Sensibilisation aux bonnes pratiques de la programmation

Qualité Logicielle

Isabelle BLASQUEZ
@iblasquez

2017

Isabelle BLASQUEZ

@iblasquez

Enseignement : Génie Logiciel

Recherche : Développement logiciel agile

ICSTUG #IUTAgile

CodeWeek.

#Software Craftsmanship

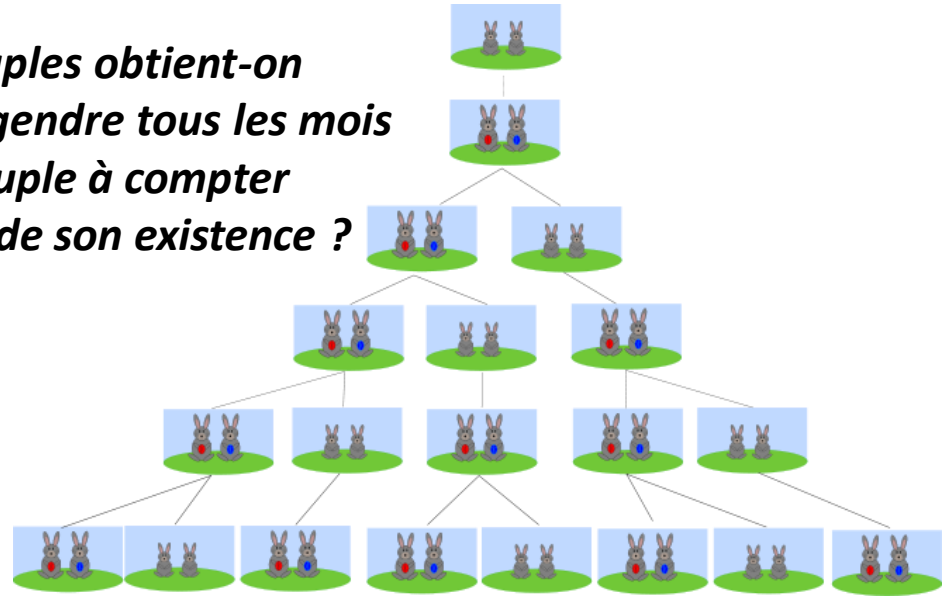
MUSEOMIX LIMOUSIN

La suite de Fibonnacci implémentée par 6 développeurs différents...

Code Written By A CS 101 Student

```java
public int fibonacci(int x) {
    if (x == 1) {
        return 1;
    } else if (x == 2) {
        return 1;
    } else {
        return fibonacci(x - 1) + fibonacci(x - 2);
    }
}
```

Combien de couples obtient-on
si chaque couple engendre tous les mois
un nouveau couple à compter
du troisième mois de son existence ?



Code Written At A Startup

```java
// TODO add Javadoc comments
/**
* getFibonacciNumber
*/
// TODO Should we move this to a different file?
public int getFibonacciNumber(int n) {
    // TODO Stack may overflow with recursive implementation, switch over to
    // iteration approach at some point?
    if (n < 0) {
        // TODO This should probably throw an exception. Or maybe just print
        // a log message?
        return -1;
    } else if (n == 0) {
        // TODO Generalize the initial conditions?
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        // TODO Spend some time with my family and kids, I've been at work for
        // over 48 hours straight.
        return getFibonacciNumber(n - 1) + getFibonnaciNumber(n - 2);
    }
}
```

Code Written At A Hackathon

```java
public int getFibonacciNumber(int n) {
    switch(n) {
        case 1: return 1;
        case 2: return 1;
        case 3: return 2;
        case 4: return 3;
        case 5: return 5;
        case 6: return 8;
        case 7: return 13;
        default:
            // good enough for the demo, lol
            return -1;
    }
}
```

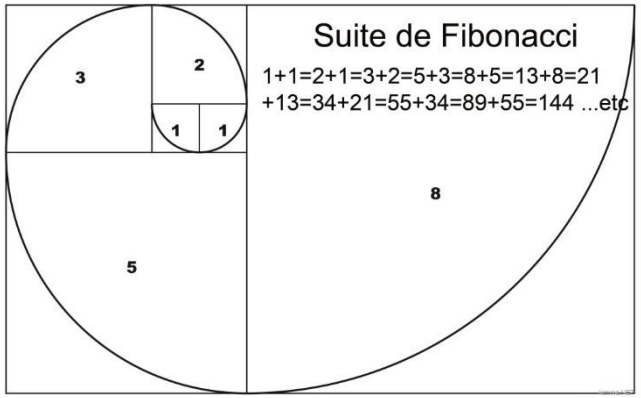
Extrait : https://twitter.com/TeaDrivenDev/status/724653521933615104 En savoir plus sur la suite de Fibonnaci : https://fr.wikipedia.org/wiki/Suite_de_Fibonacci

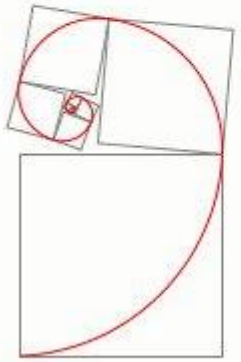
Isabelle BLASQUEZ

La suite de Fibonnacci implémentée par 6 développeurs différents... (suite)

Correspondance Fibonnacci et nombre d'or

http://www.wonderful-art.fr/le-saviez-vous-la-formule-secrete-de-l-harmonie-le-nombre-d-or





Code Written At A Large Company

```
/**
* getFibonacciNumber is a method that, given some index n, returns the nth
* Fibonacci number.
* @param n The index of the Fibonacci number you wish to retrieve.
* @return The nth Fibonacci number.
*/
public CustomInteger64 getFibonacciNumber(CustomInteger64 n) {
    FibonacciDataViewBuilder builder =
        FibonacciDataViewBuilderFactory.createFibonacciDataViewBuilder(
        new FibonacciDataViewBuilderParams(n, null, null, 0, null));
    if (builder == FibonacciDataViewBuilderConstants.ERROR_STATE) {
        throw new FibonacciDataViewBuilderFactoryException();
    }
    FibonacciDataView dataView = builder.GenerateFibonacciDataView(this);
    if (dataView == FibonacciDataViewConstants.ERROR_STATE) {
        throw new FibonacciDataViewGenerationException();
    }
    return dataView.accessNextFibonacciNumber(null, null, null);
}
```

Code Written By A Math Ph.D.

```
public int getFibonacciNumber(int n) {
    return (int) divide(subtract(exponentiate(phi(), n), exponentiate(psi(), n)),
        subtract(phi(), psi()));
}

public double exponentiate(double a, double b) {
    if (equal(b, zero())) {
        return one();
    } else {
        return multiply(a, exponentiate(a, subtract(b, one())));
    }
}

public double phi() {
    return divide(add(one(), sqrt(add(one(), one(), one(), one(), one()))),
        add(one(), one()));
}

public double psi() {
    return subtract(one(), phi());
}
```

Code Written By Your Cat

```
public static final int UNITE = 1;
public static final int UNITED = 2;

// meowwwww meow
public int meow(int KITTENS_OF_THE_WORLD) {
    // MEOW
    if (KITTENS_OF_THE_WORLD < UNITED) {
        return KITTENS_OF_THE_WORLD;
    } else {
        // meeoowwwwwwww
        // meooooowwwwwwwwwwwwwwwwwwww
        return meow(KITTENS_OF_THE_WORLD - UNITE)
            + meow(KITTENS_OF_THE_WORLD - UNITED);
    }
}
```

Extrait : https://twitter.com/TeaDrivenDev/status/724653521933615104
http://www.wonderful-art.fr/le-saviez-vous-la-formule-secrete-de-l-harmonie-le-nombre-d-or et https://twitter.com/fermatslibrary/status/838042125522587648

Isabelle BLASQUEZ

Quand vous programmez, vous dépensez plus de temps à lire du code qu'à en écrire

Vous manipulez les blocs de codes source de la même manière qu'un sculpteur le fait pour des blocs d'argile.
Donc un langage qui génère un code source désagréable à lire est aussi horripilant pour un programmeur que l'est une terre grumeleuse pour un sculpteur.

Paul Graham
(investisseur capital-risque et développeur Lisp)

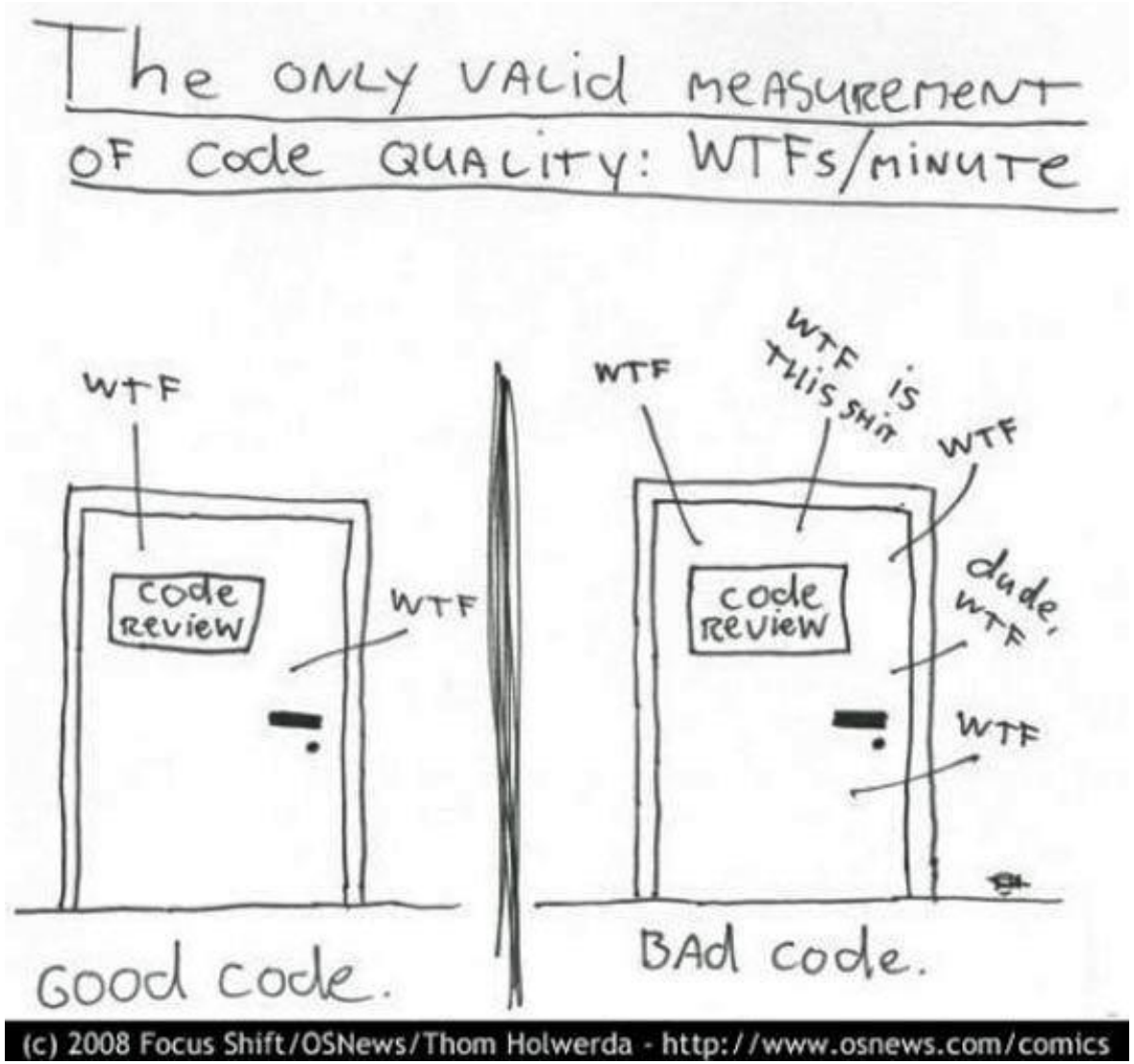
Extrait : *http://paulgraham.com/pypar.htm*
en français: http://www.developpez.com/actu/57222/Comment-reperer-la-creme-des-developpeurs-Pour-un-docteur-en-informatique-les-programmeurs-les-plus-doues-apprennent-des-langages-esoteriques/

Que fait ce code ?

```java
int f1(int a1, List<Integer> a2)
{
  int a5 = 0;
  for (int a3 = 0; a3 < a2.size() && a3 < a1; a3++) {
    int a6 = a2.get(a3);
    if (a6 >= 0) {
      System.out.println(a6 + " ");
    } else {
      a5++;
    }
  }
  System.out.println("\n");
  return a5;
}
```

Code extrait de : https://a-nickels-worth.blogspot.fr/2016/04/a-guide-to-naming-variables.html

Est-ce du code de qualité ?



Bien sûr, vous écrivez toujours un code de qualité, facile à comprendre, à étendre et à modifier …

- ❑ **En nommant correctement votre code pour bien montrer son intention,**
- ❑ **En limitant les code smells grâce au refactoring …**
- ❑ **… afin de faire émerger une conception simple**

L’art de bien nommer son code

Isabelle BLASQUEZ

Au fait, que fait ce code ?

```java
int f1(int a1, List<Integer> a2)
{
  int a5 = 0;
  for (int a3 = 0; a3 < a2.size() && a3 < a1; a3++) {
    int a6 = a2.get(a3);
    if (a6 >= 0) {
      System.out.println(a6 + " ");
    } else {
      a5++;
    }
  }
  System.out.println("\n");
  return a5;
}
```

Renommer les variables
ALT+SHIFT+R (Eclipse)

```
f1 → processElements
a1 → numResults
a2 → collection
a3 → count
a5 → result
a6 → num
```

```java
int processElements(int numResult, List<Integer> collection)
{
  int result = 0;
  for (int count = 0; count < collection.size() && count < numResult; count ++) {
    int num = collection.get(count);
    if (num >= 0) {
      System.out.println(num + " ");
    } else {
      result++;
    }
  }
  System.out.println("\n");
  return result;
}
```

Code extrait de : https://a-nickels-worth.blogspot.fr/2016/04/a-guide-to-naming-variables.html

C'est mieux comme ça ....

```java
int f1(int a1, List<Integer> a2)
{
  int a5 = 0;
  for (int a3 = 0; a3 < a2.size() && a3 < a1; a3++) {
    int a6 = a2.get(a3);
    if (a6 >= 0) {
      System.out.println(a6 + " ");
    } else {
      a5++;
    }
  }
  System.out.println("\n");
  return a5;
}
```

Renommer les variables
ALT+SHIFT+R (Eclipse)

```
f1 → printFirstNPositive
a1 → n
a2 → c
a3 → i
a5 → result
a6 → num
```

Un nommage explicite permet de contextualiser le code et montre clairement son intention

```java
int printFirstNPositive(int n, List<Integer> c)
{
  int skipped = 0;
  for (int i = 0; i < c.size() && i < n; i ++) {
    int mayBePositive = c.get(i);
    if (mayBePositive >= 0) {
      System.out.println(mayBePositive + " ");
    } else {
      skipped++;
    }
  }
  System.out.println("\n");
  return skipped;
}
```

Code extrait de : https://a-nickels-worth.blogspot.fr/2016/04/a-guide-to-naming-variables.html

Bien sûr, vous écrivez toujours un code très lisible (clair et expressif) …

c'est un tri de tableau
(qui représente quoi ?)

```
List<int[]> theList = new ArrayList<>();

public List<int[]> getFlg() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

… en fonction de la valeur stockée en index 0
(qui représente quoi ?)

et qui renvoie quoi ?

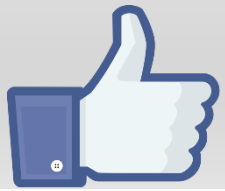
… qui doit être égale à 4
(pourquoi ?)

Exemple extrait de :

C’est pas mieux là ?...

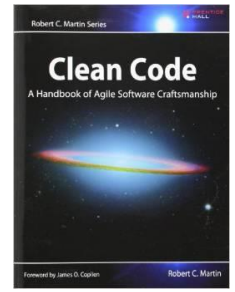
```
List<Cell> gameBoard = new ArrayList<Cell>();

public List<Cell> getFlaggedCell() {
   List<Cell> flaggedCells = new ArrayList<Cell>();
   for (Cell cell : gameBoard)
      if (cell.isFlagged())
         flaggedCells.add(cell);
   return flaggedCells;
}
```



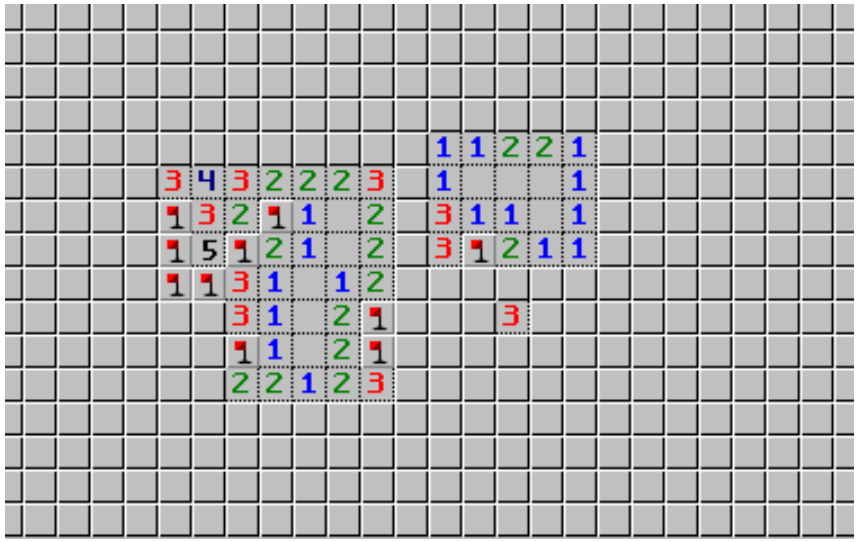
Exemple extrait de :

… un code qui révèle son intention !




```
List<int[]> theList = new ArrayList<>();

public List<int[]> getFlg() {
   List<int[]> list1 = new ArrayList<int[]>();
   for (int[] x : theList)
      if (x[0] == 4)
         list1.add(x);
   return list1;
}
```



Un nommage explicite et un partage des responsabilités permet de contextualiser le code et de révéler clairement son intention

```
List<Cell> gameBoard = new ArrayList<Cell>();

public List<Cell> getFlaggedCell() {
   List<Cell> flaggedCells = new ArrayList<Cell>();
   for (Cell cell : gameBoard)
      if (cell.isFlagged())
         flaggedCells.add(cell);
   return flaggedCells;
}
```

Renommer les variables
ALT+SHIFT+R (Eclipse)

- `getFlg()` → `getFlaggedCell()`
- `theList` → `gameBoard`
- `x` → `cell`
- `List1` → `flaggedCells`

Faire apparaître une nouvelle classe responsable de la cellule

Classe `Cell` et méthode `isFlagged`

Isabelle BLASQUEZ

Quelques recommandations pour rendre votre code expressif (1/3)

Attention à ne pas tomber dans l'excès inverse d'un nommage trop long !!!

1. Omit words that are obvious given a variable's or parameter's type

```
// Bad:
String nameString;
DockableModelessWindow dockableModelessWindow;

// Better:
String name;
DockableModelessWindow window;
```

```
// Bad:
List<DateTime> holidayDateList;
Map<Employee, Role> employeeRoleHashMap;

// Better:
List<DateTime> holidays;
Map<Employee, Role> employeeRoles;
```

```
// Bad:
mergeTableCells(List<TableCell> cells)
sortEventsUsingComparator(List<Event> events,
    Comparator<Event> comparator)

// Better:
merge(List<TableCell> cells)
sort(List<Event> events, Comparator<Event> comparator)
```

⇒ *Eviter la notation hongroise :*

```
short nIndex;
char cChar;
float* pfVal;
```

2. Omit words that don't disambiguate the name

```
// Bad:
finalBattleMostDangerousBossMonster;
weaklingFirstEncounterMonster;

// Better:
boss;
firstMonster;
```

Extraits : http://journal.stuffwithstuff.com/2016/06/16/long-names-are-long/
https://fr.wikipedia.org/wiki/Notation_hongroise

Quelques recommandations pour rendre votre code expressif (2/3)

3. Omit words that are known from the surrounding context

```
// Bad:
class AnnualHolidaySale {
  int _annualSaleRebate;
  void promoteHolidaySale() { ... }
}

// Better:
class AnnualHolidaySale {
  int _rebate;
  void promote() { ... }
}
```

4. Omit words that don't mean much of anything

In many cases, the words carry no meaningful information. They're just fluff or jargon. Usual suspects include: `data`, `state`, `amount`, `value`, `manager`, `engine`, `object`, `entity`, and `instance`.

A good name paints a picture in the mind of the reader.

Extraits de http://journal.stuffwithstuff.com/2016/06/16/long-names-are-long/
Voir aussi http://journal.stuffwithstuff.com/2009/06/05/naming-things-in-code/ pour d'autres recommandations…

Quelques recommandations pour rendre votre code expressif (3/3)

Move Simple Comments Into Variable Names

```
// BAD
String name; // First and last name
// GOOD
String fullName;

// BAD
int port; // TCP port number
// GOOD
int tcpPort;

// BAD
// This is derived from the JSON header
String authCode;
// GOOD
String jsonHeaderAuthCode;
```

Extrait de https://a-nickels-worth.blogspot.fr/2016/04/a-guide-to-naming-variables.html
A lire pour découvrir d'autres recommandations…

Use Idioms Where Meaning is Obvious

Unlike the cliches, there are some idioms that are so widely-understood and unabused that they're safe to use, even if by the letter of they law they're too cryptic. Some examples are (these are Java/C specific examples, but the same principles apply to all languages):

- use of `i`, `j` and `k` in straightforward `for` loops
- use of `n` for a limit/quantity when it's obvious what it would do
- use of `e` for an Exception in a `catch` clause

```
// OK
for (int i = 0; i < hosts.size(); i++) { }


// OK
String repeat(String s, int n);
```

Warning: idioms should only be used in cases where it's *obvious* what they mean.

Astuce pour vérifier la pertinence du « nommage » dans vos classes métiers

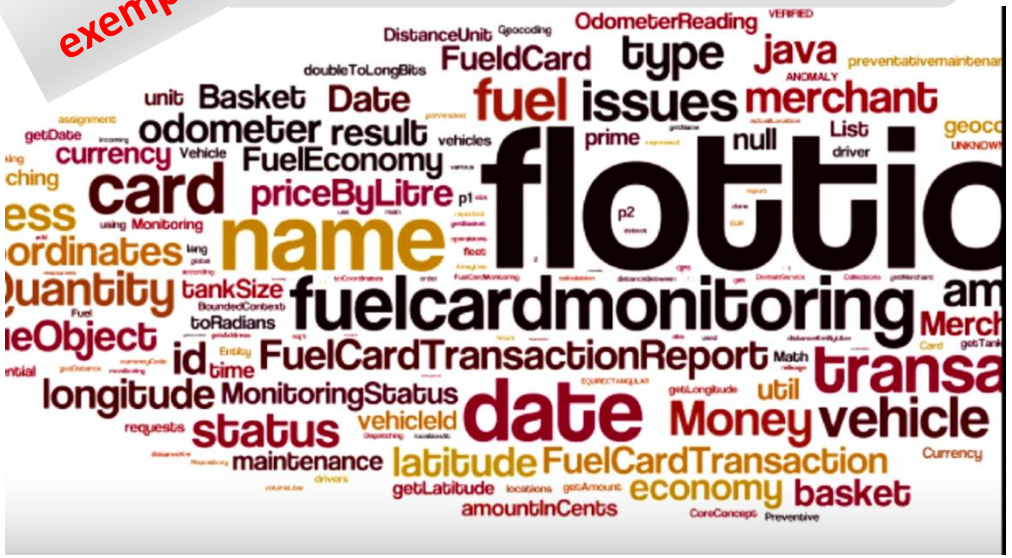
Passer son code dans un word cloud

Mauvais exemple

La technique est mise en avant

Bon exemple

Le **métier** est mis en avant



Pour la génération de world cloud :

- **http://sourcecodecloud.codeplex.com** (Java, c#, Java, VB.NET)
- **http://www.wordclouds.com/** (world cloud classique)

Extrait de la video : https://www.youtube.com/watch?v=h3DLKrvp5V8
Dont un retour est disponible ici : https://blog.groupe-sii.com/devoxx-france-2016-architecture-methodologie/#ddd

En panne d'inspiration pour trouver le bon terme métier ?

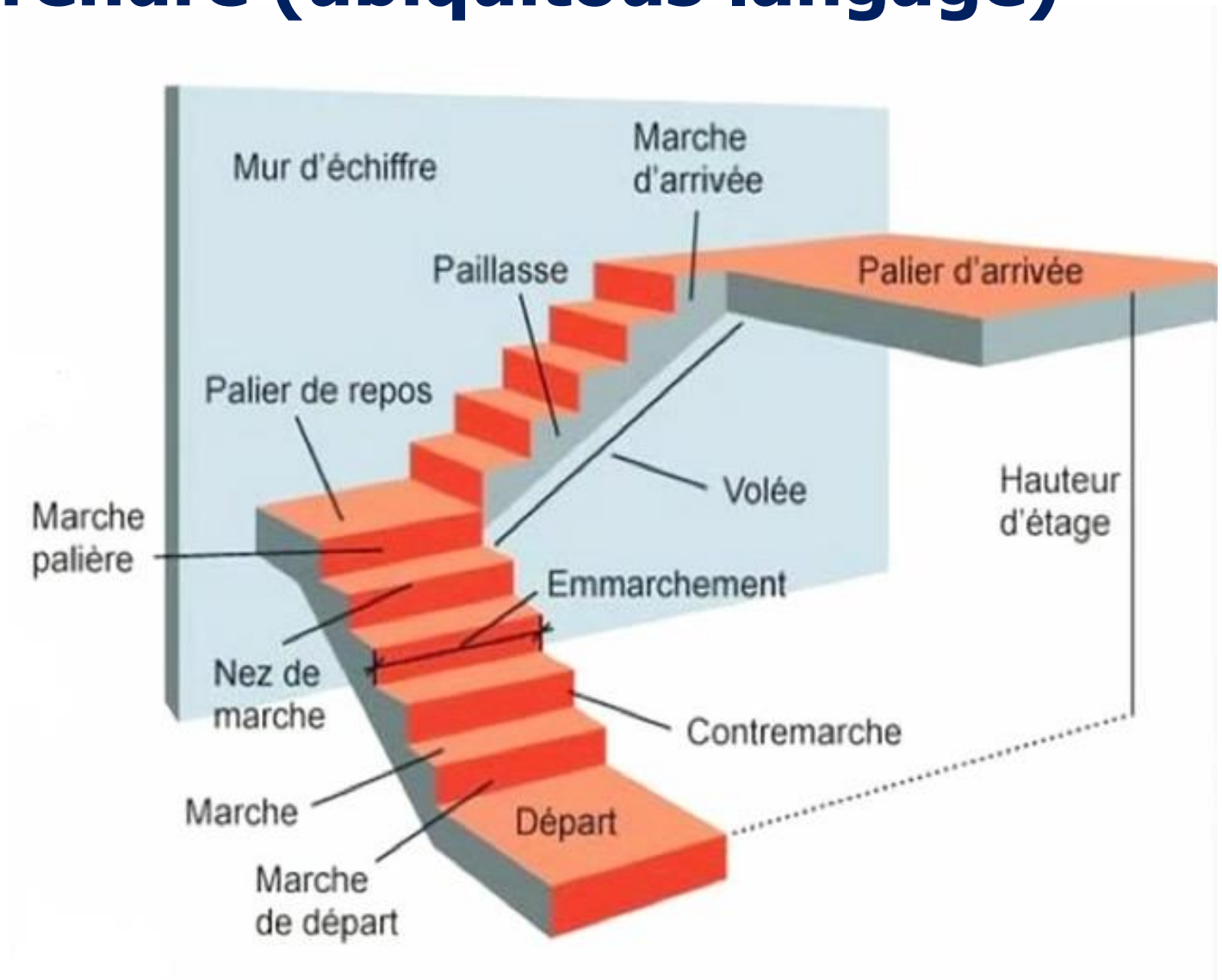
Astuce : un moteur de synonymes peut vous aider …

http://www.thesaurus.com/

http://www.synonymy.com/

…

Interaction avec le métier pour utiliser le *bon* terme et bien le comprendre (ubiquitous langage)



Extrait de la video : https://www.youtube.com/watch?v=h3DLKrvp5V8 Première approche DDD: http://blog.infosaurus.fr/public/docs/DDDViteFait.pdf
Dont un retour est disponible ici : https://blog.groupe-sii.com/devoxx-france-2016-architecture-methodologie/#ddd

Standards de codage

Isabelle BLASQUEZ

Montrer l'intention & la formater correctement

Exemples de standards de codage (Java Google Style)

Accessible sur https://google.github.io/styleguide/javaguide.html

5.2.2 Class names

Class names are written in UpperCamelCase.

Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`.

5.2.3 Method names

Method names are written in lowerCamelCase.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores may appear in JUnit *test* method names to separate logical components of the name. One typical pattern is `test<MethodUnderTest>_<state>`, for example `testPop_emptyStack`. There is no One Correct Way to name test methods.

5.2.4 Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

5.3 Camel case: defined

Prose form	Correct	Incorrect
"XML HTTP request"	`XmlHttpRequest`	`XMLHTTPRequest`
"new customer ID"	`newCustomerId`	`newCustomerID`
"inner stopwatch"	`innerStopwatch`	`innerStopWatch`
"supports IPv6 on iOS?"	`supportsIpv6OnIos`	`supportsIPv6OnIOS`
"YouTube importer"	`YouTubeImporter` `YoutubeImporter` *	

Java Google Style :
https://google.github.io/styleguide/javaguide.html

Code Conventions for the Java ™ Programming Language :
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

Respectez les standards de codage

(que l'on vous donne…
…. ou inspirez-vous de ceux qui existent)

Conventions de codage C# :
https://msdn.microsoft.com/fr-fr/library/ff926074.aspx

Google Style Guides : https://github.com/google/styleguide
(C++ Style Guide, Objective-C Style Guide, Java Style Guide, Python Style Guide, R Style Guide, Shell Style Guide,HTML/CSS Style Guide, JavaScript Style Guide, AngularJS Style Guide, Common Lisp Style Guide, and Vimscript Style Guide **)**

Swift API Design Guidelines :
https://swift.org/documentation/api-design-guidelines

etc …

Limiter les commentaires

… Des commentaires toujours très pertinents ?

COMMENTS IN REAL LIFE

```
/*
* A comment to please checkstyle
*/
```

```
/*
* Set the port
*
* @params port
*/
public void setPort(Port port) {this.port=port}
```

```
...
        } /* end for */
        dao.flush();
      default :
        break;
      } /* end switch */
    } /* end if */
  } /* end if */
} catch ...
```

Et là, je fais confiance aux commentaires ?

```
// Copie les attributs du ActionForm dans le VO
// en utilisant leur nom
return null;
```

Extrait : http://avernois.github.io/prez-clean_code/#/5/1
https://twitter.com/AlexisHassler/status/844964683870339073

Pertinence des commentaires …

Over Commented Code

Extrait de : https://twitter.com/CiaranMcNulty/status/517654863623487488

Code Comments

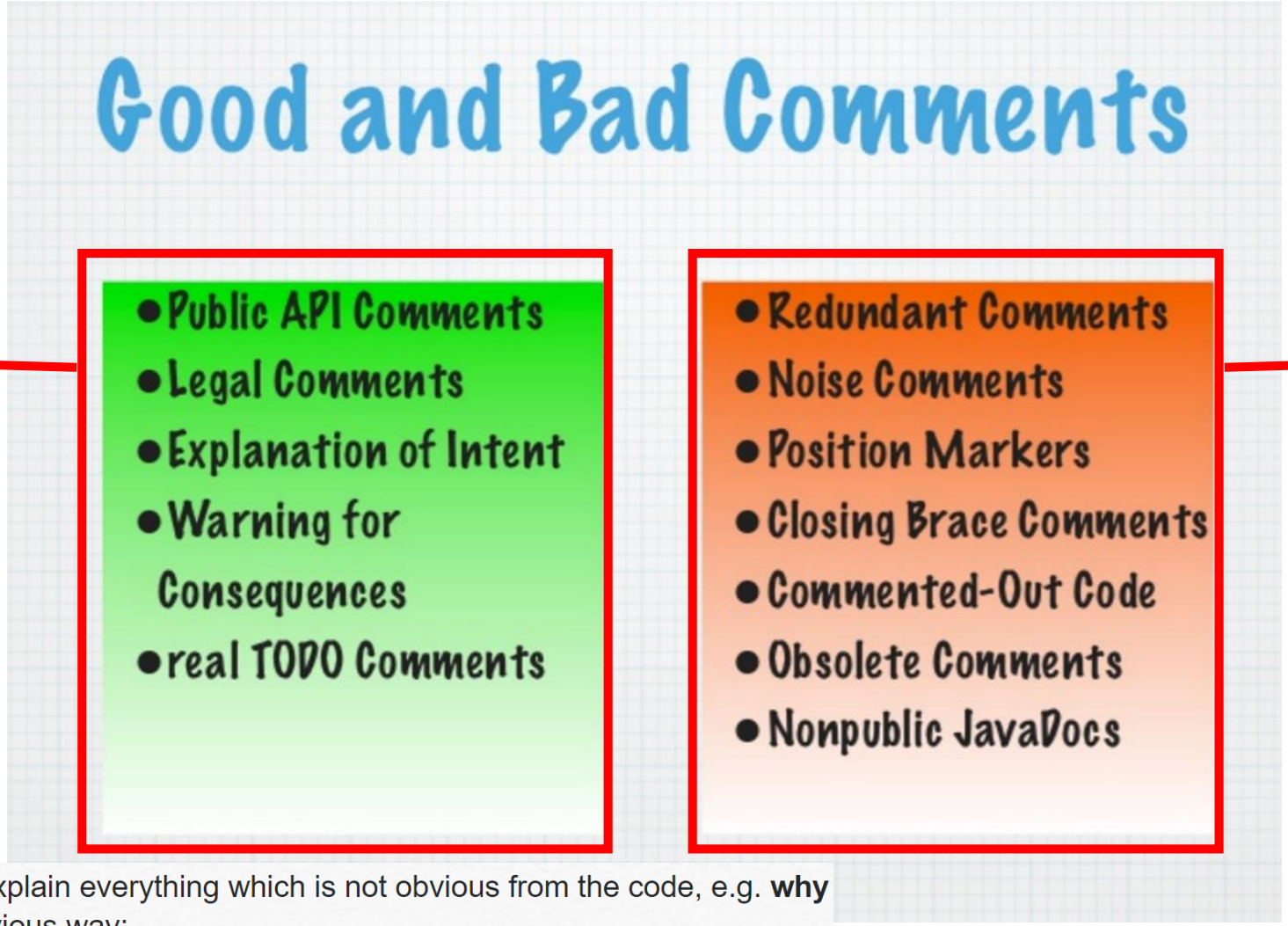
Extrait de : https://twitter.com/nzkoz/status/538892801941848064

Extrait de : https://twitter.com/moh/status/659476321999818752

Nécessité du commentaire ?

Pertinence du commentaire : Eclaircissement ou confusion ?

mais parfois avec des commentaires nécessaires ...



But comments **should** be used to explain everything which is not obvious from the code, e.g. **why** the code is written a certain non-obvious way:

```
// need to reset foo before calling bar due to a bug in the foo component.
foo.reset()
foo.bar();
```

Extrait : http://fr.slideshare.net/hebel/clean-code-vortrag032009pdf
http://softwareengineering.stackexchange.com/questions/285787/clean-code-comments-vs-class-documentation

Comments are always failure

Uncle Bob

Don't comment bad code. Rewrite it.

Brian W. Kernighan, P.J. Plaugher

Extrait : http://avernois.github.io/prez-clean_code/#/5/2

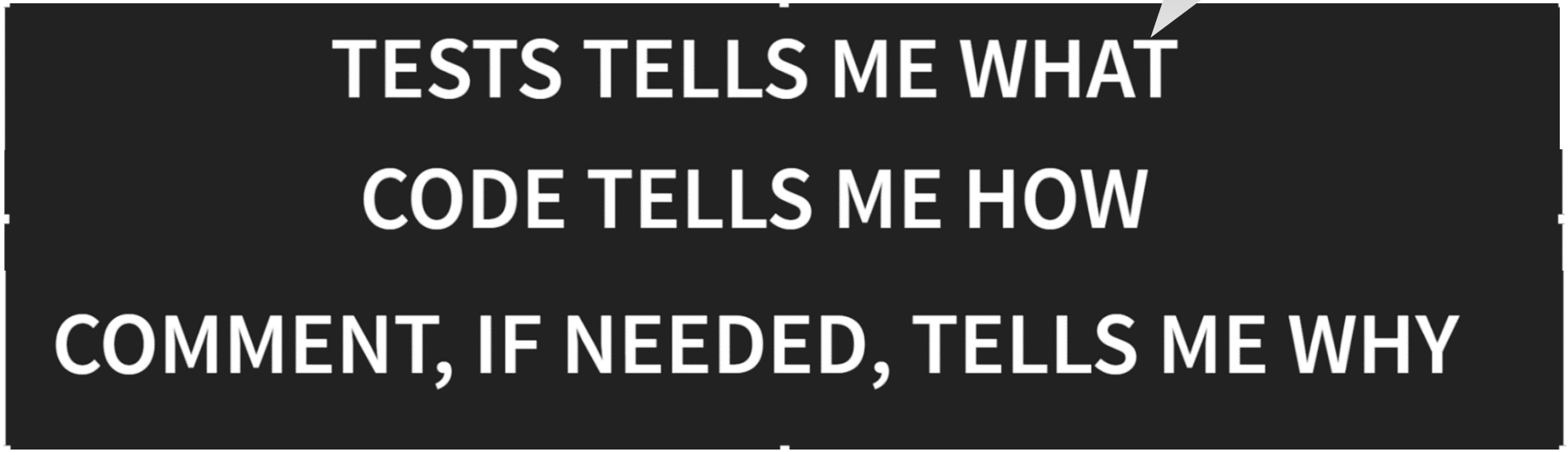
Quid des commentaires ?

Comments

There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"? Can you refactor the code so the comments aren't required? And remember, you're writing comments for people, not machines.

Extrait : http://blog.codinghorror.com/code-smells/

Les tests peuvent ainsi aussi servir de documentation…



Extrait : http://avernois.github.io/prez-clean_code/#/5/4

Bien sûr, vous écrivez toujours un code de qualité, facile à comprendre, à étendre et à modifier …

☑ **En nommant correctement votre code pour bien montrer son intention,**

➡ ❑ **En limitant les code smells grâce au refactoring …**

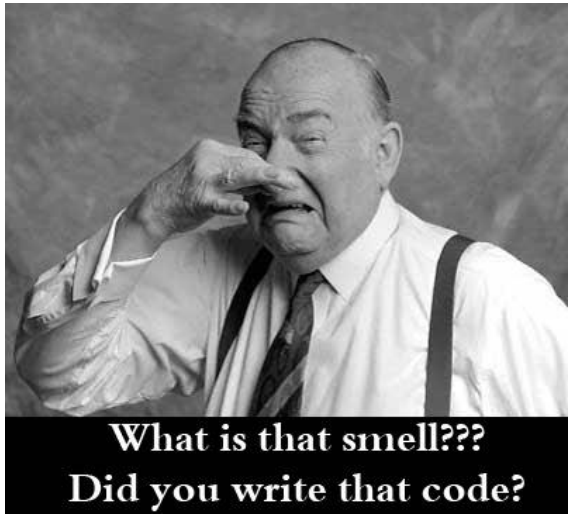
❑ **… afin de faire émerger une conception simple**

L’art de limiter les code smells

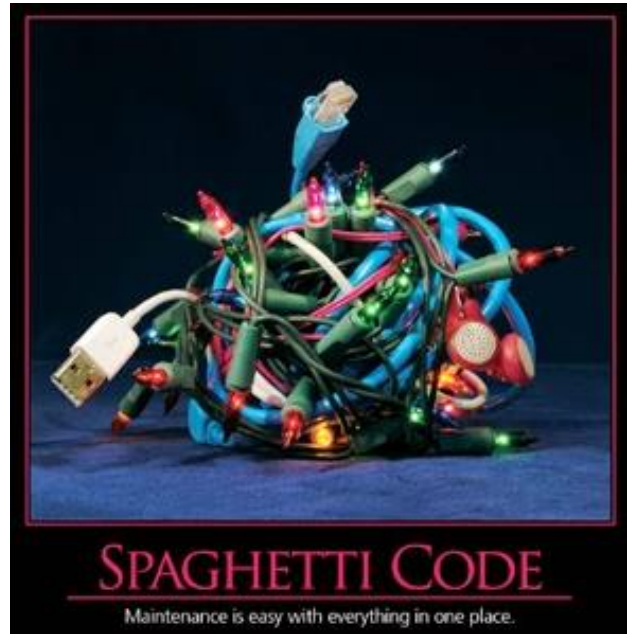
Isabelle BLASQUEZ

Quand le code n'est pas *clean*

… il *smell*

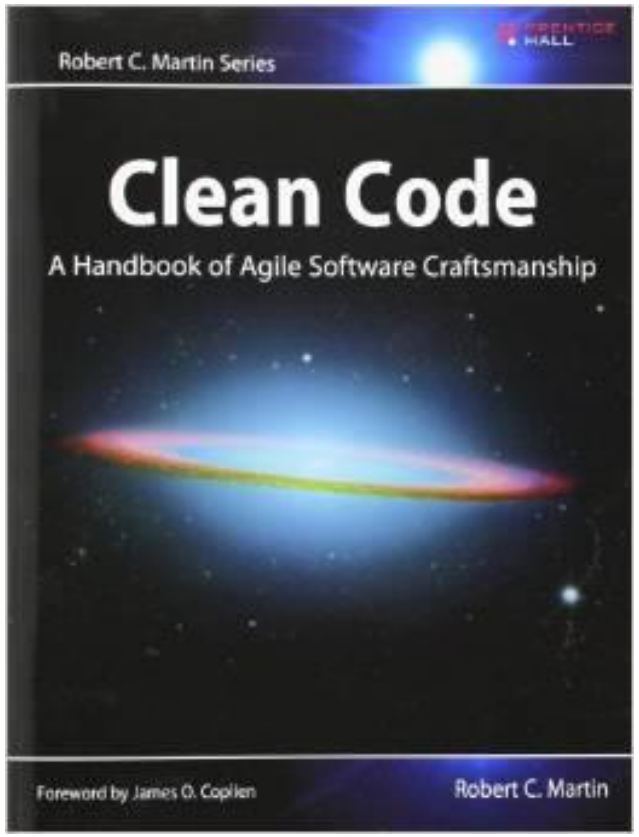


et donne bien souvent du

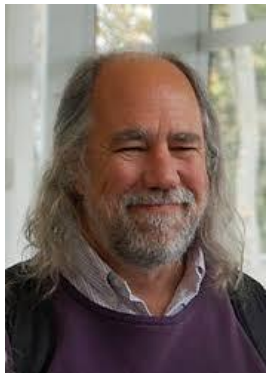


Images extraites :
http://de.slideshare.net/iksgmbh/clean-codevortraggearconf
http://www.rogoit.de/webdesign-typo3-blog-duisburg/clean-code-regeln-smells-und-heuristiken/

Bjarne Stroustrup
(inventeur du C++)
Le code *clean* doit être élégant et efficace, et la gestion des erreurs doit être totale.



Dave Thomas
(fondateur d'OTI, pré Eclipse)
Le code *clean* peut être lu et amélioré par d'autres développeurs.



Grady Booch
(co-créateur d'UML)
Le code *clean* se caractérise par sa lisibilité.

Ron Jeffries
(co-fondateur de l'eXtreme Programming)
Le code *clean* est du code simple, expressif, sans duplication et avec peu d'abstraction.

Ward Cunningham
(inventeur du Wiki)
Le code est *clean* quand il correspond à tout ce qu'on peut en attendre.

Jetez un petit coup d'œil à : http://www.drims-academy.fr/clean-code-partie-1/

Une méthode bien trop longue ⇒ un code smell …

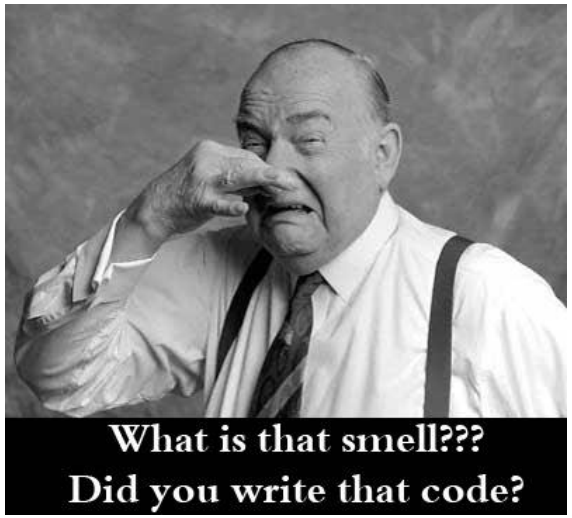


Image et Illustration extraits de : http://uneviededev.wordpress.com/2014/09/18/**le-syndrome-de-stockholm/**
http://www.rogoit.de/webdesign-typo3-blog-duisburg/clean-code-regeln-smells-und-heuristiken/

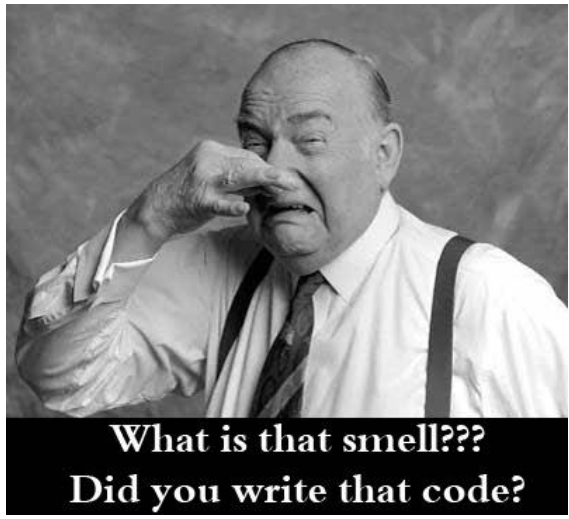
Exemple de code smells ...

Code Smells Within Classes

Comments	There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"? Can you refactor the code so the comments aren't required? And remember, you're writing comments for people, not machines.
Long Method	All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot. Refactor long methods into smaller methods if you can.
Long Parameter List	The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given methoc or use an object to combine the parameters.
Duplicated code	Duplicated code is the bane of software development. Stam out duplication whenever possible. You should always be on the lookout for more subtle cases of near-duplication, too. Don't Repeat Yourself!
Conditional Complexity	Watch out for large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time. Consider alternative object-oriented approaches such as decorator, strategy, or state.
Combinitorial Explosion	You have lots of code that does *almost* the same thing.. but with tiny variations in data or behavior. This can be difficult to refactor-- perhaps using generics or an interpreter?
Large Class	Large classes, like long methods, are difficult to read, understand, and troubleshoot. Does the class contain too many responsibilities? Can the large class be restructured or broken into smaller classes?

Code Smells Between Classes

Alternative Classes with Different Interfaces	If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.
Primitive Obsession	Don't use a gaggle of primitive data type variables as a poor man's substitute for a class. If your data type is sufficiently complex, write a class to represent it.
Data Class	Avoid classes that passively store data. Classes should contain data *and* methods to operate on that data, too.
Data Clumps	If you always see the same data hanging around together, maybe it belongs together. Consider rolling the related data up into a larger class.
Refused Bequest	If you inherit from a class, but never use any of the inherited functionality, should you really be using inheritance?
Inappropriate Intimacy	Watch out for classes that spend too much time together, or classes that interface in inappropriate ways. Classes should know as little as possible about each other.
Indecent Exposure	Beware of classes that unnecessarily expose their internals. Aggressively refactor classes to minimize their public surface. You should have a compelling reason for every item you make public. If you don't, hide it.
Feature Envy	Methods that make extensive use of another class may belong in another class. Consider moving this method to the class it is so envious of.
Lazy Class	Classes should pull their weight. Every additional class increases the complexity of a project. If you have a class that isn't doing enough to pay for itself, can it be collapsed or combined into another class?
Message Chains	Watch out for long sequences of method calls or temporary variables to get routine data. Intermediaries are dependencies in disguise.



Liste non exhaustive extraite de : http://blog.codinghorror.com/code-smells/
où vous trouverez de nombreux autres smell codes ...
Voir aussi la Taxonomy des codes Smell : http://badcodesmellstaxonomy.mikamantyla.eu/

Image : http://www.rogoit.de/webdesign-typo3-blog-duisburg/clean-code-regeln-smells-und-heuristiken/

Isabelle BLASQUEZ

Des code smell dans mes programmes ?

Source : http://www.memegen.fr/meme/zwq50s

Voyez-vous des code smell dans ce code ?

```
public class ShippingController extends Controller {
   private CartService cartService;

   public void process(ShippingForm form) throws Exception, SQLException {
      CartBean cartBean = cartService.getCart(form.getCardId());
      BigDecimal cost = null;

      if (cartBean.getTotal() <= 100 && form.getOption() == 2) { // 2 : Option Priority
         cost = new BigDecimal(4.99);
         if (form.getOption() == 1) { // expedited
            for (Item i : cartBean.getItems()) {
               if (i.getCat() == 'B') {
                  cost.add(new BigDecimal(2.99));
               }
               if (i.getCat() == 'O') { // other by weight
                  cost.add(new BigDecimal(i.getWeight() / 1000).multiply(new BigDecimal(1.99)));
               }
            }
         }

         if (form.getOption() == 2) { // priority
            for (Item i : cartBean.getItems()) {
               if (i.getCat() == 'B') {
                  cost.add(new BigDecimal(4.99));
               }
               if (i.getCat() == 'O') { // other by weight
                  cost.add(new BigDecimal(i.getWeight() / 1000).multiply(new BigDecimal(2.99)));
               }
            }
         }
      }
      form.setCost(cost);
   }
}
```

Magic Number

Duplication

Primitive Obsession

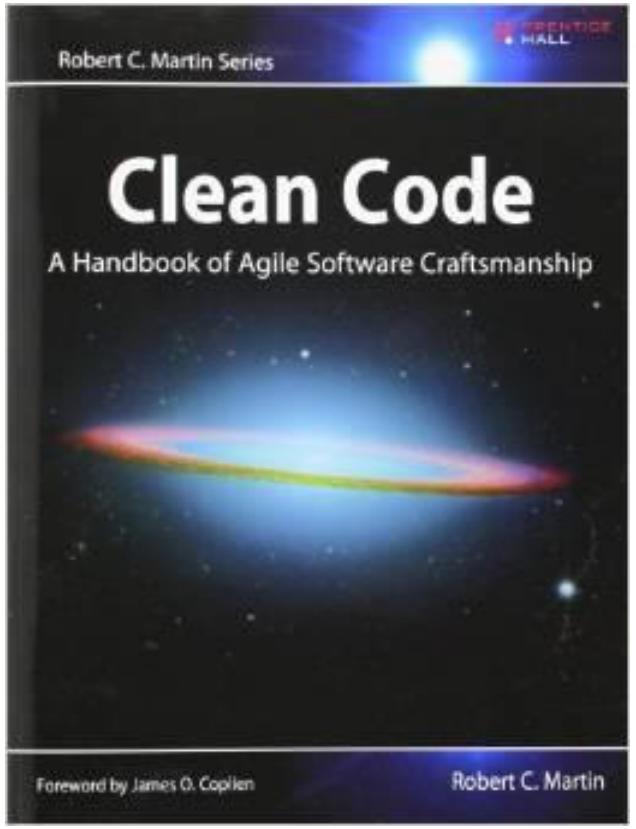
Mixed concerns (tech-biz)

Fuzzy Terminology

Code extrait de la présentation : http://www.slideshare.net/ThomasPierrain/ddd-reboot-english-version disponible en vidéo : https://www.youtube.com/watch?v=o3thkx3EuiA
Image : http://www.vie-en-zen.com/blog/comprendre-la-formule-moleculaire-dune-huile-essentielle/

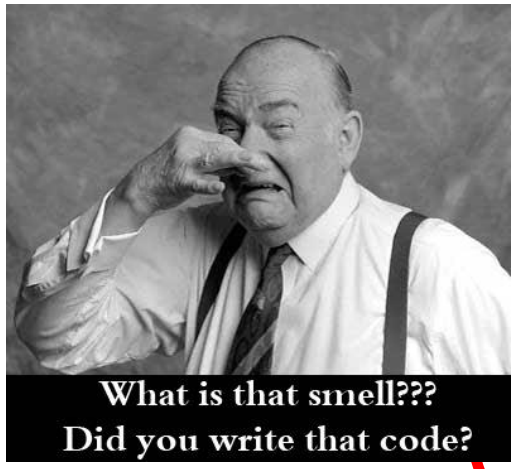
Un Code Smell : Que faire ?

Règle des boy-scouts (*The Boy Scout Rule*)
« ***Toujours laisser un endroit dans un état meilleur que celui où vous l'avez trouvé*** ».

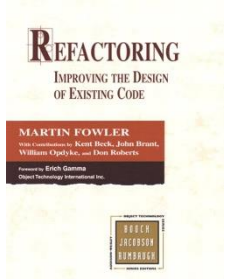


Images : http://memegenerator.net/instance/54386514
http://www.drims-academy.fr/author/athiefaine/
http://www.amazon.fr/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882

Un Code Smell : Que faire ?

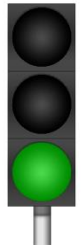


Un refactoring (remaniement)
consiste à changer
la structure interne d'un logiciel
sans en changer son comportement observable



Refactoring

Pas de refactoring sans test !



Extrait :
http://www.passetoncode.fr/cours/feux-tricolore

Techniques de refactoring

Catalog of Refactorings

http://refactoring.com/catalog/

Martin Fowler

10 December 2013

This catalog of refactorings includes those refactorings described in my original book on Refactoring, together with the Ruby Edition.

Using the Catalog ▶

Tags

- associations
- encapsulation
- generic types
- interfaces
- class extraction
- GOF Patterns
- local variables
- vendor libraries
- errors
- type codes
- method calls
- organizing data
- inheritance
- conditionals
- moving features
- composing methods

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Duplicate Observed Data
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Recompose Conditional
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Middle Man
- Remove Named Parameter
- Remove Parameter
- Remove Setting Method

Consolidate Conditional Expression

You have a sequence of conditional tests with the same result.

Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
  if (_seniority < 2) return 0;
  if (_monthsDisabled > 12) return 0;
  if (_isPartTime) return 0;
  // compute the disability amount
```

```
double disabilityAmount() {
  if (isNotEligableForDisability()) return 0;
  // compute the disability amount
```

A découvrir petit à petit ...

Isabelle BLASQUEZ

Jetez également un petit coup d'œil à …

https://sourcemaking.com/refactoring

- ★ Premium Stuff
- Design Patterns
- AntiPatterns
- Refactoring
  - Code Smells
  - Refactoring techniques
- UML

Extract Method

Problem

You have a code fragment that can be grouped together.

```
void printOwing() {
  printBanner();

  //print details
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstand
}
```

Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding
}
```

Java | C# | PHP | Python

Why Refactor

The more lines found in a method, the harder it is to figure out what the method does. This is the main reason for this refactoring.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

Benefits

- More readable code! Be sure to give the new method a name that describes the method's purpose: `createOrder()`, `renderCustomerInfo()`, etc.
- Less code duplication. Often the code that is found in a method can be reused in other places in your program. So you can replace duplicates with calls to your new method.
- Isolates independent parts of code, meaning that errors are less likely (such as if the wrong variable is modified).

How to Refactor

1. Create a new method and name it in a way that makes its purpose self-evident.
2. Copy the relevant code fragment to your new method. Delete the fragment from its old location and put a call for the new method there instead.

Find all variables used in this code fragment. If they are declared inside the fragment and not used outside of it, simply leave them unchanged – they will become local variables for the new method.

3. If the variables are declared prior to the code that you are extracting, you will need to pass these variables to the parameters of your new method in order to use the values previously contained in them. Sometimes it is easier to get rid of these variables by resorting to Replace Temp with Query.
4. If you see that a local variable changes in your extracted code in some way, this may mean that this changed value will be needed later in your main method. Double-check! And if this is indeed the case, return the value of this variable to the main method to keep everything functioning.

A découvrir petit à petit …

Isabelle BLASQUEZ

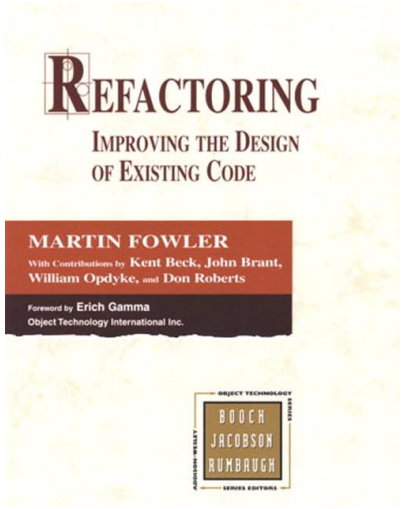
Aide mémoire : Exemple de refactorings à la rescousse de smells

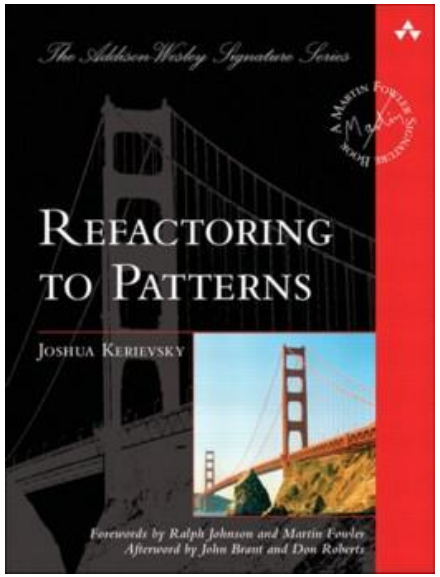
Smell	Refactoring
Alternative Classes with Different Interfaces: occurs when the interfaces of two classes are different and yet the classes are quite similar. If you can find the similarities between the two classes, you can often refactor the classes to make them share a common interface [F 85, K 43]	Unify Interfaces with Adapter [K 247]
	Rename Method [F 273]
	Move Method [F 142]
Combinatorial Explosion: A subtle form of duplication, this smell exists when numerous pieces of code do the same thing using different combinations of data or behavior. [K 45]	Replace Implicit Language with Interpreter [K 269]
Comments (a.k.a. Deodorant): When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous" [F 87]	Rename Method [F 273]
	Extract Method [F 110]
	Introduce Assertion [F 267]
Conditional Complexity: Conditional logic is innocent in its infancy, when it's simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. You implement several new features and suddenly your conditional logic becomes complicated and expansive. [K 41]	Introduce Null Object [F 260, K 301]
	Move Embellishment to Decorator [K 144]
	Replace Conditional Logic with Strategy [K 129]
	Replace State-Altering Conditionals with State [K 166]
Data Class: Classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. [F 86]	Move Method [F 142]
	Encapsulate Field [F 206]
	Encapsulate Collection [F 208]
Data Clumps: Bunches of data that that hang around together really ought to be made into their own object. A good test is to consider deleting one of the data values: if you did this, would the others make any sense? If they don't, it's a sure sign that you have an object that's dying to be born. [F 81]	Extract Class [F 149]
	Preserve Whole Object [F 288]
	Introduce Parameter Object [F 295]
Divergent Change: Occurs when one class is commonly changed in different ways for different reasons. Separating these divergent responsibilities decreases the chance that one change could affect another and lower maintenance costs. [F 79]	Extract Class [F 149]
Duplicated Code: Duplicated code is the most pervasive and pungent smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet essentially the same. [F76, K 39]	Chain Constructors [K 340]
	Extract Composite [K 214]
	Extract Method [F 110]
	Extract Class [F 149]
	Form Template Method [F 345, K 205]
	Introduce Null Object [F 260, K 301]
	Introduce Polymorphic Creation with Factory Method [K 88]
	Pull Up Method [F 322]
	Pull Up Field [F 320]
	Replace One/Many Distinctions with Composite [K 224]
	Substitue Algorithm [F 139]
	Unify Interfaces with Adapter [K 247]
Feature Envy: Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air. [F 80]	Extract Method [F 110]
	Move Method [F 142]
	Move Field [F 146]
Freeloader (a.k.a. Lazy Class): A class that isn't doing enough to pay for itself should be eliminated. [F 83, K 43]	Collapse Hierarchy [F 344]
	Inline Class [F 154]
	Inline Singleton [K 114]
Inappropriate Intimacy: Sometimes classes become far too intimate and spend too much time delving into each others' private parts. We may not be prudes when it comes to people, but we think our classes should follow strict, puritan rules. Over-intimate classes need to be broken up as lovers were in ancient days. [F 85]	Move Method [F 142]
	Move Field [F 146]
	Change Bidirectional Association to Unidirectional Association [F 20
	Extract Class [F 149]
	Hide Delegate [F 157]

A découvrir petit à petit ...

Extrait de : http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Réalisé à partir de :





Catalogue de refactoring en ligne :
http://refactoring.com/catalog/

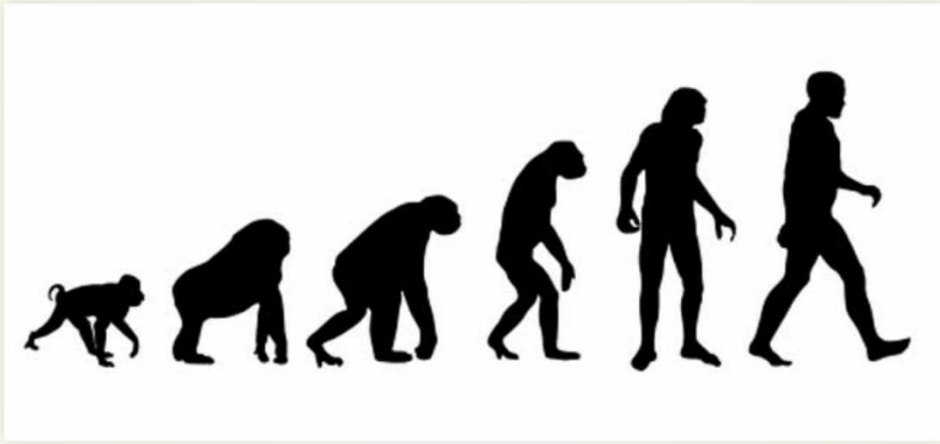
Isabelle BLASQUEZ

Vers une conception simple ...

Le refactoring contribue à faire émerger une ***conception plus simple et évolutive***

lisibilité
duplication
complexité

La bonne conception n'est jamais la première...



elle émerge

Extrait : http://michaelborde.fr/TddEn5Minutes-Presentation-RevealJs/#/15

Bien sûr, vous écrivez toujours un code de qualité, facile à comprendre, à étendre et à modifier …

☑ **En nommant correctement votre code pour bien montrer son intention,**

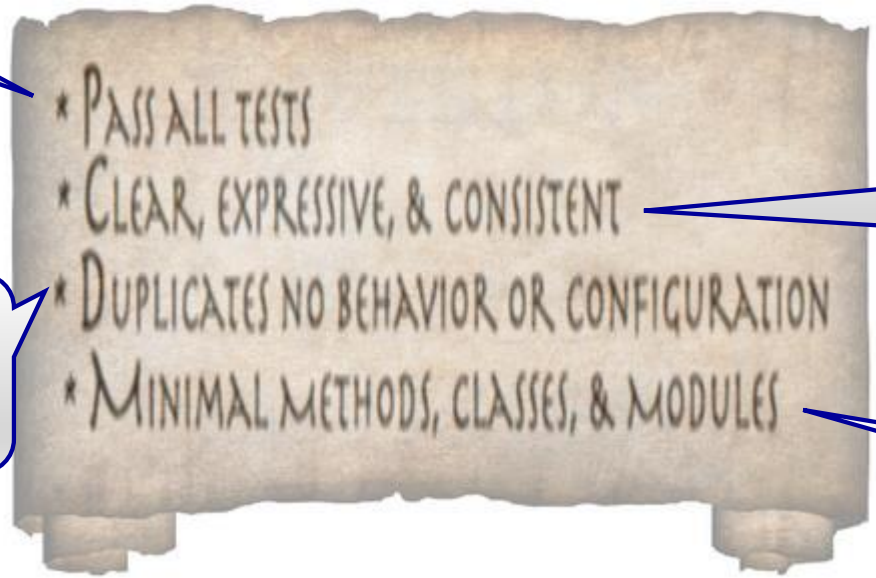
☑ **En limitant les code smells grâce au refactoring …**

➡ ❑ **… afin de faire émerger une conception simple**

Faire émerger
une conception simple …

Les règles de simplicité (eXtreme Programming)

1. Tous les tests passent



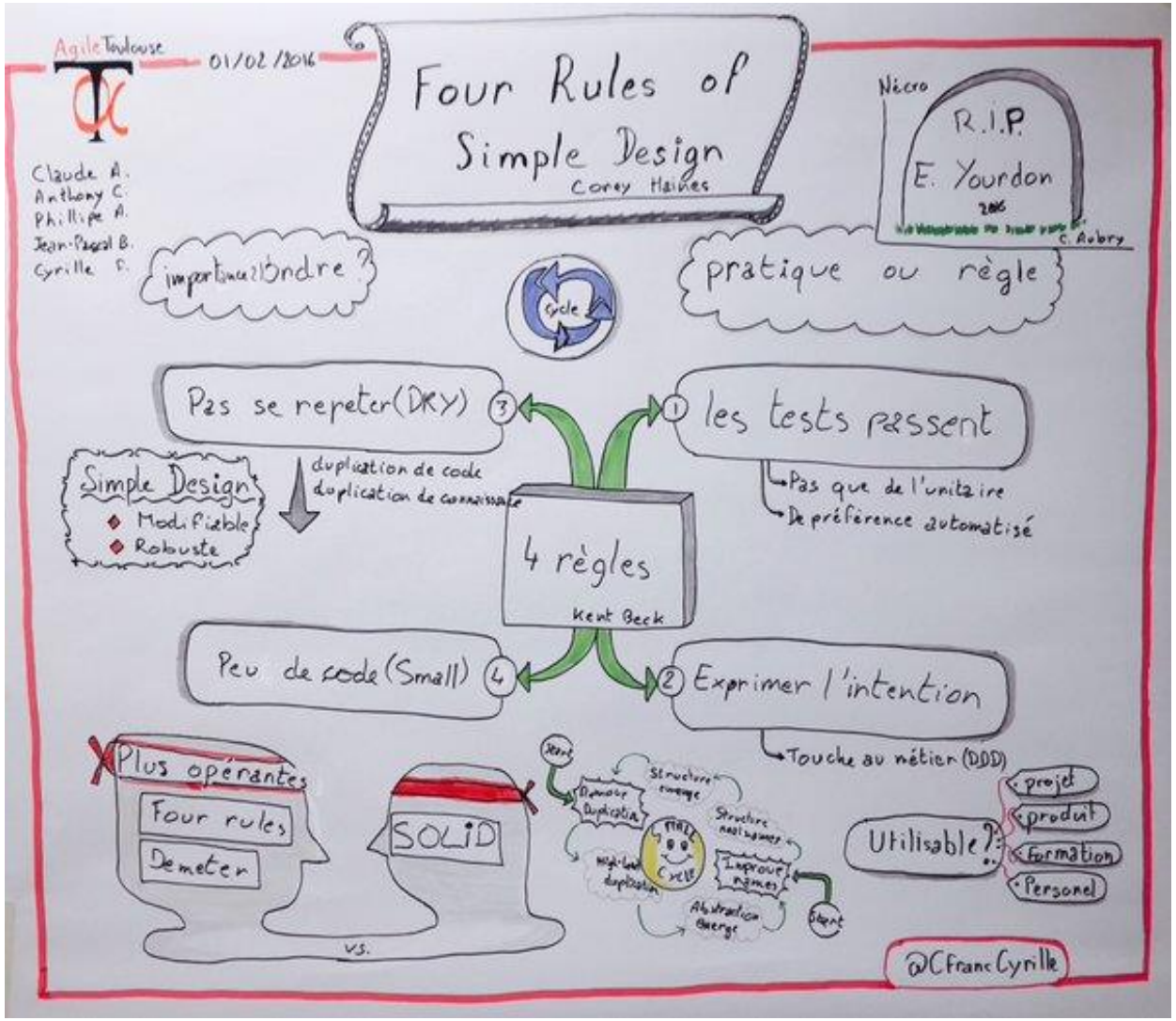
2. Révéler l'intention du code **(lisibilité)**

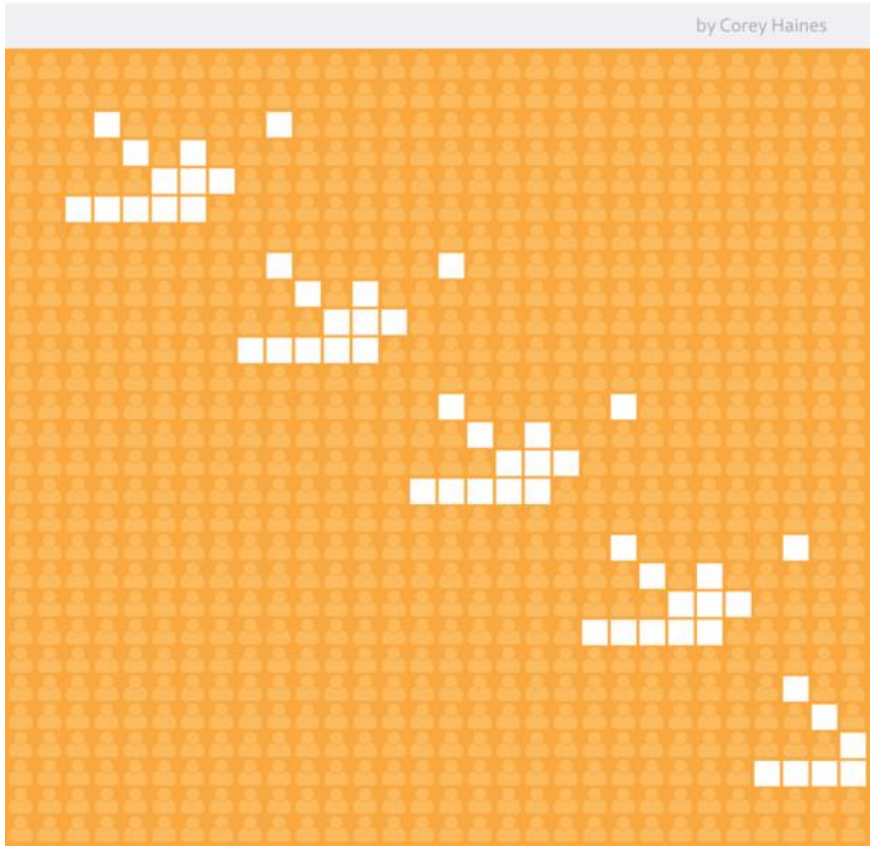
3. Pas de **duplication**: **DRY** (**D**on't **R**epeat **Y**ourself)

4. Le plus petit nombre d'éléments **(complexité)**

Extrait : http://c2.com/cgi/wiki?XpSimplicityRules
A lire également : http://martinfowler.com/bliki/BeckDesignRules.html
et https://leanpub.com/4rulesofsimpledesign

En savoir plus sur la conception simple ...





Facilitation graphique réalisée au club de lecture Agile Toulouse : https://klubat.scalingo.io/klub/19

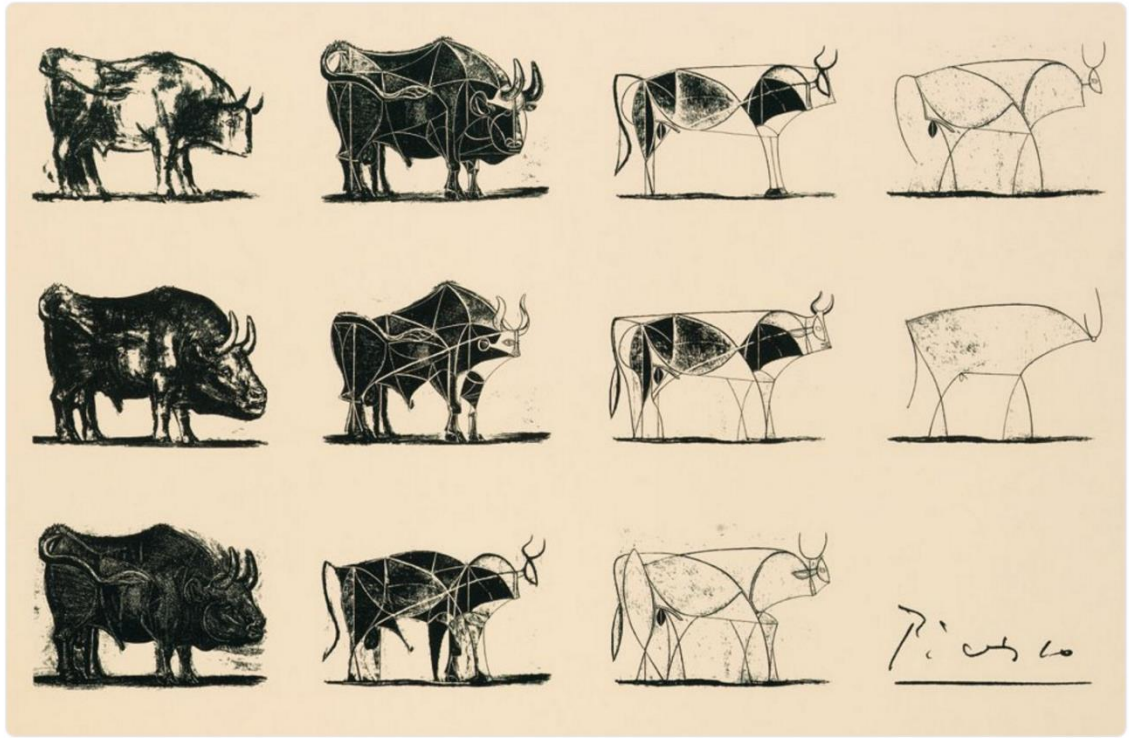
D'après le livre https://leanpub.com/4rulesofsimpledesign

Isabelle BLASQUEZ

La simplicité n'est pas *simple (facile)* à mettre en oeuvre

Simplicity is the result of working incredibly hard at planning and refinement.

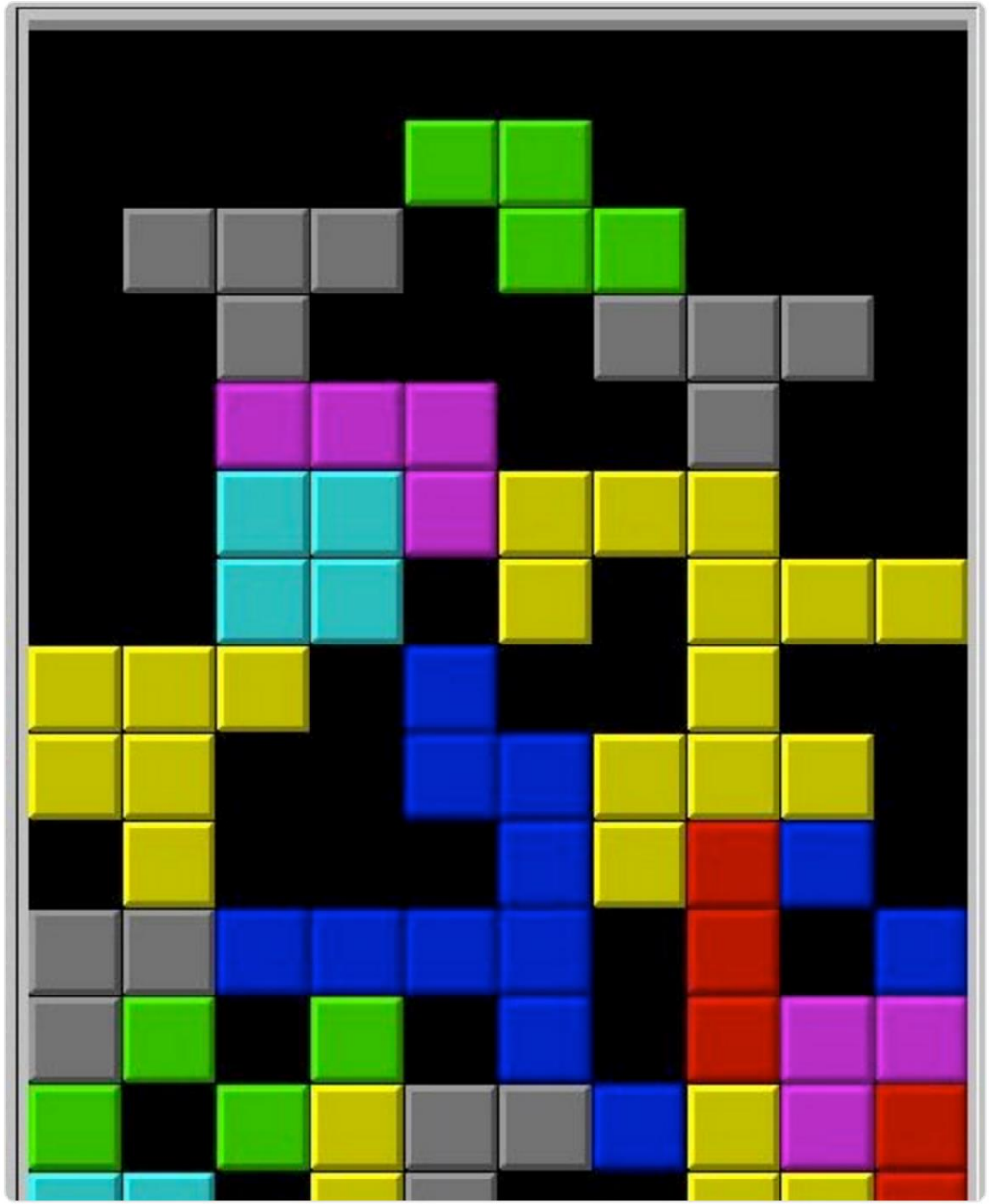
À l'origine en anglais



Extrait : https://twitter.com/hardmaru/status/842431377605713921

… et contribuer à la réduction de la dette technique

"What is technical debt?" Technical debt is hard to explain, but a picture is worth a thousand words. #programming #softwaredevelopment



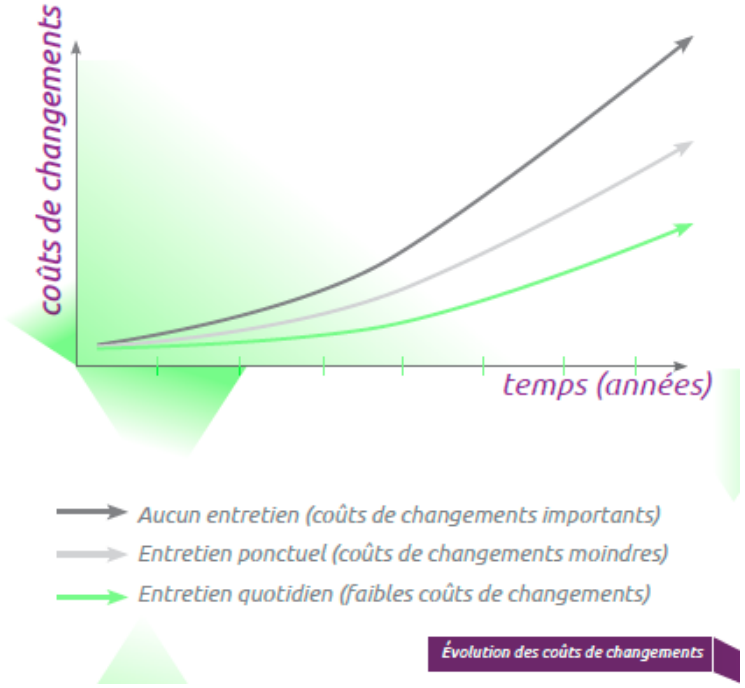
Extrait : https://twitter.com/Jedd_Ahyoung/status/826551935822077952

La dette technique

La dette technique est une *métaphore du développement logiciel* inventée par ***Ward Cunningham***.

La définition de Wikipedia est éclairante:

> Il s'inspire du concept existant de dette dans le contexte du financement des entreprises et l'applique au domaine du développement logiciel. En résumé, quand on code au plus vite et de manière non optimale, on contracte une dette technique que l'on rembourse tout au long de la vie du projet sous forme de temps de développement de plus en plus long et de bugs de plus en plus fréquents.



Sources : techtrends.xebia.fr http://blog.octo.com/maitriser-sa-dette-technique/ https://twitter.com/khellang/status/626716128379830273
A lire : http://www.occitech.fr/blog/2014/11/intro-dette-technique/ (Traduction de Technical Debt101 de Maiz Lulkin)
http://c2.com/cgi/wiki?WardExplainsDebtMetaphor et http://promyze.com/wp-content/uploads/2016/06/LaDetteTechnique.pdf (la dette technique expliquée !)
A visionner : https://www.youtube.com/watch?v=d3XYbUikeIA

Sans refactoring, attention la dette technique vous guette

Just tried to explain technical debt to a customer, had to pull this out again...

Voir la traduction

Les étapes de **refactoring** permettent au développeur de travailler sur un code propre (**clean code**) et de minimiser la dette technique

Zoom sur
la complexité cyclomatique

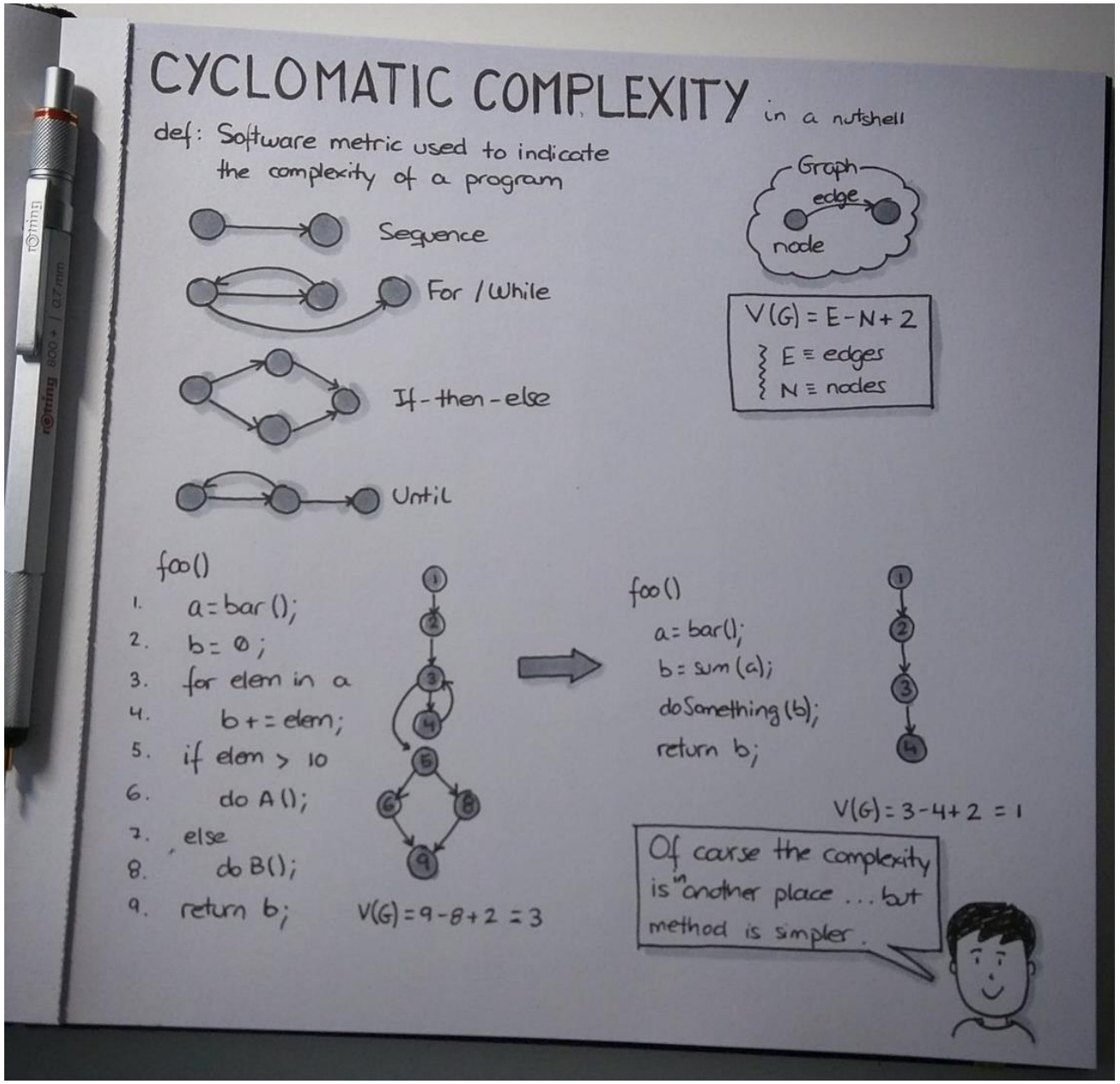
Isabelle BLASQUEZ

Complexité cyclomatique

Décompte des embranchements dans le code

```
(if  for  while  case  catch
throw  return  ||  ?)
```

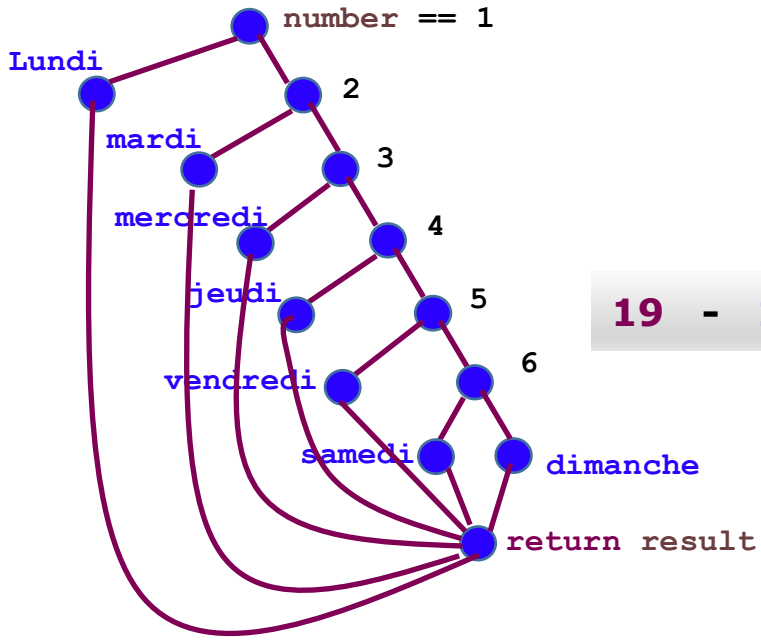
La complexité cyclomatique peut révéler une classe complexe à maintenir



Extrait : https://twitter.com/ydarias/status/841713103079383041/photo/1

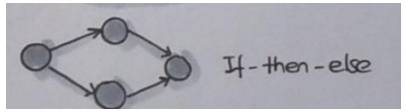
Complexité cyclomatique (1/3)

```
public class Date
{
    public static String getDay (int number){
        String result;
        if (number == 1) result = "Lundi";
        else if (number == 2) result = "Mardi";
        else if (number == 3) result = "Mercredi";
        else if (number == 4) result = "Jeudi";
        else if (number == 5) result = "Vendredi";
        else if (number == 6) result = "Samedi";
        else result = "Dimanche";
       return result;
    }
}
```



19 - 14 + 2 = **7**

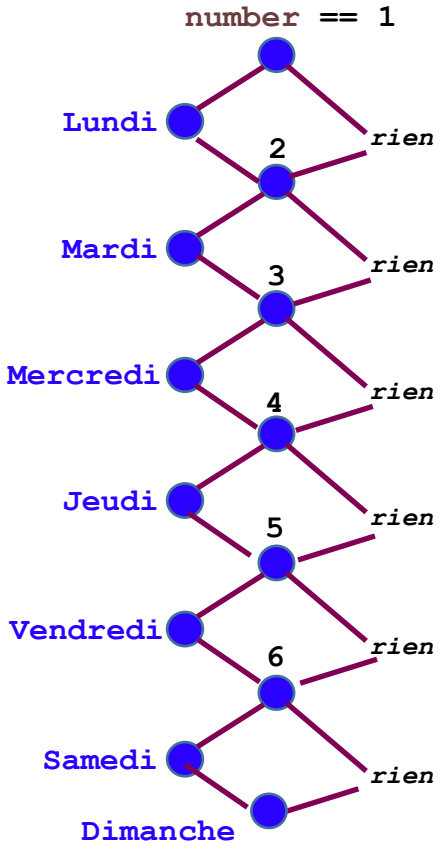
Complexité




```
public class Date
{
     public static String getDay (int number){

        if (number == 1) return "Lundi";
        if (number == 2) return "Mardi";
        if (number == 3) return "Mercredi";
        if (number == 4) return "Jeudi";
        if (number == 5) return "Vendredi";
        if (number == 6) return "Samedi";
        return "Dimanche";
    }

}
```



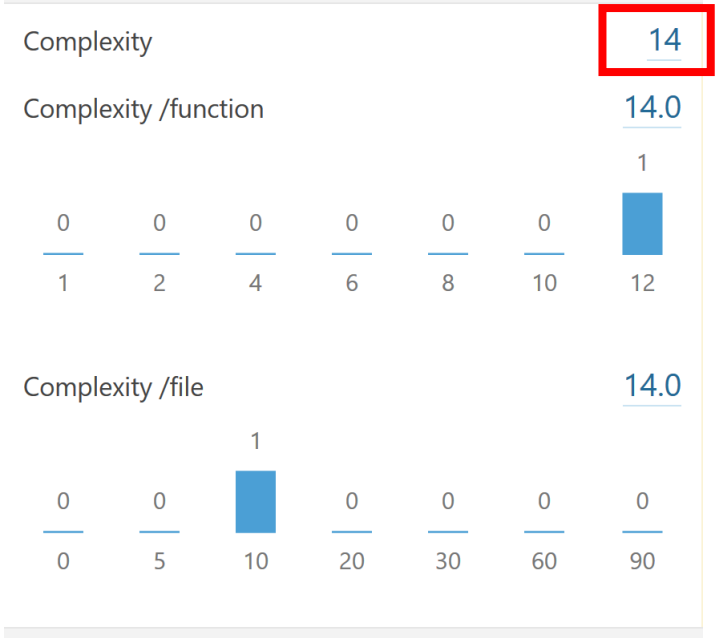
24 - 13 + 2 = **13**

Complexité

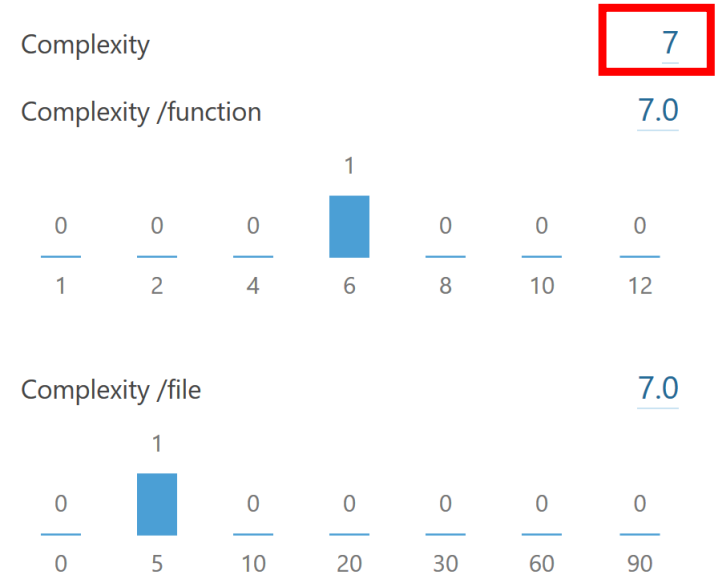
Complexité cyclomatique (2/3)

```
public class Date
{

 public static String getDay (int number){
    switch (number) {
    case 1 : return "Lundi";
    case 2 : return "Mardi";
    case 3 : return "Mercredi";
    case 4 : return "Jeudi";
    case 5 : return "Vendredi";
    case 6 : return "Samedi";
    default : return "Dimanche";
     }
  }
}
```



```
public class Date
{
   public static String getDay (int number){
      String result;
      switch (number) {
          case 1 : result="Lundi"; break;
          case 2 : result="Mardi"; break;
          case 3 : result="Mercredi"; break;
          case 4 : result="Jeudi"; break;
          case 5 : result="Vendredi"; break;
          case 6 : result="Samedi";break;
          default : result= "Dimanche";break;
          }
      return result;
    }
}
```



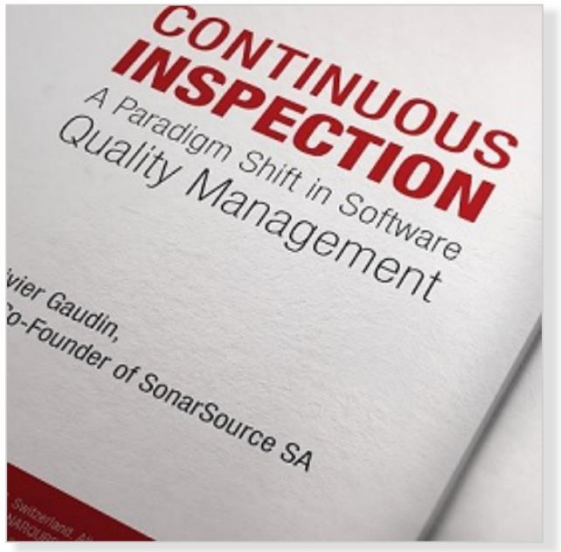
Complexité cyclomatique (3/3)

```
public class Date
{
    private static final String tabJours[] =
{"Lundi","Mardi","Mercredi","Jeudi","Vendredi","Samedi","Dimanche"};

    public static String getDay (int number){
    return tabJours[number-1];
     }
}
```

Introspection

Inspection continue pour tester la qualité de code ...



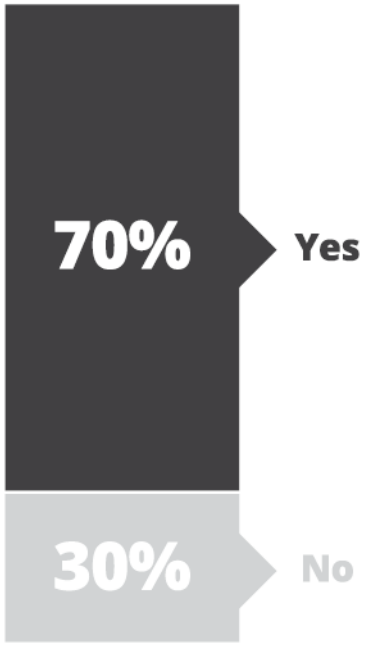
Continuous Inspection is a holistic, fully-realized process designed to make internal code quality an integral part of the software development life cycle. By raising its visibility for all stakeholders throughout the life cycle, Continuous Inspection enables enterprises to embrace code quality whole-heartedly.

Extraits : http://www.sonarsource.com/resources/continuous-inspection-white-paper/
http://www.sonarsource.com/resources/media-kit/

Inspection continue

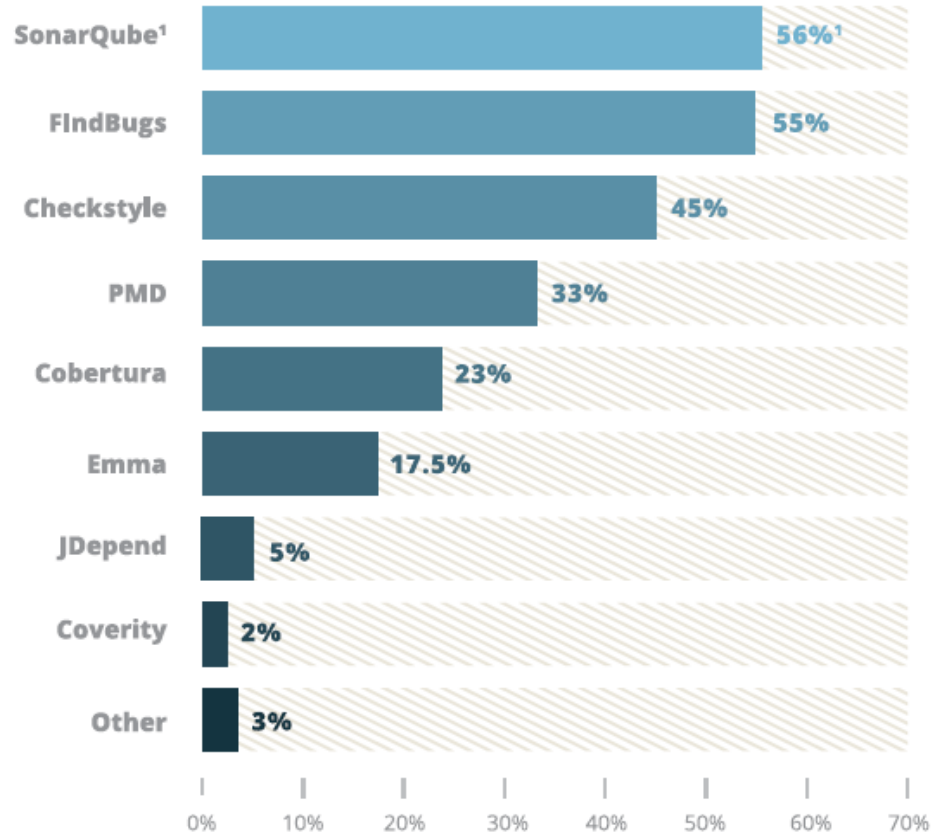
CODE METRICS

Do you use one or more code analysis tools?



REBELLABS

Code analysis tools in use *



* *Multiple selections were possible and the results were normalized to exclude non-users*
¹ *SonarQube is an integration platform for hosting analysis tools*

La mission de Sonar

Permettre de déclarer ouverte la **chasse** aux **7 péchés capitaux**

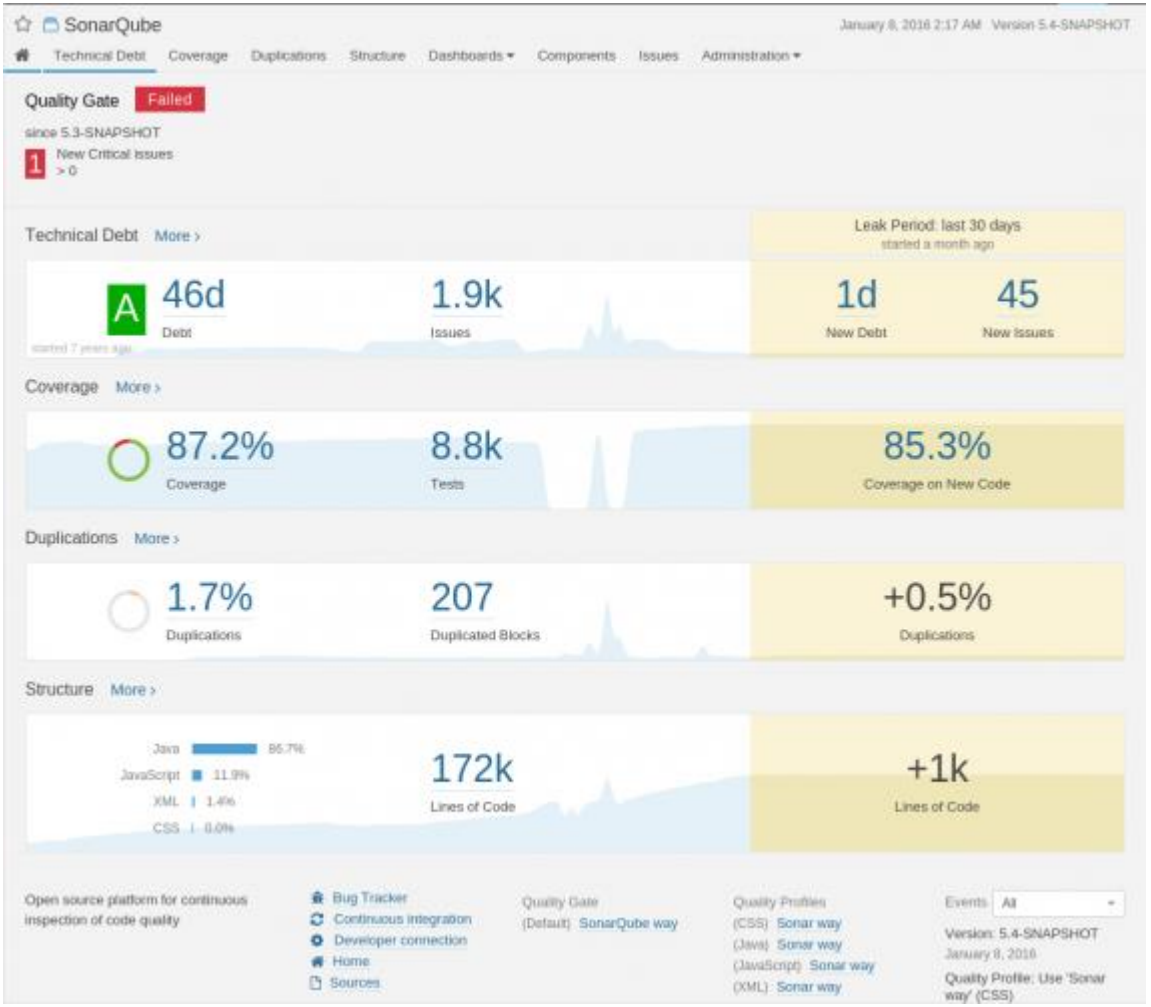
Les 7 péchés capitaux

Appliqués au code source

- Duplications
- Mauvaise distribution de la complexité
- Mauvais Design
- Pas de tests unitaires
- Pas de respect des standards
- Bugs potentiels
- Pas ou trop de commentaires

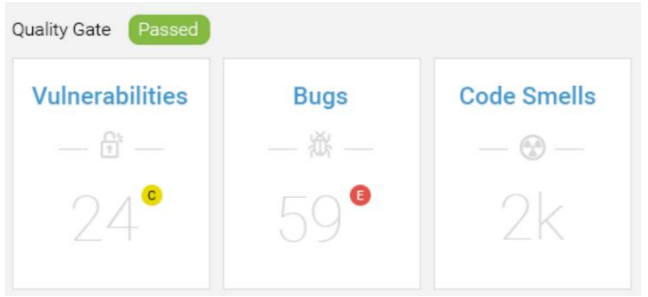
Extrait du Rapport rebelLabs 2014 http://pages.zeroturnaround.com/Java-Tools-Technologies.html?%20Technologies%20**2014**
http://fr.slideshare.net/ElsassJUG/soire-qualit-logicielle-avec-sonar **Vidéo à voir** : https://www.youtube.com/watch?v=d3XYhUikeIA

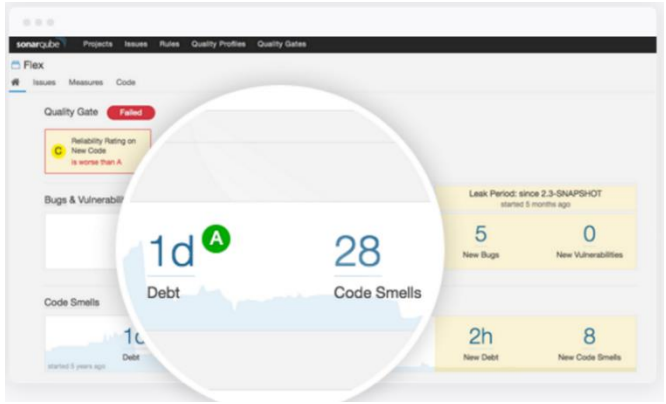
sonarqube (https://www.sonarqube.org)

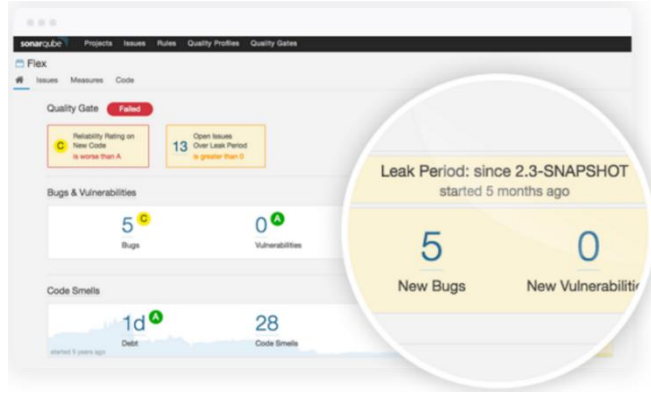


Dashboard

extrait de
https://www.supinfo.com/articles/single/2976-testez-qualite-votre-code-avec-sonarqube







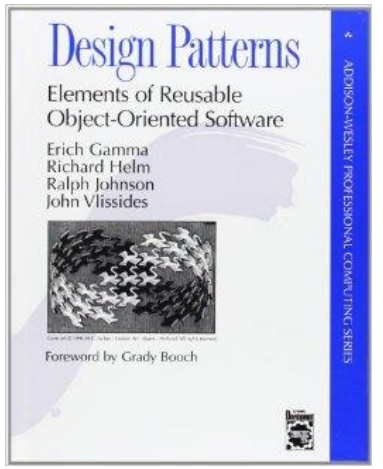

```
1   Try
2   {
3       md = MessageDigest.getInstance("MD5");
4       md.update(b);
5   } catch (NoSuchAlgorithmException e)
6   {
7       // it's got to be there
8       e.printStackTrace();
9   }
10  return (base64Encode(md.digest()));
11
12  Dereference of null pointer when NoSuchAlgorithmException is thrown
13
14
```

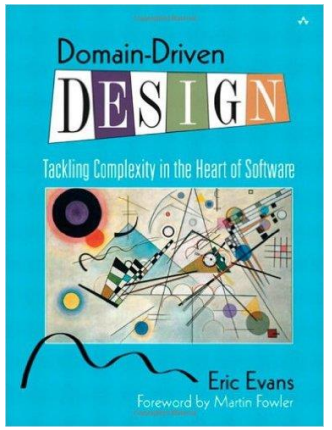
Zoom sur les éléments du dashboard et les issues

extraits de
https://www.sonarqube.org/

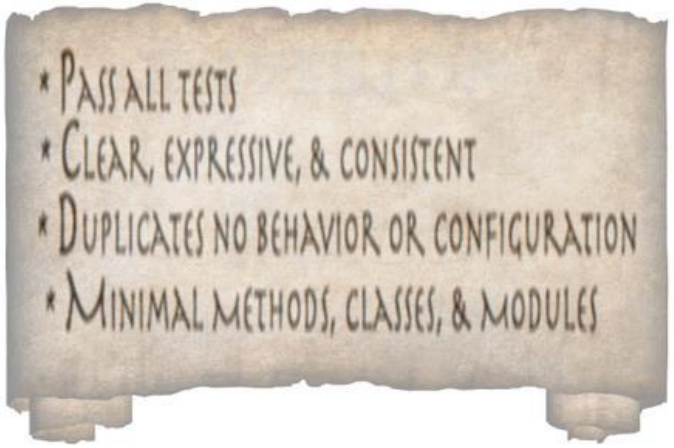
Clean Code : Bonnes Pratiques

GRASP
General **R**esponsibility **A**ssignment
Software **P**atterns

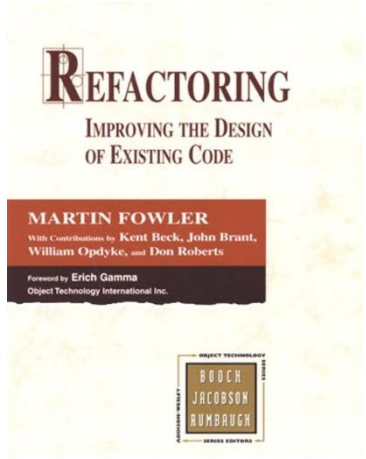




DRY (**D**on't **R**epeat **Y**ourself)



Loi de Demeter
(Tell don't ask)



… et d'autres …

Zoom sur la Loi de Demeter (Tell, don't ask)

Loi de Demeter (ou principe de connaissance minimale)
consiste à ***Ne parlez qu'à ses amis immédiats***.

J'ai un objet **Auto** et je veux savoir si l'auto est en marche.

1. **Ne** respecte **pas** la loi :

```
auto.getMotor().getState().isRunning();
```

2. Code qui **respecte** la loi :

```
auto.isRunning();
```

Pourquoi ? Si l'état `Running` n'est plus au niveau du moteur
mais plutôt au niveau des roues
(1) : instruction à modifier: `auto.getTire().getState().isRunning()`
(2) reste inchangé !

Donnez à vos collaborateurs exactement ce qu'ils demandent
et ne leur fournissez pas quelque chose qu'ils vont devoir inspecter
à la recherche de ce dont ils ont besoin.

Extrait les principes SOLID dans la vie de tous les jours : http://www.arolla.fr/blog/2017/02/principes-solid-vie-de-jours/
et Connaissez-vous la loi de Demeter? : http://savoiragile.com/2012/05/04/connaissez-vous-la-loi-de-demeter
En savoir plus sur la loi de Demeter : http://wiki.c2.com/?LawOfDemeter et https://fr.wikipedia.org/wiki/Loi_de_D%C3%A9m%C3%A9ter

Isabelle BLASQUEZ

Conclusion

Zoom sur différentes manières d'aborder la Conception …

- ✓ **Conception dirigée par les modèles** (Model Driven)

Développement classique

- ✓ **Conception dirigée par les tests** (TDD)
- ✓ **Conception dirigée par le domaine** (DDD)

Développement agile

Et pour finir …

Qu'est ce qu'une bonne conception?

En premier lieu, une bonne conception permet de développer un logiciel qui répond aux fonctionnalités demandées. Idéalement, elle permet l'évolution et la maintenance du logiciel. Les évolutions ne doivent pas remettre en question toute la conception et l'implémentation. Une bonne conception est un ensemble de considérations comme par exemple la flexibilité, la robustesse ou la maintenabilité suite à des évolutions et des corrections.

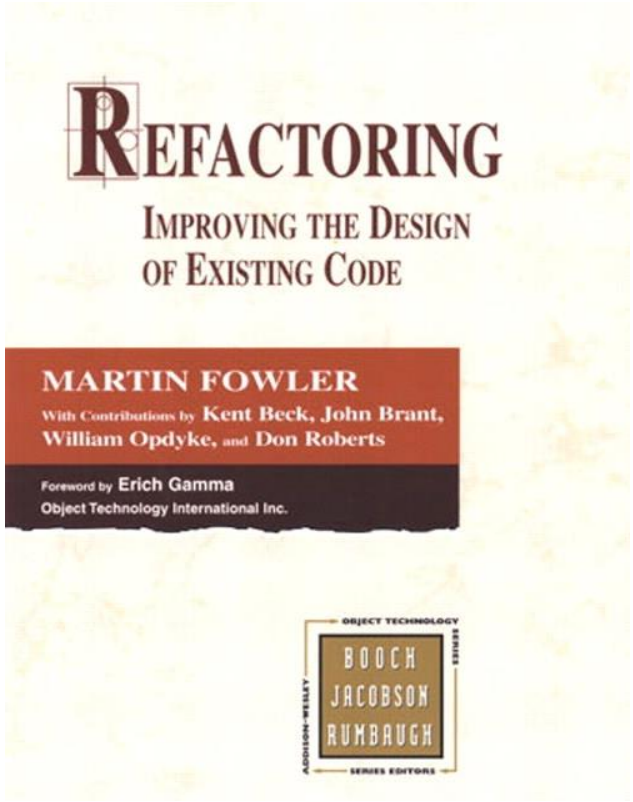
Pour tendre vers une bonne conception, une stratégie est de découper l'application en modules en suivant la règle « diviser pour mieux régner ». Ces modules suivent des règles à respecter :

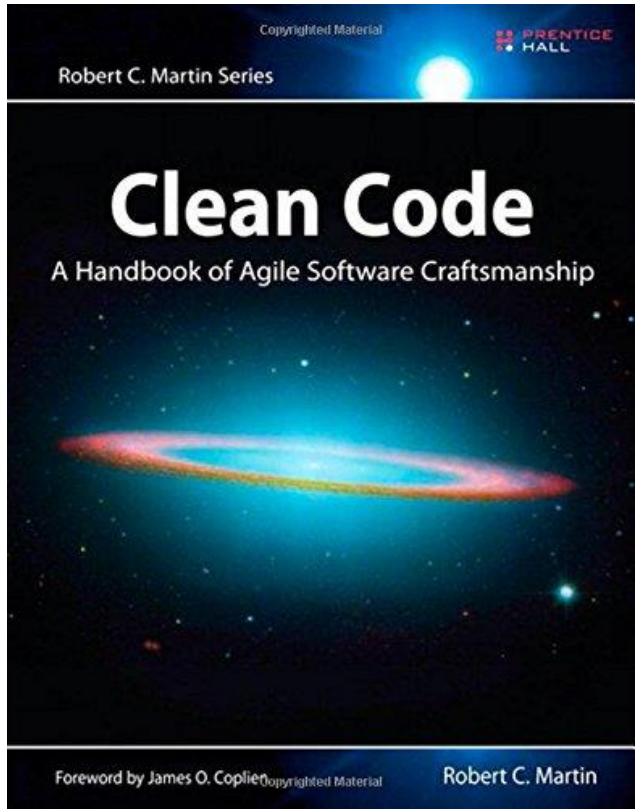
- Responsabilités identifiées dans des modules nommés
- Collaboration et interfaces entre les modules (Pour l'amélioration de la qualité, gérez vos dependances)
- Cohésion forte entre les modules
- Couplage faible entre les modules

Il existe de nombreux travaux et ouvrages sur la conception de personnes reconnues comme expert dans le domaine comme Bertand Meyer, fondateur du langage objet Eiffel, Robert C. Martin, ou encore Martin Fowler. Leurs travaux sont résumés dans le document Principes avancés de conception objet.

Extraits de :
https://www.slideshare.net/ThomasPierrain/the-art-of-software-design
http://blog.xebia.fr/2009/01/28/ddd-la-conception-qui-lie-le-fonctionnel-et-le-code/
https://www.slideshare.net/ThomasPierrain/ddd-reboot-english-version

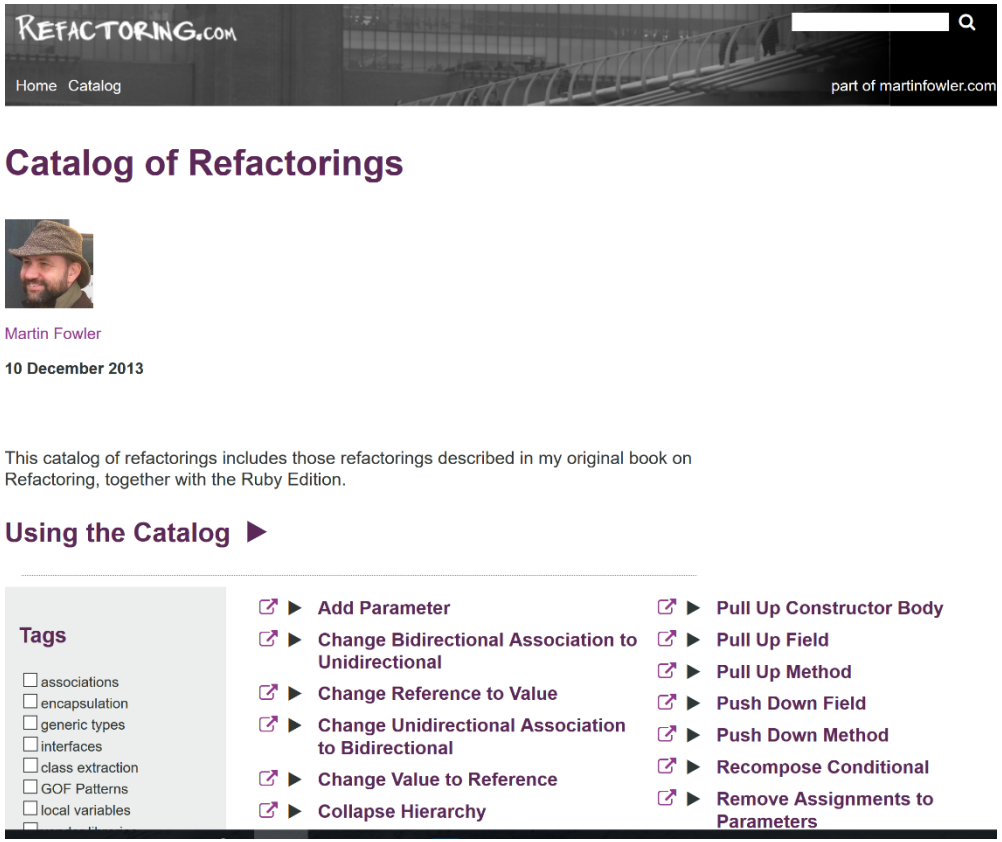
Annexe



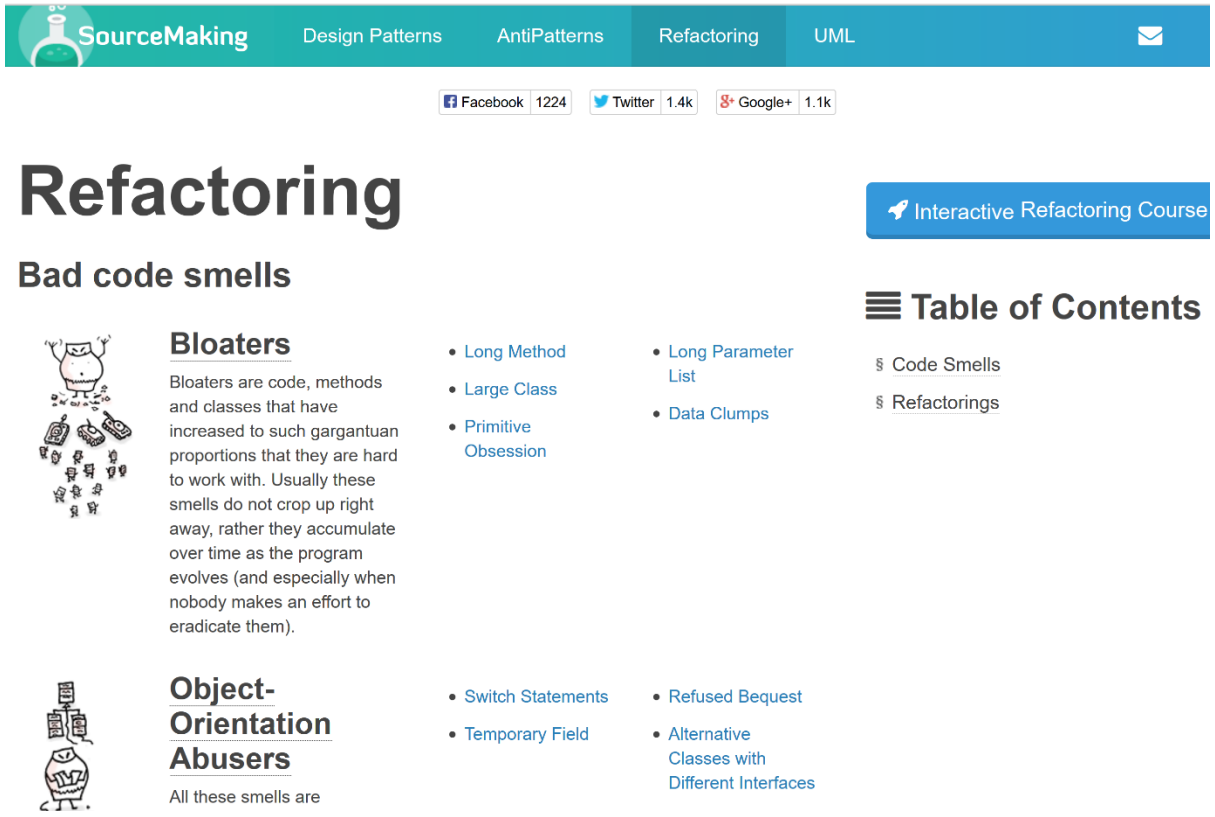


D’autres liens sur :
https://github.com/iblasquez/Refactoring_PremierExempleFowler/blob/master/references.md

Catalogues de refactoring en ligne …



http://refactoring.com/catalog/



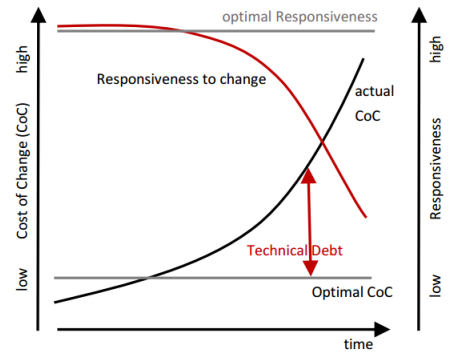
https://sourcemaking.com/

Code Cheat Sheet

bbv Software Services

Why Clean Code

Code is clean if it can be understood easily – by everyone on the team. With understandability comes readability, changeability, extensibility and maintainability. All the things needed to keep a project going over a long time without accumulating up a large amount of technical debt.



Writing clean code from the start in a project is an investment in keeping the cost of change as constant as possible throughout the lifecycle of a software product. Therefore, the initial cost of change is a bit higher when writing clean code (grey line) than quick and dirty programming (black line), but is paid back quite soon. Especially if you keep in mind that most of the cost has to be paid during maintenance of the software. Unclean code results in technical debt that increases over time if not refactored into clean code. There are other reasons leading to Technical Debt such as bad processes and lack of documentation, but unclean code is a major driver. As a result, your ability to respond to changes is reduced (red line).

In Clean Code, Bugs Cannot Hide

Most software defects are introduced when changing existing code. The reason behind this is that the developer changing the code cannot fully grasp the effects of the changes made. Clean code minimises the risk of introducing defects by making the code as easy to understand as possible.

Principles

Loose Coupling +

Two classes, components or modules are coupled when at least one of them uses the other. The less these items know about each other, the looser they are coupled.

A component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

High Cohesion +

Cohesion is the degree to which elements of a whole belong together. Methods and fields in a single class and classes of a component should have high cohesion. High cohesion in classes and components results in simpler, more easily understandable code structure and design.

Change is Local +

When a software system has to be maintained, extended and changed for a long time, keeping change local reduces involved costs and risks. Keeping change local means that there are boundaries in the design which changes do not cross.

It is Easy to Remove +

We normally build software by adding, extending or changing features. However, removing elements is important so that the overall design can be kept as simple as possible. When a block gets too complicated, it has to be

Smells

Rigidity –

The software is difficult to change. A small change causes a cascade of subsequent changes.

Fragility –

The software breaks in many places due to a single change.

Immobility –

You cannot reuse parts of the code in other projects because of involved risks and high effort.

Viscosity of Design –

Taking a shortcut and introducing technical debt requires less effort than doing it right.

Viscosity of Environment –

Building, testing and other tasks take a long time. Therefore, these activities are not executed properly by everyone and technical debt is introduced.

Needless Complexity –

The design contains elements that are currently not useful. The added complexity makes the code harder to comprehend. Therefore, extending and changing the code results in higher effort than necessary.

Needless Repetition –

Code contains lots of code duplication: exact code duplications or design duplicates (doing the same thing in a different way). Making a change to a duplicated piece of code is more expensive and more error-prone because the change has to be made in several places with the risk that one place is not changed accordingly.

Opacity –

The code is hard to understand. Therefore, any change takes additional time to first reengineer the code and is more likely to result in defects due to not understanding the side effects.

Class Design

Single Responsibility Principle (SRP) +

A class should have one, and only one, reason to change.

Open Closed Principle (OCP) +

You should be able to extend a classes behaviour without modifying it.

Liskov Substitution Principle (LSP) +

Derived classes must be substitutable for their base classes.

Dependency Inversion Principle (DIP) +

Depend on abstractions, not on concretions.

Interface Segregation Principle (ISP) +

Make fine grained interfaces that are client-specific.

Classes Should be Small +

Smaller classes are easier to grasp. Classes should be smaller than about 100 lines of code. Otherwise, it is hard to spot how the class does its job and it probably does more than a single job.

Package Cohesion

Release Reuse Equivalency Principle (RREP) +

The granule of reuse is the granule of release.

Common Closure Principle (CCP) +

Classes that change together are packaged together.

Common Reuse Principle (CRP) +

Classes that are used together are packaged together.

Package Coupling

General

Follow Standard Conventions +

Coding-, architecture-, design guidelines (check them with tools)

Keep it Simple, Stupid (KISS) +

Simpler is always better. Reduce complexity as much as possible.

Boy Scout Rule +

Leave the campground cleaner than you found it.

Root Cause Analysis +

Always look for the root cause of a problem. Otherwise, it will get you again and again.

Multiple Languages in One Source File –

C#, Java, JavaScript, XML, HTML, XAML, English, German ...

Environment

Project Build Requires Only One Step +

Check out and then build with a single command.

Executing Tests Requires Only One Step +

Run all unit tests with a single command.

Source Control System +

Always use a source control system.

Continuous Integration +

Assure integrity with Continuous Integration

Overridden Safeties –

Do not override warnings, errors, exception handling – they will catch you.

Dependency Injection

Decouple Construction from Runtime +

Decoupling the construction phase completely from the runtime helps to simplify the runtime behaviour.

Design

Keep Configurable Data at High Levels +

If you have a constant such as default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to the low-level function called from the high-level function.

Don't Be Arbitrary +

Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it.

Be Precise +

When you make a decision in your code, make sure you make it precisely. Know why you have made it and how you will deal with any exceptions.

Structure over Convention +

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance.

Prefer Polymorphism To If/Else or Switch/Case +

"ONE SWITCH": There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.

Symmetry / Analogy +

Favour symmetric designs (e.g. Load – Save) and designs that follow analogies (e.g. same design as found in .NET framework).

Fields Not Defining State –

Fields holding data that does not belong to the state of the instance but are used to hold temporary data. Use local variables or extract to a class abstracting the performed action.

Over Configurability –

Prevent configuration just for the sake of it – or because nobody can decide how it should be. Otherwise, this will result in overly complex, unstable systems.

Micro Layers –

Do not add functionality on top, but simplify overall.

Dependencies

Make Logical Dependencies Physical +

If one module depends upon another, that dependency should be physical, not just logical. Don't make assumptions.

Singletons / Service Locator –

Use dependency injection. Singletons hide dependencies.

Base Classes Depending On Their Derivatives –

Base classes should work with any derived class.

Too Much Information –

Minimise interface to minimise coupling

Feature Envy –

The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it envies the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating.

Artificial Coupling –

Things that don't depend upon each other should not be artificially coupled.

Hidden Temporal Coupling –

If, for example, the order of some method calls is important, then make sure that they cannot be called in the wrong order.

Transitive Navigation –

Aka Law of Demeter, writing shy code.
A module should know only its direct dependencies.

Naming

Choose Descriptive / Unambiguous Names +

Names have to reflect what a variable, field, property stands for. Names have to be precise.

Choose Names at Appropriate Level of Abstraction +

Choose names that reflect the level of abstraction of the class or method you are working in.

Name Interfaces After Functionality They Abstract +

The name of an interface should be derived from its usage by the client, such as IStream.

Name Classes After How They Implement Their Interfaces +

The name of a class should reflect how it fulfils the functionality provided by its interface(s), such as MemoryStream : IStream

Name Methods After What They Do +

The name of a method should describe what is done, not how it is done.

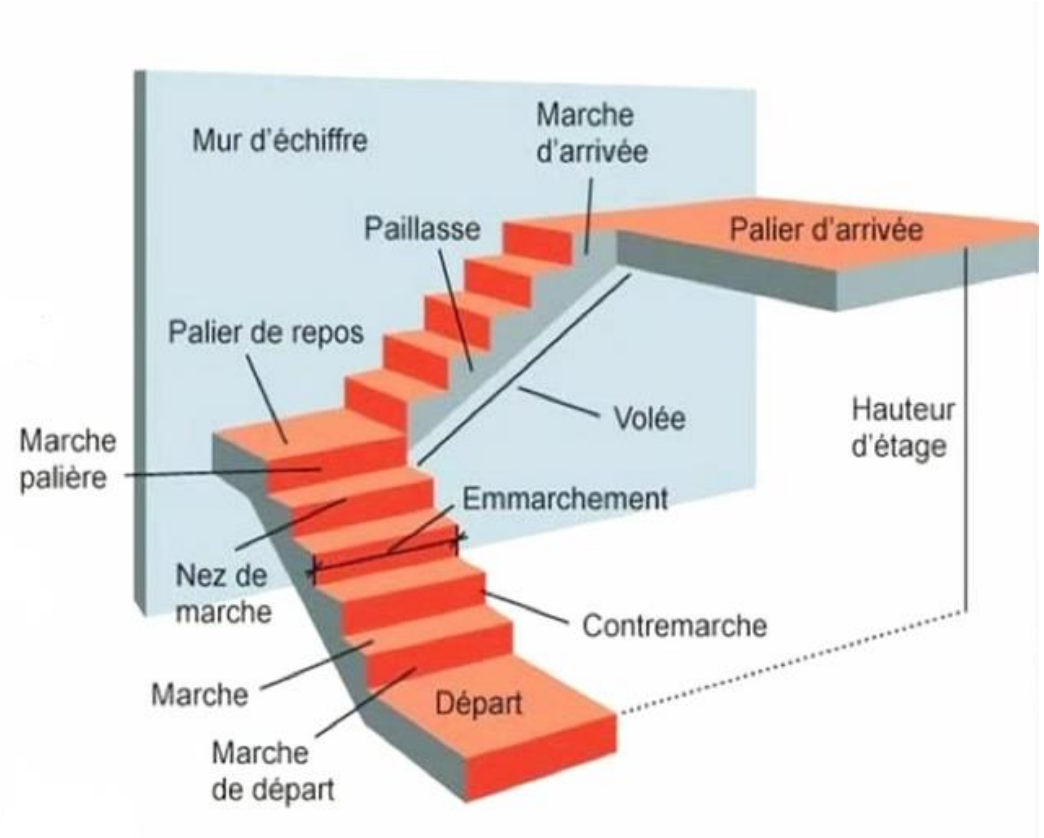
Use Long Names for Long Scopes +

fields → parameters → locals → loop variables

http://www.planetgeek.ch/wp-content/uploads/2013/06/Clean-Code-V2.1.pdf

Isabelle BLASQUEZ

L'ubiquitous langage (langage omniprésent), un premier pas vers le DDD …



Interaction et discussions avec les experts métiers pour utiliser le bon terme …

Extrait de la video : https://www.youtube.com/watch?v=h3DLKrvp5V8
Dont un retour est disponible ici : https://blog.groupe-sii.com/devoxx-france-2016-architecture-methodologie/#ddd
Première approche DDD: http://blog.infosaurus.fr/public/docs/DDDViteFait.pdf