



---

**POZNAN UNIVERSITY OF TECHNOLOGY**

---

**Patryk Dąbrowski 100584**  
**Aleksander Kędzierski 98875**  
**Paweł Lampe 99277**  
**Mateusz Sikora 99615**

# Platforma zarządzania zdarzeniami na urządzeniach mobilnych $if\{y\}$

Bachelor's Thesis

Supervisor: dr inż. Jerzy Błaszczyński

Poznań, 2014



# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>5</b>
1.1	Opis problemu i koncepcja jego rozwiązania (motywacja?) . . . . .	5
1.2	Cele i zakres pracy . . . . .	6
1.3	Omówienie pracy . . . . .	6
<b>2</b>	<b>Wymagania</b>	<b>7</b>
2.1	Wymagania funkcjonalne . . . . .	7
2.1.1	Przypadki użycia platformy . . . . .	7
2.1.2	Przypadki użycia Aplikacji - przykładowe Recepty . . . . .	7
2.2	Wymagania pozafunkcjonalne . . . . .	7
<b>3</b>	<b>Zarządzanie zdarzeniami na urządzeniach mobilnych</b>	<b>11</b>
3.1	Definicja pojęć . . . . .	11
3.2	Istniejące rozwiązania. . . . .	11
3.2.1	On X . . . . .	11
3.2.2	Tasker . . . . .	12
<b>4</b>	<b>Architektura platformy</b>	<b>13</b>
4.1	Recepty . . . . .	13
4.2	Biblioteka . . . . .	14
4.3	Aplikacja kliencka . . . . .	14
4.4	Targowisko . . . . .	14
4.5	Serwer. . . . .	14
<b>5</b>	<b>Opis implementacji</b>	<b>17</b>
5.1	Recepty . . . . .	17
5.1.1	Parametry – requestParams . . . . .	18
5.1.2	Używane Podfunkcjonalności – requestFeatures . . . . .	18
5.1.3	Logika recept – handleEvent . . . . .	19
5.1.4	Aktywacja . . . . .	19
5.2	Biblioteka . . . . .	21
5.2.1	Serwis . . . . .	21

5.3	Aplikacja kliencka . . . . .	21
5.3.1	Moduły dostępu do systemu . . . . .	21
5.3.2	Obsługa Targowiska . . . . .	21
5.3.3	Obsługa pobranych Recept. . . . .	22
5.3.4	Komunikacja serwisu z aplikacją kliencką . . . . .	22
5.4	Targowisko . . . . .	23
5.5	Serwer. . . . .	23
5.5.1	Repozytorium recept . . . . .	23
5.5.2	Serwer recept grupowych . . . . .	23
5.6	Protokół komunikacji. . . . .	23
5.7	Użyte technologie . . . . .	23
5.7.1	Android . . . . .	23
5.7.2	Android SDK . . . . .	23
5.7.3	Apache Commons . . . . .	23
5.7.4	Apache HTTP Server . . . . .	23
5.7.5	Git . . . . .	23
5.7.6	HTML 5 . . . . .	24
5.7.7	Hibernate . . . . .	24
5.7.8	JSON . . . . .	24
5.7.9	Java 6 . . . . .	24
5.7.10	JavaScript . . . . .	24
5.7.11	Apache Maven . . . . .	24
5.7.12	MySQL . . . . .	24
5.7.13	PHP . . . . .	24
5.7.14	RESTeasy . . . . .	25
5.7.15	SpringFramework . . . . .	25
5.7.16	Vaadin. . . . .	25
5.7.17	JUnit . . . . .	25
5.8	Użyte narzędzia . . . . .	25
5.8.1	Apache Tomcat . . . . .	25
5.8.2	Eclipse. . . . .	25
5.8.3	Android developer tools . . . . .	25
5.8.4	String Tool Suite. . . . .	25
5.8.5	Emacs . . . . .	25
5.8.6	Git bash for windows . . . . .	26
5.8.7	Github. . . . .	26
5.8.8	Latex . . . . .	26
5.8.9	Linux . . . . .	26
5.8.10	Notepad++. . . . .	26
5.8.11	Przeglądarki internetowe . . . . .	26
5.8.12	Windows. . . . .	26
5.9	Użyty sprzęt . . . . .	26
5.9.1	Komputery klasy PC . . . . .	26
5.9.2	LG Swift GT540. . . . .	26
5.9.3	Media-Droid IMPERIUS EN3RGY MT7013. . . . .	26

5.9.4	Motorola Defy MB525 . . . . .	27
5.9.5	Sony LT18 Xperia Arc S . . . . .	27
5.9.6	Samsung Galaxy Mini GT-S5570 . . . . .	27
5.10	Opis pakietów. . . . .	27
5.10.1	Pakiety Aplikacji . . . . .	27
5.10.2	Pakiety Biblioteki . . . . .	27
5.10.3	Pakiety Serwera . . . . .	27
<b>6</b>	<b>Testy oraz wyniki?</b>	<b>29</b>
<b>7</b>	<b>Zakończenie</b>	<b>31</b>
<b>A</b>	<b>Przewodnik użytkownika</b>	<b>33</b>
A.1	Opis Podfunkcjonalności . . . . .	33
A.1.1	Akcelerometr (YAccelerometerFeature.java) . . . . .	33
A.1.2	Battery (YBatteryFeature.java). . . . .	33
A.1.3	SMS (YSMSFeature.java) . . . . .	33
A.1.4	Wifi (YWifiFeature.java) . . . . .	33
A.1.5	GPS (YGPSFeature.java) . . . . .	33
A.1.6	Sound (YSoundFeature.java) . . . . .	33
A.1.7	RawPlayer (YRawPlayerFeature.java) . . . . .	33
A.1.8	Group (YGroupFeature.java). . . . .	33
A.1.9	Geocoder (YGeocoderFeature.java) . . . . .	33
A.1.10	Time (YTimeFeature.java). . . . .	34
A.1.11	AudioManager (YAudioManager.java) . . . . .	34
A.1.12	Text (YTextFeature.java) . . . . .	34
A.1.13	Internet (YInternetFeature.java) . . . . .	34
A.1.14	Calls (YCallsFeature.java) . . . . .	34
A.1.15	Notification (YNotificationFeature.java) . . . . .	34
A.2	TODO . . . . .	34
	<b>Bibliografia</b>	<b>35</b>



# Wstęp

## 1.1 Opis problemu i koncepcja jego rozwiązania (motywacja?)

Współczesne urządzenia mobilne dysponują ogromnym zbiorem możliwości. Nie są to już tylko telefony które dawniej służyły wyłącznie do komunikacji. Obecnie rynek pełen jest urządzeń z zakresu; od tabletu do smartfonu. Mnogość typów urządzeń oraz tendencja do upodabniania się między sobą uniemożliwia jednoznaczne stwierdzenie do czego właściwie służą.

Nie lepiej sytuacja ma się w przypadku programistów piszących aplikacje na urządzenia mobilne. Nie dość, że niemal każde urządzenie ma inny osprzęt, to co więcej, na rynku figuruje kilka wiodących mobilnych systemów operacyjnych. Wszystko to wpływa dość drastycznie na charakter rynku aplikacji mobilnych. W większej części są to proste aplikacje realizujące bardzo ściśle określone usługi. Prowadzi to do sytuacji, w której na przeciętnym smartfonie trzeba mieć 10-20 aplikacji które pozwolą osiągnąć stabilny poziom zadowolenia.

Gdyby istniała otwartoźródłowa aplikacja pozwalająca na stosunkowo prosty dostęp do możliwości telefonu oraz ułatwiająca proces programowania, wiele małych aplikacji mogło by stać się prostymi receptami instruującymi aplikację-matkę jak wykonać stosunkowo proste zadania. W tym przypadku 10-20 małych aplikacji można by zamienić na jedną zawierającą w środku 10-20 prostych kawałków kodu którymi można w pełni zarządzać.

Taka aplikacja oczywiście nie istnieje. Istnieją podobne, mniej lub bardziej udane rozwiązania. Wszystkie jednak są komercyjne co jest kluczową wadą. Zamkniętość kodu – bo o tym tutaj mowa, ogranicza zbiór osób zaangażowanych w tworzenie i pielęgnowanie kodu. Ma to ogromne implikacje na rozwój aplikacji, co z kolei uderza w jej użytkowników. W przypadku jakiegokolwiek złożonego problemu, użytkownik nie jest w stanie samemu sprawdzić czy problem leży po stronie jego kodu, czy po stronie kodu aplikacji.

Skoro wyżej wspomniana, otwartoźródłowa aplikacja nie istnieje, warto by ją stworzyć. Unieściwiło by to wszystkie wspomniane powyżej problemy. Taka też idea leży u podstaw tej pracy. Stworzyć wolną, otwartoźródłową aplikację służącą przeciętnym użytkownikom. Co jednak nawet ważniejsze, stworzyć kod, który będzie mógł zostać użyty przez zaawansowanych użytkowników-programistów.

## 1.2 Cele i zakres pracy

Postawowym celem niniejszej pracy, jest stworzenie otwartoźródłowej biblioteki uproszczającej dostęp do podzespołów urządzenia mobilnego. Jest to o tyle ważne, iż tworzy warstwę abstrakcji nad systemem operacyjnym. Dzięki temu, kod który przykładowo przetwarza dane z GPS, pozostaje identyczny dla systemów Android, iOS tudzież Windows Phone. Inna jest tylko implementacja biblioteki dla danej platformy. Niniejsza praca zakłada implementację biblioteki tylko dla systemu Android.

Drugim co do ważności celem, jest stworzenie przykładowej aplikacji prezentującej możliwości biblioteki. Z racji, iż implementacja biblioteki obejmuje tylko system Android, implementacja aplikacji również. Fakt, iż aplikacja jest tylko przykładem, nie oznacza, że większość wykonanej pracy stanowi biblioteka. Wręcz przeciwnie. Lwią część napisanego kodu stanowi aplikacja wraz z używanymi przez nią aplikacjami webowymi. Aplikacja bowiem, jako, że jest środowiskiem uruchomieniowym dla krótkich kawałków kodu – recept, potrzebuje zdalnego repozytorium będącego niczym innym jak stroną internetową. Potrzebuje również serwera utrzymującego informacje dla recept które korzystają z komunikacji w obrębie grup użytkowników.

Reasumując, kod który musi powstać, to biblioteka, aplikacja, repozytorium recept oraz serwer dla recept grupowych.

## 1.3 Omówienie pracy

Nixx nett hier



# Wymagania

## 2.1 Wymagania funkcjonalne

### 2.1.1 Przypadki użycia platformy

#### UC1 Tworzenie Recepty

- 1 Użytkownik wchodzi na stronę Targowiska.
- 2 Użytkownik tworzy nową Receptę.
- 3 Użytkownik pisze kod w edytorze online.
- 3a Użytkownik pisze kod lokalnie (np. w Eclipse) i przekazuje kod do Targowiska.
- 4 Serwer kompiluje receptę.
- 5 Użytkownik pobiera receptę na telefon.
- 6 Recepta działa na telefonie.

#### UC2 Ocena Recepty

- 1 Użytkownik wchodzi na stronę Targowiska.
- 2 Targowisko

### 2.1.2 Przypadki użycia Aplikacji - przykładowe Recepty

## 2.2 Wymagania pozafunkcjonalne

ID:	PF01
Nazwa:	System operacyjny dla aplikacji mobilnej
Kategoria:	Środowisko
Priorytet:	Wysoki
Opis:	Systemem operacyjnym aplikacji mobilnej jest Android w wersji minimum 2.1.

ID:	PF02
Nazwa:	Środowisko uruchomieniowe dla aplikacji serwerowej
Kategoria:	Środowisko
Priorytet:	Wysoki
Opis:	Aplikacja serwerowa powinna działać na maszynie wirtualnej Java.
ID:	PF03
Nazwa:	Używane technologie
Kategoria:	Technologie
Priorytet:	Wysoki
Opis:	Wykorzystane technologie nie mogą być płatne
ID:	PF04
Nazwa:	Zunifikowane środowisko programistyczne
Kategoria:	Narzędzia
Priorytet:	Wysoki
Opis:	Programiści muszą zdecydować się na wspólne narzędzie do redagowania kodu (np. Eclipse)
ID:	PF05
Nazwa:	Ograniczone zużycie energii urządzenia mobilnego
Kategoria:	Wydajność i niezawodność
Priorytet:	Średni
Opis:	Działanie aplikacji nie powinno w znaczącym stopniu skracać czasu pracy urządzenia na baterii
ID:	PF06
Nazwa:	Ograniczone zużycie zasobów urządzenia mobilnego.
Kategoria:	Wydajność i niezawodność
Priorytet:	Średni
Opis:	Aplikacja nie powinna spowalniać działania innych aplikacji.
ID:	PF07
Nazwa:	Czas reakcji aplikacji na zdarzenie
Kategoria:	Wydajność i niezawodność
Priorytet:	Wysoki
Opis:	Aplikacja powinna reagować na zdarzenia lokalne w mniej niż 2 sekundy

ID:	PF08
Nazwa:	Zgodność ze standardami kodowania dla języka Java
Kategoria:	Zgodność ze standardami
Priorytet:	Wysoki
Opis:	Zarówno kod aplikacji mobilnej, jak i serwerowej powinien być redagowany zgodnie ze standardami dla języka Java

ID:	PF09
Nazwa:	Przechowywanie haseł
Kategoria:	Bezpieczeństwo
Priorytet:	Wysoki
Opis:	Szyfrowane zapamiętywanie hasła użytkownika.

ID:	PF10
Nazwa:	Przechowywanie haseł
Kategoria:	Bezpieczeństwo
Priorytet:	Wysoki
Opis:	Przechowywanie skrótu hasła na serwerze.



# Zarządzanie zdarzeniami na urządzeniach mobilnych

## 3.1 Definicja pojęć

- Podfunkcjonalność (ang. Feature) – Część biblioteki zapewniająca Receptom dostęp do pozdbioru funkcjonalności Androida.
- Zdarzenie (ang. Event) – Zmiana stanu systemu, która powoduje uruchomienie kodu Recepty.
- Recepta (ang. Recipe) – Napisany przez użytkownika fragment kodu opisujący, co ma się zdarzyć po spełnieniu pewnych warunków.
- Targowisko (ang. Market) – Aplikacja internetowa pozwalająca tworzyć i pobierać Recepty.
- Aplikacja – Aplikacja androidowa wykorzystująca bibliotekę `if{Y}`.
- Serwer Grup – Komputer z działającą aplikacją, która zarządza grupami użytkowników i Zdarzeniami Grupowymi.
- Zdarzenie Grupowe – Zdarzenie związane z Grupą, wysyłane lub odbierane przez Aplikację z Serwera Grup.
- Grupa – Zbiór użytkowników identyfikowalny przez nazwę zdefiniowany na Serwerze Grup.

## 3.2 Istniejące rozwiązania

### 3.2.1 On X

Aplikacja Microsoftu umożliwiającą kontrolowanie telefonu z Androidem używając kodu w JavaScriptcie. Umożliwia wysyłanie Zasad (Rules) na telefon poprzez stronę internetową. Dostęp do funkcjonalności Androida jest zapewniony przez api w postaci Wyzwalaczy (Triggers) i Akcji (Actions). Cały system jest niestety połączony z Facebookiem i wymaga posiadania tam konta. Na podstawie [1].

### **3.2.2 Tasker**

# Architektura platformy

System składa się z biblioteki, przykładowej aplikacji appIFY oraz aplikacji działających na serwerze - Serwera Grup oraz Targowiska. Aplikacja korzysta z biblioteki oraz komunikuje się z serwerem. Oprócz tego Serwer Grup oraz Targowisko udostępniają z poziomu przeglądarki takie funkcje jak rejestracja użytkowników czy tworzenie Recept. Kluczowym założeniem było maksymalne uproszczenie kodu recept. TODO: Jeśli UI z weba nie będzie, to wyjechać wzmiankę

Kod Aplikacji jest podzielony na dwie części:

- bibliotekę IFY
- aplikację appIFY

Celem takiego podziału jest ułatwienie tworzenia innych aplikacji opartych o bibliotekę.

## 4.1 Recepty

Miejscem, gdzie zawarta jest główna logika Aplikacji są Recepty – są w nich opisane wszystkie zdarzenia, które mają nastąpić po spełnieniu pewnych ściśle określonych warunków. Docelowo będą one tworzone przez użytkowników i pobierane z Targowiska, jednak istnieją także przykładowe Recepty wbudowane w Aplikację, mające na celu ułatwienie użytkownikom tworzenia nowych na ich podstawie oraz rozszerzenie początkowej funkcjonalności aplikacji. Na receptę składają się:

- opis używanych podfunkcjonalności
- opis wymaganych parametrów
- opis jej właściwego działania

Deklarowanie używanych podfunkcjonalności ma dwa główne cele - po pierwsze, użytkownik widzi, czego używa recepta, co nieco poprawia jego bezpieczeństwo przy używaniu recept innych użytkowników, po drugie pozwala to inicjalizować nasłuchiwanie zdarzeń systemowych tylko wtedy, gdy istnieje aktywna recepta, która na nie reaguje - kod recepty nie musi inicjalizować większości podfunkcjonalności, wystarczy deklaracja ich używania. Wyjątkiem jest podfunkcjonalność grup, gdzie komunikację należy zainicjalizować.

Parametry pozwalają użytkownikowi na dostosowanie recepty do swoich wymagań, bez potrzeby pisania nowej. W naszych przykładowych receptach były to np. numer telefonu

do wysłania SMS lub jego tekst czy też zasięg znajdowania znajomych na podstawie GPS.

TODO: Lanie wody poniżej? Właściwa logika recepty jest zawarta w funkcji reakcji na zdarzenie. Wydaje się to sposób prostszy, niż na przykład ciągłe działanie recepty w osobnych wątku i odpytywanie Podfunkcjonalności o zdarzenia w kodzie recepty. Jest to rozwiązanie podobne do wzorca obserwatora, Recepta staje się jednak obserwatorem automatycznie na podstawie zadeklarowanych podfunkcjonalności, a wszystkie zdarzenia wywołują tą samą metodę w Receptie. Takie rozwiązanie minimalizuje ilość kodu w receptie, co było kluczowym celem.

## 4.2 Biblioteka

Biblioteka zawiera głównie API dostępne z poziomu recept, czyli między innymi Podfunkcjonalności, które agregują i upraszczają dostęp do metod z API systemu Android. Oprócz tego znajduje się tam moduł odpowiedzialny za zarządzanie cyklem życia Recept i Podfunkcjonalności, który działa cały czas w tle. Podfunkcjonalności są inicjalizowane przez serwis przy uruchamianiu recepty, która deklaruje ich użycie. Zapewniają one dostęp do określonych funkcji, takich jak odczyt danych z sensorów, odbieranie i wysyłanie SMS'ów i wiele innych.

## 4.3 Aplikacja kliencka

Najważniejszym elementem Aplikacji jest interfejs użytkownika – ekrany takie jak wyświetlanie listy dostępnych lub aktywnych recept, ustalanie ich parametrów i ich włączanie i wyłączanie. Dodatkowo aplikacja jest zintegrowana z Targowiskiem umożliwiając pobieranie z niego recept. Umożliwia też logowanie się do Serwera Grup. TODO: Dop

## 4.4 Targowisko

1. geneza 2. rozwiązanie problemu 3. forkowanie 4. schmat bazy + 5. edytor online (artykuł?)

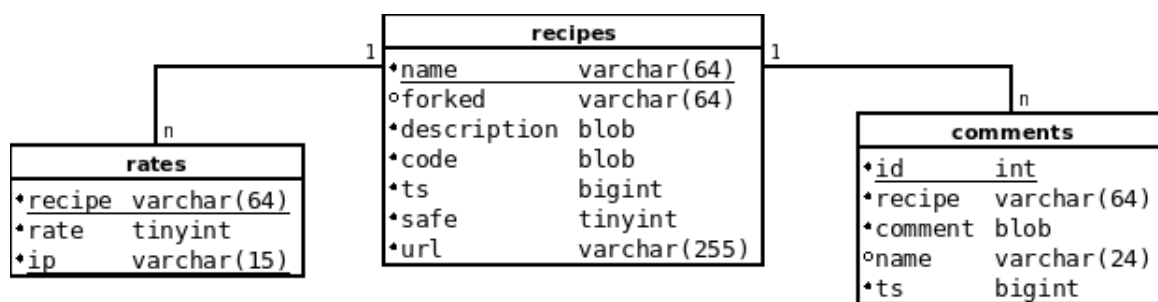
## 4.5 Serwer

Głównym zadaniem przy tworzeniu recept grupowych jest przekazywanie wiadomości umożliwiające komunikację między klientami i wymianę danych. Informacje niezbędne do działania tak ważnej funkcjonalności powinny być jednocześnie rozsyłane w prosty i łatwy do odczytania sposób przez każdą ze stron.

Bezpośrednia komunikacja z użyciem połączenia internetowego między urządzeniami mobilnymi nie jest możliwa, dlatego niezbędnym było wprowadzenie urządzenia pośredniczącego w przesyłaniu danych. Serwer pełni w takim wypadku funkcję łącznika, dzięki czemu znany jest adres na jaki należy wysłać komunikat który ostatecznie ma dotrzeć do odbiorcy. Koncepcja ta opiera się o tak zwany mechanizm odpytywania (ang. polling),



który jest prosty do zaimplementowania ale jednocześnie spełnia wszystkie wymagania stawiane w projekcie. Odbieranie danych z serwera wykonywane poprzez cykliczne zapytania eliminuje problem łączności między aplikacją a pośredniczącym serwerem



Rysunek 4.1: Schemat bazy danych targowiska

# Opis implementacji

## 5.1 Recepty

Recepty dziedziczą po klasie abstrakcyjnej YRecipe i implementują jej abstrakcyjne metody. Obrazuje to poniższy przykład recepty, która odrzuca wszystkie nadchodzące połączenia i wysyła SMS o zdefiniowanej przez użytkownika treści do dzwoniącej osoby.

```
public class YSampleCallsSMS extends YRecipe {
    @Override
    public void requestParams(YParamList params) {
        //Message to send in SMS
        params.add("MSG",YParamType.String, "Sorry, I'm busy.");
    }

    @Override
    public long requestFeatures() {
        return Y.Calls | Y.SMS;
    }

    @Override
    public void handleEvent(YEvent event) {
        //event is incoming call
        if(event.getId() == Y.Calls){
            YCallsEvent e = (YCallsEvent) event;
            //extract phone number
            String phone = e.getIncomingNumber();
            //discard call
            mFeatures.getCalls().discardCurrentCall();
            //send sms
            mFeatures.getSMS().sendSMS(phone, mParams.getString("MSG"));
        }
    }

    @Override
    public String getName() {
```

```
        return "YSampleCallsSMS";
    }
    @Override
    public YRecipe newInstance() {
        return new YSampleCallsSMS();
    }
}
```

### 5.1.1 Parametry – requestParams

Metoda requestParams ma za zadanie poinformować, jakich parametrów recepta wymaga do działania. Początkowo miała ona po prostu zwrócić listę i wyglądałaby tak:

```
public void requestParams() {
    YParamList params = new YParamList();
    params.add("MSG", YParamType.String, "Sorry, I'm busy.");
    return params;
}
```

Jednak tworzenie listy i zwracanie jej to dwie linie, które byłyby identyczne w każdej receptce - ich wpisywanie może nieco irytować. Wobec tego obecnie metoda ta przyjmuje jako argument pustą listę parametrów, którą ma za zadanie wypełnić, zgodnie z założeniem maksymalnego uproszczenia kodu recepty.

### 5.1.2 Używane Podfunkcjonalności – requestFeatures

Metoda requestFeatures ma za zadanie poinformować system, jakich Podfunkcjonalności używa Recepta. Początkowo była ona podobna do requestParams i wypełniała listę nowymi obiektami odpowiedniej klasy, co wyglądałoby tak:

```
public void requestFeatures(YFeatureList features) {
    features.add(new YCallsFeature());
    features.add(new YSMSFeature());
    return params;
}
```

Przy takim rozwiązaniu jednak tworzyło się wiele niepotrzebnych obiektów - poprawnie zainicjalizowane Podfunkcjonalności powinny być tworzone w systemie tylko raz. Wystarczyłaby zatem lista identyfikatorów, pozwalająca zainicjalizować odpowiednie Podfunkcjonalności. Identyfikatorów jest jednak na tyle mało, że tak naprawdę nie potrzeba prawdziwej listy, wystarczy maska bitowa. Ułatwia to przesyłanie takiej listy między modułami systemu, działającymi w różnych procesach - nie trzeba się martwić o implementację w liście interfejsu Parcelable, potrzebnego do przysyłania obiektów między procesami w Androidzie.

Ostatecznie zatem metoda ta zwraca liczbę typu long, będącą sumą bitów reprezentujących poszczególne Podfunkcjonalności. Mapowanie tych bitów jest zawarte w klasie Y.

```
[...]
public static final long Wifi = 0x0008;
public static final long GPS = 0x00010;
[...]
```

Dodatkowo warto zauważyć, że nazwy stałych w tej klasie odpowiadają nazwom Podfunkcjonalności oraz Zdarzeń - dla stałej **ABC** klasa z Podfunkcjonalnością nazywa się **YABCFeature**, a zdarzenie - **YABCEvent**. Powinno to ułatwić automatyczne generowanie kodu recept.

### 5.1.3 Logika recept – `handleEvent`

Metoda jest wywoływana, gdy w systemie nastąpi zdarzenie związane z Podfunkcjonalnością używaną przez receptę. W argumencie podawane jest zdarzenie – obiekt typu **YEvent**. Aby poznać szczegóły zdarzenia recepta musi sprawdzić jego typ porównując wartość zwracaną przez `getId()` ze stałymi z klasy **Y**. Następnie można rzutować zdarzenie na odpowiedni typ i poznać jego szczegóły.

Recepty mogą też zażądać pewnych danych od systemu, które są dostarczane asynchronicznie - na przykład przetłumaczenie danych z GPS na adres (Geocoder). Wyniki tego typu operacji również są przekazywane do recepty jako typ **YEvent**.

Z poziomu obsługi zdarzenia można także dostać się do listy Podfunkcjonalności oraz listy Parametrów poprzez metody `getFeatures()` i `getParams()`. Początkowo dostęp do Podfunkcjonalności odbywał się następująco:

```
YCallsFeature cf = (YCallsFeature) mFeatures.get(Y.Calls);
```

Jednak wymuszało to rzutowanie i niepotrzebnie wydłużało kod, zatem obecnie klasa **YFeatureList** zawiera metody pobierające konkretne podfunkcjonalności.

```
public YCallsFeature getCalls() {
    return (YCallsFeature) get(Y.Calls);
}
```

Ich utrzymanie może być później nieco kłopotliwe - każde dodanie Podfunkcjonalności będzie wymagało dodania odpowiedniej metody, jednak uproszczenie kodu recepty jest tego warte.

Warto również wspomnieć, że metoda `handleEvent` może rzucić dowolny wyjątek - recepta zostanie wówczas wyłączona. Ułatwia to pisanie recept zapewniając jednocześnie stabilność aplikacji.

### 5.1.4 Aktywacja

Fragmenty kodu przedstawione poniżej różnią się od oryginalnych – dla poprawy czytelności nie ma w nich tworzenia logów. Recepta jest aktywowana przez serwis, na podstawie nazwy i listy parametrów.

```
public int enableRecipe(String name, YParamList params) {
    int id = ++mRecipeID;
```

```

    int timestamp = (int) (System.currentTimeMillis() / 1000);
    YRecipe recipe = mAvailableRecipesManager.getRecipe(name).newInstance();
    long feats = recipe.requestFeatures();
    YFeatureList features = new YFeatureList(feats);
    initFeatures(features);
    params.setFeatures(feats);
    if(!recipe.initialize(this, params, features, id, timestamp)){
        return 0;
    }
    for (Entry<Long, YFeature> entry : features) {
        entry.getValue().registerRecipe(recipe);
    }
    mActiveRecipesManager.put(id, recipe);
    return id;
}

```

Generowany jest ID konkretnej instancji recepty oraz zapisywany jest czas jej uruchomienia. Następnie tworzony jest nowy obiekt typu właściwego do konkretnej recepty. W tym celu znajdujemy niezainicjalizowaną receptę w bazie i posługujemy się metodą `newInstance` - nie w tym miejscu kodu nie jest znana nazwa klasy recepty, aby móc wprost wywołać konstruktor. Innym możliwym rozwiązaniem byłby mechanizm refleksji, jednak to rozwiązanie jest szybsze, gdyż nie mogą być optymalizowane przez maszynę wirtualną [5]. Dalej na podstawie zwróconej przez receptę maski bitowej tworzona jest lista podfunkcjonalności wymaganych przez receptę do działania. Następnie podfunkcjonalności które już są aktywne są wpisywane do listy w miejsce niezainicjalizowanych, a pozostałe są aktywowane i dodawane do listy aktywnych.

```

protected void initFeatures(YFeatureList features) {
    for (Entry<Long, YFeature> entry : features) {
        Long featId = entry.getKey();
        YFeature feat = mActiveFeatures.get(featId);
        if (feat != null) {
            entry.setValue(feat);
        } else {
            feat = entry.getValue();
            feat.initialize(this);
            mActiveFeatures.add(feat);
        }
    }
}

```

Po zainicjalizowaniu Podfunkcjonalności Recepta jest w nich rejestrowana. Umożliwia to wywoływanie metody `handleEvent` w odpowiedzi na zdarzenia systemowe. Warto zauważyć, że zarówno Recepty jak i Podfunkcjonalności są leniwie inicjalizowane, co pozwala tymczasowo używać niezainicjalizowanych obiektów, a potem zastępować je innymi bez wykonywania zbędnych operacji.

```
public final boolean initialize(IYRecipeHost host, YParamList params,
    YFeatureList features, int id, int timestamp) {
    mHost = host;
    mParams = params;
    mFeatures = features;
    mId = id;
    mTimestamp = timestamp;
    Log = new YLogger(createTag(mId, getName()), host);
    try {
        init();
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

Sama inicjalizacja recepty to głównie wstrzyknięcie jej parametrów, Podfunkcjonalności, ID oraz czasu aktywacji. Oprócz tego jest tworzony Dziennik Recepty oraz jest wywoływana funkcja `init()` zawierająca kod inicjalizacyjny specyficzny dla danej recepty (na przykład otwarcie kanału komunikacji z Serwerem Grup). Takie rozwiązanie w połączeniu w modyfikatorze `final` w metodzie zapewnia jej wywołanie, a kod recepty nie ma dostępu do danych, które nie są mu potrzebne. Dodatkowo funkcja `init()` może się nie powieść - wyjątki są wówczas łapane, metoda `initialize()` zwraca wówczas wartość `false`, a recepta nie jej dodawana do listy aktywnych.

## 5.2 Biblioteka

### 5.2.1 Serwis

Wszystkie operacje odbywające się w bibliotece działają w kontekście serwisu, zaimplementowane w klasie `YAbstractRecipeService`. Serwis w Androidzie to komponent aplikacji przeznaczony do długotrwałego wykonywania operacji w tle, nieposiadający interfejsu użytkownika. [6] Wszelka komunikacja z użytkownikiem przebiega poprzez aplikację, która komunikuje się z serwisem.

## 5.3 Aplikacja kliencka

### 5.3.1 Moduły dostępu do systemu

### 5.3.2 Obsługa Targowiska

Moduł obsługi Targowiska jest odpowiedzialny za wyświetlanie danych dotyczących recept dodanych w aplikacji internetowej oraz pobieranie plików `.jar` ze skompilowanymi receptami, które następnie są zapisywane na pamięci wewnętrznej urządzenia mobilnego (w celu

zachowania tej samej bazy recept w przypadku w którym użytkownik usunie zewnętrzny nośnik pamięci z urządzenia). Informacje o plikach z receptami (ich nazwy oraz ścieżki) przechowywanymi na telefonie zapisywane są po pomyślnym pobraniu w bazie danych.

### 5.3.3 Obsługa pobranych Recept

W aplikacji klienckiej zrealizowanej w ramach pracy inżynierskiej rozróżniamy dwa typy recept - wbudowane i pobrane z Targowiska. Kod źródłowy recept pierwszego typu jest zawarty w kodzie źródłowym Aplikacji. W przypadku recept pobranych z Targowiska, w celu umożliwienia Aplikacji korzystania z takiej recepty wykorzystywane jest archiwum .jar, zawierające plik .dex (Dalvik Executable) z kodem wykonywalnym zrozumiałym dla maszyny wirtualnej Dalvik. Informacje potrzebne to załadowania kodu recepty (nazwa klasy oraz ścieżka dostępu do pliku .jar) przechowywane są w bazie danych recept pobranych na urządzenie.

### 5.3.4 Komunikacja serwisu z aplikacją kliencką

W opisie komunikacji między aplikacją kliencką a serwisem recept wykorzystane będą klasy z Android SDK - Messenger, Bundle i interfejs Parcelable. Klasa Messenger umożliwia przesyłanie danych między procesami. [7] Do opakowania danych wykorzystywana jest klasa Bundle, która przechowuje obiekty i typy prymitywne w postaci mapy. Warto wspomnieć, że aby uzyskać możliwość przechowania obiektu w tej klasie, musi on implementować interfejs Serializable lub Parcelable. Pierwszy z nich umożliwia serializację obiektów znaną z Javy, natomiast drugi został zaimplementowany w Android SDK w celu zwiększenia wydajności serializacji. W pracy inżynierskiej wykorzystujemy drugi z mechanizmów. Po uruchomieniu serwis recept wystawia obiekt implementujący interfejs IBinder służący do wiązania obiektów klasy Activity z obiektami klasy Service, z którym z kolei jest związany obiekt klasy Messenger zaimplementowany w serwisie recept. Aby ustawić połączenie, Aktywność musi stworzyć obiekt klasy Intent, sparametryzować go klasą Service z którą nawiązywane jest połączenie i zapewnić obiekt implementujący interfejs ServiceConnection, który reaguje na uzyskanie i zerwanie połączenia, a następnie wywołać metodę bindService jako parametr podając wspomniany wyżej obiekt klasy Intent. Po nawiązaniu połączenia następuje wymiana obiektów klasy Messenger, dzięki czemu możliwa jest komunikacja w obie strony. Warto dodać, że ten mechanizm komunikacji jest asynchroniczny. Wiadomości wysyłane przez klasę Messenger odbierane są przez klasę Handler, ich zawartość jest interpretowana dzięki wysłanemu kluczowi, a następnie dane są przekazywane serwisowi recept lub aktywności aplikacji klienckiej w celu dalszego przetwarzania. W pracy inżynierskiej wykorzystano dwie klasy dziedziczące po klasie Handler - ServiceHandler dla obsługi wiadomości przychodzących do serwisu recept i ActivityHandler dla obsługi wiadomości przychodzących do aplikacji klienckiej. W celu rozszerzenia komunikacji o wiadomości których obecna implementacja nie przewiduje, należy stworzyć własną klasę dziedziczącą po klasie ServiceHandler i we własnej implementacji klasy AbstractService nadpisać metodę getServiceHandler. Podobnie, aby rozszerzyć komunikację w drugą stronę należy stworzyć własną klasę dziedziczącą po ActivityHandler i użyć go do odbierania wiadomości od serwisu recept.



## 5.4 Targowisko

## 5.5 Serwer

### 5.5.1 Repozytorium recept

### 5.5.2 Serwer recept grupowych

## 5.6 Protokół komunikacji

Komunikacja aplikacji klienckich oparta jest o ciągle odpytywanie (ang. polling). Wymiana danych odbywa się przy użyciu tekstowego formatu danych JSON.

## 5.7 Użyte technologie

W tej części zaprezentowano opis technologii użytych bezpośrednio w implementacji składowych platformy.

### 5.7.1 Android

System operacyjny z rodziny Linux przeznaczony dla urządzeń mobilnych. Aktualnie rozwijane przez sojusz biznesowy Open Handset Alliance.

### 5.7.2 Android SDK

Platforma programistyczna umożliwiająca tworzenie aplikacji dla systemu Android. Zawiera wtyczkę do środowiska Eclipse, narzędzia wspierające prace programisty, emulator i biblioteki potrzebne do zbudowania aplikacji. Programy dedykowane platformie pisane są w języku Java i uruchamiane na maszynie wirtualnej Dalvik.

### 5.7.3 Apache Commons

### 5.7.4 Apache HTTP Server

Otwartoźródłowy serwer HTTP. Najpopularniejsze narzędzie tego typu na świecie. Jego wielką zaletą jest mnogość informacji na jego temat dostępnych w internecie oraz dostępność na większość znaczących systemów operacyjnych.

### 5.7.5 Git

Rozproszony oraz wieloplatformowy system kontroli wersji będący wolnym oprogramowaniem. Preferowane narzędzie programistów związanych z otwartym oprogramowaniem.

### 5.7.6 HTML 5

Język programowania służący do tworzenia współczesnych stron internetowych. Jest rozwinięciem oraz uproszczeniem języka HTML 4.

### 5.7.7 Hibernate

Narzędzie odwzorowań obiektowo-relacyjnych (ang. object-relation mapping, ORM) rozwijany na zasadzie wolnego oprogramowania. Umożliwia odwzorowania obiektowo-relacyjne, pamięć podręczną, leniwe (ang. Lazy loading), chciwe pobieranie oraz rozproszoną pamięć podręczną.

### 5.7.8 JSON

Skrót od JavaScript Object Notation. Jest to lekki, tekstowy format wymiany danych niezależny od języka programowania. Został wybrany ze względu na swoją czytelność i wsparcie ze strony bibliotek programistycznych.

### 5.7.9 Java 6

Język programowania cechujący się obiektowością (ang. Object-oriented programming, OOP) oraz silnym typowaniem. Kod źródłowy Javy kompilowany jest do kodu bajtowego interpretowanego przez maszynę wirtualną zapewnia to większa niezależność od platformy niż w innych podobnych językach np. C++.

### 5.7.10 JavaScript

Skryptowy język oprogramowania stosowany na stronach internetowych.

### 5.7.11 Apache Maven

Narzędzie automatycznego budowania oprogramowania dla języka JAVA. Głównymi problemami jakie rozwiązuje Maven przy budowaniu aplikacji są: zarządzanie zależnościami, możliwość wieloma modułami, wsparcie dla testów.

### 5.7.12 MySQL

System zarządzania relacyjnymi bazami danych. Jest to wolne oprogramowanie szczególnie upodobane przez twórców aplikacji internetowych. Bardzo dobrze współpracuje z językami takimi jak PHP czy Java

### 5.7.13 PHP

Obiektowy język programowania dedykowany generowaniu stron internetowych w czasie rzeczywistym. Szczególnie użyteczny w przypadku tworzenia prototypów tudzież niewielkich projektów wymagających stosunkowo niskiego poziomu abstrakcji.

### 5.7.14 RESTeasy

Framework oprogramowania służący do tworzenia aplikacji rozproszonych, oparty na wzorcu architektury oprogramowania Representational State Transfer (REST).

### 5.7.15 SpringFramework

Framework (Szkielet) tworzenia aplikacji w języku Java a w szczególności JavaEE. Do najważniejszych funkcji Springa zalicza się wstrzykiwanie zależności (ang. dependency injection, DI) oraz programowanie aspektowe (ang. aspect-oriented programming, AOP).

### 5.7.16 Vaadin

Framework sieciowy służący do tworzenia aplikacji sieciowych w szczególności interfejsu użytkownika w oparciu o Google Web Toolkit (GWT) w języku JAVA.

### 5.7.17 JUnit

Biblioteka służąca do tworzenia testów jednostkowych w języku Java.

## 5.8 Użyte narzędzia

### 5.8.1 Apache Tomcat

Kontener aplikacji sieciowych.

### 5.8.2 Eclipse

Popularne zintegrowane środowisko programistyczne (IDE) wspierające głównie język Java (wtyczki pozwalają obsługiwać inne języki).

### 5.8.3 Android developer tools

Wtyczka do Eclipse pozwalająca tworzyć aplikacje androidowe. Dodaje takie funkcjonalności jak edycja plików XML odpowiadających za wygląd aplikacji (również w trybie graficznym) czy debugowanie na telefonach oraz emulatorze.

### 5.8.4 Spring Tool Suite

Zintegrowane środowisko programistyczne oparte o Eclipse dostosowany do SpringFramework.

### 5.8.5 Emacs

Popularny, w pełni rozszerzalny edytor tekstowy spotykany głównie w systemach operacyjnych z rodziny Unix. Używany przez wysokiej klasy programistów oraz naukowców na całym świecie.

## 5.8.6 Git bash for windows

Narzędzie umożliwiające używanie Gita z linii poleceń w systemie Windows poprzez wbudowane środowisko MinGW.

## 5.8.7 Github

Serwis internetowy gromadzący społeczność programistów z całego świata. Służy jako hosting dla otwartoźródłowych projektów zarządzanych za pomocą systemu Git. Udostępnia szereg narzędzi wspierających - system śledzenia zadań, budowa statystyk.

## 5.8.8 Latex

## 5.8.9 Linux

Rodzina systemów operacyjnych będących wolnym oprogramowaniem oraz używających jądra Linux.

## 5.8.10 Notepad++

Prosty edytor tekstowy umożliwiający kolorowanie składni w wielu językach.

## 5.8.11 Przeglądarki internetowe

Programy takie jak Google Chrome, Mozilla Firefox czy Opera, używane w pracy do testowania rozwiązań mających postać strony internetowej.

## 5.8.12 Windows

System operacyjny firmy Microsoft.

# 5.9 Użyty sprzęt

## 5.9.1 Komputery klasy PC

Podstawowa platforma do wszystkich aspektów pracy, z wyjątkiem testowania, do którego użyliśmy także telefonów.

## 5.9.2 LG Swift GT540

Procesor: Qualcomm MSM7227 600 MHz Pamięć RAM: 256 MB System operacyjny: Android 4.0.1 (Cyanogen mod)

## 5.9.3 Media-Droid IMPERIUS EN3RGY MT7013

Procesor: dwurdzeniowy, 1GHz ARM7 MTK6577 Pamięć RAM: 256 MB System operacyjny: Android 4.1.2

### 5.9.4 Motorola Defy MB525

Procesor: TI OMAP3610 800 MHz Pamięć RAM: 512 MB System operacyjny: Android 4.3.1 (Cyanogen mod)

### 5.9.5 Sony LT18 Xperia Arc S

Procesor: Qualcomm MSM8255T 1,40 GHz Pamięć RAM: 512 MB System operacyjny: Android 4.0.4

### 5.9.6 Samsung Galaxy Mini GT-S5570

Procesor: Qualcomm MSM7227 600 MHz Pamięć RAM: 384 MB System operacyjny: Android 2.2

## 5.10 Opis pakietów

### 5.10.1 Pakiety Aplikacji

pl.poznan.put.cs.ify.app - główny pakiet Aplikacji. pl.poznan.put.cs.ify.jars - pakiet odpowiedzialny za zarządzanie plikami .jar zawierającymi recepty pobrane z Targowiska. pl.poznan.put.cs.ify.core - pakiet odpowiedzialny za zarządzanie dostępnymi i aktywowanymi Receptami. pl.poznan.put.cs.ify.appify.receipts - pakiet zawierający Recepty wbudowane w Aplikację. pl.poznan.put.cs.ify.app.ui - pakiet zawierający kontrolki interfejsu użytkownika. pl.poznan.put.cs.ify.app.ui.params - pakiet zawierający kontrolki interfejsu użytkownika wykorzystywane do wprowadzania parametrów przy inicjalizacji Recepty. pl.poznan.put.cs.ify.app.market - pakiet odpowiedzialny za pobieranie danych z Targowiska i wyświetlanie ich. pl.poznan.put.cs.ify.app.fragments - pakiet zawierający widoki ekranów aplikacji.

### 5.10.2 Pakiety Biblioteki

pl.poznan.put.cs.ify.api - pakiet główny Biblioteki. pl.poznan.put.cs.ify.api.exceptions - pakiet zawierający wyjątki, które mogą być rzucane przez metody z Biblioteki. pl.poznan.put.cs.ify.api.features - pakiet zawierający Podfunkcjonalności i Zdarzenia. pl.poznan.put.cs.ify.api.group - pakiet odpowiedzialny za obsługę Recept Grupowych. pl.poznan.put.cs.ify.api.log - pakiet odpowiedzialny za obsługę logowania i domyślny widok logów. pl.poznan.put.cs.ify.api.params - pakiet zawierający typy parametrów wykorzystywanych przez Recepty. pl.poznan.put.cs.ify.api.security - pakiet odpowiedzialny za moduł uprawnień Biblioteki. pl.poznan.put.cs.ify.api.types - pakiet zawierający typy danych wykorzystywanych przez Bibliotekę.

### 5.10.3 Pakiety Serwera

pl.poznan.put.cs.ify.webify - pakiet główny serwera. pl.poznan.put.cs.ify.webify.data.dao - pakiet zawierający warstwę dostępu do danych. pl.poznan.put.cs.ify.webify.data.entity - pakiet zawierający klasy odwzorowywane na bazę danych. pl.poznan.put.cs.ify.webify.data.enums

- pakiet zawierający potrzebne w bazie danych typy wyliczeniowe (np. lista ról). `pl.poznan.put.cs.ify.webify.gui`
- pakiet główny graficznego interfejsu użytkownika. `pl.poznan.put.cs.ify.webify.gui.windows`
- pakiet zawierający wszystkie okna aplikacji sieciowej. `pl.poznan.put.cs.ify.webify.gui.components`
- pakiet zawierający komponenty użyte w aplikacji. `pl.poznan.put.cs.ify.webify.gui.session`
- `pl.poznan.put.cs.ify.webify.service` - pakiet zawierający logikę. `pl.poznan.put.cs.ify.webify.rest`
- pakiet zawierający obsługę zapytań typu REST. `pl.poznan.put.cs.ify.webify.utils` - pakiet, w którym przechowywane są funkcje pomocnicze używane w całym projekcie.

# Testy oraz wyniki?





# Zakończenie



# Przewodnik użytkownika

## A.1 Opis Podfunkcjonalności

### A.1.1 Akcelerometr (YAccelerometerFeature.java)

Umożliwia reagowanie na odczyty akcelerometru wbudowanego w urządzenie.

### A.1.2 Battery (YBatteryFeature.java)

Umożliwia reagowanie na zmiany poziomu baterii urządzenia.

### A.1.3 SMS (YSMSFeature.java)

Umożliwia wysyłanie wiadomości SMS oraz reagowanie na wiadomości przychodzące.

### A.1.4 Wifi (YWifiFeature.java)

Umożliwia włączanie i wyłączanie modułu WiFi urządzenia.

### A.1.5 GPS (YGPSFeature.java)

Umożliwia śledzenie pozycji urządzenia za pomocą modułu GPS.

### A.1.6 Sound (YSoundFeature.java)

Pozwala odtwarzać pliki dźwiękowe.

### A.1.7 RawPlayer (YRawPlayerFeature.java)

### A.1.8 Group (YGroupFeature.java)

### A.1.9 Geocoder (YGeocoderFeature.java)

Umożliwia pobranie adresu związanego z podaną długością i szerokością geograficzną.

**A.1.10 Time (YTimeFeature.java)****A.1.11 AudioManager (YAudioManager.java)****A.1.12 Text (YTextFeature.java)****A.1.13 Internet (YInternetFeature.java)**

Umożliwia wysyłanie i pobieranie danych z podanego adresu.

**A.1.14 Calls (YCallsFeature.java)**

Umożliwia reagowanie na połączenia przychodzące i inicjowanie połączeń wychodzących.

**A.1.15 Notification (YNotificationFeature.java)**

Umożliwia wyświetlanie powiadomień w interfejsie graficznym urządzenia.

**A.2 TODO**

Cykl życia recepty i feature'a (ogólnie, rysunki, w architekturze) /sikor Dokładny opis de-aktywacji recepty (implementacja) /alx Podfunkcjonalności do sekcji o bibliotece - ogólny opis + te z przewodnika usera. Przypadki użycia /alx Wymagania pozafunkcjonalne /alx UML Serwisu i okolic /sikor Schemat komunikacji - aplikacja <-> serwis /sikor - z serwerem Tworzenie jarów - rozszerzyć /sikor Apache commons, latex - dopisać lub wyjechać narzędzia - android support v4

# Bibliografia

- [1] Projekt on{X} <http://www.onx.ms/#!findOutMorePage>. Ostatnio odwiedzone 6/02/13.
- [2] C. Walls. *Spring in action, 3rd edition*. Manning Publication Co, 2011.
- [3] Vaadin <https://vaadin.com/book/vaadin6/-/page/preface.html>
- [4] E. Gamma. *Design Patterns, First edition*. Person Education, Inc, 1995.
- [5] The Reflection API <http://docs.oracle.com/javase/tutorial/reflect/index.html> Ostatnio odwiedzone 31.01.2014
- [6] Android API Guide - Service <http://developer.android.com/guide/components/services.html> Ostatnio odwiedzone 31.01.2014
- [7] Android API Guide - Messenger <http://developer.android.com/guide/components/bound-services.html> Messenger Ostatnio odwiedzone 31.01.2014