

AGH

**AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE**

Projektowanie obiektowe

Wzorce projektowe

Laboratorium 3

Przemysław Lechowicz

1. Zdefiniuj nową wersję funkcji składowej createMaze, która będzie przyjmować jako argument obiekt budujący klasy MazeBuilder.

- 1.1. Stwórz klasę MazeBuilder, która definiuje interfejs służący do tworzenia labiryntów

Został stworzony interfejs MazeBuilder, za pomocą którego będą tworzone labirynty.

```
public interface MazeBuilder {  
    void addRoom(Room room);  
    void createDoor(Room r1, Room r2);  
    void createWall(Direction firstRoomDir, Room r1, Room r2);  
}
```

- 1.2. Po utworzeniu powyższego interfejsu zmodyfikuj funkcję składową

Po dodaniu interfejsu funkcję składową createMaze() została zmodyfikowana w ten sposób, aby jako argument przyjmowała obiekt klasy MazeBuilder.

```
public class MazeGame {  
    public Maze createMaze(MazeBuilder mazeBuilder){  
        ...  
    }  
}
```

- 1.3. Prześledź i zinterpretuj co dały obecne zmiany.

Tworzenie złożonego obiektu zostało oddzielone od jego reprezentacji. Wprowadzone zmiany pozwoliły na uniknięcie powtarzania kodu poprzez tworzenie obiektów w innej klasie. Ten sam proces konstrukcji może prowadzić do powstawania różnych reprezentacji.

1.4. Stwórz klasę StandardBuilderMaze będącą implementacją MazeBuildera.

Do enuma Direction została dodana metoda opposite() zwracająca kierunek odwrotny do obecnego:

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

    public Direction opposite() {
        switch (this){
            case North:
                return South;
            case South:
                return North;
            case East:
                return West;
            case West:
                return East;
            default:
                return null;
        }
    }
}
```

Została utworzona klasa StandardMazeBuilder będącą implementacją interfejsu MazeBuilder.

Zawiera ona zmienną currentMaze, w której jest zapisywany obecny stan labiryntu, oraz metody:

- getCurrnetMaze(), zwracającą obecny stan labiryntu
- addRoom(Room room), tworzącą nowy pokój
- createDoor(Room room1, Room room2), tworzącą drzwi między pomieszczeniami
- createWall(Direction firstRoomDir, Room room1, Room room2), tworzącą ścianę między dwoma pokojami
- commonWall(Room room1, Room room2), zwracającą wspólną ścianę między pokojami

```

public class StandardMazeBuilder implements MazeBuilder {
    private Maze currentMaze;

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

    public Maze getCurrentMaze() {
        return currentMaze;
    }

    @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());
        currentMaze.addRoom(room);
    }

    @Override
    public void createDoor(Room room1, Room room2) {
        Door door = new Door(room1, room2);
        room1.setSide(commonWall(room1, room2), door);
        assert commonWall(room1, room2) != null;
        room2.setSide(commonWall(room1, room2).opposite(), door);
    }

    @Override
    public void createWall(Direction firstRoomDir, Room room1, Room room2) {
        if (room1.getSide(firstRoomDir) == null)
            return;
        room2.setSide(firstRoomDir.opposite(), room1.getSide(firstRoomDir));
    }

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

```

1.5. Utwórz labirynt przy pomocy operacji createMaze, gdzie parametrem będzie obiekt klasy StandardMazeBuilder.

Została zmodyfikowana metoda createMaze, tak aby przyjmowała obiekt klasy StandardMazeBuilder, a następnie utworzono przy jej pomocy labirynt z dwoma pokojami.

```
public class MazeGame {
    public Maze createMaze(StandardMazeBuilder standardMazeBuilder){
        Maze maze = new Maze();

        Room room1 = new Room(1);
        Room room2 = new Room(2);

        standardMazeBuilder.addRoom(room1);
        standardMazeBuilder.addRoom(room2);
        standardMazeBuilder.createWall(Direction.South,room1,room2);
        standardMazeBuilder.createDoor(room1,room2);
        return standardMazeBuilder.getCurrentMaze();
    }
}
```

Oraz zmodyfikowano metodę Main()

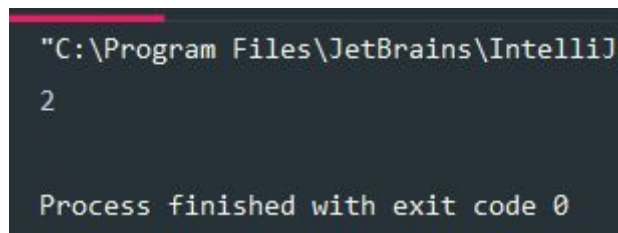
```
public class Main {

    public static void main(String[] args) {

        MazeGame mazeGame = new MazeGame();
        StandardMazeBuilder standardMazeBuilder = new StandardMazeBuilder();
        Maze maze = mazeGame.createMaze(standardMazeBuilder);

        System.out.println(maze.getRoomNumbers());
    }
}
```

Program zgodnie z oczekiwaniami zwraca liczbę 2:



```
"C:\Program Files\JetBrains\IntelliJ
2

Process finished with exit code 0
```

1.6. Stwórz kolejną podklasę MazeBuildera o nazwie CountingMazeBuilder.

Utworzona została klasa CountingMazeBuilder zliczająca ilość ścian, pokoi i drzwi w labiryncie.

```
public class CountingMazeBuilder implements MazeBuilder {
    private int rooms;
    private int doors;
    private int walls;

    public CountingMazeBuilder() {
        this.rooms = 0;
        this.doors = 0;
        this.walls = 0;
    }

    @Override
    public void addRoom(Room room) {
        rooms++;
        walls += 4;
    }

    @Override
    public void createDoor(Room r1, Room r2) throws Exception {
        walls--;
        doors++;
    }

    @Override
    public void createWall(Direction firstRoomDir, Room r1, Room r2) {
        walls--;
    }

    public int getCounts() {
        return rooms + walls + doors;
    }
}
```

Klasy MazeGame i Main zostały odpowiednio zmodyfikowane:

```
public class MazeGame {
    public void createMaze(CountingMazeBuilder countingMazeBuilder) {
        Maze maze = new Maze();

        Room room1 = new Room(1);
        Room room2 = new Room(2);

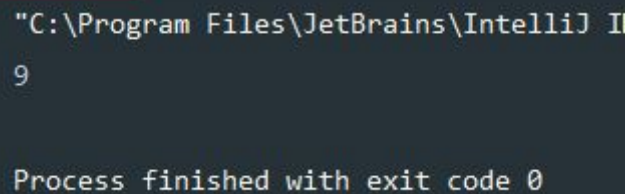
        countingMazeBuilder.addRoom(room1);
        countingMazeBuilder.addRoom(room2);
        countingMazeBuilder.createWall(Direction.South, room1, room2);
        countingMazeBuilder.createDoor(room1, room2);

        System.out.println(countingMazeBuilder.getCounts());
    }
}
```

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

        MazeGame mazeGame = new MazeGame();
        CountingMazeBuilder countingMazeBuilder = new CountingMazeBuilder();
        mazeGame.createMaze(countingMazeBuilder);
    }
}
```

Aplikacja zgodnie z oczekiwaniami wypisuje liczbę 9:



```
"C:\Program Files\JetBrains\IntelliJ I
9

Process finished with exit code 0
```

3. Fabryka abstrakcyjna

3.1. Stwórz klasę MazeFactory

Została stworzona klasa Vector2D

```
public class Vector2D {
    private final int x;
    private final int y;

    private Vector2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Vector2D next(Direction dir) {
        switch (dir) {
            case South:
                return new Vector2D(this.x, this.y - 1);
            case East:
                return new Vector2D(this.x + 1, this.y);
            case West:
                return new Vector2D(this.x - 1, this.y);
            case North:
                return new Vector2D(this.x, this.y + 1);
        }
        throw new NullPointerException("Pos doesnt exist");
    }

    public Vector2D upperRight(Vector2D other) {
        return new Vector2D(Math.max(this.x, other.x), Math.max(this.y, other.y));
    }
}
```

Do klasy Room została dodana zmienna typu Vector2D odwzorowująca pozycję pokoju.

```
public class Room extends MapSite
{
    private Map<Direction, MapSite> sides;
    private Vector2D position;

    public Room(Vector2D position) {
        this.sides = new EnumMap<>(Direction.class);
        this.position = position;
    }

    public MapSite getSide(Direction direction){
        return this.sides.get(direction);
    }

    public void setSide(Direction direction, MapSite ms){
        this.sides.put(direction, ms);
    }
}
```



```

    }

    @Override
    public void Enter(){

    }
}

```

Do klasy Maze dodano metodę findRoom, która zwraca pokój, który znajduje się na danej pozycji

```

public class Maze {
    private Vector<Room> rooms;

    Maze() {
        this.rooms = new Vector<Room>();
    }

    void addRoom(Room room) {
        rooms.add(room);
    }

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

    public int getRoomNumbers() {
        return rooms.size();
    }

    public Room findRoom(Vector2D pos) {
        Optional<Room> foundRoom = rooms.stream().filter(room ->
room.getPosition().equals(pos)).findAny();
        if (foundRoom.isPresent()) {
            return foundRoom.get();
        } else return null;
    }
}

```

Została stworzona klasa MazeFactory, która tworzy elementy labiryntu

```
public class MazeFactory {  
  
    public Door createDoor(Room room1, Room room2) {  
        return new Door(room1, room2);  
    }  
  
    public Room createRoom(Vector2D position) {  
        return new Room(position);  
    }  
  
    public Wall createWall() {  
        return new Wall();  
    }  
}
```

3.2. Przeprowadź kolejną modyfikację funkcji createMaze tak, aby jako parametr brała MazeFactory

```
public class MazeGame {  
    public Maze createMaze(MazeFactory mazeFactory, CountingMazeBuilder  
countingMazeBuilder) {  
        Maze maze = new Maze();  
  
        Room room1 = mazeFactory.createRoom(new Vector2D(1, 2));  
        Room room2 = new Room(new Vector2D(2, 2));  
  
        countingMazeBuilder.addRoom(room1);  
        countingMazeBuilder.addRoom(room2);  
        countingMazeBuilder.createWall(Direction.South, room1, room2);  
        countingMazeBuilder.createDoor(room1, room2);  
  
        return countingMazeBuilder.getCurrentMaze();  
    }  
}
```

3.4. Stwórz klasę EnchantedMazeFactory

Została utworzone klasy reprezentujące magiczne pokoje, drzwi i ściany, dziedziczące po klasach Room, Door, Wall.

```
public class EnchantedRoom extends Room {
    public EnchantedRoom(Vector2D position) {
        super(position);
    }
}

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

public class EnchantedWall extends Wall {
}
```

Została stworzona klasa EnchantedMazeFactory dziedzicząca po MazeFactory

```
public class EnchantedMazeFactory extends MazeFactory {
    @Override
    public Door createDoor(Room r1, Room r2) {
        return new EnchantedDoor(r1, r2);
    }

    @Override
    public Room createRoom(Vector2D pos) {
        return new EnchantedRoom(pos);
    }

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

3.5. Stwórz klasę BombedMazeFactory

Podobnie jak w poprzednim podpunkcie utworzone zostały nowe klasy reprezentujące pokoje, drzwi i ściany.

```
public class BombedRoom extends Room {
    public BombedRoom(Vector2D position) {
        super(position);
    }
}

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

public class BombedWall extends Wall {
}
```

Utworzona została klasa BombedMazeFactory dziedzicząca po MazeFactory:

```
public class BombedMazeFactory extends MazeFactory{
    @Override
    public Door createDoor(Room r1, Room r2) {
        return new BombedDoor(r1, r2);
    }

    @Override
    public Room createRoom(Vector2D pos) {
        return new BombedRoom(pos);
    }

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

5. Singleton

W celu zmiany MazeFactory w Singleton dodano do klas MazeFactory, EnchantedMazeFactory i BombedMazeFactory ich statyczne instancje.

```
public class MazeFactory {

    private static MazeFactory instance;

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

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

    public Room createRoom(Vector2D position) {
        return new Room(position);
    }

    public Wall createWall() {
        return new Wall();
    }
}
```

```
public class EnchantedMazeFactory extends MazeFactory {

    private static EnchantedMazeFactory instance;

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

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

    @Override
    public Room createRoom(Vector2D pos) {
        return new EnchantedRoom(pos);
    }

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

```

public class BombedMazeFactory extends MazeFactory{

    private static BombedMazeFactory instance;

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

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

    @Override
    public Room createRoom(Vector2D pos) {
        return new BombedRoom(pos);
    }

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

```

6. Rozszerzenie aplikacji labirynt

6.1. Mechanizm przemieszczania się po labiryncie

Stworzono klasę MoveDirection

```

public enum MoveDirection {
    FORWARD,
    BACKWARD,
    RIGHT,
    LEFT
}

```

Natomiast do klasy Direction zostały dodane metody next() i previous zwracające kolejno następny oraz poprzedni kierunek.

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

    public Direction opposite() {
        switch (this){
            case North:
                return South;
            case South:
                return North;
            case East:
                return West;
            case West:
                return East;
            default:
                return null;
        }
    }
    public Direction next() {
        switch (this){
            case North:
                return East;
            case South:
                return West;
            case East:
                return South;
            case West:
                return North;
            default:
                return null;
        }
    }
    public Direction previous() {
        switch (this){
            case North:
                return West;
            case South:
                return East;
            case East:
                return North;
            case West:
                return South;
            default:
                return null;
        }
    }
}
```

Oraz klasę Player z metodą move, która służy poruszaniu się po mapie:

```
public class Player {
    private Vector2D position;
    private Direction direction;

    public Player(Vector2D position){
        this.position = position;
        this.direction = Direction.North;
    }

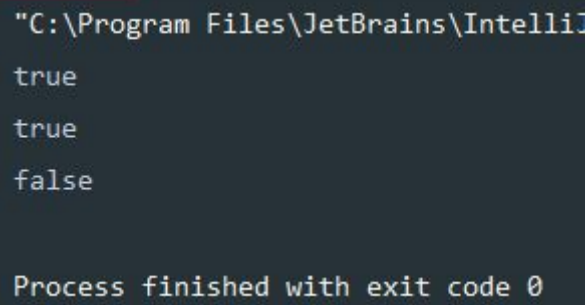
    void Move(MoveDirection moveDirection){
        switch (moveDirection){
            case FORWARD:
                switch (this.direction){
                    case North:
                        this.position = new
Vector2D(this.position.getX(),this.position.getY()+1);
                    case South:
                        this.position = new
Vector2D(this.position.getX(),this.position.getY()-1);
                    case East:
                        this.position = new
Vector2D(this.position.getX()+1,this.position.getY());
                    case West:
                        this.position = new
Vector2D(this.position.getX()-1,this.position.getY());
                }
            case LEFT:
                this.direction = this.direction.previous();
            case RIGHT:
                this.direction = this.direction.next();
            case BACKWARD:
                this.direction = this.direction.opposite();
        }
    }
}
```


6.2. Zademonstruj, że MazeFactory faktycznie jest Singletonem

W tym celu zmodyfikowano metodę main:

```
public class Main {  
    public static void main(String[] args) {  
  
        MazeFactory mazeFactory = EnchantedMazeFactory.getInstance();  
        System.out.println(mazeFactory.equals(EnchantedMazeFactory.getInstance()));  
        MazeFactory mazeFactory1 = BombedMazeFactory.getInstance();  
        System.out.println(mazeFactory1.equals(BombedMazeFactory.getInstance()));  
        System.out.println(mazeFactory.equals(mazeFactory1));  
    }  
}
```

Program wypisuje:



```
"C:\Program Files\JetBrains\IntelliJ  
true  
true  
false  
  
Process finished with exit code 0
```

Co dowodzi, że MazeFactory jest singletonem.