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 :
Mcf. 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 les outils de refactoring existant ainsi que les graphes de flot de contrôle.
Enfin on étudiera s'il est possible de faire du refactoring à l'aide des 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

Les différents type de refactoring

2.1 Pattern de méthode composée

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 ce type 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.

```
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.

```
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.

```
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 notre code, nous déclarons une variable temporaire où nous stockons un résultat quelconque. Puis plus tard, nous réutilisons cette même variable pour stocker un tout autre résultat n'ayant aucun rapport. Cette technique a pour but de ne plus utiliser la même variable temporaire pour faire différentes choses et de nommer proprement chaque variable temporaire.

```
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 ou une responsabilité 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 Supprimer les assignations aux paramètres

Présentation : Ici le problème est semblable à celui de la division des variables temporaires, si on a un paramètre, on ne doit pas lui affecter d'autre valeur car cela modifie ce que représente le paramètre et on peut se perdre dans le code car on ne sait plus ce que contient ce paramètre.
Il vaut donc mieux déclarer une variable local plutôt que de modifier le paramètre.

Bénéfices : Comme dit plus haut, chaque éléments à une unique responsabilité et la maintenance du code est plus simple.

2.1.6 Remplacement de méthode par des objets

Présentation : Il nous arrive d'écrire des méthodes très longues qui possèdent beaucoup de variable qui sont extrêmement liées entre elles. On peut avec le refactoring créer une nouvelle classe qui va contenir la méthode ainsi que les anciennes variables locales en variable de classe.

```
class Order {
  //...
  public double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // long computation.
    //...
  }
}
```

Avant remplacement par l'objet [1]

```
class Order {
  //...
  public double price() {
    return new PriceCalculator(this).compute();
  }
}

class PriceCalculator {
  private double primaryBasePrice;
  private double secondaryBasePrice;
  private double tertiaryBasePrice;

  public PriceCalculator(Order order) {
    // copy relevant information from order object.
    //...
  }

  public double compute() {
    // long computation.
    //...
  }
}
```

Après remplacement par l'objet [1]

Bénéfices : L'isolation d'une méthode longue dans sa propre classe permet d'empêcher le fait prenne de l'ampleur.
On peut aussi scinder cette méthode en sous-méthode sans "polluer" la classe d'origine avec des méthodes utilitaires.

Inconvénient : Cette méthode de refactoring nous fait créer une nouvelle classe et cela augmente la complexité global du programme.

2.2 Déplacement des fonctionnalités entre les objets

Lorsque l'on code beaucoup de fonctionnalités dans un même programme, il arrive qu'au bout d'un moment, on se rendent compte que , par exemple, on a placé une fonctionnalité dans la mauvaise classe et qu'on ne sache pas comment faire pour la déplacer sans risquer de casser tout le code.
Or les méthodes de refactoring qui vont suivre seront du type à permettre le déplacement de fonctionnalités entre des classes en toute sécurité.

2.2.1 Déplacement de méthode

Présentation : Il se peut qu'à un instant t, vous vouliez déplacer une méthode dans un autre classe car cela vous arrange ou car cela aurait plus de sens. Dans ce cas la, on peut utiliser le refactoring pour déplacer cette méthode vers l'autre classe en toute sécurité.

Comment faire : Il faut commencer par regarder toutes les variables qui sont présente dans la classe et que la méthode utilise. Si c'est variable ne sont utilisé que par la méthode à déplacer, il faut les déplacer dans la nouvelle classe aussi.
En revanche si ces variables sont utilisées par d'autre méthode il est conseillé de déplacer aussi ces méthodes vers la nouvelle classe. Ensuite, il faut s'assurer que toutes ces méthodes ne soit pas déclarer dans une classe mère ou fille.
Si toutes ces conditions sont réunis, on peut déclarer la ou les méthodes dans la nouvelle classe puis définir une méthode dans l'ancienne classe qui va appeler la nouvelle méthode.

Bénéfices : Cette méthode de refactoring nous permet d'obtenir une meilleure cohérence interne dans les classes.

2.2.2 Déplacement de variable

s

2.2.3 Extraction de classe

s

2.2.4 Cacher la délégation

s

2.2.5 Ajout de méthode étrangère

s

2.3 Organiser ses données

Ce type de refactoring va permettre de mieux gérer les données des classes en dissociant les associations de classe pour les rendre plus portable et réutilisables.

2.3.1 Auto Encapsulation des champs

s

2.3.2 Remplacer les données par des objets

s

2.3.3 Passer du stockage par valeur en référence

s

2.3.4 Remplacer les tableaux par des objets

s

2.3.5 Changer une association unidirectionnelle en bidirectionnelle

s

2.3.6 Remplacer les numéro magiques en constantes

s

2.3.7 Encapsulation des champs

s

2.3.8 Encapsulation de collection

s

2.3.9 Remplacer les type code par une classe

s

2.3.10 Remplacer des sous-classes par des champs

s

2.4 Simplifier les expressions conditionnels

Dans les programmes, il n'est pas rare de trouver des expressions conditionnels qui possèdent une logique complexe avec beaucoup de condition. Ce type de refactoring va permettre de simplifier toutes ces expressions.

2.4.1 Décomposer les expressions conditionnels

s

2.4.2 Consolider les expressions conditionnels

s

2.4.3 Consolider les duplications des fragments conditionnels

s

2.4.4 Remplacer les conditions imbriquées par des clauses de garde

s

2.4.5 Remplacer les conditions imbriquées avec du polymorphisme

s

2.5 Simplifier les appels de méthode

Ces techniques de refactoring ont pour but de simplifier l'appel de méthode ainsi que de rendre l'appel de méthode plus facile à comprendre.

2.5.1 Renommer une méthode

s

2.5.2 Ajout de paramètre

s

2.5.3 Suppression de paramètre

s

2.5.4 Séparer les fonctionnalités d'une même méthode

s

2.5.5 Méthode de paramétrage

s

2.5.6 Remplacer les paramètres avec des méthodes explicites

s

2.5.7 Conserver l'objet entier

s

2.5.8 Remplacer les paramètres avec des appels de méthodes

s

2.5.9 Ajouter des objets en paramètre

s

2.5.10 Cacher les méthodes

s

2.5.11 Remplacer les codes d'erreurs par des exceptions

s

2.5.12 Remplacer les exceptions par des tests

s

2.6 Faire face à la généralisation

Dans cette partie, nous allons étudier des techniques permettant de déplacer des fonctionnalités dans la hiérarchie d'héritage de classes, à la création de nouvelles classes et d'interfaces.

2.6.1 Remonter les champs

s

2.6.2 Remonter les méthodes

s

2.6.3 Remonter le corps du constructeur

s

2.6.4 Extraire une sous classe

s

2.6.5 Extraire une super classe

s

2.6.6 Extraire une interface

s

2.6.7 Supprimer une sous classe

s

2.6.8 Créer des template méthodes

s

Chapitre 3

Outils de refactoring

3.1 Fonctionnalités d'IDE

Dans cette partie, je vais vous présenter les fonctionnalités de refactoring que l'IDE Eclipse propose à ces utilisateurs.

3.2 AutoRefactor

3.2.1 Présentation du projet

Le projet AutoRefactor a été lancé par Jean-Noël Rouvignac. Il a décidé de se lancer dans se projet car il était fatigué de devoir prendre trop de temps pour appliquer les mêmes nettoyages de code encore et encore. L'objectif de ce projet est de faciliter la maintenance, moderniser le code, rendre le code plus léger et compacte et augmenter les performances des programmes. Il a commencé à travailler sur les expressions rationnelles pour retravailler toute la base du code, mais les faux positifs étaient trop nombreux.[7] Il a ensuite créé un greffon Eclipse (AutoRefactor) qui utilise l'API des Java Development Tools (Eclipse JDT) qui est l'API que eclipse utilise pour faire du refactoring.
Une première release du produit est sortie le 22 mars 2015 et plus récemment la version 1.2 est sortie (30 juin 2018).

3.2.2 Fonctionnement du programme

Je vais maintenant vous présenter un algorithme simplifié de comment fonctionne AutoRefactor.

Le développeur choisi les règles de refactoring à appliquer puis le greffon prend la liste des refactorings.

1) Le fichier Java à analyser est parsé et produit un arbre syntaxique abstrait.

2) Pour chaque refactoring :
    1) Cherche des opportunités de refactoring en visitant l'abre syntaxique abstrait.
    2) Génère les réécritures de code lorsqu'une opportunité de refactoring a été identifiée.

3) Lorsque tout l'arbre syntaxique abstrait a été visité, si des réécriture de code ont été générées :
    1) Alors, toutes les réécritures de code générées sont appliquées sur le fichier.
    2) Le fichier est sauvegardé.
    3) Boucle vers 1.
    4) Sinon, fin : il n'y a plus de refactoring possibles sur ce fichier.

Actuellement, tous les refactorings implémentés font du filtrage par motif et travaillent fichiers par fichiers.

3.2.3 Fonctionnalités

s

3.2.4 Objectif futur

Jean-Noël Rouvignac a pour objectif futur de réussir à construire des graphes de flot de contrôle pour pouvoir les analyser. Ceci lui permettrait d'écrire des refactorings comprenant les chemins d'exécution du code, comme le ferait un développeur qui lirait du code. En particulier, il deviendrait possible de réduire la portée des variables, comprendre quels chemins d'exécution du code sont morts (impossibles à atteindre).[7]
Il aimerait développer une fonctionnalité d'extraction automatique de méthode pour simplifier les longues méthodes.

3.3 Les graphes de flots de contrôle

Actuellement, les flots de contrôle ne sont pas utilisés pour faire du refactoring.

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.

[7] Jean-Noël Rouvignac. https ://github.com/JnRouvignac/AutoRefactor.