Wzorce projektowe cz.1

Projektowanie obiektowe

Laboratorium 3

Natalia Brzozowska

Wiktor Tarsa

1. Wzorzec Builder

1.1 klasa MazeBuilder

Utworzyliśmy interfejs **MazeBuilder**. Interfejs ten będzie implementowany przez klasy tworzące instancje klasy **Maze** reprezentującej labirynt w naszej aplikacji.

```
package pl.agh.edu.dp.labirynth;

public interface MazeBuilder {

   public void maze();
   public Room room(int number);
   public Door door(Room r1, Room r2, Direction d1, Direction d2);

   public void setSide(Direction side, Room room);
   public Maze toMaze();
}
```

Atrybuty oraz metody umieszczone w interfejsie odpowiadają składowym labiryntu.
W metodzie **createMaze** klasy **MazeGame** znajdowała się jeszcze jedna metoda - **addRoom**, która nie została dodana do interfejsu. Interfejs ma być implementowany przez klasy, które będą tworzyć labirynt. Według naszej wizji aplikacji nie ma potrzeby tworzyć pokoju w obrębie labiryntu nie dodając go do niego. Z tego powodu uważamy, że dodawanie pokoju może zostać zrealizowane w metodzie **room(int number)**, natychmiast po stworzeniu nowej instancji typu **Room**.
Do interfejsu został dodany nagłówek funkcji **Maze toMaze()**. Zgodnie z zasadami wzorca projektowego interfejs Builder powinien posiadać taką funkcję po to, by móc w dowolnym momencie zapisać aktualny stan labiryntu do nowej instancji obiektu typu **Maze**.

1.2 modyfikacja funkcji createMaze()

Zmodyfikowaliśmy funkcję **createMaze** - funkcja przyjmuje teraz jako argument instancję klasy **MazeBuilder**. Następnie skorzystaliśmy z funkcji dostarczanych przez interfejs, w celu utworzenia labiryntu.

```
package pl.agh.edu.dp.labirynth;

public class MazeGame {
   public Maze createMaze(MazeBuilder builder){
       builder.maze();

       Room r1 = builder.room(1);
       Room r2 = builder.room(2);

       builder.setSide(Direction.North, r1);
       builder.setSide(Direction.East, r1);
       builder.setSide(Direction.South, r1);
       builder.setSide(Direction.West, r1);

       builder.setSide(Direction.North, r2);
       builder.setSide(Direction.East, r2);
       builder.setSide(Direction.South, r2);
       builder.setSide(Direction.West, r2);
       Door door = builder.door(r1, r2, Direction.South, Direction.North);

       return builder.toMaze();
   }
}
```

1.3 analiza zmian

Zmiany dokonane w poprzednich podpunktach wymuszają na programiście stosowania określonej nomenklatury w każdej klasie, która będzie realizować budowanie labiryntu. Można utworzyć wiele różnych klas, które będą budowały labirynty w różny sposób, jednak dodawanie elementarnych części będzie zawsze przebiegało w ten sam, określony przez interfejs MazeBuilder sposób.
Dodanie interfejsu zwiększa estetykę kodu - każda operacja związana z budowaniem labiryntu odbywa się w instancji **MazeBuildera**. W określonych sytuacjach (np w przypadku metody **addRoom**) można przenieść całą funkcjonalność metody do innej metody - upraszczając korzystanie z klasy.

1.4 klasa StandardBuilderMaze

Utworzyliśmy klasę będącą implementacją Maze Buildera. Zawiera ona wszystkie potrzebne metody do zbudowania pokoju oraz dodatkowo metodę **commonWall** określającą kierunek ściany między dwoma pomieszczeniami.

```
blic class StandardBuilderMaze implements MazeBuilder {
   private Maze currentMaze;

   public StandardBuilderMaze(){ }

   @Override
   public void maze() {
       this.currentMaze = new Maze();
   }

   @Override
   public Room room(int number) {
       Room room = new Room(number);
       this.currentMaze.addRoom(room);
       return room;
   }

   @Override
   public Door door(Room r1, Room r2, Direction d1, Direction d2) {
      Door door = new Door(r1, r2);
      r1.setSide(d1, door);
      r2.setSide(d2, door);
      return door;
   }

   @Override
   public void setSide(Direction side, Room room) {
       room.setSide(side, new Wall());
   }

   @Override
   public Maze toMaze() {
       return this.currentMaze;
   }

   public Direction commonWall(Room r1, Room r2){
       ArrayList<Direction> directions = new ArrayList<Direction>(Arrays.asList(
               Direction.North, Direction.South,
               Direction.East, Direction.West));
       for(Direction direction: directions){
           if(r1.getSide(direction) instanceof Door){
```

```
                Door door = (Door) r1.getSide(direction);
                if(door.getRoom1().equals(r1) && door.getRoom2().equals(r2) ||
                door.getRoom1().equals(r2) && door.getRoom1().equals(r1)) return
direction;
            }
        }
        return null;
    }
}
```

1.5 tworzenie labiryntu korzystając z klasy StandardMazeBuilder

W funkcji statycznej **main()** klasy **Main** utworzyliśmy nową instancję klasy **StandardMazeBuilder**. Wykorzystaliśmy ją do utworzenia labiryntu w funkcji **createMaze** klasy **MazeGame**.

```
public static void main(String[] args) {

    MazeGame mazeGame = new MazeGame();
    MazeBuilder builder = new StandardBuilderMaze();
    Maze maze = mazeGame.createMaze(builder);

    System.out.println(maze.getRoomNumbers());
    Vector<Room> rooms = maze.getRooms();
    for(Room room: rooms){
        Vector<Door> doors = room.getDoors();
        System.out.println(doors.size());
    }
}
```

Można dopisać odpowiednie gettery w klasach Maze i Room po to, by móc sprawdzić, czy dodawane do labiryntu drzwi są powiązane z pokojami.

```
//Maze
public Vector<Room> getRooms() { return rooms; }

//Room
public Vector<Door> getDoors() {
    ArrayList<Direction> directions = new ArrayList<>(Arrays.asList(
            Direction.North, Direction.East,
            Direction.South, Direction.West
    ));
    Vector<Door> doors = new Vector<>();
    for(Direction direction: directions){
        if(this.getSide(direction) instanceof Door)
```

```
            doors.add((Door)this.getSide(direction));
    }
    return doors;
}
```

Po uruchomieniu programu otrzymaliśmy rezultat zgodny z oczekiwanym.

1.6 klasa CountingMazeBuilder

W tym podpunkcie zakładamy, że liczby zapisywane przez klasę **CountingMazeBuilder** są ilością utworzonych obiektów od początku istnienia labiryntu - dlatego też np. nie zmniejszamy liczby utworzonych ścian w momencie dodawania drzwi. Nie bierzemy pod uwagę aktualnej liczby obiektów.

Zaimplementowaliśmy klasę CountingMazeBuilder:

```
public class CountingMazeBuilder implements MazeBuilder{
    int rooms;
    int doors;
    int walls;
    @Override
    public void maze() {
        this.rooms = 0;
        this.doors = 0;
        this.walls = 0;
    }

    @Override
    public Room room(int number) {
        this.rooms++;
        return new Room(number);
    }

    @Override
    public Door door(Room r1, Room r2, Direction d1, Direction d2) {
        this.doors++;
        return new Door(r1, r2);
    }

    @Override
    public void setSide(Direction side, Room room) { this.walls++; }

    @Override
    public Maze toMaze() { return null; }

    public void getCounts(){
```

```
       System.out.println("Liczba utworzonych \npokoi: " + this.rooms + "\ndrzwi: "
+ this.doors + "\nścian: " + this.walls);
   }

}
```

Następnie w funkcji main utworzyliśmy labirynt.

```
public static void main(String[] args) {

   MazeGame mazeGameCount = new MazeGame();
   CountingMazeBuilder builderCount = new CountingMazeBuilder();
   Maze mazeCount = mazeGameCount.createMaze(builderCount);
    builderCount.getCounts();

}
```

Wyniki są zgodne z przewidywaniami.

```
/usr/lib/jvm/java-8-openjdk-amd64/bin/java ...
Liczba utworzonych
pokoi: 2
drzwi: 1
ścian: 8
```

2. Fabryka abstrakcyjna

2.1 klasa MazeFactory

Utworzyliśmy klasę **MazeFactory**, która tworzy poszczególne elementy labiryntu.

```
public class MazeFactory {

    public Room createRoom(int number) {
        return new Room(number);
    }

    public Door createDoor(Room r1, Room r2, Direction d1, Direction d2) {
        Door door = new Door(r1, r2);
        r1.setSide(d1, door);
        r2.setSide(d2, door);
        return door;
    }

    public void createWall(Direction direction, Room room) {
        room.setSide(direction, new Wall());
    }
}
```

2.2 modyfikacja funkcji createMaze

Dokonaliśmy modyfikacji funkcji **createMaze** - teraz funkcja przyjmuje jako argument obiekt klasy **MazeFactory**.

```
public Maze createMaze(MazeFactory factory){
    Room r1 = factory.createRoom(1);
    Room r2 = factory.createRoom(2);

    factory.createWall(Direction.North, r1);
    factory.createWall(Direction.East, r1);
    factory.createWall(Direction.South, r1);
    factory.createWall(Direction.West, r1);
    factory.createWall(Direction.North, r2);
    factory.createWall(Direction.East, r2);
    factory.createWall(Direction.South, r2);
    factory.createWall(Direction.West, r2);
    Door door = factory.createDoor(r1, r2, Direction.South, Direction.North);
    Maze maze = new Maze();
    maze.addRoom(r1);
    maze.addRoom(r2);
```

```
    return maze;
}
```

2.3 klasy EnchantedMazeFactory, EnchantedRoom i EnchantedDoor

Utworzyliśmy klasę **EnchantedMazeFactory** która dziedziczy z klasy **MazeFactory**. Nowa klasa nadpisuje metody **createRoom** i **createDoor**.
Utworzyliśmy również klasy **EnchantedRoom** i **EnchantedDoor** - które dziedziczą odpowiednio z klas Room i Door. Nadpisywane metody w klasie **EnchantedMazeFactory** zwracają bardziej szczegółowe instancje elementów.

```
public class EnchantedMazeFactory extends MazeFactory{
    @Override
    public Room createRoom(int number) {
        return new EnchantedRoom(number);
    }

    @Override
    public Door createDoor(Room r1, Room r2, Direction d1, Direction d2) {
        Door door = new EnchantedDoor(r1, r2);
        r1.setSide(d1, door);
        r2.setSide(d2, door);
        return door;
    }
}
```

```
public class EnchantedDoor extends Door {
    public EnchantedDoor(Room r1, Room r2) {
        super(r1, r2);
        System.out.println("Utworzono magiczne drzwi");
    }

    @Override
    public void Enter(){
        System.out.println("Przeszedłeś przez magiczne drzwi :-).");
    }
}
```

```
public class EnchantedRoom extends Room {
    public EnchantedRoom(int number) {
        super(number);
        System.out.println("Utworzono magiczny pokój");
    }
```

```
    @Override
    public void Enter(){
        System.out.println("To magiczny pokój - uważaj!");
    }
}
```

Na potrzeby testów dodaliśmy linijki w konstruktorach **EnchantedRoom** i **EnchantedDoor**, które wypisują na wyjście programu informację o utworzeniu “magicznych” elementów.

Następnie korzystając z klasy **EnchantedMazeFactory** utworzyliśmy w funkcji main prostą instancję labiryntu:

```
public static void main(String[] args) {

    MazeGame mazeGame = new MazeGame();
    MazeFactory mazeFactory = new EnchantedMazeFactory();
    Maze maze = mazeGame.createMaze(mazeFactory);
    System.out.println(maze.getRoomNumbers());
    Vector<Room> rooms = maze.getRooms();
    for(Room room: rooms) {
        Vector<Door> doors = room.getDoors();
        System.out.println(doors.size());
    }
}
```

Otrzymany wynik jest zgodny z oczekiwanym.

```
/usr/lib/jvm/java-8-openjdk-amd64/bin/java ...
Utworzono magiczny pokój
Utworzono magiczny pokój
Utworzono magiczne drzwi
2
1
1
```

2.4 klasy BombedMazeFactory, BombedWall i BombedRoom

Kroki, jakie wykonaliśmy w tym punkcie są analogiczne do poprzedniego. Poniżej zamieszczamy implementacje trzech nowo utworzonych klas.

```
public class BombedMazeFactory extends MazeFactory{

   @Override
   public Room createRoom(int roomNumber) {
       Room room  =  new BombedRoom(roomNumber);
       return room;
   }

   @Override
   public void createWall(Direction dir, Room r1) {
       r1.setSide(dir, new BombedWall());
   }
}
```

```
public class BombedWall extends Wall {
   public BombedWall(){
       System.out.println("Utworzono bombową ścianę.");
   }

   @Override
   public void Enter(){
       System.out.println("Ostrożnie! Ta ściana może wybuchnąć.");
   }
}
```

```
public class BombedRoom extends Room {

   public BombedRoom(int roomNumber) {
       super(roomNumber);
       System.out.println("Utworzyłeś pokój pełny bomb - good luck");
   }

   @Override
   public void Enter() {
       System.out.println("Jesteś w pokoju pełnym bomb - powodzenia xd");
   }
}
```

Następnie utworzyliśmy ponownie prosty labirynt, tym razem wykorzystując **BombedMazeFactory** i wypisaliśmy “strony” utworzonych pokoi oraz ich “rodzaj”.

```
public static void main(String[] args) {

   MazeGame mazeGame = new MazeGame();
   MazeFactory builder = new BombedMazeFactory();
   Maze maze = mazeGame.createMaze(builder);

   System.out.println(maze.getRoomNumbers());
   for(Room room: maze.getRooms()){
       System.out.println(room.getRoomNumber());

       for(Direction dir: Direction.values()){
           System.out.println( dir + ": " +room.getSide(dir).getClass());
       }
   }
}
```

Otrzymany wynik jest zgodny z oczekiwaniami - zostały utworzone dwa pokoje, w których wszystkie ściany są obiektami klasy **BombedWall**. Następnie po dodaniu drzwi, jedna ze ścian w obu pokojach została zamieniona na obiekt klasy **Door**.

```
Utworzyłeś pokój pełny bomb - good luck
Utworzyłeś pokój pełny bomb - good luck
Utworzono bombową ścianę
Utworzono bombową ścianę
Utworzono bombową ścianę
Utworzono bombową ścianę
Utworzono bombową ścianę
Utworzono bombową ścianę
Utworzono bombową ścianę
Utworzono bombową ścianę
2
1
North: class pl.agh.edu.dp.labirynth.BombedWall
South: class pl.agh.edu.dp.labirynth.Door
East: class pl.agh.edu.dp.labirynth.BombedWall
West: class pl.agh.edu.dp.labirynth.BombedWall
2
North: class pl.agh.edu.dp.labirynth.Door
South: class pl.agh.edu.dp.labirynth.BombedWall
East: class pl.agh.edu.dp.labirynth.BombedWall
West: class pl.agh.edu.dp.labirynth.BombedWall
```

3. Wzorzec Singleton

Przed wykonaniem tego zadania utworzyliśmy w funkcji main dwa obiekty typu **MazeFactory**. Następnie wypisaliśmy na wyjście programu wynik porównania tych obiektów:

```
MazeFactory mz1 = new MazeFactory();
MazeFactory mz2 = new MazeFactory();
System.out.println(mz2.equals(mz1));
```

```
/usr/lib/jvm/java-8-openjdk-amd64/bin/java ...
false
```

Wynik jest zgodny z oczekiwanym - zostały utworzone dwie osobne instancje **MazeFactory**.

W klasie **MazeFactory** dodaliśmy obiekt statyczny **instance** wraz z metodą statyczną - **getInstance** która pilnuje, by atrybut **instance** nie został utworzony kilka razy.

```
private static MazeFactory instance;

public static MazeFactory getInstance(){
   if(instance == null){
       MazeFactory.instance = new MazeFactory();
   }
   return instance;
}
```

Po wprowadzeniu tej zmiany zmieniliśmy sposób tworzenia obiektu w funkcji main:

```
MazeFactory mz1 = MazeFactory.getInstance();
MazeFactory mz2 = MazeFactory.getInstance();
System.out.println(mz2.equals(mz1));
```

Tym razem porównywane obiekty są tożsame.

```
/usr/lib/jvm/java-8-openjdk-amd64/bin/java ...
true
```

4. Rozszerzenie aplikacji labirynt

a) dodanie prostego mechanizmu poruszania się i wprowadzenie rozgrywki

Aby umożliwić rozgrywkę gracza wprowadziliśmy szereg zmian w aplikacji:

- dodaliśmy atrybut **end:boolean** w klasie **Room** oznaczający ostatni pokój labiryntu(wyjście)

- dodaliśmy metody: **toString** oraz **toDirection** w enumie **Direction** - były pomocne w zrealizowaniu rozgrywki.

```
public String toString(){
   switch (this){
       case North:
           return "N";
       case South:
           return "S";
       case East:
           return "E";
       case West:
           return "W";
       default:
           return "X";
   }
}

public static Direction toDirection(String direction){
   switch (direction){
       case "N":
           return North;
       case "S":
           return South;
       case "E":
           return East;
       case "W":
           return West;
       default:
           return null;
   }
}
```

- w klasie **MazeGame** utworzyliśmy metody które tworzą labirynt korzystając z wybranych fabryk.

- Dodaliśmy klasę **Player** - która steruje rozgrywką

```java
public class Player {

   private static final ArrayList<Direction> directions =
           new ArrayList<>(Arrays.asList(
           Direction.North, Direction.East,
           Direction.South, Direction.West));

   private int healthPoints;

   public Player(){
       this.healthPoints = 10;
   }

   private void showRooms(Room room){
       System.out.println("Dostępne kierunki: ");
       for(Direction direction: directions){
           if(room.getSide(direction) instanceof Door){
               Door door = (Door) room.getSide(direction);
               System.out.println(direction.toString());
           }
       }
   }

   private Direction chooseDirection(){
       System.out.print("Wybierz kierunek: ");
       Scanner scanner = new Scanner(System.in);
       String inputString = scanner.nextLine();
       return Direction.toDirection(inputString);
   }

   private Room movePlayer(Direction moveDirection, Room room){
       Door door = (Door) room.getSide(moveDirection);
       if(door.getRoom1().equals(room)) return door.getRoom2();
       else return door.getRoom1();
   }

   public void playGame(Maze maze){
       Room currentRoom = maze.getRooms().elementAt(0);
       while(!currentRoom.end){
           if(this.healthPoints < 0){
               System.out.println("Przykro mi - umarłeś.");
               return;
           }
           System.out.println("Pozostałe punkty życia: " + this.healthPoints);
           currentRoom.Enter();
           if(currentRoom instanceof BombedRoom) {
               this.healthPoints -= 3;
```

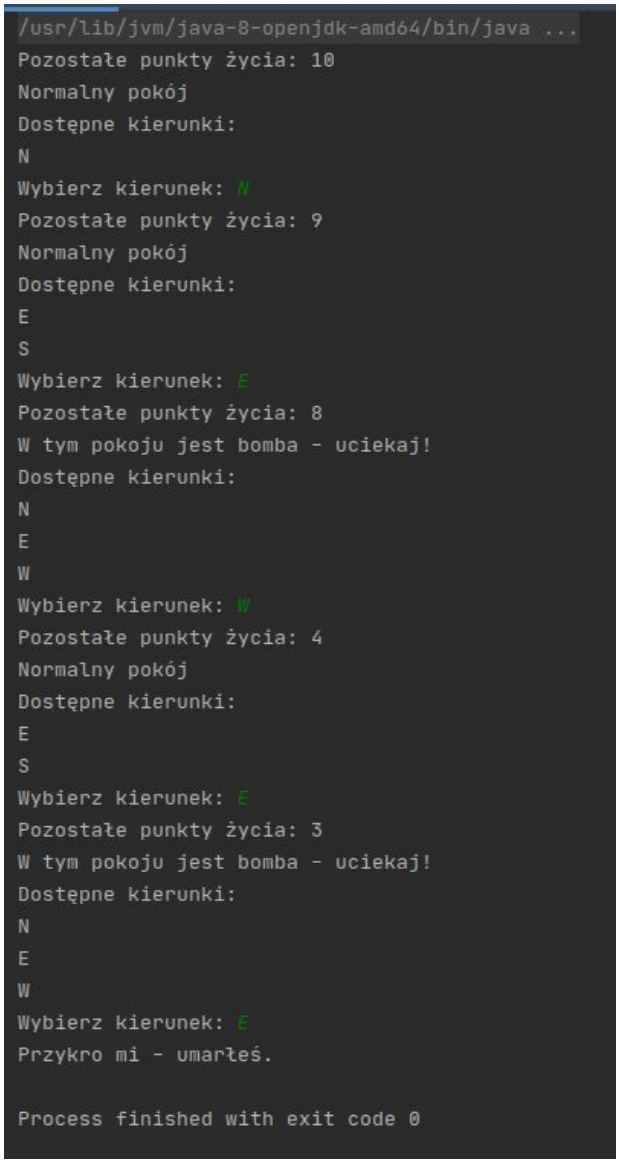
```
            }
            showRooms(currentRoom);
            Direction moveDirection = chooseDirection();
            currentRoom = movePlayer(moveDirection, currentRoom);
            this.healthPoints--;
        }
        System.out.println("Znalazłeś koniec, gratuluję :-)");
    }
}
```

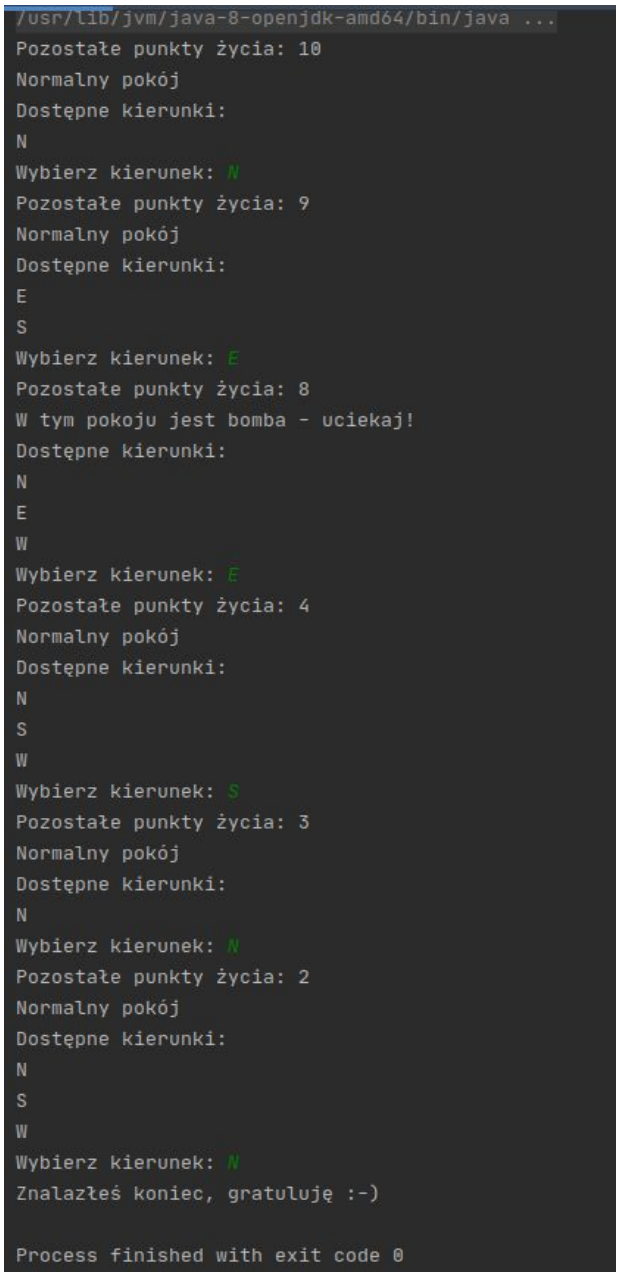
Zrealizowaliśmy rozgrywkę w lekko uproszczonej formie: w naszej aplikacji gracz widzi tylko drzwi, które są powiązane z pokojem w którym się znajduje.

Na początku rozgrywki gracz dostaje określoną ilość punktów życia. Każdy ruch kosztuje gracza 1 pkt, a wejście do pokoju typu **BombedRoom** - dodatkowe 3 pkt. Gdy liczba punktów życia gracza spadnie poniżej `0` gracz umiera. Na początku każdego ruchu gracz zostaje poinformowany o pozostałych punktach życia.
Po wejściu do pokoju gracz otrzymuje informację w jakim pokoju się znajduje (m.in **BombedRoom)** oraz w których kierunkach może się poruszać - w których ścianach znajdują się drzwi (np. N, S, W).
Gracz wpisuje kierunek, w którym chce się udać (np. N, W) i zostaje tam przeniesiony.
Jeśli gracz nie trafił do pokoju końcowego, znowu może wybrać kierunek poruszania się i rozgrywka toczy się dalej.

Po dotarciu do pokoju końcowego wyświetlana jest informacja o zakończeniu rozgrywki.

Przetestowaliśmy ręcznie aplikację, jej działanie jest zgodne z naszymi oczekiwaniami. Poniżej zamieszczamy zrzuty ekranu z przykładowych rozgrywek.





b) sprawdzenie, czy MazeFactory jest singletonem

Ten podpunkt został zrealizowany w punkcie 3 tego sprawozdania.