DESIGN PATTERNS
FABRIQUES / BUILDERS

Yohan BOICHUT
Université d'Orléans

M2 INSA

année 2019/2020

Encore un problème à résoudre

Avec une nouvelle illustration du proverbe :

Les interfaces c’est bien

Les classes concrètes c’est mal

Nouveau problème…

- new et changement

Canard canard = new Colvert( );

Interfaces -> souplesse

en opposition avec

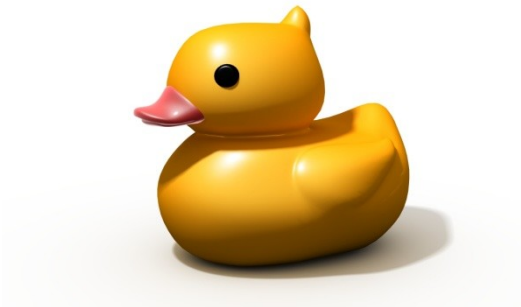
new Classe concrète

Illustration

- Pfff, encore des canards ???
- Non, Une Pizzeria !

Commande de pizzas


```
public class Pizzeria {
  …
  public Pizza commanderPizza(String type) {
    Pizza pizza = null;
    if (type.equals("fromage")) {
      pizza = new PizzaFromage();
    } else if (type.equals("poivrons")) {
      pizza = new PizzaPoivrons();
    } else if (type.equals("vegetarienne")) {
      pizza = new PizzaVegetarienne();
    }
    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
  }
```



Etape 1

- Identifier et isoler ce qui varie ! Coin Coin !

```
public class SimpleFabriqueDePizzas {
   public Pizza creerPizza(String type) {
      Pizza pizza = null;

      if (type.equals("fromage")) {
        pizza = new PizzaFromage();
      } else if (type.equals("poivrons")) {
        pizza = new PizzaPoivrons();
      } else if (type.equals("fruitsDeMer")) {
        pizza = new PizzaFruitsDeMer();
      } else if (type.equals("vegetarienne")) {
        pizza = new PizzaVegetarienne();
      }
      return pizza;
   }
}
```

Etape 2

- Modifier Pizzeria :

```
public class Pizzeria {
   SimpleFabriqueDePizzas fabrique;

   public Pizzeria(SimpleFabriqueDePizzas fabrique) {
   this.fabrique = fabrique;
   }
   public Pizza commanderPizza(String type) {
   Pizza pizza;

   pizza = fabrique.creerPizza(type);

   pizza.preparer();
   pizza.cuire();
   pizza.couper();
   pizza.emballer();

   return pizza;
   }
}
```

Plusieurs pizzerias


```
Pizzeria pizzeriaBrest = new PizzeriaBrest();
Pizzeria pizzeriaStrasbourg = new PizzeriaStrasbourg();

Pizza pizza = pizzeriaBrest.commanderPizza("fromage");
System.out.println("Luc a commandé une " + pizza + "\n");

pizza = pizzeriaStrasbourg.commanderPizza("fromage");
System.out.println("Michel a commandé une " + pizza + "\
   n");

pizza = pizzeriaBrest.commanderPizza("fruitsDeMer");
System.out.println("Luc a commandé une  " + pizza + "\n");

pizza = pizzeriaStrasbourg.commanderPizza("fruitsDeMer");
System.out.println("Michel a commandé une " + pizza + "\
   n");
...
```

Etape 3

- Une fabrication plus souple :

```
public abstract class Pizzeria {

   public Pizza commanderPizza(String type) {
     Pizza pizza = creerPizza(type);
     pizza.preparer();
     pizza.cuire();
     pizza.couper();
     pizza.emballer();
     return pizza;
   }

   abstract Pizza creerPizza(String item);

}
```

Etape 3


```
public class PizzeriaBrest extends Pizzeria {

   Pizza creerPizza(String item) {
     Pizza pizza = null;
     if (item.equals("fromage")) {
        pizza = new PizzaFromageStyleBrest();
     } else if (item.equals("vegetarienne")) {
        pizza = new PizzaVegetarienneStyleBrest();
     } else if (item.equals("fruitsDeMer")) {
        pizza = new PizzaFruitsDeMerStyleBrest();
     } else if (item.equals("poivrons")) {
        pizza = new PizzaPoivronsStyleBrest();
     }
     return pizza;
   }
}
```

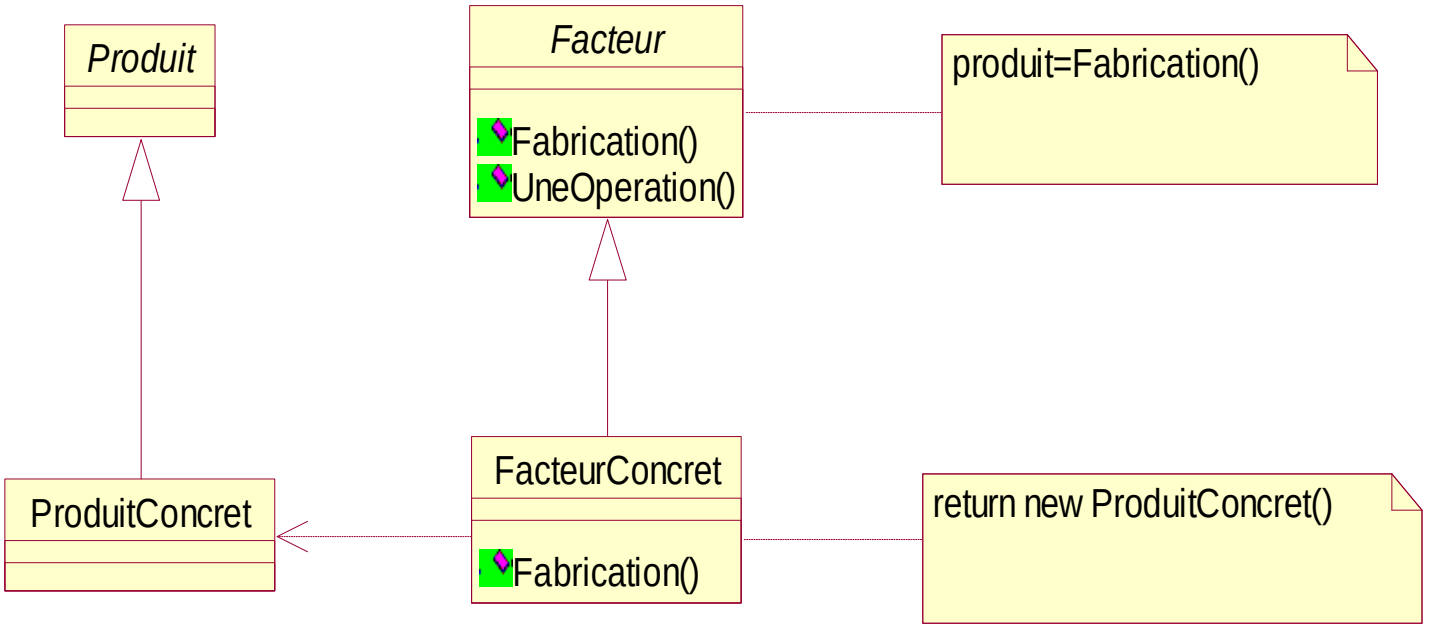
Et les pizzas ???


```
public abstract class Pizza {
   String nom;
   String pate;
   String sauce;
   List garnitures = new ArrayList();

   abstract void preparer();
   void cuire() {
   System.out.println("Cuisson 15 minutes à 180°");
   }
   void couper() {
   System.out.println("Découpage en parts triangulaires");
   }
   void emballer() {
   System.out.println("Emballage dans une boite officielle");
   }
   void setNom(String nom) {
   this.nom = nom;
   }
   String getNom() {
   return nom;
   }
}
```

Et les pizzas ???


```
public class PizzaFromageStyleBrest extends Pizza {

   public PizzaFromageStyleBrest() {
      nom = "Pizza fromage style Brest";
      pate = "Pate fine";
      sauce = "tomate";
      garniture.add("mozzarella");
   }

   void decouper() {
      System.out.println("découpage en carrés");
   }
}
```

Patron Fabrication / Factory Method



Conclusions +

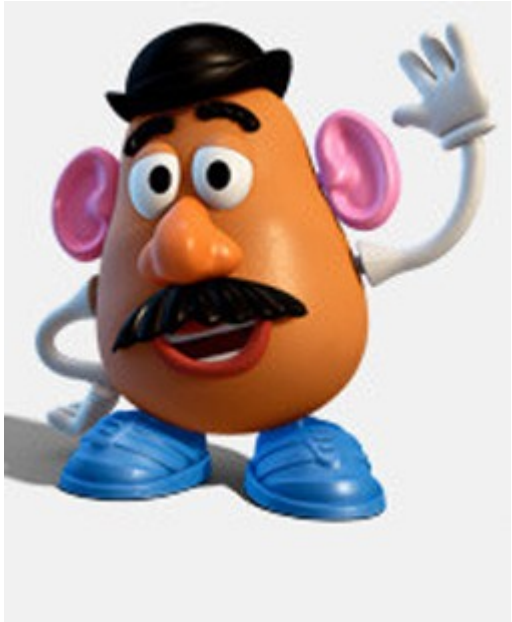
- Principes de conception
  - Encapsulez ce qui varie
  - Programmez des interfaces et non des implémentations
  - Préférez la composition à l'héritage
  - Inversion des dépendances : dépendez d'abstractions et non de classes concrètes
- Patron de conception
  - Stratégie…
  - Fabrique : définit une interface pour la création d'objets en délégant aux sous-classes le choix des classes à instancier

Un souvenir de jeunesse

Où comment le chapitre précédent sur la fabrique vous rappelle avec émotion des scènes de Toy Story ...

Mais oui...

- Monsieur patate !



Euh....

- Messieurs patates ???

Une fabrique ???

La fabrique...

- Les éléments utilisés ne sont pas les mêmes mais appartiennent à des familles

Eg : bras, pieds, nez, oreilles ...

-> une fabrique d’éléments

Fabrique d’ingrédients


```
public interface FabriqueElementPatate {

   public Chapeau      creerChapeau();
   public Nez       creerNez();
   public Yeux      creerYeux();
   public Moustache    creerMoustache();
   public Bouche       creerBouche();
   public Pieds     creerPieds();
   public Oreille[]    creerOreille();
   public Bras[]       creerBras();

}
```

Fabrique d'éléments

```
public class MaFabriquePatate implements
   FabriqueElementPatate {
    public Chapeau creerChapeau() {
       return new ChapeauNoir();
    }
    public Nez creerNez() {
       return new GrosNezOrange();
    }
    …
    public Oreille[] creerOreille() {
       Oreille oreilles [] = { new GrandeOreille(),
                               new GrandeOreille() };
       return oreilles;
    }
    public Bras[] creerBras() {
       …
    }
}
```

Fabrique d’éléments

```
public class MaKeLuigiPatate implements FabriqueEl
   {
    public Chapeau creerChapeau() {
      return new CasquetteLuigi();
    }
    public Nez creerNez() {
      return new GrosNezRouge();
    }
    …
    public Oreille[] creerOreille() {
      Oreille oreilles [] = { new GrandeOreille(),
             new GrandeOreille() };
      return oreilles;
    }
    public Bras[] creerBras() {
      Bras bras [] = { new BrasVert(), new BrasVert() };
      return bras;
    }
}
```

Retour aux Monsieurs…


```
Public class MonsieurPatate {
   FabriqueElementPatate fabLab;
   Chapeau chapeau;
   Nez nez;
   Yeux yeux;
   Moustache moustache;
   Bouche bouche;
   Pieds pieds;
   Oreille[] oreilles;
   Bras[] bras;

   public MonsieurPatate(FabriqueElementPatate fab) {
     fablab = fab;
   }
   void preparer() {
     chapeau = fabLab.creerChapeau();
     nez = fabLab.creerNez();
     ...
   }
   void assembler() {
     System.out.println("assembler les éléments");
   }
   void jouer() {
     System.out.println("Vers l'infini et au-delà !");
   }
   ...
```

Oui mais...

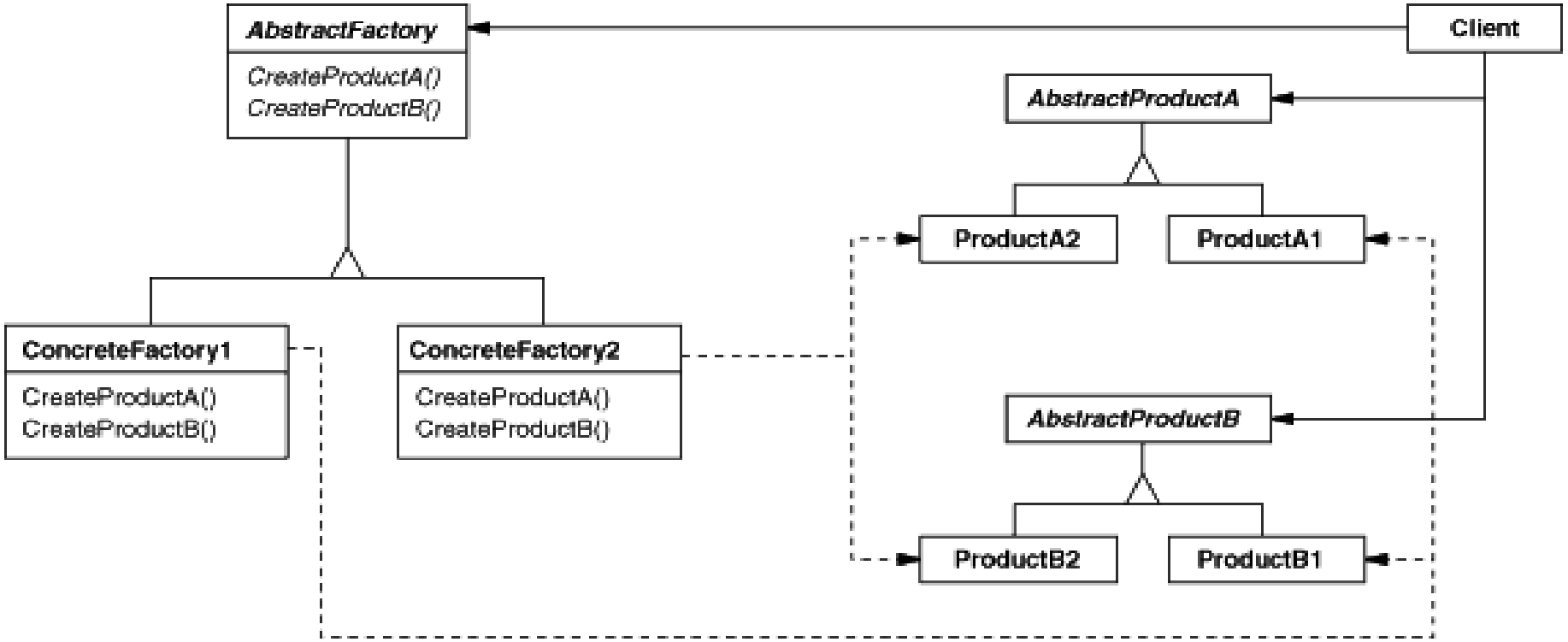
Retour aux Monsieurs...


```
public abstract class MonsieurPatate {
   String nom;
   Chapeau chapeau;
   Nez nez;
   Yeux yeux;
   Moustache moustache;
   Bouche bouche;
   Pieds pieds;
   Oreille[] oreilles;
   Bras[] bras;

   abstract void preparer();

   void assembler() {
     System.out.println("assembler les éléments");
   }
   void jouer() {
     System.out.println("Vers l'infini et au-delà !");
   }
   ...
```

Patron Fabrique Abstraite

Objectif : obtenir des instances de classes implémentant des interfaces connues, mais en ignorant le type réel de la classe obtenue

Exemple : une application gérant des documents polymorphes générateur de composants graphiques supportant une multitude de *look-and-feels*



Fabrique Abstraite d’éléments ?

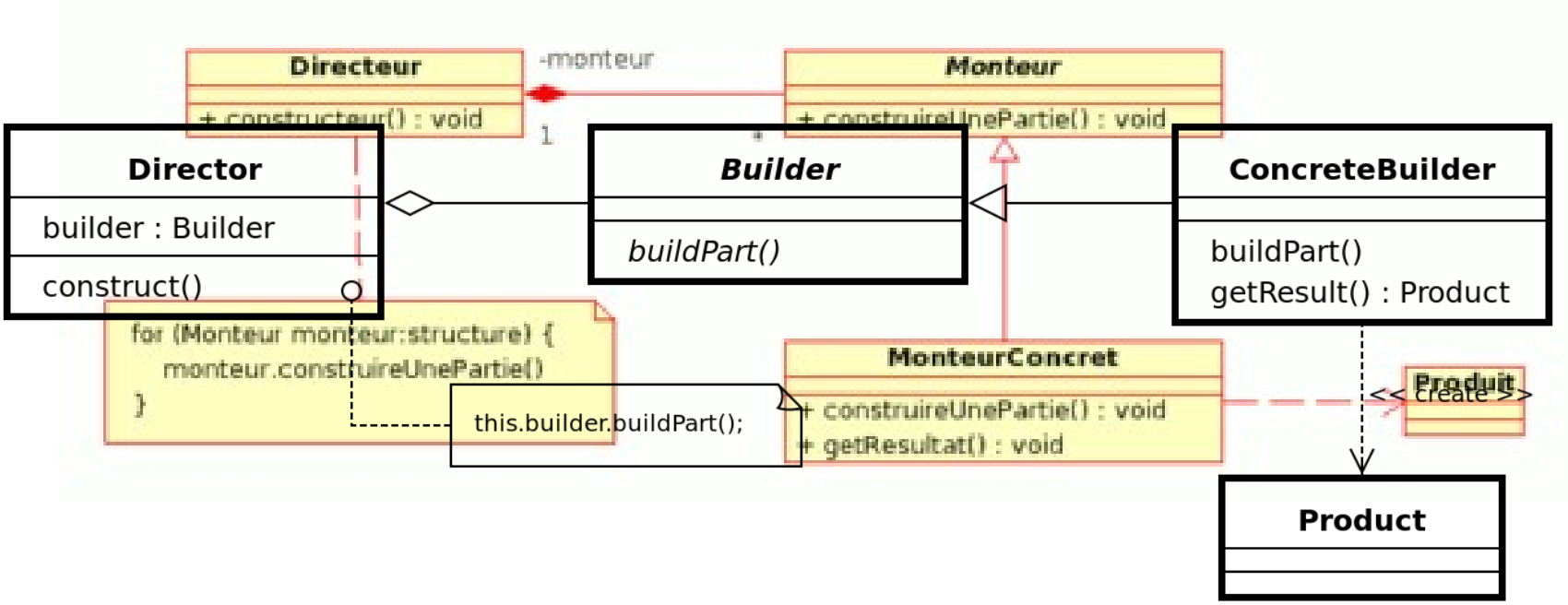
- Dessinez moi un beau diagramme…

Conclusions +

- Principes de conception
  - Encapsulez ce qui varie
  - Programmez des interfaces et non des implémentations
  - Préférez la composition à l'héritage
  - Inversion des dépendances : dépendez d'abstractions et non de classes concrètes
- Patron de conception
  - Stratégie…
  - Fabrique : définit une interface pour la création d'objets en délégant aux sous-classes le choix des classes à instancier
  - Fabrique abstraite : fournit une interface pour créer des familles d'objets apparentés sans avoir à spécifier leurs classes concrètes

Problème....

- Que faire quand un objet est « complexe » à fabriquer ?
- Idée : construction par étapes



Exemple de Monteur/Builder

- Classique en Java : Tab[ ] -> toString ?
- Réponse : StringBuilder

```
@Override
public String toString() {
  StringBuilder s = new StringBuilder();
  s.append("[");
  for(int i: tab) {
     s.append(tab[i]+" ");
  }
  s.append("]");
  return s.toString();
}
```

Conclusions ++

- Principes de conception
  - Encapsulez ce qui varie
  - Programmez des interfaces et non des implémentations
  - Préférez la composition à l'héritage
  - Inversion des dépendances : dépendez d'abstractions et non de classes concrètes
- Patron de conception
  - Stratégie…
  - Fabrique : définit une interface pour la création d'objets en délégant aux sous-classes le choix des classes à instancier
  - Fabrique abstraite : fournit une interface pour créer des familles d'objets apparentés sans avoir à spécifier leurs classes concrètes
  - Monteur : permettre la construction d'un objet complexe en plusieurs étapes indépendamment de sa représentation interne