Question 1 (au niveau de la facilité d'analyse):

Effort fournis pour diagnostiquer les déficiences et causes de défaillances Ou pour identifier les partis modifier

1. DC (densité de commentaires)
    a. On pourra voir s'il y a assez de d'explication sous forme de commentaire pour le code fournis et s'il y a assez de commentaire, le code devrait en général être mieux expliquée. Il est donc important de voir la DC dans chaque classe pour avoir une idée de si le code a bien été documente à des endroit facile et accessible (aka, la classes elle-même) lorsque c'est nécessaire de consulter la documentation.

2. Nombre de classe documente/nb de class totale (on aimerait le plus que possible que le seuil soit de 100%):
    a. Pour voir si toutes les classes sont documenté (ainsi, ça permettra facilement de comprendre la tâche de la classe)
    b. Couplé avec la première métrique, on aura une bonne approximation de à quel point le code est bien documenté (car s'il y a un ratio élevé de classe documente, alors ça sera très facile d'analyser le code pour savoir ou porter des modifications lorsque nécessaire)

Question 2 (facilité de modification):

Effort nécessaire pour modifier, remédier aux défauts ou changer d'environnement

1. Regarder la cohésion entre les classes en utilisant LCOM (Lack of cohesion in methods):
    a. En calculant le LCOM, on pourra voir la cohésion entre les méthodes des classes et ainsi savoir si le code a bien été séparer de manière logique. S'il y a peu de « manque de cohésion » dans une classe on pourrait dire que les méthodes de la classe sont indépendantes les unes des autres et ainsi les modifications apportées ne risquerons pas de briser le reste du code.
2. Egon:
    a. On pourra vérifier si la classes ne fait pas trop de chose qu'elle ne devrait peut-être pas faire, car si elle est associée a d'autre classes beaucoup de fois ça peut être un indicateur que la classe fait beaucoup de chose et donc elle se fait appeler plus souvent.

En utilisant les 2 métriques plus haut ensemble, on pourra avoir une bonne idée de la qualité du code (chaque classe/ méthodes fait sa propre tache), se qui nous donnera une bonne garantie que le code ne risque pas s'arrêter de fonctionner s'il y a une modification à apporter quelque part.

Question 3 (stabilité):

Risque des effets inattendus des modifications

1. NEC (number d'erreurs/avertissements):
    a. Si le calcul du nombre d'erreur nous donne un petit nombre, on pourra en conclure que l'implémentation a été fait de sorte à prendre en compte toutes les possibilités et les cas spéciaux nécessaires et montre donc la maturité du logiciel. C'est rentable, car c'est très facile à exécuter et ça peut potentiellement réduire les couts futures lies a des avertissements ignores.
2. PMNT (pourcentage de méthodes non testées):
    a. Si on calcul le pourcentage de couverture des méthodes, un pourcentage trop élevé nous indiquerait que les tests n'ont pas été effectue sur toutes les classes et méthodes. Ainsi, c'est rentable, car on peut vérifier visuellement quel pourcentage du code représente bien les règles d'affaires.

Question 4 (facilité de test):

Effort nécessaire pour valider le logiciel modifié

1. PMNT (pourcentage de méthodes non testés)
    a. Le pourcentage va nous permettre de vérifier quel pourcentage du code répond déjà aux critères d'acceptations. Si le pourcentage est élevé donc la validation par les tests d'un ajout future ne sera pas représentative de la facilite de test du logiciel globale.
2. Complexité cyclomatique
    a. La complexité cyclomatique représente les interactions d'une méthode avec d'autres. Lorsque la CC est élevée, il faut prendre en compte ces interactions dans les tests (en créant des moques ou en les appelant directement)
3. CNT (Classe non teste) :
    a. Ça nous permettra de voir s'il y a assez de test pour toutes les classes, s'assurant ainsi que la majorité les cas possibles soient couverts et éviter ainsi des situations inattendues.

La combinaison des 3 métriques de test nous donnerons alors une plus grande confiance sur le niveau de fonctionnement du logiciel et a quel point il peut fonctionner comme prévu, car si nous avons un bas PMNT et bas CNT, on aura une plus grande garantie de la qualité de notre logiciel.

Réponse aux questions:

Q1 : En tenant compte de la taille du projet et des sorties pour les 2 métriques (DC et nb classe documenté (DC > 1/5) /nb classes tôt), nous en concluant ce projet est adéquatement documente car pour un si gros projet, la présence de commentaire expliquant l'utilité d'une classe est primordiale. Nous avons pu le voir avec le la deuxième métrique, que presque la totalité des classes sont suffisamment commenté et donc qu'il y a assez d'explication dans chaque classe.

Nom de la classe (faisant partit du Egon10)	Lcom
Crosshair	0. 847972972972973
TextUtils	0.9393939393939394
SerialDate	0.9936886395511921
SerialUtils	0
XYSeries	0.8492063492063492
LegendItem	0.8974358974358975
NumberAxis	0.952020202020202
TimeSeries	0.8328840970350404
BarRenderer	0.9454365079365079
Plot	0.9553921568627451
CategoryAxis	0.9327731092436975
ValueAxis	0.960261569416499
JFreeChart	0.9339285714285714
Axis	0.9632034632034632
CategoryPlot	0.9761584454409566
XYPlot	0.9758513091846425

Q2 : Tout d'abord pour l'Egon, ça fonctionne comme pour le tp1 et on peut voir dépendamment du seuil donne, les classes suspecte d'être des classes divines. Pour le seuil de 10%, nous avons 16 classes suspecte d'être des classes divines et même si ça ne parait pas beaucoup avec la quantité de classe totale se trouvant dans le dossier, ça n'en n'est pas plus un indicatif qu'il y a peut-être des classes qui font beaucoup de chose alors que certaines n'en font presque pas. Pour se qui en ait de LCOM, nous avons trouvé une médiane de Lcom = 0.5857831 (calcule en Excel après avoir extrait tous les lcom du fichier csv). Donc déjà, on peut voir qu'au moins la moitié des classes ont un Lcom > 0.5 ce qui est plutôt mauvais signe car plus cette valeur s'approche de 1 et moins il y a de cohésion dans la classe. En combinant nos 2 métriques, on peut voir la majorité des classes se trouvant dans Egon10 ont presque aucune cohésion (montre dans le tableau ci-dessous). Et avec ça, nous avons conclu que la conception n'est pas bien modulaire.

Q3 : Pour un total de 91736 nloc, on obtient 103 warnings count. Lorsque le NEC est bas on est moins inquiets des effets de bords d'une modification ou d'un ajout au code. Toutefois, dans notre cas on considère que 103 c'est un chiffre assez important, ce qui pourrait causer l'apparition de d'autres erreurs/warnings en cas d'ajout ou bien de modification du projet et donc la maintenabilité du code est plus difficile. (Voir image cycTotal). Il est en est de même pour le pourcentage de couverture des méthodes on a 8140 méthodes et 3049 non testes ce qui nous donne un pourcentage de 37.45%. ce qui est un pourcentage acceptable toutefois on essaye toujours de réduire au maximum ce chiffre. D'autant plus avec mvn on obtient le pourcentage des instructions manquées ce qui est de 54%, ce qui dépasse 50% et pour répondre à la question le code est mature due à ces mesures (https://github.com/Tassa06/Ift3913_Tp2/blob/c145613ec148d83d89a80ef8675ff829a5b8a05a/jfreechart-master/jfreechart-master/target/site/jacoco/index.html) ou bien le code ajoute n'a pas été globalement teste.

Q4 : Pour la facilite des tests on a pris en compte la complexité cyclomatique, le pourcentage de méthode testées et les tests par classe. Pour la complexité cyclomatique on a obtenue un CCN moyenne de 2.3 quand on passe en argument le dossier src/main de jfreechart-master la moyenne est plus basse que la valeur de CC par défaut mise pour mesurer les CC. La moyenne trouve est relativement basse. Toutefois, quand on regarde de pres les classes on retrouve certaines dont certaines méthodes dépassent la limite par défaut (celle-ci sont soulignées en rouge dans le rapport généré cyc.html). Pour ce qui des pourcentages de méthodes non teste on a 37.45%, d'instructions manque de 54% et de branches manquées est de 45%. Bien que 37.5% est plus bas que 50%, on considère que c'est un pourcentage qui est assez élevé, surtout mis en parallèle avec le pourcentage d'instructions manque non teste qui est assez élevé, ca nous indique que seulement 46 % des instructions ont été teste et seulement 62.55% des méthodes ont été teste. Compte tenu de la grosseur du projet, ces résultats nous indiquent clairement que le code ne peut être teste automatiquement. Pour CNT on a 541 classes et 88 n,ont pas été teste ce qui donne un pourcentage de 16.26% de classe non teste. Bien que le pourcentage est bas, certaines classes sont dépendantes de d'autres ainsi, une classe teste qui utilise un objet d'une classe non teste pourrait créer des erreurs/warnings.

Conclusion :

Avec toutes les metriques utilisees, ainsi que l'analyse des résultats obtenu. On conclue que la maintenabilité du code est difficile au point de vue du chef du projet et selon les normes de ISO25010 et que le code est sensible au modifications futures.

Lien vers le repertoire git : https://github.com/Tassa06/Ift3913_Tp2.git