B. Style de programmation et qualité du code

1 Introduction

Rappels sur les caractéristiques d'un programme bien écrit — **dans un bon style de programmation**.

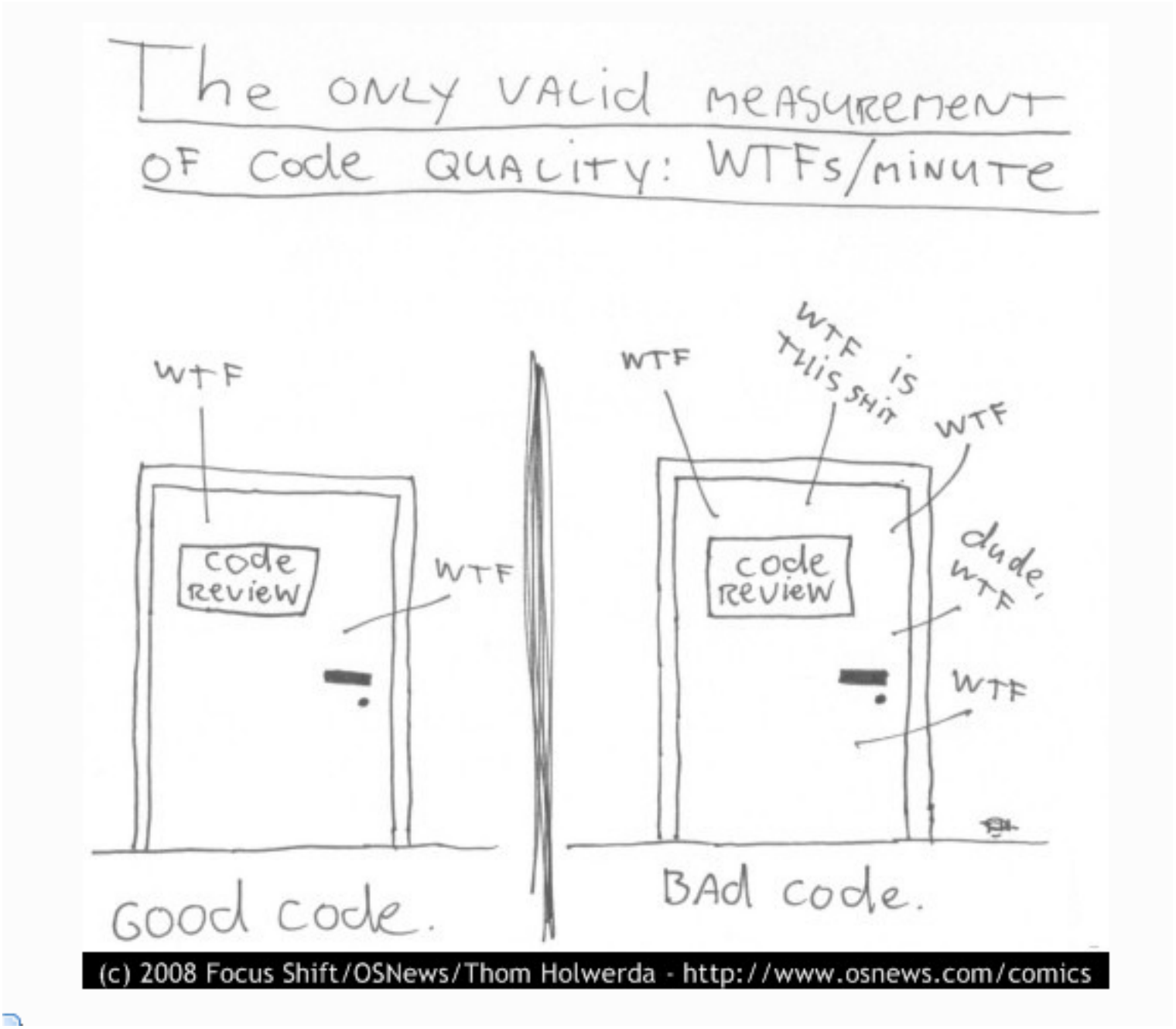

Figure B.1: La seule bonne mesure de qualité!

2 Quelques principes généraux

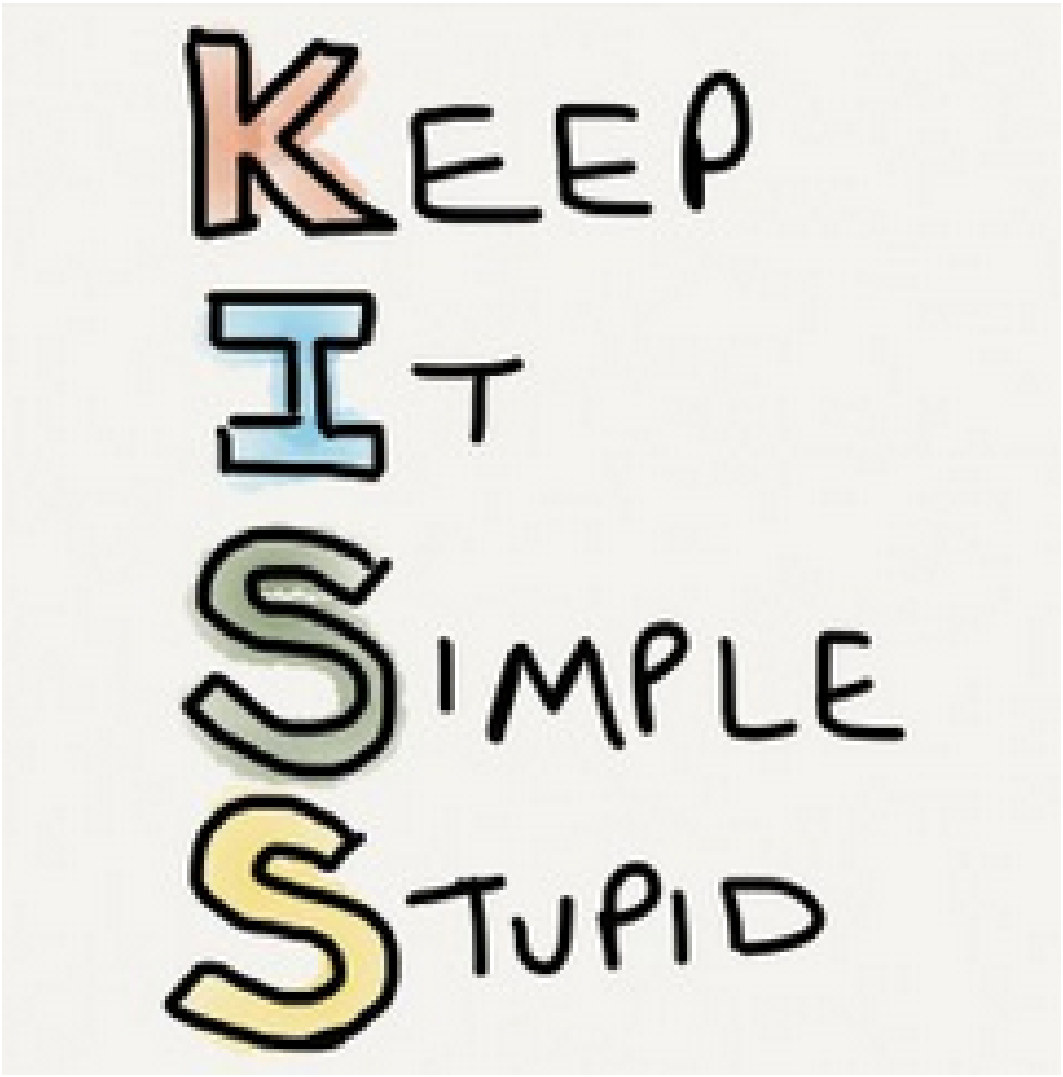

Figure B.2: KISS!

Figure B.3: DRY!

2.1 Principe KISS et principes reliés

- Principe KISS
  - *Keep It Simple, Stupid!*
  - *Keep It Short and Simple!*

- Maxime attribuée à Antoine de St-Exupéry :

  «**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 :

  «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****.»*

Et comme l'illustre la solution au laboratoire #4, la solution la plus simple est souvent la plus efficace!

The third rule of optimization: *Profile before optimizing*

2.2 Principe DRY et principes reliés

- DRY = *Don't Repeat Yourself*

- Formulation de Hunt & Thomas (*The Pragmatic Programmer—From Journeyman to Master*) :

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

- Autre formulation = ***Once and Only Once*** :

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

2.3 Règles de E. Raymond

Quelques règles suggérées par Raymond (*The Art of Unix Programming*) :

- *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.***

3 Abstraction procédurale : Caractéristiques d'une bonne méthode

3.1 Pourquoi créer une méthode

- Pour réduire la complexité du code vu par le lecteur (dissimulation de l'information)
- Pour introduire une abstraction qui permet de mieux documenter le rôle d'un segment de code (*self-documenting code*)
- **Pour éviter de dupliquer du code**
- Pour améliorer la portabilité

3.2 Quand utiliser une procédure par opposition à une fonction

Meyer a introduit le principe de **séparation** des commandes et des requêtes :

> *[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.

- **Fonction (requête)** : décrit la valeur retournée :
  `nom`,
  `prochain_client`,
  `couleur_du_fond`,
  `date`,
  `sommet`,
  `vide?`,
  `entete?`,
  etc.
  À éviter en Ruby : `get_nom`, `get_sommet`, etc.

- **Procédure (commande)** : verbe possiblement suivi d'un complément :
  `depiler`,
  `calculer_moyennes`,
  `indiquer_perte`,
  `imprimer`,
  etc.

3.4 Comment déclarer les paramètres d'une méthode

- Définir les paramètres dans un ordre logique.

  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 ⇒

- Doit utiliser tous les paramètres.

- 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 de leurs **dépendances**.

Plus le couplage est élevé, plus des changements dans une unité risquent d'avoir des répercussions sur l'autre.

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

• La force du couplage dépend aussi **de la visibilité et du type** des connexions.

Formes de couplage acceptables :

- **Aucun couplage** : les deux unités de programme ne sont aucunement liées entre elles.

- **Couplage par données simples** : les deux unités de programme s'échangent des données simples **par l'intermédiaire de paramètres**.

- **Couplage par objets**: les deux unités s'échangent des objets **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.

Formes de couplage 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.

5 *Refactoring*

Qu'est-ce que le *refactoring*?

Figure B.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** $\Rightarrow$ 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? Bon? Excellent?

```
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? Bon? Excellent?

```
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? Bon? Excellent?

```
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? Bon? Excellent?

```
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
```

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? Bon? Excellent?

```
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? Bon? Excellent?

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

Mauvais? Bon? Excellent?

```ruby
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*»

```ruby
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? Bon? Excellent?

```
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? Bon? Excellent?

```
# 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`

```ruby
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
```