Rapport d'analyse du code

Auteurs: Nicolas Han et Farès Abdelli

Projet étudié: Eclipse Collections

I. Introduction et présentation du projet

1. **Utilité du projet**

   - Eclipse Collections (EC) est une librairie, dont le développement a commencé sous un autre nom en 2004, visant à augmenter les options offertes par les outils du type Collection et autres, et améliorer leur efficacité, notamment au niveau de l'espace mémoire, en rajoutant notamment des types qui ne sont pas disponibles dans JDK.

   - Le projet étant une librairie, il ne se lance pas mais se télécharge et s'importe dans le code java où on en a besoin.

2. **Description du projet**

   - Il y a un readme dans le projet, mis à jour pour la dernière fois en décembre, donc logiquement à peu près à jour. De plus, ce readme est pertinent: il donne accès à la page web du projet, avec plusieurs langues disponibles, donne accès à des explications sur la façon de se servir du projet, pourquoi le projet existe, répertorie aussi les compatibilités entre les versions de java et les versions du projet.

   - La page web du projet donne accès à la javadoc complète de toutes les versions de EC, explique comment l'installer et s'en servir, propose des comparaisons graphiques entre différents objets proposés par JDK et leur version EC.

II. Analyse du projet

1. **Analyse du git**

   - On dénombre 100 contributeurs au projet, mais de façon très déséquilibrée: un contributeur en particulier, celui qui est à l'origine du projet, a apporté bien plus de modifications que les autres, mais on en compte 3 qui eux aussi se démarquent du reste, que ce soit par le nombre de commits ou par le nombre de modifications apportées (ajouts et suppressions). On compte aussi quelques bots parmi les 100 contributeurs, que l'on peut retirer de la liste. De plus, seuls les contributeurs principaux semblent avoir travaillé sur le projet de façon régulière dans le temps. On peut donc conclure que même si le projet compte un grand nombre de contributeurs, seuls quelques uns (en l'occurrence 3 ou 4) ont vraiment travaillé dessus.

   - Le projet est toujours actif: en effet, les derniers commits datent du 05/02/2024 (2 jours avant l'écriture de ce paragraphe). Cependant, on peut remarquer que le projet était bien plus actif entre sa création sur github (fin 2015) et l'année 2022, et que l'activité s'est réduite en 2023 (et 2024 pour l'instant).

- 15 branches sont définies, mais seules 3 peuvent être considérées comme encore actives, et une seule apparemment valide tous les tests.
- Il y a eu au total 1088 pull requests, dont 27 sont encore ouvertes. Le mécanisme des pull requests est donc utilisé.
- Il y a de plus 127 issues encore en attente d'être résolus, ce qui peut potentiellement être une bonne piste pour chercher des améliorations. En résumé, l'analyse du git révèle que le projet est encore actif, mais qu'il s'y trouve potentiellement des problèmes que l'on pourrait chercher à résoudre.

On dénombre dans ce projet une très grande quantité de classes (en l'occurrence plusieurs milliers dans l'entièreté du projet), c'est pourquoi dans la suite de ce rapport, on se concentrera seulement sur une sous-section du projet, que nous réduirons à environ 300 classes. On se concentrera sur le dossier *eclipse-collections/src/main/java/org/eclipse/collections/impl*, et en particulier sur certains sous-dossiers, à savoir **block, lazy et set, ces 3 dossiers nous amenant à un total de 285 fichiers**, dont la grande majorité sont des classes.

2. **Analyse de l'architecture**

A. Répartition des paquetages, utilisation de bibliothèques

- La première chose à remarquer est que le projet n'utilise pratiquement pas de bibliothèques extérieures. Une grande quantité d'imports sont effectués: on compte pour block, lazy et set respectivement 398, 920 et 1195 imports, mais au total, moins d'une dizaine d'imports qui ne viennent ni de java, ni des fichiers locaux. Il est intéressant de noter, entre autres, que malgré le grand nombre d'imports, aucun n'est inutilisé, et il n'y a aucun import de librairies différentes ayant des fonctionnalités similaires.
- Ensuite, on observe au total 30 paquetages pour un peu moins de 300 fichiers. Cela donne en moyenne aux alentours d'une dizaine de fichiers (package-info compris) pour un paquetage. Les paquetages sont organisés en couches, de profondeur allant de 2 à 3. En l'occurrence, la profondeur est de 3 dans block et lazy, et de 2 dans set, ce qui est en corrélation avec le nombre de fichiers des dossiers. On remarque de plus que la hiérarchie des tests (dossier *unit-tests*) semble suivre la hiérarchie des fichiers sources. En revanche, il est à noter que le package *org.eclipse.collections.impl.block.comparator.primitive* ne contient pas de fichiers, mis à part le fichier *package-info.java*.
- Comme mentionné ci-dessus, on a en moyenne un paquetage pour une petite dizaine de classes, mais en réalité, le contenu des paquetages sont parfois très éloignés de cette moyenne: on peut prendre l'exemple de *org.eclipse.collections.impl.block.procedure*, qui contient au total plus d'une cinquantaine de classes, là où d'autres paquetages en ont moins de 5, à l'instar de *org.eclipse.collections.impl.block.comparator.primitive*, comme dit ci-dessus. Ceci dit, il est important de nuancer: même si on peut constater un gros déséquilibre dans le nombre de classes entre certains paquetages, les cas observés précédemment sont des cas extrêmes, et les classes sont loin d'être toutes concentrées dans un unique paquetage. En effet, en reprenant l'exemple de *org.eclipse.collections.impl.block.procedure* qui est le paquetage le plus chargé en classes, celui-ci ne contient, par rapport à la totalité des dossiers étudiés, qu'environ un sixième de la totalité des fichiers, et même en se restreignant au dossier *block*,

ce dernier contient plus de 130 fichiers, et par conséquent plus de la moitié des fichiers se trouvent ailleurs que dans *org.eclipse.collections.impl.block.procedure*.

- En revanche, il faut noter la présence de quelques boucles de dépendances entre paquetages: dans block, par exemple, *org.eclipse.collections.impl.block.factory* et *org.eclipse.collections.impl.block.procedure* dépendent réciproquement l'un de l'autre, avec dans chacun des classes important des classes de l'autre paquetage.

B. À propos du code

- Le projet n'est pas exempt de code déprécié. En effet, on a:

  - block: 12 mentions de `@Deprecated`, dont 7 sur des classes entières et 5 sur des méthodes particulières.
  - lazy: 1 seule classe dépréciée, et aucune méthode.
  - set: aucune classe dépréciée, mais 28 méthodes dépréciées.

  On peut donc constater une présence relativement conséquente de code déprécié. Il faut ajouter de plus que ce code déprécié est en grande partie utilisé dans d'autres fichiers du projet, mais également dans les tests unitaires, ce qui laisse supposer qu'il a été testé, et donc qu'il effectue quand même ce qui est attendu de lui.

- Il semble par ailleurs que le projet ne contienne pas de code mort.

- Concernant le code dupliqué, l'analyse de SonarQube révèle que sur l'ensemble du projet, environ 16% du code est du code dupliqué, et après vérification, les sous-dossiers étudiés sont eux aussi concernés par ce problème.

- On peut de plus trouver, à certains endroits du projet, comme dans *set.strategy.UnifiedSetWithHashingStrategy* par exemple, des nombres magiques. Ils ne sont pas très fréquents, mais posent un problème dans le cadre d'une éventuelle modification de leur valeur, d'ajouts de méthodes ou d'extensions.

C. Contenu des classes

- On trouve dans les fichiers une certaine quantité de commentaires, et en particulier les javadocs:

  - block: on compte 167 javadocs et 133 commentaires commençant par `/*`, et 3 commentaires sur une ligne (commençant par `//`);
  - lazy: 26 javadocs et 94 commentaires commençant par `/*`, ainsi que 54 commentaires sur une ligne;
  - set: 84 javadocs et 58 commentaires commençant par `/*`, et 65 commentaires sur une ligne.

  On peut aussi noter l'absence de javadoc sur les méthodes héritées d'une super-classe, ce qui permet de laisser le code clair et malgré tout d'accéder à une javadoc pour ces méthodes.

- Concernant la répartition du code et la présence de potentielles god classes, on a, en termes de nombre de lignes:

        - block: au maximum 1557 lignes, au minimum 14 lignes, en moyenne 97 lignes, et la médiane est à 46 lignes;
        - lazy: au maximum 794 lignes, au minimum 14 lignes, en moyenne 114 lignes, et la médiane est à 93 lignes;
        - set: au maximum 2674 lignes, au minimum 16 lignes, en moyenne 370 lignes, et la médiane est à 188 lignes.

De toute évidence, un gros déséquilibre est à constater, en particulier dans block et set. Dans le cas de lazy, la proximité entre la moyenne et la médiane témoigne d'un meilleur équilibre du nombre de ligne, mais dans block et set, la moyenne est environ deux fois plus haute que la médiane, ce qui montre un écart colossal entre les valeurs au-dessus de la médiane et celles en-dessous. Les valeurs maximales et minimales sont elles aussi des bons témoins: dans le cas de set, la valeur maximale est plus de 13 fois plus haute que la valeur médiane, et dans block, le maximum est plus de 30 fois plus haut que la médiane, là où la valeur minimale n'est qu'environ 3 fois plus petite que la valeur médiane.

- L'utilisation de SonarQube montre aussi la complexité cyclomatique des classes entières, qui est calculée en effectuant la somme des complexités cyclomatiques des méthodes d'une classe. On a alors:
    - block: une complexité cyclomatique d'environ 10 en moyenne, avec un maximum à 213;
    - lazy: une complexité cyclomatique en moyenne à 14, et avec un maximum de 115;
    - set: une complexité cyclomatique moyenne de 57, le maximum étant à 504.

Il est important de noter une forte corrélation, et même un lien de causalité entre le nombre de lignes des classes et leur complexité cyclomatique totale: plus il y a de lignes dans une classe, plus il y a de chances que celle-ci contienne beaucoup de méthodes ou plusieurs de méthodes très complexes, ce qui va forcément faire gonfler la complexité cyclomatique totale.

D. Tests

- Pour finir, pour ce qui est des tests, les résultats semblent ne pas être cohérents entre eux, ce qui laisse supposer une erreur de mesure ou d'interprétation. En effet, on constate, dans chaque dossier:
    - block: 513 méthodes de tests pour 135 classes, soit environ 4 méthodes de tests par classe en moyenne;
    - lazy: 564 méthodes de tests pour 94 classes, donc en moyenne 6 méthodes de tests par classe;
    - set: 1264 méthodes de tests pour 58 classes, soit 21 méthodes de tests par classe en moyenne.

A priori, on en déduirait que les classes sont plutôt bien testées, avec au minimum plusieurs méthodes de tests par classe. On pourrait remarquer le déséquilibre entre les paquetages: set, par exemple, contient moins de la moitié du nombre de classes de block, et plus du double de méthodes de tests. Mais le problème vient de l'étude de la couverture de test par SonarQube: celui-ci donne une couverture de test de 2.6% sur l'ensemble du projet, ce qui est extrêmement bas, d'autant plus que le nombre de méthodes de tests sur l'ensemble du projet dépasse assez largement 10000, pour un total de plus de 2000 classes, il semble donc incroyable de penser qu'autant de méthodes de tests puissent couvrir si peu du code.

Pour ce qui est du déséquilibre entre les rapports nombre de méthodes de tests - nombre de classes, il semble raisonnable d'imaginer que celui-ci est lié à la présence des plus grosses god classes dans set, qui vont donc nécessiter bien plus de tests qu'un grand nombre de petites classes. En l'occurrence, block contient beaucoup de ces petites classes, la plupart héritant de super-classes contenant la majorité du code, sans oublier que set contient 3 classes plus grosses que la plus grosse classe de block.

III. Interprétation des résultats et pistes d'améliorations

1. **God classes**

   - La première chose évidente à constater ici, grâce aux observations du nombre de lignes des classes, est la présence de god classes, en particulier dans block et set:

     - block: *factory.Functions* et *factory.Predicates* sont, avec respectivement 1218 et 1557 lignes, les god classes évidentes de block pour lesquelles il pourrait être intéressant de chercher des divisions en plusieurs classes;
     - set: *strategy.UnifiedSetWithHashingStrategy*, *mutable.UnifiedSet* et *mutable.primitive.ByteHashSet*, avec respectivement 2674, 2529 et 1735 lignes de code, sont les plus grosses god classes de la sous-section étudiée du projet, il semble clair que c'est vers elles qu'il va falloir se concentrer en priorité pour séparer les god classes.

   - En ce qui concerne lazy, on a aussi observé la présence de classes plus fournies que d'autres (la plus grande étant composée de 794 lignes), cependant l'écart est bien moindre par rapport à block et set, il n'est donc pas nécessaire de chercher à séparer grosses classes, et encore moins de le faire en priorité avant les god classes de block et set.

   - Il est très probable que la forte complexité cyclomatique de certaines classes soit due aux god classes: les classes avec la plus forte complexité cyclomatique totale observée se trouvent aussi être les god classes mentionnées précédemment, et comme mentionné ci-dessus, on a observé un fort lien entre la complexité cyclomatique des classes et leur nombre de lignes. En somme, séparer les god classes devrait permettre de réduire ces complexités cyclomatiques. Il est cependant possible que certaines méthodes soient inutilement complexes, auquel il pourrait être intéressant de chercher à les simplifier.

2. **Code déprécié/mort**

   - Au cours de l'étude, nous avons noté que certaines parties du code sont dépréciées. Tous les bouts de code déprécié sont utilisés dans des classes de tests correspondant aux classes où ils se situent, par conséquent ce code semble être testé et ne devrait donc pas être particulièrement dangereux pour le bon fonctionnement des classes. Ceci dit, une piste d'amélioration intéressante pourrait être de le modifier de sorte à ce qu'il ne soit plus déprécié, en utilisant par exemple des fonctionnalités à jour de Java, ce qui permettrait de rendre ce code plus sûr, et donc de limiter les potentiels bugs du projet

   - L'étude n'a pas révélé de code mort, ce qui témoigne d'un assez bon entretien du projet. Si, toutefois, durant la phase d'amélioration du projet, des parties de code mort sont repérées, il va de soi que chercher à s'en débarrasser pourrait être une amélioration intéressante, et potentiellement pas trop compliquée.

3. **Potentielles améliorations mineures**

    - Le projet semble, d'après SonarQube, être concerné par des problèmes de duplication de code. Il devrait être possible de remédier à cela, notamment par l'ajout de méthodes ayant pour but d'exécuter le code dupliqué, pour juste appeler la méthode aux endroits où devrait se trouver le code.

    - S'il est vrai que le code n'est pas surchargé de commentaires, et en particulier de documentations inutiles, il faut malheureusement noter que le code est parfois un peu léger en commentaires, et le but de certaines méthodes n'est pas forcément limpide: par exemple, dans la plus grosse god class de set, *set.strategy.mutable.UnifiedSetWithHashingStrategy*, on trouve une méthode nommée chainedAdd() dont le code s'étend sur 60 lignes, dénuée de tous commentaires et javadoc. Il est à préciser qu'il ne s'agit pas d'une méthode héritée, donc que la super classe ne possède pas non plus de documentation concernant cette méthode. Elle semble chercher à ajouter un objet de type ChainedBucket à la structure de données, et renvoyer un booléen, mais la signification du booléen renvoyé n'est pas claire. Il semble donc que rajouter des commentaires et des documentations à ce type de méthodes soit une bonne idée.

    - À certains endroits, il est possible de trouver des nombres magiques, qu'il vaudrait mieux remplacer par des constantes. Par exemple, dans la classe UnifiedSetWithHashingStrategy, l'attribut loadFactor est censé être compris entre 0 (exclu) et 1 (inclus). Ces valeurs ont du sens, mais si l'on voulait, pour une raison inconnue, changer la façon dont les calculs sont faits en fonction de ce loadFactor, on risquerait de devoir attribuer à cet attribut des valeurs différentes que celles dans l'intervalle]0,1]. À ce moment-là, il serait bien plus simple de juste avoir à modifier les valeurs auxquelles seraient assignées les constantes qui représenteraient l'intervalle de valeurs de loadFactor. Bien entendu, il est très peu probable de vouloir changer cet intervalle car il est celui qui rend les calculs avec loadFactor les plus simples possibles, mais la possibilité de la présence de nombres magiques moins sensés n'est pas à exclure.

    - Il faudra aussi se pencher sur la question des paquetages. En effet, on peut trouver, d'après IntelliJ, plusieurs dépendances cycliques entre les paquetages. Il faudra donc se pencher sur celles-ci, analyser s'il s'agit réellement de dépendances cycliques problématiques, c'est-à-dire vérifier si un fichier d'un paquetage importe un fichier d'un autre paquetage qui importe le premier fichier. Autrement, il n'y aurait pas vraiment de cycles.

    - Il sera aussi intéressant d'analyser les potentielles raisons de l'existence du paquetage *org.eclipse.collections.impl.block.comparator.primitive* qui, on le rappelle ne contient aucun fichier. Il semblerait a priori qu'il s'agisse d'un paquetage inutile et qu'on puisse purement et simplement le supprimer pour simplifier légèrement le projet, cependant il semble que le paquetage primitive relève d'une habitude (bonne ou non, à voir) de développement, puisque c'est un nom de sous-paquetage qu'on retrouve dans beaucoup d'autres paquetages, souvent accompagné du sous-paquetage checked.

IV. Conclusion

Dans l'ensemble, le projet est très complet, bien développé et bien maintenu. En effet, le ratio de dette technique estimé par SonarQube est très bas, 1.3% pour block, 0.3% pour lazy 0.7% pour set, et 1% sur l'ensemble du projet, signe que le projet a été bien entretenu. De plus, le code est clair dans le sens où les

variables, attributs, méthodes et classes ont des noms assez évocateurs pour qui travaille sur le projet, malgré la complexité du projet et la variété des structures de données rajoutées.

Cependant, le projet n'est pas parfait, et contient différentes sortes de problèmes, mineurs ou majeurs, tels que des god classes, de la duplication de code que nous allons devoir nous atteler à résoudre, au moins partiellement. Il est peu probable que nous arrivions à résoudre tous les problèmes, mais une partie d'entre eux devrait être à notre portée.