Université du Québec
École de technologie supérieure

RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS MGL 804 RÉALISATION ET MAINTENANCE DE LOGICIELS

# ANALYSE DE LA MAINTENABILITÉ D'UN LOGICIEL À L'AIDE D'UN LOGICIEL D'ÉVALUATION DE LA QUALITÉ

JEAN-FRANÇOIS MARCOUX
MARJ29018206

DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

**Professeur superviseur**

**ALAIN APRIL**

MONTRÉAL, 16 AVRIL 2012
HIVER 2012

**TABLE DES MATIÈRES**

Page

## LISTE DES TABLEAUX

Page

## LISTE DES FIGURES

Page

# INTRODUCTION

Travaillant au sein d'une petite entreprise, dans un rôle plutôt technique, je désirais trouver un projet de session dont nous pourrions en tirer des bénéfices. Le choix de projet s'est donc arrêté sur une analyse de la maintenabilité d'un logiciel à l'aide d'un logiciel d'évaluation de la qualité. Ce rapport est constitué de trois grandes parties.

Dans la première partie du rapport, nous ferons un survol global du système Odotrack, le produit que l'entreprise commercialise depuis janvier 2010. Comme le système Odotrack est composé de plusieurs applications utilisant diverses technologies, un choix a dû être fait sur l'application qui allait être analysée. Le choix s'est donc arrêté sur une application client-serveur, développée en Java.

Comme deuxième partie, on présentera le logiciel d'évaluation de la qualité du code. Il s'agit de Sonar, un logiciel *open-source* très complet. Il se définit comme un logiciel d'analyse couvrant les 7 axes de la qualité du code. Ces axes vous seront alors présentés. Ensuite, on couvrira les divers principaux outils et graphiques d'analyse des résultats offerts par Sonar. Comme dernier point de cette section, on y trouvera une présentation du plug-in SQALE. SQALE étant un modèle d'évaluation du code source d'un logiciel, implémentant la norme ISO 9126.

Dans la dernière partie du rapport, les résultats de l'analyse avec Sonar vous seront présentés. Grâce aux outils de gestion de code source, nous présenterons l'évolution de la qualité du code source du logiciel. Les principales violations et erreurs de conception de suivront. À partir des résultats obtenus et suites aux constats de l'évaluation, nous terminerons par une évaluation globale de l'application et quelques recommandations pour améliorer sa qualité et maintenabilité.

CHAPITRE 1

# PRÉSENTATION DU LOGICIEL ANALYSÉ

## 1.1 Présentation de la compagnie

Le logiciel analysé provient du produit Odotrack, de la compagnie Logiciel Fiscal VL. Logiciel Fiscal VL a été fondé en 2009 et emploie une dizaine d'employés. Une équipe de vente d'environ 20 représentants se joint à l'équipe interne.

Le département informatique, dont je fais partie, est composé de 5 employés à temps plein. La totalité de la conception et la maintenance est effectuée par l'équipe de Logiciel Fiscal. À l'occasion, nous faisons appels à divers sous-traitants pour nous aider dans le développement.

## 1.2 Qu'est-ce qu'Odotrack?

Odotrack se veut un progiciel Web permettant la tenue d'un registre kilométrique électronique. Un appareil électronique, conçu en Chine spécialement pour répondre aux besoins du produit sert à noter l'ensemble des déplacements. Comme composantes, il possède un GPS, une carte SIM, un écran et de divers boutons. Ces boutons permettent à l'utilisateur d'indiquer au système s'il effectue un déplacement pour fins affaires ou personnelles.

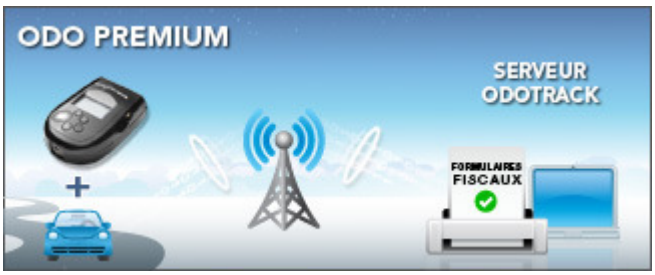


**Figure 1.1 :** Fonctionnement d'Odotrack par réseau cellulaire

L'appareil, branché dans la prise allume-cigarette du véhicule, prend donc en note les déplacements sous forme de coordonnées latitude-longitude. Ces coordonnées sont ensuite envoyées par réseau cellulaire vers les systèmes Odotrack. Les systèmes font alors une conversion des coordonnées en déplacements. Ces déplacements sont ensuite disponibles en ligne et composeront le registre kilométrique de l'utilisateur.

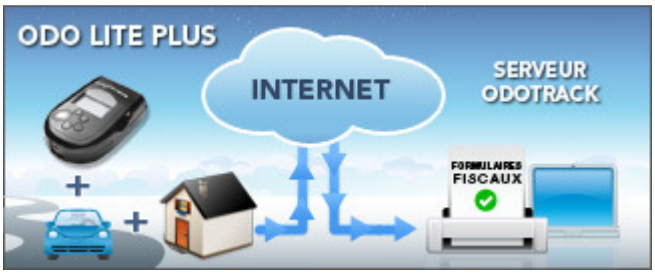


**Figure 1.2 :** Fonctionnement d'Odotrack sans réseau cellulaire

Le produit a aussi été conçu pour les personnes se déplaçant principalement dans les zones sans réseau cellulaire. Dans cette situation, la carte SIM n'est pas activée. Les données doivent donc être envoyées par Internet. Lorsque l'utilisateur désire envoyer ses déplacements vers les serveurs, il doit brancher son appareil dans son ordinateur et un petit logiciel permet l'envoi des données vers les serveurs Odotrack.

## 1.3 Composantes du système

Le système Odotrack est constitué principalement de cinq composantes :

1. Site Web Odotrack, développé en PHP
   Fait la promotion du produit, permet l'achat en ligne, possède une partie administrative permettant la gestion des comptes clients.
2. Application Odotrack, développée en Java et Flex
   Réservée aux clients. Elle permet de consulter les déplacements, dépenses, rapports fiscaux, gestion des comptes de flotte. Dans l'équipe de travail de la compagnie, on

ne lui a pas vraiment donné de nom. Elle est surnommée Flex. Dans le cadre de ce rapport, nous le nommerons plutôt Application Odotrack.

3. Server Logger, développée en Java
   Application recevant les coordonnées envoyées par les appareils. Algorithme complexe permettant de générer les voyages à partir des coordonnées reçues.
4. API Restfull, développée en Java
   API permettant la réception de coordonnées qui sont par la suite traitées par le Server Logger. Offre divers rapports.
5. Application OdoSync, développée en Java
   Application de bureau permettant aux clients d'envoyer les coordonnées dans l'appareil par Internet. Ces points sont envoyés vers l'API Restfull.

Dans le cadre du projet de session, l'analyse de la qualité du logiciel sera effectuée sur le logiciel client-serveur Application Odotrack. Le *back-end* de l'application est fait en Java et le *front-end* en Adobe Flex. Le logiciel d'analyse de la qualité du code utilisé offrait la possibilité d'analyser autant le code en Java qu'en Flex mais celui en Java s'avérait beaucoup plus riche en informations et aussi plus stable. Pour ces deux principales raisons, on a donc choisi d'analyser le back-end de l'Application Odotrack.

## 1.4 Architecture

Le début du développement de l'Application Odotrack remonte à juin 2009 et une première version fût disponible en janvier 2010. Il s'agit d'une application client-serveur traditionnelle.

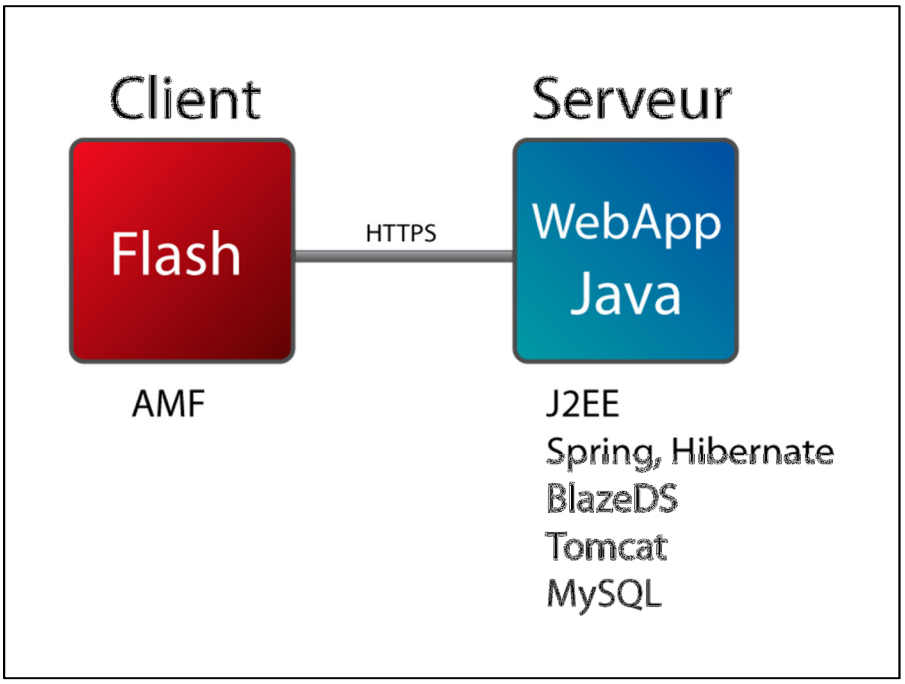


**Figure 1.3 :** Architecture haut niveau

Le client est une Rich Internet Application (RIA) développée à l'aide d'Adobe Flex.
Ses principales fonctionnalités sont :

1. Consultation des rapports
2. Gestion des déplacements
3. Gestion des dépenses
4. Gestion des favoris
5. Gestion des comptes, groupes, véhicules et appareils
6. Traçage temps réel des véhicules
7. Consultation et gestion de la feuille de temps

L'application client fût développée à l'aide de la version 3.2 de d'Adobe Flex. Une nouvelle version utilisant le nouveau *framework* 4.5 est présentement en cours de développement. De plus, de la recherche s'effectue aussi pour qu'une version HTML5 soit disponible à partir de 2014.

Quant au serveur, il s'agit une application Web Java dans une architecture orientée service (SOA). Elle utilise les *frameworks* les plus utilisés de l'industrie : Spring, Hibernate.

La communication entre le client et le serveur, sécurisée par, se fait par Action Message Format (AMF). Le *framework* utilisé pour assurer cette communication est BlazeDS.

L'application est hébergée sur un serveur Tomcat. Quant à la base de données, on utilise MySql.

### 1.4.1 Architecture SOA

Dans le but d'alléger le texte, nous nommerons l'Application Web Java « *back-end* ». Le *back-end* est bâti sous le principe d'une architecture SOA.

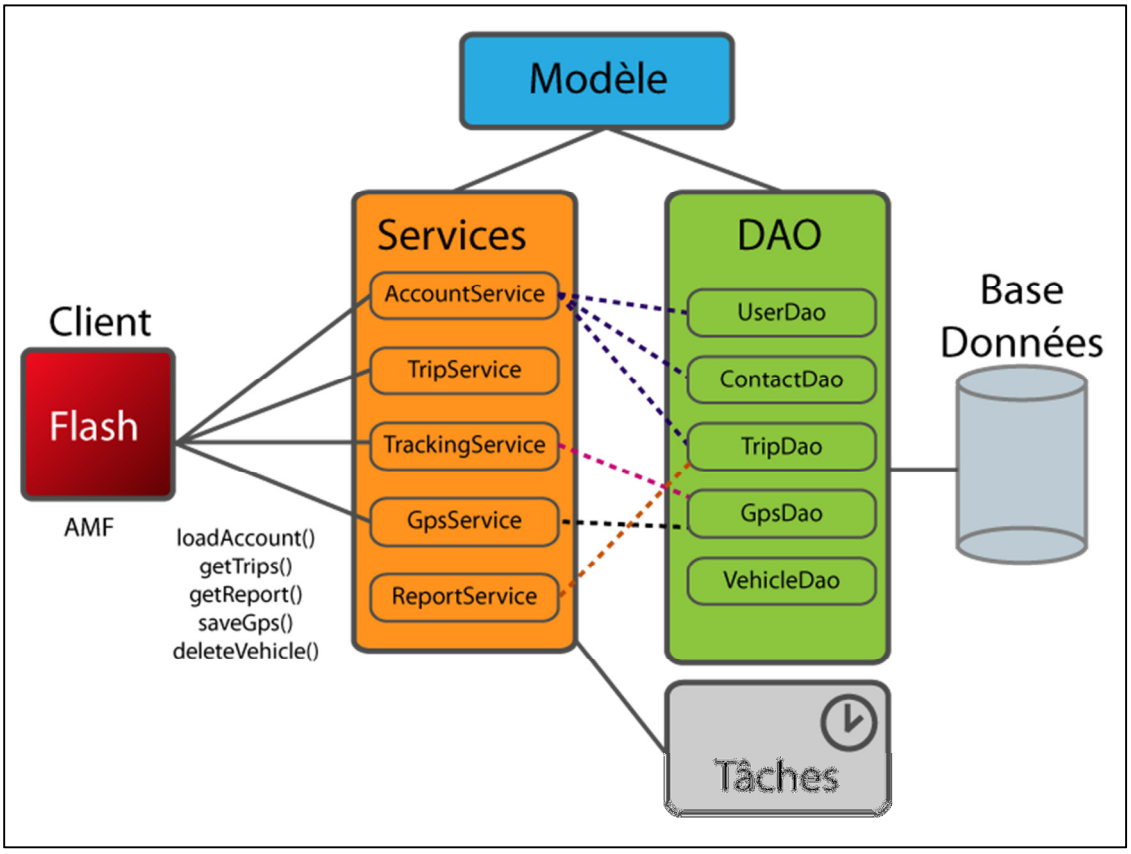


**Figure 1.4 :** Architecture détaillée

**Couche Services**

L'application offre un ensemble de services destinés à la partie client, dans ce cas-ci, l'application client bâtie en Adobe Flex. Ces services sont présentement accédés via AMF mais pourraient aisément être accessibles par exemple par SOAP ou REST. Les services sont découpés par section de l'application. Le tableau suivant liste les sections de l'application et les services utilisés par celles-ci.

| Section | Service |
|---|---|
| Compte | PersistenceService |
| Véhicules | PersistenceService |
| Appareils | PersistenceService |
| Voyages | TripService |
| Dépenses | ExpenseService |
| Rapports | ReportService |
| Odotrace (tracking temps réel) | TrackingService |
| Favoris | FavoriteService |
| Feuille de temps | TimesheetService |
| Rapport | ReportService |
| *Aucune* | JobSchedulerService |

**Tableau 1.1 :** Services de l'Application Odotrack

Dans le tableau ci-dessus, on remarque qu'un service est généralement dédié à une section de l'application à l'exception du `PersistenceService` qui est utilisé pour la gestion des comptes, véhicules et des appareils. Ce service a donc de grandes responsabilités. Quant au `JobSchedulerService`, il n'est utilisé par aucune section du client.

**Couche DAO**

La couche DAO se situe sous la couche Services. Elle pourrait aussi se nommer la couche Hibernate. Cette couche est le pont entre l'application et la base de données. Elle offre les opérations CRUD (Create, Read, Update, Delete). Pour chaque table dans la base de données, il y a un DAO représenté par une classe Java.

**Modèle**

Les objets du modèle sont les objets traités par les opérations CRUD. Ils sont utilisés dans toutes les couches de l'application. Ces mêmes objets sont aussi envoyés côté client. Comme le client est en Flash, une sérialisation AMF est effectuée par BlazeDS. Majoritairement, une classe du modèle représente une table de la base de données.

**Tâches**

Les tâches sont des blocs de code servant à remplir des besoins précis selon un évènement déclencheur ou à une fréquence déterminée. Ces tâches ont pour mandat de peupler des données manquantes relatives aux voyages, générer des rapports et envoyer des courriels à la clientèle.

### 1.4.2 Relation entre les diverses couches

Le tableau ci-dessous exprime les diverses relations entre les couches du *back-end*.

| Couche | Relations et dépendances |
|---|---|
| **Services** | • Utilise un ou plusieurs DAO.<br>• Théoriquement, il ne devrait pas avoir de couplage entre deux services.<br>  ○ Cette règle est appliquée à une exception près où PersistenceService utilise TripService. |
| **DAO** | • Utilisé par divers Services et Tâches<br>• Aucun couplage entre les DAO |
| **Modèle** | • Utilisé à toutes les couches<br>• Sérialisé entre le serveur et le client |
| **Tâches** | • Fait appel uniquement aux Services à quelques exceptions près où l'on utilise les DAO pour obtenir ou écrire des données. |

**Tableau 1.2 :** Couches de l'Application Odotrack

CHAPITRE 2

# PRÉSENTATION DU LOGICIEL D'ANALYSE

## 2.1 Présentation de Sonar

Divers logiciels auraient pu servir à l'analyse de l'application mais le choix s'est arrêté sur Sonar, un logiciel libre permettant de mesurer la qualité du code source sur les projets de développement java. [1]

Sonar se définit comme un logiciel couvrant les 7 axes suivants :

1. Duplication du code
2. Documentation
3. Règles de programmation
4. Détection bugs potentiels
5. Couverture du code par des tests unitaires
6. Répartition de la complexité
7. Analyse du design et architecture

Certains de ces axes sont identiques à ceux explorés dans les laboratoires réalisés dans le cadre du cours. C'est notamment le cas de la recherche de duplication de code (CPD), la qualité du code (PMD) et les règles de programmation (Checkstyle).

Sonar ne réinvente rien. Il utilise plutôt ces outils et en joint d'autres pour offrir une gamme complète de rapports.

Le tableau ci-dessous indique quels sont les outils utilisés pour chacun des axes couverts.

| Axe | Outil |
|---|---|
| Duplication code | CPD-PMD |

| | |
|---|---|
| Documentation | Moteur interne de Sonar |
| Règles de programmation | Checkstyle, PMD |
| Détection bugs potentiels | FindBugs |
| Couverture du code par des tests unitaires | Cobertura, JUnit, Surefire |
| Répartition de la complexité | Moteur interne de Sonar |
| Analyse du design et architecture | Moteur interne de Sonar. Même concept que Lattix |

**Tableau 2.1 :** Outils utilisés pour chacun des axes de Sonar

Son installation est très simple. L'application provient avec un serveur local et une base de données locale. Il peut facilement être déployé sur les divers systèmes d'exploitation tels que Windows, Linux, Solaris, MaxOs.

Quant à son intrégation, elle peut se faire sous Maven ou Ant. Si le projet est bâti sous Maven, rien de plus simple. Il ne suffit que de lancer le but (*goal*) Sonar via la commande `mvn` pour qu'une analyse du code s'effectue!

Sonar n'est pas uniquement disponible pour Java. La communauté a créé divers extensions afin qu'il puisse analyser la qualité du code sous :

1. C
2. C#
3. Cobol
4. Flex
5. PHP
6. Visual Basic 6

## 2.2 Résultats offerts par Sonar

Avec Sonar, on obtient des dizaines, voire mêmes des centaines de mesures gravitant autour des sept axes définis plus tôt. Ces mesures sont présentées sous diverses formes. Elles sont parfois présentées sous forme de liste mais aussi sous forme graphique. Comme principales mesures nous avons :

1. Le nombre de violations par rapport aux règles de programmation définies.
2. Le nombre de lignes de codes, le nombre de packages, le nombre de classes, le nombre de méthodes, le nombre d'accesseurs.
3. Le pourcentage du code documenté.
4. La complexité des classes, des méthodes.
5. Le niveau de cohésion et couplage par classe.
6. La duplication du code.
7. La couverture du code par les tests unitaires.

### 2.2.1 Points chauds

Une page énumère ce que l'on peut considérer les points chauds. C'est-à-dire les classes les classes les plus déficientes par rapport à une mesure. Ceci permet de facilement cerner les classes nécessitant une amélioration, par exemple, pour la complexité, duplication du code, violations, le pourcentage de ligne non testées et bien plus encore.

### 2.2.1 Regroupement des violations

Les violations de l'application sont séparées en 5 catégories : bloquantes, critiques, majeures, mineures, informations. Dans cette section, pour chacune des catégories, on y affiche le nombre de violations trouvées. À droite, on y affiche le nombre de fois où une règle n'est pas respectée. Plus bas, on voit le nombre d'erreurs par package et le nombre d'erreur par classe.

Il est à noter que chacune des règles de validation peut être activée ou désactivée selon les besoins. De plus, la catégorie d'une règle peut être modifiée, exemple, la faire passer de majeure à mineure.

### 2.2.2 Le Radiateur

Divers outils permettent de représenter les résultats sous forme graphique. C'est le cas du Radiateur, un outil graphique permettant de mettre en relation deux mesures. Le graphique se

représente comme suit : un ensemble de rectangles où chaque rectangle a sa propre couleur. Il y a un rectangle par package. La grosseur du rectangle est proportionnelle à l'importance de la Mesure 1 alors que la couleur varie selon la valeur de la Mesure 2.

Plus de 100 indices de grosseur sont proposés alors que plus de 20 indices de couleurs sont disponibles.

L'image ci-dessous représente la grosseur de chaque package en nombre de lignes de codes alors que la couleur représente la conformité des règles de programmation. On déduit donc que les règles de programmation sont très bien respectées – vert représentant valide – auprès des gros packages mais moins bien – rouge représentant invalide – auprès des plus petits packages.

**Figure 2.1 :** Radiateur avec comme première mesure, le nombre de ligne de code et deuxième mesure, la conformité des règles de programmation.

### 2.2.3 Machine à voyager dans le temps

La machine à voyager dans le temps, nommée *Time Machine*, est sans aucun doute la fonctionnalité qui permet le plus d'apprécier Sonar. Pour chaque validation, la version du code et l'heure à laquelle elle fût effectuée sont notées. Pour chaque classe, Sonar est alors en mesure de savoir s'il y a eu amélioration ou dégradation depuis la dernière validation. À cela, Sonar offre des graphiques affichant l'évolution d'une à plusieurs mesures.

Cette fonctionnalité s'avère donc fort utile pour une équipe de développement. Par exemple, lors de tests d'acceptation d'une nouvelle version d'une application, une validation par Sonar peut facilement aider l'équipe à juger si le nouveau produit est de meilleure ou de moins bonne qualité par rapport à la version précédente.

### 2.2.1 Design de l'application

Bien qu'une application ne puisse comporter aucune violation, ceci ne garantit pas qu'elle fût bien conçue. Les outils comme CheckStyle ne font qu'analyser les lignes de codes en appliquant des règles définies. Ils ne portent aucun jugement sur la conception.

Ce qui rend Sonar si complet, c'est qu'il possède de plus un outil permettant d'analyser la conception du code. Son fonctionnement est identique à celui proprosé par Lattix. Le concept est le suivant : sous forme visuelle, représenter les dépendances entre les classes. Pour s'y prendre, on utilise une matrice.

Pour une classe d'un package, cet outil permet, entre autres, de savoir combien de classes de ce package sont dépendantes de celles-ci, de combien de classes du package elle dépend. De plus, il permet aussi de détecter les cycles de dépendances. Ceci est particulièrement intéressant puisqu'il permet facilement d'identifier les classes ou package ayant une

dépendance cyclique, signe d'une mauvaise conception. Il est à noter que l'outil permet autant d'évaluer les dépendances entre les classes d'un package que les dépendances des packages entre eux.

## 2.3 Le modèle de qualité SQALE

Comme on a pu le constater jusqu'à maintenant, les résultats proposés par Sonar sont grandement d'ordre technique. Ils sont facilement compréhensibles par l'équipe de programmeurs mais peuvent l'être beaucoup moins pour les gestionnaires de projet ou toute personne responsable de la prise de décisions.

Une implémentation, du modèle de qualité logicielle SQALE (Software Quality Assessment based on Lifecycle Expectations) basé sur la norme ISO 9126 [2] (Software engineering — Product quality) est disponible pour Sonar. Il est cependant payant, à raison de 4000$ par année.

SQALE [3], tout comme ISO 9126 classifie la qualité logicielle par divers ensembles de caractéristiques. Cependant, les noms varient quelques peu entre les deux :

| Caractéristiques ISO 9126 | Équivalent SQALE |
|---|---|
| **Capacité fonctionnelle** | Sécurité |
| **Fiabilité** | Fiabilité |
| **Facilité d'utilisation** | |
| **Rendement/Efficacité** | Rendement/Efficacité |
| **Maintenabilité** | maintenabilité |
| | Testabilité |
| **Portabilité** | Portabilité |
| | *Changeability* |

**Tableau 2.2 :** Équivalence des caractéristiques de SQALE à celles d'ISO9126

Dans le tableau suivant, on remarque qu'il n'y a pas de mesure pour la facilité d'utilisation. La caractéristique Maintenabilité d'ISO 9126 est découpée en 2 caractéristiques chez SQALE : Maintenabilité et Testabilité. Même chose avec la caractéristique Portabilité, qui chez ISO 9126 est représentée par 2 caractéristiques chez SQALE : Portabilité et *Changeability*.

Le but principal du *plug-in* SQALE est assez simple : connaître la dette technique en jour-homme d'un logiciel. Chaque mesure Sonar est classifiée dans une des sept caractéristiques. Basé sur le nombre de violations et des paramètres définissant le temps moyen requis pour résoudre cette violation, SQALE permet donc de connaître le temps requis qui serait nécessaire à corriger le logiciel.


**Figure 2.2 :** Note SQALE attribuée au logiciel ActiveMQ
Source : http://nemo.sonarsource.org/dashboard/index/78577?did=6

Pour le projet ActiveMQ d'Apache, sa côte SQALE est de B. A étant la meilleure et E la moins bonne. Sa dette technique est évaluée à 1337 jours-hommes de travail.

La *SQALE Pyramid* permet de découper par caractéristique la dette technique de chacune de celles-ci, comme le démontre ci-dessous où la somme de chaque caractéristique donne une somme de 1337 jours.

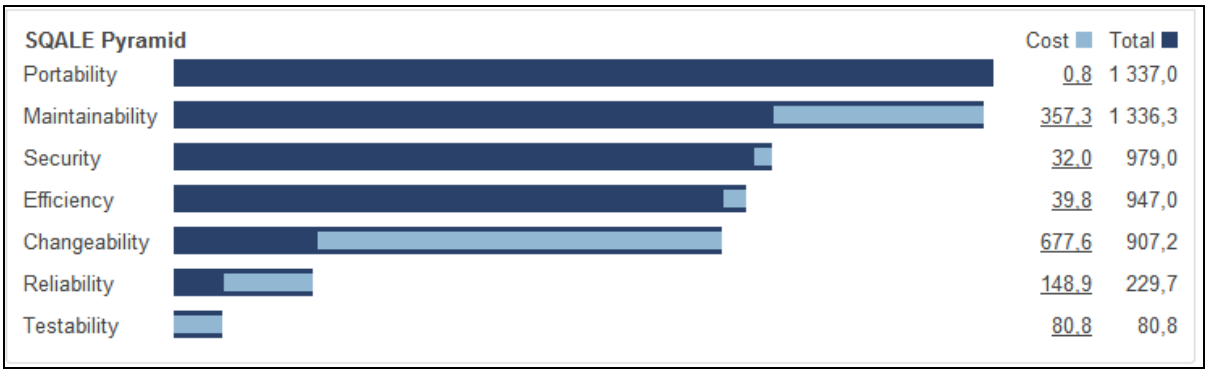


**Figure 2.3 :** Pyramide des caractéristiques SQALE pour l'application ActiveMQ

Source : http://nemo.sonarsource.org/dashboard/index/78577?did=6

Comme chaque mesure est liée à une des sept caractéristiques de SQALE, il est possible de voir la portion occupée d'une mesure par rapport aux autres. Grâce à ces graphiques, ceci peut facilement montrer à quel endroit les efforts devraient être mis afin d'améliorer la qualité du logiciel.

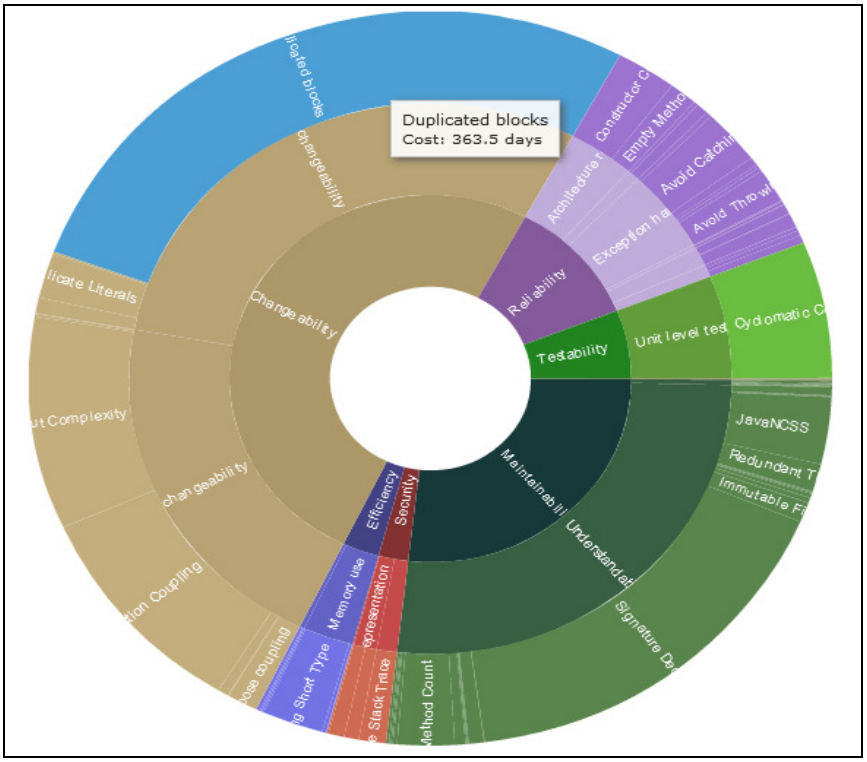


**Figure 2.4 :** Violations regroupées par caractéristique

Source : http://nemo.sonarsource.org/dashboard/index/78577?did=6

Comme le plug-in est payant et que nous ne possédions pas de licence SQALE, nous ne serons donc pas en mesure de démontrer ce qui aurait pu être obtenu avec l'Application Odotrack, mais tenions à informer le lecteur de l'existence de plug-in fort intéressant.

CHAPITRE 3

# ANALYSE DES RÉSULTATS

## 3.1 Premiers constats

Bien qu'il y ait eu plusieurs dizaines de déploiement en production, l'historique de l'application se résume à environ dix versions majeures. Comme la totalité du code source est sous l'outil de contrôle de version SVN, nous sommes revenus dans le temps afin d'avoir une image globale de l'évolution de l'application. Le graphique suivant représente l'évolution des lignes de codes (orange), la complexité (bleu) et la conformité des règles de programmation (vert).

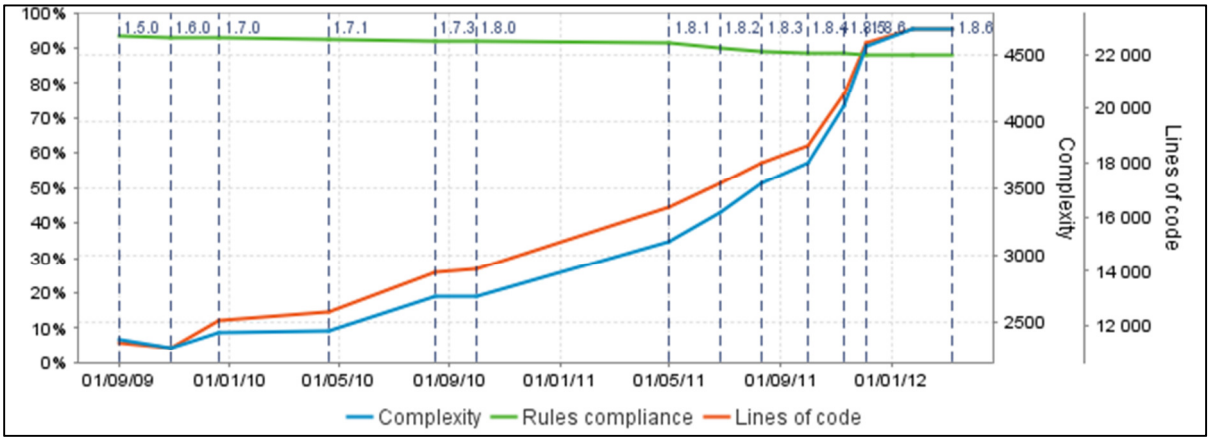


**Figure 3.1 :** Évolution de la complexité (bleu), le nombre de lignes de code (orange) et conformité des règles en 2009 et 2012.

Voici donc les principales constatations.

### 3.1.1 Deux fois plus de lignes de codes

Entre septembre 2009 et janvier 2012, l'application a passé de 12 000 à 23 000 lignes de codes. Elle a donc doublée en grosseur. Ce nombre fait grandement du sens puisqu'une multitude de fonctionnalités furent créées depuis sa première version. Entre autres, on note la

gestion des comptes de flottes de véhicules, le module de favori, de feuille de temps et le traçage temps réel.

En se référant aux autres projets d'Apache, nous réalisations qu'il s'agit d'un projet de petite taille. On peut le comparer à un projet tel que Checkstyle ou Shiro. Le lien suivant vous mènera aux projets Apache, triés par nombre de lignes de code. http://nemo.sonarsource.org/?asc=true&page_id=1&sort=3

### 3.1.2 Complexité sans cesse grandissante

La complexité, elle aussi, augmente sans cesse. La complexité d'une application est en fait la sommation de la complexité de chacune des classes de l'application. En date du 8 mars 2012, l'application comportait 179 classes ayant en moyenne une complexité de 26.2. La complexité de l'application était donc :

```
179 classes * 26.2 complexité par classe = 4689
```

**Figure 3.2 :** Complexité du *back-end* de l'Application Odotrack

Pour fins de comparaisons, Checkstyle possède une complexité de 5189. Plus il y a de classes présentes dans l'application, plus la complexité totale augmente. Pour juger la complexité d'une application il vaut mieux alors évaluer la complexité moyenne par classe. Pour Odotrack, elle est **26.2** par classe alors qu'elle est de **15.9** pour Checkstyle et **13.0** pour Shiro. Pour une application de même grosseur, Odotrack s'avère alors plus complexe que la moyenne.

| Application | Lignes de code | Classes | Complexité / classe |
|---|---|---|---|
| Application Odotrack *back-end* | 22 930 | 179 | 26.2 |
| Apache Shiro | 23 043 | 472 | 13.0 |
| Apache Checkstyle | 23 391 | 326 | 15.9 |
| Struts2 | 83 383 | 1390 | 15.1 |
| ActiveMQ | 170 586 | 2085 | 18.7 |

**Tableau 3.1 :** Comparaison de la complexité de diverses applications d'Apache
Source des données : http://nemo.sonarsource.org

Les services sont la cause principale de ce nombre très élevé. Ceux-ci affichent des nombres astronomiques. Sans surprise, `PersistenceService` affiche la plus grande complexité avec **303** de complexité. Suivi de `PunchManager` à **269**, une classe gérant les feuilles de temps où l'algorithme est assez complexe. Aussi à **269**, il y a il le `TripDao`. Cette classe possède 55 méthodes pour obtenir des déplacements sous de multiples conditions. Ceci est tout de même normal après tout. Il s'agit de la classe allant chercher les informations les plus essentielles de l'application Odotrack.

| Classe | Méthodes | Complexité |
|---|---|---|
| PersistenceService | 72 | 303 |
| PunchManager | 39 | 269 |
| TripDao | 55 | 269 |
| TripService | 37 | 194 |
| FavoriteService | 15 | 165 |

**Tableau 3.2 :** Services avec la plus grande complexité

### 3.1.3 Légère dégradation de la qualité du code

La qualité du code, évaluée par Checktyle, était basée sur les règles nommées « Sonar Way ». Plus de 115 validations ont été effectuées. À titre de comparaison, la validation de Sun est composée de 59 règles. Au 1er septembre 2009, les règles étaient respectées à 93.3%. Le 8 décembre 2012, ce nombre diminuait à 88.0%. Bien que ce ne soit pas parfait, la qualité du code est donc jugée plus qu'acceptable. Nous reviendrons plus tard sur les principales erreurs rencontrées.

## 3.2 Résultats principaux

Cette section présente les principaux résultats en lien avec la matière vue en classe ainsi que divers résultats intéressants nécessitant d'être mentionné. Avant de débuter l'analyse, voir le tableau ci-dessous.

| Auteur | Théorie | Résultat |
|---|---|---|
| Dijkstra (1968) | Éviter les *goto* | 0 *goto*<br>**100.0%** |
| Baker (1972) | Taille idéale bloc de code<br>< 50 lignes | 1714/1739 méthodes<br>**98.6%** |
| McCabe (1976) | Complexicité cyclomatique<br>< 10 | 1695/1739 méthodes<br>**97.5%** |
| | LCOM4 (couplage faible,<br>cohésion forte) | 170/179 classes<br>**95.0%** |
| | Duplication code | **5.2%** |
| | Respect des règles | 1112 violations<br>**88.0%** |
| | API publique documenté | **60.8%** |
| | Portée des tests | **10.1%** |

**Tableau 3.3 :** Résultats principaux obtenus avec Sonar

### 3.2.1 Théorie vue en classe

Les trois premières entrées, où l'on retrouve un auteur, proviennent de la théorie vue en classe. On remarque que ces théories existent depuis très longtemps dans le monde de l'informatique. Il est intéressant de voir que ces notions sont très bien respectées.

Aucun *goto* n'est présent dans l'application, uniquement 25 méthodes sur 1739 ont plus de 50 lignes et 44 méthodes ont une complexité cyclomatique supérieure à 10.

### 3.2.2 Couplage faible, cohésion forte

Une autre notion de la programmation orientée-objet vue dès les premiers cours de génie logiciel est celle du couplage faible et cohésion forte. LCOM4 (Lack Of Cohesion Methods), est une métrique expérimentale développée par Sonar évaluant la cohésion et couplage d'une classe. Une classe avec un couplage faible et cohésion forte se voit attribuer une note de 1. Plus le couplage augmente ou que la cohésion diminue, plus sa note augmente. Pour l'application Odotrack, 9 de 179 classes ont un LCOM4 supérieur à 1. La classe possédant le moins bon résultat a un LCOM4 de 4. Ironiquement, cette classe n'est pas utilisée par l'application. Il s'agit d'une classe ayant été générée par un assistant, dans les tous débuts pour avoir accès à un Web Service. Notons que la moyenne par classe est de 1.1.

### 3.2.3 Duplication du code

Le pourcentage de lignes de code dupliquées est le nombre de lignes de code dupliqué divisé par le nombre de ligne de code dans l'application. Pour l'Application Odotrack, 5.2% du code est jugé comme dupliqué. Après analyse, pratiquement aucun bloc de code dupliqué se veut un gros copier-coller d'un algorithme ou d'une fonction complexe. Les seules méthodes dupliquées s'avèrent des *getters* et *setters* d'objets du modèle possédant des attributs de même nom.

Les principales erreurs que l'on retrouve proviennent souvent des transactions. L'application Odotrack gère elle-même les transactions. On y retrouve souvent des blocs de code semblables à ceci :

```
1834                    session.flush();
1835                    tx.commit();
1836                    session.close();
1837
1838                }
1839                catch(Exception e) {
```

**Figure 3.3 :** Bloc de code dupliqué à plusieurs reprises dans l'Application Odotrack

Même si ce bloc de code se retrouve à plusieurs endroits, une duplication de celui-ci s'avère grandement moins critique qu'un bloc de code renfermant une logique complexe.

Dans le plan des futurs développements du produit, une des demandes est de laisser Spring gérer lui-même les transactions. Ce changement sera évidemment bénéfique en ce qui a trait à la duplication du code puisque le bloc suivant sera retiré à plusieurs endroits dans l'application

```
1834                    session.flush();
1835                    tx.commit();
1836                    session.close();
1838                }
1839                catch(Exception e) {
```

**Figure 3.4 :** Exemple d'une gestion des transactions par Spring

### 3.2.4 Respect des règles

Les règles de programmation sont respectées à 88%. Il y a un total de 1112 violations certaines sont présentes à plus d'une centaine de reprises. Nous nous y attarderons en détail dans la prochaine section.

### 3.2.5 API publique documenté

Comme les commentaires sont essentiels à la compréhension du code et qu'ils facilitent la maintenance d'un logiciel, nous avons décidé de nous y attarder. Environ 61% des méthodes publiques des 179 classes de l'application comportent des commentaires JavaDoc. Ce nombre semble à première vue alertant mais ce n'est pas réellement le cas. En analysant les résultats par classe, on remarque que les classes des Services et DAO sont documentés à plus de 90%. C'est au niveau des classes du Modèle qu'il y a le plus de méthodes publiques non documentées. Ces méthodes sont en fait les *getters* et *setters* de chaque attribut de la classe.

Bien qu'il soit préférable de documenter toutes les méthodes publiques, mêmes les *getters* et *setters*, le résultat s'avère donc moins alarmant qu'il peut le sembler.

### 3.2.6 Porté des tests

Les tests unitaires se font avec JUnit. Ceux-ci couvrent uniquement les classes des Services et DAO. Bien qu'il s'agisse des classes essentielles à tester, celles-ci ne sont pas suffisamment testées. Comme l'indique la portée des tests qui est uniquement 10.1%. En analysant

## 3.3 Principales violations

Comme mentionné auparavant, 1112 violations ont été détectées par Checkstyle avec la validation Sonar Way. Le graphique suivant énumère les principales violations détectées par Sonar.


**Figure 3.5 :** Principales violations de l'Application Odotrack

## 3.4 Violations critiques

Cinq violations jugées critiques ont été détectées. À 4 reprises, on retrouve la violation «Equals Hash Code». Cette erreur a comme explication :

*La définition de la méthode 'equals()' doit toujours être accompagnée de la définition de la méthode 'hashCode()'.*

Pour les classes en erreur, on avait hérité la méthode `equals()` mais avait omit d'hériter `hashCode()`.

La dernière violation critique est «Naming – Suspicious equals method name». Cette erreur a comme explication:

*The method name and parameter number are suspiciously close to equals(Object)*

Le code en erreur est :

```
141 public boolean equals(Object x, Object y) throws HibernateException {
142       Return x == y;
143 }
```

**Figure 3.6 :** Violation de la méthode `equals()`

Dans ce cas-ci, on effectue une surcharge de la méthode `equals()` avec un deuxième argument. Cette pratique n'est pas recommandée puisqu'elle peut porter à confusion avec la méthode `equals()` de la classe `java.lang.Object`.

## 3.5 Violations majeures

Des 1112 violations, 825 sont jugées comme majeures. Puisqu'il y en a quelques dizaines de violations différentes, nous allons analyser les principales.

### 3.5.1 Visibility Modifer

Violée à 181 endroits dans l'application, cette violation est détectée lorsqu'un attribut d'une classe, possédant un *getter* et *setter*, est de type `protected`. Dans cette situation, la

visibilité de l'attribut doit être `private`. Ceci n'impacte aucunement un éventuel héritage de cette classe puisque les attributs, auparavant `protected`, seraient toujours accessibles par les *getters* et *setters*. Bien que cette violation soit critique, elle se corrige très facilement. La correction de ce problème règlerait 16% de toutes les violations.

### 3.5.2 Signature Declare Throw Exception

Selon les spécifications de l'Application Odotrack, chaque méthode publique de la couche Services doit lancer une exception en cas d'erreur. Ceci est essentiel pour que le client (Flash) puisse savoir qu'une exception côté serveur s'est produite. Chaque méthode lance une exception de type `java.lang.Exception`. Or, Sonar Way recommande que les exceptions lancées ne soient pas crues (*raw*). Un exemple de solution serait que les méthodes des services lancent des exceptions de type `com.odotrack.exception.ServiceException`.

### 3.5.3 If statements must use braces

Tout bloc conditionnel (`if`) devrait être entouré d'accolades. Or à 92 endroits dans le code, cette règle n'est pas respectée comme l'exemple ci-dessous de la classe `ReportService.java`.

```
314 if(this.language != null)
142       parameters.put(ReportParameterName.LG.getValue(), this.language);
143
```

**Figure 3.7 :** Violation des déclarations `if`

Il s'agit d'une bonne pratique de mettre des accolades autour de toute condition. Malgré que cela augmente le nombre de lignes de codes, la lecture du code est facilitée. De plus, si les accolades sont présentes dès le début, on n'aura pas à les ajouter dans le cas où une deuxième instruction devait être ajoutée dans la condition.

En plus d'avoir des violations avec les `if`, nous retrouvons 75 violations pour des `else` sans accolade.

### 3.5.4 Autres violations notables

Voici les autres violations majeures méritant d'être mentionnées, nous remarquons que celles-ci ont été soulevées dans le tableau de la section 3.2.

- Duplication de code : 65 violations
- Complexité cyclomatique : 44 violations
- Couplage faible / cohésion forte (LCOM4 > 1) : 37 violations
- Constructeur appelant une méthode parent : 36 violations

## 3.6 Analyse du design

La figure ci-dessous est la matrice représentant les dépendances entre les packages de l'application.

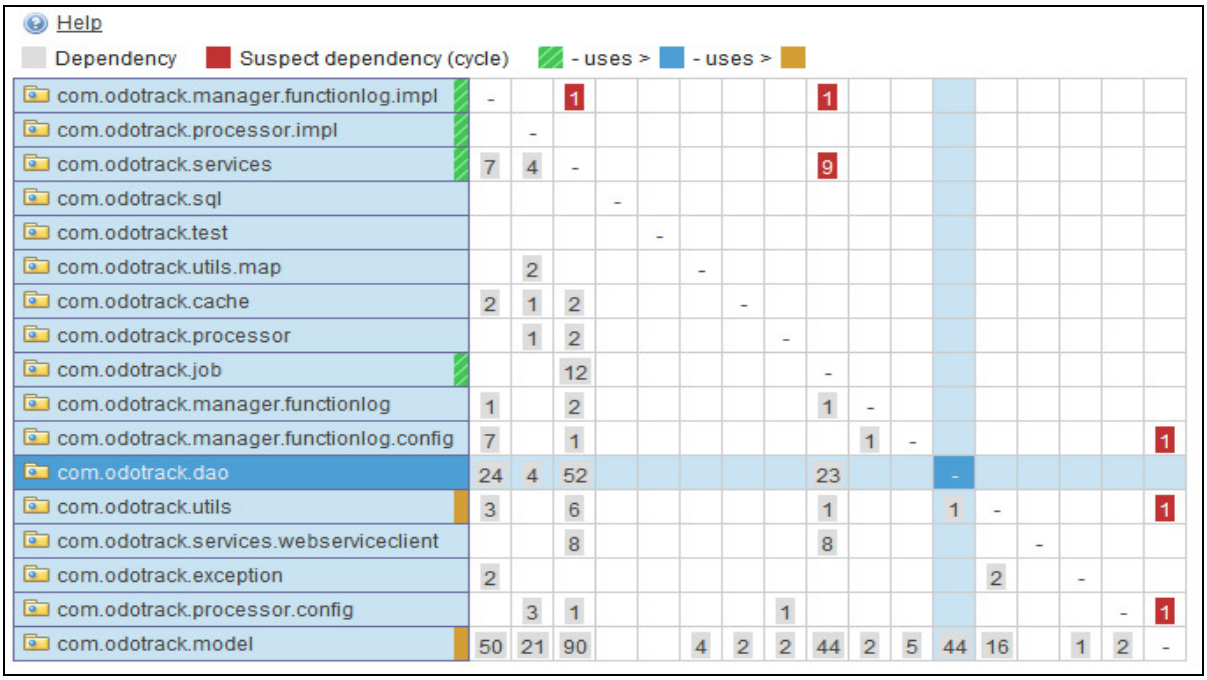


**Figure 3.8 :** Packages dont dépend `dao` (orange) et par qui `dao` est utilisé (vert)

Cette matrice affiche les packages de l'application. La sélection est sur le package `dao`. Les petits rectangles en orange, à la droite des noms de packages, sont les packages dont `dao` dépend et en vert les packages dépendant de `dao`.

Pour ce même package, le chiffre 24 représente de dépendances qu'a le package `manager.function.impl` envers le package `dao`. À droite, le 4 indique que le package `processor.impl` dépend de `dao` à 4 endroits. Plus à droite, le 52 indique que le package `services` dépend de `dao` à 52 endroits et ainsi de suite.

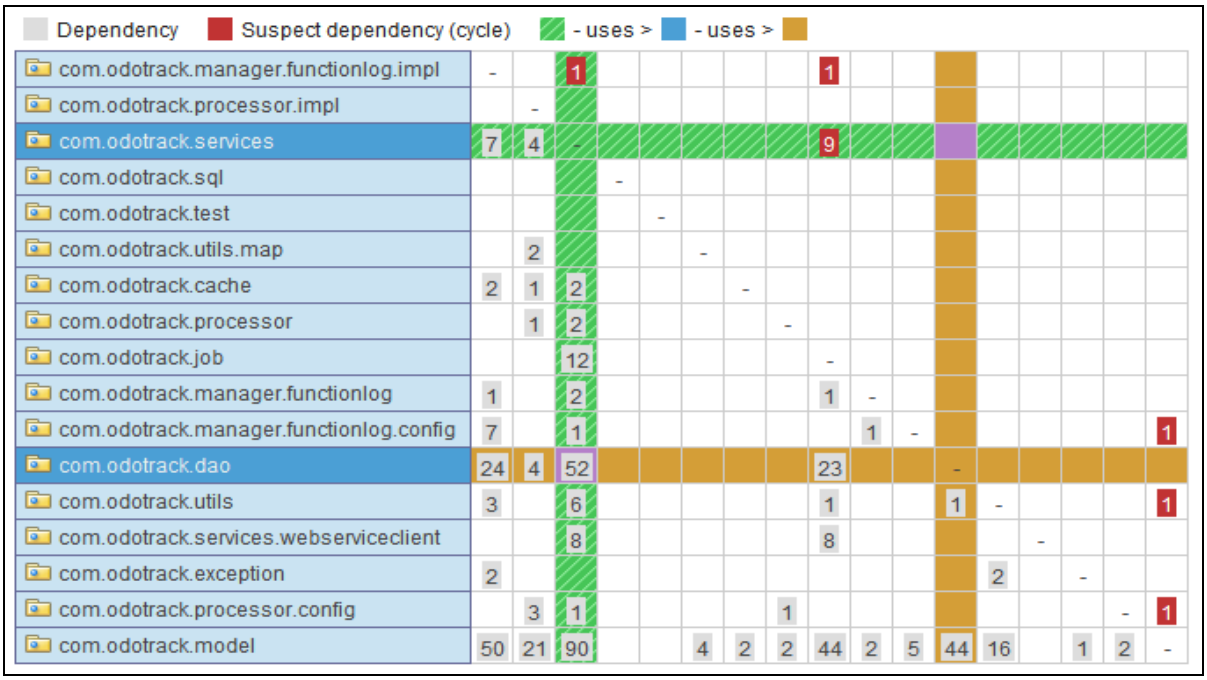


**Figure 3.9 :** Relation des classes du package `dao` avec le package `services`

Pour être en mesure de plus facilement visualiser nous pouvons cliquer dans une case afin de voir par les intersections, les dépendances entre les packages. Dans ce cas-ci, nous avons cliqué sur le 52. Elle représente les dépendances entre le package `dao` et `services`. Dans l'architecture présentée plus tôt dans le rapport, nous avions une couche Service, située au-dessus de la couche Dao. Tel que mentionné, un Service utilise un ou plusieurs Dao. Quant aux Dao, ils ne doivent pas utiliser les Services. Cette matrice permet de confirmer que le concept est bien implémenté. En effet, elle affiche que les Services dépendent à 52 reprise de la couche Dao et que la couche Dao ne dépend aucunement de la couche Services (autre carré vide, de couleur mauve, représentant 0). Une dépendance d'un Dao vers un Service aurait résulté par un carré rouge. Un carré rouge, se traduisant par un cycle de dépendance. Exemple, Service dépendrait de Dao et Dao dépendrait de Service. Or dans un design idéal, il ne devrait pas y avoir de cycle de dépendances. La zone supérieur droite, délimitée par les tirets en diagonale devrait être vide.

Pour l'application, on remarque qu'il y a tout de même quelques cycles de dépendances. À six endroits, on y retrouve un cycle de dépendance dont un endroit en particulier où l'on voit un 9.

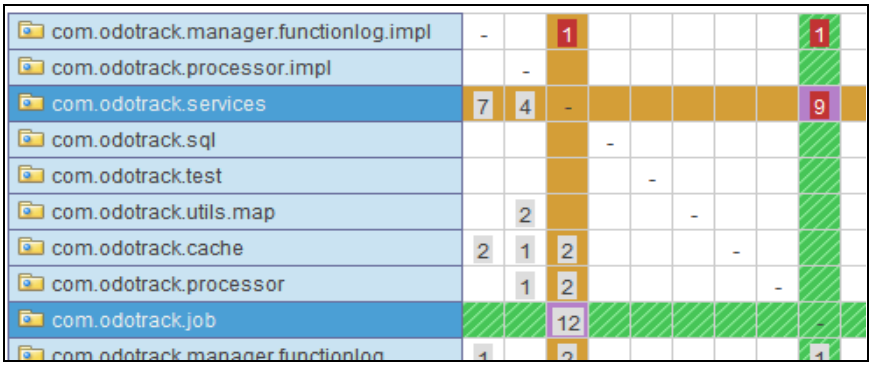


**Figure 3.10 :** Cycle de dépendances entre `job` et `services`

Neuf classes du package `job` utilisent des classes du package `services` :

| com.odotrack.job.* | | com.odotrack.services.* |
|---|---|---|
| FavoriteGeocodingJob | utilise | FavoriteService |
| FunctionLogManagerProcessJob | utilise | FunctionLogService |
| IntegrityValidationJob | utilise | MailingService |
| NmeaDataCleanUpJob | utilise | TrackingService |
| ReportCompilingJob | utilise | ReportService |
| TripETLJob | utilise | BaseService |
| TripETLJob | utilise | FavoriteService |
| TripETLJob | utilise | FunctionLogService |
| TripETLJob | utilise | TripProcessorService |

**Tableau 3.4 :** Services utilisés par les Tâches

Ceci fait grandement du sens. Les *jobs* sont des tâches déclenchées à des moments précis ou à intervalles définis dans le but d'effectuer divers traitements. Pour la plupart, elles utilisent les services, qui eux possèdent la logique d'affaire.

Cependant, si on regarde les dépendances des services envers les *jobs*, nous avons ceci :

| com.odotrack.services.* | | com.odotrack.job.* |
|---|---|---|
| JobSchedulerService | utilise | FavoriteGeocodingJob |
| JobSchedulerService | utilise | FavoriteNotificationJob |
| JobSchedulerService | utilise | FunctionLogManagerProcessJob |
| JobSchedulerService | utilise | IntegrityValidationJob |
| JobSchedulerService | utilise | NmeaDataCleanUpJob |
| JobSchedulerService | utilise | NoLogNotificationJob |
| JobSchedulerService | utilise | ReportCompilingJob |
| JobSchedulerService | utilise | ReportReaperJob |
| JobSchedulerService | utilise | TimesheetExpirationJob |
| JobSchedulerService | utilise | TripETLJob |
| JobSchedulerService | utilise | TripElapsedTimeJob |
| JobSchedulerService | utilise | TripReverseGeocodingJob |

**Tableau 3.5 :** Tâches utilisées par les Services

La classe `JobSchedulerService` utilise 12 classes du package `job`. Or, le mandat `JobSchedulerService` est de planifier les tâches. Exemple, exécuter `FavoriteGeocodingJob` à chaque minute pour importer des favoris en lot, planifier l'exécution de `IntegrityValidationJob` à 6:00 du matin afin de valider l'intégrité des données de la base de données.

Voyant le cycle de dépendance entre `job` et `services`, on se pose alors la question au sujet de `JobSchedulerJob` : rempli-t-elle vraiment la tâche d'un service? Un service avait été défini plus tôt comme une interface avec le client pour que ce dernier puisse interroger le système. Dans ce cas, on conclue rapidement que `JobSchedulerService` ne remplit aucunement cette tâche. Il s'occupe plutôt de planifier à quel moment une tâche doit être exécutée. On peut alors en conclure que `JobSchedulerService` n'est pas réellement un service et qu'il y a alors une faute dans la conception.

Comme solution, nous pourrions la déplacer vers le package vers `job` et la renommer `JobScheduler`. Par le fait même, ceci corrigerait l'erreur de cycle de dépendances.

L'outil d'analyse de design, ne propose pas de solution concrète pour améliorer la conception mais permet tout de même de soulever des interrogations quant aux dépendances entre les packages autant qu'entre les classes d'un même package. L'exemple du `JobSchedulerService` permet d'en témoigner.

# CONCLUSION

Avant ce projet de session, aucune évaluation de la qualité du code n'avait été faite auprès des applications du système Odotrack. Connaissant tout de même les bonnes pratiques du développement logiciel, mais sans pour autant connaître les normes de programmation de l'industrie, nous avons été satisfaits par les résultats obtenus.

En général, les règles de programmation, telles que définies par les 115 validations de Sonar Way, sont très bien respectées. Elles sont valides à près de 90%. Uniquement 5 des violations sont critiques et la grande majorité se corrige très facilement.

Deux des sept axes de la qualité du code sont cependant moins bien respectées. C'est le cas de la documentation du code où uniquement 61% des méthodes publiques sont documentées. Parmi toutes les mesures obtenues, la plus alarmante concerne l'axe de la testabilité. En d'autres termes, la couverture du code par les tests unitaires. À peine 10% du code est couvert par les tests unitaires. Il s'agit d'une lacune extrêmement dangereuse pour la maintenance du logiciel. Toute modification au code existant est à risque puisqu'il y a 90% des chances que cette modification ne soit pas couverte par les tests unitaires.

Suite à ce constant, une décision a été prise avec le gestionnaire de l'équipe pour que chaque ajout ou modification vienne avec par un test unitaire. Grâce à cette nouvelle mesure, le code de l'application couvert par les tests unitaire augmentera.

Du côté de Sonar, ses diverses manières de consulter et présenter les résultats, ses outils graphiques font de lui, un logiciel fort intéressant à utiliser. La machine temporelle permettant de visualiser l'évolution de la qualité du code est merveilleuse. Elle est très utile dans un test d'acceptation d'une nouvelle version d'un logiciel. L'outil d'évaluation du design est aussi superbe. Il fût même utile pour déceler une anomalie au design de l'application Odotrack.

Le modèle de qualité SQALE, basé sur la norme ISO 9126, se veut un plug-in intéressant. L'idée d'avoir une estimation de la dette technique d'un logiciel en jour-homme peut être un très utile pour une entreprise mais le coût du plug-in, environ de 4000$ par année, repoussera assurément bien des gens intéressés à l'acquérir.

Pour terminer, nous recommandons d'utiliser Sonar peu importe votre projet. Étant disponible pour plusieurs langages de programmation, une validation fréquente du code avec Sonar résultera assurément par une meilleure qualité du code.

## BIBLIOGRAPHIE

[1] Sonar (logiciel)
http://fr.wikipedia.org/wiki/Sonar_(Qualit%C3%A9_logicielle)

[2] ISO/IEC 9126
http://en.wikipedia.org/wiki/ISO/IEC_9126

[3] Sonar Source – SQALE
http://www.sonarsource.com/plugins/plugin-sqale/overview/