Podejście funkcyjne do obsługi kolekcji w języku C++

Krzysztof Osiecki

11 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 utoższamiane 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. 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 odpowienie 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 pratyce 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 bezstanowość. 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 Realizacja paradygmatu w języku JavaScript

1.7 Realizacja paradygmatu w języku C#

1.8 Realizacja paradygmatu w języku Java

1.9 Realizacja paradygmatu w języku C++

1.10 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.

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         std::vector<T> *toVector();
37
38         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<
                streamOperation<bool(T)> *> *predicates);
45
46          template<class R>
47          R executeMappingOperation(streamOperation<R(T)> *operation,
                T value);
48
49          void checkConsumed(bool consume);
50
51      private:
52          std::vector<streamOperation<bool(T)> *> *predicates;
53          std::vector<T> *underlyingVector;
54          bool consumed;
55
56      };
57
58      template<class T, class... Args>
59      class streamOperation<T(Args...)> {
60      public:
61          std::function<T(Args ...)> *fun;
62
63          streamOperation(std::function<T(Args ...)> *fun) {
64              this->fun = fun;
65          }
66      };
67
68      template<class T>
69      stream<T>::stream(const std::vector<T> &data, std::vector<
            streamOperation<bool(T)> *> *predicates) {
70          this->underlyingVector = new std::vector<T>(data);
71          this->predicates = predicates;
72          this->consumed = false;
73      }
74
75      template<class T>
76      stream<T>::stream(const std::vector<T> &data) {
77          this->underlyingVector = new std::vector<T>(data);
78          this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
79          this->consumed = false;
80      }
81
82      template<class T>
83      stream<T>::stream() {
84          this->underlyingVector = new std::vector<T>();
85          this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
86          this->consumed = false;
87      }
88
89      template<class T>
90      stream<T> *stream<T>::filter(std::function<bool(T)> predicate)
            {
91          checkConsumed(false);
92          auto *op = new streamOperation<bool(T)>(&predicate);
93          predicates->push_back(op);
94          return this;
```

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

```
152     stream<T> *stream<T>::peek() {
153         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
154             auto remove = false;
155             T v = (*it);
156             for (auto oIt = predicates->begin();
157                  oIt != predicates->end(); ++oIt) {
158                 auto val = (*(*oIt)->fun)(v);;
159                 if (!val) {
160                     remove = true;
161                     break;
162                 }
163             }
164             if (!remove) std::cout << v << " ";
165         }
166         std::cout << std::endl;
167         return this;
168     }
169
170     template<class T>
171     template<class R>
172     R stream<T>::executeMappingOperation(streamOperation<R(T)> *
            operation,
173                                          T value) {
174         auto function = (*operation->fun);
175         return function(value);
176     }
177
178     template<class T>
179     void stream<T>::checkConsumed(bool consume) {
180         if (this->consumed) throw new
                streamAlreadyConsumedException();
181         this->consumed = consume;
182     }
183 }
184
185 #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>
```

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

```
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 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 4.2.

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

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