



UNIVERSITÉ
DE MONTPELLIER



Design Patterns

Conception et Programmation Objet Avancées
M3105 (cours2)

Nadjib Lazaar (nadjib.lazaar@umontpellier.fr)

Creational Patterns

Les Patrons de Constructions

Creational Patterns

- Factory Pattern / Le Patron de Fabrique
- Builder Pattern / Le Patron Monteur
- Singleton Pattern / Le Patron Singleton

Creational Patterns

Factory Pattern / Patron de Fabrique

Isoler la création des objets

Découpler les classes concrètes de leur utilisateur

Définition d'un constructeur "virtuel"

Principe SOLID

- **Open/Closed:** Ouvert aux extensions, fermé aux modifications
- **Dependency Inversion:** Non-dépendant à l'inutile

Factory Pattern

Cas1 : Concessionnaire

```
/** Un vendeur de voitures */
public class Concessionnaire {

    /** Les marques vendues par le concessionnaire */
    public static enum Marque { Mercedes, Volkswagen, BMW};

    /** Commande la voiture demandée.
    @param type la marque du véhicule à commander
    @return la voiture demandée. */
    public Voiture Commande(Marque type){
        Voiture v ;
        if (type instanceof Mercedes) v = new ClasseA() ;
        else if (type instanceof Volkswagen) v = new Polo() ;
        else if (type instanceof BMW) v = new Serie1() ;
        v.immatriculer() ;
        return v ;
    }
}
```

Factory Pattern

Cas1 : Concessionnaire

```
/** Un vendeur de voitures */
public class Concessionnaire {

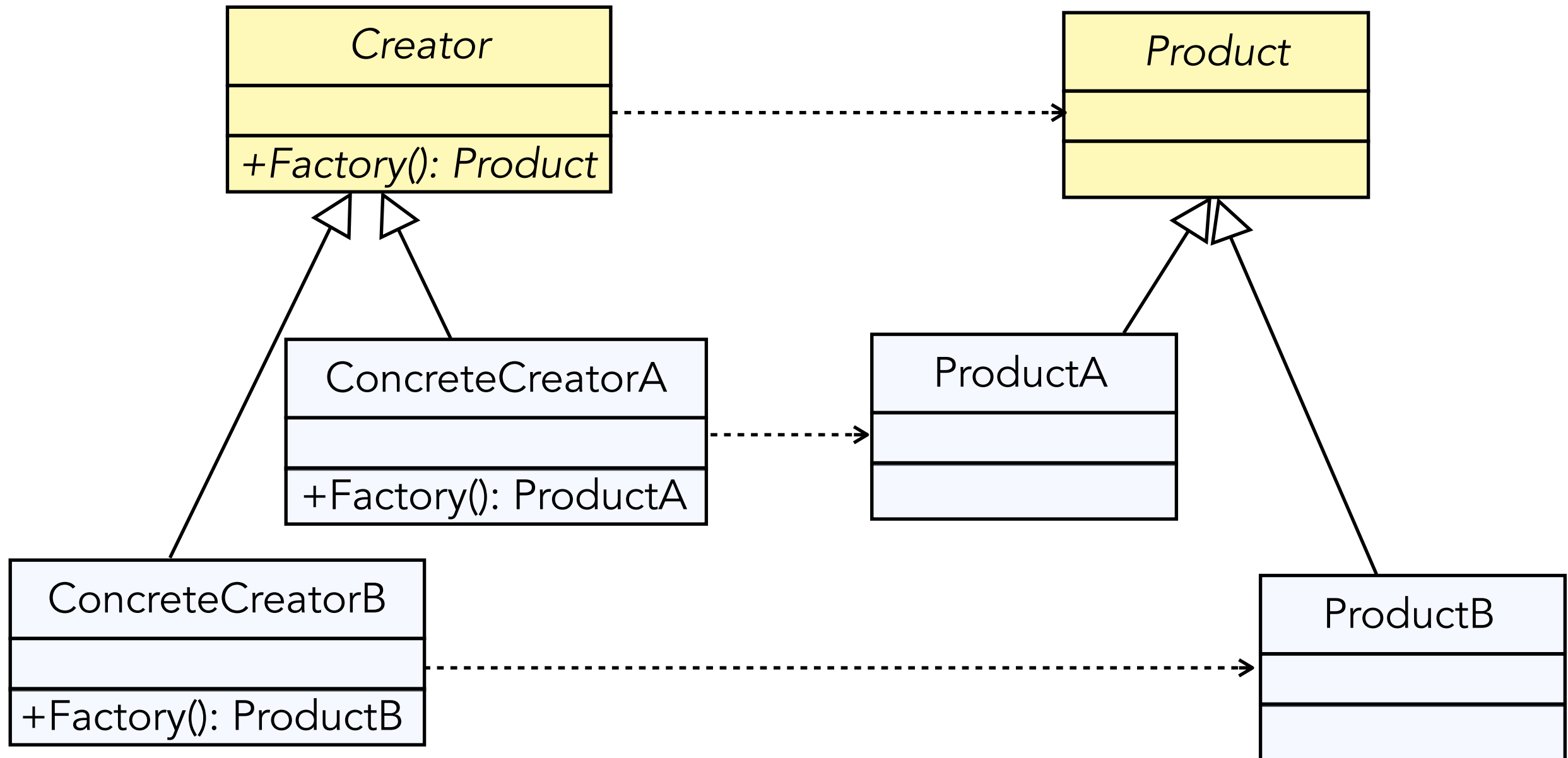
    /** Les marques vendues par le concessionnaire */
    public static enum Marque { Mercedes, Volkswagen, BMW};

    /** Commande la voiture demandée.
    @param type la marque du véhicule à commander
    @return la voiture demandée. */
    public Voiture Commande(Marque type){
        Voiture v ;
        if (type instanceof Mercedes) v = new ClasseA() ;
        else if (type instanceof Volkswagen) v = new Polo() ;
        else if (type instanceof BMW) v = new Serie1() ;
        v.immatriculer() ;
        return v ;
    }
}
```

Ajout de véhicule Audi =>
Violation du principe Open/closed

Cas1 : Concessionnaire

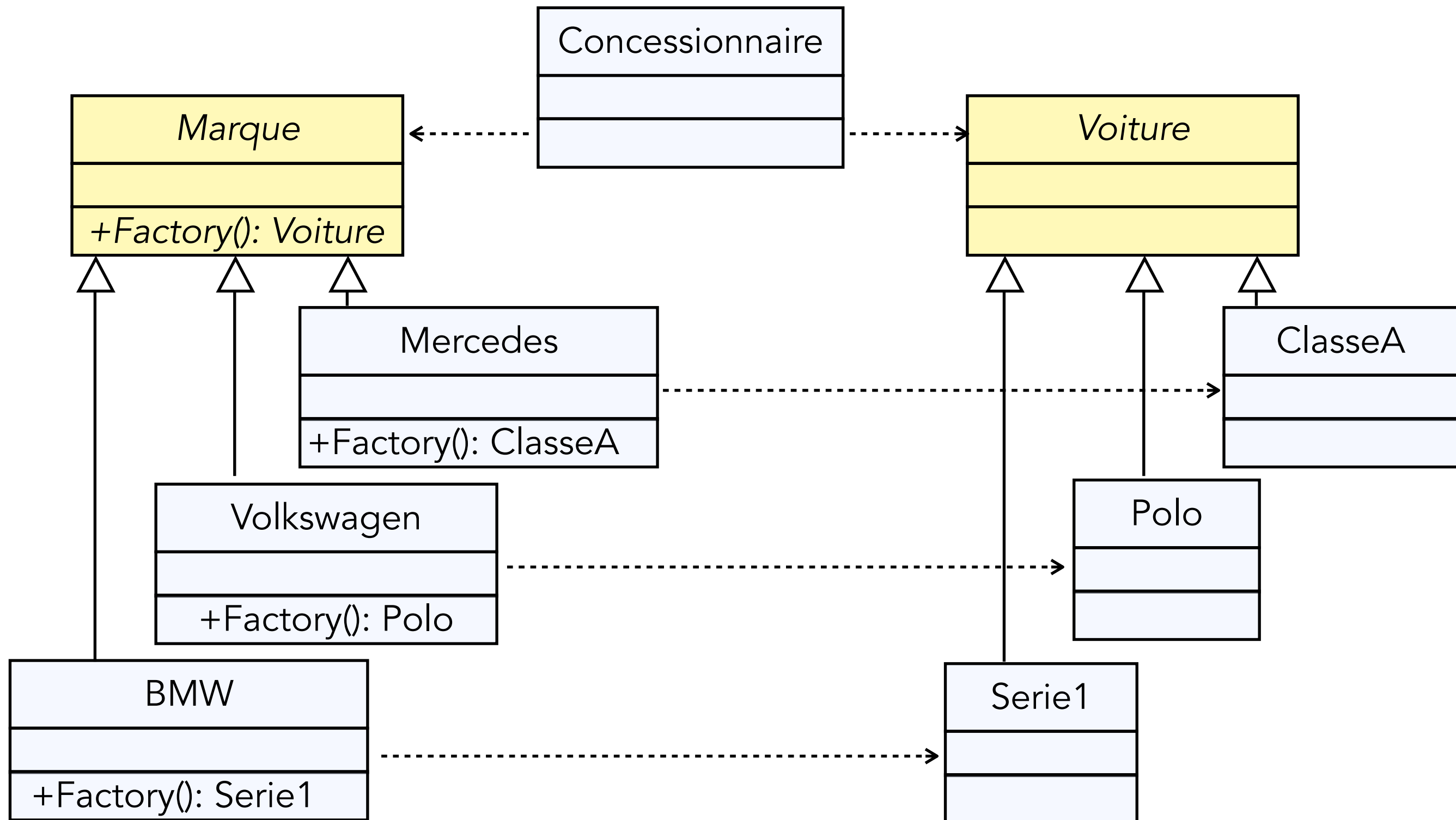
Pattern to use



Un créateur => Un type d'objet

Cas1 : Concessionnaire

UML



Cas1 : Concessionnaire

JAVA

```
/** Les marques vendues par le concessionnaire */
public class Marque {
    /** Fabrique une voiture de cette marque */
    public abstract Voiture fabrique();
};

public class Mercedes extends Marque {
    @Override
    public Voiture fabrique() { return new ClasseA(); }
};

public class Volkswagen extends Marque {
    @Override
    public Voiture fabrique() { return new Polo(); }
};

public class BMW extends Marque {
    @Override
    public Voiture fabrique() { return new Serie1(); }
};
```

Cas1 : Concessionnaire

JAVA

```
/** Les
public c
/** F
publi
};

public c
@Override
publi
};

public class Volkswagen extends Marque {
    @Override
    public Voiture fabriquer() { return new Polo(); }
};

public class BMW extends Marque {
    @Override
    public Voiture fabriquer() { return new Serie1(); }
};

/** Un vendeur de voitures */
public class Concessionnaire {
    /** Commande la voiture demandée.
    @param type la marque du véhicule à commander
    @return la voiture demandée. */
    public Voiture Commande(Marque type){
        Voiture v = type.fabriquer();
        v.immatriculer() ;
        return v ;
    }
}
```

Cas1 : Concessionnaire

JAVA

```
/** Les  
public class  
/** F  
public  
};  
  
public class  
@Override  
public  
};  
  
/** Un vendeur de voitures */  
public class Concessionnaire {  
    /** Commande la voiture demandée.  
    @param type la marque du véhicule à commander  
    @return la voiture demandée. */  
    public Voiture Commande(Marque type){  
        Voiture v = type.fabriquer();  
        v.immatriculer();  
        return v;  
    }  
}
```

```
public class Volkswagen extends Marque {  
    @Override  
    public Voiture fabriquer() { return new Polo(); }  
}  
  
public class Citroen extends Marque {  
    @Override  
    public Voiture fabriquer() { return new Spacetourer(); }  
};  
}
```

Factory Pattern

Cas2 : Pizzeria

```
/** Une pizzeria */
public class PizzaStore{

    /** Prépare la pizza dont le type est passé en argument.
    @return la pizza préparée. */
    public PizzaStore getPizza(String type){

        PizzaStore pizza ;

        if (type.equals("cheese")) pizza = new CheesePizza() ;

        else if (type.equals("pepperoni")) pizza = new PepperoniPizza() ;

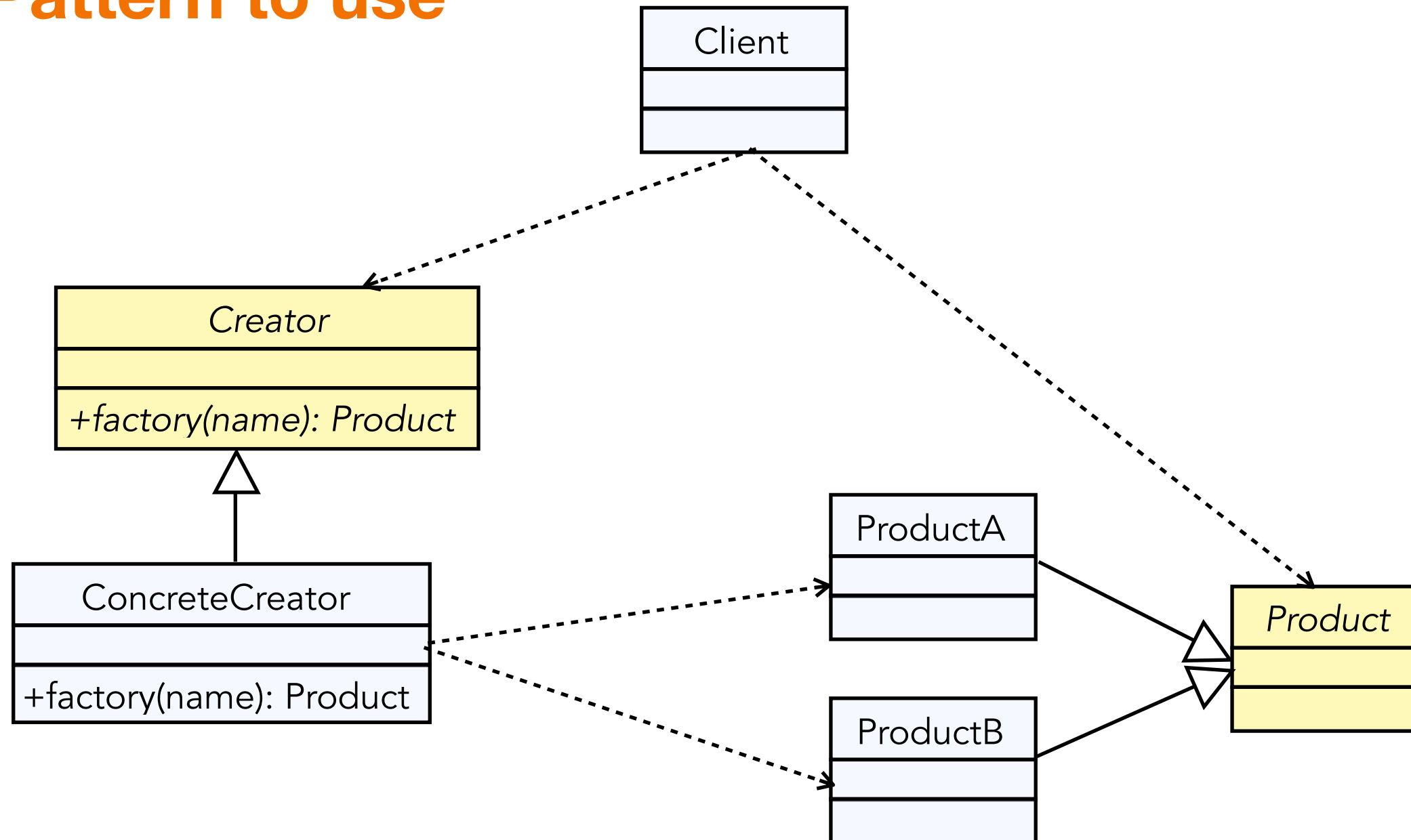
        else if (type.equals("clam")) pizza = new ClamPizza() ;

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

        return pizza ;
    }
}
```

Cas2 : Pizzeria

Pattern to use



Un créateur => différents types de produits

Cas2 : Pizzeria

JAVA

```
/** La fabrique à pizzas */
public class PizzaFactory{

    /** Crée la pizza dont le type est passé en argument. */
    public static Pizza getPizza(String type){

        Pizza pizza ;

        if (type.equals("cheese")) pizza = new CheesePizza() ;
        else if (type.equals("pepperoni")) pizza = new PepperoniPizza() ;
        else if (type.equals("clam")) pizza = new ClamPizza() ;
        return pizza ;
    }

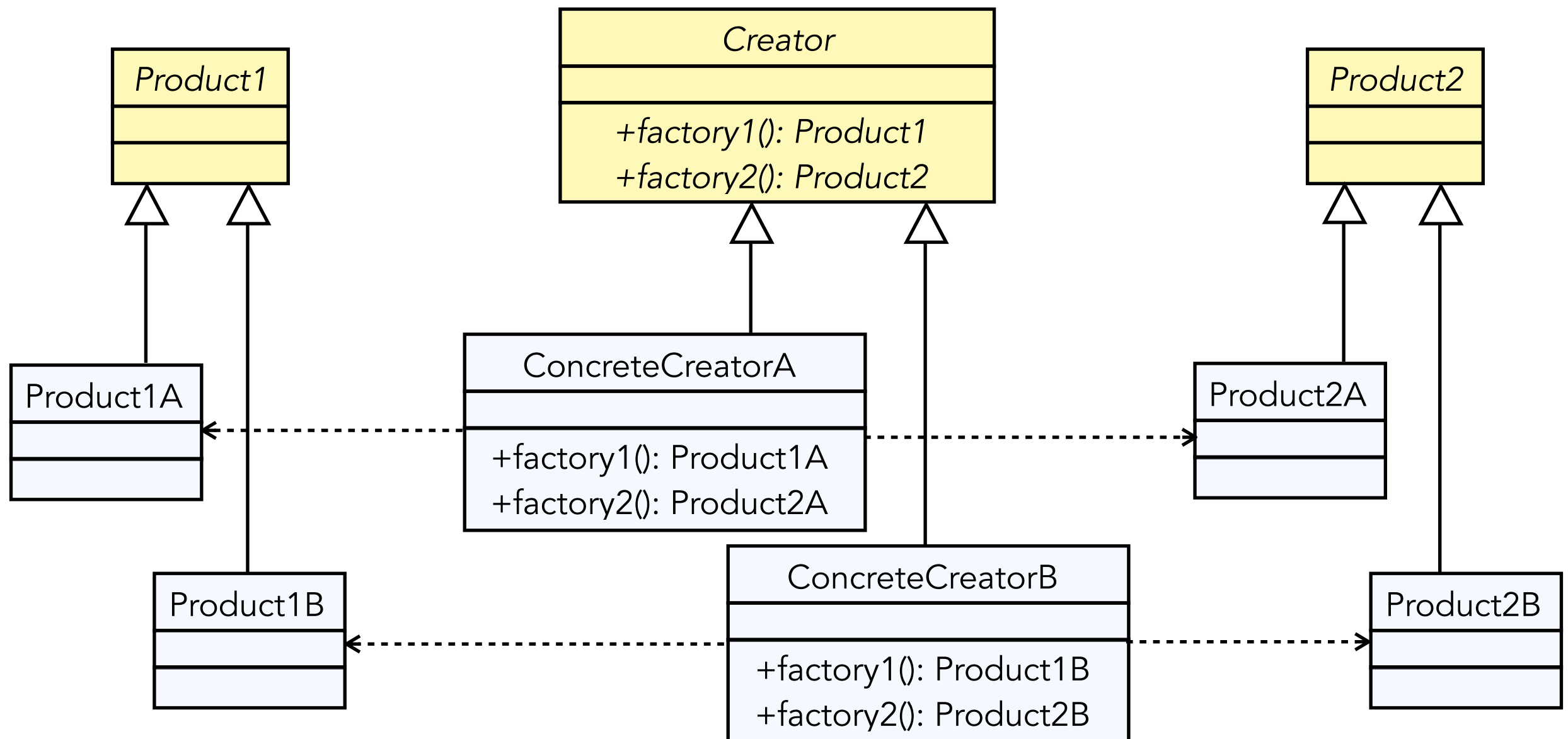
    /** Une pizzeria */
    public class PizzaStore{

        /** Prépare la pizza dont le type est passé en argument. */
        public Pizza getPizza(String type){
            Pizza pizza = PizzaFactory.createPizza(type);
            pizza.prepare() ;
            pizza.bake() ;
            pizza.cut() ;
            pizza.box() ;

            return pizza ;
        }
    }
}
```

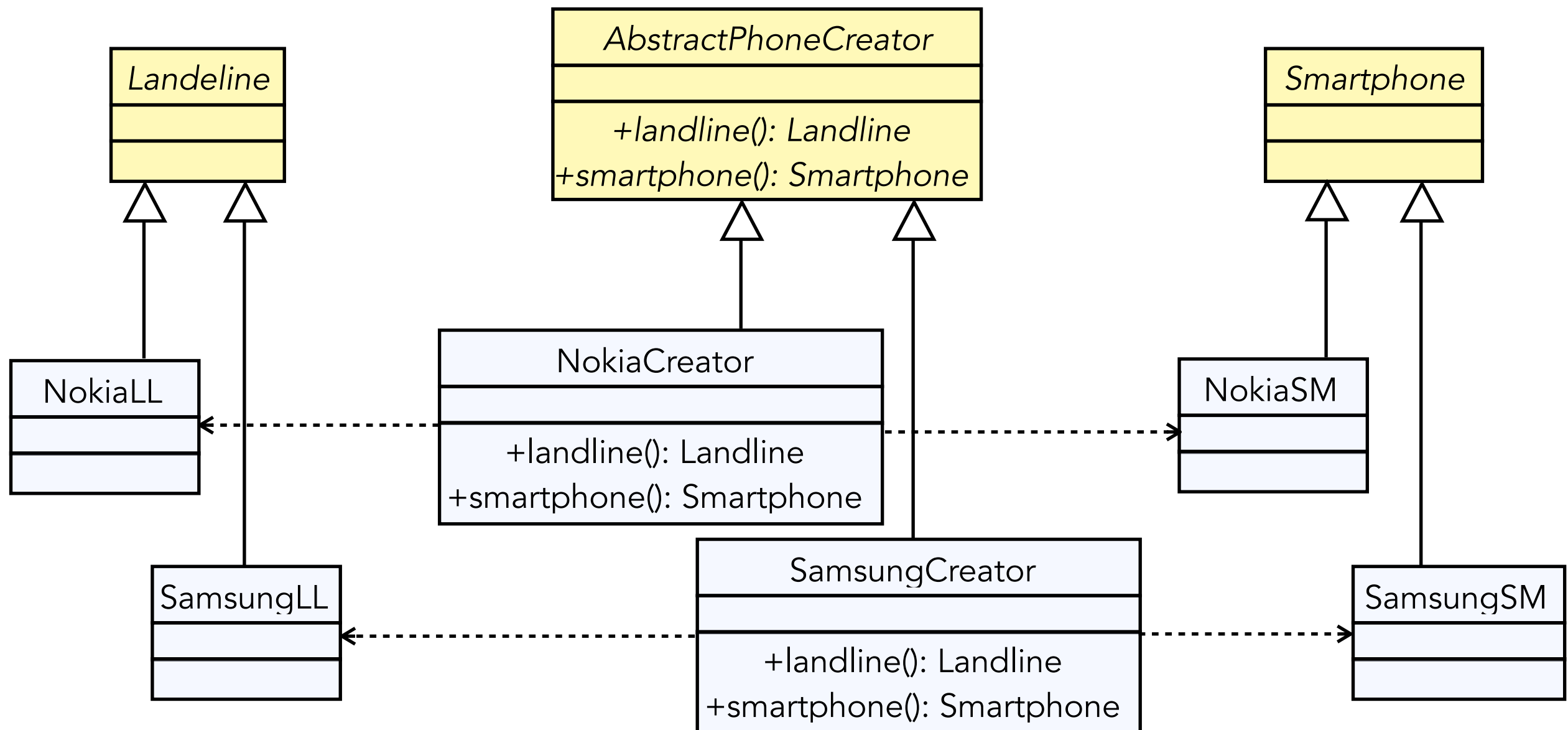
Creational Patterns

Abstract Factory Pattern / Patron Fabrique Abstraite



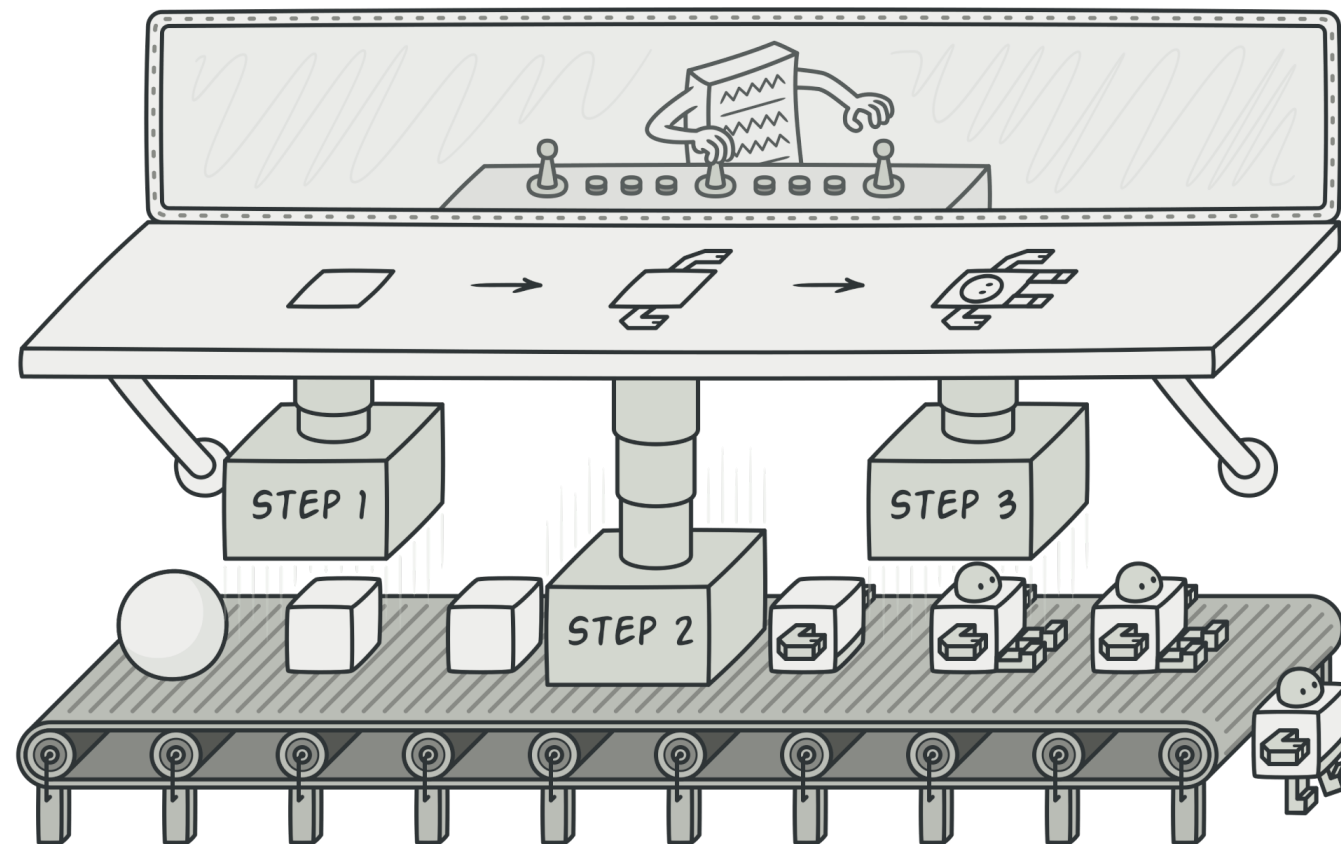
Abstract Factory Pattern

Example



Builder Pattern

Le Patron Monteur



[Dive Into Design Patterns. Alexander Shvets. 2019]

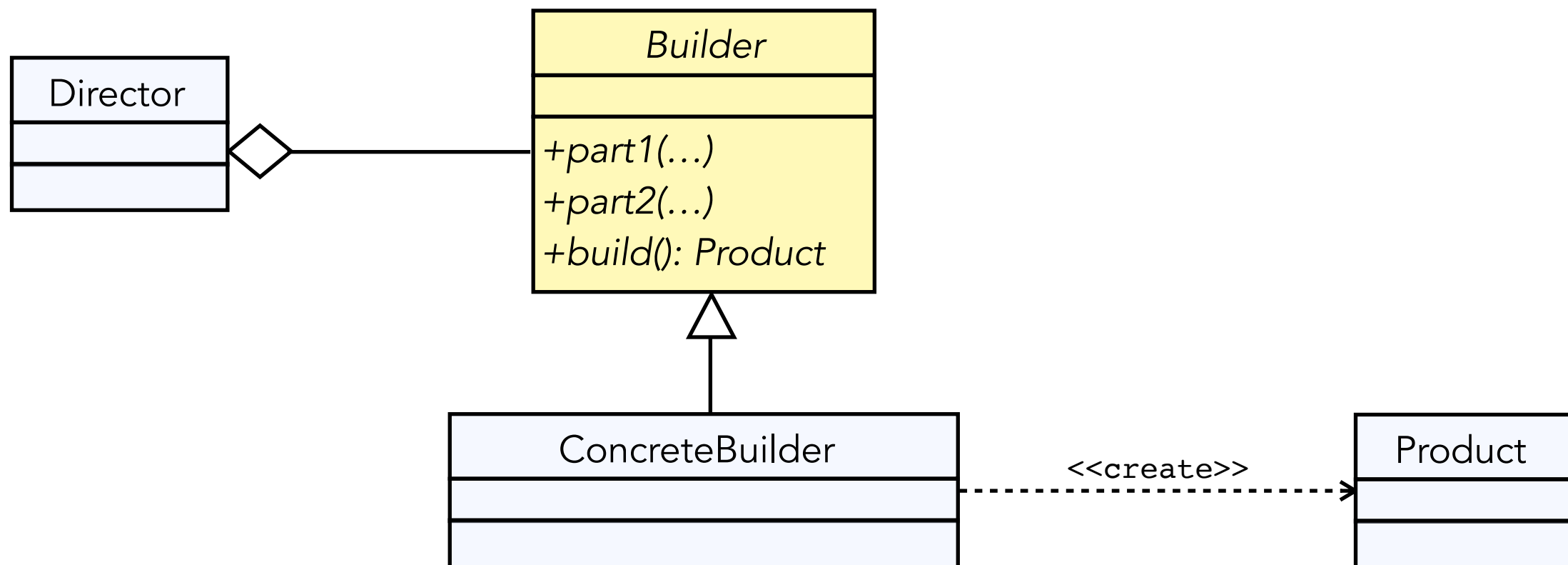
Builder Pattern

Le Patron Monteur

- Rendre facile la construction d'un objet complexe, avec beaucoup d'arguments dont certains sont optionnels.
- Encapsuler la façon dont un objet complexe est construit.
- Permet aux objets d'être construits en plusieurs et différentes étapes (contrairement à la Factory).
- Le client n'a pas accès à la représentation interne du produit.
- Souvent utilisé pour la construction des composites.
- La construction d'objets nécessite plus de connaissances comparant à la Factory.

Builder Pattern

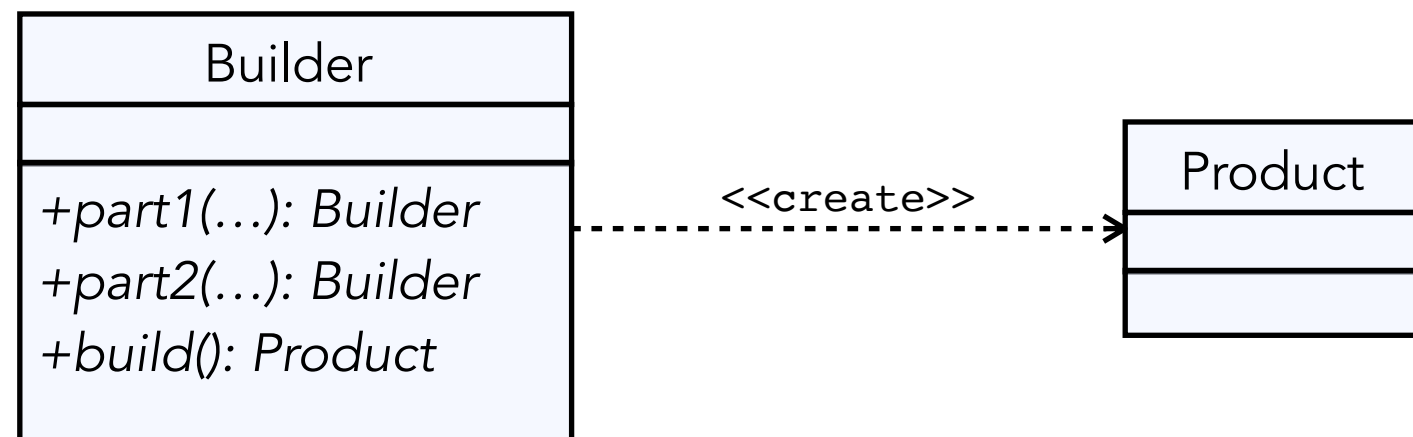
Le Patron Monteur



Fluent Builder Pattern

Le Patron Monteur Fluent

Une variante Fluent sans Directeur ni classe abstraite est beaucoup plus utilisée.



Fluent Builder Pattern

Le Patron Monteur Fluent

```
Pizza custom = new Pizza.Builder(Size.Large)
    .withMeat(Meat.Chicken)
    .withVegetable(Vegetable.Tomato)
    .withOlives(Olive.Black)
    .withCheese(Cheese.Mozzarella)
    .withCheese(Cheese.Mozzarella) // extra topping!
    .bake();
```

Singleton Pattern

Le Patron Singleton

- Pattern qui permet l'unicité d'un objet
- Assure qu'une classe ne peut avoir qu'un seul objet
- Assure un accès global à une unique et même instance

Singleton Pattern

Le Patron Singleton

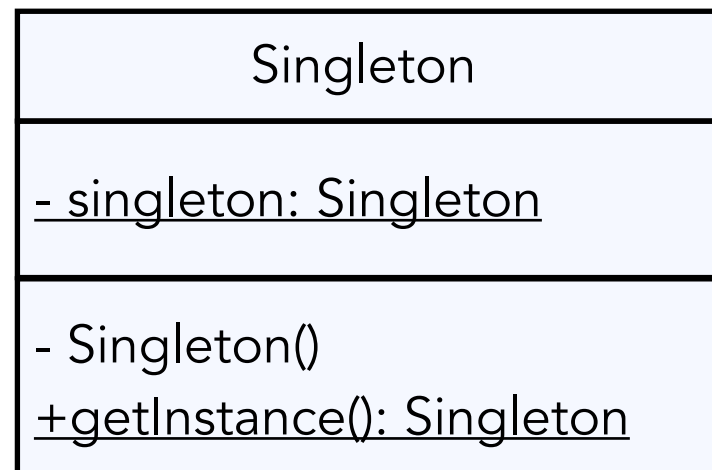
- Pattern qui permet l'unicité d'un objet
- Assure qu'une classe ne peut avoir qu'un seul objet
- Assure un accès global à une unique et même instance

Comment ?

- Pour empêcher d'autres objets d'utiliser le constructeur de la classe Singleton, il faut rendre le constructeur de la classe par défaut privé.
- Ajouter une méthode statique qui agit comme un constructeur. Cette méthode doit appeler le constructeur privé pour créer un objet et le mettre dans un attribut statique.

Singleton Pattern

Le Patron Singleton (UML)



Singleton Pattern

Le Patron Singleton (JAVA)

```
// Java program implementing Singleton class
// with getInstance() method
class Singleton
{
    // static variable single_instance of type Singleton
    private static Singleton single_instance = null;

    // private constructor restricted to this class itself
    private Singleton()
    {
        ...
    }

    // static method to create instance of Singleton class
    public static Singleton getInstance()
    {
        if (single_instance == null)
            single_instance = new Singleton();

        return single_instance;
    }
}
```

Singleton Pattern

Le Patron Singleton (exemple)

```
class Captain
{
    private static Captain captain;
    //We make the constructor private to prevent the use of "new"
    private Captain() { }
    public static synchronized Captain getCaptain(){
        // Lazy initialization
        if (captain == null)
        {
            captain = new Captain();
            System.out.println("New captain is elected for your team.");
        }
        else
        {
            System.out.print("You already have a captain for your team.");
            System.out.println("Send him for the toss.");
        }
        return captain;
    }
}

// We cannot extend Captain class. The constructor is private in this case.
//class B extends Captain{}// error
```

Singleton Pattern

Le Patron Singleton (exemple)

```
public class SingletonPatternExample {  
    public static void main(String[] args) {  
  
        System.out.println("***Singleton Pattern Demo***\n");  
        System.out.println("Trying to make a captain for your team:");  
  
        //Constructor is private. We cannot use "new" here.  
        //Captain c3 = new Captain();//error  
  
        Captain captain1 = Captain.getCaptain();  
  
        System.out.println("Trying to make another captain for your team:");  
        Captain captain2 = Captain.getCaptain();  
  
        if (captain1 == captain2)  
        {  
            System.out.println("captain1 and captain2 are same instance.");  
        }  
    }  
}
```

Books

Design Pattern

- **Design Patterns: Elements of Reusable Object-Oriented Software.** Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (GoF: Gang of Four).
- **DESIGN PATTERNS Explained simply.** Alexander Shvets. 2013
- **Dive Into Design Patterns.** Alexander Shvets. 2019
- **Head First Design Patterns.** Freeman et al. 2014
- **Java Design Patterns.** Vaskaran Sarcar. 2019

