UMCS

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **Informatyka**
Specjalność: **Informatyka medyczna**

Krzysztof Osiecki
Nr albumu: 250480

API funkcyjne w języku C++
(Functional reactive programming in C++)

Praca magisterska
napisana w Zakładzie Technologii Informatycznych
pod kierunkiem prof. dr. hab. Pawła Mikołajczaka

LUBLIN ROK 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. Niniejsza praca przedstawia ogólny podział języków programowania, ze względu na podejście do tworzenia kodu, skupiając się głównie na podejściu funkcyjnym. W pierwszym rozdziale zostaną przedstawione ogólne koncepcje, które pozwalają na funkcyjne tworzenie kodu niezależnie od języka. Rozdział drugi ma na celu porównanie dwóch głównych paradygmatów programowania oraz prezentacje zalet paradygmatu funkcyjnego. W rozdziale trzecim, opisane zostały gotowe rozwiązania, zaimplementowane w różnych językach pozwalające, na funkcyjną obsługę kolekcji, oraz posługiwania się funkcjami jako obiektami domknięcia. Ostatni rozdział przedstawia przygotowane API. API z języka angielskiego `Aplication Programing Interface`, czyli interfejs programowania aplikacji, oznacza zbiór funkcji oferowanych w tym przypadku w postaci biblioteki do języka `C++`. Przygotowane rozwiązanie ma umożliwić funkcyjną obsługę kolekcji 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.

Definicja 1. *Paradygmat - przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie.*

Paradygmat nie jest określany przez język programowania, lecz przez sposób w jaki programista opisuje problem, który rozwiązuje. Paradygmaty w dziedzinie programowania 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. Najpopularniejszymi językami akademickimi są `C++` oraz `Pascal`, które ze swej natury są imperatywne. 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. Znanym powszechnie przykładem tego podejścia może być język `SQL`. Zapytanie w tym języku polega na opisaniu, oczekiwanego wyniku. Na tej podstawie silnik bazy danych, wykonuje odpowiednie operacje. Piszący kwerendę nie musi wiedzieć w jakiej strukturze znajdują się dane, ani jak efektywnie się do nich dostać.

W praktyce rzadko spotyka się języki w całości oparte tylko na jednym paradygmacie. Spowodowane jest to ograniczeniami, jakie musiałby być realizowane w celu pełnego spełnienia założeń, każdego z nich. Najczęściej okazuje się, że idealnym rozwiązaniem jest połączenie najlepszych cech każdego z nich. Obecnie większość popularnych języków programowania, jest językami wieloparadygmatowymi. Posiadają one zarówno cechy imperatywne jak i deklaratywne. Do tego wewnętrzny podział tych głównych grup powoduje jeszcze większe rozdrobnienie kategorii danego języka. Określenie paradygmat często występuje wymiennie ze słowem 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. 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.

1.3 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 potraktowa-

nie 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.

1.4 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 2. *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.

1.5 Kolekcje jako strumienie danych

Wszystkie kolekcje, można przy przyjęciu odpowiedniego poziomu abstrakcji traktować jako 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 podrozdziałach przestawiono, sposoby implementacji tych rozwiązań.

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 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 `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, język Java

```
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, język Java z użyciem strumieni

```
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 dostrzec pewne zyski. Wystarczy spróbować dokonać analizy logicznej tego co próbujemy uzyskać. Otrzymujemy kolekcję użytkowników, przyjmijmy, że klasa `User` jest klasą przedstawiającą obiekt bazodanowy. Chcemy przefiltrować użytkowników urodzonych po 1. grudnia 1999 roku, których rola w systemie to `GUEST`. Na podstawie przefiltrowanej 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`. 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 instrukcji warunkowej 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 albo zagnieżdżamy instrukcje warunkowe, co nie jest zachowaniem pożądanym. 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: Grupowanie. 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: Grupowanie. 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 możliwości popełnienia błędu. W rozwiązaniu funkcyjnym nie ma w zasadzie miejsca w którym można się pomylić. Kolejny przykład przedstawia operacje obliczenia średniej z elementów, w posiadanej kolekcji. Tym razem rozwiązanie przedstawione zostało w języku C#.

Listing 2.5: Redukcja. Podejście funkcyjne

```
1        public double getAvg(ICollection<double> numbers)
2        {
3         return numbers.Aggregate((a, b) => a + b) / numbers.Count;
4        }
```

Listing 2.6: Redukcja. Podejście funkcyjne

```
1        public double getAvg(ICollection<double> numbers)
2        {
3         double sum = 0;
4         foreach(double number in numbers)
5         {
6          sum += number;
7         }
8         return sum / numbers.Count;
9        }
```

Tak jak w poprzednim przypadku, pomimo olbrzymiej prostoty zadania, rozwiązanie funkcyjne jest w stanie znacznie skrócić zapis. Ponad to rozwiązanie funkcyjne zwalnia programistę z obowiązku tworzenia akumulatora. Przykład funkcyjny w języku C# mógłby być w zasadzie jeszcze, krótszy ze względu na ogromną ilość rozszerzeń kolekcji oferowanych w standardzie. Zamiast stosować funkcję `Aggregate`, z lambdą odpowiedzialną za dodawanie wartości, można było zastosować funkcję `Average`, która wykona za nas wszystkie operacje, czyli zarówno sumę jak i obliczenie średniej. Tak jak w poprzednim przypadku, najważniejszą zaletą metody funkcyjnej jest jej zwięzłość. Deweloperzy pracujący w projektach biznesowych muszą przedzierać się przez tysiące linii kodu, każdego dnia. Ważne jest aby konieczny do przejrzenia kod był jak najbardziej zwięzły i co jeszcze ważniejsze użyteczny. Kod który nic nie wnosi nie jest nawet elementem neutralnym, ponieważ zajmuje miejsce. Będąc w stanie wyliczyć średnią wartość w jednej linii zamiast w sześciu zyskujemy 5 linii więcej na ekranie. Jest

to całkiem spora liczba biorąc pod uwagę, że uruchomione w standardowej konfiguracji Visual Studio, wyświetla na raz tylko 40 linii, oznacza to, że ta prosta metoda oszczędza 12.5% całości kodu widocznego na ekranie.

Rozdział 3

Realizacje w różnych językach

3.1 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 działania); `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`, będzie 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 3.1, 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 praktycznie 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 3.1: 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 ECMAScript. 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 3.2.

Listing 3.2: Funkcje strzałkowe [1]

```
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.

3.2 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 dodatkowy 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.

3.3 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 [2]. 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 3.3, zaprezentowano prosty przykład obu składni.

Listing 3.3: 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 `Aggregate`, 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 3.4.

Listing 3.4: 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.

3.4 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 3.5.

Listing 3.5: 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.

3.5 Realizacja paradygmatu w języku C++

Język `C++` do czasu wersji `C++11`, określonej standardem ISO/IEC 14882:2011, nie oferował zbyt przyjaznego API, do funkcyjnego programowania. Choć tak zwane wskaźniki na funkcje były dostępne właściwie od początku istnienia języka, to jednak ciężko powiedzieć, aby były wygodnym rozwiązaniem. O ile standard C++03 był w zasadzie wersją skupioną na poprawkach błędów nie zaś na wprowadzaniu funkcjonalności do języka, wersja C++11 zdecydowanie stawiała na jego rozbudowę. Twórcy standardu zdecydowali się pójść w kierunku,

ułatwienia tworzenia większych biznesowych aplikacji, a także ujednolicenia elementów języka które mogłyby wprowadzać niejasności dla programistów niezaznajomionych ze specyfiką języka. Wprowadzono na przykład znane z innych języków pętlę `foreach`, dotychczas istniały tylko dwie opcje, pętla z wartownikiem, lub w przypadku korzystania z biblioteki `STL`, pętla z użyciem iteratora. Konstrukcja ta w języku `C++` została nazwana `range-based for` i przedstawiona jest na listingu 3.6.

Listing 3.6: Pętla foreach w C++

```
1 int liczby[5] = {1, 2, 3, 4, 5};
2 for (int& liczba : liczby) {
3  liczba *= 2;
4 }
```

Innym kluczowym w kontekście ułatwienia tworzenia kodu dodatkiem jest wprowadzenie `delegacji` konstruktorów. Pozwala to na wywoływanie innego konstruktora przy tworzeniu obiektu, co pozwala znacznie zmniejszyć ilość linii kodu wymaganych do powtórzenia. Ilość zmian jest na prawdę ogromna, wprowadzenie krotek, biblioteka do wyrażeń regularnych, tablice haszujące i wiele usprawnień składni. Najważniejsze w kontekście tej pracy są jednak zmiany ułatwiające programowanie funkcyjne.

Pierwszym taką zmianą jest wprowadzenie słowa kluczowego `auto`, `auto` oznacza typ automatycznie wydedukowany przez kompilator. Zmienne definiowane przy użyciu tego słowa kluczowego muszą być natychmiast inicjowane w celu zapewnienia możliwości ustalenia właściwego typu. W dalszej części pracy będzie można zobaczyć przykład pokazujący, że dedukcja ta nie jest jeszcze doskonała i może powodować problemy w trakcie działania programu. Słowo to jednak znacznie ułatwia programowanie funkcyjne, ponieważ zwalnia programistę z obowiązku znajomości typów pośrednich zwracanych przez użyte metody. Pracując przy użyciu funkcji i przekazując je jako parametry nie jest konieczna wiedza o tym jakiego typu będzie zwracany obiekt, jeżeli jedyna jego funkcja polega na byciu parametrem innego wywołania. Zakładając poprawną dedukcje typu i wykorzystanie wyrażeń lambda `C++` oferowałby pełną możliwość stworzenia funkcyjnego API. Pomijając nawet fakt, że nie jest ono oferowane przez standard języka, możliwe byłoby jego utworzenie.

Najważniejszym dodatkiem są jednak właśnie wyrażenia lambda. Posiadają one w języku `C++` zdecydowanie największe możliwości ze wszystkich prezentowanych dotychczas możliwości. Kosztem tych opcji jest oczywiście wzrost skomplikowania ich deklaracji. Forma wyrażenie lambda przedstawiona jest na listingu 3.7

Listing 3.7: Składania wyrażeń lambda w języku C++

```
1 [ przechwytywane_zmienne ] ( parametry ) mutable(opcjonalne slowo
    kluczowe) exception attribute -> typ_zwracany { cialo_funkcji
    }
2 [ przechwytywane_zmienne ]( parametry ) -> typ_zwracany {
    cialo_funkcji }
3 [ przechwytywane_zmienne ]( parametry ) { cialo_funkcji }
4 [ przechwytywane_zmienne ] { cialo_funkcji }
```

Pierwsza forma to pełna deklaracja. Zawierająca specyfikacje wyjątków i atrybutów dla wywołania wyrażenia. Druga jest formą `stałej` lambdy, w której obiekty przechwycone przez wartość nie mogą być modyfikowane. W przypadku pierwszej taką możliwość daje słowo kluczowe `mutable`. W odróżnieniu od innych języków, typ przyjmowanych parametrów musi być zawsze zadeklarowany. Trzeci wariant umożliwia opuszczenie typu zwracanego, jeśli `cialo_funkcji` zawiera tylko jedną instrukcje typu `return` to typ zwracany będzie zgodny z typem wartości tej instrukcji. Typ będzie w tym wypadku wydedukowany w taki sam sposób jak gdyby był określony jako `auto`. Jeśli funkcja zawiera więcej instrukcji typem będzie `void`. Choć specyfikacja określa, że zawsze jeśli jest instrukcji jest więcej niż jedna typem powinien być `void`, to kompilatory dedukują typ lepiej. Dlatego poprawnie kompiluje się także kod z wieloma instrukcjami `return`, o ile typ wydedukowany z każdej z tych instrukcji jest zgodny. Wersja ostania to lambda nie przyjmująca, żadnego parametru. Największą różnicę pomiędzy wyrażeniami lambda w `C++` a Javą czy C#, stanowi pierwszy, człon każdego z wyrażeń, czyli lista przechwytywanych zmiennych. Lista ta nazwana jest po prostu `capture-list` i pozwala określić jakie zmienne i w jaki sposób będą dostępne wewnątrz lambdy. Można wskazać wprost zmienną do której potrzebujemy dostępu w lambdzie. Wtedy po prostu wpisujemy do listy nazwę tej zmiennej. Jest ona wtedy przekazana `przez wartość`, czyli jest efektywnie kopią zmiennej. Jeśli poprzedzimy nazwę znakiem `&`, zmienna zostanie przekazana przez referencję. Przekazać można także obiekt, w którym tworzona jest lambda poprzez umieszczenie na liście słowa kluczowego `this`. Jeśli chcemy przechwycić wszystkie zmienne z zakresu, możemy użyć znaków `&` i `=` do przechwycenia wszystkich użytych w ciele lambdy zmiennych odpowiednio przez referencje i wartość. Warto dodać że obiekt `this` niezależnie od wybranej metody zawsze zostanie przechwycony przez referencję. Poszczególne wartości należy oddzielać przecinkami. Wskazanie bezpośrednie zmiennej jest zawsze ważniejsze niż znaki przechwytujące wszystkie, zaś pozostawienie pustej listy, sprawia, że lambda nie będzie miała dostępu do żadnej zmiennej kontekstowej. Przykłady ilustrujące powyższy opis przedstawiono na listingu 3.8

Listing 3.8: Listy przechwytywania w języku C++

```
1  [a,b,c] - a, b i c przechwycone przez wartosc
2  [a,&b,c] - a i c przechwycone przez wartosc, b przez referencje
3  [this,&a] - obiekt this i a przechwycone przez referencje
4  [&, b] - b przechwycone przez wartosc, wszystkie pozostale
       zmienne przez referencje
5  [=, &a] - a i obiekt this przechwycone przez referencje,
       wszystkie pozostale zmienne przez wartosc
6  [] - zadna zmienna nie bedzie przechwycona
7  [&] - wszytkie zmienne przechwycone przez referencje
8  [=] - obiekt this przechwycony przez referencje, wszystkie
       pozostale zmienne przez wartosc
```

Listy przechwytywania wprowadzają dużo większe możliwości sterowania widocznością zmiennych niż oferowane przez Javę czy C# rozwiązania, jednak czyni to kosztem większego skomplikowania konstrukcji.

Zaprezentowane funkcjonalności pozwalają już na pseudo-funkcyjne przetwarzanie kolekcji. Biblioteka `algorithm` oferuje metody pokrywające te określone wcześniej jako niezbędne, do tego celu poza `map`. Jednak metody z tej biblioteki

trzymają się ustalonej w bibliotekach `STL` konwencji korzystania z iteratorów. Funkcje będą zatem przyjmować iterator wskazujący na początek przeszukiwanej sekwencji, iterator wskazujący koniec, oraz wykonywaną operację. Wartością zwracaną będzie najczęściej kolejny iterator wskazujący na pierwszy element wyniku. Choć tego typu operacje są bardzo użyteczne, nie wpisują się one zbyt dobrze w koncepcje programowania obiektowego i wykonania operacji na obiekcie kolekcji. Dlatego właśnie narodził się pomysł na stworzenie API w języku C++ oferującego styl programowania dostępny w Javie czy C#.

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  #include <forward_list>
9  #include <list>
10
11 namespace stream {
12
13     class streamAlreadyConsumedException {
14     };
15
16     template<class T>
17     class stream;
18
19     template<class T, class... Args>
20     class streamOperation;
21
22     template<class T>
23     class stream {
24
25     public:
26
27         /**
28          * Konstruktor tworzacy strumien z instancji std::vector
29          *
30          * @param data wektor ktory zostanie opakowany strumieniem
31          */
32         stream(const std::vector<T> &data);
33
34         /**
35          * Konstruktor tworzacy strumien z instancji std::deque
36          *
37          * @param data wektor ktory zostanie opakowany strumieniem
38          */
```

```
39          stream(const std::deque<T> &data);
40
41          /**
42           * Konstruktor tworzacy strumien z instancji std::
                 forward_list
43           *
44           * @param data wektor ktory zostanie opakowany strumieniem
45           */
46          stream(const std::forward_list<T> &data);
47
48          /**
49           * Konstruktor tworzacy strumien z instancji std::list
50           *
51           * @param data wektor ktory zostanie opakowany strumieniem
52           */
53          stream(const std::list<T> &data);
54
55          /**
56           * Konstruktor tworzacy strumien z instancji std::vector, w
                 odroznieniu
57           * od drugiego kostruktora ten nie tworzy kopii przyjetych
                 danych i moze je modyfikowac
58           *
59           * @param data wektor ktory zostanie opakowany strumieniem
60          */
61          stream(std::vector<T> *data);
62
63          /**
64           * Konstruktor tworzacy strumien z instancji tablicy
65           *
66           * @param data wektor ktory zostanie opakowany strumieniem
67           */
68          stream(T data[], int length);
69
70          /**
71           * Operacja nakladajaca na strumien filtr okreslony zadana
                 funkcja.
72           * Operacja nieterminalna
73           *
74           * @param predicate predykat okreslajacy warunek konieczny
                 do znalezienia sie w wynikowym strumieniu
75           * @return strumien z zaaplikowana funkcja filtrujaca
76           */
77          stream<T> *filter(std::function<bool(T)> predicate);
78
79          /**
80           * Operacja konwerujaca strumien, do typu wskazanego przez
                 funkcje mapujaca, poprzez
81           * zaaplikowanie jej do kazdego elementu strumienia.
82           * Operacja terminalna
83           *
84           * @tparam R typ nowego strumienia
85           * @param mappingFunction funkcja mapujaca
86           * @return nowy strumien z elementami bedacymi wynikiem
                 funkcji mapujacej
87           */
88          template<class R>
89          stream<R> *map(std::function<R(T)> mappingFunction);
90
91          /**
92           * Operacja redukcji strumienia. Wskazana funkcja bedzie
                 wywolywana na
```

```
93          * elementach strumienia, w postaci fun(poprzedni_wynik,
                aktualny_element),
94          * az do wyczerpania elementow.
95          * Operacja terminalna
96          *
97          * @param reductorFunction funkcja odpowiadajaca za
                redukcje elementow
98          * @return zredukowana wartosc ostatniego wykonania funkcji
99          */
100         T reduce(std::function<T(T, T)> reductorFunction);
101
102         /**
103          * Operacja aplikuje zadana funkcje do kazdego elementu
                strumienia
104          * Operacja terminalna
105          *
106          * @param exectutionFunction funkcja do wykonania na kazdym
                 elemencie
107          */
108         void foreach(std::function<void(T)> exectutionFunction);
109
110         /**
111          * Operacja okreslajaca czy wszystkie elementy danego
                strumienia spelniaja
112          * zaaplikowane do niego predykaty operacji filter.
113          * Operacja terminalna
114          *
115          * @return true jesli operacje filter nie wykluczyly
                zadnego z elementow strumienia
116          *         false w pozostalych przypadkach
117          */
118         bool allMatch();
119
120         /**
121          * Operacja okreslajaca czy choc jeden element danego
                strumienia spelnia
122          * zaaplikowane do niego predykaty operacji filter.
123          * Operacja terminalna
124          *
125          * @return true jesli operacje filter pozostawily choc
                jeden element strumienia
126          *         false w pozostalych przypadkach
127          */
128         bool anyMatches();
129
130         /**
131          * Operacja znajdujaca pierwszy element strumienia.
132          * Operacja terminalna
133          *
134          * @return znaleziony element, NULL dla pustego strumienia
135          */
136         T find();
137
138         /**
139          * Operacja uzytkowa, przechodzi strumien wypisujac jego
                zawartosc
140          * na standardowe wyjscie.
141          * Operacja nieterminalna
142          *
143          * @return ten sam strumien wejsciowy
144          */
145         stream<T> *peek();
```

```
146
147         /**
148          * Operacja powrotu ze strumienia do std::vector.
                Aplikowane sa wszsytkie
149          * operacje filter i zwracany nowy obiekt vectora.
150          * Operacja terminalna
151          *
152          * @return std::vector zawierajacy elementy strumienia
153          */
154         std::vector<T> *toVector();
155
156         /**
157          * Operacja powrotu ze strumienia do std::deque. Aplikowane
                sa wszsytkie
158          * operacje filter i zwracany nowy obiekt deque.
159          * Operacja terminalna
160          *
161          * @return std::deque zawierajacy elementy strumienia
162          */
163         std::deque<T> *toDeque();
164
165         /**
166          * Operacja powrotu ze strumienia do std::forward_list.
                Aplikowane sa wszsytkie
167          * operacje filter i zwracany nowy obiekt forward_list.
168          * Operacja terminalna
169          *
170          * @return std::forward_list zawierajacy elementy
                strumienia
171          */
172         std::forward_list<T> *toForwardList();
173
174         /**
175          * Operacja powrotu ze strumienia do std::list. Aplikowane
                sa wszsytkie
176          * operacje filter i zwracany nowy obiekt listy.
177          * Operacja terminalna
178          *
179          * @return std::list zawierajacy elementy strumienia
180          */
181         std::list<T> *toList();
182
183         ~stream();
184
185     protected:
186
187         void checkConsumed(bool consume);
188
189         template<class R>
190         std::vector<R> *toVector(std::function<R(T)>
                mappingFunction);
191
192     private:
193         std::vector<streamOperation<bool(T)> *> *predicates;
194         std::vector<T> *underlyingVector;
195         bool consumed;
196     };
197
198     template<class T, class... Args>
199     class streamOperation<T(Args...)> {
200     public:
201         std::function<T(Args ...)> *fun;
```

```
202
203         ~streamOperation() {
204             delete (fun);
205         }
206
207         streamOperation(std::function<T(Args ...)> *fun) {
208             this->fun = fun;
209         }
210     };
211
212     template<class T>
213     stream<T>::stream(const std::vector<T> &data) {
214         this->underlyingVector = new std::vector<T>(data);
215         this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
216         this->consumed = false;
217     }
218
219     template<class T>
220     stream<T>::stream(std::vector<T> *data) {
221         this->underlyingVector = data;
222         this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
223         this->consumed = false;
224     }
225
226     template<class T>
227     stream<T>::stream(const std::deque<T> &data) {
228         this->underlyingVector = new std::vector<T>(data.begin(),
            data.end());
229         this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
230         this->consumed = false;
231     }
232
233     template<class T>
234     stream<T>::stream(const std::forward_list<T> &data) {
235         this->underlyingVector = new std::vector<T>(data.begin(),
            data.end());
236         this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
237         this->consumed = false;
238     }
239
240     template<class T>
241     stream<T>::stream(const std::list<T> &data) {
242         this->underlyingVector = new std::vector<T>(data.begin(),
            data.end());
243         this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
244         this->consumed = false;
245     }
246
247     template<class T>
248     stream<T>::stream(T data[], int length) {
249         this->underlyingVector = new std::vector<T>(data, data +
            length);
250         this->predicates = new std::vector<streamOperation<bool(T)>
                *>();
251         this->consumed = false;
252     }
253
```

```
254     template<class T>
255     stream<T> *stream<T>::filter(std::function<bool(T)> predicate)
            {
256         checkConsumed(false);
257         auto *op = new streamOperation<bool(T)>(&predicate);
258         predicates->push_back(op);
259         return this;
260     }
261
262     template<class T>
263     T stream<T>::find() {
264         std::vector<T> *pVector = this->toVector();
265         return pVector->empty() ? NULL : pVector->front();
266     }
267
268     template<class T>
269     bool stream<T>::anyMatches() {
270         return !toVector()->empty();
271     }
272
273     template<class T>
274     bool stream<T>::allMatch() {
275         unsigned int fullSize = underlyingVector->size();
276         return toVector()->size() == fullSize;
277     }
278
279     template<class T>
280     template<class R>
281     stream<R> *stream<T>::map(std::function<R(T)> mappingFunction)
            {
282         return new stream<R>(toVector(mappingFunction));
283     }
284
285     template<class T>
286     T stream<T>::reduce(std::function<T(T, T)> reductorFunction) {
287         std::vector<T> *filtered = toVector();
288         auto previousVal = filtered->front();
289         for (typename std::vector<T>::iterator it = filtered->begin
                ()  + 1; it != filtered->end(); ++it) {
290             previousVal = reductorFunction(previousVal, (*it));
291         }
292         return previousVal;
293     }
294
295     template<class T>
296     void stream<T>::foreach(std::function<void(T)>
            exectutionFunction) {
297         std::vector<T> *filtered = toVector();
298         for (typename std::vector<T>::iterator it = filtered->begin
                ();
299               it != filtered->end();) {
300             T v = (*it);
301             exectutionFunction(v);
302             ++it;
303         }
304     }
305
306     template<class T>
307     std::vector<T> *stream<T>::toVector() {
308         checkConsumed(true);
309         if (predicates->empty()) {
```

```
310             return new std::vector<T>(underlyingVector->begin(),
                    underlyingVector->end());
311         }
312         std::vector<T> *result = new std::vector<T>();
313         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
314             bool keep = true;
315             T v = (*it);
316             for (auto oIt = predicates->begin(); oIt != predicates
                    ->end(); ++oIt) {
317                 if (!(*(*oIt)->fun)(v)) {
318                     keep = false;
319                     break;
320                 }
321             }
322             if (keep)
323                 result->push_back(v);
324         }
325         return result;
326     }
327
328     template<class T>
329     template<class R>
330     std::vector<R> *stream<T>::toVector(std::function<R(T)>
            mappingFunction) {
331         checkConsumed(true);
332         std::vector<R> *result = new std::vector<R>();
333         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
334             bool keep = true;
335             T v = (*it);
336             for (auto oIt = predicates->begin(); oIt != predicates
                    ->end(); ++oIt) {
337                 if (!(*(*oIt)->fun)(v)) {
338                     keep = false;
339                     break;
340                 }
341             }
342             if (keep)
343                 result->push_back(mappingFunction(v));
344         }
345         return result;
346     }
347
348     template<class T>
349     std::deque<T> *stream<T>::toDeque() {
350         checkConsumed(true);
351         std::deque<T> *result = new std::deque<T>();
352
353         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
354             bool keep = true;
355             T v = (*it);
356             for (auto oIt = predicates->begin(); oIt != predicates
                    ->end(); ++oIt) {
357                 auto val = (*(*oIt)->fun)(v);;
358                 if (!val) {
359                     keep = false;
360                     break;
361                 }
362             }
363             if (keep) result->push_back(v);
```

```
364         }
365         return result;
366     }
367
368     template<class T>
369     std::forward_list<T> *stream<T>::toForwardList() {
370         checkConsumed(true);
371         std::forward_list<T> *result = new std::forward_list<T>();
372
373         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
374             bool keep = true;
375             T v = (*it);
376             for (auto oIt = predicates->begin(); oIt != predicates
                    ->end(); ++oIt) {
377                 auto val = (*(*oIt)->fun)(v);;
378                 if (!val) {
379                     keep = false;
380                     break;
381                 }
382             }
383             if (keep) result->push_front(v);
384         }
385         return result;
386     }
387
388     template<class T>
389     std::list<T> *stream<T>::toList() {
390         checkConsumed(true);
391         std::list<T> *result = new std::list<T>();
392
393         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
394             bool keep = true;
395             T v = (*it);
396             for (auto oIt = predicates->begin(); oIt != predicates
                    ->end(); ++oIt) {
397                 if (!(*(*oIt)->fun)(v)) {
398                     keep = false;
399                     break;
400                 }
401             }
402             if (keep) result->push_back(v);
403         }
404         return result;
405     }
406
407
408     template<class T>
409     stream<T> *stream<T>::peek() {
410         for (auto it = underlyingVector->begin(); it !=
                underlyingVector->end(); ++it) {
411             auto keep = true;
412             T v = (*it);
413             for (auto oIt = predicates->begin(); oIt != predicates
                    ->end(); ++oIt) {
414                 auto val = (*(*oIt)->fun)(v);;
415                 if (!val) {
416                     keep = false;
417                     break;
418                 }
419             }
```

```
420             if (keep) std::cout << v << " ";
421         }
422         std::cout << std::endl;
423         return this;
424     }
425
426     template<class T>
427     void stream<T>::checkConsumed(bool consume) {
428         if (this->consumed) throw new
                streamAlreadyConsumedException();
429         this->consumed = consume;
430     }
431
432     template<class T>
433     stream<T>::~stream() {
434         delete (underlyingVector);
435         for (auto &predicate : *predicates) delete (predicate);
436         delete (predicates);
437     }
438 }
439
440 #endif
```

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

4.2 Opis zakresu możliwości przygotowanego rozwiązania

Przedstawiony kod, zawiera się w jednym, łatwym do dystrybucji pliku `.hpp`, czyli standardowym dla języka `C++` formacie zawierającym, zarówno nagłówki metod jak i ich implementacje. Przygotowane rozwiązanie pozwala na utworzenie obiektu strumienia. Strumień może zostać przetworzony tylko raz. W przypadku, próby przetwarzania już wyczerpanego strumienia, zgłoszony zostanie wyjątek `streamAlreadyConsumedException`, nie wszystkie operacje powodują jednak wykorzystanie strumienia. Operacje które wyczerpują strumień nazwane będą operacjami terminalnymi, zaś pozostałe nieterminalnymi. Pokryte zostały wszystkie wspomniane w rozdziale trzecim metody. Dostępne są zatem instrukcja `filter`, przyjmująca jednoargumentową funkcję zwracającą wartość logiczną, predykat. Zaaplikowanie tej funkcji do strumienia, nie ma natychmiastowego efektu, wykonanie tej funkcji odłożone jest do momentu w który jest ono niezbędne. Dlatego można składać wiele funkcji `filter` bez obawy o niepotrzebne iterowanie strumienia. Operacja `filter` jest jedną z dwóch operacji nieterminalnych. `Map` jest operacją terminalną, zwracającą nowy nie przetworzony strumień, typu określonego przyjętą funkcją. Funkcja mapująca jest funkcją jednoargumentową z typu strumienia wejściowego w typ oczekiwanego strumienia wyjściowego. Następną operacją jest `reduce`, również jest to operacja terminalna, polega ona na wykonaniu funkcji na kolejnych elementach strumienia, w taki sposób, że każde kolejne wywołanie przyjętej funkcji jako argumenty przyjmuje wynik poprzedniego wykonania, oraz następny element strumienia. `Foreach` powoduje wykonanie funkcji na każdym z elementów strumienia, przekazana funkcja nie powinna zwracać wartości. Następnie mamy zestaw metod `allMath`, `anyMatches` i `find`, podobnie jak w języku Java operacje te wymagają

wcześniejszego przefiltrowania strumienia, same nie przyjmują żadnych parametrów. Wszystkie te operacje są operacjami terminalnymi. Drugą i zarazem ostatnią operacją nieterminalną jest `peek`, operacja ta służy tylko jako podpowiedź dla developera. Przechodzi ona strumień, nie konsumując go i wypisując każdy z jego elementów na standardowe wyjście.

Sposób przygotowania rozwiązania, umożliwia jego rozbudowę. Kod dostępny jest w całości w otwartym repozytorium w serwisie GitHub, zaś dystrybucja w postaci pojedynczego pliku `.hpp`, powoduje zarówno łatwość w użyciu jak i dodawaniu nowych funkcjonalności.

4.3 Porównanie przygotowanego API z istniejącymi w innych językach

Przygotowane API, zawiera metody umożliwiające efektywne programowanie z wykorzystaniem funkcyjnego przetwarzania kolekcji. Można z przy jego użyciu utworzyć strumień, wykorzystując do tego podstawowe kolekcje z biblioteki `STL`, takie jak `vector`, `deque`, `list`, `forward_list` oraz zwyczajną tablicę. Metody powrotu umożliwiają zebranie strumienia do, wybranego z wymienionych kontenerów `STL`. Jest to bardzo istotne w kontekście łatwości wprowadzenie rozwiązania do istniejących projektów. Ważne jest aby możliwość przejścia do korzystania z biblioteki nie wymagała zmian w kontraktach istniejących metod, lecz pozwalała na jej zastosowanie w ich wnętrzu. Jeśli chodzi więc o sposób wplatania użyć API do języka C++ to jest on podobny do realizacji w Javie. Tam przejście z kolekcji do strumienia zrealizowane jest poprzez dodanie metody z domyślną implementacją w interfejsie. C++ nie daje takiej możliwości, dlatego tutaj operacja przejścia wiąże się z wywołaniem konstruktora z opakowywaną kolekcją. W porównaniu z językami C# i JavaScript, rozwiązanie jest dużo bardziej oderwane od wewnętrznej struktury języka. Widać iż jest ono dodatkiem, a nie podstawowym sposobem tworzenia oprogramowania. Najważniejsze jest jednak zapewnienie płynnej opcji przejścia z obiektów `stream`, do powszechnie używanych kontenerów `STL`

API nie oferuje tak dużej ilości metod pomocniczych, jak te dostępne w innych językach, jednak zestaw funkcjonalności pozwala na realizowanie większej części z nich. Celem było zapewnienie właśnie możliwości tworzenia programów przy zastosowaniu określonego podejścia. Najważniejsze operacje to `filter` i `map`. Ich istnienie zapewnia możliwość realizacji wszystkich prostych operacji, które w poprzednich rozdziałach określane były jako `boilerplate`. Podsumowując rozwiązanie nie jest tak bardzo dopracowane jak gotowe przygotowane przez twórców języków programowania, co nie powinno specjalnie dziwić. Mimo swoich niedoskonałości stanowi bardzo dobry punkt wyjścia, który umożliwia zarówno użytkowanie w realnych sytuacjach jak i zapewnia duże możliwości rozwoju.

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[3]

```
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 };
```

Przetwarzane kolekcje zawierały milion elementów. Dla wartości mniejszej wszystkie wykonania trwały mniej niż 1ms, dlatego nie przedstawiają, żadnej wartości. 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. Kompilacja odbywała się przy zastosowaniu dwóch flag, `std=c++11`, włączającej funkcjonalności wymaganego standardu języka, oraz włączonymi opcjami optymalizacji poprzez flagę `-O3` [4]. Wykorzystano pięć różnych podejść w celu sprawdzenia różnic. Początkowo testom miało podlegać także, podejście polegające na usuwaniu elementów z przetwarzanego wektora, specyfika implementacji tego kontenera sprawiła jednak, że wyniki były kilkadziesiąt razy gorsze od przedstawionych sposobów. Z tego powodu wyniki ograniczono tylko do pięciu metod. Tabele przedstawiające wyniki odnoszą się do numeracji kolejnych z nich.

1. Podejście pierwsze polegało na zastosowaniu `stream:stream` oraz operacji `filter` z użyciem wyrażenia lambda do wartościowania wyniku filtrowania.

2. Podejście drugie to iteracja wektora za pomocą pętli po iteratorach, z tworzeniem nowej listy i dodawaniem do niej elementów spełniających kryteria. Wartościowanie wyrażenia realizowane zwykłą instrukcją warunkową bez zastosowania wyrażenia lambda.

3. Podejście trzecie 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. Wartościowanie wyrażenia realizowane zwykłą instrukcją warunkową bez zastosowania wyrażenia lambda.

4. Analogiczne do podejścia drugiego, z tym, że wartościowanie warunku wykonano za pomocą wyrażenia lambda.

5. Analogiczne do podejścia trzeciego, z tym, że wartościowanie warunku wykonano za pomocą wyrażenia lambda.

Celem stosowania dwóch wariantów dla operacji bezpośrednio na wektorze 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 określają czas trwania operacji w w milisekundach. Wyniki testu filtrowania przedstawione są w tabeli 4.1. Pojedyncze filtrowanie polegało na ograniczeniu elementów do parzystych. Filtrowanie podwójne oznacza zastosowanie filtrowania pojedynczego, następnie zaś na wyniku wyfiltrowanie elementów podzielnych przez trzy. Ostatni wariant czyli filtrowanie potrójne to wykonanie filtrowania podwójnego, następnie na wyniku, ponownie filtrowania liczb parzystych, czyli w praktyce zwrócenie całego posiadanego zbioru.

Na pierwszy rzut oka widoczne są pewne prawidłowości. Po pierwsze zastosowanie pętli z wartownikiem jest wolniejsze od pętli z wykorzystaniem iteratora średnio o około 16%. Może to być zaskoczeniem biorąc pod uwagę jak popularna jest tego typu pętla. Drugim ważnym wnioskiem jest wysoki koszt wykonywania wyrażeń lambda. Warianty wykorzystujące funkcje anonimowe, były średnio o ponad 85% wolniejsze od swoich odpowiedników wykorzystujących zwykłą instrukcję warunkową. Jeżeli więc ktoś poszukuje ekstremalnej wydajności, przy czym dodać trzeba, że mowa tu o kilku milisekundach w skali miliona elementów, powinien trzymać się z daleka od wyrażeń lambda. Jeśli chodzi zaś o wydajność strumienia to przegrywa on z metodami `2` i `3` w wszystkich trzech testach, wynika to jednak z faktu konieczności użycia lambdy. Jeśli przeskalujemy wyniki strumienia przez wyliczony współczynnik wydajności lambdy, wyniki będą lepsze dla przypadków podwójnego i potrójnego filtrowania, ulegając tylko w filtrowaniu pojedynczym. W przypadku wariantów z `4` i `5` pojedyncze filtrowanie wypada gorzej, wariant `4` jest o mniej więcej 40% lepszy zaś `5` o około 15%. Całkowity czas wykonania jest jednak bardzo niewielki, w pierwszym przypadku różnica wynosi nieco ponad milisekundę, w drugim nieco ponad pół milisekundy. Zwiększanie ilości operacji filtrujących przechyla jednak wyniki na korzyść strumieni. Dla podwójnego filtra, strumień jest szybszy o około 5%, w przypadku użycia iteratora co jest w zasadzie różnicą w granicach błędu. Jednak już w przypadku pętli z wartownikiem, różnica wynosi prawie 22%. Dołożenie kolejnej operacji, zwiększa tylko tę różnicę. Dla filtrowania potrójnego mamy już odpowiednio 23% i 40% W celu sprawdzenia tempa wzrostu kosztów wykonania, test powtórzono dla dziesięciokrotnie większej liczby elementów, rezultat widoczny jest w tabeli 4.2. Wyniki były jeszcze bardziej korzystne dla strumieni, które jednak cały czas przegrywały, z pętlami bez użycia lambd. Wzrost przewagi strumieni w wielokrotnych filtrowaniach wynika wprost z faktu wzrostu znaczenia dodatkowych iteracji w stosunku do kosztu opakowania, wektora w strumień. Najważniejszym wnioskiem powtórzonego testu jest jednak fakt, mniej więcej liniowego wzrostu czasowego kosztu operacji. Część wartości wzrosła bardziej niż dziesięciokrotnie, część mniej, ale trend pokazuje, że można przyjąć liniową złożoność czasową

Tablica 4.1: Filtrowanie, 1 000 000 elementów

	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)
Pojedyncze filtrowania	4.581	1.963	2.073	3.246	4.002
Podwójne filtrowanie	5.700	3.206	3.927	5.975	6.942
Potrójne filtrowanie	7.214	4.474	5.178	8.884	10.109

Tablica 4.2: Filtrowanie, 10 000 000 elementów

	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)
Pojedyncze filtrowania	55.724	29.399	31.308	43.806	50.896
Podwójne filtrowanie	55.674	40.327	45.931	66.062	75.851
Potrójne filtrowanie	60.969	47.634	53.192	83.382	95.888

wykonania. Możliwa jest oczywiście inna implementacja rozwiązań wektorowych zwiększająca wydajność. Odbywałaby się jednak ona kosztem czytelności rozwiązania, oraz spadkiem elastyczności dodawania nowych warunków. Celem testu było pokazanie użycia w jak najbardziej zbliżonym stopniu. Czyli filtrowany jest strumień, następnie filtrowany wynik, tak samo w przypadku wektorów. Fakt leniwego przetwarzania operacji `filter` przez `stream`, powoduje wzrost wydajności przy wzroście ilości funkcji filtrujących.

Drugi przypadek testowy polegał na wykonywaniu operacji `map`. Wariant z pojedynczym mapowaniem, dokonywał konwersji liczb całkowitych znajdujących się w identycznym jak w pierwszym przypadku wektorze do liczb zmiennoprzecinkowych poprzez dodanie do każdej z nich wartości 0.1. Wariant podwójny polegał na odwróceniu konwersji na wyniku, czyli przekształcenie go z powrotem do liczb całkowitych przez, odjęcie liczby 0.1. Operacja `map` w odróżnieniu od `filter` jest operacją terminalną, co oznacza, że nie jest wartościowana leniwie. Można zatem spodziewać się, że jej działanie będzie wolniejsze od operacji bezpośrednio na wektorze w obu przypadkach, ponieważ nie ma zysku ze zmniejszenia liczby iteracji. Wyniki testu dla miliona elementów przedstawione są w tabeli 4.3, numery kolumn oznaczają metody w taki sam sposób jak w pierwszym przypadku testowym.

Wyniki ponownie potwierdzają dodatkowy koszt wynikający z zastosowania wyrażeń, lambda. Jednakże jak widać przy funkcji konwertującej jest on mniej odczuwalny niż przy zastosowaniu predykatu. Prawdopodobnie wynika to z faktu optymalizacji stosowanych przez kompilator, w przypadku użycia funkcji jako warunku instrukcji `if`. Jak się okazuje przy tym rodzaju operacji wybór pętli również nie ma tak dużego wpływu na koszt wykonania, to z kolei jest najprawdopodobniej skutkiem przeniesienia największego kosztu operacji na konwersje wartości. Ze względu na prawie stałą (0.5 ms - 0.6 ms) różnicę między przetwarzaniem wektora z lambdą i bez, oraz praktycznie zerową (średnio 0.1 ms) różnicę między dwoma rodzajami pętli, pojedyncze wyniki nie będą porównywane z operacją na strumieniu. Zamiast tego przedstawione dane będą dotyczyć różnic między strumieniem, a uśrednioną wartością, pozostałych operacji. Porównanie wyników w stosunku procentowym przedstawia się dość niekorzystnie dla zastosowania strumienia. Rozwiązanie jest o 63% wolniejsze

Tablica 4.3: Mapowanie, 1 000 000 elementów

	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)
Pojedyncze mapowanie	12.117	7.058	7.114	7.634	7.778
Podwójne mapowanie	19.342	14.926	14.729	15.485	15.585

Tablica 4.4: Mapowanie, 10 000 000 elementów

	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)
Pojedyncze mapowanie	158.278	105.3	105.329	110.642	111.69
Podwójne mapowanie	239.829	196.56	196.158	203.226	204.444

w przypadku pojedynczego mapowania. Podwójne mapowanie to już jednak tylko 27% wzrost czasu wykonania. Można jednak dostrzec ciekawszą zależność, która jeśli okazałaby się prawdziwa, zapewniłaby akceptowalny koszt operacji na strumieniach. Zarówno w przypadku pojedynczego jak i podwójnego mapowania, różnica pomiędzy działaniem z użyciem strumienia, a średnią z użyć wektora, wynosiła 4-5 milisekund. Konkretnie było to 4.721 w przypadku pojedynczego i 4.16075 w przypadku podwójnego mapowania. Jeśli więc zależność ta nie jest dziełem przypadku, oznacza to, że operacja strumieniowa, będzie wolniejsza od wektorowej o stałą wartość czasu, w zależności od ilości elementów w kolekcji. Konkretnie koszt opakowania operacji strumieniowej byłby równy mniej więcej średniej z tych dwóch wartości, podzielonej przez ilość elementów tej kolekcji czyli w tym przypadku milion. Wartość ta wyniosłaby wtedy około 0.0000044 milisekundy, można powiedzieć, że taki dodatkowy koszt jest kosztem akceptowalnym. W celu sprawdzenia tej zależności, przeprowadzony został test z użyciem list zawierających dziesięć milionów elementów, wyniki testu widoczne są w tabeli 4.4. Wyniki testu potwierdzają przypuszczenia, różnica w przypadku pojedynczego mapowania wyniosła 46.03775 milisekund zaś dla podwójnego 42.732 milisekund. Zachowany jest więc zbliżony stosunek procentowego wzrostu dla pojedynczych i podwójnych operacji, odpowiednio 43% i 21%, oraz średni dodatkowy koszt na element. Po obliczeniu średniej wartości różnic i obliczeniu kosztu na pojedynczy element kolekcji, okazuje się, że wyniki zgadzają różnią się dopiero na ósmym miejscu po przecinku. Można zatem na tej podstawie przyjąć, że koszt opakowania kolekcji w strumień dla operacji map wynosi około 4.4 nanosekundy.

Przedstawione testy pokazują, że dodatkowy narzut zastosowania API strumieniowego, choć niepodważalny jest niewielki. Ponad to w przypadku operacji filtrowania, fakt leniwego wartościowania operacji powoduje większą wydajność przy niektórych zastosowaniach. `Streamnie` nada się raczej do rozwiązań w których liczą się milionowe części sekundy, jednak nie do tego typu zadań zostało ono stworzone. W przypadku aplikacji biznesowych, czy też rozwiązań użytkowych na komputery osobiste, wydajność rozwiązania nie powinna sprawiać żadnego problemu.

4.5 Ocena ergonomii przygotowanego rozwiązania

Wygoda korzystania z konkretnej biblioteki często stanowi o jej sukcesie w równym stopniu jak oferowana funkcjonalność. Jeśli chodzi o ergonomię przygotowanego rozwiązania, to jest ona na poziomie akceptowalnym, jednakże nie idealnym. Wygoda samych metod odpowiada tej prezentowanej przez język Java. Podobieństwo składni powoduje, że osoba zaznajomiona z innymi rozwiązaniami tego typu nie powinna mieć problemów z jego opanowaniem. Istnieją jednak dwa problemy spowodowane dwoma niezależnymi od rozwiązania czynnikami, wynikającymi ze specyfiki języka.

Pierwszy z nich to automatyczna dedukcja typu. Choć `C++` wprowadził słowo kluczowe `auto`, to typy otrzymane przy jego użyciu nie zawsze odpowiadają tym, których spodziewałby się programista. Tak więc o ile operacja `filter`, przyjmująca predykat, bez najmniejszego problemu akceptuje wyrażenie lambda podane `inline`, bez rzutowania, o tyle operacja map wymaga już więcej troski. Ciekawy przykład tego problemu przedstawiony został na listingu 4.2.

Listing 4.2: Błędna dedukcja typu

```
1  std::vector<int> v = {1, -2, 3, -4};
2      stream::stream<int> *testStream1 = new stream::stream<int>(v);
3      stream::stream<int> *testStream2 = new stream::stream<int>(v);
4      stream::stream<int> *testStream3 = new stream::stream<int>(v);
5      stream::stream<int> *testStream4 = new stream::stream<int>(v);
6      std::function<double(int)> lambdaZTypem = [](int a) { return a
           + 0.1; };
7      auto lambdaAuto = [](int a) -> double { return a + 0.1; };
8      testStream1
9          ->map(lambdaZTypem)
10         ->peek();
11     testStream2
12         ->map((std::function<double(int)>) ([](int a) { return a +
               0.1; }))
13         ->peek();
14     testStream3
15         ->map([](int a) { return a + 0.1; })
16         ->peek();
17     testStream4
18         ->map(lambda)
19         ->peek();
```

Jak widać sama funkcja anonimowa jest w każdym z trzech przypadków zdefiniowana w identyczny sposób. Jest to funkcja mapując zmienną typu int do typu double, czyli dokładnie taka jaką zastosowano w przypadku testów opisanych w poprzednim rozdziale. Jednak pomimo faktu identycznej deklaracji, programista może być zaskoczony faktem, że tylko dwa z powyższych wywołań zadziałają poprawnie. Rozwiązania zastosowane na strumieniach pierwszym i drugim zakończą się sukcesem, natomiast warianty w przypadkach trzecim i czwartym spowodują błędy kompilacji. Typ generowany przez kompilator dla wyrażenia lambda wygląda mniej więcej w następujący sposób `metodaOkalajaca()::__lambdaNUMER`. Taki typ przyjmie zmienna auto w przykładzie z linii siódmej oraz lambda utworzona w linii piętnastej. Będą to różne typy jednak wzajemnie kompatybilne. I choć kompilator nie ma problemu

z przypisaniem tego typu do `std::function<double(int)>`, to jeśli chodzi o wydedukowanie tego typu w przypadku gdy oczekiwana jest funkcja typu `std:function<R(T)>` jest już dla niego niemożliwe. Zastąpiony zostaje tylko jeden z szablonowych parametrów i otrzymujemy informację o nieznalezieniu pasującej metody. Błąd jest identyczny dla przypadków trzeciego i czwartego. Jest to dość duża wada, znacznie wpływająca na wygodę korzystania z rozwiązania. Niestety o ile w języku nie zostanie poprawiona dedukcja typów, nie istnieje rozwiązanie, mogące ten problem rozwiązać.

Drugim problemem, który powoduje, że korzystanie z rozwiązania nie będzie tak wygodne jak w językach Java czy C#, jest brak automatycznego zarządzania pamięcią. Dzięki mechanizmom odśmiecania pamięci strumienie w tych językach nie muszą zajmować się destrukcją obiektów pośrednich, brak tych mechanizmów w języku `C++` powoduje, konieczność ręcznego sprzątania utworzonych strumieni. Nie stanowi to większego problemu w przypadku większości operacji ponieważ większość z nich nie tworzy zwracanych na zewnątrz obiektów pośrednich. Dlatego można bez obawy składać operacje `filter` i redukować strumienie za pomocą `reduce`, wtedy jedyną konieczną operacją będzie `delete`, na przetwarzanym strumieni. Jest to operacja naturalna dla deweloperów języka `C++`, dlatego nie można jej nawet traktować jako wadę. Inaczej sprawa wygląda z operacją `map`, która tworzy nowy strumień. W jej przypadku należy pamiętać zarówno o konieczności usunięcia strumienia bazowego, jak i tego utworzonego przez `map`. Jest to duży koszt ze względu na utrudnienie łańcuchowego wykonywania operacji. Konieczne staje się przypisanie wyniku do zmiennej w celu późniejszego usunięcia jej po użyciu. Choć nie jest to wada uniemożliwiająca korzystanie, z API, to ma znaczący wpływ na jego odbiór. Podobnie jednak jak pierwszy opisany problem jest to wada niezależna od rozwiązania i wynika wprost z wybranego języka programowania.

Bibliografia

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

[2] Zrozumieć LINQ, http://itcraftsman.pl/zrozumiec-linq/

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

[4] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[5] NoBlogDefFound Tomasz Nurkiewicz around Java and code quality http://www.nurkiewicz.com/2014/07/introduction-to-writing-custom.html

[6] Słownik języka polskiego PWN, Wydawnictwo Naukowe PWN

[7] cppreference.com http://en.cppreference.com/w/cpp/language/lambda

[8] Java 8 in Action: Lambdas, Streams, and functional-style programming 1st Edition, Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft, Manning