Raport z zadań dotyczących wzorców projektowych

Autor: Krzysztof Hardek

Zad 4.1 Builder

Wprowadzone zmiany

Interfejs MazeBuilder

```
public interface MazeBuilder {
    public abstract void reset();

    public abstract boolean buildDoor(int roomNo1, int roomNo2);

    public abstract int buildRoom();

    public abstract boolean buildWall(int roomNo1, int roomNo2, Direction
dirRoom1);
}
```

Stworzyłem interfejs buildera na potrzeby użycia wzorca projektowego. Posiada ona standardową metode reset() resetującą stan aktualnie budowanego obiektu oraz trzy metody wynikające z tego jak wyglądają labirynty. Dotychczas wprowadzone zmiany pozwoliły rozdzielić proces tworzenia kompleksowego obiektu jakim jest labirynt na poszczególne sekcje. Dzięki temu unikniemy ogromnego konstruktora który mógłby być postaci "new Maze(Array doorArray, Array roomArray, Array wallArray)" oraz w innym scenariuszu tworzenia drzewa klas. Przy trzech elementach labiryntu może to nie jest problem ale więcej elementów mogłoby komplikować sprawę.

Klasa StandardBuilderMaze

```
public class StandardBuilderMaze implements MazeBuilder {
    private Maze currentMaze;

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

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

    @Override
    public boolean buildDoor(int roomNo1, int roomNo2){
        Room room1 = currentMaze.getRoom(roomNo1);
```

```
        Room room2 = currentMaze.getRoom(roomNo2);

        Direction commonDir = this.commonWall(room1, room2);

        if(commonDir != null){
            Door newDoor = new Door(room1, room2);
            room1.setSide(commonDir, newDoor);
            room2.setSide(Direction.opposite(commonDir), newDoor);
            return true;
        }

        return false;
    }

    @Override
    public int buildRoom(){
        int newRoomNumber = this.currentMaze.getRoomNumbers();
        Room newRoom = new Room(newRoomNumber);

        currentMaze.addRoom(newRoom);

        return newRoomNumber;
    }

    @Override
    public boolean buildWall(int roomNo1, int roomNo2, Direction dirRoom1){
        // use roomNo2 = -1 to build wall only in room1
        Room room1 = currentMaze.getRoom(roomNo1);
        Room room2 = currentMaze.getRoom(roomNo2);
        Wall wall = new Wall();

        if(room1 != null){
            room1.setSide(dirRoom1, wall);
        }

        if(room2 != null){
            room2.setSide(Direction.opposite(dirRoom1), wall);
        }

        return room1 != null || room2 != null;
    }

    private Direction commonWall(Room room1, Room room2){
        Direction commonDir = null;

        for(Direction dir: Direction.values()){
            if(room1.getSide(dir) ==
room2.getSide(Direction.opposite(dir))){
                commonDir = dir;
            }
        }

        return commonDir;
    }
```

```
    public Maze getMaze(){
        Maze maze = this.currentMaze;
        this.reset();
        return maze;
    }
}
```

Standardowa implementacja metod interfejsu. Oprócz tego pojawia się metoda znajdująca wspólny kierunek dla dwóch pokoi oraz metoda zwracająca budowany labirynt.

Klasa MazeGame

```
public class MazeGame {
    public Maze createMaze(StandardBuilderMaze builder){
        boolean buildWallSuccess = true;
        boolean buildDoorSuccess = true;

        builder.buildRoom(); // 0
        builder.buildRoom(); // 1
        builder.buildRoom(); // 2


        // building non-common walls
        for(Direction dir: Direction.values()){
            buildWallSuccess &= builder.buildWall(0 , -1, dir);
            buildWallSuccess &= builder.buildWall(1 , -1, dir);
            buildWallSuccess &= builder.buildWall(2 , -1, dir);
        }

        // Overwriting common walls
        buildWallSuccess &= builder.buildWall(1 , 2, Direction.East);
        buildWallSuccess &= builder.buildWall(0 , 1, Direction.North);

        if(!buildWallSuccess) System.out.println("Wall building problems");

        buildDoorSuccess &= builder.buildDoor(0, 1);
        buildDoorSuccess &= builder.buildDoor(1, 2);

        if(!buildDoorSuccess) System.out.println("Door building problems");

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

Klasa CountingMazeBuilder

```
public class CountingMazeBuilder implements MazeBuilder{
    int wallCount;
```

```
    int doorCount;
    int roomCount;

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

    @Override
    public void reset(){
        this.wallCount = 0;
        this.doorCount = 0;
        this.roomCount = 0;
    }

    @Override
    public int buildRoom(){
        this.roomCount++;

        return -1;
    }

    @Override
    public boolean buildDoor(int roomNo1, int roomNo2){
        this.wallCount -= 2;
        this.doorCount += 1;
        return true;
    }

    @Override
    public boolean buildWall(int roomNo1, int roomNo2, Direction dir){
        if(roomNo1 != -1) wallCount++;
        if(roomNo2 != -1) wallCount++;

        return true;
    }

    public int getWallCount(){
        return wallCount;
    }

    public int getDoorCount(){
        return doorCount;
    }

    public int getRoomCount(){
        return roomCount;
    }

}
```

Metody występujące w tej klasie przy wywołaniu metody budującej dany element wliczają go do liczby poszczególnych elementów.

Klasa Maze

```
import java.util.Vector;

public class Maze {
    private Vector<Room> rooms;

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

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

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

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

    public Room getRoom(int roomNo){
        for(Room room : rooms){
            if(room.getRoomNumber() == roomNo){
                return room;
            }
        }
        return null;
    }
}
```

Dodałem metode getRoom zwracającą pokój o zadanym numerze (zakładam że numer pokoju jest unikatowy). Potrzebuje tego w StandardBuilderMaze.

Enum Direction

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

    public static Direction opposite(Direction dir){
        switch(dir){
            case North:
                return South;
            case South:
                return North;
            case East:
                return West;
```

```
            case West:
                return East;
            default:
                return null;
        }
    }
}
```

Dodałem metode statyczną opposite, która zwraca przeciwny kierunek. Przydaje sie przy pracy ze skojarzonymi pokojami.

Zad 4.2 Fabryka Abstrakcyjna

Wyspecjalizowane klasy reprezentujące konkretne ściany, drzwi czy pokoje póki co nie różnią sie od swojej klasy nadrzędnej. Dodatkowe funkcjonalności będą dodane wraz z rozwojem gry

Klasa MazeFactory

```
public class MazeFactory {
    public Room makeRoom(int roomNo){
        return new Room(roomNo);
    }

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

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

Główna fabryka abstrakcji. Klasa ta stanowi podstawę dla bardziej wyspecjalizowanych. Można za jej pomocą tworzyć zwykłe ściany, drzwi oraz pokoje

Klasa EnchantedMazeFactory

```
public class EnchantedMazeFactory extends MazeFactory{
    @Override
    public Room makeRoom(int roomNo){
        return new EnchantedRoom(roomNo);
    }

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

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

Konkretna fabryka dla zaczarowanych obiektów.

Klasa EnchantedRoom

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

Klasa EnchantedWall

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

Klasa EnchantedDoor

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

Klasa BombedMazeFactory

```
public class BombedMazeFactory extends MazeFactory{
    @Override
    public Room makeRoom(int roomNo){
        return new BombedRoom(roomNo);
    }

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

```
}
```

Konkretna fabryka dla wybychowych obiektow

Klasa BombedRoom

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

Klasa BombedWall

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

Klasa MazeGame - fragment

```
public class MazeGame {
    public Maze createMaze(StandardBuilderMaze builder, MazeFactory
mazeFactory){
        boolean buildWallSuccess = true;
        boolean buildDoorSuccess = true;

        builder.setFactory(mazeFactory);
```

Do wywołania createMaze dodałem obiekt klasy MazeFactory. Ustawiam go w builderze za pocą setera.

Klasa StandardBuilderMaze - fragmenty

```
public void setFactory(MazeFactory factory){ this.factory = factory; }
```

```
public boolean buildWall(int roomNo1, int roomNo2, Direction dirRoom1){
        // use roomNo2 = -1 to build wall only in room1
        Room room1 = currentMaze.getRoom(roomNo1);
        Room room2 = currentMaze.getRoom(roomNo2);
        Wall wall = this.factory.makeWall();//new Wall();
```

```
        if(room1 != null){
            room1.setSide(dirRoom1, wall);
        }
```

```
    public int buildRoom(){
        int newRoomNumber = this.currentMaze.getRoomNumbers();
        Room newRoom = this.factory.makeRoom(newRoomNumber);//new
Room(newRoomNumber);

        currentMaze.addRoom(newRoom);

        return newRoomNumber;
    }
```

```
    public boolean buildDoor(int roomNo1, int roomNo2){
        Room room1 = currentMaze.getRoom(roomNo1);
        Room room2 = currentMaze.getRoom(roomNo2);

        Direction commonDir = this.commonWall(room1, room2);

        if(commonDir != null){
            Door newDoor = this.factory.makeDoor(room1, room2);//new
Door(room1, room2);
```

```
    private MazeFactory factory;

    public StandardBuilderMaze(MazeFactory factory){
        this.factory = factory;
        this.reset();
    }
```

Do tej klasy dodałem setera dla obiektu MazeFactory, drugi konstruktor oraz zmieniłem wszystkie operacje tworzenia nowych obiektów na te z fabryki abstrakcji.

Zad 4.3 Singleton

Klasa MazeFactory - fragment

```
public class MazeFactory {
    private static MazeFactory instance;

    public static MazeFactory getInstance(){
        if(instance == null){
```

```
            instance = new MazeFactory();          // singleton mechanism
        }
        return instance;
    }
```

Jeżeli instancja istnieje to ją zwracam. Jeżeli nie to tworze nową. Mechanizm ten zapewnia istnienie tylko jednej instancji w jednym momencie.

Zad 4.4 Rozszerzenie aplikacji

a) Mechanizm przemieszczania się

Klasa Main

```
public class Main {

    public static void main(String[] args) {

        MazeGame mazeGame = new MazeGame();

        Maze maze = mazeGame.createMaze(new StandardBuilderMaze(), new
BombedMazeFactory());
        Player player = new Player(50, 0, maze);

        System.out.println(mazeGame);
        mazeGame.gameLoop(player);

    }
}
```

Klasa MazeGame - nowe metody

```
    public void gameLoop(Player player){
        Scanner input = new Scanner(System.in);
        int move;
        boolean playingFlag = true;

        while(playingFlag){
            move = input.nextInt();

            switch(move){
                case 0:
                    player.move(Direction.West);
                    break;
                case 1:
                    player.move(Direction.North);
                    break;
                case 2:
```

```
                    player.move(Direction.East);
                    break;
                case 3:
                    player.move(Direction.South);
                    break;
                case -1:
                    playingFlag = false;
                    break;
                default:
                    continue;
            }

            if(player.getHp() <= 0){
                System.out.println("You died, game is closing");
                playingFlag = false;
            }
        }
    }

    @Override
    public String toString(){
        return "Use numbers to move, 0 - east, 1-north, 2-west, 3-south, -1
to leave game";
    }
```

Nadpisałem metode toString, żeby wypisywać zasady działania gry oraz stworzyłem metodę z pętlą gry.

Klasa player

```
public class Player {
    private Room currentRoom;
    private int hp;
    private Maze currentMaze;

    public Player(int basicHp, int startingRoomNo, Maze currentMaze){
        this.hp = basicHp;
        this.currentMaze = currentMaze;
        this.currentRoom = this.currentMaze.getRoom(startingRoomNo);
    }

    public void move(Direction dir){
        MapSite side = currentRoom.getSide(dir);

        if(side instanceof Wall){
            side.Enter(this);
        }
        else if(side instanceof Door){
            side.Enter(this);
        }
        else if(side instanceof Room){
            side.Enter(this);
```

```
        }
        else if(side == null){
            System.out.println("Side undefined");
        }
        else{
            System.out.println("Unknown side");
        }
    }

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

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

    public void setHp(int hp) {
        this.hp = hp;
    }

    public int getHp(){
        return this.hp;
    }

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

Reprezentuje gracza oraz związane z nim rzeczy, takie jak punkty życia czy jego labirynt.

Klasa Mapsite

```
public abstract class MapSite {
    public abstract void Enter(Player player);
}
```

Dodałem to metody enter parametr klasy Player, aby móc manipulować jego statystykami, gdy wejdzie on do odpowiedniego obiektu

Klasa BombedRoom - metoda enter

```
    @Override
    public void Enter(Player player){
        player.setCurrentRoom(this);
        System.out.printf("Current room: %d\n WALLS DO DMG\n",
```

```
this.getRoomNumber());
    }
```

Implementuje zachowanie przy wejściu do pokoju

Klasa BombedWall

```
    @Override
    public void Enter(Player player){
        player.setHp(player.getHp()-5);
        System.out.printf("You lost 5hp on Wall, Current hp: %d\n",
player.getHp());
    }
```

Implementuje zachowanie przy wejściu w ściane.

Klasa Door

```
    @Override
    public void Enter(Player player){
        if(player.getCurrentRoom() == room1){
            room2.Enter(player);
        }
        else if(player.getCurrentRoom() == room2){
            room2.Enter(player);
        }
        else {
            System.out.println("Player got lost");
        }
    }
```

Implementuje zachowanie przy wejściu "do drzwi". Gdy wchodzę "do drzwi" również wchodzę do pokoju, więc wywołuje metodę enter dla pokoju.

b) Sprawdzenie, czy singleton działa

Klasa Main

```
    MazeFactory f1 = BombedMazeFactory.getInstance();
    MazeFactory f2 = BombedMazeFactory.getInstance();

    System.out.println(f1 == f2);
```

Wprowadziłem taki kod w metodzie main. Program zwrócił prawdę, więc f1 oraz f2 wskazują na dokładnie ten sam obiekt.