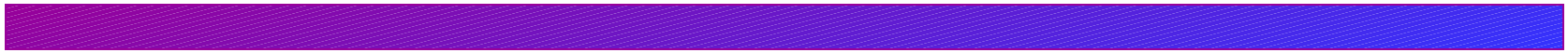


R6.A.06
Maintenance applicative

I.Borne
(isabelle.borne@univ-ubs.fr)

COURS n°1

- Refactoring
- Construire des tests
- Catalogue de refactorings
- Exemples de refactoring de code
- Refactoring d'architecture logicielle

Qu'est-ce que le refactoring ?

Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor, you are improving the design of the code after it has been written.

M.Fowler

Reengineering and Refactoring

Les deux concepts sont assez similaires

Reengineering : est l'examen et la modification d'un système pour le reconstituer sous une nouvelle forme et la mise en œuvre ultérieure de la nouvelle forme.

- **Portée** : La réingénierie affecte toujours l'ensemble du système ; le refactoring a généralement de nombreux effets locaux.
- **Processus** : La réingénierie suit une approche de démontage/remontage le refactoring est un processus de préservation du comportement et de transformation de la structure.
- **Résultat** : la réingénierie peut créer un tout nouveau système, avec une structure, un comportement et une fonctionnalité différents ; Le refactoring améliore la structure d'un existant système, en laissant son comportement et ses fonctionnalités inchangés.

Ce qui rend les programmes difficiles à utiliser

- Les programmes qui sont difficiles à lire sont difficiles à modifier.
- Les programmes qui ont de la logique dupliquée sont difficiles à modifier.
- Les programmes qui nécessitent un comportement supplémentaire qui vous oblige à modifier le code en cours d'exécution sont difficiles à modifier.
- Les programmes avec une logique conditionnelle complexe sont difficiles à modifier.

Les raisons du refactoring

- Raison pour utiliser le refactoring
  - Améliorer la conception et la maintenance
  - Une meilleure visibilité
  - Les bugs

- Les règles de trois
  - Faire le refactoring avant d'ajouter de nouvelles fonctionalités
  - Faire le refactoring quand on fixe des bugs car le refactoring aide à trouver le bug
  - Dans le cadre des révisions de code pour appliquer les améliorations

Bad smells in code

- Voir la liste complète dans l'ouvrage de M.Fowler
- Exemples de *bad smells* à identifier dans le code
  - Du code est dupliqué
  - Méthodes qui s'étendent sur plusieurs dizaines de lignes.
  - Toutes les sous-classes introduisent la même méthode
  - Variables temporaires
  - Les switch
  - ...

Quelles sources d'erreurs le refactoring permet-il de corriger ?

- Méthodes brouillonnes ou trop longues
- Les doublons (redondances)
- Les listes de paramètres trop longues
- Les classes avec un trop grand nombre de fonctions
- Les classes avec trop peu de fonctions
- Les codes trop généraux avec des cas spécifiques

Comment tirer parti du refactoring

- Il est essentiel de mettre en place une vaste suite de tests
- Les étapes de refactoring sont petites (concevoir un peu, coder un peu, changer un peu, tester)
- Les pires problèmes ou zones à risque en premier
- Si la suite de tests échoue, recommencez

Construire des tests

- Assurez-vous que tous les tests sont entièrement automatiques et qu'ils vérifient leurs propres résultats
- Une suite de tests est un puissant détecteur de bugs qui réduit le temps nécessaire à la recherche d'un bug.
- Exécutez des tests fréquemment. Exécutez ceux qui exercent le code sur lequel vous travaillez au moins toutes les quelques minutes ; effectuez tous les tests au moins quotidiennement.
- Il est préférable d'écrire et d'exécuter des tests incomplets plutôt que de ne pas exécuter de tests complets.
- Pensez aux conditions limites dans lesquelles les choses pourraient mal se passer et concentrez-y vos tests.
- Lorsque vous recevez un rapport de bug, commencez par écrire un test unitaire qui expose le bug.

Catalogue de refactorings

Le standard utilisé par les concepteurs (M.Fowler) :

- Name
- Short summary : situation dans laquelle on a besoin du refactoring
- Motivation : décrit pourquoi le refactoring doit être fait et le circonstances dans lequelles il ne doit pas être fait.
- Mechanics : description pas-à-pas de comment réaliser le refactoring
- Examples : montrent une utilisation très simple

- Martin Fowler https://refactoring.com/catalog/ : catalogue en ligne (V2 : exemple en javaScript, V1 en Java)
- Autre : https://refactoring.guru/refactoring/catalog

Les premiers refactoring à étudier (TP)

- Move Method
- Replace temp with Query
- Form Template Method
- Replace Conditional with Polymorphism
- Replace Type Code with State/Strategy
- Self Encapsulate Field

Extract Method

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

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

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

•Replace temp with Query

- Vous utilisez une variable temporaire pour conserver le résultat d'une expression
- Extraire l'expression dans une méthode. Remplacer toutes les références à cette temporaire avec l'expression. La nouvelle méthode peut alors être utilisée dans d'autres méthodes.

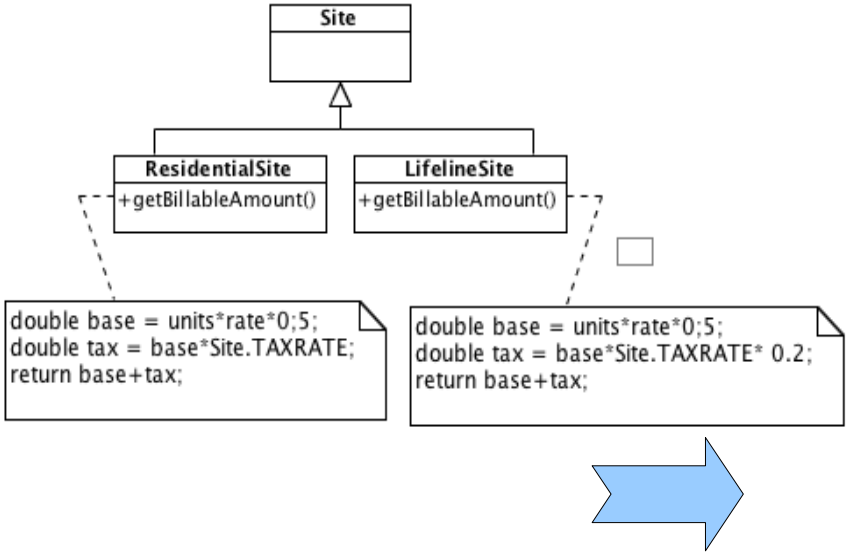
```
double basePrice = quantity * itemPrice ;
if (baseprice > 1000)
   return basePrice * 0.95
else return basePrice *0.98 ;
```

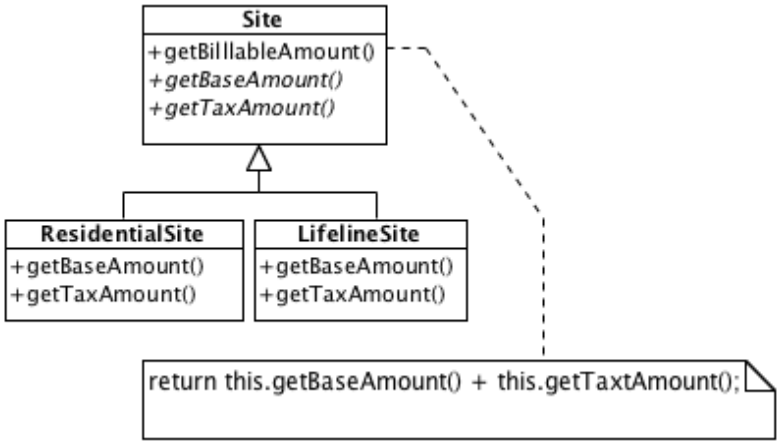
```
if (this.basePrice()>1000)
     return this.basePrice() * 0.95
else return this.basePrice() *0.98 ;


…
double basePrice() {
    return quantity* itemPrice ;
}
```

Form Template Method

- Vous avez 2 méthodes dans des sous-classes qui ont des traitements similaires dans le même ordre.

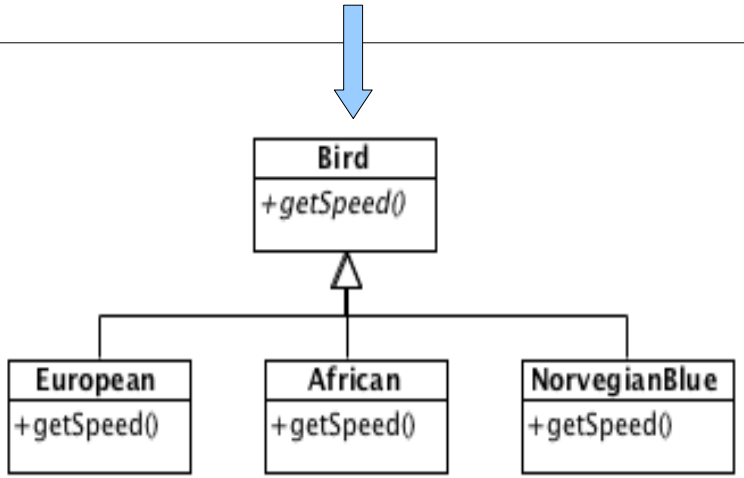




Replace Conditional with Polymorphism

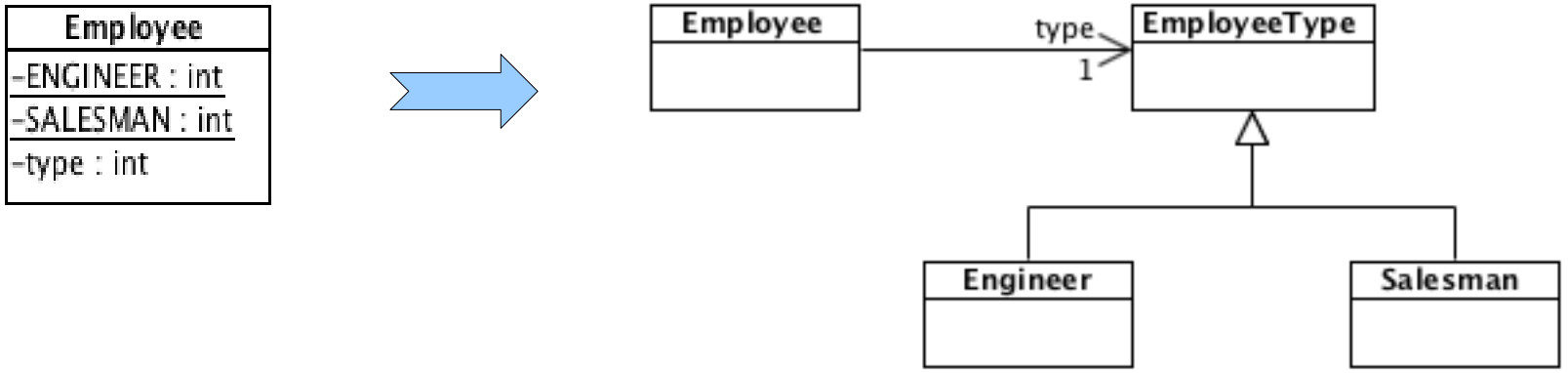
- Déplacer chaque branche de la conditionnelle dans une méthode redéfinie dans la sous-classe ; rendre abstraite la méthode d'origine.

```
double getSpeed() {
  switch (type) {
     case EUROPEAN:return this.getBaseSpeed() ;
     case AFRICAN :
       return this.getBaseSpeed() -
                this.getLoadFactor() + numberOfCounts ;
     case NORVEGIANBLUE :
       return   … ;
….
```



Replace Type Code with State/Strategy

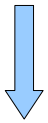
- Vous avez un code de type qui affecte la comportement d'une classe, mais vous ne pouvez pas sous-classer.
- => remplacer le code de type avec un objet état.



Self Encapsulate Field

- Vous accédez à un champ directement mais le couplage devient problématique.
- => créer des méthodes get/set pour le champ et toujours les utiliser pour accéder au champ

```
private int low, high ;
boolean includes(int arg){
    return arg>= low && arg<= high ;
```



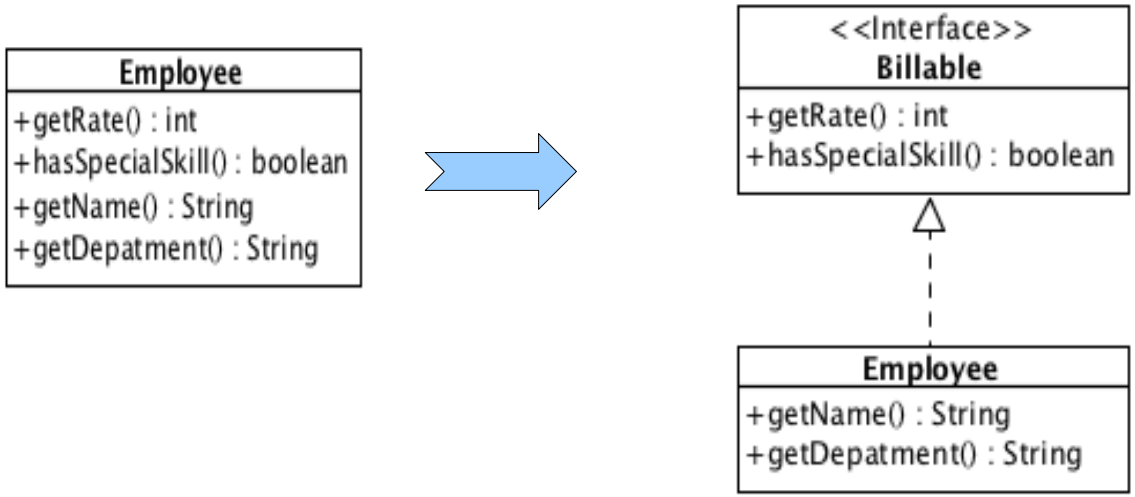
```
private int low, high ;
boolean includes(int arg){
  return arg>= this.getLow()&& arg<= this.getHigh() ;
}
int getLow() {return low;}
int getHigh() {return high;}
```

D'autres refactorings classiques

- Rename Method
- Pull Up Method
- Extract Interface
- Extract Subclass
- Collapse Hierarchy
- Inline Method
- Introduce Null Object
- Change reference to Value
- Change Value to Reference
- Replace parameter with Explicit Methods
- Remove Assignments to Parameters

Extract Interface

- Plusieurs clients utilisent le même sous-ensemble d'une interface de classe, ou deux classes ont une partie de leur interface en commun.
- => Extraire le sous-ensemble dans une interface.



Inline Method

- Le corps d'une méthode est aussi clair que son nom
- => mettre le corps de la méthode dans le corps de son appelant et la supprimer.

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

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

Replace parameter with Explicit Methods

- Une méthode fonctionne avec un code différent selon les valeurs d'un paramètre énuméré
- => Créer une méthode séparée pour chaque valeur des paramètres

```
void setValue (String name, int value) {
   if (name.equals("height")) height = value ;
   if (name .equals("width")) width = value ;
   Assert.souldNeverReachHere() ;
}
```

```
void setHeight (int arg) {
   height = arg ;
}
void setWidth (int arg) {
   width = arg ;
}
```

Remove Assignments to Parameters

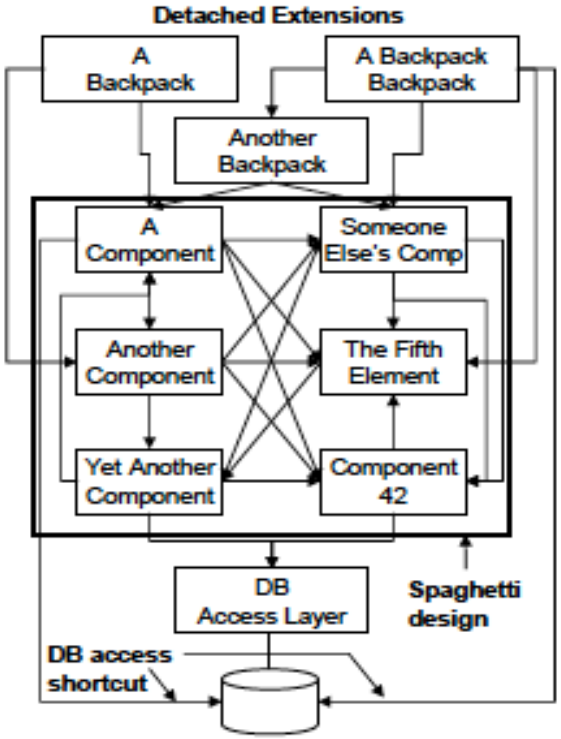
- Le code fait une affectation à un paramètre
- => Utiliser une variable temporaire à la place

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

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

Refactoring d'architecture logicielle

- Après un certain temps beaucoup d'architectures ressemblent à celle-ci



- La vision de l'architecture originale est difficilement visible
- Les défauts de conception sont étayés par de nombreuses corrections petites et locales
- Les pièces manquantes sont fixées via des backpacks

- Une telle architecture souffre des qualités de développement comme la flexibilité et la maintenabilité et des qualités opérationnelles comme la performance et l'évolutivité.

Raffinement et refactorings

Créez une architecture logicielle étape par étape via un certain nombre de petits incréments bien définis. Chaque incrément comprend :

- Des activités de raffinement descendantes pour détailler et compléter l'architecture logicielle.
- Des activités de refactoring ascendantes pour reprendre et nettoyer les décisions de conception incohérentes ou insuffisantes.
- Le processus s'arrête si l'architecture logicielle est complète et cohérente dans toutes ses parties et détails.

Types de refactoring

- Le refactoring désigne la transformation sémantique invariante d'un artefact logiciel
- Cependant, le refactoring n'est pas limité au code. Le refactoring peut également adresser :
  - Diagrammes UML, aspects, règles d'expressions DSL (Domain specific language)
  - Documents
  - Processus
  - Plans de test
  - Bases de données

Tester les refactorings d'architecture

- Pour vérifier l'exactitude des refactorings, diverses options sont disponibles
  - Approche formelle : prouver la sémantique et l'exactitude de la transformation du programme
  - Approche de mise en œuvre : tirer parti des tests unitaires pour vérifier que la mise en œuvre résultante répond toujours aux spécifications.
  - Analyse de l'architecture : vérifier l'architecture logicielle résultante pour son équivalence avec l'architecture initiale (tenir compte des exigences)

- Utiliser au moins une méthode de vérification pour garantir la qualité

Architecture smells

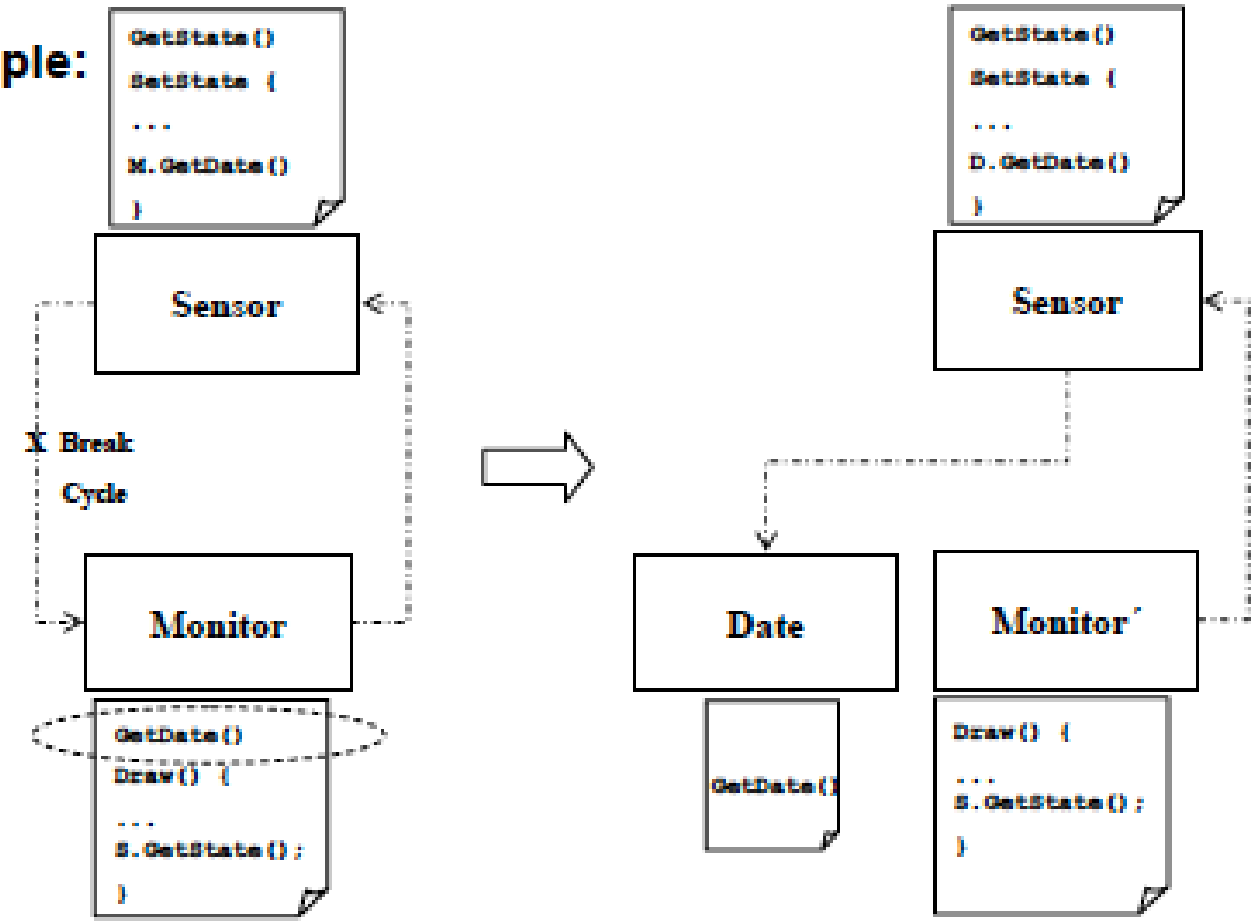
- Breaking the DRY(Dont Repeat Yourself) rule
- Unclear roles of entities
- Inexpressive or complex architecture
- Everything centralized
- Not invented here syndrome
- Over-generic design
- Asymmetric structure or behavior
- Dependency Cycles
- Design violations (such as relaxed layering)
- Inadequate partitioning
- Unnecessary dependencies
- Missing orthogonality

Exemple : Break Depedency Cycles

- Contexte
  - Cycles de dépendances de sous systèmes
- Problème
  - Le système révèle au moins un cycle de dépendance dans le graphe des relations des sous sytèmes
  - A peut dépendre directement de B ou indirectement (par exemple, A dépend de C qui dépend de B), c'est pourquoi il faut toujours considérer l'enveloppe transitive
  - Les cycles de dépendance rendent les systèmes moins maintenables, modifiables, réutilisables, testables et compréhensibles.
- Idée de solution générale
  - Se débarrasser de tout cycle de dépendance en introduisant des sous-systèmes supplémentaires
  - ou briser le cycle en utilisant des dépendances inverses

Break Depedency Cycles

- **Example:**



Break Depedency Cycles (variante)

Inversion de dépendance