LOG8371 : Ingénierie de la qualité en logiciel

Travail Pratique 1 : Rapport d'analyse

présenté à :
Fabio Petrillo

par
1686914 , Nam Lesage
1695014, Philippe Lelièvre
1750072, Joël Poulin
1744784, Vincent Rodier
1773922, Étienne Asselin

Table des matières

Table des figures

Liste des tableaux

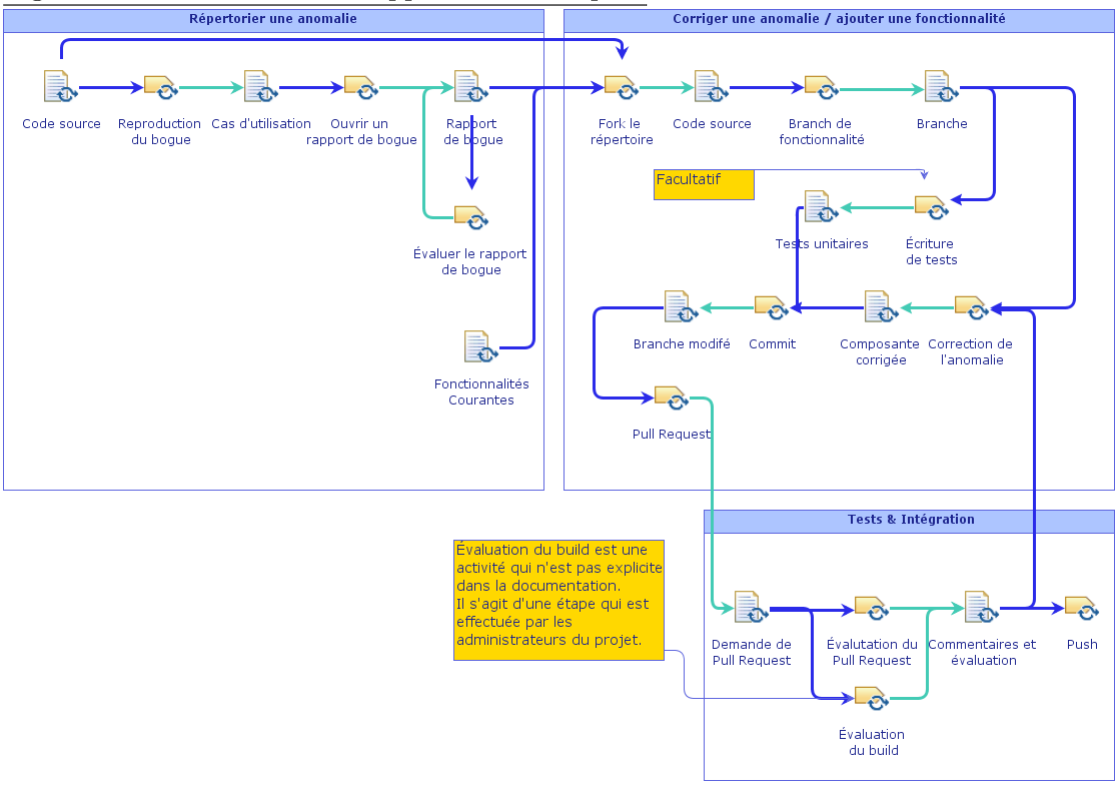
1 Introduction

Dolphin Emulator est un projet open-source qui permet aux utilisateurs d'émuler un jeu vidéo jouable sur la Wii et GameCube sur leur ordinateur. L'émulateur est multi-plateforme et est même disponible sur mobile sous le système d'exploitation Android. Le projet a démarré en 2003 et est devenu open-source en 2008 et compte aujourd'hui 310 contributeurs. Le répertoire repose sur trois branches principales : master, hotfixes et stable. Selon leur processus de développement, les contributeurs doivent se créer une branch afin d'implémenter leur fonctionnalité et faire un pull request avant de pouvoir fusionner avec une des branches principales. Le projet détient aussi un guide de codage afin d'indiquer aux nouveaux contributeurs quelles sont les attentes et les normes. Ce travail a pour but d'analyser d'un point de vue de qualité en se basant sur plusieurs modèle et métriques comme les facteurs de qualité logiciel McCall's par exemple.

2 Description du processus

À l'aide de la documentation fournis par le logiciel Dolphin, nous avons répertorié le processus nécessaire pour effectuer une modification au projet. La figure 1 ci-dessous illustre schématiquement ce processus et déquante les différentes étapes d'assurance qualité.

Figure 1 : Processus du développement de Dolphin



Le processus fournis dans la documentation de Dolphin décrit seulement le processus employé pour la correction d'un bogue ou de l'ajout d'une nouvelle fonctionnalité. Lors l'analyse de ce processus, on a réussis à ressortir quelques activités de controle de qualité. La première activité à noter lorsqu'une personne souhaite ajouter une modification au code est qu'il doit fork le répertoire localement sur ça machine. Faire des changements au code en isolation est une bonne pratique puisque ce n'est pas n'importe quelle modification qui se retrouve dans le code final directement. La deuxième activité de contrôle de qualité de ce logiciel est de faire une demande de pull request. Cette activité implique que le code est approuver par une personne autre que la personne qui à fait les modifications au code et permet donc d'avoir une deuxième avis quant à la qualité du code introduit dans le projet. Cette deuxième avis est faites en deux temps avec une évaluation du pull request et une évaluation du build. Suite à ces évaluations, deux choix sont possibles. Si le pull request est juger adéquat par un administrateur, le nouveau code est envoyé dans le répertoire pour les versions futur. Si l'administrateur juge que le code n'est pas encore

d'assez bonne qualité pour être ajouter au projet, l'administrateur donne des notes au créateur et celui-ci à la possibilité de corriger ces erreurs. Finalement, une autre activité de contrôle de qualité non obligatoire lors du processus est l'écriture de test.

3 Analyse du processus d'assurance qualité

3.1 Analyse selon l'ISO25000

3.1.1 Facteurs de fonctionnement du produit

3.1.1.1 Exactitude

Le processus de Dolphin ne semble pas bien s'occuper de l'exactitude du programme, mais en pratique le processus est suffisant. Pour contrôler le facteur d'exactitude, le processus introduit des tests unitaires facultatifs, des conventions de codage, des revues de code et des tests non-automatisés du build courant. En effet, le code ne possède pas de tests unitaires pour la majorité des composantes. Il est donc difficile de vérifier l'exactitude des sorties de l'application. Seul la revue de code et les tests du build par les codeurs s'assurent que le code est en bonne état. Avec le nombre limité de pratiques appuyant le facteur d'exactitude, on s'attend à ce que l'application donne de pauvres résultats. Cependant, si on analyse le build de l'application, celui-ci est stable et donne en effet les bons résultats sans erreur. On peut attribuer le succès au niveau de connaissances des codeurs qui effectuent les revues de code. De plus, le test de build manuel par les codeurs semble suffisant pour tester les sorties.

3.1.1.2 Fiabilité

Le processus de Dolphin semble prendre en compte au moins partiellement le critère de fiabilité. Le but principale de l'activité d'évaluation du build est de vérifier la fiabilité du logiciel afin d'éviter que le nombre d'échec d'exécution soit à un minimum. La même chose peut être dites pour les activités de tests. Bien que dans le cas du logiciel Dolphin il semble y avoir peu de test implémenter, le fait d'avoir des tests aide à la fiabilité du logiciel. La dernière chose à noter du processus du logiciel Dolphin par rapport à la fiabilité est que ce critère n'est pas explicitement défini. Par exemple, un crash "une fois de temps en temps" peut être un niveau de fiabilité acceptable pour Dolphin, mais aucune définition explicite n'est donnée pour la fiabilité du logiciel.

3.1.1.3 Efficacité

L'efficacité de l'émulateur est très importante pour que atteindre des utilisateurs avec des machines à ressources limités. Pour adresser l'utilisation des ressources dans le processus, on ne retrouve pas la présence de test de performance, ni de test de benchmark. Cependant, Dolphin fonctionne sur de nombreuses configurations de façon efficace en limitant le nombre de ressources et en offrant même des options pour améliorer la qualité du produit émulée. Faute d'avoir les tests, la communication et la collaboration avec la communauté semble aidé à tester l'efficacité, car c'est la communauté qui testera ce facteur sur leur machine. À travers les forums et le traqueur de fonctionnalités et de bogues, la communauté peut rapporter les problèmes d'efficacité et peut aussi aidé à corriger ces

problèmes en contribuant. On voit cette manière de procéder sur notre processus dans la section pour répertorier une anomalie.

3.1.1.4 Intégrité

Vu que l'émulateur ne requiert pas de données personnelles de l'utilisateur pour fonctionner, le processus n'a pas vraiment besoin d'établir des mesures ayant lien avec l'intégrité du projet.

3.1.1.5 Utilisabilité

Le critère d'assurance qualité de l'utilisabilité est quelque peu unique dans le câdre du projet Dolphin. Comme le logiciel Dolphin est un logiciel qui vise l'émulation de jeux vidéo, l'utilisabilité du logiciel est confiné à une très petite partie de son utilisation. De plus, les utilisateurs visé par le logiciel son les membres de la communauté de Doplhin, donc ce son ces membres qui définissent les critères qui définissent une bonne utilisabilité du logiciel. Dans cette optique, le processus du logiciel Dolphin n'est ni bon, ni mauvais en le sens que les critères concret qui serait utile pour testé l'utilisabilité son inexistant, mais que le feedback fournis par la communauté permet d'assurer une certaine qualité pour ce critère.

3.1.2 Facteurs de révision du produit

3.1.2.1 Maintenabilité

Le processus d'assurance qualité de Dolphin met à disposition aux contributeurs du projet une liste de recommandations de style de programmation et de formatage du code, qui seront évalués lors d'un *pull request*. Cette documentation permet de garder le code plus maintenable que si chacun des développeurs écrivaient du code sans convention. On augmente alors la capacité de compréhension du code et aide à le garder plus maintenable. Par contre, on ne donne aucune recommandation face à des éléments de programmation qui pourraient toucher l'architecture du logiciel, comme par exemple des lignes de conduite vis-à-vis des modifications de composants déjà en place, ou à la création de nouveaux modules. Une des façons d'améliorer la maintenabilité du code est de garder la couplage entre les différents composants de l'architecture à un niveau faible. La communauté pourrait proposer de faire attention lors de la programmation afin de minimiser les communications entre les composants. Cela évite le problème qu'une correction d'erreur affecte le code à plusieurs endroits et demande plus de modification.

3.1.2.2 Flexibilité

La flexibilité d'un logiciel est caractérisée par la difficulté que les contributeurs possèdent lorsqu'ils souhaitent faire des changements suite à de nouveaux requis. Pour le projet Dolphin, la flexibilité semble être d'un niveau convenable, puisque le logiciel est portable sur quatre plate-formes différentes et que le noyau du projet est isolé dans une section en

tant que *core*. Par exemple, Dolphin est déployable sur Android et existe une section dans la structure du projet pour les fichiers relatifs à cette plate-forme, mais les fonctionnalités du logiciel sont séparées de ces fichiers dans le module *core*.

3.1.2.3 Testabilité

La caractéristique de testabilité est celle qui évalue la capacité du logiciel à aider le programmeur qui tente de tester le système. Dans le cas de Dolphin, la testabilité du produit n'es pas développée comme élément important. L'écriture de tests est une action facultative aux développeurs. De plus, il n'y existe pas de documentation qui présente les résultats qu'on attend des différents composants qui pourrait donner un objectif aux développeurs avant de faire la demande d'un *pull request*. Aussi, il n'y a pas de fonctionnalités spécialisées qui permet de faire des tests automatiques et de générer des logs, résultats et rapports des modifications apportées avant le *pull request*. Ces tests automatiques pourraient permettre d'établir et d'évaluer les changements selon des critères d'assurance qualité définies par la communauté.

3.1.3 Facteurs de transition du produit

3.1.3.1 Portabilité

Dolphin fonctionne dans plusieurs environnements comme Linux, Windows et MacOS. Il fonctionne aussi avec plusieurs configurations de machine comme stipuler dans la section discutant de l'efficacité. La grande portabilité de Dolphin est déterminé dans les requis et dans le design de l'architecture du logiciel de l'application. Les principes d'assurance qualité permettant de vérifier ce facteur du logiciel serait les tests de configurations, mais on ne retrouve pas un tel concept dans le processus que nous avons illustrés. Encore une fois, Dolphin tire profit de la communauté pour tester les différentes configurations possibles. Les membres de cette communauté peuvent donc faire part des problèmes de portabilité directement sur les forums. Par la suite, les contributeurs pourront résoudre ces différents problèmes.

3.1.3.2 Réutilisabilité

Le processus de développement de Dolphin ne met pas en valeur le principe de réutilisabilité malgré son caractère multiplatforme. Les fonctionnalités demandées sont divisées selon la plateforme sur laquel l'application sera exécuté. Il s'agit d'une lacune dans le processus de développement.

3.1.3.3 Interopérabilité

Le processus de développement de Dolphin ne comprend aucune activité qui prend en compte l'interopérabilité. Il cependant est normal pour un tel logiciel de ne pas prendre en compte cette spécification vu son caractère indépendant.

3.2 Analyse selon le principe de Agile Software Quality

Cette section analyse le processus de développement de Dolphin selon les principes agiles. Elle énumérera et décrira les pratiques de l'agile que l'on retrouve dans le processus que nous avons illustré. Il est à noter qu'il est difficile pour un projet open source d'être complètement Agile. Cependant des pratiques peuvent fortement s'inspirer du concept et nous allons expliqué en quoi le processus s'apparente à l'agile.

3.2.1 Encourage le changement

L'un des principes fondamentaux de l'agile est d'accepter le changement. N'importe quel projet Open Source adhère par défaut à ce principe, car elle permet à la communauté de proposer des changements qui pourront être incorporés au code source. Dans le processus que nous avons dessiné, on voit la proposition de changements quand quelqu'un fait une demande d'une nouvelle fonctionnalité ou un rapport de bogue. Les propriétaires du projet et la communauté communiqueront ensemble pour évaluer la pertinence et la faisabilité de la proposition de changement. On peut comparer cela à la communication avec le client et l'équipe de développement dans le cadre d'un processus agile. La communication est un des piliers de l'agile.

3.2.2 Backlog

Le *backlog* dans le développement agile est une liste de *user stories* qui décrivent les exigences du logiciel. Le *backlog* inclut aussi les tâches et les bogues. Chaque item dans le *backlog* doit être noté et priorisé pour déterminer son importance. Ces items du *backlog* seront par la suite mise dans un *sprint backlog*. Un item peut être à faire, en progression ou faite pour décrire son avancement. En général, un tableau Kan Ban est utilisé pour visualiser la progression de tous les items.

Dolphin n'utilise pas nécessairement un *backlog*, mais utilise un outil de suivi des bogues et des fonctionnalités. Comme dans le *backlog*, les items peuvent être priorisés, classifiés et on peut suivre leurs progressions du début jusqu'à la fin de l'implémentation. On peut donc dire que Dolphin utilise ce principe de l'agile en utilisant un outil qui reprend beaucoup des concepts provenant du *backlog*.

3.2.3 Itération

Le développement itératif est une pratique mise à l'avant par l'idéologie Agile. C'est une méthode de développement dans laquelle on divise la réalisation du projet en courtes itérations pour réaliser successivement la conception, le développement ainsi que les tests.

Dans le cas du projet Dolphin, on semble retrouver une certaine forme d'itération : les administrateurs définisse une "Roadmap" dans laquelle ils spécifient les fonctionnalités qui doivent être implémentées ainsi que les bogues qui doivent être résolus avant la sortie de la nouvelle version. Cependant, il ne semble pas vraiment y avoir de contrainte de

temps pour la durée d'une itération, ce qui peut être problématique si on veut respecter la définition agile du développement itératif.

3.2.4 Revue de code

Une autre pratique mise à l'avant dans les processus agile est la revue de code. C'est une pratique dont l'objectif est de trouver des problèmes, des vulnérabilités ainsi que des erreurs de conception pour assurer la qualité et la maintenabilité du logiciel.

Le projet Dolphin a adopté cette pratique agile. En effet, il semble que toute nouvelle fonctionnalité proposée par un contributeur doit être révisée par un autre contributeur. En effet, par le biais d'une "pull request" sur GitHub, il est facile de pouvoir soumettre une fonctionnalité pour qu'elle soit révisée par des pairs. Ceux-ci peuvent donner des recommandations pour que l'auteur puisse améliorer son code s'il n'est pas conforme aux normes du projet.

3.2.5 Tests

Enfin, la réalisation de tests automatisés constitue un élément très important dans le développement agile. En effet, ils assurent la fiabilité des fonctionnalités développées et permette de détecter des erreurs lorsque d'autres fonctionnalités sont ajoutées.

Dans le cas du projet Dolphin, la réalisation des tests lorsqu'un contributeur soumet une nouvelle fonctionnalité est fortement encouragée. Cependant, elle n'est pas obligatoire, ce qui peut aller à l'encontre de l'idéologie agile.

4 Analyse de la qualité du code

Dans la présente section, on répertorie un extrait des erreurs de qualité de code que nous avons trouvé en analysant le code source du projet sur GitHub. Pour déterminer ce qu'était un anti-patron, nous nous sommes basé sur le livre Building Maintainable Software[1].

No	Type d'erreur	Fichier	Lignes
1	Répétition de nombre magiques	PerformanceCounter.cpp	29 et 40
2	Séquence Loop-switch	Frame.cpp	1260
3	Méthode trop longue	FileUtil.cpp	737
4	Méthode trop longue et complexe	Frame.cpp	1258

TABLE 1 – Anti-pattrons dans le code

4.1 Analyse de la qualité du projet

Afin d'analyser et de juger de la qualité du code source du projet, on doit définir une base de métriques qui serviront à quantifier le programme selon des critères prédéfinis. Pour ce faire, deux outils d'analyse de code ont été utilisés : Understand (Scitools) et Better Code Hub (Software Improvement Tools). La le résultat de Dolphin par l'outil Better Code Hub a donné un total de 4 sur 10.

4.1.1 Métriques orientées objets

Le tableau suivant présente une liste des métriques qui ont fait sujet d'étude et de résultats de recherche en génie logiciel.

Identifiant	Nom	Définition
DIT	Max Inheritance Tree	Profondeur maximale d'une classe dans un arbre d'héritage.
IFANIN	Count of Base Classes	Nombre de classes qui sont des classes de base.
LCOM	Percent Lack of Cohesion	100 pourcent soustrait par le pourcentage de méthodes d'une classe qui utilise ses attributs.
CBO	Count of Coupled Classes	Nombre de classes différentes pour lesquelles la classe en question est liée.
NOC	Count of Derived Classes	Nombre de classes qui se situent à un niveau de profondeur en dessous de la classe étudiée.
RFC	Count of All Methods	Nombre de méthodes que la classe possède, incluant les méthodes héritées.
NIM	Count of Instance Methods	Nombre de méthodes que la classe possède, incluant les méthodes héritées.
NIV	Count of Instance Variables	Nombre de lignes qui servent à écrire un programme.
WMC	Count of Methods	Nombre de classes locales, c'est-à-dire qui ne sont pas héritées.

TABLE 2 – Liste de métriques architecturales orienté-objets

4.1.2 Choix des métriques

Bien qu'il existe plusieurs types de métriques existantes, ils ne révèlent pas tous la même information lors de l'analyse de la conception. Ce concentrer sur un sous-ensemble peut permettre d'analyser plus en profondeur. Étant donné que le projet logiciel à l'étude est majoritairement écrit en C++, il est possible de choisir dans le registre des métriques

orientées-objets présentés à la figure 1. On se concentrera alors sur la suite de métriques de Chidamber et Kemerer. Les métriques architecturales orientés-objets utilisée qui feront face à interprétation sont le Percent Lack of Cohesion, le Depth of Inheritance Tree et le Count of Coupled Classes. De plus, les résultats des caractéristiques d'analyse de BetterCodeHub, listées dans le tableau 2 seront présentées et analysées.

4.1.2.1 Count of Couple Classes

Le Coupling Between Objects (CBO) est la métrique qui permet de mesurer le couplage entre les classes. Selon la théorie un système ayant un couplage faible est souhaitable puisqu'il est plus facile à maintenir et est plus compréhensible dans son ensemble. Par exemple, si une classe X est en relation avec une seule classe Y, la classe X aura un CBO de 1. Suite à l'analyse du CBO à l'aide de l'outil Understand, plusieurs classes possèdent un couplage élevé. Par exemple, la classe CFrame possède un CBO de 95, CISOProperties 47, CCodeview 32 et CCodewindow 53. Dans le système analysé, certaines classes du logiciel ont un couplage très faible, mais la valeur du CBO d'une grande proportion de classe est trop élevées. Une réduction du couplage peut être une solution à l'optimisation du code.

4.1.2.2 Lack of Cohesion in Methods

Le Lack of Cohesion in Methods (LCOM) est une métrique de la suite de Chidamber et Kemerer qui permet de donner une vue de l'utilisation des variables affectées à une classe par les méthodes de celle-ci. Dans le cas de Dolphin, les résultats des rapports montrent qu'il y a plusieurs classes qui possède une valeur élevée de LCOM.. Par exemple, quelques classes, comme Jit64, JitArm64, JitBase, JitBaseBlockCache et Jitx86Base, ont une valeur de LCOM supérieure à 75 pourcent. Ces valeurs hautes indiquent que ces classes manquent de cohésion entre ses méthodes et ses attributs internes. On repère ainsi un problème architectural.

4.1.2.3 Depth of Inheritance Tree

Le Depth of Inheritance Tree (DIT) indique a quelle profondeur d'un arbre hiérarchique d'héritage se trouve une classe. Le DIT de la plupart des classes de Dolphin n'est pas particulièrement problématique. Par contre, certaines classes pourraient faire preuve d'une révision de conception, afin de réduire leur DIT. Par exemple, NetPlayDialog (11), CFrame (12) et Jit64 (6) sont situées à une position creuse dans leur arbre d'héritage, ce qui fait en sorte qu'elles héritent de plusieurs classes mères, ce qui augmente leur complexité. Néanmoins, l'analyse de cette métrique ne révèle pas de problèmes majeurs.

4.1.2.4 Écrire des unités de code courtes

Dolphin a trop d'unités de code de son programme qui possèdent une quantité de lignes de code supérieure à ce que Better Code Hub, c'est-à-dire 15 lignes de code. L'assurance qualité d'un projet, surtout la maintenabilité, est plus efficace lorsque les unités ne possèdent pas beaucoup de lignes de code. Cela permet une meilleure réutilisabilité, est plus facile à comprendre et est mieux testable.

FIGURE 1 – Nombre de lignes de code par unité

4.1.2.5 Écrire des unités de code simples

La complexité de plusieurs unités du code source de Dolphin est supérieure a un McCabe

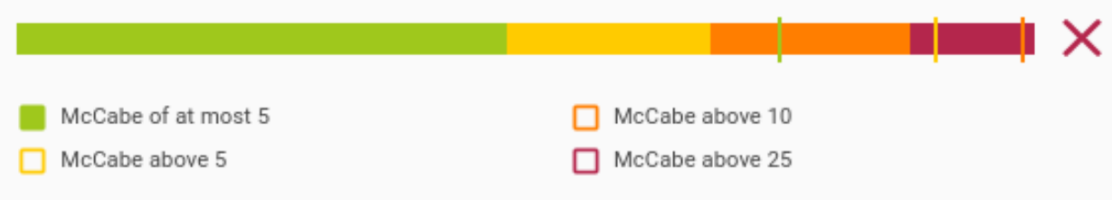


FIGURE 2 – Complexité de McCabe par unité

de 5, ce qui permet de rendre le code plus modelable et d'améliorer la capacité à tester. La plupart des unités on un McCabe supérieur à 5. Plusieurs ont une complexité de plus de cinq fois la valeur recommandée.

4.1.2.6 Éviter les répétitions de code

Les résultats de l'analyse statique nous révèlent que le code source de Dolphin ne possède

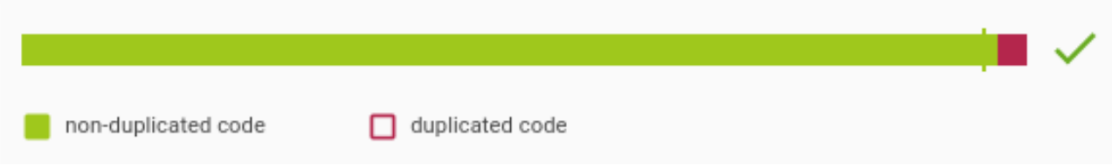


FIGURE 3 – Quantité de parties de code dupliqués

pas une quantité dangereuse de code dupliquées. Dans le cas où un problème serait repéré dans l'un des duplicatas de code, l'inconvénient est que la correction de ce problème doit être propagé à plusieurs endroits.

4.1.2.7 Garder les interfaces des unités de code petites

Comme le montre la figure 4, le code de Dolphin n'est pas très loin des standards de qualité du logiciel Better Code Hub. Il semble y avoir un surplus de méthodes possédant entre 2 et 4 paramètres que ce qui est recommandée. Cependant comme la grande majorité des fonctions possède moins de 2 paramètres, cette métrique est acceptable.

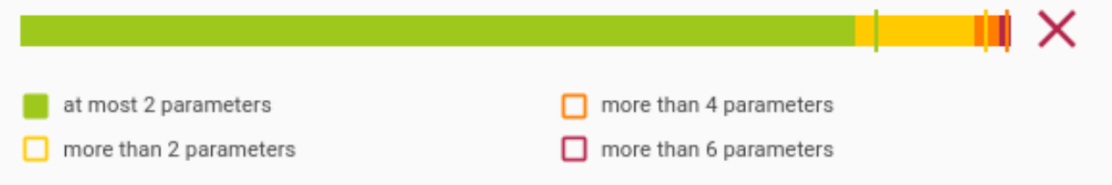


FIGURE 4 – Complexité de McCabe par unité

4.1.2.8 Garder le couplage faible entre les composants

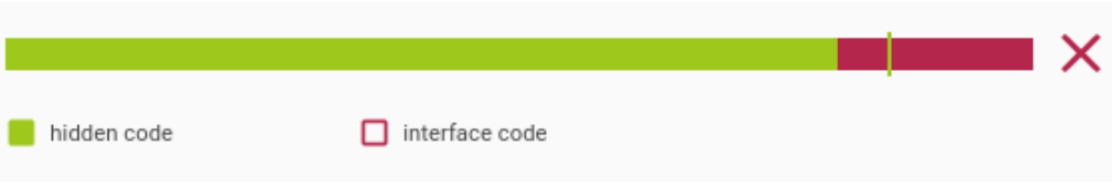


FIGURE 5 – Proportion de code d'interface dans le code

À la suite de l'analyse avec Better Code Hub, il y a généralement trop de couplage entre les composants. Le couplage faible entre les composants aident à la maintenabilité du code. Une méthode pour diminuer le couplage est d'utiliser le patron de conception abstract factory.

4.2 Analyse architecturale

Le programme Dolphin comporte certains défauts de conceptions. Nous diviseront notre analyse de code en deux parties : les problèmes architecturaux et les problèmes de qualité de code.

4.2.1 Problèmes architecturaux

Pour commencer, un des premiers problème architecturaux que notre équipe a détecté dans le projet Dolphin est l'utilisation de structure (struct) en guise de remplacement pour des classes et objets. Il s'agit d'une mauvaise pratique puisqu'il est impossible de profiter des avantages de l'orienté objet (ex : héritage).
Un deuxième problème repéré est la surutilisation de variables globales définies dans un namespace. Il peut être utile dans certains cas d'en avoir quelques unes, cependant dans le logiciel Dolphin cette pratique est beaucoup trop courante. Cette utilisation exagérée brise l'encapsulation souhaité d'un système.
Le dernier problème architectural qui sera traité dans ce document concerne la méthode utilisé pour stocker une grande de quantité de donnés du même type. En effet, la classe SConfig contient environs plus d'une centaine de booléen déclarer l'un à la suite de

l'autre. Cette façon de faire ajoute un niveau de difficulté afin de comprendre le code. L'utilisation d'un conteneur comme une hashmap rendrait plus facile l'ajout de nouvelles donnés en plus de rendre le code plus extensible.

4.2.2 Problèmes de qualités

Lors de notre analyse, certains problèmes de qualité ont aussi fait surface. L'un des premiers et plus gros problème identifié dans le logiciel Dolphin est le couplage élevé de plusieurs classes. En fait certaine classe ont un couplage énorme et ceci diminue significativement la facilité de compréhension des relations entre les différentes classes. Il est difficile de modifier une classe sans impacter les autres dépendantes ainsi la flexibilité du code est déficiente.
Un deuxième problème de qualité est le manque de cohésion. En effet, l'analyse faite lors de la qualité du code relève ce problème. En plus de rendre difficile de définir les responsabilités des classes, le problème de cohésion combiné avec le problème de couplage encourage la création de classe extrêmement longue plus connu sous le nom de "Blob". En observant le diagramme de classe, on peut dégager une tendance de création de longue classes, confirmant ainsi notre hypothèse.
Finalement, nous avons remarqué la présence d'anti-patron linguistiques. En effet, nous repéré des paires de méthodes dont le nom est identique à la différence d'un chiffre (ex : méthode1 et méthode2). Cela crée une confusion lors de la lecture du code quant au choix de la méthode à utiliser, obligeant le développeur à aller voir l'implémentation pour faire le bon choix de méthode.

5 Conclusion

En conclusion, il est clair que le logiciel Dolphin ne respecte pas particulièrement bien certaines normes de qualité logicielle. L'analyse du processus révèle qu'il est difficile pour un logiciel libre d'implémenter un processus clair et rigoureux qui aide à la qualité logicielle, mais cela ne veut pas dire qu'aucune mesure n'est mise en place ni que l'absence de ces mesure implique une qualité médiocre. L'analyse plus détailler du code source du logiciel Dolphin indique qu'il existe effectivement différents problèmes dans le code. Il est cependant important de noter que malgré l'utilisation de plusieurs logiciels d'évaluation de la qualité du code, il est parfois justifier pour une équipe de développement de faire des choix reconnu comme étant de la mauvaise pratique par un logiciel et que cela n'influence pas nécessairement la qualité du code dans son ensemble.

6 Annexe

Références

[1] J. Visser, S. Rigal, R. van der Leek, P. van Eck, and G. Wijnholds, *Building Maintainable Software, Java Edition : Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.", 2016.