Réusinage

Bruno Dufour
Université de Montréal
dufour@iro.umontreal.ca

Réusinage et maintenance

Le réusinage (*refactoring*) :

- consiste à apporter des améliorations à un programme dans le but de ralentir la dégradation due aux changements.
- est une forme de maintenance préventive
- vise à améliorer la structure, réduire la complexité ou améliorer la lisibilité du code.
- ne modifie pas la fonctionnalité d'un programme, mais seulement sa structure.
  - Seuls les changements qui n'affectent pas le comportement constituent du réusinage

Réusinage et réingénierie

- La réingénierie est utilisée après une période de maintenance, lorsque les coûts deviennent trop élevés.
  - Un système hérité est modifié pour obtenir un nouveau système plus facile (et moins coûteux) à maintenir.
- Le réusinage est un processus d'amélioration continu à travers le développement et l'évolution d'un logiciel. Son but est d'éviter la dégradation du code et de la structure qui nuit à la maintenance.

Réusinage et optimisation

- Le but du réusinage est d'améliorer la lisibilité du code sans changer le comportement visible
  - Un changement qui ne vise pas à améliorer la lisibilité n'est pas un réusinage
- L'optimisation (ex: performance) consiste aussi à modifier le programme sans changer le comportement (autre que la rapidité d'exécution)
  - Le but de l'optimisation est différent
  - L'optimisation rend habituellement le code plus difficile à lire et maintenir, mais est nécessaire pour atteindre la performance attendue
- Le réusinage peut réduire la performance
  - Il est généralement plus bénéfique d'optimiser un programme qui fonctionne à partir de résultats de profilage que d'optimiser spéculativement ou systématiquement (règle de 90/10)

Réusinage et développement

- Le réusinage est une technique qui permet d'appliquer des changement d'une manière efficace et contrôlée
- Il ne faut pas, par contre, mélanger l'ajout de fonctionnalités et le réusinage
- Lors de l'ajout de fonctionnalité, un programmeur ne devrait pas modifier le code existant :
  - Seulement du nouveau code et de nouveaux tests devraient être ajoutés
- Lors du réusinage, un programmeur ne devrait pas ajouter de fonctionnalité
  - Pas d'ajout ou de modification de tests, seulement l'amélioration de la structure
- Un développeur peut alterner rapidement entre les deux activités.

Avantages du réusinage

- Améliore la conception
  - Les changements répétés dégradent la structure et la conception, et empêchent de graduellement de voir la conception à travers le code. Le réusinage rend la conception visible à nouveau.
  - Le réusinage élimine la duplication de code. Moins de code facilite généralement les changements.
- Améliore la facilité de compréhension
  - Beaucoup de code est écrit sans se soucier de la maintenance effectuée dans le futur par un autre programmeur (ou soi-même!)
  - Le réusinage permet de modifier le code de façon à ce qu'il communique sons sens plus efficacement.
  - Le réusinage peut aussi servir « d'annotations » lors de la première lecture de code pour faciliter sa compréhension.

Avantages du réusinage

- Aide à identifier des bogues
  - Clarifier la structure du code permet d'exposer les hypothèses et les invariants du code, ce qui permet d'identifier certains types de bogues.
  - Le réusinage permet donc d'écrire du code plus robuste.
- Augment la vitesse de développement
  - Les changements à apporter coûtent plus cher en temps de développement si un programmeur doit passer du temps à chercher l'endroit à modifier ou à comprendre le code existant.
  - Une bonne structure permet de progresser plus rapidement.

Quand appliquer le réusinage ?

Tout au long du développement. En particulier :

- Lors de l'ajout de fonctionnalité
  - Si le code est difficile à comprendre, il est temps de réusiner
  - Si la structure ne permet pas d'effectuer l'ajout facilement, le réusinage peut corriger le problème.
- Lors du débogage
  - Un bogue découvert indique que le code n'était pas assez clair pour se rendre compte qu'un bogue était présent
  - Le réusinage permet aussi de mieux comprendre le code lors du débogage
- Durant une revue de code
  - Donne des résultats plus concrets puisque le code peut être réusiné durant le revue pour faciliter les discussions et en générer de nouvelles.

Problèmes du réusinage

- Bases de données
  - Le modèle de la base de données est souvent étroitement couplé au modèle d'objets dans le reste du système. Il est donc difficile d'apporter des changements à ces modèles.
  - En utilisant une couche d'abstraction entre les deux modèles, on peut isoler les changements à un modèle, au coût d'une complexité plus élevée.
- Changements d'interfaces
  - Les changements d'interface nécessitent une connaissance de tout de code qui utilise cette interface.
  - Lorsque c'est impossible, un changement peut causer des problèmes avec le code client.
  - Une solution imparfaite consiste à conserver l'ancienne interface pendant un certain temps pour permettre au code client d'être adapté.

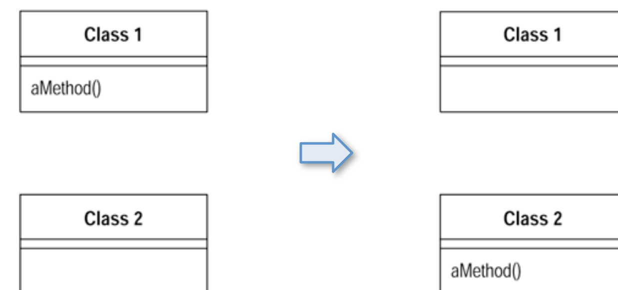
Problèmes du réusinage

- Changements de design
  - Le réusinage de remplace pas une bonne conception de départ.
  - Même les techniques de développement comme XP effectuent un peu de conception avant de coder.
  - Il peut être difficile de réusiner pour pallier des problèmes de conception
    - Ex: réusiner un logiciel pour ajouter des exigences de sécurité
  - Dans ces situation, il faut investir plus d'effort dans la conception initiale.

Déplacements et généralisation

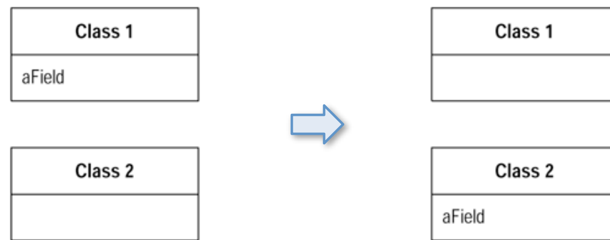
Déplacer une méthode

- Créer une méthode similaire dans la classe qu'elle utilise le plus. L'ancienne méthode pour être supprimée ou modifiée pour déléguer à la nouvelle.



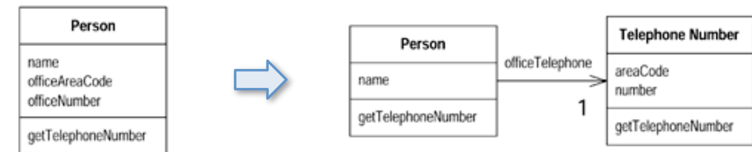
Déplacer un champs

- Créer un champs similaire dans une class cible et remplacer toutes ses utilisations.



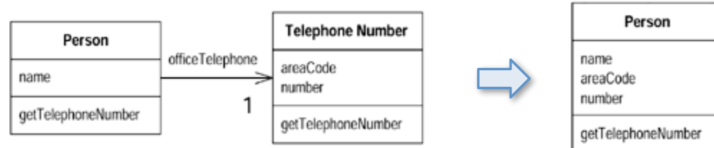
Extraire une classe

- Créer une nouvelle classe et déplacer les champs et méthodes appropriés depuis l'ancienne classe vers la nouvelle.



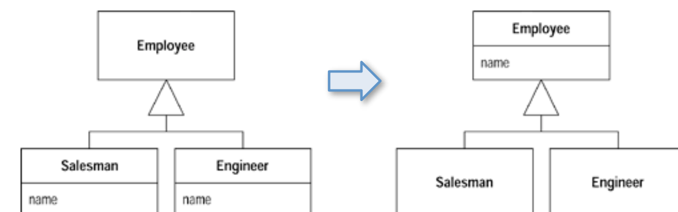
Incorporer une classe

- Déplacer tous les champs et les méthodes d'une classe vers une autre, puis la supprimer.



Remonter un champs

- Déplacer un champs commun vers une superclasse.



Aplatir la hiérarchie

- Combiner une classe et sa sous-classe qui ne sont pas suffisamment différentes.

Composition de méthodes

Extraire une méthode

- Permet de convertir un fragment de code en une méthode dont le nom indique son utilité

```
void printOwing(double amount) {
      printBanner();

      //print details
      System.out.println ("name:" + _name);
      System.out.println ("amount" + amount);
}
```

```
void printOwing(double amount) {
      printBanner();
      printDetails(amount);
}

void printDetails (double amount) {
      System.out.println ("name:" + _name);
      System.out.println ("amount" + amount);
}
```


Extraire une méthode

- Procédure
  - Créer une nouvelle méthode avec un nom représentatif de son utilité
  - Copier le code extrait de la méthode source vers la nouvelle méthode cible
  - Identifier l'utilisation de variables locales et paramètres dans le code extrait
  - Si certaines de ces variables sont temporaires et utilisées seulement par le code extrait, les déclarer dans le code cible comme variables temporaires
  - Identifier les variables modifiées par le code extrait. Si une seule variable est modifiée, déterminer si la méthode cible peut être traitée comme une requête et sa valeur de retour assignée à la variable modifiée. Les autres cas nécessitent d'autres réusinages avant de pouvoir effectuer l'extraction
  - Passer en paramètres à la variable cible toutes les variables lues dans le code original
  - Remplacer le code extrait dans la méthode source par un appel à la méthode cible

Incorporer une méthode

- Déplacer le contenu d'une méthode vers ses appelants et supprimer la méthode
- Utile lorsque le nom d'une méthode est aussi clair que son contenu

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
      return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```


Incorporer une variable temporaire

- Remplacer toutes les occurrences de cette variable par l'expression
- Utile lorsqu'une variable temporaire empêche l'application d'un autre réusinage

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```

```
return (anOrder.basePrice() > 1000);
```


Remplacer une variable par une requête

- Extraire une expression dans une méthode. Remplacer la variable temporaire par un appel à la nouvelle méthode.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
…
double basePrice() {
    return _quantity * _itemPrice;
}
```


Supprimer l'affection à un paramètre

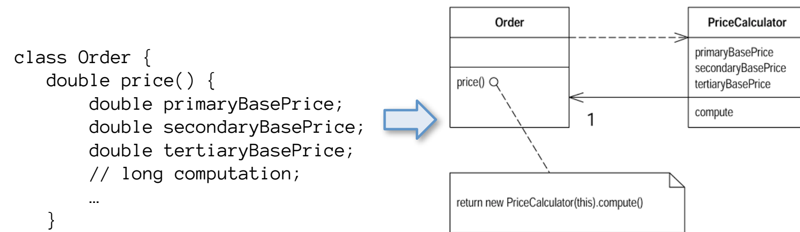
- Remplacer une affection à un paramètre par une variable.

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    …
```

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    …
```

Remplacer une méthode par un objet

- Remplacer une méthode par un objet de façon à ce que les variables locales deviennent des champs. Le nouvel objet peut être décomposé par la suite.



Simplification des données

Préserver l'objet entier

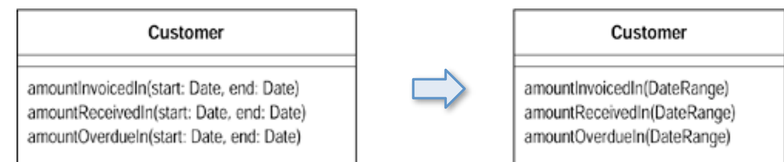
- Passer l'objet entier plutôt que des valeurs extraites de ce dernier.

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

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


Introduire un objet paramètre

- Remplacer un groupe de paramètres naturellement reliés par un objet.



Comment appliquer le réusinage ?

- Il est facile de décrire un réusinage concret
  - ex: supprimer un champs, renommer une méthode, etc.
- Dans quelles situations doit-on appliquer les réusinages ?
  - Certains symptômes (odeurs de code, ou *code smells*) suggèrent la présence d'un problème
  - Pour chaque odeur, on peut définir une stratégie de réusinage qui élimine le problème
- Une liste d'odeurs ont été identifiées à partir d'expérience avec des projets variés
  - Le symptômes identifiés ne sont que des suggestions, et laissent place à l'intuition des développeurs lorsque c'est nécessaire

Quelques odeurs

- Duplication de code
  - Un des symptômes les plus répandus
  - Même expression dans deux méthodes de la même classe
    - Appliquer Extraire une méthode et invoquer le code à partir des deux endroits
  - Même expression dans deux classes reliées
    - Appliquer Extraire une méthode et puis Remonter une méthode pour déplacer la méthode vers une superclasse commune.
    - D'autres réusinages peuvent être nécessaires pour permettre l'extraction de méthode si les deux expressions sont similaires mais pas identiques.
  - Même expression dans deux classes non-reliées

Quelques odeurs

- Duplication de code
  - Même expression dans deux classes non-reliées
    - Devraient-elles être reliées? → Introduire un ancêtre commun avec Extraire une classe, et Remonter une méthode.
    - Le code devrait-il appartenir à une seule des deux classes? → L'autre classe devrait faire appel à la fonctionnalité de la première avec Extraire une méthode.
    - Une troisième classe doit-elle être créée pour contenir la fonctionnalité dupliquée ?

Quelques odeurs

- Longue méthode
  - Un code orienté-objet devrait utiliser des méthodes courtes (« écrire une nouvelle méthode à la place d'ajouter un commentaire »)
  - Une longue méthode est souvent un symptôme de :
    - trop de responsabilités à la fois
    - mauvaises abstractions
  - Extraire une méthode est généralement suffisant pour effectuer la décomposition en méthodes plus courtes
  - Dans certains cas, il est nécessaire d'appliquer Remplacer une variable temporaire par une requête afin d'éliminer des variables locales, Introduire un objet paramètre pour réduire la liste de paramètres, etc.
  - Pour les cas encore plus difficiles, Remplacer une méthode par un objet méthode peut être nécessaire.

Quelques odeurs

- Longue classe
  - Trop de champs ?
    - Grouper les champs en composants qui sont reliés
    - Réduire la taille de la classe avec Extraire une classe et Extraire une sous-classe pour créer les composants
  - Trop de code ?
    - Éliminer la duplication dans la classe
    - Réduire la taille de la classe avec Extraire une classe et Extraire une sous-classe

Quelques odeurs

- Longue liste de paramètres
  - Avec les objets, il suffit de passer à une méthode ce qui est nécessaire pour qu'elle puisse obtenir toute l'information requise.
  - Une grande partie de cette information se trouve dans la classe de la méthode.
  - Les longues listes de paramètres sont difficiles à comprendre, à utiliser et à modifier.
  - Utiliser Remplacer un paramètre par une méthode dans les cas où l'information peut être obtenue par un objet déjà accessible.
  - Utiliser Préserver l'objet entier pour remplacer des données qui proviennent du même objet.
  - Utiliser Introduire un objet paramètre pour grouper des données qui n'ont pas de connexion existante entre elles.

Quelques odeurs

- Changements divergents
  - Avec le temps, certaines classes accumulent trop de responsabilités.
  - Des changements divergents sont présents lorsqu'une classe doit changer de façon différente pour des raisons différentes
    - ex: Lorsque qu'une nouvelle base de données est ajoutée, ces 3 méthodes doivent changer. Lorsque qu'un nouvel accord financier est ajouté, ces 4 autres méthodes doivent changer.
    - indique souvent un manque de cohésion
  - Utiliser Extraire une classe pour chaque cause de changements distincte.
- *Shotgun surgery* (l'opposé des changements divergents)
  - S'applique lorsqu'un changement requiert plusieurs petits changements répartis dans le programme
    - Il est facile d'oublier un changement à effectuer
  - Utiliser Déplacer une méthode / un champs et Incorporer une classe pour rassembler les morceaux éparpillés.

Quelques odeurs

- Jalousie fonctionnel (*feature envy*)
  - Une méthode appelle beaucoup d'accesseurs d'une autre classe pour obtenir les données dont elle a besoin.
  - Utiliser Extraire une méthode et Déplacer une méthode pour déplacer la méthode vers la classe à laquelle elle devrait appartenir.
- Grappe de données
  - Un ensemble de variables sont souvent utilisées et modifiées ensemble et devraient être transformées en un objet.
  - Extraire l'objet à partir des champs où ces variables sont utilisées à l'aide de Extraire une classe
  - Utiliser Préserver l'objet entier ou Introduire un objet paramètre pour les autres utilisations.

Quelques odeurs

- *Switch*
  - Causent la duplication de code, puisque les mêmes conditions se retrouvent souvent à plusieurs endroits dans le code.
  - Devraient être remplacés par le polymorphisme
  - Utiliser Extraire une méthode et Déplacer une méthode pour extraire et relocaliser le *switch*
  - Utiliser Remplacer un code de type par des sous-classes et Remplacer un code de type par un état/stratégie pour mettre en place la structure d'héritage nécessaire
  - Utiliser Remplacer une condition par du polymorphisme pour éliminer le *switch*

Quelques odeurs

- Généralité spéculative
  - « Nous aurons peut-être besoin de ceci un jour… »
  - Causé par l'introduction de code trop général qui traite des cas qui ne sont pas nécessaires ou utilisés.
  - Augmente la complexité du code sans bénéfice
  - Utiliser Aplatir la hiérarchie pour supprimer des classes abstraites inutiles
  - Utiliser Incorporer une classe pour éliminer la délégation inutile
  - Utiliser Supprimer un paramètre pour éliminer les paramètres superflus
  - etc.

Antipatrons

Patrons et anti-patrons

- Rappel : les patrons de conception décrivent un problème commun et une solution abstraite généralement souhaitable.
- Un anti-patron décrit deux solutions :
  - La première est une solution problématique répandue
  - La deuxième est une solution réusinée pour en améliorer la structure et éviter les effets négatifs.
  - Cette solution proposée n'est pas forcément unique, mais représente une façon possible et efficace d'obtenir plus de bénéfices.
- Un patron appliqué dans un mauvais contexte peut devenir un anti-patron

Anti-patrons et odeurs

- Les anti-patrons et les odeurs de code (*code smells*) sont étroitement reliés
  - Une odeur indique souvent un mauvais style, alors qu'un anti-patron identifie un solution précise comme étant mauvaise.
  - Une relation similaire existe entre un bon style de programmation et un patron de conception.
- Souvent, un anti-patron implique qu'une ou plusieurs odeurs peuvent être détectées dans la solution problématique.

Quelques anti-patrons

- Code spaghetti
  - Un des anti-patrons les plus répandus
  - Le code contient peu de structure, ou utilise une structure de contrôle excessivement complexe
  - Plusieurs symptômes possibles :
    - Style de programmation procédural
    - Flot de contrôle basé sur l'implémentation des objets et non pas sur les clients de ces objets
    - Utilisation de variables globales
    - Relations minimales entre objets
    - Réutilisation par duplication et non héritage
    - Difficulté à effectuer la maintenance
  - Solution réusinée :
    - Beaucoup de transformations nécessaires (introduction de nouvelles méthodes, élimination du code dupliqué, etc.)

Quelques anti-patrons

- Blob
  - Une classe s'occupe principalement du traitement alors que les autres ne font qu'encapsuler des données
  - Plusieurs symptômes possibles :
    - Une classe qui contient beaucoup de champs et/ou d'opérations
    - Une classe qui manque de cohésion
    - L'absence de structure orientée-objet (structure procédurale)
    - Difficulté de réutilisation et/ou tests
  - Solution réusinée :
    - Même que pour l'odeur « Grande classe »

Quelques anti-patrons

- Décomposition fonctionnelle
  - Les classes du système correspondent à des sous-routine plutôt qu'à des concepts OO
  - Plusieurs symptômes possibles :
    - Les classes ont des noms de fonctions (CalculerIntérêt, AfficherTableau, etc.)
    - Les classes n'effectuent qu'une seule action
    - Tous les champs sont privés et ne sont utilisés que par la classe elle-même
    - Pas ou très peu d'héritage ou de polymorphisme
    - Difficulté de réutilisation
  - Solution réusinée :
    - Définir un modèle pour l'application
    - Combiner des classes pour atteindre l'objectif
    - Transformer les classes sans état en fonctions