Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki
Instytut Informatyki Stosowanej

PRACA DYPLOMOWA INŻYNIERSKA

Wykorzystanie biblioteki RxJava do analizy niegraniczonego strumienia danych

Unbounded data stream processing with use of RxJava library

Autor: Michał Gajewski
Numer albumu: 187690

Opiekun pracy:
dr inż. Radosław Adamus

Łódź, 02.2017

Spis treści

Wstęp

W dobie rozwoju technologicznego, za którym idzie wzrost szybkości działania komputerów czy Internetu, ważny jest również rozwój technik i języków programowania oraz mechanizmów automatyzujący pracę programistów. Wiąże się to przede wszystkim z bezpieczeństwem danych, niezawodnością czy też szybkością działania aplikacji. W związku z tym powstaje wiele przydatnych systemów bądź mechanizmów, które przyspieszają pracę programistów, a jednocześnie dbają o to, aby zapewnić jak największy poziom bezpieczeństwa. Jednym z nich mogą być repozytoria i systemy kontroli wersji.

W dzisiejszych czasach serwisy czy różnego rodzaju portale przetwarzają ogromną ilość danych. Wykorzystując stare mechanizmy, byłoby to bardzo czasochłonne, co wpłynęłoby na poziom niezadowolenia użytkowników korzystających z tym usług. Dlatego też powstało między innymi strumieniowanie danych, które w znaczny sposób przyspieszyło ten mechanizm, dodając wiele funkcjonalności pozwalających modyfikować przetwarzane dane.

Pomysł na pracę dyplomową jest połączeniem tych dwóch tematyk. W związku tematem autor wyznaczył sobie dwa główne cele. Pierwszym z nich było przedstawienie zagadnień dotyczących strumieni danych wykorzystywanych w programowaniu oraz omówienie tematyki repozytorium i systemu kontroli wersji. Drugim natomiast było napisanie aplikacji w języku Java, której cel to zasymulowanie analizy nieskończonego strumienia danych, zawierających informacje odnośnie serwisu GitHub, który wykorzystuje system kontroli wersji Git oraz tworzenie statystyk odnośnie wykorzystywania konkretnych mechanizmów (eventów) na tym serwisie. Dzięki tej aplikacji możliwe będzie zobrazowanie, które eventy są najczęściej wykorzystywane na wspomnianym serwisie.

1. Strumień danych

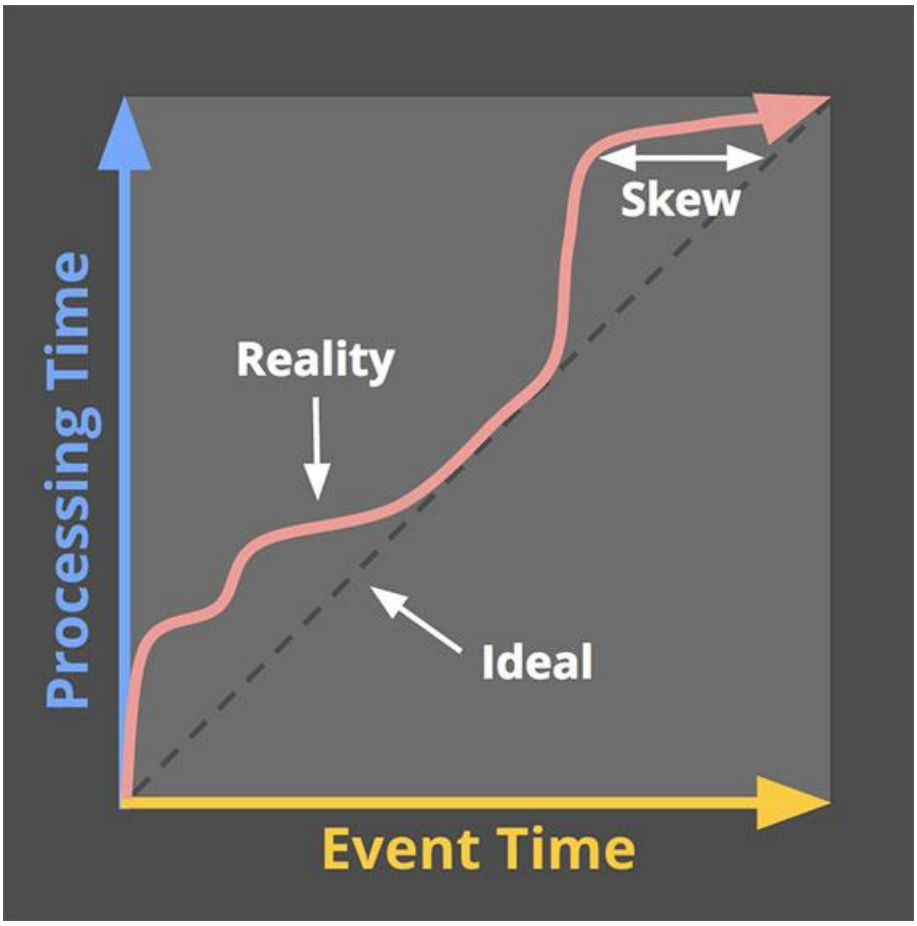
1.1 Definicja strumienia oraz idea strumieniowania danych

Strumień (ang. stream) to „połączenie między nadawcą a odbiorcą w postaci nieprzerwanej sekwencji danych, niedzielonych na komunikaty”.[1] Dane te dostarczane są w zmiennych odstępach czasowych i odczytywane tylko raz.

Ostatnimi czasy bardzo popularne jest pojęcie Big Data. Jest to termin, który odnosi się do danych o dużej objętości, szerokim zakresie typów, których analiza jest stosunkowo trudna.[2] Jednakże bardzo często niesie ona ze sobą wiele cennych informacji. Wyobraźmy sobie, że dane te zajmują kilkaset terabajtów pamięci. Gdybyśmy chcieli dokonać analizy tych danych, traktując te dane, jako statyczne, wówczas pierwszym krokiem byłoby wczytanie wszystkich tych danych do programu, który odpowiedzialny byłby za ich analizę. Przy tak dużej ilości danych, trwałoby to bardzo długo. W przypadku strumienia, dane byłyby dostarczane w mniejszych częściach i program mógłby w międzyczasie dokonywać analizy tych, które już posiada. Świadczy to o tym, ze strumienie są dobrym sposobem na przyspieszenie pracy nad dużą ilością danych.

Strumień jest mocno związany z pojęciem nieskończonych danych (ang. unbounded date). Jest on jak woda w rzece, dopóki nie odetniemy źródła, będzie płynęła bez końca. Tak samo jest z danymi płynącymi w strumieniu. Wiąże się to również z nieskończonym procesem przetwarzania tych danych. Mówiąc o nim, należy zwrócić uwagę na dwa pojęcia związane z czasem: czas zdarzenia i czas przetworzenia. Pierwsze z nich określa czas, w którym faktycznie miało miejsce dane zdarzenie, drugie natomiast czas, gdy zdarzenie to jest obserwowane przez system. Nie zawsze istotne jest, kiedy miało miejsce dane zdarzenie, jednakże w większości przypadków tak. W idealnym świecie, czas wystąpienia wydarzenia i przetworzenia go przez system powinien zawsze być taki sam. Oznacza to, że zdarzenie zostanie obsłużone natychmiast po jego wystąpieniu. W rzeczywistości różnica ta nie tylko nie jest zerowa, ale również zdarzają się większe odchylenia czasowe. Związane mogą one być miedzy innymi z:

- ograniczoną liczbą zasobów, takimi jak przeciążenie sieci czy dzielenie CPU
- przyczyny związane z oprogramowaniem, takie jak logika systemu czy błędy
- rodzajem danych dostarczanych do systemu



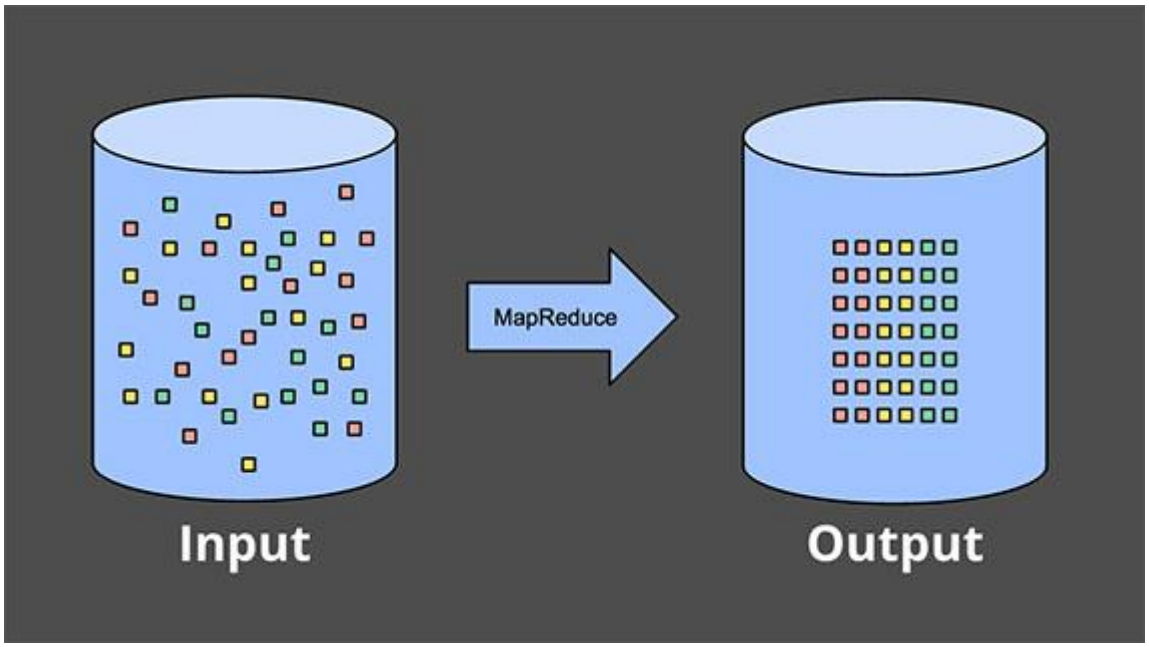
Rysunek 1.1 Wykres obrazujący czas wywarzenia i czas przetworzenia danych
(źródło: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)

Na rysunku 1.1 przedstawiony jest przykładowy wykres zależności czasu wydarzenia (Event Time), a czasu przetworzenia (Processing Time). Czarna przerywana linia obrazu to idealny przypadek, gdy oba czasy są takie same. Czerwona linia natomiast obrazuje, jak faktycznie wyglądają zależności obu czasów. Na podstawie tego przykładowego wykresu można odczytać, że w początkowej i końcowej części pracy system działa z niewielkim opóźnieniem, natomiast w środkowej części praca systemu jest bliska ideału.

W związku z tym, że odwzorowanie czasu pomiędzy czasem wydarzenia i przetworzenia nie jest statyczne, nie możemy dokonać analizy danych w przypadku, gdy zależy nam na jak dokładniejszym czasie wydarzenia. Aby sobie z tym poradzić, systemy zapewniają pewne pojęcie, jakim jest okienkowanie danych. W dużym skrócie polega ono na tym, że dzielimy zbiór danych do skończonych kawałków o określonych granicach czasowych.

1.2 Przetwarzanie danych

Przetwarzanie danych skończonych dosyć proste i prawdopodobnie znane każdemu z nas. Najprościej zobrazować to na podstawie przykładu.



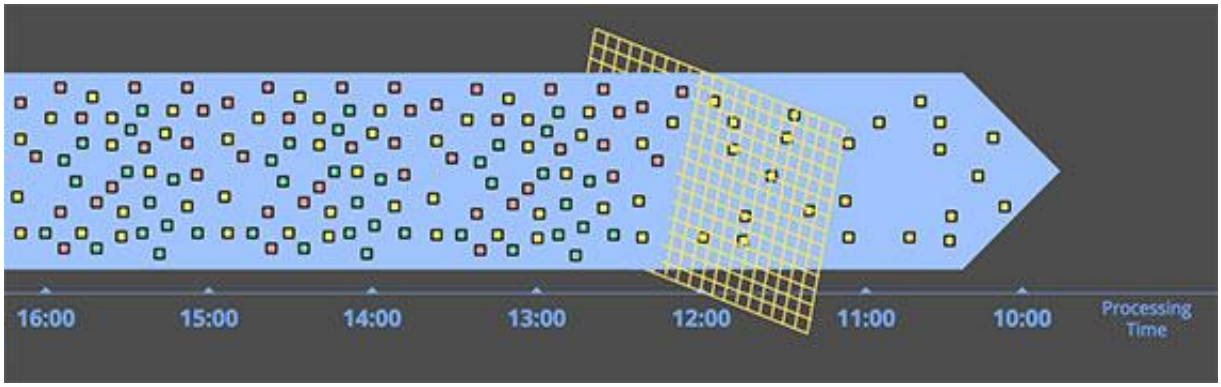
Rysunek 1.2 Przykładowe przetwarzanie danych skończonych
(źródło: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)

Na powyższym diagramie (rys. 1.2) przedstawione zostało przykładowe przetwarzania skończonych danych. Po lewej stronie znajdują się dane wejściowe (Input). Następnie przetwarzane są przez specjalny mechanizm MapReduce. Może nim być każda funkcja napisana przez programistę, której celem jest selekcja danych wejściowych względem np. wartości czy atrybutu. Po prawej stronie znajduje się nowa struktura (Output) z przetworzonymi danymi.

Pomimo tego, że istnieje mnóstwo wariacji przetwarzania takich danych, schemat obrazujący ten proces będzie dosyć podobny. Dużo ciekawiej natomiast przedstawiają się modele nieskończonych danych. Przedstawione zostanie teraz kilka przykładów modyfikacji nieskończonych danych, dostarczanych w postaci strumienia.

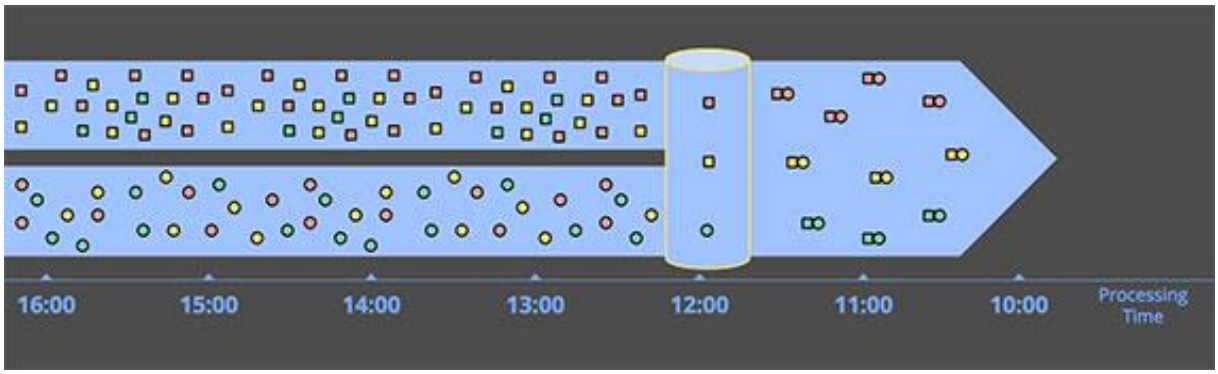
Filtrowanie strumienia jest jednym z przykładów jego modyfikacji (rys. 1.3). Wyobraźmy sobie sytuacje, że chcemy przetworzyć dziennik aktywności serwisu internetowego. Interesuje nas tylko cykliczne wystąpienie danego eventu, lecz strumień dostarcza nam mnóstwo innych, zbędnych informacji. W tym wypadku możemy na strumień

założyć specjalny filtr, który będzie przepuszczał tylko te informacje, które nas interesują w danej sytuacji.



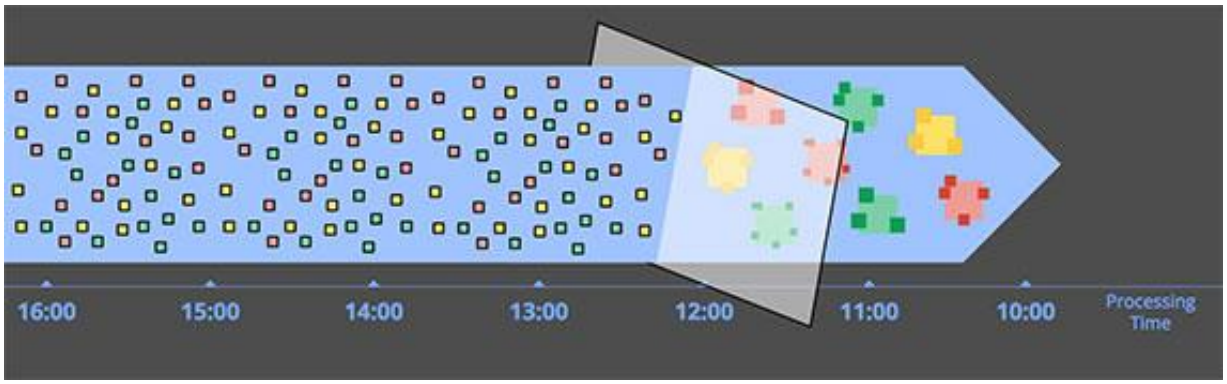
Rysunek 1.3 Przykładowe użycie filtra na strumieniu danych
(źródło: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)

Kolejnym przykładem modyfikacji jest łączenia dwóch strumieni danych (rys. 1.4) . Ważną kwestią jest informacja, czy interesuje nas w jaki sposób dojdzie do połączenia elementów dwóch różnych strumieni. W takim przypadku dane jednego strumienia umieszczane będą w specjalnym buforze, którym czekały będą na odpowiedni rodzaj danych z drugiego strumienia.



Rysunek 1.4 Przykładowe użycie łączenia dwóch strumieni
(źródło: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)

Możemy również dokonywać modyfikacji danych znajdujący się w strumieniu. Przykładem takiego mechanizmu mogą być różnego rodzaju algorytmy aproksymacji (rys 1.5). W prosty sposób można wyjaśnić to na podstawie muzyki. Przyjmijmy, że do naszego programu dostarczany jest strumień danych, zawierający informacje dźwiękowe, czyli bity i tony piosenki. Następnie modyfikujemy wszystkie tony niskie, aby bardziej podkreślić bass utworu. Dzięki temu w głośnikach słyszymy zmodyfikowaną piosenkę z podkreślonymi tonami niskimi.



Rysunek 1.5 Przykładowe użycie algorytmu aproksymacji na strumieniu
(źródło: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)

Na podstawie opisanych przez autora przykładów można przypuszczać, że strumieniowanie danych posiada wiele ciekawych i dobrze rozwiniętych mechanizmów. W dobie dzisiejszych olbrzymich baz danych jest ciekawym sposobem, przyspieszającym prace między innymi na analizie i przetwarzaniu tych danych. Staje się on coraz bardziej powszechny, coraz więcej znanych serwisów z niego korzysta. Przykładowymi mogą być: Netfix, Trello, SoundCloud, GitHub i wiele innych.

2. Repozytorium i system kontroli wersji

2.1 Definicja repozytorium

Repozytorium jest rozwiązaniem technologicznym, które służy przede wszystkim do zapisywania, składowania oraz udostępniania przeróżnego rodzaju danych. Służy ono także do wersjonowania. Oznacza to, że zapisując ponownie plik, nie nadpisujemy go, tylko staje się on nową wersją, natomiast możemy zawsze sięgnąć do wersji poprzedniej. Założeniem repozytorium jest również selekcjonowanie i porządkowanie danych zgłoszonych do przechowania.[3]

2.2 Podstawowe informacje

Jak wyjaśnia nam powyższa definicja, pracując na repozytorium możemy korzystać nie tylko z najnowszych wersji plików. Poza tym zapisywane są w nim również informacje dotyczące między innymi, kiedy została dostarczona nowa wersja pliku, jaka jest różnica między poprzednią wersją oraz kto jest autorem tych zmian. Dzięki temu w przypadku dostarczenia błędnie działającego pliku lub posiadającego niepoprawne dane, w łatwy sposób można zidentyfikować autora tego problemu oraz powrócić do starszej wersji pliku, która nie posiadała tego błędu.

Ten przykład oraz wiele innych jest możliwy dzięki systemowi obsługującemu repozytorium, który jest nazwany systemem kontroli wersji. Jest to oprogramowanie, które śledzi wszystkie zmiany dokonywane w jednym lub wielu plikach oraz umożliwia przywrócenie dowolnej wcześniejszej wersji. Systemy kontroli wersji dzielimy na:

- lokalne
- scentralizowane
- rozproszone

Lokalne systemy kontroli wersji to takie, które pozwalają tworzyć repozytoria lokalnie, to znaczy na lokalnym komputerze. Jest to, zatem duże utrudnienie dla programistów, gdyż z takiego repozytorium może korzystać tylko jedna osoba pracująca w danej chwili na komputerze, na którym znajduje się to repozytorium. Scentralizowane oraz rozproszone systemy kontroli wersji pozwalają natomiast na przechowywanie plików na serwerze, dzięki czemu rozwiązują problem ilości użytkowników, którzy mogą pracować w danej chwili nad

tymi samymi plikami. Główną różnicą, która dotyczy tych dwóch systemów, jest kwestia bezpieczeństwa danych. W przypadku systemów scentralizowanych pliki użytkowników znajdują się na jednym serwerze, przez co w przypadku uszkodzenia tego serwera, możliwa jest utrata przechowywanych tam danych. Natomiast systemy rozproszone przechowują pliki nie tylko na serwerze, ale także lokalnie na komputerach użytkowników. Jeśli zatem serwer ulegnie uszkodzeniu, dzięki temu rozwiązaniu możliwe jest łatwe i szybkie odzyskanie danych. Najpopularniejszymi rozproszonymi systemami kontroli wersji, z których korzystają programiści są Git oraz Mercurial.[4,5] Poniższe podrozdziały opisywały będą zagadnienia związane z jednym z wymienionych systemów, którym jest Git.

2.3 Definicja Gita

Git – rozproszony system kontroli wersji, który został stworzony przez Linusa Torvaldsa, czyli fińskiego programisty, który jest również twórcą jądra Linux. Został on stworzony, jako narzędzie wspomagające rozwój wspominanego jądra. Pierwsza wersja tego narzędzia została wydana 7 kwietnia 2005 roku. Główne cechy tego narzędzia to:

- efektywna praca z dużymi projektami, gdyż jest on dużo szybszy od konkurencji,
- posiada możliwość tworzenia gałęzi od danej wersji plików oraz algorytmy łączenia dwóch gałęzi w jedną, jest również możliwość dodania własnego,
- programiści pracujący na repozytorium nie potrzebują połączenia z siecią, gdyż każdy posiada jego lokalną kopie,
- każda rewizja, czyli wprowadzenie zmian do repozytorium, powoduje stworzenie kopii nie tylko zmienionych plików, ale całego obrazu aktualnego stanu projektu

Do najpopularniejszych serwisów, które umożliwiają tworzenie repozytorium gita są między innymi: GitHub, Bitbucket, Gitorious i Stash.[4]

2.4 Obsługa Gita

Podrozdział ten przedstawia podstawowe funkcje systemu kontroli wersji Git oraz jego mechanizmów wraz z możliwościami, jakie otrzymujemy korzystając z niego. Poniższe funkcje tego systemu będą wywoływane poprzez wpisywanie komend w terminalu systemu operacyjnego. Jeżeli posiadamy już zainstalowane oprogramowanie do obsługi gita, pierwszym etapem pracy z repozytorium jest jego zainicjowanie. Do tego celu służy komenda *git init.* Przyjmijmy, że posiadamy już lokalnie stworzone repozytorium gita. Jest to miejsce

w przestrzeni dyskowej, przykładowo folder na pulpicie, w którym znajduje się ukryty folder ‘.git’. W nim znajdują się między innymi pliki związane z ustawieniami repozytorium czy archiwum starszych rewizji projektów lub plików dla tego konkretnego repozytorium. Kolejnym etapem jest dodawanie plików do naszego nowo stworzonego repozytorium. Należy zwrócić uwagę, że pliki, które znajdują się w przestrzeni dyskowej naszego repozytorium mogą być w 4 stanach:

- untracked
- staged
- modified
- unmodified

Przedstawmy, zatem różnice tych czterech stanów. Untracked files są to, w dosłownym tłumaczeniu pliki nieśledzone. Oznacza to, że system kontroli wersji widzi takie pliki, ale nie traktuje ich, jako plików należących do repozytorium, nie ma z nimi nic wspólnego, co za tym idzie nie będzie zapisywał zmian, jakie wykonywane są na nich. Staged files są to pliki, które nie są jeszcze ostatecznie w repozytorium, ale git posiada wszystkie informacje na temat zamian, jakie w nich nastąpiły w porównaniu z ich starszymi rewizjami znajdującymi się w repozytorium. Modified files to pliki, które zostały zmodyfikowane w porównaniu z rewizją, jaka znajduje się w repozytorium, natomiast unmodified files to takie, których stan jest identyczny ze stanem znajdujący się w repozytorium. Posiadamy więc już informacje odnośnie stanów plików, jak zatem spowodować, aby zostały one dodane do repozytorium. Pierwszym etapem jest zamiana plików, które chcemy dodać do repozytorium lub które zostały zmodyfikowane na stan staged. Służy do tego komenda *git add.* Gdy system posiada już wszystkie informacje dotyczące zmian w plikach, należy zatwierdzić wszystkie wprowadzone zmiany. Służy do tego komenda *git commit.* W tym momencie zostały dodane wszystkie zmiany do repozytorium i stworzona została nowa rewizja plików. Każda z komend, opisany powyżej, posiada szereg parametrów, które modyfikują ich działanie. Jednakże w tym momencie najważniejsze było to, aby przedstawić podstawowe działanie dodawania plików do repozytorium.

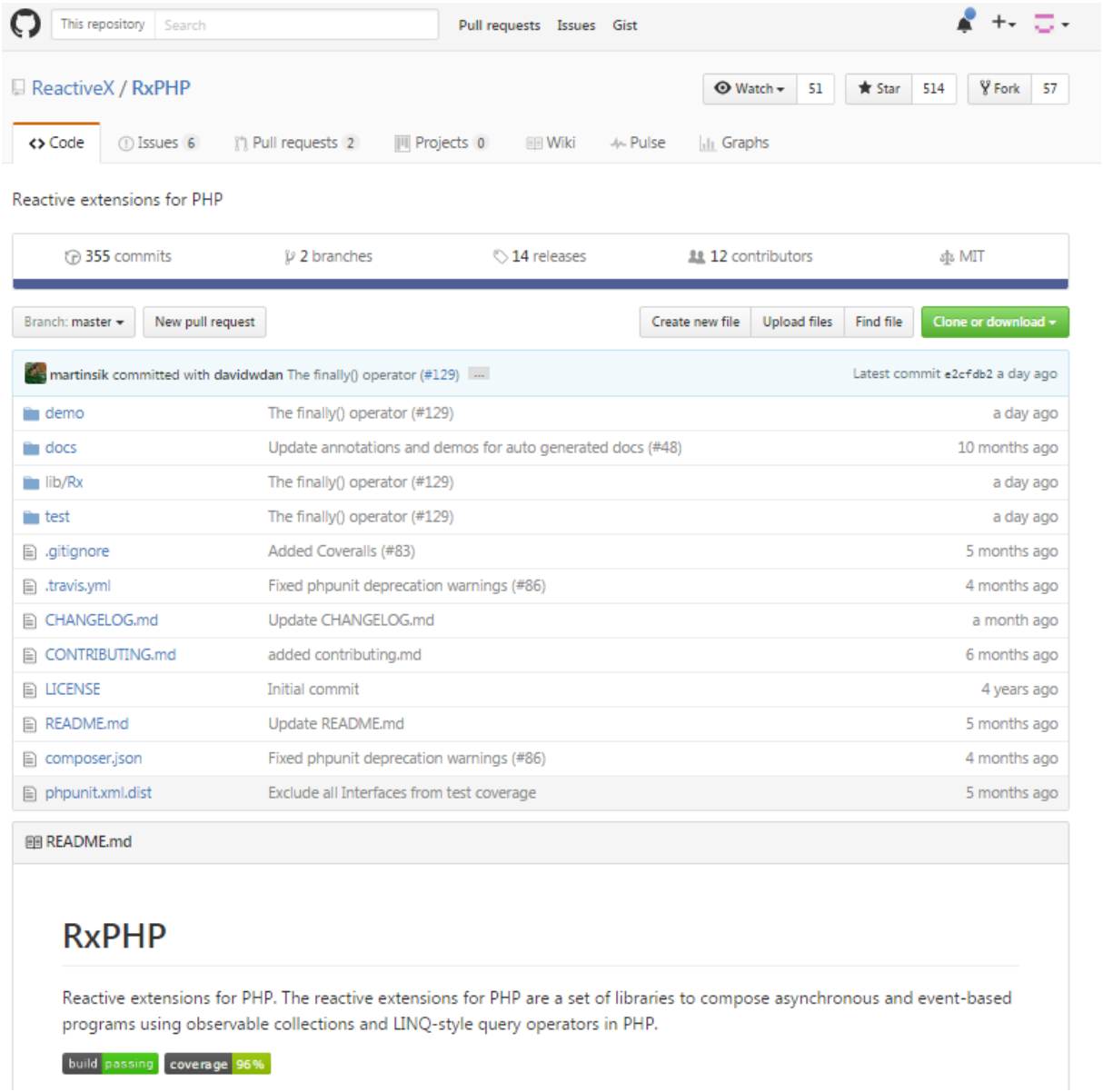
Git posiada również wiele innych przydatnych funkcjonalności. Oto kilka przykładowych:

- *git push* – pozwala na wypychanie zatwierdzonych zmian na repozytorium do podłączonego z nim serwerem
- *git status* – wyświetla informacje odnośnie stanów plików
- *git branch* – pozwala na stworzenie nowej gałęzi od aktualnego stanu plików
- *git amend* – pozawala na modyfikowanie ostatnio zatwierdzonych zmian
- *git log* – wyświetla informacje odnośnie ostatnich commit'ów
- *git config* – pozwala na dostosowanie konfiguracji gita do własnych potrzeb
- *git checkout* – pozwala na przejście na inną gałąź repozytorium
- *git reset* – pozwala na resetowanie zmian
- *git clone* – pozwala na sklonowanie repozytorium do innej lokalizacji
- *git pull* – pozwala na aktualizacje lokalnego repozytorium do stanu, jaki znajduje się na serwerze

2.5 GitHub

W powyższym rozdziale opisana została podstawowa obsługa systemu Git. Jak wynika z definicji rozproszonego systemu kontroli wersji, pliki repozytorium znajdują się lokalnie jak i na serwerze. Jednym z najbardziej popularnych serwisów, które umożliwiają tworzenie i przechowywanie repozytorium gita jest serwis GitHub.

GitHub to serwis, którego data powstania przypada na kwiecień 2008. Został stworzony przy użyciu frameworka Ruby on Rails i języka Erlang. Według danych z kwietnia 2016 roku serwis ten posiada ponad 14 milionów użytkowników oraz ponad 35 milionów repozytoriów. Aktualnie znajduje się na 14 miejscu rankingu Forbes'a odnośnie prywatnych firm, które udostępniają swoje usługi, jako chmurę obliczeniową, czyli różnego rodzaju usługi obliczeniowe za pośrednictwem serwerów czy baz danych (stan na 29.01.2017). Został on napisany przez programistów Chris Wanstrath, PJ Hyett, and Tom Preston-Werner.[6]



Rysunek 2.1 Przykładowy projekt z serwisu GitHub.com *(źródło: opracowanie własne)*

Na podstawie przykładowego projektu znajdującego się na serwisie GitHub (rys. 2.1), możemy zaobserwować kilka podstawowych funkcjonalności tego serwisu. Przedstawione zostanie kilka tych, które widać na głównej stronie danego repozytorium. Na samym środku widnieje menedżer wszystkich plików i katalogów danego repozytorium. Możliwe jest podejrzenie pliku, jeżeli tylko GitHub udostępnia możliwość podejrzenia pliku z tym rozszerzeniem. Poniżej niego wyświetlony jest plik README.md, w którym zazwyczaj zapisywane są przez programistów informacje dla osób, które weszły na to repozytorium, czego dotyczy dany projekt. W przypadku oprogramowania znajdują się tam zazwyczaj informacje odnośnie wersji i sposobu instalacji. W górnej części strony natomiast znajduje się między innymi zakładka ‘commits’, po wejściu w którą wyświetli się posortowana lista

wszystkich commitów, z najnowszym na samej górze. Daje ona możliwość podejrzenia zmian wykonanych na plikach w danym commicie porównując ze stanem plików z poprzednim. Obok zakładki ‘commits’ znajdują się też, 4 inne zakładki, między innymi lista użytkowników przypisanych do danego repozytorium, lista gałęzi w danym repozytorium czy też lista wersji danego projektu. Jest również możliwość pobrania danego repozytorium. To tylko kilka wybranych funkcjonalności, które oferuje nam serwis GitHub.

2.6 GitHub Archive

Programiści z całego świata pracują nad milionami projektów znajdującymi się w serwisie GitHub: piszą kod i dokumentacje do niego, naprawiają błędy itp. GitHub Archive jest projektem, który w obserwuje aktywność na publicznych repozytoriach GitHub’a, archiwizuje ją i w łatwy sposób udostępnia do analizy i tworzenia statystyk. Ciekawą informacją odnośnie tego projektu jest to, że znajduje się na repozytorium w serwisie GitHub. Swoją prace rozpoczął od 12 lutego 2011 roku. Jego główna funkcjonalność polega na zbieraniu informacji odnośnie eventów wykonywanych na publicznych repozytoriach GitHub’a, a następnie tworzeniu z nich plików w formacje .json z godzinnej pracy serwisu.

JSON (JavaScript Object Notation) jest to lekki format wymiany danych komputerowych, obsługiwany przez wiele języków programowania, zazwyczaj poprzez dodatkowe biblioteki bądź pakiety. Pliki JSON budowane są z kolekcji obiektów, z który każdy z nich składa się z pary nazwa : wartość.[7]

```
{
    "id": "2489651045",
    "type": "CreateEvent",
    "actor": {
        "id": 665991,
        "login": "petroav",
        "gravatar_id": "",
        "url": "https://api.github.com/users/petroav",
        "avatar_url": "https://avatars.githubusercontent.com/u/665991?"
    },
    "repo": {
        "id": 28688495,
        "name": "petroav/6.828",
        "url": "https://api.github.com/repos/petroav/6.828"
    },
    "payload": {
        "ref": "master",
        "ref_type": "branch",
        "master_branch": "master",
        "description": "Solution to homework and assignments from MIT's 6.828
        "pusher_type": "user"
    },
    "public": true,
    "created_at": "2015-01-01T15:00:00Z"
}{
    "id": "2489651051",
    "type": "PushEvent",
    "actor": {
        "id": 3854017,
```


Rysunek 2.2 Przykładowy obiekt w formacie JSON *(źródło: opracowanie własne)*

Każdy przykładowy obiekt w formacie JSON wygenerowany przez GitHub Archive (rys. 2.2) posiada kilka wspólnych cech. Obiekty najbardziej zewnętrzne to obiekty jednego konkretnego eventu wykonanego na publicznym repozytorium GitHub'a. Każdy z nich posiada w sobie inne obiekty typu JSON. Istnieje kilka obiektów, które znajdują się w każdym evencie. Pierwszym z nich jest id, który posiada unikalny numer wykonanego eventu. Kolejnym z nich jest typ eventów, którego dotyczy dany obiekt. Poniżej znajdują się obiekty takie jak:

- actor – zawiera informacje odnośnie użytkownika, który wykonał ten event, czyli między innymi id użytkownika, login czy link do profilu,
- repo – zawiera informacje odnośnie repozytorium, na którym został wykonany konkretny event, czyli id repozytorium, nazwę oraz adres URL,
- payload – zawiera informacje zależne od rodzaju eventu
- created_at – zawiera informacje odnośnie czasu wykonania eventu

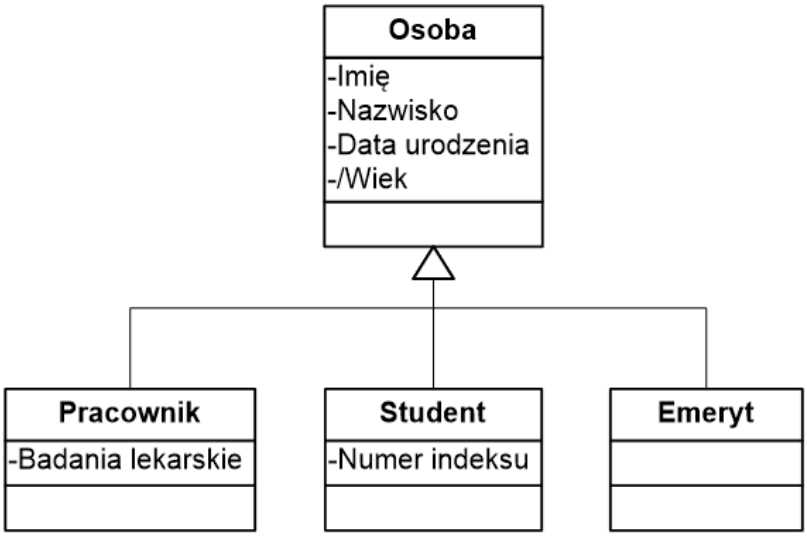
3 Wykorzystane technologie

W poprzednich rozdziałach omówiona została podstawowa tematyka strumieni oraz repozytoriów wraz z systemem kontroli wersji. Drugim problemem postawionym sobie przez autora było napisanie aplikacji w języku Java, której celem było zasymulowanie analizy nieskończonego strumienia danych, zawierających informacje odnośnie eventów wykonywanych na publicznych repozytoriach serwisu GitHub, który wykorzystuje system kontroli wersji Git oraz tworzeniem statystyk odnośnie wykorzystywania konkretnych mechanizmów na tym serwisie. Poniższe podrozdziały przedstawiały będą technologie, jakie zostały użyte podczas pisania tej aplikacji.

3.1 Java

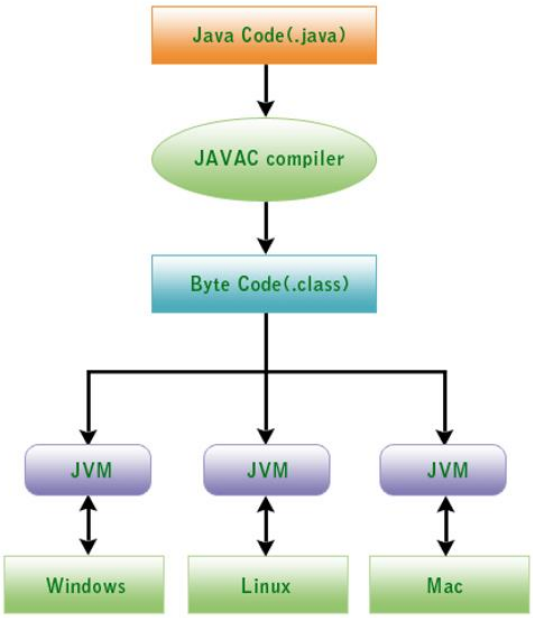
Java jest obiektowym językiem programowania, który został stworzony przez firmę Sum Microsystems 1995 roku pod kierunkiem Jamesa Goslinga. Inspirowany był językiem C++, z którego zaczerpnięta jest duża część składni i słów kluczowych oraz z języka Samlltalk poprzez elementy takie jak maszyna wirtualna czy zarządzanie pamięcią.[8] Java jest nastawiona na programowanie obiektowe. W związku z tym, wszystkie operacje wykonywane są na obiektach konkretnych klas. Wyjątkiem są jedynie typy prymitywne, np. int, double. Jednakże, typy te mają również swoje obiektowe odpowiedniki. Są to tak zwane ‘Wrapper Classes’. Stosuje się wtedy, gdy potrzebujemy typu obiektowego zamiast typu prymitywnego.

Jednym z podstawowych mechanizmów, jakie występują w tym języku, jest dziedziczenie. Polega on na tym, że klasa pochodna dziedziczy (zyskuje po klasie dziedziczącej) zmienne bądź metody. Rysunek 3.1 pokazuje przykładowy schemat dziedziczenia. Przedstawia on 4 klasy: Osoba, Pracownik, Student oraz Emeryt. Klasa osoba posiada takie pola jak: *Imię*, *Nazwisko*, *Data urodzenia* i *Wiek*. Klasa Pracownik posiada pole *Badania lekarskie,* klasa Student posiada pole *Numer Indeksu*, natomiast klasa Emeryt nie posiada żadnych pól. Jak wiadomo, Student, Pracownik czy Emeryt są osobami i powinny posiadać wszystkie pola, które posiada klasa Osoba. W związku z tym klasy te dziedziczą po klasie Osoba i posiadają jej wszystkie pola. Dzięki dziedziczeniu w prosty sposób można pokazać hierarchie klas. Mówi się, że w języku Java wszystko jest obiektem. Świadczy o tym fakt, że na samym szycie hierarchii wszystkich klas znajduje się klasa java.lang.Object, która jest swojego rodzaju korzeniem drzewa hierarchii klas.



Rysunek 3.1 Przykładowy schemat dziedziczenia
(źródło: https://edux.pjwstk.edu.pl/mat/205/lec/Wyklad-MAS-nr-06.html)

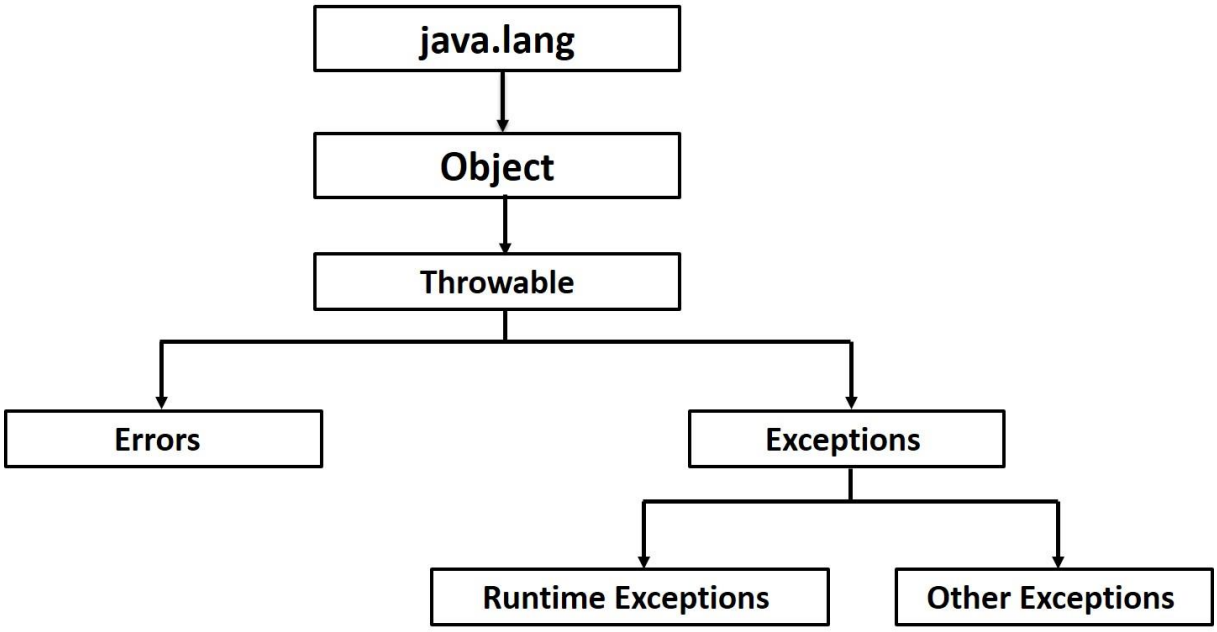
Java jest językiem niezależnym od platformy. Oznacza to, że programy napisane w Javie mogą będą działać niezależnie od systemu operacyjnego czy procesora. Jest to możliwe dzięki temu, że kod źródłowy programu wykonywany jest na wirtualnej maszynie Javy (ang. Java Virtual Machine), która następnie przetwarza go na kod dostosowany do danego systemu operacyjnego i procesora. Mechanizm ten przedstawiony jest na rysunku 3.2.



Rysunek 3.2 Schemat uruchomienia kodu źródłowego Javy
(źródło: http://studyur.blogspot.com/2016/06/java.html)

Java cechuje się dużym bezpieczeństwem oraz efektywnym zarządzaniem pamięcią operacyjną. W porównaniu z językiem C++, z którego zaczerpnięta jest spora część składni,

w Javie nie dojdzie nigdy do sytuacji nazywanej wyciekiem pamięci. Dzieje się tak za sprawą *Garbage Collector'a*. To program, który uruchamiany jest na wirtualnej maszynie Javy. Jego zadaniem jest zwolnienie pamięci zaalokowanej (zarezerwowanej) przez obiekty, które już nigdy więcej nie będą użyte w programie. Kwestia bezpieczeństwa związana jest również z wystąpieniem sytuacji wyjątkowych. Przykładem takiej może być odwołanie się do elementu, który znajduje się poza granicą tablicy lub próba wczytania do programu pliku, który nie istnieje. Wtedy program wyrzuci nam tak zwany wyjątek, czyli problem, który pojawił się podczas wykonywania programu. W związku z tym przewidywany przepływ programu jest zakłócony i w związku z tym program zostaje zatrzymany. Java dostarcza nam hierarchie klas wyjątków, przez co program jest w stanie określić, jaki wyjątek powinien wyrzucić (rys.3.3).



Rysunek 3.3 Uproszczony schemat umiejscowienia klasy wyjątków
(źródło: http://scriptinglogic.com/programming/java/exception-handling/)

Co jednak w sytuacji, jeżeli programista wie, iż w danej sytuacji może wystąpić wyjątek? Z pomocą przychodzi wtedy mechanizm try - catch. Polega on na tym, że programista oznacza dany fragment kodu, jako blok try, gdyż przypuszcza, iż na tym fragmencie może pojawić się wyjątek. Gdy taka sytuacja będzie miała miejsce, blok catch przechwyci i obsłuży wyjątek, a następnie zacznie wykonywać kod, który znajduje się tuż za blokiem catch. Implementacja tego bloku zależy od programisty. Może on przechwycić każdy wyjątek, jako *Exception* i w przypadku wystąpienia jakiegokolwiek, obsłużyć go tak samo lub też przechwytywać wybrane wyjątki i każdy z nich obsłużyć na własny sposób. Istnieje również możliwość napisania własnych wyjątków i ich obsługę.

Java dostarcza również narzędzie zwane JavaDoc. Generuje ono dokumentacje w formacie HTML na podstawie znaczników zamieszczonych w komentarzach kodu źródłowego. Komentarze takie muszą się znajdować pomiędzy znacznikami /** i */. Pierwsza linia takiego komentarza odpowiada za opis metody lub klasy, która znajduję się pod tymi znacznikami. W następnych linijkach komentarzu za pomocą opcjonalnych tagów możemy miedzy innymi wskazać parametry przyjmowane przez klasę czy metodę lub wartość zwracaną.[9]

3.2 Java 8

Java w wersji 8 została oficjalnie wydana 18 marca 2014 roku. Wersja ta jest olbrzymim krokiem naprzód w programowaniu w tym języku. Przybliża ona programistów w styl programowania funkcjonalnego, zachowując jasność i prostotę pisanego kodu. Programowanie funkcyjne (ang. functional programming) to sposób konstruowania programów, w którym funkcje są wartością podstawową i jest kładziony nacisk na łączenie ze sobą funkcji, a nie wykonywanie poleceń.[10]

3.2.1 Wyrażenia lambda

Styl programowania funkcyjnego nie jest jedyną nowością, jaką wprowadza ze sobą Java 8. Według autora jedną z największy funkcjonalności są wyrażenia lambda. Umożliwiają one tworzenie funkcji anonimowych, czyli takich, które nie posiadają nazwy, ale posiadają ciało. Jest wygodne podejście w przypadku, gdy chcemy użyć funkcji w programie tylko raz. Uproszczona składnia wyrażenia lambda wygląda następująco: „Parametry -> ciało funkcji”.

```
1 () -> System.out.println(this)
2 (String str) -> System.out.println(str)
3 str -> System.out.println(str)
4 (String s1, String s2) -> { return s2.length() - s1.length(); }
5 (s1, s2) -> s2.length() - s1.length()
```

Rysunek 3.4 Przykłady składni wyrażeń lambda w Java 8
(źródło: https://leanpub.com/whatsnewinjava8/read)

Istnieją kilka zasad składni wyrażenia lambda (rys. 3.4):

- Deklarowanie typów parametrów jest opcjonalne (np. linia 5)
- Jeżeli jest tylko jeden parametr, używanie nawiasu jest opcjonalne(np. linia 3)
- Słowo kluczowe ‘return’ jest opcjonalne, jeżeli tylko jedno wyrażenie ciała funkcji zwraca wartość (np. różnica w ciele funkcji między linią 4 i 5)

Jak można zauważyć, zapis takiej funkcji jest dużo krótszy i zajmuje mniej czasu. Bardzo często są to funkcje, które można w czytelny sposób zapisać w jednej linijce. Jest to dużo szybsze w porównaniu z zapisem, jaki był dostępny w Java 7. Przykładowy zapis takich samych funkcji, napisanych w Java 7 i Java 8, znajduje się na rysunku 3.5 i 3.6.

```
1 // Java 7
2 ActionListener al = new ActionListener() {
3     @Override
4     public void actionPerformed(ActionEvent e) {
5         System.out.println(e.getActionCommand());
6     }
7 };
8 // Java 8
9 ActionListener al8 = e -> System.out.println(e.getActionCommand());
```

Rysunek 3.5 Tworzenie klasy ActionListener
(źródło: https://leanpub.com/whatsnewinjava8/read)

```
1  // Java 7
2  Collections.sort(list, new Comparator<String>() {
3      @Override
4      public int compare(String s1, String s2) {
5          return s1.length() - s2.length();
6      }
7  });
8  //Java 8
9  Collections.sort(list, (s1, s2) -> s1.length() - s2.length());
10 // or
11 list.sort(Comparator.comparingInt(String::length));
```

Rysunek 3.6 Sortowanie listy zawierającej elementy typu String
(źródło: https://leanpub.com/whatsnewinjava8/read)

3.2.2 Strumienie

Strumień jest interfejsem dodanym w Java 8. Zlokalizowany jest w pakiecie java.util.steram. Reprezentuje sekwencje obiektów, przypominający nieco interfejs iteratora. Jednakże, w przeciwieństwie do iteratora, obsługuje on wykonania równoległe, czyli sekwencje, które wykonują się jednocześnie w tym samym czasie. Strumień posiada wbudowane mechanizmy mapowania i filtrowania. Istnieje wiele sposobów, aby utworzyć strumień w Java 8. Najczęściej wykorzystywanym jest, aby pochodził z interfejsu Collection. Posiada on dwie podstawowe metody tworzące strumień:

- stream() – zwraca sekwencyjny strumień z kolekcji, np. listy
- parallelStream() – zwraca możliwie równoległy strumień z kolekcji, np. listy

Najbardziej podstawową rzeczą, do jakiej można wykorzystać strumień, jest stworzenie pętli z wykorzystaniem metody '.forEach()'. Rysunek 3.7 przedstawia przykład wykorzystania strumienia i metody '.forEach()', aby wydrukować wszystkie pliki znajdujące się w bieżącym katalogu.

```
1 Files.list(Paths.get("."))
2     .forEach(System.out::println);
```

Rysunek 3.7 Wypisanie wszystkich plików znajdujących się w bieżącym katalogu
(źródło: https://leanpub.com/whatsnewinjava8/read)

Kolejnymi przykładami metod, które możemy wywołać na strumieniu, są '.map()', '.filter()' oraz '.reduce()'. Są to metody, możemy modyfikować nasz strumień, np. mapować dane czy filtrować tylko te informacje, które nas interesują. Na poniższym przykładzie (rys.3.8) przedstawione jest przykładowe użycie tych metod, w celu znalezienia obiektu klasy PlayerPoints, który posiada największą liczbę punktów.

```
1 PlayerPoints highestPlayer =
2   names.stream().map(name -> new PlayerPoints(name, getPoints(name)))
3    .reduce(new PlayerPoints("", 0.0),
4        (s1, s2) -> (s1.points > s2.points) ? s1 : s2);
```

Rysunek 3.8 Znalezienie obiektu klasy PlayerPoints z największą ilość punktów
(źródło: https://leanpub.com/whatsnewinjava8/read)

3.3 JavaFX

Java FX jest domyślną biblioteką definiowania graficznego interfejsu użytkownika w języku Java począwszy od wersji 8. Zastąpiła ona starsze biblioteki takie jak Swing i AWT. Charakteryzuje się przede wszystkim dobrą separacją warstw, wsparciem dla architektury MVC (Model View Controller) i udostępnia do tego użyteczne narzędzie o nazwie Scene Builder.[11]

W porównaniu z poprzednio używaną biblioteką Swing, JavaFX posiada nowe i ulepszone narzędzia interfejsu, np. różnego rodzaju wykresy. Dodany został też nowy język FXML, który podobnie jak HTML, służy tylko do określenia interfejsu aplikacji, utrzymując ją całkowicie oddzieloną od logiki aplikacji. Interfejs ten, czyli wszelkiego rodzaju przyciski, pola tekstowe, wykresy oraz wiele innych elementów wraz z ich rozmieszczeniem zapisywany jest przy użyciu tego języka w specjalnych plikach z rozszerzeniem '.fxml" (rys. 3.9). Jedyną rzeczą, którą musimy zrobić, aby wyświetlić w programie ten interfejs, jest użycie klasy *FXMLLoader*, która wyręcza nas w ręcznym ustawianiu elementów interfejsu w naszej aplikacji, i wyświetla przygotowany wygląd naszej aplikacji z pliku .fxml.

Kolejną ważną cechą JavaFX jest wykorzystanie wzorca MVC (Model View Controller). Jest to wzorzec służący do organizowania struktury aplikacji posiadających graficzne interfejsy użytkownika. Zakłada on podział aplikacji na 3 główne części:

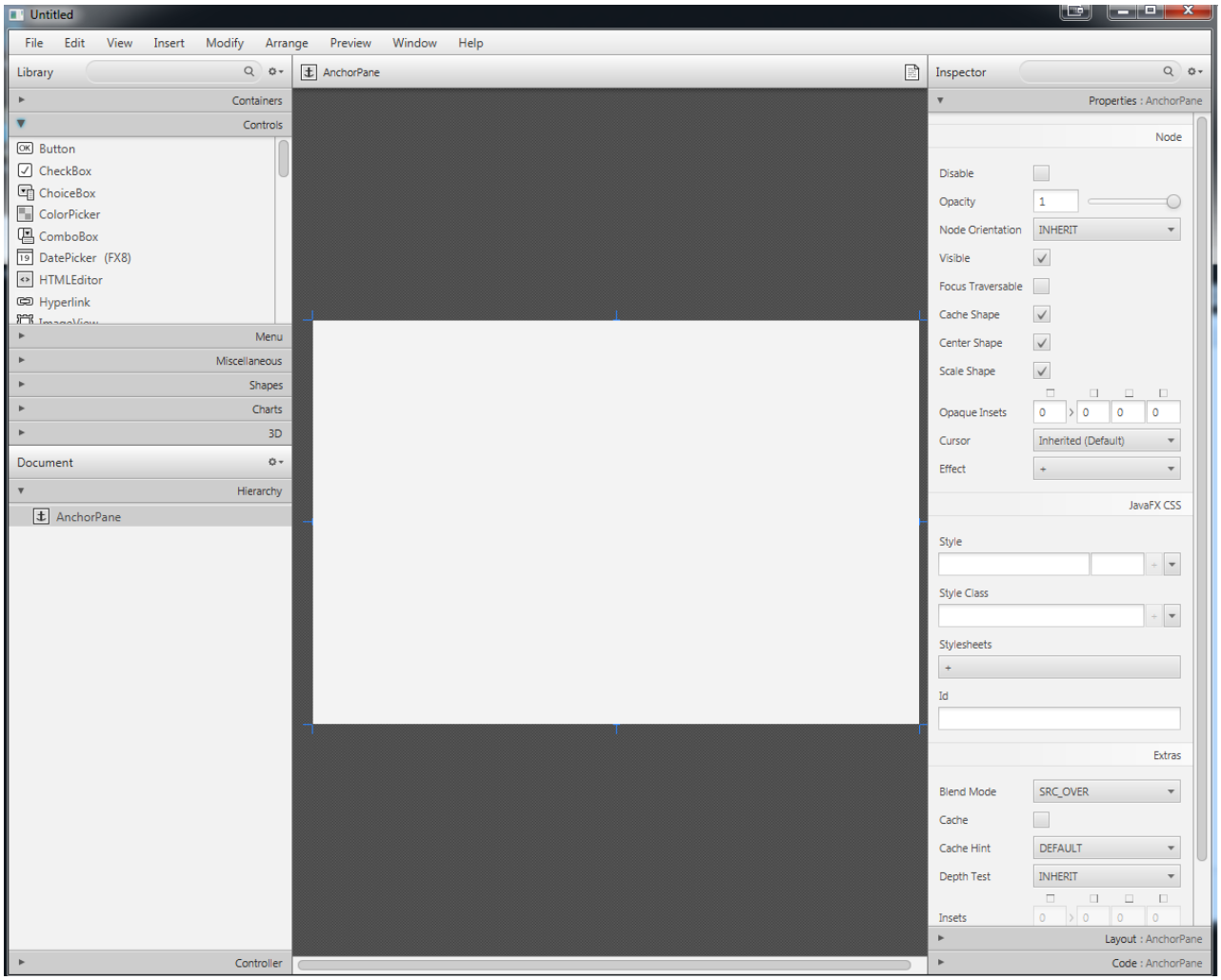
- **Model** – jest pewną reprezentacją problemu bądź logiki aplikacji.
- **Widok** – opisuje, jak wyświetlić pewną część modelu w ramach interfejsu użytkownika. Może składać się z podwidoków odpowiedzialnych za mniejsze części interfejsu.
- **Kontroler** – przyjmuje dane wejściowe od użytkownika i reaguje na jego poczynania, zarządzając aktualizacje modelu oraz odświeżenie widoków.

Wszystkie te trzy części są ze sobą połączone.[12] Dzięki tego typu rozwiązaniu udało się poprawić czytelność kody, oddzielając od siebie logikę i widok w aplikacji. Ułatwia to również dalsze rozbudowywanie aplikacji, poprzez dodawanie nowych komponentów, czy też dodatkowych funkcjonalności do już istniejących, nie zaburzając przy tym czytelności kodu.

```
<VBox id="vbox" prefHeight="400" prefWidth="800"
      xmlns:fx="http://javafx.com/fxml"
      fx:controller="dustin.examples.MenuController">
   <MenuBar fx:id="menuBar" onKeyPressed="#handleKeyInput">
      <menus>
         <Menu text="File">
            <items>
               <MenuItem text="New"/>
               <MenuItem text="Open"/>
               <MenuItem text="Save"/>
               <MenuItem text="Save As"/>
               <SeparatorMenuItem  />
               <MenuItem text="Exit"/>
            </items>
         </Menu>
         <Menu text="JavaFX 2.0 Examples">
            <items>
               <MenuItem text="Text Example"/>
               <MenuItem text="Objects Example"/>
               <MenuItem text="Animation Example"/>
            </items>
         </Menu>
         <Menu text="Help">
            <items>
               <MenuItem text="Search" disable="true"/>
               <MenuItem text="Online Manual" visible="false"/>
               <SeparatorMenuItem />
               <MenuItem text="About" onAction="#handleAboutAction"/>
            </items>
         </Menu>
      </menus>
   </MenuBar>
</VBox>
```

Rysunek 3.9 Przykładowy plik z rozszerzeniem .fxml *(źródło: opracowanie własne)*

Interfejs aplikacji zapisywany jest w plikach z rozszerzeniem .fxml. W związku z tym, pliki te muszą posiadać odpowiednią składnie, wytyczoną przez język FXML. Na rysunku 3.9 przedstawione jest użycie tylko kilku elementów, odpowiadających za wygląd aplikacji, a mimo tego użyta już została spora ilość słów kluczowych. W związku z tym, osoba projektująca wygląd (dzięki zastosowaniu nowego języka FXML, bazującego na języku XML, nie musi być ona wcale programistą Javy) musi znać sporą ilość własności każdego elementu interfejsu. Aby ułatwić pracę projektantom wyglądu aplikacji, zostało wydane oprogramowanie Scene Builder, które w łatwy i szybki sposób pozwala na tworzenie i generowanie plików FXML, wykorzystując metodę przeciągnij i upuść (rys. 3.10).



Rysunek 3.10 Interface główny aplikacji JavaFX Scene Builder 2.0 *(źródło: opracowanie własne)*

Wygląd aplikacji to nie tylko odpowiednie rozmieszczenie komponentów. JavaFX dostarcza również możliwość zmiany wyglądu poszczególnych komponentów z wykorzystaniem klasy CSS. Jest to sposób zapożyczony z projektowania stron internetowych, gdzie w plikach z rozszerzeniem .css możliwe było, przy wykorzystaniu odpowiedniej składni, zmodyfikować wygląd każdego komponentu, zmieniając kolor tła, dodając obramowanie itp. Poprzednia biblioteka Swing również dostarczała możliwość edycji przycisku czy pola tekstowego. Jednakże, aby tego dokonać, wiązało się to z napisaniem wielu linijek kodu, gdyż każdy komponent należało ustawiać osobno. Wyobraźmy sobie, co w sytuacji, gdy mieliśmy ustawione tło w stu przyciskach w różnych oknach naszej aplikacji, lecz stwierdziliśmy, że kolor ten powoduje, iż przycisk jest nieczytelny ma małych ekranach. W tym momencie, po wyborze odpowiedniego koloru, musimy we wszystkich tych przyciskach osobno zmieniać kolor tła. Jest to dosyć żmudne i długotrwałe. A co, jeżeli w tym przypadku nie jest to jedyna zmiana? Czas trwania zmiany jednego parametru przycisku wydłuża się w związku z ilością przycisków, w której musimy dokonać zmiany. Natomiast

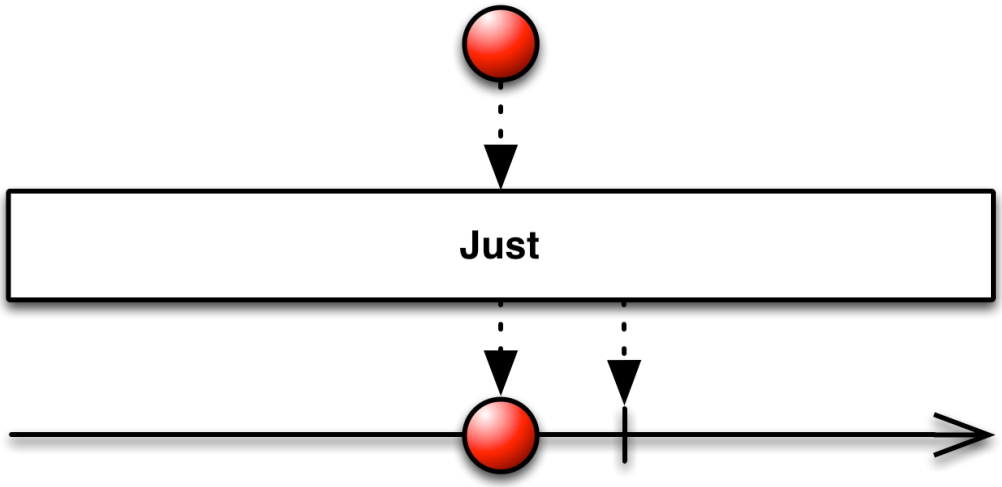
dzięki zastosowaniu pliku CSS, możemy stworzyć w nim klasę, która opisuje wygląd przycisku. Następnie do każdego przycisku przypisujemy identyfikator klasy. W tym wypadku każdy przycisk korzysta z wyglądu, jaki jest opisany w klasie CSS. Dzięki temu w przypadku modyfikacji jednego parametru wyglądu, jego wartość zmieniamy tylko w jednym miejscu, jakim jest klasa CSS, a nie w każdym przycisku osobno. Powoduje to wielokrotne przyspieszenie pracy nad wyglądem większych aplikacji.

3.4 RxJava

RxJava jest napisaną w języku Java biblioteką wykorzystywaną do tworzenia asynchronicznych programów opartych na zdarzaniach w postaci obserwowanych sekwencji danych. Jest ona dużym ułatwieniem w wypadku stosowania wielowątkowości. Rozszerza ona wzorzec obserwatora. W związku z tym posiada ona dwa główne typy: *Observable* i *Observer*. Głównym zadaniem *Observable* jest dostarczanie danych. Może on przetwarzać sekwencyjne dane, czyli w większości przypadków strumień danych, który przychodzi do programu, w strumień, który zawiera interesujecie dla danej sytuacji informacje między innymi za pomocą wbudowanej metody *.filter(),* czy też mapowania ich za pomocą metody *.map().* Celem drugiego głównego typu, a mianowicie *Observer* jest analiza każdego dostarczonego elementu jeden po drugim. Każda klasa, która jest typu *Observer* posiada 3 główne metody

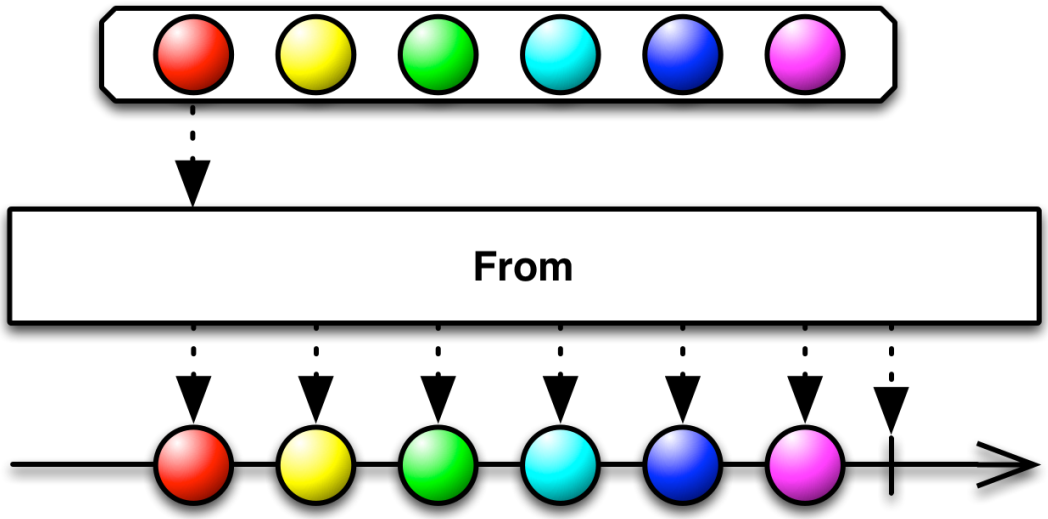
- onNext(): wywoływana jest w przypadku, gdy dostarczony został nowy element
- onError(): wywoływana jest w przypadku wystąpienia błędu dostarczania elementu ze strumienia
- onComplete(): wywoływana jest w przypadku, gdy poprawnie dostarczone zostały wszystkie elementy strumienia

Używając bibliotekę RxJava tworzy się obiekty Observable, emitujące zbiory danych, które możemy modyfikować, aby zawierały interesujące nas w danej chwili informacje. Jedną z możliwości jest utworzenie ich z istniejących już struktur danych. W tym celu wykorzystuje się wbudowane metody *just()* i *from().*



Rysunek 3.11 Graficzne przedstawienie działania metody *just()*
(źródło: http://reactivex.io/documentation/operators/just.html

Na rysunku 3.11 przedstawione jest graficzne działanie wbudowanej metody *just()*. Polega ona na utworzeniu obiektu Observable, który będzie emitował tylko jeden blok danych.



Rysunek 3.12 Graficzne przedstawienie działania metody *from()*
(źródło: http://reactivex.io/documentation/operators/from.html

Na rysunku 3.12 przedstawione jest graficzne działanie metody *from()*. Polega ona na utworzeniu obiektu Observable, który będzie emitował dostarczone do niej listy bądź tablice.

Obiekty powstałe przy użyciu tych metod będą synchronicznie wykonywały metodę *onNext()* na klasach, które sprawują zadanie Observer'a. W niej dostarczane będą emitowane dane, które można w dowolny sposób wykorzystać. Po dostarczeniu wszystkich danych, wywołana zostanie metoda *onComplete(),* sygnalizująca poprawne dostarczenie paczki danych.

4 Autorska aplikacja

4.1 Cel autorskiej aplikacji

Głównym celem aplikacji napisanej przez autora jest:

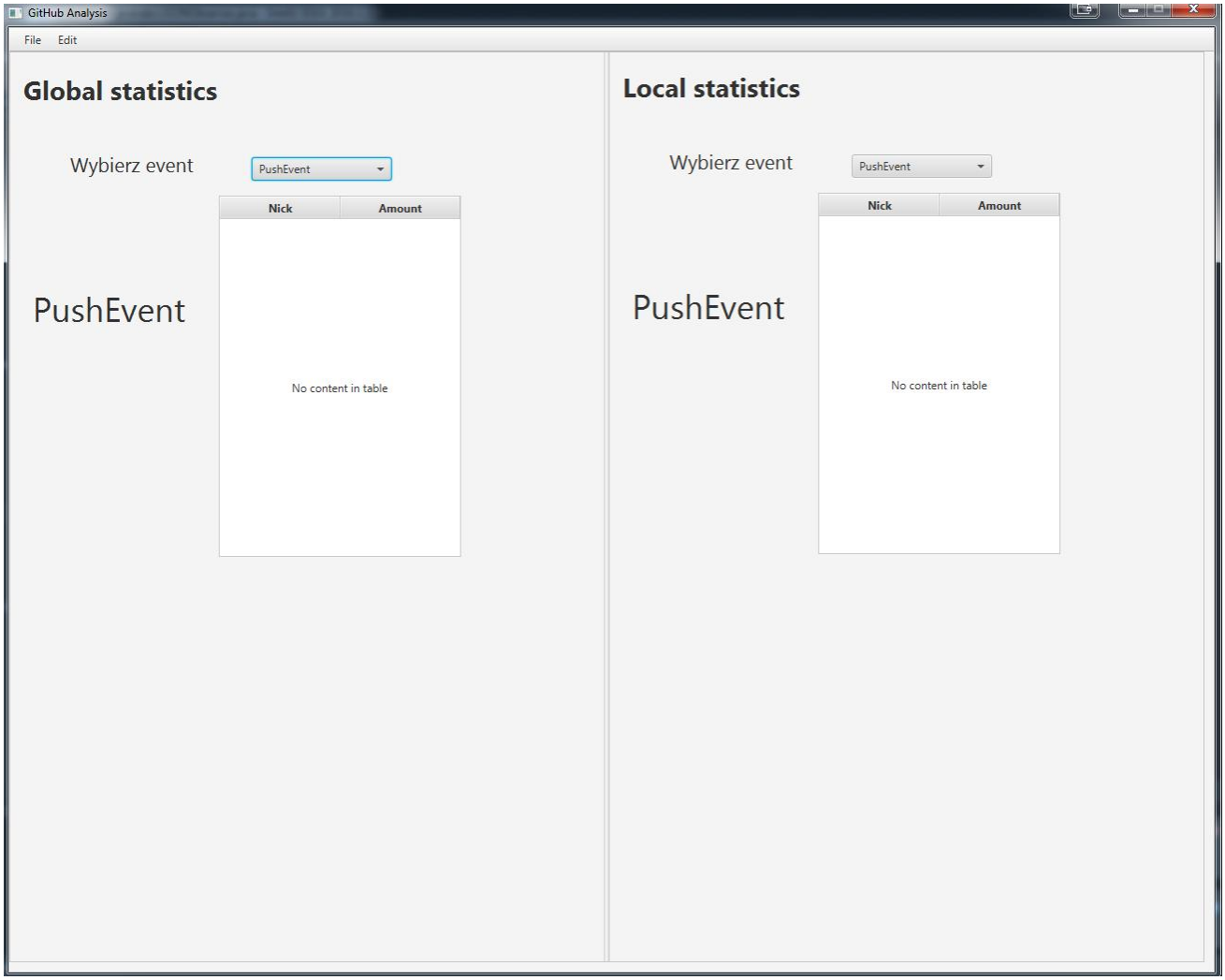
- utworzenie strumienia danych z archiwalnych plików, zawierających eventy wykonywane na publicznych repozytoriach serwisu GitHub
- napisanie logiki odczytania strumienia i przetworzenia go
- zrobienie statystyk zawierające informacje odnośnie eventów, które zostały odczytane podczas działania funkcjonalności przetwarzania strumienia
- przedstawienie graficzne otrzymanych statystyk

4.2 Opis funkcjonalności

Autorska aplikacja „GitHub analysis" jest przykładem prostego programu, który wyświetla zebrane informacje odnośnie eventów na repozytoriach GitHub'a w postaci statystyk. Jest ona typu open source oraz znajduje się na jednym z publicznych repozytoriów GitHub'a. Można wydzielić w niej dwie główne funkcjonalności: stworzenie i przetworzenie strumienia danych w celu utworzenia statystyk oraz graficzne wyświetlenie otrzymanych wyników. Strumień danych tworzony jest z archiwalnych paczek GitHub'a, które aplikacja pobiera ze strony www.githubarchive.org w postaci spakowanych plików z rozszerzeniem .gz. Następnie są one wypakowywane i przetwarzane na strumień danych. Aplikacja zbiera dostarczane dane ze strumienia w paczki wielkości ustalonego timebox'a (przedziału czasowego) oraz robi z nich statystyki. W międzyczasie specjalny mechanizm opóźnia dostarczanie danych do timebox'a, symulując prace strumienia w czasie rzeczywistym. „GitHub analysis" posiada dwie możliwości zmiany konfiguracji strumienia danych dostarczanego do aplikacji. Pierwsza z nich polega na ustawieniu daty początkowej, od której aplikacja zacznie tworzyć strumień danych do przetworzenia. Autor przyjął datę 1 stycznia 2012 roku jako najwcześniejszą, od której można zacząć się przetwarzanie strumienia danych do dnia wczorajszego (aplikacja a podstawie daty uruchomienia programu obliczy datę wczorajszą). Druga natomiast polega na ustawieniu przedziału czasowego (timebox'a) danych wykonywanych na repozytoriach, które będą wypychane ze strumienia do aplikacji, w celu utworzenia statystyk. Aplikacja wykorzystuje w tym celu obiekt *created_at*, który posiada

każdy event. W nim zapisana jest data i godzina wykonania konkretnego na repozytorium. Dzięki niemu można zebrać te eventy, które należą do konkretnego timebox'a.

Główny widok aplikacji podzielony jest na dwie części (rysunek 4.1). Po lewej stronie przygotowane są narzędzia, które wyświetlały będą statystyki zebrane od momentu ustalonego jako datę początkową przetwarzanego strumienia danych do momentu wystąpienia ostatniej paczki z danymi (timebox'a). Prawa strona jest lustrzaną kopią narzędzi, które znajdują się po lewej stronie. Jedyna różnica jest taka, że wyświetlała będzie informacje odnośnie ostatniej paczki z danymi, która została dostarczona do statystyk.



Rysunek 4.1 Widok główny po włączeniu aplikacji *(źródło: opracowanie własne)*

Statystyki są zbierane dla czterech eventów:

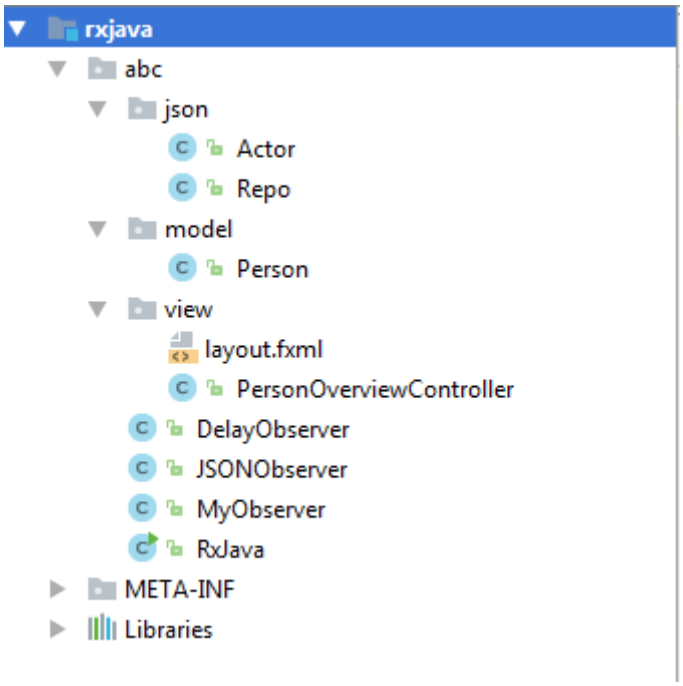
- PushEvent
- PullEvent
- WatchEvent
- CreateEvent

Statystyki wyświetlane są w dwojaki sposób. Pierwszy z nich polega na wyświetleniu posortowanej listy wybranych użytkowników, którzy wykonali najwięcej eventów danego typu wraz z ich ilością, drugi natomiast dla lepszego zobrazowania tych różnic w postaci wykresu kołowego. Dzieje się tak zarówno dla danych globalnych jak i danych lokalnych (z ostatniego timebox'a).

Zwróćmy teraz uwagę na strumień danych, który jest analizowany przez aplikacje. Tworzony jest on z plików JSON zawierających zarchiwizowane informacje odnośnie godzinnej pracy publicznych repozytoriów GitHub'a. Pliki pobierane są bezpośrednio z serwera strony www.githubarchive.org.

4.3 Implementacja aplikacji

Aplikacja została zaimplementowana w języku Java przy użyciu darmowej wersji środowiska IntelliJ. Główny wygląd aplikacji powstał w wyniku wykorzystania biblioteki JavaFX, a layout powstał dzięki aplikacji JavaFx Scene Builder. W celu utworzenia i przeanalizowania strumienia wykorzystana została biblioteka RxJava. Struktura katalogu przedstawiona została na rysunku 4.2.



Rysunek 4.2 Struktura aplikacji *(źródło: opracowanie własne)*

- RxJava – główna klasa programu, której celem jest wyświetlenie aplikacji oraz pobieranie plików z eventami z serwera
- MyObserver – klasa, która odpowiedzialna jest za odczytywanie eventów z timebox'a oraz mapowania ich na podstawie typów i robienia statystyk lokalnych i globalnych
- JSONObserver – klasa odpowiedzialna za tworzenie timeboxów
- DelayObserver – klasa odpowiedzialna za opóźnienie przetwarzania timebox'a, w celu zasymulowania pracy strumienia w czasie rzeczywistym
- PersonOverviewController – znajdująca w package'u *view* klasa głównego kontrolera aplikacji
- layout.fxml – znajdujący się w package'u *view* plik odpowiedzialny za wygląd aplikacji
- Person - znajdująca w package'u *model* klasa, która opisuje model użytkowników zapisywanych w tabelach
- Actor - znajdująca w package'u *json* klasa odpowiedzialna za tworzenie użytkowników do map, odczytanych ze strumienia
- Repo - znajdująca w package'u *json* klasa odpowiedzialna za tworzenie repozytoriów do map, odczytanych ze strumienia

Pierwszym etapem implementacji aplikacji było utworzenie strumienia danych. Wykorzystano w tym celu metodę *from()*, wbudowaną w bibliotekę RxJava, do której dostarczane są dane z pobranych i rozpakowanych plików przy pomocy między innymi klas GZIPInputStream oraz InputStream.(rys 4.3)

```
while (!start_date.equals(yesterday)) {
    while (hour < 24) {
        String url = "http://data.githubarchive.org/" + start_date + "-" + hour + ".json.gz";
        InputStream input = new URL(url).openStream();
        InputStreamReader isr = new InputStreamReader(new GZIPInputStream(input));
        BufferedReader in = new BufferedReader(isr);
        String content = null;
        while ((content = in.readLine()) != null)
            events.add(content);
        finish_date = start_date;
        Observable<String> stringObservable = Observable.from(events);
        stringObservable.subscribe(mo);
        hour++;
        events.clear();
    }
    hour = 0;
    c.add(Calendar.DATE, 1);  // number of days to add
    start_date = dateFormat.format(c.getTime());
}
```

Rysunek 4.3 Tworzenie strumienia *(źródło: opracowanie własne)*

Następnie zaimplementowana została klasa odpowiedzialna za obsługę dostarczanych ze strumienia danych. Posiada ona metodę onNext(), do której dostarczane są kolejne partie danych. Dzięki klasie pakietu JSON metoda ta odczytuje czas wykonania eventu oraz na jego podstawie dzieli otrzymywane informacje na timeboxy o ustalonym przedziale czasowym, zbierając je w listę i tworząc z niej kolejny strumień.(rys 4.4)

```
public void onNext(String line) {
    JSONObject obj = new JSONObject(line);
    String date = obj.getString("created_at");
    String[] date_splited = date.split("T");
    String[] time_splited = date_splited[1].split(":");
    Integer minute = Integer.valueOf(time_splited[1]);
    if (minute == temp_timestamp) {
        rxJava.finish_minute = temp_timestamp;
        rxJava.finish_hour = Integer.valueOf(time_splited[0]);
        rx.Observable<JSONObject> jsonObservable = rx.Observable.from(time_package);
        jsonObservable.subscribe(delayObserver);
        temp_timestamp += rxJava.timestamp;
        if (temp_timestamp > 59) temp_timestamp -= 60;
        time_package.clear();
    }
    time_package.add(obj);
}
```

Rysunek 4.4 Tworzenie timeboxów *(źródło: opracowanie własne)*

Kolejnym etapem było zaimplementowanie mechanizmu symulującego przetwarzanie danych w czasie rzeczywistym oraz analizę utworzonych timeboxów. Ponownie wykorzystując klasy pakietu JSON, odczytywane są typy eventów (rys 4.5), a następnie konkretne funkcje odczytują informacje o konkretnych typach eventów i zapisują je postaci map i setów.

```
public void onNext(JSONObject obj) {

    if (obj.getString("type").equals("CreateEvent")) {
        createEvent(obj);
    } else if (obj.getString("type").equals("PushEvent")) {
        pushEvent(obj);
    } else if (obj.getString("type").equals("WatchEvent")) {
        watchEvent(obj);
    } else if (obj.getString("type").equals("PullRequestEvent")) {
        pullEvent(obj);
    }
}
```

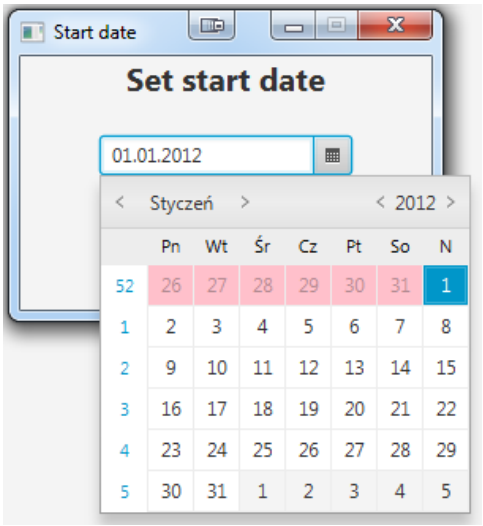
Rysunek 4.5 Rozpoznanie typów eventu *(źródło: opracowanie własne)*

Po przeanalizowaniu wszystkich danych z konkretnego timebox'a, następuję posortowanie zebranych informacji względem użytkowników, którzy wykonali najwięcej eventów w danych przedziale czasowym, a następnie przygotowanie bądź zaktualizowanie dla konkretnych rodzajów eventów map, zawierających pary użytkownik-licznik wykonanych eventów danego typu.

Ostatnim etapem było zaimplementowanie mechanizmu odpowiedzialnego na wyświetlenie map zebranych danych oraz konfiguracje strumienia. Layout aplikacji powstał przy użyciu Scene Buildera, wykorzystując metodę przeciągnij i upuść. W celu obsłużenia tego layoutu, zaimplementowana została klasa kontrolera. Posiada ona między innymi metody konfigurujące strumień: setStartDate() oraz setTimebox(). Najważniejszą metodą tej klasy jest initialize(), która odpowiada za aktualizacje informacji znajdujących się w tabelkach i na wykresach.

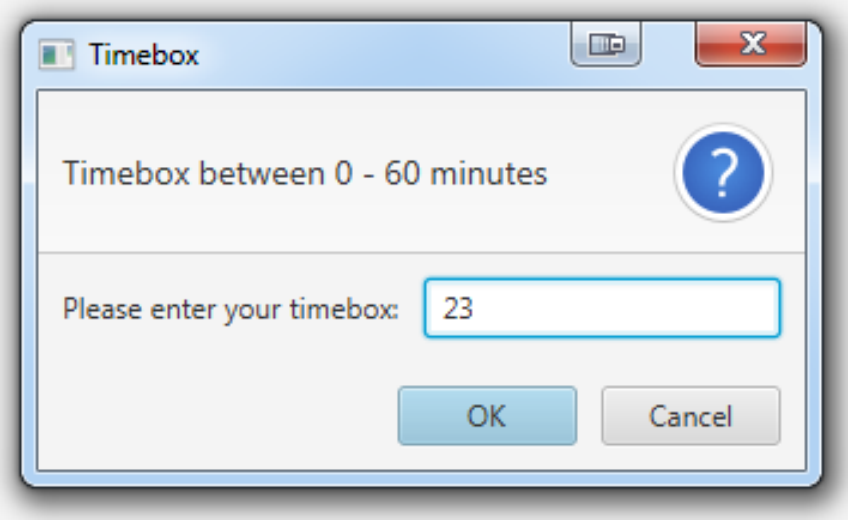
4.4 Przykładowe użycie aplikacji

W tym podrozdziale opisane jest przykładowe użycie aplikacji.



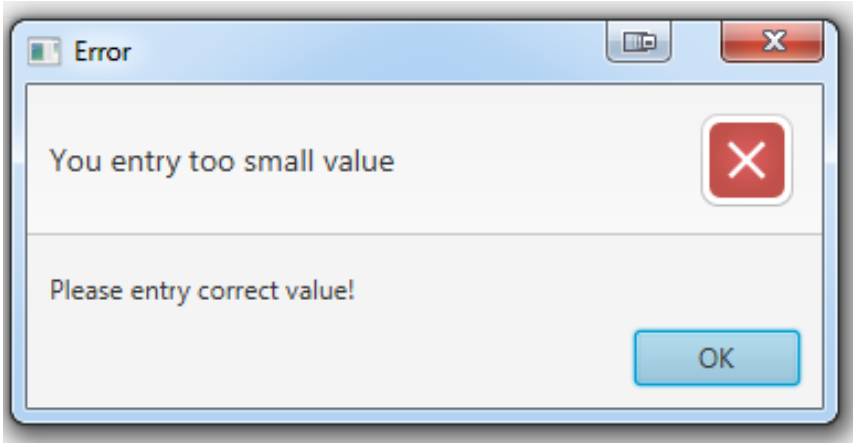
Rysunek 4.6 Ustawienie daty początkowej *(źródło: opracowanie własne)*

Po włączeniu programu wyświetla się nam wspominany już wcześniej widok główny (rys 4.1). Następnie użytkownik może dokonać modyfikacji daty rozpoczęcia, od której zacznie być tworzony, a następnie przetwarzany strumień danych. Opcja ta ukaże się po wybraniu z menu głównego *Edit -> Start date*. Pojawi się okienko, w którym będzie można dokonać modyfikacji daty (rys 4.6). W przypadku, gdy użytkownik nie dokona tej modyfikacji, data początkowa ustawiona będzie na 1 stycznia 2012 roku.



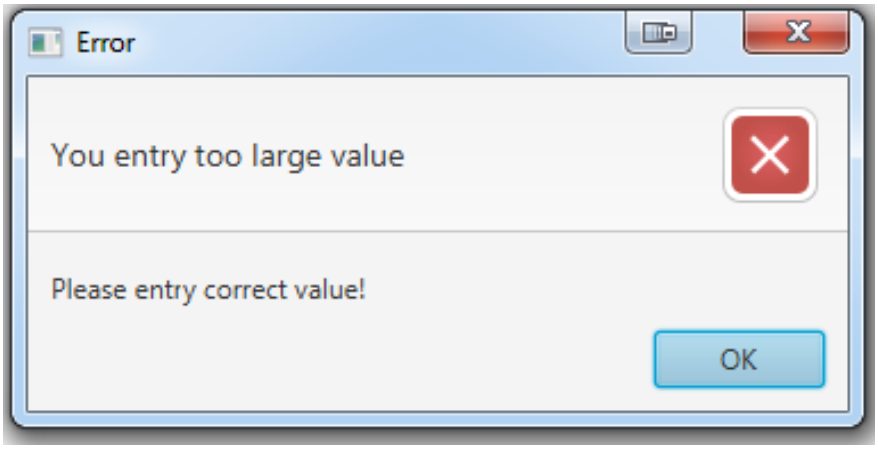
Rysunek 4.7 Ustawienie timebox'a *(źródło: opracowanie własne)*

Kolejnym etapem jest ustawienie timebox'a. Opcja ta ukaże się po wybraniu z menu głównego *Edit -> Set timebox.* Pojawi się okienko (inputbox), w którym będzie można dokonać modyfikacji wielkości timebox'a w minutach (rys. 4.7). Autor przyjął sobie przedział 0-60 minut jako ten, w którym można ustawić wielkość timebox'a. Inputbox jest błędoodporny. W przypadku ustawienia wartości mniejszej lub równej 0, użytkownik zostanie poinformowany o wprowadzonej zbyt małej liczbie (rys. 4.8) i zostanie poproszony o wprowadzenie nowej wartości.



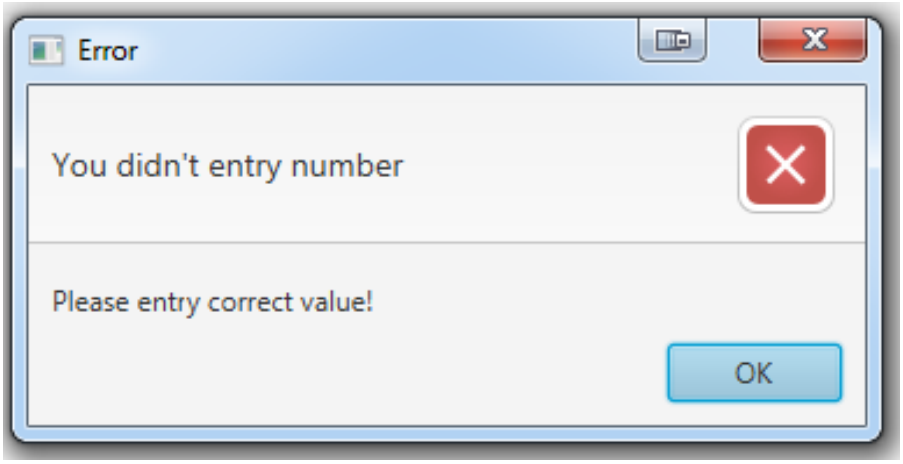
Rysunek 4.8 Wprowadzenie zbyt małej wartości timebox'a *(źródło: opracowanie własne)*

W przypadku ustawienia wartości większej lub równej 60, użytkownik zostanie poinformowany o wprowadzonej zbyt dużej liczbie (rys. 4.9) i zostanie poproszony o wprowadzenie nowej wartości.



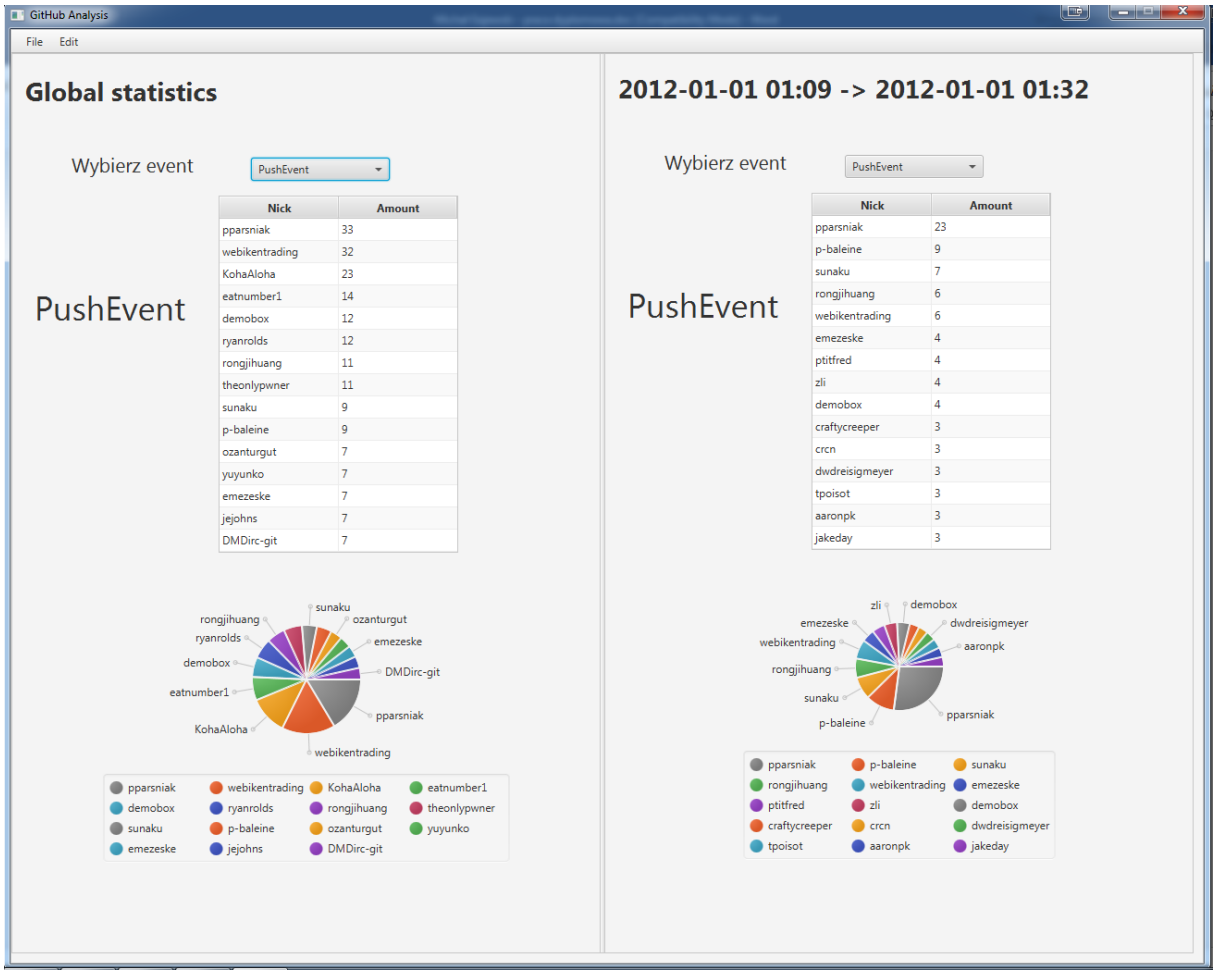
Rysunek 4.9 Wprowadzenie zbyt dużej wartości timebox'a *(źródło: opracowanie własne)*

W przypadku ustawienia wartości, która nie jest liczbą, użytkownik zostanie poinformowany o wprowadzeniu wartości, która nie jest liczbą (rys. 4.10) i zostanie poproszony o wprowadzenie nowej wartości.



Rysunek 4.10 Wprowadzona wartość nie jest liczbą *(źródło: opracowanie własne)*

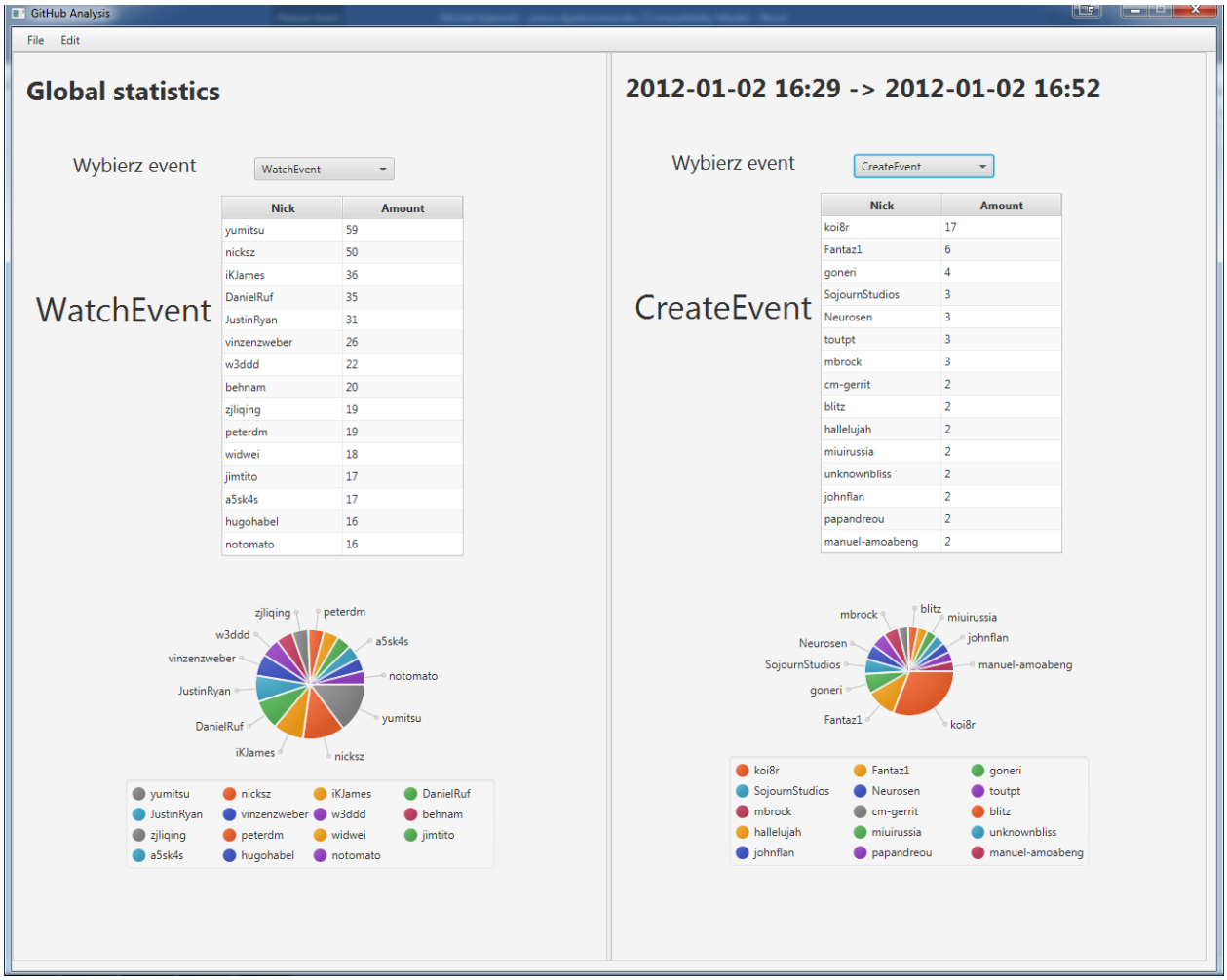
W przypadku, gdy użytkownik nie dokona tej modyfikacji, wartość ta ustawiona jest od momentu włączenia programu na 23 minuty. Gdy mamy już ustawione wszystkie parametry, kolejnym etapem jest uruchomienie dostarczania strumienia danych do aplikacji. Możemy tego dokonać, wybierając z menu *File -> Start Analysis*. W tym momencie, pobierany zostaje pierwszy pik z serwera, tworzony jest na jego podstawie strumień, który jest następnie analizowany i zbierane są statystyki. Nie ma już możliwości zamiany konfiguracji strumienia. Opcje *Set date* oraz *Set timebox* zostają wyłączone. Na rysunku 4.11 przedstawiony jest widok główny aplikacji, która przetworzyła już trzy 23 minutowe timeboxy od daty początkowej.



Rysunek 4.11 Widok główny aplikacji po zebraniu trzech 23 minutowych timeboxów

(źródło: opracowanie własne)

Widok ten jest aktualizowany w czasie rzeczywistym w momencie, gdy aplikacja przetworzy kolejny timebox informacji. Użytkownik podczas przetwarzania strumienia ma możliwość przełączania typu eventów. W przypadku wybrania innego, ekran aplikacji odświeży się, pokazując statystyki nowo wybranego eventu, które zostały już zebrane (rys. 4.12). Typy wyświetlanych eventów lokalnych i globalnych są od siebie niezależne i mogą być różne.



Rysunek 4.12 Wyświetlanie statystyk lokalnych i globalnych różnych typów

(źródło: opracowanie własne)

4.5 Możliwości rozwoju

Celem niniejszej pracy było wykazanie przydatności biblioteki RxJava do obsługi nieskończonych strumieni danych. Efekt w postaci aplikacji „GitHub Analysis" realizuje przede wszystkim te założenia i z tego względu nie posiada wielu, potencjalnie przydatnych mechanizmów i funkcjonalności. Autor dostrzega wiele możliwości dalszego rozwoju systemu, np.:

- Zwiększenie timeboxów (ponad 59 minut)
- Dodanie nowych eventów do analizy
- Dodanie liczników do odczytanych eventów (zarówno do statystyk lokalnych jak i globalnych)

- Dodanie możliwości wprowadzenia godziny rozpoczęcia tworzonego strumienia (aktualnie możliwe jest wprowadzenie początkowej daty)
- Dodanie możliwości zmiany typów wykresów
- Dodanie możliwości zatrzymania strumienia
- Dodanie możliwości zapisywania statystyk do plików
- Dodanie nowej szaty kolorystycznej

Podsumowanie

Opisana w poprzednich rozdziałach praca dyplomowa dotyczyła tematu wykorzystania biblioteki RxJava do analizy „nieskończonego” strumienia danych z serwisu GitHub. Autorowi udało się zrealizować postawione sobie dwa główne cele. W pierwszych dwóch rozdziałach zawarł podstawową wiedzę teoretyczną związaną z tematem pracy. W jednym z nich przedstawił podstawowe informacje odnośnie strumieni danych, opisując definicję oraz ideę strumieniowania danych wraz z przykładami modyfikacji nieskończonego strumienia danych za pomocą filtrów czy funkcji aproksymacyjnych. W drugim natomiast przedstawił podstawowe informacje odnośnie repozytoriów, systemu kontroli wersji Git, serwisu GitHub przechowującego repozytoria Gita oraz serwisu GitHub Archive, na którym znajdują się zarchiwizowane paczki logów, zawierające informacje odnośnie eventów wykonywanych na publicznych repozytoriach GitHub’a. Drugim celem postawionym sobie przez autora, który został zrealizowany było napisanie aplikacji w języku Java, której głównym zadaniem było stworzenie „nieskończonego” strumienia danych z archiwalnych paczek serwisu GitHub oraz stworzenie statystyk na podstawie czterech wybranych przez autora eventów. W tym celu zostały wykorzystane między innymi biblioteka RxJava do ułatwienia procesu analizy strumienia danych i jego przetworzenia oraz biblioteki JavaFX do graficznego wyświetlenia utworzonych statystyk.

Napisana przez autora aplikacja posiada głównie podstawowe funkcjonalności. Autor planuje dalszy rozwój aplikacji. Planowane jest między innymi dodanie liczników odnośnie odczytanych eventów przez aplikacje, dodanie nowych eventów do analizy czy też zapisywanie statystyk do zewnętrznych plików. Miejmy nadzieje, że znajdzie ona zastosowanie wśród osób związanych z serwisem GitHub, które odpowiedzialne są za statystyki związane z wykonywaniem eventów na serwisie GitHub.

Bibliografia

[1] https://en.wikipedia.org/wiki/Data_stream, stan na 07.02.2017

[2] http://searchcloudcomputing.techtarget.com/definition/big-data-Big-Data, stan na 07.02.2017

[3] http://e-swoi.pl/media/userfiles/jarek.zok/repo-doc.pdf , stan na 02.02.2017

[4] https://git-scm.com/book/pl/v1/ , stan na 02.02.2017

[5] http://blog.undicom.pl/systemy-kontroli-wersji-ktore-wybrac/, stan na 01,02.2017

[6] https://en.wikipedia.org/wiki/GitHub, stan na 02.02.2017

[7] http://www.json.org/, stan na 02.02.2017

[8] https://4programmers.net/Java, stan na 03.02.2017

[9] http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html, stan na 03.02.2017

[10] http://searchsoftwarequality.techtarget.com/definition/functional-programming, stan na 03.02.2017

[11] http://javastart.pl/b/java/dlaczego-javafx-jest-lepsza-od-swinga/, stan na 03.02.2017

[12] http://www.artima.com/articles/dci_vision.html, stan na 03.02.2017

[13] Eckelo Bruce, *Thinking in Java*, Wydanie IV, Wydawnictwo Helion, 2006

[14] Gajda Włodzimierz, *Git. Rozproszony system kontroli wersji,* Wydawnictwo Helion, 2013

[15] Horstmann Cay S., *Java 8. Przewodnik doświadczonego programisty*, Wydawnictwo Helion, 2015

[15] https://github.com/igrigorik/githubarchive.org, stan na 02.02.2017

[17] https://leanpub.com/whatsnewinjava8/read, stan na 02.02.2017

[18] http://www.javafxtutorials.com/whatisjavafx/, stan na 03.02.2017

[19] https://github.com/ReactiveX/RxJava/wiki, stan na 03.02.2017

[20] https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101, stan na 08.02.2017

[21] http://reactivex.io/documentation, stan na 12.02.2017

Spis rysunków