Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Herní aplikace pro demonstraci použití komponent na OS Android

Plzeň 2012 Jan Záruba

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2012

Bc. Jan Záruba

Abstract

My thesis Development of component-based mobile applications with OSGi is supposed to demostrate possibilities of usage OSGi framework in mobile application development. First part of my thesis summarizes prerequisities for using OSGi on Android. Then capabilities of data persistence on JPA are presented. Second practical part is aimed at development of gaming application, which consists of two parts: server application (OSGi on servlet container), client application (based on Android).

Keywords: android development, OSGi, JPA, smartphone

Obsah

1 Úvod

Žijeme v době, kdy každým dnem roste počet smartphonů a tabletů, a to přes očekávání některých odborníků. Rozvoj „chytrých“ mobilních zařízení nabízí obrovské množství služeb, které mohou tato zařízení zprostředkovávat (různé herní aplikace, propojení se sociálními sítěmi, propojení s různými ERP systémy).

Některé aplikace časem nabývají na složitosti a zároveň se jejich části velmi často opakují, například části komunikující se serverem, části komunikující se senzory zařízení atd. Zde se přímo nabízí myšlenka využít komponentového přístupu k vývoji aplikací, který poskytuje možnost účinné dekompozice aplikace, znovupoužití vytvořených komponent, případnou snadnou výměnu některé komponenty (komponenta se má chovat jako „blackbox“, tzn. známe rozhraní jak s komponentou komunikovat, ale ne jak je implementována vnitřní logika).

Jedním z cílů této práce bude prověřit možnosti tvorby aplikací zejména pro iOS a Android. Hlavním cílem bude vytvořit herní aplikaci, která bude demonstrovat architekturu a vývoj klient-server aplikace založené na komponentách, konkrétně takové, kdy komponentovým modelem je OSGi a klientská část je provozována na OS Android. Výstupem práce budou dále zkušenosti, popřípadě rady, získané při vývoji aplikace.

2 Platformy pro mobilní zařízení

Podle Ablesona v [1] jsou dnes na trhu dva typy mobilních zařízení:

- „hloupé“ telefony - feature phones

  Telefony, které slouží především na volání, případně posílání SMS zpráv, mohou mít například fotoaparát nebo na nich lze procházet webové stránky (zde ovšem většinou zákazník naráží na ne moc velký uživatelský komfort).

- „chytré“ telefony - smart phones

  Chytré telefony poskytují ty samé možnosti, co „hloupé“ telefony. Ale poskytují je trochu jiným způsobem. Většinou obsahují větší množství funkcí a jejich charakteristickou vlastností je „nutnost“ připojení na internet, bez kterého přichází o mnoho svých funkcí.

Dále bych se rád zabýval pouze „chytrými“ telefony. Na trhu máme několik rozšířených OS:

- iOS
- Android
- Windows Phone 7

Ve výčtu jsem nezmínil Symbian, který pravděpodobně skončí, protože společnost Nokia, která ho vyvíjela, přechází na Windows Phone 7. Dále jsou tu další menší OS jako Maemo a jeho nástupce MeeGo (také vyvíjené společností Nokia). Tyto systémy byly nasazeny na málo přístrojů a podle prohlášení čelních představitelů Nokia nebudou nasazeny již nikam.

2.1 iOS

iOS je operační systém pro zařízení vyráběné společností Apple. Oproti systému Android není otevřený a je vyvíjený pro konkrétní architekturu zařízení

vyráběných Applem, jako je v současné době iPhone a iPad. Tak jako se při vývoji pro Android používá programovací jazyk Java, tak pro iOS se využívá jazyk Objective-C.

2.1.1 Publikování aplikací

Abychom mohli aplikaci publikovat, musíme ji v případě iOS dát na App-Store. K tomu je nutná registrace do iOS Developer Program[1]. K dispozici jsou dvě možnosti: enterprise pro větší společnosti a standard, za který se v roce 2012 platí $ 99. V rámci této ceny vývojář dostane vývojové prostředí, přístup na fóra a možnosti dotazů na podporu společnosti Apple.

2.1.2 Vývoj aplikací

K tomu, abyste mohli vyvíjet aplikace pro nějaký z produktů Apple, je nutné vlastnit Mac, bez něho je vývoj nemožný. Apple poskytuje pro vývoj SDK, které obsahuje mimo jiné i integrované vývojové prostředí Xcode. Pro vývoj aplikací na iOS se používá Cocoa. Cocoa je aplikační prostředí jak pro Mac OS X tak i pro iOS, je to sada různých objektových frameworků, které se používají pro běh aplikací na Mac OS X a iOS. Obsahuje velké množství tříd, na základě kterých lze vytvářet robustní i poměrně rozsáhlé aplikace.

Základním programovacím jazykem je Objective-C, ale lze využít i jiných. Díky tomu, že je Objective-C založený na jazyku ANSI C, lze bez problémů využít zdrojového kódu z ANSI C. Dále lze volat funkce z různých již zkompilovaných knihoven, dokonce lze do kódu zamíchat i části kódu z C++.

Cocoa obsahuje 2 nejdůležitější balíky tříd [19]

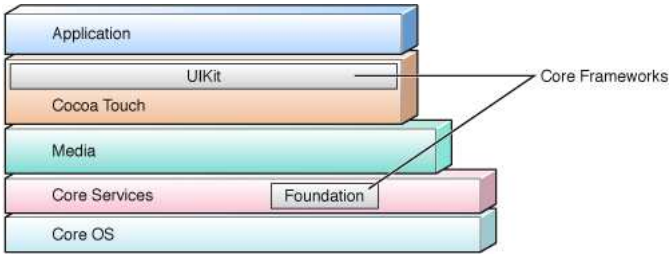
- Pro Mac OS X jsou to Foundation a AppKit
- Pro iOS jsou to Foundation a UIKit

Rozdělením knihoven lze dosáhnout oddělení prezentační vrstvy, což například poskytuje možnost, aby aplikační logika, popřípadě zpracování dat, bylo stejné pro zařízení na Mac OS X a iOS.

[1] https://developer.apple.com/programs/ios/

Architektura Cocoa na iOS

Následující obrázek(2.1) ukazuje jak vypadá Cocoa na zařízení iOS.



Obrázek 2.1: Cocoa v architektuře iOS[2]

- Core OS

  Jádro systému obsahuje, jak lze předpokládat, základní funkce jako je správa souborového systému, přístup k síti...

- Core Services

  Core services poskytují služby, které umožňují zařízení různé funkce s řetězci a jinými programovými strukturami a také přístup k hardwaru zařízení (GPS sensory, síťová komunikace).

- Media

  Jak napovídá název vrstvy poskytuje různé grafické a multimediální služby.

- Cocoa Touch

  Vrstva, na které přímo běží koncová aplikace.

Objective-C

Je objektový jazyk, ve kterém se píšou aplikace pro iOS. Jazyk a jeho syntaxe je založena na jazyku ANSI C (je jeho nadmnožinou)[20], od kterého převzal například primitivní datové typy jako int, float... Objective-C je proti ANSI C rozšířené o objekty a zasílání zpráv z jazyka Smalltalk.

[2]převzato z http://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/CocoaFundamentals/Art/architecture_stack.jpg dne 30.4.2012

Jistě zajímavou vlastností Objective-C je několik možností, jak spravovat paměť':

- ARC (Automatic Reference Counting) - překladač sám řídí životní cyklus objektů.
- MRC (Manual Reference Counting) - Někdy se nazývá také MRR (Memory Retain/Release). Programátor má objekty plně pod kontrolou a zároveň je plně odpovědný za životní cyklus objektů.
- Garbage Collection - V tomto režimu předává programátor správu objektů sběrači, který se za běhu programu stará o životní cyklus objektů.

Pro srovnání uvedu pár rozdílných zápisů mezi C++ a Objective-C

C++	Objective-C
#include "library.h"	#import "library.h"
this	self
private:	@private
protected:	@protected
public:	@public
Y = new MyClass();	Y = [[MyClass alloc] init];
try, throw, catch, finally	@try, @throw, @catch, @finally

Tabulka 2.1: Rozdíly zápisů mezi C++ a Objective-C[3]

Z uvedené tabulky (tabulka 2.1) lze vidět podobnosti i rozdílnosti zápisu v obou jazycích.

2.2 Android

Android je platforma pro mobilní zařízení jako jsou smartphony nebo tablety obsahující operační systém, middleware a další aplikace[18]. Operační systém Android je vyvíjen konsorciem Open Handset Alliance, které je vedené společností Google. Systém je vydáván pod open-source licencí a je

[3]převzata z http://www.codeproject.com/Articles/88929/Getting-Started-with-iPhone-and-iOS-Development - tabulka popisující rozdíl mezi C++/Objective-C 30.4.2012

založen na jádru operačního systému Linux. Při vývoji muselo být myšleno na omezení, která plynou z užívaní na mobilních zařízeních jako je hlavně výdrž baterie, menší výkon v porovnání s desktopovými systémy nebo méně paměti. Android je vytvořen tak, aby mohl běžet na různých typech hardware.

2.2.1 Historie

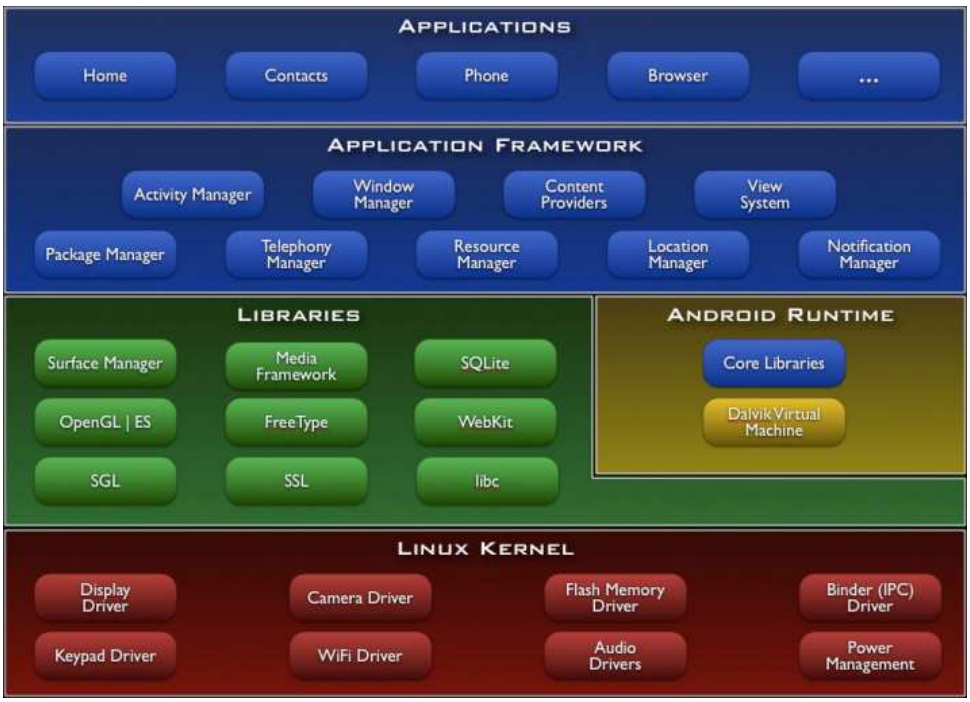
Krátce bych zmínil historický vývoj, který vedl k současnému stavu systému Android. Vše začalo založením Android Inc. v Palo Altu ve Spojených Státech Amerických v říjnu roku 2003. Společnost byla založena skupinou lidí, kteří měli zkušenosti s komunikacemi a mobilním zařízeními, a nějakou dobu vyvíjela téměř tajně operační systém na mobilní platformy. V srpnu roku 2005 firmu plně převzala společnost Google Inc. a její zaměstnanci se stali zaměstnanci Googlu, který měl již v té době nejspíše v úmyslu vstoupit na trh s mobilními zařízeními. V listopadu 2007 byla vytvořena Open Handset Alliance, vedená Googlem, ale sdružující 34 členů[12], kteří mají něco společného s vývojem pro mobilní zařízení (výrobci zařízení, tvůrci software, výrobci dalšího hardware apod.). Vlajkovou lodí tohoto sdružení je Android[2].

2.2.2 Architektura systému Android

Android je připraven hlavně pro hardwarovou architekturu ARM, ačkoliv existují i projekty pro použití Androidu na x86. Při dalším popisu architektury bych se rád držel poměrně známého obrázku (viz obrázek 2.2) architektury systému Android, jak je zveřejněna na jejich stránkách [18]. Jádro operačního systému je napsáno v C/C++, ale uživatelské i vestavěné aplikace jsou psány v Javě.

Linuxové jádro

Systém je založen linuxovém jádru. Oproti běžnému Linuxu zde ovšem chybí systém X Window, také musely být upraveny některé vlastnosti kvůli řízení spotřeby, které je na mobilních zařízení naprosto kritické.



Obrázek 2.2: Architektura systému Android[4]

Aplikace

Součásti balíku, ve kterém je systém dodáván, je samozřejmě i množství aplikací. Tyto aplikace poskytují základní funkčnosti zařízení jako je psaní SMS zpráv, správa kontaktů, dnes již asi nezbytný webový prohlížeč apod. Tyto aplikace mohou být nahrazeny jinými aplikacemi šířenými buď přes Google Play (dříve Android market), nebo přímo přes .apk soubory.

Aplikační framework

K systému Android je k dispozici Software Development Kit (zkráceně SDK). Vývojáři pro tvorbu svých aplikací mohou využít prakticky neomezeně zdroje zařízení, které jsou k dispozici prostřednictvím API. Aplikační framework právě toto API poskytuje. Pod prakticky každou aplikací na Android běží sada služeb a systémů. Například bych zmínil systém View, který poskytuje uživatelské rozhraní. Tyto knihovny obsahují poměrně velké množství

[4]převzato z http://developer.android.com/ dne 18.4.2012

různých komponent grafického uživatelského prostředí (formulářové komponenty, seznamy, různé layouty atd.).

Knihovny

Android poskytuje sadu různých knihoven, které jsou psané v C/C++, a poskytují různé další možnosti pro vývojáře Androidu jako jsou manipulace s různými médii, 2D a 3D grafika a databáze.

Android runtime

Na obrázku malá, avšak velmi podstatná součást systému. Pro chod aplikací je zde použit Dalvik Virtual Machine. Dalvik Virtual Machine používá registrovou architekturu na rozdíl od Java VM, který má zásobníkovou architekturu. Další vlastností Dalvik VM je využívání Just-in-time compilation. Dalvik VM spouští tzv. Dalvik Executable (také dex-code nebo jen dex) soubory. DEX soubory lze vytvořit z javovských .class souborů, tedy z javovského bytecodu.

2.2.3 Vývoj aplikaci pro Android

Aplikace pro Android se vyvíjí v jazyce Java. Android je postaven tak, že veškeré aplikace (samozřejmě kromě jádra) používají stejné API. To v praxi znamená, že každý vývojář může vytvářet aplikace, které jsou stejně dobré, rozsáhlé, popř. mocné jako jsou vestavěné aplikace. To je jedna z opravdu velkých předností Androidu.

Pro účely vývoje je možné (a nutné) stáhnout SDK ze stránek Android[5]. SDK obsahuje základní nástroje pro vývoj jako je například emulátor a nástroje pro jeho správu. K vývoji je samozřejmě nutné mít nainstalováno Java JDK, pro překlad zdrojových kódů. Výrobce doporučuje používat Eclipse IDE[6], pro který je k dispozici plugin, který umožňuje vývoj aplikací na Android přímo v IDE.

[5] http://developer.android.com
[6] http://www.eclipse.org/

Publikace aplikací

Ve chvíli, kdy programátor dokončí aplikaci, nebo i dříve, se začne zajímat o to, jak svoji aplikaci publikovat. Zde se nabízí několik možností:

- Šířit pouze apk soubor

  Výslednou aplikaci je možné exportovat do apk archivu, který, když se přenese na zařízení, je jednoduše nainstalovatelný.

- Dát aplikaci volně k dispozici na Google Play (dříve Android Market)

  Výslednou aplikaci je možné dát k dispozici na Google Play, což je obdoba AppStore společnosti Apple pro iOS. Instalace je potom pro uživatele jednodušší, aplikaci si jednoduše najde, zmáčkne nainstalovat a o více se nemusí starat (Google Play potom hlídá i případnou dostupnost aktualizací).

- Prodávat aplikace na Google Play (dříve Android Market)

  Aplikace může být na Google Play zpoplatněna. Princip je úplně stejný jako v minulém bodě, zmiňuji ho jenom kvůli jedné poznámce. U nás v ČR byla podpora prodávání na Google Play slabá, skoro žádná (některé společnosti to obcházely například tím, že si zakládaly účty například v sousedním Německu). Dnes už je zavedená plná podpora vývojářů, takže s prodejem aplikací není žádný větší problém.

Základní prvky aplikací na Android

Každá aplikace na Android obsahuje (používá) určité prvky (lze říci, že každá aplikace používá alespoň jeden z těchto objektů).

- Intent

  Dalo by se říci, že každá aplikace využívá principu Intentů. Intent (angl. záměr nebo úmysl) se používá, pokud chceme v aplikaci například něco otevřít. Intent je vlastně model události, na kterou systém reaguje.

  Intenty lze dělit v zásadě do dvou skupin:

  - Implicitní

    Zde je znám účel operace (například výběr kontaktu v telefonu). Vyhodnocení a výběr aplikace zajišťuje systém sám.

  - Explicitní

    V tomto případě znám přesně aktivitu (třídu), která má na Intent reagovat. Tudíž vyhodnocení se přesouvá od systému k programátorovi.

  Pokud se zmiňujeme o Intentech, nemůžeme se nezmínit o Intent filtrech. Tyto filtry mohou jistým způsobem rozšiřovat funkčnost systému. Říká vlastně systému, že je tu něco, co může využít k zpracování Intentu. K rozšiřování systému by se, ale mělo přistupovat opatrně ([1]).

- Activity

  Třída Activity je jednou ze základních tříd obsažená v Runtime. Každá aplikace, která má UI, musí mít i alespoň jednu aktivitu. Aktivitu lze velmi často chápat jako jednu obrazovku na zařízení, ale je možné je využít i jinak. Všechny komponenty UI jsou potomky třídy View. Activity je spouštěna pomocí Intentu příkazem startActivity(Intent).

  Každá Activita má svůj životní cyklus, který je dobře vidět na obrázku 2.3. Z obrázku lze vidět, že aplikace může být kdykoliv zastavena systémem, pokud usoudí, že potřebuje její prostředky. Je zde jasná hierarchie aplikací, jak budou zavírány:

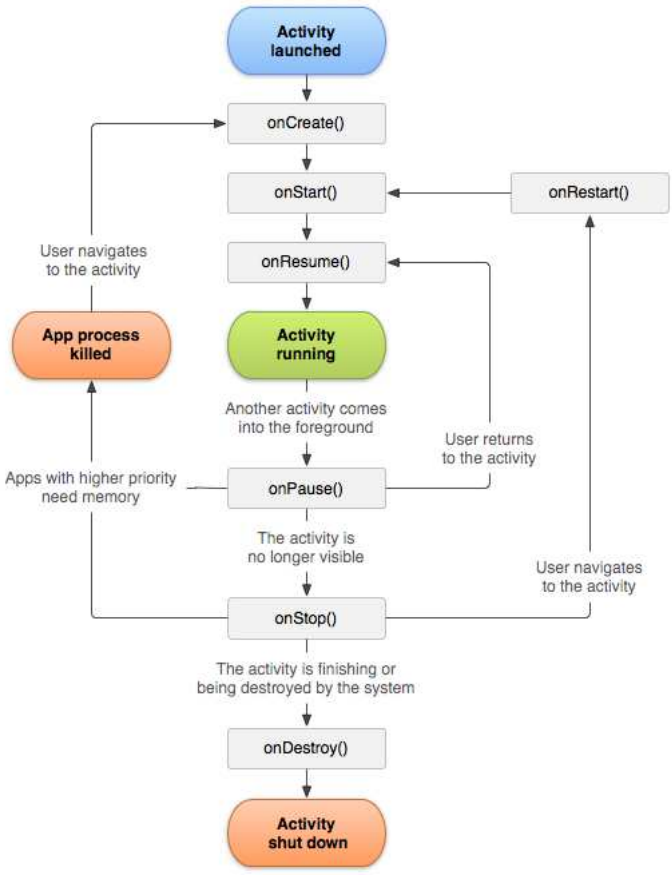
  - Nejdůležitější jsou aktivity, které uživatel přímo používá, tzn. jsou zobrazeny
  - Další jsou aktivity, které jsou jen vidět (například jsou vidět aktivity, které jsou pod modálním dialogovým oknem).
  - Pak následují ostatní.

- Service

  V mnohém je podobná activity, ale většinou běží na pozadí, vykonává buď nějakou periodickou nebo jednorázovou činnost, která je časově náročná. Spouští se podobně jako Activity pomocí Intentu. Hlavní rozdíl oproti Activity je, že nemá UI.

- Broadcast receiver

  Podobně jako service nemá UI. Funguje jako příjemce událostí v systému. Použijeme ho například, pokud bychom si chtěli napsat vlastního SMS klienta. Po registraci není nutné, aby aplikace běžela. Aplikace reaguje na podněty ze systému (pro zaregistrování do systému se používá výše zmíněný Intent filter). Hlavní metodou této třídy je *onReceive*, většinou neobsahuje rozsáhlý kód.



Obrázek 2.3: Životní cyklus aktivity[7]

- Content provider

  Používá se hlavně, pokud je potřeba sdílet data aplikace s jinými aplikacemi na zařízení. Content provider může poskytovat data jak Activity, tak i Service. Sdílení dat přímo pomocí souborového systému se nedoporučuje, aplikace vyžaduje povolení na přístup k souborovému systému a i tak může být přístup kvůli bezpečnosti systémem omezen. Content provider může poskytovat velké množství různých typů dat - soubory, databáze...

[7]převzato 26.4.2012 z http://developer.android.com/images/activity_lifecycle.png

- AndroidManifest

  Manifest není v pravém slova smyslu komponentou aplikace, ale je její nedílnou součástí. Po tom, co jsme si představili základní stavební prvky aplikace, přichází na řadu něco, co je spojí dohromady, a tím je *AndroidManifest*.xml. Možností, co lze dát do manifestu je mnoho, ale potřeby této práce ho jen krátce představím.

  Základní informací obsaženou v manifestu je název aplikace. Dále pak musí obsahovat všechny activity (pokud má aplikace UI) s jejich popisem - jméno, třída, dále pak popis dalších dříve zmíněných základních prvků aplikace. Další velmi důležitou částí jsou použité knihovny, samozřejmě velmi často není nutné zmiňovat vůbec nic, protože základní knihovny Androidu se nemusí uvádět (pokud chceme například použít Google Maps, je to již nutné).

2.2.4 Komunikace zařízení s Android

Zařízení s Androidem muže komunikovat mnoha způsoby (Bluetooth, NFC - Near Field Communication, Wi-Fi). Díky tomu, že zařízení umějí používat Wi-Fi, tak lze komunikovat přes libovolný protokol. Pro účely této práce se budu zabývat komunikací zařízení se serverem přes HTTP.

Komunikace Android zařízení se serverem přes HTTP

Zařízení s Androidem samozřejmě mohou komunikovat s vnějším prostředím. Pro komunikaci je možné použít přímo sockety a komunikaci si řídit od začátku nebo je možné pro komunikaci přes HTTP použít přímo balík rozhraní a tříd *org.apache.http*. Komunikace přes sockety je příliš obecná, proto se nyní budu krátce zabývat komunikací pomocí balíku *org.apache.http*. Následující příklady jsou založené na příkladech ze serveru *w3cmentor.com*. Odkazy:

- http://w3mentor.com/learn/java/android-development/ android-http-services/example-of-http-get-request-using-httpclient-in-android/
- http://w3mentor.com/learn/java/android-development/ android-http-services/performa-a-http-post-request-with-the-httpclient-in-android/

Aby jakákoliv komunikace, ať už přes sockety nebo pomocí HTTP fungovala, je nutné nastavit aplikaci správné oprávnění. V tomto případě oprávnění k přístupu na internet. To znamená přidat do *AndroidManifest.xml*:

```
<uses-permission android:name="android.permission.INTERNET" />
```

2.3 Windows Phone 7

.

Windows Phone 7 byl vyvinut společností Microsoft jako nástupce Windows Mobile (bývalých Windows CE). Narozdíl od svého předchůdce je postaven úplně jinak. Zatímco Windows Mobile byl vyvinut mimo jiné pro použití v náročných podmínkách, jako jsou různé továrny a podobně. Windows Phone 7 byl vyvinut s velkým důrazem na uživatelské pohodlí.

Windows Phone 7 používá grafické uživatelské prostředí pojmenované Metro UI (viz [10]). Metro se zaměřuje na jednoduchost pro uživatele. Metro má uživateli poskytovat jednotný přístup pro podobné činnosti. Zajímavý je také systém tzv. HUBů (hub lze do češtiny přeložit jako uzlový bod). Příklad použití HUBu je komunikace s někým dalším, HUB umožňuje mít veškerou komunikaci různými způsoby (e-mail, SMS, Facebook atd.) na jednom místě. Podobně lze získat například komunikaci, která proběhla před delší dobou.

Veškerá data na platformě Windows Phone 7 se synchronizují na cloudu(viz [9]). Uživatel se tedy nemusí obávat ztráty dat (ačkoliv si umím představit, že právě synchronizace může být, zvlášť za podmínek mobilního internetu u nás, problémem, pokud uživatel nemůže přistupovat například přes Wi-Fi).

2.3.1 Vývoj pod platformou Windows Phone 7

Vývoj pro Windows Phone 7 je pod platformou .NET, tedy v jazyce C#. Zařízení používající Windows Phone 7 mají unifikovaný hardware, jsou tedy určeny podmínky, co musí zařízení mít (například CPU ARMv7 Cortex/Scorpion nebo novější, akcelerometr. . .). V současné době existuje pouze jeden standard, v blízké době mají vzniknout dva další - pro low-endové zařízení a pro zařízení splňující HD([21]).

Pro vývoj je k dispozici SDK, které spolupracuje s Microsoft Visual Studiem. Pro vývoj lze využít dva frameworky:

- Silverlight

  Framework původně určený pro vývoj webových aplikací. V kontextu vývoje na mobilních zařízeních se hodí prakticky pro všechny aplikace, které nepoužívají graficky náročné operace (například herní aplikace nebo aplikace, které různě manipulují s obrázky, například fotky - jejich otáčení a jiné transformace).

- XNA

  XNA je platforma určená pro snadný vývoj her. Je však možné ji i využít i pro jiné účely.

Mobilní aplikace nemusí být nutně vyvíjená jen pomocí jednoho z frameworků. Je možné je kombinovat. Například programová nabídka (menu) se dělá mnohem snáze s pod Silverlightem, na druhou stranu na zobrazení a manipulace s 3D objekty je mnohem vhodnější XNA ([21]).

Vyvíjí se obvykle proti emulátoru, který je celkem dobře vybavený, umí emulovat GPS polohu, akcelerometr, dokonce určitým způsobem i fotoaparát.

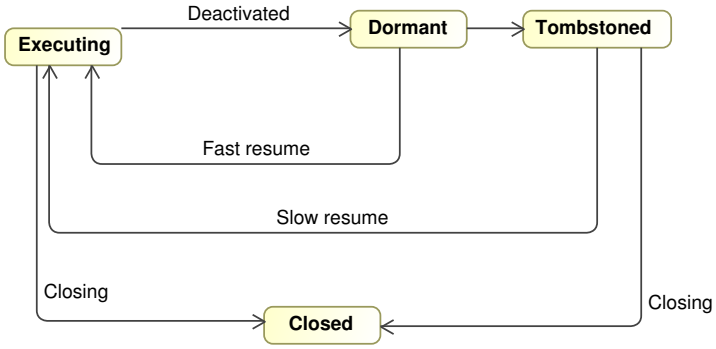
2.3.2 Publikace aplikací

Podobně jako Apple (AppStore) nebo Google (Google Play) má i Microsoft pro svou platformu místo, kde se dá aplikace zveřejnit - Windows Phone Marketplace. Aplikace se publikuje ve formě tzv. XAP souborů. Lze publikovat několika způsoby z pohledu ceny - zadarmo nebo za poplatek (v ČR plně podporováno), může být k dispozici trial verze. Aplikace může být přístupná jen v některých zemích nebo jen některým uživatelům.

Po nahrání aplikace na Marketplace musí být aplikace nejdříve certifikována z technického hlediska (jestli je bezpečná, jestli dodržuje základní ideály systému) a z hlediska obsahu (neobsahuje-li závadné materiály...). Certifikace aplikace (SLA - Service-level Agreement) trvá 5 dnů ([21]).

Aplikace je podobně jako u Androidu rozdělená na různé obrazovky. Lze na ně pohlížet jako na webové stránky, mají svoji URL. Je možné mezi nimi

navigovat (jsou tu událostí jako OnNavigateTo nebo OnNavigateFrom). Systém Windows Phone 7 nemá přímo multitasking (z uživatelského hlediska ano, ale z hlediska vývojáře ne). Pro ilustraci bych rád uvedl stavový diagram aplikace na Windows Phone 7 (obrázek 2.4).



Obrázek 2.4: Životní cyklus aplikace na Windows Phone 7 - [21]

3 OSGi

OSGi je platforma (framework) pro modularizaci aplikací a zjednodušeného použití různých služeb. OSGi je určeno pro programovací jazyk Java. Je zaměřeno na spolupráci, nahraditelnost a znovupoužitelnost jednotlivých částí aplikací. OSGi samo o sobě je pouze specifikace, jakýsi nástin toho, jak má být framework implementován (ačkoliv zde je nutné podotknout, že některé části specifikace OSGi jsou popsány velmi podrobně). Každá z implementací musí plně dodržovat stanovené API.

Existuje několik různých implementací OSGi, asi nejznámější jsou dva: Apache Felix a Equinox (na Equinoxu běží například známý programovací nástroj Eclipse). Pod OSGi jsou aplikace rozdělené do tzv. bundlů. Tyto bundly mohou být (i vzdáleně) instalovány, spouštěny, aktualizovány nebo odinstalovávány bez potřeby restartu celého frameworku.

Pro účely této práce jsem se rozhodl používat Apache Felix. Důvod je velmi snadný. Zatímco pro serverovou část aplikace by se dal použít Equinox, na mobilním zařízení zatím nevím, že by bylo možné Equinox spustit.

3.1 Bundle

Bundle je základní stavební jednotkou každé OSGi aplikace. Jako takový lze definovat bundle jako skupinu Java tříd, zpravidla zařazených do několika balíků (package) a různých zdrojů (ikony, obrázky...). Každý bundle je zabalen do .jar archivu, který ještě musí obsahovat tzv. manifest. Manifest je uložen v kořenovém adresáři v archivu jako soubor *MANIFEST.MF*, popisuje vlastnosti bundlu jako je název, symbolické jméno nebo může obsahovat poskytované balíky a služby. Účelem této práce není detailně rozebrat, co všechno může manifest obsahovat. Zde je ukázka jak může vypadat manifest:

```
Bundle-Name: Hello World
Bundle-SymbolicName: cz.zcu.kiv.zaruba
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
```

```
Bundle-Version: 1.0.0
Bundle-Activator: cz.zcu.kiv.zaruba.Activator
Export-Package: cz.zcu.kiv.zaruba.libraries;version="1.1.1"
Import-Package: cz.zcu.kiv.zaruba.runtime;version="2.2.2"
```

Bundly mohou vyžadovat pro svůj chod různé balíky (package), pak jsou uvedené v manifestu jako Import-Package. Nebo naopak mohou poskytovat nějaké svoje balíky pro ostatní bundly. Tím logicky může vzniknout i poměrně složitý graf závislostí, který ovšem dvě nejpoužívanější implementace tedy Apache Felix a Equinox nijak neřeší.

Všechny bundly (kromě těch, které fungují jako ryze knihovní) mají svůj Activator, což je třída, která implementuje rozhraní BundleActivator a slouží k nastartování bundlu ve frameworku. Většinou obsahuje inicializaci bundlu, nastavení jeho proměnných, vytvoření tříd, navázání služeb, případně zaregistrování svých služeb do frameworku pro jiné bundly. V Activatoru je také metoda, která umožňuje provedení operací při skončení života bundlu. Tím se dostáváme k životnímu cyklu bundlu, kterým se budu lehce zabývat dále. Zde bych rád ukázal velmi jednoduchý Activator:

```
package cz.zcu.kiv.zaruba;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
        private BundleContext context;
        public void start(BundleContext context)
                                throws Exception {
                // inicializace
                this.context = context;
        }
        public void stop(BundleContext context)
                                throws Exception {
                // inicializace
                this.context = null;
        }
}
```

3.1.1 Životní cyklus bundlu

Při popisu OSGi se nelze nezmínit o životním cyklu bundlu. Bundle může být v různých stavech. Pokud ho nainstaluji do frameworku, pak má stav INSTALLED. Po instalaci mohu chtít bundle spustit, potom přechází do stavu STARTING, ze kterého po skončení metody start přechází do stavu ACTIVE. Podobné to je při zastavování bundlu, kde nejdřív přejde do stavu STOPPING a dále do stavu RESOLVED. Stav RESOLVED je ještě zvláštní tím, že do něj přechází okamžitě bundle, který nemá Activator. RESOLVED je stav, kdy framework (jeho resolver) již zanalyzoval jeho obsah a je připravený ho použít. Poslední stav, který může bundle mít je UNINSTALLED.

3.1.2 Služby v OSGi

Jednotlivé bundly mohou různě využívat nebo poskytovat služby. Je to jeden ze základních principů, jak mohou bundly mezi sebou komunikovat. Frameworku sám (resp. jeho implementace) poskytuje služby. Jako příklad bych uvedl *logging service*, *configuration admin service* apod. Každá služba by měla implementovat nějaké rozhraní. Toto rozhraní je pak vyhledáváno pomocí resolveru frameworku.

Pro ilustraci bych rád uvedl způsob, jak lze služby na bundle navázat a jak lze službu ve frameworku poskytovat. Získat službu z frameworku lze několika způsoby, já bych tu rád zmínil dva základní.

První z nich přímé získání reference služby[7]:

```
ServiceReference[] refs = m_context.getServiceReferences(
DictionaryService.class.getName(),"(Language=*)");
Object service = m_context.getService(refs[i]);
```

Ve výše uvedeném kódu je naznačeno, že hledám službu, která poskytuje rozhraní DictonaryService a navíc je zaregistrována s vlastností *Language= jakýkoliv řetězec*. K tomuto způsobu je ale nutné dodat, že je celkem hrubý a ne moc spolehlivý. Kamenem úrazu je, že pokud není služba k dispozici ve chvíli, kdy ji poptávám, tak ji nedostanu, což většinou není příliš žádoucí.

Druhý způsob, který jsem chtěl zmínit, je použití tzv. ServiceTrackeru. ServiceTracker je mechanizmus, kterým dáme najevo frameworku, že máme

zájem o danou službu. Jakmile je k dispozici spustí se metoda, ve které můžeme na tuto událost reagovat. Většinou nám stačí si pouze uložit referenci na službu.

3.2 OSGi Enterprise

Pojem OSGi Enterprise je v několika ohledech těžko uchopitelný. Takovou aplikaci, která je založená na principech OSGi a poskytuje určité „obchodní“ možnosti, lze považovat za OSGi Enterprise. Ovšem pro potřeby této práce bych se radši přiklonil k definici zmíněné Cumminsem a Wardem v [6]. Tedy, že Enterprise OSGi aplikace je aplikace založená na OSGi, která poskytuje jednu z enterprise služeb, popsanou ve specifikaci OSGi Service Platform Enterprise Specification(viz [14]), která poskytuje „obchodní“ služby a hodnoty.

Standardní specifikace je zaměřena spíše na embedded systémy. Časem začalo být jasné, že OSGi musí být pružnější, a proto vznikla specifikace pro Enterprise. Enterprise specifikace je sadou různých služeb.

- Component Models
  - Declarative Services Specification
  - Blueprint Container Specification
- Distributed Services
  - Remote Services
  - Remote Service Admin Specification
  - SCA Configuration Type
- Web Applications and HTTP Servlets
  - Web Application Specification
  - Http Service Specification
- Event models
  - Event Admin Service Specification

- Management and Configuration services
  - JMX™ Management Model Specification
  - User Admin Service Specification
  - Initial Provisioning Specification
  - Configuration Admin Service Specification
  - Metatype Service Specification
- Naming and Directory services
  - JNDI Services Specification
- Database Access
  - JDBC™ Service Specification
  - JPA Service Specification
- Transaction Support
  - JTA Transaction Services Specification
- Miscellaneous Supporting Services
  - Log Service Specification
  - XML Parser Service Specification
  - Tracker Specification

Z přehledu lze vidět, že služby jsou řazeny do kategorií. Nebudu zde popisovat všechny kategorie. Zmíním jen krátce dvě, které se dotýkaly této práce: *Web application and HTTP Servlets* a *Database Access*.

Web application and HTTP Servlets - poskytuje možnosti psát aplikaci do servletů, které je pak možné registrovat. Dále poskytuje obecně komunikaci pomocí HTTP, HTML, XML.

Database Access - poskytuje možnosti komunikace s databázemi. A to dvěma základními možnostmi: JDBC a JPA.

Implementací OSGi Enterprise je celá řada. Ve své práci jsem zvažoval dvě:

- Eclipse Gemini
- Apache Aries

Pro účely této práce jsem se rozhodl používat projekt Eclipse Gemini ([8]). Důvod byl jednoduchý. Tento projekt poskytuje funkčnost, kterou aplikace potřebovala a zároveň je poměrně jednoduchý na použití (lze použít jen některé jeho části). Při výběru frameworku jsem ještě zvažoval použít projekt Apache Aries ([3]), který se mi ovšem nepodařilo zprovoznit (framework má být nejspíše spouštěn jako celek, což se nepodařilo, nebo se mi nepodařilo zjistit jak extrahovat jen některé jeho části). Protože cílem mé práce nebylo porovnat nebo vyzkoušet více Enterprise frameworků, rozhodl jsem po zprovoznění Eclipse Gemini, že se tím již dále nebudu zabývat.

3.2.1 Využití webové služby frameworku vs. framework embedded na serveru

V rámci OSGi běžet i servletový kontejner. Tím se nabízí dvě možnosti, jak můžeme případnou servletovou OSGi aplikaci koncipovat:

- Servletový kontejner může běžet v rámci frameworku
- Další možností je spustit framework (embedded) v servletovém kontejneru

Samostatný framework na serveru

Framework může být spuštěn samostatně a pak v něm mohou být spuštěny bundly, které zajistí komunikaci. Apache Felix implementuje HTTP Service z definice OSGi balíkem *org.apache.felix.http.jetty*, který zajišťuje implementaci servletového kontejneru.

Instalace Jetty pod Felixem je velmi snadná. Jsou dvě cesty, jak Jetty nainstalovat. Jednou z nich je stažení příslušného jar ze stránek Apache Felix, druhou možností je stažení přímo ve spuštěném frameworku z Bundle repository pomocí příkazu obr (v tomto případě *obr:deploy "Apache Felix HTTP Service Jetty"*). Je důležité, aby výraz v uvozovkách byl přesný.

Pro získání seznamu dostupných bundlů lze použít příkaz *obr:list*. Nastavení repositáře většinou bývá v adresáři *conf/config.properties*, kde na konci souboru najdeme:

```
obr.repository.url=http://felix.apache.org/obr/releases.xml
```

Pokud jsme tedy nainstalovali Jetty (framework zároveň mohl nainstalovat bundly, na kterých je Jetty závislý), je na čase tento bundle (bundly) spustit. Je možné, že při spuštění se operační systém zeptá na oprávnění ke spuštění (testováno pod Windows 7). Standardně je pak Jetty přístupný na adrese *localhost:8080*.

Konfigurace Jetty Konfigurace HTTP Service může být provedena několika způsoby. Pro základní konfiguraci lze využít přímo výše zmíněné properties frameworku (standardně *conf/config.properties*). Zde lze nastavit například port, na kterém Jetty běží:

```
org.osgi.service.http.port=8888
```

Další možností je použití Config Admin Service. Touto možností se budu více zabývat v kapitole o Web Console. Seznam možných parametrů Jetty lze najít na stránkách Apache Felix[1].

Vytvoření bundlu se servletem Pro vytvoření bundlu se servletem je možné využít *maven-bundle-plugin*. Závislosti a přibalené knihovny potom stačí nastavit v pom.xml projektu a plugin zajistí zbytek (vytvoření manifestu, zabalení do jar...). V tomto případě byly potřeba tyto závislosti:

```
<dependency>
<groupId>org.apache.felix</groupId>
<artifactId>org.osgi.core</artifactId>
<version>1.0.0</version>
</dependency>
```

[1] http://felix.apache.org/site/apache-felix-http-service.html#ApacheFelixHTTPService-ConfigurationProperties

```
<dependency>
<groupId>org.apache.felix</groupId>
<artifactId>org.apache.felix.framework</artifactId>
   <version>1.0.0</version>
</dependency>
<dependency>
<groupId>org.apache.felix</groupId>
<artifactId>org.apache.felix.http.bundle</artifactId>
<version>2.0.4</version>
</dependency>
<dependency>
            <groupId>javax.persistence</groupId>
            <artifactId>persistence-api</artifactId>
            <version>1.0</version>
</dependency>
```

A tato konfigurace maven-bundle-plugin:

```
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<extensions>true</extensions>
<configuration>
<instructions>
  <Private-Package>cz.zcu.kiv.zaruba.*</Private-Package>
            <Bundle-Activator>
                cz.zcu.kiv.zaruba.activator.Activator
            </Bundle-Activator>
</instructions>
</configuration>
</plugin>
```

Zde je konfigurace velmi jednoduchá. Pomocí tohoto pluginu, lze nastavit více, například lze nastavit, které package se budou exportovat, které se budou importovat, případně které se vloží do bundlu přímo jako knihovna.

Samotná třída servletu není ničím zajímavá. Je to standardní třída dědící od třídy *HttpServlet*. Příklad:

```
public class HelloWorld extends HttpServlet {
```

```
@Override
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
    throws ServletException, IOException
  {
      resp.setContentType("text/plain");
    resp.getWriter().write("Hello World");
  }
}
```

Pokud máme nastavené *pom.xml* a vytvořený servlet, pak lze přikročit přímo k vytvoření třídy *Activator* třídy servletu. Ve třídě *Activator* zaregistrujeme servlet k HTTP Service frameworku. Příklad:

```
public final class Activator implements BundleActivator {
  private ServiceTracker tracker;
  private HttpServlet servlet = new HelloWorld();

public void start(BundleContext context) throws Exception {
    this.tracker = new ServiceTracker(context,
         ExtHttpService.class.getName(),~null) {
@Override
public Object addingService(ServiceReference ref) {
        Object service = super.addingService(ref);
        try {
        // You can map the servlet to any context. This will
        // actually map it to all requests.
         ((HttpService) service).registerServlet("/hello",
           servlet, null, null);
        } catch (Exception e) {
          e.printStackTrace();
        }
        return service;
      }

      @Override
      public void removedService(ServiceReference ref,
                                 Object service) {
        ((ExtHttpService) service).unregisterServlet(servlet);
        super.removedService(ref, service);
```

```
      }
    };
    this.tracker.open();
  }

  public void stop(BundleContext context) throws Exception {
    this.tracker.close();
  }
}
```

V uvedeném kódu jsme získali pomocí ServiceTrackeru HTTP Service, zaregistrovali a namapovali servlet *HelloWorld* na adresu /hello. V případě, že používáme samostatný Felix, bude standardně dostupný na adrese **http://localhost:8080/ hello**. Pokud používáme svůj servletový kontejner, potom ho najdeme na adrese: **adresa_kontejneru/nazev_aplikace/-hello**. Jestliže používáme svůj servletový kontejner, je tu ještě jeden rozdíl. Je potřeba pomocí *ServiceTrackeru* hledat ne *HttpService* ale *ExtHttpService*.

Spuštění frameworku v rámci servletového kontejneru

Framework může být spuštěn v rámci Java kódu v embedded módu. To poskytuje možnosti pro spuštění přímo v rámci již spuštěného servletového kontejneru (jako je například Apache Tomcat, GlassFish nebo Jetty). Instanci Felixu potom lze zprovoznit pouze v rámci jednotlivého servletu, který bude používat služby nainstalovaných bundlů. Nebo můžeme zajistit, aby se požadavky na servletový kontejner přímo přesměrovávaly do frameworku.

První možnost poskytuje pouze omezené možnosti a není příliš zajímavé se jí zabývat, proto se budu zabývat druhou. Aby bylo přesměrování možné, musíme zařídit pár věcí.

Následující příklad je zprovozněn na základě příkladů ze stránek Apache Felix [5] a popisu blogu Chrise Uptona ([16]).

První věcí, kterou musíme zařídit, je vytvoření instance Felixe dříve, než kontejner začne přijímat požadavky. Nejlepší místo pro to je *ServletContextListener*, což je rozhraní, které má pouze dvě metody: *contextInitialized* a *contextDestroyed*. Taková třída může vypadat takto:

```
public class StartupListener implements ServletContextListener {

  private FrameworkService service;

  @Override
  public void contextInitialized(ServletContextEvent event) {
    // framework initialization
    this.service =
             new FrameworkService(event.getServletContext());
    this.service.start();
  }
  @Override
  public void contextDestroyed(ServletContextEvent event) {
    // stop framework
    this.service.stop();
  }
}

import org.apache.felix.framework.Felix;

public class FrameworkService {
  private final ServletContext context;
  private Felix felix;
  public FrameworkService(ServletContext context) {
    this.context = context;
  }
  public void start() {
    try {
      doStart();
    } catch (Exception e) {
      log("Failed to start framework", e);
    }
  }
  public void stop() {
    try {
      doStop();
    } catch (Exception e) {
      log("Error stopping framework", e);
    }
  }
  private void doStart() throws Exception {
```

```
    Felix tmp = new Felix(createConfig());
    tmp.start();
    this.felix = tmp;
     log("OSGi framework started", null);
  }
   private void doStop() throws Exception {
    if (this.felix != null) {
      this.felix.stop();
    }
    log("OSGi framework stopped", null);
  }
  private Map<String, Object> createConfig() throws Exception {
    Properties props = new Properties();
    // load config from properties file
    props.load(this.context
        .getResourceAsStream("/WEB-INF/framework.properties"));

    HashMap<String, Object> map = new HashMap<String, Object>();
    for (Object key : props.keySet()) {
      map.put(key.toString(), props.get(key));
    }
     map.put(FelixConstants.SYSTEMBUNDLE_ACTIVATORS_PROP,
        Arrays.asList(new ProvisionActivator(this.context)));
    return map;
  }
  private void log(String message, Throwable cause) {
    this.context.log(message, cause);
  }
}
```

Když napíšeme takovou třídu, je nutné dát o ní vědět kontejneru. Což není příliš složité, stačí připsat do *web.xml* následující:

```
<listener>
   <listener-class>
     cz.zcu.kiv.zaruba.StartupListener
   </listener-class>
</listener>
```

Když máme tedy zajištěno, že instance frameworku bude včas vytvořena

a spuštěna, musí se zařídit, aby byly požadavky přesměrovány do něj. V rámci spuštění Felixe byla zmíněna instalace bundlu *org.apache.felix.http.bridge* do frameworku. Instalace bundle do frameworku se provádí přes *BundleContext* získaný z instance frameworku:

```
BundleContext context = felix.getBundleContext();
```

Instalace potom probíhá manuálně:

```
felix.getBundleContext().installBundle("cesta k bundlu");
```

Nebo lze například nainstalovat postupně všechny bundly v adresáři:

```
ArrayList<String> stringList = new ArrayList<String>();
for (Object o :
      this.context.getResourcePaths("/WEB-INF/MyBundles/")) {
  String name = (String) o;
  if (name.endsWith(".jar")) {
    URL url = this.context.getResource(name);
    if (url != null) {
    Bundle bundle = felix.getBundleContext().installBundle(
        url.toExternalForm());
    bundle.start();  // start bundle
    }
  }
}
```

K tomu je ještě zapotřebí zajistit pár závislostí buď přidáním těchto .jar souborů do **WEB-INF/lib** (nebo odpovídajících závislostí v rámci Maven projektu):

- felix.jar
- org.apache.felix.http.api-x.x.x.jar
- org.apache.felix.http.base-x.x.x.jar
- org.apache.felix.http.proxy-x.x.x.jar

- osgi_R4_compendium-x.x.jar

Přesměrování požadavků se potom zajistí další úpravou souboru *web.xml*:

```
<listener>
  <listener-class>
    org.apache.felix.http.proxy.ProxyListener
  </listener-class>
</listener>
<servlet>
  <servlet-name>proxy</servlet-name>
  <servlet-class>
    org.apache.felix.http.proxy.ProxyServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>proxy</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

3.3 Felix na OS Android

Implementaci OSGI Apache Felix je možné spustit embeddovanou v prakticky jakékoliv java aplikaci. Ani aplikace pro android není výjimkou.

3.3.1 Spuštění Felixe na Androidu

Základní stavební jednotkou pro Android jsou *Activity*. Základní strukturou jednotlivé *Activity* jsou dvě metody:

- onCreate – metoda volaná při vytvoření *Activity*. Je jí možné využít k inicializaci proměnných a v našem případě k vytvoření instance *Felixu*.

- onStart – metoda volaná při startu aplikace. V našem případě jsem ji využil k nastartování *Felixe* a k instalaci bundlů do frameworku.

Před samotným spuštěním frameworku je nutné nejdříve vytvořit konfiguraci. Konfigurace je mapa reprezentována instancí třídy *StringMap*. Před spuštěním frameworku je nutné do konfigurace přidat několik věcí:

- *FelixConstants.LOG_LEVEL_PROP* – **volitelné**
  - nastavení logování ve frameworku
  - například String.valueOf(Logger.LOG_DEBUG)
- *FelixConstants.FRAMEWORK_SYSTEMPACKAGES* – **nutné**
  - seznam balíků, které budeme potřebovat uvnitř frameworku
- *Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA* – **nutné**
  - musí být nastaven na "android.content"
  - balíček, který budeme využívat pro získání aplikačního kontextu androidu
- *Constants.FRAMEWORK_STORAGE* – **nutné**
  - umístění cache pro framework

Cache pro framework

Ke svému fungování potřebuje framework mít přístup k adresáři, kde si bude moct ukládat bundly a informace o frameworku. Cache je možné mít jak ve vnitřní paměti zařízení, tak na SD kartě. Důvody pro umístění cache na SD kartu:

- u některých zařízení nepříliš velká vnitřní paměť
- omezený přístup do vnitřní paměti

Vzhledem k tomu je lepší umístit cache frameworku na SD kartě. Aby měla aplikace na Androidu přístup k SD kartě, je nutné v *AndroidManifest.xml* přidat aplikaci právo k přístup na SD kartu.

```
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE">
```

Zde uvedu kód pro inicializaci cache:

```
String baseDir = Environment.getExternalStorageDirectory()
              .getAbsolutePath();
try {
  // overeni jestli cache existuje
  File felixCache = new File(baseDir + File.separator
         + FELIX_DIR + File.separator + "felix_cache");
  // pokud adresar cache neexistuje pak ho vytvorime
  if(!felixCache.exists()) felixCache.mkdir();
  // vytvoreni docasneho souboru pro cache
  m_cache = File.createTempFile("felix-cache", null,felixCache);
} catch (IOException ex) {
  throw new IllegalStateException(ex);
}
  // prevzato z luminis - nastaveni umísteni cache
  m_configMap.put(Constants.FRAMEWORK_STORAGE,
                  m_cache.getAbsolutePath());
  // konec luminis
  m_cache.delete(); // vymazani cache
  m_cache.mkdirs(); // vytvorení nove cache
```

Když máme připravenou cache, už nám nic nebrání ve spuštění samotného frameworku. Samotné spuštění frameworku je velmi jednoduché:

```
try {
  m_felix = new Felix(m_configMap);
  m_felix.start();
} catch (BundleException e) {
  e.printStackTrace();
}
```

Chování ihned při spuštění frameworku je možné ovlivnit pomocí seznamu Activatorů (*ArrayList* instancí třídy *BundleActivator*), který se uloží do konfigurace jak **FelixConstants.SYSTEMBUNDLE_ACTIVATORS_PROP**, nebo přímo přístupem do frameworku.

3.3.2 Přístup do spuštěného frameworku

Pokud tedy máme spuštěnou instanci frameworku, dostaneme se k *bundleContextu*. Díky tomu můžeme spravovat framework (instalovat bundly, zastavovat bundly, získávat a registrovat služby). Ze správy frameworku bych rád zmínil dvě možnosti, které jsou důležité pro mou aplikaci pro Android.

- Registrace služby
- Využití služby frameworku

Přístup ke službám frameworku

Obvykle máme ve frameworku nainstalováno několik bundlů, které jsou spojené službami a importy. Díky přístupu k *bundleContextu* je možné získat instanci služby ve frameworku a tudíž k celé službě. Příklad kódu:

```
ServiceReference sRefAct =
       m_felix.getBundleContext()
         .getServiceReference(Activity.class.getName());
Activity act = m_felix.getBundleContext().getService(sRefAct);
```

Instance z vnějšku jako služba frameworku

Na podobném principu funguje dodání instance, např. activity, ze kterého je Felix spuštěný, aby byla přístupná ve frameworku:

```
m_felix.getBundleContext()
       .registerService(Activity.class.getName(), this, null);
```

3.3.3 Využití služeb Android z OSGi

Vytvoření bundlu, který využívá služby, které si zaregistrujeme z aplikace v Androidu, je v podstatě stejné jako vytváření jakéhokoliv jiného bundlu.

Vytvoříme bundle, který z frameworku získá například Android *bundle-Context* nebo instanci *Activity*. Abychom toho mohli dosáhnout, je nutné vložit do našeho frameworku něco, co poskytne třídy Androidu. Lze to buď speciálním bundlem, který v sobě bude mít buď *android*.jar* nebo přímo rozbalené třídy z téhož archivu. Nebo tuto knihovnu vložíme do našeho bundlu nebo lze balíky uvést do *FelixConstants.FRAMEWORK_SYSTEM-PACKAGES*.

Na vložení závislostí lze opět využít už výše zmíněný *maven-bundle-plugin* do konfigurace vložíme:

```
<Embed-Dependency>
  *;scope=compile|runtime;inline=false
</Embed-Dependency>
```

Tím zajistíme, že veškeré závislosti potřebné ke kompilaci a k běhu (samozřejmě z *pom.xml*) budou vloženy do archivu bundlu.

Bohužel to ale nefunguje úplně bez chyby. Protože jakmile je aktivní tato volba, tak plugin nereaguje na nastavení import a export package (je možné, že jsem přehlédl nějakou možnost konfigurace). Tyto problémy se pak dají řešit pouze ruční úpravou manifestu bundlu.

Jako příklad bundlu, který využívá služby z androidu, uvedu bundle, který po startu nastaví uživatelské rozhraní (samozřejmě pouze velmi jednoduché, v *TextView* vypíše Ḧello World").

```
public class Activator implements BundleActivator {
  private ServiceTracker trackerAndroid;
  private ServiceTracker trackerActivity;
  private Context androidContext;
  private Activity activity;

 public void start(BundleContext context) throws Exception {
   this.trackerAndroid = new ServiceTracker(context,
    Context.class.getName(),~null) {
       @Override
       public Object addingService(ServiceReference ref) {
         Object service = super.addingService(ref);
         // here we can do some inicialization
```

```
        return service;
      }
      @Override
      public void removedService(ServiceReference ref,
                                          Object service) {
        super.removedService(ref, service);
      }
    };
    this.trackerAndroid.open();
    this.trackerActivity = new ServiceTracker(context,
      Activity.class.getName(), null) {
      @Override
      public Object addingService(ServiceReference ref) {
        Object service = super.addingService(ref);
        Activity activity = (Activity)service;
        // here we can do some inicialization
        return service;
      }
      @Override
      public void removedService(ServiceReference ref,
                                        Object service) {
        super.removedService(ref, service);
      }
    };
    this.trackerActivity.open();
    this.activity = (Activity) trackerActivity.getService();
    this.androidContext = (Context) trackerAndroid.getService();
    android.widget.TextView tv = new TextView(androidContext);
    tv.setText("Hello, Android Bundle");
    activity.setContentView(tv);
}
 public void stop(BundleContext arg0) throws Exception {
 }
}
```

3.3.4 Příprava bundlu pro použití na Android

Aplikace na Androidu běží na Dalvik VM. Dalvik VM není stejný jako standardní Java VM. Pro tuto práci nejsou rozdíly moc důležité. Důležité je, že

musíme výsledný bundle (.jar soubor) upravit před tím, než ho můžeme na Android použít.

Android SDK poskytuje naštěstí nástroje na tuto úpravu (postup je převzat z [4]). Postup je následující:

1. Každý .jar souboru musí obsahovat svůj DEX ekvivalent. Toho můžeme dosáhnout pomocí nástroje dx.

   ```
   dx --dex --output=classes.dex JAR_file.jar
   ```

2. Potom co jsme vytvořili soubor classes.dex ho musíme přidat do .jar souboru.

   ```
   aapt add JAR_file.jar classes.dex
   ```

3. Posledním krokem je nyní umístit bundle na místo, kde bude přístupný ze systému Android (vnitřní paměť, paměťová karta). Potom je bundle připraven pro použití na Android.

4 Persistence dat

Téměř každá aplikace musí někam ukládat data. V zásadě se zde nabízí dvě možnosti ukládání: do souboru a ukládání do databáze. Ukládání do souborů nebudu řešit, protože jsou pro účely této práce nevhodné, práce s databází nabízí pohodlnější přístup.

Co se týče ukládání do databáze nabízí Java dvě možnosti:

- JDBC
- Objektově relační mapování (ORM)

4.1 JDBC

Do databáze je možné v Javě přistupovat pomocí JDBC (The Java Database Connectivity). JDBC je průmyslový standard pro databáze (čerpáno z [13]), který poskytuje programátorovi spojení a možnosti manipulace se širokou škálovou databází ovládaných pomocí jazyka SQL (Structured Query Language - jazyk používaný pro práci s databázemi).

Při práci s JDBC se nejdříve vytvoří spojení s databází, které se realizuje pomocí třídy Connection. K vytvoření spojení s databází je potřeba mít k dispozici ovladač pro danou databázi. Způsoby instalace ovladače jsou sice různé, ale většinou stačí dát cestu k ovladači do classpath.

Přes toto spojení je potom možné s databází manipulovat SQL dotazy. Tyto dotazy lze spouštět několika různými metodami. Pro zavedení dotazu je potřeba nejdříve vytvořit instanci třídy Statement, popřípadě PreparedStatement. Tyto instance se vytvářejí pomocí třídy spojení Connection. Pokud je tedy k dispozici instance jedné z tříd, je zde několik možností, jak provést dotaz. Záleží na tom, jestli chceme provést výběrový dotaz nebo manipulační dotaz.

Možnosti spuštění SQL dotazu:

- execute

Tato metoda vykoná libovolný SQL dotaz, a to jak výběrový dotaz, tak i manipulační dotaz. V případě manipulačního dotazu samozřejmě nedostaneme z databáze žádná data. Tato konkrétní metoda vrací boolean podle toho, jestli je ResultSet roven null (pokud je null pak vrací false).

- executeQuery

  Metoda slouží pouze k vykonávání výběrového dotazu. Výsledky dotazu se vrací ve třídě ResultSet.

- executeUpdate

  Metoda sloužící k vykonávání manipulačních dotazů. Návratová hodnota této funkce je počet řádků, které byly dotazem ovlivněny.

Pokud byl vykonán výběrový dotaz, jsou data vrácena ve třídě ResultSet. Třída ResultSet poskytuje základní metody pro práci s výsledky. Výsledky je nutné číst sekvenčně po jednotlivých řádcích (čehož lze například dosáhnout použitím while cyklu). Každý řádek se načítá pomocí metody next(). Data z každého řádku lze získat sadou metod jako je getString(), getInt()...

4.2 Objektově relační mapování - použití JPA

JPA (Java Persistence API) je speciálním případem ORM. Principem tohoto přístupu je namapování entit (tabulek) z databáze na Java třídy. Programátor potom pracuje s entitami jako s objekty tříd. JPA je Java framework pro správu relačních dat, který umožňuje objektově relační mapování. Lze ho využít jak v Java SE tak i v Java EE. Specifikace JPA 1.0 bylo uveřejněna v květnu roku 2006, nynější verze JPA 2.0 byla uveřejněna na konci roku 2009.

Základem použití JPA je mít nějakou třídu, která má být uložena v databázi. K tomu, aby framework věděl, že má být objekt uložen (v angličtině persist odtud persistence), je možné použít dva základní způsoby, a to buď využití souboru orm.xml, ve kterém specifikujeme, jak má persistence vypadat nebo lze přímo využít anotace v kódu.

Specifikace JPA doporučuje používat spíše anotace než mapovací XML soubor. Ačkoliv je zřejmé, že i mapování pomocí XML má své výhody. Obecně lze těžko rozsoudit, který přístup je lepší. Obecně se lze shodnout

na tom, že ani jedno řešení není stříbrná kulka. Oba přístupy mají své opodstatnění a využití. Pro anotace například mluví to, že je dobrým zvykem mít informace o třídě přímo u zdrojového kódu. Na druhou stranu pokud chceme používat pro stejné třídy různé mapování, pak je mapovací soubor jasnou volbou. Nakonec lze konstatovat, že oba přístupy lze kombinovat a obecné řešení neexistuje.

Pro JPA je důležitý další soubor, který je povinný pro oba přístupy, a tím je *persistence.xml*. V tomto souboru definujeme přístup do databáze, případně další vlastnosti a chování ukládání a načítání dat. Pro ilustraci uvádím ukázku tohoto souboru:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name="GPSGame"
         transaction-type="RESOURCE_LOCAL">
    <provider>
      org.eclipse.persistence.jpa.PersistenceProvider
    </provider>

    <class>cz.zcu.kiv.zarubaDiplomka.server.model.Game</class>
    <class>
      cz.zcu.kiv.zarubaDiplomka.server.model.Player
    </class>
    <class>
      cz.zcu.kiv.zarubaDiplomka.server.model.GameType
    </class>
    <class>
      cz.zcu.kiv.zarubaDiplomka.server.model.Position
    </class>
    <class>
      cz.zcu.kiv.zarubaDiplomka.server.model.Scoreboard
    </class>
    <class>
      cz.zcu.kiv.zarubaDiplomka.server.model.PositionLog
    </class>
```

```
    <class>
      cz.zcu.kiv.zarubaDiplomka.server.model.AlertLog
    </class>

    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby://localhost:1527/GPSGameDB;
        create=true" />
      <property name="javax.persistence.jdbc.user"
        value="app" />
      <property name="javax.persistence.jdbc.password"
        value="app" />
      <property name="javax.persistence.jdbc.driver"
              value="org.apache.derby.jdbc.ClientDriver" />

    </properties>
  </persistence-unit>
</persistence>
```

V souboru například vidíme, že se chceme připojit do databáze derby na portu 1527.

Dále je dobré si ukázat, jak může vypadat anotace třídy:

```
@Entity
public class Game implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  @SequenceGenerator(name = "GAME_ID_GENERATOR",
                             sequenceName = "GAME_ID")
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
                             generator = "GAME_ID_GENERATOR")
  private int id;

  @Column(name = "GAME_STARTED")
  private Timestamp gameStarted;
```

```
  private String name;

  @Column(name = "SHOOT_RANGE")
  private int shootRange;

  @Column(name = "DUEL_RANGE")
  private int duelRange;

  @Column(name = "GAMELENGTH")
  private int gamelength;

  @Column(name = "REFRESH_RATE")
  private int refreshRate;

  // bi-directional many-to-one association to GameType
  @ManyToOne
  @JoinColumn(name = "GAME_TYPE")
  private GameType gameTypeBean;

  // bi-directional many-to-one association to Player
  @OneToMany(mappedBy = "gameBean")
  private Set<Player> players;

  // bi-directional many-to-one association to Position
  @OneToMany(mappedBy = "game", cascade = CascadeType.REMOVE)
  private Set<Position> positions;

  // bi-directional many-to-one association to Position
  @OneToMany(mappedBy = "game", cascade = CascadeType.REMOVE)
  private Set<Scoreboard> scoreboards;
}
```

K tomu, aby naše třída mohla být uložena do paměti, jsou důležité dvě věci: anotace *@Entity* u třídy a pak také uvedení třídy v *persistence.xml*, jak lze vidět v ukázce. Dále je nutné, aby byla třída jako bean, tzn. měla implicitní konstruktor, gettery a settery pro každou datovou položku.

Implementací JPA je samozřejmě několik. ORM se používá již nějakou dobu a JPA se dostává na vrchol až před nedávnou dobou (do té doby se používal hlavně Hibernate). Dnes je referenční implementací JPA

EclipseLink. Samozřejmě jsou zde i konkurenti přímo na JPA jako je OpenJPA, tak i dále běží Hibernate, který má stále velkou komunitu uživatelů.

5 Aplikace GPS game

Úkolem práce bylo vytvořit aplikaci, která ukáže použití komponentového frameworku OSGi na zařízení na platformě Android. Byla zvolena herní aplikace pro více hráčů s použitím GPS. Aplikace je rozdělená na 2 části. Jedna část je serverová (zde je také použito OSGi). Druhá část (klientská) je pro Android.

5.1 Záměr aplikace

Cílem moderních her již není pouze zabavit hráče. Stále více se v herním prostředí prosazují socializační a interaktivní aspekty spojené s rozvojem online hraní. Tato aplikace zprostředkovává hru na honěnou za použití GPS. Tedy předpokládá se, že bude hraná na větším nekrytém prostranství, kvůli přesnosti GPS sensorů (které jsou v řádech 10 metrů). Aplikace je určena jak pro děti, tak i pro dospělé. V podstatě pro každého kdo vlastní chytrý telefon[15].

Hra může být nastavena i tak, že bude sloužit jako pomůcka při hraní jiných her, jako je například paintball (pak může hra sloužit jen jako orientační pomůcka). Důležitým úkolem aplikace je ale dostat lidi ven, v době, kdy se příliš mnoho času tráví za počítačem v kanceláři. Hra je založená na fyzické aktivitě a není možné ji provozovat doma z obýváku.

5.1.1 Co hra dělá

Začneme od začátku. Hráč se přihlásí do aplikace pod svou přezdívkou. Poté buď vytvoří hru nebo se připojí k již vytvořené hře. Hra má několik základních parametrů:

- Název
- Vzdálenost pro střelbu
- Vzdálenost pro duel

- Délka hry
- Obnovovací doba

Nastavením parametrů říkáme, jakou hru chceme hrát, pokud například chceme hrát hru bez střílení a duelů, například pokud chceme hru použít jako orientační pomůcku na paintball, nastavím vzdálenost pro duel i pro střelbu na 0.

Pokud jsme hru vytvořili nebo se k ní přidali objeví se nám herní obrazovka. Ta nám ukazuje křížkem, kde jsme a vodorovnou či svislou čarou nám ukazuje jednu ze souřadnic protivníka. Hráč se poté pohybuje v herním prostoru, kde vyhledává protivníka. Pokud ho uvidí je možné ho zastřelit „na dálku“, tzn. nedochází k duelu. Pokud se protivníci přiblíží na kratší vzdálenost, než je nastavena, nastává duel. Duel spočívá v tom, že hráči se zobrazí hláška, že byl započat duel. Musí co nejrychleji reagovat, protože ten z nich, který zareaguje rychleji, vítězí a přežije. Každý z hráčů má pouze jeden život.

5.2 Architektura aplikace

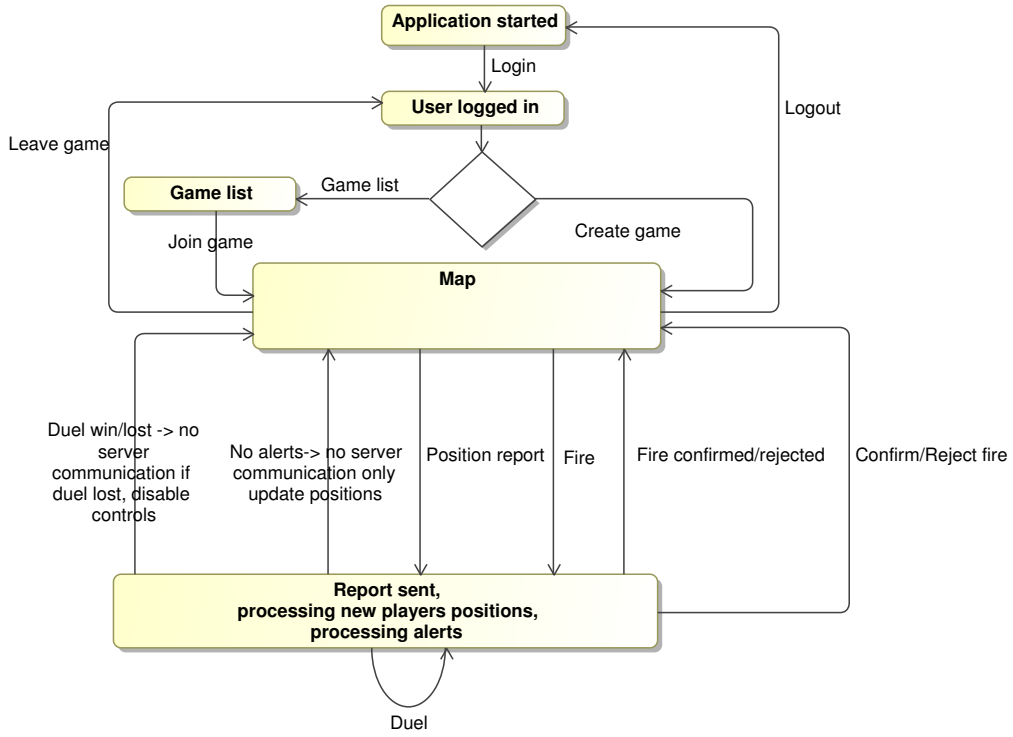
Aplikace je rozdělena na dvě nezávislé části. Serverová část je servletová aplikace, takže běží na nějakém servletovém kontejneru (v mém případě jsem používal Apache Tomcat 6.0.35). Klientská část běží na Androidu. Obě části spolu komunikují pomocí HTTP protokolu. Klientská aplikace posílá požadavky pomocí HTTP GET a serverová část vrací data ve formát XML. Abych dokončil výčet použitých technologií, tak na straně serveru se používá JPA - EclipseLink jako persistenční vrstva a databáze Apache Derby.

5.3 Popis komunikačního protokolu

Klient a server spolu komunikují pomocí HTTP protokolu.

Klient posílá požadavky serveru pomocí HTTP GET, například:

```
http://localhost:8080/GameServerApplication/game
```



Obrázek 5.1: Diagram přechodů v závislosti na komunikačním protokolu

```
?action=report&messageType=positionReport
&playerId=305&longitude=11.11&latitude=22.22
```

Server zasílá odpovědi jako XML, například:

```
<message>
  <alerts>
    <alert action="Fire" game="3001" sourcePlayerId="305"
              type="Fire"/>
  </alerts>
  <positions coordinates="longitude">
    <position longitude="11.11" playerId="205"/>
  </positions>
</message>
```

5.3.1 Login

Při spuštění aplikace nebo po odhlášení je uživatel vyzván k zadání svého uživatelského jména. Po jeho zadání se posílá na server požadavek s parametrem *action* nastaveným na login. Jako odpověď dostane od serveru XML s id hráče nebo chybovou hlášku, pokud není zadaný login dostupný.

Povinné parametry požadavku na server

- action - v tomto případě login
- name - uživatelské jméno

Ukázka:

```
serverURL?action=login&name=chicko
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

  ```
  <message type="login">
    <player id="id_uživatele" name="uživatelské_jméno"/>
  </message>
  ```

- pokud nastane chyba server vrací XML v tomto tvaru

  ```
  <message type="error">
    <errorMessage>
      Chybové hlášení (například "Login is not available")
    </errorMessage>
  </message>
  ```

5.3.2 Game list

Po přihlášení má hráč možnost přidat se k nějaké započaté hře. K tomu potřebuje seznam her s jejich *id* (využívá se v *joinGame* viz dále). Příkaz *gameList* dostane ze serveru seznam dostupných her s jejich parametry včetně *id*.

Povinné parametry požadavku na server

Jediným povinným parametrem je parametr *action*.

- action - gameList

Ukázka:

```
serverURL?action=gameList
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

```
· <message type="gamelist">
    <games>
      <game duelRange="vzdalenost_pro_duel"
            gameStarted="čas_startu_hry"
            gameType="typ_hry"
            gamelength="delka_hry_v_minutach"
            id="id_hry"
            name="jmeno_hry"
            shootRange="vzdalenost_pro_strelbu_na_dalku"/>
    </games>
  </message>
```

- pokud nastane chyba server vrací XML v tomto tvaru

```
· <message type="error">
    <errorMessage>
```

```
      Chybové hlášení (například "Server not responding")
    </errorMessage>
  </message>
```

5.3.3 Join game

Jakmile má hráč k dispozici seznam her, může se k nějaké z nich připojit. To obstarává požadavek typu *joinGame*. Jeho parametry jsou jen *id* hry a *id* hráče. Odpověď serveru, pokud vše proběhlo v pořádku je zopakování id hry a četnost zasílání reportů.

Povinné parametry požadavku na server

Jediným povinným parametrem je parametr action.

- action - joinGame
- playerId - id hráče, který se chce připojit
- gameId - id hry, ke které se chce hráč připojit

Ukázka:

```
serverURL?action=joinGame&playerId=205&gameId=1851
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

  ```
  <message type="joinGame">
    <game id="id_hry" refreshRate="2"/>
  </message>
  ```

- pokud nastane chyba server vrací XML v tomto tvaru

```
· <message type="error">
    <errorMessage>
      Chybové hlášení (například "Game does not exists")
    </errorMessage>
  </message>
```

5.3.4 Create game

Hráč si nemusí nutně vybrat pouze hru, ke které se chce přidat, může chtít novou hru založit. K tomu slouží požadavek na server typu createGame. Na server se musí odeslat všechny parametry vytvářené hry, takže požadavek je trochu "delší".

Povinné parametry požadavku na server

Jediným povinným parametrem je parametr action.

- action - createGame
- playerId - id hráče, který se chce hru vytvořit
- name - název hry
- gameType - typ hry, reprezentován číslem (aplikace je připravena na více typů her, ale implementován je reálně jen jeden a proto volba zatím nemá smysl)
- shootRange - vzdálenost v metrech, na kterou je možné zastřelit protivníka
- duelRange - vzdálenost v metrech, ve které je vyvolán duel
- gamelength - čas v minutách, po kterou má trvat hra
- refreshRate - čas v sekundách, za který se bude pravidelně posílat report serveru

Ukázka:

```
serverURL/game?action=createGame&playerId=205
               &name=nova hra&gameType=1
               &shootRange=200&duelRange=100
               &gamelength=222&refreshRate=2
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

  ```
  <message type="createGame">
    <game id="id_hry" refreshRate="2"/>
  </message>
  ```

- pokud nastane chyba server vrací XML v tomto tvaru

  ```
  <message type="error">
    <errorMessage>
      Chybové hlášení (například "You are not logged or
      you were too long inactive")
      </errorMessage>
  </message>
  ```

5.3.5 Position report

Po tom, co se hráč připojí ke hře, ať už ji vytvoří nebo ne, je nutné, aby posílal informace o své poloze (GPS souřadnice). Současně při hlášení těchto souřadnic dostává řídící zprávy v podobě Alertů. Ty mohou být více typů a podrobně se o nich zmiňuji jinde.

Povinné parametry požadavku na server

- action - v tomto případě report
- messageType - positionReport

- playerId - id hráče, který se chce hru vytvořit
- longitude - údaj o zeměpisné délce
- latitude - údaj o zeměpisné šířce

Ukázka:

```
serverURL/game?action=report&messageType=positionReport
   &playerId=305&longitude=11.11&latitude=22.22
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

```
<message>
  <alerts>
    <alert action="Fire" game="6651"
        sourcePlayerId="205" type="Fire"/>
    <alert action="Time over" game="6651"
        type="Time over"/>
  </alerts>
  <positions coordinates="latitude">
    <position latitude="22.22" playerId="205"/>
  </positions>
</message>
```

- pokud nastane chyba server vrací XML v tomto tvaru

```
<message type="error">
    <errorMessage>
      You are not logged or you were too long inactive
    </errorMessage>
</message>
```

5.3.6 Fire

Pokud hráč vidí jiného hráče, může na něj vystřelit. Serveru to dá najevo, modifikaci zprávy *Position report*.

Povinné parametry požadavku na server

- action - report
- messageType - fire
- playerId - id hráče, který střílí
- enemyId - id hráče, na kterého se střílí
- longitude - pozice hráče
- latitude - pozice hráče

Ukázka:

```
serverURL/game?action=report&messageType=fire
 &playerId=205&enemyId=305
 &longitude=11.11&latitude=22.22
```

Formát XML ze serveru

Server vrací zprávu ve stejném formátu jako u *Position report*.

5.3.7 Duel

Pokud se hráči k sobě přiblíží blíže než je specifikováno v *duel range*, pak hráčům přijde *Alert*, že se mají účastnit duelu.

Povinné parametry požadavku na server

- action - duel
- playerId - id hráče
- enemyId - id nepřítele
- reaction - reakční doba na duel v ms

Ukázka:

```
serverURL/game?action=duel&playerId=205&enemyId=305
 &reaction=2100
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

  ```
  <message type="duelParticipation">
    <player1 id="205"/>
    <player2 id="305"/>
  </message>
  ```

- pokud nastane chyba server vrací XML v tomto tvaru

  ```
  <message type="error">
    <errorMessage>
      Duel does not exist
    </errorMessage>
  </message>
  ```

5.3.8 Get players

Kromě pravidelných zpráv o poloze se hráč ještě musí pravidelně ptát, kteří hráči jsou ve hře.

Povinné parametry požadavku na server

- action - getPlayers
- playerId - id hráče
- gameId - id hry

Ukázka:

```
serverURL/game?action=getPlayers&gameId=6901&playerId=305
```

Formát XML ze serveru

- pokud vše proběhne jak má, pak server vrátí XML v tomto tvaru

  ```
  <message>
    <players>
      <player id="305" name="chicker"/>
    </players>
  </message>
  ```

- pokud nastane chyba server vrací XML v tomto tvaru

  ```
  <message type="error">
    <errorMessage>
      You are not logged or you were too long inactive
    </errorMessage>
  </message>
  ```

5.4 Serverová aplikace

Serverová aplikace běží v rámci serveru Apache Tomcat. Byl tedy zvolen přístup, kdy je OSGi framework embeddován v servletovém kontejneru. Výhody tohoto přístupu byly již zmíněny, velmi důležitým důvodem je, že pro správce serverů je jednoduché restartovat aplikaci.

Aplikace musí ukládat stav hry, tzn. seznam her, hráčů apod. To zajišťuje persistenční vrstva, která mohla být implementována několika způsoby (viz kapitola Persistence dat). Pro aplikaci je vhodnější použít JPA, a to z několika důvodů. Jedním z nich je zadání práce, které hovoří jasně. Vytvořit aplikaci za použití Enterprise OSGi, jehož součástí je i JPA a webová služba. Byla použita databáze Apache Derby DB. Důvodem byla snadnost použití. Apache Derby DB je přímo podporovaná Eclipse Gemini (zvolenou implementací Enterprise OSGI), takže jsou k dispozici přímo bundly, které zprostředkují připojení do databáze.

Samotná komunikace byla již naznačena v popisu komunikačního protokolu. Servlet dostane údaje, které zpracuje a pošle zpět informace pro klienta. Zde je důležité zmínit, že pro herní aplikaci tohoto druhu by se velmi hodilo, aby server mohl používat push zprávy na klienta. Aplikace je navržena tak, že je zde „falešný“ push - aplikace kontroluje údaje pokaždé, když je zavolána. Pokud je nějakou dobu nečinná, není schopna kontaktovat klienta. Pro aplikaci by bylo lepší použít nějakou jinou technologii, která by dovolovala push (duplexní kanál, jiný protokol pro přenos než HTTP...). Taková technologie by byla ovšem drahá, takže byl nakonec použit HTTP protokol, který je sice bezestavový, ovšem lze s ním docílit požadovaného výsledku.

Serverová aplikace se skládá z následujících bundlů:

- SupportBundle

  Bundle obsahuje rozhraní pro služby a model dat persistenčního bundle (IGameDAO a IPlayerDAO).

- PersistenceBundle

  Odpovídá za ukládání dat do databáze. Bundle poskytuje dvě služby do frameworku, odpovídající rozhraním ze SupportBundle.

- GameServletBundle

  Nese hlavní část činnosti serverové aplikace, zaregistruje se jako servlet, přijímá požadavky klienta, odpovídá za kontroly her a hráčů.

- DashboardBundle

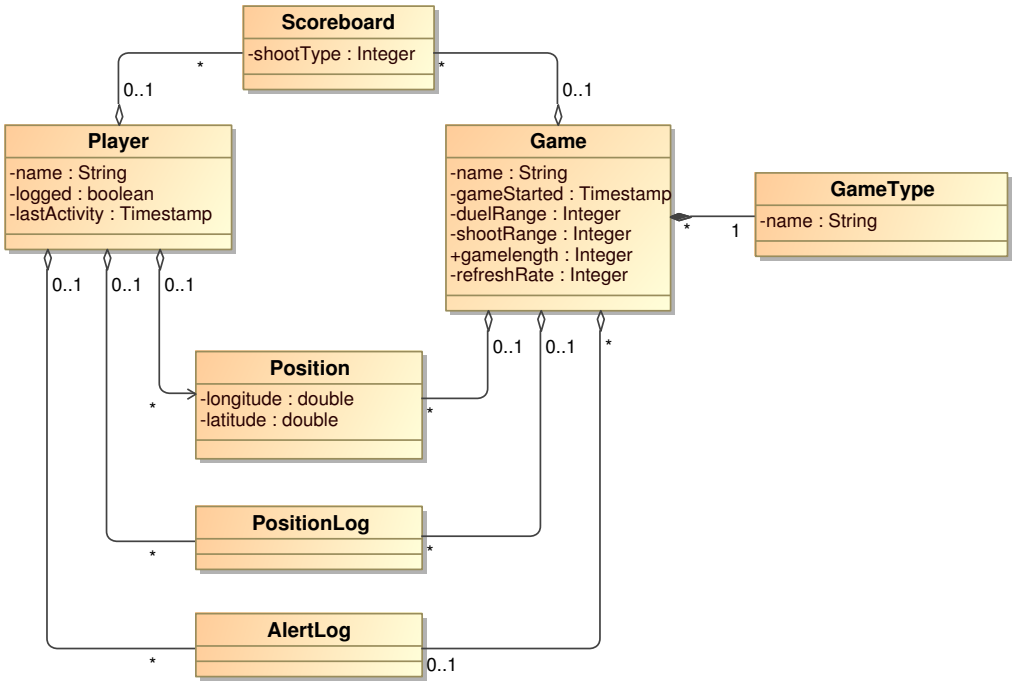
  Slouží jako bundle, kterým můžeme monitorovat a ovládat hry, hráče a obecně stav serveru.

5.4.1 Model dat a persistenční bundle

Většina dat, které aplikace používá, jsou ukládány do databáze. V této kapitole bych rád popsal, jak vypadají tabulky databáze a jakým způsobem se z nich čte v aplikaci.

Databáze

Jako databáze byla zvolena Apache Derby DB. Databáze obsahuje 7 tabulek a sama o sobě není příliš složitá (viz obrázek 5.2).



Obrázek 5.2: E-R-A model databáze

Pro hru jsou zásadní 4 tabulky:

- Player - tabulka, která obsahuje všechny informace o hráčích (login, poslední aktivitu a jestli je momentálně přihlášen)
- Game - tabulka obsahuje všechny aktivní hry (po ukončení hry se hra maže z databáze) včetně důležitých parametrů hry (název, kdy byla hra

založena, jak dlouho bude trvat, jak často se hráči hlásí, vzdálenosti pro duel a střelbu)

- Scoreboard - tabulka zásahů hráčů, obsahuje vždy, jaký hráč zasáhl jakého a kterým způsobem
- Position - tato tabulka udržuje aktuální pozice hráčů ve hře. Teoreticky je možné, aby hráč měl v tabulce více pozic, ale vždy má pouze jednu pozici pro danou hru

Přístup do databáze

Pro přístup do databáze bylo zvoleno JPA, konkrétně jeho implementace EclipseLink. Přístup do databáze je zprostředkován bundlem *PersistenceBundle*, který poskytuje aplikaci a frameworku služby typu *IPlayerDAO* a *IGameDAO*, které jsou specifikovány v *SupportBundle*. *PersistenceBundle* využívá modelu dat, který je také poskytován *SupportBundle*. Nebudu zde popisovat každou metodu rozhraní *IGameDAO* nebo *IPlayerDAO*, pro ilustraci dodám ukázku anotované třídy modelu a ukázku metody z DAO.

Jako ukázku jsem zvolil část třídy Player, která je navázaná stejnojmennou tabulku.

```
@Entity
public class Player implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  @SequenceGenerator(name = "PLAYER_ID_GENERATOR",
       sequenceName = "PLAYER_ID")
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "PLAYER_ID_GENERATOR")
  private int id;

  private String name;

  // bi-directional many-to-one association to Game
  @ManyToOne
  @JoinColumn(name = "GAME")
```

```
  private Game gameBean;

  @Column(name = "LOGGED")
  private boolean logged;

  @Column(name = "LAST_ACTIVITY")
  private Timestamp lastActivity;

  // bi-directional many-to-one association to Position
  @OneToMany(mappedBy = "player")
  private Set<Position> positions;

  // bi-directional many-to-one association to Position
  @OneToMany(mappedBy = "player")
  private Set<Scoreboard> sourcePlayerScoreboards;

  // bi-directional many-to-one association to Position
  @OneToMany(mappedBy = "destinationPlayer")
  private Set<Scoreboard> destinationPlayerScoreboards;
```

Jak je vidět, kousek třídy chybí. Třída má pro každou svoji datovou položku ještě getter a setter. Ze třídy jde vidět také, jak jsou definovány vztahy s jinými entitami v databázi. Na ukázce lze vidět vztah N..1 s entitou Position, nebo s Game. V JPA je totiž nutné vztah definovat na obou stranách, takže ve třídách Position a Game musí být anotován vztah 1..N.

Následuje ukázka metody z DAO, konkrétně z metody savePosition:

```
public void savePosition(Game game,~Player player,
                    double longitude,~double latitude) {

    EntityManager em = emf.createEntityManager();
    // nejdriv musim najit jestli tam uz pozice neni
    Query query = em.createQuery
      ("SELECT X FROM Position x WHERE x.player.id = "
        + player.getId()
        + " AND x.game.id = " + game.getId());
```

```
    Position position;
    boolean createRecord = false;
    if (query.getResultList().size() > 0) {
      position = (Position) query.getResultList().get(0);
    } else {
      position = new Position();
      createRecord = true;
    }
    position.setGame(game);
    position.setPlayer(player);
    position.setLatitude((float) latitude);
    position.setLongitude((float) longitude);

    PositionLog positionLog = new PositionLog();
    positionLog.setCreationTime(new Timestamp(
                    System.currentTimeMillis()));
    positionLog.setGame(game);
    positionLog.setLatitude((float) latitude);
    positionLog.setLongitude((float) longitude);
    positionLog.setPlayer(player);

    EntityTransaction transaction = em.getTransaction();
    if (!transaction.isActive())
      transaction.begin();
    if (createRecord) {
      em.persist(position);
    } else {
      em.merge(position);
    }

    em.persist(positionLog);

    em.flush();
    transaction.commit();
    em.close();
  }
```

Metoda je trochu delší, protože při uložení současné pozice hráče se musí vykonat několik kroků:

1. Nejdříve se musí zjistit, jestli hráč již nemá pozici uloženou. Pokud ano, musí se daná pozice přepsat a ne přidat další.
2. Dále se musí nastavit správné hodnoty do tabulky Position.
3. Dalším krokem je zalogování pozice do tabulky PositionLog.
4. Nakonec se data uloží do databáze.

5.4.2 Aplikační logika

Aplikační logika je obsažena hlavně v *GameServlet* bundlu. Bundle si v aktivátoru ověří, jestli má k dispozici služby persistenčního bundlu (implementované v rozhraní *IGameDAO* a *IPlayerDAO*). Pokud ano, pak se zaregistruje jako servlet a čeká na požadavky.

Popis aplikačního protokolu naznačil, že hráč se pravidelně hlásí zprávou typu report. Jako odpověď dostává nejen jednu ze souřadnic všech soupeřů, ale také seznam varování (na serveru je implementován pomocí třídy Alert, takže dále budu používat pojem alert).

Při každém požadavku je nutné zkontrolovat stav hry (jestli už hra nemá skončit, vzdálenost hráčů kvůli duelu, stáří alertů a duelů). Při každém požadavku proběhnou 3 důležité metody:

- checkGames - kontroluje, jestli hry již nemají končit...
- checkPlayers - kontroluje vzdálenost hráčů od sebe, dobu od poslední aktivity - hráč je totiž při neaktivitě delší než 5 minut odhlášen ze hry...
- checkAlerts - kontroluje jestli již nebyly některé Alerty a Duely na serveru příliš dlouho (standardní čas je 5 minut) a případně je řeší

Třída Alert

Třída Alert je klasická datová třída, které obsahuje následující položky:

- sourcePlayer - hráč, od kterého pochází Alert (může být null, pokud varování vzniklo ze serveru)

- destinationPlayer - hráč, kterého se Alert týká
- game - hra, ve které k Alertu došlo
- createTime - čas, kdy byl Alert vytvořen. Je to důležitý údaj, protože je pravidelně kontrolováno, jestli Alerty nejsou příliš staré
- alertType - mohou být následující druhy varování:
  - Fire

    Hlásí, že na hráče někdo vystřelil, třída samozřejmě obsahuje i hráče, který byl zdrojem tohoto výstřelu, hráče poté má možnost buď střelbu přijmout nebo odmítnout. Alerty i pozice hráčů jsou logovány do databáze, takže je možné později posoudit situaci, kdy hráč odmítl uznat, že byl zastřelen, ačkoliv měl. Pokud hráč neodpovídá na Alert po dobu delší než 1 minutu, je bráno, jako že výstřel přijal.

    Při každém výstřelu je nejdříve ověřena vzdálenost mezi hráči, nesmí být větší než ta, která byla nastavena při zakládání hry.
  - Time over

    Tento alert, jak název napovídá, říká hráči (klientské aplikaci), že hra skončila a má opustit hru.
  - Fire confirmed

    Pokud hráč uzná, že byl zastřelen, samozřejmě nepokračuje dále aktivně ve hře. Vidí ostatní hráče, ale nemůže střílet ani není uvažován do duelu.
  - Fire rejected

    Hráč může samozřejmě výstřel odmítnout (předpokládá se, že hráč, který vystřelil, se druhému ukáže, aby bylo vidět, že na něj opravdu mohl vystřelit a jen tak to nezkoušel). Pokud hráč výstřel odmítne, je to oznámeno střílejícímu hráči tímto Alertem.
  - Duel

    Pokud se hráči přiblíží na vzdálenost stanovenou při vytváření hry nastává duel. Oběma hráčům se zobrazí hláška o tom, že se účastní duelu a musí rychle zmáčknout tlačítko. Oba hráči odesílají serveru svůj reakční čas. Ten, který byl rychlejší vyhrává a přežívá.
  - Duel win

    Oznámení hráči, že vyhrál duel.

– Duel lost
Oznámení hráči, že prohrál duel.

Třída Duel

Třída podobná třídě Alert, slouží k ukládání stavu duelu. Duel musí být ukládán, protože není možné vracet hráči okamžitě výsledek jeho volání o účasti v duelu (musí se počkat, až se ohlásí protivník nebo uplyne časový limit). Podobně jako třída Alert je i tato třída poměrně jednoduchou datovou třídou. V následujícím výpisu ukáži jednotlivé položky třídy:

- player1 - odkaz na prvního hráče v duelu
- player2 - odkaz na druhého hráče v duelu
- game - odkaz na hru, ve které se duel odehrává
- player1Reaction - reakční čas prvního hráče v milisekundách (milisekundy zvoleny, protože je vrací systémová metoda System.currentMillis())
- player2Reaction - reakční čas druhého hráče
- createTime - poslední, ale velmi důležitou datovou položkou třídy, je čas vytvoření duelu (je uložen jako Timestamp, který je vytvořen ze systémového času)

Výpočet vzdálenosti

Server potřebuje počítat vzdálenosti mezi hráči hned při dvou příležitostech - při ověřování, jestli má dojít k duelu a pokud jeden hráč vystřelí na druhého.

Pro výpočet vzdálenosti dvou bodu podle GPS souřadnic jsem použil tzv. Haversinovu formuli. Algoritmus byl částečně převzat od Chrise Venesse ([17]). Pro ilustraci uvádím celou metodu výpočtu:

```
private long getDistance(double lat1, double lon1,
                         double lat2, double lon2) {
```

```
    int R = 6371000; // km
    double dLat = Math.toRadians(lat2 - lat1);
    double dLon = Math.toRadians(lon2 - lon1);
    double a = Math.sin(dLat / 2) * Math.sin(dLat / 2)
              + Math.cos(Math. toRadians(lat1))
              * Math.cos(Math.toRadians(lat2))
              * Math.sin(dLon / 2)
              * Math.sin(dLon / 2);
    double c = 2 * Math.asin(Math.sqrt(a));
    return Math.round(R * c);
}
```

Uvedený výpočet vrací vzdálenosti mezi body v metrech.

5.4.3 Dashboard

Aplikace může být částečně spravována a monitorována pomocí Dashboard-Servletu.

Bohužel způsob, který je použit pro aplikaci (Felix běží jako embedded v rámci servletového kontejneru), neumožňuje použít JSP ani podobný způsob pro zobrazení informací. Konkrétní implementace ExtHttpService a HttpService od Apache Felix vrací totiž pro getDispatcher vždy null, což v praxi znamená, že není možné jakékoliv přesměrování (forward) na jiný servlet nebo i JSP. Obecně se zdá, že forwardování není dobře napsané.

Jako náhradní variantu jsem tedy zvolil použití *PrintWriteru*, který je získáván z *HttpServletResponse*:

```
PrintWriter = response.getWriter();
```

Pomocí něho je tedy zapisován přímo HTML kód. Toto řešení není zrovna šťastné a je poměrně pracné pomocí něho něco tvořit nebo upravovat, ale na jiný způsob jsem nepřišel.

5.5 Klientská aplikace

Klientská aplikace je vytvořena pro mobilní zařízení s operačním systémem Android. Pro svoje fungování potřebuje, aby zařízení mělo paměťovou kartu a GPS sensor.

5.5.1 Architektura aplikace

V rámci aplikace běží framework Apache Felix (viz kapitola 3.3). Při tvorbě jsem stál před několika problémy, jak zajistit, aby instance běžícího frameworku byla k dispozici všem aktivitám, jakým způsobem přepínat z frameworku aktivity a jakým způsobem vůbec aktivity definovat.

Framework pro všechny aktivity

Je jasné, že framework se nastartuje v první aktivitě (v tomto případě *LoginActivity*). Dále je nutné nějakým způsobem předávat instance frameworku. Mezi aktivitami lze předávat data například pomoci tzv. Extra. Bohužel pro tento případ se moc nehodí. Hodí se například pro předávání jednoduchých dat, reference na objekt takto předávat jednoduše nejdou. Tedy možné to je, ale předávat lze pouze objekty, které implementují rozhraní Parcelable.

Protože není jednoduché použít Extra, tak jsem zvolil jinou variantu - spustit framework v rámci služby. Na platformě Android je možné spustit v zásadě dva druhy služeb: služba může být navázaná přímo na aktivitu (což se příliš nehodí), nebo může být aktivitou pouze spuštěna a svůj životní cyklus si řídí sama (takovéto služby se využívají například pro dlouhé výpočty, nebo stahování rozsáhlých dat). Pro náš problém jsem zvolil druhou jmenovanou variantu. V rámci *LoginActivity* se vždy spustí služba, která poskytuje přístup k frameworku. Tato služba má ovšem jedno specifikum, je zároveň vždy i navázaná na aktivitu.

Přepínání aktivit z frameworku

Bylo žádoucí, aby aktivity byly přepínány (obecně řízení životního cyklu aktivity) přímo z frameworku (ne přímo z aktivit). Aktivita se v Androidu

přepíná pomocí Intentu. Pro tento Intent ovšem potřebujeme konkrétní třídu (objekt typu Class) aktivity. Tyto objekty jsou k dispozici pouze v rámci Android aplikace, nikoliv v rámci frameworku. Proto je nutné objekty aktivit uložit do frameworku jako OSGi služby. Potom je již možné použít tyto třídy pro přepínání.

Ukázka registrace aktivity do frameworku:

```
// login activity
Properties props = new Properties();
props.put("ActivityName","LoginActivity");
m_felix.getBundleContext().registerService(Class.class.getName(),
        LoginActivity.class, props);
```

Ukázka přepnutí aktivity v rámci frameworku:

```
public String loginUser(Activity activity, String login,
                                    String password) {
  ServiceReference[] sRef2;
  sRef2 =
    bundleContext.getServiceReferences(Class.class.getName(),
                          "(ActivityName=JoinOrCreateActivity)");
  Class activityClass =
    (Class) bundleContext.getService(sRef2[0]);
  Intent myIntent = new Intent(activity, activityClass);
  activity.startActivity(myIntent);
```

K ukázce je tu jedna důležitá připomínka. Je velmi důležité, aby se metoda *startActivity* volala pomocí instance minulé aktivity. Android totiž aktivity skládá do zásobníku, ve kterém se tvoří řetěz aktivit, jak byly spouštěny, aby bylo možné po skončení aktivity (tlačítko back, nebo metoda *finish()*) se vrátit k minulé aktivitě (je možné pomocí vlastností Intentu zajistit, aby se některé aktivity neukládaly do zásobníku, nebo dokonce aby ho celý přepsaly).

Oddělení prezentační vrstvy od aplikační

Při tvorbě nastala ještě další otázka, kde definovat aktivity aplikace. Bylo by hezké, kdyby všechny aktivity byly definovány v rámci Apache Felix, to

ovšem naráží na problém koncepce aplikace na Android. Aplikace na Android musí mít všechny své aktivity vypsány v souboru *Androidmanifest.xml*. Pokud bychom chtěli definovat aktivity ve frameworku, máme problém. Proto došlo k oddělení prezentační vrstvy aplikace, která je celá v Androidu (aktivity, služby, řetězce, layouty ...), od aplikační a datové, která běží celá v rámci bundlů uvnitř frameworku.

5.5.2 Prezentační vrstva - Android

Prezentační vrstva se skládá ze dvou velkých částí: aktivit (včetně layoutů a řetězců) a služby, která má poskytuje rozhraní do frameworku.

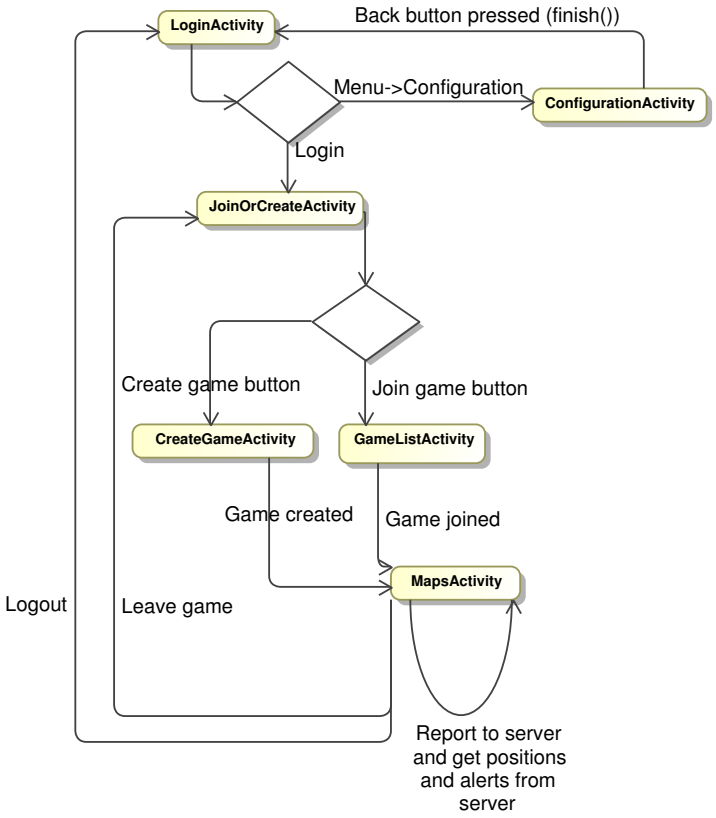
Aktivity klientské aplikace

Na následujícím obrázku (obrázek 5.3) lze vidět, jak v aplikaci dochází k přepínání jednotlivých aktivit. Každá aktivita si v metodě *onStart()* naváže *FelixService*.

Na obrázku lze vidět tyto aktivity (přechody mezi aktivitami budou dále popsány v kapitolách o jednotlivých aktivitách):

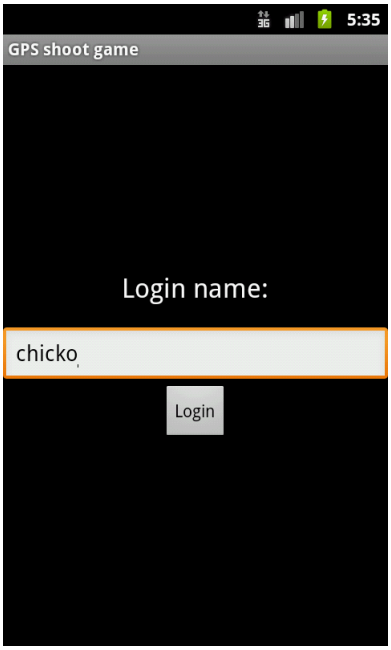
- LoginActivity
- ConfigurationActivity
- JoinOrCreateActivity
- CreateGameActivity
- GameListActivity
- MapsActivity

LoginActivity Je to úvodní aktivita, která se spouští s aplikací. Na začátku vždy vytvoří instanci Apache Felix. A zobrazí textové pole pro zadání uživatelského jména a tlačítko pro zalogování se do hry(viz obrázek 5.4). Uživatel zde zadá své jméno a klikne na **Login**, poté aplikace přejde do aktivity *JoinOrCreateGameActivity*. Dalším možným přechodem je přes tlačítko **Menu->Configuration**, kde aplikace přejde na *ConfigurationActivity*.



Obrázek 5.3: Diagram přechodů v závislosti na komunikačním protokolu

Aktivita si pamatuje poslední zadané jméno pomocí *Preferences*. Zde je opět využita služba *FelixService*.

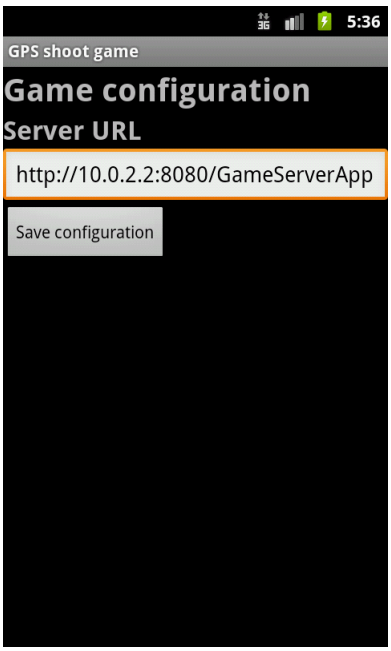


Obrázek 5.4: Login Activity

ConfiguratinActivity V této aktivitě lze nastavit pouze URL adresu serveru(viz obrázek 5.5). Z aktivity se lze dostat pouze tlačítkem zpět do *LoginActivity*.

JoinOrCreateActivity Další v řadě poměrně jednoduchých aktivit. Uživatel zde má pouze 4 možnosti (viz obrázek 5.6): přejít k vytvoření hry, přejít k připojení hry nebo tlačítkem zpět se odhlásit a přejít do *LoginActivity*, stejnou možnost má přes tlačítko menu->logout.

CreateGameActivity Uživateli se zobrazí formulář, k zadání parametrů nové hry. Přejít odsud může jen zpět (*JoinOrCreateActivity*) nebo tlačítkem hru vytvořit a připojit se k ní, tedy přejít do *MapActivity*. Poslední možností je tlačítko menu->logout, čímž uživatel přejde do *LoginActivity*.



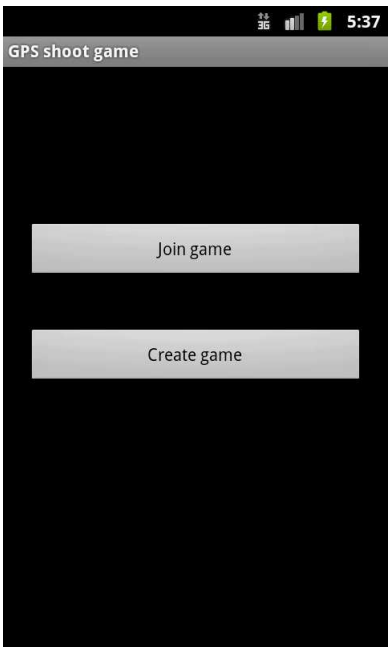
Obrázek 5.5: Configuration activity

GameListActivity Uživateli se zobrazí seznam her, které jsou k dispozici (případně prázdná obrazovka, pokud není k dispozici žádná hra)(viz obrázek 5.8. Odsud podobně jako v *CreateGameActivity* může jen dvěma směry zpět k *JoinOrCreateGameActivity* nebo klepnutím na hru se připojí k hře. Poslední možností je tlačítko menu->logout, čímž uživatel přejde do *LoginActivity*.

MapsActivity Z hlediska hry je to nejdůležitější aktivita. Většinu obrazovky pokrývá mapa (viz obrázek 5.9). K zobrazení mapy je použita komponenta z Google API MapView. Zde je důležité poznamenat, že k tomu, aby se aplikaci zobrazila mapa, je nutné se zaregistrovat a opatřit Google API Key. Na spodní části obrazovky je jeden Spinner (pro výběr hráčů), 2 tlačítka (**Fire** pro výstřel na hráč, **Center Me** pro vycentrování mapy na moji polohu) a zaškrtávací tlačítko **Center Me?**, jestli se má po každé aktualizaci polohy vycentrovat mapa.

Co se týče přechodů tak je zde jen pár možností, jak vyplývá i z obrázku (obrázek 5.3). Můžeme opustit hru nebo úplně odhlásit.

Důležité v této metodě je, že musí pravidelně podle nastavení obnovo-



Obrázek 5.6: JoinOrCreate activity

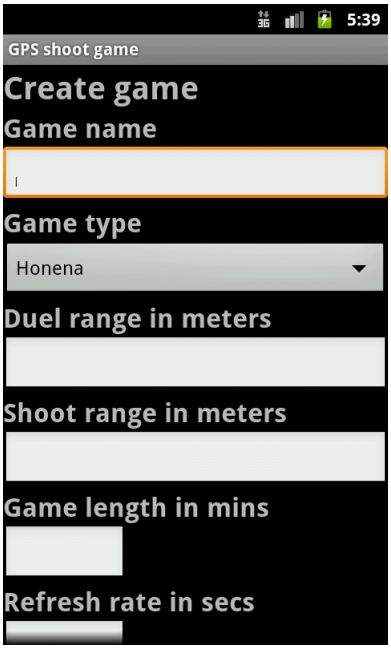
vacího času posílat reporty na server. Pro implementaci opakovaného spuštění je zde několik možností, je určitě nutné, aby „updater“ běžel v samostatné vlákně. Zde jsem využil třídy *Handler*. Této třídě se nastaví zpoždění, kdy se má spustit. Při každém spuštění si vždy naplánuje další spuštění. Samotný report je obsažen ve třídě *MapUpdateTask*, která se vytváří na začátku aktivity.

Obnova zobrazených dat Obnovu zajišťuje třída *MapUpdateTask*, která zajistí spuštění metod *MapsActivity*: *updatePositions()* a *updatePlayers()*.

- updatePositions

  Metoda, která zjistí, jestli jsou k dispozici GPS souřadnice uživatele a pokud ano, pak posílá report serveru. Jako odpověď dostává HashMap<String,Object>, který obsahuje: informaci o tom, jestli souřadnice soupeřů jsou zeměpisná délka nebo šířka, seznam Alertů, seznam pozic hráčů.

  Potom nastává fáze zpracování informací. Nejdřív upraví pozici hráče na mapě. Pozice hráče je zobrazena v tzv. *Overlay*, kde je vykreslen symbol křížku na dané pozici. Daleko zajímavější je ovšem vytvoření



Obrázek 5.7: Creategame activity

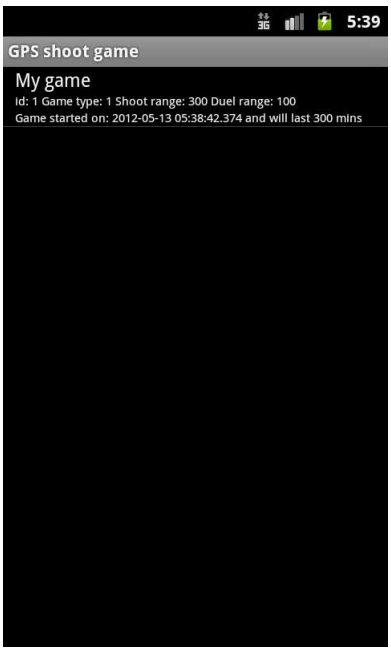
Overlay pro hráče, vzhledem k tomu, že máme k dispozici pouze jednu souřadnici. Naštěstí v rámci *MapView* je k dispozici položka *Projection*, která reprezentuje aktuálně zobrazený výřez mapy. Pomocí této souřadnice lze přepočítat bod určený GPS souřadnicí na souřadnice v pixelech přímo na mapě. Pozice jsou pak zobrazeny jako čáry od kraje ke kraji mapy vodorovně nebo svisle, podle toho, jestli je zadána šířka nebo délka.

- updatePlayers

  Tato metoda se pravidelně ptá serveru, kteří hráči jsou ve hře. Tato informace je využívána dvěma způsoby: pro zobrazení nabídky hráčů k výstřelu a pro ověření, že je hráč ještě na živu - pokud je mrtev, nemůže střílet.

Předávání uživatelského jména a id mezi aktivitami

Každá aktivita, kromě LoginActivity, potřebuje znát, který hráč je přihlášen, protože je to nutné v komunikaci se serverem. Proto je nutné tyto informace předávat. K tomu je využit systém Extra hodnot, které se dají uložit do Intentu. Zde je ukázka ukládání dat do Intentu:



Obrázek 5.8: Gamelist activity

```
Intent myIntent = new Intent(activity, activityClass);
myIntent.putExtra("userId", playerId);
myIntent.putExtra("login", login);
activity.startActivity(myIntent);
```

Třída FelixService

Třída FelixService slouží jako brána aplikace do frameworku, slouží jako jakási aplikační logika v rámci prezentační vrstvy. Při vytvoření této třídy se vytvoří a spustí framework Apache Felix. Po spuštění se nainstalují a spustí všechny bundly, které se nachází na předem známém místě na SD kartě (*/mnt/sdcard/gpsGame/bundles*). Potom se do frameworku zaregistrují všechny aktivity, které v aplikaci jsou. Poslední operací je získání OSGi služby IUIBundle z frameworku.

Přístup do frameworku Podobně jako *FelixService* slouží jako brána aplikace do frameworku, tak na druhé straně této brány stojí UIBundle (s službou odpovídající IUIBundle). Po získání UI služby z frameworku

Obrázek 5.9: Map activity

můžeme říkat frameworku, co dělat. Bohužel je zde jedna komplikace. Objekt získaný z frameworku je nám sice známý, ale nemůžeme ho přímo použít, důvodem jsou rozdílné ClassLoadery, které třídy identifikují. Pokud bychom tedy chtěli získaný objekt přetypovat na IUIBundle (získané z jar souboru bundlu), ohlásí nám *ClassCastException*.

Musel jsem tedy zvolit jiný přístup - reflexe. Protože znám jména jednotlivých tříd, mohu je vždy v objektu projít a zavolat s parametry, které jsou mi také známé. Tento přístup není vůbec elegantní, ale je funkční. Zde je ukázka volání funkce z UIBundlu:

```
public String loginUser(Activity activity, String login,
                                          String password)
      throws IllegalArgumentException, IllegalAccessException,
      InvocationTargetException {
    Class myclass = this.uiBundleFunctions.getClass();
    Method[] methods = myclass.getMethods();
    String message = null;
    for (int i = 0; i < methods.length; i++) {
      if (methods[i].getName().equals("loginUser")) {
        message = (String) methods[i].invoke(uiBundleFunctions,
```

```
          activity, login, password);
        break; // neni dale nutne prohledavat
      }
    }
    return message;
  }
```

5.5.3 Aplikační logika a síť'ová komunikace

Aplikační logika aplikace je obsažena v bundlech, které jsou spuštěny v rámci OSGi frameworku. Zde je seznam bundlů, které jsou použity v aplikaci, jejichž funkčnost budu postupně probírat:

- SupportBundle
- UIBundle
- CommnunicationBundle
- LocationBundle

Použité bundly

SupportBundle Support bundle, podobně jako u serverové aplikace, obsahuje rozhraní použitých služeb. V tomto případě:

- IUIBundleFunctions
- ICommunicationBundle
- ILocationBundle

UIBundle UIBundle je vlastně řídící bundle celé aplikace. UIBundle poskytuje frameworku důležité rozhraní *IUIBundleFunctions*. Pomocí tohoto rozhraní k němu přistupuje Androidí část aplikace. Fakticky potom řídí celou aplikaci, protože právě v UIBundlu se přepínají aktivity apod.

K tomu, aby bundle mohl poskytovat službu frameworku, sám potřebuje další dvě služby: *ILocationBundle* a *ICommunicationBundle*. Do té doby, než má k dispozici obě tyto služby, tak službu frameworku neposkytuje.

Jednotlivé metody rozhraní *IUIBundleFunctions* poskytují veškerou funkčnost, která je na prezentační vrstvě potřeba. Pokud je například potřeba nějaká síťová komunikace, je volána metoda rozhraní *ICommunicationBundle*. Nemá cenu zde vypisovat veškeré metody, které bundle poskytuje, ale pro představu uvedu příklady: *loginUser*, *getGames*, *joinGame*.

CommunicationBundle Další součástí aplikace je komunikační bundle. Bundle zajišťuje síťovou komunikaci se serverem. Má na starosti vlastní předání dat a zpracování odpovědi. Komunikace probíhá pomocí HTTP protokolu a server posílá odpovědi v XML. Způsob komunikace se serverem je popsán výše v kapitole o HTTP komunikaci.

LocationBundle Lokační bundle je asi nejjednodušší bundle v aplikaci. Jeho hlavní náplní je, že se zaregistruje k naslouchání změn na GPS sensoru zařízení (k tomu slouží třída *MyLocationListener*) a poslední pozici ukládá.

Listener je možné zaregistrovat přes aplikační kontext. Registrace listeneru je dvoukroková:

```
locationManager = (LocationManager) androidContext
        .getSystemService(Context.LOCATION_SERVICE);
locationManager
        .requestLocationUpdates(LocationManager.GPS_PROVIDER,
              0, 0, locationListener);
```

Zástupné komponenty

Díky tomu, že každá služba implementuje rozhraní, je možné mít různé komponenty, které poskytují služby. Jako příklad zde uvedu dvě možnosti zástupných komponent (bundlů).

Simulace polohy Jednou z možností zástupných komponent je vytvoření lokačního bundlu tak, že ve skutečnosti nepotřebuje GPS sensor zařízení, ale

poskytuje nějakou sadu GPS souřadnic cyklicky.

Simulovaná síťová komunikace Další možností je simulovat i síťovou komunikaci, což se může hodit pro to, když má aplikace sloužit jenom jako nástroj pro zjištění polohy a její zobrazení na mapě.

5.6 Ověření výsledné aplikace

Výsledkem práce jsou dvě aplikace: aplikace pro server a aplikace pro mobilní zařízení s Android. Vývoj serverové aplikace probíhal na lokálně na nainstalovaném Apache Tomcat. Požadavky klienta byly simulovány pomocí prohlížeče. Klientská část byla vyvíjena na emulátoru. Požadavky na server byly posílány na adresu 10.0.2.2, což je speciální adresa Android emulátoru pro komunikaci aplikací emulátoru s hostitelským počítačem.

Aplikace byla následně testována na reálných zařízeních (HTC Desire, Xperia Mini Pro). Na zařízeních byl nainstalován Android verze 2.2.2 (HTC Desire) a 2.3.3 (Xperia Mini). Jako server byl využit školní server Ares (students.kiv.zcu.cz) na portu 9999. Zařízení HTC Desire lze s použitím HTC Sync využít jako vývojové zařízení (lze na něj aplikaci přímo nahrávat z Eclipse, lze monitorovat přímo jeho LogCat...). Díky tomu šli ověřit aspekty aplikace, které se neprojevili při provozu na emulátoru (viz kapitola 6.1).

5.7 Rozšíření aplikace

Aplikaci jako takovou je možné dále rozšiřovat, a to jak na straně serveru, tak i na straně klienta. Zčásti lze využít rozdělení aplikace na bundly.

5.7.1 Více typů her

V současné podobě podporuje aplikace pouze jeden typ hry. Přitom je ale připravena na to, aby bylo k dispozici více typů her. Příkladem jiných typů her je souboj týmů nebo hon na lišku.

K implementování této možnosti je připravena databáze, která obsahuje tabulku pro typ hry, která je v současné době nevyužitá. Pro podporu více typů her je ovšem zapotřebí zasáhnout do aplikační logiky. Funkce, které kontrolují pozice hráčů, stáří her, Alertů a Duelů, by musely být upraveny, aby vždy zkontrolovaly, o jaký typ hry se jedná a podle čeho se chovat.

5.7.2 Více životů pro hráče

Dalším rozšířením hry by byla možnost určení počtu životů hráče. Implementace této změny by byla složitější než předchozí rozšíření. Protože by se musela rozmyslet koncepce oživování hráče, jestli by se oživoval u nějaké vlajky (pak by se muselo toto místo někam ukládat, což by znamenalo úpravu datového modelu), nebo po nějaké době, která by se ovšem musela také někde specifikovat.

5.7.3 Lepší lokační bundle

Současný lokační bundle je velmi jednoduchý. Lokační bundle by šel vylepšit, aby zohledňoval i přesnost údajů. Také by zde šel zavést algoritmus, který by vyhlazoval případné výkyvy informací ze sensorů.

5.7.4 Dashboard na mobil

Další dobře použitelnou pomůckou pro aplikaci by byla další aplikace, která by sloužila jako současný bundle *DashboardServlet*. Uživatel by mohl spravovat server i ze svého mobilního zařízení. Pro implementaci tohoto řešení by se muselo rozšířit rozhraní pro správu přes HTTP.

6 Zkušenosti s použitím OSGi

Během vývoje aplikace jsem narazil na řadu předpokládaných i méně předpokládaných překážek. Některé jsem již zmínil, například zprovoznění samostatného frameworku na Android. Další jsem zmínil jen okrajově nebo vůbec ne.

Poměrně krátce po začátku vývoje jsem zjistil, že pro klientskou a serverovou aplikaci bude potřeba jiný přístup. Zatímco vývoj na serveru je po zprovoznění základního rámce, který spustí framework, nainstaluje a spustí bundly, velmi podobný vývoji jakékoliv jiné Java aplikace. Při vývoji klientské části je situace trošku odlišná a to hlavně tím, že je nutné používat buď emulátor nebo zařízení, na které máte ovladače, což práci docela dost zpomaluje.

6.1 Použití emulátoru nebo reálného zařízení

Obecně lze říct, že se vyplatí aplikaci nejdříve odladit na emulátoru a teprve potom aplikaci zkoušet na reálném zařízení (minimálně kvůli bezpečnosti zařízení a dat na něm). Menším problémem je rychlost emulátoru, která je v porovnání s reálným zařízením mnohem menší (porovnáváno s HTC Desire) a také poměrně dlouho trvá, než vůbec emulátor naběhne (někdy to trvá i několik minut). Proto se vyplatí emulátor nastartovat a nevypínat ho.

Použití reálného zařízení na ladění má také své překážky. Předně, aby bylo možné zařízení pro takové účely použít, musí být k dispozici ovladač (standardně je k dispozici zařízení Nexus - vývojářský telefon Google). Pro můj telefon (HTC Desire) je také ovladač k dispozici, ale jen v rámci aplikace HTC Sync. Zařízení se musí připojit k PC, nastavit mód synchronizace, ale synchronizace se nesmí provádět, pak je zařízení k dispozici k ladění.

Při ladění na emulátoru si musíme dát pozor ještě na jednu věc: výpis na *System.out* a to jak přímo *System.out.println()*, tak i nepřímo ve vyjímkách *e.printStackTrace*. Na emulátoru aplikace funguje spolehlivě, ale na reálném zařízení aplikace spadne s *RunTimeException*.

6.2 ADT plugin pro Eclipse

Klientskou aplikaci jsem vyvíjel v Eclipse za použití ADT pluginu. Obecně mohu toto řešení doporučit. Pluginu se pouze nastaví cesta k SDK Androidu a dále již není třeba nic řešit. Plugin poskytuje obdobu nástroje DDMS pro správu zařízení (souborový systém, ovládání sensorů).

6.3 Příprava bundlů

Pro kompilování OSGi bundlů jsem používal Maven, kde jsem využil *maven-bundle-plugin* od Apache. Tento plugin poskytuje poměrně jednoduché nastavení bundlů, sám vytvoří manifest, ve kterém určí exportované a importované balíky.

Po kompilaci je nutné do bundlů dodat soubor *classes.dex*, jak již bylo zmíněno v kapitole o přípravě bundlů. Zde jsem ovšem narazil na problém pod Windows 7, a to konkrétně s právy. Pokud je SDK nainstalováno na standardní umístění do *C:\ Program File*, pak je nutné mít správcovská práva pro spuštění potřebných nástrojů. Situaci ovšem úplně nevyřeší ani instalace jinam, protože je pravděpodobně chyba ve skriptu *dx.bat*, který nemůže najít další potřebné nástroje. Problém jsem řešil tak, že jsem bundly (jar soubory) připravoval přímo v adresáři se skriptem *dx.bat*.

6.4 Správné nastavení pořadí spouštění bundlů

Při instalaci a spuštění bundlů ve frameworku je třeba dbát na správné pořadí bundlů. Aplikace je postavená tak, že prohledává určený adresář a instaluje všechny jar soubory. Bohužel může nastat situace, kdy framework sám o sobě nedokáže rozebrat strom závislostí a nainstalovat bundly ve správném pořadí. Problém jsem řešil, i když ne moc elegantně, tím, že jsem bundly přejmenoval tak, že první bundle začíná znakem 0, další 1 ...

6.5 Správné nastavení frameworku

Velmi důležité je nastavit správně, které balíky budou jako systémové, před spuštěním frameworku, protože například balíky s nástroji pro zpracování XML nejsou standardně poskytovány. Systémové balíky jsou pak poskytovány systémovým bundlem (bundlem s id 0). Aplikace potom někdy padá se špatně čitelnými chybami.

6.6 Nastavení závislostí pro Android pod Maven

V oficiálním maven repositáři jsou k dispozici artefakty pro Android. Tyto artefakty bohužel nejsou úplně kompletní, neobsahují knihovnu Google API. Nastalou situaci lze vyřešit tím, že se použije lokální SDK, vytvoří se z něho artefakty, které potom poskytují API. Postup je převzatý od Manfreda Mosera ([11]). Po instalaci se do závislostí v *pom.xml* uvede toto:

```
<dependency>
  <groupId>android</groupId>
  <artifactId>android</artifactId>
  <version>2.1_r3</version>
  <scope>provided</scope>
</dependency>
```

6.7 Zhodnocení použití OSGi komponent

Velkou výhodou použití OSGi je snadná výměna komponent. Pro výměnu komponent stačí přepsat soubory s bundly. Takže rozšíření aplikace je potom velmi jednoduché.

Na druhou stranu použití bundlů ve skutečnosti porušuje bezpečnostní pravidla Androidu. Aplikace na Android jsou založené na tom, že se napřed ví, co dělají, kde jsou apod. Bundly jsou části kódu programu, nad kterými nemá systém přehled. Zároveň samozřejmě může pokulhávat verzování aplikací, protože může být rozdílné pro bundly a pro aplikaci.

Použití OSGi komponent také trochu komplikuje rychlost vývoje. Celkově posoudit použití OSGi komponent na mobilních zařízeních Android není jednoduché. V celkovém obraze si myslím, že snadná možnost výměny komponent vyváží ostatní nevýhody.

7 Závěr

Cílem mé práce bylo prostudovat možnost OSGi komponent na mobilních zařízeních. Tyto možnosti jsem měl následně ověřit vytvořením herní aplikace na mobilní zařízení s operačním systémem Android.

Díky nabytým znalostem se mi podařilo vytvořit funkční aplikaci, která potvrzuje použitelnost OSGi na mobilních zařízeních. Hlavním přínosem mé aplikace není hra samotná, ale ukázka mobilní aplikace založené na OSGi se server-klient modelem. Výhodou použití komponentového frameworku je i snadná rozšiřitelnost aplikace.

Během tvorby mé práce jsem získal mnoho zkušeností s tvorbou aplikací pro Android. Prohloubil jsem svoje zkušenosti s vývojem servletových aplikací. Dále jsem si rozšířil znalosti o využití OSGi a v neposlední řadě jsem se zdokonalil v práci s JPA.

V širším využití komponentových frameworků na mobilních zařízení vidím velkou budoucnost díky jejich modularitě a znovupoužitelnosti. Otázkou však je, jestli jejich širšímu rozšíření a integraci nezabrání možná bezpečnostní rizika spojená s jejich modularitou.

Literatura

[1] ABLESON, W. F. et al. *Android in Action*. Shelter Island : Manning, 2012. ISBN 1-61729-050-5.

[2] *Android (operating system)* [online]. 2012. [cit. 15.4.2012]. Dostupné z: `http://www.openhandsetalliance.com/oha_members.html`.

[3] *Apache Aries Home* [online]. apr 2012. [cit. 15.4.2012]. Dostupné z: `http://aries.apache.org/`.

[4] *Apache Felix Framework and Google Android* [online]. 2012. [cit. 15.3.2012]. Dostupné z: `http://felix.apache.org/site/apache-felix-framework-and-google-android.html`.

[5] *Apache Felix HTTP Service* [online]. 2012. [cit. 15.4.2012]. Dostupné z: `http://felix.apache.org/site/apache-felix-http-service.html-#ApacheFelixHTTPService-UsingtheServletBridge`.

[6] CUMMINS, H. – WARD, T. *Enterprise OSGi in Action*. Shelter Island : Manning, 2012. ISBN 9781617290138.

[7] *Example 6 - Spell Checker Service Bundle* [online]. 2012. [cit. 15.4.2012]. Dostupné z: `http://felix.apache.org/site/-apache-felix-tutorial-example-6.html`.

[8] *Gemini Home* [online]. 2012. [cit. 15.4.2012]. Dostupné z: `http://www.eclipse.org/gemini/`.

[9] *Windows Phone and the Cloud–an Introduction* [online]. 2012. [cit. 30.4.2012]. Dostupné z: `http://msdn.microsoft.com/en-us/magazine/ff872395.aspx`.

[10] *Metro User Interface Controls* [online]. 2012. [cit. 30.4.2012]. Dostupné z: `http://metroui.com/`.

[11] MOSER, M. *maven-android-sdk-deployer* [online]. 2012. [cit. 15.4.2012]. Dostupné z: https://github.com/mosabua/maven-android-sdk-deployer.

[12] *Members* [online]. 2012. [cit. 15.4.2012]. Dostupné z: http://en.wikipedia.org/wiki/Android_(operating_system).

[13] *JDBC* [online]. 2012. [cit. 20.1.2012]. Dostupné z: http://www.orafaq.com/wiki/JDBC.

[14] *OSGi Service Platform Enterprise Specification* [online]. 2010. [cit. 15.4.2012]. Dostupné z: http://www.osgi.org/download/r4v42/r4.enterprise.pdf.

[15] TEAM, T. C. GPS HUNTING ADVENTURE. 2009.

[16] UPTON, C. *OSGi Enabled War* [online]. 2012. [cit. 15.4.2012]. Dostupné z: http://heapdump.wordpress.com/2010/03/25/osgi-enabled-war/.

[17] VENESS, C. *Calculate distance, bearing and more between Latitude/Longitude points* [online]. 2012. [cit. 20.4.2012]. Dostupné z: http://www.movable-type.co.uk/scripts/latlong.html.

[18] *What is Android?* [online]. 2012. [cit. 15.4.2012]. Dostupné z: http://developer.android.com/guide/basics/what-is-android.html.

[19] *What Is Cocoa?* [online]. 2012. [cit. 30.4.2012]. Dostupné z: http://developer.apple.com/library/ios/#DOCUMENTATION/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html#//apple_ref/doc/uid/TP40002974-CH3-SW16.

[20] *Write Objective-C Code* [online]. 2012. [cit. 29.4.2012]. Dostupné z: https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/RoadMapiOS/Languages/WriteObjective-CCode/WriteObjective-CCode/WriteObjective-CCode.html.

[21] ŘEHOříK, F. *Windows Phone 7* [online]. 2012. [cit. 19.4.2012].

A Uživatelská příručka

A.1 Serverová část

A.1.1 Instalace na server

Serverová část potřebuje pro svůj chod servletový kontejner a databázi. V instalačním adresáři je k dispozici jak databáze(Apache Derby), tak i servletový kontejner (Apache Tomcat). Pro použití stačí adresáře nakopírovat na server a spustit. Tomcat je nakonfigurován na port 9999, servisní port 9998, Derby běží na portu 9990. Samozřejmě je možné toto nastavení změnit. Porty Tomcat se mění pomocí konfiguračních souborů (více v manuálu pro Tomcat na stránkách výrobce). Port Derby lze snadno změnit ve spouštěcím skriptu (za argumentem -p změnit číslo portu).

Nastavení portu databáze v PersistenceBundle

Pokud změníme číslo portu databáze, je nutné nastavit správně port v souboru *persistence.xml* v *PersistenceBundle*. Stejným způsobem je samozřejmě možné použít i jinou databázi.

```
<property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:9990/GPSGameDB;create=true" />
```

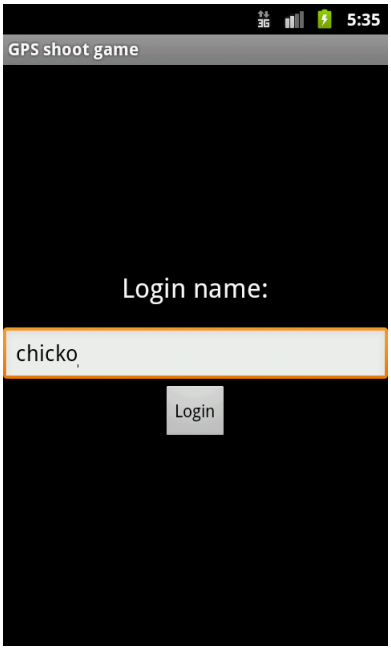
Obecně úpravou *persistence.xml* a dodáním JDBC ovladačů lze použít i jinou databázi než Apache Derby.

A.2 Klientská část

Účelem hry je pohyb a vyhledávání protihráčů v herním prostoru a jejich zasažení. Po uplynutí herní doby hráč, který má nejvíce bodů vítězí.

A.2.1 Přihlášení

Po spuštění aplikace se úvodní přihlašovací obrazovka (obrázek A.1). Zde uživatel zadá své přihlašovací jméno a stiskne tlačítko *Login*. Pokud se již uživatel přihlásil dříve, aplikace si pamatuje jeho přihlašovací jméno.



Obrázek A.1: Login Activity

A.2.2 Konfigurace serveru

Uživatel může nakonfigurovat na jaký server se bude přihlašovat. Na přihlašovací obrazovce stiskne tlačítko **Menu** -> **Configuration**. Objeví se mu obrazovka, kde zadá adresu serveru a potvrdí. Nyní se může po návratu zpět (tlačítko **Zpět**) přihlásit na požadovaný server.

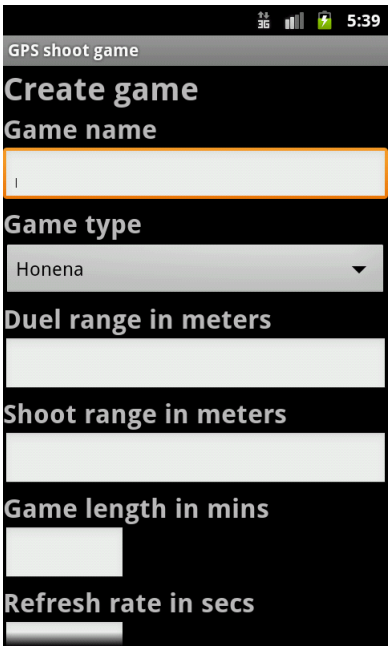
A.2.3 Vytvoření a připojení se ke hře

Po přihlášení si uživatel musí vybrat, jestli chce hru založit, nebo se připojit k vytvořené hry. Pokud se chce uživatel připojit ke hře stačí klepnout na hru ze seznamu (obrazovka je prázdná, pokud není k dispozici žádná hra).

Vytvoření hry

Pokud chce uživatel vytvořit hru objeví se formulář (obrázek A.2), kde musí zadat několik údajů:

- Game name – název hry může být libovolný neprázdný
- Game type – je nastaven na Honěná, reálně nemá vliv na hru
- Duel range – vzdálenost v metrech, na kterou když se hráči přiblíží nastane duel
- Shoot range – vzdálenost, na kterou po sobě hráči mohou střílet, když se vidí
- Game length – údaj v minutách, jak dlouhá má hra být
- Refresh rate – jak často se aktualizují data ze serveru (v sekundách)



Obrázek A.2: Creategame activity

A.2.4 Samotná hra

Po přihlášení do hry se hráči objeví obrazovka s mapou (obrázek A.3). Na té vidí křížkem označenou svou pozici (případně může dole blikat hláška *Waiting for signal GPS...*) a vodorovnými, či svislými čarami vidí své soupeře. Pokud hráč vidí protihráče, zvolí ve spodní části jeho jméno a stiskne **Fire**. Po nějaké době hráč zjistí výsledek výstřelu (protihráč může výstřel odmítnout s tím, že ho hráč neviděl). Pokud se hráči přiblíží natolik, že je vzdálenost menší než nastavena v *Duel range* nastává duel. V tomto případě hráč musí co nejrychleji zmáčknout tlačítko dialogu, protože vyhrává ten, který byl rychlejší.

Obrázek A.3: Map activity

A.2.5 Výměna komponent

K dispozici jsou dvě další komponenty: jedna simulující polohu, druhá simulující komunikaci se serverem sloužící pouze pro orientaci na mapě. Komponenty použijeme následujícím postupem:

1. Přepsání souborů bundlů – nejdříve musíme přepsat soubory v *SD karta* \ *gpsGame* \ *felix* \ *bundles*
2. Vymazání cache Felix – poté je nutné vymazat cache Felix – smazání adresáře *felix_cache*
3. Ukončení aplikace – pokud aplikace stále běží je třeba ji ukončit (včetně spuštěné služby)
4. Spuštění aplikace – nyní můžeme aplikaci spustit s novými komponentami

A.2.6 Ukončení aplikace

Po ukončení aplikace občas zůstává běžet služba *FelixService*. Pro řádné ukončení je potřeba službu ukončit (běžící služba nijak nebrání fungování aplikace, nebo čehokoliv jiného). Služba jde ukončit přes **Menu->Nastavení->Aplikace->Správa služeb**. Poté se někdy znova spustí aplikace, která když se znova ukončí již nic není spuštěno.