

# Podójście funkcyjnej do obsługi kolekcji w języku C++

Krzysztof Osiecki

30 marca 2017

# Spis treści

<b>1</b>	<b>Omówienie funkcyjnego paradygmatu programowania</b>	<b>3</b>
1.1	Czym jest paradygmat programowania . . . . .	3
1.2	Cechy paradygmatu funkcyjnego . . . . .	3
1.3	Efekty uboczne, w programowaniu funkcyjnym . . . . .	4
1.4	Dlaczego paradygmat funkcyjny? . . . . .	4
1.5	Wyrażenia lambda i referencje metod jako sposób realizacji pa- radygmatu . . . . .	5
1.6	Realizacja paradygmatu w języku Java . . . . .	5
1.7	Realizacja paradygmatu w języku C# . . . . .	5
1.8	Realizacja paradygmatu w języku JavaScript . . . . .	5
1.9	Realizacja paradygmatu w języku C++ . . . . .	5
1.10	Kolekcje jako strumień danych . . . . .	5
<b>2</b>	<b>Zalety programowania funkcyjnego w rozwiązaniach praktycz- nych</b>	<b>6</b>
2.1	Podstawowe problemy i zadania w programowaniu aplikacji biz- nesowych . . . . .	6
2.2	Porównanie różnic podejścia funkcyjnego i imperatywnego w roz- wiązywaniu problemów . . . . .	7
2.3	Analiza przykładowych rozwiązań w przypadku różnych paradyg- matów . . . . .	9
<b>3</b>	<b>Analiza wydajności istniejących rozwiązań funkcyjnych</b>	<b>11</b>
3.1	Prezentacja zakresu testów i opis środowiska . . . . .	11
3.2	Kryteria wyboru rozwiązań wraz z przedstawieniem wybranych .	11
3.3	Przedstawienie i analiza wyników . . . . .	11
<b>4</b>	<b>Prezentacja opracowanego rozwiązania</b>	<b>12</b>
4.1	Kod źródłowy . . . . .	12
4.2	Opis zakresu możliwości przygotowanego rozwiązania . . . . .	16
4.3	Porównanie przygotowanego API z istniejącymi w innych językach	16
4.4	Testy weryfikujące wydajność rozwiązania względem natywnego języka C++ . . . . .	16
4.5	Testy przygotowanego API w porównaniu z innymi gotowymi roz- wiązaniami . . . . .	17
4.6	Ocena ergonomii przygotowanego rozwiązania . . . . .	17

# 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 wstępie 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. MORE... W paradygmacie deklaratywnym z kolei 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. MORE... 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 tę 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 bezstanowość. Każda funkcja matematyczna posiada swoją definicję, która określa jej wynik w zależności od zadanych argumentów. Dla określonego zbioru argumentów funkcja zawsze zwróci ustaloną wartość. 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. Sprawily 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 oczekwanego 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, które w momencie jej tworzenia mogły być nie dostępne. Daje to w ręce programisty naprawdę 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 Wyrażenia lambda i referencje metod jako sposób realizacji paradygmatu
- 1.6 Realizacja paradygmatu w języku Java
- 1.7 Realizacja paradygmatu w języku C#
- 1.8 Realizacja paradygmatu w języku JavaScript
- 1.9 Realizacja paradygmatu w języku C++
- 1.10 Kolekcje jako strumień danych

## 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 chcąc 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 kosztu 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 za trzy tysiące złotych, z radością kupi procesor i przypisze sobie zasługi oszczędzenia dwóch tysięcy. Oczywiście gdyby tego typu problemy były częste, należałoby 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 funkcyjnego 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 **getter** i **setter**, 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             ;
8             people.add(person);
9         }
10    }
11    return people;
12 }
```

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             )
5         .filter(user -> user.getRole() == UserRole.GUEST)
6         .map(user -> new Person(user.getName(), user.getLastName()))
7         .collect(toList());
8 }
```

Choć przykład jest prosty i niewiele zyskujemy w kontekście ilości linii kodu. To już tutaj da się dotrzeć 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 linii 5 przykładu znajduje się mapowanie obiektów `User` na odpowiadające im `Person`. Ostatnia 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 `if` 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 kolejną instrukcję `filter`, a kolejne mapowanie (na przykład na ciągi znaków z imieniem) to instrukcja `map`. Dla podejścia imperatywnego, albo rozbudowujemy warunek `if` albo zagnieżdżamy jednego w drugim, co nie jest zachowaniem porządnym. 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(
3         groupingBy(User::getRole));
4 }
```

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 jakiejś 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 rolę z obiektu użytkownika.

Funkcyjne rozwiązanie problemu jest zdecydowanie prostsze. Mniej kodu do napisania i mniej analizy tego co należy napisać, oznacza mniej możliwości 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     public:
23         stream();
24
25         stream(const std::vector<T> &data);
26
27         bool allMatch();
28
29         bool anyMatches();
30
31         T find();
32
33         stream<T> *filter(std::function<bool(T)> predicate);
34
35         std::vector<T> *toVector();
36
37         template<class R>
```

```

39         stream<R> *map(std::function<R(T)> mappingFunction);
40
41         stream<T> *peek();
42
43     protected:
44         stream(const std::vector<T> &data, std::vector<
45             streamOperation<bool(T)> *> *predicates);
46
47         bool executePredicateOperation(streamOperation<bool(T)> *
48             operation, T value);
49
50         template<class R>
51         R executeMappingOperation(streamOperation<R(T)> *operation,
52             T value);
53
54         void checkConsumed(bool consume);
55
56     private:
57         std::vector<streamOperation<bool(T)> *> *predicates;
58         std::vector<T> *underlyingVector;
59         bool consumed;
60
61 };
62
63 template<class T, class... Args>
64 class streamOperation<T(Args...)> {
65     public:
66         std::function<T(Args...)> *fun;
67
68         streamOperation(std::function<T(Args...)> *fun) {
69             this->fun = fun;
70         }
71 };
72
73 template<class T>
74 stream<T>::stream(const std::vector<T> &data, std::vector<
75     streamOperation<bool(T)> *> *predicates) {
76     this->underlyingVector = new std::vector<T>(data);
77     this->predicates = predicates;
78     this->consumed = false;
79 }
80
81 template<class T>
82 stream<T>::stream(const std::vector<T> &data) {
83     this->underlyingVector = new std::vector<T>(data);
84     this->predicates = new std::vector<streamOperation<bool(T)>
85         *>();
86     this->consumed = false;
87 }
88
89 template<class T>
90 stream<T>::stream() {
91     this->underlyingVector = new std::vector<T>();
92     this->predicates = new std::vector<streamOperation<bool(T)>
93         *>();
94     this->consumed = false;
95 }
96
97 template<class T>
98 stream<T> *stream<T>::filter(std::function<bool(T)> predicate)
99 {
100     checkConsumed(false);

```

```

94         auto *op = new streamOperation<bool(T)>(&predicate);
95         predicates->push_back(op);
96         return this;
97     }
98
99     template<class T>
100     T stream<T>::find() {
101         std::vector<T> *pVector = this->toVector();
102         return pVector->empty() ? NULL : pVector->front();
103     }
104
105     template<class T>
106     bool stream<T>::anyMatches() {
107         return !toVector()->empty();
108     }
109
110     template<class T>
111     bool stream<T>::allMatch() {
112         unsigned int fullSize = underlyingVector->size();
113         return toVector()->size() == fullSize;
114     }
115
116     template<class T>
117     template<class R>
118     stream<R> *stream<T>::map(std::function<R(T)> mappingFunction)
119     {
120         streamOperation<R(T)> *op = new streamOperation<R(T)>(&
121             mappingFunction);
122         std::vector<T> *filtered = toVector();
123         std::vector<R> *result = new std::vector<R>();
124         for (typename std::vector<T>::iterator it = filtered->begin
125             ();
126             it != underlyingVector->end(); ) {
127             T v = (*it);
128             auto val = this->executeMappingOperation(op, v);
129             result->push_back(val);
130             ++it;
131         }
132         return new stream<R>(*result);
133     }
134
135     template<class T>
136     std::vector<T> *stream<T>::toVector() {
137         checkConsumed(true);
138         std::vector<T> *result = new std::vector<T>();
139         for (auto it = underlyingVector->begin(); it !=
140             underlyingVector->end(); ++it) {
141             auto remove = false;
142             T v = (*it);
143             for (auto oIt = predicates->begin(); oIt != predicates
144                 ->end(); ++oIt) {
145                 auto val = ((*oIt)->fun)(v);
146                 if (!val) {
147                     remove = true;
148                     break;
149                 }
150             }
151             if (!remove) result->push_back(v);
152         }
153         return result;
154     }

```

```

151
152     template<class T>
153     stream<T> *stream<T>::peek() {
154         for (auto it = underlyingVector->begin(); it !=
            underlyingVector->end(); ++it) {
155             auto remove = false;
156             T v = (*it);
157             for (auto oIt = predicates->begin();
158                 oIt != predicates->end(); ++oIt) {
159                 auto val = ((*oIt)->fun)(v);
160                 if (!val) {
161                     remove = true;
162                     break;
163                 }
164             }
165             if (!remove) std::cout << v << " ";
166         }
167         std::cout << std::endl;
168         return this;
169     }
170
171     template<class T>
172     bool stream<T>::executePredicateOperation(streamOperation<bool(
        T)> *operation,
173                                              T value) {
174         auto function = (*operation->fun);
175         return function(value);
176     }
177
178     template<class T>
179     template<class R>
180     R stream<T>::executeMappingOperation(streamOperation<R(T)> *
        operation,
181                                         T value) {
182         auto function = (*operation->fun);
183         return function(value);
184     }
185
186     template<class T>
187     void stream<T>::checkConsumed(bool consume) {
188         if (this->consumed) throw new
            streamAlreadyConsumedException();
189         this->consumed = consume;
190     }
191 }
192
193 #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 kryterium były usuwane z wektora za pomocą funkcji `vector::erase`. Podejście czwarte to iteracja za pomocą pętli wykorzystującej zmienną całkowitą z dostępem do elementu wektora 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 lambda. 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

Tablica 4.1: Pojedyncze filtrowanie, 10000 elementów

1	2	3	4	5	6	7
3.66	0.07	29.11	0.04	0.68	30.99	0.72

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 ergonomii przygotowanego rozwiązania

# Bibliografia

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