Specyfikacja implementacyjna

Projekt zespołowy

Mateusz Smoliński Łukasz Knigawka

23 grudnia 2018

Spis treści

1 Wstęp

Celem projektu jest napisanie programu prezentującego interaktywną mapę na podstawie danych z pliku wejściowego. Program otrzymuje punkty tworzące kontur mapy, punkty kluczowe, na podstawie których kontur zostaje podzielony na obszary, a także definicje obiektów oraz listę takowych obiektów, które mają być brane pod uwagę przy analizie wyżej wspomnianych obszarów.

Jeśli dane będą podane prawidłowo, program na ich podstawie generuje planszę, która dalej może być modyfikowana przez użytkownika. Może on dodawać i usuwać elementy konturu, zmieniając kształt mapy, a także dodawać i usuwać punkty kluczowe, wpływając tym samym na podział na obszary. Po kliknięciu na jeden z punktów kluczowych użytkownik otrzyma informacje o obiektach znajdujących się na wyznaczonym przez niego obszarze.

Projekt „LUPA" zostanie napisany w języku C# wersji 7.3 (.NET Framework 4.7.2), w środowisku Microsoft Visual Studio 15.9.4. Interfejs użytkownika zostanie utworzony dzięki platformie WPF. Implementacja oraz testowanie programu odbywać się będą na komputerach o następujących parametrach:

- 64-bitowy 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,

oraz:

- 64-bitowy system operacyjny Windows 10 Home ver. 1803,
- procesor Intel Core i7-6700HQ,
- pamięć RAM 16,00 GB,
- karty graficzne Intel HD Graphics 530 + NVIDIA GeForce 940MX,

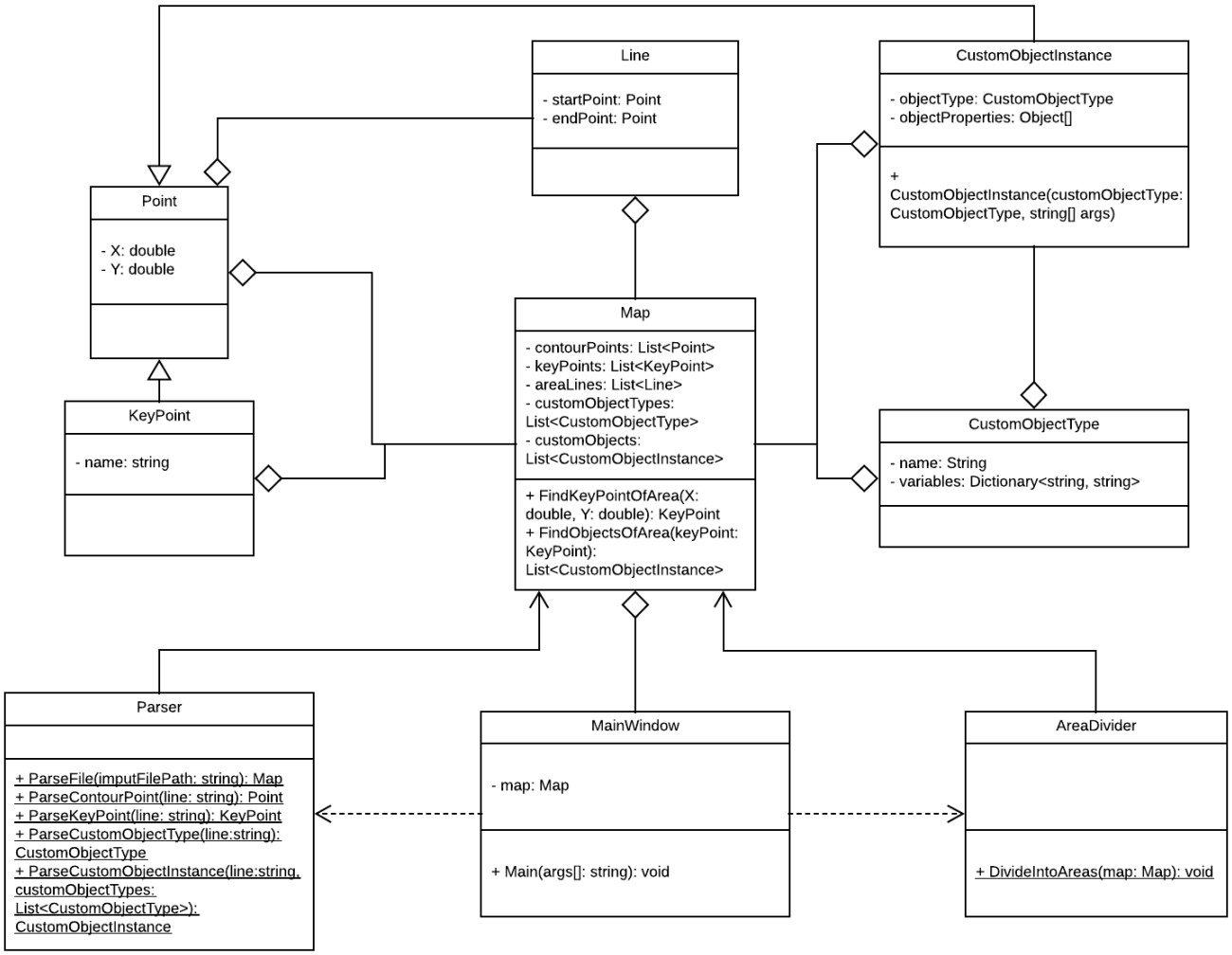
2 Diagram klas

Program składać się będzie z 9 klas. Zależności pomiędzy poszczególnymi klasami obrazuje Rysunek 1.

Na diagramie klas nie zostały uwzględnione:

- zależności pomiędzy klasami Parser, MainWindow, AreaDivider oraz klasami będącymi składowymi klasy Map, w celu poprawy czytelności,
- metody wygenerowane przez WPF, odpowiadające za rysowanie na planszy oraz reakcję na zachowanie użytkownika.

Nie występują na nim także metody dostępowe ze względu na zastosowanie automatycznie generowanych właściwości klas występujące w wybranym przez nas środowisku.



Rysunek 1: Diagram klas

3 Opis klas/metod

3.1 Klasa MainWindow

Jest to bazowa klasa tego programu, odpowiadająca przede wszystkim za interfejs graficzny. Składa się ona z wielu metod wygenerowanych przez WPF, odpowiadających za reakcję na działania użytkownika. Wśród najważniejszych logicznych implementacji, które muszą być zawarte w implementacjach tych metod, warto wspomnieć o:

- metodzie, która po zbliżeniu kursora w okolicę punktu kluczowego wypisze na panelach bocznych listy obiektów znajdujących się na obszarze wyznaczonym przez ten punkt kluczowy (pogrupowane i niepogrupowane według typu) oraz liczbę mieszkańców na tym obszarze,
- metodzie, która przy wybranej opcji punktu kluczowego po kliknięciu lewym przyciskiem myszy dodaje punkt kluczowy, a prawym przyciskiem usuwa najbliższy punkt kluczowy,
- analogicznej metodzie która dodaje i usuwa elementy konturu po wybraniu opcji konturu na pasku górnym.

Po wybraniu opcji usuwania/dodawania punktów na mapie następuje wywołanie funkcji *DivideIntoAreas* ponownie dzielącej mapę na obszary, z klasy *AreaDivider*.

3.2 Klasa Parser

Ta klasa odpowiada za wczytanie danych z pliku tekstowego, które zostaną użyte w trakcie działania programu. Wszystkie metody w tej klasie są statyczne, przez co obiekt klasy Parser nie będzie powstawać w żadnym momencie trwania programu. Metody te w przypadku niepowodzenia korzystają z wyjątków, które przekazują do metody nadrzędnej – metoda *ParseFile* przekazuje te wyjątki do klasy *MainWindow*, która odpowiada za przekazanie komunikatu o błędzie użytkownikowi.

3.2.1 Metoda ParseFile

- `Map ParseFile(string imputFilePath)`

Metoda *ParseFile* jest główną metodą tej klasy. Otrzymuje ona ścieżkę do pliku w postaci napisu, po czym ma za zadanie otworzyć ten plik, przeanalizować każdą jego linię oraz zapisać dane w postaci zrozumiałej dla programu. Na podstawie składni pliku oraz linii oddzielających etapy wczytywania, oznaczonych # na początku, odczytuje linię tekstu i przekazuje ją do odpowiedniej metody parsującej. Po zakończeniu kolejnego etapu metoda sprawdza, czy żaden z warunków działania programu nie został złamany – na przykład, czy użytkownik nie zamieścił przecinających się linii konturu lub nie zapomniał o którymś z obowiązkowych typów obiektów. Odczytane w ten sposób dane zapisuje do list w obiekcie klasy *Map*, po czym w przypadku powodzenia zwraca referencję do tak przygotowanego obiektu.

3.2.2 Metoda ParseContourPoint

- `Point ParseContourPoint(string line)`

Jest to pierwsza ze szczegółowych metod tej klasy, odpowiadających za odczytanie konkretnych elementów podanych w pliku. Ta konkretna metoda odczytuje linię zawierającą współrzędne punktów konturowych, po czym w przypadku powodzenia zwraca przygotowany obiekt klasy *Point*.

3.2.3 Metoda ParseKeyPoint

- `KeyPoint ParseKeyPoint(string line)`

Jest to kolejna metoda odczytująca linię. Odczytuje ona linię zawierającą współrzędne punktów kluczowych i ich nazwy, po czym w przypadku powodzenia zwraca przygotowany obiekt klasy *KeyPoint*.

3.2.4 Metoda ParseCustomObjectType

- `CustomObjectType ParseCustomObjectType(string line)`

Ta metoda odpowiada za odczytanie deklaracji typu obiektu przygotowanej przez użytkownika. Do jej zadań należy sprawdzenie, czy użytkownik napisał dla każdego parametru nazwę i typ, oraz czy program potrafi rozpoznać ten typ.

3.2.5 Metoda ParseCustomObjectInstance

- `CustomObjectInstance ParseCustomObjectInstance(string line, List<CustomObjectType> customObjectTypes)`

Ta metoda odczytuje obiekty wypisane przez użytkownika, zgodnie z zadeklarowanymi przez niego typami na poprzednim etapie odczytu. Do jej zadań należy sprawdzenie, czy podane obiekty są zgodne z deklaracjami dokonanymi wcześniej. Tak jak każda z wymienionych powyżej metod poprawnie odczytany obiekt zwraca do metody głównej tej klasy – *ParseFile*.

3.3 Klasa AreaDivider

Ta klasa zawierać będzie logikę podziału mapy na obszary. W finalnej wersji programu będzie ona składać się z większej liczby metod, których ilość oraz treść zależeć będzie od wybranego algorytmu rozwiązującego problem podziału mapy.

3.3.1 Metoda DivideIntoAreas

- `void DivideIntoAreas (Map map)`

Ta metoda ma być kluczowym etapem dzielenia mapy na obszary. Dostaje ona obiekt typu *Map*, który zawiera wszystkie odczytane z pliku lub zmodyfikowane w trakcie trwania programu dane, po czym wykonuje przetworzenie linii zgodnie z wybranym algorytmem oraz zapisuje je w postaci listy *areaLines*, należącej do właściwości klasy *Map*.

3.4 Klasa Map

Klasa *Map* jest głównym kontenerem danych tego programu – wszystkie inne klasy albo znajdują się w tej klasie, albo w pewien sposób się do niej odwołują. Składa się ona z pięciu list przechowujących wszystkie najważniejsze elementy prezentowanej na ekranie mapy:

- punkty konturu,
- punkty kluczowe,
- linie tworzące obszary,
- typy obiektów deklarowane przez użytkownika,
- obiekty podane przez użytkownika.

Poza tymi listami klasa zawiera także metody odwołujące się bezpośrednio do matematycznej reprezentacji mapy. Poniżej zostały opisane dwie takie metody.

3.4.1 Metoda FindKeyPointOfArea

- `KeyPoint FindKeyPointOfArea (double X, double Y)`

Ta metoda ma na podstawie współrzędnych punktu określić, w jakim obszarze się on znajduje. Zwraca punkt kluczowy *KeyPoint* reprezentujący dany obszar lub *null* w przypadku, gdy punkt nie znajduje się w żadnym obszarze.

3.4.2 Metoda FindObjectsOfArea

- `List<CustomObjectInstance> ObjectsOfArea (KeyPoint keyPoint)`

Ta metoda ma działanie odwrotne do poprzedniej metody – przyjmuje punkt kluczowy jako argument, po czym zwraca listę wszystkich obiektów zawartych w wyznaczanym przez niego obszarze lub *null*, gdy punkt kluczowy nie zostanie rozpoznany.

3.5 Klasy Point, KeyPoint

Te dwie proste klasy reprezentują punkty na mapie. Klasa *Point* przechowuje jedynie współrzędne X i Y danego punktu, zaś dziedzicząca po niej klasa *KeyPoint* zyskuje dodatkowo nazwę danego punktu kluczowego. Nie posiadają one żadnych dodatkowych metod.

3.6 Klasa Line

Jest to klasa reprezentująca linię, wykorzystywana do podziału mapy na obszary. Składa się ona z dwóch punktów reprezentujących początek i koniec linii.

3.7 Klasa CustomObjectType

Klasa przechowuje dokładne dane o zadeklarowanych przez użytkownika parametrach obiektu. Oprócz nazwy będącej prostym napisem przechowuje ona słownik zmiennych – każdy parametr musi mieć swoją charakterystyczną nazwę oraz przypisany do niej typ. Klasa ta będzie stanowić bazę do tworzenia konkretnych obiektów, które reprezentuje następna klasa – *CustomObjectInstance*.

3.8 Klasa CustomObjectInstance

Ostatnia klasa tego programu ma prezentować konkretny obiekt na planszy, spełniający parametry zadeklarowanego typu. Dziedziczy ona po klasie *Point*, co gwarantuje, że każdy podany przez użytkownika obiekt ma sens na mapie i posiada współrzędne, na podstawie których można go ulokować w konkretnym obszarze.

Składa się on z dwóch istotnych właściwości – jedną z nich jest typ obiektu, przechowywany w postaci referencji na dany typ. Drugą jest tablica *objectProperties*, która ma przechowywać obiekty o typach zgodnych z zadeklarowanymi przez użytkownika w pliku.

3.8.1 Konstruktor klasy CustomObjectInstance

- `CustomObjectInstance (CustomObjectType customObjectType, string[] args)`

Warty odnotowania jest konstruktor tej klasy. Dostaje on listę argumentów w postaci napisów *string*, które następnie sam konwertuje na odpowiednie typy i zapisuje je do tablicy *objectProperties*. W przypadku niezgodności typów wyrzuca wyjątek, który może być potem obsłużony w klasie *Parser*, w metodzie *ParseCustomObjectInstance*.

4 Działanie programu i zastosowanie algorytmu

Działanie programu rozpoczyna się wraz z uruchomieniem funkcji *Main*. Platforma WPF od razu uruchamia główne okno interfejsu graficznego, choć początkowo część funkcji nie jest dostępna. Wygląd okna zdefiniowany jest w języku opisu interfejsu *XAML*, co jest typowe dla wykorzystanej technologii *WPF*. W tym momencie, w zależności od wybranej przez użytkownika opcji z pasku górnego program może:

- wczytać mapę na podstawie danych z pliku,
- wczytać obraz tła,
- wyświetlić pomoc.

W przypadku wybrania 2. lub 3. opcji program wykonuje pojedynczą czynność – wczytanie obrazu z pliku lub wyświetlenie tekstu na panelu bocznym, za co odpowiadają pojedyncze metody z klasy *MainWindow*. W przypadku wybrania wczytania obrazu sterowanie zostanie przekazane klasie *Parser*, która podejmie próbę przekazania danych do przygotowanych struktur w obiekcie klasy *Map*. W przypadku niepoprawnego pliku program wyświetli komunikat błędu na dolnym panelu tekstowym. Jeśli wczytanie odbędzie się poprawnie, *Parser* zwróci tak przygotowany obiekt do *MainWindow*, która z kolei prześle obiekt *Map* do klasy dzielącej na obszary – *AreaDivider*.

W tej klasie generowany będzie diagram Woronoja, za pomocą algorytmu Fortune'a. Złożoność czasowa tego algorytmu wynosi $O(n \log n)$, a pamięciowa $O(n)$. Metoda *DivideIntoAreas*, która będzie łączyła mechanizm podziału z zewnętrznym programem, w otrzymanym obiekcie klasy *Map* umieści listę linii dzielących mapę na obszary. Użytkownik w dowolnym momencie po wczytaniu mapy może dodawać i usuwać punkty kluczowe – w takim przypadku symulacja krawędzi zostaje powtórzona i obiekt *Map* ponownie otrzyma listę linii, już zaktualizowaną po zmianie. Metody obsługujące ruchy myszki w obrębie mapy znajdują się w klasie *MainWindow*.

Użytkownik może w dowolnym momencie przerwać działanie programu, zamykając okno przyciskiem X znajdującym się w prawym górnym rogu. Może też w dowolnym momencie wczytać kolejny plik wybierając tą samą opcję z górnego paska, co poprzednio.

5 Testy jednostkowe

Projekty testów zakładają użycie narzędzia *NUnit*. Testy zostaną wykonane według zasady AAA (Arrange Act Assert). Poniżej znajduje się lista testów planowanych dla kluczowych metod programu, dla każdej z nich przewidziane są różne przypadki otrzymanych danych.

5.1 Testy klas MainWindow

Klasa ta jest punktem startowym programu i sama wywołuje inne klasy. Posiada metody ściśle związane z platformą *WPF*, odpowiedzialne za interpretację klikania lewym lub prawym przyciskiem myszy bądź najeżdżania kursorem na konkretny element mapy, będą one jednak testowane manualnie. Należy zwrócić uwagę, czy przy dodawaniu punktu kluczowego lub punktu konturu, wstawiany element tworzony jest dokładnie w miejscu kliknięcia, a nie na przykład niżej od kursora o wartość wysokości górnego paska narzędzi.

5.2 Testy klasy Parser

5.2.1 Metoda ParseFile

Aby umożliwić przetestowanie tej metody, utworzone zostaną testowe pliki tekstowe. Należy jednak przetestować także zachowanie programu, gdy ścieżka do pliku wejściowego jest niepoprawna. Sprawdzone zostanie także zachowanie programu, gdy wczytane kontury terenu ułożone są w ten sposób, że linie się przecinają. Przetestowana zostanie także sytuacja, w której wśród definicji obiektów zabraknie obowiązkowych obiektów

5.2.2 Metody ParseContourPoint, ParseKeyPoint, ParseCustomObjectType

Wymienione metody jest najłatwiej przetestować, gdyż otrzymują one jako argument dokładnie jedną linijkę tekstu. Oczywiście dla każdej z nich należy spróbować podać w pierwszej kolejności poprawnie sformatowane dane, a następnie porównać oczekiwane wartości zwracane z faktycznie zwracanymi wartościami. W przypadku wczytywania punktu konturu, należy sprawdzić z ilu wyrazów składa się wczytywana linia. Spodziewamy się dokładnie trzech elementów – indeksu punktu konturu, oraz dwóch wartości liczbowych, odpowiadających jego współrzędnym. Przy wczytywaniu punktu kluczowego, nie możemy już sporządzać takich założeń, gdyż nazwa punktu może składać się z wielu wyrazów.

Należy także dla każdej z metod przygotować niepoprawnie sformatowane ciągi znaków. Przy pomocy narzędzia testowego sprawdzone zostanie, czy wystąpił błąd którego się spodziewaliśmy. Podane zostaną:

- linie niezawierające informacji o położeniu obiektu,
- linie zawierające napis w miejscu na współrzędne liczbowe określające położenie obiektu,
- linie zawierające niepotrzebne informacje, na przykład punkt konturu posiadający nazwę,
- linie zawierające informacje o punkcie kluczowym lub o definicji obiektu bez nazwy,
- linie przyporządkowujące obiektowi położenie poza granicami terenu, to jest o współrzędnych ujemnych bądź większych od 600.

5.3 Testy AreaDivider

Test metody *DivideIntoAreas* wymagał będzie utworzenia obiektu typu *Map*. Należy utworzyć stosunkowo prostą planszę, na której łatwo jest przewidzieć podział na obszary. Skoro obszary podzielone są liniami, możemy sprawdzić ich zgodność z naszymi przewidywaniami, przy pomocy punktów startowych i końcowych linii, na przykład sprawdzając czy *areaLines* zawierają linie o wskazanych wartościach.

5.4 Testy klasy Map

5.4.1 Metody FindKeyPointOfArea oraz FindObjectsOfArea

Obie metody są w gruncie rzeczy dość podobne w budowie. Podobnie będzie ze sporządzonymi dla nich testami jednostkowymi. Będą wymagały one utworzenia obiektu klasy macierzystej, który powinien być na tyle mało skomplikowany, aby dało się przewidzieć odpowiedź, którą powinna zwrócić jedna z funkcji szukających. Test powiedzie się, gdy zwracany poszukiwany punkt kluczowy lub lista obiektów będzie zgodna z przewidywaniami.

5.5 Testy klasy Point, Line oraz KeyPoint

Klasy te zostały skumulowane w jednej sekcji, gdyż wszystkie są nieskomplikowanej budowy i składają się z szeregu automatycznie zaimplementowanych właściwości, które pełnią rolę metod dostępowych. Możemy przetestować ich poprawność, po wcześniejszym utworzeniu obiektów danych klas.

5.6 Testy klasy CustomObjectType

Klasa *CustomObjectType* przechowuje nazwy i typy zmiennych, dlatego musi zostać przetestowana pod kątem poprawnego zapisu tych typów. Po wczytaniu danych zostanie sprawdzone poprawne działanie zapisanego słownika, przez próbę utworzenia danych na podstawie zapisanych wartości. Przeprowadzona zostanie też próba dodania wielu parametrów o tej samej nazwie.

5.7 Testy klasy CustomObjectInstance

Klasa ma reprezentować pojedynczy obiekt tworzony na podstawie wzorca omówionego w poprzednim teście. Jej testy ograniczą się do przygotowania poprawnego obiektu *CustomObjectType* oraz do uruchomienia konstruktora tej klasy, podając:

- dane poprawne,
- dane zawierające mniej danych niż we wzorcu,
- dane zawierające więcej danych niż we wzorcu,
- dane zawierające inne typy danych niż we wzorcu.