ETS-RT - 2009-000

ÉVALUATION DE LA MAINTENABILITÉ DE S3MDSS AVEC L'OUTIL SONAR

HASSENE LAARIBI

ETS-RT - 2009-000

ÉVALUATION DE LA MAINTENABILITÉ DE S3MDSS AVEC L'OUTIL SONAR

RAPPORT TECHNIQUE DE L'ÉTS

HASSENE LAARIBI

Département du Génie Logiciel

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MONTRÉAL, 02 AVRIL 2009

© École de technologie supérieure 2009

La citation d'extraits ou la reproduction de courtes sections est permise à la condition de mentionner le nom de l'auteur et la source. Toute reproduction de parties quantitativement ou qualitativement importantes requiert l'autorisation du titulaire des droits d'auteur.

ISBN x-xxxxxx-xx-x

Dépôt légal : Bibliothèque et Archives nationales du Québec, 2009
Dépôt légal : Bibliothèque et Archives Canada, 2009

EVALUATION OF THE MAINTAINABILITY OF S3MDSS USING SONAR UTILITY TOOL

HASSENE LAARIBI

ABSTRACT

Several commercial and open source tools allow the evaluation of software quality, using measurements on the source code.

Thos document describes the evaluation of the quality of S3MDSS application, by using the open source code analyzer “SONAR”, in order to detect problems and to submit recommendations that will help improve code quality.

ÉVALUATION DE LA MAINTENABILITÉ DE S3MDSS AVEC L'OUTIL SONAR

HASSENE LAARIBI

RÉSUMÉ

Plusieurs outils commerciaux et open source permettent l'évaluation de la qualité du logiciel en effectuant des mesures sur le code source.

Le présent document se base sur l'outil d'analyse de code Source « SONAR » dans le but d'évaluer la qualité de l'application S3MDSS, de détecter les problèmes et d'émettre des recommandations pour l'amélioration du code.

TABLE DES MATIÈRES

ABSTRACT ... III

RÉSUMÉ ... IV

TABLE DES MATIÈRES ... V

LISTE DES TABLEAUX ... VII

LISTE DES FIGURES ... VIII

1 INTRODUCTION ... 1

1.1 Objectif ... 1

1.2 Approche ... 1

2 L'OUTIL SONAR ... 2

2.1 Principe ... 2
2.1.1 Mesures effectuées par SONAR ... 2
2.1.2 Règles de codage SONAR ... 3

2.2 Correspondances SONAR et ISO 9126 ... 6

3 EXÉCUTION DE SONAR ... 8

3.1 Configuration SONAR ... 8

3.2 Opération SONAR ... 8

4 ANALYSE DES RÉSULTATS ... 9

4.1 Mesure des duplications de code ... 9
4.1.1 Résultats de la mesure ... 9
4.1.2 Interprétation des résultats ... 10
4.1.3 Recommandations ... 10

4.2 Mesure de la complexité par classe ... 10
4.2.1 Résultats de la mesure ... 10
4.2.2 Interprétation des résultats ... 12
4.2.3 Recommandations ... 13

4.3 Mesure de la complexité par méthode ... 14
4.3.1 Résultats de la mesure ... 14
4.3.2 Interprétation des résultats ... 14
4.3.3 Recommandations ... 14

4.4 Respect des règles de codage obligatoires 15
4.4.1 Résultats de la mesure 15
4.4.2 Interprétation des résultats 16
4.4.3 Recommandations 19
4.5 Respect des règles de codage optionnelles 20
4.5.1 Résultats de la mesure 20
4.5.2 Interprétation des résultats 22
4.5.3 Recommandations 22
4.6 Classes les plus critiques 23
5 COMPARAISON DES RÉSULTATS DE SONAR AVEC CEUX DE L'INSPECTION « VISUELLE » 24
6 CONCLUSION 25
6.1 L'application S3MDSS 25
6.2 L'outil SONAR 25
6.3 Pour une utilisation optimale de l'outil SONAR 26
BIBLIOGRAPHIE 27

LISTE DES TABLEAUX

Table 1 Définitions des caractéristiques de la qualité ISO 9126 7

Table 2 Résumé de la mesure de la duplication de code 9

Table 3 Duplication de code par classe.. 10

Table 4 Les classes ayant les plus grandes complexités par classe 11

Table 5 Les classes qui ont les méthodes les plus complexes 14

Table 6 Violations des règles par catégorie 16

Table 7 Classes ayant le plus de violations de règles de Fiabilité 17

Table 8 Classes pour lesquelles SONAR a remonté le plus de problèmes 23

LISTE DES FIGURES

Figure 1 SONAR, sommaire des mesures d'un projet 3

Figure 2 Caractéristiques de la qualité ISO 9126 6

Figure 3 Répartition de la complexité / classe 11

Figure 4 Résumé de la mesure de la conformité aux règles obligatoires 15

Figure 5 Exemple d'assignation de valeur à un paramètre de méthode 19

Figure 6 Résumé de la mesure de la conformité aux règles optionnelles 21

1 INTRODUCTION

Plusieurs outils commerciaux et open source permettent l'évaluation de la qualité du logiciel en effectuant des mesures sur le code source.

Le présent document se base sur l'analyseur de code source SONAR pour évaluer la qualité du code de l'application S3MDSS.

1.1 Objectif

L'objectif est d'analyser la maintenabilité d'un logiciel à l'aide d'un outil d'évaluation de la qualité. L'analyse doit être reproductible, impartiale et objective.

1.2 Approche

Dans ce document, la maintenabilité regroupera tous les aspects qui peuvent nécessiter ou impacter tout type d'activité de maintenance du logiciel.

La maintenabilité ne sera donc pas limitée à la définition plus restreinte donnée par le standard ISO 9126 [1] (celle-ci sera abordée plus loin dans ce document).

L'outil open source SONAR sera utilisé pour évaluer la qualité de l'application S3MDSS.

La démarche qui sera adoptée est la suivante :

- utiliser le progiciel libre SONAR pour effectuer une série de mesures;
- interpréter chacune des mesures SONAR et émettre des recommandations

2 L'OUTIL SONAR

2.1 Principe

SONAR est un « outil open source pour la gestion de qualité du logiciel, dédié à l'analyse et la mesure continues de la qualité du code source » [2].

L'outil permet d'effectuer une évaluation de la qualité d'un produit logiciel, en effectuant un ensemble de mesures sur le code source, et sans exécuter le programme.

Les mesures effectuées par SONAR peuvent permettre de :

- ✓ détecter certaines erreurs potentielles, ou certaines mauvaises pratiques de programmation;
- ✓ prendre des décisions quant à la refactorisation de certaines composantes du logiciel.

2.1.1 Mesures effectuées par SONAR

Les types de mesure que SONAR peut effectuer ont les suivants :

- ✓ Taille du code source : nombre de packages, classes, méthodes, lignes de code, NCSS (Non Commenting Source Statements);
- ✓ Nombre de lignes de commentaires;
- ✓ Duplication de code;
- ✓ Taux de couverture (pourcentage d'instructions qui sont couvertes par un test unitaire);
- ✓ Complexité cyclomatique
- ✓ Conformité à un ensemble de règles de codage.

Dans la version 1.7 de SONAR qui sera utilisée dans cette analyse, la mesure du taux de couverture est disponible uniquement pour les projets construits avec MAVEN. D'où cette mesure ne sera pas effectuée dans le cas de l'application S3MDSS.

La copie d'écran ci-dessous représente la page sommaire qui récapitule, à haut niveau, les mesures effectuées par SONAR sur le code d'un projet (en l'occurrence, S3MDSS).

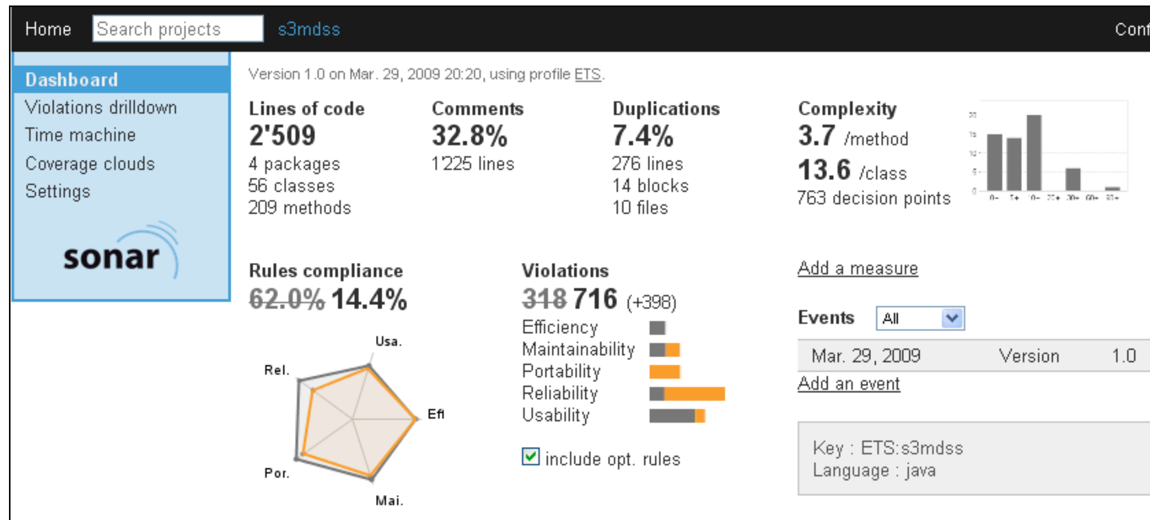

Figure 1 SONAR, sommaire des mesures d'un projet

2.1.2 Règles de codage SONAR

SONAR permet d'évaluer la conformité du produit logiciel vis-à-vis d'un ensemble de règles de codage. Les règles de codage sont réparties en 5 catégories : Rendement, Maintenabilité, Portabilité, Fiabilité et Facilité d'utilisation.

Les règles sont spécifiques au langage Java. Leur compréhension et l'estimation de leur importance nécessite la connaissance de ce langage.

Les sections ci-dessous donnent des exemples de règles de codage. Bien que la documentation n'indique pas la logique de regroupement des règles au sein des catégories, la revue de toutes les règles de SONAR a permis de déduire des objectifs communs qui relient les règles de chacune des catégories.

2.1.2.1 Rendement

Ces règles visent à améliorer la performance de l'application et à optimiser l'utilisation des ressources

Des exemples de règles de cette catégorie sont :

- ✓ Utiliser la méthode System.arrayCopy pour copier des éléments entre deux tableaux, au lieu de faire des itérations,
- ✓ Éviter l'instanciation d'objets de type Boolean, et utiliser les objets prédéfinis (Boolean.TRUE et Boolean.FALSE)
- ✓ Éviter la création de variables locales dans une méthode juste avant de retourner la valeur.
- ✓ Pour les attributs « final » utiliser le mot-clé « static » lorsque possible (afin d'éviter l'overhead lors de l'instanciation de chaque objet)

2.1.2.2 Maintenabilité

Ces règles visent à simplifier le code et faciliter le débogage et le suivi de l'exécution, ainsi que le traçage des problèmes.

Des exemples de ces règles sont :

- ✓ Utiliser les interfaces (ex : List) au lieu des types d'implémentation (ex : ArrayList) lorsque c'est possible
- ✓ Éviter de propager « RuntimeException » ou « Error » à l'aide des instructions « throw »
- ✓ Simplifier les expressions booléennes

- ✓ Préserver la trace d'exécution : lors de la propagation d'une exception, préserver l'exception d'origine.

2.1.2.3 Portabilité

Ces règles visent à permettre aux programmes Java d'être portables vers d'autres versions de la JVM.

Des exemples de règles de portabilité incluent :

- ✓ Éviter d'importer les packages « sun.* ». Ils ne sont pas portables (vers d'autres versions de Java) et peuvent changer
- ✓ Éviter l'utilisation de « enum » comme identificateur (dans Java 1.4) car il devient n mot-clef dans Java 5
- ✓ Éviter l'identificateur « assert » car il devient un mot-clef à partir de Java 5.

2.1.2.4 Fiabilité

Ces règles visent à détecter les erreurs de programmation conduisant à un comportement inexact du logiciel, ou à des erreurs d'exécution inattendues.

Des exemples de ces règles incluent :

- ✓ Éviter d'utiliser « equals » avec le paramètre « null »
- ✓ Comparer les objets avec « equals » et non avec « == » (qui retourne un résultat inexact)
- ✓ Fermer les ressources (fichiers, connexions aux BD…) après usage.

2.1.2.5 Facilité d'utilisation

Ces règles visent à faciliter la compréhension du code de l'application par le développeur

Des exemples de ces règles incluent :

- ✓ Utiliser les accolades pour délimiter les blocs de code contenus dans des itérations
- ✓ Éviter l'assignation de valeurs aux paramètres dans les méthodes
- ✓ Éviter « System.println » et utiliser un outil de journalisation (log)

2.2 Correspondances SONAR et ISO 9126

La norme ISO 9126 est la norme qui définit les critères de mesure de la qualité d'un produit logiciel. Le modèle de qualité de cette norme se base sur un ensemble de caractéristiques et de sous-caractéristiques de la qualité.

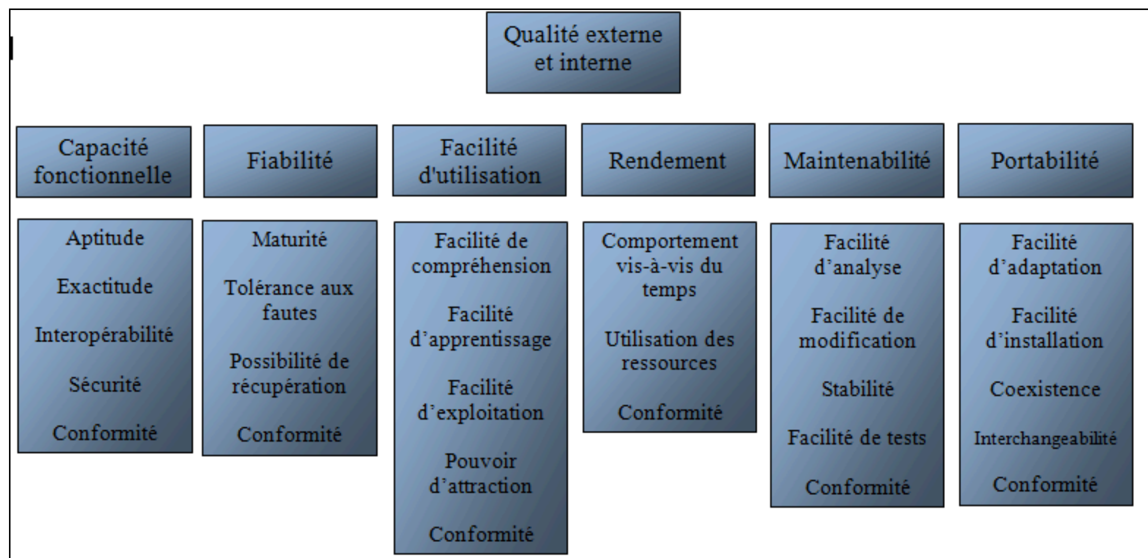

Figure 2 Caractéristiques de la qualité ISO 9126

Le tableau suivant regroupe les définitions des caractéristiques de qualité selon l'ISO 9126 [1]

Caractéristique	Définition
Rendement (efficiency)	*“The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions”*
Maintenabilité	*“The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.”*
Fiabilité (reliability)	*“The capability of the software product to maintain a specified level of performance when used under specified conditions”*
Portabilité	*“The capability of the software product to be transferred from one environment to another.”*
Facilité d’utilisation (usability)	*“The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions”*

Table 1 Définitions des caractéristiques de la qualité ISO 9126

En consultant ces définitions des différentes caractéristiques de qualité, on peut dire que les catégories de règles de codage de SONAR correspondent aux caractéristiques de qualité qui portent le même nom.

Pour la caractéristique de Facilité d’utilisation, on remarque que les règles de codage SONAR garde les mêmes concepts (facilité de compréhension, d’apprentissage), mais d’une perspective de développeur (ou de mainteneur) qui utilise le code du logiciel, au lieu de la perspective d’utilisateur qui utilise le logiciel.

3 EXÉCUTION DE SONAR

3.1 Configuration SONAR

Comme tout instrument de mesure, SONAR nécessite un calibrage. Le calibrage de SONAR se fait à travers le choix et la personnalisation du « profil ». Le profil regroupe les règles de codage à utiliser, leurs catégories, et l'indication des règles obligatoires (versus les règles optionnelles).

Par défaut, SONAR utilise un profil prédéfini appelé « SONAR way » qui comporte 91 règles de codage obligatoires, et 25 règles optionnelles.

Les utilisateurs peuvent créer un ou plusieurs personnalisés selon les besoins de l'entreprise ou la nature des projets.

3.2 Opération SONAR

SONAR se base sur d'autres outils pour effectuer les mesures (CheckStyle, PMD pour les règles de codage et le code dupliqué, Cobertura pour la couverture). Cependant, l'apport de SONAR est de son interface utilisateur très facile, et qui permet une approche « Drill-down », c'est-à-dire pouvoir passer d'une vue à très haut niveau qui agrège plusieurs projets, à une vue très pointue qui focalise sur une classe ou une méthode déterminée, en passant par des vues intermédiaires à différents niveaux de détails.

De plus, SONAR offre une vue axée sur le temps (« Time machine ») qui permet de visualiser et de comparer les différentes mesures effectuées sur différentes versions du logiciel pour des fins de comparaison et d'analyse des tendances.

4 ANALYSE DES RÉSULTATS

4.1 Mesure des duplications de code

4.1.1 Résultats de la mesure

SONAR a retourné les résultats suivants :

Métrique	Valeur
Pourcentage du code dupliqué	7.4%
Nombre de blocs de code dupliqués	14
Nombre de lignes de code dupliquées	276
Fichiers / Classes contenant du code dupliqué	10

Table 2 Résumé de la mesure de la duplication de code

La concentration du code dupliqué dans 10 fichiers de classes a permis de dresser le tableau ci-dessous :

Classe avec duplication	Nombre blocs dupliqués	Classe contenant les mêmes blocs de code
GestionObjets	4	GestionObjets (même classe)
TopConcept	2	TopConcept (même classe)
ModTCServlet2	1	AddTCServlet
AddTCServlet2	1	ModTCServlet3
AddKWServlet2	1	ModKWServlet3

AddCPServlet2	1	ModCPServlet3

Table 3 Duplication de code par classe

4.1.2 Interprétation des résultats

SONAR a permis aussi de visualiser les blocs de code dupliqués à l'intérieur du code des classes.

Les blocs dupliqués des classes GestionObjets et TopConcept représentent des boucles pour l'obtention de collections d'objets à partir de chaines de caractères.

Chacun des autres cas de duplication de code a eu lieu entre la classe de servlet qui crée un nouvel enregistrement d'une donnée et la classe qui modifie un enregistrement de cette donnée (ex : création d'un Keyword / Modification d'un keyword, création d'un Cas de problème / Modification d'un cas de problème).

4.1.3 Recommandations

- ✓ Rassembler le code dupliqué dans des méthodes réutilisables
- ✓ Revoir la conception et le découpage des traitements pour bénéficier des similarités entre les traitements d'ajout et de modification dans la base de données.

4.2 Mesure de la complexité par classe

4.2.1 Résultats de la mesure

SONAR indique une complexité par classe moyenne de 13.6. Cette valeur ne soulève pas de soucis particuliers quant à la complexité du code. Cependant, le graphe de

répartition de la complexité par classe représente des différences importantes, et des valeurs extrêmes :

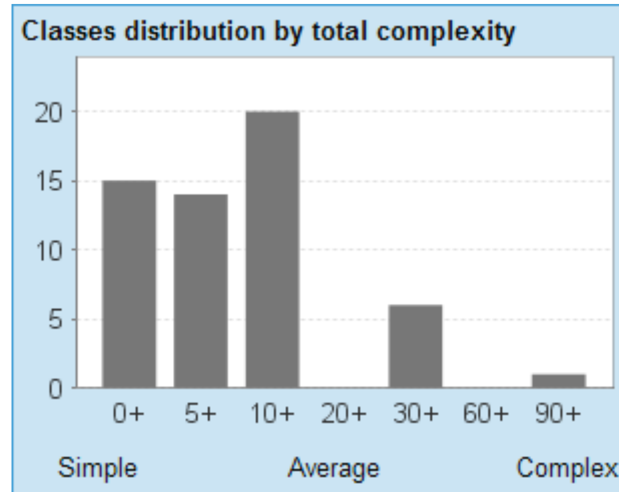

Figure 3 Répartition de la complexité / classe

Les classes qui possèdent les plus grandes valeurs pour la complexité par classe sont :

Classe	**Complexité par classe**	**Nombre de méthodes**	**Méthodes complexes**
GestionObjets	136	37	createVectors (16)
Utils	55	11	✓
MainServlet	37	2	doGet (18.5)
ExpertServlet	37	2	doGet (18.5)

Table 4 Les classes ayant les plus grandes complexités par classe

4.2.2 Interprétation des résultats

La classe GestionObjets

Une seule méthode possède une complexité cyclomatique au dessus de 7 (la méthode createVectors, qui a une complexité de 16). C'est elle qui cause une valeur élevée de al complexité par classe.

En lisant le code de cette méthode, on trouve que l'algorithme manque d'optimisation (lectures inutiles à partir de la base de données et algorithme inefficace pour éliminer les doublons dans des tableaux)

Par ailleurs, la classe GestionObjets est une classe utilitaire, qui offre des méthodes pour extraire différents types d'enregistrements à partir de la base de données, et instancier les objets ou les listes d'objets correspondants.

La classe Utils

En vérifiant le code de cette classe, on y trouve des méthodes qui offrent des utilités génériques (validation de date, conversion de chaines de caractères en date…) mais aussi des méthodes qui vérifient l'existence d'enregistrements dans la base de données (ieExistingCP, isExistingTH, isExistingREC).

Remarque : En analysant le code des deux classes GestionObjets et Utils, on déduit que le concepteur de ces classes s'est basé dans son découpage sur la nature du traitement (toutes les vérifications d'existence d'enregistrements sont dans Utils et les extractions de données sont dans GestionObjets). Ceci entraine un couplage très fort entre les composantes responsables de manipuler les différents types d'objets (Cas de Problèmes, Index, Recommandation…etc.)

Classes MainServlet et ExpertServlet

Ces classes sont des servlets, et ne possèdent, chacune, que deux méthodes doGet et doPost. Les valeurs élevées de complexité cyclomatique par classe sont dues à la complexité par méthode des méthodes doGet.

La méthode doGet du servlet MainServlet comporte 5 instructions « case », à l'intérieur de chacune d'elles il existe des instructions « if ».

La méthode doGet du servlet ExpertServlet comporte 17 instructions « case ».

Le nombre important d'instructions « case » explique la complexité élevée de ces deux classes.

Les instructions « case » sont utilisées pour choisir le type de traitement à effectuer, selon un paramètre de la requête http envoyée.

4.2.3 Recommandations

- ✓ Il est recommandé de réécrire la méthode createVectors afin de simplifier et optimiser son algorithme.

- ✓ Il recommande de découper le code de la classe GestionObjets afin de spécialiser les traitements selon les types d'objets manipulés. Ceci sera simple car il existe très peu d'interaction entre les méthodes, mais permettra de bien clarifier le code et de limiter le « couplage » entre les composantes.

- ✓ Il est recommandé de découper le code de la classe Utils sur des classes spécialisées selon le type de données manipulées. Cela permettra de réduire le « couplage » entre les composantes.

- ✓ Il est préférable de revoir la conception du cheminement des requêtes pour pouvoir simplifier les traitements dans les servlets MainServlet et ExpertServlet

4.3 Mesure de la complexité par méthode

4.3.1 Résultats de la mesure

SONAR a calculé une complexité moyenne par méthode de **3.6**. Celle-ci est considérée comme une bonne valeur. Cependant, certaines classes comprennent des méthodes avec une complexité supérieure à 10 (qui est au dessus de la limite acceptable).

Les classes qui possèdent une complexité par méthode « inacceptable » sont les suivantes :

Classe	Complexité par méthode
MainServlet	18.5
ExpertServlet	18.5
ModTCServlet2	9

Table 5 Les classes qui ont les méthodes les plus complexes

4.3.2 Interprétation des résultats

Le code des méthodes doGet des servlets MainServlet et ExpertServlet a déjà été discuté dans la section « 4.2.2 Interprétation des résultats » (de la complexité par classe)

Quant á la méthode doGet du servlet ModTCServlet2, elle ne comporte pas d'instructions « case » (elle effectue un seul type de traitement) mais elle comporte plusieurs instructions « if ».

4.3.3 Recommandations

- ✓ Il est recommandé de réécrire la méthode doGet du Servlet ModTCServlet2 pour simplifier l'algorithme

- ✓ Il est préférable de revoir la conception du cheminement des requêtes pour pouvoir simplifier les traitements dans les servlets MainServlet et ExpertServlet (même recommandation émise suite á la mesure de Complexité / Classe)

4.4 Respect des règles de codage obligatoires

4.4.1 Résultats de la mesure

Cette section présente et analyse les résultats retournés par SONAR relativement au respect des règles de codage, et ne tient pas compte des règles optionnelles.

La conformité globale de l'application est de 62%, avec 318 violations, qui impliquent 14 règles (sur un total de 91).

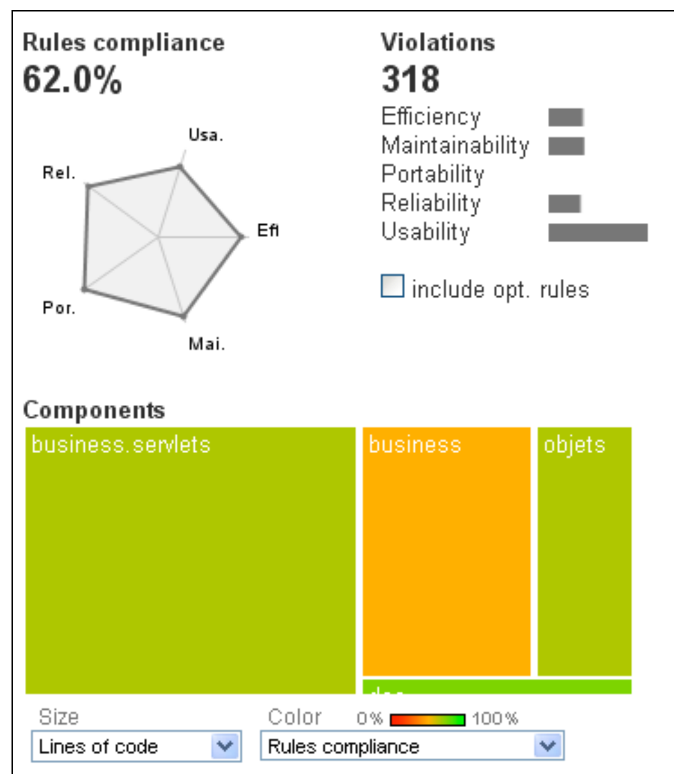

Figure 4 Résumé de la mesure de la conformité aux règles obligatoires

La répartition des violations des règles de codage sur les catégories est comme suit :

Catégorie	Nombre de violations
Rendement	55
Maintenabilité	56
Portabilité	0
Fiabilité	51
Facilité d'utilisation	156

Table 6 Violations des règles par catégorie

4.4.2 Interprétation des résultats

Violation des règles de fiabilité

Toutes les violations des règles de codage relatives à la catégorie de Fiabilité sont dues au problème de non fermeture de ressources.

L'inspection des classes a démontré qu'il s'agit pour la plupart, de la non fermeture des objets « ResultSet » lors des extractions à partir de la base de données. Cependant, pour tous les fragments de code qui ont été vérifiés, il a été constaté l'existence de la fermeture de l'objet « Statement » parent (ex : rs.getStatement().close())

Alors que SONAR indique 15 classes qui ne ferment pas les ressources après usage, la majorité des cas se trouve dans les classes suivantes :

Classe	Nombre de violations
GestionObjets	23

Utils	7
GestionUsers	6

Table 7 Classes ayant le plus de violations de règles de Fiabilité

Violation des règles de rendement

Les règles non respectées sont :

- L'utilisation du type « ArrayList » au lieu du type « Vector »,
- Cacher les constructeurs des classes utilitaires,
- Éviter la création de variables locales avant l'instruction « return ».

La première règle est particulièrement difficile à appliquer dans le contexte du S3M DSS car elle nécessiterait beaucoup de refactorisation du code (puisque le changement des signatures de plusieurs méthodes est requis).

Cependant, peu d'effort est requis pour faire respecter la deuxième règle. Il suffit d'ajouter un constructeur privé aux classes suivantes : GestionObjets, GestionEngine, GestionUsers et Utils.

De même, peu d'effort est requis pour la troisième règle.

Violation des règles de maintenabilité

Parmi les règles de cette catégorie, SONAR inclut :

- empêcher la complexité cyclomatique par méthode de dépasser une valeur de 7.

- empêcher une méthode d'avoir plus que 50 NCSS (Non Commenting Source Statements)

Ces règles ne seront pas discutées dans cette section car la complexité a été discutée dans les sections précédentes.

Outre ces règles mentionnées ci-dessus, la plupart des violations (28) concernent la visibilité des attributs des classes.

Violation des règles de la facilité d'utilisation

Les règles non respectées de la facilité d'utilisation concernent :

- l'utilisation d'accolades (« {«, « } ») dans les blocs de code sous les instructions « if », « else » et au sein des boucles (« loop », « for ») (117 violations),

- la convention de nommage des variables locales (23 violations),

- l'assignation de valeurs à des paramètres de méthodes.

Alors que le respect de toutes ces règles aurait permis une meilleure lisibilité du code et aurait facilité sa compréhension, l'assignation de paramètres représente un problème particulièrement important. Une telle modification change la valeur du paramètre pour la suite de l'exécution, mais risque de ne pas être « vue » par un autre développeur ou mainteneur qui essaie de comprendre le programme. Comme dans l'exemple qui suit :

```
22 public class GestionObjets {
23
24         /**
25          * Fonction permettant une liste de type "-X-X-X" en un Vecteur de Doublon
26          * @param String La liste a reproduire
27          * @return Un Vecteur de Doublon semblable à la liste
28          */
29         public static Vector<Doublon> createPartialVector(String list){
30                 Vector<Doublon> v = new Vector<Doublon>();
31                 list = list.substring(1);
32                 StringTokenizer tok = new StringTokenizer(list, "-");
33                 int i=0;
34                 while(i<list.length()){
35                         String id = tok.nextToken("-");
36                         v.add(new Doublon(id,null));
37                         i=i+id.length()+1;
38                 }
39                 return v;
40         }
```

Figure 5 Exemple d'assignation de valeur à un paramètre de méthode

Un mainteneur qui verrait un appel à la méthode GestionObjets.createPartialVector (String list) à partir d'une autre classe, ne pourrait pas s'attendre à ce que la valeur du paramètre soit modifiée par la méthode (qui est censée le paramètre pour produire la valeur à retourner).

4.4.3 Recommandations

Règles de fiabilité

- La fermeture des objets ResultSet est seulement souhaitable, puisque le fonctionnement de l'application ne sera pas affecté (les ressources étant fermées en demandant la fermeture de l'objet parent : Statement).

Règles de rendement

- ✓ Il est préférable de cacher les constructeurs des classes GestionObjet, GestionUsers, GestionEngine et Utils

- ✓ Il est préférable d'éliminer les variables créées juste avant l'instruction « return », puisque peu d'occurrences existent dans le code.

- ✗ Le remplacement de "Vector" par "ArrayList" nécessite la modification de plusieurs classes les sigantures de plusieurs méthodes. La modification demande beaucoup d'effort.

Règles de maintenabilité

- ✗ Il est recommandé d'éliminer les attributs publics en ajoutant des méthodes accesseurs (getters et setters), et ce malgré les efforts considérables que cela peut nécessiter. Les classes concernées sont : User, TopConcept, CasProbleme, Theme, Recom, Keyword, Index, Doublon.

Règles de facilité d'utilisation

- ✗ Éliminer l'assignation de paramètres dans tout le code de l'application, malgré les efforts importants qui seront nécessaires.

4.5 Respect des règles de codage optionnelles

4.5.1 Résultats de la mesure

SONAR a indiqué 4868 violations de règles de codage optionnelles. Cependant, 4470 parmi elles sont relatives á la règle « caractère tab » qui est marquée comme non respectée dans presque toutes les lignes de code de toutes les classes.

En vérifiant sur Internet, il a été trouvé que ceci est un problème faisant l'objet de problème ouvert [3]. D'où les résultats ont été régénérés en désactivant cette règle.

La figure ci-dessous présente la conformité de S3MDSS aux règles de codage optionnelles avant et après l'élimination de la règle « caractère tab » (le pourcentage initial de 0% ne permettait pas d'extraire de résultats).

Figure 6 Résumé de la mesure de la conformité aux règles optionnelles

SONAR a indiqué les violations suivantes :

Règle	Nombre de violations
« Design pour héritage »	122
« Remplacer Vector par ArrayList »	106
« Nombres magiques »	86
« Appel de System.println »	35
« Utilisation des interfaces dans les déclarations »	30

« Éviter la duplication des littérales »	17
« Déclaration throw du type Exception »	2

Table 8 Violations des règles optionnelles

4.5.2 Interprétation des résultats

La première règle indique que les méthodes de classes qui ne sont pas privées ou statiques doivent être finales ou abstraites ou avoir une implémentation vide. [3]

La dernière règle concerne les signatures de méthodes qui propagent une exception « Exception ». Cette exception trop générique donne peu d'information sur le type d'erreur attendu. Deux classes sont concernées : GestionUsers et Utils.

L'inspection du code des classes, a permis de constater que les deux méthodes dans les deux classes s'appellent « stringToDate » et acceptent les mêmes paramètres. Il s'agit en fait de deux blocs de code identiques : une duplication que SONAR n'a pas détectée.

Quant aux autres règles, les violations sont éparpillées à travers la majorité des classes.

4.5.3 Recommandations

- ✓ Il est recommandé d'introduire une librairie de journalisation (Log) au lieu des "System.println"

- ✓ Il est recommandé de modifier la signature des méthodes qui propagent une exception générique (Exception)

- ✗ Il n'est pas recommandé de respecter la règle « Design pour héritage » qui est jugée trop rigide, surtout qu'elle diminue la souplesse lors de l'héritage de classes. Il est recommandé de la désactiver.

Par ailleurs, un problème sera ouvert auprès de SONAR afin d'investiguer les raisons pour lesquelles l'outil n'a pas détecté la duplication de code dans les méthodes « stringToDate »

4.6 Classes les plus critiques

Comme la taille du code de l'application n'est pas très grande, il a été possible de compiler le tableau ci-dessous qui représente les classes ayant le plus de problèmes, selon les mesures SONAR :

Classe	Duplication du code	Complexité / classe	Complexité / méthode	Règles obligatoires	Règles optionnelles
GestionObjets	☑	☑		☑	
TopConcept	☑			☑	
Utils		☑		☑	☑
MainServlet		☑	☑		
ExpertServlet		☑	☑		

Table 9 Classes pour lesquelles SONAR a remonté le plus de problèmes

D'après ce tableau, la priorité la plus haute doit être accordée aux classes Utils et GestionObjets, puis aux classes GestionUsers, TopConcept et ModTCServlet2.

5 COMPARAISON DES RÉSULTATS DE SONAR AVEC CEUX DE L'INSPECTION « VISUELLE »

Cette section tente de comparer brièvement les résultats trouvés grâce à SONAR, et les résultats d'une inspection du code, qui a été effectuée en se basant uniquement sur les possibilités offertes par Eclipse (recherche texte, recherche de références, recherches d'hiérarchie d'héritage… etc.). Cette inspection a permis de détecter les problèmes suivants :

- L'application a un problème de **couplage fort** : Plusieurs classes ont des responsabilités trop larges. Une refactorisation est recommandée pour la spécialisation des classes et le découplage

- L'application a un problème de **portabilité** : Les chaines de caractères des chemins des fichiers et des paramètres d'accès à la BD sont "en dur" dans le code

- L'application a un problème de rendement : L'obtention d'une connexion à la base de données se fait en instanciant le driver JDBC (sans bénéficier des capacités des pools de connexions du serveur)

- Les algorithmes d'extraction de données sont généralement inutilement complexes et manquent d'optimisation.

6 CONCLUSION

6.1 L'application S3MDSS

- ✓ Le code source ne comporte pas de bogues potentiels, selon les l'outil SONAR.

- ✓ Une refactorisation d'une partie importante du code est recommandée pour améliorer la qualité, et faciliter la maintenance

- ✓ Une réécriture de certaines parties du code est recommandée pour simplifier et optimiser les algorithmes

- ✓ Une réécriture de certaines parties du code est recommandée afin de faciliter la compréhension du logiciel, et de minimiser le risque d'effets de bord lors de modifications futures

Par ailleurs, il est recommandé de démarrer un projet visant à une revue de code massive et à une refactorisation agressive de l'application S3MDSS pour améliorer la maintenabilité

6.2 L'outil SONAR

- ✓ SONAR a permis d'effectuer un ensemble de reproductibles et objectives (basées sur un ensemble de critères concrets)

- ✗ SONAR présente une limitation importante quant à la mesure du couplage entre les composantes applicatives. Cette mesure serait très bénéfique dans le cas de S3MDSS.

- ✗ SONAR n'a pas détecté le "code mort" (classes et méthodes non utilisées)

- L’inspection du code est souvent nécessaire pour interpréter les résultats des mesures

- La connaissance du langage de programmation utilisé est nécessaire pour comprendre les règles de codage et estimer leur importance

6.3 Pour une utilisation optimale de l’outil SONAR

Pour maximiser l’efficacité de l’outil SONAR, la liste des règles de codage doit être revue et enrichie au besoin

De plus, l’utilisation du “Time Machine” de SONAR permet de suivre la qualité du code au fil du temps et d’analyser les tendances.

L’utilisation de SONAR ou d’un autre analyseur de code ne peut remplacer la revue de code et la revue de la conception, mais elle offre un moyen complémentaire facile et rapide pour détecter plusieurs problèmes dans le code source du logiciel.

BIBLIOGRAPHIE

[1] Standard ISO / IEC 9126 – Partie 1: 2001

[2] Documentation SONAR, [en ligne] : < http://sonar.codehaus.org/> (consulté le 02 Avril 2009)

[3] Référence du problème ouvert concernant la règle de codage «caractères tab», [en ligne] : <http://jira.codehaus.org/browse/SONAR-717> (consulté le 29 Mars 2009)

[4] Documentation de l'outil CheckStyle, règle de codage « design pour l'héritage », [en ligne] : <http://checkstyle.sourceforge.net/config_design.html#DesignForExtension> (consulté le 29 Mars 2009