Wzorce projektowe cz. 1

1. Builder

1. Stworzono klasę MazeBuilder, która definiuje interfejs służący do tworzenia labiryntów.

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

public abstract class MazeBuilder {
    abstract void addRoom(Room room);
    abstract void addDoor(Room r1, Room r2) throws Exception;
}
```

2. Po utworzeniu powyższego interfejsu zmodyfikowano funkcję składową tak, aby przyjmowała jako parametr obiekt tej klasy.

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

public class MazeGame {
    public void createMaze(MazeBuilder builder) {
        Room r1 = new Room(1);
        Room r2 = new Room(2);

        Door door = new Door(r1, r2);

        builder.addRoom(r1);
        builder.addRoom(r2);
        builder.addDoor(r1, r2);

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

        r2.setSide(Direction.North, new Wall());
        r2.setSide(Direction.East, new Wall());
        r2.setSide(Direction.South, new Wall());
        r2.setSide(Direction.West, new Wall());
    }
}
```

3. Poniżej zinterpretowano obecne zmiany i krótko opisano swoje spostrzeżenia.

Wyeliminowano szczegóły implementacyjne tworzenia labiryntu. Tę funkcjonalność przeniesiono do klas implementujących interfejs MazeBuilder.

4. Stworzono klasę StandardBuilderMaze będącą implementacją MazeBuildera oraz dodano tam dodatkowo metodę prywatną CommonWall, która określa kierunek standardowej ściany pomiędzy dwoma pomieszczeniami.

- Zmodyfikowano typ wyliczeniowy Enum:

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

public enum Direction {
    North, South, East, West;

    public Direction getOppositeDirection() {
        switch (this) {
            case North:
                return South;
            case South:
                return North;
            case East:
                return West;
            case West:
                return East;
            default:
                throw new IllegalArgumentException("Nieprawidłowy
kierunek");
        }
    }
}
```

Dodana metoda `getOppositeDirection()` będzie pomocna przy dodawaniu drzwi pomiędzy dwoma wybranymi pokojami.

- Dodano do klasy abstrakcyjnej nowe metody:

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

public abstract class MazeBuilder {
    abstract void addRoom(Room room);
    abstract void addDoor(Room r1, Room r2) throws Exception;
    abstract void commonWall(Direction firstRoomDir, Room r1, Room r2);
    abstract Maze build();
}
```

- Usunięto z metody `createMaze(MazeBuilder builder)` fragmenty odpowiedzialne za tworzenie pokoju i przeniesiono je do Buildera:

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

public class MazeGame {
    public Maze createMaze(MazeBuilder builder) throws Exception {
        Room r1 = new Room(1);
        Room r2 = new Room(2);

        builder.addRoom(r1);
        builder.addRoom(r2);
        builder.addDoor(r1, r2);

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

- Z klasy Maze usunięto metodę `setRooms(Vector<Room> rooms)`:

```
public void setRooms(Vector<Room> rooms) {
    this.rooms = rooms;
}
```

- Cały Builder wygląda teraz następująco:

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


public class StandardBuilderMaze extends MazeBuilder {

    private final Maze currentMaze;

    public StandardBuilderMaze() {
        this.currentMaze = new Maze();
    }

    @Override
    public void addRoom(Room room) {
        room.setSide(Direction.North, new Wall());
        room.setSide(Direction.East, new Wall());
        room.setSide(Direction.South, new Wall());
        room.setSide(Direction.West, new Wall());
        this.currentMaze.addRoom(room);
```

```
    }

    @Override
    public void addDoor(Room r1, Room r2) throws Exception {
        if (checkIfRoomsHaveADoor(r1, r2)) {
            throw new Exception("These rooms already have a common door!");
        }
        this.commonWall(r1, r2);
        Direction firstRoomCommonWallDirection = null;
        for (Direction direction: Direction.values()) {
            MapSite r1CommonWallDirection = r1.getSide(direction);
            MapSite r2CommonWallDirection =
r2.getSide(direction.getOppositeDirection());
            if (r1CommonWallDirection.equals(r2CommonWallDirection)) {
                firstRoomCommonWallDirection = direction;
                break;
            }
        }
        if (firstRoomCommonWallDirection == null) throw new
Exception("These rooms don't have a common wall!");
        else {
            Door door = new Door(r1, r2);
            r1.setSide(firstRoomCommonWallDirection, door);
            r2.setSide(firstRoomCommonWallDirection.getOppositeDirection(),
door);
        }
    }

    @Override
    public void commonWall(Room r1, Room r2) {
        Wall wall = new Wall();
        Direction direction = findDirectionForCommonWall(r1, r2);
        r1.setSide(direction, wall);
        r2.setSide(direction.getOppositeDirection(), wall);
    }

    public Maze getCurrentMaze() {
        return this.currentMaze;
    }

    private Direction findDirectionForCommonWall(Room r1, Room r2) {
        Direction[] directionList = Direction.values();
        for (Direction currentDirection : directionList) {
            Direction oppositeDirection =
currentDirection.getOppositeDirection();
            if (r1.getSide(currentDirection).getClass() ==
```

```
r2.getSide(oppositeDirection).getClass()) {
                return currentDirection;
            }
        }
        throw new IllegalArgumentException("Cannot make door between Room1
(id: " + r1.getRoomNumber() +
                ") i Room2 (id: " + r2.getRoomNumber() + ")");
    }

    private boolean checkIfRoomsHaveADoor(Room r1, Room r2) {
        Direction[] directions = Direction.values();
        for (Direction direction: directions) {
            if ((r1.getSide(direction).getClass() == Door.class) &&

(r2.getSide(direction.getOppositeDirection()).getClass() == Door.class)) {
                return true;
            }
        }
        return false;
    }
}
```

W szczególności warto zwrócić uwagę na metody dodające nowe obiekty do labiryntu (drzwi oraz pokoje), a także tę, która dodaje drzwi pomiędzy dwoma wybranymi pokojami.

5. Utworzono labirynt przy pomocy operacji createMaze, gdzie parametrem był obiekt klasy StandardMazeBuilder.

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

import pl.agh.edu.dp.labirynth.*;

public class Main {

    public static void main(String[] args) throws Exception {
        MazeGame mazeGame = new MazeGame();
        StandardBuilderMaze standardBuilderMaze = new
StandardBuilderMaze();
        Maze maze = mazeGame.createMaze(standardBuilderMaze);
        System.out.println(maze.getRoomNumbers());
    }
}
```

6. Stworzono kolejną podklasę MazeBuildera o nazwie CountingMazeBuilder. Budowniczy tego obiektu w ogóle nie tworzy labiryntu, a jedynie zlicza utworzone

komponenty różnych rodzajów. Powinien mieć metodę GetCounts, która zwraca ilość elementów.

Jako, iż kod w paru miejscach uległ modyfikacjom, zaktualizowane fragmenty programu zamieszczam poniżej.

- Kod klasy MazeBuilder:

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

public abstract class MazeBuilder {
    abstract void addRoom(Room room);
    abstract void addDoor(Room r1, Room r2) throws Exception;
    abstract void commonWall(Room r1, Room r2);
}
```

Usunięto metodę abstrakcyjną `build()` oraz parametr kierunku w metodzie `commonWall(Room r1, Room r2)`. Od tej pory logika wybierania odpowiedniej ściany, aby połączyć dwa pokoje, będzie zaszyta w Builderze.

- Kod klasy StandardMazeBuilder:

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


public class StandardBuilderMaze extends MazeBuilder {

    private final Maze currentMaze;

    public StandardBuilderMaze() {
        this.currentMaze = new Maze();
    }

    @Override
    public void addRoom(Room room) {
        room.setSide(Direction.North, new Wall());
        room.setSide(Direction.East, new Wall());
        room.setSide(Direction.South, new Wall());
        room.setSide(Direction.West, new Wall());
        this.currentMaze.addRoom(room);
    }

    @Override
    public void addDoor(Room r1, Room r2) throws Exception {
        this.commonWall(r1, r2);
```

```
        Direction firstRoomCommonWallDirection = null;
        for (Direction direction: Direction.values()) {
            MapSite r1CommonWallDirection = r1.getSide(direction);
            MapSite r2CommonWallDirection =
r2.getSide(direction.getOppositeDirection());
            if (r1CommonWallDirection.equals(r2CommonWallDirection)) {
                firstRoomCommonWallDirection = direction;
                break;
            }
        }
        if (firstRoomCommonWallDirection == null) throw new
Exception("These rooms don't have a common wall!");
        else {
            Door door = new Door(r1, r2);
            r1.setSide(firstRoomCommonWallDirection, door);
            r2.setSide(firstRoomCommonWallDirection.getOppositeDirection(),
door);
        }
    }

    @Override
    public void commonWall(Room r1, Room r2) {
        Wall wall = new Wall();
        Direction direction = findDirectionForCommonWall(r1, r2);
        r1.setSide(direction, wall);
        r2.setSide(direction.getOppositeDirection(), wall);
    }

    public Maze getCurrentMaze() {
        return this.currentMaze;
    }

    private Direction findDirectionForCommonWall(Room r1, Room r2) {
        Direction[] directionList = Direction.values();
        for (Direction currentDirection : directionList) {
            Direction oppositeDirection =
currentDirection.getOppositeDirection();
            if (r1.getSide(currentDirection).getClass() ==
r2.getSide(oppositeDirection).getClass()) {
                return currentDirection;
            }
        }
        throw new IllegalArgumentException("Cannot make door between Room1
(id: " + r1.getRoomNumber() +
                ") i Room2 (id: " + r2.getRoomNumber() + ")");
    }
```

```
}
```

Można zauważyć wspomnianą wyżej prywatną metodę `findDirectionForCommonWall(Room r1, Room r2)`, która jest odpowiedzialna za tworzenie wspólnej ściany dla dwóch wybranych pokoi, o ile jest to oczywiście możliwe.

- Kod klasy MazeGame:

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

public class MazeGame {
    public void createMaze(MazeBuilder builder) throws Exception {
        Room r1 = new Room(1);
        Room r2 = new Room(2);

        builder.addRoom(r1);
        builder.addRoom(r2);
        builder.addDoor(r1, r2);
    }
}
```

Zmieniono typ zwracanej wartości metody `createMaze(MazeBuilder builder)` z powrotem na void.

- Kod klasy Main:

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

import pl.agh.edu.dp.labirynth.*;

public class Main {

    public static void main(String[] args) throws Exception {
        MazeGame mazeGame = new MazeGame();
        StandardBuilderMaze standardBuilderMaze = new
StandardBuilderMaze();
        CountingMazeBuilder countingMazeBuilder = new
CountingMazeBuilder();

        mazeGame.createMaze(standardBuilderMaze);

System.out.println(standardBuilderMaze.getCurrentMaze().getRoomNumbers());

        mazeGame.createMaze(countingMazeBuilder);
        System.out.println(countingMazeBuilder.getCounts());
```

```
    }
}
```

oraz wynik działania programu na standardowym wyjściu:

```
Liczba pokojów w labiryncie: 2
Liczba elementów w labiryncie: 9
```

Dwa pokoje zostały stworzone przez Buildera, a więc mieliśmy na początku dziesięć komponentów różnych rodzajów (osiem ścian i dwa pokoje). Potem pokoje zostały połączone a więc liczba ścian zmniejszyła się o jeden, a następnie ta ściana została zamieniona na drzwi. Mamy więc dziewięć komponentów (dwa pokoje, jedne drzwi oraz sześć ścian).

2. Fabryka abstrakcyjna

1. Stworzono klasę MazeFactory, która służy do tworzenia elementów labiryntu. Można jej użyć w programie, który np. wczytuje labirynt z pliku .txt, czy generuje labirynt w sposób losowy.

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

import pl.agh.edu.dp.labirynth.Door;
import pl.agh.edu.dp.labirynth.Room;
import pl.agh.edu.dp.labirynth.Wall;

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

2. Przeprowadzono kolejną modyfikacje funkcji createMaze tak, aby jako parametr brała MazeFactory.

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

public class MazeGame {
```

```
    public void createMaze(MazeFactory mazeFactory) throws Exception {
        Room r1 = new Room(1);
        Room r2 = new Room(2);

        /*builder.addRoom(r1);
        builder.addRoom(r2);
        builder.addDoor(r1, r2);*/
    }
}
```

3. Stworzono klasę EnchantedMazeFactory (fabryka magicznych labiryntów), która dziedziczy z MazeFactory. Przesłania kilka funkcji składowych i zwraca różne podklasy klas Room, Wall itd. (takie klasy również stworzono).

- Kod klasy EnchantedMazeFactory:

```
public class EnchantedMazeFactory extends MazeFactory {

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

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

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

- Kod klasy EnchantedRoom:

```
package pl.agh.edu.dp.products.enchanted;

import pl.agh.edu.dp.labirynth.Room;

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

- Kod klasy EnchantedDoor:

```
package pl.agh.edu.dp.products.enchanted;

import pl.agh.edu.dp.labirynth.Door;
import pl.agh.edu.dp.labirynth.Room;

public class EnchantedDoor extends Door {
    public EnchantedDoor(Room r1, Room r2) {
        super(r1, r2);
    }
}
```

- Kod klasy EnchantedWall:

```
package pl.agh.edu.dp.products.enchanted;

import pl.agh.edu.dp.labirynth.Wall;

public class EnchantedWall extends Wall {
    public EnchantedWall() {
        super();
    }
}
```

- Kod klasy MazeGame:

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

import pl.agh.edu.dp.builders.StandardBuilderMaze;
import pl.agh.edu.dp.factories.MazeFactory;

public class MazeGame {
    public void createMaze(StandardBuilderMaze mazeBuilder, MazeFactory
mazeFactory) throws Exception {
        Room r1 = mazeFactory.createRoom(1);
        Room r2 = mazeFactory.createRoom(2);
        Room r3 = mazeFactory.createRoom(3);

        mazeBuilder.addRoom(r1);
        mazeBuilder.addRoom(r2);
        mazeBuilder.addRoom(r3);
        mazeBuilder.addDoor(r1, r2);
        mazeBuilder.addDoor(r2, r3);
```

```
    }
}
```

Jak łatwo można zauważyć użyto tutaj dwóch wzorców projektowych: fabryki abstrakcyjnej do tworzenia zestawu powiązanych ze sobą obiektów oraz budowniczego do utworzenia labiryntu poprzez zestawienie poprzednio utworzonych obiektów.

- Na sam koniec prezentuję kod klasy Main:

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

import pl.agh.edu.dp.builders.StandardBuilderMaze;
import pl.agh.edu.dp.factories.EnchantedMazeFactory;
import pl.agh.edu.dp.factories.MazeFactory;
import pl.agh.edu.dp.labirynth.*;

public class Main {

    public static void main(String[] args) throws Exception {
        MazeGame mazeGame = new MazeGame();
        MazeFactory mazeFactory = new EnchantedMazeFactory();
        StandardBuilderMaze standardBuilderMaze = new
StandardBuilderMaze();
        mazeGame.createMaze(standardBuilderMaze, mazeFactory);

System.out.println(standardBuilderMaze.getCurrentMaze().getRoomNumbers());
    }
}
```

- oraz wynik działania programu (w metodach odpowiedzialnych za budowanie labiryntu: dodawanie pokojów oraz drzwi, dopisano linijkę, która wyświetli wyniki na standardowym wyjściu):

```
Dodano pokój nr 1
Dodano pokój nr 2
Dodano pokój nr 3
Dodano drzwi pomiędzy pokojem 1 a 2
Dodano drzwi pomiędzy pokojem 2 a 3
Liczba pokojów w labiryncie: 3
```

4. Stworzono klasę BombedMazeFactory, która zapewnia, że ściany to obiekty klasy BombedWall, a pomieszczenia to obiekty klasy BombedRoom (teoretycznie wystarczy przesłonić jedynie 2 metody - MakeWall(...) / MakeRoom(...)).

- Zmieniono zakres widoczności metod abstrakcyjnych na publiczny (szerszy niż pakietowy):

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

import pl.agh.edu.dp.labirynth.Room;

public abstract class MazeBuilder {
    public abstract void addRoom(Room room);
    public abstract void addDoor(Room r1, Room r2) throws Exception;
    abstract void commonWall(Room r1, Room r2);
}
```

- Zmieniono typ parametru metody, która tworzy labirynt:

```
public void createMaze(MazeBuilder mazeBuilder, MazeFactory mazeFactory)
```

- Kod klasy BombedMazeFactory:

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

import pl.agh.edu.dp.labirynth.Door;
import pl.agh.edu.dp.labirynth.Maze;
import pl.agh.edu.dp.labirynth.Room;
import pl.agh.edu.dp.labirynth.Wall;
import pl.agh.edu.dp.products.bombed.BombedRoom;
import pl.agh.edu.dp.products.bombed.BombedWall;

public class BombedMazeFactory extends MazeFactory {
    @Override
    public Room createRoom(int number) {
        return new BombedRoom(number);
    }

    @Override
    public Door createDoor(Room r1, Room r2) {
        return null;
    }

    @Override
    public Wall createWall() {
```

```
        return new BombedWall();
    }
}
```

- Kod klasy BombedRoom:

```
package pl.agh.edu.dp.products.bombed;

import pl.agh.edu.dp.labirynth.Room;

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

- Kod klasy BombedWall:

```
package pl.agh.edu.dp.products.bombed;

import pl.agh.edu.dp.labirynth.Wall;

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

- Do klasy Main dodano jeszcze dodatkowo budowniczego, który zlicza ilość komponentów w labiryncie:

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

import pl.agh.edu.dp.builders.CountingMazeBuilder;
import pl.agh.edu.dp.builders.MazeBuilder;
import pl.agh.edu.dp.builders.StandardBuilderMaze;
import pl.agh.edu.dp.factories.BombedMazeFactory;
import pl.agh.edu.dp.factories.MazeFactory;
import pl.agh.edu.dp.labirynth.*;

public class Main {

    public static void main(String[] args) throws Exception {
        MazeGame mazeGame = new MazeGame();
```

```
        MazeFactory mazeFactory = new BombedMazeFactory();
        StandardBuilderMaze standardBuilderMaze = new
StandardBuilderMaze();
        CountingMazeBuilder countingMazeBuilder = new
CountingMazeBuilder();
        mazeGame.createMaze(standardBuilderMaze, mazeFactory);
        mazeGame.createMaze(countingMazeBuilder, mazeFactory);
        System.out.println("Liczba pokojów w labiryncie: " +
                standardBuilderMaze.getCurrentMaze().getRoomNumbers());
        System.out.println("Liczba utworzonych komponentów w labiryncie: "
+
                countingMazeBuilder.getCounts());
    }
}
```

- Wynik działania aplikacji na standardowym wyjściu:

```
Dodano pokój nr 1
Dodano pokój nr 2
Dodano pokój nr 3
Dodano drzwi pomiędzy pokojem 1 a 2
Dodano drzwi pomiędzy pokojem 2 a 3
Liczba pokojów w labiryncie: 3
Liczba utworzonych komponentów w labiryncie: 13
```

3. Singleton

Wprowadzono w powyżej stworzonej implementacji mechanizm, w którym MazeFactory jest Singletonem. Jest on dostępny z pozycji kodu, który jest odpowiedzialny za tworzenie poszczególnych części labiryntu.

- w celu wprowadzenia wzorca Singleton do trzech fabryk zrezygnowałem z dziedziczenia po MazeFactory, tworząc nową klasę abstrakcyjną AbstractFactory :

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

import pl.agh.edu.dp.labirynth.Door;
import pl.agh.edu.dp.labirynth.Room;
import pl.agh.edu.dp.labirynth.Wall;

public abstract class AbstractFactory {
    public abstract Room createRoom(int number);
```

```
    public abstract Door createDoor(Room r1, Room r2);
    public abstract Wall createWall();
}
```

Klasa ta definiuje abstrakcyjne metody, które są implementowane w klasach pochodnych (MazeFactory, EnchantedMazeFactory, BombedMazeFactory).

- teraz wprowadzono sam wzorzec Singleton do każdej z klas:

```
private static MazeFactory instance;

private MazeFactory() {}

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

```
private static EnchantedMazeFactory instance;

private EnchantedMazeFactory() {}

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

```
private static BombedMazeFactory instance;

private BombedMazeFactory() {}

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

- Zaktualizowany kod klasy Main wygląda następująco:

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

import pl.agh.edu.dp.builders.CountingMazeBuilder;
import pl.agh.edu.dp.builders.StandardBuilderMaze;
import pl.agh.edu.dp.factories.AbstractFactory;
import pl.agh.edu.dp.factories.BombedMazeFactory;
import pl.agh.edu.dp.labirynth.*;

public class Main {

    public static void main(String[] args) throws Exception {
        MazeGame mazeGame = new MazeGame();
        AbstractFactory bombedMazeFactory =
BombedMazeFactory.getInstance();
        StandardBuilderMaze standardBuilderMaze = new
StandardBuilderMaze();
        CountingMazeBuilder countingMazeBuilder = new
CountingMazeBuilder();
        mazeGame.createMaze(standardBuilderMaze, bombedMazeFactory);
        mazeGame.createMaze(countingMazeBuilder, bombedMazeFactory);
        System.out.println("Liczba pokojów w labiryncie: " +
                standardBuilderMaze.getCurrentMaze().getRoomNumbers());
        System.out.println("Liczba utworzonych komponentów w labiryncie: "
+
                countingMazeBuilder.getCounts());
    }
}
```

Widzimy, że obiekt dziedziczący po klasie AbstractFactory nie jest tworzony bezpośrednio w metodzie main a jedynie wywołujemy metodę `getInstance()` odpowiedzialną za jego utworzenie.

4. Rozszerzenie aplikacji labirynt

1. Korzystając z powyższych implementacji, dodano prosty mechanizm przemieszczania się po labiryncie. Po realizacji wcześniejszych zadań pozostało stworzyć prostą klasę Player, która za pomocą np. strzałek + tekstu w konsoli będzie mogła zadecydować o kierunku chodzenia. Rozpatrzono stosowne warianty rozgrywki (czy ściana ma drzwi, przez które możemy przejść itp. itd.). Wprowadzono elementy BombedRoom/BombedWall. Gdy gracz trafi na BombedRoom, wędrówka po labiryncie kończy się niepowodzeniem i symulacja się kończy.

- kod klasy Player, która odpowiedzialna jest za mechanikę ruchu gracza po planszy:

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

import pl.agh.edu.dp.products.bombed.BombedRoom;

public class Player {
    private Room currentRoom;
    private static Player instance;

    private Player() {}

    public static Player getInstance(){
        if (instance == null){
            instance = new Player();
        }
        return instance;
    }

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

    public void move(char x) throws Exception {
        switch (x) {
            case 'a' -> {
                this.moveLeft();
            }
            case 'd' -> {
                this.moveRight();
            }
            case 'w' -> {
                this.moveUp();
            }
            case 's' -> {
                this.moveDown();
            }
        }
        if (this.currentRoom.getClass() == BombedRoom.class) {
            throw new Exception();
        }
        System.out.println(this.currentRoom.getRoomNumber());
    }

    public Room getCurrentRoom() {
        return this.currentRoom;
    }
```

```
    private void moveLeft() {
        if (canGoTo(Direction.East)) {
            currentRoom.getSide(Direction.East).Enter(this);
        }
    }

    private void moveRight() {
        if (canGoTo(Direction.West)) {
            currentRoom.getSide(Direction.West).Enter(this);
        }
    }

    private void moveUp() {
        if (canGoTo(Direction.North)) {
            currentRoom.getSide(Direction.North).Enter(this);
        }
    }

    private void moveDown() {
        if (canGoTo(Direction.South)) {
            currentRoom.getSide(Direction.South).Enter(this);
        }
    }

    private boolean canGoTo(Direction direction) {
        return currentRoom.getSide(direction) instanceof Door;
    }
}
```

Warto zauważyć, że podczas tworzenia nowego obiektu powyższej klasy zastosowałem wzorzec Singleton, aby po labiryncie wędrował tylko jeden gracz.

- metoda Enter odpowiedzialna za ruch po planszy (zmianę pokoju przypisanego do gracza):

  Dla klasy Room:

```
@Override
public void Enter(Player player) {
    player.setCurrentRoom(this);
}
```

  Dla klasy Door:

```
@Override
public void Enter(Player player){
```

```
    Room r1 = player.getCurrentRoom();
    if (this.room1.equals(r1)) {
        room2.Enter(player);
    } else {
        room1.Enter(player);
    }
}
```

- Kod klasy MazeGame odpowiedzialny za tworzenie labiryntu oraz całą symulację:

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

import pl.agh.edu.dp.builders.StandardBuilderMaze;
import pl.agh.edu.dp.factories.AbstractFactory;
import pl.agh.edu.dp.factories.BombedMazeFactory;
import pl.agh.edu.dp.factories.MazeFactory;

import java.util.Scanner;

public class MazeGame {

    private final Player player = Player.getInstance();

    public MazeGame() throws Exception {
        this.createMaze();
    }

    public void start() {
        System.out.println("Witaj w labiryncie!");
        System.out.println("Spróbuj znaleźć wyjście, ale uważaj na
wybuchowe pokoje!");
        System.out.println("Użyj przycisków (a, w, s, d) do poruszania
się.");
        Scanner scanner = new Scanner(System.in);
        while (true) {
            try {
                char input = scanner.next().charAt(0);
                this.player.move(input);
                if (input == 'q') {
                    break;
                }
            } catch (Exception e) {
                System.out.println("Przegrałeś! Trafiłeś na wybuchowy
pokój!");
```

```
                return;
            }
        }
    }

    private void createMaze() throws Exception {
        StandardBuilderMaze standardBuilderMaze = new
StandardBuilderMaze();

        AbstractFactory bombedMazeFactory =
BombedMazeFactory.getInstance();
        AbstractFactory mazeFactory = MazeFactory.getInstance();
        Room r1 = mazeFactory.createRoom(1);
        Room r2 = mazeFactory.createRoom(2);
        Room r3 = bombedMazeFactory.createRoom(3);

        standardBuilderMaze.addRoom(r1);
        standardBuilderMaze.addRoom(r2);
        standardBuilderMaze.addRoom(r3);
        standardBuilderMaze.addDoor(r1, r2);
        standardBuilderMaze.addDoor(r2, r3);

        this.player.setCurrentRoom(r1);
    }
}
```

- I w końcu kod klasy Main:

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

import pl.agh.edu.dp.labirynth.*;

public class Main {

    public static void main(String[] args) throws Exception {
        MazeGame mazeGame = new MazeGame();
        mazeGame.start();
    }
}
```

- wyniki na standardowym wyjściu:

```
Witaj w labiryncie!
Spróbuj znaleźć wyjście, ale uważaj na wybuchowe pokoje!
Użyj przycisków (a, w, s, d) do poruszania się.
w
Aktualny numer pokoju: 2
s
Aktualny numer pokoju: 1
s
Aktualny numer pokoju: 1
a
Aktualny numer pokoju: 1
d
Aktualny numer pokoju: 1
w
Aktualny numer pokoju: 2
w
Aktualny numer pokoju: 3
Przegrałeś! Trafiłeś na wybuchowy pokój!
```

2. Zademonstrowano, że MazeFactory faktycznie jest Singletonem (stworzono przykład, w którym sprawdzono, czy obiekt zwracany przy 2 konstrukcji to faktycznie ten sam, który został stworzony na początku).

```
AbstractFactory bombedMazeFactory = BombedMazeFactory.getInstance();
System.out.println(bombedMazeFactory);
bombedMazeFactory = BombedMazeFactory.getInstance();
System.out.println(bombedMazeFactory);
```

Powyższy kod generuje następujące dane na standardowym wyjściu:

```
pl.agh.edu.dp.factories.BombedMazeFactory@404b9385
pl.agh.edu.dp.factories.BombedMazeFactory@404b9385
```

Widzimy zatem, iż po wywołaniu drugi raz metody tworzącej obiekt MazeFactory nadal mamy do czynienia z tym samym obiektem.