Programowanie reaktywne na przykładzie platformy Android z użyciem biblioteki RxJava

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

# Wstęp

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 proces budowania interfejsu użytkownika w platformie Android. W dalszej części przedstawia także w jaki sposób można udoskonalić działanie aplikacji posługując się programowaniem reaktywnym.

## 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. (1) Jednym z kolejnych podstawowych elementów UI w systemie Android jest klasa Fragment. Można używać ją 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. (2) 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. (3) 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. (4)

### OnRestart

Metoda ta wywoływana jest po zatrzymaniu aktywności w celu przygotowania jej do restartu. Zawsze następuje przed metodą onStart. (5)

### OnStart

Ta metoda skutkuje pojawieniem się interfejsu aplikacji na ekranie urządzenia. Jest ona zawsze wywoływana po metodach OnCreate lub OnRestart. (6)

### 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). (7)

### 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. (7)

### 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. (8)

### 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.



https://developer.android.com/guide/comp 1 - Cykl życia aktywności

## 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.



Rysunek 1‑1 Definicja szablonu

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. Łatwiejsze jest też, zdefiniowanie różnego rodzaju szablonów w zależności od szerokości ekranu lub jego orientacji. (9)

## 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 przykład pliku definiującego UI w systemie Android zawierający pole tekstowe i przycisk umieszczone w jednym z podstawowych szablonów typu LinearLayout.

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
              android:layout\_width="match\_parent"  
              android:layout\_height="match\_parent"  
              android:orientation="vertical" >  
    <TextView android:id="@+id/text"  
              android:layout\_width="wrap\_content"  
              android:layout\_height="wrap\_content"  
              android:text="Hello, I am a TextView" />  
    <Button android:id="@+id/button"  
            android:layout\_width="wrap\_content"  
            android:layout\_height="wrap\_content"  
            android:text="Hello, I am a Button" />  
</LinearLayout>

Rysunek 1‑2 - przykład szablonu interfejsu użytkownika

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 (Zazwyczaj odbywa się to w metodzie OnCreate()). Poniżej zaprezentowany został przykładowy kod w języku Java, który odpowiada za utworzenia instancji przycisku zdefiniowanego w szablonie.

Button myButton = (Button) findViewById(R.id.my\_button);

Rysunek 1‑3 - stworzenie referencji do elementu zdefiniowanego w szablonie

W dalszej części rozdziału znając już podstawy budowania UI w systemie Android przedstawione zostaną teoretyczne podstawy programowania reaktywnego.

## Istota paradygmatu reaktywnego

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 m.in wzorzec obserwatora zapewnia wygodny sposób reagowania na zdarzenia występujące w aplikacji.

## 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. (10)

## Czyste funkcje

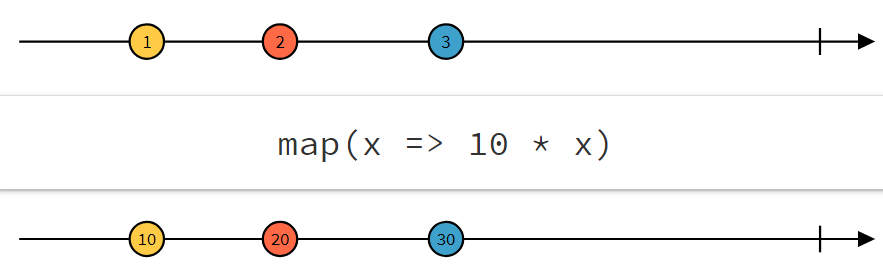
Jako czystą funkcje należy rozumieć funkcję, która nie wywołuje efektów ubocznych tj. może wpływać na otoczenie tylko po przez swój wynik i jej wynik jest tylko i wyłącznie zależy od danych wejściowych. (ang. Referential Transparency). Jako efekty uboczne można rozumieć m.in. wyświetlenie tekstu na ekran, wysłanie danych do serwera etc. Czyste funkcje są łatwiejsze w testowaniu, ponieważ ich wyjście zależy tylko od danych wejściowych. Ponadto może je składać i są łatwiejsze do zrównoleglania. (11) Oczywiście niemożliwym jest operowanie tylko czystymi funkcjami, ponieważ prawi każdy program wypisuje dane na ekran lub je wypisuje. 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. (12)

## 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

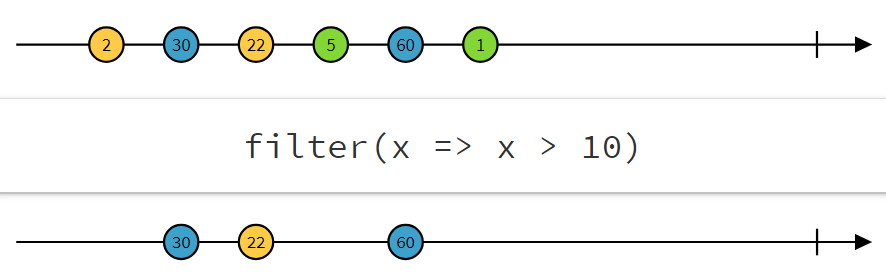
Ten typ funkcji wyższego rzędu przyjmuje jako argument funkcję i listę. Podczas przechodzenia po liście na każdym jej elemencie aplikowana jest funkcja podana w argumencie. W rezultacie wynikiem funkcji map jest nowa lista, która składa się z elementów zwróconych przez funkcje, która została zaaplikowana. (13)



http://rxmarbles.com/#map 1 - Przykład zastosowania funkcji map

* Filter

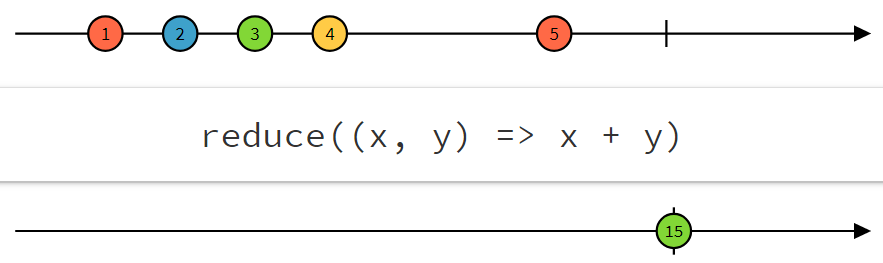
Podobnie jak w funkcji map argumentami są funkcja i lista. Jednak w tym przypadku jako argument musi zostać podana funkcja boolowska (funkcja, która zwraca tylko wartość true lub false). Funkcja zostaje zaaplikowana do każdego elementu z listy, aby następnie zwrócić elementy dla, których została zwrócona została wartość true. (13)



http://rxmarbles.com/#filter 1 - Przykład zastosowania funkcji filter

* 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.



http://rxmarbles.com/#reduce 1 - Przykład zastosowania funkcji reduce

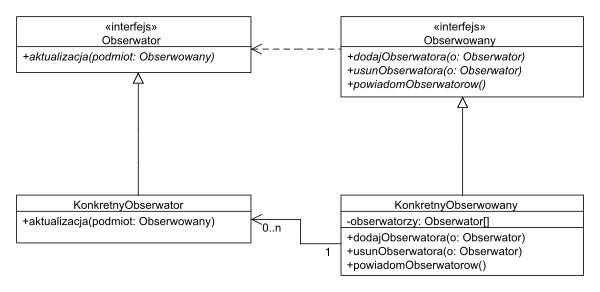
## Po słowie

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 już argument z listą, po której operujemy. Z racji tego, że funkcje wyższego zwracają nową listę możliwe jest składania ich przez co otrzymany kod jest jeszcze bardziej przejrzysty. (14)

## Wzorzec obserwator

Drugim składnikiem po paradygmacie funkcyjnym w programowaniu reaktywnym jest wzorzec obserwator (ang. Observer Pattern). Jest to wzorzec typu behawioralnego, który odpowiada za nasłuchiwanie czy też obserwacje jakiegoś zdarzenia. Wzorzec ten odpowiada, za powiadomienie wszystkich obiektów, które nasłuchują na dany rodzaj zdarzenia. Mamy, więc do czynienia z podstawową relacją jeden do wielu. Przykładem zastosowania takiego wzorca może być powiadamianie użytkowników serwisu YouTube o fakcie pojawienia się nowych filmów wśród kanałów, które subskrybują.

## Schemat wzorca obserwator



https://tomasz-tomczykiewicz.blog/2017/0 1 - Diagram klas wzorca obserwator

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.

## Wady i zalety

Jedną z największych zalet tego wzorca projektowego jest osłabienie zależności pomiędzy danymi bytami. Skutkuje to odpowiednią izolacją, ponieważ obserwowany i obserwator nie muszą mieć dużej wiedzy o sobie. Niestety nie można uzależnić aplikacji od określonej kolejności nadchodzących powiadomień. Powiadomienie może także dotrzeć do obserwatora zbyt późno. (13)

* + - 1. Wzorzec projektowy iterator

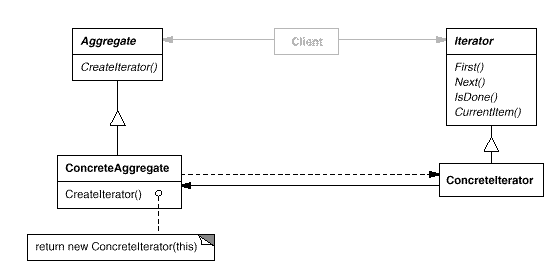
Jego celem jest stworzenie jednolitego interfejsu, który zapewnia sekwencyjny dostęp do obiektów znajdujących się w danej kolekcji bez ujawniania ich wewnętrznej struktury.

## Schemat wzorca iterator

Wzorzec ten składa się z:

* Dwóch klas abstrakcyjnych: Aggregate i Iterator
* Dwóch klas konkretnych: ConcreteAggregate i ConcreteIterator

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. (14) Iteratory posiadają metody, które pozwalają na sekwencyjny dostęp do wszystkich elementów kolekcji. W niektórych implementacjach pozwala on także na modyfikację kolekcji.



Contents

[1. Wstęp 2](#_Toc533350002)

[Interfejs użytkownika w platformie Android 2](#_Toc533350003)

[Cykl życia aktywności 3](#_Toc533350004)

[ OnCreate 3](#_Toc533350005)

[ OnRestart 3](#_Toc533350006)

[ OnStart 4](#_Toc533350007)

[ OnResume 4](#_Toc533350008)

[ OnStop 4](#_Toc533350009)

[ OnPause 4](#_Toc533350010)

[ OnDestroy 4](#_Toc533350011)

[Layout jak podstawowa jednostka widoku 6](#_Toc533350012)

[Format pliku szablonu 7](#_Toc533350013)

[Istota paradygmatu reaktywnego 8](#_Toc533350014)

[Elementy paradygmatu funkcyjnego w programowaniu reaktywnym 8](#_Toc533350015)

[Czyste funkcje 9](#_Toc533350016)

[Funkcje wyższego rzędu 9](#_Toc533350017)

[Po słowie 11](#_Toc533350018)

[1. Wzorzec obserwator 11](#_Toc533350019)

[Schemat wzorca obserwator 12](#_Toc533350020)

[Wady i zalety 12](#_Toc533350021)

[1.1.1.1. Wzorzec projektowy iterator 13](#_Toc533350022)

[Schemat wzorca iterator 13](#_Toc533350023)

[Bibliografia 15](#_Toc533350024)

[Rysunek 1 - cykl życia klasy Activity 5](#_Toc532422555)

# Bibliografia

1. https://developer.android.com/guide/components/fundamentals. [Online]

2. https://developer.android.com/reference/android/app/Fragment. [Online]

3. **https://developer.android.com/reference/android/app/Activity#activity-lifecycle.** [Online]

4. **https://developer.android.com/guide/components/activities/activity-lifecycle.** [Online]

5. **https://developer.android.com/reference/android/app/Activity.html#onRestart().** [Online]

6. **https://developer.android.com/reference/android/app/Activity#onStart().** [Online]

7. **http://www.android4devs.pl/2011/07/activity-podstawowe-informacje-cykl-zycia/.** [Online]

8. **https://google-developer-training.gitbooks.io/android-developer-fundamentals-course-concepts/content/en/Unit%201/22\_c\_the\_activity\_lifecycle\_and\_managing\_state.html.** [Online]

9. **https://developer.android.com/guide/topics/ui/declaring-layout.** [Online]

10. **Thompson, Simon.** Haskell the craft of functional programming 3rd edition. s.l. : Pearson Education, 2011.

11. **https://alvinalexander.com/scala/fp-book/benefits-of-pure-functions.** [Online]

12. **https://tomasz-tomczykiewicz.blog/2017/03/28/wzorzec-projektowy-obserwator/.** [Online]

13. **http://zasoby.open.agh.edu.pl/~09sbfraczek/obserwator%2C1%2C47.html.** [Online]

14. **http://zasoby.open.agh.edu.pl/~09sbfraczek/iterator%2C1%2C43.html.** [Online]

15. https://alvinalexander.com/scala/fp-book/benefits-of-pure-functions. [Online]