Specyfikacja funkcjonalna
Projekt „Group of Visionaires”

Mateusz Smoliński

11 listopada 2018

Spis treści

1 Wstęp

Celem projektu jest napisanie programu, który wykona dwa zadania:

- wykrywanie korzystnej ścieżki wymiany dla wskazanej waluty,
- wykrywanie sytuacji ekonomicznego arbitrażu, czyli kombinacji, gdy kursy walut są tak ułożone, aby wymiana cykliczna zwracała więcej niż kwota wejściowa.

Program będzie uruchamiany z 3 lub 4 argumentami:

- nazwę pliku z danymi, w którym zawarte są bieżące kursy walut,
- kwotę wejściową,
- walutę wejściową,
- walutę wyjściową (argument opcjonalny).

Efektem pracy programu będzie wypisanie na standardowym strumieniu wyjścia ciągu najkorzystniejszej wymiany walut, informacja o braku takiego ciągu lub odpowiedni komunikat o błędzie w przypadku niepoprawnych danych.

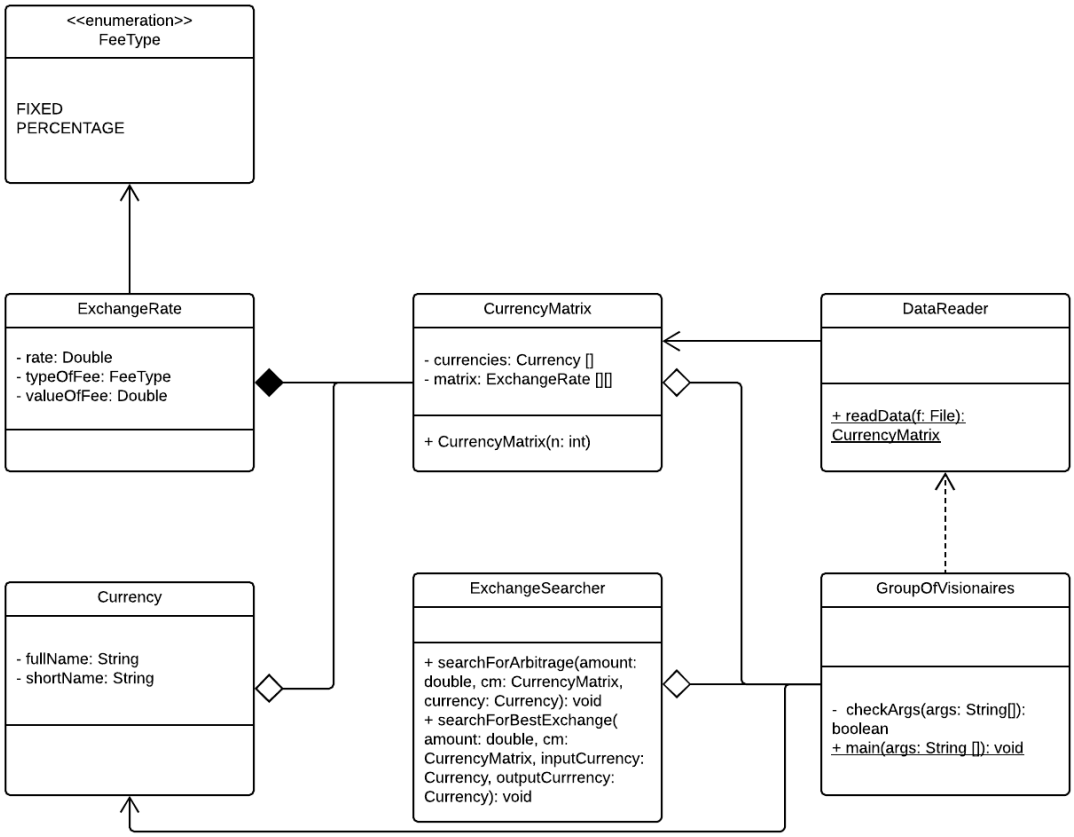
Projekt „Group of Visionaires" zostanie napisany w języku Java, w środowisku NetBeans IDE 8.2 na wersji Javy 1.8.0_162. Implementacja oraz testowanie programu odbywać się będą na komputerze o następujących parametrach:

- system operacyjny Windows 10 Home ver. 1803,
- procesor Intel Core i5-7200U,
- pamięć RAM 8,00 GB,
- karty graficzne Intel HD Graphics 620 + NVIDIA GeForce 940MX.

2 Diagram klas

Program składać się będzie z 6 klas i jednego typu wyliczeniowego. Zależności pomiędzy poszczególnymi klasami obrazuje Rysunek 1.

W celu poprawy czytelności na diagramie klas nie zostały uwzględnione metody dostępowe dla klas Currency, ExchangeRate oraz CurrencyMatrix oraz konstruktory dwóch pierwszych klas, które we wszystkich przypadkach wypełniają pola danych klas.



Rysunek 1: Diagram klas

3 Opis klas/metod

3.1 Klasa GroupOfVisionaires

Jest to główna klasa tego programu. Jest zbudowana z dwóch kluczowych metod odpowiedzialnych za interpretację przyjętych argumentów i przesłanie ich do dalszej części programu, po upewnieniu się, że zostały one poprawnie podane. Jedną z tych metod jest *main*, która rozpoczyna działanie programu. Nazwa tej klasy odpowiada nazwie całego projektu.

3.1.1 Metoda checkArgs

- `private boolean checkArgs(String[] args)`

Jest to metoda sprawdzająca podane przez użytkownika argumenty. Jej zadaniem jest przede wszystkim zweryfikowanie, czy użytkownik wprowadził prawidłowy skrót waluty (lub skróty walut). Metoda sprawdza również, czy nie została wprowadzona niepoprawna kwota. W przypadku napotkania błędu lub niezidentyfikowanej waluty wypisze ona odpowiedni komunikat na standardowym wyjściu i zwróci fałsz, uniemożliwiając kontynuowanie programu dla niepoprawnych danych. Po udanej weryfikacji metoda zwraca prawdę.

3.1.2 Metoda main

- `public static void main(String[] args)`

Poza rozpoczęciem działania programu, zadaniem metody *main* będzie także zweryfikowanie ilości argumentów oraz próba otwarcia pliku z danymi. W przypadku powodzenia wywoła statyczną metodę *readData*, która została opisana poniżej. Jeśli *readData* nie zgłosi żadnych wyjątków, metoda *main* wywoła opisaną w poprzednim podpunkcie metodę *checkArgs*. Jeżeli ona zwróci prawdę, nastąpi rozpoczęcie właściwego działania programu i zebrane dane (w postaci *args* oraz zwróconego przez *readData* obiektu *CurrencyMatrix*) zostaną przekazane do odpowiedniej metody z klasy *ExchangeSearcher*. Po zakończeniu jej działania program zostanie zamknięty.

3.2 Klasa DataReader

Klasa *DataReader* jest odpowiedzialna za odczyt danych z pliku tekstowego oraz ewentualną obsługę błędów związanych z tym plikiem. Składa się z jednej, kluczowej metody statycznej, która nie tylko odczytuje dane z pliku podanego przez użytkownika, ale także przygotowuje je w formie zrozumiałej dla reszty programu – w postaci obiektu klasy *CurrencyMatrix*.

3.2.1 Metoda readData

- `public static CurrencyMatrix readData (File f)`

Metoda *readData* otrzymuje jako argument plik z danymi, wcześniej sprawdzony w metodzie *main* jako poprawny plik tekstowy. Odczytuje ona dane z tego pliku, trzymając się konwencji ustalonej w specyfikacji funkcjonalnej – deklaracje walut i kursów wymian są poprzedzone linią rozpoczynającą się od znaku „#”. Pierwotnie odczytane dane są zapisywane do list. Po wczytaniu wszystkich danych metoda tworzy obiekt klasy *CurrencyMatrix* i zapisuje zinterpretowane dane do macierzy. W przypadku zauważenia niepożądanych danych lub danych podanych w niewłaściwy sposób metoda wypisuje odpowiednie komunikaty na standardowym wyjściu, a w przypadku, gdy działanie programu nie może zostać kontynuowane – zwraca wyjątek, który musi zostać obsłużony w metodzie *main*.

3.3 Klasa CurrencyMatrix

Klasa reprezentująca najważniejsze dane, jakie dostajemy od użytkownika. Ma postać macierzy reprezentującej kursy pomiędzy dwoma walutami o danym ID. Poza tą macierzą przechowywana jest również tablica walut, o indeksach odpowiadających ID danej waluty. Oprócz metod dostępowych klasa posiada jedynie konstruktor.

3.3.1 Konstruktor klasy CurrencyMatrix

- `public CurrencyMatrix (int n)`

Jest to konstruktor tej klasy, który tworzy macierz na podstawie podanej liczby walut. Podana liczba będzie odpowiadać zarówno rozmiarowi tablicy walut, jak i będzie stanowić oba rozmiary (wysokość i szerokość) macierzy.

3.4 Klasa Currency

Klasa reprezentuje waluty w programie. Większość logiki programu opierać się będzie na ID walut, a nie ich nazwach, przez co program będzie się odwoływać do tej klasy głównie na początku i na końcu programu – przy wczytywaniu oraz tworzeniu wynikowych napisów. Przechowuje ona pełne nazwy walut oraz ich nazwy skrócone. Klasa posiada jedynie metody dostępowe oraz konstruktor wprowadzający oba pola klasy.

3.5 Klasa ExchangeRate

W tej klasie zawarty jest kurs pomiędzy dwoma walutami oraz informacje dotyczące opłaty ponoszonej w wyniku wymiany walut. Ze względu na to, że opłata może mieć postać procentową lub stałą od waluty wynikowej, jednym z jej pól jest *typeOfFee*, reprezentowany przez wyliczeniowy typ danych przedstawiony na diagramie klas (Rysunek 1). Klasa ta, podobnie jak klasa *Currency* posiada jedynie metody dostępowe oraz konstruktor wypełniający wszystkie pola tej klasy.

3.6 Klasa ExchangeSearcher

W tej klasie zawarta jest najważniejsza logika programu. Zawiera ona dwie kluczowe metody, które wyszukują wybraną przez użytkownika ścieżkę. Nie posiada ona żadnych pól.

3.6.1 Metoda searchForArbitrage

- `public void searchForArbitrage (double amount, CurrencyMatrix cm, Currency currency)`

Metoda jest odpowiedzialna za wyszukanie ekonomicznego arbitrażu (jeśli takowy występuje). Otrzymuje jako argumenty macierz *CurrencyMatrix*, wartość początkową oraz walutę. Wynikiem jej działania jest wypisanie na ekran odnalezionej ścieżki lub informacji o jej braku.

3.6.2 Metoda searchForBestExchange

- `public void searchForBestExchange (double amount, CurrencyMatrix cm, Currency inputCurrency, Currency outputCurrency)`

Metoda jest odpowiedzialna za wyszukanie najlepszej ścieżki wymiany walut. Otrzymuje jako argumenty macierz *CurrencyMatrix*, wartość początkową, walutę początkową oraz walutę docelową. Wynikiem jej działania jest wypisanie na ekran najlepszej ścieżki wymiany lub informacji o jej braku.

4 Działanie programu i zastosowanie algorytmu

Uruchomienie programu powoduje rozpoczęcie wykonywania funkcji *main*. Po weryfikacji ilości argumentów i wczytaniu pliku z danymi przez metodę statyczną *readData* następuje weryfikacja pozostałych argumentów metodą *checkArgs*. Następnie, na podstawie ilości podanych argumentów program uruchamia odpowiednią metodę wyszukującą.

Wyszukiwanie oparte będzie na algorytmie przeszukiwania grafu wszerz (BFS). Zastosowanie algorytmów znajdujących najkrótszą ścieżkę w grafach ważonych, takich jak algorytm Dijkstry nie pomoże w rozwiązaniu problemu wymiany walut, gdyż wartość po przejściu przez kolejny wierzchołek (walutę) wartość może się zmniejszyć i wymagałoby to dodatkowych modyfikacji algorytmu. Algorytm przeszukiwania wszerz pozwala na odczytanie wszystkich możliwych ścieżek, których porównanie pozwoli na pewne ustalenie ścieżki najkorzystniejszej. Działanie metod wyszukujących będzie oparte na wyborze faworyta – gdy zostanie odnaleziona ścieżka spełniająca warunek, zostanie ona zapisana w postaci listy walut oraz wartości końcowej. Działanie algorytmu będzie kontynuowane i w przypadku korzystniejszej wartości końcowej wartości dotychczas faworyzowane zostaną zastąpione.

Ostatnim etapem algorytmu jest wypisanie odnalezionej ścieżki wraz z wartością końcową. W przypadku niepowodzenia wyszukiwania zostanie wypisany odpowiedni komunikat. Po wykonaniu tej czynności program zakończy działanie.

5 Testy jednostkowe

Projekty testów zakładają użycie narzędzia JUnit. Poniżej znajduje się lista testów planowanych dla kluczowych metod programu, dla każdej z nich przewidziane są różne przypadki otrzymanych danych.

5.1 Testy klas Currency i ExchangeRate

Testy w tych klasach ograniczą się do sprawdzenia poprawności zapisywanych danych. Dla pierwszej z nich zostaną wprowadzone przykładowe nazwy skrócone i pełne, dla drugiej wartości kursów, wartości opłat oraz oba typy opłat, po czym zostanie sprawdzone, czy nazwy wyjmowane getterami są równoważne tym ustawionym setterami.

5.2 Testy klasy CurrencyMatrix

Klasa *CurrencyMatrix* będzie testowana podobnie do dwóch poprzednich klas. Główną różnicą w tej klasie jest to, że składa się ona z tablicy i macierzy, dlatego przetestowane zostanie ustawianie wartości w różnych miejscach macierzy – zarówno w poprawnych lokacjach, jak i poza ustalonymi na początku granicami macierzy. Podobnie jak w poprzednim podpunkcie testowanie tej klasy ograniczy się do używania getterów i setterów.

5.3 Testy klasy ExchangeSearcher

Klasa *ExchangeSearcher* przechowuje główną logikę programu i będzie wymagała odpowiednio dopasowanych testów. Dla wyszukiwania arbitrażu (metody *searchForArbitrage*) wprowadzone zostaną następujące zestawy danych:

- macierz zawierającą wyłącznie jeden korzystny arbitraż, spodziewanym efektem jest jego wypisanie,
- macierz zawierającą dwa arbitraże, efektem spodziewanym jest wypisanie korzystniejszego,
- macierz zawierającą cykl, który nie będzie jednak korzystną wymianą – spodziewany efekt to informacja o braku arbitrażu,
- macierz, która nie będzie posiadała żadnego cyklu – efektem ponownie powinna być informacja o braku arbitrażu.

Podobnie dla metody *searchForBestExchange* zostaną przygotowane zestawy danych:

- macierz zawierającą wyłącznie jedną ścieżkę, spodziewanym efektem jest jej wypisanie,
- macierz zawierającą dwie ścieżki, efektem spodziewanym jest wypisanie korzystniejszej,
- macierz nie zawierająca ścieżki docelowej – efektem powinno być wypisanie komunikatu o braku ścieżki.

5.4 Testy klasy DataReader

Ta klasa ma za zadanie odczytać poprawnie plik tekstowy, zatem jej testowanie opierać się będzie na podawaniu przygotowanych zestawów danych, podanych prawidłowo oraz z różnymi błędami. W przygotowanych plikach znajdą się:

- plik poprawny, efektem powinno być zwrócenie prawidłowej macierzy,
- pliki z błędami na pojedynczych znakach w nazwach walut/liczbach/typach opłat, efektem powinno być wypisanie ostrzeżeń na standardowym wyjściu i zwrócenie macierzy ignorujących te linie,
- pliki z błędami kluczowymi dla programu, bez linii oddzielających lub całkiem bez danych, oczekiwanym efektem będzie wyrzucenie wyjątku przez metodę *readData*.

5.5 Testy klasy GroupOfVisionaires

Ta klasa odpowiada za złączenie całego programu w jedno, zatem testowanie jej opierać się będzie głównie na testach akceptacyjnych opisanych w specyfikacji funkcjonalnej. Wśród testów jednostkowych dla tej klasy powinny znaleźć się testy metody *checkArgs*, żeby sprawdzić jej poprawną weryfikację dla niewystępujących w macierzy walut oraz dla niepoprawnie podanych kwot.