Projektowanie obiektowe

Lab 3 Wzorce projektowe – cz. 1

Dawid Białka, Piotr Tekielak

1. Builder

Tworzymy interfejs MazeBuilder – będzie on zawierał metody budujące komponenty labiryntu.

```
public interface MazeBuilder {

    Room createRoom();

    void addDoor(Room room1, Room room2);

    void addWall(Room room1, Room room2, Direction direction);
}
```

Od teraz w funkcji createMaze w klasie MazeGame będziemy korzystali z MazeBuildera.

Następnie tworzymy klasę StandardBuilderMaze będącą implementacją MazeBuildera.

```
public class StandardBuilderMaze implements MazeBuilder {

    private Maze currentMaze;

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

    public Maze getCurrentMaze() { return currentMaze; }
```

W niej implementujemy metody z klasy MazeBuilder:
createRoom – tworzy nowy pokój i ściany wokół niego, po czym dodaje go do obecnego labiryntu
addDoor – używając metody commonWall do znalezienia wspólnej ściany między dwoma pokojami tworzy (o ile taka istnieje) tworzy między nimi drzwi
addWall – tworzy ścianę między dwoma pokojami

```java
    @Override
    public Room createRoom() {
        Room room = new Room(currentMaze.getRoomNumbers());
        for(Direction direction : Direction.values()) {
            room.setSide(direction, new Wall());
        }
        currentMaze.addRoom(room);
        return room;
    }

    @Override
    public void addDoor(Room room1, Room room2) {
        Direction direction = commonWall(room1, room2);
        if(direction == null) {
            return;
        }
        Door door = new Door(room1, room2);
        room1.setSide(direction, door);
        room2.setSide(Direction.getOpposite(direction), door);
    }

    @Override
    public void addWall(Room room1, Room room2, Direction direction) {
        Wall wall = new Wall();
        room1.setSide(direction, wall);
        room2.setSide(Direction.getOpposite(direction), wall);
    }

    private Direction commonWall(Room room1, Room room2) {
        for (Direction direction : Direction.values()) {
            if (room1.getSide(direction).equals(room2.getSide(Direction.getOpposite(direction)))) {
                return direction;
            }
        }
        return null;
    }
}
```

Dodatkowo w enumie Direction tworzymy metodę getOpposite, która zwraca kierunek przeciwny do podanego.

```java
public static Direction getOpposite(Direction direction) {
    switch (direction) {
        case North: return South;
        case East: return West;
        case South: return North;
        default: return East;
    }
}
```

W metodzie createMaze klasy MazeGame używamy StandardBuilderMaze’a.

```java
public Maze createMaze(StandardBuilderMaze mazeBuilder) {

    Room r1 = mazeBuilder.createRoom();
    Room r2 = mazeBuilder.createRoom();

    mazeBuilder.addDoor(r1, r2);

    mazeBuilder.addWall(r1, r2, Direction.East);

    mazeBuilder.addDoor(r1, r2);

    return mazeBuilder.getCurrentMaze();
}
```

Tworzymy też podklasę MazeBuildera o nazwie CountingMazeBuilder. Budowniczy tego obiektu w ogóle nie tworzy labiryntu, a jedynie zlicza utworzone komponenty różnych rodzajów. Posiada metodę GetCounts, która zwraca ilość elementów.

```java
public class CountingMazeBuilder implements MazeBuilder {

    private int roomsNumber;
    private int wallNumber;
    private int doorNumber;

    public CountingMazeBuilder() {
        this.roomsNumber = 0;
        this.wallNumber = 0;
        this.doorNumber = 0;
    }

    @Override
    public Room createRoom() {
        roomsNumber += 1;
        wallNumber += 4;
        return new Room( number: 1);
    }

    @Override
    public void addDoor(Room room1, Room room2) {
        wallNumber -= 1;
        doorNumber += 1;
    }

    @Override
    public void addWall(Room r1, Room r2, Direction direction) { wallNumber -=1; }

    public String getCounts(){
        return "Room number: " + this.roomsNumber + "\nDoor number: " + this.doorNumber + "\nWall number: " + this.wallNumber;
    }

    public int getRoomsNumber() { return roomsNumber; }

    public int getWallNumber() { return wallNumber; }

    public int getDoorNumber() { return doorNumber; }
}
```

2. Fabryka abstrakcyjna

Tworzymy klasę MazeFactory – będzie ona tworzyć elementy labiryntu.

```
public class MazeFactory {

    private static MazeFactory factory;

    public static MazeFactory getFactory() {
        if(factory == null)
            factory = new MazeFactory();
        return factory;
    }

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

    public Wall makeWall() { return new Wall(); }

    public Door makeDoor(Room room1, Room room2) { return new Door(room1, room2); }
}
```

Zmieniamy metodę createMaze w klasie MazeGame aby przyjmowała obiekt typu MazeFactory.

```
public Maze createMaze(MazeFactory factory) {

    StandardBuilderMaze mazeBuilder = new StandardBuilderMaze(factory);
    Room r1 = mazeBuilder.createRoom();
    Room r2 = mazeBuilder.createRoom();

    mazeBuilder.addDoor(r1, r2);

    mazeBuilder.addWall(r1, r2, Direction.East);

    mazeBuilder.addDoor(r1, r2);

    return mazeBuilder.getCurrentMaze();
}
```

Zmieniamy też implementację klasy StandardBuilderMaze – od teraz posiada ona też prywatny atrybut typu MazeFactory, który będzie wykorzystywany do tworzenia nowych elementów labiryntu.

```java
public class StandardBuilderMaze implements MazeBuilder {

    private Maze currentMaze;
    private MazeFactory factory;

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

    public Maze getCurrentMaze() { return currentMaze; }

    @Override
    public Room createRoom() {
        Room room = factory.makeRoom(currentMaze.getRoomNumbers());
        for(Direction direction : Direction.values()) {
            room.setSide(direction, factory.makeWall());
        }
        currentMaze.addRoom(room);
        return room;
    }

    @Override
    public void addDoor(Room room1, Room room2) {
        Direction direction = commonWall(room1, room2);
        if(direction == null) {
            return;
        }
        Door door = factory.makeDoor(room1, room2);
        room1.setSide(direction, door);
        room2.setSide(Direction.getOpposite(direction), door);
    }

    @Override
    public void addWall(Room room1, Room room2, Direction direction) {
        Wall wall = factory.makeWall();
        room1.setSide(direction, wall);
        room2.setSide(Direction.getOpposite(direction), wall);
    }

    private Direction commonWall(Room room1, Room room2) {
        for (Direction direction : Direction.values()) {
            if (room1.getSide(direction).equals(room2.getSide(Direction.getOpposite(direction)))) {
                return direction;
            }
        }
        return null;
    }
}
```

Następnie tworzymy klasę EnchantedMazeFactory, która dziedziczy z MazeFactory. Tworzy ona nowe typy elementów labiryntu, których klasy dziedziczą z klas pierwotnych.

```java
public class EnchantedMazeFactory extends MazeFactory {

    public Room makeRoom(int number) {
        return new EnchantedRoom(number);
    }

    public Wall makeWall() {
        return new EnchantedWall();
    }

    public Door makeDoor(Room room1, Room room2) {
        return new EnchantedDoor(room1, room2);
    }
}
```

```java
public class EnchantedRoom extends Room {

    public EnchantedRoom(int number) {
        super(number);
    }
}
```

```java
public class EnchantedWall extends Wall {
}
```

```java
public class EnchantedDoor extends Door {

    public EnchantedDoor(Room room1, Room room2) {
        super(room1, room2);
    }
}
```

Tworzymy też klasę BombedMazeFactory, która także dziedziczy z MazeFactory oraz dodatkowe wersje pokoju i ściany.

```java
public class BombedMazeFactory extends MazeFactory{

    public Room makeRoom(int number) {
        return new BombedRoom(number);
    }

    public Wall makeWall() {
        return new BombedWall();
    }
}
```

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

```java
public class BombedWall extends Wall {
}
```

3. Singleton

MazeFactory już jest singletonem.

```
public class MazeFactory {

    private static MazeFactory factory;

    public static MazeFactory getFactory() {
        if(factory == null)
            factory = new MazeFactory();
        return factory;
    }

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

    public Wall makeWall() { return new Wall(); }

    public Door makeDoor(Room room1, Room room2) { return new Door(room1, room2); }
}
```

Jest dostępny z pozycji kodu, który jest odpowiedzialny z tworzenie poszczególnych części labiryntu.

```
public Maze createMaze(MazeFactory factory) {

    StandardBuilderMaze mazeBuilder = new StandardBuilderMaze(factory);
    Room r1 = mazeBuilder.createRoom();
    Room r2 = mazeBuilder.createRoom();

    mazeBuilder.addDoor(r1, r2);

    mazeBuilder.addWall(r1, r2, Direction.East);

    mazeBuilder.addDoor(r1, r2);

    return mazeBuilder.getCurrentMaze();
}
```

4. Rozszerzenie aplikacji labirynt

a) Dodajemy nową klasę Player – gracz będzie obracał się i poruszał. Jeśli wejdzie do pokoju z bombą, ginie. Jeśli odkryje wszystkie pokoje to gracz wygrywa.

```java
public class Player {
    private boolean alive;
    private Direction direction;
    private Room room;

    public Player(Direction direction, Room room) {
        this.alive = true;
        this.direction = direction;
        this.room = room;
    }

    public void turnLeft() { this.direction = Direction.getPrevious(this.direction); }

    public void turnRight() {
        this.direction = Direction.getNext(this.direction);
    }

    public void moveAhead() {
        MapSite mapSite = room.getSide(this.direction);
        mapSite.Enter( player: this);
    }

    public Direction getDirection() { return direction; }

    public void setDirection(Direction direction) { this.direction = direction; }

    public Room getRoom() { return room; }

    public void setRoom(Room room) { this.room = room; }

    public boolean isAlive() {
        return alive;
    }

    public void setAlive(boolean alive) {
        this.alive = alive;
    }
}
```

```java
public void startGame() {
    this.createMaze(MazeFactory.getFactory());
    Player player = new Player(Direction.North, this.maze.getRooms().get(0));
    while(player.isAlive() && !player.getRoom().equals(this.maze.getRooms().get(this.maze.getRoomNumbers() - 1))){
        cycle(player);
    }
    if(player.getRoom().equals(this.maze.getRooms().get(this.maze.getRoomNumbers() - 1)))
        System.out.println("You've explored all the rooms");
    System.out.println("Game over");
}
```

Tworzymy przykładowy labirynt z dwoma pokojami i rozpoczynamy grę:

```
Press:
a - to turn left
d - to turn right
w - to go forward
d

Press:
a - to turn left
d - to turn right
w - to go forward
w
You go through a door
You're in a room
You've explored all the rooms
Game over
```

b) Weryfikujemy czy MazeFactory jest singletonem:

```
MazeFactory m1 = MazeFactory.getFactory();
MazeFactory m2 = MazeFactory.getFactory();
System.out.println(m1 == m2);
```

```
true
```