

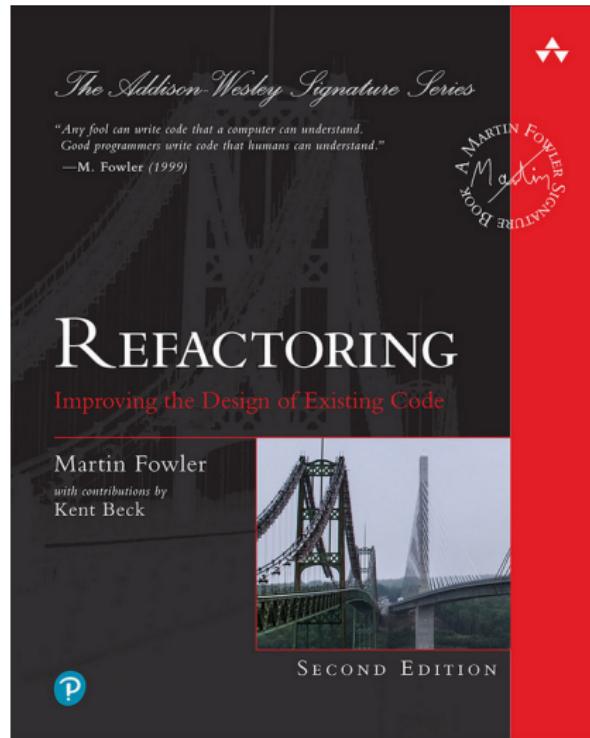
# Maintenance applicative

---

Chouki Tibermacine

Chouki.Tibermacine@univ-ubs.fr

# Livre de référence



# Plan du cours

1. Introduction générale
2. Bad Smells
3. Catégories de refactorings
  - 3.1 Refactorings de base
  - 3.2 Refactorings pour l'encapsulation
  - 3.3 Refactorings pour le déplacement de features
  - 3.4 Refactorings pour l'organisation des données
  - 3.5 Refactorings pour la simplification de la logique
  - 3.6 Refactorings pour les API
  - 3.7 Refactorings pour traiter l'héritage

# Qu'est-ce que le refactoring ?

- Réusinage en français (terme utilisé principalement au Québec) : amélioration de la qualité interne (maintenabilité : lisibilité & modifiabilité, ...) d'un logiciel
- Refactoring du code ou des modèles d'une application : les 2 sont possibles (refactoring de code étant plus répandu)
- Pratique courante dans (et surtout promue par) les méthodes agiles :
  - Utilisation systématique de cette pratique lors du développement

*“Any fool can write code that a computer can understand. Good programmers write code that humans can understand” (M. Fowler)*

## Quelques objectifs du refactoring

- Supprimer les parties redondantes
- Simplifier l'algorithmique
- Réduire la complexité des classes/méthodes
- Améliorer la lisibilité (des noms, par exemple)

Modifier la structure pour **améliorer la lisibilité et simplifier les changements futurs, sans altérer son comportement observable** par l'utilisateur

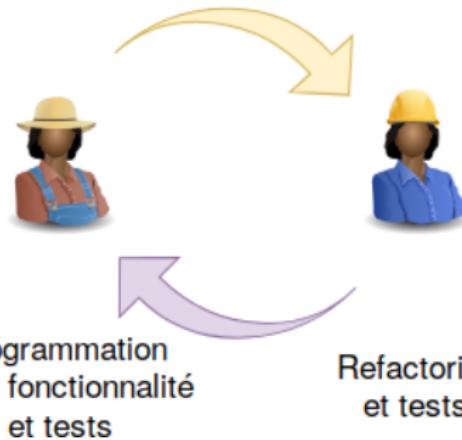
## Quelques exemples de refactorings

- Extraire une fonction (à partir d'un code au milieu d'une fonction trop longue)
- Extraire une classe ou une interface
- Changer la déclaration d'une fonction (ses paramètres)
- ...

# Terminologie

- Refactoring (nom) : une opération nommée qui consiste à effectuer un petit changement dans la structure (*Extract Function, Replace Conditional with Polymorphism, ...*)
- Refactoring (verbe) : appliquer une opération de changement
- Refactoring : un cas particulier de restructuration, qui consiste à appliquer une **petite** opération, qui ne casse pas le code (pendant un long moment)

# Porter les deux casquettes



- Réaliser les deux activités de façon séparée (à un instant t, soit l'une soit l'autre) :
  - On écrit et exécute les tests
  - On modifie le code puis on ré-exécute les tests (on n'en écrit pas de nouveaux, sauf exceptionnellement s'ils en manquent)

# Processus de refactoring

## Avant de faire un refactoring

- Les refactorings sont des petits changements qui ne doivent pas altérer le comportement du système
- S'assurer que l'on dispose d'une solide suite de tests et que ces tests passent tous, avant d'appliquer les refactorings
- Rendre les tests self-checking (passent au rouge ou au vert) pour ne pas passer beaucoup de temps à vérifier les résultats des tests

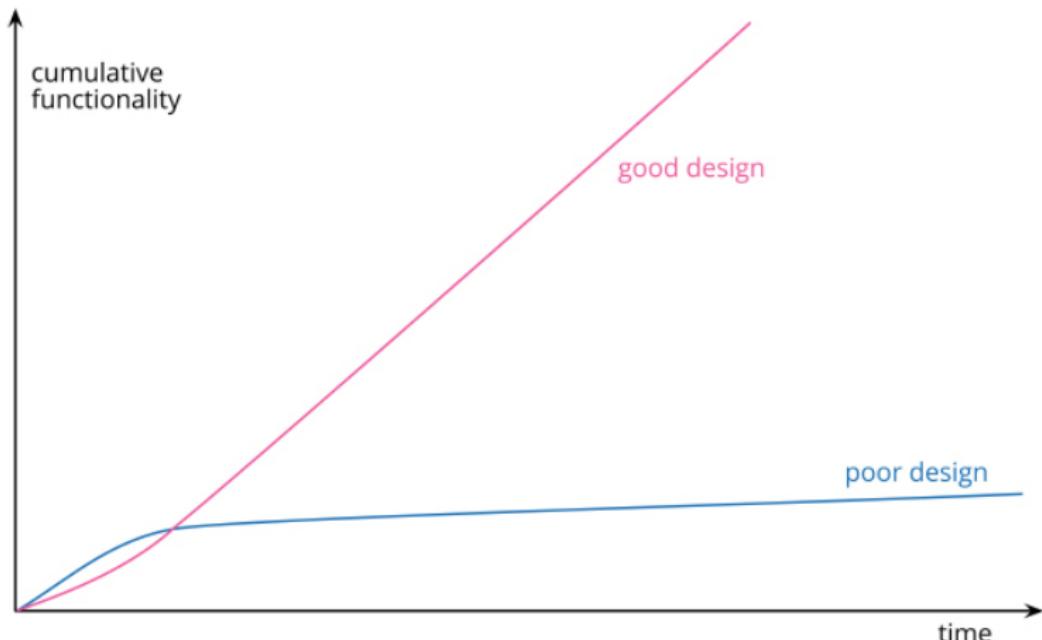
## Après avoir fait un refactoring

Systématiquement re-compiler et re-exécuter les tests, pour avancer par **petits** pas/changements sûrs

## “No Silver Bullet”

- Le refactoring n'est pas la solution miracle répondant à tous les problèmes de conception
- Mais c'est un outil efficace pour améliorer la conception et donc la qualité interne (pas les performances) :
  - Au fil du temps, sans refactoring régulier, l'architecture/la conception “se désintègre” (*Architecture/Design Decay*)
  - Il devient difficile de reconstruire l'architecture à partir du code (en lisant le code)
  - Code mal conçu implique un code plus long pour faire les mêmes choses (duplication de code)

## Capacité d'ajouter des fonctionnalités vs bonne/mauvaise conception (l'hypothèse d'endurance)



# Quand faire du refactoring ?

## Refactoring préparatoire

- Avant d'ajouter une nouvelle fonctionnalité
- Il se peut qu'on ait une fonction qui fait presque la même chose (mais qui contient des littéraux différents, par ex)
- Souvent, on copie/colle la fonction et on la modifie (mauvaise idée : code dupliqué ⇒ des modifications futures à faire 2 fois)
- Le mieux est d'utiliser le refactoring *Parameterize Function* et ensuite appeler cette fonction

# Quand faire du refactoring ?

## Refactoring de compréhension

- Avant de modifier du code, on doit d'abord le comprendre
- On peut être amené à faire en sorte que le code soit plus lisible (changer les noms, améliorer des structures conditionnelles, ...)
- A travers ce refactoring, on va déplacer notre compréhension du code de notre esprit vers le code
- Et ça devient persistant dans le temps

# Quand faire du refactoring ?

## Refactoring de nettoyage (*Litter-Pickup*)

- Le refactoring ne concerne pas directement ce que je dois faire
- Mais je vois qu'il y a du code mal écrit
- Sans se laisser disperser (et passer beaucoup de temps loin de ses tâches), appliquer un refactoring sur ce code, ou au moins ajouter un TODO et le faire(-faire) plus tard

# Quand faire du refactoring ?

## Refactoring planifié (opportuniste)

- Refactroing fait sur un code bien écrit (on ne le fait pas que sur du code mal écrit)
- Même si on a bien pensé certaines parties du code (bon découpage en fonctions, bons paramètres, ...), ce qui était vrai hier peut ne pas l'être aujourd'hui
- On pense qu'un programmeur, pour ajouter une nouvelle fonctionnalité, doit principalement ajouter du code (faux)
- Il doit d'abord modifier la base de code pour rendre l'ajout facile et ensuite ajouter un code "minimaliste"

*"For each desired change, make the change easy (warning : this may be hard), then make the easy change", Kent Beck (XP & JUnit)*

# Quand faire du refactoring ?

## Lors des revues de code

- Beaucoup d'entreprises ont recours à des séances de revue de code (pour certaines, 1 à 2 heures chaque jour)
- ça permet à des développeurs expérimentés de communiquer/partager leurs connaissances
- Ce qui représente un code lisible pour l'un, ne l'est pas forcément pour l'autre (converger les points de vue)
- Refactorings : propositions concrètes de revue de code
- Assis côté à côté, les propositions sont encore plus efficaces (*pair programming*)

## En présence de branchement (dév. en //)

- Si une équipe de développeurs travaille sur plusieurs branches en parallèle et font des merges/intégrations à la fin de leur travail
- Les branchements doivent être les plus courts/brefs possibles pour ne pas trop diverger
- Dans l'approche par intégration continue (CI), les merges sont (au moins) quotidiens
- Cela réduit le risque que les refactorings cassent le code des autres (changer le nom d'une fonction, qui est utilisée dans d'autres branches ⇒ On s'en aperçoit vite)

## Refactoring automatisé

- Beaucoup d'outils/IDE, comme IntelliJ IDEA, Eclipse ou VS Code, proposent des refactorings automatisés
- Ces refactorings s'appuient sur l'arbre syntaxique des programmes et non sur le texte (ce qui les rend plus fiables)
- Certains refactorings sont semi-automatisés (l'outil vous propose une liste de modifications à valider)
- Ceci n'exclut pas le fait qu'il faut ré-exécuter les tests après chaque refactoring pour être sûr de son coup

# Plan du cours

1. Introduction générale

**2. Bad Smells**

3. Catégories de refactorings

3.1 Refactorings de base

3.2 Refactorings pour l'encapsulation

3.3 Refactorings pour le déplacement de features

3.4 Refactorings pour l'organisation des données

3.5 Refactorings pour la simplification de la logique

3.6 Refactorings pour les API

3.7 Refactorings pour traiter l'héritage

# Quand faire le refactoring ?

- Quand on retrouve dans le code des “*bad smells*”

## C'est quoi un “*bad smell*” ?

Une structure dans le code qui suggère une possibilité de faire du refactoring (ça “sent mauvais”, mais on n'est pas sûr que ça soit mauvais)

L'expérience humaine est très importante dans la détection de ces structures

## Exemples de *bad smells*

**Noms mystérieux :** Noms de variables, fonctions, classes, modules  
... doivent communiquer leur rôle et comment on peut  
s'en servir

- **Refactorings :** *Change Function Declaration, Rename Variable, et Rename Field*

**Code dupliqué :** La même structure qui se répète (pas exactement identique, mais qui ressemble beaucoup)

- **Refactorings :** *Extract Function, Slide Statements* (ré-ordonner les instructions, avant de faire une extraction), *Pull Up Method* (si le code dupliqué est dans deux sous-classes d'une même classe de base)

## Exemples de *bad smells*

**Fonction trop longue (*Large Function*) :** Les fonctions durables sont celles qui ne sont pas longues (facilement compréhensibles)

- **Heuristique** : quand on a envie d'écrire un commentaire pour expliquer ce que fait un bloc de code (sémantique propre), il faudrait définir ce bloc comme une fonction
- **Refactorings** : *Extract Function*, *Introduce Parameter Object* et *Preserve Whole Object* (pour regrouper une longue liste de paramètres), *Decompose Conditional* et *Replace Conditional with Polymorphism* si de longues conditionnelles, et *Split Loop* pour les boucles

## Exemples de *bad smells*

### Longue liste de paramètres :

- Refactorings : *Replace*

*Parameter with Query* (pour remplacer un paramètre par un accès à un autre paramètre), *Introduce Parameter Object*, *Combine Functions into Class* pour regrouper plusieurs fonctions dans une classe et transformer les paramètres communs en attributs, ...

### Données globales :

Des données touchées à différents endroits (variables globales, attributs et singlenton)

- Refactorings : *Encapsulate Variable* pour y accéder depuis des fonctions, dont les appels sont facilement identifiables, puis la déplacer pour limiter sa portée (la mettre dans une classe, par ex)

## Exemples de *bad smells*

**Données mutables** : C'est un problème lorsque la donnée a une grande portée

- **Refactorings** : *Encapsulate Variable* pour créer des fonctions d'accès à la donnée, *Split Variable* si la donnée est composée, ...

**Changement divergent** : 1 module qui peut être changé de différentes façons pour diverses raisons (changement non isolé par module)

- **Refactorings** : *Split Phase* Décomposer des traitements différents dans le module, ensuite *Extract Function* et enfin *Move Function* (vers 1 autre module)

## Exemples de *bad smells*

***Shotgun Surgery*** : Inverse du changement divergent (pour effectuer un changement, plein d'endroits doivent être touchés)

- **Refactorings** : *Move Function* et *Move Field*, *Combine Functions into Class* pour regrouper les différentes parties du code, et ensuite les décomposer pour avoir de petites fonctions, mais selon d'autres critères (métier)

***Feature Envy*** : Une fonction par exemple qui interagit beaucoup avec des fonctions et variables dans d'autres modules

- **Refactorings** : *Move Function*, parfois *Extract Function* puis *Move Function* pour extraire uniquement la partie de la fonction concernée

## Exemples de *bad smells*

**Agrégats de données (*Data Clumps*) :** des attributs ou des paramètres présents ensemble à plusieurs endroits

- **Refactorings :** *Extract Class* pour les regrouper dans une classe, *Introduce Parameter Object* pour les paramètres, ...

**Primitive Obsession :** Utilisation abusive des types primitifs (utiliser des nombres au lieu d'un type utilisateur Monnaie, ou des String pour des adresses, “*stringly typed*” variables) et ceci provoque souvent des erreurs de traitement/calcul

- **Refactorings :** *Replace Primitive with Object* et *Replace Type Code with Subclasses*

## Exemples de *bad smells*

**Switchs répétés :** Les longs if-then-else et switch, qui se répètent (à chaque ajout d'un else/case, il faut le faire partout)

- **Refactorings :** *Replace Conditional with Polymorphism* pour utiliser le polymorphisme de la POO à la place

**Boucles :** Certaines boucles peuvent être remplacées par des fonctions de première classe (*filter and map, ...*)

- **Refactorings :** *Replace Loop with Pipeline*

## Exemples de *bad smells*

**Lazy Element :** Ex : une classe avec une seule méthode par exemple (à coups de refactorings, sa taille a diminué, et la structure-classe devient obsolète

- **Refactorings :** *Inline Function*, *Inline Class* ou *Collapse Hierarchy* pour l'héritage

**Speculative Generality :** Du code prévu pour anticiper une certaine généralisation (paramètres d'une fonction, une classe abstraite, ...), qui devient au fil du temps inutile

- **Refactorings :** *Collapse Hierarchy*, *Change Function Declaration*, ou *Remove Dead Code* si par exemple le code était prévu pour des cas de tests (supprimer les cas de tests et ensuite appliquer ce refactoring)

## Exemples de *bad smells*

**Attribut temporaire :** Attribut dans une classe, qui n'est initialisé/utilisé que dans certains cas (rend le code difficile à comprendre)

- **Refactorings :** *Extract Class* pour créer une structure pour cet attribut orphelin et *Move Function* pour déplacer tout le code qui le concerne dans cette nouvelle classe

**Chaînes de messages :** Obtenir un objet (`getXXX()`), à qui on demande un autre objet, et ainsi de suite jusqu'à l'invocation d'une méthode métier (couplage fort dans toute la structure de chaînage)

- **Refactorings :** *Hide Delegate* à un certain niveau de la chaîne, ou bien *Extract Function* et *Move Function* pour déplacer le code utilisé par l'objet au début de la chaîne

## Exemples de *bad smells*

**Middle Man :** Une classe qui délègue beaucoup d'invocations de méthodes

- **Refactorings :** *Remove Middle Man* pour invoquer directement l'objet final

**Délit d'initié (*Insider Trading*) :** Couplage fort entre modules, qui savent trop de choses les uns sur les autres (une sous-classe qui sait trop de choses sur sa super-classe)

- **Refactorings :** *Replace Subclass with Delegate* ou *Replace Superclass with Delegate*

## Exemples de *bad smells*

**Classe trop grande (Large Class)** : Une classe avec beaucoup d'attributs ou de fonctions

- **Refactorings** : *Extract Class* pour encapsuler quelques attributs (ceux qui partagent des suffixes ou préfixes, par ex.), *Extract Superclass* ou *Replace Type Code with Subclasses* si ça correspond à une sous-classe

**Classes alternatives avec différentes interfaces** : Des classes qui se présentent comme alternatives, mais qui ont des interfaces différentes (pas de substitution possible)

- **Refactorings** : *Change Function Declaration* ou *Move Function*, puis potentiellement *Extract Superclass*

## Exemples de *bad smells*

**Classe de données :** Classes avec des attributs et des accesseurs uniquement

- **Refactorings :** *Encapsulate Record* pour transformer les attributs publics (en cachette, avant que personne ne voit ça), *Remove Setting Method* pour les attributs qui ne doivent pas être modifiés.
- Parfois, c'est intéressant de déplacer du comportement (*Extract/Move Function*) du client de cette classe vers la classe

## Exemples de *bad smells*

**Legs refusés (Refused Bequest) :** Des sous-classes qui héritent de méthodes et d'attributs dont elles ne se servent pas.  
Problème de conception de la hiérarchie

- **Refactorings :** *Push Down Method* et *Push Down Field* dans la super-classe pour créer une nouvelle classe fille (soeur de la classe qui refuse le legs)
- Parfois *Replace Subclass with Delegate* ou *Replace Superclass with Delegate* si la sous-classe refuse l'interface de la super-classe (erreur de conception de l'héritage)

## Exemples de *bad smells*

**Commentaires :** Ce n'est pas un vrai *bad smell*, mais plutôt un "déodorant" à un *smell* (les commentaires restent quelque chose de positif dans certains cas). Mais quand on trouve un commentaire, il faudrait voir si on ne peut pas extraire une fonction à partir du code commenté, et du coup, le commentaire devient superflus (on l'élimine donc)

- **Refactorings :** *Extract Function* ou *Change Function Declaration* pour changer le nom de la fonction si celle-ci existe déjà et qu'il faut encore commenter le code
- *Introduce Assertion* si on veut définir une règle sur l'état du système
- Un commentaire reste utile pour décrire des doutes, ou indiquer "le pourquoi"

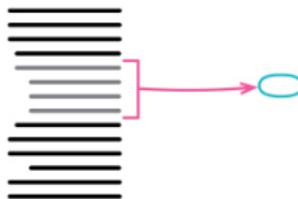
# Plan du cours

1. Introduction générale
2. Bad Smells
3. Catégories de refactorings
  - 3.1 Refactorings de base
  - 3.2 Refactorings pour l'encapsulation
  - 3.3 Refactorings pour le déplacement de features
  - 3.4 Refactorings pour l'organisation des données
  - 3.5 Refactorings pour la simplification de la logique
  - 3.6 Refactorings pour les API
  - 3.7 Refactorings pour traiter l'héritage

# Plan du cours

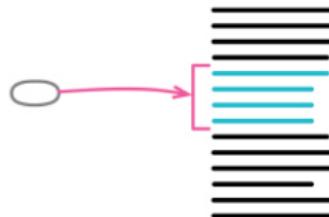
1. Introduction générale
2. Bad Smells
3. Catégories de refactorings
  - 3.1 Refactorings de base
  - 3.2 Refactorings pour l'encapsulation
  - 3.3 Refactorings pour le déplacement de features
  - 3.4 Refactorings pour l'organisation des données
  - 3.5 Refactorings pour la simplification de la logique
  - 3.6 Refactorings pour les API
  - 3.7 Refactorings pour traiter l'héritage

# Extract Function



```
function printOwing(invoice) {  
    printBanner();  
    let outstanding = calculateOutstanding();  
    printDetails(outstanding);  
  
    function printDetails(outstanding) {  
        console.log(`name: ${invoice.customer}`);  
        console.log(`amount: ${outstanding}`);  
    }  
}
```

# Inline Function



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

```
function moreThanFiveLateDeliveries(driver) {  
    return driver.numberOfLateDeliveries > 5;  
}
```



```
function getRating(driver) {  
    return (driver.numberOfLateDeliveries > 5) ? 2 : 1;  
}
```

# Extract Variable



```
return order.quantity * order.itemPrice -  
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +  
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
```



```
const basePrice = order.quantity * order.itemPrice;  
const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;  
const shipping = Math.min(basePrice * 0.1, 100);  
return basePrice - quantityDiscount + shipping;
```

# Inline Variable



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



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

# Change Declaration

The diagram illustrates a common pitfall in programming where variable declarations and assignments are confused. It shows two separate declarations: `f(x, y, z)` and `g`, each pointing to its own function definition below it. The first declaration `f(x, y, z)` points to `f(x, y, z) { }` . The second declaration `g` points to `f(x, y, z) { }`.

```
f(x, y, z)
g
f(x, y, z) { }
```

```
function circum(radius) {...}
```



```
function circumference(radius) {...}
```

# Introduce Parameter Object

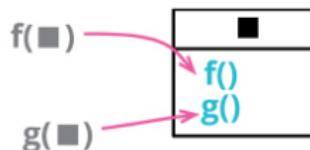


```
function amountInvoiced(startDate, endDate) {...}  
function amountReceived(startDate, endDate) {...}  
function amountOverdue(startDate, endDate) {...}
```



```
function amountInvoiced(aDateRange) {...}  
function amountReceived(aDateRange) {...}  
function amountOverdue(aDateRange) {...}
```

# Combine Functions into Class

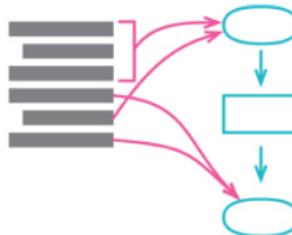


```
function base(aReading) {...}  
function taxableCharge(aReading) {...}  
function calculateBaseCharge(aReading) {...}
```



```
class Reading {  
    base() {...}  
    taxableCharge() {...}  
    calculateBaseCharge() {...}  
}
```

## Split Phase



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```



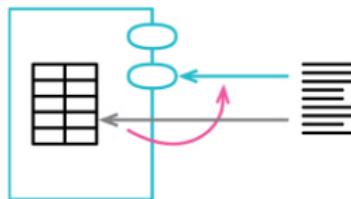
```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

# Plan du cours

- 1. Introduction générale**
- 2. Bad Smells**
- 3. Catégories de refactorings**
  - 3.1 Refactorings de base**
  - 3.2 Refactorings pour l'encapsulation**
  - 3.3 Refactorings pour le déplacement de features**
  - 3.4 Refactorings pour l'organisation des données**
  - 3.5 Refactorings pour la simplification de la logique**
  - 3.6 Refactorings pour les API**
  - 3.7 Refactorings pour traiter l'héritage**

# Encapsulate Record

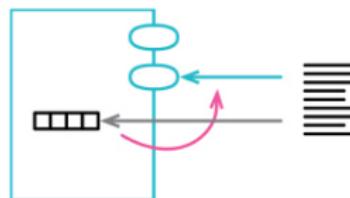


```
organization = {name: "Acme Gooseberries", country: "GB"};
```



```
class Organization {
    constructor(data) {
        this._name = data.name;
        this._country = data.country;
    }
    get name()    {return this._name;}
    set name(arg) {this._name = arg;}
    get country() {return this._country;}
    set country(arg) {this._country = arg;}
}
```

# Encapsulate Collection



```
class Person {  
    get courses() {return this._courses;}  
    set courses(aList) {this._courses = aList;}}
```



```
class Person {  
    get courses() {return this._courses.slice();}  
    addCourse(aCourse) { ... }  
    removeCourse(aCourse) { ... }}
```

## Replace Temp with Query

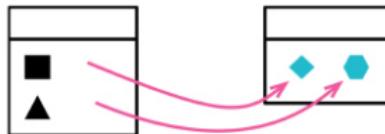
```
a ←  
a() {  
}  
}
```

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



```
get basePrice() {this._quantity * this._itemPrice;}  
...  
if (this.basePrice > 1000)  
    return this.basePrice * 0.95;  
else  
    return this.basePrice * 0.98;
```

# Extract Class

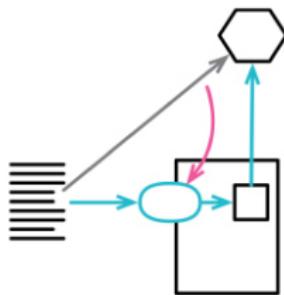


```
class Person {  
    get officeAreaCode() {return this._officeAreaCode;}  
    get officeNumber() {return this._officeNumber;}}
```



```
class Person {  
    get officeAreaCode() {return this._telephoneNumber.areaCode;}  
    get officeNumber() {return this._telephoneNumber.number;}  
}  
class TelephoneNumber {  
    get areaCode() {return this._areaCode;}  
    get number() {return this._number;}  
}
```

## Hide Delegate



```
manager = aPerson.department.manager;
```



```
manager = aPerson.manager;  
  
class Person {  
    get manager() {return this.department.manager;}}
```

# Plan du cours

1. Introduction générale
2. Bad Smells
3. Catégories de refactorings
  - 3.1 Refactorings de base
  - 3.2 Refactorings pour l'encapsulation
  - 3.3 Refactorings pour le déplacement de features**
  - 3.4 Refactorings pour l'organisation des données
  - 3.5 Refactorings pour la simplification de la logique
  - 3.6 Refactorings pour les API
  - 3.7 Refactorings pour traiter l'héritage

# Move Function



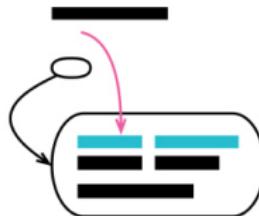
```
class Account {  
    get overdraftCharge() {...}
```



```
class AccountType {  
    get overdraftCharge() {...}
```

Même chose pour Move Field

# Move Statements into Function

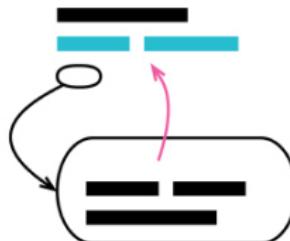


```
result.push(`<p>title: ${person.photo.title}</p>`);  
result.concat(photoData(person.photo));  
  
function photoData(aPhoto) {  
  return [  
    `<p>location: ${aPhoto.location}</p>`,  
    `<p>date: ${aPhoto.date.toDateString()}</p>`,  
  ];  
}
```



```
result.concat(photoData(person.photo));  
  
function photoData(aPhoto) {  
  return [  
    `<p>title: ${aPhoto.title}</p>`,  
    `<p>location: ${aPhoto.location}</p>`,  
    `<p>date: ${aPhoto.date.toDateString()}</p>`,  
  ];  
}
```

# Move Statements to Callers



```
emitPhotoData(outStream, person.photo);

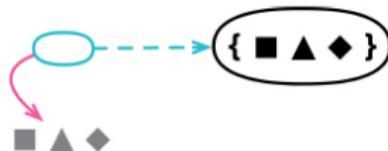
function emitPhotoData(outStream, photo) {
    outStream.write(`<p>title: ${photo.title}</p>\n`);
    outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```



```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location: ${person.photo.location}</p>\n`);

function emitPhotoData(outStream, photo) {
    outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

# Replace Inline Code with Function Call

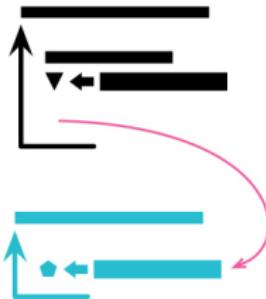


```
let appliesToMass = false;  
for(const s of states) {  
    if (s === "MA") appliesToMass = true;  
}
```



```
appliesToMass = states.includes("MA");
```

# Split Loops (meilleure compréhension)



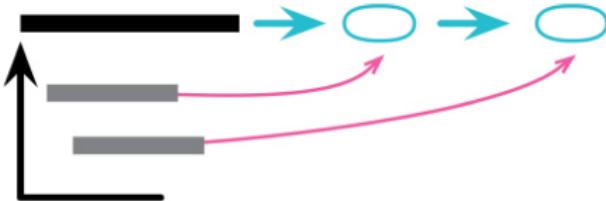
```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```



```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

# Replace Loop with Pipeline



```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```



```
const names = input
  .filter(i => i.job === "programmer")
  .map(i => i.name)
  ;
```

# Remove Dead Code



```
if(false) {  
    doSomethingThatUsedToMatter();  
}
```



# Plan du cours

- 1. Introduction générale**
- 2. Bad Smells**
- 3. Catégories de refactorings**
  - 3.1 Refactorings de base**
  - 3.2 Refactorings pour l'encapsulation**
  - 3.3 Refactorings pour le déplacement de features**
  - 3.4 Refactorings pour l'organisation des données**
  - 3.5 Refactorings pour la simplification de la logique**
  - 3.6 Refactorings pour les API**
  - 3.7 Refactorings pour traiter l'héritage**

## Split Variable



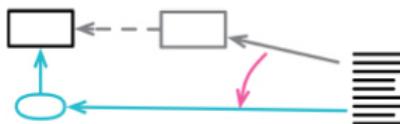
```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```



```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Beaucoup de variables ont une seule responsabilité (on stocke une valeur dedans pour la référencer plus tard, et on ne la change pas)

# Replace Derived Variable with Query



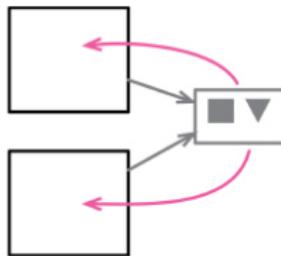
```
get discountedTotal() {return this._discountedTotal;}
set discount(aNumber) {
    const old = this._discount;
    this._discount = aNumber;
    this._discountedTotal += old - aNumber;
}
```



```
get discountedTotal() {return this._baseTotal - this._discount;}
set discount(aNumber) {this._discount = aNumber;}
```

Remplacer l'accès à une variable qui stocke une valeur calculée à partir d'autres variables par un appel de fonction nouvelle, qui contient ce calcul (style prog. fonctionnelle)

## Change Reference to Value



```
class Product {  
    applyDiscount(arg) {this._price.amount -= arg;}
```



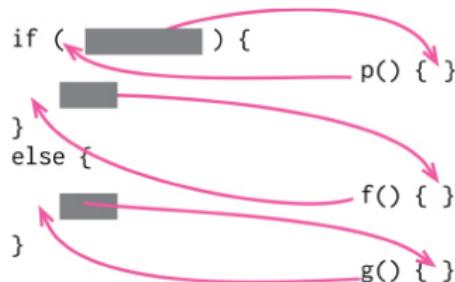
```
class Product {  
    applyDiscount(arg) {  
        this._price = new Money(this._price.amount - arg, this._price.currency);  
    }  
}
```

Même chose pour Change Value to Reference

# Plan du cours

- 1. Introduction générale**
- 2. Bad Smells**
- 3. Catégories de refactorings**
  - 3.1 Refactorings de base**
  - 3.2 Refactorings pour l'encapsulation**
  - 3.3 Refactorings pour le déplacement de features**
  - 3.4 Refactorings pour l'organisation des données**
  - 3.5 Refactorings pour la simplification de la logique**
  - 3.6 Refactorings pour les API**
  - 3.7 Refactorings pour traiter l'héritage**

## Decompose Conditional



```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
    charge = quantity * plan.summerRate;
else
    charge = quantity * plan.regularRate + plan.regularServiceCharge;
```



```
if (summer())
    charge = summerCharge();
else
    charge = regularCharge();
```

# Consolidate Conditional Expression

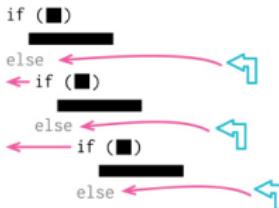
```
if (■) ⊕  
if (△) ⊕  
if (▼) ⊕  
  
↓  
if (p()) ⊕
```

```
if (anEmployee.seniority < 2) return 0;  
if (anEmployee.monthsDisabled > 12) return 0;  
if (anEmployee.isPartTime) return 0;
```



```
if (isNotEligableForDisability()) return 0;  
  
function isNotEligableForDisability() {  
    return ((anEmployee.seniority < 2)  
        || (anEmployee.monthsDisabled > 12)  
        || (anEmployee.isPartTime));  
}
```

# Replace Nested Conditional with Guard Clauses

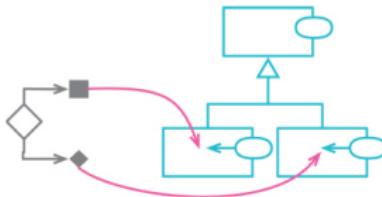


```
function getPayAmount() {  
    let result;  
    if (isDead)  
        result = deadAmount();  
    else {  
        if (isSeparated)  
            result = separatedAmount();  
        else {  
            if (isRetired)  
                result = retiredAmount();  
            else  
                result = normalPayAmount();  
        }  
    }  
    return result;  
}
```



```
function getPayAmount() {  
    if (isDead) return deadAmount();  
    if (isSeparated) return separatedAmount();  
    if (isRetired) return retiredAmount();  
    return normalPayAmount();  
}
```

# Replace Conditional with Polymorphism



```
switch (bird.type) {
    case 'EuropeanSwallow':
        return "average";
    case 'AfricanSwallow':
        return (bird.numberOfCoconuts > 2) ? "tired" : "average";
    case 'NorwegianBlueParrot':
        return (bird.voltage > 100) ? "scorched" : "beautiful";
    default:
        return "unknown";
```



```
class EuropeanSwallow {
    get plumage() {
        return "average";
    }
}
class AfricanSwallow {
    get plumage() {
        return (this.numberOfCoconuts > 2) ? "tired" : "average";
    }
}
class NorwegianBlueParrot {
    get plumage() {
        return (this.voltage > 100) ? "scorched" : "beautiful";
    }
}
```

# Introduce Assertion

assert (assumption)



```
if (this.discountRate)
    base = base - (this.discountRate * base);
```



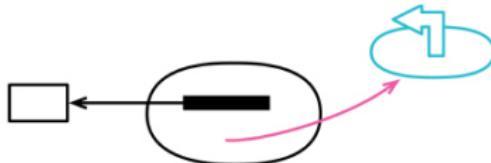
```
assert(this.discountRate >= 0);
if (this.discountRate)
    base = base - (this.discountRate * base);
```

Rendre explicite une hypothèse implicite

# Plan du cours

- 1. Introduction générale**
- 2. Bad Smells**
- 3. Catégories de refactorings**
  - 3.1 Refactorings de base**
  - 3.2 Refactorings pour l'encapsulation**
  - 3.3 Refactorings pour le déplacement de features**
  - 3.4 Refactorings pour l'organisation des données**
  - 3.5 Refactorings pour la simplification de la logique**
  - 3.6 Refactorings pour les API**
  - 3.7 Refactorings pour traiter l'héritage**

## Separate Query from Modifier



```
function getTotalOutstandingAndSendBill() {  
    const result = customer.invoices.reduce((total, each) => each.amount + total, 0);  
    sendBill();  
    return result;  
}
```



```
function totalOutstanding() {  
    return customer.invoices.reduce((total, each) => each.amount + total, 0);  
}  
function sendBill() {  
    emailGateway.send(formatBill(customer));  
}
```

Enrichir l'API avec des fonctions, avec et sans effet de bord

## Parameterize Function

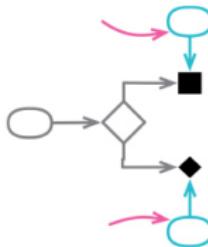
f() { }

```
function tenPercentRaise(aPerson) {
    aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
    aPerson.salary = aPerson.salary.multiply(1.05);
}
```



```
function raise(aPerson, factor) {
    aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

## Remove Flag Argument



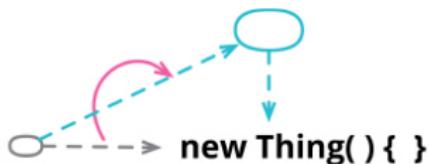
```
function setDimension(name, value) {  
    if (name === "height") {  
        this._height = value;  
        return;  
    }  
    if (name === "width") {  
        this._width = value;  
        return;  
    }  
}
```



```
function setHeight(value) {this._height = value;}  
function setWidth (value) {this._width = value;}
```

Un Flag argument est un argument qui change le comportement de la fonction

## Replace Constructor with Factory Function



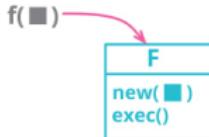
```
leadEngineer = new Employee(document.leadEngineer, 'E');
```



```
leadEngineer = createEngineer(document.leadEngineer);
```

Un constructeur est moins compréhensible qu'une fonction factory.  
Il a un nom rigide

## Replace Function with Command



```
function score(candidate, medicalExam, scoringGuide) {  
    let result = 0;  
    let healthLevel = 0;  
    // long body code  
}
```



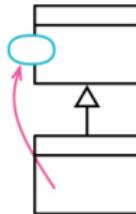
```
class Scorer {  
    constructor(candidate, medicalExam, scoringGuide) {  
        this._candidate = candidate;  
        this._medicalExam = medicalExam;  
        this._scoringGuide = scoringGuide;  
    }  
  
    execute() {  
        this._result = 0;  
        this._healthLevel = 0;  
        // long body code  
    }  
}
```

Command = objet dans le patron Commande, avec une méthode execute ou call (nouvelle classe facilement extensible)

# Plan du cours

- 1. Introduction générale**
- 2. Bad Smells**
- 3. Catégories de refactorings**
  - 3.1 Refactorings de base**
  - 3.2 Refactorings pour l'encapsulation**
  - 3.3 Refactorings pour le déplacement de features**
  - 3.4 Refactorings pour l'organisation des données**
  - 3.5 Refactorings pour la simplification de la logique**
  - 3.6 Refactorings pour les API**
  - 3.7 Refactorings pour traiter l'héritage**

## Pull Up Method



```
class Employee {...}

class Salesman extends Employee {
    get name() {...}
}

class Engineer extends Employee {
    get name() {...}
}
```

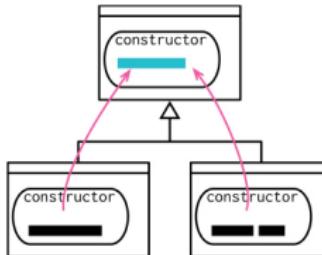


```
class Employee {
    get name() {...}
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

Même chose pour Push Down Method et Pull-Up/Push-Down Field

# Pull Up Constructor Body



```
class Party {...}

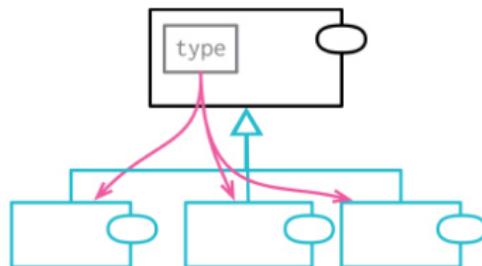
class Employee extends Party {
    constructor(name, id, monthlyCost) {
        super();
        this._id = id;
        this._name = name;
        this._monthlyCost = monthlyCost;
    }
}
```



```
class Party {
    constructor(name){
        this._name = name;
    }
}

class Employee extends Party {
    constructor(name, id, monthlyCost) {
        super(name);
        this._id = id;
        this._monthlyCost = monthlyCost;
    }
}
```

## Replace Type Code with Subclasses

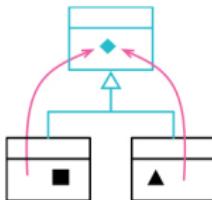


```
function createEmployee(name, type) {  
    return new Employee(name, type);  
}
```



```
function createEmployee(name, type) {  
    switch (type) {  
        case "engineer": return new Engineer(name);  
        case "salesman": return new Salesman(name);  
        case "manager": return new Manager (name);  
    }  
}
```

# Extract Superclass



```
class Department {  
    get totalAnnualCost() {...}  
    get name() {...}  
    get headCount() {...}  
}
```

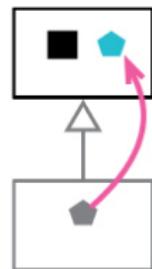
```
class Employee {  
    get annualCost() {...}  
    get name() {...}  
    get id() {...}  
}
```

```
class Party {  
    get name() {...}  
    get annualCost() {...}  
}
```

```
class Department extends Party {  
    get annualCost() {...}  
    get headCount() {...}  
}
```

```
class Employee extends Party {  
    get annualCost() {...}  
    get id() {...}  
}
```

# Collapse Hierarchy

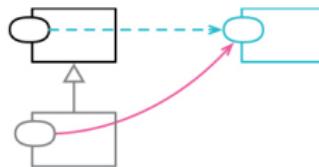


```
class Employee {...}  
class Salesman extends Employee {...}
```



```
class Employee {...}
```

# Replace Subclass with Delegate



```
class Order {
    get daysToShip() {
        return this._warehouse.daysToShip;
    }
}

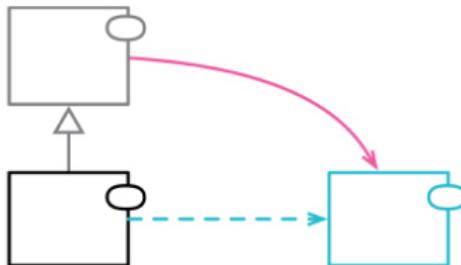
class PriorityOrder extends Order {
    get daysToShip() {
        return this._priorityPlan.daysToShip;
    }
}
```



```
class Order {
    get daysToShip() {
        return (this._priorityDelegate)
            ? this._priorityDelegate.daysToShip
            : this._warehouse.daysToShip;
    }
}

class PriorityOrderDelegate {
    get daysToShip() {
        return this._priorityPlan.daysToShip
    }
}
```

## Replace Superclass with Delegate



```
class List {...}  
class Stack extends List {...}
```



```
class Stack {  
    constructor() {  
        this._storage = new List();  
    }  
}  
class List {...}
```

## Et pour finir ... que propose IDEA comme refactoring ?

- Outil interactif : donner les arguments nécessaires au refactoring, visualiser un aperçu des changements, ...
- Opérations de refactoring (menu Refactor) :
  - Rename ...
  - Move ... (Method, Field, ...)
  - Pull ... Up et Push ... Down ...
  - Inline ...
  - Extract/Introduce Interface, Superclass, ...
  - Convert raw types to generics
  - ...
- Plus d'infos :

<https://www.jetbrains.com/help/idea/refactoring-source-code.html>

# Pour aller plus loin

