Université de Lille 1 SCIENCES ET TECHNOLOGIES

UFR IEEA
Formations en Informatique de Lille 1

Conception Orientée Objet

Romain Rouvoy
Licence mention Informatique
Université Lille 1

Menu du jour

1. Dettes
2. Analyses
3. Métriques
4. Outils

Pourquoi améliorer la qualité ?

Motivations

« *Better, faster, stronger* »

- Développer un logiciel
  - Plus performant
  - Plus léger
  - Plus efficient
  - Plus évolutif
  - Plus maintenable
  - Moins de bugs

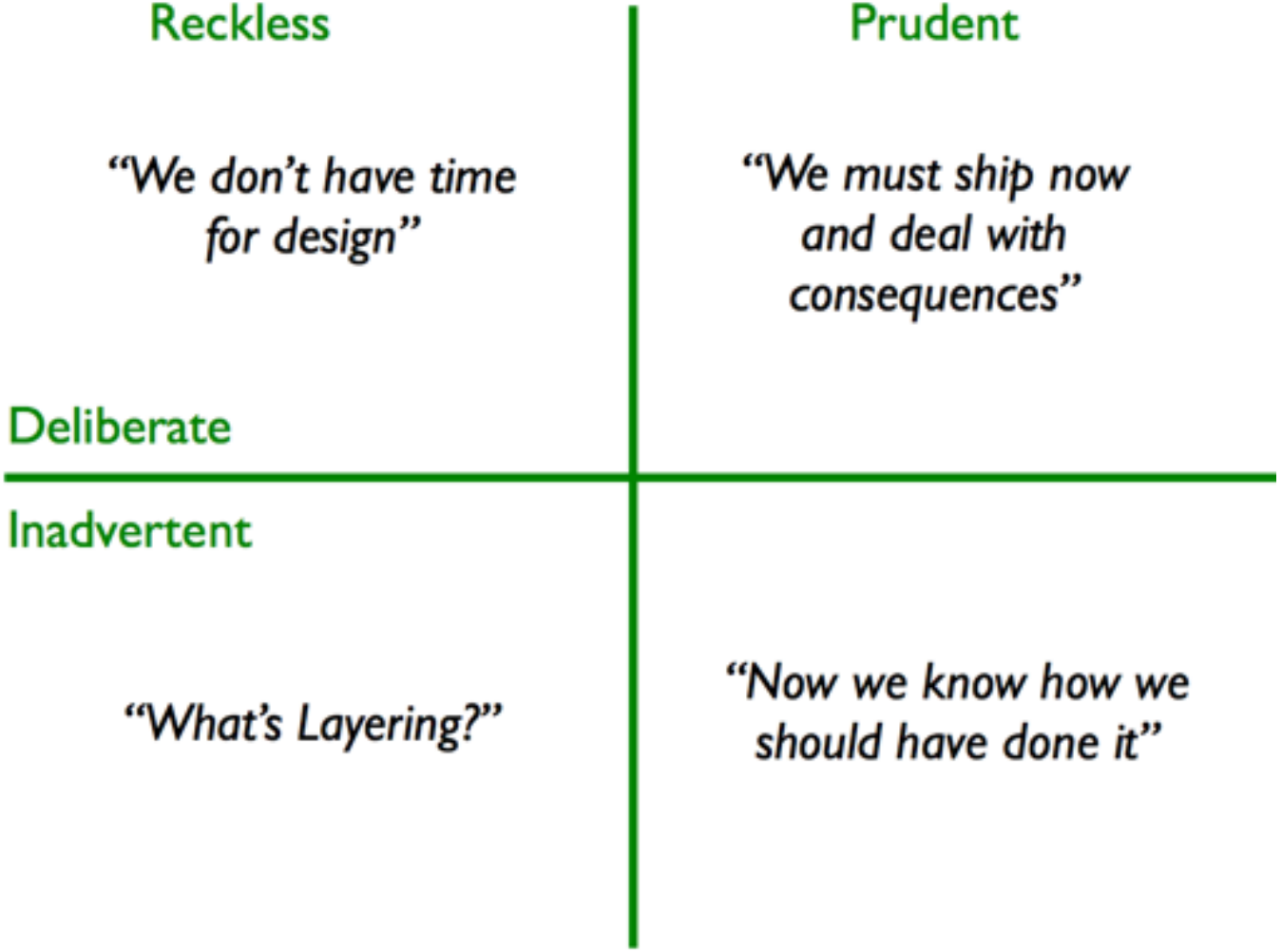
« Une **grande boule de boue** est une vaste jungle de code mal structuré, programmé en spaghetti, peu soigné et souvent rafistolé. Ces systèmes témoignent clairement des traces d'une **expansion incontrôlée**, ainsi que de **fréquentes réparations opportunes et improvisées**. Les informations sont partagées sans distinction parmi les éléments distants du système, souvent jusqu'au point où presque toutes les **informations importantes deviennent globales ou dupliquées**. La structure du système dans son ensemble n'a peut-être **jamais été bien définie**. Si elle l'a été, le système a été tellement **détérioré** qu'il est impossible de la reconnaître. Les programmeurs avec un brin de sensibilité architecturale fuient ces bourbiers. Seuls ceux qui ne sont pas préoccupés par l'architecture, et qui, peut-être, ne sont pas dérangés par l'inertie d'une corvée quotidienne consistant à **coller des rustines sur ces digues défaillantes**, sont heureux de travailler sur de tels systèmes. »

La dette technique

- Métaphore inventée par W. Cunningham
- Inspiration du terme «dette» dans la finance
- Peut-être non-intentionnelle ou intentionnelle

«[…] *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**.*»

La dette technique



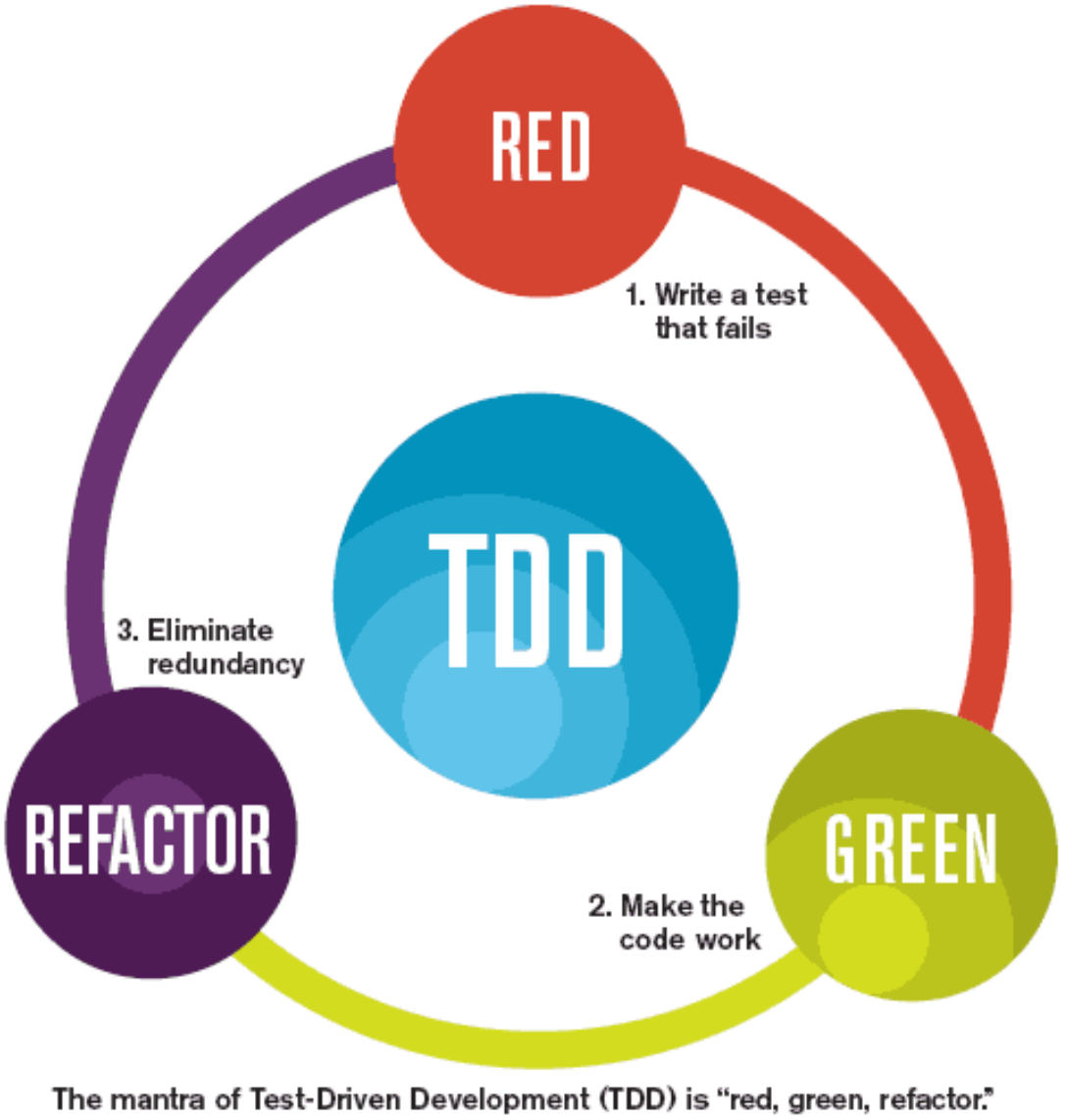
Les 7 péchés du développeur

1. Duplication de code
2. Pas ou trop de commentaires
3. Pas de tests unitaires
4. Pas de respect des standards
5. Mauvaise conception
6. Bugs potentiels
7. Mauvaise distribution de la complexité

KISS

DRY

YAGNI



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Types d’analyses

- Dynamiques
  - Tests
  - Oracles
- Statiques
  - Code / artefacts
  - Métriques

Les métriques les plus simples

- Nombre de classes / méthodes / attributs

- LoC (*Lines of Code*)
  - **LoC_{phy}** : nb lignes physiques
  - **LoC_{pro}** : nb lignes de programme
  - **LoC_{com}** : nb lignes de commentaire
  - **LoC_{bl}** : nb lignes vides

Quelle utilité ?

- Approximation de la complexité d'un projet
- Déterminer la longueur max d'une méthode (< 40)
- Déterminer la longueur max d'un fichier (< 400)
- Déterminer le ratio de commentaires (30–75 %)

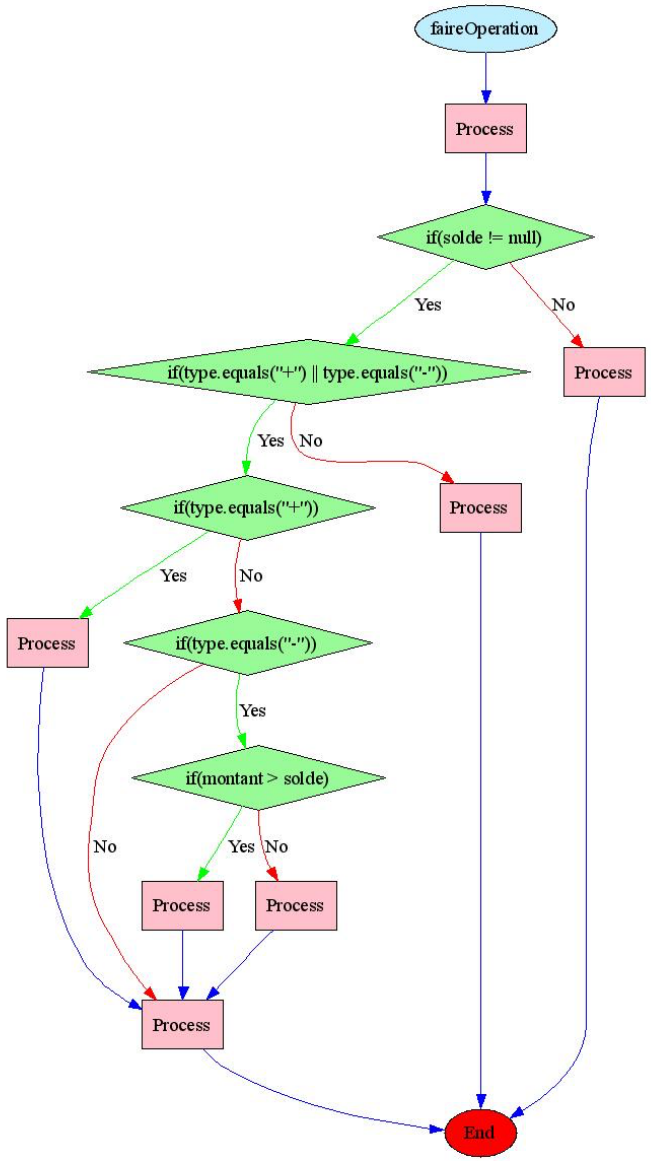
Mesure de McCabe

Ou « nombre/complexité cyclomatique »

- Introduite par Thomas McCabe en 1976
- Mesure de la complexité d'un programme info
  - Indépendamment du langage de programmation
- Mesure du nombre de chemins empruntables
  - ⊕ de chemins ⇒ ⊕ difficile à tester (couverture)

Mesure de McCabe (2)

- $M = E - N + 2P$
- Où
  - M = complexité cyclomatique
  - E = le nombre d'arêtes du graphe
  - N = le nombre de nœuds du graphe
  - P = le nombre de composantes connexes du graphe



Mesure de McCabe (3)

1. pour un programme qui consiste en seulement des états séquentiels, la valeur pour `v(G)` est `1`
2. chaque **if-statement** introduit une nouvelle branche au programme => incrémente `v(G)` ;
3. chaque **itération** (une boucle for ou while) introduit des branches => augmente le facteur `v(G)` ;
4. chaque **switch-statement** du commutateur => incrémente `v(G)`
   - une branche de cas n'augmente pas la valeur de `v(G)`, car il ne doit pas augmenter le nombre de branches dans le flux de commande. S'il y a deux ou plus de cas... : qui n'ont pas de code entre eux, la mesure McCabe est augmentée uniquement une fois pour l'ensemble des cas
5. chaque morceau (...) attrapé dans un **bloc d'essai** (try-block) incrémente `v(G)`
6. les **constructions** exprl ? expr2 : expr3 incrémentent `v(G)`

```
#include <iostream>
#include <string>
// Cyclomatic complexity :
int main(int argc, char * argv[]) {                                    //CC += 1
        if (argc != 2) {                                               //CC += 1
                std::cerr << "un argument nécessaire" << std::endl ;
                return 1 ;
        }
        try {
                int a = std::stoi(argv[1]) ;
                int sum = 0 ;
                for (int i = 1; i < a ; ++i) {                         //CC += 1
                        if (a % i == 0) {                              //CC += 1
                                sum += i ;
                }       }
                if (sum == a) {                                        //CC += 1
                        std::cout << "a est parfait" << std::endl ;
                } else if (sum < a) {                                  //CC += 1
                        std::cout << "a est déficient" << std::endl ;
                } else { std::cout << "a est abondant" << std::endl ; }
        } catch (std::invalid_argument const & e) {                    //CC += 1
                std::cout << "Error : " << e.what() << std::endl ;
                return 1 ;
        }
        return 0 ;
}                                                                      //Total CC = 7
```

Quelle utilité ?

- Mesure de McCabe
  - < 15 pour une fonction
  - < 100 pour un fichier
- Détection de l'antipatron *Objet Divin*
  - Concentration de méthodes complexes

Autres métriques

- Mesure de Halstead
  - Basée sur la notion d'opérateur / opérande
- Indice de maintenabilité
  - Combine Halstead et McCabe

...

Métrique de couplage

Selon R. Pressman, il existe 7 niveaux de couplage, du plus faible au plus fort :

- **Sans couplage** : les composants *n'échangent pas d'information*
- **Par données** : les composants échangent de l'information par des méthodes utilisant des *arguments de type simple* (nombre, chaîne de caractères, tableau)
- **Par paquet** : les composants échangent de l'information par des méthodes utilisant des *arguments de type composé* (structure, classe)
- **Par contrôle** : les composants se passent ou modifient leur contrôle par *changement d'un drapeau* (verrou)
- **Externe** : les composants échangent de l'information par un moyen de *communication externe* (fichier, pipeline, lien de communication)
- **Commun (global)** : les composants échangent de l'information via un *ensemble de données* (variables) *commun*
- **Par contenu (interne)** : les composants échangent de l'information en *lisant et écrivant directement dans leurs espaces de données* (variables) respectifs

Une bonne architecture logicielle nécessite le couplage le plus faible possible

Quelle utilité ?

- Un couplage fort génère l'antipattern **plat de spaghetti**
  - On ne peut pas déterminer le qui, le quoi et le comment d'une modification de données
- Un couplage fort implique nécessairement une faible indépendance fonctionnelle :
  - Le composant logiciel est **difficilement réutilisable**
  - Le composant logiciel est **difficilement testable**
- Si deux tâches accèdent, par couplage fort, à une ressource commune (ressource critique) et qu'elles s'exécutent en exclusion mutuelle, alors si une des tâches reste bloquée en section critique elle bloque automatiquement l'autre :
  - **Risque d'interblocage**

Les composants perdent leur autonomie. On peut **difficilement remplacer un composant par un autre**. Les structures fonctionnant avec du couplage fort sont donc **peu souples et peu ouvertes**.

Métrique de cohésion

Selon R. Pressman, il existe 7 niveaux de cohésion :

1. **Accidentel** : décrivant le niveau le plus faible où le lien entre les différentes méthodes est inexistant ou bien créé sur la base d'un critère futile
2. **Logique** : lorsque les méthodes sont reliées logiquement par un ou plusieurs critères communs
3. **Temporel** : lorsque les méthodes doivent être appelées au cours de la même période de temps
4. **Procédural** : lorsque les méthodes doivent être appelées dans un ordre spécifique
5. **Communicationnel** : lorsque les méthodes manipulent le même ensemble spécifique de données
6. **Séquentiel** : lorsque les méthodes qui manipulent le même ensemble de données doivent être appelées dans un ordre spécifique
7. **Fonctionnel** : réalise le niveau le plus élevé lorsque la classe ou le module est dédié à une seule et unique tâche bien spécifique

Le niveau accidentel est celui de plus faible cohésion, le niveau fonctionnel celui de plus forte cohésion ; **une bonne architecture logicielle nécessite la plus forte cohésion possible**.

Quelle utilité ?

- En programmation objet, le respect des **principes d'encapsulation des données** permet d'obtenir le niveau de *cohésion communicationnel*

- Le *niveau séquentiel* est atteint par observation du **principe de masquage de l'information** et **l'utilisation de patrons de conception** reconnus qui permettent de créer des interfaces dont l'ordre des appels est normalisé (vous en connaissez une, vous les connaissez toutes)

- Le *niveau fonctionnel* est ici un idéal, rappelant sans cesse que **moins une interface contient de méthodes, plus elle est simple à utiliser**

Méthode SQALE

- développée pour combler un besoin général relatif à l'évaluation du code source d'une application

- Comment répondre aux questions récurrentes
  - Quelle est la qualité du code que les développeurs m’ont livré ?
  - Est-ce que ce code est évolutif, maintenable, portable, réutilisable ?
  - Quelle est la dette technique accumulée par le projet ?
  - Je souhaite rembourser une partie de ma dette technique, par où commencer ?

Méthode SQALE

- D'identifier et de définir de façon claire tout ce qui crée de la dette technique
- D'estimer précisément et objectivement cette dette
- De décomposer cette dette en différentes parties relatives à la testabilité, la fiabilité, l'évolutivité, la maintenabilité… décomposition permettant d'analyser l'impact de la dette sur le projet et d'identifier ainsi les actions de refactoring à lancer en priorité.
- D'analyser les risques encourus dans le cas où on ne procèderait pas à la remédiation des non-conformités constatées.
- D'établir des stratégies d'amélioration de la qualité du code tenant compte du contexte de la situation.

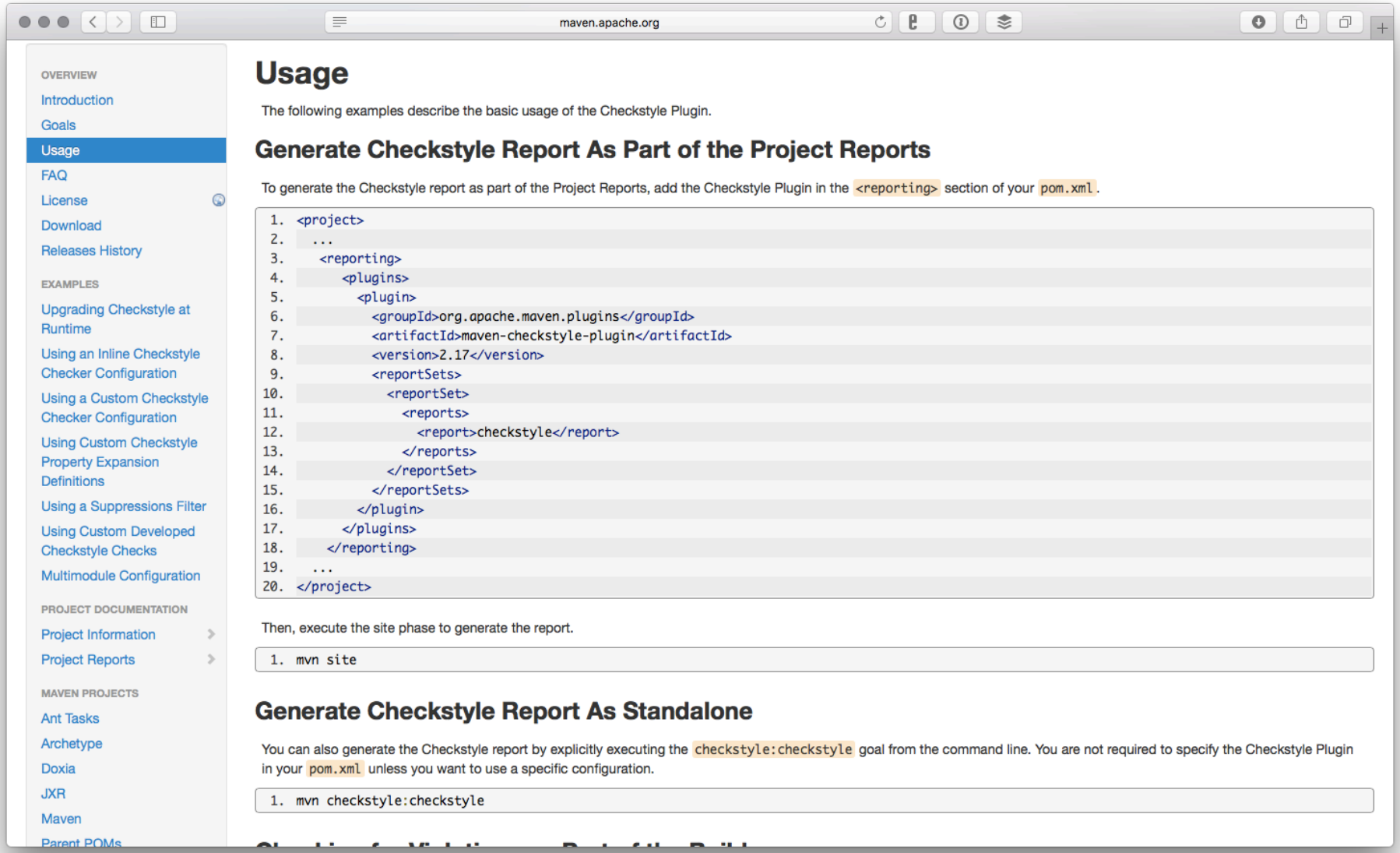
Équipez-vous !

Améliorer le qualité en pratique

- Connaître ses **fondamentaux**
  - SOLID, TDD, DP, AP, etc. (cours)
- Travailler en équipe
  - Et être ouvert à ses coéquipiers
- Utiliser les outils appropriés
  - Pas seulement Eclipse
- Mettre en œuvre une veille continue
  - Être curieux
- Vouloir perpétuellement s'améliorer

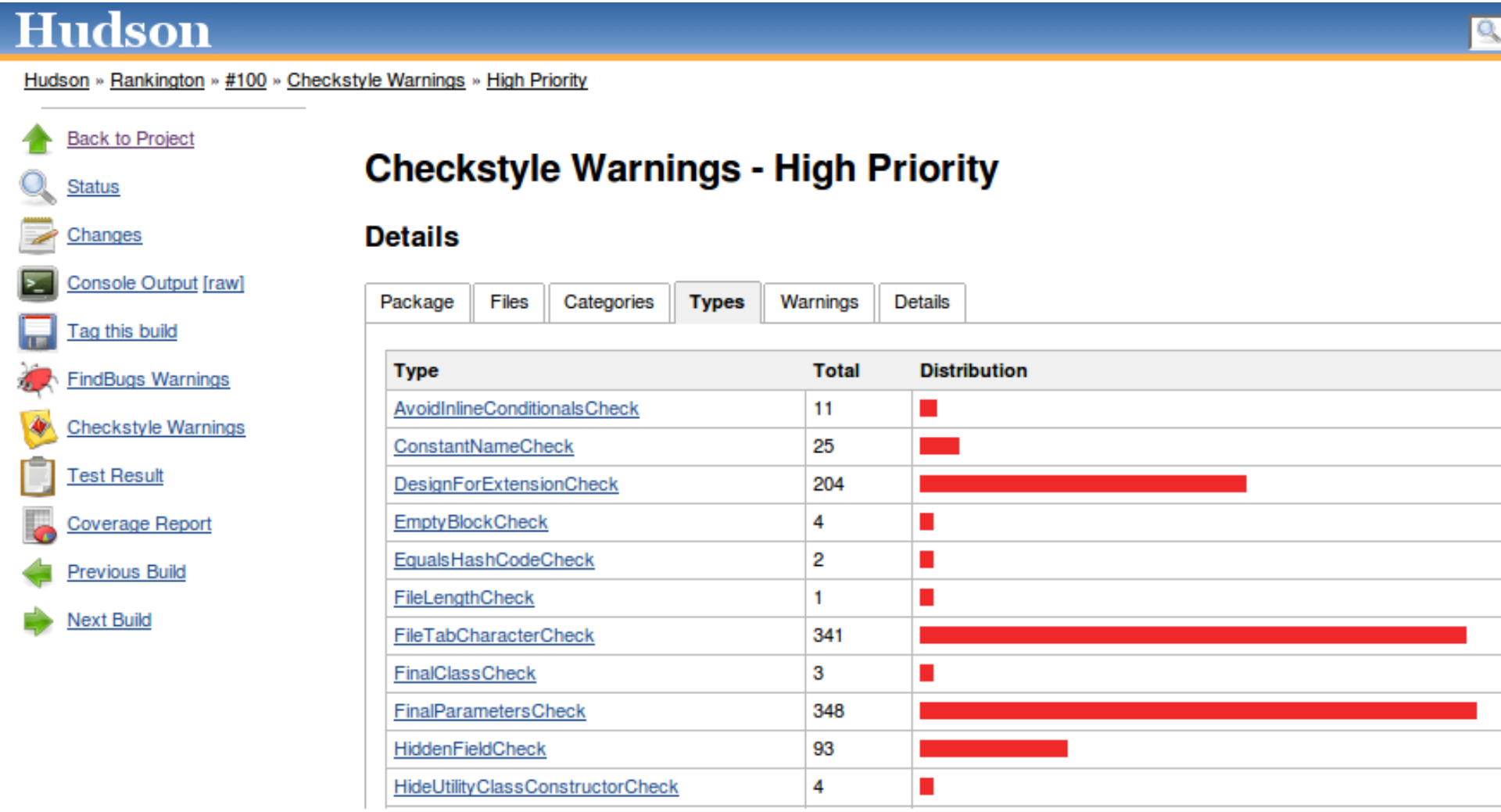
Quelques outils

- **Checkstyle**: code style validation and design checks
- **PMD**: Code checks (possible bugs, dead code, sub-optimal code, etc.)
- **PMD CPD**: Duplicate code (always a bad idea) detection
- **FindBugs**: fantastic tool to detect potential bugs (really!)
- **Cobertura**: Coverage tool
- **Simian**: excellent duplicate code detection
- **SonarQube**: SQALE reporting


maven.apache.org
OVERVIEW
Introduction
Goals
Usage
FAQ
License
Download
Releases History
EXAMPLES
Upgrading Checkstyle at Runtime
Using an Inline Checkstyle Checker Configuration
Using a Custom Checkstyle Checker Configuration
Using Custom Checkstyle Property Expansion Definitions
Using a Suppressions Filter
Using Custom Developed Checkstyle Checks
Multimodule Configuration
PROJECT DOCUMENTATION
Project Information
Project Reports
MAVEN PROJECTS
Ant Tasks
Archetype
Doxia
JXR
Maven
Parent POMs
Usage
The following examples describe the basic usage of the Checkstyle Plugin.
Generate Checkstyle Report As Part of the Project Reports
To generate the Checkstyle report as part of the Project Reports, add the Checkstyle Plugin in the <reporting> section of your pom.xml.
1. <project>
2. ...
3. <reporting>
4. <plugins>
5. <plugin>
6. <groupId>org.apache.maven.plugins</groupId>
7. <artifactId>maven-checkstyle-plugin</artifactId>
8. <version>2.17</version>
9. <reportSets>
10. <reportSet>
11. <reports>
12. <report>checkstyle</report>
13. </reports>
14. </reportSet>
15. </reportSets>
16. </plugin>
17. </plugins>
18. </reporting>
19. ...
20. </project>
Then, execute the site phase to generate the report.
1. mvn site
Generate Checkstyle Report As Standalone
You can also generate the Checkstyle report by explicitly executing the checkstyle:checkstyle goal from the command line. You are not required to specify the Checkstyle Plugin in your pom.xml unless you want to use a specific configuration.
1. mvn checkstyle:checkstyle

Configuration depuis Maven

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.17</version>
        <reportSets>
          <reportSet>
            <reports>
              <report>checkstyle</report>
            </reports>
          </reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```


Hudson
Hudson » Rankington » #100 » Checkstyle Warnings » High Priority
Back to Project
Status
Changes
Console Output [raw]
Tag this build
FindBugs Warnings
Checkstyle Warnings
Test Result
Coverage Report
Previous Build
Next Build
Checkstyle Warnings - High Priority
Details
Package
Files
Categories
Types
Warnings
Details
Type
Total
Distribution
AvoidInlineConditionalsCheck 11
ConstantNameCheck 25
DesignForExtensionCheck 204
EmptyBlockCheck 4
EqualsHashCodeCheck 2
FileLengthCheck 1
FileTabCharacterCheck 341
FinalClassCheck 3
FinalParametersCheck 348
HiddenFieldCheck 93
HideUtilityClassConstructorCheck 4

Configuration depuis Maven

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.7</version>
        <reportSets>
          <reportSet>
            <reports>
              <report>checkstyle</report>
            </reports>
          </reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Back to Project
Status
Changes
Console Output
Edit Build Information
Delete Build
Git Build Data
No Tags
PMD Warnings
Previous Build

PMD Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
17	0	0

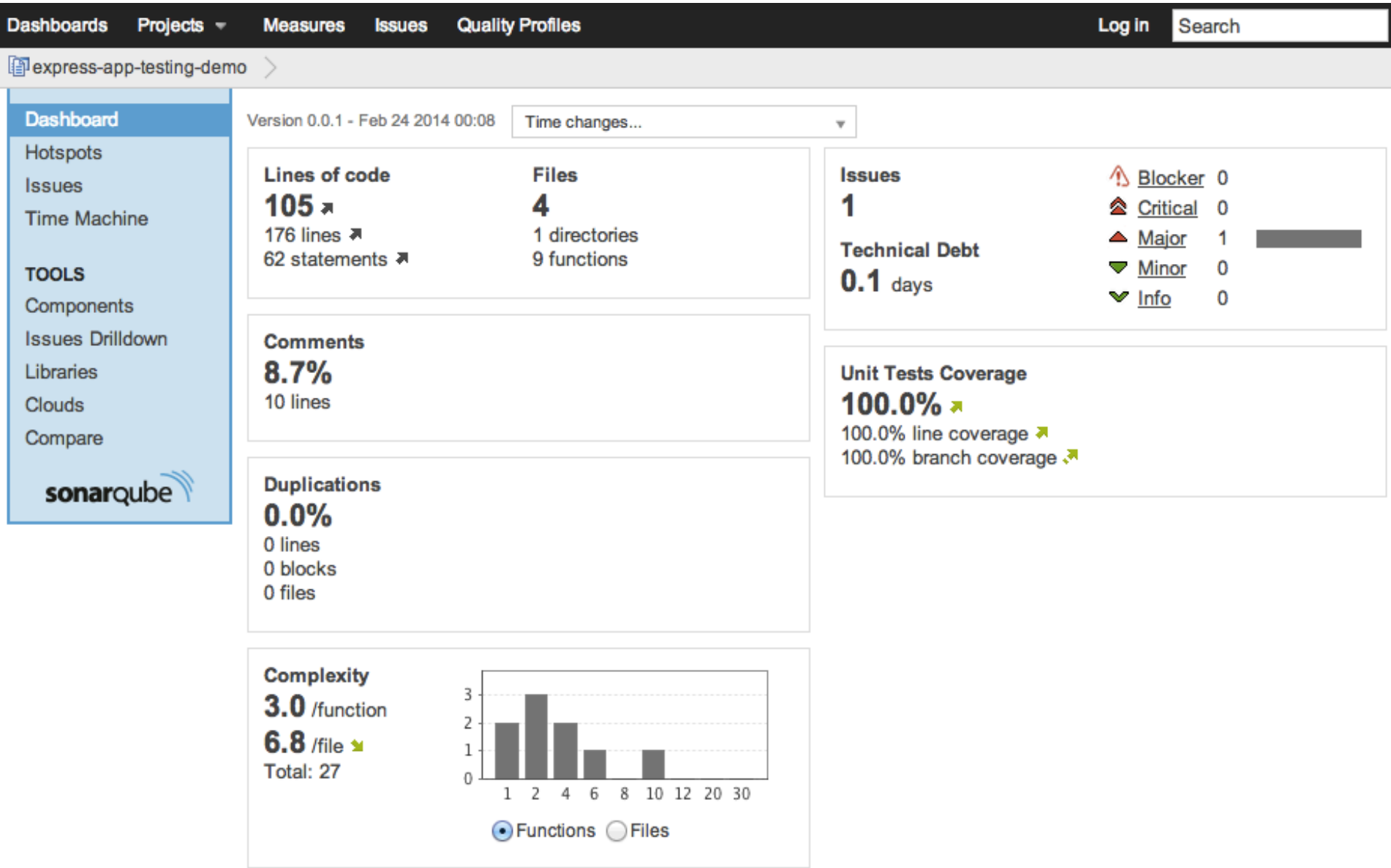
Summary

Total	High Priority	Normal Priority	Low Priority
17	0	7	10

Details

Files | **Types** | Warnings | Details | Normal | Low

Type	Total	Distribution
collapsible if statements	1	
dead code	1	
empty for statement	1	
empty while statement	1	
high cyclomatic complexity	2	
high ncss method	1	
long line	1	
long method	1	
missing break in switch statement	2	
parameter reassignment	2	
switch statements should have default	1	
too few branches in switch statement	2	
unused method parameter	1	
Total	**17**	



https://www.youtube.com/watch?v=-7lbpA_iclc

En résumé

- Dette technique
- Analyse dynamique Vs. Statique
- Métriques de code
- Liens avec les antipatrons
- Outils et méthodes pour (s')améliorer

- PRACTICE, PRACTICE, PRACTICE !!!

Au prochain épisode…

- Générer le code pour garantir la qualité ?
- Transformer le code pour maintenir la qualité ?