Qualité du code et *Refactoring*

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

INF600A
2 octobre 2018
? novembre 2018

Contenu

A. Quel est le rapport avec le devoir #1. . . et #2 ?

1. Qualités des produits logiciels

Qu’est-ce que la «qualité» ?

Au sens large, la qualité est la «manière d’être», bonne ou mauvaise, de quelque chose.

Dans le langage courant, la qualité tend à désigner ce qui rend quelque chose supérieur à la moyenne.

Source: *https://fr.wikipedia.org/wiki/Qualit%C3%A9*

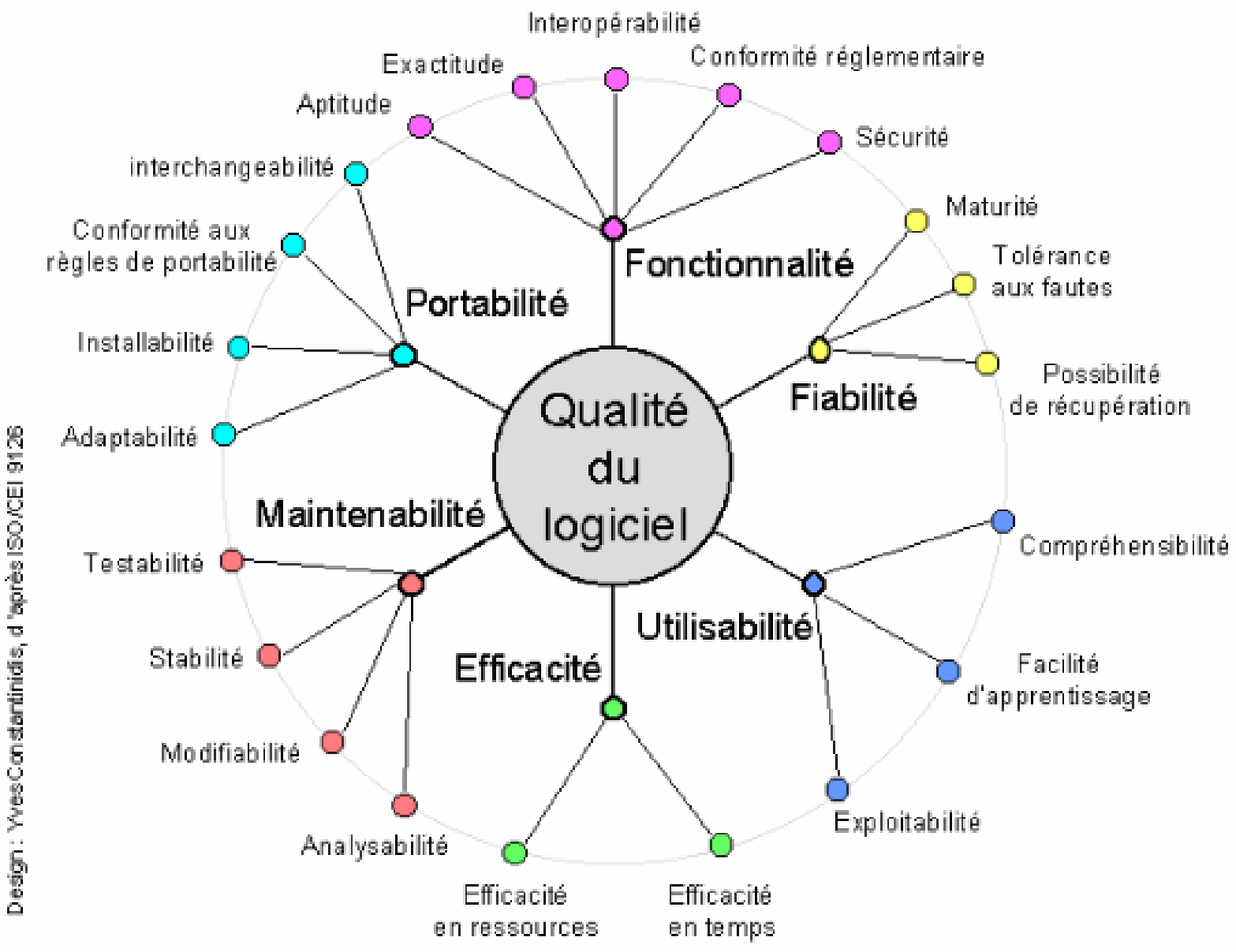
Qualités des produits logiciels : ISO/IEC 25000

http://photos1.blogger.com/blogger/1941/1049/1600/SW%20Quality%20Iceberg.jpg

«Norme [qui] définit et décrit une série de caractéristiques qualité d'un produit logiciel (caractéristiques internes et externes, caractéristiques à l'utilisation)»

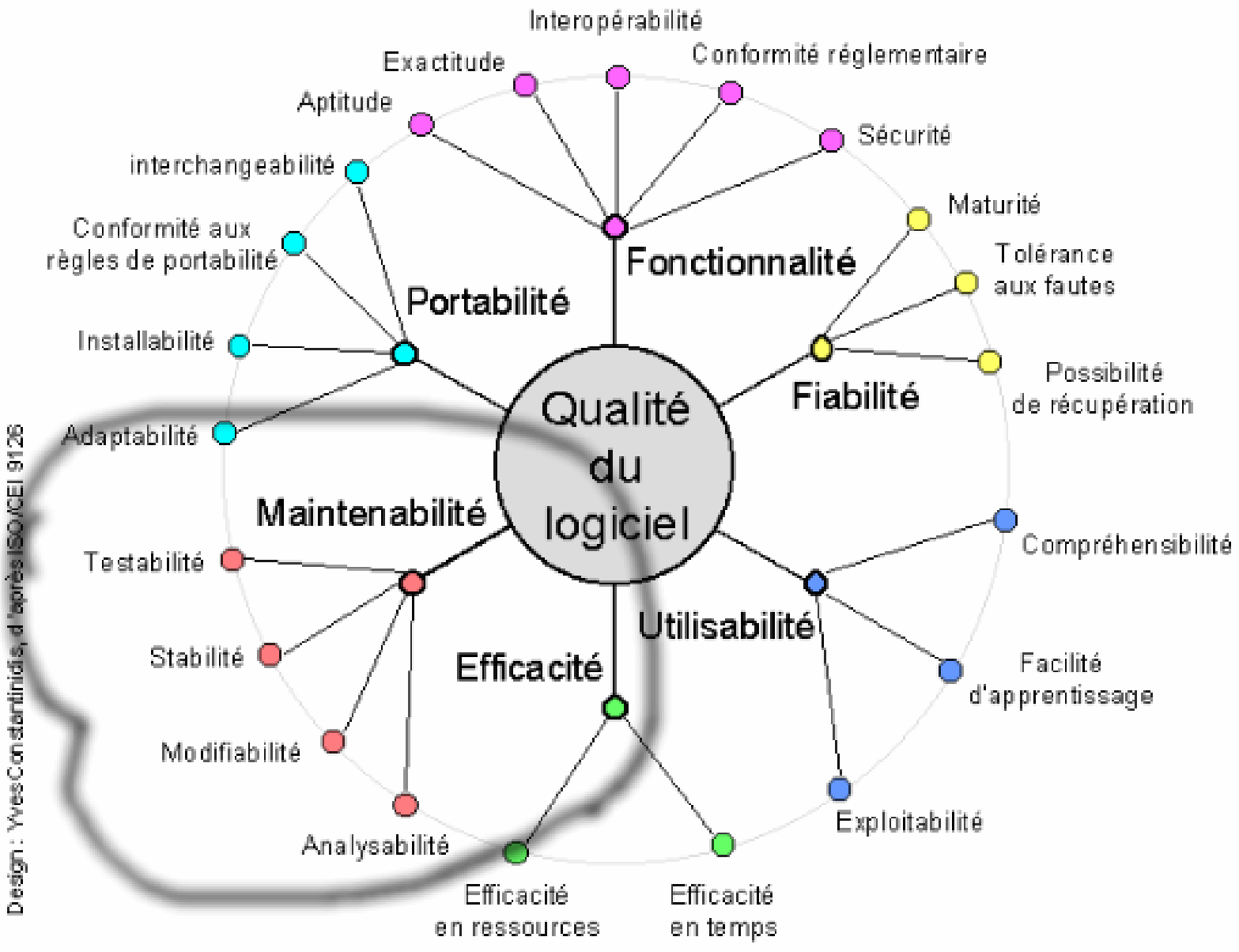
Qualités des produits logiciels : ISO/IEC 25000

https://prezi.com/15ldplotttpg/copy-of-iso-25000-is3/



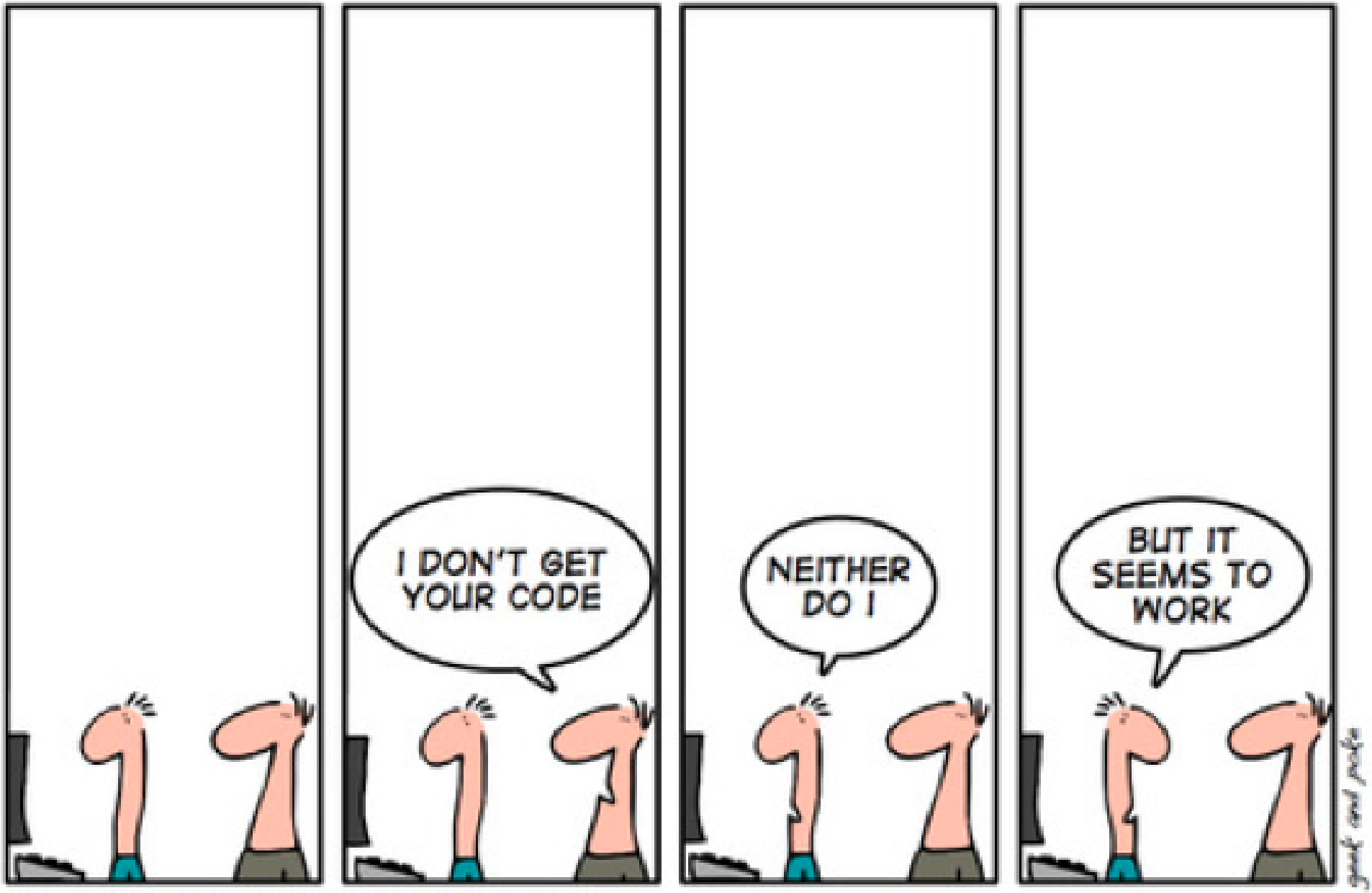
Qualités des produits logiciels : ISO/IEC 25000. . . et qualité du code

https://prezi.com/15ldplotttpg/copy-of-iso-25000-is3/

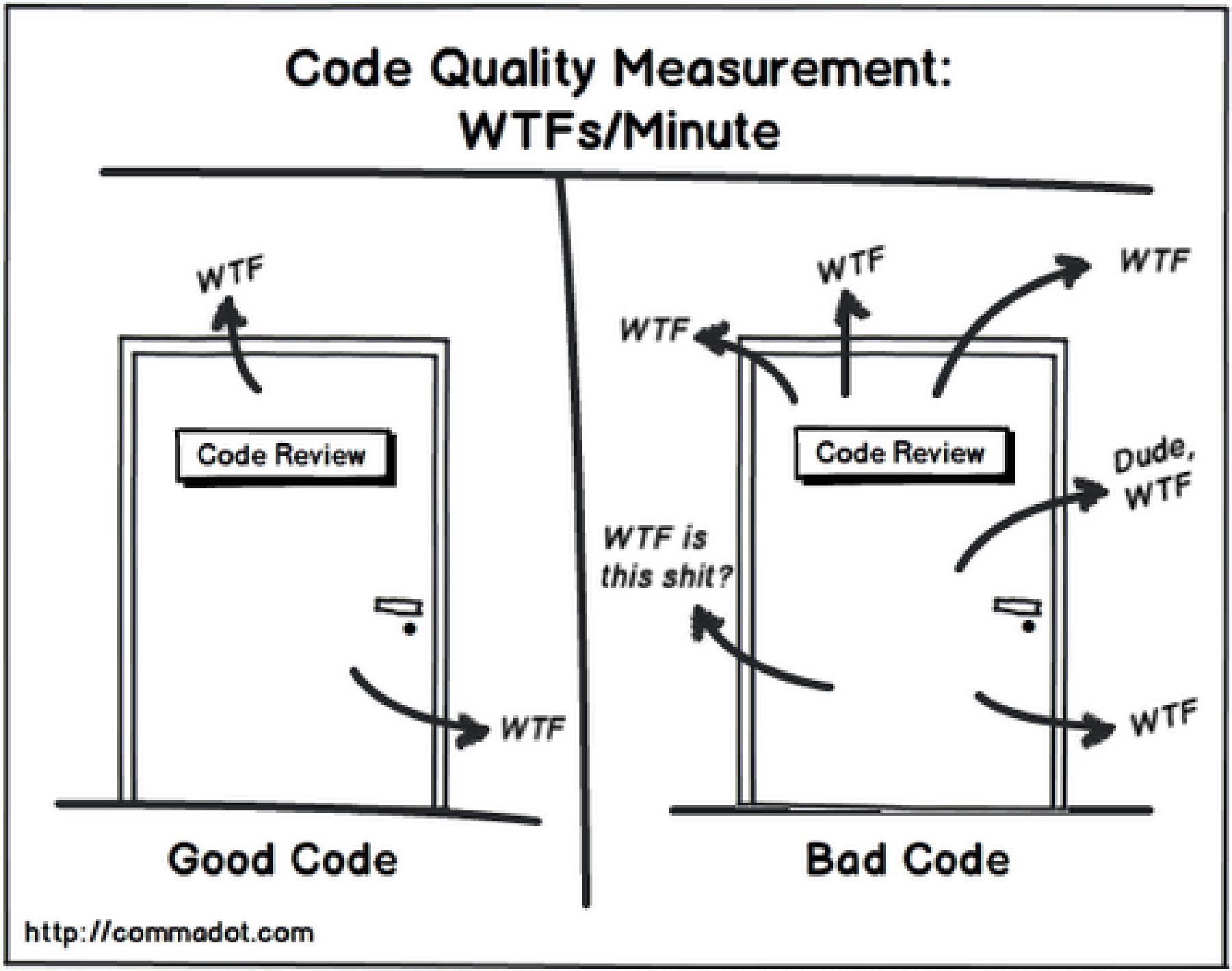


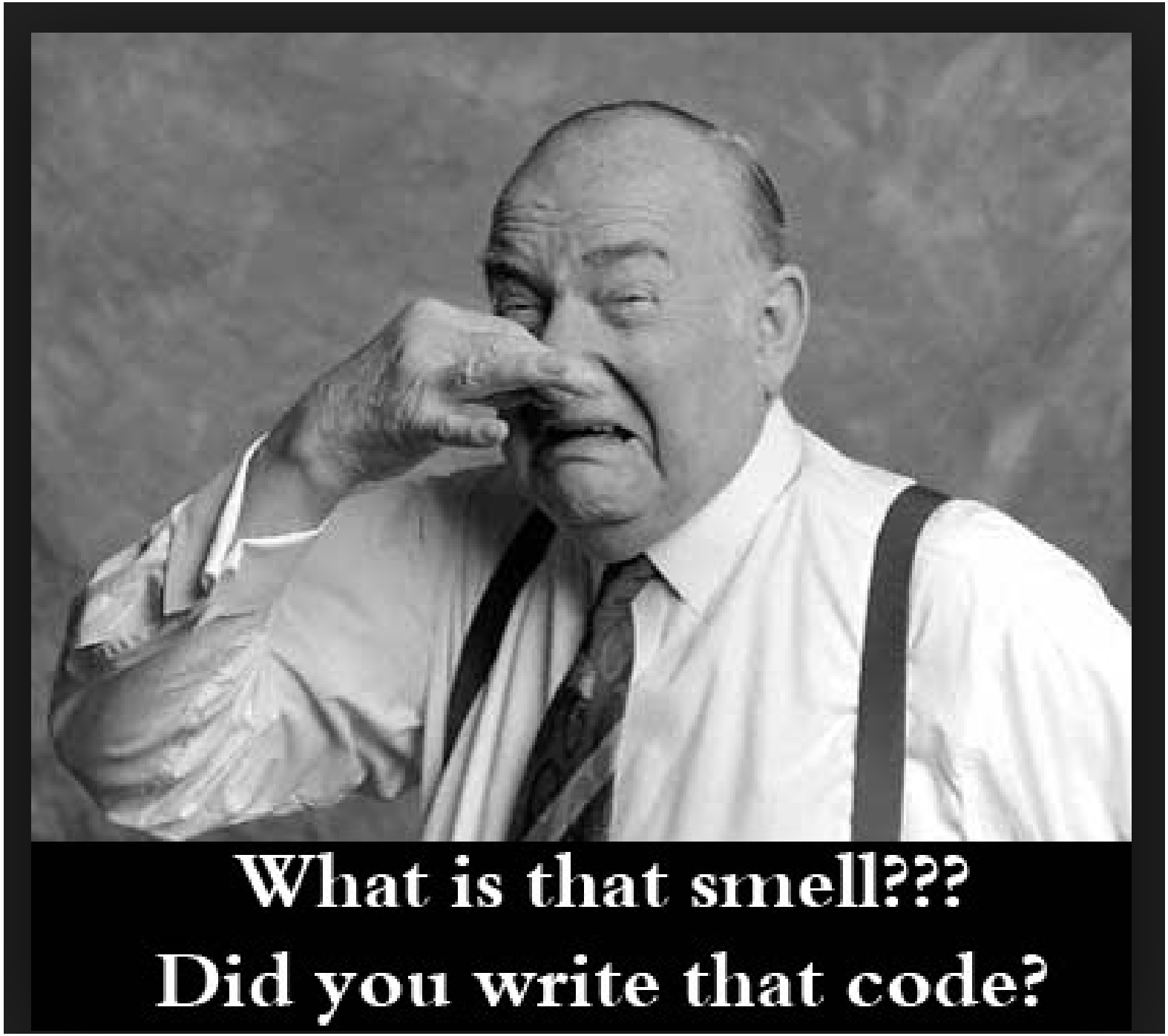
2. Qualité du code :*Code Smells vs. Clean Code*

Qu’est-ce du code de mauvaise qualité ?



THE ART OF PROGRAMING


Code Quality Measurement:
WTFs/Minute
WTF
Code Review
WTF
Good Code
WTF
WTF
WTF
WTF
Code Review
Dude,
WTF
WTF is
this shit?
WTF
Bad Code
http://commadot.com



http://2.bp.blogspot.com/_IbkhPOy5cIs/TJNZU7VFF9I/AAAAAAAAAUM/0VvUbw5mQ6Y/s1600/Bad+smell+with+text.jpg

Question : Qu'est-ce qu'un «*Code Smell*» ?

Question : Qu'est-ce qu'un «*Code Smell*» ?

– *What ? How can code «smell» ? ?*

– *Well it doesn't have a nose. . . but it definitely can stink !*

Source: `https://sourcemaking.com/refactoring/smells`

Question : Qu'est-ce qu'un «*Code Smell*» ?

– What ? How can code «smell» ? ?

– Well it doesn't have a nose. . . but it definitely can stink !

Source: `https://sourcemaking.com/refactoring/smells`

Code Smell

Code smell, also known as bad smell, in computer programming code, refers to any symptom in the source code of a program that **possibly** *indicates a deeper problem.*

Source: `https://en.wikipedia.org/wiki/Code_smell`

Question : Qu'est-ce qu'un «*Code Smell*» ?

– What ? How can code «smell» ? ?

– Well it doesn't have a nose. . . but it definitely can stink !

Source: *https://sourcemaking.com/refactoring/smells*

Code Smell (bis)

"Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality".

Source: https://en.wikipedia.org/wiki/Code_smell

Cinq catégories de *code smells* selon les travaux de Fowler (sur les logiciels orientés objets)

1. *Bloaters*
2. *Object-Orientation Abusers*
3. *Change Preventers*
4. *Dispensables*
5. *Couplers*

Source: `https://sourcemaking.com/refactoring/smells`

Voir aussi : `https://quizlet.com/201301/refactoring-code-smells-flash-cards/`

Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- *Long Method*
- *Large Class*
- *Primitive Obsession*
- *Long Parameter List*
- *Data Clumps*

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

- *Comments*
- *Duplicate Code*
- *Lazy Class*
- *Data Class*
- *Dead Code*
- *Speculative Generality*

Qu’est-ce que du code propre (*Clean code*) ?

Clean Code selon M. Feathers

So what is in fact the definition of clean code ?

Clean Code selon M. Feathers

So what is in fact the definition of clean code ?

Clean code is a code that *is written by someone* **who cares**

M. Feathers

Suggestions et lignes directrices pour obtenir du code maintenable et propre

Top 9 qualities of clean code (P. Bejger)

https://blog.goyello.com/2013/01/21/top-9-principles-clean-code/

1. *Bad code does too much – Clean code is focused*
2. *The language you wrote your code with should look like it was made for the problem*
3. *It should not be redundant*
4. *Reading your code should be pleasant*
5. *Can be easily extended by any other developer*
6. *It should have minimal dependencies*
7. *Smaller is better*
8. *It should have unit and acceptance tests*
9. *It should be expressive*

«Seven developer “Boy Scout rules” that [help] prevent code smells that impact maintainability most»

«Key “Boy Scout rules” = Leave no trace»

1. *Leave no unit-level code smells behind.*
2. *Leave no bad comments behind.*
3. *Leave no code in comments behind.*
4. *Leave no dead code.*
5. *Leave no long identifier names behind.*
6. *Leave no magic constants behind.*
7. *Les no badly handled exceptions behind.*

Dix (10) lignes directrices (*guidelines*) pour produire du logiciel facilement maintenable

«*Building Maintenable Software*», Visser, 2016

1. *Write short units of code*
2. *Write simple units of code*
3. *Write code once*
4. *Keep unit interfaces small*
5. *Separate concerns in modules*
6. *Couple architecture components loosely*
7. *Keep architecture components balances*
8. *Keep your codebase small*
9. *Automate tests*
10. *Write clean code*

1. *Write short units of code*

Shorter units (that is, methods and constructors) are easier to analyze, test, and reuse.

- ***Limit the length of code units to 15 lines of code.***

Motivation

- *Short units are easy to test*
- *Short units are easy to analyze*
- *Short units are easy to reuse*

2. *Write simple units of code*

Units with fewer decision points are easier to analyze and test.

- ***Limit the number of branch points per unit to 4.***

Motivation

- *Simpler units are easier to modify*
- *Simpler units are easier to test*

3. *Write code once*

Duplication of source code should be avoided at all times, since changes will need to be made in each copy. Duplication is also a source of regression bugs.

- ***Do not copy code.***

Motivation

- *Duplicated code is harder to analyze*
- *Duplicated code is harder to modify*

3. *Write code once*

Duplication of source code should be avoided at all times, since changes will need to be made in each copy. Duplication is also a source of regression bugs.

- ***Do not copy code.***

Motivation

- *Duplicated code is harder to analyze*
- *Duplicated code is harder to modify*

Pourquoi ?

4. *Keep unit interfaces small*

Units (methods and constructors) with fewer parameters are easier to test and reuse.

- ***Limit the number of parameters per unit to at most 4.***

Motivation

- *Small interfaces are easier to understand and reuse*
- *Small interfaces are easier to modify*

10. *Write clean code*

Having irrelevant artifacts such as TODOs and dead code in your codebase makes it more difficult for new team members to become productive. Therefore, it makes maintenance less efficient.

- *Write clean code.*
- *Do this **by not leaving code smells behind** after development work.*

Motivation

- *Clean code is maintainable code*

Deux principes généraux pour du code de qualité

Deux principes généraux pour du code de qualité

KISS

DRY

De quoi s’agit-il ?

KEEP

IT

SIMPLE

STUPID

Principe KISS

http://en.wikipedia.org/wiki/KISS_principle

- *Keep It Simple, Stupid !*

Principe KISS

http://en.wikipedia.org/wiki/KISS_principle

- *Keep It Simple, Stupid !*

- *Keep It Short and Simple !*

Principes similaires au principe KISS

Rasoir d'Occam

«Les hypothèses les plus simples sont les plus vraisemblables».

Source: `http://fr.wikipedia.org/wiki/Rasoir_d'Ockham`

Maxime attribuée à A. 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.»

Source: `http://www.drop-zone-city.com/article.php3?id_article=221`

Principes similaires au principe KISS

Simplicité selon 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*.»

Source: «*The Emperor's Old Clothes*», , February 1981


I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
DON'T REPEAT YOURSELF
Repetition is the root of all software evil

Principes similaires au principe DRY

Formulation de Hunt & Thomas

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

Source: «*The Pragmatic Programmer—From Journeyman to Master*, Hunt & Thomas, 2000

Principe `OnceAndOnlyOnce`

«*Each and every declaration of behavior should appear* `OnceAndOnlyOnce`.»

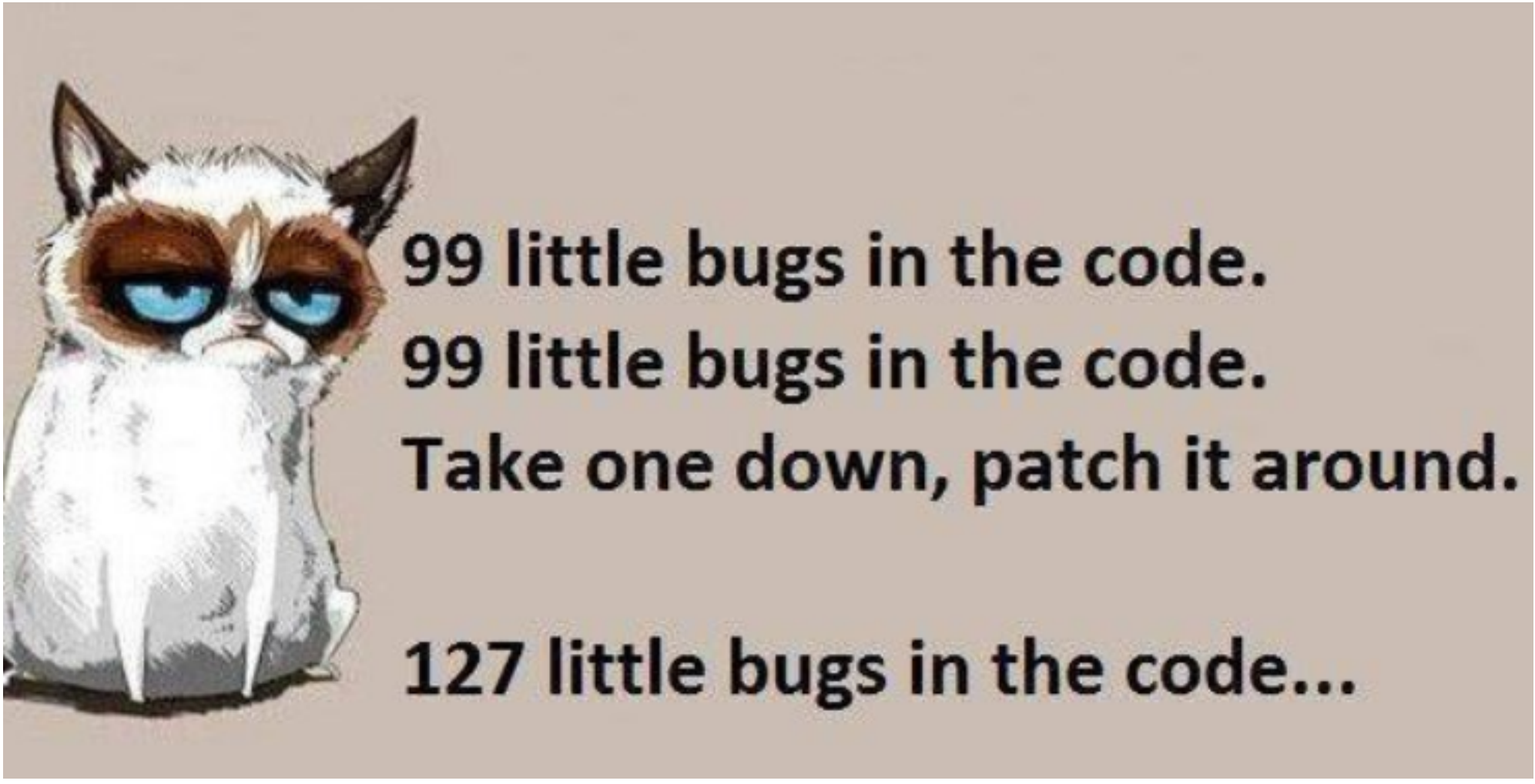
Source: `http://c2.com/cgi/wiki?OnceAndOnlyOnce`

3. Dette technique et *Refactoring*

Qu’est-ce que la «dette technique» (*technical debt*) ?

Source: https://www.theguild.nl/dealing-with-technical-debt

Source: http://svsg.co/category/blog/



Source: `http://starecat.com/99-little-bugs-in-the-code/`

Lyrics of the song 99 Bottles of Beer

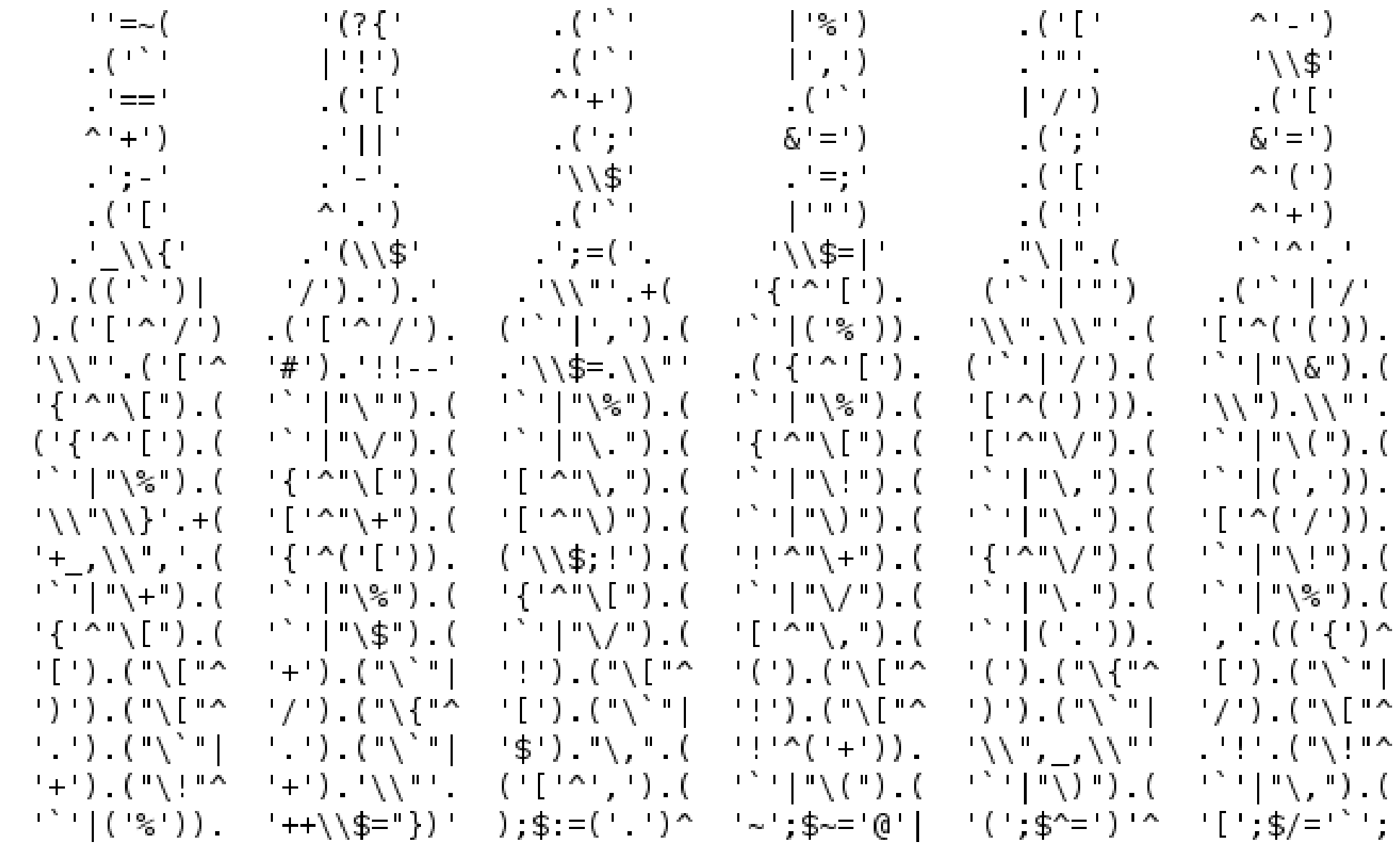
99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.

97 bottles of beer on the wall, 97 bottles of beer.
Take one down and pass it around, 96 bottles of beer on the wall.

96 bottles of beer on the wall, 96 bottles of beer.
Take one down and pass it around, 95 bottles of beer on the wall.

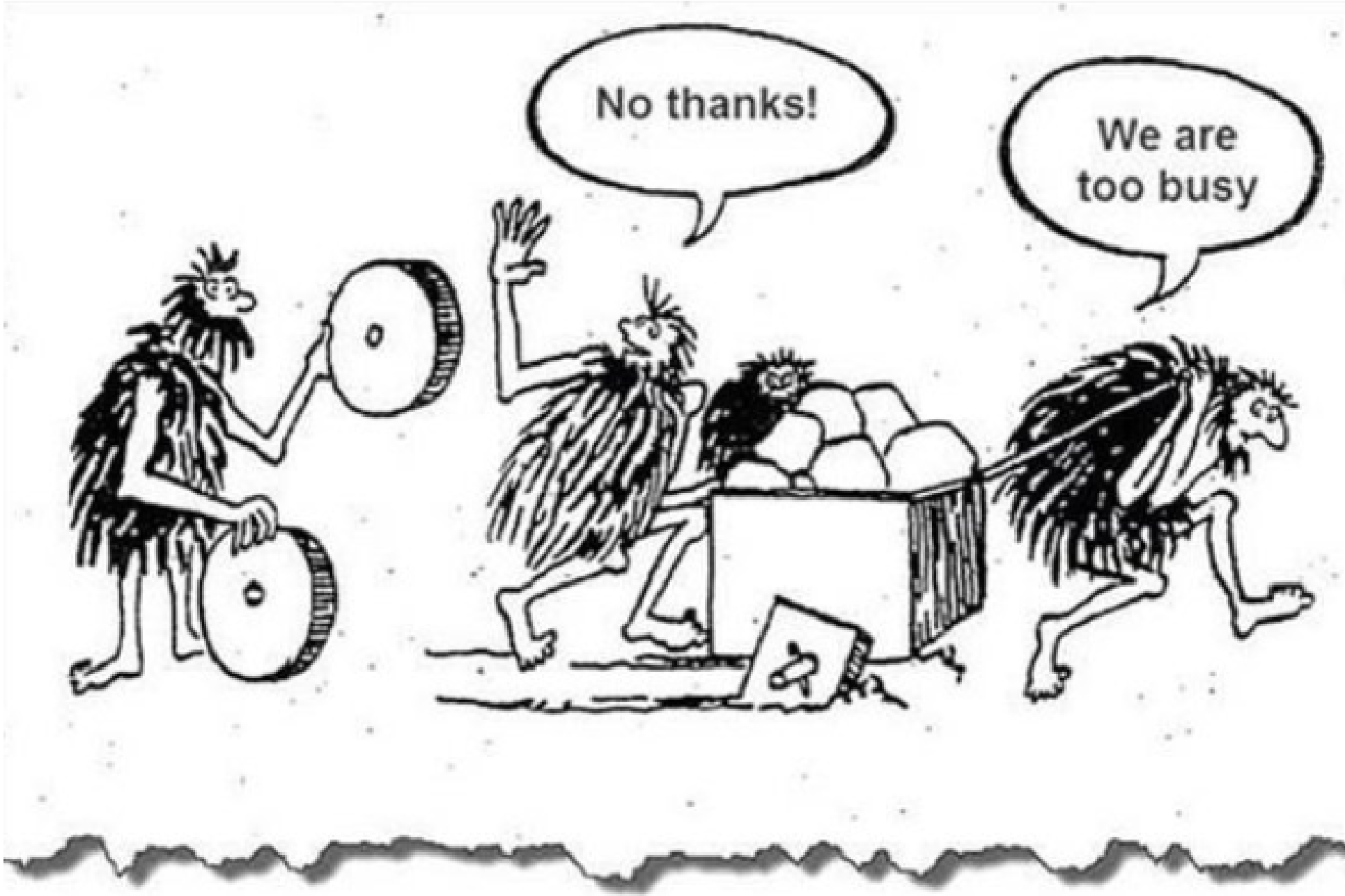
95 bottles of beer on the wall, 95 bottles of beer.
Take one down and pass it around, 94 bottles of beer on the wall.



Dette technique

Technical debt is the continuous accumulation of shortcuts, hacks, duplication, and other sins we regularly commit against our code base in the name of speed and schedule.
[. . .]
Technical debt can take many forms (spaghetti code, excessive complexity, duplication, and general slopiness), but what makes it really dangerous is how it just kind of sneaks up on you. Each transgression initially made against the code base seem small or insignificant. But like all forms of debt, it's the cumulative effect that adds up over time that hurts.

Source*: «*The Agile Samurai*», Rasmusson, 2010*



Source: http://shhetri.github.io/unit-testing-to-tackle-technical-debt/#/cover

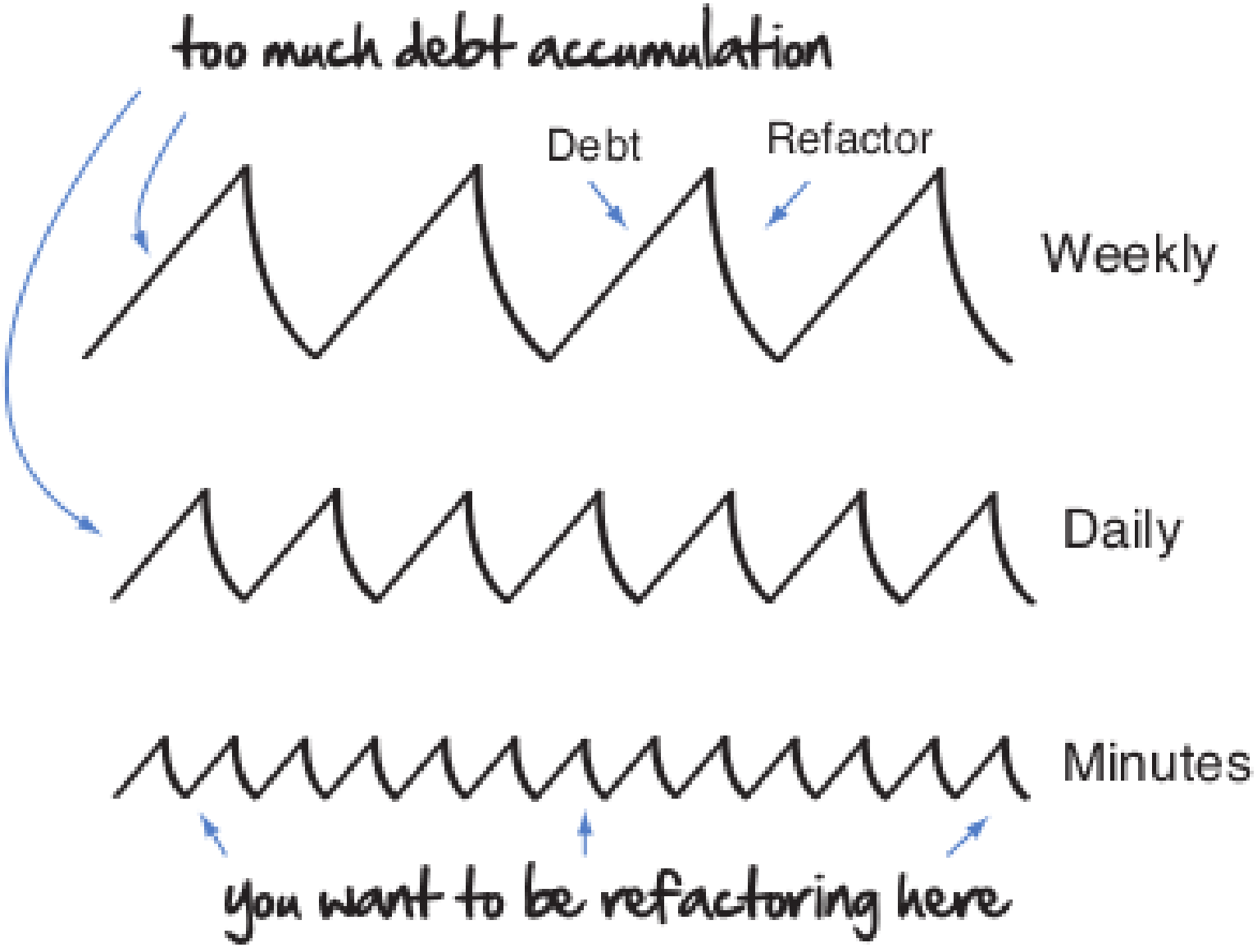
Dette technique

«Technical debt» is a metaphor to describe not-quite-right code. The technical-debt metaphor helps us communicate that if we want to build something on top of not-quite-right code, it will be expensive to do something on this code base later on. So, it takes longer to implement a new feature on a not-so-good-code base.

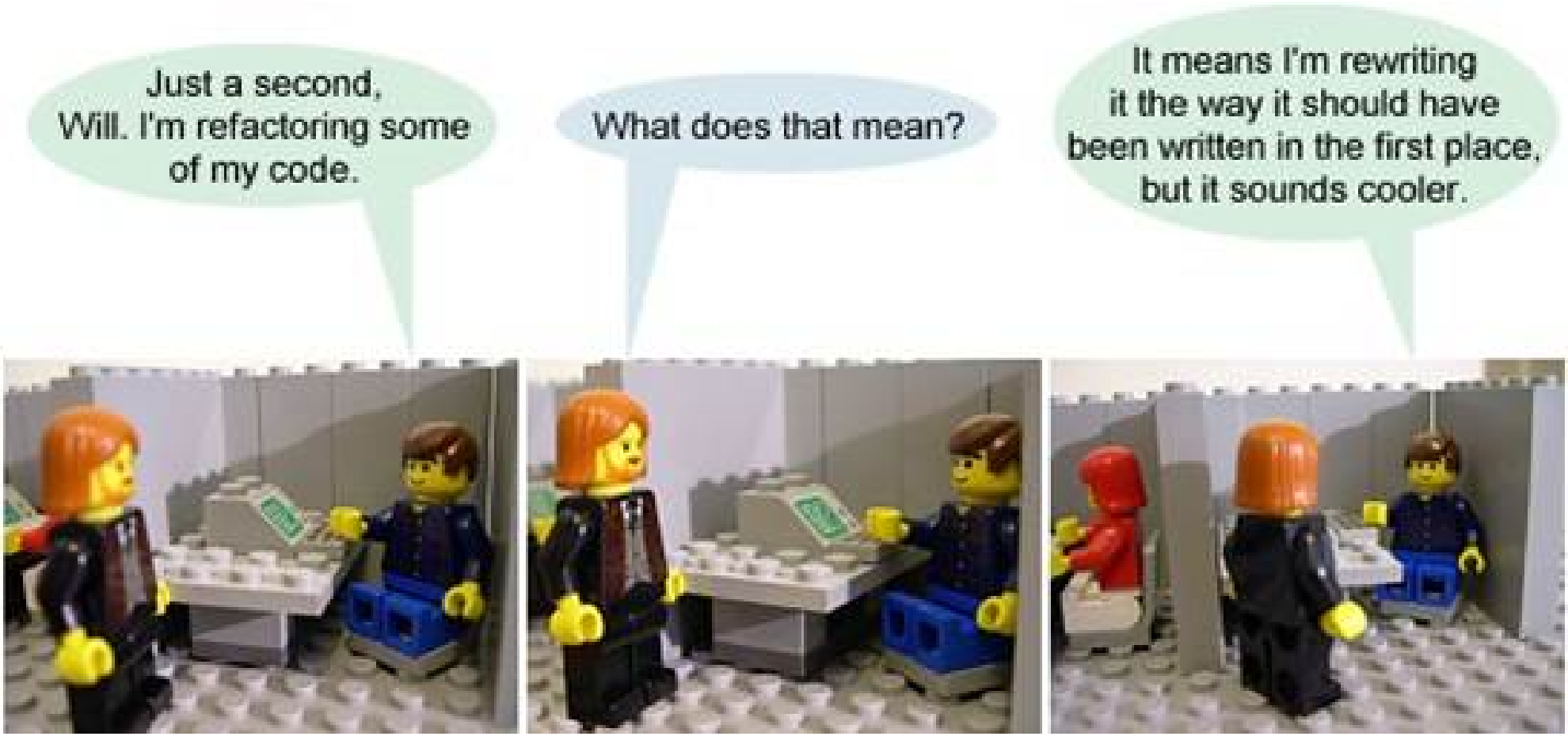
Source: «Technical Debt», *Wolff and Johann, IEEE Software, vol. 32, no. 4, 2015.*

De quelle façon peut-on «rembourser sa dette technique» ?

En nettoyant le code, en faisant du *refactoring* !



Qu’est-ce que le *refactoring* ?



Qu’est-ce que le *refactoring* ?

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*

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*

REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International Inc.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

SERIES EDITORS

ADDISON-WESLEY

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Source: `http://www.refactoring.com/`

Diverses catégories de *Refactorings*

1. *Composing methods*
2. *Moving features between objects*
3. *Organizing data*
4. *Simplifying conditional expressions*
5. *Simplifying method calls*
6. *Dealing with generalisation*

Pour les détails de ces *refactoring*s, voir. . .

- Le site de M. Fowler :
  `http://refactoring.com/catalog/`

- Le site de `refactoring.guru`
  `https://refactoring.guru/catalog`

Fait : De nombreux outils fournissent du support pour le refactoring, par exemple, en Java

- Eclipse
- IntelliJ
- NetBeans

Sous quelles conditions peut-on faire du *refactoring* sans crainte de tout briser, c'est-à-dire sans crainte de régresser ?

Réponse

Lorsqu’on a des tests unitaires et que notre programme exécute avec succès ces tests !

Réponse

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

Tests eliminate fear

Tests allow you to make changes without the risk of breaking something

Source*: «*The Clean Coder : A Code of Conduct for Professional Programmers*», R.C. Martin*

4. Petit exemple de simplification/nettoyage de code (Ruby)

Le problème

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

Première solution : Quel est le problème ?


```
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
```

Première solution : Quel est le problème ?


```
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
```

Le problème = Il y a du code qui se répète


```
ai = i < a.size ? a[i] : 0
bi = i < b.size ? b[i] : 0
```

Donc, il faut le factoriser pour le rendre **DRY**

Le problème = Il y a du code qui se répète


```
  ai = i < a.size ? a[i] : 0
  bi = i < b.size ? b[i] : 0
```

Donc, il faut le factoriser pour le rendre **DRY**

$\Rightarrow$ Il faut introduire une méthode auxiliaire

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

def additionner_element( i, a, b, c )
  ai = obtenir_element( i, a )
  bi = obtenir_element( i, b )
  c[i] = ai + bi
end
```

Le problème = Il y a du code qui se répète


```
  ai = i < a.size ? a[i] : 0
  bi = i < b.size ? b[i] : 0
```

Donc, il faut le factoriser pour le rendre **DRY**
$\Rightarrow$ Il faut introduire une méthode auxiliaire

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

def additionner_element( i, a, b, c )
  ai = obtenir_element( i, a )
  bi = obtenir_element( i, b )
  c[i] = ai + bi
end
```

Question : Peut-on encore améliorer ce code ?

Le principe de séparation des commandes et des requêtes de Bertrand Meyer

[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.***

Source: `http://en.wikipedia.org/wiki/Command-query_separation`

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

Le choix des noms de méthodes

- Le nom d'une méthode devrait décrire *ce que fait* la méthode (quoi ?) et non pas comment elle le fait.

- Il ne faut pas hésiter à utiliser des identificateurs avec plusieurs mots — sans toutefois abuser.

Le choix des noms de méthodes

Fonction (requête) : devrait décrire la valeur retournée

(prédicat lorsque résultat booléen) :

- `nom`
- `prochain_client`
- `couleur_du_fond`
- `date`
- `sommet`
- `est_vide`
- `est_valide`
- etc.

Le choix des noms de méthodes

Procédure (commande) : devrait être un verbe actif (infinitif) possiblement suivi d'un complément d'objet si approprié :

- `depiler`
- `calculer_moyennes`
- `indiquer_perte`
- `imprimer`
- `lister`
- `ajouter`
- etc.

Le problème = Le nom choisi pour la nouvelle méthode ☹


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

def additionner_element( i, a, b, c )
  ai = obtenir_element( i, a )
  bi = obtenir_element( i, b )
  c[i] = ai + bi
end
```

Un meilleur nom pour la nouvelle méthode


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

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

Un meilleur nom pour la nouvelle méthode


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

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

Question : Peut-on encore simplifier ce code ?

Étant donné la fonction `element`, les variables auxiliaires ne sont plus nécessaires


```
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
```

On a éliminé le code dupliqué (plus DRY) et simplifié (KISS) ! Mais peut-on encore améliorer ?


```
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
```

On a éliminé le code dupliqué (plus DRY) et simplifié (KISS) ! Mais peut-on encore améliorer ?


```
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
```

Niveau de couplage entre unités de programme

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 unité.

La force du couplage dépend...

- du nombre de connexions — e.g., nombre de paramètres
- de la visibilité et du type de connexion

Types de couplage

Formes de couplage acceptables ☺

- **Aucun couplage**
- **Couplage par données simples** — i.e., valeurs primitives passées en paramètres
- **Couplage par objets** — i.e., objets passés en paramètres
- **Couplage par variables d'instance** — i.e., accès aux attributs par les méthodes d'une même classe

Types de couplage

Formes de couplage à éviter ☹

- ***Stamp (*data structure*) coupling :***
  Une structure de données complexe est passée en paramètre alors qu'un seul champ de cette structure est utilisée/nécessaire.

- **Couplage par variable globale**

- **Couplage de contrôle**

Un exemple de *stamp coupling*


```
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
```

Problème : On passe en argument le tableau `c` au complet, même si on modifie seulement `c[i]` ☹

Solution : Transformer la procédure (commande) en fonction (requête)


```
def addition_element( i, a, b )
  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|
    c[i] = addition_element( i, a, b )
  end

  c
end
```

Solution : Transformer la procédure (commande) en fonction (requête)


```
def addition_element( i, a, b )
  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|
    c[i] = addition_element( i, a, b )
  end

  c
end
```

Question : Peut-on encore faire mieux ?

Solution : Éliminer la méthode !


```
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
```

On a réduit le couplage, et ce avec avec une méthode en moins
$\Rightarrow$ plus simple (KISS) !

5. Autres exemples de simplification/nettoyage de code (Ruby)

Boucle définie vs. boucle indéfinie vs. réduction

Somme des éléments d'un tableau `a`

Mauvais? Bon? Excellent?


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

Somme des éléments d'un tableau `a`

Mauvais? Bon? Excellent?

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

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

Somme des éléments d'un tableau `a`

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

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

Somme des éléments d'un tableau `a`

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

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

Somme des éléments d'un tableau `a`

Encore mieux — Réduction

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

Traitement d’un «cas spécial»

Somme des éléments de `a`, possiblement `nil`

Mauvais? Bon? Excellent?

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

Somme des éléments de `a`, possiblement `nil`

Mauvais? Bon? Excellent?

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

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

Somme des éléments de `a`, possiblement `nil`

Mauvais? Bon? Excellent?

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

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

Somme des éléments de `a`, possiblement `nil`

Mauvais? Bon? Excellent?

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

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

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

Somme des éléments de `a`, possiblement `nil`

Mieux — Une méthode avec une 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
```

Déclarations locales et simplification de code

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
```

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
```

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

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

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
```

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

Encore mieux — Réduction explicite

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

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


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

Duplication de code *presque* pareil — avec des différences au niveau des expressions

Lorsque des bouts de code sont presqu'identiques, sauf pour certaines expressions...

Avant

```
ai = i < a.size ? a[i] : 0
bi = i < b.size ? b[i] : 0
...
foo( j < x.size ? x[j] : 0 )
```

Lorsque des bouts de code sont presqu'identiques, sauf pour certaines expressions...

Avant

```
ai = i < a.size ? a[i] : 0
bi = i < b.size ? b[i] : 0
...
foo( j < x.size ? x[j] : 0 )
```

Lorsque des bouts de code sont presqu'identiques, sauf pour certaines expressions...

Avant

```
ai = i < a.size ? a[i] : 0
bi = i < b.size ? b[i] : 0
...
foo( j < x.size ? x[j] : 0 )
```

Après

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

ai = element(i, a)
bi = element(i, b)
..
foo( element(j, x) )
```

Lorsque des bouts de code sont presqu'identiques, sauf pour certaines expressions...

Avant

```
ai = i < a.size ? a[i] : 0
bi = i < b.size ? b[i] : 0
...
foo( j < x.size ? x[j] : 0 )
```

Après

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

ai = element(i, a)
bi = element(i, b)
..
foo( element(j, x) )
```

... on introduit une méthode pour la partie commune, avec des paramètres pour les parties variables

Duplication de code *presque* pareil — avec des différences au niveau de suites d'instructions

Création d'un fichier temporaire avec un contenu, puis suppression du fichier après traitement pour des tests


```
# Test 1
File.open( "foo.txt", "w" ) do |fich|
  fich.puts ["abc", "def", "ghi"]
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
File.open( "bar.txt", "w" ) do |fich|
  fich.puts ["123", "456", "789"]
end
assert_equal "12 bar.txt\n", %x{wc -c bar.txt}
FileUtils.rm_f "bar.txt"
```

Création d'un fichier temporaire avec un contenu, puis suppression du fichier après traitement pour des tests[94]

```ruby
# Test 1
File.open( "foo.txt", "w" ) do |fich|
  fich.puts ["abc", "def", "ghi"]
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
File.open( "bar.txt", "w" ) do |fich|
  fich.puts ["123", "456", "789"]
end
assert_equal "12 bar.txt\n", %x{wc -c bar.txt}
FileUtils.rm_f "bar.txt"
```

Mauvais : Code répétitif... mais pas identique ☹

Lorsque des bouts de code sont presqu'identiques, sauf pour certaines instructions...


```
# Test 1
File.open( "foo.txt", "w" ) do |fich|
  fich.puts ["abc", "def", "ghi"]
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
File.open( "bar.txt", "w" ) do |fich|
  fich.puts ["123", "456", "789"]
end
assert_equal "12 bar.txt\n", %x{wc -c bar.txt}
FileUtils.rm_f "bar.txt"
```

Lorsque des bouts de code sont presqu'identiques, sauf pour certaines instructions...


```
# Test 1
File.open( "foo.txt", "w" ) do |fich|
  fich.puts ["abc", "def", "ghi"]
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
File.open( "bar.txt", "w" ) do |fich|
  fich.puts ["123", "456", "789"]
end
assert_equal "12 bar.txt\n", %x{wc -c bar.txt} #<---------
FileUtils.rm_f "bar.txt"
```

On introduit une méthode qui reçoit un bloc — argument implicite !

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 "bar.txt", ["123", "456", "789"] do
  assert_equal "12 bar.txt\n", %x{wc -c bar.txt}
end
```

On introduit une méthode qui reçoit un bloc — argument implicite !

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 "bar.txt", ["123", "456", "789"] do
  assert_equal "12 bar.txt\n", %x{wc -c bar.txt} #<--
end
```

A. Quel est le rapport avec le devoir #1… et #2 ?

Quel est le rapport avec le devoir #1… et #2 ?

Une partie de la note est attribuée pour la «Qualité générale du code»

Quel est le rapport avec le devoir #1… et #2 ?

Une partie de la note est attribuée pour la «Qualité générale du code»

Des tests automatiques détaillés sont disponibles

Quel est le rapport avec le devoir #1… et #2 ?

Une partie de la note est attribuée pour la «Qualité générale du code»

Des tests automatiques détaillés sont disponibles

$\Rightarrow$ Une fois que votre script fonctionne correctement (tous les tests passent), alors vous pouvez sans danger retravailler votre code pour améliorer sa qualité

Quel est le rapport avec le devoir #1… et #2 ?

Une partie de la note est attribuée pour la «Qualité générale du code»

Des tests automatiques détaillés sont disponibles

⇒ Une fois que votre script fonctionne correctement (tous les tests passent), alors vous pouvez sans danger retravailler votre code pour améliorer sa qualité

KISS & DRY !

Références

M. Fowler.

Refactoring—Improving the Design of Existing Code.

Addison-Wesley, 1999.

J. Rasmusson.

The Agile Samurai—How Agile Masters Deliver Great Software.

The Pragmatic Bookshelf, 2010.

J. Visser.

Building Maintainable Software—Ten Guidelines for Future-Proof Code (Java Edition).

O'Reilly, 2016.

Autres références

- «*Get a Whiff of This*», par Sandy Metz (RailsConf 2016) : Une vidéo (38 minutes) qui présente une vue d'ensemble des *Code Smells* ainsi que quelques exemples.
  `https://www.youtube.com/watch?v=PJjHfa5yxlU`

- Version préliminaire de «*Building Maintainable Software*» (Visser, 2016) : `http://www.labunix.uqam.ca/~tremblay_gu/MGL7460/Liens/Building_Maintainable_Software_SIG.pdf`