

Szymon Rogus – Raport z Patterns Design

4.1 Builder:

Efekt po wykonaniu 6 podpunktów:

Klasa Maze:

```
public class Maze {  
  
    private Vector<Room> rooms;  
    private Vector<Door> doors;  
  
    public Maze() {  
        this.rooms = new Vector<Room>();  
        this.doors = new Vector<Door>();  
    }  
  
    public void addRoom(Room room) {  
        rooms.add(room);  
    }  
  
    public void setRooms(Vector<Room> rooms) {  
        this.rooms = rooms;  
    }  
  
    public int getRoomNumbers() {  
        return rooms.size();  
    }  
}
```

Dodałem kolejkę w której umieszczam drzwi dodane do labiryntu

Interfejs MazeBuilder:

```
public interface MazeBuilder {  
  
    Vector<Room> rooms = new Vector<Room>();  
    Vector<Door> doors = new Vector<Door>();  
  
    public Maze build();  
}
```

Tylko metoda build jest niezbędna w builderze.

Klasa StandardBuilderMaze:

- ta klasa implementuje interfejs MazeBuilder plus dodaje metody niezbędne do tworzenia i dodawania komponentów do labiryntu

Są tutaj takie metody jak:

```
public StandardBuilderMaze addRoom( Room room) {  
    if(!checkForRoom(room.getRoomNumber())){  
        throw new IllegalStateException  
            ("Room with " + room.getRoomNumber() + " already in Maze");  
    }  
    rooms.add(room);  
    return this;  
}
```

```
public StandardBuilderMaze createRoomWithWalls(int roomId) {  
    if(!checkForRoom(roomId)){  
        throw new IllegalStateException  
            ("Room with " + roomId + " already in Maze");  
    }  
    Room room = new Room(roomId);  
    setWalls(room);  
    rooms.add(room);  
    return this;  
}
```

(jest ich więcej – odsyłam do kodu)

Klasa CountingMazeBuilder:

```
public class CountingMazeBuilder implements MazeBuilder {  
    public int getCounts() {  
        int counter = rooms.size() + doors.size();  
        return counter;  
    }  
    public Maze build() {  
        int counter = getCounts();  
        Maze maze = new Maze();  
        return maze;  
    }  
}
```

A tak wygląda tworzenie labiryntu za pomocą zaimplementowanego buildera:

```
public Maze createMaze(Maze.StandardBuilderMaze mazeBuilder) {  
    Maze maze = mazeBuilder  
        .addRoom(new Room(1))  
        .createRoomWithWalls(2)  
        .addRoom(new Room(3))  
        .build();  
    return maze;  
}
```

Generalnie zarówno interfejs Maze jak i StandardBuilder który go implementuje zawiera te same atrybuty co klasa Maze. Każda metoda buildera zwraca instancję siebie, oprócz metody build, w której tworzymy instancję klasy którą budujemy (w tym przypadku Maze)

```
public Maze build() {  
    if(rooms.isEmpty()) {  
        throw new IllegalStateException("Cannot create empty Maze");  
    }  
  
    Maze maze = new Maze();  
    maze.rooms = this.rooms;  
    maze.doors = this.doors;  
  
    return maze;  
}
```

4.2 Fabryka abstrakcyjna:

Klasa MazeFactory:

```
public class MazeFactory {  
    public Maze createMaze() {  
        return new Maze();  
    }  
    public Room createRoom(int roomId) {  
        return new Room(roomId);  
    }  
    public Wall createWall() {  
        return new Wall();  
    }  
    public Door createDoor(Room room1, Room room2) {  
        return new Door(room1, room2);  
    }  
}
```

MazeFactory dostarcza interfejs do tworzenia komponentów do labiryntów, włącznie z samym labiryntem (udostępniając nowe instancje obiektów).

Klasa EnchantedMazeFactory:

```
public class EnchantedMazeFactory extends MazeFactory{  
    @Override  
    public Room createRoom(int roomId) {  
        return new Room.SmallRoom(roomId);  
    }  
    @Override  
    public Wall createWall() {  
        return new Wall.CommonWall();  
    }  
    @Override  
    public Door createDoor(Room room1, Room room2) {  
        return new Door.SpecialDoor(room1, room2);  
    }  
}
```

Metody w tej fabryce zwracają podklasy konkretnych klas (wszystkie metody są nadpisywane a nie przesłaniane – nie tworzyłem metod statycznych)

```
public Maze createMaze(MazeFactory mazeFactory){  
    Maze maze = mazeFactory.createMaze();  
    return maze;  
}
```

4.3 Singleton:

Modyfikacja MazeFactory do Singletona:

```
public class MazeFactory {  
    public static final MazeFactory SINGLETON = new MazeFactory();  
  
    public static MazeFactory getInstance() {  
        return SINGLETON;  
    }  
  
    ...  
}
```

Dzięki temu dowolny labirynt jaki stworzymy, będzie miał tą samą wspólną instancję:

```
MazeGame mazeGame = new MazeGame();  
  
Maze maze = mazeGame.createMaze(MazeFactory.getInstance());  
  
System.out.println(maze.getRoomNumbers());  
}
```

Dostęp do stworzonego w ten sposób obiektu będzie globalny.

4.4 Rozszerzenie:

Klasa player pozwala na poruszanie się po labiryncie za pomocą prostych komend:

```
public class Player {  
    private Room actualRoom;  
  
    public Player(Room actualRoom) {  
        this.actualRoom = actualRoom;  
    }  
  
    public void move(Direction direction) {  
        Object mapSite = actualRoom.getSide(direction);  
    }  
}
```

```

        if(mapSite != null && !(mapSite instanceof Wall)){
            Door door = (Door) mapSite;
            if(actualRoom.getRoomNumber() != door.getRoom1().getRoomNumber()){
                actualRoom = door.getRoom1();
                info();
            }
            else
            if(actualRoom.getRoomNumber() != door.getRoom2().getRoomNumber()){
                actualRoom = door.getRoom2();
                info();
            }
        }
        else
            System.err.println("Cannot cros the Wall!");
    }
    private void info() {
        System.out.println
            ("Moved to room of id: " + actualRoom.getRoomNumber());
    }
}

```

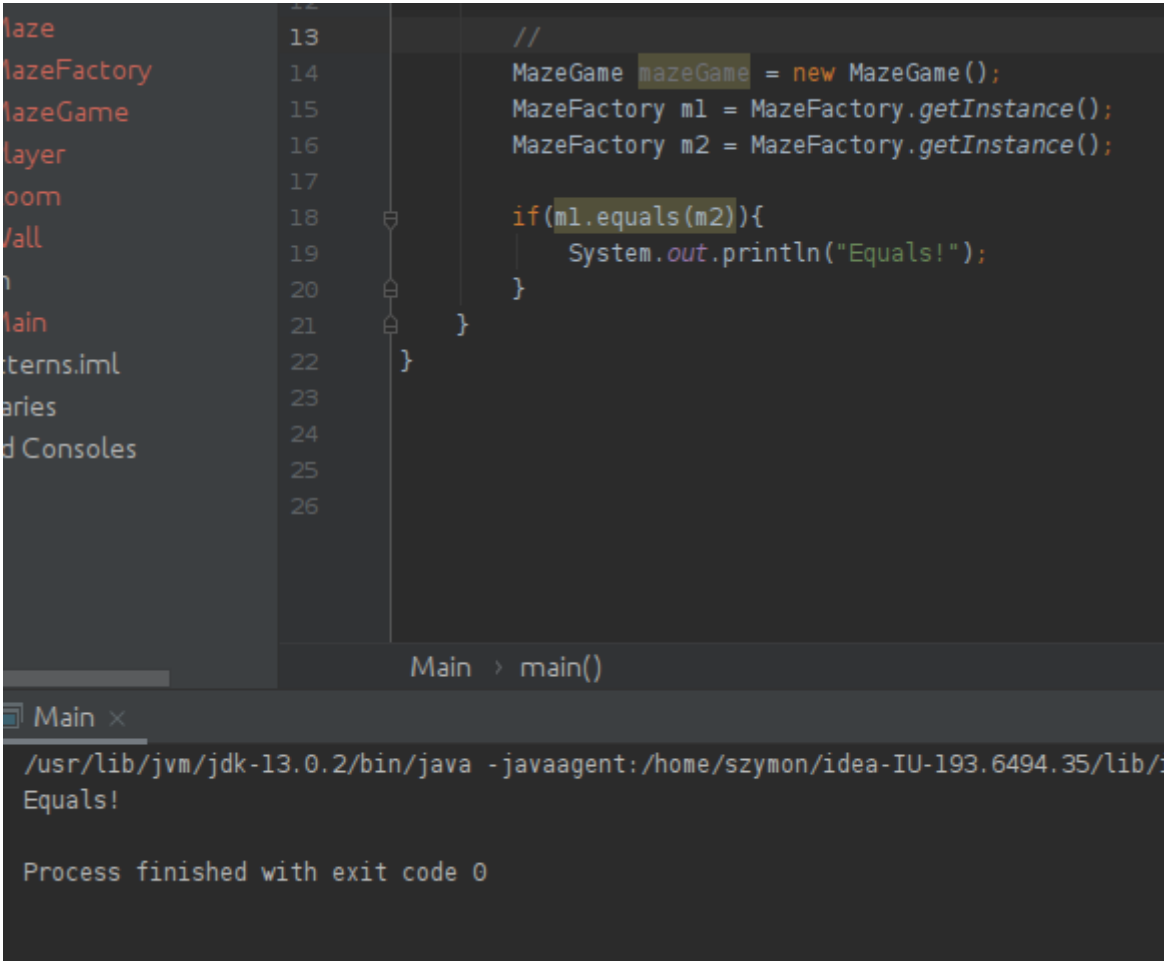
Player przechowuje tylko aktualny pokój w którym się znajduje. Za pomocą kierunku który przekazujemy do metody, wykrywamy czy mamy przed sobą obiekt typu Wall czy Door. W zależności od tego następuje ruch lub komunikat błędu.

```

0
[North - N, South - S, East - E, West - W]
N
Moved to room of id: 3
E
Moved to room of id: 4
N
Cannot cros the Wall!
S
Moved to room of id: 6
E
Cannot cros the Wall!
W
Moved to room of id: 2

```

Na koniec sprawdzam czy MazFactory jest Singletonem:



The screenshot shows an IDE with a Java file named `Main`. The code is as follows:

```
12
13
14 //
15 MazeGame mazeGame = new MazeGame();
16 MazeFactory m1 = MazeFactory.getInstance();
17 MazeFactory m2 = MazeFactory.getInstance();
18
19 if(m1.equals(m2)){
20     System.out.println("Equals!");
21 }
22
23
24
25
26
```

The IDE's breadcrumb at the bottom indicates the current location is `Main > main()`. Below the code editor, the `Main` tab is active, showing the execution output:

```
/usr/lib/jvm/jdk-13.0.2/bin/java -javaagent:/home/szymon/idea-IU-193.6494.35/lib/
Equals!

Process finished with exit code 0
```