# M3105 – Qualité & harnais de tests sur le Kata Trivia : SonarQube, Golden Master, Approval Testing

***Cet énoncé est disponible en ligne sur :***
**https://github.com/iblasquez/enseignement-iut-m3105**

Vous venez de rejoindre le projet **`trivia`** et vous héritez donc d'un code legacy existant.
L'objectif de ce TP est d'une part de faire un point sur l'état du projet
et d'autre part de mettre en place un harnais de tests
pour pouvoir procéder à un refactoring ultérieur.

## Exercice 1 : A la découverte du projet `trivia` ...

### 1. Mise en place du projet

Dans votre IDE préféré, créez un projet Maven **`trivia`** dans lequel vous y importerez **les sources** disponibles dans le répertoire **ressources `trivia`** de ce dépôt : https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee.

**Faites en sorte que ce code compile !**

**Le formatage de ce code** ne convient peut-être pas à certains d'entre vous car il ne correspond pas à la *forme* des programmes que vous avez l'habitude d'écrire avec votre IDE préféré (en terme d'accolades, d'indentation…)…
Pas de problème, commencez donc pas reformater ce code de manière à ce qu'il corresponde mieux à vos standards habituels (sous Eclipse sélectionnez **Source->Format**)

Faites en sorte que le code compile toujours !
(normalement le formatage n'a rien changé et le code doit continuer à compiler !)

**Mettez votre projet sous git et procédez à votre premier commit avec un message du genre :** « Commit initial : récupération du code legacy ».

### 2. Se familiariser avec le code existant ...

#### 2.1 Comprendre le comportement du code ... (état des lieux « fonctionnel »)

Que fait ce code ? Y-a-t-il des tests ?
Le code legacy peut être difficile à comprendre, surtout si vous ne savez pas comment il est censé se comporter (manque de spécifications, manque de tests,…)

La première étape pour se familiariser avec un code legacy peut consister à exécuter le code et à essayer de comprendre son comportement.

Disposez-vous d'un fichier qui pourrait exécuter ce code et vous aider à comprendre son comportement ? Si, oui exécutez-le …

Qu'apprenez-vous sur ce code suite à cette exécution ?



Grâce à la méthode `main` de la classe `GameRunner`, nous pouvons exécuter le code et constater qu'il produit un *output* qui n'est autre qu'un affichage dans la console.
Nous allons maintenant consulté attentivement cet affichage et analyser son contenu pour en apprendre plus sur ce que fait ce code…

- ➔ Nous savons que ce projet est un **jeu de `trivia`** (questions/réponses)

- ➔ Combien de joueurs sont impliqués dans la partie dans cet exemple ?
  - ⇨ A priori trois joueurs : `Chet`, `Pat` et `Sue`

- ➔ Comment le jeu progresse-t-il ?
  - ⇨ A priori, il y a une sorte de lancer de dé (`have rolled`) ou un concept similaire

- ➔ A quoi sert ce lancer de dé ?
  - ⇨ A priori, à déplacer le joueur puisque le joueur atteint une sorte de position après chaque lancer de dé (`new location is xx`). Peut-être une sorte de plateau de jeu ?

- ➔ Que se passe-t-il lorsqu'un joueur atteint une nouvelle position ?
  - ⇨ Une question lui est posée. Il semblerait qu'il y ait différentes catégories de questions. (`The category is XXX - XXX Question numeroQuestion`)

- ➔ Que se passe-t-il ensuite ?
  - ⇨ Le joueur répond à la question …
    - Si la réponse est correcte le joueur récupère 1 pièce d'or (`Gold Coins`)
    - Si la réponse est incorrecte, le joueur reçoit une pénalité : il est envoyé vers une `penalty box`
      ⇒ A moment donné il semblerait que le joueur puisse sortir de la `penalty box`, mais peut-être ne comprenez-vous pas encore vraiment comment …

- ➔ Quand la partie se termine-t-elle ?
  - ⇨ A priori, lorsqu'un des joueurs a récupéré 6 pièces d'or…

Maintenant que nous en savons un peu plus sur le comportement de ce système, il est temps de revenir vers le code…

#### 2.2 Jetez un petit coup d'œil sur le code (revue de code : état des lieux « technique »)

Nous allons commencer par une **revue de code** (manuelle) de l'existant…
Ce projet se compose de deux classes : `Game` et `GameRunner`.

- ➔ Commencez par faire une rapide revue de code de la classe **`GameRunner`.**
  Que pensez-vous de cette classe ?
  C'est la classe qui permet, au travers de la méthode `main`, de lancer l'exécution du programme. On retrouve les trois joueurs, les lancers de dés, le traitement des réponses justes ou fausses au travers d'une succession d'instructions qui constituent un scénario (test) pour simuler une partie de trivia.
  Pour que les parties soient différentes, ce scénario fait appel à un générateur de nombre aléatoire pour simuler le lancer de dé.

- ➔ Continuez par une rapide revue de code de la classe **`Game`.**
  Que pensez-vous de cette classe ? Détectez-vous des code smells ?
  Cette classe mériterait-elle un petit refactoring ?

Effectivement, la classe `Game` présente de nombreux signes de mauvais design.
Ce n'est pas pour rien que ce code se trouve dans un package nommé `uglytrivia` 😉

***Remarque :*** ce code est celui d'un kata de refactoring disponible dans de nombreux autres langages sur le dépôt suivant : https://github.com/jbrains/trivia

Avant de se lancer dans un quelconque refactoring, nous allons profiter de la « mauvaise » qualité de ce code pour prendre en main **SonarQube**.

### 2.3 En savoir plus sur la qualité du code via SonarQube (métriques : état des lieux « technique » - analyse statique du code)

**SonarQube** est un logiciel libre développé par la société Sonarsource.
Le site de SonarQube est : https://www.sonarqube.org/
Ce logiciel permet de mesurer la qualité de votre projet de plusieurs manières (respect des règles de codage, documentation du code, analyse des tests unitaires mis en place, duplication du code,…) et fournit de nombreuses métriques liées à la qualité de votre projet.

**Remarques** (plus de détails en annexe) **:**

- ➔ **SonarQube** supporte plus 20 langages comme le Java, C, C++, C#, PHP, Javascript…
- ➔ Nous avons déjà utiliser le un plug-in **SonarLint**, plus léger que SonarQube, qui fonctionne uniquement dans l'IDE et permet de détecter au travers d'**issues** le non-respect de règles de programmation.
  Dans ce TP, nous allons installer et utiliser **SonarQube** qui, outre la détection d'issues, est capable de fournir des mesures quantitatives et qualitatives sur la qualité d'un projet.
  A noter qu'il est également possible d'utiliser une version en ligne de SonarQube appelé **SonarCloud** sur https://about.sonarcloud.io/

#### 2.3.a Prendre en main SonarQube…

Pour prendre en main SonarQube, rendez-vous à l'adresse suivante : **http://unil.im/sonar.**

Vous devez **effectuer ce tutoriel directement sur le projet `trivia`** (et non sur le projet `tennisKataRefactoring1` comme le suggère initialement le tutoriel).

Vous pouvez donc commencer le tutoriel directement à la partie **Installation de SonarQube** après vous être assuré que vous disposez bien d'une **variable d'environnement** `JAVA_HOME`.

***Remarque :*** http://unil.im/sonar est un raccourci pour
https://github.com/iblasquez/tutoriel_SonarQube/blob/master/Analyse_SonarQubeServer.md

#### 2.3.b Relever quelques métriques fournies par SonarQube

Pour vous familiariser avec le **tableau de bord** (**dashboard**) de **SonarQube**, et les différentes métriques et informations que ce logiciel peut fournir, nous vous proposons de compléter le tableau suivant :



**Informations disponibles sur le dashboard (tableau de bord accessible au lancement ou via 🏠 )**

| | | |
|---|---|---|
| **Nombre de lignes de code** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires (à noter que les détails correspondent à un sous menu de la vue **Measures**)* |
| **Dette technique (durée, jours, heures…)** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires (à noter que les détails correspondent à la vue **Issues**)* |
| **Nombre de code smells** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires (à noter que les détails affichés correspondent à la vue **Issues**)* |
| **Taux de Duplications :** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires (à noter que les détails affichés correspondent à un sous menu de la vue **Measures**)* |
| **Nombre de blocs dupliqués** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires (à noter que les détails affichés correspondent à un sous menu de la vue **Measures**) Cliquez sur le fichier contenant des blocs dupliqués… Cherchez une ligne orange dans le code qui va montrer quels blocs sont dupliqués (vous pouvez cliquer sur cette ligne orange pour plus d'information)* |
| **Taux de couverture** | …… | |

Ouvrir l'onglet **Measures :**

| | | |
|---|---|---|
| **Complexité cyclomatique**<br>- dans `Game`<br>- dans `GameRunner` | ……<br>……<br>…… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires* |
| **Nombre Issues**<br>- dans `Game`<br>- dans `GameRunner` | ……<br>……<br>…… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires* |
| **Nombre de lignes de code**<br>- dans `Game`<br>- dans `GameRunner` | ……<br>……<br>…… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires* |
| **Nombre de classes** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires* |
| **% de commentaire** | …… | *Cliquez sur cette valeur pour obtenir des détails supplémentaires* |

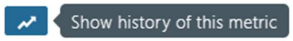
Consultez les onglets **Code** et **Activity** pour d'autres vues offertes par SonarQube…

En résumé, nous pouvons dire que **SonarQube** fournit un **tableau de bord** des projets analysés qui permet une détection rapide du code « à risque » et des points faibles d'un projet.

Au travers de ce tableau, SonarQube fournit :
- des **mesures quantitatives** : nombre de classes, duplication de code, etc.
- des **mesures qualitatives** : complexité du code, couverture et taux de réussite des tests, complexité du code, respect des règles de codage, etc.

**Pour connaître les règles de codage standard** sur lesquelles s'est basé Sonar pour son analyse statique cliquez sur le menu **Rules** dans le bandeau noir en haut de votre fenêtre, puis sur **Java**.
Relevez le nombre de règles actuellement présentes dans Sonar pour le langage Java : ……
Il est à noter que vous pouvez ajouter vos propres règles de codage, mais nous ne le ferons pas maintenant 😉

Si vous lancez plusieurs analyses, vous constaterez que Sonar peut également fournir un **historiques des métriques**.
Pour accéder à cette historique, utilisez le menu **Activity** ou directement à côté des métriques cliquez sur l'icône suivante :




La revue de code manuelle et les métriques Sonar vous ont permis d'identifier des problèmes de qualité dans ce code : un certain nombre de code smells, un certain nombre de signes de mauvais design, une complexité cyclomatique un peu trop élevée, un peu de duplication,…

Il semblerait que ce projet ait donc besoin d'un petit refactoring…
Mais attention, pour garantir le comportement du système, il est indispensable, avant de procéder à un quelconque refactoring, de disposer d'un ensemble de tests automatisés, *filet de sécurité* pour remanier le code en toute sécurité. 😉

Et pour l'instant, vous ne disposez d'aucun fichier de test pour couvrir ce code dans **src/test/java** 😉 La première chose à faire est donc d'écrire des tests sur ce code legacy !

Comme dans le TP précédent, nous allons écrire ces tests (dits **Characterization tests**) en nous appuyant sur la technique du **Golden Master Testing**.



## Exercice 2 : Mise en place d'un harnais de test via un Golden Master manuel

La liste suivante énonce les différentes étapes à suivre pour mettre en place la place la technique du **Golden Master Testing** sur un projet.
Elle est extraite de http://blog.adrianbolboaca.ro/2014/05/golden-master/

1. **Find the way the system delivers its outputs**
   Check for clear outputs of the system: console, data layer, logger, file system, etc.

2. **Find a way to capture the output of the system without changing the production code**
   Think if that output can be "stolen" without changing the production code. For example you could "steal" the console output by redirecting the stream of the console to an in-memory stream. Another example would be injecting an implementation of a data layer interface that would write to another data source than the production code would.

3. **Find a pattern of the outputs**
   The output of the system could be text or a data structures tree or another type of stream. Starting from this output type you can decide if you could go on to the next step.

4. **Generate enough random inputs and persist the tuple input/output**
   In the case you can persist the outputs to a simple data repository, you can think about what are the inputs for the corresponding outputs. Here the magic consists in finding a good balance of random or pseudo-random inputs. We need inputs that would cover most of the system with tests, but in the same time the test would run in seconds. For example in the case of a text generating algorithm we need to decide if we want to feed the system with 1000 or one million entry data. Maybe 10 000 input data are enough. We will anyway check the tests coverage during the last stage.
   We need to persist the pair input-output. The output would be the Golden Master, the reference data we will always check our SUT against.

5. **Write a system test to check the SUT against the previously persisted data**
   Now that we have a way to "steal" the outputs and we have a guess about how to generate enough input-output pairs, but not to many, we can call the system and check it against the outputs for a given input. We need to check if the test touches the SUT and if it passes. We also need to check that the test runs fast enough, in seconds.

6. **Commit the test**
   Always when a green test passes we need to commit the code to a local source control system. Why? So that we can easily revert to a stable testable system.
   **Important:** Do not forget to commit also the golden masters (files, database, etc)!

7. **Check test behaviour and coverage**
   In this stage I tend to do two things:
   - Use a test coverage tool to see where the system tests touch the SUT
   - Start changing the SUT in order to see the golden master test go red.

   If the test does not go red, I can understand that the code base is not covered by tests in that area and I should not touch it during the next stages, until I have a basic safety net. Always after this step I will revert to the previous commit, not matter how small the change to the SUT was.

8. **If not enough behaviours are covered, go to 3**
   In the case we found during the last stage some behaviours that were not covered by the golden master test, we need to write some more tests with other inputs and outputs. We go on until all the visible behaviours needing to be covered by tests are covered.

Pour mettre en place le **Golden Master** sur notre projet, nous allons donc suivre pas à pas ces différentes étapes :

➔ **Etape n°1 : Find the way the system delivers its outputs**
L'exercice précèdent nous a permis de constater que la sortie du système s'effectuait sur la console…

➔ **Etape n°2 : Find a way to capture the output of the system without changing the production code c-a-d récupérer les résultats d'une partie (`getGameResult`) pour être en mesure de créer le `GoldenMaster`**
D'une part, la sortie du système s'effectue directement sur la console, d'autre part les résultats ne sont pas déterministes. En effet, pour qu'une partie ne ressemble à aucune autre, un tirage aléatoire est utilisé dans le `main` pour simuler le lancer de dé. Ce tirage aléatoire est instancié par l'instruction `Random rand = new Random();` pour que le comportement diffère d'une partie à l'autre.

Dans cette étape, nous allons donc devoir traiter deux points :
- ➔ Faire en sorte que l'on puisse contrôler le générateur de nombre aléatoire pour « biaiser » le côté aléatoire du scénario de test et être en mesure de rejouer des parties aux comportements identiques
- ➔ Etre capable de rediriger l'affichage de la partie de la console vers un autre flux, plus facile à manipuler et à persister, dans le but de servir de référence (Golden Master) pour une comparaison ultérieure des comportements.

L'étape n°2 se focalise sur la création du **Golden Master** qui nécessite de pouvoir récupérer les résultats d'une partie donnée (pour pouvoir les comparer ensuite).
Créez dans **src/test/java**, une classe **`GoldenMaster`** que vous commencerez à implémenter de la manière suivante :

```
public class GoldenMaster {

      public String getGameResult(long seed) {
            GameRunner.play(new Random(seed));
      }
}
```

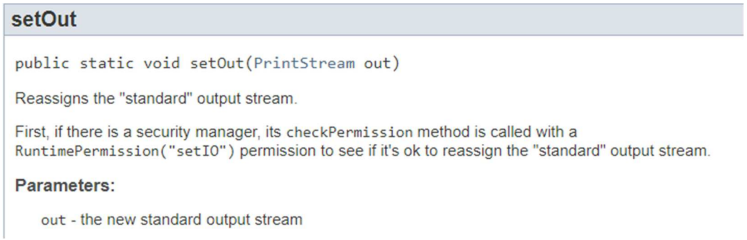
➔ **S'abstraire du caractère *aléatoire* de la partie :**
**(à propos de `play` et de `new Random(seed)`)**
Pour être sûr de jouer toujours la même partie (et comparer des comportements identiques), nous devons prendre le contrôle du générateur de nombre aléatoire.
Or pour l'instant, rien ne peut être paramétré dans le `main`.

- L'idée est donc d'extraire le comportement du `main` dans une méthode `play` qui permettra d'une part d'être facilement appelée en dehors de la classe `GameRunner`, d'autre part de « bouchonner » le caractère aléatoire de la partie en passant à cette méthode le générateur aléatoire que l'on souhaite.

- Le bouchonnage du générateur aléatoire consistera à choisir une valeur de `seed` donnée lors de l'instanciation du générateur.



➔ **Rediriger le flux et récupérer les informations affichées sur la console sans toucher au code de production.**
**(à propos de `setOut`)**



**setOut**

`public static void setOut(PrintStream out)`

Reassigns the "standard" output stream.

First, if there is a security manager, its `checkPermission` method is called with a `RuntimePermission("setIO")` permission to see if it's ok to reassign the "standard" output stream.

**Parameters:**

`out` - the new standard output stream

En Java, la méthode setOut permet de rediriger la console (« standard » output) vers un nouveau flux PrintStream. La classe PrintStream ajoute à un flux la possibilité de faire des écritures sous forme de texte des types primitifs java et des chaînes de caractères. Comme type de PrintStream, nous choisirons un flux très simple (`ByteArrayOutputStream`) pour accueillir les informations redirigées depuis la console.

Les trois instructions suivantes permettent donc de rediriger le contenu de la console vers le flux `consoleStream` de type **`ByteArrayOutputStream`.**

```
ByteArrayOutputStream consoleStream = new ByteArrayOutputStream();
PrintStream printStream = new PrintStream(consoleStream);
System.setOut(printStream);
```

Ajoutez ces trois instructions au début de la méthode `getGameResult` pour mettre en place la redirection de flux dans cette méthode.

Comme le flux `consoleStream` est aussi un PrintStream, pour accéder à son contenu, il suffit d'appeler `toString()`.
Ajoutez à la fin de la méthode `getGameResult` l'instruction suivante :

```
return consoleStream.toString();
```

➔ **Faire apparaître la méthode `play` dans le `GameRunner` :**
Pour que la méthode `getGameResult` puisse compiler, il faut maintenant créer la méthode `play` dans la classe `GameRunner`.
Extraire donc (à l'aide de l'IDE), le contenu de la méthode `main` dans une méthode `play` (hormis l'instruction relative à l'instanciation du `Random` puisque la méthode `play` est censée prendre en paramètre un `Random` 😉)
Remarque :
- Pour que la méthode `play` puisse être appelée dans la classe `GoldenMaster`, veillez à ce que cette méthode soit publique.
- `notAWinner` doit-elle être une variable globale ou locale ?

Votre code compile-t-il ? Si oui, vous pouvez continuer… 😉

➔ **Etape n°3 : Find a pattern of the outputs**

Cette étape consiste à choisir le format de sortie du GoldenMaster.
Nous choisissons **un fichier texte** comme format de sortie pour le Golden Master pour pouvoir comparer facilement le déroulement des parties.
Pour écrire facilement (et rapidement) un `String` (résultat de `getGameResult`) dans un fichier texte, nous utiliserons la bibliothèque **Apache Commons.io**

**Etape 3.1 : Mettre à jour votre `pom.xml`** en ajoutant la dépendance vers `commons-io`.
Relevez dans la rubrique ***Where can I get the latest release?*** du dépôt **https://github.com/apache/commons-io** les instructions maven correspondantes (instructions que vous retrouvez également sur https://commons.apache.org/proper/commons-io/dependency-info.html)

**Etape 3.2 : Générer un modèle de référence pour le Golden Master**
Cette étape doit permettre à la classe `GoldenMaster` d'offrir un nouveau service, celui de pouvoir générer un Golden Master.
Implémentez donc dans la classe `GoldenMaster` la méthode suivante :

```
public void generateGoldenMaster() throws IOException{
      FileUtils.writeStringToFile(new File("goldenMasterData.txt"),
                                                  getGameResult(0), "UTF-8");
          }
```

***Remarque :*** Cette instruction est inspirée de l'exemple de http://unil.im/commonsioexemple, en tenant compte du fait que `writeStringToFile` est **deprecated** et nécessite désormais un paramètre supplémentaire l'encodage (classe Charset)

`writeStringToFile(File file, String data)`
**Deprecated.**
2.5 use `writeStringToFile(File, String, Charset)` instead (and specify the appropriate encoding)

***Si vous souhaitez en savoir plus sur Apache* Commons.io ...**

- Un exemple pour mieux visualiser en quoi la bibliothèque **Commons.io** simplifie l'écrire d'un `String` dans un fichier texte : http://unil.im/commonsioexemple : 1 instruction avec `writeToStringFiles` d'Appache Commons.io, un certain nombre d'instructions avec un `FileWriter` classique 😉…
- La présentation de la bibliothèque Apache Commons.io est disponible ici : https://commons.apache.org/proper/commons-io/index.html

La javadoc sur la bibliothèque Apache Commons.io est disponible ici : https://commons.apache.org/proper/commons-io/javadocs/api-release/



Il ne reste plus qu'à vérifier que ce qui est généré dans le fichier texte est bien identique à ce qui était affiché précédemment dans la console…

**Etape 3.3 : Vérification le contenu du modèle généré**
Comme la méthode `generateGoldenMaster()` est destinée à être utilisée dans une classe de tests, nous allons procéder à cette vérification en demandant l'exécution de cette méthode directement dans une méthode de test et non dans un `main`.

Créez donc dans **src/test/java**, la classe **`GameTest`** (puisque nous cherchons à couvrir par les tests le comportement implémenté par la classe `Game`) et commencer à implémenter de la manière suivante :

```
import java.io.IOException;
import org.junit.Test;
import static org.junit.Assert.*;

public class GameTest {

      @Test
      public void test_goldenMaster() throws IOException {
            GoldenMaster goldenMaster = new GoldenMaster();
            goldenMaster.generateGoldenMaster();

            assertTrue(true);
      }

}
```

Compilez et lancez le test… Il doit passer AU VERT !!!
Rafraîchissez votre vue Package Explorer (via un F5 par exemple) , le fichier **`GoldenMasterData.txt`** devrait apparaître dans l'arborescence de votre projet **`trivia`**.
Double-cliquez pour visualiser le contenu de ce fichier.
N'oubliez pas refermer ce fichier avant de continuer 😉….

Vous êtes donc maintenant capable d'envoyer la sortie actuelle du système (l'affichage de la console) vers un fichier texte

Lancez la couverture de code. Quelle est-elle ?
C'est pour augmenter cette couverture de code que la partie suivante préconise de générer un certain nombre de parties pour disposer d'un certain nombre de scénarios différents.

➔ **Etape n°4 : Generate enough random inputs and persists the tuple input/output**

Le test précédent ne permet que de jouer une seule partie, mais les parties peuvent être plus ou moins longues, se comporter plus ou moins différemment et donc il se pourrait que des instructions non couvertes par ce test, le soit dans le cas d'une autre parti (donc d'un autre scénario de test)…

**Etape 4.1 :** Il faut donc commencer par **choisir le nombre de parties à tester**.
Il est souhaitable de disposer d'un assez grand nombre échantillons de parties pour être sûr que le comportement soit bien couvert.
Pour ce projet, nous pouvons essayer de commencer à écrire un test sur **1000 parties**
Il sera toujours possible d'ajuster ce chiffre plus tard si l'exécution des tests prend trop de temps ou la couverture reste insuffisante 😉

**Etape 4.2 :** Il faut ensuite faire en sorte de **générer un nombre aléatoire d'entrée et persister les couples d'entrée/sortie.**

- Pour générer un certain nombre de parties au comportement différent, il suffit de faire varier la graine du générateur pseudo-aléatoire (**seed**).
- Pour persister les couples entrées/sorties, il suffit :
  - pour chaque **entrée** (c-a-d chaque partie jouée avec un générateur pseudo-aléatoire dont la graine est **seed**) de nommer le **fichier de sortie** en reprenant la valeur d'entrée : **seed.txt**
  - puis de regrouper tous ces fichiers dans un répertoire GoldenMasterData (faire apparaître 1000 fichiers directement dans l'arborescence de votre projet ne serait pas le bienvenu, mieux vaut dédié au répertoire au stockage de ces fichiers 😉 …)

… ce qui nous amène à modifier la méthode generateGoldenMaster de la manière suivante :

```
public void generateGoldenMaster() throws IOException {
	for (long seed = 0; seed < 1000; seed++) {
		FileUtils.writeStringToFile(
					new File("goldenMasterData/" + seed + ".txt"),
					getGameResult(seed),
					"UTF-8");
	}
}
```

**Remarque : Ne pas oublier le /** après **GoldenMasterData** pour que les fichiers soient bien persistés dans ce répertoire spécifique et pas directement dans l'arborescence du projet !!!!

Compilez et lancez le test !

Ouvrez votre workspace **trivia** dans votre explorateur de fichier (clic droit sur le projet trivia dans la vue **Package Explorer** puis **Show In -> System Explorer**)

Constatez que vous disposez bien d'un dossier **goldenMasterData** qui contient un échantillon de **1000 fichiers** (de **0.txt** à **999.txt**) : c'est le Golden Master !

***Remarque :*** Relancez la couverture. Cette fois-ci toutes les questions sont couvertes.
Seul la méthode isPlayable ne semble pas couverte…Mais n'est-ce pas du code mort car il semblerait que ce méthode ne soit utilisée nulle part ailleurs 😉
La couverture de test (avec 1000 parties) nous convient et nous souhaitons donc conserver ce Golden Master comme référence.
Attention !!! Les instructions qui permettent de constituer le **GoldenMaster** ne doivent être effectuée qu'une seule fois.

**Pour éviter toute modification malencontreuse du Golden Master, supprimez donc maintenant la méthode test_goldenMaster de la classe GameTest (qui pour l'instant redevient temporairement vide)**



## ➔ Etape n°5 : Write a system test to check SUT against previously persisted data

- **Une fois le Golden Master généré, il faut être capable de le récupérer** 😉
  C'est ce que nous allons faire dans cette étape en mettant en place une nouvelle méthode getGoldenMaster dans la classe GoldenMaster qui permettra, pour une entrée donnée (seed), de récupérer dans un String la sortie associée (c-a-d le contenu du fichier seed.txt)
  Implémentez la méthode getGoldenMaster dans la classe GoldenMaster à partir de la méthode statique **readFileToString** de **org.apache.commons.io.FileUtils**

```
public class GoldenMaster {

	//…

	public String getGoldenMaster(long seed) throws IOException {
		return //… A vous d'écrire la bonne instruction !!! …
	}
}
```

- **Une fois le Golden Master récupéré, il ne reste plus qu'à l'utiliser dans une méthode de test** comme référence (c-a-d comme valeur attendue).
  Couvrir le code de la classe Game à l'aide du **Golden Master** revient donc à écrire un test qui doit comparer le comportement des parties générées par le code actuel du système (code que l'on va faire évoluer) au comportement des parties de référence enregistrées à l'étape précédente dans le **GoldenMaster**.
  Implémentez dans la classe GameTest la méthode test_checkTriviaAgainstGoldenMaster qui va jouer ce rôle…

```
public class GameTest {

	@Test
	public void test_checkTriviaAgainstGoldenMaster() throws IOException{
		GoldenMaster goldenMaster = new GoldenMaster();

		for (int seed = 0; seed < 1000; seed++) {
			String expected = goldenMaster.getGoldenMaster(seed);
			String actual = goldenMaster.getGameResult(seed);
			assertEquals(expected, actual);
		}
	}

}
```

Remarque : Le comportement des parties actuelles est capturée sous forme de String grâce à la méthode getGameResult.

Compilez ce code est exécutez les tests… Ils doivent passer AU VERT si vous avez correctement implémenté getGoldenMaster 😉

Avant de passer à l'étape suivante, il faut **s'assurer de la rapidité d'exécution du test faisant appel au Golden Master** pour éventuellement ajuster ou le nombre d'échantillons d'entrée/sortie.
Que pensez-vous du temps d'exécution du test `test_checkTriviaAgainstGoldenMaster` ?
Ce test s'exécute rapidement, nous pouvons donc conserver la comparaison sur **1000** échantillons.

➔ **Etape n°6 : Commit the test**
Il est temps de commiter votre travail avec un message de commit explicite du genre :
«`Harnais de tests sur Game avec un Golden Master de 1000 seed de 0 à 999`»

➔ **Etape n°7 : Check test behavior and Coverage**
**Dans cette étape, nous allons vérifier la pertinence et l'efficacité du test faisant appel au GoldenMaster.**

- Tout d'abord en terme de **couverture du code** …
  Lancez la couverture de code. Et sans surprise, comme nous l'avions déjà constaté dans l'étape 4, seule la méthode `isPlayable` ne semble pas couverte et semble être du code mort (obsolète, jamais utilisé).
  On peut donc supprimer cette méthode. Relancez les tests et la couverture…
  La couverture du code nous convient donc tout à fait.

- Ensuite en terme de **robustesse du Golden Master**…
  Pour tester la robustesse, nous allons provoquer volontairement quelques erreurs de comportement dans le SUT (System Under Test) afin de s'assurer que le Golden Master les détecte bien …
  - Par exemple dans le constructeur de `Game`, insérez volontairement une erreur dans l'instruction suivante :
    ```
    popQuestions.addLast("Pop Question " + i);
    ```
    en ajoutant un espace, ou un caractère dans la chaine `"Pop Question ".`
    Sauvegardez votre modification et relancez les tests.
    Si le Golden Master fonctionne correctement, les tests devraient maintenant passer AU ROUGE !!!
    Pour continuer, revenez dans la configuration initiale et assurez-vous que les tests passent bien AU VERT !!!

  - Essayons de modifier le code à un autre endroit, par exemple dans :
    ```
    int[] places = new int[6]
    ```
    ⇒ en remplaçant le 6 par un 5. Exécutez les tests. Que se passe-t-il ?
    ⇒ en remplaçant le 6 par un 4. Exécutez les tests. Que se passe-t-il ?
    ⇒ en remplaçant le 6 par un 3. Exécutez les tests. Que se passe-t-il ?
    ⇒ revenez dans la configuration initiale avec un 6.
    Nous remarquons que les tests passent AU VERT uniquement pour la valeur 3 alors qu'ils devraient être au ROUGE pour chaque modification.
    Pourquoi ? La partie que nous testions a été créée à partir de 3 joueurs. Or pour avoir un Golden Master plus robuste il faudrait aussi pouvoir paramétrer le nombre de joueurs lors de la génération du Golden Master…
    …ce qui nous emmène à la dernière étape …



➔ **Etape n°8 : If not enough behaviours are covered, go to 3**
Si les tests ne passent pas AU ROUGE dès qu'on change volontairement une valeur dans le système sous test, c'est le signe qu'il faut générer un nouveau Golden Master qui devra être en mesure de prendre en compte cette modification …
D'après le comportement observé lors du dernier point de l'étape précédente (modification du magic number `6`), il faudrait retourner à l'étape 3 pour générer un golden master offrant plus d'échantillons en terme de nombre de joueurs…
Nous ne le ferons pas maintenant, mais pour avoir un Golden Master efficace, cette étape serait indispensable 😉

Tiens d'ailleurs, en parlant de magic number.
Ne pourrait-on pas améliorer la qualité du code que nous venons d'écrire (classe `GoldenMaster` et `GameTest`). Une petite revue de code s'impose ….

➔ **Revue de code des classes `GoldenMaster` et `GameTest` et refactoring :**

- Identifiez les code smells suivants dans la classe **`GoldenMaster`** :
  - magic number
  - et duplication

  … et nettoyez les à l'aide de l'IDE…
- Faites disparaître le magic number de la classe **`GameTest`** en tenant compte de votre refactor précédent.

Les tests doivent bien sûr continuer de passer AU VERT !!!

➔ **Commiter**
….avec un message de commit explicite sur le refactoring que vous venez d'effectuer.

➔ **Pousser**
Si le cœur vous en dit, vous pouvez pousser sur votre dépôt distant 😉

Cet exercice nous a montré qu'il pouvait être un peu long et fastidieux d'écrire un Golden Master à la main, surtout les étapes autour de la manipulation des flux de sortie (fichiers).
Il existe une technique de tests **l'Approval Testing** qui peut grandement vous faciliter la vie dans la mise en place d'un golden master 😉

***Approval testing*** *is a test technique which compares the current output of your code with an "approved" version. The approved version is created by initially examining the test output and approving the result. You can revisit the approved version and easily update it when the requirements change. (Extrait* https://www.infoq.com/news/2017/02/approval-testing-texttest)

Pour mettre en place **l'approval testing**, il existe différents outils comme **Approval Test** (http://approvaltests.com/), **Text Test** (http://texttest.sourceforge.net/)...
Dans ce TP, nous utiliserons **Approval Test**… Mais avant de l'utiliser dans le projet **`trivia`**, découvrons cet outil au travers d'un petit exercice…

## Exercice 3 : Découverte d'Approval Test et de l'Approval Testing

Dans cet exercice, nous vous proposons de découvrir l'outil **Approval Test** et la démarche d'**approval testing** qui vous permettra de mettre en place rapidement un harnais de tests utilisant un Golden Master.

- ➔ Le site de référence d'Approval Test est: http://approvaltests.com/
- ➔ Le dépôt github de ce projet est : https://github.com/approvals
- ➔ La partie java est accessible sur : https://github.com/approvals/ApprovalTests.Java

### 1. Créez un nouveau projet

Dans votre IDE préféré, créez un nouveau projet **`helloapprovaltest`**.

### 2. Installation de l'outil Approval Test dans votre projet

- ➔ Rendez-vous dans la rubrique **Download** du dépôt https://github.com/approvals/ApprovalTests.Java et cliquez sur le lien donné qui vous redirigera vers https://github.com/approvals/ApprovalTests.Java/releases
- ➔ Téléchargez la dernière version de **`ApprovalTests.0xx.zip`**
- ➔ Ajoutez le jar **`ApprovalTests.jar`** dans le ***class path*** de votre projet.
  A partir d'un clic droit sur votre projet, sélectionnez **Properties**, puis **Java Build Path** puis **Add External Jar**. Recherchez ApprovalTest.jar dans l'archive que vous venez de télécharger puis **Ouvrir**, **Apply et OK** !

A partir de maintenant, vous devriez pouvoir utiliser **ApprovalTest** avec votre framework de tests préféré 😉

### 3. Ecrire un premier test avec Approval Test (`verify`)

Jusqu'à présent, vous écriviez un test unitaire en 3 étapes (pattern AAA)

```
@Test
public void uneMethodeDeTestClassique() {
	// Arrange
	// Act
	// Assert
}
```



Avec **ApprovalTest**, un test unitaire va s'écrire en seulement 2 étapes (**Do** et **Verify**) où l'étape de vérification sera gérée par l'outil **ApprovalTest** :

```
@Test
public void uneMethodeDeTestAvecApprovalTest() {
	// Do
	// Verify
}
```

Le premier test va permettre d'**approuver le Golden Master**, c'est-à-dire la référence : la(les) valeur(s) du Golden Master deviendront le(les) valeur(s) attendue(s) lors des prochaines assertions de tests exécutées lors des autres appels de **`verify`**.

Le premier exemple pour prendre en main **ApprovalTest** consiste à concaténer deux **`String`**. La règle de concaténation choisie est concaténer directement, sans espace ni autre caractère.

#### 3.1 Approuver le Golden Master (fichier `approved`)

Commencez donc par écrire ce premier test dans une classe **HelloApprovalTest** dans **src/test/java**.

```
import org.junit.Test;
import org.approvaltests.Approvals;

public class HelloApprovalTest {

	@Test
	public void test_concatenerDeuxStrings_enUnStringSansEspace() {

		// Do : procéder à la concaténation
		String string = "Approval";
		string += "Tests";

		// verifiy : verifier la concaténation
		Approvals.verify(string);
	}

}
```

Pour utiliser **ApprovalTest**, n'oubliez pas l'import : **`import`** `org.approvaltests.Approvals;`

Lancez ce test avec le runner de **JUnit**. Que constatez-vous ?

→ Le test passe au rouge.

→ Le test crée un fichier **`.received`** dans le même package que la classe de test. Le nom de ce fichier est construit de la manière suivante :
**`YourTestClass.yourTestMethod.received.txt`** (ou `.png`, `.html`, ...)

Rafraîchissez la vue Package Explorer pour voir apparaître le fichier :
**`HelloApprovalTest.test_concatenerDeuxStrings_enUnStringSansEspace.received`**

Double-cliquez sur ce fichier pour visualiser son contenu…
N'oubliez pas de refermer ce fichier avant de continuer…

Pour en faire le GoldenMaster c-a-d faire passer ce test et faire en sorte que ce contenu devienne la référence (valeur attendue : `expected`) lors des prochains tests, il faut renommer ce fichier de manière à ce que le **`received`** devienne **`approved`** :
`YourTestClass.youTestMethod.`**`approved`**`.txt`
Pour renommer ce fichier, vous pouvez choisir entre :

- renommer « à la main » le **`received`** en **`approved`**
- ou exécuter la ligne de commande **`move`** qui s'affiche dans la console

Une fois renommé, relancez le test avec le runner JUnit, cette fois-ci le test passe AU VERT !!!
Le contenu du fichier précédent constitue donc bien le Golden Master.

***Remarque :*** Il faudra peut-être faire un petit **`Project -> Clean`**… pour qu'**Infinest** repasse au vert…

### 3.2 Exécuter le test *contre* le Golden Master

Pour vous assurer que dorénavant quand ce test est lancé, la valeur attendue (`expected`) sera récupérée dans le Golden Master, nous allons faire volontairement échouer ce test.

Transformer par exemple l'instruction :

```
String string = "Approval";
```

en

```
String string = "Approval2";
```

Lancez le test, il passe au ROUGE !!!

Si vous lancez avec le runner JUnit, vous pouvez cliquer dans la vue **Failure Trace** sur l'onglet **Compare Actual With Expected Result** pour constater que :

- **`Expected`** contient **`ApprovalTest`**
- **`Actual`** contient **`Approval2Test`**

Remettez l'instruction dans son état initial, c-a-d :

```
String string = "Approval";
```

Lancez le test, il repasse au VERT !!!

Pour pouvoir *approuver* le Golden Master plus facilement, il est possible de s'appuyer sur un outil de diff (c-a-d un outil de comparaison et de fusion des fichiers) …



## 4. Ecrire un deuxième test avec Approval Test (approbation via un outil de diff)

### 4.1 Installer un outil de diff

La rubrique Supported Reporters du Getting Started indique la liste des outils de diff (comparaison et de fusion de fichiers) qu'**Approval Test** peut lancer.

Vous devez donc commencer par télécharger un outil de diff :
→ si vous êtes sous Windows, installez **Winmerge** : https://sourceforge.net/projects/winmerge/
→ si vous êtes sous Linux ou Max, installez **Meld** : http://meldmerge.org/

### 4.2 Approuver le Golden Master via l'outil de diff

Dans la classe **HelloApprovalTest,** écrire le test suivant :

```
@Test
public void test_StringBuilder() {

	// Do : Concaténation de deux objets de type StringBuilder
	StringBuilder sb = new StringBuilder();
	sb.append("Hello");
	sb.append("World");

	// Verifiy
	Approvals.verify(sb.toString());
}
```

Lancez ce test. Que se passe-t-il ?
L'outil de diff s'ouvre automatiquement avec :

- dans la **fenêtre de gauche**, le fichier **`HelloApprovalTest.test_StringBuilder.received`** qui contient **HelloWorld** (le résultat du test)
- dans la **fenêtre de droite**, le fichier **`HelloApprovalTest.test_StringBuilder.approved`** c-a-d **le Golden Master** qui pour l'instant de ne contient rien (puisque c'est la première exécution du test …)

**Pour approuver le Golden Master**, il suffit d'utiliser les options de l'outil de diff (comme *Copier vers la droite* sous **winmerge**) et enregistrer le fichier **`approved`**.

Revenez sous Eclipse, relancez le test, il passe AU VERT !!!!

### 4.3 Exécuter le test *contre* le Golden Master

Faire un petit changement, par exemple : `sb.append("World2");`
Relancez le test… Comme le test ne passe pas, l'outil de diff s'ouvre pour montrer d'où vient la différence… Ne touchez pas à ces fichiers…
Revenez dans la configuration initiale `sb.append("World");` et relancez les tests…
Les tests passent AU VERT !!!!

### 5. Ecrire un troisième test avec Approval Test (`verifyall` et outil de diff)

#### 5.1 Approuver un Golden Master composé de plusieurs échantillons

Dans la classe **HelloApprovalTest,** écrire le test suivant :

```
@Test
public void testArray() {

      // Do : create a String Array and set values in the indexes
      String[] s = new String[2];
      s[0] = "Approval";
      s[1] = "Tests";

      // Verify the array
      Approvals.verifyAll("Text", s);
}
```

Vous remarquerez que :

- pour vérifier **une seule valeur**, on fait appel à **`verify`** (cf. tests précédents)
- pour vérifier **un échantillon de valeurs**, on fait appel à **`verifyAll`**

Lancez ce test. Que se passe-t-il ?
L'outil de diff s'ouvre automatiquement.
Notez que le label `"Text",` premier paramètre de **`verifyAll,`** permet d'améliorer la lisibilité des valeurs de sortie dans le fichier **`HelloApprovalTest.testArray.received`**

**Approuvez** le Golden Master **via votre outil de diff**.

#### 5.2 Exécuter le test *contre* le Golden Master

Faire un petit changement dans le test et le relancez pour vérifier que le test s'effectue bien contre le Golden Master…
Remettez tous les tests AU VERT !!!

**Remarque :** Tous les fichiers **`approved`** et **`received`** devront bien sûr être soumis à votre contrôleur de version 😉



## Exercice 4 : Mise en place d'un harnais de test via un outil d'approval testing (`ApprovalTest`)

### 1. Mise en place du projet

Pour cet exercice vous allez travailler dans un nouveau projet.
Dans votre IDE préféré, créez un projet Maven **`triviaapprovaltest`** dans lequel :

→ vous importerez **les sources** disponibles dans le répertoire **ressources** **`trivia`** de ce dépôt : https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee.

→ vous ajouterez le **`ApprovalTests.jar`** dans le ***class path*** du projet (clic droit **`Properties`**→**`Java Build Path`** → **`Add External Jar…`** ou plus rapide clic droit **`Java Build Path`** → **`Add External Jar…`** ) pour pouvoir utiliser **ApprovalTest**

**Faites en sorte que ce code compile !**
**Mettez votre projet sous git et procédez à votre premier commit avec un message du genre :** « Commit initial : récupération du code legacy ».

### 2. Mise en place du harnais de test à partir d'un Golden Master et Approval Test

Fort(e) de votre expérience de l'exercice 2 et de l'exercice 3, mettez en place une couverture du code legacy de la classe `Game` à l'aide d'un Golden Master approuvé via **Approval Test**.

Contrairement à l'exercice 2, vous n'aurez besoin que d'une seule classe **`GameTest`** .
En effet, la classe `GodenMaster` est désormais inutile puisque l'outil **ApprovalTest** permet l'approbation du Golden Master et sa vérification.

Pour approuver le Golden Master :

- vous commencerez par approuver une seule partie pour vous familiariser avec **ApprovalTest** sur ce projet (**`verify`**). Comme dans l'exercice 2, vous devez commencer par extraire une méthode publique `play` dans la classe `GameRunner` pour extraire le comportement de la partie du `main` et utiliser une méthode privée `getGameResult` pour transcrire le comportement de la partie en `String` à transmettre à `verify` 😉

- puis vous transformerez votre code, pour au final pouvoir approuver un Golden Master sur un échantillon de 100 parties

**N'oubliez pas de commiter régulièrement et**
**de pousser votre travail sur le dépôt distant à la fin de l'exercice !**

**Pour les plus rapides …**

## Exercice 5 : Mise en place d'un harnais de test via une bibliothèque java d'approval testing (`approval`)

`approval` est une bibliothèque Java inspirée de **ApprovalTest** qui permet de faciliter la gestion du Golden Master et peut directement être ajoutée comme dépendance de votre projet.

➔ Le site de référence d'**`approval`** : http://approval.readthedocs.io/ ,
➔ Le dépôt github associé à ce projet est : https://github.com/nikolavp/approval

→ Créez un nouveau projet Maven **`triviaapproval`** avec **les sources** disponibles dans le répertoire **ressources** `trivia` de ce dépôt : https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee.
Versionnez votre projet.

→ **Mettez en place du harnais de test via un Golden Master et Approval Test**

Pour pouvoir utiliser **Approval** dans votre projet maven, vous devez ajouter la **dépendance suivante** à votre **`pom.xml`** :

```
<dependency>
      <groupId>com.github.nikolavp</groupId>
      <artifactId>approval-core</artifactId>
      <version>0.3</version>
</dependency>
```

Rendez-vous sur le dépôt github : https://github.com/nikolavp/approval pour connaître la `version` en cours. N'oubliez pas de remettre à jour votre **`pom.xml`**!!!

Fort(e) de vos expériences précédentes et de la rubrique Getting Started (http://approval.readthedocs.io/en/latest/getting-started.html) qui indique comment utiliser cette bibliothèque, mettez en place une couverture du code legacy de la classe `Game` à l'aide d'un Golden Master en vous appuyant sur la **bibliothèque Java `approval` .**

***Remarque :*** Avec la **bibliothèque `approval`**, il est possible d'indiquer le chemin du répertoire où on veut ranger le(s) fichier(s) contenant le Golden Master. Comme le montre cet exemple extrait de la rubrique **Approvals utility** du **getting started**

```
@Test
  public void testMyCoolThingReturnsProperString() {
      String result = MyCoolThing.getComplexMultilineString();
      Approvals.verify(result, Paths.get("src", "resources", "approval", "result.txt"));
  }
```

Notez que dans un projet Maven, de tels fichiers (comme d'éventuels fichiers images ou autres) sont habituellement stockés dans un package **src/resources** (à ajouter à l'arborescence de votre projet si nécessaire) : ici c'est le package **src/resources/approval** qui doit être créé.

**N'oubliez pas de commiter régulièrement et**
**de pousser votre travail sur le dépôt distant à la fin de l'exercice !**



# Annexe : Zoom sur l'inspection de code : SonarLint, SonarQube et SonarCloud…

**Le site officiel de Sonar est SonarQube (https://www.sonarqube.org/) :**

- **SonarLint** est un plug-in Sonar qui fonctionne uniquement dans l'IDE. Son principal objectif est de donner, au travers des issues (détection du respect ou non des règles de programmation), un feedback immédiat sur le code que vous êtes en train d'écrire.

- **SonarQube** est un serveur qui permet de réaliser **une analyse plus complète de votre code**. En plus des issues (règles de programmation) un serveur SonarQube est en capable d'identifier des duplications de code, de mesurer le niveau de documentation, de détecter des bugs potentiels, d'évaluation de la couverture de code par les tests unitaires, d'analyser de la répartition de la complexité cyclomatique, d'analyse du design et de l'architecture d'une application

  Il existe deux possibilités pour accéder à ces métriques liées à la qualité de votre code

  ➔ **Soit installer un serveur SonarQube en local sur votre machine**.
  Pour cela, vous pouvez suivre le *tutoriel Analyse d'un projet maven/Eclipse sur le serveur SonarQube* disponible sur :
  https://github.com/iblasquez/tutoriel_SonarQube/blob/master/Analyse_SonarQubeServer.md
  Veillez bien au préalable à vérifier si votre **JAVA_HOME** est connue de votre système !

  ➔ **Soit utiliser le service en ligne SonarCloud** (pas de serveur à installer).
  Pour utiliser la version en ligne rendez-vous sur : https://about.sonarcloud.io/ .
  La version gratuite en ligne de Sonar ne peut être utilisée que pour des projets open source (pour vous ce sera donc un projet public sous Github).
  Cliquez sur **Configure & Sign up** pour ouvrir la page https://about.sonarcloud.io/get-started/ et vous créez un compte (**Log in with Github** par exemple).
  Vous êtes alors redirigé vers https://sonarcloud.io/projects
  Cliquez sur le point d'interrogation en haut à droite à côté de votre profil et sur **Tutorials** puis **Analyze a new project** : ce tutoriel vous aidera à mettre en place une analyse Sonar de votre projet avec SonarCloud.

**Remarque : Liens autour de Sonar**
https://github.com/iblasquez/tutoriel_SonarQube/blob/master/Analyse_SonarLintEclipse.md
https://github.com/iblasquez/tutoriel_SonarQube/blob/master/Analyse_SonarQubeServer.md
https://stackoverflow.com/questions/46462540/how-to-configure-sonarcloud
https://maartenderaedemaeker.be/2017/07/16/getting-started-with-sonarqube-on-a-csharp-project/
https://blog.sonarsource.com/sonarqube-5-3-in-screenshots/
https://docs.sonarqube.org/display/SONAR/
https://github.com/SonarSource/sonarcloud_examples (sonar & travis ci)