المعهد الوطني للبريد والمواصلات
ⵓⵔⴰⵔⵉ ⵏⵓⵟⵏⴰⵢ ⵏ ⵜⵓⵙⵙⵓⵔⵉⵏ ⴰ ⵜⵜⵓⵙⵙⵓⵔⵉⵏ
Institut National des Postes et Télécommunications

Design patterns et architectures logicielles

Étude de cas : le design pattern *Abstract Factory*

Filière : SUD-Cloud & IoT

Travail réalisé par :

- Chrif El Asri Hanane
- El Gamous Khalid

Encadrant:

Prof. Abdeslam EN-NOUAARY

2020/2021

Introduction

Fabrique abstraite est un patron de conception qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète. On va comprendre comment il fonctionne grâce à une étude de cas.

I. Problématique

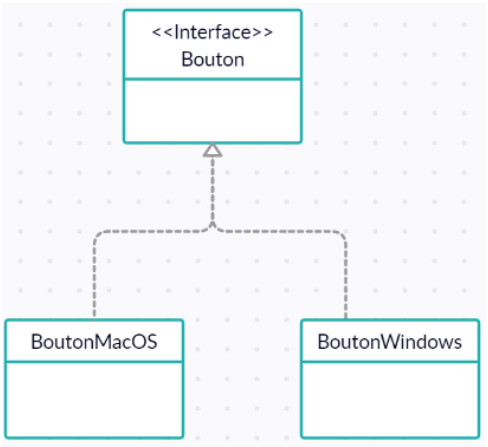
Imaginons la création d'un bouton et de cases à cocher. Notre code contient les classes suivantes :

1. Une famille de produits appartenant à un même thème : bouton + cases à cocher.
2. Plusieurs variantes de cette famille. Par exemple, les produits bouton + cases à cocher sont disponibles dans les variantes suivantes : macOS + Windows.

On doit trouver une solution pour créer des objets individuels et les faire correspondre à d'autres objets de la même famille (bouton pour macOS, bouton pour Windows, cases à cocher pour macOS et cases à cocher pour Windows). De plus, on n'a pas envie de réécrire notre code chaque fois que nous ajoutons de nouvelles familles de produits à notre programme.

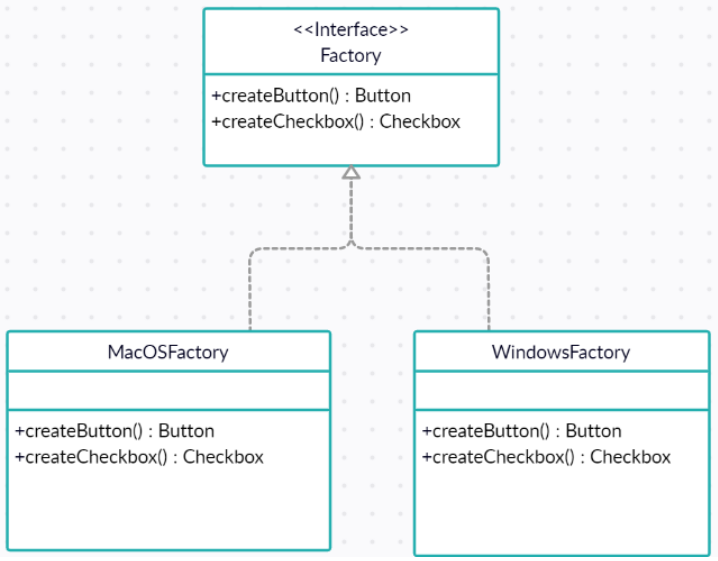
II. Solution avec Abstract Factory

La première chose que propose la fabrique abstraite est de déclarer explicitement des interfaces pour chaque produit de la famille de produits (dans notre cas : bouton + cases à cocher). Toutes les autres variantes de produits peuvent ensuite se servir de ces interfaces. Par exemple, toutes les variantes de boutons peuvent implémenter l'interface `Bouton` ; toutes les variantes de cases à cocher peuvent implémenter `CaseACocher`, etc.



La prochaine étape est de déclarer la *fabrique abstraite* — une interface armée d'une liste de méthodes de création pour toutes les familles de produits (par exemple : `créerBouton` et `créerCaseACocher`). Ces méthodes doivent renvoyer tous les types de produits **abstraits** des interfaces que nous avons créées précédemment : `bouton`, `CaseACocher`, etc.

Pour chaque variante d'une famille de produits, nous créons une classe fabrique qui implémente l'interface `AbstractFactory`. Une fabrique est une classe qui retourne un certain type de produits. Par exemple, la `MacOSFactory` peut uniquement créer des `MacOSButton` et des `MacOSCheckbox`.



Le code client travaille simultanément avec les interfaces abstraites respectives des fabriques et des produits. Nous pouvons ainsi changer le type de fabrique passé au code client et la variante de produit qu'il reçoit, sans avoir à le modifier.

Imaginons un cas où le client désire une fabrique qui peut produire un bouton. Le client n'a pas à se préoccuper de la classe de la fabrique ni du type de bouton qu'il va obtenir. Il doit traiter les boutons de la même manière, que ce soit un modèle de style moderne ou victorien, en utilisant l'interface abstraite `Buton`. Grâce à cette approche, le client ne sait qu'une seule chose à propos du bouton, c'est qu'elle implémente la méthode `OnClik`.

Il ne reste plus qu'un point à éclaircir : si le client n'est lié qu'aux interfaces abstraites, comment les objets de la fabrique sont-ils créés ? En général, l'application crée un objet fabrique concret lors de l'initialisation. Mais avant cela, l'application doit choisir le type de la fabrique en fonction de la configuration ou des paramètres d'environnement.

III. Etude de cas

Dans cet exemple, les boutons et cases à cocher feront office de produits. Ils possèdent deux variantes : macOS et Windows.

La fabrique abstraite définit une interface pour la création des boutons et des cases à cocher. Deux fabriques concrètes retournent les deux produits d'une seule variante.

Le code client manipule les fabriques et les produits via leurs interfaces abstraites. Ainsi, le code client peut travailler avec plusieurs variantes de produits en fonction du type de l'objet Fabrique.

Ici on va créer un bouton et des cases à cocher selon la nature de l'os du l'ordinateur où le code est exécuté. D'où la ligne de code *System.getPreperty("os.name ")* qui vérifie la nature du os.

```
private static Application configureApplication() {
        Application app;
        Factory factory;
        String osName = System.getProperty("os.name").toLowerCase();
        if (osName.contains("mac")) {
            factory = new MacOSFactory();
            app = new Application(factory);
        } else {
            factory = new WindowsFactory();
            app = new Application(factory);
        }
        return app;
    }

    public static void main(String[] args) {
        Application app = configureApplication();
        app.paint();
    }
```

Puisque l'ordinateur dans lequel on a essayé ce code fonctionne avec un système d'exploitation Windows, on a obtenu le résultat suivant :

```
You have created WindowsButton.
You have created WindowsCheckbox.
PS C:\Users\dell\Desktop\abstractFactory>
```

Conclusion

On constate qu'il est préférable d'utiliser la fabrique abstraite si le code a besoin de manipuler des produits d'un même thème, mais qu'on ne veut pas qu'il dépende des classes concrètes de ces produits — soit on ne les connait pas encore, soit on veut juste rendre notre code évolutif.

Ps : Vous pouvez trouver le code source sous le lien :
https://github.com/hanane-ca/Design-Patterns.git