Paweł Kiełbasa

Sprawozdanie do Lab 3 – Wzorce obiektowe

1. Builder
   1. Stworzenie interfejsu MazeBuilder, w którym zawarte są metody dodawania obiektów w labiryncie

public interface MazeBuilder {  
 void addRoom(Room room);  
 void addDoor(Room r1, Room r2) throws Exception;  
 void addCommonWall(Direction roomDirection, Room r1, Room r2) throws Exception;  
}

* 1. Zmodyfikowanie klasy MazeGame, tak aby metoda createMaze przyjmowała jako argument MazeBuilder

public void createMaze(MazeBuilder builder) throws Exception {  
 …  
}

* 1. Dzięki powyższym zmianom umożliwione zostało przeniesienie metod tworzących obiekty w labiryncie do innej klasy. Zaowocuje to zmniejszeniem ilości powtarzanego kodu.
  2. Stworzenie klasy StandardMazeBuilder która implementuje interfejs MazeBuilder.

|  |  |  |
| --- | --- | --- |
| Nazwa metody | Zwracany typ | Co robi |
| addRoom | void | Dodaje pokój z czterema ścinami |
| addDoor | void | Dodaje drzwi między dwoma pokojami |
| addCommonWall | void | Dodaje wspólną ścianę między dwoma pokojami |
| getCurrentMaze | Maze | Zwarca stan labiryntu |

public class StandardMazeBuilder implements MazeBuilder {  
 private Maze currentMaze;  
  
 public StandardMazeBuilder(MazeFactory factory) {  
 this.currentMaze = new Maze();  
 }  
  
 @Override  
 public void addRoom(Room room) {  
 room.setSide(*South*, new Wall());  
 room.setSide(*North*, new Wall());  
 room.setSide(*East*, new Wall());  
 room.setSide(*West*, new Wall());  
 currentMaze.addRoom(room);  
 }  
  
 @Override  
 public void addDoor(Room r1, Room r2) throws Exception {  
 Direction r1Door = null;  
 for (Direction dir : *values*()) {  
 if (r1.getSide(dir).equals(r2.getSide(dir.getOppositeSide()))) {  
 r1Door = dir;  
 break;  
 }  
 }  
 if (r1Door == null) throw new Exception("Rooms don't have common door");  
 else {  
 Door newDoor = new Door(r1, r2);  
 r1.setSide(r1Door, newDoor);  
 r2.setSide(r1Door.getOppositeSide(), newDoor);  
 }  
 }  
  
 @Override  
 public void addCommonWall(Direction r1Direction, Room r1, Room r2) throws Exception {  
 MapSite side = r1.getSide(r1Direction);  
 if (side == null) throw new Exception("Room doesn't exist");  
 r2.setSide(r1Direction.getOppositeSide(), side);  
 }  
  
 public Maze getCurrentMaze() {  
 return this.currentMaze;  
 }  
}

Aby otrzymać powyższą funkcjonalność w enumie Direction stworzono metodą getOppositeSide, która zwraca przeciwny kierunek

public Direction getOppositeSide() {  
 return evaluateDirection((this.value + 2) % 4);  
}

* 1. Stworzenie nowego labiryntu

public static void main(String[] args) throws Exception {  
 MazeGame mazeGame = new MazeGame();  
 StandardMazeBuilder builder = new StandardMazeBuilder();  
 Maze maze= mazeGame.createMaze((StandardMazeBuilder) builder);  
 System.out.println(maze.getRoomNumbers());  
}

* 1. Dodanie klasy CountingMazeBuilder, która liczy ilość obiektów w labiryncie. Implementuje ona interfejs MazeBuilder

public class CountingMazeBuilder implements MazeBuilder {  
 private int elementsNumber = 0;  
  
 @Override  
 public void addRoom(Room room) {  
 elementsNumber += 5;  
  
 }  
  
 @Override  
 public void addDoor(Room r1, Room r2) throws Exception {  
 elementsNumber++;  
 }  
  
 @Override  
 public void addCommonWall(Direction roomDirection, Room r1, Room r2) throws Exception {  
 elementsNumber--;  
 }  
  
 int GetCounts() {  
 return elementsNumber;  
 }  
}

1. Fabryka abstrakcyjna
   1. Stworzenie klasy MazeFactory

|  |  |  |
| --- | --- | --- |
| Nazwa metody | Zwracany typ | Co robi |
| createDoor | Door | Tworzy nowe drzwi między dwoma pokojami |
| createRoom | Room | Tworzy nowy pokój |
| createWall | Wall | Tworzy nową ścianę |

public class MazeFactory {  
  
 public Door createDoor(Room r1, Room r2) {  
 return new Door(r1, r2);  
 }  
  
 public Room createRoom(int number) {  
 return new Room(number);  
 }  
  
 public Wall createWall() {  
 return new Wall();  
 }  
}

* 1. Wykorzystanie MazeFactory w metodzie createMaze

public void createMaze(StandardMazeBuilder builder, MazeFactory factory) throws Exception {  
…  
}

* 1. Stworzenie klasy EnchantedMazeFactory, w której tworzone są klasy EnchantedRoom, EnchantedDoor i EnchantedWall

public class EnchantedMazeFactory extends MazeFactory {  
  
 @Override  
 public Door createDoor(Room r1, Room r2) {  
 return new EnchantedDoor(r1, r2);  
 }  
  
 @Override  
 public Room createRoom(int number) {  
 return new EnchantedRoom(number);  
 }  
  
 @Override  
 public Wall createWall() {  
 return new EnchantedWall();  
 }  
}

Aby klasa działała poprawnie konieczne jest stworzenie klas EnchantedDoor, EnchantedRoom i EnchantedWall

public class EnchantedDoor extends Door {  
  
 public EnchantedDoor(Room r1, Room r2) {  
 super(r1, r2);  
 }  
  
 @Override  
 public void Enter() {  
 System.*out*.println("You opened enchanted door");  
 }  
}

public class EnchantedRoom extends Room {  
 public EnchantedRoom(int number) {  
 super(number);  
 }  
  
 @Override  
 public void Enter() {  
 System.*out*.println("Entered enchanted room");  
 }  
}

public class EnchantedWall extends Wall {  
 public EnchantedWall(){  
 super();  
 }  
  
  
 @Override  
 public void Enter() {  
 System.*out*.println("Entered enchanted room");  
 }  
}

* 1. Stworzenie klasy BombedMazeFactory z metodami createRoom i createWall (tworzą wybuchające ściany/drzwi)

public class BombedMazeFactory extends MazeFactory {  
 @Override  
 public Room createRoom(int number) {  
 return new BombedRoom(number);  
 }  
  
 @Override  
 public Wall createWall() {  
 return new BombedWall();  
 }  
}

W tym celu stworzono klasy BombedWall, BombedRoom i BombedDoor

public class BombedWall extends Wall{  
 public BombedWall(){  
 super();  
 }  
 @Override  
 public void Enter() {  
 System.*out*.println("Entered bombed wall");  
 }  
}

public class BombedRoom extends Room {  
 public BombedRoom(int number) {  
 super(number);  
 }  
  
 @Override  
 public void Enter() {  
 System.*out*.println("Entered bombed room");  
 }  
}

public class BombedDoor extends Door {  
 public BombedDoor(Room r1, Room r2) {  
 super(r1, r2);  
 }  
  
 @Override  
 public void Enter() {  
 System.*out*.println("You opened bombed door");  
 }  
}

1. Singleton – aby MazeFactory było singletonem należało zmodyfikować klasę MazeFactory i wszystkie klasy dziedziczące po niej
   1. Klasa MazeFactory –dodanie metody getInstance zwracającej instacje klasy

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

* 1. Klasa EnchantedMazeFactory – dodanie metody getInstance zwracjającej instacje klasy

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

* 1. Klasa BombedMazeFactory – dodanie metody getInstance zwracjającej instacje klasy

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

* 1. Zmiana w klasie StandardMazeBuilder – dodanie atrybutu factory i generowanie ścian, drzwi za pomocą metod dostępnych w klasie MazeFactory

public class StandardMazeBuilder implements MazeBuilder {  
 private Maze currentMaze;  
 private MazeFactory factory;  
  
 public StandardMazeBuilder(MazeFactory factory) {  
 this.currentMaze = new Maze();  
 this.factory = factory;  
 }  
  
 @Override  
 public void addRoom(Room room) {  
 room.setSide(*South*, factory.createWall());  
 room.setSide(*North*, factory.createWall());  
 room.setSide(*East*, factory.createWall());  
 room.setSide(*West*, factory.createWall());  
 currentMaze.addRoom(room);  
 }  
  
 @Override  
 public void addDoor(Room r1, Room r2) throws Exception {  
 Direction r1Door = null;  
 for (Direction dir : *values*()) {  
 if (r1.getSide(dir).equals(r2.getSide(dir.getOppositeSide()))) {  
 r1Door = dir;  
 break;  
 }  
 }  
 if (r1Door == null) throw new Exception("Seems like rooms don't have common door");  
 else {  
 Door newDoor = factory.createDoor(r1, r2);  
 r1.setSide(r1Door, newDoor);  
 r2.setSide(r1Door.getOppositeSide(), newDoor);  
 }  
 }  
  
 @Override  
 public void addCommonWall(Direction r1Direction, Room r1, Room r2) throws Exception {  
 MapSite side = r1.getSide(r1Direction);  
 if (side == null) throw new Exception("Seems like such a room doesn't exist");  
 r2.setSide(r1Direction.getOppositeSide(), side);  
 }  
  
 public Maze getCurrentMaze() {  
 return this.currentMaze;  
 }  
}

1. Rozszerzenie aplikacji labirynt
   1. Dodanie klasy Player z możliwością poruszania się po labiryncie z polem currentRoom
2. public class Player {  
     
    private Room currentRoom;  
     
    public Player(Room currentRoom) {  
    this.currentRoom = currentRoom;  
    }  
     
    public void move(Direction dir) {  
    MapSite side = this.currentRoom.getSide(dir);  
    side.Enter();  
    if (side instanceof Door) {  
    this.currentRoom = ((Door) side).getRoomAtOthersSide(currentRoom);  
    }  
    }  
     
    public void status() {  
    System.*out*.println("Room number: " + this.currentRoom.getRoomNumber());  
    for (Direction dir : Direction.*values*()) {  
    if (this.currentRoom.getSide(dir) instanceof Door) {  
    System.*out*.println(dir + " door");  
    } if (this.currentRoom.getSide(dir) instanceof Wall) {  
    System.*out*.println(dir + " wall");  
    } if (this.currentRoom.getSide(dir) instanceof Room) {  
    System.*out*.println(dir + " Room");  
    }  
    }  
    }  
     
    public void setCurrentRoom(Room currentRoom) {  
    this.currentRoom = currentRoom;  
    }  
     
    public Room getCurrentRoom() {  
    return this.currentRoom;  
    }  
   }

W celu pełnej funkcjonalności klasy do klasy Door dodano metodę getRoomAtOthersSide zwracającą pokój po drugiej stronie drzwi

public Room getRoomAtOthersSide(Room firstR) {  
 return room1 == firstR ? room2 : room1;  
}

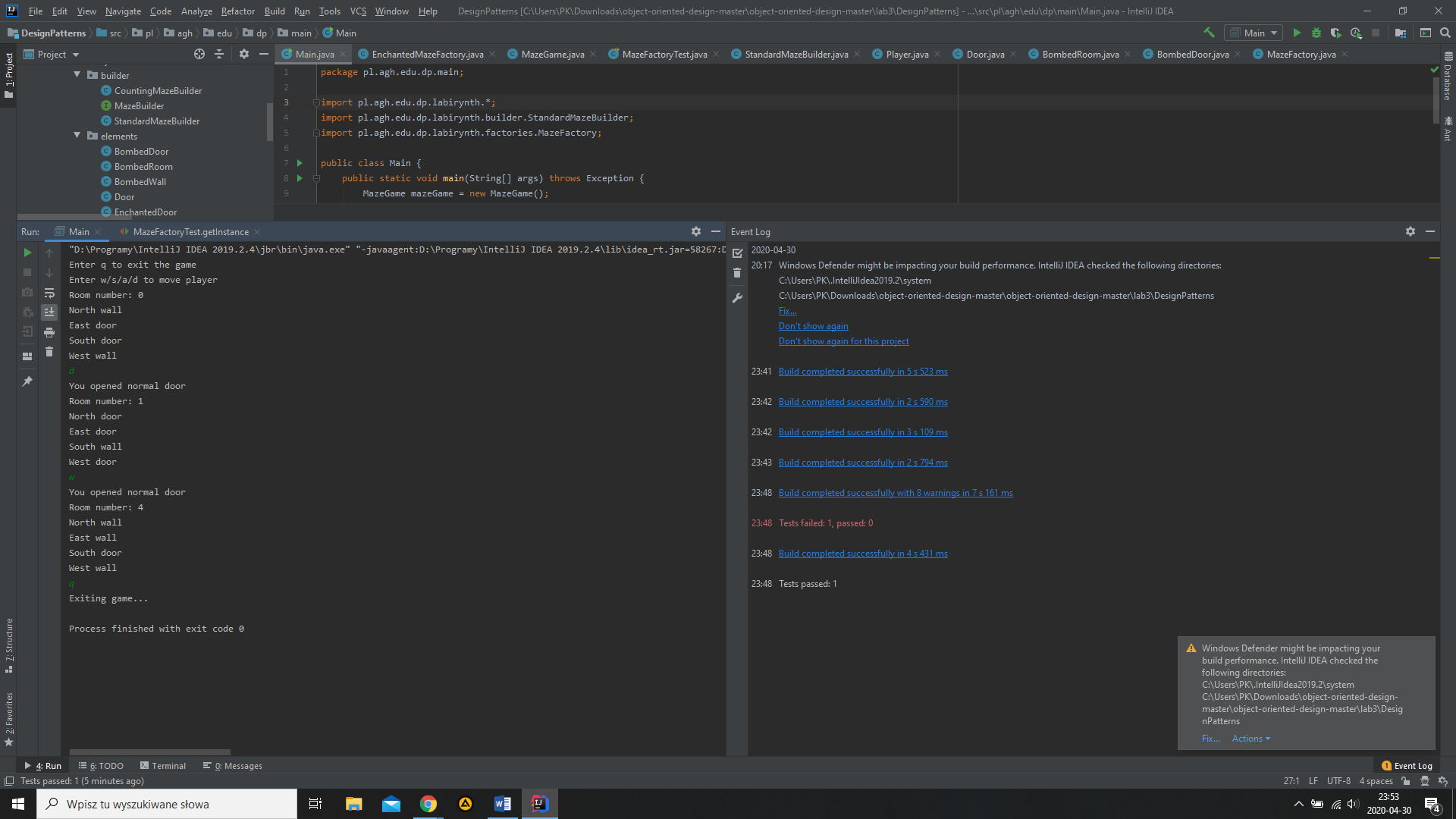
4.2. Modyfikacja klasy MazeGame – dodanie labiryntu wraz z pokojami

public class MazeGame {  
  
 private Player player;  
  
 private static MazeGame *instance*;  
  
 public static MazeGame getInstance() {  
 if (*instance* == null) {  
 *instance* = new MazeGame();  
 }  
 return *instance*;  
 }  
  
 public void start() {  
 System.*out*.println("Enter q to exit the game");  
 System.*out*.println("Enter w/s/a/d to move player");  
 loop();  
 }  
  
 private void loop() {  
 Scanner scan = new Scanner(System.*in*);  
 while (true) {  
 player.status();  
 char c = scan.next().charAt(0);  
 switch (c) {  
 case 'w':  
 this.player.move(*North*);  
 break;  
 case 's':  
 this.player.move(*South*);  
 break;  
 case 'a':  
 this.player.move(*West*);  
 break;  
 case 'd':  
 this.player.move(*East*);  
 break;  
 case 'q':  
 stop();  
 return;  
 default:  
 System.*out*.println("Unknown command: " + c);  
 break;  
 }  
 }  
 }  
  
 public void stop() {  
 System.*out*.println("Exiting game...");  
 }  
  
 public void createMaze(StandardMazeBuilder builder, MazeFactory factory) throws Exception {  
 this.player = new Player(buildExampleMaze(builder, factory));  
 Maze maze = builder.getCurrentMaze();  
 }  
  
 private Room buildExampleMaze(StandardMazeBuilder builder, MazeFactory factory) throws Exception {  
 Room[] rooms = new Room[9];  
 for (int i = 0; i < 5; i++) {  
 rooms[i] = factory.createRoom(i);  
 builder.addRoom(rooms[i]);  
 }  
  
 builder.addCommonWall(Direction.*East*, rooms[0], rooms[1]);  
 builder.addCommonWall(Direction.*South*, rooms[0], rooms[3]);  
 builder.addCommonWall(Direction.*East*, rooms[1], rooms[2]);  
 builder.addCommonWall(Direction.*North*, rooms[1], rooms[4]);  
 builder.addCommonWall(Direction.*East*, rooms[3], rooms[4]);  
  
 builder.addDoor(rooms[0], rooms[1]);  
 builder.addDoor(rooms[1], rooms[2]);  
 builder.addDoor(rooms[0],rooms[3]);  
 builder.addDoor(rooms[1],rooms[4]);  
  
 return rooms[0];  
 }  
}

4.3. Modyfikacja klasy Main tak aby gra działała

public static void main(String[] args) throws Exception {  
 MazeGame mazeGame = new MazeGame();  
 MazeFactory mazeFactory = MazeFactory.*getInstance*();  
 StandardMazeBuilder builder = new StandardMazeBuilder(mazeFactory);  
 mazeGame.createMaze(builder, mazeFactory);  
 mazeGame.start();  
}

Wynik działania programu



4.4. Sprawdzenie czy MazeFactory jest singletonem

public class MazeFactoryTest {  
 @Test  
 public void getInstance() {  
 MazeFactory factory = MazeFactory.*getInstance*();  
  
 *assertEquals*(factory, MazeFactory.*getInstance*());  
 *assertEquals*(factory, MazeFactory.*getInstance*());  
  
 MazeFactory factory2 = MazeFactory.*getInstance*();  
  
 *assertEquals*(factory2, MazeFactory.*getInstance*());  
 *assertEquals*(factory2, MazeFactory.*getInstance*());  
  
 *assertEquals*(factory, factory2);  
 }  
}

Wynik wywołania testu

