Table des matières

Appendix

Style de programmation et qualité du code

1 Introduction

Ce document présente quelques rappels[1] sur les caractéristiques d'un programme bien écrit — un programme écrit dans un bon style de programmation. Il présente tout d'abord quelques principes généraux. Il présente ensuite les notions d'abstraction procédurale et de couplage. Finalement, il présente divers contre-exemples et exemples, exprimés pour la plupart en Ruby, illustrant concrètement quelques-unes de ces mauvaises, ou bonnes, caractéristiques.

[1] INF3135 ?

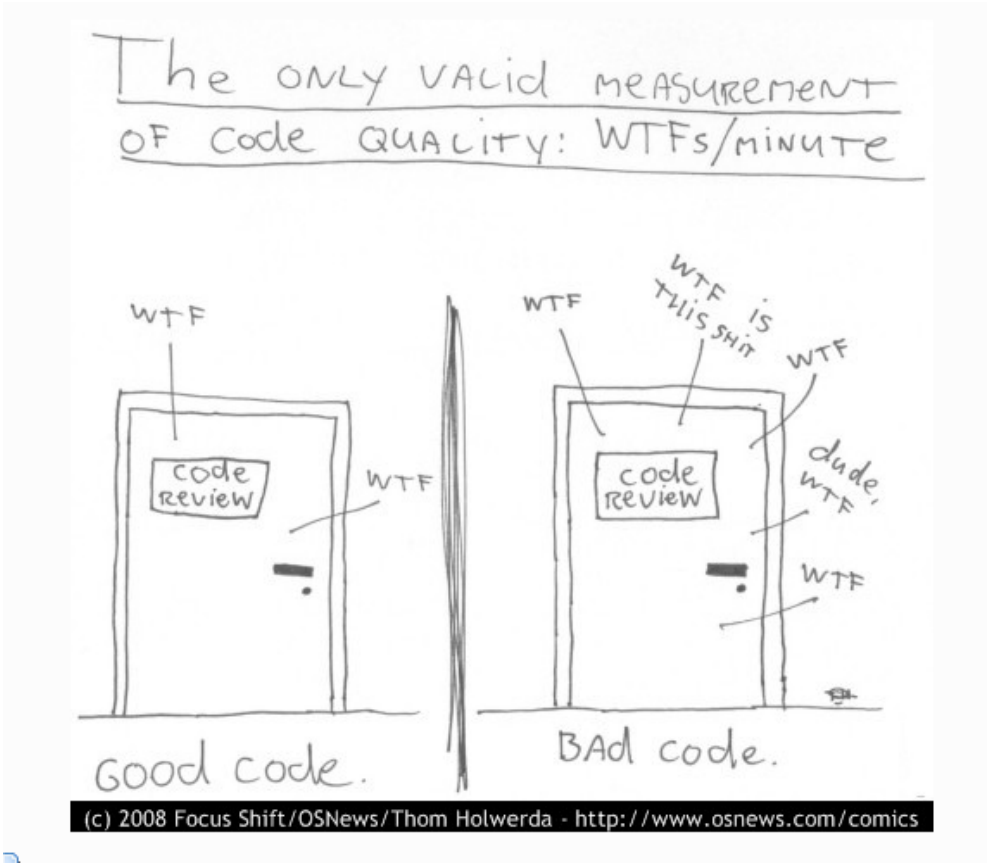

Figure .1: La seule bonne mesure de qualité!

2 Quelques principes généraux

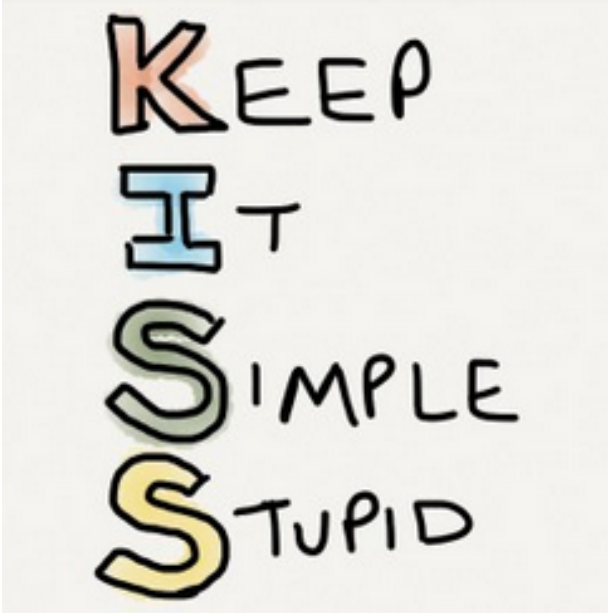

Figure .2: KISS!

Figure .3: DRY!

2.1 Principe KISS et principes reliés

- Principe KISS[2]
  - *Keep It Simple, Stupid!*
  - *Keep It Short and Simple!*
- Rasoir d'Occam[3] :
  - «Les hypothèses les plus simples sont les plus vraisemblables».
- Maxime attribuée à A. Einstein[4] :

  «*Everything should be made as simple as possible, but no simpler*».
- Maxime attribuée à Antoine de St-Exupéry[5][6] :

  «La perfection est atteinte non pas quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retirer.»
- Citation de C.A.R. Hoare (*The Emperor's Old Clothes*, CACM, February 1981) :

  «There are two ways of constructing a software design. One is to ***make it so simple*** *that there are obviously no deficiencies; the other is to* ***make it so complicated*** *that there are no obvious deficiencies.* ***The first method is far more difficult****.»*

[2] `http://en.wikipedia.org/wiki/KISS_principle`
[3] `http://fr.wikipedia.org/wiki/Rasoir_d'Ockham`
[4] `http://en.wikipedia.org/wiki/KISS_principle`
[5] Auteur du «Petit Prince»
[6] `http://www.drop-zone-city.com/article.php3?id_article=221`

2.2 Principe DRY et principes reliés

- DRY = *Don't Repeat Yourself*
- Formulation de A. Hunt & D. Thomas (*The Pragmatic Programmer—From Journeyman to Master*, 2000) :

  > *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

- Autre formulation équivalente = *Once and Only Once*[7] *:*

  > Each and every declaration of behavior should appear `OnceAndOnlyOnce`.

2.3 Règles de E. Raymond

Quelques règles suggérées par E. Raymond (*The Art of Unix Programming*, 2004) :

- *Rule of Modularity: Write simple parts connected by clean interface.*
- *Rule of Clarity:* ***Clarity is better then cleverness****.*
- *Rule of Simplicity:* ***Design for simplicity****; add complexity only where you must.*
- *Rule of Transparency: Design for visibility to make inspections and debugging easier.*
- *Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.*
- *Rule of Optimization: Prototype before polishing.* ***Get it working before you optimize it****.*

[7] `http://c2.com/cgi/wiki?OnceAndOnlyOnce`

3 Abstraction procédurale : Caractéristiques d'une bonne méthode

3.1 Pourquoi créer une méthode

Diverses raisons justifient la création d'une méthode :

- Pour réduire la complexité du code vu par le lecteur : le corps de la méthode permet de dissimuler les détails.
- Pour introduire une abstraction (procédurale) qui permet, à l'aide d'un nom bien choisi, de mieux documenter le rôle d'un segment de code (*self-documenting code*).
- Pour éviter de dupliquer du code.
- Pour améliorer la portabilité : les détails liées à un environnement ou une machine peuvent être dissimulés dans la méthode.

3.2 Quand utiliser une procédure par opposition à une fonction

Plusieurs auteurs considèrent qu'une **fonction**, qu'on utilise pour retourner un résultat utilisable dans une expression, ne devrait jamais avoir d'effets de bord, c'est-à-dire ne devrait modifier ni ses arguments, ni des variables globales. Seules les **procédures** devraient avoir des effets de bord.

Notamment, B. Meyer a introduit le principe de *séparation* des commandes et des requêtes :[8]

> *[A] method should either be a* **command** *that performs an action, or a* **query** *that returns data to the caller, but not both. In other words,* ***asking a question should not change the answer.***

Remarque : Mais, comme dans toute règle, il y a des exceptions!

3.3 Comment nommer une méthode

Le nom d'une méthode devrait décrire *ce que fait* la méthode (**quoi?**) et non pas comment elle le fait. Ce nom qui décrit ce que fait la méthode devrait être le plus complet possible. Il ne faut pas hésiter à utiliser des identificateur comptant plusieurs mots — mais sans toutefois abuser.

[8] `http://en.wikipedia.org/wiki/Command-query_separation`

Plus spécifiquement :

- Pour nommer une fonction, décrit la valeur retournéepar la fonction, en utilisant un prédicat dans le cas d'une fonction retournant un booléen, par exemple : `nom`, `prochain_client`, `couleur_du_fond`, `date`, `sommet`, `vide?`, `entete?`, etc.

  À éviter en Ruby : `get_nom`, `get_sommet`, etc.

- Pour nommer une procédure, verbe actif (à l'infinitif) préférablement possiblement suivi d'un complément d'objet lorsqu'approprié, par exemple : `depiler`, `calculer_moyennes`, `indiquer_perte`, `imprimer`, etc.

3.4 Comment déclarer les paramètres d'une méthode

- Il est préférable de définir les paramètres dans un ordre logique et d'utiliser le même ordre pour les méthodes semblables.

  Notamment, l'ordre suivant est proposé par certains auteurs :

  i) les paramètres associés à des entrées qui ne sont pas modifiées ;

  ii) les paramètres associés à des entrées mais qui sont aussi modifiées ;

  iii) les paramètres utilisés uniquement pour retourner un résultat.

  Note : Ne pas oublier qu'une méthode Ruby retourne toujours un résultat, même s'il peut être `nil` ou être ignoré ⇒

- La méthode doit utiliser tous les paramètres. Si un paramètre n'est pas du tout utilisé, alors il devrait être supprimé de l'en-tête.

- Les paramètres utilisés pour retourner un code de statut ou un code d'erreur devraient apparaître à la fin de la liste des paramètres.

- Sauf exception, une méthode ne devrait pas avoir plus de sept (7) paramètres.

4 La notion de couplage

Le niveau de couplage entre deux unités de programme décrit la force des **dépendances** qui existent entre ces unités. Plus le couplage est élevé, plus des changements dans une unité risquent d'avoir des répercussions sur l'autre unité.

La force du couplage dépend du nombre de connexions entre les unités.

Par exemple, une méthode avec un seul paramètre possède un niveau de couplage plus faible avec ses clients qu'une méthode qui compte six ou sept paramètres.

La force du couplage dépend aussi de la visibilité et du type des connexions entre les unités de programme.

Par exemple, un couplage *explicite* par l'intermédiaire d'arguments est préférable (plus faible) à un couplage *implicite* par l'intermédiaire de variables globales (plus fort).

Il existe un certain nombre de formes typiques de couplage entre unités de programme. Les formes suivantes, qui représentent des niveaux de couplage nul ou faibles, sont considérées acceptables :

- **Aucun couplage** : les deux unités de programme ne sont aucunement liées entre elles. Dans ce cas, on peut changer une unité de programme sans que cela n'ait d'impact sur l'autre.
- **Couplage par données simples** : les deux unités de programme s'échangent des données simples (types de base ou tableaux homogènes) *par l'intermédiaire de paramètres.*
- **Couplage par objets** : les deux unités s'échangent des objets (structures de données complexes mais fonctionnellement cohésives) *par l'intermédiaire de paramètres.*
- **Couplage par variables d'instance** : les deux unités sont dans la même classe et s'échangent de l'information par l'intermédiaire de variables d'instance de la classe.

Par contre, les formes de couplage suivantes sont généralement à éviter :

- ***Stamp (*data structure*) coupling :*** Une structure de données est passée en paramètre alors qu'un seul champ de cette structure est nécessaire.
- **Couplage par variable globale :** Les deux unités de programme communiquent par l'intermédiaire de variables globales.

- **Couplage de contrôle :** Un module passe un indicateur de contrôle à l'autre module pour lui indiquer ce qu'il doit faire. Cette forme de couplage n'est acceptable que si l'indicateur de contrôle est un type spécifiquement défini pour cette tâche (par exemple, type défini avec un `enum`).

En conclusion, l'objectif est de minimiser le plus possible le couplage entre deux unités de programme, de rendre les liens clairs et explicites autant que possible.

5 *Refactoring*

Qu'est-ce que le *refactoring*?

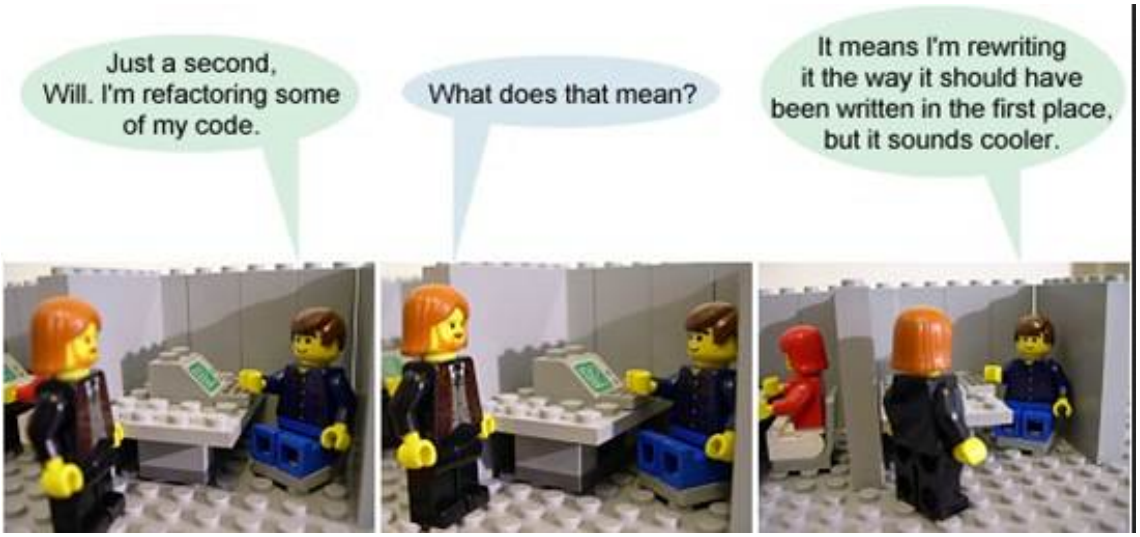

Figure .4: Qu'est-ce que le *refactoring*?

Refactoring (noun) a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Source : «Refactoring—Improving the Design of Existing Code», Fowler, 1999

Refactor (verb) to restructure software by applying a series of refactorings without changing its observable behavior.

Source : «Refactoring—Improving the Design of Existing Code», Fowler, 1999

Question : Sous quelles conditions peut-on faire du *refactoring* sans crainte de tout briser — sans crainte de régresser?

Réponse : Lorsqu'on a des tests unitaires et que notre programme exécute avec succès ces tests!

Remarque : Pour le devoir #1, *vous avez des tests* ⇒ Faites du *refactoring* pour améliorer la qualité de votre code!

6 Exemples et contre-exemples en Ruby

Exemple 1. Boucle définie vs. indéfinie

Problème : On veut calculer la somme des éléments d'un tableau `a`

Mauvais — Boucle indéfinie (`while`) pour un nombre *fixe* d'itérations

```
tot = 0
i = 0
while i < a.size
  tot += a[i]
  i += 1
end
```

Mieux — Boucle définie sur les index (`each_index`)

```
tot = 0
a.each_index do |i|
  tot += a[i]
end
```

Mieux — Boucle définie sur les éléments (`each`)

```
tot = 0
a.each do |x|
  tot += x
end
```

Encore mieux — Réduction

```
tot = a.reduce(0, :+)
```

Exemple 2. Factorisation de code répétitif

Problème : On veut calculer la somme de deux tableaux de longueurs différentes — en utilisant 0 pour les valeurs manquantes du tableau plus court

Mauvais — Code répétitif... et contenant une erreur à cause d'un copier/coller mal corrigé (laquelle?)

```
def additionner( a, b )
  if a.size <= b.size
    c = Array.new(b.size)
    (0...a.size).each do |i|
      c[i] = a[i] + b[i]
    end
    (a.size...c.size).each do |i|
      c[i] = b[i]
    end
  else
    c = Array.new(a.size)
    (0...b.size).each do |i|
      c[i] = a[i] + b[i]
    end
    (b.size...c.size).each do |i|
      c[i] = a[i]
    end
  end

  c
end
```

Mieux — Élimination du code répétitif

```
def additionner( a, b )
  a, b = b, a if a.size > b.size

  c = Array.new(b.size)

  (0...a.size).each do |i|
    c[i] = a[i] + b[i]
  end
  (a.size...c.size).each do |i|
    c[i] = b[i]
  end

  c
```

```
end
```

Encore mieux — Utilisation de tranches

```
def additionner( a, b )
  a, b = b, a if a.size > b.size

  c = Array.new(b.size)

  (0...a.size).each do |i|
    c[i] = a[i] + b[i]
  end
  c[a.size..-1] = b[a.size..-1]

  c
end
```

Exemple 3. Factorisation de code répétitif et réduction du couplage

Problème : On veut calculer la somme de deux tableaux de longueurs différentes (bis)

Mauvais — Code répétitif

```
def additionner_element( i, a, b, c )
  ai = i < a.size ? a[i] : 0
  bi = i < b.size ? b[i] : 0
  c[i] = ai + bi
end

def additionner( a, b )
  n = [a.size, b.size].max
  c = Array.new(n)

  (0...n).each do |i|
    additionner_element( i, a, b, c )
  end

  c
end
```

Mauvais — Couplage inutilement fort pour `additionner/additionnerElement` relativement au tableau `c` : on n'utilise que l'item `i`, pas tout `c` (*stamp coupling*)[9]

```
def element( i, a )
  i < a.size ? a[i] : 0
end

def additionner_element( i, a, b, c )
  c[i] = element(i, a) + element(i, b)
end

def additionner( a, b )
  n = [a.size, b.size].max
  c = Array.new(n)

  (0...n).each do |i|
    additionner_element( i, a, b, c )
  end

  c
end
```

[9] http://en.wikipedia.org/wiki/Coupling_(computer_programming)

Mieux — Couplage faible, bonne abstraction procédurale, code simple

```
def element( i, a )
  i < a.size ? a[i] : 0
end

def additionner( a, b )
  n = [a.size, b.size].max
  c = Array.new(n)

  (0...n).each do |i|
    c[i] = element(i, a) + element(i, b)
  end

  c
end
```

Encore mieux — Couplage faible, bonne abstraction procédurale, code simple

```
def element( i, a )
  i < a.size ? a[i] : 0
end

def additionner( a, b )
  n = [a.size, b.size].max

  (0...n).map do |i|
    element(i, a) + element(i, b)
  end
end
```

Exemple 4. Réduction du couplage

Problème : On veut calculer l'age d'une personne

Mauvais — Couplage de niveau *stamp*

```
class Personne
  attr_reader :nom, :adresse, :date_naissance

  def initialize( nom, adresse, date_naissance )
    @nom = nom
    @adresse = adresse
    @date_naissance = date_naissance
  end

  ...
end


def age( personne )
  ...(Time.now - personne.date_naissance) ...
end


joe = Personne.new( "Joe Bidon", "...",
                    Time.new(2000, 12, 12) )
puts age(joe)
```

Mieux — Couplage de niveau *objets* (`Time`)

```
class Personne
  ... # Inchangee ...
  ...
end


def age( date_naissance )
  ... (Time.now - date_naissance) ...
end


joe = Personne.new( "Joe Bidon", "...",
                    Time.new(2000, 12, 12) )

puts age(joe.date_naissance)
```

Encore mieux — Méthode d'instance

```
class Personne
  ... # Inchangee ...
  ...
end


class Time
  def age
    ... (Time.now - self) ...
  end
end


joe = Personne.new( "Joe Bidon", "...",
                    Time.new(2000, 12, 12) )

puts joe.date_naissance.age
```

Exemple 5. Traitement d'un cas spécial

Problème : On veut calculer la somme des éléments d'un tableau, mais en identifiant et traitant de façon spéciale le cas où `a` serait `nil`

Mauvais — Le cas spécial semble simplement une alternative «comme une autre»

```
def somme( a )
  if a.nil?
    0
  else
    a.reduce(0, :+)
  end
end
```

Mauvais — On se «débarrasse» du cas spécial au début du traitement, mais la valeur 0 reste quand même arbitraire `:(`

```
def somme( a )
  return 0 if a.nil?

  a.reduce(0, :+)
end
```

Mieux — Une méthode qui a comme précondition que a n'est pas `nil` = Principe «*Fail early, fail fast*»

```
def somme( a )
  fail "*** Dans somme: a = nil!?" if a.nil?

  a.reduce(0, :+)
end
```

Exemple 6. Déclarations locales et simplification de code

Problème : On veut trier un tableau d'entiers (tri par sélection)

Mauvais — Variables déclarées inutilement de façon globale, initialisées sans besoin, nom inutilement complexe pour une variable d'itération (`index`)

```
def trier( a )
  n = a.size
  index_min = 0

  (0..n-1).each do |i|
    index_min = i
    (i+1..n-1).each do |j|
      if a[j] < a[index_min]
        index_min = j
      end
    end

    tmp = a[i]
    a[i] = a[index_min]
    a[index_min] = tmp
  end
end
```

Mieux — Code simple, intervalle avec borne exclusive, variable introduite au point d'utilisation, garde, instruction simple pour échanger

```
def trier( a )
  n = a.size

  (0...n).each do |i|
    index_min = i
    (i+1...n).each do |j|
      index_min = j if a[j] < a[index_min]
    end
    a[index_min], a[i] = a[i], a[index_min]
  end
end
```

Exemple 7. Duplication de code *presque pareil... mais pas tout à fait!*

Problème : On veut créer un fichier temporaire avec un certain contenu, effectuer un traitement sur le fichier, puis supprimer le fichier temporaire

Mauvais — Code répétitif

```
# Test 1
contenu = ["abc", "def", "ghi"]
File.open( "foo.txt", "w" ) do |fich|
  fich.puts contenu
end
assert_equal "3 foo.txt\n", %x{wc -l foo.txt}
assert_equal "3\n", %x{wc -l <foo.txt}
FileUtils.rm_f "foo.txt"


# Test 2
contenu = ["abc", "def", "ghi"]
File.open( "foo.txt", "w" ) do |fich|
  fich.puts contenu
end
assert_equal "12 foo.txt\n", %x{wc -c foo.txt}
FileUtils.rm_f "foo.txt"
```

Mieux — Code DRY avec bloc et `yield`

```
def avec_fichier( nom_fichier, contenu )
  File.open( nom_fichier, "w" ) do |fich|
    fich.puts contenu
  end

  yield

  FileUtils.rm_f nom_fichier
end

# Test 1
avec_fichier "foo.txt", ["abc", "def", "ghi"] do
  assert_equal "3 foo.txt\n", %x{wc -l foo.txt}
  assert_equal "3\n", %x{wc -l <foo.txt}
end

# Test 2
avec_fichier "foo.txt", ["abc", "def", "ghi"] do
  assert_equal "12 foo.txt\n",  %x{wc -c foo.txt}
end
```

Références