

Jonathan Racaud

Music Sheet Writer

Music Sheet Writer

Stratégie de Tests

Objectifs du document

Résumé

Glossaire

***– C –***

CPU : Un processeur (ou unité centrale de traitement, UCT, en anglais central processing unit, CPU) est un composant présent dans de nombreux dispositifs électroniques qui exécute les [instructions machine](https://fr.wikipedia.org/wiki/Instruction_machine) des [programmes informatiques](https://fr.wikipedia.org/wiki/Programmes_informatiques).

CocoaPod :

***– K –***

***– S –***

Stress test: Un test de résistance (stress testing en anglais) est une technique issue de l’[ingénierie](https://fr.wikipedia.org/wiki/Ing%C3%A9nierie). Elle est destinée à évaluer la stabilité d’un produit, d’une entité ou d’un système dans des conditions de fonctionnement sévères ou inhabituelles.

***– X –***

Xcode : Xcode est un environnement de développement pour [Mac OS X](https://fr.wikipedia.org/wiki/Mac_OS_X) ainsi que pour [iOS](https://fr.wikipedia.org/wiki/IOS_(Apple)).

Xcpretty : c’est un framework capable de rendre plus lisible et clair la sortie des tests pour pouvoir en créer des livrables.

XCTest :

***– M –***

Mock : sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée. On utilise un mock dans le but de tester le comportement d'autres objets.

***– R –***

Release : Mise à disposition d’une version de l’application. Elle peut être privée, semi publique ou publique. En général une release fait état de la version commerciale (ou plus détaillée pour des releases fermées).

Description du document

|  |  |
| --- | --- |
| Titre | Music Sheet Writer : Stratégie de Tests |
| Date de création | 22/10/2015 |
| Date de publication | 15/11/2015 |
| Auteur | Jonathan Racaud |
| Responsable | Jonathan Racaud |
| E-mail | musicsheetwriter\_2017@labeip.epitech.eu |
| Sujet | Stratégie de Tests |
| Version du modèle | 1.0 |

Tableau des révisions

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Auteur** | **Section(s)** | **Commentaire** |
| 22/10/2015 | Jeremy HARRAULT | Toutes | Création du document |
| 29/10/2015 | Antoine SIMON | Toutes | Rédaction |
| 10/11/2015 | Antoine SIMON | Toutes | Correction suite aux remarques |

Sommaire

[1. Introduction 1](#_Toc433295550)

[1.1. Objectif de l’EIP 1](#_Toc433295551)

[1.2. Principe de base du système 1](#_Toc433295552)

[1.3. Documents de référence 1](#_Toc433295553)

[2. Version/release 2](#_Toc433295554)

[3. Types de tests et périmètre 3](#_Toc433295555)

[4. Livrables 4](#_Toc433295556)

[4.1. Cas de tests 4](#_Toc433295557)

[4.2. Rapport d’exécution de tests 4](#_Toc433295558)

[4.3. Rapport sur le suivi global de la qualité 4](#_Toc433295559)

[5. Types de tests 5](#_Toc433295560)

[5.1. Tests XXX-1 5](#_Toc433295561)

[5.2. Tests XXX-2 5](#_Toc433295562)

[6. Estimation Globales de la charge de travail 6](#_Toc433295563)

[7. Outils 7](#_Toc433295564)

[7.1. Dépôt de tests 7](#_Toc433295565)

[7.2. Gestion de rejets/erreurs 7](#_Toc433295566)

[7.3. Autres outils 7](#_Toc433295567)

# Introduction

## Objectif de l’EIP

## Principe de base du système

## Documents de référence

# Version/release

2.1 Iphone

2.1.1 plan de release

Voici la liste des releases majeurs de l’application Iphone. Leurs numéros de version seront notés sous la forme X.0 où X est le numéro de la release majeur. Ce nombre sera incrémenté de 1 à chaque release.

D’autres releases mineures pourront être faites si :

* Une fonctionnalité rapide à implémenter est fortement demandée par la communauté et/ou devient urgent à implémenter pour satisfaire l’expérience MusicSheetWriter.
* Des « bugs » sont trouvés sur l’application, soit par l’équipe de développement, soit pas la communauté. Ces bugs pourront être remontés aux développeurs via la fonctionnalité de report de bug intégré de base au système Android.

Une release mineure sera notée sous la forme X.Y où X est le numéro de la release majeure en cours et Y est le numéro de la release mineure.

2.1.2 Plan des releases majeurs

|  |  |  |  |
| --- | --- | --- | --- |
| Numéro de version | Date de sortie prévue | Description | Fonctionnalités ajoutées |
| 1.0 | 4/08/2016 | Cette version sera la première version de l’application IOS. Elle sortira en même temps que la première version du logiciel. Aucune version n’est prévus avant car la gestion des partitions ne sera pas utilisé sans le logiciel et l’aspect communautaire ne sera pas assez poussez pour se suffire à lui seul. | * Création et connexion à un compte utilisateur * Gestion des informations personnelles * Consultation des partitions de l’utilisateur * Gestion des partitions favorites de l’utilisateur * Gestion des abonnements de l’utilisateur * Recherche d’utilisateur * Recherche de partition * Lecture sonore d’une partition |
| 2.0 | 30/03/2017 | Cette version sera tournée vers les fonctionnalités communautaires et commencera son approche vers l’implémentation des fonctionnalités d’édition de partitions. | * Correction de bugs * Commenter les partitions * Envoie de message à d’autre utilisateur * Lecture d’une partition * Téléchargement d’une partition depuis le serveur sous format PDF |
| 3.0 | 30/09/2017 | Cette version rapprochera les fonctionnalités de lecture et d’édition de partition du logiciel et de l’application IOS. | * Correction de bugs * Création d’un projet MusicSheetWriter * Edition « note-à-note » d’une partition * Téléchargement d’une partition vers et depuis le serveur sous forme de projet MusicSheetWriter |

# Types de tests et périmètre

3.1 IPhone

Les tests de de l’iPhone se découperont en 2 parties majeur :

* Les tests de l’interface.
* Les tests de toutes les méthodes et du code qui peuvent être testées.

Les premiers tests sont ceux de l’interface. Ils sont essentiels pour vérifier certaines fonctions de l’application qui ne peuvent pas être vérifiées autrement que par des actions de l’utilisateur.

Chaque test et construit pour tester chaque partie de l’application, comme les menus ou le changement de code.

Des *Mocks* sont utilisés pour les tests de l’interface. Ce sont des objets permettant de simuler les actions d’un utilisateur. Certaines erreurs ne venant pas du code mais du *Main.Storyboard* (c’est le fichier qui permet la création et le management de toute l’interface d’une application IPhone si l’on utilise Xcode) ne peuvent être visibles que par cette méthode. Chaque test est écris après la création de l’interface. Ces tests sont donc là pour vérifier la bonne conduite de l’interface malgré une maintenance.

Par la même occasion la vérification des données hardware comme la mémoire ou le *CPU* sont disponible et devront être lu par le développeur lors des phases de tests de *l’UI*. Cela peut être vu comme un *stress test*.

De plus viendront se greffer sur les tests, des vrais tests de performance.

De plus le framework *xcpretty* vient compléter *xcodebuilder*, la solution pour lancer xcode en ligne de commande*,* pour permettre un export clair et détaillé des tests.

## IPhone

### Test unitaire

Ces tests vérifieront le bon fonctionnement de classes ou méthode particulière. Ils seront effectués hors d’un contexte spécifique à l’application IOS MusicSheetWriter. En effet, ils ne nécessitent pas d’élément visuel particulier. Ce type de test sera notamment pour les classes et méthodes utilitaires et susceptible d’être réutilisé dans d’autres projets comme les classes d’appel API, les classes de validation de formulaire ou encore d’authentification.

Les tests d’appel à API vérifieront le bon fonctionnement des actions nécessitant ou incluant l’envoie de requête à l’API MusicSheetWriter suite à une action utilisateur particulière ainsi que les conséquences sur l’interface utilisateurs de chacune des réponses possible renvoyée par le serveur. Chaque activité et fragment contenant un appel à l’API sera soumis à des tests d’appel à l’API. C’est test ne vérifierons pas le contenu de la réponse de l’API mais seulement

### Test d’intégration

Ces tests vérifient le bon fonctionnement des classes et méthode couplé avec l’Interface utilisateur de l’application. Les activités et fragments concernés par ces tests seront testés indépendamment les uns des autres. Ces tests seront de deux types différents :

#### Test composant UI

Ils sont essentiels pour vérifier certaines fonctions de l’application qui ne peuvent pas être vérifiées autrement que par des actions de l’utilisateur.

Chaque test et construit pour tester chaque partie de l’application, comme les menus ou le changement de code.

Des *Mocks* sont utilisés pour les tests de l’interface. Ce sont des objets permettant de simuler les actions d’un utilisateur. Certaines erreurs ne venant pas du code mais du *Main.Storyboard* (c’est le fichier qui permet la création et le management de toute l’interface d’une application IPhone si l’on utilise Xcode) ne peuvent être visibles que par cette méthode. Chaque test est écris après la création de l’interface. Ces tests sont donc là pour vérifier la bonne conduite de l’interface malgré une maintenance.

### Test de non-régression

Ces tests seront établis à partir des tests unitaires et d’intégration et permettront de vérifier l’intégration d’une nouvelle partie du code ou d’une modification du code existant en s’assurant que les fonctionnalités de l’application sont toujours valides.

### Test d’acceptation

Ces tests vérifieront le bon fonctionnement de certains scénario prédéfinit et couvrant l’ensemble des fonctionnalités de l’application. A la différence des tests des tests d’intégrations, ces tests vérifieront l’application dans son ensemble, et pas les activités et fragments de manière unitaire.

# Livrables

## Cas de tests

4.1.1 Iphone

Les cas de tests pour l’IPhone seront chaque nouvelle implémentation de fonctionnalité (le login, la création de compte, ..). Ce système permet une fréquence de mise à jour des tests corrects.

#### Tests Unitaire

|  |  |
| --- | --- |
| *Chaque test comportera…* | *Afin de…* |
| Une courte description présentant le test | Connaitre l’élément testé |
| Les données utilisées en temps qu’entré pour le test | Savoir quelle valeur sont testés et par substitution, lesquels ne le sont pas |
| Les conditions à succès | Connaitre quelle et comment est la sortie du test |
| La durée approximative du test | Connaitre le temps nécessaire à son exécution normale. |

#### Test Intégration et d’acceptation

|  |  |
| --- | --- |
| *Chaque test comportera…* | *Afin de…* |
| Une courte description présentant le test | Connaitre la fonctionnalité testé et comprendre le contexte dans lequel se trouve le user s’il devait faire le test manuellement |
| Les données utilisées en temps qu’entré pour le test | Savoir quelles valeurs sont utilisées pour le test |
| Les conditions à succès | Connaitre quelle et comment est la sortie du test |
| Le type de scénario qu’il simule (voir ci-dessous) | Connaitre la nature du test |
| La durée approximative du test | Connaitre le temps nécessaire à son exécution normale. |

Le type de scénario peut être :

* Basic : c’est le flow d’exécution tel qu’il est censé se produire en condition d’utilisation normale
* Alternative : c’est le flow d’exécution tel qu’il est censé se produire en condition d’utilisateur particulière
* Exception : c’est le flow d’exécution tel qu’il est censé se produire en condition d’erreur

Le livrable des cas de tests devra être mis à jour aussitôt qu’une campagne de test est terminé d’être codé, même si tous les tests ne sont pas en succès.

## Rapport d’exécution de tests

Il y a deux formes de rapport :

* Avec export
* Sans export

La version sans export est celle lancé directement depuis *xcode*. Les résultats sont clairs à la fin de l’exécution et si une erreur se produit, la ligne d’où provient l’erreur est clairement mise en valeur avec une explication de l’erreur.

Le problème est qu’il n’y a aucun export et donc après fermeture de *xcode* aucune preuve.

Pour le test avec export, il faut lancer depuis un terminal en ligne de commande. Le seul problème est qu’on ne peut pas lancer de test unique, ils seront tous exécutés mais en générant un rapport clair pour prouver le bon fonctionnement de l’application.

Voici Un exemple d’export possible suite à la réussite de tous les tests :

▸ Building MSW/MSW2 [Debug]

▸ Check Dependencies

▸ Building Pods/KIF [Debug]

▸ Check Dependencies

▸ Building Pods/Pods-MSW2Tests [Debug]

▸ Check Dependencies

▸ Building MSW/MSW2Tests [Debug]

▸ Check Dependencies

▸ Running script 'Check Pods Manifest.lock'

▸ Running script 'Embed Pods Frameworks'

▸ Running script 'Copy Pods Resources'

All tests

Test Suite MSW2Tests.xctest started

LogTest

✓ test00LogFail (3.399 seconds)

✓ test01MotDePasseOublie (8.548 seconds)

✓ test02Inscription (7.422 seconds)

✓ test03Log (3.926 seconds)

✓ test04Deconnexion (5.903 seconds)

✓ test05ChangerDeMotPasse (14.272 seconds)

✓ test06ChangeFirstname (5.414 seconds)

✓ test07Menu (6.868 seconds)

✓ test08Swipe (5.708 seconds)

✓ test09SwipeMenu (2.007 seconds)

✓ test10Ajouter (5.572 seconds)

✓ test11 (0.000 seconds)

LoginTest

✓ testConnexion (2.451 seconds)

✓ testConnexionfausse (1.171 seconds)

✓ testGetMethodWithId (0.687 seconds)

✓ testGetMethodWithString (0.704 seconds)

✓ testOptionsMethodWithString (0.678 seconds)

✓ testPostMethodForSuscribeWithUsername (0.778 seconds)

✓ testPostMethodWithIdentifier (1.185 seconds)

✓ testPostMethodWithString (1.170 seconds)

✓ testPutMethodWithString (0.740 seconds)

testApiMethod

✓ testGetMethodWithId (0.697 seconds)

✓ testPostMethodForSuscribeWithUsername (0.707 seconds)

✓ testPostMethodWithIdentifier (1.173 seconds)

✓ testPostMethodWithString (1.170 seconds)

Executed 25 tests, with 0 failures (0 unexpected) in 82.353 (82.365) seconds

## Rapport sur le suivi global de la qualité

# Types de tests

## Tests Unitaire

### Objectif

L’objectif de ce test est très simple, il doit simplement vérifier que chaque méthode de la classe API se lance correctement et s’exécute. Si la méthode retourne une erreur cela n’est pas grave, cela veut dire malgré tout qu’il y a bien une communication avec l’API et que la méthode est donc fonctionnelle. De plus il y a des tests sur la globalité du code, comme un test de connexion réussi, un test de connexion raté, le retour de certaine fonction, …

### Environnement et conditions de réalisation

Ce test peut être réalisé dans n’importe quel cas ou contexte. Il est bien de le lancer quand l’API est modifiée ou que l’on doute sur son bon fonctionnement. Il ne vérifie en rien l’interface de l’application.

### Configurations particulières

Aucune configuration particulière n’est requise pour ce test.

### Planning et charge

Les tests unitaires peuvent être écrits à la suite de la spécification de la ou des classes qu’ils testent. La charge de travail dépend de ce qui est testé

### Critères de passage/échec

Xcode indiquera le passage ou non du test grâce à une coche verte ou une crois rouge. Il faut passer tous les tests pour valider cette partie. Il faut donc qu’aucune méthode ne retourne *NULL* ou ne vienne à répondre incorrectement.

## Tests Interface

### Objectif

L’objectif de ce test est de vérifier la partie interface de l’application. Certaine partie n’étant pas détectable via le code, la création de *mocks* fut donc la meilleur solution. Ces tests servirons à mettre en évidence les problèmes lié à l’interface mais par la même occasion de tester à nouveau une partie des fonctionnalités de l’application.

### Environnement et conditions de réalisation

Les tests seront effectués à des moments clé de chaque étape. Dès qu’une nouvelle fonctionnalité est produite elle se doit d’être testée. Par exemple si le login vient d’être implémentée, il doit être testé grâce à une *Mock* (mais aussi par un test unitaire).

Tous les tests se feront via *Xcode* qui contient une interface claire de test et seront aussi lancé grâce à *xcodebuilder* pour permettre un export des tests.

### Configurations particulières

Pour réaliser ce test il faudra avoir une interface terminé. De plus Il faudra aucun abonnement à part « bob » pour le compte « bob » et le mot de passe de ce dernier devra être « b ». Si cela venez à changer le test actuel ne pourrait être exécuté et une partie du code sera à changer.

### Planning et charge

Les tests unitaires peuvent être écrits à la suite de la spécification de la ou des classes qu’ils testent. La charge de travail dépend de ce qui est testé

### Critère de démarrage des tests

Pour démarrer ce test l’application doit être fermé et compiler sur le téléphone ou simulateur.

### Critères de passage/échec

Il y a plusieurs critères de passage pour chaque partie de ce test. IL faut que le temps d’exécution soit inférieur à 10 secondes pour chaque simulation de l’utilisateur et l’exécution des commandes faites dans le code doivent correspondre à ce qui s’affiche.

# Estimation Globales de la charge de travail

# Outils

IPhone

## Dépôt de tests

## Gestion de rejets/erreurs

Chaque tests grâce à la ligne de commande créer un document directement enregistrer dans le dépôt de l’application. Il suffira de l’analyser pour connaitre les erreurs et d’avoir de l’aide pour les régler.

Sinon *Xcode* contient une interface claire avec un soulignement de la ligne ou l’erreur a eu lieu.

## 7.3. Autres outils

### 7.3.1 Outils d’automatisation des tests

Pour l’automatisation des tests de l’interface, il sera nécessaire d’utiliser le *framework* *KIF* installé à l’aide *cocoaPod*. Pour le reste *Xcode* se charge de l’automatisation des tests de performance, de méthode, de classe, …

### Outils pour les tests de performance

L’outil pour les tests de performances est déjà intégré dans *Xcode*. Il suffit d’utiliser *XCTest* pour avoir la performance d’un bloc de code.