SPECYFIKACJA IMPLEMENTACYJNA DLA PROJEKTU INDYWIDUALNEGO

Wykonała: Agata Lachowiecka
Sprawdzający: mgr inż. Paweł Zawadzki
Data: Marzec 2021

Spis treści

1 Wstęp

1.1 Cel dokumentu

Celem dokumentu jest przedstawienie sposobu implementacji aplikacji desktopowej **„aMAZEing”** wykonywanej w celu realizacji przedmiotu Projekt Indywidualny. Zaprezentowane zostaną środowisko deweloperskie, struktura programu i sposób przechowywania danych, niezbędne algorytmy oraz sposoby testowania. Opis działania i obsługi programu został zaprezentowany w *Specyfikacji funkcjonalnej*.

1.2 Cel projektu

Celem projektu jest stworzenie aplikacji desktopowej, która wykorzystuje kontenerową bazę danych.

2 Środowisko deweloperskie

2.1 System operacyjny

Podczas pracy nad programem używanym systemem operacyjnym będzie **Windows 10 Pro** wersja 2004 kompilacja 19041.867.

2.2 Język programowania

Kod źródłowy programu będzie napisany w języku **Java** w wersji 13.0.2. Dodatkowo wykorzystana zostanie technologia **JavaFX**.

2.3 Środowisko programistyczne

Do pracy nad programem wykorzystane zostanie zintegrowane środowisko programistyczne **IntelliJ IDEA Community Edition 2020.3.3.**

3 Zasady wersjonowania

Do przechowywania kodu źródłowego programu oraz jego dokumentacji wykorzystywane będzie repozytorium wydziałowe w **Projektorze,** a systemem kontroli wersji będzie **Git.** W repozytorium znajdować się będą dwa katalogi. W pierwszym z nich o nazwie „dokumentacja” przechowywane będą dokumenty związane z projektem, natomiast katalog „program” będzie

przechowywał pliki z kodem źródłowym aplikacji, skrypty oraz testy jednostkowe. Dodawanie i aktualizowanie plików będzie opatrywane odpowiednimi komentarzami opisującymi zmiany.

4 Przechowywanie danych

4.1 Baza danych

Do realizacji bazy danych wykorzystany zostanie system relacyjnych baz danych **PostgreSQL** oraz oprogramowanie służące do realizacji konteneryzacji **Docker**. Baza danych przechowywana będzie w kontenerze Dockera, i za jego pośrednictwem aplikacja będzie się z nią łączyć. W celu zarządzania bazą danych wykonywane będą na niej operacje DML (instrukcje manipulowania danymi), DDL (instrukcje definiujące) oraz DCL (instrukcje sterujące uprawnieniami). Natomiast połączenie z aplikacją wykonane będzie za pomocą czterech podstawowych operacji CRUD – utworzenie/dodanie nowych informacji, odczytanie/wyświetlenie istniejących danych, modyfikowanie/edycja danych, usuwanie istniejących informacji. Baza danych służyć będzie do stworzenia i przechowywania rankingów wyników czasowych graczy dla poszczególnych poziomów trudności gry.

4.2 Struktury danych

W programie zostaną wykorzystane następujące struktury danych:

- **tablica statyczna** – zbiór danych tego samego typu o ustalonym rozmiarze;
- **ArrayList** z Java Collection – zbiór danych uporządkowanych liniowo.

4.3 Zasoby

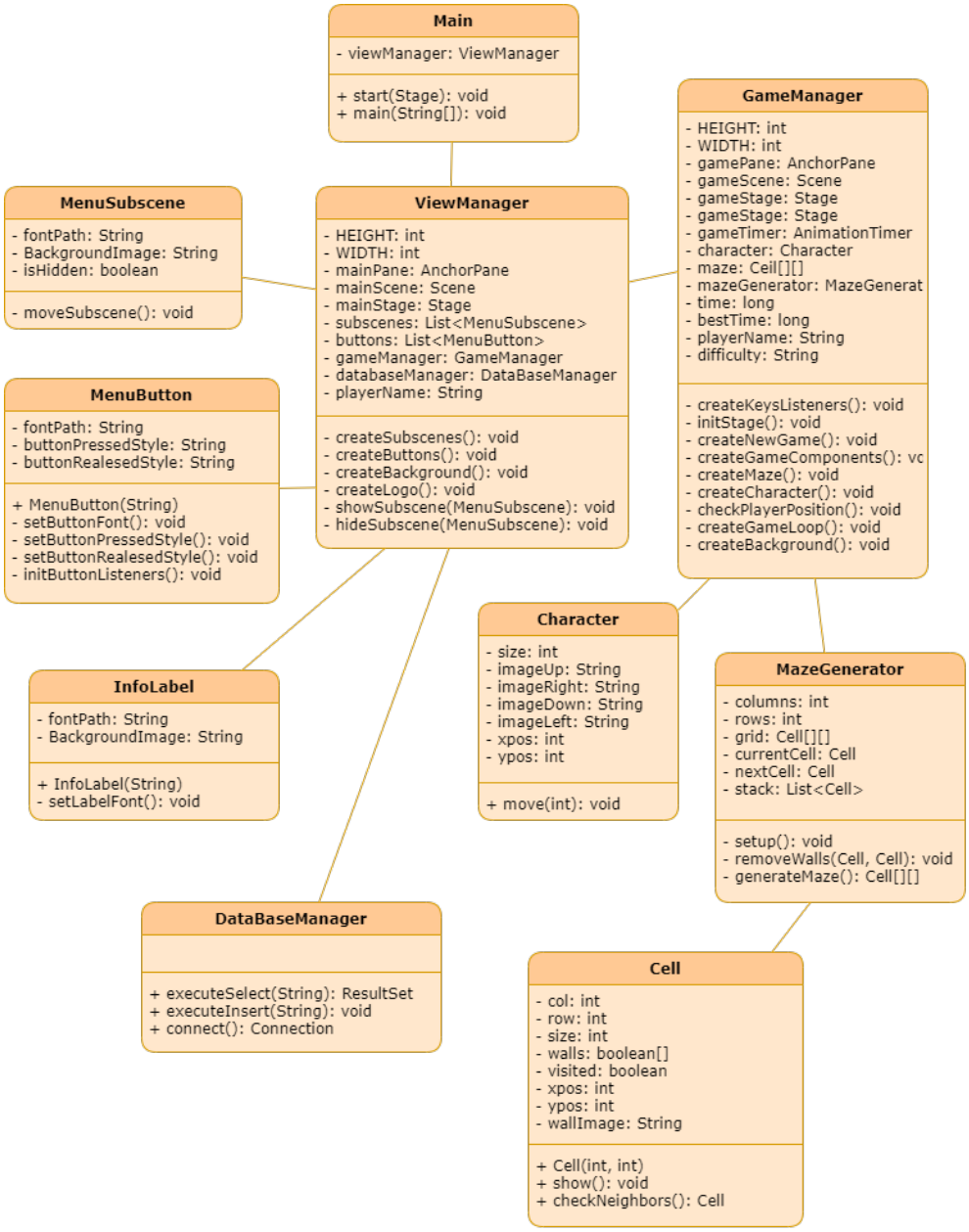
Do realizacji graficznych elementów wykorzystane zostaną **obrazy PNG**, które umieszczane będą w katalogu *resources*.

5 Opis pakietów i klas

Pliki klas zwierające kod źródłowy programu zostaną zgrupowane w 4 pakiety. Pierwszy będzie zawierał klasę główną Main, drugi klasy odpowiadające za komponenty GUI, w trzecim umieszczone będą klasy zarządzające oknami programu i połączeniem z bazą danych, a czwarty przechowywał będzie komponenty gry. Aplikacja będzie miała następującą strukturę:

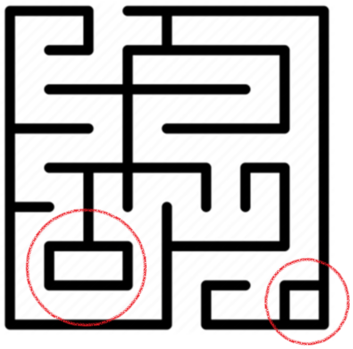
- **main**
  - **Main** – klasa główna rozszerzająca klasę Application, w niej będzie uruchomienie aplikacji.
- **manager**
  - **ViewManager** – klasa odpowiadająca za okno menu głównego, zarządzać będzie przyciskami i polami pojawiającymi się po ich naciśnięciu.
  - **GameManager** – klasa zarządzająca oknem z grą w labirynt, odpowiadać będzie również za interakcję z graczem.
  - **DataBaseManager** – klasa zarządzająca połączeniem z bazą danych, obsługiwać będzie pobieranie informacji oraz ich zapis.
- **guiComponents**
  - **MenuButton** – klasa rozszerzająca Button, służy do tworzenia przycisków menu.
  - **MenuSubscene** – klasa rozszerzająca Subscene, będzie modelem pola pojawiającego się po naciskaniu przycisków.
  - **InfoLabel** – klasa rozszerzająca Label, za jej pomocą umieszczane będą napisy.
- **gameComponents**
  - **MazeGenerator** – klasa służąca do generacji labiryntu o odpowiedniej wielkości.
  - **Character** – klasa reprezentująca postać, którą gracz będzie sterował.
  - **Cell** – klasa odzwierciedlająca komórkę siatki labiryntu.

6 Diagram klas



7 Algorytmy

Podstawowym problemem algorytmicznym w programie jest generowanie losowych prostokątnych labiryntów o różnych wielkościach. Konieczne jest aby istniała ścieżka prowadząca z wejścia do wyjścia. Można to zapewnić poprzez warunek istnienia drogi między wszystkimi dowolnymi dwoma punktami w labiryncie – aby nie istniały odseparowane fragmenty takie jak na Rysunku 1.



Rysunek 1: Na czerwono zaznaczone są niewłaściwe fragmenty labiryntu

W celu stworzenia takiego labiryntu wykorzystany zostanie **algorytm randomizowanego przeszukiwania w głąb.** Labirynt potraktowany zostanie jako siatka komórek, z których każda ma cztery ściany. Najpierw losowo wybierana jest komórka początkowa znajdująca się na krawędzi siatki. W kolejnych krokach w sposób również losowy wybierana jest komórka sąsiadująca, która nie została jeszcze odwiedzona i usuwana jest ściana między nimi. Nowa komórka zostaje oznaczona jako odwiedzona i dodana do stosu. Proces przechodzenia do kolejnych sąsiednich komórek jest kontynuowany dopóki nowa komórka ma nieodwiedzonych sąsiadów. W przypadku ich braku następuje cofanie się według komórek pobieranych ze stosu aż nie pojawi się komórka z nieodwiedzonym sąsiadem, kiedy następuje ponowne generowanie dalszej ścieżki. Proces ten trwa do momentu odwiedzenia każdej komórki, co zasygnalizowane będzie cofnięciem się aż do komórki początkowej.

Mechanizm ten można zrealizować rekurencyjnie lub iteracyjnie z jawnym stosem. W przypadku tej aplikacji zostanie wykorzystany wariant iteracyjny, a przebieg algorytmu wyglądać będzie następująco:

1. Wybranie początkowej komórki z brzegu siatki, oznaczenie jej jako od-

wiedzonej i umieszczenie na stosie.

2. Dopóki stos nie jest pusty:
   (a) Zdjęcie komórki ze stosu i ustawienie jako bieżącej.
   (b) Jeśli bieżąca komórka ma sąsiadów, których nie odwiedzono:
      i. Umieszczenie bieżącej komórki na stosie.
      ii. Wybranie jednego z nieodwiedzonych sąsiadów.
      iii. Usunięcie ściany między bieżącą komórką a wybranym sąsiadem.
      iv. Oznaczenie wybranego sąsiada jako odwiedzonego i umieszczenie go na stosie.

Wyjściem labiryntu będzie, podobnie jak wejściem, losowa komórka znajdująca się na brzegu siatki, jednak w tym losowaniu nie będą brane pod uwagę komórki znajdujące się na tym samym boku prostokąta, co komórka będąca wejściem.

8 Testowanie

Sposób testowania aplikacji został opisany w *Specyfikacji funkcjonalnej*. Poprawność działania będzie głównie weryfikowana poprzez uruchamianie programu i obserwowanie jego zachowania w różnych sytuacjach. W razie potrzeby przetestowania fragmentów kodu stworzone zostaną odpowiednie testy jednostkowe.

8.1 Narzędzia

Do tworzenia testów jednostkowych wykorzystywana będzie przeznaczona do tego biblioteka **JUnit 4**.

8.2 Konwencja

Klasy testujące będą umieszczane w pakiecie testującym. W przypadku potrzeby stworzenia większej liczby takich klas, pakiet ten zostanie podzielony na mniejsze podpakiety o odpowiednich nazwach. Metody testujące będą nazywane w sposób informujący o sprawdzanej sytuacji i oczekiwanym rezultacie.

Na przykład: *shouldThrowExceptionWhenDataIsInvalid(){}*

8.3 Warunki brzegowe

Testom podlegać będą niepożądane zdarzenia opisane w specyfikacji funkcjonalnej w sekcji *3.2 Sytuacje wyjątkowe*. Szczególna uwaga zwrócona będzie na mechanizm poruszania się postacią po labiryncie i poprawny pomiar czasu przechodzenia gry. Elementy te będą sprawdzane poprzez uruchamianie gry i ręczne testowanie zachowania się aplikacji. Kolejnym ważnym do weryfikacji aspektem jest algorytm generowania poprawnego labiryntu o odpowiedniej wielkości. Aby był uznany za poprawny musi tworzyć labirynty zawierające wszystkie komórki siatki, a między każdą dowolną parą komórek musi istnieć ścieżka. Ponieważ w algorytmie tym iteracje będą wykonywane dopóki stos nie będzie pusty, to ważne jest również aby upewnić się, że algorytm skończy działanie – stos zostanie opróżniony.

9 Źródła

- grafika labiryntu:
  https://www.clker.com/clipart-818473.html
- algorytm generowania labiryntu:
  https://pl.qaz.wiki/wiki/Maze_generation_algorithm