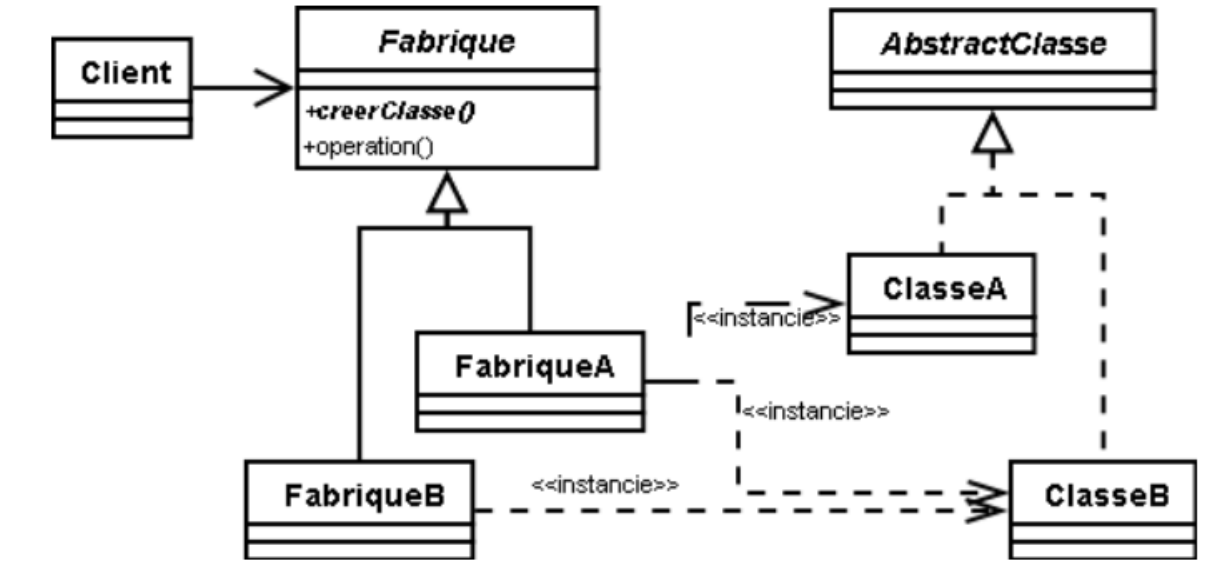
Le patron de conception : Factory

I. Présentation :

Diagramme de classes :



OBJECTIFS :

• Définir une interface pour la création d'un objet, mais laisser les sous-classes décider quelle classe instancier.

• Déléguer l'instanciation aux sous-classes.

RAISONS DE L'UTILISER :

Dans le fonctionnement d'une classe, il est nécessaire de créer une instance. Mais, au niveau de cette classe, on ne connaît pas la classe exacte à instancier.

Cela peut être le cas d'une classe réalisant une sauvegarde dans un flux sortant, mais ne sachant pas s'il s'agit d'un fichier ou d'une sortie sur le réseau.

La classe possède une méthode qui retourne une instance (interface commune au fichier ou à la sortie sur le réseau).

Les autres méthodes de la classe peuvent effectuer les opérations souhaitées sur l'instance (écriture, fermeture). Les sous-classes déterminent la classe de l'instance créée (fichier, sortie sur le réseau). Une variante du Pattern existe : la méthode de création choisit la classe de l'instance à créer en fonction de paramètres en entrée de la méthode ou selon des variables de contexte.

RESULTAT :

Le Design Pattern permet d'isoler l'instanciation d'une classe concrète.

RESPONSABILITES :

- **AbstractClasse** : définit l'interface de l'objet instancié.
- **ClasseA et ClasseB** : sont des sous-classes concrètes d'AbstractClasse. Elles sont instanciées par les classes Fabrique.
- **Fabrique** : déclare une méthode de création (creerClasse). C'est cette méthode qui a la responsabilité de l'instanciation d'un objet AbstractClasse. Si d'autres méthodes de la classe ont besoin d'une instance de AbstractClasse, elles font appel à la méthode de création. Dans l'exemple, la méthode operation() utilise une instance de AbstractClasse et fait donc appel à la méthode creerClasse. La méthode de création peut être paramétrée ou non. Dans l'exemple, elle est paramétrée, mais le paramètre n'est significatif que pour la FabriqueA.
  - **FabriqueA et FabriqueB** : substituent la méthode de création. Elles implémentent une version différente de la méthode de création.
  - **La partie cliente** utilise une sous-classe de Fabrique.

IMPLEMENTATION JAVA :

AbstractClasse.java

```
/**
 * Définit l'interface de l'objet instancié.
 */
public interface AbstractClasse {

    /**
     * Méthode permettant d'afficher le nom de la classe.
     * Cela permet de mettre en evidence la classe créée.
     */
    public void afficherClasse();
}
```

ClasseA.java

```
/**
 * Sous-class de AbstractClasse.
 */
public class ClasseA implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage.
     * Indique qu'il s'agit d'un objet de classe ClasseA
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseA'");
    }
}
```

ClasseB.java

```
/**
 * Sous-class de AbstractClasse.
 */
public class ClasseB implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage.
     * Indique qu'il s'agit d'un objet de classe ClasseB
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseB'");
    }
}
```

Fabrique.java

```
/**
 * Déclare la méthode de création.
 */
public abstract class Fabrique {

    private boolean pIsClasseA = false;

    /**
     * Méthode de création
     */
    public abstract AbstractClasse creerClasse(boolean pIsClasseA);

    /**
     * Méthode appelant la méthode de création.
     * Puis, effectuant une opération.
     */
    public void operation() {
        // Change la valeur afin de varier le paramètre
        // de la méthode de création
        pIsClasseA = !pIsClasseA;

        // Récupère une instance de classe "AbstractClasse"
        AbstractClasse lClasse = creerClasse(pIsClasseA);

        // Appel la méthode d'affichage de la classe
        // afin de savoir la classe concrète
        lClasse.afficherClasse();
    }
}
```

FabriqueA.java

```
/**
 * Substitue la méthode "creerClasse".
 * Instancie un objet "ClasseA".
 */
public class FabriqueA extends Fabrique {

    /**
     * Méthode de création
     * La méthode retourne un objet ClasseA, si le paramètre est true.
     * La méthode retourne un objet ClasseB, sinon.
     * @return Un objet de classe ClasseA ou ClasseB.
     */
    public AbstractClasse creerClasse(boolean pIsClasseA) {
        if(pIsClasseA) {
            return new ClasseA();
        }
        else {
            return new ClasseB();
        }
    }
}
```

FabriqueB.java

```
/**
 * Substitue la méthode "creerClasse".
 * Instancie un objet "ClasseB".
 */
public class FabriqueB extends Fabrique {

    /**
     * Méthode de création
     * La méthode ne tient pas compte du paramètre
     * et instancie toujours un objet "ClasseB"
     * @return Un objet de classe ClasseB.
     */
    public AbstractClasse creerClasse(boolean pIsClasseA) {
        return new ClasseB();
    }
}
```

FactoryMethodPatternMain.java

```
public class FactoryMethodPatternMain {

    public static void main(String[] args) {
        // Création des fabriques
        Fabrique lFactoryA = new FabriqueA();
        Fabrique lFactoryB = new FabriqueB();

        // L'appel de cette méthode avec FabriqueA provoquera
        // l'instanciation de deux classes différentes
        System.out.println("Avec la FabriqueA : ");
        lFactoryA.operation();
        lFactoryA.operation();
        lFactoryA.operation();
        // L'appel de cette méthode avec FabriqueB provoquera
        // toujours l'instanciation de la même classe
        System.out.println("Avec la FabriqueB : ");
        lFactoryB.operation();
        lFactoryB.operation();
        lFactoryB.operation();

        // Affichage :
        // Avec la FabriqueA :
        // Objet de classe 'ClasseA'
        // Objet de classe 'ClasseB'
        // Objet de classe 'ClasseA'
        // Avec la FabriqueB :
        // Objet de classe 'ClasseB'
        // Objet de classe 'ClasseB'
        // Objet de classe 'ClasseB'
    }
}
```

II. Exemple d'application :

On suppose que l'on souhaite faire une machine automatique cuisinant des tartes aux fruits. Pour cela, on développe dans un premier temps une application Java permettant de simuler le comportement de la machine.

La classe principale de l'application est **MachineTarte** et dispose d'une méthode **commanderTarte** qui aura l'allure suivante :

```
public Tarte commanderTarte() {
    Tarte tarte = new Tarte();
tarte.preparer();
tarte.cuire();
tarte.emballer();
return tarte;}
```

Les méthodes « preparer », « cuire » et « emballer » sont des méthodes simulant le travail effectif de la machine.

1. on suppose que l'on a une machine évoluée qui peut faire plusieurs types de tartes aux fruits. On va donc rendre Tarte abstraite et créer une hiérarchie de classes représentée sur la figure 1.

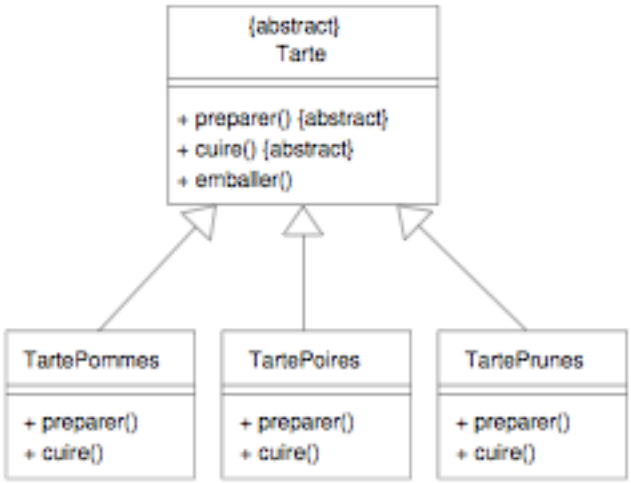


Fig. 1 – Hiérarchie de classes représentant les tartes

On va donc modifier la méthode commanderTarte pour utiliser les nouveaux types de tartes : on va passer en paramètre de commanderTarte une chaîne de caractères précisant le type de tarte choisi (par exemple "pommes" ou "poires" ou "prunes"). En fonction de cette chaîne de caractères, on construira la tarte correspondante.

   a. Modifier la méthode commanderTarte (sachant qu'elle retourne la Tarte demandée qui peut être égale à « null » si le type de tarte n'existe pas). Attention, les seuls types connus actuellement sont "pommes", "poires" et "prunes".

2. si l'on souhaite ajouter de nouvelles tartes, il va falloir modifier la méthode commanderTarte. Or on souhaite fermer commanderTarte à la modification. Pour résoudre ce problème, on va déléguer la création des tartes à une classe **FabriqueTarte** à travers une méthode **creerTarte**.

(a) Complétez le diagramme de classes de la figure 1 en représentant, en plus de la hiérarchie des tartes, les classes MachineTarte qui permet de commander une tarte et FabriqueTarte qui permet de fabriquer la tarte commandée.

(b) représenter sur un diagramme de séquences les interactions entre les classes lors de la commande d'une tarte aux pommes.

(c) donner le code Java des classes FabriqueTarte et MachineTarte.

3. On suppose maintenant que l'on veut pouvoir créer deux grands types de tartes : des tartes normales et des tartes sans gluten (pour les personnes allergiques). La hiérarchie de classes représentant les tartes va donc s'en trouver modifiée. L'utilisateur choisira au départ la « famille » de tarte qu'il veut (sans gluten ou normale), puis le type de la tarte (aux pommes, aux poires etc.). Dans notre simulation, nous voulons donc avoir deux types de MachineTarte : un pour les tartes classiques et l'autre pour les tartes sans gluten.

   Une solution est de spécialiser FabriqueTarte en FabriqueTarteNormale et FabriqueTarteSansGluten.

(a) Que pensez-vous de cette solution ? En particulier, comment créer une machine pour un type de tarte particulier ? Contrôle-t-on l'utilisation qui sera faite des fabriques de tarte ?

(b) Donner le patron permettant de résoudre ces problèmes.

(c) Pourquoi utilise-t-on une interface (ou une classe abstraite) pour Product ?

(d) Proposer une adaptation de ce pattern à notre problème via un diagramme de classes.

(e) Ecrire les classes MachineTarte et MachineTarteSimple.