Nazar Kordiumov

Laboratorium 3

Wzorce Projektowe

Własne zmiany:

1. Klasa Room.
    a. Dodałem prywatne pole count, które odpowiada za nadawanie Id pokojom. Użytkownik nie musi przekazywać Id poprzez konstruktor.
    b. Dodałem prywatną metodę fillSides(), która przyjmuje Supplier, dla tworzenia ścian pokoju. Teraz użytkownik nie musi za każdym razem tworzenia pokoju, ręcznie tworzyć każdą ze ścian.

```java
public class Room extends MapSite {
    private static final AtomicInteger count = new AtomicInteger( initialValue: 0);
    private final int roomId;
    private Map<Direction, MapSite> sides;

    public Room(Supplier<Wall> wallSupplier) {
        this.sides = new EnumMap<>(Direction.class);
        fillSides(wallSupplier);
        this.roomId = count.incrementAndGet();
    }

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

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

    public int getRoomId() { return this.roomId; }

    @Override
    public void enter() { System.out.println("You came to the room"); }

    private void fillSides(Supplier<Wall> wallSupplier) {
        this.setSide(Direction.North, wallSupplier.get());
        this.setSide(Direction.East, wallSupplier.get());
        this.setSide(Direction.South, wallSupplier.get());
        this.setSide(Direction.West, wallSupplier.get());
    }
}
```

4.1 Builder

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

Dzięki metodzie reset() możemy użyć tego samego Buildera, to tworzenia kilku labiryntów po kolei.

```java
public interface MazeBuilder {
    void reset();
    void addRoom(Room room);
    void attachWall(Wall wall, Direction direction, Room r1, Room r2);
    void attachDoor(Door door);
}
```

2. Po utworzeniu powyższego interfejsu zmodyfikuj funkcje składową tak, aby przyjmowała jako parametr obiekt tej klasy.
3. Prześledź i zinterpretuj co dały obecne zmiany (krótko opisz swoje spostrzeżenia).

```
public class MazeGame {

    public Maze createMaze(){
        Maze maze = new Maze();

        Room r1 = new Room(Wall::new);
        Room r2 = new Room(Wall::new);

        Door door = new Door(r1, r2);

        maze.addRoom(r1);
        maze.addRoom(r2);

        return maze;
    }
```

```
public void createMaze(MazeBuilder builder) {
    Room A = new Room(Wall::new);
    Room B = new Room(Wall::new);
    Room C = new Room(Wall::new);
    builder.addRoom(A);
    builder.addRoom(B);
    builder.addRoom(C);
    builder.attachWall(new Wall(), Direction.West, A, B);
    builder.attachDoor(new Door(A, B));
    builder.attachWall(new Wall(), Direction.North, B, C);
    builder.attachDoor(new Door(B, C));
}
```

Teraz to klasa Builder tworzy labirynt. Użytkownik nie ma do czynienia z procesem tworzenia labiryntu, jedynie przekazuje do Builder'a potrzebne do tego składowe, wywołując odpowiednie metody.

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.

```
private Direction getCommonWallDirection(Room r1, Room r2) {
    //loops for finding wall (directions) that joins rooms
    Direction[] directionList = Direction.values();
    for (Direction currentDirection : directionList) {
        Direction oppositeDirection = Direction.getOppositeTo(currentDirection);
        if(r1.getSide(currentDirection).equals(r2.getSide(oppositeDirection))) {
            return currentDirection;
        }
    }
    throw new IllegalArgumentException("Cannot make door [no common wall] between Room1 " +
            "(id: " + r1.getRoomId() + ") i Room2 (id: " + r2.getRoomId() + ")");
}
```

Metoda getCommonWallDirection(Room r1, Room r2) zwraca kierunek (dla peirwszego pokoju) wspólnej ściany pomiędzy dwoma pomieszczeniami. Jeżeli takiej ściany nie ma, rzuca wyjątek.

```java
public class StandardBuilderMaze implements MazeBuilder {

    private Maze currentMaze;

    public StandardBuilderMaze() { this.reset(); }

    @Override
    public void reset() { this.currentMaze = new Maze(); }

    @Override
    public void addRoom(Room room) { currentMaze.addRoom(room); }

    @Override
    public void attachWall(Wall joiningWall, Direction direction, Room r1, Room r2) {
        r1.setSide(direction, joiningWall);
        r2.setSide(Direction.getOppositeTo(direction), joiningWall);
    }

    @Override
    public void attachDoor(Door door) {
        Room r1 = door.getRoom1();
        Room r2 = door.getRoom2();
        Direction doorDirection = getCommonWallDirection(r1, r2);
        r1.setSide(doorDirection, door);
        r2.setSide(Direction.getOppositeTo(doorDirection), door);
    }

    public Maze getResultedMaze() {
        Maze product = currentMaze;
        this.reset();
        return product;
    }
```

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

Przykład tworzenia labiryntu za pomocą metody createMaze oraz stworzonych Builde'ów.

```java
StandardBuilderMaze standardBuilderMaze = new StandardBuilderMaze();
mazeGame.createMaze(standardBuilderMaze);
System.out.println(standardBuilderMaze.getResultedMaze().getRoomNumbers());


CountingMazeBuilder countingMazeBuilder = new CountingMazeBuilder();
mazeGame.createMaze(countingMazeBuilder);
System.out.println(countingMazeBuilder.getCount());
```

```
3
Rooms=3, Walls=8, Doors=2

Process finished with exit code 0
```

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

Stworzyłem pomocniczą klasę Counters, która zawiera w sobie 3 liczniki odpowiednio dla: Pokojów, Ścian i Drzwi.

```java
public class Counters {
    private int roomCounter;
    private int wallCounter;
    private int doorCounter;

    public Counters() {
        this.roomCounter = 0;
        this.wallCounter = 0;
        this.doorCounter = 0;
    }

    @Override
    public String toString() {
        return "Rooms=" + roomCounter +
                ", Walls=" + wallCounter +
                ", Doors=" + doorCounter;
    }
```

```java
public class CountingMazeBuilder implements MazeBuilder {

    private Counters counters;

    public CountingMazeBuilder() { this.reset(); }

    @Override
    public void reset() { counters = new Counters(); }

    @Override
    public void addRoom(Room room) {
        counters.setRoomCounter(counters.getRoomCounter() + 1);
        counters.setWallCounter(counters.getWallCounter() + 4);
    }

    @Override
    public void attachWall(Wall wall, Direction direction, Room r1, Room r2) {
        counters.setWallCounter(counters.getWallCounter() - 1);
    }

    @Override
    public void attachDoor(Door door) {
        counters.setWallCounter(counters.getWallCounter() - 1);
        counters.setDoorCounter(counters.getDoorCounter() + 1);
    }

    public Counters getCount() {
        Counters result = counters;
        this.reset();
        return result;
    }
}
```

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.

```
public interface MazeFactory {
    Room createRoom();
    Wall createWall();
    Door createDoor(Room r1, Room r2);
}
```

Zamiast klasy stworzyłem interfejs MazeFactory tak jak to jest pokazane -> https://refactoring.guru/design-patterns/abstract-factory.

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

```
public void createMaze(MazeBuilder builder, MazeFactory factory) {
    Room A = factory.createRoom();
    Room B = factory.createRoom();
    Room C = factory.createRoom();
    builder.addRoom(A);
    builder.addRoom(B);
    builder.addRoom(C);
    builder.attachWall(factory.createWall(), Direction.West, A, B);
    builder.attachDoor(factory.createDoor(A, B));
    builder.attachWall(factory.createWall(), Direction.North, B, C);
    builder.attachDoor(factory.createDoor(B, C));
}
```

Teraz użytkownik nie tworzy sam składowych labiryntu, lecz tworzone są za pomocą interfejsu MazeFactory. Użytkownik nie musi wiedzieć jak konkretna fabryka jest zaimplementowana oraz jakiego rodzaju element otrzyma, ponieważ korzysta z interfejsu, który zwraca klasy nadrzędne elementów.

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).

```
public class EnchantedRoom extends Room {

    public EnchantedRoom(Supplier<Wall> wallSupplier) {
        super(wallSupplier);
    }

    @Override
    public void enter() {
        System.out.println("You entered the enchanted room");
    }
}
```

Dodałem klasy pomocnicze nadpisując metodę enter() w każdej z klas: EnchantedRoom/Wall/Door oraz BombedRoom/Wall.

```
public class EnchantedMazeFactory implements MazeFactory {

    @Override
    public Room createRoom() {
        return new EnchantedRoom(EnchantedWall::new);
    }

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

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

EnchantedMazeFactory zwraca poszczególne elementy z rodziny Enchanted.

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(...)).

```
public class BombedMazeFactory implements MazeFactory {

    @Override
    public Room createRoom() { return new BombedRoom(BombedWall::new); }

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

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

BombedMazeFactory zwraca poszczególne elementy z rodziny Bombed (oprócz drzwi).

4.3 Singleton

Ponieważ przy tworzeniu Abstract Factory korzystałem z https://refactoring.guru/design-patterns/abstract-factory to stowrzyłem interfejs MazeFactory, zamiast klasy, przez co nie da się stworzyć jego instancji. Zatem klasy EnchantedMazeFactory oraz BombedMazeFactory zrobiłem Singletonem.

```
private static EnchantedMazeFactory instance;

private EnchantedMazeFactory() {
    try {
        Thread.sleep( millis: 1000);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}

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

```
private static BombedMazeFactory instance;

private BombedMazeFactory() {
    try {
        Thread.sleep( millis: 1000);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}

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

Ponieważ konkretne fabryki powinne być tworzone z pozycji kodu, który jest odpowiedzialny za tworzenie poszczególnych części labiryntu, czyli MazeGame, stworzyłem dodatkowe dwie metody: createEnchantedMaze oraz createBombedMaze. Tworzą one odpowiednią, konkretną fabrykę za pomocą mechanizmu Singleton następnie przekazują ją wraz z Builderem do metody createMaze. Ta metoda tworzy labirynt za pomocą Buildera (interfejsu MazeBuilder), a poszczególne części labiryntu za pomocą Fabryki (interfejsu MazeFactory). W tej sytuacja klasa MazeGame zachowuje prawie jak klasa Director ze wzorca Builder.

```
public void createEnchantedMaze(MazeBuilder builder) {
    createMaze(builder, EnchantedMazeFactory.getInstance());
}

public void createBombedMaze(MazeBuilder builder) {
    createMaze(builder, BombedMazeFactory.getInstance());
}

private void createMaze(MazeBuilder builder, MazeFactory factory) {
    Room A = factory.createRoom();
    Room B = factory.createRoom();
    Room C = factory.createRoom();
    builder.addRoom(A);
    builder.addRoom(B);
    builder.addRoom(C);
    builder.attachWall(factory.createWall(), Direction.West, A, B);
    builder.attachDoor(factory.createDoor(A, B));
    builder.attachWall(factory.createWall(), Direction.North, B, C);
    builder.attachDoor(factory.createDoor(B, C));
}
```

4.3 Rozszerzenie aplikacji labirynt

a) Dodanie klasy Player i możliwości grania

1. *Najpierw dodałem do wszystkich klas metodę toString(), aby można było wypisać w konsoli cały labirynt w przystępny sposób.*

```
------------------HELP------------------
a <- left | d -> right | w - up | s - down | q - exit
----------------------------------------
Maze rooms:[
BombedRoom{1, {North=BombedWall, South=BombedWall, East=BombedWall, West=Door{from: 1, to: 2}},
BombedRoom{2, {North=Door{from: 2, to: 3}, South=BombedWall, East=Door{from: 1, to: 2}, West=BombedWall},
BombedRoom{3, {North=BombedWall, South=Door{from: 2, to: 3}, East=BombedWall, West=BombedWall}]
Player: health=100, currentRoom=1
```

2. *Dodałem klasę Player z polami health oraz currentRoom. Dodałem również metody umożliwiające poruszanie się po labiryncie, które z pola currentRoom pobierają odpowiednią MapSite (w zależności od kierunku) i wywołują metodę enter. W każdej klasie dziedziczącej po MapSite, zmieniłem metodę enter tak, aby przyjmowała Player i robiła odpowiednią do tego elementu labiryntu czynność. Np. uderzenie w ścianę zabiera zdrowie, a drzwi zmieniają pole currentRoom w Playerze.*

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

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

public class Player {
    private int health;
    private Room currentRoom;

    public Player(int health) { this.health = health; }

    public void receiveDamage(int i) {
        health -= i;
        if(health > 0) {
            System.out.println("You received " + i + " damage");
        }
    }

    public void moveLeft() {
        currentRoom.getSide(Direction.East).enter( player: this);
    }

    public void moveRight() {
        currentRoom.getSide(Direction.West).enter( player: this);
    }

    public void moveForward() {
        currentRoom.getSide(Direction.North).enter( player: this);
    }

    public void moveBackward() {
        currentRoom.getSide(Direction.South).enter( player: this);
    }

    @Override
    public String toString() {
        return "Player: " +
                "health=" + health +
                ", currentRoom=" + currentRoom.getRoomId();
    }
```

3. *Następnie stworzyłem klasę MazeGame (wcześniejszą przemianowałem na MazeDirector) z polami Player oraz Maze, która obsługuje wciśnięte przyciski.*

```java
private Maze maze;
private Player player;

public MazeGame(Maze maze, Player player) {
    this.maze = maze;
    player.setCurrentRoom(maze.getStartRoom());
    this.player = player;
}

private void beforeStart() {
    System.out.println("------------------HELP------------------");
    System.out.println("a <- left | d -> right | w - up | s - down | q - exit");
    System.out.println("----------------------------------------");
    System.out.println(maze);
    System.out.println(player);
}

public void startGame() throws IOException {
    beforeStart();
    char input = 0;
    while(input != 'q') {
        input = (char) System.in.read();
        switch (movePlayer(input)) {
            case -1 : {
                System.out.println("You died");
                System.out.println(GAME_OVER);
                return;
            }
            case 1 : {
                System.out.println(WON);
                return;
            }
            case 0 : {
                System.out.println(maze);
            }
        }
    }
}
```

```java
public int movePlayer(int x) {
    switch (x) {
        case 'a' : {
            player.moveLeft();
            break;
        }
        case 'd' : {
            player.moveRight();
            break;
        }
        case 'w' : {
            player.moveForward();
            break;
        }
        case 's' : {
            player.moveBackward();
            break;
        }
        default: {
            return -2;
        }
    }
    return checkAfterMovement();
}

private int checkAfterMovement() {
    if(player.getHealth() < 0) {
        return -1;
    }
    if(player.getCurrentRoom().equals(maze.getEndRoom())) {
        return 1;
    }
    System.out.println(player);
    return 0;
}
```

4. *Ostatecznie w klasie Main wywołuje metodę startGame*

```
public class Main {

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

        MazeDirector mazeDirector = new MazeDirector();
        StandardBuilderMaze standardBuilderMaze = new StandardBuilderMaze();
        mazeDirector.createBombedMaze(standardBuilderMaze);

        Maze maze = standardBuilderMaze.getResultedMaze();
        Player player = new Player( health: 100);
        MazeGame mazeGame = new MazeGame(maze, player);
        mazeGame.startGame();
    }
}
```

b) Zademonstrowanie, że Singletony działają poprawnie.

Stworzyłem dodatkową metodę checkSingletons w klasie MazeDirector

```
public void checkSingletons() {
    MazeFactory mazeFactory1 = EnchantedMazeFactory.getInstance();
    MazeFactory mazeFactory2 = EnchantedMazeFactory.getInstance();
    if(mazeFactory1.equals(mazeFactory2)) {
        System.out.println("The same objects!");
    }

    MazeFactory mazeFactory3 = BombedMazeFactory.getInstance();
    MazeFactory mazeFactory4 = BombedMazeFactory.getInstance();
    if(mazeFactory3.equals(mazeFactory4)) {
        System.out.println("The same objects!");
    }
}
```

Jej wyłowanie w Mainie zwraca:

```
The same objects!
The same objects!
------------------HELP------------------
a <- left | d -> right | w - up | s - down | q - exit
----------------------------------------
```