v různých fázích nebo dokonce v různých business případech. Z názvu obrazovky by se tedy nedala jednoznačně určit fáze a případ, do které patří.

Jelikož má aplikace sloužit jen jako proof of concept, bylo pro ulehčení přidáno omezení, které určuje, že se obrazovka může nacházet pouze v jedné fázi jednoho business případu. Díky tomu lze snadno jednoznačně určit business případ a fázi. V případě, že by měla jít aplikace do produkce, bylo by nutné vymyslet a implementovat mechanismus, který by držel stav uživatele, např. stavový automat, ve kterém by se uživatel pohyboval pomocí svých akcí. Díky tomu, jak fungují stavové automaty, by se z principu uživatel nacházel vždy v právě jednom uzlu, který by příslušel konkrétní fázi v business případu. Takový stavový automat může být netriviální a k prezentaci myšlenky této práce by jeho tvorba nijak nepřispěla, proto byla zvolena možnost omezení obrazovky pouze na jednu fázi. Klient proto s požadavkem na získání komponent z proxy posílá identifikátor obrazovky, na které se nachází, v hlavičce `Screen`.

Poslední věc, kterou lze v uživatelském rozhraní proxy aplikace spravovat, jsou konfigurace limitů ohodnocení pro jednotlivé způsoby chování polí, které byly uvedeny v tabulce 3.3. Pro každý způsob chování je možné zadat spodní a horní limit, přičemž intervaly by se neměly překrývat, aby bylo možné jednoznačně chování určit. Platí, že čím větší ohodnocení, tím přísnější pravidlo [56]. Hierarchie způsobů chování je následující: `REQUIRED > VALIDATION >`

`ONLY_DISPLAY > HIDDEN > NOT_PRESENT`. Konfigurace se přiděluje k business fázi při jejím vytváření nebo editaci.

4.3.2 Klasifikační část

Klasifikační část je zodpovědná za ohodnocení polí komponenty a určení způsobů jejich chování. Skládá se ze tří na sebe navazujících částí, ohodnocovací, klasifikační a adaptační. Ohodnocovací vypočítává ohodnocení neboli skóre pole, klasifikační na základě tohoto skóre určí chování pole a adaptační podle chování přizpůsobí pole v komponentě. První dvě části si může uživatel navolit u definice business fáze v uživatelském rozhraní proxy aplikace, tj. zvolí si strategiialgoritmus, podle kterého se určí ohodnocení, nakonfiguruje důležitost a účel polí a zvolí konfiguraci a klasifikační jednotku, která použije vypočítané ohodnocení a zvolenou konfiguraci k určení chování pole. Poslední adaptační část je pro všechny situace stejná, jakým způsobem jsou pole upravena se lze dočíst v tabulce 3.3. Proces ohodnocení a klasifikace metamodelu je zachycen v sekvenčním diagramu B.7.

Klasifikační část je implementována třídou `AFClassification`, která disponuje jedinou veřejnou metodou `classifyMetaModel()`, která přijímá jako argumenty / objekt třídy `Client`, který obsahuje kontextová data, konfiguraci pro určení chování polí, list business polí, které se mají klasifikovat, model aplikace pro případ, že by bylo potřeba něco o aplikaci v průběhu klasifikace zjistit, a definici komponenty, která má být na základě klasifikace modifikována. Definice komponenty je deserializována a předávána jako objekt třídy `AFMetaModelPack`, která pochází z frameworku AFRest [55] a slouží právě k reprezentaci metadat o modelové třídě. `AFMetaModelPack` obsahuje popis třídy v objektu třídy `AFClassInfo`, jejíž struktura je popsána v diagramu tříd B.8.

Klient se musí před výpočtem naplnit kontextovými daty. Ty se získají z aplikace NSRest pomocí HTTP požadavku, konkrétně na endpoint, který vrátí nejbližší záznam k aktuální časové značce pro dané zařízení nezávisle na akci uživatele, který by měl být velmi podobný aktuálnímu kontextu. Měření kontextu probíhá v krátkých intervalech (například 2-3 minuty) a ve většině situací můžeme předpokládat, že se během této doby uživatelův kontext moc nezměnil. Samozřejmě v případě, že se bude u používání aplikace uživatel pohybovat, nelze toto s jistotou předpokládat. Takové případy lze řešit zkrácením intervalů měření. Ideální by bylo měřit kontext v reálném čase, ale to bohužel není možné z důvodů, které byly rozebrány dříve v této práci. Získaná kontextová data je třeba převést z JSON formátu do modelu klienta. K tomuto účelu byl vytvořen speciální parser `JsonContextParser`.

Třída `AFClassification` poté obsahuje jako atributy ohodnocovací a klasifikační jednotku, které je nutné před výpočtem nainicializovat. Jak již bylo popsáno, ohodnocovacích a klasifikačních jednotek může být navrženo mnoho. Každý z ohodnocovacích modulů musí implementovat rozhraní `Scoring` a klasifikační moduly rozhraní `Classification`. Při inicializace třídy `AFClassification` musí být na zakladě toho, co si uživatel vybral v uživatelském rozhraní, vybrána konkrétní implementace těchto modulů. K tomu se hodí návrhový vzor Factory [50].

4.3.2.1 Klasifikační a ohodnocovací jednotky

Z diagramu B.6 lze odvodit, které konkrétní třídy v systému implementují ohodnocovací a klasifikační rozhraní. Instance konkrétní jednotky je vybrána na základě výčtových typů,

které lze najit v pravé horní části diagramu.

Klasifikační jednotka má nyní dvě implementace. První implementace určuje chování pole na základě ohodnocení a konfigurace, které jsou předány jako parametry. Druhá implementace vrací vždy, bez ohledu na skóre, chování `REQUIRED`.

Ohodnocovací jednotka má aktuálně tři implementace. Nejednodušší je `BaseScoringUnit` označovaná výčtovou konstantou `BASIC`, která pro téměř všechna pole vrátí skóre 80. Maximální skóre 100 přidělí polím, které mají nastavený učel `SYSTEM_IDENTIFICATION` a míru potřeby informace `CRITICAL` [56].

Druhou implementací je `BatteryConnectionScoringUnit`, které přísluší výčtová konstanta `BATTERY_AND_CONNECTION_SCORING`. Jak už název napovídá, tato implementace bere v potaz kapacitu baterie a typ připojení. Obdobně jako základní implementace přiřazuje polím označeným jako `SYSTEM_IDENTIFICATION` a `CRITICAL` nejvyšší možné ohodnocení [56]. Pokud je zařízením telefon nebo tablet, je pro každé pole vypočítána hodnota podle ohodnocení účelu a míry informace, které si ohodnocovací jednotka drží v hashmapách. Pokud má pole navíc účel `INFORMATION_MINING` a míru potřeby informace `NICE_TO_HAVE` nebo `NEEDED`, je toto základní skóre modifikováno na základě kapacity baterie a typu připojení. Čím méně baterie zařízení má, tím méně jsou pole, sloužící pro těžbu infomací, potřebná, jelikož podle tabulky 3.1 nejsou kritické pro dokončení úkolu v systému. Od skóre je tak odečtena tím vetší hodnota, čím menší kapacitu baterie uživatel má. Pokud se však baterie nabíjí, není odečteno nic, neboť kapacita baterie v tomto případě už není relevantní, jelikož se zařízení už nevybíjí a není třeba na dokončení úkolu spěchat. Co se týče připojení, tato ohodnocovací jednotka rozlišuje Wi-Fi a mobilní připojení. Mobilní připojení je často méně kvalitní a uživatelé často nemají moc dat, takže je žádoucí opět modifikovat skóre směrem dolů. V případě, že je uživatel na Wi-Fi, se naopak něco ke skóre přidá, neboť není důvod, aby tato pole nevyplnil.

Třetí a poslední implementací je `NearbyDevicesScoringUnit` pod výčtovou konstantou `NEARBY_DEVICE_SCORING`. Jako všechny implementace přiřazuje polím označeným jako `SYSTEM_IDENTIFICATION` a `CRITICAL` nejvyšší možné ohodnocení [56]. Stejně jako druhá implementace, počítá skóre pouze pro mobilní zařízení a tablety. Cílem této ohodnocovací jednotky je porovnat aktuální okolní zařízení se zařízeními, které kolem sebe měl uživatel naposledy, když prováděl stejnou akci v systému. Seznam okolních zařízení získá z aplikace NSRest pomocí HTTP požadavku na endpoint poskytující okolní zařízení posledního známého záznamu pro dané zařízení a akci uživatele. Na základě porovnání těchto dvou kontextů jednotka usoudí, jestli se jedná o stejného uživatele, a polím, jejichž účel je `SYSTEM_INFORMATION` nebo `INFORMATION_MINING`, sníží skóre podle toho, jak moc jsou množiny zařízení podobné. Podobnost se určuje následovně. Nejdříve se porovná, zda se jedná o stejné uživatele, personalizace uživatelského rozhraní pro jednoho uživatele na základě kontextu jiného uživatele by totiž nedávala smysl. Pokud se jedná o stejného uživatele, vypočítá se, kolik procent okolních zařízení je stejných. Podle tohoto čísla můžeme určit jeden z šesti intervalů definovaných ve výčtovém typu `NearbyDeviceSetupSimilarity`, které jsou popsány v tabulce 4.1. Pro tyto intervaly jsou v hashmapě `rankedNearbyDeviceSetup` definovány hodnoty, které se v případě zvolení intervalu odečtou od skóre.

4.3.2.2 Adaptace polí

Adaptace pole podle zvoleného chování probíhá následovně. Nejdříve je třeba získat z metamodelu, který obsahuje popis celé modelové třídy, správnou definici třídy, ve které se pole nachází. Metamodel totiž obsahuje kromě hlavní třídy i třídy vnitřní. Například v případě, že třída `Person` obsahuje jako atribut adresu typu `Address`, bude v metamodelu třídy Person adresa popsána jako vnitřní třída. Pole se může nacházet tedy v této vnitřní třídě nebo i ve vnitřní třídě vnitřní třídy atd., proto `AFClassification` implementuje rekurzivní metodu, která nalezne, v jaké definici třídy se pole nachází. Pokud je pole zanořené ve vnitřní třídě, při jeho ukládání do databáze proxy aplikace během procesu konfigurace polí, které popsáno v předchozí sekci, je jeho jméno upraveno tak, aby popisovalo, kde a jak moc je zanořeno, a to pomocí tečkové notace [55]. Například pokud existuje pole `street`, které se nachází v modelu `Address`, která je vnitřní třídou, bude pole uloženo jako `address.street`. Poté, když se hledá správný popis třídy v metamodelu (v tomto případě by to měl být právě popis třídy `Address`), rekurzivní metoda ví podle množství teček, kolikrát se bude muset zanořit, a podle názvů tříd, které jsou před tečkami ví i do jaké vnitřní třídy.

Po nalezení popisu správné třídy, která pole obsahuje, je možné pole upravit, podle pravidel v tabulce 3.3. Toto se provede pro každé pole. Výsledkem je upravený metamodel, který se serializuje do JSONu a pošle na klienta, kde klient sestaví komponentu.

Tabulka 4.1: Intervaly podobnosti množin okolnich zařízení

Konstanta	**Podobnost**	**Popis**
SAME	90%-100%	Zařízení jsou shodná či téměř shodná, uživatel se nachází ve stejném kontextu.
MOSTLY_SIMILAR	70% - 90%	Zařízení jsou velmi podobná, pravděpodobnost, že se uživatel nachází ve shodném kontextu je velmi vysoká.
MORE_SIMILAR	50% - 70%	Zařízení jsou víceméně podobná, pravděpodobnost shodných kontextů je již velice malá.
MORE_DIFFERENT	30% - 50%	Zařízení jsou spíše rozdílná, pravděpodobnost rozdílného kontextu stoupá.
MOSTLY_DIFFERENT	10% - 30%	Zařízení jsou velmi rozdílná, pravděpodobnost rozdílného kontextu je vysoká.
DIFFERENT	0% - 10%	Zařízení se vůbec či téměř vůbec neshodují, uživatel se nachází v jiném kontextu.

4.3.3 RESTové rozhraní

Jak již bylo zmíněno, proxy komunikuje s backendovou aplikací AFServer a aplikací NSRest pomocí HTTP požadavků. Pro komunikaci s klientskými aplikacemi je vystaveno RESTové API. Klientská aplikace potřebuje z proxy získávat definice obrazovek a komponent. Jsou tedy implementovány dvě skupiny endpointů. Enpointy pro obrazovky umožňují získat definici menu a definici obrazovky podle klíče nebo identifikátoru, které obrazovka obsahuje.

Pro účely popisu menu byl implementován speciální model `MenuItem`, který popisuje jednu položku menu. Obsahuje klič obrazovky, podle které lze obrazovku identifikovat, text určený k zobrazení, URL adresu endpointu na proxy, z něhož je možné získat definici příslušné obrazovky, a pořadí, které bude položka v menu mít.

Co se týče komponent, jsou definovány tři endpointy - první pro získání definice komponent, druhý pro získání dat komponenty a třetí pro zaslání dat z formuláře, přičemž kontextová data využívá pouze první. V případě datového endpointu se data, která se získají z backendové aplikace AFServer, nijak nemění. Stejně tak při posílání dat z formuláře pošle proxy data na AFServer v nezměněném stavu.

Všechny endpointy produkují výstup ve formátu JSON, endpoint pro zaslání dat z formuláře navíc očekává v tomto formátu i vstup.

4.4 Úprava existujících řešení

V předchozím textu bylo popsáno, jakým způsobem se musí adaptovat existující řešení, aby splňovalo nové požadavky a zároveň byla původní funkcionalita zachována. Pro rekapitulaci: Musí být upraveny klientské frameworky AFAndroid a AFSwinx, aby uměly interpretovat popisy menu a obrazovek. Dále je nutné upravit klientské aplikace, aby tuto novou funkcionaitu frameworků využívaly pro stavbu uživatelského rozhraní. Také se musí nahradit v inicializaci builderu XML soubor s popisy připojení na zdroje informací o komponentách za popisy připojení, které se nyní posílají z proxy. V neposlední řadě se musí mírně upravit i backendová aplikace AFServer, neboť dříve bylo popsáno, že proxy aplikace potřebuje ke konfiguraci business vlastností polí komponent vědět, jaká všechna pole se mohou v komponentě nacházet. Pro tyto účely bylo zvažováno, že se použije edpoint poskytující popis komponenty. Ten je ale nevyhovující, neboť pro různé role v systému může tento endpoint vrátit definici komponenty s různou množinou polí. Pro každý model na serveru je tedy potřeba vytvořit endpoint, který seznam těchto polí poskytuje.

4.4.1 Úprava AFAndroid a AFSwinx

Pro účely interpretace menu byl vytvořen v obou frameworcích `AFMenuBuilder`. Ten na základě definice menu, kterou získá z prox, vytvoří objekt `AFMenu`. `AFMenu` obsahuje mapu, která by měla obsahovat jednotlivá tlačítka menu. Ty je opět potřeba nejdříve vyrobit. O to se stará `AFScreenButtonBuilder`, který vytváří tlačítka následujícím způsobem. Vytvoří objekt třídy `AFScreenButton`, který v případě AFSwinx dědí od třídy `JButton` a v případě AFAndroid od třídy `AppCompatButton`, které standardně reprezentují tlačítka v uživatelském rozhraní. To umožní nakládat s tlačítky klasickým způsobem, například je možné upravit jejich vzhled, rozměry, zarovnání a podobně. Tlačítkům je dále nastaven klíč obrazovky, na kterou odkazují, pořadí v menu, URL adresa endpointu na proxy, z něhož lze získat definici obrazovky, na kterou odkazuje, a text tlačítka.

Po kliknutí na tlačítko by se měl podle předchozího textu zpřístupnit popis obrazovky s předpřipravenými buildery komponent. Frameworky tedy tlačítku nastaví i posluchač kliknutí, ve kterém se tato definice obrazovky pomocí `AFScreenDefinitionBuilderu` stáhne a sestaví do modelu `AFProxyScreenDefinition`. Do definice obrazovky se během tohoto procesu nastaví klíč obrazovky a její URL adresa na proxy. Poté se pro všechny komponenty,

které má obrazovka obsahovat, vytvoří objekt třídy `AFProxyComponentDefinition`, do kterého se vloží informace o komponentě, jako je její název nebo typ a nainicializuje se její builder, ze kterého půjde komponenta vytvořit.

Původně se při inicializaci builderu udával XML soubor, ze kterého se pomocí XML parseru vyextrahovaly informace o tom, kde se nachází zdroj informací o komponentě. Nyní se při inicializaci uvádí `JSONObject` s popisem těchto zdrojů, který je součástí definice obrazovky, která byla stažena z proxy. Místo XML parseru je pro zpracování tohoto popisu připojení implementován JSON parser. Použití tohoto parseru a následné reálné stažení definice komponenty se provede, až když se vývojář používající tyto frameworky rozhodne komponentu z předpřipraveného builderu sestavit a vložit do UI.

Na konci přípravy popisu obrazovky je potřeba předat popis s přednastavenými buildery vývojáři. K tomuto účelu byl vytvořen `ScreenPreparedListener`, který deklaruje abstraktní metodu, která se zavolá v momentě, kdy je obrazovka připravena. Když tuto metodu vývojář implementuje, je mu v argumentech poskytnuta celá definice obrazovky, včetně builderů, které může použít k sestavení obrazovky. Definice obrazovky disponuje metodami, které usnadní získání konkrétního builderu komponenty. Kromě toho disponuje metodou, která z builderů postaví všechny komponenty a vrátí je v podobě listu. Uživatel frameworku tedy nemusí znát konkrétní komponenty, které se na obrazovce nacházejí, ale jednoduše tímto způsobem komponenty získá, proiteruje a vloží do UI. Nevýhodou tohoto přístupu je, že komponenty vyžadují různé parametry připojení. Pokud některý parametr připojení chybí, není definice připojení validní a komponentu nelze postavit, neboť se nepodaří získat její definici z proxy aplikace. Další nevýhodou je, že pokud se uživatel rozhodne vložit komponenty hromadně, nelze jim změnit vzhled pomocí skinů, neboť skiny musí být dostupné před začátkem procesu stavění komponenty [55]. Naopak nespornou výhodou tohoto přístupu je, že bude klientská aplikace pružně reagovat na změny množiny komponent obrazovky na proxy, tj. pokud je u obrazovky přidána nebo odebrána komponenta, je tato změna ihned propsána do UI klientské aplikace. Také obsahuje metodu `reload()`, která umožní znovunačtení celé definice obrazovky z proxy.

V případě, že by uživatel frameworku chtěl provést po kliknutí na tlačítko kromě sestavení obrazovky ještě jinou akci, framework poskytuje možnost nastavit na `AFScreenButtonu` vlastní posluchač události. Po dokončení tvorby tlačítka menu je vloženo pod klíčem obrazovky, na kterou odkazuje do mapy, která je atributem třídy `AFMenu`. Vývojáři je předán objekt této třídy, ze které lze tlačítka vyextrahovat a vložit do grafického uživatelského rozhraní.

Nové buildery jsou zpřístupněny skrz fasádu frameworku. Lze získat builder celého menu, ale i zvlášť builder definice obrazovky a builder tlačítka pro případ, že by bylo třeba tyto dvě části získat samostatně mimo proces tvorby menu.

Před spoluprací s proxy aplikací je třeba nejprve na straně klientské aplikace nakonfigurovat atributy, které klient využívá při komunikaci s proxy aplikací. Mezi tyto atributy patří například URL adresa, na které je proxy nasazená, nebo identifikátor aplikace, pod kterým se aplikace v požadavcích na proxy identifikuje. Frameworky pro interpretaci UI komponent, které klientské aplikace využívají, mají implementovánu logiku, která s těmito atributy pracuje. Klientská aplikace musí těmto frameworkům hodnoty atributů poskytunout. Pro tyto účely disponují frameworky abstraktní třídou `UIProxySetup`, která deklaruje abstraktní metody pro získání těchto potřebných informací. V klientských aplikacích je třeba z této třídy

dědit a tyto metody implementovat. Mezi informace, které je nutné definovat, patří

- URL adresa, na které je nasazená proxy aplikace,
- unikátní identifikátor sloužící pro autorizaci požadavků na proxy,
- typ zařízení (MOBILE, TABLET, PC, OTHER).

Pokud bude aplikace chtít využívat kontextových dat, tak je potřeba také

- identifikátor zařízení (mac adresa),
- URL adresa, na které je nasazená aplikace NSRest.

V příloze C.3 lze nahlédnout na příklad, jak může konfigurace komunikace s proxy aplikací vypadat. Tento úryvek kódu popisuje konkrétně konfiguraci pro aplikace využívající AFAndroid. Konfiguraci je potřeba nastavit do fasády frameworku pomocí metody `setProxySetup()`.

Nakonec je třeba zmínit úpravu posílání HTTP požadavků. Proxy aplikace očekává hlavičky `Application` s UUID aplikace, `Screen` s aktuální obrazovkou, kterou uživatel právě zobrazuje, `real_endpoint` obsahující kompletní URL adresu endpointu aplikace AFServer, kde lze získat definici komponenty. Význam a důvod potřeby těchto hlaviček byl již popsán dříve v textu. HTTP požadavky, které frameworky posílají, byly o tyto hlavičky doplněny.

4.4.2 Úprava klientských aplikací

Původně se frameworky AFAndroid a AFSwinx využívaly jen pro tvorbu komponent UI. Nyní je nutné aplikace upravit tak, aby automaticky vytvářely celé obrazovky a menu. Proces tvorby menu probíhá následovně. Nejprve se získá z fasády frameworku menu builder, který menu sestaví, což popisuje úryvek kódu 4.1.

Listing 4.1: Tvorba menu s použitím frameworku AFAndroid

```
1 AFMenu afmenu = AFAndroid.getInstance()
2                           .getMenuBuilder(getContext())
3                           .setUrl(getProxyUrl() + "/api/screens")
4                           .buildComponent();
```

Předtím, než jsou tlačítka vloženy do UI, je nutné jednotlivým tlačítkům nastavit posluchače, jejichž kód se provede po kliknutí na tlačítko po tom, co je obrazovka připravena. V těchto posluchačích je dostupná definice obrazovky se všemi buildery komponent, které se mají v obrazovce nacházet. Inicializaci posluchače je možné nalézt v úryvku kódu 4.2.

Listing 4.2: Inicializace ScreenPreparedListeneru s použitím frameworku AFAndroid

```
1 AFAndroidScreenPreparedListener someListener =
2         screenDefinition -> showScreen(screenDefinition);
3 Map<String, AFScreenButton> menuButtons =
4         afmenu.getMenuButtons();
5 menuButtons.get(SOME_BTN_KEY)
6         .setScreenPreparedListener(someListener);
```

Buildery komponent, které popis obrazovky obsahuje, je třeba použít pro vytvoření a vložení komponent do uživatelského rozhraní. Na konci kódu posluchače je obrazovka zobrazena uživateli. Samotný kód vytvoření komponent musel být také mírně upraven. Zatímco v původním řešení [47] se buildery komponent získávaly z fasády `AFAndroid`, nyní se musí získávat z definice obrazovky. Ukázka tvorby formuláře je v úryvku kódu 4.3. Obdobně to funguje pro tabulku i list.

Listing 4.3: Aktuální způsob vytváření komponenty s použitím frameworku AFAndroid

```
1  public void showScreen(AFScreenDefinition screenDefinition) {
2          HashMap<String, String> connectionParams
3                  = new HashMap<>();
4          connectionParams.put("username", "sa2");
5          //build form from screen definition
6          AFForm form = screenDefinition
7                  .getFormBuilderByKey(SOME_FORM)
8                  .setConnectionParameters(connectionParams)
9                  .setSkin(new SomeFormSkin(getContext()))
10                 .createComponent();;
11         //insert form into screen and display it
12 }
```

Pro framework AFSwinx a její ukázkovou Java SE aplikaci jsou zmiňované úryvky kódu téměř totožné. V tomto duchu byly upraveny všechny obrazovky a tvorby komponent v obou ukázkových aplikacích využívající frameworky AFAndroid nebo AFSwinx. Dále byly smazány XML soubory s popisem připojení na zdroje komponent, jelikož již nejsou potřeba.

Kapitola 5

Testování

Vývojáři se při implementaci aplikace mohou dopustit chyb, které mohou vést k neočekáváným výsledkům nebo dokonce způsobit selhání systému. Z tohoto důvodu je potřeba software testovat. Druhů testů je mnoho, lze je rozdělit do dvou hlavních skupin, automatické a manuální. Mezi automatické testování patří například jednotkové testy nebo testy regresní, které testují, zda přidáním nové funkcionality nevznikly chyby v již testované funkcionalitě. Manuální testy se často týkají testování použitelnosti aplikace, která se většinou týká uživatelského rozhraní. Lze mezi ně zařadit například testování s uživatelem, kdy je uživateli předložena sada úkolů, u jejichž plnění uživatele pozorujeme, nebo A/B testování, které statisticky porovnává několik různých variant uživatelského rozhraní.

Testování zkoumá, zda systém splňuje požadavky a pracuje tak, jak bylo popsáno v požadavcích. K tomu využívá mnoho různých metod, které mohou být prováděny v různých částech vývoje softwaru. Většinou se však v tradičních aplikacích vkládá nejvíce úsilí do testování po tom, co jsou jasně definované požadavky aplikace a probíhá nebo je dokončena implementace [59].

Jelikož aplikace a frameworky vytvořené či upravené v tomto řešení slouží jen jako proof of concept, testování proběhlo pouze v omezené míře. Bylo vytvořeno několik referenčních unit testů, které používají testovací framework JUnit. Také byla provedena statická analýza kódu. Dále byla funkcionalita, která byla vytvořena v rámci této práce, otestována v praxi na ukázkových aplikacích.

5.1 Unit testy

Unit neboli jednotkové testy slouží k otestování jednotlivých částí softwaru. Granularita unit testů by měla být vysoká, testované části by měly být co nejmenší. Z pohledu objektově orientovaného programování bývá jednotkou obvykle třída či metoda.

Testy by měly být prováděny pro různé vstupy. Kompletní pokrytí vstupů a všech případů, které mohou nastat, není reálné, neboť by to bylo velmi časově náročné. Proto se určují pouze krajní hodnoty různých případů, pro které se testy provedou. Pokud je potřeba spusit stejný test na množině různých dat, využívají se tzv. parametrické testy.

Každý testovací případ by měl být nezávislý na ostatních a pokud možno izolovaný od ostatních částí systému. Často je ale nutné otestovat funkcionalitu, kterou lze provést

jen v určitém kontextu. Ve vztahu k této práci jsou to například metody, které získávají data ze senzorů o zařízení, a správně fungují pouze na Android zařízení. Aby bylo možné tuto funkcionalitu otestovat i bez použití emulátoru nebo reálného zařízení, je nutné tyto metody simulovat. K tomu se používá tzv. mockování. V této práci je použit konkrétně framework Mockito [20] a Robolectric [25]. Robolectric je framework, který poskytuje kopie tříd z Android SDK, které jsou jinak dostupné jen při běhu aplikace na konktrétním zařízení nebo emulátoru. Takovými třídami jsou například systémové služby jako `WifiManager` nebo `BluetoothAdapter`. Tyto kopie jsou poskytnuty místo originálu pomocí frameworku Mockito. Díky tomuto nahrazování je možné funkcionalitu otestovat přímo v IDE i bez spuštění aplikace.

V této práci byly provedeny tyto ukázkové testy, přičemž testy 2 až 5 využívají mockovacího mechanismu, který byl popsán výše, a test číslo 6 je parametrický:

1. Test úpravy metamodelu komponenty na základě daného chování (Behavior), tj.
   (a) vymazaní pole z metamodelu - odpovídá chování NOT_PRESENT,
   (b) odstranění required validace - odpovídá chování VALIDATION,
   (c) odstranění všech validací - odpovídá chování ONLY_DISPLAY, HIDDEN,
   (d) zneviditelnění pole - odpovídá chování HIDDEN,
   (e) nastavení required validace - odpovídá chování REQUIRED.
2. Test získání informací o poloze a rychlosti zařízení - skrz GPS i skrz síť, s uděleními právy i bez.
3. Test získání informací o baterii zařízení.
4. Test získání informací o rozlišení zařízení a výpočtu palců z šířky a výšky displaye.
5. Test získání informací o aktuálním stavu uživatele v aplikaci.
6. Test získání informací o připojení - zvlášť pro mobilní a Wi-Fi připojení.
7. Parametrický test na kontrolu formátu IP adresy.
8. Test na získání následující IP adresy.
9. Test na získání množství IP adres z masky sítě.

5.2 Statická analýza kódu

Všechna vytvořená i upravovaná řešení byla podrobena statické analýze kódu pomocí nástroje FindBugs [10]. Tento nástroj analyzuje kód aplikace bez toho, aby jej spouštěl, a nalezne místa, kde může potenciálně vzniknout chyba.

V aplikacích bylo nalezeno několik chyb různých typů, některé jsou nyní uvedeny.

Internationalization

Tato chyba vzniká, když se část kódu spoléhá na defaultní kódování, tj. například při tvorbě objektů třídy `InputStream` nebo při převodu pole bytů na `String`. Tato chyba byla opravena tak, že bylo na všech takových místech přidáno kódování UTF-8.

Malicious code vulnerability

Tento problém se konkrétně objevil v geterech a seterech na atributy typu `Date`. `Date` je totiž mutable. Může se tedy stát následující situace. Datum je nasetováno do objektu třídy a hned poté na dalším řádku kódu změněno. Díky tomu, že je mutable, změní se i datum v objektu třídy, kam bylo předtím nastaveno, což je většinou nežádoucí situace. Opravou tohoto problému je na všech místech, kde by se toto mohlo stát, z objektu data vytvářet objekt nový, neboli vytvořit tzv. deep kopii objektu.

Další situace, kdy tato chyba nastala, bylo opomenutí klíčového slova u public static konstanty. FindBugs v tomto případě varuje předtím, že by statické pole mohlo být změněno. U konstant, kterých se to týkalo, bylo toto klíčové slovo doplněno.

Performance

Performance bug naznačuje, že je část kódu neefektivní. Nejedná se tedy o chybu, ale spíše o doporučení. V aplikacích se tento problém vyskytl u způsobu, jakým se iterovala hashmapa. V kódu se iterovalo přes množinu klíčů mapy a hodnoty se získávaly pomocí metody `map.get(key)`. To je podle FindBugs neefektivní a mělo by se iterovat vždy, když je potřeba klíč i hodnota, přes `entrySet`, což je množina objektů obsahující pár klíč-hodota. Dalším případem je vnitřní třída, která by podle nástroje měla být vždy, když je to možné, statická.

Dodgy code

Tato chyba se objevila v případě, že se v kódu vyskytovala například lokální proměnná, do které se něco přiřadilo, a poté již nebyla využita (tzv. dead local store), nebo dvě větve v podmínkách dělaly to samé. Tyto chyby vznikly spíše nepozorností a byly velmi jednoduše odstraněny.

Bad practice

Tímto nástroj FindBugs upozorňuje na části kódu, kde bylo něco opomenuto. Například když třída implementuje rozhraní `Serializable` a třídy neprimitivních atributů, které třída má, toto rozhraní neimplementují, jedná se o bad practice, neboť v případě, že by se třída serializovala, by došlo k chybě.

Multithreaded correctness

Tento typ chyby se do projektu dostal pouze nedopatřením, neboť na místě, kde se tato chyba objevila, nebylo použito více vláken. Původem chyby bylo nadbytečné klíčové slovo synchronized v deklaraci metody, což byla copy-paste chyba.

5.3 Ukázkové projekty

Tato sekce popisuje demonstraci funkcionality řešení na ukázkových projektech. Nejdříve bude ukázáno, jakým způsobem lze definovat obrazovky a komponenty v proxy aplikaci. Poté bude předvedena specifikace business případů a fází. Jedné z fází byla přiřazena jedna či více obrazovek, jejichž komponenty by měly být upraveny na základě kontextových dat s využitím ohodnocovacích a klasifikačních jednotek definovaných u této fáze. Výsledky klasifikace a změny v UI budou názorně ukázány na Android klientské aplikaci.

5.3.1 Stručný popis klientských aplikací

Klientské aplikace slouží jako frontend k aplikaci AFServer, což je systém pro zadávání absencí. AFServer disponuje zdroji, ze kterých lze získat popisy komponent UI a jejich data, kterými se komponenty naplní. Také umožňuje vytváření či úpravu těchto dat v databázi. Některé ze zdrojů jsou zabezpečené a přístupné pouze pod určitou rolí, například administrátorem. Skrz uživatelské rozhraní klientských aplikací lze provádět následující akce. Uživateli je umožněno vytvořit absenci určitého typu, který lze libovolně vytvářet a upravovat. Typy absencí se liší podle země, do které uživatel spadá. K dispozici je tedy nejen správa absenčních typů, ale i možnost vytvářet či upravovat dostupné země, ke kterým se typy absencí řadí. Vytvořená absence vzniká jako žádost a je potřeba ji schválit, což může udělat jen uživatel, který má na to práva. Takový uživatel může absenci i zamítnout či zrušit. Uživatelům je také dovoleno si zobrazit a upravit svůj osobní profil. Do aplikace od původního řešení přibyla ještě správa služebních cest. Cesty podobně jako absence vznikají jako žádost a je potřeba je schválit či zamítnout. Také se dá služební cestě přiřadit stav *probíhá*, ve kterém je možné do cesty přidávat její části nebo zastávky. Služebním cestám lze přidat vozidlo, proto do aplikace přibyla ještě správa aktuálně dostupných vozidel.

Všechny komponenty v klientské aplikaci jsou interpretovány a vytvořeny za pomoci AFSwinx nebo AFAndroid z definic komponent, které AFServer generuje a poskytuje skrz webové rozhraní. Jak už bylo dříve zmíněno, tento proces byl modifikován a pro zachování stejné funkcionality bylo potřeba klientské aplikace a frameworky upravit. Cílem bylo, aby klientské aplikace fungovaly stejným způsobem jako před úpravou. V původní verzi bylo menu aplikace vytvořeno manuálně, nyní se definice menu generuje v proxy aplikaci a klientská strana z ní staví tlačítka menu. Vygenerované menu lze zhlédnout na snímku obrazovky B.9. Do aplikace, jak bylo zmíněno výše, byla přidána také správa vozidel a služebních cest. Tyto přidané obrazovky si lze prohlédnout v přiložených obrázcích B.10 a B.11. Obdobně byly tyto změny provedeny v ukázkové Java SE aplikaci, snímky obrazovek lze nalézt ve zdrojových kódech této práce (viz. *Obsah přiloženého CD*).

5.3.2 Experiment

Jak je již známo, proxy aplikace spravuje klientské aplikace, jejich obrazovky a komponenty a také business případy, které v nich mohou nastat. To vše lze provádět v uživatelském rozhraní. Část z něj týkající se správy obrazovek lze najít v příloze B.12. Odtud jsou patrné všechny obrazovky, které mají být v klientských aplikacích, včetně komponent, které obsahují, a pořadí obrazovek v menu klientské aplikace, přičemž klientské aplikace tuto strukturu

duplikují. Pro demonstraci úpravy definic komponent s využitím kontextových dat byl proveden experiment. Na základě kontextových dat a nastavení v proxy aplikaci by na konci experimentu měla vzniknout upravená komponenta.

5.3.2.1 Příprava experimentu

V rozhraní proxy aplikace je potřeba provést následující akce, které zajistí, že se na obrazovku klientské aplikace a její komponenty aplikují kontextová data:

1. Nadefinovat obrazovku a její komponenty, které se k obrazovce přiřadí.
2. Vytvořit business případ.
3. Vytvořit konfiguraci jednotlivých chování.
4. Vytvořit v business případu business fázi a zvolit u ní konfiguraci, klasifikační a ohodnocovací jednotku.
5. Přiřadit business fázi jednu či více obrazovek.
6. Nakonfigurovat business vlastnosti polí jednotlivých komponent, které se v přiřazené obrazovce nacházejí.

První krok je již hotový, neboť bylo potřeba tento krok udělat pro obnovení původní fukcionality klientských aplikací. Pro splnění druhého kroku je potřeba vyplnit formulář B.13. Mějme business případ, který se týká správy profilu uživatele. Business případ se po vytvoření objeví v seznamu business případů. Po kliknutí na tlačítko Phases je možné konfigurovat jeho fáze. Před vytvořením fáze je však nutné mít vytvořenou konfiguraci chování, kterou lze provést v sekci Configurations. Při přidávání nové konfigurace se limity pro jednotlivá chování předvyplní stejně jako v obrázku B.15. Pokud jsou hodnoty vyhovující, stačí konfiguraci pojmenovat. Nyní je možné přejít k definici business fáze. Při popisu business fáze se nastavuje právě tato konfigurace společně s klasifikační a ohodnocovací jednotkou. Na obrázku B.14 lze vidět, že byla pro tento experiment zvolena zmíněná základní konfigurace, základní klasifikační jednotka a ohodnocování se řídí nalezenými okolními zařízeními, a to způsobem, který byl popsán dříve v této práci. Business fázi byla přiřazena obrazovka My profile, která obsahuje formulář, který slouží k vyplnění dat o uživateli. Po vytvoření fáze je uživatel přesměrován na konfiguraci business vlastností polí. Ve formuláři jsou zobrazena všechna pole, která komponenty na obrazovkách, jež jsou přiřazené k fázi, obsahují. V tomto případě jsou to pouze pole zmiňovaného formuláře pro vyplnění osobních dat. Vlastnosti, které byly jednotlivým polím nastaveny v rámci tohoto experimentu, jsou zobrazeny v příloze B.16.

5.3.2.2 Průběh experimentu

Objevují se zde celkem čtyři kombinace účelu a důležitosti polí. Zvolená ohodnocovací jednotka je nastavena tak, že pro tato pole vypočítá základní skóre, které se poté modifikuje na základě podobnosti blízkých okolních zařízení. Tyto kombinace a základní skóre jsou uvedeny v tabulce 5.1.

Tabulka 5.1: Kombinace účelu a důležitosti informace pro jednotlivá pole komponenty z experimentu

Účel	**Důležitost**	**Základní skóre**	**Pole**
CRITICAL	SYSTEM_IDENTIFICATION	100	login, password, active
REQUIRED	SYSTEM_INFORMATION	88	firstName, lastName, street, city, postCode, country, email
REQUIRED	FUTURE_INTERACTION	82	confidentialAgreement
NICE_TO_HAVE	INFORMATION_MINING	46	gender, hireDate, age

Je nutné poznamenat, že se základní skóre počítá pouze v případě, že je požadavek na komponentu zaslán z mobilního zařízení. Pro Java SE aplikaci nebyl implementován způsob získávání těchto informací a nelze je využít, proto se v případě těchto aplikací nevrací žádné skóre a UI není nijak tímto procesem ovlivněno. Ohodnocovací jednotka použitá v experimentu byla navržena tak, aby fungovala pouze u polí, které mají účel nastaven na `SYSTEM_INFORMATION` nebo `INFORMATION_MINING`, týká se to tedy pouze polí ve druhém a posledním řádku tabulky 5.1. Skóre polí, které mají kombinaci stejnou jako je uvedena v prvním řádku, tedy `SYSTEM_IDENTIFICATION` a `CRITICAL`, už dále není upravováno, jelikož jsou tato pole nezbytná k dokončení úkolu v systému. Cílem této ohodnocovací jednotky je nedůležitá pole schovat a na důležitějších polích zmírnit přísnost validačních pravidel. Na základě podobnosti okolních zařízení je od základního skóre odečtena experimentálně určená hodnota.

Tento experiment byl proveden celkem třikrát. Jednou byl proces spuštěn v knihovně ze stejného místa bezprostředně po původním měření kontextu. Podruhé šlo o měření, ze stejného místa v knihovně po zhruba hodinovém časovém odstupu. Třetí běh experimentu se odehrával na jiném místě, než bylo provedeno původní měření, konkrétně před budovou knihovny. V prvním případě byla okolní zařízení naprosto stejná, jednalo se tedy o 100% shodu, od skóre byla odečtena hodnota 35. V druhém případě byla zařízení shodná přibližně ze 46,5%, což je už spíše rozdílná množina zařízení, byla tedy odečtena pouze hodnota 5. Ve třetím případě činila shoda přibližně 6%, což je téměř nulová shoda, k hodnotě je tedy pro změnu připočteno 15. Pokud skóre přesáhne hodnotu 100, je na ni zaokrouhleno. V tabulce 5.2 jsou uvedeny výsledky pro daná pole a jejich příslušná chování v upravené komponentě, která byla určena pomocí konfigurace, jež byla business fázi přiřazena a jejíž hodnoty jdou vidět v obrázku B.15. Proces klasifikace se na straně proxy aplikace logoval. V příloze C.4 jsou vypsány logy, které vznikly během experimentů.

V prvním experimentu bylo přiřazeno první skupině polí chování ONLY_DIPSLAY, pole by tedy měla být zobrazena, ale neměla by být vyžadována ani validována, druhá skupina polí má přiřazeno chování HIDDEN, neměla by tedy být zobrazena. Seznam polí a jejich validace, které by měl tento formulář mít, lze porovnat s Java SE aplikací, která formulář žádným způsobem neupravuje. Snímek obrazovky zachycující stav formuláře při neúspěšném pokusu o odeslání v Java SE aplikaci lze nalézt v příloze B.17. Na obrázku B.18 je možno vidět, že pole age, gender a hireDate opravdu ve formuláři chybí. Také je nyní možné pole,

Tabulka 5.2: Výsledky ohodnocení skupin polí na základě podobnosti okolních zařízení

Skupina polí	**Pokus 1**	**Pokus 2**	**Pokus 3**
firstName, lastName, street, city, postCode, country, email	53 ONLY_DISPLAY	83 VALIDATION	100 REQUIRED
gender, hireDate, age	11 HIDDEN	41 ONLY_DISPLAY	61 VALIDATION

která byla předtím povinná, nevyplnit. Pole active, login a password mají nejvyšší možné ohodnocení a jsou tedy povinná.

Ve druhém případě by první skupina polí neměla být vyžadována, nicméně pokud jsou pro pole definovaná jiná validační pravidla, měla by se vyplněná hodnota podle nich zkontrolovat. Takovými pravidly může být omezení počtu znaků nebo minimální či maximální číselná hodnota. Druhá skupina polí by se měla jen zobrazit a věškerá validační pravidla by měla být z definice odstraněna. Seznam polí je kompletní, žádné není schováno. Na snímku obrazovky z Java SE aplikace lze vidět, že příjmení může mít maximálně 255 znaků a věk uživatele může být maximálně 60. Zatímco validace příjmení přetrvává, omezení na věku uživatele je odstraněno, čehož si lze všimnout na obrázku B.19. Také je zde možné vidět, že pole jako jméno nebo ulice již nejsou povinná.

V posledním případě je kontext rozdílný, což znamená, že nelze nic předpokládat. Z toho důvodu budou informace z první skupiny polí vyžadovány a z druhé skupiny validovány. V příloze B.20 si lze povšimnout, že jsou pole první skupiny, tj. jméno, příjmení, ulice atd., povinná. Důkazem, že se pole z druhé skupiny opravdu validují, je pole s věkem uživatele, u kterého se ověřuje maximální možná hodnota. Seznam polí je opět kompletní jako při druhém běhu experimentu.

Druhá ohodnocovací jednotka, která k vyhodnocení využívá informace o baterii a typu připojení, funguje obdobně. Ohodnocovací mechanismus byl původně mírnější a polím bylo v některých situacích přiřazeno i chování `NOT_PRESENT`. To však způsobilo problémy, neboť backendová aplikace AFServer není připravena na to, že ji některá data z formuláře vůbec nepřijdou. Pokud je validace pouze na úrovni logiky backendu, mohl by být teoreticky vytvořen mechanismus, který by dočasně validace zakazoval, nebo povoloval v závislosti na zvoleném chování pole. Kdyby byly tyto validace i na úrovni databáze, byl by problém ještě komplikovanější, neboť databázová omezení nelze měnit za běhu aplikace. Na druhou stranu je velmi nepravděpodobné, že pole, které je potřebné natolik, že je na backendu validováno, bude mít na proxy nastavené business vlastnosti, které povedou k jeho vymazání z formuláře. Nicméně ohodnocovací strategie byla pro tento experiment upravena tak, aby byla pole pouze neviditelná, ale přítomná, čímž se eliminovaly chyby, jež nebyly pro experiment důležité.

Kapitola 6

Nasazení

Pro účely otestování řešení na reálných Android zařízeních byly serverové aplikace tj. proxy, NSRest a AFServer nasazeny na Forpsi hosting. Aplikace jsou dostupné na těchto URL adresách:

- AFServer - <`http://81.2.216.197:8080/AFServer`>
- Proxy - <`http://81.2.216.197:8080/UIxy/apps/list`>
- NSRest - <`http://81.2.216.197:8080/NSRest`>

V této kapitole je popsán proces nasazení těchto aplikací a problémy, které musely být při procesu řešeny.

6.1 Instalace aplikačního serveru Glassfish

Na hostingu běží CentOS, což je open-source ekosystém na bázi Linuxu. Do tohoto ekosystému bylo nejdříve nutné nainstalovat aplikační server Glassfish, na který mají být aplikace nasazeny. Před instalací aplikačního serveru bylo třeba nainstalovat nejdříve Javu. Jelikož byl použit Glassfish verze 4.1.2, bylo potřeba nainstalovat JDK verze 8u20 nebo výše. Po instalaci Glassfishe byla zaregistrována systémová služba, díky které se server nastartuje automaticky při nabootování CentOS. Také bylo potřeba upravit pravidla Firewallu, která zpřístupní základní porty, které Glassfish používá. Na portu 8080 poslouchá Glassfish HTTP požadavky, port 8181 slouží pro HTTPS požadavky a 4848 je port, přes který se lze dostat do administrační konzole serveru. Tyto tři porty přísluší jedné tzv. doméně, na které může běžet libovolné množství aplikací. Domén lze mít i více, přičemž každá může mít svou vlastní konfiguraci a administraci [13].

6.2 Nasazení AFServer

Nasazení aplikace AFServer bylo ze všech tří aplikací nejjednodušší. AFServer totiž využívá in-memory databázi DerbyDB, která je dodávána s Glassfishem a má předkonfigurované

připojení na zdroj dat. V případě této aplikace tedy není třeba žádného dodatečného instalování či nastavování. Pomocí příkazu `mvn clean install` byl sestaven WAR soubor, který stačí vložit do adresáře autodeploy, jež se nachází v adresáři

<glassfish_root>/glassfish/domains/<nazev_domeny>/

Po vložení souboru proběhne nasazení během několika vteřin.

6.3 Nasazení NSRest

NSRest využívá oproti AFServer navíc MongoDB databázi. Tu je opět potřeba nejdříve nainstalovat. Obdobně jako u Glassfishe je vhodné zaregistrovat systémovou službu, která databázi nastartuje při bootování systému. Pro komunikaci s databází z Java kódu je využit MongoDB Java Driver. Připojení na databázi lze provést pouze z kódu aplikace, na aplikačním serveru není nutné cokoliv konfigurovat.

Databázi je vhodné zabezpečit. Pro tyto účely je třeba vytvořit nejdříve uživatele způsobem, který je uvedený v 6.1, a poté databázi restartovat s přepínačem `-auth`.

Listing 6.1: Vytvoření uživatele v MongoDB

```
use NSDatabase
db.createUser(
  {
    user: "matyapav",
    pwd: "abc123",
    roles: [ { role: "readWrite", db: "NSDatabase" } ]
  }
)
```

Ve výše zmíněném úryvku je nejdříve zvolena databáze NSDatabase, ve které má být uživatel vytvořen. Příkazem db.createUser je vytvořen uživatel se jménem matyapav a heslem abc123 a s právy pro čtení a zápis do databáze NSDatabase. Z aplikace se opět sestavil WAR soubor a přesunutím do adresáře autodeploy byl projekt nasazen.

6.4 Nasazení Proxy aplikace

Proxy aplikace byla na nasazení nejsložitější, protože bylo třeba nainstalovat PostgreSQL databázi, přidat do Glassfishe příslušný JDBC driver a přes administrační konzoli jej nakonfigurovat. Poté bylo třeba naplnit databázi daty, která reprezentují obrazovky a komponenty ukázkových aplikací pro správu absencí, které byly popsány v rámci kapitoly o testování. Po instalaci PostgreSQL byla opět zaregistrována systémová služba pro nastartování při bootování systému. Po nastartování databáze bylo třeba vytvořit uživatele a databázi pomocí příkazů 6.2.

Listing 6.2: Vytvoření uživatele a databáze v PostgreSQL

```
CREATE USER uixy_user WITH PASSWORD 'DoIt4Tweety';
CREATE DATABASE uixy_db WITH ENCODING='UTF8' OWNER=uixy_user CONNECTION LIMIT=-1;
```

Dále bylo potřeba nakonfigurovat připojení na databázi. Nejdříve bylo třeba stáhnout JDBC driver a umístit ho do složky

<glassfish_root>/glassfish/domains/<nazev_domeny>/lib

a aplikační server restartovat. Poté bylo třeba driver nakonfigurovat přes admin konzoli. Aby bylo možné do admin konzole přistoupit, musí existovat alespoň jeden uživatel aplikačního serveru s heslem. To lze nastavit pomocí příkazu `asadmin change-admin-password`. Po změně hesla je možné spustit příkaz `asadmin enable-secure-admin`, který na portu 4848 zpřístupní administrační konzoli. V administraci je potřeba vytvořit tzv. JDBC connection pool, kterému jsou nastaveny vlastnosti, jako jméno uživatele, heslo a port. Po jeho vytvoření je možné vytvořit a pojmenovat data source. V aplikaci je poté v persistence.xml třeba pod stejným jménem nakonfigurovat persistentní jednotku.

Při nasazování proxy aplikace došlo k chybě, která hlásila, že neexistuje metoda `debugf()` třídy `Logger` z modulu `org.jboss.logging`, která vzniká, pokud aplikace využívá Hibernate verze 5.x v kombinaci s Glassfish verze 4.x. Tento problém nastává, protože Glassfish obsahuje modul `org.jboss.logging`, který se snaží zahrnout i Hibernate a vzniká konflikt. Hibernate se snaží využít funkci `debugf()`, která je dostupná pouze ve verzi jboss-logging modulu 3.3.0 a výše, ale Glassfish obsahuje starší verzi [32]. Tento problém musel být vyřešen nahrazením tohoto modulu za novější verzi. Po nahrazení musel být Glassfish opět restartován, aby se změny projevily.

Nakonec se pomocí Mavenu sestavil WAR soubor a přesunutím do autodeploy byla aplikace nasazena. Při nasazení aplikace se vytvořila automaticky struktura databáze pomocí Hibernatu. Nyní bylo do tabulek možné vložit data o obrazovkách a komponentách aplikací pro správu absencí. K tomu je připravený v projektu v adresáři configuration SQL soubor data.sql, který lze nahrát pomocí příkazu `psql uixy_db < data.sql`.

Kapitola 7

Závěr

7.1 Budoucí vývoj

Automatický sběr kontextových dat, zejména dat o blízkých zařízeních, je bohužel v současné době značně omezený. Co se týče okolních zařízení, je tomu tak hlavně z důvodu, že výrobci ke svým zařízením zřídkakdy poskytují veřejně dostupné rozhraní. Pokud se tato situace časem změní a bude možné automaticky bez zásahu uživatele z těchto zařízení získávat více informací, je rozhodně plánováno sběr těchto informací doimplementovat. S novými daty vznikají nové možnosti, jak data využít. Zajímavá jsou především data z tělesných senzorů, která mohou vypovídat o psychickém a emočním stavu uživatele. Sběr těchto dat byl v rámci této práce vyzkoušen s použitím BLE, nicméně se zařízeními se skrz tuto technologii nepodařilo spojit. Tento problém je nutné v rámci budoucího vývoje vyřešit.

Je také žádoucí rozšířit proces generování uživatelských rozhraní o další komponenty. Nyní je možné za základě kontextu upravovat pouze vlastnosti formuláře, tabulky či listu. Potenciál využití kontextových dat je ale mnohem větší, například by se mohly generovat tímto stylem i různé dialogy či ovládací prvky aplikace.

Do budoucna je také žádoucí vylepšit rozhodování o tom, v jakém kontextu se uživatel nachází a co to pro uživatelské rozhraní znamená. Nyní se bere v úvahu jen jedno měření dat nebo se porovnává aktuální kontext s předešlým. Kdyby v této roli figuroval složitejší systém, který by se například dokázal i učit, mohly by být výsledky personalizace uživatelského rozhraní ještě přívětivější.

Dalším aspektem, který je v budoucnu potřeba zvážit, je optimalizace řešení z hlediska využití baterie. Aplikace, jež využívají tento přístup, jsou Android zařízeními detekovány jako náročné na baterii. Prostor pro zlepšení je například v použití Bluetooth Low energy místo klasického Bluetooth, problémem však je, že BLE ještě nepodporují některá ze starších zařízení.

Aktuální řešení bylo implementováno pro Java SE aplikace a mobilní platformu Android. Migrovat řešení na Windows phone již nyní kvůli úpadku tohoto systému a velmi malému počtu zařízení na trhu nemá význam. Nicméně bylo by vhodné zpřístupnit celé aktuální řešení i pro platformu iOS.

7.2 Zhodnocení práce

Cílem této práce bylo navrhnout a implementovat způsob sběru, uložení a využití kontextových dat v procesu generování uživatelského rozhraní. K úpravě komponent uživatelského rozhraní měl být implementován klasifikační systém, který komponentu na základě kontextu a business pravidel ohodnotí a upraví její vlastnosti. Pro správu částí uživatelského rozhraní klientských aplikací a jejich business pravidel mělo být vytvořeno uživatelské rozhraní. To slouží i ke konfiguraci procesu klasifikace komponent. Na základě nového způsobu generování uživatelského rozhraní, u kterého už se negenerují jen komponenty, nýbrž i menu a obrazovky, bylo nutné modifikovat klientské frameworky, které interpretují části UI. S tím souvisela i úprava klientských aplikací.

Byly tedy naimplementovány další dvě serverové aplikace a framework pro sběr kontextových dat. Android framework pro sběr dat dokáže získávat data ze senzorů zařízení a také zjistit seznam okolních zařízení pomocí Bluetooth a sítě. Uživateli frameworku je nabídnuta možnost určit, které z dat chce sbírat. Získávání dat může být velmi náročné na baterii zařízení, proto je možné omezit některé sběrače minimální kapacitou baterie, kterou k běhu potřebují. Framework také nabízí možnost spouštět proces sběru informací periodicky. Se získanými daty může naložit uživatel po svém nebo je může odeslat do aplikace NSRest, kde se ukládají v NoSQL databázi MongoDB. NSRest poskytuje pro komunikaci RESTové rozhraní, do kterého lze kontextová data buď poslat nebo je z něj získat. Proxy aplikace slouží jako mezičlánek mezi klientskou a backendovou aplikací AFServer. Uživatel může v UI proxy aplikace vytvořit nové obrazovky a jejich komponenty, ze kterých se má skládat UI klientských aplikací. Také lze definovat business případy, které v klientské aplikaci mohou nastat. Ty se mohou dělit na fáze, každá fáze může obsahovat několik obrazovek. Poté, pokud je z klientské aplikace na obrazovku přistoupeno, je určen business případ a fáze, jejíž definované vlastnosti se využijí v procesu úpravy komponenty na základě kontextových dat. Mezi vlastnosti fáze patří účel a míra potřeby informace, které lze specifikovat u každého pole komponent, které se nachází v obrazovkých přidružených k fázi.

U business fáze se také definuje, jaká ohodnocovací a klasifikační jednotka se při úpravě komponenty použije. Ohodnocovací jednotka vypočítá pro každou část komponenty skóre, které předá klasifikační jednotce. Výpočet skóre se nyní děje na základě dvou algoritmů, jeden uvažuje baterii a typ připojení, druhý porovnává aktuální a předešlou množinu okolních zařízení. Kontextová data proxy aplikace získá z NSRest pomocí HTTP požadavku, ve kterém je nutné specifikovat mac adresu zařízení, prováděnou akci a čas. Podle těchto tří vlastností se identifikuje příslušný záznam v databázi. Tyto informace jsou poté na proxy parsovány a poskytnuty ohodnocovacím jednotkách, které je mohou při výpočtu ohodnocení využít. Klasifikační jednotka na základě tohoto ohodnocení vybere způsob (neboli chování), jakým se pole v komponentě upraví. Za základě vybraného chování je každá definice pole upravena. Mezi tyto úpravy patří změna validačních paravidel pole, změna viditelnosti pole nebo může být pole úplně z definice komponenty vyřazeno. Modifikovaná definice je zaslána klientovi, který z ní vytvoří prvky uživatelského rozhraní.

Výsledná komponenta může mít mírnější validace jednotlivých polí nebo může některé méně důležité části neobsahovat. To může uživateli usnadnit splnění úkolu, neboť není zatěžován zbytečnými informacemi. Další výhodou je, že pokud upravená komponenta obsahuje méně informací, využije se na její stažení méně mobilních dat.

Nevýhodou aplikací, které tento přístup využívají, je zvýšená náročnost na baterii. Pro efektivnější využití je doporučené mít kromě internetu zapnutý Bluetooth a GPS, což baterii opět moc nepřidává. Další nevýhodou je fakt, že tyto aplikace kvůli sběru informací vyžadují velké množství přístupových práv, což může uživatele od instalace aplikace odradit.

Literatura

[1] *Android 6.0 changes - Access to Hardware Identifier* [online]. [cit. 26. 04. 2018]. Dostupné z: <https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-hardware-id>.

[2] *Android Developers - Bluetooth* [online]. [cit. 21. 04. 2018]. Dostupné z: <https://developer.android.com/guide/topics/connectivity/bluetooth.html>.

[3] *Android developers* [online]. [cit. 13. 04. 2018]. Dostupné z: <https://developer.android.com>.

[4] *Request App Permissions* [online]. [cit. 13. 04. 2018]. Dostupné z: <https://developer.android.com/training/permissions/requesting.html>.

[5] *Android SDK tutorial for beginners* [online]. [cit. 25. 04. 2018]. Dostupné z: <https://www.androidauthority.com/android-sdk-tutorial-beginners-634376/>.

[6] *Distribution of Android operating systems used by Android phone owners in February 2018, by platform versions* [online]. [cit. 25. 04. 2018]. Dostupné z: <https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/>.

[7] *AspectFaces framework* [online]. [cit. 13. 04. 2018]. Dostupné z: <http://www.aspectfaces.com/overview>.

[8] *What is AWARE?* [online]. [cit. 16. 04. 2018]. Dostupné z: <http://www.awareframework.com/what-is-aware/>.

[9] *Easy Admin* [online]. [cit. 16. 04. 2018]. Dostupné z: <https://github.com/EasyCorp/EasyAdminBundle>.

[10] *FindBugs^{TM} - Find Bugs in Java Programs* [online]. [cit. 02. 05. 2018]. Dostupné z: <http://findbugs.sourceforge.net/>.

[11] *Fitbit - Web API reference* [online]. [cit. 16. 04. 2018]. Dostupné z: <https://dev.fitbit.com/build/reference/web-api/>.

[12] *GATT Specifications* [online]. [cit. 14. 04. 2018]. Dostupné z: <https://www.bluetooth.com/specifications/gatt>.

[13] *GlassFish Server Open Source Edition* [online]. [cit. 25. 04. 2018]. Dostupné z: <https://glassfish.java.net/docs/4.0/quick-start-guide.pdf>.

[14] *The Google Fit SDK* [online]. [cit. 13. 04. 2018]. Dostupné z: <https://developers.google.com/fit/>.

[15] *Google Fit SDK - Support Additional Sensors* [online]. [cit. 14. 04. 2018]. Dostupné z: <https://developers.google.com/fit/android/new-sensors>.

[16] *Fitness Data Types - Restricted Data Types* [online]. [cit. 14. 04. 2018]. Dostupné z: <https://developers.google.com/fit/android/data-types>.

[17] *Human-Computer Interaction (HCI)* [online]. [cit. 15. 05. 2018]. Dostupné z: <https://www.interaction-design.org/literature/topics/human-computer-interaction>.

[18] *Is JavaFX replacing Swing as the new client UI library for Java SE?* [online]. [cit. 25. 04. 2018]. Dostupné z: <http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6>.

[19] *JSON and BSON* [online]. [cit. 21. 04. 2018]. Dostupné z: <https://www.mongodb.com/json-and-bson>.

[20] *Mockito - Tasty mocking framework for unit tests in Java* [online]. [cit. 02. 05. 2018]. Dostupné z: <http://site.mockito.org/>.

[21] *MongoDB Java Driver* [online]. [cit. 25. 04. 2018]. Dostupné z: <https://mongodb.github.io/mongo-java-driver/>.

[22] Top 5 Considerations When Evaluating NoSQL Databases. Feb 2018. Dostupné z: <https://www.mongodb.com/collateral/top-5-considerations-when-evaluating-nosql-databases?jmp=hero>.

[23] *ObjectId* [online]. [cit. 28. 04. 2018]. Dostupné z: <https://docs.mongodb.com/manual/reference/method/ObjectId/>.

[24] *PostgreSQL JDBC Driver - About* [online]. [cit. 25. 04. 2018]. Dostupné z: <https://jdbc.postgresql.org/about/about.html>.

[25] *Robolectric - test-drive your Android code* [online]. [cit. 02. 05. 2018]. Dostupné z: <http://robolectric.org/>.

[26] *Sensors Overview* [online]. [cit. 13. 04. 2018]. Dostupné z: <https://developer.android.com/guide/topics/sensors/sensors_overview.html>.

[27] *Mobile Operating System Market Share Worldwide* [online]. [cit. 15. 04. 2018]. Dostupné z: <http://gs.statcounter.com/os-market-share/mobile/worldwide>.

[28] *What is a Subnet Mask?* [online]. [cit. 26. 04. 2018]. Dostupné z: <https://www.iplocation.net/subnet-mask>.

[29] *Tizen developers* [online]. [cit. 14. 04. 2018]. Dostupné z: <https://developer.tizen.org>.

[30] *Automatic Form Generation* [online]. [cit. 16. 04. 2018]. Dostupné z: <https://vaadin.com/docs/v7/framework/jpacontainer/jpacontainer-fieldfactory.html>.

[31] ABOWD, G. D. et al. Towards a Better Understanding of Context and Context-Awareness. In GELLERSEN, H.-W. (Ed.) *Handheld and Ubiquitous Computing*, s. 304–307, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48157-7.

[32] AUAD, T. *NOSUCHMETHODERROR: ORG.JBOSS.LOGGING.LOGGER.DEBUGF(LJAVA/LAN* [online]. [cit. 05. 05. 2018]. Dostupné z: <https://tassioauad.com/2018/01/05/nosuchmethoderror-org-jboss-logging-logger-debugfljava-lang-stringi/>.

[33] BLOGS, G. *Identifying App Installations* [online]. [cit. 23. 04. 2018]. Dostupné z: <https://android-developers.googleblog.com/2011/03/identifying-app-installations.html>.

[34] BUTTER, T. et al. Context-aware User Interface Framework for Mobile Applications. In *Distributed Computing Systems Workshops, 2007. ICDCSW '07. 27th International Conference on*. IEEE, July 2007. doi: 10.1109/ICDCSW.2007.31. Dostupné z: <https://doi.org/10.1109/ICDCSW.2007.31>.

[35] FOWLER, M. *Destilované UML*. Knihovna programátora. : Grada, 2009. Dostupné z: <https://books.google.cz/books?id=rv1JK3b63igC>. ISBN 9788024720623.

[36] GUPTA, A. *Java EE: The Basics* [online]. [cit. 25. 04. 2018]. Dostupné z: <https://dzone.com/articles/java-ee-basics>.

[37] HOFFMAN, C. *How (and Why) to Change Your MAC Address on Windows, Linux, and Mac* [online]. [cit. 23. 04. 2018]. Dostupné z: <https://www.howtogeek.com/192173/how-and-why-to-change-your-mac-address-on-windows-linux-and-mac/>.

[38] ISSA, T. – ISAIAS, P. *Usability and Human Computer Interaction (HCI)*, s. 19–36. Springer London, London, 2015. doi: 10.1007/978-1-4471-6753-2_2. Dostupné z: <https://doi.org/10.1007/978-1-4471-6753-2_2>. ISBN 978-1-4471-6753-2.

[39] K, R. *What is: ARP (Address Resolution Protocol), ARP Cache Table, ARP Poisoning/ Broadcast Storm* [online]. [cit. 27. 04. 2018]. Dostupné z: <http://www.excitingip.com/2362/what-is-arp-address-resolution-protocol-arp-cache-table-arp-poisoning-broadcast-sto

[40] KILIAN, K. *Google po vzoru Applu schvaluje aplikace do Obchodu Play ručně* [online]. [cit. 15. 04. 2018]. Dostupné z: <http://www.svetandroida.cz/google-play-schvalovani-201503>.

[41] LANEY, D. 3D Data Management: Controlling Data Volume, Velocity, and Variety. Technical report, META Group, February 2001. Dostupné z: <http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf>.

[42] LASATER, C. *Design Patterns.* : Jones & Bartlett Learning, LLC, 2010. Dostupné z: <https://books.google.cz/books?id=vmZ5lvTyEYEC>. ISBN 9781449612887.

[43] MACIK, M. – CERNY, T. – SLAVIK, P. Context-sensitive, cross-platform user interface generation. *Journal on Multimodal User Interfaces.* Jun 2014, 8, 2, s. 217–229. ISSN 1783-8738. doi: 10.1007/s12193-013-0141-0. Dostupné z: <https://doi.org/10.1007/s12193-013-0141-0>.

[44] MAGESH, A. M. *My journey towards Reverse Engineering a Smart Band* [online]. [cit. 13. 04. 2018]. Dostupné z: <goo.gl/rgYgFW>.

[45] MANAR, M. – SHRESTHA, B. – SAXENA, N. SMASheD: Sniffing and Manipulating Android Sensor Data for Offensive Purposes. *IEEE Transactions on Information Forensics and Security.* Oct 2016, 12, 4, s. 901–913. doi: 10.1109/TIFS.2016.2620278. Dostupné z: <https://doi.org/10.1109/TIFS.2016.2620278>.

[46] MARTINEZ-RUIZ, F. J. – VANDERDONCKT, J. – ARTEAGA, J. M. Context-Aware Generation of User Interface Containers for Mobile Devices. In *2008 Mexican International Conference on Computer Science.* IEEE, Oct 2008. doi: 10.1109/ENC.2008.34. Dostupné z: <https://doi.org/10.1109/ENC.2008.34>.

[47] MATYAS, P. Servisně orientovaný aspektový vývoj uživatelských rozhraní pro mobilní aplikace, 5 2016.

[48] OLIYNYK, A. *App Store vs Google Play: Stores in Numbers* [online]. [cit. 13. 04. 2018]. Dostupné z: <https://masterofcode.com/blog/app-store-vs-google-play>.

[49] OSTRANDER, J. *Android UI Fundamentals: Develop & Design*, s. 66–72. Develop and Design. Pearson Education, 2012. Dostupné z: <https://books.google.cz/books?id=z5jAHfSeeJMC>. ISBN 9780132929028.

[50] PANKAJ, K. *Factory Design Pattern* [online]. [cit. 29. 04. 2018]. Dostupné z: <https://www.journaldev.com/1392/factory-design-pattern-in-java>.

[51] ČáPKA, D. *Úvod do UML* [online]. [cit. 20. 04. 2018]. Dostupné z: <https://www.itnetwork.cz/navrh/uml/uml-uvod-historie-vyznam-a-diagramy/>.

[52] RICHARDS, M. *Chapter 1. Layered Architecture* [online]. [cit. 28. 04. 2018]. Dostupné z: <https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html>.

[53] SHARKEY, J. *Coding for Life–Battery Life, That Is* [online]. [cit. 21. 04. 2018]. Dostupné z: <https://dl.google.com/io/2009/pres/W_0300_CodingforLife-BatteryLifeThatIs.pdf>.

[54] SVOBODA, M. *Database systems 2 - Introduction* [online]. [cit. 21. 04. 2018]. Dostupné z: <http://www.ksi.mff.cuni.cz/~svoboda/courses/171-B4M36DS2/lectures/Lecture-01-Introduction-FEL.pdf>.

[55] TOMASEK, M. Aspektově orientovaný vývoj uživatelských rozhraní pro Java SE aplikace. Master's thesis, České vysoké učení technické v Praze Fakulta Elektrotechnická, 1 2015.

[56] TOMASEK, M. – CERNY, T. Automated User Interface Generation Involving Field Classification. *Software Networking*. Jan 2017, 8, 2, s. 53–78. doi: 10.13052/jsn2445-9739.2017.004. Dostupné z: <http://www.riverpublishers.com/journal_read_html_article.php?j=JSN/2017/1/004>.

[57] TOMASEK, M. – CERNY, T. On Web Services UI in User Interface Generation in Standalone Applications. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, s. 363–368, New York, NY, USA, 2015. ACM. doi: 10.1145/2811411.2811537. Dostupné z: <http://doi.acm.org/10.1145/2811411.2811537>. ISBN 978-1-4503-3738-0.

[58] TRNKA, M. – TOMASEK, M. – CERNY, T. Context-Aware Security Using Internet of Things Devices. In KIM, K. – JOUKOV, N. (Ed.) *Information Science and Applications 2017*, s. 706–713, Singapore, 2017. Springer Singapore. ISBN 978-981-10-4154-9.

[59] WIKIPEDIANS, B. *Software Testing*. : PediaPress. Dostupné z: <https://books.google.cz/books?id=o2mFgGjktncC>.

[60] WONG, E. *User Interface Design Guidelines: 10 Rules of Thumb* [online]. [cit. 15. 04. 2018]. Dostupné z: <https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb>.

Příloha A

Seznam použitých zkratek

3G třetí generace mobilních telekomunikačních technologií

LTE Long Term Evolution

GPS Global positioning system

SDK Software Development Kit

API Application Programming Interface

UI User Interface

GUI Graphical user interface

BLE Bluetooth Low Energy

GATT Generic attribute

REST Representaitonal State Transfer

HCI Human computer interaction

UCD User centered design

SMS Short message service

XML Extensible Markup Language

JPA Java Persistance API

JSON Javascript Object Notation

LGPL GNU Lesser General Public License

C# C Sharp

ORM Object Relational Mapping

PHP Hypertext preprocesor (dříve Personal Home Page)

CRUD Create, Read, Update, Delete

XUL XML User Intergace Language

CSS Cascading Style Sheets

Java SE Java Standard Edition

IP Internet protocol

UML Unified modeling language

HTTP Hypertext Transfer Protocol

Java EE Java Enterprise Edition

3G Standard pro sítě 3. generace

4G Standard pro sítě 4. generace

SQL Structured Query Language

BSON Binary JSON

DB Database

JSP JavaServer Pages

AWT Abstract Window Toolkit

ADB Android Debug Bridge

POJO Plain Old Java Object

JDBC Java Database Connectivity

MIT Massachusetts Institute of Technology

USB Universal Serial Bus

BSSID Basic Service Set Identifier

SSID Service Set Identifier

ARP Address Resolution Protocol

URL Uniform Resource Locator

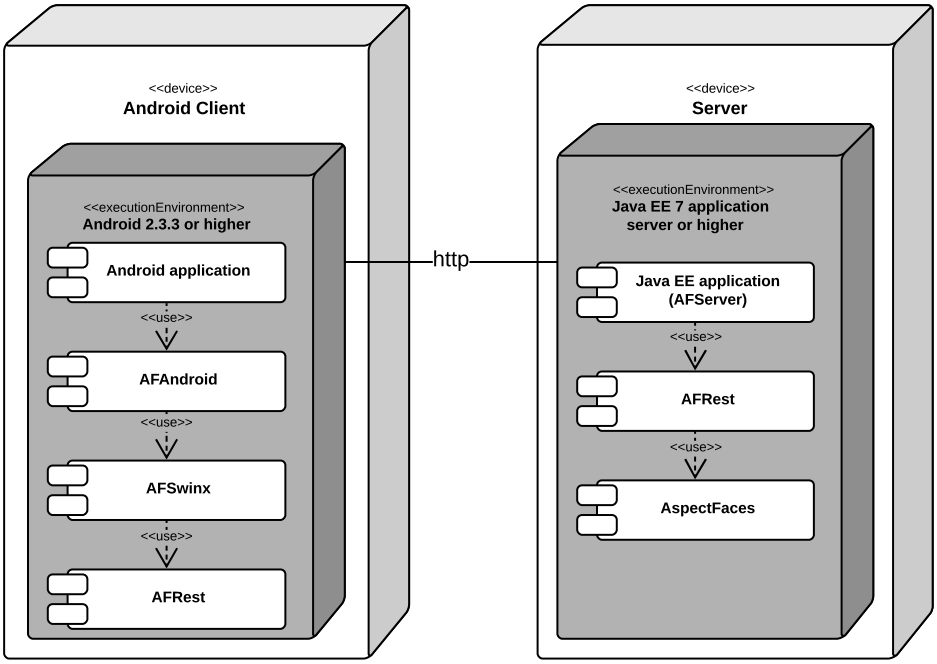
DAO Data Access Object

CentOS Community Enterprise Operating System

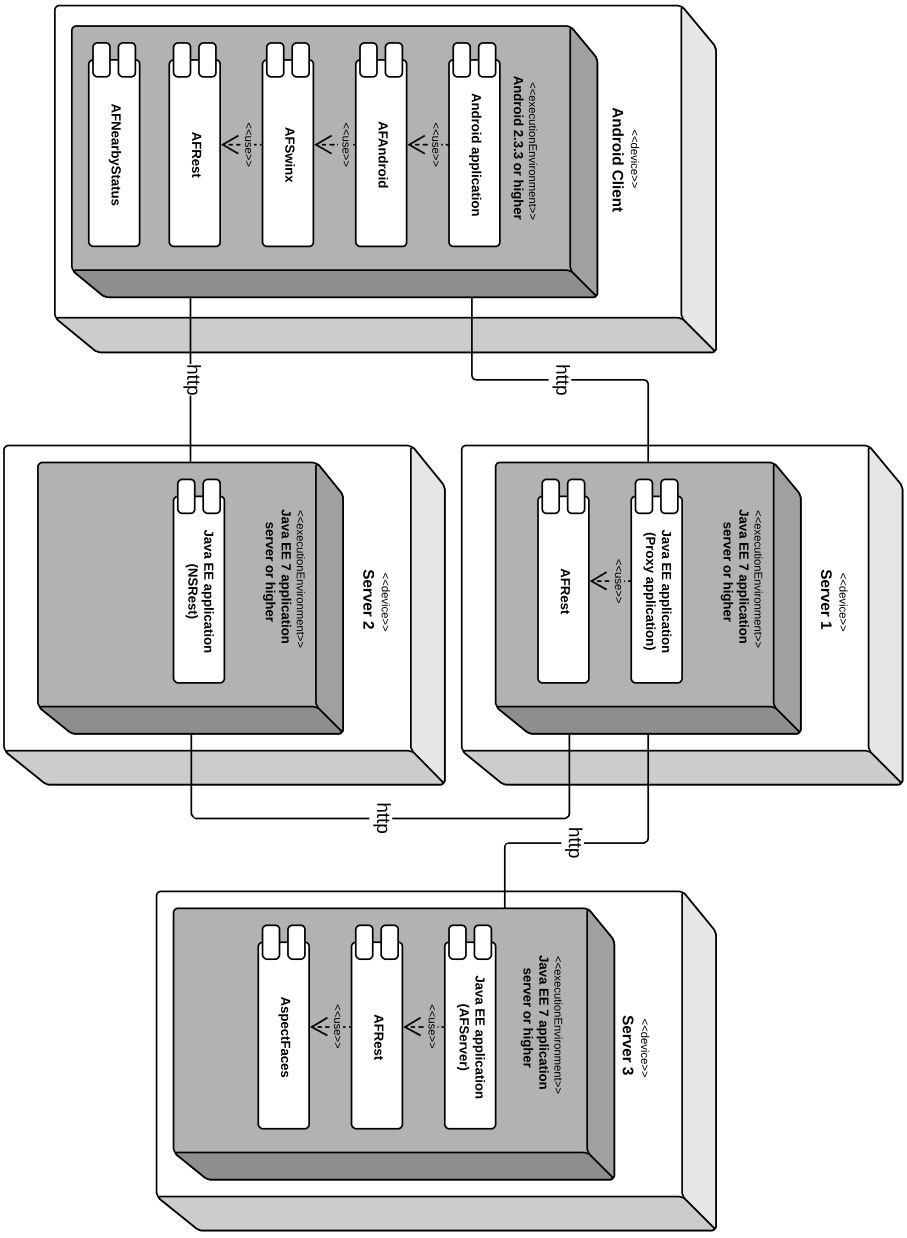
Příloha B

UML diagramy a obrázky

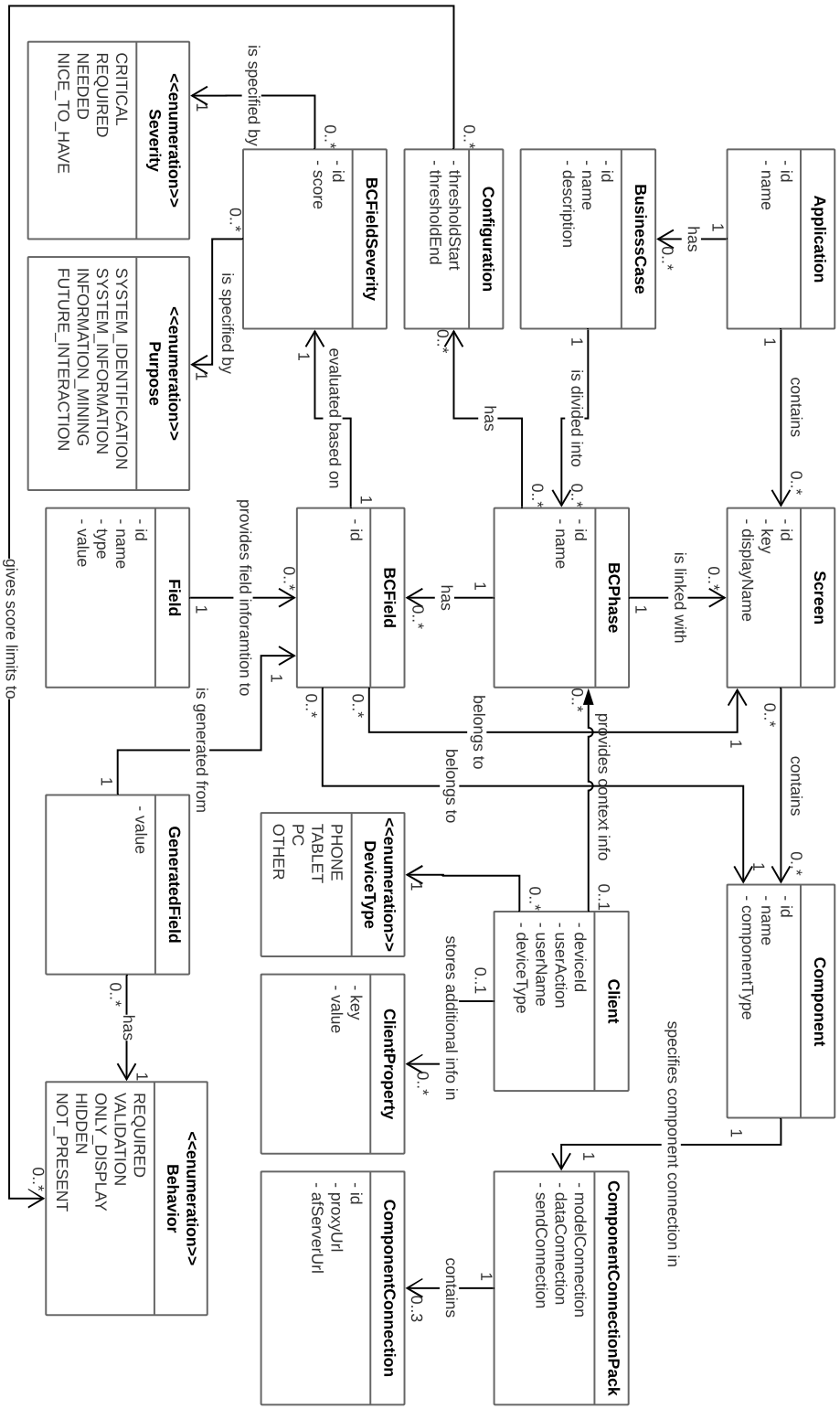
V této sekci naleznete použité UML diagramy a velké obrázky, na které bylo v textu odkazováno.



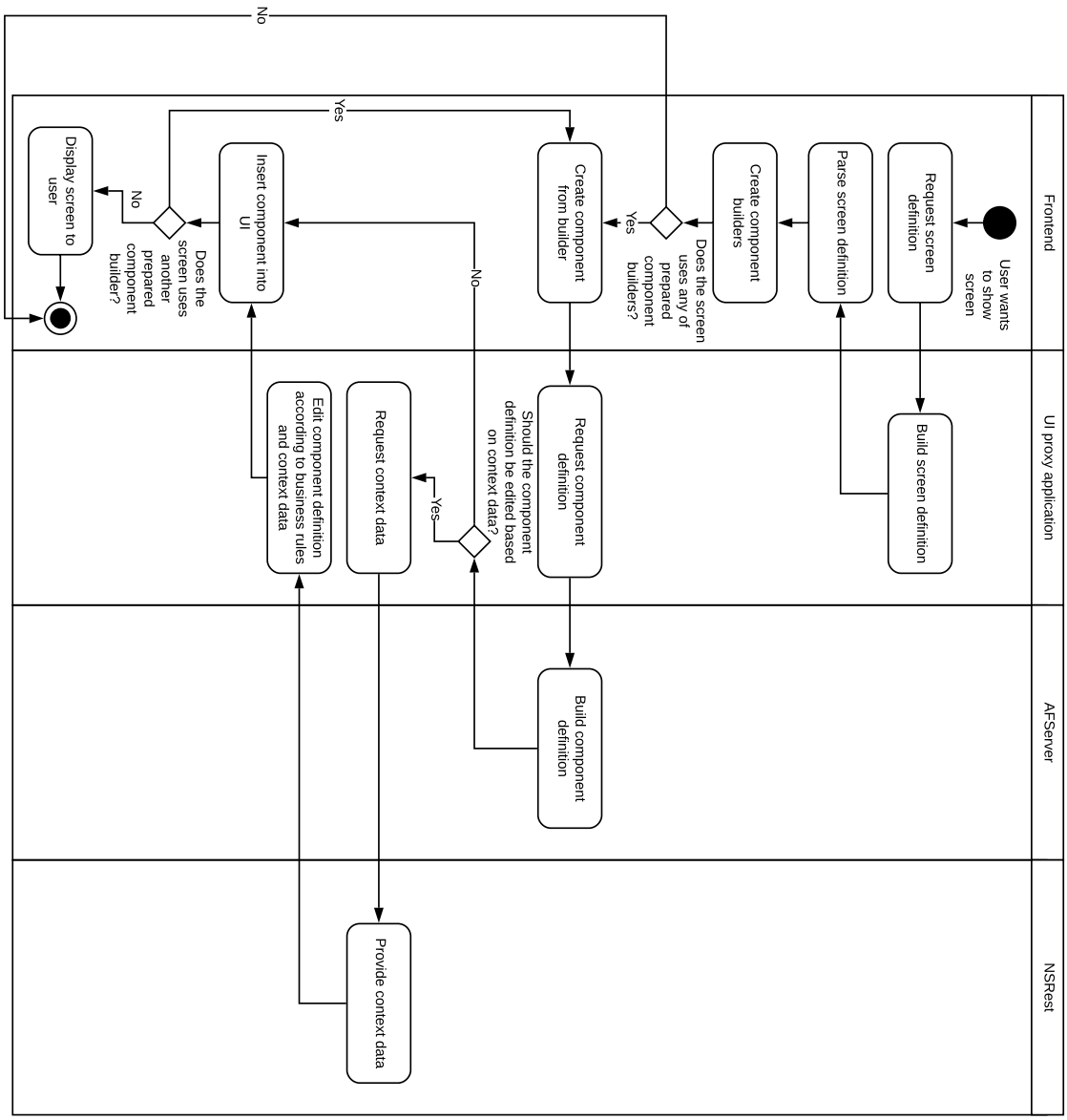
Obrázek B.1: Diagram nasazení původního řešení



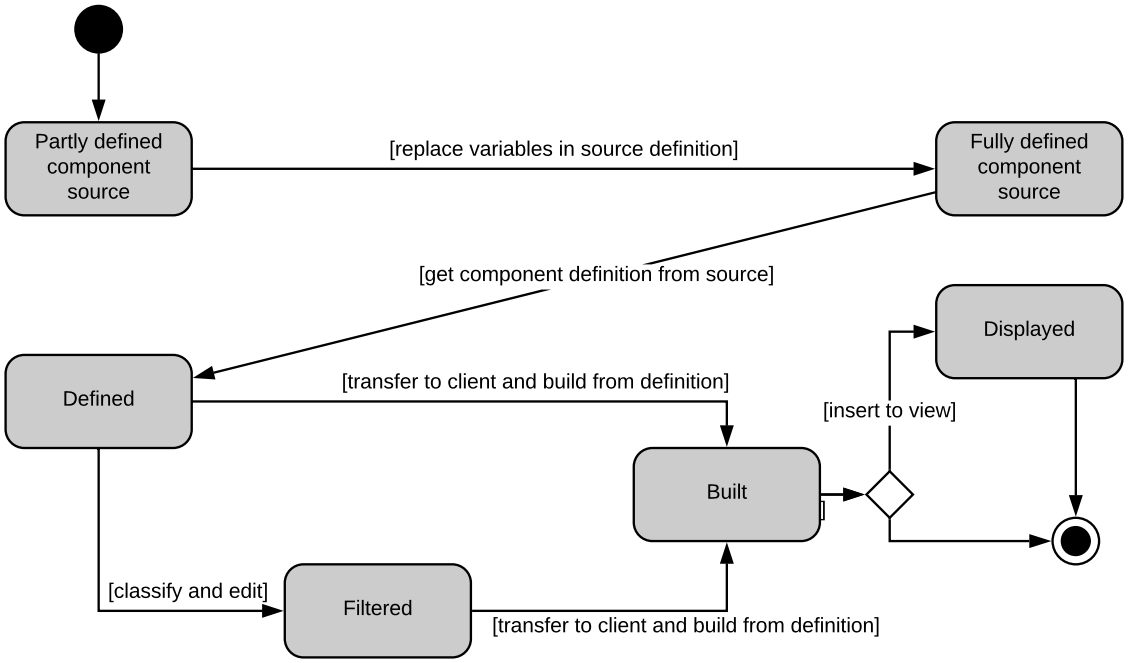
Obrázek B.2: Diagram nasazení rozšířeného řešení



Obrázek B.3: Doménový model proxy aplikace

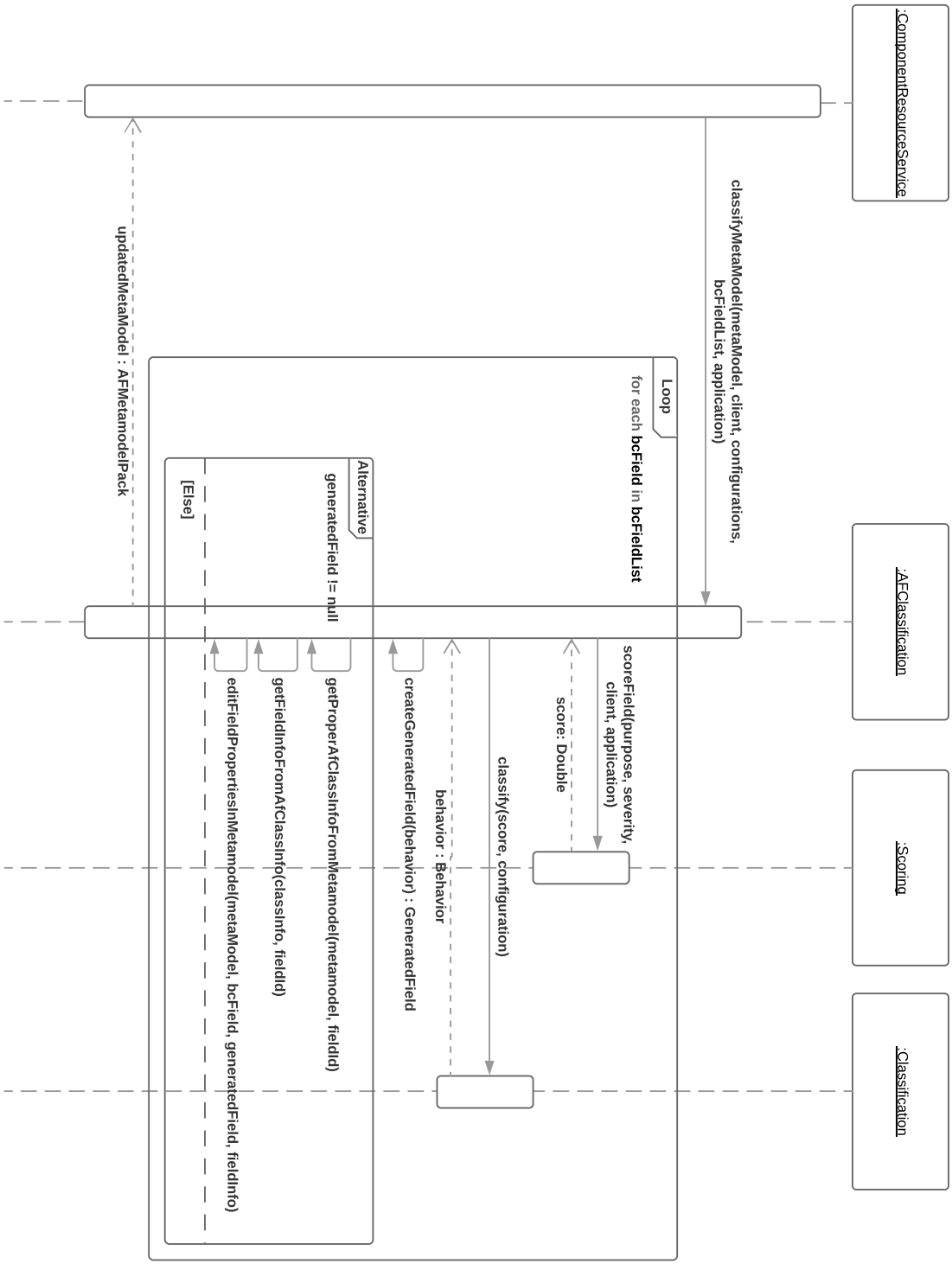


Obrázek B.4: Activity diagram zobrazující spolupráci aplikací při tvorbě obrazovky

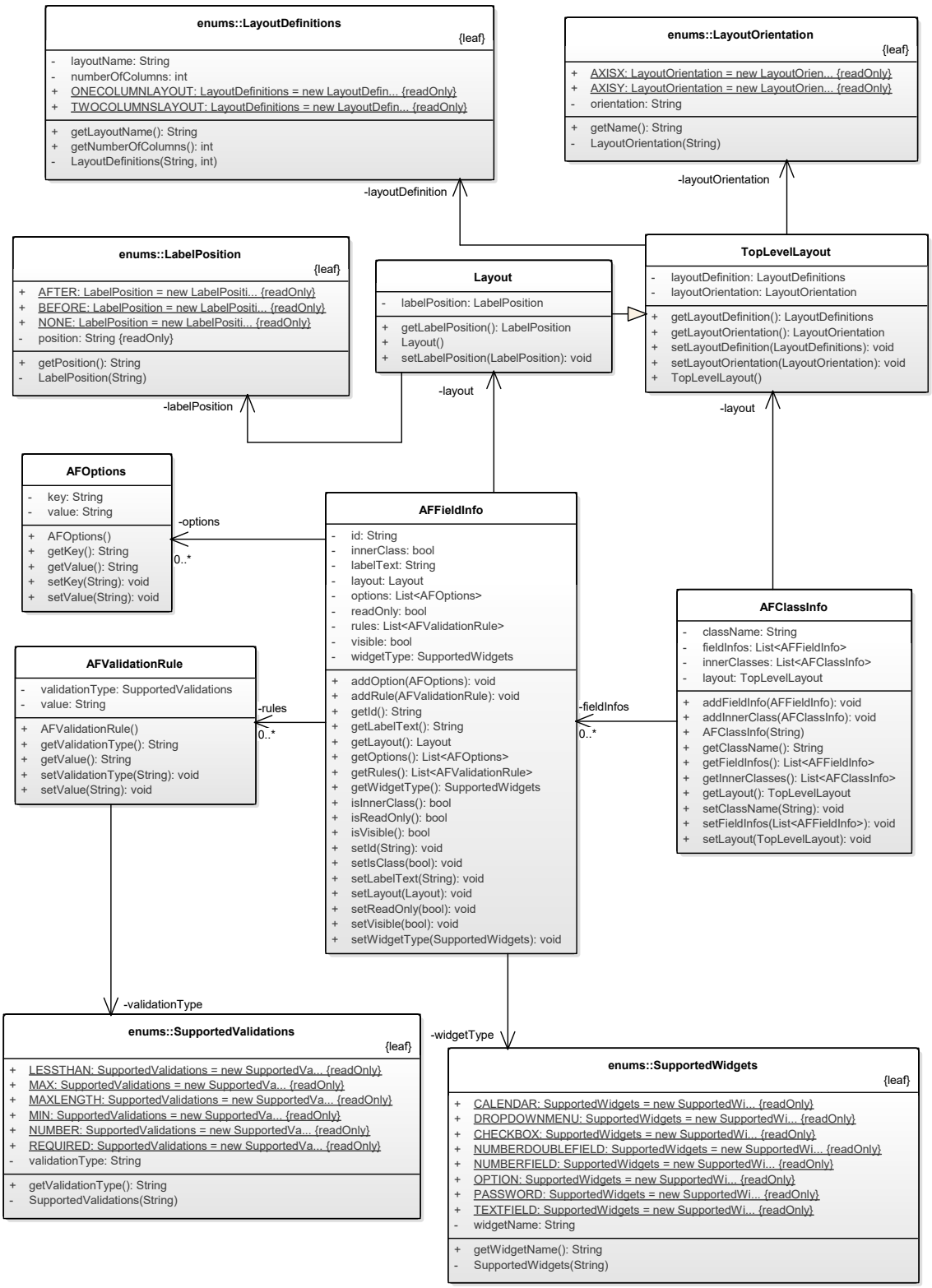


Obrázek B.5: Stavový diagram zachycující stavy komponenty při procesu jejího vytváření

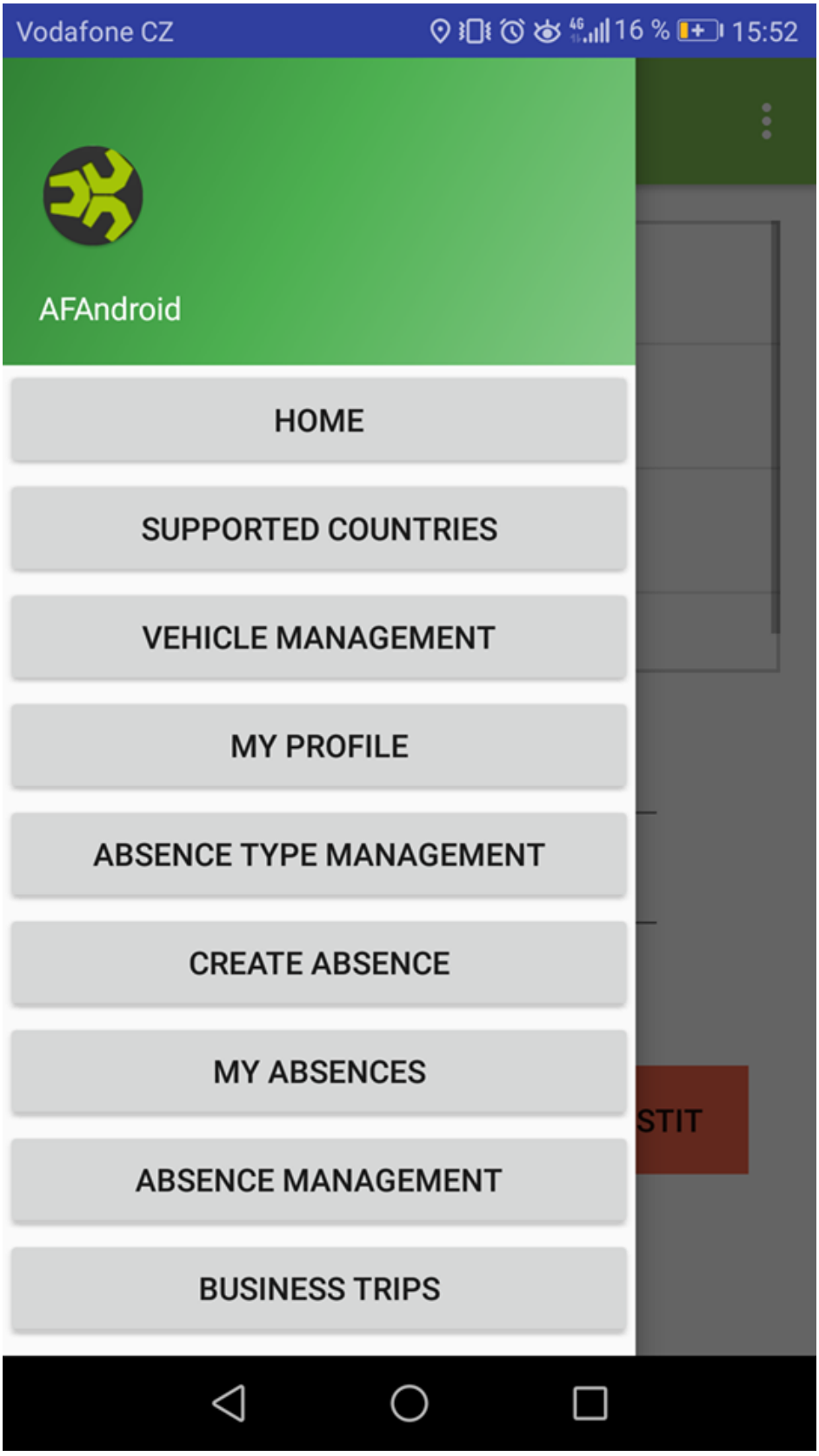
Obrázek B.6: Diagram tříd zachycující strukturu klasifikační části proxy aplikace využívající návrhový vzor Factory



Obrázek B.7: Sekvenční diagram popisující proces klasifikace metamodelu

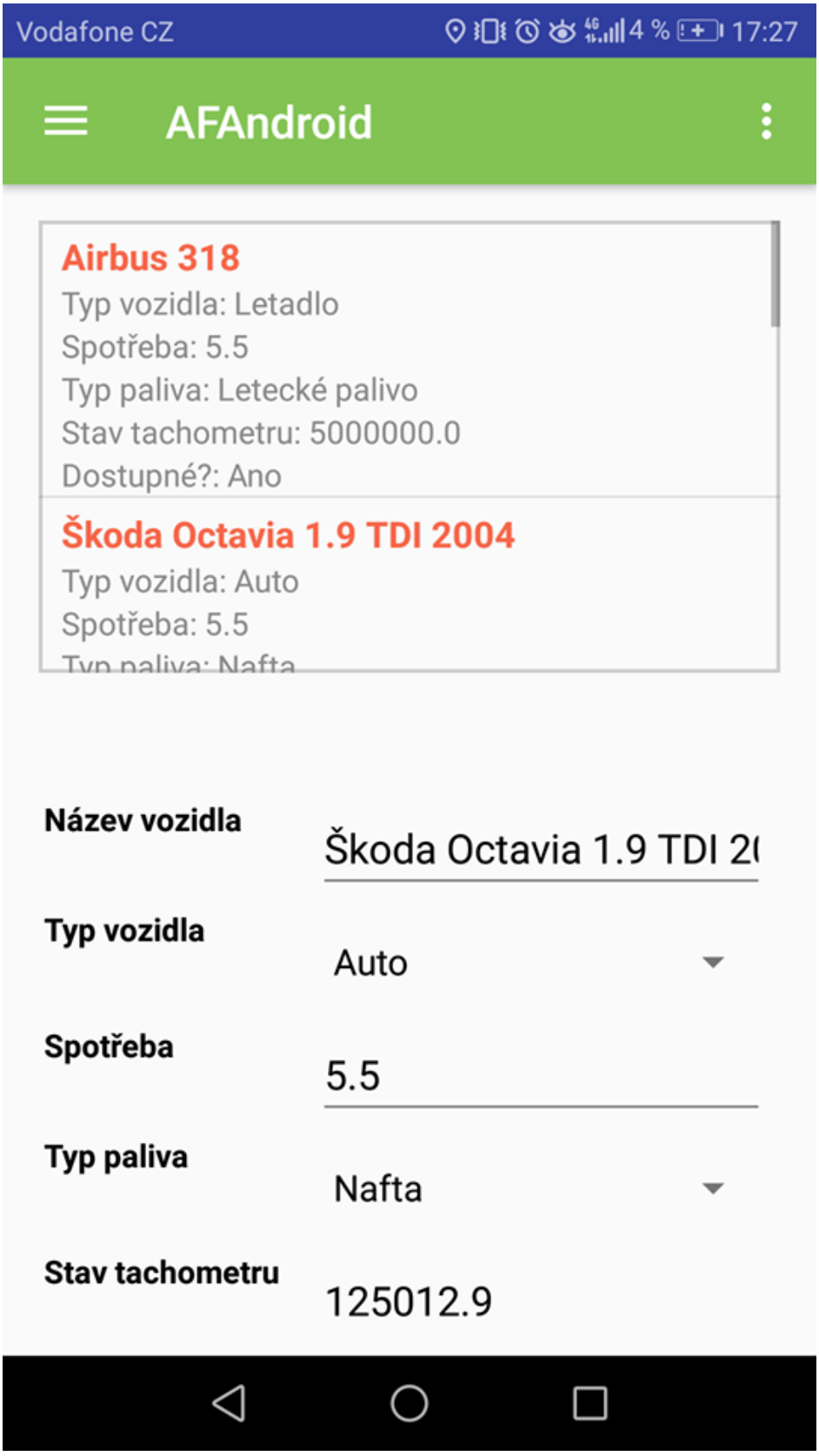


Obrázek B.8: Diagram tříd popisující strukturu uložení metadat



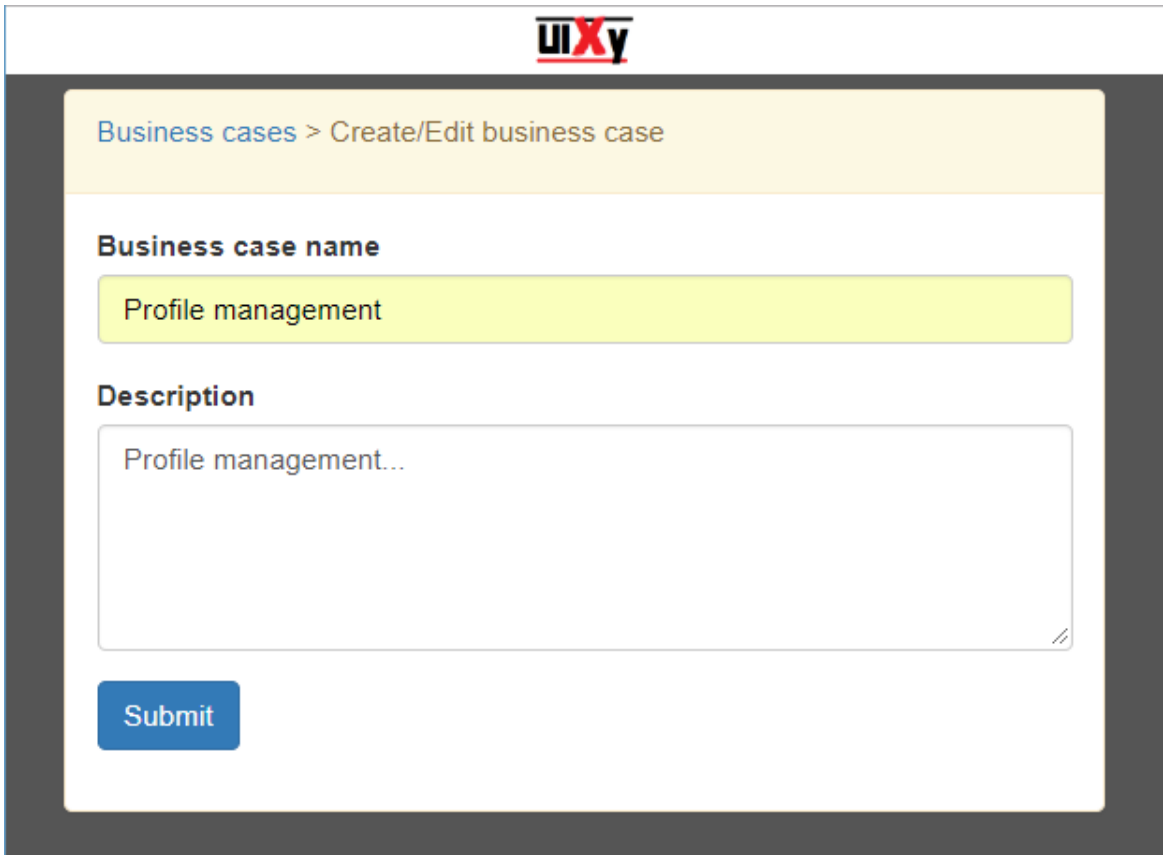
Obrázek B.9: Vizualizace menu aplikace, které je generováno a poskytováno proxy aplikací

Obrázek B.10: Nové obrazovky klientské aplikace pro správu služebních cest

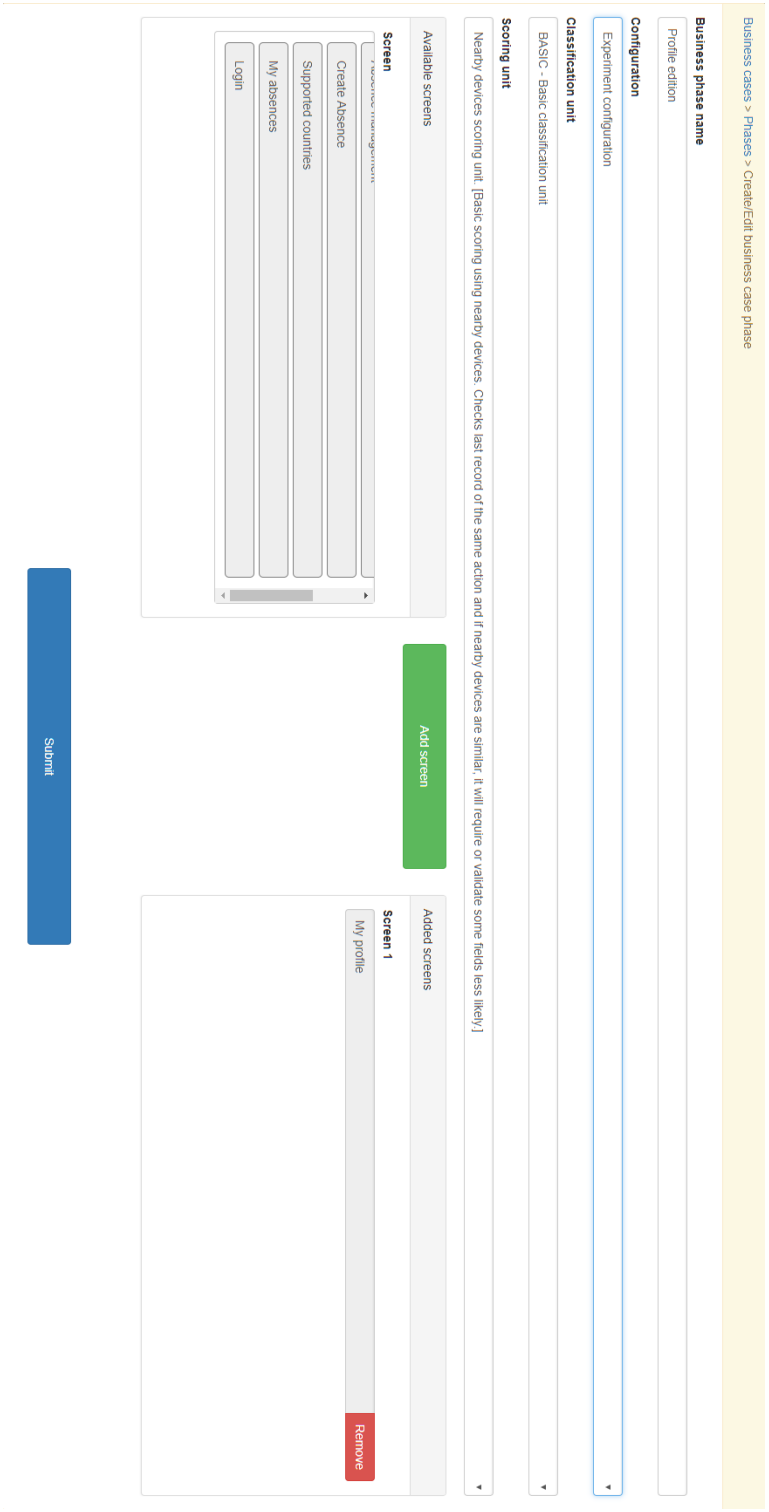


Obrázek B.11: Nová obrazovky klientské aplikace pro správu vozidel

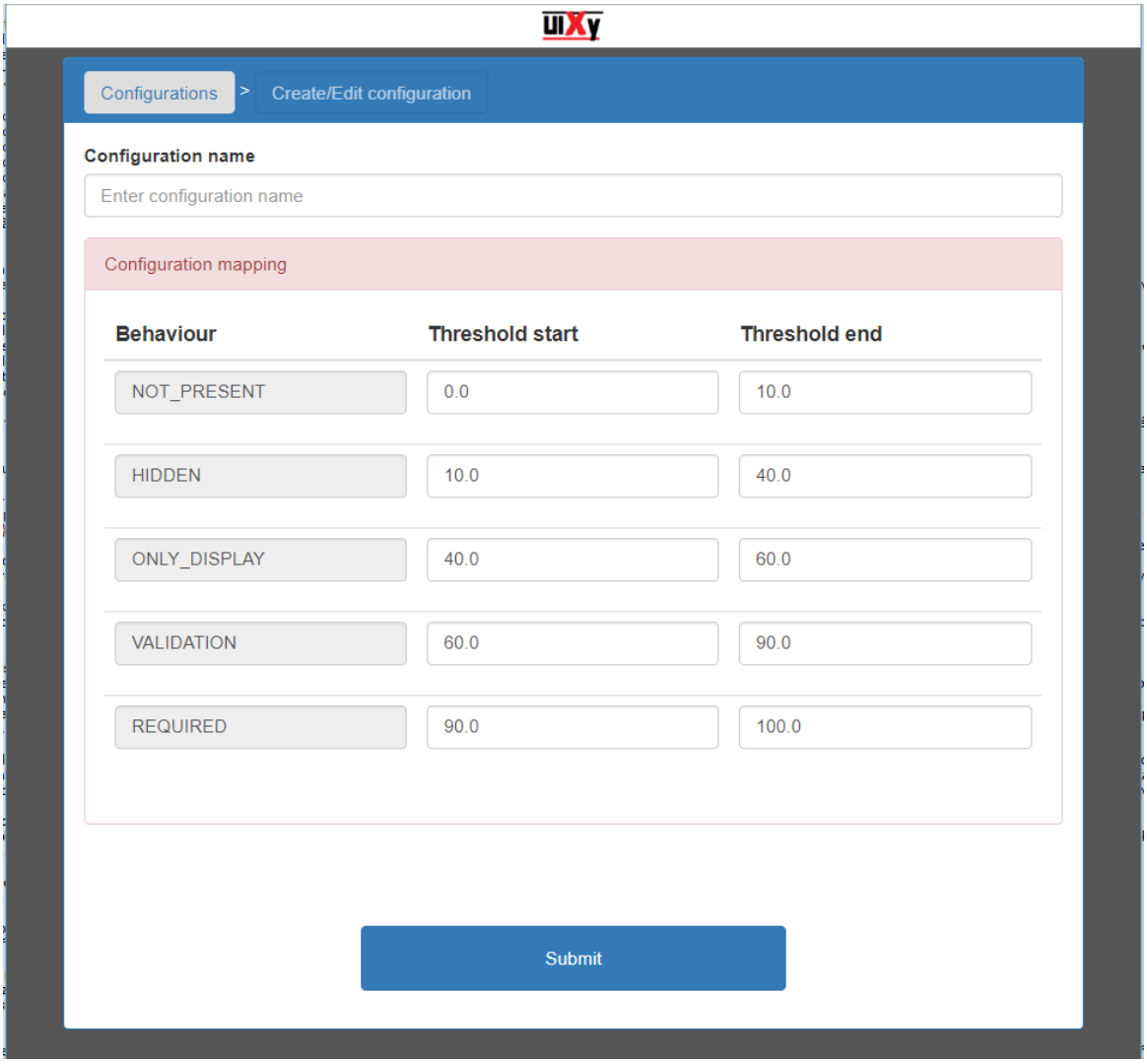
Obrázek B.12: Část uživatelského rozhraní proxy aplikace týkající se správy obrazovek



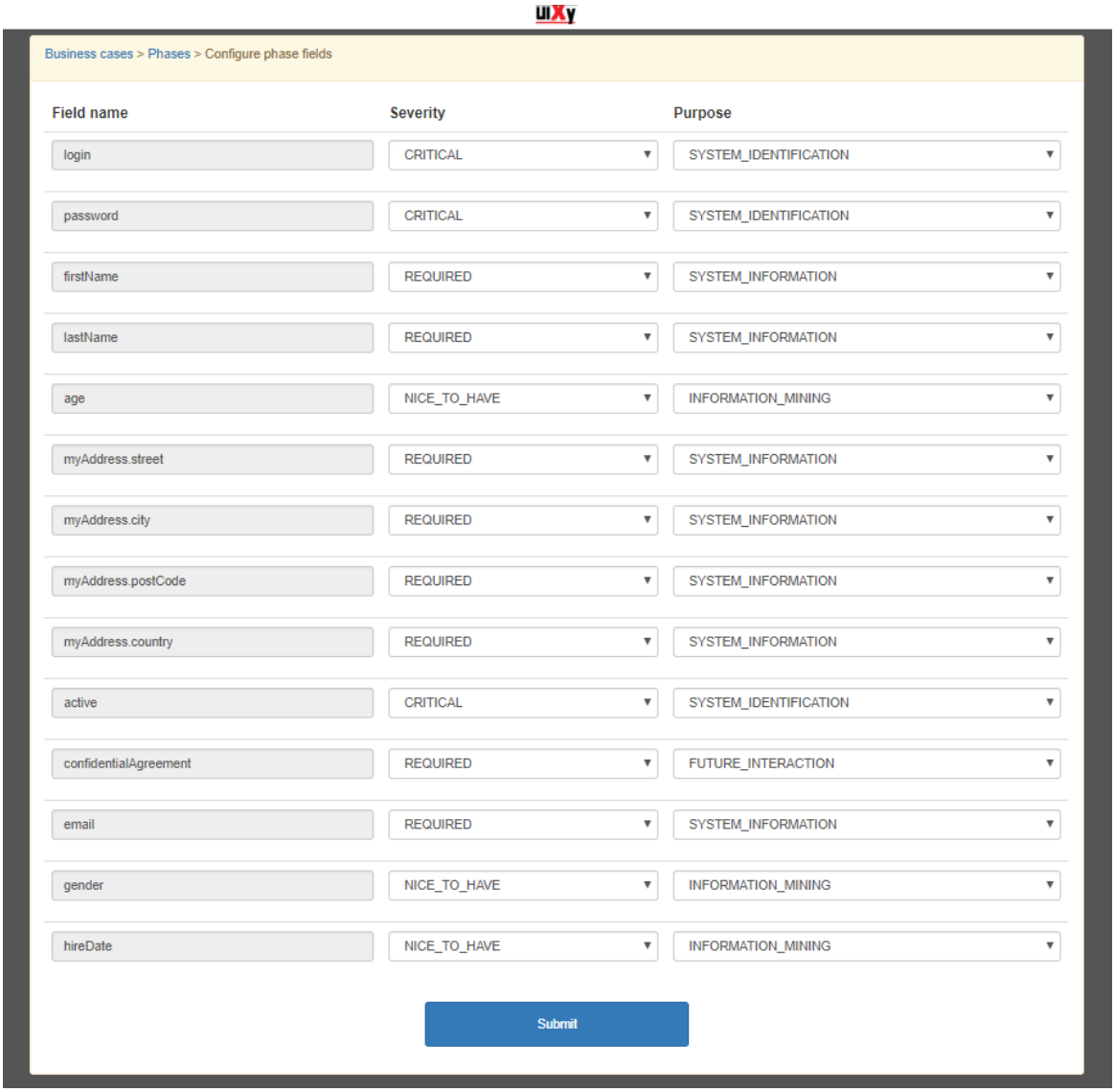
Obrázek B.13: Formulář pro vytvoření a editaci business případu v proxy aplikaci



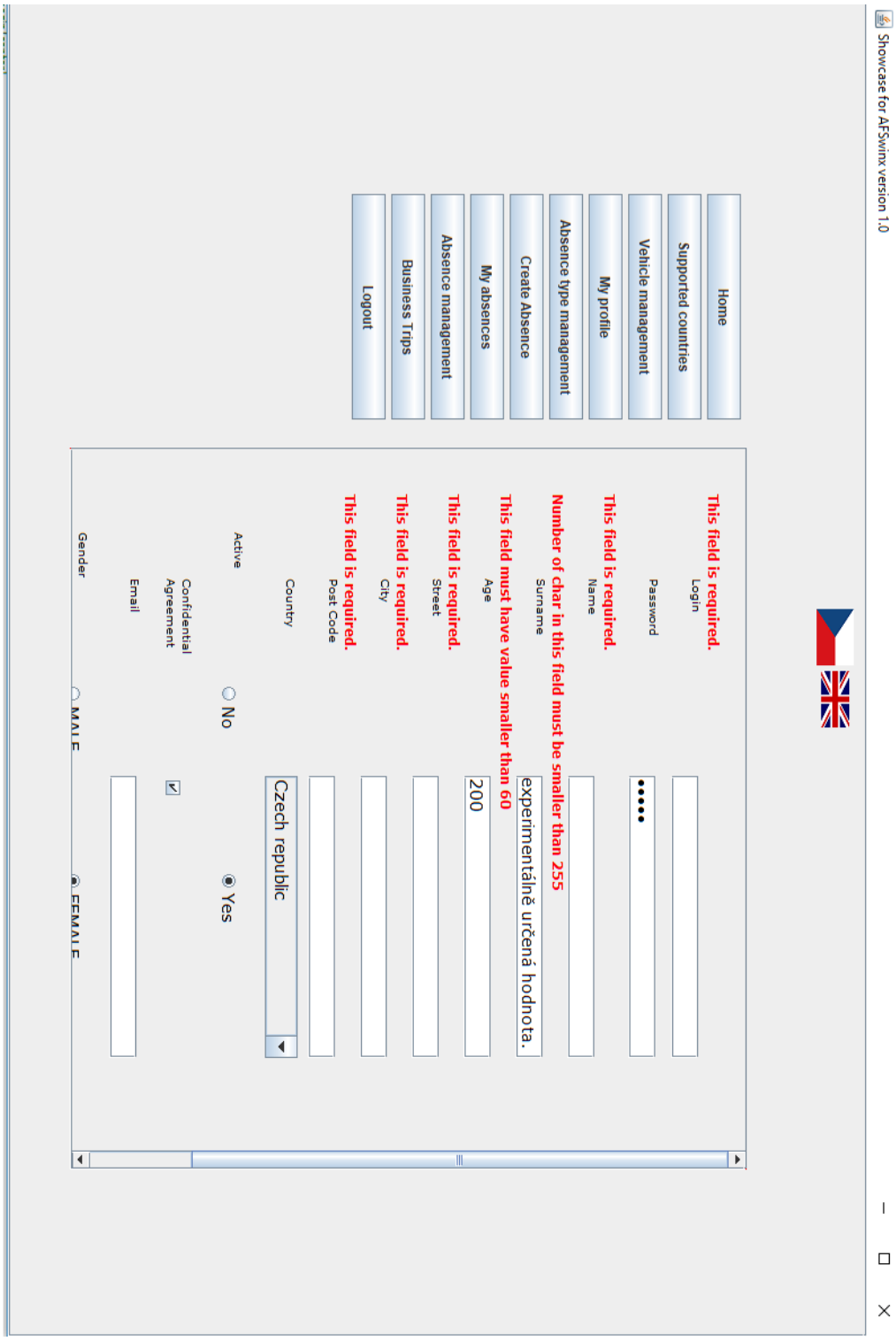
Obrázek B.14: Formulář pro vytvoření a editaci business fáze v proxy aplikaci



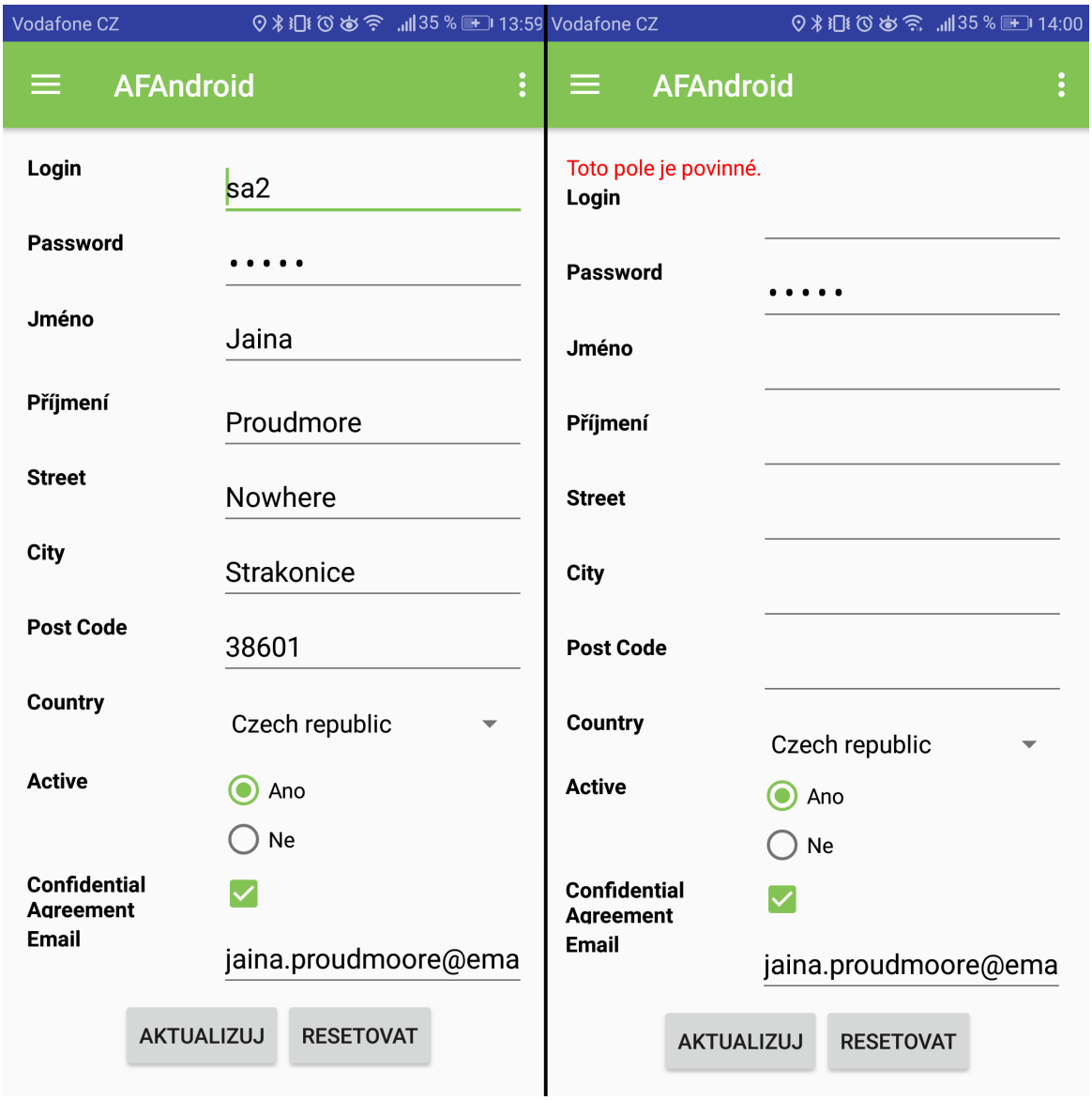
Obrázek B.15: Formulář pro vytvoření a konfigurace chování jednotlivých polí v upravené komponentě



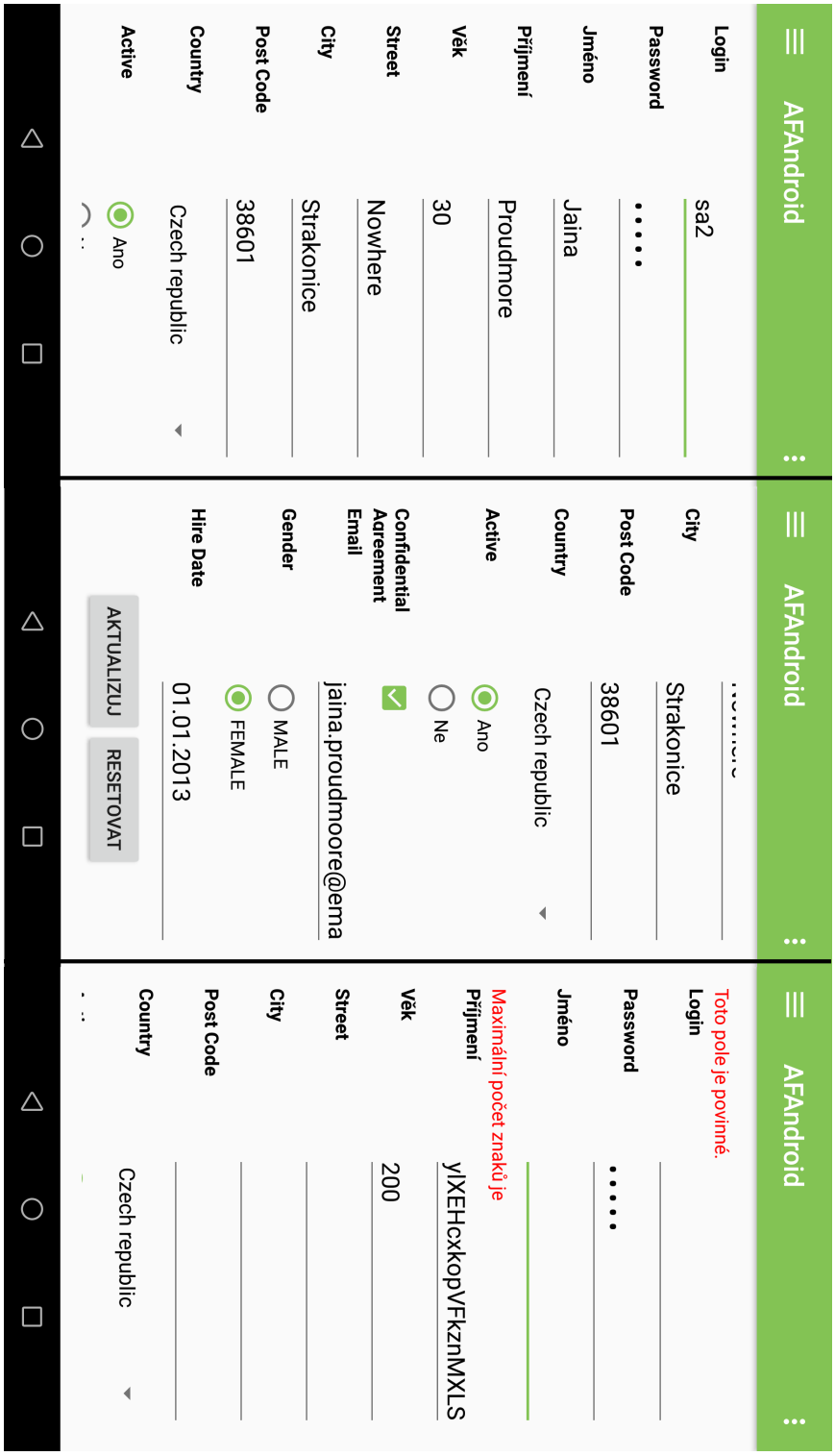
Obrázek B.16: Konfigurace business vlastností polí jednotlivých komponent vyskytujících se v dané business fázi



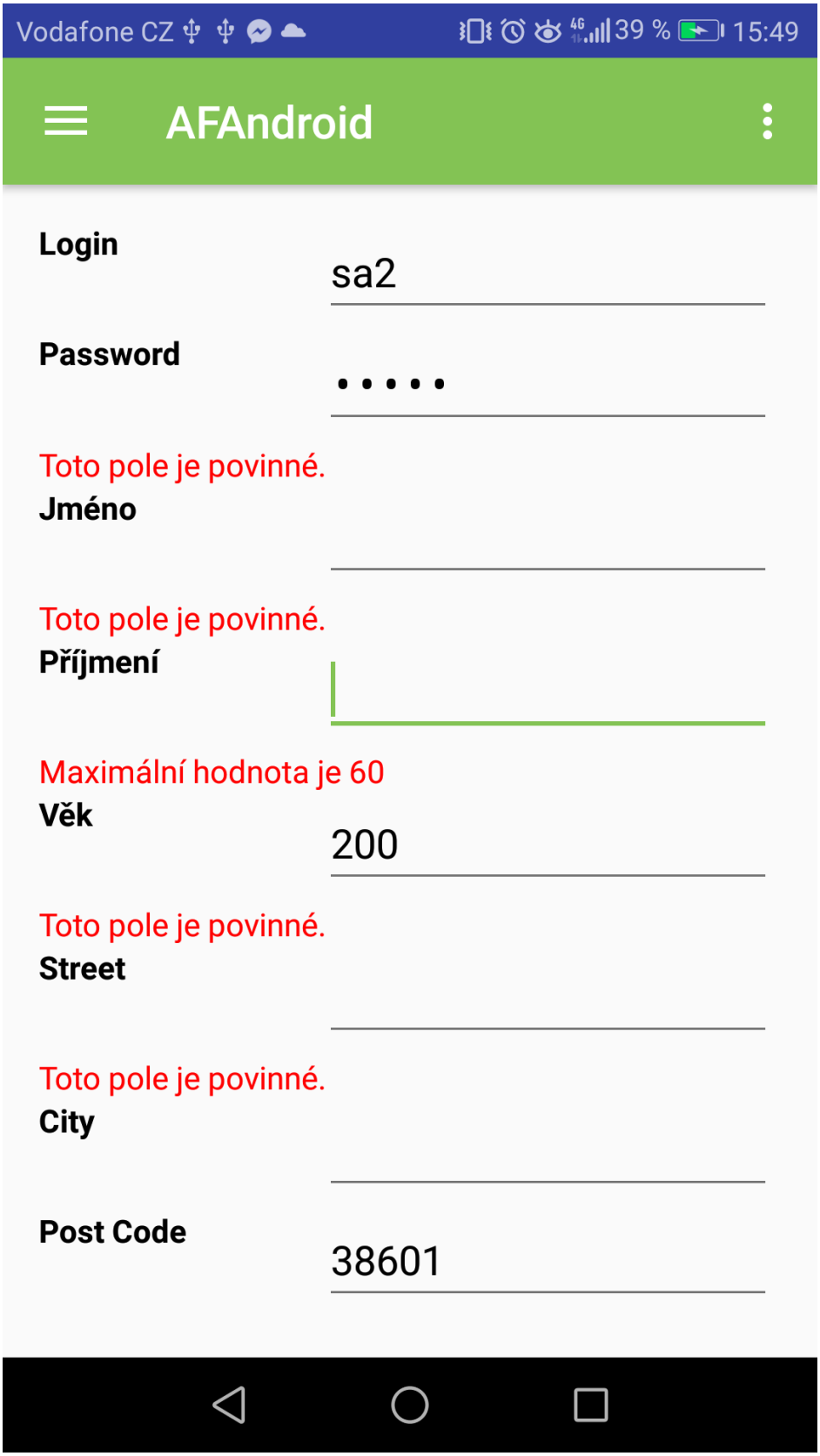
Obrázek B.17: Profilový formulář v Java SE aplikaci



Obrázek B.18: Výsledek prvního běhu experimentu v klientské Android aplikaci



Obrázek B.19: Výsledek druhého běhu experimentu v klientské Android aplikaci



Obrázek B.20: Výsledek třetího běhu experimentu v klientské Android aplikaci

Příloha C

Ukázky zdrojových kódů

V této sekci naleznete ukázky zdrojového kódu, na které bylo v textu odkazováno.

Listing C.1: Příklad použití AFNearbyStatus frameworku

```
NearbyStatusFacade nearbyStatusFacade =
    ↪ NearbyStatusFacadeBuilder.getInstance()
     .initialize(getApplicationContext())
      //mine info about battery
     .addStatusMiner(new BatteryStatusMiner())
      //mine info about location
     .addStatusMiner(new LocationStatusMiner())
      //mine info about network
     .addStatusMiner(new NetworkStatusMiner())
      //mine info about user status in application
     .addStatusMiner(new ApplicationStateMiner() {
         @Override
         public String getUsername() {
             //get current username
         }

         @Override
         public String getAction() {
             //get current user action
         }
       })
       //find bluetooth devices
       .addNearbyDevicesFinder(new BTDevicesFinder())
       //find nearby wifi networks - needs 20% or more battery
          ↪ capacity
       .addNearbyDevicesFinder(new NearbyNetworksFinder(), 20)
       //find devices on same wifi network - needs 30% or more
          ↪ battery capacity
       .addNearbyDevicesFinder(new SubnetDevicesFinder(), 30)
       //end process after 10 seconds
       .setRecommendedTimeoutForNearbySearch(10000)
       //run every 3 minutes
       .executePeriodically(1000 * 60 * 3)
       //send data to this url
       .sendDataToServer("www.example.com")
       .build();

nearbyStatusFacade.runProcess();
```

Listing C.2: Google FIT Sensors API - příklad zaregistrování posluchače na určitý typ dat

```
Fitness.getSensorsClient(
    getActivity(),
    GoogleSignIn.getLastSignedInAccount(getActivity()))
   .findDataSources(new DataSourcesRequest.Builder()
       .setDataTypes(DataType.TYPE_BPM)
       .setDataSourceTypes(DataSource.TYPE_RAW)
       .build())
   .addOnSuccessListener(
       new OnSuccessListener<List<DataSource>>() {
          @Override
          public void onSuccess(List<DataSource> dataSources) {
             OnDataPointListener mListener =
                new OnDataPointListener() {
                   @Override
                    public void onDataPoint(DataPoint
                       ↪ dataPoint) {
                     //process data from sensors
                    }
                };
             for (DataSource dataSource : dataSources) {
                //Data source found
                Fitness.getSensorsClient(
                   getActivity(),
                   GoogleSignIn.getLastSignedInAccount(
                      getActivity()
                   ))
                  .add(new SensorRequest.Builder()
                  .setDataSource(dataSource)
                  .setDataType(dataType)
                  .setSamplingRate(10, TimeUnit.SECONDS)
                  .build(), mListener)
                  .addOnCompleteListener(
                     new OnCompleteListener<Void>() {
                     @Override
                     public void onComplete(@NonNull Task<Void>
                        ↪ task) {
                        //listener registered
                     }
                  );
             }
          }
      }
})
```

Listing C.3: Příklad konfigurace komunikace s proxy aplikací v klienstkých aplikacích využívající framework AFAndroid

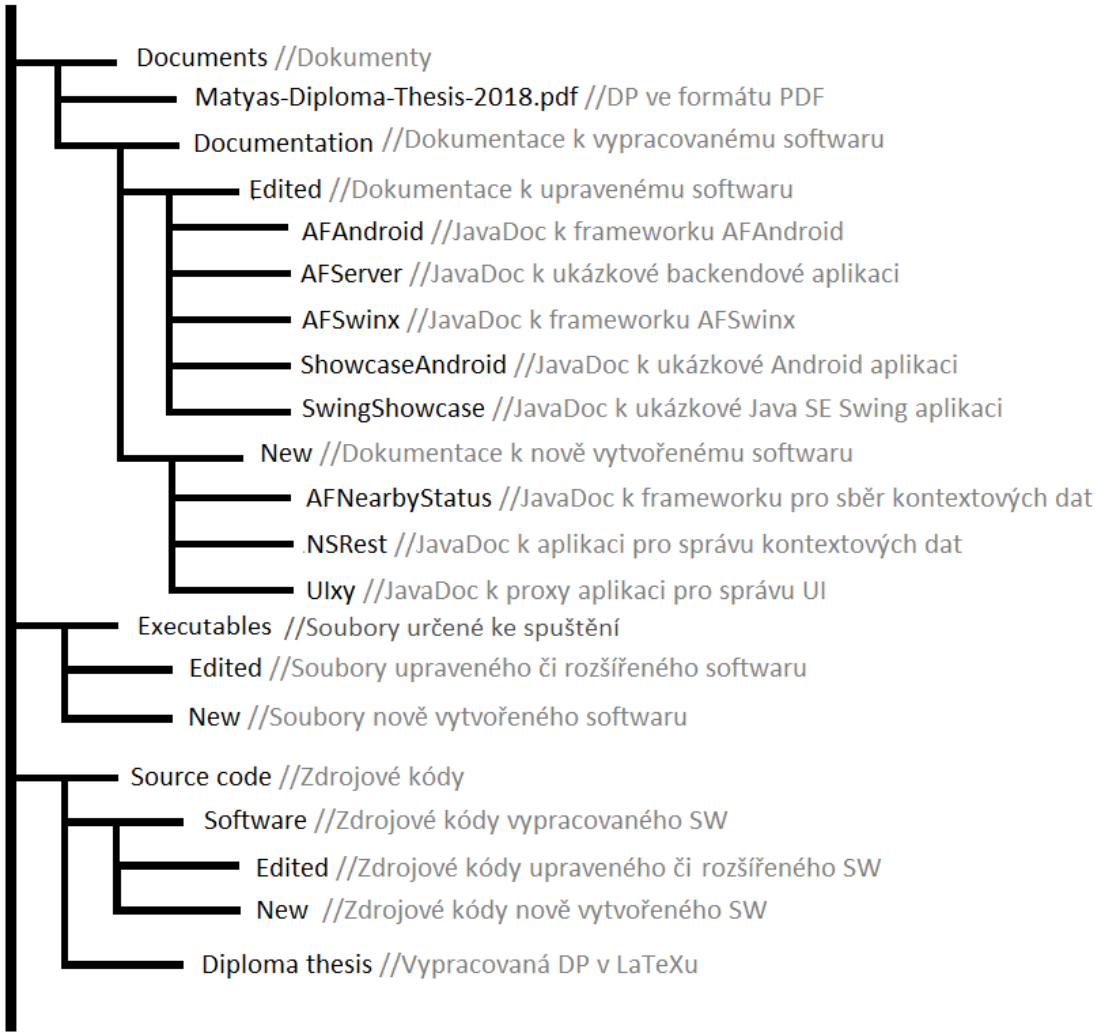
```
1  public class ApplicationContext extends UIProxySetup {
2
3      public static final String APP_CONFIG_FILE =
           ↪ "application.properties";
4
5      @Override
6      protected String loadUIProxyApplicationUuid(Context
           ↪ context) {
7          //this value is actualy loaded from properties file
8          return "4c7649c1-45b2-40f3-8b94-98fba303c1f0";
9      }
10
11     @Override
12     protected String loadUIProxyUrl(Context context) {
13         //this value is actualy loaded from properties file
14         return "http://81.2.216.197:8080/UIxy";
15     }
16
17     @Override
18     protected Device loadDeviceType(Context context) {
19         if(Utils.deviceHasTabletSize(context)){
20             return Device.TABLET;
21         } else {
22             return Device.PHONE;
23         }
24     }
25
26     @Override
27     protected String loadDeviceIdentifier(Context context) {
28         return NetworkUtils.getMacAddress();
29     }
30
31     @Override
32     protected String loadNearbyAppUrl(Context context) {
33         //this value is actualy loaded from properties file
34         return "http://81.2.216.197:8080/NSRest"
35     }
36
37 }
```

Listing C.4: Úryvky z logů proxy aplikace vzniklé při provádění experimentů

```
//Experiment 1
Classifing field: gender
START score for purpose INFORMATION_MINING and severity NICE_TO_HAVE is 46.0
New context data has 100.0% of last context data nearby devices
It is considered to be SAME setup. Similarity was from 90.0 to 100.0
FINAL score for purpose INFORMATION_MINING and severity NICE_TO_HAVE is 11.0
Behavior HIDDEN was chosen! Config thresholds were from 10.0 to 40.0
Clearing validations of gender and hiding field

Classifing field: email
START score for purpose SYSTEM_INFORMATION and severity REQUIRED is 88.0
New context data has 100.0% of last context data nearby devices
It is considered to be SAME setup. Similarity was from 90.0 to 100.0
FINAL score for purpose SYSTEM_INFORMATION and severity REQUIRED is 53.0
Behavior ONLY_DISPLAY was chosen! Config thresholds were from 40.0 to 60.0

//Experiment 2
Classifing field: gender
START score for purpose INFORMATION_MINING and severity NICE_TO_HAVE is 46.0
New context data has 46.42857142857143 % of last context data nearby devices
It is considered to be MORE_DIFFERENT setup. Similarity was from 30.0 to 50.0
FINAL score for purpose INFORMATION_MINING and severity NICE_TO_HAVE is 41.0
Behavior ONLY_DISPLAY was chosen! Config thresholds were from 40.0 to 60.0
Clearing validations of gender

Classifing field: email
START score for purpose SYSTEM_INFORMATION and severity REQUIRED is 88.0
New context data has 46.42857142857143 % of last context data nearby devices
It is considered to be MORE_DIFFERENT setup. Similarity was from 30.0 to 50.0
FINAL score for purpose SYSTEM_INFORMATION and severity REQUIRED is 83.0
Behavior VALIDATION was chosen! Config thresholds were from 60.0 to 90.0
Disabling REQUIRED on field email

//Experiment 3
Classifing field: gender
START score for purpose INFORMATION_MINING and severity NICE_TO_HAVE is 46.0
New context data has 5.878957 % of last context data nearby devices
It is considered to be DIFFERENT setup. Similarity was from 0.0 to 10.0
FINAL score for purpose INFORMATION_MINING and severity NICE_TO_HAVE is 61.0
Behavior VALIDATION was chosen! Config thresholds were from 60.0 to 90.0
The field gender has behavior: VALIDATION
 Disabling REQUIRED on field gender

Classifing field: email
START score for purpose SYSTEM_INFORMATION and severity REQUIRED is 88.0
New context data has 5.878957 % of last context data nearby devices
It is considered to be DIFFERENT setup. Similarity was from 0.0 to 10.0
FINAL score for purpose SYSTEM_INFORMATION and severity REQUIRED is 100.0
Behavior REQUIRED was chosen! Config thresholds were from 90.0 to 100.0
Enabling REQUIRED on field email
```

Příloha D

Obsah přiloženého CD



Obrázek D.1: Obsah přiloženého CD