Académie de Montpellier
Université Montpellier II
Sciences et Techniques du Languedoc

MÉMOIRE DE STAGE DE MASTER M2

effectuée à L'école Polytechnique de Montréal

Spécialité : **Professionnelle et Recherche unifiée en Informatique**

Squaner System QUality AnlyzER

par **HADERER Nicolas**

Date de soutenance : **06/09/10**

Sous la direction de **Y.G. Guéhéneuc, G. Antoniol, M. Huchard, C. Tibermacine**

Résumé

En dépit de nombreux modèles de qualité et d'outils permettant l'évaluation de la qualité telles que PMD, Checkstyle ou FindBugs, très peu d'études ont été réalisées pour voir comment les modèles de qualité sont utilisés par les équipes de développement quotidiennement. Une raison de ce manque d'étude est l'absence d'environnement permettant le suivi de l'évolution de la qualité logiciel. Dans ce document, nous proposons SQUANER (Software QUality ANalyzER), un framework permettant le suivi de l'évolution de la qualité des systèmes orientés objet. SQUANER se connecte directement sur le serveur SVN du système à analyser, analyse le code et procède à l'évaluation de qualité du système à chaque fois qu'une modification est apportée aux codes sources. Après l'évaluation de la qualité, un courriel est envoyé à l'équipe de développement avec les détails sur la qualité du code nouvellement ajouté.

Abstract

Despite the large number of quality models and publicly available quality assessment tools like PMD, Checkstyle, or FindBugs, very few studies have investigated the use of quality models by developers in their daily activities. One reason for this lack of studies is the absence of integrated environments for monitoring the evolution of software quality. We propose SQUANER (Software QUality ANalyzER), a framework for monitoring the evolution of the quality of object-oriented systems. SQUANER connects directly to the SVN of a system, extracts the source code, and perform quality evaluations and faults predictions every time a commit is made by a developer. After quality analysis, a feedback is provided to developers with instructions on how to improve their code.

Table des matières

1 Introduction

Des études ont montré que couramment, plus de la moitié du coût total du développement d'un logiciel est destiné à sa maintenance [23] [38]. En effet, la deuxième lois de Lehman [37] stipule *"Au fur et à mesure qu'un logiciel évolue, sa structure a tendance à se complexifier. Il sera nécessaire de consacrer des ressources supplémentaires si l'on veut inverser sa dégradation"*. Il est donc important de pourvoir contrôler la qualité d'un logiciel au départ de la conception d'un logiciel, mais surtout tout au long de son cycle de vie.

La maintenance et l'évolution des logiciels furent abordées pour la première fois par le Dr. Lehman, en 1969. S'étendant sur une vingtaine d'années, ses recherches aboutirent à la formulation de huit règles de l'évolution d'un logiciel (1997). Elles ont mis en lumière le fait que la maintenance est un processus évolutif et que les logiciels évoluent avec le temps. En évoluant, ils deviennent plus complexes à moins qu'une action spécifique ne soit engagée pour en réduire la complexité.
Les lois de Lehman [37] :

1. (1974) Continuing Change
2. (1974) Increasing Complexity
3. (1974) Self Regulation
4. (1978) Conservation of Organisational Stability (invariant work rate)
5. (1978) Conservation of Familiarity
6. (1991) Continuing Growth
7. (1996) Declining Quality
8. (1996) Feedback System (first stated 1974, formalised as law 1996)

Les lois sur l'évolution des logiciels ont représenté et représentent toujours une théorie très attractive Pour la communauté de l'ingénierie du logiciel, accepter, réfuter, ou bien redéfinir ces lois, a été un très grand challenge pour l'étude empirique sur l'évolution des logiciels. Beaucoup d'études empiriques se sont basées sur certaines de ces lois comme ligne directrice pour comprendre le problème de l'évolution des logiciels et y apporter ainsi des solutions.

Godrey and Tu [26] ont étudié le noyau Linux de 1994 à 1999 et découvrent une croissance linéaire de la taille du projet (plus de 2 millions SLOC). Cette même croissance a été confirmée sur l'éditeur de texte VIM. Ces résultats ne sont pas vraiment surprenant, car au fur et à mesure des versions, de nouvelles fonctionnalités sont rajoutées. Il est donc normal que la taille générale du projet augmente. Mais plus surprenant, une autre étude [51] qui a aussi étudié l'évolution du noyau Linux,

a découvert une croissance exponentielle du couplage entre les différents modules. Cette même étude conclut que si aucun effort n'est fait pour altérer l'explosion du couplage, le système ne pourrait plus être maintenu.

Parnas [48] en 1994 attribue la dégradation du logiciel à un phénomène appelé le chirurgien de l'ignorant *"Ignorant surgery"*. Ce phénomène intervient lorsqu'une succession de développeurs effectuent des modifications sans connaître les choix de conception faits initialement. Ces nouveaux changements parfois peuvent être non conformes par apport aux choix conceptuels, ce qui a pour conséquence de rendre l'ensemble du système beaucoup moins compréhensible. Rendant ainsi la détection d'erreur beaucoup plus difficile et une maintenance beaucoup plus coûteuse. Kemerer [34], attribue le problème de l'évolution du logiciel à un manque de moyen pour contrôler le processus de l'évolution et de la maintenance. Cette même conclusion a été apporté par Mockus, en étudiant le projet Mozilla et Apache [43].

1.1 Problématique du stage

Nous avons vu précédemment qu'il est important de contrôler l'évolution d'un logiciel si on veut éviter sa dégradation. Nous allons nous placer dans le contexte des OSS (Open Source Software) car les équipes développant ces projets sont soumises à plus de difficultés qu'une équipe évoluant dans une entreprise. Ces projets sont donc plus sensibles à la dégradation au cours du temps. Cela pour différents facteurs 1) les équipes de développement sont souvent dans des lieux géographiques différents 2) la rotation des équipes est assez fréquente en fonction des volontaires disponibles 3) manque de documentation explicite 4) manque de planning général du projet 5) manque de moyen formel pour contrôler l'évolution du projet.

Dans ce document, nous allons essayer de trouver un moyen plus formel pour contribuer à l'amélioration du contrôle de la qualité tout au long du cycle de vie d'un système OSS. Pour cela nous avons identifié deux étapes majeures. La première consiste donc à pouvoir mesurer la qualité d'un logiciel (les modèles de qualité). La deuxième sera ensuite de voir dans quelle mesure on va pouvoir analyser et divulguer les informations de qualité aux équipes de développement tout au long de la durée de vie du logiciel. C'est ce qu'on appellera le monitorat de projet. Nous allons dans un premier temps faire un état de l'art concernant ces deux domaines, discuter des différentes solutions disponibles pour enfin conclure sur nos objectifs.

1.2 État de l'art sur la qualité logicielle

La qualité logicielle est un domaine qui a fait l'objet à de nombreux débats. En effet, ce terme généraliste peut avoir plusieurs significations. Pour certains, la qualité logicielle sera la stricte conformité aux cahiers des charges, d'autres seront

plus sensible à la faculté à pouvoir faire évoluer les fonctionnalités du logiciel à moindre coût. Pour évaluer la qualité du logiciel, beaucoup de chercheurs ont essayé de proposer des modèles de qualité et ainsi pouvoir poser une définition. La définition proposée par la norme ISO 9126 [1] est *" Un modèle de qualité est un ensemble d'attributs de qualité liés à un ensemble de métriques. La relation entre les attributs de qualité et métriques précise le processus d'évaluation de qualité."* Nous allons vous présenter les différents modèles proposés qui ont marqué principalement ce domaine, et quels sont les nouveaux concepts émergeant sur lesquels nous pourrons potentiellement nous baser pour la suite de notre travail.

"Le principal but d'un modèle de qualité est de faciliter continuellement l'amélioration d'un système." Boehm 78, Dromey 1996 [13] [21].

Modèle de qualité McCall : Le premier modèle de qualité a été introduit par Jim McCall en 1977 pour US Air Force. Ce modèle tente de concilier l'aspect qualité du logiciel sur le plan utilisateurs est sur les priorités du développeur. Il possède trois perspectives majeures pour définir et identifier la qualité du produit logiciel. La première **produit révision** consiste à évaluer la capacité du logiciel à subir des changements. La deuxième **produit transition** détermine la capacité à s'introduire dans un nouvel environnement et le **produit opération** définissant les caractéristiques des opérations. Ces trois perspectives majeures sont ensuite redéfinies en plusieurs sous-caractéristiques.

- **Produit Révision**
  - *maintenabilité :* l'effort requis pour localiser et fixer une faute dans le programme .
  - *flexibilité :* la facilité à effectuer des changements.
  - *testabilité :* la facilité de tester le programme.
- **Produit transition**
  - *réutilisabilité :* la facilité de réutiliser le logiciel dans un nouveau contexte.
  - *interopérabilité :* l'effort requis pour coupler le logiciel avec un autre logiciel.
- **Produit opération**
  - *fiabilité :*la capacité du logiciel à ne pas avoir d'erreur.
  - *efficacité : au niveau temps processeur, utilisation des ressources ...*
  - *intégrité : protection du programme.*
  - *utilisabilité :* facilité d'utiliser le logiciel.

Le modèle décrit en outre les trois attributs de qualité (les perspectives majeures) dans une hiérarchie de facteurs, critères et indicateurs :

11 facteurs (pour spécifier) : ils décrivent les vues externes du logiciel telles la vue utilisateur.

23 critères de qualité (pour construire) : ils décrivent la vue interne du logiciel

Métriques (pour contrôler) : elles sont définies et utilisées pour mettre au point des méthodes pour mesurer les différents critères de qualité.

L'idée principale de McCall que les 11 facteurs de qualité devaient fournir une vision complète d'un logiciel de qualité. Les différentes métriques étaient calculées par des réponses de type "oui" et "non" et mises ensuite en relation. S'il y avait ensuite autant de réponses positives que négatives aux réponses permettant de mesurer un critère, celui aurait donc une valeur de 50%. Le manque d'objectivité pour mesurer les différents critères de qualité ont fait que le modèle de qualité de McCall a été beaucoup critiqué et peu utilisé, mais il reste néanmoins le précurseur dans un domaine qui va beaucoup évoluer par la suite.

ISO 9126 En 1991, dans une tentative de standardiser l'évaluation d'un système, l'International Organization for Standardization (ISO) propose le standard ISO 9126 qui est divisé en quatre parties (modèle de qualité, métriques internes, métriques externes, caractéristiques à l'utilisation) qui peuvent être utilisées pour spécifier les exigences fonctionnelles et non fonctionnelles des clients et des utilisateurs. Chaque caractéristique est détaillée en sous-caractéristiques, et pour chacune d'elle, la norme propose une série de mesures à mettre en place pour évaluer la conformité du produit développé par rapport aux exigences formulées. Les caractéristiques de haut niveau sont :

- **Capacité fonctionnelle**
  Ensemble d'attributs portant sur l'existence d'un ensemble de fonctions et leurs propriétés données. Les fonctions sont celles qui satisfont aux besoins exprimés ou implicites.
- **Fiabilité**
  Ensemble d'attributs portant sur l'aptitude du logiciel à maintenir son niveau de service dans des conditions précises et pendant une période déterminée.
- **Facilité d'utilisation**
  Ensemble d'attributs portant sur l'effort nécessaire pour l'utilisation et sur l'évaluation individuelle de cette utilisation par un ensemble défini ou implicite d'utilisateurs.
- **Rendement** Ensemble d'attributs portant sur le rapport existant entre le niveau de service d'un logiciel et la quantité de ressources utilisées, dans des conditions déterminées.
- **Maintenabilité**
  Ensemble d'attributs portant sur l'effort nécessaire pour faire des modifications données.
- **Portabilité**
  Ensemble d'attributs portant sur l'aptitude de logiciel à être transféré d'un environnement à l'autre.

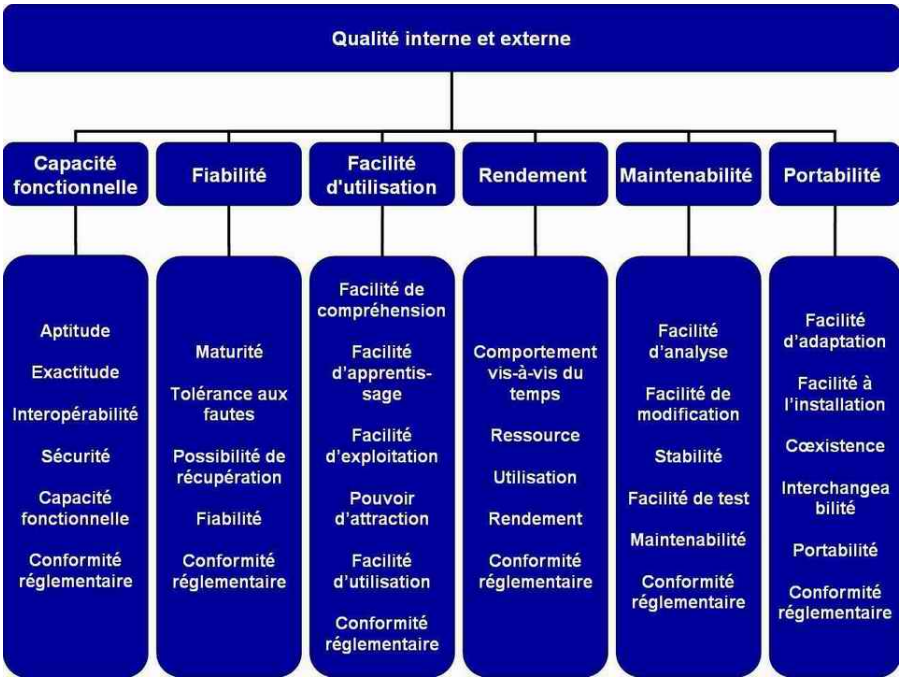


FIGURE 1 – Extrait de la hiérarchie des caractéristiques internes et externes de l'ISO 9126 [47]

Ce modèle indique un lien de causalité entre des caractéristiques internes et des caractéristiques externes d'un produit, mais ne permet toutefois pas de les prédire, ni de connaître bonnes et mauvaises plages de valeurs.

QMOOD Aujourd' hui, QMOOD introduit par Bansiya et Davis [10] est sans conteste le modèle de qualité le plus utilisé et le plus référencé dans les dernières études dans ce domaine. QMOOD ,basé sur les travaux de Dromey's [20] [21], est un modèle hiérarchique, pour l'évaluation des systèmes orientés Objet. Le modèle de qualité consiste en six équations établissant des relations entre six caractéristiques (réutilisabilité, flexibilité, compréhensibilité, fonctionnalité, extensibilité et efficacité) avec onze propriétés structurelles du paradigme orienté objet (encapsulation, couplage, polymorphisme, donnée abstraite et hiérarchie). Un ensemble de métriques orienté objet a été introduit pour mesurer ces propriétés. Ce modèle a été validé par beaucoup de systèmes industriels tels que : Microsoft Foundation classes (5 versions), Borland Object Windows Library (4 versions) et 14 versions de systèmes industrielles de tailles moyennes.

Qualité et patron de conception Avec l'évolution du paradigme Objet, et l'introduction des patrons de conceptions ,devenus très populaire avec l'apparition du livre *"Design Patterns : Elements of Reusable Object-Oriented Software"* [25],certains chercheurs se sont posés la question en quoi les patrons de conception pouvaient influer sur la qualité d'un logiciel. En effet, jusqu'à présent, les différentes caractéristiques étaient calculées selon des métriques uniquement structurelle (nombre de ligne de code, complexité, couplage) et non pas le plan architectural. Bieman en 2002-03 [11] [12] et Vokac [53] en 2004, sont reconnus pour leurs travaux dans ce domaine. Voka par exemple analyse le taux d'erreurs des classes jouant un rôle dans un patron de conception avec les autres classes. Un exemple de ses résultats est que les classes appartenant au patron Fabrique "Factory Method" sont plus compactes, moins couplées et sont sujettes à moins de fautes que les autres. Vokac fournit la première preuve qualitative de la relation entre les fautes des classes et les patrons de conception.

Qualité et Anti-patron de conception En 1998, Brow [16] décrit la notion d'anti-patron qui sont le résultat d'erreur courante dans le génie logiciel comme par exemple le code spaghetti dans lequel il serait impossible de modifier une petite partie du logiciel sans altérer le fonctionnement de tous les autres composants. Ignatios et Du Bois [22] [19] effectuent les premières études permettant de refléter qualitativement l'impact de certains anti-patrons sur la qualité. Pour aller encore plus loin, Wei and Raed en 2007 tentent d'établir une relation entre la probabilité qu'une classe puisse commettre des erreurs et certains anti-patrons.

SQUAD En se basant sur les travaux de Guéhéneuc et Antoniol [29] pour l'identification des patrons de conceptions initialement instauré par Brown [15] en 1996 et de Moha [45] en 2009 pour l'identification des anti-patron, Foutse Kohmh [36] met au point le premier modèle de qualité instaurant l'aspect architectural du code. Le modèle est basé sur la fiabilité, fournissant une approche probabiliste sur les erreurs et les changements des différentes structures du code. Les modèles de prédiction ont suscité beaucoup d'intérêt, notamment le modèle de Zimmermann [54] pour le plus connu. Le principe est de se baser sur l'historique qualitatif du code, pour apprendre par la suite la relation entre les différents facteurs de qualité et les fautes commises. La grande valeur ajoutée de se baser sur l'aspect architectural du code est donc d'apporter une sémantique à la prédiction de faute. C'est- à-dire que la prédiction des erreurs ne sera donc pas porté pour un élément précis du code (une classe) mais sur une combinaison d'éléments. De plus Kohmh montre que les patrons et anti-patrons de conception ont une relation directe avec les fautes. Il obtient qu'en se basant uniquement sur les patrons et anti-patrons pour la prédiction de fautes, la prédiction se trouve similaire voir supérieur qu'en ne se basant que sur les métriques sur lesquelles se basent le modèle Zimmermann. Ceci représente une grande avancée dans le domaine.

Discussion Depuis ces 30 dernières années, beaucoup de modèle ont vu le jour, certains privilégiant un aspect de la qualité plutôt qu'un autre. Nous vous avons défini les modèles qui ont le plus influencé la direction générale des modèles d'aujourd'hui. Il en reste néanmoins que ce domaine est encore très actif dans le milieu de la recherche. Notamment, une des directions suivies par le laboratoire SOCCER à l'école Polytechnique de Montréal est de voir en quelle mesure les identifiants présents dans le code puissent avoir un impact sur la compréhensibilité. Avec l'étude des différents modèles de qualité présentés ci-dessus, nous avons retenu plusieurs types de modèles. Tout d'abord, les modèles se basant sur un aspect structurel (utilisant les métriques) pour effectuer une analyse selon plusieurs facteurs de qualité tels la flexibilité, la maintenance ... Et ensuite les modèles se basant sur l'aspect architectural du logiciel tel que les patrons et les anti-patrons de conception. Les verdicts issus des différents modèles de qualité se décomposent en verdict qualitatif, exprimant une évaluation générale "un score" sur plusieurs facteurs de qualité. Les verdicts nominatifs, exprimant une évaluation "bonne ou mauvaise". Et enfin les verdicts prédictifs se basant sur l'expérience de l'évolution des différents facteurs de qualité pour émettre une probabilité d'une erreur. Nous souhaiterions donc pouvoir inclure l'ensemble de ces modèles dans notre solution. Nous allons voir maintenant les outils de monitorat de projet pouvant nous permettre de distribuer nos modèles de qualité.

1.3 État de l'art sur le monitorat

Les systèmes permettant de réaliser le monitorat de projet distant sont très peu présents dans la littérature. Non pas qu'il n'en existe pas, mais surtout que les solutions proposées sont souvent internes aux entreprises. Donc pas mise à la disposition de la communauté.
Cependant, plusieurs chercheurs se sont portés sur le problème du monitorat distant.

CVSAnalY Tool CVSAnaly Tool [50] est un système se basant sur le répertoire CVS (Concurrent Version System) d'un OSS qui fournit des mesures et des analyses automatiques. L'ensemble des données collectées sur le CVS leurs permet d'analyser l'évolution générale d'un OSS (nombre de fichiers ajoutés, nombre de participants, nombre de jours ...). L'avantage de se baser un répertoire CVS est de pouvoir analyser les ressources de manière non-intrusive, c'est-à-dire d'être complètement indépendant de l'équipe de développement, et du projet. Cependant, ce système n'effectue pas d'analyse au niveau qualité, et ne se base aucunement sur le code du projet pour effectuer les analyses d'évolutions.

PAM Architecture Se basant sur les travaux de Skoll [42] [49] permettant de contrôler les processus d'assurance qualité, PAM architecture [14] propose un système distant permettant une analyse continue du code afin surveiller l'évolution

des systèmes Open-source en se basant le répertoire CVS du système. Le principe de PAM est de récupérer les courriels envoyés par CVS lorsqu'une modification a été apportée sur le code "les commit" de télécharger le code source directement à partir du répertoire et d'effectuer une série d'analyse permettant de contrôler l'évolution de la dégradation et de l'augmentation de la complexité. Nous discuterons plus en détail de cette solution dans la deuxième partie car, c'est dans cette approche que nous allons construire notre solution.

Squale Squale est une plateforme Open-source permettant d'analyser des systèmes multi-langage et de donner une évaluation qualité. Squale offre des modèles de qualité inspirés de la norme ISO 9126 et des approches de McCall traitant à la fois des côtés technique et économique de la qualité. Le projet a été initié conjointement par Air-France et Qualixo. Un des grands points fort de Squale est sa capacité à analyser une large gamme de langage tels que C++,Java,.Net,PHP et Cobol. Il offre ainsi ses propres outils d'analyse sur le plan syntaxique, structurel permettant d'élaborer de développer ses propres modèles de qualité.

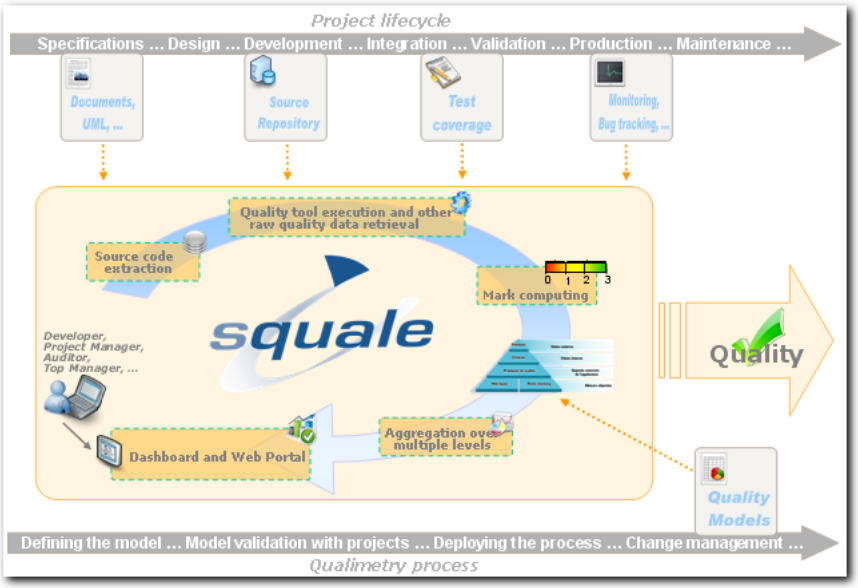


FIGURE 2 – Diagramme de Squale

SONAR Sonar est un outil open-source initialement développé par la société suisse Hortis. Depuis novembre 2008, c'est la société suisse SonarSource qui se charge du développement et du support de Sonar. Le but principal de cet outil est de

fournir de nombreuses statistiques (ou "metrics") sur des projets. Ces données permettent ainsi d'évaluer la qualité du code, et d'en connaître l'évolution au cours du développement. La figure 1.3 exprime les différentes étapes de l'analyse effectuée par sonar.

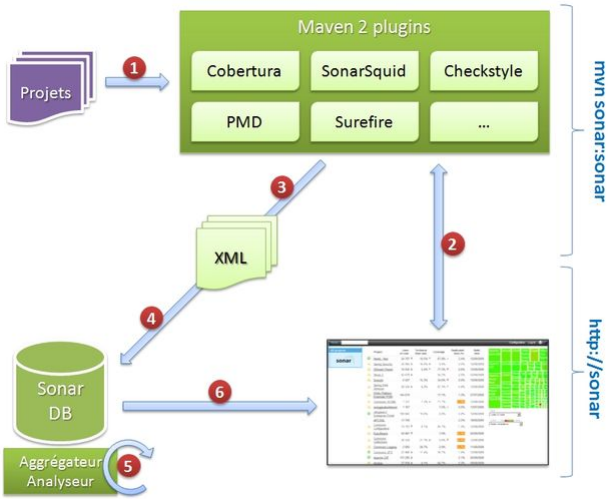


FIGURE 3 – L'architecture de Sonar

1. Le plugin Sonar va lancer les différents plugins Maven 2 (Checkstyle, Cobertura, Surefire, PMD, etc.) nécessaires à l'analyse du projet.
2. Le serveur Sonar est également un mini-repository utilisé par Maven pour récupérer les différents artifacts, ainsi que les configurations Checkstyle, PMD, Cobertura.
3. L'exécution des plugins d'analyse va générer des fichiers XML que le plugin Sonar va regrouper...
4. ... puis envoyer dans la base de données de Sonar.
5. Là, au niveau du serveur Sonar, les données fraîchement glanées vont être agrégées puis analysées.
6. Le tout sera finalement consultable via l'interface web de Sonar.

Sonar est outil très intéressent, offrant beaucoup de fonctionnalités telles que détection de code à risque, support de règle de qualité selon la norme ISO 9126, gestion de profil et un tableau bord complet des différents projets suivis.

Discussion Nous vous avons présenté quelques projets relatifs au travail que nous souhaiterions accomplir. Leurs études nous ont permis d'établir une bonne base de ce que nous voulions. Sonar et Squaner sont les deux projets les plus matures concernant la mesure de la qualité logicielle Open-source. Pour étendre les outils en évaluant d'autres langage que le Java, il faut télécharger de nouveaux modules qui sont malheureusement payants. Les analyses effectuées par ces deux solutions se basent sur du code statique et ne peuvent pas prendre en compte l'aspect conception du logiciel comme les patrons de conception. Or, c'est cet aspect que nous voulons étudier pour voir en quelle mesure la conception du logiciel a un impact sur la qualité. Une autre limitation se trouvant dans SONAR est qu'il ne permet pas de poursuivre un suivi continu de la qualité. Seul l'analyse de la dernière version évaluée est disponible. Squale possède un système d'audit qui peut être activé manuellement ou périodiquement pour lancer l'analyse. Une dernière limitation trouvée sur ces deux projets est la complexité pour l'installation et ils ont besoin d'un certain temps d'entraînement pour pouvoir les utiliser efficacement.

1.4 Objectif du stage

Les objectifs se sont définis tout au long de la période du stage qui était de six mois. Dans un premier temps, le but était de reprendre l'architecture PAM vue dans l'état de l'art, pour y effectuer quelques améliorations. Ce système, initié il y a quelques années en collaboration avec une équipe de développement open-source, n'est pas encore une version stable. Comportant encore beaucoup d'anomalies, la première mission a été d'étudier la solution mise en place, d'y détecter les principaux défauts, puis de décider si nous devions repartir sur une nouvelle solution ou pas. Les laboratoires Ptidej et SOCCER, initiateurs du stage, sont deux laboratoires travaillant en commun sur la qualité logicielle de manière globale. Actuellement, ces laboratoires concentrent leurs efforts sur l'élaboration de nouveaux modèles de qualité en prenant en compte de nouveaux aspects du logiciel autres que les métriques internes comme les lignes de code ou le couplage. Les pistes suivies sont de voir en quelle mesure la conception du logiciel peut avoir un impact sur la qualité du logiciel. Nous entendons par conception du logiciel la manière de coupler les classes entre elles formant ainsi des motifs de conceptions. Les patrons de conception par exemple pour les motifs qui devraient améliorer le code, et les anti-patrons de conception qui devraient au contraire le dégrader. L'équipe Ptidej possédant des modèles de qualité prenant en compte les motifs de conception du logiciel et des algorithmes et des outils pour les identifier nous voulons donc réfléchir à un système pouvant les inclure. Ce nouveau système pourrai donc convenir à deux types d'acteurs. Toujours les équipes de développement, qui pourraient avoir un suivi continu de la qualité de leurs logiciels. Puis les équipes de recherche, les données collectées des différents projets analysés pourraient permettre l'amélioration des outils de détections des

motifs, l'étude de l'impact des motifs sur la qualité du code et l'amélioration des modèles de qualité. Une autre piste suivie par le laboratoire SOCCER est d'étudier l'impact des identifiants des variables dans le code sur la qualité logicielle. Mais les recherches à ce niveau étant encore trop jeunes, nous n'allons pas en tenir compte pour l'élaboration de notre système.

2 Un premier prototype

Une première version de notre solution appelée *"Grass Monitoring Projet"* basé sur l'architecture PAM [14] a été développée dans le cadre d'un cas d'utilisation sur l'assurance qualité en continue d'un projet Open-source Grass [4], qui est un logiciel permettant la gestion et l'analyse de données géospatiales principalement développé en C/C++. Nous allons maintenant analyser cette solution pour y voir les limites et les défauts principaux.

2.1 Analyse

La figure 2.1 montre le principe de fonctionnement général du premier prototype réalisé. Le prototype est un ensemble de composants répartis entre le serveur principal *Soccer* et le serveur *Tomcat* lui-même inclus dans le serveur principal. Le serveur *Soccer* possède donc des composants permettant d'effectuer différentes actions d'analyse et de gestion de base de données. Le serveur *Tomcat* possède un ensemble de services web permettant d'actionner les différents composants du serveur. Ces services web sont donc actionnés par une application extérieure appelé Client Manager qui les actionne selon le processus d'analyse qualité.
La solution se base pour récupérer le code à analyser sur son dépôt SVN, ce qui permet d'être complètement indépendant de son emplacement. Une fois qu'un membre de l'équipe de développement effectue une modification sur le serveur SVN, les serveurs SVN envoient un courriel concernant les différentes modifications à un serveur mail dédié au projet. Le Client Manager actionne donc un premier service permettant de récupérer ce mail et l'analyse pour y récupérer différentes informations "fichier modifié, date, auteur,...".Il actionne ensuite une série de services permettant de récupérer et analyser le code nouvellement ajouté. Une fois toutes les données analysées, le *Client Manager* envoie un mail à l'équipe de développement pour leur donner les différents résultats calculés et leur permettre de voir ainsi la qualité de leurs modifications.

2.2 Les défauts

En analysant la structure déjà mise en place, nous avons constaté plusieurs défauts majeurs rendant l'application difficile à maintenir et surtout à étendre. En effet, le prototype étant développé pour une étude de cas particulier, l'effort premier n'a donc pas été porté sur la réutilisation du logiciel.

Point d'entrée, les courriels Le premier défaut majeur que nous avons observé est le point d'entrée du processus qui est le courriel reçu. Pour connaître les fichiers modifiés par l'équipe de développement, un service web est chargé de récupérer le

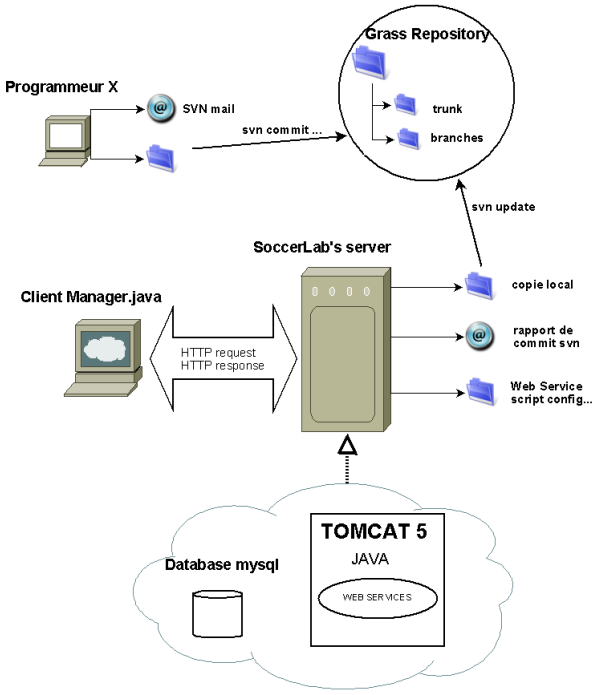


FIGURE 4 – Première architecture Gutsy

courriel sur une messagerie qui lui est dédié. Un autre service dédié permet d'analyser le courriel pour y récupérer différentes informations (nom de l'auteur, fichier modifié, date ...). Ce principe de fonctionnement comporte plusieurs inconvénients. Le premier pose un problème de concurrence. Si plusieurs projets sont analysés en même temps, avec le système mis en place, il est impossible de savoir quel courriel correspond à quel projet. Une re-direction étant faite à partir d'un serveur smtp local, nous ne pouvons pas identifier le destinataire du mail. Ce qui a pour effet de mélanger toutes les données correspondant aux divers projets analysés. Un autre problème lié aux courriels est que chaque type de serveur SVN possède son propre formatage de courriel, un courriel venant d'un serveur SVN X sera donc différent d'un autre courriel venant d'un serveur SVN Y. Ce qui complique grandement l'analyse du mail pour y extraire les différentes informations dont on a besoin pour la suite du processus.

Appel des outils d'analyse Un autre problème que nous avons constaté, est que les différents outils d'analyse ne se situent pas dans le serveur *Tomcat*. En effet, ils se situent dans le serveur principal et sont actionnés par des scripts qui sont eux-même appelés par le serveur *Tomcat*. Ce qui a pour inconvénient de complexifier les appels, donc de complexifier l'application et sa compréhension. Cette manière de procéder

rend la détection très difficile car cela nécessite de passer par des fichiers d'erreurs intermédiaire.

Utilisation des services web Le deuxième défaut que nous avons observé est l'utilisation des services web. Une de leurs caractéristiques est d'être complètement indépendants du langage utilisé et d'offrir les fonctionnalités d'une application sur le monde extérieur utilisant des protocoles légers tels que HTTP. Or nous ne retrouvons aucun avantage d'utiliser les services web dans ce cas précis. Les services web étant destinés à n'être utilisés que par le *Client Manager*, une solution plus simple serait de positionner les Clients Manager en tant que composant à part entière dans le serveur *Tomcat*. De plus, les informations demandées pour leurs utilisations sont trop complexes pour être utilisé par une application tierce.

Général Nous avons aussi repéré d'autres défauts d'autres plus technique comme le couplage fort entre les différents éléments, des classes comportant plus mille lignes, beaucoup de variables statiques utilisées à mauvais escient, manque de documentation. Tous ces défauts font que l'application est d'autant plus difficile à comprendre et à maintenir.

Nous avons donc analysé le premier prototype et avons remarqué quelques défauts d'ordre structurel et d'ordre plus technique notamment dans l'utilisation des services web. Or cette nouvelle technologie devenue très populaire peut vraiment apporter un vrai plus dans le monitorat de projet à distance. Pour notre nouvelle solution, nous allons donc garder le concept de base, certains outils d'analyse que nous ne remettons pas du tout en cause et voir en quelle mesure nous pouvons apporter une amélioration à ce système.

3 Gutsy

Nous allons maintenant étudier la nouvelle solution mise au point durant le stage. Nous y verrons aussi quelles sont les améliorations apportées par rapport au premier prototype. Sur la figure 3, nous pouvons voir l'architecture générale de la nouvelle solution. Nous reviendrons sur cette figure tout au long de cette section. Dans un premier temps, nous allons étudier l'architecture contenue dans le serveur *Tomcat* et les différents composants dont il dispose. Dans un second temps, nous verrons le scénario général afin de voir un cycle complet de notre processus pour l'analyse d'une version d'un logiciel.

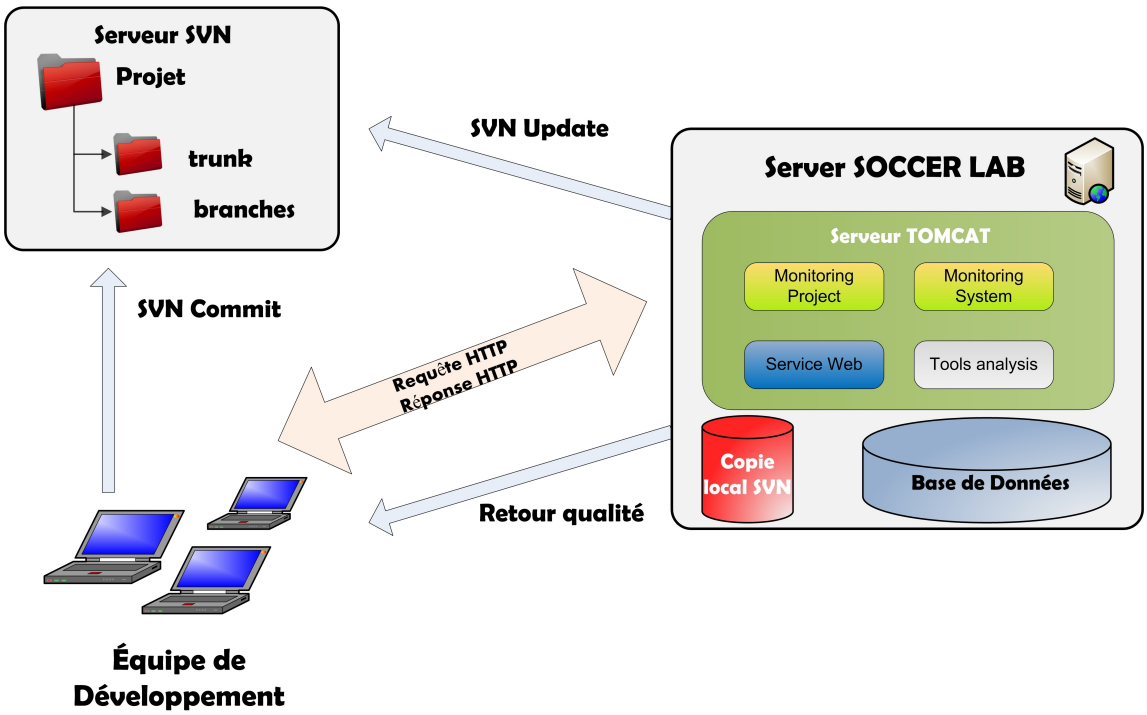


FIGURE 5 – Nouvelle architecture de Gusty

3.1 Architecture Tomcat

Tomcat [8] est un conteneur d'application Java (Servlet) de la fondation apache permettant le déploiement d'applications Java Web. La figure 3.1 montre les différents composants inclus dans le serveur *Tomcat* constituant l'ensemble de notre solution. Notre solution est donc un ensemble d'applications Java que nous appellerons composants, permettant d'effectuer des fonctionnalités spécifiques correspondant à une

ou plusieurs étapes du processus de l'analyse qualité d'un projet. Le composant *MonitoringProject*, anciennement appelé *ClientManager* dans la première version du prototype est devenue un composant à part entière du serveur *Tomcat*. Cette manière de procéder a pour but de simplifier les appels et d'avoir une plus grande maîtrise de l'exécution simplifiant grandement son implémentation. Son rôle est toujours de coordonner les différents composants selon un processus que nous allons déterminer plus tard. Dans notre architecture, nous nous retrouvons avec deux types de services web. Un premier type a accès à la configuration du système, permettant d'avoir un contrôle général sur notre système permettant de le démarrer, de le stopper, de le configurer. Et un autre type de service Web accès utilisateur permettant d'interroger le système sur les différents résultats qualité de son projet ou d'ajouter son propre projet pour qu'il soit analysé par notre système. Nous allons maintenant décrire toutes les étapes et les différents composants constituant le processus de l'analyse d'un projet.

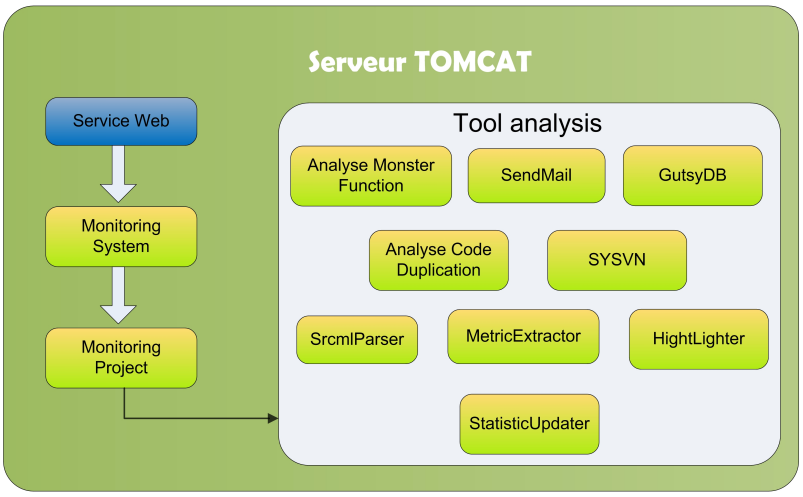


FIGURE 6 – Architecture Tomcat

3.2 Étape 1 : Téléchargement du code source

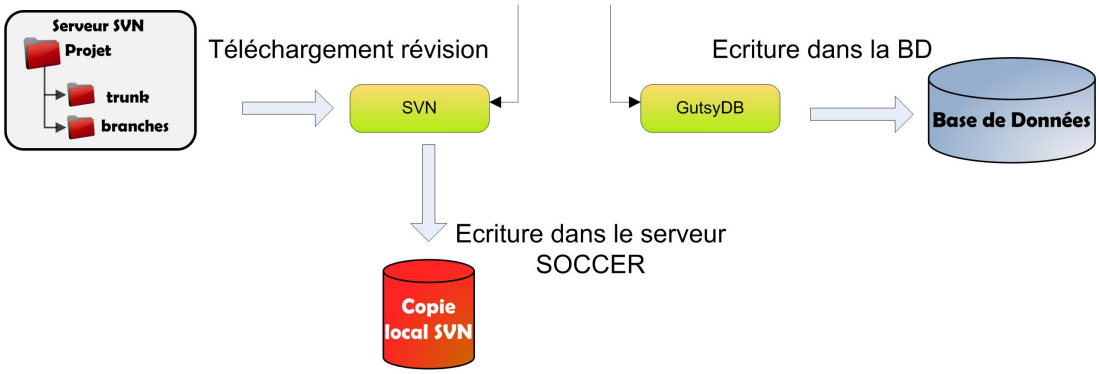


FIGURE 7 – Étape 1 : Téléchargement du code

L'étape 1 3.3 consiste donc à récupérer le code source devant être analysé. Un principal apport de notre version consiste donc à se baser directement sur le serveur svn pour savoir lorsqu'une modification sur le code source a été apporté. Dans un premier temps, nous vérifions si une nouvelle version est disponible. Si tel est le cas, nous récupérons les différentes informations de la révision à télécharger tels que :

- Nom de l'auteur
- Numéro de la révision et date
- Fichier ajouté, modifié ou supprimé
- Message concernant la révision (Commit message)

Grâce à ces informations, nous pourrons par la suite établir une relation entre les différents auteurs et la qualité du code qu'ils viennent d'ajouter. Ces informations sont ensuite stockées dans la base de données à l'aide du composant **GutsyDB**. Nous appellerons l'ensemble de ces informations *Unité de Travail*.Une fois les écritures des différentes informations dans la base de donnée, le composant **SYSVN** commence le téléchargement du code dans le serveur principal.
Dans cette étape, le composant Monitoring Projet fait appel aux composants SYSVN et GutsyDB respectivement pour le téléchargement du code et l'écriture dans le base de données. Une nouveauté de cette nouvelle version est de se baser directement sur le serveur SVN du projet à analyser pour savoir si une nouvelle révision est disponible et connaître les différentes informations de la révision. Cela nous évite donc d'effectuer un accès supplémentaire sur un serveur mail comme le propose le premier prototype. Se baser directement sur le serveur SVN nous permet d'éviter aussi de devoir analyser le courriel qui dépend fortement du serveur SVN utilisé.

3.3 Étape 2 : Analyse du code source

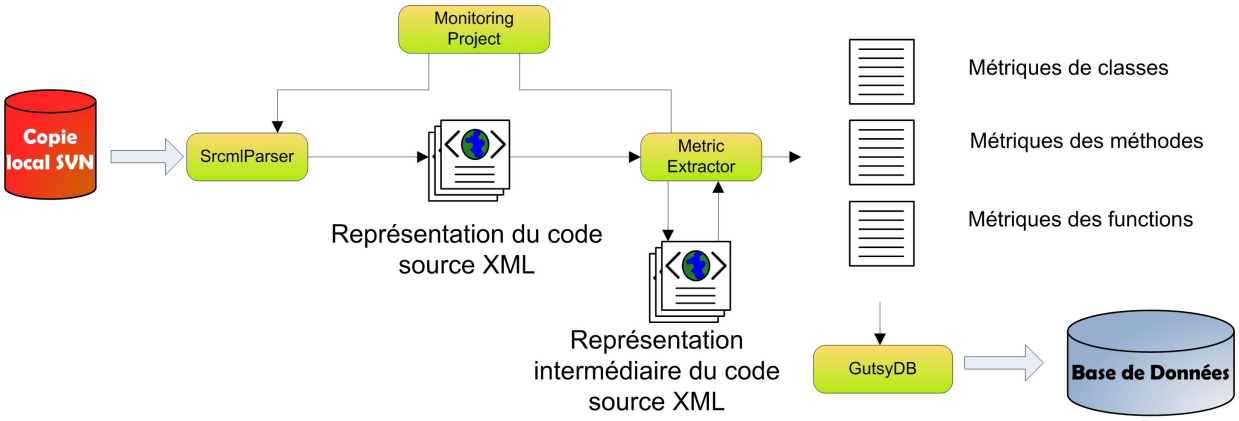


FIGURE 8 – Étape 2 : Analyse du code source

Les analyses qualité du code source se basent essentiellement sur des métriques. Des métriques sont des séries de mesure des propriétés d'un logiciel ou ses spécifications. À partir du répertoire local SVN contenu dans le serveur SOCCER, **Srcml** [2] qui est un logiciel Open-source, permet de convertir le code source de différents langages de programmation sous format Xml(Extensible Markup Language). Un exemple de transformation disponible sur le listing 1. Cette forme de représentation permet de définir le contenu du code sous forme hiérarchique. **Srcml** a la particularité de représenter tout le code y compris les sauts de ligne et les commentaires gardant ainsi intacts tous les aspects du code source.

Listing 1 – exemple de transformation de code source en code XML

```
1
2 <unit xmlns="http://www.sdml.info/srcML/src"
3       xmlns:cpp="http://www.sdml.info/srcML/cpp"
4       language="C++" filename="ex.cpp">
5     <comment type="line">
6         // copy the input to the output
7     </comment>
8     <while>while
9         <condition>
10            (<expr>
11                <name><name>std</name>::<name>cin</name></name> &gt;&gt; <name>n</name>
12            </expr>)
13        </condition>
14        <expr_stmt>
15            <expr>
16                <name><name>std</name>::<name>cout</name></name> &lt;&lt; <name>n</name> &lt
17            </expr>;
18        </expr_stmt>
19    </while>
```

```
20 </unit>
```

À partir de la représentation XML du code source, le composant **MetricExtractor** va en déduire les différentes métriques nécessaires pour l'analyse qualité. Ce composant procède donc en deux étapes pour parvenir à l'extraction des métriques. La première étape consiste à définir une grammaire intermédiaire toujours sous format XML, afin de définir les principales caractéristiques générales des classes et fonctions dont la *complexité cyclomatique* (Voir listing 2).
La *complexité cyclomatique* également appelé complexité d'un programme ou complexité de McCabe a été introduite par Thomas McCabe en 1976 [40]. La métrique de McCabe consiste indique le nombre de chemins linéaires indépendants dans un module de programme. Plus cette valeur est grande, plus il y aura de chemin possible, donc le module sera plus complexe. Dans notre cas, nous l'utiliserons pour déterminer la complexité dans les fonctions.

Listing 2 – Format xml intermédiaire pour le calcul des métriques

```
1   <?xml version="1.0" encoding="ISO-8859-1" ?>
2       <file name="sslmutex.h">
3       <classe>
4           <line_begin>66</line_begin>
5           <line_end>152</line_end>
6            <class_name>sslmutex/without_name_0</class_name>
7           <super_class />
8           <inner_class />
9           <operation_list>
10              <operation>
11                  <c_cyclo>1</c_cyclo>
12                  <line_begin>133</line_begin>
13                  <line_end>135</line_end>
14                  <visibility>public</visibility>
15                  <name>sslMutex_Init</name>
16                  <type>SECStatus</type> ?
```

3.4 Étape 3 : Calcul de la qualité

Dans cette partie, nous ne détaillerons pas les calculs permettant d'effectuer les différentes analyses de la qualité, nous en expliquerons seulement le principe de fonctionnement. Le système possède deux modules de qualité permettant de détecter les fonctions dupliquées et les fonctions considérées comme complexes appelées Monster. Les calculs s'effectuent à partir des métriques calculées par les étapes précédentes qui ont été sauvegardées dans la base de données. Les différents résultats seront ensuite sauvegardées dans la base de données.

Monster function analysis Le composant Monster function analysis consiste à déterminer la complexité d'une fonction. Les différentes fonctions sont notées selon quatre niveaux qui sont : **Correct**, **Warning**, **Baby-Monster** et **Monster**. Une

fonction considérée comme Monster est une fonction qu'il est fortement recommandé de la réviser. Quatre critères (métrique) sont utilisés pour l'analyse (Complixé de McCabe, Nombre de paramètres, Nombre de fonctions appelées, Nombre de lignes de codes (LOC)). Chaque critère sera donc noté selon les quatre niveaux. Le niveau final de la fonction sera du niveau le plus élevé de ses critères. Par exemple si la métrique LOC de la fonction est considérée comme **Monster**, la fonction sera notée comme **Monster**.
Les différents intervalles permettant de déterminer les différents niveaux sont calculés par le composant **StatisticUpdater**. La définition de ces intervalles utilise les techniques des statistiques descriptives. Les différents niveaux calculés correspondent aux différents quartiles obtenus à partir de la Loi normale [3]. La complexité d'une fonction dépend donc de l'ensemble des fonctions du projet à analyser.

Duplicate code analysis **Duplication code analysis** permet de déterminer les fonctions possédant du code identique. L'identification des clones s'effectuent à partir des métriques structurelles des fonctions telles le Nombre de structures conditionnelles, lignes de code, nombre de paramètres etc. [9].

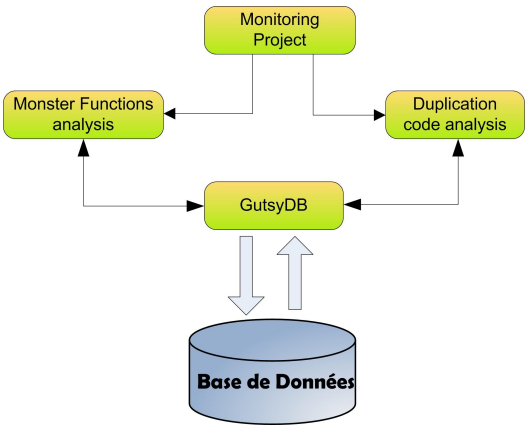


FIGURE 9 – Étape 3 : Calcul qualité

3.5 Étape 4 : Retour d'information

Le retour d'information est le coeur de la partie réactive du système. En effet, lorsque l'analyse est terminée et que toutes les données sont enregistrées, un courriel sera envoyé à l'équipe de développement pour les informer des dernières modifications au point de vue qualité sur le code source. Comme le montre le listing 3, le courriel fournit les informations sur les métriques de classe, la présence ou non de code dupliqué et la complexité des différentes fonctions modifiées. Le grand avantage

est de pouvoir avoir un aperçu de la qualité des différentes modifications effectuées par le développeur après chaque modification.

3.6 Les Web Services

Dans notre application les Web services vont servir de point de communication entre le serveur et l'extérieur. Nous pouvons en identifier deux types distincts.
Les Web services destinés aux administrateurs permettant de contrôler et surveiller l'application de l'extérieur. En offrant la possibilité de démarrer ou de stopper le système et surtout de le re-configurer sans avoir à redémarrer le système.
Et les services destinés aux utilisateurs. Les services primaires permettent aux clients de pouvoir inscrire leurs projets dans notre système et de le configurer. Cette manière de procéder ne nécessite donc aucune installation particulière. D'autres services, représentés par des fonctions retournant directement des informations de la base de données, permettent d'obtenir toutes les informations de la dernière analyse. Permettant de retrouver les fonctions dupliquées ou complexes dans le cadre d'une opération de maintenance général.

3.7 Conclusion

Le nouveau système remplit les premiers objectifs que nous nous étions fixés. C'est-à-dire que nous avons une architecture stable et performante du premier prototype. Le temps mis pour renvoyer le courriel contenant l'analyse est de 1 min à 5 min selon l'ampleur de la modification effectuée par l'équipe de développement. Le projet tourne depuis quelques mois sur le projet open source Grass [4]. Cependant, les perspectives de cet outil sont assez limitées. Nous pouvons relever encore deux majeures dans la structure actuelle. La première est que la conception ne permet pas d'étudier l'évolution tout au long de son cycle de vie car seulement les données de la dernière analyse sont sauvegardée dans la base de données. Le second défaut majeur est la difficulté de rajouter une analyse de qualité autre que celles déjà disponibles. Néanmoins, nous avons montré que le principe de fonctionnement général mérite d'être étudié plus en profondeur. La deuxième partie du stage consistera à recréer une nouvelle structure, tout en gardant l'idée générale, comblant les principaux défauts relevés.

Listing 3 – Exemple de courriel envoyé après analyse

```
Hello mmetz,

here's a report on your recent grass-SVN change for the revision : 43312

ANALYSIS

your changes to files/class/methods/functions :

*****************************************************************
File :/include/iostream/ami_stream.h
*****************************************************************
Class
AMI_STREAM<T>(Begins at: -9999, Ends at:-9999)
+---------+-----+-----+-----+-----+------+
| DIT | NOC | WMC | RFC | CBO | LCOM |
+---------+-----+-----+-----+-----+------+
|NEW Metrics | 1 | 0 | 38 | 66 | 0 | 66 |
+---------+-----+-----+-----+-----+------+
|OLD Metrics | 1 | 0 | 38 | 66 | 0 | 66 |
+---------+-----+-----+-----+-----+------+

Method
stream_len(Begins at: 382, Ends at:428)
-----------------+-----------+-------+---------------+-------------+
| New Metrics |Old Metrics|
-----------------+-----------+-------+---------------+-------------+
Complexity: | 4 | | OK | 4 |
ParamNBR: | 1 | | OK | 1 |
CalledNBR: | 20 | * | WARNING | 20 |
LOC: | 47 | | OK | 47 |
-----------------+-----------+-------+---------------+-------------+
Conclusion: | this function is a WARNING |
-----------------------------------------------------------------+
at least 1 values are close to an outlier please consider revising code!

Clone analysis:
This file does not contain cloned functions

Measures (see below Web page for details) :
-Complexity (Cyclo): a complexity index
-ParamNBR: number of passed parameters
-CalledNBR: number of called functions
-LOC: Lines Of Code
-Global analysis: based on 21 metrics(the fourth preceding include)

Message interpretation :
OK : Nothing to say at all
* : this is getting close to be out of range, consider revising the code
** : this is a baby monster, you should revise this function
***: this is a monster, definitely you should revise this function

Conclusion interpretation :
CORRECT : if all metrics are situated before warning limits
WARNING : if 1 to more metrics are between warning and middle limits
BABY_MONSTER: if 1 to more metrics are between middle and extreme limits
MONSTERS : if 1 to more metrics are in the extreme limits
-----------------------------------------------
Brought to you by :
SOftware Cost-effective Change and Evolution Research (SOCCER) laboratory

Please visite our web site on your project at :
http://web.soccerlab.polymtl.ca/project-manager/grass-website/index.html
```

4 Squaner

La deuxième partie du stage consistera à élaborer un nouveau projet nommé Squaner (Software QUality ANalyzER), reprenant les principes fondamentaux du chapitre précédent Gutsy. Cependant, nous voulons mettre l'accent sur deux points dans ce nouveau projet. Le premier sera celui de l'extensibilité, en créant un modèle permettant l'ajout de nouveau modèle de qualité et de nouvelles analyses facilement. Les laboratoires Ptidej et SOCCER concentrant essentiellement leurs effort sur la qualité logicielle et la détection des motifs dans le code source, l'objectif majeur de Squaner sera d'inclure les différents outils existant dans ces laboratoires. Le deuxième point sera de pouvoir avoir un suivi de toutes les analyses effectuées. À plus long terme, ces données pourront servir à faire des études sur l'évolution de la qualité d'un logiciel et l'évolution des motifs de conception. Nous verrons comment on pourra effectuer de telles études dans les perspectives.

4.1 Architecture de Squaner

La figure 4.1 et la figure 4.1 nous montrent l'architecture de Squaner. Nous pouvons remarquer que l'architecture globale est semblable à celle vue au chapitre précédent. Le principe de fonctionnement reste identique, c'est-à-dire que lorsqu'une modification est apportée au code source disponible dans le serveur SVN, le serveur télécharge le code modifié pour y effectuer une série d'analyses qui sera ensuite sauvegardée dans une base de données. Pour visualiser les résultats de ces analyses, un mail est toujours envoyé à l'équipe de développement. En plus d'un accès via web-service pour obtenir les résultats des analyses, nous avons créé en plus une interface web que nous décrirons plus tard.
Comme nous l'avons vu dans l'état de l'art, beaucoup de travaux dans la qualité logicielle commencent à étudier l'impact des patrons de conception et des anti-patrons de conception sur la qualité du logiciel. Avec Squaner, nous voulons donc inclure les dernières techniques de détection pour pouvoir par la suite y insérer des modèles s'appuyant sur les motifs de conception et non plus uniquement sur des éléments uniquement structurels. Comme le montre la figure 4.1, Squaner est une application construite sur plusieurs couches s'appuyant sur le modèle Squaner que nous allons voir dans le chapitre suivant. Nous aurons deux types de composants. Les composants d'analyse *(Composant en dessous du modèle Squaner)* permettant d'enrichir le modèle par les structures du modèle objet ainsi que les métriques et le design du code. Puis les différents modèles de qualité *(Composant au-dessus du modèle Squaner)* permettant l'évaluation du modèle. Dans les chapitres suivants, nous allons brièvement démontrer les différentes techniques utilisées pour l'analyse du code source ainsi que la détection des motifs de conception. Nous montrerons les différents modèles de qualité disponibles actuellement dans Squaner puis le nouveau processus permettant l'analyse complète du code source.

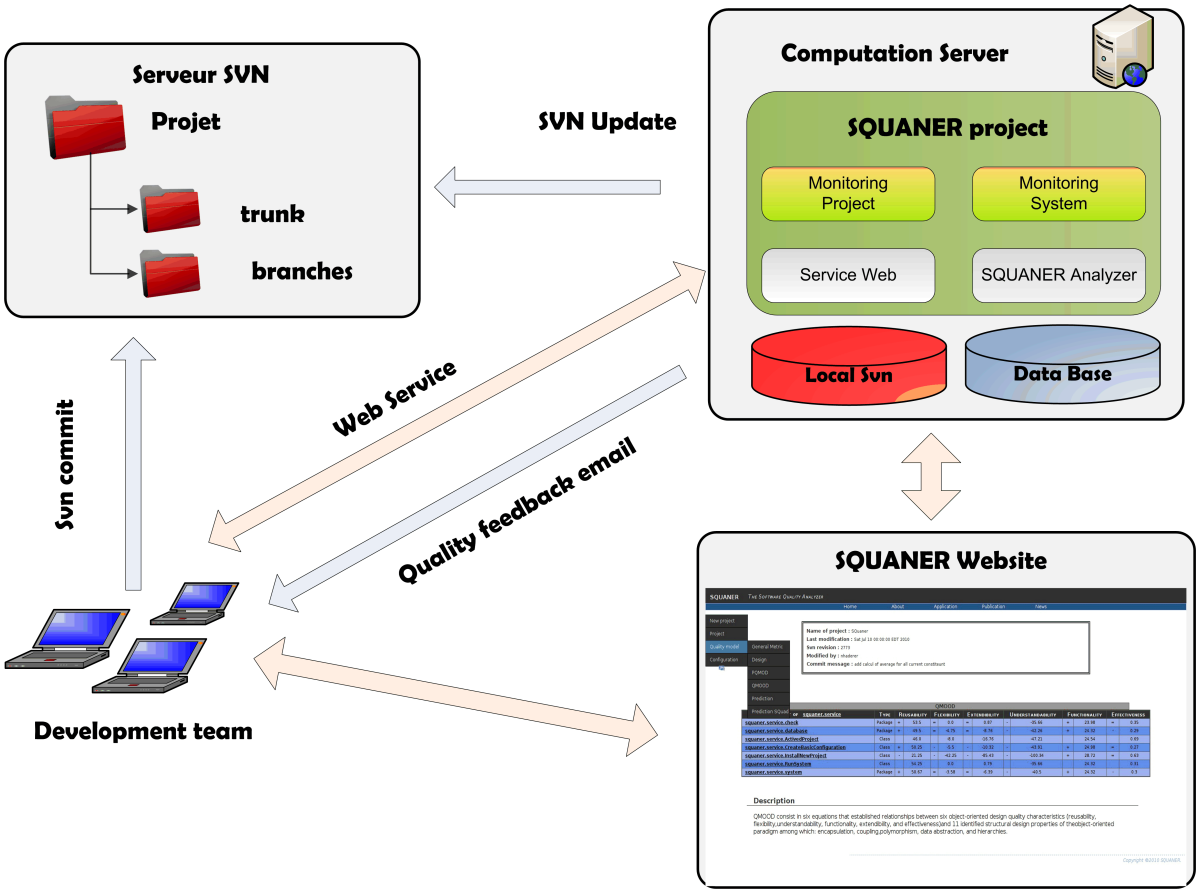


FIGURE 10 – Architecture de Squaner

4.2 Le modèle Squaner

Pour palier le problème de l'extensibilité du premier prototype, nous avons décidé de créer un modèle intermédiaire entre les calculs et la base de données pour plusieurs raisons. En effet, dans la première solution Gutsy, les différents résultats obtenus après l'analyse du code source étaient sauvegardés directement dans la base de données, rendant difficile l'ajout de nouvelles métriques ou bien des analyses d'autre type comme les patrons de conceptions. De plus, les différents algorithmes de qualité prenant directement leurs informations dans la base de données, leurs implémentations étaient plus difficiles car en plus de l'algorithme, nous devions gérer l'écriture et la lecture dans la base de données rendant le code moins compréhensible.

La figure 4.2 montre le noyau le modèle de Squaner. Ce modèle représente l'abstraction d'un modèle de qualité définie à partir de notre étude sur les différents

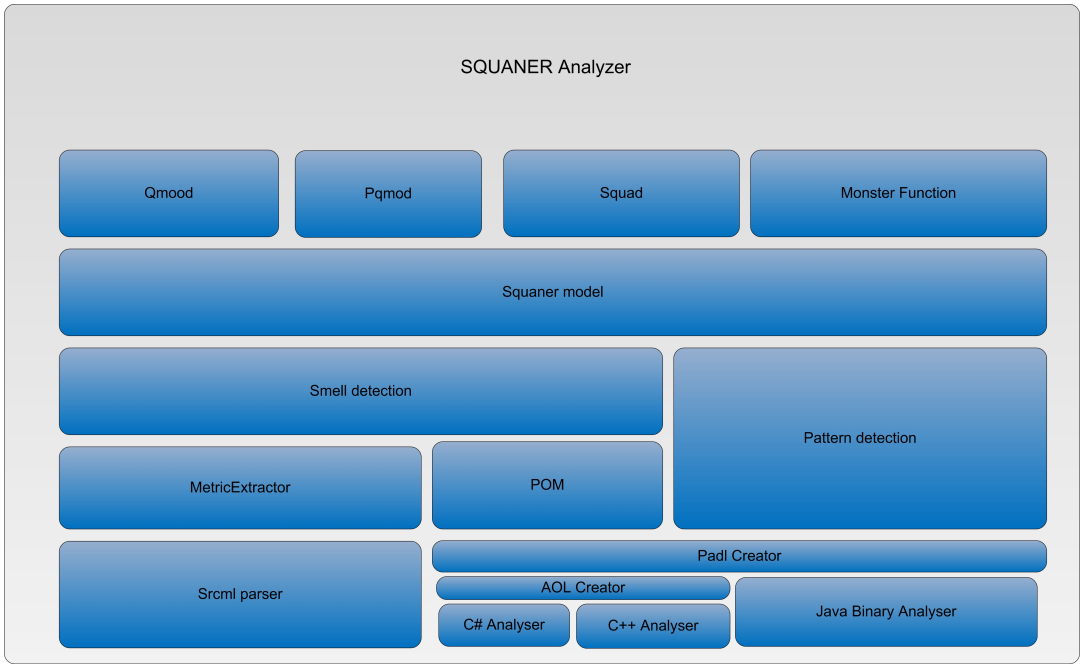


FIGURE 11 – Architecture des composants de Squaner

modèles de qualité vue dans l'état de l'art. Le point central du modèle est l'interface ISoftware correspondant à une version du projet à analyser. Chaque analyse possède un ensemble d'éléments (IConstituent) correspondant aux différents éléments du code Objet. Chaque élément possède un ensemble d'attributs de qualité correspondant à une analyse de celui-ci. Comme nous l'avons vu dans l'état de l'art sur les modèles de qualité, nous avons différencié les attributs de qualité en trois types. Les attributs numériques, correspondant à une évaluation numérique sur un élément du système comme des métriques (LOC, McCabe ...). Les attributs nominaux correspondant à une évaluation qualitative sur un élément ("Bien", "Neutre", "Mauvais"). Et enfin les attributs design correspondant à une structure bonne ou mauvaise comme les Patrons de conception et les Anti-Patrons de conception. Un modèle de qualité sera donc un objet parcourant l'ensemble de la structure grâce à un visiteur prenant les différents attributs de qualité nécessaire à son calcul. La création d'un nouveau modèle de qualité consistera en plusieurs étapes. La première sera d'étendre le modèle par héritage, en ajoutant les différents éléments que l'on veut analyser, (les packages, les classes par exemple pour le code Objet. Puis dans un second temps d'enrichir les différents éléments par des attributs de qualité. Nous aurons donc une structure en couche, avec des analyses de bas niveau telles que la détection du code source, des métriques, puis des analyses de plus haut niveau tel que les modèles de qualité ou la détection des motifs de conception.

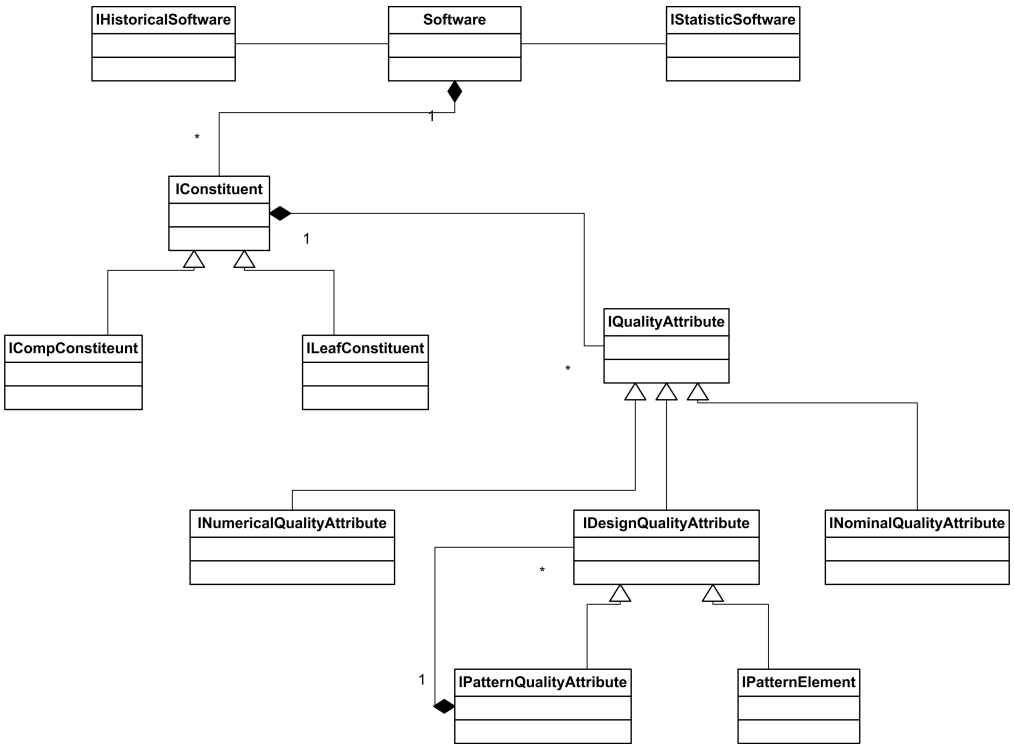


FIGURE 12 – Squaner modèle

4.2.1 Détection des motifs de conception

Pour l'analyse du code source, Squaner utilise le framework **Ptidej** (Pattern Trace Identication, Detection, and Enhancement in Java) by Guéhéneuc [28]. **Ptidej** est un Framework implémentant l'approche DeMINA initiée par Guéhéneuc et Antoniol [30] et l'approche de Kaczor [33] pour la détection des patrons de conception. En combinant ces deux approches, **Ptidej** est capable de détecter 13 motifs *(Abstract Factory, Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method, and Visitor.)*. **DeMINA** est une approche multi-couches pour l'identification des patrons de conception. La première couche consiste à créer un modèle abstrait (PADL) du code source incluant les relations binaires entre les classes puis une couche d'identification des motifs dans le modèle abstrait. Cette approche utilise la programmation par contrainte et permet de fournir les relations entre les différents rôles joués dans un patron et les classes. Dans la Figure 4.1, **Ptidej** regroupe les composants *Padl-Creator*, *PatternDetection*.

La détection des métriques s'effectue à l'aide du composant **POM**. POM est un framework s'appuyant sur le méta-modèle **PADL** de **Ptdej**, offrant plus de 60 métriques issues de la littérature incluant le couplage entre les objets (CBO), la complexité total d'une classe (WMC) [18], la manque de cohésion dans les méthodes (LCOM5) [31], la connectivité d'une classe (C) [32], nombres de méthodes héritées ou surchargées [39], la complexité cyclomatique (CC) [41], la profondeur d'héritage d'une classe [52] etc. La définition de toutes les métriques implémentées par **POM** est disponible en annexe.

Toujours sur le modèle abstrait PADL présent dans Ptidej qui modélise le code source, Moha [46], [44] creer la méthode DECOR qui est une méthode de détection de mauvais codes *(code smell)* et des Anti-patrons de conceptions. On rapelle que les Anti-patrons sont des erreurs courantes en programmation lors de la conception des logiciels. La méthode DECOR reporte des algorithmes pour la détection des Anti-patrons définis par Brown et Flowler
[17], [24] qui sont :

ClassDataShouldBePrivate (CDSBP), ComplexClass, Large-Class, LazyClass, LongMethod, LongParameterList (LPL), MessageChains, RefusedParentBequest (RPB), SpaghettiCode, SpeculativeGenerality (SG), SwissArmyKnife.

Ces anti-patrons sont tous représentatifs des problèmes d'implémentation avec les données, la complexité, la taille et les fonctionnalités fournies par les classes. En supplément de ces algorithmes de détection, DECOR fournit des algorithmes de détection de code smell (Les anti-patterns sont définis comme par un ensemble de code smell, cependant certain code smell peuvent être considérés comme anti-patron) avec une précision de 80% en moyenne sur les code smells suivants :

AbstractClass, ChildClass, ClassGlobalVariable, ClassOneMethod,ComplexClassOnly, ControllerClass, DataClass, FewMethods, FieldPrivate, FieldPublic, FunctionClass, HasChildren, LargeClass, LargeClassOnly, LongMethod, LongParameterList- Class, LowCohesionOnly, ManyAttributes, MessageChainsClass, MethodNoParameter, MultipleInterface, NoInheritance, NoPolymorphism, NotAbstract, NotComplex, OneChildClass, ParentClassProvidesProtected, RareOverriding, TwoInheritance. Une description des différents patrons et anti-pattrons de conception est disponible en annexe.

4.3 Les modèles de qualité

Le but des modèles de qualité est d'effectuer une évaluation sur le code source d'un projet. Une fois réalisé la première partie qui est l'analyse du code source, le Squaner se retrouve enrichi des divers éléments du code tels que les packages et les classes et des métriques fournies par le framework **POM** vu précédent ainsi que la détection des différents motifs de conceptions. Les modèles de qualité que nous allons présenter maintenant s'appuient sur les éléments déjà présents dans le modèle.

4.3.1 QMOOD

QMOOD introduit par Bansiya et Davis [10], définit une série de six équations permettant d'évaluer les systèmes orientés objet selon six caractéristiques qui sont réutilisabilité, flexibilité, compréhensibilité, fonctionnalité, extensibilité et efficacité. Pour établir ces équations, l'auteur établit des relations entre 11 propriétés du paradigme objet et les caractéristiques qu'il veut obtenir. Malheureusement, l'auteur ne donne que très peu d'information sur l'élaboration de ces relations et la pondération de ces équations. Il en reste néanmoins que c'est un des modèles références d'aujourd'hui ayant fait ses preuves sur un framework très populaire appartenant à la suite Microsoft.
Voici les équations et la définition des caractéristiques du modèle **QMOOD**

Réutilisabilité Reflète la capacité d'un logiciel à pour pouvoir s'adapter à un nouveau problème avec peu d'effort.

*Réutilisabilité = -0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size*

Flexibilité Capacité du code à pouvoir intégrer de nouvelles fonctionnalités dans la conception.
*Flexibilité = 0.25 * Encapsulation - 0.25 * Coupling + 0..5 * Composition + 0.5 * Polymorphism*

Compréhensibilité Capacité du code à être appris et compris facilement. Cette caractéristique est directement relié à la complexité de la structure du code.
*Compréhensibilité = -0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism -0.33 * Complexity - 0.33 * Design Size*

Fonctionnalité Responsabilités confiées aux classes à travers leurs interfaces publiques.

*Fonctionnalité = 0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * Design Size + 0.22 * Hierarchy*

Extensibilité Réfère à la présence et l'utilisation des propriétés dans un modèle existant qui permettent l'incorporation de nouvelles exigences dans la conception.

*Extensibilité = 0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism*

Efficacité Capacité du code à effectuer les fonctionnalités désirées.

*Efficacité = 0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism*

Propriété	Métrique Associée	Acronyme
Coupling	Direct Class Coupling	DCC
Cohesion	Cohesion Among Methods of Class	CAM
Messaging	Class Interface Size	CIS
Design Size	Design Size in Classes	DSC
Encapsulation	Data Acces Metric	DAM
Composition	Measure of Functional Abstraction	MOA
Polymorphisme	Number of Polymorphic Methods	NOP
Abstraction	Average Number of Ancestors	ANA
Complexity	Number of Methods	NOM
Hierarchies	Number of Hierarchies	NOH
Inheritance	Measure of Functional Abstraction	MFA

*Correspondance entre les propriétés et les métriques disponibles dans **POM***
Une définition plus approfondie des métriques est disponible en annexe.

QMOOD nous donne une série d'équations pour mesurer certaines propriétés du paradigme objet. Cependant, le modèle ne nous permet pas de définir si une propriété est bonne ou mauvaise. Cela permet juste d'étudier l'évolution de ces propriétés au cours de la conception du code et voir ainsi son amélioration et sa dégradation. Une piste suivie pour pouvoir déterminer si une propriété est bonne ou mauvaise est de comparer les différentes valeurs obtenues avec les valeurs obtenues à partir de logiciel réputé pour leurs qualités. Nous pourrions alors définir des intervalles à travers lesquels on pourrait définir si la valeur d'une propriété est bonne ou mauvaise. La difficulté rencontrée est que les valeurs obtenues dépendent non seulement des propriétés mesurées mais aussi de la taille générale du projet analysé. Des travaux futurs nous permettrons d'établir ces coefficients de proportionnalité en étudiant des logiciels réputés comme étant de qualité et de différentes tailles.

4.3.2 PQMOD

PQMOD issu de la méthode **DEQUALITE** [35] (Design Enhanced QUALITy Evaluation) est un modèle de qualité prenant en compte non seulement les attri-

buts interne des systèmes (métriques) mais aussi de leurs conceptions. La méthode **DEQUALITE** est une méthode définissant 5 étapes pour permettre l'élaboration de modèles de qualité prenant en compte la conception d'un système. **PQMOD** est le premier modèle issu de cette méthode. Comme le modèle **QMOOD**, **PQMOD** définit 7 règles permettant d'effectuer une évaluation qualitative sur 7 propriétés du paradigme objet. Le modèle se base sur deux hypothèses qui sont que les patrons de conception améliorent la qualité d'un logiciel et que les objets (les classes) mesurés jouent au moins un rôle dans un patron de conception. À partir de ces hypothèses, un ensemble de règles a été défini pour établir l'impact (Bonne Neutre ou Mauvaise) d'une métrique interne sur une propriété à partir des arbres de décision. Au final, PQMOD définit un ensemble de règles permettant de définir à partir de métriques internes du système, si une propriété est bonne mauvaise.

Voici les règles permettant d'évaluer 7 propriétés du paradigme objet défini par **PQMOD** :

Extensibilité Degrés auquel la conception du système peut être étendu.

Listing 4 – Règle de la propriété Extensibilité

```
1 NOA <= 2
2     AID <= 0,5 : Valeur : =
3     AID > 0,5 : Valeur : -
4 NOA > 2
5     NOC <= 0
6         DIT <= 0.83 : Valeur : +
7         DIT > 0,83 : Valeur : -
8     NOC > 0 : Valeur : +
```

Généralité Le degré selon lequel un système fournit une vaste gamme de fonctions à l'exécution.

Listing 5 – Règle de la propriété Généralité

```
1 NOC <= 0
2     AID <= 0,83
3         AID <= 0,11 : Valeur : =
4         AID > 0,11 : Valeur : +
5     AID > 0,83 : Valeur : =
6 NOC > 0
7     ACMIC <= 1 : Valeur : +
8     ACMIC > 1 : Valeur : =
```

Modularité Degré selon lequel les fonctions d'un système sont indépendantes les unes des autres.

Listing 6 – Modularité

```
1 NCM <= 50,63
```

```
2     NOC <= 0
3         AID <= 0,83
4             AID <= 0,11 : Valeur : =
5             AID > 0,11 : Valeur : +
6         AID > 0,83 : Valeur : =
7     NOC > 0 : Valeur : +
8 NCM > 50,63
9     NOP <= 1,65 : Valeur : =
10    NOP > 1,65 : Valeur : +
```

Modularité pendant l'exécution Degré selon lequel les fonctions d'un système sont indépendantes les unes des autres pendant l'exécution.

Listing 7 – Règle de la propriété Modularité pendant l'exécution

```
1 (LCOM5 >= 1,02) et (DCAEC <= 0 ) : Valeur : =
2 (NCM >= 62,78) : Valeur : =
3 (NOA <= 2) : Valeur : =
4 : valeur : +
```

Réutilisation Degré selon lequel une pièce de la conception peut être réutilisée dans une autre conception.

Listing 8 – Règle de la propriété Réutilisation

```
1 NOC <= 0,23 et AID >=1 : Valeur : =
2 NMA >= 16 : Valeur : =
3 :Valeur : +
```

Passage à l'échelle Degré selon lequel le système peut gérer une grande quantité de données pendant l'exécution.

Listing 9 – Règle de la propriété Passage à l'échelle

```
1     (DCMEC >= 0,1) et (NOC <= 0,65) et (DCMEC <= 0,2)   : valeur : +
2     (AID >= 2,83) et (AID <= 2,92) : Valeur : +
3     (connectivity <= 0,66) et (NOC >= 0,75) et ACMIC <= 0,2 : Valeur : -
4     (LCOM5 <= 0,63) et (ICHClass >= 0,25) et (NMA <= 5,88) : Valeur : -
5     : Valeur : =
```

Compréhensibilité Degré selon lequel le système peut être compris facilement

Listing 10 – Règle de la propriété Compréhensibilité

```
1 NCM <= 4
2     CBO <= 1 : Valeur : =
3     CBO > 1 : Valeur : +
4 NCM > 4
5     ACMIC <= 0 : Valeur : +
6     ACMIC > 0
7         CLD <= 0,46
8             NOD <= 1,63
9                 NOP <= 4,67
10                    DCMEC <= 0,2 : Valeur : +
```

```
11                      DCMEC > 0,2
12                          AID <= 2,15 : Valeur : -
13                          AID > 2,15 : Valeur : +
14                  NOP > 4,67 : Valeur : -
15              NOD > 1,63 : Valeur : -
16          CLD > 0,46 : Valeur : +
```

PQMOD est un modèle ayant une bonne capacité de prédiction sur les systèmes comportant un grand nombre de patrons de conception et reste néanmoins limité pour les systèmes faisant peu usage d'héritage et n'implémentant pas beaucoup les patrons de conceptions.

4.3.3 HBugPredict

HBugPredict de Kohmh [36] est le premier modèle de qualité se basant sur l'aspect de la conception en prenant en considération les patrons de conception et les anti-patrons. Ce modèle est un modèle de prédiction tentant de donner une probabilité aux futures erreurs d'un logiciel. Basé sur les réseaux Bayésiens [6] pour l'apprentissage, le modèle détermine grâce l'historique des erreurs du logiciel, la relation entre les classes jouant un ou plusieurs rôles dans des patrons ou un anti-patron de conception, la probabilité pour cette même classe d'être sujette à une erreur dans le futur. L'apprentissage du modèle se déroule sur une durée de six mois et donne une probabilité pour les six mois à venir. En résultat, comme nous montre la figure 4.3.3, le réseau nous donne une table de probabilités qui nous permet d'inférer selon les noeuds (caractéristique de la classe) d'entrée du réseau Bayésien :

- PRE : nombre de fautes de la classe depuis six mois.
- AP : joue un rôle dans un antipatron de conception.
- DP : joue un rôle dans un patron de conception.

Le modèle utilisé dans Squaner a été entraîné sur les versions majeures d'eclipse et possède une fiabilité de 70 % sur la prédiction. Cependant, pour utiliser pleinement ce modèle, nous avons besoin d'être en mesure de pouvoir reporter toutes les erreurs d'un système analysé dans Squaner. Une solution retenue est de passer par les web-services, ainsi l'équipe de développement pourra avertir Squaner dès qu'une erreur sera détectée dans leur système. Mais cette solution nécessite une grande rigueur de la part de l'équipe de développement et ne permet pas de reporter les erreurs des utilisateurs de leurs systèmes. Nous verrons dans les perspectives comment nous pourrions aborder ce problème d'une autre manière.

4.3.4 BugPredict

Ce système est une variante du modèle **HBugPredict**. Toujours basé sur les réseaux Bayésiens, ce modèle a été entraîné sans prendre en compte l'historique des erreurs du système. En effet, le dernier demandait un suivi d'au moins six mois avant de pouvoir être utilisé à pleine capacité. Nous prenons toujours en compte les rôles

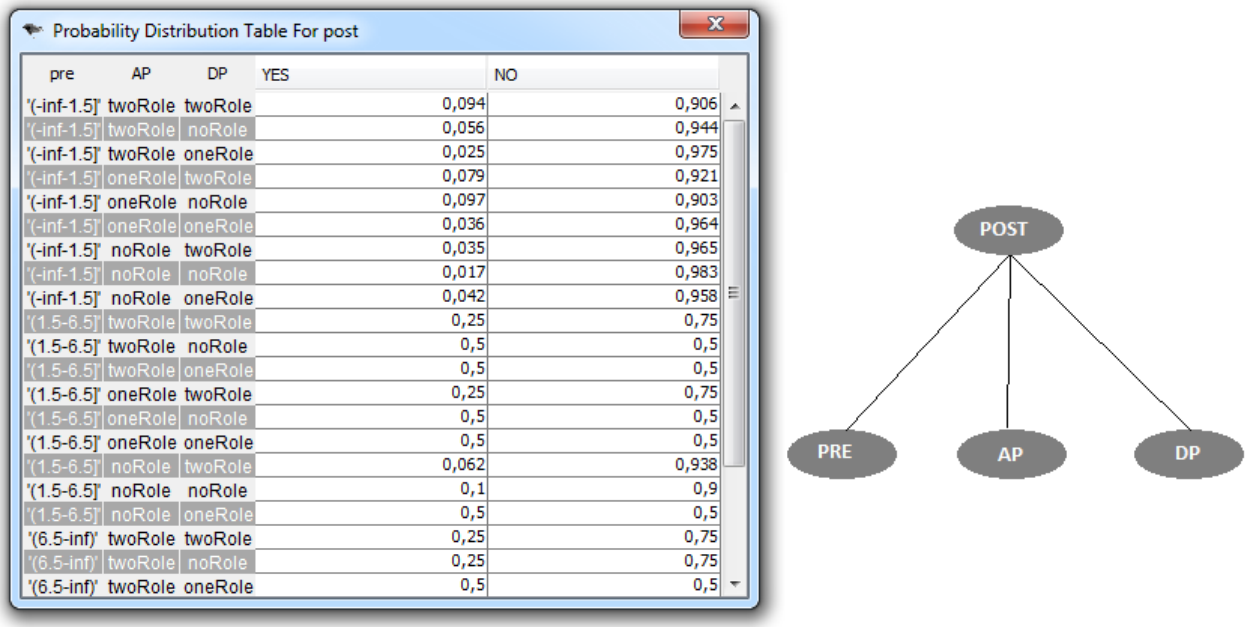


FIGURE 13 – Réseau Bayésien du modèle HBugPredict

des classes joués dans les patrons de conceptions et anti-patrons de conceptions mais nous avons rajouté les métriques proposées par Zimermann [54] proposé dans son modèle de prédiction.

Comme le montre la figure 4.3.4, les inférences dans le réseau se feront à partir des attributs suivants :

- TLOC : Nombre de ligne de code d'un classe
- MLOC : Somme des lignes de code des fonctions incluent dans la classe
- AP : joue un rôle dans un anti-patron de conception.
- DP : joue un rôle dans un patron de conception.

Nous avons mis en place ce modèle car avec le modèle précédent **HBugPredict**, nous avions besoin d'un suivi du logiciel par Squaner de six mois avec report d'erreur pour pouvoir profiter pleinement du modèle. Ce modèle a donc l'avantage de pouvoir être utilisé directement. Bien que les nouvelles métriques et le rôle joué dans les différents patrons ayant une influence sur la prédiction, il en reste néanmoins que l'expérience du logiciel est l'attribut le plus significatif. Nous nous retrouvons donc avec beaucoup de cas d'indécision (une probabilité de 0,5) pour la prédiction.

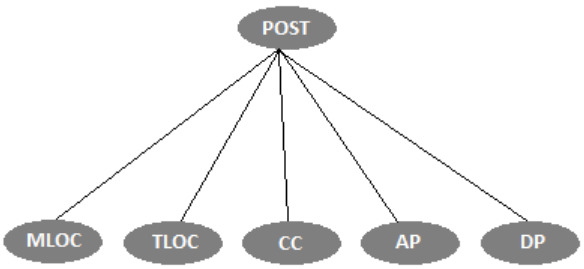


FIGURE 14 – Réseau Bayésien du modèle BugPredict

4.4 Analyse qualité du code source

Dans cette section, nous allons expliquer les différentes étapes afin d'établir une analyse complète du code source. Comme nous pouvons le remarquer sur la figure 4.4

Étape 1 : Création du modèle *PADL* Une fois le code source téléchargé depuis le serveur SVN, le code est analysé par le composant *PADLCreator*. Ce dernier est un composant regroupant d'autres composants destinés théoriquement à analyser le code source Java, C++,C# et le code exécutable Java. Dans cette version de Squaner, seuls les exécutables Java et les fichier C++ sont disponibles. Les autres composants devraient être disponible prochainement. Le rôle du composant PADL-Creator, basé sur le méta-modèle PADL [27], est d'analyser le code source pour créer une instance du méta-modèle comportant tous les éléments du code objet ainsi que les différentes relations entre les entités. Nous allons ensuite parcourir le modèle *PADL* pour enrichir le modèle Squaner uniquement avec les éléments destinés à être analysés. Nous ne garderons que les classes et les packages.

Étape 2 : Détection La seconde étape sera de lancer les différents algorithmes de détection pour enrichir à nouveau le modèle. Les métriques sont calculées par le composant *POM*. La détection des patrons de conception est effectuée par le composant *Pattern detection* et la détection des anti-patrons par le composant *Smell detection*. Les calculs sont effectués directement dans le PADL, possédant toutes les informations nécessaires. Les résultats seront les attributs de qualité des entités du modèle Squaner détecté à l'étape 1.

Étape 3 : Calcul de qualité Au début de l'étape 3, le modèle Squaner est enrichi par les entités du code objet ainsi que les attributs de qualité associés (métriques et détection de motifs). Les modèles vont s'appuyer sur ces attributs pour évaluer chaque entité. Les modèles évaluant essentiellement les classes, nous verrons dans l'étape suivante comment mettre en place un procédé pour évaluer les entités supérieures hiérarchiquement comme les packages. Après avoir effectué l'évaluation des entités, le modèle de qualité va de nouveau ré-enrichir le modèle de Squaner par ses résultats. Nous pouvons donc avoir ce que l'on appelle des modèles hiérarchiques, c'est-à-dire des modèles s'appuyant sur les résultats d'autres modèles.

Étape 4,5 Les analyses qualité terminées, nous allons à présent sauvegarder la version du système analysé dans la base donnée. La base de données contient toutes les versions d'un système ainsi que ses analyses. Une série de calculs est effectuée pour y détecter les changements du système et y établir des statistiques. Comme nous l'avons vu dans l'étape 3, les modèles de qualité évaluent essentiellement les classes du système. Pour donner une vue globale de la qualité et donner une évaluation pour

les entités de plus haut niveau comme les packages, nous procédons à une série de calculs. Les valeurs numériques des attributs de qualité correspondant aux packages auront la moyenne de tous les éléments qu'ils possèdent.

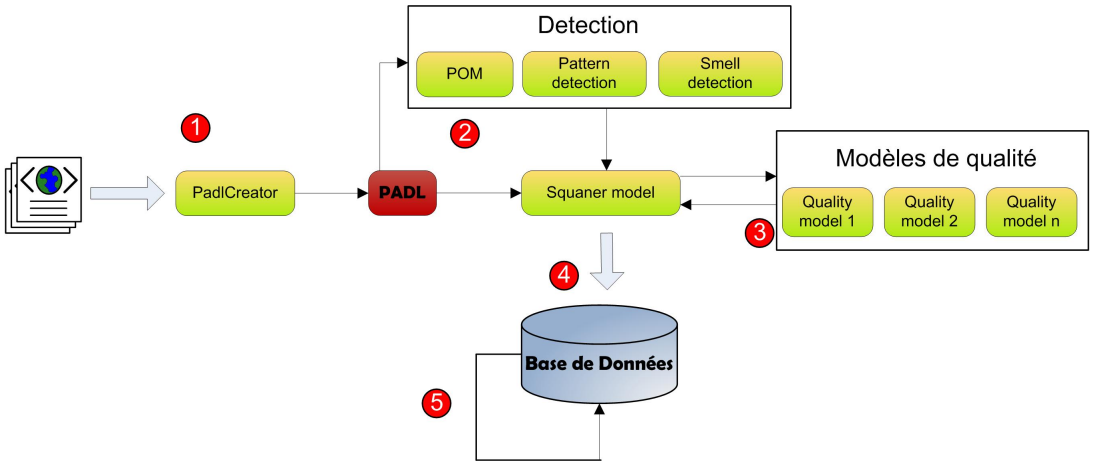


FIGURE 15 – Étape pour l'analyse complète du code source

Étape 6 L'Étape 6 consiste à envoyer un courriel à l'équipe de développement pour les informer sur la qualité modifiée. Le mail fournit donc les métriques globales du système avec leurs évolutions par apport à la dernière modification. Puis dans un second temps, 20 % des classes comportant le maximum de défauts (classe incluse dans un anti-patron) ainsi que leurs défauts associé. À court terme, nous voulons ajouter en accompagnant les défauts, les restructurations de code disponible dans la littérature. Le listing 11 montre un exemple de courriel envoyé.

4.5 Interface Web

Une autre amélioration que nous avons apportée est la création d'une interface web pour visualiser toutes les analyses qualité des projets suivis. Nous avons utilisé la technologie JSP, basée sur le langage Java, permettant de générer du code HTML dynamiquement. L'interface web est directement connectée à notre base données, contenant les analyses de Squaner.

Lorsqu'un utilisateur se connecte sur le site. L'utilisateur a la possibilité de choisir de sélectionner son projet. Comme le montre la figure 4.5, l'utilisateur peut sélectionner un modèle de qualité correspondant aux différentes analyses effectuées sur le projet. Les informations relatives à la dernière modification du SVN sont disponibles à travers un panel.

Comme les figures 4.5 et 4.5, les évaluations des modèles de qualité sont disponibles à travers des tableaux montrant la valeur pour chaque caractéristique mesurée ainsi que l'évolution de cette caractéristique par rapport à la dernière version analysée. Nous pouvons parcourir toute la structure du code en cliquant directement sur les éléments tels que les packages ou les classes. Les différents motifs (patron ou anti-patron de conception) détectés peuvent être afficher directement en mettant la souris en surbrillance sur l'élément sélectionné.

Pour enregistrer un nouveau projet, il suffit de remplir le formulaire présenté sur la figure 4.5. Le formulaire demande des informations sur le serveur SVN associé au projet, le langage utilisé puis une adresse email pour envoyer les différents résultats des analyses.

Actuellement, le site web n'est pas sécurisé et ne possède pas de compte utilisateur. Toutes les données calculées par Squaner sont donc accessibles par tous pour l'instant. Le site web est accessible à travers l'adresse [7].

Listing 11 – Extrait de courriel envoyé après analyse avec Squaner (1/2)

```
1
2
3 Hello
4 here's a report on your recent commit change for the revision : 3152
5 ************************************************
6 Project : JHOT
7 Date : Wed Sep 01 01:08:00 CEST 2010
8 Commited by : nhaderer
9 Commited messsage :
10 ************************************************
11 Average of metrics of the system :
12 ++-------------------+-------------------+-------------------+-------------------++
13 |************|  DIT      |        SIX      |        WMC      |
   CBO      |        LCOM5    |        McCabe   |        LOC      |
14 ++-------------------+-------------------+-------------------+-------------------++
15 |New metrics   | 2.29     |        0.28     |        230.03   |
   84.45    |        0.63     |        109.5    |        378.26   |
16 ++-------------------+-------------------+-------------------+-------------------++
17 |Old metrics   | -1.0     |        -1.0     |        -1.0     |
   -1.0     |        -1.0     |        -1.0     |        -1.0     |
18 ++-------------------+-------------------+-------------------+-------------------++
19
20 Definition :
21
22 DIT : (Average) Returns the DIT (Depth of inheritance tree) of the entity. Uses a recursive
23 SIX : (Average) Returns the SIX (Specialisation IndeX) of an entity.
24 WMC : (Average) Weight of an entity as the number of method invocations in each method
25 CBO : (Average) Coupling Between Objects of one entity
26 LCOM5 : (Average) Lack of COhesion in Methods of an entity.
27 McCabe : (Average) McCabe Complexity: Number of points of decision + 1
28 LOC : (Average) Return the numbers of lines of code of all the methods of a class.
29
30
31
32
33 ****************************************
34 Class DecoratorFigure
35 ****************************************
36
37 Bug probability : 0.3427083333333333
38
39 Design analysis :
40 This class contains 2 smells
41
42 +-----+-----+-----+-----+-----+-----+-----+-----+
43 Smell : Blob
44 +-----+-----+-----+-----+-----+-----+-----+-----+
45 LargeClass = CH.ifa.draw.standard.DecoratorFigure
46 Cause :
47 1.100.LargeClass-0.NMD_NAD-0=36.0
48 1.100.LargeClass-0.NMD_NAD_MaxBound-0={NMD_NAD_MaxBound=34.0}
49 DataClass = CH.ifa.draw.framework.FigureChangeEvent
50 Cause :
51 2.100.DataClass-0.NMD+NAD-0=7.0
52 2.100.DataClass-0.NMD+NAD_MaxBound-0={}
53 +-----+-----+-----+-----+-----+-----+-----+-----+
54
55
56 +-----+-----+-----+-----+-----+-----+-----+-----+
57 Smell : ComplexClass
58 +-----+-----+-----+-----+-----+-----+-----+-----+
59 LargeClassOnly = CH.ifa.draw.standard.DecoratorFigure
60 Cause :
61 1.100.LargeClassOnly-0.NMD_NAD-0=36.0
62 1.100.LargeClassOnly-0.NMD_NAD_MaxBound-0={NMD_NAD_MaxBound=34.0}
63 +-----+-----+-----+-----+-----+-----+-----+-----+
64 ****************************************
65 ------------------------------------------------
66 Brought to you by :
67 SOftware Cost-effective Change and Evolution Research (SOCCER) laboratory
68 Pattern Trace Idententification, Detection and Enhancement in Java (Ptidej)
69 Please visite our web site on your project at :
70 http://squaner.khomh.net/
```

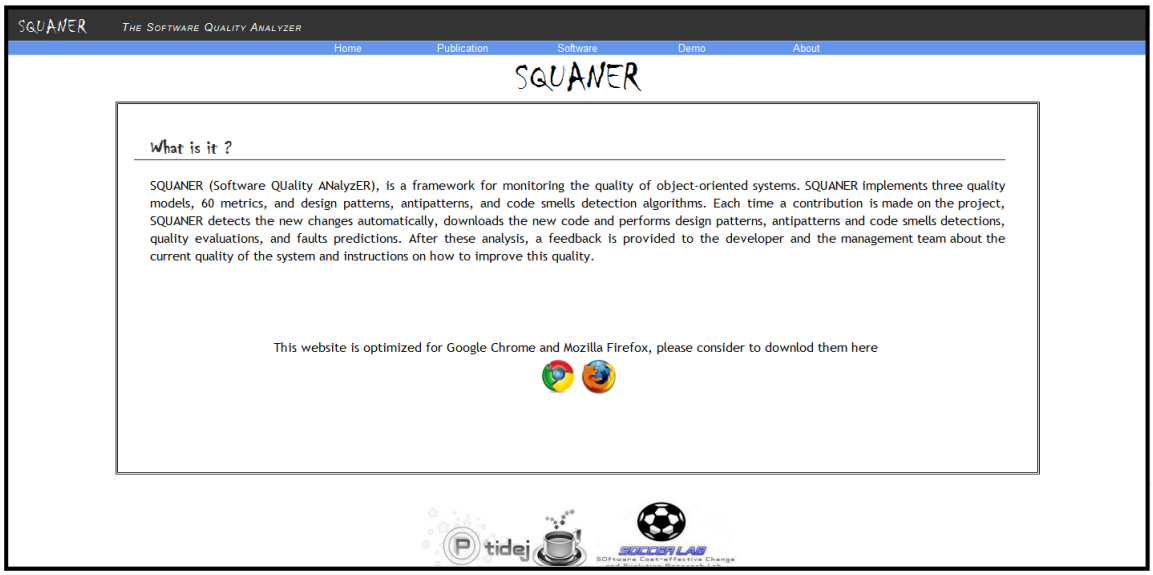


FIGURE 16 – Page d'accueil de Squaner

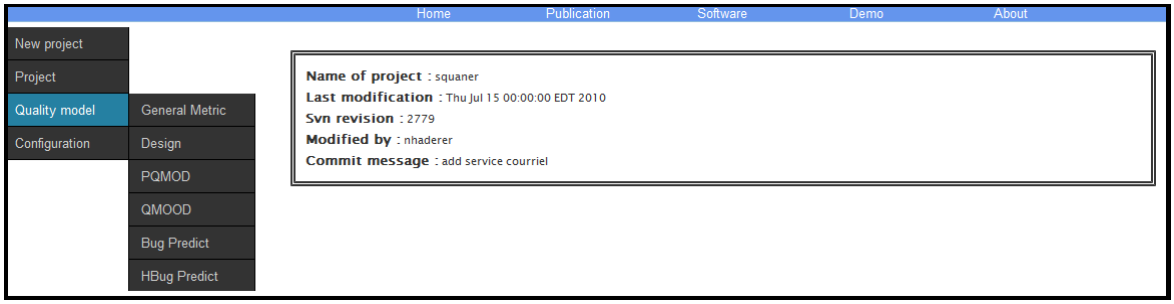
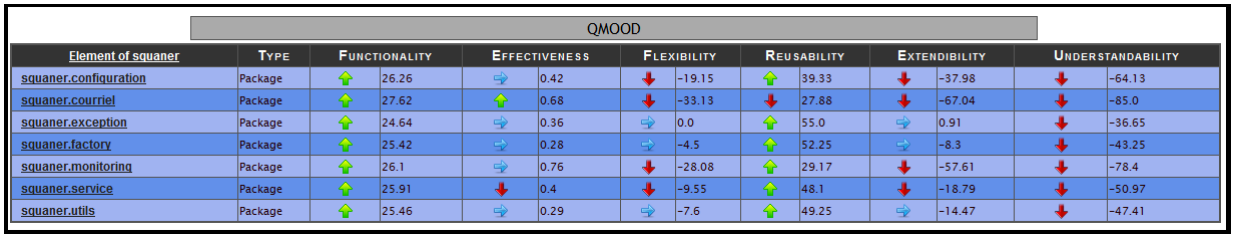


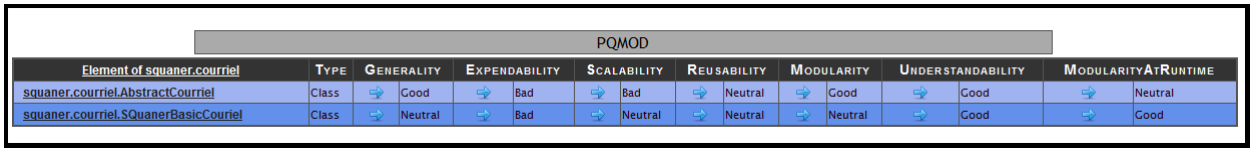
FIGURE 17 – Capture d'écran de Squaner



QMOOD

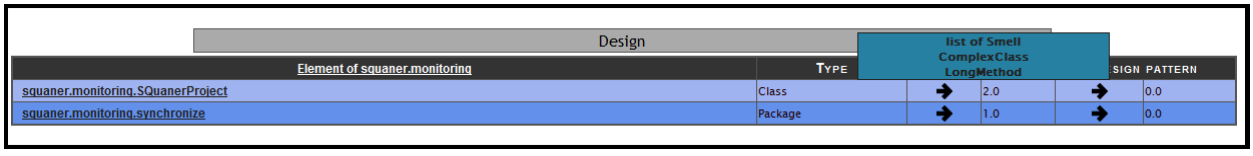
Element of squaner	Type	Functionality		Effectiveness		Flexibility		Reusability		Extendibility		Understandability	
squaner.configuration	Package	↑	26.26	→	0.42	↓	-19.15	↑	39.33	↓	-37.98	↓	-64.13
squaner.courriel	Package	↑	27.62	↑	0.68	↓	-33.13	↓	27.88	↓	-67.04	↓	-85.0
squaner.exception	Package	↑	24.64	→	0.36	→	0.0	↑	55.0	→	0.91	↓	-36.65
squaner.factory	Package	↑	25.42	→	0.28	→	-4.5	↑	52.25	→	-8.3	↓	-43.25
squaner.monitoring	Package	↑	26.1	→	0.76	↓	-28.08	↑	29.17	↓	-57.61	↓	-78.4
squaner.service	Package	↑	25.91	↓	0.4	↓	-9.55	↑	48.1	↓	-18.79	↓	-50.97
squaner.utils	Package	↑	25.46	→	0.29	→	-7.6	↑	49.25	→	-14.47	↓	-47.41

FIGURE 18 – Capture d'écran du modèle de qualité QMOOD



PQMOD															
Element of squaner.courriel	Type	Generality		Expendability		Scalability		Reusability		Modularity		Understandability		ModularityAtRuntime	
squaner.courriel.AbstractCourriel	Class		Good		Bad		Bad		Neutral		Good		Good		Neutral
squaner.courriel.SQuanerBasicCouriel	Class		Neutral		Bad		Neutral		Neutral		Neutral		Good		Good

Figure 19 – Capture d'écran du modèle de qualité PQMOD



Design					
		list of Smell ComplexClass LongMethod			
Element of squaner.monitoring	Type				sign pattern
squaner.monitoring.SQuanerProject	Class		2.0		0.0
squaner.monitoring.synchronize	Package		1.0		0.0

Figure 20 – Capture d'écran de la détection des motifs

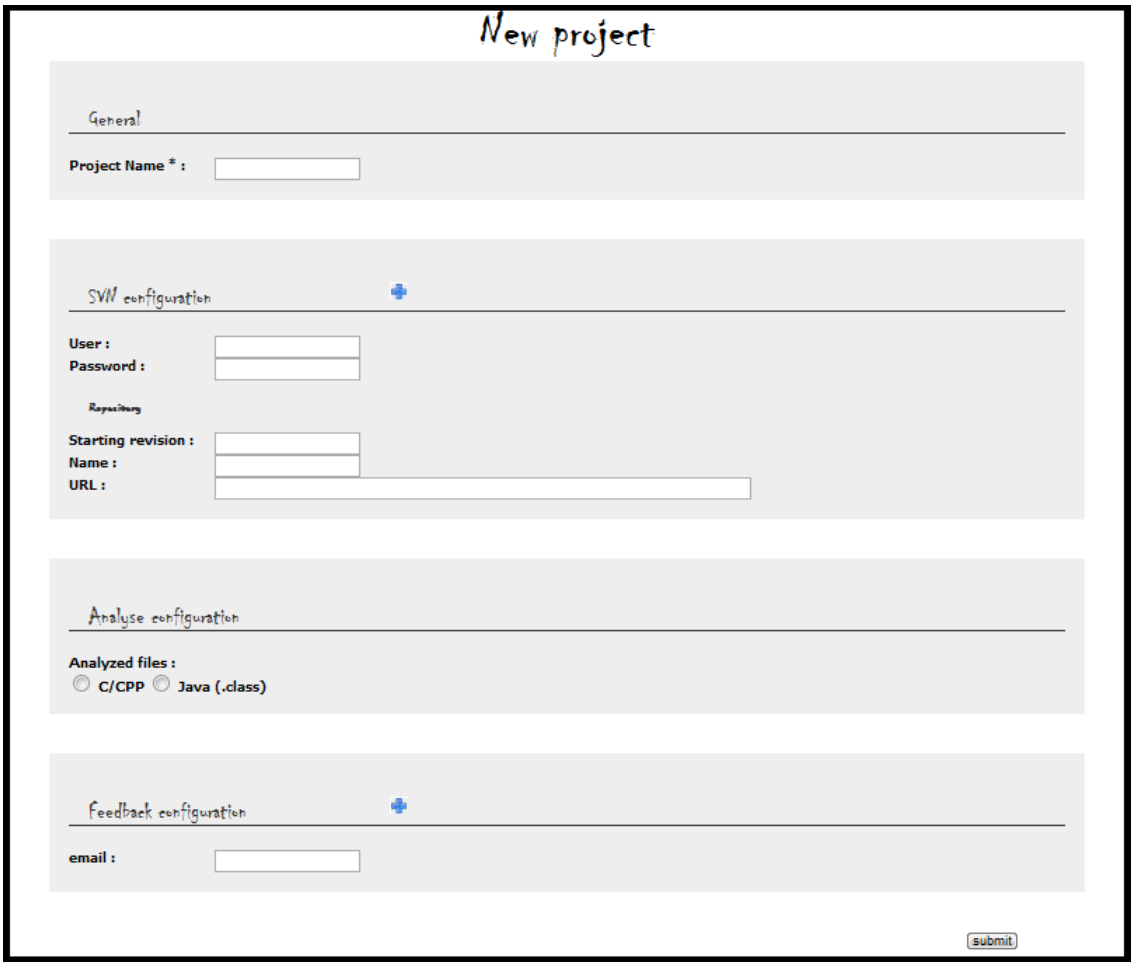


FIGURE 21 – Capture d'écran du formulaire d'ajout d'un nouveau projet

5 Test et performance

Expérience A présent nous allons essayer de montrer comment avec Squaner nous pouvons étudier l'impact des anti-patrons sur la qualité du logiciel. Pour l'expérience nous allons utiliser un module de Squaner possédant une trentaine de classes. Dans un premier temps nous avons lancé l'analyse une première fois sur notre module et observer les différents résultats. Dans un second temps, nous avons ajouté un anti-patron *"LongParametersList"* en ajoutant une fonction possédant une longue liste de paramètres volontairement dans notre module. Nous avons relancé par la suite l'analyse. La figure 5 nous montre les évaluations calculées par le modèle de qualité *QMOOD*. Nous pouvons observer que l'ajout de l'anti-patron a eu un impact positif très sensiblement sur deux propriétés *Efficacité* et *Fonctionnalité* surement dû à l'ajout d'une nouvelle fonction. Mais surtout un effet néfaste sur toutes les autres propriétés et surtout sur la *Compréhensibilité* et l'Extensibilité du logiciel. Comme nous pouvons le voir avec cette petite expérience, l'anti-patron a un impact plutôt négatif sur la qualité du code. Une des perspectives à court terme serait d'injecter à tour de rôle les anti-patrons les plus répandus pour essayer d'établir quels sont les anti-patrons les plus néfastes pour la qualité du code.

	Efficacité	Fonctionnalité	Flexibilité	Réutilisabilité	Extensibilité	Compréhensibilité
Avant injection	0.41	25.21	-10.4	45.65	-20.52	-51.35
Après injection	0.46	25.92	-14.57	43	-29.04	-57.97

FIGURE 22 – Résultat du modèle PQMOD

Performance JHotDraw est un logiciel qui a été conçu pour montrer les bonnes pratiques de conception de logiciel. Beaucoup de modèles de qualité se réfèrent à ce logiciel (et d'autres bien sûr) pour tester et valider leurs modèles. L'avantage de [5] *JHotDraw* est que c'est un logiciel de taille moyenne (environ 180 classes), qui implémente beaucoup de patrons de conceptions. Nous mettons 25 à 30 minutes pour l'analyse complète du logiciel. La performance a été une difficulté majeure rencontrée avec Squaner. La détection des patrons de conceptions utilisant des algorithmes de complexité élevée, la détection de tous les patrons de conception sont très coûteuses en temps. Nous avons donc mis en place une heuristique en place pour ne pas à avoir lancé la détection sur l'ensemble du projet si seulement quelques classes ont été modifiées sur le SVN. Cependant, nous ne pouvons pas échapper à cette complexité pour la première analyse du projet.

6 Conclusion et Perspective

Squaner est un système permettant le suivi de la qualité d'un logiciel tout au long de son cycle de vie. Basé sur le serveur SVN du logiciel à analyser, Squaner télécharge le code directement du serveur pour y effectuer l'évaluation de sa qualité grâce à ses modèles de qualité comprenant la détection des patrons et des anti-patrons de conception. Une fois les analyses effectuées, un mail est envoyé à l'utilisateur pour l'informer de la qualité du code nouvellement ajouté. Squaner possède aussi une interface web permettant de visualiser les résultats des différents modèles de qualité à tout moment.

Bien qu'il soit encore un projet jeune, Squaner offre des perspectives intéressantes pour l'étude sur l'évolution de la qualité logiciel. En effet, il est le premier système de monitorat permettant de suivre l'évolution des motifs de conceptions. Une perspective à court terme serait d'étudier l'impact des différents motifs de conception des logiciels. Nous savons que les patrons conception ont un impact sur la qualité du logiciel. Mais est-ce que tous les patrons sont bénéfiques ? Est-ce que l'association de plusieurs patrons peuvt être néfaste sur la qualité du code ? Sur quels propriétés de la qualité agit un patron précis ou un anti-patron ? Aujourd'hui, très peu d'études ont tenté de répondre à ces questions. Avec Squaner, une marche à suivre pour tenter d'y répondre serait d'injecter du bon ou du mauvais code volontairement, en ajoutant des anti-patrons ou patron de conceptions par exemple et voir comment réagissent les différents modèles de qualité. Nous pourrions alors voir plus précisément quelles propriétés de la qualité logicielle se dégradent ou s'améliore. Cela pourrait apporter une piste intéressante pour comprendre l'impact des motifs de conception. Pour réaliser une telle expérience, l'hypothèse forte serait que les modèles de qualité soient fiables. Avec ce même procédé, injection de code, nous pourrions ainsi utiliser Squaner pour valider les modèles de qualité en ajoutant du code plus ou moins flexible par exemple manuellement et observer comment la propriété flexibilité est évaluée après injection.

Nous voulons aussi améliorer l'interface web en la sécurisant pour permettre aux utilisateurs de pouvoir prendre la décision s'ils veulent partager leurs résultats ou pas. Mais nous voulons surtout créer un échange entre les utilisateurs et nous. Aujourd'hui, de nouveaux algorithmes de détection des motifs de conception tentent d'ajouter un cycle d'apprentissage aux algorithmes existants. Nous voulons dans l'avenir laisser la possibilité à l'utilisateur de pouvoir valider ou pas une détection de motifs fournis par Squaner. Ces informations pourraient servir pour l'apprentissage et améliorer ainsi la précision des algorithmes de détections.

D'un point de vue personnel j'ai trouvé le stage très enrichissant. L'ayant effectué à l'école Polytechnique de Montréal, cela m'a permis de découvrir une nouvelle

culture et de m'ouvrir l'esprit notamment au niveau de la recherche. Ayant intégré une équipe de recherche composée majoritairement de doctorants, j'ai pu voir de près ce que représentait une thèse et surtout m'ouvrir à un nouveau domaine que je connaissais peu qui est la qualité logicielle.

7 Remerciement

Je tiens à remercier mes tuteurs de stage Yann-Gaël Guéhéneuc et Giuliano Antoniol pour m'avoir permis d'effectuer mon stage dans leur école et d'y avoir passé six mois inoubliables. Je remercie également mes partenaires des équipes Ptidej et SOCCER qui m'ont intégré facilement au sein de leur groupe, Marianne Huchard et Chouki Tibermacine pour tout ce qu'ils m'ont apporté depuis le début du MASTER et Olivier Bendavid avec qui j'ai partagé cette expérience. Et enfin un grand remerciement à Foutse Kohmh pour tous ses conseils, à ma famille et ma copine Marie CILIA.

Références

[1]

[2] Srcml website. http://www.sdml.info/projects/srcml/.

[3] Statistique descriptive Wiki. http://fr.wikipedia.org/wiki/Statistique_descriptive.

[4] Grass project. http://grass.itc.it/.

[5] JHotDraw Web Site. http ://www.jhotdraw.org/.

[6] Réseau bayésien. http ://fr.wikipedia.org/wiki/R

[7] Squaner Web Site. http ://squaner.khomh.net/.

[8] Tomcat web site. http://tomcat.apache.org/.

[9] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13) :755–765, 2002.

[10] J. Bansiya and CG Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, pages 4–17, 2002.

[11] J.M. Bieman, R. Alexander, P.W. Munger III, and E. Meunier. Software design quality : Style and substance. In *ICSE 2002 Workshop on Software Quality, 2002*. Citeseer.

[12] JM Bieman, G. Straw, H. Wang, PW Munger, and RT Alexander. Design patterns and change proneness : An examination of five evolving systems. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 40–49, 2003.

[13] BW Boehm, JR Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, page 605. IEEE Computer Society Press, 1976.

[14] Salah Bouktif, Giuliano Antoniol, and Ettore Merlo. A feedback based quality assessment to support open source software evolution : the grass case study.

[15] K. Brown. Design reverse-engineering and automated design-pattern detection in Smalltalk. 1996.

[16] W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray, J. Wiley, and I. Sons. Refactoring Software, Architectures, and Projects in Crisis, 1998.

[17] W.J. Brown, H.W. McCormick, T.J. Mowbray, and R.C. Malveau. *AntiPatterns : refactoring software, architectures, and projects in crisis*. Wiley Chichester, 1998.

[18] S.R. Chidamber, C.F. Kemerer, and C. MIT. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6) :476–493, 1994.

[19] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2) :127–139, 2003.

[20] R.G. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2) :146–162, 1995.

[21] R.G. Dromey. Cornering the chimera. *IEEE SOFTWARE*, pages 33–43, 1996.

[22] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, M. Temmerman, and B. Antwerpen. Does god class decomposition affect comprehensibility ? *SE*, pages 346–355, 2006.

[23] L. Erlikh and R. Technol. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3) :17–23, 2000.

[24] M. Fowler and K. Beck. *Refactoring : improving the design of existing code*. Addison-Wesley Professional, 1999.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[26] M.W. Godfrey and Q. Tu. Evolution in open source software : A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142. Citeseer, 2000.

[27] Y.G. Guéhéneuc. A meta-model (and parsers) to represent and to manipulate object-oriented programs and design motifs. PADL, since July 1999.

[28] Y.G. Guéhéneuc. Ptidej : Promoting patterns with patterns. In *European Conference on Object Oriented Programming, workshop on Building a System with Patterns, Glasgow, Scotland*. Citeseer, 2005.

[29] Y.G. Guéhéneuc and G. Antoniol. DeMIMA : A multi-layered framework for design pattern identification. *Transactions on Software Engineering*, 34(5) :667–684, 2008.

[30] Y.G. Guéhéneuc and G. Antoniol. DeMIMA : A multi-layered framework for design pattern identification. *Transactions on Software Engineering*, 34(5) :667–684, 2008.

[31] B. Henderson-Sellers. Object-Oriented Metrics : Measures of Complexity. Prentice Hall, 1st edition, December 1995.

[32] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, volume 50, pages 75–76, 1995.

[33] O. Kaczor, Y.G. Gueheneuc, and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. 2006.

[34] CF Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4) :493–509, 1999.

[35] F. Khomh and Y.G. Guéhéneuc. DEQUALITE : building design-based software quality models. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, pages 1–7. ACM, 2008.

[36] Foutse Kohmh. *Patron et qualité des programmes orienté objet*. PhD thesis, Faculté des arts et de la science,Université de Montréal, Avril 2010.

[37] M. Lehman. Laws of software evolution revisited. *Software process technology*, pages 108–124.

[38] B.P. Lientz and E.B. Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11) :769, 1981.

[39] M. Lorenz and J. Kidd. Object-Oriented Metrics : Measures of Complexity. Prentice-Hall, 1st edition, July 1994.

[40] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.

[41] T.J. McCabe and C.W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12) :1425, 1989.

[42] A.M. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D.C. Schmidt, and B. Natarajan. Distributed Continuous Quality Assurance.

[43] A. Mockus, R.T. Fielding, and J.D. Herbsleb. Two case studies of open source software development : Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3) :346, 2002.

[44] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur. DECOR : A method for the specification and detection of code and design smells.

[45] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur. DECOR : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1) :20–36, 2010.

[46] N. Moha, Y.G. Guéhéneuc, A.F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. *Fundamental Approaches to Software Engineering*, pages 276–291, 2008.

[47] oumoumsack. La qualité logiciel. http ://lil.univ-littoral.fr/ oumoumsack.

[48] D.L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[49] A. Porter, C. Yilmaz, A.M. Memon, A.S. Krishna, D.C. Schmidt, and A. Gokhale. Techniques and processes for improving the quality and performance of open-source software. *Software Process Improvement and Practice*, 11(2) :163–176, 2006.

[50] G. Robles, S. Koch, and J.M. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS), Edinburg, Scotland, UK*, pages 51–55. Citeseer, 2004.

[51] S.R. Schach, B. Jin, D.R. Wright, G.Z. Heller, and A.J. Offutt. Maintainability of the Linux kernel. *IEE Proceedings-Software*, 149(1) :18–23, 2002.

[52] D.P. Tegarden, S.D. Sheetz, and D.E. Monarchi. A software complexity model of object-oriented systems. *Decision Support Systems*, 13(3-4) :241–262, 1995.

[53] M. Vok. Defect Frequency and Design Patterns : An Empirical Study of Industrial Code (HTML). *IEEE Transactions on Software Engineering*, 30 :12.

[54] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. 2007.

A Article associé à Squaner

Article accepté à la conférence **ICSM** (International Conference on Software Maintenance) pour une session sur les outils de maintenance en Juillet 2010.

SQUANER: A Framework for Monitoring the Quality of Software Systems

Nicolas Haderer
LIRMM, Université Montpellier 2, France
Email:nicolas.haderer@etud.univ-montp2.fr

Foutse Khomh
PTIDEJ Team, DIRO, Université de Montréal, QC, Canada
Email: foutsekh@iro.umontreal.ca

Giuliano Antoniol
SOCCER Lab., DGIGL, École Polytechnique de Montréal, QC, Canada
Email: giuliano.antoniol@polymtl.ca

***Abstract*—Despite the large number of quality models and publicly available quality assessment tools like PMD, Checkstyle, or FindBugs, very few studies have investigated the use of quality models by developers in their daily activities. One reason for this lack of studies is the absence of integrated environments for monitoring the evolution of software quality. We propose SQUANER (Software QUality ANalyzER), a framework for monitoring the evolution of the quality of object-oriented systems. SQUANER connects directly to the SVN of a system, extracts the source code, and perform quality evaluations and faults predictions every time a commit is made by a developer. After quality analysis, a feedback is provided to developers with instructions on how to improve their code.**

I. Introduction

Large object-oriented software systems are now pervasive in our society. They play a vital role in our everyday life and are increasingly more and more complex. Their quality is thus of major importance. Moreover, the current trend in outsourcing development and maintenance requires means to measure quality with great details and monitor its evolution. Because the code is geographically distributed and developers generally have a limited knowledge of the entire system under development, changes on parts of the code sometimes invalidate the original design of the system, causing the design to degrade and therefore negatively impacting the overall quality of the system.

Thus, being able to monitor the quality of the system and provide quick feedback to developers working on parts of a system is essential to control and improve the quality of the system. Many pioneers of software engineering like Pfleeger recommend that more time be spent to communicate the big picture of systems under development to everyone in every position: "The people working on the pieces need to know how their one piece fits into the entire architecture" [12]. Early detection of defects in code is very important to reduce software development costs. Megen and Meyerhoff [20] observed that the costs of fixing defects in systems grow dramatically when defects are found late.

In this paper, we propose SQUANER (Software QUality ANalyzER), a framework for monitoring the quality of object-oriented systems. The contribution of SQUANER comparatively to other quality assessment tools like Squale and Sonar is it continuous evaluation of the systems under development, and its non-dependance on specific technologies, like Maven: SQUANER connects directly to the SVN of a system, extracts the source code, perform design patterns, antipatterns and code smells detections, perform quality evaluations and faults predictions every time a commit is made by a developer. After quality analysis, a feedback is provided to developers with instructions on how to improve their code. Contrary to Sonar, SQUANER is not dependant on Maven, and can be extended to analyze Maven projects as well.

With its focus on early defects detection and quality evaluation, SQUANER will help reduce a substantial amount of systems' budgets and save development time [20]. In a distributed development scenario, SQUANER provides to all developers working on parts of a system, a feedback on the overall quality of the system; thus increasing their knowledge of the system and enabling an effective quality control.

With SQUANER, the research community will benefit a new tool for evolution studies. Its continuous quality assessment, design patterns and design smells detections offer the possibility to perform interesting studies such as: the analysis of system decay, the effectiveness of quality analysis tools, or their impact on developers behavior and software development practices.

II. SQUANER

SQUANER is a framework built in layers. Figure 1 gives an overview of SQUANER's architecture and Figure 2 describes the components of SQUANER analyzer.

Once a commit is made in the SVN of a project, the source code is checked out and the parser **PADL creator** is used to build a PADL model of the system. **PADL creator** is based on the meta-model PADL [7] and provide parsers for C♯, C++, and Java. From this model of the system, metrics values are computed with the component **POM** by applying each metric on each class of the model. **POM** is a framework that offers more than 60 different metrics from the literature, including class-method import and export coupling [2]; Coupling Between Objects (CBO), and Weighted Method Count (WMC) [4]; Lack of Cohesion in Methods (LCOM5) [10]; 'C' connectivity of a class [11]; numbers

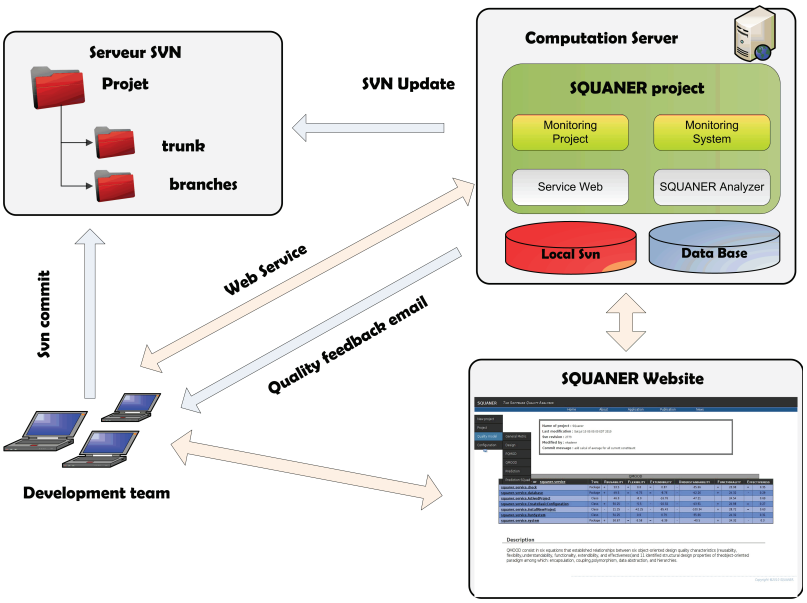


Fig. 1. Architecture of SQUANER.

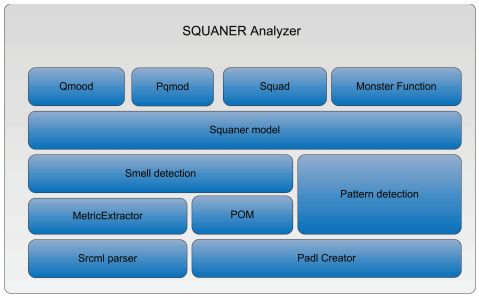


Fig. 2. SQUANER Analyzer.

of new, inherited, and overridden methods and total number of methods [14]; Cyclomatic Complexity Metric (CC) [15]; numbers of hierarchical levels below a class and class-to-leaf depth [19].

The component **Pattern detection** is responsible for design patterns detection. This component is based on PTIDEJ [8] which implements the detection approach DeMIMA by Guéhéneuc and Antoniol [9] and the approaches by Kaczor *et al.* [18] and Ng *et al.* [16]. Combining these approaches, PTIDEJ is able to detect 13 design motifs, namely: Abstract Factory, Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method, and Visitor.

The component **Smell detection** is responsible for antipatterns and code smells detections. This component implements the method DECOR (Defect dEtection for CORrection) by Moha *et al.* [17], to specify and identify antipatterns and code smells. DECOR is a detection approach based on a thorough domain analysis of code smells and antipatterns in the literature; analysis on which is based a domain-specific language. DECOR is able to detect the following antipatterns from [3], [5]: AntiSingleton, Blob, ClassDataShouldBePrivate (CDSBP), ComplexClass, LargeClass, LazyClass, LongMethod, LongParameterList (LPL), MessageChains, RefusedParentBequest (RPB), SpaghettiCode, SpeculativeGenerality (SG), SwissArmyKnife. These antipatterns are representative of design and implementation problems with data, complexity, size, and the features provided by a class. In addition to these antipatterns, DECOR also detects the following code smells: AbstractClass, ChildClass, ClassGlobalVariable, ClassOneMethod, ComplexClassOnly, ControllerClass, DataClass, FewMethods, FieldPrivate, FieldPublic, FunctionClass, HasChildren, LargeClass, LargeClassOnly, LongMethod, LongParameterListClass, LowCohesionOnly, ManyAttributes, MessageChainsClass, MethodNoParameter, MultipleInterface, NoInheritance, NoPolymorphism, NotAbstract, NotComplex, OneChildClass, ParentClassProvidesProtected, RareOverriding, TwoInheritance.

The component **Squaner model** implements a meta-model taking into account the history of systems as an explicit entity similarly to Hismo [6]. This meta-model builds on top of our existing meta-model PADL which describes structural information. The Squaner model can also analyse Maven projects and therefore is able to integrate and extend Sonar modules.

The component **Qmood** implements the quality model QMOOD by Bansiya and Davis [1]. QMOOD consists in six equations that established relationships between six object-oriented design quality characteristics (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) and 11 identified structural design properties of the object-oriented paradigm among which: encapsulation,

coupling, polymorphism, data abstraction, and hierarchies. QMOOD is one of the most used and most referenced models in recent studies.

The component **Pqmod** implements the quality model PQMOD by Khomh and Guéhéneuc [13]. This quality model is composed of seven rules for the evaluation of the quality of systems by taking into account their design. These rules are defined for expandability, generality, modularity, modularity at runtime, understandability, reusability, scalability.

The component **Squad** implements two BBNs models that allow the prediction of faulty classes in systems by taking into account their metric values, their design smells and design patterns.

The component **Monitoring project** contains project monitors that are responsible for the scheduling of all the tasks involved in the analysis of systems, while the component **Monitoring system** schedules the executions of projects monitors.

III. Usage Scenario

After each developer contribution in the SVN repository, SQUANER detects the new changes automatically and downloads the new code. A series of analysis of this source code is then performed through the various components described in Section II. These analysis are performed following a configuration set beforehand by developers or quality managers. The configuration consists in choosing metrics, quality models, design patterns and design defects of interest and the frequency of their analysis.

Following the analysis, an email is sent to the development team summarizing the results, highlighting the more risky classes and providing advices on how to improve the code. With these emails, developers can have an overview of the evolution of the quality of the system since the last revision. Moreover, all the development team is aware of the impact of the new contribution on the overall quality of the system. Figure 3 presents the structure of the emails.

All the analysis results are stored in the SQUANER database to enable monitoring the evolution of the quality. SQUANER provides two means to access these information in the database: First, via a web interface at http://www.ptidej.net/research/squaner/; developers and quality analysts can browse through the packages and classes of a system and visualize detailed data on different quality characteristics. Figure 4 illustrates the visualization of results through the web site of SQUANER. Second, developers and quality analysts can use web services to schedule regular automatic quality evaluations or access the information on the quality of systems stored in the database. They can also use web services to suggest or comment on some evolution actions as a feedback to other developers; for example, when they identify problems that they cannot resolve because they do not fit their skills and knowledge.

```
Hello nhaderer,

here's a report on your recent commit change for the revision : 2773

****************************************************************
File :squaner.service.InstallNewProject.java
****************************************************************

Class
+--------+--------+--------+--------+--------+--------+
          | DIT | SIX | WMC | McCabe | CBO | LCOM5 |
+----------+--------+--------+--------+--------+----|
|NEW Metrics | 1   | 0   | 56  | 25     | 22     | 0  |
+----------+--------+--------+--------+--------+----|
|OLD Metrics | 1   | 0   | 53  | 24     | 22     |0.3 |
+----------+--------+--------+--------+--------+----+

Design analysis :
This class contains 2 smells

+--------+--------+--------+--------+--------+--------+
Smell: ComplexClass
+--------+--------+--------+--------+--------+--------+
ComplexClassOnly-0 = squaner.service.InstallNewProject
McCabe-0 = 25.0
McCabe_MaxBound-0 = {McCabe_MaxBound =40.0, McCabe_UpperQuartile= 20.0}
+--------+--------+--------+--------+--------+--------+
Advice : consider restructuring this class
+--------+--------+--------+--------+--------+--------+

+--------+--------+--------+--------+--------+--------+
Smell : LongMethod
+--------+--------+--------+--------+--------+--------+
LongMethodClass-0 = squaner.service.InstallNewProject
MethodName-0 = installNewProject()
LOC-0 = 245.0
LOC_MaxBound-0 = {METHOD_LOC_MaxBound=401.5, METHOD_LOC_UpperQuartile=178.0}
+--------+--------+--------+--------+--------+--------+
Advice : consider restructiring method
+--------+--------+--------+--------+--------+--------+

Please visite the web site of Squaner project for more details :
http://www.squaner.khomh.net

Brought to you by :
SQUANER The Software QUality AnalyzER
```

Fig. 3. Example of generated quality feedback email (The e-mail was edited to fit the space)

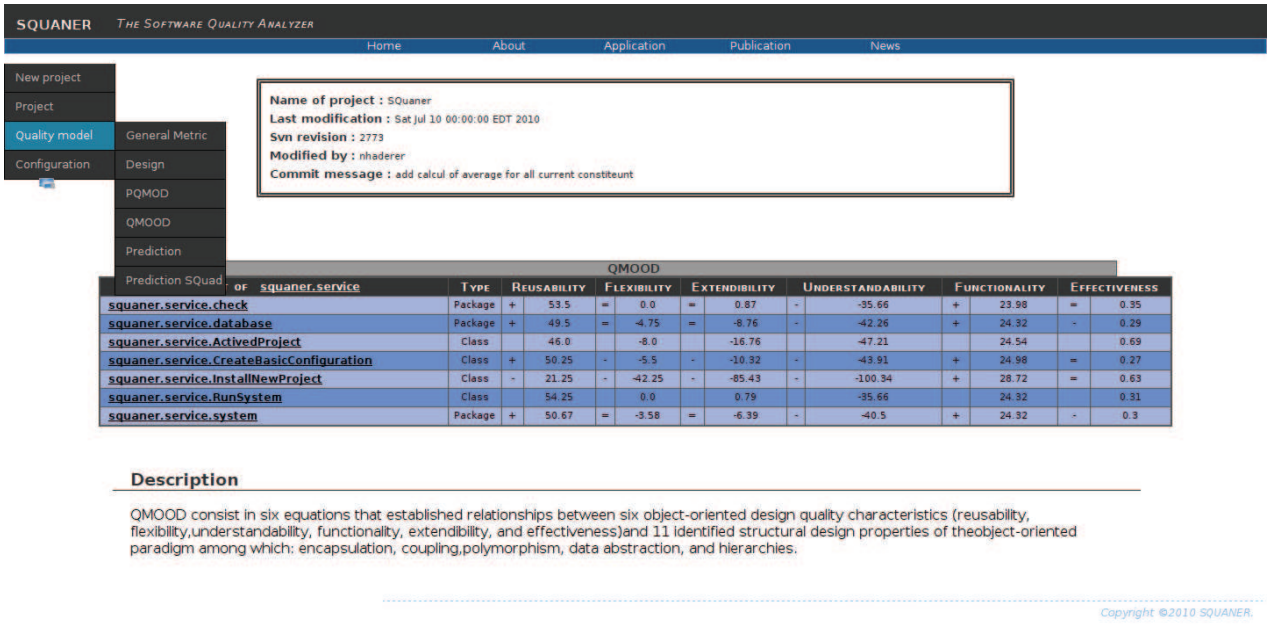
IV. Related Work

Some quality assessment tools have been proposed in the open source community, among these, Sonar[1], and Squale [2] are the most mature projects. Sonar uses various static code analysis tools such as Checkstyle, PMD, FindBugs, and Clover to extract software metrics for quality assessments and statistical visualization tools to report information on the quality of the systems. Squale implements existing standard quality models, such as ISO/IEC 9126, McCall and use a weather metaphor to report the quality of the systems. Although these tools can be apply on systems to evaluate their quality, they do not provide means to perform a continuous quality evaluation. Moreover, contrary to SQUANER which is nonintrusive as it accesses the source code directly from SVN repositories, Sonar and Squale are tools which need some training from developers and quality managers to be fully effective.

V. Conclusion

In this paper we presented SQUANER, a framework for monitoring the quality of object-oriented systems. SQUANER connects directly to the SVN of a system, extracts the source code, perform design patterns, antipatterns and code smells detections, perform quality evaluations and faults predictions

[1]http://www.sonarsource.org/
[2]http://www.squale.org/



OF squaner.service	Type	Reusability		Flexibility		Extendibility		Understandability		Functionality		Effectiveness	
squaner.service.check	Package	+	53.5	=	0.0	=	0.87	-	-35.66	+	23.98	=	0.35
squaner.service.database	Package	+	49.5	=	-4.75	=	-8.76	-	-42.26	+	24.32	-	0.29
squaner.service.ActivedProject	Class		46.0		-8.0		-16.76		-47.21		24.54		0.69
squaner.service.CreateBasicConfiguration	Class	+	50.25	-	-5.5	-	-10.32	-	-43.91	+	24.98	=	0.27
squaner.service.InstallNewProject	Class	-	21.25	-	-42.25	-	-85.43	-	-100.34	+	28.72	=	0.63
squaner.service.RunSystem	Class		54.25		0.0		0.79		-35.66		24.32		0.31
squaner.service.system	Package	+	50.67	=	-3.58	=	-6.39	-	-40.5	+	24.32	-	0.3

Fig. 4. Visualization of results through the web

every time a commit is made by a developer. After quality analysis, a feedback is provided to developers with instructions on how to improve their code. With SQUANER, developers and quality analysts can also use web services to schedule regular automatic quality evaluations, to suggest or comment on some evolution actions as a feedback to other developers, or to access information on the quality of systems stored in a database. SQUANER is freely available online at the following location: http://www.ptidej.net/research/squaner/.

Acknowledgements. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

References

[1] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. on Software Engineering*, 28:4–17, Jan. 2002.

[2] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In W. R. Adrion, editor, *Proceedings of the* 19^{th} *International Conference on Software Engineering*, pages 412–421, ACM Press, May 1997.

[3] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, March 1998.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management, December 1993.

[5] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999.

[6] T. Grba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2005.

[7] Y.-G. Guéhéneuc. PADL, since July 1999. A meta-model (and parsers) to represent and to manipulate object-oriented programs and design motifs.

[8] Y.-G. Guéhéneuc. PTIDEJ, since July 2001. A tool suite to evaluate and to enhance the quality of object-oriented programs.

[9] Y.-G. Guéhéneuc and G. Antoniol. DeMIMA: A multi-layered framework for design pattern identification. *Transactions on Software Engineering (TSE)*, 34(5):667–684, September 2008. 18 pages.

[10] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1^{st} edition, December 1995.

[11] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the* 3^{rd} *International Symposium on Applied Corporate Computing*, pages 25–27. Texas A & M University, October 1995.

[12] C. Jones. *Patterns of Software System Failure and Success*. Computer Press, Boston, Massachusetts, 1996.

[13] F. Khomh and Y.-G. Guéhéneuc. DEQUALITE: Building design-based software quality models. In *Proceedings of the 2nd PLoP Workshop on Software Patterns and Quality (SPAQu)*, october 2008.

[14] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1^{st} edition, July 1994.

[15] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[16] Janice Ka-Yee Ng, Y.-G. Guéhéneuc, and G. Antoniol. Identification of behavioral and creational design motifs through dynamic analysis. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, November 2009. 30 pages.

[17] Naouel Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *Transactions on Software Engineering (TSE)*, 2009. 16 pages.

[18] Olivier Kaczor, Y.-G. Guéhéneuc, and S. Hamel. Identification of design motifs with pattern matching algorithms. *Information and Software Technology (IST)*, August 2009. 46 pages.

[19] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi. A software complexity model of object-oriented systems. *Decision Support Systems*, 13(3–4):241–262, March 1995.

[20] R. van Megen and D. B. Meyerhoff. Costs and benefits of early defect detection: experiences from developing client server and host applications. *Software Quality Journal*, 4(4):247–256, 1995.

B Définition des métriques disponible dans POM

ACAIC : ancestor Class-Attribute Import Coupling.

ACMIC : ancestors Class-Method Import Coupling.

AID : average Inheritance Depth of an entity.

ANA : count the average number of classes from which a class inherits informations.

CAM : computes the relatedness among methods of the class based upon the parameter list of the methods.

CBOin : coupling Between Objects of one entity.

CBOout : coupling Between Objects of one entity.

CIS : counts the number of public methods in a class.

CLD : class to Leaf Depth of an entity.

cohesionAttributes : returns the degree of cohesion between methods and attributes of a class.

connectivity : returns the degree of connectivity of an entity in a system.

CP : the number of packages that depend on the package containing entity.

DAM : returns the ratio of the number of private (protected) Attributes to the total number of Attributes declared in a class.

DCAEC : returns the DCAEC (Descendants Class-Attribute Export Coupling) of one entity.

DCC : returns the number of classes a class is directly related to (by attribute declarations and message passing.

DCMEC : returns the DCMEC (Descendants Class-Method Export Coupling) of one entity.

DIT : returns the DIT (Depth of inheritance tree) of an entity.

DSC : count of the total number of classes in the design.

EIC : the number of inheritance relationships in which superclasses are in external packages.

EIP : the number of inheritance relationships where the superclass is in the package containing entity and the subclass is in another package.

ICHClass : compute the complexity of an entity as the sum of the complexities of its declared and inherited methods.

LCOM1 : returns the LCOM (Lack of COhesion in Methods) of an entity.

LCOM2 : returns the LCOM (Lack of COhesion in Methods) of an entity.

LOC : returns the number of line of code of an entity.

MFA : the ratio of the number of methods inherited by a class to the number of methods accessible by member methods of the class.

MOA : count the number of data declarations whose types are user defined classes.

NAD : number of attributes declared.

NADExtended : number of attributes declared in a class and in its member classes.

NCM : returns the NCM (Number of Changed Methods) of an entity.

NCP : the number of classes package containing entity.

NMA : returns the NMA (Number of New Methods) of an entity.

NMD : number of methods declared.

NMDExtended : number of methods declared in the class and in its member classes.

NMI : returns the NMI (Number of Methods Inherited) of an entity.

NMO : returns the NMO (Number of Methods Overridden) of an entity.

NOA : returns the NOA (Number Of Ancestors) of an entity.

NOC : returns the NOC (Number Of Children) of an entity.

NOD : returns the NOD (Number Of Descendents) of an entity.

NOH : count the number of class hierarchies in the design.

NOM : counts all methods defined in a class.

NOP : returns the NOP (Number Of Parents) of an entity.

NOParam : compute the average number of parameters of methods.

NOPM : count of the Methods that can exhibit polymorphic behavior.

PIIR : the number of inheritance relationships existing between classes in the package containing entity.

PP : the number of provider packages of the package containing entity.

REIP : EIP divided by the sum of PIIR and EIP.

RFP : the number of class references from classes belonging to other packages to classes belonging to the package containing entity.

RPII : PIIR divided by the sum of PIIR and EIP.

RRFP : RFP divided by the sum of RFP and the number of internal class references.

RRTP : RTP divided by the sum of RTP and the number of internal class references.

RTP : the number of class references from classes in the package containing entity to classes in other packages.

SIX : returns the SIX (Specialisation IndeX) of an entity.

WMC1 : computes the weight of an entity considering the complexity of a method to be unity.

McCabe : number of points of decision + 1.

CBO : coupling Between Objects of one entity.

LCOM5 : returns the LCOM (Lack of COhesion in Methods) of an entity.

WMC : computes the weight of an entity by computing the number of method invocations in each method.

PageRank : measures the relative importance of a class in the overall structure of relations among classes.

C Définition des Patrons de conception détectés avec Squaner

AbstractClass : this code smell is characteristic of the Speculative Generality Antipattern. This odor exists when we have generic or abstract code that isn't actually needed today. Such code often exists to support future behavior, which may or may not be necessary in the future.

ChildClass : this code smell occurs when the number of methods declared in a class and the number of it's declared attributes is very high. It is a symptom of poor object decomposition. The public interface of the class differing greatly from the one of its super-class. This code smell characterises the Tradition Breaker antippatern.

ClassGlobalVariable : this code smell occurs when a class declares public class variable that are used as "global variable" in procedural programming.

ClassOneMethod : this code smell occurs when a class has only one method.

ComplexClassOnly : this code smell is present when a class both declares many fields and methods and which methods realise complex treatments, using many if and switch instructions. Such a class is probably providing lots of services while being difficult to maintain and fragile due to its complexity.

ControllerClass : this odor is present when a class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes.

DataClass : this code smell is present when a class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

FewMethod : this code smell characterise Lazy classes that declare few methods.

FieldPrivate : this code smell is present when many private fields are declared. It's generally symptomatic of the Functional Decomposition antipattern.

FieldPublic : this code smell is symptomatic of the Class Data Should Be Private antippatern. It occurs when the data encapsulated by a class is public, thus allowing client classes to change this data without the knowledge of the declaring class.

FunctionClass : this code smell occurs when we have a main class, i.e., a class with a procedural name, such as Compute or Display. It is symptomatic of the Functional Decomposition antipattern.

HasChildren : this code smell describes classes with many children.

LargeClass : this odor concerns classes that are trying to do too much. These classes do not follow the good practice of divide-and-conquer which consists of decomposing a complex problem into smaller problems. These classes also have low cohesion.

LargeClassOnly : this code smell concerns classes with a very high number of attributes and/or methods defined.

LongMethod : this odor is a method with a high number of lines of code. A lot of variables and parameters are used. Generally, this kind of method does more than its name suggests it.

LongParameterListClass : this odor corresponds to a method with high number of parameters. This smell occurs when the method has more than four parameters. Long lists of parameters in a method, though common in procedural code, are difficult to understand and likely to be volatile.

LowCohesionOnly : this code smell characterises the lack of cohesion in a class.

ManyAttributes : this code smell occurs when the number of attributes declared in a class is too high.

MessageChainsClass : this code smell is present when you see a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects.

MethodNoParameter : this code smell occurs when a class declares methods with no parameter.

MultipleInterface : this code smell occurs when a class implements a high number of interfaces. It is generally symptomatic of the Swiss Army Knife antipattern.

NoInheritance : this odor is present when inheritance is scarcely used.

NoPolymorphism : this odor is present when polymorphism is scarcely used.

NotAbstract : this odor occurs when a developer haven't yet seen how a higher-level abstraction can clarify or simplify his code.

NotClassGlobalVariable : this odor manifest itself in the anipattern Anti-Singleton when a class declares public class variable that are used as "global variable" in procedural programming. It reveals procedural thinking in object-oriented programming and may increase the difficulty to maintain the program.

NotComplex : this code smell characterises classes performing "atomic" functionality, with little complexity.

OneChildClass : this code smell occurs when a class does not have child class.

ParentClassProvidesProtected : this code smell occurs when a subclass does not use attributes and/or methods protected inherited by a parent.

RareOverriding : this code smell occurs when a class rarely overrides inherited attributes and/or methods.

TwoInheritance : this odor characterises a hierarchy with a depth greater than two.

D Définition des Anti-patrons de conception détectés par Squaner

Anti-Singleton : it is a class that declares public class variable that are used as "global variable" in procedural programming. It reveals procedural thinking in object-oriented programming and may increase the difficulty to maintain the program.

Blob : (called also God class [**?**]) corresponds to a large controller class that depends on data stored in surrounded data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes [**?**].

Class Data Should Be Private : it occurs when the data encapsulated by a class is public, thus allowing client classes to change this data without the knowledge of the declaring class.

Complex Class : it is a class that both declares many fields and methods and which methods realise complex treatments, using many if and switch instructions. Such a class is probably providing lots of services while being difficult to maintain and fragile due to its complexity.

Large Class : it is a class with too many responsibilities. This kind of class declares a high number of usually unrelated methods and attributes.

Lazy Class : it is a class that does not do enough. The few methods declared by this class have a low complexity.

Long Method : it is a method with a high number of lines of code. A lot of variables and parameters are used.Generally, this kind of method does more than its name suggests it.

Long Parameter List : it corresponds to a method with high number of parameters. This smell occurs when the method has more than four parameters.

MessageChains : it Occurs when you have a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects [**?**].

Speculative Generality : it is an abstract class without child classes. It was added in the system for future uses and this entity pollutes the system unnecessarily.

Swiss Army Knife : it refers to a tool fulfilling a wide range of needs. The Swiss Army Knife design smell is a complex class that offers a high number of services, for example, a complex class implementing a high number of interfaces. A Swiss Army Knife is different from a Blob, because it exposes a high complexity to address all foreseeable needs of a part of a system, whereas the Blob is a singleton monopolising all processing and data of a system. Thus, several Swiss Army Knives may exist in a system, for example utility classes.

The Refused Parent Bequest : it appears when a subclass does not use attributes and/or methods public and/or protected inherited by a parent. Typically, this means that the class hierarchy is wrong or badly organized.

The Spaghetti Code : it is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables for processing. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, polymorphism and inheritance.

Table des figures