Exercice Refactoring

mai 2024

Sujet

Cet exercice comporte un ensemble de classes qui permettent l'édition d'une facture à partir d'un achat.

Objectif

- Appliquer les principes **SOLID**
- Travailler le principe de **identify/isolate/extract/Inject**
- Identifier les **Code Smell**
- Implémenter des Tests Unitaire

HOW TO

Sélection

Comment commencer un refactoring ? On choisit une seule classe et on reste concentré sur elle seule. Avant de passer à la suivante.
Dans l'exemple suivant nous allons l'appeler **Classe A**

identify/isolate/extract/Inject

Derrière ces 4 mots se cache la méthodologie de refactoring d'une classe trop grosse.

1. **Identify & Isolate:** On découpe en méthode privé les gros blocs de la méthode publique
    a. Chaque contenu d'une instruction de contrôle (If, while, witch) peut être transformé en méthodes privés de **Classe A**
    b. Chaque ensemble technique très liés doit être transformé en méthode privé de classe **Classe A**
        i. Exemple : le block de ligne http/json de la classe Checkout

2. **Extract** Extraction de classe :
    a. On identifie les méthodes privés ou ensemble de méthode privé qui peuvent être extraite
    b. On transforme une ou plusieurs méthodes privés du même sujet en une **Classe B**

    c. On renomme le nom de la nouvelle méthode public de **Classe B** en se plaçant du côté du client, celui qui va appeler la méthode. C'est a dire **Classe A** De quoi a t'elle besoin?
        i. Exemple : *getShippingCost()* dans Checkout, et pas *postRequest().* Car *Checkout* à juste besoin d'un *ShippingCost*.
    d. On masque au maximum les détails technique / d'implémentation :
        i. La méthode public de la **classe B** ne doit pas avoir de paramètre technique (url, config etc)
        ii. Ces paramètres doivent être dans le constructeur de la **Classe B**, en config, en injection.

3. **Extract** Extraction d'interface
    a. On crée une interface à partir de la classe B
    b. On vérifie que la méthode publique de cette interface est générique. Qu'elle n'est pas polluée par les détails de l'implémentation.
        i. Mauvaise pratique : *getShippingCost(URL url);*
4. **Inject**
    a. On passe l'interface dans le constructeur de la classe A
    b. On remplace dans le code de la classe A les appels à vers classe B par les appels vers l'interface

Nettoyage

On profite du refactoring

5. Renommage des variables
    a. cf bonne pratiques en dessous

6. Commentaires :
    a. Les prendre en compte mais se méfier également
    b. Les supprimer en renommant des méthodes privés

7. Supprimer les codes morts :
    a. Code commenté
    b. Code non atteints
    c. Code dupliqué

Bonne pratiques

Classe

Chaque classe doit n'avoir qu'une seule méthode publique. Idéalement.

- Avec un nom orienté business = le service qu'elle rend
    - Exemple : nommage de la methode public *getShippingCost()* et pas *connectToLaPosteApi()*
- Avec des appels vers ses méthodes privés
    - Repensez au refactoring de CheckoutService

- Avoir un même niveau de lecture
    - on évite toute forme de boucle
    - Avoir moins de 5 lignes

Chaque Classe doit exposer clairement ses dépendances :
- Les services doivent TOUS être passé en constructeur
- Pas de new() dans le code

Nommage

Le nom de nos méthodes doit décrire le service qu'elles rendent, le *pourquoi*.
Et pas le *comment*.
Exemple : *GetTotalCost()* et pas *perfomAdditionOnParam()*

Le nom des variables doit être lisible, complet et uniforme.
Exemple : List<String> listOfBookName()
Mauvaise pratique : *lstBook ; lBookName ; BookNames*

Refactoring

- On n'ajoute pas de nouvelle features ou de nouvelles fonctionnalités au code qu'on refactore. Jamais.
    - On ne pense pas à "plus tard"
    - Exemple : vouloir créer un SuperHttpClient designé pour plus tard avec des methodes qui ne sont pas utilisé dans le code qu'on refactor.
- On termine toujours de refactorer une classe et de la tester avant de passer à la suite
    - Exemple :On ne refactore par LaPosteCheckoutService tant qu'on a pas fini de refactorer Checkout et de tester Checkout

Testing

- On écrit un premier test et on va au bout avant d'écrire, voir même de penser au suivant :
    - Mauvaise pratique : créer toutes les méthodes de tests vides avant d'avoir implémenter un premier test
- Un test unitaire ne doit pas avoir de dépendances avec d'autres classes de service
    - Exemple : pas de *LaPosterShippingService* dans le *CheckoutUnitTests*
- Un test unitaire doit tester une seule chose