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

Bartosz Woliński

17 stycznia 2017

Spis treści

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, jako 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 danych w czasie a nie na pojedynczych zdarzeniach. Ponadto takie podejście pozwala oszczędzić dużo kodu i rozważań nad sensownością wydarzeń w czasie m.in dlatego, że zdarzenia synchroniczne jak i asynchroniczne traktowane są w ten sam sposób co znacząco ułatwia tworzenie logiki programu. Podstawą paradygmatu reaktywnego jest wzorzec obserwatora czyli wzorzec obiektu obserwowanego (nadawcy zdarzeń) i obiektu obserwującego (reagującego na odebrane zda-

rzenia).

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 posiadając 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 ninejszej 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

ŹRÓDŁA 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[1] 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.[5][6] Jądrem systemu Darwin jest XNU. Jest to hybrodowe 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 [7]

[1] dane firmy Gartner Inc. za rok 2015: urządzenia mobilne z systemem android - 1,3 mld
urządzenia z systemem iOS - 297 milionów

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 wprowadzał ten system był sklep App Store. Dzięki niemu programiści mogli rozpocząć rozprowadzanie swoich aplikacji na urządzenia iPhone i iPod Touch. [8]

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.[9]

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 tyle za pomocą podwójnego kliknięcia przycisku home. Poprzednie urządzenia ze względu na swoją architekturę nie pozwalały na to.[10]

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 istostny z punktu widzenia programistów, ponieważ oferuje on możliwość tworzenia i wysyłania powiadomień od aplikacji do systemu i wyświetlania ich w górnym pasku.[11]

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.[12]

iOS 7 - Najbardziej znaczący update systemu iOS wg Craiga Federighi'ego.[2] Zaprezentowany w czerwcu 2013 roku. System zaprezentował 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.[13]

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.[14]

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 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 wywoływania funkcji. Ponadto swoją premierę miała druga implementacja języka Swift czyli Swift 2.0 oraz stał się projektem open source.[15]

iOS 10 - Ostatnia opisywana przeze mnie iteracja systemu iOS ujrzała światło dzienne w czerwcu 2016 roku. Według Craiga Federighi'ego jest to największe i najbardziej znaczące wydanie iOS'a. Istotne nowości wprowadzone przez system to m.in. otwarcie API wiadomości dla programistów,

[2]Wiceprzewodniczący działu inżynierii oprogramowania Apple

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.[16][17]

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 [3] 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 MacOS 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 platformę Linux.[3]

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 zwrócą 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 stuktury lub klasy. W Objective-C zwrócenie przez funkcję wartości pustej (pomimo innej spodziewanej) nie powoduje błędów kompilacji ani błędów w czasie działania. W Swficie zaś w takiej sytuacji mielibyśmy do czynienia z błędem kompilacji lub błędem krytycznym w czasie działania co chroni nas przed niespodziewanymi zachowaniami.

[3] 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 funkcji wielu wartości.

Guard - wyrażenie warunkowe w składni Swifta. Zapewnia weryfikacje poprawności oczekiwanego 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 Swficie, podejście do enumeratorów zostało bardzo rozbudowane. Mogą one zawierać metody i być przekazywane przez wartość.

Podejście do funkcji - Każda funkcja w Swifcie 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 jęzków t.j. "try catch". [4]

3.3 Paragydmat reaktywny

Programowanie reaktywne funkcyjne ma swoje początki już w roku 1997[18] lecz popularne stało się za sprawą firmy Microsoft i biblioteki Reactive Extensions dla platformy .NET z 2009 roku[19].
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, defniuje się mechanizm odszukiwania i przekształcania danych, w formie tak zwanego obiektu obserowanego (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.
Niewątpliwą 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

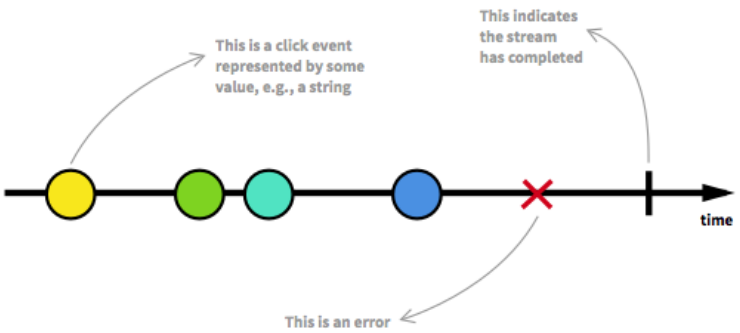
Podstawowym założeniem podejścia reaktywnego jest programowanie oparte o asynchroniczne strumienie (sygnały) niemodyfikowalnych danych.
Źródłem sygnałów mogą być dowolne zdarzenia w czasie takie jak: modyfikacja zmiennych, interakcja użytkownika, pozycja kursora, struktury danych, zapytania sieciowe, 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".

Sygnał gorący zużywa zasoby i emituje zdarzenia bez względu na to czy istnieje zasubskrybowany do niego obserwator. Zdarzeniami są tutaj najczęściej dane o 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.

Sygnał zimny nie zużywa zasoboów ani nie emituje zdarzeń dopóki obserwator nie zacznie nasłuchiwać na zmiany. Zdarzeniami najczęściej są tutaj operacje asynchroniczne, połączenia HTTP, połączenia TCP, połączenia strumieniowane. Zwykle w swojej sekwencji zawiera jedno zdarzenie będące np. wynikiem odpytania sieciowego. Ponadto wyemitowany wynik najczęściej kierowany jest to 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: nasłuchiway 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.



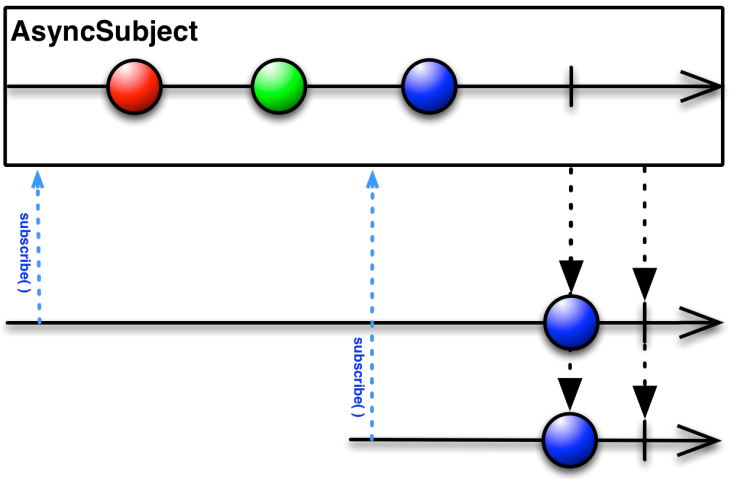
Rysunek 3.1: Schemat emitowanych danych w sygnale

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

Obiekty Subject - stanowią w programowaniu reaktywnym swego rodzaju most, w większości implementacji podejścia reaktywnego, 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 źródołwy 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.

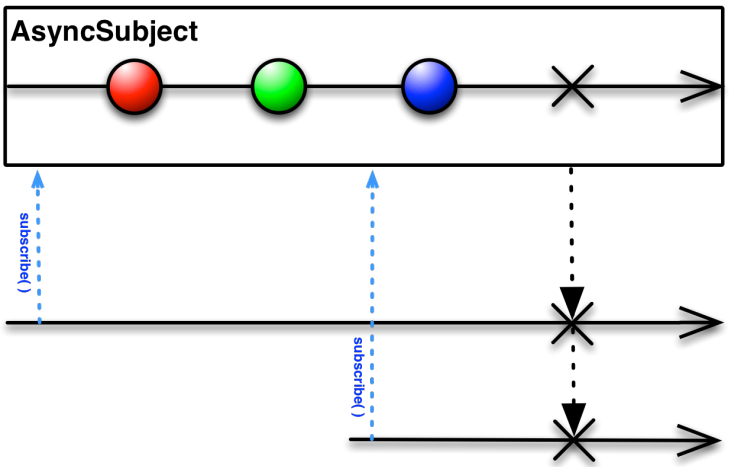


Rysunek 3.2: Schemat poprawnie zakończonego działania obiektu AsyncSubject

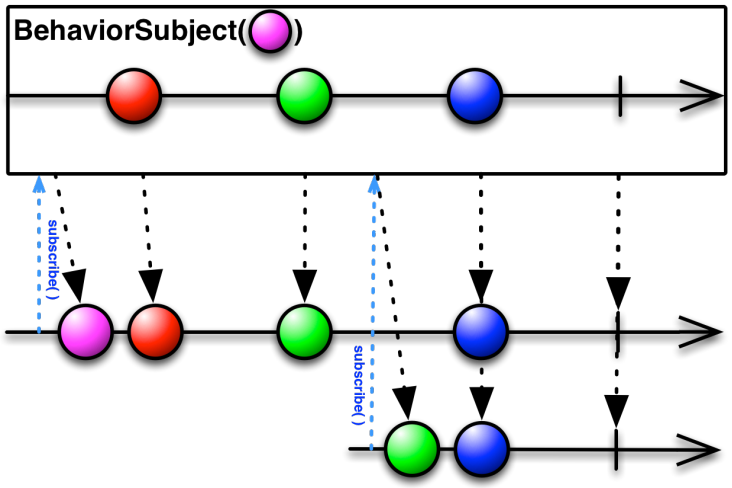
Jeśli zaś źródłowy sygnał zakończy się błędem, obiekt AsyncSubject nie wyemituje żadnych wartości lecz informacje o błędzie od sygnału źródłowego.

BehaviorSubject - gdy obserwator zasubskrybuje się do obiektu BehaviorSubject, rozpoczyna on emitowanie zdarzeń rozpoczynając od ostatniego uprzednio wyemitowanego przez źródłowy sygnał a następnie kontynuuje emitowanie kolejnych zdarzeń sygnału źródłowego.

Jeśli zaś źródłowy sygnał zakończy się błędem, obiekt BehaviorSubject nie



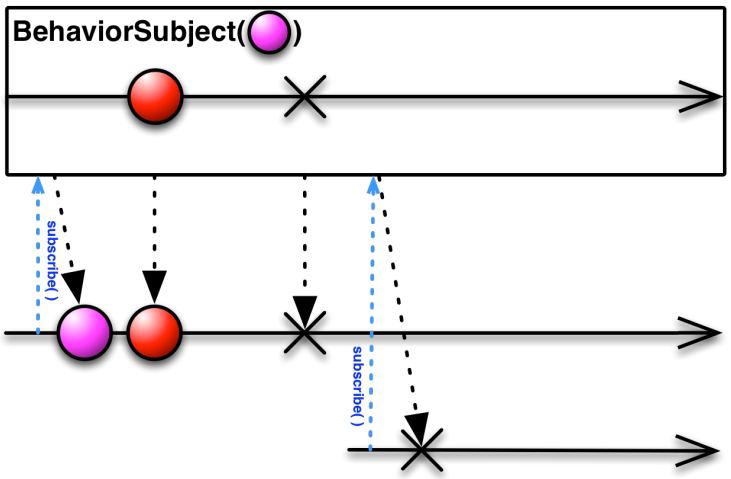
Rysunek 3.3: Schemat działania obiektu AsyncSubject w przypadku błędu, *źródło: http://reactivex.io/documentation/subject.html*



Rysunek 3.4: Schemat działania obiektu BehaviorSubject

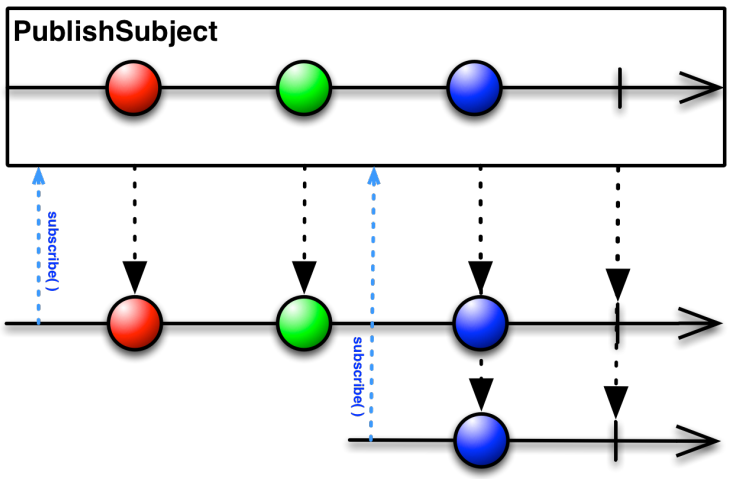
wyemituje żadnych danych do kolejnych obserwatorów, lecz prześle informacje o błędzie z obiektu źródłowego.

PublishSubject - emituje do obserwatora wszystkie dane wyemitowane od momentu subskrypcji. Obiekt PublishSubject może rozpocząć 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



Rysunek 3.5: Schemat działania obiektu BehaviorSubject w przyadku błędu

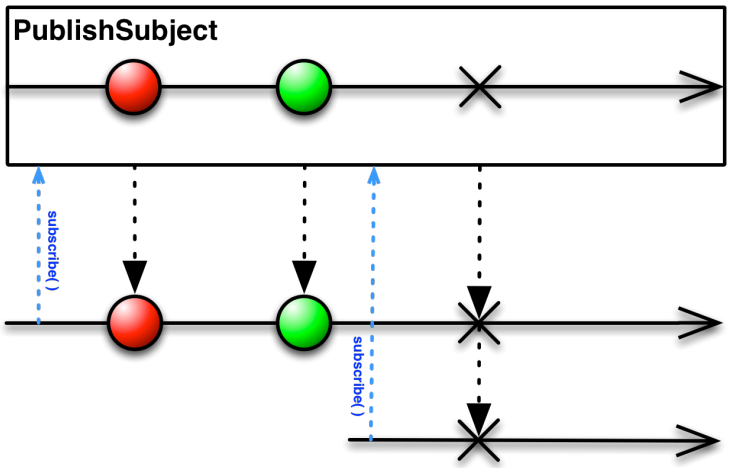
Create i upewnienie się, że zdarzenia nie są emitowane dopóki wszyscy obserwatorzy sie nie zasubskrybują.



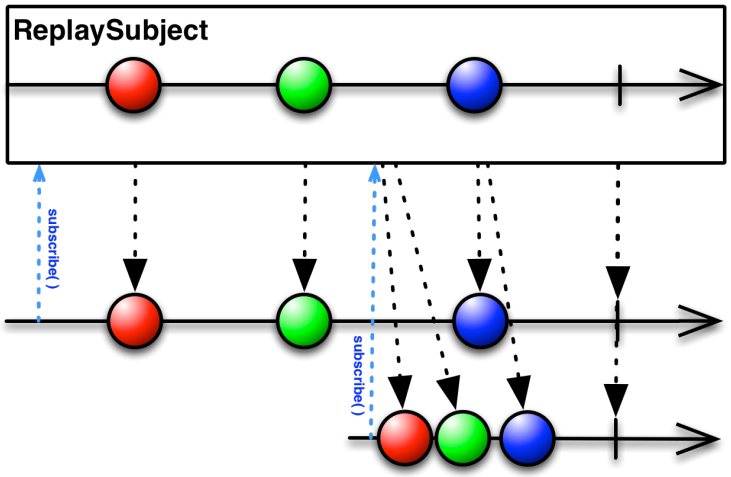
Rysunek 3.6: Schemat działania obiektu PublishSubject

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

ReplaySubject - emituje do obserwatorów wszystkie zdarzenia wyemitowane przez sygnał źródłowy, bez względu na moment subskrybcji. Możliwa jest implementacja obiektu ReplaySubject, której działanie będzie



Rysunek 3.7: Schemat działania obiektu PublishSubject w przypadku błędu



Rysunek 3.8: Schemat działania obiektu ReaplySubject

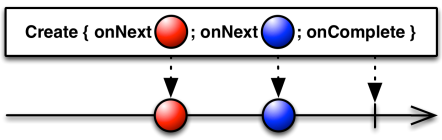
polegało na odrzuceniu starych zdarzeń, czyli obostrzonych rozmiarem bufora bądź upływem zadanej ilości czasu.
Obiekt ReplaySubject używany jako obserwator, dba o to by jego metoda onNext (lub inne metody on) nie były wywoływane z różnych wątków. Może to doprowadzić do powielania już raz wyemitowanych zdarzeń i niejednoznaczności, co stoi w sprzeczności z wytycznymi projektowymi paradgymatu reaktywnego.[20]

3.3.2 Operatory reaktywne

Znaczna część operatorów reaktywnych została zaczerpnięta z podejścia funkcyjnego. Mają na celu umożliwienie tworzenia sygnałów, przekształcania w inne, filtrowania, łączenia, obsługi błędów, itp.
Większość operatorów programowania reaktywnego działa na obiekcie typu Observable a wynikiem 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.

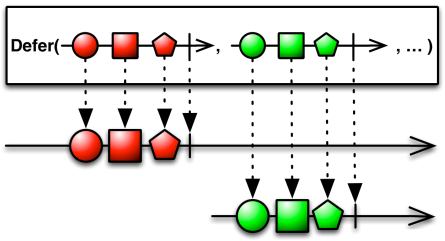
Operatory tworzenia sygnałów

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



Rysunek 3.9: Schemat działania operatora Create

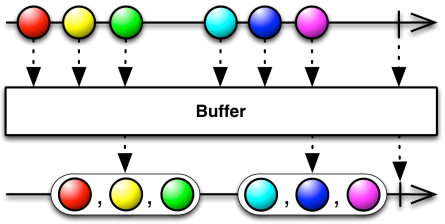
Defer - tworzy obiekt Observable, ale dopiero gdy obserwator się do niego zasubskrybuje. Działanie to jest powielane dla każdego subskrybenta, dlatego, pomimo iż wszyscy nasłuchują z tego samego sygnału, otrzymują indywidualne sekwencje zdarzeń.



Rysunek 3.10: Schemat działania operatora Defer

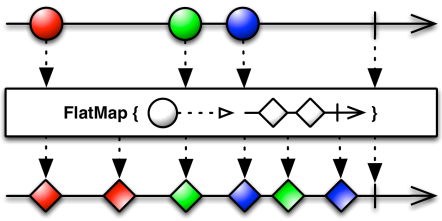
Operatory przekształcania sygnałów

Buffer - cyklicznie zbiera generowane przez obiekt Observable zdarzenia i emituje w postaci pakietów zdarzeń. W przypadku napotkania błędu przez obiekt obserwowany, operator Buffer wyemituje zdarzenie informujące o błędzie bez wytworzenia pakietu danych.



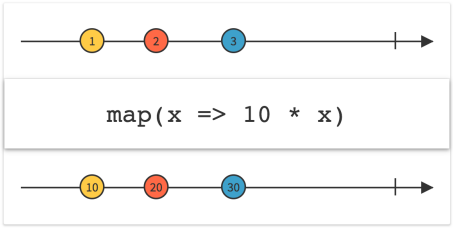
Rysunek 3.11: Schemat działania operatora Buffer

FlatMap - przekształca obiekt Observable przez zastosowanie funkcji zdefiniowanej dla każdego wyemitowanego zdarzenia przez źródłowy obiekt obserwowany, gdzie funkcja ta zwraca obiekt Observble emitujący własne zdarzenia. Nastepnie operator flatMap scala emisje wynikowych obiektów Observable w jedną sekwencję danych.



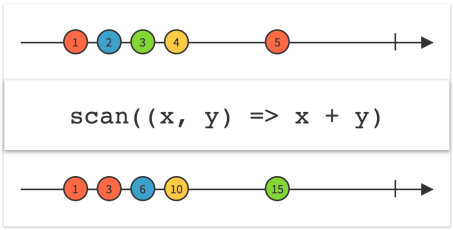
Rysunek 3.12: Schemat działania operatora FlatMap

Map - stosuje zdefiniowaną funkcję do każdego wyemitowanego przez źródłowy obiekt Observable zdarzenia a następnie zwraca obiekt Observable emitujący przekształcone tą funkcją zdarzenia.



Rysunek 3.13: Schemat działania operatora Map

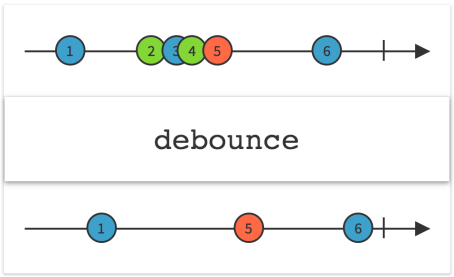
Scan - stosuje zdefiniowaną funkcję do pierwszego wyemitowanego przez źródło zdarzenia a następnie emituje to przekształcone zdarzenie jako swoje pierwsze. Wynik tego zdarzenia podawany jest spowrotem to funkcji wraz z kolejnym zdarzeniem z obiektu źródłowego i emitowany.



Rysunek 3.14: Schemat działania operatora Scan

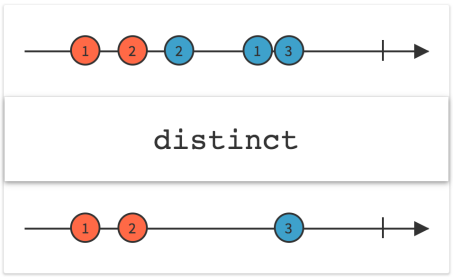
Operatory filtrowania sygnałów

Debounce - emituje zdarzenie z obiektu Observable tylko po upływie zadanego interwału czasowego bez emisji zdarzenia.



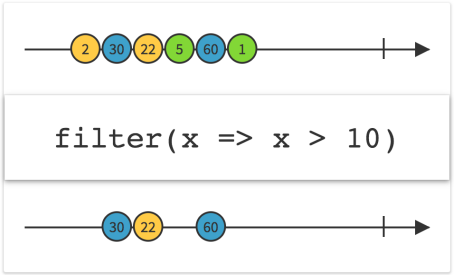
Rysunek 3.15: Schemat działania operatora Debounce

Distinct - ignoruje wartości już raz wyemitowane.



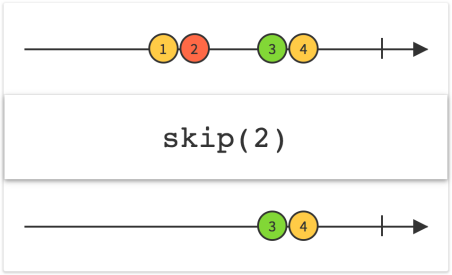
Rysunek 3.16: Schemat działania operatora Distinct

Filter - odfiltrowuje zdarzenia emitowane przez obiekt Observable, spełniające wymagania zadanej funkcji lub predykatu.



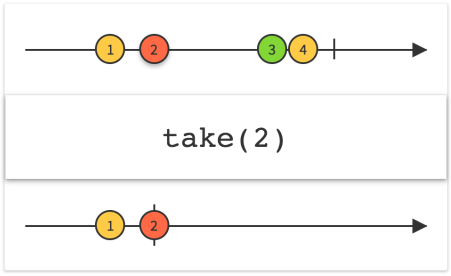
Rysunek 3.17: Schemat działania operatora Filter

Skip - ignoruje n wygenerowanych przez obiekt Observable zdarzeń, gdzie n jest liczbą naturalną, będącą parametrem.



Rysunek 3.18: Schemat działania operatora Skip

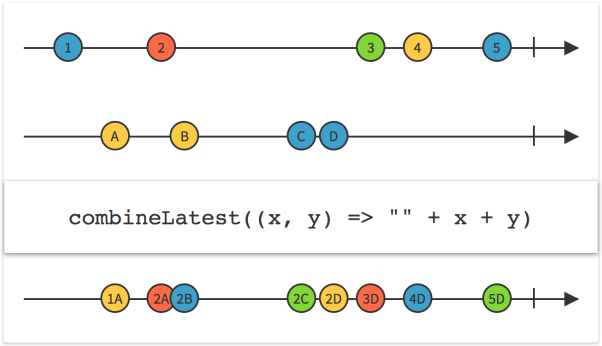
Take - emituje pierwsze n wydarzeń wygenerowanych przez obiekt Observable, gdzie n jest liczbą naturalną, będącą parametrem.



Rysunek 3.19: Schemat działania operatora Take

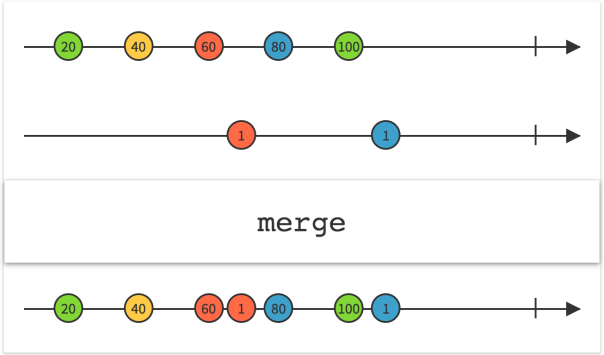
Operatory łączenia

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



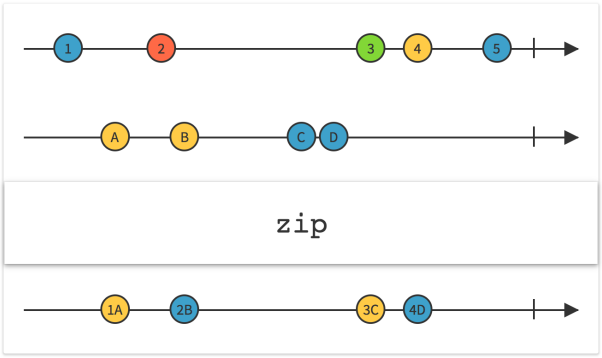
Rysunek 3.20: Schemat działania operatora CombineLatest

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



Rysunek 3.21: Schemat działania operatora Merge

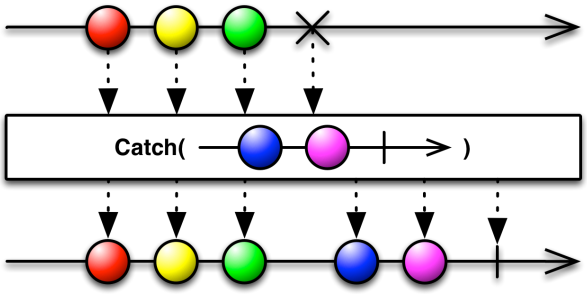
Zip - łączy zdarzenia emitowane z dwóch lub więcej sygnałów w pojedyncze zdarzenia reprezentowane krotką.



Rysunek 3.22: Schemat działania operatora Zip

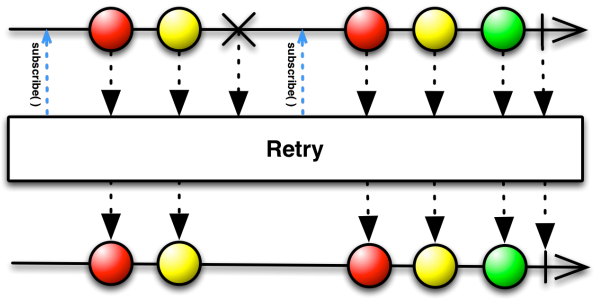
Operatory obsługi błędów

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 w cale.



Rysunek 3.23: Schemat działania operatora Catch

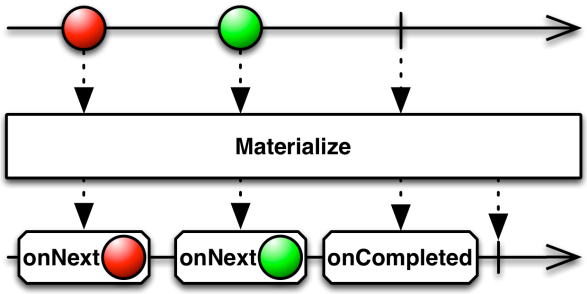
Retry - w przypadku odebrania od obiektu Observable zdarzenia onError, operator Retry nie propaguje błędu do obiektów obserwujących. Zamiast tego wykonywana jest resubskrybcja do źródłowego sygnału. Operator Retry zawsze przekazuje zdarzenie onNext do swoich obserwatorów, nawet z ciągów kończących się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



Rysunek 3.24: Schemat działania operatora Retry

Operatory działań na obiektach Observable

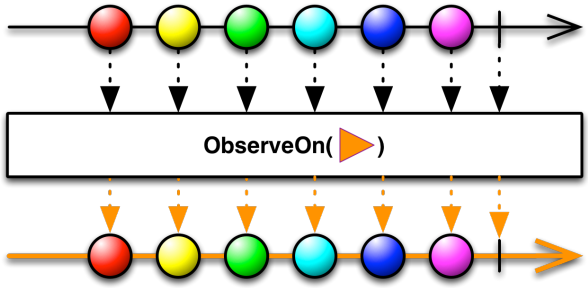
Materialize - konwertuje sekwencje emitowanych z obiektu Observable zdarzeń, na sekwencję zdarzeń reprezentowanych przez wywołania metod onNext i ostatecznie onCompleted lub onError.



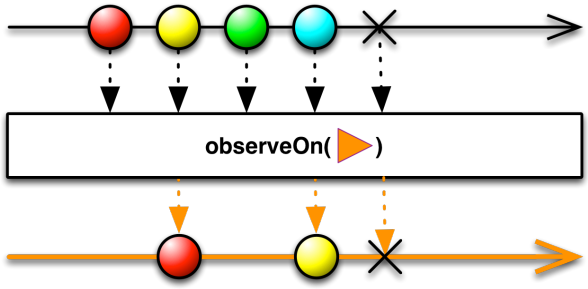
Rysunek 3.25: Schemat działania operatora Materialize

ObserveOn - w środowisku wielowątkowym, określa wątek, na którym obserwatorzy będą nasłuchiwali na generowane zdarzenia.
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 ten obserwator.

SubscribeOn - w środowisku wielowątkowym, określa wątek, na którym obiekt Observable będzie wykonywał pracę. Domyślnie obiekt Observa-

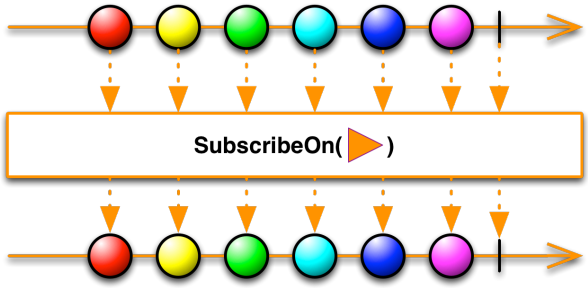


Rysunek 3.26: Schemat działania operatora ObserveOn



Rysunek 3.27: Schemat działania operatora ObserveOnError

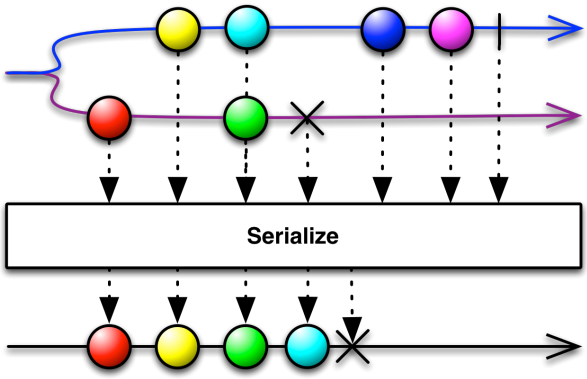
ble jak i ciąg zastosowanych operatorów reaktywnych, będą działały i powiadamiały obserwatorów o zdarzeniach, na tym samym wątku, na którym wywołano metodą Subscribe. Operator SubscribeOn zmienia to zachowanie, przez określenie wątku, na którym obiekt Observable będzie działał.



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 próbo-

wał przesłać zdarzenie onCompleted lub onError przed jakimś innym zdarzeniem onNext lub nawet przesłać kilka różnych zdarzeń onNext z różnych wątków, współbieżnie. Takie zachowania narusza wspomniane wcześniej zalecenia odnośnie projektowania zadań w paradygmacie reaktywnym. Operator Serialize zmusza obiekt Observable do emitowania zdarzeń w sposób synchroniczny.



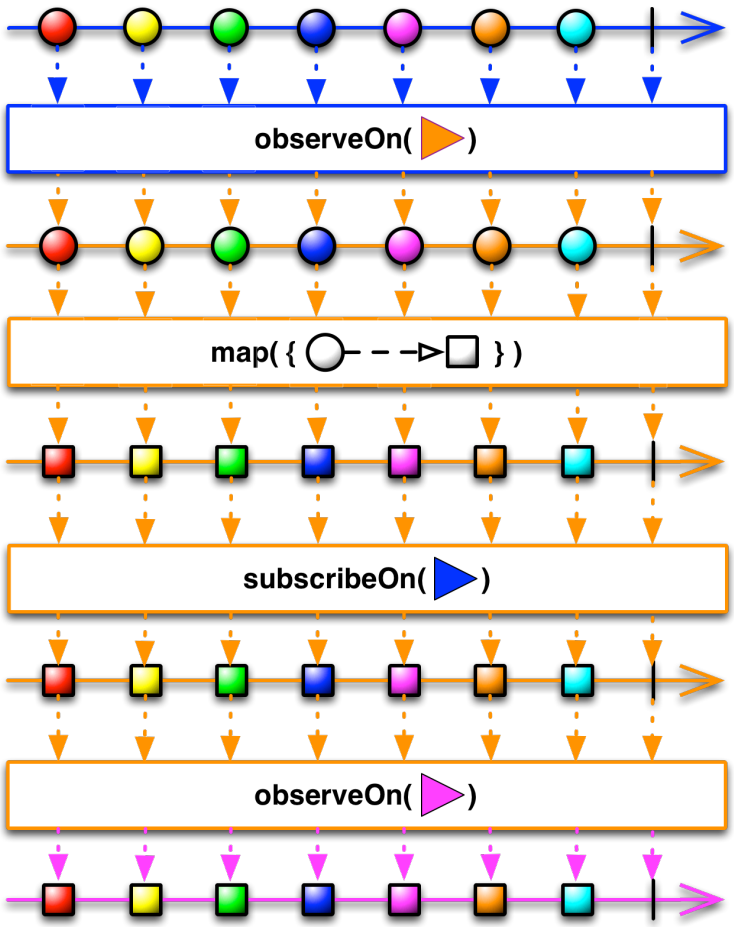
Rysunek 3.29: Schemat działania operatora Serialize

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.

Planiści (Schedulers) 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 poniższej ilustracji, operator SubscribeOn określa wątek, na którym obiekt Observable rozpocznie 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, zaplanowywana 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, wywoływane są już na głównym wątku to zadanie zostane 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) Odsyłą pracę, która ma być wykonana z użyciem konkretnej współbieżnej kolejki typu dispatch_queue_t. Można w argumencie przesłać również kolejkę typu szeregowego i nie powinno stanowić to 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 znacząco 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 zatwierdzenia długiego formularza, przesłania go do aplikacji serwerowej (backend) i po prostym przetwarzaniu, wysłania odpowiedzi spowrotem 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 zamieszczane przez innych użytkowników są ze sobą połączone i również prezentowane w czasie rzeczywistym.

Korzyści z użycia RxSwift (iOS)

Bindowanie Za pomocą mechanizmu bindowania 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 (opisanego 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 reaktywne 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 podejrzanego sygnału i jego pracy na każdym z używanych wątków.

3.3.5 Podejście praktyczne w implementacji na platformie iOS

Na podstawie przykładów[21] 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.
Kolejną 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 porządanym, 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 wyświetlone 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:

```
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 realizujący powyższy przykład 1. *źródło: https://github.com/ReactiveX/RxSwift/blob/master/Documentation/Why.md*

Jak widać na powyższym listingu, niepotrzebne są żadne dodatkowe flagi ani stany.

Przykład 2 - częściowe anulowanie

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.

Oto w jaki sposób można zrealizować powyższą logikę z użyciem podejścia reaktynwego w RxSwift:

```
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 realizujący przykład 2. *źródło: https://github.com/ReactiveX/RxSwift/blob/master/Documentation/Why.md*

Powyższy listing ukazuje 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.

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 dostrzeć, że obsłużona została częsta sytuacja, w której odpowiedzi od API obsługiwane są na wątku w tle a binding tych odpowiedzi do interfejsu użytkownika powininen odbyć się na wątku głównym. Ten problem, rozwiązany został przy użyciu operatora **observeOn**.

```
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 realizujący przykład 3. *źródło: https://github.com/ReactiveX/RxSwift/blob/master/Documentation/Why.md*

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 zdecydowałem się użyć właśnie tego środowiska programistycznego. Ponadto użyłem także 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 stworzyli 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 pozwoli mi na kontrolę postępów mojej pracy jak i na dokumentowanie jej. Oprócz tego użycia GITa powoduje, że błąd popełniony przeze mnie na dalszym etapie mogę odwrócić przywracając poprzedni stan projektu.

4.2 Budowa aplikacji właściwej

4.2.1 Wymagania funkcjonalne

4.2.2 Wymagania niefunkcjonalne

4.2.3 Diagram przypadków użycia aplikacji

4.2.4 Zależności w bazie danych

4.2.5 Diagram klas

4.2.6 Opis wybranych klas

4.2.7 Opis użytych bibliotek i frameworków

4.2.8 Implementacja aplikacji mobilnej - zastosowana architektura

4.2.9 Prezentacja aplikacji

Bibliografia

[1] https://msdn.microsoft.com/en-us/library/hh242985(v=vs.103).aspx - *informacje na temat Reactive Extensions od Microsoft*

[2] https://developer.apple.com/swift/blog/?id=14 - *historia języka Swift*

[3] https://developer.apple.com/swift/blog/?id=34 - *Swift jako język open-source*

[4] https://redwerk.com/blog/10-differences-objective-c-swift - *Różnice między Swiftem a Objective - C*

[5] www.puredarwin.com - *informacje o systemie Darwin*

[6] www.opengroup.org - *informacje o systemach unixowych*

[7] https://web.archive.org/web/20071020040652/http://www.apple.com/hotnews - *informacja o stworzeniu i udostępnieniu iPhone SDK*

[8] http://www.apple.com/pr/library/2008/07/10iPhone-3G-on-Sale-Tomorrow.html - *informacje o iPhone 3G i powstaniu usługi App Store*

[9] http://www.apple.com/pr/library/2009/06/08Apple-Announces-the-New-iPhone-3GS-The-Fastest-Most-Powerful-iPhone-Yet.html - *system iPhone 3.0 i implementacja podstawowych funkcjonalności*

[10] http://www.apple.com/pr/library/2010/04/08Apple-Previews-iPhone-OS-4.html - *informacje o systemie iOS 4*

[11] http://www.apple.com/pr/library/2011/10/04Apple-Launches-iPhone-4S-iOS-5-iCloud.html - *informacje o systemie iOS 5*

[12] http://www.apple.com/pr/library/2012/06/11Apple-Previews-iOS-6-With-All-New-Maps-Siri-Features-Facebook-Integration-Shared-Photo-Streams-New-Passbook-App.html - *informacje o systemie iOS 6*

[13] http://www.apple.com/pr/library/2013/06/10Apple-Unveils-iOS-7.html - *informacje o systemie iOS 7*

[14] http://www.apple.com/pr/library/2014/06/02Apple-Releases-iOS-8-SDK-With-Over-4-000-New-APIs.html - *informacje o systemie iOS 8*

[15] http://www.apple.com/pr/library/2015/06/08Apple-Previews-iOS-9.html - *informacje o systemie iOS 9*

[16] https://github.com/apple/swift-evolution/blob/master/proposals/0005-objective-c-name-translation.md - *informacje o poprawieniu kompatybilności Swift z Objective-C*

[17] http://www.apple.com/pr/library/2016/06/13Apple-Previews-iOS-10-The-Biggest-iOS-Release-Ever.html - *informacje o systemie iOS 10*

[18] Czaplicki, Evan (Apr 2012), Elm: Concurrent FRP for Functional GUIs (PDF) (thesis), Harvard - *pierwsze wzmianki o programowaniu reaktywnym*

[19] http://www.introtorx.com/content/v1.0.10621.0/00foreword.html - *wypuszczenie biblioteki Reactive Extensions*

[20] https://go.microsoft.com/fwlink/?LinkID=205219 - *wytyczne projektowe paradygmatu reaktywnego*

[21] https://github.com/ReactiveX/RxSwift/blob/master/Documentation/Why.md - *przykłady użycia RxSwift*

Rozdział 5

PODSUMOWANIE