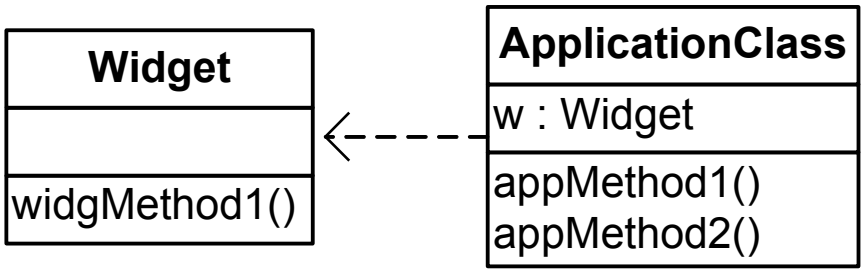
Abstract Factory Pattern

CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
October 29, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Motivation



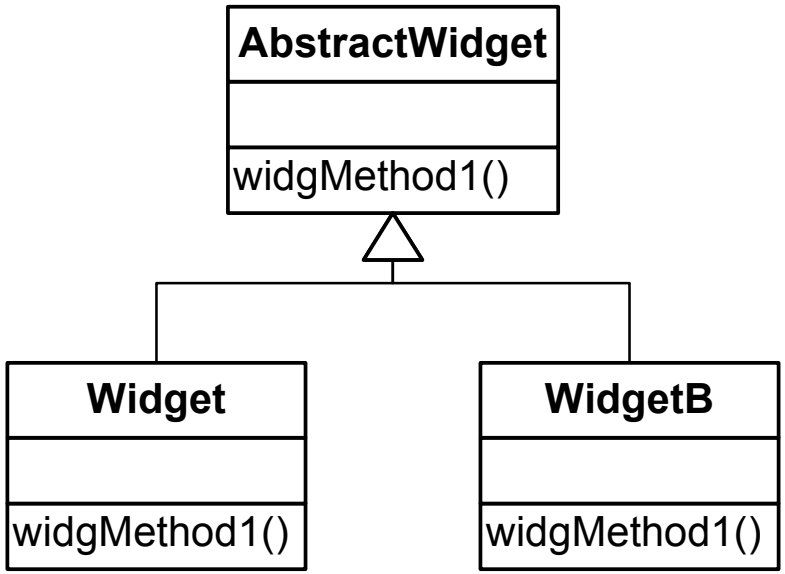

```
class ApplicationClass {
  Widget w;

  public appMethod1() {
    Widget w = new Widget();
    w.widgMethod1();
  }
  ...
```

- We can modify the internal `Widget` code without modifying the `ApplicationClass`
- What happens when we discover a new widget and would like to use it in the `ApplicationClass`?

Problems with Changes

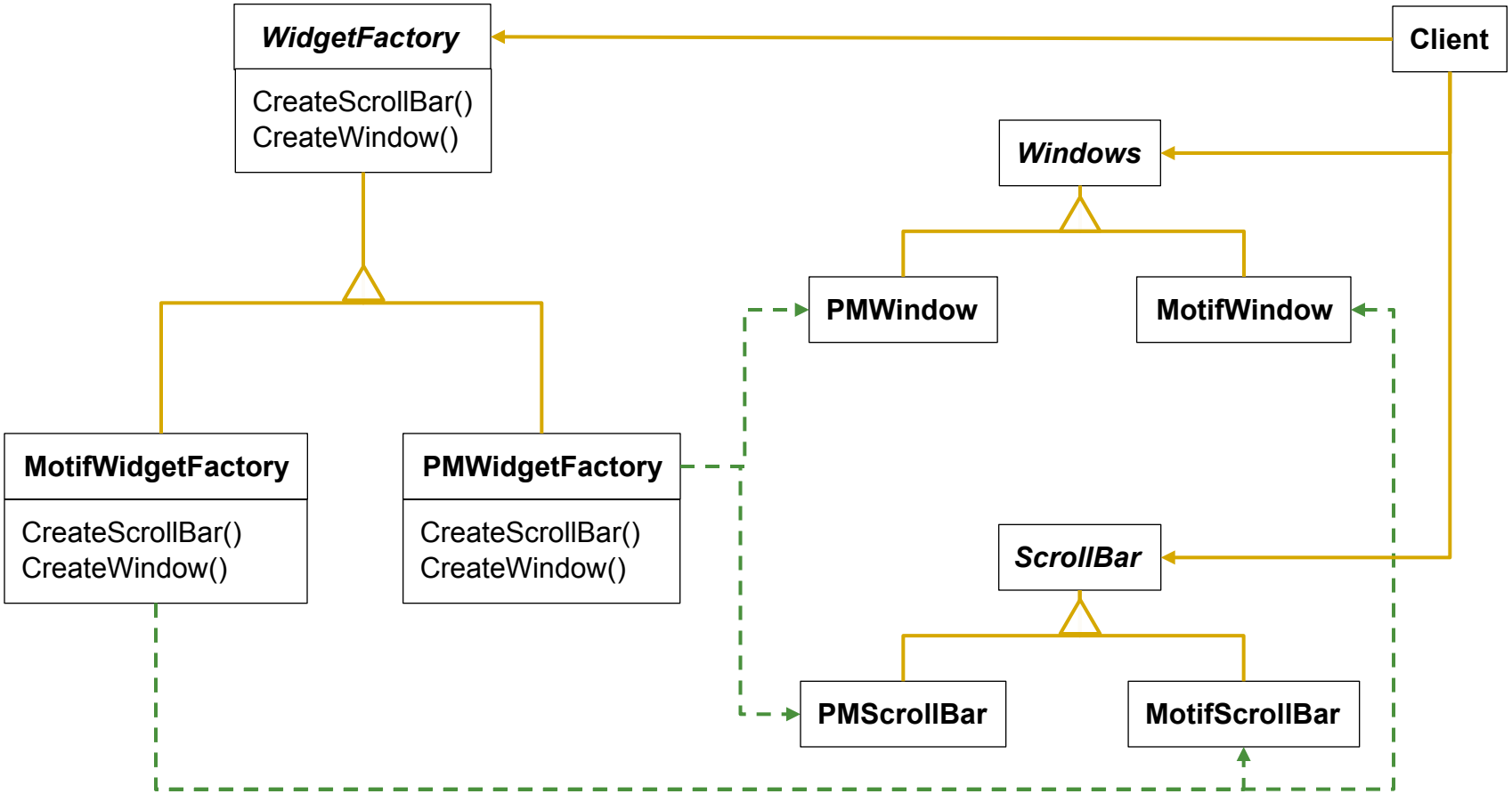
- `Multiple coupling between Widget and ApplicationClass`
- `ApplicationClass` knows the interface of `Widget`
- `ApplicationClass` explicitly uses the `Widget` type
  - Hard to change because `Widget` is a concrete class
- `ApplicationClass` explicitly creates new Widgets in many places
  - If we want to use the new `Widget` instead of the initial one, changes are spread all over the code

Apply "Program to an Interface"



- `ApplicationClass` depends now on an (abstract) interface
  - Helps to solve the problem of explicit use
- But we still have to hard code which widget to create!
  - Problem of explicit creation not solved

WidgetFactory



Abstract Factory *aka* Kit

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

- Motivation
  - User interface toolkit supports multiple look-and-feel standards (Motif, Presentation Manager)
  - Different appearances and behaviors for UI widgets
  - Apps should not hard-code its widgets

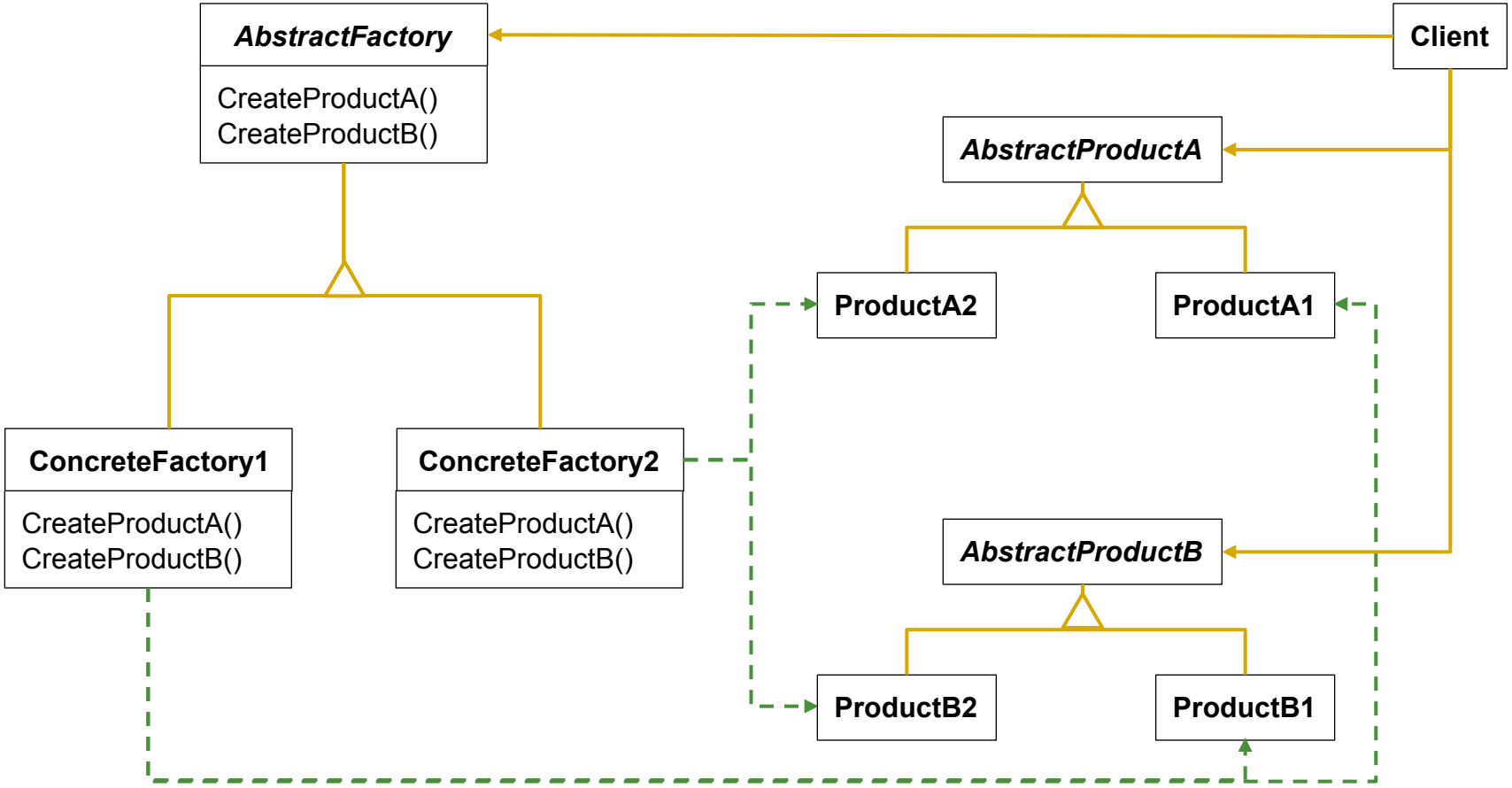
Solution

- Place *Abstract Factory* Class between application layer and Concrete Factory Class(es)
  - Create an Abstract *WidgetFactory* class from a related set of Abstract Interfaces (Abstract Products)
    - e.g., Abstract Interfaces for creating each basic kind of widget
  - Create Concrete WidgetFactory for specific implementations
    - e.g., classes implement specific look-and-feel, and allow for different look-and-feel

Applicability

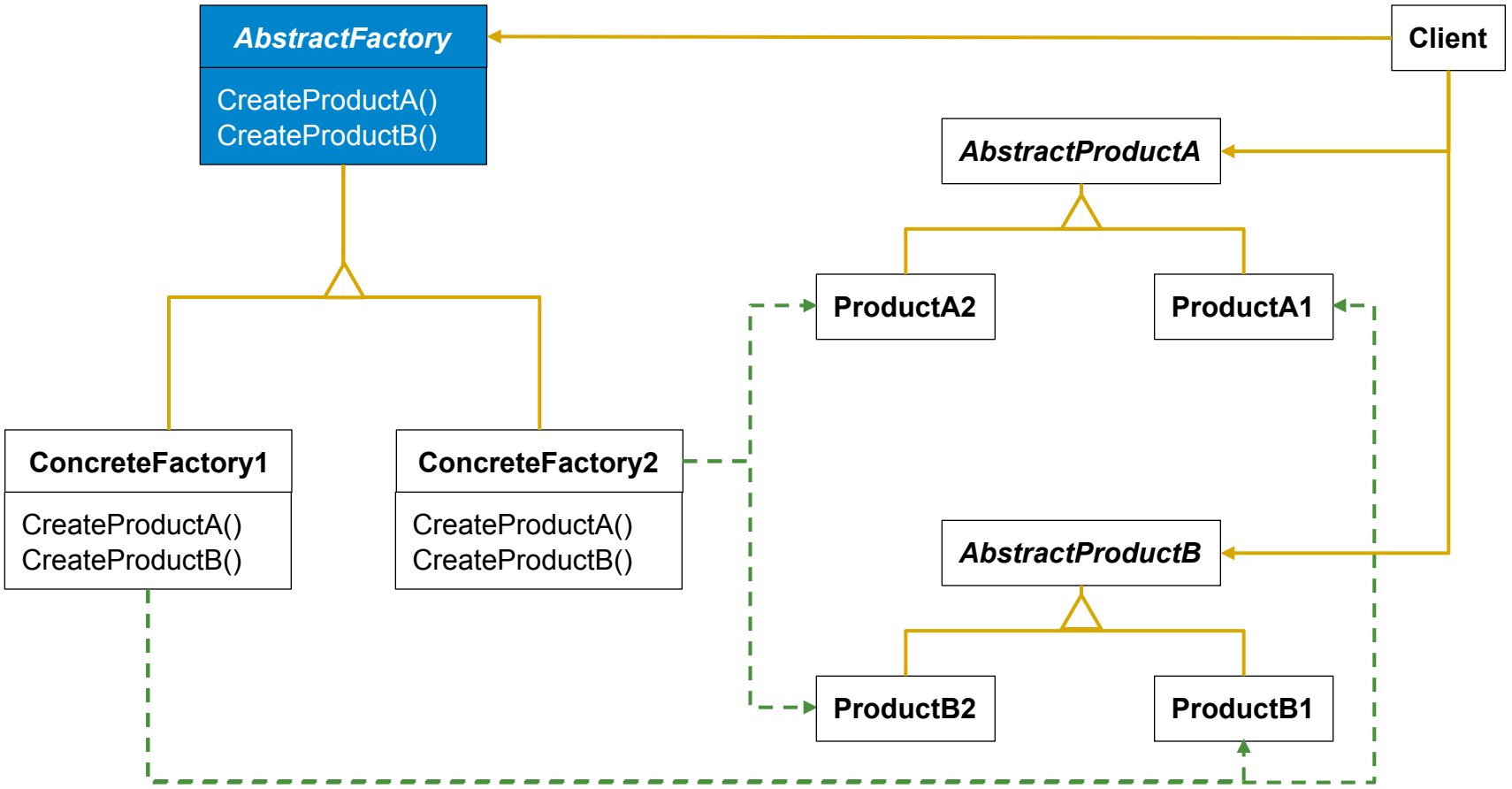
- Use the Abstract Factory pattern when
  - A system should be independent of how its products are created, composed, and represented
  - A system should be configured with one of multiple families of products
  - A family of related product objects is designed to be used together, and you need to enforce this constraint
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Structure


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

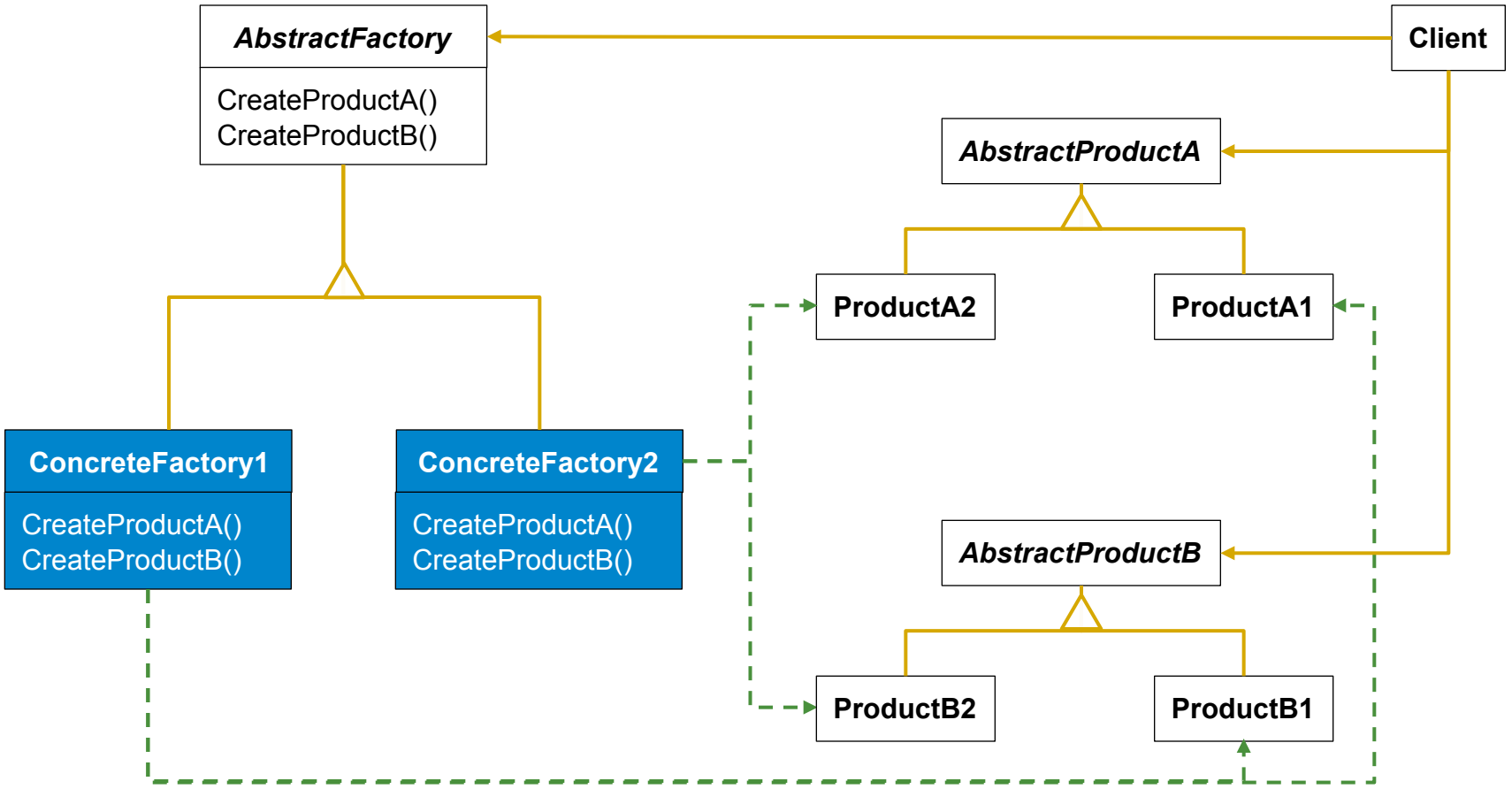
AbstractFactory

- Declares interface for operations that create abstract product objects



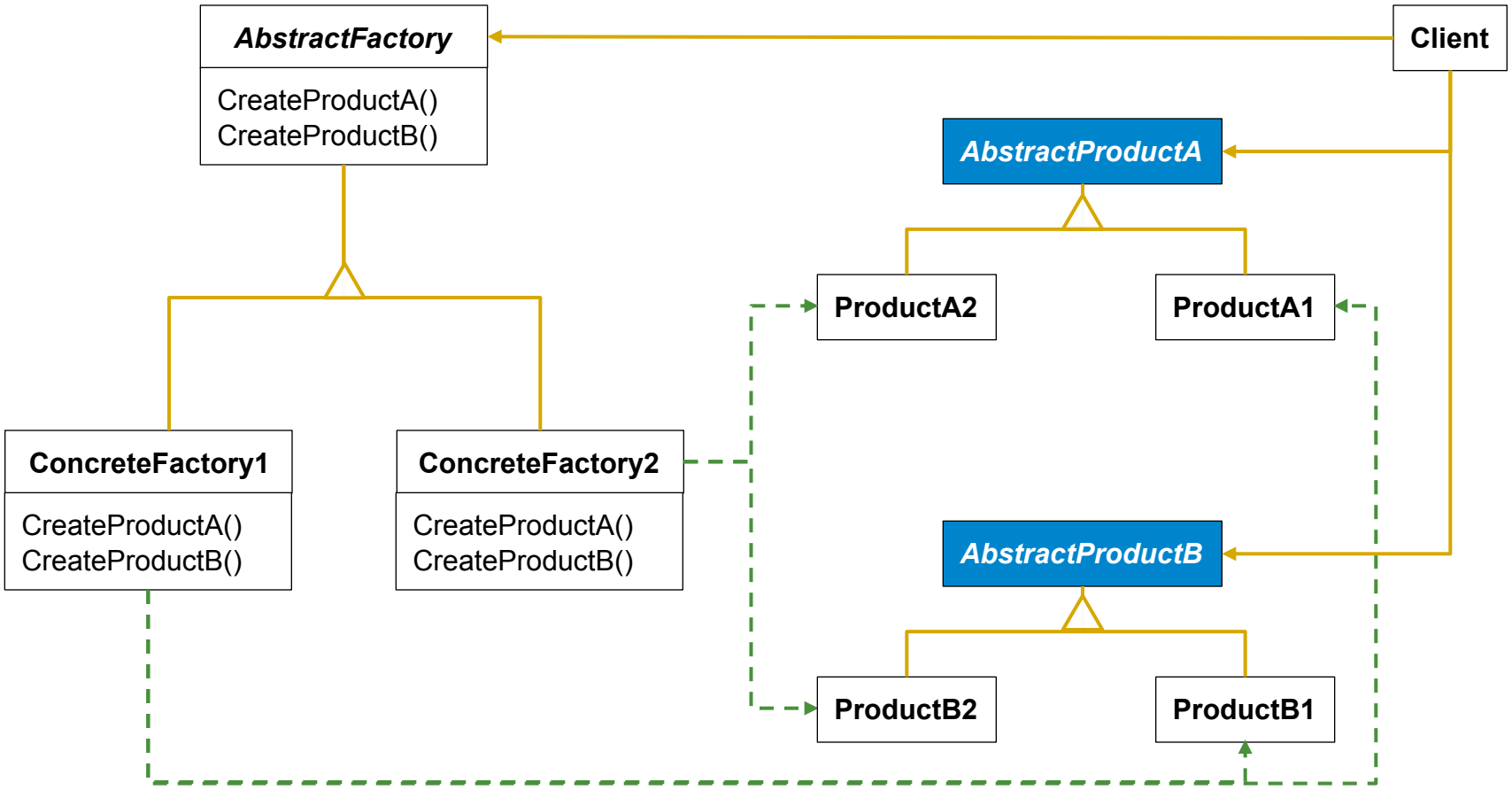
ConreteFactory

- Implements operations to create concrete product object



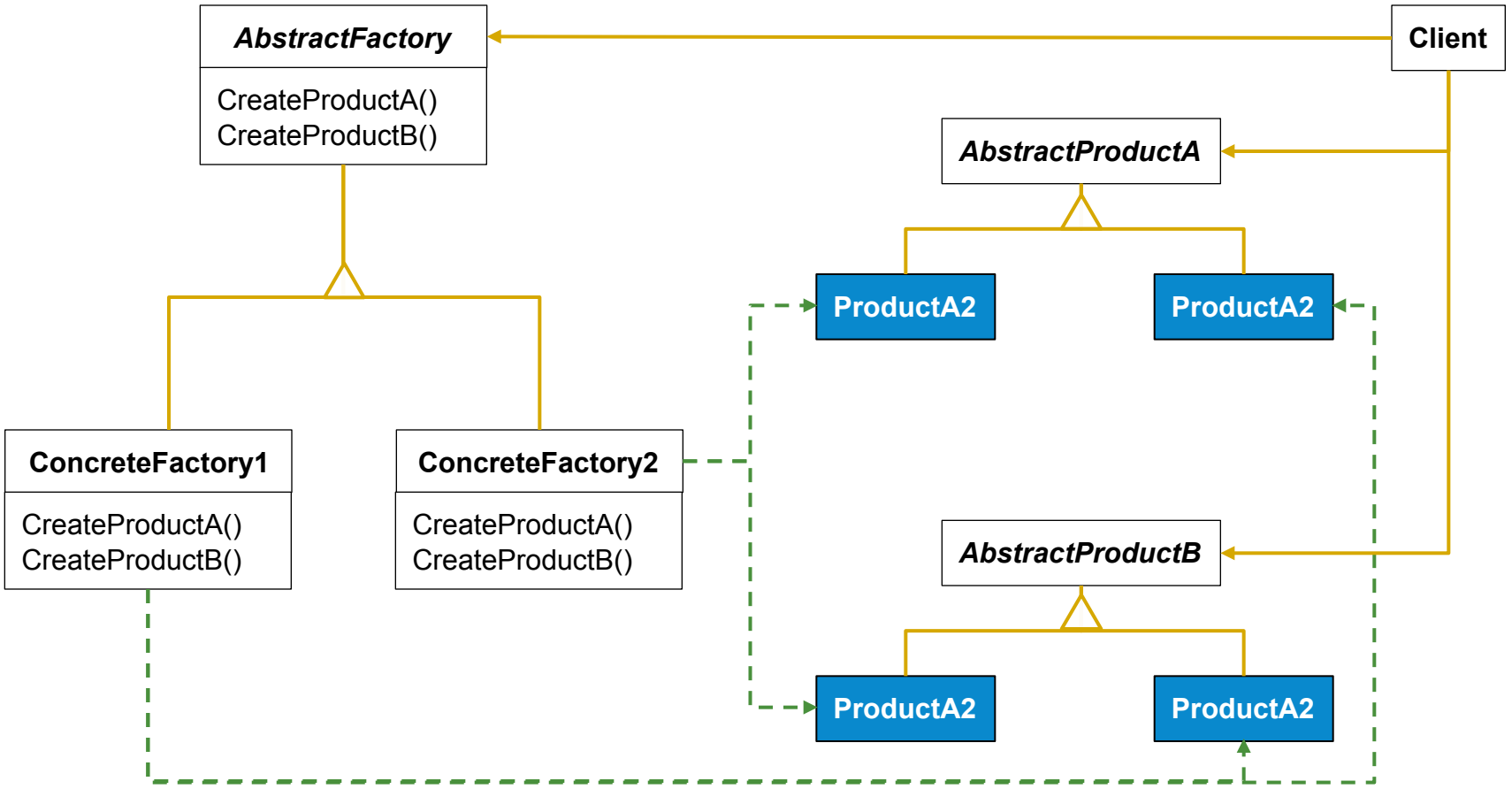
AbstractProduct

- Declares an interface for a type of product object



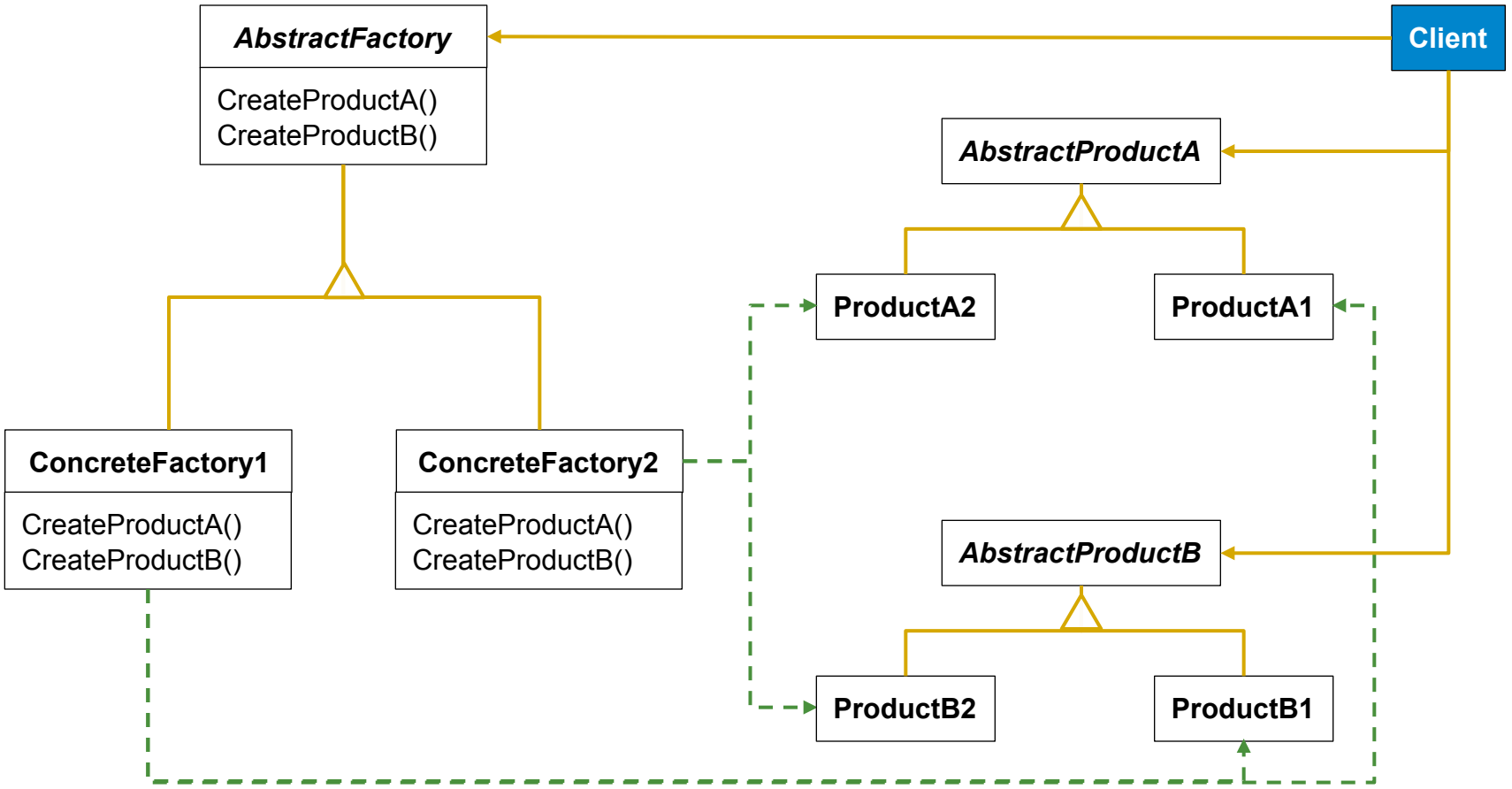
ConcreteProduct

- Defines a product object to be created by concrete factory
- Implements the abstract product interface



Client

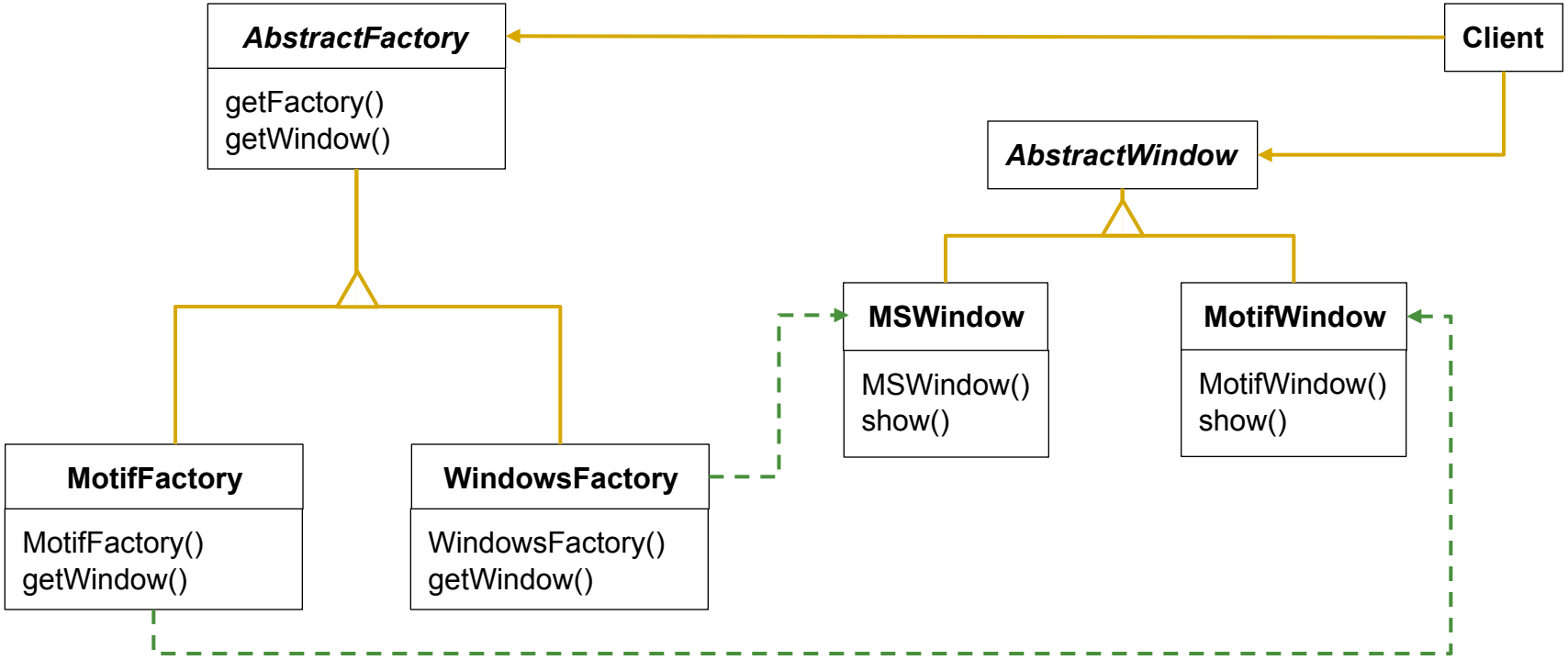
- Uses only interfaces declared by AbstractFactory and AbstractProduct classes



Consequences

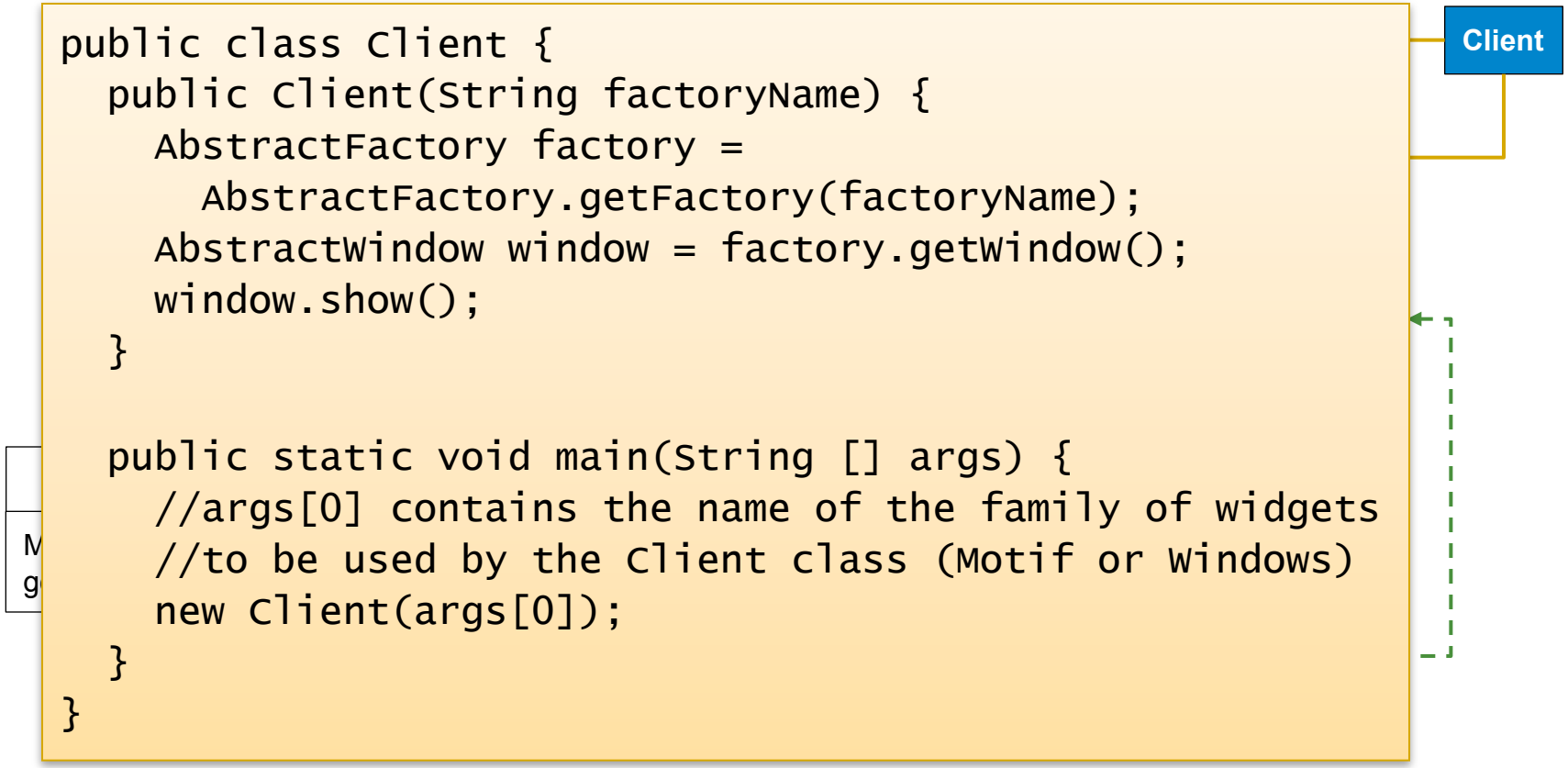
- Explicit creation of `Widget` objects is not anymore dispersed; hence, is easier to change
- `Functional methods in ApplicationClass` are decoupled from various concrete implementations of widgets
  - Product class names do not appear in client code
- Use of Factories forces adherence to interfaces and `encapsulates both interface definitions and` implementations
- Supporting new kinds of products is difficult
  - AbstractFactory and all its subclasses need to be changed

Example – Windows

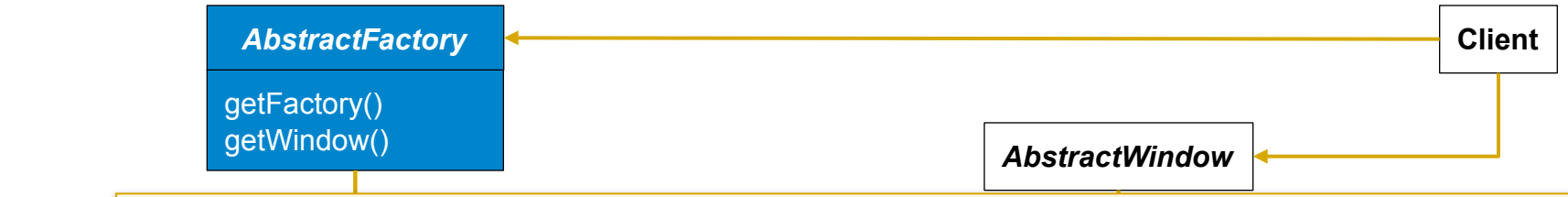


Code for Class Client

```
public class Client {
  public Client(String factoryName) {
    AbstractFactory factory =
      AbstractFactory.getFactory(factoryName);
    AbstractWindow window = factory.getWindow();
    window.show();
  }

  public static void main(String [] args) {
    //args[0] contains the name of the family of widgets
    //to be used by the Client class (Motif or Windows)
    new Client(args[0]);
  }
}
```



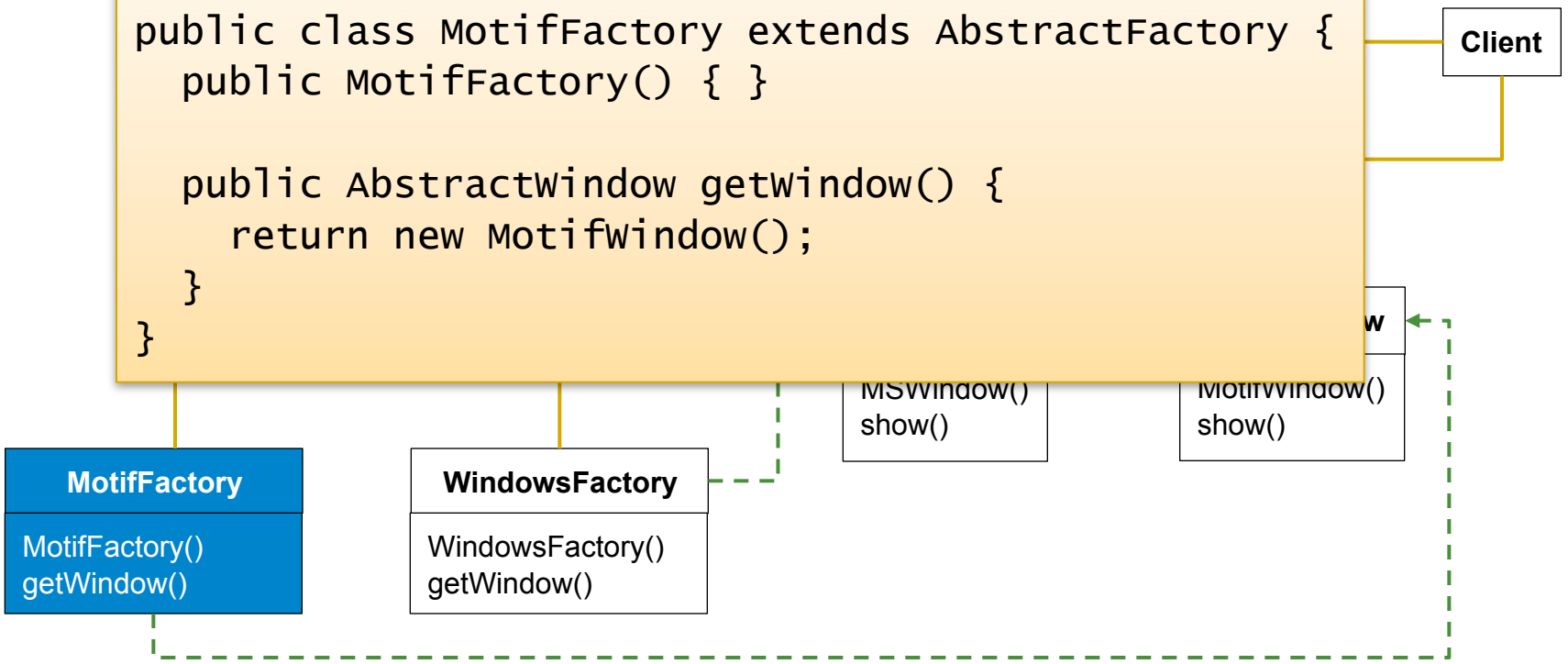
Code for Class AbstractFactory



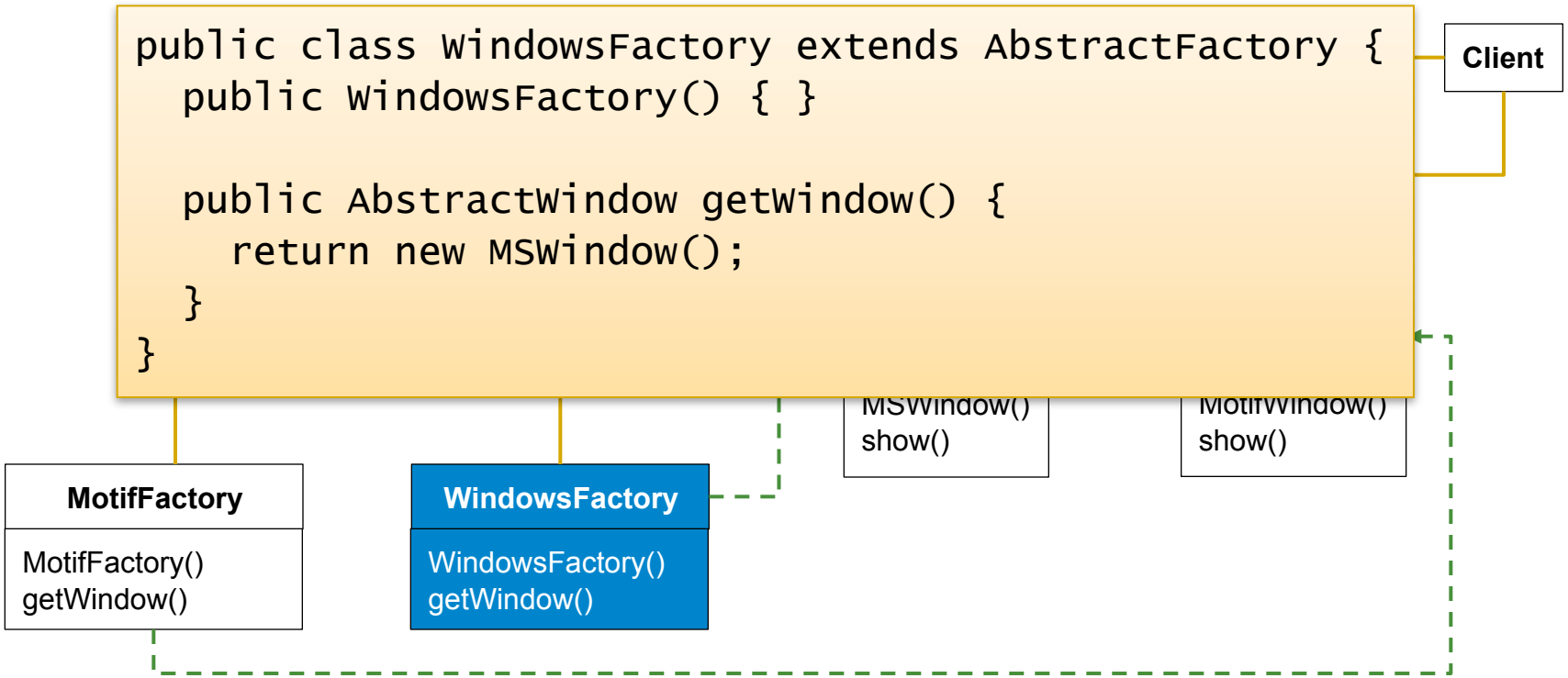

```
public abstract class AbstractFactory {
  public static final String MOTIF_WIDGET_NAME = "Motif";
  public static final String WINDOWS_WIDGET_NAME = MSWindows";

  public static AbstractFactory getFactory(String name) {
    if (name.equals(MOTIF_WIDGET_NAME))
      return new MotifFactory( );
    else if (name.equals(WINDOWS_WIDGET_NAME))
      return new WindowsFactory( );
    return null;
  }

  public abstract AbstractWindow getWindow();
}
```

Code for Class MotifFactory

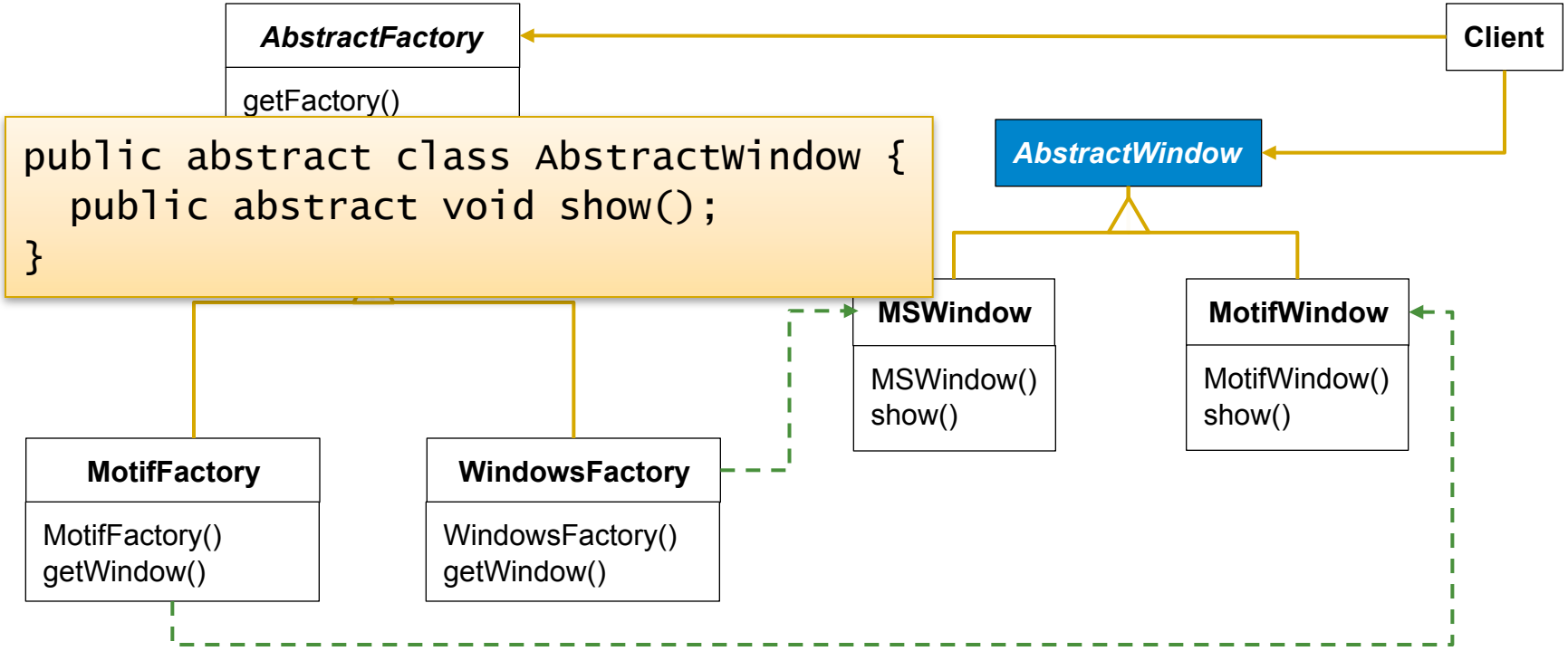
```
public class MotifFactory extends AbstractFactory {
  public MotifFactory() { }

  public AbstractWindow getWindow() {
    return new MotifWindow();
  }
}
```



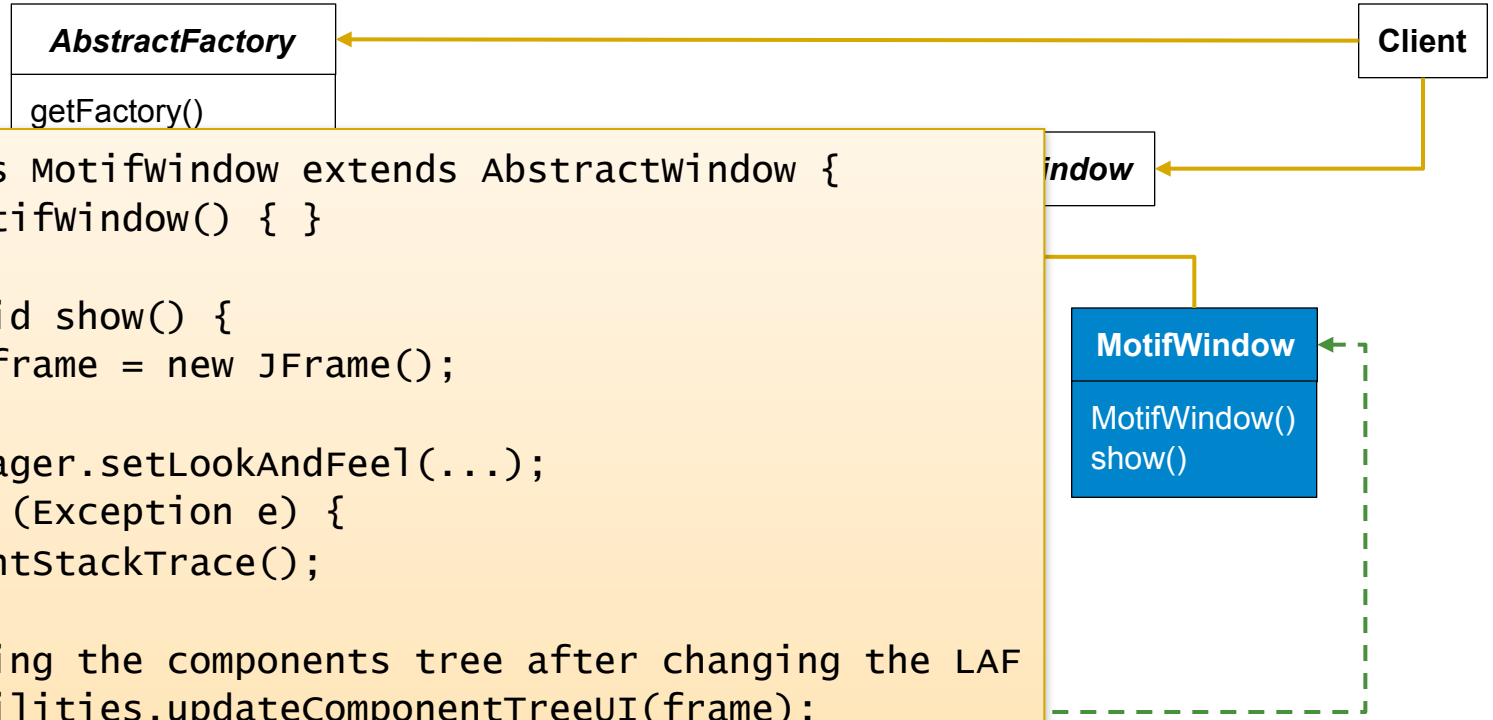
Code for Class WindowsFactory



Code for Class AbstractWindow

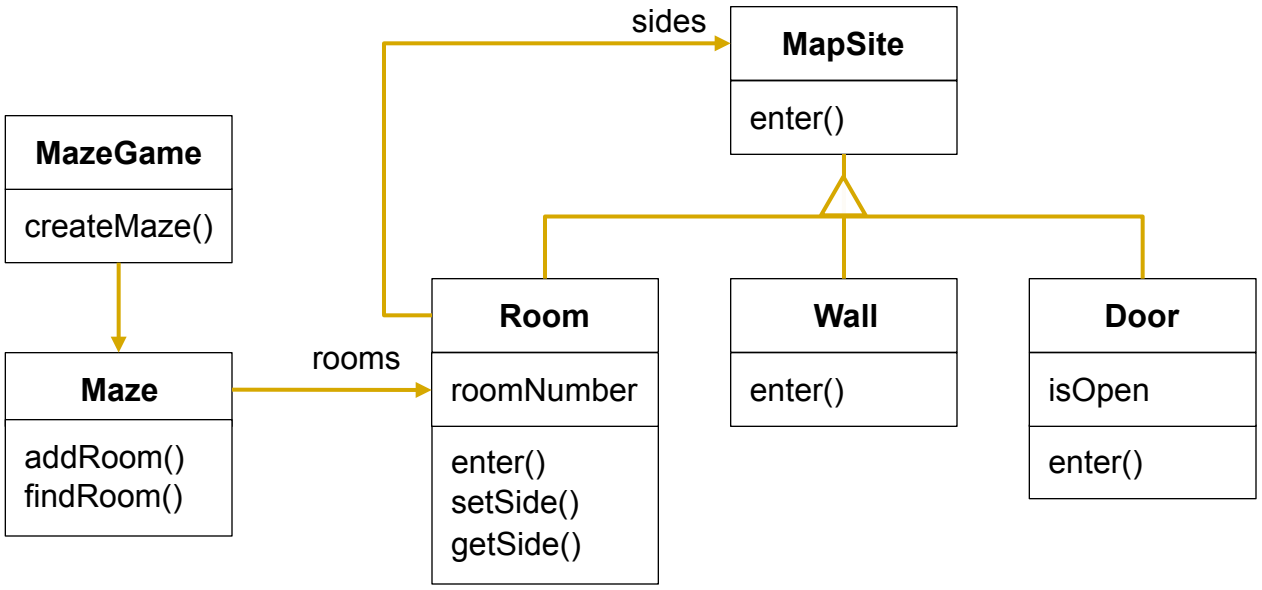


Code for Class MotifWindow




```
public class MotifWindow extends AbstractWindow {
  public MotifWindow() { }

  public void show() {
    JFrame frame = new JFrame();
    try {
      UIManager.setLookAndFeel(...);
    } catch (Exception e) {
      e.printStackTrace();
    }
    //updating the components tree after changing the LAF
    SwingUtilities.updateComponentTreeUI(frame);
    frame.setSize(300, 300);
    frame.setVisible(true);
  }
}
```

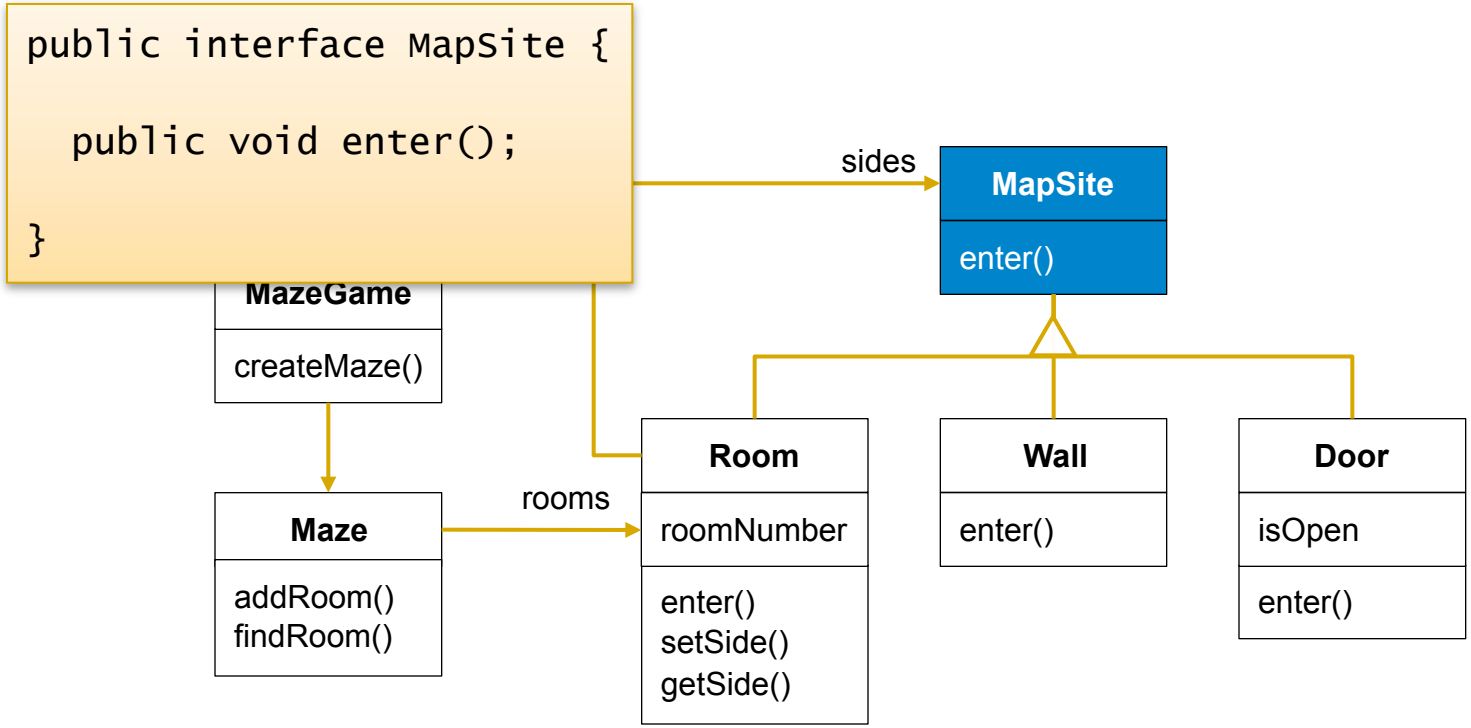
Example – Maze Game



Class Mapsite

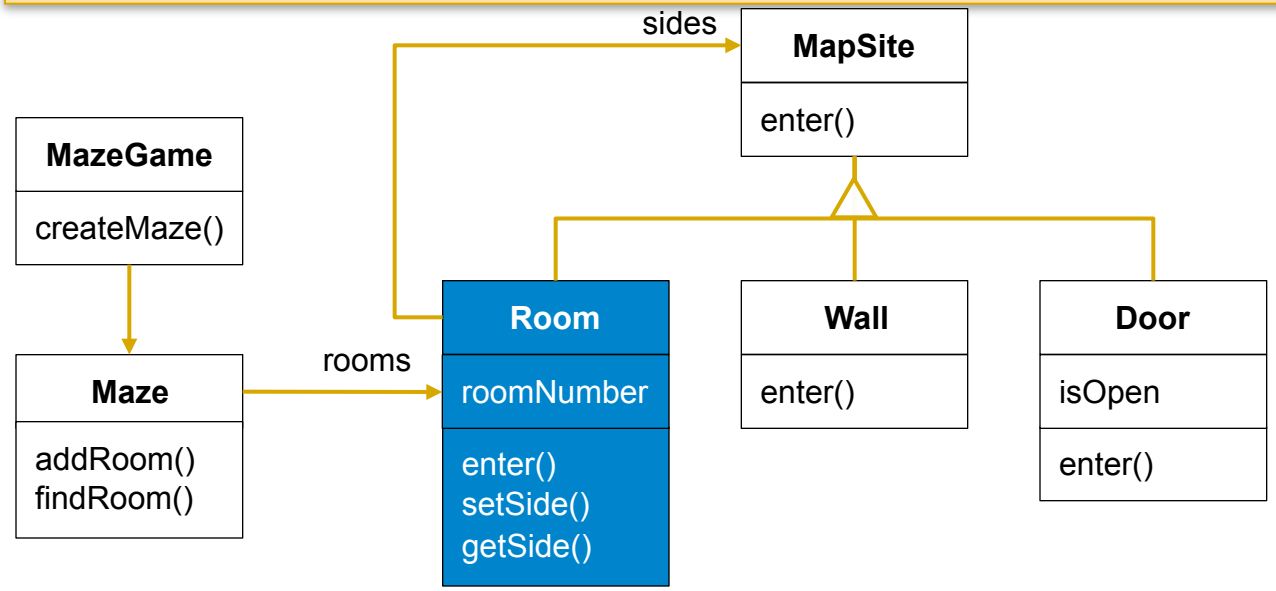
- Meaning of enter() depends on what you are entering
- room → location changes
- door → if door is open go in; else hurt your nose ;)
- wall → ouch

```
public interface MapSite {

  public void enter();

}
```



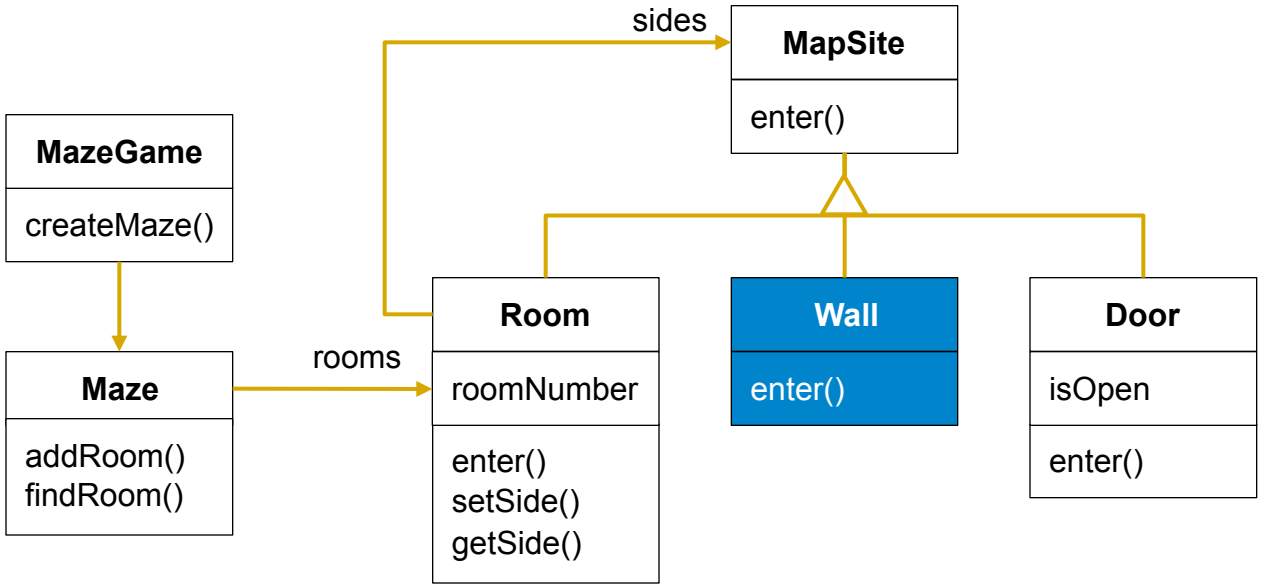
Class Room

```
public class Room implements MapSite {

  public Room(int roomNumber) {...}

  public MapSite getSide(Direction dir) {...}

  public void setSide(Direction dir, MapSite site) {...}

  public void enter() {...}

  protected int roomNumber = 0;
  protected MapSite[] sides = new MapSite[4];
}
```



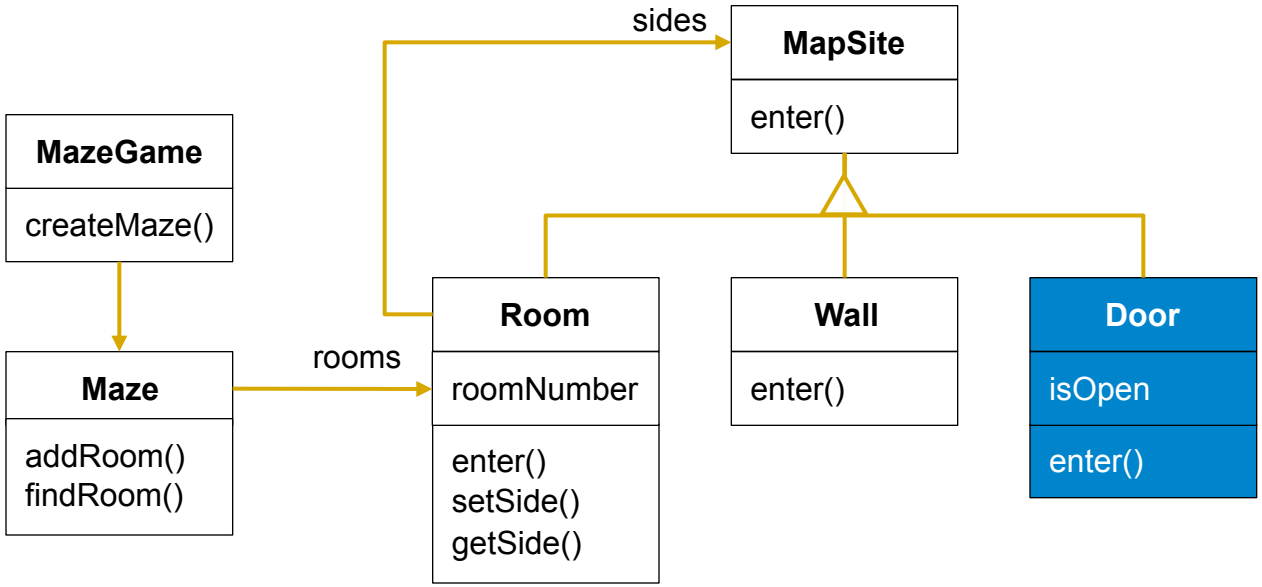
Class Wall

```
public class Wall implements MapSite {

  public void enter() {...}
}
```



Class Door

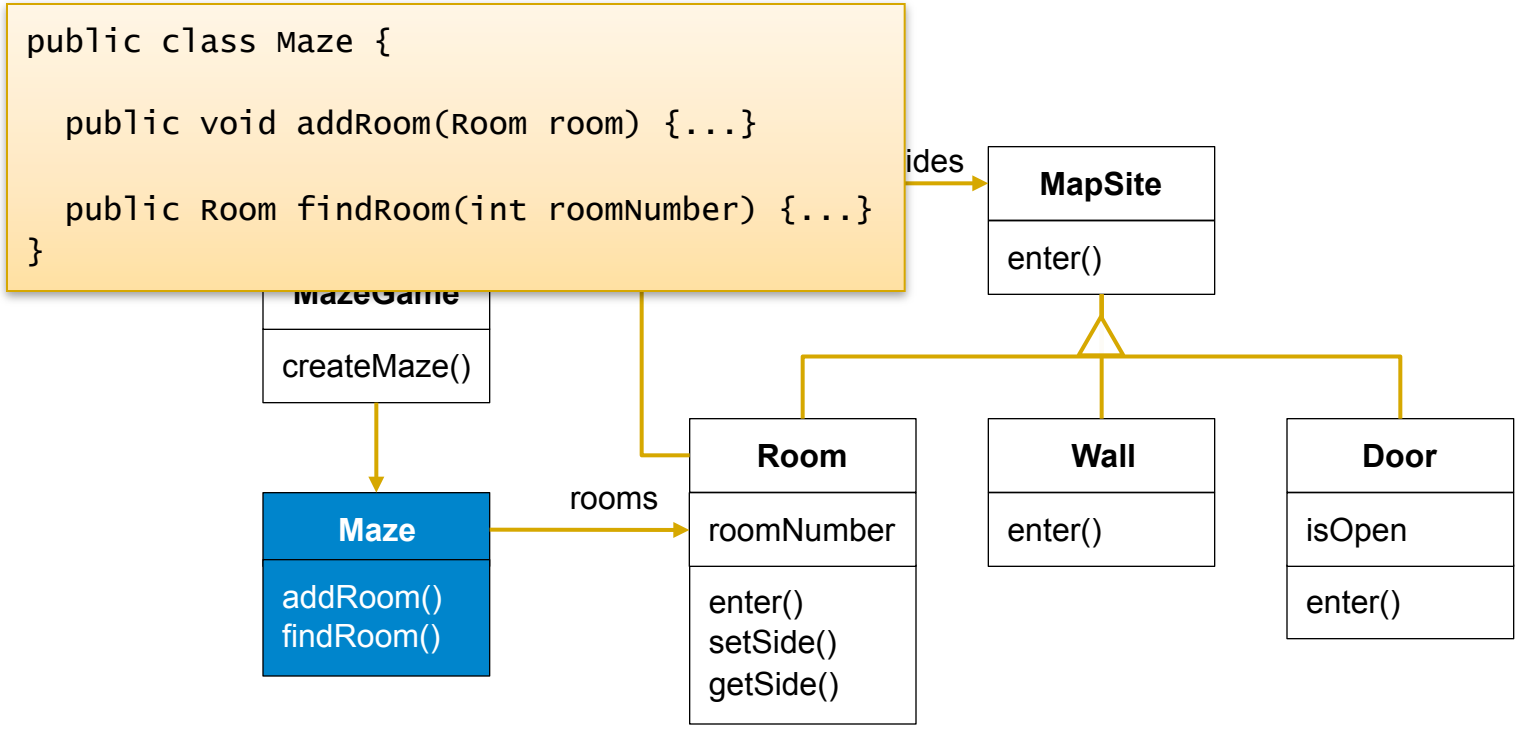
```
public class Door implements MapSite {

  public Door(Room room1, Room room2) {...}

  public Room otherSideFrom(Room room) {...}

  public void enter() {...}

  protected Room room1;
  protected Room room2;
  protected boolean open;
}
```



Class Maze

- A maze is a collection of rooms
- Maze can find a particular room given the room number
- `findRoom()` could do a lookup using a linear search or a hash table or a simple array

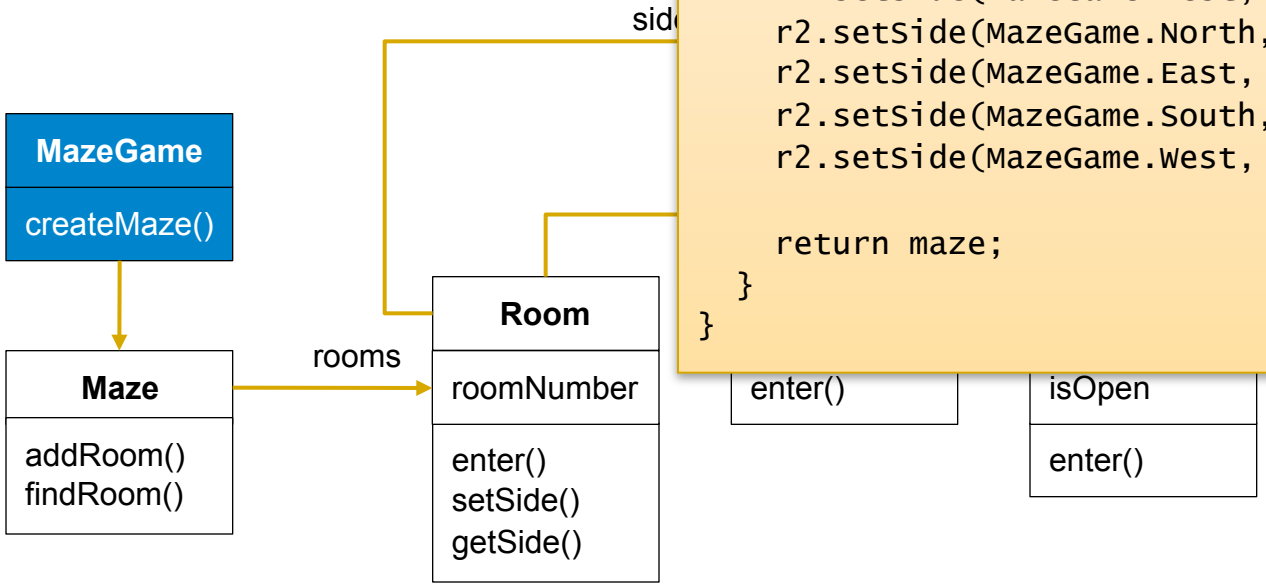
```
public class Maze {

  public void addRoom(Room room) {...}

  public Room findRoom(int roomNumber) {...}
}
```



Creating the Maze

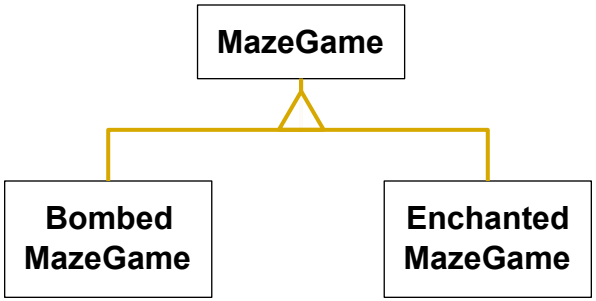
- The problem is inflexibility
  - Hard-coding of maze layout

```
public class MazeGame {
  public Maze createMaze() {
    Maze maze = new Maze();

    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door door = new Door(r1, r2);
    maze.addRoom(r1); maze.addRoom(r2);

    r1.setSide(MazeGame.North, new Wall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, new Wall());
    r1.setSide(MazeGame.West, new Wall());
    r2.setSide(MazeGame.North, new Wall());
    r2.setSide(MazeGame.East, new Wall());
    r2.setSide(MazeGame.South, new Wall());
    r2.setSide(MazeGame.West, door);

    return maze;
  }
}
```



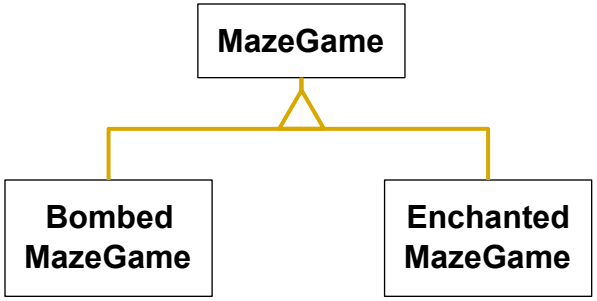
We Want Flexibility in Maze Creation

- Be able to vary the kinds of mazes
  - Rooms with bombs
  - Walls that have been bombed
  - Enchanted rooms
    - Need a password to enter the door!



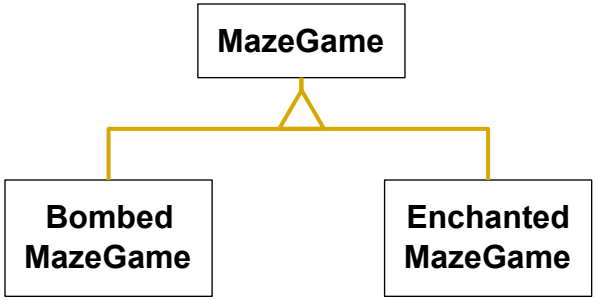
Idea 1: Subclass BombedMazeGame

```
public class BombedMazeGame extends MazeGame {
  public Maze createMaze() {
    Maze maze = new Maze();

    Room r1 = new RoomWithABomb(1);
    Room r2 = new RoomWithABomb(2);
    Door door = new Door(r1, r2);

    maze.addRoom(r1); maze.addRoom(r2);

    r1.setSide(MazeGame.North, new BombedWall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, new BombedWall());
    r1.setSide(MazeGame.West, new BombedWall());
    ...
```



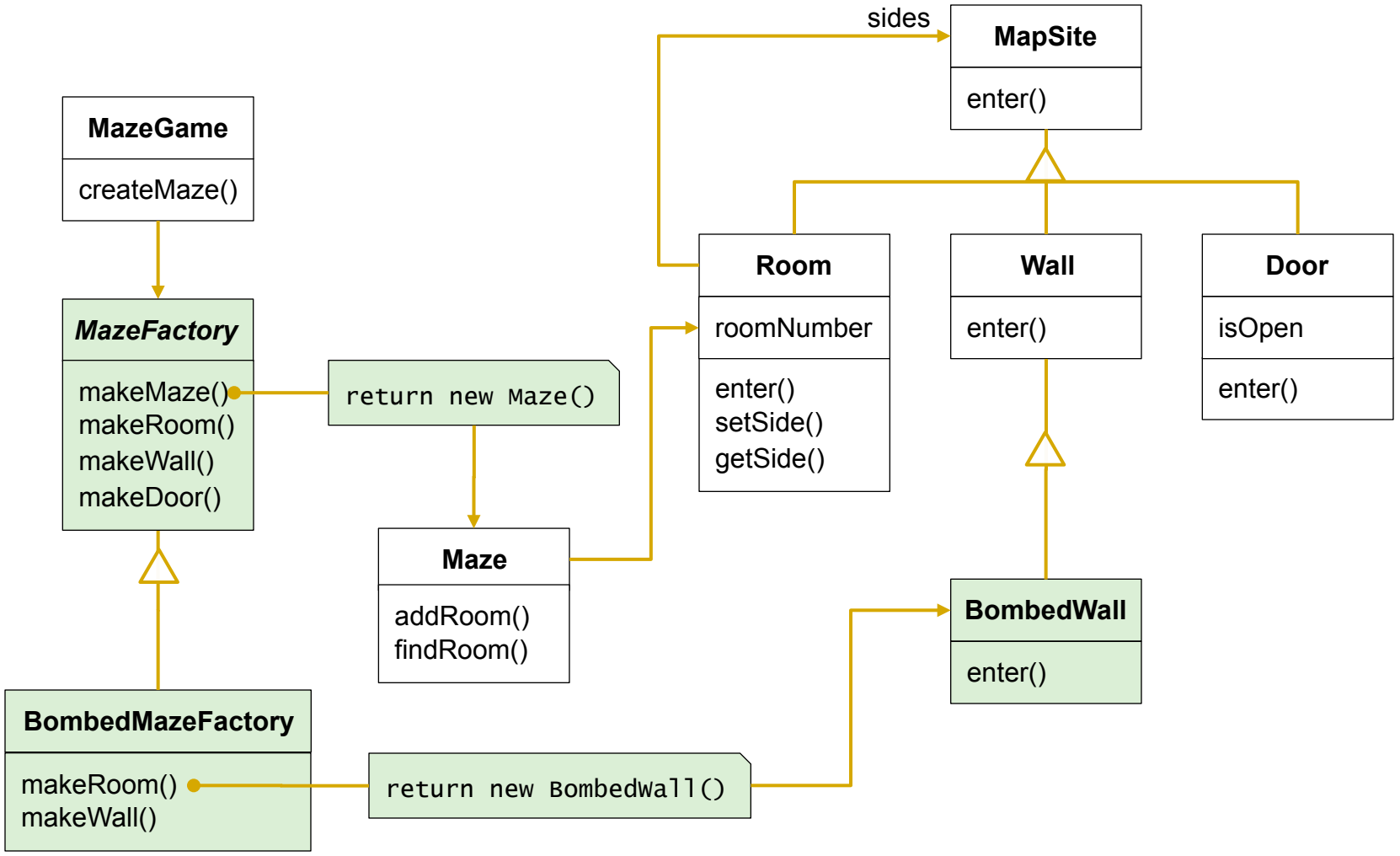
Subclass EnchantedMazeGame

```
public class EnchantedMazeGame extends MazeGame {
  public Maze createMaze() {
    Maze maze = new Maze();

    Room r1 = new EnchantedRoom(1);
    Room r2 = new EnchantedRoom(2);
    Door door = new DoorNeedingSpell(r1, r2);

    maze.addRoom(r1); maze.addRoom(r2);

    r1.setSide(MazeGame.North, new Wall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, new Wall());
    r1.setSide(MazeGame.West, new Wall());
    ...
```

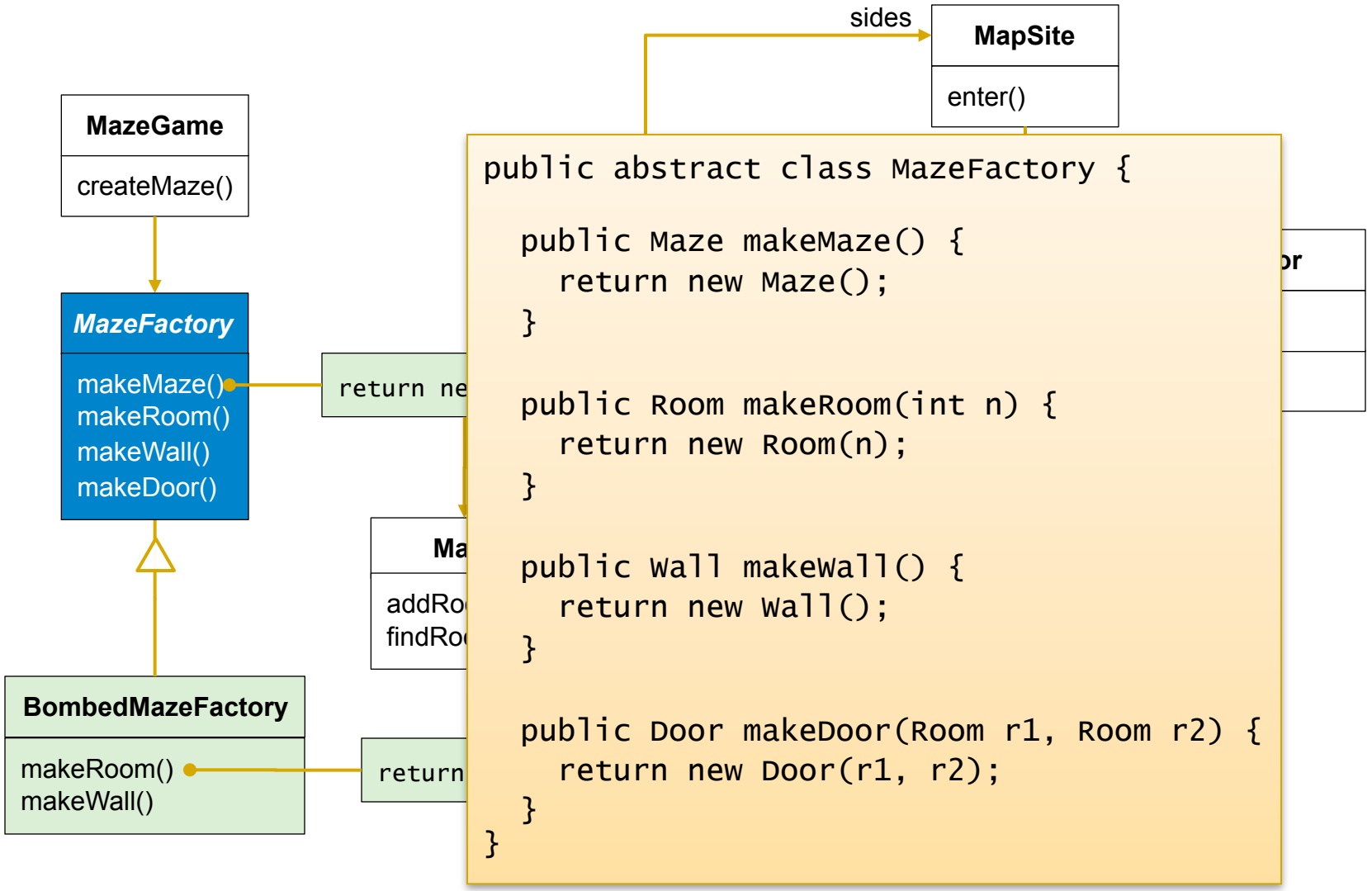


- Lots of code duplication for each type of maze... :((

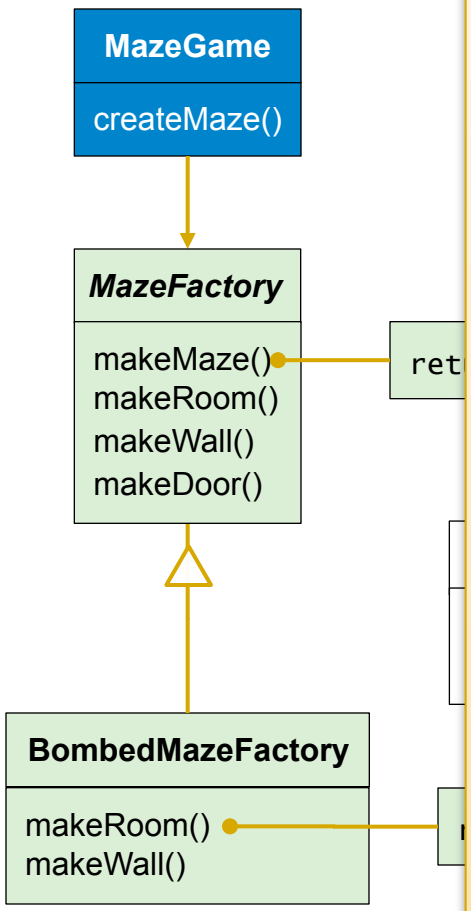
Idea 2: Use Abstract Factory



MazeFactory Abstract Class

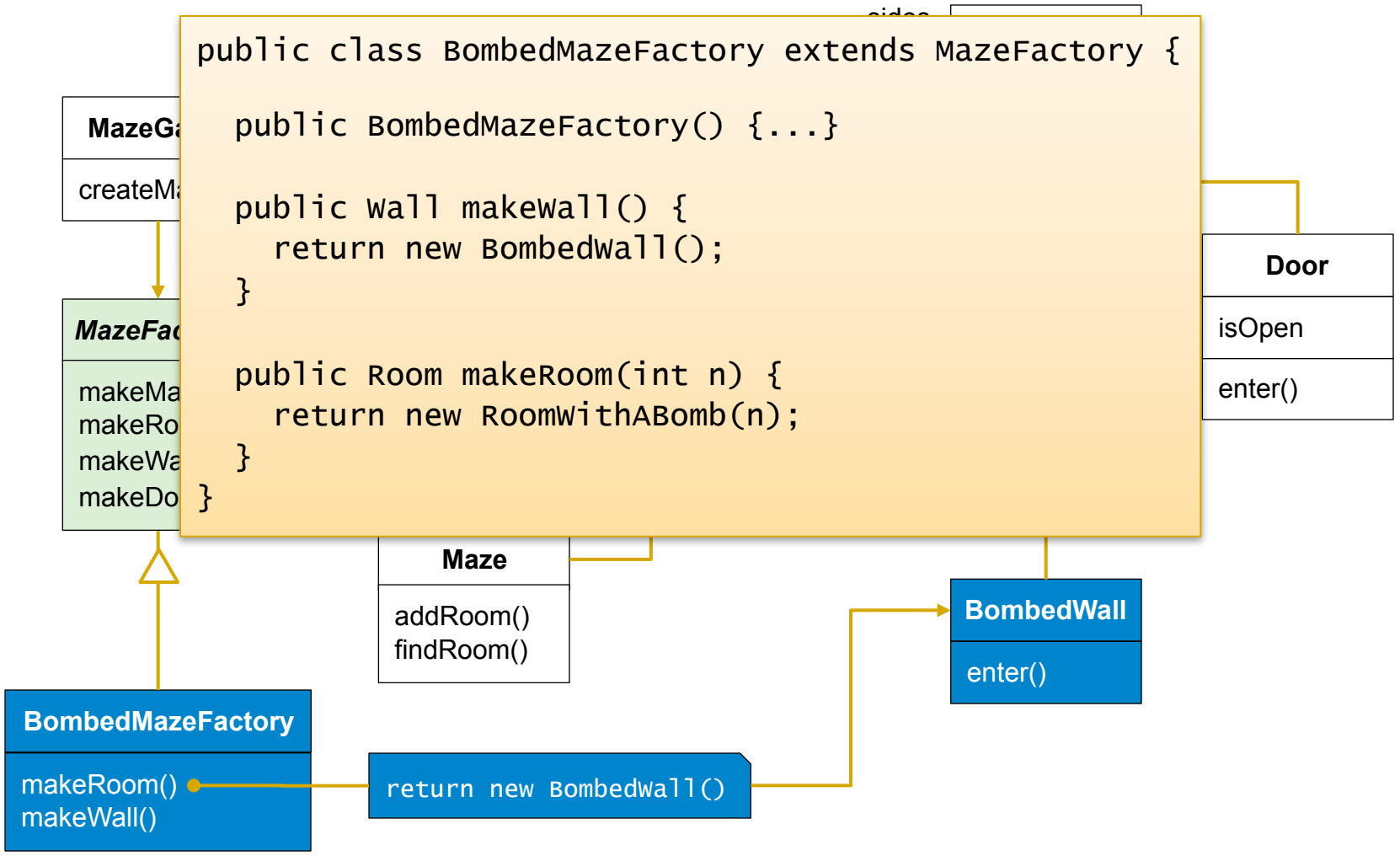


Modified Method createMaze




```
public class MazeGame {
  public Maze createMaze(MazeFactory factory) {
    Maze maze = factory.makeMaze();

    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door door = factory.makeDoor(r1, r2);
    maze.addRoom(r1); maze.addRoom(r2);

    r1.setSide(MazeGame.North, factory.makeWall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, factory.makeWall());
    r1.setSide(MazeGame.West, factory.makeWall());
    r2.setSide(MazeGame.North, factory.makeWall());
    r2.setSide(MazeGame.East, factory.makeWall());
    r2.setSide(MazeGame.South, factory.makeWall());
    r2.setSide(MazeGame.West, door);

    return maze;
  }
}
```

Subclass BombedMazeFactory



Subclass Enchanted Factory

```
public class EnchantedMazeFactory extends MazeFactory {

  public EnchantedMazeFactory() {...}

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

  public Door makeDoor(Room r1, Room r2) {
    return new DoorNeedingSpell(r1, r2);
  }
}
```

Localizing Change in One Place: the Instantiation

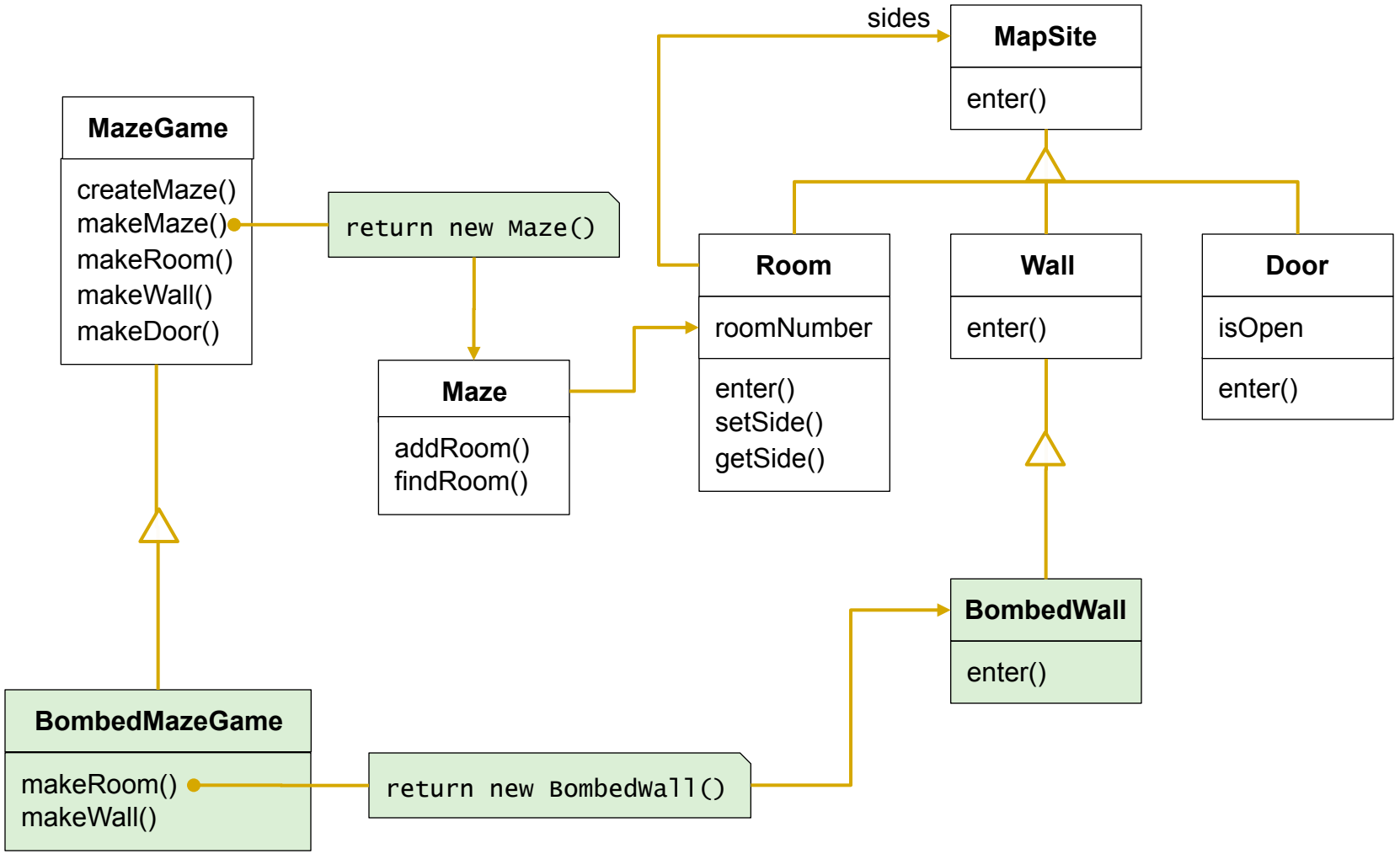
- To build an Enchanted Maze

```
MazeFactory factory = new EnchantedMazeFactory();
aMaze = game.createMaze(factory);
```

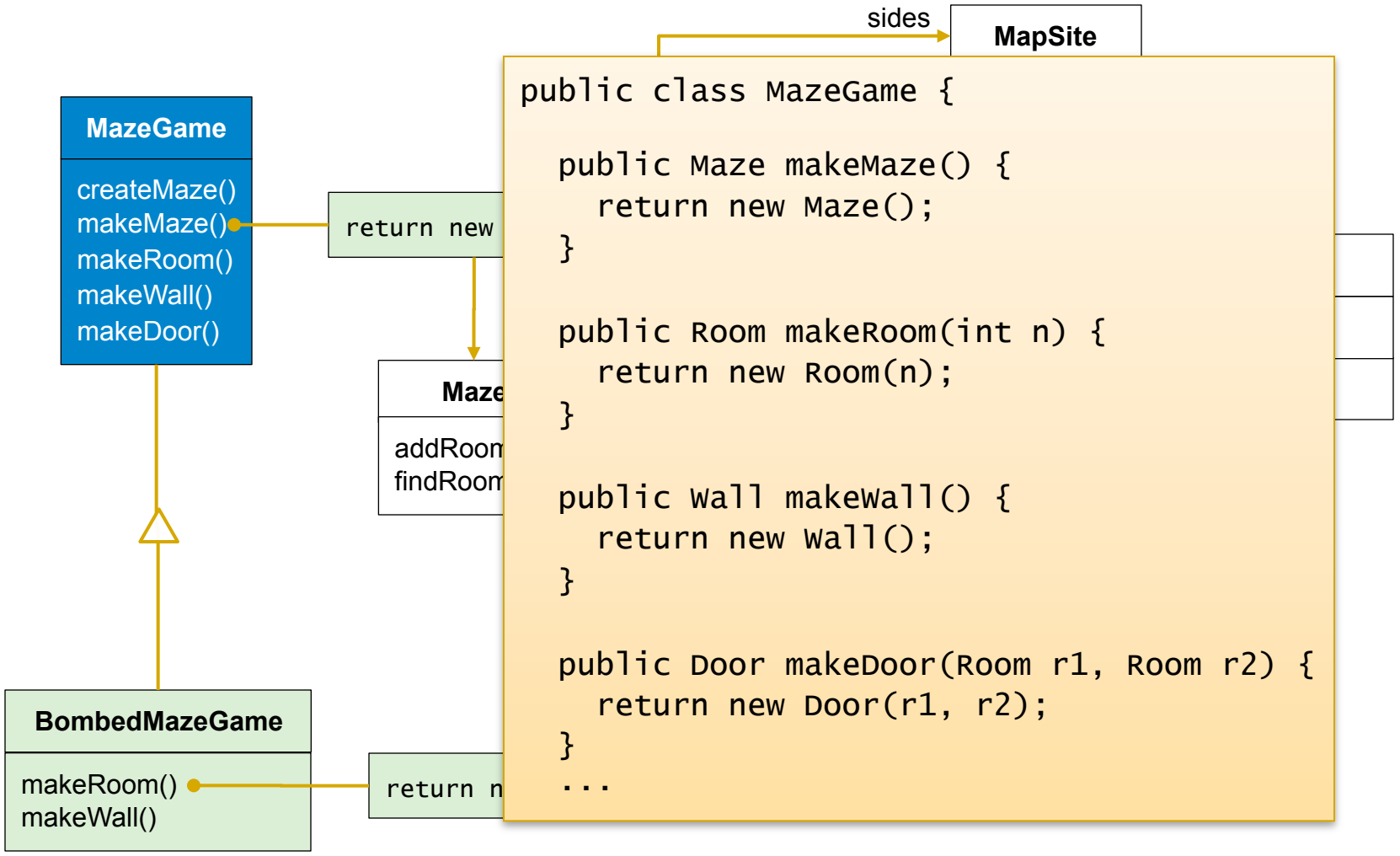
- To build a Bombed Maze

```
MazeFactory factory = new BombedMazeFactory();
maze = game.createMaze(factory);
```

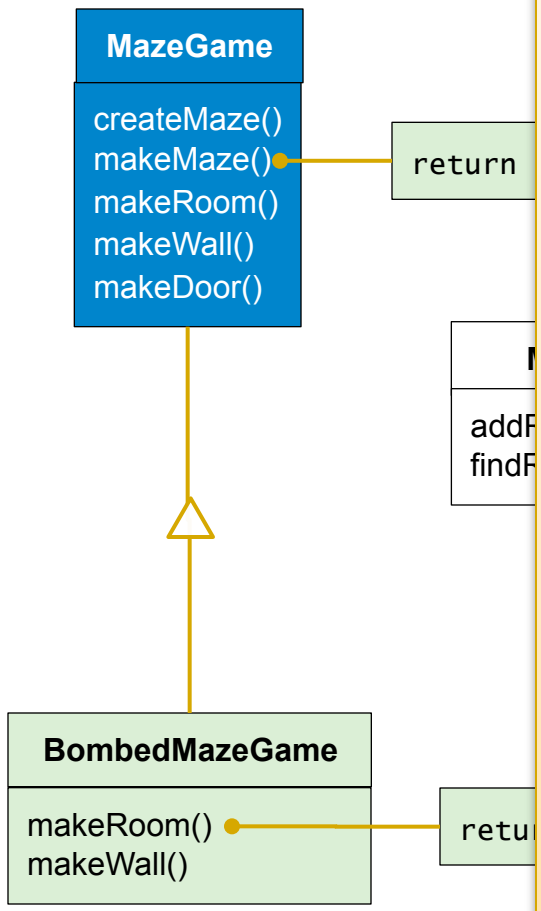
Idea 3: Use Factory Method



Factory Methods in MazeGame



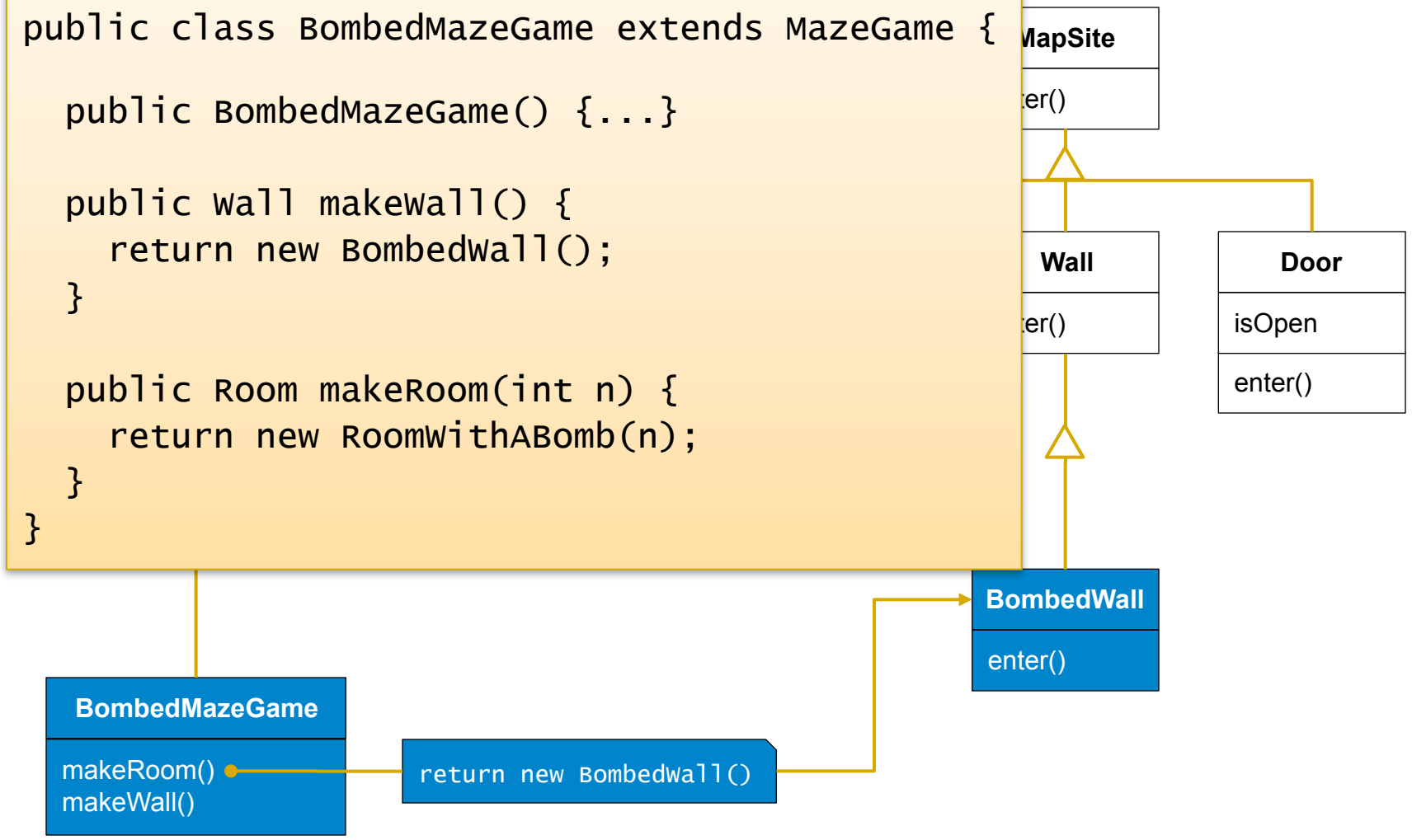
Modified Method createMaze




```
public class MazeGame {
  public Maze createMaze() {
    Maze maze = makeMaze();

    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door door = makeDoor(r1, r2);
    maze.addRoom(r1); maze.addRoom(r2);

    r1.setSide(MazeGame.North, makeWall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, makeWall());
    r1.setSide(MazeGame.West, makeWall());
    r2.setSide(MazeGame.North, makeWall());
    r2.setSide(MazeGame.East, makeWall());
    r2.setSide(MazeGame.South, makeWall());
    r2.setSide(MazeGame.West, door);

    return maze;
  }
}
```

Subclass BombedMazeGame

```
public class BombedMazeGame extends MazeGame {

  public BombedMazeGame() {...}

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

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



Abstract Factory vs. Factory Method

- In Abstract Factory, a class delegates the responsibility of object instantiation to another one via composition
- The Factory Method pattern uses inheritance to handle the desired object instantiation