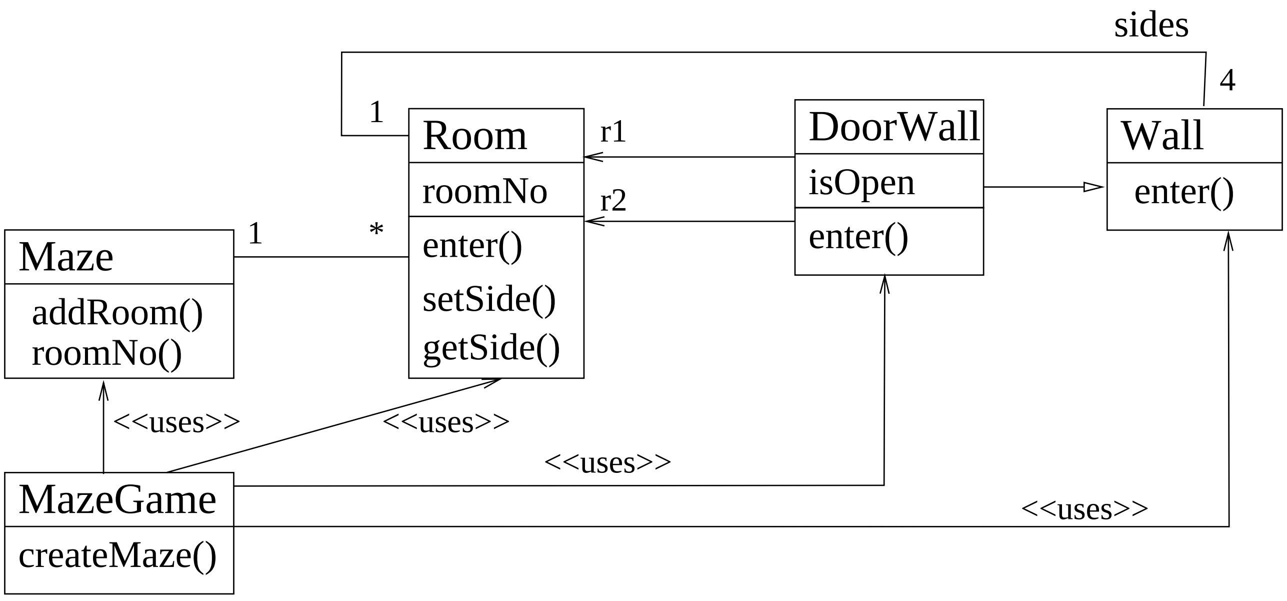
Design Patterns and Frameworks	**Exercise Sheet No. 2**
Dipl.-Inf. Steffen Zschaler	Software Engineering Group
INF 2097	Institute for Software and Multimedia Technology
`http://st.inf.tu-dresden.de/teaching/dpf`	Department of Computer Science
	Technische Universität Dresden
	01062 Dresden

Variability Patterns II: Creational Patterns

Task 1: Amazing Creation

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 neighbouring room (`DoorWall`) or without a door (`Wall`). The map of the whole system (`Maze`) consists of these three element types.



Here is an excerpt of the core parts of the corresponding Java-Code:

```
public interface Directions {
  static final int north = 0, east = 1, south = 2, west  = 3;
}

// ------------------------------------------------------------

public class Maze {
  private Map mpRooms = new HashMap();
  public void addRoom (Room r) { mpRooms.put (r.getRoomNo(), r); }
  public Room roomNo (int r) { return (Room) mpRooms.get (r); }
}

// ------------------------------------------------------------
```

```
public class Room {
  private Wall[] sides = new Wall[4];
  private int roomNo;

  public Room(int roomNo) {
    this.roomNo = roomNo;
  }

  public Wall getSide (int direction) { return sides[i]; }

  public void setSide(int direction, Wall ms) {
    sides[direction] = ms;
  }

  //...
}

// -----------------------------------------------------------

public class DoorWall extends Wall {
  private Room r1;
  private Room r2;
  private boolean isOpen;

  public DoorWall (Room r1, Room r2) {
    this.r1 = r1;
    this.r2 = r2;
    this.isOpen = false;
  }
}

// -----------------------------------------------------------

public class Wall { /* ... */
}

// -----------------------------------------------------------

public class MazeGame implements /* uses */ Directions {
  public static void main(String[] argv) {
    createMaze();
  }

  private static Maze createMaze() {
    Maze aMaze = new Maze();
    Room r1 = new Room (1);
    Room r2 = new Room (2);
    DoorWall d  = new DoorWall (r1, r2);

    aMaze.addRoom(r1);
    aMaze.addRoom(r2);

    r1.setSide(north, new Wall());
    r1.setSide(east,  d);
    r1.setSide(south, new Wall());
    r1.setSide(west,  new Wall());

    r2.setSide(north, new Wall());
    r2.setSide(east,  new Wall());
    r2.setSide(south, new Wall());
    r2.setSide(west,  d);

    return aMaze;
  }
}
```

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 like this: 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 be used for the old and the new game.

1a)

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

1b)

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

1c)

Now, look back to your design from (a). You have programmed 2 concrete factories, which were subclasses of the abstract factory. Change the design to have only one concrete subclass, using the Prototype pattern.

1d)

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.

2a)

Now, use the pattern Builder 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. How does Builder enforce structural rules?

Task 3: Creation in Parallel Hierarchies

You are working for a company designing a 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 can be found at http://argouml.tigris.org . It is rather complex, but handles some of the same issues treated in this exercise.

$\boxed{\text{3a)}}$ What design pattern is useful for this type of application where users need different views on their data?

$\boxed{\text{3b)}}$ 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?

$\boxed{\text{3c)}}$ 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.

$\boxed{\text{3d)}}$ Now, refactor your design towards an AbstractFactory for manipulators. Discuss the advantages and disadvantages of this approach.

Task 4: Interpretative Factory*

This is a task of additional complexity—and benefit! We may not discuss it in the exercise, but if you come up with a solution and send it to me, I will be happy to give feedback on your solution.

$\boxed{\text{4a)}}$

Design an interpretative factory for GUI widgets. The factory uses two string parameters to determine the class to instantiate: One parameter is kept as an instance variable and indicates the 'theme' of the GUI, the other parameter indicates the widget to be created. Widget class names follow a naming pattern of 'widgets.'<ThemeName>'.'<WidgetName>'.class'. Use reflection to instantiate the widgets.