Design Patterns

Enumerations, and Creational patterns

Dr. Chad Williams
Central Connecticut State University

Enumerations

- Finite set of values
- Assign “name” to a finite value to improve readability
- Orientation
  - HORIZONTAL
  - VERTICAL
- C/C++ enumeration separate type allows compile time checking, Java ~~does~~ did not

Simple/common approach

```
public interface Orientation{
  public static final int HORIZONTAL = 0;
  public static final int VERTICAL = 1;
}

public class GameCharacter{
  int orientation;
  public void setDirection(int direction){
    orientation = direction;
  }
  public void someFunction(){
    setDirection(Orientation.HORIZONTAL);
  }
}
```

Problems

- No compile time checking
- setDirection could be called with an integer that is an invalid value

```
System.out.println("orientation="+orientation);
orientation=1
```

- To output Orientation in readable format must decode value

```
public static String name(int orientation){
  if (orientation == 0){
    return "horizontal";
  }else if(orientation == 1){
    return "vertical";
  }else{
    return "ERROR";
  }
}
```

Better approach – type safe enumeration

```
public enum Orientation{
   VERTICAL(“vertical”),
   HORIZONTAL(“horizontal);
   private final String name;
   private Orientation(String name){
      this.name = name;
   }
   public String toString(){ return name;}
   public Orientation rotateRight(Orientation
      curOrientation){
   ...
  }
}
```

Use

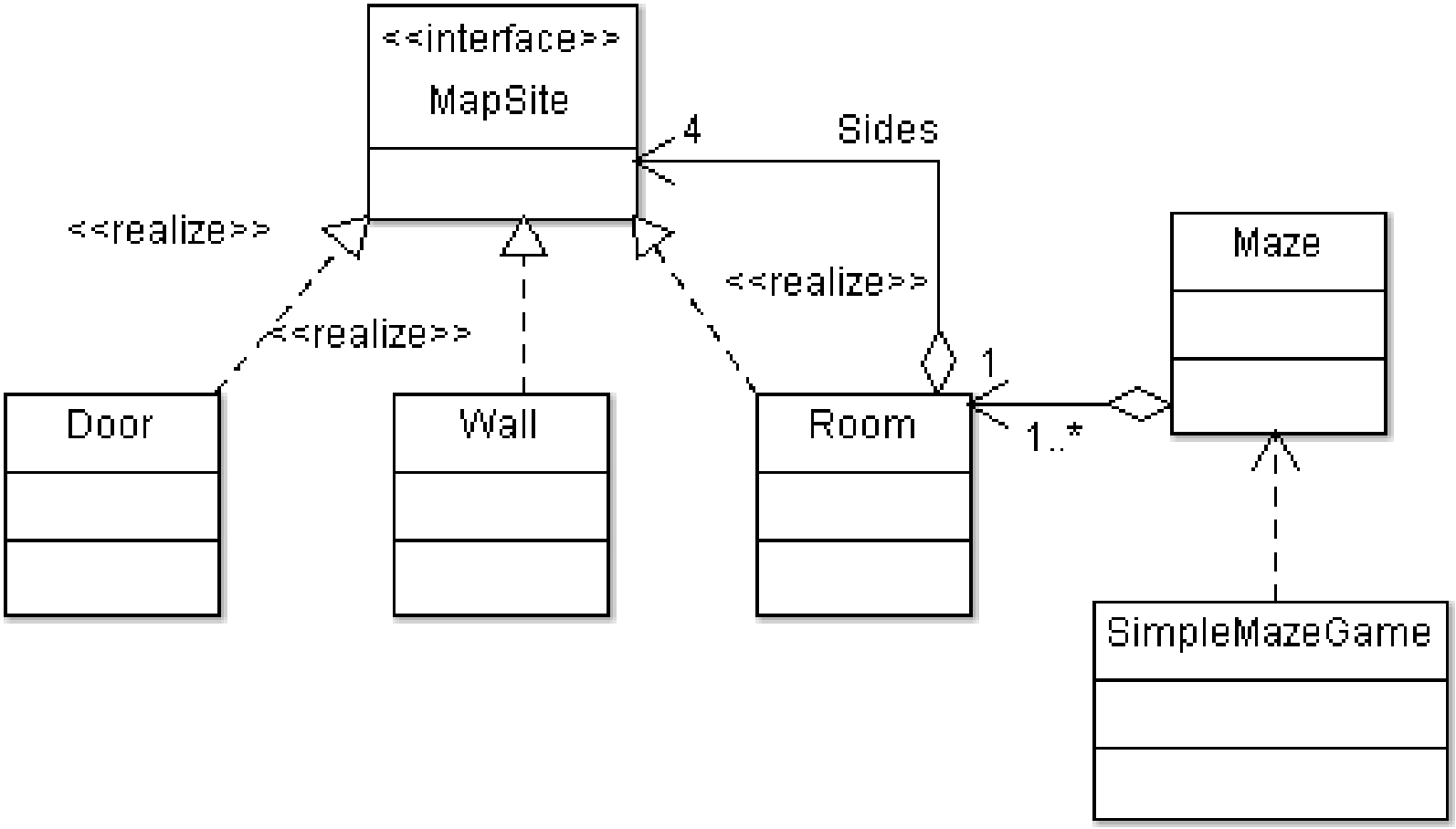
```
public class GameCharacter{
   Orientation orientation;
   public void setDirection(Orientation direction){
      orientation = direction;
   }
   public void someFunction(){
      setDirection(Orientation.HORIZONTAL);
   }
}

System.out.println("orientation="+orienta
 tion);
```

```
orientation=horizontal
```

Creational patterns

- **Please see the code demos there are lots of additional very detailed examples of these**

Maze example



Basic approach to creation

```
public static Maze
  createHarryPotterMaze(){
  Maze maze = new Maze();
   Room room1 = new HarryPotterRoom(1);
   Room room2 = new HarryPotterRoom(2);
   Door door = new
  HarryPotterDoor(room1,room2);
   room1.setSide(Direction.NORTH, new
  HarryPotterWall();
   room1.setSide(Direction.EAST, door);
   …
```

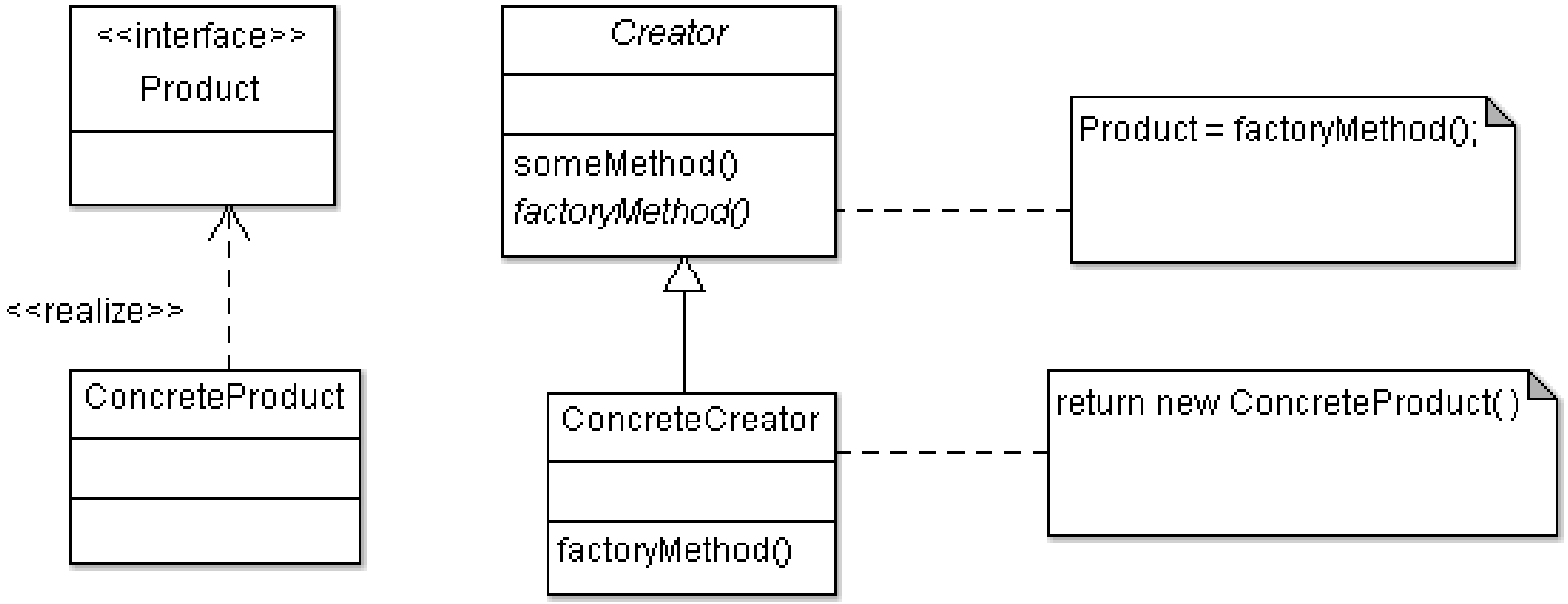
Problem with the approach

- Duplicate code for each different type of style
- Difficult to maintain – change in one place must be made in all
- Maze creation tightly tied to each of the different maze types

Factory method

- **Category:** Creational design pattern
- **Intent:** Define interface for creating an object but defer instantiation to subclasses
- Also known as *virtual constructor*
- **Applicability**
  - When class can't anticipate class of objects it must create
  - When class defers to subclasses to specify objects it creates.

Factory Method UML



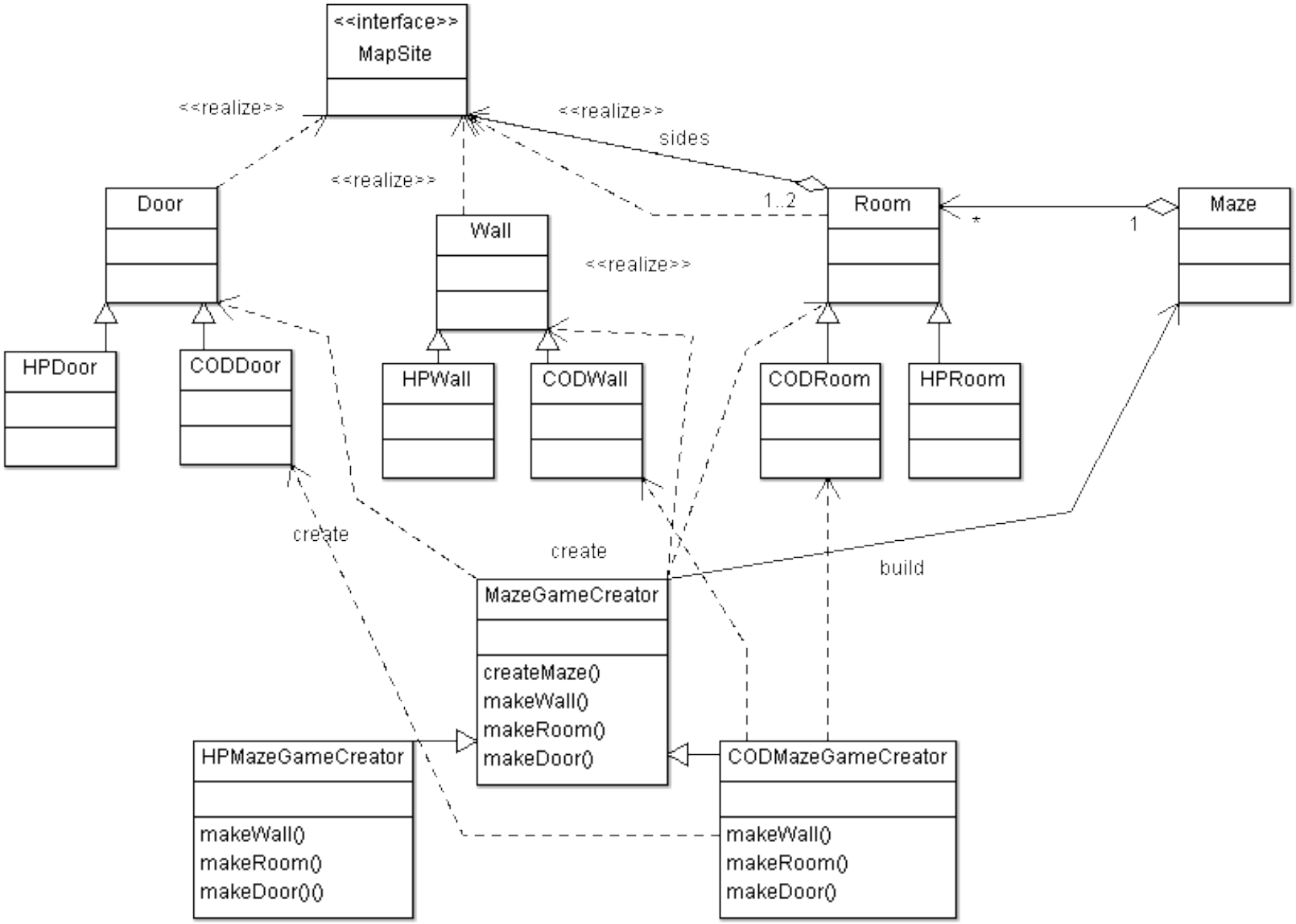
Factory method roles

- **Product** – defines interface of objects to be created
- **ConcreteProduct** – implements Product interface defines implementation
- **Creator** – defines 1 or more factory methods that create abstract products. May define default behavior calling one or more factory methods to create products
- **ConcreteCreator** – overrides factory methods to return instance of a ConcreteProduct

Difference Factory vs. Factory Method

- **Factory pattern -** defines a class whose sole responsibility is to create new objects
- **Factory method –** class that defers creation of certain objects to its subclasses

Maze Factory method



Creator pattern

```
public class HarryPotterMazeGameCreator
                  extends MazeGameCreator{
  public Wall makeWall(){
    return new HarryPotterWall();
  }
  public Room makeRoom(int roomNum){
    return new
 HarryPotterRoom(roomNum);
  }
  public Door makeDoor(Room r1,
                       Room r2){
    return new HarryPotterDoor(r1,r2);
  }
```

Creator

```
public class MazeGameCreator{
   public Maze createMaze(){
      Maze maze = makeMaze();
      Room room1 = makeRoom(1);
      Room room2 = makeRoom(2);
      Door door1 = makeDoor(room1, room2);
      room1.setSide(Direction.NORTH,
  door1);
      room1.setSide(Direction.EAST,
  makeWall();
      room1.setSide(Direction.SOUTH,
  makeWall();
      …
}
```

Factory method main

```
public static void main(String[]
  args){
   MazeGameCreator creator = new
  HarryPotterMazeGameCreator();
   maze = creator.createMaze();
}
```

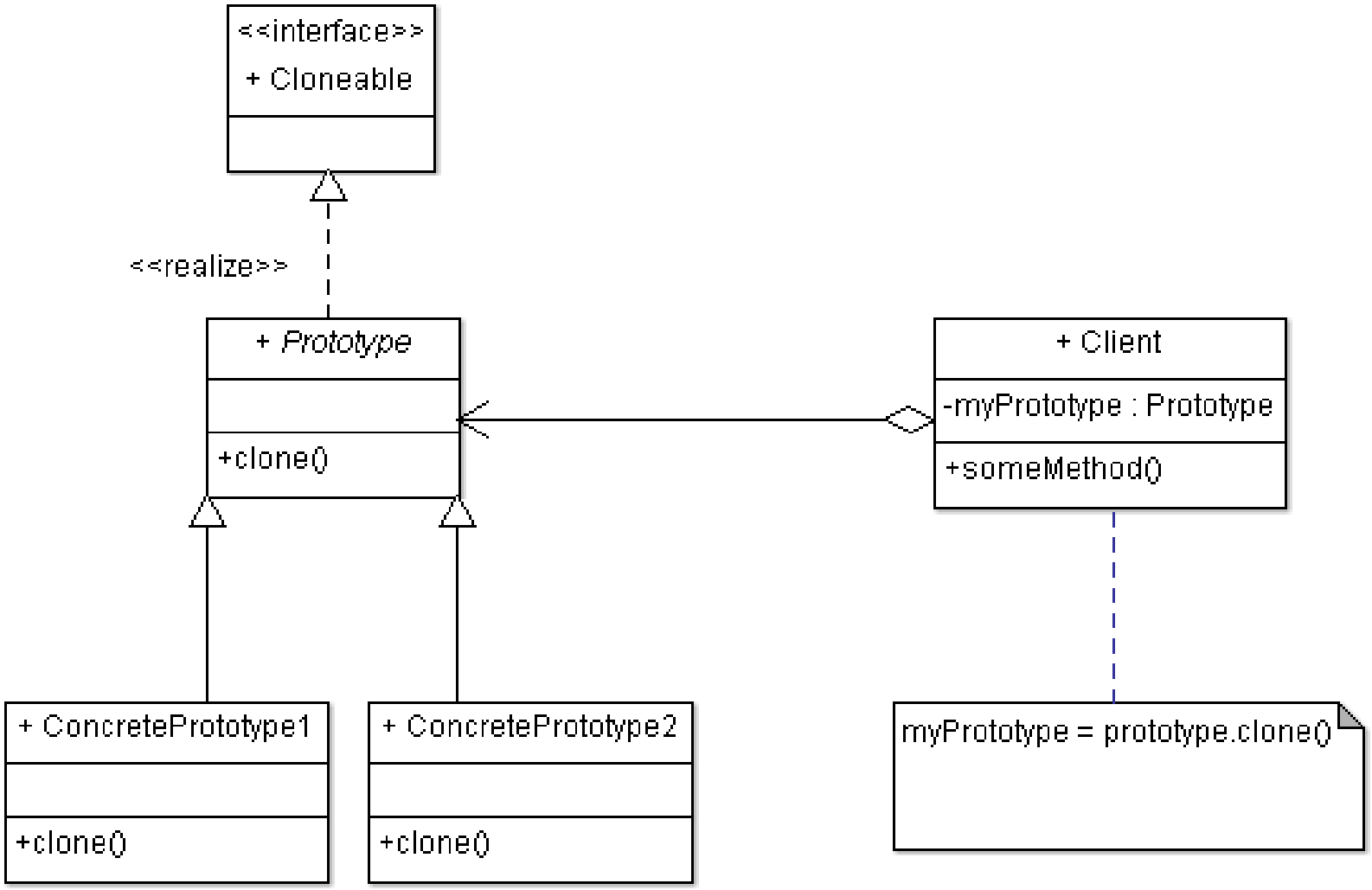
Prototype pattern

- Downside of AbstractFactory and FactoryMethod is they are tied to a specific set of subclasses
- Many different themes can result in explosion of number of subclasses needed
- Prototype pattern allows product families to be defined or altered at runtime

Design pattern: Prototype

- **Category**: Creational design pattern
- **Intent**: Specify kinds of objects using prototypical instance and create new objects by cloning the passed prototype
- **Applicability**:
  - When system should be independent of how components or products are created
  - When classes to instantiate are specified at runtime
  - Avoid class hierarchy of factories that parallels the class hierarchy of products

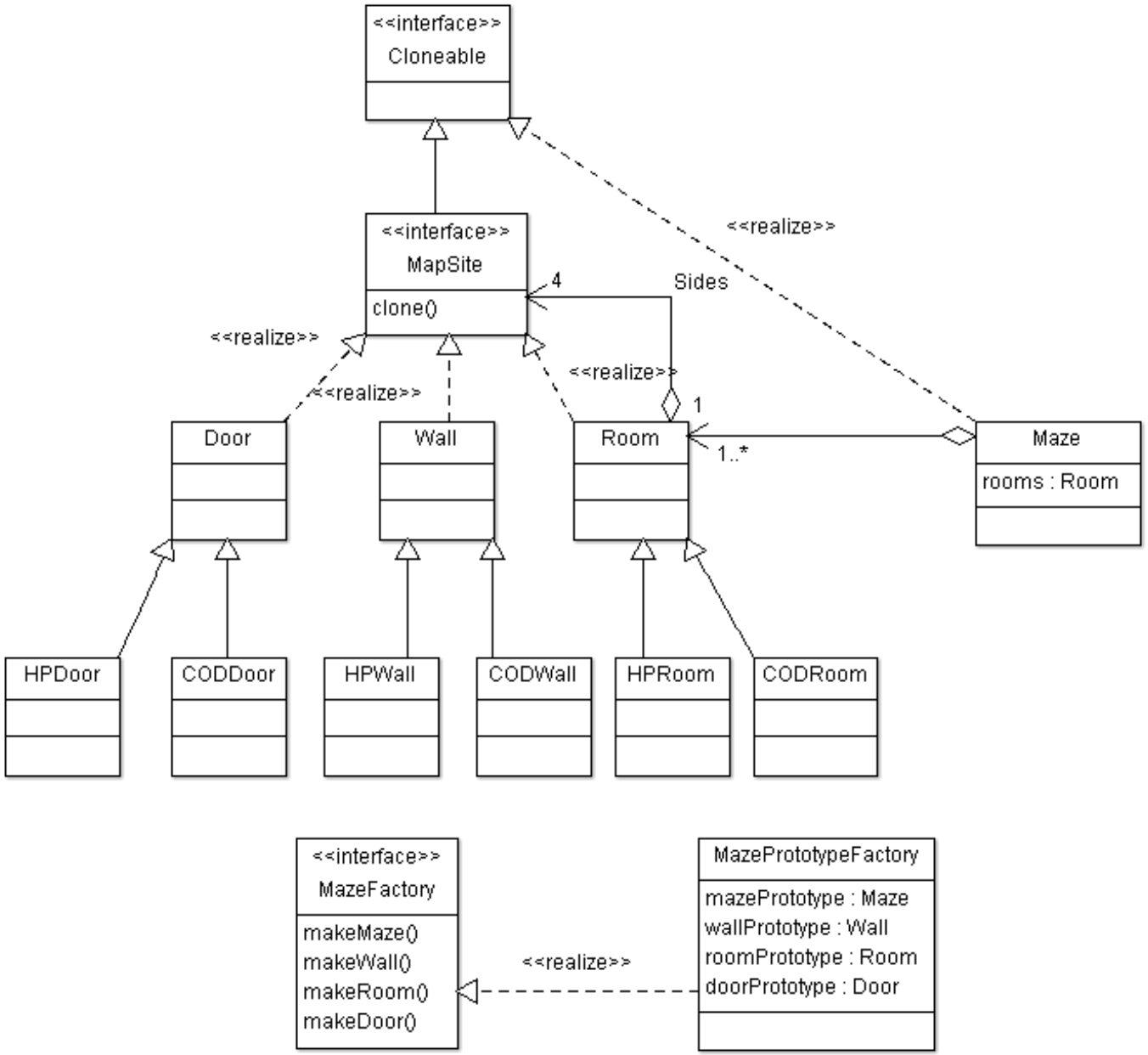
Prototype UML



Prototype roles

- Prototype – defines interfaces of objects to be created. Implements Cloneable interface and defines clone method
- ConcretePrototype – implements Prototype interface and clone method
- Client – creates new instances by cloning the prototype

Maze prototype



MazePrototypeFactory

```
public class MazePrototypeFactory
                extends MazeFactory{
  public MazePrototypeFactory(
      Maze mazePrototype,
      Wall wallPrototype,
      Room roomPrototype,
      Door doorPrototype){
    // Set all to attribute
    // values on the class
  }
```

MazePrototypeFactory cont.

```
public Room makeRoom(int roomNum){
   Room room = (Room)
  roomPrototype.clone();
   room.setRoomNumber(roomNum);
   return room;
}
public static void main(String[] args){
   MazeFactory prototypeFactory;
   protoTypeFactory = new HPMazeFactory();
   MazePrototypeFactory factory =
      new MazePrototypeFactory(
          prototypeFactory.makeMaze(),
          prototypeFactory.makeWall();
          prototypeFactory.makeRoom();

  prototypeFactory.makeDoor(null,null);
   }
```

Builder pattern

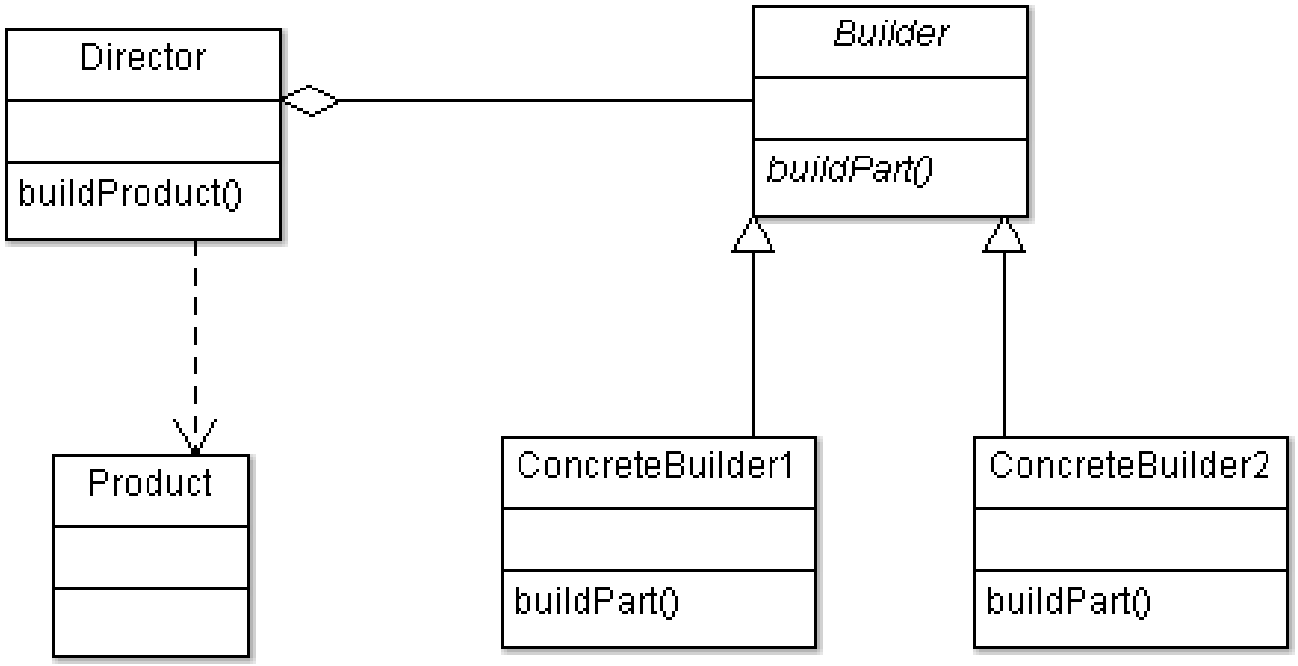
- Pattern used when constructing objects is complex and repetitive
- Example for maze:

```
Room room1 = factory.makeRoom(1);
Room room2 = factory.makeRoom(2);
Door door1 = factory.makeDoor(room1,room2);
room1.setSide(Direction.NORTH,door1);
room1.setWall(Direction.EAST,
  factory.makeWall());
room1.setWall(Direction.WEST,
  factory.makeWall());
room1.setWall(Direction.SOUTH,
  factory.makeWall());
room2.setSide(Direction.SOUTH,door1);
…
```

Design pattern: Builder

- **Category**: Creational design pattern
- **Intent**:
  - Separate construction of complex objects so same construction process can create complex object from different implementation parts
- **Applicability**:
  - When process for creating object should be independent of parts that make up the object
  - When construction process should allow various implementations of the parts used for construction

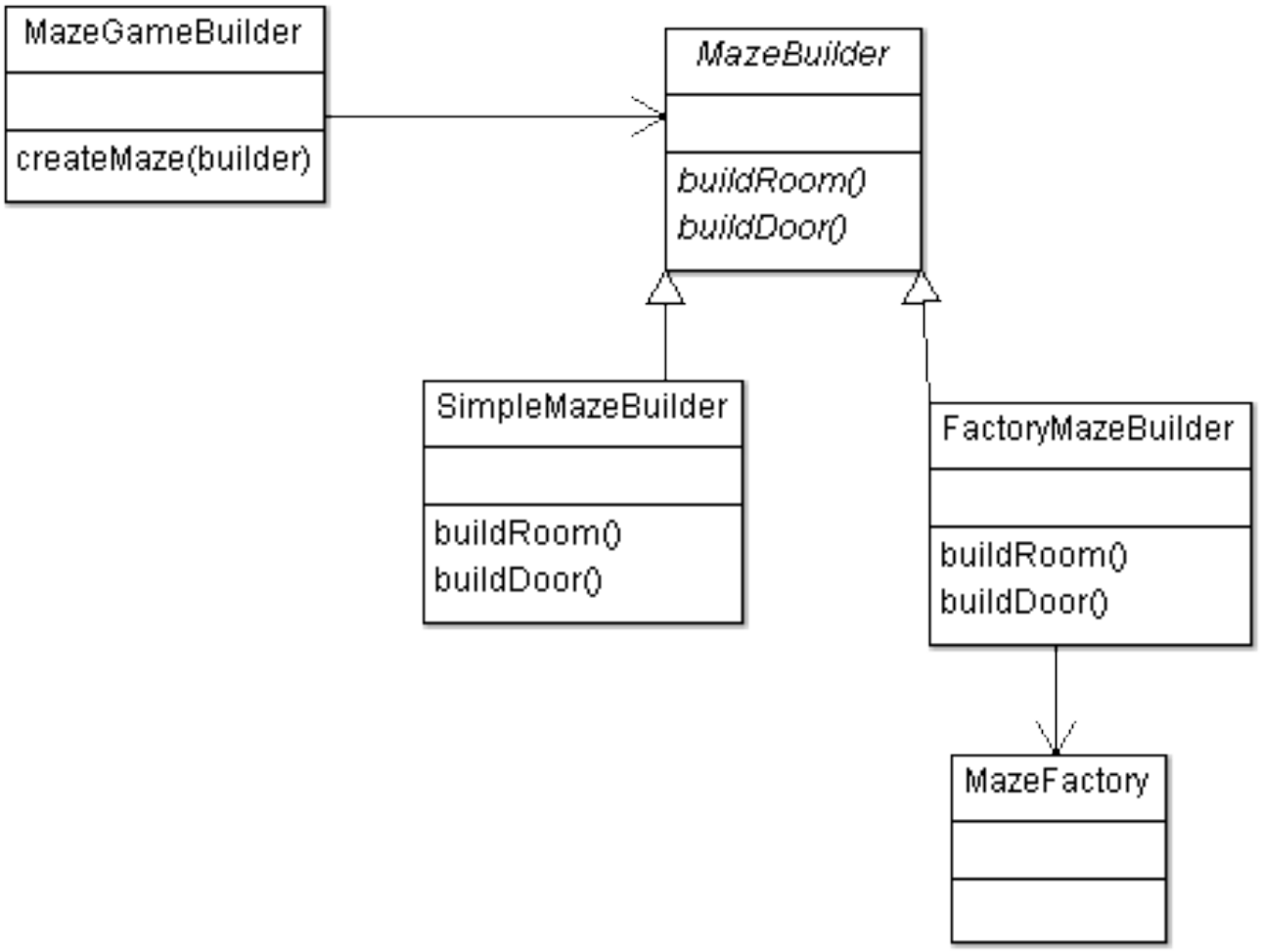
Builder UML


Director
buildProduct()
Builder
buildPart()
Product
ConcreteBuilder1
buildPart()
ConcreteBuilder2
buildPart()

Builder roles

- **Builder** – Defines interface for creating parts of Product object
- **ConcreteBuilder** – Constructs and assembles parts of the product by implementing Builder interface
- **Director** – Constructs a Product using the Builder interface
- **Product** – complex object under construction

Maze builder



SimpleMazeBuilder buildRoom

```
public class SimpleMazeBuilder
                implements MazeBuilder{
  Maze maze = new Maze();
  public void buildRoom(int roomNum){
    Room room = newRoom(roomNum);
    for (Direction
 dir=Direction.first();
          dir != null; dir =
 dir.next()){
      room.setSide(dir,new Wall());
    }
    maze.addRoom(room);
  }
```

SimpleMazeBuilder buildDoor

```
public void buildDoor(int roomNum1,
  int roomNum2, Direction dir){
     Room room1 =
  maze.findRoom(roomNum1);
     Room room2 =
  maze.findRoom(roomNum2;
     if (room1 != null && room2 !=
         null && dir !=null){
       Door door = new
  Door(room1,room2);
       room1.setSide(dir,door);
       room1.setSide(dir.opposite(),
  door);
     }
}
```

Resulting code

```
public static Maze
  createMaze(MazeBuilder builder);
   builder.buildRoom(1);
   builder.buildRoom(2);

  builder.buildDoor(1,2,Direction.NO
  RTH);
   …
   return builder.getMaze();
}
```

Group work

- Describe in detail how you could use the Prototype pattern and Builder pattern within the context of the final project (i.e. it doesn't have to be specifically your current code but how would you apply the concepts to the problem in general)