TECHNISCHE UNIVERSITÄT DRESDEN

Software Technology Group

Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

WS2019/20 – Design Patterns and Frameworks

Creational Patterns

Professor: Prof. Dr. Uwe Aßmann
Lectuer: Dr.-Ing. Sebastian Götz
Tutor: Dr. rer. nat. Marvin Triebel

Task 1 Amazing Creation

This exercise focuses on *Creational Patterns* as outlined in [1]. You are designing a maze-based computer game (**MazeGame**). The game is based on a system of rectangular rooms. Every room (`Room`) has 4 walls, either with a door to a neighboring room (`DoorWall`) or without a door (`Wall`). The map of the whole system (`Maze`) consists of these three element types. Figure 1 sketches the corresponding class diagram. Conversely, Listings 1 show an excerpt of the implementation of the `Maze` and Listing 2 the construction of a `MazeGame`.

Listing 1: Java implementation of the Maze

```
1  public class Maze {
2      private Map<Integer, Room> rooms = new HashMap<Integer, Room>();
3      public void addRoom (Room r) { rooms.put (r.getRoomNo(), r); }
4      public Room roomNo (int r) { return rooms.get (r); }
5  }
6
7  public enum Direction { NORTH, EAST, SOUTH, WEST }
8  public class Room {
9      private Map<Direction, Wall> sides = new HashMap<Direction, Wall>();
10     private int roomNo;
11     public Room(int roomNo) {
12         this.roomNo = roomNo; }
13     public Wall getSide (Direction direction) {
14         return sides.get(direction); }
15     public void setSide(Direction direction, Wall wall) {
16         sides.put(direction, wall); }
17     /*...*/
18 }
19 public class Wall { /* ... */ }
20 public class DoorWall extends Wall {
21     private Room r1;
22     private Room r2;
23     private boolean isOpen;
24     public DoorWall (Room r1, Room r2) {
25         this.r1 = r1;   this.r2 = r2;   this.isOpen = false;
26     }
27 }
```

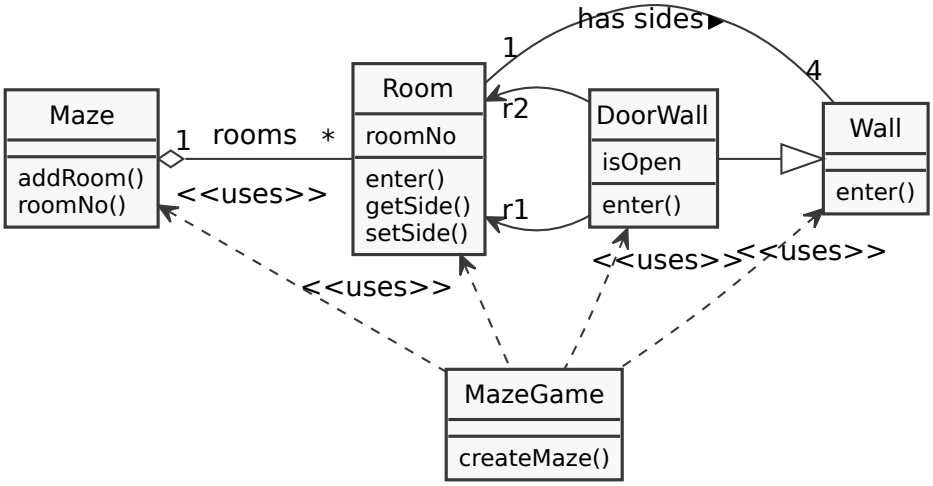

Figure 1: Class diagram of the MazeGame.

Listing 2: Construction of a Maze in the MazeGame

```
1  public class MazeGame {
2      public static void main(String[] argv) { createMaze(); }
3      private static Maze createMaze() {
4          Maze aMaze = new Maze();
5          Room r1 = new Room (1);
6          Room r2 = new Room (2);
7          DoorWall d = new DoorWall (r1, r2);
8          aMaze.addRoom(r1);
9          aMaze.addRoom(r2);
10         r1.setSide(Direction.NORTH, d)
11         r1.setSide(Direction.EAST, new Wall());
12         r1.setSide(Direction.SOUTH, new Wall());
13         r1.setSide(Direction.WEST, new Wall());
14         r2.setSide(Direction.NORTH, new Wall());
15         r2.setSide(Direction.EAST, new Wall());
16         r2.setSide(Direction.SOUTH, d);
17         r2.setSide(Direction.WEST, new Wall());
18         return aMaze;
19     }
20 }
```

Develop another game that uses the same plan of rooms. However, instead of simple rooms, use **magic rooms** (containing booby traps that can only be survived if you know a certain spell), and instead of simple doors, use doors that can be opened only with a spell. **Spells** work, such that invoking a spell brings it into effect for the room in which the player is located and for a certain amount of time, after which the effect *wears off*. If, during this time, the player attempts to pass an enchanted door and if the spell invoked is the spell required for the door, the player can pass the door. Otherwise, the player cannot pass the door.

Since the construction of rooms (`createMaze`) is complex, do not duplicate the code. Change the above design, such that the new program can create both the old and the new game.

a) Use the design pattern `AbstractFactory` to achieve the desired flexibility. Draw a modified class diagram and realize the implementation.

b) Alternatively, use the pattern `FactoryMethod` to achieve the desired flexibility. Draw a modified class diagram and realize the implementation.

c) Now, look back to your design from (a). You have programmed 2 different factories. Change the design to have only one concrete factory. To retain the same flexibility employ the `Prototype` pattern.

d) Compare and evaluate your solutions of (a)–(c). Which solution is best for which context?

Task 2 Building Mazes

In the previous task, we have looked at transparently and flexibly exchanging different implementation classes for specific concepts. Creating a maze is still very complex, though. In particular, it is easy to make mistakes because structural rules are not enforced in any way.

a) To remedy this, use the `Builder` pattern to hide the complexity of the creation of the rooms (`createMaze`). Design a builder for the original game and also for the extension. Draw a modified UML class diagram and realize the implementation.

b) How does the `Builder` enforces structural rules?

Task 3 Creation in Parallel Hierarchies

Consider, you work for a company designing an UML figure editor. The model under development can be looked at from different views. These views may be different diagrams showing different parts of the model, but they may also vary in the options for manipulation they offer. Most importantly, there will be read-only views and mutable views.

To this end, a class model for *class diagrams*, *statecharts*, and *activity diagrams* should be developed. It should contain class hierarchies for some of the diagram elements: *classes* and their *inheritance* arrows, *states* and their *state transition* arrows, *activities* and their *activity transition* arrows.

An example of such an editor is *ArgoUML*.[1] It is rather complex, but handles some of the same issues treated in this exercise.

[1] http://argouml.tigris.org

a) What design pattern is useful for this type of application, where users need different views on their data?

b) Design the required class hierarchies: Use separate class hierarchies for the *read-only* and the *mutable* facets of a view. How are these *facets* linked? Which constraint would you like to enforce for the classes in these hierarchies?

c) Now, in a second step, extend the *read-only* hierarchies with factory methods for the writeable diagrams and diagram elements. Sketch the implementations of the factory methods.

d) Now, refactor your design towards an `AbstractFactory` for manipulators. Discuss the advantages and disadvantages of this approach.

the following exercise.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[2] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.