Wojciech Kosztyła
nr. indeksu 305 338

Projektowanie obiektowe

Laboratorium 3

Wzorce projektowe

4.1 Builder

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

1. Stwórz klasę MazeBuilder, która definiuje interfejs służący do tworzenia labiryntów. Co musi tam być zawarte? Wykorzystaj wiedzę nt. składowych, które są w labiryncie.

1. Stworzyłem interfejs MazeBuilder, a w niej:
    a. dodawanie pokoju
    b. dodawanie drzwi
    c. „uwspółdzielanie” (czynienie wspólną) ściany

1.
```java
package pl.agh.edu.dp.labirynth.builder;

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

public interface MazeBuilder {

    void addRoom(Room room);

    void addDoor(Room room_1, Room room_2) throws Exception;

    void CommonWall(Room room_1, Room room_2, Direction room_1_direction);
    // Skoro ściany przechowywane są jako "Map<Direction, MapSite>",
    // to ściany, które są "przy sobie" mogę traktować jako jeden obiekt, zamiast przechowywać "podwójnej ściany"

```

2.

2.

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

1.

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

import pl.agh.edu.dp.labirynth.builder.MazeBuilder;

public class MazeGame {
    public Maze createMaze(MazeBuilder mazeBuilder) throws Exception{

        Room room_1 = new Room( number: 1);
        Room room_2 = new Room( number: 2);

        mazeBuilder.addRoom(room_1);
        mazeBuilder.addRoom(room_2);

        mazeBuilder.CommonWall(room_1, room_2, Direction.East);
        mazeBuilder.addDoor(room_1, room_2);

        return mazeBuilder.getMaze();
    }
}
```

3.

3. Prześledź i zinterpretuj co dały obecne zmiany (krótko opisz swoje spostrzeżenia).

   a. Wprowadzone zmiany znacznie zwiększyły czytelność kodu.
   b. Zachowanie zasady DRY – Don’t Repeat Yourself
   c. Ułatwienie wprowadzania zmian poprzez „modularyzację”, tworzenie klas implementujących MazeBuilder.

4.

4. Stwórz klasę StandardBuilderMaze będącą implementacją MazeBuildera. Powinna ona mieć zmienną currentMaze, w której jest zapisywany obecny stan labiryntu. Powinniśmy móc: tworzyć pomieszczenie i ściany w okół niego, tworzyć drzwi pomiędzy pomieszczeniami (czyli musimy wyszukać odpowiednie pokoje oraz ścianę, która je łączy). Dodaj tam dodatkowo metodę prywatną CommonWall, która określi kierunek standardowej ściany pomiędzy dwoma pomieszczeniami.

a. aby łatwiej implementować te metody, stworzyłem metodę w „Direction", która zwraca mi przeciwny kierunek

1.
```java
package pl.agh.edu.dp.labirynth;

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

    public Direction oppositeDirection(){
        switch(this){
            case North:
                return South;
            case South:
                return North;
            case East:
                return West;
            case West:
                return East;
            default:
                System.out.println("Nie ma takiego kierunku! - " + this);
                System.out.println("Zwracam North jako defaultowy kierunek.");
                return North;
        }
    }

}
```

b. Następnie wykorzystując metody zdefiniowane w „Room.java" stworzyłem zadane metody.

i. Nie byłem pewny, co do „dodatkowej metody CommonWall", która „określi kierunek standardowej ściany pomiędzy dwoma pomieszczeniami". Pomieszczenia nie są ustawione w macierzy AxB, sam labirynt może być nieskończenie długi i nieregularny, a numery pokojów nie muszą odzwierciedlać ich „sąsiedztwa".
Uznałem, że metoda CommonWall „określi" = „ustawi" wspólną ścianę dla dwóch pomieszczeń.

1.
```java
package pl.agh.edu.dp.labirynth.builder;

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

public class StandardMazeBuilder implements MazeBuilder {
    private Maze currentMaze;

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

    @Override
    public void addRoom(Room room) {
        for (Direction direction : Direction.values()){
            room.setSide(direction, new Wall());
        }
        currentMaze.addRoom(room);
    }
```

2.
```java
    @Override
    public void addDoor(Room room_1, Room room_2) throws Exception {
        Direction room_1_direction = null;
        for (Direction direction : Direction.values()){
            if(room_1.getSide(direction) == room_2.getSide(direction.oppositeDirection()) ){
                room_1_direction = direction;
                break;
            }
        }
        if(room_1_direction == null){
            throw new Exception("Pokoje " + room_1 + " i " + room_2 + ", nie mają wspólnej ściany.");
        }
        else{
            Door door = new Door(room_1, room_2);
            room_1.setSide(room_1_direction, door);
            room_2.setSide(room_1_direction.oppositeDirection(), door);
        }
    }
```

3.
```java

    @Override
    public void CommonWall(Room room_1, Room room_2, Direction room_1_direction){
        MapSite wall = room_1.getSide(room_1_direction);
        room_2.setSide(room_1_direction.oppositeDirection(), wall);
    }

    @Override
    public Maze getMaze() {
        return currentMaze;
    }


}
```

5.

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

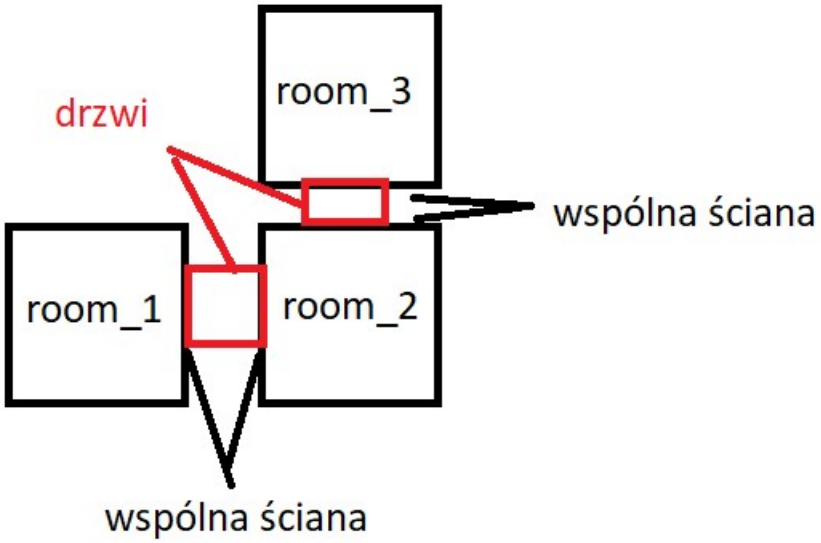
a. Zmieniłem „Main.java”:

1.
```java
package pl.agh.edu.dp.main;

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

public class Main {

    public static void main(String[] args) throws Exception {

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

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

b. Następnie zmodyfikowałem „MazeGame.java”, aby tworzyła nieco bardziej „skomplikowany” labirynt.

1.
```java
public class MazeGame {
    public Maze createMaze(MazeBuilder mazeBuilder) throws Exception{

        Room room_1 = new Room( number: 1);
        Room room_2 = new Room( number: 2);
        Room room_3 = new Room( number: 3);

        mazeBuilder.addRoom(room_1);
        mazeBuilder.addRoom(room_2);
        mazeBuilder.addRoom(room_3);

        mazeBuilder.CommonWall(room_1, room_2, Direction.East);
        mazeBuilder.addDoor(room_1, room_2);

        mazeBuilder.CommonWall(room_2, room_3, Direction.North);
        mazeBuilder.addDoor(room_2, room_3);

        return mazeBuilder.getMaze();
    }
}
```

c. Szkic labiryntu:



1.

6.

6. Stwórz 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.

1.

```java
package pl.agh.edu.dp.labirynth.builder;
import pl.agh.edu.dp.labirynth.*;

public class CountingMazeBuilder implements MazeBuilder {
    private int counter;

    public CountingMazeBuilder(){
        this.counter = 0;
    }

    @Override
    public void addRoom(Room room) {
        this.counter = this.counter + 5;
    }

    @Override
    public void addDoor(Room room_1, Room room_2) throws Exception {
        this.counter++;
    }

    @Override
    public void CommonWall(Room room_1, Room room_2, Direction room_1_direction){
        this.counter--;
    }

    public int GetCounts(){
        return this.counter;
    }
}
```

4.2 Fabryka abstrakcyjna

1. Stwórz 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.

   a. Stworzona klasa:

1.
```java
package pl.agh.edu.dp.labirynth.factory;

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 room_1, Room room_2){
        return new Door(room_1, room_2);
    }

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

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

1.
```java
public class MazeGame {
    public Maze createMaze(StandardMazeBuilder mazeBuilder, MazeFactory mazeFactory) throws Exception{
```

3.

Stwórz klasę EnchantedMazeFactory (fabryka magicznych labiryntów), która dziedziczy z MazeFactory. Powinna przesłaniać kilka funkcji składowych i zwracać różne podklasy klas Room, Wall itd. (należy takie klasy również stworzyć).

a. Rozpocząłem od stworzenia „enchanted” elementów:

1.
```java
package pl.agh.edu.dp.labirynth.factory;

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

public class EnchantedRoom extends Room {

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

    @Override
    public void Enter(){
        System.out.println("Enchanted Room");
        super.Enter();
    }
}
```

2.
```java
package pl.agh.edu.dp.labirynth.factory;

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);
    }

    @Override
    public void Enter(){
        System.out.println("Enchanted Door");
        super.Enter();
    }
}
```

3.
```java
package pl.agh.edu.dp.labirynth.factory;

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

public class EnchantedWall extends Wall {

    @Override
    public void Enter(){
        System.out.println("Enchanted Door");
        super.Enter();
    }
}
```

b. Następnie stworzyłem EnchantedMazeFactory:

1.
```java
package pl.agh.edu.dp.labirynth.factory;


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

public class EnchantedMazeFactory extends MazeFactory {

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

    @Override
    public Door createDoor(Room room_1, Room room_2){
        return new EnchantedDoor(room_1, room_2);
    }

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

4.

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

a. Podobnie do przykładu powyżej, stworzyłem BombedWall, BombedRoom i BombedMazeFactory:

1.
```java
package pl.agh.edu.dp.labirynth.factory;

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

public class BombedWall extends Wall {

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

2.
```java
package pl.agh.edu.dp.labirynth.factory;

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

public class BombedRoom extends Room {

    public BombedRoom(int number) {
        super(number);
    }

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

3.
```java
package pl.agh.edu.dp.labirynth.factory;

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

public class BombedMazeFactory extends MazeFactory {

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

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

4.3 Singleton

1.

Wprowadź w powyżej stworzonej implementacji mechanizm, w którym MazeFactory będzie Singletonem. Powinien być on dostępny z pozycji kodu, który jest odpowiedzialny z tworzenie poszczególnych części labiryntu.

a. Stworzyłem statyczne pole w MazeFactory:

1.
```java
public class MazeFactory {

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

b. I w klasach pochodnych MazeFactory:

1.
```
public class EnchantedMazeFactory extends MazeFactory {

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

2.
```
public class BombedMazeFactory extends MazeFactory {

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

2. Zmodyfikowałem StandardMazeBuilder, by korzystał z MazeFactory:

1.
```
public class StandardMazeBuilder implements MazeBuilder {
    private Maze currentMaze;
    private MazeFactory factory;

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

2.
```
    @Override
    public void addRoom(Room room) {
        for (Direction direction : Direction.values()){
            //room.setSide(direction, new Wall());
            room.setSide(direction, factory.createWall());
        }
        currentMaze.addRoom(room);
    }
```

3.
```java
@Override
public void addDoor(Room room_1, Room room_2) throws Exception {
    Direction room_1_direction = null;
    for (Direction direction : Direction.values()){
        if(room_1.getSide(direction) == room_2.getSide(direction.oppositeDirection()) ){
            room_1_direction = direction;
            break;
        }
    }
    if(room_1_direction == null){
        throw new Exception("Pokoje " + room_1 + " i " + room_2 + ", nie mają wspólnej ściany.");
    }
    else{
        //Door door = new Door(room_1, room_2);
        Door door = factory.createDoor(room_1, room_2);
        room_1.setSide(room_1_direction, door);
        room_2.setSide(room_1_direction.oppositeDirection(), door);
    }
}
```

1.

4.4 Rozszerzenie aplikacji labirynt

a) Korzystając z powyższych implementacji dodaj prosty mechanizm przemieszczania się po labiryncie. Po realizacji wcześniejszych zadań pozostaje stworzyć prostą klasę Player, która za pomocą np. strzałek + tekstu w konsoli będzie mogła zadecydować o kierunku chodzenia. Rozpatrz stosowne warianty rozgrywki (czy ściana ma drzwi przez które możemy przejść itp. itd.). Wprowadź elementy BombedRoom/BombedWall (rozwiązanie co się wtedy stanie zostawiam twórcy. Może być timer, który po 15s bez decyzji zabija gracza etc.).

a.

i. Rozpocząłem od problemu „przechodzenia przez pokoje”, jako że nie miałem sposobu, na zyskanie referencji do pokoju „za tą ścianą”. Dodałem więc w klasie Door metodę, która zwróci mi „ten inny pokój”:

1.
```java
public Room otherRoom(Room room){
    if (room == room1){
        return room2;
    }
    else return room1;
}
```

ii. Stworzyłem klasę Player, która umie przemieszczać się po pokojach i przechodzić przez drzwi:

1.
```java
public class Player {

    private Room currentRoom;

    public Player(Room startingRoom){
        this.currentRoom = startingRoom;
    }

    public void move(Direction direction){
        MapSite obiekt = this.currentRoom.getSide(direction);
        obiekt.Enter();
        if(obiekt instanceof Door){
            Room nextRoom = ((Door) obiekt).otherRoom(this.currentRoom);
            nextRoom.Enter();
            this.currentRoom = nextRoom;
        }
    }
}
```

b. Zmodyfikowałem klasę MazeGame, aby wprowadzić tam gracza. Skorzystałem z wbudowanej w Java.util klasy „Scanner”, aby zczytywać wejście konsoli.

1.
```java
package pl.agh.edu.dp.labirynth;

import ...

public class MazeGame {

    private Maze maze;
    private Player player;

    private boolean gameOver = false;
    private Scanner keyboard;

    public void init(StandardMazeBuilder mazeBuilder, MazeFactory mazeFactory, Player player) throws Exception{
        this.maze = this.createMaze(mazeBuilder, mazeFactory);
        System.out.println("Created new Maze Game.");

        this.keyboard = new Scanner(System.in);
        System.out.println("Controlls: WASD - movement.");
        System.out.println("L - leave game");

        while(!gameOver){
            this.gameLoop();
        }

        System.out.println("Ended the Maze Game.");
    }
```

2.

```
private void gameLoop(){
    System.out.println(" ");    // "spacer", ku poprawie widoczności.
    char buttonPressed = keyboard.next().charAt(0);
    switch(buttonPressed){
        case 'w':
            this.player.move(Direction.North);
            break;
        case 'a':
            this.player.move(Direction.West);
            break;
        case 's':
            this.player.move(Direction.South);
            break;
        case 'd':
            this.player.move(Direction.East);
            break;
        case 'l':
            System.out.println("So you have chosen... Death.");
            this.gameOver = true;
            break;
        default:
            System.out.println("No such control. Try using WASD(movement) or L(lose the game).");
            break;
    }
}
```

c. Aby dodać funkcjonalność BombedWall/BombedRoom rozszerzyłem klasę Player o atrybut Health, który będzie się zmniejszał z wartości początkowej (100) o 10 za każdym razem, gdy gracz wejdzie w taką ścianę/pokój. Gdy życie gracza spadnie do 0 - gameOver, gra przerwana.

1.
```
public class Player {

    private Room currentRoom;
    private int health;

    public Player(Room startingRoom){
        this.currentRoom = startingRoom;
        this.health = 100;
    }

    public void move(Direction direction){
        MapSite obiekt = this.currentRoom.getSide(direction);
        obiekt.Enter();
        if(obiekt instanceof Door){
            Room nextRoom = ((Door) obiekt).otherRoom(this.currentRoom);
            nextRoom.Enter();
            this.currentRoom = nextRoom;
        }
        else if(obiekt instanceof BombedWall){
            this.getDamaged(10);
        }
        else if(obiekt instanceof BombedRoom){
            this.getDamaged(10);
        }
    }

    public void getDamaged(int damage){
        this.health -= damage;
    }

    public int getHealth(){
        return this.health;
    }
}
```

ii. I na końcu gameLoop() dodałem funkcję sprawdzającą, czy gracz aby nie umarł.

```
if (this.player.getHealth()<=0){
    this.gameOver = true;
    System.out.println("You run out of live. Game over.");
}
```

d. Zmodyfikowałem klasę main, żeby korzystała z nowych klas i metod.

1.
```
public class Main {

    public static void main(String[] args) throws Exception {

        MazeGame mazeGame = new MazeGame();
        MazeFactory mazeFactory = new MazeFactory();
        StandardMazeBuilder mazeBuilder = new StandardMazeBuilder(mazeFactory);

        mazeGame.init(mazeBuilder, mazeFactory);
    }
}
```

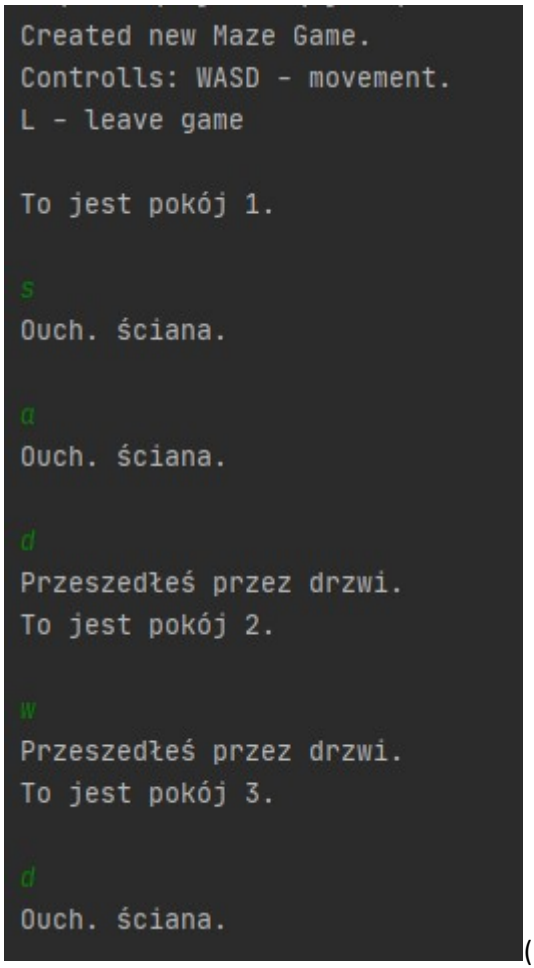
ii. Zauważyłem problem, jak owóż zwykłe „Room”, „Door” i „Wall”, w metodzie „Enter()” nie miały żadnego wypisywania „Przeszedłeś przez drzwi”, więc pewne dodałem.

iii. Przeniosłem również tworzenie „Player” do MazeGame, jako że musi mieć on „wstrzyknięty” pokój startowy.

iv. Dodałem do klasy „Maze” metodę „getFirstRoom()”.

v. „Player” teraz „wchodzi” do pokoju początkowego, a nie znajduje się w nim od razu (kwestia wiadomości gdzie się znajdujemy na początku gry).

2. Grę przetestowałem i nie znalazłem w niej błędów.



(mapa ta sama co wcześniej (z diagramu))

3.

b) Zademonstruj, że MazeFactory faktycznie jest Singletonem (najłatwiej stworzyć przykład, w którym się sprawdza, czy obiekt zwracany przy 2 konstrukcji to faktycznie ten sam, który został stworzony na początku).

a.

```java
import static org.junit.Assert.*;

public class MazeFactoryTest {

    @Test
    public void getInstance() {
        MazeFactory factory = MazeFactory.getInstance();
        MazeFactory factory2 = MazeFactory.getInstance();

        assertTrue( condition: factory == MazeFactory.getInstance());
        assertEquals(factory,MazeFactory.getInstance());
        assertEquals(factory, factory2);
    }
}
```

b.