Specyfikacja Implementacyjna

Paweł Skiba, Jakub Świder

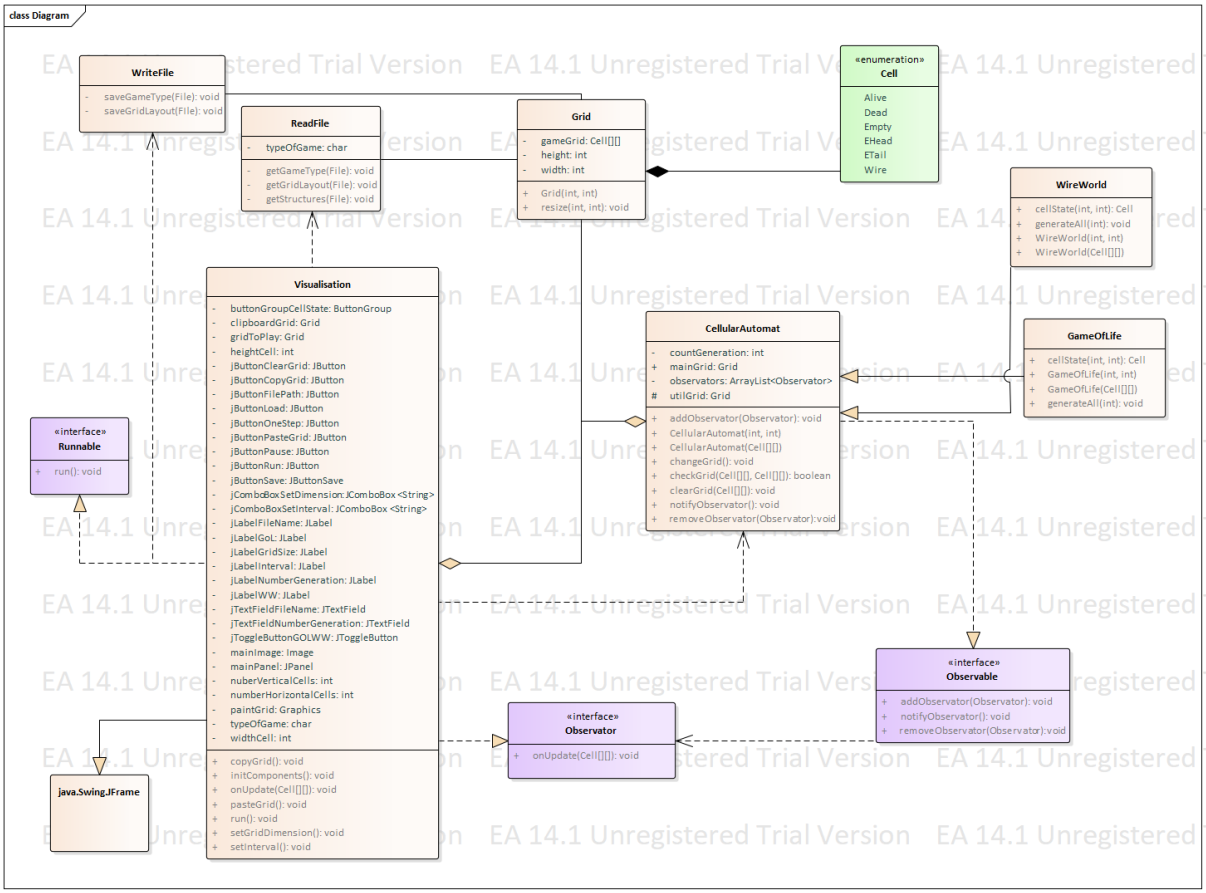
18 maja 2019

Spis treści

1 Wstęp

Projekt *CA Simulator* zostanie stworzony w języku *JAVA*. Program będzie w pełni kompatybilny z wersją *Java SE Development Kit 8*. Zostanie on wytworzony w zintegrowanym środowisku programistycznym *Apache NetBeans 11.0*. Podczas tworzenia programu będzie używane narzędzie dostarczone przez *Apache NetBeans IDE 11.0* do tworzenia okien aplikacji. Graficzny interfejs użytkownika będzie budowany w oparciu o bibliotekę graficzną *Swing*.

2 Diagram klas



Rysunek 1: Diagram klas

3 Pakiety

Program CA Simulator będzie podzielony na 3 pakiety, które zostaną opisane poniżej.

3.1 GUI

W skład pakietu *GUI* będą wchodziły następujące elementy:

- *Visualisation* - klasa;
- *ReadFile* - klasa;
- *WriteFile* - klasa;
- *Runnable* - interfejs;
- *Observator* - interfejs.

Pakiet *GUI* przede wszystkim będzie w pełni odpowiedzialna za stworzenie graficznego interfejsu użytkownika. Dodatkowo pakiet będzie odpowiedzialny za obsługę plików, taką jak wczytywanie i zapisywanie do pliku planszy. Program będzie uruchamiany z poziomu tego pakietu, a konkretniej z poziomu klasy *Visualisation*.

3.2 Grid

W skład pakietu *Grid* będą wchodziły następujące elementy:

- *Grid* - klasa;
- *Cell* - klasa wyliczeniowa.

Pakiet *Grid* będzie odpowiedzialny za stworzenie dwuwymiarowej siatki komórek, słuzącej do przeprowadzania symulacji automatów komórkowyh. Dodatkowa klasa wyliczeniowa *Cell* przechowuje wszystkie możliwe stany w grach *Wire World* i *Game of Life*.

3.3 Core

W skład pakietu *Core* będą wchodziły następujące elementy:

- *Celluar Automat* - klasa;
- *GameOfLife* - klasa;

- *WireWorld* - klasa;
- *Observable* - interfejs.

Pakiet *Core* będzie zawierał klasy, które będą odpowiadały za przeprowadzenie symulacji dla konkretnych automatów komórkowych. Główna klasa *Cellular Automat* może być rozszerzana przez dowolną ilość różnych automatów komórkowych. W naszym przypadku będzie to *Wire World* i *Game of Life*.

4 Klasy

4.1 GUI

1. **Visualisation**

   Klasa *Visyualsation* jest jedną z najważniejszych klas powyższego programu. Jej głównym zadaniem jest stworzenie graficznego interfejsu użytkownika oraz obsługa zdarzeń zadanych przez użytkownika. **Atrybuty:**

   (a) **-Grid clipboardGrid** - obiekt zawierający dwuwymiarową tablicę typu *Cell*, w której będą przechowywane stany komórek, służy do przechowywania planszy danego automatu komórkowego wybranego przez użytkownika;

   (b) **-Grid gridToPlay** - obiekt zawierający dwuwymiarową tablicę typu *Cell*, w której będą przechowywane stany komórek, służy do edytowania i przechowywania planszy wejściowej;

   (c) **-int heightCell** - wysokość komórki dla danego formatu panelu;

   (d) **-int widthCell** - szerokość komórki dla danego formatu panelu;

   (e) **-int numberVerticalCell** - ilość wierszy dla danego formatu panelu;

   (f) **-int numberHorizontalCell** - ilość kolumn dla danego formatu panelu;

   (g) **-Graphics paintGrid** - obiekt klasy *Graphics* służy do wygenerowania rysunku/szkicu na podstawie obiektu *gridToPlay* typu *Grid*;

   (h) **-Image mainImage** - obiekt klasy *Image* służy do wygenerowania obrazu na podstawie obiektu *paintGrid* typu *Graphics*;

(i) **-char typeOfGame** - zmienna przechowywująca znak odpowiadający używanemu automatowi komórkowemu, dla automatu komórkowego Wire World jest to znak 'W', naotmiast dla automatu komórkowego Game of Life 'G';

(j) **-JPanel mainPanel** - jest to główny komponent (o wymiarach 600 x 400 pikseli) powyższego projektu. Jest on odpowiedzialny za możliwość edycji wejściowej generacji oraz za wyświetlanie generowanych obrazów.

Metody:

(a) **+Visualisation()** - konstruktor, który będzie wywoływał funkcję *initComponents()*;

(b) **-initComponents(): void** - funkcja będzie generowana przez zintegrowane środowisko programistyczne *Appache NetBeans IDE 11.0*, zawierać będzie wszystkie inicjowane obiekty w *Design*;

(c) **+run(): void** - funkcja uruchamiająca wątek, wewnątrz niej będą przeprowadzane wszystkie symulacje automatów komórkowych;

(d) **-onUpdate(Cell[][] gridToDisplay): void** - funkcja będzie odpowiedzialna za deklarowanie wszystkich obiektów służących do tworzenia obrazu oraz za jego utworzenie i wyświetlenie;

(e) **-setGridDimension(): void** - funkcja będzie ustawiała wymiary siatki, przerysowywała obraz tworzony na obiekcie*mainPanel* oraz będzie tworzyć obiekt *gridToPlay* zainicjowany martwymi/pustymi komórkami;

(f) **-setInterval(): void** - funkcja będzie ustawiała interwał czasowy z jakim będą wyświetlane kolejne generacje automatów komórkowych;

(g) **-pasteGrid(): void** - funkcja będzie odpowiedzialna za wyświetlenie przechowywanej w schowku planszy oraz ustawienie komórek obiektu *gridToPlay* przechowywanymi stanami;

(h) **-copyGrid(): void** - funkcja będzie odpowiedzialna za skopiowanie dwuwymiarowej obecnie wyświetlonej tablicy do schowka (obiektu *clipboardGrid*);

(i) **clearPlayGrid(): void** - funkcja ma na celu ustawienie wszystkich komórek obiektu *gridToPlay* na stan martwy/pusty oraz przerysować planszę.

2. **ReadFile**

   Klasa ReadFile jest odpowiedzialna za odczytywanie plików zapisu planszy, tak aby mogły być one poprawnie wyświetlone w programie;

   Atrybuty:

   (a) **-char typeOfGame** - określa rodzaj automatu komórkowego. Jeśli równa się 'W' program odczyta plik jako plik wejściowy do WireWorld, a jeśli 'G' jako plik wejściowy do Game of Life.

   Metody:

   (a) **-getGameType(File saveFile): int** - metoda na podstawie pliku tekstowego określa, do którego automatu komórkowego odnosi się opis planszy w pliku;

   (b) **-getGridLayout(File saveFile): void** - metoda na podstawie pliku tekstowego zapełni planszę w programie odpowiednimi komórkami;

   (c) **-getStructures(File saveFile): void** - metoda określi czy użytkownik dodał w pliku współrzędne dodatkowych elementów (np. bramki logiczne, diody). Jeśli tak to umieści je na planszy.

3. **WriteFile**

   Klasa WriteFile jest odpowiedzialna za tworzenie plików zapisu stanu planszy z programu, tak aby mogły by one później odczytane przez program;

   Metody:

   (a) **-saveGameType(File newSaveFile): void** - metoda na podstawie obecnie używanego automatu komórkowego w programie zapisze odpowiednią informację do pliku zapisu planszy;

   (b) **-saveGridLayout(File newSaveFile): void** - metoda na podstawie obecnie widocznej planszy w programie zapisze jej stan do pliku zapisu.

4.2 Grid

1. **Grid**

   Klasa *Grid* jest klasą implementującą dwuwymiarową tablicę stanów komórek z klasy *Cell*.

Atrybuty:

(a) **-int height** - atrybut przechowujący liczbę wierszy macierzy reprezntującej planszę komórek automatu komórkowego;

(b) **-int width** - atrybut przechowujący liczbę kolumn macierzy reprezntującej planszę komórek automatu komórkowego;

(c) **-Cell[][] gameGrid** - dwuwymiarowa macierz reprezentująca planszę służącą do gry, przechowuje stany poszczególnych komórek.

Metody:

(a) **+Grid(int hei, int wi)** - konstruktor tworzący macierz stanów komórek z ustawionym wartościami na Empty/Dead o podanych parametrach wymiarów;

(b) **+resize(int hei, int wi): void** - funkcja, która ma za zadanie przewymiarować istniejącą planszę na podane przez użytkowika wymiary, za pomocą zgubienia referencji i stworzenia nowej istancji tablicy.

2. **Cell**

Klasa *Cell* jest klasą wyliczeniową. Implementuje ona wszytskie możliwe stany wszystkich automatów komórkowych.

Enum:

(a) **Alive** - komórka żywa; automat komórkowy GameOfLife;

(b) **Dead** - komórka martwa; automat komórkowy GameOfLife;

(c) **Empty** - komórka pusta; automat komórkowy WireWorld;

(d) **EHead** - komórka reprezentująca głowę elektronu; automat komórkowy WireWorld;

(e) **ETail** - komórka reprezentująca ogon elektronu; automat komórkowy WireWorld;

(f) **Wire** - komórka reprezentująca przewodnik; automat komórkowy WireWorld.

4.3 Core

1. **CellularAutomat**

   Klasa ta zawiera atrybuty i metody wykorzystywane w symulacji obu automatów komórkowych, bez względu na ich rodzaj.

   Atrybuty:

   (a) **#Grid mainGrid** - dwuwymiarowa macierz reprezentująca planszę dla komórek automatu komórkowego. Na podstawie planszy mainGrid tworzone są nowe generacje i wyświetlany obraz;

   (b) **#Grid utilGrid** - dwuwymiarowa macierz reprezentująca planszę dla komórek automatu komórkowego. Do utilGrid zapisywana jest obecnie generowana plansza;

   (c) **-int countGeneration** - oznacza liczbę generacji, którą program ma wykonać;

   (d) **-observators ArrayList<Observator>** - jest to obiekt klasy ArrayList przechowujący obserwatorów dla danego automatu komórkowego.

   Metody:

   (a) **+changeGrid(): void** - nadpisuje do głównej macierzy (tej na podstawie, której wytwarzana jest nowa generacja) stan z macierzy pomocniczej (tej, do której zapisywana jest początkowo nowa generacja);

   (b) **+clearGrid(Cell[][] gridToClear): void** - wypełnia macierz symbolami reprezentującymi komórki puste/martwe, w zależności od aktywnego automatu;

   (c) **+checkGrid(Cell[][] firstGrid, Cell[][] secondGrid): boolean** - porównuje dwie dwuwymiarowe tablice, jeżeli będą takie same zwróci on wartość *true*, jeżeli nie *false*;

   (d) **+CellularAutomat(int hei, int wi)** - uruchamia konstruktor tworzący obie plansze potrzebne w implementacji automatu komórkowego (*mainGrid* i *utilGrid*) nadając im wymiary równe argumentom wywołania (*hei* oznacza ilość wierszy, a *wi* oznacza ilość kolumn planszy). Następnie wywołuje dla obu plnasz metodę *clearGrid()*;

   (e) **+addObservator(Observator o): void** - funkcja dodaje do *ArrayList<Observator>* obiekt, który będzie obserwował dany automat komórkowy;

(f) **+removeObservator(Observator o): void** - funkcja usuwa z *ArrayList<Observator>* obiekt, który obserwował dany automat komórkowy;

(g) **+notifyObservator(): void** - funkcja powiadamia obserwujące obiekty o zmianie stanu obserwowanego obiektu, wywołuje ona funkcję *onUpdate(Cell[][])*.

2. **WireWorld**

WireWorld jest klasą dziedziczącą po klasie CellularAutomat. Jej metody pozawalają na impplementację zasad działania automatu komórkowego WireWorld w programie.

Metody:

(a) **+WireWorld(int hei, int wi)** - konstruktor tworzy nowy obiekt *WireWorld* przypuisując atrybutowi *mainGrid* dwuwymiarową tablicę wypełnioną pustymi komórkami o wymiarach hei x wi;

(b) **+WireWorld(Cell[][] grid)** - konstruktor tworzy nowy obiekt *WireWorld* przypisując atrybutowi *mainGrid* dwuwymiarową tablicę przyjmowaną jako argument;

(c) **+cellState(int i,int j): Cell** - metoda jako argumenty przyjmuje liczby będące indeksami komórki i na podstawie obecnego stanu jej i jej ośmiu sąsiadów ustala stan tej komórki w następnej generacji i zapisuje go do planszy pomocniczej;

(d) **+generateAll(int generationCount): void** - metoda przyjmuje jako argument liczbę generacji jaką ma wykonać program. Następnie wywołuje dla komórek głównej planszy metodę cellState() zapełniając w ten sposób planszę pomocniczą, wywołuje metodę notifyObservator(), wywołuje metodę clearGrid() dla planszy głównej, wywołuje metodę change() i clearGrid() dla planszy pomocniczej. Metoda generateAll() wykonuje te operacje tyle razy ile wynosi generationCount.

3. **GameOfLife**

   GameOfLife jest klasą dziedziczącą po klasie CellularAutomat. Jej metody pozawalają na implementację zasad działania automatu komórkowego GameOfLife w programie.

   Metody:

   (a) **+GameofLife(int hei, int wi)** - konstruktor tworzy nowy obiekt *GameOfLife* przypuisując atrybutowi *mainGrid* dwuwymiarową tablicę wypełnioną pustymi komórkami o wymiarach hei x wi;

   (b) **+GameofLife(Cell[][] grid)** - konstruktor tworzy nowy obiekt *GameOfLife* przypisując atrybutowi *mainGrid* dwuwymiarową tablicę przyjmowaną jako argument;

   (c) **+cellState(int i, int j): Cell** - metoda jako argumenty przyjmuje liczby będące indeksami komórki i na podstawie obecnego stanu jej i jej ośmiu sąsiadów ustala stan tej komórki w następnej generacji i zapisuje go do planszy pomocniczej;

   (d) **+generateAll(int generationCount): void** - metoda przyjmuje jako argument liczbę generacji jaką ma wykonać program. Następnie wywołuje dla komórek głównej planszy metodę cellState() zapełniając w ten sposób planszę pomocniczą, wywołuje metodę notifyObservator(), wywołuje metodę clearGrid() dla planszy głównej, wywołuje metodę change() i clearGrid() dla planszy pomocniczej. Metoda generateAll() wykonuje te operacje tyle razy ile wynosi generationCount.

5 Interfejsy

5.1 GUI

1. **Runnable**

   Interfejs *Runnable* jest implementowany przez klasę *Visualisation*. Dzięki niemu możliwe jest uruchomienie wątku, w którym przeprowadzana będzie symulacja wybranego automatu komórkowego.

   Implementowane metody:

   (a) *+run(): void*;

2. **Observator**

   Interfejs *Observator* jest implementowany przez klasę *Visualisation*. Wykorzystywany jest do implementacji wzorca projektowego *Observator*.

   Implementowane metody:

   (a) *onUpdate(Cell[][]): void*;

5.2 Core

1. **Observable**

   Interfejs *Observable* jest implementowany przez klasę *CellularAutomat*. Wykorzystywany jest do implementacji wzorca projektowego *Observator*.

   (a) *addObservator(Observator o)*;

   (b) *removeObservator(Observator o)*;

   (c) *notifyObservator()*;

6 Wzorzec projektowy

6.1 Definicja

W programie zostanie zastosowany wzorzec projektowy *Obserwator*. Wzorzec projektowy *Obserwator* jest wzorcem czynnościowym, który jest używany do powiadamiania subskrybujących obiektów o zmianie stanu subskrybowanego obiektu.

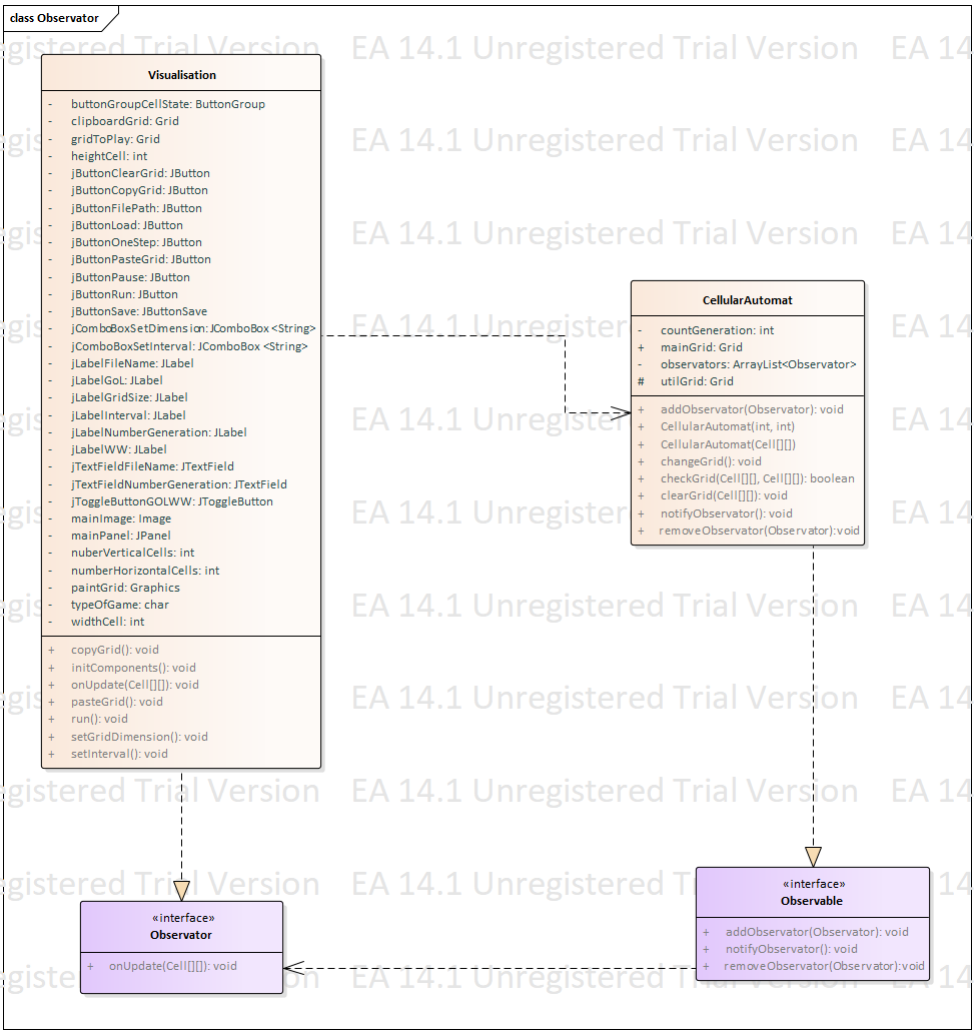
6.2 Cel

Użyty w programie wzorzec projektowy *Obserwator* służy do aktualizowania panelu przedstawiającego przebieg gry.

6.3 Sposób zastosowania

Klasa *Visualisation* oraz klasa *CellularAutomat* implementują interfejsy. Pierwsza z nich implementuje interfejs *Observator*, natomiast druga implementuje interfejs *Observable*. Z poziomu klasy *Visualisation* tworzony jest obiekt dowolnego automatu komórkowego, który następnie dodaje do swojej listy

obserwatorów obiekt klasy *Visualisation*. Następnie wywoływana jest funkcja *generateAll(int)*, która jest odpowiedzialna za przeprowadzenie danej ilości generacji dla danego automatu komórkowego. Po każdej iteracji funkcja *generateAll(int)* wywołuje funkcję *notifyObservator()*, natomiast ta funkcja wywołuje dla każdego obserwującego obiektu funkcję *onUpdate(Cell[][])*, po której zostaje zaktualizowana plansza. Następnie zostaje wykonana następna generacja. Poniżej przedstawiony jest diagram pokazujący powiązania między interfejsami i klasami, które wspólnie wykorzystują wzorzec projektowy.



Rysunek 2: Diagram przedstawiający zastosowanie wzorca projektowego *Obserwator*

7 Testy

Testy jednostkowe będą wykonywane z użyciem biblioteki JUnit. Testowany będzie przede wszystkim pakiet Core. Do testów przeznaczona będzie oddzielna klasa, w której napisane zostaną metody testujące pojedyncze metody z klas w tym pakiecie.

1. **+testChangeGrid()** - test metody **change()** z klasy *CellularAutomat*. Test będzie polegał na stworzeniu dwóch macierzy dwuwymiarowych, o tych samych wymiarach zapełnionych różnymi danymi. Przy użyciu metody **change()** do jednej z macierzy przepisana zostanie zawartość drugiej macierzy. Następnie obie macierze zostaną porównane metodą **deepEquals()**, która powinna zwrócić *True*. Do testu zostanie wykorzystana metoda **Assert.assertTrue()**.

2. **+testClearGrid()** - test metody **+clearGrid()** z klasy *CellularAutomat*. Test będzie polegał na stworzeniu macierzy dwuwymiarowej i zapełnieniu jej wartościami nieoznaczającymi w programie komórki pustej, stworzeniu drugiej macierzy o tych samych wymiarach zapełnionej wartościami oznaczającymi komórki puste. Następnie dla macierzy pierwszej wywołana będzie metoda **clearGrid()**. Obie macierze zostaną porównane metodą **deepEquals()**, która powinna zwrócić *True*. Do testu zostanie wykorzystana metoda **Assert.assertTrue()**.

3. **+testCheckGridTrue()** - test metody **checkGrid()** z klasy *CellularAutomat*. Test będzie polegał na stworzeniu dwóch macierzy dwuwymiarowych o tych samych rozmiarach i zapełnieniu ich tymi samymi danymi w ten sam sposób. Następnie zostaną one porównane metodą **deepEquals()**, która powinna zwrócić wartość *True*. Do testu zostanie wykorzystana metoda **Assert.assertTrue()**.

4. **+testCheckGridFalse()** - test metody **checkGrid()** z klasy *CellularAutomat*. Test będzie polegał na stworzeniu dwóch macierzy dwuwymiarowych o tych samych rozmiarach i zapełnieniu ich różniącymi się od siebie danymi, tak aby macierze nie były identyczne. Następnie zostaną one porównane metodą **deepEquals()**, która powinna zwrócić wartość *False*. Do testu zostanie wykorzystana metoda **Assert.assertFalse()**.

5. **+testGameOfLifeCellState()** - test metody **+cellState()** z klasy *GameOfLife*. Test będzie polegał na stworzeniu macierzy dwuwymiarowej mającej reprezentować planszę i zapełnieniu jej wartościami oznaczającymi komórki w taki sposób, aby otrzymać planszę, dla której

będziemy znać stan każdej z komórek w kolejnej generacji. Plansza będzie zawierała takie układy komórek, które pozwolą na przetestowanie warunków granicznych zasad danego automatu komórkowego. Dla każdej komórki znajdującej się w takim układzie zostanie wywołana metoda **cellState()**, a wynik będzie porównany z oczekiwaną wartością. Wszystkie porównania powinny zwrócić wartość *True*. Do testu zostanie wykorzystana metoda **Assert.assertTrue()**.

6. **+testWireWorldCellState()** - test metody **+cellState()** z klasy *WireWorld*. Test będzie polegał na stworzeniu macierzy dwuwymiarowej mającej reprezentować planszę i zapełnieniu jej wartościami oznaczającymi komórki w taki sposób, aby otrzymać planszę, dla której będziemy znać stan każdej z komórek w kolejnej generacji. Plansza będzie zawierała takie układy komórek, które pozwolą na przetestowanie warunków granicznych zasad danego automatu komórkowego. Dla każdej komórki znajdującej się w takim układzie zostanie wywołana metoda **cellState()**, a wynik będzie porównany z oczekiwaną wartością. Wszystkie porównania powinny zwrócić wartość *True*. Do testu zostanie wykorzystana metoda **Assert.assertTrue()**.