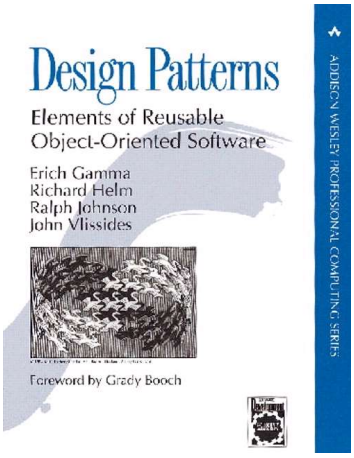
Design Patterns
Factories
Singleton

Roberta Gori, Laura Semini
Ingegneria del Software
Dipartimento di Informatica
Università di Pisa

Se interessati: libri suggeriti



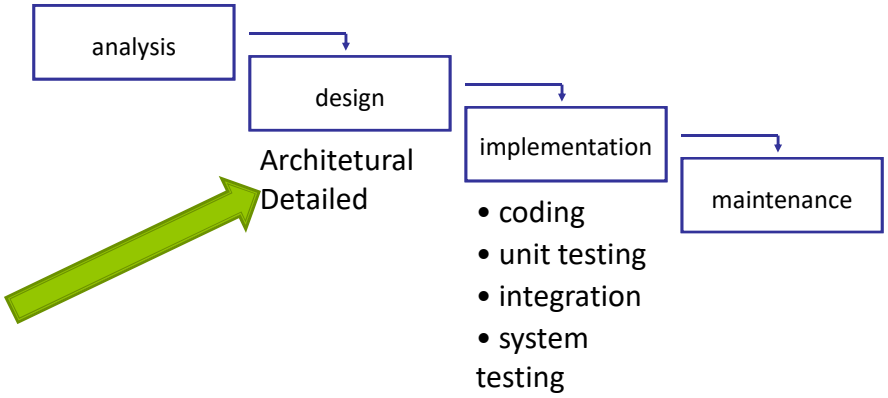
Definizione di design pattern?

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
  - -- Christopher Alexander A Pattern Language, 1977

Definizione data da Christopher Alexander

- C. Alexander ha definito i design patterns studiando tecniche per migliorare il processo di progettazione di edifici e aree urbane
- Ogni pattern è una regola in tre parti, che esprime una relazione tra
  - Un contesto
  - Un problema
  - Una soluzione
- DEF: "una soluzione a un problema in un contesto"
- I pattern possono essere applicati a diverse aree, compreso lo sviluppo software

In che fase si applicano



GoF Design Patterns

- Sono 23 design pattern suddivisi in base al loro scopo
- Creazionali:
  - propongono soluzioni per creare oggetti
- Comportamentali:
  - propongono soluzioni per gestire il modo in cui vengono suddivise le responsabilità delle classi e degli oggetti
- Strutturali:
  - propongono soluzioni per la composizione strutturale di classi e oggetti

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

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

Vedremo 3 tipi di pattern Factories

- Simple Factory (detto anche Concrete Factory)
  - non è un pattern GoF
  - è una semplificazione molto diffusa di AF
- Abstract Factory (AF)
- Factory Method (FM)

Cosa sono i *Pure Fabrication*

- 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.

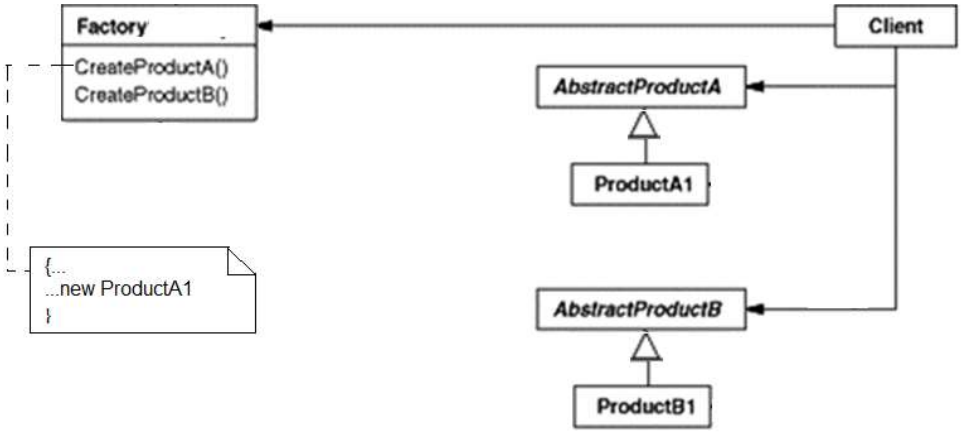
Pure Fabrication: discussion

- 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

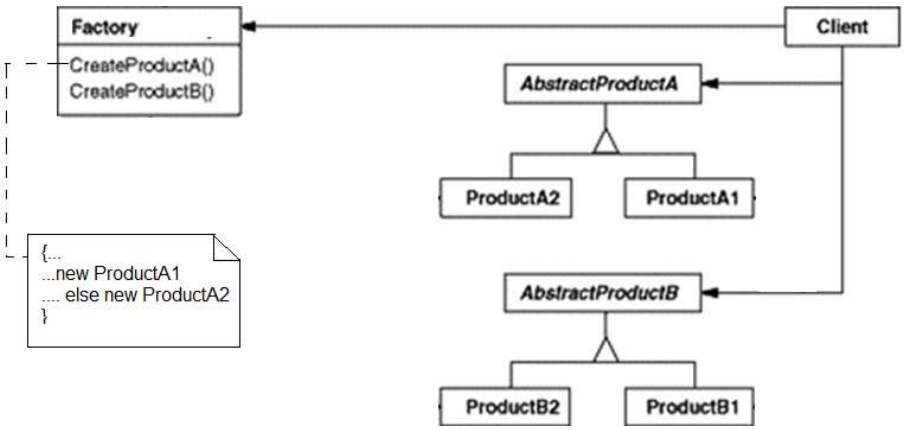
Simple Factory (aka concrete factory)

- 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 (*Pure Fabrication)* 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 becomes unwieldy as we add to our menu

This part stays the same

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

Example3: Pizza Simple solution: just a factory

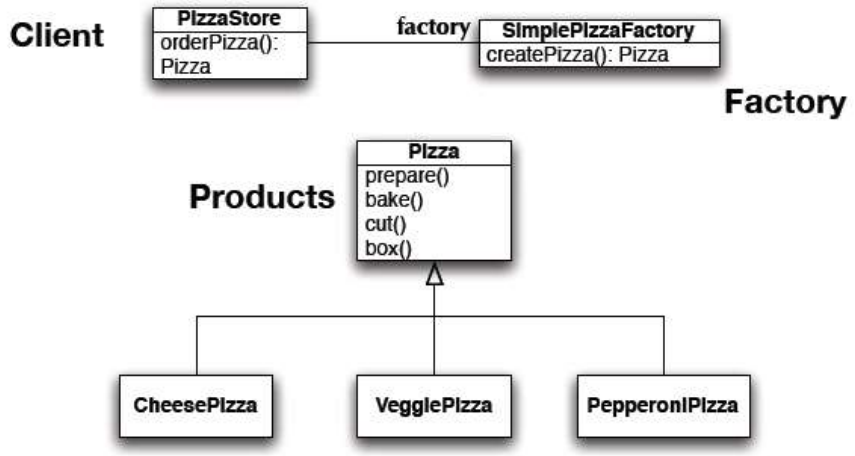
```
public class PizzaStore {
  private SimplePizzaFactory factory;
     public PizzaStore(SimplePizzaFactory factory)
     {
     this.factory = factory;
     }
public Pizza orderPizza(String type) {
     Pizza pizza = factory.createPizza(type);
     pizza.prepare();
     pizza.bake();
      pizza.cut();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

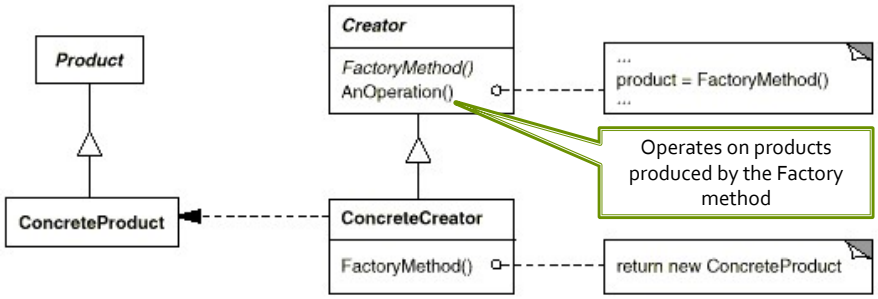


While this is nice, its not as flexible as it can be: to increase flexibility we need to look at two design patterns: Factory Method and Abstract Factory

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

The Factory Method Pattern



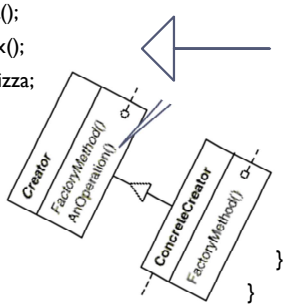
In the official definition:
Factory method lets the subclasses *decide* which class to instantiate
Decide: --not because the classes themselves decide at runtime
-- but because the creator is written without knowlwdge of the actual products that will be created, which is decided by the choice of the subclass that is usd

Example3: 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

```
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;
              }
}
```



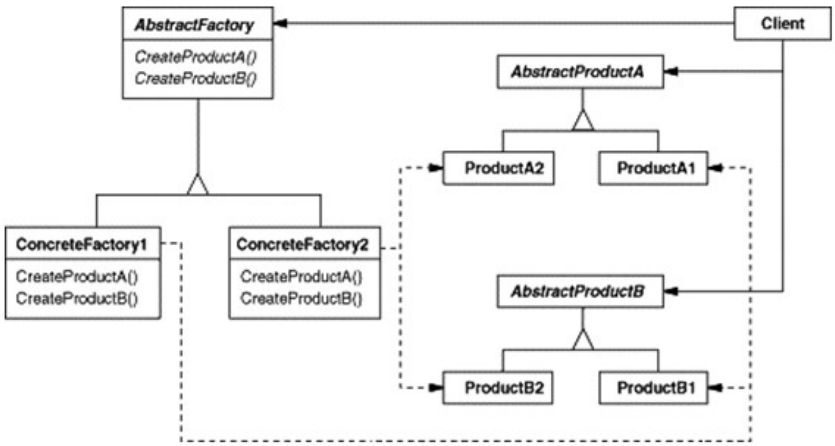

```
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
            if (type.equals("cheese")) {
                return new NYCheesePizza();
            } else if (type.equals("greek")) {
                return new NYGreekPizza();
            } else if (type.equals("pepperoni"))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


Abstract Factory

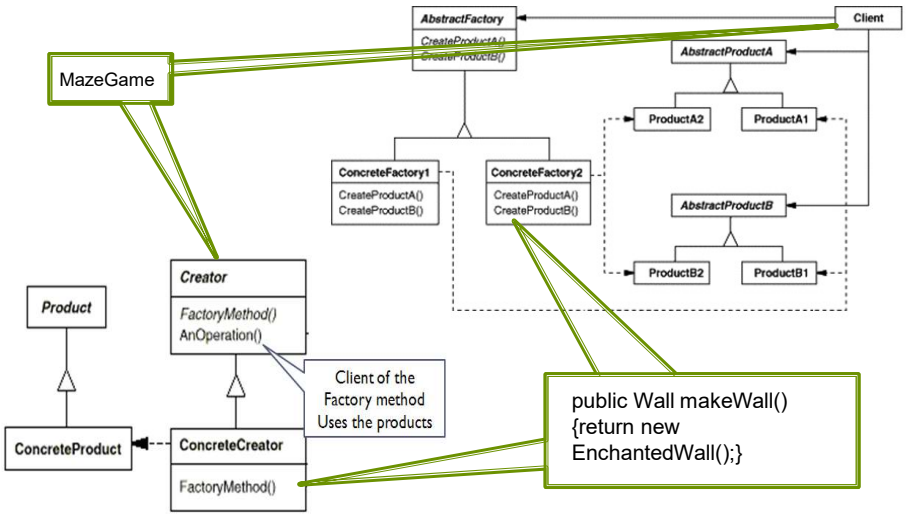
Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- The Abstract Factory pattern is very similar to the 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!

Abstract Factory: structure



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?

Within Pizza Subclasses… (I)

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

```
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() {
```

Within Pizza Subclasses… (II)

- Then, update Pizza subclasses to make use of the factory! Note: we no longer need subclasses like NYCheesePizza and ChicagoCheesePizza because

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

One last step…

```
public class ChicagoPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
        new ChicagoPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Chicago Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("Chicago Style Veggie Pizza");
                        ...
```

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?

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

Singleton

Singleton pattern

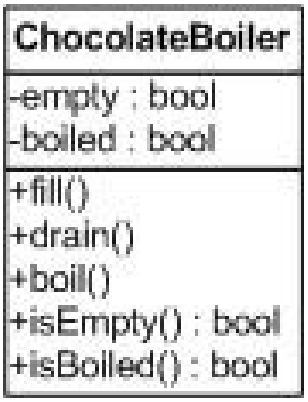
- Intent
  - Ensure a class only has one instance
  - Provide a global point of access to it
- Motivation
  - Sometimes we want just a single instance of a class to exist in the system;
    - For example, we want just one window manager. Or just one factory for a family of products.
  - We need to have 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

Chocolate Factory Case Study

- Choc-O-Holic Inc's industrial strength Chocolate Boiler mixes ingredients and milk at a high temperature to make liquid chocolate
- The ChocolateBoiler class also has two boolean attributes empty and boiled
- The ChocolateBoiler class contains five methods fill(), drain(), boil(), isEmpty() and isBoiled()



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

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 public static GetChocolateBoiler() method

Lazy Initialization

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

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

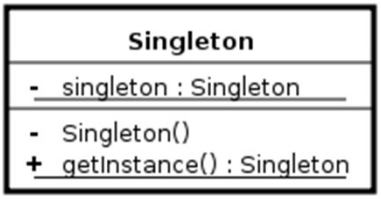
Why use Lazy Initialization?

- 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

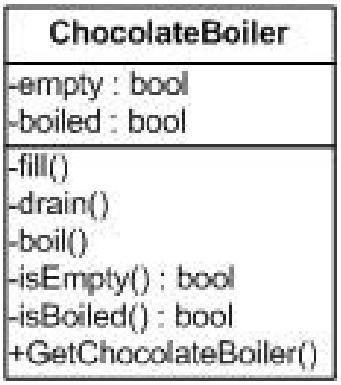
Full Picture

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

UML Class Diagram



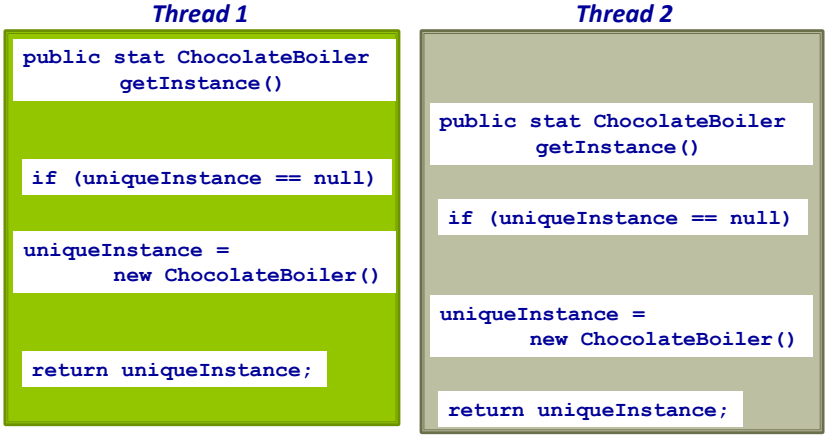
Our class so far…



- **as it is, problems with threads …**

Thread Example

- environment it is possible for two threads to initialize two singletons at roughly the same time



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.

Problems With Subclassing

- What if we want to be able to subclass Singleton and have the single instance be a subclass instance?
- How could we do this?
  - Have the static instance() method 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.
  - Have each subclass provide a static instance() method. Now the subclass constructors can be private.

Better using singletons or static classes?

- With a singleton you can pass the object as a parameter to another method;
- With a singleton you can implement interfaces or 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).

- In both cases care with multithreading.

Homework

- Apply the factory patterns to produce:
  - Products: TVs and Remote controls (RC)
  - Of two brands: Samsung and Philips

1. With Simple Factory: using parameters. Fare in modo di avere una sola factory
2. With Factory method: creator builds a TV and its RC, then packs them togheter.
3. With Abstract Factory: a client chooses the factory and asks for the product(s) he needs, for instance one TV and two (compatible) remote controllers.

Homework

- Apply Singleton to define a class of the television receiving commands from the remote controller(s) to set the volume of the TV:
- Offered methods are:
  - void increase() {volume++}
  - void decrease() {volume--}