RAPORT

Paweł Skiba, Jakub Świder

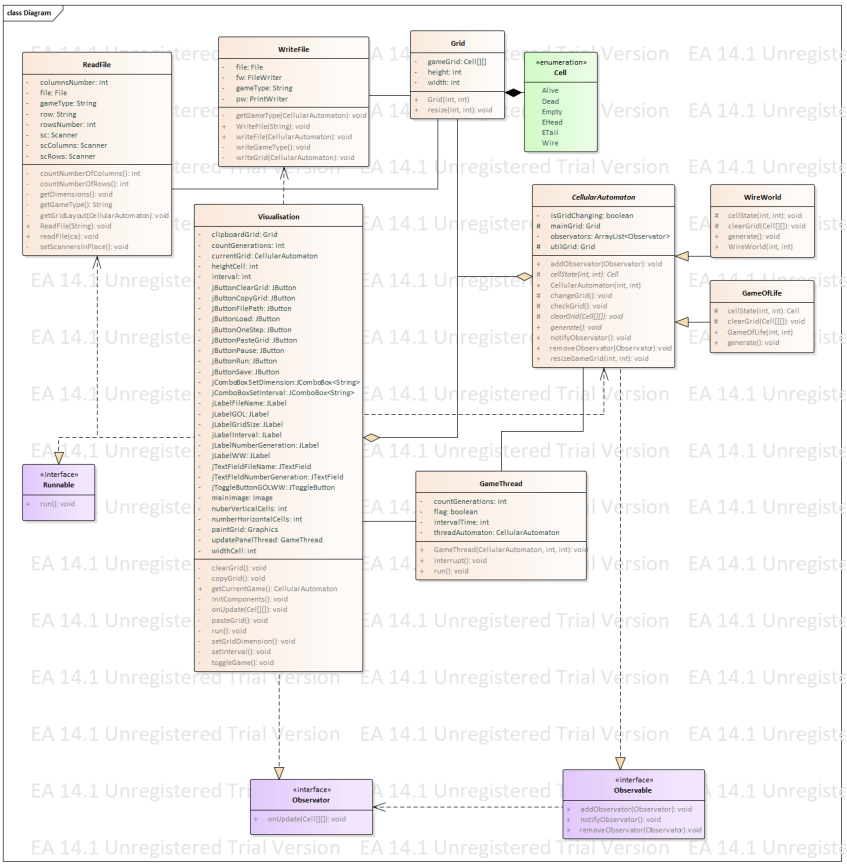
6 czerwca 2019

Spis treści

1 Wstęp

W niniejszym raporcie zostaną przedstawione ostateczne wyniki pracy nad projektem *CA Simulator*, w którego skład wchodzą dwa automaty komórkowe, główny - *WireWorld* oraz opcjonalny - *Game of Life*. Dodatkowo w raporcie zostaną umieszczone wszystkie zmiany, które zostały wprowadzone podczas budowy programu. Przedstawione zostaną również ostateczne założenia projektowe oraz ostateczny diagram modułów. Kolejnym aspektem będzie przedstawienie prezentacji działania oraz przedstawienie i omówienie testów. W projekcie zostało dokonanych wiele modyfikacji, jednak największą z nich jest rozbudowanie pakietu *GUI* poprzez zbudowanie nowej klasy *GameThread* na rzecz implementacji interfejsu *Runnable*.

2 Ostateczny diagram klas



Rysunek 1: Diagram klas

3 Opis modyfikacji klas

W niniejszym rozdziale zostaną opisane wszelkie modyfikacje, które zostały wykonane podczas budowy aplikacji. Raport szczególnie w tym punkcie ma bardzo duże powiązanie ze specyfikacją implementacyjną, ponieważ zostaną przedstawione tylko zmiany jakie istnieją między pierwszą wersją projektową, a finalnym projektem. Ze względu na niezmienny charakter wzorca projektowego *Obserwator* oraz jego implementacji zostaną pominięte dwa interfejsy:

- *Observator* - w klasie *GUI*, który jest odpowiedzialny za implementację funkcji, która aktualizuje obserwujący obiekt;
- *Observable* - w klasie *Core*, który jest odpowiedzialny za implementację funkcji dodającej, usuwającej oraz powiadamiającej obiekty obserwujące dany obiekt oraz pole klasy ArrayList<Observator>, która je przetrzymuje.

3.1 Pakiet GUI

W pakiecie *GUI* można zauważyć najwięcej zmian architektonicznych. Największą z nich jest całkowita zmiana podejścia do obsługi wątków. W trakcie tworzenia architektury projektu została założona koncepcja, że klasa *Visualisation* będzie implementowała interfejs *Runnable*. Ostatecznie zdecydowaliśmy się na nową klasę *GameThread*, która zastąpiła wymieniony wyżej interfejs. Szczegółowe zmiany każdej klasy pakietu *GUI* zostaną przedstawione poniżej.

3.1.1 Visualisation

- **Pola**

  1. **currentGrid**

     Aktualnie:
     -currentGrid: CellularAutomaton

     Poprzednio:
     -gridToPlay: Grid

     Opis:
     Najistotniejszą zmianą jest zmiana klasy pola. Koncepcja została zmieniona z powodu dużo szerszego zastosowania tego pola, niż w poprzednim przypadku. Poprzednio był to obiekt zawierający

dwuwymiarową tablicę typu *Cell*, w której miały być przechowywane stany komórek oraz miał służyć do edytowania i przechowywania planszy wejściowej. Aktualnie poprzednia koncepcja cały czas jest wykorzystywana, natomiast edycji ulega pole klasy abstrakcyjnej*AutomatonCellular* o nazwie -*Grid mainGrid*. Dodatkowo możemy tworzyć nową instancję klasy *GameOfLife*, bądź *WireWorld* przy przełączeniu oraz korzystać z wzorca projektowego, który został zaimplementowany w powyższej wymienionej klasie;

2. **interval**
   Aktualnie:
   -interval: int
   Poprzednio:
   Brak
   Opis:
   Pole, któremu przy deklaracji zostaje przypisana wartość równa 1000 ms. Przechowuje ono czas, na który jest wstrzymywany wątek pomiędzy kolejnymi iteracjami;

3. **countGenerations**
   Aktualnie:
   -countGenerations: int
   Poprzednio:
   Brak
   Opis:
   Pole, które zawiera liczbę generacji zadaną przez użytkownika do przeprowadzenia, istnienie tego pola jest konieczne. ponieważ jego wartość jest przekazywana jako argument przy tworzeniu nowego wątku;

4. **updatePanelThread**
   Aktualnie:
   -updatePanelThread: GameThread
   Poprzednio:
   Brak
   Opis:
   Jest to pole, będące obiektem klasy *GameThread*, które jest tworzony w momencie interackji użytkownia z programem poprzez

użycie przycisku *Play*. Pole to jest odpowiedzialne za uruchomienie nowego wątku, aktualizującego pole -*JPanel mainPanel* oraz za jego przerywanie;

5. **-**
   Aktualnie:
   Brak
   Poprzednio:
   -typeOfGame: char
   Opis:
   Pole *typeOfGame* okazało się być zbędnym. Jego rolę pełnią metody *getClass()* oraz *getName()* klasy bazowej *Class*. Dzięki temu ograniczyliśmy ilość pól oraz program można uznać za mniej ograniczony;

- **Metody**

1. **onUpdate()**
   Aktualnie:
   +onUpdate(): void
   Poprzednio:
   +onUpdate(Cell[][]): void
   Opis:
   W powyższej metodzie została zmieniona sygnatura. W poprzedniej wersji metoda przyjmowała jako argument dwuwymiarową tablicę *Cell[][]* i na jej podstawie aktualizowany był panel *mainPanel*. Obecnie w momencie wykonania generacji zostaje również zaktualizowany panel *mainPanel*, natomiast odbywa się to na podstawie pola *currentGame*, dla którego wykonywane są generacje;

2. **isGridChanged()**
   Aktualnie:
   -isGridChanged(): boolean
   Poprzednio:
   Brak
   Opis:
   Jest to metoda wywoływana w trakcie zmiany rozmiaru planszy lub zmiany typu automatu komórkowego. W momencie, gdy obecnie wyświetlona plansza zawiera jakiekolwiek komórki. które nie

są komórkami pustymi, bądź martwymi, powyższa funkcja zwróci wartość *true*, co spowoduje wyświetlenie komunikatu z informacją o utracie obecnego stanu planszy oraz zapytaniem czy użtkownik pomimo tego chce zmienić obecne ustawienia;

3. **toggleGame()**
   Aktualnie:
   -toggleGame(): void
   Poprzednio:
   Brak
   Opis:
   Metoda jest uruchamiana w momencie interakcji użytkownia z systemem poprzez użycie przełącznika. Metoda jest odpowiedzialna za stworzenie nowej instancji konretnego automatu komórkowego przypisywanej do pola *currentGame* oraz wyświetlanie, bądź ukrywanie grupy przycisków zależnie od wybranego automatu;

4. **-**
   Aktualnie:
   Brak
   Poprzednio:
   +run(): void
   Opis:
   Metoda miała być implementowana w klasie *Visualisation*, która miała implemetować interfejs *Runnable*. Nie została ona zaimplementowana na rzecz stworzenia nowej klasy *GameThread* oraz umieszczonej w tej klasie metody *run()*;

3.1.2 GameThread

- **Pola**

1. **threadAutomaton**
   Aktualnie:
   -threadAutomaton: CellularAutomaton
   Poprzednio:
   Brak
   Opis:
   Jest to pole, któremu podczas tworzenia obiektu klasy *GameThread* w konstruktorze jest przypisywana referencja do obiektu

przechowywanego w polu *currentGrid* w klasie *Visualisation*;

2. **flag**
   Aktualnie:
   -flag: boolean
   Poprzednio:
   Brak
   Opis:
   Jest to pole przechowywujące wartość wartość logiczną *true*, podczas gdy użytkownik uruchomi wizualizację automatu komórkowego poprzez przycisk *Play*;

3. **countGenerations**
   Aktualnie:
   -countGenerations: int
   Poprzednio:
   Brak
   Opis:
   Jest to pole przechowywujące liczbę, która jest inicjowana w trakcie tworzenia nowego obiektu klasy *ThreadGame*. Odpowiada ono za ilość wykonanych generacji;

4. **intervalTime**
   Aktualnie:
   -intervalTime: int
   Poprzednio:
   Brak
   Opis:
   Jest to pole przechowywujące liczbę odpowiadającą długości trwania czasu, na który jest wstrzymywany wątek pomiędzy kolejnymi generacjami. Jest ono inicjowane w trakcie tworzenia nowego obiektu klasy *ThreadGame*;

- **Metody**

1. **run()**
   Aktualnie:
   +run(): void

Poprzednio:
Brak

Opis:
Jest to metoda przesłaniająca metodę *run()* z klasy *Thread*. Jest ona wywoływana w momencie interakcji użytkownika z programem poprzez przycisk *Play*. Metoda *run()* jest przede wszystkim odpowiedzialna za wywoływanie metody *generate()* z klasy *CellularAutomaton* oraz usypianie wątku na czas, który przechowuje pole *intervalTime*. Metoda ta może działać na dwa sposoby. Pierwszy z nich, gdy pole *countGenerations* jest większe od 0, wątek kończy swoje działanie w momencie, gdy wykona liczbę generacji zawartą w polu *countGenerations* lub w momencie, gdy zostanie wywołana metoda *interrupt()*. Drugi sposób, gdy pole *countGenerations* jest równe 0, polega na działaniu wątku do momentu, gdy kolejna generacja jest taka sama jak poprzednia, co sprawdza metoda *isIsGridChanging()* lub w momencie, gdy zostanie wywołana metoda *interrupt()*.

2. **interrupt()**

   Aktualnie:
   +interrupt(): void

   Poprzednio:
   Brak

   Opis:
   Jest to metoda przesłaniająca metodę *run()* z klasy *Thread*. Jej działanie polega na przypisaniu polu *flag* wartości logicznej *false*. W momencie takiego przypisania, metoda *run()* zakończy swoje działanie;

3.1.3 ReadFile

- **Pola**

1. **file**

   Aktualnie:
   -file: File

   Poprzednio:
   Brak

   Opis:
   Klasa *File* służy do przeprowadzania podstawowych operacji na

pliku wejściowym. Ponadto instancja tej klasy posłuży jako parametr do stworzenia instancji klasy *Scanner*, dzięki którym możliwe będzie skanowanie pliku linijka po linijce;

2. **sc**

   Aktualnie:
   -sc: Scanner

   Poprzednio:
   Brak

   Opis:
   Ten obiekt klasy *Scanner* służy do czytania pliku wejściowego linijka po linijce i ustalenia typu symulatora oraz później układu siatki;

3. **scRows**

   Aktualnie:
   -scRows: Scanner

   Poprzednio:
   Brak

   Opis:
   Ten obiekt klasy *Scanner* służy do ustalenia liczby wierszy zapisanej w pliku siatki;

4. **scColumns**

   Aktualnie:
   -scColumns: Scanner

   Poprzednio:
   Brak

   Opis:
   Ten obiekt klasy *Scanner* służy do ustalenia liczby kolumn zapisanej w pliku siatki;

5. **rowsNumber**

   Aktualnie:
   -rowsNumber: int

   Poprzednio:
   Brak

Opis:
Jest to pole przechowywujące liczbę wierszy zapisanej w pliku planszy, odczytaną przez obiekt *scRows*;

6. **columnsNumber**
   Aktualnie:
   -columnsNumber: int
   Poprzednio:
   Brak
   Opis:
   Jest to pole przechowywujące liczbę kolumn zapisanej w pliku planszy, odczytaną przez obiekt *scColumns*;

7. **row**
   Aktualnie:
   -row: String
   Poprzednio:
   Brak
   Opis:
   Jest to pole umożliwiające obiektowi *scColumns* policzenie liczby kolumn, poprzez zapisanie wiersza z pliku wejściowego, usunięcie spacji poiędzy cyframi w wierszu i policzenie liczby znaków w wierszu;

8. **gameType**
   Aktualnie:
   -gameType: String
   Poprzednio:
   -typeOfGame: char
   Opis:
   Jest to pole przechowywujące nazwę klasy implementującej odpowiedni rodzaj automatu komórkowego. We wcześniejszej wersji do pola zapisywany był tylko jeden znak, który byłby interpretowany przez program gdzie indziej podczas daleszego działania;

- **Metody**

1. **getGameType()**
   Aktualnie:
   -getGameType(): String
   Poprzednio:
   -getGameType(File): int
   Opis:
   Jest to metoda ustalająca nazwę symulatora zapisaną w pliku i zapisująca ją do pola *gameType* jako nazwę klasy, którą program może zinterppretować. Wcześniejsza wersja metody przyjmowała jako parametr obiekt klasy *File* i zwracała rodzaj automatu komórkowego w inny sposób;

2. **setScannersInPlace()**
   Aktualnie:
   -setScannersInPlace(): void
   Poprzednio:
   Brak
   Opis:
   Jest to metoda ustawiająca obiekty *scColumns* i *scRows* na takich liniach w pliku wejściowym, aby mogły one wyznaczyć rozmiar wpisanej planszy;

3. **countNumberOfColumns()**
   Aktualnie:
   -countNumberOfColumns(): int
   Poprzednio:
   Brak
   Opis:
   Jest to metoda, która wykorzystując obiekt *scColumns* liczy liczbę kolumn w planszy zapisanej w pliku wejściowym;

4. **countNumberOfRows()**
   Aktualnie:
   -countNumberOfRows(): int
   Poprzednio:
   Brak

Opis:
Jest to metoda, która wykorzystując obiekt *scRows* liczy liczbę wierszy w planszy zapisanej w pliku wejściowym;

5. **getDimensions()**
   Aktualnie:
   -getDimensions(): void
   Poprzednio:
   Brak
   Opis:
   Jest to metoda wywołująca metody *countNumberOfColumns()* i *countNumberOfRows()* i zapisująca ich wyniki do odpowiednich pól;

6. **getGridLayout()**
   Aktualnie:
   -getGridLayout(CellularAutomaton): void
   Poprzednio:
   -getGridLayout(File): void
   Opis:
   Ta metoda przechodzi przez wszystkie znaki opisujące zapisaną planszę i w zależności od rodzaju gry zdefiniowanego wcześniej nadpisuje komórki planszy obiektu klasy *CellularAutomaton* (przyjętego jako parametr), która jest obecnie wyświetlana. We wcześniejszej wersji metoda przyjmowała jako parametr obiekt klasy *File* jednka nie jest to konieczne ze względu na pracę w instancji jednej klasy;

7. **readFile()**
   Aktualnie:
   +readFile(CellularAutomaton): void
   Poprzednio:
   Brak
   Opis:
   Jest to metoda wywołująca po kolei inne metody z tej klasy w celu poprawnego przeczytania pliku wejściowego. Jako parametr musi przyjąć ona obiekt klasy *CellularAutomaton*, żeby przekazać go dalej do metody *getGridLayout()*;

3.1.4 WriteFile

- **Pola**

1. **gameType**
   Aktualnie:
   -gameType: String
   Poprzednio:
   Brak
   Opis:
   Jest to pole przechowujące nazwę klasy implemenującej automat komórkowy, którego planszę chcemy zapisać;

2. **file**
   Aktualnie:
   -file: File
   Poprzednio:
   Brak
   Opis:
   Dzięki temu obiektowi klasy *File* możliwe będzie stworzenie pliku zapisu i operacje na nim;

3. **fw**
   Aktualnie:
   -fw: FileWriter
   Poprzednio:
   Brak
   Opis:
   Ten obiekt klasy *FileWriter* do stworzenia przyjmuje jako parametr obiekt file i sam posłuży jako parametr do utworzeniu obiektu pw klasy *PrintWriter*;

4. **pw**
   Aktualnie:
   -pw: PrintWriter
   Poprzednio:
   Brak

Opis:
Ten obiekt jest rzeczywiście wykorzystywany do zapisywania danytch w pliku zapisu;

- **Metody**

1. **getGameType()**
   Aktualnie:
   -getGameType(CellularAutomaton): void
   Poprzednio:
   Brak
   Opis:
   Metoda ta zapisuje do pola *gameType* nazwę klasy implemenującej automat komórkowy, którego planszę chcemy zapisać;

2. **writeGameType()**
   Aktualnie:
   -writeGameType(): void
   Poprzednio:
   -saveGameType(File): void
   Opis:
   Metoda zapisuje nazwę automatu do pliku w taki sposób, który umożliwi ponowną interpretację przez program podczas wczytywania tego pliku. Metoda nie potrzebuje obiektu klasy *File* jako parametru do poprawnego działania;

3. **writeGrid()**
   Aktualnie:
   -writeGrid(CellularAutomaton): void
   Poprzednio:
   -saveGridLayout(File): void
   Opis:
   Metoda na podstawie obecnie widocznej na ekranie planszy zapisze jej układ do pliku. Metoda nie potrzebuje obiektu klasy *File* jako parametru do poprawnego działania. Potrzebuje natomiast obiektu klasy *CellularAutomaton*, żeby móc zebrać informacje o układzie planszy;

4. **writeFile()**

   Aktualnie:
   +writeFile(CellularAutomaton): void

   Poprzednio:
   Brak

   Opis:
   Ta metoda wywołuje po kolei poprzednie metody z tej klasy celem utorzenia pliku zapisu i zapisania do niego wszystkich informacji. Metoda musi przyjąć obiekt klasy *CellularAutomaton*, żeby przekazać go dalej do metody *writeGrid()*;

3.2 Pakiet Core

W pakiecie *Core* podobnie jak w pakiecie *GUI* zostały wprowadzone znaczące poprawki architektoniczne. Przede wszystkim dotyczą one klasy *CellularAutomaton*. Klasa została zastąpiona klasą abstrakcyjną zachowując poprzednio zdefiniowane właściowości.

3.2.1 CellularAutomaton

- **Pola**

1. **isGridChanging**

   Aktualnie:
   -isGridChanging: boolean

   Poprzednio:
   Brak

   Opis:
   Jest to pole przyjmujące wartość logiczną *true* w przypadku, gdy tablice dwuwymiarowe *mainGrid* i *utilGrid* są rózne oraz wartość logiczną *false*, gdy są takie same;

2. **-**

   Aktualnie:
   Brak

   Poprzednio:
   -countGenerations: int

Opis:
Pole zostało usunięte z powodu modyfikacji metody *generateAll()* na metodę *generate()*;

- **Metody**

1. **clearGrid()**
   Aktualnie:
   # abstract clearGrid(Cell[][] gridToClear): void
   Poprzednio:
   +clearGrid(Cell[][] gridToClear): void
   Opis:
   Metoda została zmieniona na abstrakcyjną, ponieważ jest ona uzależniona od typu automatu komórkowego;

2. **checkGrid()**
   Aktualnie
   #checkGrid(): boolean
   Poprzednio:
   +checkGrid(Cell[][] firstGrid, Cell[][] secondGrid): boolean
   Opis:
   W powyższej metodzie została zmodyfikowana sygnatura. Nie potrzebuje ona wymienionych wyżej argumentów, ponieważ są one częścią tej klasy, przez co ma do nich bezpośredni dostęp;

3. **resizeGameGrid()**
   Aktualnie:
   +resizeGameGrid(int hei, int wi): void
   Poprzednio:
   Brak
   Opis:
   Metoda została zaimplementowana w celu zmieniania rozmiaru dwuwymiarowych tablic *mainGrid* i *utilGrid* na tablice o wymiarach hei x wi. Metoda ta wywołuje metodę *resize()* z klasy *Grid* dla obu wymienionych planszy;

4. **cellState()**
   Aktualnie:
   # abstract Cell cellState(int i, int j): Cell

Poprzednio:
Brak

Opis:
Metoda została dodana jak metoda abstrakcyjna, z powodu powstania klasy abstrakcyjnej, jej szczegółowa implementacja znajduje się w klasach *GameOfLife* oraz *WireWorld*;

5. **generate()**

   Aktualnie:
   + abstract generate(): void

   Poprzednio:
   Brak

   Opis:
   Metoda została dodana jak metoda abstrakcyjna, z powodu powstania klasy abstrakcyjnej, jej szczegółowa implementacja znajduje się w klasach *GameOfLife* oraz *WireWorld*;

3.2.2 WireWorld

- **Metody**

1. **generate()**

   Aktualnie:
   +generate(): void

   Poprzednio:
   +generateAll(int countGenerations): void

   Opis:
   Nazwa metody została zmodyfikowana z powodu zmiany charektru metody. W poprzedniej wersji metoda była odpowiedzialna za wygenerowanie zadanej liczby iteracji, natomiast teraz jest ona odpowiedzialna za generację tylko jednej iteracji zgodnie z zasadami określonymi dla automatu komorkowego *Wire World*;

2. **clearGrid()**

   Aktualnie:
   #clearGrid(Cell[][] gridToClear): void

   Poprzednio: Brak

Opis:
W poprzednim wydaniu metoda była zaimplementowana w klasie *CellularAutomaton*. jednakże metoda ta jest zależna od automatu komórkowego, ponieważ ustawia stan każdej komórki na właściwy dla jej typu, w tym przypadku jest to *Cell.EMPTY*;

3.2.3 GameOfLife

- **Metody**

1. **generate()**
   Aktualnie:
   +generate(): void
   Poprzednio:
   +generateAll(int countGenerations): void
   Opis:
   Nazwa metody została zmodyfikowana z powodu zmiany charektru metody. W poprzedniej wersji metoda była odpowiedzialna za wygenerowanie zadanej liczby iteracji, natomiast teraz jest ona odpowiedzialna za generację tylko jednej iteracji zgodnie z zasadami określonymi dla automatu komorkowego *Game of Life*;

2. **clearGrid()**
   Aktualnie:
   #clearGrid(Cell[][] gridToClear): void
   Poprzednio: Brak

   Opis:
   W poprzednim wydaniu metoda była zaimplementowana w klasie *CellularAutomaton*. jednakże metoda ta jest zależna od automatu komórkowego, ponieważ ustawia stan każdej komórki na właściwy dla jej typu, w tym przypadku jest to *Cell.DEAD*;

3.3 Pakiet Grid

W pakiecie Grid nie zostały wykonane jakiekolwiek zmiany. Zarówno w klasie *Grid* nie zostały zmodyfikowane żadne pola, ani metody, jak i W klasie wyliczeniowej *enum Cell* nie zostały zmodyfikowane żadne atrybuty. Pakiet ten jest w pełni zaimplementowany jak przy oczątkowych założeniach w specyfikacji implementacyjnej;

4 Uniwersalność projektu

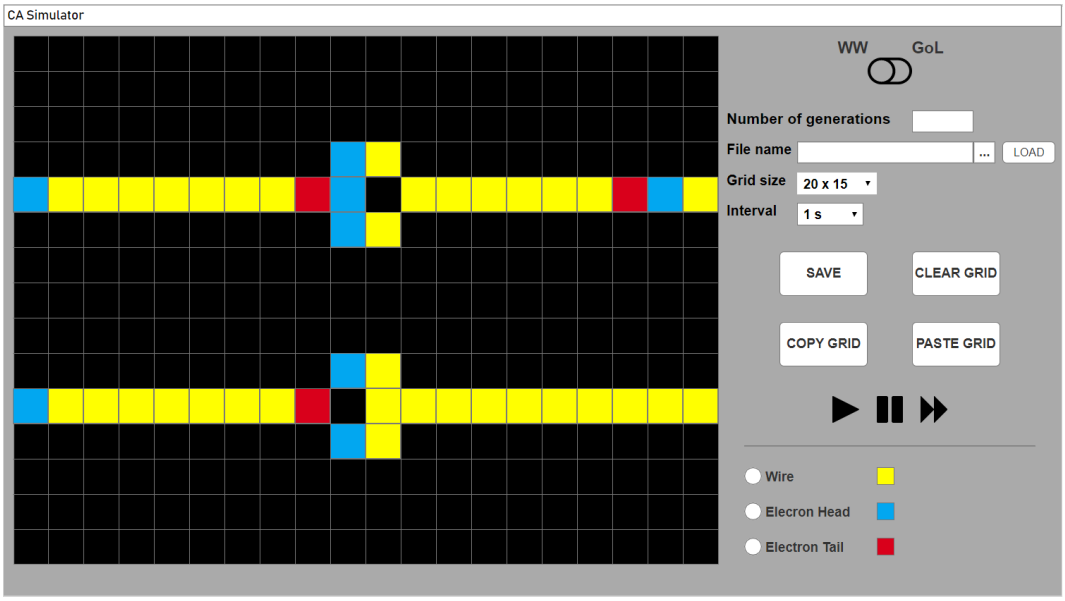
Podczas pisania specyfikacji oraz tworzenia projektu, skupiliśmy się na zachowaniu jego uniwersalności i niezależności od danego automatu komórkowego. Program został zaimplementowany w taki sposób, aby klasa wizualizująca dany automat komórkowy nie miała dostępu do jego typu. Zastosowany został polimorfizm, poprzez stworzenie abstrakcyjnej klasy *CellularAutomaton*, która była klasą bazową klasy *GameOfLife* i klasy *WireWorld*. Pokazuje to, że w ninejszym programie jesteśmy w stanie dodać kolejny automat komórkowy z dużą łatwością. Aby to uczynić. należy zaimplementować pożądaną klasę, jako klasę pochodną klasy bazowej *CellularAutomaton* oraz uzupełnić klasę *enum Cell* o pożądane atrybuty. W tym momencie nie musimy ingerować w żadne połączenia między klasami, a jedynie w część wizualizacyjną pożądanego automatu komórkowego. W celu rozbudowanie programu niezbędna byłaby dodatkowa implementacja wizualizacji pożądanego automatu komórkowego w metodzie *onUpdate()* w klasie *Visualisation*. Kolejnym krokiem byłaby rezygnacja z komponentu *jToggleButtonGoLWW* na rzecz grupy przycisków lub menu. Niezbędną rzeczą byłaby modyfikacja graficznego interfejsu użytkownika i dostosowanie go do pożądanego automatu komórkowego.

5 Graficzny interfejs użytkownika

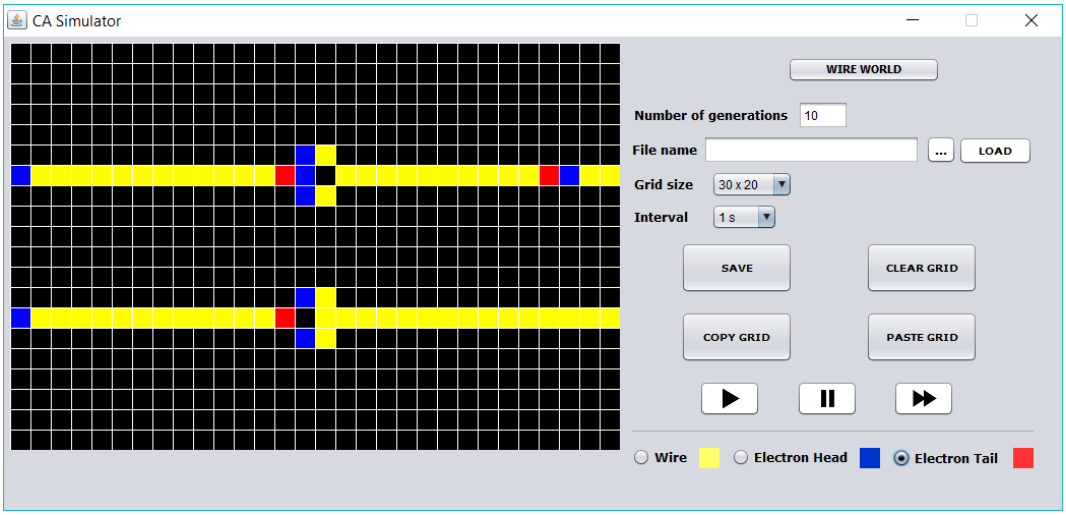
Graficzny interfejs użytkownika znacząco nie odbiega od zaproponowanego interfejsu użytkownika w specyfikacji funkcjonalnej. Końcowy interfejs użytkownika został nieco zmniejszony. Modyfikacji uległ przełącznik odpowiadający za zmianę automatu komórkowego. Kolejna rzecz, która uległa zmianie to wygląd i układ przycisków odpowiadających za obsługę symulacji (przycisk *Play*, przycisk *Pause*, przycisk *OneStep*). Ostatnim komponentem, który uległ zmianie jest grupa przycisków opcji (*Wire*, *Electron Head*, *Electron Tail*), których orientacja została zmieniona na poziomą.

5.1 Wire World

Poniżej przedstawione są dwa graficzne interfejsy użytkownika, przedstawiające automat komórkowy *WireWorld*. Pierwszy z nich jest prototypem, natomiast drugi stanowi finalną koncepcję.



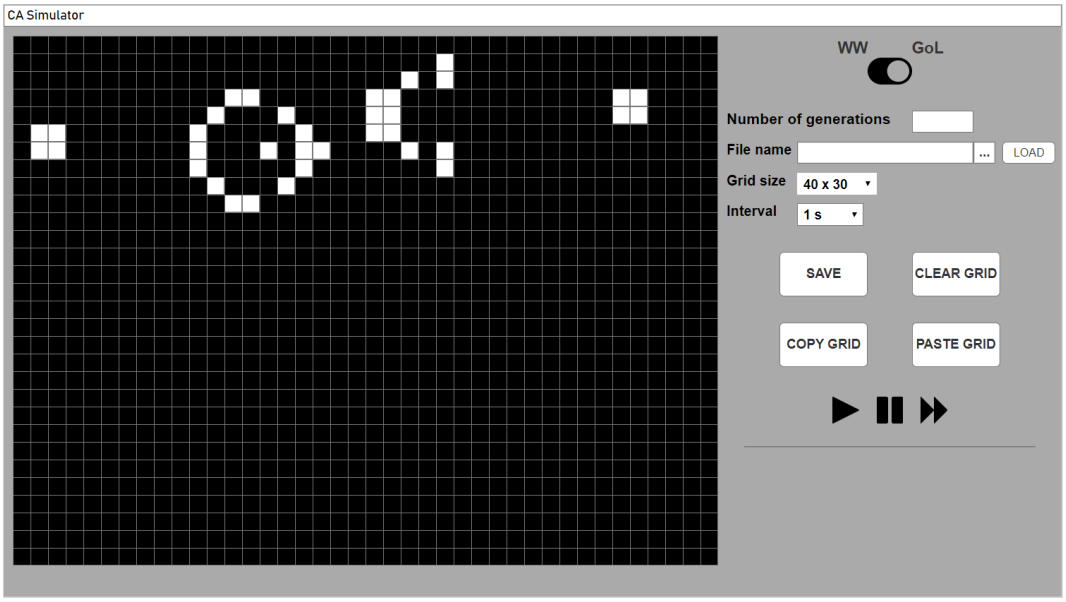
Rysunek 2: Prototyp graficznego interfejsu użytkownika - WireWorld



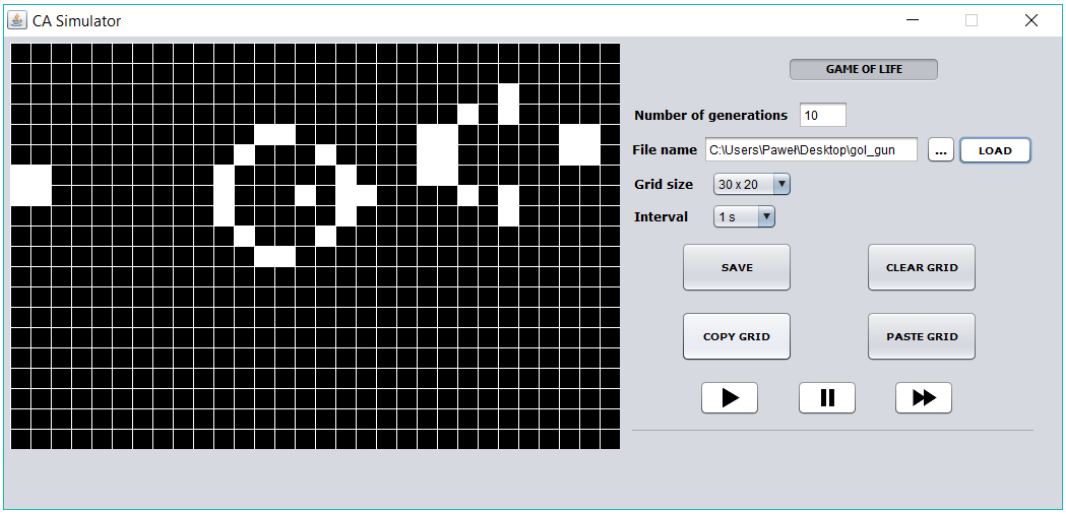
Rysunek 3: Ostateczna wersja graficznego interfejsu użytkownika - WireWorld

5.2 Game of Life

Poniżej przedstawione są dwa graficzne interfejsy użytkownika, przedstawiające automat komórkowy *Game of Life*. Pierwszy z nich jest prototypem, natomiast drugi stanowi finalną koncepcję.



Rysunek 4: Prototyp graficznego interfejsu użytkownika - Game of Life

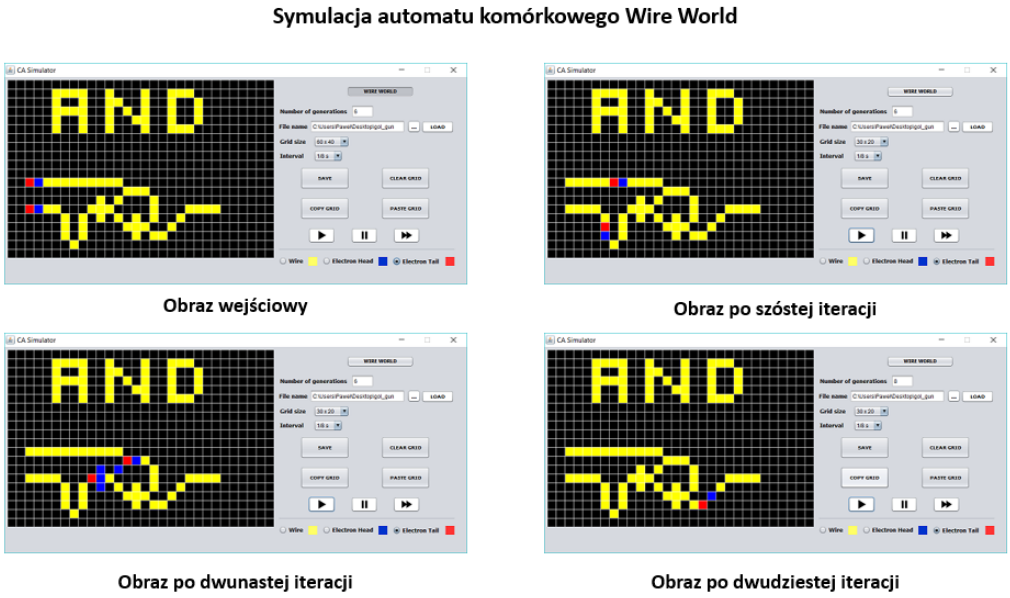


Rysunek 5: Ostateczna wersja graficznego interfejsu użytkownika - Game of Life

6 Prezentacja działania

6.1 Wire World

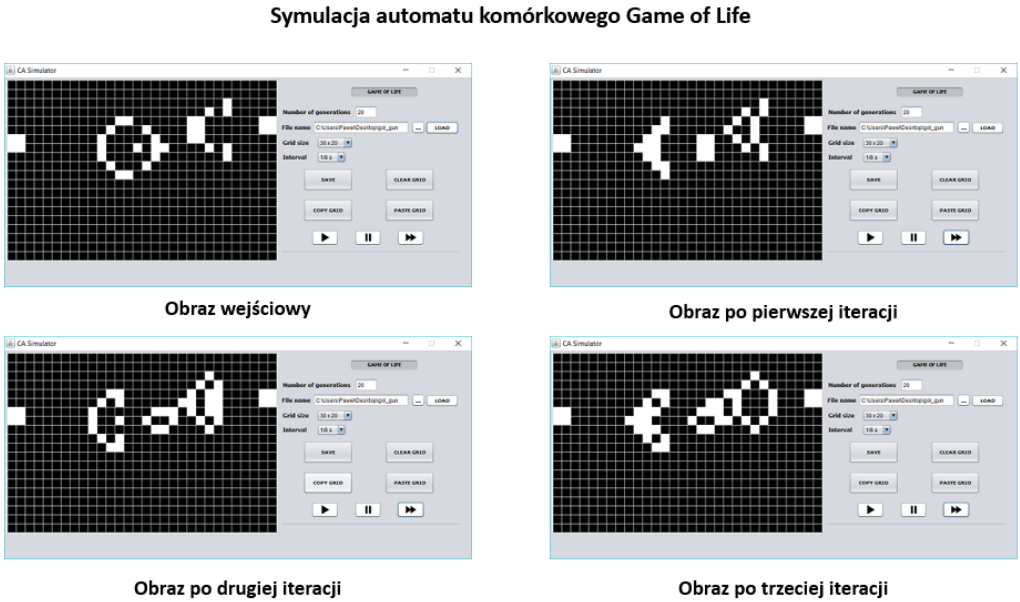
Poniżej zostanie przedstawiona prezentacja działania automatu komórkowego *WireWorld* na przykładzie bramki logicznej *AND*.



Rysunek 6: Symulacja automatu komórkowego Wire World

6.2 Game of Life

Poniżej zostanie przedstawiona prezentacja działania automatu komórkowego *Game of Life* na przykładzie animacji *Glider Gun*.



Rysunek 7: Symulacja automatu komórkowego Game of Life

7 Testy

Do przeprowadzenia testów jednostkowych klas z pakietu Core użyta została biblioteka **TestNG 6.8.1**. Możliwe, że konieczne będzie ręczne załączenie biblioteki. W środowisku **NetBeans IDE 11.0** należy kliknąć prawym przyciskiem myszy na ikonę projektu w oknie **Projects** po lewej stronie. Następnie należy wybrać **Properties** z listy. W oknie, które pojawi się w tym momencie należy wybrać kategorię **Libraries** w oknie **Categories** po lewej stronie. W **Libraries** należy wybrać zakładkę **Compile Tests** i w liście **Classpath** kliknąć '+' po lewej stronie okienka i wybrać **Add Library** z listy. W okienku, które się pojawi należy odszukać bibliotekę **TestNG 6.8.1**, zaznaczyć ją i kliknąć **Add Library**. Ze względu na trudność przeprowadzenia typowego testu jednostkowego klas *ReadFile* i *WriteFile* klasy te przetestowane zostały podczas działania aplikacji.

7.1 CellularAutomatonNGTest

Ta klasa testująca testuje metody z klasy *CellularAutomaton*. Testowane są w niej metody **resizeGameGrid()**, **changeGrid()** oraz **isGridChanging()**. Metody abstrakcyjne tej klasy są testowane osobno w testach jednostkowych klas po niej dziedziczących. W testach pominięte zostały również metody

dostępowe oraz metody związane z implementacją wzorca projektowego *Observer*. Przed każdym testem wywołany jest konstruktor dla obu automatów komórkowych (*WireWorld(int,int) i GameOfLife(int,int)*) celem przygotowania do testów obu automatów z planszami o rozmiarach 3x3.

Metody:

1. **+testResizeGameGrid(): void**
   Opis:
   Metoda testuje metodę *resizeGameGrid()*. Na początku w teście ustalane są wartości *hei* i *wi* oznaczające wysokość i szerokość planszy. W tym przypadku są one większe niż początkowy rozmiar planszy. Następnie dla planszy WireWorld wywoływana jest metoda *resizeGameGrid()* od tych wartości. Następnie metoda *assertTrue()* sprawdza czy obecne wymiary planszy pokrywają się z wartościami dla, których przeprowadzona została zmiana rozmiaru planszy. Dalej testowana metoda wywołana jest dla planszy GameOfLife z wartościami mniejszymi niż początkowy rozmiar planszy. Ponownie metoda *assertTrue()* sprawdza czy faktyczne wymiary planszy są takie same jak wcześniej ustalone wartości;

2. **+testChangeGrid(): void**
   Opis:
   Metoda testuje metodę *changeGrid()*. Na początku plansza *mainGrid* automatu WireWorld zostaje zapełniona wartościami oznaczającymi głowy elektronu, a plansza *utilGrid* tego samego automatu zostaje zapełniona wartościami oznaczającymi ogony elektronu. Następnie dla automatu wywołana zostaje metoda *chageGrid()*. Później metoda *assertTrue()* sprawdza czy metoda *deepEquals()* z klasy **java.util.Arrays** wywołana dla obu plansz zwróci wartość *True*. Następnie analogiczne działania zostają powtórzone dla plansz automatu GameOfLife z tą różnicą, że plansza *mainGrid* zostaje zapełniona komórkami żywymi, a plansza *utilGrid* martwymi;

3. **+testCheckGrid(): void**
   Opis:
   Metoda testuje metodę *checkGrid()*. Na początku plansza *mainGrid* automatu WireWorld zostaje zapełniona wartościami oznaczającymi przewód, a plansza *utilGrid* tego samego automatu zostaje zapełniona wartościami oznaczającymi ogony elektronu. Następnie dla automatu wywołana zostaje metoda *checkGrid()*. Dalej metoda *assertTrue()* sprawdza czy wynikiem działania metody jest zmiana wartości odpowiedniej

flagi na *True*. Potem plansza *utilGrid* zostaje zapełniona takimi samymi komórkami jak *mainGrid*, zostaje wywołana metoda *checkGrid()*, a metoda *assertFalse()* sprawdza czy wartość flagi jest równa *False*. Następnie analogiczne działania zostają powtórzone dla plansz automatu GameOfLife z tą różnicą, że plansza *mainGrid* zostaje zapełniona komórkami martwymi, a plansza *utilGrid* na początku żywymi, a potem martwymi;

7.2 WireWorldNGTest

Ta klasa testująca tstuje metody z klasy *WireWorld*. Testowane są w niej metody **cellState()**, **generate()** oraz **clearGrid()**. W testach pominięte zostały metody dostępowe. Przed każdym testem wywołany jest konstruktor *WireWorld(int,int)* celem przygotowania do testu automatu z lanszami o rozmiarach 3x3.

1. **+testCellState(): void**
   Opis:
   Metoda testuje metodę *cellState()*. W przypadku tego testu obiekt stworzony na potrzeby testu zostaje zastąpiony nowym obiektem *WireWorld* z planszami o wymiarach 5x5. W tym teście testowana metoda jest wywoływana wiele razy i porównywanych jest wiele wyników z oczekiwanymi wynikami za pomocą metody *assertEquals()*. W teście zadbano o to, aby sprawdzić czy metoda poprawnie zwraca wszystkie możliwe wartości komórek. Dla każdej wartości komórki zostały przygotowane i sprawdzone różne przypadki graniczne;

2. **+testGenerate(): void**
   Opis:
   Metoda testuje metodę *generate()*. W przypadku tego testu obiekt stworzony na potrzeby testu zostaje zastąpiony nowym obiektem *WireWorld* z planszami o wymiarach 5x5. W teście plansza *mainGrid* zostaje zapełniona w znany sposób, taki który da znany znany układ planszy w kolejnej generacji. Następnie tworzony jest nowy obekt klasy *WireWorld* o tych samych wymiarach. Plansza *mainGrid* tego obiektu zostaje zapełniona w sposób mając replikować planszę pierwszego obiektu po jednej generacji. Dla pierwszego obiektu zostaje wywołana metoda *generate()*. Na koniec metoda *assertTrue()* sprawdza czy metoda *deepEquals()* wywołana dla plansz *mainGrid* obydwu obiektów zwróci wartość *True*;

3. **+testClearGrid(): void**
   Opis:
   Metoda testuje metodę *clearGrid()*. Na początku dla planszy *mainGrid* zostaje wywołana testowana metoda. Potem plansza *utilGrid* zostaje zapełniona komórkami pustymi przy użyciu dwóch pętli przechodzących przez każdą komórkę tej planszy. Następnie metoda *assertTrue()* sprawdza czy metoda *deepEquals()* wywołana dla obydwu plansz zwróci wartość *True*. Następnie plansza *utilGrid* w taki sam sposób zostaje zapełniona komórkami innymi niż puste. Na koniec metoda *assertTrue()* sprawdza czy metoda *deepEquals()* wywołana dla obydwu plansz zwróci wartość *False*;

7.3 GameOfLifeNGTest

Ta klasa testująca tstuje metody z klasy *GameOfLife*. Testowane są w niej metody **cellState()**, **generate()** oraz **clearGrid()**. W testach pominięte zostały metody dostępowe. Przed każdym testem wywołany jest konstruktor *GameOfLife(int,int)* celem przygotowania do testu automatu z planszami o rozmiarach 3x3.

1. **+testCellState(): void**
   Opis:
   Metoda testuje metodę *cellState()*. W przypadku tego testu obiekt stworzony na potrzeby testu zostaje zastąpiony nowym obiektem *GameOfLife* z planszami o wymiarach 5x5. W tym teście testowana metoda jest wywoływana wiele razy i porównywanych jest wiele wyników z oczekiwanymi wynikami za pomocą metody *assertEquals()*. W teście zadbano o to, aby sprawdzić czy metoda poprawnie zwraca wszystkie możliwe wartości komórek. Dla każdej wartości komórki zostały przygotowane i sprawdzone różne przypadki graniczne;

2. **+testGenerate(): void**
   Opis:
   Metoda testuje metodę *generate()*. W przypadku tego testu obiekt stworzony na potrzeby testu zostaje zastąpiony nowym obiektem *GameOfLife* z planszami o wymiarach 5x5. W teście plansza *mainGrid* zostaje zapełniona w znany sposób, taki który da znany znany układ planszy w kolejnej generacji. Następnie tworzony jest nowy obekt klasy *GameOfLife* o tych samych wymiarach. Plansza *mainGrid* tego obiektu zostaje zapełniona w sposób mając replikować planszę pierwszego obiektu po jednej generacji. Dla pierwszego obiektu zostaje wywołana metoda

generate(). Na koniec metoda *assertTrue()* sprawdza czy metoda *deepEquals()* wywołana dla plansz *mainGrid* obydwu obiektów zwróci wartość *True*;

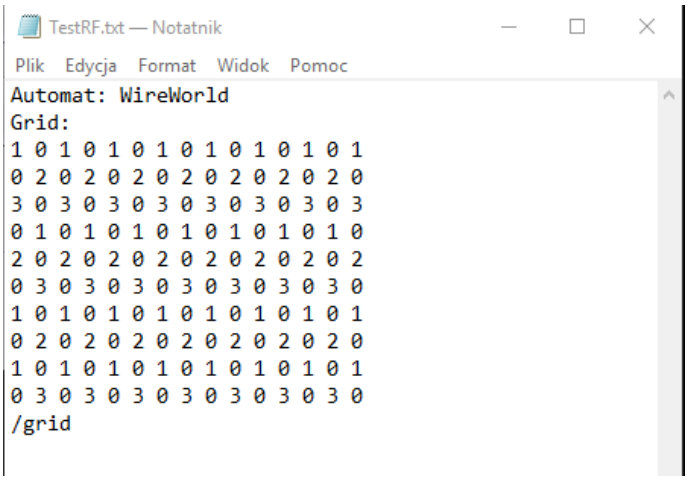
3. **+testClearGrid(): void**
   Opis:
   Metoda testuje metodę *clearGrid()*. Na początku dla planszy *mainGrid* zostaje wywołana testowana metoda. Potem plansza *utilGrid* zostaje zapełniona komórkami martwymi przy użyciu dwóch pętli przechodzących przez każdą komórkę tej planszy. Następnie metoda *assertTrue()* sprawdza czy metoda *deepEquals()* wywołana dla obydwu plansz zwróci wartość *True*. Następnie plansza *utilGrid* w taki sam sposób zostaje zapełniona komórkami innymi niż martwe. Na koniec metoda *assertTrue()* sprawdza czy metoda *deepEquals()* wywołana dla obydwu plansz zwróci wartość *False*;

7.4 Test klasy ReadFile

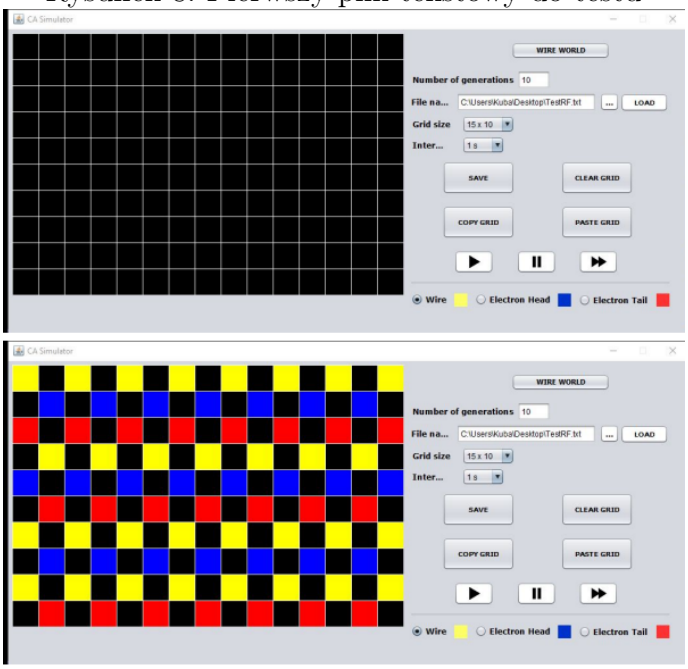
Do przeprowadzenia tego testu napisany został plik tekstowy, po którego załadowaniu program powinien wyświetlić określony układ planszy. Następnie oczekiwany wynik został porównany z faktycznym wynikiem działania aplikacji. Test został przeprowadzony dla różnych automatów i różnych rozmiarów planszy. Na rysunku 10 zamieszczony jest przebieg pierwszego testu dla automatu **WireWorld** o planszy 15x10. Najpierw napisany został plik tekstowy "TestRF.txt", który widnieje na rysunku 8, aby później zaadować go w programie co widać na rysunku 9. Analogiczne działania zostały przeprowadzone dla automatu **GameOfLife** o planszy 30x20. Przygotowany plik tekstowy "TestRF2.txt", który znajduje się na rysunku 11, a wynik załadowania go w programie na rysunku 12.

7.5 Test klasy WriteFile

Do przeprowadzenia tego testu w programie utworzona została plansza, po której zapisaniu program powinien stworzyć znany plik tekstowy. Następnie oczekiany wynik został porównany z faktycznym wynikiem działania aplikacji. Test został przeprowadzony dla różnych automatów i różnych rozmiarów planszy. Pierwszy test polegał na stworzeniu w automacie **WireWorld** o rozmiarach planszy 30x20 przykładowego układu planszy i zapisaniu go do pliku tekstowego "TestWF.txt". Na rysunku 14 zamieszczony został wynik tego testu. Test drugi polegał na analogicznych czynnościach dla automatu **GameOfLife** o planszy 15x10 i z zapisem do pliku "TestWF2.txt"

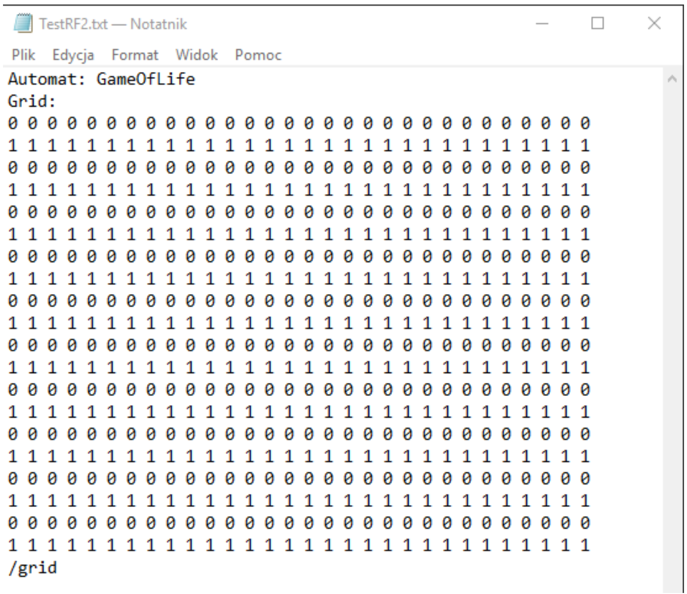


Rysunek 8: Pierwszy plik tekstowy do testu



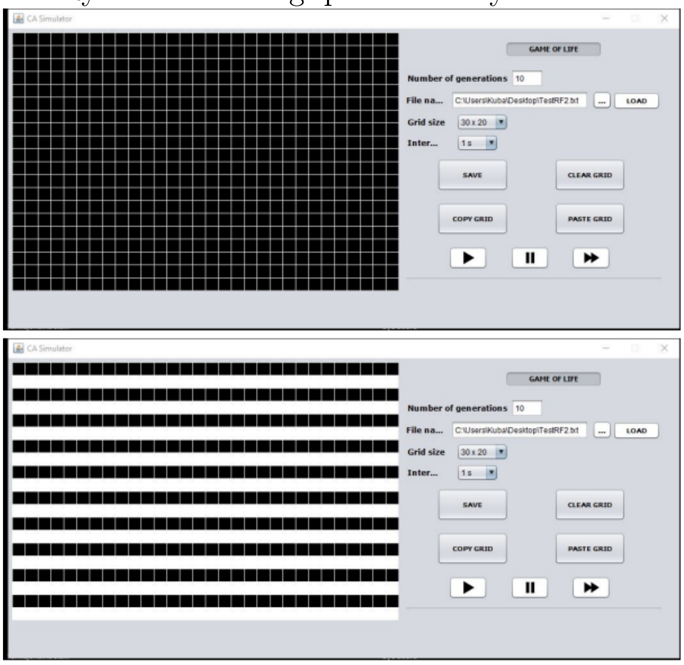
Rysunek 9: Wyniki załadowania pierwszego pliku

Rysunek 10: Pierwszy test ReadFile



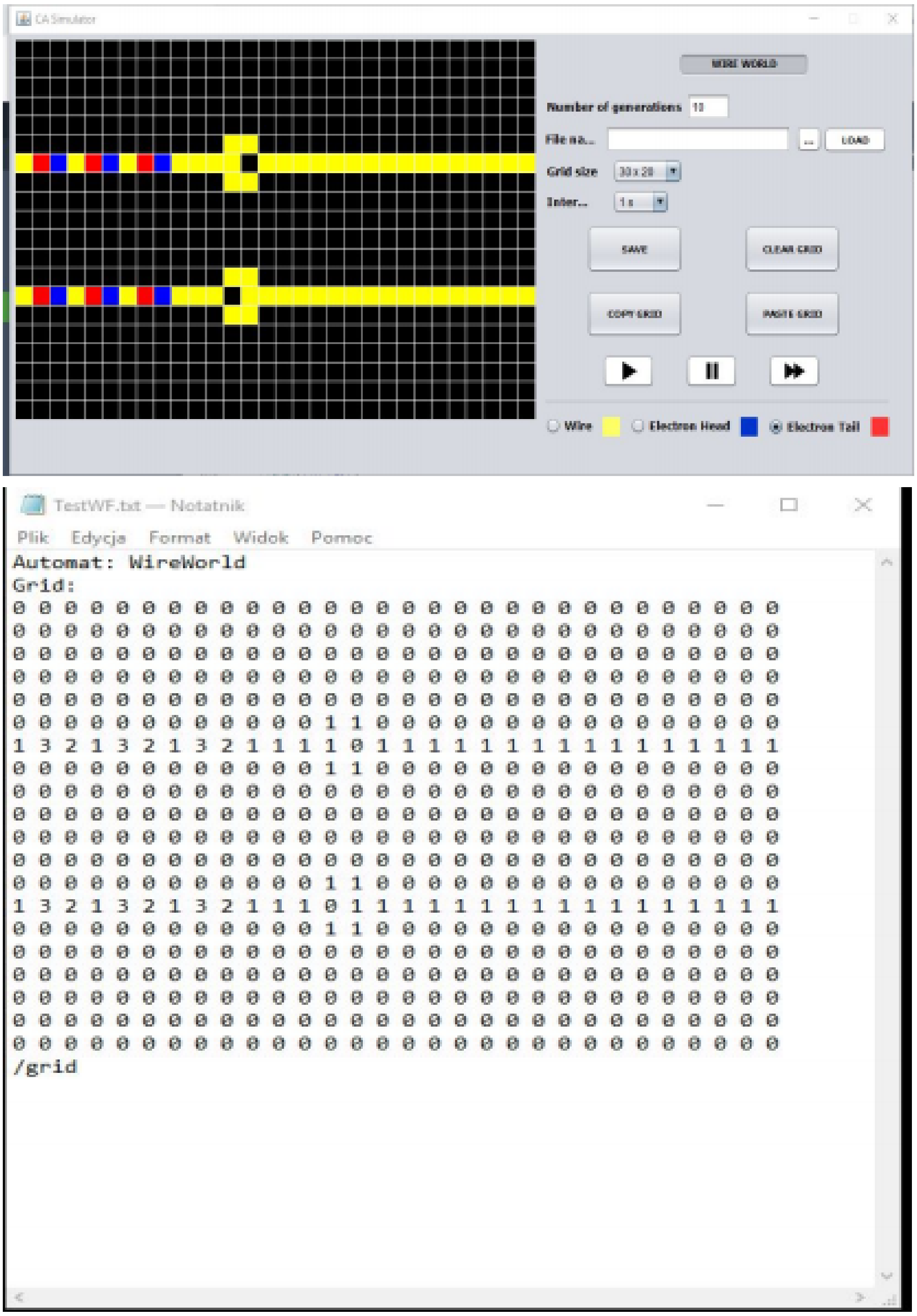
```
Automat: GameOfLife
Grid:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
/grid
```

Rysunek 11: Drugi plik tekstowy do testu

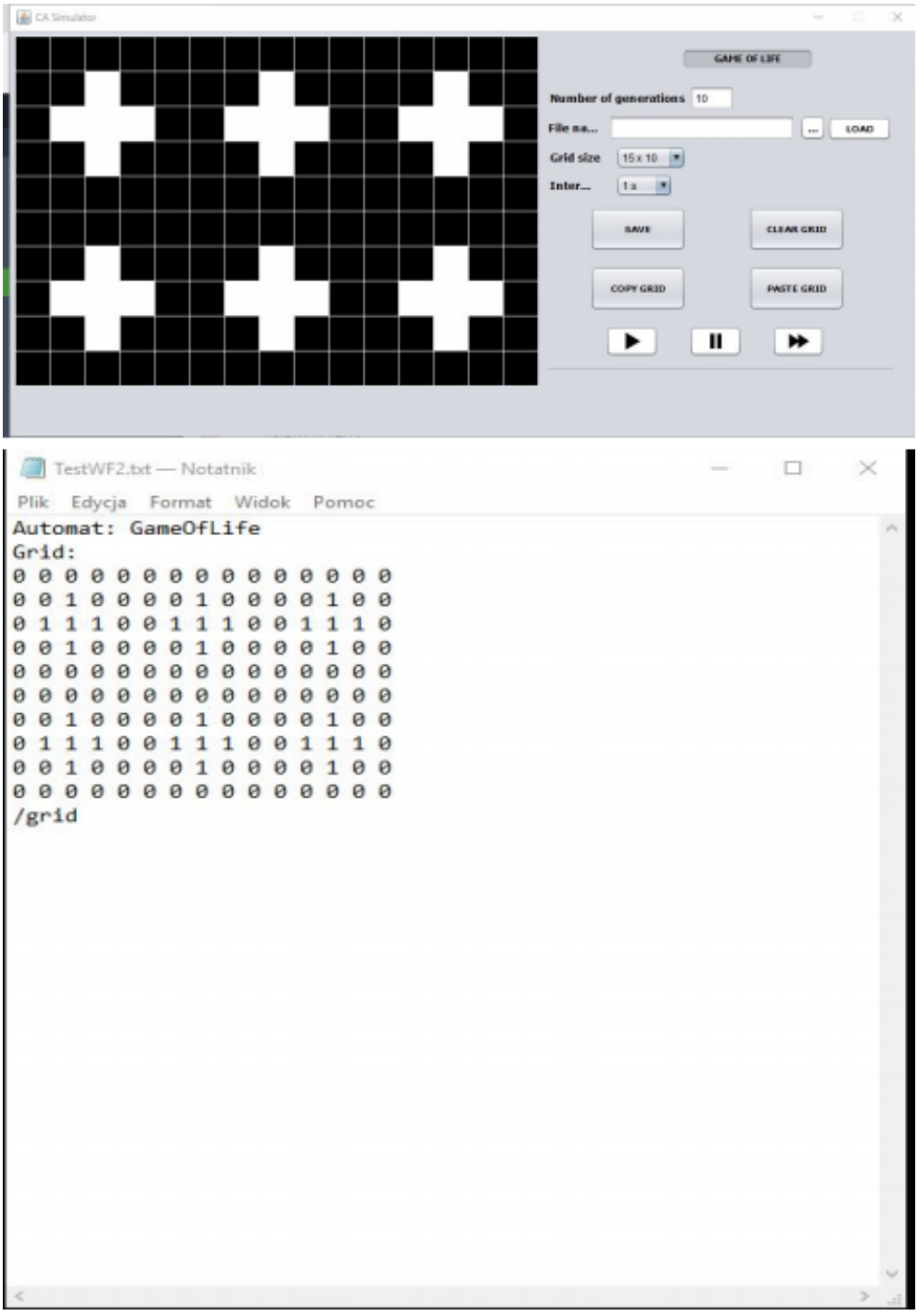


Rysunek 12: Wyniki załadowania drugiego pliku

Rysunek 13: Drugi test ReadFile



Rysunek 14: Wynik pierwszego testu WriteFile



Rysunek 15: Wynik drugiego testu WriteFile