VaccDistributor

Kamil Sztandur

14/11/2020

SPECYFIKACJA IMPLEMENTACYJNA

Rozdział 1: INFORMACJE OGÓLNE.

Podrozdział 1.1: Nazwa programu.

Program nazywa się **VaccDistributor**.

Podrozdział 1.2: Poruszany problem.

Problemem rozwiązywanym przez ten program jest zagadnienie jak najbardziej optymalnego pod kątem ekonomicznym rozdysponowania szczepionek pomiędzy poszczególnymi dostawcami, a aptekami, na podstawie podanych przez użytkownika połączeń. Program skupia się na tym, aby koszt dystrybucji szczepionek był jak najmniejszy. Nie gwarantuje, że zapotrzebowanie każdej apteki zostanie zaspokojone. Także jest to wariant egoistyczny.

Podrozdział 1.3: Uruchomienie programu.

Uruchomienie programu ze względu na brak oprawy graficznej może sprawiać trudności mniej wtajemniczonym w komputery użytkownikom. Program uruchamia się z poziomu terminalu systemu operacyjnego, na którym jest zainstalowana najnowsza wersja oprogramowania Java. Aby uruchomić program wystarczy wpisać komendę w terminalu systemu operacyjnego spełniającego opisane wymagania:

```
java -jar vaccDistributor.jar input.txt
```

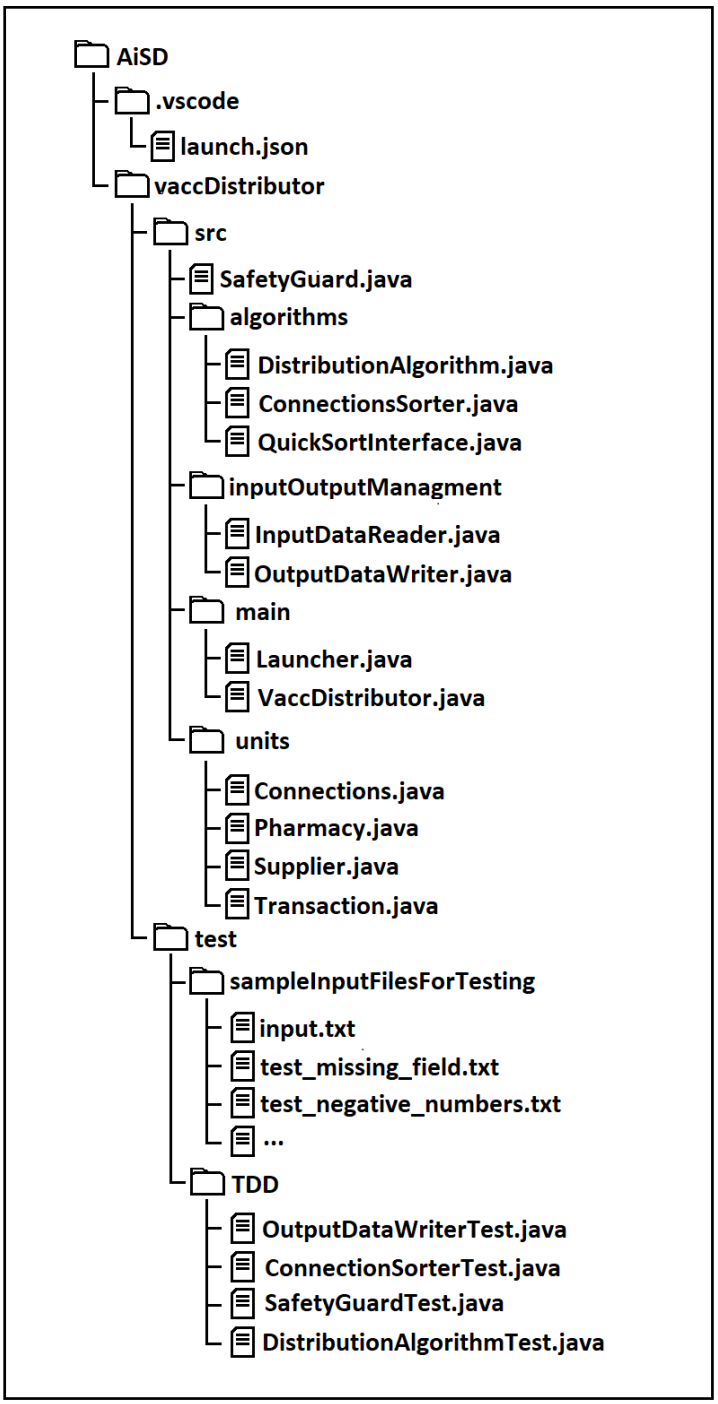
znajdując się w katalogu, w którym umieszczone zostały równocześnie plik programu **vaccDistributor.jar** oraz tekstowy plik wejściowy z danymi (tutaj **input.txt**). Należy pamiętać o dopisaniu rozszerzenia pliku. Sama nazwa (tutaj **input**) nie pozwoliłaby na odnalezienie wskazanego pliku.

Należy dokładnie przestudiować strukturę pliku wejściowego na podstawie poniższego przykładu, ponieważ wszelkie odstępstwa od ogólnoprzyjętego szablonu mogą spowodować nieprawidłową pracę, a nawet zatrzymanie pracy programu. Tekstowy plik wejściowy powinien zawierać trzy nagłówki, oznaczające kolejno listy producentów, aptek i dostępnych połączeń. Kolejne parametry w linijce powinny być oddzielna ciągiem znaków " | ". Nie należy powielać nagłówków ani umieszczać ich w innej kolejności.

Przykładowy plik wejściowy:

```
# Producenci szczepionek (id — nazwa — dzienna produkcja)
0 | BioTech 2.0 | 900
1 | EkoPolska 2020 | 1300
# Apteki (id | nazwa | dzienne zapotrzebowanie)
0 | CentMedEko Centrala | 450
1 | CentMedEko 24h | 690
# Połączenia producentów i aptek (id producenta | id apteki | dzienna maksymalna liczba dostarczanych szczepionek | koszt szczepionki [zł] )
0 | 0 | 800 | 70.5
0 | 1 | 600 | 70
1 | 0 | 900 | 100
1 | 1 | 600 | 80
```

Rozdział 2: OPIS MODUŁÓW I PAKIETÓW.



Podrozdział 2.1: PAKIET 'vaccDistributor'

Ten pakiet jest głównym pakietem programu. Zawiera jedynie dwa podpakiety odpowiadające za logikę oraz testowanie programu:

- **src** - pakiet stanowiący całą logikę programu.
- **test** - pakiet odpowiedzialny za testowanie programu.

Podrozdział 2.1.1: PAKIET 'vaccDistributor.src'

Ten pakiet zawiera całą logikę programu, czyli podpakiety odpowiadające za pewne segmenty programu oraz klasę ds. bezpieczeństwa pracy programu:

- **SafetyGuard.java** - klasa stanowiąca zbiór metod pozwalających kontrolować bezpieczeństwo przebiegu programu, wykorzystywana we wszystkich podpakietach pakietu src.
- **algorithms** - podpakiet zawierający algorytmy zastosowane w programie.
- **inputOutputManagment** - podpakiet zawierający zbiór obiektów zajmujących się wyjściem/wejściem w programie.
- **main** - podpakiet zawierający główne ciało programu, zarządzające i korzystające z pozostałych pakietów i obiektów programu.
- **units** - podpakiet zawierający obiekty jednostek używanych w programie przedstawiających interesujące nas realne obiekty.

Podrozdział 2.1.1.1: PAKIET 'vaccDistributor.src.algorithms'

Podpakiet zawierający algorytmy zastosowane w programie.

- **DistributionAlgorithm.java** - klasa zawierająca algorytm tworzący najbardziej ekonomiczne (tanie) połączenia pomiędzy producentami szczepionek, a aptekami.
- **ConnectionsSorter.java** - klasa zawierająca algorytm sortowania połączeń (używany przez główny algorytm programu) bazujący na sortowaniu szybkim QuickSort.
- **QuickSortInterface.java** - interfejs pomocniczy i pilnujący bezpieczeństwa właściwego formatu sortowania szybkiego w algorytmie sortowania połączeń.

Podrozdział 2.1.1.2: PAKIET 'vaccDistributor.src.inputOutputManagment'

Podpakiet zawierający klasy zajmujące się wejściem i wyjściem w programie.

- **InputDataReader.java** - klasa odpowiedzialna za czytanie danych podanych przez użytkownika i utworzenie z nich możliwych do pobrania z tej klasy 'kontenerów' z przeczytanymi danymi.
- **OutputDataWriter.java** - klasa odpowiedzialna za utworzenie katalogu z danymi wyjściowymi oraz wypisywanie danych wyjściowych (najbardziej optymalnych połączeń z aptekami) do pliku tekstowego w utworzonym katalogu.

Podrozdział 2.1.1.3: PAKIET 'vaccDistributor.src.main'

Podpakiet zawierający główne ciało programu. Korzysta ono z pozostałych obiektów i pakietów dostępnych w ramach pakietu **src**.

- **Launcher.java** - klasa zawierająca główną metodę **main**, która odpowiada tylko za uruchomienie programu.
- **VaccDistributor.java** - klasa zawierające główne ciało programu, realizujące krok po kroku wszystkie wymagania biznesowe programu.

Podrozdział 2.1.1.4: PAKIET 'vaccDistributor.src.units'

Podpakiet zawierający obiekty jednostek używanych w programie przedstawiających interesujące nas realne obiekty takie jak apteki czy producentów. To na nich pracuje algorytm ustalający dystrybucję szczepionek.

- **Supplier.java** - klasa stanowiąca reprezentację producenta szczepionek (in. dostawcy) w programie.
- **Pharmacy.java** - klasa stanowiąca reprezentację placówkę apteki w programie.
- **Connection.java** - klasa stanowiąca reprezentację połączenia producent - apteka w programie.
- **Transaction.java** - klasa stanowiąca reprezentację optymalnego połączenia ustalonego przez algorytm, która jest de facto umową na dostarczenie pewnej ilości szczepionek za pewną cenę danym połączeniem, bazującym na istniejącej klasie **Connection**.

Podrozdział 2.1.2: PAKIET 'vaccDistributor.test'

Pakiet zawierający pakiet i katalog z przykładowymi danymi wejściowymi. Pozwala na testowanie manualne i automatyczne programu.

- **sampleInputFilesForTesting** - nie jest to pakiet, lecz katalog zawierający zestaw przykładowych danych wejściowych w tym poprawnych i błędnych.
- **TDD** - pakiet zawierający zestaw klas oferujących zbiór testów jednostkowych, testujących metody i klasy oferowane przez pakiet **src**.

Podrozdział 2.1.2.1: PAKIET 'vaccDistributor.test.TDD'

Pakiet zawierający zestaw klas oferujących zbiór testów jednostkowych, testujących metody i klasy oferowane przez pakiet **src**.

- **OutputDataWriter.java** - klasa testująca oferująca szereg metod sprawdzających poprawność działania klasy OutputDataWriter z pakietu **vaccDistributor.src.inputOutputManagment**.
- **ConnectionsSorterTest.java** - klasa testująca oferująca szereg metod sprawdzających poprawność działania algorytmu QuickSort z klasy ConnectionsSorter z pakietu **vaccDistributor.src.inputOutputManagment**.
- **SafetyGuardTest.java** - klasa testująca oferująca szereg metod sprawdzających skuteczność klasy ds. bezpieczeństwa pracy programu SafetyGuard z pakietu **vaccDistributor.src**.
- **DistributionAlgorithmTest.java** - klasa testująca oferująca szereg metod sprawdzających poprawność działania algorytmu rozdysponowującego szczepionki i ustalającego jak najbardziej ekonomicznie optymalne połączenia aptek z producentami, czyli klasy DistributionAlgorithm z pakietu **vaccDistributor.src.algorithms**.

Rozdział 3: OPIS KLAS.

Podrozdział 3.1: KLASA 'SafetyGuard'

Klasa stanowiąca zbiór metod pozwalających kontrolować bezpieczeństwo przebiegu programu, wykorzystywana we wszystkich podpakietach pakietu **src**. Nie potrzebuje konstruktora. Zawiera jedynie statyczne metody.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
areConnectionsEfficient EnoughForThisPharmacy	Metoda sprawdzająca, czy połączenia do danej apteki są z góry skazane na niemożność ich zaopatrzenia	Pharmacy:pharmacy, ArrayList<Connection>: connections	boolean
checkIfSupplierData LineCorrect	Metoda sprawdzająca, czy wyczytany w pliku wiersz z danymi producenta jest poprawny. Rzuca wyjątek IAEx, jeżeli nie.	String[]:dataLine, int:lineNumber	void
checkIfPharmacyData LineCorrect	Metoda sprawdzająca, czy wyczytany w pliku wiersz z danymi apteki jest poprawny. Rzuca wyjątek IAEx, jeżeli nie.	String[]:dataLine, int:lineNumber	void
checkIfConnectionData LineCorrect	Metoda sprawdzająca, czy wyczytany w pliku wiersz z danymi połączenia jest poprawny. Rzuca wyjątek IAEx, jeżeli nie.	String[]:dataLine, int:lineNumber	void
isThisPharmacy Duplicated	Metoda sprawdzająca, czy podana apteka jest duplikatem jakiejś już wczytanej.	Pharmacy : newPharmacy, HashMap<Integer, Pharmacy> : pharmacies	boolean
isThisSupplier Duplicated	Metoda sprawdzająca, czy podany producent jest duplikatem jakiegoś już wczytanego.	Supplier : newSupplier, HashMap<Integer, Supplier> : suppliers	boolean
isThisConnection Duplicated	Metoda sprawdzająca, czy podane połączenie jest duplikatem jakiegoś już wczytanego.	Connection: newConnection, ArrayList<Connection>: connections	boolean
isThisTransaction Duplicated	Metoda sprawdzająca, czy podana transakcja jest duplikatem jakiejś już ustalonej przez algorytm	Transaction: newTransaction, ArrayList<Transaction>: transactions	boolean
isThisConnection PharmacyRegistered	Metoda sprawdzająca, czy apteka do której prowadzi podane połączenie istnieje na liście wczytanych aptek.	Connection: connection, HashMap<Integer, Pharmacy>: pharmacies	boolean
isThisConnection SupplierRegistered	Metoda sprawdzająca, czy producent od którego prowadzi podane połączenie istnieje na liście wczytanych producentów.	Connection: connection, HashMap<Integer, Supplier>: suppliers	boolean

Podrozdział 3.2: KLASA 'InputDataReader'

Klasa odpowiedzialna za czytanie danych podanych przez użytkownika i utworzenie z nich możliwych do pobrania z tej klasy 'kontenerów' z przeczytanymi danymi. Konstruktor przyjmuje zmienną typu String z nazwą pliku wejściowego.

ATRYBUTY:

- **private String inputFilename** - zmienna przechowująca nazwę pliku wejściowego z danymi podanego przez użytkownika.
- **private HashMap¡Integer, Supplier¿ suppliers** - kontener przechowujący wczytanych producentów z pliku wejściowego.
- **private HashMap¡Integer, Pharmacy¿ pharmacies** - kontener przechowujący wczytane apteki z pliku wejściowego.
- **private ArrayList¡Connection¿ connections** - kontener przechowujący wczytane połączenia z pliku wejściowego.
- **private boolean suppliersListHasBeenRead** - flaga określająca, czy pole z listą producentów zostało przeczytane. Służy do wykrywania zduplikowanych pól w pliku.
- **private boolean pharmaciesListHasBeenRead** - flaga określająca, czy pole z listą aptek zostało przeczytane. Służy do wykrywania zduplikowanych pól w pliku.
- **private boolean connectionsListHasBeenRead** - flaga określająca, czy pole z listą połączeń zostało przeczytane. Służy do wykrywania zduplikowanych pól w pliku.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
read	Metoda czytająca plik, którego nazwę podano w konstruktorze. Wczytuje do odpowiednich kontenerów wyczytane dane z pliku. Rzuca wyjątek w przypadku błędu.	void	void
getSuppliers	Metoda zwracająca HashMapę z wyczytanymi producentami.	void	HashMap<Integer, Supplier>
getPharmacies	Metoda zwracająca HashMapę z wyczytanymi aptekami.	void	HashMap<Integer, Pharmacy>
getConnections	Metoda zwracająca ArrayListę z wyczytanymi połączeniami.	void	ArrayList<Connection>
readRecordFor	Metoda prywatna używana przez read() czytająca bieżącą linijkę i zapisującą uzyskaną daną do odpowiedniego kontenera na podstawie parametru nowReading.	String:nowReading, String:line, int:lineNumber	void
getInputFile	Metoda prywatna używana przez read() zwracająca wskaźnik do pliku z danymi wejściowymi lub rzucająca wyjątek, jeżeli plik nie istnieje.	void	File

Podrozdział 3.3: KLASA 'OutputDataWriter'

Klasa odpowiedzialna za utworzenie katalogu z danymi wyjściowymi oraz wypisywanie danych wyjściowych (najbardziej optymalnych połączeń z aptekami) do pliku tekstowego w utworzonym katalogu. W konstruktorze przyjmuje nazwę, którą ma przyjąć plik z danymi wyjściowymi.

ATRYBUTY:

- **private String directoryName** - zmienna przechowująca nazwę katalogu z danymi wyjściowymi.
- **private String filename** - zmienna przechowująca nazwę pliku tekstowego z danymi wyjściowymi.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
createOutputDir	Metoda tworząca katolog z rozwiązaniem. Rzuca wyjątek IOEx w przypadku problemu.	void	void
save	Metoda zapisująca listę transakcji do pliku wyjściowego. Rzuca wyjątek IOEx w przypadku problemu.	ArrayList<Transaction>: transactions	void
deleteFileIfExists	Prywatna metoda usuwająca folder z danymi wyjściowymi, jeżeli ten już istnieje.	File file	void

Podrozdział 3.4: KLASA 'Launcher'

Klasa zawierająca główną metodę **main**, która odpowiada tylko za uruchomienie programu.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
main	Metoda główna uruchomieniowa programu. Tworzy obiekt VaccDistributor i uruchamia go, podając tablicę args.	String[]:args	void

Podrozdział 3.5: KLASA 'VaccDistributor'

Klasa zawierające główne ciało programu, realizujące krok po kroku wszystkie wymagania biznesowe programu. W konstruktorze przyjmuje tablicę argumentów typu String[], podaną z metody głównej main().

- **private ArrayList¡Transaction¿ transactions** - Lista utworzonych w wyniku działania algorytmu transakcji.
- **private InputDataReader inputDataReader** - obiekt odpowiedzialny za czytanie danych wejściowych.
- **private OutputDataWriter outputDataWriter** - obiekt odpowiedzialny za zapisywanie danych wyjściowych.
- **private String[] args** - tablica z argumentami, przekazana z meto głównej main().

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
launch	Ciało główne programu. Wykonuje kolejne kroki wymagań biznesowych.	void	void
getInputFilename	Prywatna metoda sprawdzająca, czy użytkownik podał dane wejściowe i zwracająca nazwę pliku wejściowego. Rzuca wyjątek w przypadku, gdy użytkownik nie podał żadnych danych.	void	String
readData	Prywatna metoda inicjująca czytnik danych wejściowych i uruchamiająca go. Rzuca wyjątek w przypadku problemów.	void	void
detectAnyLogical ProblemsInData	Prywatna metoda przeszukująca dane wejściowe w przypadku problemów logicznych, które są sprzeczne z wymaganiami biznesowymi.	void	void
calculateBest Transactions	Prywatna metoda inicjująca algorytm ustalania optymalnych transakcji, uruchamiająca ją i pobierająca wynik do atrybutu transactions.	void	void
saveData	Prywatna metoda inicjująca writer danych wyjściowych i uruchamiająca go, podając atrybut transactions. Rzuca wyjątek w przypadku problemów.	void	void

Podrozdział 3.6: KLASA 'Connection'

Klasa stanowiąca reprezentację połączenia producent - apteka w programie. Konstruktor przyjmuje numer identyfikacyjny producenta, numer identyfikacyjny apteki, maksymalną ilość szczepionek, która może zostać dostarczona tym połączeniem, oraz koszt za jedną szczepionkę, dostarczaną tym połączeniem.

- **private int supplierID** - numer identyfikacyjny producenta
- **private int pharmacyID** - numer identyfikacyjny apteki
- **private int maxTransfer** - maksymalna ilość szczepionek, która może zostać dostarczona tym połączeniem.
- **private int availableTransfer** - bieżąca dostępna ilość szczepionek, która może zostać dostarczona tym połączeniem.
- **private double costPerSingleVaccine** - koszt za jedną szczepionkę, dostarczaną tym połączeniem.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
getSupplierID	Metoda zwracająca numer identyfikacyjny producenta.	void	int
getPharmacyID	Metoda zwracająca numer identyfikacyjny apteki.	void	int
getMaxTransfer	Metoda zwracająca maksymalną ilość szczepionek, która może zostać dostarczona tym połączeniem.	void	int
getAvailableTransfer	Metoda zwracająca bieżącą ilość szczepionek, która może zostać dostarczona tym połączeniem.	void	int
getCostPer SingleVaccine	Metoda zwracająca koszt za jedną szczepionkę, dostarczaną tym połączeniem.	void	double
removeFrom AvailableTransfer	Metoda odejmująca pewną ilość od bieżącego dostępnego transferu np. po zawarciu umowy pewnym połączeniem. Zwraca false, jeżeli ta ilość jest większa od pozostałej ilości dostępnego transferu.	int:amount	boolean
@Override toString	Przysłonięta metoda konwertująca obiekt na typ String według szablonu '[Nazwa producenta -¿ Nazwa apteki]	void	String
@Override equals	Przysłonięta metoda rozstrzygająca równość dwóch obiektów typu Connection tak, aby sprawdzała równość ID producentów i ID aptek obu połączeń.	void	boolean

Podrozdział 3.7: KLASA 'Pharmacy'

Klasa stanowiąca reprezentację placówkę apteki w programie. Konstruktor przyjmuje numer identyfikacyjny apteki, nazwę apteki oraz ilość dziennego zapotrzebowania na szczepionek.

- **private int id** - numer identyfikacyjny apteki.
- **private String name** - nazwa apteki.
- **private int dailyNeed** - dziennie zapotrzebowanie na szczepionki apteki.
- **private int currentDailyNeed** - bieżące dzienne zapotrzebowanie na szczepionki po odjęciu już podpisanych umów.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
getId	Metoda zwracająca numer identyfikacyjny apteki.	void	int
getName	Metoda zwracająca nazwę apteki.	void	String
getDailyNeed	Metoda zwracająca dziennie zapotrzebowanie na szczepionki apteki..	void	int
getCurrent DailyNeed	Metoda zwracająca bieżące dzienne zapotrzebowanie na szczepionki apteki.	void	int
removeFrom CurrentDailyNeed	Metoda odejmująca pewną ilość od bieżącego dziennego zapotrzebowania apteki np. po zawarciu umowy pewnym połączeniem. Zwraca false, jeżeli ta ilość jest większa od pozostałej ilości zapotrzebowania.	int:amount	boolean
@Override toString	Przysłonięta metoda konwertująca obiekt na typ String według szablonu '[Nazwa apteki (id = numer identyfikacyjny apteki)]	void	String
@Override equals	Przysłonięta metoda rozstrzygająca równość dwóch obieków typu Pharmacy tak, aby sprawdzała równość ID lub nazw obu aptek.	void	boolean

Podrozdział 3.8: KLASA 'Supplier'

Klasa stanowiąca reprezentację producenta (in. dostawcy) w programie. Konstruktor przyjmuje numer identyfikacyjny producenta, nazwę producenta oraz ilość dziennej produkcji szczepionek.

- **private int id** - numer identyfikacyjny producenta.
- **private String name** - nazwa producenta.
- **private int dailyProduction** - dzienna produkcja szczepionek.
- **private int currentDailyNeed** - bieżąca dostępna dzienna produkcja szczepionek po odjęciu już podpisanych umów na część.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
getId	Metoda zwracająca numer identyfikacyjny producenta	void	int
getName	Metoda zwracająca nazwę producenta.	void	String
getDailyProduction	Metoda zwracająca dzienną produkcję szczepionek tego producenta.	void	int
getCurrent DailyProduction	Metoda zwracająca bieżącą dostępną dzienną produkcję szczepionek po odjęciu już podpisanych umów na część.	void	int
removeFromCurrent DailyProduction	Metoda odejmująca pewną ilość od bieżącego dziennej produkcji szczepionek np. po zawarciu umowy pewnym połączeniem. Zwraca false, jeżeli ta ilość jest większa od pozostałej ilości dostepnej produkcji.	int:amount	boolean
@Override toString	Przysłonięta metoda konwertująca obiekt na typ String według szablonu '[Nazwa producenta (id = numer identyfikacyjny producenta)]	void	String
@Override equals	Przysłonięta metoda rozstrzygająca równość dwóch obieków typu Supplier tak, aby sprawdzała równość ID lub nazw obu producentów.	void	boolean

Podrozdział 3.9: KLASA 'Transaction'

Klasa stanowiąca reprezentację optymalnego połączenia ustalonego przez algorytm, która jest de facto umową na dostarczenie pewnej ilości szczepionek za pewną cenę danym połączeniem, bazującym na istniejącej klasie **Connection**. Konstruktor przyjmuje nazwę apteki, nazwę producenta, liczbę szczepionek do dostarczenia, koszt pojedynczej szczepionki dostarczonej tą trasą.

- **private String pharmacy** - nazwa apteki w transakcji.
- **private String supplier** - nazwa producenta w transakcji.
- **private int transfer** - ilość szczepionek, która ma zostać dostarczona
- **private double costPerSingleVaccine** - koszt pojedynczej szczepionki dostarczanej tą trasą.
- **private double totalCost** - całkowity koszt tej transakcji, iloczyn transfer · costPerSingleVaccine.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
getSupplier	Metoda zwracająca nazwę producenta.	void	String
getPharmacy	Metoda zwracająca nazwę apteki.	void	String
getTransfer	Metoda zwracająca ilość szczepionek, która ma zostać dostarczona.	void	int
getCostPer SingleVaccine	Metoda zwracająca koszt pojedynczej szczepionki dostarczanej tą trasą.	void	double
getTotalCost	Metoda zwracająca całkowity koszt tej transakcji, iloczyn transfer · costPerSingleVaccine.	void	double
@Override toString	Przysłonięta metoda konwertująca obiekt na typ String zgodnie z wymaganiem biznesowym danych wyjściowych.	void	String
@Override equals	Przysłonięta metoda rozstrzygająca równość dwóch obiektów typu Transaction tak, aby sprawdzała równość nazw producentów i nazw aptek obu transakcji.	void	boolean

Podrozdział 3.10: KLASA 'DistributionAlgorithm'

Klasa zawierająca algorytm tworzący najbardziej ekonomiczne (tanie) połączenia pomiędzy producentami szczepionek, a aptekami. Konstruktor przyjmuje HashMapę producentów, HashMapę aptek i ArrayListę połączeń.

- **private ConnectionsSorter sorter** - obiekt algorytmu sortowania szybkiego.
- **private HashMap<Integer, Supplier> suppliers** - kontener przechowujący producentów.
- **private HashMap<Integer, Pharmacy> pharmacies** - kontener przechowujący apteki.
- **private ArrayList<Connection> connections** - lista dostępnych połączeń.
- **private ArrayList<Transaction> transactions** - lista obliczonych przez algorytm najbardziej opłacalnych transakcji.
- **private ArrayList¡Integer¿ extremeIndexes** - kolejne pary ideksów początkowych i końcowych podwektorów, które mają równe elementy pod względem kosztów szczepionki w liście.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
getTransactions	Metoda zwracająca listę obliczonych transakcji.	void	ArrayList <Transactions>
getConnections	Metoda zwracająca listę połączeń. Funkcja przydatna do testowania, gdy chcemy monitorować zawartość listy połączeń w czasie pracy algorytmu.	void	ArrayList <Connections>
getExtremeIndexes	Metoda zwracająca listę z indeksami początkowymi i końcowymi podwektorów o równych elementach. Funkcja przydatna do testowania, gdy chcemy sprawdzać poprawność metody tworzącej tę listę.	void	ArrayList <Integer>
calculate	Metoda główna, w której wyniku obliczone zostaną najbardziej optymalne transakcje i zapisane w atrybucie transactions. Jej działanie zostało wyjaśnione powyżej.	void	void
sortConnections ByVaccineCost	Metoda prywatna sortująca całą listę połączeń w atrybucie obiektu względem kosztu szczepionek.	void	void
sortEqualPartsOf ConnectionsListBy RealTransfer	Metoda prywatna sortująca kolejne fragmenty całej listy, które są ciągami równych wartości względem kosztu szczepionek. Metoda sortuje te podwektory względem wartości rzeczywistego transferu (def. w opisie klasy).	void	void
findFirstAndLast IndexesOfEqual Parts	Metoda prywatna zapisująca kolejne pary ideksów początkowych i końcowych podwektorów, które mają równe elementy pod względem kosztów szczepionki w liście, do atrybutu klasy extremeIndexes.	void	void
areTheseConnections CostValuesEqual	Metoda prywatna sprawdzająca przyjmująca indeksy połączeń na liście i porównująca ich równość względem kosztu szczepionki.	int:firstIndex, int:secondIndex	boolean

W momencie pisania tej specyfikacji (15.11.2020 r.) ten algorytm nie jest jeszcze w pełni dokończony i wciąż pracuję nad jego optymalizacją oraz skutecznością.

Podrozdział 3.11: KLASA 'ConnectionsSorter'

Klasa zawierająca algorytm sortowania połączeń (używany przez główny algorytm programu) bazujący na sortowaniu szybkim QuickSort. Konstruktor przyjmuje typ wyliczeniowy enum "By", który pełni funkcję stanu obiektu i określa według czego powinien aktualnie sortować listę połączeń (By.COST - według kosztu, By.REALTRANSFER - według realnego możliwego transferu). Oprócz tego wymaga podanie mu odpowiednimi funkcjami HashMapę producentów, HashMapę aptek i ArrayListę połączeń przed rozpoczęciem sortowania.
Klasa implementuje interfejs QuickSortInterface, skąd czerpie umowy na metody publiczne oraz typ wyliczeniowy "By".

- **private HashMap<Integer, Supplier> suppliers** - kontener przechowujący producentów.
- **private HashMap<Integer, Pharmacy> pharmacies** - kontener przechowujący apteki.
- **private ArrayList<Connection> connections** - lista dostępnych połączeń.
- **private By comparator** - typ wyliczeniowy sugerujący, według czego algorytm powinien sortować połączenia. Przyjmuje dwie wartości
  - **By.COST** - według kosztu
  - **By.REALTRANSFER** - według realnego możliwego transferu
- **private int start** - indeks końcowy wektora, na którym kończy się odcinek wektora do posortowania.
- **private int end** - indeks początkowy wektora, od którego zaczyna się odcinek do posortowania.

NAZWA METODY	OPIS METODY	PARAMETRY METODY	ZWRACANA WARTOŚĆ
setSuppliers	Metoda przekazująca algorytmowi HashMapę producentów.	HashMap<Integer, Supplier>: suppliers	void
setPharmacies	Metoda przekazująca algorytmowi HashMapę aptek.	HashMap<Integer, Pharmacy>: pharmacies	void
sortWholeList	Metoda zwracająca całkowicie posortowaną listę połączeń. Wywołuje metodę sortOnlyPartOfList z początkowym i końcowym indeksem całej listy.	ArrayList<Connection>: connections	ArrayList <Connection>
sortOnlyPartOfList	Metoda zwracająca listę połączeń z posortowanym wskazanym po indeksach fragmentem. Sprawdza atrybuty i listę połączeń pod kątem wartości null i uruchamia właściwą funkcję sortującą.	ArrayList<Connection>: connections, int:start, int:end	ArrayList <Connection>
quicksortConnections NonDecreasinglyBy Cost	Właściwa prywatna metoda sortująca. Zwraca posortowaną podaną listę połączeń rosnąco po cenie szczepionki.	void	void
quicksortConnections NonDecreasinglyBy RealTransfer	Właściwa prywatna metoda sortująca. Zwraca posortowaną podaną listę połączeń malejąco po ilości realnie dostępnego transferu.	void	void
splitDataForCost	Prywatna metoda dzieląca wektor o podanych indeksach na podwektory, licząca pivota z mediany. Zwraca rzeczywisty indeks pivotu. Wersja dla sortowania po cenie	int:start, int:end	int
splitDataFor RealTransfer	Prywatna metoda dzieląca wektor o podanych indeksach na podwektory, licząca pivota z mediany. Zwraca rzeczywisty indeks pivotu. Wersja dla sortowania po transferze	int:start, int:end	int
swap	Prywatna metoda zamieniająca elementy listy połączeń o podanych indeksach miejscami.	int:firstId, int:secondId	void
getPivotByMedian AndSetAsFirst ElementForCost	Prywatna metoda licząca wartość pivota na podstawie mediany skrajnych elementów i losowej wartości ze środka. Wersja dla sortowania po cenie	int:start, int:end	int
getPivotByMedian AndSetAsFirst ElementForRealTransfer	Prywatna metoda licząca wartość pivota na podstawie mediany skrajnych elementów i losowej wartości ze środka. Wersja dla sortowania po transferze.	int:start, int:end	int
getRealTransfer Worthiness	Zwraca realną dostępną wartość transferu, czyli najmniejszą wartości z bieżącego zapotrzebowania apteki, dostępnego transferu, dziennej produkcji producenta tego połączenia. Korzysta z metody klasy Connection. Istnieje po to, aby zachować czystość kodu.	Connection:connection	int

Rozdział 4: ANALIZA ALGORYTMÓW I STRUKTUR DANYCH.

Podrozdział 4.1: ALGORYTM SORTOWANIA.

Algorytm sortowania zawarty w klasie ConnectionsSorter jest oparty na algorytmie szybkiego sortowania QuickSort wzbogaconego o wybieranie osi podziału (pivota) poprzez medianę trzech wartości, czyli wartości o pierwszym i ostatnim indeksie w wektorze oraz losowej z pomiędzy nich. Takie rozwiązanie znacznie amortyzuje złożoność algorytmu przy pesymistycznych danych i zmniejsza prawdopodobieństwo na wystąpienie przypadku pesymistycznego, kiedy za oś podziału wybierana jest skrajna wartość co wydłuża proces sortowania. Dzieje się tak, ponieważ QuickSort najlepiej spisuje się, gdy w wyniku podziału wektora powstają dwa jak najbardziej równe podwektory. Zmniejsza to liczbę potrzebnych iteracji. W przypadku ciągłego wybierania skrajnych wartości czas potrzebny na posortowanie listy wzrasta drastycznie.

Złożoność obliczeniowa typowego przypadku jest równa: $O_{(N)} = n \cdot log(n)$, a przypadku pesymistycznym (którego szansa na wystąpienie została znacznie zredukowana) $O_{(N)} = n^2$.

Do rozwiązania problemu sortowania został użyty QuickSort, ponieważ do pracy na takich danych jakie będzie musiał w tym programie sortować nadaje się najlepiej. W czasach pandemii różne firmy chwytają się najbardziej przebiegłych sztuczek, aby dla ogromnych zysków umiejętnie podbić ceny szczepionek. Ich cena zależy od rozmiaru, lokalizacji czy nawet polityki samej firmy. W efekcie najprawdopodobniej do analizy program dostanie listę połączeń o bardzo zróżnicowanych cenach za pojedynczą szczepionkę. Szansa, że wszystkie ceny będą takie same wyklucza sens istnienia tego programu, ponieważ wtedy głębsze rozważania o sposobach ich rozdysponowania są stratą czasu. Natomiast szansa, że dostaniemy dane, w którym lista połączeń będzie tak ułożona, że akurat będą one już "posortowane" według ceny szczepionek oraz przepustowości transferu jednym połączeniem jest tak mała i niesamowicie wyjątkowa, że nie warto jej nawet rozważać.

Podrozdział 4.2: ALGORYTM DYSTRYBUCJI SZCZEPIONEK.

W momencie pisania tej specyfikacji (15.11.2020 r.) ten algorytm nie jest jeszcze w pełni dokończony i wciąż pracuję nad jego optymalizacją oraz skutecznością.

Przebieg algorytmu:

- Posortuj quicksortem listę połączeń rosnąco pod względem kosztu pojedynczej szczepionki.
- Przeszukaj listę połączeń pod kątem połączeń o tych samych cenach. Dla każdego z nich dłuższego od 1 zapisz indeksy początkowe i końcowe.
- Takie duplikaty posortuj quicksortem malejąco pod względem rzeczywistego transferu. Za rzeczywisty transfer przyjmij najmniejszą liczbę z:
  - aktualne zapotrzebowanie apteki danego połączenia
  - aktualnie dostępna liczba szczepionek u producenta danego połączenia
  - aktualna przepustowość połączenia
- Realizuj kolejne połączenia z początku listy w transakcje. W przypadku wyczerpania producenta, połączenia lub zaspokojenia apteki skreślaj z listy wszystkie pozycje je zawierające.
- Znajdź wszystkie niewykorzystane połączenia i utwórz z nich transakcje o ilości przesłanych szczepionek równej 0.
- Zwróć listę transakcji.

Złożoność sortowania szybkiego 'quicksort' wchodzącego w skład tego algorytmu jest równa $O(n \cdot log(n))$. Algorytm jednorazowo sortuje całą tablicę na początku pracy. Następnie zależnie od ilości elementów równych sortuje osobno fragmenty listy. Na przykład, jeżeli wszystkie połączenia mają różne koszty szczepionek, wówczas algorytm nie będzie sortować poszczególnych fragmentów w ogóle. W przypadku, gdy wszystkie połączenia mają tę samą cenę szczepionek, algorytm posortuje całą tablicę od nowa. Ten przypadek jest mało prawdopodobny, ponieważ wówczas głębsze rozważania nad dystrybucją szczepionek byłyby niepotrzebne. Producenci będą mieli różne ceny, a zadaniem tego programu będzie ekonomcizna optymalizacja ich dystrybucji.

Po co taki precedens? Otóż ten algorytm działa na podobnej zasadzie co tzw. problem plecaka. Określa hierarchię pod względem wartości pojedynczych połączeń względem całości, wypychając te najbardziej wartościowe na górę listy kosztem tych mniej wartościowych. Oczywiście, najważniejszą cechą decydującą o wartości danego połączenia jest cena szczepionek dostarczanych tym połączeniem. To połączenia o najmniejszych cenach powinny powędrować na zaszczytne pierwsze miejsca na liście. Co w przypadku, gdy na liście utworzą się odcinki, w których wszystkie połączenia mają tę samą cenę szczepionek? Jak dodatkowo rozróżnić wartość takich połączeń? Wtedy o wartości takiego połączenia decyduje też rzeczywista liczba szczepionek jaką można tym połączeniem dostarczyć. Im jest ona większa, tym bardziej wartościowe z ekonomicznego punktu widzenia jest to połączenie.

Wbrew pozorom liczba szczepionek jaka może zostać dostarczona tym połączeniem nie jest rzeczywistą liczbą szczepionek, jaka może zostać tym połączeniem dostarczona. Od rzeczywistej liczby zależy także aktualna ilość dostępnej dziennej produkcji producenta, od którego wychodzi to połączenie (w końcu nie zmusimy producenta do obowiązkowych nadgodzin). Rzeczywista liczba zależy także od bieżącego zapotrzebowania apteki, które mogło zostać już częściowo lub całkowicie zaspokojone przez inną transakcję. Nadmiar dostarczonych szczepionek nie zostałby zużyty, a one same mogłyby stracić na ważności i ulec zniszczeniu. Zatem rzeczywista liczba szczepionek możliwych do dostarczenia danym połączeniem jest równa najmniejszej wartości z trzech wymienionych. Wyszukanie takiej wartości to koszt rzędu $O(1 \cdot 3)$, czyli $O(1)$ dzięki złożoności oferowanej przez HashMap, w której składowane są informacje o aptekach i dostawcach.

Następnie począwszy od początku listy algorytm podpisywałby umowy na kolejne transakcje, usuwując na bieżąco z listy wykorzystane połączenia lub takie, w których występuje wyczerpany producent lub zaspokojona apteka. Iteracja po całej liście to koszt rzędu $O(n)$. Usuwanie elementów ze struktury ArrayList z pakietu java.util ma złożoność $O(n)$ przy równoczesnej iteracji przez listę i występuje tylko w przypadku wyczerpania któregoś z podmiotów połączenia. Ponieważ połączenia na początku listy mają duże przepustowości transferów oraz na początku producenci są w pełni sił, a apteki niezaopatrzone, to większość takich sytuacji wydarzy się na samym początku pracy algorytmu. Razem usunięcie n aptek lub n producentów (usunięcie wszystkich aptek będzie równoznaczne z usunięciem wszystkich połączeń, co jest równoważne z usunięciem wszystkich producentów) dla n^2 połączeń jest równe $O(n^3)$.

Po wyczyszczeniu listy algorytm sprawdzi, które połączenia zostały całkowicie pominięte i dla formalności biznesowych utworzy z nich transakcje o zerowej ilości dostarczanych szczepionek. Złożoność obliczeniowa tego procesu jest równa $O(n \cdot log(n))$.
Razem: $n \cdot log(n) + O(1) + O(n) + n \cdot n^2 + n \cdot log(n)$, czyli $O(n^3)$.

Podrozdział 4.3: STRUKTURY DANYCH REPREZENTUJĄCE RZECZYWISTE PODMIOTY PROBLEMU.

W programie występują cztery obiekty reprezentujące podmioty problemu. Pierwszym z nich jest **Pharmacy**, reprezentujący placówkę apteki, do której należy dostarczyć szczepionki. Drugim jest **Supplier**, reprezentujący producenta będącego też dostawcą szczepionek. Trzecim jest **Connection**, czyli możliwe połączenie pomiędzy tamtymi dwoma podmiotami wzbogacone o przepustowość takiego

połączenia oraz koszt dostarczania szczepionek tą trasą. Czwartą jest **Transaction**, czyli ustalone i zaakceptowane już połączenie przez algorytm. Od Connection różni się tym, że Transaction nie jest już możliwością, a zleceniem. Connection określiła grupa menadżerska jako możliwą trasę dostawy, a Transaction określiła grupa analityków jako zlecone polecenie dostarczenia pewnej ilości szczepionek za pewną cenę. Wszystkie obiekty mają przysłoniete metody ToString(), equals() oraz zaimplementowany szereg metod, których zadaniem jest upraszczenie wykonywania operacji na nich.

Same obiekty nie wystarczą do rozwiązania problemu. Potrzebne są też kontenery, które będą służyć do przechowywania ich zbiorów. Oczywiście, można byłoby po prostu spakować je wszystkie do ogromnych tablic, które przeszukiwalibyśmy pętlami, ale tutaj liczy się prędkość. Dlatego należy się zastanowić, czego będziemy oczekiwać od takiego kontenera przechowującego dany typ obiektów.

- Zacznijmy od **Pharmacy**. Nie potrzebujemy sortować naszych aptek, a dodawać je do naszego kontenera będziemy tylko na początku programu w momencie wyczytywania. Będziemy natomiast często odwoływać się do tych obiektów. Będziemy pobierać i modyfikować dane. Nie zależy nam na ich kolejności, ale pojedynczych wartości, których będziemy w danych momentach potrzebować. Obiektem spełniającym te wymagania biznesowe jest HashMapa z pakiet java.util. Złożoność obliczeniowa pobrania i modyfikacji danej w niej się znajdującej jest równa $O(1)$. Jako zestaw kluczy reprezentujące obiekty aptek użyjemy ich unikalnych numerów identyfikacyjnych.

- Przejdźmy do **Supplier**. Jest to praktycznie identyczny przypadek jak przy obiektach Pharmacy. Będziemy dodawać dane tylko na początku programu, bardzo często pobierać dane i równie często modyfikować je. Nie zależy nam na ich kolejności, ale pojedynczych wartości, których będziemy w danych momentach potrzebować. Użyjemy więc struktury HashMap z pakietu java.util, której złożoność obliczeniowa tych operacji jest równa $O(1)$.

- Co w przypadku obiektu **Connection**? Tutaj kolejność będzie mieć ogromne znaczenie. Będziemy je też sortować i na pewno niejednokrotnie przeiterujemy po kolei przez zawartość tego kontenera w poszukiwaniu najbardziej optymalnych kombinacji. Do tego celu użyjemy kontenera ArrayList z pakietu java.util. Operacje, które będziemy wykonywać na obiektach tego kontenera mają złożoność $O(N)$, czyli np. wyszukanie, usunięcie bądź modyfikacja wartości po indeksie. Umieszczenie elementu nowego elementu lub pobranie już istniejącego na podstawie znanego indeksu ma złożoność równą $O(1)$, co jest bardzo przychylne przeznaczeniu tego kontenera.

- Pozostał tylko obiekt **Transaction**. Tutaj mamy nieco więcej swobody przy wyborze odpowiedniego kontenera. Wykonamy na nim tylko dwie operacje. Pierwsza to operacja umieszczenia nowego elementu (wraz z postępami pracy algorytmu) oraz na sam koniec przeiterujemy po każdym jego obiekcie celem wypisania wszystkich transakcji do pliku z danymi wyjściowymi. Do tego celu użyjemy ponownie ArrayList z java.util. Koszt umieszczenia nowego elementu na końcu listy wynosi $O(1)$, a iteracja po każdym elemencie $O(N)$.

Rozdział 5: TESTOWANIE.

Podrozdział 5.1: UŻYTE NARZĘDZIA.

Testowanie programu nie wymaga zaangażowania osób trzecich. Do testowania wewnętrznej logiki programu posłużą testy jednostkowe dołączone do programu oraz folder z zestawem testowych plików wejściowych, zawierających różne konfiguracje błędów, które wystarczy podawać programowi i sprawdzać, czy jego reakcja jest zgodna z założoną. Testy jednostkowe zostały wykonane przy użyciu **JUNIT**.

Podrozdział 5.2: KONWENCJA.

1. Testy jednostkowe powinny być napisane zgodnie z zasadami TDD.
    - Każdy test powinien testować jedną rzecz.
    - Kolejność wykonywania testów nie powinna mieć znaczenia.
    - Wynik testów nie powinien zależeć od środowiska uruchomieniowego.
    - Pamiętaj o sprawdzaniu warunków brzegowych i sytuacji wyjątkowych.
    - Oddzielaj kod testujący od kodu produkcyjnego.
2. Należy używać intuicyjnych nazw oraz **nie należy się bać długich nazw funkcji**. Liczy się zrozumiałość i szybka diagnoza problemu w przypadku zapalenia czerwonej lampki przy niej po uruchomieniu testów.
3. Używaj przedrostków @Begin oraz @After, aby odchudzić funkcje testujące.
4. Lepiej napisać 10 porządnych testów jednostkowych niż setkę średnich testów, mających rozmiar 10-krotnie większy niż kod samego programu.
5. Pamiętaj o posprzątaniu zainicjalizowanych obiektów po zakończeniu testów.
6. Uruchamiaj algorytm dla każdego z przykładowych danych wejściowych w folderze sampleInputFilesForTesting. Nie bój się dodawać własnych plików, jeżeli wpadniesz na nowy pomysł sytuacji wyjątkowej.

Podrozdział 5.3: WARUNKI BRZEGOWE.

Warunki brzegowe i zbiór wrażliwych miejsc, które należy przetestować:

1. InputDataReader:
    - Nie podanie żadnych danych.
    - Podanie nieprawidłowych danych wejściowych, czyli pliku:
        - który nie istnieje,
        - w którym nie zgadza się ilość argumentów w jednej linii, w którejś sekcji.
        - którego argumenty oddzielane są w inny sposób niż " | ".
        - w którego jednym z wierszy jest za dużo argumentów.
        - w którego jednym z wierszy jest za mało argumentów.
        - w którym sekcje są nie po kolei (właściwa kolejność: producenci, apteki, połączenia).
        - który nie zawiera jednej sekcji.
        - który nie zawiera ostatniej sekcji.

- który zawiera zduplikowane sekcje.
- który zawiera zduplikowane rekordy dla połączeń (powinien pominąć, nie kończąc pracy).
- który zawiera zduplikowane rekordy dla aptek (powinien pominąć, nie kończąc pracy).
- który zawiera zduplikowane rekordy dla producentów (powinien pominąć, nie kończąc pracy).
- w którym jeden z wierszy zawiera ujemne argumenty numeryczne (np. cenę szczepionki, która nie może być ujemna z punktu logicznego).
- w którym jeden z wierszy zawiera zmiennoprzecinkową liczbę transferu, zapotrzebowania lub produkcji (Nie można wyprodukować 10 i pół szczepionki).
- w którym jeden z wierszy zawiera słowo zamiast liczby (np. słownie napisane 10 "dziesięć").
- który zawiera połączenie do apteki, która nie istnieje.
- który zawiera połączenie do producenta, który nie istnieje.

2. OutputDataWriter:

   - Reakcja programu, gdy chce utworzyć folder z danymi wyjściowymi, gdy taki już istnieje.
   - Reakcja programu, gdy chce utworzyć plik z danymi wyjśćiowymi, który już istnieje.
   - Prześledzenie wyjściowego pliku pod kątem zgodności z wymaganiami biznesowymi (przyjętym szablonem).
   - Pewność, że każde wejściowe połączenie zostało zapisane także tutaj, nawet jeżeli z wartością ilości szczepionek równą 0.

3. ConnectionSorter:

   - Przyjęta lista połączeń jest pusta.
   - Przyjęta lista aptek jest pusta.
   - Przyjęta lista producentów jest pusta.
   - Przyjęta lista połączeń zawiera tylko jeden element.
   - Przyjęta lista aptek zawiera tylko jeden element.
   - Przyjęta lista producentów zawiera tylko jeden element.
   - Weryfikacja, czy aby na pewno lista jest posortowana rosnąco po koszcie pojedynczej szczepionki, a połączenia o tym samym koszcie malejąco po ilości realnego transferu (najmniejszej wartości z bieżącego zapotrzebowania apteki, bieżącego dostępnego transferu, pozostałej dziennej produkcji producenta).
   - Należy przetestować, czy sortowanie tylko wydzielonego fragmentu listy wpływa w jakikolwiek sposób na pozostałą część.

4. SafetyGuard:

   - Weryfikacja każdej z metod poprzez podawanie jej celowo nieprawidłowych argumentów.
   - Testowanie metod sprawdzających poprawność składni linii pliku zgodnie z testowaniem InputDataReadera.
   - Weryfikacja znajdowania duplikatów poprzez tworzenie szeregu podobnych obiektów do obiektu kontrolnego, różniących się maksymalnie jednym kluczowym parametrem.

- Weryfikację wydajności połączeń dla danej apteki należy testować tworząc trzy zbiory połączeń i producentów:
  - W pierwszym producenci powinni być w stanie zaopatrzyć aptekę, ale przepustowość połączeń nie powinna na to pozwalać.
  - W drugim producenci powinni mieć niewystarczające produkcje do zaopatrzenia apteki, ale przepustowość połączeń powinna być wystarczająca.
  - Trzecia to grupa kontrolna, gdzie zarówno producenci jak i apteki mają wystarczające moce, aby zaopatrzyć daną aptekę.

Rozdział 6: DIAGRAM KLAS.