
INF 4067 - UML ET DESIGN PATTERNS

Devoir 1 - Implémentation des patterns de construction

Author

CHEDJOUN KENGUEP Dave 20U2757

Département Informatique - Master 1 - Genie Logiciel

Octobre 2023

Diagrammes UML et résultats

Il nous a été donné comme devoir d'implémenter les designs patterns plus précisément ceux de structuration. Pour se faire, nous avons pour chaque implémentation, deux versions deux codes ainsi que deux versions de diagrammes. Nous vous les présenterons ci dessous en vous donnant, une brève description, les liens vers les dépôts gitHub, des screenshots des résultats et les images avec les modèles UML.

1 Factory Pattern

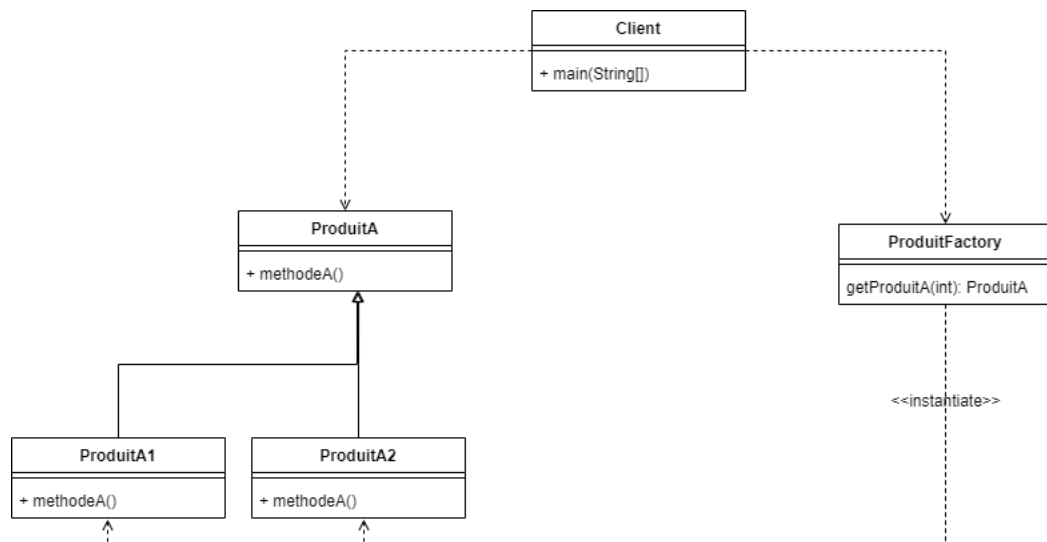
Pour se faire, ici nous avons implémenté les deux méthodes présentées en cours à savoir, l'utilisation d'une fabrique abstraite, et d'une methode de fabrique.

1.1 Implementation avec la factory Method

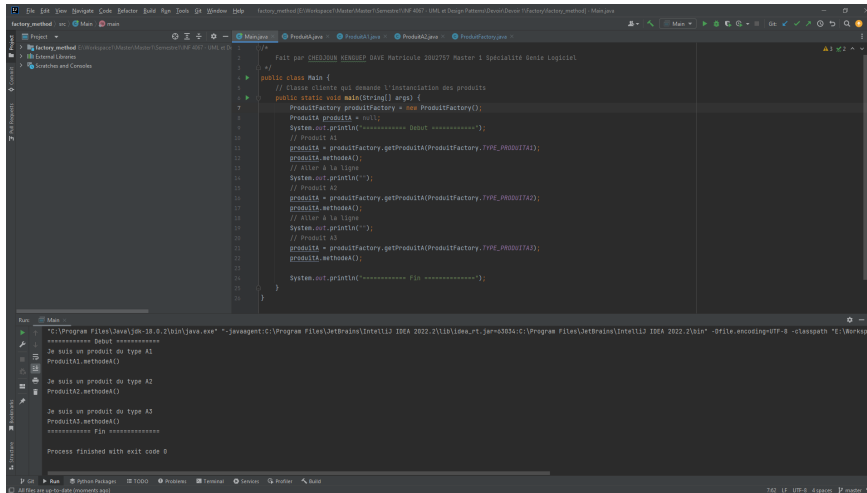
1.1.1 Diagrammes UML

En se basant sur la structure générale que nous offre cette méthode, et le client pouvant avoir 3 types de produits à fabriquer nous avons obtenu le diagramme suivant:

- Pour l'implémentation avec deux produits A1 et A2 :



- Pour l'implémentation avec trois produits A1, A2 et A3 :

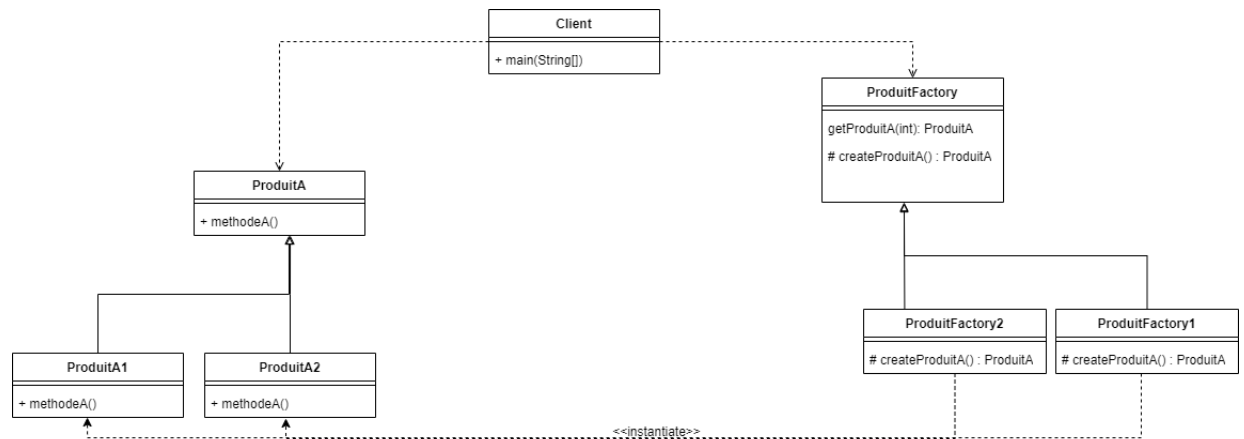


1.2 Implementation avec la factory

1.2.1 Diagrammes UML

En se basant sur la structure générale que nous offre la methode factory, et le client pouvant avoir 3 ou 2 types de produits à fabriquer nous avons obtenu le diagramme suivant:

- Pour l'implémentation avec deux produits A1 et A2 :



- Pour l'implémentation avec trois produits A1, A2 et A3 :


```

1 // Fait par [00000000000000000000000000000000] Master 1 Spécialité Santé Logiciel
2 // A. Bouhass
3
4 public class Main {
5     // Classe cliente qui demande l'instanciation des produits
6     // A. Bouhass
7     public static void main(String[] args) {
8         // Initialisation des fabricques abstraites
9         ProductFactory productFactory1 = new ProductFactory1();
10        ProductFactory productFactory2 = new ProductFactory2();
11        ProductFactory productFactory3 = new ProductFactory3();
12
13        Produits produits;
14        System.out.println("----- debut -----");
15        // Produit A1
16        System.out.println("Utilisation de la premiere fabrique");
17        produits = productFactory1.getProduitA();
18        produits.methodA();
19        // Produit A2
20        System.out.println("");
21        System.out.println("Utilisation de la seconde fabrique");
22        produits = productFactory2.getProduitA();
23        produits.methodA();
24    }
25 }

```

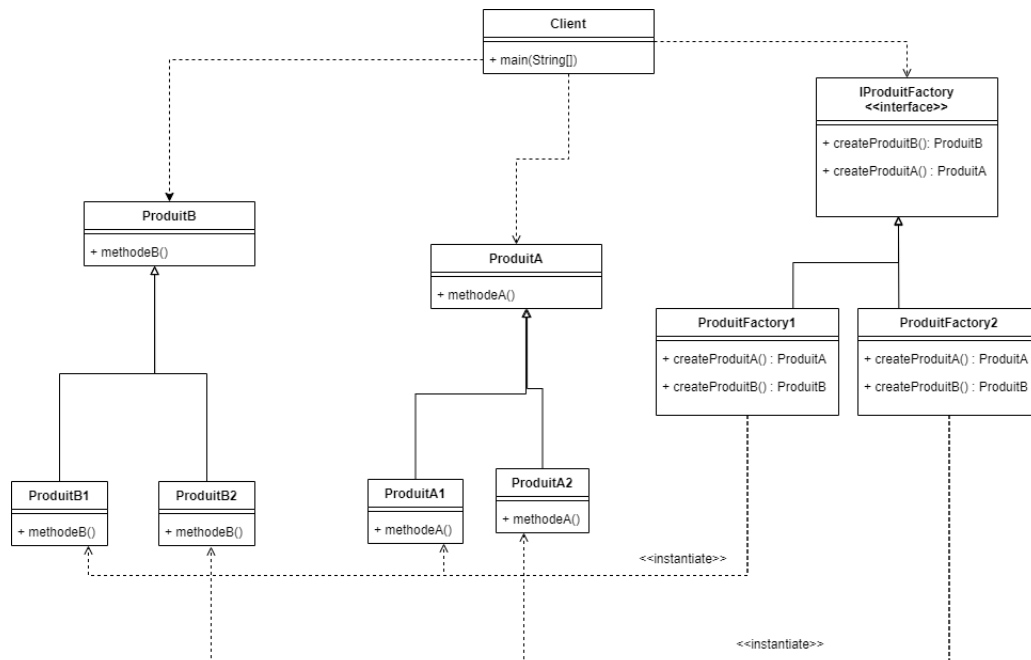
2 Abstract Factory Pattern

2.1 Implementation de l' abstract factory

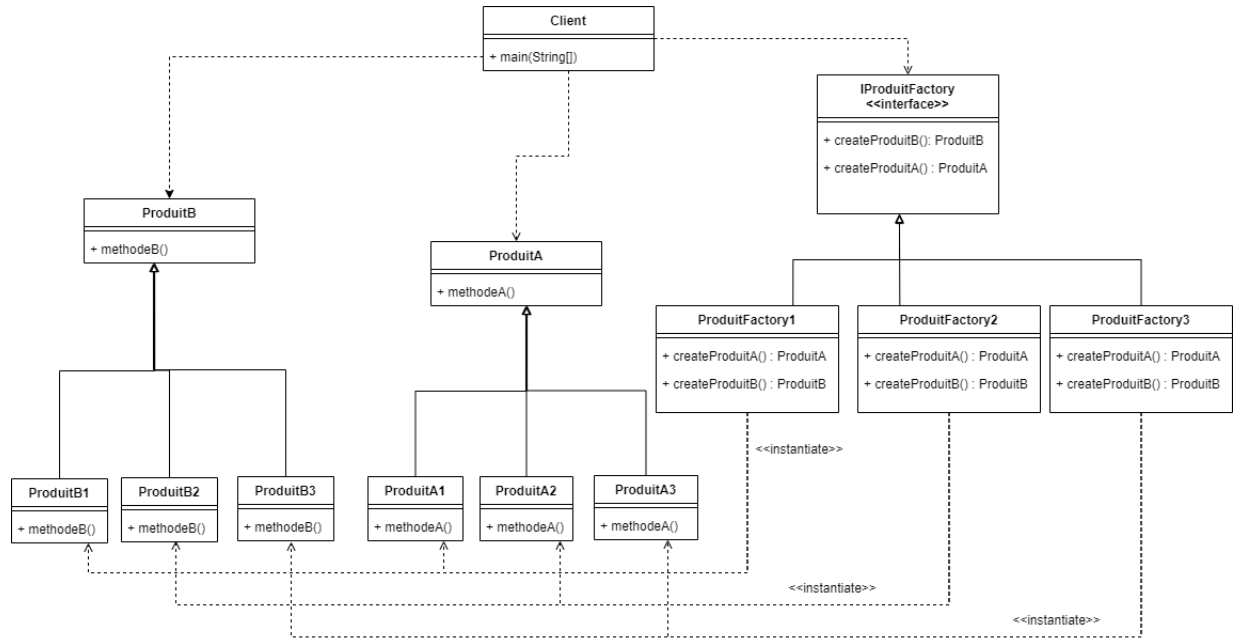
2.1.1 Diagrammes UML

En se basant sur la structure générale que nous offre la methode, et le client pouvant avoir 3 ou 2 familles de produits à fabriquer nous avons obtenu le diagramme suivant:

- Pour l'implémentation avec deux produits A1 et A2 :



- Pour l'implémentation avec trois produits A1, A2 et A3 :



2.1.2 Codes et Résultats

Le code ayant permis l'implémentation se trouve dans le dossier code, avec le nom `abstract_factory`. Le code ayant été mis sur `gitHub`, le lien pour y accéder est le suivant : https://github.com/DavePhil/abstract_factory.git. On retrouvera deux branches: la branche **master** pour l'implémentation avec 3 produits et la branche **2products** pour l'implémentation avec 2 produits. Les résultats sont les suivants:

- Pour l'implémentation avec deux produits A1 et A2 :

```

// IProduitFactory.java
interface IProduitFactory {
    ProduitA createProduitA();
    ProduitB createProduitB();
}

// ProduitA.java
abstract class ProduitA {
    abstract void methodeA();
}

// ProduitB.java
abstract class ProduitB {
    abstract void methodeB();
}

// ProduitFactory1.java
class ProduitFactory1 implements IProduitFactory {
    @Override
    public ProduitA createProduitA() {
        return new ProduitA1();
    }
    @Override
    public ProduitB createProduitB() {
        return new ProduitB1();
    }
}

// ProduitFactory2.java
class ProduitFactory2 implements IProduitFactory {
    @Override
    public ProduitA createProduitA() {
        return new ProduitA2();
    }
    @Override
    public ProduitB createProduitB() {
        return new ProduitB2();
    }
}

// ProduitFactory3.java
class ProduitFactory3 implements IProduitFactory {
    @Override
    public ProduitA createProduitA() {
        return new ProduitA3();
    }
    @Override
    public ProduitB createProduitB() {
        return new ProduitB3();
    }
}

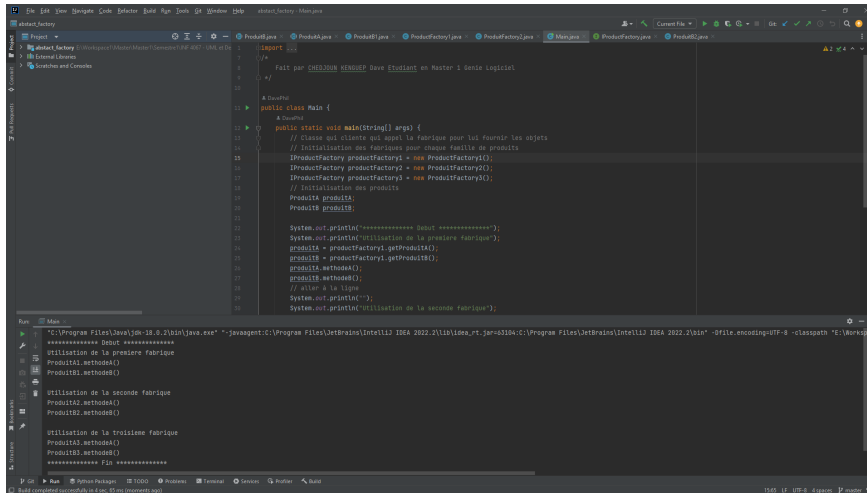
// Client.java
class Client {
    public static void main(String[] args) {
        // Classe qui appelle la fabrique pour les fournir les objets
        // Initialisation des fabriques pour chaque famille de produits
        IProduitFactory productFactory1 = new ProduitFactory1();
        IProduitFactory productFactory2 = new ProduitFactory2();
        // Initialisation des produits
        ProduitA produitA;
        ProduitB produitB;

        System.out.println("===== Utilisation de la premiere fabrique =====");
        System.out.println("Utilisation de la premiere fabrique");
        produitA = productFactory1.getProduitA();
        produitB = productFactory1.getProduitB();
        produitA.methodeA();
        produitB.methodeB();

        // Utilisation de la seconde
        System.out.println("===== Utilisation de la seconde fabrique =====");
        produitA = productFactory2.getProduitA();
        produitB = productFactory2.getProduitB();
        produitA.methodeA();
        produitB.methodeB();
    }
}

```

- Pour l'implémentation avec trois produits A1, A2 et A3 :



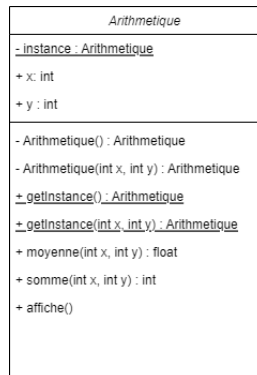
3 Singleton Pattern

3.1 Implementation du singleton

3.1.1 Diagrammes UML

En se basant sur la structure générale que nous offre la methode, et le client pouvant deux attributs (x et y) ou alors 3 attributs (x, y, name); ainsi que deux constructeurs avec paramètres et sans paramètres et enfin des fonctions de récupération des instances fonction des constructeurs nous avons obtenu les diagrammes suivants:

- Pour l'implémentation avec les attributs x et y :



- Pour l'implémentation avec les attributs x, y et name:

Arithmetique
- instance : <u>Arithmetique</u> + x : int + y : int + name : String
- Arithmetique() : Arithmetique - Arithmetique(int x, int y) : Arithmetique - Arithmetique(int x, int y, String name) : Arithmetique + <u>getInstance() : Arithmetique</u> + <u>getInstance(int x, int y) : Arithmetique</u> + <u>getInstance(int x, int y, String name) : Arithmetique</u> + moyenne(int x, int y) : float + somme(int x, int y) : int + multiplication(int x, int y) : int + soustraction(int x, int y) : int + affiche()

3.1.2 Codes et Résultats

Le code ayant permis l'implémentation se trouve dans le dossier code, avec le nom singleton. Le code ayant été mis sur gitHub, le lien pour y accéder est le suivant : <https://github.com/DavePhil/singleton.git>. On retrouvera deux branches: la branche **master** pour l'implémentation avec 2 attributs et la branche **version2** pour l'implémentation avec 3 attributs. Les resultats sont les suivants:

- Pour l'implémentation avec les attributs x et y :

```

1 package fr.davephil.singleton;
2
3 public final class Arithmetique {
4     // Attributs
5     private static Arithmetique instance;
6
7     // Constructeur sans paramètre
8     // La présence d'un constructeur privé, supprime le constructeur par défaut
9     private Arithmetique() {
10         super();
11     }
12
13     // Constructeur avec paramètres
14     private Arithmetique(int x, int y) {
15         this.x = x;
16         this.y = y;
17     }
18
19     // Méthode de récupération de l'instance sans paramètre
20     // La présence d'un constructeur privé, supprime le constructeur par défaut
21     public static Arithmetique getInstance() {
22
23     }
24 }

```

Run: C:\Program Files\Java\jdk-10.0.2\bin\java.exe -Djava.class.path=C:\Program Files\JetBrains\IntelliJ IDEA 2022.2\lib\idea_rt.jar -Didea.encoding=UTF-8 -classpath "E:\Workspace\Singleton\src\main\java\fr\davephil/singleton\Arithmetique.class" fr.davephil.singleton.Arithmetique

La somme est: 0

Je crée une instance mes valeurs sont: x: 0 et y: 0

Je crée une instance mes valeurs sont: x: 0 et y: 0

Process finished with exit code 0

- Pour l'implémentation avec les attributs x, y et name :

```

// Implementation de la classe singleton
4 @Singleton
5 public final class Arithmetique {
6     // Singleton
7     private static Arithmetique instance;
8
9     // Singleton
10    private int x;
11    // Singleton
12    private int y;
13    // Singleton
14    private String name;
15
16    // constructeur sans parametre
17    // la presence d'un constructeur privé, supprime le constructeur par défaut
18    private Arithmetique() {
19        super();
20    }
21
22    // Constructeur avec 2 parametres
23    // Singleton
24    private Arithmetique(int x, int y) {
25        this.x = x;
26        this.y = y;
27    }
28 }

```

Run: Main

```

C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2\lib\idea_rt.jar=42950:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2\bin" -file.encoding=UTF-8 -classpath "C:\Worksp
La somme est: 9
La multiplication est: 6
La division est: 3
La moyenne est: 4.5
Je sais une instance mes valeurs sont: x: 5, y: 7 et name: Math
Je sais une instance mes valeurs sont: x: 5, y: 7 et name: Math
Process finished with exit code 0

```

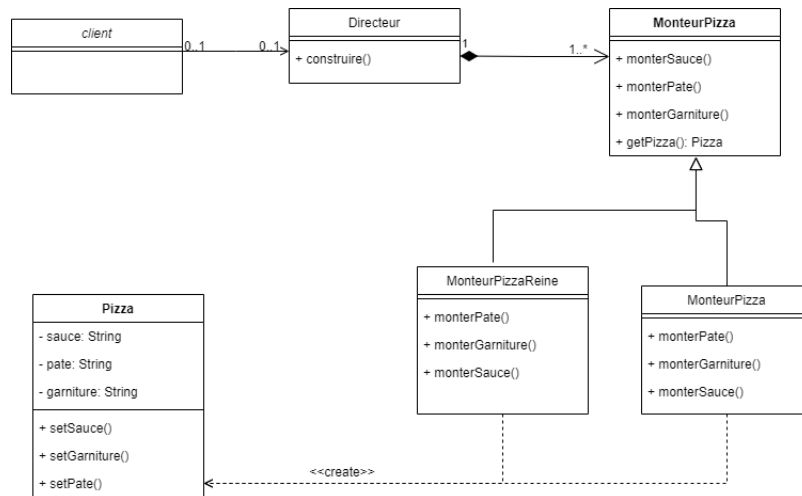
4 Builder Pattern

4.1 Implementation du builder

4.1.1 Diagrammes UML

En se basant sur la structure générale que nous offre la methode, nous avons obtenu le diagramme suivant pour l'exercice des pizza du cours:

- Pour l'implémentation avec deux pizza:



- Pour l'implémentation avec trois pizza:

