Design Patterns
Factories
Singleton

Laura Semini, Ingegneria del Software

Dipartimento di Informatica, Università di Pisa

Dovreste aver già visto dei metodi factory

INETADDRESS: FACTORY METHODS

- metodi statici di una classe che restituiscono oggetti di quella classe
- i seguenti metodi contattano il DNS per la risoluzione di indirizzo/hostname

```
static InetAddress getLocalHost() throws UnknownHostEception
static InetAddress getByName (String hostname) throws UnknownHostException
static InetAddress [] getAllByName (String hostName)
       throws UnknownHostException
static InetAddress getLoopBackAddress()
```

- i seguenti metodi statici costruiscono oggetti di tipo `InetAddress`, ma non contattano il `DNS` (utile se `DNS` non disponibile e conosco indirizzo/host)
- nessuna garanzia sulla correttezza di hostname/IP, `UnknownHostException` sollevata solo se l'indirizzo è malformato

```
static InetAddress getByAddress(byte IPAddr[]) throws UnknownHostException
static InetAddress getByAddress (String hostName, byte IPAddr[])
                                        throws UnknownHostException
```

Pattern creazionali: le factories

Factory: a class whose sole job is to easily create and return instances of other classes

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.
- They make it easier to construct complex objects instead of calling a constructor, use a method in a "factory" class to set up the object saves lines and complexity to quickly construct / initialize objects

examples in Java:

- borders (BorderFactory),
- key strokes (KeyStroke),
- network connections (SocketFactory)

The Problem With “New”

Each time we invoke the “new” command to create a new object, we violate the “Code to an Interface” design principle

Example

- List list = new ArrayList()

Even though our variable’s type is set to an “interface”, in this case “List ”, the class that contains this statement depends on “ArrayList”

In addition

- if you have code that checks a few variables and instantiates a particular type of class based on the state of those variables, then the containing class depends on each referenced concrete class
  - if (condition) { return new ArrayList(); }
    else { return new LinkedList();}

- Obvious Problems: needs to be recompiled if classes change
  - add new classes → change this code
  - remove existing classes → change this code

- This means that this code violates the open-closed and the information hiding design principles

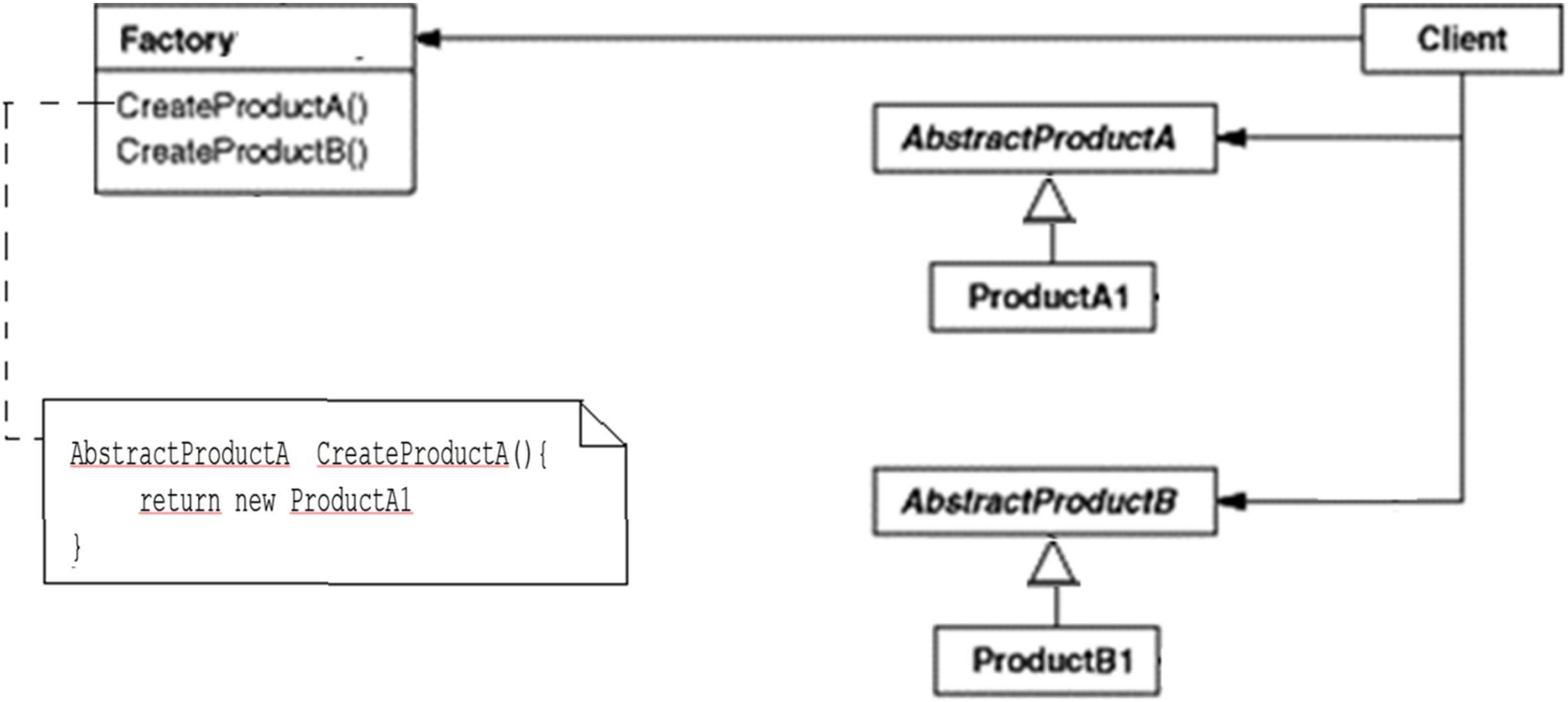
Vedremo 3 tipi di pattern Factories

1. Simple Factory (detto anche Concrete Factory)
   - non è un pattern GoF
   - è una semplificazione molto diffusa di Abstract Factory
2. Abstract Factory
3. Factory Method

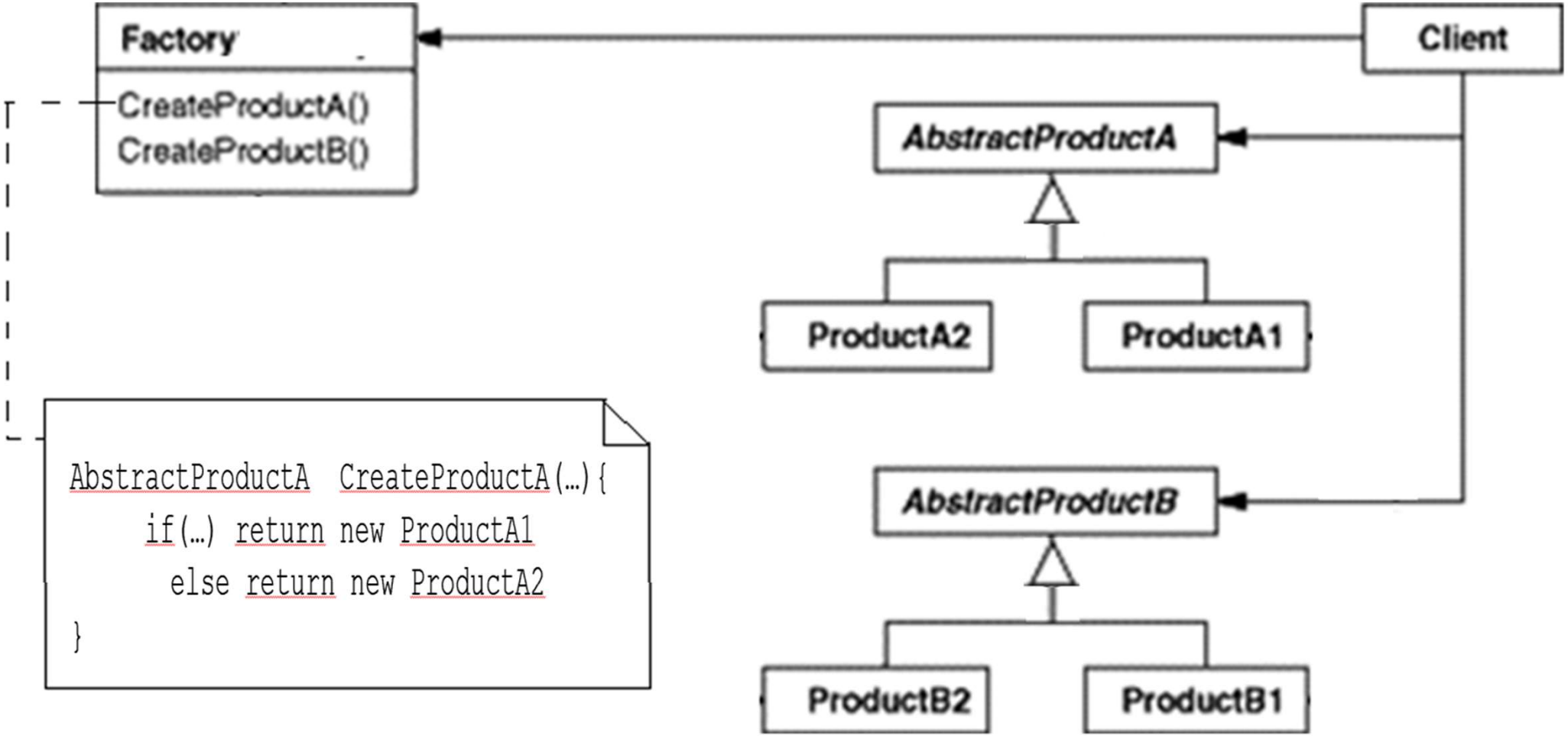
Simple Factory (aka concrete factory)

- Problema
  - Chi deve essere responsabile di creare gli oggetti quando la logica di creazione è complessa e si vuole separare la logica di creazione dalle altre funzionalità di un oggetto?
- Soluzione
  - La delega a un oggetto chiamato Factory che gestisce la creazione

Simple(st) Factory: structure



Another simple Factory: structure



Example: Consider a pizza store that makes different types of pizzas

```
public class PizzaStore {

Pizza orderPizza(String type){

    Pizza pizza;

    If (type == CHEESE)
            pizza = new CheesePizza();
      else if (type == PEPPERONI)
            pizza = new PepperoniPizza();
         else if (type == PESTO)
            pizza = new PestoPizza();

    pizza.prepare();
    pizza.bake();
    pizza.package()
    pizza.deliver();
    return pizza
  }
}
```

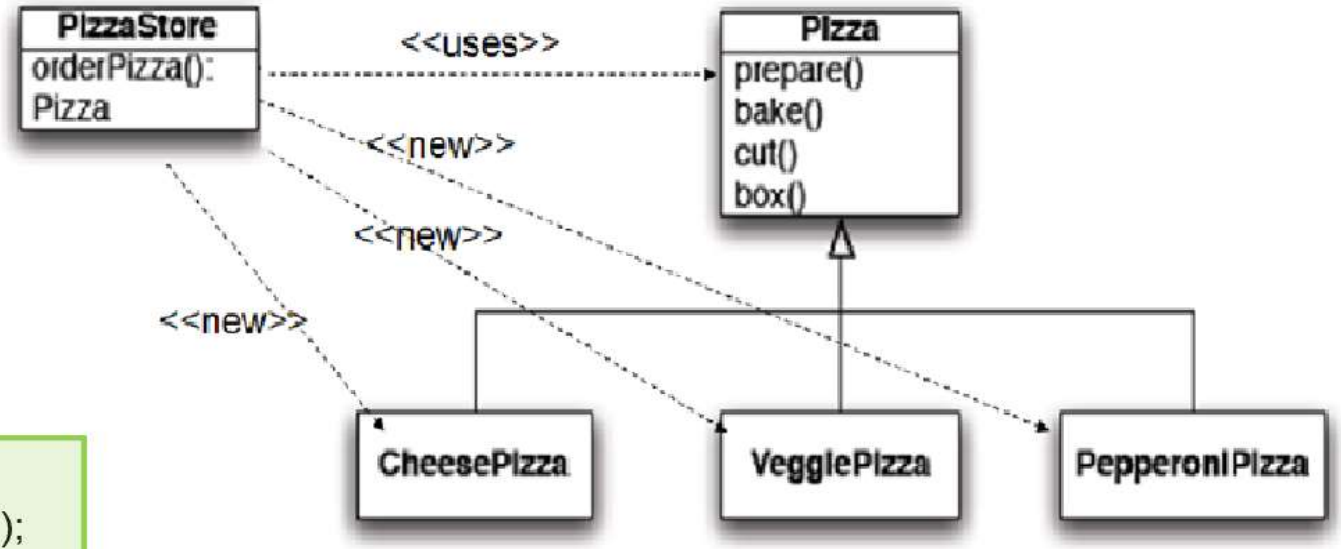
This is creation code: it becomes cumbersome as we add to our menu

This is the preparation of the pizza, this part stays the same (independent on the pizza type)

Idea: pull out the creation code and put it into an object that only deals with creating pizzas - the PizzaFactory

Class diagram

```
Pizza orderPizza(String type){

 Pizza pizza;

 If (type == CHEESE)
            pizza = new CheesePizza();
 else if (type == PEPPERONI)
            pizza = new PepperoniPizza();
 else if (type == PESTO)
            pizza = new PestoPizza();
 pizza.prepare();
 pizza.bake();
 pizza.package();
 pizza.deliver();
 return pizza
  }
}
```



Simple solution: a SimplePizzaFactory

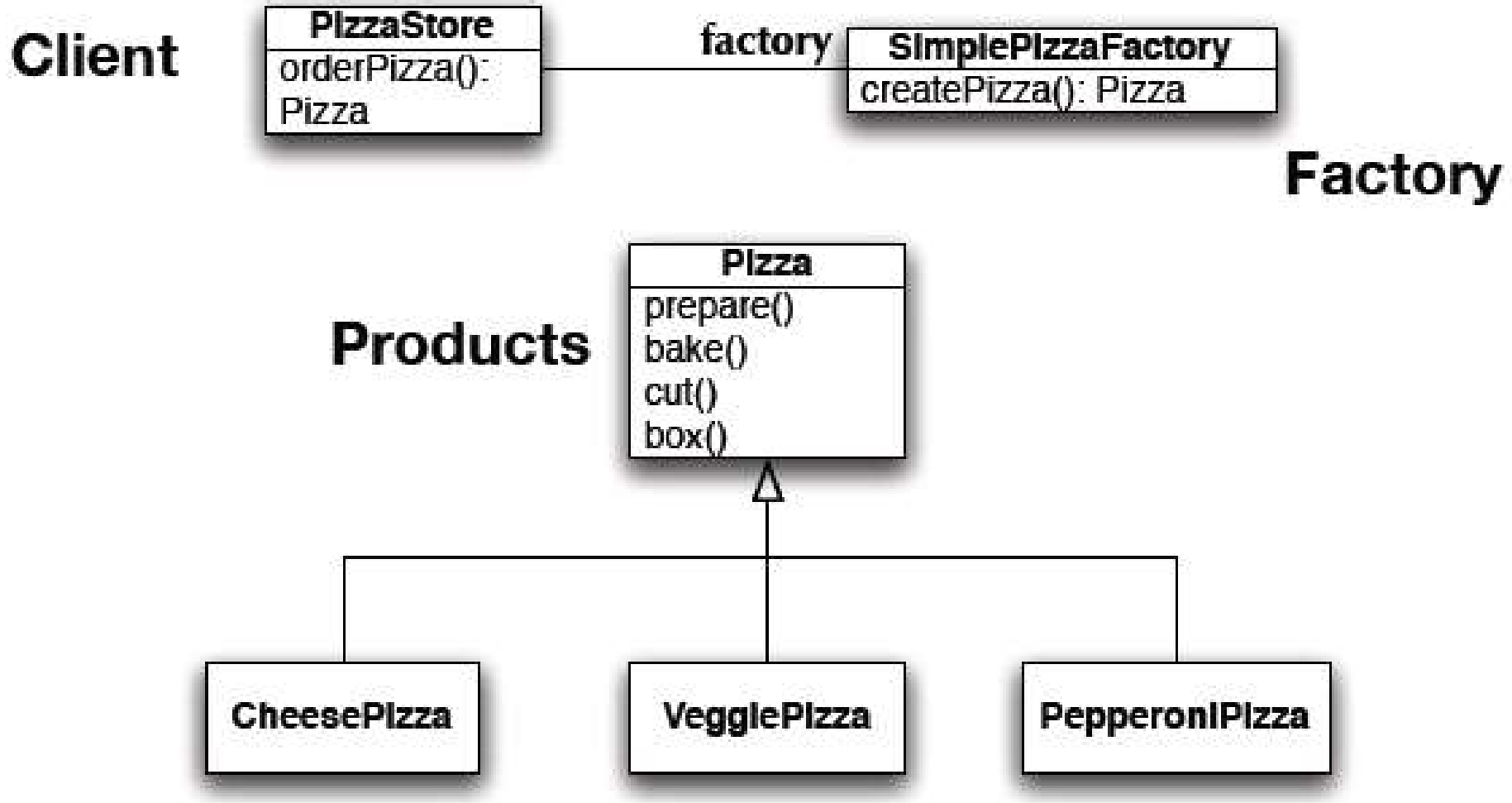
```
public class PizzaStore {
  private SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
    }
  public Pizza orderPizza(String type) {
    Pizza pizza = factory.createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.package();
    pizza.deliver();
    return pizza
    }
}
```

```
public class SimplePizzaFactory {
  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();
        }
    }
}
```

Replace concrete instantiation with call to the PizzaFactory to create a new pizza

Now we don't need to mess with this code if we add new pizzas

Class Diagram of New Solution



While this is nice, its not as flexible as it can be: to increase flexibility we need to look at two design patterns: Factory Method and Abstract Factory

GoF Factory Patterns

Class creational patterns focus on the use of inheritance to decide the object to be instantiated

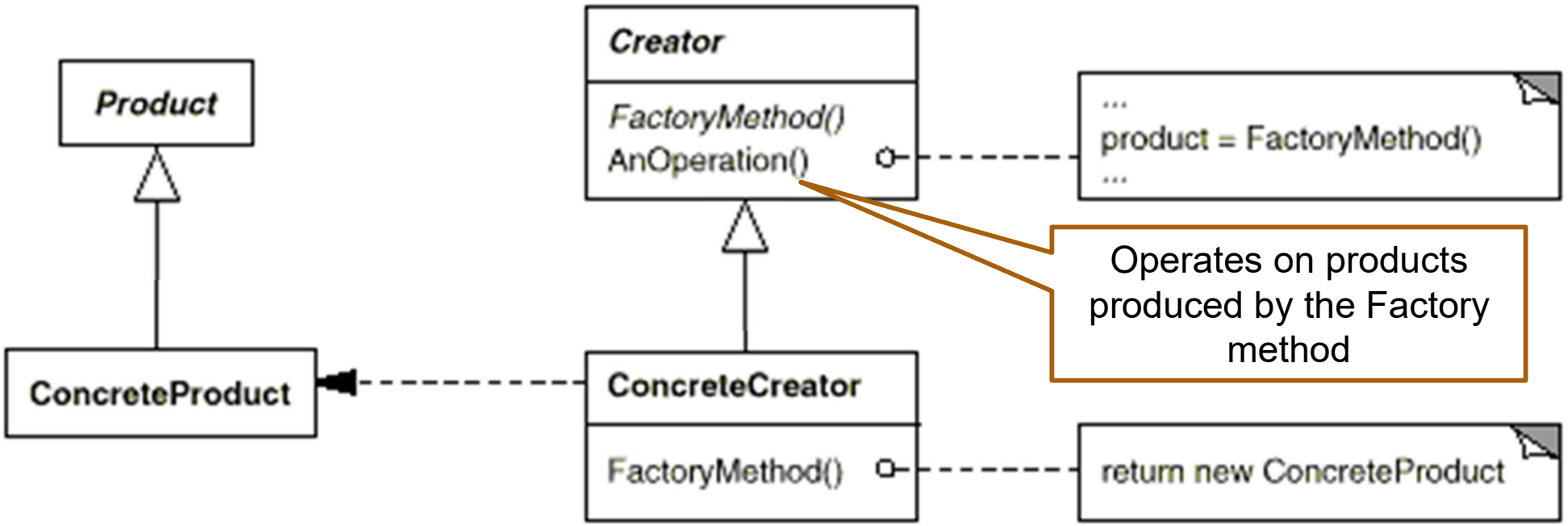
- Factory Method

Object creational patterns focus on the delegation of the instantiation to another object

- Abstract Factory

Factory Method

The Factory Method Pattern



In the official definition:

Factory method lets the subclasses *decide* which class to instantiate

Decide: --not because the classes themselves decide at runtime

-- but because the creator is written withont knowledge of the actual products that will be created, which is decided by the choice of the subclass that is used

The Factory Method Pattern: Participants

Product

- Defines the interface for the type of objects the factory method creates

ConcreteProduct

- Implements the Product interface

Creator

- Declares the factory method, which returns an object of type Product

ConcreteCreator

- Overrides the factory method to return an instance of a ConcreteProduct

Example: Pizza
Simple Factory to Factory Method

To demonstrate the factory method pattern, the pizza store example evolves

- to include the notion of different franchises
- that exist in different parts of the country (California, New York, Chicago)

Each franchise will need its own factory to create pizzas that match the proclivities of the locals

- However, we want to retain the preparation process that has made PizzaStore such a great success

The Factory Method Design Pattern allows you to do this by

- placing abstract, “code to an interface” code in a superclass
- placing object creation code in a subclass
- PizzaStore becomes an abstract class with an abstract createPizza() method

We then create subclasses that override createPizza() for each region

Example3: Pizza: Factory Method

CREATOR

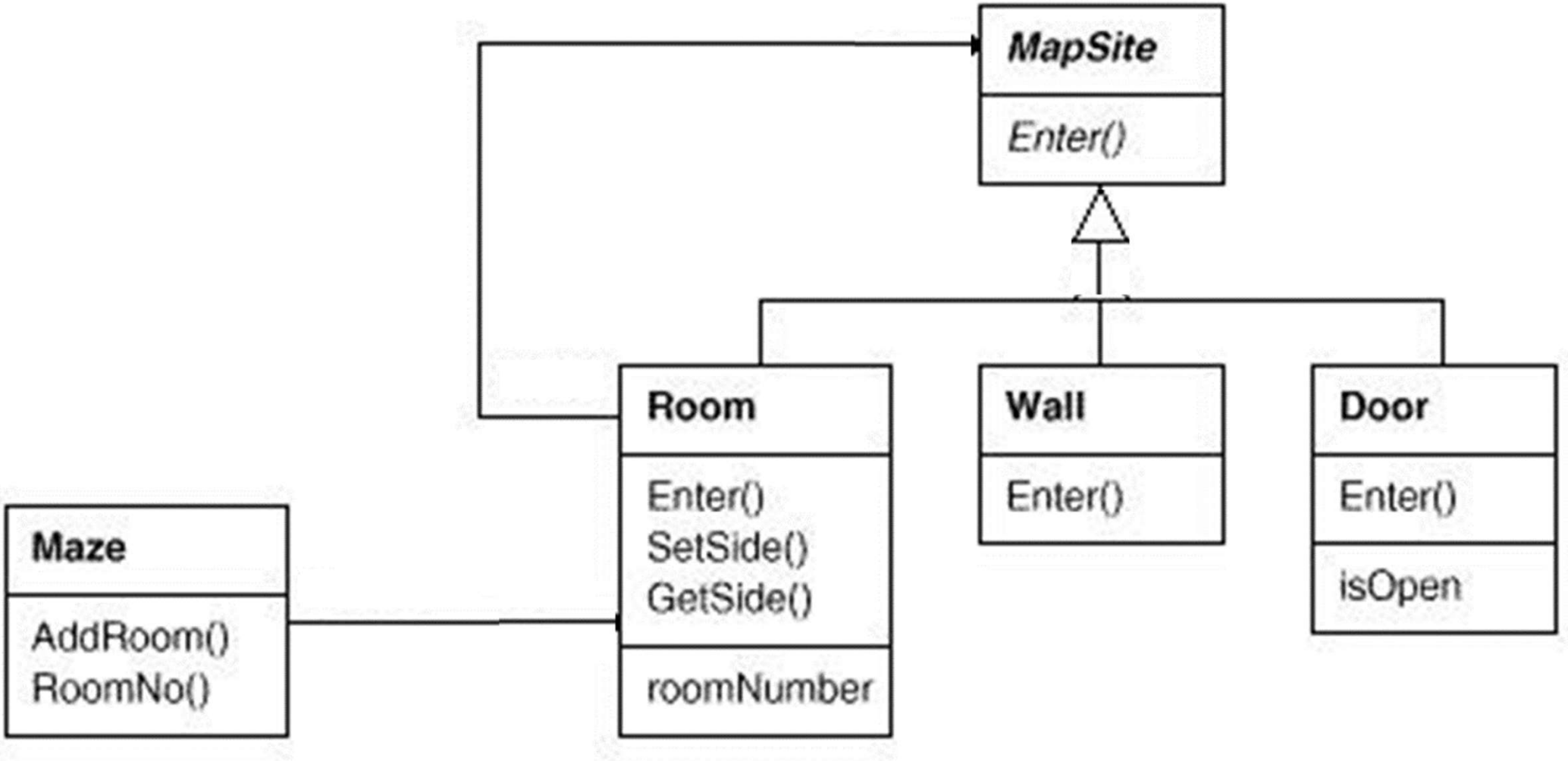
```
public abstract class PizzaStore {

        protected abstract createPizza(String type);

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

CONCRETE CREATOR

```
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();
            }
            return null;
    }
}
```

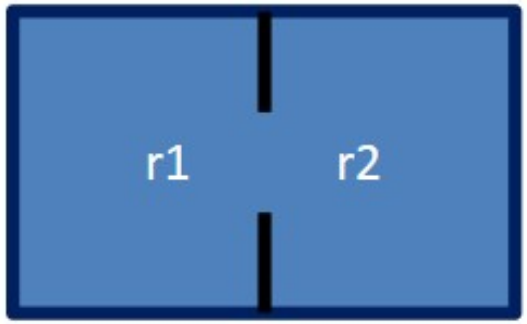
We want to build a Maze

Example: Maze



MazeGame class ha un metodo createMaze() che crea le component e le assembla (2 responsabilità)

```
/**
* 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;
      }
}
```



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?

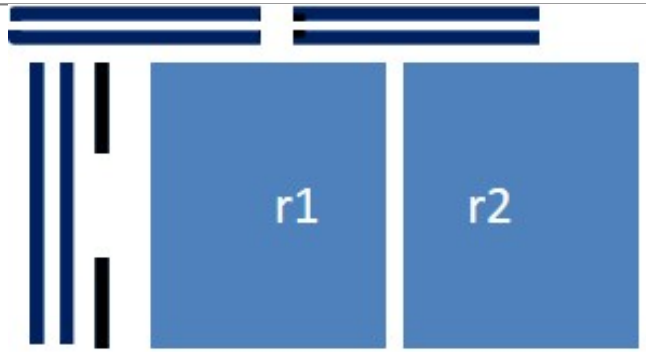
Let's add factory methods to the MazeGame class

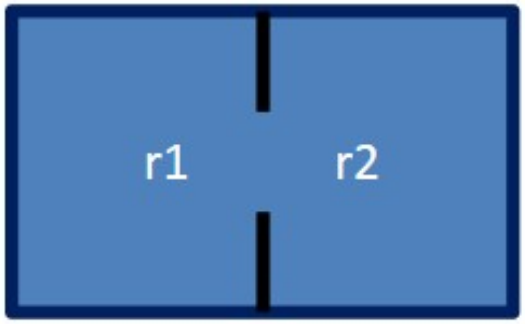
```
/**
* MazeGame with a factory methods.
*/
public class MazeGame {
    public Maze makeMaze() {return new Maze();}
    public Room makeRoom(int n) {return new Room(n);}
    public Wall makeWall() {return new Wall();}
    public Door makeDoor(Room r1, Room r2)  {return new Door(r1, r2);}
```

- Astratti (la creazione delle parti del labirinto è realizzata nelle sottoclassi)
- o concreti (viene data una implementazione di default della creazione delle parti

createMaze implementa l'assemblaggio

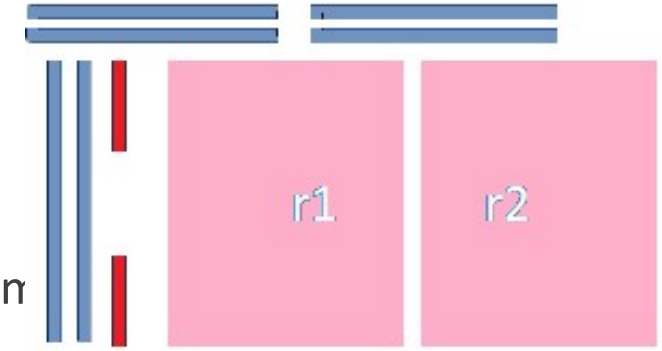
```
public Maze createMaze() {
    Maze maze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door door = makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, makeWall());
    r1.setSide(MazeGame.East, door);
    ……..
    r2.setSide(MazeGame.West, door);
    return maze;
  }
}
```





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

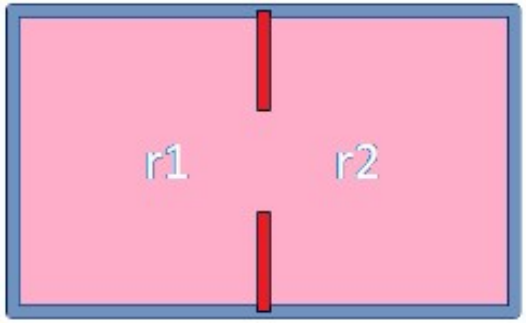
Consider this EnchantedMazeGame class:




```
public class EnchantedMazeGame extends MazeGame {

    public Room makeRoom(int n) {return new EnchantedRoom

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

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

}
```

The createMaze() method of MazeGame is inherited by EnchantedMazeGame

- It can be used to create regular mazes
- or enchanted mazes *without modification!*



Factory Method pattern at work: Maze

The reason this works is that the createMaze() method of MazeGame defers the creation of maze objects to its subclasses.

In this example, the correlations are:

- Creator => MazeGame
- ConcreteCreator => EnchantedMazeGame
  (MazeGame is also a ConcreteCreator)
- Product => Wall, Room, Door
- ConcreteProduct => EnchantedWall, EnchantedRoom, EnchantedDoor
- Maze is a concrete Product (but also Product)

The Factory Method Pattern

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

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
- Should the factory method be able to create multiple kinds of products? If so, then the factory method has a parameter (possibly used in an if-else!) to decide what object to create.

Abstract Factory

Intent

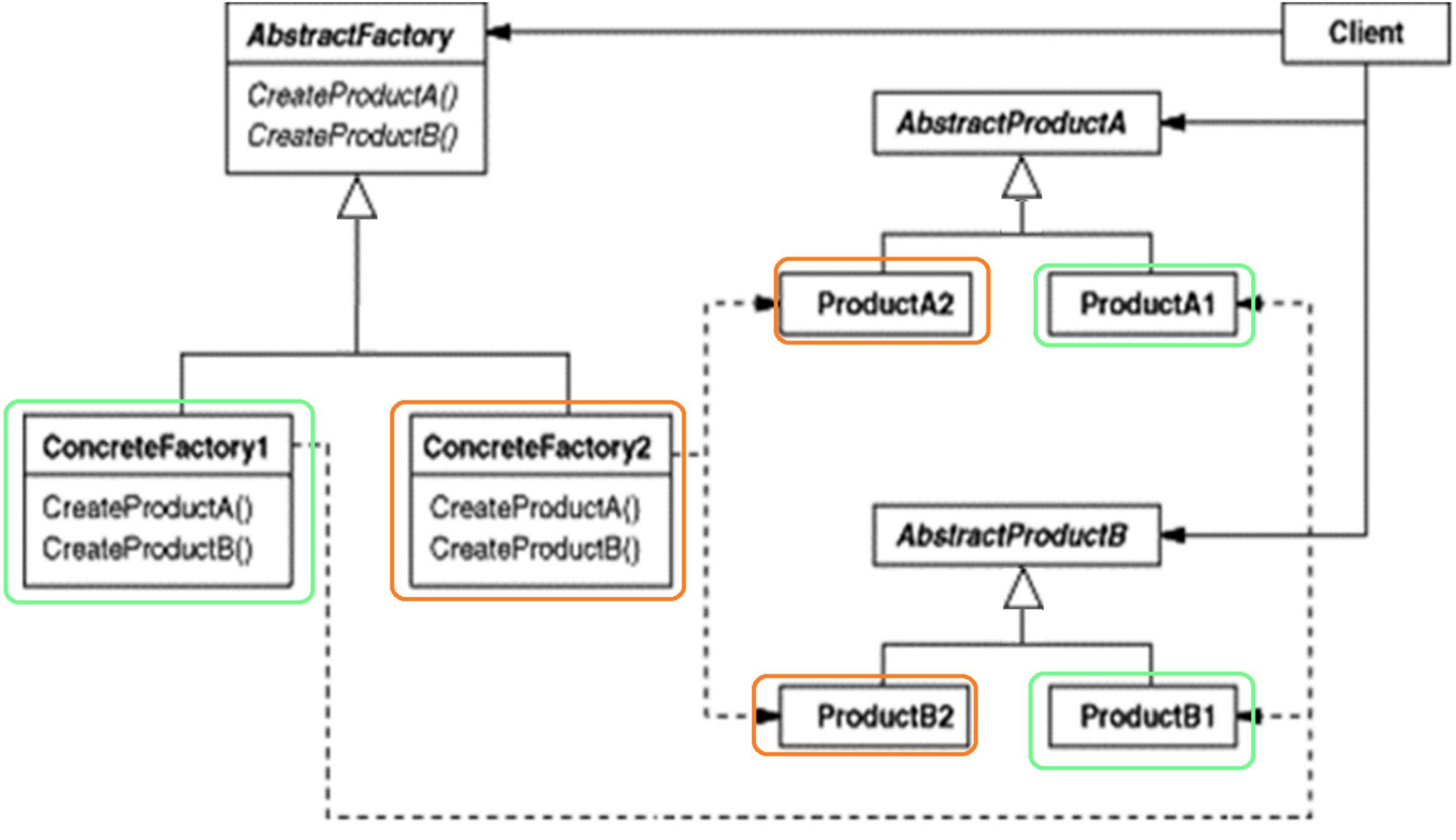
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

The Abstract Factory pattern vs Factory Method pattern.

- One difference between the two is that with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

Actually, the delegated object frequently uses factory methods to perform the instantiation, thus applying both patterns

Abstract Factory: structure



Abstract Factory applied to the MazeGame: the abstract factory

```
// MazeFactory – the abstract factory

public interface MazeFactory {
        public Maze makeMaze();
        public Room makeRoom(int n);
        public Wall makeWall();
        public Door makeDoor(Room r1, Room r2);
        }
```

Abstract Factory applied to the MazeGame: a concrete factory

```
// BasicMazeFactory – a concrete factory producing basic parts

public class BasicMazeFactory implements MazeFactory {
    public Maze makeMaze() {return new BasicMaze();}
    public Room makeRoom(int n) {return new BasicRoom(n);}
    public Wall makeWall() {return new BasicWall();}
    public Door makeDoor(Room r1, Room r2) {
        return new BasicDoor(r1, r2);}
    }
```

Abstract Factory applied to MazeGame: client implements assembly and delegates construction of parts.

The createMaze() method of the MazeGame class takes a MazeFactory reference as a parameter:

```
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());

        …

        return maze;
```

createMaze() delegates the responsibility for creating maze parts to the MazeFactory object

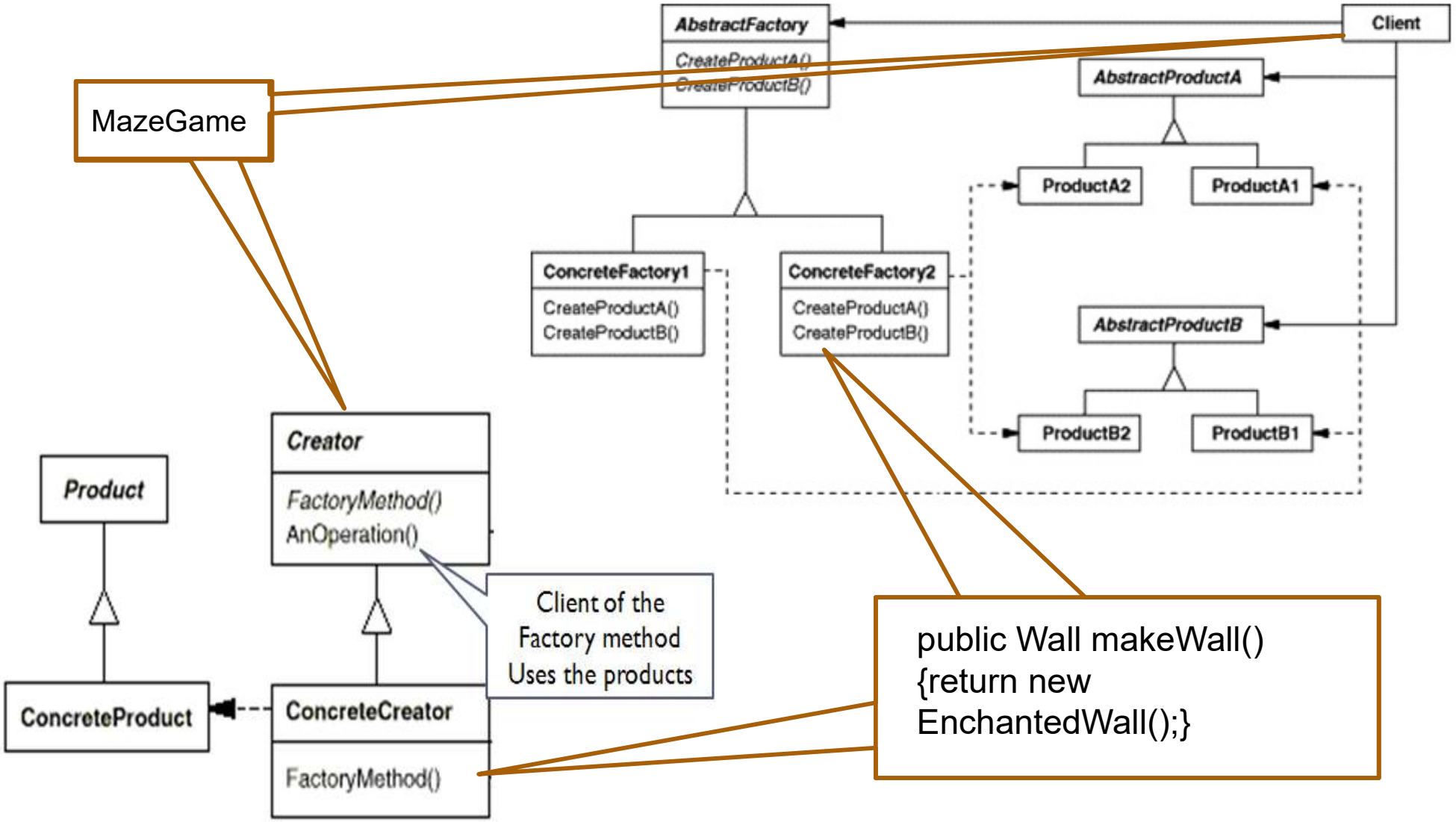
Another concrete factory

```
public class EnchantedMazeFactory implements MazeFactory {
    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);}
    }
```

In this example, the correlations are:

- AbstractFactory => MazeFactory
- ConcreteFactories => BasicMazeFactory and EnchantedMazeFactory
- AbstractProduct => Wall, Room, Door
- ConcreteProduct => BasicWall, BasicRoom, BasicDoor, EnchantedWall, EnchantedRoom, EnchantedDoor

Factory Method Abstract Factory



Moving On the Pizza Store

The factory method approach to the pizza store is a big success allowing our company to create multiple franchises across the country quickly and easily

But, bad news, we have learned that some of the franchises

- while following our procedures (the abstract code in PizzaStore forces them to)
- are skimping on ingredients in order to lower costs and increase margins

Our company's success has always been dependent on the use of fresh, quality ingredients

- so "Something Must Be Done!"

Abstract Factory to the Rescue!

We will alter our design such that a factory is used to supply the ingredients that are needed during the pizza creation process

- Since different regions use different types of ingredients, we'll create region-specific subclasses of the ingredient factory to ensure that the right ingredients are used
- But, even with region-specific requirements, since we are supplying the factories, we'll make sure that ingredients that meet our quality standards are used by all franchises
  - They'll have to come up with some other way to lower costs. ☺

First, We need a Factory Interface

```
1  public interface PizzaIngredientFactory {
2
3      public Dough createDough();
4      public Sauce createSauce();
5      public Cheese createCheese();
6      public Veggies[] createVeggies();
7      public Pepperoni createPepperoni();
8      public Clams createClam();
9
10 }
11
```

Note the introduction of more abstract classes:
Dough, Sauce, Cheese, etc.

Second, We implement a Region-Specific Factory

```
1  public class ChicagoPizzaIngredientFactory
2      implements PizzaIngredientFactory
3  {
4
5      public Dough createDough() {
6          return new ThickCrustDough();
7      }
8
9      public Sauce createSauce() {
10         return new PlumTomatoSauce();
11     }
12
13     public Cheese createCheese() {
14         return new MozzarellaCheese();
15     }
16
17     public Veggies[] createVeggies() {
18         Veggies veggies[] = { new BlackOlives(),
19                               new Spinach(),
20                               new Eggplant() };
21         return veggies;
22     }
23
24     public Pepperoni createPepperoni() {
25         return new SlicedPepperoni();
26     }
27
28     public Clams createClam() {
29         return new FrozenClams();
30     }
31 }
32
```

This factory ensures that quality ingredients are used during the pizza creation process…

… while also taking into account the tastes of people who live in Chicago

But how (or where) is this factory used?

Pizza abstract base class

```
1  public abstract class Pizza {
2      String name;
3
4      Dough dough;
5      Sauce sauce;
6      Veggies veggies[];
7      Cheese cheese;
8      Pepperoni pepperoni;
9      Clams clam;
10
11     abstract void prepare();
12
13     void bake() {
14         System.out.println("Bake for 25 minutes at 350");
15     }
16
17     void cut() {
```

First, alter the Pizza abstract base class to make the prepare method abstract…

A Pizza Subclass

```
1  public class CheesePizza extends Pizza {
2      PizzaIngredientFactory ingredientFactory;
3
4      public CheesePizza(PizzaIngredientFactory ingredientFactory) {
5          this.ingredientFactory = ingredientFactory;
6      }
7
8      void prepare() {
9          System.out.println("Preparing " + name);
10         dough = ingredientFactory.createDough();
11         sauce = ingredientFactory.createSauce();
12         cheese = ingredientFactory.createCheese();
13     }
14 }
15
```

Then, update Pizza subclasses to make use of the factory! Note: we no longer need subclasses like NYCheesePizza and ChicagoCheesePizza because the ingredient factory now handles regional differences

PizzaStore subclasses

```
1  public class ChicagoPizzaStore extends PizzaStore {
2
3      protected Pizza createPizza(String item) {
4          Pizza pizza = null;
5          PizzaIngredientFactory ingredientFactory =
6          new ChicagoPizzaIngredientFactory();
7
8          if (item.equals("cheese")) {
9
10             pizza = new CheesePizza(ingredientFactory);
11             pizza.setName("Chicago Style Cheese Pizza");
12
13         } else if (item.equals("veggie")) {
14
15             pizza = new VeggiePizza(ingredientFactory);
16             pizza.setName("Chicago Style Veggie Pizza");
17
                        ...
```

We need to update our PizzaStore subclasses to create the appropriate ingredient factory and pass it to each Pizza subclass in the createPizza factory method.

Summary: What did we just do?

1. We created an ingredient factory interface

1. This abstract factory gives us an interface for creating a family of products
    1. The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products
2. Our client code (PizzaStore) can then
    1. pick the factory appropriate to its region,
    2. plug it in, and
    3. get the correct style of pizza (Factory Method) with the correct set of ingredients (Abstract Factory)

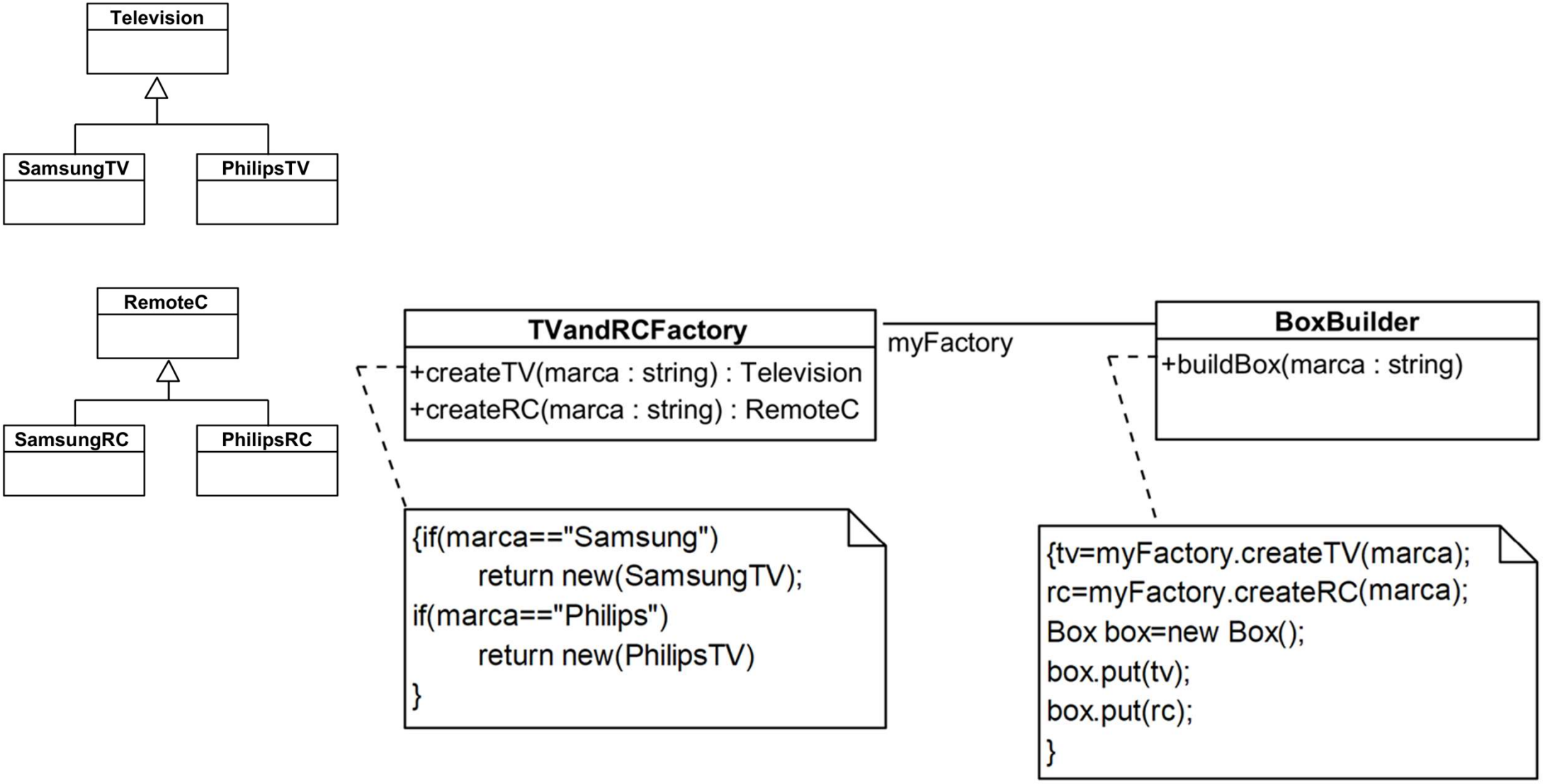
Exercise

We want to apply the factory patterns to produce and package:

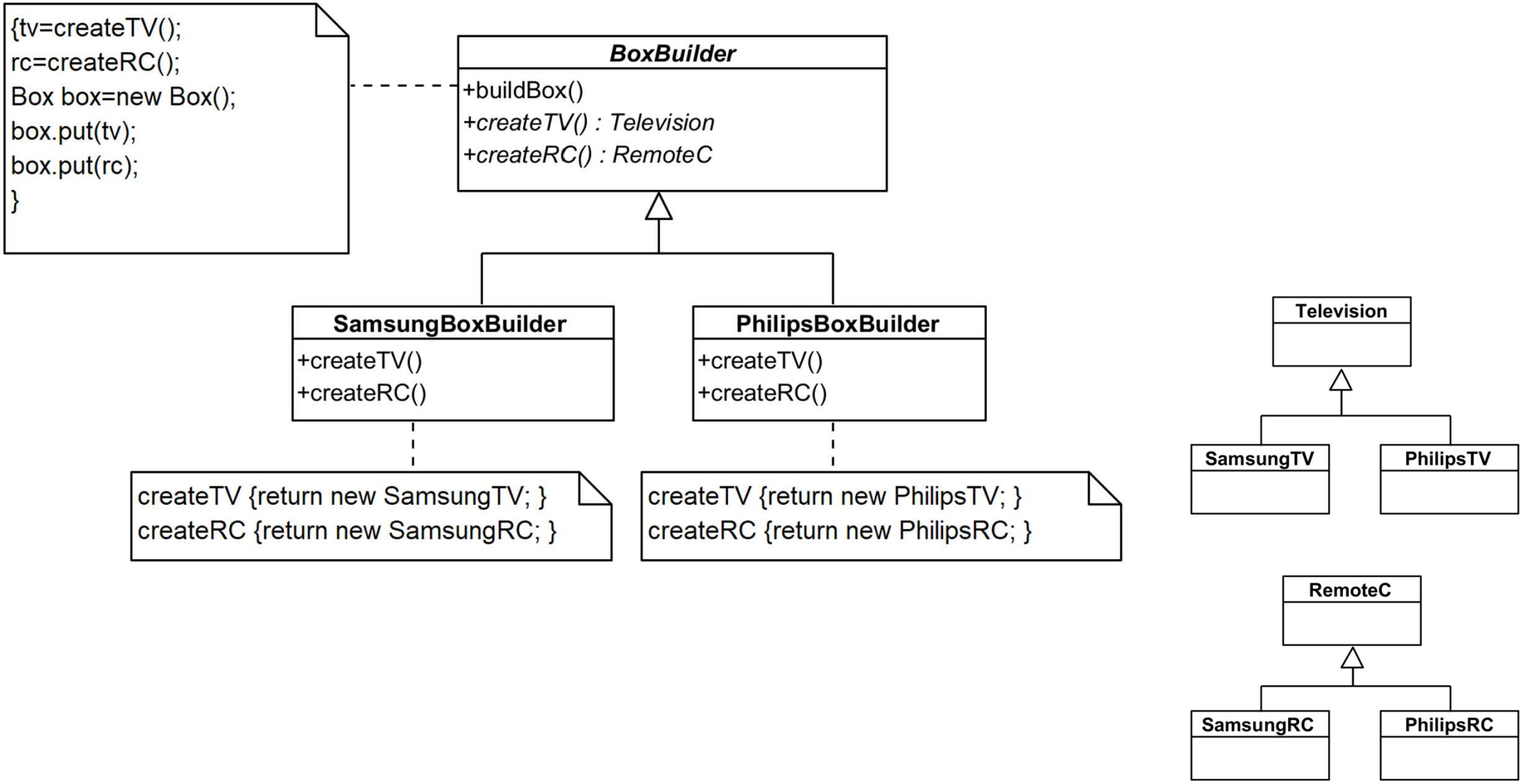
- Products: TVs and Remote controls (RC)
- Of two brands: Samsung and Philips
- (note that a Samsung TV uses a Samsung RC and a Philips TV uses a Philips RC)

1. Implement using Simple Factory: using parameters. (one unique factory)
2. Implement using Factory method: creator invoke (Abstract) build to have a TV and its RC, then packs them in a box
3. Implement using Abstract Factory: a client chooses the factory and asks for the products he needs, then it packs them in a box

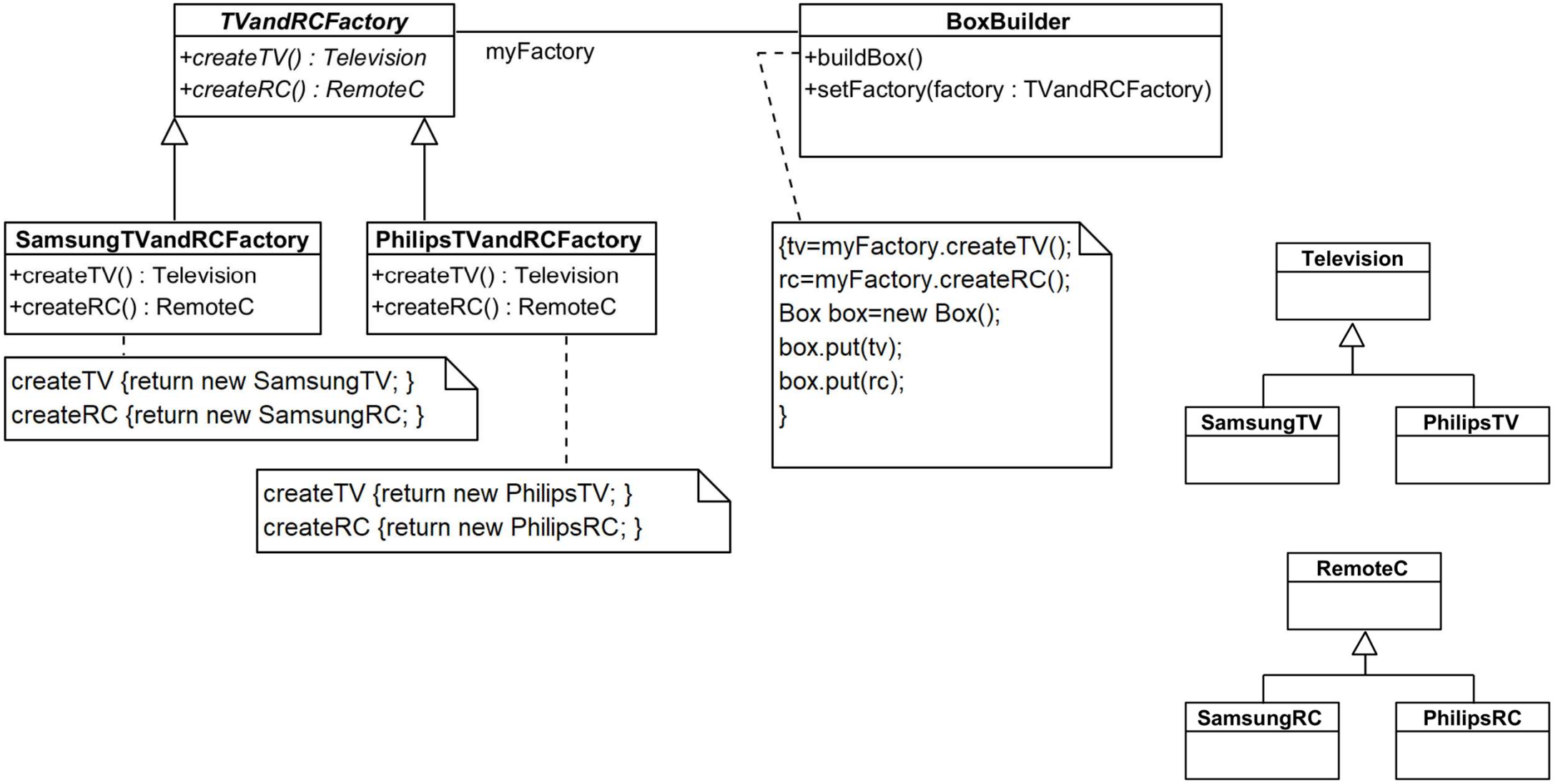
SimpleFactory



Factory Method



Abstract Factory



Cosa sono i *Pure Fabrication*

Pattern GRASP

Problem:

- Not to violate High Cohesion and Low Coupling

Solution:

- Assign a highly cohesive set of responsibilities to an artificial class
- that does not represent anything in the problem domain,
- in order to support high cohesion, low coupling, and reuse.

Pure Fabrication: discussion

The design of objects can be roughly partitioned to two groups

- Those chosen by representational decomposition
- Those chosen by behavioral decomposition

The latter group does not represent anything in the problem domain, they are simply made up for the convenience of the designer, thus the name pure fabrication: The classes are designed to group together related behavior

A pure fabrication object is a kind of functioncentric (or behavioral) object

una Factory è un Pure Fabrication

- In generale una Factory è un Pure Fabrication con l'obiettivo di:
  - Confinare la responsabilità di creazioni complesse in oggetti coesi
  - Incapsulare la complessità della logica di creazione

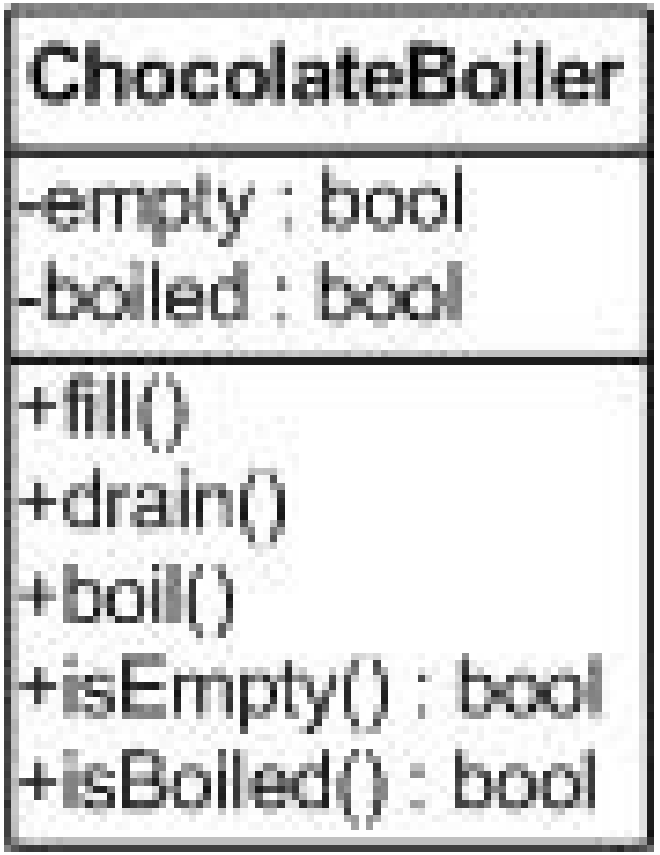
Singleton

The Chocolate Factory Example

The chocolate factory has computer controlled chocolate boilers. The job of the boiler is to take chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars. One of the main functions of the system is to prevent accidents such as draining 500 litres of unboiled mixture, or filing the boiler when is already full, or boiling an empty boiler.

Chocolate Boiler Controller

- The Boiler is controlled by the ChocolateBoiler class
- The ChocolateBoiler class has
  - two boolean attributes empty and boiled
  - five methods fill(), drain(), boil(), isEmpty() and isBoiled()



Chocolate Boiler methods

{empty} fill() {!empty}

{!empty and !boiled} boil() {!empty and boiled}

{!empty and boiled} drain() {empty}

Problems...

The Chocolate Boiler has overflowed! It added more milk to the mix even though it was full!!

What happened?

Hint: What happens if more than two instances of ChocolateBoiler are created?

The problem is with two instances controlling the same phisycal boiler

Singleton pattern

Intent

- Ensure a class only has one instance
- Provide a global point of access to it

Motivation

- Sometimes we want just a single instance of a class to exist in the system;
  - For example, we want just one window manager. Or just one factory for a family of products.
- We need to have that one instance easily accessible
- And we want to ensure that additional instances of the class can not be created

Recognizing Singleton

Unique objects are not uncommon

Most objects in an application bear a unique responsibility

Yet singleton classes are relatively rare

Fact that an object/class is unique doesn't mean that the Singleton pattern is at work

Prevent multiple instances

How can you prevent other developers from constructing new instances of your class?

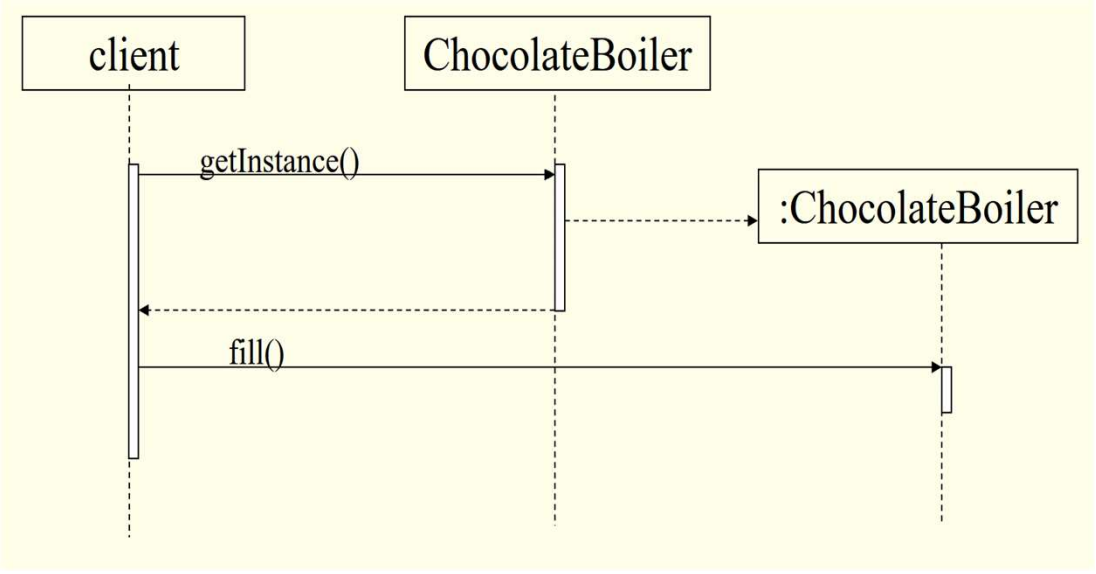
- Create a single constructor with private access
  - `private static ChocolateBoiler _chocolateboiler = new ChocolateBoiler()`
- Make the unique instance available through a method:
  - `public static ChocolateBoiler() GetChocolateBoiler()`

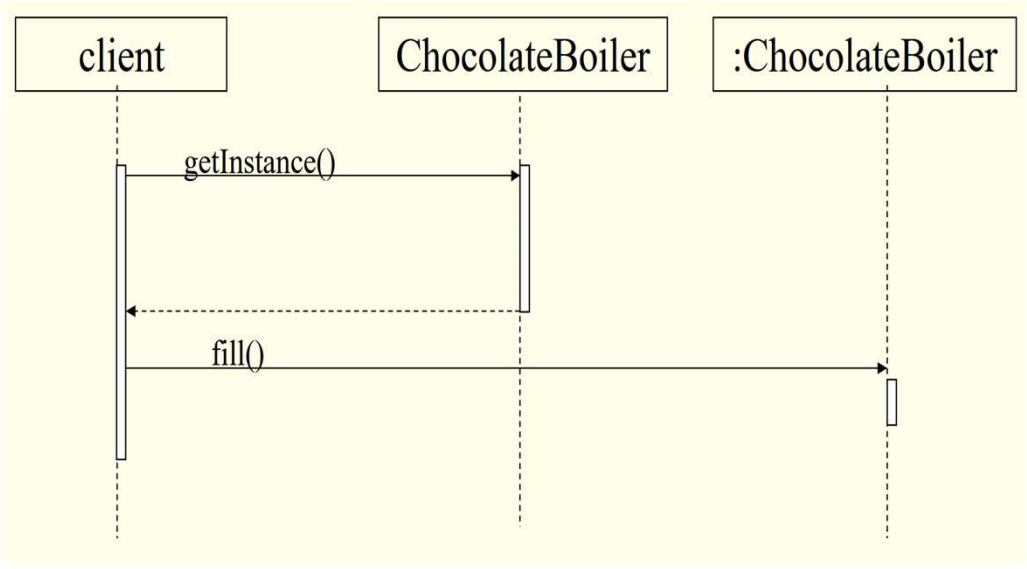
GetChocolateBoiler() (with lazy Initialization)

```
public static ChocolateBoiler GetChocolateBoiler()
{
    if (_chocolateboiler == null)
    {
                    _chocolateboiler = new ChocolateBoiler();
                    // ...
    }
    return _chocolateboiler
}
```

Rather than creating a singleton instance ahead of time – wait until instance is first needed

_chocolateboiler == null vs _chocolateboiler != null





Why use Lazy Initialization?

Might not have enough information to instantiate a singleton at static initialization time

- Example: a ChocolateBoiler singleton may have to wait for the real factory's machines to establish communication channels

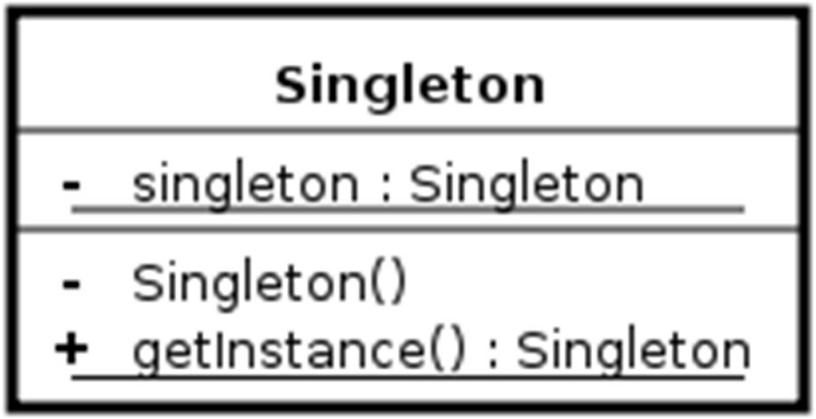
If the singleton is resource intensive and may not be required

- Example: a program that has an optional query function that requires a database connection

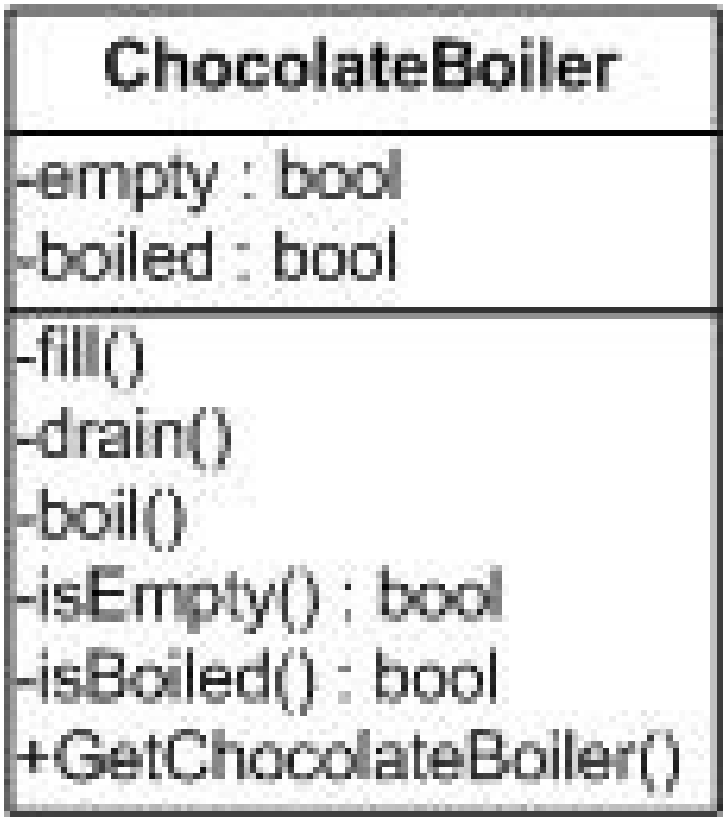
Full Picture

```
public class ChocolateBoiler {
   private static ChocolateBoiler _chocolateboiler;
   private ChocolateBoiler () {};
   public static ChocolateBoiler GetChocolateBoiler()
   {
     if (_chocolateboiler == null)
     {
      _chocolateboiler = new ChocolateBoiler();
            // ...
     }
     return _chocolateboiler
   }
}
```

UML Class Diagram



Our class so far...



as it is, problems with threads …

We don't know what happened! The new Singleton code was running fine. The only thing we can think of is that we just added some optimizations to the Chocolate Boiler Controller that makes use of multiple threads.

Thread Example

If the program is run in a multi-threaded environment it is possible for two threads to initialize two singletons at roughly the same time

Thread 1

```
public stat ChocolateBoiler
        getInstance()

if (uniqueInstance == null)

uniqueInstance =
        new ChocolateBoiler()

return uniqueInstance;
```

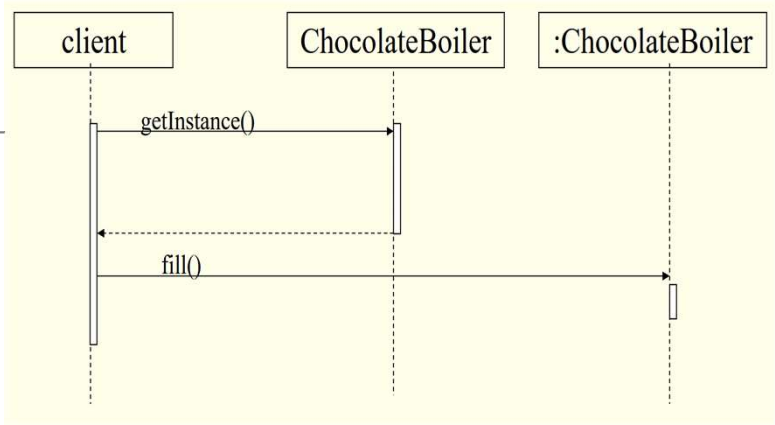
Thread 2

```
public stat ChocolateBoiler
        getInstance()

if (uniqueInstance == null)

uniqueInstance =
        new ChocolateBoiler()

return uniqueInstance;
```

Problems with Multithreading

- In the case of multithreading with more than one processor the getInstance() method could be called at more or less the same time resulting in to more than one instance being created.
- Possible solutions:
  1. Move to an eagerly created instance rather than a lazily created one.
     - Easy! But memory may be allocated and not used.
  2. Synchronize the getInstance() method
     - Disadvantage – synchronizing can decrease system performance.
  3. Use double—checked—locking
     - The idea is to avoid the costly synchronization for all invocations of the method except the first.

Use an Eagerly Created Instance



- Code:

```
//Data elements
private static Singleton uniqueInstance = new Singleton()

private Singleton() {}

public static Singleton getInstance() {
   return uniqueInstance
}
```

- Easy! But memory may be allocated and not used.

Synchronize the getInstance() Method

Code

```
public static synchronized Singleton getInstance()
  {...
   }
```

Disadvantage – synchronizing can decrease system performance.

Synchronization is expensive, however, and is really only needed the first time the unique instance is created.

Use only if the performance of the getInstance() method is not critical to the application.

Double—checked—locking

- Code:

```
private volatile static Singleton uniqueInstance
private Singleton() {}
public static Singleton getInstance() {
   if (uniqueInstance == null)
      synchronized (Singleton.class) {
         if (uniqueInstance == null) {
            uniqueInstance  = new Singleton()
   return uniqueInstance
}
```

- If a variable is declared as volatile then is guaranteed that any thread which reads the field will see the most recently written value.

Double—checked—locking explainded

Check first to see if the instance exists or not. If not, then lock up a block of code.

```
// Danger!  This implementation of Singleton not
// guaranteed to work prior to Java 5
//
public class Singleton {
        private volatile static Singleton uniqueInstance;
        private Singleton() {}
        public static Singleton getInstance() {
                if (uniqueInstance == null) {
                        synchronized (Singleton.class) {
                                if (uniqueInstance == null) {
                                        uniqueInstance = new Singleton();
                                } } }
                return uniqueInstance;
                } }
```

A thread's copy of a volatile attribute is reconciled with the "master" copy each time it is referenced.

Note these two checks on uniqueInstance

Second check is necessary to verify uniqueInstance is still null

A synchronized block of code.

Only one thread at a time will execute this.

Result is very little overhead compared to synchronizing a whole method/class.

Singleton With Subclassing

What if we want to be able to subclass Singleton and have the single instance be a subclass instance?

For example, suppose MazeFactory has subclasses EnchantedMazeFactory and AgentMazeFactory. We want to instantiate just one of them.

How could we do this?

1. Have the static getInstance() method of MazeFactory determine the particular subclass instance to instantiate. This could be done via an argument or environment variable. The constructors of the subclasses can not be private in this case, and thus clients *could instantiate other instances of the* subclasses.

2. Have each subclass provide a static getInstance() method. Now the subclass constructors can be private.

Singleton With Subclassing (Method 1)

The MazeFactory instance() method determines the subclass to instantiate

```
public abstract class MazeFactory {
        // The private reference to the one and only instance.
        private static MazeFactory uniqueInstance = null;
        // Create the instance using the specified String name.
        public static MazeFactory getInstance(String name) {
                if(uniqueInstance == null)
                    if (name.equals("enchanted"))
                            uniqueInstance = new EnchantedMazeFactory();
                    else if (name.equals("agent"))
                            uniqueInstance = new AgentMazeFactory();
            return uniqueInstance;}} //this may not be the one specified by the parameter
```

Singleton With Subclassing (Method 1)

Problem!!!

```
if(uniqueInstance == null)
    if (name.equals("enchanted"))
            uniqueInstance = new EnchantedMazeFactory();
    else if (name.equals("agent"))
            uniqueInstance = new AgentMazeFactory();
```

The constructors of EnchantedMazeFactory and AgentMazeFactory can not be private, since MazeFactory must be able to instantiate them. Thus, clients could potentially instantiate other instances of these subclasses.

Singleton With Subclassing (Method 1)

Moreover, the getInstance(String name) methods violates the Open-Closed Principle, since it must be modified for each new MazeFactory subclass

We could use Java class names as the argument to the instance(String) method, yielding simpler code:

```
public static MazeFactory getInstance(String name) {
    if (uniqueInstance == null)
        //Usa la reflection per creare un'istanza della classe specificata da nome
        uniqueInstance = Class.forName(name).newInstance();
    return uniqueInstance;
}
```

Singleton With Subclassing Method 2

Have each subclass provide a static instance method()

```
public abstract class MazeFactory {
    // The protected reference to the one and only instance.
    protected static MazeFactory uniqueInstance = null;
    // The MazeFactory constructor. If you have a default
    // constructor, it can not be private here!
    protected MazeFactory() {}
    // Returns a reference to the single instance.
    public static MazeFactory getInstance() {return uniqueInstance;}
}
```

Singleton With Subclassing Method 2

```
public class EnchantedMazeFactory extends MazeFactory {
        // Return a reference to the single instance.
    public static MazeFactory getInstance() {
        if(uniqueInstance == null)
            uniqueInstance = new EnchantedMazeFactory();
        return uniqueInstance;
    }
    // Private subclass constructor
    private EnchantedMazeFactory() {}
}
```

Singleton With Subclassing Method 2

Client code to create factory the first time:

```
MazeFactory factory = EnchantedMazeFactory. getInstance();
```

Client code to access the factory:

```
MazeFactory factory = MazeFactory. getInstance();
```

Note that now the constructors of the subclasses are private. Only one subclass instance can be created!

Also note that the client can get a null reference if it invokes MazeFactory.getInstance() before the unique subclass instance is first created

Finally, note that uniqueInstance is now protected!

Static Attributes in a Class (memento)

Each object of a class has its own copy of all the instance variables of that class.

However, in certain cases all class objects should share only one copy of a particular variable.

- Such variables are called static variables. A program contains only one copy of each of a class's static variables in memory, no matter how many objects of the class have been instantiated.

A static variable represents class-wide information. All class objects share the same static data item.

The public static attributes of a class can be accessed through the class name and dot operator (e.g. Math.PI). Private static attributes can only be accessed through methods and properties of that class.

Better using singletons or static classes?

- PRO for Singleton:
  - With a singleton you can pass the object as a parameter to another method;
  - With a singleton you can derive a base class;
  - With a singleton you can use a factory pattern to build up your instance (and/or choose which class to instantiate).
- CONS for Singleton
  - Those seen
  - https://www.oracle.com/technical-resources/articles/java/singleton.html
- In both cases care with multithreading.