Chapitre 7 : Refonte de code (refactoring)

« Si déboguer c'est supprimer des bugs, alors programmer ne peut être que de les ajouter. » ***Edsger Dijkstra***

1. Objectif du chapitre

Ce chapitre présente la refonte de code (*code refactoring*), qui vise à améliorer sans cesse la qualité du code, et souligne son influence sur la façon de coder.
À l'issue de ce chapitre :

- Vous serez en mesure de lutter contre le pourrissement du code (*rotten code*).
- Vous saurez refondre du « code pourri » (*rotten code*).

2. Nécessité de la refonte de code

2.1. Qu'est-ce que la refonte de code ?

Définition : La refonte de code est le processus qui consiste à changer le code d'un système logiciel de telle manière qu'il n'altère pas le comportement extérieur du code, tout en améliorant sa structure interne.

- Elle va de pair avec la notion de « code propre ».
- Sa mise en œuvre nécessite :
  - l'utilisation d'environnements de développement appropriés (cf. menu « *refactor* » de IntelliJ).
  - la présence de tests exécutables qui vont permettre de vérifier que l'on n'introduit pas de régression (voir le chapitre suivant).

2.2. Qu'est-ce que n'est pas la refonte de code ?

La refonte de code ne correspond pas à l'introduction de nouvelles fonctionnalités. Elle ne correspond pas, non plus, à l'optimisation des performances qui conduit souvent à un code qui devient difficile à comprendre.

2.3. Pourquoi faire de la refonte de code ?

- Pour ne pas accumuler de **dette technique**.
- Pour améliorer la logique de conception du logiciel.
- Pour rendre le système plus simple à comprendre et donc à maintenir, étendre et vérifier. C'est le principe KISS : « Keep It Simple, Stupid ». Attention, faire simple, c'est compliqué et relève généralement d'un processus d'affinage. Plusieurs refontes peuvent être nécessaires avant d'atteindre la simplicité.
- Pour aider à trouver les bugs de type Schrödinbug (voir chapitre suivant).

Remarque : L'amélioration du code correspond principalement à la suppression de code : « *La perfection n'est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à enlever.* » Antoine de Saint-Exupéry.

2.4. Quand faire de la refonte de code ? Une nouvelle façon de coder

La refonte de code s'inscrit pleinement dans l'approche Agile. Elle intervient au moment de la création et au moment de la maintenance du code. Elle profite alors de la malléabilité du code.

- **Lors du développement**. Cela constitue une nouvelle façon de coder :
    1. Codage « Quick and Dirty » : coder rapidement la fonctionnalité (code « sale ») en restant focalisé sur le développement de la fonctionnalité.
    2. Coder les tests et vérifier que le code passe ces tests.
    3. Refondre le code pour le rendre propre en s'assurant que les tests passent encore et éviter ainsi la régression de code.
- **Lors de la reprise du code** pour ajouter une nouvelle fonctionnalité ou corriger un bug. Profiter de cette reprise de code pour améliorer sa structure en application de la règle des boy-scouts : « Laissez le campement plus propre que vous l'avez trouvé en arrivant ».

2.5. Quelques indices de la nécessité d'une refonte : « *bad smells in the code* »

- Duplication de code.
- Longues méthodes.
- Grandes classes.
- Longue liste de paramètres.
- Nécessité de commenter le code.
- …

3. Exemples de refonte de code

Voici une sélection de cas de refonte de code extraite du catalogue de Martin Fowler (*http://refactoring.com/catalog/index.html*).

3.1. Factoriser le code redondant

Vous voyez la même structure de code à plus d'un endroit et qui porte la même sémantique. C'est l'application du principe DRY (*Don't Repeat Yourself)*. La duplication de code posera un problème de maintenance plus tard. Quand il s'agira de corriger un bug dans la partie dupliquée ou de la faire évoluer, il n'y a plus rien d'automatique qui rappellera qu'il faudra faire les mêmes modifications au code dupliqué.

- Rassemblez le code commun dans une méthode à part entière.

Avant :

```
int total(int a, int b) {
    return a + b;     // Duplication
}
double average(int a, int b) {
    int sum = a + b;  // Duplication
    return sum / 2;
}
```

Après :

```
int total(int a, int b) {
    return a + b;
```

```
}
double average(int a, int b) {
    int sum = total(a, b);
    return sum / 2;
}
```

Après la refonte du code, si l'on doit ajouter du code pour traiter le cas du dépassement de la valeur limite des entiers, les deux occurrences profitent de cet ajout automatiquement.

3.2. Remplacer un « nombre magique » par une constante symbolique

Vous avez un nombre avec un sens particulier (*magic number*).

- Créez une constante, nommez-la en fonction de son rôle, et remplacez le nombre par cette constante.

Avant :

```
double potentialEnergy( double mass, double height ) {
    return mass * 9.81 * height;
}
```

Après :

```
static final double GRAVITATIONAL_CONSTANT = 9.81;
double potentialEnergy( double mass, double height ) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
```

3.3. Supprimer les doubles négations

Vous avez une condition avec une double négation.

- Rendez cette condition positive.

Avant :

```
if (!item.isNotFound()) { }
```

Après :

```
if (item.isFound()) { }
```

3.4. Remplacer une méthode par un objet méthode

Vous avez une longue méthode qui utilise des variables locales.

- Transformez la méthode en une classe interne de telle manière que les variables locales deviennent des attributs de cette classe puis décomposez-la en sous-méthodes privées (alias *abstraction hypostatique*).

Avant :

```
public final class Order {
  float price(int a, Obj o) {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // long code
```

```
  }
}
```

Après :

```
public final class Order {
  float price() {
    return new PriceCalculator.compute();
  }
  private final class PriceCalculator {
    private double _primaryBasePrice;
    private double _secondaryBasePrice;
    private double _tertiaryBasePrice;

    protected static float compute() {
    // long code qui utilise en autre la méthode method1, method2...
    }

    private int method1() {
    ... // utilise les attributs
    }
    ...
  }
}
```

3.5. Introduire une variable explicative

Vous avez une expression conditionnelle difficilement compréhensible.

- Mettez le résultat de l'expression, ou une partie de cette expression, dans une variable temporaire avec un nom qui explicite son rôle.

Avant :

```
if (platform.indexOf("MAC") > -1
    && browser.indexOf("IE") > -1
    && wasInitialized() && resize > 0) {
  // code
}
```

Après :

```
boolean isMacOs = platform.indexOf("MAC") > -1;
final boolean isIEBrowser = browser.indexOf("IE") > -1;
final boolean wasResized = resize > 0;
if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
  // code
}
```

3.6. Préserver un objet entier

Vous utilisez plusieurs valeurs d'un objet et passez ces valeurs comme paramètres d'une méthode.

- Envoyez l'objet en entier.

Avant :

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

Après :

```
withinPlan = plan.withinRange(daysTempRange());
```

3.7. Remplacer les conditions par le polymorphisme

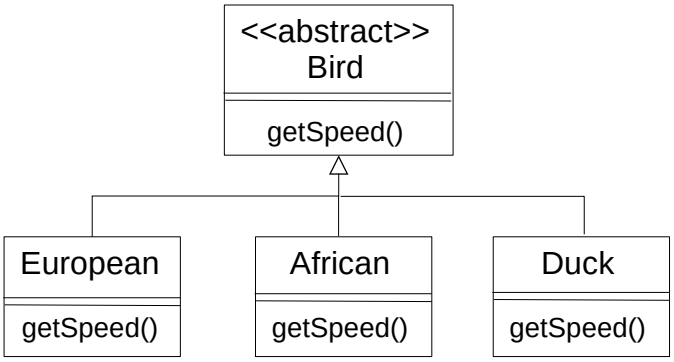
Vous avez une expression conditionnelle qui sélectionne différents comportements en fonction du type de l'objet.

- Créez des sous-classes et déplacez chaque partie de l'expression conditionnelle dans une méthode redéfinie dans une sous-classe puis utilisez le polymorphisme.

Avant :

```
double getSpeed() {
  switch (_type) {
    case EUROPEAN: return getBaseSpeed();
    case AFRICAN: return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case DUCK: return (_isWood) ? 0 : getBaseSpeed();
  }
}
```

Après :



3.8. Remplacer un code d'erreur par une exception

Vous avez une méthode qui retourne un code spécial pour indiquer une erreur.

- Levez une exception à la place.

Avant :

```
int withdraw( int amount ) {
  if (amount > _balance) {
    return -1; // Cas erreur
  }
  _balance -= amount;
  return 0;
}
```

Après :

```
void withdraw( int amount ) throws BalanceException {
  if (amount > _balance) {
    throw new BalanceException();
  }
  _balance -= amount;
}
```

3.9. Remplacer un paramètre par une méthode

Vous avez un objet qui appelle une méthode, et passe le résultat comme paramètre d'une autre méthode.

- Supprimez le paramètre et laissez le receveur appeler la méthode.

Avant :

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

Après :

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice(basePrice);
// getDiscountLevel() est appelée à l'intérieur de discountedPrice()
```

3.10. Introduire un objet paramètre

Vous avez un groupe de paramètres qui vont naturellement ensemble.

- Remplacez les paramètres par un objet qui les rassemble.

Avant :

```
amountInvoicedIn(start: Date, end: Date)
amountReceivedIn(start: Date, end: Date)
amountOverdueIn(start: Date, end: Date)
```

Après :

```
amountInvoicedIn(DateRange)
amountReceivedIn(DateRange)
amountOverdueIn(DateRange)
```

4. Que retenir de ce chapitre ?

- Le code logiciel est malléable. Il faut en profiter pour faire une amélioration continue de la propreté et de la qualité du code.
- La refonte de code permet d'améliorer la structure interne du code.
- La refonte s'inscrit après le codage d'une fonctionnalité et au moment de la reprise de code (correction de bug, ajout d'une fonctionnalité).
- Elle ne peut pas se faire sereinement sans tests dynamiques qui permettent de se prémunir contre la régression. C'est l'objet du chapitre suivant.

Savoir quand et comment faire de la refonte de code est un art et une quête perpétuelle.

Le travers du « geek » est le « *refuctoring* » qui est le processus qui consiste à prendre un morceau de code

bien conçu et grâce à une série de petites modifications irréversibles, à le rendre impossible à maintenir par quiconque sauf par lui-même.

5. Lectures

- Martin Fouler, Refactoring « *Improving the Design of Existing Code* », Addison-Wesley Professional, 1999.
- Consultez la liste des cas de refonte de code :
  - Martin Fowler, « *Refactoring Home Page* », http://refactoring.com/catalog/index.html