Projektowanie Obiektowe

Albert Gierlach

1 Builder

1) Na początku stworzyłem interfejs MazeBuilder oraz własną klasę wyjątku NoCommonWallException.

```java
public interface IMazeBuilder {
    void addRoom(Room room);
    void addDoor(Room r1, Room r2) throws NoCommonWallException;
    void addCommonWall(Room r1, Room r2, Direction commonDirRelativeToRoomOne);
}
```

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

public class NoCommonWallException extends Exception {
    private final Room r1;
    private final Room r2;

    NoCommonWallException(Room r1, Room r2){
        this.r1 = r1;
        this.r2 = r2;
    }

    public String toString(){
        return "Room " + r1.getRoomNumber() + " and " + r2.getRoomNumber() + " don't have common wall";
    }
}
```

2,3,4) Później zaimplementowałem klasę StandardMazeBuilder tak, a by wykorzystywała uprzednio zdefiniowane klasy.

```java
public class StandardMazeBuilder implements IMazeBuilder {
    private Maze m = new Maze();

    public Maze getCurrentMaze(){
        return m;
    }

    @Override
    public void addRoom(Room room) {
        for(var d : Direction.values()){
            room.setSide(d, new Wall());
        }
        m.addRoom(room);
    }

    @Override
    public void addDoor(Room r1, Room r2) throws NoCommonWallException {
        Direction res = commonWall(r1, r2);
        Door d = new Door(r1, r2);
        r1.setSide(res, d);
        r2.setSide(res.getOpposite(), d);
    }

    private Direction commonWall(Room r1, Room r2) throws NoCommonWallException {
        return EnumSet.allOf(Direction.class).stream()
                .filter(d -> r1.equals(r2.getSide(d.getOpposite())) && r2.equals(r1.getSide(d)))
                .findFirst().orElseThrow(() -> new NoCommonWallException(r1, r2));
    }

    @Override
    public void addCommonWall(Room r1, Room r2, Direction commonDirRelativeToRoomOne) {
        r1.setSide(commonDirRelativeToRoomOne, r2);
        r2.setSide(commonDirRelativeToRoomOne.getOpposite(), r1);
```

5) Metoda createMaze wygląda teraz następująco:

```java
public class MazeGame {
    public Maze createMaze(StandardMazeBuilder builder) throws NoCommonWallException{
        Room r1 = new Room(1);
        Room r2 = new Room(2);
        builder.addRoom(r1);
        builder.addRoom(r2);
        builder.addCommonWall(r1, r2, Direction.North);
        builder.addDoor(r1, r2);

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

Wynik działania programu pozostał ten sam:

```
Main
C:\portable\ideaIC-2019.2.3.win\jbr\bin\java.e
2

Process finished with exit code 0
```

6) Stowrzyłem klasę CountingMazeBuilder zgodnie z instrukcją oraz przetestowałem wynik jej działania, po uprzednich modyfikacjach metody createMaze, tak aby współdziałała z nowym builderem.

```java
public class CountingMazeBuilder implements IMazeBuilder {
    private final Map<String, Integer> counter = new HashMap<>();

    private void add(String s, int val){
        counter.putIfAbsent(s, 0);
        counter.put(s, counter.get(s) + val);
    }

    @Override
    public void addRoom(Room room) {
        add("rooms", 1);
        add("walls", 4);
    }

    @Override
    public void addDoor(Room r1, Room r2) throws NoCommonWallException {
        add("doors", 1);
        add("walls", -1);
    }

    @Override
    public void addCommonWall(Room r1, Room r2, Direction commonDirRelativeToRoomOne) {
        add("walls", -1);
    }

    public Integer getCounts(){
        return counter.values().stream().reduce(0, Integer::sum);
    }
}
```

Wynik działania zgadza się z rzeczywistością (dwa pokoje, drzwi i każdy pokój ma 3 „solidne” ściany)

```
Main
C:\portable\ideaIC-2019.2.3.win\jbr\b
9

Process finished with exit code 0
```

2 Fabryka abstrakcyjna

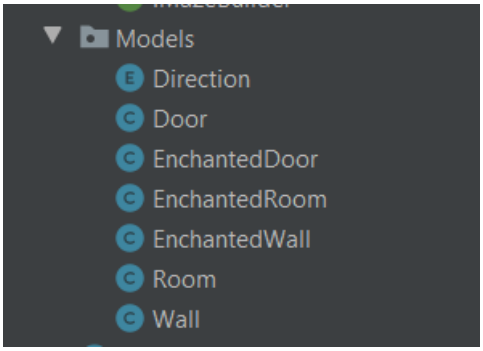
1, 2) Stworzyłem klasę MazeFactory i użyłem jej do tworzenia elementów labiryntu:

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

public class MazeFactory {
    Room createRoom(Integer id){
        return new Room(id);
    }
    Wall createWall(){
        return new Wall();
    }
    Door createDoor(Room r1, Room r2){
        return new Door(r1, r2);
    }
}
```

```java
public class MazeGame {
    public Maze createMaze(StandardMazeBuilder builder, MazeFactory factory) throws NoCommonWallException{
        Room r1 = factory.createRoom(1);
        Room r2 = factory.createRoom(2);
        builder.addRoom(r1);
        builder.addRoom(r2);
        builder.addCommonWall(r1, r2, Direction.North);
        builder.addDoor(r1, r2);

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

3) Stworzyłem klasę EnchantedMazeFactory oraz wszystkie potrzebne klasy Enchanted*, których kodu pozwolę sobie nie załączać, a jedynie pokażę, iż są one zdefiniowane.

```java
public class EnchantedMazeFactory extends MazeFactory {

    @Override
    public EnchantedRoom createRoom(Integer id) {
        return new EnchantedRoom(id);
    }

    @Override
    EnchantedWall createWall() {
        return new EnchantedWall();
    }

    @Override
    EnchantedDoor createDoor(Room r1, Room r2) {
        return new EnchantedDoor(r1, r2);
    }
}
```



4) Stworzyłem BombedMazeFactory wraz z towarzyszącymi jej modelami.

```java
public class BombedMazeFactory extends MazeFactory {
    @Override
    public BombedRoom createRoom(Integer id) {
        return new BombedRoom(id);
    }

    @Override
    public BombedWall createWall() {
        return new BombedWall();
    }
}
```

```java
public class BombedWall extends Wall {
    public BombedWall() {
        super();
    }

    @Override
    public void Enter() {
        System.out.println("Bombed wall!");
    }
}
```

```java
public class BombedRoom extends Room {
    public BombedRoom(int number) {
        super(number);
    }

    @Override
    public void Enter() {
        System.out.println("It's a trap!");
    }
}
```

3. Singleton

Klase MazeFactory przekształciłem na Singleton. Konstruktor uczyniłem chroniony, ze względu na klasy dziedziczące.

```
public class MazeFactory {
    private static final MazeFactory instance = new MazeFactory();
    protected MazeFactory() {};

    public static MazeFactory getInstance(){
        return instance;
    }

    public Room createRoom(Integer id){
        return new Room(id);
    }
    Wall createWall(){
        return new Wall();
    }
    Door createDoor(Room r1, Room r2){
        return new Door(r1, r2);
    }
}
```

4 Rozszerzenie aplikacji labirynt

a) Do projektu dodałem klasę gracza.

```java
public class Player {
    private Room currentRoom;
    private Boolean alive = Boolean.TRUE;

    public Player(){
    }

    public void kill(){
        alive = Boolean.FALSE;
    }

    public boolean isAlive(){
        return alive;
    }

    public Room getCurrentRoom() {
        return currentRoom;
    }

    public void setCurrentRoom(Room r) {
        this.currentRoom = r;
    }
}
```

Następnie zmodyfikowałem klasę MapSite tak, aby metoda Enter przyjmowała jako parametr instancję gracza.

```java
public abstract class MapSite {
    public abstract void Enter(Player p);
}
```

Wszystkie typy ścian, drzwi oraz pokoi implementują tą metodę w zależności od potrzeby. Przykładowe implementacje poniżej:

```java
public class BombedWall extends Wall {
    public BombedWall() {
        super();
    }

    @Override
    public void Enter(Player p) {
        System.out.println("Bombed wall!");
        p.kill();
    }
}
```

```java
public class EnchantedRoom extends Room {
    public EnchantedRoom(int number) {
        super(number);
    }

    @Override
    public void Enter(Player p) {
        System.out.println("You are in enchanted room!");
    }
}
```

```java
public class BombedRoom extends Room {
    public BombedRoom(int number) {
        super(number);
    }

    @Override
    public void Enter(Player p) {
        System.out.println("It's a trap!");
        p.kill();
    }
}
```

```java
public class Wall extends MapSite {
    public Wall(){

    }

    @Override
    public void Enter(Player p){
        System.out.println("You can't move through wall.");
    }
}
```

```java
public class Door extends MapSite {
    private final Room room1;
    private final Room room2;

    public Door(Room r1, Room r2){
        this.room1 = r1;
        this.room2 = r2;
    }

    @Override
    public void Enter(Player p){
        System.out.println("You went through the door.");
        Room otherSideRoom = getOtherSideRoom(p.getCurrentRoom());
        p.setCurrentRoom(otherSideRoom);
        otherSideRoom.Enter(p);
    }

    public Room getOtherSideRoom(Room r){
        if(r == room1){
            return room2;
        }else if(r == room2){
            return room1;
        }
        return null;
    }
}
```

Dodałem także również klasę wyliczeniową stanowiącą o statusie aktualnej gry:

```java
public enum GameState{
    NONE, WIN, LOSE, PENDING;

    public String toString(){
        switch (this){
            case NONE: return "Game not started";
            case WIN: return "You won :)";
            case LOSE: return "You lost :(";
            case PENDING: return "Game in progress";
        }
        return "???";
    }
}
```

Inicjalizacja gry także doczekała się refactoringu i teraz wygląda następująco:

```java
public class Main {

    public static void main(String[] args) {
        MazeGame mazeGame = new MazeGame(new Player());
        try{
            mazeGame.createMaze(new StandardMazeBuilder(),
                    MazeFactory.getInstance(),
                    BombedMazeFactory.getInstance(),
                    EnchantedMazeFactory.getInstance());
        }catch (NoCommonWallException e){
            System.out.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
```

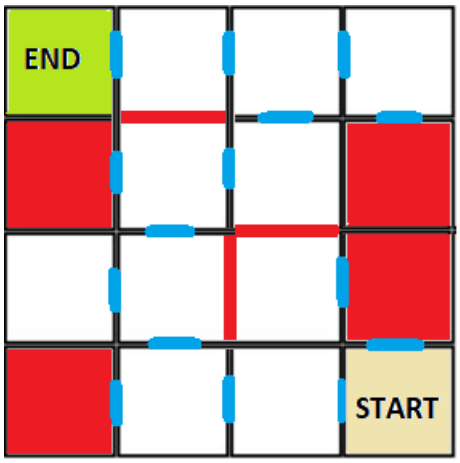
Do klasy MazeGame dodałem metodę, która odpowiada za przemieszczanie gracza w zależności od żądanego kierunku. Metoda ta wywołuje metodę Enter() z klas implementujących klase/interfejs MapSite, dzięki czemu uniknąłem ciągłego sprawdzania i rzutowania obiektów na żądany typ, a także znacznie uprościł się sposób obslugi elementów mapy. Cała klasa wygląda jak na załączonym zrzucie:

```java
public class MazeGame {
    private final Player player;
    private Maze maze;

    GameState gameState = GameState.NONE;

    public MazeGame(Player p){
        this.player = p;
    }

    public Player getPlayer(){
        return player;
    }

    public void movePlayer(Direction dir){
        Room currentRoom = player.getCurrentRoom();
        MapSite toMoveSite = currentRoom.getSide(dir);

        toMoveSite.Enter(player);

        if(!player.isAlive()){
            gameState = GameState.LOSE;
        }else if(player.getCurrentRoom().equals(maze.getEndRoom())){
            gameState = GameState.WIN;
        }else{
            gameState = GameState.PENDING;
        }
    }

    public GameState getState() {
        return gameState;
    }
```

Celowo pominąłem metodę createMaze, gdyż jak się okaże jest ona dość obszerna. Dalej zimplementowałem obsługę klawiszy oraz parsowanie danych wejściowych. Kod ten został umiejscowiony w głównej metodzie zaraz po inicjalizacji gry.

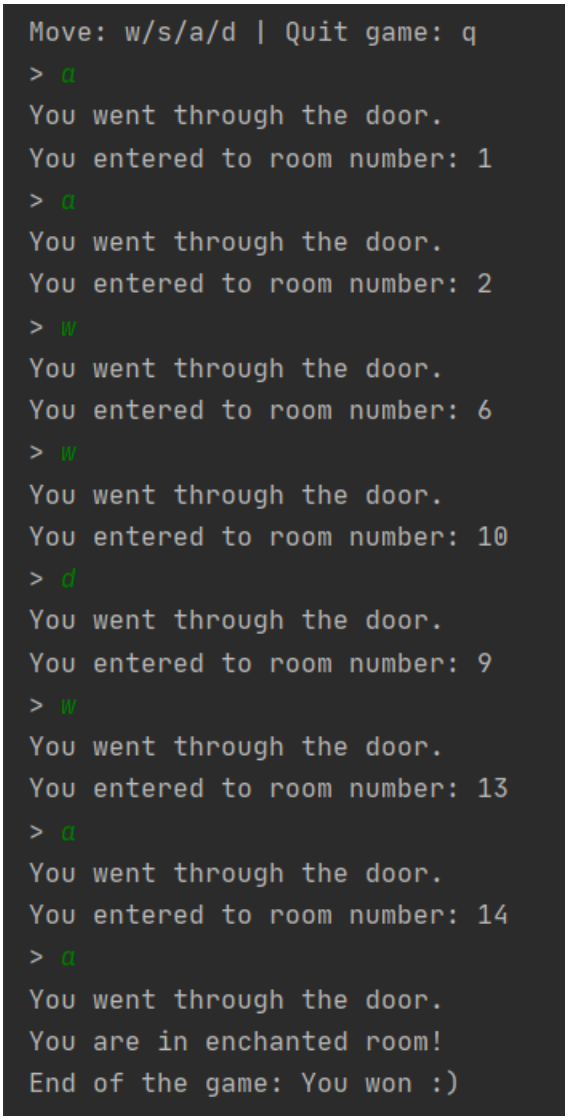
```
        Map<Character, Direction> moves = new HashMap<>();
        moves.put('a', Direction.West);
        moves.put('d', Direction.East);
        moves.put('w', Direction.North);
        moves.put('s', Direction.South);

        System.out.println("Move: w/s/a/d | Quit game: q");
        System.out.print("> ");
        String s;
        Scanner in = new Scanner(System.in);
        while(!(s = in.nextLine()).equals("q")){
            if(s.length() == 0){
                continue;
            }
            Character c = s.charAt(0);
            if(!moves.containsKey(c)){
                System.out.println("Wrong option!");
                System.out.println("Move: w/s/a/d | Quit game: q");
            }else{
                mazeGame.movePlayer(moves.get(c));
                GameState state = mazeGame.getState();
                if(state != GameState.PENDING && state != GameState.NONE){
                    break;
                }
            }
            System.out.print("> ");
        }

        System.out.println("End of the game: " + mazeGame.getState().toString());
    }
```

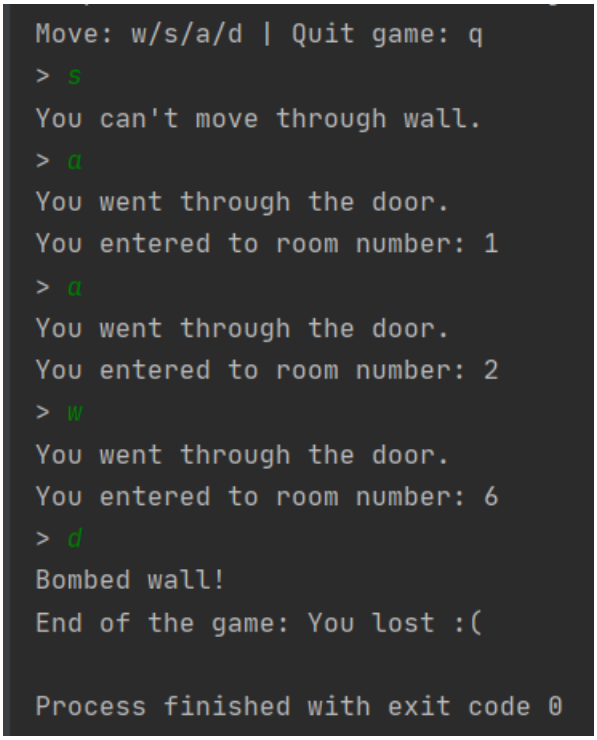
Po implementacji czas na testy. W profesjonalnym edytorze graficznym stworzyłem prosty szkic testowej mapy, którą później zaimplementowałem w kodzie programu. Mapa prezentuje się następująco:

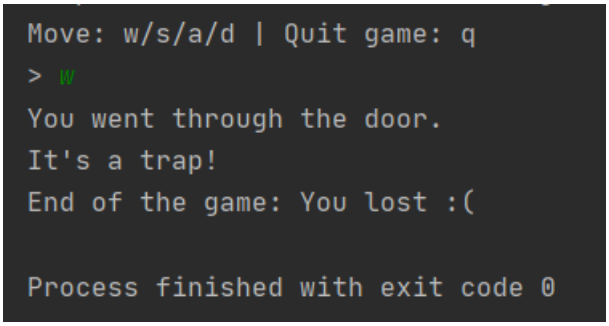


Czerwony pokój – BombedRoom, Niebieska kreska – drzwi, Czarna kreska – Wall, Biały pokój – Room, Zielony pokój – Enchanted Room (nasz cel gry), START – pokój startowy

Przykładowe uruchomienia gry:







Jak widać implementacja działa poprawnie.

b) Sprawdzenie czy MazeFactory jest rzeczywiście Singletonem. W tym celu napisałem prosty test używając biblioteki JUnit4

```java
import org.junit.Test;
import pl.agh.edu.dp.labirynth.Factories.MazeFactory;

import static org.junit.Assert.assertEquals;

public class MazeFactoryTester {
    @Test
    public void MazeFactoryTest(){
        MazeFactory m = MazeFactory.getInstance();
        MazeFactory m2 = MazeFactory.getInstance();
        assertEquals(m, m2);
        assertEquals(MazeFactory.getInstance(), m);
        assertEquals(MazeFactory.getInstance(), m2);
    }
}
```