IFT3913 - Rapport

Louis-André Brassard
Xavier Lapalme

October 2022

Question 1

Les deux métriques que nous avons choisie pour cette question sont la densité de commentaire ainsi que la complexité cyclomatique de McCabe. Nous avons choisi ces métriques puisqu'elles dresse un portrait global du code sans devoir analyser chaque fonctions de prêt. En effet, les deux sont basée sur un dénombrement quelqu'onque et de ces chiffres, nous pouvons inférer une réponse rapidement. Nous avons premièrement la taille, le nombre de ligne ici, qui est un indicateur de la complexité et les lignes de commentaires qui sont un indication de la quantité de documentation. Plus le rapport entre les deux est élevé plus il y a de documentation par rapport à la quantité totale de code. Ici nous avons 1.25% des lignes qui sont des commentaires. Selon une étude[1] le moyenne de densité de commentaire est de 19%. Avec cette métrique nous concluons qu'il n'y a pas assez de documentation pour la complexité du projet comme nous nous situons près de 17% sous la moyenne de densité de commentaire. De plus, si nous regardons la complexité cyclomatique de McCabe, nous obtenons un score de 21110, ce qui est assez important considérant la taille du code de 133000. Donc, avec ces deux informations, nous pouvons conclure que le niveau de documentation n'est pas approprié. Non seulement nous avons un taux de documentation bien en dessous de la moyenne, mais notre complexité de McCabe est très élevé ce qui peu causer des problèmes lorsque la personne qui à écrit le code initialement ne travailleras plus dessus, alors il n'y auras pas assez de documentation pour la complexité que représente le code. Ceci peu éventuellement causer des problèems de maintenance.

Question 2

Pour évaluer la modularité du code, nous avons choisie, LCOM et LCSEC. LCOM mesure la cohésion entre les classes tandis que LCSEC mesure le couplage. Ensemble elles nous donnent une bonne idée de la modularité du code et nous permet d'évaluer à très haut niveau si des changements doivent être fait dans le code. En utilisant l'outils de Spinellis, CKJM, nous avons pu remarquer que la moyenne du LCOM pour toutes les classes est de 0.41 alors que 247 classes ont un score de 0. Ces statistique nous montre que plusieurs classes ont une cohésion parfaite ce qui est bien, cependant, on voit que celles qui n'ont pas 0 , sont relativement élevés avec une moyenne de 0.67. Ceci indique que les classes ont soit peu de cohésion ou un cohésion parfaite. Ceci peut entrainer des problèmes de maintenance puisqu'il manque de continuitée entre les classes. Pour ce qui est du LCSEC, nous avons un score moyen de 18.9, ce qui très bas pour un code aussi gros et relativement complexe comme celui-ci. Ceci nous indique donc que le couplage entre classe de jfreechart est très bas. Ce qui indique une très bonne modularité. En combinant le faible couplage et une cohésion générale au delà de la moyenne, on peut conclure que le code est modulaire. Un code modulaire, surtout un faible couplage rend la maintenabilité du code plus simple puisque les changement son prévisible et ainsi corriger un bug n'en créer pas un autre à un endroit qui n'a pas de lien direct. Par contre, pour augmenter la maintenabilité du code, il pourrais être intéressant de rassembler les conecepts commun dans un seul et même endroit, ceci éviterais de devoir chercher un peu partout dans le code puisque tout serais au même endroit.

Question 3

Pour évaluer la maturité du code de jFreeChart, nous avons utilisé le nombre de commit par année et le nombre de "issues" générées par l'outil de qualimétrie SonarQube. Avec un peu plus de 4200 commits répartis sur 16 ans et avec la majorité de ceux-ci concentré dans les premières années, on peux déduire que le code est en effet probablement à maturité. En effet le nombre de commit tend à réduire avec les années. Ceci peux être interprété comme il ne reste plus beaucoup de bug majeur dans le code et que la version disponible est stable et complète. Pour ce qui des résultats de sonarQube, l'analyse du code a détecté plus de 3100 "issues". Ceci peu paraître énorme considérant que le code fait un peu plus de 130 000 lignes de code. Cependant, il est imporant de noté que les issues soulevées par sonarQube ne sont pas de problèmes majeurs comme des fautes dans la logique ou des bugs d'affichage. En effet, ces issues sont pour la plupart des "best practice" qui ne sont pas respecter ou encore des morceaux de code dupliqués. En soi ceci n'est pas tragique, mais ce n'est pas optimal non plus.

Bref, si nous prenons ces deux métriques pour déterminer la maturité du code, nous pouvons constater rapidement que le code est très mature avec de nombreuses contributions réparties à travers une longue période de temps. Cependant, les issues soulevées par SonarQube pourrait nous indiquer le contraire et nous faire penser qu'il reste beaucoup de chose à améliorer, mais ce ne sont pas des éléments important qui valent le temps et l'effort nécessaire pour le corriger. Ces métriques, bien que de très haut niveau, nous donne un apperçu global de la maturité du code tout en facilitant l'analyse du code.

Question 4

La première métrique que nous avons choisi pour cette question est la complexité cyclomatique de McCabe qui est estimé. Alors, selon cette métrique, nous pouvons dire que, oui, il peut bien être testé automatiquement puisqu'il y a moins de cas à tester car le programme peut prendre un petit nombre de chemins différents. La deuxième métrique que nous avons choisie est le RFC, elle nous donne la même réponse que la première puisque le RFC de chaque classe est très faible (souvent même zéro). Nous avons choisies c'est deux mmétriques car elles courvrent, et la création de test (complexité cyclomatique de McCabe) et le débbugage des bugs trouvées par ces test (RFC). Donc, si les tests sont plutôt facile à créer et que les problèmes trouvés par ces tests sont facile à régler, du moins trouver la source. Nous affirmons alors que le code peut facilement être testé automatiquement.

Conclusion

Pour conclure, à la lumière de nos résultats, nous pouvons dire que le niveau de maintenabilité de JFreeChart est plutôt bon. En effet, il est relativement modulaire et donc les correctifs apportés dans le code ne devrais pas causer de surprises plus loin dans le code. Cependant, comme soulevé à la question 1, la quantité de documentation n'est pas adéquate à la complexité du code. Ceci peut éventuellement freiner la maintenabilité du code si de nouveaux développeurs se joignaient au projet. Par contre, la maturité du produit viens équilibrer le tout et nous permet de conclure que le code est bel et bien maintenable dans son ensemble. Comme expliquer plus tôt, le code est maitenant très mature et ne comporte plus de bug majeur qui bloque les utilisateurs. Après 16 ans de commit réguliers, nous pouvons conclure que le code est a terme et que si jamais il a besoin d'être maintenu, cela risque d'être de petits changements qui s'effectue rapidement. Bref, nous croyons qu'une image claire émerge, soit que le code est maintenable, cependant il y a un effort qui doit être fait au niveau de la documentation pour assurer la pérénité du projet.

1

[1] https://dirkriehle.com/wp-content/uploads/2009/02/icse-2009-nier-for-web.pdf