

Rapport de test



Chakib Benkebir
June Benvegnu-Sallou
Antoine Ferey
Emmanuel Loisançe
Christophe Planchais
Youssef Roudani

<https://github.com/Kwodhan/SynthLabC>

La particularité de la réalisation des tests de ce projet était que nous ne savions pas si le son en sortie était correct.

Organisation du développement

La première étape pour assurer la qualité du code écrit et le bon fonctionnement de l'application a été de mettre en place un pair-programming presque systématique. En effet, lorsque deux développeurs codent ensemble une fonctionnalité sur le même ordinateur, le contrôle du code est immédiat : le risque de bug est diminué.

Nous avons aussi instauré une phase de tests "manuels" effectuée par des membres de l'équipe qui n'ont pas produit le code à tester. Il s'agissait de tester la fonctionnalité en manipulant l'application et réalisant différentes situations pour la mettre à l'épreuve.

Tests unitaires

Pour les tests unitaires, nous avons principalement utilisé JUnit. Dans certains cas, nous avons préféré utiliser des tests paramétrés. Lors du Sprint 1, nous avons commencé à écrire les tests unitaires portant sur les vérifications suivantes :

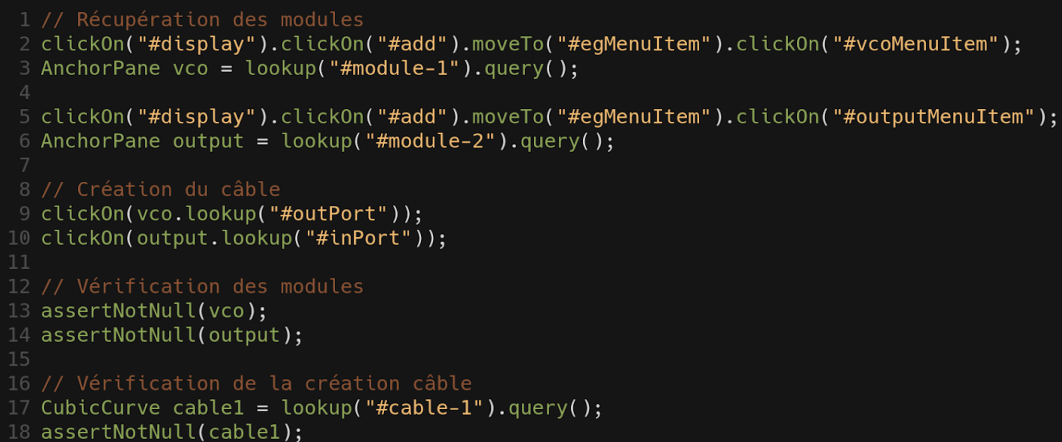
- Câblage entre deux ports.
- Bonne fréquence du VCO en fonction du réglage.

Nous avons vite réalisé que ces tests n'étaient pas très efficaces. Si nous avions continué dans cette voie, la couverture de code n'aurait pas été très importante. D'autre part comment savoir si les valeurs de nos variables reflètent réellement le son produit ? Nous avons donc décidé de compléter notre analyse par des tests IHM.

Tests IHM automatisés

Pour tester un maximum de situations, nous nous sommes orientés vers des tests IHM automatisés. Pour les tests IHM, nous avons décidé d'utiliser la bibliothèque *TestFX* qui permet de lancer l'application et de jouer un scénario graphique de façon automatique. Ces tests permettent de s'assurer au fil du temps que l'IHM reste fonctionnelle et que nous ne perdons pas en comportement (cas de régression).

TestFX fonctionne beaucoup à partir des IDs sur les différents éléments graphiques de l'interface pour retrouver les noeuds de l'application. A partir de ces noeuds, on peut demander au RobotFX de cliquer sur différents composants pour afficher ou reproduire le comportement souhaité.

A screenshot of a code editor with a dark background and light-colored text. The code is written in Java and uses TestFX for GUI testing. It includes comments in French and various TestFX methods like clickOn, lookup, moveTo, query, and assertions. The code is numbered from 1 to 18.

```
1 // Récupération des modules
2 clickOn("#display").clickOn("#add").moveTo("#egMenuItem").clickOn("#vcoMenuItem");
3 AnchorPane vco = lookup("#module-1").query();
4
5 clickOn("#display").clickOn("#add").moveTo("#egMenuItem").clickOn("#outputMenuItem");
6 AnchorPane output = lookup("#module-2").query();
7
8 // Création du câble
9 clickOn(vco.lookup("#outPort"));
10 clickOn(output.lookup("#inPort"));
11
12 // Vérification des modules
13 assertNotNull(vco);
14 assertNotNull(output);
15
16 // Vérification de la création câble
17 CubicCurve cable1 = lookup("#cable-1").query();
18 assertNotNull(cable1);
```

exemple d'utilisation de TestFX au sein d'un test IHM

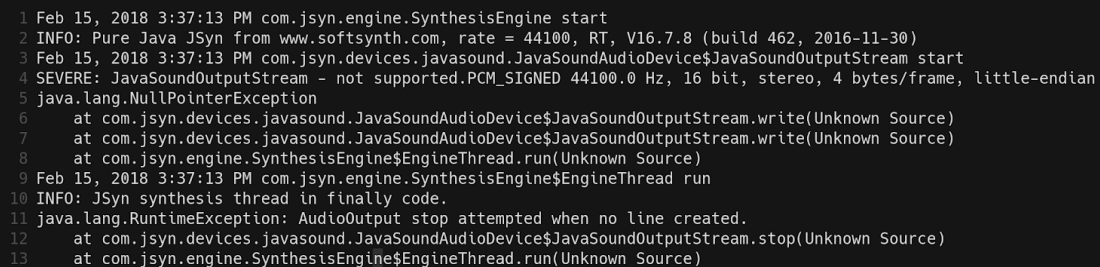
La prise en main de TestFX est simple et intuitive. Cela nous a donné la possibilité de nous focaliser sur l'écriture des scénarios de tests plutôt que sur la complexité du code. Les outils proposés par TestFX nous ont permis de reproduire tous les comportements qu'un utilisateur peut effectuer comme le clic d'une souris, le drag and drop, les entrées clavier, etc... Ainsi de nombreuses situations ont été testées, ainsi que des scénarios imprévisibles tels que la suppression de module alors que l'opération de câblage était en pleine exécution.

Outils d'aide au développement

Travis

Cet outil d'intégration continue permet de s'assurer après chaque commit qu'il n'y ait pas de problème pour builder l'application. L'outil nous permet aussi de réaliser les tests.

Nous nous sommes rendus compte tardivement que travis n'était pas capable de gérer nos tests IHM. En effet, nos tests IHM produisent du son via le module de sortie Output et Travis ne possède pas de sortie audio. JSyn renvoie donc une exception à Travis faisant échouer les tests.

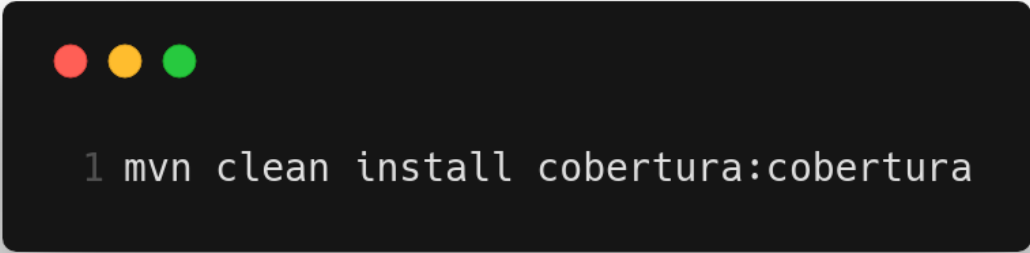


```
1 Feb 15, 2018 3:37:13 PM com.jsyn.engine.SynthesisEngine start
2 INFO: Pure Java JSyn from www.softsynth.com, rate = 44100, RT, V16.7.8 (build 462, 2016-11-30)
3 Feb 15, 2018 3:37:13 PM com.jsyn.devices.javasound.JavaSoundAudioDevice$JavaSoundOutputStream start
4 SEVERE: JavaSoundOutputStream - not supported.PCM_SIGNED 44100.0 Hz, 16 bit, stereo, 4 bytes/frame, little-endian
5 java.lang.NullPointerException
6   at com.jsyn.devices.javasound.JavaSoundAudioDevice$JavaSoundOutputStream.write(Unknown Source)
7   at com.jsyn.devices.javasound.JavaSoundAudioDevice$JavaSoundOutputStream.write(Unknown Source)
8   at com.jsyn.engine.SynthesisEngine$EngineThread.run(Unknown Source)
9 Feb 15, 2018 3:37:13 PM com.jsyn.engine.SynthesisEngine$EngineThread run
10 INFO: JSyn synthesis thread in finally code.
11 java.lang.RuntimeException: AudioOutput stop attempted when no line created.
12   at com.jsyn.devices.javasound.JavaSoundAudioDevice$JavaSoundOutputStream.stop(Unknown Source)
13   at com.jsyn.engine.SynthesisEngine$EngineThread.run(Unknown Source)
```

exception dans travis

Cobertura

Pour récupérer le rapport cobertura sur la couverture de code de notre application, nous devons nous-même lancer la commande cobertura et attendre l'exécution de l'ensemble des tests. Ce processus prenait de plus en plus de temps au fur et à mesure de l'avancée du projet. De plus, cela bloquait l'ordinateur d'un des développeurs pendant le temps d'exécution, il en profitait donc pour faire du pair programming ou avancer sur des questions techniques.



```
1 mvn clean install cobertura:cobertura
```

Ligne de commande pour générer le rapport cobertura

Le rapport cobertura nous donne un résultat de 94% de couverture du code sur l'ensemble de notre application. En le générant régulièrement pendant les développements et la réalisation des tests, cela nous a permis de voir l'évolution de la couverture de code. En vérifiant pour chaque fichier l'état de la couverture de code, nous avons pu ajouter des tests afin de couvrir quasiment l'ensemble du code l'application.

| Package | # Classes | Line Coverage |
|---|-----------|----------------|
| All Packages | 56 | 94 % 1987/2100 |
| com.istic | 1 | 0 % 0/14 |
| com.istic.cable | 2 | 91 % 115/126 |
| com.istic.eq | 1 | 100 % 16/16 |
| com.istic.fileformat | 1 | 82 % 19/23 |
| com.istic.keyboard | 2 | 97 % 159/163 |
| com.istic.mixer | 2 | 100 % 67/67 |
| com.istic.modulesController | 15 | 93 % 1044/1114 |
| com.istic.oscillo | 2 | 100 % 31/31 |
| com.istic.out | 1 | 88 % 32/36 |
| com.istic.port | 13 | 100 % 100/100 |
| com.istic.rep | 2 | 100 % 34/34 |
| com.istic.sequencer | 1 | 100 % 25/25 |
| com.istic.util | 4 | 96 % 186/192 |
| com.istic.vca | 2 | 100 % 33/33 |
| com.istic.vcfhp | 2 | 100 % 32/32 |
| com.istic.vcflp | 2 | 100 % 38/38 |
| com.istic.vco | 2 | 100 % 51/51 |
| com.istic.whitenoise | 1 | 100 % 5/5 |

rapport de couverture de code Cobertura

Cependant, il est difficile d'arriver à une couverture de code de 100%, en effet certains cas sont compliqués à reproduire avec un test IHM automatisé :

- Les "try/catch" présents à certains endroits dans le code de l'application.
- Vérifier qu'il n'est pas possible de connecter deux câbles sur un seul et même port. Ce cas était impossible à reproduire. En effet, lors du clic simulé par TestFX sur le port déjà connecté, il supprimait le câble déjà présent plutôt que d'essayer d'en rajouter un second.

D'autres cas ont pu être testés en utilisant au maximum les fonctionnalités de TestFX. Pour la sauvegarde d'un son, la sauvegarde ou l'ouverture d'une configuration, la fenêtre de sélection du fichier (nom du fichier, chemin, et validation) était hors de notre application. Il était donc impossible de se baser sur les IDs pour cliquer/sélectionner les éléments de la fenêtre. En utilisant la fonction "type" de TestFX qui permet de presser une touche à partir de son "KeyCode", nous avons réussi à écrire des scénarios permettant de couvrir l'ensemble de ces cas.

| Classes in this Package | Line Coverage |
|-------------------------|---------------|
| App | 0 % 0/14 |

Le 0% correspond à la seule classe présente dans le package “com.istic” qui est la classe contenant le “main” qui permet de charger le FXML et de démarrer l’application. En passant par les tests IHM avec TestFX, nous utilisons une autre méthode permettant de charger la scène de l’application sans passer par la classe App.

```

1 public class IHMSprint3Test extends ApplicationTest {
2
3     @Override
4     public void start (Stage stage) throws Exception {
5         Parent mainNode = FXMLLoader.load(App.class.getResource("../main.fxml"));
6         stage.setScene(new Scene(mainNode));
7         stage.show();
8         stage.toFront();
9     }
10
11     ...
12 }

```

chargement de la scène dans un TestFX

Conclusion

Au final, l’écriture des scénarios de tests nous aura pris beaucoup de temps sur l’ensemble du temps alloué au projet mais cet investissement de temps se révèle utile. En effet, ces tests nous ont permis de vérifier que le code ajouté pendant les développements n’introduisait pas de régression dans notre code.

L’intérêt de nos tests aurait été encore plus important si Travis avait été capable d’exécuter les tests IHM de TestFX pour valider le bon fonctionnement des scénarios après chaque commit.

Nous avons souvent fait appel au Product Owner pour avoir un avis sur le son produit par les montages. Ce fût plus facile à partir du moment où notre oscilloscope est devenu fonctionnel.