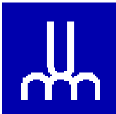
Refactorings

Saliha Bouden

IFT6251

Houari Sarhaoui

2004/02/26

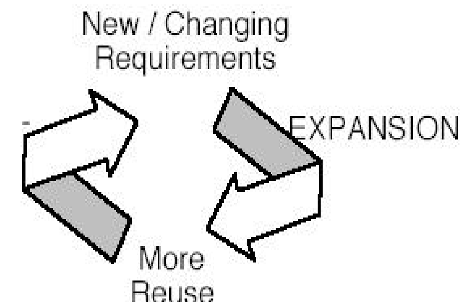


Département d'informatique et de recherche opérationnelle
Université de Montréal

- Introduction
- C'est quoi un refactoring ?
- Pourquoi est-ce nécessaire ?
- Quand doit-on faire un refactoring ?
- Catégories de refactorings
- Conclusion
- Références

Introduction

- Trois phases dans le développement itératif [Foo95]
  - Développement initial en utilisant le prototypage rapide et des changements incrémentaux
  - Expansion
    - Ajouter de nouvelles fonctionnalités
    - Consolidation
  - Restructurer le logiciel
    - Introduire des patrons de conception
    - Utiliser des refactorings



- Un refactoring est une transformation logicielle qui préserve le comportement externe du logiciel et améliore la structure interne du logiciel

- De la notion de factorisation dans les mathématiques
- Ward Cunningham et Kent Beck avec Smalltalk (1980)
- Ralph Johnson et Bill Opdyke (1990) : « the importance of refactoring in the design »
- John Brant et Don Roberts ont construit le Refactoring Browser pour Smalltalk [RefBr]
- Adopté dans le développement « eXtreme Programming » [Bec00]

Définitions

- De [Fow99]
  - (Noun) a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
  - (Verb) to restructure software by applying a series of refactorings without changing its observable behaviour

- De [Rob98]
  - – A behaviour-preserving source-to-source program transformation
- De [Bec99]
  - – A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability

- Pour améliorer la conception du logiciel
  - Pour réduire
    - L'affaiblissement de code (*software decay*)
    - La complexité du logiciel
    - Les coûts d'entretient du logiciel
  - Pour augmenter
    - La compréhensibilité du logiciel
  - Pour introduire des patrons de conception
  - Pour faciliter les futurs changements

- Seulement si on pense que c'est nécessaire
  - Pas sur une base périodique
  - Appliquer la règle de trois (Don Roberts) [Fow99]
    - Première fois : implanter à partir de zéro
    - Deuxième fois : implanter quelque chose de semblable en dupliquant le code
    - Troisième fois : ne pas re-implanter, factoriser !

- Avant d'ajouter une nouvelle fonctionnalité
  - Particulièrement si la fonctionnalité est difficile à intégrer avec le code existant
- Pendant la réparation des erreurs
  - Si il est très difficile de tracer une erreur, factoriser d'abord pour rendre le code plus compréhensible
- Pendant les revues de code

Quand doit-on faire le refactoring?

- Identifier les « mauvaises odeurs » (*bad smells*) dans le code source [Bec99]
  - « Structures in the code that suggest (sometimes scream for) the possibility of refactoring »
  - « Code that can make the design harder to change »
- Exemples
  - Duplicated Code
  - Long Method
  - Large Class (too many responsibilities)
  - …

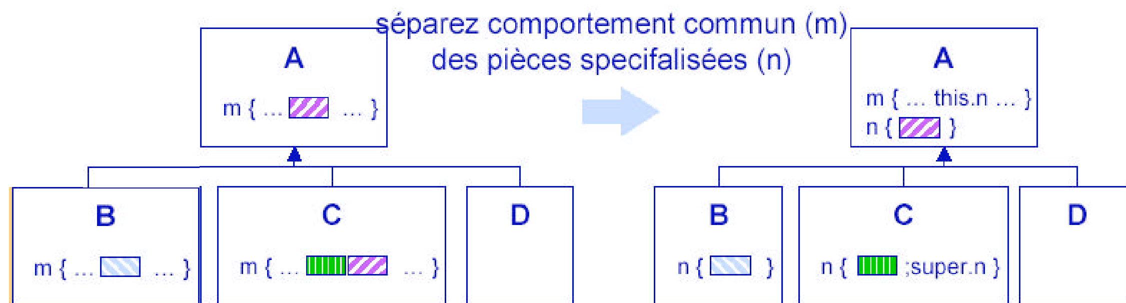
- Selon le langage de programmation
  - Spécifique à un langage (Java, Smalltalk...)
  - Indépendant du langage
- Selon le degré de formalité
  - Formel
  - Non formel [Fow99]
  - Semi-formel
- Selon le degré d'automatisation
  - Entièrement automatisé [Moo96]
  - Entièrement manuel [Fow99]

Catégories de refactorings

- Trois catégories de refactorings se produisant fréquemment dans les logiciels orientés objets
  - Création des méthodes « template »
  - Optimisation des hiérarchies de classes
  - Introduction de relations de composition

Catégories de refactoring

- Création des méthodes « template »
  - Diviser les méthodes en plus petites pièces pour séparer le comportement commun des pièces spécialisées
  - Utilisez l'héritage pour redéfinir ces méthodes



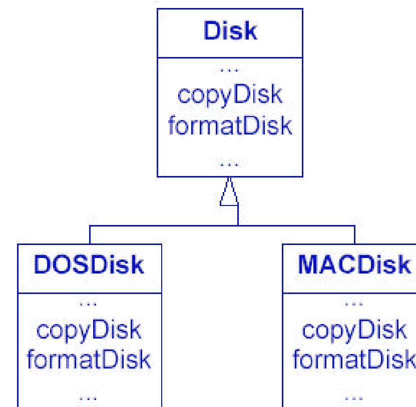
Catégories de refactoring

- Optimisation des hiérarchies de classe
  - Insérer ou enlever des classes dans une hiérarchie et redistribuer les fonctionnalités
  - Utilisée pour augmenter la cohésion et diminuer le couplage
    - Refactoring pour spécialiser
    - Refactoring pour généraliser

Catégories de refactoring

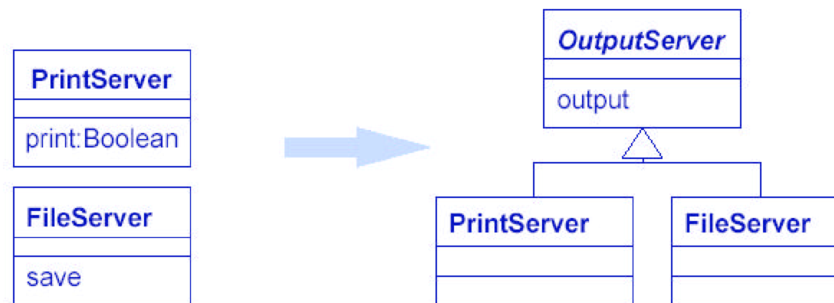
- Refactoring pour spécialiser
  - Améliorer la conception en décomposant une grande classe complexe en plusieurs petites classes

```
formatDisk
  self diskType = #MSDOS
    ifTrue: [ .. code1 ..].
  self diskType = #MAC
    ifTrue: [ .. code2 ..].
```



Catégories de refactoring

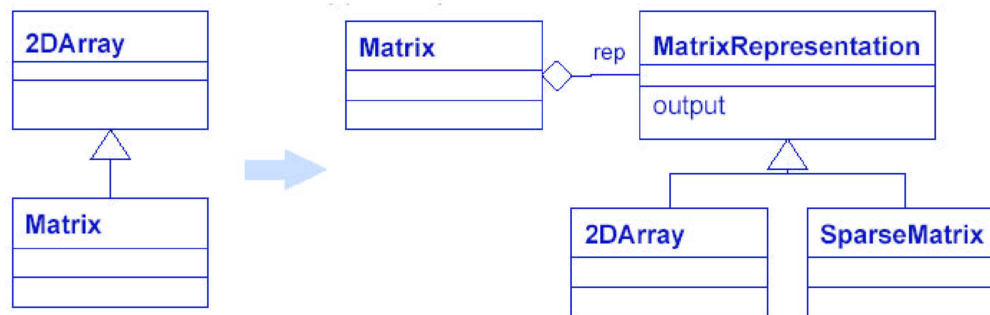
- Refactoring pour généraliser
  - Identifier des abstractions appropriées (par exemple des classes abstraites) en examinant des exemples concrets



- Refactoring pour généraliser
  - Étapes pour créer une super-classe abstraite
    - Créer une super-classe commune
    - Rendre compatibles les signatures des méthodes
    - Ajouter les signatures des méthodes à la super-classe
    - Rendre les corps des méthodes compatibles
    - Rendre les variables d'instance compatibles
    - Déplacer les variables communs à la super-classe
    - Déplacer le code commun à la super-classe
    - Ajouter la nouvelle fonctionnalité aux autres classes
  - Utiliser pour réduire le couplage

Catégories de refactoring

- Introduction des relations de composition
  – Convertir l'héritage en agrégation quand l'héritage est inexactement utilisé



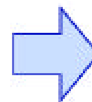
Exemples de refactorings

- De [Fowler99]
  - Refactoring primitifs
    - Composing methods (9 refactorings)
    - Moving features between objects (8 refactorings)
    - Organising data (16 refactorings)
    - Simplifying conditional expressions (8 refactorings)
    - Dealing with generalisation (12 refactorings)
    - Simplifying method calls (15 refactorings)
  - Refactorings composés
    - Tease apart inheritance
    - Extract hierarchy
    - Convert procedural design to objects
    - Separate domain from presentation

Refactorings primitifs

- Encapsulate Field [Fow99]
  - – There is a public field make it private and provide accessors

```
public class Node {
  public String name;
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
    nextNode.accept(p); }
  }
```



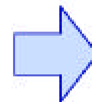
```
public class Node {
  private String name;
  private Node nextNode;
  public String getName() {
    return this.name; }
  public void setName(String s) {
    this.name = s; }
  public Node getNextNode() {
    return this.nextNode; }
  public void setNextNode(Node n) {
    this.nextNode = n; }
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
    this.getNextNode().accept(p); }
  }
```

Refactorings primitifs

- Consolidate Conditional Expression [Fow99]
  – You have a sequence of conditional tests with the same result, combine them into a single conditional expression and extract it

```
public class ASCIIPrinter extends Printer
{ ...

  public void accept(Packet p) {
    if p.getAddressee() == this
      if this.isASCII(p.getContents())
        this.print(p);
      else
        super.accept(p); }
}
```



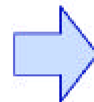
```
public class ASCIIPrinter extends Printer
{ ...

  public void accept(Packet p) {
    if ((p.getAddressee() == this) &&
        (this.isASCII(p.getContents())))
      this.print(p);
    else
      super.accept(p); }
}
```

Refactorings primitifs

- Extract Method [Fow99]
  - You have a code fragment that can be grouped together, turn the fragment into a method whose name explains the purpose of the method

```
public class ASCIIPrinter extends Printer
{ ...
  public void accept(Packet p) {
    if ((p.getAddressee() == this) &&
        (this.isASCII(p.getContents())))
      this.print(p);
    else
      super.accept(p); }
}
```

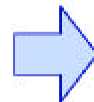


```
public class ASCIIPrinter extends Printer
{ ...
    public void accept(Packet p) {
    if this.isDestFor(p)
      this.print(p);
    else
      super.accept(p); }
  public boolean isDestFor(Packet p) {
    return ((p.getAddressee() == this) &&
           (this.isASCII(p.getContents())));
}}
```

Refactorings primitifs

- Pull Up Method [Fow99]
  – You have methods with identical results on subclasses, move them to the superclass

```
public abstract class Printer
{ ...
}
public class ASCIIPrinter extends Printer
{ ...
  public void accept(Packet p) {
    if this.isDestinationFor(p)
      this.print(p);
    else
      super.accept(p); }
}
public class PSPrinter extends Printer
{ ...
  public void accept(Packet p) {
    if this.isDestinationFor(p)
      this.print(p);
    else
      super.accept(p); }
}
```



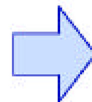
```
public abstract class Printer
{ ...
  public void accept(Packet p) {
    if this.isDestinationFor(p)
      this.print(p);
    else
      super.accept(p); }
  abstract boolean isDestFor(Packet p);
}
public class ASCIIPrinter extends Print
{ ... }
public class PSPrinter extends Printer
{ ... }
```

Refactorings primitifs

- Replace data value with object [Fow99]
  – You have a data item that needs additional data or behaviour, turn the data item into an object

```
public class Packet
{ ...
  private String contents;

  public Packet(String s, Node n)
  {
    this.setContents(s);
    ... }
  public String getContents() {
    return this.contents; }
  public void setContents(String n)
s) {
    this.contents = s; }
}
```



```
public class Packet
{ ...
  private Document doc;

  public Packet(String s, Node n)
  {
    this.setContents(s);
    ... }
  public String getContents() {
    return this.doc.contents; }
  public void setContents(String
s) {
    this.doc.contents = s; }
}

public class Document {
  ...
  public String contents;
}
```

- Exigent beaucoup de temps (des mois ou des années sur les logiciels courants)
- Exigent un accord parmi l'équipe de développement
- Exemples
  - « tease apart inheritance » (hiérarchie de transmission qui fait deux tâches en même temps)
  - « extract hierarchy » (simplifie une classe complexe en un groupe de sous-classes)
  - « convert procedural design into objects » (aide à résoudre le problème classique « quoi faire avec le code procédural ? »)
  - « separate domain from presentation » (pour isoler la logique d'affaire de l'interface utilisateur)

Tease apart inheritance

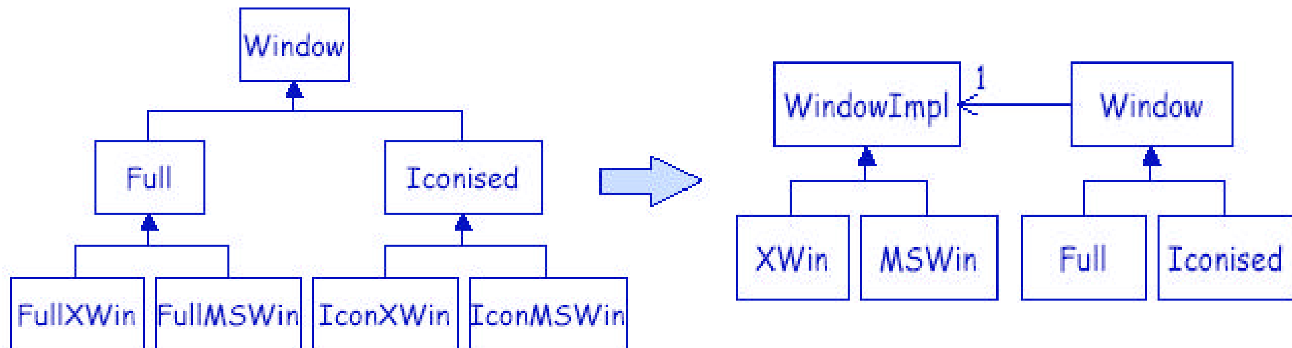
- Problème
  - Une hiérarchie de transmission qui fait deux tâches en même temps
- Solution
  - Créer deux hiérarchies séparées et utiliser la délégation pour appeler l'une de l'autre

Tease apart inheritance

- Approche
  - Identifier les différentes tâches faites par la hiérarchie
  - Extraire la tâche la moins importante dans une hiérarchie différente et garder la plus importante
  - Employer « extract class » pour créer le parent commun de la nouvelle hiérarchie
  - Créer les sous-classes appropriées
  - Employer « move method » pour déplacer une partie du comportement de la hiérarchie initiale
  - Éliminer les sous-classes inutiles (vides) dans la hiérarchie initiale
  - Appliquer d'autres refactorings (« pull up method »…)

Tease apart inheritance

- Exemple
- Patrons de conception relatifs
  - Pont (*bridge*) : découple une abstraction de son implémentation de sorte que les deux puissent changer indépendamment



Outils

- Activités de refactoring
  1. Détecter quand une application devrait être restructurée
  2. Identifier quels refactorings devraient être appliqués
  3. Exécuter les refactorings
  4. Vérifier l'effet des refactorings
- Le support fourni aujourd'hui se limite à l'étape 3 (et 4)
  - Aucun support pour les refactorings composés

Outils

- Eclipse Refactoring Browser
- Flywheel (Velocitis)
- .NET Refactoring
- C# Refactory (eXtreme Simplicity)

- Employer des métriques pour voir si la qualité du logiciel est améliorée
- Contrôler les mauvaises odeurs pour voir si elles ont été éliminées

Limites des refactorings

- Base de données
  - Domaine problématique aux refactorings
  - Difficile de changer la base de données (couplage faible entre les applications d’affaires et le schéma de la base de données)
  - Transfert de données (changer le schéma de base de données force à émigrer les données)
- Changer les Interfaces
  - Il y a un problème seulement si l’interface est employée par un code introuvable : interface éditée « published interface »
- Comportement externe du logiciel pas préservé à cent pour cent !
- …

« Any fool can write code that computer understand. Good programmers write code that humans can understand »

Martin Fowler [Fow99]

« I'm not a great programmer; I'm just a good programmer with great habits. Refactoring helps me to be much more effective at writing robust code »

Kent Beck [Fow99]

Réferences

- [Bec99] K. Beck ; *Smalltalk best practice patterns* ; Prentice Hall, 1997
- [Bec00] Kent Beck ; *eXtreme Programming eXplained: Embrace Change* ; Addition-Wesley, 2000
- [Foo95] Brian Foote and William Opdyke ; *Life Cycle and Refactoring Patterns that Support Evolution and Reuse* ; in Pattern languages of Program Design, p. 239–257, Addison-Wesley, May 1995.
- [Fow99] Martin Fowler ; *Refactoring: improving the design of Existing programs* ; Addison-Wesley, 1999
- [Gam94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides ; *Design Patterns – Elements of Reusable Object-Oriented Software* ; Addison-Wesley, 1st edition, 1994
- [Moore96] Ivan Moore ; *Automatic Inheritance Hierarchy Restructuring and Method Refactoring* ; proceedings of the 11th conference on Object-Oriented Programming, Systems, Languages, and Applications, p. 235–250, October 1996
- [RefBr] John Brant, Ralph E. Johnson , and Don Roberts; *Refactoring Browser Tool* ; available at: http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser
- [Rob98] Don Roberts, John Brant, and Ralph E. Johnson ; A refactoring tool for Smalltalk ; TAPOS Journal 3(4), p. 253–263, October 1997
- [Tok99a] Lance Tokuda and Don Batory ; *Automating Three Modes of Evolution for Object-Oriented Software Architectures* ; proceedings of the 5th Conference on Object-Oriented Technologies, May 1999
- [Tok99b] Lance Tokuda and Don Batory ; *Evolving object-oriented designs with refactorings* ; proceedings of the 15th conference on Automated Software Engineering, p. 89–120, October 1999
- Voir aussi :
  - http://www.artima.com/intv/refactor.html
  - http://www.refactoring.com
  - http://www.martinfowler.com
  - http://csc.noctrl.edu/f/opdyke
  - http://www.martinfowler.com/articles/xp2000.html
  - http://www.parlezuml.com/tutorials/dotnet_refactoring/index_files/frame.htm
  - ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z
  - http://c2.com/cgi/wiki?HistoryOfRefactoring