MASTER MIAGE 2ème année
Université Paris Nanterre

Mémoire de fin d'études présenté pour
l'obtention du grade de master

Comment les flots de contrôle peuvent-ils nous permettre de faire du refactoring de code en Java.

Présenté par :
Thibault Sartre

Tuteur :
Emmanuel Hyon

Septembre 2018 — Juillet 2019

Sommaire

Chapitre 1

Contexte

1.1 Introduction

Le refactoring est une activité d'ingénierie logiciel consistant à modifier le code source d'une application de manière à améliorer sa qualité sans altérer son comportement vis-à-vis des utilisateurs. L'objectif du refactoring est de réduire les coûts de maintenance et de pérenniser les investissements tout au long du cycle de vie du logiciel en se concentrant sur la maintenabilité et l'évolutivité.[6]
"With refactoring you can take a bad design, chaos even, and rework it into well-designed code."[4]
Le refactoring permet donc de passer d'un code possédant de mauvaise base à un code propre.
Un bon refactoring doit pouvoir améliorer la qualité d'un code tout en gardant son fonctionnement du point de vue de l'utilisateur. Concernant la partie des tests, tout les tests qui fonctionnaient avant le refactoring se doivent d'être fonctionnels après.
Dans ce mémoire, nous allons analyser différentes techniques de refactoring. Puis nous allons étudier le principe des flots de contrôle.
Enfin je vous proposerais et évaluerais une solution pour faire du refactoring de code en utilisant les flots de contrôle pour le langage Java.
Ce sujet est intéressant et actuel car il faut commencer à ce soucier du code que l'on produit car plus l'on avance dans le temps plus les programmes sont lourds et contiennent de lignes de code. Avec la puissance des ordinateurs actuels, la plus part des développeurs ne prennent plus le temps d'écrire des codes de qualité car la machine sera de toute façon assez rapide pour compenser un code de mauvaise qualité.[3]
Avec le temps, la relecture et la modification de code sera difficile avec les programmes qui deviennent de plus en plus gros.[3]
Pour essayer de diminuer cette quantité de travail à l'avenir, il faut commencer à produire du code de qualité et bien structuré. Concernant les codes déjà existant il faudra donc faire du refactoring de code. Or le refactoring peut-être long selon la qualité des projets que l'on traite. Il serait donc intéressant et très utile d'avoir un programme permettant d'automatiser des parties de refactoring pour permettre de gagner du temps précieux. Ce mémoire a pour but de fournir une solution qui permettrais de faire du refactoring de code facilement pour gagner du temps tout en produisant du code de qualité.[2]

Chapitre 2

Le refactoring

2.1 Composed Method Pattern

L'un des grands ennemie des développeurs est de faire des méthodes trop longue qui sont difficilement compréhensible. Une grande partie du refactoring est donc consacrée à la composition correcte des méthodes.[1]

L'objectif de cette méthode de refactoring est de :
- rendre les méthodes facilement compréhensible.
- simplifier les méthodes en les brisants en plusieurs méthodes plus petites.
- supprimer la duplication de code.
- nommer proprement les variables, méthodes et paramètres pour comprendre leurs utilités au premier coup d'œil.
- pouvoir faire des tests plus facilement car les morceaux de méthodes peuvent être testés individuellement.[5]

2.1.1 Extraction de méthode

Présentation : La technique d'extraction de méthode permet comme son nom l'indique, d'extraire des méthodes (ou fonction) du code. Car plus il y a de ligne dans une méthode plus il est difficile de comprendre ce que fait la méthode.
Il faut donc lorsque c'est possible extraire du code pour former d'autre méthode et simplifier le code.

```java
void affichagePrix(int nbProduit, int prixProduit) {

    //Calcule prix
    int prix = nbProduit * prixProduit;

    //Affiche prix
    System.out.println("Le prix est de " + prix);
}
```

Avant extraction de méthode

Exemple : Au dessus nous avons une méthode qui va afficher le prix d'un produit tout en calculant elle même le prix.
Si l'on applique l'extraction de méthode, on va obtenir l'exemple du bas, c'est à dire extraire la méthode de calcule du prix car on en aura surement besoin ailleurs. On remplace donc la calcule du prix dans la méthode affichagePrix par un appel à la méthode calculePrix.

```
void affichagePrix(int nbProduit, int prixProduit) {

    System.out.println("Le prix est de " + calculePrix(nbProduit,prixProduit));
}

int calculePrix(int nbProduit, int prixProduit) {

    return nbProduit * prixProduit;
}
```

Après extraction de méthode

Bénéfices : Pour que cette technique de refactoring soit la plus efficace, il faut absolument nommer ses méthodes et les paramètres de manière à comprendre très facilement qu'elle est sont but et que représente les paramètres. Le code est donc bien plus lisible.

A l'aide de cette méthode, on évite la duplication de code. Puisque dans l'exemple on aurait pu vouloir calculer le prix d'un produit dans une autre méthode, avec le code au dessus on aurait dupliqué le code pour calculer le prix d'un produit.

L'extraction de méthode nous permet aussi d'isoler les parties indépendantes du code. Cela est pratique lorsque l'on cherche un potentiel bug, on peut plus facilement tester toutes les méthodes du codes car elles sont toutes isolées les une des autres.

Technique inverse : La technique de refactoring Inline Method est l'exact opposé de l'extraction de méthode. Cette technique consiste à supprimé les méthodes jugées inutile qui sont très courtes ou qui sont plus facilement compréhensible par le code à l'intérieure de la méthode que par son nom. Cela à pour seul but de simplifier le code en diminuant le nombre de méthode dans le code.

2.1.2 Extraction de variable

Présentation : La technique d'extraction de variable permet de créer des variables claires qui nous permet de rendre des expressions complexe plus compréhensible. Elle peut s'appliquer, par exemple, sur les conditions de if ou aussi des expressions arithmétiques sans résultats intermédiaires.

```
void maFonction(String state) {
    if(state.toUpperCase().indexOf("DONE") > -1){
        //do something
    }
}
```

Avant extraction de variable

Exemple : Au dessus, nous pouvons voir une fonction qui va faire quelque chose si l'état est done. Si l'on applique l'extraction de variable, on obtient une variable isDone qui contient un boolean. Cette méthode nous permet de passer d'un code qui contient une expression peu lisible à un code avec une variable explicite.

```
void maFonction(String state) {
    final boolean isDone = state.toUpperCase().indexOf("DONE") > -1;

    if(isDone){
        //do something
    }
}
```

Après extraction de variable

Bénéfices : Pour que le résultat de cette méthode soit optimal, il faut nommer efficacement les variables créées pour qu'elles soient le plus lisible possible. Cela va permettre de produire un code plus lisible et qui contiendra moins de long commentaire pour expliquer les longues expressions.

Inconvénient : Le code va contenir beaucoup de variable mais cela est dérisoire comparé à la lisibilité du code qui est nettement améliorée.

Technique inverse : La technique Inline Temp va permettre de supprimer les variables superflue qui contienne uniquement le résultat d'une opération simple qui va être utilisé une seul fois. Cette technique n'a pas de vrai bénéfice dans cette état, en revanche elle peut être couplé avec la technique suivante.

2.1.3 Remplacement des temporaires avec des méthodes

Présentation : Le remplacement des temporaires avec des méthodes va comme son nom l'indique, remplacer des variables temporaires par le résultat de méthode. On va extraire le code des variables temporaire avec la technique Inline method puis les placer dans des méthodes.

```java
boolean isAdditionSup10(int nbr1, int nbr2) {
    int addition = nbr1 + nbr2;
    if(addition > 10){
        return true;
    }else{
        return false;
    }
}
```

Avant remplacement du temporaire

Exemple : Au dessus, nous pouvons voir que l'addition des deux paramètre est stocké dans une variable temporaire. Après le refactoring, la variable temporaire disparait et on obtient une méthode qui va s'occuper de faire l'addition à la place.

```java
boolean isAdditionSup10(int nbr1, int nbr2) {
    if(addition(nbr1,nbr2) > 10){
        return true;
    }else{
        return false;
    }
}

int addition(int nbr1, int nbr2){
    return nbr1 + nbr2;
}
```

Après remplacement du temporaire

Bénéfices : Le code est plus compréhensible grâce au nom de la méthode qui est explicite.
Si j'ai besoin plus tard dans mon code de faire une addition, j'ai une méthode que je peux réutiliser.

2.1.4 Diviser les variables temporaires

Présentation : Parfois dans nos code, nous déclarons une variable temporaire où nous stockons un résultat quelconque. Puis plus tard, nous réutilisons cette même variable pour stocker un autre résultat n'ayant aucun rapport. Cette technique a pour but de ne plus utiliser la même variable temporaire pour faire différentes choses.

```java
void maFonction(int nbr1, int nbr2){
    int temp = nbr1 + nbr2;
    System.out.println("L'addition des deux valeurs vaut : " + temp);
    temp = nbr1 - nbr2;
    System.out.println("La soustraction des deux valeurs vaut : " + temp);
}
```

Avant division

Exemple : On peut voir au dessus que je réutilise la variable temp pour faire à la fois une addition et une soustraction. Après le refactoring on obtient deux variable proprement nommées addition et soustraction.

```
void maFonction(int nbr1, int nbr2){
    final int addition = nbr1 + nbr2;
    System.out.println("L'addition des deux valeurs vaut : " + addition);
    final int soustraction = nbr1 - nbr2;
    System.out.println("La soustraction des deux valeurs vaut : " + soustraction);
}
```

Après division

Bénéfices : Le fait d'avoir chaque variable,méthode ou n'importe quelle composant qui a un unique but permet de faciliter grandement la maintenance du code. Puisque on peut modifier des parties du code sans que ça affecte une autre partie.
Cela permet aussi une meilleure relecture du code car on supprime les variables nommées temp ou value pour donner des noms facilement compréhensible.

2.1.5 Remove Assignments to Parameters

s

2.1.6 Replace Method with Method Object

s

2.1.7 Substitute Algorithm

s

2.2 Déplacement des fonctionnalités entre les objets

s

2.2.1 Move Method

s

2.2.2 Move Field

s

2.2.3 Extract Class

s

2.2.4 Inline Class

s

2.2.5 Hide Delegate

s

2.2.6 Remove Middle Man

s

2.2.7 Introduce Foreign Method

s

2.2.8 Introduce Local Extension

s

2.3 Organiser ses données

s

2.3.1 Self Encapsulate Field

s

2.3.2 Replace Data Value with Object

s

2.3.3 Change Value to Reference

s

2.3.4 Change Reference to Value

s

2.3.5 Replace Array with Object

s

2.3.6 Duplicate Observed Data

s

2.3.7 Change Unidirectional Association to Bidirectional

s

2.3.8 Change Bidirectional Association to Unidirectional

s

2.3.9 Replace Magic Number with Symbolic Constant

s

2.3.10 Encapsulate Field

s

2.3.11 Encapsulate Collection

s

2.3.12 Replace Type Code with Class

s

2.3.13 Replace Type Code with Subclasses

s

2.3.14 Replace Type Code with State/Strategy

s

2.3.15 Replace Subclass with Fields

s

2.4 Simplifier les expressions de condition

s

2.4.1 Decompose Conditional

s

2.4.2 Consolidate Conditional Expression

s

2.4.3 Consolidate Duplicate and Conditional Fragments

s

2.4.4 Remove Control Flag

s

2.4.5 Replace Nested Conditional with Guard Clauses

s

2.4.6 Replace Conditional with Polymorphism

s

2.4.7 Introduce Null Object

s

2.4.8 Introduce Assertion

s

2.5 Simplifier les appels de méthode

s

2.5.1 Rename Method

s

2.5.2 Add Parameter

s

2.5.3 Remove Parameter

s

2.5.4 Separate Query from Modifier

s

2.5.5 Parameterize Method

s

2.5.6 Replace Parameter with Explicit Methods

s

2.5.7 Preserve Whole Object

s

2.5.8 Replace Parameter with Method Call

s

2.5.9 Introduce Parameter Object

s

2.5.10 Remove Setting Method

s

2.5.11 Hide Method

s

2.5.12 Replace Constructor with Factory Method

s

2.5.13 Replace Error Code with Exception

s

2.5.14 Replace Exception with Test

s

2.6 Faire face à la généralisation

s

2.6.1 Pull Up Field

s

2.6.2 Pull Up Method

s

2.6.3 Pull Up Constructor Body

s

2.6.4 Push Down Method

s

2.6.5 Push Down Field

s

2.6.6 Extract Subclass

s

2.6.7 Extract Superclass

s

2.6.8 Extract Interface

s

2.6.9 Collapse Hierarchy

s

2.6.10 Form Template Method

s

2.6.11 Replace Inheritance with Delegation

s

2.6.12 Replace Delegation with Inheritance

s

Bibliographie

[1] Refactoring techniques. https ://sourcemaking.com/refactoring/refactorings.

[2] Jean-Michel Doudoux. Le refactoring, Janvier 2007. https ://www.jmdoudoux.fr/java/dejae/chap009.htm.

[3] Romain Fallet. Le désenchantement du logiciel, Septembre 2018. https ://blog.romainfallet.fr/desenchantement-logiciel/.

[4] Martin Fowler. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[5] Zoran Maksimovic. Refactoring with composed method pattern, Janvier 2019. https ://www.agile-code.com/blog/refactoring-with-composed-method-pattern/.

[6] Jean-Philippe Retaillé. *Refactoring des applications JAVA/J2EE*. Eyrolles, 2005.