Specyfikacja implementacyjna – Gra w życie

Krzysztof Dąbrowski i Jakub Bogusz

19 maja 2019

Spis treści

1 Projekt systemu **3**
- 1.1 Graficzny interfejs użytkownika . . . 3
- 1.2 Diagram klas . . . 3
- 1.3 Podział na pakiety . . . 3
  - 1.3.1 Pakiet models . . . 3
  - 1.3.2 Pakiet controllers . . . 3
  - 1.3.3 Pakiet views . . . 7
- 1.4 Wzorce projektowe . . . 7
  - 1.4.1 Wzorzec MVC . . . 7
  - 1.4.2 Wzorzec mostu . . . 7
- 1.5 Przepływ sterowania . . . 9
- 1.6 Zastosowane algorytmy . . . 9

2 Opis klas **11**
- 2.1 Package „Controllers” . . . 11
  - 2.1.1 SetupController . . . 11
  - 2.1.2 CellularAutomatonController . . . 11
  - 2.1.3 GameOfLifeController . . . 14
  - 2.1.4 WireWorldController . . . 15
  - 2.1.5 PatternsController . . . 16
  - 2.1.6 GameOfLifePatternsController . . . 17
  - 2.1.7 WireWorldPatternsController . . . 18
- 2.2 Pakiet „Models” . . . 18
  - 2.2.1 CellularAutomaton . . . 19
  - 2.2.2 GameOfLife . . . 20
  - 2.2.3 WireWorld . . . 21
- 2.3 Pakiet „Views” . . . 22
  - 2.3.1 CellularAutomatonView . . . 22
  - 2.3.2 FXCellularAutomatonView . . . 22

3 Testy **23**
- 3.1 Zaplanowane testy: . . . 23
  - 3.1.1 Parsowanie planszy . . . 23
  - 3.1.2 Wczytywanie planszy . . . 24

4 Biblioteki **26**
4.0.1 Java FX 26
4.0.2 Jackson 26
4.0.3 JUnit 26

Rozdział 1

Projekt systemu

1.1 Graficzny interfejs użytkownika

Graficzny interfejs użytkownika zawarty będzie w pliku `.fxml`, napisanym ręcznie lub wygenerowanym przez służący do tego program. Następnie zostanie połączony z logiką działania programu przez klasę `SetupController`.
Główne okno programu zawierać będzie 2 zakładki. Każda z nich będzie obsługiwała i przedstawiała działanie jednego automatu komórkowego.

1.2 Diagram klas

Diagram przedstawiający wszystkie klasy programu i zależności między nimi znajduje się na rysunku 1.1

1.3 Podział na pakiety

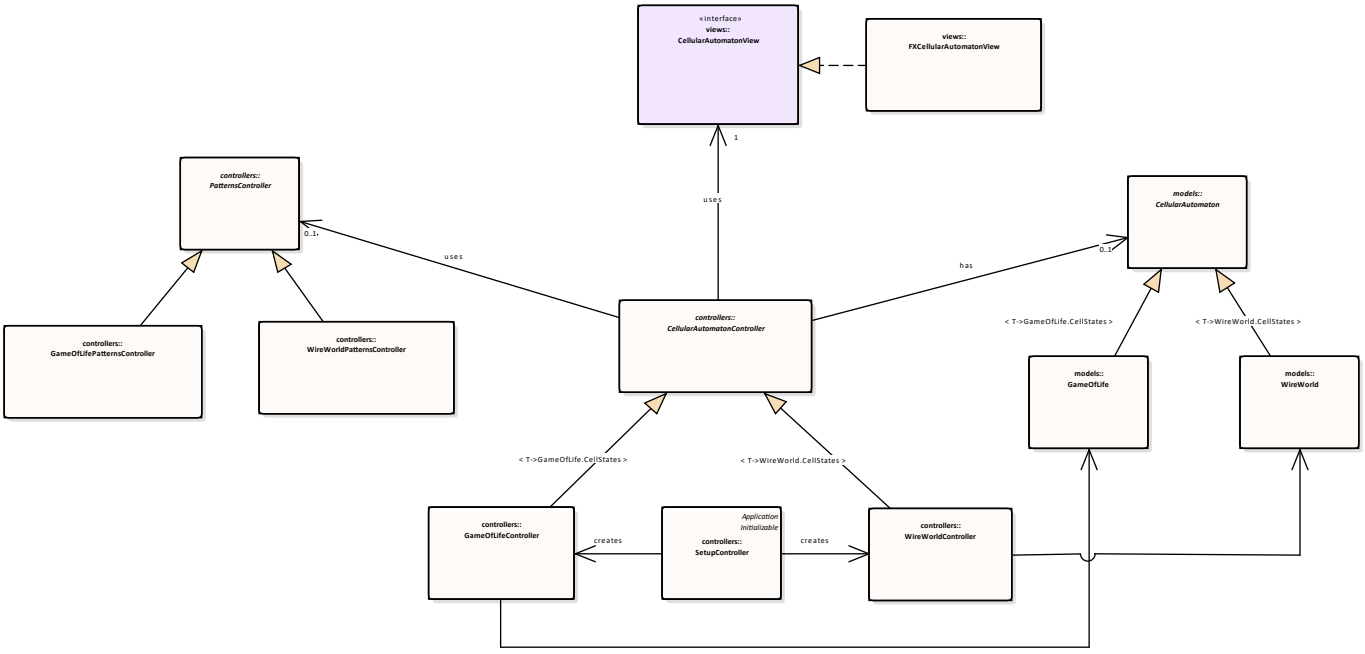
W celu wydzielenia klas projektu względem funkcji, które pełnią zostały one podzielone na pakiety. Przynależności klas do pakietów przedstawia rysunek 1.2.

1.3.1 Pakiet models

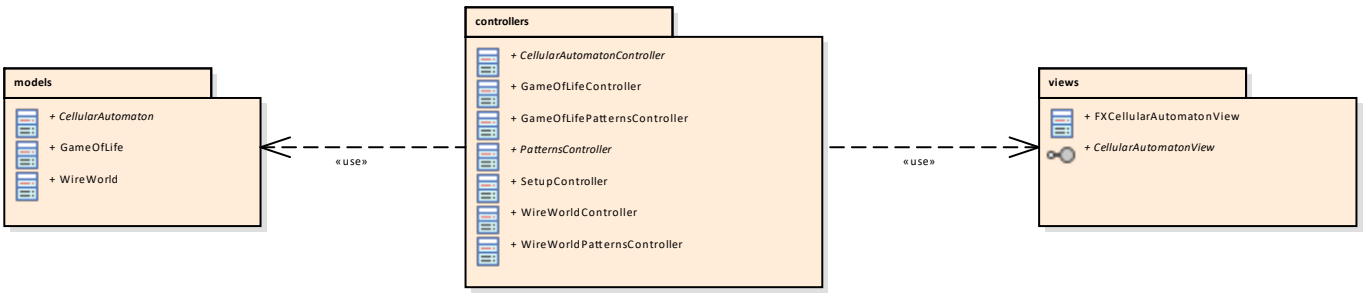
Pakiet ten zawiera klasy modelujące poszczególne automaty komórkowe. Szczegółowy schemat klas z tego pakietu przedstawia diagram klas na rysunku 1.3.

1.3.2 Pakiet controllers

Pakiet ten zawiera klasy przyjmujące dane oraz reagujące na akcje użytkownika. Zażądają one odpowiednimi modelami i widokami. Szczegółowy schemat klas z tego pakietu przedstawia diagram klas na rysunku 1.4.

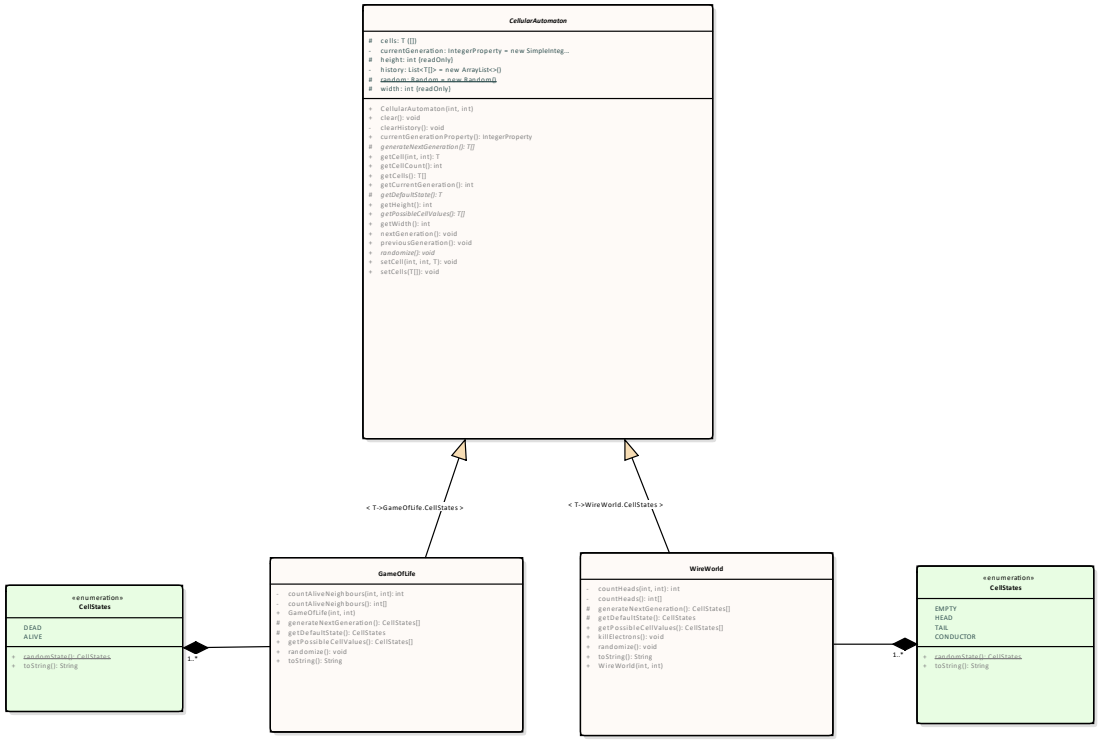


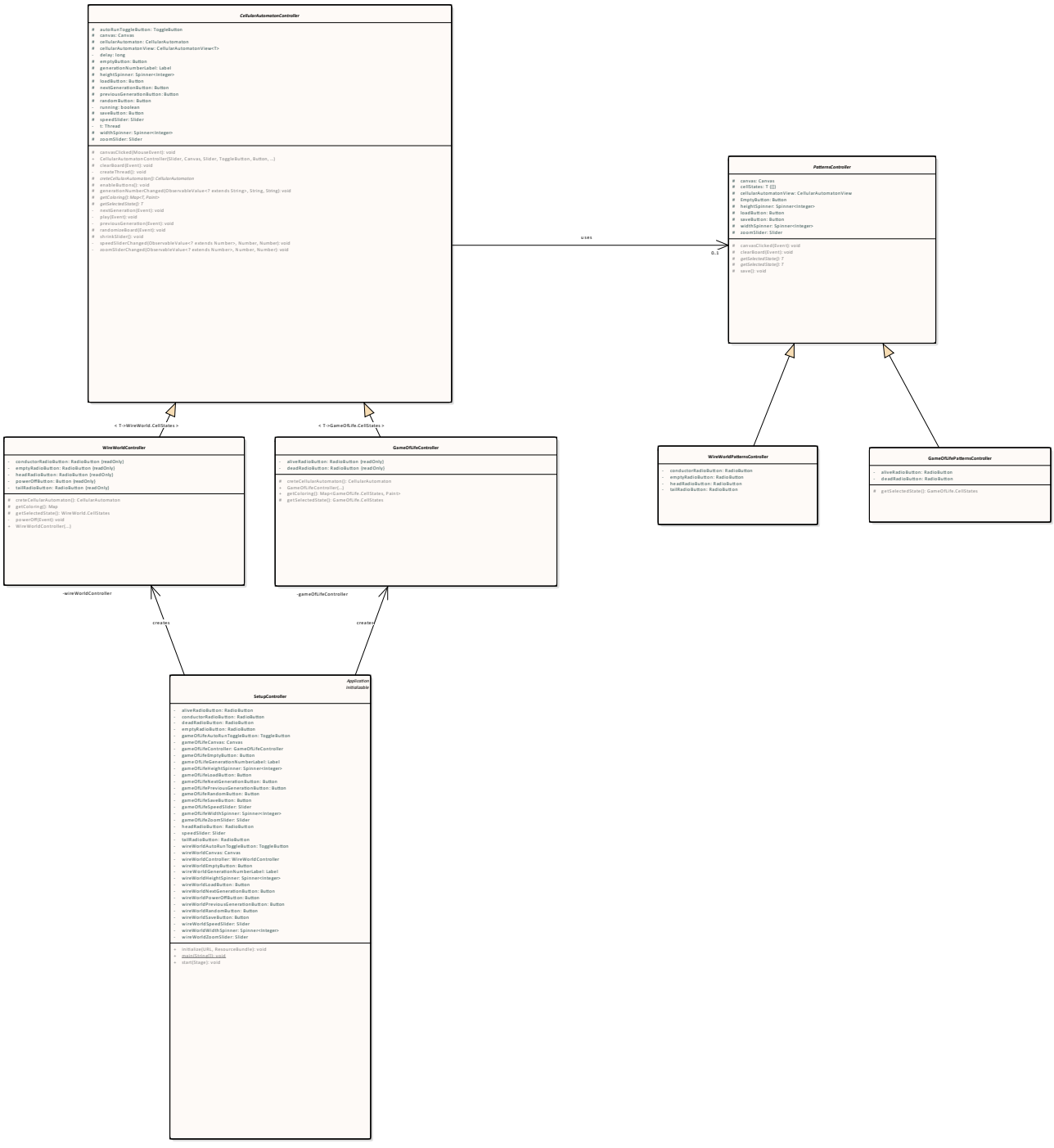
Rysunek 1.1: Diagram klas



Rysunek 1.2: Podział na pakiety

Rysunek 1.3: Pakiet models





Rysunek 1.4: Pakiet controllers

1.3.3 Pakiet views

Pakiet ten zawiera klasy odpowiedzialne za wyświetlanie elementów graficznego interfejsu użytkownika. Szczegółowy schemat klas z tego pakietu przedstawia diagram klas na rysunku 1.5.

1.4 Wzorce projektowe

W celu skorzystania z rozwiązań wymyślonych przez doświadczonych programistów z całego świata w projekcie zostaną zastosowane wzorce projektowe pasujące do napotkanych problemów.

1.4.1 Wzorzec MVC

Aby wprowadzić spójność wśród klas aplikacji, wyznaczyć zakres funkcji realizowany przez dany element oraz sprecyzować zasady komunikacji architektura projektu będzie bazować na wzorcu Model View Controller. Wyraźny podział elementów zaproponowany przez ten wzorzec będzie bezpośrednio realizowany przez podział klas na odpowiadające pakiety.

Klasy z pakietu *models* reprezentują automaty komórkowe Game of Life i WireWorld. Odpowiadają za prowadzenie symulacji.

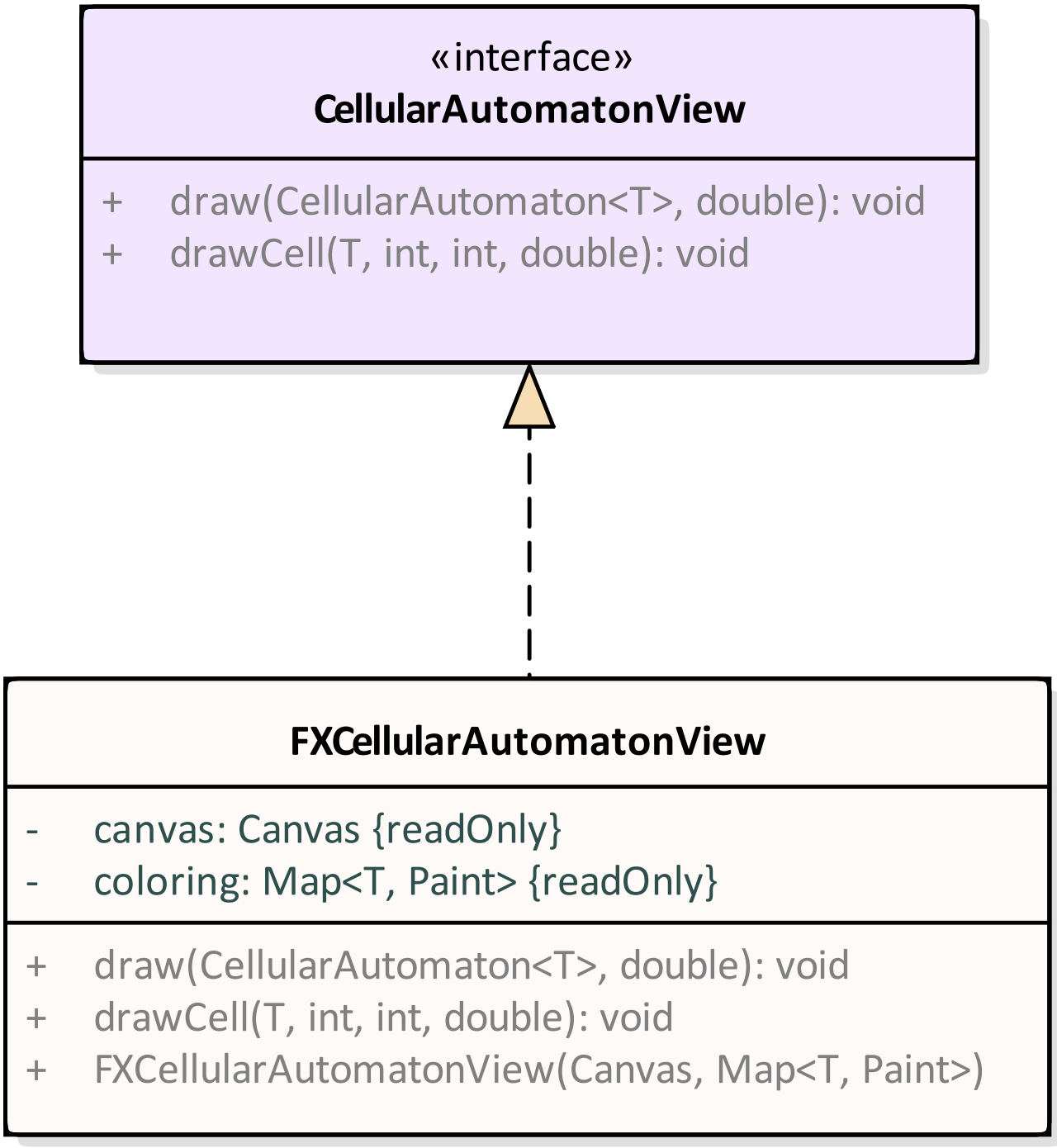
Klasy z pakietu *vies* są odpowiedzialne za wygląd i wyświetlanie interfejsu programu. Są to pliki `.fxml` opisujące wygląd okienek programu oraz klasa `FXCellularAutomatonView` odpowiedzialna za rysowanie automatu komórkowego na płótnie.

Klasy z pakietu *controllers* bezpośrednio reagują na akcje użytkownika. Logika w nich zawarta opowiada za odpowiednią aktualizację modeli oraz odświeżanie widoków.

1.4.2 Wzorzec mostu

Pozwala na oddzielenie abstrakcji, jaką są różne rodzaje automatów komórkowych, od implementacji, czyli poszczególnych widoków wyświetlających automat, tak by każde z nich mogło się zmieniać osobno.

Jest zrealizowany w projekcie poprzez zastosowanie hermetyzacji. Klasy automatów komórkowych zą generalizowane przez klasę abstrakcyjną `CellularAutomaton` dzięki czemu łatwo jest dodać kolejne konkretne automaty do projektu. Klasy widoków rysujących automaty komórkowe implementują wspólny interfejs `CellularAutomatonView`. Pozwala to na proste dodanie nowych widoków rysujących automaty przy pomocy innych bibliotek, czy wyświetlające na inne urządzenia wyjściowe.



Rysunek 1.5: Pakiet views

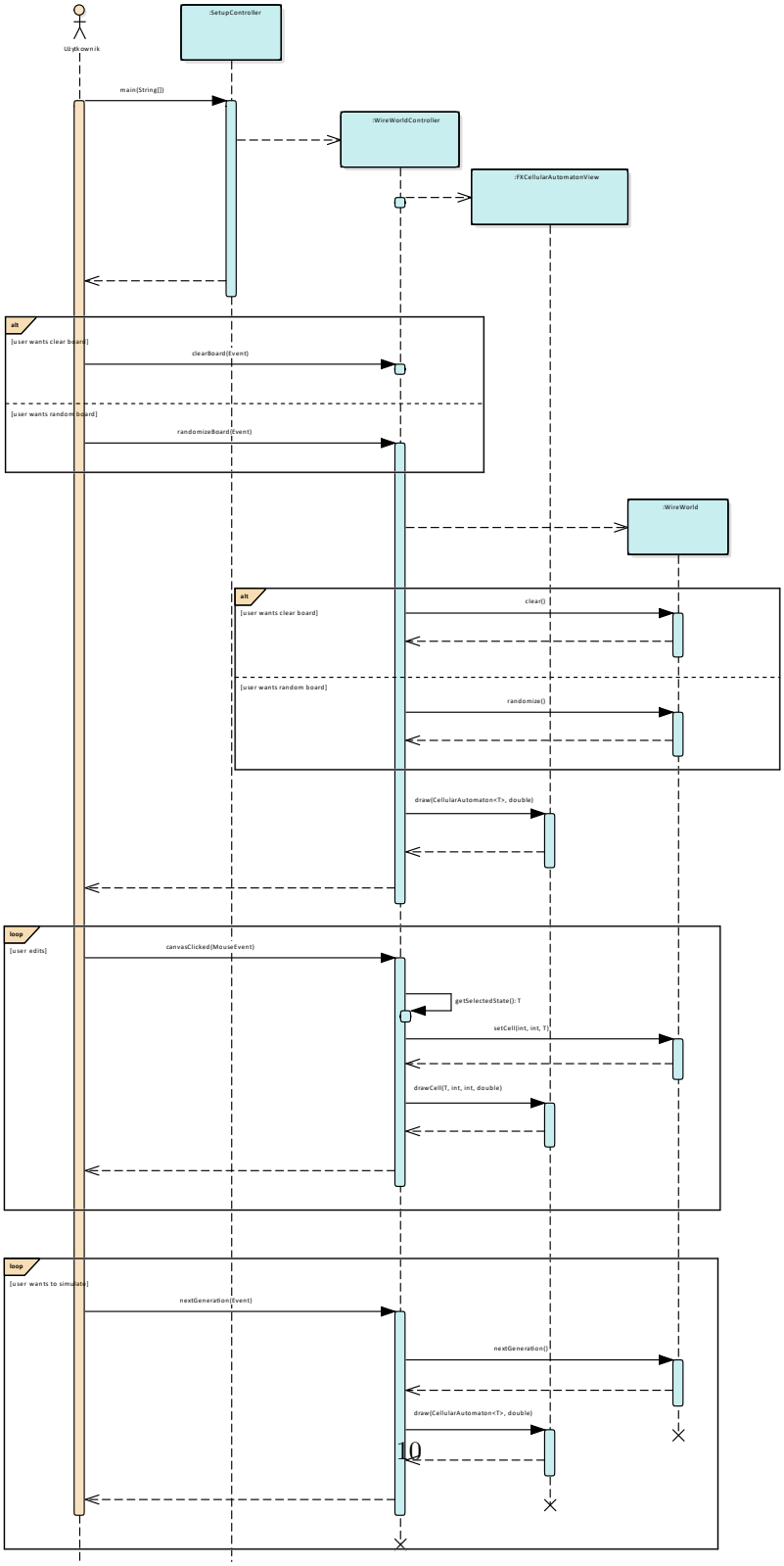
1.5 Przepływ sterowania

Diagram obrazujący przepływ sterowania dla przypadku użycia *Utworzenie automatu WireWorld zmiana kilu wybranych komórek a następnie symulacja przyszłych pokoleń* znajduje się na rysunku 1.6.

1.6 Zastosowane algorytmy

Obydwa algorytmy muszą przejść przez całą plansze i sprawdzić całe otoczenie (8 komórek) do o koła tej aktualnie rozpatrywanej, co daje nam złożoność $8 \cdot n \cdot m$, gdzie n to wysokość planszy, m to szerokość planszy, czyli działają w czasie $\Theta(n \cdot m)$.

Podczas przeglądania otoczenia komórki, automat przepisuje komórki do tablicy wynikowej lub zmienia ich wartość, zgodnie z zasadami opisanymi w Specyfikacji funkcjonalnej.



Rysunek 1.6: Diagram sekwencji

Rozdział 2

Opis klas

2.1 Package „Controllers”

Package składający się z klas mających na celu połączenie graficznego interfejsu użytkownika z logiką działania automatów komórkowych. Będzie zawierać 4 klasy, jedną ogólna ”CellularAutomatonController”, łączącą w sobie cechy wspólne obsługi interfejsu obydwu automatów, oraz z dwóch klas dziedziczących z poprzedniej, zawierających elementu różne dla GameOfLife i WireWorld. Ostatnia z klas będzie pełniła rolę pośrednika między resztą klas a widokiem z pliku `.fxml`.

2.1.1 SetupController

Klasa będąca punktem startowym programu. Jej zadaniem będzie przekazanie referencji do elementów interfejsu kontrolerom konkretnych zakładek.

2.1.2 CellularAutomatonController

`public abstract class CellularAutomatonController<T extends Enum>`
Klasa łącząca w sobie wspólne cechy kontrolerów obydwu automatów skończonych - opisująca powtarzające się elementy graficznego interfejsu użytkownika i ich funkcjonalności.

Pola

Pola chronione:

- `protected Canvas canvas` - płótno na którym rysowana będzie plansza,
- `protected Slider zoomSlider` - suwak reprezentujący przybliżenie planszy ,
- `protected Slider speedSlider` - suwak reprezentujący prędkość wyświetlania kolejnych generacji w trybie automatycznym,

- `protected ToggleButton autoRunToggleButton` - przycisk włączający i wyłączający tryb automatyczny,
- `protected Button nextGeneration` - przycisk służący do stworzenia i wyświetlenia kolejnej generacji,
- `protected Button previousGeneration` - przycisk służący do wyświetlenia poprzedniej generacji,
- `protected Spinner<Integer> widthSpinner` - pole reprezentujące szerokość generowanej planszy,
- `protected Spinner<Integer> heightSpinner` - pole reprezentujące wysokość generowanej planszy,
- `protected Button RandomButton` - przycisk służący do wygenerowania i wyświetlenia losowej planszy początkowej,
- `protected Button EmptyButton` - przycisk służący do wygenerowania i wyświetlenia pustej planszy początkowej,
- `protected Button saveButton` - przycisk służący do zapisania aktualnego stanu planszy,
- `protected Button loadButton` - przycisk służący do wczytania planszy,
- `protected MenuButton menuButton` - przycisk służący do zapisania części planszy lub narysowania i zapisania wzoru,
- `protected Label generationLabel` - napis reprezentujący numer aktualnie wyświetlanej generacji,
- `protected CellularAutomatonView cellularAutomatonView` - obiekt odpowiedzialny za narysowanie planszy.

Pola prywatne:

- `private Boolean running` - zmienna typu prawda/fałsz, określająca czy tryb automatyczny jest włączony,
- `private Thread t` - wątek w którym generowane i wyświetlane są kolejne pokolenia w trybie automatycznym,
- `private long delay` - odstęp czasowy między wyświetlaniem kolejnych generacji w trybie automatycznym.

Metody

Konstruktory:

- `public Controller(Slider speedSlider, Canvas canvas, Slider zoomSlider, ToggleButton autoRunToggleButton, Button previousGenerationButton, Button nextGenerationButton, Spinner widthSpinner, Spinner heightSpinner, Button randomButton, Button emptyButton, Button saveButton, Button loadButton, Label generationNumberLabel)` - metoda odpowiedzialna za zainicjowanie wszystkich zmiennych, połączenie elementów graficznym z odpowiednimi metodami,

Metody publiczne:

- `public void setCanvas` - metoda dostępowa pozwalająca ustawić wartość pola `canvas` funkcjom spoza tego pakietu.

Metody chronione:

- `protected abstract CellularAutomaton createCellularAutomaton()` - metoda odpowiedzialna za stworzenie odpowiedniego modelu automatu komórkowego,
- `protected abstract Map<T, Paint> getColoring()` - metoda zwracająca mapę przyporządkowującą kolory do stanów automatów komórkowych,
- `protected abstract T getSelectedState()` - metoda odpowiedzialna za pobranie aktualnie wybranego stanu na podstawie zaznaczonych zaznaczonych checkbox'ów,
- `protected void randomizeBoard(Event event)` - metoda odpowiedzialna za stworzenie modelu automatu komórkowego z losowo wygenerowaną planszą początkową,
- `protected void randomizeBoard(Event event)` - metoda odpowiedzialna za stworzenie modelu automatu komórkowego z pustą planszą początkową,
- `protected void canvasClicked(Event event)` metoda odpowiedzialna za przekazanie informacji do modelu automatu komórkowego, o tym że należy na podstawie zaznaczonych checkbox'ów należy zmienić stan odpowiedniej komórki oraz ją narysować,
- `protected void shrinkSlider()` - metoda odpowiedzialna za dopasowanie maksymalnej wartość suwaka przybliżenia, tak aby wielkość wyświetlanego obrazu mieściła się w maksymalnym rozmiarze płótna,

- `protected void enableButtons()` - metoda odpowiedzialna za aktywowanie przycisków, które przy starcie programu były nieaktywne ze względu na brak funkcjonalności,
- `protected generationNumberChanged(ObservableValue<? extends Number> observable, Number oldValue, Number newValue)` - metoda odpowiedzialna za aktywowanie i dezaktywowanie przycisku `previousGeneration`, gdy wyświetlenie poprzedniej generacji jest nie możliwe.

Metody prywatne:

- `private createThread()` - metoda odpowiedzialna za stworzenie nowego wątku t,
- `private zoomSliderChanged(ObservableValue<? extends Number> observable, Number oldValue, Number newValue)` - metoda odpowiedzialna za zmianę rozmiaru rysowanych komórek, na podstawie wartości suwaka przybliżenia,
- `private speedSliderChanged(ObservableValue<? extends Number> observable, Number oldValue, Number newValue)` - metoda odpowiedzialna za zmianę prędkości generowania i wyświetlania kolejnych generacji, na podstawie wartości suwaka prędkości,
- `private void nextGeneration(Event event)` - metoda odpowiedzialna za przekazanie informacji do modelu automatu komórkowego, o tym że należy wygenerować następne pokolenie,
- `private void previousGeneration(Event event)` - metoda odpowiedzialna za przekazanie informacji do modelu automatu komórkowego, o tym że należy wygenerować poprzednie pokolenie,
- `private play()` - metoda odpowiedzialna za uruchomienie tryb automatycznego.

2.1.3 GameOfLifeController

`public class GameOfLifeController extends CellularAutomatonController<GameOfLife.CellStat`
Klasa opisująca części kontrolera specyficzne dla automatu GameOfLife.

Pola

Pola chronione:

- `private RadioButton aliveRadioButton` - checkbox oznaczający, że rysowane komórki będą żywe,
- `private RadioButton deadRadioButton` - checkbox oznaczający, że rysowane komórki będą martwe.

Funkcje

Konstruktory:

- `public GameOfLifeController(Slider speedSlider, Canvas canvas, Slider zoomSlider, ToggleButton autoRunToggleButton, Button previousGenerationButton, Button nextGenerationButton, Spinner widthSpinner, Spinner heightSpinner, Button randomButton, Button emptyButton, Button saveButton, Button loadButton, Label generationNumberLabel, RadioButton aliveRadioButton, RadioButton deadRadioButton)` - inicjuje wszystkie zmienne, łączy elementy graficzne z odpowiednimi funkcjami.

Klasa implementuje poniższe metody abstrakcyjne z klasy bazowe:

- `protected CellularAutomaton createCellularAutomaton()`
- `protected Map<GameOfLife.CellStates, Paint> getColoring()`
- `protected GameOfLife.CellStates getSelectedState()`

Szczegółowy opis każdej z metod znajduje się w klasie bazowej CellularAutomatonController.

2.1.4 WireWorldController

Klasa opisująca części kontrolera specyficzne dla automatu WireWorld.

Pola

Pola chronione:

- `private RadioButton emptyRadioButton` – checkbox oznaczający, że rysowane komórki będą puste,
- `private RadioButton headRadioButton` – checkbox oznaczający, że rysowane komórki będą głowami elektronu,
- `private RadioButton tailRadioButton` – checkbox oznaczający, że rysowane komórki będą ogonami elektronu,
- `private RadioButton conductorRadioButton` – checkbox oznaczający, że rysowane komórki będą przewodnikami,
- `private Button powerOff` – przycisk zamieniający wszystkie komórki będące głowami lub ogonami elektronu na przewodniki.

Funkcje

Konstruktory:

- `public WireWorld(Button powerOff, Slider speedSlider, Canvas canvas, Slider zoomSlider, ToggleButton autoRunToggleButton, Button previousGenerationButton, Button nextGenerationButton, Spinner widthSpinner, Spinner heightSpinner, Button randomButton, Button emptyButton, Button saveButton, Button loadButton, Label generationNumberLabel, RadioButton emptyRadioButton, RadioButton tailRadioButton, , RadioButton headRadioButton, , RadioButton conductorRadioButton)` - inicjuje wszystkie zmienne, łączy elementy graficzne z odpowiednimi funkcjami.

Klasa implementuje poniższe metody abstrakcyjne z klasy bazowe:

- `protected CellularAutomaton createCellularAutomaton()`
- `protected Map<WireWorld.CellStates, Paint> getColoring()`
- `protected WireWorld.CellStates getSelectedState()`

Szczegółowy opis każdej z metod znajduje się w klasie bazowej CellularAutomatonController.

2.1.5 PatternsController

`public abstract class PatternsController<T extends Enum>` Klasa łącząca w sobie wspólne cechy kontrolerów edytorów wzorów do obydwu automatów komórkowych.

Pola

Pola chronione:

- `protected Canvas canvas` – płótno na którym rysowana będzie plansza,
- `protected Slider zoomSlider` – suwak reprezentujący przybliżenie planszy,
- `protected Spinner<Integer> widthSpinner` – pole reprezentujące szerokość generowanej planszy,
- `protected Spinner<Integer> heightSpinner` – pole reprezentujące wysokość generowanej planszy,
- `protected Button EmptyButton` – przycisk służący do wygenerowania i wyświetlenia pustej planszy początkowej,
- `protected Button saveButton` – przycisk służący do zapisania wzoru,
- `protected Button loadButton` – przycisk służący do wczytania wzoru,

- `protected T[] cellStates` – tablica reprezentująca tablicę stanów automatu,
- `protected CellularAutomatonView cellularAutomatonView` – obiekt odpowiedzialny za narysowanie planszy.

Pola prywatne:

- `private Map<T, Paint> colouring` - mapa przyporządkowująca stanowi automatu kolor.

Metody

Metody chronione:

- `protected void clearBoard(Event event)` – metoda odpowiedzialna za stworzenie i narysowanie pustej planszy odpowiedniego automatu komórkowego,
- `protected void canvasClicked(Event event)` metoda odpowiedzialna za zmianę stanu komórki na podstawie zaznaczonych checkbox'ów i przekazanie informacji o tym że, trzeba ją narysować,
- `protected abstract T getSelectedState()` – metoda odpowiedzialna za pobranie aktualnie wybranego stanu na podstawie zaznaczonych zaznaczonych checkbox'ów,
- `protected void save()` - metoda odpowiedzialna za zapis wzoru,
- `protected abstract T getSelectedState()` – zwraca wybrany przez użytkownika stan komórki.

2.1.6 GameOfLifePatternsController

`public class GameOfLifePatternsController extends PatternsController<GameOfLife.CellState`
Klasa opisująca części kontrolera specyficzne dla edytora wzorów automatu komórkowego GameOfLife.

Pola

Pola prywatne:

- `private RadioButton deadRadioButton` – checkbox oznaczający, że rysowane komórki będą martwe,
- `private RadioButton aliveRadioButton` – checkbox oznaczający, że rysowane komórki będą żywe.

Metody

Klasa implementuje poniższe metody abstrakcyjne z klasy bazowe:

- `protected GameOfLife.CellStates getSelectedState()`.

Szczegółowy opis każdej z metod znajduje się w klasie bazowej WireWorldPatternsController.

2.1.7 WireWorldPatternsController

`public class WireWorldPatternsController extends PatternsController<WireWorld.CellStates>`
Klasa opisująca części kontrolera specyficzne dla edytora wzorów automatu komórkowego WireWorld.

Pola

Pola prywatne:

- `private RadioButton emptyRadioButton` – checkbox oznaczający, że rysowane komórki będą puste,
- `private RadioButton headRadioButton` – checkbox oznaczający, że rysowane komórki będą głowami elektronu,
- `private RadioButton tailRadioButton` – checkbox oznaczający, że rysowane komórki będą ogonami elektronu,
- `private RadioButton conductorRadioButton` – checkbox oznaczający, że rysowane komórki będą przewodnikami,

Metody

Klasa implementuje poniższe metody abstrakcyjne z klasy bazowe:

- `protected WireWorld.CellStates getSelectedState()`.

Szczegółowy opis każdej z metod znajduje się w klasie bazowej WireWorldPatternsController.

2.2 Pakiet „Models”

Pakiet składający się z klas reprezentujących odpowiednie automaty komórkowe, odpowiedzialnych za przechowywanie ich zasad, przeprowadzanie symulacji i generowanie kolejnych pokoleń. Będzie on zawierać 3 klasy, jedną ogólną `CellularAutomaton`, łączącą w sobie cechy wspólne wszystkich automatów komórkowych oraz 2 klasy dziedziczące z poprzedniej, opisujące działanie konkretnych automatów (`GameOfLife` oraz `WireWorld`).

2.2.1 CellularAutomaton

`public abstract class CellularAutomaton<T extends Enum>`

Klasa abstrakcyjna reprezentująca dowolny automat komórkowy. Typ `T` jest typem wyliczeniowym możliwych stanów komórki automatu.

Konstruktory:

- `public CellularAutomaton(int width, int height)` – Tworzy automat o podanym rozmiarze z komórkami o domyślnym stanie.

Pola chronione:

- `protected final int width` – Liczba kolumn planszy automatu,
- `protected final int height` – Liczba wierszy planszy automatu,
- `protected T[] cells` – Tablica wszystkich komórek automatu,
- `protected static Random random` – Zmienna używana do losowania stanów.

Pola prywatne:

- `private List<T[]> history` – Lista poprzednich stanów automatu. Umożliwia przejście od poprzedniego stanu,
- `private IntegerProperty currentGeneration` – Liczba reprezentująca number aktualnego pokolenia automatu.

Metody publiczne:

- `public abstract T[] getPossibleCellValues()` – Zwraca tablicę możliwych stanów komórki automatu,
- `public void setCells(T[] cells)` – Ustawia wszystkie komórki automatu na nowe wartości,
- `public T getCell(int row, int column)` – Zwraca wartość konkretnej komórki,
- `public int getCellCount()` – Zwraca ilość komórek w automacie,
- `public void nextGeneration()` – Przeprowadza automat do następnego stanu,
- `public void previousGeneration()` – Przeprowadza automat do poprzedniego stanu
- `public void clear()` – Ustawia stan wszystkich komórek na domyślny,
- `public abstract void randomize()` – Ustawia stan wszystkich komórek na losowy.

Metody chronione:

- `protected abstract T[] generateNextGeneration()` – Zwraca stany komórek następnej generacji,
- `protected abstract T getDefaultState()` – Zwraca domyślny stan dla danego automatu.

Metody prywatne:

- `private void clearHistory()` – Ustawia aktualny stan automatu jako jedyny stan w historii.

2.2.2 GameOfLife

`public class GameOfLife extends CellularAutomaton<GameOfLife.CellStates>`
Klasa jest konkretną implementacja automatu komórkowego *Game of Life*.

Klasy wewnętrzne:
Typ wyliczeniowy reprezentujący możliwe stany komórki.

```
public enum CellStates {
    DEAD,
    ALIVE;

    public static CellStates randomState() - Zwraca losowy stan komórki.
}
```

Konstruktory:

- `public GameOfLife(int width, int height)` – Tworzy automat o podanym rozmiarze z komórkami o domyślnym stanie.

Metody prywatne:

- `private int[] countAliveNeighbours()` – Oblicza ilość żywych sąsiadów dla każdej komórki automatu,
- private int countAliveNeighbours(final int cellX, final int cellY) – Oblicza ilość żywych sąsiadów dla konkretnej komórki automatu.

Klasa implementuje poniższe metody abstrakcyjne z klasy bazowe:

- `CellStates[] getPossibleCellValues()`,
- `protected CellStates[] generateNextGeneration()`,
- `public void randomize()`,

- `protected CellStates getDefaultState()`.

Szczegółowy opis każdej z metod znajduje się w klasie bazowej CellularAutomaton.

2.2.3 WireWorld

`public class WireWorld extends CellularAutomaton<WireWorld.CellStates>`
Klasa jest konkretną implementacja automatu komórkowego *WireWorld*.

Klasy wewnętrzne:
Typ wyliczeniowy reprezentujący możliwe stany komórki.

```
public enum CellStates {
    EMPTY,
    HEAD,
    TAIL,
    CONDUCTOR;

    public static CellStates randomState() - Zwraca losowy stan komórki.
}
```

Konstruktory:

- `public WireWorld(int width, int height)` – Tworzy automat o podanym rozmiarze z komórkami o domyślnym stanie.

Metody publiczne:

- `public void killElectrons()` – Zamienia wszystkie głowy elektronów i ogony elektronów na przewodniki.

Metody prywatne:

- `private int countHeads(final int cellX, final int cellY)` – Oblicza ilość sąsiadów konkretnej komórki będących głową elektronu,
- `private int[] countHeads()` – Dla każdej komórki oblicza ilość sąsiadów będących głową elektronu.

Klasa implementuje poniższe metody abstrakcyjne z klasy bazowe:

- `CellStates[] getPossibleCellValues()`,
- `protected CellStates[] generateNextGeneration()`,
- `public void randomize()`,
- `protected CellStates getDefaultState()`.

Szczegółowy opis każdej z metod znajduje się w klasie bazowej CellularAutomaton.

2.3 Pakiet „Views”

Pakiet składający się z interfejsu generalizującego wyświetlanie planszy, który umożliwia dołączanie do projektu nowych implementacji wizualizacji automatu komórkowego oraz z klasy implementującej ten interfejs, renderującej widok 2D przy użyciu obiektu Canvas z biblioteki JavaFX.

2.3.1 CellularAutomatonView

Interfejs generalizujący wizualizacje planszy.

Metody publiczne:

- `public void drawCell(T CellState, final int x, final int y, final double cellSize)` - metoda rysująca pojedynczą komórkę,
- `public void draw(CellularAutomaton ca, final double cellSize)` - metoda rysująca całą planszę, na podstawie tablicy komórek.

2.3.2 FXCellularAutomatonView

Implementacja interfejsu `CellularAutomatonView` przy użyciu obiektu Canvas z biblioteki JavaFX. Wizualizacja 2D automatu komórkowego.

Pola

Pola prywatne:

- `private final Canvas canvas` - płótno, na którym rysowana będzie plansza z aktualnym stanem automatu komórkowego,
- `private final Map<T, Paint> colouring` - mapa przyporządkowująca stanowi automatu kolor.

Konstruktory:

- `public FXCellularAutomatonView(Canvas canvas, Map<T, Paint> colouring )` - inicjacja pól.

Metody publiczne:

- `public void drawCell(T CellState, final int x, final int y, final double cellSize)` - metoda rysująca pojedynczą komórkę,
- `public void draw(CellularAutomaton ca, final double cellSize)` - metoda rysująca całą planszę, na podstawie tablicy komórek.

Rozdział 3

Testy

Poprawność działania poszczególnych metod zostanie zweryfikowane przez przeprowadzenie testów jednostkowych. Do ich realizacji zastosowana zostanie biblioteka **JUnit**.

3.1 Zaplanowane testy:

3.1.1 Parsowanie planszy

Test będzie polegał na zamianie kilku przykładowych obiektów plansz na pliki w formacie `*.xml` oraz `*.json` oraz porównaniu ich z przygotowanymi wcześniej, prawidłowymi plikami. Na przykład:

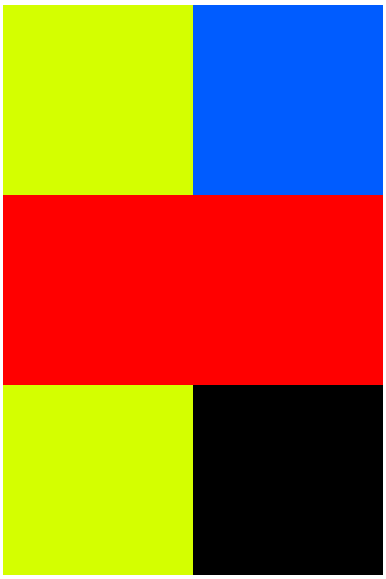
Obiekt o następujących wartościach pól:

```
width = 2
height = 3
cells = WireWorld.CONDUCTOR, WireWorld.HEAD, WireWorld.TAIL, WireWorld.TAIL,
WireWorld.CONDUCTOR, WireWorld.EMPTY
```

Powinien zostać przekonwertowany do plików:

Plik `*.xml`

```
<Board>
  <width>2</width>
  <height>3</height>
  <states>
    <states>CONDUCTOR</states>
    <states>HEAD</states>
    <states>TAIL</states>
    <states>TAIL</states>
    <states>CONDUCTOR</states>
```



Rysunek 3.1: Rysunek 3.1 - Plansza omawiana w punktach 3.1.1. i 3.1.2.

```
    <states>EMPTY</states>
  </states>
</Board>
```

Plik `*.json`

```
{
  "width" : 2,
  "height" : 3,
  "states" : [ "HEAD", "TAIL", "HEAD", "HEAD", "HEAD", "EMPTY" ]
}
```

3.1.2 Wczytywanie planszy

Test będzie polegać na wczytaniu kilku przykładowych plików `*.json` oraz `*.xml`, przekonwertowaniu ich na obiektu odpowiednich klas i porównaniu ich z wcześniej przygotowanymi, prawidłowymi obiektami. Na przykład:

Plik `*.xml`

```
<Board>
  <width>2</width>
  <height>3</height>
  <states>
    <states>CONDUCTOR</states>
```

```
    <states>HEAD</states>
    <states>TAIL</states>
    <states>TAIL</states>
    <states>CONDUCTOR</states>
    <states>EMPTY</states>
  </states>
</Board>
```

Plik `*.json`

```
{
  "width" : 2,
  "height" : 3,
  "states" : [ "HEAD", "TAIL", "HEAD", "HEAD", "HEAD", "EMPTY" ]
}
```

Powinny reprezentować obiekt o następującej strukturze:

```
width = 2
height = 3
cells = WireWorld.CONDUCTOR, WireWorld.HEAD, WireWorld.TAIL, WireWorld.TAIL,
WireWorld.CONDUCTOR, WireWorld.EMPTY
```

3.1.3 Generacja następnego pokolenia

Test będzie polegać na wyznaczeniu następnego pokolenia dla kilku przykładowych planszy oraz porównaniu ich z wcześniej przygotowanymi, poprawnymi planszami. Na przykład:

Obiekt planszy o poniższej strukturze:

```
width = 2
height = 3
cells = WireWorld.CONDUCTOR, WireWorld.HEAD, WireWorld.TAIL, WireWorld.TAIL,
WireWorld.CONDUCTOR, WireWorld.EMPTY
```

Powinien w następnym pokoleniu wygenerować obiekt o poniższej strukturze:

```
width = 2
height = 3
cells = WireWorld.HEAD, WireWorld.TAIL, WireWorld.CONDUCTOR, WireWorld.CONDUCTOR,
WireWorld.CONDUCTOR, WireWorld.EMPTY
```

Rozdział 4

Biblioteki

Zastosowanie bibliotek z poza biblioteki standardowej Javy pozwoli na szybsze i łatwiejsze wprowadzenie do programu wymaganej funkcjonalności. Biblioteki będą instalowane przy pomocy wbudowanego narzędzia w programie IntelliJ IDE.

4.0.1 Java FX

Do przygotowania graficznego interfejsu użytkownika zostanie wykorzystana biblioteka *Java FX*. Została ona wybrana z uwagi na poniższe aspekty.

Jest ona nowsza niż alternatywa w postaci biblioteki Swing oraz nie polega na jeszcze starszych zależnościach. Domyślnie wspiera korzystanie z kontrolerów do obsługi akcji użytkownika oraz oddzielenie widoków od logiki, co pasuje do zastosowanego wzorca MVC. Umożliwia stylizację elementów pryz pomocy języka *CSS*.

Dodatkowo posiada mechanizm *Property*, który pozwoli na łatwe zastosowanie wzorca projektowego obserwator do śledzenia zmian wybranych atrybutów.

4.0.2 Jackson

W celu parsowania oraz serializacji obiektów do plików XML i JSON w projekcie zostanie wykorzystania biblioteka *Jackson*. Pozwala ona na proste wykonywanie wspomnianych zadań. Dodatkową zaletą jest fakt, że obsługuje obydwa typy plików.

4.0.3 JUnit

Do przeprowadzania testów jednostkowych zostanie użyta biblioteka *JUnit*. Jest to standardowe rozwiązanie do przeprowadzania tego typu testów. Pozwala na automatycznie wykonywanie wszystkich testów lub wybranych podgrup testów. Po zakończeniu testów generuje czytelne raporty oraz jest dobrze zintegrowana ze środowiskiem *IntelliJ IDE*.