28/10/2013

ACTIVE UP

MÉTRIQUES ET CRITÈRES D'ÉVALUATION DE LA QUALITÉ DU CODE SOURCE D'UN LOGICIEL

Revue d'un professionnel de l'industrie du logiciel | Pierre Mengal

Table des matières

1. Introduction 3
2. Métriques standards 4
   2.1. Complexité cyclomatique 4
   2.2. SLOC (Source Lines Of Code) 5
   2.3. Densité des commentaires 8
   2.4. Couverture de code 9
   2.5. Duplication de code 10
   2.6. Afferent & efferent coupling 11
   2.7. Instability 13
   2.8. Abstractness 14
   2.9. Distance from main sequence 14
   2.10. Lack of Cohesion Of Methods (LCOM) 15
   2.11. Relational Cohesion 16
   2.12. Size of instance 16
   2.13. Specialization index 17
   2.14. Number of parameters 17
   2.15. Number of variables 18
   2.16. Number of overloads 18
   2.17. Number of coding rules violations 18
3. Critères supplémentaires 19
   3.1. Architecture 19
      3.1.1. Maintenabilité 20
      3.1.2. Evolutivité 20
      3.1.3. Performance 20
      3.1.4. Pertinence 22
   3.2. Style & lisibilité 22
   3.3. Documentation 23
   3.4. Fiabilité 23
   3.5. Portabilité 23

3.6. Sécurité 24
3.7. Nombre de bugs 24
4. Tableau récapitulatif 25
5. Références bibliographiques 27

1. Introduction

Avant de commencer à décrire les différents métriques et critères d'évaluation de « qualité du code », il y a lieu d'en expliquer la différence avec le concept de « qualité logicielle ». En effet, la qualité du code, aussi appelée *qualité structurelle* du logiciel, est différente de la qualité d'un logiciel, aussi appelée *qualité fonctionnelle* d'un logiciel, dans le sens où la première concerne la manière avec laquelle une fonctionnalité est implémentée et la seconde le résultat final. Dans son ouvrage, « *Metrics and Models in Software Quality Engineering* », Stephen H. Kan (2003) définit la qualité fonctionnelle d'un logiciel comme la "conformité aux exigences". On pourrait dire qu'un logiciel est de qualité s'il a un nombre faible de bogues fonctionnels. La qualité structurelle d'un logiciel est donc la manière avec laquelle ces fonctionnalités sont implémentées et inclut entre autres la robustesse, la maintenabilité, la lisibilité ou encore l'évolutivité. Il existe probablement un lien entre les deux, mais pas nécessairement de causalité. Un logiciel avec un code de très mauvaise qualité pourrait tout à fait être jugé de très haute qualité par les utilisateurs. Ou à l'inverse, le résultat final d'un code d'une qualité extrêmement haute pourrait être jugé par les utilisateurs comme étant de mauvaise qualité.

C'est la mesure de la qualité structurelle du code qui nous intéresse dans le présent document. Elle peut en partie être mesurée automatiquement par des outils d'analyse statique du code source. Ces outils sont disponibles pour la plupart des langages et proposent de nombreuses métriques[1] et conventions standards dont les principales seront décrites dans le présent document. La qualité du code peut être également évaluée par d'autres critères comme l'architecture qui inclut la maintenabilité, l'évolutivité, la performance et la pertinence mais aussi la documentation, la portabilité et la sécurité. D'autres aspects peuvent être mesurés en partie automatiquement comme la fiabilité et le nombre de bugs connus. Tous ces critères sont fortement dépendants du contexte comme la spécificité du logiciel au niveau fonctionnel, son avenir mais aussi d'autres facteurs importants comme le marché et ses utilisateurs.

Bien que ces métriques obtenues automatiquement soient qualifiées d'imparfaites par certains de leurs auteurs et que se baser uniquement sur elles pour évaluer la qualité du code serait d'après eux « téméraire » (Martin 2000, page 27), leur usage pourrait avoir une certaine validité prédictive. Et ce plus particulièrement dans l'évaluation ponctuelle d'un code source à des fins diagnostiques. Ainsi, **un mauvais rapport pourrait être considéré comme un signal d'alarme qui devrait inciter son lecteur à aller plus loin dans sa démarche**. Leur apport est donc intéressant mais très limité dans cette situation en particulier car ces métriques ne couvrent que certains aspects de la qualité du code.

Par contre, dans un environnement de développement moderne, le contrôle de ces métriques peut être un moyen efficace pour les développeurs d'obtenir des informations utiles en continu. On pourrait ajouter le contrôle de ces valeurs dans un cycle d'intégration continue. L'intégration continue consiste à vérifier que chaque fonctionnalité implémentée ne fait pas régresser le code (introduction de bugs). **Un problème avec cette approche est que les développeurs ainsi mesurés pourraient adopter un comportement contreproductif en tentant d'éviter les mauvaises valeurs**. L'équipe de développement

[1] Une métrique est un moyen d'associer une valeur à un attribut d'une entité

pourrait donc contribuer à augmenter cette qualité au détriment d'autres variables importantes pour le business comme le temps nécessaire pour compléter le projet, les ressources nécessaires et les fonctionnalités implémentées. De ce fait, du code d'une qualité extrême pourrait témoigner de pratiques de management inadaptées. Certains critères sont plus importants que d'autres et il est fortement conseillé de pondérer les résultats. Un tableau comparatif est disponible à la fin de ce document.

2. Métriques standards

2.1. Complexité cyclomatique

La **complexité cyclomatique** (Cyclomatic complexity) , dans sa version la plus stricte, est la mesure du nombre de chemins linéaires possibles dans une fonction (McCabe 1976). Beaucoup de logiciels d'analyse statique du code sont incapables d'obtenir la valeur réelle de cette mesure. Le calcul est alors effectué sur le nombre de fois que certaines expressions sont rencontrées (ex: if | while | for | foreach).

Cette mesure peut s'effectuer à plusieurs niveaux, par exemple au niveau des méthodes ou du programme tout entier. Ce dernier cas n'a pas vraiment d'utilité puisque la valeur dépendra de la complexité générale du programme. Par contre, ces valeurs, quand elles sont évaluées au niveau des méthodes ou fonctions, peuvent s'avérer utiles pour identifier les endroits qui nécessitent du code refactoring[2]. Une bonne méthode ne devrait avoir qu'une seule fonction.

Une méthode trop complexe ou trop longue poserait les problèmes suivants:

- **Manque de lisibilité du code**: il est plus difficile pour un développeur externe de comprendre le fonctionnement du code. Ce problème pourra avoir un impact considérable sur la maintenabilité (voir Dette Technique).
- **Perte d'informations pendant le débogage**: moins les méthodes sont décomposées en sous-méthodes, moins riches seront les messages d'erreurs (stack trace moins précis). Cela ralentit le processus d'identification des problèmes.
- **Tests unitaires moins efficaces**: tester une méthode très complexe est souvent inefficace car un bon test unitaire doit pouvoir couvrir la plupart des cas de figure (chemins possibles).

Un code source qui présenterait trop de méthodes avec une complexité cyclomatique importante verrait sa valeur sensiblement réduite étant donné sa maintenabilité moindre.

Une autre façon de mesurer la complexité du code est d'utiliser les métriques d'Halstead (1977) qui base également ses calculs sur les opérateurs et expressions. Pour une revue en français des deux métriques, voir Lambertz (2007). D'après Watson & McCabe (1996), la complexité cyclomatique d'une fonction ne devrait pas dépasser 10.

[2] Consiste à retravailler le code source d'un code informatique sans ajouter de fonctionnalité au logiciel ni corriger de bogues, mais en améliorant sa lisibilité pour simplifier sa maintenance, ou le rendre plus générique (par exemple afin de faciliter le passage de simple en multiple précision); on parle aussi de remaniement (Wikipédia)

Actions possibles

Utiliser la technique de refactoring appelée **extraction de méthodes** et s'assurer que chaque méthode ne remplit qu'une seule et unique fonction.

Faire un contrôle du niveau de complexité cyclomatique en continu pendant le développement permet d'obtenir une meilleure conception du code.

2.2. SLOC (Source Lines Of Code)

Il s'agit de la mesure de la taille d'une partie ou de la totalité d'un logiciel en se basant sur le nombre de lignes de code. On différencie principalement deux manières de calculer : la mesure « physique » et la mesure « logique ». La première se base sur le nombre de lignes présentes dans le fichier sans autre traitement. La seconde ne prend que les lignes qui seront effectivement exécutées, ce qui permet d'éliminer l'impact du style différent d'un programmeur à l'autre.

Il n'y a actuellement aucun consensus sur la méthode de calcul de cette valeur. Il existe malgré tout une classification proposée par Boehm (1984) qui est toujours utilisée aujourd'hui mais qui est à interpréter avec beaucoup de précautions.

Classification	**Size (KLOC)**
Small (S)	2
Intermediate (I)	8
Medium (M)	32
Large (L)	128
Very Large (VL)	512

L'estimation de la production ou valeur d'un logiciel sur base du nombre de lignes de code est aussi populaire que controversée. La principale critique est qu'il y a beaucoup trop de facteurs influençant la valeur finale de la mesure. Robert E. Park (1992, page 140), fervent défenseur de la méthode, répondait aux critiques avec l'exemple suivant :

"When we hear criticism of SLOC as a software measure, we are reminded of a perhaps apocryphal story about a ditch digger who, when asked one day how he was doing, replied, "Dug seventeen feet of ditch today." He didn't bother to say how wide or how deep, how rocky or impenetrable the soil, how obstructed it was with roots, or even how limited he was by the tools he was using. Yet his answer conveyed information to his questioner. It conveyed even more information, we suspect, to his boss, for it gave him a firm measure to use as a basis for estimating time and cost and time to completion."

A l'origine, cette technique aurait probablement pu être utilisée dans les conditions évoquées par l'auteur. Des modèles comme CoCoMo (Boehm 1981) permettaient d'ailleurs de tenir compte d'un certain nombre de paramètres dont la variabilité était sans doute raisonnable à l'époque. Mais depuis, le nombre de facteurs impactant le nombre de lignes de code est devenu si important qu'il n'est que très peu recommandé de prendre cette mesure au sérieux tant au niveau de l'évaluation d'un logiciel que de la productivité de l'équipe qui l'a conçu. Je vais tenter d'illustrer la problématique en proposant 8 arguments.

1. Les différences entre les langages et les frameworks

Il existe aujourd'hui des centaines de langages différents (Wikipedia 2013). Pour chacun de ces langages, plusieurs frameworks sont proposés. Pour un nombre de fonctionnalité égal, on peut avoir un nombre de lignes de code très différent en fonction des choix technologiques. De plus, les architectures modernes utilisent des technologies différentes, ce qui complique encore les choses au niveau des calculs. Des facteurs de correction existent mais ils semblent peut défendables étant donnée la grande diversité de types d'applications qui sont développées aujourd'hui.

2. L'expérience et la compétence des développeurs

Il faut également tenir compte de l'expérience des développeurs concernés car cela peut influer sur le calcul à de nombreux égards. Quelqu'un de très compétent écrira souvent moins de lignes de code qu'une autre car il emploiera des méthodes de designs créées dans l'unique but d'en réduire la quantité et ainsi augmenter la lisibilité et la maintenabilité. De plus, ils exploitent beaucoup mieux les fonctionnalités proposées par les outils utilisés (technology stack). En effet, par méconnaissance de ces derniers, beaucoup de programmeurs réécrivent des fonctionnalités existantes, ce qui augmente considérablement le nombre de lignes de code. A ce propos, beaucoup d'experts reconnus dans le secteur n'hésitent pas à parler de « lignes de code dépensées » en opposition aux « lignes de code produites » (Dijkstra 1983).

3. Les pratiques de refactoring

Le fait qu'une même partie de code puisse évoluer dans le temps avec le refactoring (réusinage de code) peut biaiser les résultats. Cette pratique consiste à retravailler le code source sans lui ajouter de fonctionnalité (Wikipedia 2013) et elle devient de plus en plus courante car elle permet d'augmenter la qualité du code et réduire la dette technique. Cela peut provoquer des situations inattendues : si des développeurs pratiquent beaucoup cette technique dans un intervalle de mesure donnée, le résultat pourrait laisser percevoir une régression (moins de lignes de code que lors de la précédente mesure) alors que c'est clairement l'inverse qui se produit.

4. Les pratiques de réutilisation et/ou génération de code

La réutilisation de code existant est très fréquente et hautement recommandée sous le principe DRY (Don't Repeat Yourself). Ainsi, de nombreuses parties du code peuvent avoir été récupérées d'un projet précédent ou copiées depuis un projet open source, librairie ou autre blog post. En outre, les outils modernes de développement peuvent générer automatiquement du code pour le développeur à travers différents outils de conception de haut niveau.

5. Les tâches hors développement

Les activités dans le cadre du développement d'un logiciel ne se limitent pas à écrire du code sur un clavier. En fait, beaucoup d'autres tâches sont nécessaires pour produire du code de qualité. Ici, une variabilité très importante peut émerger en fonction des différentes méthodes utilisées, la composition de l'équipe ou les exigences en matière de documentation.

6. La fiabilité de l'outil de mesure

Une grande variété d'outils de mesure est proposée sur le marché. Etant donné le manque de consensus sur la méthode de calcul de la quantité de lignes de code dans un fichier source, le résultat peut être sensiblement différent. De plus, certains problèmes techniques peuvent survenir quand il s'agit d'identifier ce qui doit effectivement être comptabilisé ou pas. Par exemple, certains logiciels éprouvent des difficultés à différencier les commentaires des instructions quand ceux-ci sont mélangés (Danial 2013). L'efficacité et la qualité de ces logiciels est elle aussi très variable.

7. Les (potentielles) manipulations

Quand une mesure peut avoir un impact sur une ou plusieurs personnes, il faut pouvoir envisager la possibilité que certains d'entre elles tentent de les manipuler à leur avantage. Ainsi, si la productivité d'un développeur est mesurée sur base du nombre de lignes de code (ou même fonctions), il pourrait très facilement manipuler le code source pour gonfler les résultats. Ce problème est très courant dans les entreprises qui se servent de KPI (Key Performance Indicator) pour effectuer les évaluations de leurs employés. On peut aussi très bien imaginer qu'une entreprise tente de valoriser au mieux son actif si elle sait qu'il sera évalué sur base de cette métrique.

8. Le temps

Quasiment tous les éléments précités sont sensibles au temps. Par exemple, le niveau du développeur évolue forcément avec la pratique (ceci inclut la fameuse learning curve). De plus, les fonctionnalités des langages et frameworks évoluent eux aussi afin d'augmenter la productivité de ces derniers. Plus un projet dure longtemps, plus la mesure sera sensible à ce biais.

En conclusion, on peut dire qu'estimer l'effort de production ou la valeur d'un logiciel à l'aide de cette métrique est très risquée. Cette technique reste cependant très utilisée. Certains spécialistes de l'estimation comme Steve McConnell (2006) sont très conscients de son inefficacité mais proposent de l'utiliser quand même faute de mieux. D'autres méthodes basées sur les « function point » (fonctionnalité business) ont tentés de résoudre certains des problèmes adressés ci-dessus mais les valeurs obtenues restaient fortement corrélées avec le nombre de lignes de code (Albrecht 1983). Les informations obtenues par ces métriques et celles qui en découlent, ne peuvent en aucun cas être considérées comme fiables et doivent être utilisées avec beaucoup de prudence dans vos processus de prise de décision.

2.3. Densité des commentaires

En fonction de la formule de calcul employée, il s'agit du rapport entre le nombre de lignes de code (SLOC) et le nombre de commentaires (CLOC). La formule la plus simple étant DC = CLOC / LOC.

La qualité du code source dépend souvent de la manière avec laquelle il a été commenté. L'absence partielle ou totale de commentaires pourrait être un signe de mauvaise qualité. Le problème se pose quand il s'agit de mesurer cette qualité car cette métrique dépend directement du nombre de lignes de code et par conséquent doit être considérée avec prudence. En effet, la plupart des problèmes liés à la mesure du nombre de lignes de code s'appliquent aux commentaires.

Premièrement, le nombre de commentaires nécessaires dépend grandement du type de langage et de framework utilisé. D'un langage à l'autre, il y aura plus ou moins de lignes de code pour la même fonctionnalité ou traitement. Le choix technologique va donc impacter grandement sur le poids d'un commentaire dans le calcul.

Deuxièmement, la qualité du texte rédigé par le développeur va également jouer sur le nombre de lignes nécessaire pour rendre compréhensible le même bloc de code. Certains programmeurs sont très méticuleux quand il s'agit de commenter leur code quand d'autres vont bâcler le travail au point de rendre le commentaire incompréhensible.

Troisièmement, le code généré n'inclut pas souvent les commentaires, ce qui fait baisser la valeur générale. De plus, il existe de nombreux outils qui commentent le code automatiquement en fonction de différents paramètres contextuels comme le nom des variables et les références. Il y a lieu de vérifier que ce code généré soit bien exclut des analyses.

Quatrièmement, la nécessité de commentaire dépend directement de la clarté du code. Un excellent programmeur écrira du code expressif. C'est-à-dire que la façon de le structurer et de nommer ses fonctions ou variables sera auto-explicative. C'est d'ailleurs cette approche qui est généralement recommandée par les experts du secteur (Subramaniam, 2006). Etant donné que le code expressif permet de décrire ce qu'il fait, les commentaires efficaces doivent se contenter d'expliquer pourquoi uniquement quand c'est nécessaire (Hunt & Thomas, 1999). Plus le développeur devient efficace dans cette tâche, plus la densité baisse, ce qui est en contradiction avec l'évolution attendue de cette métrique.

Pour toutes ces raisons, la quantité des commentaires ne peut en aucun cas être un critère objectif de la qualité du code. Il peut néanmoins être un indice de mauvaise qualité quand la densité est beaucoup trop faible. La meilleure manière de mesurer la qualité objective du code est de le faire lire par un autre développeur pour ensuite en évaluer son degré de compréhension.

Des éditeurs de logiciels d'analyse statique de code (RefactorIt & nDepend) recommandent généralement de se situer entre 20% et 40% selon diverses sources comme. Il est fortement recommandé de contrôler tout ce qui est en dessous de 20% et instaurer du code reviewing systématique.

Actions possibles

Pour les raisons évoquées ci-dessus, il n'est pas recommandé d'imposer un tel niveau de commentaire mais d'instaurer un système de contrôle lors des revues de code. Il serait également intéressant de former les développeurs à écrire de meilleurs commentaires. Le document « The fine Art of Commenting » de Bernhard Spuida (2002) est un bon début.

2.4. Couverture de code

La couverture de code (code coverage) représente la proportion du code source qui est couverte par des tests, généralement automatisés. Le calcul est exprimé en pourcentage et peut être obtenu sur base du nombre d'instructions, de méthodes ou de classes parcourues. Cela inclut les tests unitaires (unit tests) mais aussi les tests fonctionnels et de validation. Ces derniers sont plus difficiles à automatiser.

Un taux de couverture élevé par des tests unitaires présente de nombreux avantages (Wikipedia : Unit Testing, 2013). Trop souvent négligés car considérés comme superflus, ces tests s'avèrent pourtant essentiels pour maintenir le code dans un état de maintenabilité acceptable et limiter l'évolution de la dette technique. De plus, la mise en œuvre de ceux-ci induit un meilleur design du code, ce qui contribue également à la qualité du code.

Tout d'abord, ils permettent d'éviter l'introduction de nouveaux bugs en plus de confirmer le bon fonctionnement du code existant. Plus le code sera protégé par des tests, moins il sera possible d'introduire de nouveaux bugs en le modifiant. Ces tests peuvent être exécutés à chaque intégration car ils sont automatisés et rapides (idéalement). C'est ce qu'on appelle les tests de non-régression. Le code source qui possède un nombre important de tests sera donc beaucoup plus facile à maintenir et à faire évoluer.

De plus, les tests s'avèrent être une documentation technique très intéressante. La lecture des tests d'une classe permet au développeur de voir comment elle doit être utilisée en pratique. Si les tests sont correctement rédigés, ils comportent à la fois les bonnes et les mauvaises pratiques. L'autre avantage est que cette documentation évolue en même temps que le code.

Enfin, ils permettent d'améliorer le design du code principalement de deux manières. La première est liée au fait que le développeur est poussé à découper son code en plus petites unités testables. Cela permet de réduire grandement la complexité du code. De la même manière, il va souhaiter donner une seule responsabilité à une classe comme c'est généralement conseillé. La deuxième est liée à une méthode de développement appelée Test Driven Development. Cette technique propose d'écrire le test avant d'écrire la fonctionnalité. Une fois qu'il est créé, le développeur doit écrire le code le plus simple possible pour faire passer le test au vert. Le code étant couvert à 100%, l'étape suivante est le refactoring.

En conclusion, les tests unitaires contribuent fortement à augmenter la qualité du code source. La présence de ceux-ci dans le code source est un bon indicateur de qualité. Il faut cependant tenir compte

du fait qu'il est possible, comme pour le reste, d'écrire de très mauvais tests unitaires. Certains développeurs pourraient être également tentés d'augmenter la couverture des tests en le faisant intentionnellement. Les outils d'analyse statique doivent être impérativement accompagnés d'une analyse qualitative effectué par un développeur expérimenté.

La valeur souhaitée est de 100% (Cornett, 2011) mais 80% est généralement considéré comme acceptable. Certains spécialistes des tests unitaires déclarent même que 100% n'est pas suffisant (Grenning 2012). D'autres auteurs estiment qu'il n'est pas nécessaire de mesurer la couverture du code par les tests et proposent plutôt de se focaliser sur ce qui est utile (Savoia 2004). La dernière proposition devrait être privilégiée. Les tests unitaires s'appliquent presqu'à tous les langages utilisés de nos jours (Wikipedia : List of unit testing frameworks, 2013).

Actions possibles

Consulter les bonnes pratiques (Kolawa, 2009). S'il faut ajouter des tests unitaires sur du code existant, il est conseillé de consacrer 10% du temps de développement à cette tâche jusqu'à atteindre le niveau de couverture souhaité (qu'il faudra maintenir). L'utilisation d'un serveur d'intégration continue pour contrôler ce niveau est également conseillée.

2.5. Duplication de code

La duplication de code (ou code clone) consiste en la répétition d'instructions similaires ou identiques dans un code source. Toute duplication du code viole le principe « DRY » (Don't Repeat Yourself) formulé pour la première fois par Hunt & Thomas dans l'ouvrage « The Pragmatic Programmer » (1999). Il est explicité par la phrase suivante : « Dans un système, toute connaissance doit avoir une représentation unique, non-ambiguë, faisant autorité ». Les auteurs suggèrent que la duplication de code tombe généralement dans quatre catégories.

1. **La duplication imposée**. Il arrive parfois que les exigences fonctionnelles imposent la duplication. Par exemple au niveau de la représentation d'une même information qui peut être différente à certains égards dans certaines parties du système. Les limitations de certains langages peuvent également contribuer à la duplication de code (ex : avec CORBA).

2. **La duplication par inadvertance**. Cela peut se produire en faisant des erreurs de design.

3. **La duplication par impatience**. Sous la pression des délais, les développeurs peuvent être tentés de dupliquer le code pour aller plus vite. Certains outils d'insertion de bouts de code préenregistrés favorisent d'ailleurs ce comportement.

4. **La duplication interdéveloppeur**. Des développeurs différents travaillant sur le même code source implémentent des fonctionnalités similaires sans s'en rendre compte.

Le code dupliqué réduit incontestablement la maintenabilité du projet et augmente le risque d'introduction de bugs. Si les outils de détection de code dupliqué permettent d'identifier les erreurs les plus grossières, ils sont limités à plusieurs niveaux.

Tout d'abord, il peut s'agir de parties de code dupliqué dans plusieurs endroits distincts du projet mais aussi des répétitions d'instructions placées dans un même endroit. Ce dernier cas est très courant chez les développeurs débutants. Il est moins fréquent chez les développeurs compétents car ils exploitent mieux les possibilités des langages orientés objet et les design patterns (patron de conception). Le pattern « Strategy » est un bon exemple de solution pour réduire, par exemple, le nombre d'instruction « if – else ». Les détecteurs de duplication de code ignorent généralement ce type de duplication car il engendrerait beaucoup trop de faux positifs. De plus, les duplications interdéveloppeur sont difficilement détectables de par le fait que le code produit peut répondre identiquement aux besoins fonctionnels mais avec un code très différent. Enfin, la mesure est bien inutile dans le cas de duplication imposée.

La détection de code dupliqué reste néanmoins un bon moyen de détection de mauvaise qualité de code ainsi que des processus de développement peu efficaces.

Les outils mesurent le nombre de portions de code dupliquées ainsi que le nombre de lignes de code dupliquées (ou clonées). Les techniques sont très variées et paramétrables. Il existe des centaines de méthodes et d'outils de détection différents (Tairas 2013). Il y a lieu d'atteindre un pourcentage de code dupliqué égal à zéro.

Actions possibles

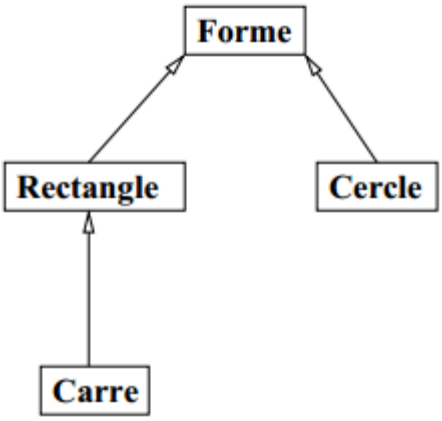
La détection du code dupliqué est donc un bon moyen d'augmenter la qualité du code en permettant d'identifier les parties à retravailler. L'utilisation d'un outil de détection de code dupliqué est donc requise dans tous les projets de développement professionnels.

La duplication du code est souvent due à un manque d'abstraction. Retravailler ces aspects permettra souvent de diminuer la duplication en augmentant la réutilisation.

2.6. Afferent & efferent coupling

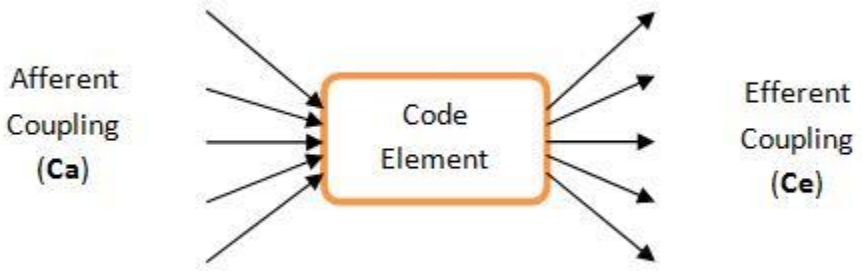
L'Afferent coupling représente le nombre de références vers la classe mesurée. Ces références doivent être externes à cette classe. Les références internes ne comptent pas. Cette métrique donne une bonne indication de l'importance de la classe dans le code.

L'Efferent coupling est la mesure du nombre de types que la classe « connait ». Cela comprend: l'héritage, l'implémentation d'interfaces, les types des paramètres, les types de variable, les exceptions levées et capturées. En bref, tous les types mentionnés dans le code source de la classe mesurée.



Prenons un exemple avec le principe de l'héritage en orienté objet. La classe **Carre** référence la classe **Rectangle** qui elle référence la classe **Forme**. La classe **Cercle** quant à elle référence la classe Forme également.

Dans cet exemple, l'Afferent coupling de **Forme** est de 2 puisque 2 autres classes la référencent directement. Il est de 1 pour **Rectangle**, mais de 0 pour **Carre** et **Cercle**. L'Efferent coupling est de 0 pour Forme car il ne référence aucune autre classe. Il est donc de 1 pour toutes les autres classes du schéma. Le schéma ci-dessous résume le principe du couplage.



Le couplage est effectif pour tout appel de méthode, accès à une propriété ou champs, héritage, mention dans des arguments ou type de retour de méthodes, ainsi que les exceptions. Il est à noter que certains logiciels d'analyse de qualité du code utilisent une formule différente (Codenforcer 2013).

Plus une classe est référencée (Afferent coupling), plus elle est importante dans le code source. Plus cette valeur est élevée, plus l'impact des modifications effectuées sur cette classe sera grand. Cela pourrait également indiquer une bonne réutilisation du code. Si on reprend l'exemple, toute modification de Forme impactera directement Rectangle et Cercle, et indirectement Carre.

Par contre, une valeur zéro peut également communiquer des informations intéressantes. Une valeur de zéro signifie que personne n'utilise cette classe et qu'il s'agit peut-être de code mort. Cependant dans certains cas, le code n'est vraiment référencé qu'à l'exécution, ce qui rend la détection impossible par

les outils de mesure statique. Un nombre important de classes avec une telle valeur pourrait être un signe de mauvaise gestion du code, ce qui peut potentiellement affecter la qualité du code.

De plus, le code mort utilise inutilement l'espace mémoire et pourrait contribuer à une réduction des performances de l'application.

Un Efferent coupling important indique quant à lui que la classe ne respecte pas le principe de responsabilité unique (Wikipedia : Single responsability principle). En outre, plus le nombre de dépendances augmente, plus le risque d'instabilité est élevé. RefactorIT propose de limiter ces références à 20.

Actions possibles

Vérifier que le code source est bien utilisé dans son entièreté et éliminer tout code superflu.

Séparer les responsabilités dans des classes différentes. Respecter la loi de Demeter ou « Law of Demeter » (Lieberherr & Holland 1989) peut également contribuer à réduire un trop fort couplage. Une étude de Basili & al. (1996) suggère que le respect de cette loi permet de réduire les bugs de manière significative. Il faut cependant vérifier que cette action ne soit pas exagérément appliquée car elle pourrait contribuer à réduire la lisibilité du code en le rendant plus complexe inutilement.

2.7. Instability

L'instabilité d'un module est définie par son niveau de résistance au changement. Plus un module est stable, moins il est facile de le changer. Elle est obtenue en calculant le rapport entre l'efferent coupling au couplage total. Cette valeur est donc obtenue en divisant l'efferent coupling à la somme de l'efferent coupling et l'afferent coupling.

Exemple de calcul de l'instabilité avec l'exemple cité dans la section de l'Afferent & efferent coupling :

Classe	Afferent Coupling (Ca)	Efferent Coupling (Ce)	Instability (Ce / (Ce + Ca))
Forme	2	0	0 / (0 + 2) = 0
Rectangle	1	1	1 / (1 + 1) = 0.5
Cercle	0	1	1 / (1 + 0) = 1
Carre	0	1	1 / (1 + 0) = 1

Ainsi Forme est très stable car le rapport entre ses dépendances efferantes sont nulles par rapport à ses dépendances afférentes. Le cas de Rectangle démontre une instabilité moyenne. Par contre Cercle et Carre sont très instables.

L'une des particularités d'un code source de qualité est qu'il est très aisé de le modifier (Martin 2000). Dans un contexte très compétitif où l'entreprise doit sans cesse intégrer de nouvelles fonctionnalités, c'est un atout majeur. Il est donc intéressant d'avoir un certain niveau d'instabilité dans le code pour permettre ces changements. Les classes doivent être soit très stables (entre 0,0 à 0,3) soit très instables (0,7 à 1,0). Dans le cas des classes très stables, il est recommandé d'avoir un niveau d'abstraction élevé (voir Abstractness).

Actions possibles

Séparer les responsabilités dans des classes différentes. Il est également possible d'avoir un nombre important de modules stables s'ils sont conçus pour être facilement extensibles (Martin 2000, page 25).

2.8. Abstractness

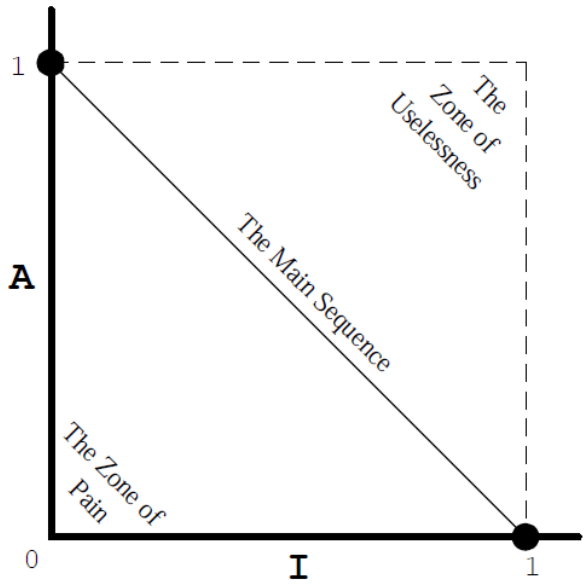
L'abstractness d'un module indique son niveau d'abstraction par rapport aux autres classes présentes dans le code. On obtient cette valeur en calculant le rapport entre les types abstraits internes (classes abstraites & interfaces) et les autres types internes. La valeur se situe entre 0 et 1, 0 indiquant un module complètement concret et 1 un module complètement abstrait.

Par exemple une classe abstraite en Java ou en C# est une classe comme les autres à une différence près : elle ne peut pas être instanciée. Elle est donc utilisée principalement comme classe de base dans l'héritage. Par exemple la classe « Chien » hérite de la classe « Animal ». On peut instancier un « Chien », mais on veut empêcher d'instancier un objet « Animal ». Pour ce faire, on déclare « Animal » comme classe abstraite. Une interface peut être comparée à une classe mais dont on a gardé que les signatures de ses membres (méthodes & propriétés par exemple). On n'hérite pas d'une interface, mais on l'implémente. C'est-à-dire que la classe qui utilise une interface donnée doit fournir le code pour les membres présents dans cette dernière.

Cette métrique, à elle seule, n'est pas utile. Combinée avec la métrique **Instability**, elle permet de calculer la **Distance from main sequence** (section suivante). Elle dépend de la mesure d'instabilité.

2.9. Distance from main sequence

La distance from main sequence indique l'équilibre du module entre l'abstractness et l'instability. Cette valeur s'obtient avec la formule $D' = |A + I - 1|$. Une autre manière de calculer est de diviser le résultat par la racine carrée de 2 afin d'obtenir la mesure non normalisée (Martin 2000).



Une classe très stable mais très abstraite sera donc considérée comme très bonne. Une classe très instable mais très concrète également. En réalité, plus on se trouve proche de la ligne qui relie ces deux points, mieux c'est.

Tout comme les autres métriques en relation avec le coupling, elle peut apporter des informations intéressantes qui peuvent être utilisées pour améliorer l'architecture et la qualité du logiciel. Il faut cependant tenir compte du fait qu'elles ne sont pas parfaites et l'utilisation de ces dernières comme seul indicateur est téméraire (Martin 2000, page 27).

Plus la valeur est proche de zéro, mieux c'est. Une valeur supérieure à 0.7 peut être problématique mais il est difficile d'éviter de tels résultats dans certains cas (Smacchia 2013).

2.10. Lack of Cohesion Of Methods (LCOM)

Cette métrique nous informe sur le niveau de cohésion d'une classe (Wikipédia 2013 : Cohesion). Elle permet ainsi d'évaluer le respect du principe de responsabilité unique (Wikipedia 2013: Single responsability principle). Ce principe indique qu'une classe ne devrait avoir qu'une seule raison de changer.

La cohésion diminue quand les fonctionnalités d'une classe ont peu de choses en commun et quand ces dernières s'occupent d'une variété importante de tâches différentes (Wikipédia 2013).

On peut obtenir cette métrique avec les formules suivantes (Smacchia 2013):

$$LCOM = 1 - \frac{\sum MF}{M \times F}$$

$$LCOM\ HS\ = \left(M - \frac{\sum MF}{F}\right) \times (M - 1)$$

- M est le nombre de méthodes dans la classe (inclut les méthodes statiques et d'instance, les constructeurs, les propriétés et les events).
- F est le nombre de champs d'instance.
- MF est le nombre de méthodes de la classe appelant un champs donné.
- Sum(MF) est donc la somme totale de la valeur obtenue par MF sur l'ensemble des champs de la classe.

Une classe qui contient des méthodes et des champs d'instance qui se référencent beaucoup entre elles aura une très bonne cohésion. Cela signifie en clair que ces méthodes et champs répondent à un même besoin. Si la cohésion est faible, cela pourrait indiquer que la classe ne respecte pas ces principes et qu'il faut la diviser.

Cette métrique est directement liée au principe de responsabilité unique (Wikipedia 2013: Single responsability principle). Elle également lié à l'inversion de dépendance qui analyse la testabilité, la réutilisabilité et la maintenabilité d'une classe (Goodman 2013). Il s'agit donc là d'une métrique intéressante pour évaluer la qualité du code. Une valeur supérieure à 0.8 avec un nombre de champs supérieur à 10 et un nombre de méthodes également supérieur à 10 pourrait être problématique (Smacchia 2013).

Actions possibles

Il faut tenter de respecter autant que possible le principe de responsabilité unique (Wikipedia 2013: Single responsability principle) en divisant les classes qui violeraient ce principe.

2.11. Relational Cohesion

La relational cohesion est le nombre moyen de relations internes à un module (Smacchia 2013). Si R représente le nombre de références qui sont internes au module et N le nombre total de types qu'il contient, alors la relational cohesion sera calculée à l'aide de la formule H = (R + 1) / N.

Si un module contient 50 classes et que ces 50 classes référencent au moins 2 autres classes du même module, le calcul donnera: (100 + 1 / 50) soit 2.02. Une module ou ensemble de classes devrait avoir une valeur élevée. Cela signifie que les différentes classes qui le composent sont fortement en relation.

Il s'agit donc là d'une métrique intéressante pour évaluer la qualité du code. Une valeur se situant entre 1.5 & 4.0 est conseillée (Smacchia 2013). Une valeur trop élevée pourrait néanmoins témoigner d'une complexité trop grande.

2.12. Size of instance

La taille de l'instance (Size of instance) correspond au nombre de bytes utilisé en mémoire par un objet instancié (Smacchia 2013). Il comprend la somme de la taille de tous les champs (fields) du type, ainsi que de ceux dont il hérite.

Les gros objets, s'ils sont instanciés souvent et en grand nombre, peuvent contribuer à dégrader fortement les performances. Identifier ces objets dans le code a donc une utilité certaine. Il est recommandé de ne pas dépasser une valeur de 64 (Smacchia 2013). Il faut cependant relativiser cette recommandation puisque dans de nombreux cas, c'est tout simplement impossible ou non souhaitable.

Actions possibles

Créer des objets en fonction de l'usage qu'on en fait et profiter de l'héritage ou des interfaces pour tenir compte des différents cas de figures.

2.13. Specialization index

L'index de spécialisation (specialization index) donne comme indication le degré de spécialisation d'une classe. Il est calculé avec la formule suivante :

$$\frac{NORM \times DIT}{NOM}$$

- NORM (Number of Overriden Methods) est le nombre de méthodes redéfinies
- DIT (Depth in Inheritance Tree) est la profondeur de l'arbre d'héritage (0 = pas d'héritage)
- NOM (Number of Methods) est le nombre de méthodes de la classe

Plus la classe redéfinit des méthodes qu'elle hérite d'autres classes et plus la profondeur de l'héritage est grand, plus cet indice augmente. Il diminue quand le nombre de méthodes spécifiques augmente et quand les méthodes redéfinies diminuent. Une valeur de 1.5 est considérée comme trop élevée (Larive 2013).

Actions possibles

L'héritage n'est pas toujours idéal quand des classes se spécialisent trop. L'utilisation des interfaces est recommandée. Cela permet également de clarifier le code et d'éviter des problèmes de maintenance (Larive 2013).

2.14. Number of parameters

Il s'agit du nombre de paramètres d'une méthode. Les méthodes avec trop de paramètres peuvent rendre le code moins performant mais aussi en réduire la visibilité. Il est recommandé d'avoir un nombre de paramètres réduit. Il est parfois contreproductif de contrôler cette valeur dans le code généré. Il est conseillé d'avoir un nombre maximum de 5 arguments (Smacchia 2013).

Actions possibles

Il existe principalement deux manières de réduire les paramètres d'une méthode. La première est de privilégier l'usage de propriétés ou champs sur la classe pour traiter les différents états. La seconde consiste à fournir l'instance d'une classe ou structure dont le rôle est de contenir les différents paramètres.

2.15. Number of variables

Il s'agit du nombre de variables déclarées dans le corps d'une méthode. Un nombre trop élevé peut avoir un impact sur la lisibilité de celle-ci. Comme pour le nombre de paramètres d'une méthode, il peut être contreproductif de calculer cette valeur sur du code généré. Il est conseillé d'avoir un nombre maximum de 8 variables dans le corps d'une méthode (Smacchia 2013).

Actions possibles

Diviser la méthode ou fonction en plusieurs méthodes ou fonctions.

2.16. Number of overloads

Le nombre de surcharges (number of overloads) permet de déterminer le nombre de méthodes du même nom. Si une méthode n'est pas surchargée, la valeur sera égale à 1. Un nombre trop important de surcharges peut réduire la lisibilité du code et réduire la qualité. Trop de méthodes surchargées peuvent indiquer dans certains cas une mauvaise utilisation des langages (Smacchia 2013). Il est conseillé d'avoir un nombre maximum de 6 surcharges de méthodes (Smacchia 2013).

Actions possibles

Réduire le nombre de méthodes surchargées en utilisant les fonctionnalités d'initialisation d'objet du langage quand elles sont disponibles.

2.17. Number of coding rules violations

En plus des métriques standards, il existe des règles de coding qui sont dépendantes du langage ou des technologies utilisées. Ces règles sont généralement assez basiques mais la présence d'un nombre trop important d'alertes les concernant peut être le symptôme d'un problème plus lourd. On appelle cela les « code smells » (Wikipédia : Code smell, 2013). Une taxonomie a été définie par Mäntylä & Lassenius (2006).

La pertinence de ces alertes est discutable et le développeur a d'ailleurs la possibilité de désactiver celles-ci via des attributs dans le code. Elles couvrent en principe toutes les catégories généralement rencontrées dans les métriques évaluant la qualité du code :

- Maintainability
- Reliability
- Efficiency
- Portability
- Usability

Dans de nombreux cas, ces métriques renseignent de vrais problèmes. L'utilisation d'outils repérant ces violations est indispensable dans tous projets de développement de logiciels. L'équipe de développement a même la possibilité de créer ses propres règles ou en sélectionner parmi les existantes en fonction de ce qui est pertinent ou non dans leurs projets.

Il existe une limitation à ces alertes : on peut avoir un nombre important de violations sans aucun bug. En d'autres termes, les violations peuvent indiquer des problèmes potentiels mais la correction de ceux-ci n'apporte parfois aucune valeur au résultat final. C'est cette raison qui pousse parfois les développeurs à désactiver certaines alertes.

Il est donc conseillé de rester raisonnable et de prendre en considération les contraintes du business. On est vite tenté de vouloir tout corriger, ce qui dans de nombreux cas est contreproductif. Se concentrer sur les violations sévères en priorité semble être un bon compromis.

Un nombre peu élevé de violations sévères peut être un critère de qualité du code.

A titre d'exemple, le lecteur peut consulter le document Philips Healthcare - C# Coding Standard (2009).

Actions possibles

Résoudre les problèmes progressivement pour du code existant. Par exemple, à chaque itération, décider de corriger un nombre donné de problèmes. Il faudra évidement prendre en compte les problèmes introduits par l'ajout de nouveau code dans le calcul.

3. Critères supplémentaires

3.1. Architecture

L'architecture d'un logiciel peut être comparée à celle d'un bâtiment à certains égards. Il ne s'agit pas vraiment de la qualité de la maçonnerie, du plafonnage ou des électros de la cuisine, mais de l'agencement des pièces et tous les autres aspects fonctionnels globaux. La qualité de cette architecture

est également très importante pour évaluer la qualité du code car elle impacte de nombreux facteurs de qualités.

3.1.1. Maintenabilité

Tout d'abord, la **maintenabilité** est directement liée à l'architecture. Elle doit être simple et efficace. En effet, une architecture lourde et ambitieuse peut être un frein énorme pour la maintenabilité sans apport de valeur réelle. A l'inverse, une architecture inadaptée peut également poser de gros problèmes à ce niveau. Elle doit répondre aux besoins mais ne pas créer de nouveaux problèmes comme par exemple l'impossibilité de trouver des développeurs compétents pour travailler dessus. Ce dernier point rend souvent le concepteur de l'architecture indispensable, ce qui met parfois en péril un projet tout entier quand ce dernier décide de quitter l'entreprise.

3.1.2. Evolutivité

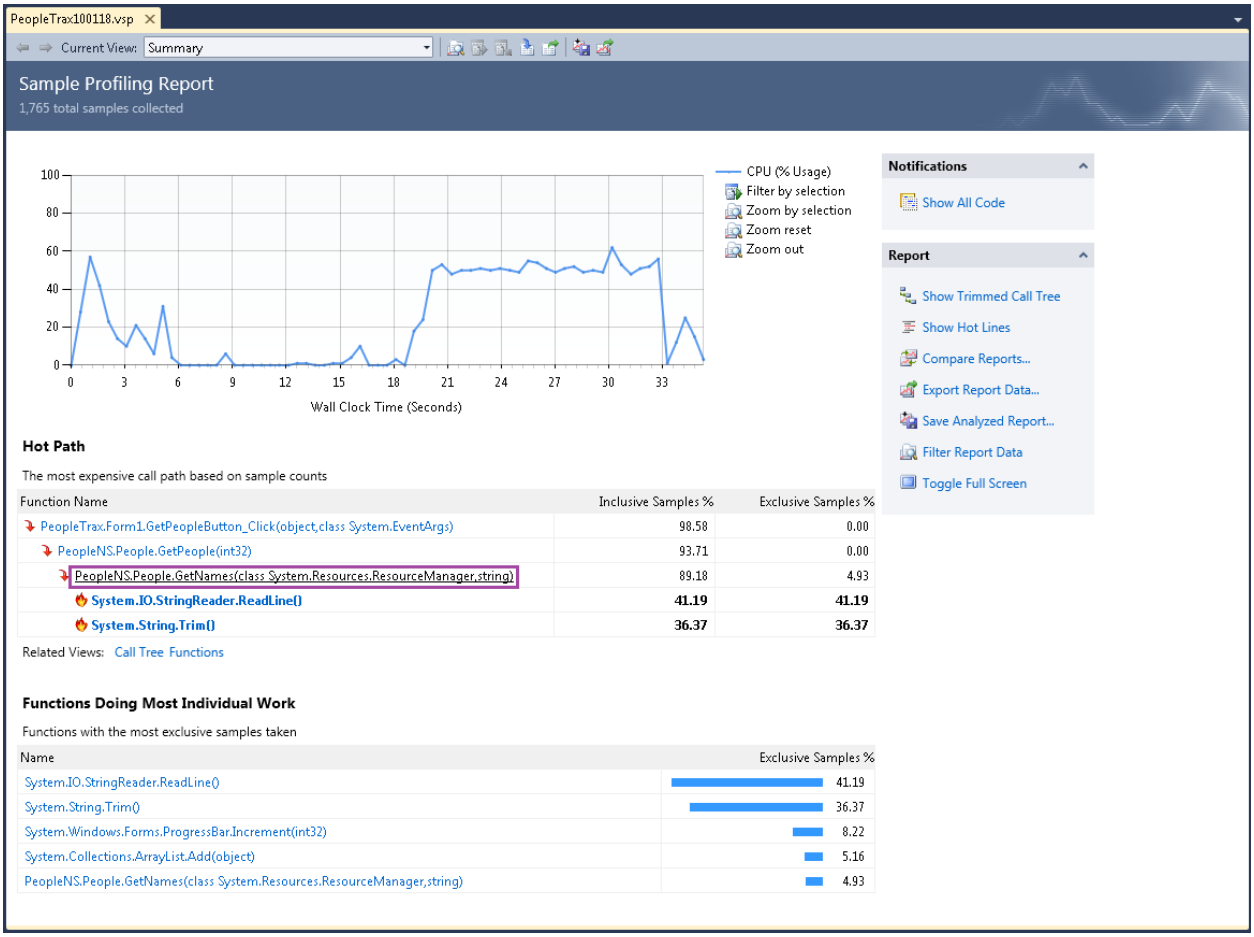
Ensuite, une architecture de qualité doit permettre une bonne **évolutivité**. Ceci est particulièrement important dans ces secteurs où le changement est très rapide comme l'IT et les nouvelles technologies en général. Cette évolutivité doit être prise en compte de manière raisonnable. Il n'est pas rare de tomber sur des architectures très complexes qui ont été conçues de cette manière pour permettre une évolution plus aisée. Le problème, c'est que dans des entreprises où l'évolution du marché est incertaine, beaucoup des efforts investis pourraient s'avérer contreproductif.

3.1.3. Performance

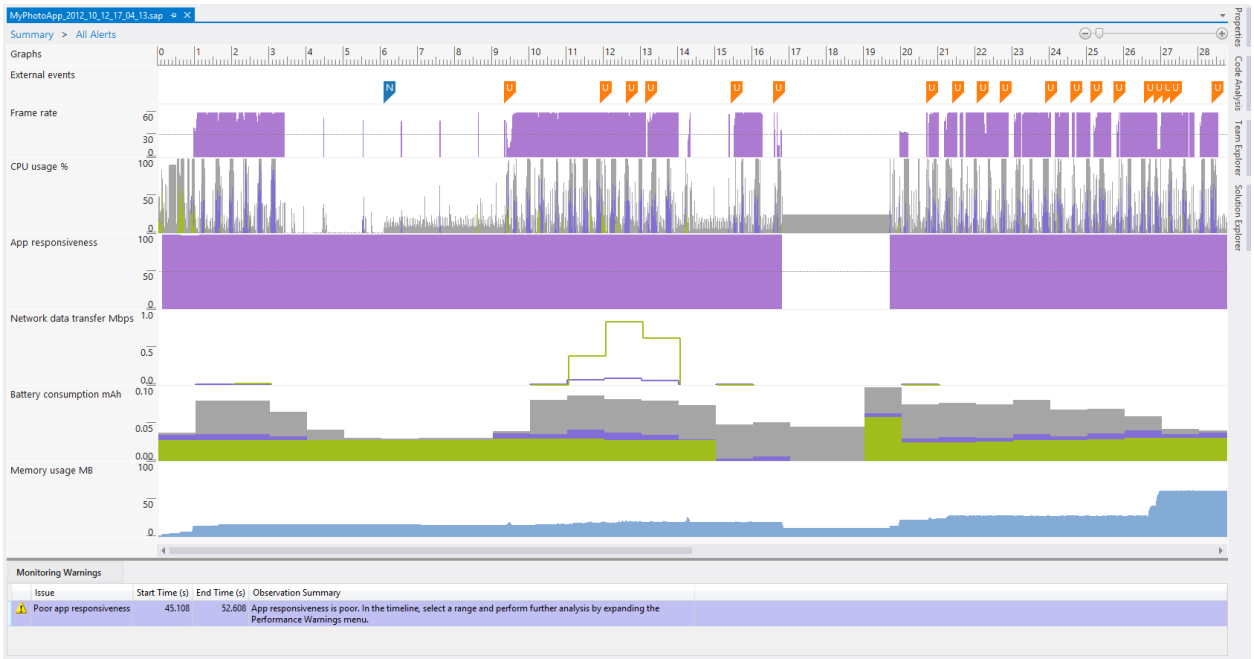
La **performance** est très souvent l'un des critères principaux dans l'élaboration de l'architecture. Pour évaluer la qualité d'un logiciel, la plupart des gens a tendance à trouver une architecture très performante (mais plus complexe) comme de meilleure qualité qu'une moins performante (mais plus simple). Le problème se trouve dans le fait qu'il y a énormément de moyens différents d'implémenter une fonctionnalité et il n'est pas rare de trouver des façons de concevoir un logiciel très différentes. Le concepteur du logiciel est souvent tenté de rendre son logiciel très puissant (certains peuvent s'identifier personnellement à leur travail). Cette recherche de performance maximale peut contribuer à complexifier inutilement le logiciel pour tenter de résoudre des problèmes que l'on ne rencontrera peut-être jamais. C'est pour cette raison que je privilégie des architectures simples, qui répondent aux besoins actuels mais qui envisagent également l'avenir incertain. Il faut par contre identifier les parties du logiciel qui sous-performent et qui constituent ainsi ce qu'on appelle des goulots d'étranglement. Il s'agit là de problèmes de conception non souhaités qui empêchent le logiciel d'atteindre son niveau de performance normal.

La performance peut être évaluée par des tests fonctionnels car les outils d'analyse statique ne peuvent vraiment donner que des indications (par exemple via les violations de règles). Il existe néanmoins des outils qui permettent d'automatiser cette tâche : les profilers, les load tests (stress) et les « performance counters ».

Des outils appelés « profilers » permettent d'évaluer les performances des différentes parties du code ainsi que l'utilisation de la mémoire. Voici un exemple de vue d'un rapport mesurant la performance des différentes méthodes d'un logiciel :



Les outils les plus récents permettent de réaliser des rapports semblables à celui-ci-dessous :



Les outils de profiling sont variés. Il en existe des dizaines (Wikipédia 2013 : List of performance analysis tools).

On trouve aussi des outils qui testent les performances des applications en simulant des montées de charge. Ces applications sont capables d'exécuter des scripts enregistrés sur base de vraies actions d'utilisateurs, ce qui rend les résultats très réalistes. Ces tests automatiques peuvent être ajoutés dans le processus d'intégration continue et ainsi alerter l'équipe de développement si certains seuils sont atteints ou s'il y a régression.

D'une manière générale, il est conseillé d'avoir des mesures de performance dans des endroits stratégiques du code pour permettre l'usage d'outils comme par exemple les Performance Counters de Microsoft.

3.1.4. Pertinence

Tous les points évoqués ci-dessus impliquent de faire des choix technologiques ou des types d'implémentation (comment une fonctionnalité est traduite en code). Un autre problème se pose alors, c'est la **pertinence** de ces choix avec la réalité. Cela inclut le code produit pour répondre à des besoins non existants ou bien la prise en compte de problèmes futurs potentiels, mais de manière peu raisonnable. Dans le premier cas, on peut prendre comme exemple la construction d'une voiture de course avec système de freinage révolutionnaire quand la demande initiale demandait simplement une solution pour transporter 2 personnes du point A au point B. Une moto aurait pu faire l'affaire. Dans le second cas, on peut prendre comme exemple une maison construire dans le nord de la Norvège qui contiendrait un système de climatisation. On ne sait jamais avec le réchauffement climatique... Une mauvaise connaissance du business pourrait également amener une équipe de programmeurs à faire de mauvais choix d'architecture.

Mais ces choix peuvent être faits pour des raisons toutes autres et moins bien intentionnées et ces cas sont suffisamment fréquents pour être mentionnés. Des développeurs pourraient être tentés de choisir tel ou telle solution technique pour unique but d'augmenter leurs propres connaissances et ainsi d'augmenter leur valeur sur le marché. C'est ce que j'appelle le « CV Driven Development ».

Le manque de pertinence peut couter très cher. Tout d'abord parce que concevoir des choses inutiles coute de l'argent mais aussi rend le code plus difficile à maintenir. De plus, si des concepts de programmation avancés sont généralisés dans le produit, il devient beaucoup plus difficile pour le propriétaire du logiciel de recruter des développeurs compétents. Ce cas de figure devient dramatique quand le programmeur, qui s'est approprié le code de cette manière, décide de démissionner. Il est conseillé d'utiliser les standards de l'industrie autant que possible pour éviter les problèmes de recrutement.

3.2. Style & lisibilité

Le style peut être assimilé à la structuration d'un texte : chaque paragraphe est physiquement distinct et la ponctuation correctement utilisée. Le code étant un langage écrit, il n'échappe pas aux problèmes de mise en page. Pour le code, on parle également d'indentations, de structure, de nommage et de

ponctuation d'une certaine manière. Toutes ces manières de présenter n'affectent pas directement le code compilé, et donc le fonctionnement du logiciel, mais uniquement la lisibilité du code. Il s'agit donc d'un critère de maintenabilité.

Le style est la première chose qu'on voit quand on regarde le code. Un effet de halo est inévitable. Cela pourrait jouer en défaveur d'une évaluation humaine. Il existe des outils d'analyse de style statique qui ne sont évidemment pas affectés par ce biais.

Enfin bonne nouvelle, beaucoup des mises en forme peuvent être automatisées dans des éditeurs de code.

Il est possible d'améliorer le style à postériori en utilisant des outils spécifiques, ce qui est d'ailleurs fortement conseillé. En amont, le contrôle du style peut être effectué par un serveur d'intégration continue et des outils de type CheckStyle ou StyleCop.

3.3. Documentation Technique

La documentation accompagnant le code est-elle suffisante pour permettre à une personne externe de comprendre ce qui a été mis en œuvre sans l'assistance de l'équipe qui l'a produit ? En général, les outils modernes permettent de documenter les différentes classes et méthodes à l'intérieur de celles-ci, mais plus rapidement l'architecture en général. La présence d'un document décrivant cette dernière et justifiant les choix effectués est un très bon critère de qualité.

A cela s'ajoute les documents qui doivent permettre au développeur de configurer son environnement de travail afin de pouvoir commencer à travailler sur le code. Il s'agit par exemple d'une liste de SDK (Software Development Kit) ou d'outils d'une version bien déterminée.

Enfin dans certains cas, l'accès à l'historique du code dans le contrôleur de code source pourrait aider les nouveaux arrivants à mieux comprendre comment du code a évolué depuis sa création.

3.4. Fiabilité

La fiabilité correspond au nombre de défaillances rencontrées dans un intervalle de temps donné. Plus cette valeur est élevée, moins le logiciel est fiable. On peut mesurer cette fiabilité en procédant à des tests manuels mais aussi avec des tests de montée en charge automatisés.

La fiabilité inclut également la tolérance aux pannes. En cas de problème, le logiciel réagit-il de manière appropriée ? Les problèmes sont-ils enregistrés pour traitement ultérieurs ? La présence de ces mécanismes est un critère de qualité important. On peut citer par exemple la bonne gestion des erreurs et les systèmes de journaux les consignant.

3.5. Portabilité

La portabilité est la capacité d'un logiciel à fonctionner sur des systèmes différents. Les langages de haut niveau de type Java permettent généralement d'obtenir une portabilité importante.

La dépendance à une plateforme pourrait dans certains cas être un facteur influençant l'évaluation de la qualité du code.

3.6. Sécurité

La sécurité est l'un des aspects clés dans la mesure de la qualité du code. Souvent négligée, elle devient de plus en plus importante avec la montée en puissance des applications en ligne. On peut détecter certains problèmes par l'intermédiaire de logiciel d'analyse statique du code, mais cela est très limité. De plus, les exigences en matière de sécurité varient fortement d'un projet à l'autre. Elles seront très élevées pour une application bancaire, mais probablement beaucoup moins pour les logiciels dont l'usage est local ou qui traitent des informations disponibles publiquement.

Il existe cependant des logiciels qui permettent de réaliser un audit de sécurité automatiquement afin de détecter certaines des failles les plus courantes. Ces failles incluent l'injection de code SQL ou CRLF par exemple. A titre d'illustration, voyez Wapiti.

3.7. Nombre de bugs

Il s'agit du nombre de problèmes perturbant l'usage normal du logiciel. Ils sont extrêmement difficiles à identifier en lisant le code ou par analyse statique et sont généralement découverts au moment où ils se manifestent. De ce fait, le nombre de bugs réel est rarement connu. Ces bugs peuvent correspondre à des violations de règles, mais la plupart du temps, il s'agit de problèmes fonctionnels très difficilement identifiables par des outils automatisés.

La liste des bugs « connus », c'est-à-dire enregistrés dans l'outil de bug tracking, peut donner une indication de la qualité du code. Plus il y en a de sérieux, plus il sera nécessaire d'intervenir.

4. Tableau récapitulatif

Métrique	Méthode	Langage	Localisation	Valeurs possibles	Valeur repère	Intérêt
Complexité cyclomatique	AS	Tous	A-MO-C-ME	1 à ∞	Max 10	Elevé
SLOC	AS	Tous	A-MO-C-ME	1 à ∞	-	Faible
Densité des commentaires	AS	Tous	A-MO-C-ME	0 à 100%	Entre 20% & 40%	Modéré
Couverture de code	AS*	Tous	A-MO-C-ME	0 à 100%	80%	Elevé
Duplication de code	AS	Tous	A-MO-C-ME	0 à 100%	0	Faible
Afferent coupling	AS	Tous	MO-C	0 à ∞	-	Elevé
Efferent coupling	AS	Tous	MO-C	0 à ∞	Max 20	Elevé
Instability	AS	Tous	MO	0 à 1	Entre 0.0 & 0.3 et entre 0.7 et 1.0	Elevé
Abstractness	AS	Tous	MO	0 à 1	-	Elevé
Distance from main sequence	AS	OO	MO	0 à 1	Max 0.7	Elevé
LCOM	AS	OO	ME	0 à ∞		Elevé
Relational Cohesion	AS	OO	MO	0 à ∞	Entre 1.5 & 4.0	Elevé
Specialization Index	AS	OO	C	0 à ∞	Maximum 1.5	Elevé
Size of instance	AS	OO	C	0 à ∞	Max 64	Faible
Number of variables	AS	Tous	ME	0 à ∞	Max 5	Faible

Number of methods	AS	OO	C	0 à ∞	Max 8	Faible
Number of overloads	AS	OO	C	0 à ∞	Max 6	Faible
Number of coding rules violations	AS	Tous	A-MO-C-ME	0 à ∞	-	Elevé
Maintenabilité	EX	Tous	A-MO-C-ME	-	-	Elevé
Evolutivité	EX	Tous	A-MO-C-ME	-	-	Elevé
Performance	EX + AS + AU	Tous	A-MO-C-ME	-	-	Modéré
Pertinence	EX	Tous	A-MO-C-ME	-	-	Modéré
Style & lisibilité	EX + AS	Tous	A-MO-C-ME	-	-	Modéré
Documentation	EX	Tous	A-MO-C-ME	-	-	Modéré
Fiabilité	EX	Tous	A-MO-C-ME	-	-	Elevé
Portabilité	EX	Tous	A-MO-C-ME	-	-	Modéré
Securité	EX	Tous	A-MO-C-ME	-	-	Modéré
Nombre de bugs	EX	Tous	A-MO-C-ME	0 à ∞	-	Modéré

Légende : AS = Analyse statique |EX = Expert |AU = Audit |A = Application |MO = Module | C = Classe |ME = Méthode |OO = Orienté Objet

5. Références bibliographiques

Albrecht, A. (1983). *Software Function, Source Lines of Code, and Development Effort Estimation.* http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1703110&searchWithin%3Dp_Authors%3A.QT.Albrecht%2C+%2FA%2F.J..QT.%26searchWithin%3Dp_Author_Ids%3A37850740200

Basili, V. R., Briand, L., Melo, W. L. (1995). *A Validation of Object-Oriented Design Metrics as Quality Indicators*. IEEE Transactions on Software Engineering 22 (10): 751–761. http://drum.lib.umd.edu/bitstream/1903/715/2/CS-TR-3443.pdf

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ. http://userfs.cec.wustl.edu/~cse528/Boehm-SE-Economics.pdf

codEnforcer (2013). http://codenforcer.com/coupling.aspx

Code smell. (2013, July 1). In Wikipedia, The Free Encyclopedia. Retrieved 18:51, August 9, 2013, from http://en.wikipedia.org/w/index.php?title=Code_smell&oldid=562371002

Cohesion (computer science). (2013, May 21). In *Wikipedia, The Free Encyclopedia*. Retrieved 16:05, August 8, 2013, from http://en.wikipedia.org/w/index.php?title=Cohesion_(computer_science)&oldid=556081531

Cornett, S. (2011). *Code Coverage Analysis*. http://www.bullseye.com/coverage.html#release

Danial, A. (2013). *CLOC Limitations*. Récupéré le 2 aout 2013 depuis http://cloc.sourceforge.net/#Limitations

Dijkstra, E. W. (1983). *The fruit of misunderstanding*. Page consulté le 2 aout 2013 à partir de http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD854.html

Glover, A. (2013). In pursuit of code quality. http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=code+quality

Goodman, D. (2013). *Software Design Principles (SOLID)*. http://davidgoodman.co.uk/tag/solid-principles/

Grenning, J. (2013), Code Coverage's Mixed Message. http://www.renaissancesoftware.net/blog/archives/273

Halstead, Maurice H. (1977). *Elements of Software Science*. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7. http://www.amazon.com/Elements-Software-Science-Operating-programming/dp/0444002057/

Hunt, A., Thomas, D. (1999). *The Pragmatic Programmer*. Addison Wesley. http://www.amazon.fr/The-Pragmatic-Programmer-Journeyman-Master/dp/020161622X/

Kan, S. (2003). Metrics and models in software quality engineering. (Second ed.). Boston: Addison-Wesley. http://www.amazon.fr/Metrics-Models-Software-Quality-Engineering/dp/0201729156

Kolawa, A. (2009). *Unit Testing Best Practices*. http://www.parasoft.com/jsp/printables/unittestting.pdf

List of performance analysis tools. (2013, August 25). In Wikipedia, The Free Encyclopedia. Retrieved 08:28, September 4, 2013, from http://en.wikipedia.org/w/index.php?title=List_of_performance_analysis_tools&oldid=570104690

List of unit testing frameworks. (2013, August 3). In *Wikipedia, The Free Encyclopedia*. Retrieved 12:47, August 6, 2013, from http://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=566951015

Lambertz, K. (2007). Complexité et qualité: comment mesurer la complexité d'un logiciel. MSCoder. http://www.verifysoft.com/fr_cmtpp_mscoder.pdf

Larive, A. (2013). *Mesure de la qualité du code source - Algorithmes et outils*. http://www-igm.univ-mlv.fr/~dr/XPOSE2008/Mesure%20de%20la%20qualite%20du%20code%20source%20-%20Algorithmes%20et%20outils/indice-de-specialisation.html

Lieberherr, K. J., Holland, I. (1989). *Assuring Good Style for Object-Oriented Programs*. IEEE Software, September 1989, pp 38-48. http://www-public.it-sudparis.eu/~gibson/Teaching/CSC5021/ReadingMaterial/LieberherrHolland89.pdf

List of programming languages. (2013, July 30). In Wikipedia, The Free Encyclopedia. Retrieved 12:48, August 2, 2013, from http://en.wikipedia.org/w/index.php?title=List_of_programming_languages&oldid=566431816

List of tools for static code analysis. (2013, August 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 18:54, August 9, 2013, from http://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=566849932

Mäntylä, M. V. and Lassenius, C. "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study". Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431. http://www.soberit.hut.fi/~mmantyla/ESE_2006.pdf

Martin, R. C. (2000). *Design Principles and Design Patterns*. Object Mentor. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

McCabe (December 1976). "A Complexity Measure". IEEE Transactions on Software Engineering: 308–320. http://www.literateprogramming.com/mccabe.pdf

McConnell, S. (2006). *Software Estimation : Demystifying the Black Art*.Microsoft Press. http://www.amazon.com/Software-Estimation-Demystifying-Practices-Microsoft/dp/0735605351/

Park, R. E. (1992). *Software Size Measurement : A Framework for Counting Source Statements*. http://www.sei.cmu.edu/reports/92tr020.pdf

Philips Healthcare - C# Coding Standard
(2009). http://www.tiobe.com/content/paperinfo/gemrcsharpcs.pdf

RefactorIt. Website. http://refactorit.sourceforge.net/

Refactoring if/else logic, StackOverflow.com. http://stackoverflow.com/questions/2913495/refactoring-if-else-logic/2913518

Réusinage de code. (2013, juillet 5). *Wikipédia, l'encyclopédie libre*. Page consultée le 12:04, août 2, 2013 à partir de http://fr.wikipedia.org/w/index.php?title=R%C3%A9usinage_de_code&oldid=94719037

Savoia, A. (2004) How Much Unit Test Coverage Do You Need? – The Testivus Answer. Retrieved 15:01, August 6, 2013 from http://www.artima.com/forums/flat.jsp?forum=106&thread=204677

Single responsibility principle. (2013, February 26). In *Wikipedia, The Free Encyclopedia*. Retrieved 06:40, September 4, 2013,
from http://en.wikipedia.org/w/index.php?title=Single_responsibility_principle&oldid=540764887

Smacchia, P. (2013). *nDepend Metrics*. http://www.ndepend.com/Metrics.aspx#RelationalCohesion

Spuida, B. (2002). *The fine Art of Commenting*.
http://www.icsharpcode.net/TechNotes/Commenting20020413.pdf

Strategy pattern. (2013, July 23). In Wikipedia, The Free Encyclopedia. Retrieved 14:07, August 6, 2013, from http://en.wikipedia.org/w/index.php?title=Strategy_pattern&oldid=565439992

Subramaniam, V. (2006). *Practices Of An Agile Developer: Working In The Real World*. The Pragmatic Programmers. http://www.amazon.fr/Practices-Agile-Developer-Working-World/dp/097451408X/

Tairas, R. (2013). *Code Clones Literature*. University of Alabama at Birmingham.
http://students.cis.uab.edu/tairasr/clones/literature/

Unit testing. (2013, August 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 12:35, August 6, 2013, from http://en.wikipedia.org/w/index.php?title=Unit_testing&oldid=566905149

Watson, A. H., & McCabe, T. J. (1996). Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric (NIST Special Publication 500-235). http://www.mccabe.com/pdf/mccabe-nist235r.pdf