

## Synthlab : Rapport de test

Auteurs :  
ANNE Aurélien,  
COISNARD-SIMON Marie,  
FORTUN Nicolas,  
LAURENT Julien,  
NOMANE Ahmed  
SALMON Kevin

7 mars 2018

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Technologies utilisées</b>	<b>1</b>
2.1	Phabricator : Relecture de code . . . . .	1
2.2	Jenkins : Intégration continue . . . . .	1
2.3	SonarQube : Indicateur de qualité . . . . .	2
<b>3</b>	<b>Tests unitaires</b>	<b>2</b>
3.1	Cobertura . . . . .	3
<b>4</b>	<b>Tests manuels</b>	<b>4</b>
<b>5</b>	<b>Annexes</b>	<b>5</b>

# 1 Introduction

Pour ce projet de synthétiseur numérique, nous avons réalisé une suite de tests permettant de vérifier le bon fonctionnement de notre application par rapport aux demandes du product owner.

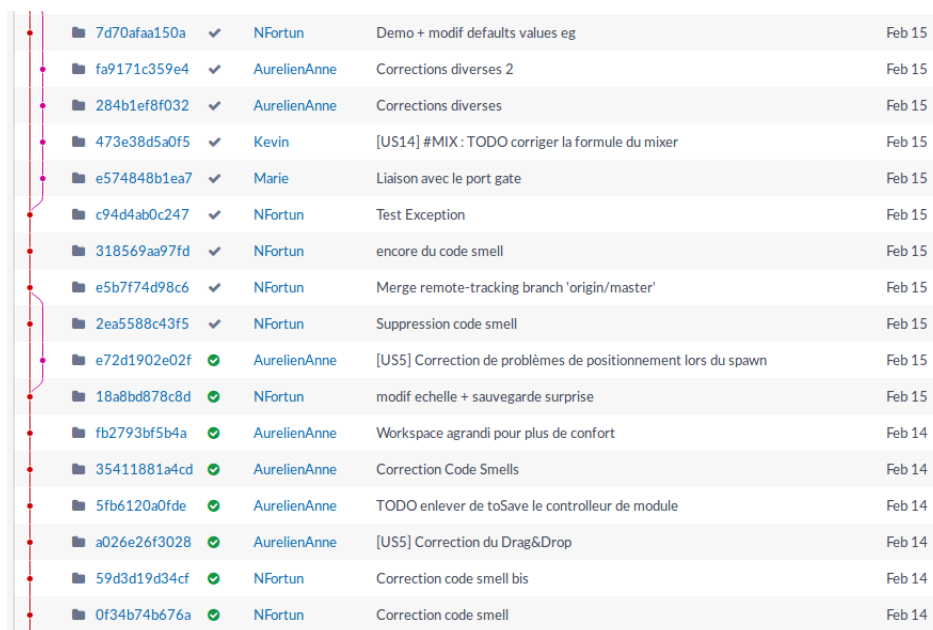
## 2 Technologies utilisées

Nous avons utilisé plusieurs technologies afin de produire un code de qualité, que ce soit localement ou sur notre serveur. Nous avons tout d'abord choisi Junit pour effectuer des tests unitaires. Ceci a pour but de vérifier le comportement des méthodes de chaque classe du projet. Ensuite, chaque membre du groupe utilise l'outil maven cobertura localement afin de vérifier la bonne compilation et visualiser la couverture de code du projet.

Enfin, nous avons déployé trois outils sur une VM de l'ISTIC avec Docker : Phabricator, Jenkins et SonarQube.

### 2.1 Phabricator : Relecture de code

Lorsqu'un membre de l'équipe commit, celui-ci est récupéré dans un outil de relecture de code : Phabricator. Chaque modification est donc relue par un membre de l'équipe. Les commits sont ensuite acceptés ou non par un autre membre. Il est possible de laisser des commentaires sur chaque ligne de code.



7d70afaa150a	✓	NFortun	Demo + modif defaults values eg	Feb 15
fa9171c359e4	✓	AurelienAnne	Corrections diverses 2	Feb 15
284b1ef8f032	✓	AurelienAnne	Corrections diverses	Feb 15
473e38d5a0f5	✓	Kevin	[US14] #MIX : TODO corriger la formule du mixer	Feb 15
e574848b1ea7	✓	Marie	Liaison avec le port gate	Feb 15
c94d4ab0c247	✓	NFortun	Test Exception	Feb 15
318569aa97fd	✓	NFortun	encore du code smell	Feb 15
e5b7f74d98c6	✓	NFortun	Merge remote-tracking branch 'origin/master'	Feb 15
2ea5588c43f5	✓	NFortun	Suppression code smell	Feb 15
e72d1902e02f	✓	AurelienAnne	[US5] Correction de problèmes de positionnement lors du spawn	Feb 15
18a8bd878c8d	✓	NFortun	modif echelle + sauvegarde surprise	Feb 15
fb2793bf5b4a	✓	AurelienAnne	Workspace agrandi pour plus de confort	Feb 14
35411881a4cd	✓	AurelienAnne	Correction Code Smells	Feb 14
5fb6120a0fde	✓	AurelienAnne	TODO enlever de toSave le controleur de module	Feb 14
a026e26f3028	✓	AurelienAnne	[US5] Correction du Drag&Drop	Feb 14
59d3d19d34cf	✓	NFortun	Correction code smell bis	Feb 14
0f34b74b676a	✓	NFortun	Correction code smell	Feb 14

FIGURE 2 – Extrait de la relecture de code sur Phabricator

### 2.2 Jenkins : Intégration continue

Jenkins, quant à lui, nous permet de vérifier que chaque commit envoyé sur Github fait bien compiler le projet et ne fait pas apparaître de régression. Après chaque commit, Jenkins lance une analyse SonarQube.



FIGURE 3 – Extrait de l'interface Jenkins

## 2.3 SonarQube : Indicateur de qualité

SonarQube nous permet de voir la couverture de test globale de notre projet. Il nous informe du pourcentage de code testé pour chaque classe, ainsi que le nombre de mauvaises pratiques, de duplications de code, de vulnérabilités ou de bugs.

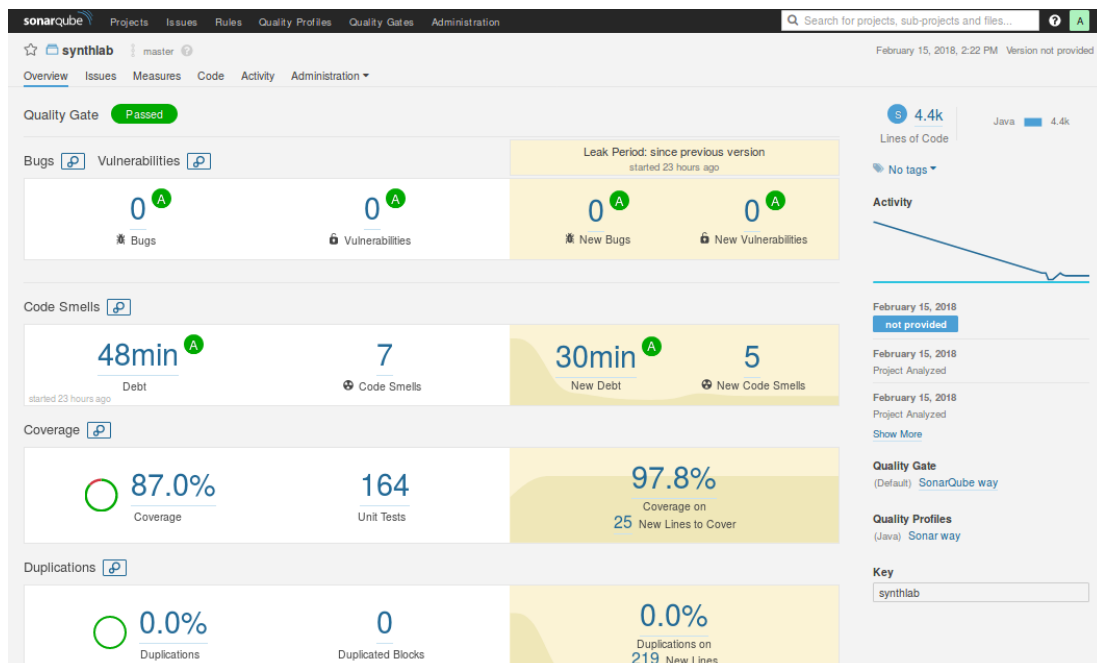


FIGURE 4 – Tableau de bord de SonarQube

## 3 Tests unitaires

Nous avons majoritairement concentré nos tests unitaires sur les différents modules, chacun ayant sa classe de test. Les tests ont été effectués dès que le développement du module était

terminé afin de trouver d'éventuels bugs. Nous n'avons pas attendu la fin du projet ou d'un sprint pour tout tester d'un coup, préférant tester au fur et à mesure.

Dans le cadre de notre projet, nous avons été amenés à vérifier et prouver le bon fonctionnement de notre programme. Pour cela, nous avons réalisé des tests de différentes natures sur certains éléments-clés du programme.

Après chaque création d'un nouveau module on lance des tests unitaires avec Junit, pour vérifier notre programme.

Nous avons exclu des tests unitaires tout ce qui concerne l'IHM. En effet, cette partie est dépendante de JavaFX. Nous savons qu'il est quand même possible de faire des tests, cependant nous avons jugé qu'il serait plus pertinent de formaliser des tests manuels, surtout au vu de la complexité de configurer JUnit afin de gérer un thread JavaFX.

Le but de ces tests est de vérifier le bon fonctionnement des méthodes, c'est-à-dire que chaque méthode fait ce qu'elle est censée faire, et qu'elle n'a pas de comportement inattendu. Au total, notre projet comporte plus de 165 tests unitaires.

La partie interface utilisateur de l'application n'a pas eu de tests autre que des tests manuels.

### 3.1 Cobertura

#### Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	78	34 %	801/2343	37 %	217/583	1,471
<a href="#">controller</a>	1	0 %	0/261	0 %	0/106	0
<a href="#">example</a>	3	0 %	0/103	0 %	0/36	1
<a href="#">exceptions</a>	2	50 %	2/4	N/A	N/A	1
<a href="#">filter</a>	1	100 %	12/12	100 %	2/2	1,333
<a href="#">ihm</a>	16	0 %	2/841	0 %	0/177	1
<a href="#">ihm.observer</a>	15	N/A	N/A	N/A	N/A	1
<a href="#">module</a>	17	92 %	660/717	91 %	199/218	1,82
<a href="#">sauvegarde</a>	14	0 %	0/228	N/A	N/A	1
<a href="#">utils</a>	9	70 %	125/177	36 %	16/44	2

FIGURE 5 – Couverture de test par Cobertura

Cobertura permet d'avoir un grain plus fin de la couverture de code par rapport à Sonar. Sur notre projet (fig. 5), on peut voir que les packages importants sont testés (exceptions, filter, module et utils). Les autres ne sont pas testés et ignorés par SonarQube car :

- controller : nous avons essayé de tester l'unique classe qu'il contient mais elle est très dépendante de JavaFX. À cause de l'utilisation d'un thread propre à JavaFX, il fallait préparer une classe spécial pour JUnit. De plus, lorsque les tests devaient passer sur Jenkins, l'absence d'écran et de système permettant de contourner le problème faisait échouer les tests. Face à la complexité de la mise en place des tests, nous avons préféré ne pas utiliser JUnit ;
- example : ce package contient des programmes principaux utilisés lors du développement. Il n'est pas nécessaire de les tester ;
- ihm : les contrôleurs JavaFX ne sont pas nécessairement testables et les interfaces utilisées par le pattern observer ne sont pas testables ;
- sauvegarde : il s'agit des memento utilisés pour la sauvegarde de l'espace de travail. Ce sont uniquement des POJO, donc il n'est pas nécessaire de les tester ;

## 4 Tests manuels

L'interface utilisateur ne peut pas être testée avec des tests unitaires. Par conséquent, nous avons opté pour des tests manuels, où un utilisateur lambda doit suivre différents scénarios d'utilisation. De plus, lors des sprint reviews, le product owner effectuait ses propres tests et nous donnait des retours.

Dans un premier temps, il nous fallait vérifier la bonne connexion des câbles entre les modules pour vérifier si le son émis ou l'affichage observé avec l'oscilloscope était celui attendu.

Les tests effectués ont révélé certains bugs plus ou moins importants. Cela nous a notamment permis de corriger un problème de rafraîchissement au niveau de l'oscilloscope.

## 5 Annexes

JUnit : <https://junit.org/junit5/>

SonarQube : <https://www.sonarqube.org/>