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

Bakalářská práce

Vytvoření aplikace na bázi OS Android pro ovládání stimulátoru pro neuroinformatické experimenty

Plzeň 2017 Petr Štechmüller

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 1. května 2017

Petr Štechmüller

Poděkování

Rád bych poděkoval Ing. Pavlu Mautnerovi, Ph.D za pomoc při tvorbě mé bakalářské práce. Děkuji za cenné rady, věcné připomínky a poskytnuté konzultace.

Velký dík patří mým rodičům, kteří mě podporovali a motivovali po celou dobu studia.

Abstract

The topic of this bachelor thesis is creationg of an application for OS Android. It's supposed to control a stimulator for neuroinformatic experiments which are developed in University of West Bohemia in Pilsen, dept. of Computer Science and Engineering. Theortic part is explaining some basic concepts as EEG and evoked potentials. In another part is introduced OS Android it self and its options for testing an application. Last part is describing an implementation of application it self to device using OS Android.

Abstrakt

Předmětem této bakalářské práce je vytvoření aplikace pro Android pro ovládání stimulátoru pro neuroinformatické experimenty vyvíjeného na katedře informatiky na Západočeské univerzitě v Plzni. Teoretická část se zabývá vysvětlením základních pojmů jako je elektroencefalografie, evokované potenciály. Dále je představena samotná platforma Android a možnosti testování aplikace. Realizační část popisuje vlastní implementaci aplikace do zařízení s operačním systémem Android.

Obsah

1 Úvod

V dnešní době je populární, aby každý kus elektroniky, ať už je to domácí spotřebič, hodinky, nebo auto, měl vlastní aplikaci na ovládání. Stimulátor na katedře informatiky v Plzni měl dosud pouze nešikovné ovládání pomocí dotykového displaye. Cílem této bakalářské práce je vytvořit aplikaci, která by nahradila stávající ovládání. Protože je Android nejrozšířenější platformou na trhu, byl jasnou volbou při volbě operačního systému.

Teoretická část práce čtenáře seznámí s elektroencefalografií, ze které vycházejí evokované potenciály. Dále se rozebere problematika stimulátoru. Zbylá část teorie se věnuje platformě Android.

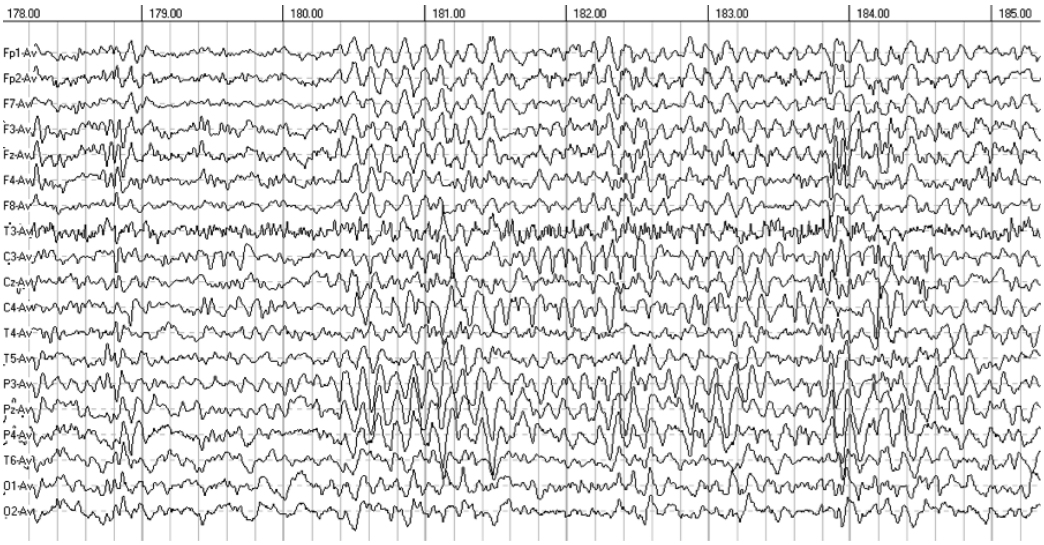
Praktická část zahrnuje tvorbu samotné aplikace pro ovládání stimulátoru.

2 Teoretická část

V následující části bude stručně objasněno, co to je elektroencefalografie, evokované potenciály, a jaké jsou možnosti současných stimulátorů. Dále popíšu, jak se programuje pro platformu Android.

2.1 Elektroencefalografie

Elektroencefalografie [6] je jedna ze základních diagnostických metod v neurologii a psychiatrii. Je to neinvazivní metoda, která snímá povrchovými elektrodami elektrickou aktivitu mozku. Existuje i invazivní metoda, při které se umístí elektrody přímo na mozek. EEG umožňuje diagnostiku a monitorování různých chorob, jako je: epilepsie, migréna, poruchy vědomí nebo spánku,... Výstupem měření je elektroencefalogram, což je záznam, který graficky znázorňuje mozkovou aktivitu. Příklad elektroencefalogramu je vidět na obrázku 2.1.

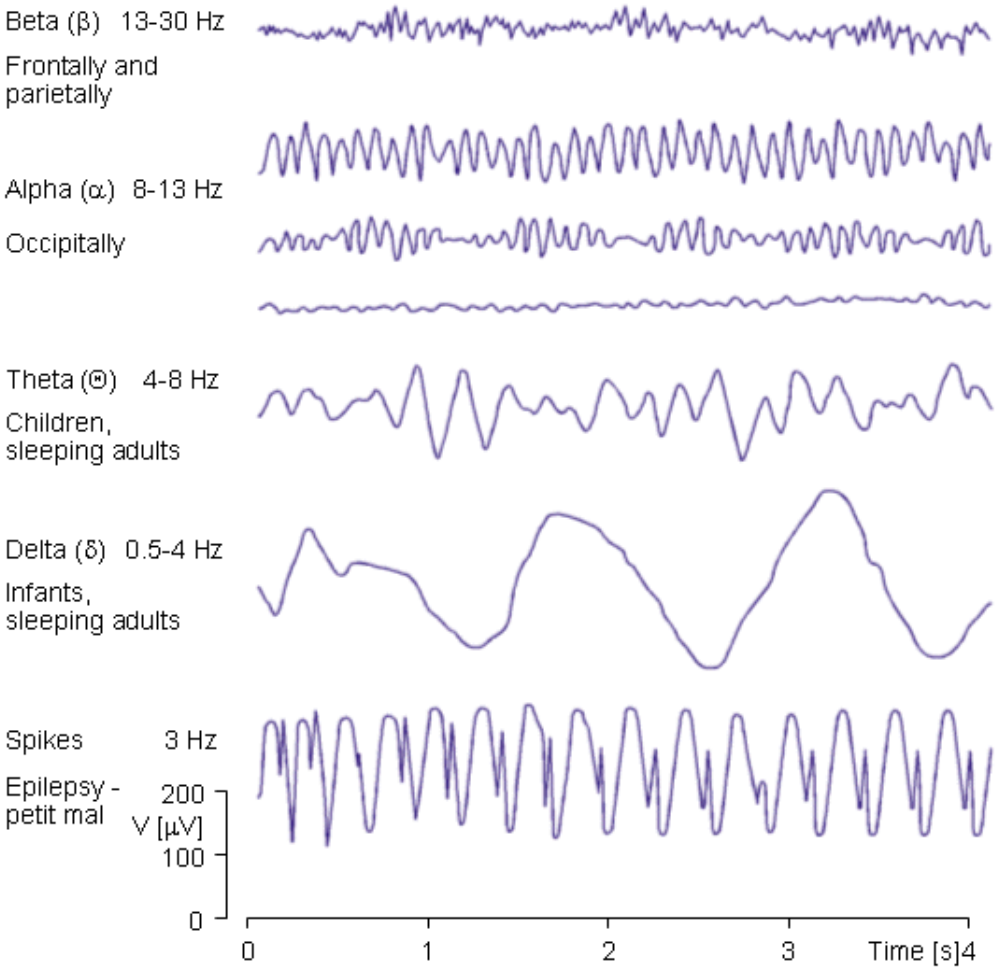


Obrázek 2.1: Příklad EEG záznamu

EEG křivku na první pohled tvoři nepravidelné a chaotické vlny, ve kterých je zkušený odborník schopen vidět periodicitu a na základě toho diagnostikovat poruchu mozku. Existuje několik typů vln 2.2:

- *Alfa* vlny o frekvenci 8-13 Hz s amplitudou 5-100 µV. Vyskytují se v bdělém stavu, při relaxaci a při zavřených očích.

- *Beta* vlny o frekvenci 13-30 Hz s amplitudou 2-20 µV. Vyskytují se v bdělém stavu, při přemýšlení, soustředění a ve stresových situacích.
- *Gama* vlny o frekvenci 30-50 Hz s amplitudou 2-10 µV. Vyskytují se v bdělém stavu. Když nejsou přítomny, tak to znamená, že subjekt není schopen se učit.
- *Delta* vlny o frekvenci 0.5-4 Hz s amplitudou 20-200 µV. V dospělosti se vyskytují v hlubokém spánku, v bdělém stavu znamenají poruchu soustředění.
- *Theta* vlny o frekvenci 4-8 Hz s amplitudou 5-100 µV. Vyskytují se při ospalosti. Objevují se hlavně u dětí a adolescentů.



Obrázek 2.2: Základní typy vln [2]

2.2 Evokované potenciály

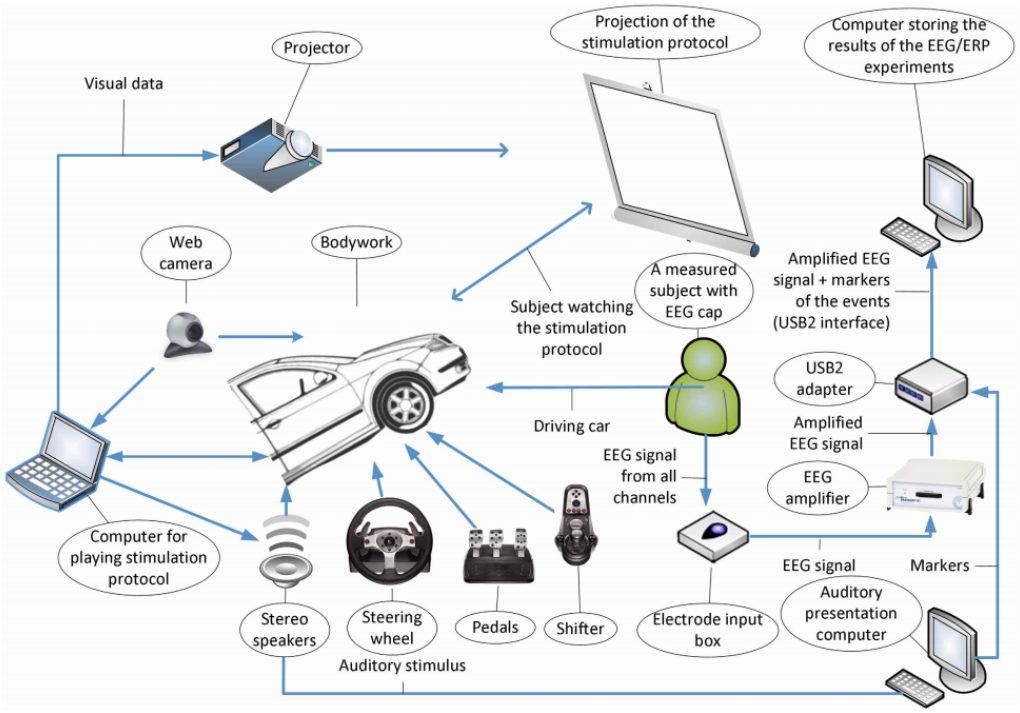
Evokovaný potenciál lze chápat jako elektrofyziologickou odpověď centrálního nervového systému na rozdílné typy stimulací (zvukové, obrazové) [4]. Evokované potenciály lze zaznamenávat pomocí elektrod umístěných na hlavě. Získaný záznam má tvar krátkodobých vln s velmi nízkou amplitudou. Tyto krátkodobé vlny velmi závisí na síle stimulu a na mentálním stavu subjektu. K dostatečnému určení ERP je nutné subjekt opakovaně stimulovat stejným podnětem. Záznam pro evokované potenciály vzniká jako složka při měření EEG. Naměřený signál je potřeba odfiltrovat od ostatních složek v EEG. K filtrování se používá buď průměrování, nebo tzv. "single-trial"analýza.

Abychom mohli provádět průměrování, je nutné měřený subjekt opakovaně stimulovat a poté průměrovat epochy, které odpovídají stejným stimulům. K tomu je potřeba synchronizovat stimulační zařízení se záznamovým zařízením EEG a umístit k EEG signálu značky odpovídající výskytu jednotlivých stimulů. Na základě těchto značek se pak EEG signál segmentuje a segmenty (epochy) odpovídající stejným stimulům se následně průměrují.

Podle druhu stimulace, lze ERP rozdělit na:

- sluchové - stimuluje se pomocí krátkého pípnutí o určité frekvenci
- zrakové - stimulace probíhá na monitoru, kde se objevuje obraz ve tvaru šachovnice, jejíž políčka střídavě mění barvu
- somatosenzorické - je vyšetřována periferní nervová soustava, kterou tvoří všechny nervy mimo centrální nervovou soustavu, tj. v mozku a míše[3]

K realizaci samotného experimentu je nutné mít k dispozici kromě EEG čepice ještě stimuly, na které subjekt reaguje, počítač s programem pro nahrávání dat, zvukotěsnou kabinu (při stimulaci zvukem) a hlavně stimulátor, pomocí kterého se zobrazují jednotlivé stimuly. V učebně UU403 na katedře informatiky v Plzni je toto vybavení k dispozici. Kompletní výčet pomůcek je vidět na obrázku 2.3.



Obrázek 2.3: Pomůcky pro realizaci experimentu

2.3 Stimulátory

Stimulátory lze v současné době rozdělit na dva druhy:

- Softwarové - stimulátor je realizován v podobě programu běžícího na osobním počítači. Nevýhodou těchto stimulátorů je zpoždění mezi výskytem stimulu a výskytem synchronizačního impulzu a zpoždění, které vzniká při zpracování odezvy od měřeného subjektu. Korekce těchto zpoždění je obtížná a prakticky nerealizovatelná. Jednou z možností řešení by bylo používat program na realtimovém operačním systému.

- Hardwarové - stimulátor je realizován samostatným hardwarem, takže odstraňuje problém se zpožděním. Nevýhodou těchto stimulátorů je jejich jednoúčelnost a uzavřenost celého systému, takže je není možné modifikovat a upravovat pro jiné experimenty.

Mezi nejznámější softwarové stimulátory patří program Presentation, který je vyvíjen firmou Neurobehavioral Systems Inc.[1] a opensource projekt OpenSesame[2].

[1]Webové stránky firmy Neurobehavioral Systems Inc. - http:// www.neurobs.com/
[2]Webové stránky projektu OpenSesame - http://osdoc.cogsci.nl/

- Presentation je placený nástroj pro tvorbu experimentů pro neurovědy na operačním systému Windows. Podporuje sluchové, vizuální a multimodální podněty. Pro tvorbu experimentů se využívají dva jazyky: SDL[3] a PCL[4]. SDL je jazyk, který popisuje vlastnosti jednotlivých stimulů. PLC je programovací jazyk, pomocí kterého se implementují scénáře experimentů.

- OpenSesame je multiplatformní opensource nástroj, pomocí kterého lze s využitím grafického rozhraní vytvářet experimentální programy pro neurovědy. Náročnější experimenty je možné programovat pomocí skriptovacího jazyka Python.

Na KIV ZČU je vyvíjen hardwarový programovatelný EEG stimulátor Ing. Jiřím Novotným. Hlavním důvodem vývoje vlastního hardwarového stimulátoru je snadná přenositelnost, nezávislost na dalším počítači a hlavně minimalizace zpoždění reakční doby. Primárně je určen pro měření zvukových a vizuálních ERP experimentů. Díky dobře zvolené architektuře je stimulátor modulární a snadno rozšiřitelný o další experimenty. V příloze B.1 je vidět blokové schéma stimulátoru. Stimulátor se zatím může ovládat pomocí vestavěného dotykového displaye. Další možností je ovládat stimulátor pomocí programu, který napsala Tereza Štanglová[5].

Ke stimulátoru je pomocí UART a GPIO sběrnice zapojeno Raspberry Pi. Na raspberry je spuštěn server, jehož autorem je Milan Hajžman. Na serveru jsou uloženy obrázky a zvuky, které se při spuštěné stimulaci, a pokud to nastavení vyžaduje, zobrazují nebo přehrávají místo blikání LED diod. Dále je server zodpovědný za synchronizaci souborů s mobilní aplikací. Server je napsán v C++ pomocí toolkitu Qt[5] verze 4. Vedle serveru je připraven ke spuštění program sdl_output, který se přímo stará o zobrazení obrázků nebo přehrání zvuků. Tento program se musí spustit ještě před začátkem samotné stimulace. Během stimulace vysílá mikrokontroler stimulátoru TTL[6] signály přes GPIO. Tyto signály zachytává program sdl_output na raspberry pomocí přerušení a na základě vyhodnocení přerušení zobrazí obrázek, nebo přehraje zvuk.

[3]Scenario Description Language
[4]Presentation Control Language
[5]Webové stránky společnosti Qt - https:// www.qt.io/
[6]tranzistorově-tranzistorová logika

2.4 Programování pro platformu Android

Programování aplikace pro platformu Android se velmi liší od programování standartní aplikace v Javě. Největší rozdíl je v neexistenci metody *main*, místo toho se musí vytvořit třída, která bude dědit od třídy **Activity** nebo od jejího potomka, např.: **AppCompatActivity** pro zachování zpětné kompatibility. Pro vývoj aplikace poskytuje platforma Android čtyři základní komponenty:

- Activity - definuje uživatelské rozhraní, obsahuje metody pro interakci s uživatelem
- Service - slouží pro zpracování dlouhotrvajících akcí
- Broadcast receiver - zajišťuje komunikaci mezi operačním systémem a aplikací
- Content providers - slouží pro správu dat a jejich sdílení

Activity

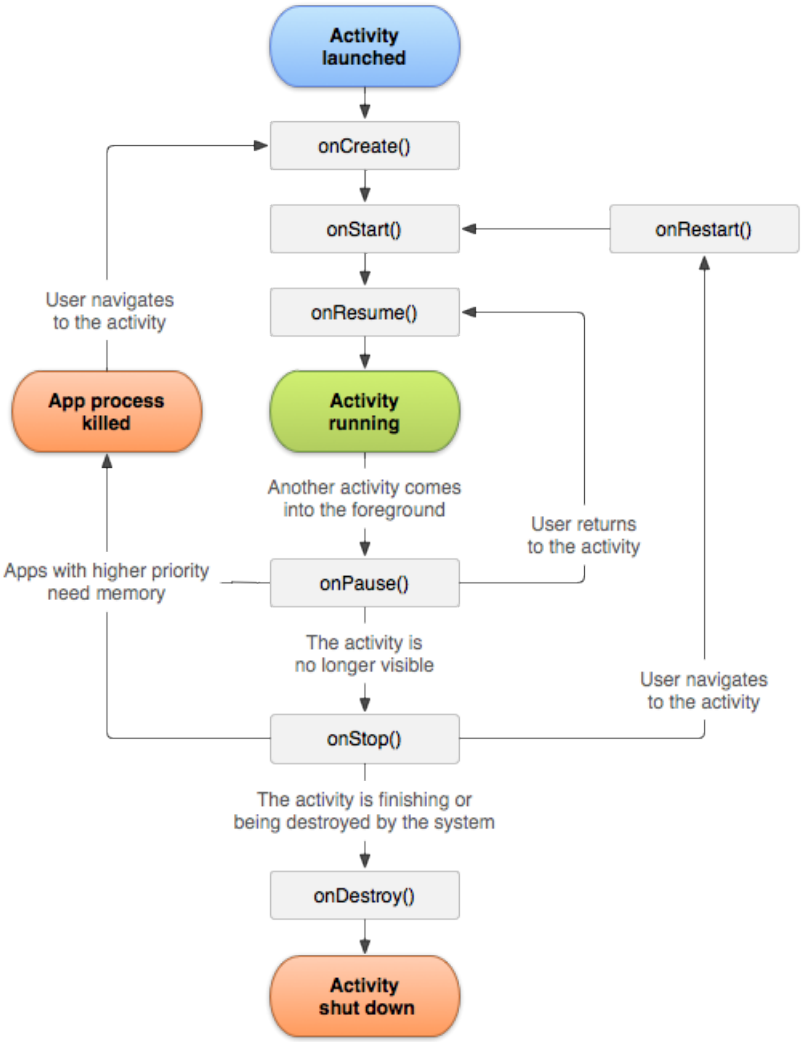
Aktivita je třída, která reprezentuje obrazovku, která se aktuálně zobrazuje uživateli na display. Je zodpovědná za veškerou interakci s uživatelem. V aktivitě se nastavuje, jaký soubor s rozložením obrazovky (layout) se vybere. Layout může pokrývat celou obrazovku, nebo může být ve formě dialogu. Aktivit může být v každé aplikaci více. Všechny aktivity se registrují v souboru *AndroidManifest.xml*. Pokud je v aplikaci více aktivit, tak jedna musí být nastavena jako spouštěcí. V ukázce kódu 2.1 je vidět nastavení *filtru záměru (intent filtru)* pro spouštěcí aktivitu.

```
1 <intent-filter>
2   <action android:name="android.intent.action.MAIN" />
3   <category android:name="android.intent.category.LAUNCHER" />
4 </intent-filter>
```

Listing 2.1: Nastavení filtru záměru (intent filtru)

Každá aktivita má vlastní životní cyklus. Životní cyklus se dá chápat jako stavový automat. Aktivita během své existence prochází různými stavy. Tyto stavy jsou zobrazeny na obrázku 2.4

- *onCreate* - V této metodě se inicializuje aktivita. Pomocí metody *setContentView* se nastaví, jaký se zobrazí layout. Metoda přijímá jako parametr objekt typu **bundle**, který může obsahovat uložená data



Obrázek 2.4: Životní cyklus aktivity[7]

z předchozího běhu. Do bundlu lze uložit například seznam položek, které byly uživatelem označeny.

- *onStart* - Metoda je zavolána po metodě *onCreate*, nebo *onRestart*. V tomto stavu je aktivita viditelná a připravena k použití.
- *onPause* - Aktuální aktivita je pozastavena a překryta jinou aktivitou. V tomto stavu je doporučeno ukládat data do perzistentního úložiště, aby se předešlo jejich ztrátě.
- *onStop* - Aktivita je zastavena a již není viditelná pro uživatele.

Service

Jedná se o služby, které běží v systému na pozadí a starají se o dlouho trvající operace. Služby se používají k I/O operacím, síťové komunikaci (přes internet, bluetooth). Je důležité vědět, že služba při svém vytvoření běží v UI vlákně, je tedy potřeba vytvořit nové vlákno, ve kterém se budou provádět výpočty. Služby obecně nepotřebují komunikovat s uživatelem, nemají definované žádné uživatelské rozhraní a nejsou závislé na životním cyklu aktivity. Každá služba musí být zaregistrována v souboru *AndroidManifext.xml*.

Zjednodušenou verzí Service je třída **IntentService**, která dědí od **Service**. **IntentService** lze použít k jednorázové dlouhotrvající činnosti. Na rozdíl od Service, IntentService si vytvoří vlastní vlákno, ve kterém provádí výpočet. Po skončení výpočtu se sama ukončí.

Broadcast receiver

Broadcast (v případě Androidu) lze chápat jako zprávu, kterou přijmou všichni, kteří jsou zaregistrovaní k odběru.

Broadcast receiver se používá pro komunikaci mezi systémem a aktivitou či službou, nebo aktivitou a službou. Přihlášení k odběru broadcastu se řeší v metodě *onCreate* životního cyklu aktivity případně service. Jeden broadcast receiver může reagovat na více akcí. Akce se nastavuje při registraci receiveru pomocí třídy **IntentFilter**. V následující ukázce kódu 2.2 je vidět registrace broadcast receiveru, který bude reagovat na změnu stavu připojení bluetooth zařízení. Pro registraci více akcí se použije metoda *addAction*, která přijímá v parametru název akce.

Pokud chceme komunikovat pomocí broadcast receiveru pouze uvnitř aplikace, je lepší využít registraci pomocí třídy **LocalBroadcastManager**. Tím se docílí lepšího výkonu a jistoty, že vysílané broadcasty nezachytí žádná cizí aplikace.

```
private final BroadcastReceiver mBluetoothStateReceiver =
  new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
      // Reakce na zpravu
    }
}

@Override
protected void onCreate(Bundle savedInstanceState) {
  // Registrace receiveru
  registerReceiver(mBluetoothStateReceiver,
    new IntentFilter(BluetoothService.ACTION_STATE_CHANGE));
}
```

Listing 2.2: Registrace broadcast receiveru

Content providers

Content provider slouží pro sdílení dat mezi aplikacemi. Content provider definuje abstraktní vrstvu pro správu souborů. Abstraktní vrstvu proto, že nás nezajímá, jak se soubor získá nebo uloží. Pro použití Content provideru je potřeba vytvořit novou třídu a tu oddědit od třídy **ContentProvider**. Dále se musí implementovat metody pro CRUD[7] operace. Tyto metody jsou vidět v ukázce kódu 2.3

```
@Override boolean onCreate() {}
@Override @Nullable Cursor query() {}
@Override @Nullable String getType() {}
@Override @Nullable Uri insert() {}
@Override int delete() {}
@Override int update() {}
```

Listing 2.3: Registrace broadcast receiveru

[7]Create Read Update Delete

2.4.1 Databinding framework

Databinding framework byl představen v květnu v roce 2015 na konferenci Google I/O. Jedná se o framework, který má mimo jiné usnadnit propojení mezi proměnnou deklarovanou v modelové třídě a její reprezentací pomocí uživatelské kontrolky, například *textField*.

Pro základní pochopení problematiky uvedu příklad. Mějme třídu **Foo**, která bude obsahovat proměnnou *bar* typu **String**. Dále budeme chtít, aby tuto proměnnou mohl uživatel libovolně měnit. Pro načtení hodnoty od uživatele použijeme **TextField**. Když bude uživatel psát do textFieldu, tak je potřeba tuto hodnotu ukládat také do proměnné *bar*. Bez Databinding frameworku bychom museli vytvořit **TextWatcher**, který by reagoval na změnu hodnoty v textFieldu a novou hodnotu ručně nastavoval do proměnné *foo*. Tím by se vyřešila propagace změny hodnoty z uživatelské kontrolky do modelu, ale ještě by se musela dopsat propagace v opačném směru, tedy z modelu do kontrolky. Tohle všechno se musí napsat, aby se spárovala jedna proměnná s jednou kontrolkou. Kdyby bylo více kontrolek na jednu proměnnou, tak by kód začal být velmi nepřehledný.

Před Databinding frameworkem řešily tyto problémy knihovny třetích stran, některé se používají dosud. Příkladem těchto frameworků je RoboBinding, nebo Butter Knife.

Pro použití Databinding frameworku je potřeba mít v projektu nastavenou minimální verzi Gradle pluginu na 1.5.0. Dále je potřeba informovat android studio, že se bude používat databinding framework. To se zařídí v konfiguračním souboru *app/build.gradle*, do kterého se přidá následující kus kódu 2.4

```
1 android {
2 ...
3 dataBinding {
4         enabled = true
5     }
6 }
```

Listing 2.4: Aktivace Databinding frameworku

V následujícím kódu 2.5 je vidět implementace třídy **Foo** tak, aby mohla být použita frameworkem.

```
1  public class Foo extends BaseObservable {
2      @Bindable
3      private String bar = "test";
4      public String getBar() {
5          return bar;
6      }
7      public void setBar(String bar) {
8          this.bar = bar;
9          notifyPropertyChanged(BR.bar);
10     }
11 }
```

Listing 2.5: Implementace třídy Foo pro framework

Třída **Foo** musí dědit od třídy **BaseObservable**. Každá proměnná, která má být propojena s uživatelskou kontrolkou, musí mít anotaci **@Bindable**. Tím se zajistí, že framework bude hledat její gettery a settery pro získání a nastavení hodnot. V setteru je důležité po nastavení nové hodnoty zavolat metodu *notifyPropertyChanged*, která jako parametr přijímá ID proměnné, která se změnila. Pokud se metoda nezavolá, tak se změna nepropaguje do uživatelské kontrolky. Třída **BR** je automaticky generovaná třída frameworkem.

Když máme třídu **Foo** implementovanou, můžeme vytvořit jednoduchý layout, který bude obsahovat jeden **textView** a jeden **editText**. Cílem bude, aby se to, co se napíše do editTextu, zobrazilo i v textView. Layout je vidět v kódu C.1 Od standartní definice layoutu se liší hlavně v horní části. Kořenový prvek celého layoutu je párový tag *layout*. Následuje tag **data**, který obsahuje veškeré proměnné, které se budou používat v layoutu. Nakonec následuje standartní definice layoutu. K vytvoření proměnné je potřeba vložit do dat nepárový tag **variable**, který obsahuje název proměnné a typ třídy, která proměnnou reprezentuje. O propojení proměnné *foo* a kontrolky se postará `android:text="@{foo.bar}"` v případe textView a `android:text="@=={foo.bar}"`v případě editTextu. V druhém případě je třeba upozornit na odlišnost od textView. Díky zápisu `"@={foo.bar}"` se budou změny automaticky propagovat do modelu. Tato funkce přibyla až v roce 2016 opět na Google I/O konferenci. Od té doby je Databinding framework dobře použitelný.

Nyní už chybí pouze ukázat, jak inicializovat layout v aktivitě. Místo tradičního volání metody `this.setContentView` se použije `DataBindingUtil.setContentView`. Konkrétní volání je vidět v tomto kódu 2.6

```java
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding binding = DataBindingUtil
          .setContentView(this, R.layout.activity_main);
        binding.setFoo(new Foo());
    }
}
```

Listing 2.6: Inicializace layoutu v aktivitě

Metoda `DataBindingUtil.setContentView` vrací referenci na třídu **ActivityMainBinding**. Tato třída je generovaná frameworkem automaticky. Název třídy vychází z názvu souboru s layoutem, ke kterému se přidá slovo *Binding*. Třída obsahuje všechny kontrolky, které mají definované ID. Tím odpadá nutnost volání metody `findViewById` pokaždé, když je potřeba získat referenci na nějakou kontrolku.

2.5 Testování aplikace

Testování aplikace patří neodmyslitelně k vývojovému cyklu aplikace. Testy se dají rozdělit na:

- Jednotkové testy (Unit testy) - Tyto typy testů testují jednotlivé metody tříd. Metody můžou být různě komplexní, od toho se odvíjí, kolik testů by mělo připadnout na jednu metodu. Komplexnost metody zvyšují konstrukce, jako jsou podmínky (*if*, *switch*) a cykly. Testy by měly být co nejkratší, ideálně na pár řádek kódu.

- Integrační testy - Testy, které testují komunikaci mezi dvěma a více komponentami. Komponentu lze chápat například jako jednu aktivitu, nebo službu. Testuje se tedy komunikace mezi dvěma aktivitami, nebo aktivitou a službou. Tyto testy se většinou píšou až po jednotkových testech, protože kdyby jednotlivé komponenty nebyly dostatečně otestované, tak integrační testy selžou, nebo budou chybné.

- Systémové testy - Během těchto testů se testuje aplikace jako celek. Vytváří se scénáře, pomocí kterých se při testu postupuje. Testování může probíhat manuálně, za pomocí člověka, nebo automaticky s pomocí vybraného nástroje.

2.5.1 Jednotkové testy v jazyce Java

Hlavním představitelem jednotkových testů v jazyce Java je JUnit[1]. Jedná o framework, který obsahuje nástroje k testování. Aktuální verze frameworku je 4.12. Testy se píší do tříd. Každy test představuje jedna metoda, která obsahuje anotaci *@Test*. JUnit nabízí před spuštěním každého testu možnost inicializovat potřebné proměnné. Pro inicializaci se vytvoří metoda, obvykle pojmenovaná jako *setUp* s anotací *@Before*. Pro úklid po testu lze vytvořit metodu, obvykle pojmenovanou jako *tearDown* s anotací *@After*.

V Unit testech by se neměly vyskytovat žádné komplexní konstrukce, jako jsou podmínky, cykly a try catch bloky. Pokud se testuje metoda, která obsahuje podmínku, tak se vytvoří tolik testů, kolik je větví v podmínce. Pro zachycení výjimky z testované metody lze použít parametr v anotaci *@Test*: `@Test(expected = IllegalArgumentException.class)`.

V případě, že máme proměnnou, která může nabývat pouze hodnot ze zadaného intervalu (např.: <-10; 10>), není dobré vytvářet pro každou hodnotu nový test. Obvykle je potřeba zkontrolovat hranice intervalu a pár hodnot uvnitř intervalu. Kdybychom psali pro každou hodnotu samostatný test, tak bychom se upsali. Místo toho lze použít tzv. parametrizovaný test.

Parametrizovaný test, jak už název napovídá, využívá parametrů, které se dosazují do testovací metody. K použití parametrizovaného testu je potřeba, aby třída splňovala následující požadavky:

1. Třída musí mít následující anotaci: `@RunWith(Parameterized.class)`.
2. Ve třídě musí existovat veřejná statická metoda s anotací `@Parameters`, která bude vracet kolekci dvourozměrného pole objektů, které představují parametry testu.
3. Nastavení parametrů pro jeden test lze udělat dvěma způsoby:
   - Parametry se předají pomocí konstruktoru třídy. Proměnné musí být ve správném pořadí.
   - Vytvoří se veřejné proměnné, které budou mít anotaci `@Parameterized.Parameter()`, která jako parametr bere pořadí parametru (značené od 1)
4. Třída musí obsahovat testovací metodu s anotaci `@Test`, ve které se používají parametry k testování.

V následujícím příkladu kódu 2.7 je vidět metoda s anotací `@Parameters` pro vytvoření parametrů testu.

```
@Parameterized.Parameters(name = "{index}: Output - Brightness↩
    value: {0}, isValid: {1}")
public static Collection<Object[]> outValues() {
    return Arrays.asList(new Object[][]{
            {AConfiguration.MIN_BRIGHTNESS - 1, false},
            {AConfiguration.MIN_BRIGHTNESS, true},
            {AConfiguration.MIN_BRIGHTNESS + 1, true},
            {(AConfiguration.MAX_BRIGHTNESS + AConfiguration.↩
               MIN_BRIGHTNESS) / 2, true},
            {AConfiguration.MAX_BRIGHTNESS - 1, true},
            {AConfiguration.MAX_BRIGHTNESS, true},
            {AConfiguration.MAX_BRIGHTNESS + 1, false},
    });
}
```

Listing 2.7: Příklad metody na vytvoření parametrů pro parametrizovaný test

Pro lepší výpis logů u testů lze předat jako parametr anotaci `@Parameters`↩ logovací hlášku, která se bude vypisovat u každého testu.

2.5.2 Systémové testy pro Android

Při vzniku platformy Android žádné frameworky na testování uživatelského rozhraní neexistovaly. Postupem času rostla potřeba testovat i uživatelské rozhraní, protože Android běžel na nejrůznějších zařízeních v různých konfiguracích. Na to reagovala komunita a v roce 2010 vznikl projekt Robotium[8]. Projekt si velmi rychle vybudoval dobré jméno a dodnes se mu velmi dobře daří. Hlavní problém frameworku byl v synchronizaci. Během testů se tedy muselo uměle čekat, až se akce v androidu vykoná. To zvyšovalo celkovou dobu testů. V říjnu roku 2013 Google zveřejnil vlastní framework pro testování uživatelského rozhraní pod názvem Espresso.

Espresso framework stojí na třech základních pilířích:

1. ViewMatchers - Kolekce objektů, které implementují `Matcher<? super`↩ `View>` rozhraní. Slouží k vyhledání konkrétního prvku v aktuálním stromu prvků. Můžou se do sebe libovolně zanořovat.

2. ViewActions - Kolekce příkazů, které lze na jednotlivých prvcích vykonat. Jedná se například o: krátké či dlouhé kliknutí, posun do různých směrů (swipe) a další.

[8]Webové stránky projektu Robotium - https://robotium.com

3. ViewAssertions - Kolekce příkazů, pomocí kterých se kontroluje, zda-li vybraný prvek splňuje požadované vlastnosti, například: je viditelný, klikatelný

K použití základního testu s využitím Espresso frameworku jsou potřeba udělat dvě věci:

1. Třída, která obsahuje testy musí mít anotaci: `@RunWith(AndroidJUnit4.class)`. Často se ještě přidává anotace `@LargeTest`, aby se označilo, že test může trvat dlouhou dobu.

2. Ve třídě musí být veřejná proměnná typu **ActivityTestRule** s anotací `@Rule`. Toto pravidlo zajistí, že před každým spuštěním testu, se zobrazí nová aktivita připravená k použití. Během testu lze s aktivitou manipulovat.

V ukázce kódu 2.8 je vidět příklad testu, který zjišťuje, zda-li je zobrazen vybraný text.

```
onView(
        withId(R.id.textNoConfigurationFound))
        .check(matches(isDisplayed()))
        .check(matches(withText(
          R.string.no_configuration_found)));
```

Listing 2.8: Ukázka použití view asserce pro otestování, zda-li je zobrazen vybraný test

V další ukázce kódu 2.9 je vidět využití view akce *kliknutí*.

```
onView(
        allOf(
                withId(R.id.fab_new_configuration),
                isDisplayed()))
        .perform(click());
```

Listing 2.9: Ukázka použití view akce kliknutí

3 Realizační část

Realizační část se věnuje implementaci aplikace pro vzdálené ovládání stimulátoru a komunikaci se stimulátorem.

3.1 Použité technologie

V této sekci proberu zařízení a technologie, které jsem použil během vývoje a testování.

Jako primární zařízení, na kterém probíhal vývoj, byl telefon značky Nexus 5, který je velmi vhodný pro vývoj aplikací zejména proto, že obsahuje v základu čistý android bez žádné nadstavby. Je tedy velká pravděpodobnost, že aplikace bude fungovat na ostatních zařízeních. Cílové zařízení, na kterém aplikace bude provozována, je tablet, který je v laboratoři spolu se stimulátorem. Tablet běží na systému Android ve verzi 4.1.2, což mimo jiné definuje nejnižší verzi systému Android, kterou bude aplikace podporovat.

K programování Android aplikací se nejčastěji používá programovací jazyk Java verze 7. Jako vývojový program se dříve používal Eclipse[1] s pluginem ADT (android development tools). Dnes se již od Eclipse upustilo a Google vyvíjí vlastní vývojové prostředí Android studio [2] postavené na IntelliJ[3], což je vývojové prostředí od české firmy Jetbrains [4]. Dále je potřeba Android SDK[5], který obsahuje veškeré potřebné knihovny pro vývoj aplikace.

Při testování byly použity dvě primární technologie: JUnit[6] pro testování jednotlivých metod a Espresso [7] pro otestování uživatelského rozhraní.

[1] Webové stránky projektu Eclipse - https://www.eclipse.org/org/

[2] Vývojové studio pro Android

[3] Vývojové studio pro jazyk Java

[4] Webové stránky firmy Jetbrains - https://www.jetbrains.com/

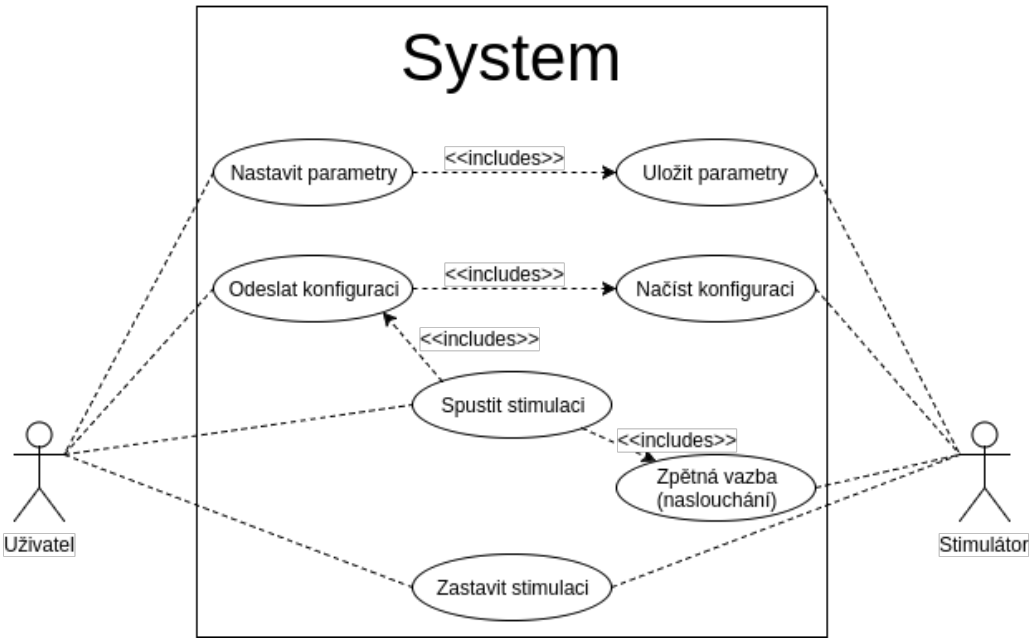
[5] Software development toolkit

[6] Webové strány projektu JUnit - http://junit.org/junit4/

[7] Webové stránky projektu Espresso - https://google.github.io/android- testing-support-library/docs/espresso/index.html

3.2 Požadavky na aplikaci

Základní požadavky jsou zobrazeny v usecase diagramu 3.1. Z diagramu je vidět, že uživatel by měl být schopen spravovat jednotlivé experimenty, tedy vytvářet nové, mazat a upravovat. Do úprav se zahrnuje nastavování parametrů jednotlivých experimentů. Tyto experimenty se budou ukládat do perzistentního úložiště zařízení. Samozřejmostí je také načítání z úložiště. Dále je vidět požadavek na přenos parametrů experimentu mezi aplikací a stimulátorem. Pomocí aplikace se bude spouštět a zastavovat experiment. Pokud se bude jednat o experiment se zpětnou vazbou, je potřeba zajistit, aby aplikace dokázala reagovat i na zpětnou vazbu a tu vizualizovat.



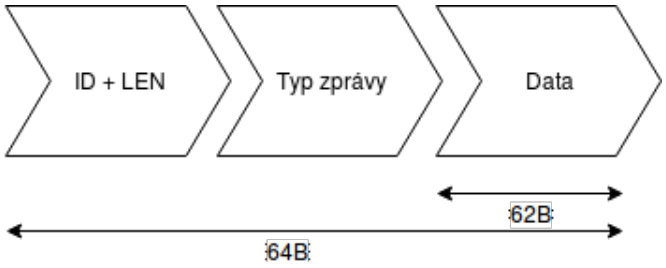
Obrázek 3.1: Usecase diagram

Aplikace by měla nahradit stávající rozhraní pro nastavování parametrů. Aktuálně se parametry jednotlivých experimentů nastavují na dotykové obrazovce, která je přímo na stimulátoru. Z hlediska uživatelského rozhraní je tedy požadavek na zachování, nebo alespoň napodobení rozhraní, které je nyní použito. Na obrázku 3.2 je ukázka aktuálního rozhraní, které je použito na display stimulátoru.

Obrázek 3.2: zleva nahoře: příklad hlavního menu ve stimulátoru, příklad obrazovky pro nastavení parametrů ERP experimentu, obrazovka pro výběr BCI experimentu, následují obrazovky pro konkrétní BCI experiment f-VEP, t-VEP, c-VEP

3.3 Komunikační protokol

Pro komunikaci se stimulátorem byl navržen jednoduchý komunikační protokol. Každý příkaz, který se bude posílat do stimulátoru, je zabalen do Packetu. Struktura packetu je vidět na obrázku 3.3.



Obrázek 3.3: Struktura packetu pro komunikaci se stimulátorem

Tento packet se skládá ze tří částí:

1. Hlavička - je obsažena v prvním bytu packetu. Obsahuje ID a počet datových bytů.
2. Typ zprávy - nachází se ve druhém bytu packetu. Obsahuje unikátní konstantu pro rozlišení, o jakou zprávu se jedná.
3. Data - vlastní data.

Celý packet má pevnou délku 64B, z čehož vyplývá, že na data zbývá maximálně 62B. Popis celého komunikačního protokolu pro stimulátor je k dispozici v příloze stim_protocol.pdf. Příkazy stimulátoru se dají rozdělit na dvě skupiny:

- Přikazy bez dat - Tyto příkazy mají prázdnou datovou část packetu. Slouží ke změně vnitřního stavu stimulátoru, například spuštění/zastavení stimulace.
- Příkazy s daty - Pomocí příkazů s daty se nastavují parametry stimulátoru.

Nyní uvedu menší výčet packetů pro lepší představu.

- Spuštění stimulace - {0x00, 0x01, zbytek packetu je vyplněn 0x00}
- Zastavení stimulace - {0x00, 0x02, zbytek packetu je vyplněn 0x00}
- Doba svícení LED0 (30ms) - {0x02, 0x10, 0x01, 0x2C}

- Pauza LED0 (15ms) - {0x02, 0x11, 0x00, 0x96}

Během vývoje aplikace se ukázalo, že je potřeba synchronizovat různé obrázky a zvuky (použité během stimulace) mezi mobilní aplikací a serverem, na kterém jsou obrázky uloženy. Pro synchronizaci se musel vyvinout nový protokol, který pouze rozšiřuje stávající protokol tak, aby stimulátor nic nepoznal.

Pro komunikaci s Raspberry Pi se používá standartní packet délky 64B. Všechny packety mají stejné první dva byty. V prvním bytu se nastaví, že data zaberou celou délku packetu, tedy 62B. Ve druhém bytu se nastaví speciální typ zprávy, pomocí kterého stimulátor pozná, že data má přeposlat dál. Tato zpráva má hodnotu: 0xBF. Zbylých 62B tedy zbývá na data. Do těchto dat se zabalí nový packet, kterému rozumí program na Raspberry. Packet je opět rozdělen, tentokrát na hlavičku, iterátor a data. Do hlavičky se nastavuje, jaký příkaz se má vykonat. Iterátor obsahuje unikátní identifikátor packetu pro jednu akci. Na surová data, tedy zbývá pouze 60B. Technika zapouzdřování packetů se odborně nazývá encapsulation.

3.4 Implementace jednotlivých částí

Implementaci lze rozdělit na pět hlavních částí:

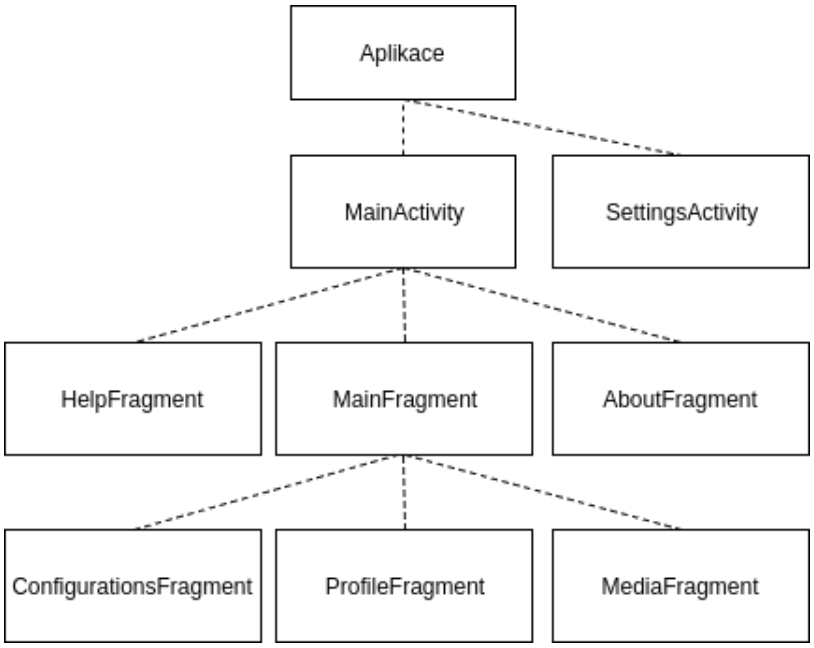
1. Implementace hlavní aktivity
2. Nastavování parametrů jednotlivým experimentům
3. Správa profilů jednotlivých výstupů/stimulů
4. Správa obrázků a zvuků
5. Ovládání stimulátoru

Vizualizace rozdělení aplikace je vidět na obrázku 3.4

Na následujících řádcích proberu podrobně implementaci každé části aplikace.

3.4.1 Implementace hlavní aktivity

Hlavní aktivita je deklarována pomocí souboru *main_activity.xml* a je reprezentována třídou **MainActivity**. Jedná se o srdce celé aplikace. V této aktivitě jsou definovány základní ovládací prvky aplikace. Jedná se o toolbar obsahující navigační menu a správu spojení přes bluetooth a *FrameLayout*, do kterého se vkládají jednotlivé fragmenty.

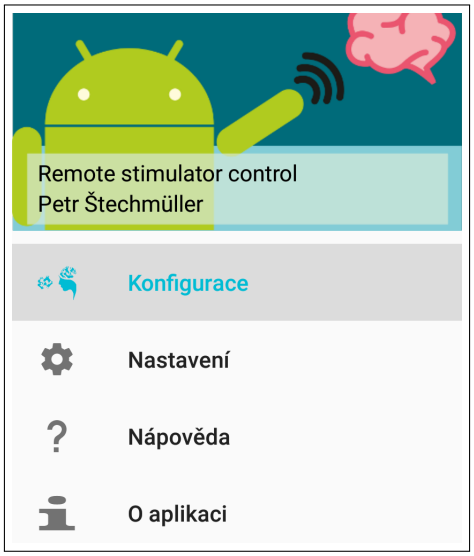


Obrázek 3.4: Rozdělení aplikace

Toolbar je na obrázku 3.5. Z toolbaru je možné zobrazit postranní navigační menu. Dále je zobrazen název aktuálně vybraného fragmentu. V podnadpisu je vidět aktuální stav spojení přes bluetooth zařízení. Pokud zařízení není připojeno, je zobrazen nápis *nepřipojeno*. V pravé části toolbaru je tlačítko pro připojení se k zařízení přes bluetooth. Po stisknutí tohoto tlačítka se zobrazí nabídka všech spárovaných zařízení. Po vybrání zařízení se aplikace pokusí připojit. Když se podaří spojení navázat, zobrazí se v podnadpisu aplikace: *připojeno k název_zařízení*. Při neúspěchu aplikace informuje uživatele o neúspěšném pokusu tzv. Toastem, což je způsob, jak informovat uživatele o nějaké akci.

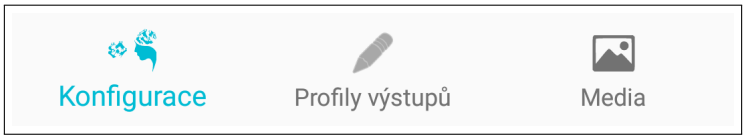
Obrázek 3.5: Toolbar

Pomocí navigačního menu lze přepínat mezi jednotlivými fragmenty, kromě položky nastavení, která otevře novou aktivitu. Navigační menu je na obrázku 3.6.



Obrázek 3.6: Navigační menu

Nejdůležitější je první položka, pomocí které se zobrazí fragment pro přístup k jednotlivým konfiguracím, profilům a médiím. Tento fragment je definován pomocí souboru *main_fragment.xml* a je reprezentován třídou **MainFragment**. Jedná se pouze o mezikrok. Ve fragmentu jsou pouze dva prvky: *FrameLayout* a *BottomNavigationBar*. Ve FrameLayoutu se zobrazí fragment, který se vybere pomocí spodní navigační lišty. Navigační lišta je na obrázku 3.7

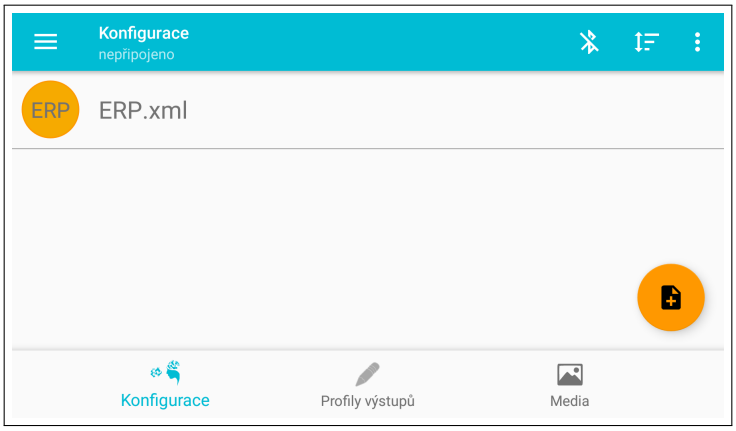


Obrázek 3.7: Spodní navigační lišta

Každá položka v navigační liště představuje jeden frame layout.

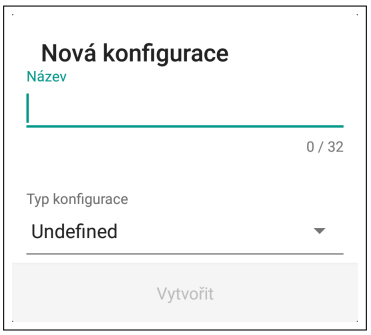
Záložka *konfigurace* obsahuje jeden **RecyclerView**, ve kterém se zobrazují jednotlivé konfigurace a tlačítko pro vytvoření nové konfigurace. Dále je rozšířena nabídka v toolbaru o možnost tyto konfigurace řadit podle názvu a typu konfigurace a přibyla možnost importovat existující konfiguraci

z úložiště. Na obrázku 3.8 je vidět rozložení layoutu. Pokud nebude nalezena žádná konfigurace, místo recyclerView se zobrazí hláška informující, že nebyly nalezeny žádné konfigurace.



Obrázek 3.8: Rozložení layoutu pro správu konfigurací

Tlačítko pro vytvoření nové konfigurace se nachází ve spodní části obrazovky. Po kliknutí na tlačítko se zobrazí dialogové okno 3.9, které je definované v souboru *activity_configuration_factory.xml* a je reprezentováno třídou **ConfigurationFactoryActivity**, do kterého se vyplní název a typ konfigurace. Tlačítkem *vytvořit* se vytvoří nová konfigurace experimentu, která má všechny parametry nastavené na svoje výchozí hodnoty.



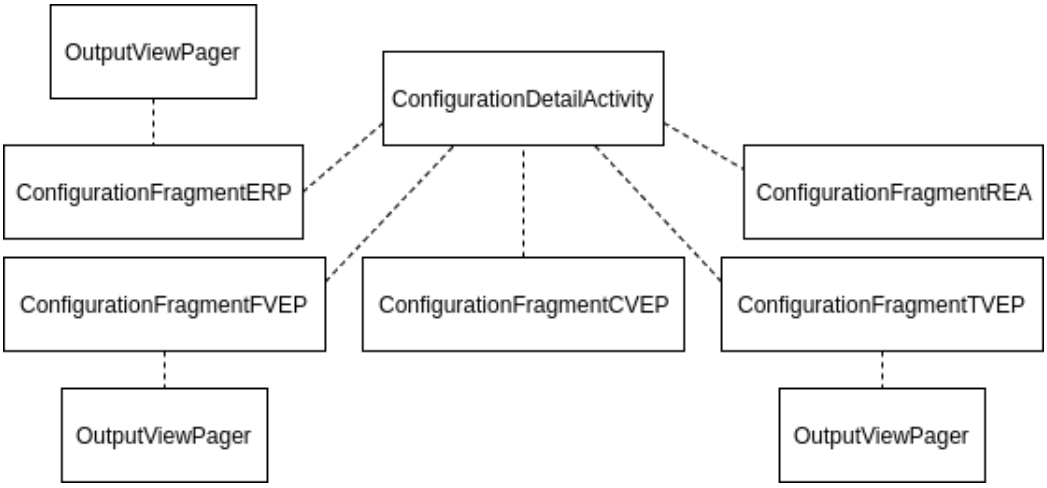
Obrázek 3.9: Dialogové okno pro novou konfiguraci

Jednotlivé položky představující konfigurace v recyclerView jsou popsány pomocí souboru *configuration_item.xml*. Každá položka obsahuje název s volitelně zobrazitelnou koncovkou souboru a vlevo od názvu je barevné rozlišení podle typu konfigurace.

3.4.2 Nastavování parametrů jednotlivým experimentům

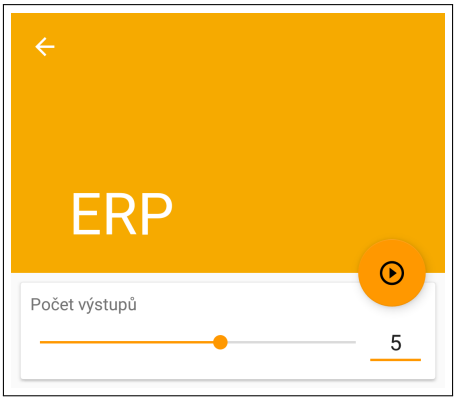
Po kliknutí na vybranou konfiguraci se zobrazí nová aktivita, ve které se nastavují jednotlivé parametry podle konfigurace. Tato aktivita je definována v souboru *activity_configuration_detail.xml* a reprezentována třídou **ConfigurationDetailActivity**

Rozložení fragmentů podle typu konfigurace je znázorněno na obrázku 3.10.



Obrázek 3.10: Rozložení fragmentů podle typu experimentu

V horní části aktivity (obrázek 3.11) se nastavují parametry, které mají všechny experimenty společné.

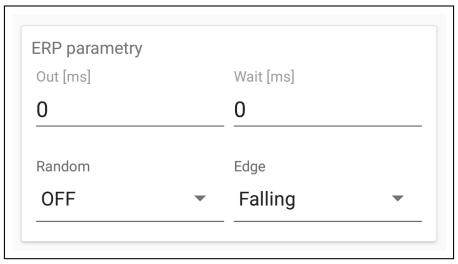


Obrázek 3.11: Společné nastavení parametrů pro všechny experimenty

Jedná se o počet výstupů/stimulů, na které bude člověk reagovat. Tyto stimuly se rozdělují na tři druhy:

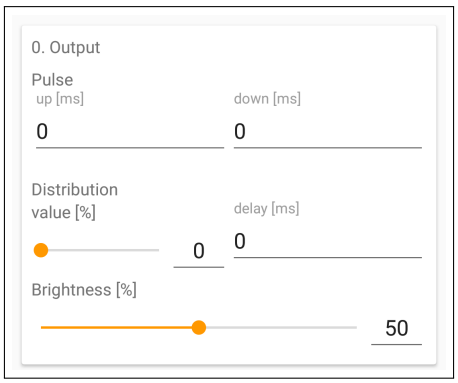
1. LED - stimuly budou představovat LED diody umístěné před subjektem.
2. Obrázkové - stimuly se budou reprezentovat ve formě obrázků, které subjekt uvidí před sebou na monitoru.
3. Zvukové - stimuly se budou reprezentovat ve formě zvuků, které subjekt uslyší.

Na obrázku 3.12 je příklad nastavení parametrů pro experiment typu ERP.



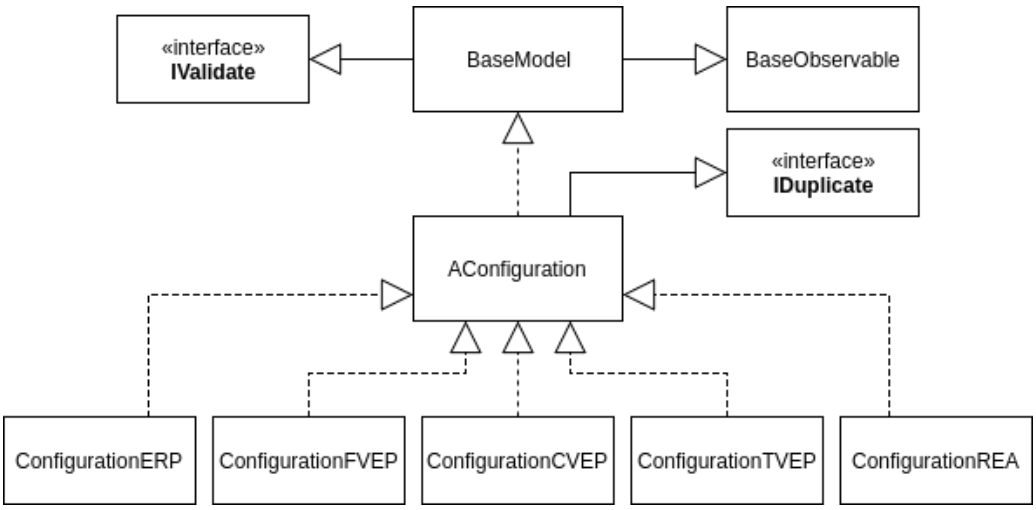
Obrázek 3.12: Parametry pro experiment typu ERP

Některé experimenty ještě vyžadují dodatečné nastavení parametrů jednotlivých výstupů/stimulů. Mezi tyto experimenty patří ERP a f-VEP. Na obrázku 3.13 je příklad nastavení parametrů výstupů/stimulů pro konfiguraci typu ERP.



Obrázek 3.13: Parametry pro jeden výstup/stimul konfigurace ERP

Každá konfigurace experimentu je definována ve vlastní třídě. Všechny konfigurace dědí od jedné základní konfigurace **AConfiguration**, která obsahuje společné vlastnosti konfigurací. Třída **AConfiguration** dědí od třídy **BaseModel**, která implementuje metody pro kontrolu validity obsahu objektu. Na obrázku 3.14 je vidět UML diagram hierarchie tříd pro aktuální verzi aplikace. Je implementováno 5 různých experimentů.



Obrázek 3.14: Hierarchie dědičnosti tříd konfigurací experimentů

U implementace třídy **BaseModel** bych se rád pozastavil. Při nastavování jednotlivých parametrů konfigurace experimentu bylo potřeba validovat vstupní hodnoty, jestli jsou v souladu s definovanými pravidly. Číselné parametry jsou validní, pokud spadají do definovaného intervalu. Pro textové parametry, např.: název konfigurace, se kontroluje, zda-li odpovídají definovanému patternu složeného z regulárního výrazu. Třída **BaseModel** obsahuje tři důležité proměnné pro kontrolu validity modelu. Tyto proměnné jsou vidět v kódu 3.1.

```
1 @Bindable
2 protected int validityFlag = 0;
3 @Bindable
4 protected boolean valid = true;
5 @Bindable
6 protected boolean changed = false;
```

Listing 3.1: Proměnné ve třídě BaseModel

Proměnná *changed* říká, jestli byla nějaká hodnota modelu změněna od svého vzniku. Tato proměnná hraje důležitou roli hlavně v dialogu pro vytvoření nové konfigurace. V dialogu se kontroluje, zda-li je zadaný název validní. Název je validní právě tehdy, když není prázdný a splňuje zadaný pattern. Pokud by proměnná *changed* neexistovala, tak by dialog hlásil, že zadaný název není validní ještě před tím, než by uživatel stačil cokoliv zadat.

Proměnná *valid* představuje validitu všech hodnot v modelu. Pokud alespoň jedna hodnota není validní, tak tato proměnná bude mít hodnotu false. Použije se opět v dialozích, tentokrát ve všech, co v aplikaci jsou. Pokud není nějaký parametr v dialogu validní, tak nelze požadovanou akci splnit, to je dosaženo zneaktivněním potvrzovacího tlačítka.

Nejvýznamnější proměnná je *validityFlag*. Zde se využívá toho, že celočíselný typ *int* je reprezentován pomocí bitů. Všechny moderní telefony používají 32 bitové procesory a ty novější jsou už 64 bitové. To znamená, že do jedné proměnné lze uložit až 32 případně 64 stavů.

Pro nastavování jednotlivých bitů se využívají bitové operace. Pro pohodlnější práci s bitovými operacemi jsem vytvořil knihovní třídu **BitUtils**, která obsahuje metody pro nastavování, mazání a kontrolu jednotlivých bitů. Nejdůležitější je metoda *setBit*, která je vidět v kódu 3.2

```
1  public static int setBit(int original, int flag,
2    boolean value) {
3      if (value) {
4          original |= flag;
5      } else {
6          original &= ~flag;
7      }
8
9      return original;
10 }
```

Listing 3.2: Metoda *setBit*

Metoda přijímá tři parametry:

1. *original* - hodnota, která obsahuje validitu jednotlivých bitů
2. *flag* - index určující, se kterým bitem se bude pracovat. Nabývá hodnot mocniny 2
3. *value* - pokud je hodnota true, se nastaví na danou pozici 1, jinak 0

Tabulka 3.1: Bitový součin

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Tabulka 3.2: Bitový součin

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Tabulka 3.3: Bitová negace

A	B
0	1
1	0

Nastavení 1 na pozici se provádí pomocí bitové operace *OR*[8]. Pro vynulování je použita bitová operace *AND*[9] s inverzní hodnotou indexu. V tabulkách 3.1, 3.2, 3.3 jsou vidět tabulky pro bitové operace OR, AND a NOT[10].

Třída **BaseModel** dále obsahuje metodu *setValidityFlag* 3.3, pomocí které se nastavuje validita jednotlivých proměnných v modelu. Pomocí metody *setBit* se nastaví na určenou pozici 1, nebo 0. Pokud se proměnná *validityFlag* změnila, tak se pošle zpráva, aby si ostatní přečetli novou hodnotu. Proměnná *changed* se nastaví na *true*, protože se provedla změna nějaké proměnné. Nakonec se testuje, jestli je v proměnné *validityFlag* hodnota 0. Pokud obsahuje 0, tak to znamená, že všechny proměnné v modelu jsou validní a tak se nastaví proměnná *valid* na hodnotu true.

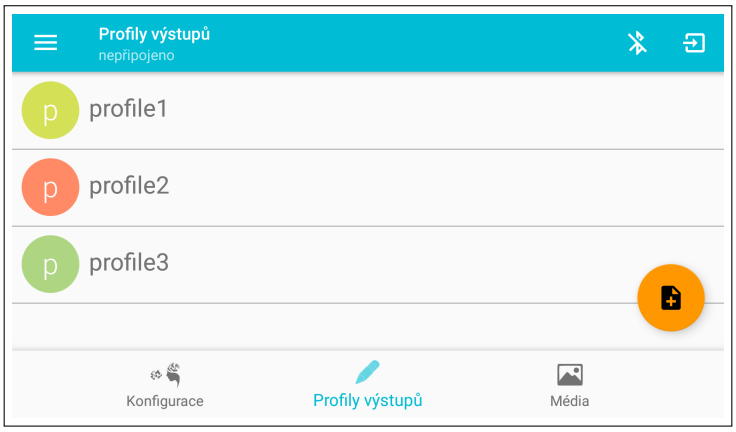
```
1  protected void setValidityFlag(int flag, boolean value) {
2      int oldFlagValue = validityFlag;
3      validityFlag = BitUtils.setBit(validityFlag, flag, value);
4      if (validityFlag == oldFlagValue) {
5          return;
6      }
7      notifyPropertyChanged(BR.validityFlag);
8      setChanged(true);
9      if (validityFlag == 0) {
10         setValid(true);
11     }
12 }
```

Listing 3.3: Metoda *setValidityFlag*

[8]Bitový součet
[9]Bitový součin
[10]Bitová negace

3.4.3 Správa profilů jednotlivých výstupů/stimulů

Pod druhou položkou ve spodním navigačním menu 3.7 se ukrývá správa profilů výstupů/stimulů. Rozložení layoutu je identické s layoutem pro konfigurace, je definováno v souboru *fragment_profile.xml* a reprezentováno třídou **ProfileFragment**. Jedná se tedy o recyclerView, které obsahuje jednotlivé profily a tlačítko pro vytvoření nového profilu. Jednotlivé položky v recyclerView jsou definovány v souboru *profile_item.xml*. Na obrázku 3.15 je vidět rozložení layoutu.



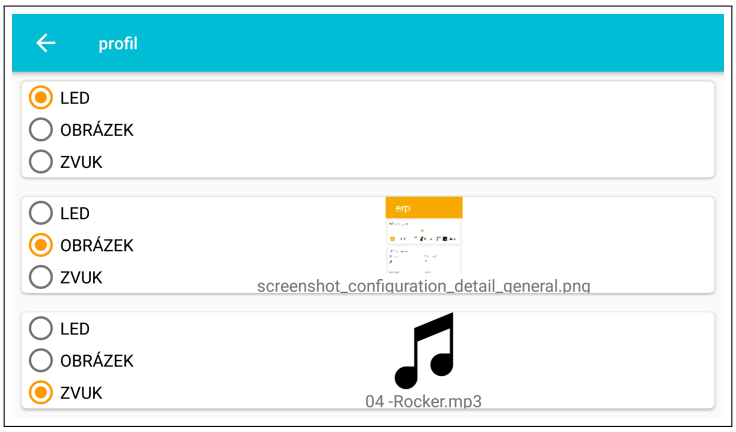
Obrázek 3.15: Rozložení layoutu pro správu profilů

Pokud nebude žádný profil nalezen, místo recyclerView se zobrazí hláška informující, že nebyl nalezen žádný profil. Nabídka toolbaru je v případě profilů rozšířena pouze o tlačítko pro import existujícího profilu z úložiště.

Tlačítko pro vytvoření nového profilu funguje stejně jako v případě konfigurací. Opět se zobrazí dialog tentokrát pouze s výzvou zadat název profilu. Tlačítkem vytvořit se vytvoří nový profil.

Po kliknutí na vybraný profil se zobrazí nová aktivita představující detail profilu, ve které se nastavují jednotlivé výstupy/stimuly. Aktivita je definována v souboru *activity_profile_detail.xml* a reprezentována třídou **ProfileDetailActivity**. Na obrázku 3.16 je vidět rozložení layoutu pro nastavení jednoho profilu.

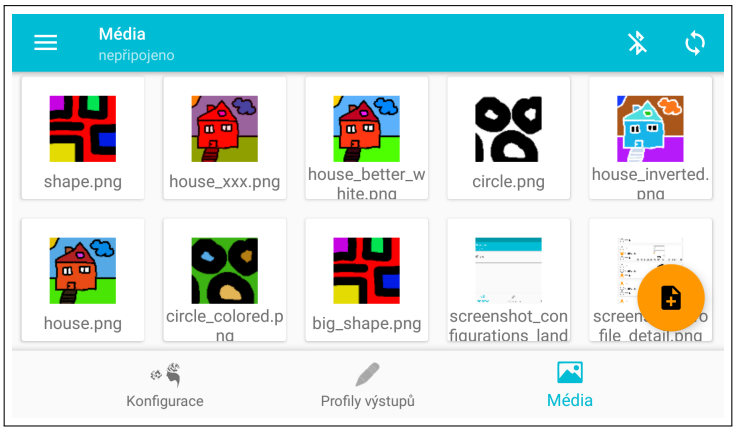
Layout obsahuje celkem 8 stejných položek, kde jedna položka reprezentuje jeden výstup/stimul. Každá položka je definovaná v souboru *output_row.xml* a je reprezentována třídou **ProfileConfigurationWidget**.



Obrázek 3.16: Nastavení profilu

3.4.4 Správa obrázků a zvuků

Poslední položka ve spodním navigačním menu 3.7 je pro správu obrázků a zvuků. Layout se opět skládá z recyclerView pro zobrazení obrázků a zvuků a tlačítka pro import média. Layout je definován v souboru *fragment_media.xml* a reprezentován třídou **MediaFragment**. Jednotlivé položky v recyclerView jsou definovány v souboru *media_item.xml*. Na obrázku 3.17 je vidět rozložení layoutu.



Obrázek 3.17: Rozložení layoutu pro správu obrázků a zvuků

Do toolbaru přibylo pouze tlačítko pro synchronizaci médií se serverem, který běží v Raspberry Pi. Když začne synchronizační proces, zobrazí se dialog informující uživatele o aktuálním stavu.

Když se klikne na položku s obrázkem, zobrazí se dialog s jeho náhledem. Pokud se klikne na položku se zvukem, tak se tento zvuk přehraje. Během přehrávání zvuku se u vybrané položky zobrazí ikona, že se zvuk přehrává.

3.4.5 Ovládání stimulátoru

Pro samotné ovládání stimulátoru slouží aktivita **StimulatorControl**. Aktivita obsahuje tři tlačítka:

- Spuštění stimulace - pošle do stimulátoru příkaz ke spuštění stimulace
- Zastavení stimulace - pošle příkaz do stimulátoru pro zastavení stimulace
- Nahrání vybrané konfigurace - nahraje vybranou konfiguraci experimentu do stimulátoru

3.4.6 Služby aplikace

Na pozadí aplikace běží služba **BluetoothService**, která se stará o spojení s bluetooth zařízením. Služba může během svého životního cyklu nabývat tří stavů:

1. *disconnect* - není vytvořeno žádné spojení
2. *connecting* - když přijde požadavek na připojení k novému zařízení
3. *connected* - spojení je vytvořeno

Komunikace se službou je zajištěna pomocí broadcast receiverů. Služba má zaregistrované dva broadcast receivery:

- StatusReceiver - jedná se o receiver, pomocí kterého se nastavuje, zdali se má služba připojit k zařízení nebo odpojit
- SenderReceiver - pomocí tohoto receiveru se odesílají data ven z mobilu do připojeného zařízení

Další věc, o kterou se služba musí starat je příjem dat. Při vytvoření stabilního spojení se spustí samostatné vlákno *ConnectedThread*, ve kterém se vytvoří **InputStream** a **OutputStream**. Pomocí těchto streamů se posílají data dovnitř a ven z aplikace. Jak jsem již psal v teorii, komunikace probíhá pomocí packetů o velikosti 64B. InputStream může přečíst pokaždé jiný počet bytů, takže bylo potřeba vymyslet, jak získat 64B. InputStream má metodu *read*, která v parametru přijímá referenci na pole bytů, do kterého může vložit přečtené byty. Je zajištěno, že nepřečte víc, než je velikost pole, ale může přečíst méně. Návratová hodnota metody read je přesný počet bytů, které přečetla. Tohoto faktu jsem využil a za pomoci dvou polí bytů o velikosti 64B a jednoduché modulové aritmetiky skládám přijaté byty až do velikosti packetu, který pomocí broadcastu rozešlu do aplikace. Celý kód přijímání dat a skládání packetu je vidět v ukázce kódu C.2.

V proměnné *data* se vytváří celý packet. Do proměnné *tempBuffer* se ukládají přijaté byty z input streamu. Z počtu přijatých bytů se určí, kolik se jich ještě vejde do naplnění celého packetu. Do proměnné *data* se nakopírují příslušné byty. Pokud už je v datech 64B, tak se tyto byty odešlou do aplikace. Protože se v podmínce kontroluje: `if (totalSize >= BtPacket.PACKET_SIZE)`, tedy, jestli je počet celkem přijatých bytů větší nebo roven velikosti jednoho packetu, je potřeba nakopírovat byty, které se nevešly do packetu do nových dat pro další packet.

FileSynchronizerService, FileLsService, FileUploadService, FileDownloadService, FileDeleteService

O synchronizaci souborů se vzdáleným serverem se stará služba **FileSynchronizerService** reprezentovaná stejnojmenou třídou. Nejedná se o obyčejnou službu, ale o **IntentService**, která, jak jsem psal v teorii, si vytvoří vlastní pracovní vlákno a po skončení úkolu zase zanikne. **FileSynchronizerService** se stará o postupné spouštění dílčích služeb a z jejich výsledků se rozhoduje, co bude následovat. Synchronizace souborů se skládá ze čtyř kroků:

1. Získání informací o souborech uložených na serveru. O získání informací se stará služba **FileLsService**.

2. Zpracování lokálních a vzdálených souborů. V této části se rozhodne, které soubory je potřeba stáhnout a které je potřeba nahrát na server. Rozhodování probíhá v metodě *mergeFiles*.

3. Nahrání potřebných souborů na server. O nahrání jednoho souboru se stará služba **FileUploadService**.

4. Stažení potřebných souborů na server. Stažení jednoho souboru má na starosti služba **FileDownloadService**.

Služba **FileDeleteService** se zavolá pouze v případě, když uživatel smaže soubor s obrázkem nebo se zvukem z telefonu.

Když nějaká z výše uvedených služeb dokončí svoji práci, tak informuje svého rodiče (služba, která ji spustila) o výsledku operace. Například služba **FileLsService** musí po získání informací informovat službu **FileSynchronizerService** o výsledku. Tím přichází na řadu další problém: jak zajistit, aby volající služba počkala na výsledek. Odpověď se nachází v použití semaforu. Jedná se synchronizační primitivum, které obsahuje celé nezáporné číslo. Dále je možné nad tímto primitivem volat dvě operace:

- P(S[11]) - Pokud S > 0, sníží S o 1, jinak zastaví proces.
- V(S) - Pokud je nad semaforem S zablokovaný jeden nebo více procesů, vzbudí jeden z nich (náhodně), jinak zvýší S o 1.

Operace P a V jsou atomické (nedělitelné), to znamená, že jakmile jedno vlákno zavolá P nebo V, další vlákno musí počkat, než první vlákno opustí metody P nebo V.

V Javě je semafor reprezentován třídou **Semaphore**, operace P(S) je zastoupena metodou *acquire* a operaci V(S) představuje metoda *release*.

Pro okamžité uspání vlákna jsem nastavil semaforu výchozí hodnotu 0. Tím jsem zajistil, že jakmile se zavolá P (acquire), tak se vlákno uspí, což je očekávaný efekt.

Při implementaci služby pro upload souboru na server se neobjevily žádné komplikace. Při nahrávání se soubor lineárně čte pomocí **FileInputStreamu** a data se rovnou odesílají přes bluetooth na server.

Na proti tomu u služby pro stažení souboru se objevil problém. Původně jsem stažení souboru implementoval pomocí semaforu, kdy jsem přečetl přijatá data, zpracoval je a zavolal jsem metodu P nad semaforem. Když byla k dispozici nová data, byla zavolána metoda V, čímž se data přečetla a zpracovala a takhle pořád dokola. Zádrhel byl v tom, že zpracování dat mohlo trvat o pár milisekund déle, takže se ztrácely celé packety. Řešením bylo, že jsem zrušil čekací semafor a místo něj jsem použil datovou strukturu **Fronta** pro příchozí packety. Fronta je abstraktní datový typ typu FIFO[12]. Velmi často se používá pro meziprocesovou nebo mezivláknovou komunikaci. Zde je použita pro mezivláknovou komunikaci. Standartní implementace fronty

[11] Hodnota semaforu

[12] First In First Out = první dovnitř, první ven

není *thread safe*, což znamená, že pokud k frontě přistoupí ve stejný okamžik dvě různá vlákna a upraví hodnoty, tak není vždy jasné, co se stane. Od javy verze 1.5 přišel *Java Concurrent Framework*, který obsahuje třídy pro pokročilou práci s vlákny. Mimo jiné obsahuje všechny kolekce ve verzi thread safe. Pro frontu je to rozhraní **BlockingQueue**, které je implementováno devíti různými způsoby. Já jsem vybral nejjednodušší implementaci pomocí pole, tedy třídu **ArrayBlockingQueue**. Při vytváření nové instance blokovací fronty je potřeba nadefinovat maximální velikost fronty. Před vložením prvku do fronty se zkontroluje, zda-li má fronta volné místo na vložení. Pokud volné místo nemá, tak se vlákno zablokuje a bude čekat do doby, než se místo uvolní. Toto je nežádoucí jev, takže jsem zvolil maximální velikost fronty dostatečně velkou. Při výběru prvku z fronty se kontroluje, zda-li fronta není prázdná. Pokud je prázdná, vlákno se opět zablokuje a počká, dokud se ve frontě neobjeví prvek. Díky frontě se přestaly ztrácet přijaté packety a přijaté soubory nebyly porušené.

4 Závěr

Cílem práce bylo vytvořit aplikaci pro ovládání stimulátoru. To se povedlo velmi dobře. Grafické zpracování rozhraní je velmi intuitivní a dobře se ovládá. Při implementaci DataBinding frameworku jsem narazil na nespočet překážek, na které nebylo nikde na internetu řešení, protože jde o velmi mladou technologii. Díky dobře napsaným testům jsem odhalil nemalé množství chyb způsobené kopírováním jednou vymyšlených konstrukcí do dalších implementací. V aplikaci je implementováno pět různých experimentů. Přenosový protokol podporuje pouze jeden experiment - ERP.

Aplikaci lze snadno rozšířit o další typy experimentů. Při rozšíření se musí upravit i komunikační protokol, aby podporoval nové typy experimentů. Další možnost, jak rozšířit aplikaci, se nabízí implementace módu pro simulaci vybraného experimentu přímo na mobilu, aby si uživatel mohl zobrazit, jak to bude fungovat, než konfiguraci experimentu pošle do stimulátoru.

Při ověřování funkčnosti ovládání stimulátoru jsem objevil chybu ve firmware stimulátoru. Když se nahraje nová konfigurace do stimulátoru, tak se neaktualizuje display na stimulátoru. Jedná se o nepřijemnou chybu, protože uživatel neví, jestli se data nahrála v pořádku, či nikoliv. V budoucí verzi firmware bude chyba odstraněna. Zatím stačí poslat po nahrání nového experimentu příkaz pro obnovení displaye a hodnoty se vykreslí správně.

Literatura

[1] Herout, P. *Testování pro programátory.* KOPP, Plzeň 2016. ISBN 978-80-7232-481-1.

[2] Jaakko Malmivuo, R. P. *Základní typy EEG vln* [online]. OXFORD UNIVERSITY PRESS, 1995. [cit. 2016/10/19]. Dostupné z: `http://www.bem.fi/book/13/13.htm`.

[3] MUDr. Petr Kaňovský, M. J. D. C. *Evokované potenciály v klinické praxi.* NCO NZO, Brno 2000. ISBN 80-7013-306-6.

[4] Prof. Miroslav Kuba, P. M. *Motion-onset Visual Evoked Potentials and their Diagnostic Applications.* Nucleus HK, Praha 2006. ISBN 80-86225-89-5.

[5] Štanglová, T. *Test HW stimulátoru proměření ERP experimentů.* Plzeň, 2014. Bakalářská práce. Západočeská univerzita v Plzni. Fakulta aplikovaných věd.

[6] *Elektroencefalografie* [online]. Wikiskripta, 2016. [cit. 2016/11/14]. Dostupné z: `http://www.wikiskripta.eu/index.php/Elektroencefalografie`.

[7] *Životní cyklus aktivity* [online]. Google, 2017. [cit. 2017/01/29]. Dostupné z: `https://developer.android.com/images/activity_lifecycle.png`.

Seznam obrázků

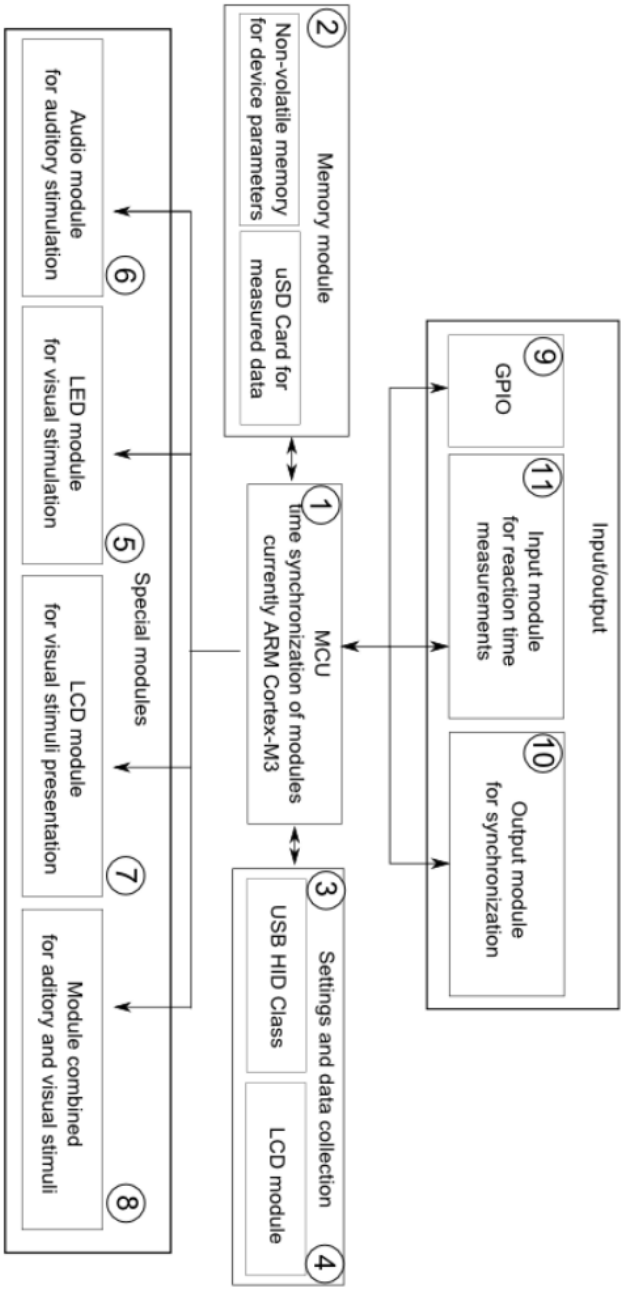
Seznam tabulek

A Uživatelská dokumentace

Pro nainstalování aplikace je potřeba povolit v nastavení mobilu instalování aplikací z cizích zdrojů: *Nastavení -> Zabezpečení -> zaškrtnout možnost "Neznámé zdroje"* Aplikace zatím není dostupná na Google Play.

Pro instalaci aplikace je nejvhodnější použít Android studio a v něm se řídit pokyny pro instalaci. Když není k dispozici Android studio, lze aplikaci nainstalovat příkazem `gradle installDebug` za předpokladu, že na počítači je přítomný Gradle.

B Blokové schéma HW stimulátoru na KIV ZČU



Obrázek B.1: Blokové schéma HW stimulátoru na KIV ZČU

C Příklady zdrojových kódů

```
<?xml version="1.0" encoding="utf-8"?>
<layout>
    <data>
        <variable
            name="foo"
            type="cz.zcu.fav.bindingexample.Foo" />
    </data>
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:id="@+id/textView"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{foo.bar}" />
        <EditText
            android:id="@+id/editText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@={foo.bar}"/>
    </LinearLayout>
</layout>
```

Listing C.1: XML soubor pro deklaraci rozložení obrazovky s využitím DataBinding frameworku

```
private final byte[] data = new byte[BtPacket.PACKET_SIZE];
public void run() {
  byte[] tempBuffer = new byte[BtPacket.PACKET_SIZE];
  int count;
  int totalSize = 0;
  while (true) {
    try {
      count = mmInStream.read(tempBuffer);
      int freeBytes = BtPacket.PACKET_SIZE - totalSize;
      int byteCount = count > freeBytes ? freeBytes : count;
      System.arraycopy(tempBuffer, 0, data,
        totalSize, byteCount);
      totalSize += count;
      if (totalSize >= BtPacket.PACKET_SIZE) {
        Intent intent = new Intent(ACTION_DATA_RECEIVED);
        intent.putExtra(EXTRA_DATA_CONTENT,
            Arrays.copyOf(data, data.length));
        LocalBroadcastManager.getInstance(
            BluetoothService.this).sendBroadcast(intent);
        totalSize %= BtPacket.PACKET_SIZE;
        // Zkopirovani zbyvajicich dat z bufferu do
        // hlavnich dat pro pristi pouziti
        Arrays.fill(data, totalSize, data.length, (byte) 0);
        System.arraycopy(tempBuffer, count - totalSize,
            data, 0, totalSize);
      }
    } catch (Exception e) {
      connectionLost();
      break;
    }
  }
}
```

Listing C.2: Vytváření packetů

D Obsah přiloženého DVD

Obsah přiloeného DVD:

- složka *bin* - obsahuje soubor s aplikaci Remotestimulatorcontrol.apk
- složka *doc* - obsahuje Stechmuller_BPINI.pdf - dokument bakalářské práce
- složka *src* - obsahuje zdrojové kódy samotné aplikace a veškeré konfigurační soubory pro Android studio
- složka *tex* - obsahuje zdrojové kódy dokumentu bakalářské práce psané v LaTeXu
- soubor *readme.txt* - zkopírovaný obsah této přílohy