Design patterns

UNIVERSITÉ DE LORRAINE

Conception Architectures Distribuées

Horatiu CIRSTEA

Factory Method

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

- Comment standardiser un modèle architectural pour un ensemble d'applications mais permettre à chaque application de définir ses propres objets et la manière de les définir?

Commander un menu

Client

orderMenu()

PizzaStore

Solution 1

```
class PizzaStore{
  public void orderMenu(){
    Menu menu = new Menu();
    Pizza pizza = new Pizza();
    Drink drink = new Drink();

    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
  }
}
```

```
class Client{
    public ... main(){
        PizzaStore store = new PizzaStore();
        store.orderMenu();
    }
}
```

Solution 1

```
class PizzaStore{
  public void orderMenu(){
    Menu menu = new Menu();
    Pizza pizza = new Pizza();
    Drink drink = new Drink();

    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
  }
}
```

```
class MasterPizzaStore{
  public void orderMenu(){
    Menu menu = new Menu();
    Pizza pizza = new MasterPizza();
    Drink drink = new MasterDrink();

    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
  }
}
```

```
class Client{
     public ... main(){
          MasterPizzaStore store = new MasterPizzaStore();
          store.orderMenu();
     }
}
```

Solution 1

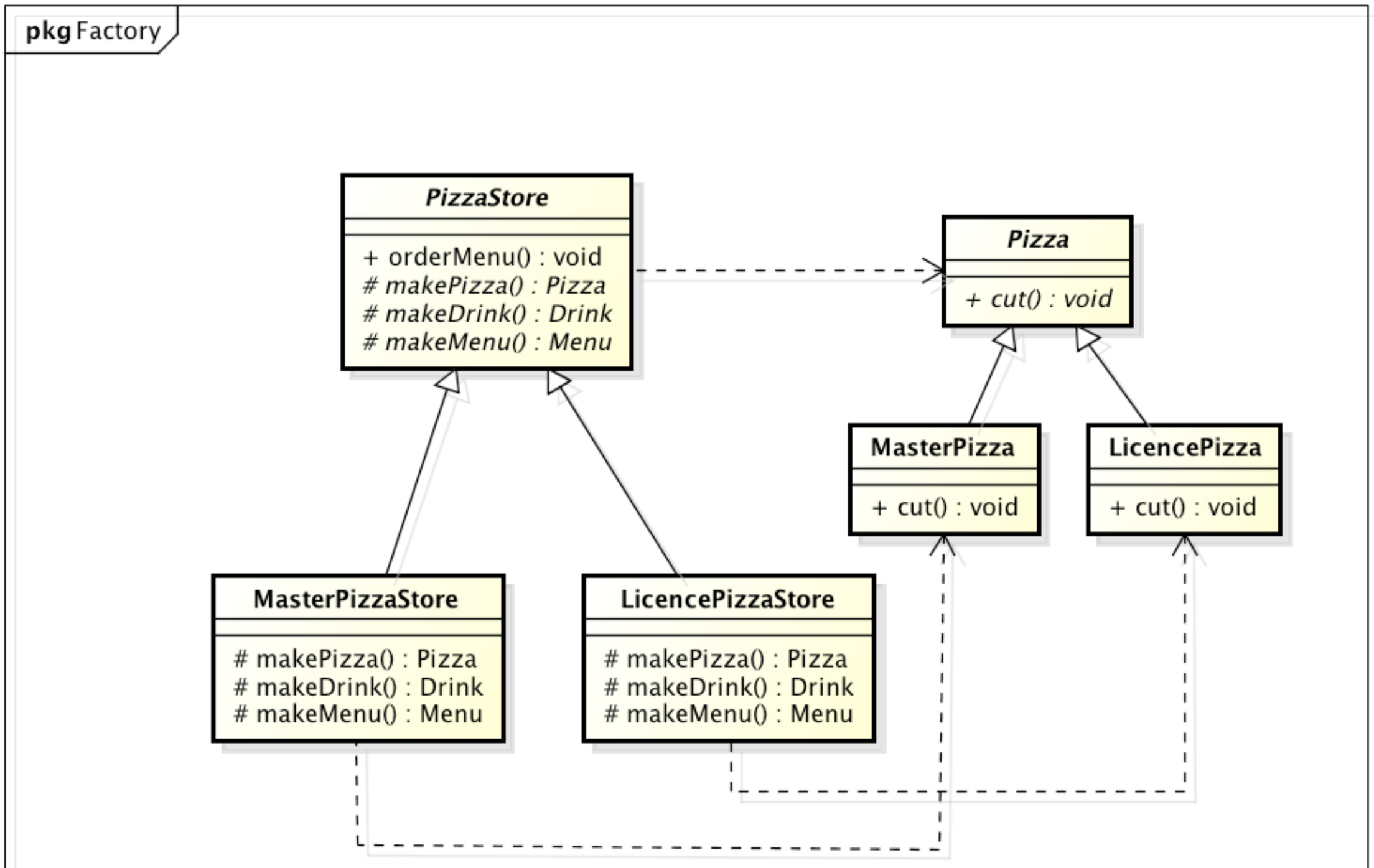
```
class PizzaStore{
  public void orderMenu(){
    Menu menu = new Menu();
    Pizza pizza = new Pizza();
    Drink drink = new Drink();

    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
  }
}
```

```
class MasterPizzaStore{
  public void orderMenu(){
    Menu menu = new Menu();
    Pizza pizza = new MasterPizza();
    Drink drink = new MasterDrink();

    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
  }
}
```

création objet(s)
hard-coded

```
class Client{
     public ... main(){
          MasterPizzaStore store = new MasterPizzaStore();
          store.orderMenu();
     }
}
```

Solution 2


pkg Factory
PizzaStore
+ orderMenu() : void
# makePizza() : Pizza
# makeDrink() : Drink
# makeMenu() : Menu
Pizza
+ cut() : void
MasterPizza
+ cut() : void
LicencePizza
+ cut() : void
MasterPizzaStore
makePizza() : Pizza
makeDrink() : Drink
makeMenu() : Menu
LicencePizzaStore
makePizza() : Pizza
makeDrink() : Drink
makeMenu() : Menu

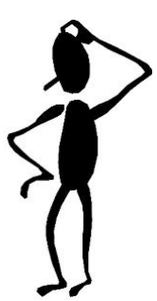
Solution 2

```
class abstract PizzaStore{
  public Menu orderMenu(){
    Menu menu = prepareMenu();
    Pizza pizza = preparePizza();
    Drink drink = prepareDrink();
    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
    return menu;
  }
  public abstract Menu prepareMenu();
  public abstract Pizza preparePizza();
  public abstract Drink prepareDrink();

}
```

Solution 2

```
class abstract PizzaStore{
  public Menu orderMenu(){
    Menu menu = prepareMenu();
    Pizza pizza = preparePizza();
    Drink drink = prepareDrink();
    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
    return menu;
  }
  public abstract Menu prepareMenu();
  public abstract Pizza preparePizza();
  public abstract Drink prepareDrink();

}
```

```
class MasterPizzaStore
                extends PizzaStore{
  public Menu prepareMenu(){
   return new MasterMenu();
  }
  public Pizza preparePizza(){
   return new MasterPizza();
  }
  public Drink prepareDrink(){
    return new MasterDrink();
} }
```



```
class Client{
     public ... main(){
          PizzaStore store = new MasterPizzaStore();
          Menu menu = store.orderMenu();
     }
}
```

Solution 2

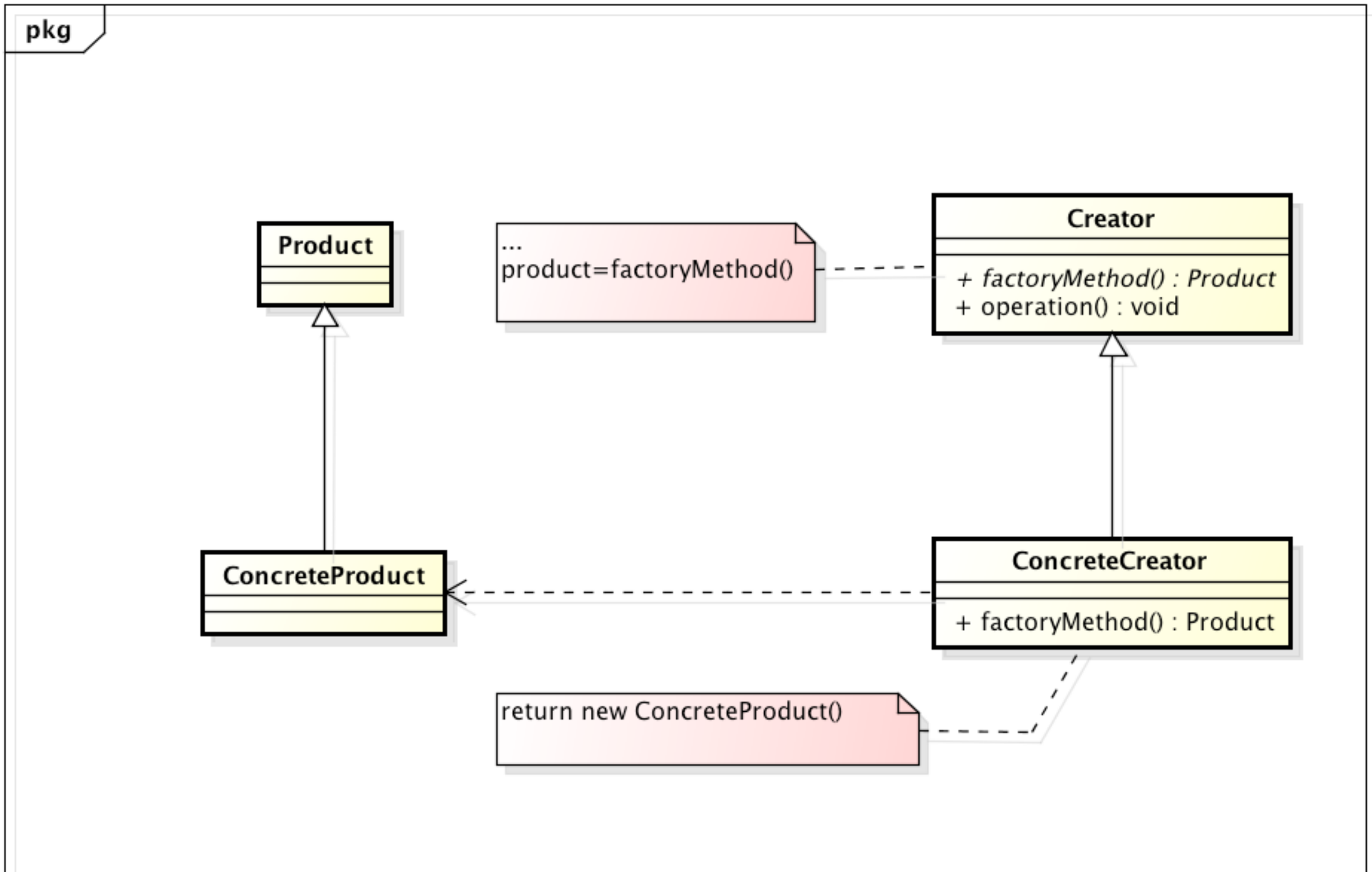
```
class abstract PizzaStore{
  public Menu orderMenu(){
    Menu menu = prepareMenu();
    Pizza pizza = preparePizza();
    Drink drink = prepareDrink();
    pizza.cut()
    menu.addPizza(pizza);
    menu.addDrink(drink);
    menu.box();
    return menu;
  }
  public abstract Menu prepareMenu();
  public abstract Pizza preparePizza();
  public abstract Drink prepareDrink();

}
```

```
class MasterPizzaStore
                extends PizzaStore{
  public Menu prepareMenu(){
   return new MasterMenu();
  }
  public Pizza preparePizza(){
   return new MasterPizza();
  }
  public Drink prepareDrink(){
   return new MasterDrink();
} }
```

factory methods

```
class Client{
     public ... main(){
          PizzaStore store = new MasterPizzaStore();
          Menu menu = store.orderMenu();
     }
}
```

Factory method



Factory method

Intention

- définir une interface de création d'un objet en déléguant aux sous-classes concrètes le choix de la classe à instancier
- définir un constructeur virtuel

Constructeur virtuel

```
class PizzaStore{
   public Menu orderMenu(){
      Menu menu = prepareMenu();
      Pizza pizza = preparePizza();
      Drink drink = prepareDrink();

      pizza.cut()
      menu.addPizza(pizza);
      menu.addDrink(drink);
      menu.box();

      return menu;
   }
}
```

Constructeur virtuel

```
class PizzaStore{
   public Menu orderMenu(){
      Menu menu = prepareMenu();
      Pizza pizza = preparePizza();
      Drink drink = prepareDrink();

      pizza.cut()
      menu.addPizza(pizza);
      menu.addDrink(drink);
      menu.box();

      return menu;
   }
}
```

```
            class Client{
                public ... main(){
                   PizzaStore store = new MasterPizzaStore();
                   Menu menu = store.orderMenu();
                   menu.eatPizza();
                }
            }
```

Factory method

Indications d'utilisation

- quand une classe ne peut pas anticiper les classes des objets qu'elle doit créer
- **une classe veut transmettre à ses sous-classes les choix d'instanciation des objets qu'elle doit créer/ utiliser** → le choix peut être fait en fonction d'un paramètre

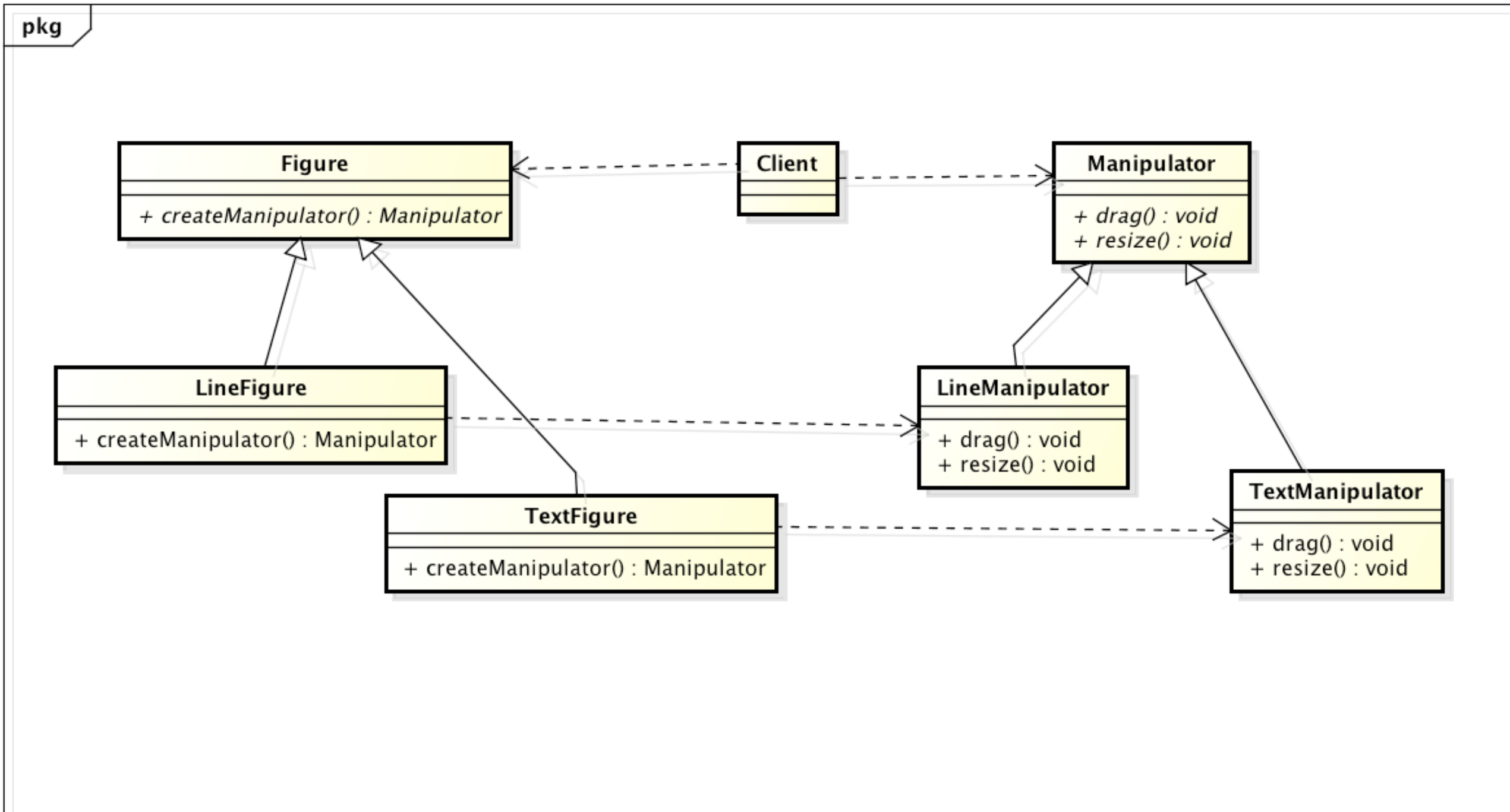
```
class PizzaStore{
  public void orderMenu(String type){
    ...
    Pizza pizza = preparePizza(type);
    ...
  }
  ...
  public abstract Pizza preparePizza(String type);
  ...
}
```

```
class PizzaStore{
  public void orderMenu(String type){
    ...
    Pizza pizza = preparePizza(type);
    ...
  }
  ...
  public abstract Pizza preparePizza(String type);
  ...
}

class MasterPizzaStore extends PizzaStore{
  ...
  public Pizza preparePizza(String type){
    if(type.equals(“Pepperonni”)) {
       return new PepperonniMasterPizza();
    } else if(type.equals(“Cheese”)) {
       return new CheeseMasterPizza();
    } else if(type.equals(“Vegetarian”)) {
       return new VegetarianMasterPizza();
    }
    return new TodayMasterPizza();
  }
  ...
}
```

```
class PizzaStore{
  public void orderMenu(String type){
    ...
    Pizza pizza = preparePizza(type);
    ...
  }
  ...
  public abstract Pizza preparePizza(String type);
  ...
}

class MasterPizzaStore extends PizzaStore{
  ...
  public Pizza preparePizza(String type){
    if(type.equals("Pepperonni")) {
       return new PepperonniMasterPizza();
    } else if(type.equals("Cheese")) {
       return new CheeseMasterPizza();
    } else if(type.equals("Vegetarian")) {
       return new VegetarianMasterPizza();
    }
    return new TodayMasterPizza();
  }
  ...
}
```

factory methods paramétrées

Hiérarchies parallèles



Résumé Factory method

- on peut considérer un pool d'objets et les réutiliser au lieu de les créer (chaque fois)
- variétés
  - classe `Creator` abstraite sans implantation **de la** factory method ➙ **sous-classes imprévisibles**
  - classe `Creator` concrète avec implantation **de la** factory method ➙ **flexibilité**

Abstract Factory

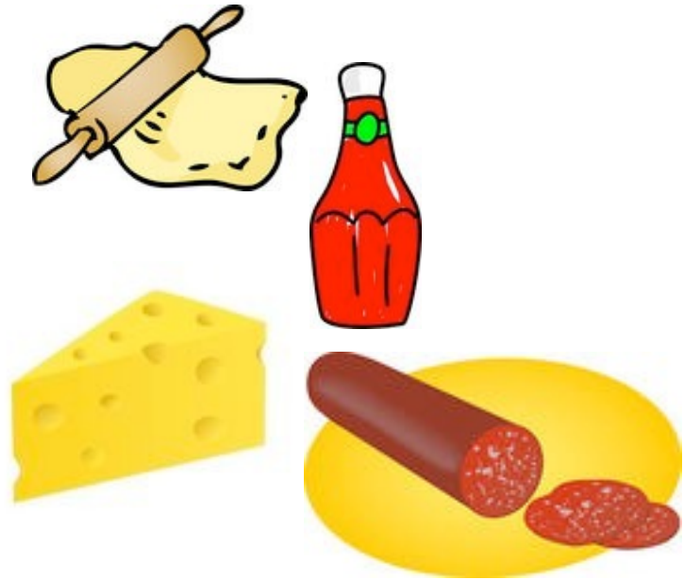
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

- Spécifier les dépendances des plateformes et faciliter leur remplacement pour garantir la portabilité de l’application

Préparer les pizzas

cook()



PizzaStore

Pizza

Solution 1

```
abstract class Pizza{
  Dough dough;
  Sauce sauce;
  Cheese cheese;
  Meat meat;
  ...
  abstract void cook();
  ...
}
```

Solution 1

```
abstract class Pizza{
  Dough dough;
  Sauce sauce;
  Cheese cheese;
  Meat meat;
  ...
  abstract void cook();
  ...
}
```

```
class PepperonniMasterPizza
      extends Pizza{
    public void cook(){
      dough=new MasterDough();
      sauce=new MasterSauce();
      cheese=new MasterCheese();
      meat=new MasterPepperonni();
    }
}

class PepperonniLicencePizza
      extends Pizza{
    public void cook(){
      dough=new LicenceDough();
      sauce=new LicenceSauce();
      cheese=new LicenceCheese();
      meat=new LicencePepperonni();
    }
}
```

Solution I

```
abstract class Pizza{
  Dough dough;
  Sauce sauce;
  Cheese cheese;
  Meat meat;
  ...
  abstract void cook();
  ...
}
```

code dupliqué

```
class PepperonniMasterPizza
      extends Pizza{
    public void cook(){
      dough=new MasterDough();
      sauce=new MasterSauce();
      cheese=new MasterCheese();
      meat=new MasterPepperonni();
    }
}

class PepperonniLicencePizza
      extends Pizza{
    public void cook(){
      dough=new LicenceDough();
      sauce=new LicenceSauce();
      cheese=new LicenceCheese();
      meat=new LicencePepperonni();
    }
}
```

Solution I

```
abstract class Pizza{
  Dough dough;
  Sauce sauce;
  Cheese cheese;
  Meat meat;
  ...
  abstract void cook();
  ...
}
```

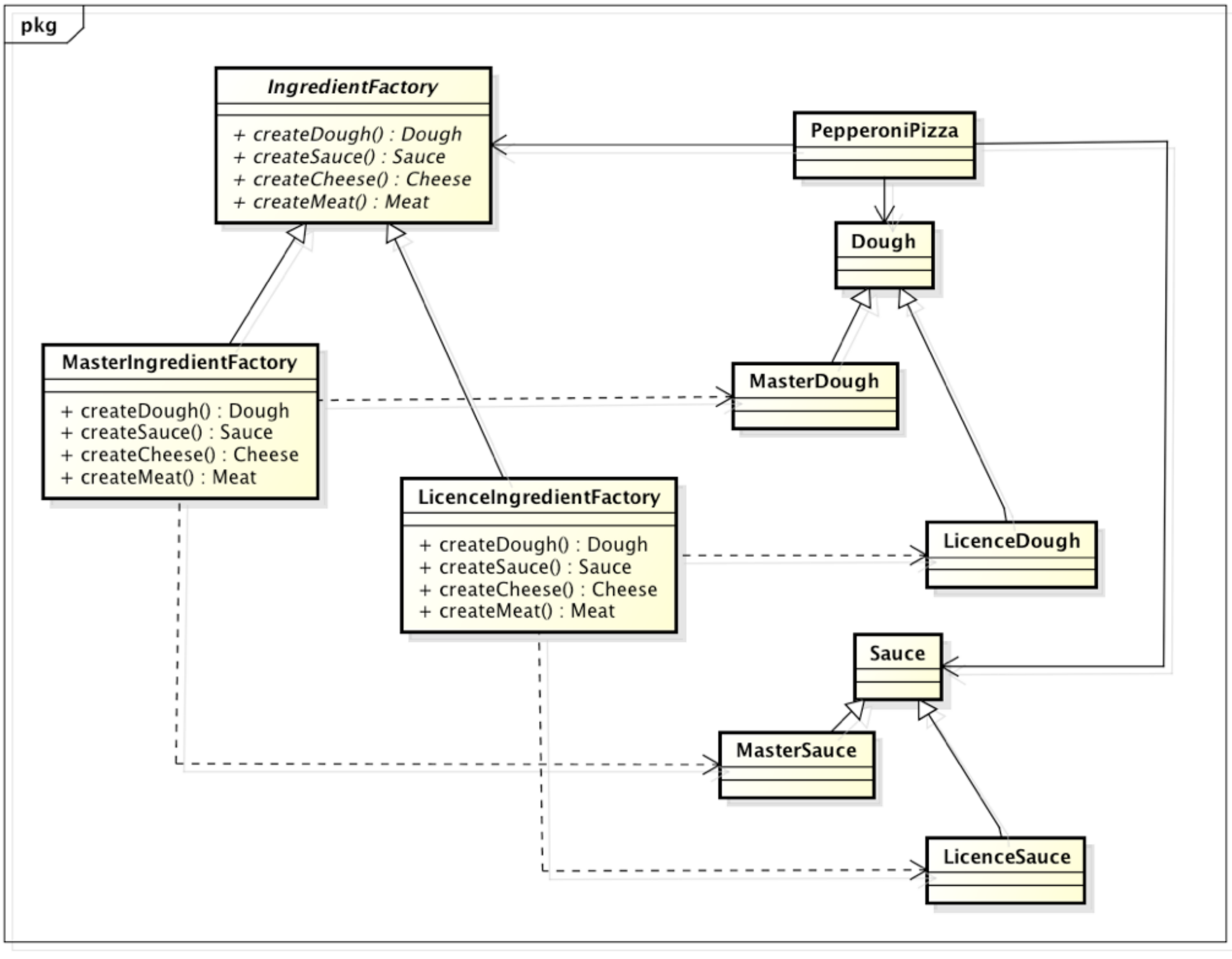
```
class PepperonniMasterPizza
      extends Pizza{
    public void cook(){
      dough=new MasterDough();
      sauce=new MasterSauce();
      cheese=new MasterCheese();
      meat=new MasterPepperonni();
    }
}
```

```
class PepperonniLicencePizza
      extends Pizza{
    public void cook(){
      dough=new LicenceDough();
      sauce=new LicenceSauce();
      cheese=new LicenceCheese();
      meat=new LicencePepperonni();
    }
}
```

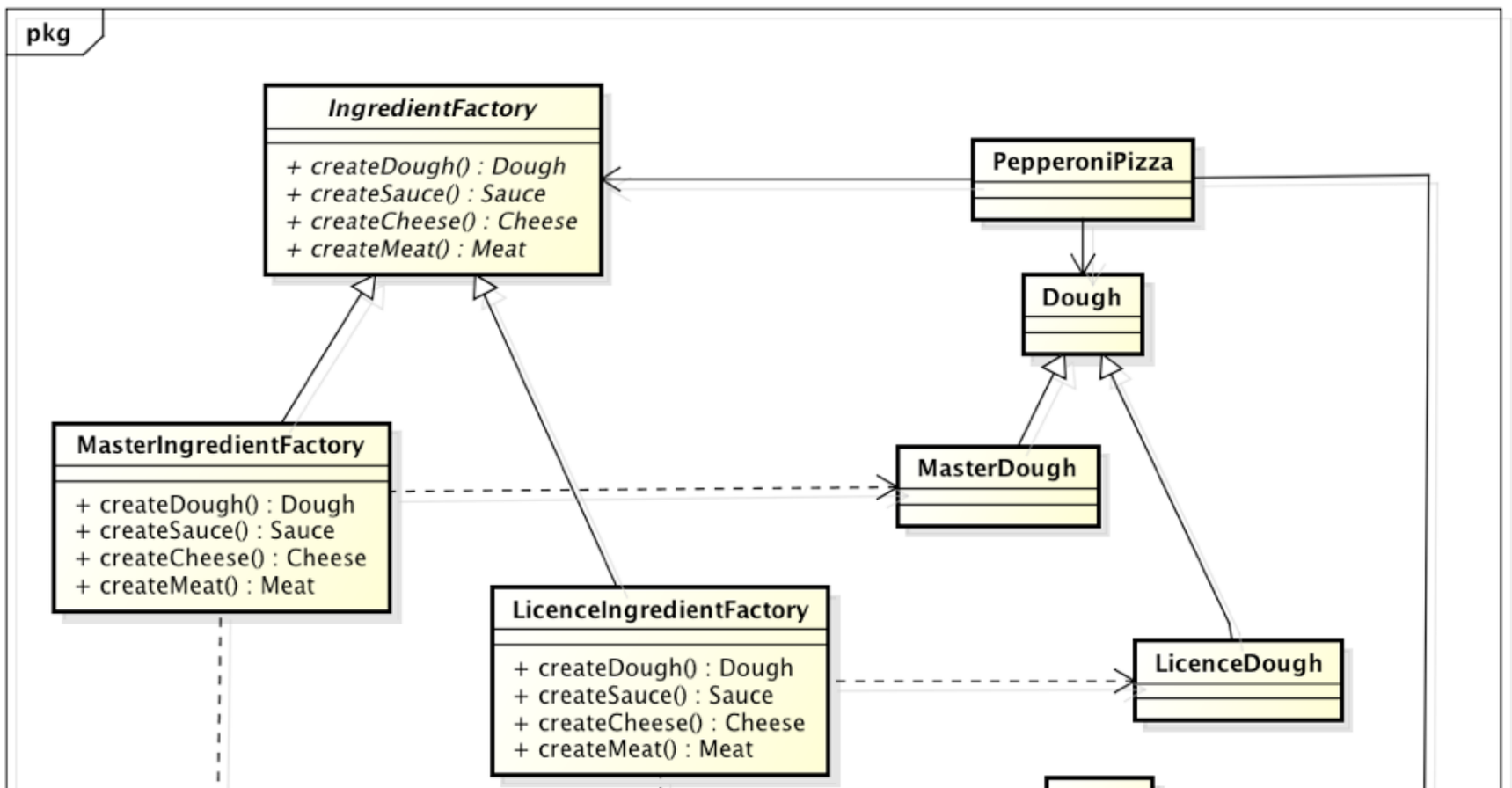
code dupliqué

les ingrédients sont liés

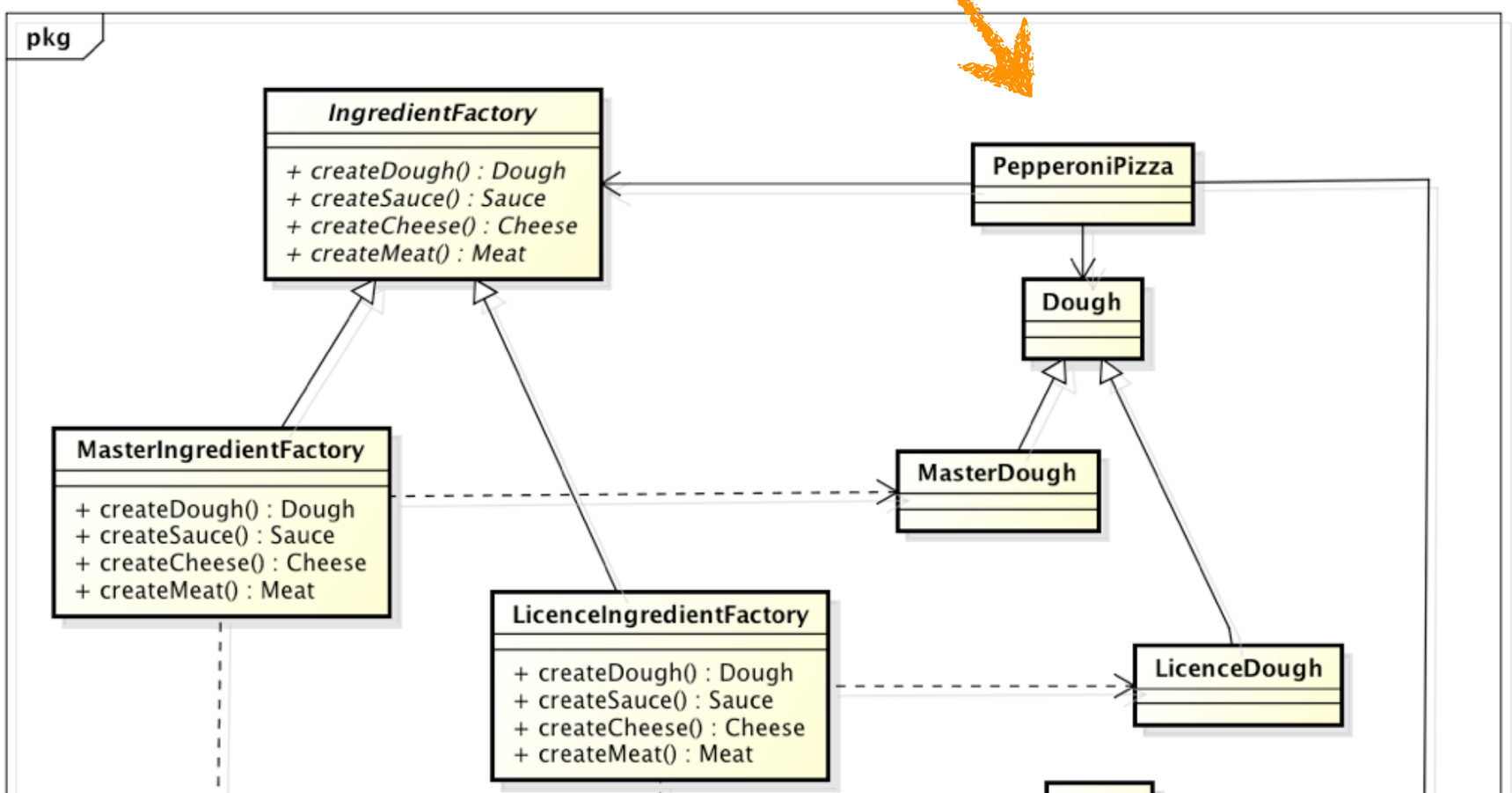
Solution 2


pkg
IngredientFactory
+ createDough() : Dough
+ createSauce() : Sauce
+ createCheese() : Cheese
+ createMeat() : Meat
PepperoniPizza
Dough
MasterIngredientFactory
+ createDough() : Dough
+ createSauce() : Sauce
+ createCheese() : Cheese
+ createMeat() : Meat
MasterDough
LicenceIngredientFactory
+ createDough() : Dough
+ createSauce() : Sauce
+ createCheese() : Cheese
+ createMeat() : Meat
LicenceDough
Sauce
MasterSauce
LicenceSauce

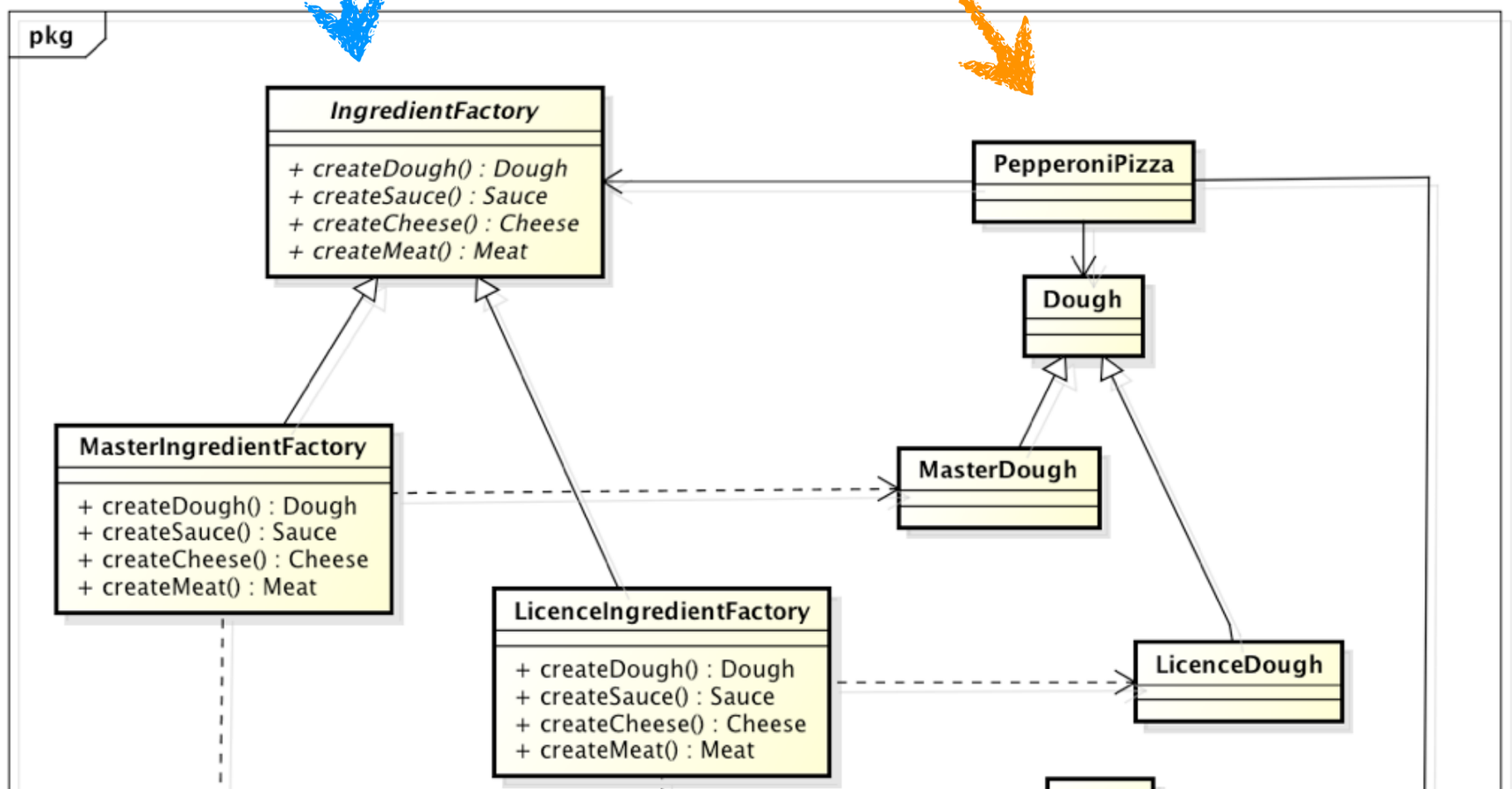
```
class PepperonniPizza extends Pizza{
  IngredientFactory ingredientFactory;

  public PepperonniPizza(IngredientFactory factory){
    ingredientFactory = factory;
  }
  public void cook(){
      dough = ingredientFactory.createDough();
      sauce = ingredientFactory.createSauce();
      cheese = ingredientFactory.createCheese();
      meat = ingredientFactory.createPepperonni();
  }
}
```



```
class PepperonniPizza extends Pizza{
  IngredientFactory ingredientFactory;

  public PepperonniPizza(IngredientFactory factory){
    ingredientFactory = factory;
  }
  public void cook(){
      dough = ingredientFactory.createDough();
      sauce = ingredientFactory.createSauce();
      cheese = ingredientFactory.createCheese();
      meat = ingredientFactory.createPepperonni();
  }
}
```



```
class PepperonniPizza extends Pizza{
  IngredientFactory ingredientFactory;

  public PepperonniPizza(IngredientFactory factory){
    ingredientFactory = factory;
  }
  public void cook(){
      dough = ingredientFactory.createDough();
      sauce = ingredientFactory.createSauce();
      cheese = ingredientFactory.createCheese();
      meat = ingredientFactory.createPepperonni();
  }
}
```



PizzaStore

```
class MasterPizzaStore extends PizzaStore{
   ...
   public Pizza preparePizza(String type){
     if(type.equals(“Pepperonni”)) {
        return new PepperonniMasterPizza();
     } else if(type.equals(“Cheese”)) {
        return new CheeseMasterPizza();
     } else if(type.equals(“Vegetarian”)) {
        return new VegetarianMasterPizza();
     }
     return new TodayMasterPizza();
   }
   ...
}
```

PizzaStore

```
class MasterPizzaStore extends PizzaStore{
  ...
  public Pizza preparePizza(String type){
    if(type.equals(“Pepperonni”)) {
       return new PepperonniMasterPizza();
    } else if(type.equals(“Cheese”)) {
       return new CheeseMasterPizza();
    } else if(type.equals(“Vegetarian”)) {
       return new VegetarianMasterPizza();
    }
    return new TodayMasterPizza();
  }
```

```
class MasterPizzaStore extends PizzaStore{
  IngredientFactory masterFactory =
                  new MasterIngredientFactory();

  public Pizza preparePizza(String type){
    Pizza pizza = new TodayPizza(masterFactory);
    if(type.equals(“Pepperonni”)) {
       pizza = new PepperonniPizza(masterFactory);
    } else if(type.equals(“Cheese”)) {
       pizza = new CheesePizza(masterFactory);
    } else if(type.equals(“Vegetarian”)) {
       pizza = new VegetarianPizza(masterFactory);
    }
    pizza.cook();
    return pizza;
  }
  ...
}
```

PizzaStore

```
class MasterPizzaStore extends PizzaStore{
  ...
  public Pizza preparePizza(String type){
    if(type.equals(“Pepperonni”)) {
       return new PepperonniMasterPizza();
    } else if(type.equals(“Cheese”)) {
       return new CheeseMasterPizza();
    } else if(type.equals(“Vegetarian”)) {
       return new VegetarianMasterPizza();
    }
    return new TodayMasterPizza();
  }
```

```
class MasterPizzaStore extends PizzaStore{
  IngredientFactory masterFactory =
                  new MasterIngredientFactory();

  public Pizza preparePizza(String type){
    Pizza pizza = new TodayPizza(masterFactory);
    if(type.equals(“Pepperonni”)) {
       pizza = new PepperonniPizza(masterFactory);
    } else if(type.equals(“Cheese”)) {
       pizza = new CheesePizza(masterFactory);
    } else if(type.equals(“Vegetarian”)) {
       pizza = new VegetarianPizza(masterFactory);
    }
    pizza.cook();
    return pizza;
  }
  ...
}
```

```
class MasterIngredientFactory
        extends IngredientFactory{
  public Dough createDough(){
    return new MasterDough();
  }...
}
```

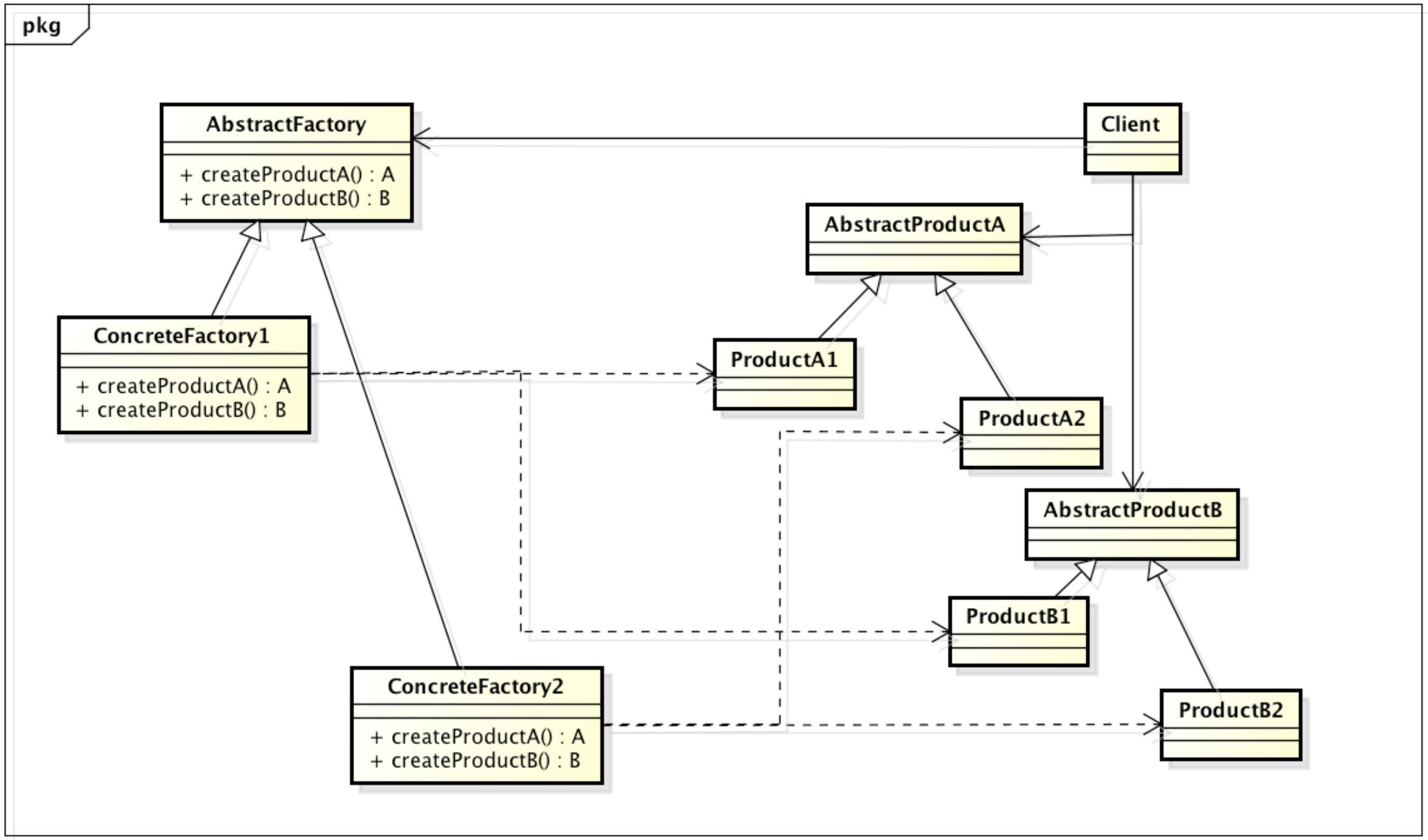
PizzaStore

```
class PizzaStore {
  IngredientFactory factory;
  public PizzaStore(IngredientFactory factory){
       this.factory = factory;
  }
  public Pizza preparePizza(String type){
    Pizza pizza = new TodayPizza(factory);
    if(type.equals(“Pepperonni”)) {
       pizza = new PepperonniPizza(factory);
    }   ...
}
```

PizzaStore

```
class PizzaStore {
  IngredientFactory factory;
  public PizzaStore(IngredientFactory factory){
       this.factory = factory;
  }
  public Pizza preparePizza(String type){
    Pizza pizza = new TodayPizza(factory);
    if(type.equals(“Pepperonni”)) {
       pizza = new PepperonniPizza(factory);
    }   ...
}

class Client{
     public ... main(){
          PizzaStore store =
                new PizzaStore(new MasterIngredientFactory());
          Menu menu = store.orderMenu();
    }
}
```

Abstract Factory



Abstract Factory

Intention

- fournir une interface pour la création de familles d'objets dépendants ou associés sans connaître les classes concrètes destinées à la création de ces objets
- fabriquer des fabriques

Abstract Factory

Indications d'utilisation

- un système doit être indépendant de la façon dont ses produits sont créés, composés et représentés
- une classe ne peut pas anticiper les classes des objets qu'elle doit créer
- un système doit utiliser juste une des familles d'un ensemble de familles de produits
- garantir que les produits d'une famille sont utilisés ensemble

Abstract Factory

Conséquences

- isolation des classes concrètes (seules les classes abstraites sont connues)
- échange facile des familles de produit
- encouragement de la cohérence entre les produits
- prise en compte difficile de nouvelles formes de produit

Abstract Factory

Implémentation

- souvent une seule instance d'une factory **concrète**
  ➞ **les** factories **sont souvent des singletons**
- **ajouter de nouveaux produits à l'interface ➞** une seule méthode `create(parameter)`
- **création des produits ➞** factory method

Résumé Factory

- **`Abstract Factory`** → une classe délègue la responsabilité de l'instanciation des objets à un autre objet (via la composition)
- **`Factory Method`** → les sous-classe réalisent l'instanciation des objets (via l'héritage)
  - **souvent utilisé par l'objet délégué dans `Abstract Factory` pour l'instanciation**

Questions ?