UQÀM | Université du Québec à Montréal

Cours	MGL7460 – Réalisation et maintenance des logiciels
Session	Automne 2018
Nom de l'enseignant	Jacques Berger
Titre	Projet 1 – Évaluer la maintenabilité d'un logiciel
Groupe	040
Étudiants	Olivier Charrier (CHAO25098007) Maxens Manach (MANM14019309) Hugo Sanchez (SANH18029706) François Gratton (GRAF22097405) Marcio Elizeu (DEOM09057408)
Adresses de courriel	charrier.olivier@courrier.uqam.ca manach.maxens@courrier.uqam.ca hugo.sanchez@viacesi.fr gratton.francois@courrier.uqam.ca marcio.elizeu.ti@gmail.com
Date de remise	2018-10-11

UQÀM | Université du Québec à Montréal

Table des matières :

UQÀM | Université du Québec à Montréal

I) Introduction

Ce travail consiste à faire une évaluation détaillée de la maintenabilité du logiciel Pergen. Ce programme permet de générer, grâce à un fichier de spécification donné en entrée, un script SQL de création de tables et les classes Java DTO-DAO.

Pour analyser la maintenabilité du logiciel Pergen, nous choisirons les critères d'évaluation de la maintenabilité qui nous semblent les plus intéressants pour mener à bien cette étude. Dans le cadre de l'analyse de chacun de ces critères, trois volets seront abordés:

- La description du critère et la justification de son choix
- L'évaluation du logiciel Pergen en fonction du critère
- Les recommandations visant à améliorer la maintenabilité du logiciel en fonction du critère

Le code source du logiciel Pergen peut être divisé en 2 grandes parties: le code écrit par le programmeur et le code généré par SableCC. L'étude portera principalement sur le code écrit par le programmeur car c'est ce code qui pourra être modifié et refactorisé pour en augmenter la lisibilité et la maintenabilité. Dans le code généré par SableCC, on trouve du code très difficile à lire comme dans le fichier Lexer.java. Il contient, entre autre, une nomenclature peu verbeuse comme l'utilisation d'un tableau gotoTable[i][j][k][l] ou une boucle while de 326 lignes constituée de dizaines de boucles If, Else, For et Switch imbriquées qui pourrait être divisée en plusieurs fonctions pour plus de clarté.

Le problème au niveau de SableCC réside dans le fait que le vrai besoin de maintenabilité ne se trouve pas dans le code généré mais bien dans le code source du programme en tant que tel. Si on voulait que le code généré soit très lisible, il faudrait que le code source du programme SableCC soit beaucoup plus compliqué. Il devrait implémenter une forme de refactorisation automatique ce qui rendrait le programme moins facile à comprendre, tester et maintenir et pourrait occasionner plus d'erreurs. C'est un dilemme souvent présent dans les programmes qui génèrent du code comme les compilateurs.

II) Méthodologie :

Pour pouvoir analyser la maintenabilité du logiciel Pergen nous avons suivi la méthodologie suivante:

- Transfert des fichiers source avec git clone
- Inspection de tous les fichiers de code .java
- Rétro-ingénierie du code en diagramme de classe. outils: astah UML avec plugin "Easy Code Reverse" pour Java
- Installation de l'environnement de compilation: Maven, Java 1.8.0
- Installation de l'outil SonarQube 7.3 community
- Compilation de PerGen (avec tests, jacoco)
- Exécution de SonarQube
- Génération de la javadoc

III) Critères d'évaluation de la maintenabilité

1) Outil d'analyse Sonarqube et les code smells

Dans l'industrie du développement logiciel, ils nous arrive souvent d'avoir à maintenir le code d'un logiciel fait par une autre équipe ou une tout autre compagnie. Cette situation est généralement accompagnée d'une tâche d'évaluation de la maintenabilité du code. De façon générale, plusieurs évaluent un logiciel en essayant l'interface utilisateur, en comptant le nombre de fonctionnalités et en essayant de deviner leurs complexités. Certains vont probablement lire quelques fichiers de code mais pas plus car il serait impensable de lire tous les fichiers de code d'un logiciel de taille moyenne qui pourrait comporter quelques dizaines de milliers de lignes de code. Étant donné que nous sommes des professionnels, il est dans notre devoir de fournir une évaluation à la hauteur et d'appuyer notre analyse et nos résultats avec des mesures et des métriques. Pour ce faire, nous n'avons pas d'autre choix que d'utiliser des outils d'analyse statique et dynamique de code.

Étant donné la durée et la portée de ce travail, cette partie couvrira uniquement l'utilisation d'un outil d'analyse statique de code comme critère d'évaluation de la maintenabilité d'un logiciel. L'outil en question est SonarQube, un outils de type intégration continue, car il s'intègre facilement à l'outil de compilation Maven et permet d'avoir un suivi des métriques pour chacune des versions déployées. Cet outil comporte plusieurs mesures regroupés sous différentes rubriques telles que la complexité, la taille, la duplication, la couverture, la maintenabilité, etc.

Une des métriques intéressante fait partie de la rubrique *Maintenabilité* et est intitulée "code smells". La documentation de SonarQube décrit ce concept de qualité comme étant des problèmes reliés à la maintenabilité dans le code. Ces problèmes sont relevés à partir de règles de style, des bonnes pratiques de codage ou par exemple le dépassement de valeurs mesurées par rapport à une valeur limite prédéfinie. Lorsque l'analyse est complété, les problèmes sont classifiés selon leur sévérité: bloquante, critique, majeure ou mineure. Cette métrique permet d'avoir une vue d'ensemble des problèmes reliés à la maintenabilité du code, et offre une valeur qualitative aux différentes règles et mesures.

Évaluation des résultats

L'analyse statique du code révèle initialement un total de 508 code smells. De ces problèmes, 40 sont bloquants, 33 sont critiques, 144 sont majeurs et 291 sont mineurs. Cependant, le code auto généré par SableCC ne devrait pas faire partie de l'analyse comme expliqué dans l'introduction du document. Par conséquent, en excluant les fichiers de code sous le répertoire "generated", l'analyse exclus 434 code smells, et affiche maintenant 73 problèmes de type code smells. L'inspection de ces problèmes a révélé que 42 d'entre eux n'étaient pas réellement problématique. Par exemple, 24 d'entre eux demandent de définir une constante pour certaines chaînes de caractères dans Java6Provider et MySql5Provider. 14 de ces problèmes portent sur java 7 et l'opérateur "<>". Ceci est simplement un nouveau moyen d'instancier des objets génériques sans avoir à tapper le type générique des deux côtés de l'assignation. Finalement, on retrouve deux problèmes qui nous paraissent mineur et qui porte sur l'utilisation d'un caractère au lieux d'une chaîne de caractère dans la

méthode “lastIndexOf(...)”. Nous proposons donc de mettre la mention “Won’t fix” pour ces 42 problèmes.

En excluant ces 42 problèmes, on se retrouve avec 31 problèmes code smells dont un est critique, 14 sont majeurs et 16 sont mineurs. SonarQube nous offre une mesure de synthèse pour régler les problèmes code smells, sous la rubrique *Maintainability*. L’outils évalue l’effort de résolution de ces problèmes à 5 heures 18 minutes,

Recommandations
La recommandation serait de régler les 31 problèmes restants, comme par exemple la refactorisation de la méthode “provideDAOSavemethod” totalisant 25 en complexité cognitive pour ne pas dépasser la valeur acceptée de 15 et la refactorisation de deux blocs de code dupliqué dans “provideDAOGetMethod” et “provideDAOGetAllMethod”.

2) Lignes de code(LOC)

Lorsqu’il s’agit de maintenir un système, une des première tâche consiste à évaluer sa taille, son importance, ainsi que l’effort consenti pour comprendre le code. De cette information, croisée avec d’autres, nous pouvons estimer le coût et proposer des processus de maintenance. C’est pourquoi il est crucial de déterminer le nombre de lignes de code ainsi que le temps moyen pour comprendre ledit code.

L’importance de cette métrique pour évaluer la maintenabilité d’un système peut être démontrée de la manière suivante :

Soit deux systèmes dont les métriques de complexité sont similaires, et dont la différence principale réside dans le nombre de lignes logiques, terme qui sera détaillé ci-après. Un des systèmes possède 1000 lignes de code logiques, pendant que l’autre n’en présente que 10. Dans ce cas, il faudrait déployer 100 fois plus d’efforts pour comprendre le premier système, ce qui conduirait à une allocation des ressources bien différente dans les deux cas. Compter les lignes est donc fondamental.[1]

PerGen

Total Lines	3080
Blank Lines	459
Comments Whole	931
Physical SLOC	1690
Logical SLOC	1083

Ces métriques ont été obtenues à l’aide du UCC - Unified CodeCount de l’université de Southern California. Avec un compte de 1690 lignes de code source physique et 1083 lignes de code source logique, on peut considérer que le logiciel est de petite taille et que sa maintenance devrait donc en être facilité. Il n’y a pas de recommandation en lien avec ce critère.

3) Temps moyen de compréhension

Étroitement lié au nombre de lignes dans le projet, la métrique temps moyen de compréhension d'un bout de code peut nous donner une estimation du temps nécessaire pour comprendre le fonctionnement d'un logiciel.

Afin de mesurer le temps moyen de compréhension d'un morceau de code, on isole le nombre moyen de lignes logiques par classe, qui s'avère être 34. On sélectionne ensuite les classes de taille similaire à cette moyenne, et on évalue le temps de compréhension pour chacune. Les classes ainsi évaluées représentent 14.7% du projet. On peut donc extrapoler de ces résultats un temps de compréhension plus global.

Logical SLOC	Fichier	Temps de compréhension
40	./domain/Field.java	0:27
37	./explorers/RelationExplorer.java	0:48
30	./main/InputFileParser.java	1:27
28	./main/PerGen.java	1:32
25	./domain/Relation.java	0:19
160	TOTAL	273 secondes
	Moyen (sec/lignes)	0:01.7
	Pire cas (sec/lignes)	0:03.3

Comme indiqué dans le tableau ci-dessus, un programmeur compétent dans les technologies utilisées dans le projet devrait pouvoir comprendre le logiciel en seulement une heure environ. Cette donnée tend à faire croire que la maintenabilité du logiciel ne devrait pas être trop difficile. Il n'y a pas de recommandation en lien avec ce critère.

4) Uniformité du code

L'uniformité du code est un critère subjectif, mais aussi très important en ce qui a trait à la lisibilité du code et donc à sa maintenance facile. Généralement le code est fait par plusieurs développeurs et la lecture du code est beaucoup plus facile lorsque tout le code a un le style uniforme ou au minimum, lorsqu'il est uniforme pour chaque développeur. Si l'uniformité est absente, le code sera plus difficilement assimilable et cela démontre un manque de professionnalisme de l'équipe qui a conçu le logiciel.

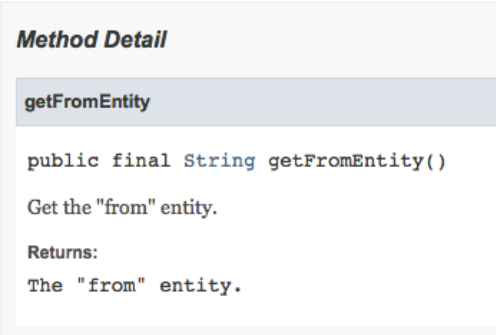
Pour évaluer ce critère, tous les fichiers .js du code, excluant ceux auto générés, ont été ouverts et inspectés. Les critères portent sur des différences de style notable, par exemple la position de l'accolade ouvrante, l'indentation, les commentaires javadoc, etc.

Évaluation

Lors de l'inspection des fichiers de code, les points suivants ont été notés: l'indentation est faite aussi bien avec des espaces qu'avec des tabulations, ces différences ont même été trouvés dans un même fichier. Ce problème n'est généralement pas remarqué par l'équipe de développement car ils utilisent le même éditeur de code et qu'il est configuré de la même façon. Cependant lorsque le code est ouvert dans un éditeur configuré différemment, par exemple avec un nombre d'espaces différent pour la tabulation, la structure d'indentation du code est brisé et il devient impossible de le lire. Il nous faut alors configurer l'éditeur de façon à voir le code comme il a été écrit initialement, mais aussi de retourner à notre configuration initial pour nos propres besoins.

Nous avons aussi remarqué l'utilisation exhaustive des commentaires javadoc dans certains fichiers et dans d'autres, pas du tout. Par exemple, tous les fichiers avec un copyright 2011 et 2012 n'ont pas de javadoc tandis que certains fichiers avec un copyright 2007, en sont saturé.

Dans les fichiers avec des commentaires javadoc, on remarque des commentaires inutiles et qui polluent donc ce code. C'est le cas dans le commentaire de certaines méthodes ou la description est identique au nom de la méthode. Voici un exemple tiré du fichier RawRelations.js:



Méthode: getFromEntity, commentaire javadoc: get the "from" entity, returns: the "from" entity.

Autre point, la longueur maximum d'une ligne de code dans la majorité des fichiers est de 80 caractères mais on retrouve quelques lignes ou cette limite n'est pas respectée.

Recommandations

Les recommandations proposées visent à réduire les problèmes discutés précédemment. Nous pensons qu'en général, l'uniformité du code ne devrait pas être un grand frein à la maintenabilité du logiciel Pergen. Voici quand même les tâches proposé pour augmenter l'uniformité du code:

- Remplacer les tabulation par des espaces
- Enlever les commentaires javadoc dupliquant presque textuellement le nom des méthodes
- Ne pas corriger les lignes de code dépassant 80 caractères, mais spécifier dans une règle de style qu'on ne peut avoir plus de 80 caractères sur une ligne

5) Qualité et couverture des tests

Dans l'industrie comme dans le monde universitaire, il est communément admis que les tests sont une "*bonne pratique*". On sacralise le test unitaire, et on le présente comme une sorte de panacée qui protégerait le développeur contre des modifications irréfléchies. En testant, le développeur obtient l'assurance que son code fonctionne. Mieux encore, il voit en temps réel l'impact des modifications qu'il s'apprête à partager avec ses pairs.

Mais les tests unitaires ne sont-ils qu'un outil de plus visant à éviter au programmeur l'humiliation d'avoir "cassé" le code ? Ou sont-ils plus que cela, un indicateur pertinent de la maintenabilité et de la qualité du logiciel ? Cette partie du rapport s'intéressera à la couverture de test, notamment des tests unitaires, et à la qualité desdits tests, en tant que mesure plus subjective.

Justification des critères

La norme ISO 25010, promulguée par l'International Organization for Standardisation, qui traite des Exigences de Qualité et de leur évaluation pour les Systèmes Logiciels nous apprend que :

- S'assurer de la qualité d'un logiciel est primordial, dans un monde où l'on développe des logiciels en réponse à un nombre croissant de problèmes complexes.
- Des spécifications et une évaluation pertinente de ladite qualité permettent au logiciel d'avoir plus de valeur auprès des parties prenantes. Les développeurs comme les clients étant concernés par cette appellation.
- La maintenabilité fait partie des critères d'évaluation pour déterminer la qualité d'un logiciel.

De ces informations sommaires, on peut déduire que dans le cadre du développement logiciel, on devrait aspirer à rendre le code maintenable. Peu surprenante au premier abord, cette constatation nous pousse tout de même à nous interroger sur la notion de "maintenabilité". Un programme maintenable est un programme qui peut être aisément modifié afin de mieux remplir sa mission. Les améliorations, les corrections et les ajouts de nouvelles fonctionnalités devraient être facilités. Cinq caractéristiques découlant de ce postulat devraient être appliquées à tous les logiciels qui se veulent de qualité :

- **La modularité** : à quel point les différents composants de notre application sont-ils indépendants, et quel est l'impact d'une modification dans l'un d'eux sur les autres modules ?
- **La réutilisabilité** : lesdits modules sont ils réutilisables dans d'autres contextes, ou sont-ils dépendants de l'environnement spécifique de notre application ?
- **L'analysabilité** : lorsqu'une erreur survient, à quelle vitesse décèle-t-on la nature du problème, et avec quelle facilité retrouve-t-on sa source ?
- **La modifiabilité** : à quel point peut-on modifier le logiciel sans avoir des effets indésirables sur la qualité du développement ?
- **La testabilité** : enfin, peut-on facilement déterminer des critères objectifs de test pour notre solution, et peut-on facilement évaluer ces critères tout au long du processus de développement ?

Deux de ces points nous intéressent particulièrement ici. Nominativement, l'analysabilité et la testabilité du produit. Lorsque l'on s'intéresse à PerGen en tant qu'application, du point de

vue des tests unitaires, on peut remarquer une chose : des tests existent, et ils font partie intégrante du projet. Nous allons donc pouvoir étudier l'application sous cet angle et évaluer de manière pertinente l'impact des tests actuels sur notre capacité à analyser le code et à tester le fonctionnement du projet.

Robert C. Martin nous apporte dans "Clean Code" la preuve empirique que les tests sont une bonne pratique, si et seulement si ils sont aussi bien écrits et maintenus que le reste du code. Nous suivrons cette approche, et les enseignements du chapitre portant sur les tests unitaires pour déterminer si le jeu de test de PerGen sert ou dessert la maintenabilité de l'application. Nous évaluerons par là-même la qualité desdits tests, par rapport à des critères personnels et du point de vue de développeurs plus expérimentés.

"Le code de test est aussi important que le code de production. […] Il nécessite de la réflexion, de la conception et de l'entretien. Il doit être gardé aussi propre que le code de production peut l'être.[2]"

- Robert C. Martin

Évaluation de la maintenabilité

Dans un premier temps, nous avons utilisé le plugin Maven de JaCoCo (Java Code Coverage) afin d'analyser le code de PerGen, et d'en déterminer les caractéristiques, par le prisme des statistiques de tests.

Cette analyse statique nous révèle immédiatement un certain nombre d'éléments. Entre autres, la couverture du code n'est pas totale, et ce, que l'on s'intéresse aux lignes, aux méthodes, ou même aux classes. Il nous faut nuancer ici notre propos, car JaCoCo s'est également attaqué au code généré, ce qui fait fortement pencher la balance. Mais même en faisant abstraction des packages qui n'ont pas été développés par un humain, l'on constate un taux de couverture qui atteint seulement 64.3% des méthodes.

Un total de 62 fonctions ne sont pas testées unitairement, ce qui implique que des modifications ultérieures pourraient impacter toutes ces méthodes sans que l'on ne s'en rende compte. De plus, on peut voir que dans certains packages, la couverture descend à 0%. Du point de vue de la maintenabilité, on peut relever deux problèmes majeurs :

- Tout ajout ou modification du code sera plus risqué pour le développeur, car il fera face à une absence de certitude concernant le fonctionnement global du programme. Cette incertitude aura un impact négatif sur la productivité de quiconque cherchera à reprendre le projet. En effet, en rendant les modifications plus "risquées", on encourage le développeur à innover le moins possible.
- En cas de dysfonctionnement, il sera plus difficile voire impossible de localiser avec précision la source du bug ou du problème. En l'absence de couverture totale, on ne peut affirmer avec certitude que telle partie du code est la source de nos maux. En d'autres termes, le temps passé à traquer les bugs sera démultiplié.

Une étude datant de 2017 réalisée par Codacy -un fournisseur de métriques automatisées relatifs au code- nous informe que les développeurs qui mettent en place des stratégies de contrôle de la couverture de code perçoivent une amélioration (de l'ordre de 8%) de la qualité de leur code. Ces développeurs sont plus enclins à décrire leur code comme "de qualité", et s'évitent par là même une appréhension considérable lorsqu'il s'agit de modifier le code existant.

Il a été relevé au sein de PerGen, au cours d'une review comparative des classes et des classes de test correspondantes, que certaines classes ne respectaient pas la nomenclature utilisée la plupart du temps. Un exemple parlant est celui de la classe de test "MockFileWriter" qui importe et teste la classe "FileWriterWrapper". Si le "mock" au début du nom peut être excusé -bien qu'il diminue grandement la facilité avec laquelle on trouve la classe correspondante au sein du projet-, il n'y a en revanche aucune raison d'omettre le "Wrapper" à la fin.

La classe testée ici est bel et bien "FileWriterWrapper" et non pas "FileWriter". Nommer la classe de test de manière non-orthodoxe oblige le développeur à ouvrir la classe pour voir de quel jeu de tests dont elle dépend. De plus, elle apporte une confusion, en poussant le développeur à penser que c'est la classe java originelle qui est testée ici. Alors qu'en réalité, il s'agit non pas de tests unitaires mais bien d'un cas d'utilisation que l'on fait tourner afin de s'assurer du fonctionnement du composant. Une source d'imbroglio supplémentaire pour celui qui sera amené à travailler sur le projet.

Une fois ces éléments objectifs repérés, une relecture supplémentaire des classes de tests a permis de déceler plusieurs incohérences, plus subjectives et qui nous serviront donc à évaluer la qualité desdits tests.

```
public final Field getField(final
    return fields.get(originalFie
}

public final void addUnicityConst
    unicityList.add(constraint);
}

public final void addRelation(fin
    relations.add(relation);
}

public String getSqlName() {
    return sqlName;
}

public final Collection<Field> ge
    return fields.values();
}

public final Collection<Relation>
    return relations;
}

public final Collection<UnicityCo
    return unicityList;
}

public final String getJavaName()
    return javaName;
}
```

```
@Test
public final void testGetField() {
    Entity entity = new Entity("Rain");
    String fieldName = "field";
    Field field = new Field(fieldName,
            FieldType.Type.INTEGER, true)

    entity.addField(field);
    assertSame(field, entity.getField(fie
}

@Test
public final void testGetUndefinedField()
    Entity entity = new Entity("Rain");
    assertNull(entity.getField("broken"))
}

@Test
public final void testSqlName() {
    Entity entity = new Entity("entity_na
    assertEquals("ENTITY_NAME", entity.ge
}

@Test
public final void testJavaName() {
    Entity entity = new Entity("entity_na
    assertEquals("EntityName", entity.get
}

@Test(expected = AmbiguousFieldNameExcept
public final void testJavaNameAmbiguity()
    Entity entity = new Entity("entity_na
    Field field1 = new Field("field_name"
```

Dans l'exemple ci-dessus, tiré de la classe entité, on peut voir que le nommage des fonctions de test n'est pas uniforme. On ne peut pas en un seul regard lier les tests à leurs fonctions de références, ce qui ralentit la lecture et la compréhension des classes. Cette erreur est répétée à de nombreuses reprises au sein du code.

Les tests existants sont relativement bien placés, notamment sur les fonctions les plus importantes du programmes. Les classes centrales ont reçu un niveau d'attention suffisant, à l'exception notable du package "main" qui n'est pas testé. Toutes les fonctions dépendant de ce point d'entrée, il serait pertinent de mettre en place un test, afin de garantir un fonctionnement irréprochable, ou, à tout le moins, une chasse au bugs plus aisée.

On peut aussi s'attarder sur le code des tests en lui-même, qui présente de nombreux défauts d'uniformité et de style. Accolades flottantes, indentations hasardeuses et bizarreries stylistiques sont légions dans les classes de tests. Certaines classes semblent avoir été soignées et re-travaillées pendant que d'autres présentent un aspect désolé. Il est également important de noter que deux manières de nommer les tests coexistent au sein de l'application. Une manière descriptive, et une manière usuelle.

- **Descriptive** : les tests sont nommés d'après l'action ou le cas d'utilisation. Par exemple, un test nommé "testCamelCaseTwoWords" s'assure que la fonction gère bien un nom composé de deux mots, en l'adaptant correctement avec la bonne casse.
- **Usuelle** : les tests sont nommés d'après les fonctions qu'ils testent (avec comme exceptions les erreurs que nous avons relevées plus haut).

Bien que les deux manières de faire soient tout à fait correctes, il faut comprendre que la cohabitation des deux obscurcit le code et fait perdre du temps au développeurs qui serait amené à retravailler les tests.

Recommandations

Nous avons soulevé dans la partie précédente que la couverture de tests était parcellaire au mieux, voire complètement absente au sein de certains packages, et que même lorsque les tests avaient été développés, ils présentaient des incohérences stylistiques, une nomenclature bâclée et semblaient ne pas avoir été retravaillés pour la plupart. Forts de ces informations, nous pouvons proposer deux types de mesures afin d'améliorer la couverture et d'augmenter la qualité des tests.

Mesures curatives : ces mesures visent à retravailler directement le projet PerGen afin de corriger les défauts repérés et d'améliorer rapidement et à court terme la maintenabilité du code.

- <u>Déplacer ce qui ne relève pas des tests unitaires dans l'arborescence :</u>

D'autres types de tests que les tests unitaires côtoient ces derniers au sein même des classes de tests du projet. Il est très difficile pour un développeur de les différencier lorsqu'il relit le code, et les trouver au sein des nombreuses classes relève du défi. Il serait pertinent de les séparer au sein de l'arborescence du projet, afin de permettre aux développeurs d'analyser en un coup d'œil PerGen et de voir si des tests sont présents.

- <u>Créer de nouveau types de tests :</u>

La présence de tests unitaires est une bonne chose, et comme nous l'avons dit, d'autres types de tests sont présents. Cependant, force est de constater qu'ils se font rares. Il serait bénéfique pour la maintenabilité du code de développer également des tests d'intégration ainsi que des tests de validation, afin d'assurer une automatisation complète du processus.

Un test qui n'est pas automatisé est une perte de temps, car les développeurs n'ont que rarement l'occasion de se lancer dans une série de tests non-automatisés, ce qui les rend

quasiment inutiles. Il est donc important d'étudier la question et de mettre en place, en complément des tests unitaires d'autres jeux de tests automatisés capables d'évaluer le fonctionnement de l'application au-delà du simple scope des fonctions prises séparément.

- Refactorer les tests existants :

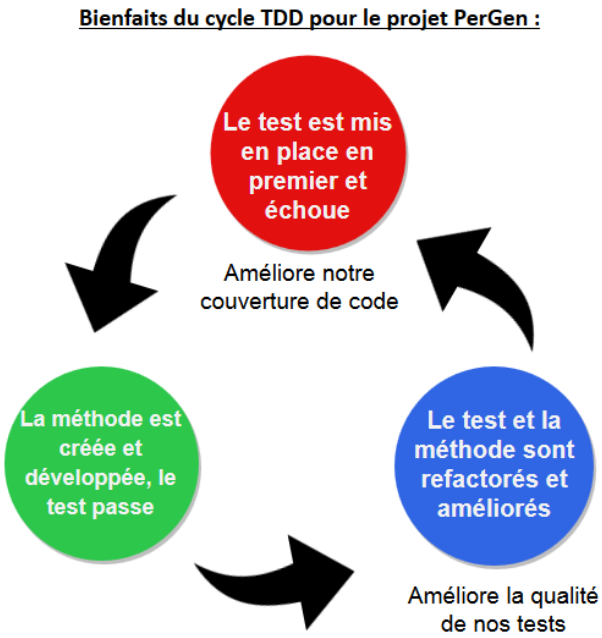
Les tests existants sont comme on l'a vu desservis par une nuée de défauts qui les rendent difficile à comprendre et à développer. En se basant sur des critères qui sont la lisibilité et la compréhension, il faut chercher à améliorer les tests déjà en place. Cela passera par le nettoyage du code, notamment stylistique, ainsi que par l'ajout de tests pour les fonctions qui ne sont pas couvertes dans notre code. À minima, afin de couvrir toutes les fonctions considérées comme essentielles, ainsi que des classes les plus "centrales".

Dans un second temps, il sera intéressant pour l'équipe de créer des tests situationnels qui n'évaluent qu'un seul concept. Ce qui signifie qu'il faudra parfois déployer plusieurs tests pour une seule et même fonction, dans l'optique de couvrir tous les concepts présents dans cette méthode, mais aussi et surtout afin de faciliter la compréhension par l'humain.

Mesures proactives : Ces mesures de développement visent à améliorer la maintenabilité du code futur, et sur le long terme.

- Appliquer désormais les principes du Test Driven Development :

Le manque de couverture de code peut être expliqué par le fait que les tests n'ont pas été développés au même rythme que les fonctionnalités attendues. Un bon moyen de remédier à cela pour de bon est de mettre en place une approche TDD. En créant les tests avant de programmer les fonctions, l'on s'assure que notre couverture sera optimale. Cette approche pousse aussi les développeurs à accorder plus d'importance aux tests, et par conséquent améliore leur qualité.



- Définir une nomenclature et un style pour les tests et s'y tenir :

Dans l'optique de réduire les disparités au sein du code, il serait intéressant de définir, pour tous les développeurs amenés à participer au projet, un style commun. Faire cet effort

permettra de gagner un temps considérable, à la fois lors de l'écriture mais aussi lors de la relecture du code de nos tests. Un style homogène fait actuellement défaut et serait un atout majeur pour améliorer la maintenabilité du code.

6) Documentation

La documentation logicielle est une notion importante lorsqu'il est question d'évaluer la maintenabilité d'un programme. La documentation sous toutes ses formes, pourvu qu'elle soit pertinente, peut permettre à un programmeur de maintenir plus facilement un logiciel car elle répond aux questions et facilite et accélère la compréhension d'un système. Pour le travail, nous divisons cette notion de documentation en trois grandes parties: la documentation technique ayant but d'expliquer comment le logiciel fonctionne, les commentaires du code source et finalement la documentation généré automatiquement qui permet entre autre de documenter, d'ordonner et de lister les classes, les méthodes, les variables ainsi que les commentaires si rapportant.

La documentation technique

Il n'existe pas de documentation écrite pour le logiciel Pergen mis à part le fichier README du repository qui explique en une seule phrase le but du logiciel et la liste des technologies utilisées. C'est dommage car ce logiciel, qui se veut académique, aurait profité d'un petit texte technique qui aurait mis en lumière le fonctionnement et les particularités du système. Il aurait été intéressant, surtout pour un programmeur novice, d'avoir accès à une explication des points importants à connaître au niveau du fonctionnement du logiciel, des intrants et des extrants ou des classes ou des fichiers qui devrait être évoqué comme par exemple le fichier Grammar. Ce court texte n'a pas à être tenu dans un nouveau document. Il peut être placé dans le fichier README du projet proposé par Github. Ce fichier permet même de gérer efficacement, et ce depuis quelque temps, les liens hypertextes. On peut donc mettre des liens vers par exemple la Javadoc du projet ou vers une classe ou un fichiers du code.

Les commentaires du code source

Le code source est légérement commenté pour la javadoc mais ils existent de façon exhaustive dans certains fichiers (copyright 2007) et dans d'autres, pas du tout (copyright 2011 et 2012). Il n'y a pas de commentaire interne dans les fonctions pour en comprendre le fonctionnement. Pour ce type de commentaire on est au niveau du style peu c'est mieux. Dans les fichiers avec des commentaires javadoc, on remarque des commentaires inutiles et qui polluent le code. Nous en avons parlé dans la section Uniformité du code.

Documentation généré automatiquement

La Javadoc du projet est un outil intéressant pour faciliter la maintenance du logiciel Pergen. Il permet en quelques clics d'obtenir des informations importantes pour faciliter la compréhension du système ou par exemple faciliter une modification du code source lors d'une maintenance. Vu qu'il est auto généré il a aussi tendance à être complet. Puisque le logiciel est académique, nous proposons de la rendre accessible directement dans le répertoire du code source et par lien dans le fichier README.

Recommandation
Bonifier le fichier README avec une explication du fonctionnement du logiciel et de ses particularités et explication pour les intrants, extrants, classes, méthodes ou fichiers qui méritent des précisions et qui permettront à un développeur de mieux cerner le logiciel. Rendre accessible la Javadoc et la proposer dans le fichier README.

7) Complexité cyclomatique et cognitive

Complexité cyclomatique

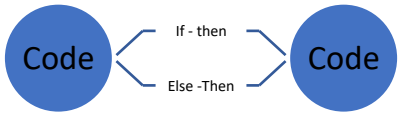
Une des métriques formelles sur laquelle on peut se pencher pour étudier la maintenabilité d'un logiciel est sa complexité cyclomatique qui comme son nom le laisse deviner, sert à donner un degré de « complexité » d'un programme.

Créée par McCabe dans les années 1970 [4], elle a pour but de mesurer le nombre de chemins indépendants à travers lesquels on peut exécuter un code source. Cette mesure peut être calculée sur divers éléments d'un programme tels que des méthodes ou des classes.

Concrètement, cette complexité peut-être simplement calculée à partir d'un graphe d'un programme et de la formule suivante :

Complexité Cyclomatique v(G) = Nombre de liens – Nombre de nœuds + 2

Sur l'exemple basique qui suit, nous avons effectivement une complexité cyclomatique de 2-2+2 = 2 qui correspond au nombre de chemins indépendants différents empruntables dans cette méthode.



Une méthode possède donc toujours une complexité minimum de 1, et si elle est égale à 1 alors la méthode ne contient pas de structure logique et est une séquence d'actions. 20 ans après la parution de son article sur la complexité cyclomatique, McCabe a approfondi le sujet en confirmant qu'il fallait conseiller au programmeur de diviser son module en deux dès que ce dernier dépassait les 10 de complexité cyclomatique [5]. Dans les faits, on peut retrouver sur des forums de discussion de développeurs les chiffres suivants pour estimer de la qualité d'un code :

Complexité cyclomatique d'une méthode	Qualité
1 à 4	Bonne
5 à 7	OK
8 à 10	Moyen
11 et plus	Méthode à réusiner

Enfin, puisque ce nombre représente le nombre de chemins possible dans un programme, il peut également servir pour savoir le nombre de tests unitaires nécessaire à la couverture complète du programme.

Pour notre programme, après analyse avec Sonarqube nous obtenons les chiffres suivants :

Complexité cyclomatique	Nombre de méthodes concernées
1 à 4	144
5 à 7	11
8 à 10	1
11 et plus	1

D'une manière générale, la complexité du code est excellente. Il n'y a vraiment qu'une méthode nommée ProvideDAOSaveMethod dans la classe Java6Provider qui est en revanche mauvaise de ce côté là puisqu'elle possède une complexité cyclomatique de 19.

La complexité cyclomatique d'une classe en revanche est considérée comme étant une mesure obsolète puisque peu représentative de la nature d'une classe.

Complexité cognitive

Toutefois, en se renseignant sur le critère qu'est la complexité cyclomatique, il apparaît que cette mesure n'est pas exemptée de toute critique puisque selon certains elle n'est pas parfaite d'un point de vue maintenabilité. L'équipe de développement Sonarsource, à l'origine de Sonarqube à par exemple décidé de créer sa propre mesure de complexité (complexité cognitive), et s'en justifie en disant que la complexité cyclomatique n'est pas très juste dans sa manière d'être incrémentée, en ne considérant pas assez la nature complexe d'une imbrication de structure logiques (if, while, …) ou en considérant trop complexe le cas des switch/case par exemple.

Leur complexité cognitive se veut représentative de la complexité réelle d'une méthode, et veut également donner plus de signification à la complexité d'une classe. Leur idée est donc de faire passer la complexité minimale d'une méthode est de 0 et non 1. Ainsi, cela leur permet de juger plus réellement la complexité d'une classe, qu'elle soit composée de 10 méthodes très basiques de complexité 0 (la classe sera donc de complexité 0 également) ou bien d'une seule méthode de complexité 10 (la classe sera donc de complexité 10 également) quand avec la complexité cyclomatique elles ces deux classes seraient considérées aussi complexe l'une que l'autre.

Pour ce qui est de la qualité de cette mesure dans le code, il est impossible de trouver un bon « barème » à l'inverse de la cyclomatique. Toutefois la règle de base de Sonarqube est de réusiner si la complexité cyclomatique est supérieure à 15. Ce qui est le cas d'une seule méthode, la même que précédemment. Même en abaissant arbitrairement la limite acceptable de 15 à 10, on ne trouve qu'au total 3 méthodes au-dessus de ce nombre, on peut donc en déduire que le code est de manière générale à la fois facilement testable (cyclomatique) et compréhensible (cognitive).

Recommandations

La seule recommandation possible pour ce programme serait de réusiner la méthode provideDAOSaveMethod de manière à la rendre moins complexe (et longue par la même

occasion), ce qui est en lien avec les codesmells trouvés dans la première partie avec Sonarqube.

Pour être plus « propre », on pourrait suggérer de couper cette méthode (qui paraît assez procédurale dans son fonctionnement) en 3 sous-méthodes, par exemple après certaines des boucles for. Une des solutions serait la suivante :

ProvideDAOSaveMethod1 : Jusqu'à la boucle for ligne 653 (Complexité de 7)
ProvideDAOSaveMethod2 : Jusqu'à la boucle for ligne 691 (Complexité de 5)
ProvideDAOSaveMethod3 : Jusqu'à la fin de la méthode (Complexité de 7)

8) Qualité de la Nomenclature

Nous avons également fait le choix d'analyser la qualité de la nomenclature du projet. Il s'agit d'un critère subjectif puisque les termes employés pour décrire une classe, méthode, attribut ou autre peut convenir à un développeur mais pas à un autre. Il existe toutefois, indépendamment du vocabulaire choisi, des lignes directrices données par Sun dans la documentation pour tout ce qui est trait à l'aspect esthétique de la nomenclature [7]. On peut par exemple relever parmi ces indications certaines règles intéressantes :

- Classes : Utiliser des noms, qui respectent le « Camel Case » qui revient à mettre une majuscule au début de chacun des mots utilisés. Il est conseillé également d'utiliser des noms clairs et descriptifs du rôle de la classe, en évitant d'utiliser des abréviations.
- Méthodes : Utiliser des verbes pour décrire clairement la fonction supposée d'une méthode. Pour ce qui est de l'esthétique, utiliser le Camel Case sauf pour la première lettre de la méthode.
- Variables : Plus que pour tout autre élément, le nom de variable doit être court et clair, afin de permettre à la lecture de ce dernier de savoir directement le rôle d'une variable, qui peut être délicat à déterminer sinon. Ne pas utiliser de caractère unique comme nom, sauf pour des variables temporaires comme i,j, … pour les boucles et autre.
- Constantes : N'utiliser que des majuscules et des underscores afin de faire la distinction avec les variables.

La question du choix de la langue utilisée n'est pas soulevée par Oracle, et c'est ici une question de préférences mais on aurait le droit de penser qu'afin de rendre son logiciel le plus accessible à tous, et donc le plus maintenable possible, il serait de bon ton d'employer l'anglais pour nommer ses éléments.

Dans le cadre du projet, nous pouvons dans un premier temps utiliser CheckStyle pour vérifier que tous les noms utilisés, indépendamment du vocabulaire utilisé, respectent la convention de Sun. Après paramétrage du plugin, il n'apparaît que ces règles sont respectées sur tous les noms, à l'exception de la méthode getAllMANYRelations() dans la classe Entity.java.

Ceci apparaît comme étant une erreur puisqu'il y a utilisation abusive de majuscules pour écrire un mot, toutefois le MANY ici désigne un type énuméré « MANY » défini dans RelationType.java, par conséquent nous pouvons ignorer cette erreur et en déduire que le projet respecte parfaitement les directives de Sun.

Pour ce qui est de la vérification de la logique derrière les noms de variable choisis, aucun outil ne le permet. Il faut donc faire une liste de toutes les noms donnés par le développeur et regarder si non seulement la logique derrière le nommage des éléments est de bonne qualité, mais surtout si les noms sont donnés de manière uniforme (toutes les méthodes commencent par un verbe, …).

N'ayant trouvé d'outil logiciel permettant de lister tous les noms créés, une lecture de plusieurs classes du projet permet de dire que de manière générale la logique est bonne derrière le nommage. Toutefois comme ceci n'est pas objectif, c'est avant tout une question de préférences et il faut tout de même passer un peu de temps à comprendre certains aspects du programme pour comprendre les noms. Par exemple, dans le constructeur de la classe relation, les affectations sont les suivantes :

entity = entityInfo;
type = relationType;
maybeZero = canBeZero;
isManyToMany = manyToMany;
nameOfLinkTable = tableLink;

Il s'agit finalement plus d'un problème de compréhension du code que de nommage, mais il est intéressant de noter que pour comprendre ce constructeur, il faut explorer plus en détail le code que de simplement lire l'affectation des variables.

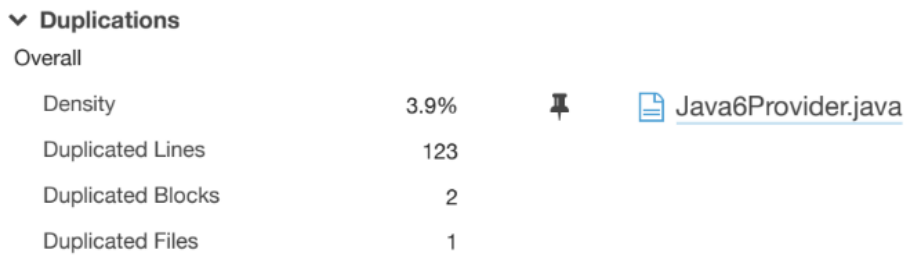
9) Duplication de code

La duplication de code dans un logiciel est connue comme une erreur courante qui affecte la facilité de maintenance des programmes. Une étude sur la maintenabilité d'un logiciel ne serait pas complète si on ne s'attardait pas à la notion de duplication du code qui est présente dans l'application. La duplication de code est une suite d'instructions identiques ou très similaires qu'on retrouve plus d'une fois dans le code source du programme. Ces clones sont souvent créés à la suite d'un copier-coller de la part du programmeur ou lorsque plusieurs programmeurs, sur un même projet, conçoivent chacun de leur côté une même fonctionnalité. La duplication de code ne s'applique pas seulement à des bouts de code exactement identiques. Deux passages similaires de code qui répondent au même besoin et offrent la même fonctionnalité peuvent être considérés dupliqués et devraient être factorisés. Pour éviter la duplication de code, le programmeur doit premièrement factoriser son code lors de sa réalisation et ensuite, c'est le réusinage du code qui permettra de diminuer la duplication.

La duplication de code entraîne des problèmes lourds de maintenance qui sont proportionnels avec la quantité de code qui peut être qualifié de dupliqué. Une modification de maintenance devra, la plupart du temps, être faite à chaque endroit où le code a été dupliqué. Souvent le programmeur n'a pas conscience que le code qu'il modifie est en fait dupliqué à plusieurs endroits dans le code source. La modification ne se retrouvera pas à chaque endroit et ceci peut causer des bugs très difficiles à identifier. Ceci rend les futures évolutions du programme plus risquées. Voici quelques points intéressants tirés de wikipédia(1):

- si le code copié avait une erreur de programmation, alors tout le code collé dispose de la même erreur de programmation. Corriger l'erreur nécessite de modifier tous les endroits où le code est dupliqué
- si le code dupliqué doit évoluer, il faut alors modifier tous les endroits où le code est dupliqué pour y parvenir
- un code dupliqué peut masquer des différences minimes, mais essentielles, qui existent avec une autre portion de code similaire

Puisque les problèmes de maintenance sont directement reliés à la quantité de code dupliqué, la métrique % de code dupliqué dans un logiciel est un bon indicateur de la maintenabilité d'un logiciel. Dans ce travail, nous avons utilisé les résultats sur la duplication calculé avec l'outil Sonarqube. Puisque le code généré par SabbleCC contient une grande quantité de code dupliqué et que nous n'avons pas vraiment de contrôle sur celui-ci, nous avons omis ces fichiers pour l'analyse. Voici les résultats fournis pas Sonarqube:



Les résultats sont assez satisfaisants avec 3.9% du code qui est considéré dupliqué. De plus, le code cloné est un bloc unique de 123 lignes et le problème se retrouve dans un seul fichier nommée Java6provider :

```
public static void provideDAOGetMethod(final FileWriterWrapper writer,
                                        final Entity entity)
                                        throws IOException {


    writer.write("    public " + entity.getJavaName() + " get"
                 + entity.getJavaName() + "(final Integer id) "
                 + "throws DAOException {\n");
    writer.write("        try {\n");
    writer.write("            PreparedStatement query = "
                 + "connection.prepareStatement(\"select * from "
                 + entity.getSqlName() + " where "
                 + entity.getSqlName() + "_ID=?\");\n");
    writer.write("            query.setInt(1, id.intValue());\n\n");
    writer.write("            ResultSet result = query.executeQuery();\n");
    writer.write("            result.beforeFirst();\n");

    writer.write("            if (result.next()) {\n");

        String variableName = entity.getJavaName().toLowerCase();
    writer.write("                " + entity.getJavaName()
                 + " " + variableName + " = new " + entity.getJavaName()
                 + "();\n");
    writer.write("                " + variableName
                 + ".setId(result.getInt(\"" + entity.getSqlName()
                 + "_ID\"));\n\n");
    for (Field field : entity.getFields()) {
        writer.write("                " + variableName + "."
                     + field.getSetterName() + "(result."
                     + mapResultSetGetter(field.getOriginalDataType())
                     + "(\"" + field.getSqlName() + "\"));\n");
    }
    writer.write("\n");
    for (Relation manyToOne : entity.getAllManyToOneRelations()) {
        Entity link = manyToOne.getEntity();
        writer.write("                " + variableName + ".set"
                     + link.getJavaName() + "Id(result.getInt(\""
...
    writer.write("        return null;\n");
    writer.write("    }\n\n");
}
```

```
public static void provideDAOGetAllMethod(final FileWriterWrapper writer,
                                           final Entity entity)
                                           throws IOException {


    writer.write("    public ArrayList<" + entity.getJavaName()
                 + "> getAll" + entity.getJavaName()
                 + "s() throws DAOException {\n");
    writer.write("        try {\n");
    writer.write("            PreparedStatement query = "
                 + "connection.prepareStatement(\"select * from "
                 + entity.getSqlName() + "\");\n\n");
    writer.write("            ResultSet result = query.executeQuery();\n");
    writer.write("            result.beforeFirst();\n");
    writer.write("            ArrayList<" + entity.getJavaName()
                 + "> list = new ArrayList<" + entity.getJavaName()
                 + ">();\n\n");

    writer.write("            while (result.next()) {\n");

    String variableName = entity.getJavaName().toLowerCase();
    writer.write("                " + entity.getJavaName()
                 + " " + variableName + " = new " + entity.getJavaName()
                 + "();\n");
    writer.write("                " + variableName
                 + ".setId(result.getInt(\"" + entity.getSqlName()
                 + "_ID\"));\n\n");
    for (Field field : entity.getFields()) {
        writer.write("                " + variableName + "."
                     + field.getSetterName() + "(result."
                     + mapResultSetGetter(field.getOriginalDataType())
                     + "(\"" + field.getSqlName() + "\"));\n");
    }
    writer.write("\n");
    for (Relation manyToOne : entity.getAllManyToOneRelations()) {
        Entity link = manyToOne.getEntity();
        writer.write("                " + variableName + ".set"
                     + link.getJavaName() + "Id(result.getInt(\""
...
    writer.write("        return null;\n");
    writer.write("    }\n\n");
}
```

Après étude rapide du code, on voit que le cas concerne une duplication de code de type copier-coller dans deux fonctions connexes *provideDAOGetMethod()* et *provideDAOGetAllMethod()*. Notre recommandation pour la duplication du code est de factoriser ce bout de code pour créer une fonction unique qui sera appelée par les deux fonctions. Les environnements de développement modernes proposent des fonctionnalités rapides qui permettent de refactoriser en créant une nouvelle fonction automatiquement. Dans Eclipse, on sélectionne le code puis *Refactorisation* > *Créer nouvelle méthode* et on remplace le code dans les deux sections par un appel à la nouvelle fonction. Dès lors, on élimine la duplication de code et on réduit la longueur du fichier de 60 lignes. S'il y a maintenance dans cette partie, il n'y a plus de risque d'oublier de faire les modifications dans le code dupliqué.

La duplication du code ne devrait pas concerner que les lignes de code littéralement identiques (les types copier-coller) mais devrait inclure les duplications de fonctionnalité et de comportement sous toutes les formes. Les logiciel qui calculent la duplication de code comme celui utilisé ici se concentrent sur les bouts de code facilement identifiables car identiques. Pour bien cerner la duplication au niveau des fonctionnalités et des comportements, le code source doit être analysé et réusiné par un programmeur pour diminuer les duplications dans les comportements et les fonctionnalités.

IV) Recommandations visant à améliorer la maintenabilité

Voici une liste récapitulative des recommandations que nous faisons après analyse des problèmes de maintenabilité du code:

- De manière générale, il peut être utile de paramétrer et utiliser un logiciel d'analyse statique comme Sonarqube pour repérer le gros des problèmes facilement. Ce genre d'outil permet au développeur après le paramétrage de ses propres critères (nombre de LOC acceptables, …) d'avoir un accès direct aux problèmes trouvés par le logiciel ainsi que sa gravité.
- Quelques modifications seraient envisageables pour améliorer l'uniformité du code
    - Remplacer les tabulations par des espaces
    - Eviter le commentaire javadoc paraphrase
    - Éventuellement limiter la longueur des lignes de code
- La testabilité est plus problématique, et la liste des recommandations en rapport à sa qualité et quantité est la suivante :
    - Bien marquer la distinction entre test unitaire et test fonctionnel pour que le développeur se repère plus rapidement
    - Augmenter le nombre de tests non-unitaires (intégration, validation, …)
    - Améliorer l'uniformité et la couverture des tests unitaires
    - Il pourrait être pertinent de reprendre une erreur faite au début de la conception en intégrant plus les tests dans le développement : Pour cela on pourrait penser au développement du code après avoir écrit les tests et également uniformiser le style des tests.
- La documentation n'est pas très riche, le logiciel aurait pu bénéficier d'un texte de présentation au moins des concepts et du fonctionnement. Également certains documents javadocs sont de la paraphrase et donc sont d'une utilité réduite.
- Corriger la méthode très complexe du système
- Refactoriser le bloc de duplication de code

V) Conclusion

Après cette analyse du système, nous pouvons dire que sur la base des métriques que nous avons choisis, le logiciel nous apparaît dans l'ensemble de qualité acceptable moyennant cependant quelques aménagements préalables.

Le programme Pergen est un logiciel de seulement 1800 lignes avec une complexité que nous jugeons normale, avec un système de test sommaire, une logique raisonnable, un besoin réel de maintenance qu'on peut imaginer faible. Malgré ces constatations, plusieurs éléments peuvent être améliorés ou auraient dû ou pu être être réalisés de meilleure façon dès le début.

Nous avons commencé notre travail en appliquant un outil d'analyse statique au code pour repérer les principaux problèmes du système PerGen et pour nous aider dans le choix des métriques objectives et subjectives.

Pour plus de la moitié des métriques, nous avons décelé des problèmes, parfois peu prononcés (complexité, duplication, …). Nous estimons encore que dans deux de ces domaines lesdits problèmes pouvaient être dérangeants pour un développeur découvrant le code :

- L'uniformité du style: Bien que cela puisse sembler relever du détail, l'utilisation d'espaces ou de tabulation aux mêmes endroits peut causer une vraie perte de temps en fonction de l'IDE utilisée.
- La documentation: À travers une absence quasi-totale d'explications sur le système, et le commentaire parfois inutile, la compréhension du code au sens général est plus difficile pour un programmeur qui n'aurait pas d'expérience avec ce type de logiciel.

De plus un grand domaine a attiré notre attention. C'est la couverture et la qualité des tests. Nous estimons que c'est la partie qui nuit le plus à la maintenabilité de PerGen. Les problèmes décelés dans cette partie sont plus profonds et ne nécessitent pas simplement une retouche du code, puisque leur genèse remonte au début du développement.

Une solution simple, bien que coûteuse à mettre en place pour faire face à la "dette" contractée au niveau de la couverture de test serait d'implémenter un système en "test-driven development". Les failles détaillées dans la partie d'études des tests deviendraient toutes bien moins importantes et impactantes.

Pour résumer, PerGen est un projet de petite taille, dont le code souffre de nombreux défauts mineurs. La couverture de tests inégale sur les différents package de l'applications nous empêche de nous lancer dans un refactoring de grande envergure avec sérénité. Par conséquent, et malgré la portée restreinte du code, il apparaît que PerGen pourrait être difficile à maintenir sur le long terme, sauf à implémenter nos différentes recommendations.

VI) Références

[1] V. Nguyen, S. Deeds-Rubin, T. Tan, B. Boehm. "A SLOC Counting Standard", COCOMO® Forum, 2007.
http://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf
[2] R. C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship ", Prentice Hall PTR Upper Saddle River, NJ, USA, 2008.
[3] What does SVN do better than Git?, Software Engineering,
https://softwareengineering.stackexchange.com/questions/111633/what-does-svn-do-better-than-git
[4] T. McCabe. “A Complexity Measure”, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 1976.
http://www.literateprogramming.com/mccabe.pdf
[5] Arthur Watson, Thomas McCabe. “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric”, NIST Special Publication 500-235, 1996.
http://www.mccabe.com/pdf/mccabe-nist235r.pdf
[6] G. Ann Campbell. “COGNITIVE COMPLEXITY, A new way of measuring understandability”, 2016.
https://www.sonarsource.com/docs/CognitiveComplexity.pdf
[7] Code conventions for the Java Programming Language : Naming Conventions.
https://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html
[8] Duplication de code
https://fr.wikipedia.org/wiki/Duplication_de_code