Université Montpellier II

Faculté des Sciences

MASTER 2 INFORMATIQUE

SPECIALITE « AIGLE »

RAPPORT DE STAGE

effectué à l'ADULLACT

du 3 mars au 8 août 2014

par

Luc DEBÈNE

Tuteur pédagogique

Clémentine NEBUT

Tuteur en entreprise

Pierre-Emmanuel VIVER

Remerciements

Avant tout développement sur cette expérience professionnelle, il apparaît opportun de commencer ce rapport de stage par des remerciements, à ceux qui m’ont beaucoup appris au cours de ce stage, et à ceux qui ont eu la gentillesse de faire de ce stage un moment profitable.

Aussi, je remercie Pierre-Emmanuel Viver, mon maître de stage qui m’a formé et accompagné tout au long de cette expérience professionnelle, Clémentine Nebut ma tutrice pédagogique pour l'intérêt qu'elle a porté à mon stage, Christian Buffin pour l'aide qu'il m'a apporté concernant la mise en place et l'utilisation de certains outils, Romain Monin et Rémi Dubourget pour m'avoir laissé expérimenter le serveur d'intégration continue et ses outils sur leur projet.

Je remercie également le corps enseignant de l'Université de Montpellier II grâce auquel j'ai tant appris durant mes années à la faculté.

Enfin, je remercie l’ensemble des employés de l'ADULLACT pour les conseils qu’ils ont pu me prodiguer au cours de ces six mois.

Table des matières

Index des illustrations

I. Introduction

A. Annonce du stage

Du 3 mars au 8 août 2014, j’ai effectué un stage au sein de l’Association des Développeurs et Utilisateurs de Logiciels Libres pour les Administrations et Collectivités Territoriales (ADULLACT), située à Montpellier. Au cours de ce stage, j’ai pu m’intéresser aux différents aspects de la qualité logicielle ainsi qu'à l'environnement de travail d'une entreprise.

Plus largement, ce stage a été l’opportunité pour moi d’appréhender les différentes notions applicables au domaine de la qualité logicielle, les outils utilisables, la mise en pratique des compétences acquises au cours de ma formation ainsi que la vie en entreprise.

Au-delà d’enrichir mes connaissances, ce stage m’a permis de comprendre l'importance pour une entreprise, quelque soit sa taille, de mettre en place un système de suivi de la qualité de leur logiciels.

En effet, tout comme c'est le cas dans l'industrie classique, un logiciel est un produit qui doit répondre à des normes de qualité afin d'assurer son fonctionnement sur la durée.

En revanche, contrairement à un produit classique, la durée de vie d'un logiciel n'est pas limitée par le monde matériel : le logiciel est destiné à évoluer. C'est dans ce cadre que la qualité logicielle intervient, afin de permettre une évolution suivie et sans régressions, les résultats obtenus justifiants la confiance que le client peut avoir sur le fonctionnement du logiciel.

B. Déroulement du stage

L'association m'ayant accueilli est l'ADULLACT, une association faisant la promotion du logiciel libre auprès des collectivités locales. Cette association est en lien étroit avec ADULLACT-Projet, une Société Coopérative d'Intérêt Collectif (SCIC) proposant du développement logiciel et du support technique. Ces deux entités seront décrites de façon plus détaillée plus loin.

Mon stage a consisté essentiellement en la découverte et la prise en main de différents outils permettant de mettre en place un suivi de la qualité des logiciels développés par l'entreprise.

Dans ce cadre j'ai pu apprendre un certain nombre de pratiques intéressantes à mettre en place pour le développement d'une application, notamment les tests unitaires que nous avions déjà vu durant notre cursus et dont j'ai pu comprendre le

réel intérêt, les tests fonctionnels ainsi que la possibilité de mettre en place un suivi outillé de la qualité d'un projet.

C. Problématique et objectifs du rapport

L'ADULLACT ne disposant pas encore d'un processus automatisé de suivi qualité sur leurs logiciels, j'ai été affecté à la mise en place de tests unitaires et fonctionnels appliqués à l'un des logiciels, le Service Sécurisé Libre inter-Opérable pour la Vérification et la Validation (S²LOW) ainsi qu'à la mise en place d'un serveur d'intégration continue afin de permettre un suivi sur ce logiciel. L'objectif à long terme de ce stage aura été de fournir une base de connaissances aux membres de l'ADULLACT concernant la qualité logicielle et son suivi.

Dans ce rapport, nous présenterons ainsi la mise en place d'un suivi qualité automatisé dans une structure telle que celle du couple ADULLACT/ADULLACT-Projet, comprenant les outils et techniques utilisés.

L’élaboration de ce rapport a pour principale source les différents enseignements tirés de la pratique journalière des tâches auxquelles j’étais affecté. Enfin, les nombreux entretiens que j’ai pu avoir avec les employés des différents services de la société m’ont permis de donner une cohérence à ce rapport.

En vue de rendre compte de manière fidèle et analytique ces mois passés au sein de l'ADULLACT, il apparaît logique de présenter le cadre du stage (II). Je présenterais les principaux outils utilisés et mis en place au cours du stage (III). Il sera ensuite précisé les différentes missions et tâches que j’ai pu effectuer concernant la mise en place d'un suivi de la qualité logicielle au sein de l'entreprise (IV), et les nombreux apports que j’ai pu en tirer (V).

II. L'ADULLACT

Cette association créée en 2002 et relevant de la loi de 1901 fait la promotion du logiciel libre auprès des collectivités et des administrations pour proposer une alternative aux logiciels propriétaires.

L'association propose la mise en place de groupes de travail pour l'élaboration de cahiers des charges et de plans de financement pour les projets qu'elle soutient. Elle propose également une forge de développement coopératif hébergeant les projets soutenus ainsi que du support et des formations sur les logiciels disponibles sur la forge.

Cette structure œuvre en coopération avec la coopérative ADULLACT-Projet créée en 2006 et propose aux collectivités de se charger du développement de logiciels répondant aux besoins précis émis par ces collectivités et administrations.

III. Les outils utilisés

Au cours de ce stage, j'ai eu l'occasion de me former ou d'approfondir mes connaissances sur certains outils. Je présenterais ici quelques-uns de ces outils : xUnit, le plugin Sélénium IDE et le serveur d'intégration continue Jenkins.

A. xUnit

Le nom xUnit désigne l'ensemble des frameworks de tests unitaires dérivés de celui créé par Kent Beck en 1998 (*SUnit* pour Smalltalk). Le 'x' de xUnit est souvent remplacé par la première lettre du langage associé (*JUnit* pour JAVA, *PHPUnit* pour PHP, etc...).

Illustration 2: Logo de JUnit

Illustration 1: Logo de PHPUnit

Les tests unitaires sont importants pour trois raisons principales :

- Ils permettent de vérifier la non-régression de l'application.
- Ils permettent de documenter chaque partie de l'application et de fournir des exemples d'utilisation.
- Ils permettent une refactorisation plus aisée et plus contrôlée du code.

Cependant, leur mise en place est souvent retardée – voire annulée – pour cause de délais trop courts accordés aux projets ou de manque de formation.

Selon Kent Beck, un développeur devrait passer plus d'un quart de son temps à écrire les tests unitaires associés à son projet.

Il est important de noter que les frameworks xUnit distinguent les erreurs des échecs durant l'exécution des tests unitaires :

- Un échec signifie que le résultat obtenu par le test en cours est différent du résultat attendu.
- Une erreur est bloquante et révèle qu'un problème majeur est survenu.

a) Bonnes pratiques

Un certain nombre de bonnes pratiques sont à connaître pour réaliser des tests unitaires convenables.

- Ne pas vérifier (via *Assert*) autre chose que la partie en cours de test. Cela alourdi le test inutilement et risque de provoquer un échec non significatif.
- Simuler (en anglais *Mock*) le comportement services externes tels que les FTP ou les bases de données. Certains frameworks proposent des outils pour effectuer ces simulations. Dans le cas particulier des bases de données, un *mock* permet de ne pas impacter sur une base de données réelle et de réduire le temps des requêtes à une simple lecture de fichier.
- Nommer de façon claire et explicite les tests. Cela permet de savoir de façon précise où se situe un éventuel échec / une éventuelle erreur. De plus, un système de nommage consistant au sein d'une même équipe facilite la maintenance.
- Ne pas écrire un test pour écrire un test. La production de mauvais test unitaire ou de test unitaire inutile est contre-productive. Cela alourdi la structure, donne parfois de fausses informations et rend difficile la compréhension du code testé. Il vaut mieux ne pas écrire de test que d'en écrire de mauvais. Dans le même ordre d'idée, ne pas écrire de test dans le seul but d'augmenter la couverture de code : un mauvais test couvre le code, mais ne fais que le survoler.
- Utiliser des *fixtures*[1] afin de contrôler au mieux l'état du système testé. Par exemple, pour un accès à une base de données, il faudrait extraire les quelques tables nécessaires au test en cours dans un fichier XML.

Concernant la couverture de code, il est admis qu'une couverture de 100 % n'est pas des plus souhaitable. En effet, cela sous-entend mettre sous test des procédures très simples comme les accesseurs (*Getters/Setters*). Une couverture de code considérée comme correcte se situe entre 60 % et 70 % pour de gros projets et environ 80 % pour des projets plus modestes.

1 Élément dont l'état est connu au début d'un test, potentiellement modifié par le test et qui retourne à son état initial une fois le test fini (fichier, configuration spécifique de l'environnement, base de données avec un jeu de données connus, etc...)

b) Exemple avec JUnit

Pour illustrer le propos, voici un exemple simple de classe et des tests unitaires associés.

```
//Class.java
public class Class {
 private int a ;
 private int b ;
 private Class(int a, int b) {
      this.a = a ;
      this.b = b ;
 }
 public int plus() {
      return this.a + this.b ;
 }
 public int times() {
      return this.a * this.b ;
 }
 /*...*/
}
```

```
//ClassTest.java
import org.junit.Test ;
import org.junit.Assert.* ;

public class ClassTest {
 private Class object ;
 @Before
 public void setUp() {
      this.object = new Class(3, 3) ;
 }
 @After
 public void tearDown() {
      this.object = null ;
 }
 @Test
 public void testPlus() {
```

```
        int expected = 6 ; //Résultat attendu
        int result = object.plus() ; //Appel de la méthode testée
        assertEquals("plus() result should be 6", expected,
                    result) ; //Assertion
    }
    @Test
    public void testTimes() {
        int expected = 9 ; //Résultat attendu
        int result = object.times() ; //Appel de la méthode testée
        assertEquals("times() result should be 9", expected,
                    result) ; //Assertion
    }
    /*...*/
}
```

Un test unitaire suivra toujours le même schéma :

- Préparation de l'état du système testé (méthode `setUp()`),
- Stockage du résultat prévu (dans l'exemple, la variable `expected`),
- Stockage du résultat de l'unité testée,
- Assertion comparant la prévision et le résultat obtenu,
- Retour à l'état d'origine du système testé (méthode `tearDown()`)

B. Sélénium IDE

Sélénium est un ensemble d'outils visant à automatiser des tests fonctionnels d'application web. Il permet entre autres d'enregistrer les actions effectuées dans un navigateur sous forme de tables HTML contenant les commandes propres à Sélénium.

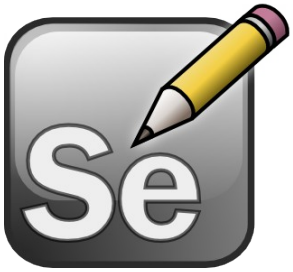

Illustration 3: Logo de Sélénium IDE

Les tests peuvent être regroupés en suites de tests et joués de façon automatique, ils peuvent être exportés vers différents frameworks de test (*JUnit*/*TestNG*, *NUnit*, *Test::Unit*/*RSpec*, *unittest*). L'export vers un de ces frameworks permet d'exécuter les tests sur d'autres navigateurs que Firefox en précisant le système d'exploitation et les caractéristiques du navigateur souhaité.

Sélénium propose un IDE[2] sous la forme d'un plugin Firefox permettant de créer des tests de façon simple grâce à un nombre fini de commandes, présentées sous forme de tables HTML et interprétées par un serveur fourni par Sélénium. Ceci permet de s'abstraire des différents langages (Java pour *JUnit*/*TestNG*, C# pour *NUnit*, Ruby pour *Test::Unit*/*RSpec*, Python pour *unittest*) pour construire les tests.

Sélénium propose également un serveur d’exécution permettant de lancer les tests sur une machine distante.

Le plugin Firefox se présente comme dans l'illustration 4 et propose plusieurs éléments :

1. Une adresse sur laquelle se baseront les tests
2. Une liste des cas de tests présents dans la suite de tests courante
3. Un bouton permettant d'activer/désactiver le mode "Enregistrement"
4. Une zone dans laquelle on trouvera les commandes qui composent le test actuel. Une zone d'édition des commandes se trouve juste en-dessous.

2 Integrated Development Environment

5. Une zone permettant d'afficher les logs, des aides concernant les commandes, etc...

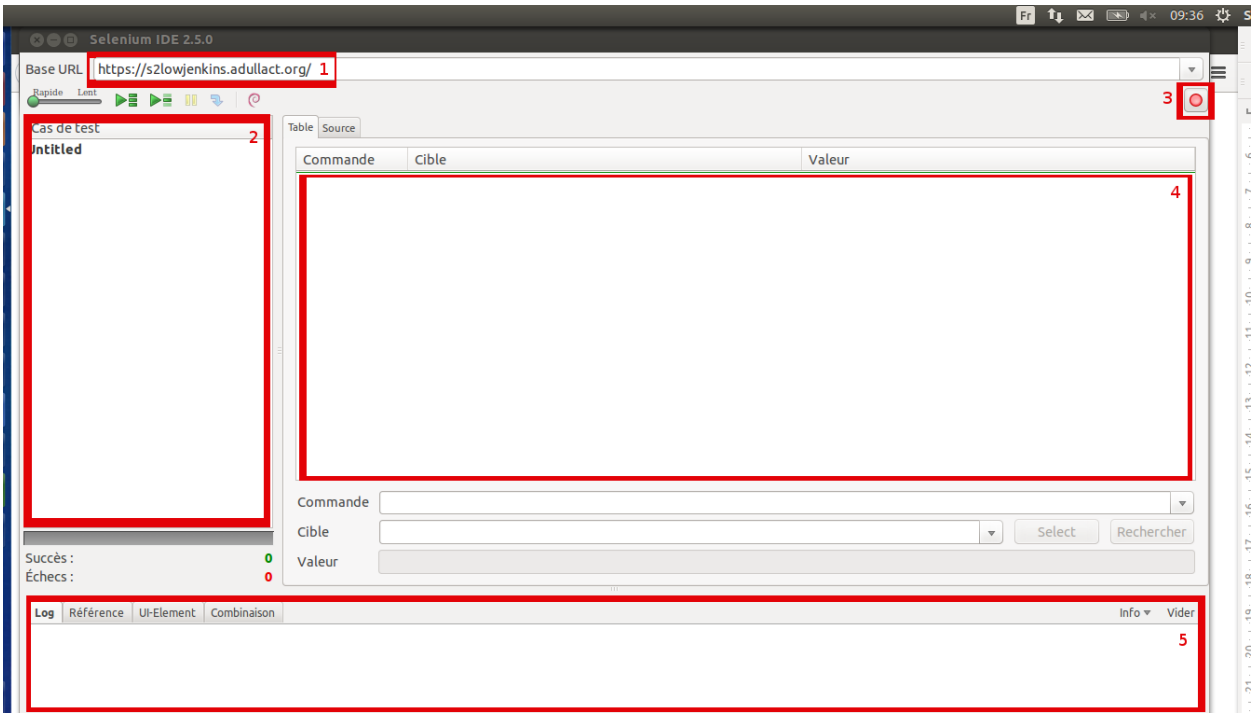

Illustration 4: Interface de Sélénium IDE

Les commandes créées par ou dans l'IDE sont toujours composées de trois valeurs :

- la commande, parmi celles fournie par Sélénium. `Click` simule un clic de souris, `VerifyX` effectue une vérification sur l'élément cible, `WaitForX` met l'exécution du test en attente tant que X n'est pas présent sur la page, `Type` simule le clavier pour remplir un formulaire, etc...
- la cible, un élément HTML dans la plupart des cas. Cet élément peut être accessible via son tag HTML, un identificateur (`link`), son descripteur CSS (`id`, `class`, etc.) ou un chemin Xpath (`//div[@id='div1']/p`)
- la valeur.

Le mode "Enregistrement" proposé par cet outil permet d'accélérer et de faciliter grandement la création de cas de tests, au prix de modifications si nécessaire (l'outil n'est pas infaillible). Ce mode scrute les actions effectuées sur le navigateur par l'utilisateur et les intègre aux commandes du test en cours de traitement.

Par exemple, cliquer sur un lien pointant sur http://google.fr/ crée la commande

Click	Link=http://google.fr	

Autre exemple : ouvrir la page d'accueil du site de Sélénium IDE, aller sur la page de téléchargement et vérifier que la version actuelle du plugin est la 2.5.0. On aura alors l'enchaînement de commandes suivant :

Open	http://docs.seleniumhq.org/projects/ide/	
Click	link=Download	
VerifyText	Xpath=//a[@href='*release*selenium-ide*']	2.5.0

Il est bien entendu possible de concevoir des tests bien plus poussés et complexes.

Les tests créés peuvent être joués de façon répétée sur le navigateur via l'IDE ou joués par le serveur fourni par Sélénium. Dans le deuxième cas, il est possible de lancer automatiquement ces tests en planifiant une tâche sur un serveur d'intégration continue.

Les tests s'exécutant dans un navigateur une interface graphique est nécessaire. Si le serveur d'intégration continue ne dispose pas d'une interface graphique, il est possible d'en simuler une avec XVFB (un serveur X virtuel).

C. Jenkins

> « Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:
>
> - Unit testing
> - Integration testing
>
> The purpose of construction testing is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. » [1]

Selon Bourque et Fairley, l'intégration continue est une étape importante à mettre en place dans le processus de développement logiciel. Nous parlerons donc ici de cette pratique et de l'outil utilisé au cours du stage.

L'intégration continue est une pratique qui a été promue dans les méthodologies de l'*Extreme Programming* (XP) qui automatise la construction des logiciels et un certain nombre de vérifications à chaque modification du code source sur un serveur (*commit*).

Durant mon stage, j'ai mis en place un serveur de ce type. Jenkins est un serveur d'intégration continue développé à l'origine par Kohsuke Kawaguchi sous le nom de Hudson puis *forké* après un différend avec Oracle.

Illustration 5: Logo de Jenkins

La mise en place d'un processus d'intégration continue à pour intérêt de permettre une détection rapide de problèmes pouvant survenir au cours du développement d'un logiciel tels que :

- une incompatibilité entre deux modules développés

- un problème de régression[3] entre deux constructions

Un serveur tel que Jenkins permet de mettre en place une automatisation de la construction des applications (via des fichiers de configuration *ANT*, *MAVEN,* etc...), un suivi de différentes métriques de qualité (telles que les tests unitaires, les tests fonctionnels, la conformité à des normes d'écriture du code source...), ces traitements pouvant être appliqués automatiquement à chaque *commit* sur un gestionnaire de versions.

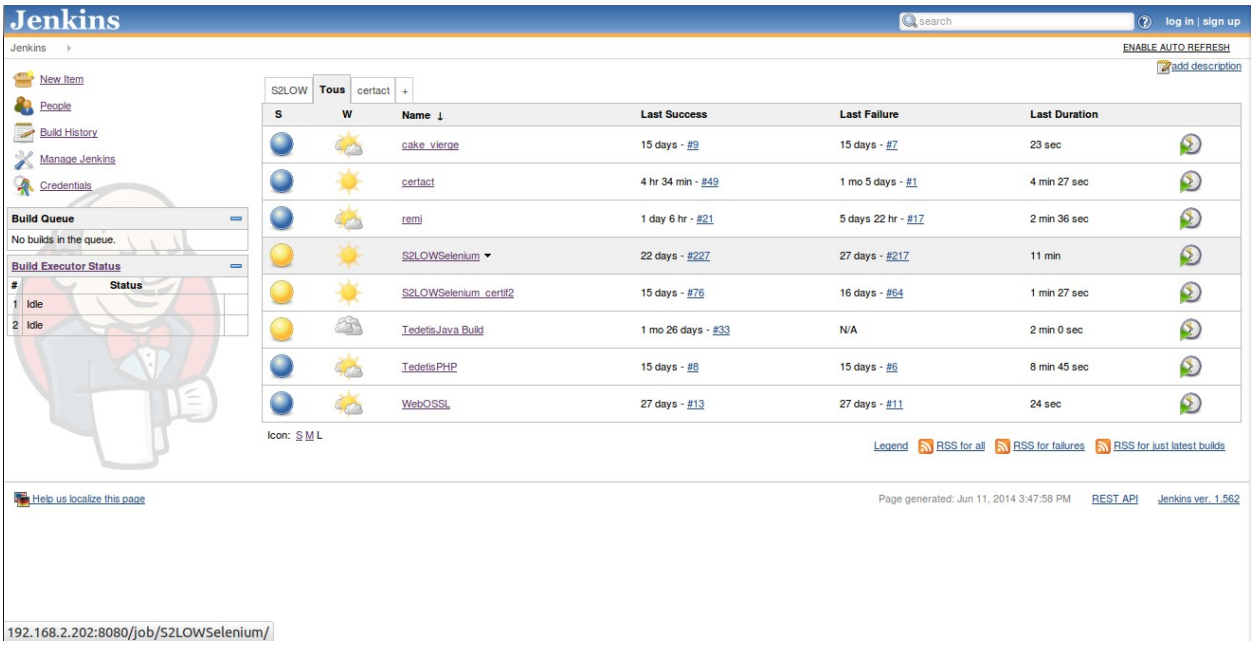

Illustration 6: Page d'accueil de Jenkins

Pour mettre à profit de façon optimale un outil tel que Jenkins, il convient de publier fréquemment, dès le début du projet, le code produit sur le gestionnaire de versions afin que l'analyse des outils installés rende compte des modifications récentes. Pour répondre au besoin du développeur de connaître l'état du code sur lequel il travaille, l'écriture de tests unitaires est nécessaire et ceux portant sur le code actuellement modifié doivent être exécutés à chaque *commit*.

Plusieurs fois par jour, selon la taille de l'équipe et la fréquence de *commit*, la totalité des tests unitaires doit être exécutée pour vérifier que les modifications croisées du code ne provoquent pas de conflit.

3 Fonctionnalité opérationnelle devenant non-opérationnelle sans la volonté expresse des développeurs.

Enfin, une fois par jour, tous les tests (unitaires, de charge, fonctionnels, etc.) doivent être exécutés pour s'assurer que le produit est conforme aux attentes à cette phase du développement.

La philosophie derrière une utilisation efficace d'un serveur d'intégration continue telle que décrite plus haut est que plus les erreurs sont détectées tôt, plus elles sont facile à corriger et moins elles impactent le projet dans sa globalité. Il faut pour cela écrire et maintenir un ensemble de tests automatisés et utiliser des outils produisant des indications par rapport à certaines métriques de qualité pour analyser leurs résultats et en tirer des conclusions concernant les modifications à apporter pour se diriger vers la production d'un logiciel fonctionnel et maintenable.

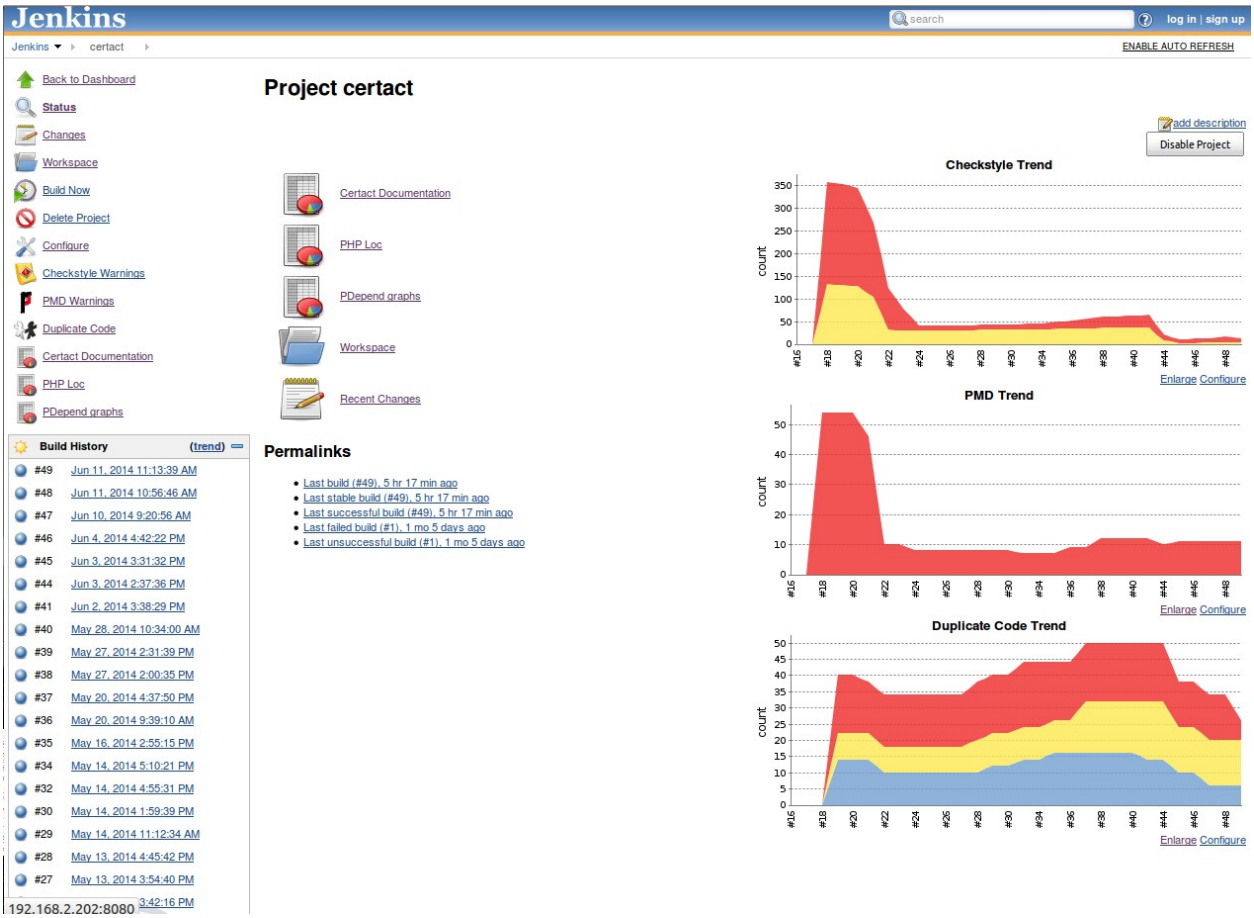

Illustration 7: Page présentant les résultat d'un projet

IV. Les travaux effectués

Au cours de ce stage, j’ai passé le plus clair de mon temps à découvrir et apprendre à utiliser correctement des outils de qualité logicielle tels que des frameworks de tests unitaires, un outil de tests fonctionnels, un serveur d'intégration continue...

J'ai donc eu l'occasion de mettre en place de tels outils et d'appliquer leur utilisation à un logiciel développé par la structure, ainsi que d'appliquer les outils de suivi aux projets attribués à Romain Monin et Rémi Dubourget (tous deux également stagiaires chez ADULLACT).

A mesure que j’apprenais, mes recherches se sont approfondies. Ce n’est donc qu’à partir du troisième mois de mon stage que j’ai été véritablement opérationnel, du fait de ma meilleure maîtrise de ces outils.

D. Les Missions du poste occupé

a) Présentation

Durant ce stage, mon travail s'est concentré autour d'un logiciel de télétransmission entre collectivités et organismes de l'état (S²LOW) développé et maintenu par l'ADULLACT. La plupart des techniques et pratiques acquises pendant ces 5 mois ont été appliquées à cette logiciel en vue d'en améliorer la qualité et d'en faciliter la maintenance.

J'ai également pu obtenir la coopération de deux autres stagiaires de l'association qui ont bien voulu soumettre leur projet aux outils de qualité mis en place (nous en parlerons plus bas).

L'objectif principal de ce stage, pour moi, étant de mettre en place un système de suivi de la qualité des logiciels qui apporterait au couple ADULLACT/ADULLACT-Projet et d'apporter des connaissances aux membres de l'association dans ce domaine, nous présenterons différents résultats obtenus grâce aux outils utilisés.

b) Le stage en question

Au cours de ce stage, différentes sortes d’activités m’ont été confiées :

- Mettre en place un serveur d'intégration continue (Jenkins), comprendre son fonctionnement, rédiger une documentation concernant son utilisation.

- Produire des jeux de tests unitaires correspondants au code source d'une application développée par l'ADULLACT.
- Produire des jeux de tests fonctionnels s'appuyant sur une instance de la même application.

Ces tâches se sont révélées interdépendantes, aussi les descriptions de celles-ci seront croisées et sans ordre chronologique.

Mise en place et utilisation de Jenkins

Le serveur d'intégration continue Jenkins a été mis en place au tout début de mon stage et a été utilisé pendant toute sa durée. J'ai donc du créer une machine virtuelle, y installer les différents outils nécessaires et apprendre à configurer Jenkins. Pour m'aider à effectuer cette tâche, j'avais à disposition une documentation rédigée par Christian BUFFIN (employé d'ADULLACT Projet) ainsi que les différents documents disponibles sur l'internet.

Une fois le serveur en place, j'ai pu prendre en main son fonctionnement tant du point de vue de l'administration que de l'utilisation.

Après avoir créé une suite de tests unitaires avec *JUnit*, j'ai pu lancer de façon automatique cette suite grâce à Jenkins et en récupérer les résultats.

Suivi de projets

Comme dit plus haut, deux stagiaires se sont prêtés au jeu et on accepté de soumettre leur projet PHP aux outils de qualité. J'ai ainsi pu obtenir des résultats concrets et suivis concernant leur respect d'une norme de développement (outil CheckStyle), de détection de « désordre » (outil Mess Detector), de présence de duplication de code (outil Copy/Paste Detector), ainsi qu'un suivi sur le nombre de lignes de code et la complexité cyclomatique de leur projet.

> Complexité cyclomatique : La complexité cyclomatique d'une méthode est définie par le nombre de chemins indépendants qu'il est possible d'emprunter pour parcourir le corps de cette méthode.
>
> Si on prend n_{dec} le nombre de points de décision de la méthode (for, while, switch, if, etc...), la complexité cyclomatique de la méthode vaut $cc = n_{dec} + 1$.

Ici, on étend la notion de complexité cyclomatique à la totalité d'un projet, il convient donc d'y appliquer des modificateurs prenant en compte la taille du projet. J'ai choisi de me baser sur deux modificateurs appliqués par l'outil PHPLoc : le nombre de lignes logiques de code (les lignes de commentaires ne sont pas prises en compte) et le nombre total de méthodes.

J'ai choisi de présenter en parallèle les résultats obtenus sur ces deux projets. Le premier projet (WebAED) d'une taille de l'ordre de 40k LOC (illustration 8) et le second (CG48) d'une taille de l'ordre de 20k LOC (illustration 10), on peut donc

comparer l'évolution des deux calculs de complexité au cours de leur développement.

Le projet WebAED ayant commencé avant la mise en place du suivi, il présentait alors une complexité par lignes logiques de code de 2,5 et une complexité moyenne par méthode de 2,3 pour 30 000 lignes de code. Le suivi sur le projet CG48 a quand à lui été mis en place tôt dans le développement. Il présentait alors une complexité par lignes logiques de code de 1,9 et une complexité moyenne par méthode de 2,1 pour 15 000 lignes de code.

On peut constater sur les deux projets qu'avec l'augmentation du nombre de lignes de code (illustrations 9 et 11), la complexité moyenne par méthode est maintenue proche de celle d'origine (2,3 pour WebAED et 2,1 pour CG48). Concernant la complexité par lignes logiques de code, on peut voir sur le projet WebAED la tendance à la baisse de cette valeur pour tendre vers 1,9. Sur le projet CG48, cette valeur reste très proche de 1,9.

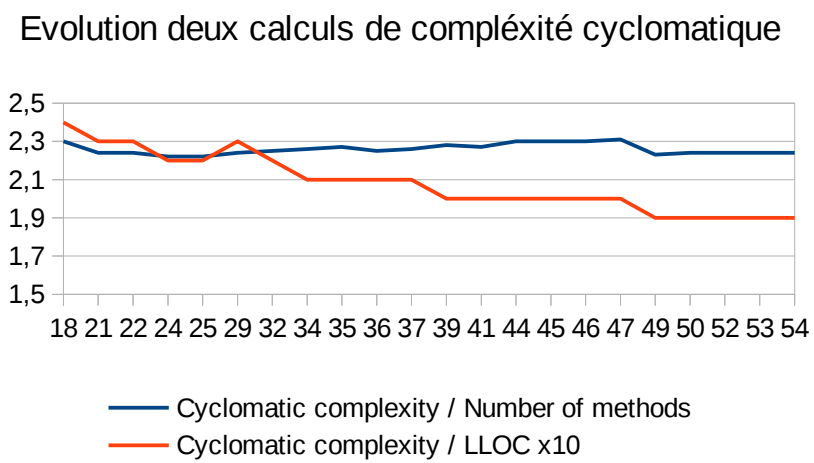

Illustration 8: Evolution de la complexité cyclomatique sur le projet WebAED

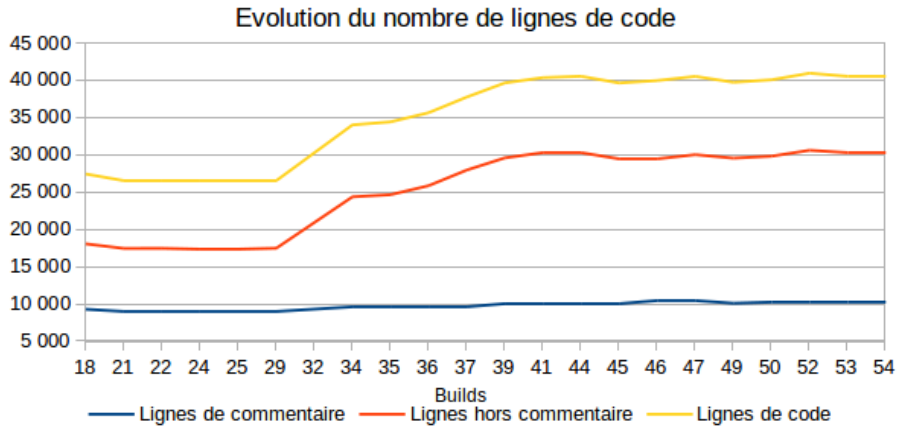

Illustration 9: Evolution du nombre de lignes de code sur le projet WebAED

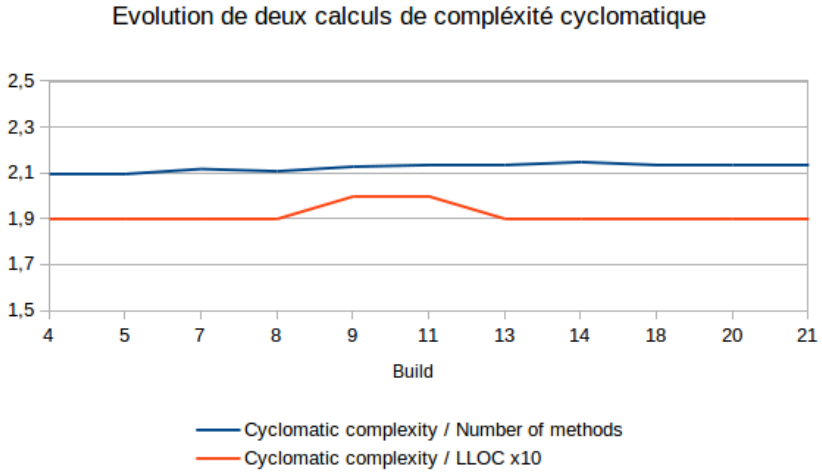

Illustration 10: Evolution de la complexité cyclomatique sur le projet CG48

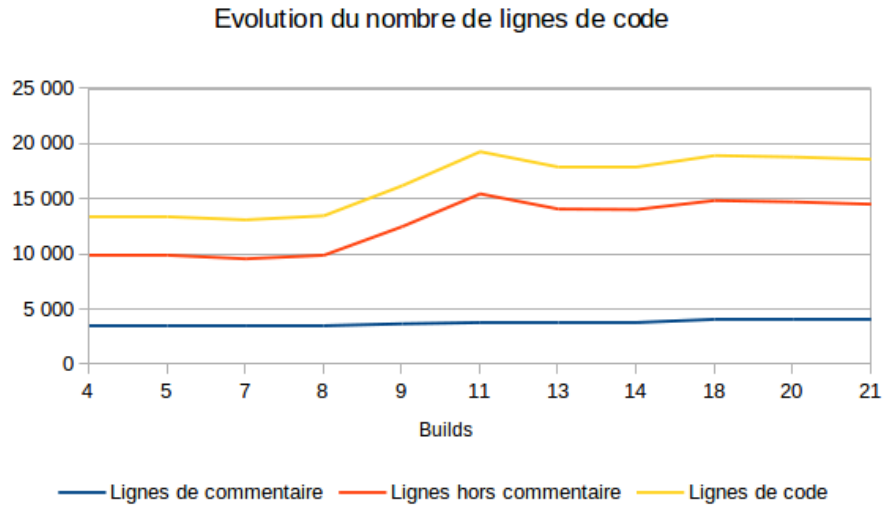

Illustration 11: Evolution du nombre de lignes de code sur le projet CG48

Le suivi mis en place grâce au serveur Jenkins n'a pas simplement permis de construire ces graphiques, il en propose lui-même en fonction des outils installés et paramétrés. Voici les graphes construits pour chacun des projets avec les résultats de trois outils : Checkstyle, PHPMD et PHPCPD (illustrations 12 et 13).

Romain comme Rémi avaient accès aux résultats produits par les outils et se sont basé sur ceux-ci pour améliorer la qualité de leur projet, d'où la tendance à la baisse des valeurs.

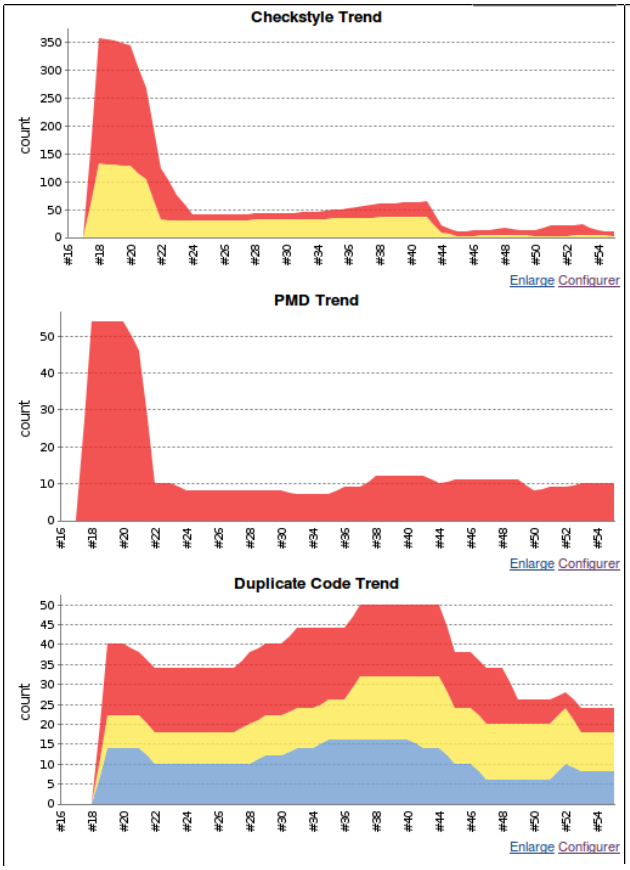

Illustration 12: Graphes produits par Jenkins sur le projet WebAED

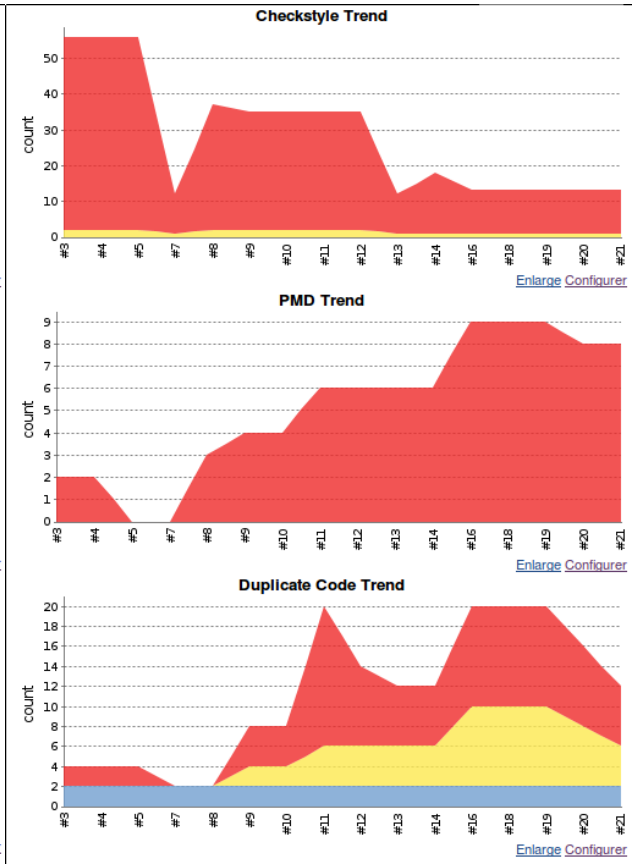

Illustration 13: Graphes produits par Jenkins sur le projet CG48

Comme dit plus haut, l'outil Checkstyle permet de suivre le degré de conformité à une norme de développement, l'outil PMD permet de suivre des valeurs considérées comme faisant « désordre » (trop de méthodes dans une même classe, méthodes trop longues, etc...) et l'outil Duplicate Code permet de suivre les problème de duplication de code.

Ces derniers sont les plus problématiques dans le cadre de la maintenabilité, puisque une portion de code dupliquée n'est pas forcément documentée comme telle et il est possible de ne pas modifier toutes ces duplications laissant alors la porte ouverte à des comportements non souhaités et difficilement repérables.

Lien avec Sélénium

En parallèle du suivi de ces deux projets, j'ai eu à automatiser les tests du cahier de recette de S²LOW à l'aide de l'IDE Sélénium.

Pour obtenir un suivi des résultats des tests automatisés, j'ai dû configurer le serveur pour permettre l'exécution de ceux-ci sans interface graphique (des précisions seront données en annexe B). J'ai ensuite pu construire les suites de tests et les exécuter régulièrement.

Sur l'illustration 14, on peut voir un pic de tests ayant échoués durant le build 142. Ceci provient du fait que le logiciel a changé de version et a été « bootstrapé »[4]. Les tests fonctionnels avec Sélénium sont dépendants des identifieurs CSS ou Xpath, si ceux-ci sont modifiés il faut modifier le test pour correspondre au nouveau design.

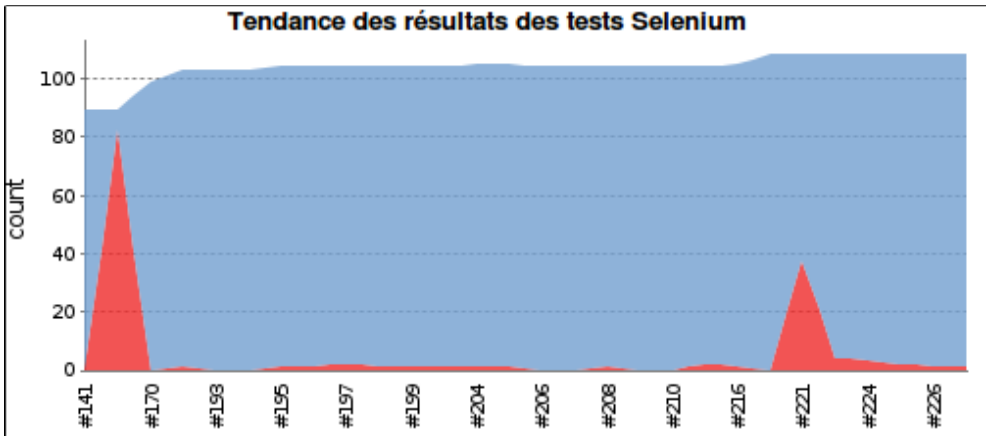

Illustration 14: Résultats obtenus après exécution des tests Sélénium

4 http://getbootstrap.com/

Création des tests unitaires sur le logiciel S²LOW

Durant le deuxième mois de mon stage, je me suis auto-formé aux tests unitaires en cherchant diverses références sur le sujet. J'ai ainsi pu intégrer ce qui ressortait comme « bonnes pratiques » (ne tester qu'une chose à la fois, donner des noms clairs et précis aux tests pour savoir où et ce qu'ils testent, réduire le plus possible les dépendances, etc...) pour la création de tests unitaires. Deux ouvrages m'ont particulièrement aidé dans ma compréhension de la chose : *« Working effectively with legacy code »*[2] de Michael Feathers et *« XUnit Test Patterns »*[3] de Gerard Meszaros.

La création des tests unitaires s'est déroulée en deux étapes.

La première étape a été de reprendre les tests unitaires existant dans le code JAVA du logiciel. J'ai ainsi pu avoir une première visibilité sur le fonctionnement du framework *JUnit* (déjà vu au cours de mon cursus, mais dont l'utilité ne m'apparaissait pas alors) et d'avoir quelques exemples concrets de son utilisation. J'ai ensuite commencé à construire les suites de tests unitaires en m'efforçant de respecter les bonnes pratiques citées plus haut. Le plugin *JUnit* pour *Eclipse* proposant un outil de couverture de code, j'ai pu constater que les tests produits couvraient environ 34 % du code JAVA pour ce logiciel.

La seconde étape a été de produire des suites de tests unitaires pour la partie PHP de S²LOW. J'ai pour cela dû me former au framework *PHPUnit*. Ayant déjà intégré les bonnes pratiques à appliquer pour créer des tests unitaires, cette phase s'est révélée plus rapide que pour le framework *JUnit*. J'ai ensuite produit une série de tests unitaires pour cette partie du logiciel. Pour ce faire, j'ai dû comprendre comment s'agençait le code, surmonter les méthodes non factorisées et m’accommoder des autres surprises que réserve le PHP (déclaration d'attributs de classe à la volée, possibilité d'accéder à des éléments privés, etc...).

J'ai bien entendu profité du serveur d'intégration continue préalablement créé pour lancer régulièrement les tests unitaires et en constater les résultats.

Durant ce stage, mon point de vue à propos des tests unitaires a fortement évolué. Je suis maintenant convaincu que, sans aller jusqu'au *Test-Driven Development* (TDD), un socle de tests unitaires durant le développement d'un logiciel est nécessaire, ne serait-ce que pour les fonctionnalités critiques.

Comme j'ai pu le lire au cours de mes recherches sur le sujet, une couverture de code de 100 % n'est pas forcément utile – voire pas souhaitable – car cela signifierait que la moindre déclaration de variable ou le moindre accesseur serait testé. Cela alourdirait inutilement les tests et augmenterait leur temps d'exécution. Selon la taille de l'application, une couverture de code située entre 60 et 80 % se révèle suffisante pour couvrir la plupart des fonctionnalités.

Cependant, le pourcentage de couverture de code reste une indication, il est tout a fait possible de couvrir du code sans effectuer de test sur les valeurs qu'il retourne ou les effets de bord qu'il provoque. Il faut donc veiller à produire des tests ayant un objectif précis.

Prise en main et création de tests fonctionnels avec Sélénium IDE

J'ai eu l'occasion de découvrir Sélénium IDE, un outil permettant d'automatiser les tests fonctionnels d'une application web. Après une période de prise en main de l'outil, j'ai été à même de construire les premiers tests fonctionnels automatisés du logiciel S²LOW.

J'ai par la suite continué à produire les tests fonctionnels en me basant sur le cahier de recette fourni par mon tuteur tout en vérifiant régulièrement leur fonctionnement. Plusieurs suites de tests ont été produites car S²LOW nécessitant des authentifications via certificats, il m'a fallu vérifier les comportements de chaque type d'autorisation (et donc plusieurs certificats).

Durant mon stage, le logiciel S²LOW a plusieurs fois été modifié par le développeur et les tests fonctionnels mis en place ont permis de détecter plusieurs malfonctions qui ont pu être corrigées rapidement. Le logiciel a également changé de version et a été « bootstrapé ». Ce changement ayant grandement modifié l'arborescence des pages web proposées par le logiciel, l'exécution des tests précédemment conçus a échoué (voir illustration 11). J'ai donc dû reprendre une partie des tests existants et les modifier de façon à prendre en compte la nouvelle arborescence.

Au final, le cahier de recette a été couvert par les tests produits et ceux-ci sont réutilisables dans un environnement similaire à celui dans lequel ils ont été conçus.

De mon point de vue, l'utilisation d'un outil tel que Sélénium IDE est très intéressante dans le cadre d'une application web car cet outil permet de contrôler le bon fonctionnement de l'application non d'un point de vue de développeur mais d'un point de vue d'utilisateur et car il est aisé de prendre en main l'outil sans

aucune connaissance en développement web. La création des tests peut donc être confiée à une équipe ayant une bonne connaissance de la partie utilisateur du logiciel et disposant d'un cahier de recette complet, une sensibilisation aux notions de CSS et Xpath peut cependant être profitable.

L'interprétation des résultats des tests, en revanche nécessite d'avoir une bonne représentation des éléments en place pour déterminer d'où proviennent les éventuelles erreurs. Comme pour tout test, un nom suffisamment descriptif et une seule tâche à vérifier sont nécessaires pour avoir le plus d'informations possible sur la nature de l'erreur.

La conception des tests fonctionnels automatisés peut sembler rébarbative. Mais une fois un socle de base posé, les tests présents peuvent être réutilisés pour en construire d'autres. De plus, la mise en place de tests tels que ceux-ci permet de détecter les erreurs tel que l'utilisateur peut les voir et qui ne sont pas facilement détectables dans le code source. Il faut cependant penser au plus d'actions possible réalisables via l'interface utilisateur et, si possible, imaginer des scénarios d'utilisation non souhaités.

E. Les tâches périphériques

Au cours de mon stage, j’ai pu effectuer trois tâches se situant à la périphérie des tâches effectuées :

J'ai écris des manuels de prise en main rapide pour le plugin Sélénium IDE et le serveur d'intégration continue Jenkins. Ces documentations ont été mises à disposition des membres de l'ADULLACT et des employés d'ADULLACT-Projet pour leur permettre d'utiliser ces outils dès la fin de mon stage. Vous pourrez les trouver en annexe à la fin de ce rapport.

J'ai également eu pour tâche de développer un site web minimaliste permettant de tester le bon fonctionnement de l'API[5] fournie par S²LOW. Cette API permettant d'utiliser les fonctionnalités du logiciel sans interface, il était nécessaire de disposer d'un moyen pour en vérifier le fonctionnement. Ce site permet donc d'effectuer toutes les opérations disponibles via l'API et d'en constater les résultats.

Enfin, j'ai pu transmettre les connaissances acquises sur le plugin Sélénium IDE et les tests fonctionnels au pôle « formation » d'ADULLACT-Projet via la documentation ainsi qu'une réunion visant à expliquer les techniques, les avantages et inconvénients, et comment interpréter les résultats obtenus grâce à cet outil.

5 Application programming interface

V. Les apports du stage

Ce stage a selon moi été intéressant tant pour l'association que d'un point de vue personnel. Nous verrons ici les différents apports de ce stage.

A. Apports pour l'entreprise

Ce stage a été l'occasion pour l'association de constater l'intérêt de la mise en place d'un système de suivi de la qualité logicielle.

De plus, les documentations rédigées permettront une utilisation rapide des outils mis en place au cours du stage.

Ces processus, une fois appliqués à plusieurs logiciels, permettront à l'association d'en améliorer la qualité ainsi que de disposer de critères pour renforcer la confiance que les utilisateurs peuvent avoir dans ces applications.

B. Apports personnels

Au cours de ce stage, j'ai pu développer une plus forte sensibilité à la qualité logicielle dans son ensemble.

J'ai ainsi découvert l'intégration continue grâce au serveur que j'ai créé au début de mon stage, pu constater et comprendre l'efficacité de la mise en place d'un tel serveur dans une entreprise – voir pour un développeur seul – (pour peu que l'on sache choisir les outils à utiliser) pour effectuer un suivi et apporter des modifications au projet si l'on franchi certains seuils.

Pour autant, un tel serveur ne doit pas être détourné par des supérieurs hiérarchique pour monitorer l'avancement d'un projet et justifier des réductions de délai arbitraires. Une telle utilisation ne pourrait être, selon moi, que contre-productive.

Concernant les tests unitaires, j'avais déjà eu un aperçu de la chose durant mon cursus universitaire. Cependant, ce stage m'en a fait comprendre l'importance pour l'amélioration d'un code par refactorisation, pour pouvoir garantir qu'une modification ne perturbe pas le fonctionnement du logiciel et pour conserver la cohérence des fonctionnalités, indispensable dans une structure proposant des versions LTS[6] de sa gamme de logiciels.

6 Long Time Service

Les tests fonctionnels m'ont permis d'apprendre à suivre et créer un cahier de recette, tout en découvrant l'utilisation d'un plugin d'automatisation de ceux-ci. Ces tests permettant de vérifier le fonctionnement d'un logiciel du point de vue d'un utilisateur, il est nécessaire d'en tester les fonctionnalités principales afin de s'assurer de leur état de fonctionnement.

Ces tests, une fois automatisés, sont particulièrement fragiles quand aux changements dans l'interface utilisateur. Il est toutefois assez aisé de les modifier pour les faire correspondre à la nouvelle interface. Le fait qu'il ne nécessitent pas d'être exécutés à la main est une charge en moins pour le personnel. Il est pourtant parfois nécessaire d'effectuer quelques tests fonctionnels manuellement, tous les comportements de l'utilisateur ne pouvant être prévus.

J'ai également été confronté à l'utilisation de certificats électroniques et aux contraintes que cela impose du point de vue des tests.

C. Difficultés rencontrées et solutions apportées

Durant mon stage, j'ai été confronté à un certain nombre de difficultés dont voici les deux plus marquantes :

Lors de l'écriture des tests unitaires, j'ai constaté que beaucoup de fonctionnalités accédaient à la base de données du logiciel, opéraient avec un outil FTP, etc. Ne pouvant écrire les tests pour ces fonctionnalités sans les avoir installées et configurées sur ma machine, j'ai recherché un moyen de contourner le problème. J'ai ainsi découvert les techniques de *mocking*[7].

Il existe donc divers moyens de découpler les fonctionnalités pour les tester indépendamment les unes des autres tels que :

- Créer une sous-classe de la classe à substituer et redéfinir les méthodes pour renvoyer le résultat souhaité.
- Utiliser un framework de *mocking* qui se charge de ce travail.

Il est toutefois nettement plus aisé de procéder à du *mocking* si une interface est définie pour chaque classe à simuler.

A cause de mon manque de recul sur le processus de *mocking*, je n'ai pas pu mettre celui-ci en place dans les tests unitaires du logiciel S²LOW.

La deuxième difficulté est survenue pendant l'écriture du site appelant les scripts de l'API de S²LOW. J'avais à ma disposition la documentation de cette API et m'y suis malheureusement trop conformé. J'ai donc passé un temps bien trop long sur une erreur bloquante avant de comprendre qu'une erreur dans la documentation provoquait ce blocage. Une fois cette erreur détectée, une simple modification a suffit pour débloquer la situation.

7 Le mocking (« subterfuge ») consiste à remplacer une fonctionnalité par un dérivé inerte, de manière à découpler la fonctionnalité à tester et la fonctionnalité à mocker.

VI. Conclusion

Pour conclure ce rapport, je tiens à présenter certains avis que j'ai développé au cours de ce stage.

D'après les différents articles que j'ai pu lire et ce que j'ai pu constater au sein de l'ADULLACT, il est fréquent de voir des projets dont les délais sont trop courts, pour lesquels le temps d'une vraie conception (langage adapté, framework adapté, architecture pensée en amont et évolutive) n'a pas été pris, y compris dans un contexte de méthodologie Agile.

De plus, les étapes de tests ne sont pas souvent effectuées car elles ne sont pas comprises dans les délais attribués aux projets. Les conséquences de cela :

- Un logiciel peut être une plaie à développer car le langage ou le framework est mal choisi (habitudes de l'entreprise, pas assez de recherches, etc...)
- Les retours des clients sont parfois les seuls permettant de détecter des bugs

Partant de ce constat, il me paraît bon de rappeler l’existence d'une notion adoptée en gestion de projet : le triangle du projet (Illustration 15).

Illustration 15: Triangle de gestion de projet

Ce triangle présente une vision simplifiée des notions indispensables pour un projet. Chaque sommet correspond à l'une de ces notions et influe sur les les deux autres.

Cette figure nous montre que l'équilibre d'un projet est fragile et qu'il est difficile de tout avoir. Selon cette représentation, trois formes sont possibles :

- Un projet de qualité avec un coût raisonnable augmentera les délais, car nécessitant plus d'approfondissements, passant au second plan par rapport à des projets plus rentables...

- Un projet de qualité avec un délai restreint augmentera le coût, car nécessitant plus de ressources humaines et matérielles.
- Un projet avec un coût faible et des délais restreints nuira à la qualité globale du projet.

Pour bien continuer, il convient de définir la notion de qualité dont nous parlons. La qualité ici correspond à l'ensemble des critères de respect des spécifications client, la maintenabilité, la ré-utilisabilité, la stabilité du produit délivré.

Ainsi, pour pouvoir proposer des logiciels de qualité, il convient dès le début du projet de définir des délais cohérents afin de laisser le temps aux développeurs de concevoir correctement le produit, de ne pas être en permanence l'œil rivé sur la montre avant une livraison et d'avoir le temps d'assurer des étapes de tests.

Ceci est un travail de pédagogie envers les supérieurs hiérarchiques et les clients qui n'ont pas forcément une vision de la réalité du processus de développement d'un logiciel, surtout dans le cas d'une structure de taille réduite. Les SSII peuvent se permettre des délais plus courts car ces entreprises disposent d'équipes de taille conséquentes et de processus plus cadrés.

Des entreprises de taille modeste peuvent, en y attribuant le temps, atteindre certains critères de qualité sur leurs logiciels grâce à l'utilisation d'outils tels que ceux présentés dans ce rapport, mais aussi des outils d'aide à la conception et à la documentation (documentation générée à partir des commentaires dans le code, modélisations UML, etc...)

Une sensibilisation à ces outils coûte évidemment du temps, mais permet sur le long terme d'améliorer la qualité globale du code produit, la stabilité des produits et la confiance que le client peut accorder à l'entreprise. Plus ces habitudes s'ancreront, moins le temps pris pour utiliser ces outils, écrire des tests unitaires comme fonctionnels, sera long.

Références bibliographiques

[1] Bourque, P. & Fairley, R.E. (2007) *Guide to the Software Engineering Body Of Knowledge (SWEBOK Guide v3.0).* 335p.

[2] Feathers, M. (2004) *Working effectively with Legacy Code.* Prentic Hall Professional. 426p.

[3] Mezaros, G. (2007) *xUnit Test Patterns : Refactoring Test Code.* Addison Wesley Professional. 883p.

Uribe, V. (2012) *Intégration continue et Team Foundation Server* [en ligne] (page consultée le 22 juillet 2014)

http://blog.soat.fr/2012/07/integration-continue-et-team-foundation-server-12/

Anderson, S. (2009) *Writing Great Unit Tests: Best and Worst Practices* [en ligne] (page consultée le 18 juillet 2014)

http://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/