Programowanie reaktywne na przykładzie platformy Android z użyciem biblioteki RxJava

Reactive programming on the example of the Android platform using the RxJava library

Nie sposób zaprzeczyć, iż jednym z najszybciej rozwijających się obszarów w świecie IT jest obszar budowania interfejsów aplikacji. Podejście programistów zarówno do interfejsów webowych, jak i mobilnych cały czas ewoluuje, a coraz to nowsze biblioteki i frameworki wymuszają ciągłą potrzebę poznawania nowych rozwiązań. Jednym z kluczowych aspektów budowania interfejsu użytkownika jest możliwość zapanowania nad zdarzeniami, które wywołuje użytkownik aplikacji. Z pomocą przychodzi podejście reaktywne, które łączy w sobie paradygmat programowania funkcyjnego i takie wzorce projektowe jak obserwator i iterator. Następujący rozdział opisuje elementy, z których składa się programowanie reaktywne jak i jego zastosowanie. W dalszej części przedstawia także proces budowania interfejsu użytkownika w platformie Android z użyciem podejścia reaktywnego.

# Programowanie reaktywne

W tej części rozdziału przedstawione zostaną teoretyczne pojęcia, które kryją się pod pojęciem programowania reaktywnego. Raczej każdy programista rozpoczyna swoją przygodę z programowaniem od paradygmatu imperatywnego, w którym kroki w programie są realizowane w sposób sekwencyjny. Taki sposób programowania mocno odbiega od podejścia reaktywnego, które na pierwszy rzut oka może wydawać się złożone. Samo pojęcie nie jest jednak niczym nowym i istnieje już od pewnego czasu w świecie IT. Istota programowania reaktywnego to próba odzwierciedlenia otaczającego nas świata, który zbudowany jest z asynchronicznych zdarzeń. Takie same asynchroniczne zdarzenia zachodzą także w interfejsie użytkownika jak i w całej aplikacji. Takimi zdarzeniami w aplikacji mogą być: pobranie danych z pliku/serwera, wypełnienie pola tekstowego, naciśnięciu przycisku. Podejście reaktywne pozwala nam zapanować nad asynchronicznymi zdarzeniami za pomocą paradygmatu funkcyjnego i kilku wzorców projektowych. Funkcyjne operacje na danych zapewniają nam zwięzłość w sposobie manipulacji nimi a wzorzec obserwator zapewnia wygodny sposób reagowania na zdarzenia występujące w aplikacji.



## Wzorzec projektowy obserwator

Jednym z najważniejszych elementów programowania reaktywnego jest wzorzec obserwator (ang. Observer Pattern). Jest to wzorzec typu behawioralnego, który odpowiada za nasłuchiwanie czy też obserwacje jakiegoś zdarzenia. Głównymi podmiotami tego wzorca jest obiekt obserwowany oraz obiekty go obserwujące. Pomiędzy podmiotami zachodzi, więc relacja jeden-do-wielu. Jeżeli obiekt, który jest obserwowany zmieni swój stan to wszystkie obiekty, który zdecydowały się na obserwowanie go zostaną powiadomione o tym fakcie.

Jednym najprostszych przykładów może być stacja pogodowa, która po otrzymaniu nowych danych, może automatycznie powiadomić o aktualizacji swojego stanu takich obserwatorów jak strona internetowa czy wyświetlacz. Podstawowym sposobem implementacji tego wzorca jest zaprojektowanie dwóch interfejsów, które odpowiedzialne są za deklaracje odpowiednich metod (1). Aby jeszcze lepiej zrozumieć budowę tego wzorca poniżej zaprezentowany został diagram klas wzorca obserwator.

### Schemat wzorca projektowego obserwator

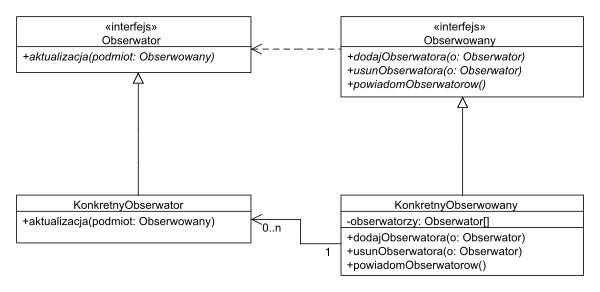


Diagram klas wzorca obserwator (2)

Wyróżnić należy dwa podstawowe typy obiektów, które składają się na strukturę tego wzorca:

* Obserwowany (ang. Observable, Subject) – obiekt, który odpowiedzialny jest za publikowanie zmian do wszystkich swoich subskrybentów.
* Obserwator (ang. Observer, Listener) – obiekt, który oczekuje na powiadomienie opublikowane przez obiekt obserwowany.

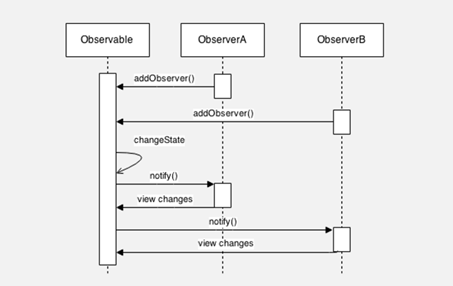


Diagram sekwencji wzorca obserwator (3)

Nowe obiekty, które zdecydują się na obserwacje mogą zostać dodane w dowolnym momencie. Obserwujący obiekt może zaprzestać obserwowania, lecz może zostać także usunięty listy obserwatorów w obiekcie obserwowanym.

Język Java posiada własną implementacje wzorca projektowego obserwator (java.util.Observable). Niestety w tym przypadku jest to klasa a nie interfejs, co znacząco ogranicza jego zastosowanie. Aby skorzystać z gotowej implementacji należy utworzyć klasy podrzędne, co skutkuje tym, że nie można dodać zachowania klasy Observable do innej już istniejącej (1).

### Wady i zalety

Jedną z największych zalet tego wzorca projektowego jest osłabienie zależności pomiędzy obiektami, które obserwują obiekt obserwowany. Skutkuje to odpowiednią izolacją, ponieważ obserwowany i obserwator nie muszą mieć dużej wiedzy o sobie. Jedyną informacją, jaką posiada obserwowany podmiot o obiektach go obserwujących jest fakt, że implementują one odpowiedni interfejs (1). Niestety nie można uzależnić aplikacji od określonej kolejności nadchodzących powiadomień. Powiadomienie może także dotrzeć do obserwatora zbyt późno.

## Wzorzec projektowy iterator

Jego celem stworzenie jest jednolitego interfejsu, który zapewnia sekwencyjny dostęp do obiektów znajdujących się w danej kolekcji bez ujawniania ich wewnętrznej reprezentacji (1).



### Schemat wzorca projektowego iterator



Wzorzec ten składa się z:

* Dwóch klas abstrakcyjnych: Agregat i Iterator
* Dwóch klas konkretnych: AgregatKonkretny i IteratorKonkretny

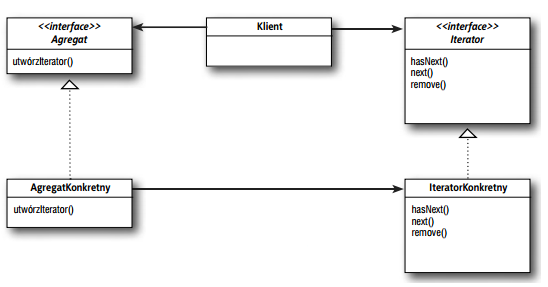


Diagram klas wzorca iterator (1)

Wszystkie kolekcje stworzone za pomocą tego wzorca implementują interfejs Aggregate. Klient, który odwoła się do metody CreateIterator(), otrzymuje klasę, która implementuje interfejs Iterator. Skutkuje to tym, że klient zna tylko interfejs, do, którego musi się odwołać. Dokładnie taki scenariusz działania zachodzi w obrębie biblioteki Collections w języku Java. Każda kolekcja posiada swój własny iterator, lecz dostęp do niego realizowany jest za pomocą wspólnego interfejsu Iterator (4). Iteratory posiadają metody, które pozwalają na sekwencyjny dostęp do wszystkich elementów kolekcji. W niektórych implementacjach pozwala on także na modyfikację kolekcji.

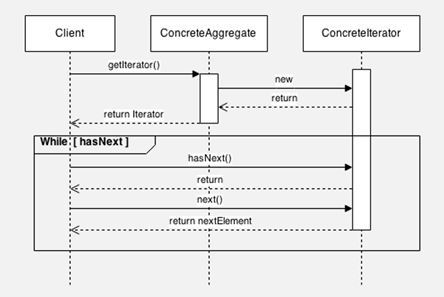


Diagram sekwencji wzorca iterator (5)

Język Java posiada domyślną implementacje wzorca projektowego Iterator z metodami: *hasNext*, *next*, *remove* i *forEachRemaining*. Poniższy przykład ilustruje użycie wbudowanego iteratora celu wyświetlenia elementów przechodzonej kolekcji na standardowe wyjście programu (4).



Przykład użycia wbudowanego iteratora w języku Java (6)

### Po słowie

W wielu językach programowania powszechna stała się konstrukcja for-each, która tak naprawdę w niejawny sposób wywołuje Iterator. Dowodzi to, że wzorzec Iterator jest już niejako standardem zaimplementowanym praktycznie w każdym języku programowania (7).

## Biblioteka ReactiveX

Jest to biblioteka, która łączy w sobie zastosowanie zarówno wzorców obserwator i iterator, a ponadto zawiera też implementacje elementów paradygmatu funkcyjnego. Połącznie tych trzech głównych składowych pozwala na manipulacje danymi w bardziej deklaratywny sposób, co skutkuje wysokim poziomem abstrakcji w zakresie wielowątkowości, synchronizacji danych jak i operacjami, które mogą blokować działania aplikacji. Biblioteka ta została zaimplementowania w wielu językach programowania takich jak:.Net, Java, JavaScript, Scala czy Swift (8). W przypadku projektu FindMyTutor została wykorzystana implementacja w języku Java nosząca nazwę RxJava.



### Zastosowania wzorca obserwator w bibliotece RxJava

W bibliotece RxJava wyróżnić możemy następujące typy bytów obserwowanych:

|  |  |
| --- | --- |
| Typ obserwowanego obiektu (ang. Observable) | Opis |
| Flowable<T> | Emituje on od 0 do N elementów. Może zakończyć swoje działanie w dwojaki sposób, zwracając sukces lub błąd. Wspiera sprzężenie danych (ang. Backpressure), co skutkuje możliwością kontrolowania jak szybko dane zostaną opublikowane. |
| Observable<T> | Bardzo podobny do Flowable, lecz nie wspiera sprzężenia danych. |
| Single<T> | Emituje pojedynczą wartość lub błąd. Przydatny podczas tworzenia serwisów HTTP. Przykładowo zapytanie GET może zwrócić wynik zapytania lub błąd. |
| Maybe<T> | Kończy swoje działanie emitując jeden ze stanów: pojedynczy element, brak elementu lub błąd. |
| Completable<T> | Kończy swoje działanie zwracając sukces lub błąd. Nigdy nie emituje elementów. Przykładem zastosowania może być obsługa zapytania POST w serwisie HTTP. |

Opis poszczególnych typów obserwatorów w bibliotece RxJava (9)

Sama biblioteka dostarcza też wiele metod za pomocą, których można utworzyć różne rodzaje obserwatorów. Warto zaznaczyć jednak, że nie wszystkie podane niżej metody są dostępne dla typów wyszczególnionych w powyższej tabeli. Przykładowo metoda fromArray dostępna jest jedynie dla typów: Flowable i Observable (10).

|  |  |
| --- | --- |
| Metoda | Opis |
| Observable.just() | Pozwala na utworzenie obiektu obserwowanego po przez opakowanie innego typu danych. |
| Observable.fromIterable() | Jako argument oczekuje typu java.lang.Iterable<T> i emituje elementy tego typu z zachowaniem kolejności elementów. |
| Observable.fromArray() | Jako argument przyjmuje tablicę i emituje jej zawartość z zachowaniem kolejności elementów. |
| Observable.fromCallable() | Pozwala na utworzenie obiektu obserwowanego z typu java.util.concurrent.Callable<V>. |
| Observable.fromFuture() | Pozwala na utworzenie obiektu obserwowanego z typu java.util.concurrent.Future. |
| Observable.interval() | Tworzy obiekt obserwowany, który emituje nieskończoną sekwencje obiektów typu Long w podanym odstępie czasowym. |

Opis poszczególnych metod umożliwiających tworzenie obserwatorów w

bibliotece RxJava (9)

W celu utworzenia obserwatora należy posłużyć się operatorem Subscribe, który odpowiedzialny jest za połączenie ze sobą obserwatora i obiektu obserwowanego. Wyróżnia się cztery podstawowe typy obserwatorów: Observer, SingleObservable, MaybeObservable i CompletableObserver (11). Ich powiązanie z obiektem obserwowanym zostało zaprezentowane w poniższej tabeli.

|  |  |
| --- | --- |
| Typ obiektu obserwowanego | Typ obserwatora |
| Observable | Observer |
| Single | SingleObserver |
| Maybe | MaybeObserver |
| Flowable | Observer |
| Completable | CompletableObserver |

Typy obiektów obserwowanych wraz z typami obserwatorów (11)

Prawie każdy typ obiektu obserwującego musie zawierać implementacje metod wyszczególnionych w poniższej tabeli.

|  |  |
| --- | --- |
| Metoda obserwatora (ang. Subscriber) | Opis |
| onNext() | Metoda ta zostaje wywołana, gdy obserwator wyemituje element. |
| onError() | Obserwator wywoła tę metodę, gdy nastąpi błąd podczas emitowania danych. Wywołanie tej metody zatrzymuje dalsze przetwarzanie kolejnych elementów i skutkuje brakiem wywołań pozostałych metod: onNext i onComplete. |
| onComplete() | Metoda ta zostanie wywołana, gdy po raz ostatni zostanie wywołania metoda onNext i niewywołana została metoda onError w trakcie emitowania danych. |

Podstawowe metody obserwatorów dostępne w bibliotece RxJava (11)

## Elementy paradygmatu funkcyjnego w programowaniu reaktywnym

Najważniejszym komponentem, z którego zbudowany jest paradygmat funkcyjny są funkcje. W podejściu tym nie występują pętlę ani zmienne. Zamiast tego posługiwać musimy się zdefiniowanymi stałymi i rekurencją. Jednymi z największych zalet tego paradygmatu są funkcje wyższego rzędu, strumienie oraz funktory. W rezultacie otrzymany kod zawiera mniej efektów ubocznych i jest bardziej przewidywalny w działaniu. To z tej dziedziny pochodzą takie pojęcia jak czysta funkcja (ang. Pure Function) lub niezmienne dane (ang. Immutable Data). Programując funkcyjnie odpowiadamy raczej na pytanie: „Co chcemy osiągnąć?” (Jest to paradygmat deklaratywny), a nie „W jaki sposób chcemy to osiągnąć?” (To pytanie zadajemy sobie programując w paradygmacie imperatywnym). Poznając programowanie reaktywne bardzo często spotykamy się z takimi funkcjami wyższego rzędu jak: map, filter czy fold, które znajdują swoje odzwierciedlenie w programowaniu reaktywnym (12).



### Czyste funkcje

Jako czystą funkcje należy rozumieć funkcję, która nie wywołuje efektów ubocznych tj. może wpływać na otoczenie tylko po przez swój wynik a wynik ten jest tylko i wyłącznie zależny od danych wejściowych (ang. Referential Transparency). Jako efekty uboczne można rozumieć m.in. wyświetlenie tekstu na ekran, wysłanie danych do serwera etc. Efekty uboczne, które mogą wystąpić podczas użycia funkcji, która nie jest czysta mogą powodować sprzężenia i zależności czasowe, co skutkuje niepoprawnym działaniem aplikacji.

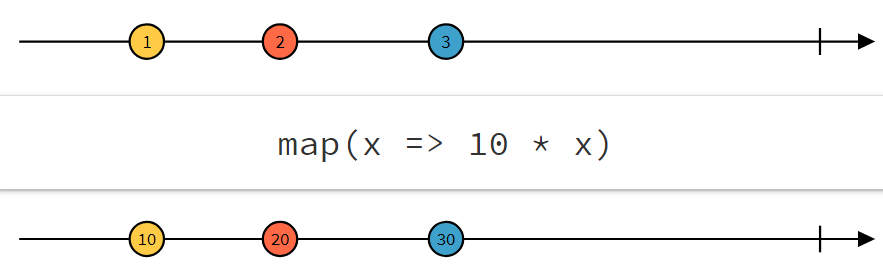
Czyste funkcje są łatwiejsze w testowaniu, ponieważ ich wyjście zależy tylko od danych wejściowych. Ponadto może je składać i są łatwiejsze do zrównoleglania (13). Oczywiście niemożliwym jest operowanie tylko czystymi funkcjami, ponieważ prawie każdy program wypisuje dane na ekran, co samym w sobie jest efektem ubocznym. Należy jednak dążyć do jak największej liczby takich funkcji w programie. Efekty uboczne mogą zostać zniwelowane za pomocą innych narzędzi i definicji matematycznych takich jak na przykład Monady (14).

### Funkcje wyższego rzędu

Innym podstawowym mechanizmem używanym w programowaniu funkcyjnym jest mechanizm funkcji wyższego rzędu (ang. Higher order function). Funkcja wyższego rzędu to funkcja, która przyjmuje, jako argument inną funkcję lub zwraca funkcję. Można wyróżnić kilka rodzajów takich funkcji:

* Map

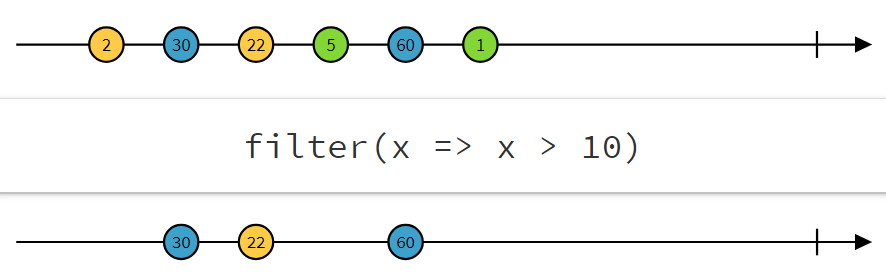
Ten typ funkcji wyższego rzędu przyjmuje, jako argument funkcję i listę. Podczas przechodzenia po liście na każdym jej elemencie aplikowana jest funkcja podana w argumencie. W rezultacie wynikiem funkcji map jest nowa lista, która składa się z elementów zwróconych przez funkcje, która została zaaplikowana (12).



Obrazowe przedstawienie działania funkcji map (15)

* Filter

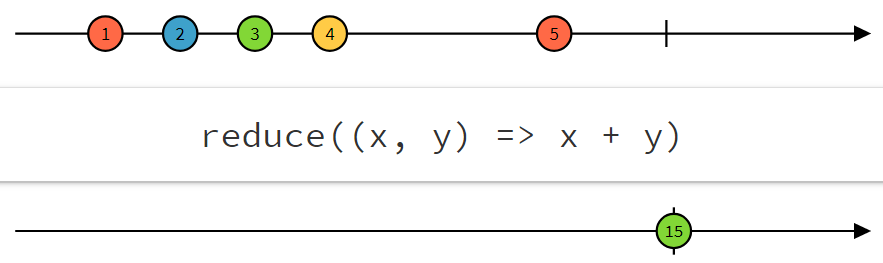
Podobnie jak w funkcji map argumentami są funkcja i lista. Jednak w tym przypadku, jako argument musi zostać podana funkcja boolowska (funkcja, która zwraca tylko wartość true lub false). Funkcja zostaje zaaplikowana do każdego elementu z listy, aby następnie zwrócić elementy, dla których została zwrócona wartość true (12).



Obrazowe przedstawienie działania funkcji filter (16)

* Reduce

Funkcja reduce przyjmuje, jako argumenty: funkcję, wartość początkową i listę. Jej zadaniem jest obliczenie pewnej wartości na podstawie wszystkich elementów listy. Funkcja podana, jako argument operuje na wartości obliczonej do n-tego elementu listy i jej następnym elemencie. Przykładowym zastosowaniem funkcji reduce może być obliczenie sumy wszystkich elementów listy (12).



Obrazowe przedstawienie działania funkcji reduce (17)

### Po słowie

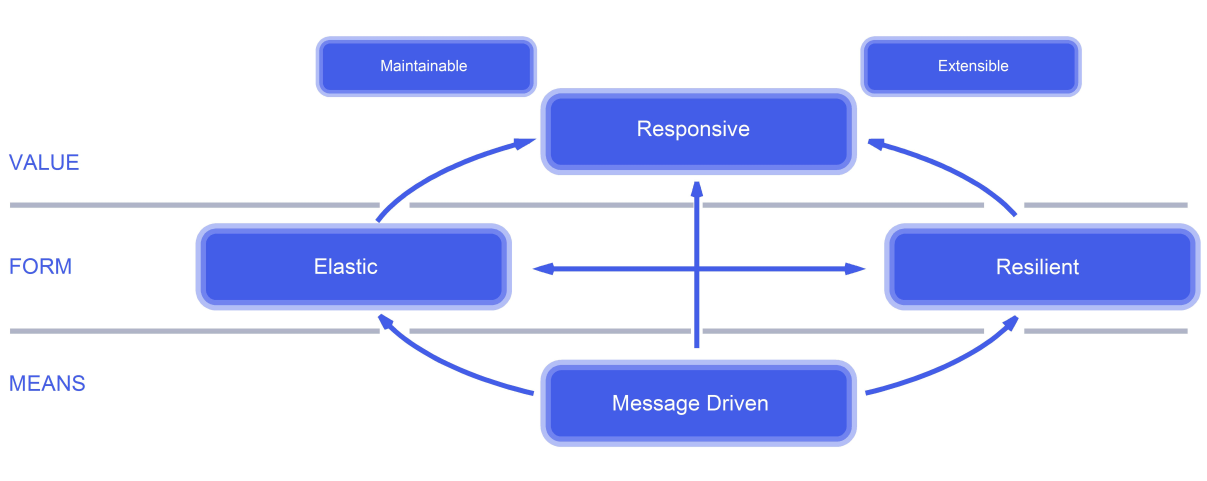
Implementacja funkcji wyższego rzędu może zależeć od danego języka programowania. Przykładowo w języku JavaScript funkcje te są zdefiniowane w prototypie danego obiektu, co skutkuje tym, że zbędny jest argument z listą, po której operujemy. Z racji tego, że funkcje wyższego rzędu zwracają nową listę możliwe jest ich składania, przez co otrzymany kod jest jeszcze bardziej przejrzysty. Kod napisany z użyciem paradygmatu funkcyjnego w większości przypadków jest bardziej zwięzły od kodu napisanego w paradygmacie imperatywnym, co skutkuje mniejszą złożonością i ułatwia jego zrozumienie przez programistę (18).

## The Reactive manifesto

Nie sposób zaprzeczyć, iż wymagania, które twórcy oprogramowania stawali przed sobą jeszcze kilka lat temu uległy drastycznej zmianie. Nie tak dawno temu istniały aplikacje, w których czas odpowiedzi serwera był liczony w sekundach a liczba godzin niedostępności aplikacji była wyrażana w godzinach. W dzisiejszym świecie, w którym aplikację uruchamiane są na niezliczonej ilości różnych urządzeń a użytkownik oczekuje czasu odpowiedzi serwera w milisekundach i dostępności serwisu 24 godziny na dobę, wymagania stawiane przed twórcami oprogramowania uległy znaczącej zmianie. The reactive manifesto to próba odpowiedzenia na problemy, z którymi musi zmierzyć się system informatyczny w dzisiejszych czasach.

System reaktywny to system, który jest:

* + Responsywny – system reaguje tak szybko jak to tylko możliwe. Responsywność jest podstawą użyteczności. Ponadto pozwala ona na szybkie wykrycie potencjalnych błędów. Systemy reaktywne zapewniają szybki i niezmienny czas reakcji, który buduje zaufanie użytkownika systemu (19).
  + Odporny – system pozostaje responsywny nawet w przypadku wystąpienia błędów. Odporność uzyskiwana jest za pomocą takich pojęć jak: replikacja, izolowanie czy delegowanie. Wystąpienie błędu w jednym z komponentów systemu nie może skutkować pojawieniem się błędu w innych obszarach (19).
  + Elastyczny – system jest w stanie dostosować swoje działanie do obciążenia. Jest on w stanie zarządzać swoimi zasobami w celu odpowiedzenia na napotkane zapotrzebowanie (19).
  + Sterowany wiadomościami (ang. Message driven) – systemy reaktywne opierają się na asynchronicznym przekazywaniu danych w celu ustalenia granic pomiędzy danymi komponentami. Skutkuje to izolacją i przejrzystością ich lokalizacji. Ustalenie granic pomiędzy komponentami systemu umożliwia przekazywanie błędów, jako wiadomości. Skutkuje to możliwością zarządzania obciążeniem systemu i lepszą obsługę awarii poszczególnych komponentów (19).



Z

Zależności pomiędzy komponentami w systemie reaktywnym (19)

# Interfejs użytkownika w platformie Android

Jednym z podstawowych komponentów, który używany jest w procesie budowania aplikacji jest aktywność (ang. Activity). Jest to klasa, której zadaniem jest umożliwienie interakcji użytkownika z aplikacją. Za jej pomocą można utworzyć okno, w którym umieszony zostanie widok interfejsu użytkownika. Posługując się przykładem aplikacji, która pełni rolę klienta poczty elektronicznej, zakłada się, że lista z wiadomościami będzie prezentowana za pomocą jednej aktywności, a inna aktywność posłuży do utworzenia wiadomości. Istnieje możliwość grupowania kilku aktywności w grupy, ale zazwyczaj należy utożsamiać widoczne okno aplikacji z jedną instancją pod klasy Activity.

Definiując własną aktywność należy stworzyć podklasę klasy Activity dziedzicząc w ten sposób niezbędne metody, za pomocą, których programista może wpływać na interfejs użytkownika (20). Jednym z kolejnych podstawowych elementów UI w systemie Android jest klasa Fragment. Można używać jej w wieloraki sposób, lecz najczęściej jest ona ściśle połączona z Activity. W odróżnieniu od klasy Activity klasa Fragment ma raczej za zadanie odpowiadać za pewną mniejszą część interfejsu użytkownika. Warto zaznaczyć, iż zachowanie instancji Fragmentu jest ściśle powiązane z zachowaniem jej rodzica, czyli Activity. W przypadku zakończenia działania danej aktywności, wszystkie instancje klasy Fragment, które utworzone zostały w danym Activity również zakończą swoje działanie (21). Aby lepiej zrozumieć zachowanie się interfejsu użytkownika w systemie Android należy zapoznać się z cyklem życia aktywności.



### Cykl życia aktywności

Podczas używania aplikacji użytkownik może wprowadzać aktywność w różnego rodzaju stany. Sam system operacyjny także może oddziaływać na stan, w jakim znajduje się aplikacja. Taką sytuacją może być przykładowo wyświetlenie natywnego ekranu z przychodzącym połączeniem telefonicznym. W takiej sytuacji programista musi posiadać narzędzia, które umożliwią mu reagowanie na zmianę stanu interfejsu użytkownika. Za pomocą odpowiednich metod zdefiniowanych w klasie Activity (Często nazywanych również z ang. callbackami) programista jest w stanie określić jak zachowa się aplikacja podczas przechodzenia w nowy stan (22).

Aby zapanować nad zmianami, które zachodzą w obrębie aplikacji, zdefiniowanych zostało siedem zapytań zwrotnych (ang. Callbacks) za pomocą, których możemy reagować na zachodzące zmiany. Cykl życia instancji klasy Fragment nie różni się znacząco od cyklu życia aktywności, dlatego omówiony zostanie tylko cykl życia dotyczący aktywności. Poszczególne metody zostały wyszczególnione i opisane szerzej poniżej.

* OnCreate

Jest to obligatoryjne wywołanie zwrotne, który zostaje wywołane podczas utworzenia danego Activity. W tej metodzie powinien znajdować się taki kod programu, który zostanie wywołany tylko raz w ramach danego cyklu życia Activity. To w obrębie tej metody należy zdefiniować referencję do wszelkiego rodzaju przycisków, pól tekstowych i innych elementów interfejsu użytkownika. Po zakończeniu wykonywania tej metody system wykonuje następny callback OnStart (23).

* OnRestart

Metoda ta wywoływana jest po zatrzymaniu aktywności w celu przygotowania jej do restartu. Zawsze następuje przed metodą onStart (23).



Cykl życia aktywności (23)

* OnStart

Ta metoda skutkuje pojawieniem się interfejsu aplikacji na ekranie urządzenia. Jest ona zawsze wywoływana po metodach OnCreate lub OnRestart. W przeciwieństwie do metody OnCreate może ona zostać wywołana wiele razy w trakcie życia aplikacji. W tym miejscu należy zainicjalizować takie byty jak BrodcastReceiver, który pozwala na odbieranie powiadomień z całego systemu, innej aplikacji czy też procesu (23).

* OnResume

Metoda ta zostaje wywołana, gdy użytkownik powraca do aplikacji na przykład po odebraniu połączenia lub po ponownym włączeniu ekranu telefonu. W tym miejscu należy umieść kod odpowiedzialny za odświeżenie widoku lub pobranie nowych danych. Po zakończeniu działania tej metody następuje aktywny czas życia aplikacji (ang. The Active Lifetime) (24).

* OnStop

Metoda ta zostaje wywołana, kiedy aplikacja nie jest już widoczna dla użytkownika. W tym miejscu umieszczony może zostać kod programu odpowiedzialny za zatrzymanie wszelkiego rodzaju animacji, serwisów i innych bytów, które związane są z interfejsem użytkownika (24).

* OnPause

Wywołanie tej metody może nastąpić, kiedy użytkownik zacznie opuszczać aktywność lub nastąpi przysłonięcie jej przez inną aktywność. Aplikacja nadal może być widoczna na ekranie, dlatego nie zaleca się spowalnia wywoływania OnPause różnego rodzaju operacjami takimi jak zapis danych. Po tej metodzie mogą zostać wywołane metody: OnResume lub OnStop (24).

* OnDestroy

Metoda ta zostaje wywołana przed zakończeniem działania aktywności. W tym miejscu aplikacji powinna zadbać o zwolnienie zasobów, z których korzysta. Może to być na przykład zamknięcie połączenia z bazą danych.

### Layout jak podstawowa jednostka widoku

Podstawowym narzędziem, które umożliwia zdefiniowane interfejsu użytkownika jest szablon (ang. Layout). Wszystkie elementy używane podczas tworzenia UI budowane są za pomocą hierarchii klas *View* i *ViewGroup*. Klasa *View* odpowiedzialna jest za renderowanie bytów, z którymi użytkownik może wchodzić w interakcję. Natomiast klasa *ViewGroup* to rodzaj niewidzialnego kontenera, który odpowiedzialny jest za definicje struktury interfejsu użytkownika. Poniżej przedstawiono prosty schemat budowy szablonu w systemie Android.



Przykładowy schemat layoutu (23)

Deklarowanie UI może zostać zrealizowane na dwa sposoby:

* Po przez zdefiniowanie elementów UI bezpośrednio w pliku XML.
* Po przez stworzenie poszczególnych elementów UI podczas działania programu.

Zazwyczaj stosuje się pierwsze podejście w celu odseparowania widoku od logiki aplikacji. W ten sposób łatwiejsze staje się też, zdefiniowanie różnego rodzaju szablonów w zależności od szerokości ekranu lub jego orientacji.

### Format pliku szablonu

Każdy plik zawierający definicje szablonu UI musi być zdefiniowany w formacie XML. Ponadto musi zawierać dokładnie jeden element nadrzędny (rodzica), który jest typu *View* lub *ViewGroup*. Poniżej zaprezentowany został szablon odpowiedzialny za wygląd ekranu logowania w projekcie FindMyTutor. Szablon skłąda się z rodzica, którym jest *LinearLayout* i takich elementów potomnych jak: *ImageView*, *ProgressBar* czy *Button*.

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:fontFamily="@font/lato\_regular"  
 android:gravity="center\_horizontal"  
 android:orientation="vertical"  
 android:paddingLeft="@dimen/activity\_horizontal\_margin"  
 android:paddingTop="@dimen/activity\_vertical\_margin"  
 android:paddingRight="@dimen/activity\_horizontal\_margin"  
 android:paddingBottom="@dimen/activity\_vertical\_margin"  
 tools:context=".activity.LoginActivity">  
  
 <ImageView  
 android:id="@+id/imageView"  
 android:layout\_width="match\_parent"  
 android:layout\_height="120dp"  
 android:contentDescription="@string/logo\_find\_my\_tutor"  
 app:srcCompat="@drawable/logo\_design\_black2" />  
  
 <ProgressBar  
 android:id="@+id/login\_progress"  
 style="?android:attr/progressBarStyleLarge"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginBottom="8dp"  
 android:visibility="gone" />  
  
 <ScrollView  
 android:id="@+id/login\_form"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
  
 <LinearLayout  
 android:id="@+id/email\_login\_form"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:orientation="vertical">  
  
 <android.support.design.widget.TextInputLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content">  
  
 <android.support.design.widget.TextInputEditText  
 android:id="@+id/email"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_login"  
 android:inputType="textEmailAddress"  
 android:maxLines="1"  
 android:singleLine="true" />  
  
 </android.support.design.widget.TextInputLayout>  
  
 <android.support.design.widget.TextInputLayout  
 android:layout\_width="match\_parent"  
 app:passwordToggleEnabled="true"  
 android:layout\_height="wrap\_content">  
  
 <android.support.design.widget.TextInputEditText  
 android:id="@+id/password"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_password"  
 android:imeActionId="6"  
 android:imeActionLabel="@string/action\_sign\_in\_short"  
 android:imeOptions="actionUnspecified"  
 android:inputType="textPassword"  
 android:maxLines="1"  
 android:singleLine="true" />  
  
 </android.support.design.widget.TextInputLayout>  
  
 <Button  
 android:id="@+id/email\_sign\_in\_button"  
 style="?android:textAppearanceSmall"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="16dp"  
 android:text="@string/action\_log\_in"  
 android:textStyle="bold" />  
  
 </LinearLayout>  
 </ScrollView>  
</LinearLayout>

Schemat widoku ekranu logowania w aplikacji FindMyTutor

Podczas kompilacji programu plik z szablonem jest kompilowany do zasób typu *View*. Następnie w metodzie *OnCreate* możemy załadować dany szablon za pomocą metody *setContentView()*, która przyjmuje nazwę pliku zawierającego szablon interfejsu użytkownika. W celu umożliwienia stworzenia referencji do danego elementu zdefiniowanego w szablonie należy w kodzie programu przypisać do każdego elementu unikalny identyfikator. Następnie, aby stworzyć instancję danego elementu należy wyszukać go za pomocą zdefiniowanego wcześniej identyfikatora (Powinno odbywa się to w metodzie *OnCreate()*). Poniżej zaprezentowany został przykładowy kod w języku Java, który odpowiada za utworzenia instancji przycisku zdefiniowanego w szablonie.

Button mEmailSignInButton = findViewById*(*R.id.*email\_sign\_in\_button)*;

Utworzenie referencji do przycisku logowania w aplikacji FindMyTutor

Powyższy przykład tworzenia referencji do obiektów zdefiniowanych plikach XML wydaje się proste jednak posiada jedną zasadniczą wadę, utworzenie referencji musi odbywać się w metodzie *OnCreate*. Pozornie nie wydaje się to problemem jednak warto pamiętać, że w takim przypadku wszelkiego rodzaju obsługa zdarzeń takich jak naciśnięcie przycisku lub wpisanie wartości to pola tekstowego także, musi zostać zdefiniowana w tej metodzie. Aby uniezależnić tworzenie referencji do obiektów wyszczególnionych w szablonie XML w projekcie FindMyTutor została użyta biblioteka ButterKnife.

### Biblioteka ButterKnife

Celem zastosowania tej biblioteki jest zmniejszenie ilości kodu, który musi zostać wygenerowany podczas tworzenia referencji i obsługi zdarzeń elementów budujących widok aplikacji. Za jej pomocą możliwe staję się nie tylko wyekstrahowanie kodu odpowiedzialnego za obsługę interfejsu użytkownika z metody *OnCreate*, ale także grupowanie poszczególnych obiektów w listy, co skutkuje większą zwięzłością.

public class WhiteList extends AppCompatActivity *{* @BindView*(*R.id.*recycler\_view\_whitelist)* RecyclerView recyclerView;  
 @BindView*(*R.id.*white\_list\_empty\_text\_view)* TextView noNotesView;  
 @BindView*(*R.id.*switch\_whitelist\_toggle)* Switch aSwitch;  
 @BindView*(*R.id.*add\_to\_white\_list\_fab)* FloatingActionButton addToWhiteListFab;  
  
 @BindView*(*R.id.*loader)* SpinKitView loader;

…

}

*Przykład użycia biblioteki ButterKnife w projekcie FindMyTutor*

Jak widać w zaprezentowanym po wyżej kawałku kodu, logika odpowiedzialna za zdefiniowanie odwołań do elementów interfejsu użytkownika nie została zdefiniowana w metodzie *OnCreate*. Ponadto możliwe stało się także, zadeklarowanie w tym samym miejscu logiki odpowiedzialnej za obsługę zdarzeń.

## 

## 

## 

## 

### Cykl życia widoku a biblioteka RxJava

Decydując się na utworzenie obiektu obserwowanego i jego obserwatorów w aktywności lub innym komponencie odpowiedzialnym za interfejs użytkownika należy zapewnić odpowiednie powiązanie pomiędzy nimi a cyklem życia widoku aplikacji. Obserwatorzy nie powinni emitować żadnych danych, gdy widok przestaje być już dostępny dla użytkownika. Aby uzależnić stan obserwatorów od stanu, w którym znajduje się aktualnie widok należy posłużyć się obiektem *Disposable* z biblioteki RxJava.

private void loginProcess*(*String email, String password*) {* ValidateUser user = new ValidateUser*(*email, password*)*;  
  
 disposable.add*(*ldapService.validate*(*user*)* .subscribeOn*(*Schedulers.*io())* .observeOn*(*AndroidSchedulers.*mainThread())* .subscribe*(*this::handleResponse, this::handleError*))*;  
*}*

Przykład dodania obserwatora do obiektu typu Disposable

w aplikacji FindMyTutor

Następnie, aby obserwatorzy zaprzestali obserwowania należy w odpowiedniej metodzie cyklu życia aplikacji anulować wszystkie aktualne subskrypcje. Zabezpiecza to przed potencjalnymi problemami na przykład z wyciekami pamięci.

@Override  
public void onDestroy*() {* super.onDestroy*()*;  
 disposable.dispose*()*;  
*}*

Przykład anulowania wszystkich subskrypcji danego obserwatora w

metodzie onDestroy

# Przykłady zastosowanie biblioteki RxJava w projekcie FindMyTutor

Jednym z głównych zastosowań biblioteki RxJava było utworzenie serwisów HTTP komunikujących się z warstwą backendową aplikacji. Podejście reaktywne zostało także z zastosowane w niektórych elementach aplikacji odpowiedzialnych za interfejs użytkownika takich jak: wyszukiwarka użytkowników obsługa przycisków i pól tekstowych. Aby zamieszczone dalej przykłady zostały w pełni zrozumiałe należy omówić kilka dodatkowych operatów, które są niezbędne podczas budowania wydajnych rozwiązań za pomocą biblioteki RxJava.



### Obsługa wielowątkowości

W celu obsługi wielowątkowości w bibliotece RxJava należy posłużyć się odpowiednim dyspozytorem (ang. Scheduler). Przekazując wszelkiego rodzaju kosztowne obliczenia, które mogą zachodzi na danych emitowanych przez obiekt obserwowany, do odpowiedniego rodzaju despozytora możemy zapewnić, że główny wątek aplikacji nigdy nie zostanie zablokowany. Idea wydaje się prosta, zamiast wykonywać przetwarzanie danych np. w tym samym wątku, który odpowiedzialny jest za obsługę interfejsu użytkownika, należy kosztowne operacje oddelegować do innego wątku. Aby wybrać, w jakim rodzaju dyspozytora dane zostaną przetworzone należy posłużyć się metodą *SubscribeOn*. W bibliotece RxJava jednymi z najważniejszych dyspozytorów są:

* + Schedulers.computation( ) – służy do obsługi kosztownych manipulacji na danych. Domyślne liczba wątków w tym dyspozytorze jest taka sama jak liczba wątków procesora.
  + Schedulers.io( ) – przeznaczony jest do obsługi operacji wejścia/wyjścia takich jak asynchroniczny zapis czy odczyt danych.

Kolejnym bardzo przydatnym operatorem jest operator *ObserveOn*. Jest on odpowiedzialny za określenie dyspozytora, w którym obserwator będzie mógł obserwować dane wyemitowane przez obiekt obserwowany.

### Serwisy HTTP

W ramach obsługi serwisów HTTP w projekcie FindMyTutor dane zwracane z serwera aplikacji najpierw były kierowane do dyspozytora IO, a później obserwowane były w głównym wątku systemu Android *AndroidSchedulers.mainThread()*, (aby użyć wspomnianego dyspozytora niezbędne jest użycie biblioteki RxAndroid). Poniżej zaprezentowany został kod aplikacji odpowiedzialny za zdefiniowanie serwisu HTTP, który służy do obsługi modelu *WhiteList*. Model ten pozwala na dodawanie osób w aplikacji, dla który użytkownik chce być widoczny. Do podstawowej implementacji serwisu wykorzystana została biblioteka Retrofit w wersji drugiej.

package com.uam.wmi.findmytutor.service;  
  
  
import com.uam.wmi.findmytutor.model.IsUsingListBool;  
import com.uam.wmi.findmytutor.model.StudentIdModel;  
import com.uam.wmi.findmytutor.model.User;  
  
import java.util.List;  
  
import io.reactivex.Completable;  
import io.reactivex.Observable;  
import io.reactivex.Single;  
import retrofit2.http.Body;  
import retrofit2.http.DELETE;  
import retrofit2.http.GET;  
import retrofit2.http.POST;  
import retrofit2.http.PUT;  
import retrofit2.http.Path;  
  
public interface WhiteListService *{*

@GET*(*"api/users/{id}"*)* Single<User> getUserById*(*@Path*(*"id"*)* String userID*)*;  
  
 @GET*(*"api/users/whitelist/{tutorId}"*)* Single<List<String>> getTutorWhitelist*(*@Path*(*"tutorId"*)* String tutorID*)*;  
  
 @PUT*(*"api/users/whitelist/{tutorId}"*)* Completable setTutorWhitelist*(*@Path*(*"tutorId"*)* String tutorID, @Body IsUsingListBool isUsing*)*;  
  
 @POST*(*"api/users/whitelist/{tutorId}"*)* Observable<User> addStudentToWhitelist*(*@Path*(*"tutorId"*)* String tutorID,

@Body StudentIdModel student*)*;  
  
 @DELETE*(*"api/users/whitelist/{tutorId}"*)* Completable removeStudentFromWhitelist*(*@Path*(*"tutorId"*)* String tutorID,

@Body StudentIdModel student*)*;  
*}*

Przykład definicji serwisu HTTP w aplikacji FindMyTutor

W powyższym przykładzie serwisu należy zwrócić uwagę na typy obserwatorów użytych dla poszczególnych metod. Przykładowo metoda *getTutorWhitelist* zwraca obserwator typu *Single*, ponieważ wykonywanie jest zapytanie GET. Wszystkie użyte typy obserwatorów i ich zastosowania zostały szerzej umówione w rozdziale 5.

Następny krokiem w projekcie było użycie wyżej wyszczególnionego serwisu w aplikacji. Aby wyświetlić użytkowników, którzy zostali dodani do białej listy należało pobrać listę, która zawiera ich identyfikatory i następnie dla każdego z nich pobrać ich szczegółowe dane. Pobranie listy użytkowników zostało zrealizowane w następujący sposób:

private Observable<List<String>> getListOfWhitelistedUsers*(*String userId*) {* return whiteListService.getTutorWhitelist*(*userId*)* .toObservable*()* .subscribeOn*(*Schedulers.*io())* .observeOn*(*AndroidSchedulers.*mainThread())*;  
*}*

Metoda zwracająca listę identyfikatorów osób,

które znajdują się w białej liście

W zdefiniowanej metodzie warto zwrócić uwagę na zwracany typ. W definicji serwisu HTTP, metoda *getTutorWhiteList* zwraca typ Single jednak w wyżej wymienionym przykładzie użycie metody *getListOfWhiteListedUsers* skutkuje zwróceniem typu *Observable*. Odpowiedzialny jest za to operator *toObservable*, który pozwala na utworzenie obserwatora, który emitować będzie elementy listy o typie String (w tym przypadku są to identyfikatory użytkowników). Utworzenie obserwatora jest w tym przypadku niezbędne, ponieważ w dalszej części aplikacji dla każdego emitowanego elementy należy na jego podstawie wykonać kolejne zapytanie GET. W identyczny sposób zdefiniowała została metoda odpowiedzialna za pobranie szczegółowych informacji o danym użytkowniku.

private Observable<User> getUserObservable*(*String userId*) {* return whiteListService  
 .getUserById*(*userId*)* .toObservable*()* .subscribeOn*(*Schedulers.*io())* .observeOn*(*AndroidSchedulers.*mainThread())*;  
*}*

Metoda zwracająca szczegółowe dane na temat danej użytkownika

Aby ostatecznie otrzymać listę użytkowników należy dla każdego wyemitowanego elementu o typie String (identyfikator użytkownika) z metody *getListOfWhitelistedUsers* wywołać metodę *getUserObservable*, która pozwali na otrzymanie listy z obiektami o typie User. Poniżej zaprezentowano implementację tego rozwiązania.

private void fetchWhiteListedUsers*() {* disposable.add*(*getListOfWhitelistedUsers*(*tutorId*)* .doOnSubscribe*(*this::handleDoOnSubscribe*)* .subscribeOn*(*Schedulers.*io())* .observeOn*(*AndroidSchedulers.*mainThread())* .flatMap*(*Observable::*fromIterable)* .flatMap*(*this::getUserObservable*)* .subscribe*(*user -> whitelistedUsers.add*(*user*)*,  
 this::handleError,  
 this::handleComplete*)  
 )*;  
*}*

Metoda odpowiedzialna za pobranie listy osób dodanych do białej listy

O to kroki, które zostają wywołane w metodzie *fetchWhiteListedUsers*:

1. Pobranie listy elementów o typie String, która zawiera identyfikatory użytkowników, którzy zostali dodani do białej listy. W tym miejscu zostaje zwrócony obserwator.
2. Skierowanie danych do odpowiedniego dyspozytora i obserwacja ich w głównym wątku aplikacji.
3. Podczas rozpoczęcia obserwowania wykonanie metody *handleDoOnSubscribe*. Jest to metoda odpowiedzialna m.in. za wyświetlenie informacji o rozpoczęciu pobierania danych w interfejsie użytkownika.
4. Używając operatora *flatMap* przekształcenie otrzymanych elementów z obserwatora zwróconego przez metodę *getListOfWhitelistedUsers* w jedno-wymiarową listę, której elementami będą nowo utworzone obiekty typu obserwator. W tym przypadku, aby zmapować listę składającą się z elementów o typie String użyta została metoda *FromIterable*, która skutkuje utworzeniem obserwatora, który emitować będzie kolejne obserwatory o typie String. Jest to niezbędny zabieg, który pozwala na równoległe odpytywanie warstwy serwerowej aplikacji.
5. Powtórne użycie operatora *flatMap* dla każdego obserwatora wraz z metodą *getUserObservable* skutkuje otrzymaniem obserwatora emitującego już elementy o typie User.
6. Używając operatora *subscribe* stworzenie obserwatora, który otrzymane elementy o typie User zapisuje do listy połączonej z interfejsem użytkownika.
7. Posługując się metodą *onComplete* w obserwatorze odebrana zostaje informacja o zakończeniu przetwarzania danych, co skutkuje wywołaniem metody *handleComplete*, która odpowiedzialna jest za aktualizacje interfejsu użytkownika.

private void handleComplete*() {* Collections.*sort(*whitelistedUsers, this::sortByUserName*)*;  
 didFetched = true;  
 refreshUI*()*;  
*}*

Przykład metody wywołanej po zakończeniu przetwarzania danych

### Wyszukiwarka użytkowników

W celu stworzenia wyszukiwarki użytkowników utworzona została specjalna klasa *RxSearchObservable*, w której zaimplementowano logikę odpowiedzialną za obsługę interfejsu użytkownika. Działanie tej klasy opiera się o obiekt typu *Subject*, który pochodzi z biblioteki RxJava. *Subject* jest rodzajem połączenia ze sobą bytu obserwowanego i obserwatora. Ponadto, jako byt obserwowany emituje on swoje dane dla obserwatorów tylko od momentu rozpoczęcia obserwowania przez danego obserwatora. Jest to, więc obiekt typu „hot” (25). W ten sposób dany obserwator nie jest w stanie otrzymać danych, które zostały wyemitowane przed jego subskrypcją.

package com.uam.wmi.findmytutor.utils;  
  
import android.support.v7.widget.SearchView;  
  
import io.reactivex.Observable;  
import io.reactivex.subjects.PublishSubject;  
  
public class RxSearchObservable *{* public static Observable<String> fromView*(*SearchView searchView*) {* final PublishSubject<String> subject = PublishSubject.*create()*;  
  
 searchView.setOnQueryTextListener*(*new SearchView.OnQueryTextListener*() {* @Override  
 public boolean onQueryTextSubmit*(*String s*) {* subject.onNext*(*s*)*;  
 searchView.clearFocus*()*;  
 return false;  
 *}* @Override  
 public boolean onQueryTextChange*(*String text*) {* subject.onNext*(*text*)*;  
 return false;  
 *}  
 })*;  
  
 return subject;  
 *}  
}*

Klasa odpowiedzialna za implementację obserwatora emitującego wpisane wartości w pole wyszukiwarki

Aby przekształcić dane pochodzące z pola odpowiedzialnego za wyszukiwanie użytkowników posłużono się kilkoma operatorami, które odpowiedzialne były między innymi za konwersję wpisanych danych tekstowych do mały liter i odfiltrowanie pustych wartości. Jednym z najważniejszych operatorów, który został zastosowany podczas implementacji wyszukiwania użytkowników był operator *debounce*. Sprawia on, iż obserwator wyemituje dane dopiero po upływie danego interwału czasowego, co skutkuje tym, że zapytanie o wyszukanie nie nastąpi po wpisaniu każdego kolejnego znaku, lecz zostanie wywołane tylko po zdefiniowanej przerwie od wprowadzenia ostatniej wartości w polu wyszukiwarki.

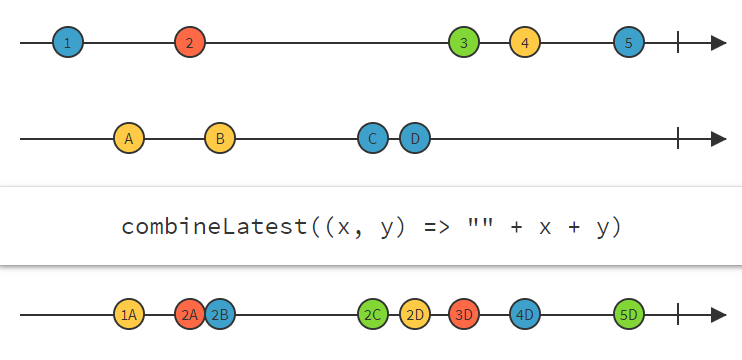
RxSearchObservable.*fromView(*searchView*)* .skip*(*0*)* .map*(*String::toLowerCase*)* .filter*(*t -> !t.isEmpty*())* .debounce*(*250, TimeUnit.*MILLISECONDS)* .subscribeOn*(*Schedulers.*io())* .observeOn*(*AndroidSchedulers.*mainThread())* .subscribe*(*this::executeSearch*)*;

Przykład kodu, który obrazuje przetwarzanie danych

emitowanych przez obserwatora wyszukiwania

### Walidacja formularza logowania

Celem walidacji formularza służącego do logowania użytkowników było zablokowanie lub odblokowanie przycisku, który umożliwia próbę zalogowania się w aplikacji. Aby ułatwić implementację obserwatora emitującego dane z pól tekstowych w projekcie FindMyTutor użyta została biblioteka RxBinding, która odpowiedzialna jest za tworzenie obserwatorów na podstawie danego elementu interfejsu użytkownika (26). W aplikacji stworzone zostały dwa obiekty, które emitują zmianę stanu pola tekstowego. Jeden z nich odpowiedzialny jest za obsługę pola *login* a drugi za pole *password*. Obsługa przycisku umożliwiającego zalogowanie została zaimplementowana za pomocą operatora *CombineLatest*. Łączy on ze sobą ostatnio wyemitowane dane dwóch obserwatorów za pomocą podanej funkcji, która zwraca nowe dane (27).



Obrazowe przedstawienie działania funkcji combineLatest

W przypadku projektu FindMyTutor przycisk logowania jest dostępny tylko, gdy obydwa pola formularza zostaną pomyślnie zwalidowane.

private void validateForm*() {* sign\_in\_button.setEnabled*(*false*)*;  
 mLoginNameView.setError*(*null*)*;  
 mPasswordView.setError*(*null*)*;  
  
 rx.Observable<TextViewTextChangeEvent> emailChangeObservable =

RxTextView.*textChangeEvents(*email*)*;

rx.Observable<TextViewTextChangeEvent> passwordChangeObservable =

RxTextView.*textChangeEvents(*password*)*;  
  
 rx.Observable.*combineLatest(*emailChangeObservable, passwordChangeObservable,  
 *(*emailObservable, passwordObservable*)* -> *{* boolean emailCheck = isEmailValid*(*emailObservable.text*())*;  
 boolean passwordCheck = passwordObservable.text*()*.length*()* > 0;  
  
 if *(*!emailCheck*) {* mLoginNameView.setError*(*getString*(*R.string.*error\_invalid\_login\_name))*;  
 *}* return emailCheck && passwordCheck;  
 *})*.subscribe*(*aBoolean -> sign\_in\_button.setEnabled*(*aBoolean*))*;  
*}*

Metoda odpowiedzialna za walidację formularza w ekranie logowania aplikacji FindMyTutor

# Podsumowanie

Użycie programowania reaktywnego wraz z biblioteką RxJava w projekcie FindMyTutor ukazuje szeroki wachlarz możliwych zastosowań paradygmatu reaktywnego. Omawiane podejście wraz z takimi bibliotekami jak RxAndroid i RxBinding pozwoliło na prostą integracje omawianego paradygmatu ze środowiskiem Android. Biblioteka RxJava okazała się być stabilnym i dojrzałym narzędziem, którego użycie poskutkowało mniejszą złożonością aplikacji. Warto zaznaczyć, że tego typu podejście do budowania aplikacji nie musi kojarzyć się tylko z budowaniem interfejsu użytkownika. Wieloplatformowe implementacje biblioteki ReactiveX ukazują wysokie zainteresowanie i zapotrzebowanie na tego typu narzędzia przez programistów.

**Spis treści**

[1. Programowanie reaktywne 2](#_Toc536216895)

[1. Wzorzec projektowy obserwator 2](#_Toc536216897)

[1.1.1 Schemat wzorca projektowego obserwator 3](#_Toc536216898)

[2.1.1 Wady i zalety 4](#_Toc536216899)

[2. Wzorzec projektowy iterator 5](#_Toc536216900)

[3.1.1 Schemat wzorca projektowego iterator 5](#_Toc536216903)

[4.1.1 Po słowie 7](#_Toc536216907)

[3. Biblioteka ReactiveX 8](#_Toc536216908)

[5.1.1 Zastosowania wzorca obserwator w bibliotece RxJava 8](#_Toc536216910)

[4. Elementy paradygmatu funkcyjnego w programowaniu reaktywnym 11](#_Toc536216911)

[6.1.1 Czyste funkcje 12](#_Toc536216913)

[7.1.1 Funkcje wyższego rzędu 12](#_Toc536216914)

[8.1.1 Po słowie 14](#_Toc536216915)

[5. The Reactive manifesto 14](#_Toc536216916)

[2. Interfejs użytkownika w platformie Android 16](#_Toc536216917)

[9.1.1 Cykl życia aktywności 17](#_Toc536216920)

[10.1.1 Layout jak podstawowa jednostka widoku 20](#_Toc536216921)

[11.1.1 Format pliku szablonu 21](#_Toc536216922)

[12.1.1 Biblioteka ButterKnife 23](#_Toc536216923)

[13.1.1 Cykl życia widoku a biblioteka RxJava 24](#_Toc536216928)

[3. Przykłady zastosowanie biblioteki RxJava w projekcie FindMyTutor 25](#_Toc536216929)

[14.1.1 Obsługa wielowątkowości 26](#_Toc536216931)

[15.1.1 Serwisy HTTP 26](#_Toc536216932)

[16.1.1 Wyszukiwarka użytkowników 30](#_Toc536216933)

[17.1.1 Walidacja formularza logowania 32](#_Toc536216934)

[4. Podsumowanie 34](#_Toc536216935)

**Bibliografia**

1. **Elisabeth Freeman Eric Freeman, Kathy Sierra, Bert Bates.** *Wzorce projektowe rusz głową .* 2017.

2. **Tomkiewicz Tomasz.** Wzorzec projektowy Obserwato. [Online] https://tomasz-tomczykiewicz.blog/2017/03/28/wzorzec-projektowy-obserwator/.

3. **Blancart Oscar.** Observer behavioral pattern. [Online] https://reactiveprogramming.io/books/design-patterns/en/catalog/observer.

4. **Oracle.** Interface Iterator documentation. [Online] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html.

5. **Blancarte Oscar.** Iterator behavioral pattern. [Online] https://reactiveprogramming.io/books/design-patterns/en/catalog/iterator.

6. **Posa Rambabu.** Java Iterator – Iterator in Java. [Online] https://www.journaldev.com/13460/java-iterator.

7. **Nowak Maciej.** Iterator - Wzorce projektowe. [Online] http://androidcode.pl/blog/wzorce/iterator/.

8. **Contributors ReactiveX.** [Online] http://reactivex.io/intro.html.

9. **Vogella.** Using RxJava 2 - Tutorial’. [Online] http://www.vogella.com/tutorials/RxJava/article.html.

10. **Contributors RxJava.** Creating observables. [Online] https://github.com/ReactiveX/RxJava/blob/2.x/docs/Creating-Observables.md.

11. **Tamada Ravi.** RxJava Understanding Observables. [Online] https://www.androidhive.info/RxJava/rxjava-understanding-observables/.

12. **Thompson S.** Haskell the craft of functional programming 3rd edition. brak miejsca : Pearson Education, 2011.

13. **Alexande Alvin.** The Benefits of Pure Functions. [Online] https://alvinalexander.com/scala/fp-book/benefits-of-pure-functions.

14. **Zawłock Artur.** Monady. [Online] https://www.mimuw.edu.pl/~zawlocki/haskell/Monady.html .

15. **Staltz Andre.** [Online] https://rxmarbles.com/#map.

16. —. [Online] http://rxmarbles.com/#filter.

17. —. [Online] http://rxmarbles.com/#reduce.

18. **Foundation Mozilla.** JavaScript Documentation. [Online] https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Array/map.

19. **Jonas Bonér Dave Farley, Roland Kuhn, and Martin Thompson.** The Reactive Manifesto. [Online] https://www.reactivemanifesto.org/ .

20. **Project Android Open Source.** Application Fundamentals. [Online] https://developer.android.com/guide/components/fundamentals.

21. —. Fragment documentation. [Online] https://developer.android.com/reference/android/app/Fragment.

22. —. Activity documentation. [Online] https://developer.android.com/reference/android/app/Activity#activity-lifecycle.

23. —. Understand the Activity Lifecycle. [Online] https://developer.android.com/guide/components/activities/activity-lifecycle.

24. **Stanek Mirosław.** Activity – podstawowe informacje, cykl życia. [Online] http://www.android4devs.pl/2011/07/activity-podstawowe-informacje-cykl-zycia/.

25. **Contributors ReactiveX.** [Online] http://reactivex.io/documentation/subject.html.

26. **Wharton Jake.** [Online] https://github.com/JakeWharton/RxBinding.

27. **Contributors ReactiveX.** [Online] http://reactivex.io/documentation/operators/combinelatest.html.