Refactorings

Restructuration de code

Gerson Sunyé
gerson.sunye@univ-nantes.fr

LINA - Université de Nantes

Plan

Cycle de vie en cascade (ou en V)

1. Spécification des besoins.
2. Analyse.
3. Conception.
4. Implémentation.
5. Test.
6. Et c'est tout. . . (ou presque)

La réalité

- Il est très difficile de bien faire la première fois. Il est difficile de comprendre:
  - Le domaine du problème.
  - Les besoins des utilisateurs.
  - L'évolution du système.
- Résultat:
  - La conception initiale est inadéquate.
  - Le système devient compliqué et fragile.
  - Les changements deviennent de plus en plus coûteux.

Maintenance

- Un logiciel n'est jamais fini.
- La plupart des opérations de maintenance ne sont que du développement continu.

Développement logiciel évolutionnaire

"Grow, don't build software" – Fred Brooks.

- Prototypage.
  - Donne consistance à l'analyse de besoins.
  - Croquis de la conception.
- Expansion.
  - Ajout de fonctionnalités.
  - Détermination des points d'extensions.
- Consolidation.
  - Correction des erreurs de conception.
  - Introduction de nouvelles abstractions.

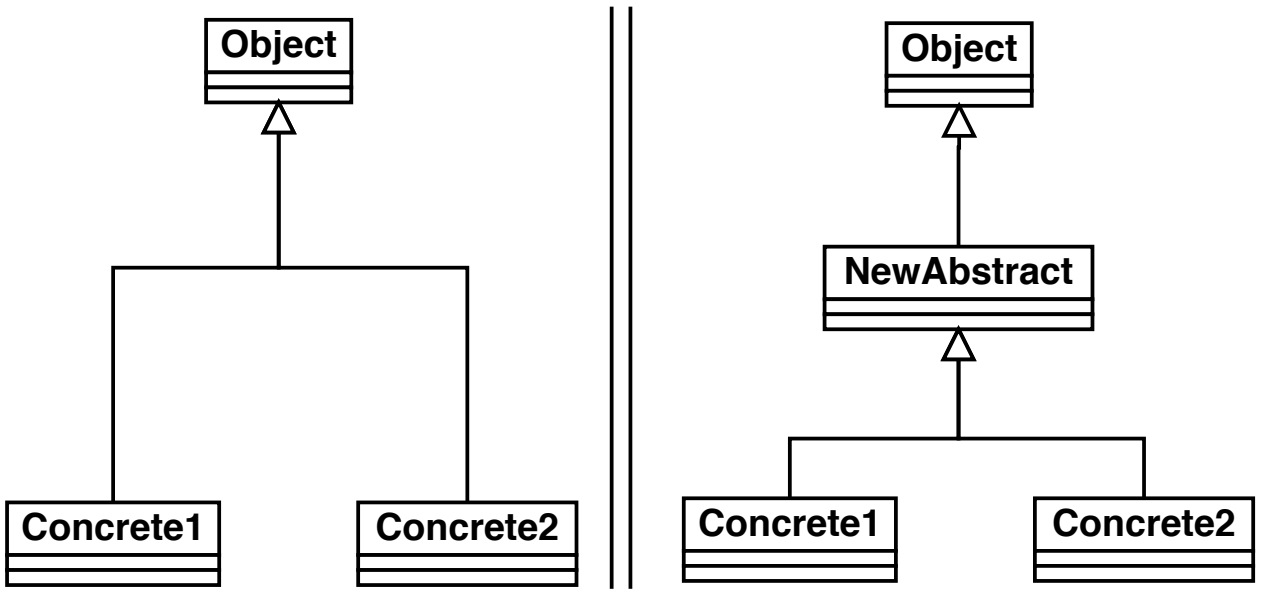
Définition

Refactoring:

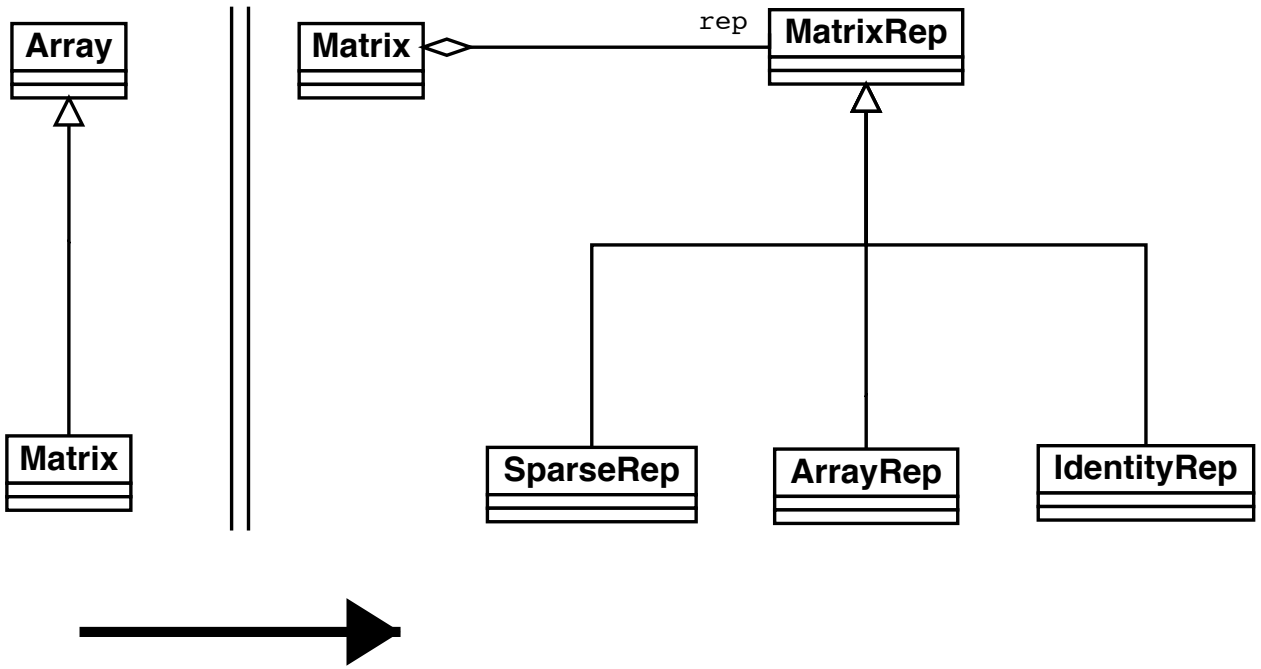
- Le processus d'amélioration de la conception d'un programme.
- Une opération de transformation du code source d'un programme qui préserve son comportement visible.
- En français: ré-usinage, refactorisation (!).

Un refactoring simple

- Créer une classe vide.



Un refactoring complexe



Origines

- Maintenance.
- Extension.
- Développement d'applications.
- Développement de cadres d'applications.

Motivation

- L'éternelle quête de l'unicité de code ...
- Le code est lu et modifié plus souvent qu'il n'est écrit.

Difficultés d'utilisation (1/2)

Complexité:

- Il est difficile de comprendre la conception.
- Modifier une conception existante est tout aussi difficile.
- Les modifications peuvent introduire des bugs.
- Le résultat n'est pas visible

Difficultés d'utilisation (2/2)

- Tout projet est toujours sous pression (délais).
- Les développeurs sont payés pour ajouter des nouvelles fonctionnalités.
- On ne change pas une équipe qui gagne (France 2002? :-).
- La restructuration d'un logiciel est parfois coûteuse.

Conséquences de la non restructuration

- Les changements sont faits de la manière la plus chère.
- La conception devient de plus en plus corrompue.
- Le code devient plus fragile.
- Les changements deviennent plus chers et plus fréquents.

Restructuration et le cycle en cascade

- Approche inverse.
- Modifier petit à petit le code pour obtenir une conception saine.
- La conception se fait continuellement pendant le développement En construisant le système, on découvre comment l'améliorer.

Les approches de restructuration

- Comme on suppose que les refactorings préservent le comportement, il est possible de les composer.
- L'apprentissage des opérations de base est semblable à celle de l'arithmétique.

Opérations de base

- Ajout et suppression d'une entité;
- Changement de nom d'une entité;
- Changement de signature d'une méthode;
- Déplacement d'une entité;
- Intra-méthode;
- Opérations composées.

Ajout d'une entité

- Ajout d'un attribut (instance ou classe).
- Ajout d'une classe.
- Ajout d'une méthode (instance ou classe).

Suppression d'une entité

- Suppression d'un attribut (instance ou classe).
- Suppression d'une classe.
- Suppression d'une méthode (instance ou classe).

Changement de nom d'une entité

- Changement de nom d'un attribut (instance ou classe).
- Changement de nom d'une variable locale.
- Changement de nom d'une classe.
- Changement de signature d'une méthode (*).

Changement de signature d'une méthode

- Changement de nom.
- Permutation de paramètres.
- Ajout d'un paramètre.
- Suppression d'un paramètre.

Déplacement d'entités

- Spécialisation/Généralisation (PushDown, PullUp) d'un attribut (instance ou classe).
- Spécialisation/Généralisation d'une méthode (instance ou classe).
- Déplacement d'une méthode (ou attribut) à une autre classe.
- Changement de superclasse.

Intra méthode

- Extraire un morceau de code en tant que méthode.
- Extraire un morceau de code en tant que variable locale.
- ”Inline” d’une méthode.
- ”Inline” d’une variable locale.
- Généralisation des références à une classe (remplacement par sa superclasse).

Opérations composées

- Encapsuler un attribut.
- Rendre un attribut en lecture seule.
- Extraire l'interface d'une classe.
- Phagocyter/Extraire une classe.
- Former une méthode template.
- etc.

Autres opérations

- Changement de visibilité d'une entité.
- Introduire méthode fabrique.
- Convertir variable en attribut.
- Extraire classe interne.
- Transformer classe interne en statique.

Utilisation d'outils standards

- Il faut tester!
- Utiliser les tests avant et après la restructuration.
- (. . .)
- Avancer à petits pas.

Scripts

- Réalisés par des experts (en Perl, Rubby, Python...).
- Renommer une méthode:
  1. Trouver les implémenteurs et les clients.
  2. Editer et renommer tous les implémenteurs.
  3. Editer et renommer tous les clients.
  4. Effacer les anciens implémenteurs.
  5. Tester !!!

Stratégies (I)

- Séparer ce qui varie de ce qui ne varie pas.
- Utiliser les patrons de conception.

Patrons de conception (1/2)

Point de variabilité	**Patron de conception**
Algorithmes	Stratégie, Visiteur.
Actions	Commande
Implémentations	Pont
Changements	Observateur
Interactions	Médiateur
Création d'objets	Fabrique abstraite, Prototype
Création de structures	Constructeur (Builder)

Patrons de conception (2/2)

Point de variabilité	**Patron de conception**
Algorithme de parcours	Itérateur
Interfaces des objets	Adaptateur
Comportement d'un objet	Décorateur, État.

Exemple de refactoring (1/3)

```
public class Client {
    public void writeAsciiOn(OutputStream o) {
        o.print('name: ');
        o.print(this.name);
        (...)}
    }
```

Et si on souhaite imprimer en HTML, XML, Framemaker, etc. ?

Exemple de refactoring (2/3)

- Créer la classe AsciiStrategy.
- Ajouter une variable d'instance à la classe Client et l'initialiser à AsciiStrategy.
- Déplacer la méthode writeAsciiOn() à la classe AsciiStrategy.
- Renommer la méthode writeAsciiOn() en writeOn().

Exemple de refactoring (3/3)

```
public class Client {
    public void writeOn(OutputStream o) {
        writeStrategy.writeOn(this, o)}
}

public class AsciiStrategy {
    public void writeOn(Client c, OutputStream o)
        o.print('name: ');
        o.print(c.name);
         (...)}
}
```

Stratégies (II)

- Ecouter le code: laisser le programme dire où restructurer (Kent Beck).
- Code smells: des heuristiques pour trouver rapidement le code qui doit être restructuré.

The good, the bad and the ugly

L'odeur du code:

- Une indication de que quelque chose n'est pas correcte dans le code.
- Utiliser le flair pour trouver le problème.
- Ce n'est pas une certitude.
- Une odeur n'est pas forcément mauvaise.

Odeurs

1. Méthode outil.
2. Code dupliqué.
3. Classe donnée.
4. Classe trop grande.
5. Méthode trop longue.
6. Paramètres rattachés.
7. Excès de commentaires.
8. Conditionnels imbriqués.
9. Désir des attributs d'autrui.
10. Généralisation spéculative.
11. Nombre excessif de paramètres.
12. Hiérarchies parallèles d'héritage.
13. Attributs d'instance peu utilisées.
14. Même nom, différentes significations.

Méthodes outil

- Ce sont des méthodes sans référence à this (ou self, current, . . .).
- Souvent, peuvent être placées ailleurs: vérifier les paramètres.
- Doivent au moins être marquées (e.g. ≪utility≫).

Code dupliqué

- Tout faire une fois et seulement une fois.
- Le code dupliqué rend complexe la compréhension du code.
- Le code dupliqué est plus difficile à maintenir:
  - Tout changement doit être dupliqué.
  - Celui qui maintient le code doit le savoir.

Code dupliqué - réparation

- Généralisation des méthodes identiques.
- Déplacer la méthode à un composant commun (Patron Stratégie?).
- (Voir: méthode trop longue)
- Refactorings: Extract Method, PullUp Method, Form Template Method

Classes données

- Classes contenant seulement des attributs, leurs accesseurs et leurs modificateurs (aka, Data Transfer Objects - DTO).
- Refactorings: Move Method.

Classe trop grande

- Souvent, l'excès de méthodes ou d'attributs cache une duplication de code.
- Une fois de plus, aucune métrique n'est exacte pour tous les cas.
- Trouver des ensembles disparates de méthodes et attributs.

Classe trop grande - réparation

- Créer des *compositions* de classes plus petites.
- Trouver les composants logiques de la classe originale et créer des nouvelles classes pour les représenter.
- Déplacer les méthodes et les attributs vers les nouveaux composants.
- Refactorings: Extract Class, Extract Subclass

Méthode trop longue

- Plus une méthode est longue, plus il est difficile de comprendre son fonctionnement.
- La méthode est la plus petite unité de surcharge.
- Aucune métrique n'est exacte pour tous les cas.
- Le corps d'une méthode doit être homogène (au même niveau d'abstraction).

Méthode trop longue - réparation

- Extraire des petits morceaux et créer des petites méthodes.
  - Si toute la méthode est longue et de bas niveau, essayer de trouver les grandes étapes (patron *template method*).
  - Habituellement, les commentaires au milieu du code sont des bons points d'extraction.
- Souvent, les petits morceaux peuvent être réutilisés.

Paramètres rattachés

- Cachent souvent un manque d'abstraction.
- (e.g. Point).
- Une fois la classe créée, il est souvent facile d'y ajouter un comportement spécifique.

Excès de commentaires

- Un commentaire doit décrire une intention et non expliquer une action.
- L'excès de commentaires inutiles surcharge le code et le rend illisible.
- Donner un nom plus significatif à la méthode, extraire le code commenté et en créer une nouvelle méthode. Introduire des assertions.

Conditionnels imbriqués

- Symptôme d'une méthode mal placée.
- Plutot que faire un choix, permettre à la surcharge de le faire.
- Les nouveaux cas ne demandent pas que le code existant soit changé (le but ultime).

Conditionnels - réparation

- Si le test de condition est un test de type (isKindOf(), type(), getClass(), etc.), transférer la méthode à cette classe (double-dispatch).
- Si la condition est isEmpty(), isNil(), null, empty(), utiliser le patron Null Object.

Désir des attributs d'autrui

- Une méthode appelle les accesseurs d'une autre classe – déplacer la méthode.
- Parfois, seulement une partie de la méthode le fait – extraire cette partie et la déplacer.

Généralisation spéculative

- Classes trop génériques en prévision d'un futur besoin.

Généralisation spéculative - réparation

- Éliminer les classes abstraites vides ou presque;
- Éliminer les paramètres non utilisés.

Nombre excessif de paramètres

- Toutes les données demandées par une méthode sont passées en paramètre.

Nombre excessif de paramètres

- Passer le minimum d'information pour que la méthode retrouve ces données.
- Créer une classe contenant tous les paramètres d'occurrence commune.
- Passer la classe à la place des paramètres.
- Trouver les méthodes qui devraient être dans la nouvelle classe.

Hiérarchies parallèles d'héritage

- L'ajout d'un classe à une hiérarchie implique l'ajout d'une classe à l'autre hiérarchie.
- Souvent, les classes des deux hiérarchies partagent le même préfixe.
- Exemples: Transaction et Comptes (assurances, bancaire, etc.) – Un Compte n'accepte que des Transactions de même type.

Attributs d'instance peu utilisés

- Si certaines instances les utilisent, et d'autres non: créer des sous-classes.
- Si utilisés seulement lors d'une opération particulière, considérer la création d'un objet-opérateur.

Même nom, différentes significations.

Conduit à une mauvaise interprétation du code:

- Identificateurs identiques: réutiliser une variable locale à une autre fin est souvent signe d'une méthode trop longue.
- Vocabulaire surchargé (in english): Order, Serialize, Thread, etc.

Nombre excessif de méthodes privées (ou protégées).

- Les méthodes doivent être publiques, sauf si elles violent l'invariant de classe.
- Les méthodes publiques peuvent être testées plus facilement.

Stratégies (III)

- Etendre et restructurer.
- Restructurer et étendre.
- Déboguer et restructurer.
- Restructurer et déboguer.
- Restructurer pour comprendre.

Etendre et restructurer

- Trouver une classe/méthode similaire et la copier.
- La faire marcher.
- Eliminer les redondances

Restructurer et étendre

- Quelque chose semble trop difficile à implémenter.
- Restructurer la conception pour simplifier les changements.
- Implémenter.

Déboguer et restructurer

- Trouver le bug.
- Restructurer le code de manière à rendre le bug évident:
  - Extraire la méthode.
  - Donner des noms significatifs.
  - Se débarrasser des nombres et des expressions magiques.

Restructurer et déboguer

- Comme les refactorings préservent le comportement, ils préservent aussi le mauvais comportement.
- Restructurer les méthodes complexes.
- Déboguer.

Restructurer pour comprendre

- Ce qui était évident pour l'auteur ne l'est pas toujours pour les autres.
- Partitionner les méthodes trop longues.
- Supprimer les expressions et nombres magiques.
- **DONNER DES NOMS SIGNIFICATIFS.**
- Ne pas s'inquiéter pour la performance.

Intégration au développement (1/4)

La métaphore du prêt:

- Coder rapidement (et cradement) est comme prendre un prêt.
- Vivre avec le mauvais code correspond aux intérêts.
- La dette est nécessaire pour commencer une affaire.
- Avoir trop de dette n'est pas salutaire et peut vous conduire à une faillite.

Intégration au développement (2/4)

Restructurer après une livraison (release):

- Un peu de temps pour respirer.
- La conception est encore fraîche dans la mémoire.

Intégration au développement (3/4)

Programmation extrême:

- Ecouter.
- Tester.
- Coder.
- Restructurer de façon continue.

Intégration au développement (4/4)

Même si vous n'êtes pas extrême, quelques principes restent valables:

- Chaque fois que quelque chose semble trop complexe, restructurer pour la simplifier.
- Laisser le programme dire où il veut être réparé.
- Restructurer après chaque copier et coller.

Conclusion

- Développement évolutionaire.
- Refactorings.
- Les odeurs de code.
- Manières d'intégrer les refactorings aux développement.

Références

- Les thèses de William Opdyke, Don Roberts et John Brant.
- Le livre de Martin Fowler et son site web:
  - http://www.refactoring.com
- La programmation extreme:
  - http://www.extremeprogramming.org
- Wiki wiki web:
  - http://c2.com/cgi/wiki?refactorings

Outils

Smalltalk : Refactoring Browser, Lint.

Python : Bicycle Repair Man, pycheck.

Java : Eclipse, IntelliJ Idea, JFactor, XRefactory, JBuilder, RefactorIt, JRefactory, Transmogrify, JafaRefactor, CodeGuide, jLint.

C++ : SlickEdit, Ref++, Xrefactory.

Ruby : Ruby Refactoring