Environnements de développement (intégrés)

JDT (refactorisation)

Patrick Labatut
labatut@di.ens.fr
http://www.di.ens.fr/~labatut/

Département d'informatique
École normale supérieure

Centre d'enseignement et de recherche en technologies de l'information et systèmes
École des ponts

Transparents inspirés de ceux du cours de Mihaela Sighireanu en 2006-2007.

Plan

1. Rappels
2. Refactorisation
   - Introduction
   - Origine
   - Différence avec un simple « nettoyage »
   - Nomenclature
   - Refactorisation dans Eclipse
   - Démonstration

Rappels

Édition :

- *Toggle Comment* : *Ctrl-/*.

Mise en forme du code :

- *Correct Indentation* : *Ctrl-I*,
- *Format* : *Shift-Ctrl-F*.

Aide :

- *Word Completion* : *Alt-/*,
- *Quick Fix* : *Ctrl-1*,
- *Content Assist* : *Ctrl-Space*.

Navigation :

- Tâches automatiques : `FIXME`, `TODO`, `XXX` en commentaire (+ description).

Introduction 1/2

Définition

Refactoriser[1] un/plusieurs module(s) source(s) consiste à modifier sa structure afin d'en améliorer sa lisibilité sans toutefois affecter son comportement externe.

Il s'agit d'une partie intégrante du cycle de développement logiciel :

1. Spécification
2. Écriture du code
3. Écriture des tests
4. Débogage
5. Besoin d'ajouter de nouvelles fonctionnalités / Difficulté à assurer la maintenance
6. Refactorisation

[1] *Refactor*, en anglais

Introduction 2/2

Attention

La refactorisation ne doit pas :

- corriger des bugs existants (ni en ajouter de nouveaux. . .),
- ajouter de nouvelle fonctionnalité.

Objectif

Améliore la « compréhensibilité » du code : en changeant la structure[2] et en éliminant le code mort, la refactorisation facilite la maintenance future, et rend possible des extensions inenvisageables auparavant.

[2] *design*, en anglais.

Origine

La refactorisation existait depuis longtemps de manière « informelle » mais la première étude apparaît dans un article de William F. Opdyke en 1990 : *Refactoring : An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems* (puis sa thèse en 1993).

L'expression « refactorisation » vient de l'analogie simple avec la factorisation de polynômes, qui est une réécriture qui dévoile la structure interne d'un polynôme (ses racines) :

$$\begin{aligned} P(X) &= X^2 - 3X + 2 \\ &= (X-1)(X-2) \end{aligned}$$

En quoi est-ce différent d'un simple « nettoyage » ?

La refactorisation est une étape est bien séparée du débogage.

Comment assurer l'invariance du comportement visible malgré le changement de structure ?

- Combiner une suite d'opérations élémentaires qui chacune ont peu de chance d'altérer la sémantique,
- Lancer les tests unitaires (idéalement) après chaque opération.

Il existe une nomenclature[3] des opérations élémentaires de refactorisation à réaliser sur le code source.

[3] une liste à peu près exhaustive est disponible à l'adresse : `http://www.refactoring.com/catalog/` et dans le livre de référence *Refactoring. Improving the Design of Existing Code* de Martin Fowler.

Nomenclature

Quelques-unes des principales opérations élementaires de refactorisation :

- *Encapsulate Field* : rendre un champ privé et fournir des accesseurs (et éventuellement des affecteurs),
- *Extract Method* : transformer un morceau de méthode en une méthode à part entière (avec un nom explicite),
- *Inline* : l'inverse d'*Extract Method*,
- *Extract Interface* : deux classes ont un ensemble de méthodes en commun ou bien plusieurs utilisateurs d'une classes ne se servent que du même sous-ensemble de méthodes de cette classe,
- *Pull Up (Method/Field)* : mettre en commun des méthodes qui renvoient le même résultat dans deux classes filles,
- *Pull Down (Method/Field)* : une méthode/un champ d'une classe n'est pertinent que pour l'une des classes filles,
- *Rename Class/Method/Field* : rendre plus clair le rôle d'une classe/méthode/champ.

Attention : il n'est pas toujours évident de choisir l'opération à réaliser (d'où l'existence d'opérations « inverses » dans la nomenclature).

Exemple : *Encapsulate Field* - Avant refactorisation 1/4

```
1  class Vector2f {
2      public float x, y;
3  }
4
5      /* [...] */
6
7      Vector2f dv = new Vector2f();
8      dv.x = 1.5f;
9      dv.y = -0.5f;
10
11     Vector2f v = new Vector2f();
12     v.x = 0.0f;
13     v.y = 1.0f;
14
15     v.x += dv.x;
16     v.y += dv.y;
```

Exemple : *Encapsulate Field* - Après refactorisation 2/4

```
1  class Vector2f {
2      private float x, y;
3
4      Vector2f(float x, float y) {
5          this.x = x;
6          this.y = y;
7      }
8
9      void add(Vector2f v) {
10         x += v.x;
11         y += v.y;
12     }
13 }
14
15     /* [...] */
16
17     Vector2f dv = new Vector2f(1.5f, -0.5f);
18
19     Vector2f v = new Vector2f(0.0f, 1.0f);
20
21     v.add(dv);
```

Exemple : *Encapsulate Field* - Avant refactorisation 3/4

```
1  class GrowableArray {
2      private static final int INITIAL_CAPACITY = 4;
3      public int[] data;
4      public int size;
5      public GrowableArray() {
6          data = new int[INITIAL_CAPACITY];
7          size = 0;
8      }
9     public void addElement(int i) {
10         if (size == data.length) {
11             final int newCapacity = 2 * data.length;
12             int[] t = new int[newCapacity];
13             System.datacopy(data, 0, t, 0, data.length);
14         }
15         data[size] = i;
16         size++;
17     }
18 }
19 public static void main(String[] args) {
20     GrowableArray myArray = new GrowableArray();
21     myArray.addElement(1);
22     myArray.addElement(2);
23     myArray.addElement(3);
24     final int i = myArray.data[1];
25  // myArray.data = null; /* Oops! */
26  // myArray.size = 0; /* Oops! */
27 }
```

Exemple : *Encapsulate Field* - Après refactorisation 4/4

```
1  class GrowableArray {
2      private static final int INITIAL_CAPACITY = 4;
3      private int[] data;
4      private int size;
5      public GrowableArray() {
6          data = new int[INITIAL_CAPACITY];
7          size = 0;
8      }
9      public void clear() {
10         size = 0;
11     }
12     public void addElement(int i) {
13         if (size == data.length) {
14             final int newCapacity = 2 * data.length;
15             int[] t = new int[newCapacity];
16             System.datacopy(data, 0, t, 0, data.length);
17         }
18         data[size] = i;
19         size++;
20     }
21     public int elementAt(int index) {
22         return data[index];
23     }
24 }
25 public static void main(String[] args) {
26     GrowableArray myArray = new GrowableArray();
27     myArray.addElement(1);
28     myArray.addElement(2);
29     myArray.addElement(3);
30     final int i = myArray.elementAt(1);
31  // myArray.data = null; /* Impossible */
32  // myArray.size = 0; /* Impossible */
33 }
```

Refactorisation dans Eclipse

La plupart des EDI (Eclipse, Visual Studio, NetBeans, Delphi) modernes proposent des outils implémentant les opérations de refactorisation. Ces outils mettent à jour (semi-)automatiquement références, importations, utilisations de classe, méthode... pour s'efforcer de conserver un code (compilable et) équivalent à l'original.

Dans Eclipse :

1. *Source / Clean Up...* : plutôt « nettoyage » de code,
2. Tout le menu *Refactor* du JDT... (propose un assistant).

Démonstration