MGL7460 — Automne 2016

Maintenance de logiciels et qualité du code

Guy Tremblay
Professeur

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

3 novembre 2016

Contenu

1 Introduction : Qu'est-ce que la maintenance ?

2 *Clean Code vs. Code Smells*

3 Qualités des produits logiciels

4 Maintenabilité

5 Lignes directrices du SIG pour la maintenabilité

6 Outils d'évaluation de la qualité du code

7 Dette technique et *Refactoring*

8 Code légataire

1. Introduction : Qu’est-ce que la maintenance ?

Qu’est-ce que la maintenance ?

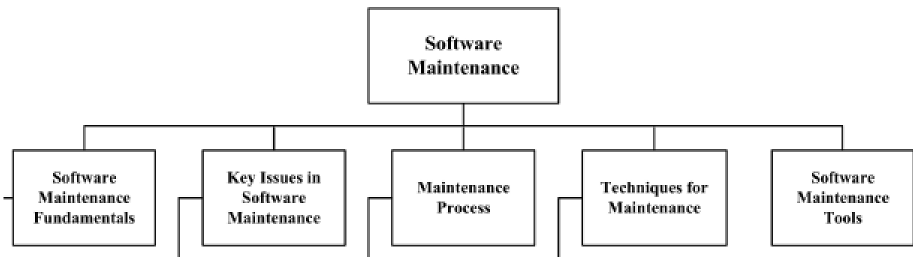
Définition de la maintenance selon l'IEEE

Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Source: «*IEEE Standard Glossary of Software Engineering Terminology*»

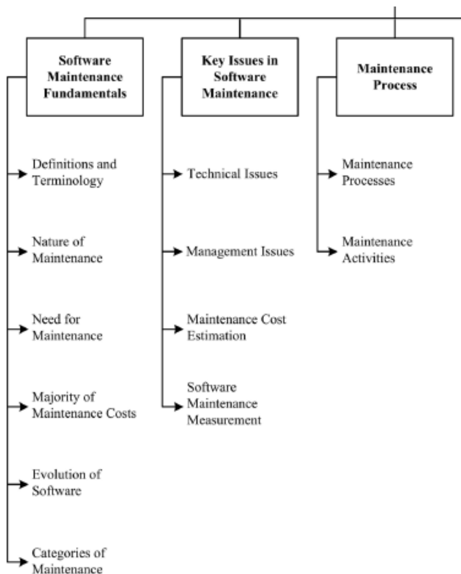
Guide to the SWEBOK

Breakdown of Topics for the Software Maintenance KA



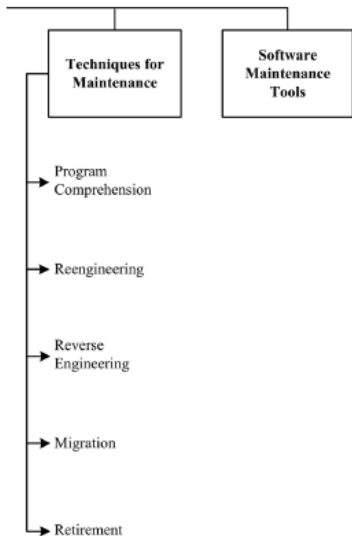
Guide to the SWEBOK

Breakdown of Topics for the Software Maintenance KA



Guide to the SWEBOK

Breakdown of Topics for the Software Maintenance KA



Définition de la maintenance selon l'IEEE

Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Source: «*IEEE Standard Glossary of Software Engineering Terminology*»

Définition de la maintenance selon l'IEEE

Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Source: «*IEEE Standard Glossary of Software Engineering Terminology*»

Définition de la maintenance selon l'IEEE

Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Source: «*IEEE Standard Glossary of Software Engineering Terminology*»

Définition de la maintenance selon l'IEEE

Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Source: «*IEEE Standard Glossary of Software Engineering Terminology*»

Donc, il y a diverses raisons pour lesquelles on doit faire de la maintenance

Pour...

- corriger des erreurs ;
- faire des améliorations pour mieux supporter les besoins des usagers ;
- adapter le logiciel suite à des changements dans l'environnement d'affaires, dans l'environnement technologique (par ex., nouvelles bases de données) ;
- améliorer la conception, le code, etc. ;

Les quatre catégories de maintenance selon l'IEEE

Table 5.1. Software Maintenance Categories		
	Correction	**Enhancement**
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

Source: «Guide to the SWEBOK»

Les quatre catégories de maintenance selon l'IEEE

Maintenance préventive

Pour prévenir la détérioration du logiciel face aux changements, pour améliorer sa «maintainabilité» future.

Maintenance perfective

Pour ajouter ou modifier les fonctionnalités d'un système existant (les usagers découvrent de nouveaux besoins).

Maintenance corrective

Pour réparer des défauts (erreurs conception, codage, logique, etc.).

Maintenance adaptative

Pour adapter le logiciel à un environnement changeant (règles d'affaires, lois, procédures de travail, plates-formes).

Est-ce que la maintenance est vraiment uniquement «après» la livraison du logiciel ?

Une vision plus «pragmatique» — plus agile — de la maintenance

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong.

Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes.

Whatever the reason, maintenance is not a discrete activity but a routine part of the entire development process.

A. Hunt & D. Thomas, «The Pragmatic Programmer»

Une vision plus «pragmatique» — plus agile — de la maintenance

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong.

Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes.

Whatever the reason, maintenance is not a discrete activity but a routine part of the entire development process.

A. Hunt & D. Thomas, «The Pragmatic Programmer»

Une vision plus «pragmatique» — plus agile — de la maintenance

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong.

Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes.

Whatever the reason, maintenance is not a discrete activity but a routine part of the entire development process.

A. Hunt & D. Thomas, «The Pragmatic Programmer»

Coûts de la maintenance

Selon une étude des années ’90 ayant analysé un grand nombre de projets :

Total des dépenses en logiciel	100 G $
Maintenance de systèmes existants	70 G $
Développement de nouveaux systèmes	30 G $

Note : 1 G $ = 1,000,000,000 $ (1 Milliard)

Coûts de la maintenance

De nombreuses études ont été faites sur les coûts de la maintenance par rapport à l'ensemble des coûts.

Conclusion : la maintenance accapare, selon les études, les pourcentages suivants des coûts totaux :

60 %
40–80 %
60–70 %
75 %
65 %
60–80 %
…

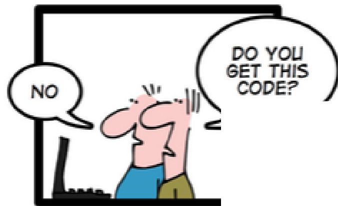
Coûts de la maintenance

Répartition des différents types de maintenance :

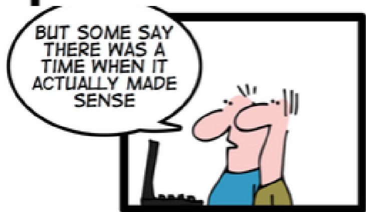
Corrective	20 %
Adaptative	25 %
Perfective	55 %

Donc :
$\approx$ 80 % des coûts sont liés à des améliorations — et non à des corrections !

Pourquoi les coûts de maintenance sont-ils si importants ?



geek & poke



ENTROPY

Les deux premières «lois» de Lehman sur la dynamique d'évolution des systèmes

Changement continu

Un programme qui est utilisé dans un environnement réel devra nécessairement s'adapter et changer, sous peine de devenir progressivement moins utile dans cet environnement.

Les deux premières «lois» de Lehman sur la dynamique d'évolution des systèmes

Changement continu

Un programme qui est utilisé dans un environnement réel devra nécessairement s'adapter et changer, sous peine de devenir progressivement moins utile dans cet environnement.

Complexité croissante

Au fur et à mesure qu'un programme évolue, sa structure tend à devenir plus complexe. Des ressources additionnelles doivent être consacrées à la préservation et à la simplification de sa structure.

Source: `https://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution`

La loi d’entropie semble donc aussi s’appliquer aux logiciels

The second law of thermodynamics, in principle, states that a closed system’s disorder cannot be reduced, it can only remain unchanged or increased. A measure of this disorder is ***entropy****.*

La loi d'entropie semble donc aussi s'appliquer aux logiciels

The second law of thermodynamics, in principle, states that a closed system's disorder cannot be reduced, it can only remain unchanged or increased. A measure of this disorder is **entropy***.*

This law also seems plausible for software systems ; as a system is modified, its disorder, or entropy, always increases. This is known as ***software entropy****.*

Source*: «Object-Oriented Software Engineering : A Use Case Driven Approach», Jacobson, Christerson, Jonsson & Övergaard*

La loi d’entropie semble donc aussi s’appliquer aux logiciels

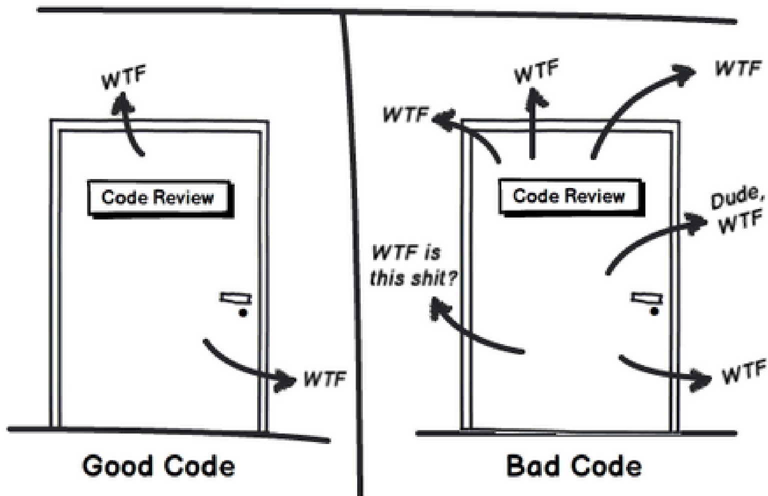
Software Rot

Bad things happen to good code. No matter how well you start, no matter how honorable your intentions, no matter how pure your design and how clean the first release’s implementation, time will warp and twist your masterpiece. Never underestimate the ability of code to acquire warts and blemishes during its life.

Source: «Code Craft—The Practice of Writing Excellent Code», P. Goodliffe, 2007.

Quels sont les signes de la «pourriture du logiciel» ?

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



http://commadot.com/wtf-per-minute/

2. *Clean Code vs. Code Smells*

Qu’est du code propre (*Clean code*) ?

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*

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 ***<u>who cares</u>***

M. Feathers

«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.*

Code smells

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.*

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.*

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 M. Fowler sur le *refactoring* de 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

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 Classe*
- *Primitive Obsession*
- *Long Parameter List*
- *Data Clumps*

Object-Orientation Abusers

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

- *Switch Statement*
- *Temporary Field*
- *Refused Bequest*
- *Alternative Classes with Different Interfaces*

Change Preventers ⋆

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

- *Divergent Change*
- *Shotgun Surgery*
- *Parallel Inheritance Hierarchies*

Dispensables

⋆

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*

Couplers

⋆

Couplers

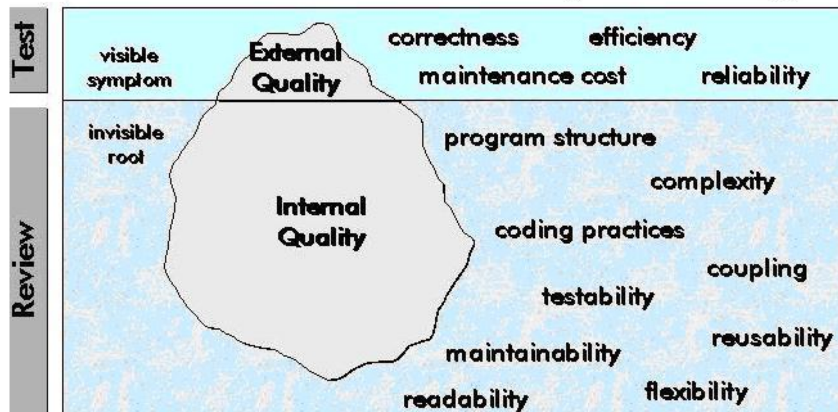
All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- *Feature Envy*
- *Inappropriate Intimacy*
- *Message Chains*
- *Middle Man*
- *Incomplete Library Class*

3. Qualités des produits logiciels

Qualités des produits logiciels

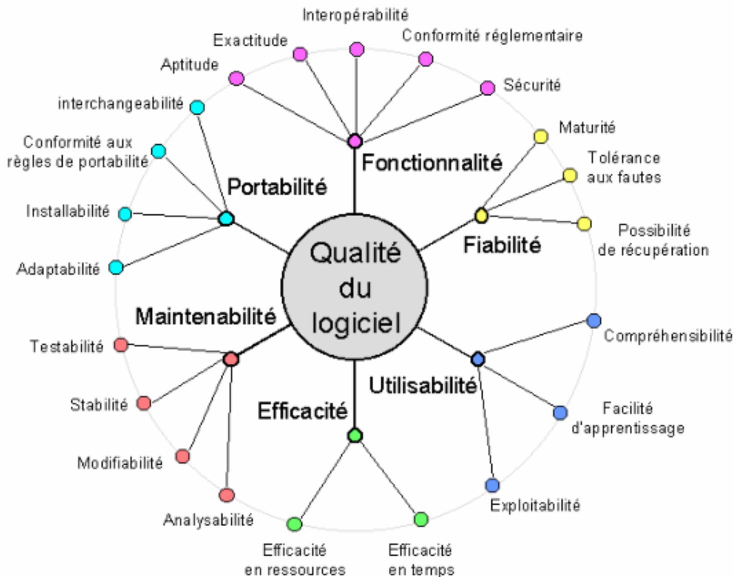
http://photos1.blogger.com/blogger/1941/1049/1600/SW%20Quality%20Iceberg.jpg



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

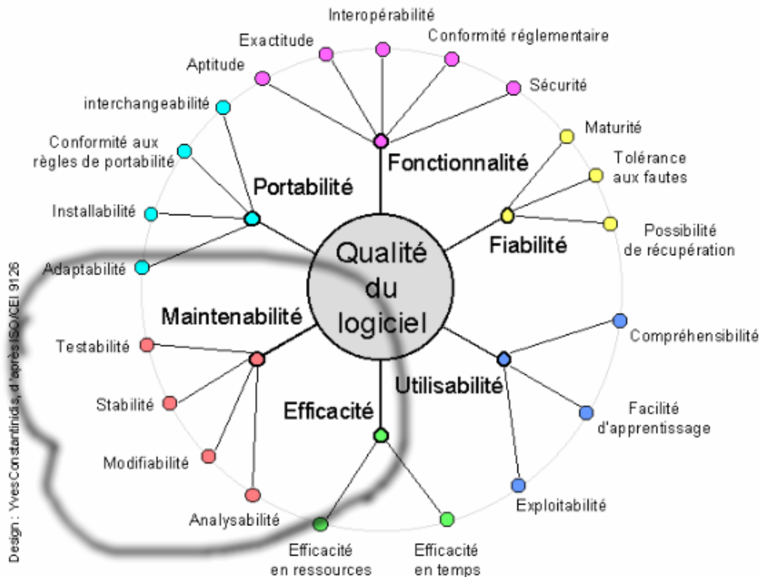
Qualités des produits logiciels

https://prezi.com/15ldp1otttpg/copy-of-iso-25000-is3/



Qualités des produits logiciels

https://prezi.com/15ldp1otttpg/copy-of-iso-25000-is3/



4. Maintenabilité

Qu’est-ce que la maintenabilité ?

The ease or difficulty with which a software system can be modified is known as its maintainability.

Source: «*Building Maintenable Software*», Visser, 2016

Maintenabilité et qualité du code

When a change is needed in the software, the quality of its source code has an impact on how easy it is :
1) to determine where and how that change can be performed ;

2) to implement that change ;

3) to avoid unexpected effects of that change and

4) to validate the changes performed.

Source: «*Standardized Code Quality Benchmarking for Improving Software Maintainability*», Baggen, Correia, Schil and Visser, Software Quality Journal, vol. 20, no. 2, 2012.

`http://link.springer.com/article/10.1007/s11219-011-9144-9`

Maintenabilité et qualité du code

When a change is needed in the software, the quality of its source code has an impact on how easy it is :
1) to determine where and how that change can be performed ;

2) to implement that change ;

3) to avoid unexpected effects of that change and

4) to validate the changes performed.

Source: «*Standardized Code Quality Benchmarking for Improving Software Maintainability*», Baggen, Correia, Schil and Visser, Software Quality Journal, vol. 20, no. 2, 2012.

`http://link.springer.com/article/10.1007/s11219-011-9144-9`

Maintenabilité et qualité du code

When a change is needed in the software, the quality of its source code has an impact on how easy it is :
1) to determine where and how that change can be performed ;

2) to implement that change ;

3) to avoid unexpected effects of that change and

4) to validate the changes performed.

Source: «*Standardized Code Quality Benchmarking for Improving Software Maintainability*», Baggen, Correia, Schil and Visser, Software Quality Journal, vol. 20, no. 2, 2012.

http://link.springer.com/article/10.1007/s11219-011-9144-9

Maintenabilité et qualité du code

When a change is needed in the software, the quality of its source code has an impact on how easy it is :

1) to determine where and how that change can be performed ;

2) to implement that change ;

3) to avoid unexpected effects of that change and

4) to validate the changes performed.

Source: «*Standardized Code Quality Benchmarking for Improving Software Maintainability*», Baggen, Correia, Schil and Visser, Software Quality Journal, vol. 20, no. 2, 2012.

`http://link.springer.com/article/10.1007/s11219-011-9144-9`

Caractéristiques de la maintenabilité selon ISO25000

https://prezi.com/15ldp1otttpg/copy-of-iso-25000-is3/

Maintenabilité

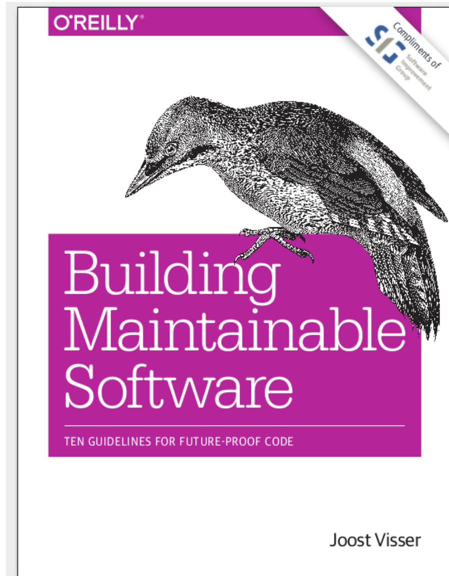
- Modularité
- Réutilisabilité
- Potentiel d'Analyse
- Potentiel de changement
- Stabilité des modifications
- Potentiel de test

5. Lignes directrices du SIG pour la maintenabilité

SIG = *Software Improvement Group*

https://www.sig.eu/en

«*SIG has designed the SIG 5-star classification system for software quality, based on the ISO/IEC 25010 model.*»



Trois principes sous-tendent les lignes directrices$^{(\star)}$ proposées

1. *Maintainability benefits most from adhering to simple guidelines.*
2. *Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts.*
3. *Some violation are worse than others. The more a software system complies with the guidelines, the more maintainable it is.*

$(\star)$ Lignes directrices = *Guidelines* = *A general rule, principle, or piece of advice.*

Trois principes sous-tendent les lignes directrices(⋆) proposées

1. *Maintainability benefits most from adhering to simple guidelines.*

2. *Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts.*

3. *Some violation are worse than others. The more a software system complies with the guidelines, the more maintainable it is.*

(⋆) Lignes directrices = *Guidelines = A general rule, principle, or piece of advice.*

Trois principes sous-tendent les lignes directrices(⋆) proposées

1. *Maintainability benefits most from adhering to simple guidelines.*

2. *Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts.*

3. *Some violation are worse than others. The more a software system complies with the guidelines, the more maintainable it is.*

(⋆) Lignes directrices = *Guidelines* = *A general rule, principle, or piece of advice.*

Trois principes sous-tendent les lignes directrices$^{(\star)}$ proposées

1. *Maintainability benefits most from adhering to simple guidelines.*

2. *Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts.*

3. *Some violation are worse than others. The more a software system complies with the guidelines, the more maintainable it is.*

($\star$) Lignes directrices = *Guidelines* = *A general rule, principle, or piece of advice.*

La maintenabilité est évaluée selon des profils de qualité

Utilise un système de notation avec 1–5 étoiles

Unit size quality profile of Jenkins

64% · 18% · 12% · 6%

1-15 · 16-30 · 31-60 · 61+

Unit size quality profile of an anonymous ★★☆☆☆ system

27% · 18% · 20% · 35%

1-15 · 16-30 · 31-60 · 61+

Unit size quality profile of an anonymous ★★★★☆ system

81% · 18% · 1%

1-15 · 16-30 · 30-60 · 61+

Figure 1-1. Example of three quality profiles

Source: «*Building Maintainable Software*», Visser, 2016

La maintenabilité est évaluée selon des profils de qualité

Donc, ce n'est pas une question de «tout ou rien», 0 % ou 100 %

To keep the metrics simple but also practical, we determine the quality of a complete codebase not by the code's number of violations but by its quality profiles.

A quality profile divides metrics into distinct categories, ranging from fully compliant code to severe violations.

By using quality profiles, we can distinguish between moderate violations (for example, a method with 20 lines of code) from severe violations (for example, a method with 200 lines of code).

Source: «*Building Maintainable Software*», Visser, 2016

Les profils de qualité sont obtenus à partir d'un grand échantillon de logiciels

SIG maintains a benchmark repository holding the results of several hundreds of standard system evaluations carried out so far. [. . .] The benchmark repository holds results for over 500 evaluations (measurements computed for a language within a system) encompassing around 200 systems [with a] total of 45 different computer languages [. . .]

Source: «*Standardized Code Quality Benchmarking for Improving Software Maintainability*», Baggen, Correia, Schil and Visser, Software Quality Journal, vol. 20, no. 2, 2012.

`http://link.springer.com/article/10.1007/s11219-011-9144-9`

Dix (10) lignes directrices (*guidelines*) sont proposées pour obtenir un logiciel maintenable

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*

Liens entre certaines propriétés d'un logiciel et des caractéristiques de qualité

Ces liens sont fondés sur l'opinion d'experts du domaine

	VOLUME	DUPLICATION	UNIT SIZE	UNIT COMPLEXITY	UNIT INTERFACING	MODULE COUPLING	COMPONENT BALANCE	COMPONENT INDEPENDENCE
Analyzability	X	X	X				X	
Modifiability		X		X		X		
Testability	X			X				X
Modularity						X	X	X
Reusability			X		X			

Source: «SIG/TÜViT Evaluation Criteria Trusted Product Maintainability, Version 8.0», Visser et al., 2015

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.***

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*

Minimum thresholds for a 4-star unit size rating (2015 version of the SIG/TÜViT Evaluation Criteria)

Lines of code in methods with...	Percentage allowed for 4 stars for unit size
... more than 60 lines of code	At most 6.9 %
... more than 30 lines of code	At most 22.3 %
... more than 15 lines of code	At most 43.7 %
... at most 15 lines of code	At most 56.3 %

Source: «*Building Maintainable Software*»

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.***

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*

Complexité cyclomatique de McCabe

McCabe's cyclomatic complexity is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring the number of linearly independent paths through the program.

The higher the number the more complex the code.

The cyclomatic complexity number also indicates the number of test cases that would have to be written to execute all paths in a program.

Source: `http://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php`

Complexité cyclomatique de McCabe

McCabe's cyclomatic complexity is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring the number of linearly independent paths through the program.

The higher the number the more complex the code.

The cyclomatic complexity number also indicates the number of test cases that would have to be written to execute all paths in a program.

Source: `http://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php`

La complexité indique combien de tests sont requis pour assurer une bonne couverture du code

Couverture des instructions

L'ensemble des tests assure que toutes les instructions sont exécutées au moins une fois.

La complexité indique combien de tests sont requis pour assurer une bonne couverture du code

Couverture des instructions

L'ensemble des tests assure que toutes les instructions sont exécutées au moins une fois.

Couverture des branchements et conditions

L'ensemble des tests assure que toutes les branches de chacune des instructions conditionnelles sont exécutées au moins une fois.

Couverture des instructions vs. Couverture des branchements et conditions

Soit la fonction `foo`

```
void foo( int var, int cond )
{
  int* p = NULL;
  if( cond ) {
    p = &var;
    *p = 1;
  }
  *p = 0;
}
```

Ce test est-il suffisant même s'il couvre toutes les instructions ?

```
  foo(x, true)
  assert_equal 0, x
```

Note : Exemple inspiré de `http://www.bullseye.com/statementCoverage.html`

Couverture des instructions vs. Couverture des branchements et conditions

Soit la fonction `foo`

```
void foo( int var, int cond )
{
  int* p = NULL;
  if( cond ) {
    p = &var;
    *p = 1;
  }
  *p = 0; // Oops! Erreur si cond != true
}
```

Ce test est-il suffisant même s'il couvre toutes les instructions ?

```
  foo(x, true)
  assert_equal 0, x
```

Note : Exemple inspiré de `http://www.bullseye.com/statementCoverage.html`

Une façon simple de calculer la complexité cyclomatique

Complexité cyclomatique

Complexité cyclomatique = Nombre de conditions + 1

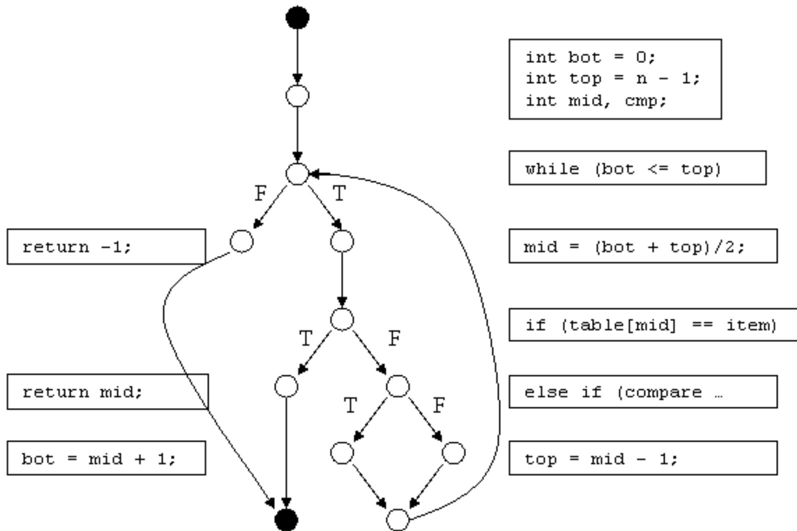
Note : Définition valable sous certaines hypothèses, généralement satisfaites dans les langages modernes — avec du code structuré, sans `goto`.

Un exemple : Fonction `BinSearch`

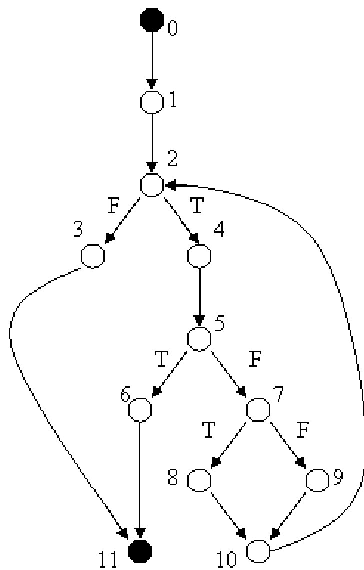
Source: http://www.whiteboxtest.com/cyclomatic-complexity.php

```
int BinSearch (char *item, char *table[], int n)
{
  int bot = 0;
  int top = n - 1;
  int mid, cmp;
  while (bot <= top) {
    mid = (bot + top) / 2;
    if (table[mid] == item)
      return mid;
    else if (compare(table[mid], item) < 0)
      top = mid - 1;
    else
      bot = mid + 1;
  }
  return -1; // not found
}
```

Source: http://www.whiteboxtest.com/cyclomatic-complexity.php



Source: `http://www.whiteboxtest.com/cyclomatic-complexity.php`



Source: `http://www.whiteboxtest.com/cyclomatic-complexity.php`

In this example the CC is 4, thus this module has 4 linearly independent paths and this is the minimum number of test paths to achieve maximum coverage

TC0: 0->1->2->3->11

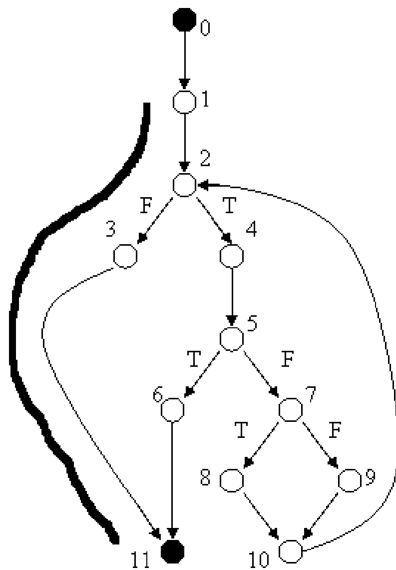
TC1: 0->1->2->4->5->6->11

TC2: 0->1->2->4->5->7->8->10->2->3->11

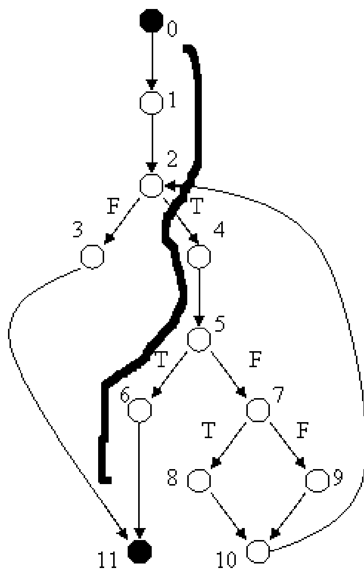
TC3: 0->1->2->4->5->7->9->10->2->4->5->6->11

These test paths achieve maximum coverage and any path other than these will dependent on these paths

Source: `http://www.whiteboxtest.com/cyclomatic-complexity.php`

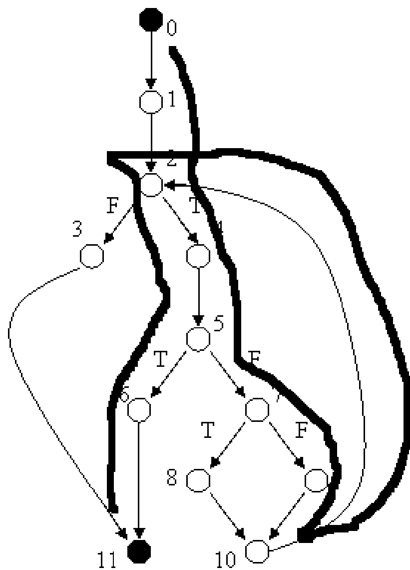


Source: `http://www.whiteboxtest.com/cyclomatic-complexity.php`



Source: `http://www.whiteboxtest.com/cyclomatic-complexity.php`

Source: `http://www.whiteboxtest.com/cyclomatic-complexity.php`



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.***

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.***

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*

5. *Separate concerns in modules*

Modules (classes [en Java]) that are loosely coupled are easier to modify and lead to a more modular system.

- ***Avoid large modules in order to achieve loose coupling between them.***
- *Do this by assigning responsibilities to separate modules and hiding implementation details behind interfaces.*

5. *Separate concerns in modules*

Modules (classes [en Java]) that are loosely coupled are easier to modify and lead to a more modular system.

- ***Avoid large modules in order to achieve loose coupling between them.***
- *Do this by assigning responsibilities to separate modules and hiding implementation details behind interfaces.*

Motivation

- *Small, loosely coupled modules allow developers to work on isolated parts of the codebase*
- *Small, loosely coupled modules ease navigation through the codebase*

6. *Couple architecture components loosely*

Top-level components of a system that are more loosely coupled are easier to modify and lead to a more modular system.

- ***Achieve loose coupling between top-level components.***
- *Do this by organizing the relative amount of code within modules that is exposed to (i.e., can receive calls from) modules in other components.*

6. *Couple architecture components loosely*

Top-level components of a system that are more loosely coupled are easier to modify and lead to a more modular system.

- ***Achieve loose coupling between top-level components.***
- *Do this by organizing the relative amount of code within modules that is exposed to (i.e., can receive calls from) modules in other components.*

Motivation

- *Low component dependence allows for isolated maintenance*
- *Low component dependence separates maintenance responsibilities codebase*

7. *Keep architecture components balanced*

A well-balanced architecture, with not too many and not too few components, of uniform size, is the most modular and enables easy modifications through separation of concerns.

- ***Balance the number and relative size of top-level components in your code.***
- *Do this by organizing source code in a way that the number of components is close to 9 (i.e., between 6 and 12) and that the components are of approximately the same size.*

7. *Keep architecture components balanced*

A well-balanced architecture, with not too many and not too few components, of uniform size, is the most modular and enables easy modifications through separation of concerns.

- ***Balance the number and relative size of top-level components in your code.***
- *Do this by organizing source code in a way that the number of components is close to 9 (i.e., between 6 and 12) and that the components are of approximately the same size.*

Motivation

- *A good component balance eases findind and analyzing code*
- *A good component balance better isolates maintenance effects*
- *A good component balance separates maintenance responsibilities*

8. *Keep your codebase small*

A large system is difficult to maintain, because more code needs to be analyzed, changed, and tested. Also, maintenance productivity per line of code *is lower in a large system than in a small system.*

- ***Keep you codebase size as small as feasible.***
- *Do this by avoiding codebase growth and actively reducing system size.*

8. *Keep your codebase small*

A large system is difficult to maintain, because more code needs to be analyzed, changed, and tested. Also, maintenance productivity per line of code *is lower in a large system than in a small system.*

- ***Keep you codebase size as small as feasible.***
- *Do this by avoiding codebase growth and actively reducing system size.*

Motivation

- *A project that sets out to build a large codebase is more likely to fail*
- *Large codebases are harder to maintain*
- *Large systems have higher defect density*

9. *Automate tests*

Automated tests (that is, tests that can be executed without manual intervention) enable near-instantaneous feedback on the effectiveness of modifications. Manual tests do not scale.

- ***Automate tests for your codebase.***
- *Do this by writing automated tests using a test framework.*

9. *Automate tests*

Automated tests (that is, tests that can be executed without manual intervention) enable near-instantaneous feedback on the effectiveness of modifications. Manual tests do not scale.

- ***Automate tests for your codebase.***
- *Do this by writing automated tests using a test framework.*

Motivation

- *Automated testing makes testing repeatable*
- *Automated testing makes development efficient*
- *Automated testing makes code predictable*
- *Tests document the code that is tested*
- *Writing tests make you write better code*

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.*

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*

6. Outils d’évaluation de la qualité du code

De nombreux outils sont disponibles pour évaluer la qualité du code

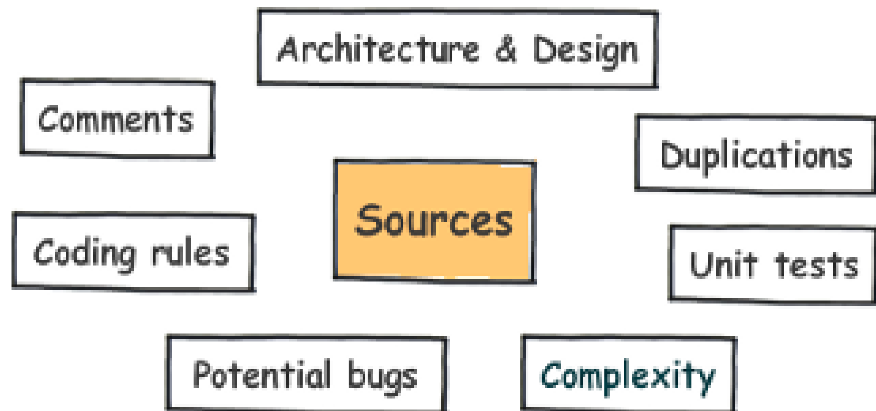
- C : `lint`, `BLAST`, `clang`, ...
- Ruby : `flog`, `flay`, `reek`, `rubocop`, ...
- Java : `CheckStyle`, `FindBugs`, `PMD`, `Coverity`, ...

SonarQube

http://www.sonarqube.org/

SonarQube is an open platform to manage code quality.

As such, it covers the 7 axes of code quality:



Utilisation de `SonarQube`

Intègre de nombreux autres outils

Utilisation de `SonarQube`

Intègre de nombreux autres outils

Dans un laboratoire à venir…

Utilisation de `SonarQube`

Intègre de nombreux autres outils

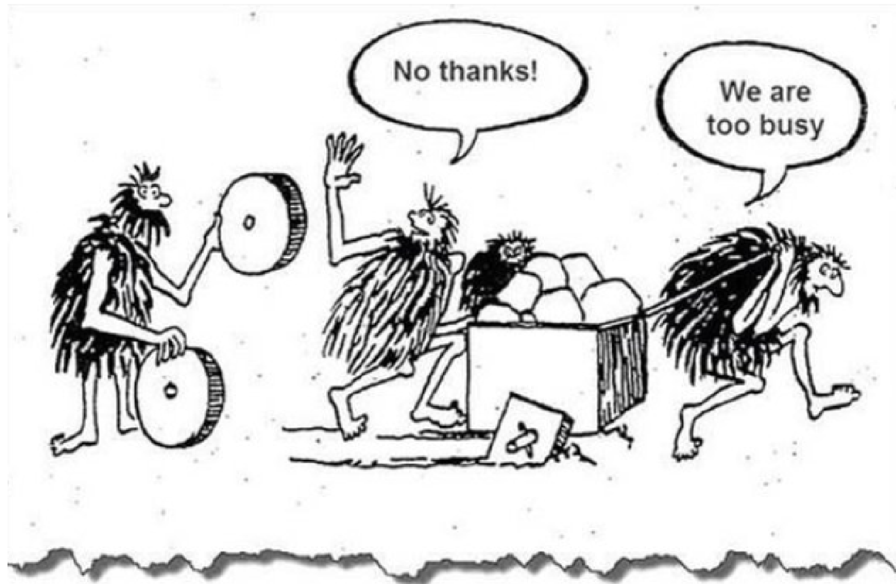
Dans un laboratoire à venir…

Notion importante en `SonarQube` = Dette technique

7. Dette technique et *Refactoring*

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

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



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

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

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*

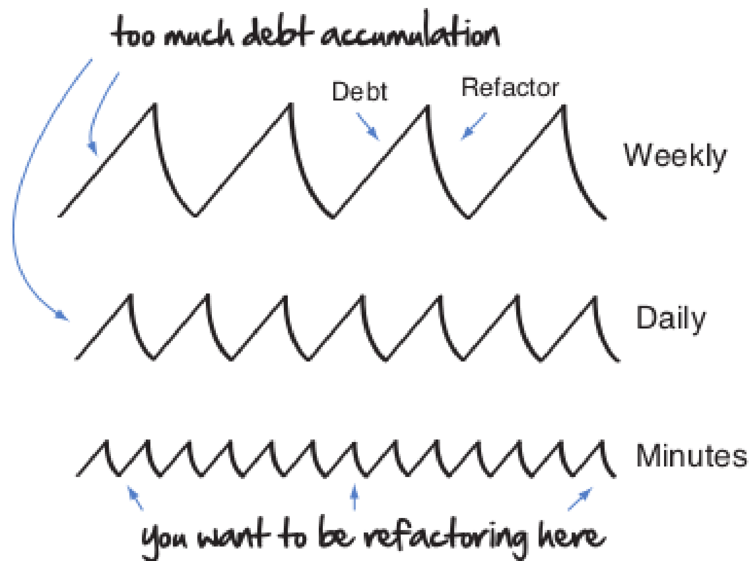
Dette technique

«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 faisant du *refactoring* !



Question : Qu’est-ce que le «*Refactoring*» ?

Question : 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.

Question : 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.

Refactor

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

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

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/*

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.

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*

Composing Methods

⋆

- Extract Method
- Inline Method
- Extract Variable
- Inline Temp
- Replace Temp with Query
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm

Moving Features between Objects

- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension

Organizing Data

★

- Self Encapsulate Field
- Replace Data Value with Object
- Change Value to Reference
- Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

Simplifying Conditional Expressions

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

Simplifying Method Calls

- Rename Method
- Add Parameter
- Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method Call
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Replace Error Code with Exception
- Replace Exception with Test

Dealing with Generalisation

⋆

- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

Pour les détails de ces *refactorings*, 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, notamment divers IDE pour Java

- Eclipse
- IntelliJ
- NetBeans

8. Code légataire

Question : Qu’est-ce que du code légataire ?

Question : Qu’est-ce que du code légataire ?

leg•a•cy code |ˈlegəsē kōd| **noun**

code you didn’t write (this morning)

Ashar Javed @soaj1664ashar · 26 juin 2014

Legacy Code === Code you didn't write this morning (via @pcreux)

16 2

Source: `https://twitter.com/soaj1664ashar/status/482117175080394753`

Question : Qu’est-ce que du code légataire ?

Source: <<Dilbert>>, S.~Adams

Qu’est-ce que du code légataire ?

[T]he software engineering community has developed [various] interpretations for the term legacy code. Among the most prevalent are source code inherited from someone else and source code inherited from an older version of the software.

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

Qu’est-ce que du code légataire ?

[T]he software engineering community has developed [various] interpretations for the term legacy code. Among the most prevalent are source code inherited from someone else and source code inherited from an older version of the software.

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

Qu’est-ce que du code légataire ?

[T]he software engineering community has developed [various] interpretations for the term legacy code. Among the most prevalent are source code inherited from someone else and source code inherited from an older version of the software.

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

Qu’est-ce que du code légataire ?

Legacy Code : That stuff that other people wrote. They’ve all since left the company, so there’s nobody left who knows how it works.

Source*:* `http://c2.com/cgi/wiki?LegacyCode`

Robert C. Martin Series
WORKING
EFFECTIVELY
WITH
LEGACY CODE
Michael C. Feathers

Qu’est-ce que du code légataire ?

«To me [M. Feathers], legacy code is simply. . .

Qu’est-ce que du code légataire ?

«To me [M. Feathers], legacy code is simply ***code without tests.***»

Qu'est-ce que du code légataire ?

«To me [M. Feathers], legacy code is simply ***code without tests.****»*

Code without tests is bad code. It doesn't matter how well written it is ; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.

Source: «*Working Effectively with Legacy Code*», *Feathers, 2005.*

Help ! I've Inherited Legacy Code

Extrait de «*Ship It !*»

You've inherited a legacy product that you will be maintaining and enhancing. What's the quickest way to get a handle on it ?

Help ! I've Inherited Legacy Code

Extrait de «*Ship It !*»

You've inherited a legacy product that you will be maintaining and enhancing. What's the quickest way to get a handle on it ?

Learn to build it, automate it, and finally test it.

Help ! I've Inherited Legacy Code

Extrait de «*Ship It !*»

You've inherited a legacy product that you will be maintaining and enhancing. What's the quickest way to get a handle on it ?

Learn to build it, automate it, and finally test it.

Tip 25 : Don't change legacy code until you can test it.

Source*: «Ship It !», Richardson & Gwaltney, 2005.*

1. *Build it : First, figure out how to build it, and then script that build process.*

1. Tests d'acceptation.

1. *Build it : First, figure out how to build it, and then script that build process.*
2. *Automate it : Your goal is to automatically build and test the entire product on a clean machine with an absolute minimum of manual intervention.*

1. Tests d’acceptation.

Help ! I've Inherited Legacy Code

Build it ! Automate it ! Test it !

1. *Build it : First, figure out how to build it, and then script that build process.*
2. *Automate it : Your goal is to automatically build and test the entire product on a clean machine with an absolute minimum of manual intervention.*
3. *Test it : Figure out what the code does, then begin testing by writing mock client tests[1] for it.*

---

1. Tests d'acceptation.

Help ! I've Inherited Legacy Code

Build it ! Automate it ! Test it !

1. *Build it : First, figure out how to build it, and then script that build process.*

2. *Automate it : Your goal is to automatically build and test the entire product on a clean machine with an absolute minimum of manual intervention.*

3. *Test it : Figure out what the code does, then begin testing by writing mock client tests[1] for it.*

4. *Test it more : Figure out the product's innards (things such as structure, flow-of-control, performance, and scalability), and write more tests for it. [. . .] Write a new test for every bug you fix and for every enhancement you add to the product.*

---

1. Tests d'acceptation.

Références

M.C. Feathers.
Working Effectively with Legacy Code.
Prentice-Hall, 2005.

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.

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`

- «*Smells to Refactorings Quick Reference Guide*»
  `https://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/`

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