Specyfikacja implementacyjna projektu grupowego z przedmiotu *Algorytmy i struktury danych*

Krzysztof Kowalski, Szymon Motłoch, Aleksander Wodnicki

17 grudnia 2020

Spis treści

1 Wstęp

1.1 Cel dokumentu

Niniejszy dokument przybliża informacje związane z projektem grupowym z przedmiotu *Algorytmy i struktury danych* od strony programistycznej i opisuje strukturę poszczególnych klas i przedstawia zawarte w nich metody. Znajdują się tu również informacje dotyczące sposobu wersjonowania kodu źródłowego, a także informacje o testach jednostkowych oraz wykorzystany algorytm.

Użytkownik, po zapoznaniu się z tym dokumentem, powinien być w stanie napisać na jego podstawie działający program, który zrealizuje założenia projektowe.

1.2 Cel projektu

Celem projektu, jak to zostało opisane również w specyfikacji funkcjonalnej, jest stworzenie programu, który pomoże rozwiązać problem opisany w dziale o tej samej nazwie, we wspomnianej specyfikacji funkcjonalnej. Program, po uruchomieniu i otrzymaniu pliku z danymi o szpitalach, drogach między nimi oraz o obiektach na mapie, analizuje go i wyświetla mapę na graficznym interfejsie. Program otrzymuje również plik z danymi o miejscu pobytu pasażerów lub też otrzymuje te dane z poziomu interfejsu graficznego, po podaniu ich przez użytkownika w przeznaczonych do tego opisanych polach. Znając te informacje, program będzie pokazywać trasę karetki pogotowia od momentu odbioru pacjenta, przez przejście kolejnych, najbliższych szpitali, aż do momentu znalezienia się szpitala, w którym są wolne łóżka lub dotarcia do ostatniego nieodwiedzonego szpitala.

1.3 Środowisko pracy

Praca nad programem będzie odbywała się na 64-bitowym systemie *Windows 10 Home* (wersja *2004*, build *19041.685*) w programie *Apache NetBeans IDE* (wersja *11.3 LTS*) w *JDK 8*.

1.3.1 Parametry stacji roboczej

Poniżej opisane zostały podzespoły urządzeń, na których opracowany jest projekt:

1. Laptop 1:
    - model: HP Pavilion Gaming 15-cx0006nw
    - procesor: Intel Core i5-8300H (2.3 GHz, 4.0 GHz Turbo, 8 MB Cache)
    - pamięć RAM: DDR4 (2666 MHz) 8 GB
    - dyski twarde: SSD 240 GB; HDD 1000 GB
    - karta graficzna: Nvidia Geforce GTX 1050 4 GB
    - system operacyjny: Microsoft Windows 10 Home.
2. Laptop 2:
    - model: Latitude E5250
    - procesor: Intel Core i5-5300U CPU @ 2.30GHz 2.29GHz
    - pamięć RAM: 8 GB
    - dyski twarde: Samsung SSD 850 EVO 1TB
    - karta graficzna: Intel(R) HD Graphics 5500
    - system operacyjny: Microsoft Windows 10 Pro.
3. Laptop 3:
    - model: Acer Aspire 7
    - procesor: Intel Core i5-7300U CPU @ 2.50GHz 3.5GHz
    - pamięć RAM: 16 GB
    - dyski twarde: SSD 500 GB; HDD 1000 GB
    - karta graficzna: Nvidia Geforce GTX 1050 2 GB
    - system operacyjny: Microsoft Windows 10 Home.

1.3.2 Kompilator i język programowania

Jak opisano wcześniej, językiem programowania używanym przy tym projekcie, będzie język *Java*. Kompilacja odbędzie się dzięki programowi *javac*, a uruchamianie za pomocą programu *java*. Alternatywą dla tego rozwiązania jest skorzystanie ze środowiska programistycznego (np. `NetBeans`) w celu kompilacji oraz uruchomienia programu.

2 Sposób wersjonowania

Wszystkie zmiany w kodzie źródłowym będą rejestrowane w zdalnym repozytorium, które znajduje się po wejściu w poniższy link:

`https://projektor.ee.pw.edu.pl/repo/git/2020Z_AISD_proj_zesp_GR3_gr10`

Wersjonowanie zaś będzie realizowane zgodnie z poniższym schematem:

`X.Y.Z`

`X` oznacza etap rozwoju programu (0 – faza początkowa, pojedyncze metody, 1 – podstawowa funkcjonalność etc.), `Y` to kolejne znaczne poprawki wprowadzone do kodu źródłowego, a `Z` to ewentualne drobne poprawki (np. literówki, usunięcie komentarzy, zmiana nazwy zmiennej). Zmiany w kodzie będą dokonywane na gałęzi `master` - każdy *push*, oprócz kolejnej wersji kodu, dostarczy również komentarz i tag z wersją tego kodu. Zostanie również utworzony plik w formacie `.txt`, w którym to zostaną opisane zmiany wprowadzone w danej wersji.

3 Algorytm

Algorytmem, który zostanie wykorzystany przy realizacji omawianego projektu będzie przykład algorytmu zachłannego – `algorytm Dijkstry`.

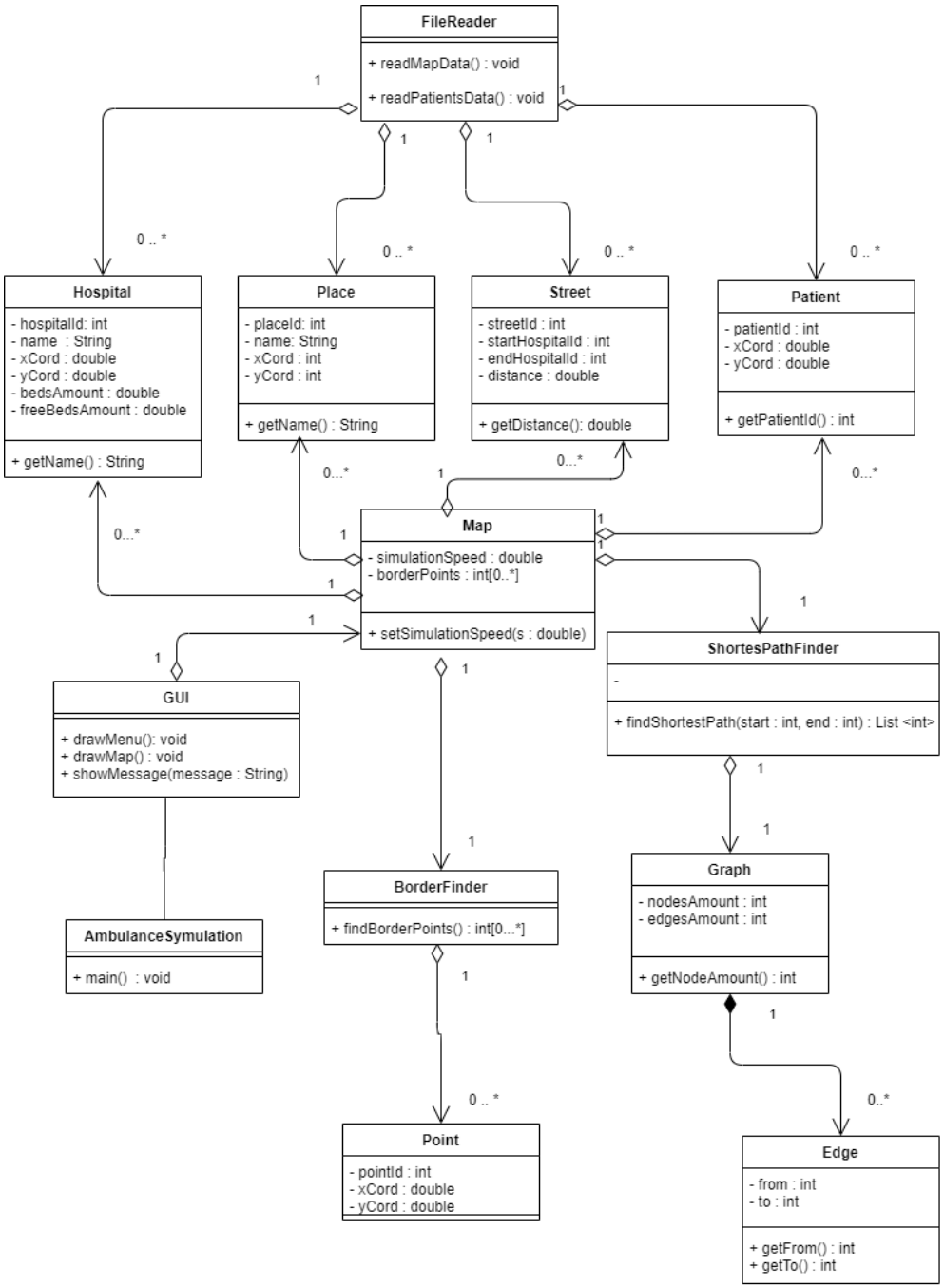
Algorytm ten rozwiązuje jeden z klasycznych problemów w teorii grafów związany ze znajdowaniem ścieżki między dwoma wierzchołkami, w tym przypadku ścieżki najkrótszej. Działa on wtedy, gdy mamy do czynienia z krawędziami o wartościach nieujemnych, tak że w przypadku tego problemu będzie on najbardziej odpowiedni, gdyż odległość między szpitalami nie może być ujemna.

Algorytm Dijkstry znajduje w grafie wszystkie ścieżki od wierzchołka początkowego do docelowego oraz wylicza koszt ich przejścia, czyli w naszym przypadku przejechaną przez karetkę odległość. W ten sposób pomaga on wyznaczyć ścieżkę najkrótszą, która jest niezbędna do realizacji założeń projektowych.

Złożoność tego algorytmu jest zależna od wykorzystanych struktur oraz od liczby V wierzchołków i krawędzi E w danym grafie. W przypadku skorzystania ze zwykłych tablic, złożoność określimy jako $O(V^2)$, zaś w przypadku kopca, ten sam algorytm będzie określony za pomocą złożoności $O(ElogV)$. Pierwsza z nich jest najbardziej optymalna w przypadku grafów gęstych, gdzie liczba krawędzi jest większa od liczby wierzchołków, druga najbardziej optymalna w przypadku przeciwnym, kiedy to liczba wierzchołków jest większa od krawędzi tego grafu.

4 Diagram klas

Poniżej został przedstawiony diagram klas programu:



Rysunek 1: Diagram klas

4.1 Opis klas

Poniżej przestawiono listę klas z powyższego diagramu, dodatkowo zamieszczono krótki opis każdej z nich:

- **AmbulanceSymulation** - klasa zawierająca metodę `main`, która uruchamia wszystkie elementy programu;
- **FileReader** - klasa służąca do wczytywania danych wejściowych z pliku;
- **Hospital** – klasa reprezentują dane szpitali;
- **Place** – klasa reprezentująca dane obiektów;
- **Street** – klasa reprezentująca dane dróg łączących szpitale;
- **Patient** – klasa reprezentująca dane pacjentów;
- **Graph** – klasa reprezentująca grafy;
- **Edge** – klasa reprezentująca krawędzie grafu;
- **ShortesPathFinder** – klasa umożliwiająca wyznaczanie najkrótszej ścieżki między dwoma punktami na postawie dostarczonego spisu dróg;
- **BorderFinder** – klasa umożliwiająca znalezienie granic państwa na bazie istniejących szpitali i obiektów;
- **Map** – klasa przechowująca wszystkie dane związane ze stanem mapy;
- **GUI** – klasa generująca interfejs użytkownika;
- **Point** – klasa reprezentująca punkt na mapie ze współrzędnymi X i Y.

5 Testowanie

Jak przedstawiono w specyfikacji funkcjonalnej dla tego projektu, testowanie będzie odbywało się z wykorzystaniem testów jednostkowych *JUnit 5*, dodatkowo również napisane zostaną testy jednostkowe, w których skorzystamy z *AssertJ* – te będą stanowiły większą część testów. Wszystkie testy zostaną przygotowane tak, aby przetestować możliwie jak najwięcej metod, ale także plików w formacie `.txt`, w celu sprawdzenia reakcji programu na nie. Testy jednostkowe zostaną umieszczone w klasach testowych odpowiednich dla testowanych metod (np. metoda x pochodzi z klasy y, więc klasa testowa będzie nosiła nazwę $yTest$).

References

[1] ISOD — Wirtualny Dziekanat:
`isod.ee.pw.edu.pl`

[2] Wikipedia:
`https://pl.wikipedia.org/wiki/Algorytm_Dijkstry`
`https://pl.wikipedia.org/wiki/Graf_(matematyka)`
`https://mattomatti.com/pl/a0123`
`https://eduinf.waw.pl/inf/alg/001_search/0138.php`