I3305-GUI
Chapter 2
Design Patterns, Creational Patterns

Abed Safadi,Ph.D

What is a Design Pattern?

Christopher Alexander [1] says
“Each pattern describes a problem which happen 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"

- Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns.
- When organizing a program, it is useful to understand the ways that people have organized in the past!
- Design patterns are like a bag of tricks that every competent programmer should understand.
- This doesn't mean that you use them indiscriminately
  - You can recognize situations where they might apply and then
  - Decide whether their use merited in that particular case.

[1] : is an architect noted for his theories about design, and for more than 200 building projects in California, Japan, Mexico and around the world.

Definition

In software engineering, a **design pattern** is a general repeatable solution to a *commonly occurring problem* in the design of object-oriented applications.

Example: Visitor pattern lets you define a new operation without changing the classes of the elements on which it operates.

Note: At the design level, Design Patterns are independent of the programming languages used.

Representation and Types

Essential elements

- **Name** : which allows to identify it clearly
- **Problem** : explains the problem and its context.
- **Solution** : describes the elements that make up the design, their relationships, responsibilities, and collaborations (UML)
  - The solution doesn' t describe a particular concrete design or implementation, because a pattern is like a ***template*** that can be applied in many different situations.

Types of Design Patterns

As per the design pattern reference book Design Patterns - Elements of Reusable Object-Oriented Software , there are 23 design patterns which can be classified in three categories:

- Creational,
- Structural and
- Behavioral patterns.

Creational Patterns

- These design patterns provide a way to create objects while *hiding the creation logic*, rather than instantiating objects directly using new operator.

- This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural and Behavioral Patterns

- **Structural Patterns**
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral Patterns**
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Design Pattern Catalog

- Creational patterns: concern the process of object creation
  - Factory method
  - Builder
  - Singleton
- Structural patterns: deal with the composition of classes or objects
  - Flyweight
  - Bridge
  - Adapter
  - Composite
  - Proxy
  - Facade
- Behavioral patterns: characterize the ways in which classes or objects interact and distribute responsibility.
  - Visitor
  - Observer-MVC
  - Strategy
  - Iterator

Plan

- Factory method
- Builder
- Singleton

Intent

The Factory Pattern is used to create different objects from a **factory**

If we have a super class and *n* sub-classes, and based on input parameters, we have to return the object of one of the sub-classes, we use a factory pattern.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

So instead of having object creation code on client side we encapsulate inside a Factory method

The *new operator* considered harmful

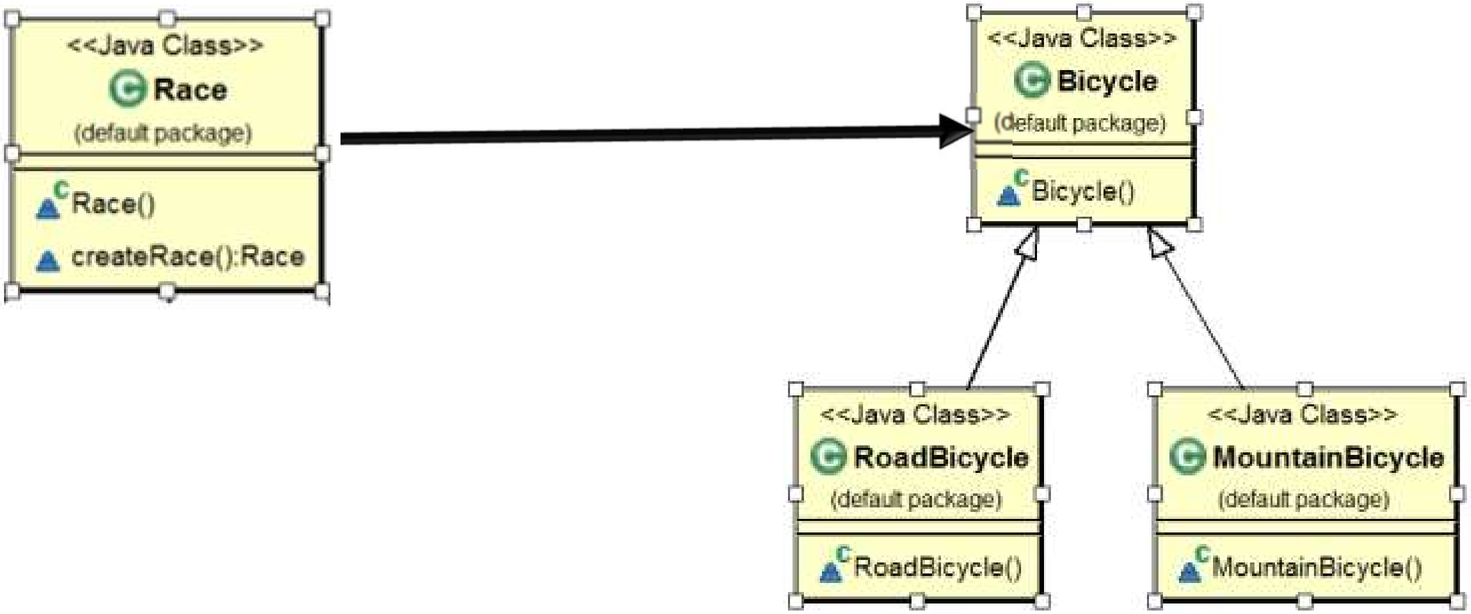
Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Example

Imagine that you're creating a simulator to represent a bicycle race, a race consists of many bicycle, i.e (**bicycle for normal race**, **Road bicycle for the tour de France race** and **Mountain Bicycle for Cyclocross race)**

Traditionnel version of solution



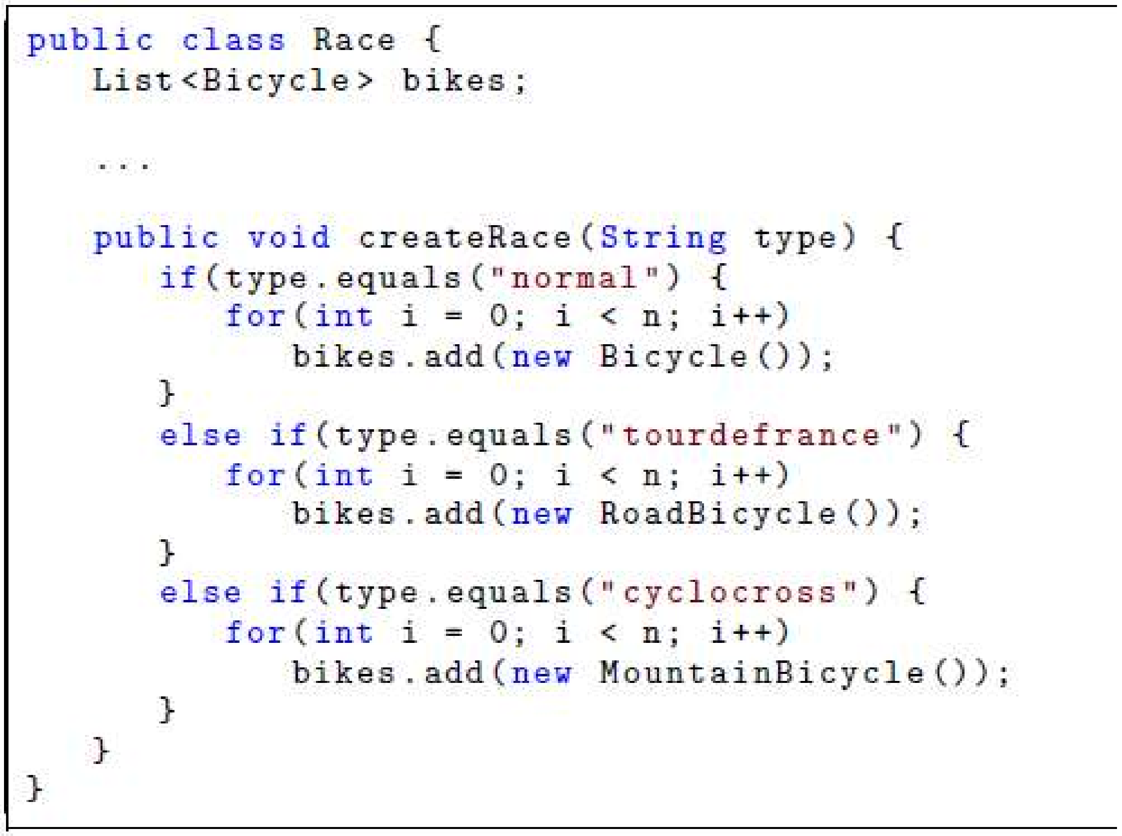
Creator

Product

Concrete Product

Concrete Product

Bad Solution



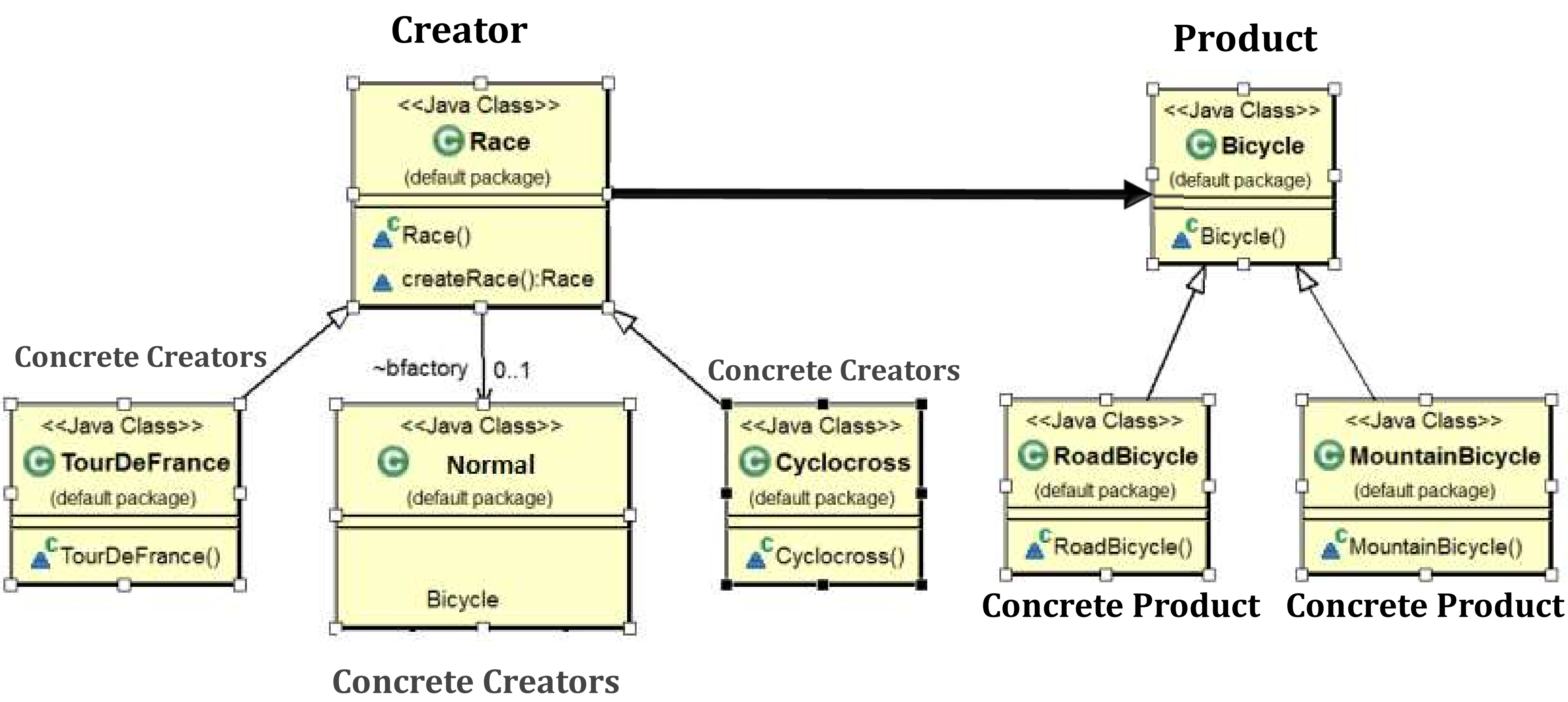
```java
public class Race {
   List<Bicycle> bikes;

   ...

   public void createRace(String type) {
      if(type.equals("normal") {
         for(int i = 0; i < n; i++)
            bikes.add(new Bicycle());
      }
      else if(type.equals("tourdefrance") {
         for(int i = 0; i < n; i++)
            bikes.add(new RoadBicycle());
      }
      else if(type.equals("cyclocross") {
         for(int i = 0; i < n; i++)
            bikes.add(new MountainBicycle());
      }
   }
}
```

- New Race type ➔ update createRace method
- Assume that factory of each Race Type contains different additional work to do (variables, methods)!
- How you can make the solution more clean (e.g., easy to add new race type without modifying existing methods)

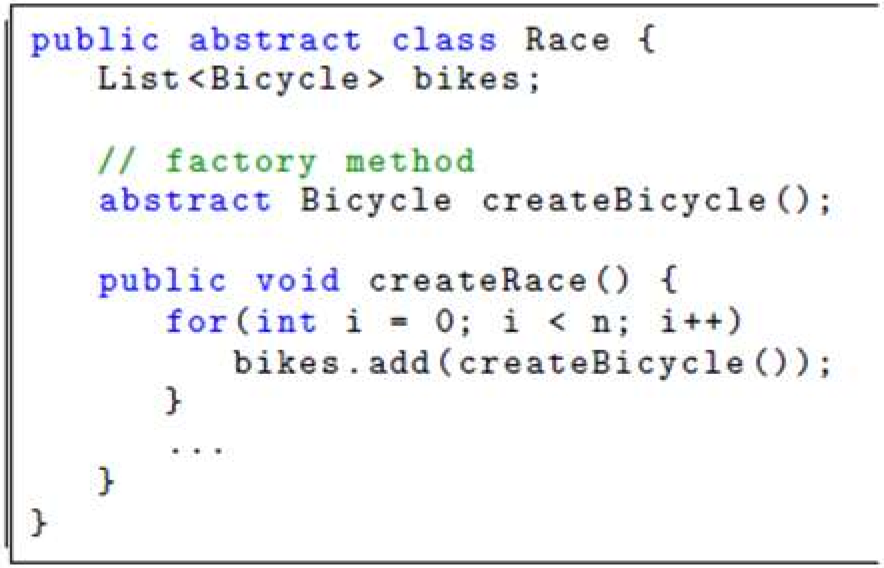
Solution

The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. Don't worry: the objects are still created via the new operator, but it's being called from within the factory method.

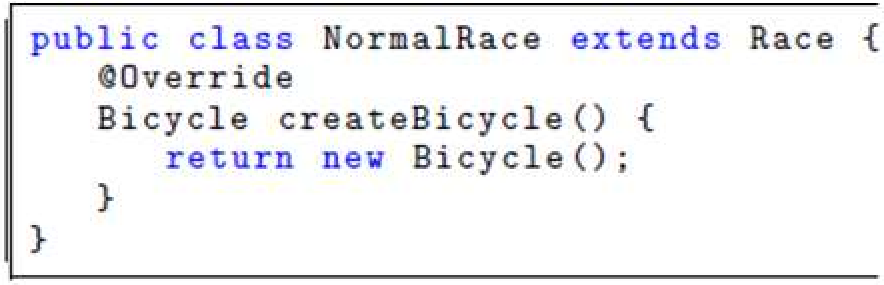
Good Solution



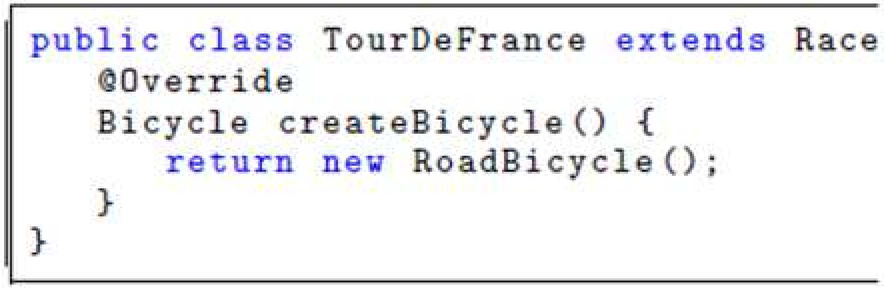
Good Solution



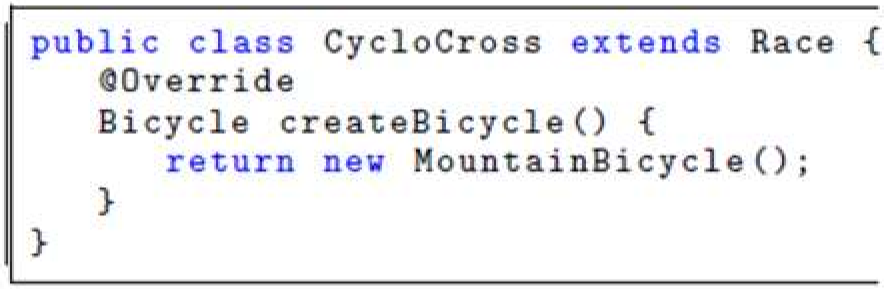
```java
public abstract class Race {
   List<Bicycle> bikes;

   // factory method
   abstract Bicycle createBicycle();

   public void createRace() {
      for(int i = 0; i < n; i++)
         bikes.add(createBicycle());
      }
      ...
   }
}
```



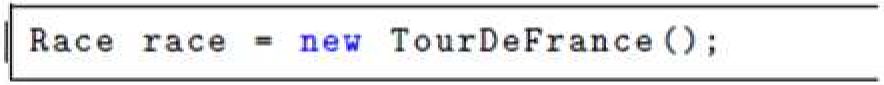
```java
public class NormalRace extends Race {
   @Override
   Bicycle createBicycle() {
      return new Bicycle();
   }
}
```



```java
public class TourDeFrance extends Race
   @Override
   Bicycle createBicycle() {
      return new RoadBicycle();
   }
}
```



```java
public class CycloCross extends Race {
   @Override
   Bicycle createBicycle() {
      return new MountainBicycle();
   }
}
```



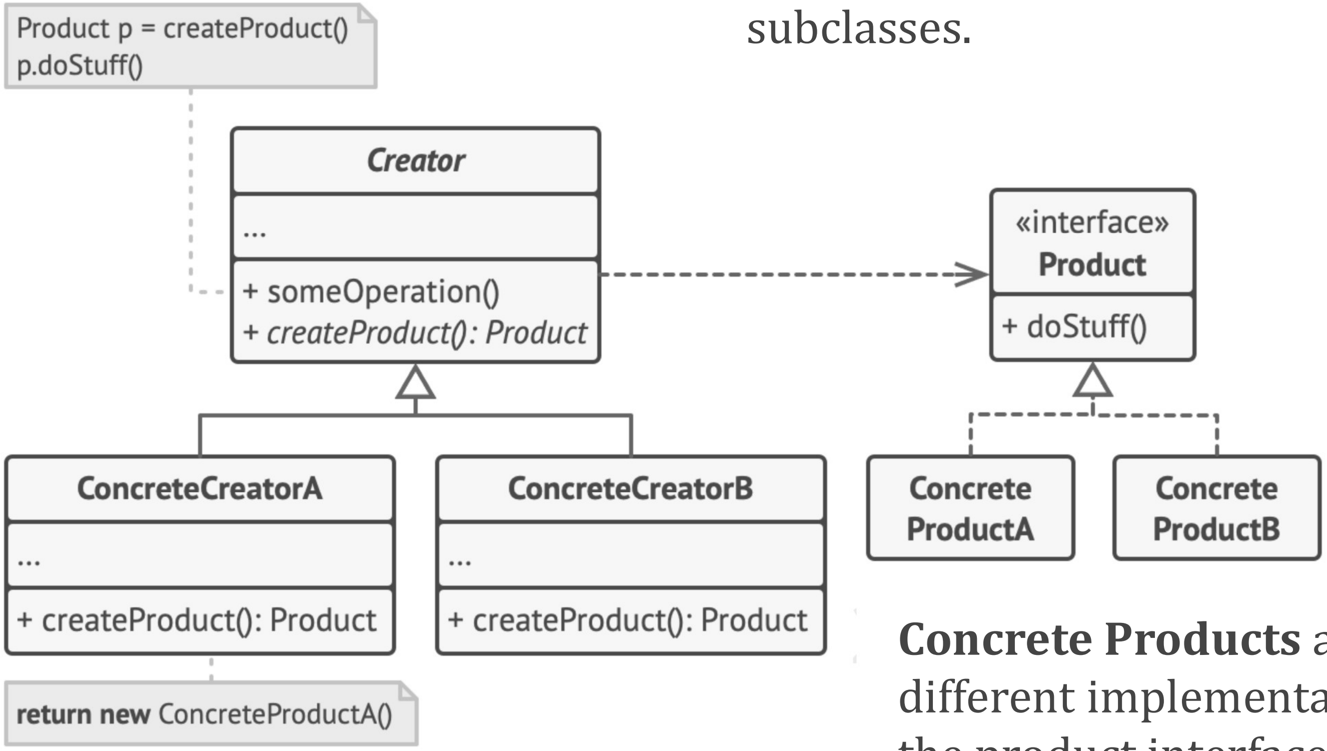
```java
Race race = new TourDeFrance();
```

```java
race.createRace()
```

Structure

The **Creator** class declares the factory method that returns new product objects. It’s important that the return type of this method matches the product interface.

The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.



Concrete Products are different implementations of the product interface.

Concrete Creators override the base factory method so it returns a different type of product.

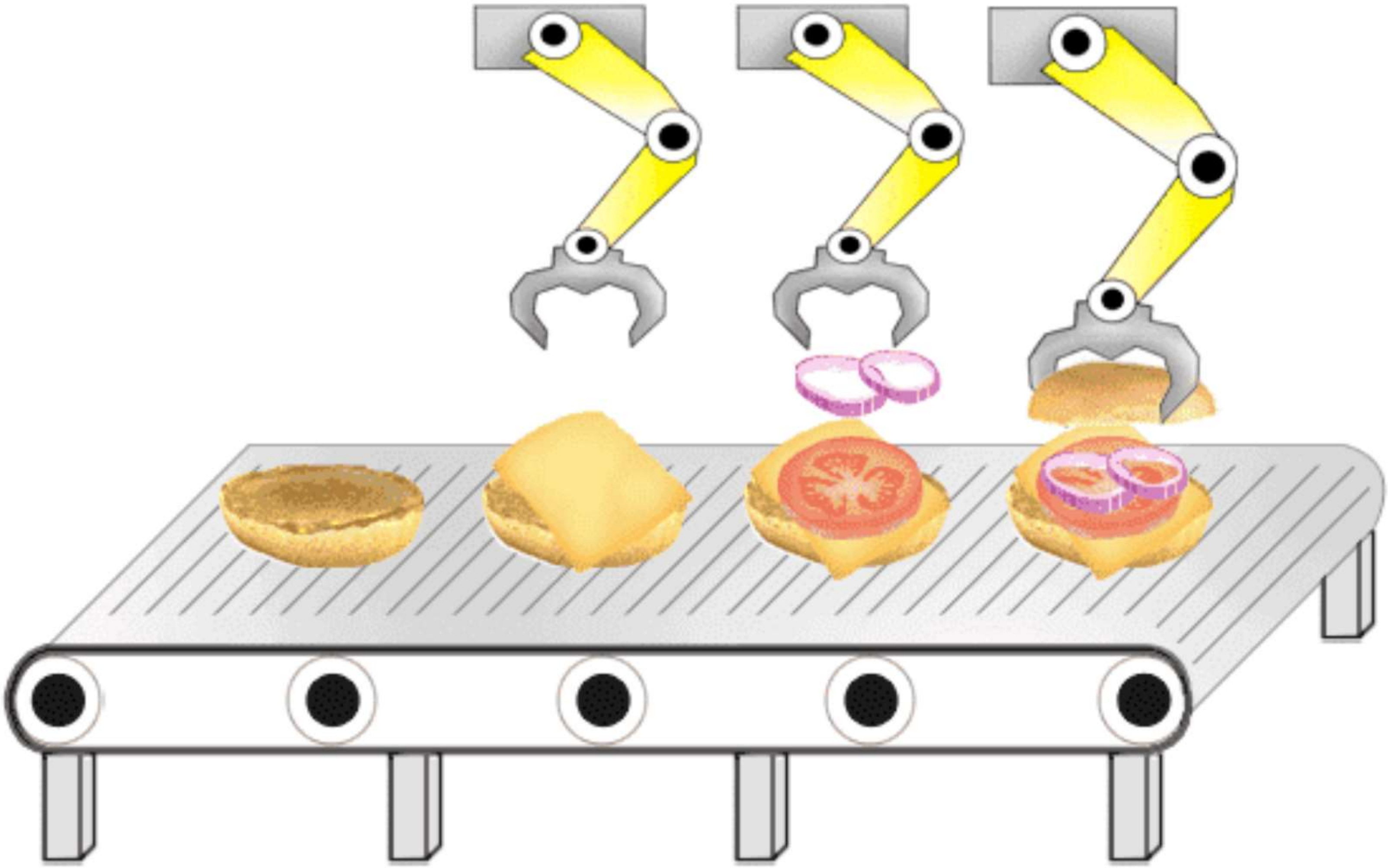
Plan

- Factory method
- Builder
- Singleton

Intent

Builder is a creational design pattern that lets you **construct complex objects** step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

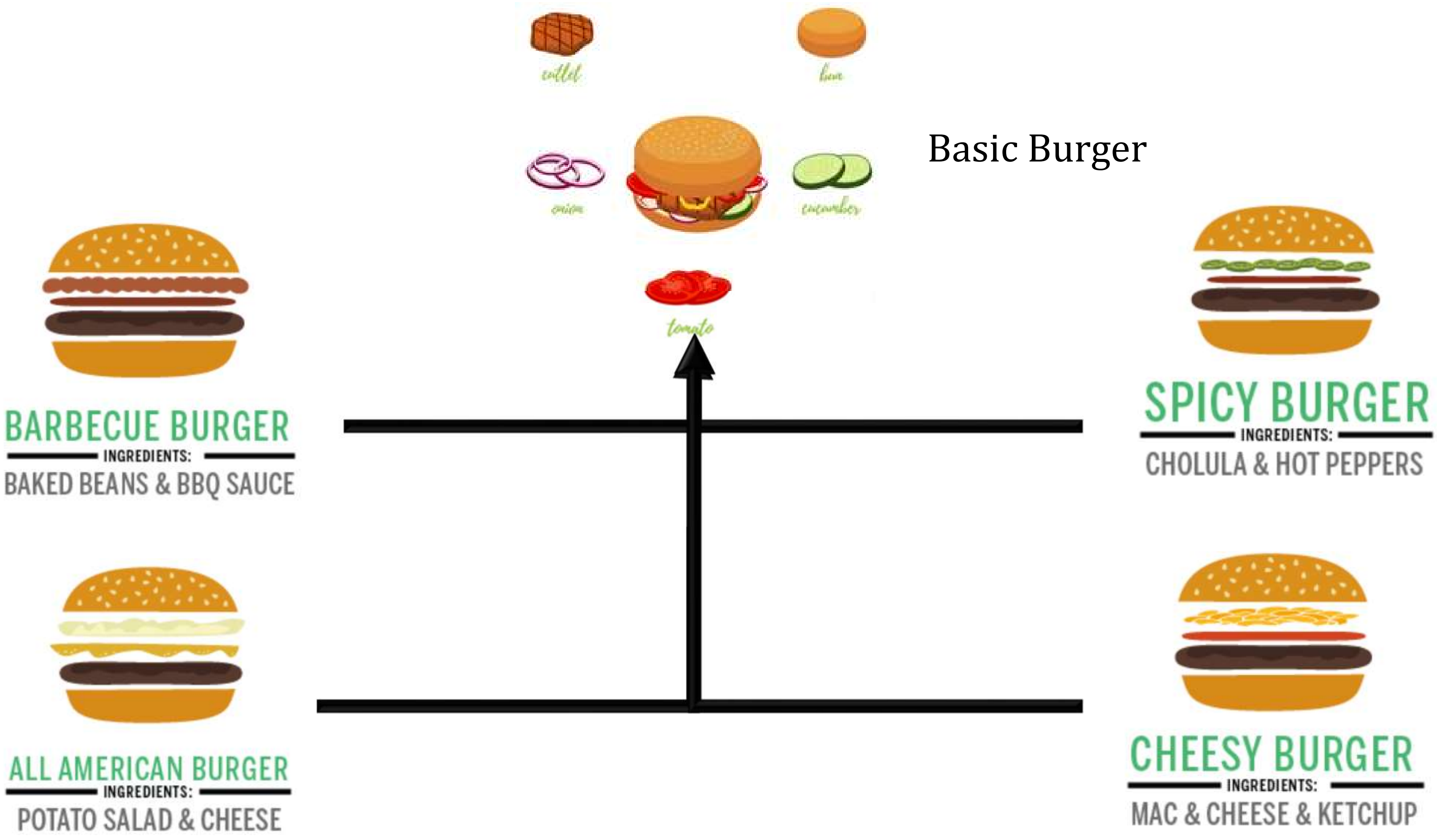
Example



The process of building the burger.

Lets you build complex objects step by step.

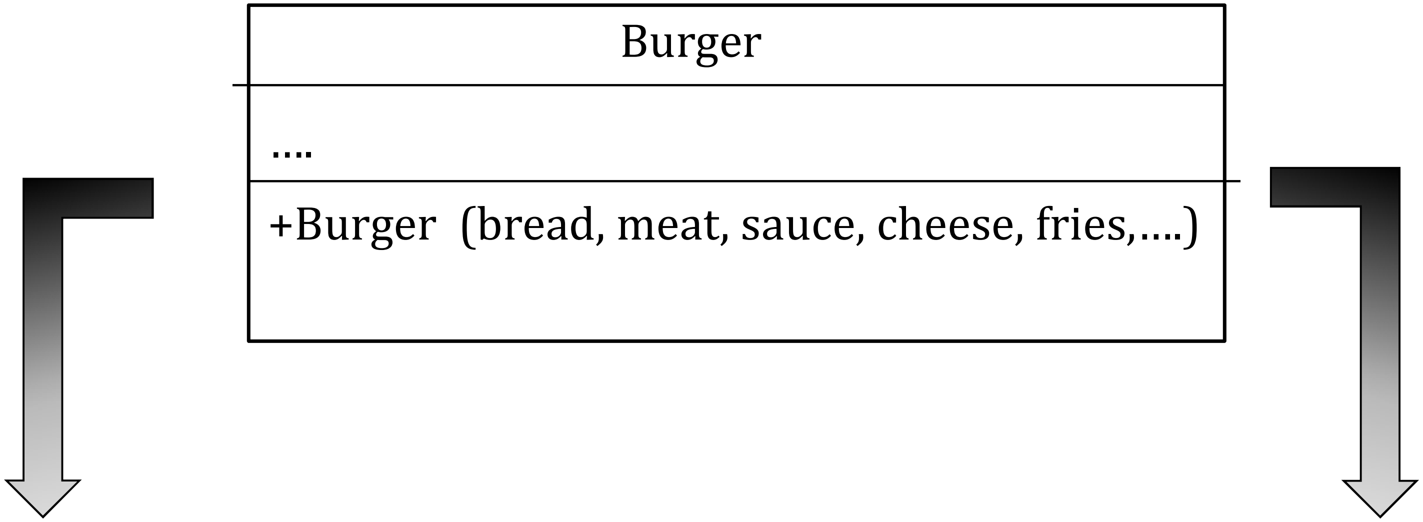
Problem-First Approach



Basic Burger

Creating a subclass for each possible configuration of an object may **make the program too complex..**

Problem-Second Approach



Burger

....

+Burger (bread, meat, sauce, cheese, fries,....)

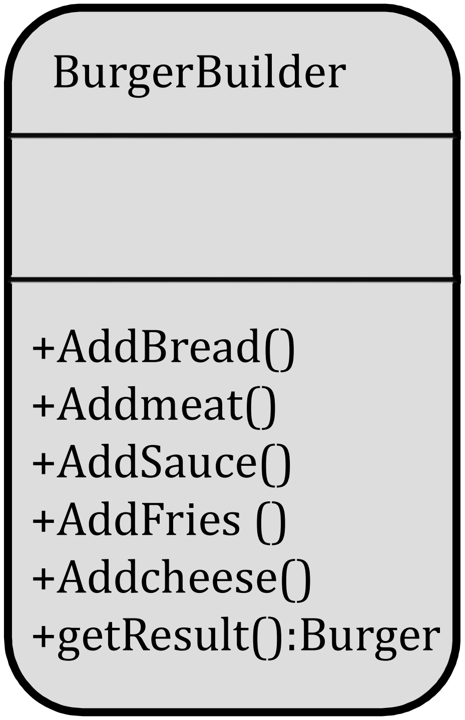
New Burger(True, True,True,False,True,....)

New Burger(True, True,True,True,False,....)

A constructor that has many parameters : not all of them are always used.

Solution

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.

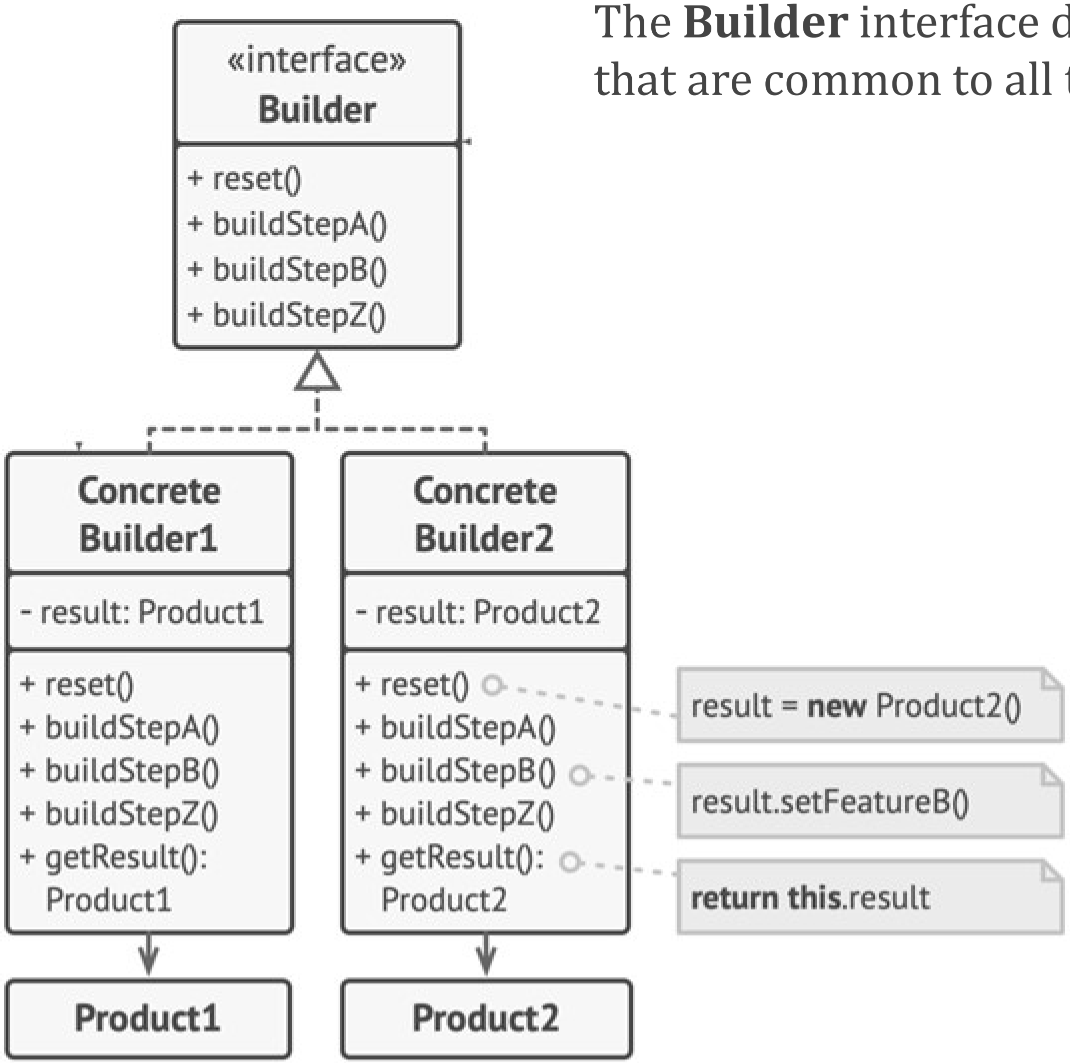


BurgerBuilder

+AddBread()
+Addmeat()
+AddSauce()
+AddFries ()
+Addcheese()
+getResult():Burger

In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process

Structure Part a



The **Builder** interface declares product construction steps that are common to all types of builders.

Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

Products are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.

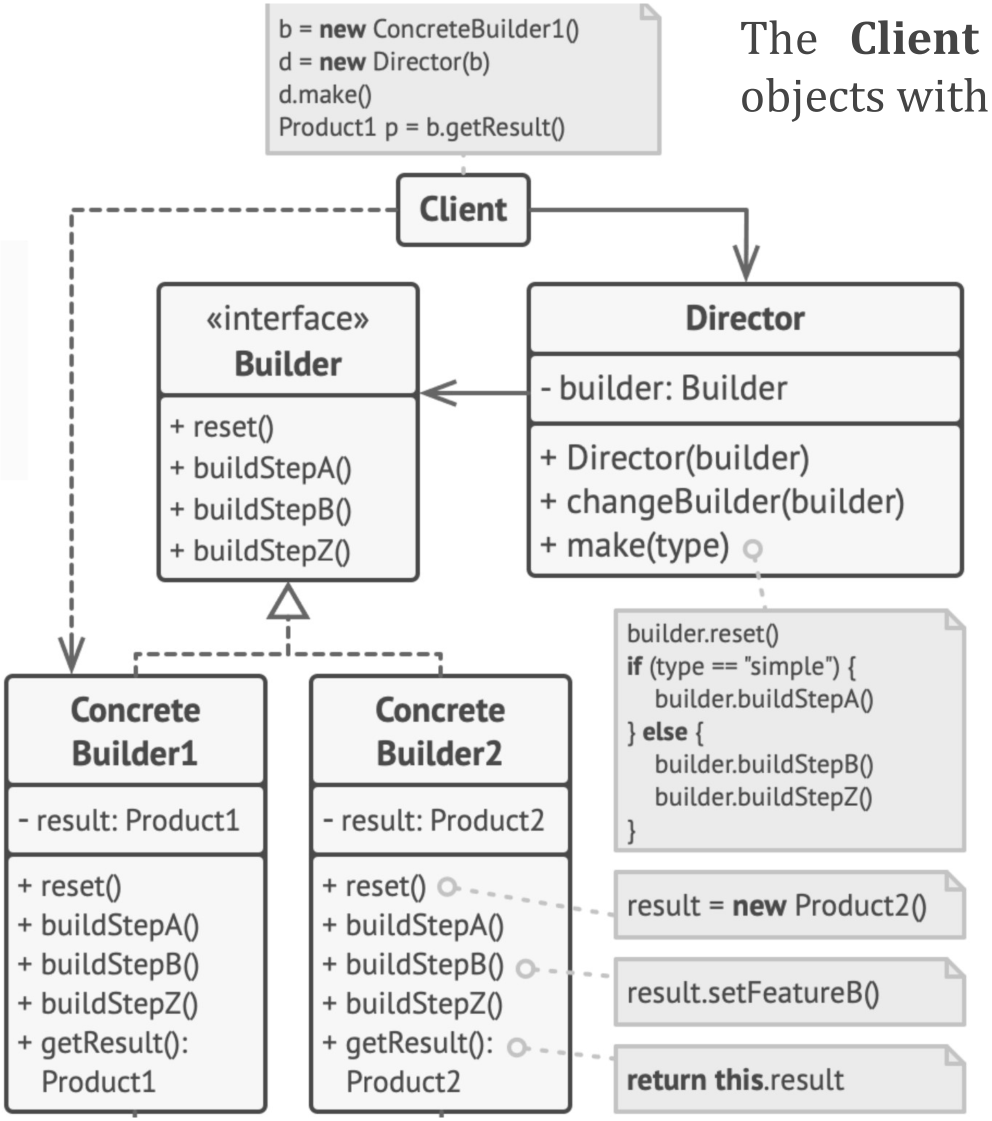
Director

The director class defines **the order in which to execute the building steps**, while the builder provides the implementation for those steps.

Having a director class in your program **isn’t strictly necessary.** You can always call the building steps in a specific order directly from the client code.

In addition, the director class completely **hides the details of product construction from the client code**. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

Structure



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

```
builder.reset()
if (type == "simple") {
    builder.buildStepA()
} else {
    builder.buildStepB()
    builder.buildStepZ()
}
```

```
result = new Product2()
```

```
result.setFeatureB()
```

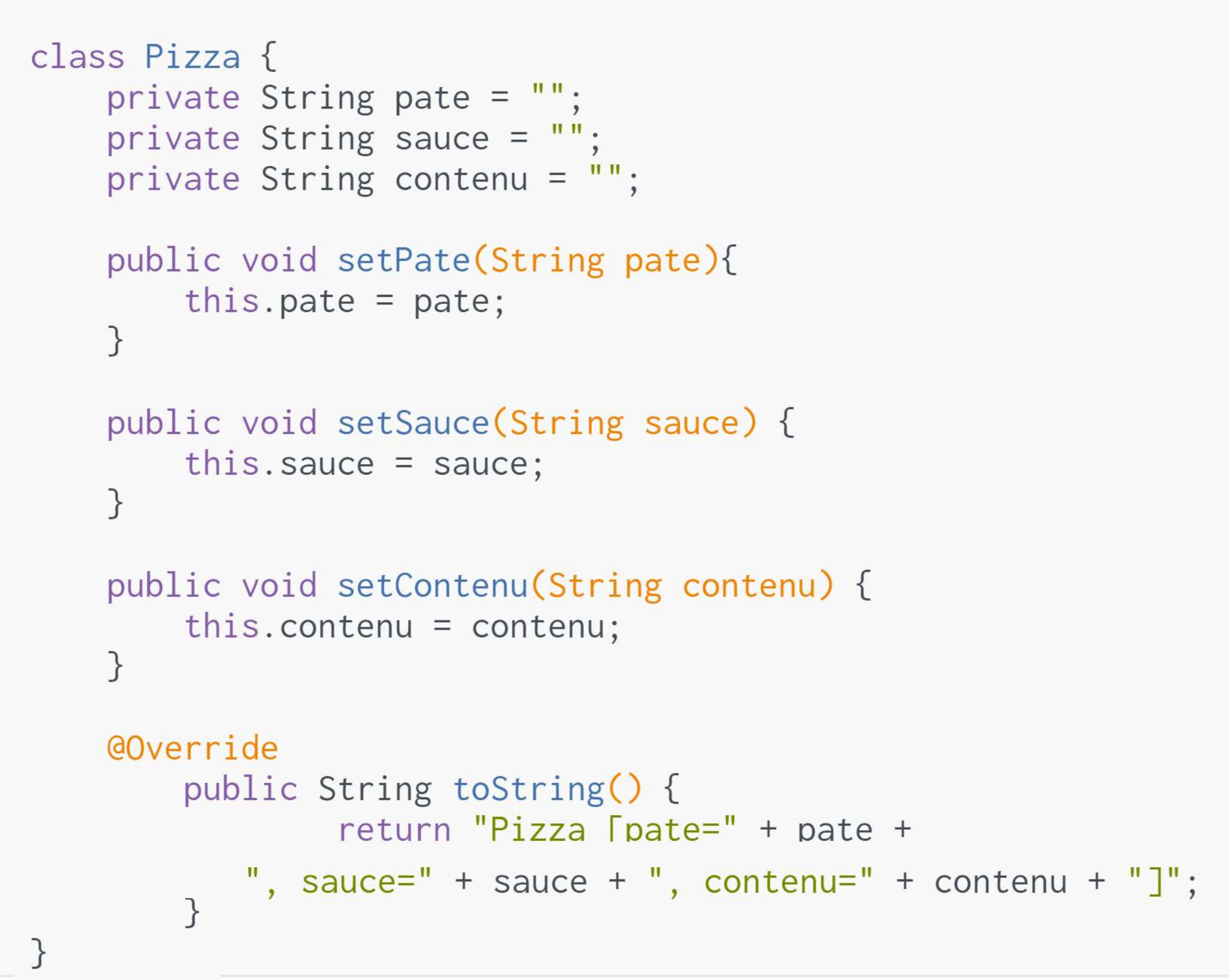
```
return this.result
```

The **Client** must associate one of the builder objects with the director.

The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

Example-PizzaBuilder

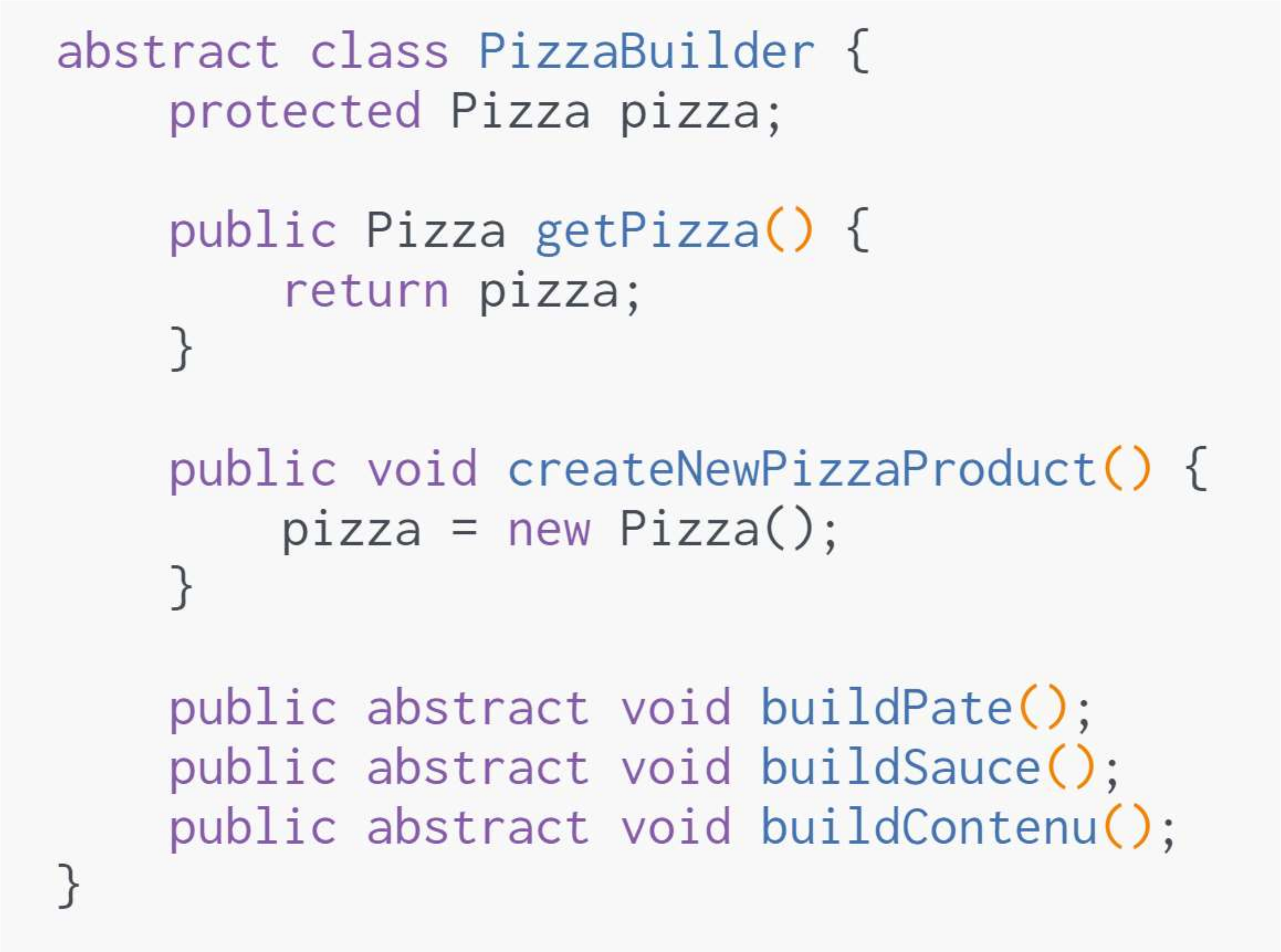
1) Product class



```java
class Pizza {
    private String pate = "";
    private String sauce = "";
    private String contenu = "";

    public void setPate(String pate){
        this.pate = pate;
    }

    public void setSauce(String sauce) {
        this.sauce = sauce;
    }

    public void setContenu(String contenu) {
        this.contenu = contenu;
    }

    @Override
        public String toString() {
                return "Pizza [pate=" + pate +
           ", sauce=" + sauce + ", contenu=" + contenu + "]";
        }
}
```

Example-PizzaBuilder

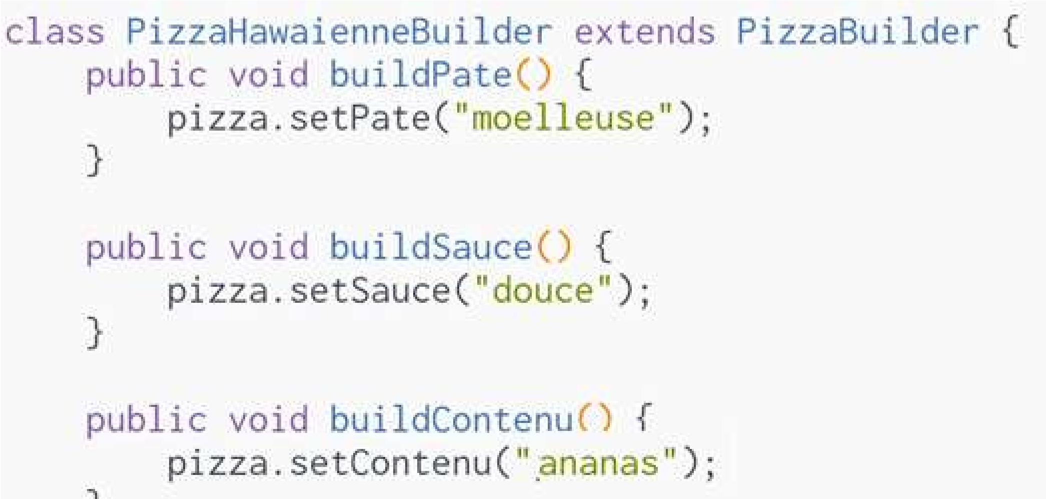
2) Abstract builder class



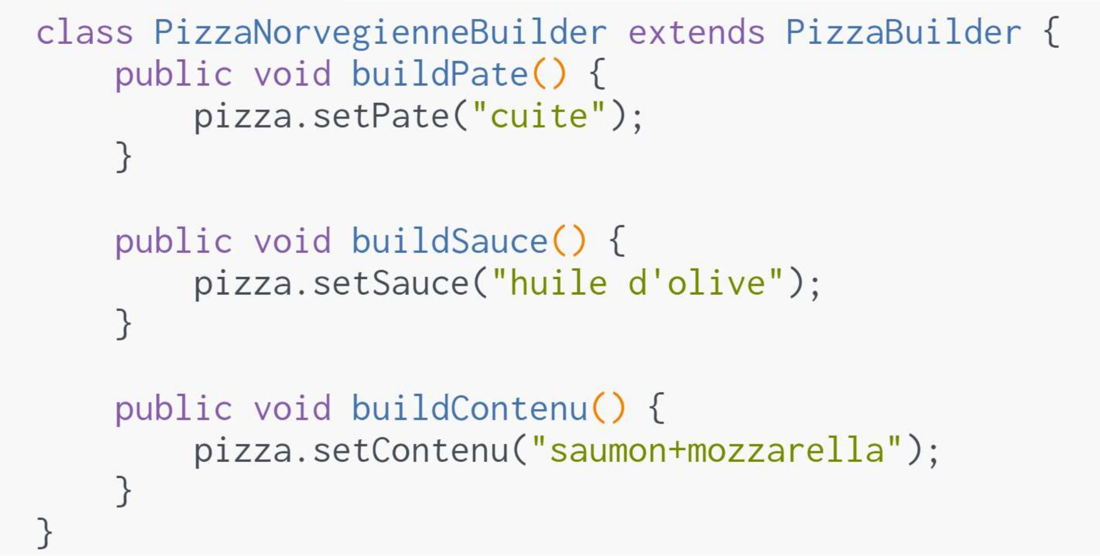
```
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }

    public abstract void buildPate();
    public abstract void buildSauce();
    public abstract void buildContenu();
}
```

Example-PizzaBuilder

3) Concrete builder class



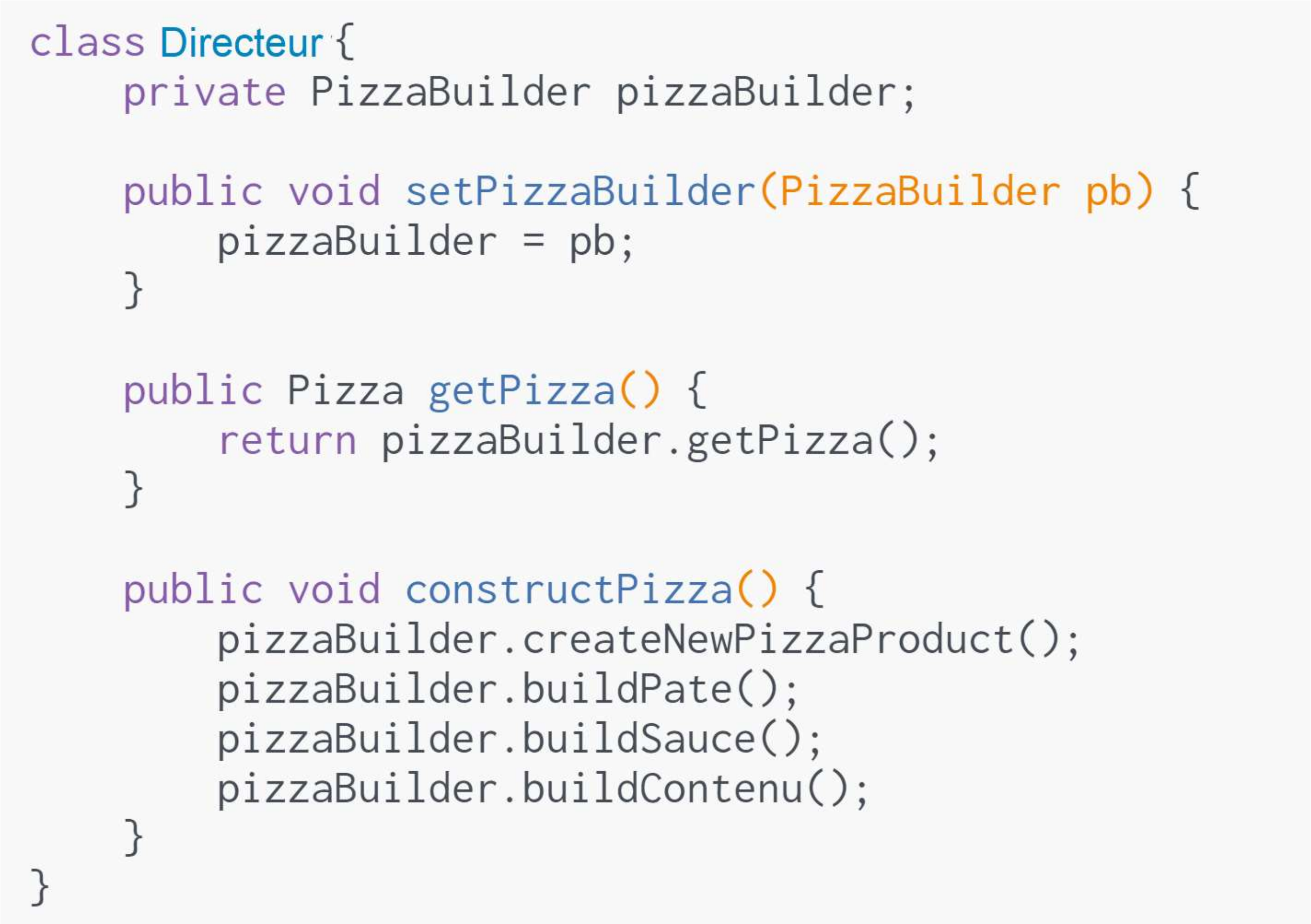
```
class PizzaHawaienneBuilder extends PizzaBuilder {
    public void buildPate() {
        pizza.setPate("moelleuse");
    }

    public void buildSauce() {
        pizza.setSauce("douce");
    }

    public void buildContenu() {
        pizza.setContenu("ananas");
    }
}
```



```
class PizzaNorvegienneBuilder extends PizzaBuilder {
    public void buildPate() {
        pizza.setPate("cuite");
    }

    public void buildSauce() {
        pizza.setSauce("huile d'olive");
    }

    public void buildContenu() {
        pizza.setContenu("saumon+mozzarella");
    }
}
```

Example-PizzaBuilder

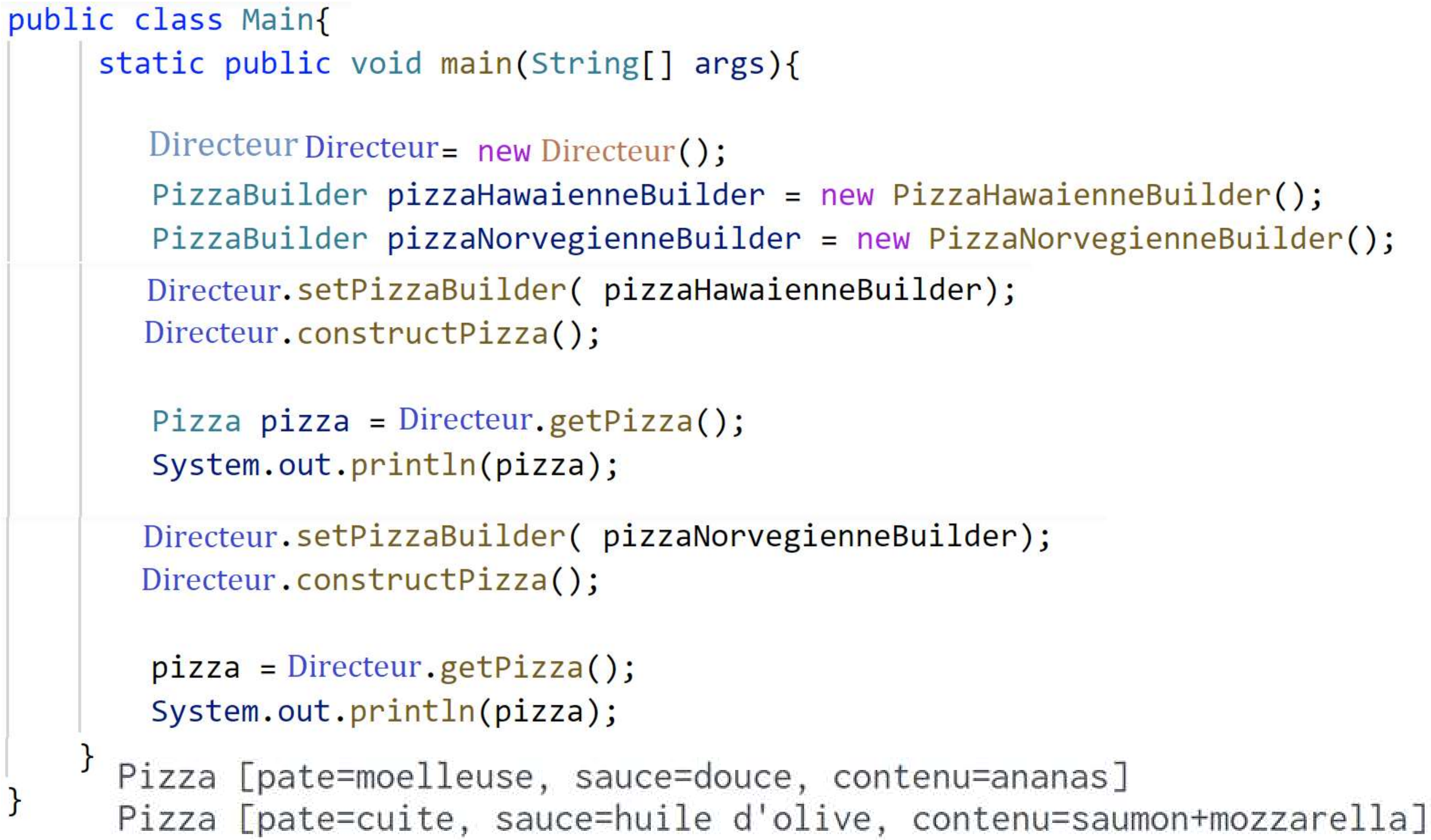
4) Director Class



```java
class Directeur {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }

    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }

    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildPate();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildContenu();
    }
}
```

Example-PizzaBuilder

5) Main Program



```
public class Main{
      static public void main(String[] args){

         Directeur Directeur= new Directeur();
         PizzaBuilder pizzaHawaienneBuilder = new PizzaHawaienneBuilder();
         PizzaBuilder pizzaNorvegienneBuilder = new PizzaNorvegienneBuilder();
         Directeur.setPizzaBuilder( pizzaHawaienneBuilder);
         Directeur.constructPizza();

         Pizza pizza = Directeur.getPizza();
         System.out.println(pizza);
         Directeur.setPizzaBuilder( pizzaNorvegienneBuilder);
         Directeur.constructPizza();

         pizza = Directeur.getPizza();
         System.out.println(pizza);
      }
}
```

```
Pizza [pate=moelleuse, sauce=douce, contenu=ananas]
Pizza [pate=cuite, sauce=huile d'olive, contenu=saumon+mozzarella]
```

Plan

- Factory method

- Builder

- Singleton

Intent

Singleton is a creational design pattern that lets you ensure that a class has **only one instance**, while providing a global access point to this instance

Problem

Case Study: In the context of a dynamic web application, the connection to the database server is unique. Imagine that you're creating a class that manages the database connection, the implementation of this class should allow clients to **access the same instance** of the database connection throughout the program

Solution

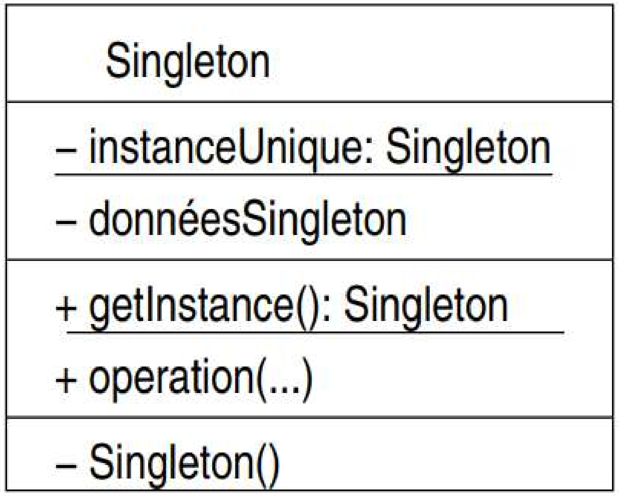
All implementations of the Singleton have these two steps in common:

1. **Make the default constructor private**, to prevent other objects from using the new operator with the Singleton class.

2. **Create a static creation method that acts as a constructor.** Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

Structure

Class Diagram



Singleton

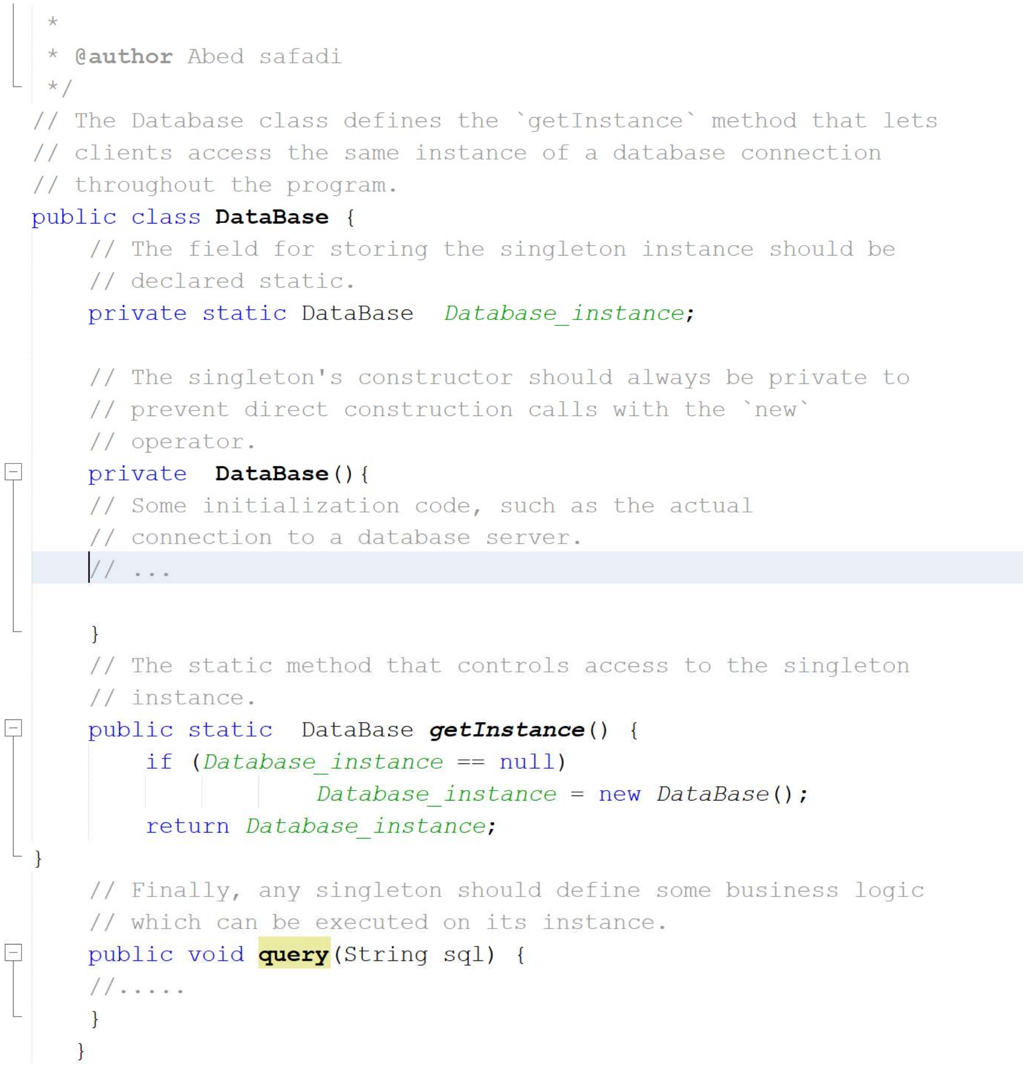
– instanceUnique: Singleton

– donnéesSingleton

+ getInstance(): Singleton

+ operation(...)

– Singleton()



```
 *
 * @author Abed safadi
 */
// The Database class defines the `getInstance` method that lets
// clients access the same instance of a database connection
// throughout the program.
public class DataBase {
    // The field for storing the singleton instance should be
    // declared static.
    private static DataBase  Database_instance;

    // The singleton's constructor should always be private to
    // prevent direct construction calls with the `new`
    // operator.
    private  DataBase(){
    // Some initialization code, such as the actual
    // connection to a database server.
    // ...

    }
    // The static method that controls access to the singleton
    // instance.
    public static  DataBase getInstance() {
        if (Database_instance == null)
                Database_instance = new DataBase();
        return Database_instance;
}
    // Finally, any singleton should define some business logic
    // which can be executed on its instance.
    public void query(String sql) {
    //.....
    }
   }
```

Questions?