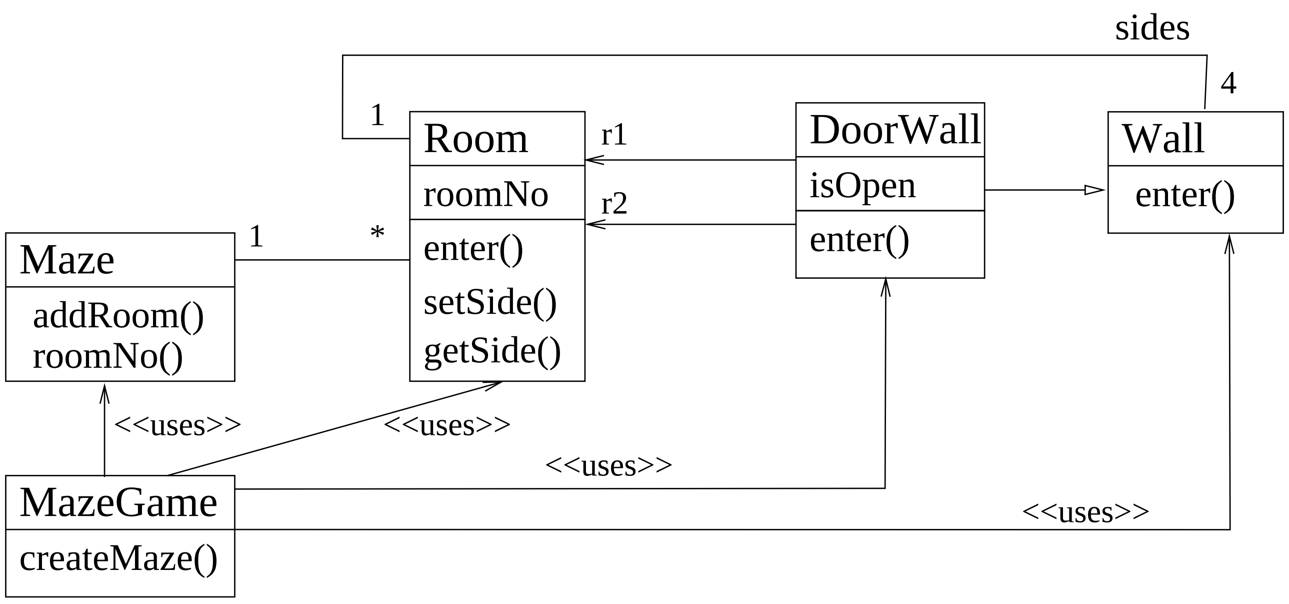
Design Patterns and Frameworks
Dipl.-Inf. Steffen Zschaler
INF 2097
`http://st.inf.tu-dresden.de/teaching/dpf`

Exercise Sheet No. 2
Software Engineering Group
Institute for Software and Multimedia Technology
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.

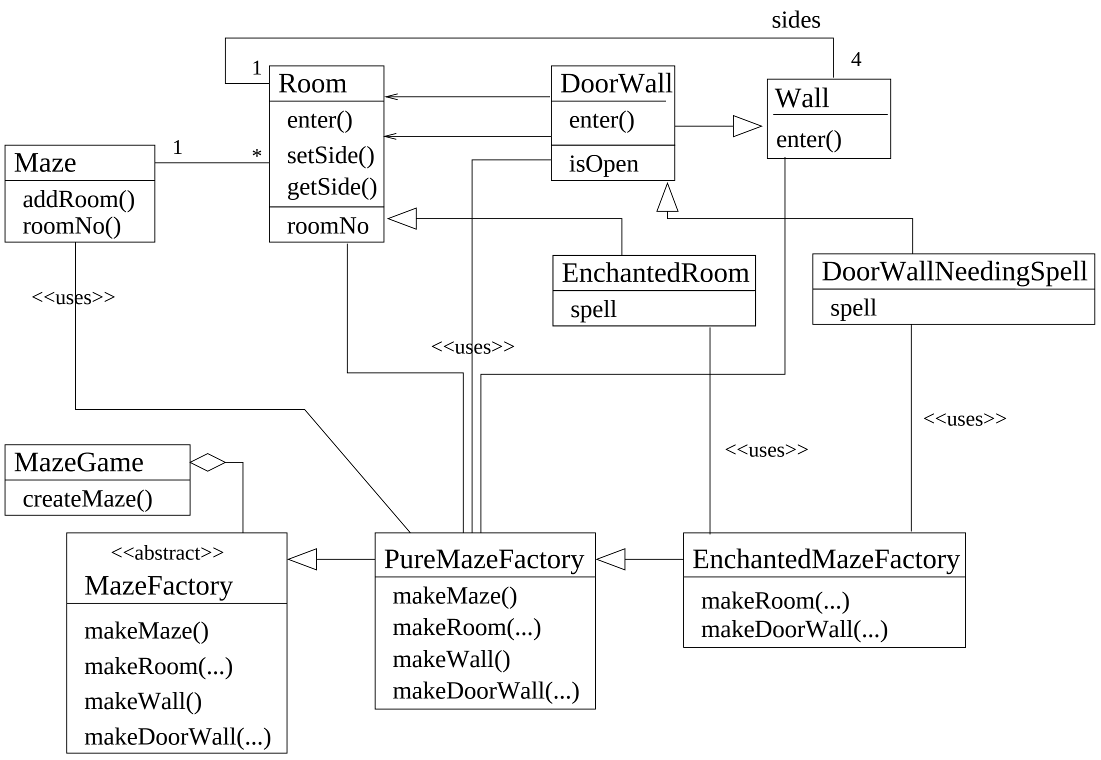
Spells work like this: Invoking a spell brings it into effect for the room in which the player is located and for a certain amount of time, after which the effect "wears off". If during this time the player attempts to pass an enchanted door and if the spell invoked is the spell required for the door, 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) **Task:**

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

Solution: The Java code is sketched below.

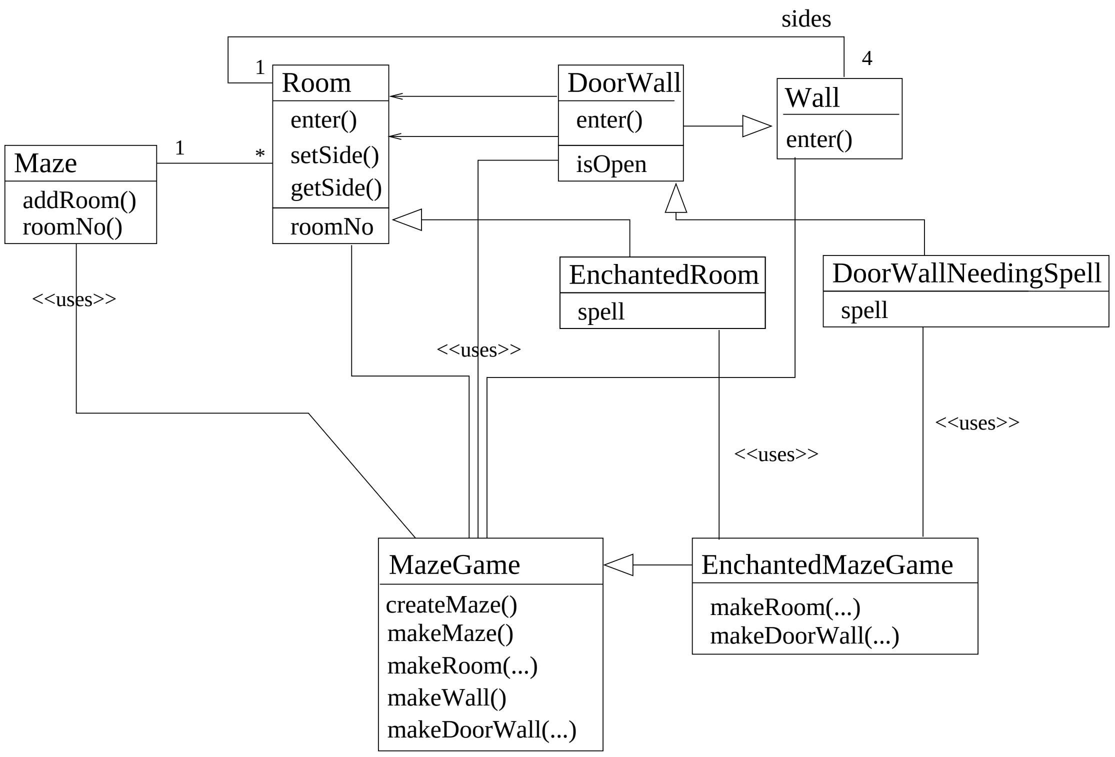


```
public class EnchantedRoom extends Room {

  private String spell;

  EnchantedRoom (int roomNo, String spell) {
    super(roomNo);
    this.spell  = spell;
  }
}

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

public class DoorWallNeedingSpell extends DoorWall {
  private String spell;

  DoorWallNeedingSpell (Room r1, Room r2, String spell) {
    super(r1, r2);
    this.spell = spell;
  }
}

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

public interface MazeFactory {
  public Maze makeMaze();
  public Wall makeWall();
  public Room makeRoom(int n);
  public DoorWall makeDoorWall(Room r1, Room r2);
}

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

public class PureMazeFactory implements MazeFactory {
  //Singleton
  private static PureMazeFactory TheFactory;

  protected PureMazeFactory() {}

  public static MazeFactory instance() {
    if (TheFactory == null)
      TheFactory = new PureMazeFactory();
    return TheFactory;
  }

  public Maze makeMaze() {
    return new Maze();
  }

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

  public Room makeRoom (int n) {
    return new Room (n);
  }

  public DoorWall makeDoorWall (Room r1, Room r2) {
    return new DoorWall (r1, r2);
  }
}

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

class EnchantedMazeFactory extends PureMazeFactory {
  //Singleton
  private static EnchantedMazeFactory TheFactory;

  protected EnchantedMazeFactory() {}

  public static MazeFactory instance() {
    if (TheFactory == null)
      TheFactory = new EnchantedMazeFactory();
    return TheFactory;
  }

  public Room makeRoom (int n) {
    return new EnchantedRoom (n, someSpell());
  }

  public DoorWall makeDoorWall (Room r1, Room r2) {
    return new DoorWallNeedingSpell (r1, r2, someSpell());
  }

  private String someSpell() {
    return "Abrakadabra";  // for now
  }
}

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

class MazeGame implements /* uses */ Directions {
  public static void main(String[] argv) {
    // MazeFactory factory = PureMazeFactory.instance();
    MazeFactory factory = EnchantedMazeFactory.instance();

    createMaze(factory);
  }

  private static Maze createMaze (MazeFactory factory) {
    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 (north, factory.makeWall());
    r1.setSide (east,  d);
    r1.setSide (south, factory.makeWall());
    r1.setSide (west,  factory.makeWall());

    r2.setSide (north, factory.makeWall());
    r2.setSide (east,  factory.makeWall());
    r2.setSide (south, factory.makeWall());
    r2.setSide (west,  d);

    return aMaze;
  }
}
```

1b) **Task:**

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

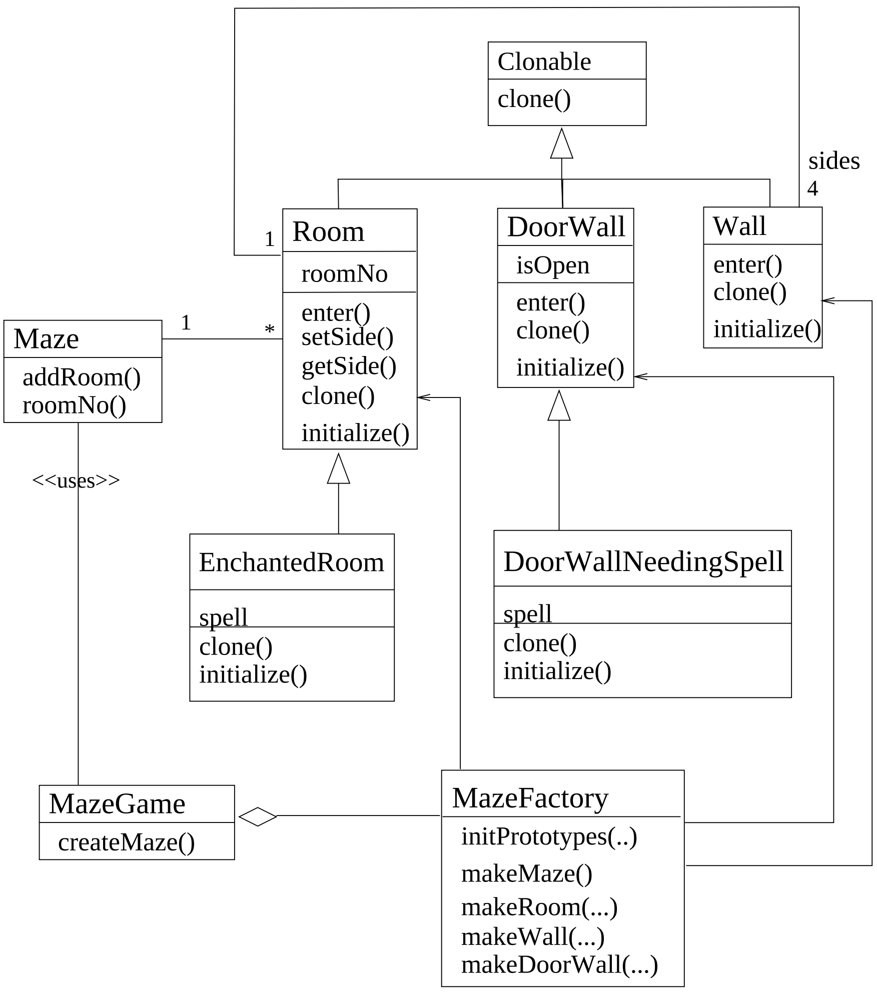
Solution:



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

  protected Maze makeMaze() {
    return new Maze();
  }

  protected Room makeRoom (int r) {
    return new Room (r);
  }

  protected DoorWall makeDoorWall (Room r1, Room r2) {
    return new DoorWall (r1,r2);
  }

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

  protected 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 (north, makeWall());
    r1.setSide (east,  d);
    r1.setSide (south, makeWall());
    r1.setSide (west,  makeWall());

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

    return aMaze;
  }
}

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

class EnchantedMazeGame extends MazeGame {
  public static void main(String[] argv) {
    new EnchantedMazeGame().createMaze();
  }

  protected Room makeRoom (int r) {
    return new EnchantedRoom (r, someSpell());
  }

  protected DoorWall makeDoorWall (Room r1, Room r2) {
    return new DoorWallNeedingSpell (r1, r2, someSpell());
  }

  private String someSpell() {
    return "Abrakadabra";  // for now
  }
}
```

1c) **Task:**

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.



Solution:

```
public class Room implements Cloneable {

  private Wall[] sides = new Wall[4];
  private int roomNo;

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

  public Room() {}

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

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

  public Object clone() {
   return super.clone();
  }

  public void initialize (int roomNo) {
    this.roomNo = roomNo;
  }
}

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

```
public class EnchantedRoom extends Room {

  private String spell;

  public EnchantedRoom (int roomNo, String spell) {
    super (roomNo);
    this.spell  = spell;
  }

  public EnchantedRoom() {};

  public void initialize (int roomNo) {
    super.initialize (roomNo);
    spell  = someSpell();
  }

  private String someSpell() {
    return "Abrakadabra";  // for now
  }
}

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

public class DoorWall extends Wall implements Cloneable {
  ...
}

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

public class DoorWallNeedingSpell extends DoorWall {
  ...
}

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

public class Wall implements Cloneable {
  public Object clone() {
    return super.clone();
  }
}

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

public class Maze implements Cloneable {
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 Object clone() {
    return super.clone();
  }
}

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

public class PrototypeFactory {
  //Singleton
  private static PrototypeFactory TheFactory;

  //Prototypes
  private Maze protoMaze;
  private Room protoRoom;
  private Wall protoWall;
  private DoorWall protoDoorWall;
  private PrototypeFactory (Maze m, Room r, Wall w, DoorWall d) {
    protoMaze = m;
    protoRoom = r;
    protoWall = w;
    protoDoorWall = d;
  }

  public static PrototypeFactory instance (Maze m, Room r, Wall w, DoorWall d) {
    if (TheFactory == null)
      TheFactory = new PrototypeFactory (m,r,w,d);

    return TheFactory;
  }

  public Maze makeMaze() {
    return (Maze) protoMaze.clone();
  }

  public Wall makeWall() {
    return (Wall) protoWall.clone();
  }

  public Room makeRoom (int n) {
    Room newRoom = (Room) protoRoom.clone();

    newRoom.initialize (n);

    return newRoom;
  }

  public DoorWall makeDoorWall (Room r1, Room r2) {
    DoorWall newDoorWall = (DoorWall) protoDoorWall.clone();

    newDoorWall.initialize (r1, r2);

    return newDoorWall;
  }
}

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

class MazeGame implements /* uses */ Directions {
  public static void main(String[] argv) {
/*
    PrototypeFactory factory = PrototypeFactory.instance(
        new Maze(),new Room(), new Wall(), new DoorWall());
*/
    PrototypeFactory factory = PrototypeFactory.instance(
        new Maze(), new EnchantedRoom(), new Wall(), new DoorWallNeedingSpell());

    createMaze (factory);
  }

  static Maze createMaze (PrototypeFactory factory) {
    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(north, factory.makeWall());
    r1.setSide(east,  d);
    r1.setSide(south, factory.makeWall());
    r1.setSide(west,  factory.makeWall());
    r2.setSide(north, factory.makeWall());
    r2.setSide(east,  factory.makeWall());
    r2.setSide(south, factory.makeWall());
    r2.setSide(west,  d);

    return aMaze;
  }
}
```

1d) **Task:**

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 a 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.

For prototypes, methods for cloning and intializing must be programmed.

The ABSTRACTFACTORY pays off in comparison to factory methods, if there are several users of the set of factory methods. In the implementation of (a), we could have done without the abstract factory `MazeFactory`, since `PureMazeFactory` and `EnchantedMazeFactory` are subtypes. That is the realization of (b). Then, `PureMazeFactory` would play the role of an abstract and a concrete factory.

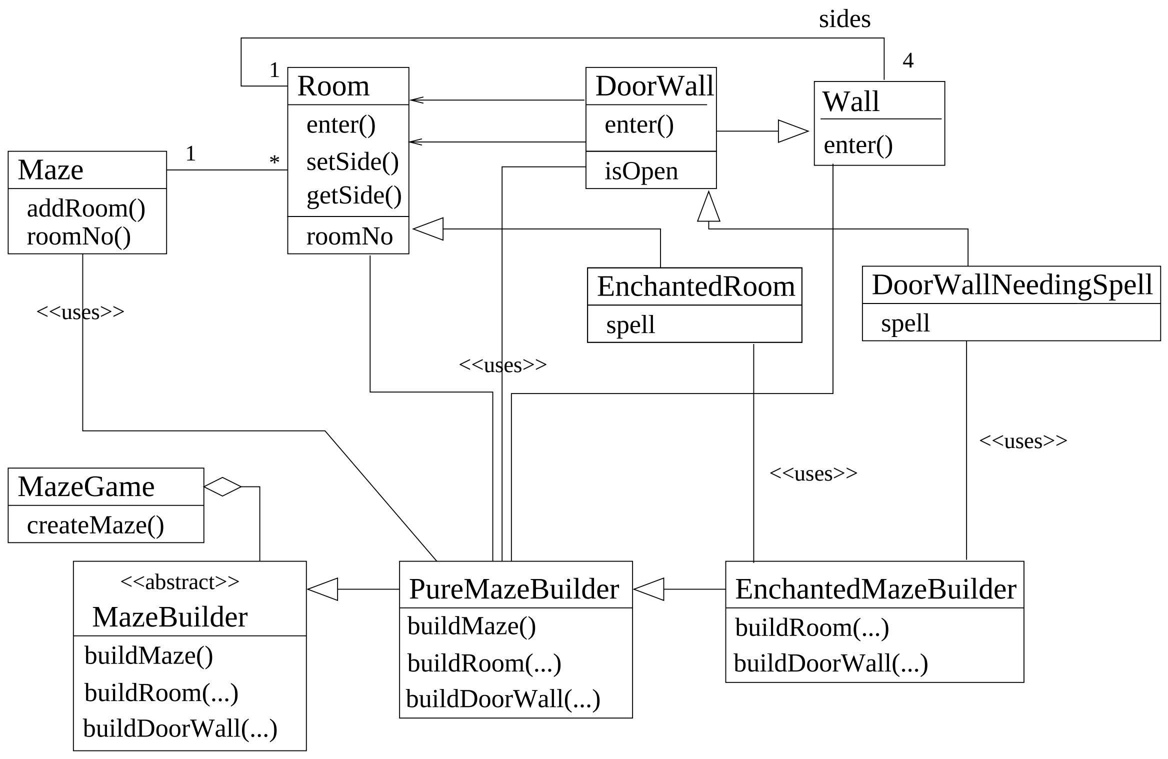
Task 2: Building Mazes

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) **Task:**

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?

Solution:



```
public class MazeGame implements /* uses */ Directions {
  public static void main(String[] argv) {
    // MazeBuilder builder = new PureMazeBuilder();
    MazeBuilder builder = new EnchantedMazeBuilder();

    createMaze(builder);
  }

  private static Maze createMaze (MazeBuilder builder) {
    builder.buildMaze();
    builder.buildRoom (1);
    builder.buildRoom (2);
    builder.buildDoorWall (1, east, 2, west);
    return builder.getMaze();
  }
}

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

public interface MazeBuilder extends /* uses */ Directions {
  public void buildMaze();
  public void buildRoom (int r);
  public void buildDoorWall (int r1, int side1, int r2, int side2);
  public Maze getMaze();
}

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

public class PureMazeBuilder implements MazeBuilder {
  protected Maze currentMaze;

  public void buildMaze() {
    currentMaze = new Maze();
  }

  public Maze getMaze() {
    return currentMaze;
  }

  public void buildRoom (int r) {
    if (currentMaze.roomNo (r) == null) {
      Room room = new Room (r);
      currentMaze.addRoom (room);

      room.setSide(north, new Wall());
      room.setSide(east,  new Wall());
      room.setSide(south, new Wall());
      room.setSide(west,  new Wall());
    }
  }

  public void buildDoorWall (int r1, int side1, int r2, int side2) {
    Room room1 = currentMaze.roomNo(r1);
    Room room2 = currentMaze.roomNo(r2);

    if ((room1 != null) && (room2 != null)) {
      DoorWall d = new DoorWall (room1, room2);

      room1.setSide (side1, d);
      room2.setSide (side2, d);
    }
  }
}

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

public class EnchantedMazeBuilder extends PureMazeBuilder {
  public void buildRoom (int r) {
    if (currentMaze.roomNo(r) == null) {
      Room room = new EnchantedRoom (r, someSpell());
      currentMaze.addRoom (room);

      room.setSide (north, new Wall());
      room.setSide (east,  new Wall());
      room.setSide (south, new Wall());
      room.setSide (west,  new Wall());
    }
  }

  public void buildDoorWall (int r1, int side1, int r2, int side2) {
    Room room1 = currentMaze.roomNo(r1);
    Room room2 = currentMaze.roomNo(r2);

    if ((room1 != null) && (room2 != null)) {
      DoorWall d = new DoorWallNeedingSpell (room1, room2, someSpell());

      room1.setSide(side1,d);
      room2.setSide(side2,d);
    }
  }

  private String someSpell() {
    return "Abrakadabra";  // for now
  }
}
```

Builder encapsulates the current product and allows its modification only through specifically provided operations in Builder's interface.

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.

3a) **Task:** What design pattern is useful for this type of application where users need different views on their data?

Solution: The Model-View-Controller design pattern is very often used in such situations. It allows to separate the data (model) from the views and the manipulation gestures (controller).

3b) **Task:** 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 parallelity constraint between these hierarchies, so that a class manipulator is always associated with a class figure, which in turn is always associated with a class model object.

3c) **Task:** 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 parallelity constraint.

3d) **Task:** 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 allows different kinds of manipulation (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.

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.

4a) **Task:**

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.