Podejście funkcyjne do obsługi kolekcji w języku C++

Krzysztof Osiecki

18 kwietnia 2017

Spis treści

Wstęp

Programowanie samo w sobie jest procesem mającym na celu stworzenie działającego oprogramowania. W celu wykonania całego tego procesu podejmuje się wiele działań. Etapami procesu programowania są: projektowanie, testowanie i utrzymanie kodu, a także przede wszystkim to co może być przez niektórych utożsamiane z programowaniem, czyli pisanie kodu programów. Obecnie dąży się do strukturalizowania każdego z tych etapów. Powstały różne metodyki oraz modele, mające ułatwić zarządzanie wytwarzaniem oprogramowania. Są one różne w zależności od skali projektów, terminów czy stopnia wydajności bądź zdolności do podejmowania autonomicznych decyzji przez programistów. U podstaw każdej z metodyk leży ostatecznie kod programu. Kod, który może być napisany na wiele sposobów w wielu językach programowania. Podobnie jak w przypadku metodyk, języki różnią się między sobą stopniem dopasowania do konkretnych rozwiązań. W pierwszej części niniejszej pracy przedstawiona została szersza kategoria podejścia do tworzenia programów, wykraczająca poza określenie jej w obrębie jednego języka. W drugiej zaś części zaprezentowano próbę szerszej realizacji tego podejścia w języku C++.

Rozdział 1

Omówienie funkcyjnego paradygmatu programowania

1.1 Czym jest paradygmat programowania

Wspomniana we stępie kategoria podejścia do tworzenia programów określana jest mianem paradygmatu programowania. Paradygmat nie jest określany przez język programowania, lecz przez sposób w jaki programista opisuje problem, który rozwiązuje. Paradygmaty podzielić można na dwie główne grupy. Paradygmat imperatywny oraz deklaratywny. W paradygmacie imperatywnym, motywem przewodnim programu jest instrukcja. Instrukcja stanowi rozkaz dla procesora, zaś program składa się z wykonania odpowiedniej ilości takich rozkazów. Jest to chyba najbardziej rozpowszechniona, kategoria programowania. Powodem takiego stanu rzeczy jest najprawdopodobniej fakt, iż języki imperatywne są tymi, od których w większości przypadków zaczyna się naukę programowania. Wydawać by się jednak mogło, że bliższy ludzkiemu sposobowi rozwiązywania problemów będzie podejście deklaratywne. W paradygmacie deklaratywnym program stanowi opis rozwiązania. Programista nie odpowiada, za instruowanie procesora jak ma wykonywać poszczególne czynności. Opisuje tylko spodziewany rezultat, a właściwie cechy tego rezultatu. Zadaniem komputera, a w zasadzie kompilatora, bądź interpretera, konkretnego języka jest odpowiednie dopasowanie działań w celu uzyskania odpowiednich rezultatów. W praktyce rzadko spotyka się języki w całości oparte tylko na jednym paradygmacie. Wynika to z ograniczeń jakie muszą być realizowane w celu pełnego spełnienia założeń, każdego z nich. Najczęściej okazuje się, że idealnym rozwiązaniem jest połączenie najlepszych cech każdego z nich. W praktyce określenie paradygmat często wymienia się na programowanie. Oznacza to, że pojęcia programowanie deklaratywne i paradygmat deklaratywny są tożsame, dlatego będą w niniejszej pracy stosowane wymiennie.

1.2 Cechy paradygmatu funkcyjnego

Programowanie funkcyjne jest odmianą programowania deklaratywnego. Główną cechą tego paradygmatu jest skupienie uwagi na funkcjach i obliczaniu

ich wartości. W programowaniu funkcyjnym, funkcja zachowuje się tak samo jak funkcja matematyczna. To znaczy, że dla parametrów dla których jest poprawnie zdefiniowana, będzie zawsze zwracać tą samą wartość. Wykonanie programu polega zatem na obliczeniu wartości składających się na niego funkcji.

1.3 Efekty uboczne, w programowaniu funkcyjnym

Ważną cechą funkcji w sensie matematycznym jest jej bez stanowość. Każda funkcja matematyczna posiada swoją definicję, która określa jej wynik w zależności od zadanych argumentów. Dla określonego zbioru argumentów funkcja zawsze zwróci ustaloną warość. Niezależnie od czynników zewnętrznych ani ilości jej wywołań. Programowanie funkcyjne jako bazujące silnie na matematycznych podstawach, stara się emulować takie zachowanie funkcji. W programowaniu imperatywnym sytuacja wygląda inaczej. Można powiedzieć, że większą część programowania imperatywnego stanowi coś co określa się mianem `efektów ubocznych`. Każde zachowanie funkcji (tutaj programistycznej, nie matematycznej) nie będące wyliczeniem i zwróceniem wartości nazywane jest efektem ubocznym. W przypadku imperatywnego programowania będą to na przykład: operacje obsługi wejścia/wyjścia, utworzenie zmiennej, czy obiektu w pamięci, modyfikacja struktury danych. Z funkcyjnego punktu widzenia, tego typu operacje są `brudne` i nie pozwalają na zachowanie czystości języka funkcyjnego. Trudno jednak nie zauważyć jak wielkie znaczenie w świecie obecnego tworzenia oprogramowania odgrywa możliwość interakcji użytkownika z systemem. A skoro operacje wejścia/wyjścia nie są dozwolone w języku funkcyjnym, nie trudno się domyślić, że tego typu język raczej nie ma szans podbić komercyjnego rynku.

1.4 Dlaczego paradygmat funkcyjny?

Pomimo tej wady, paradygmat funkcyjny posiada cechy, które niewątpliwie stanowią jego ogromną zaletę. Sprawiły one, że nie został on zapomniany, a raczej powraca co jakiś czas w różnych formach i znajduje drogę do najpopularniejszych współcześnie języków. Przede wszystkim jako podzbiór paradygmatu deklaratywnego, programowanie funkcyjne nie wymaga od nas, konkretnego opisu działania, lecz opisu oczekiwanego przez nas wyniku. Dodatkowo potraktowanie funkcji jako `obywatela pierwszej kategorii` sprawia, że normalnym staje się możliwość przekazania jednej funkcji jako parametru drugiej i wywołania jej kiedy będzie potrzebna, z argumentami, któe w momencie jej tworzenia mogły być nie dostępne. Daje to w ręce programisty naprawde potężne narzędzie, w przypadku aplikacji sterowanych zdarzeniowo. Na koniec zostawiam cechę, która sprawia, że w ogóle następują próby wplatania paradygmatu funkcyjnego do imperatywnych języków (przecież, imperatywnie i tak można te operacje wykonać). Cechą tą jest składnia, która w językach funkcyjnych jest niezwykle atrakcyjna dla programisty. Korzystając z funkcyjnych interfejsów, nie spotkamy kodu w którym przez piętnaście kolejnych linii definiujemy zmienne, po to żeby w szesnastej utworzyć z ich pomocą obiekt, a w siedemnastej wykonać na tym obiekcie jakąś metodę. Operacja sortowania w programowaniu funkcyjnym wymaga od

nas kolekcji do posortowania i funkcji definiującej liniowy porządek na obiektach tej kolekcji. Jest to oczywiście tylko i wyłącznie kwestia odpowiednio wysokopoziomowego API. Jednak utworzenie go w sposób przyjazny dla użytkownika jest dużo łatwiejsze w językach, które paradygmat funkcyjny starają się adaptować, niż w tych, które zdecydowanie go odrzucają.

1.5 Domknięcia i wyrażenia lambda jako sposób realizacji paradygmatu

Jednym ze sposobów realizacji funkcyjnych aspektów w językach imperatywnych, są wyrażenia lambda. Wyrażenia te są w rzeczywistości, anonimowymi funkcjami, które można łatwo definiować i przekazywać jako parametry. Pojęcie lambd pozwoliło traktować je jako obiekty w językach w których funkcje, nie są domyślnie traktowane jako obiekty pierwszej kategorii. Najważniejszą ich cechą, która zapewnia ich użyteczność jest łatwość ich definiowania. Lambdę można zdefiniować w miejscu w którym jest potrzebna i przekazać od razu do wykonania innej funkcji, bądź jeśli istnieje potrzeba ponownego użycia, przypisać do zmiennej i wykorzystywać wielokrotnie. Od strony formalnej, lambdy stanowią konkretną implementację koncepcji domknięcia.

Definicja 1. *Domknięcie – w metodach realizacji języków programowania jest to obiekt wiążący funkcję lub referencję do funkcji oraz środowisko mające wpływ na tę funkcję w momencie jej definiowania.*

Domknięcie stanowi więc pojęcie szersze, którym można określić wszystkie implementacje pozwalające na użycie funkcji w kontekście obiektu, w paradygmacie programowania obiektowego.

1.6 Kolekcje jako strumienie danych

Wszystkie kolekcje, można przy przyjęciu odpowiedniego poziomu abstrakcji patrzeć jak na strumień danych. Zarówno listy jak i drzewa, poza faktem, zapewnienia struktury, dla danych w której łatwo można nimi zarządzać, stanowią często źródło, które należy zużyć. Oczywiście kolejność iteracji po poszczególnych elementach może być różna, różne może być także podejście do konieczności przejścia po wszystkich elementach. Kluczową jednak cechą jest chęć wykonania pewnej operacji na całym, bądź przefiltrowanym w jakiś sposób zbiorze danych. Podejście takie i korzyści, które z niego płynął, są główną przyczyną powstania niniejszej pracy. Tego typu rozwiązania, zaimplementowano już w wielu językach programowania. W następnych podroździałach przestawiono, sposoby implementacji tych rozwiązań.

1.7 Realizacja paradygmatu w języku JavaScript

Javascript jest językiem, wieloparadygmatowym. Jednak można powiedzieć, że najistotniejsze ze składających się na sukces tego języka cech to jego `zdarzeniowość` (z angielskiego `event-drivent`), oraz funkcyjność. O sile

javacriptu stanowi właśnie łatwość definiowania zachowań obiektów poprzez określenie funkcji mającej wykonać się w przypadku wystąpienia konkretnego zdarzenia. Obiekty opakowujące funkcje, czyli `domknięcia`, dokonują `przechwycenia` (z angielskiego `capture`), zmiennych przez referencję. Oznacza to, że mają do nich pełen dostęp i możliwość ich modyfikacji. Koncepcja ta choć niezwykle praktyczna, może jednak powodować niekiedy problemy, kiedy jakaś nieznana deweloperowi funkcja zaczyna modyfikować jego obiekty. Specyfika wykorzystania języka powoduje czasami nieporozumienia w kontekście referencji do aktualnego obiektu, `this`. Każda z funkcji definiowanych w ten sposób definiuje swój własny obiekt `this`. W przypadku konstruktorów będzie to nowo utworzony obiekt. W trybie `strict`; mniej łaskawym dla niejasnych instrukcji, blokującym niektóre konstrukcje, traktującym część `cichych` błędów, jako błędy diałnia); `undefined`, czyli brak obiektu. Zaś w przypadku wykonania funkcji, jako metody konkretnego obiektu, będzie to tak zwany `context object`. Tego typu zachowania, często prowadzą do błędów w rozumieniu i poprawnym pisaniu kodu. Deweloper spodziewa się, `this`, będze obiektem okalającym, co nie będzie prawdą. Tego typu zachowanie, jest jednak omijane poprzez przypisanie potrzebnej wartości `this` do obiektu, który będzie `przechwycony` przez domknięcie.

Na listingu 1.1, zobrazowany jest przykład definiowania funkcji w języku javascript. Jak widać definicja funkcji rozpoczyna się od słowa kluczowego `function`, po którym następuje opcjonalna nazwa (w przypadku funkcji wewnętrznej pominięta). Dalej w nawiasach okrągłych znajduje się lista parametrów funkcji, zaś na końcu ograniczony przez nawiasy klamrowe blok działania funkcji. Jak pokazuje przykład, możliwa jest praktcznie całkowita dowolność w traktowaniu funkcji jako obiektu. W linii siódmej, do obiektu `domknięcie` przypisany zostaje wynik funkcji `licznik`, czyli w tym przypadku funkcja anonimowa zdefiniowana, której definicja rozpoczyna się w linii trzeciej. W tym też miejscu utworzona zostaje zmienna lokalna `liczba` funkcji `licznik`, a jej referencja przekazana jest do anonimowej funkcji, która przypisana jest do zmiennej `domknięcie`. Dlatego też kolejne wywołania tej funkcji, powodują zwiększanie licznika i wypisywanie w wyniku liczby o jeden większej od poprzedniej. W linii dziesiątej zastosowano inne rozwiązanie. Do zmiennej `obiektLicznika` przypisano funkcję licznik, zamiast wyniku jej wykonania. Dwukrotne wywołanie funkcji określających ten obiekt powoduje kolejno, wywołanie funkcji `licznik`, czyli zwrócenie funkcji anonimowej, oraz jej wywołanie. W tym przypadku `licznik` wywołuje się za każdym razem na nowo, co za tym idzie, zmienna `liczba`, także tworzona jest na nowo, czyli wynikiem kolejnych instrukcji będzie, za każdym razem 1.

Listing 1.1: Funkcyjność w języku javascript

```
1  function licznik() {
2   var liczba = 0;
3   return function() {
4    return ++liczba;
5   };
6  }
7  var domkniecie = licznik();
8  console.log(domkniecie()); //wypisze 1 w konsoli, o ile srodowisko
       udostepnia obiekt console
9  console.log(domkniecie()); //wypisze 2 w konsoli, o ile srodowisko
       udostepnia obiekt console
```

```
10 var obiektLicznika = licznik;
11 console.log(obiektLicznika()()); //wypisze 1 w konsoli, o ile
       srodowisko udostepnia obiekt console
12 console.log(obiektLicznika()()); //wypisze 1 w konsoli, o ile
       srodowisko udostepnia obiekt console
```

JavaScript stanowi implementacje standardu skryptowych języków ECMA-Script. Standard ten dopiero w swojej szóstej wersji, zamkniętej w czerwcu 2015 roku wprowadził konstrukcje, które można określić jako odpowiedniki wyrażeń lambda. Użycie tego standardu może jednak bywać problematyczne, ze względu na niespójną implementację pomiędzy przeglądarkami, które stanowią główne środowisko uruchomieniowe JavaScriptu. Funkcje strzałkowe (`arrow functions`), są JavaScriptową wersją wyrażeń lambda. Ich składnia przedstawiona jest na listingu 1.2.

Listing 1.2: Funkcje strzałkowe [2]

```
1  (param1, param2, ..., paramN) => { statements }
2  (param1, param2, ..., paramN) => expression
3  // rownowazne z: (param1, param2, ..., paramN) => { return
       expression; }
4
5  // Nawiasy sa opcjonalne w przypadku pojedynczego parametru
6  (singleParam) => { statements }
7  singleParam => { statements }
8
9  // Przy braku parametrow nawiasy sa wymagane
10 () => { statements }
11 () => expression // rownowazne z: () => { return expression; }
```

Tego typu funkcje, rozwiązują problem niejasnego zachowania, referencji `this`. Ponieważ nie definiują własnego kontekstu, w ich wnętrzu odwołanie `this`, zawsze określa obiekt, okalający funkcję w miejscu tworzenia. Zapewniają one także, bardziej skompresowaną konstrukcję. Są one doskonałym rozwiązaniem do funkcyjnych operacji na kolekcjach, takich jak filtrowanie czy mapowanie. Jedyną ich wadę stanowi fakt, niespójności implementacji JavaScriptu, przez różne przeglądarki. Dla przykładu Internet Explorer, nie wspiera tej konstrukcji. Do tego ze względu na różny stopień pokrycia specyfikacji ES6 przez przeglądarki, używanie tego standardu jest często niemile widziane, w produkcyjnych rozwiązaniach. Biorąc pod uwagę silny nacisk na funkcyjność języka JavaScript, istotne wydaje się aby, funkcje strzałkowe upowszechniły się jak najszybciej, jako, że znacznie zwiększają przejrzystość i łatwość tworzenia kodu.

Od strony zagadnienia najistotniejszego w kontekście niniejszej pracy, językowi JavaScript w zasadzie niczego nie brakuje. Podstawowa klasa stanowiąca kolekcję `Array`, oferuje metody obsługi przyjmujące obiekt funkcji odpowiednio aplikowany do elementów kolekcji. Dostępne są zatem między innymi operacje: `filter`, `map`, `forEach`, `reduce`, `find`, `every`, `some`. Operacje te, występują w zasadzie w każdym API funkcyjnym o obsługi kolekcji. Niejednokrotnie w innej niż ta obecna w języku JavaScript formie, bądź pod innymi nazwami, jednak oferowana przez nie funkcjonalność jest w jakiś sposób dostępna. Z tego właśnie względu ważna jest wiedza na temat tego co konkretnie oferują te operacje.

1.8 Opis podstawowych operacji funkcyjnych

Operacja `filter` polega na ograniczeniu kolekcji poprzez wyeliminowanie elementów, które nie spełniają kryterium określonego przez funkcję filtrującą. Funkcje filtrujące, czyli takie, które przyjmują jako argument, element kolekcji zaś w wyniku zwracają wartość logiczną, określającą spełnialność kryterium, zwane są predykatami. Operacja `filter` jest zatem operacją przyjmującą predykat i zwracającą kolekcję, bądź obiekt pośredni w przypadku strumieni, zawierającą tylko elementy spełniające ten predykat. Przykładem takiego działania może być na przykład wyfiltrowanie tylko kobiet z listy ludzi.

`Map` to operacja polegająca na przekształceniu, kolekcji elementów typu `A` w typ `B`. Do każdego elementu zbioru wejściowego, aplikowana jest funkcja, mająca zwrócić obiekt do wyjściowej kolekcji. Tak jak w przypadku operacji `filter`, wynikiem jest konkretna lista, bądź obiekt przejściowy, który może być przetworzony do wynikowego zbioru danych.

Jeśli chodzi o operację `forEach`, cechuje się ona tym, że w odróżnieniu od pozostałych, nie zwraca wartości. `ForEach` jest w zasadzie, funkcyjnym wariantem pętli `for`. Polega na wykonaniu instrukcji zdefiniowanych funkcją, dla każdego elementu. Operacje wykonywane przez tę operację, mogą być także wykonane przy użyciu `map`. Jednakże `map` zwraca wynik, co powoduje dodaktowy narzut na kreowanie obiektu wynikowego. W przypadku `forEach` wyniku nie ma, zaś jedynym jej celem jest przetworzenie wszystkich elementów.

Operacja `reduce`, polega na `spłaszczeniu`, `zredukowaniu` przetwarzanego zbioru. Funkcja w typ przypadku operuje na dwóch elementach kolekcji. Elemencie `poprzednim`, który stanowi wynik poprzedniego wykonania funkcji (bądź pierwszy element kolekcji w przypadku pierwszego wykonania), oraz element `aktualny`, czyli `n + 1` obiekt, dla `n` stanowiącego numer iteracji. Wynikiem jest zredukowanie parametrów funkcji, do pojedynczej wartości. Przykładem takiej operacji może być choćby sumowanie wartości listy, czy łączenie listy słów w wynikowe zdanie. Operacja ta bywa niekiedy określana jako `flatten`. Jednym z wyspecjalizowanych wariantów tej metody jest `joiner`, służący właśnie do łączenia listy łańcuchów znakowych, przy zastosowaniu separatora.

`Find` znajduje pierwszy element wejściowego zbioru, który spełnia dany predykat. Zachowania w przypadku nieznalezienia elementu, są różne w zależności od konkretnego języka. Metoda ta różni się od `filter`, różni się właśnie faktem przerwania procesowania po znalezieniu pierwszego rozwiązania.

Metody `every` oraz `some`, występujące także odpowiednio pod nazwami `all` i `any`, czy metoda przeciwna do `some`, czyli `none`, są metodami zwracającymi wartość logiczną. Cel tych metod jest możliwy do osiągnięcia także przy użyciu metody `filter`, jednakże dla tych metod ważniejsze jest sprawdzenie istnienia konkretnych elementów niż określenie, jakie elementy dane kryterium spełniają lub nie.

1.9 Realizacja paradygmatu w języku C#

Język C# został stworzony przez firmę Microsoft. Pierwsze informacje o języku pojawiły się w roku 2000. Jednak wygodne funkcyjne programowanie, stało się w tym języku możliwe, dopiero od wersji C# 3.0 działającej na platformie .NET Framework w wersji 3.5, wprowadzonej siedem lat później. LINQ czyli Language INtegrated Query, jest jedną ze składowych pakietu .NET Framework, które pozwala na wykonywanie zapytań przypominających te znane z języka SQL, na obiektach. LINQ definiuje na tyle przejrzysty standard, że doczekał się implementacji w językach takich jak PHP, JavaScript, TypeScript i ActionScript. Natywnym środowiskiem tej technologii jest jednak język C#. Patrząc na kształt tego czym jest i co oferuje LINQ, można odnieść wrażenie, że jego twórcom przyświecały podobne cele, jak autorowi niniejszej pracy. Oddzielenie logiki operacji na obiektach, od konkretnych realizacji tych obiektów. Zapytania do kolekcji; w przypadku LINQ do dokumentów XML, klas będących implementacjami interfejsu `IEnumerable`, czy też bezpośrednio baz danych; nie powinny być różne w zależności od tego na jakim źródle danych pracujemy. Programistę powinno interesować tylko opisanie kryteriów wyniku. Resztą zajmie się już środowisko uruchomieniowe.

LINQ oferuje dwa rodzaje zapisu operacji [3]. Pierwszy z nich to notacja z kropką, jest to sposób, który będzie zdecydowanie bliższy ludziom, wywodzącym swe korzenie z obiektowych języków programowania. Na istniejącej kolekcji używamy kropki, tak ja wywoływalibyśmy dowolną inną metodę. Drugi wariant to notacja zapytań, będzie ona bliższa osobom zaznajomionym z bazami danych, ponieważ jej składnia jest bardzo podobna do składni języka SQL. Na listingu 1.3, zaprezentowano prosty przykład obu składni.

Listing 1.3: Dwie formy LINQ

```
1  // Notacja z kropka
2  var querry = kolekcja.Select(n=>n).Where(s=>s <= 4).ToArray();
3
4  // Notacja zapytan
5  from n in kolekcja
6  where n <= 4
7  select n
```

Jako API do funkcyjnej obsługi kolekcji LINQ pokrywa wszystkie wymienione główne funkcje takiego rozwiązania. Nazwy funkcji są jednak bliższe tym znanym z języka SQL niż tym wykorzystywanym w innych tego rodzaju bibliotekach. Mamy zatem `Where` odpowiadające opisywanej instrukcji `filter`. `Select` jest realizacją `map`. Jeśli chodzi o instrukcję `forEach` to należy najpierw zebrać elementy do listy za pomocą `ToList`, następnie korzystać z `ForEach`. Jest to instrukcja, do której twórcy przywiązali najmniej wagi, najprawdopodobniej ze względu na fakt, że LINQ ma służyć jako rozszerzenie do zapytań o obiektu, nie zaś jako sposób wykonywania na nich określonych czynności. Operacja `reduce`, określona jest jako `SelectMany`, zaś `find` jako `First`. Metody `every` i `some` to odpowiednio `All` i `Any`.

Jako, że LINQ stanowi rozszerzenie, mające na celu symulowanie zachowania języka zapytań, posiada ono także inne możliwości. Oferowane są zatem takie

funkcje jak `GroupBy`, czyli w praktyce funkcja odwrotna do `reduce`, `OrderBy` czyli operacja sortująca, przyjmująca komparator, oraz spora ilość metod, których realizacja jest możliwa przy użyciu już opisanych, zostały jednak zdefiniowane dla czystej wygody programisty. Przykładami takich metod mogą być `Max`, czy `Sum`.

Dotychczas opis dotyczył tylko funkcyjnych rozwiązań do obsługi kolekcji. Do realizacji definiowania funkcji przekazywanych jako parametry wymienionych procedur, wykorzystano wyrażenia lambda. Ich składnia jest prawie identyczna do tej prezentowanej w przypadku funkcji strzałkowych. Jedyną różnicą jest opcjonalne wskazanie typu parametru, stosowane w przypadku kiedy niemożliwe dla kompilatora jest wywnioskowanie typu na podstawie wartości. Składnie tej instrukcji wygląda wówczas tak jak zaprezentowano na listingu 1.4.

Listing 1.4: Wyrażenie lambda z jawnym wskazaniem typu

```
1 (int x, string s) => s.Length > x
```

Tak jak w przypadku języka JavaScript, lambda przechwytuje swój kontekst przez referencje.

Język C# jest zdecydowanie dobrym miejscem do rozpoczęcia nauki funkcyjnego przetwarzania kolekcji. Jego funkcyjne aspekty zostały stworzone głównie w tym właśnie celu. Natomiast możliwość stosowania alternatywnej, podobnej do języka SQL składni, sprawia, że łatwo odnajdą się w nim zarówno osoby programujące do tej pory imperatywnie, jak i te które znają głównie język zapytań bazodanowych.

1.10 Realizacja paradygmatu w języku Java

Java jako język doczekała się API funkcyjnego dopiero w 2014 roku. Konkretnie wydanie wersji Java 8, które nastąpiło 14 marca 2014 wprowadziło zarówno obsługę wyrażeń lambda jak i strumieni. Wersja 8 języka Java była bez wątpienia wersją przełomową, dotychczas język nie był prawie wcale przygotowany na stosowanie funkcyjnego podejścia. Pomimo tego istniała biblioteka oferowana przez firmę Google, pozwalająca na tworzenie pseudo-funkcyjnego kodu. Guava oferowała, przetwarzanie kolekcji poprzez przekazywanie, implementacji odpowiedniego interfejsu do metod realizujących filtrowanie czy mapowanie kolekcji. Jednakże brak wyrażeń lambda powodował konieczność bezpośredniego tworzenia wymaganych obiektów. Dodatkowo, konieczność używania dodatkowej biblioteki bywa czasami przeszkodą jeśli chodzi o pisanie aplikacji biznesowych. Java 8 wprowadziła cztery elementy, które umożliwiły całkowitą zmianę sposobu programowania, są to:

1. pakiet `java.util.stream`
2. metody domyślne
3. interfejsy funkcyjne
4. wyrażenia lambda

Dopiero mając te cztery elementy programowanie funkcyjne w Javie stało się możliwe.

Po pierwsze pakiet `java.util.stream`. Jest to pakiet zawierający głównie interfejsy, oraz klasy pomocnicze. Dwa najważniejsze z nich to `Stream` oraz `Collector`. Pierwszy z nich to główny interfejs programowania funkcyjnego, to właśnie ten interfejs zaopatruje kolekcje w niezbędne metody. Oferuje on między innymi, metody określone w niniejszej pracy jako niezbędne, czyli `filter`, `map`, `forEach`, `reduce`, `findAny`, `allMatch`, `anyMatch`. W odróżnieniu od poprzednich języków, operacja `findAny`, nie przyjmuje konkretnej funkcji filtrującej, lecz wymaga wcześniejszego przefiltrowania strumienia i wykonania jej na wyniku. Drugi z interfejsów czyli `Collector`, zapewnia możliwość powrotu z interfejsu `Stream`, do standardowych kolekcji. Mając strumień często będziemy chcieli źebrać"go do listy czy mapy. Do tego celu służą właśnie instancje obiektów implementujących interfejs `Collector`. Implementacja `Collectora` nie jest zadaniem trywialnym, dla niezaznajomionych z koncepcją funkcyjnego programowania. Wymaga implementacji, aż pięciu funkcji, odpowiedzialnych odpowiednio za:

1. Zapewnienie funkcji tworzącej obiekt akumulatora. Może to być na przykład obiekt implementujący wzorzec budowniczego.

2. Funkcji akumulatora, zbierającego pośrednie wyniki operacji. Trzymając się przykładu budowniczego, będzie to funkcja która przyjmuje obiekt akumulatora stworzony w pierwszym punkcie, oraz obiekt, który należy dodać do kolekcji, a następnie po prostu dodaje obiekt przy pomocy otrzymanego budowniczego.

3. Funkcji łączącej akumulatory, potrzebnej w przypadku równoległego wykonania operacji. W naszym przykładzie funkcja przyjmująca dwóch budowniczych i łączących przetrzymywane przez nich elementy.

4. Funkcji kończącej operację, konwertującej akumulator w docelową kolekcję. W kontekście wzorca budowniczego byłaby to metoda zwracająca operację kończącą czyli `build`.

5. Charakterystyk przetwarzanego strumienia, charakterystyki służą jako podpowiedź dla kompilatora i wirtualnej maszyny, w celu określenia możliwości kolektora. Przykładowe charakterystyki to `CONCURRENT` czy `UNORDERED`. Poprawne określenie charakterystyk może znacznie poprawić wydajność kolektora.

Na szczęście jedna z klas z pakiety `java.util.stream` posiada przygotowany zestaw podstawowych kolektorów. Pozwalając na łatwe zebranie wyników operacji na strumieniach na przykład do listy, czy słownika. Dostępna jest na przykład także metoda `groupingBy`, o jakiej była wcześniej mowa w przypadku LINQ. W ostateczności zawsze można zaimplementować własny kolektor dopasowany do konkretnych potrzeb.

Sam interfejs `Stream` nic by jednak nie dawał, gdyby standardowe kolekcje nie stanowiły jego implementacji. Tutaj na przeszkodzie stanął fakt powszechności języka Java i ilości kolekcji implementowanych nie w JDK, lecz przez użytkowników. Jeśli ktoś uznał domyślną implementację interfejsu `List` czy `Map` za nieoptymalną, mógł zaimplementować ten interfejs po swojemu, zaś w swoim API oferować jako typ wyjściowy po prostu interfejs. Ciężko w zasadzie nawet mówić o czymś takim jak domyślna implementacja, skoro zarówno

dla listy jak i mapy w pakiecie `java.util` znajdujemy po dwie implementacje (`ArrayList`, `LinkedList` dla `List`, oraz `HashMap` i `LinkedHashMap` dla `Map`). Twórcy stanęli więc przed następującym problemem. Należało dodać metody pozwalające na prostą konwersję interfejsów kolekcji do strumienia, bez niszczenia aktualnych implementacji tych interfejsów. Znalezione rozwiązanie to metody domyślne. Dotychczas interfejs mógł tylko deklarować nagłówki metod, bez zapewnienia choćby szkieletowej implementacji. Metody domyślne to metody w interfejsie oznaczone słowem kluczowym `default`. Metody te nie muszą, choć mogą, być implementowane przez klasy implementujące dany interfejs. Po wprowadzeniu tej funkcjonalności do języka wystarczyło tylko zaimplementować metodę `stream` oraz inne wymagane do operacji na strumieniach w interfejsie `Collection`.

Wspomniana wcześniej Guava, bazowała na wykorzystywaniu specjalnych interfejsów, w których najczęściej wystarczyło zaimplementowanie jednej metody, `apply`. Choć metoda była tylko jedna to wymuszała implementację konkretnego interfejsu. Nawet jeśli mieliśmy obiekt, udostępniający tylko jedną metodę, nie mogliśmy potraktować go jako domknięcia ponieważ, środowisko uruchomieniowe nie wiedziałoby, jaką metodę powinno wykonać. Java 8 wprowadza pojęcie interfejsu funkcyjnego. Jest to interfejs posiadający tylko jedną nie domyślną metodę. Dobrą praktyką jest oznaczanie tego typu interfejsów specjalną adnotacją `@FunctionalInterface`, nie jest jednak ona wymagana do działania. Dzięki wprowadzeniu interfejsów funkcyjnych, możliwe stało się wprowadzenie metod operujących na instancjach tych właśnie interfejsów. Nie rozwiązywało to jednak do końca problemu który miała Guava. Dalej należało tworzyć obiekt z użyciem pełnej składni i implementować konkretną metodę. Na tym etapie składnia przypominałaby w dużym stopniu JavaScript, z tą różnicą, że zamiast słowa kluczowego `function` mielibyśmy konstrukcje `new ObiektFunkcyjny{}` z implementacją, konkretnej metody. Nie byłoby to zbyt wygodne rozwiązanie.

W tym właśnie momencie przechodzimy do ostatniego punktu, czyli wyrażeń lambda. Jak można się już domyślić wyrażenia lambda są realizacją interfejsów funkcyjnych. Konkretna lambda jest obiektem implementującym funkcyjny interfejs. Podstawowe pakiety Javy zawierają wiele interfejsów funkcyjnych będących docelowymi dla wyrażeń lambda i referencji metod. Co najważniejsze lambdy stanowią wygodne rozwiązanie i nie wymagają tworzenia obiektu wprost. Składnia jest bardzo podobna do tej znanej już z języka C#. Jedyną różnicą jest zastosowanie pojedynczej strzałki, czyli `->` zamiast `=>`. Zmienne z bloku tworzenia lambdy są przechwytywane, z jednym zastrzeżeniem. Muszą być one oznaczone słowem kluczowym `final`, bądź zachowywać się jak zmienne tym słowem oznaczone, określane jest to zwrotem `effectively final`. Oznacza to, że nie można przypisać do nich, innego obiektu. Można jednak zmieniać własności tego obiektu, dlatego, rzadko stanowi to jakikolwiek problem. Dostępny jest także inny wariant wykorzystania funkcyjnych interfejsów. Zamiast używać wyrażenia lambda można zastosować referencję metody. Referencja metody jest wskazaniem na konkretną funkcję z danego typu, po jej użyciu tworzy się instancja funkcyjnego interfejsu, zawierająca tą właśnie metodę. Przykład zastosowania, wraz z porównaniem do składni wyrażenia lambda, pokazany został na listingu 1.5.

Listing 1.5: Wyrażenie lamda, a referencja metody

```
1 List<Klienci> zieloniKlienciLamda = klienci.stream().filter(k -> k.
      jestZielony()).collect(toList());
2 List<Klienci> zieloniKlienciReferencja metody = klienci.stream().
      filter(Klient::jestZielony).collect(toList());
```

Składnia referencji metody bywa na początku trochę mniej intuicyjna, jako że nie wskazuje bezpośrednio obiektu na którym wykonana zostanie metoda.

Java w wersji 7 nie dawała praktycznie, żadnych możliwości funkcyjnego rozwiązywania problemów. Jeśli zaś chodzi o wersję 8, to pokrywa ona całe spektrum możliwości. Choć można wytknąć pewne wady wybranym rozwiązaniom, na przykład metody domyślne wprowadzają, możliwość zachowań takich jak przy wielodziedziczeniu, to zakres udostępnionych funkcjonalności, zdecydowanie wynagradza te braki.

1.11 Realizacja paradygmatu w języku C++

Rozdział 2

Zalety programowania funkcyjnego w rozwiązaniach praktycznych

2.1 Podstawowe problemy i zadania w programowaniu aplikacji biznesowych

Biznes wymaga oprogramowania. Ta kwestia nie podlega wątpliwości. Aplikacje bankowe, kasy w sklepach, giełda czy obecnie nawet telefony menadżerów, wszystkie te urządzenia i dziedziny wymagają odpowiednich programów pozwalających im wykonywać określone zadania. Popularność technologii informatycznych sprawia, że w zasadzie każda firma chcąca się rozwijać musi z tychże technologii korzystać. Czy to tworząc stronę internetową, czy monitorując pracowników, zarządzając urlopami czy katalogując zamówienia. O ile istnieją oczywiście sektory, w których od programisty wymaga się algorytmicznego myślenia, takie jak rynek procesorów, kart graficznych, badania naukowe czy związana blisko z biznesem kryptografia, to doświadczenie autora pozwala twierdzić, że większość informatyków znajdzie zatrudnienie przy znacznie mniej wymagających zadaniach. Poziom mocy obliczeniowej obecnych komputerów pozwala nam w wielu przypadkach nie przejmować się złożonością obliczeniową czy pamięciową podejmowanych przez programy działań (oczywiście w granicach rozsądku). Do tego z poziomu biznesu, oprogramowanie jest niczym innym jak tylko kosztem. Kosztem uzasadnionym z którym się pogodzono, ale jednak kosztem. A koszta należy minimalizować, dlatego jeśli menadżer projektu stanie przed wyborem pozostawienia programisty zarabiającego pięć tysięcy złotych na miesiąc z problemem optymalizacji algorytmu, albo zakupienia szybszego procesora z trzy tysiące złotych, z radością kupi procesor i przypisze sobie zasługi oszczędzenia dwóch tysięcy. Oczywiście gdyby tego typu problemy były częste, należałby szukać programisty, który na tego typu zadanie będzie potrzebował tydzień zamiast miesiąca. Jednak w codziennej pracy tego typu problemy prawie nie występują. Deweloperzy produkują prawie seryjnie aplikacje, które z ich punktu widzenia niczym się od siebie nie różnią. Implementacja sklepu internetowego sprzedającego traktory i sklepu sprzedającego koszule, są tym sa-

mym zadaniem. Różnią się etykiety, zdjęcia i cena. Zadanie dla programisty jest jednak w obu przypadkach takie samo.

1. Odczyt danych z bazy.
2. Przekazanie ich do warstwy widoku.
3. Wyświetlenie ich dla klienta.
4. Przyjęcie danych od klienta.
5. Przekazanie ich do warstwy bazy danych
6. Zapis danych do bazy.

Jak widać punkty trzeci i czwarty są nieuniknione. W końcu to klient jest powodem powstania aplikacji, nie da się go z niej wyeliminować. Punkty pierwszy i drugi również są konieczne. Czy nazwiemy to bazą danych, czy systemem plików, musi istnieć miejsce, w którym dane będą przechowywane. Chociażby po to, żeby wiedzieć co należy zrobić. Wspomniane punkty, można oczywiście rozszerzać o dodatkowe operacje, takie jak wysłanie maila, wydrukowanie faktury czy wykonanie przelewu on-line. Nie są to jednak zagadnienia wpływające na kształt procesu. Pozostają więc punkty drugi i piąty. Łatwo zauważyć, że gdyby dane w bazie znajdowały się w postaci możliwej do zrozumienia przez klienta można byłoby te kroki wyeliminować. Wiemy jednak, że tak nie jest. Komputery preferują inny sposób prezentacji danych od człowieka. Skoro te kroki są więc konieczne, należy sprawić; z punktu widzenia menadżera, żeby były jak najtańsze; z punktu widzenia programisty, żeby były jak najmniej uciążliwe. I w tym właśnie momencie pojawia się okazja do wykorzystania programowania funkcyjnego.

2.2 Porównanie różnic podejścia funcyjnego i imperatywnego w rozwiązywaniu problemów

Z poprzedniego paragrafu wiemy, że w celu optymalizacji procesu tworzenia należy skupić się na przyspieszeniu obsługi przetwarzania danych. Konkretnie tłumaczenia ich z formatu zrozumiałego przez komputer do formatu rozumianego przez użytkownika. Operacje tego typu również są zazwyczaj schematyczne i przewidywalne. Najczęściej należy iterować uzyskaną z bazy danych kolekcję odpowiednio ją filtrując i modyfikując. Kończy się to zazwyczaj serią pętli oraz tworzenia tymczasowych kolekcji. Kod odpowiedzialny za tego typu zadania określany jest w żargonie `boilerplate`. Określenie to oznacza kod, który nie służy żadnym praktycznym celom, jest jednak konieczny ze względu na wymagania języka; na przykład `gettery` i `settery`, czy instrukcje tworzenia tymczasowych obiektów. Jak już zostało wspomniane, zadania tego typu są kosztem, który należy minimalizować. Z pomocą przychodzą aspekty programowania funkcyjnego. Strumienie i wyrażenia lambda pozwalają na iterowanie kolekcji `w miejscu`, obiekty pośrednie są tworzone automatycznie. Wyrażenia filtrujące mogą być składane w łańcuchy, pozwalając na szybkie i proste tworzenie zaawansowanych przekształceń. Choć początkowe może nie wyglądać to

na duży zysk, w szerszej perspektywie kumuluje się do pokaźnych zysków czasowych podczas tworzenia oprogramowania. Porównajmy zatem rozwiązania tego samego problemu na sposób imperatywny (listing 2.1) oraz funkcyjny (listing 2.2).

Listing 2.1: Podejście imperatywne

```
1  public Collection<People> imperative(Collection<User> users) {
2   List<Person> people = new ArrayList<>();
3   for (User user : users) {
4    if (user.getBirthday().after(new Date(1999, 12, 1))
5        && user.getRole() == UserRole.GUEST) {
6       Person person = new Person(user.getName(), user.getLastName())
            ;
7       people.add(person);
8       }
9   }
10  return people;
11 }
```

Listing 2.2: Podejście funkcyjne

```
1  public Collection<People> functional(Collection<User> users) {
2   return users.stream()
3       .filter(user -> user.getBirthday().after(new Date(1999, 12, 1)
            ))
4       .filter(user -> user.getRole() == UserRole.GUEST)
5       .map(user -> new Person(user.getName(), user.getLastName()))
6       .collect(toList());
7  }
```

Choć przykład jest prosty i niewiele zyskujemy w kontekście ilości linii kodu. To już tutaj da się dotrzec pewne zyski. Wystarczy spróbować dokonać analizy logicznej tego co próbujemy uzyskać. Otrzymujemy kolekcję użytkowników, przyjmijmy, że klasa `User` jest klasą przedstawiającą obiekt bazodanowy. Chcemy wyfiltrować użytkowników urodzonych po 1. grudnia 1999 roku, których rola w systemie to `GUEST`. Na podstawie wyfiltrowanej kolekcji chcemy utworzyć kolekcję obiektów typu `People` (przyjmijmy, że są to obiekty zrozumiałe dla użytkownika, które można przedstawić na interfejsie).

Jeśli problem postawiony jest w ten sposób, funkcyjne podejście dużo bardziej odpowiada temu co chcemy uzyskać. Bierzemy otrzymaną kolekcję, musimy zawołać na niej metodę `stream()` (to w zasadzie cały `boilerplate` w tej metodzie), a następnie dokonujemy po kolei opisanych transformacji. Operacja filter ograniczająca datę urodzenia. Następnie operacja sprawdzająca rolę użytkownika (operacja ta wydzielona jest dla czytelności przykładu, można złożyć oba wywołania filter w jedno przy pomocy operatora `&&`). W lini 5 przykładu znajduje się mapowanie obiektów `User` na odpowiadające im `Person`. Ostania linia to zebranie wartości w kolekcję, którą chcemy zwrócić.

Podejście iteracyjne od początku prezentuje inny sposób myślenia. Na początku musimy stworzyć wynikową kolekcję, potem iterować tą którą otrzymaliśmy i dopiero wewnątrz pętli, dokonujemy analizy warunku. Samodzielnie też dokonujemy dodania elementu do wynikowej kolekcji. Można powiedzieć, że żadna z tych operacji nie jest specjalnie skomplikowana, dlatego nie ma żadnej

różnicy w zastosowanych podejściach. Różnice jednak są i przy dużej ilości tego typu kodu objawiają się w sposób znaczący.

Po pierwsze podejście funkcyjne jest bliższe opisowi rozwiązania problemu, a co za tym idzie wymaga mniejszej analizy tego co należy napisać. Jako osoba przyzwyczajona do programowania funkcyjnego, pisząc przykład imperatywny, zacząłem od pętli iterującej po użytkownikach, dopiero po napisaniu `ifa` orientując się, że potrzebuje kolekcji do której dodam wyniki. Często pomijanym szczegółem jest stopień zagnieżdżenia kodu. W przypadku dodania kolejnego warunku w podejściu funkcyjnym, dodamy po prostu koleją instrukcję `filter`, a kolejne mapowanie (na przykład na ciągi znaków z imieniem) to instrukcja `map`. Dla podejścia imperatywnego, albo rozbudowujemy warunek `ifa` albo zagnieżdżamy jednego w drugim, co nie jest zachowaniem porządanym. Większość programistów uważa kod z ilością zagnieżdżonych instrukcji większą niż trzy za mało czytelny. Kolejnym elementem jest samo stworzenie wyniku. Gdyby okazało się, że metoda powinna zwracać `Set` osób to w przykładzie funkcyjnym zmieni się tylko funkcja przekazana do metody `collect` (będzie to `toSet()`), zaś w przykładzie imperatywnym zarówno linia tworząca kolekcję jak i dodająca element (`Set` posiada metodę `put` zamiast `add`).

2.3 Analiza przykładowych rozwiązań w przypadku różnych paradygmatów

Przykłady z poprzedniego rozdziału pokazują, część zysków które wynikają z funkcyjnego podejścia do rozwiązania problemów. Zalet jest jednak więcej. Listingi 2.3 i 2.4 pokazują różnice o ile łatwiej można pogrupować kolekcję przy użyciu podejścia funkcyjnego.

Listing 2.3: Podejście imperatywne

```
1  public void imperativeGrouping(Collection<User> users) {
2      Map<UserRole, List<User>> result = new HashMap<>();
3      for (User user : users) {
4       List<User> usersWithRole = result.get(user.getRole());
5       if (usersWithRole == null) {
6        usersWithRole = new ArrayList<>();
7        usersWithRole.add(user);
8        result.put(user.getRole(), usersWithRole);
9       } else {
10       usersWithRole.add(user);
11      }
12     }
13 }
```

Listing 2.4: Podejście funkcyjne

```
1  public void functionalGrouping(Collection<User> users) {
2   Map<UserRole, List<User>> result = users.stream().collect(
        groupingBy(User::getRole));
3  }
```

Przykład imperatywny wymaga tworzenia wszystkich kolekcji, specyfika mapy powoduje konieczność sprawdzania czy lista już istnieje i w razie konieczności

utworzenia jej. Po raz kolejny, żadna z operacji nie jest specjalnie skomplikowana. Jest to jednak dość długa sekwencja zadań. Po raz kolejny także rozwiązanie nie jest prostą realizacją opisu w języku naturalnym. Oczekiwane rozwiązanie to otrzymanie kolekcji użytkowników pogrupowanych na podstawie roli jaką pełnią. Tymczasem program polega na tworzeniu list użytkowników, tworzeniu mapy wynikowej, sprawdzaniu czy dane listy już istnieją i uzupełnianiu ich wartości.

Rozwiązanie funkcyjne pokazuje w pełni deklaratywną stronę programowania funkcyjnego. Cały program to jedna linia, która jest w zasadzie opisem problemu. Na początek `boilerplate` w postaci `stream`, czyli instrukcja dla komputera aby traktować kolekcję jako strumień danych. Następnie `collect` czyli rozkaz zebrania użytkowników do jakieś wynikowej kolekcji. `groupingBy` stanowi opis sposobu wykonania tego rozkazu, jest to funkcja zbierająca iterowane obiekty w grupy (mapę). Parametrem instrukcji `groupingBy`, jest funkcja określająca własność, która stanowi o przynależności do konkretnej grupy. Funkcja przekazana jest jako referencja metody pobierającej role z obiektu użytkownika.

Funkcyjne rozwiązanie problemu jest zdecydowanie prostsze. Mniej kodu do napisania i mniej analizy tego co należy napisać, oznacza mniej możlwości popełnienia błędu. W rozwiązaniu funkcyjnym nie ma w zasadzie miejsca w którym można się pomylić.

Rozdział 3

Analiza wydajności istniejących rozwiązań funkcyjnych

3.1 Prezentacja zakresu testów i opis środowiska

3.2 Kryteria wyboru rozwiązań wraz z przedstawieniem wybranych

3.3 Przedstawienie i analiza wyników

Rozdział 4

Prezentacja opracowanego rozwiązania

4.1 Kod źródłowy

```
1  #ifndef STREAM_API
2  #define STREAM_API
3
4  #include <iostream>
5  #include <vector>
6  #include <queue>
7  #include <functional>
8
9  namespace stream {
10
11     class streamAlreadyConsumedException {
12     };
13
14     template<class T>
15     class stream;
16
17     template<class T, class... Args>
18     class streamOperation;
19
20     template<class T>
21     class stream {
22
23     public:
24         stream();
25
26         stream(const std::vector<T> &data);
27
28         bool allMatch();
29
30         bool anyMatches();
31
32         T find();
33
34         stream<T> *filter(std::function<bool(T)> predicate);
35
36         void foreach(std::function<void(T)> mappingFunction);
37
38         T reduce(std::function<T(T, T)> reductorFunction);
```

```
39
40          std::vector<T> *toVector();
41
42          template<class R>
43          stream<R> *map(std::function<R(T)> mappingFunction);
44
45          stream<T> *peek();
46
47      protected:
48          stream(const std::vector<T> &data, std::vector<
                streamOperation<bool(T)> *> *predicates);
49
50          template<class R>
51          R executeMappingOperation(streamOperation<R(T)> *operation,
                T value);
52
53          void checkConsumed(bool consume);
54
55      private:
56          std::vector<streamOperation<bool(T)> *> *predicates;
57          std::vector<T> *underlyingVector;
58          bool consumed;
59      };
60
61      template<class T, class... Args>
62      class streamOperation<T(Args...)> {
63      public:
64          std::function<T(Args ...)> *fun;
65
66          streamOperation(std::function<T(Args ...)> *fun) {
67              this->fun = fun;
68          }
69      };
70
71      template<class T>
72      stream<T>::stream(const std::vector<T> &data, std::vector<
            streamOperation<bool(T)> *> *predicates) {
73          this->underlyingVector = new std::vector<T>(data);
74          this->predicates = predicates;
75          this->consumed = false;
76      }
77
78      template<class T>
79      stream<T>::stream(const std::vector<T> &data) {
80          this->underlyingVector = new std::vector<T>(data);
81          this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
82          this->consumed = false;
83      }
84
85      template<class T>
86      stream<T>::stream() {
87          this->underlyingVector = new std::vector<T>();
88          this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
89          this->consumed = false;
90      }
91
92      template<class T>
93      stream<T> *stream<T>::filter(std::function<bool(T)> predicate)
            {
94          checkConsumed(false);
```

```
95          auto *op = new streamOperation<bool(T)>(&predicate);
96          predicates->push_back(op);
97          return this;
98      }
99
100     template<class T>
101     T stream<T>::find() {
102         std::vector<T> *pVector = this->toVector();
103         return pVector->empty() ? NULL : pVector->front();
104     }
105
106     template<class T>
107     bool stream<T>::anyMatches() {
108         return !toVector()->empty();
109     }
110
111     template<class T>
112     bool stream<T>::allMatch() {
113         unsigned int fullSize = underlyingVector->size();
114         return toVector()->size() == fullSize;
115     }
116
117     template<class T>
118     template<class R>
119     stream<R> *stream<T>::map(std::function<R(T)> mappingFunction)
            {
120         streamOperation<R(T)> *op = new streamOperation<R(T)>(&
                mappingFunction);
121         std::vector<T> *filtered = toVector();
122         std::vector<R> *result = new std::vector<R>();
123         for (typename std::vector<T>::iterator it = filtered->begin
                ();
124              it != filtered->end();) {
125             T v = (*it);
126             auto val = this->executeMappingOperation(op, v);
127             result->push_back(val);
128             ++it;
129         }
130         return new stream<R>(*result);
131     }
132
133     template<class T>
134     T stream<T>::reduce(std::function<T(T, T)> reductorFunction) {
135         streamOperation<T(T, T)> *op = new streamOperation<T(T, T)
                >(&reductorFunction);
136         std::vector<T> *filtered = toVector();
137         auto previousVal = filtered->front();
138         for (typename std::vector<T>::iterator it = filtered->begin
                () + 1;
139              it != filtered->end();) {
140             previousVal = reductorFunction(previousVal, (*it));
141             ++it;
142         }
143         return previousVal;
144     }
145
146     template<class T>
147     void stream<T>::foreach(std::function<void(T)> foreachOperation
            ) {
148         std::vector<T> *filtered = toVector();
149         for (typename std::vector<T>::iterator it = filtered->begin
                ();
```

```
150                 it != filtered->end();) {
151                 T v = (*it);
152                 foreachOperation(v);
153                 ++it;
154             }
155         }
156
157         template<class T>
158         std::vector<T> *stream<T>::toVector() {
159             checkConsumed(true);
160             std::vector<T> *result = new std::vector<T>();
161
162             for (auto it = underlyingVector->begin(); it !=
                    underlyingVector->end(); ++it) {
163                 auto remove = false;
164                 T v = (*it);
165                 for (auto oIt = predicates->begin(); oIt != predicates
                        ->end(); ++oIt) {
166                     auto val = (*(*oIt)->fun)(v);;
167                     if (!val) {
168                         remove = true;
169                         break;
170                     }
171                 }
172                 if (!remove) result->push_back(v);
173             }
174             return result;
175         }
176
177         template<class T>
178         stream<T> *stream<T>::peek() {
179             for (auto it = underlyingVector->begin(); it !=
                    underlyingVector->end(); ++it) {
180                 auto remove = false;
181                 T v = (*it);
182                 for (auto oIt = predicates->begin();
183                      oIt != predicates->end(); ++oIt) {
184                     auto val = (*(*oIt)->fun)(v);;
185                     if (!val) {
186                         remove = true;
187                         break;
188                     }
189                 }
190                 if (!remove) std::cout << v << " ";
191             }
192             std::cout << std::endl;
193             return this;
194         }
195
196         template<class T>
197         template<class R>
198         R stream<T>::executeMappingOperation(streamOperation<R(T)> *
                operation,
199                                              T value) {
200             auto function = (*operation->fun);
201             return function(value);
202         }
203
204         template<class T>
205         void stream<T>::checkConsumed(bool consume) {
206             if (this->consumed) throw new
                    streamAlreadyConsumedException();
```

```
207             this->consumed = consume;
208         }
209     }
210
211     #endif
```

Prezentowany kod dostępny jest także w sieci pod adresem https://github.com/krzysztof-osiecki/StreamAPI

4.2 Opis zakresu możliwości przygotowanego rozwiązania

4.3 Porównanie przygotowanego API z istniejącymi w innych językach

4.4 Testy weryfikujące wydajność rozwiązania względem natywnego języka C++

Czas wykonania każdego z testów mierzony był z pomocą następującej struktury:

Listing 4.1: Pomiar czasu[1]

```
1  template<typename TimeT = std::chrono::milliseconds>
2  struct measure {
3   template<typename F, typename ...Args>
4   static typename TimeT::rep execution(F &&func, Args &&... args) {
5   auto start = std::chrono::steady_clock::now();
6   std::forward<decltype(func)>(func)(std::forward<Args>(args)...);
7   auto duration = std::chrono::duration_cast<TimeT>
8    (std::chrono::steady_clock::now() - start);
9   return duration.count();
10  }
11 };
```

Testy wykonane były w dwóch wartiantach, `dużym`, w którym kolekcja zawierała milion elementów, oraz `małym` w którym elementów było dziesięć tysięcy. Dla wartości mniejszej wszystkie wykonania trwały mniej niż 1ms. Bazową kolekcję stanowił `std:vector`. Czas przygotowywania danych nie był wliczany do czasu trwania testu. Wszystkie testy wykonano na tej samej maszynie testowej. Był to komputer wyposażony w procesor Intel Core i5-4460 3.2GHz oraz 8GB pamięci RAM. Komputer pracował na 64 bitowej wersji systemu Windows 10. Do kompilacji posłużył kompilator g++ zapewniany przez środowisko MinGW w wersji 3.20. Do budowania projektu posłużyło narzędzie CMake. Wykorzystano siedem różnych podejść w celu sprawdzenia różnic. Podejście pierwsze polegało na zastosowaniu `stream:stream` oraz operacji filter z użyciem wyrażenia lambda do wartościowania wyniku filtrowania. Podejście drugie to iteracja wektora za pomocą pętli po iteratorach, z tworzeniem nowej listy i dodawaniem do niej elementów spełniających kryteria. Podejście trzecie to analogiczna iteracja za pomocą iteratora, z tą różnicą, że tym razem elementy nie spełniające

Tablica 4.1: Pojedyncze filtrowanie, 10 000 elementów

1	2	3	4	5	6	7
0.55	0.27	2.91	0.17	0.48	3.62	0.44

Tablica 4.2: Pojedyncze filtrowanie, 1 000 000 elementów

1	2	3	4	5	6	7
60.8	27	40186.4	21.6	46.6	40246.2	43.2

kryterium były usuwane z wektora za pomocą funkcji `vector:erase`. Podejście czwarte to iteracja za pomocą pętli wykorzystującej zmienną całkowitą z dostępem do elementu vectora za pomocą funkcji `vector:at`, wynikiem była nowo tworzona lista, podobnie jak w podejściu pierwszym. W powyższych trzech metodach wartościowanie kryterium było wykonywane za pomocą zwykłej instrukcji warunkowej bez użycia wyrażenia lamda. Metody rozwiązania piąta, szósta i siódma, są analogicznymi odpowiednikami drugiej, trzeciej i czwartej. Różnicą jest fakt, iż wartościowanie kryterium zostało w nich wykonane przy użyciu wyrażenia lambda. Celem było sprawdzenie jak duży narzut czasowy nakłada dodatkowe opakowanie instrukcji w funkcję anonimową. W tabelach prezentujących wyniki, odniesienia do poszczególnych metod przedstawione, są za pomocą numerów w kolejności opisu. Każdy z testów powtórzono tysiąckrotnie, przedstawione wyniki są wartością średnią. Wszystkie wartości określone są w milisekundach.

Pierwszy test polegał na ograniczeniu kolekcji tylko do elementów parzystych. Zastosowana była więc tylko jedna operacja filtrowania. Wyniki przedstawione są w tabeli 4.1.

4.5 Testy przygotowanego API w porównaniu z innymi gotowymi rozwiązaniami

4.6 Ocena ergonomi przygotowanego rozwiązania

Bibliografia

[1] Nikos Athanasiou, http://stackoverflow.com/questions/2808398/easily-measure-elapsed-time

[2] MDN, https://developer.mozilla.org/pl/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[3] http://itcraftsman.pl/zrozumiec-linq/

[4] http://www.nurkiewicz.com/2014/07/introduction-to-writing-custom.html