Krzysztof Dąbrowski i Jakub Bogusz

Raport końcowy —- Gra w życie

8 czerwca 2019

Spis treści

1. Ostateczny projekt klas

2. Opis modyfikacji

Podczas pracy na programem natknęliśmy się na problemy nie przewidziane w specyfikacji implementacyjnej, w związku z czym powstały nowe klasy, a projekt innych został zmodyfikowany.

2.1. SetupController

Dodano (przeniesiono z klasy CellularAutomatonController) deklaracje pól zawierających elementy interfejsu graficznego z pliku `mainWindow.fxml`.

2.2. CellularAutomatonController

Usunięto (przeniesiono do klasy SetupController) deklaracje pól zawierających elementy interfejsu graficznego z pliku `mainWindow.fxml`.

Dodano metodę `protected void saveCurrentGeneration(Event event)` odpowiedzialną za zapis aktualnego pokolenia do pliku o wybranym formacie.

Dodano metodę abstrakcyjną `protected abstract Class getCellularAutomatonInstanceClass()` zwracającą klasę danej instancji automatu.

Dodano zestaw metod umożliwiających obsługę edytora wzorów:

- `protected boolean isInPatternInsertionMode()` – metoda zwracająca informację o tym czy kontroler jest aktualnie w trybie edycji lub wstawiania nowego wzorca,
- `protected void createFigure(Event event)` – metoda odpowiedzialna za otwarcie okna tworzenia figur,
- `protected void editFigure(Event event)` – metoda odpowiedzialna za otwarcie okna edycji wybranej figury,
- `protected FigureEditorController openFigureEditorWindow(Event event)` – metoda odpowiedzialna za stworzenie okna tworzenia figury,
- `protected void addPatternToList(Pattern<T> pattern)` – metoda odpowiedzialna za dodanie nowo utworzonej figury do listy,
- `protected abstract void loadInitialPatterns()` – metoda odpowiedzialna za wczytanie i dodanie do listy bazowych wzorów automatu
- `protected abstract FXMLLoader loadEditorFXMLLoader()` – metoda odpowiedzialna za stworzenie obiektu klasy FXMLLoader służącego do wczytywania plików `*.fxml`.

2.3. GameOfLifeController

Dodano implementację poniższych metod abstrakcyjnych z klasy bazowej:

- `protected FXMLLoader loadEditorFXMLLoader()`,
- `protected Class getCellularAutomatonInstanceClass()`,
- `protected Class loadInitialPatterns()`.

2.4. WireWorldController

Dodano implementację poniższych metod abstrakcyjnych z klasy bazowej:

- `protected FXMLLoader loadEditorFXMLLoader()`,
- `protected Class getCellularAutomatonInstanceClass()`,
- `protected Class loadInitialPatterns()`.

2.5. FigureEditorController

Klasa PatternsController została całkowicie przebudowana: `public abstract class FigureEditorController<T extends Enum> implements Initializable`

Pola chronione:

- `protected Canvas canvas` – płótno na którym rysowany jest wzór,
- `protected TextField figureNameTextField` – pole tekstowe do którego wprowadza się nazwę wzoru,
- `protected Spinner<Integer> widthSpinner` – pole do którego wprowadza się szerokość wzoru,
- `protected Spinner<Integer> heightSpinner` – pole do którego wprowadza się wysokość wzoru,
- `protected Button resetButton` – przycisk odpowiedzialny za resetowanie stanu planszy,
- `protected Button cancelButton` – przycisk odpowiedzialny za anulowanie operacji tworzenia lub edycji wzoru i zamykający okno,
- `protected Button resetSave` – przycisk odpowiedzialny za zapis wzoru oraz dodanie go do listy gotowych wzorów,
- `protected double cellSize` – zmienna przechowująca rozmiar rysowanych komórek,
- `protected CellularAutomatonView<T> cellularAutomatonView` – obiekt odpowiedzialny za rysowanie planszy,

- `protected CellularAutomaton<T> cellularAutomaton` – model automatu mający za zadanie przechowywać stan planszy,
- `protected Consumer<Pattern<T>> saveCallback` – metoda uruchamiana podczas zapisu wzoru,
- `protected Pattern<T> loadedPattern` – obiekt wzoru.

Metody publiczne:

- `public void initialize(URL location, ResourceBundle resources)` – metoda przypisująca elementom interfejsu graficznego ich funkcjonalności,
- `public void initialize(public void setSaveCallback(Consumer<Pattern<T>> saveCallback))` – metoda dostępowa dla pola saveCallBack,

Metody chronione:

- `protected boolean validateFigure()` – metoda sprawdzająca poprawność utworzonej figury (czy wzór został narysowany i nazwany),
- `protected void saveFigure(Event event)` – metoda zapisująca utworzony lub wyedytowany wzór,
- `protected void canvasClicked(MouseEvent event)` – metoda odpowiedzialna za zmianę stanu i przerysowanie kliknięte komórki,
- `protected abstract CellularAutomaton creteCellularAutomaton()` – metoda tworząca model automatu komórkowego,
- `protected abstract T getSelectedState()` – metoda zwracająca wybrany stan komórki,
- `protected abstract Map<T, Paint> getColoring()` – metoda zwracająca mapę przyporządkowującą kolor stanowi automatu.

Metody prywatne:

- `private void createNewDrawingBoard(Event event)` – metoda tworząca model automatu komórkowego w celu przygotowania edytora figur.

2.6. GameOfLifeFigureEditorController

Dodano implementację poniższych metod abstrakcyjnych z klasy bazowej:

- `protected Map getColoring()`,
- `protected CellularAutomaton creteCellularAutomaton()`.

2.7. WireWorldFigureEditorController

Dodano implementację poniższych metod abstrakcyjnych z klasy bazowej:

- `protected Map getColoring()`,
- `protected CellularAutomaton creteCellularAutomaton()`.

2.8. Pattern

Nowa klasa będąca modelem wzoru: `public class Pattern<T extends Enum>`

Pola prywatne:

- `protected int width` – zmienna reprezentująca szerokość wzoru,
- `protected int height` – zmienna reprezentująca wysokość wzoru,
- `protected String name` – zmienna reprezentująca nazwę wzoru,
- `protected T[] cells` – tablica kolejnych komórek wzoru (komórka o współrzędnych (x,y) znajduję się na width·y + x miejscu),
- `protected final UUID id` – identyfikator wzoru.

Konstruktory:

- `public Pattern(int width, int height, T[] cells)` – konstruktor tworzący wzór bez nazwy,
- `public Pattern(String name, int width, int height, T[] cells)` – konstruktor tworzący wzór z nazwą.

Metody publiczne:

- metody dostępowe do pól,
- `public String toString()` – metoda zwracająca nazwę wzoru,
- `public T getCell(int row, int column)` – metoda zwracająca wartość komórki o współrzędnych (row, column).

2.9. CellularAutomaton

Do części metod i pól dodano adnotację „JsonIgnore" w celu pominięcia ich podczas parsowania.

Dodano metodę `public IntegerProperty currentGenerationProperty()` zwracającą numer aktualnie wyświetlanego pokolenia. Poprzez zastosowanie klasy `Property` z JavaFX możliwe jest „obserwowanie" zmian numeru pokolenia.

Dodano metodę `public void insertPattern(Pattern<T> pattern, final int x, final int y)` wstawiającą do automatu komórkowego wybrany wzór.

2.10. WireWorld

Dodano konstruktor umożliwiający utworzenie klasy na podstawie zparsowanego pliku: `public WireWorld(@JsonProperty("width") final int width, @JsonProperty("height")final int height, @JsonProperty("cells") final WireWorld.CellStates[] cells)`.

Sprawdza on poprawność wczytanych informacji – czy rozmiar planszy zgadza się z liczbą komórek.

2.11. GameOfLife

Dodano konstruktor umożliwiający utworzenie klasy na podstawie zparsowanego pliku: `public GameOfLife(@JsonProperty("width") final int width, @JsonProperty("height")final int height, @JsonProperty("cells") final GameOfLife.CellStates[] cells)`.

Sprawdza on poprawność wczytanych informacji – czy rozmiar planszy zgadza się z liczbą komórek.

2.12. Parser

Klasa PatternsController została całkowicie przebudowana: `public class Parser`

Metody publiczne:

- `public static CellularAutomaton loadCellularAutomaton(File file, Class automatonClass)` – metoda tworząca model automatu komórkowego na podstawie pliku `*.json` lub `*.xml`.

2.13. Serializer

Nowa klasa odpowiadająca za konwersję i zapis stanu automatu komórkowego do pliku: `public class Serializer`

Metody publiczne

- `public static void serializeToJson(CellularAutomaton cellularAutomaton, File file)` – metoda zapisująca stan automatu komórkowego do pliku w formacie `*.json`.

- `public static void serializeToXml(CellularAutomaton cellularAutomaton, File file)` – metoda zapisująca stan automatu komórkowego do pliku w formacie `*.xml`.

2.14. Utils

Nowa klasa będąca kontenerem metod niezwiązanych z żadną konktetną klasą używanych w projekcie: `public final class Utils`

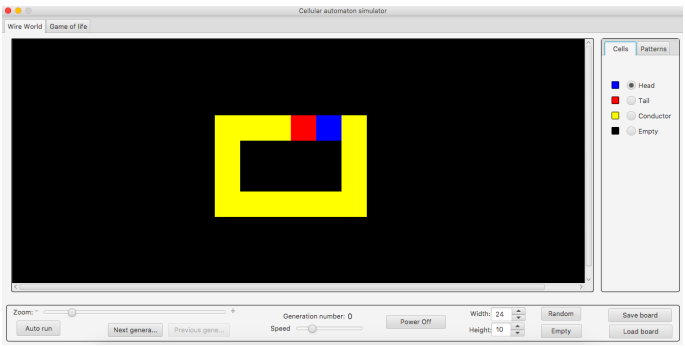
Statyczne metody publiczne:

- `public static boolean isNullOrEmpty(String str)` - metoda sprawdzająca czy dana zamienna z łańcuchem jest niepusta,
- `public static String extractFileExtension(String fileName)` - metoda zwracające rozszerzenie pliku na podstawie jego nazwy,
- `public static void makeSpinnerUpdateValueOnFocusLost(Spinner spinner)` - metoda usprawniająca działanie komponentu `Spinner`, tak żeby aktualizował swoją wartość po faktycznej zmianie zawartości pola (bez oczekiwania na wciśnięcie klawisza „Enter").

3. Prezentacja działania

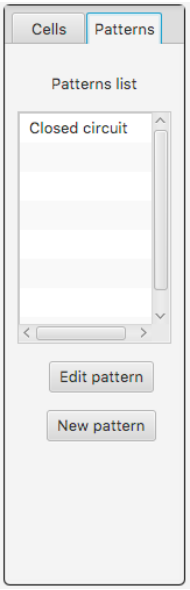
Program pozwala symulować działanie 2 automatów komórkowych - GameOfLife oraz WireWorld. Daje nam możliwość oglądania po kolei następnych pokoleń automatu lub automatycznej generacji. Jesteśmy w stanie kontrolować prędkość symulacji oraz oglądać dowolny fragment planszy z wybranych przez nas przybliżeniem, a także w czasie rzeczywistym, podczas symulacji, edytować komórki na planszy. Program daje również możliwość wczytywania wcześniej przygotowanych plansz oraz zapisu własnych kreacji. Istnieje też edytor figur, który pozwala na proste powielanie wzorów bez konieczności wielokrotnego ich rysowania.

Opis elementów interfejsu:

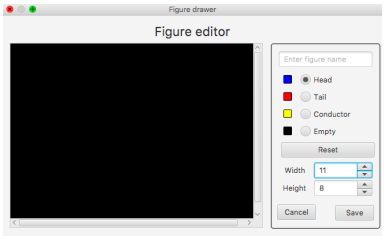


Rysunek 1. 2.1 – GUI WireWorld

Rysunek 2. 2.2 – GUI GameOfLife



Rysunek 3. 2.3 – Zakładka „Patterns” z bocznego menu



Rysunek 4. 2.4 – Edytor wzorów

Elementy wspólne interfejsów obydwu automatów:

- przycisk „Next generation” – przycisk generujący następne pokolenie,

- przycisk „Previous generation” – przycisk wracający do poprzedniej generacji,
- przycisk „Auto run” – przycisk włączający tryb automatycznej generacji kolejnych pokoleń,
- suwak „Zoom” – suwak pozwalający przybliżyć lub oddalić widoczną na ekranie planszę,
- suwak „Speed” – suwak pozwalający zmienić tempo automatycznej generacji następnych pokoleń,
- przycisk „Random” – przycisk odpowiedzialny za stworzenie losowo wygenerowanej planszy,
- przycisk „Empty” – przycisk odpowiedzialny za stworzenie pustej planszy,
- pole „Width” – pole reprezentujące szerokość generowanej planszy,
- przycisk „Height” – pole reprezentujące wysokość generowanej planszy,
- przycisk „Save board ” – przycisk odpowiedzialny za rozpoczęcie procesu zapisu planszy,
- przycisk „Load board ” – przycisk odpowiedzialny za rozpoczęcie procesu wczytywania planszy z odpowiedniego pliku,
- „generation number” – liczba reprezentująca numer aktualnie wyświetlanego pokolenia,
- plansza wyświetlająca aktualny stan automatu,
- zakładka „Patterns” w panelu bocznym:
  — lista dostępnych figur,
  — przycisk „New pattern” – przycisk odpowiedzialny za otwarcie okna służącego do tworzenie nowego wzoru,
  — przycisk „Edit pattern” – przycisk odpowiedzialny za otwarcie okna służącego do edycji istniejącego wzoru.

Elementy właściwe dla automatu GameOfLife:

- pole „DEAD” – stan „pędzla” – rysowane będą komórki martwe,
- pole „ALIVE” – stan „pędzla” – rysowane będą komórki żywe.

Elementy właściwe dla automatu WireWorld:

- pole „TAIL” – stan „pędzla” – rysowane będą komórki reprezentujące ogon elektronu,
- pole „HEAD” – stan „pędzla” – rysowane będą komórki reprezentujące głowę elektronu,
- pole „CONDUCTOR” – stan „pędzla” – rysowane będą komórki reprezentujące przewodnik,
- pole „EMPTY” – stan „pędzla” – rysowane będą komórki puste,
- przycisk „Power off” – przycisk odpowiedzialny za zamianę wszystkich głów oraz ogonów elektronu w przewodniki.

Edytor wzorów:

- plansza służąca do narysowania wzoru,
- pole służące do nazwania wzorca,
- przycisk „Reset” – przycisk służący do zresetowania planszy,
- przycisk „Save” – przycisk służący do zapisania wzorca,
- przycisk „Cancel” – przycisk służący do anulowania procesu edycji lub tworzenia wzorca oraz do zamknięcia okna,
- pole „Width” – pole reprezentujące szerokość tworzonego wzorca,
- przycisk „Height” – pole reprezentujące wysokość tworzonego wzorca.

4. Wzorce projektowe

Zastosowanie wzorców projektowych nadało wyraźną strukturę projektowy. Szczególnie istotny wpływ na postać projektu miał wzorzec MVC. Dzięki niemu

łatwo było wprowadzić nieprzewidziane na poprzednim etapie elementy związanie z edycją figur.

4.1. Nowe wzorce

W trakcie pracy nad projektem oraz poznawania przez nas większej liczby wzorców doszliśmy do wniosku, że należy skorzystać ze wzorców nieprzewidzianych w *Specyfikacji Implementacyjnej*.

Stan
Ponieważ główne okno symulacji udostępnia możliwość wstawiania figur jak i rysowania poszczególnych komórek za pomocą klikania widoku automatu myszą należało wyróżnić te dwie sytuacje. W tym celu został zastosowany wzorzec *Stan*. `CellularAutomatonController` może znajdować się w stanie rysowania komórek lub wstawiania figur i w zależności od swojego stanu inaczej zareaguje na kliknięcie myszą.

Strategia
Podczas pisania klasy `FigureEditorController` okazało się, że sporo z jej zachowań jest wspólnych z klasą `CellularAutomatonController`. By zenkapsulować te zachowania i ograniczyć duplikację kodu można by skorzystać ze wzorca *Strategii*.

Niestety nie mamy doświadczenia w korzystanie z tego wzorca i nie wystarczyło czasu na jego wprowadzenie. Przy refaktoryzacji należałoby wprowadzić ten wzorzec.

5. Potencjalna rozbudowa – dodanie nowego automatu komórkowego

Schemat postępowania w celu dodania nowego automatu komórkowego:

- stworzyć nową kartę interfejsu graficznego w pliku `mainWindow.fxml` zawierającą sterowanie właściwego dla dodawanego automatu,
- zaimplementować kontroler dla danego automatu,
- zaimplementować model logiczny działania danego automatu,
- połączyć kontroler i model automatu,
- utworzyć pola będące referencjami do elementów interfejsu graficznego z pliku ***.fxml**,
- utworzyć obiekt modelu automatu w metodzie initialize klasy SetupController
- **potencjalnie:** zaimplementować edytor figur dla danego automatu.

5.1. Dodanie modelu nowego automatu

Dzięki wykorzystaniu typów szablonowych dodanie modelu nowego automatu jest bardzo proste. W tym celu należy:

1. Utworzyć typ wyliczeniowy reprezentujący możliwe stany automatu,
2. Rozszerzyć klasę `CellularAutomaton` konkretyzując T jako typ wyliczeniowy stanów automatu,
3. Napisać metodę zwracającą stan następnego pokolenia na podstawie aktualnego,
4. Uzupełnić metody pomocnicze takie jak zwracanie losowego stanu.

Ponieważ większość zachowań jest realizowanych przez klasę abstrakcyjną `CellularAutomaton` tworzenie nowych automatów sprowadza się tylko do opisania jak będą generowane następne pokolenia.

Dzięki temu, że jest to odpowiedzialność automatu a nie poszczególne komórki możliwe są automaty delegujące generację następnego stanu do swoich komórek oraz niesymetryczne automaty, których stan następnego pokolenia może zależeć od czynników zewnętrznych.

6. Zmiana interfejsu programu

W związku z faktem, iż modele automatów komórkowych są niezależne od interfejsu użytkownika, można ich używać w połączeniu z dowolnym innym interfejsem – inną biblioteką graficzną, trybem wsadowym itp. Sposób działania tych modeli opisany jest w specyfikacji implementacyjnej. Zmiany pozwalające na współpracę modeli z Parserm i Serializerem plików typu JSON lub XML – wyżej w tym dokumencie.

7. Podsumowanie testów jednostkowych

Przy pomocy biblioteki *JUnit* wygenerowaliśmy raport z testów w formacie html. Jest on dostępny pod linkiem.