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

```java
public class Maze {
    private Map<Integer, Room> rooms = new HashMap<Integer, Room>();
    public void addRoom (Room r) { rooms.put (r.getRoomNo(), r); }
    public Room roomNo (int r) { return rooms.get (r); }
}

public enum Direction { NORTH, EAST, SOUTH, WEST }
public class Room {
    private Map<Direction, Wall> sides = new HashMap<Direction, Wall>();
    private int roomNo;
    public Room(int roomNo) {
        this.roomNo = roomNo; }
    public Wall getSide (Direction direction) {
        return sides.get(direction); }
    public void setSide(Direction direction, Wall wall) {
        sides.put(direction, wall); }
    /*...*/
}
public class Wall { /* ... */ }
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;
    }
}
```

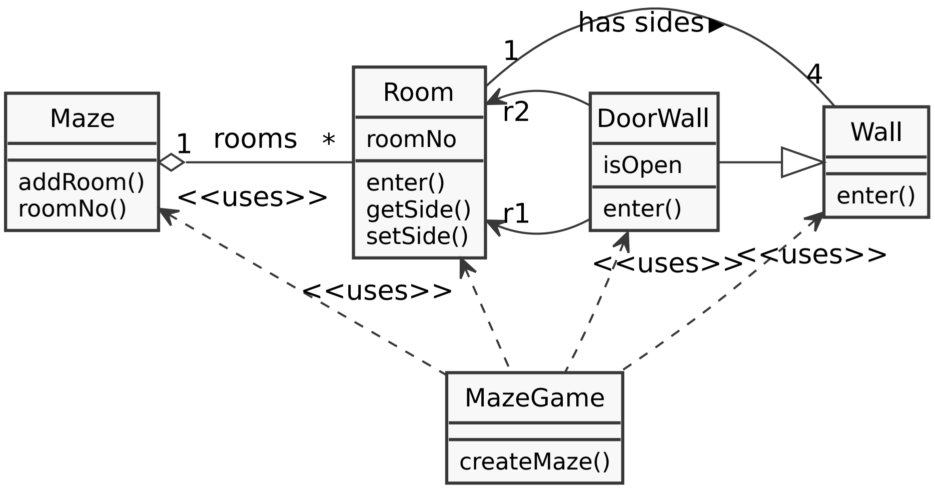


Figure 1: Class diagram of the MazeGame.

Listing 2: Construction of a Maze in the MazeGame

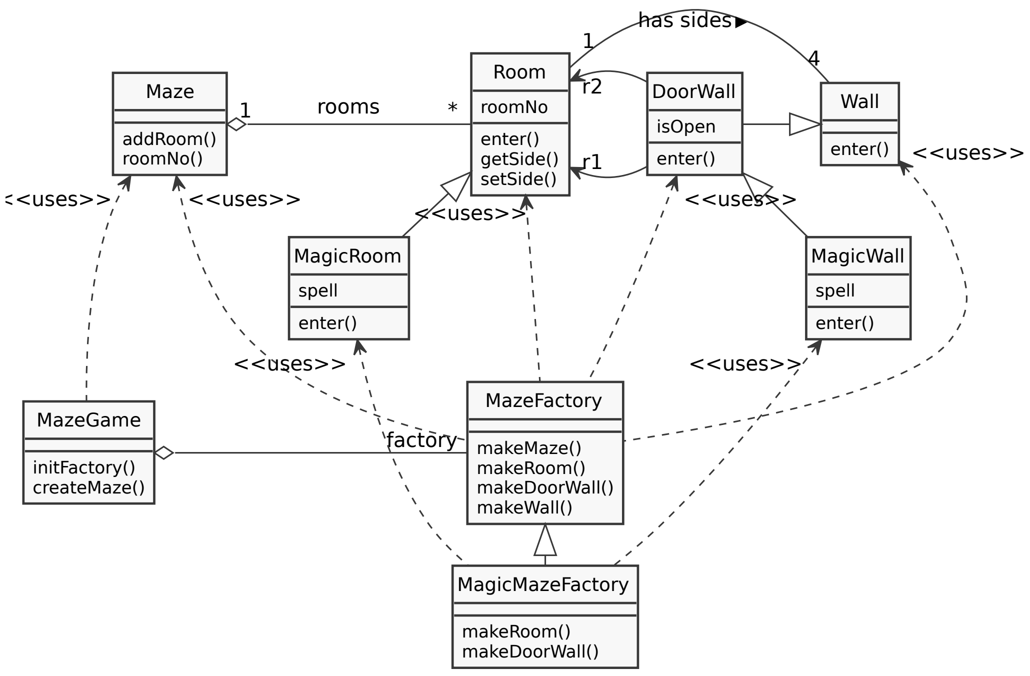
```java
public class MazeGame {
    public static void main(String[] argv) { createMaze(); }
    private static Maze createMaze() {
        Maze aMaze = new Maze();
        Room r1 = new Room (1);
        Room r2 = new Room (2);
        DoorWall d = new DoorWall (r1, r2);
        aMaze.addRoom(r1);
        aMaze.addRoom(r2);
        r1.setSide(Direction.NORTH, d)
        r1.setSide(Direction.EAST, new Wall());
        r1.setSide(Direction.SOUTH, new Wall());
        r1.setSide(Direction.WEST, new Wall());
        r2.setSide(Direction.NORTH, new Wall());
        r2.setSide(Direction.EAST, new Wall());
        r2.setSide(Direction.SOUTH, d);
        r2.setSide(Direction.WEST, new Wall());
        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, 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.

Solution: The following class diagram and listings illustrates the design of the `AbstractFactory`.



Listing 3: Implementation of the AbstractFactory design pattern.

```java
public class MazeFactory {
  public Maze makeMaze(){ return new Maze(); }
  public Maze makeRoom(int n){ return new Room(n); }
  public Maze makeWall(){ return new Wall(); }
  public Maze makeDoorWall(Room r1, Room r2){
    return new DoorWall(r1,r2);
  }
}
public class MagicMazeFactory extends MazeFactory {
  private String pickSpell(){ /*...*/ }
  public Maze makeRoom(int n){
    return new MagicRoom(n,pickSpell());
  }
  public Maze makeDoorWall(Room r1, Room r2){
    return new MagicDoorWall(r1,r2,pickSpell());
  }
}
```

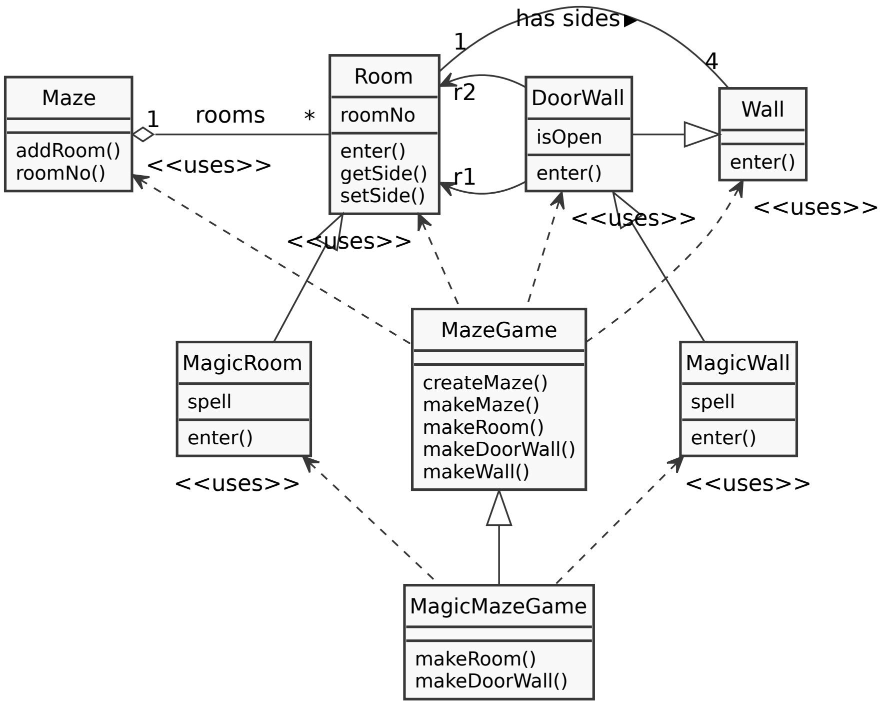
The effects of this design on the `createMaze` method is illustrated below.

Listing 4: Construction of a Maze using the AbstractFactory.

```
public class MazeGame {
    public static void main(String[] argv) { initFactory();
        createMaze(); }
    private static MazeFactory factory;
    private static void initFactory(){
        if (newversion)
          factory=new MagicMazeFactory();
        else
          factory=new MazeFactory();
    }
    private static Maze createMaze() {
        Maze aMaze = factory.makeMaze();
        Room r1 = factory.makeRoom(1);
        Room r2 = factory.makeRoom(2);
        DoorWall d = factory.makeDoorWall(r1, r2);
        aMaze.addRoom(r1);
        aMaze.addRoom(r2);
        r1.setSide(Direction.NORTH, d)
        r1.setSide(Direction.EAST, factory.makeWall());
        r1.setSide(Direction.SOUTH, factory.makeWall());
        r1.setSide(Direction.WEST, factory.makeWall());
        r2.setSide(Direction.NORTH, factory.makeWall());
        r2.setSide(Direction.EAST, factory.makeWall());
        r2.setSide(Direction.SOUTH, d);
        r2.setSide(Direction.WEST, factory.makeWall());
        return aMaze;
    }
}
```

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

Solution: The following class diagram and listings illustrates the `FactoryMethod`.



Listing 5: Implementation of the `FactoryMethod` design pattern.

```java
public class MazeGame {
    public static void main(String[] argv) { new
        MazeGame().createMaze(); }
  protected Maze createMaze() {/*...*/}
  protected Maze makeMaze(){ return new Maze(); }
  protected Maze makeRoom(int n){ return new Room(n); }
  protected Maze makeWall(){ return new Wall(); }
  protected Maze makeDoorWall(Room r1, Room r2){
    return new DoorWall(r1,r2);
  }
}
public class MagicMazeGame extends MazeGame {
    public static void main(String[] argv) { new
        MagicMazeGame().createMaze(); }
  private String pickSpell(){ /*...*/ }
  public Maze makeRoom(int n){
    return new MagicRoom(n,pickSpell());
  }
  public Maze makeDoorWall(Room r1, Room r2){
    return new MagicDoorWall(r1,r2,pickSpell());
  }
}
```

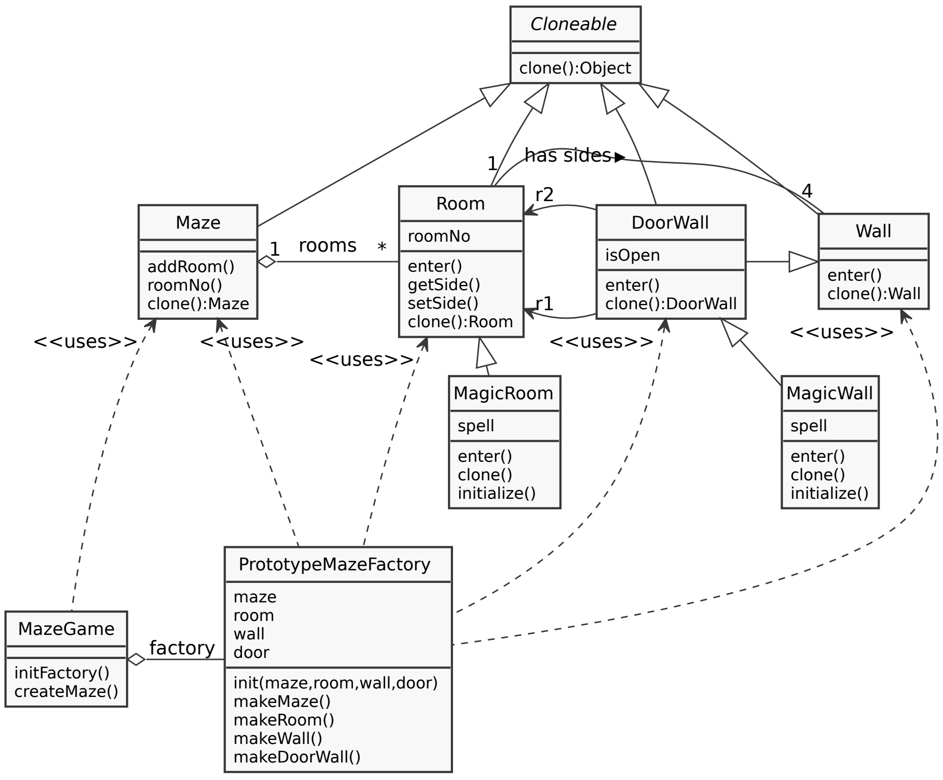
The effects of this design on the `createMaze` method is illustrated below.

Listing 6: Construction of a Maze using the `FactoryMethod`.

```java
private static Maze createMaze() {
    Maze aMaze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    DoorWall d = makeDoorWall(r1, r2);
    aMaze.addRoom(r1);
    aMaze.addRoom(r2);
    r1.setSide(Direction.NORTH, d)
    r1.setSide(Direction.EAST, makeWall());
    r1.setSide(Direction.SOUTH, makeWall());
    r1.setSide(Direction.WEST, makeWall());
    r2.setSide(Direction.NORTH, makeWall());
    r2.setSide(Direction.EAST, makeWall());
    r2.setSide(Direction.SOUTH, d);
    r2.setSide(Direction.WEST, makeWall());
    return aMaze;
}
```

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.

Solution: The following class diagram and listings illustrates the `Prototype` pattern for factories.



Listing 7: Implementation of the `Prototype` design pattern.

```java
public class PrototypeMazeFactory {
  private Maze m;  private Room r;
  private Wall w;  private DoorWall dw;

  public init(Maze m, Room r, Wall w, DoorWall dw){
   this.m=m; this.r=r; this.w=w; this.dw=dw;
  }
  public Maze makeMaze(){ return m.clone(); }
  public Maze makeRoom(int n){
    Room room=r.clone();
    room.setNumber(n);
    return room;
  }
  public Maze makeWall(){ return w.clone(); }
  public Maze makeDoorWall(Room r1, Room r2){
    DoorWall door=dw.clone();
    door.setR1(r1);
    door.setR2(r2);
    return door;
  }
}
```

The effects of this design on the `createMaze` method is illustrated below.

Listing 8: Construction of a Maze using the Prototype factory.

```java
public class MazeGame {
    public static void main(String[] argv) { initFactory();
        createMaze(); }
    private static PrototypeMazeFactory factory=new
        PrototypeMazeFactory();
    private static void initFactory(){
        if (newversion)
          factory.init(new Maze(), new MagicRoom(0),
                       new Wall(), new MagicDoorWall(null,null));
        else
          factory.init(new Maze(), new Room(0),
                       new Wall(), new DoorWall(null,null));
    }
    private static Maze createMaze() {
        Maze aMaze = factory.makeMaze();
        Room r1 = factory.makeRoom(1);
        Room r2 = factory.makeRoom(2);
        DoorWall d = factory.makeDoorWall(r1, r2);
        aMaze.addRoom(r1);
        aMaze.addRoom(r2);
        r1.setSide(Direction.NORTH, d)
        r1.setSide(Direction.EAST, factory.makeWall());
        r1.setSide(Direction.SOUTH, factory.makeWall());
        r1.setSide(Direction.WEST, factory.makeWall());
        r2.setSide(Direction.NORTH, factory.makeWall());
        r2.setSide(Direction.EAST, factory.makeWall());
        r2.setSide(Direction.SOUTH, d);
        r2.setSide(Direction.WEST, factory.makeWall());
        return aMaze;
    }
}
```

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

   Solution: Employing `Prototypes` does not pay off, if the prototype objects need a lot of memory, because this would mean wasting of otherwise unused space. Usually, this is negligible, but software for embedded systems is an example where such issues become highly relevant. Also, the construction of a new object with the standard allocator should be compared to copying. There may be differences: use profiling to find out about it. Moreover, for prototypes, methods for cloning and initializing must be programmed, which can be particularly difficult for deep copies of complex objects.

   The `AbstractFactory` pays off in comparison to `FactoryMethods`, if there are several users of the set of factory methods, as then the factory can be published to its users. Moreover, in contrast to `FactoryMethods`, `AbstractFactories` permit changing the factory at runtime, allowing for switching from the old version of the game to the new version.

   If you solely want to allow for changing the kinds of objects that are constructed in a given class or simply defer the decision to a later point in time, the simplest solution would be to employ `FactoryMethods`.

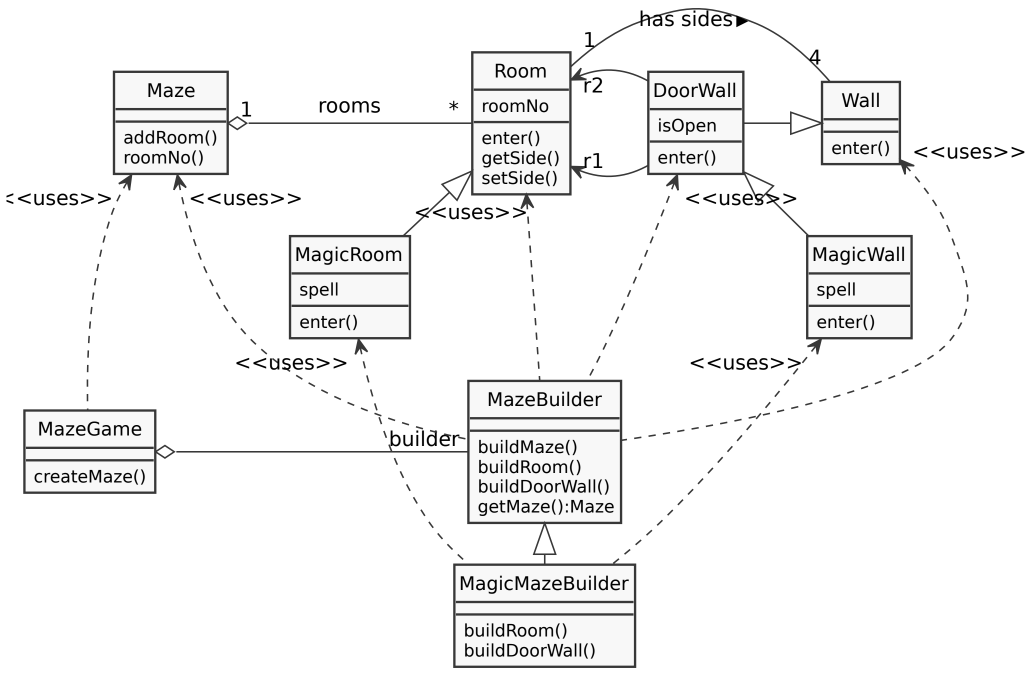
Task 2 Building Mazes

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.

   Solution: The following class diagram and listings illustrates the `Builder` pattern to simplify the creation of the Maze.

   Note: The design could be improved if it utilizes `FactoryMethods` for creating the objects inside the builder.



has sides ▸
1
4
Maze
addRoom()
roomNo()
1
rooms
*
Room
roomNo
enter()
getSide()
setSide()
r2
r1
DoorWall
isOpen
enter()
Wall
enter()
<<uses>>
<<uses>>
<<uses>>
<<uses>>
<<uses>>
<<uses>>
<<uses>>
MagicRoom
spell
enter()
MagicWall
spell
enter()
MazeGame
createMaze()
builder
MazeBuilder
buildMaze()
buildRoom()
buildDoorWall()
getMaze():Maze
MagicMazeBuilder
buildRoom()
buildDoorWall()

Listing 9: Implementation of the `Builder` design pattern.

```java
public class MazeBuilder {
  private Maze maze;

  public getMaze(){ return maze; }
  public void buildMaze(){
    maze=new Maze();
  }
  public void buildRoom(int n){
    if (maze.roomNo(n)==null){
      Room r=new Room(n);
      maze.addRoom(r);
            r.setSide(Direction.NORTH, new Wall());
            r.setSide(Direction.EAST, new Wall());
            r.setSide(Direction.SOUTH, new Wall());
            r.setSide(Direction.WEST, new Wall());
      }
  }
  private static Direction oposite(Direction d){
    switch(d){
        case Direction.NORTH: return Direction.SOUTH;
        case Direction.SOUTH: return Direction.NORTH;
        case Direction.EAST:  return Direction.WEST;
        case Direction.WEST:  return Direction.EAST;
    }
  }
  public void buildDoor(int nr1,Direction d, int nr2){
    Room r1=maze.roomNo(nr1);
    Room r2=maze.roomNo(nr2);
    if (r1!=null && r2!=null){
      DoorWall dw = new DoorWall (r1, r2);
      r1.setSide(d, dw);
      r2.setSide(oposite(d), dw);
    }
  }
}
public class MagicMazeBuilder extends MazeBuilder {
  private String pickSpell(){ /*...*/ }
  public void buildRoom(int n){
    if (maze.roomNo(n)==null){
        Room r=new MagicRoom(n,pickSpell());
        /*...*/
    }
  }
  public void buildDoor(int nr1,Direction d, int nr2){
    /*...*/
    if (r1!=null && r2!=null){
      DoorWall dw=new MagicDoorWall(r1,r2,pickSpell());
      /*...*/
    }
  }
}
```

The effects of this design on the `createMaze` method is illustrated below.

Listing 10: Construction of a maze using the Builder.

```java
public class MazeGame {
    public static void main(String[] argv) { initBuilder();
        createMaze(); }
    private static MazeBuilder builder;
    private static void initBuilder(){
        if (newversion)
          builder=new MagicMazeBuilder();
        else
          builder=new MazeBuilder();
    }
    private static Maze createMaze() {
        builder.buildMaze();
        builder.buildRoom(1);
        builder.buildRoom(2);
        builder.buildDoor(1,Direction.NORTH,2);
        return builder.getMaze();
    }
}
```

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

Solution: The `Builder` encapsulates the current product (`Maze`) and allows for modifying it only through specifically provided operations in its interface. Internally the builder can check and verify logical and structural constraints.

Listing 11: Structural constraints.

```java
public void buildDoor(int nr1,Direction d, int nr2){
  Room r1=maze.roomNo(nr1);
  if (r1==null)
    throw new IllegalArgumentException();
  if (r1.getSide(d) instanceof DoorWall)
    throw new IllegalArgumentException(
      "Room "+nr1+" already has a Door in Direction "+d);
  Room r2=maze.roomNo(nr2);
  if (r2==null)
    throw new IllegalArgumentException();
  if (r2.getSide(oposite(d)) instanceof DoorWall)
    throw new IllegalArgumentException(
      "Room "+nr2+" already has a Door in Direction "+oposite(d));
  DoorWall dw = new DoorWall (r1, r2);
  r1.setSide(d, dw);
  r2.setSide(oposite(d), dw);
}
```

While this still allows to connect arbitrary rooms with doors, the `Builder` could also employ a local grid storing the room numbers. This is then used to add rooms to a grid location and check their gird location when connecting rooms with doors.

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.

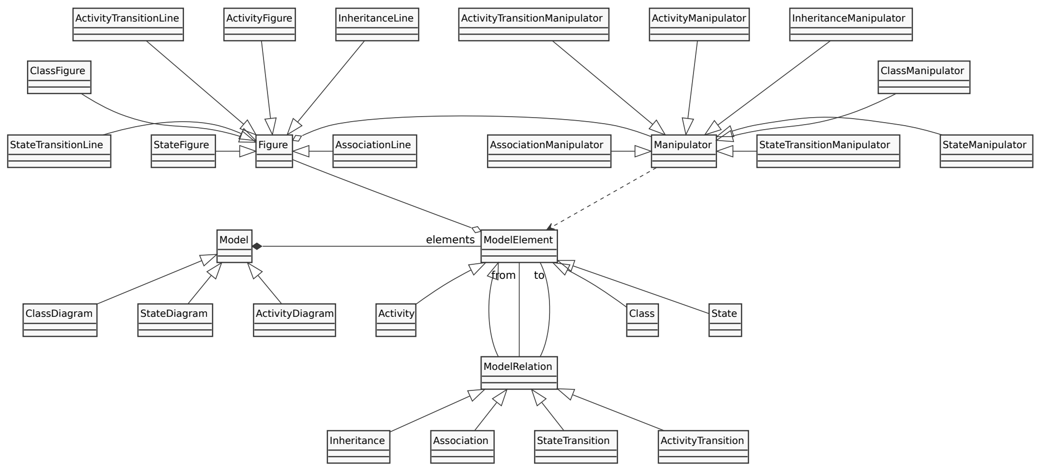
An example of such an editor is *ArgoUML*.[1] It is rather complex, but handles some of the same issues treated in this exercise.

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

   Solution: The `ModelViewController` design pattern is very often used in such situations. It allows for separating the data (model) from the views and the manipulation gestures (controller).

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?

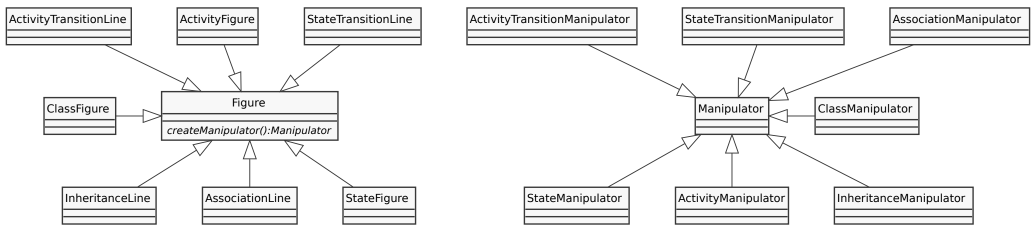
   Solution: There are three hierarchies, one for the model, one for the read-only figures, and one for the manipulators. We need to enforce a parallelism constraint between these hierarchies, so that a `ClassManipulator` is always associated with a `ClassFigure`, which in turn is always associated with a `Class` model object.



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

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.

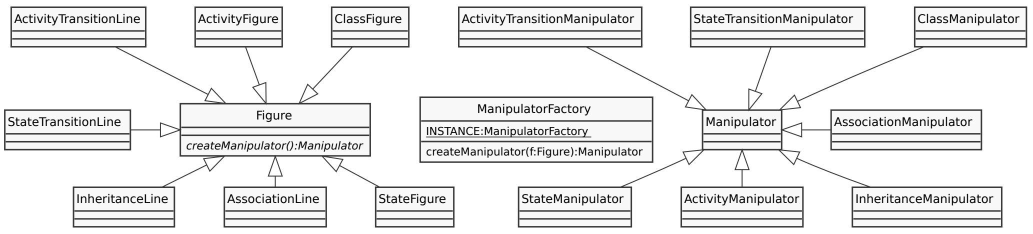
Solution: Add a `createManipulator()` method to the top-level class of the figure hierarchy. This is implemented in concrete figures, thus enforcing the parallelism constraint.



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

Solution: The abstract factory has only one method `createManipulatorFor(Figure f)`, which uses the class of `f` to decide on the manipulator to create.

This solution has the advantage that it permits different kinds of manipulations (e.g., resizing, deleting) for the same figure. On the downside, we need to use reflection to obtain the class of the figure and switch on this information. This decision is better handled by the polymorphic `createManipulator()` method from the factory method design.



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.