CptS 487
Software Design and Architecture

Lesson 4

Design Patterns 1:

Factory Method

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Instructor: Bolong Zeng

Before you start…

- Read the FactoryPattern_handout.pdf file before proceeding with these slides.

Outline

- Introduction on Design Patterns
- Factory Patterns
  - Factory Method
  - Abstract Factory
- More on Factory Method

Intro on Design Patterns

- Proposed “patterns” that are useful in common problems.
  - Classic Categories: creational, structural, behavioral.
    - What we’ll cover for this semester.
  - Customized patterns also widely adopted in the industry.

Intro on Design Patterns

- Key issues
  - Motivation
  - Structure
    - typically represented by a class diagram
  - Participants
  - Applicability
  - Benefits & Drawbacks
  - Collaborations

Factory Patterns

- Factory patterns are examples of creational patterns
- Creational patterns abstract the object instantiation process
- They hide how objects are created and help make the overall system independent of how its objects are created and composed.
- Class creational patterns focus on the use of inheritance to decide the object to be instantiated
  - Factory Method
- Object creational patterns focus on the delegation of the instantiation to another object
  - Abstract Factory

Factory Patterns

- Creating objects:
  - What's wrong with "new newClass()"?
  - Compare these two:
    - String name = new String();
    - Student newStudent = new Student(name);

Factory Patterns

- All OO languages have an operator for object creation. In Java this idiom is the **`new`** operator.
- Creational patterns allow us to write methods that create new objects without explicitly using the `new` operator. This allows us to write methods,
  - that can instantiate different objects and
  - that can be extended to instantiate other newly-developed objects,

all without modifying the method's code!

Factory Patterns

- Creating objects:
  - What's wrong with "new newClass()"?
  - Compare these two:
    - String name = new String();
    - Student newStudent = new Student(name);

Detailed Explanation

- http://www.oodesign.com/factory-method-pattern.html

Factory Method Example - 1

- PizzaStore Example
  - We have a pizza store program that wants to separate the process of creating a pizza from the process of preparing/ordering a pizza
  - There are different franchises for pizza that exist in different parts of the country
  - Each franchise needs its own factory to match the preferences of the locals
  - However, we want to retain the preparation process for the original PizzaStore (since they are uniformed).
  - To conclude: the store program should be able to produce different styles of pizzas w.r.t different franchises/cities.

Factory Method Example - 1

Initial Code: mixes creation and preparation

```
public class PizzaStore {

    Pizza orderPizza(String type) {

        Pizza pizza;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek")) {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Creation
needs to be recompiled each time a new pizza type is added or removed

Preperation

Factory Method Example - 1

```
public class PizzaStore {

   public PizzaStore() {}

   public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
   }
}
```

```
public Pizza createPizza(String type){
    if (type.equals("cheese")) {
            return new CheesePizza();
    } else if (type.equals("greek")) {
            return new GreekPizza();
    } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
    }
 }
```

- Let's separate the creation code.

Factory Method Example - 1

- `PizzaStore` becomes an abstract class (Creator) with an abstract `createPizza()` method (factory method)
- We then create subclasses that override `createPizza()` for each franchise

```
public abstract class PizzaStore {

    protected abstract Pizza createPizza(String
    type);

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

}
```

Factory Method
We would subclass this class (for each franchise) and provide an implementation of the `createPizza()` method.
Any dependencies on concrete "product" classes are encapsulated in the subclass.

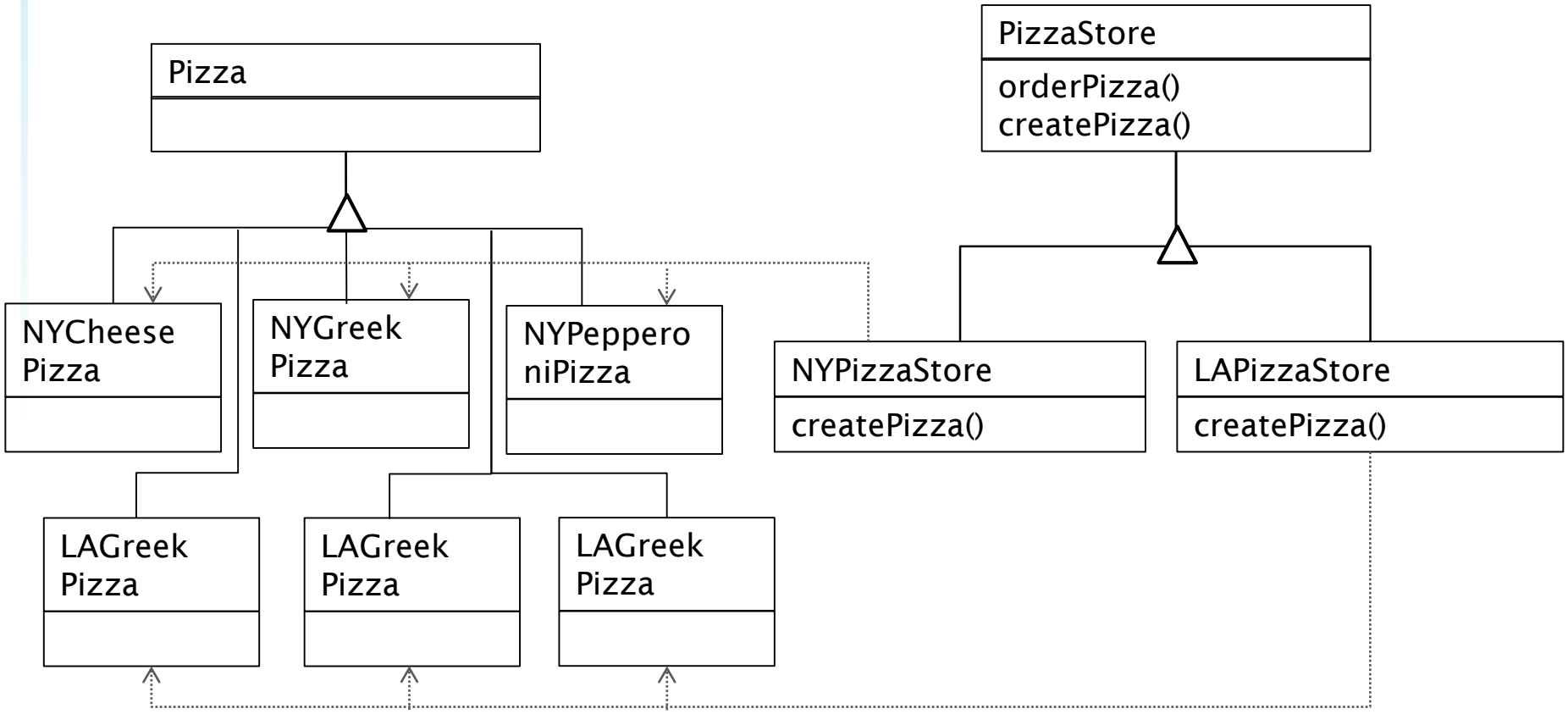
Factory Method Example - 1

```
public class NYPizzaStore extends PizzaStore {

    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
                return new NYCheesePizza();
        } else if (type.equals("greek")) {
                return new NYGreekPizza();
        } else if (type.equals("pepperoni")) {
                return new NYPepperoniPizza();
        } else
                return null;
        }

}
```

- If you want a NYStyle pizza, you create an instance of this class and call `orderPizza()` passing in the type. The subclass makes sure that the pizza is created using the correct style.

Factory Method Example - 1

- Creator => PizzaStore
- ConcreteCreator => LAPizzaStore, NYPizzaStore
- Product => Pizza
- ConcreteProduct => NYCheesePizza, NYGreekPizza, NYPepperoniPizza, LACheesePizza, LAGreekPizza, LAPepperoniPizza



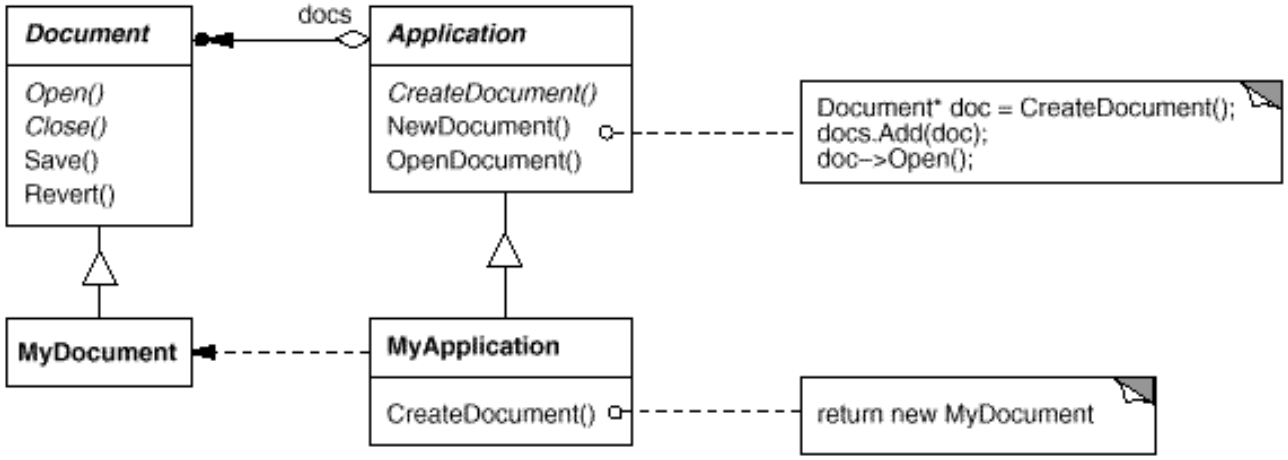
Factory Method Example - 1

```
public class NYPizzaStore extends PizzaStore {

    public Pizza createPizza() {
        return new NYPizza();
    }

}
```

- If you want a NYStylePizza, you create an instance of this class and call `orderPizza()`. The subclass makes sure that the pizza is created using the correct style.

1. Factory Method (Class creational pattern)

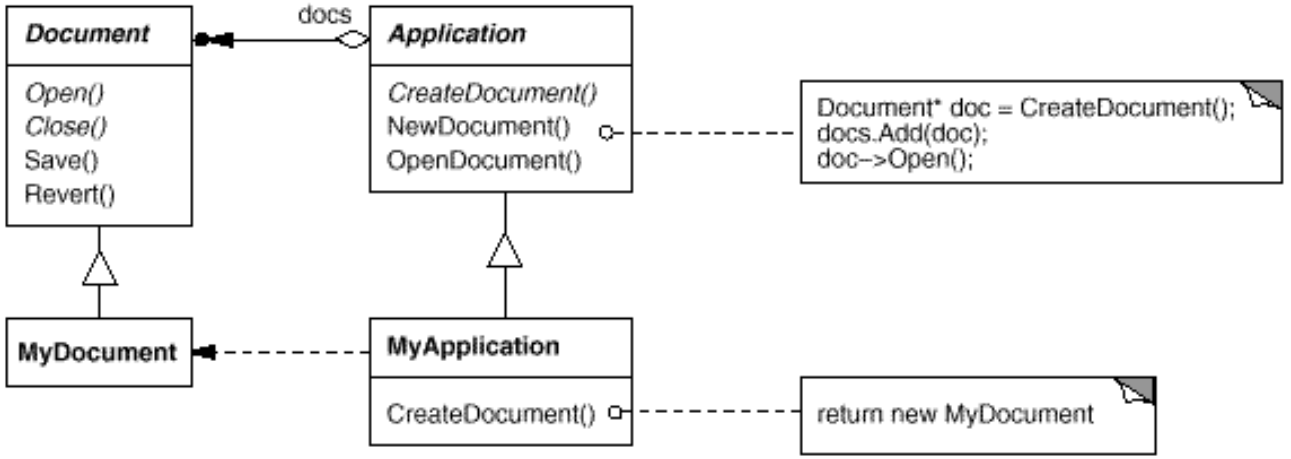
- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Motivation
  - Consider the following framework:



  - The createDocument() method is a factory method.

1. Factory Method (Class creational pattern)

- Motivation
  - Consider the following framework:

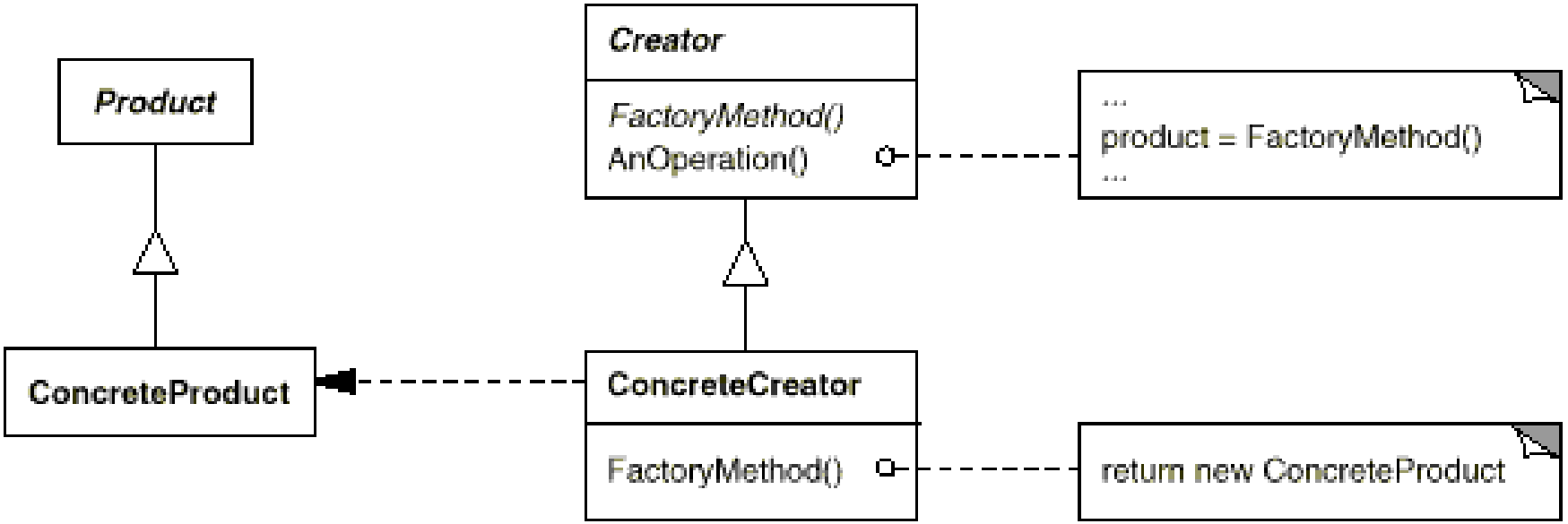


  - The createDocument() method is a factory method.

- Applicability
  - Use the Factory Method pattern in any of the following situations:
    - A class can't anticipate the class of objects it must create
    - A class wants its subclasses to specify the objects it creates

1. Factory Method

- Structure



1. Factory Method

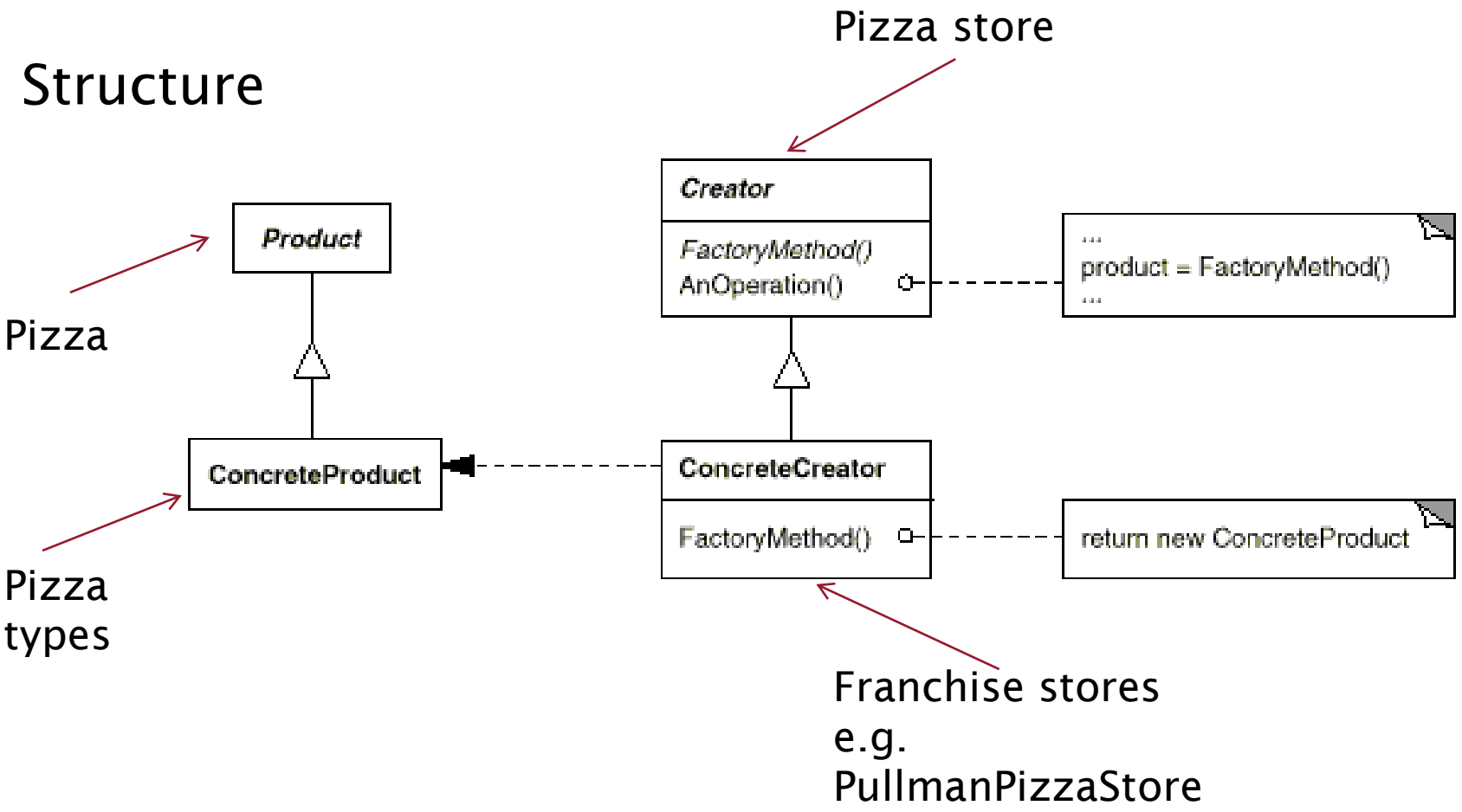
- What does the following sentence mean:
  - "the Factory Method Pattern lets subclasses decide which class to instantiate?"

- It means that Creator class is written without knowing what actual ConcreteProduct class will be instantiated. The ConcreteProduct class which is instantiated is determined solely by which ConcreteCreator subclass is instantiated and used by the application.

- It does *not mean that somehow the subclass decides at runtime which* ConreteProduct class to create

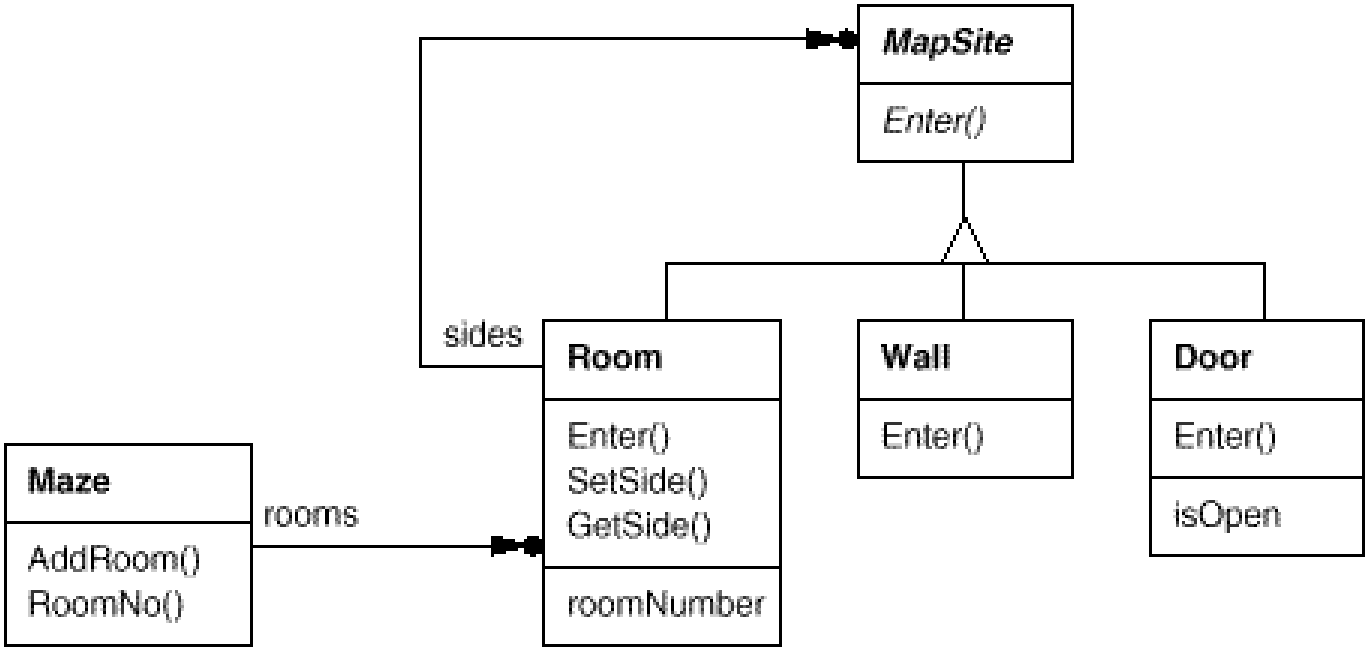
1. Factory Method

- Structure



Factory Method Example - 2

- Consider this maze game:



Factory Method Example - 2

- Here's a MazeGame class with a `createMaze()` method:

```
/*** MazeGame.*/
public class MazeGame {
  // Create the maze.
  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;
  }
}
```

Create a new Maze

Create new Rooms

Create new Door

Create new Walls

Factory Method Example - 2

- The problem with this `createMaze()` method is its *inflexibility*.
- What if we wanted to have enchanted mazes with EnchantedRooms and EnchantedDoors? Or a secret agent maze with DoorWithLock and WallWithHiddenDoor?
- What would we have to do with the `createMaze()` method?
  - As it stands now, we would have to make significant changes to it because of the explicit instantiations using the *new operator of the* objects that make up the maze.
  - How can we redesign things to make it easier for `createMaze()` to be able to create mazes with new types of objects?

Factory Method Example - 2

- Let's add factory methods to the `MazeGame` class:

```
/*** MazeGame with Factory methods.*/

public class MazeGame {

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

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

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

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

Factory Method Example - 2

```
/*** MazeGame class continued.*/
  ...
  // Create the maze with the Factory
  methods
  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;
  }
}
```


— We made `createMaze()` just slightly more complex, but a lot more flexible!

Factory Method Example - 2

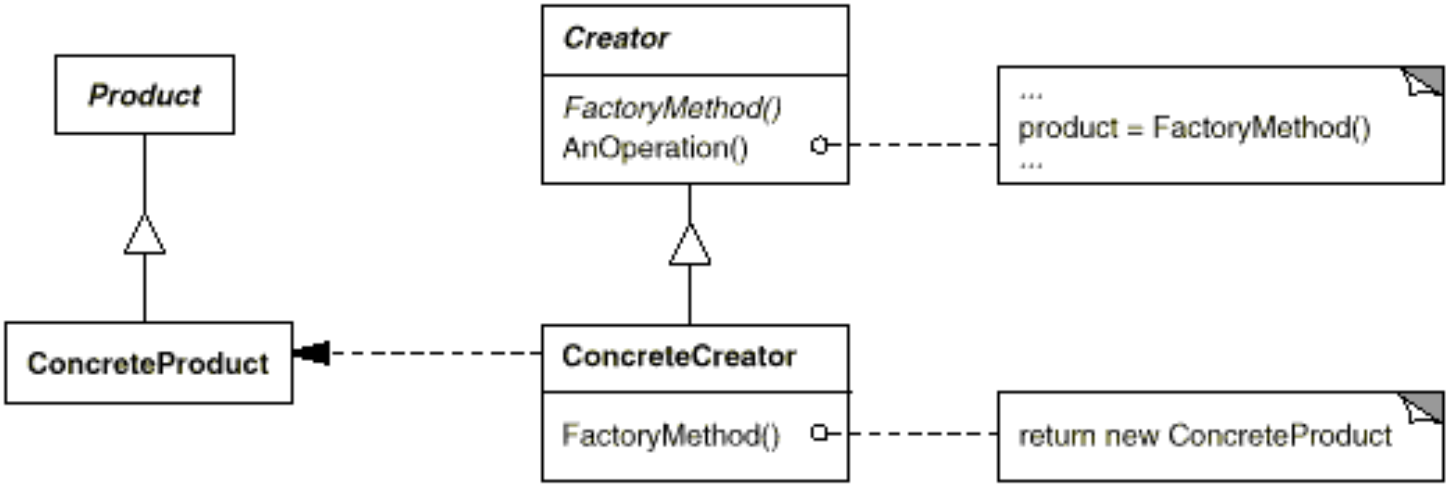
- Consider these `EnchantedMazeGame` and `Bombed MazeGame` classes:

```
public class EnchantedMazeGame extends MazeGame {
    public Room makeRoom(int n)
                {return new EnchantedRoom(n);}
    public Wall makeWall()
                {return new EnchantedWall();}
    public Door makeDoor(Room r1, Room r2)
                {return new EnchantedDoor(r1, r2);}
}
public class BombedMazeGame extends MazeGame {
    public Room makeRoom(int n)
                {return new RoomWithABomb(n);}
    public Wall makeWall()
                {return new BombedWall();}
}
```

- The `createMaze()` method of `MazeGame` is inherited by `EnchantedMazeGame` and `BombedMazeGame`, and can be used to create regular mazes, enchanted mazes, or bombed mazes *without modification!*

Factory Method Example - 2

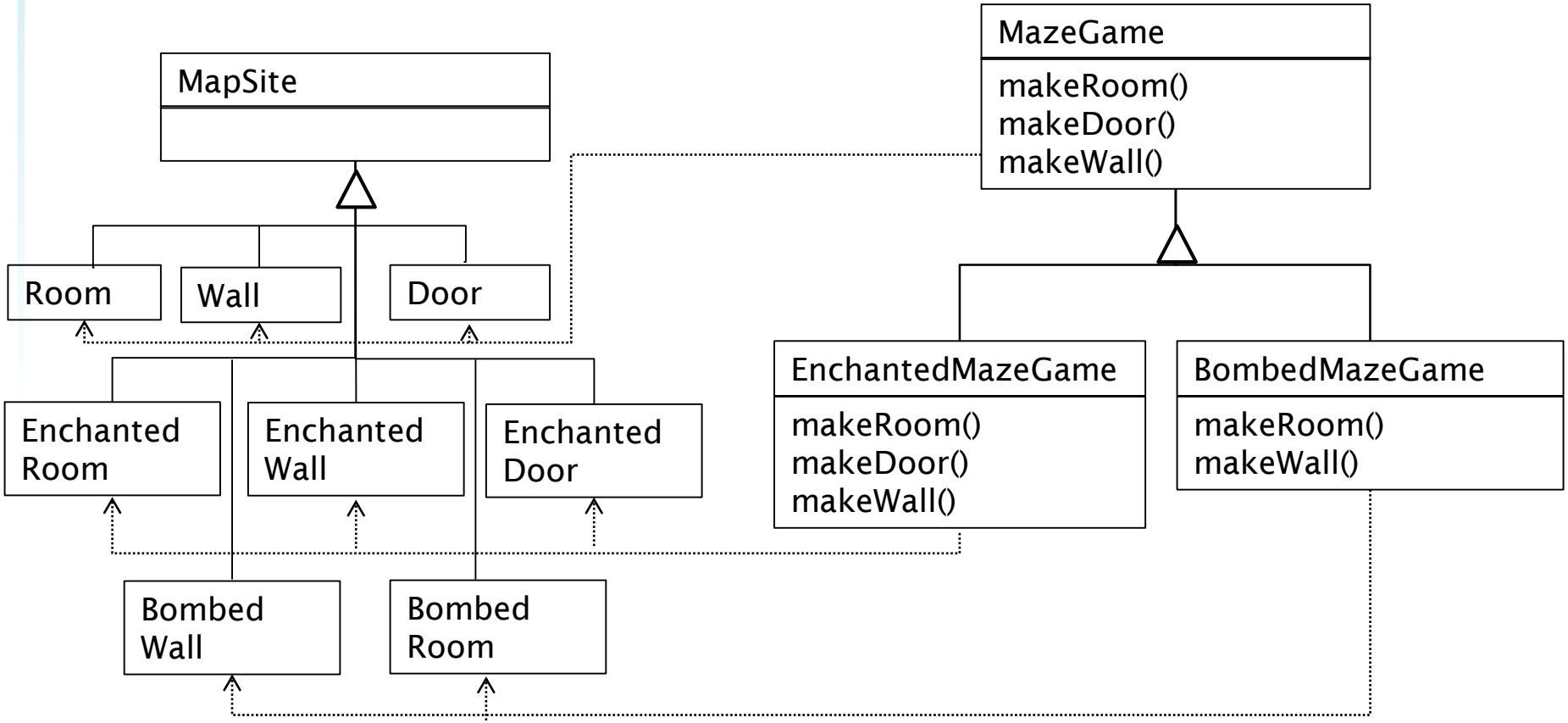
- `createMaze()` method of `MazeGame` defers the creation of maze objects to its subclasses.
  - That's the Factory Method pattern at work!



- In this example, the correlations are:
  - Creator => MazeGame
  - ConcreteCreator => EnchantedMazeGame, BombedMazeGame (MazeGame is also a ConcreteCreator)
  - Product => MapSite
  - ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor, BombedWall, BombedRoom

Factory Method Example - 2

- Creator => MazeGame
- ConcreteCreator => EnchantedMazeGame, BombedMazeGame, (MazeGame is also a ConcreteCreator)
- Product => MapSite
- ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor, BombedWall, BombedRoom



Factory Method

- Consequences
  - Benefits
    - Code is made more flexible and reusable by the elimination of instantiation of application-specific classes
    - Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface
  - Liabilities
    - Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct
- Implementation Issues
  - Creator can be abstract or concrete
  - If the factory method should be able to create multiple kinds of products, then the factory method has a parameter (possibly used in an if-else!) to decide what object to create.