



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

1	Wstęp	5
1.1	Motywacje	5
1.2	Cele i zakres pracy	5
1.3	Podział prac	6
1.4	Omówienie pracy	6
2	Wymagania	7
2.1	Wymagania funkcjonalne	7
2.1.1	Przypadki użycia platformy	7
2.1.2	Przypadki użycia Aplikacji - przykładowe Recepty	8
2.2	Wymagania pozafunkcjonalne	8
3	Zarządzanie zdarzeniami na urządzeniach mobilnych	11
3.1	Definicja pojęć	11
3.2	Istniejące rozwiązania.	12
3.2.1	On X	12
3.2.2	Tasker	12
4	Architektura platformy	13
4.1	Recepty	13
4.1.1	Cykl życia	14
4.1.2	Recepty Grupowe	14
4.2	Biblioteka	14
4.3	Aplikacja appIFY	16
4.4	Targowisko	16
4.4.1	Zintegrowane środowisko programistyczne.	17
4.4.2	Rozwidlanie.	17
4.4.3	Repozytorium recept	18
4.5	Serwer Grup	18
5	Opis implementacji	21
5.1	Recepty	21
5.1.1	Parametry – requestParams	22
5.1.2	Używane Podfunkcjonalności – requestFeatures	22

5.1.3	Logika recept – handleEvent	23
5.1.4	Aktywacja	23
5.1.5	Deaktywacja	25
5.1.6	Recepty Grupowe	26
5.2	Biblioteka	27
5.2.1	Zarządzanie receptami	27
5.2.2	Podfunkcjonalności	28
5.2.3	Dziennik	29
5.3	Aplikacja kliencka	30
5.3.1	Obsługa Targowiska	30
5.3.2	Obsługa pobranych Recept.	30
5.3.3	Komunikacja serwisu z aplikacją kliencką	31
5.4	Targowisko	31
5.4.1	Wzorzec MVC.	32
5.4.2	Interfejs graficzny	32
5.4.3	Edytor Ace	33
5.4.4	API.	36
5.4.5	Kompilacja Recept	37
5.5	Serwer recept grupowych	38
5.5.1	Baza danych	38
5.5.2	Grupy i użytkownicy	38
5.5.3	Uwierzytelnianie i autoryzacja	39
5.5.4	Kolejka komunikatów	39
5.5.5	Zarządzanie grupami	39
5.5.6	Komunikacja z receptami	39
5.6	Użyte technologie	41
5.7	Użyte narzędzia	43
5.8	Urządzenia mobilne	43
5.9	Opis pakietów.	44
5.9.1	Pakiety Aplikacji	44
5.9.2	Pakiety Biblioteki	44
5.9.3	Pakiety Serwera	45
6	Wyniki	47
6.1	Cykl życia recepty w praktyce	47
6.1.1	Pisanie kodu	48
6.1.2	Kompilacja i budowa	49
6.1.3	Dystrybucja	51
6.1.4	Pobranie na urządzenie mobilne.	51
6.1.5	Uruchomienie	51
6.1.6	Działanie.	51
7	Zakończenie	53
7.1	Podsumowanie	53
7.2	Dalszy rozwój (platformy)	53

Spis treści	3
A Recipe creation guide	55
Bibliografia	57

Wstęp

1.1 Motywacje

Współczesne urządzenia mobilne dysponują ogromnym zbiorem modułów. Większość z nich generuje zdarzenia. Zdarzeniami mogą być przykładowo; zmiana poziomu naładowania baterii, przychodzące połączenie czy odczyt danych akcelerometru. Niektóre zdarzenia obsługuje właściciel urządzenia, podczas gdy innymi zajmują się dedykowane aplikacje. Aby przykładowo wyłączyć moduł WiFi po rozłączeniu z siecią, należy pobrać aplikację *WiFi Auto Turn Off*. Aplikacji podejmujących akcje w reakcji na zdarzenia jest na rynku bardzo dużo.

Wyzwanie mające na celu zbudowanie pojedynczej aplikacji do zarządzania zdarzeniami oraz akcjami zostało obecnie podjęte przez zaledwie kilka firm. Wszystkie implementacje powstałe do tej pory są komercyjne. Hamuje to rozwój samych aplikacji jak i dziedziny. TODO: more

1.2 Cele i zakres pracy

Postawowym celem niniejszej pracy jest stworzenie otwartoźródłowej biblioteki upraszczającej dostęp do podzespołów urządzenia mobilnego. Podjęcie takie ma zadanie utworzyć warstwę abstrakcji nad systemem operacyjnym. Ma to potencjalnie zapewnić przenośność kodu pisanego przy użyciu biblioteki pomiędzy dowolnymi systemami operacyjnymi. Niniejsza praca zakłada implementację biblioteki tylko dla systemu Android.

Pozostałe cele to stworzenie prostej aplikacji mobilnej prezentującej możliwości biblioteki oraz dwóch internetowych aplikacji pomocniczych. Fakt implementacji biblioteki tylko dla systemu Android wymusza implementację aplikacji mobilnej również dla tego systemu operacyjnego.

Implementacja aplikacji pomocniczych jest nieunikniona ze względu na potrzebę centralizacji niektórych danych poza obrębem pojedynczego urządzenia.

Reasumując, należy stworzyć platformę w skład której wchodzi; biblioteka dla systemu Android, aplikacja mobilna dla systemu Android oraz dwie aplikacje internetowe.

1.3 Podział prac

Podział obowiązków został dokonany na podstawie kompetencji poszczególnych osób. Częściami platformy związanymi ściśle z systemem Android zajęli się Aleksander Kędziński oraz Mateusz Sikora. Częściami związanymi z technologiami internetowymi zajęli się Patryk Dąbrowski i Paweł Lampe. Nad kwestiami wspólnymi dla obu części pracował cały zespół.

Ostatecznie, poszczególne osoby dokonały:

- Patryk Dąbrowski
 - text
 - text
- Aleksander Kędziński
 - Zaprojektowania i implementacji części biblioteki IFY, w szczególności związanej z obsługą recept.
 - Implementacji części aplikacji appIFY.
- Paweł Lampe
 - Implementacji targowiska
 - Administracji serwerem z systemem Linux
- Mateusz Sikora
 - text
 - text

1.4 Omówienie pracy

Wymagania

W niniejszym rozdziale zebrano wymagania funkcjonalne oraz pozafunkcjonalne. Stanowią one opis pożądanego zachowania platformy. Wszystkie wymagania powstały w procesie zbierania wymagań w który zaangażowany był promotor pracy oraz TODO: grupa inżynierów ze specjalizacji TWO.

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.

ID:	PF11
Nazwa:	Długość recepty
Kategoria:	Użyteczność
Priorytet:	Wysoki
Opis:	Długość kodu recepty nie powinna w średnim przypadku przekraczać 100 linii

ID:	PF12
Nazwa:	Tworzenie recept
Kategoria:	Użyteczność
Priorytet:	Wysoki
Opis:	Proces tworzenia recepty nie powinien zajmować więcej niż 15 minut
ID:	PF13
Nazwa:	Dystrybucja recept
Kategoria:	Użyteczność
Priorytet:	Średni
Opis:	Proces dystrybucji recepty nie powinien zajmować więcej niż 5 minut
ID:	PF14
Nazwa:	Bezpieczeństwo recept
Kategoria:	Bezpieczeństwo
Priorytet:	Wysoki
Opis:	Recepta powinna korzystać tylko z biblioteki <code>if{y}</code> oraz pakietów narzędziowych Java

Zarządzanie zdarzeniami na urządzeniach mobilnych

Jak wspomniano w rozdziale 1, zarządzanie zdarzeniami na urządzeniach mobilnych oczekiwało się zaledwie kilku implementacji ze strony zainteresowanych firm. Tematyka jest więc otwarta zarówno na pomysły jak i terminologię.

W niniejszym rozdziale wprowadzono przede wszystkim pojęcia które pojawiają się w dalszych częściach pracy. Dokonano również przeglądu istniejących rozwiązań pod kątem mocnych oraz słabych stron.

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 Klientka – 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.
- Dziennik (ang. Log) – Moduł systemu odpowiedzialny za zapis zdarzeń.

3.2 Istniejące rozwiązania

3.2.1 On X

Aplikacja firmy Microsoft umożliwiającą kontrolowanie telefonu z systemem Android używając kodu napisanego w JavaScript. Umożliwia wysyłanie Zasad (ang. Rules) na telefon poprzez stronę internetową. Dostęp do funkcjonalności systemu Android jest zapewniony przez API w postaci Wyzwalaczy (ang. Triggers) i Akcji (ang. Actions). Cały system jest [niestety] połączony z Facebookiem i wymaga posiadania tam konta. Na podstawie [1].

3.2.2 Tasker

TODO:

Architektura platformy

System składa się z biblioteki IFY, przykładowej aplikacji appIFY oraz aplikacji działających na serwerze - Serwera Grup oraz Targowiska. Biblioteka zawiera moduł zarządzania receptami i zbiór Podfunkcjonalności. Aplikacja korzysta z biblioteki, umożliwia przeglądanie zasobów Targowiska i zarządzanie grupami. Targowisko umożliwia przechowywanie Recept i tworzenie ich z poziomu przeglądarki. Serwer grup zapewnia komunikację między Receptami na różnych urządzeniach w ramach grupy oraz zarządzanie grupami.

4.1 Recepty

Miejscem, gdzie zdefiniowana jest właściwe działanie Aplikacji są Recepty – są w nich opisane wszystkie zdarzenia, które mają nastąpić po spełnieniu pewnych 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. Właściwa logika recepty jest zawarta w funkcji reakcji na zdarzenie. Jest to rozwiązanie podobne do wzorca obserwatora, Recepta staje się jednak obserwatorem automatycznie na podstawie zadeklarowanych podfunkcjonalności, co pozwala zmniejszyć ilość kodu w receptach.

4.1.1 Cykl życia

W kontekście platformy `if{y}` zdefiniować można następujący cykl życia recepty:

1. Pisanie kodu
2. Kompilacja i budowa
3. Dystrybucja
4. Pobranie na urządzenie mobilne
5. Uruchomienie
6. Działanie

4.1.2 Recepty Grupowe

Główną ideą recept grupowych jest działanie jednocześnie na wielu urządzeniach, tak, aby zdarzenia na jednym urządzeniu mogły wpływać na drugie. Zatem komunikacja jest możliwa między instancjami tej samej recepty na różnych urządzeniach. Jest ona możliwa na dwa sposoby:

- Wysyłanie zdarzeń
Polega na utworzeniu zdarzenia, które jest obsługiwane na innym urządzeniu, które nasłuchuje go automatycznie.
- Wysyłanie danych
Polega na wysyłaniu na serwer danych, które potem mogą być odczytane na innym urządzeniu.

Nie da się natomiast napisać w Receptce kodu, który byłby wywoływany na serwerze. Klóciłoby się to z prostotą mechanizmu recepty - musiałaby się ona składać zarówno z kodu dla aplikacji, jak i dla serwera, a programista musiałby zadbać o ich współpracę. Mogłoby to też stanowić zagrożenie dla stabilności serwera w wypadku złośliwego kodu. Nie udało się też znaleźć scenariuszy użycia, w których taka funkcja byłaby konieczna. Dodatkowo przyjęta architektura pozwala na odseparowanie Serwera Grup od Targowiska.

4.2 Biblioteka

Biblioteka zawiera API dostępne z poziomu recept, czyli między innymi Podfunkcjonalności, które agregują i upraszczają dostęp do metod z API systemu Android oraz umożliwiają reagowanie na zdarzenia zachodzące na urządzeniu mobilnym. Istotnym elementem Biblioteki jest moduł zarządzający cyklem życia Recept i Podfunkcjonalności. W kontekście Biblioteki, cykl życia Recepty wygląda tak jak na rysunku 4.1.

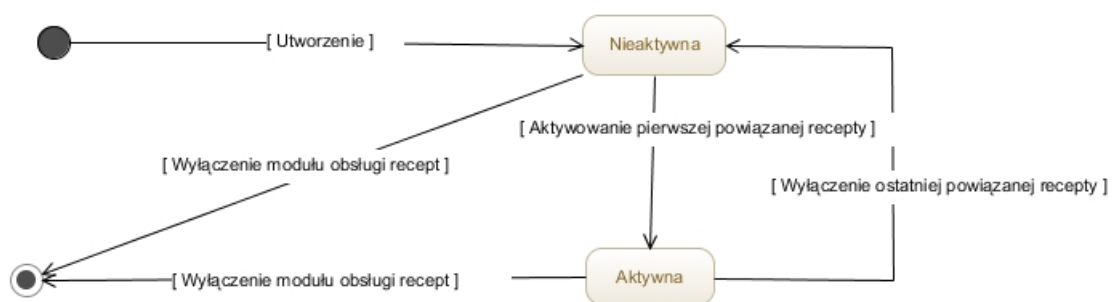


Rysunek 4.1: Cykl życia recepty z punktu widzenia biblioteki

Wyróżniamy dwa stany w których znajdują się Recepty:

- Dostępna - udostępnia podstawowe informacje - nazwę, zbiór parametrów potrzebnych do uruchomienia i deklaracje Podfunkcjonalności, z których będzie korzystać.
- Aktywna - została uruchomiona na urządzeniu mobilnym, działa cały czas w tle nasłuchując na Zdarzenia, podczas jej uruchamiania zostały podane parametry i przydzielone Podfunkcjonalności.

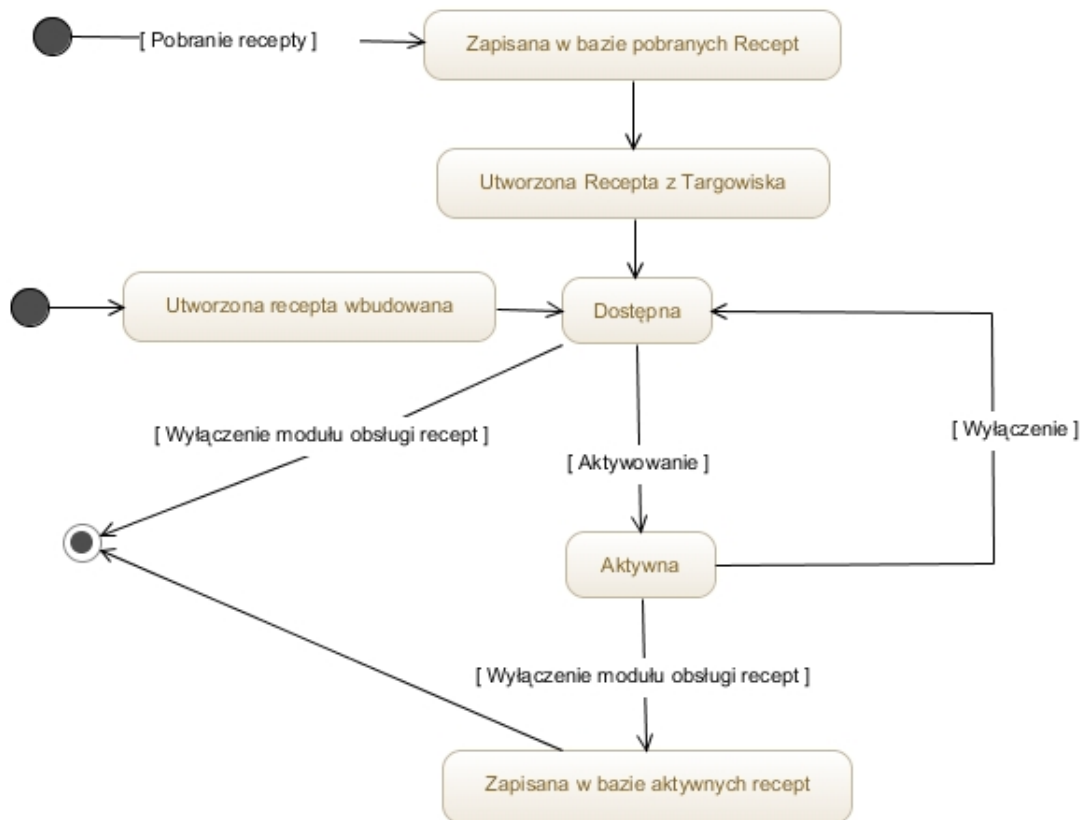
Warto zauważyć, że możliwe jest uruchomienie wielu instancji tej samej Recepty - mogą się różnić nadanymi parametrami. Cykl życia Podfunkcjonalności jest ściśle związany z Receptami - dana Podfunkcjonalność jest aktywna w systemie tylko jeżeli istnieją aktywne Recepty, które z niej korzystają. Moduł zarządzania Receptami aktywuje Podfunkcjonalność podczas uruchamiania pierwszej Recepty która zadeklarowała jej użycie i wyłącza ją kiedy nie pozostaną w systemie żadne aktywne Recepty które mogłyby z niej korzystać. Podczas wyłączania zwalniane są wykorzystywane zasoby. Biblioteka zawiera również mechanizm komunikacji z Aplikacjami Klientkimi. Podczas projektowania wzięto pod uwagę zapewnienie rozszerzenia istniejącego zbioru Podfunkcjonalności. Moduł zarządzania Receptami składa się z listy aktywnych i dostępnych Recept oraz wykorzystywanych Podfunkcjonalności.



Rysunek 4.2: Cykl życia podfunkcjonalności

4.3 Aplikacja applFY

Aplikacja applFY stanowi interfejs graficzny biblioteki - umożliwia dostęp do listy dostępnych i aktywnych Recept, ustalanie ich parametrów przy włączaniu i wyłączaniu. Umożliwia korzystanie z zasobów Targowiska. Zapewnia możliwość zarządzania grupami użytkowników korzystając z API wystawionego przez Serwer Grup. W Aplikacji rozszerzono funkcje modułu obsługi Recept - możliwe jest wykorzystywanie tych pobranych z Targowiska i zapisywanie stanu aktywnych recept w bazie danych, co było potrzebne ze względu na mechanizm zarządzania zasobami w systemie Android - każdy proces może zostać zatrzymany w dowolnej chwili, dlatego potrzebny był sposób na odtworzenie stanu Aplikacji sprzed zamknięcia. Zmodyfikowany cykl życia Recepty przedstawia rysunek 4.3.



Rysunek 4.3: Cykl życia recepty w applFY

4.4 Targowisko

Podrozdział 2.2 definiuje zbiór wymagań pozafunkcjonalnych. Wymagania bezpośrednio dotyczące recept takie jak PF12 czy PF13 są w praktyce niemożliwe do zrealizowania na aplikacji mobilnej. W związku z powyższym postanowiono stworzyć aplikację pomocniczą

realizującą w praktyce pierwsze trzy fazy cyklu życia recepty w ramach określonych przez wymagania pozafunkcjonalne.

Pierwsze trzy fazy cyklu życia recepty to kolejno:

1. Pisanie kodu
2. Kompilacja i budowa
3. Dystrybucja

Pierwsze dwa punkty powyższej listy można zrealizować poprzez dostarczenie zintegrowanego środowiska programistycznego. Trzeci, poprzez stworzenie aplikacji internetowej korzystającej z bazy danych. Powstało więc Targowisko – aplikacja internetowa przypominająca nieco Google Play, z dodatkowymi funkcjami. Pierwszą z nich jest stworzenie zintegrowanego środowiska programistycznego wbudowanego bezpośrednio w Targowisko. Drugą, zastosowanie idei rozwidlania Recept.

4.4.1 Zintegrowane środowisko programistyczne

Pomysł zakładał osadzenie edytora na stronie internetowej. Strona ta posiadałaby jednocześnie mechanizmy umożliwiające kompilację i budowę kodu recepty. Oczekiwanym plikiem powstawałym na wyjściu tego procesu byłoby archiwum Java (*jar*) gotowe do wgrania na aplikację kliencką `if{y}`.

Okazuje się, że istnieją darmowe edytory kodu nadające się do umieszczenia na stronie. Co więcej, wywoływanie komend systemu operacyjnego z poziomu aplikacji internetowej również okazało się w pełni możliwe. Wybrano więc napisany w JavaScript edytor Ace – z racji na najlepszą dokumentację oraz język PHP który świetnie radzi sobie również z zadaniami niskiego poziomu.

4.4.2 Rozwidlanie

Pierwotne założenie dotyczące budowania nowych recept zakładało istnienie generatora kodu. Miał on na celu umożliwienie wygenerowania części kodu w zależności od wybranych przez użytkownika, podfunkcjonalności z biblioteki `if{y}`. Generator taki był dyskusyjny z powodu pracochłonnej implementacji oraz specyfiki biblioteki. Generator powinno implementować się po ostatecznej implementacji biblioteki. Biblioteka jednak z założenia jest kodem który dynamicznie rozwija się w czasie.

Ostatecznie, zamiast generatora, postanowiono wykorzystać ideę rozwidlania (ang. fork). Idea ta w praktyce oferuje prawie to samo co generator kodu. Istnieje jednak jedna spora różnica. Rozwidlanie pielęgnuje się samo w sobie. Wystarczy stworzyć jeden działający kawałek kodu i opublikować go. Od tego momentu, każdy może zacząć pisanie swojego kodu przyjmując jako punkt wyjścia kod opublikowany wcześniej. Proces ten może powtarzać się rekursywnie w nieskończoność.

Stosowane podejście nie jest wolne od wad. Kluczową może być fakt, iż działający kod niskiej jakości może być niepotrzebnie propagowany. Propagacja może się też dotyczyć wysokiej jakości kodu, przez co problem propagacji w ogólności można uznać za mało szkodliwy.

4.4.3 Repozytorium recept

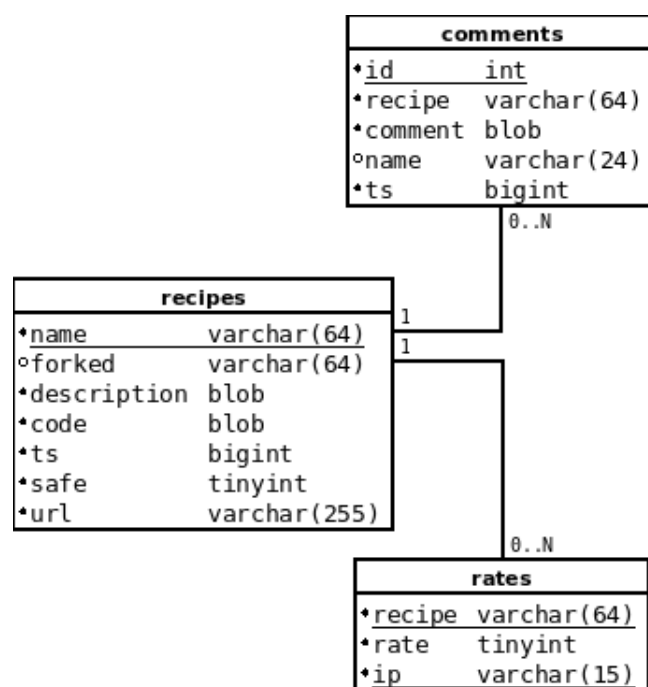
Zarządzanie receptami postanowiono zrealizować angażując strukturę katalogową targowiska oraz bazę danych. Pliki ze zbudowanym kodem zgromadzono w katalogu *jar*. Wszelkie informacje pomocnicze umieszczono w bazie danych o schemacie widocznym na rysunku 4.4, gdzie:

- *recipes* – jest to relacja której każda krotka utożsamiana jest z pojedynczą receptą. Każda recepta posiada własną, unikalną nazwę, opcjonalnie nazwę recepty z której dana została wywiedziona, opis, kod źródłowy, znacznik czasu dodania, flagę informującą o niebezpiecznych konstrukcjach w kodzie oraz łącze do pliku *jar*.
- *comments* – jest to relacja której krotki reprezentują komentarze użytkowników na temat recept. Każdy komentarz posiada unikalny identyfikator liczbowy, nazwę recepty której się dotyczy, treść komentarza, opcjonalnie nazwę autora oraz znacznik czasu dodania.
- *rates* – jest to relacja której krotki reprezentują oceny w całkowitej skali od 1 do 5 przyznawane receptom. Na każdą ocenę składa się nazwa recepty ocenianej, wartość całkowitoliczbowa oceny oraz adres IP oceniającego.

Stworzono także interfejs programowania aplikacji (API). Ma on na celu umożliwienie dostępu do targowiska z poziomu aplikacji klienckiej `if{y}`. Interfejs ten pozwala na uzyskanie całej zawartości bazy danych w formacie JSON.

4.5 Serwer Grup

Głównym zadaniem Serwera Grup jest przekazywanie wiadomości umożliwiających komunikację między klientami i wymianę danych. Informacje te 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. Rozwiązaniem tego problemu mogła być usługa Google Cloud Messaging (GCM), lecz nie spełniała jednych z założeń projektu o odseparowaniu aplikacji od sieci społecznościowych i istniejących serwisów. Kolejna koncepcja opiera się o tak zwany mechanizm odpytywania (ang. *polling*), który jest prosty do zaimplementowania, a jednocześnie spełnia wszystkie wymagania stawiane w projekcie - odbieranie danych z serwera wykonywane poprzez cykliczne zapytania eliminuje problem łączności między aplikacją a pośredniczącym serwerem. Innym z pomysłów było wykorzystanie protokołu MQTT (MQ Telemetry Transport), który jest jednak trudny w implementacji i prawdopodobnie zwiększyłby liczbę używanych bibliotek, nie przynosząc większych korzyści w porównaniu z odpytywaniem. Istotnym aspektem w wymianie danych między użytkownikami są ograniczenia do komunikacji aby niemożliwe było otrzymanie wiadomości od niezidentyfikowanego użytkownika. Rozwiązaniem jest połączenie użytkowników w grupy, w obrębie których będą mogli rozsyłać wiadomości. W związku z tym serwer musi także umożliwiać zarządzanie grupami.



Rysunek 4.4: 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 przesyłania 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.

5.1.5 Deaktywacja

Deaktywacją recepty również zajmuje się serwis. Polega ona na usunięciu wyrejestrowaniu jej z Podfunkcjonalności, co powoduje, że nie dostanie ona powiadomienia o zdarzeniach, a następnie usunięciu jej z listy dostępnych recept. Dodatkowo są odinicjalizowane podfunkcjonalności, z których nie korzysta żadna inna recepta. Ich usuwanie z listy odbywa się w drugim przebiegu pętli, aby zabezpieczyć się przed wyjątkiem `ConcurrentModificationException`.

```

public void disableRecipe(Integer id) {
    YRecipe recipe = mActiveRecipesManager.get(id);

    List<Long> toDelete = new ArrayList<Long>();
    for (Entry<Long, YFeature> entry : recipe.getFeatures()) {
        YFeature feat = entry.getValue();
        YLog.d("SERVICE", "UnregisterRecipe: " + recipe.getName()
            + " from " + entry.getKey());
        feat.unregisterRecipe(recipe);
        if (!feat.isUsed()) {
            toDelete.add(entry.getKey());
        }
    }
}

```

```

        YLog.d("SERVICE", "UninitializeFeature: " + feat.getId());
        feat.uninitialize();
    }
}
mActiveFeatures.removeAll(toDelete);
mActiveRecipesManager.remove(id);
}

```

5.1.6 Recepty Grupowe

Specyficzną grupą recept są recepty grupowe. Wyróżniają się one używaniem podfunkcjonalności `YGroupFeature`, co pozwala im się komunikować z innymi urządzeniami. Do komunikacji służy klasa `YComm`, którą należy zainicjować w metodzie `init()`:

```

private YComm comm;
@Override
public void init() {
    comm = getFeatures().getGroup()
        .createPoolingComm(this, getParams().getString("GROUP"), 5);
}

```

Obecnie jedynym sposobem na inicjalizację jest użycie metody `createPoolingComm`, podając jej jako argumenty receptę, nazwę grupy, której dotyczy oraz okres czasu między kolejnymi odpytywaniami serwera o nowe zdarzenia. Nic jednak nie stoi na przeszkodzie, aby rozwijając bibliotekę umożliwić tworzenie obiektów klasy `YComm` opartych na innych rozwiązaniach komunikacyjnych niż odpytywanie.

Gdy obiekt `YComm` zostanie utworzony, recepta automatycznie odbiera zdarzenia.

Aby wysłać zdarzenie należy użyć metod `sendEvent` lub `broadcastEvent`. Pozwalają one wyzwoić zdarzenie odpowiednio na konkretnym urządzeniu (na podstawie nazwy użytkownika) lub na wszystkich należących aktualnie do grupy. Metody te mają kilka wersji różniących się parametrami. Dzięki temu można wysłać samo zdarzenie bez danych, jedną zmienną dołączoną do zdarzenia, lub też cały zbiór. Parametrem który zawsze występuje jest tag - liczbowy identyfikator typu zdarzenia, który można później sprawdzić w kodzie jego obsługi, co pozwala stworzyć kilka typów komunikatów bez potrzeby dodawania do nich danych. Jego wartość powinna być dodatnia - ujemne są zarezerwowane dla zdarzeń systemowych.

W reakcji na odebranie zdarzenia wywoływana jest metoda `handleEvent` z parametrem typu `YGroupEvent`, zawierającego obiekt typu `YCommData`, który umożliwia odczyt taga, danych oraz informacji o nadawcy.

Oprócz wysyłania zdarzeń można też po prostu wysyłać dane na serwer. Służy do tego metody `sendVariable` oraz `sendVariables` wysyłające odpowiednią jedną zmienną lub ich zbiór. Aby je odczytać należy użyć metody `getVariables`, pobierającej dane konkretnego użytkownika lub `getAllVariables`, pobierającej dane całej grupy. Odpowiedź przychodzi do Recepty w formie zdarzenia.

Dane przysyłane między receptami są typu `YParam` - ta sama klasa jest używana do parametrów, jednak potrzeby są tutaj takie same - jest to klasa opakowująca obiekty

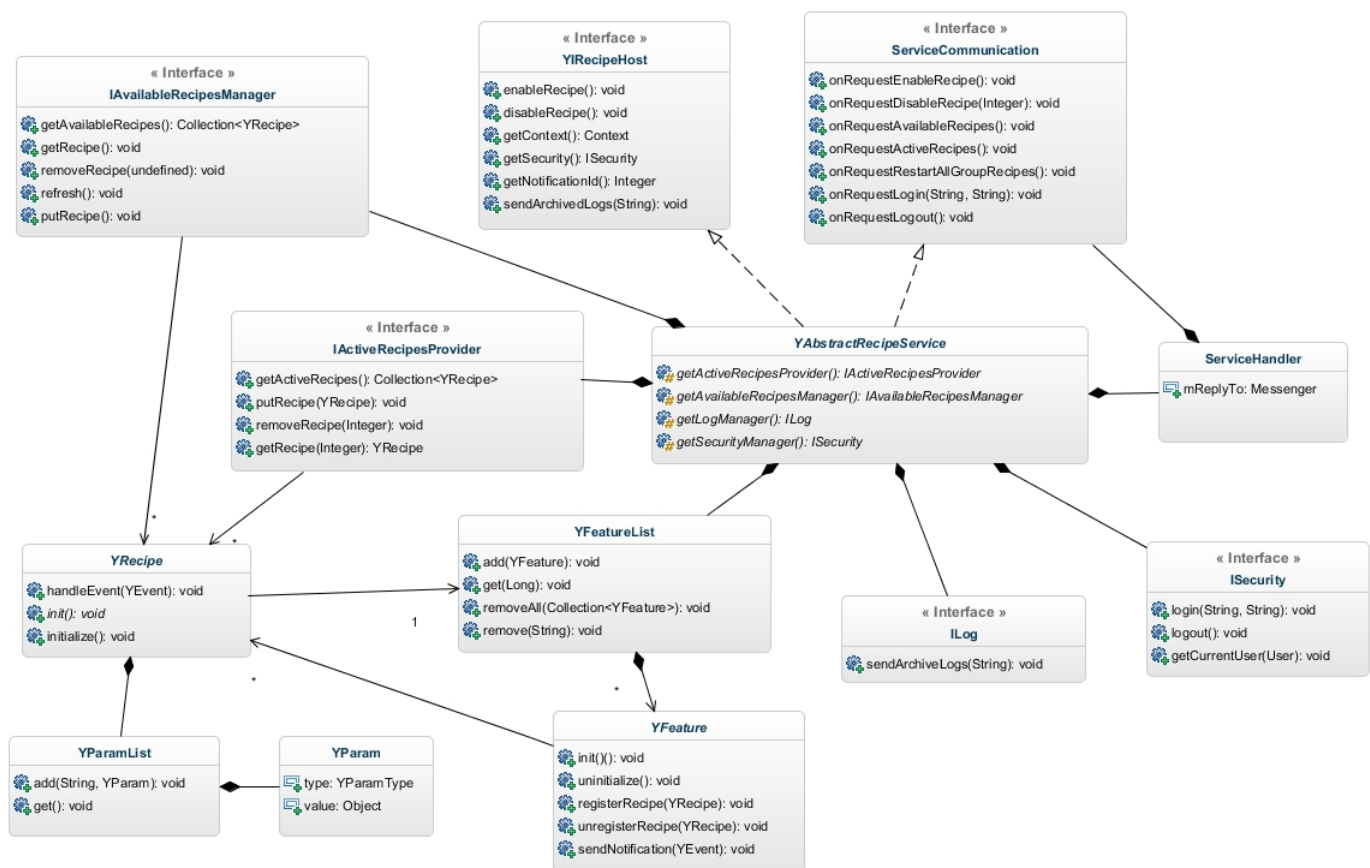
różnych typów, łatwo konwertowalna do ciągu znaków i zawierająca informacje o przechowywanym typie.

Zbiory danych zapisywane są w formie słownika, gdzie kluczem jest nazwa zmiennej (String), a wartością zmienna (YParam).

Odczyt danych odbywa się poprzez metodę `getData` na obiekcie typu `YComm`, przyjmującym jako argument nazwę zmiennej. W przypadku zdarzeń będących odpowiedzią na metody `getVariables` i `getAllVariables` należy użyć wersji dwuargumentowej, z dodatkowym argumentem określającym właściciela zmiennej. Innym sposobem dostępu jest całego słownika z danymi metodą `getValues`.

5.2 Biblioteka

5.2.1 Zarządzanie receptami



Rysunek 5.1: Diagram klas modułu zarządzania receptami

Moduł zarządzania Receptami został zaimplementowany w klasie abstrakcyjnej `YAbstractRecipeService`, dziedziczącej po `Serwisie`. Serwis w Androidzie to komponent aplikacji przeznaczony do długotrwałego wykonywania operacji w tle, nieposiadający interfejsu użytkownika. [6] Zaleca się uruchamianie Serwisu Biblioteki w osobnym procesie, dzięki czemu uruchomione Recepty mogą działać niezależnie od Aplikacji. Warto w tym miej-

scu nawiązać do zarządzania pamięcią w systemie Android - każda aplikacja uruchomiona przez użytkownika może zostać w dowolnym momencie zamknięta ze względu na konieczność zwolnienia zasobów. Serwis Biblioteki został zaimplementowany w taki sposób, aby być z powrotem uruchamiany po zamknięciu. Możliwe jest także uruchamianie go po starcie systemu. Podczas projektowania tego modułu kluczowe było zapewnienie maksymalnego uproszczenia mechanizmów zarządzania Receptami z punktu widzenia programisty Aplikacji przy jednoczesnej dowolności implementacji list dostępnych i aktywnych Recept. Pozwoliło na stworzenie appIFY, w której użytkownik ma pełną kontrolę nad modułem zarządzania, jak i wykorzystanie Biblioteki do napisania aplikacji wykorzystującej jedną wbudowaną Receptę, niewidoczną dla użytkownika końcowego.

5.2.2 Podfunkcjonalności

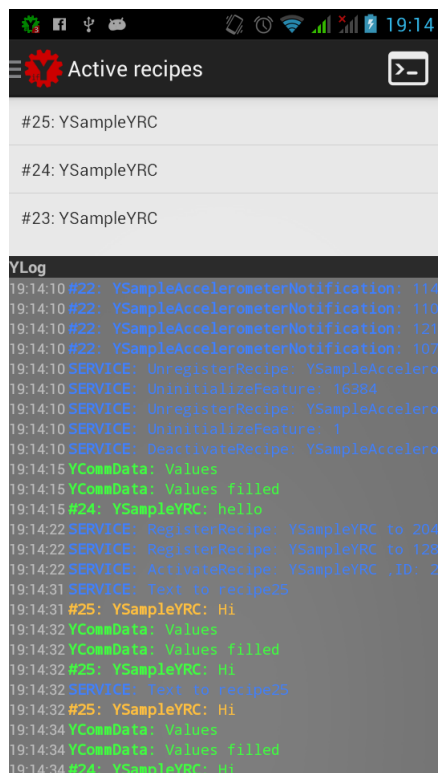
Podfunkcjonalności to klasy agregujące pewne funkcje związane z systemem. Muszą być inicjalizowane, gdy znajdzie taka potrzeba i przechwytywać zdarzenia systemowe, przekazując je odpowiednim receptom. Klasą bazową jest dla nich YFeature. Są w niej zaimplementowane metody związane z czasem życia Podfunkcjonalności i Recepty, odpowiedzialne za rejestrowanie i odrejestrowywanie Recept, inicjalizację i deinicjalizację Podfunkcjonalności oraz sprawdzanie, czy Podfunkcjonalność jest używana przez recepty. Poza tym znajduje się w niej metoda odpowiedzialna za wysyłanie zdarzenia do Recept - `sendNotification`, wykorzystywana w poszczególnych Podfunkcjonalnościach. Zaimplementowano następujące podfunkcjonalności:

- Akcelerometr (YAccelerometerFeature.java)
Umożliwia reagowanie na odczyty akcelerometru wbudowanego w urządzenie.
- AudioManager (YAudioManager.java)
Umożliwia zarządzanie poziomem głośności dzwonka.
- Battery (YBatteryFeature.java)
Umożliwia reagowanie na zmiany poziomu baterii urządzenia.
- Calls (YCallsFeature.java)
Umożliwia reagowanie na połączenia przychodzące i inicjowanie połączeń wychodzących.
- Files (YFilesFeature.java)
Umożliwia tworzenie i odczytywanie plików z pamięci urządzenia lub nośnika zewnętrznego.
- Geocoder (YGeocoderFeature.java)
Umożliwia pobranie adresu związanego z podaną długością i szerokością geograficzną.
- GPS (YGPSFeature.java)
Umożliwia śledzenie pozycji urządzenia za pomocą modułu GPS.
- Group (YGroupFeature.java)
Niezbędny do obsługi zdarzeń grupowych.

- Intent (YIntentFeature.java)
Pozwala wysyłać intencje[8] umożliwiające m. in. uruchamianie innych aplikacji.
- Internet (YInternetFeature.java)
Umożliwia wysyłanie i pobieranie danych z podanego adresu.
- Notification (YNotificationFeature.java)
Umożliwia wyświetlanie powiadomień w interfejsie graficznym urządzenia.
- RawPlayer (YRawPlayerFeature.java)
Umożliwia odtwarzanie dźwięków na podstawie tablicy częstotliwości.
- Shortcut (YShortcutFeature.java.java)
Pozwala na tworzenie skrótów do Recepty na głównym ekranie.
- SMS (YSMSFeature.java)
Umożliwia wysyłanie wiadomości SMS oraz reagowanie na wiadomości przychodzące.
- Sound (YSoundFeature.java)
Pozwala odtwarzać pliki dźwiękowe.
- Text (YTextFeature.java)
Umożliwia wprowadzanie tekstu do recepty z poziomu aplikacji.
- Time (YTimeFeature.java)
- Wifi (YWifiFeature.java)
Umożliwia włączanie i wyłączanie modułu WiFi urządzenia.

5.2.3 Dziennik

Biblioteka posiada moduł dziennika, pozwalającego na zapisywanie i wyświetlanie informacji. Dostęp do niego miał przypominać jak najbardziej klasę Log [9] w Androidzie. Powstała zatem klasa YLog zawierająca między innymi statyczne metody v,d,i,w,e, identyczne jak w Androidzie. Wywołują one swoje odpowiedniki, aby umożliwić debugowanie poprzez narzędzie logcat. Oprócz tego są wpisy są przechowywane wewnętrznie. Do ich wyświetlania służy specjalny widok, wyświetlany jako nakładka. Nie reaguje on na dotyk, przez co umożliwia użytkowanie telefonu gdy jest widoczny. Jest to jedyny widok obsługiwany przez Bibliotekę, a nie Aplikacje Klientkie. Dodatkowo istnieje klasa YLogger, której instancje są przypisane do recepty. Wpisy przez nią utworzone są przypisane do konkretnej Recepty. Dzięki temu można je łatwo odfiltrować i wyświetlać na ekranie Recepty w Aplikacji Klientkiej. Recepty mają do niej dostęp poprzez pole Log – jego nazwa wydaje się przeczyć konwencji, jednak przypomina dzięki temu oryginalną klasę Log ze statycznymi metodami. Jediną różnicą są argumenty metod – zamiast taga i wiadomości przyjmowana jest jedynie wiadomość, tag jest tworzony automatycznie na podstawie Recepty.



Rysunek 5.2: Widok dziennika nad ekranem Aplikacji Klientkiej

5.3 Aplikacja kliencka

5.3.1 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.

5.3.2 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 do załadowania kodu recepty (nazwa klasy oraz ścieżka dostępu do pliku .jar) przechowywane są w bazie danych recept pobranych na urządzenie.

5.3.3 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 serwisu 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 ustanowić 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 wysyланemu 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 YAbstractService 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

Targowisko zaimplementowano w formie aplikacji internetowej. Łączy ono w sobie technologie takie jak PHP, MySQL, HTML, CSS oraz JavaScript. Jedyną różnicą w stosunku do przeciętnych stron internetowych jest integracja z systemem operacyjnym poprzez skrypty powłoki BASH.

Targowisko ma trzy cele:

1. Zapewnić prosty w użyciu interfejs użytkownika.
2. Udostępnić zunifikowany interfejs programowania aplikacji (API) który pozwoli uzyskać dostęp do wszystkich informacji z bazy danych.
3. Dać dostęp do skryptów powłoki BASH oraz zapewnić im bezpieczne wykonanie.

W kontekście celów oraz technologii, jako wzór postępowania przyjęto wzorzec projek-

towy Model-Widok-Kontroler (MVC). Z racji na swoje organizacyjne właściwości, jest on najlepszym wyborem pozwalającym oddzielić logikę od interfejsu.

5.4.1 Wzorzec MVC

```
scony@scony-netbook:~/Inzynierka$ tree marketify/ -L 1
marketify/
├── api
├── api.php
├── app
├── bootstrap
├── CHMOD.md
├── css
├── doc
├── examples.php
├── explore.php
├── fonts
├── fork.php
├── futureIdeas.md
├── gfx
├── index.php
├── jar
├── js
├── Makefile
├── mock
├── README.md
├── recipe.php
├── sh
├── sql
├── tpl
└── upload.php

13 directories, 11 files
```

Rysunek 5.3: Struktura katalogowa targowiska

Poprzez użycie Wzorca MVC w targowisku, uniemożliwiono technologiom mieszanie się ze sobą. Zjawisko takie jest częstym błędem popełnianym przez twórców aplikacji internetowych. Utrudnia ono w znacznym stopniu rozwój oraz pielęgnowanie kodu. Konsekwencje użycia wzorca MVC, widać najwyraźniej w strukturze katalogowej. Prezentuje ją rysunek 5.3. W głównym katalogu targowiska znajdują się pliki o rozszerzeniu *php*. Są to kontrolery przetwarzające dane pochodzące od użytkownika. Pozostałe kontrolery znajdują się w podkatalogu *api*. Mają one na celu przetworzenie żądań kierowanych do API.

Widoki znajdują się w katalogu *tpl*. Ich poprawne działanie jest jednak gwarantowane poprzez style oraz skrypty zgromadzone kolejno w podkatalogach *css* oraz *js*.

Modele znaleźć można w podkatalogach *app* oraz *sql*.

5.4.2 Interfejs graficzny

Do realizacji graficznego interfejsu użytkownika użyto darmowy framework; Twitter Bootstrap. Jest to obszerny zbiór stylów CSS wraz z rzetelną dokumentacją. Pozwala on osobie nie będącej uzdolnionej artystycznie, tworzyć dobrze wyglądające widoki aplikacji.

Framework jest bardzo prosty w użyciu. Po pierwsze, należy dołączyć jego pliki do struktury katalogowej swojej aplikacji. Po drugie, do kodu HTML należy dodać następującą linię:

```
<link href="./css/bootstrap.css" rel="stylesheet">
```

Aby przykładowo utworzyć efektywne menu, do znacznika UL będącego kontenerem dla hiperłączy wystarczy dodać odpowiednie klasy tak jak w przykładzie poniżej:

```
<ul class="nav nav-pills pull-right">
<li><a href="explore.php">Recipes</a></li>
<li><a href="examples.php">Examples</a></li>
<li><a href="upload.php">Upload</a></li>
<li><a href="doc" target="_blank">Doclava</a></li>
<li><a href="api.php">API</a></li>
</ul>
```

Dla lepszego wyobrażenia możliwości oferowanych przez framework, wystarczy spojrzeć na rysunek 5.4.

5.4.3 Edytor Ace

W celu umożliwienia pisania kodu recept z poziomu targowiska, skorzystano z darmowego edytora Ace. Jest to narzędzie dające możliwości znane z popularnych narzędzi takich jak Vim czy Eclipse (rysunek 5.5). Napisano je w JavaScript, przez co osadzenie go we własnej aplikacji internetowej ogranicza się do modyfikacji wyłącznie widoków.

Kod edytora należy najpierw dołączyć do kodu widoku. Można to zrobić dwojako; poprzez dołączenie plików edytora do struktury katalogowej lub poprzez dołączenie kodu edytora ze zdalnej lokalizacji. W targowisku skorzystano z drugiej możliwości:

```
<script src="http://ace.c9.io/build/src-min/ace.js" type="text/javascript">
</script>
<script src="http://ace.c9.io/build/src-min/ext-language_tools.js" type="text/javascript">
</script>
```

Pierwsze dwie linie tyczą się samego edytora. Pozostałe dwie dołączają rozszerzenie używane w targowisku celem uzyskania autouzupełniania widocznego na rysunku 5.5.

Samo dołączenie kodu nie wystarczy do działania edytora. W treści strony trzeba dodać znacznik *pre* o odpowiednim parametrze id:

```
<pre id="editor"></pre>
```

Na koniec wykonać należy prosty skrypt JavaScript który uruchomi edytor wraz z rozszerzeniem:

```
var langTools = ace.require("ace/ext/language_tools");
var editor = ace.edit("editor");
editor.setOptions({enableBasicAutocompletion: true});
editor.setTheme("ace/theme/twilight");
```

Recipe repository

Our repository contains all recipes that ever existed. You can access all of them in order to download or to fork

Rate	Name	Description
4.00	YSampleEmptyRecipe	Sample recipe doing nothing
4.00	YSampleGroupSMS	Sample recipe forwarding all received SMS to group and showing as notifications and saves to logs.
3.00	YSampleFindFriend	Sample recipe tracking position of people in group and sending notification to people who are near. Uses GPS, Group and Notification.
3.00	YSampleWifiOffWhenLowBattery	Sample recipe turning off WiFi connection when battery is low.
1.00	YSampleAccelerometerNotification	Sample recipe reading data from Accelerometer and showing notification if it's low enough.
1.00	YSampleAccelerometerSMS	Sample recipe reading data from Accelerometer and Sending SMS if it's high enough.
1.00	YSampleCalls	Simple recipe, that discards all incoming calls.
1.00	YSampleCallsSMS	Simple recipe, that discards all incoming calls and sends SMS to caller.
1.00	YSampleGPSGeocoderSMS	Recipe activated by receiving SMS with given text (MESSAGE param) reading location from GPS, converting it to address and sending SMS.
1.00	YSampleRawPlayerAccelerometer	Sample recipe using YRawPlayer play sounds with frequency based on YAccelerometr

Rysunek 5.4: Framework Twitter Bootstrap zastosowany do widoku listy recept tar-gowiska

Code

```
1 import pl.poznan.put.cs.ify.api.*;
2 import pl.poznan.put.cs.ify.api.exceptions.*;
3 import pl.poznan.put.cs.ify.api.features.*;
4 import pl.poznan.put.cs.ify.api.features.events.*;
5 import pl.poznan.put.cs.ify.api.group.*;
6 import pl.poznan.put.cs.ify.api.log.*;
7 import pl.poznan.put.cs.ify.api.params.*;
8 import pl.poznan.put.cs.ify.api.security.*;
9 import pl.poznan.put.cs.ify.api.types.*;
10 import pl.poznan.put.cs.ify.api.Y;
11 import pl.poznan.put.cs.ify.api.YEvent;
12 import pl.poznan.put.cs.ify.api.YRecipe;
13 import pl.poznan.put.cs.ify.api.features.YSMSFeature;
14 import pl.poznan.put.cs.ify.api.features.events.YAccelerometerEvent;
15 import pl.poznan.put.cs.ify.api.params.YParamList;
16 import pl.poznan.put.cs.ify.api.params.YParamType;
17
18 /**
19  * Sample recipe reading data from Accelerometer and Sending SMS if it's high enough.
20  */
21 public class YAccelerometerSMS extends YRecipe {
22
23     //flag preventing from sending multiple SMS
24     private boolean alreadySend = false;
25
26     @Override
27     public long requestFeatures() {
28         return Y.SMS | Y.Accelerometer;
29     }
30
31     @Override
32     public void requestParams(YParamList params) {
33         //String param with phone number of SMS recipient
34         params.add("SEND_TO", YParamType.Number, "");
35         //Integer param with value of squared acceleration which triggers recipe
36     }
37 }
```

Submit 

Rysunek 5.5: Edytor Ace oferuje między innymi kolorowanie składni oraz autouzupełnianie

```

editor.session.setMode("ace/mode/java");
var ifyCompleter = {
  getCompletions: function(editor, session, pos, prefix, callback) {
    $.getJSON(
      "./api/completer.php" + (prefix.length === 0 ? "" : "?prefix=" + prefix),
      function(wordList) {
        callback(null, wordList.map(function(ea) {
          return {name: ea.word, value: ea.word, score: ea.score, meta: "if{y}"}
        }));
      });
  }
}
langTools.addCompleter(ifyCompleter);

```

5.4.4 API

Jako alternatywą dla interfejsu użytkownika metodę dostępu do danych, w targowisku zaimplementowano interfejs API. Jest on dedykowany aplikacjom na urządzenia mobilne. Pracuje w trybie tylko do odczytu, dzięki czemu nie występuje ryzyko ingerencji w dane.

Działanie API opiera się o wywołania GET protokołu HTTP. Dane będące wynikiem wywołania zwracane są w formie wielowymiarowej tablicy zapisanej w formacie JSON. API targowiska wyróżnia pięć możliwych żądań:

- O najnowsze recepty
 - `http://ADRES_TARGOWISKA/api/new.php?limit={limit}`
 - `{limit}`, liczba naturalna – ilość recept do pobrania
- O wszystkie recepty
 - `http://ADRES_TARGOWISKA/api/recipes.php?page={page}&limit={limit}`
 - `{page}`, liczba naturalna – numer strony z której zostaną pobrane recepty
 - `{limit}`, liczba naturalna – ilość recept przypadających na każdą stronę
- O recepty których nazwy pasują do wzorca
 - `http://ADRES_TARGOWISKA/api/search.php?phrase={phrase}`
 - `{phrase}`, łańcuch znaków – wzorzec wyszukiwania
- O konkretną receptę
 - `http://ADRES_TARGOWISKA/api/recipe.php?name={name}`
 - `{name}`, łańcuch znaków – pełna nazwa recepty
- O losową receptę
 - `http://ADRES_TARGOWISKA/api/random.php`

Przykładowa odpowiedź targowiska na żądanie o losową receptę może wyglądać następująco:

[

```

{
  "name":"YSampleYRC",
  "forked":null,
  "description":"Simple IRC - like chat, using YTextEvent, YGroupFeature and YLog.",
  "code":"(...)",
  "ts":1391197304,
  "safe":1,
  "url":"http:\\\\ADRES_TARGOWISKA\\jar\\YSampleYRC.jar",
  "rate":null,
  "comments":[

  ]
}
]

```

5.4.5 Kompilacja Recept

Po dodaniu recepty do bazy danych Targowiska jest ona kompilowana oraz przekształcana do wynikowego pliku *jar* poprzez następujący skrypt powłoki BASH:

```

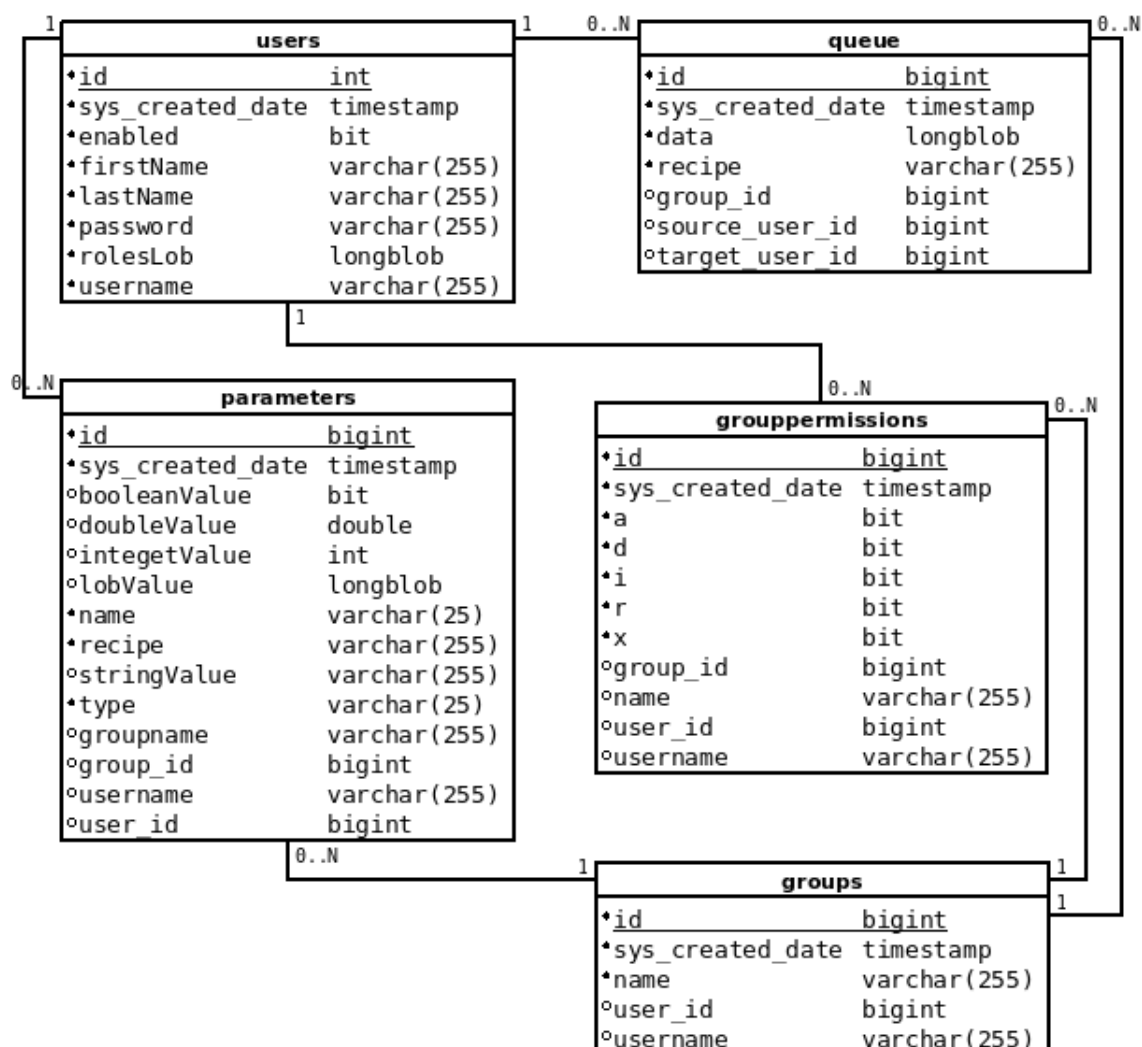
#!/bin/bash

if [ $# -eq 1 ];
then
  # clean
  rm -f recipe.jar
  # build
  javac -source 1.6 -target 1.6 -bootclasspath rt.jar -cp ify.jar:android.jar:rt.jar $1.java
  || exit 1 # compability with jdk 1.6
  # javac -cp ify.jar:android.jar $1.java || exit 1
  jar cf tmp.jar $1.class
  ./dx --dex --output=classes.dex tmp.jar
  jar cf recipe.jar $1.class classes.dex
  # post-clean
  rm -f *.class
  rm -f classes.dex
  rm -f tmp.jar
  rm -f *.java
  exit 0
else
  echo 'Usage: ./build.sh [recipe class name without .java]'
  exit 2
fi

```

5.5 Serwer recept grupowych

5.5.1 Baza danych



Rysunek 5.6: Schemat bazy danych serwera grup

5.5.2 Grupy i użytkownicy

Łączenie użytkowników w grupy realizowane jest za pomocą relacji w bazie danych. Każda z grup jest opisana przez swoją unikalną nazwę dzięki której może być w łatwy sposób wyszukana, przy próbie utworzenia grupy o tej samej nazwie zostanie zwrócony wyjątek a z poziomu interfejsu błąd walidacji. Tabele reprezentujące grupy i użytkowników połączone są za pomocą tabeli pośredniej "groupsperrmissions" w której zapisane są informacje o uprawnieniach jakie posiada użytkownik w grupie. Uprawnienia nadawane są dla każdej grupy indywidualnie. Każdy użytkownik w grupie posiada prawo do wypisywania listy użytkowników w grupie oraz tak prawo do wykonywania czyli możliwość wysyłania wiadomości do innych użytkowników w ramach tej grupy oraz zapisywanie własnych danych na

serwerze. Twórca grupy dodatkowo posiada wpis z uprawnieniem do dodawania nowych członków.

5.5.3 Uwierzytelnianie i autoryzacja

Po odebraniu każdej wiadomości od użytkownika jest on uwierzytelniany za pomocą loginu i hasła podanych w przypadku połączeń typu HTTP w JSONie lub dla połączeń typu REST w ścieżce URL. W celu zapobiegnięcia przesyłaniu haseł w postaci jawnej, szyfrowane są one funkcją skrótu SHA-1.

Autoryzacja użytkownika odbywa się poprzez sprawdzenie uprawnień użytkownika do grupy którą podał w danych do komunikacji. Jako pierwsze sprawdzane jest czy użytkownik jest członkiem grupy, w ramach której wysła wiadomość. Jeżeli użytkownik nie należy do grupy nie posiada żadnych uprawnień w ramach tej grupy. Dostępnych jest 5 uprawnień jakie może posiadać użytkownik dla każdej z grup do której należy: Uprawnienia reprezentowane są przez flagi:

- d (ang. delete) pozwala usuwać użytkowników z grupy
- a (ang. add) pozwala dodawać nowych użytkowników
- r (ang. read) pozwala odczytywać członków grupy oraz nie widzi grupy na swojej liście grup do których należy.
- x (ang. execute) umożliwia wysyłanie i odbieranie wiadomości w ramach grupy.
- i (ang. invitation) oznacza że użytkownik został zaproszony do grupy i nie potwierdził zaproszenia. Jeżeli ustawiona jest ta flaga pozostałe uprawnienia traktowane są jak by nie były ustawione. Użytkownik może tylko potwierdzić lub odrzucić zaproszenie do grupy.

Dla zapytań typu REST w przypadku niewwierzytelnionej, nieautorysowanej komunikacji lub w przypadku błędu zwracana jest wartość "false". Przy komunikacji z użyciem protokołu HTTP zostaje wysłana odpowiedź ze statusem błędu.

5.5.4 Kolejka komunikatów

Serwer pośrednicząc w komunikacji odbiera wiadomości, które do czasu wysłania do klienta docelowego są zapisywane w bazie danych. Wiadomość w bazie jest w formie zserializowanej do ciągu bajtów. Z każdej wiadomości która zostanie odebrana pobierane są informacje niezbędne do zapisania jej oraz późniejszego zidentyfikowania takie jak nazwy grupy, recepty, użytkownika docelowego i nadawcy.

5.5.5 Zarządzanie grupami

5.5.6 Komunikacja z receptami

Komunikacja recept z serwerem grup odbywa się za pomocą odpowiedniego zapytania do serwera:

- Adres docelowy: ADRES_SERWERA/rest/recipe

- Typ zapytania: POST
- Content-Type: application/json

W ciele zapytania znajduje się JSON o następującej strukturze:

```
{
  "user": {
    "username": STRING,
    "recipe": STRING,
    "group": STRING,
    "password": STRING
  },
  "event": {
    "target": STRING,
    "tag": INT
  },
  "values": {
    STRING: {
      "value": OBJECT,
      "type": STRING
    },
    STRING: {
      "value": OBJECT,
      "type": STRING
    },
    [...]
  }
}
```

Oczywiście wyrazy pisane wersalikami to typy danych, których wartości są zależne od konkretnego zapytania.

Pole user ma na celu identyfikację nadawcy wiadomości - zawiera on nazwę użytkownika, recepty i grupy. Znajduje się tutaj także hasło, a ściślej - skrót SHA-1 hasła używający loginu jako soli.

Pole event to login użytkownika do którego skierowany jest komunikat lub wartość specjalna BROADCAST oznaczająca skierowanie do wszystkich oraz tag określający typ zdarzenia.

Pole values to wartości przesłane wraz ze zdarzeniem. Mają formę słownika indeksowanego ciągami znaków. Każdy obiekt ma typ i wartość. Typy są tożsame z typami zdefiniowanymi w klasie YParamType, przy czym przy przesyłaniu JSON-en wartości typów innych niż Integer i Boolean są zamieniane na String.

Taki schemat danych odpowiada podstawowej operacji - sendEvent. Tag jest wówczas dodatni. Pozostałe operacje jednak posługują się tym samym schematem z drobnymi różnicami:

- sendValues
Tag jest równy stałej YCommand.SEND_DATA = 0. Pole event.target jest ignorowane.

wane. Nazwy tych danych różnią się od tych określonych w receptce - mają formę nazwa@użytkownik, co zapewnia ich unikalność kluczy przy pobieraniu przez `getAllValues`. Oczywiście znak @ nie może być częścią loginu użytkownika, co jest weryfikowane przy rejestracji.

- `getValues`
Tag jest równy stałej `YCommand.GET_DATA = -1`. Pole `event.target` to użytkownik, którego dane nas interesują. W przypadku pustej wartości serwer zwraca dane całej grupy (operacja `getAllValues`). Pole `values` jest ignorowane.
- `poll`
Tag jest równy stałej `YCommand.POOLING = -4`. Jest to techniczna operacja odpytywania serwera o zdarzenia kierowane do recepty. Pola `event.target` oraz `values` są ignorowane.

Odpowiedzi na zapytania, jeśli wymagają przesłania danych są przekazywane w podobnej formie. W wypadku zdarzeń pole `user` identyfikuje nadawcę. Oczywiście jest z niego usuwane pole `password`. Pola `event` i `values` są takie, jak w wiadomości tworzącej zdarzenie. Przy odpowiedziach na `GET_DATA` pole `values` jest wypełnione wartościami danych zapisanymi na serwerze. Pola `user` i `event` są kopią tych wysłanych, jednak ich wartość jest ignorowana.

5.6 Użyte technologie

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

- **Android**
System operacyjny z rodziny Linux przeznaczony dla urządzeń mobilnych. Aktualnie rozwijane przez sojusz biznesowy Open Handset Alliance.
- **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.
- **Apache Commons**
- **Apache HTTP Server**
- **Git**
Rozproszony oraz wieloplatformowy system kontroli wersji będący wolnym oprogramowaniem.
- **HTML 5**

- **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.
- **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.
- **Java**
- **JavaScript**
Skryptowy język oprogramowania stosowany na stronach internetowych.
- **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.
- **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
- **PHP**
Obiektowy język programowania dedykowany generowaniu stron internetowych w czasie rzeczywistym. Szczególnie użyteczny w przypadku tworzenia prototypów tużdzień niewielkich projektów wymagających stosunkowo niskiego poziomu abstrakcji.
- **RESTeasy**
Framework oprogramowania służący do tworzenia aplikacji rozproszonych, oparty na wzorcu architektury oprogramowania Representational State Transfer (REST).
- **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).
- **Vaadin**
Framework sieciowy służący do tworzenia aplikacji sieciowych w szczególności interfejsu użytkownika w oparciu o Google Web Toolkit (GWT) w języku JAVA.
- **JUnit**
Biblioteka służąca do tworzenia testów jednostkowych w języku Java.

5.7 Użyte narzędzia

- Apache Tomcat
Kontener aplikacji sieciowych.
- Eclipse
Popularne zintegrowane środowisko programistyczne (IDE) wspierające głównie język Java (wtyczki pozwalają obsługiwać inne języki).
- 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.
- String Tool Suite
Zintegrowane środowisko programistyczne oparte o Eclipse dostosowany do Spring-Framework.
- Emacs
Popularny, w pełni rozszerzalny edytor tekstowy spotykany głównie w systemach operacyjnych z rodziny Unix.
- Git bash for windows
Narzędzie umożliwiające używanie Gita z linii poleceń w systemie Windows poprzez wbudowane środowisko MinGW.
- 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.
- Latex
- Linux
Rodzina systemów operacyjnych będących wolnym oprogramowaniem oraz używających jądra Linux.
- Notepad++
Prosty edytor tekstowy umożliwiający kolorowanie składni w wielu językach.
- Przeglądarki internetowe
Google Chrome, Mozilla Firefox, Opera
- Windows

5.8 Urządzenia mobilne

Aplikacja była testowana na następujących urządzeniach mobilnych:

- LG Swift GT540
Procesor: Qualcomm MSM7227 600 MHz Pamięć RAM: 256 MB System operacyjny: Android 4.0.1 (Cyanogen mod)
- Media-Droid IMPERIUS EN3RGY MT7013 Procesor: dwurdzeniowy, 1GHz ARM7 MTK6577 Pamięć RAM: 256 MB System operacyjny: Android 4.1.2
- Motorola Defy MB525
Procesor: TI OMAP3610 800 MHz Pamięć RAM: 512 MB System operacyjny: Android 4.3.1 (Cyanogen mod)
- Sony LT18 Xperia Arc S
Procesor: Qualcomm MSM8255T 1,40 GHz Pamięć RAM: 512 MB System operacyjny: Android 4.0.4
- Samsung Galaxy Mini GT-S5570
Procesor: Qualcomm MSM7227 600 MHz Pamięć RAM: 384 MB System operacyjny: Android 2.2

5.9 Opis pakietów

5.9.1 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.

5.9.2 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.

5.9.3 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
- 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.

Wyniki

6.1 Cykl życia recepty w praktyce

W efekcie ukończenia prac nad wszystkimi modułami platformy, możliwym stało się odtworzenie cyklu życia recepty w praktyce. Użytkownik kompetentny do pisania kodu w języku Java, jest od tego miejsca w stanie tworzyć recepty ograniczając się wyłącznie interfejsem biblioteki oraz własną wyobraźnią.

W dalszej części rozdziału, przedstawiona zostanie recepta informująca o wzajemnej bliskości osób należących do grupy oraz przeanalizowany zostanie każdy etap jej cyklu życia.

6.1.1 Pisanie kodu

if{y}

RecipesExamplesUploadDocumentationAPI

YSampleEmptyRecipe

★★★★☆

Sample recipe doing nothing

Download

Fork

```
1 import pl.poznan.put.cs.ify.api.*;
2 import pl.poznan.put.cs.ify.api.exceptions.*;
3 import pl.poznan.put.cs.ify.api.features.*;
4 import pl.poznan.put.cs.ify.api.features.events.*;
5 import pl.poznan.put.cs.ify.api.group.*;
6 import pl.poznan.put.cs.ify.api.log.*;
7 import pl.poznan.put.cs.ify.api.params.*;
8 import pl.poznan.put.cs.ify.api.security.*;
9 import pl.poznan.put.cs.ify.api.types.*;
10 import pl.poznan.put.cs.ify.api.Y;
11 import pl.poznan.put.cs.ify.api.YEvent;
12 import pl.poznan.put.cs.ify.api.YRecipe;
13 import pl.poznan.put.cs.ify.api.params.YParamList;
14 import pl.poznan.put.cs.ify.api.params.YParamType;
15
16 /**
17  * Sample recipe doing nothing
18  */
19 public class YSampleEmptyRecipe extends YRecipe {
20     @Override
21     public void requestParams(YParamList params) {
22         params.add("Level", YParamType.Integer, 90);
23     }
24
25     @Override
26     protected void init(){
27         // It's called when recipe is enabled
28         // Can be used for some internal initialization after creating recipe.
29         // Initialize your internal fields here.
30     }
31
32     @Override
33     public long requestFeatures() {
34         // Return bitmask with ID's of features used by recipe.
35         // Use ID's from Y class and OR (|) operator.
36         return Y.Battery | Y.Wifi;
```

Date	Name	Comment
08.02.14 17:17:16	Guest	Very useful useful scratch

Comment

Name

Submit

Rysunek 6.1: Widok przykładowej recepty w targowisku

Pierwszym krokiem do napisania kodu własnej recepty jest zapoznanie się z przykładami dostępnymi na stronie targowiska w dziale *examples*. Przykładowa recepta może wyglądać jak na rysunku 6.1. Każda przykładowa recepta to działający kod zawierający przydatne informacje zawarte w komentarzach. Gdy komentarze i przykłady nie wystarczą, sięgnąć należy do dokumentacji dostępnej w targowisku w dziale *documentation*.

Po nabyciu niezbędnej wiedzy, najprościej zacząć od rozwidlenia gotowej recepty. Sprowadza się to do kliknięcia przycisku *fork* z poziomu widoku recepty oraz wprowadzenia nowej nazwy. Alternatywną drogą, jest rozpoczęcie pisania recepty od podstaw. W tym celu należy przejść do działu targowiska o nazwie *upload*.

Pisanie kodu odbywa się w edytorze który szerzej opiany został w rozdziale 5.5.3. W celu wywołania okna autouzupełniania należy użyć skrótu klawiszowego *CTRL+SPACJA*.

6.1.2 Kompilacja i budowa

Po zakończeniu pisania kodu wystarczy nacisnąć przycisk *submit* i odczekać kilka sekund.

if{y}
Recipes
Examples
Upload
Documentation
API

Error: can not process your code

YFindFriend.java:70: error: cannot find symbol
Strign user = data.getUserData().getId();
^
symbol: class Strign
location: class YFindFriend
1 error

Share your code

If you'd like to prepare your own recipe, you can write it down right here

Description

Sample recipe tracking position of people in group and sending notification to people who are near. Uses GPS, Group and Notification.

Code

```

59      data.put("position", pos);
60      data.put("time", time);
61
62      // sends created data to group
63      comm.broadcastEvent(1, data);
64  }
65  if (event.getId() == Y.Group) {
66      YGroupEvent ge = (YGroupEvent) event;
67      YCommData data = ge.getData();
68
69      // get sender of message
70      Strign user = data.getUserData().getId();
71
72      if (user.equals(comm.getMyId())) {
73          return; // Message from myself, ignore that.
74      }
75
76      YPosition otherPos = (YPosition) data.getData("position")
77          .getValue();
78      Integer otherTime = (Integer) data.getData("time").getValue();
79
80      // only do anything if we know our position
81      if (mLastPos != null) {
82          // calculate distance in meters to other person
83          float dist = otherPos.getDistance(mLastPos);
84          boolean isNear = dist < mParams.getInteger("Range");
85
86          // user was already near but is no more...
87          if (mAlreadyNear.contains(user) && !isNear) {
88              //...so remove him
89              mAlreadyNear.remove(user);
90          }
91          // user wasn't near but now he is
92          else if (!mAlreadyNear.contains(user) && isNear) {
93              int currentTime = (int) (System.currentTimeMillis() / 1000);

```

Submit

Rysunek 6.2: Błąd w trakcie kompilacji kodu recepty

W przypadku niepowodzenia etapu kompilacji czy budowy, użytkownik dostaje pełną informację o błędzie jak na rysunku 6.2. Po przeanalizowaniu komunikatu o błędzie, użytkownik może wprowadzić niezbędne zmiany oraz ponownie nacisnąć przycisk *submit*.

W przypadku poprawnego przetworzenia kodu recepty, użytkownik zostaje przekierowany bezpośrednio do jej widoku.

6.1.3 Dystrybucja

Zbudowana poprawnie recepta jest natychmiastowo zarówno w targowisku jak i module targowiska aplikacji if{y} w urządzeniu mobilnym. Aby recepta zdobyła popularność i uznanie wśród społeczności, warto aby jej kod był możliwie wysokiej jakości oraz z dużą ilością komentarzy.

O sukcesie recepty decydują oceny nadawane przez użytkowników oraz pozostawione przez nich komentarze.

6.1.4 Pobranie na urządzenie mobilne

Aby pobrać receptę na urządzenie mobilne należy wybrać w appIFY zakładkę Market. Pojawi się wówczas lista dostępnych recept TODO-OBRAZEK. Przy niektórych pojawiają się czaszki – oznacza to, że odwołują się do bibliotek Androida lub mechanizmu refleksji i są potencjalnie niebezpieczne – mogą uzyskać dostęp do innych funkcji telefonu niż wynika z zadeklarowanych podfunkcjonalności. Po wybraniu recepty wystarczy użyć przycisku Download.

6.1.5 Uruchomienie

Aby uruchomić receptę należy wybrać zakładkę Available recipes, a następnie wybrać receptę z listy. Pojawi się okno w należy określić parametry, a następnie użyć przycisku Start. W przypadku recept grupowych użytkownik musi być zalogowany aby uruchomić receptę.

6.1.6 Działanie

Po aktywacji recepty działa ona w tle, nasłuchując zdarzeń i reagując na nie. Działanie recepty kończy się po wybraniu zakładki *Active recipes* z listy i kliknięciu przycisk *Disable*. Praca recepty może się też zakończyć automatycznie w wyniku wyrzucenia przez nią wyjątku lub wylogowania się użytkownika w przypadku recepty grupowej.

Zakończenie

7.1 Podsumowanie

7.2 Dalszy rozwój (platformy)

Recipe creation guide

Best way to create recipe is forking existing one from the market - just choose one from Examples or Recipes tab. Then you can edit it's code in browser. You don't have to care about ify imports - they will be added automatically when uploading recipe. More information is available in Doclava tab in market.

Currently market is available here:

<http://ify.cs.put.poznan.pl/scony/marketify>

You can access some fields and methods generated by system which can be useful:

`getParams()`

List of params requested in `requestParams`. It's populated with values when `init()` and `handleEvent()` code is called. Use `getString(String name)`, `getInteger(String name)`, etc. to get param with given name depending what type param is.

`getFeatures()`

Contains required features. Use specified getters like `getSMS()`, `getWifi()`, etc. or use general method `get(Long id)` inserting id from Y and cast returned feature to right type.

`Log`

Use if to create logs with methods `wtf(String)`, `e(String)`, `w(String)`, `i(String)`, `d(String)`, `v(String)`. These logs will be visible on recipe screen and saved to Android logs (use Log-Cat to see it)

Here is quick overview of methods you need to implement in your recipe, you can find same information in `YEmptyRecipe` example.

`protected void init()`

It's called when recipe is enabled, can be used for some internal initialization after creating

recipe. Initialize your internal fields here.

```
public long requestFeatures()
```

Return bitmask with ID's of features used by recipe. Use ID's from Y class and OR ('|') operator.

```
public void requestParams(YParamList params)
```

Populate params object with definitions of needed params using params.add(String paramName, YParam param)

```
protected void handleEvent(YEvent event)
```

It's place for main recipe logic. It's called on event connected with requested feature occurs. Check what event is that by checking event.getId() and compare it to id's from Y.

```
public String getName()
```

Return String same like your recipe class name here. It's needed to avoid using reflection which is slow.

```
public YReceipt newInstance()
```

Just call default constructor and return recipe instance. Also needed to avoid reflection.

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
- [8] Android API Guide - Intents and Intent Filters <http://developer.android.com/guide/components/intents-filters.html> Ostatnio odwiedzone 31.01.2014
- [9] Android API Guide - Reading and Writing Logs <http://developer.android.com/tools/debugging/debugging-log.html> Ostatnio odwiedzone 31.01.2014
- [10] Introducing Google Play: All your entertainment, anywhere you go - <http://googleblog.blogspot.com/2012/03/introducing-google-play-all-your.html> Ostatnio odwiedzone 31.01.2014