UQÀM

Rapport de session

Équipe Delta

MGL7361

Julien Lussier, Stéphane Tessier et Paul-Olivier Trudeau

Automne 2011

Table des matières

1. Projet initial

Le projet qui est réalisé dans le cadre de ce travail est l'écriture d'une librairie permettant de spécifier et d'exécuter des tests unitaires. L'évolution de cette librairie durant plus de sept semaines se doit de suivre les nouveaux requis demandés au fil des semaines par le professeur.

Le défi de ce travail est de mettre continuellement à l'épreuve notre design et de le faire évoluer afin qu'il soit toujours en mesure de répondre efficacement, le plus facilement et le plus rapidement possible aux nouvelles fonctionnalités demandées. L'efficacité en question est très subjective, et c'est pourquoi chacune des équipes devra attaquer la qualité du design d'un angle particulier. En ce qui nous concerne, notre équipe a le mandat de réaliser un design mettant l'accent sur la flexibilité et la clarté.

Les fonctionnalités de base de la librairie de tests (semblable à JUnit) demandée par le professeur sont les suivantes :

- possibilité de déclarer des cas de tests qui contiennent plusieurs tests individuels ;
- mécanisme d'assertion permettant de vérifier que deux objets sont identiques au sens de Java ;
- possibilité d'exécuter tous les tests d'un seul coup;
- exécution de tous les tests même si un test échoue durant la vérification ;
- affichage d'un message indiquant le résultat des tests à la fin de l'exécution.

1.1 Réflexion sur la clarté et la flexibilité

Notre réflexion sur la flexibilité et la clarté du design fut très intéressante, nous avons identifié 2 points de vue différents, la sémantique et le concret.

D'un point de vue strictement sémantique, le terme « flexibilité » est plutôt large. Il peut être difficile de définir, dans un contexte de design, le sens de ce mot. Si on le définit dans le sens où le design doit être prêt à toute éventualité de changement et en subissant un minimum de réécriture, alors il nous parait difficile de s'y conformer.

Comment pourrions-nous rendre facilement un design simple et clair tout en planifiant toutes les éventualités? À première vue, nous sommes aux antipodes! Tout prévoir implique avoir beaucoup de croisements et de code non utilisé. Effectuer un changement sur ce genre de design impliquerait alors le changement de code qui n'est pas encore utilisé et donc, d'appliquer des changements pour rien. Cette approche complexifierait alors le design et le code pour rien !

De l'autre côté, d'un point de vue plus concret, et puisque nous devons réaliser un design flexible et clair, notre philosophie pourrait être de tenter de développer notre librairie de la manière la plus simple possible et de la rendre la plus fonctionnelle possible et avec un minimum de design (« No design up front »). Selon nous, cette approche ferait plus de sens puisque notre réflexion nous amène à penser que le moins on utiliserait de code, tout en respectant les bonnes pratiques et les requis, le plus simple notre code restera. En appliquant cette logique, si notre code est léger, simple et clair, il devient donc facile à comprendre et facile à adapter aux changements, donc il est plus flexible.

L'approche concrète nous semble plus appropriée et c'est donc celle-ci que nous allons privilégier pour l'exécution de notre travail.

De plus, pour ajouter un peu plus d'assurance à notre décision, un membre de notre équipe nous a expliqué ce qu'il avait appris durant un cours sur la productivité et les outils (*Qualité et productivité des outils logiciels*). En fait, cet étudiant a fait l'étude de certaines méthodologies de développement dites agile durant ce cours et pour lui, il était évident que ces méthodologies prônaient à leurs façons la flexibilité et la simplicité par le « réusinage », la diminution des lignes de code, le développement par les tests, le design continuel, le repoussement des décisions difficiles ou incertaines à plus tard et ne faire que le strict minimum demandé.

Nous avons donc décidé de prôner comme équipe, ces mêmes valeurs puisqu'elles étaient remplies de gros bon sens et qu'elles ressemblaient à la conclusion de notre propre réflexion.

La prochaine section présentera de manière plus concrète, ce que nous ferons pour rester flexible et clair dans le design et le développement de notre librairie de tests.

1.2 Notre approche globale

Voici donc ce que nous avons décidé de faire pour répondre à la clarté et à la flexibilité:

- Nous allons démarrer avec le design le plus simple possible (aucun « surdesign ») puisque nous savons que nous ne pourrons pas trouver le design parfait dès le départ ;
- Nous tenterons de ne faire que ce qui est demandé (voyager léger sans « gold plating ») ;
- Nous allons répondre aux exigences les plus importantes pour la première partie des itérations et nous allons ensuite améliorer et « réusiner » le code lorsque les requis seront tous remplis ;
- Nous supprimerons les méthodes, classes inutiles et le code inutile pour éviter de les traîner pour rien durant plusieurs semaines ;
- Nous tenterons de faire le plus simple possible et de simplifier encore plus si possible ;
- Nous allons reporter les décisions difficiles le plus loin possibles dans le temps, au moment où il est clair que ça constitue un avantage.

De plus, nous allons suivre certains autres principes qui faciliteront la clarté et la flexibilité de notre code tel que :

- Nous allons tenter d'encapsuler ce qui peut changer comme le type de Log, l'Assert, etc.) ;
- Nous tenterons de programmer en utilisant les interfaces et des classes abstraites plutôt qu'en se liant aux implémentations ;
- Nous allons favoriser la composition, face à l'héritage si le besoin survient. Pour nous, l'approche de la composition procure normalement une encapsulation plus forte ;
- Nous allons tenter d'attribuer une seule responsabilité par classe ;
- Nous allons tenter de faire des interfaces granulaires, spécifiques aux besoins des clients.

Finalement, nous voulions faire des tests unitaires automatisés avec jUnit, nous n'avons pas eu la chance d'en faire puisque nous ne trouvions pas de méthode simple. Cependant, nous avons fait plusieurs tests de notre libraire. Si nous avions la chance de poursuivre le développement de la librairie, avant de faire tout autre changement, nous ferions le développement de tests unitaires afin que notre code puisse évoluer plus facilement et plus rapidement.

1.3 Design initial

Notre premier design fut effectué quelques jours après la présentation des requis par le professeur et suite à notre réflexion sur la flexibilité et la clarté dans le design présentée à la section précédente.

Présentation d'un croquis simple de notre design initial:

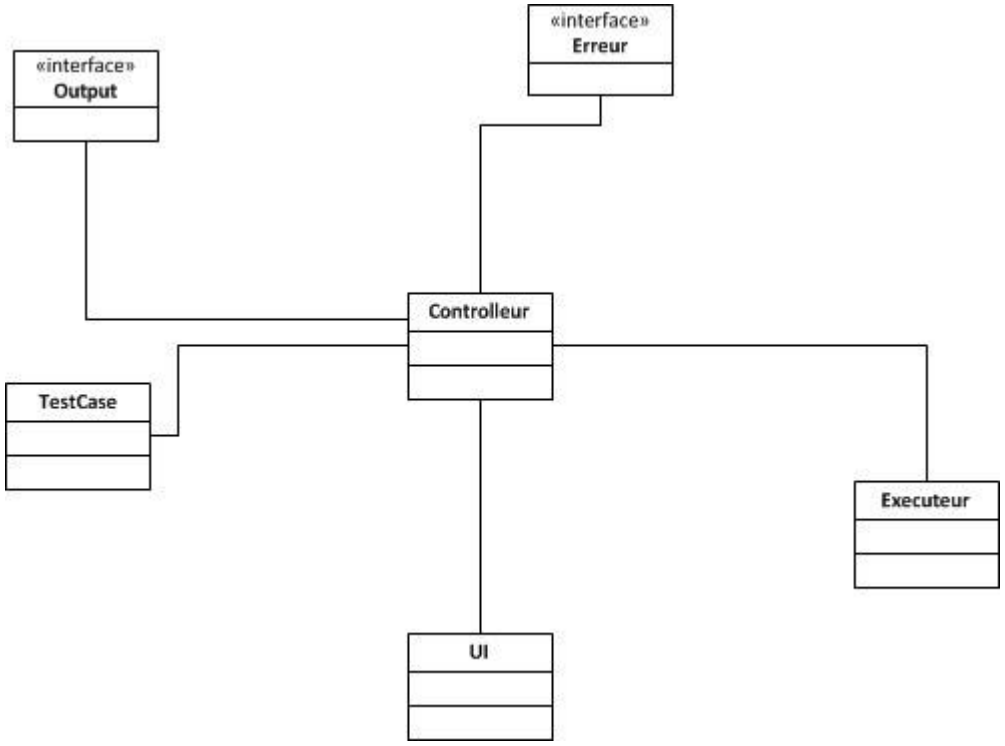

Figure 1 - Design initial

Lors de la première itération de notre librairie, nous avons répondu aux exigences les plus importantes en créant une classe contenant des tests, une classe classTester pour exécuter les tests, et une interface pour les logs à afficher. Les tests étaient représentés par des annotations, alors que la classe principale contenait une grande méthode main() contenant tout le code, dans laquelle on bouclait sur toutes les méthodes (trouvées par réflexion) annotées avec un @test. Notre première version de la librairie adoptait une approche un peu procédurale. Pour ce qui est des logs, malgré le fait qu'ils étaient demandés seulement à la console, nous croyions qu'il s'agissait d'une bonne idée d'en faire une interface en raison du faible coût et du potentiel volatile d'une telle fonctionnalité. En effet, on se doutait bien que le besoin d'écriture dans un fichier allait apparaître tôt ou tard. À l'inverse, à ce point

du projet, nous avons pris la classe Assert de Java, sans la changer ni même l'encapsuler, sachant que ce couplage n'était pas forcément le design le plus optimal. Finalement, les résultats des tests étaient simplement envoyés à la sortie, au fur et à mesure de l'exécution de ceux-ci.

1.4 Conclusion de notre travail

À notre grande surprise, puisqu'aucun de nous n'est un programmeur Java dans sa vie professionnelle, nous nous attendions à avoir beaucoup de difficulté à exécuter le travail.

En fait, nous nous sommes posé la question à savoir pourquoi nos changements semblaient être faciles à appliquer chaque semaine! Nous avons compris que puisque le code était très léger et faisait le minimum requis, il devenait possible rapidement de le changer, d'extraire une méthode ou une classe si nécessaire et de repartir dans une autre direction. Évidemment, tous les changements demandés avaient du sens alors c'était plus simple de faire évoluer notre code. Si le professeur avait demandé des changements à l'orientation de l'application alors nous aurions eu beaucoup plus de problèmes, mais selon nous, il en aurait été de même pour n'importe quel type d'approche de développement.

Les problèmes dits « majeurs » que nous avons rencontrés étaient tous reliés à une incompréhension d'un requis (par exemple les suites de suites de tests et les paramètres à passer en ligne de commande) et le temps que nous avons perdu à tenter de trouver des solutions à des problèmes que nous n'avions pas vraiment (des tests qui peuvent recevoir des paramètres et qui finalement après discussions avec le professeur, n'étaient pas requis). Nous avons traîné cette « fausse » fonctionnalité, en raison d'un manque de communication avec le client.

Durant les sept semaines de développement, chaque modification semblait aller dans le sens de notre design. Disons que notre design était si simple qu'en fait, un changement au design n'en paraissait pas un très important. Évidemment, notre application a changée, ou plutôt elle a évoluée et certains patrons de design se sont installés doucement sans que nous le voyions.

1.5 Liste des demandes de changement

Les prochaines sections présenteront les demandes de changement effectuées par le professeur ainsi que les impacts et changements dans le design que nous avons dû apporter. Nous discuterons également des problèmes encourus ainsi que les solutions trouvées.

1.5.1 Première modification

Voici la liste de changements demandés :

- Possibilité d'écrire dans un fichier les résultats des tests ;
- Avoir une méthode setup() avant l'exécution et une méthode teardown() après l'exécution de tests unitaires.

1.5.1.1 Design préconisé

Pour permettre l'écriture dans un fichier, nous avons choisi d'ajouter une implémentation additionnelle pour l'interface Log, et de récupérer le type de log souhaité par un paramètre à la ligne de commande. En ce qui concerne le setup() et le teardown(), nous avons choisi de faire une interface pour nos classes de test (TestCase) afin d'obliger les classes de tests à implémenter ces deux méthodes afin qu'elle s'exécute pour tous les tests de la même classe.

Pour les logs, nous avons choisi que transformer l'interface Log en classe abstraite, et nous avons ajouté une implémentation (la classe LogFile) pour gérer le nouveau type de sortie. Le choix de la classe abstraite (versus l'interface) constituait une solution plus efficace, nous permettant de voyager léger. Ainsi, les sous-classes n'ont qu'à implémenter seulement ce qui est nécessaire pour chacun des types de sortie.

1.5.1.2 Conception réussie

L'écriture dans un fichier fût plutôt simple à implanter, puisque nous avions déjà une interface qui permettait à notre classe principale qui exécute les tests de ne pas être couplée avec le mécanisme de log. Nous avons simplement changé pour une classe abstraite, après avoir implémenté la classe LogFile, il ne restait qu'à attraper le paramètre de la ligne de commande et à instancier le bon type de log.

1.5.1.3 Erreurs de conception et leçons apprises

On pourrait se demander si l'absence de l'interface TestCase constituait une erreur de conception. En fait, puisque notre stratégie était de faire le minimum pour répondre aux requis, à ce point du projet nous croyions que ce n'était pas une erreur à proprement parler.

1.5.2 Deuxième modification

Voici la liste de changements demandés :

- Possibilité de créer des suites de tests.

1.5.2.1 Design préconisé

Afin de permettre l'utilisation des suites de tests, nous avons d'abord créé la classe SingleTest, qui représentait le test à effectuer. Cette classe n'avait alors que deux attributs (le nom de la classe, et le nom de la méthode contenant le test). Au départ, nous avons placé un attribut ArrayList<SingleTest> dans notre classe principale. Mais nous avons tout de suite après effectué un « réusinage » alors que nous avons créé une classe Inventaire, implémentant l'interface Iterable de Java. Ainsi, on bouclait sur toutes les méthodes @test en remplissant l'inventaire. Le système capturait donc tous les tests à exécuter et après, les exécutait en affichant les résultats. Cette façon de fonctionner permettait de répondre au requis des suites de tests.

À ce moment du projet, comme nous avions du code fonctionnel et du temps pour le changer, nous avons choisi d'encapsuler les Assert de java afin de le découpler de notre classe principale. On a simplement offert dans cette classe une méthode assertEqual qui appelle l'assertEqual de Java.

1.5.2.2 Conception réussie

En raison de l'approche globale décrite plus haut, il nous a été très facile d'adapter notre code aux nouveaux requis.

1.5.2.3 Erreurs de conception et leçons apprises

Au départ, nous avons tenté d'utiliser les annotations pour définir les suites de tests. Cette piste s'est avérée mauvaise, puisque l'annotation java nous obligeait à insérer une déclaration immédiatement après, ce qui rendait impossible la syntaxe @suite suivie immédiatement par un @test.

Aussi, nous nous sommes aperçu que nous aurions dû faire une classe SingleTest bien avant, et que nous n'étions pas assez orienté objet dans notre conception initiale. C'est pour cette raison que nous avons choisi de faire la classe Inventaire et d'encapsuler le Assert tout de suite, afin de ne pas répéter cette erreur et d'avoir à subir un plus gros impact inutilement, advenant un nouveau requis sous la responsabilité de l'inventaire ou du Assert.

1.5.3 Troisième modification

Voici la liste de changements demandés :

- Pouvoir afficher la sortie en format XML ;
- Passer les paramètres à la ligne de commande.

1.5.3.1 Design préconisé

Pour répondre au besoin d'effectuer la sortie en format XML, nous devons générer le XML à la fin de l'exécution des tests, puisque pour offrir un tel format, il faut d'abord construire tout l'arbre du contenu de la réponse. Pour ce faire, nous avons choisi d'ajouter deux attributs dans notre classe représentant un test (SingleTest), testExecutionResult de type booléen qui indique si le test a réussi ou non, et testResultMessage pour conserver le message d'erreur dans le cas d'un test qui a échoué. Au moment de l'exécution, on capture les résultats dans les objets de type SingleTest et à la fin, nous générons le résultat d'un seul trait. La production de ces résultats, précédemment dans notre méthode main(), a fait l'objet d'un « réusinage » grâce au principe « Extraire la méthode ». En effet, nous avons extrait de la méthode launchTests() le code servant à l'affichage des tests, pour l'isoler dans une méthode à part, DisplayResult(). Cette nouvelle méthode boucle sur notre inventaire de tests et envoie en texte simple ou en format XML (selon le paramètre fourni) les résultats à notre instance de log qui s'occupe de gérer l'affichage à la console ou dans un fichier texte.

Pour permettre le traitement des paramètres à la ligne de commande nous avons fait l'ajout d'une nouvelle classe ClassTesterRunner dont le but est de permettre d'exécuter des tests via la ligne de commande. Cette classe contient seulement une méthode main() qui sert à recevoir les paramètres de la ligne de commande, à instancier le nouveau test pour ensuite l'exécuter, ainsi qu'une simple méthode de validation des paramètres reçus. Cette classe représente très bien le patron Facade puisqu'elle sert à exposer notre classe de test à la paramétrisation en utilisant la ligne de commande.

1.5.3.2 Conception réussie

Grâce à note interface Log, l'endroit du log (dans un fichier ou à la ligne de commande) n'a pas été impacté lors de notre ajout du format de Log (en XML).

Même si on effectuait l'affichage des résultats à mesure de l'exécution des tests, il nous a été relativement facile de conserver le résultat, puisque nous avions déjà un objet SingleTest, auquel nous avons simplement ajouté deux attributs.

1.5.3.3 Erreurs de conception et leçons apprises

Il a tout de même fallu revoir notre mécanique de traitement des résultats, puisqu'on ne pouvait pas de facto accumuler les résultats des tests.

L'attribut booléen de la classe SingleTest (testExecutionResult) a été implémenté pour rien, violant ainsi notre règle de ne jamais en faire trop. Un « réusinage » a été effectué à la dernière minute pour se débarrasser de cet attribut et de ses modificateurs/accesseurs.

1.6 Présentation UML de notre design final

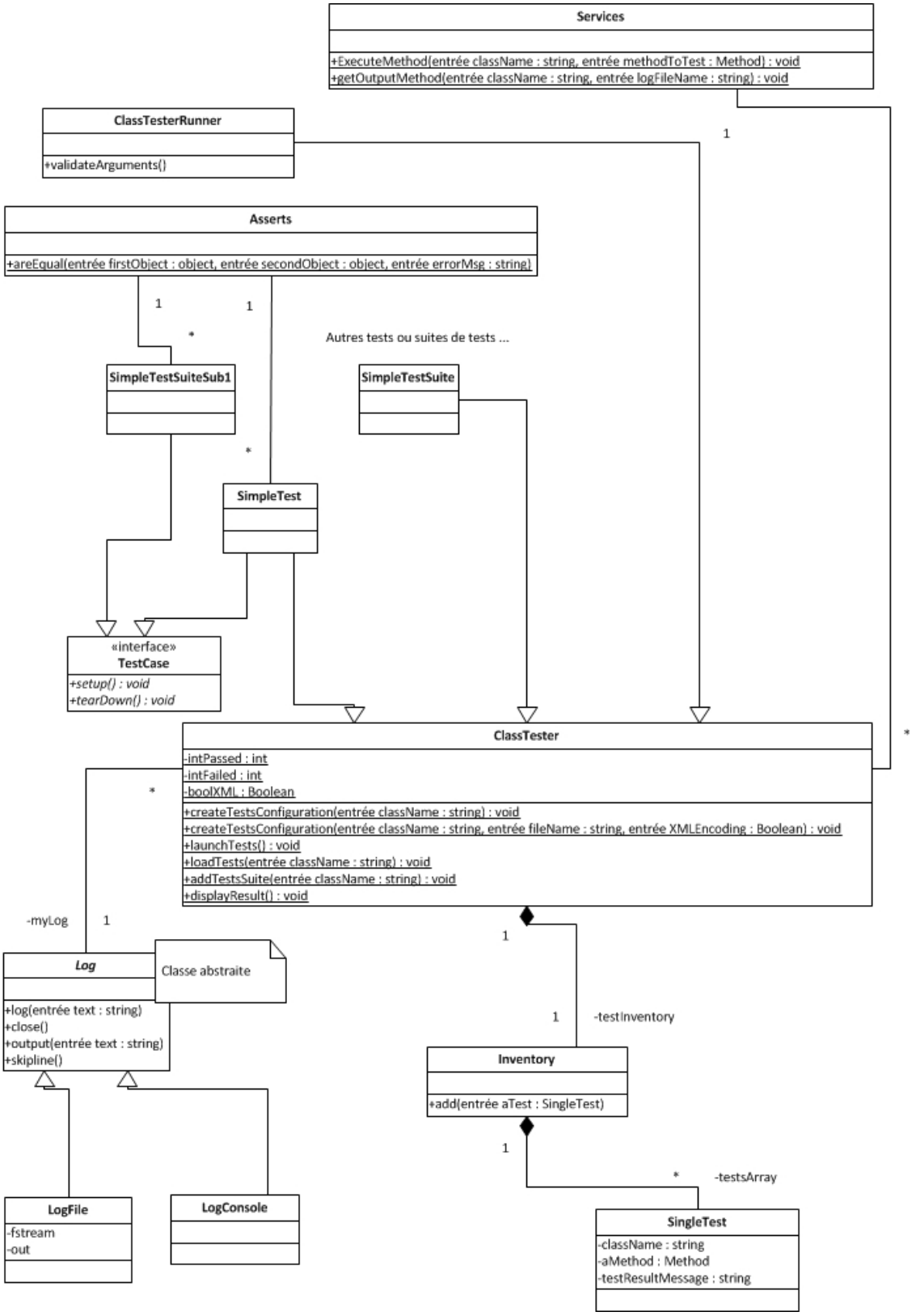

Figure 2 - Design final