Tutoriel sur les mauvaises odeurs

En génie logiciel, les mauvaises odeurs sont des mauvaises pratiques de conception logicielle qui conduisent à des problèmes de maintenance, compréhension du code. Ces problèmes sont souvent issus d'un mauvais choix d'implémentations ou de conceptions et conduisent à une complexification du code source et de la maintenance et évolutivité de celui-ci. Afin de corriger un code smell, il est nécessaire de procéder à une restructuration du code source, c'est-à-dire modifier le code sans en altérer son comportement.

Ce document a pour objectif de vous aider à comprendre certaines mauvaises odeurs et les actions correctives à entreprendre pour les corriger. Vous trouverez ci-dessous une liste de mauvaises odeurs avec les symptômes et des exemples de restructurations.

1. Méthode longue

1.1. *Signes et symptômes*

Une méthode contient trop de lignes de code. Généralement, toute méthode de plus de dix lignes devrait vous inciter à commencer à poser des questions.

1.2. Restructuration

- Isoler un fragment de code dans une nouvelle méthode : Mettre le fragment dans une méthode dont le nom explique le but de la méthode

Problème	Solution

`void printOwing() {`
`  printBanner();`
 	
`  //print details`
`  cout<<"name: "<<name<<"\n";`
`  cout<<"amount: "<<`
`getOutstanding()<<"\n";`
`}`
	`void printOwing() {`
	` printBanner();`
	` printDetails(getOutstanding());`
	`}`
	`void printDetails(double outstanding){`
	`  cout<<"name: "<< name<<"\n";`
	`  cout<<"amount: "<<outstanding<<"\n";`
	`}`

- Remplacer variable temporaire par requête : Si une variable temporaire est utilisée pour contenir le résultat d'une expression, on pourrait extraire cette expression et la mettre dans une méthode. Remplacer toutes les occurrences de la variable temporaire par un appel à la méthode. La nouvelle méthode pourra ainsi être utilisée par d'autres méthodes.

Problème	Solution

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

- Conserver objet entier : Au lieu d'extraire plusieurs valeurs d'un objet, qui sont ensuite passées en paramètre à une méthode, on passe l'objet entier à la méthode.

Problème	Solution

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

2. Longue liste de paramètres

2.1. *Signes et symptômes*

Plus de trois ou quatre paramètres pour une méthode.

2.2. Restructuration

- Remplacer paramètre par l'appel d'une méthode : Au lieu d'appeler une méthode et passer son résultat à une autre méthode, on laisse ce dernier faire l'appel elle-même pour obtenir le résultat.

Problème	Solution

`int basePrice = quantity * itemPrice;`
`double seasonDiscount = this.getSeasonalDiscount();`
`double fees = this.getFees();`
`double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);`
	`int basePrice = quantity * itemPrice;`
	`double finalPrice = discountedPrice(basePrice);`

- Conserver objet entier : Au lieu d'extraire plusieurs valeurs d'un objet, qui sont ensuite passées en paramètre à une méthode, on passe l'objet à la méthode.

Problème	Solution

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

3. Code mort

3.1. *Signes et symptômes*

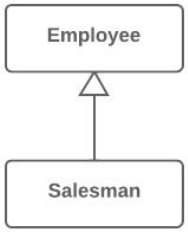
Une variable, un paramètre, un champ, une méthode ou une classe n'est plus utilisé (généralement parce qu'il est obsolète).

3.2. Restructuration

- Supprimer le paramètre /attribut : Si un paramètre ou un attribut n'est plus utilisé dans la méthode/classe, il faut le supprimer.

Problème	Solution
`display(Data);`	`display();`

- Réduire la hiérarchie : Si une sous classe est pratiquement identique à sa classe mère, on supprime la sous classe.

Problème	Solution
	

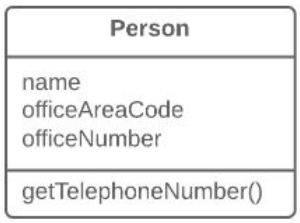
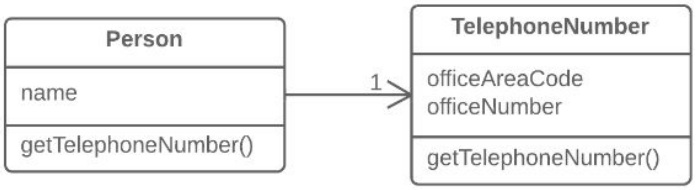
4. Grosse classe

4.1. *Signes et symptômes*

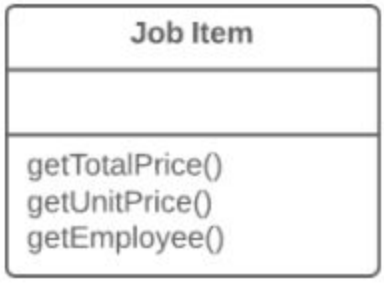
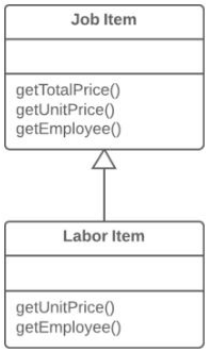
Une classe contient de nombreux champs / méthodes / lignes de code.

4.2. Restructuration

- Extraire classe : lorsqu'on se retrouve avec une classe qui fait le travail qui devrait être fait par deux classes. Il faut créer une nouvelle classe et déplacer les attributs et méthodes pertinents vers la nouvelle classe.

Problème	Solution
	

- Extraire sous-classe : Une classe a des méthodes qui ne sont utilisées que dans certaines instances. Alors il faut créer une sous-classe pour ce sous-ensemble de méthodes.

Problème	Solution
	

5. Code dupliqué

5.1. *Signes et symptômes*

Deux ou plusieurs fragments de code semblent presque identiques.

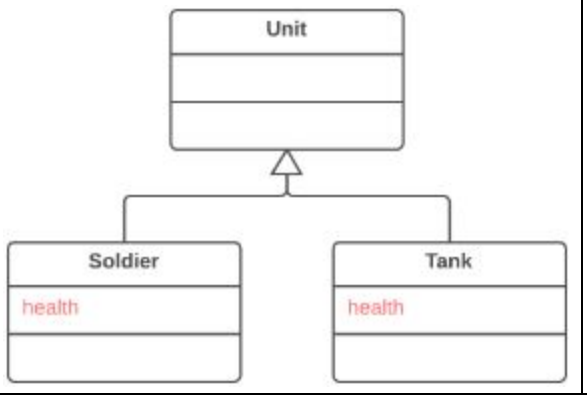
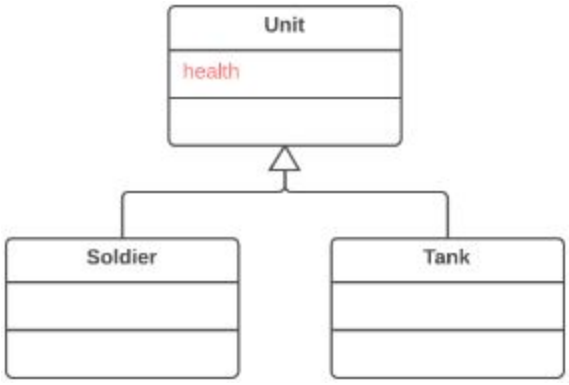
5.2. Restructuration

- extraire méthode : Mettre le fragment (qui se ressemble) dans une méthode dont le nom explique le but de la méthode

Problème	Solution

`void printOwing() {`
`  printBanner();`
 	
`  //print details`
`  cout<<"name: "<<name<<"\n";`
`  cout<<"amount: "<<`
`getOutstanding()<<"\n";`
`}`
	`void printOwing() {`
	` printBanner();`
	` printDetails(getOutstanding());`
	`}`
	`void printDetails(double outstanding){`
	`  cout<<"name: "<< name<<"\n";`
	`  cout<<"amount: "<<outstanding<<"\n";`
	`}`

- Monter attribut : Si deux ou plusieurs sous-classes ont le même attribut, il faut déplacer l'attribut dans la super-class

Problème	Solution
	

- Changer algorithme : Si la méthode est difficile à comprendre, alors il faut changer l'implémentation par celle d'un autre algorithme.

Problème	Solution

`string foundPerson(vector<string> people){`
`for (int i = 0; i < people.size(); i++) {`
`  if (people[i] == "Don")`
`  return "Don";`
`  if (people[i] == "John")`
`  return "John";`
`  if (people[i] == "Kent")`
`  return "Kent";`
`  }`
`return "";`
`}`
	`string foundPerson(vector<string> people){`
	`vector<string> candidates{"Don", "John", "Kent"};`
	`for (int i = 0; i < people.size(); i++) {`
	`if(find(candidates.begin(),candidates_.end(),*i) != candidates.end())`
	`  return *i;`
	`}`
	`return "";`
	`}`

Référence:

1. https://refactoring.guru
2. http://www.professeurs.polymtl.ca/michel.gagnon/Smells/
3. https://fr.wikipedia.org/wiki/Code_smell