CPOA - Support TD 3

Dut/Info-S3/M3105 - (Semaine 47)

Version corrigée 

Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. Je sais programmer en Java. 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise les concepts objet de base (héritage, polymorphisme, …). 4. J'ai compris ce qu'est un patron et j'ai grand soif d'en apprendre d'autres que *Strategy* et Singleton
ObjTD	Aborder le patron **fabrique**.
Durée	**1** TD et **2** TP de 1,5h (sur 2 semaines).

1. La pizzeria O'Reilly

Vous êtes embauché dans une pizzeria pour faire … de l'informatique (il y en a bien qui font leur PTUT pour une boulangerie…)!

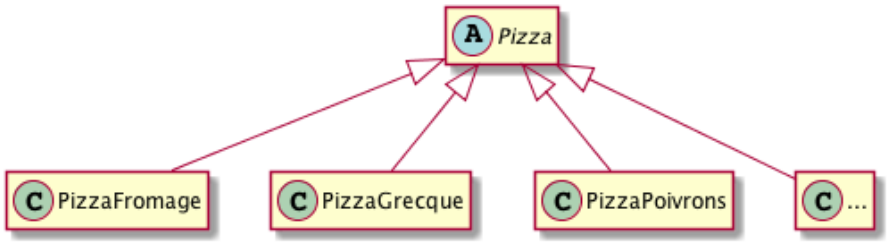
Le stagiaire de l'an dernier qui avait travaillé sur le code est parti avec la caisse (de Chianti). Vous n'avez à votre disposition que :

1. Le code de départ suivant :

Code de `Pizzeria.java`

```
/**
 * @author bruel (from O'Reilly Head-First series)
 */
public class Pizzeria {

    public Pizza commanderPizza(String type) {

        Pizza pizza;

        if (type.equals("fromage")) {
            pizza = new PizzaFromage();
        } else if (type.equals("grecque")) {
            pizza = new PizzaGrecque();
        } else {
            pizza = new PizzaPoivrons();
        }

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

        return pizza;
        }
}
```

2. L'ébauche de diagramme de classe des pizzas suivant :



3. Le bout de code de test suivant :

```
Pizzeria boutiqueBrest = new Pizzeria();
boutiqueBrest.commanderPizza("fromage");
...
Pizzeria boutiqueStrasbourg = new Pizzeria();
boutiqueStrasbourg.commanderPizza("grecque");
```

QUESTION

1. Identifiez ce qui varie dans ce code (si la pression du marché fait ajouter des pizzas à la carte ou si une pizza n'a plus de succès et doit disparaître, etc.).
2. Isolez dans une classe `SimpleFabriqueDePizzas` ce code.
3. Réalisez le diagramme de classe obtenu.
4. Quel est l'avantage de procéder ainsi ? Ne transfère-t'on pas simplement le problème à un autre objet ?

Bien sûr vous héritez de cet horrible "if then else" et dans votre implémentation en TP vous remplacerez ce code avantageusement par un "switch case" et utiliserez une `enum` comme vu en cours.

Solution 

1. Le `if` à remplacer par un `pizza = fabrique.creerPizza(type);` et on ajoute le code suivant dans la classe :

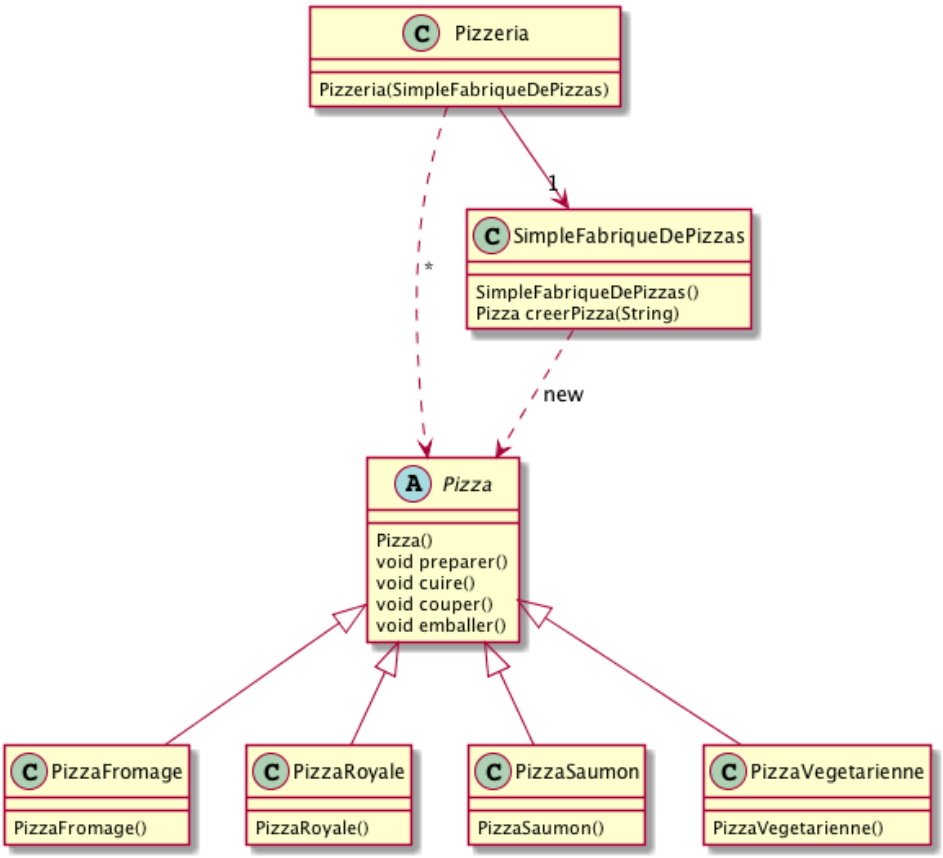
```
    SimpleFabriqueDePizzas fabrique;

    public Pizzeria(SimpleFabriqueDePizzas fabrique) {
        this.fabrique = fabrique;
    }
```

Faire remarquer que la pizza n'est plus la propriété de la pizzeria, mais de la fabrique!

2. Code de `SimpleFabriqueDePizzas`

```java
/**
 * @author bruel
 * @depend - new - Pizza
 */
public class SimpleFabriqueDePizzas {

    public Pizza creerPizza(String type) {
        Pizza pizza;

        if (type.equals("fromage")) {
            pizza = new PizzaFromage();
        } else if (type.equals("saumon")) {
            pizza = new PizzaSaumon();
        } else if (type.equals("royale")) {
            pizza = new PizzaRoyale();
        } else {
            pizza = new PizzaVegetarienne();
        }

        return pizza;
    }

}
```

3. Diagramme de classe :



4. En encapsulant la création des pizzas dans une seule classe, nous n'avons plus

qu'un seul endroit auquel apporter des modifications quand l'implémentation change.

2. On y est presque...

Nous sommes arrivés à une situation propre, qui s'apparente à un patron de conception. Mais avant d'en arriver à la définition du patron lui-même, nous allons améliorer un peu les choses.

2.1. Succès des pizzerias O'Reilly : les franchises

Plusieurs villes veulent ouvrir des pizzerias comme la vôtre. Votre patron, très content de vos programmes souhaite imposer à toutes les futures pizzerias d'utiliser vos codes.

Le problème : les pizzas au fromage de Strasbourg sont différentes des pizzas aux fromages de Corse!

QUESTION

Proposez une solution où `SimpleFabriqueDePizzas` serait une classe abstraite.

Solution 

Simplement ajouter abstract, créer plusieurs sous-classes et avoir une utilisation du style :

```
SimpleFabriqueDePizzas fabriqueBrest = new FabriqueDePizzasBrest();
Pizzeria boutiqueBrest = new Pizzeria(fabriqueBrest);
boutiqueBrest.commander("Végétarienne");
...
SimpleFabriqueDePizzas fabriqueStrasbourg = new
FabriqueDePizzasStrasbourg();
Pizzeria boutiqueStrasbourg = new Pizzeria(fabriqueStrasbourg);
boutiqueStrasbourg.commander("Végétarienne");
```

2.2. La dérive : chacun travaille comme il l'entend!

Les pizzerias utilisent bien vos fabriques mais ont changé leurs procédures : certaines ne coupent pas les pizzas, changent les temps de cuissons, et les pizzerias O'Reilly perdent leur identité. Il nous faut donc **restructurer** les pizzerias.

Un consultant italien payé fort cher (heureusement en pizzas!) propose de revenir à la structure suivante :

```
public abstract class Pizzeria {
    public final Pizza commanderPizza(String type) {
        Pizza pizza;

        pizza = creerPizza(type);
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();

        return pizza;
    }

    ..... Pizza creerPizza(String type);
}
```

QUESTION

Quelles sont les différences avec notre conception actuelle?

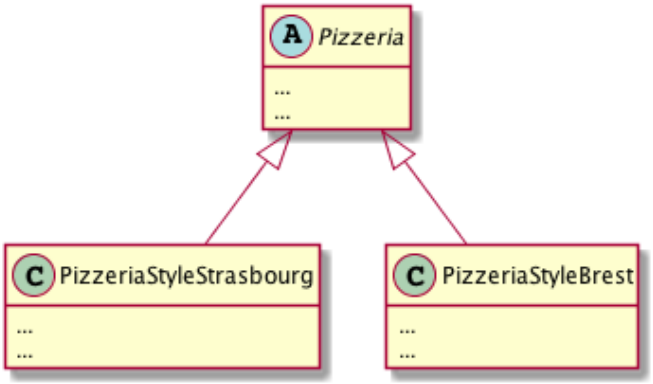
Solution 

- Pizzeria est maintenant abstraite (vous allez voir pourquoi ci-dessous).
- Maintenant, `creerPizza()` est de nouveau un appel à une méthode de Pizzeria et non à un objet fabrique.
- Notre "méthode de fabrication" est maintenant abstraite dans `Pizzeria`.
- Et nous avons transféré la fonctionnalité de notre objet fabrique à cette méthode.

2.3. Laisser les sous-classes décider

QUESTION

Dans le schéma suivant, placez les méthodes au bon endroit de façon à ce que les procédures soient respectées tout en ayant des pizzas à variantes "régionales".



Solution 

Chaque sous-classe redéfinit la méthode `creerPizza()`, tandis que toutes les sous-classes utilisent la méthode `commanderPizza()` définie dans `Pizzeria`.

Voici un exemple de Pizzeria concrète :

```
public class PizzeriaBrest extends Pizzeria {
    Pizza creerPizza(String item) {
    if (choix.equals("fromage")) {
        return new PizzaFromageStyleBrest();
        } else if (choix.equals("vegetarienne")) {
        return new PizzaVegetarienneStyleBrest();
        } else if (choix.equals("fruitsDeMer")) {
        return new PizzaFruitsDeMerStyleBrest();
        } else if (choix.equals("poivrons")) {
        return new PizzaPoivronsStyleBrest();
        } else
        return null;
    }
}
```

2.4. Déclarer une méthode de fabrique

Rien qu'en apportant une ou deux transformations à `Pizzeria`, nous sommes passés d'un objet gérant l'instanciation de nos classes concrètes à un ensemble de sous-classes qui assument maintenant cette responsabilité.

QUESTION

Quelle est la déclaration exacte de la méthode `creerPizza()` de la classe `Pizzeria` ?

Solution

```
protected abstract Pizza creerPizza(String type);
```

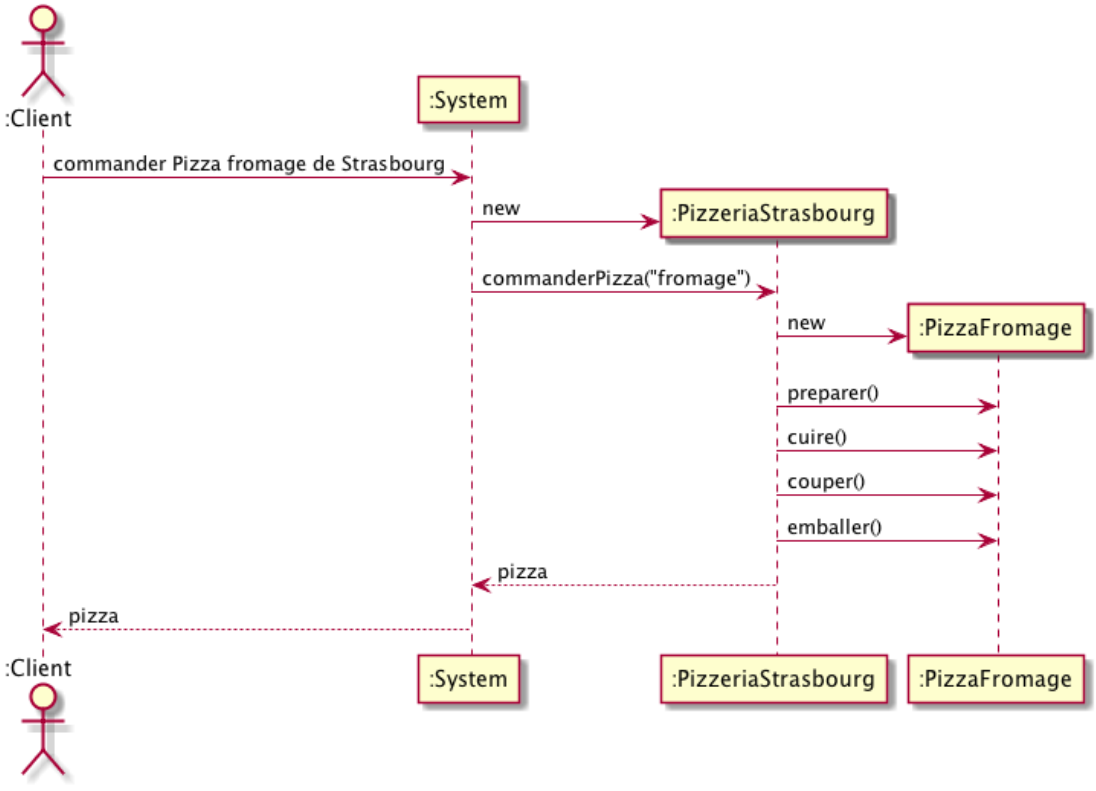
- `protected abstract` : Comme une méthode de fabrication est abstraite, on compte sur les sous-classes pour gérer la création des objets.
- `Pizza` : Une méthode de fabrication retourne un Produit qu'on utilise généralement dans les méthodes définies dans la superclasse.
- `creerPizza`: Une méthode de fabrique isole le client (le code de la superclasse, tel `commanderPizza()` : elle lui évite de devoir connaître la sorte de Produit concret qui est réellement créée.

2.5. Récapitulons

QUESTION

Donnez le diagramme de séquence d'une "commande de pizza au fromage de type Strasbourg".

Solution



Vous implémenterez les classes manquantes en TP.

3. Le patron Fabrique (simple)

Nous y sommes, vous venez de décortiquer le patron Fabrique Simple

Design pattern : **Fabrique (simple)**

Fabrique (simple) définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier (voir aussi Fabrique abstraite).

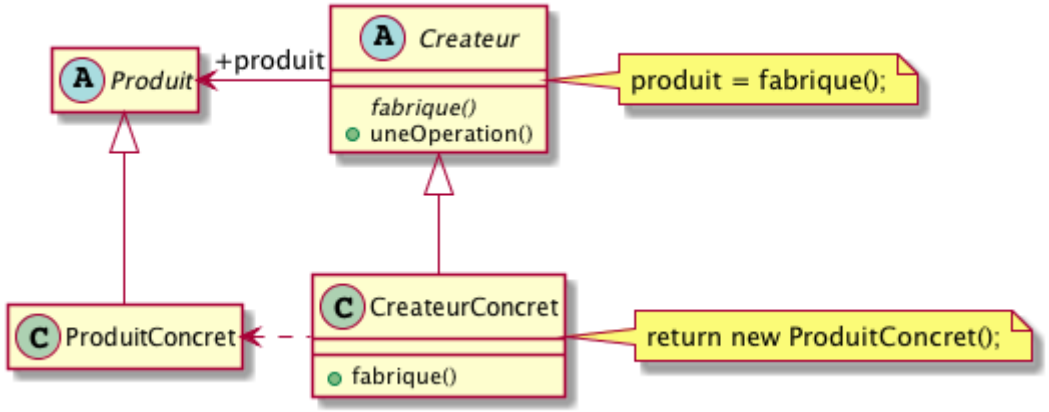


Figure 1. Modèle UML du patron Fabrique

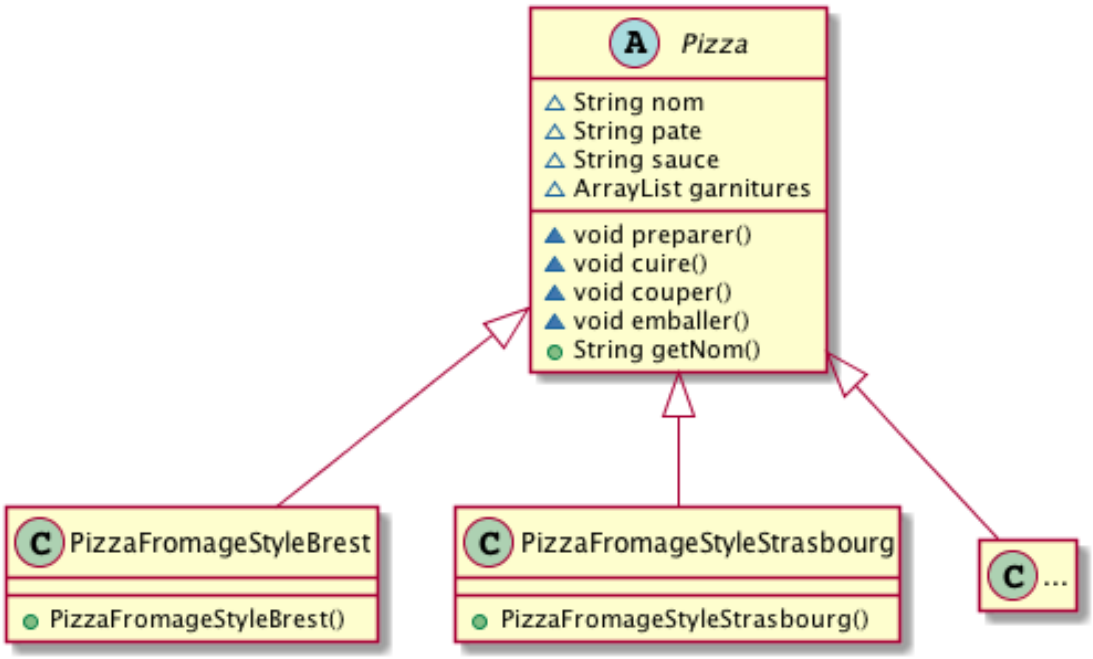
Pour aller plus loin

Et les pizzas dans tout ça !?

QUESTION

Proposez un diagramme de classe UML pour les pizzas (classes, attributs et méthodes).

Solution 



Exemple de code pour `Pizza` :

```
public abstract class Pizza {
    private String nom;
    private String pate;
    private String sauce;
    private ArrayList <Ingredient> garnitures = new ArrayList<>();

    public void preparer() {
        System.out.println("Préparation de " + nom);
        System.out.println("Étalage de la pâte...");
        System.out.println("Ajout de la sauce...");
        System.out.println("Ajout des garnitures: ");
        for (int i = 0; i < garnitures.size(); i++) {
            System.out.println(" " + garnitures.get(i));
        }
    }
    public void cuire() {
        System.out.println("Cuisson 25 minutes à 180°");
    }
    public void couper() {
        System.out.println("Découpage en parts triangulaires");
    }
    public void emballer() {
        System.out.println("Emballage dans une boîte officielle");
              }
    public String getNom() {
        return nom;
    }
}
```

Et sans patron, ça donne quoi ?

Un stagiaire de 2013 (les patrons n'étaient pas au programme du PPN!) a réalisé le programme suivant :

```
public class PizzeriaDependante {
    public Pizza creerPizza(String style, String type) {
        Pizza pizza = null;

        if (style.equals("Brest")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleBrest();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleBrest();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleBrest();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleBrest();
            }
        } else if (style.equals("Strasbourg")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleStrasbourg();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleStrasbourg();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleStrasbourg();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleStrasbourg();
            }
        } else {
            System.out.println("Erreur : type de pizza invalide");
            return null;
        }
        pizza.preparer(); pizza.cuire(); pizza.couper(); pizza.emballer();
        return pizza;
    }
}
```

QUESTION

1. Faites le compte du nombre de classes concrètes dont cette classe dépend.
2. Et si vous ajoutez des pizzas de style Marseille à cette Pizzeria ?

Solution 

1. 8
2. 12

À comparer aux 2 (fabrique et pizza) des Pizzérias avec Fabrique

Problème du main de test du jeu d'aventure

Vous avez sûrement dans votre `main` de l'application de jeu d'aventure une partie du code ressamblant à ceci :

Adaptation des comportements à la situation

```
if (choix.equals("Epee")) {
    perso.setArme(new ComportementEpee());
}
else if (choix.equals("Arc")) {
    perso.setArme(new ComportementArc());
    else if ...
    ...
}
```

Ce code est peu adaptatif et va soufrir des évolutions, par exemple :

- changement de la liste des armes possibles
- rajouter des `if then else` à chaque nouvelle arme
- suppression de certaines armes
- ...

QUESTION

1. Isoler ce code dans une classe `SimpleFabriqueArme` qui possèdera une méthode `creerComportementArme(String type)` qui retourne le comportement adapté en fonction du paramètre reçu.
2. Donnez le diagramme de classe UML™ de la nouvelle organisation.
3. Quelles différences a-t-on avec le patron Fabrique ? Donner quelques exemples d'extension du jeu d'aventure qui correspondraient à l'usage d'une Fabrique.
4. Donnez le diagramme de séquence du main. Par exemple avec le code de test suivant :

```
Chevalier perso = new Chevalier("JMI");

SimpleFabriqueArme fabrique = new SimpleFabriqueArme();
ComportementArme c = fabrique.creerComportementArme("Epee");

perso.setArme(c);
perso.frapper();
```

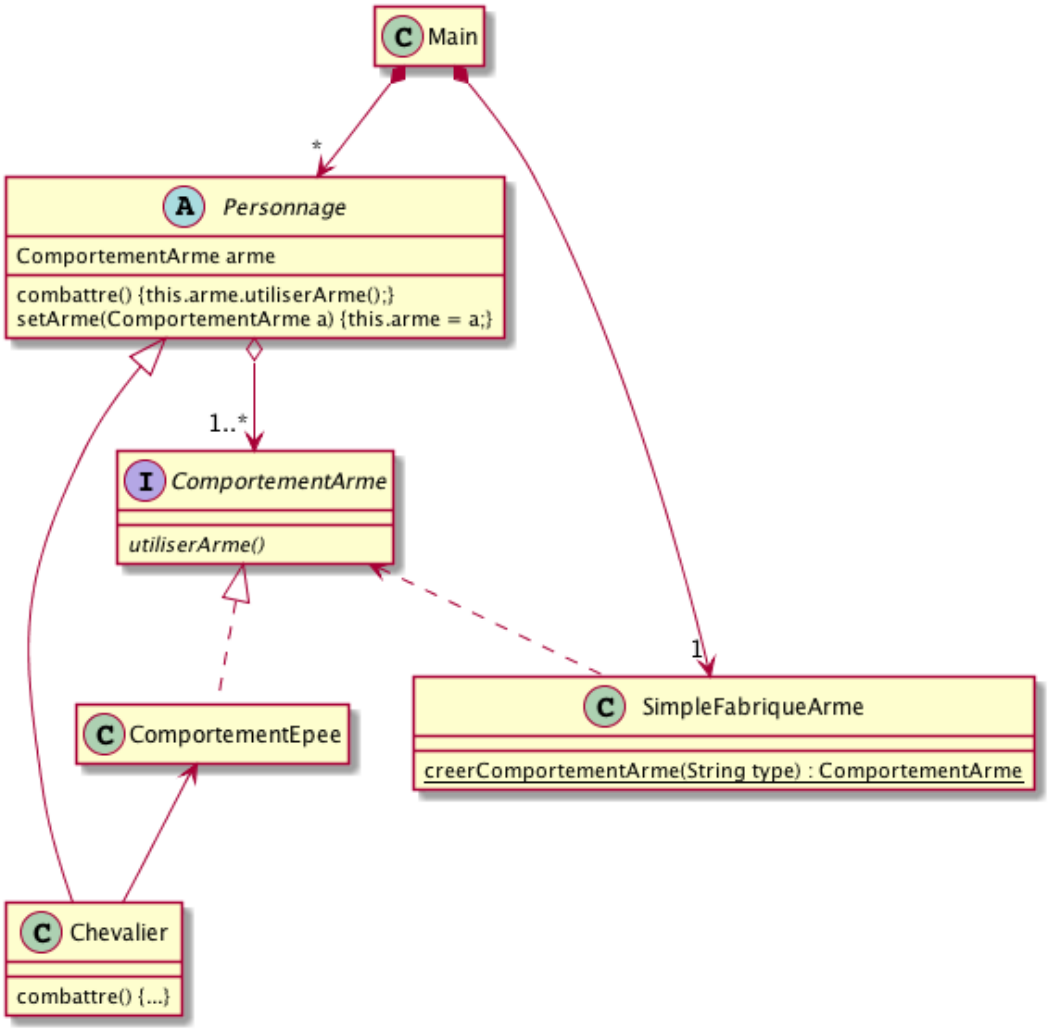
Solution 

1. Implémentation

   Extrait de FabriqueArme.java

```
public class SimpleFabriqueArme {
    public ComportementArme creerComportementArme(String type) {
        ComportementArme compAdequat = null;
        if (type.equals("Epee")) {
            compAdequat = new ComportementEpee();
        }
        else if (type.equals("Arc")) {
            compAdequat = new ComportementArc();
        }
        else compAdequat = new ComportementArmeless();
        return compAdequat;
    }
}
```

2. Diagramme de classe de la fabrique de comportements d'armes



3. Le créateur et le créateur concrêt sont dans une unique classe (SimpleFabriqueArme) car on a une seule fabrique unique pour le moment. On peut imaginer deux modes de jeu, avec des armes médiévales (épée en métal) ou modernes (épée sabre laser) qui seront traités dans deux fabriques concrètes différentes "FabriqueArmeMedievales" et "FabriqueArmeModernes".
4. Diagramme de séquence du `main`.

Diagramme de séquence illustrant Singleton et *Strategy*