Noms Significatifs

Ron Jeffries (C#)

- Règles de code simple de Beck
  - ➔ Passer tous les tests
  - ➔ Ne pas être **redondant**
  - ➔ Exprimer toutes les idées de conception présentes dans le système
  - ➔ Minimiser le nombre d'entités (classes/méthodes/fonctions assimilées)
- Le code doit être simple et doit faire penser que le langage était adapté au problème

Redondance

- Lorsque la même chose se répète plusieurs fois : l'idée n'est pas parfaitement représentée dans le code
  - ➔ Il fait essayer de la représenter plus clairement
- L'expressivité se fonde sur des noms significatifs
  - ➔ Bien choisir le nom de ses variables/fonctions
  - ➔ Vérifier qu'un objet/méthode n'a pas de plusieurs rôles
    - → L'objet devra certainement être décomposé en plusieurs objets
    - → La méthode doit s'exprimer plus clairement/des méthodes complémentaires peuvent indiquer comment elle procède

I Noms significatifs

1) Choisir des noms révélateurs des intentions

- Ne pas hésiter à changer de nom si on en trouve un meilleur
- Nom d'une variable/fonction/classe :
  - ➔ Raison de son existence
  - ➔ Rôle
  - ➔ Utilisation
- Si un nom exige un commentaire c'est qu'il ne répond pas à ces questions

2) Eviter la désinformation

- Eviter les mots dont le sens établi est différent du sens voulu (hp pour hypoténuse par ex)
- l et O ne doivent pas servir comme noms de variables ressemblant trop au 1 et au 0

3) Faire des distinctions significatives

- Impossible d'utiliser le même nom pout faire référence à 2 choses différentes
  - ➔ La compilation ne serait plus possible
- Il ne suffit pas d'ajouter des numéros/noms parasites
  - ➔ Les noms doivent être différents pour représenter des choses différentes
- Classe **Product** si on crée **ProductInfo** ou **ProductData**
  - ➔ On a choisi des noms différents sans qu'ils ne représentent quelque chose de différent
  - ➔ **Info** et **Data** sont des mots parasites vagues
- Les mots parasites sont redondants
- Le mot **variable** ne doit JAMAIS apparaître dans une variable et **table** JAMAIS dans un tableau

4) Choisir des compatibles avec une recherche

- Les noms d'une seule lettre/constantes numériques : difficiles à localiser

5) Eviter la codification

- Les noms codifiés sont rarement prononçables/sujets aux erreurs de saisie
- Inutile de préfixer les variables membres par un m :

```
public class Part {
 private String m_dsc; // La description textuelle.
void setName(String name) {
m_dsc = name;
        }
}
```

```
public class Part {
String description;
void setDescription(String description) {
 this.description = description;
        }

}
```

6) Interfaces/implementations

- Création d'une FABRIQUE ABSTRAITE pour créer des formes : interface implémentée par une classe concrète
  ➔ La nommer **IShapeFactory** ou **ShapeFactory** ? (I : interface)
- En soit on s'en fout que les utilisateurs sachent que c'est une interface
  ➔ On veut juste qu'ils sachent qu'il existe un **ShapeFactory**
- Si je dois codifier l'interface ou l'implémentation, je choisis l'implémentation
  ➔ **ShapeFactoryImp**

7) Eviter les associations mentales

- Ne pas obliger le lecteur à convertir mentalement nos noms en noms qu'il connaît
- Noms de variables en une seule lettre sont problématiques
  ➔ Sauf pour les compteurs de boucle (i, j ou k mais **PAS L**)

8) Noms de classes

- Classes/objets : choisir des noms ou groupes nominaux (**customer**, **WikiPage**, **Account**, **AdressParser**)
  ➔ Préférable d'éviter **Manager**, **Processor**, **Data** ou **Info**
- Ne doit pas être un verbe

9) Nom des méthodes

- Verbes/groupes verbaux (**postPayement**, **delatePage**, **save**)
- Accesseurs/mutateurs/prédicats nommés d'après leur valeur & préfixés par **get/set/is**

  string name = employee.getName();

  customer.setName("mike");

  if (paycheck.isPosted())...
- Quand les constructeurs sont surchargés : méthodes de fabrique statiques avec des noms décrivant les arguments

  Complex fulcrumPoint = Complex.FromRealNumber(23.0);

  ➔ Préférable à : ***Complex fulcrumPoint = new Complex(23.0);***
- Pour imposer l'emploi de ces méthodes de fabrique : constructeurs correspondants doivent être rendus privés

10) Choisir un mot par concept

- Déroutant d'avoir **fetch**/**get** et **retrieve** pour représenter des méthodes équivalentes dans des classes différentes
- Pareil pour **controller**/**manager**/**driver** dans la même base de code
  - ➔ Quelle est la différence entre **DeviceManager** et **ProtocolController** ?
  - ➔ Pourquoi les 2 ne sont pas des **Controller** ou des **Manager** et pourquoi pas des **Driver** ?
  - ➔ Suppose qu'on va avoir 2 objets de type différent/classe différentes

11) Choisir des noms dans le domaine de la solution

- Employer des termes informatiques, des noms d'algorithmes, des noms de motifs, des termes mathématiques…
- Donner des noms techniques : généralement la meilleure option

12) Choisir des noms dans le domaine du problème

- S'il n'existe aucun terme informatique : utiliser le nom issu du domaine du problème
- Séparation concepts du domaine de la solution/du problème fait partie du travail du bon programmateur/concepteur
- Code fortement lié aux concepts du domaine du problème doit employer des noms tirés de ce domaine

13) Ajouter un contexte significatif

- Redonner aux noms leur contexte en les englobant dans des classes/fonctions aux noms appropriés
- En dernier ressort on peut ajouter un préfixe
- **firstName**, **lastName**, **houseNumber**, **city**, **state** et **zipcode**
  - ➔ En considérant l'ensemble : forme une adresse
  - ➔ Si on rencontre uniquement **state** on en déduit pas que ça fait partie d'une adresse
- Ajouter des contextes en ajoutant des préfixes :
  - ➔ **addrFirstName**, **addrLastName**, **adrrState**… : on comprend qu'elles font partie d'une structure plus vaste
- LA meilleure solution : créer une classe **Address** (même le compilateur sait que la variable appartient à un concept plus vaste)

Exemple

```
private void printGuessStatistics(char candidate, int count) {
 String number;
String verb;
 String pluralModifier;
 if (count == 0) {
 number = "no"; verb = "are"; pluralModifier = "s";
} else if (count == 1) {
number = "1";
verb = "is";
pluralModifier = "";
} else {
number = Integer.toString(count);
verb = "are";
pluralModifier = "s";
}
String guessMessage = String.format(
"There %s %s %s%s", verb, number, candidate, pluralModifier
 );
print(guessMessage);
}
```

- La fonction n'est pas vraiment courte
- Pour décomposer la fonction en éléments plus petits : créer la classe **GuessStatisticsMessage** et transformer ces 3 variables en champs dans cette classe (pour leur attribuer un contexte clair)

```
public class GuessStatisticsMessage {
private String number;
 private String verb;
private String pluralModifier;
public String make(char candidate, int count) {
createPluralDependentMessageParts(count);
return String.format(
"There %s %s %s%s",
verb, number, candidate, pluralModifier );
}

private void createPluralDependentMessageParts(int count) {
 if (count == 0) {
thereAreNoLetters();
} else if (count == 1) {
thereIsOneLetter();
} else {
thereAreManyLetters(count);
            }
}

private void thereAreManyLetters(int count) {
 number = Integer.toString(count);
verb = "are";
pluralModifier = "s";
}

private void thereIsOneLetter() {
number = "1";
verb = "is";
 pluralModifier = "";
}

private void thereAreNoLetters() {
 number = "no";
verb = "are";
pluralModifier = "s";
            }
 }
```

14) Ne pas ajouter de contexte inutile

- Dans une application fictive appelée « **Gas Sation Deluxe** » : déconseillé de préfixer chaque classe par GSD
- On crée une classe **MailingAddress** dans un module de comptabilité de **GSD** qu'on nomme **GSDAccountAddress**
  - ➔ Plus tard on a besoin d'une adresse postale pour l'application de gestion de contacts clients
  - ➔ Va-t-on utiliser **GSDAccountAddress** ?
- Préférable de choisir des noms courts tant qu'ils restent clairs (ne pas ajouter de contexte inutile)
- **accountAddress** et **customerAddress** sont parfaits pour les instances de la classe **Address** mais pas pour des noms de classes
- **Address** convient parfaitement pour un nom de classe
- S'il faut différencier des adresses postales, MAC et web
  - ➔ **PostalAdress**, **MAC** et **URI** (plus précis)