GL_analyse_logiciel_jhotdraw

Authors :

- **Mahiedine Ferdjoukh**
- **Hamza bikine**

Utilité du projet :

Le projet JHotDraw est un framework conçu pour la création d'éditeurs graphiques.

Bien qu'il s'agisse d'une bibliothèque, il ne propose pas d'exécutable en tant que tel. Néanmoins, des exemples d'utilisation sont disponibles dans le répertoire /jhotdraw-samples.

Pour exécuter un exemple, vous pouvez suivre ces étapes :

1- Placez-vous à la racine du projet.

2- Compilez l'ensemble du projet en utilisant la commande suivante :

mvn compile

Vous pouvez ensuite essayer de lancer le programme situé dans le répertoire jhotdraw-samples/jhotdraw-samples-misc/src/main/java/org/jhotdraw/samples/draw/Main.java.

Pour simplifier le processus et éviter les complications liées à l'utilisation de la commande java en ligne de commande, vous pouvez également lancer le programme directement depuis un environnement de développement intégré (IDE) tel qu'Eclipse.

Description du projet :

Le projet est accompagné d'un *readme*, cependant, il ne contient pas de instructions explicites sur la manière d'utiliser la bibliothèque.

La documentation du projet est générée à l'aide de Javadoc. Pour générer cette documentation, vous pouvez exécuter la commande suivante à la racine du projet :

mvn javadoc:javadoc

Pour consulter la documentation de l'API du projet, il suffit d'ouvrir le fichier jhotdraw-api/target/site/apidocs/index.html dans un navigateur web.

Étant donné que ce projet est conçu comme une bibliothèque, il n'est pas nécessaire de l'installer. Pour l'utiliser dans votre propre projet, vous pouvez simplement l'importer en

l'ajoutant aux dépendances de votre projet, par exemple en spécifiant ces dépendances dans le fichier POM si votre projet utilise Maven.

3 Historique du logiciel

3.1 Analyse du git

Le projet présente un historique de développement intéressant, avec deux contributeurs principaux et un bot, ainsi qu'un total de 804 commits, 2 branches et 6 tags marquant différentes versions du logiciel.

Les contributions des deux principaux contributeurs se sont déroulées à des périodes différentes. Le premier a travaillé sur le projet de novembre 2006 à février 2015, totalisant 494 commits. Le deuxième a pris le relais de février 2015 à mai 2020, avec 105 commits. On observe que le premier contributeur a eu une activité plus régulière, avec environ 55 commits par an, tandis que le deuxième a commis en moyenne 21 fois par an. Bien que les deux aient contribué de manière assez constante, la cadence du deuxième contributeur était légèrement moins soutenue.

Le développement du projet semble s'être arrêté le 22 mai 2020, après la sortie de la version 9.0 le 11 mai 2020, à laquelle ont été apportés les derniers des 804 commits.

En ce qui concerne les branches, la branche principale "develop" est utilisée pour la production, tandis que la branche "master" représente la version 9.0 avec quatre commits de moins que la branche principale.

Concernant les contributions extérieures, huit pull requests ont été soumises, provenant notamment d'étudiants de l'année précédente et du bot. Quatre de ces pull requests ont été fusionnées, tandis que les quatre autres restent ouvertes.

4 Architecture logicielle

4.1 Utilisation de bibliothèques extérieures

Dans le répértoire jhotdraw-core on trouve la bibliothéque des tests Junit.

L'analyse des bibliothèques référencées par rapport à celles effectivement utilisées révèle que certaines dépendances pourraient être considérées comme inutiles. En particulier, le fait que seul JUnit soit utilisé parmi les bibliothèques référencées indique qu'il n'y a pas trop de dépendances superflues. Cela suggère que le projet a été développé de manière relativement efficace et légère en termes de dépendances externes.

En ce qui concerne les bibliothèques réellement utilisées, l'utilisation de JUnit indique que des tests unitaires ont été mis en place, ce qui est une pratique recommandée pour assurer la qualité du code. L'absence de multiples bibliothèques servant à la même fonctionnalité, comme plusieurs ORM ou bibliothèques graphiques, suggère une approche

cohérente dans le choix des outils. Dans ce cas, l'utilisation unique de JUnit comme bibliothèque de tests unitaires est justifiée, car elle offre une solution robuste et largement acceptée dans la communauté de développement logiciel pour les tests automatisés.

Il est également pertinent de noter que l'utilisation de JUnit et le retrait de la bibliothèque "testing" suggèrent un ajustement récent dans les choix technologiques du projet. Cette transition peut indiquer une volonté d'adopter des normes de test plus standardisées et mieux soutenues par la communauté, ce qui peut améliorer la maintenabilité du code à long terme. En outre, cela pourrait refléter une volonté d'optimiser les dépendances du projet en éliminant celles qui sont redondantes ou moins utilisées, ce qui contribue à simplifier la structure du projet et à réduire sa complexité.

4.2 Organisation des packages

Il y a un total de 13 packages.

Des relations cycliques existent entre certains packages, telles que celles observées entre les packages org.jhotdraw.draw.action et org.jhotdraw.draw.

La profondeur maximale des packages est fixée à 4, comme illustré par le package org.jhotdraw.draw.action.

La structure hiérarchique des packages de test reflète celle des packages sources.

En général, les noms des packages ne fournissent pas nécessairement d'informations sur les *design patterns* utilisés. Cependant, il existe quelques exceptions notables :

- **Le package org.jhot.draw.event suggère l'utilisation du *pattern observable*.**
- **Le package org.jhotdraw.draw.decoration suggère l'utilisation du *pattern decorator*.**

En dehors de ces cas, certains packages contiennent des classes qui suivent des *design patterns* qui ne sont pas explicitement indiqués par leur nom. Par exemple, le packages org.jhotdraw.draw.figure.Figure implémente les *patterns Composite*, *Framework*, *Decorator*, *Observer*, *Prototype*, *Strategy*, entre autres. Le terme "figure" dans le nom du package ne reflète pas nécessairement l'utilisation de ces *patterns*.

De manière similaire à l'exemple précédent (org.jhotdraw.draw.figure.Figure), il serait ardu d'avoir des noms de packages toujours très explicites quant aux *design patterns* utilisés par leurs classes. Une telle approche conduirait, par exemple, à des noms de packages excessivement longs.

Pour pallier ce manque d'information, les *design patterns* implémentés sont généralement spécifiés dans la documentation Java (Javadoc) de chaque classe. Cependant, cette pratique rend la compréhension du code plus complexe, car il est nécessaire de consulter fréquemment la Javadoc.

4.3 Répartition des classes dans les packages

La diversité du nombre de classes par paquetage est notable. À titre d'exemple, le paquetage org.jhotdraw.draw.action abrite le plus grand nombre de classes, atteignant 33. Cette abondance pourrait s'expliquer par la multitude de possibilités d'"actions" qu'il offre. En contraste, le paquetage org.jhotdraw.draw.print contient seulement une classe, car la majeure partie du travail lié à l'impression est déjà pris en charge par la bibliothèque Java java.awt.print.

En ce qui concerne le nombre total de classes dans le noyau (core):

- ***TOTAL*: 193 classes**
- ***Total de classes par packages:***
- **org.jhotdraw.draw: 22**
- **org.jhotdraw.draw.action: 33**
- **org.jhotdraw.draw.connector: 10**
- **org.jhotdraw.draw.decoration: 6**
- **org.jhotdraw.draw.event: 25**
- **org.jhotdraw.draw.figure: 27**
- **org.jhotdraw.draw.handle: 25**
- **org.jhotdraw.draw.io: 7**
- **org.jhotdraw.draw.layouter: 5**
- **org.jhotdraw.draw.liner: 4**
- **org.jhotdraw.draw.locator: 7**
- **org.jhotdraw.draw.print: 1**
- **org.jhotdraw.draw.text: 2**
- **org.jhotdraw.draw.tool: 19**

Analyse approfondie

5.1 Évaluation des tests

Le projet affiche une faible couverture de tests, avec seulement deux ensembles de tests, AbstractFigureNGTest dans jhotdraw-core et BezierPathNGTest dans jhotdraw-utils. Ces tests ne représentent que 0.2% du code total, ce qui est insuffisant pour assurer une robustesse adéquate. En outre, la répartition des tests dans seulement deux des neuf répertoires du projet signifie qu'une grande partie du code n'est pas soumise à des tests unitaires.

Bien que les tests actuels réussissent tous sans erreur d'assertion, il est nécessaire d'augmenter considérablement la couverture des tests pour garantir la qualité du logiciel.

5.2 Annotations

Le projet compte un total de 22 111 lignes d'annotations, ce qui représente environ 21.5% du code. La majorité de ces annotations sont des Javadoc, avec une proportion

significative de licences. Le code lui-même est peu annoté, à l'exception de quelques sections mises en annotation.

Étant donné que le développement du projet est en pause depuis deux ans, il est peu probable que l'ajout d'annotations supplémentaires soit nécessaire à ce stade.

5.3 Obsolescence

Il y a peu de codes obsolètes dans le projet. Ceux qui existent sont principalement localisés dans quelques fichiers spécifiques. De plus, bien qu'il y ait des appels à des méthodes ou des classes obsolètes, ceux-ci semblent provenir d'imports externes et ne sont pas utilisés dans le code du projet lui-même.

5.4 Duplication de code

Le projet souffre d'un taux significatif de duplication de code, représentant 16.8% du code total. Cette duplication est présente dans plusieurs dossiers du projet, avec des fichiers entiers montrant une structure similaire et du code répété. La consolidation de ces duplications, par exemple en utilisant l'héritage ou la création de classes abstraites, pourrait grandement améliorer la maintenabilité du code.

5.5 Classes monolithiques

Le projet compte un nombre non négligeable de "classes monolithiques", avec certaines classes dépassant largement les conventions recommandées en termes de taille et de complexité. Ces classes comportent un nombre élevé de méthodes, variables d'instances et lignes de code, ce qui peut rendre la maintenance et l'extension du code plus difficiles. La répartition inégale des importations externes dans ces classes suggère une forte dépendance et une complexité accrue.

5.6 Analyse des méthodes

La complexité cyclomatique des classes varie de 1 à 499, avec une moyenne de 21 par classe, ce qui se traduit par une moyenne beaucoup moins élevée par méthode. Cependant, cela ne signifie pas nécessairement l'absence de méthodes présentant une complexité cyclomatique élevée.

Lorsque cela se produit, il est observé que le nombre de commentaires destinés à décrire les différents cas est souvent limité, principalement en raison de la prédominance de la documentation Javadoc.

Bien que certaines méthodes puissent contenir un grand nombre de lignes, en général, la longueur des méthodes n'est pas excessive. De plus, la plupart du temps, le nombre d'arguments par méthode ne dépasse pas 3, même pour les classes les plus volumineuses. C'est le cas, par exemple, de jhotdraw-samples/jhotdraw-samples-misc/src/main/java/org/jhotdraw/samples/svg/io/SVGInputFormat.java, la classe

présentant le plus grand nombre de lignes de code et la plus grande complexité cyclomatique.

Dans cette même classe, on remarque la présence de méthodes renvoyant des codes d'erreur, bien que ces méthodes déclenchent ultimement une exception dans un bloc catch. Une approche alternative pourrait consister à créer une méthode englobante qui fait appel à ces méthodes tout en gérant les exceptions, plutôt que d'inclure des blocs try dans chaque méthode individuelle.

6 Nettoyage de Code et Code Smells

6.1 Règles de nommages

Les noms de classes sont aisément prononçables et présentent une clarté suffisante. Il est rare de constater l'utilisation d'abréviations dans ces noms. De plus, les conventions de nommage Java sont respectées de manière adéquate.

Par ailleurs, les appellations semblent être des termes relevant du domaine de l'imagerie 2D et de l'édition d'image, englobant des concepts tels que "Rectangle", "Drawing", "Translation", "Rotation", "Bezier", "Perpendicular", "Figure", "Image", "View", "Editor" et autres.

Ainsi, il apparaît que le processus de nomination ne requiert pas nécessairement d'amélioration, puisque les termes choisis sont pertinents et conformes aux standards du domaine.

6.2 Nombre magique

Des occurrences de nombres magiques sont observées dans le code, spécifiquement dans org.jhotdraw.draw.decoration.PerpendicularBar.java, au sein des méthodes getDecoratorPathRadius et PerpendicularBar.

6.3 Organisation du code

Afin de simplifier l'analyse, concentrons-nous sur un seul package représentatif de l'ensemble. Prenons par exemple le package org.jhotdraw.draw.decoration. Ce package comprend 1 interface, 1 classe abstraite et 4 classes concrètes. Bien qu'il y ait quelques exceptions où des méthodes publiques sont placées après des méthodes protégées (par exemple, dans PerpendicularBar.java, AbstractLineDecoration.java), en règle générale, les méthodes publiques, plus utilisées, précèdent celles qui sont protégées / privées.

6.4 Code inutilisé

Globalement, il n'y a pas de code inutilisé. Cependant, une petite exception concerne jhotdraw-app/src/main/java/org/jhotdraw/app/AbstractApplication.java, où la méthode privée initComponents() n'est pas utilisée.

6.5 Duplication de code

Pour l'ensemble du code source de jhotdraw, SonarQube indique un taux de duplication de 16.8 %. Plus spécifiquement, pour le répertoire jhotdraw-core, le taux de duplication est de 14.7 %.