Rapport de test

SynthLab (Équipe 2)

Corentin Beaucé, Antoine Brioy, Johan Champion, Anthony Cobac, Pol Le Tue,
Salim Nadour & Jérémy Yziquel

Master 2 Génie Logiciel
Année 2015/2016

istic Informatique Électronique

Université de Rennes 1, Campus de Beaulieu
UFR Informatique et Électronique
263, avenue du Général Leclerc
CS 74205, 35 042 Rennes Cedex

Table des matières

Introduction

Dans le cadre de notre projet de Synthétiseur, 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. Lors de leurs réalisations, certains tests ont entraîné une modification du code pour corriger les erreurs. Pour tester au mieux le synthétiseur, différents types de tests ont été réalisés.

1 Tests

1.1 Tests fonctionnels

Les premiers tests réalisés étaient les tests fonctionnels. Dans notre cas, l'objectif premier était de brancher les nouveaux modules à des modules déjà opérationnels pour vérifier si le son émis ou l'affichage observé (avec l'oscilloscope) était celui attendu. L'interface utilisateur était utile pour vérifier le visuel et la connexion des modules.

1.2 Tests unitaires

À chaque nouveau module créé, des tests unitaires, avec le framework JUnit, furent implémentés pour avérer la conformité des modules. Les getters, les setters et les méthodes changeant les paramètres de la classe furent testés. Le but de ces tests est de vérifier, dans tous les cas, le bon résultat des méthodes. Certaines méthodes ont donc plusieurs cas de tests différents pour essayer toutes les possibilités.

Avec l'évolution du logiciel des tests unitaires ont été ajoutés, notamment avec l'utilisation du framework Mockito. Des Mock ont été utilisés pour remplacer des classes dans les tests et des Spy sont utilisés pour vérifier les bons appels de méthodes aux endroits critiques (les tests sur les ports principalement).

La partie interface utilisateur de l'application n'a pas eu de tests autres que les tests fonctionnels. Néanmoins, les composants JavaFX créés pour faciliter la création de l'interface et des modules (Knob, Cable, ...) ont été testés avec JUnit. Étant donné qu'il s'agit d'extensions aux classes JavaFX, ces classes ont besoin d'une règle JUnit pour fonctionner. Cette règle permet l'utilisation de composant JavaFX pour les tests sans avoir de Thread dédié.

1.3 Tests de mutation

Des tests par mutation ont été installés pour inspecter les tests déjà présents. Le but principal était de couvrir les zones critiques d'interfaçage entre notre application et la librairie utilisée, JSyn. En effet, la majorité de nos modules ont besoin d'un filtre qui implémente un UnitFilter ou bien un UnitGenerator de JSyn. Ces filtres doivent implémenter une méthode "generate()" qui va être, bien souvent, le coeur du module. L'analyse par mutation a donc permis d'améliorer les tests des modules.

Les mutants générés, par Pitest, pour le programme sont :

- Négation des conditions
- Mutation des conditions sur les limites
- Mutation des incrémentations
- Suppression des méthodes ne retournant rien
- Inversion des négations
- Mutation du type de retour
- Mutation des opérateurs mathématiques

2 Outils maven

Nous avons utilisé l'outil maven pour notre projet. Nous avons rajouté différents plugins pour nous aider lors des tests. Lorsque l'on utilise la commande `mvn site` plusieurs rapports sont générés.

Surefire

Le plugin maven Surefire génère un rapport sur les tests permettant de tous les vérifier rapidement.

JaCoCo

JaCoCo est outil de couverture de code. Il permet de vérifier facilement si passe au moins une fois dans toutes les lignes de toutes les méthodes.

Pitest

Pitest est un système de test par mutation. Il se lance avec la commande `mvn pitest:mutationCoverage`. Il permet vérifier nos tests.

Checkstyle

Un plugin maven générant un rapport en vérifiant le style général du code. Ce rapport permet ainsi de faire des corrections mineures, de facilité la lecture du code. Un fichier, `checkstyle-suppressions.xml`, supprime certaines règles de la vérification qui empêcherai le fonctionnement les règles étant un peu forte sur certain points notamment les magicNumbers, ou des règles qui aurai nécessiter une correction prenante en temps ou empêcher le fonctionnement du programme.

3 Résultats des tests

Le tableau suivant est un résumé de nos résultats sur les tests effectués.

Partie	Nb tests effectués	Nb tests qui passent	Couverture	Mutants éliminés
Modèle	273	273	85.2%	46.1%
Composant	90	90	52%	21%
IHM	0	0	0%	0%

La grande majorité des tests se concentre sur la partie Modèle puisque c'est cette partie qui fait l'interface avec la librairie JSyn. Une bonne partie des mutants survivants sont dus à des suppressions d'appel de fonctions retournant `void`.

Les tests effectués nous ont permis de corriger certains bug plus ou moins important. Certains problèmes se situaient aux niveaux des câbles, ou encore il y avait des problèmes au niveau des fichiers CSS lors du changement d'apparence de l'interface ou desboutons des modules.

On peut, bien sûr, améliorer les tests. Il faudrait vérifier comment marchent les modules sans l'intervention des Mocks. Il faudrait également rajouter des tests sur la partie IHM, avec la librairie TestFX et des outils similaires à Selenium.

Annexes

- **JUnit V4 :** `http://junit.org/`
- **Mockito :** `http://mockito.org/`
- **Pitest :** `http://pitest.org/`
- **JaCoCo :** `http://eclemma.org/jacoco/`
- **Surefire :** `https://maven.apache.org/surefire/maven-surefire-plugin/`
- **Checkstyle :** `http://checkstyle.sourceforge.net/`
- **Règle pour tester le JavaFX :**
  `http://andrewtill.blogspot.fr/2012/10/junit-rule-for-javafx-controller-testing.html`