



**Politechnika Krakowska  
im. Tadeusza Kościuszki**  
Wydział Fizyki, Matematyki i  
Informatyki



**Bartosz Woliński**  
Nr albumu: 108737

**Użycie języka Swift i elementów  
paradygmatu reaktywnego na przykładzie  
aplikacji mobilnej do polecania filmów i  
seriali.**

## **TYTUŁ PO ANG**

**Praca inżynierska  
na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem  
**dra hab. Jana Kucwaja**  
Instytut Informatyki

**Uzgodniona ocena: .....**

.....  
podpisy promotora i recenzenta

**Kraków 2017**

# Spis treści

<b>1 WSTĘP</b>	<b>2</b>
<b>2 CEL I ZAKRES PRACY</b>	<b>3</b>
<b>3 ŹRÓDŁA I DEFINICJE</b>	<b>4</b>
3.1 Historia i charakterystyka systemu iOS . . . . .	4
3.2 Historia i charakterystyka języka Swift . . . . .	6
3.3 Paradygmat reaktywny . . . . .	7
3.3.1 Idea programowania reaktywnego . . . . .	8
3.3.2 Operatory reaktywne . . . . .	12
3.3.3 Współbieżność . . . . .	20
3.3.4 Zalety i wady podejścia reaktywnego . . . . .	22
3.3.5 Podejście praktyczne w implementacji na platformie iOS . . . . .	24
<b>4 PRACA WŁASNA</b>	<b>27</b>
4.1 Czynności przygotowawcze . . . . .	27
4.1.1 Instalacja narzędzi . . . . .	27
4.1.2 Wstępna konfiguracja projektu . . . . .	27
4.1.3 Stworzenie repozytorium . . . . .	27
4.2 Budowa aplikacji właściwej . . . . .	28
4.2.1 Wymagania funkcjonalne . . . . .	28
4.2.2 Wymagania niefunkcjonalne . . . . .	28
4.2.3 Diagram przypadków użycia aplikacji . . . . .	29
4.2.4 Baza danych . . . . .	29
4.2.5 Diagram wybranych klas . . . . .	31
4.2.6 Opis wybranych klas . . . . .	32
4.2.7 Opis użytych bibliotek i frameworków . . . . .	34
4.2.8 Zastosowana architektura . . . . .	35
4.2.9 Prezentacja aplikacji . . . . .	37
<b>5 PODSUMOWANIE</b>	<b>41</b>
Bibliografia . . . . .	42

# Rozdział 1

## WSTĘP

Motywacją do powstania pracy inżynierskiej o tej tematyce, była chęć poznania i zrozumienia idei paradygmatu reaktywnego w programowaniu na platformę mobilną. W mojej pracy postaram się pokazać, że podejście reaktywne jest dobrym pomysłem, szczególnie w środowiskach oferujących niewielkie zasoby. Co prawda dzisiejsze smartfony posiadają coraz wydajniejsze procesory i coraz szybsze pamięci RAM, jednakże nadal nie są to komponenty tak sprawne jak te używane w komputerach klasy PC. Z tego powodu uważam, że zrównoleglanie procesów i obliczeń ma sens na platformach mobilnych, a w tym celu warto pochylić się nad paradygmatem reaktywnym.

Po raz pierwszy idea programowania reaktywnego została wdrożona przez firmę Microsoft, która stworzyła framework Reactive Extensions na platformę .NET i oparła go głównie o popularny wzorzec obserwatora [1]. Używany termin programowanie reaktywne odnosi się do programowania reaktywnego funkcyjnego jako, że te dwa pojęcia są ze sobą nierozerwalnie związane.

Celem programowania reaktywnego jest przede wszystkim bardziej efektywne podejście do celu i sensu działania aplikacji w ogólnym ujęciu, a nie skrupulatne skupienie się na tym w jaki sposób konkretne zadania powinny być realizowane. Na takie podejście można sobie pozwolić ze względu na to, że programowanie reaktywne wymusza na systemie jego bezstanowość. Jest to możliwe m.in. dzięki temu, że programując w sposób reaktywny, działamy na strumieniach niezmiennych i niemodyfikowalnych w czasie danych a nie na pojedynczych zdarzeniach. Ponadto takie podejście pozwala oszczędzić dużo kodu i rozważać nad sensownością wydarzeń w czasie m.in dlatego, że zdarzenia synchroniczne jak i asynchroniczne traktowane są w ten sam sposób, co znaczco ułatwia tworzenie logiki programu. Podstawą paradygmatu reaktywnego jest wzorzec obserwatora czyli wzorzec obiektu obserwowanego (nadawcy zdarzeń) i obiektu obserwującego (reagującego na odebrane zdarzenia).

# Rozdział 2

## CEL I ZAKRES PRACY

Niniejsza praca dyplomowa składa się z dwóch zasadniczych części. Pierwsza z nich ma na celu podejście od strony teoretycznej do treści tematu. Rozumiem przez to objaśnienie idei paradygmatu reaktywnego i pojęć z nim związanych. Druga część pracy ma za zadanie przedstawienie praktycznego podejścia do tematu i zaprezentowanie sposobu użycia paradygmatu reaktywnego na przykładzie aplikacji mobilnej. Zadaniem tej aplikacji jest umożliwienie użytkownikowi wyszukania filmu lub serialu przy użyciu internetu i publicznego API OMDB. Znaleziony film lub serial może zostać zapisany w lokalnej bazie aplikacji. Projekt ma na celu ułatwienie decyzji użytkownikowi na temat tego co ma obejrzeć posiadającą zadaną ilość czasu. Z pośród zapisanych filmów i seriali zaproponowane zostaną te spełniające wymagania czasowe. Po obejrzeniu danej produkcji, użytkownik będzie mógł ją ocenić, a ocena ta również będzie zapisywana w lokalnej bazie.

Podsumowując powyższe stwierdzenia, przedstawiam zagadnienia opisane w obu częściach niniejszej pracy:

### Część teoretyczna:

- Krótka historia systemu iOS,
- Użyty język programowania,
- Istotne zagadnienia z tematu programowania funkcyjnego,
- Opis zagadnień związanych z paradygmatem reaktywnym, t.j.: operatory reaktywne; współbieżność; wady i zalety tego podejścia,

### Część opisująca pracę własną:

- Opis czynności przygotowawczych,
- Zdefiniowanie wymagań funkcjonalnych i niefunkcjonalnych,
- Opis modelu bazodanowego i zależności między klasami,
- Charakterystyka użytych bibliotek i frameworków,
- Implementacja aplikacji w środowisku XCode z wykorzystaniem języka Swift i opisem zastosowanej architektury,

# Rozdział 3

## ŽRÓDŁA I DEFINICJE

### 3.1 Historia i charakterystyka systemu iOS

iOS to mobilny system operacyjny stworzony przez firmę Apple Inc. na urządzenia iPod, iPhone i iPad. Został zaprezentowany w styczniu 2007 roku na konferencji Macworld. Jest jednym z dwóch<sup>1</sup> najpopularniejszych systemów mobilnych.

#### Specyfikacje techniczne

System iOS oparty jest na jądrze systemu Darwin. Jest to unixowy system operacyjny typu open-source wypuszczony przez Apple Inc. w 2000 roku. Zbudowany jest w oparciu o projekty firm Apple, NeXTSTEP, BSD, Mach i kilka innych. System działa na architekturach PowerPC, Intel x86 i ARM [?, ?]. Jądrem systemu Darwin jest XNU. Jest to hybrydowe jądro łączące w swojej implementacji Mach 3 microkernel i elementy BSD t.j. stos sieciowy czy wirtualny system plików [?].

#### Wersje systemu iOS

**iPhone OS** - pierwsza iteracja mobilnego systemu Apple. Nie nadano żadnej oficjalnej nazwy. Jedyne co utrzymywano to, że iPhone korzysta z jednej z desktopowych wersji systemu OSX [?]. W marcu 2008 roku wypuszczono zestaw narzędzi do programowania na tę platformę: iPhone SDK [?].

**iPhone OS 2.0** - w lipcu 2008 roku, wraz z premierą urządzenia iPhone 3G, miała miejsce premiera systemu iPhone OS 2.0. Najistotniejszą usługą jaką wprowadził ten system był sklep App Store. Dzięki niemu programiści mogli rozpocząć rozpowszechnianie swoich aplikacji na urządzenia iPhone i iPod Touch [?].

**iPhone OS 3.0** - Odświeżony smartfon Apple iPhone3GS został wypuszczony wraz z nowym systemem - iPhone OS 3.0. Jego najbardziej rozpoznawalną cechą było wprowadzenie czegoś co dziś w każdym systemie mobilnym musi być i jest oczywiste. Chodzi o funkcje kopiowania i wklejania, bez których nie wyobrażamy sobie systemu operacyjnego. Ponadto dostarczał funkcjonalności takich jak: Spotlight Search, klawiatura w układzie horyzontalnym i możliwość wysyłania wiadomości MMS [?].

---

<sup>1</sup>dane firmy Gartner Inc. za rok 2015: urządzenia mobilne z systemem android - 1,3 mld urządzeń z systemem iOS - 297 milionów

**iOS 4 -** Pierwszy system o tej nazwie, wprowadzony na rynek w kwietniu 2010 roku. Wówczas Apple zaprezentowało urządzenie iPhone 4 oraz zrezygnowało ze wsparcia urządzeń iPhone i iPod Touch. Wraz z premierą nowego systemu, firma oddała do dyspozycji około 1500 API dla programistów. Najistotniejszym z nich było to obłusugujące multitasking, wprowadzony po raz pierwszy w mobilnych systemach Apple. Multitasking pozwalał użytkownikom na m.in. przełączanie się między aplikacjami działającymi w tle za pomocą podwójnego kliknięcia przycisku home. Poprzednie urządzenia ze względu na swoją architekturę nie pozwalały na to [?].

**iOS 5 -** W październiku 2011 swoją premierę miał iPhone 4s - urządzenie z dwurdzeniowym procesorem - a wraz z nim system iOS 5. Nowy system oferował takie usługi jak Siri (asystent głosowy), iMessage (system wiadomości oparty o połączenie internetowe), synchronizacja z iCloud czy Notification Center. Bardzo istotny z punktu widzenia programistów, ponieważ oferuje on możliwość tworzenia i wysyłania powiadomień od aplikacji do systemu i wyświetlania ich w górnym pasku [?].

**iOS 6 -** Po raz pierwszy zaprezentowany w czerwcu 2012 roku. Wraz z nowym urządzeniem - iPhonem 5 - miał wprowadzać kilka udogodnień. Były to między innymi: kompletnie odświeżona aplikacja map i nawigacji GPS, która uwzględniała ruch uliczny. Ponadto dostarczono pełną integrację z systemem Facebook oraz usługę FaceTime - wideorozmowy za pośrednictwem telefonii komórkowej [?].

**iOS 7 -** Jeden z bardziej znaczących update'ów systemu iOS wg Craiga Federighiego.<sup>2</sup> Zaprezentowany w czerwcu 2013 roku. System posiadał kompletnie odświeżony interfejs użytkownika. Najważniejszą funkcjonalnością był Control Center czyli zestaw najczęściej używanych opcji obsługiwany przez wysunięcie za pomocą paska od dołu ekranu. Ponadto wszelkie notyfikacje systemu były wyświetlane również na zablokowanym ekranie urządzenia. Oprócz tego nowy system dostarczył wiele nowych API dla programistów, w tym ulepszony multitasking, dzięki któremu aplikacje mogły wykonywać wiele operacji w tle. Poza powyższymi, iOS 7 wprowadził jeszcze jedną istotną usługę - AirDrop czyli nowy sposób przesyłania danych między użytkownikami w sposób zaszyfrowany, korzystając z połączenia typu peer-to-peer [?].

**iOS 8 -** Największy dotychczasowy zestaw 4000 nowych API został zaprezentowany wraz z systemem iOS 8 w czerwcu 2014 roku. Zestaw ten obejmował całkowicie nowy język programowania na platformę jakim jest Swift. iOS 8 wprowadzał następujące nowości: widgety Notification Center, klawiatury dostępne z poziomu zewnętrznych aplikacji, HealthKit - system zdrowotny pomagający użytkownikowi w przechowywaniu i analizowaniu parametrów stanu zdrowia, HomeKit - aplikacja do integracji z systemami typu smart house. Wraz z prezentacją urządzenia iPhone 6 z procesorem A7, pojawił się również nowy silnik graficzny - Metal, w pełni wykorzystujący nową architekturę urządzenia do tworzenia gier i animacji [?].

**iOS 9 -** Czerwiec 2015 roku był datą pierwszej publicznej prezentacji systemu iOS 9. Na urządzeniach iPad wprowadzono funkcję używania dwóch aplikacji jednocześnie w trybie side-by-side i Picture-in-Picture. Ponadto, ulepszono Mapy uwzględniając

---

<sup>2</sup>Wiceprzewodniczący działu inżynierii oprogramowania Apple

nawigację przy użyciu publicznego transportu. Oprócz tego system wprowadził kilka optymalizacji pozwalających na dłuższy czas pracy na baterii. Jeśli chodzi o narzędzia dla programistów to stworzono nowy framework do produkcji gier - GameKit. Wprowadzono również usługę CarPlay, bardzo istotną dla producentów samochodów. Dzięki niej możliwe jest sparowanie urządzenia z systemem pojazdu i wywoływanie funkcji. Ponadto swoją premierę miała druga implementacja języka Swift czyli Swift 2.0, który stał się projektem open source [?].

**iOS 10 -** Ostatnia opisywana przeze mnie iteracja systemu iOS ujrzała światło dzienne w czerwcu 2016 roku. Według Craiga Federighiego jest to największe i najbardziej znaczące wydanie iOS'a. Istotne nowości wprowadzone przez system to m.in. otwarcie API wiadomości dla programistów, otwarcie API Siri, kompletny redesign aplikacji Mapy, nowy design Apple Music, obsługa 3D touch dla urządzeń iPhone 6 i nowszych. W tym samym roku premierę ma kolejna implementacja języka Swift - Swift 3.0 będąca bardziej kompatybilną wstecznie z językiem Objective-C [?, ?].

## 3.2 Historia i charakterystyka języka Swift

Swift to kompilowany, hybrydowy język programowania stworzony przez firmę Apple Inc. Jego premiera odbyła się podczas Worldwide Developers Conference w czerwcu 2014 roku [2].

Jeden z twórców<sup>3</sup> opisuje Swifta jako narzędzie czerpiące idee z wielu innych języków t.j. Objective-C, Rust, Haskell, Ruby, Python, C#, CLU i wielu innych.

Swift został stworzony jako nowoczesny następca Objective-C na platformy Mac OS i iOS, ale w grudniu 2015 roku stał się językiem open source. Oznacza to, że została stworzona społeczność przy użyciu serwisu Swift.org a oprócz tego udostępniono publiczne repozytorium Gita. Ponadto uwolniono narzędzia takie jak kompilator LLVM, biblioteki standardowe czy menedżer zależności projektu. Dodatkowo Swift otrzymał wsparcie na platformie Linux [?].

### Główne różnice między Swiftem a Objective-C

Swift jako język mający na celu zastąpienie leciwego już Objective-C, oferuje wiele nowych mechanizmów.

**Wartości opcjonalne -** pozwalają funkcjom, które nie zawsze zwrócią konkretną wartość lub obiekt na zwrócenie obiektu enkapsulowanego w wartość opcjonalną bądź wartość nil. W języku C i Objective-C funkcje mogą zwrócić wartość pustą (nil) nawet jeżeli spodziewana wartość jest typu struktury lub klasy. W Objective-C zwrócenie przez funkcję wartości pustej (pomimo innej spodziewanej) nie powoduje błędów komplikacji ani błędów w czasie działania. W Swiftie zaś w takiej sytuacji mielibyśmy do czynienia z błędem komplikacji lub błędem krytycznym w czasie działania co chroni nas przed niespodziewanymi zachowaniami.

---

<sup>3</sup>Chris Lattner - inżynier Apple

**Wnioskowanie typów** - kompilator języka Swift jest w stanie wywnioskować typ tworzonej zmiennej. Ponadto zmienna o zadeklarowanym (wywnioskowanym) typie nie może go zmienić.

**Krotki** - Swift wspiera obiekty krotkowe, czyli takie, które mogą przechowywać na raz kilka wartości różnych typów. Dzięki temu możliwe jest zwracanie przez funkcje wielu wartości.

**Guard** - wyrażenie warunkowe w składni Swifta. Zapewnia weryfikacje poprawności oczekiwanej typu zmiennej a w razie błędu, może spowodować wcześniejsze wyjście z funkcji.

**Elementy programowania funkcyjnego** - Swift posiada możliwość programowania funkcyjnego, co niejednokrotnie jest dużo bardziej czytelne i wydajne od tradycyjnego podejścia. Z tego powodu oferuje on operatory funkcyjne typu **map** czy **filter**.

**Enumeratory** - W Swfcie, podejście do enumeratorów zostało bardzo rozbudowane. Mogą one zawierać metody i być przekazywane przez wartość.

**Podejście do funkcji** - Każda funkcja w Swfcie posiada typ, który składa się z typów parametrów oraz typu zwracanego. To oznacza, że można przypisywać funkcje do zmiennych, a nawet przesyłać je jako parametry innych funkcji.

**Słowo kluczowe "do"** - pozwala na utworzenie nowego zakresu w kodzie a ponadto może zawierać mechanizm obsługi błędów, znany z innych języków t.j. "try catch" [?].

### 3.3 Paradygmat reaktywny

Programowanie reaktywne funkcyjne ma swoje początki już w roku 1997 [?] lecz popularne stało się za sprawą firmy Microsoft i biblioteki Reactive Extensions dla platformy .NET z 2009 roku [?].

Najczęściej podczas tworzenia programu, oczekuje się, że instrukcje będą wykonywane stopniowo, po jednej na raz, w kolejności w jakiej zostały napisane. W przypadku programowania reaktywnego, wiele instrukcji może wykonywać się współbieżnie, a ich wyniki przechwytywane są w późniejszym czasie, w losowej kolejności przez tak zwanych obserwatorów (ang. observer). Zamiast wywoływania metody, definiuje się mechanizm odszukiwania i przekształcania danych, w formie tak zwanego obiektu obserwowanego (ang. observable) a następnie zasubskrybowuje się (rozpoczyna nasłuchiwanie) obserwatora do tego obiektu. Na tym etapie uprzednio zdefiniowany mechanizm rozpoczyna działanie z obserwatorem będącym swego rodzaju strażnikiem, gotowym na przechwycenie i odpowiedź na emitowane zdarzenia.

Niewątpliwa zaletą takiego podejścia jest możliwe współbieżne wykonywanie wielu niezależnych od siebie zadań. W ten sposób czas wykonania wszystkich tych zadań będzie mniej więcej równy czasowi wykonania najdłuższego z nich.

### 3.3.1 Idea programowania reaktywnego

Podstawowym założeniem podejścia reaktywnego jest programowanie oparte o asynchronousne strumienie (sygnały) niemodyfikowalnych danych.

Źródłem sygnałów mogą być dowolne zdarzenia w czasie takie jak: modyfikacja zmiennych, interakcja użytkownika, pozycja kurSORA, struktury danych, zapytania sieciowe, operacje bazodanowe itp. Dane otrzymywane ze strumienia są przetwarzane przez obserwatora i na tej podstawie podejmowane są decyzje i skutki uboczne.

Sygnały w programowaniu reaktywnym mogą zostać scharakteryzowane ze względu na sposób działania, na dwie grupy: sygnały "zimne" i sygnały "gorące".

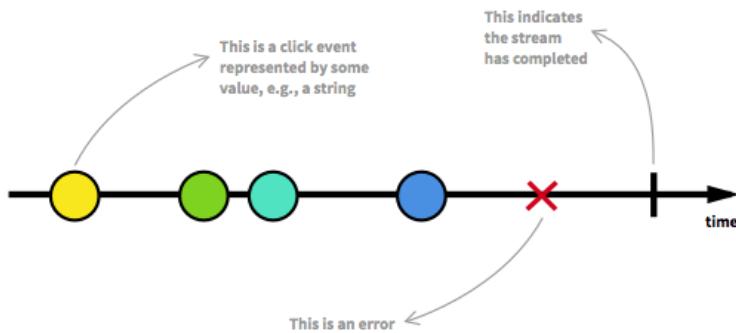
**Sygnal gorący -** zużywa zasoby i emituje zdarzenia bez względu na to czy istnieje zasubskrybowany do niego obserwator. Zdarzeniami są tutaj najczęściej dane informujące o modyfikacji zmiennych lub obiektów, koordynaty kliknięć lub kurSORA myszy, informacje o kliknięciach w kontrolki UI, bieżący czas, itp. Jak widać są to raczej sygnały ciągłe i posiadające zwykle więcej niż jedną daną w sekwencji. Nie ma zatem sensu mówienie o końcu ciągu zdarzeń w tym sygnale. Ponadto kolejno emitowane dane powielane są do wszystkich nasłuchujących.

**Sygnal zimny -** nie zużywa zasobów ani nie emituje zdarzeń dopóki obserwator nie zacznie nasłuchiwać na zmiany. Zdarzeniami najczęściej są tutaj operacje asynchronousne, połączenia HTTP, połączenia TCP, połączenia strumieniowane. Zwykle w swojej sekwencji zawiera jedno zdarzenie będące np. wynikiem odpytania sieciowego. Ponadto wyemitowany wynik najczęściej kierowany jest do jednego obserwującego.

**Subskrybcja -** za pomocą operatora subskrybcji, w programowaniu reaktywnym, łączony jest obiekt obserwowany i obserwator. Aby obiekt nasłuchujący mógł odbierać nadawane sygnały lub odebrać nadany błąd, musi najpierw zasubskrybować się do obiektu obserwowanego. Typowa implementacja operatora subskrybcji operuje na poniższych trzech metodach:

- onNext: nasłuchiwanie obiekt wywołuje tę metodę, gdy tylko wyemituje zdarzenie. Parametrem tej metody jest emitowane zdarzenie.
- onError: obserwowany obiekt wywołuje tę metodę, aby powiadomić o niepowodzeniu przy generowaniu oczekiwanych danych lub o napotkaniu innego błędu. Zatrzymuje to działanie obserwowanego i przerywa kolejne wywołania metody onNext lub onCompleted. Metoda onError, jako parametr, powinna przyjmować obiekt reprezentujący napotkany błąd.
- onCompleted: obserwowany wywołuje tę metodę po ostatnim wywołaniu metody onNext, jeżeli nie napotkano błędu.

Jak widać na obrazku 3.1, sygnał jest ciągiem poszeregowanych zdarzeń w czasie. Przechwytywanie powyższych zdarzeń odbywa się asynchronousnie a każdy z wymienionych stanów sygnału musi być obsłużony przez zdefiniowane w tym celu funkcje.

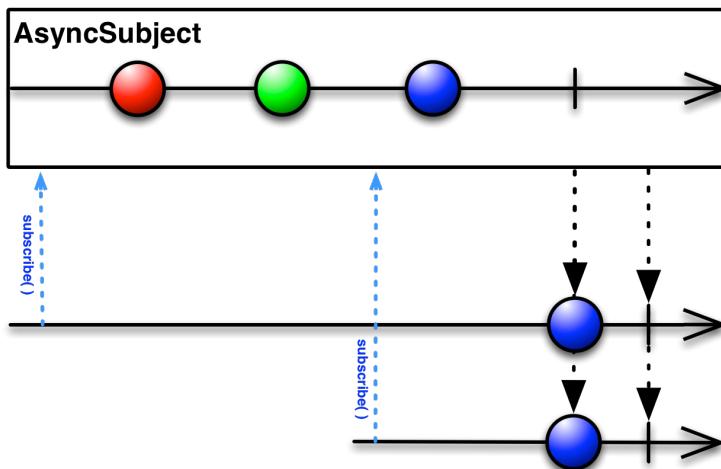


Rysunek 3.1: Schemat emitowanych danych w sygnale [?]

**Obiekty Subject -** w większości implementacji podejścia reaktywnego, stanowią one swego rodzaju most, ponieważ zachowują się jednocześnie jak obiekty obserwowane i obserwujące. Jako obserwatorzy, mogą zasubskrybować się do jednego lub więcej obserwowanych obiektów. Jako obserwowani mogą emitować i reemitować zdarzenia do obiektów obserwujących.

Rozróżnia się cztery rodzaje obiektów Subject zaprojektowanych pod konkretne przypadki.

- **AsyncSubject** - emituje ostatnią (i tylko ostatnią) wartość wyemitowaną przez źródłowy sygnał i tylko, gdy źródłowy sygnał zakończy się poprawnie. Jeśli źródłowy obiekt obserwowany zakończy się nie emitując żadnych wartości, obiekt AsyncSubject również zakończy się nie emitując żadnych wartości.

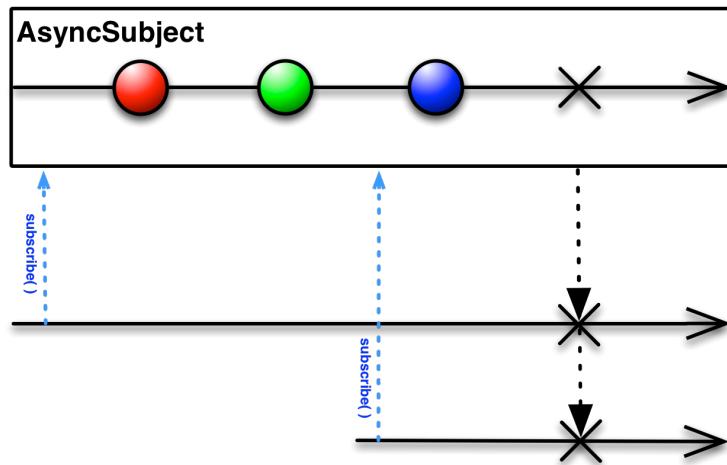


Rysunek 3.2: Schemat poprawnie zakończonego działania obiektu AsyncSubject [?].

Rysunek 3.2 przedstawia sposób działanie obiektu AsyncSubject. Jeśli zaś źródłowy sygnał zakończy się błędem, obiekt AsyncSubject nie wyemituje żadnych wartości lecz informacje o błędzie od sygnału źródłowego.

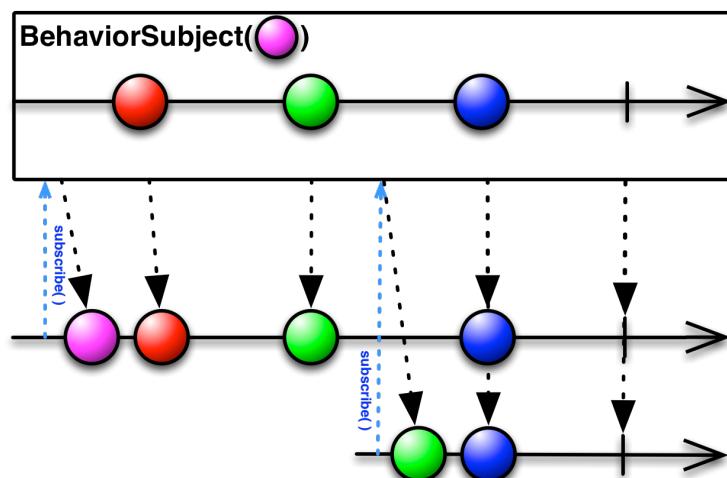
Rysunek 3.3 przedstawia sposób działanie obiektu AsyncSubject w przypadku błędu.

- **BehaviorSubject** - gdy obserwator zasubskrybuje się do obiektu BehaviorSubject, rozpoczyna on emitowanie zdarzeń rozpoczynając od ostatniego uprzednio



Rysunek 3.3: Schemat działania obiektu AsyncSubject w przypadku błędu[?]

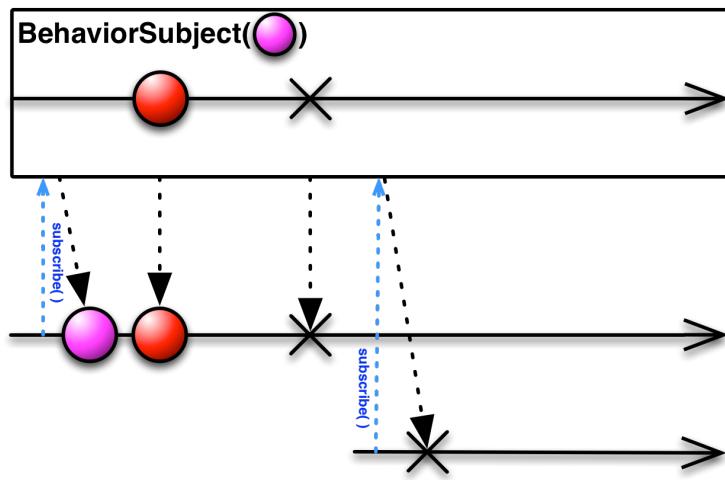
wyemitowanego przez źródłowy sygnał a następnie kontynuuje emitowanie kolejnych zdarzeń sygnału źródłowego (Rys. 3.4).



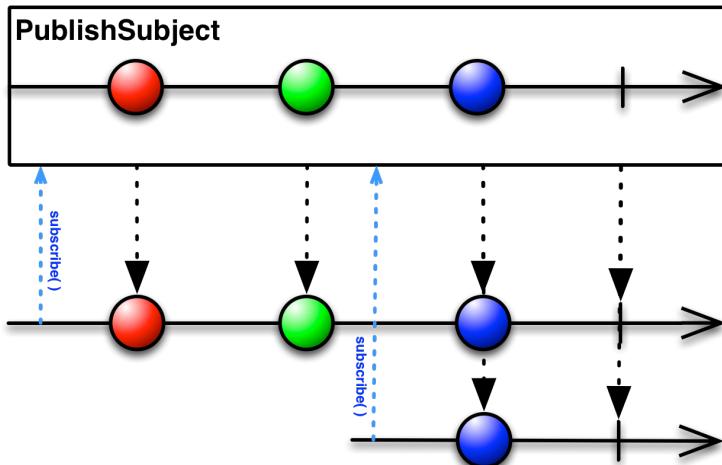
Rysunek 3.4: Schemat działania obiektu BehaviorSubject [?].

Jeśli zaś źródłowy sygnał zakończy się błędem, obiekt BehaviorSubject nie wyemituje żadnych danych do kolejnych obserwatorów, lecz prześle informacje o błędzie z obiektu źródłowego.

- **PublishSubject** - emituje do obserwatora wszystkie dane wyemitowane od momentu subskrypcji. Obiekt PublishSubject może rozpoczęć emitowanie danych bezpośrednio po utworzeniu. Istnieje zatem ryzyko, że jedno lub więcej wyemitowanych zdarzeń może zostać zgubiona od czasu powstania obiektu Subject i obserwatora subskrybującego się do niego. Aby temu zapobiec, można przekształcić sygnał w "zimny" przez manualne użycie funkcji Create i upewnienie się, że zdarzenia nie są emitowane dopóki wszyscy obserwatorzy się nie zasubskrybują (Rys. 3.6).

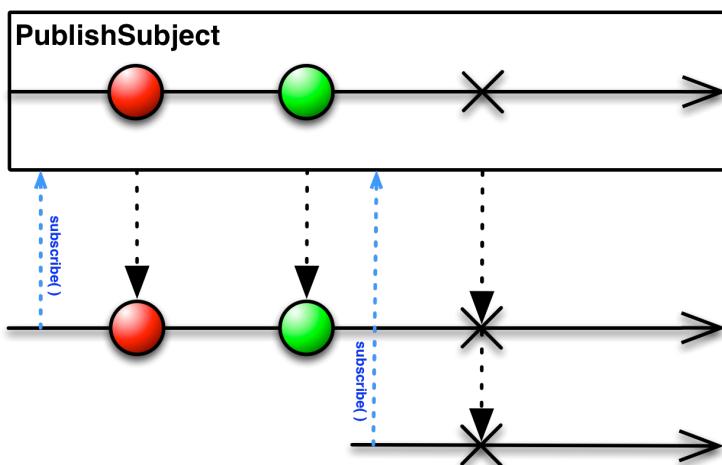


Rysunek 3.5: Schemat działania obiektu BehaviorSubject w przypadku błędu [?].



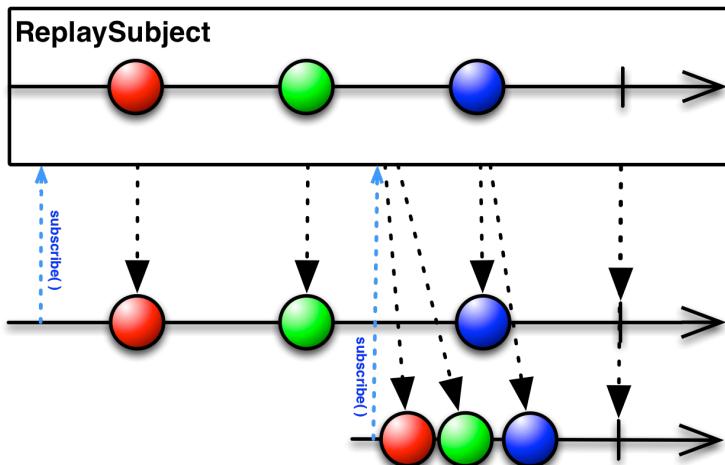
Rysunek 3.6: Schemat działania obiektu PublishSubject [?].

Jeśli zaś źródłowy sygnał zakończy się błędem, obiekt PublishSubject nie wyemituje danych, lecz prześle informacje o błędzie z obiektu źródłowego (Rys. 3.7).



Rysunek 3.7: Schemat działania obiektu PublishSubject w przypadku błędu [?].

- **ReplaySubject** - emituje do obserwatorów wszystkie zdarzenia wyemitowane przez sygnał źródłowy, bez względu na moment subskrybcji.



Rysunek 3.8: Schemat działania obiektu ReplaySubject [?].

Rys. 3.8 przedstawia sposób działania obiektu ReplaySubject. Możliwa jest implementacja obiektu ReplaySubject, której działanie będzie polegało na odrzuceniu starych zdarzeń, czyli obostrzonych rozmiarem bufora bądź upływem zadanej ilości czasu.

Obiekt ReplaySubject używany jako obserwator, dba o to by jego metoda onNext (lub inne metody on) nie były wywoływanie z różnych wątków. Może to doprowadzić do powielania już raz wyemitowanych zdarzeń i niejednoznaczności, co stoi w sprzeczności z wytycznymi projektowymi paradgymatu reaktywnego [?].

### 3.3.2 Operatory reaktywne

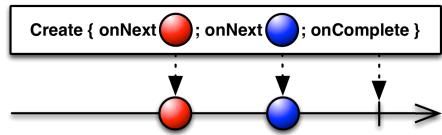
Znacząca część operatorów reaktywnych została zaczerpnięta z podejścia funkcyjnego. Mają one na celu umożliwienie tworzenia sygnałów, przekształcania w inne, filtrowania, łączenia, obsługi błędów, itp.

Większość operatorów programowania reaktywnego działa na obiekcie typu Observable, a wynikiem ich działania jest obiekt typu Observable. Dlatego stosowane jest podejście łączenia operacji w całe łańcuchy działań na sygnałach. Należy tu jednak podkreślić, że kolejne operatory łańcucha nie działają na oryginalnym obiekcie Observable niezależnie. Operują one kolejno na wynikach poprzednich działań w łańcuchu czyli na kolejnych obiektach Observable.

W dalszej części tego paragrafu zostaną scharakteryzowane najczęściej stosowane operatory reaktywne.

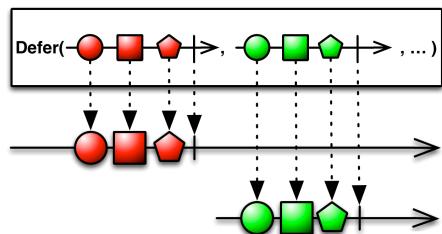
#### Operatory tworzenia sygnałów

**Create** - tworzy od podstaw obiekt Observable. Do operatora podawana jest funkcja przyjmująca w argumencie obiekt obserwującego. Tworzy się za jej pomocą obiekt obserwowany, przez zdefiniowanie działań funkcji onNext, onError i onCompleted (Rys. 3.9).



Rysunek 3.9: Schemat działania operatora Create [?].

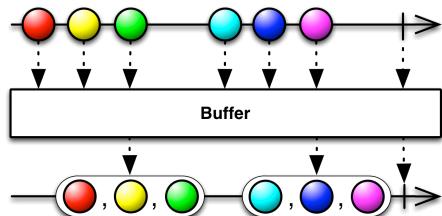
**Defer** - tworzy obiekt Observable, ale dopiero gdy obserwator się do niego za-subskrybuje. Działanie to jest powielane dla każdego subskrybenta, a zatem wszyscy nasłuchujący z tego samego sygnału, otrzymują indywidualne sekwencje zdarzeń (Rys. 3.10).



Rysunek 3.10: Schemat działania operatora defer [?].

## Operatory przekształcania sygnałów

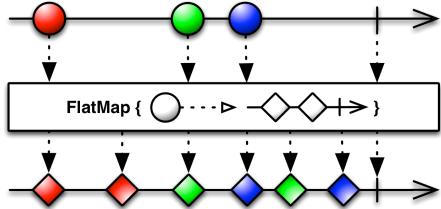
**Buffer** - cyklicznie zbiera generowane przez obiekt Observable zdarzenia i emituje w postaci pakietów zdarzeń (Rys. 3.11). W przypadku napotkania błędu przez obiekt obserwowany, operator buffer wyemituje zdarzenie informujące o błędzie bez wytwarzania pakietu danych.



Rysunek 3.11: Schemat działania operatora buffer [?].

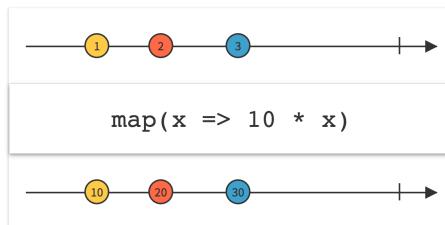
**FlatMap** - przekształca obiekt Observable przez zastosowanie funkcji zdefiniowanej dla każdego zdarzenia wyemitowanego przez źródłowy obiekt obserwowany, gdzie funkcja ta zwraca obiekt Observable emitujący własne zdarzenia. Następnie operator

flatMap scala emisje wynikowych obiektów Observable w jedną sekwencję danych (Rys. 3.12).



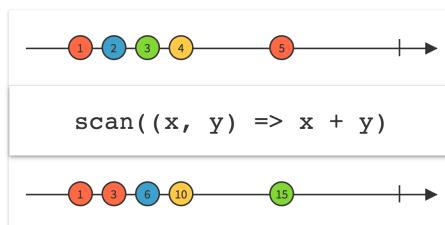
Rysunek 3.12: Schemat działania operatora flatMap [?].

**Map** - stosuje zdefiniowaną funkcję do każdego wyemitowanego przez źródłowy obiekt Observable zdarzenia, a następnie zwraca obiekt Observable emitujący przekształcone tą funkcją zdarzenia (Rys. 3.13).



Rysunek 3.13: Schemat działania operatora map [?].

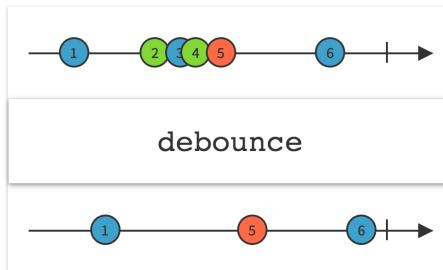
**Scan** - stosuje zdefiniowaną funkcję do pierwszego wyemitowanego przez źródło zdarzenia, a następnie emituje to przekształcone zdarzenie jako swoje pierwsze. Wynik tego zdarzenia podawany jest z powrotem do funkcji wraz z kolejnym zdarzeniem z obiektu źródłowego i emitowany. Rysunek 3.14 przedstawia działanie operatora Scan.



Rysunek 3.14: Schemat działania operatora scan [?].

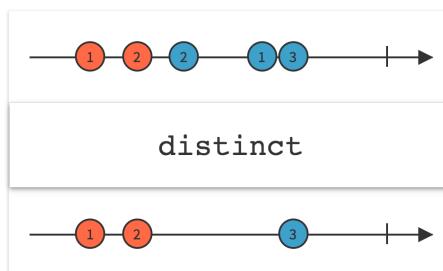
## Operatory filtrowania sygnałów

**Debounce** - emituje zdarzenie z obiektu Observable tylko po upływie zadanego interwału czasowego bez emisji zdarzenia (Rys. 3.15).



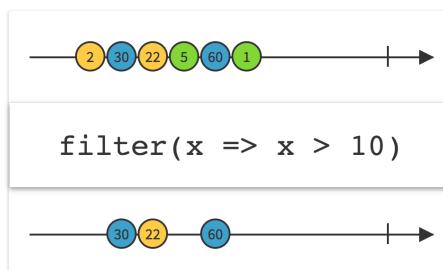
Rysunek 3.15: Schemat działania operatora debounce [?].

**Distinct** - ignoruje wartości już raz wyemitowane (Rys. 3.16).



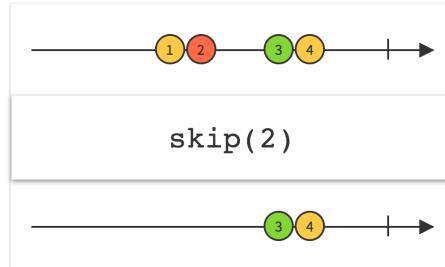
Rysunek 3.16: Schemat działania operatora distinct [?].

**Filter** - odfiltrowuje zdarzenia emitowane przez obiekt Observable, spełniające wymagania zadanej funkcji lub predykatu (Rys. 3.17).



Rysunek 3.17: Schemat działania operatora filter [?].

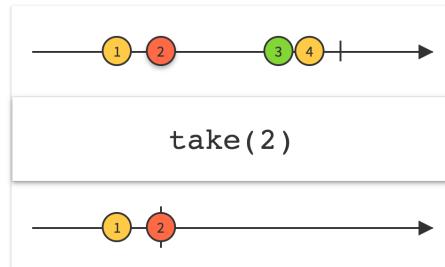
**Skip** - ignoruje n wygenerowanych przez obiekt Observable zdarzeń, gdzie n jest liczbą naturalną, będącą parametrem. Rysunek 3.18 przedstawia działanie operatora



Rysunek 3.18: Schemat działania operatora skip [?].

Skip.

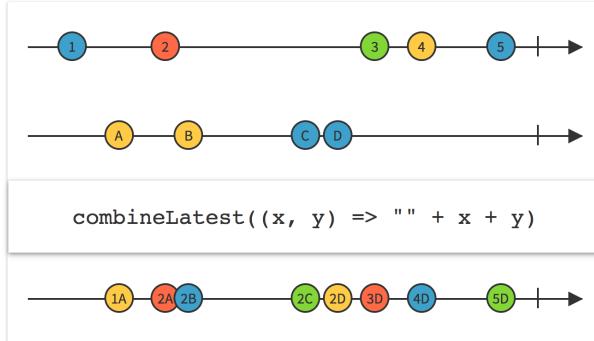
**Take** - emittuje pierwsze n wydarzeń wygenerowanych przez obiekt Observable, gdzie n jest liczbą naturalną, będącą parametrem.



Rysunek 3.19: Schemat działania operatora take [?].

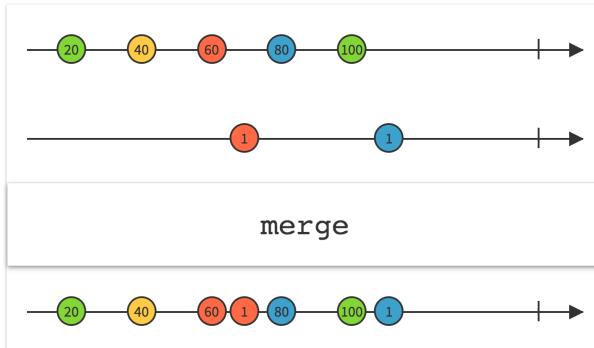
## Operatory łączenia

**CombineLatest** - łączy ostatnie wyemitowane zdarzenia zadaną funkcją (strategią) z dwóch lub więcej źródeł i emiteme nowe zdarzenie będące wynikiem powyższych (Rys. 3.20).



Rysunek 3.20: Schemat działania operatora `CombineLatest`[?]

**Merge** - łączy zdarzenia emitowane z wielu sygnałów w jeden ciągły sygnał zdarzeń

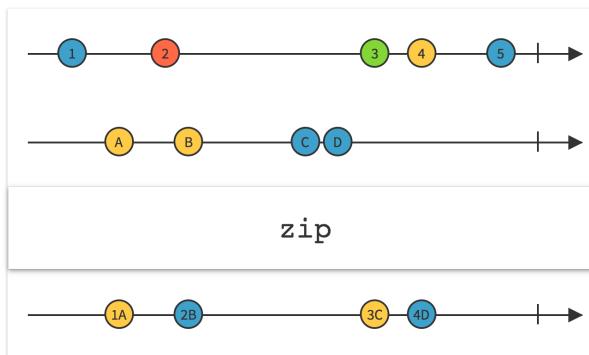


Rysunek 3.21: Schemat działania operatora `merge` [?].

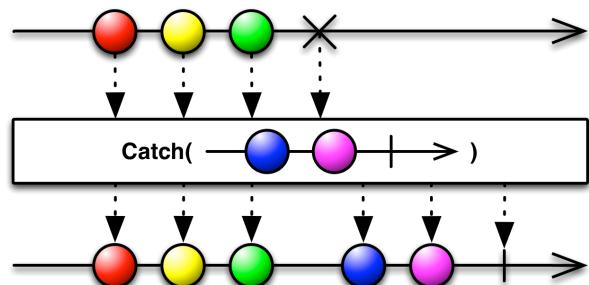
**Zip** - łączy zdarzenia emitowane z dwóch lub więcej sygnałów w pojedyncze zdarzenia reprezentowane krotką.

## Operatory obsługi błędów

**Catch** - operator ten przechwytuje zdarzenie `onError` od obiektu `Observable` i zamiast propagować błąd do obserwujących, podmienia zdarzenie na inne dane lub inny zestaw danych, co pozwala zakończyć się wynikowemu obiektowi `Observable` w normalny sposób (`onCompleted`) lub nie zakończyć się wcale (Rys. 3.23).

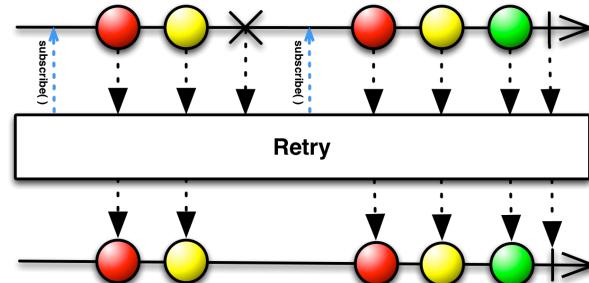


Rysunek 3.22: Schemat działania operatora zip [?].



Rysunek 3.23: Schemat działania operatora catch [?].

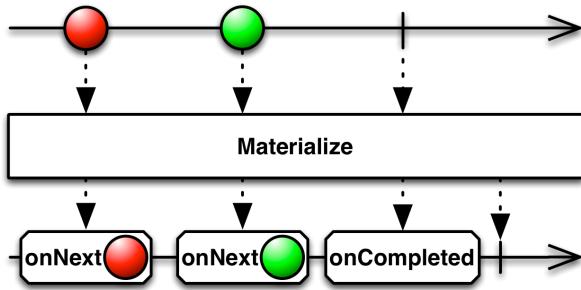
**Retry** - w przypadku odebrania od obiektu Observable zdarzenia onError, operator Retry nie propaguje błędu do obiektów obserwujących. Zamiast tego wykonywana jest resubskrypcja do źródłowego sygnału. Operator Retry zawsze przekazuje zdarzenie onNext do swoich obserwatorów, nawet z ciągów kończących się błędem (Rys. 3.24). Może to powodować zduplikowanie wyemitowania zdarzenia.



Rysunek 3.24: Schemat działania operatora retry [?].

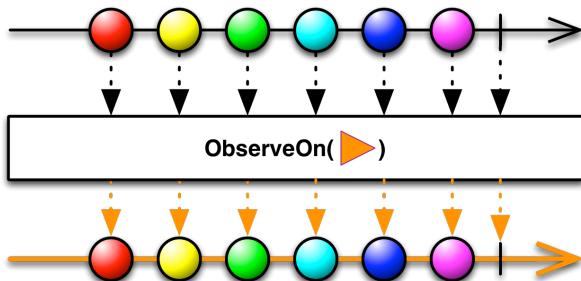
## Operatory działań na obiektach Observable

**Materialize** - konwertuje sekwencje emitowanych z obiektu Observable zdarzeń, na sekwencję zdarzeń reprezentowanych przez wywołania metod onNext i ostatecznie onComplete lub onError (Rys. 3.25).



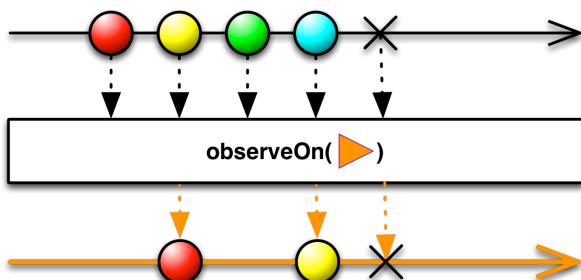
Rysunek 3.25: Schemat działania operatora materialize [?].

**ObserveOn** - w środowisku wielowątkowym, określa wątek, na którym obserwatorzy będą nasłuchiwać na generowane zdarzenia.



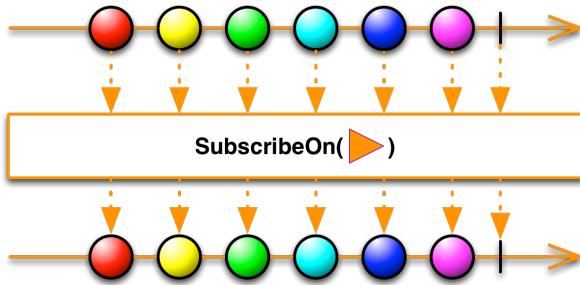
Rysunek 3.26: Schemat działania operatora observeOn [?].

Rysunek 3.26 przedstawia działanie operatora observeOn. Należy ponadto zauważyć, że observeOn prześle zdarzenie onError niezwłocznie, gdy je otrzyma. Oznacza to, że jeśli istnieje zasubskrybowany obserwator, wolniej przetwarzający otrzymywane zdarzenia, to możliwe jest, iż zdarzenie onError przeskoczy na miejsce przed zdarzeniami jeszcze nieprzetworzonymi przez tego obserwatora. Rysunek 3.27 przedstawia działanie operatora observeOnError.



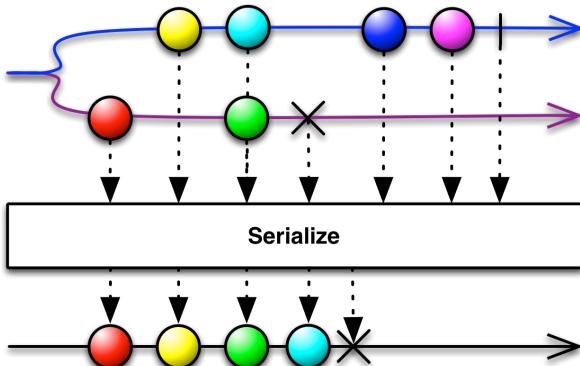
Rysunek 3.27: Schemat działania operatora observeOnError [?].

**SubscribeOn** - w środowisku wielowątkowym, określa wątek, na którym obiekt Observable będzie wykonywał pracę. Domyślnie obiekt Observable jak i ciąg zastosowanych operatorów reaktywnych, będą działały i powiadamiały obserwatorów o zdarzeniach, na tym samym wątku, na którym wywołano metodą Subscribe. Operator subscribeOn zmienia to zachowanie, przez określenie wątku, na którym obiekt Observable będzie działał (Rys. 3.28).



Rysunek 3.28: Schemat działania operatora `subscribeOn` [?].

**Serialize** - z racji tego, że obiekt `Observable` może wywoływać swoje metody asynchronicznie, na różnych wątkach, możliwe jest, że będzie próbował przesłać zdarzenie `onCompleted` lub `onError` przed jakimś innym zdarzeniem `onNext` lub nawet przesłać kilka różnych zdarzeń `onNext` z różnych wątków, współbieżnie. Takie zachowanie narusza wspomniane wcześniej zalecenia odnośnie projektowania zadań w paradigmie reaktywnym. Operator `serialize` zmusza obiekt `Observable` do emitowania zdarzeń w sposób synchroniczny. Rysunek 3.29 przedstawia działanie operatora `serialize`.



Rysunek 3.29: Schemat działania operatora `serialize` [?].

### 3.3.3 Współbieżność

Podstawowym założeniem podejścia reaktywnego jest przetwarzanie danych w sposób asynchroniczny. Aby zapewnić efektywny poziom asynchroniczności wykonywanych działań, wymagany jest pewien poziom współbieżności, czyli przetwarzania danych w oparciu o współistnienie wielu wątków lub procesów.

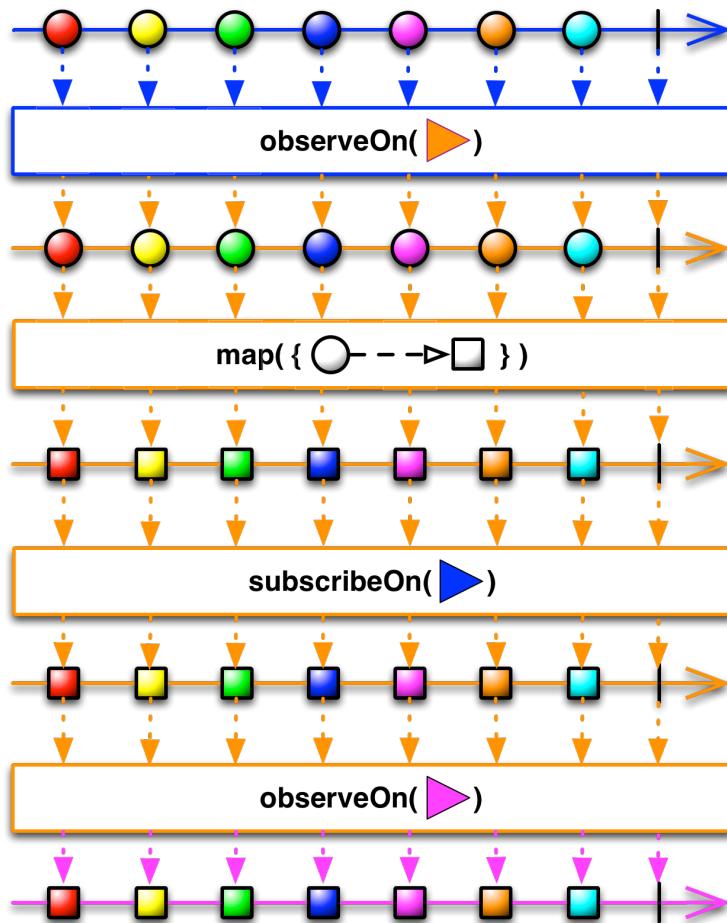
Narzędziem do zaimplementowania wielowątkowości podczas tworzenia łańcuchów sygnałów połączonych operatorami reaktywnymi, są tzw. planiści (ang. schedulers). Niektóre operatory reaktywne, jako parametr przyjmują planistę, obiekt `Scheduler`, reprezentujący wątek, na którym ma być wykonywana praca.

Domyślnie, obiekt `Observable` i zastosowany do niego łańcuch operatorów, wykonują

pracę i informują obserwatorów na tym samym wątku, na którym wywołano subskrybcję (metoda subscribe).

Jak opisano wcześniej, operator subscribeOn zmienia to zachowanie, przez określenie innego planisty (schedulera), na którym ma operować obiekt Observable. Operator observeOn zaś, określa planistę (scheduler), którego obiekt Observable ma użyć do wysyłania zdarzeń do swoich obserwatorów.

Jak pokazano na rysunku 3.30, operator subscribeOn określa wątek, na którym obiekt Observable rozpoczęte swoje działanie, bez względu na to, w którym miejscu łańcucha operatorów zostanie on wywołany. Z drugiej zaś strony, operator observeOn, określa wątek, na którym ma działać obiekt Observable od momentu wywołania tego operatora. Z tego powodu, operator observeOn można wywoływać wielokrotnie i w różnych miejscach łańcucha, aby sterować używanymi przez Observable wątkami.



Rysunek 3.30: Schemat ukazujący sposób przełączania między wątkami w środowisku reaktywnym [?].

## Rodzaje planistów w implementacji RxSwift

**CurrentThreadScheduler (Serial scheduler)** - jednostka zadania, zaplanowana jest do wykonania na bieżącym wątku. Czasami nazywany planistą trampolinowym.

Jeśli metoda CurrentThreadScheduler.instance.schedule(state) {} jest wywoływana po raz pierwszy na jakimś wątku, to planowane zadanie zostanie wykonane natychmiastowo i zostanie utworzona ukryta kolejka, w której wszystkie rekursywnie zaplanowane zadania zostaną tymczasowo umieszczone.

**MainScheduler (Serial scheduler)** - odsyła pracę do wykonania na główny wątek (MainThread). W przypadku, gdy metody zaplanowujące wywoływanie są już na głównym wątku, to zadanie zostanie wykonane natychmiastowo, bez planowania. Najczęściej ten planista używany jest do wykonywania pracy związanej z interfejsem użytkownika (UI).

**SerialDispatchQueueScheduler (Serial scheduler)** - odsyła pracę, która ma być wykonana z użyciem konkretnej szeregowej kolejki zadań typu dispatch\_queue\_t. Zapewnia, że nawet jeżeli w argumencie zostanie przesłana kolejka współbieżnych zadań, to zostanie ona przekształcona do szeregowej kolejki zadań.

Planiści kolejkowi stosują pewne optymalizacje do metody observeOn.

Planista główny (Main Scheduler) jest instancją typu SerialDispatchQueueScheduler.

**ConcurrentDispatchQueueScheduler (Concurrent scheduler)** - odsyła pracę, która ma być wykonana z użyciem konkretnej współbieżnej kolejki typu dispatch\_queue\_t. Można w argumencie przesyłać również kolejkę typu szeregowego i nie powinno stanowić tego problemu.

Ten planista jest odpowiedni, gdy jakaś praca powinna być wykonana w tle.

**OperationQueueScheduler (Concurrent scheduler)** - odsyła pracę, która ma być wykonana z użyciem konkretnej kolejki typu NSOperationQueue.

Ten planista jest odpowiedni do przypadków, gdzie musi zostać wykonana większa, bardziej wymagająca praca w tle, a użytkownik chce dostroić przebieg współbieżnych procesów używając do tego stałej maxConcurrentOperationCount.

### 3.3.4 Zalety i wady podejścia reaktywnego

Programowanie reaktywne znaczco podnosi poziom abstrakcji kodu, dzięki czemu można skupić się na wzajemnych zależnościach między zdarzeniami, tak aby zdefiniować logikę biznesową, zamiast na zawiłościach i szczegółach implementacyjnych.

Korzyść ta jest szczególnie widoczna w przypadku nowoczesnych aplikacji webowych i mobilnych, które oferują bardzo dużą interakcję z użytkownikiem. Owa interakcja ma bezpośredni wpływ na dane i decyzje podejmowane w systemie. Na początku XXI wieku, interakcja z aplikacjami webowymi sprowadzała się najczęściej do zatwierdzania długiego formularza, przesyłania go do aplikacji serwerowej (backend) i po prostym przetwarzaniu, wysłania odpowiedzi z powrotem do klienta (frontend). Od tamtego czasu aplikacje wyewoluowały do działania bardziej w czasie rzeczywistym: zmodyfikowanie pola formularza może automatycznie spowodować zapis w aplikacji serwerowej;

reakcje użytkowników serwisów społecznościowych (np. "like'i") na treści zamieszczone przez innych użytkowników są ze sobą połączone i również prezentowane w czasie rzeczywistym.

## Korzyści z użycia RxSwift (iOS)

**Bindowanie** - za pomocą tego mechanizmu w RxSwift, możliwe jest połączenie zachowania elementów UI z emitowanymi sygnałami. Mechanizm ten doskonale uzupełnia podejście w architekturze MVVM.

**Powtórzenia** - narzędzie szczególnie przydatne przy zapytaniach sieciowych do API. Niestabilne lub wolne połączenie z Internetem może być przyczyną wielu błędów. Dzięki operatorowi retry (opisane w sekcji "Operatory reaktywne"), możliwe jest powtórzenie wykonania zapytania sieciowego, bez implementowania skomplikowanej logiki i przetrzymywania stanów połączenia.

**Delegaty** - możliwe jest wykluczenie używania wbudowanych mechanizmów delegatów na rzecz bindingu reaktywnego. Daje to dużą przejrzystość kodu.

**Anulowanie** - czasami logika biznesowa wymaga, aby kosztowne operacje (zapytania sieciowe, przetwarzanie grafiki) mogły być anulowane w trakcie ich trwania. Przykładem może być tutaj pobranie tablicy zdjęć i nałożenie na nich filtra, a następnie załadowanie do obiektu TableView. Z punktu widzenia User Experience, nie ma sensu załadowywanie wszystkich zdjęć jeśli tylko część z nich jest widoczna w danym momencie, albo gdy użytkownik przewija tabelę zbyt szybko. Z pomocą przychodzi tutaj dobra obsługa wielowątkowości i mechanizm anulowania i dealokowania sygnałów w RxSwift.

**Agregowanie zapytań sieciowych** - gdy istnieje potrzeba odpytania wielu serwisów webowych i zagregowania ich odpowiedzi, podejście reaktywne zdaje się być odpowiednim narzędziem. Bez względu na to czy odpytania te powinny odbywać się współbieżnie, czy może od odpowiedzi jednego będzie zależało uruchomienie następnego. Dzięki wcześniej opisanym operatorom możliwe jest zaimplementowanie wielu zapytań sieciowych w dowolnej kombinacji i zależności.

## Wady podejścia reaktywnego

**Wysoki próg wejścia** - idea paradygmatu reaktywnego nie jest trywialna. Osoba rozpoczynająca naukę tego podejścia, zmuszona jest przestawić myślenie na takie postrzegające każdy obiekt jako sygnał. Nieistotne są stany lecz przepływ danych i efekty uboczne.

**Zarządzanie pamięcią** - większość implementacji paradygmatu reaktywnego oferuje automatyczne mechanizmy zarządzania pamięcią. Niekiedy jednak użytkownik sam musi zadbać o taki aspekt jak dealokowanie obiektu Observable. W przypadku implementacji na platformie iOS przykładem może być obiekt TabBarController, który trzyma referencje do wszystkich kontrolerów reprezentujących zakładki. W związku z

tym nie zostaną one zdealokowane nigdy w czasie działania programu. Należy zatem zadbać o to, by sygnały używane w projekcie nie zajmowały niepotrzebnie zasobów jeśli zdarzenia od nich nie są w danym momencie przetwarzane.

**Debugowanie** - działanie wielu sygnałów na różnych wątkach może utrudniać debugowanie w razie błędów. W przypadku wycieków pamięci niezbędne jest szczegółowe zbadanie podejrzanej sygnału i jego pracy na każdym z używanych wątków.

### 3.3.5 Podejście praktyczne w implementacji na platformie iOS

Na podstawie przykładów w tym paragrafie, ukazane zostało praktyczne podejście do tematu programowania reaktywnego, poprzez ich realizację w implementacji RxSwift [?].

#### Przykład 1 - stan przejściowy

W czasie tworzenia programów działających asynchronicznie, można napotkać wiele problemów z przejściowymi stanami. Typowym przykładem jest tutaj pole wyszukiwania z autouzupełnianiem.

Chcąc stworzyć obiekt pola wyszukiwania z autouzupełnianiem, pierwszym problemem jaki zostanie napotkany to sytuacja, gdy użytkownik chcący wyszukać wyniki dla frazy "abc", wpisze literę "c" w polu wyszukiwania, podczas gdy zapytanie sieciowe dla treści "ab" jest już oczekujące i musi zostać anulowane. Co prawda bez użycia podejścia reaktywnego, problem nie jest trudny do rozwiązania. Wystarczy stworzyć dodatkową zmienną utrzymującą referencję do oczekującego zapytania sieciowego.

Koleijną sytuacją, którą trzeba obsłużyć to potencjalny błąd w procesie zapytania sieciowego. Należy wówczas zastosować często zagmatwaną logikę do powtórzenia całego procesu.

Ponadto idealną sytuacją byłoby, gdyby program zaczekał pewien czas zanim zapytanie sieciowe byłoby rzeczywiście realizowane. W końcu, nie jest pożądanym, aby serwer był odpytywany za każdym razem, gdy użytkownik wpisze znak w polu wyszukiwania. Można próbować w tej sytuacji zastosować dodatkowy licznik czasu weryfikujący liczbę zapytań w czasie, tak aby nie przeciążyć serwera.

Pozostaje jeszcze kwestia, co powinno być wyświetlane w czasie wykonywania zapytania oraz co powinno być wyświetcone w przypadku błędu i to pomimo kilku prób.

Zaprogramowanie powyższego przykładu jest oczywiście możliwe bez podejścia reaktywnego, ale może wprowadzać niepotrzebne zawiłości. O to jak powyższa logika może zostać zrealizowana przy użyciu RxSwift:

Jak widać na rysunku 3.31, niepotrzebne są żadne dodatkowe flagi ani stany.

#### Przykład 2 - częściowe anulowanie

```

searchTextField.rx.text
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query in
        API.getSearchResults(query)
            .retry(3)
            .startWith([]) // clears results on new search term
            .catchErrorJustReturn([])
    }
    .subscribe(onNext: { results in
        // bind to ui
    })
    .addDisposableTo(disposeBag)

```

Rysunek 3.31: Przykładowy program rozwiązuający problem stanu przejściowego (opis w tekście) [?].

Scenariuszem tego przykładu będzie następujący ciąg działań: ma nastąpić pobranie zestawu zdjęć z zewnętrznego adresu URL, następnie zdjęcia mają zostać zdekodowane i obłożone filtrem rozmywającym. Na koniec wyświetlane są w komórkach tabeli TableView.

Założenia:

- Całość procesu powinna być anulowana dla komórki wychodzącej z obszaru widoczności, tak aby nie zużywać pasma internetowego i czasu procesora na nakładanie filtru, jako że są to operacje kosztowne.
- Całość procesu nie powinna się rozpoczynać bezpośrednio w chwili, gdy komórka wejdzie obszar widoczności. Użytkownik może przewijać widok tabeli bardzo szybko, co powodowałoby uruchamianie wielu zapytań sieciowych, które natychmiastowo musiałyby zostać anulowane.
- Dobrą praktyką byłoby ograniczenie liczby współbieżnych operacji nakładania filtru na zdjęcia, bo jest to operacja kosztowna.

Sposób realizacji powyższego scenariusza przedstawia rysunek 3.31. Oto w jaki sposób można zrealizować powyższą logikę z użyciem podejścia reaktywnego w RxSwift: Ukazuje on kod, który zrealizuje założenia oraz, gdy obiekt imageSubscription zostanie zdealokowany, zapewni że wszystkie zależne od niego asynchroniczne operacje zostaną anulowane, i że żadne błędne obrazy nie będą połączone z interfejsem użytkownika.

```

let imageSubscription = imageURLs
    .throttle(0.2, scheduler: MainScheduler.instance)
    .flatMapLatest { imageURL in
        API.fetchImage(imageURL)
    }
    .observeOn(operationScheduler)
    .map { imageData in
        return decodeAndBlurImage(imageData)
    }
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: { blurredImage in
        imageView.image = blurredImage
    })
    .addDisposableTo(reuseDisposeBag)

```

Rysunek 3.32: Przykładowy program rozwiązuający problem częściowego anulowania (opis w tekście) [?].

### Przykład 3 - agregowanie zapytań sieciowych

W tym przykładzie zostanie pokazana sytuacja, w której potrzebne jest zrealizowanie dwóch niezależnych zapytań sieciowych i zagregowanie ich wyników, gdy oba się zakończą. W tym celu, użyty zostanie operator **zip**.

W poniższym listingu można dostrzec, że obsłużona została częsta sytuacja, w której odpowiedzi od API obsługiwane są na wątku w tle a binding tych odpowiedzi do interfejsu użytkownika powinien odbyć się na wątku głównym. Ten problem, rozwiązany został przy użyciu operatora **observeOn**. Obrazek 3.33 przedstawia sposób implemen-

```

let userRequest: Observable<User> = API.getUser("me")
let friendsRequest: Observable<[Friend]> = API.getFriends("me")

Observable.zip(userRequest, friendsRequest) { user, friends in
    return (user, friends)
}
.observeOn(MainScheduler.instance)
.subscribe(onNext: { user, friends in
    // bind them to the user interface
})
.addDisposableTo(disposeBag)

```

Rysunek 3.33: Przykładowy program rozwiązujący problem agregacji zapytań sieciowych (opis w tekście) [?].

tacji przykładu.

# Rozdział 4

## PRACA WŁASNA

### 4.1 Czynności przygotowawcze

Stworzenie aplikacji mobilnej było możliwe dzięki wykonaniu uprzednio czynności przygotowawczych. Do czynności tych należało: wybór środowiska programistycznego, instalacja bibliotek i frameworków, wstępna konfiguracja projektu oraz stworzenie repozytorium.

#### 4.1.1 Instalacja narzędzi

Program XCode jest jednym z niewielu dostępnych środowisk programistycznych na platformę iOS. Jest on darmowy i dostarczany wraz systemem MacOS. XCode posiada wbudowany kompilator LLVM oraz szereg przydatnych narzędzi tj. interface builder - graficzny edytor do tworzenia elementów interfejsu, dynamiczną kontrolę składni czy menedżer systemu kontroli wersji. Ze względu na powyższe właściwości użyto właśnie tego środowiska programistycznego. Ponadto użyto narzędzia do rozwiązywania zależności - CocoaPods. Jest ono bardzo przydatne szczególnie przy instalacji bibliotek i frameworków do projektu.

#### 4.1.2 Wstępna konfiguracja projektu

Podczas konfigurowania projektu w środowisku XCode istotne jest abyśmy stworzyliśmy go z użyciem CoreData. Jest to baza danych środowiska iOS i może stanowić integralną część aplikacji.

#### 4.1.3 Stworzenie repozytorium

Dostępnych jest wiele systemów kontroli wersji, ale najpopularniejszym z nich jest GIT. Zdecydowałem się na jego wykorzystanie, ponieważ jest dosyć prosty w użyciu, a większość serwerów GITa jest darmowa. Użycie systemu typu GIT pozwoliło na kontrolę postępów pracy jak i na dokumentowanie jej. Oprócz tego użycie GITa pozwala na przywrócenie poprzedniego stanu projektu w przypadku popełnienia jakiegoś błędu.

## 4.2 Budowa aplikacji właściwej

Niejszy projekt powstawał w oparciu o niektóre elementy modelu kaskadowego cyklu życia oprogramowania [?]. Model ten stosowany jest często w praktyce do projektów o niewielkiej złożoności. Obejmuje następujące etapy:

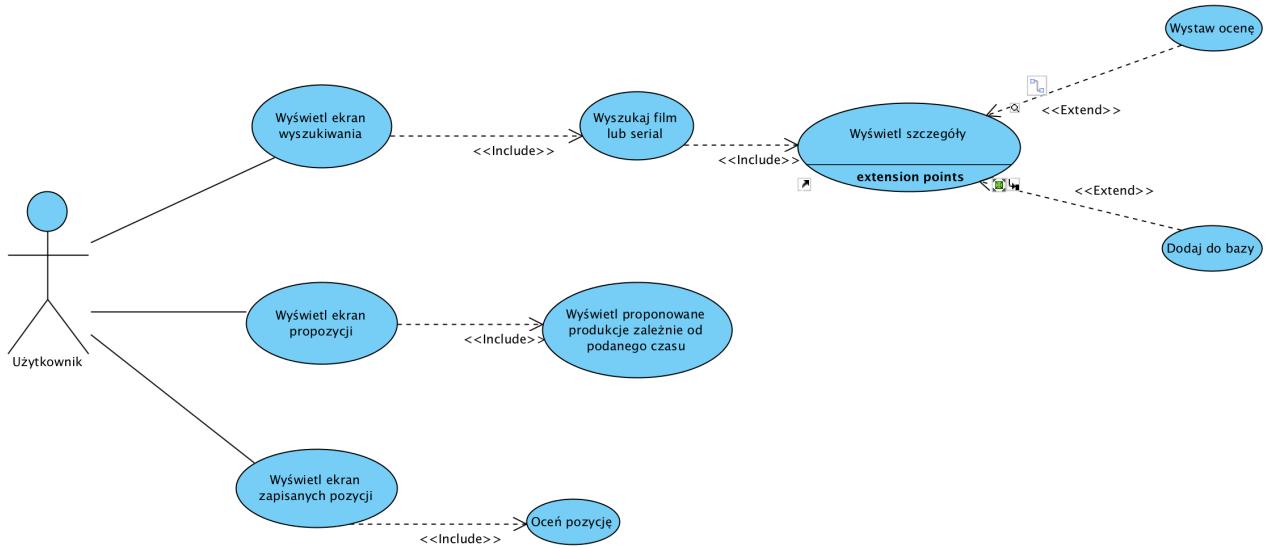
- Określenie wymagań
- Projektowanie systemu
- Implementacja i testowanie modułów
- Testowanie połączeń modułów i całości systemu
- Użytkowanie i pielęgnacja

### 4.2.1 Wymagania funkcjonalne

- Nawigacja pomiędzy zakładkami
- Wyszukiwanie filmu lub serialu i wyświetlenie wyniku w tabeli
- Podgląd szczegółów dot. filmu lub serialu tj. oceny użytkowników serwisu filmowego IMDB czy plakatu
- Możliwość dodania filmu bądź serialu do swojej bazy
- Możliwość oznaczenia filmu bądź serialu jako obejrzany i wystawienie mu oceny w swojej bazie
- Podgląd wszystkich dotychczas zapisanych filmów i seriali wraz z ocenami jeśli zostały nadane
- Możliwość wyświetlenia propozycji do obejrzenia z bazy zapisanych filmów i seriali w zależności od wprowadzonego przez użytkownika czasu

### 4.2.2 Wymagania niefunkcjonalne

- Spójność danych - dane podczas przesyłania nie mogą ulec zniekształceniu
- Szybkość transmisji - wszystkie operacje muszą odbywać się w czasie umożliwiającym płynne działanie aplikacji
- Stabilność połączenia - aplikacja, do prawidłowego działania, wymaga stabilnego połączenia internetowego
- Ograniczenie liczby zapytań API do minimum
- Brak zasobozerności aplikacji pomimo używania przez niej wielu wątków



Rysunek 4.1: Schemat przypadków użycia [opracowanie:własne ]

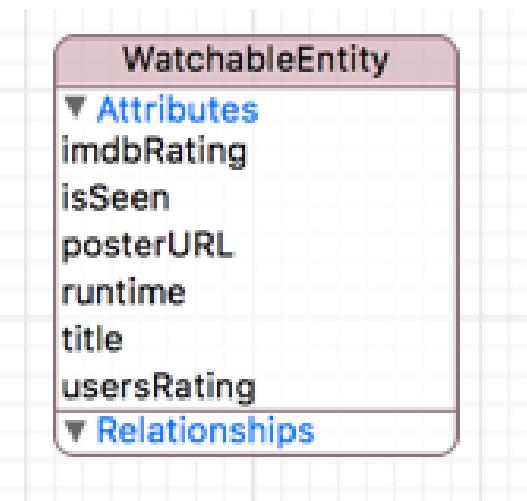
#### 4.2.3 Diagram przypadków użycia aplikacji

Schemat 4.1 przedstawia ogólne przypadki użycia aplikacji mobilnej. Użytkownik może wykonywać następujące czynności:

- Wyświetlić ekran wyszukiwania
- Wyszukać film lub serial
- Wyświetlić szczegóły dot. znalezionej produkcji
- Wystawić ocenę znalezionemu filmowi lub serialowi
- Zapisać w bazie znaleziony film lub serial
- Wyświetlić ekran propozycji
- Wyświetlić proponowane pozycje do obejrzenia przy uwzględnieniu czasu wprowadzonego przez użytkownika
- Wyświetlić ekran zapisanych pozycji
- Oceń zapisane pozycje

#### 4.2.4 Baza danych

W bazie danych przechowywane są informacje o zapisanych przez użytkownika pozycjach reprezentujących film lub serial, które użytkownik chciałby obejrzeć. Baza ta nie jest skomplikowana i nie wymaga więcej niż modelu jednej encji. Encja ta wystarcza do przechowywania żądanego informacji i w związku z tym nie wymaga relacji z innymi encjami.

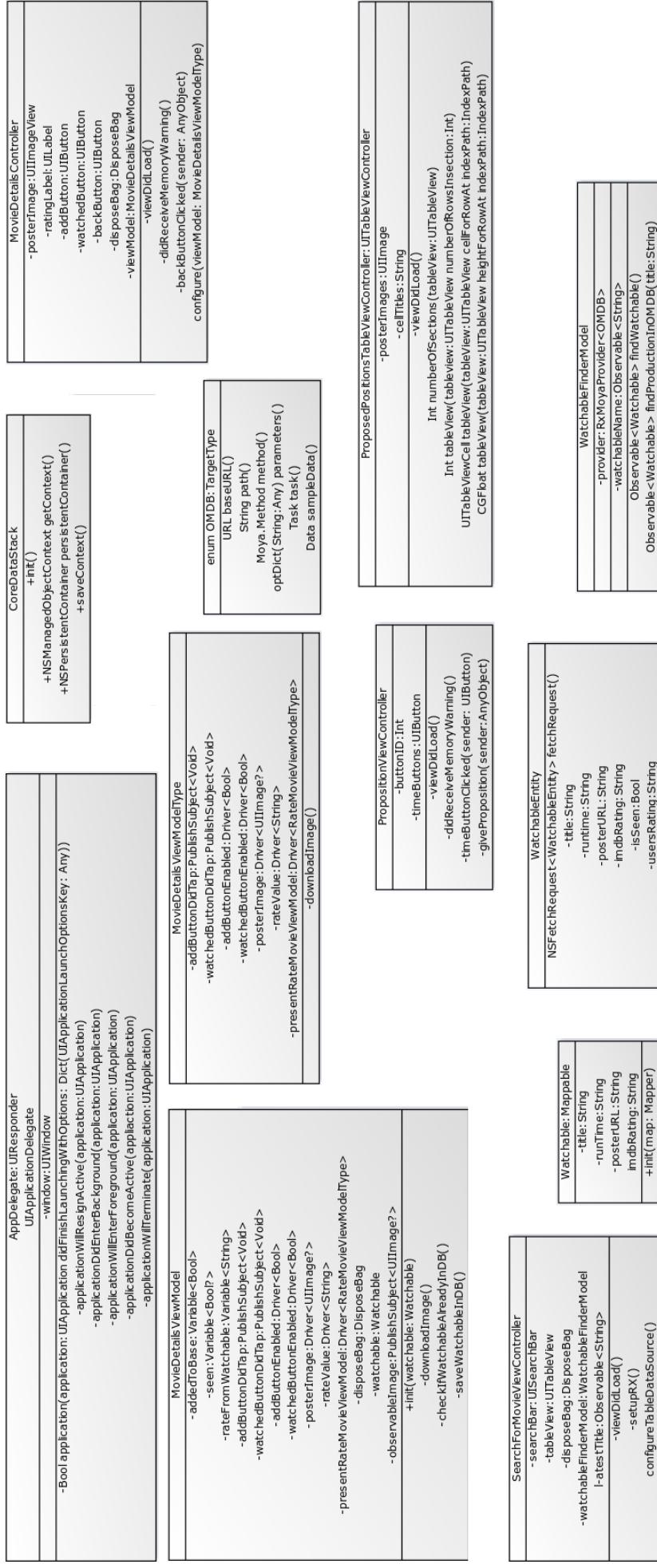


Rysunek 4.2: Diagram bazy danych *[opracowanie: własne]*

Na rysunku 4.2, można zobaczyć encję posiadającą następujące atrybuty:

- **imdbRating** - reprezentuje ocenę użytkowników serwisu IMDB
- **isSeen** - binarna wartość typu Bool, określająca czy użytkownik obejrzał już daną pozycję
- **posterURL** - zewnętrzny adres URL informujący o tym skąd można pobrać grafikę dla danego filmu lub serialu
- **runtime** - reprezentuje czas trwania filmu lub serialu, brany pod uwagę przy mechanizmie proponowania
- **title** - reprezentuje tytuł pozycji
- **userRating** - ocena pozycji, wystawiona przez użytkownika aplikacji

## 4.2.5 Diagram wybranych klas



Rysunek 4.3: Diagram klas /opracowanie: wlasne/

#### 4.2.6 Opis wybranych klas

- Klasa **AppDelegate** - główna klasa programu, tworzona automatycznie. Implementuje metody interfejsu UIApplicationDelegate, który jest odpowiedzialny za obsługę metod wywoływanych przez singleton UIApplication w odpowiedzi na ważne zdarzenia w cyklu życia aplikacji. Te ważne zdarzenia to m. in. zmiany stanu aplikacji takie jak przejście aplikacji w background, przejście w foreground, obsługa notyfikacji, interakcja z innymi aplikacjami w systemie czy interakcja z samym systemem iOS.
- Klasa **CoreDataStack** - reprezentuje menedżera bazy danych CoreData. Jej metody pozwalają na pobranie bieżącego kontekstu, czyli stanu danych i relacji w bazie danych oraz zapisanie tego kontekstu.
- Klasa **WatchableEntity** - reprezentuje model encji bazodanowej. Jej metoda pozwala na wykonanie zapytania bazodanowego, a jej pola reprezentują poszczególne atrybuty encji.
- Klasa **SearchForMovieViewController** - kontroler widoku wyszukiwania produkcji. Jej atrybutami są elementy interfejsu użytkownika t.j. tabela czy pasek wyszukiwania, który jest tutaj obsługiwany reaktywnie. Ponadto korzysta z modelu wyszukiwania produkcji w celu wyświetlenia znalezionych wyników. Nie posiada informacji o tym w jaki sposób przebiega proces wyszukiwania.
- Klasa **MovieDetailsController** - kontroler widoku szczegółów na temat znalezionej produkcji. Odpowiedzialny jest za operacje interfejsu użytkownika takie jak pobranie obrazka czy wyświetlenie oceny znalezionej filmu lub serialu. Komunikuje się tylko z modelem MovieDetailsViewModel za pośrednictwem interfejsu MovieDetailsViewModelType, aby otrzymywać informacje na temat danych do wyświetlenia.
- Klasa **MovieDetailsViewModel** - stanowi model widoku dla klasy kontrolera poprzednio opisanego. Implementuje metody i pola interfejsu MovieDetailsViewModelType. Odpowiedzialna jest za reaktywne zmiany modelu, widoku takie jak: blokowanie/odblokowywanie możliwości kliknięcia przycisków, podawanie do kontrolera danych do wyświetlenia ze znalezionej produkcji, przygotowanie modelu widoku oceny filmu czy zapisanie pozycji w bazie danych.
- Enumerator **OMDB** - implementuje metody i pola wymagane przez interfejs Mappable, potrzebny do działania modułu Moya odpowiedzialnego za zapytania sieciowe w sposób reaktywny. Enumerator ten odpowiedzialny jest za dostarczenie adresów URL wszystkich używanych endpointów API, metod odpytania, sposobu kodowania odpowiedzi oraz przykładowy model zwracany przez API danych.
- Klasa **WatchableFinderModel** - model wyszukiwania filmów i seriali. Izoluje logikę i sposób odpytywania API od reszty klas. Korzysta z dostarczanego przez framework Moya providera zależnego od uprzednio opisanego enumeradora dostarczającego wszystkich potrzebnych danych. Jego metody generują sygnały będące wynikiem odpytywania API o zadany tytuł.

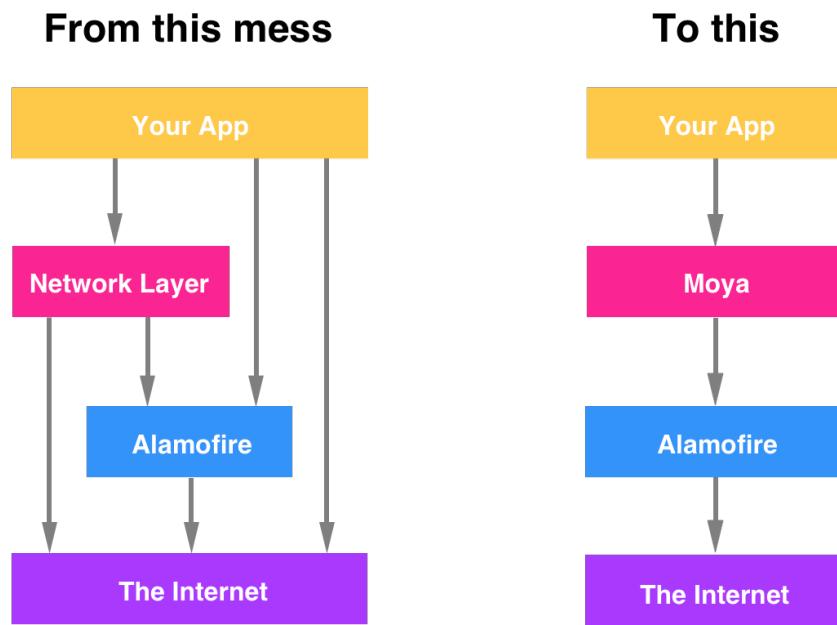
- Klasa **Watchable** - model danych reprezentujących znalezione przez API film lub serial. Implementuje interfejs Mappable, który odpowiedzialny jest za przeprowadzenie próby zmapowania otrzymanego w odpowiedzi (od API) obiektu typu JSON na obiekt typu Mappable.

#### 4.2.7 Opis użytych bibliotek i frameworków

Stworzenie projektu aplikacji mobilnej wymagało zastosowania niezbędnych bibliotek. Zgodnie z tematem pracy, aplikacja powinna opierać się o paradygmat reaktywny, który został zaimplementowany dzięki stworzonym do tego, zewnętrznym modułom i bibliotekom.

Poniżej, zostały przedstawione użyte moduły i ich krótka charakterystyka:

- **CocoaPods** - menedżer zależności w projektach XCode. Dzięki niemu, możliwe jest określenie zależności w projekcie dzięki prostemu plikowi textowemu **Podfile**. CocoaPods rekursively rozwiązuje zależności pomiędzy użytymi bibliotekami, wyszukuje kod źródłowy dla wszystkich zależności oraz tworzy i utrzymuje obszar roboczy w celu budowania projektu XCode (plik `.xcworkspace`) [?].
- **UIKit** - dostarcza niezbędnej infrastruktury potrzebnej do skonstruowania i zarządzania aplikacją na platformie iOS i tvOS. Jest to framework zapewniający architekturę okna i widoku do zarządzania interfejsem użytkownika w aplikacji. Ponadto służy do obsługi zdarzeń potrzebnych do odpowiedzi na reakcje użytkownika oraz dostarcza modelu aplikacji odpowiedzialnego za działanie pętli głównej programu i interakcji z systemem [?].
- **CoreData** - graf obiektów [?] i framework warstwy modelu zapewniający trwałość tych obiektów. Pozwala, aby dane zorganizowane w model relacyjny encja-trybut, były serializowane do postaci XML, binarnej lub zasobów SQLite. Dany mi można zarządzać używając wysokopoziomowych obiektów reprezentujących encje i ich relacje. CoreData zarządza wersjami zserializowanymi, zapewniając cykl życia obiektu i możliwość zarządzania grafem obiektów włączając w to trwałość tych obiektów.  
CoreData jest bezpośrednim interfejsem SQLite izolującym programistę od działającego "pod spodem" języka SQL.
- **RxSwift** - implementacja biblioteki .NET Reactive Extensions w języku Swift na platformę iOS/macOS. Jest to próba przeportowania tak wielu jak to tylko możliwe konceptów z oryginalnej biblioteki Rx. Głównym zamierzeniem RxSwift jest umożliwienie łatwej kompozycji asynchronicznych operacji i strumieni zdarzeń/dane.
- **RxCocoa** - nakładka na frameworki Cocoa i Cocoa Touch, rozszerzająca możliwości dostarczane przez nie (obiekty UI i ich obsługa) o reaktywne zachowanie w oparciu o framework RxSwift.
- **RxOptional** - rozszerzenie RxSwift dla obiektów opcjonalnych w Swift, pozwalające na ich reprezentację w podejściu reaktywnym.
- **Moya** - biblioteka stanowiąca nakładkę reaktywną na framework Alamofire [?] umożliwiający izolację logiki odpowiedzialnej za wszelkie operacje sieciowe. Dzięki jej użyciu możliwe jest przekształcenie architektury obsługi operacji sieciowych zgodnie ze schematem zaprezentowanym na rysunku 4.4.
- **Moya\_ModelMapper** - biblioteka stanowiąca nakładkę dla frameworku Mapper [?] ułatwiającego konwersję JSON do silnie typowanych obiektów Swift. Moya\_ModelMapper dostarcza potrzebnych bindingów do biblioteki Moya do łatwiejszej serializacji obiektów JSON przy użyciu RxSwift.



Rysunek 4.4: Przekształcenie logiki zapytań sieciowych dzięki Moya[?]

#### 4.2.8 Zastosowana architektura

Wybranie właściwej architektury w projekcie było niezbędne, aby w pełni ukazać co oferuje programowanie reaktywne w języku Swift.

**MVC** - czyli Model View Controller jest najczęściej stosowanym podejściem jeśli chodzi o aplikacje na iOS. Nie zawsze jednak jest to podejście właściwe.

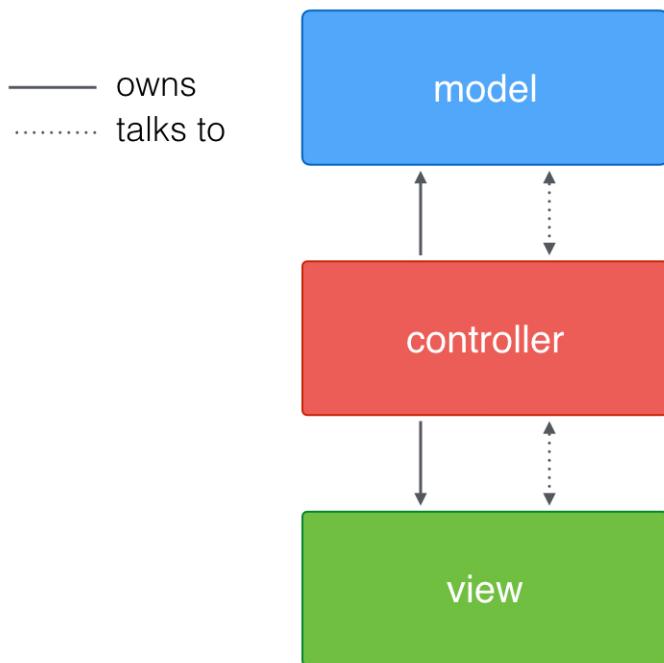
Typowo, architektura ta powinna się sprowadzać do schematu przedstawionego na rysunku 4.5.

Niestety, role poszczególnych komponentów w tej architekturze (Rys. 4.5) nie zawsze są jasno określone i tak np. typowe zadanie formatowania daty nie pasuje jako rolą widoku ani tym bardziej modelu, dlatego staje się wówczas odpowiedzialnością kontrolera. Wiele tego typu zadań zostaje rozwiązywane właśnie w obszarze kontrolera co powoduje poważne konsekwencje takie jak rozrost kontrolerów, trudności w testowaniu czy niemożność ich ponownego użycia w innych częściach aplikacji.

Z powyższych powodów, architekturą wybraną do zaprojektowania ninejszej aplikacji został **MVVM**, czyli **Model-View-ViewModel**, który w środowisku iOS można sprowadzić do schematu przedstawionego na rysunku 4.6.

W architekturze ViewModel stoi pomiędzy modelem a kontrolerem i dostarcza danych kontrolerowi, które powinien wyświetlić w widoku. Dzięki temu ViewController nie ma już bezpośredniego dostępu do modelu. Poprzednio rozpatrywane, typowe zadanie formatowania daty zostanie zlecone właśnie do ViewModelu, który przygotuje odpowiednio dane i już sformatowane, poda dalej do kontrolera.

Nie należy jednak oczekwać od ViewModelu, aby wiedział w jaki sposób dane są wyświetlane przez ViewController.



Rysunek 4.5: architektura MVC [?].



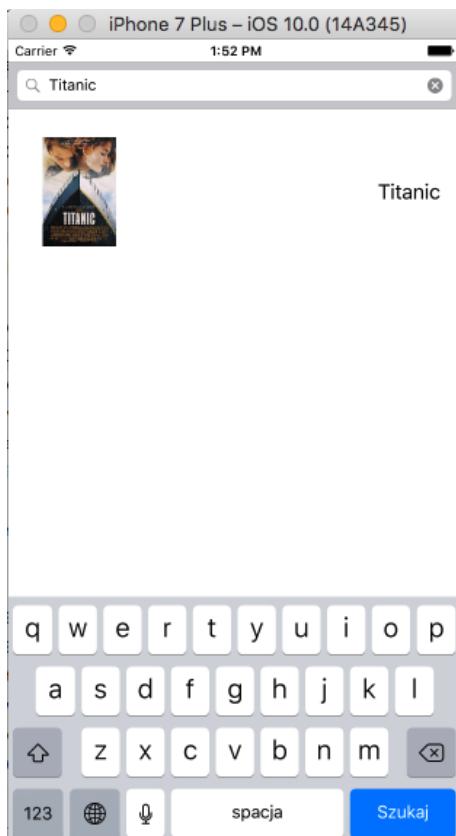
Rysunek 4.6: architektura MVVM [?].

Z racji, że w przedstawionej architekturze, **ViewModel** "nie wie" nic o kontrole-rze, komunikacja między tymi komponentami odbywa się na zasadzie nasłuchiwanie. Kontroler przy pomocy reaktywnych bindingów obserwuje zmiany w **ViewModelu** i na tej podstawie reaguje, wyświetlając odpowiednie elementy interfejsu użytkownika, wykorzystując w ten sposób potencjał programowania reaktywnego.

#### 4.2.9 Prezentacja aplikacji

Główne funkcje aplikacji prezentowane są w widokach wybieranych spośród trzech zakładek paska dolnego.

##### Widok wyszukiwania filmu/serialu

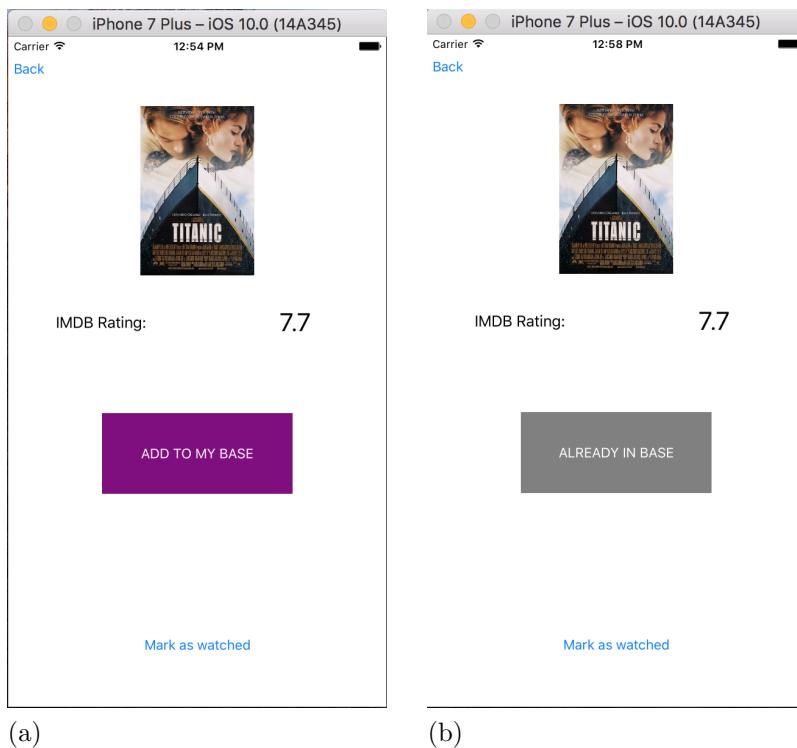


Rysunek 4.7: Widok wyszukiwania [opracowanie: własne]

W widoku przedstawionym na rysunku 4.7, możliwe jest kliknięcie paska wyszukiwania i wpisywanie tytułu. Wprowadzany z klawiatury ciąg znaków przetwarzany jest w sposób reaktywny z użyciem operatora **debounce**. Ma to duże znaczenie, ponieważ użytkownik nie musi potwierdzać chęci wyszukiwania w żaden sposób - odbywa się to automatycznie. Należy jednak zwrócić uwagę, że zapytania sieciowe kierowane do API nie są wysyłane z każdą wprowadzoną literą. Dzięki operatorowi debounce, możliwe jest spokojne wpisanie całości tytułu, a po upłynięciu zadanego interwału czasowego (tutaj kilkudziesiąt milisekund), rozpocznie się proces wysyłania zapytania do API. Odnaleziona pozycja wyświetlana jest natychmiastowo w chwili znalezienia.

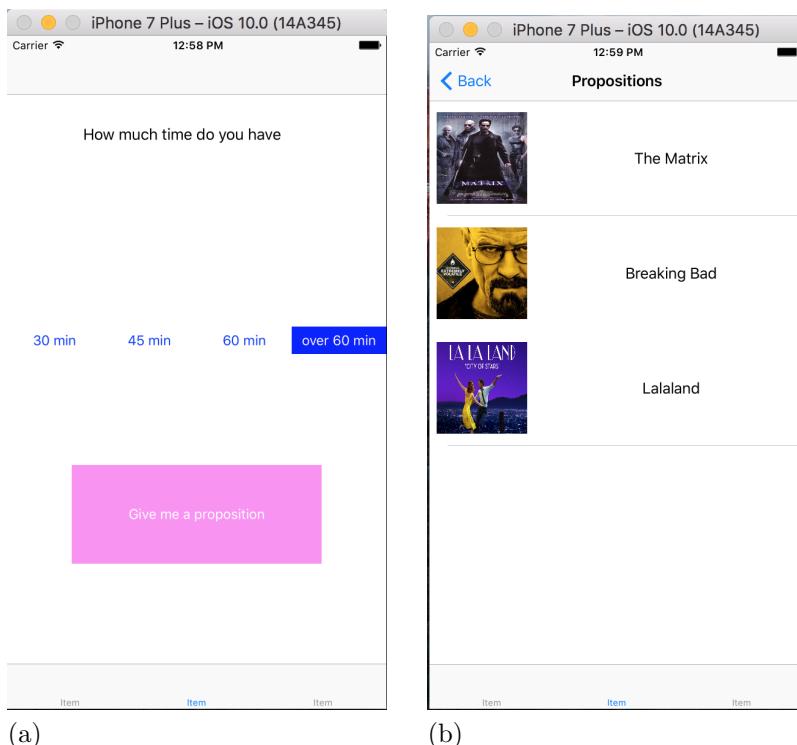
##### Widok szczegółów filmu/serialu

Na widoku przedstawionym na rysunku 4.8, prezentowane są szczegóły dotyczące znalezionej pozycji. Widoczny jest plakat danej produkcji oraz jej ocena wystawiona przez użytkowników serwisu IMDB. Ponadto użytkownik ma możliwość dodania filmu/serialu do swojej bazy jeżeli jeszcze tego nie zrobił. Lokalna baza danych przeszukiwana jest w celu odnalezienia bieżącej produkcji i na tej podstawie decyduje się o sposobie wyświetlenia przycisku.



Rysunek 4.8: Widok szczegółów filmu/serialu

### Widok propozycji

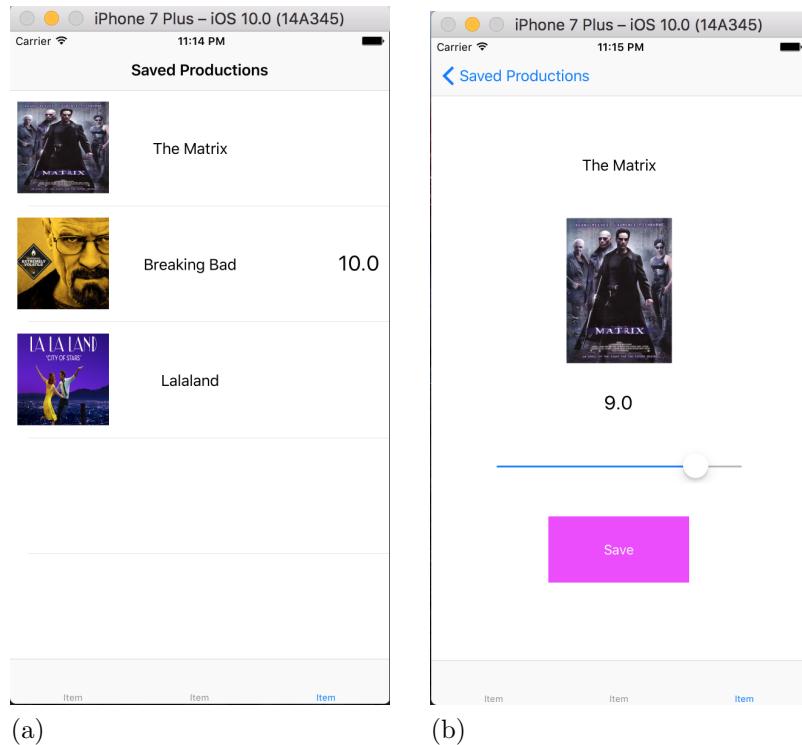


Rysunek 4.9: Widok proponowania pozycji do obejrzenia

Rysunek 4.9 prezentuje ekran propozycji. Użytkownik wybiera jedną z czterech opcji czasowych. Po zaznaczeniu opcji, kliknięcie przycisku spowoduje przeszukanie bazy zapisanych pozycji pod kątem filmów lub seriali spełniających wymóg czasowy, to znaczy,

że czas ich trwania nie będzie dłuższy od czasu wybranego przez użytkownika. Ponadto przez filtr nie przejdą filmy i seriale już obejrzane (a przez to ocenione).

## Widok oceniania



Rysunek 4.10: Widok oceniania zapisanych w bazie produkcji

Na rysunku 4.10 zaprezentowany jest ekran oceniania zapisanych filmów i seriali. Użytkownik ma możliwość przeglądnięcia jakie filmy i seriale zapisał do swojej bazy. Przy tych, które zostały ocenione, wyświetlana jest wartość liczbową reprezentującą jego ocenę. Kliknięcie w dowolną zapisaną w bazie produkcje, przenosi użytkownika do ekranu wystawienia oceny. Wartość wybierana jest przez przesuwanie paska. Przyciśnięcie przycisku powoduje zapisanie oceny do bazy.

# Rozdział 5

## PODSUMOWANIE

Celem niniejszej pracy było zgłębienie tematu programowania reaktywnego oraz za projektowanie i zbudowanie aplikacji mobilnej na system iOS w oparciu o omawiane zagadnienia.

Istnieje wiele implementacji tego podejścia, na różne języki programowania w tym kilka dla omawianego w tej pracy Swifta. Są to m.in.: RxSwift, Reactive Cocoa czy Reactive Swift. Jednakże wszystkie implementacje podejścia reaktywnego bazują na tej samej idei. Dzięki temu nie ma większego znaczenia ani to w jakim języku się tworzy ani to, przy użyciu której biblioteki programowania reaktywnego.

W części teoretycznej, została zawarta duża liczba rysunków i schematów. Zostało to podyktowane specyfiką omawianego tematu. Próg zrozumienia idei programowania reaktywnego, postawiony został dość wysoko. Z tego powodu, dla lepszego przyswojenia poruszanych zagadnień, załączono dużo grafik mających na celu ułatwienie i zobrazowanie mechanizmów reaktywnych.

Pomimo iż programowanie reaktywne nie jest tematem kompletnie nowym, to udokumentowanie go dla języka Swift pozostawia wiele do życzenia. Z tego powodu, w niniejszej pracy posłużowano się dokumentacją do takich platform jak .NET Microsoftu, jak również Java czy JavaScript.

Część praktyczna również stanowiła duże wyzwanie. Za cel, postawiono stworzenie aplikacji mobilnej w oparciu o architekturę MVVM. Po połączeniu z podejściem reaktywnym, wynikiem stało się zadanie o dużym stopniu trudności od strony programistycznej (wysoki poziom abstrakcji i kompletnie inny sposób myślenia) jak i od strony dostępności wiedzy (mała dokumentacja). Należy również zaznaczyć, że projekt aplikacji stworzony został przy użyciu najnowszej implementacji języka Swift (iteracja 3.0) wprowadzającej szereg zmian w API. W czasie powstawania aplikacji, wykorzystywane biblioteki do programowania reaktywnego, uległy zmianie i aktualizacji do najnowszych implementacji.

Pomimo powyższych trudności, udało się stworzyć aplikację mobilną, opartą o architekturę MVVM i implementującą podejście reaktywne.

Podsumowując: programowanie reaktywne wraz z architekturą MVVM na platformy mobilne stanowi nie lada wyzwanie. Jednakże jest to podejście wykorzystujące cały potencjał urządzeń mobilnych i zdecydowanie warte podjęcia próby wdrożenia w projektach komercyjnych. Jest polecane przez doświadczonych i znanych w branży

programistów a coraz więcej firm decyduje się na zastosowanie tego właśnie podejścia [?].

# Bibliografia

- [1] Microsoft Corporation.  
[https://msdn.microsoft.com/en-us/library/hh242985\(v=vs.103\).aspx](https://msdn.microsoft.com/en-us/library/hh242985(v=vs.103).aspx) [online 03.03.2017].
- [2] Apple Inc. "swift has reached 1.0", apple inc.,  
<https://developer.apple.com/swift/blog/?id=14>, [online 03.03.2017].