Université du Québec
École de technologie supérieure

RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS MGL804

ANALYSE DE LA MAINTENABILITÉ D'UN PROJET

CHRISTOPHE COMMEYNE
COMC09038002

DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

Professeur superviseur

ALAIN APRIL

MONTRÉAL, MARS 2012
HIVER 2012

ANALYSE DE LA MAINTENABILITÉ D'UN PROJET

CHRISTOPHE COMMEYNE
COMC09038002

RÉSUMÉ

Faites une analyse de la maintenabilité d'un logiciel à l'aide de logiciels d'évaluation de la qualité (Checkstyle, Logiscope, etc.). Suivez l'approche proposée en classe qui décrit les étapes à suivre pour effectuer une analyse reproductible, impartiale et objective.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

LPR	License Plate Recognition
IL	Intermediate Language
CIS	Continuous Integration Service
SDP	Stable Dependency Principle
LCOM	Lack of Cohesion Methods

INTRODUCTION

Genetec a été fondée en 1998 et a développé un logiciel de vidéosurveillance sur IP, *Omnicast*. Par la suite, une application de contrôle d'accès, *Synergis*, également sur IP, a été développée. Finalement, le troisième module chargé de faire de la reconnaissance de plaques d'immatriculation, *Autovu* a été ajouté à la liste d'applications développées par la compagnie.

En 2009, la plateforme de sécurité unifiée *Security Center* a vu le jour et regroupait les applications *Synergis* et *Autovu* dans un premier temps, puis, en 2010, *Omnicast* a finalement rejoint la plateforme. Entre temps, de nouveaux modules tels que l'intégration des Panneaux d'alarme, le système de fédération, des outils de moniteurs d'états, de redondance ont été intégrés. À l'heure actuelle, on recense plus de 350 projets C# dans *Security Center*. On estime à près de 3 millions le nombre de lignes de code de la plateforme.

CHAPITRE 1

PRÉSENTATION

1.1 Choix du projet

Tel que mentionné dans l'introduction, la plateforme *Security Center* contient un nombre importants de modules. La sélection d'un sous-ensemble de projets m'a paru plus intéressante car elle permettra porter l'attention sur l'impact que peuvent avoir différentes métriques sur la maintenabilité du code.

Ainsi le module de reconnaissance de plaques d'immatriculation (LPR) d'*Autovu* fera l'objet de mon analyse de maintenabilité. Les origines (connues) de ce module remontent à l'année 2005 et il est aujourd'hui considéré *Legacy*. Dans ce contexte, nous allons pouvoir voir son état après sept ans d'existence.

1.2 Méthodologie

Pour analyser la maintenabilité du module, je me suis appuyé sur la norme ISO9126 qui établit les caractéristiques définissant la qualité d'un logiciel. La maintenabilité est déterminée par les quatre sous-caractéristiques suivantes :

- Facilité d'analyse;
- Facilité de modification;
- Stabilité;
- Testabilité.

J'ai utilisé le compendium intitulé « Software Quality Standards and Metrics » rédigé en 2007 par les Dr Rüdiger Lincke et Welf Löwe de l'université Växjö (Suède) [1]. Ce

compendium définit des liens entre certaines métriques et leurs impacts sur la maintenabilité du logiciel. J'ai effectué le choix des métriques suivantes :

- Couplage (afférent et efférent) d'un assembly;
- Instabilité d'un assembly;
- Abstraction d'un assembly;
- Cohésion relationnel d'un assembly;
- Cohésion d'un type de données (LCOM);
- Complexité cyclomatique d'une méthode;
- Nombre de paramètres d'une méthode;
- Nombre de variables dans une méthode.

Ces métriques me permettront de cerner d'éventuel problème la maintenabilité du module à différents niveaux, en commençant par les assemblies, puis en poursuivant par les types de données et pour finalement terminer par les méthodes. Le point important à souligner à cette étape est que de bonnes métriques n'impliquent pas pour autant que le logiciel n'aura aucun problème de maintenabilité. De bonnes métriques signifient simplement que le logiciel respecte ces métriques en particulier. Par contre, de mauvaises métriques impliquent que des problèmes de maintenabilité sont à entrevoir, et ce, à différents degrés de sévérité.

1.3 Les outils pour l'analyse

1.3.1 ReSharper

L'outil est développé par la compagnie JetBrains facilite la réingénierie de code. Une des autres fonctionnalités de ce programme est de pouvoir informer le développeur en temps réel des différents problèmes dans son code, tels que des erreurs de syntaxe, des optimisations de toutes sortes, du non-respect de conventions. Cet outil sera utilisé pour analyser la forme du code après avoir été calibré.

1.3.2 Metrix

Le second outil à notre disposition est une application prototype développée à l'interne chez Genetec et qui offre le calcul de certaines métriques de projets .NET.

1.3.3 Visual Studio

Les tests unitaires et systèmes sont exécutés par l'environnement de développement Visual Studio. Pour certains projets, ces tests sont exécutés toutes les nuits et des rapports sont envoyés aux équipes pour les informer des résultats. Un rapport de couverture de code est généré pour certains de ces tests.

1.4 Vue d'ensemble

Le module LPR regroupe 15 assemblies contenant un peu plus de 45000 lignes de code logiques. On y dénombre 489 types de données proposant un éventail de 4115 méthodes. La Figure 1 présente une vue d'ensemble du module avec les liens entre les différents assemblies et, pour chacun, le nombre de lignes de code s'y trouvant.

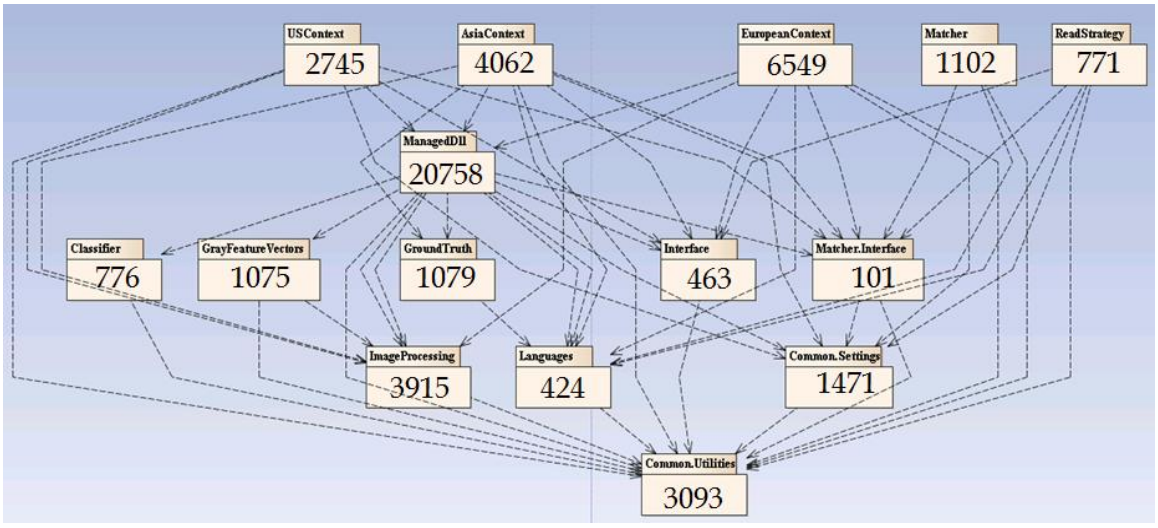

Figure 1 – Structure du module LPR

1.5 Calibration de ReSharper

La compagnie a défini un ensemble de règles à respecter lorsqu'on utilise le langage C#. Ces règles ont trouvé leurs origines auprès d'un livre écrit par des architectes de Microsoft il y a quelques années maintenant [2]. Un comité a été créé chez Genetec et a reçu la responsabilité de maintenir ce document et de le mettre à jour à chaque nouvelle version du langage.

Malgré l'existence de ce document et la disponibilité d'outils dans notre environnement de développement permettant de valider la conformité du code source, ces règles n'ont pas été incluses dans la CIS. En conséquence, le respect de ces règles est laissé à la discrétion des différentes équipes de développement.

Les règles de la compagnie ont déjà été définies dans ReSharper. L'outil effectue l'analyse en temps réel du code source visualisé et fournit des indications aux quels endroits des problèmes surviennent ou des suggestions sont disponibles. L'outil génère plusieurs informations en plus des règles que nous avons définies. Parmi ces informations additionnelles, il y a des suggestions d'amélioration du code basé sur les fonctionnalités du langage (précision d'intention, lisibilité), des suggestions de problèmes pouvant générer des comportements non voulus (variable inutile, exception non trappées) ainsi que des avertissements du compilateur. Par contre, les propositions qui encouragent l'utilisation d'expression lambda, LINQ et plusieurs autres ont été retiré car la plupart de ces suggestions entrent en conflit avec les règles émises par le comité.

1.6 Résultats

Le résultat de l'inspection du module LPR est présenté dans la Figure 2, où on peut constater qu'un peu de plus de 3500 problèmes sont énumérés. La grande majorité de ces problèmes se trouvent le nommage des classes et champs, où 2185 problèmes ont été détectés. Il s'agit d'un problème extrêmement aisé à corriger, particulièrement avec ce type d'outils à notre disposition.

Les problèmes relevant des bonnes pratiques d'utilisation du langage sont au nombre de 620. Ces erreurs sont plus délicates à corriger mais le gain indéniable est une meilleure précision des intentions du programmeur et une diminution des risques d'une mauvaise utilisation des capacités du langage. Par exemple : réduction de la portée d'une variable, réduction de la visibilité d'un champ, utilisation de la méthode `string.IsNullOrEmpty()`.

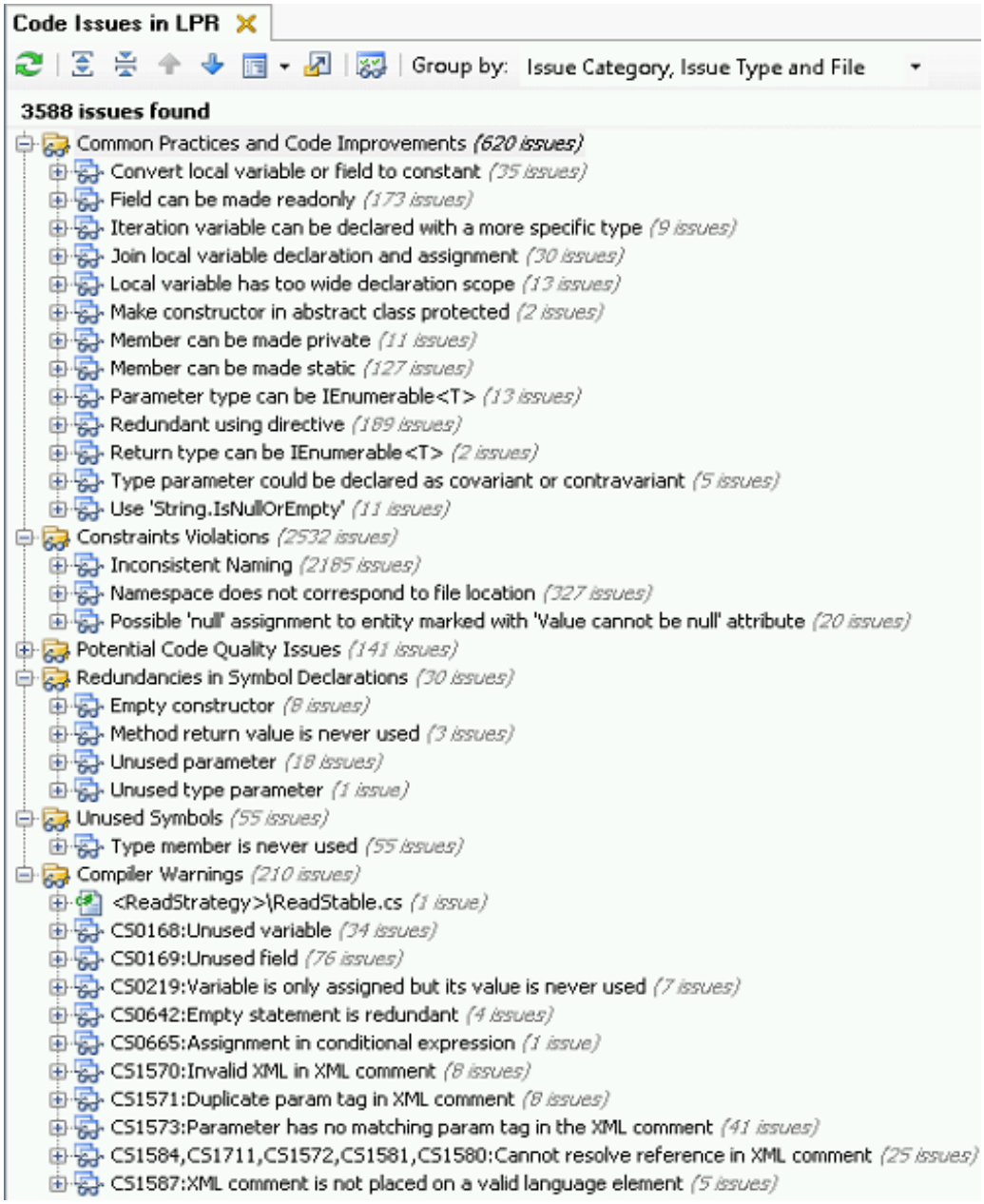

Figure 2 – Inspection du code

CHAPITRE 2

MÉTRIQUES

2.1 Métriques des assemblies

Lorsque les applications dépassent 50000 lignes de code, Robert Cecil Martin [3] explique que les développeurs se doivent de respecter certains principes fondamentaux en design afin de faciliter les futurs changements. Le module LPR a dépassé ce seuil sachant que le nombre de lignes de code calculées correspond au nombre de lignes de code logique. Le nombre de lignes physique est bien supérieur à ce nombre. L'auteur décrit [4], ainsi dans son livre [5], plusieurs métriques qui permettent d'analyser les relations entre différents assemblies : l'analyse des assemblies suit la logique qui est présentée dans ces documents.

2.1.1 (In) stabilité

Nous allons nous attarder en premier lieu à la stabilité des assemblies qui fait appel à trois métriques :

- Couplage afférent (Ca);
- Couplage efférent (Ce);
- Instabilité (I).

Pour un assembly donné, le couplage afférent mesure le nombre de types de données externes qui dépendent des types internes à l'assembly en question. À l'opposé, le couplage efférent mesure le nombre de types de données externes dont dépendent les types internes. L'instabilité I est alors calculée à partir des données de couplage :

$$I = \frac{Ce}{(Ca + Ce)}$$

Un indice d'instabilité de 0 indique que le composant est stable. À l'inverse, une valeur de 1 définie un composant instable. Les termes « stable » / « instable » méritent d'être expliqués car ils peuvent amener à une mauvaise interprétation.

Ici, le concept d'instabilité provient du principe de design des dépendances stables (SDP). Dans ce principe, on dit qu'une application ne peut pas être composée uniquement d'éléments statiques. Un certain niveau de volatilité est nécessaire pour qu'elle puisse évoluer durant la maintenance. Par « stable », on désigne un composant qui a beaucoup de responsabilités et qui sera difficile à modifier : il est rigide. Inversement, un composant instable est un composant qui a peu ou pas de responsabilités et qui pourra être modifié sans grandes conséquences sur le reste du programme. Si le composant n'est ni stable, ni instable, il est considéré flexible. Ainsi, selon le principe énoncé, un composant stable ne doit pas être dépendant d'un composant moins stable que lui (dont sa métrique d'instabilité est supérieure à la sienne) [4].

Tableau 1 - Stabilité des assemblies (Metrix)

	Couplage afférent	Couplage efférent	Instabilité
AsiaContext	0	37	1,00
Classifier	3	1	0,25
Common.Settings	45	4	0,08
Common.Utilities	142	0	0,00
EuropeanContext	0	49	1,00
GrayFeatureVectors	2	2	0,50
GroundTruth	2	3	0,60
ImageProcessing	4	11	0,73
Interface	43	1	0,02
Languages	16	0	0,00
ManagedDll	109	35	0,24
Matcher	0	13	1,00
Matcher.Interface	16	1	0,06
ReadStrategy	0	9	1,00
USContext	0	30	1,00

Le Tableau 1 présente ces métriques. Le premier problème constaté est que l'assembly ManagedDll rassemble près de la moitié du nombre de lignes de code du module. Les risques de changements y sont donc très élevés. Or, cet assembly a un indice d'instabilité de 0,24, ce qui signifie une rigidité importante causée par les nombreuses dépendances des types externes sur cet assembly. L'outil en a dénombré 109. En conséquence, les modifications dans cet assembly (qui seront tôt ou tard nécessaires) risquent d'avoir de nombreux impacts sur les assemblies USContext, AsiaContext et EuropeanContext.

La Figure 3 présente les résultats du Tableau 1 sous forme graphique et met en valeur le principe de design mentionné précédemment. Les violations du principe de dépendances stables ont été mises en rouge. Trois des quatre assemblies en rouge ont un indice d'instabilité bien supérieur à celui de ManagedDll avec pour conséquence principale que si des changements dans ces assemblies nécessitent des ajustements dans ManagedDll, ceux-ci risquent de ne pas être évident à réaliser à cause du niveau de rigidité élevé.

Figure 3 - Dépendances des projets

2.1.2 Abstraction

L'indice d'instabilité I peut être nuancé par le concept d'abstraction *A* qui est déterminé en calculant le ratio *Nombre de classes abstraites* (*Na*) sur le *Nombre total de classes* (*Nc*). Cet indice nous permettra de répondre à la question suivante : si l'assembly est rigide, est-ce qu'il possède un niveau d'abstraction suffisamment élevé pour permettre des interventions sans trop d'impacts ?

Toujours selon Robert Martin [5], si un composant est extrêmement stable (I=0), il devrait être extrêmement abstrait (A=1) afin de permettre une évolution sans problèmes. Inversement, un composant instable (I=1) n'a pas besoin d'un niveau d'abstraction (donc A=0) puisque son évolution est implicitement permise à cause de son instabilité.

À partir de cette définition, l'objectif est de dénombrer les classes abstraites (et interfaces) de chacun des composants. Le Tableau 2 présente les résultats de la métrique d'abstraction.

Tableau 2 - Abstraction des assemblies

	Classes abstraites (Na)	Classes (Nc)	Abstraction
AsiaContext	2	29	0,07
Classifier	1	11	0,09
Common.Settings	4	22	0,18
Common.Utilities	18	59	0,31
EuropeanContext	6	46	0,13
GrayFeatureVectors	2	7	0,29
GroundTruth	1	21	0,05
ImageProcessing	23	42	0,55
Interface	9	15	0,60
Languages	1	5	0,20
ManagedDll	20	132	0,15
Matcher	3	21	0,14
Matcher.Interface	4	7	0,57
ReadStrategy	4	12	0,33

USContext	0	60	0,00

Seule, cette mesure ne nous apprend rien. C'est lorsqu'elle est mise en relation avec la métrique d'instabilité qu'elle prend tout son sens. La Figure 4 (page ci-dessous) présente cette relation et voici son interprétation.

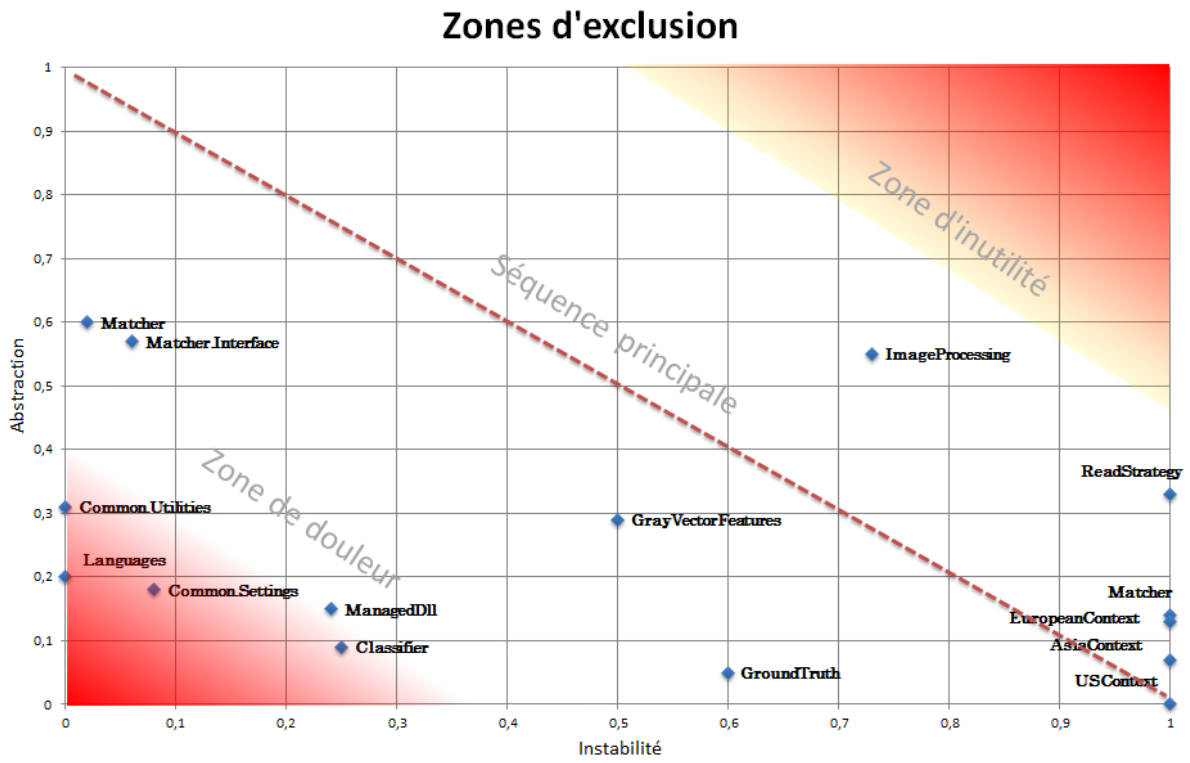

Figure 4 - Relation Instabilité / Abstraction

Le coin en haut à gauche montre des composants stables (rigides) et abstraits : ils ne sont pas supposés changer. Mais si le cas se présentait, leur niveau d'abstraction devrait être suffisant pour permettre une évolution sans trop d'impacts.

Le coin en bas à droite définit des composants instables (concrets) dont personne ne dépend et donc qui pourront s'adapter aux futures évolutions avec peu ou pas de conséquences.

Le coin en haut à droite montre des composants instables et abstraits. Être dans cette zone n'est pas désirable parce que cela signifie essentiellement que les assemblies comportent du code inutile : ils sont abstraits mais aucun autre composant n'en dépend.

Finalement, le coin en bas à gauche contient les composants stables mais avec peu ou pas de couche d'abstraction : il s'agit d'une zone à éviter car les changements des assemblies s'y trouvant peuvent être lourds de conséquences à cause des dépendances qu'ils détiennent et des responsabilités qu'ils possèdent. L'objectif recherché avec cette figure est d'obtenir des composants qui suivent la ligne tracée du coin en bas à droite au coin en haut à gauche.

On peut constater que la majorité des composants sont positionnés le long de la ligne, à une distance acceptable. Toutefois, cinq assemblies se trouvent dans la zone de douleur. Parmi ces assemblies se trouve ManagedDll. Cela signifie que l'assembly est rigide mais possède un faible niveau d'abstraction confirmant ainsi le problème mentionné précédemment.

Toutefois, il existe un cas dans lequel avoir une assembly dans cette zone est acceptable : les librairies utilitaires. Ce type de librairie étant un rassemblement de fonctionnalités qui n'ont pas nécessairement de liens, il est tout à fait justifiable d'avoir des types très utilisés par d'autres assemblies mais avec aucune couche d'abstraction. Il existe un indicateur permettant de savoir si ManagedDll pourrait être considérée comme une librairie utilitaire : sa cohésion. Une très faible cohésion indiquerait que l'assembly est composé d'un ensemble de classes qui n'ont pas ou peu de relations entre elles, très typique d'une librairie utilitaire.

2.1.3 Cohésion relationnelle

La cohésion (H) d'un assembly mesure le nombre moyen de relations d'un type de données d'un assembly avec les autres types de données de ce même assembly. Contrairement aux métriques précédentes, il s'agit d'une mesure indépendante et qui ne met pas en relation l'assembly avec les autres. La valeur de cohésion devrait être comprise entre 1,5 et 4,0. Une

valeur trop faible indique un assembly composé de classes ayant un couplage trop faible. Un indice dépassant 4,0 est signe d'un couplage trop fort [6] [7].

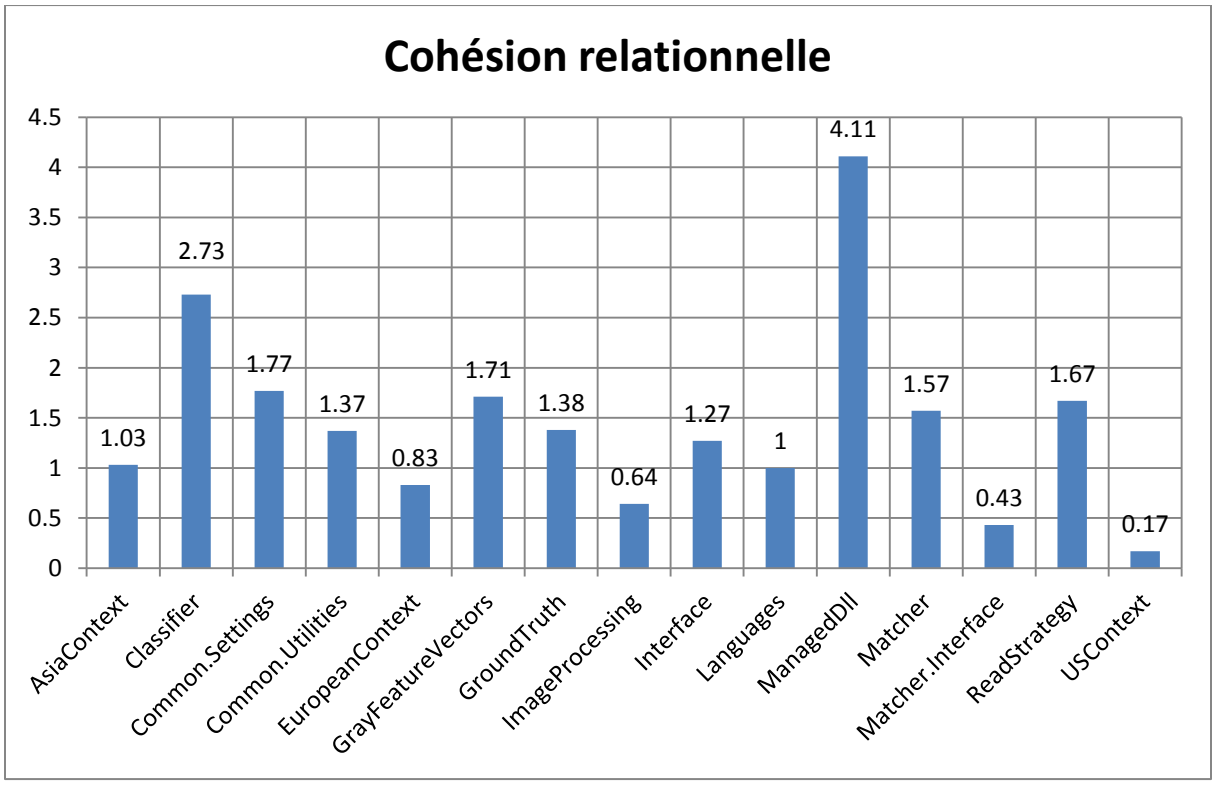

Figure 5 - Cohésion relationnelle

La Figure 5 confirme que ManagedDll n'est pas une librairie utilitaire. Bien au contraire, celle-ci est extrêmement cohésive et donc devrait se trouver aussi loin que possible de la zone de douleur. De manière similaire, la librairie Classifier devrait également être sortie de cette zone puisque qu'elle ne semble pas être une librairie utilitaire. En revanche, il n'y pas d'inconvénients à laisser les assemblies Languages, Common.Settings et Common.Utilities là où elles se trouvent présentement. Leur cohésion semble indiquer qu'il s'agit de librairies utilitaires.

2.1.4 Couverture de code des assemblies

Une dernière métrique intéressante concernant les assemblies est la couverture de code obtenue par les tests unitaires et systèmes. Cette information laisse place à beaucoup d'interprétations différentes. Dans le contexte qui nous intéresse, cette valeur nous donne un aperçu des problèmes que nous pourrions rencontrer lors d'interventions de maintenance. De bons tests et une bonne couverture sont toujours d'une grande aide dans la mesure où ils nous permettent de vérifier si les cas prévus par l'auteur original sont encore fonctionnels.

Nous pouvons constater que qu'à peine la moitié du code est couvert par les tests (Figure 6).

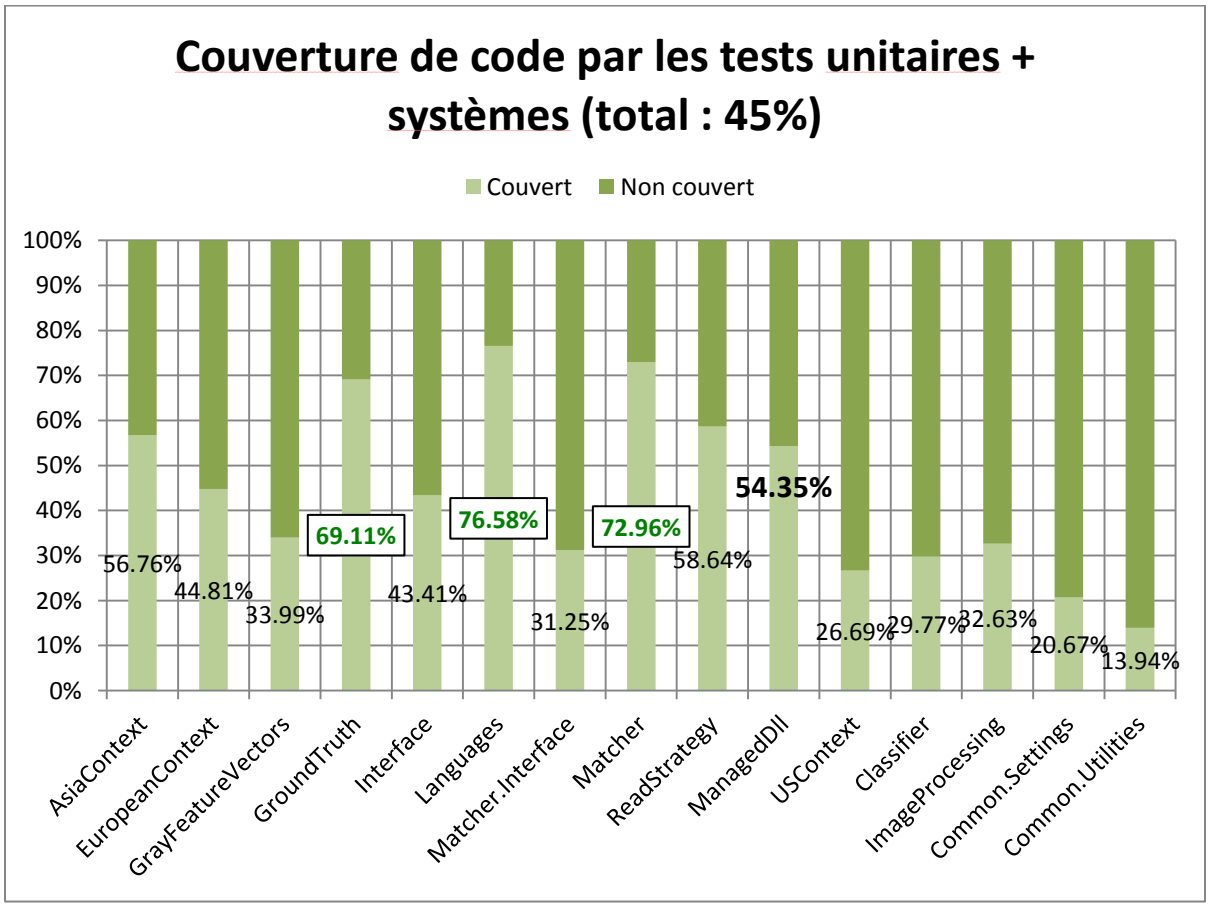

Figure 6 - Couverture du code par les tests

2.1.5 Synthèse de l'état des assemblies

L'instabilité d'une assembly à elle seule ne donne pas directement d'information sur la maintenabilité d'une application. Comme nous l'avons vu, des assemblies peuvent avoir une instabilité I de 1 sans que cela soit source de problèmes, dans la mesure où le composant a été conçu pour être instable. À l'inverse, un composant peut avoir une instabilité I de 0 si c'est là l'intention de son créateur. Tout réside dans l'intention.

Par contre, les relations entre les différentes assemblies se doivent de respecter le principe de dépendances stables compte tenu de la taille du projet. Il est critique que les composants reposent sur des composants plus stables qu'eux. Or, une partie importante du module LPR ne respecte pas ce principe : l'assembly le plus important en taille ManagedDll possède beaucoup de responsabilités et est amené à subir des interventions de maintenance de par sa taille. En conséquence, les risques de complications lors d'une intervention de maintenance dans un de ces modules sont fort probables.

2.2 Métrique des types de données utilisateurs

Les métriques intéressantes pour évaluer la maintenabilité des types de données des composants sont les suivantes :

- LCOM;
- Nombre de champs;
- Nombre de méthodes;
- Nombre de lignes de code.

La métrique LCOM se base sur le nombre de champs et le nombre de méthodes. Son calcul se base sur le principe que chaque méthode devrait accéder à tous les champs du type de données.

Voici la formule utilisée :

$$LCOM = 1 - \frac{\sum Nombre\ d'accès\ aux\ champs}{Méthodes\ .\ Champs}$$

2.2.1 Cohésion des types de données

J'ai choisi de débuter par la métrique LCOM accompagnée du nombre de champs et du nombre de méthodes dans les types de données. Il est recommandé d'avoir un indice LCOM inférieur à 0,8 et un nombre de champs et de méthodes inférieur à 10 [6].

La Figure 7 présente le nombre de types de données violant la métrique par assembly. Ce que l'on peut constater premièrement est que ManagedDll concentre la moitié de toutes les violations (28 des 56 types). Ce n'est pas une surprise en soi sachant que cet assembly concentre presque la moitié du code du module LPR.

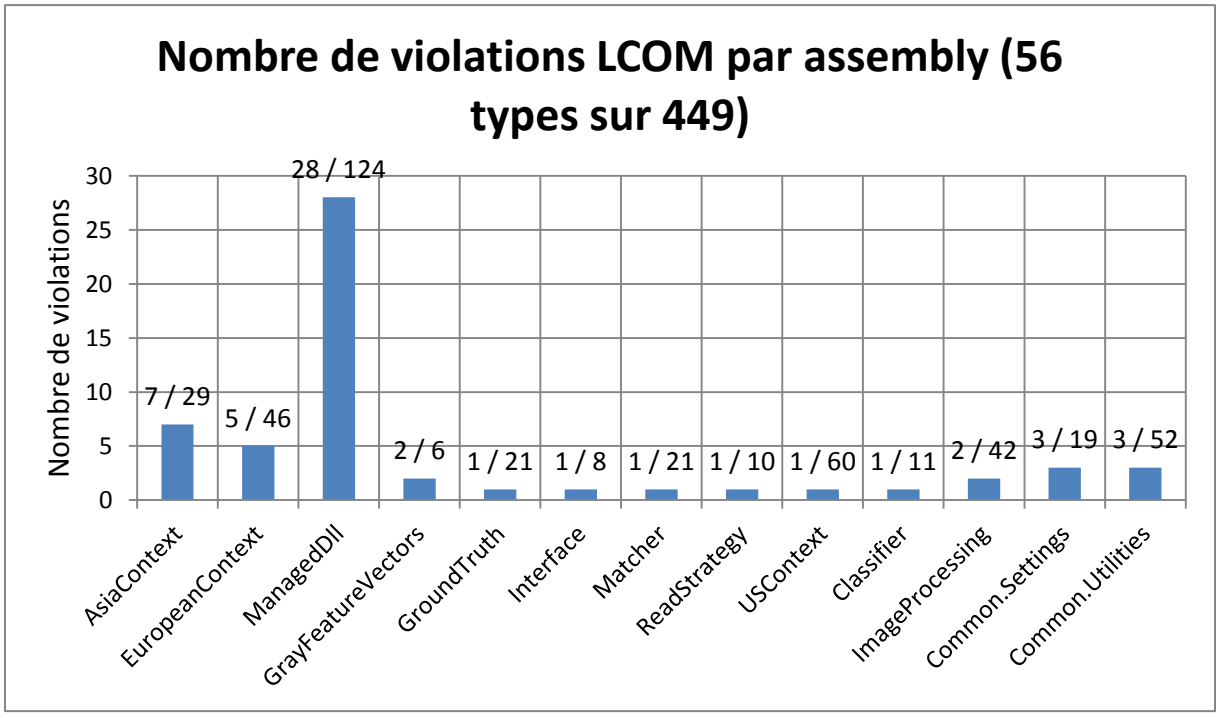

Figure 7 - LCOM - Nombre de violations par assembly

Au total, 12 % des types de données du module présentent une mauvaise cohésion. Ce que cela signifie, c'est que ces types de données semblent détenir plusieurs responsabilités.

2.2.2 Nombre de lignes de code

Cette problématique de responsabilités se traduit habituellement par un nombre de lignes de code très élevé. La Figure 8 met cette situation en valeur : on y présente la moyenne du nombre de lignes de code des types de données qui manquent de cohésion par rapport à la moyenne du nombre de lignes de code de tous les types de données d'un assembly. On peut constater un rapport minimum de deux entre les types en violations et le total de tous les types de données.

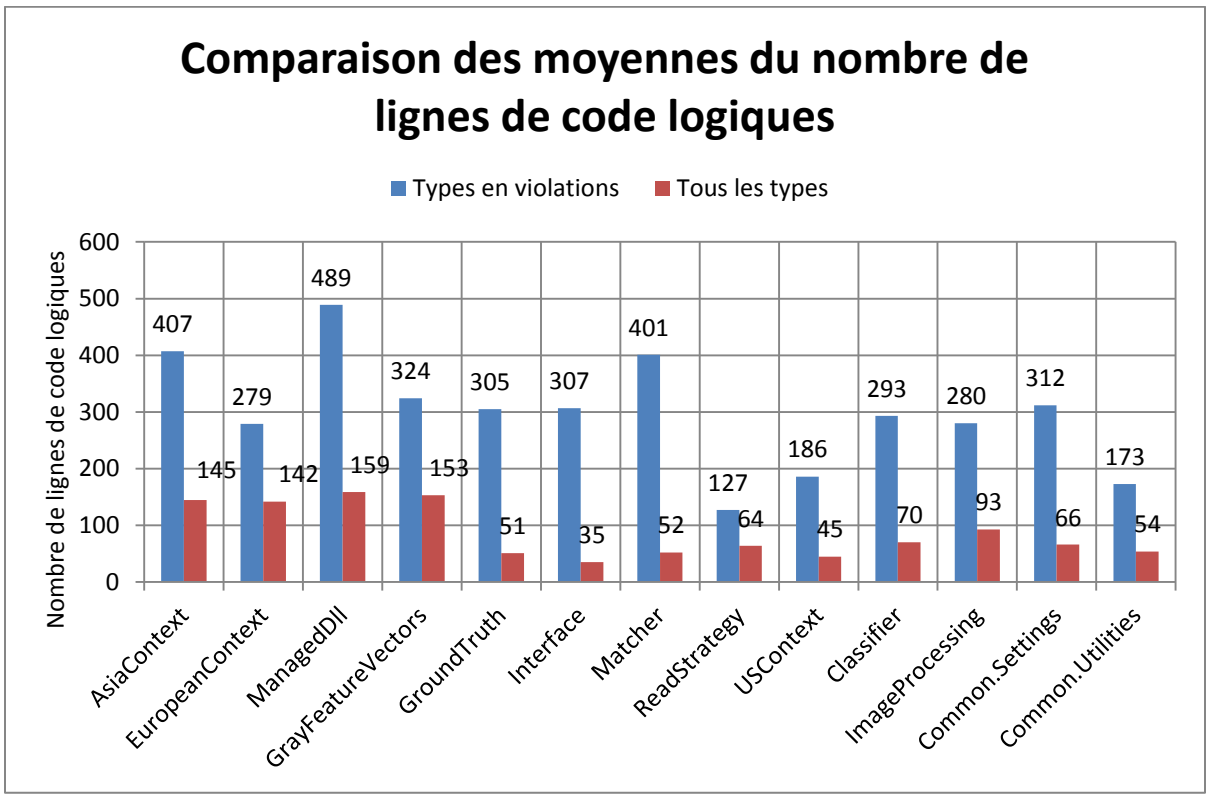

Figure 8 - Comparaison des moyennes des lignes de code

L'autre point intéressant à prendre en compte est le potentiel de violation, c'est-à-dire le nombre de types de données sur le point de dépasser le seuil. Pour ce faire, j'ai uniquement abaissé le nombre de champs et de méthode à 8 au lieu de 10. La Figure 9 présente les résultats. Au total, nous passons à 18 % de types de données ayant une faible cohésion (au lieu de 12 %). Le cas le plus flagrant concerne l'assembly GrayFeatureVectors. Il comporte peu de types mais néanmoins, tous ont un grand potentiel de violations rendant ainsi l'assembly difficile à analyser et à tester.

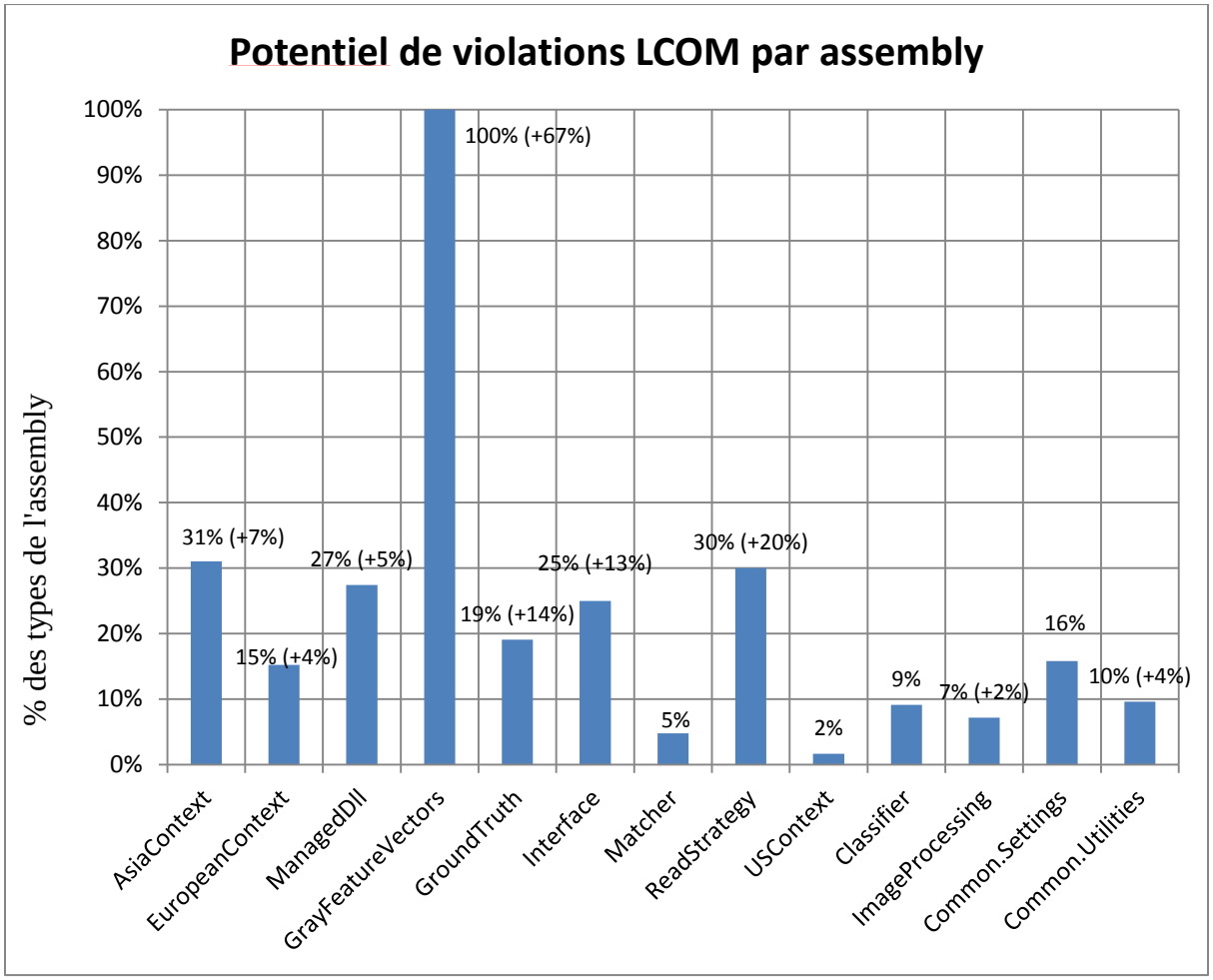

Figure 9 - Potentiel de violations LCOM par assembly

2.2.3 Nombre de champs et de méthodes

Les deux dernières métriques apportant une information supplémentaire est le nombre de champs et de méthodes dans un type de données. La métrique précédent définit le seuil à 10 mais, lorsque les types dépassent cette valeur, jusqu'à combien peuvent-ils en contenir ? La Figure 10 présente le nombre de champs maximum dans un type de données par assembly, ainsi que le nombre moyen de ces champs par classes.

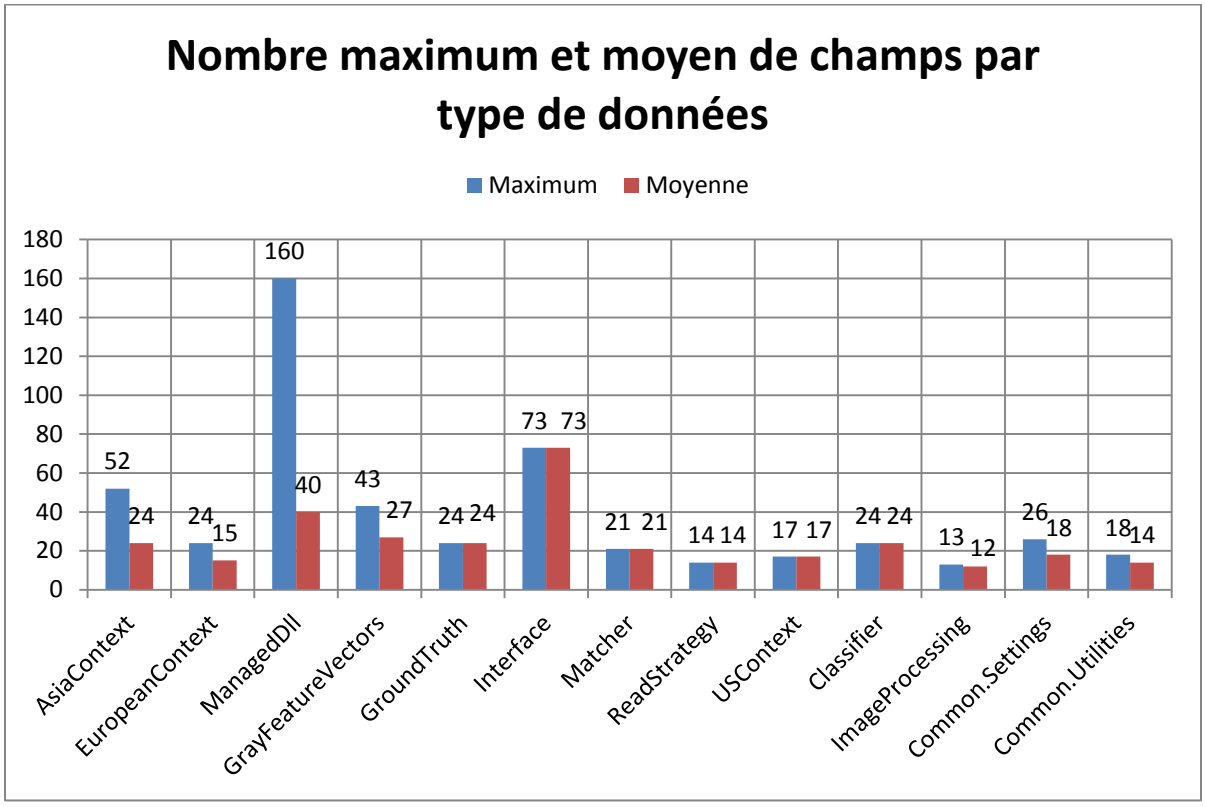

Figure 10 - Nombre de champs par type de données

Pour certains types, ces valeurs élevées s'expliquent par le fait que des interfaces graphiques sont présentes dans les assemblies. Mais pour le reste, ces valeurs sont difficilement justifiables.

La Figure 11 présente également des résultats sans équivoques : le nombre de méthodes est très élevé. Dans les deux cas, cela nécessitera un travail important de réingénierie si on souhaite améliorer la maintenabilité de ces types de données.

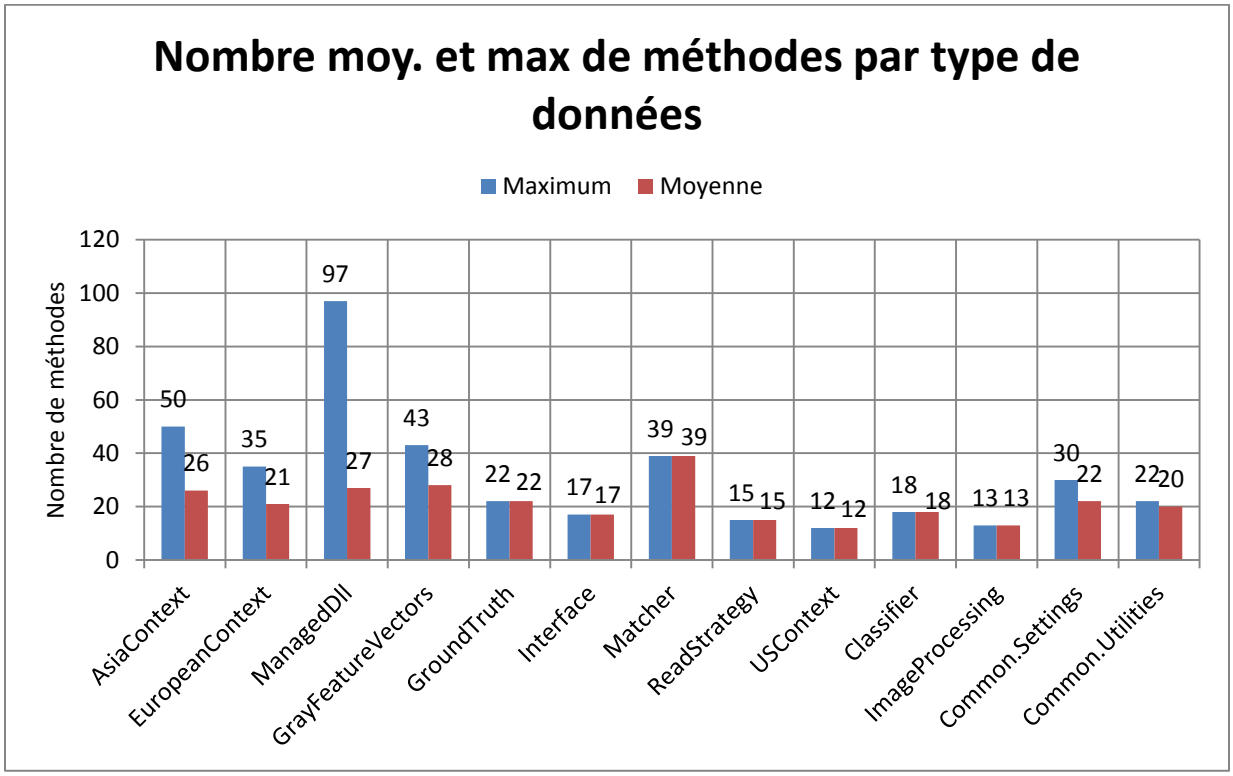

Figure 11 - Nombre de méthodes par type de données

2.2.4 Synthèse de l'état des types de données

À cette étape, nous savons que les assemblies pourront poser des problèmes lors de la maintenance de par leur architecture. On peut constater désormais que certains types de données (près de un sur cinq) ont une faible cohésion et un nombre de lignes passablement élevé. Concernant le nombre de lignes de code, il ne faut pas oublier qu'il s'agit ici de blocs d'instructions telles que perçus par le compilateur et, qu'en réalité, ce nombre sera de deux ou trois supérieur.

Compte tenu de ces résultats, il me parait indéniable que la facilité d'analyse et de modification de ces types sera très affectée : non seulement ManagedDll concentre beaucoup de responsabilités mais beaucoup de types de données se retrouvent dans la même situation.

2.3 Métriques des méthodes

Pour la dernière série de métriques, j'ai choisi d'utiliser les métriques suivantes pour les méthodes :

- Complexité cyclomatique;
- Nombre de paramètres et de variables.

2.3.1 Complexité cyclomatique

La complexité cyclomatique cherche à déterminer le nombre de décisions possible au sein d'un algorithme. Un seuil maximum de 10 a été fixé. La Figure 12 montre le pourcentage de méthodes par assembly ayant une complexité supérieure à 10. Près de 9 % des méthodes du module LPR sont problématiques, soit un peu plus de 350 méthodes. Encore une fois, la majorité des dépassements sont situés dans l'assembly ManagedDll.

L'impact immédiat de la violation de cette métrique est la difficulté à tester efficacement la méthode. Théoriquement, à chaque décision devrait correspondre un test unitaire pour couvrir la possibilité.

Voyons maintenant la répartition de ces dépassements présenté dans la Figure 13. Nous pouvons voir que certaines sont exagérément complexes. Un nombre important de méthodes paraissent difficilement analysable. De plus, on peut constater qu'un peu plus de 250 méthodes sont sur le point de dépasser le seuil de complexité de 10.

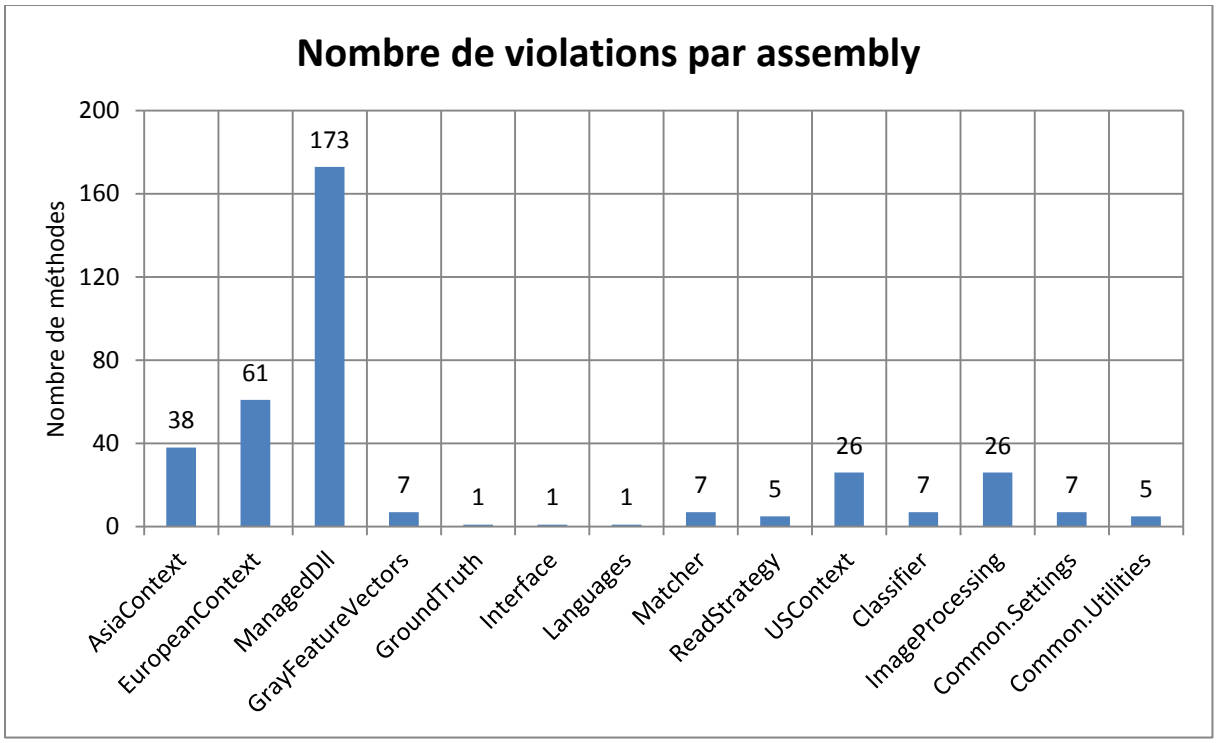

Figure 12 - Complexité cyclomatique des méthodes

2.3.2 Nombre de paramètres et nombre de variables

Toujours selon Robert Martin [8], une méthode ne devrait pas avoir plus de deux paramètres. Au-delà de trois, cela devient difficilement justifiable. Avoir un nombre élevé de paramètres indique un problème de définition de responsabilité : trop de paramètres signifient que la méthode fait certainement trop de choses. La Figure 14 présente le nombre de paramètres maximum et moyen d'une méthode lorsque celle-ci en a plus de 3, trié par assembly. Nous pouvons voir qu'il existe plusieurs méthodes avec un nombre élevé de paramètres, le maximum étant de 20.

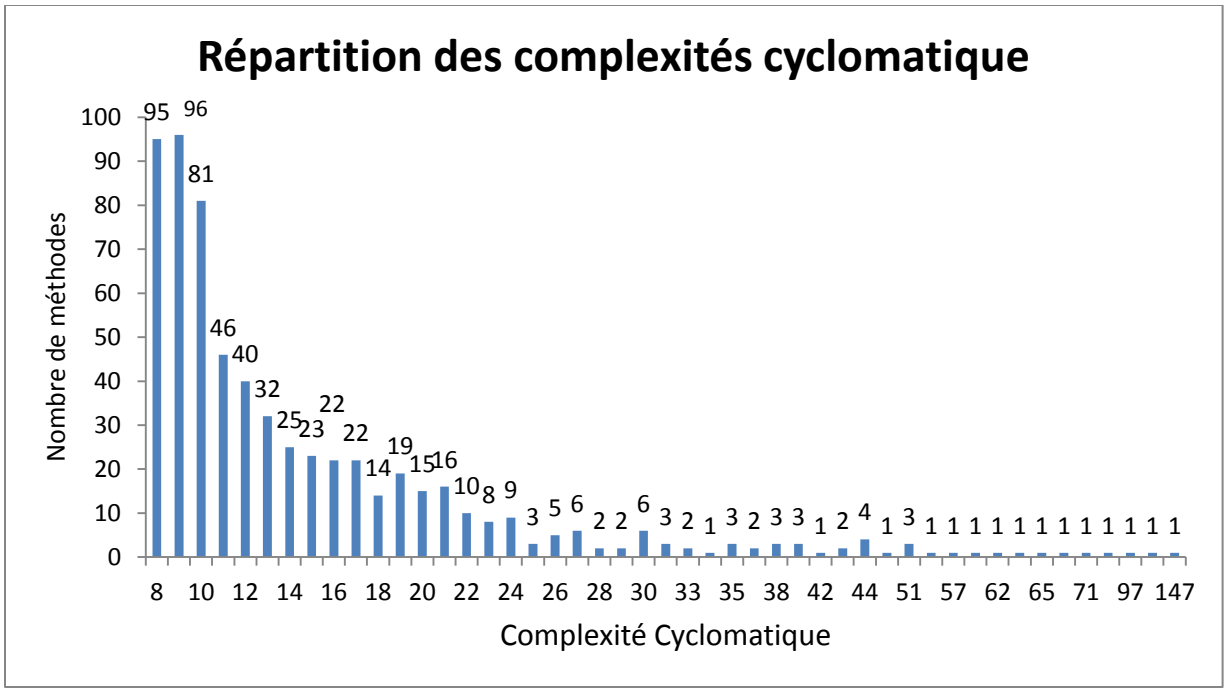

Figure 13 - Répartition des complexités cyclomatique

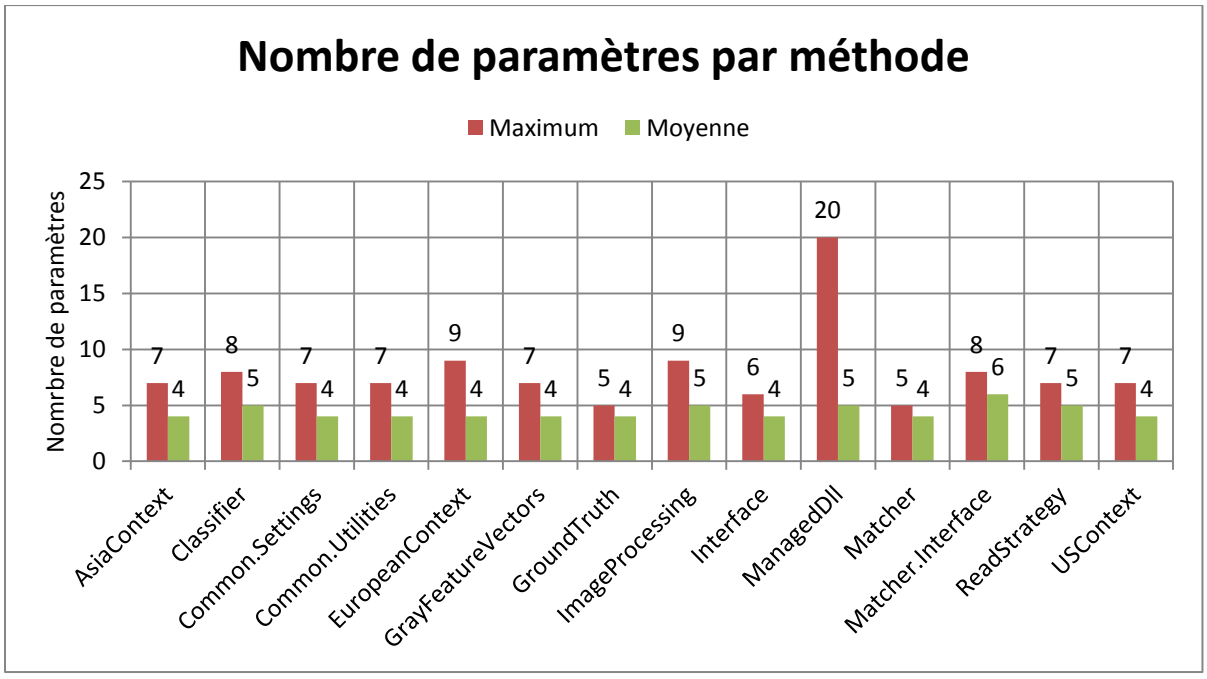

Figure 14 - Nombre de paramètres par méthode

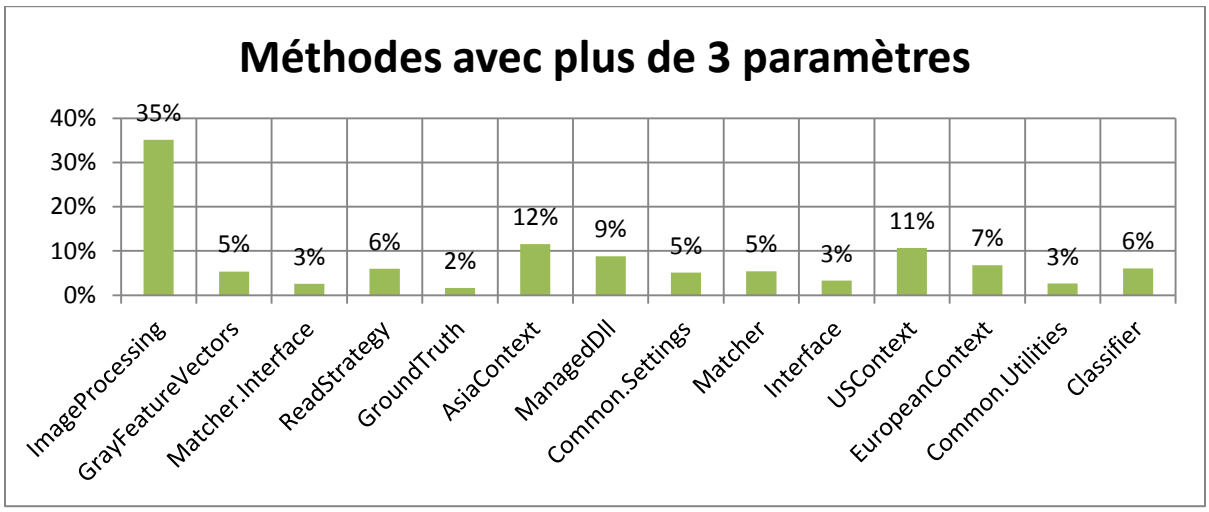

Figure 15 - Méthodes avec plus de 3 paramètres

Finalement, la Figure 17 présente le nombre moyen et maximum de variables par méthodes lorsque ce nombre dépasse le seuil de dix. Encore une fois, nous pouvons constater des valeurs maximales très élevées. Dans le cas de ImageProcessing, il s'agit d'une méthode sur quatre qui dépasse le seuil de dix variables (Figure 16) et une méthode sur trois qui possède plus de trois paramètres (Figure 15).

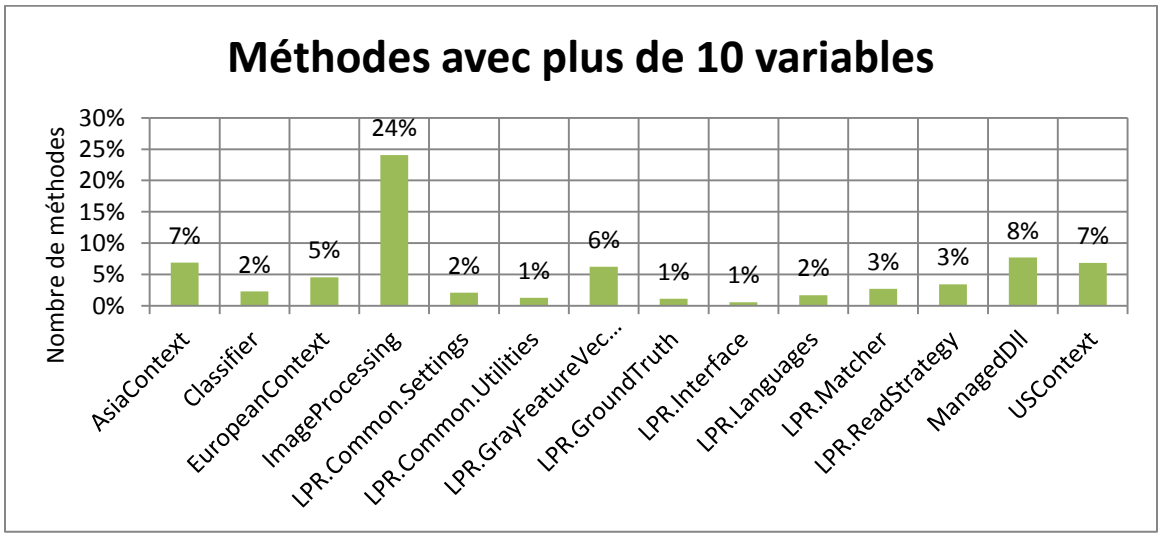

Figure 16 - Méthodes avec plus de 10 variables

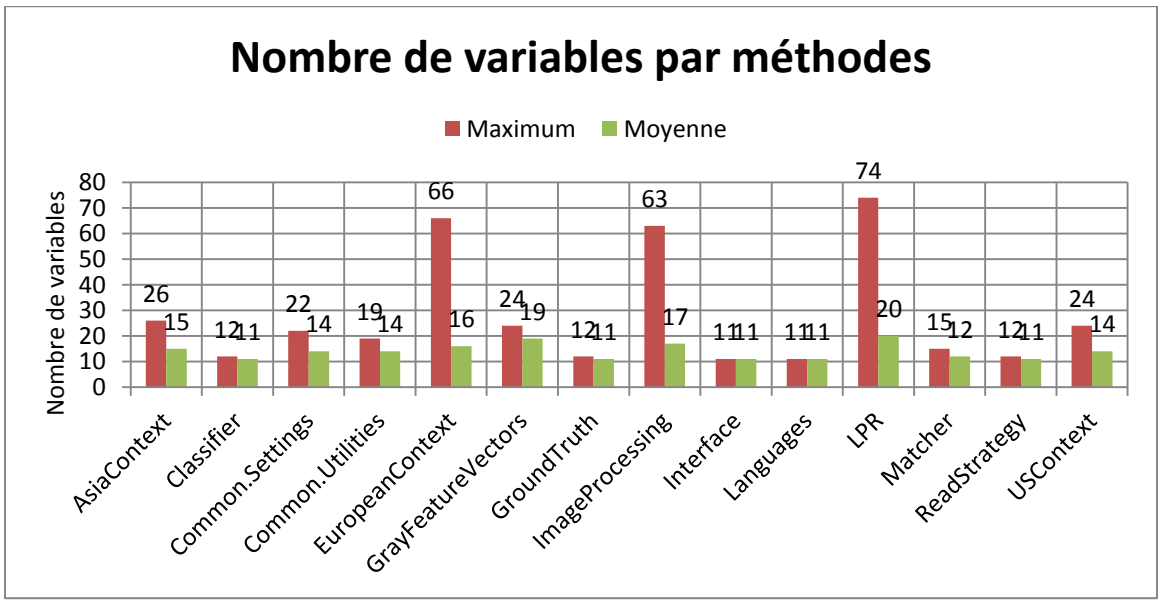

Figure 17 - Nombre de variables par méthodes

2.3.3 Synthèse de l'état des méthodes

On ne peut que constater que, même au plus bas niveau, des problèmes importants sont présents. Une grande partie des méthodes n'est ni facile à analyser, ni facile à modifier, ni facile à tester. De plus, la complexité de certaines méthodes rend leur vérification et leur validation par les tests unitaires et systèmes extrêmement difficiles, et ce, que ce soit la complexité cyclomatique, la quantité de paramètres ou de variables.

CONCLUSION

Le module LPR est un module atypique chez Genetec. Il s'agit du seul module qui est développé par une équipe qui exige que ses membres aient des compétences pointues dans un domaine précis : la reconnaissance d'image. Ce n'est pas n'importe quel développeur qui pourra se plonger dans ces modules aussi aisément que d'autres modules. Malgré ça, les métriques indiquent tous un problème récurrent à tous les niveaux de l'analyse : les assemblies, les types de données et les méthodes concentrent trop de responsabilités.

Tout d'abord, il n'est pas facile à analyser. Près d'un type de données sur cinq présente une faible cohésion et un nombre de lignes de code très élevé. À cela, il faut prendre en compte de nombreuses méthodes qui possèdent soit un arbre décisionnel extrêmement complexe, soit un nombre de paramètres et / ou de variables élevé.

Il n'est pas facile à modifier. Un assembly rigide concentre près de la moitié du code du module et est bâti sur des fondations instables. Et, là encore, la grosseur de certains types de données et des méthodes ont un impact défavorable sur la capacité du module à être modifier. La couverture du code par les tests est présente mais reste insuffisante pour des activités de réingénierie.

En conséquence, les risques de complications que l'on peut rencontrer lors d'une intervention de maintenance sont élevés : il peut être difficile dans certaines situations de déterminer efficacement l'impact des changements. Cela a donc un effet négatif sur la stabilité du module. Finalement, la facilité de tester certains types de données et méthodes est également problématique à cause du manque de cohésion et de leur complexité.

En conclusion. Si on souhaite que la maintenance du module LPR se fasse de manière efficace, nous n'aurons pas d'autres choix que d'investir d'importants efforts dans la correction des problèmes mentionnés afin de pouvoir répondre à ce besoin.

BIBLIOGRAPHIE

[1] R. Lincke et W. Löwe, «Compendium of Software Quality Standards and Metrics,» 4 April 2007. [En ligne]. Available: www.arisa.se/compendium/quality-metrics-compendium.pdf. [Accès le 28 02 2012].

[2] K. Cwalina et B. Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, Addison-Wesley Professional (2ème édition), 2008.

[3] R. C. Martin, «Object Mentor - Published Articles,» 02 1997. [En ligne]. Available: http://www.objectmentor.com/resources/articles/stability.pdf. [Accès le 03 2012].

[4] R. C. Martin, «Object Mentor - OO Design Quality Metrics,» 1994. [En ligne]. Available: http://www.objectmentor.com/resources/articles/oodmetrc.pdf. [Accès le 03 2012].

[5] R. C. Martin et M. Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall; 1ère édition , 2006.

[6] SMACCHIA.COM, «NDepend Metrics,» [En ligne]. Available: http://ndepend.com/Metrics.aspx#RelationalCohesion.

[7] S. Celarier, 2007. [En ligne]. Available: http://www.hanselman.com/blog/content/binary/NDepend%20metrics%20placemats%201.1.pdf.

[8] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008.