Ahallouch Kaoutare M18
MASTROLLILI Bob M18
SOULEIMAN Ilsan M18

Création d'un outil d'analyse de code (God class)

UE : Génie logiciel et conduite de projets informatiques 2
(*AA : Techniques et conduite des tests logiciels*)

Date de rentrée du rapport : 19 mai 2016

Année académique : 2014-2015

Table des matières

Chapitre 1

Définitions

1.1 God Class

Avant de débuter ce travail, il parait nécessaire de définir ce qu'on entend par *God Class*. Les *God Class* sont des classes qui ont trop de responsabilités ce qui va à l'encontre de la programmation orientée objet. Les *God Class* sont caractérisées par un nombre important de méthodes, d'appels à des méthodes extérieures ou d'accès à des variables membres. Afin de mettre en évidence ce type de classes, différents indicateurs peuvent être utilisés.

1.2 Indicateurs de God Class

Nous avons décidés d'utiliser principalement trois indicateurs : le WMC, le TCC et l'AFTD qui seront explicités d'avantage ci-dessous.

1.2.1 WMC

Le *Weighted Method Count* (WMC) est un métrique qui permet de mesurer la complexité d'une classe. Pour ce faire, ce critère peut se baser notamment sur le nombre de méthodes d'une classe ou la complexité cyclomatique de MCCabe peut être utilisée. Pour cet indicateur, une valeur faible est préconiséee. Mais qu'entend-on par *faible* ? Dans la littérature, certains auteurs considèrent comme faible une valeur entre 20 et 50 d'autres s'accordent sur une valeur de 10. Mais quoiqu'il en soit, une valeur supérieure à 50 est à éviter. En effet, la maintenance d'une classe ayant plus de 50 méthodes est très difficile ainsi que sa lisibilité et son refactoring.

1.2.2 TCC

Le *Tight Class Cohesion* (TCC) est un métrique qui permet de mesurer la cohésion d'une classe. Cet indicateur s'appuie sur l'analyse des relation entre méthodes. Il se situe généralement entre 0 et 1. La détection d'une classe avec peu de cohésion permettra, par exemple, de déterminer si une classe à besoin ou non d'être restructurée en deux autres classes. Pour cet indicateur, une haute valeur est préférable.

1.2.3 AFTD

L'*Acces to Foreign Data* (AFTD) est un indiacteur qui permet d'analyser les accès à des méthodes extérieures par voies indirectes ou directes. Il s'agit d'un indicateur qui renseigne sur le degré d'encapsulation d'une classe.

Chapitre 2

Consignes

2.1 Enoncé

Consignes générales

- *Travail a effectué par groupe de 3. Chaque groupe choisit un thème. Pour chaque thème, les explications complémentaires nécessaires pour la réalisation du travail vous seront fournies.*
- *Utilisation d'un serveur d'intégration*
- *Utilisation d'au moins une technique de tests en boîte noire et une technique de tests en boîte blanche (vue au cours ou autre)*
- *Remise d'un rapport explicitant les objectifs fixés pour les tests, la ou les méthodologie(s) utilisée(s) ainsi que les résultats obtenus.*
- *Prises d'initiatives requises pour la réalisation du projet en respectant toutefois les quelques contraintes imposées !*

Thème 2 : Création d'un outil d'analyse de code

Ecrire une application permettant de détecter des classes qualifiées de "God class". Une telle classe a trop de responsabilités et accède généralement directement aux données d'autres classes. La réutilisabilité et la facilité de compréhension de cette classe n'est dès lors pas facile. Une bonne pratique de programmation orientée objet veut que l'on attribue au plus une seule responsabilité à une classe. Les métriques utilisés pour détecter une "God class" sont (cf. ouvrage Object-oriented Metrics in Practice de Stéphane Ducasse et Michèle Lanza)

- *Mesure de la complexité de la classe : Weighted Method Count (WMC) http ://www.aivosto.com/project/help/pm-oo-ck.html*
  1. *Le nombre de méthodes dans une classe ou*
  2. *La somme des complexités statiques de toutes les méthodes d'une classe. Dans ce dernier cas, la complexité cyclomatique de McCabe peut être utilisée comme mesure de la complexité.*
- *Mesure de la cohésion d'une classe : Tight Class Cohesion (TCC) : TCC) http ://www.aivosto.com/project/help/pm-oo-cohesion.html Le métrique sert à évaluer la cohésion d'une classe. Sa valeur est située en 0 et 1. Il correspond au nombre relatif de pairs de méthodes d'une classe qui accède au moins à un même attribut de cette classe. En dessous de 1/3, on considérera que la classe a peu de cohésion. Votre solution doit permettre l'utilisation d'un autre métrique de cohésion.*
- *Indicateur du respect de l'encapsulation des autres classes : Access to Foreign Data (ATFD) : représente le nombre de classes extérieures auxquelles la classe accède les attributs (de manière directe ou par les accesseurs). Pas plus que quelques-uns (à définir) comme limite.*

Les classes que l'on cherche à détecter dans un projet pour lequel on se propose d'étudier le code sont donc celles qui sont trop complexes, avec peu de cohésion et qui accèdent directement aux variables membres des autres classes. Remarque : cela ne signifie pas que la classe détectée est mal conçue mais simplement qu'il y a là un indicateur qui nous invite à pousser la réflexion quant à la conception du code.

2.2 Outils utilisés

Pour l'élaboration de ce travail, nous avons , d'une part, utilisé un serveur d'intégration continue Jenkins mais également les librairies JUnit et Mockito. Pour les exemples d'utilisation de ces outils Cfr. le chapitre 4

Chapitre 3

Méthodologie de travail

Avant de débuter le projet et la partie implémentation, une phase de recherche a débuté. Pour optimiser notre temps de travail, nous nous sommes répartis les recherches. Ces dernières concernaient essentiellement les outils (Jenkins, JUnit, Mockito) et les concepts théoriques (indicateurs, doublures ...) que nous avons été amenés à utiliser. Chaque membre du groupe a ainsi acquis le rôle de référant dans son domaine de recherche.

Une fois la phase de recherche terminée, la phase de mise en commun a alors pu débuter. Il était impérativement nécessaire que chaque membre du groupe est les mêmes connaissances concernant tous les concepts théoriques. De plus, cette phase a permis à chacun de donner son point de vue sur la manière à suivre.

Enfin, une fois cette phase de mise en commun passée, la phase d'implémentation à proprement dit a débutée. Chaque membre du groupe a implémenté un indicateur et les tests correspondants. L'implémentation s'est faite en parrallè. L'intégration de toutes les unités de travail au sein d'un seul et même projet s'est faite de manière assez aisée étant donné que la marche à suivre a été décidée au départ.

Chapitre 4

Implémentation

Le projet se pésente sous la forme de quatre packages ; un package pour chaque indicateur et un package pour le gui. Les trois packages concernant les indicateurs ont pu être implémentés de manière indépendante étant donné que l'objet passé en paramètre est une *japa.parser.ast.CompilationUnit*. cette manière de procéder permet de ne parser qu'une seule fois le code de la classe. En effet, le résultat du parsing est une *japa.parser.ast.CompilationUnit*. Tous les caluls de métriques s'effectuent à partir de cet objet. Cette indépendance au niveau du code des package a permis à chacun d'effectuer ses tests unitaires de manière isolée.

4.1 Package wmc

Les méthodes qui sont testées sont les méthodes de la classe *WMCAnalyseur* et de la classe *WMCCalculateur*. Pour les premières méthodes testées, il s'agit d'une analyse par valeurs limites. Pour les intervalles dont il est question dans la figure 4.1, trois valeurs sont testées ; les valeurs aux extrêmités et une valeur au centre de l'intervalle. Ce qui donne en tout 9 tests ; trois par chaque intervalle. En outre, trois autres tests sont effectués à l'aide mocks. Le test de la figure 4.3 permet de tester que la méthode *getResult* appelle bien la méthode *getMetrique*. Ce qui l'utilité première d'un mock (vérification des appels entre méthodes). La figure **??** montre également l'initialisation avec un mock aus sein de la classe de test.

4.1.1 Extraits de code

```
public String analyseurWMC(int metrique)
{
    if (metrique >= 0 && metrique <= 10){
        return "Classe à complexité cyclomatique faible [+]";
    }
    else if (metrique >= 11 && metrique <= 30){
        return "Classe à complexité cyclomatique moyenne [+/-]";
    }
    else{
        return "Classe à complexité cyclomatique élevée [-]";
    }
}
```

Fig. 4.1 – Intervalles de classification pour le WMC

```java
@Before
public void setUp()
{
    this._analyseur = new WMCAnalyseur();

    //Création du mock
    this._persistanceMock = mock(IWMCMockAnalyseur.class);

    //Injection du mock
    this._analyseur.setPersistance(_persistanceMock);
}
```

FIG. 4.2 – Initialisation d'un mock

```java
@Test
public final void testResultatWMC3()
{
    System.out.println("resultatWMC [résultat : appel getMetrique ok]");

    this._analyseur.getResult(_metrique1);

    verify(this._persistanceMock).getMetrique();
}
```

FIG. 4.3 – Exemple d'utilisation d'un mock

4.2 Package aftd

Concernant ce package les tests effectués vont par paires. En effet, pour chaque méthode de la classe *MeusureEncapsulationAFTD* testée il y a à chaque fois un test qui donne une valeur correcte et un autre qui donne une erreur.

4.2.1 Extrait de code

```java
/*********************************************************************************
                        TEST MESURER
*********************************************************************************/

    /**
     * RESULT NULL
     */
    @Test
    public void testMeusurer() {
        System.out.println("meusurer_retour NULL");
        CompilationUnit cu = null;
        MeusureEncapsulationAFTD instance = new MeusureEncapsulationAFTD();
        double expResult = 0.0;
        double result = instance.meusurer(cu);
        assertEquals(expResult, result, 0.0);
    }

    /**
     * RESULT 2
     */
      @Test
    public void testMeusurer_2() throws ParseException, IOException {
        System.out.println("meusurer_retour 3");
        CompilationUnit cu  = JavaParser.parse(new File("C:\\Users\\PC\\Documents"
                + "\\GitHub\\God-Class\\GodClassApplic\\test_AFTD.java"));

        MeusureEncapsulationAFTD instance = new MeusureEncapsulationAFTD();
        double expResult = 2;
        double result = instance.meusurer(cu);
        assertEquals(expResult, result, 0.0);
    }
```

FIG. 4.4 – Paire de test de la méthode *meusurer()*

4.3 Package gui

Une God Class est détectée si les trois indicateurs indiquent su-imultanément qu'il s'agit d'une God Class. Le résultat est affiché au niveau du gui. Un exemple d'application du gui est donné à la figure 4.5.

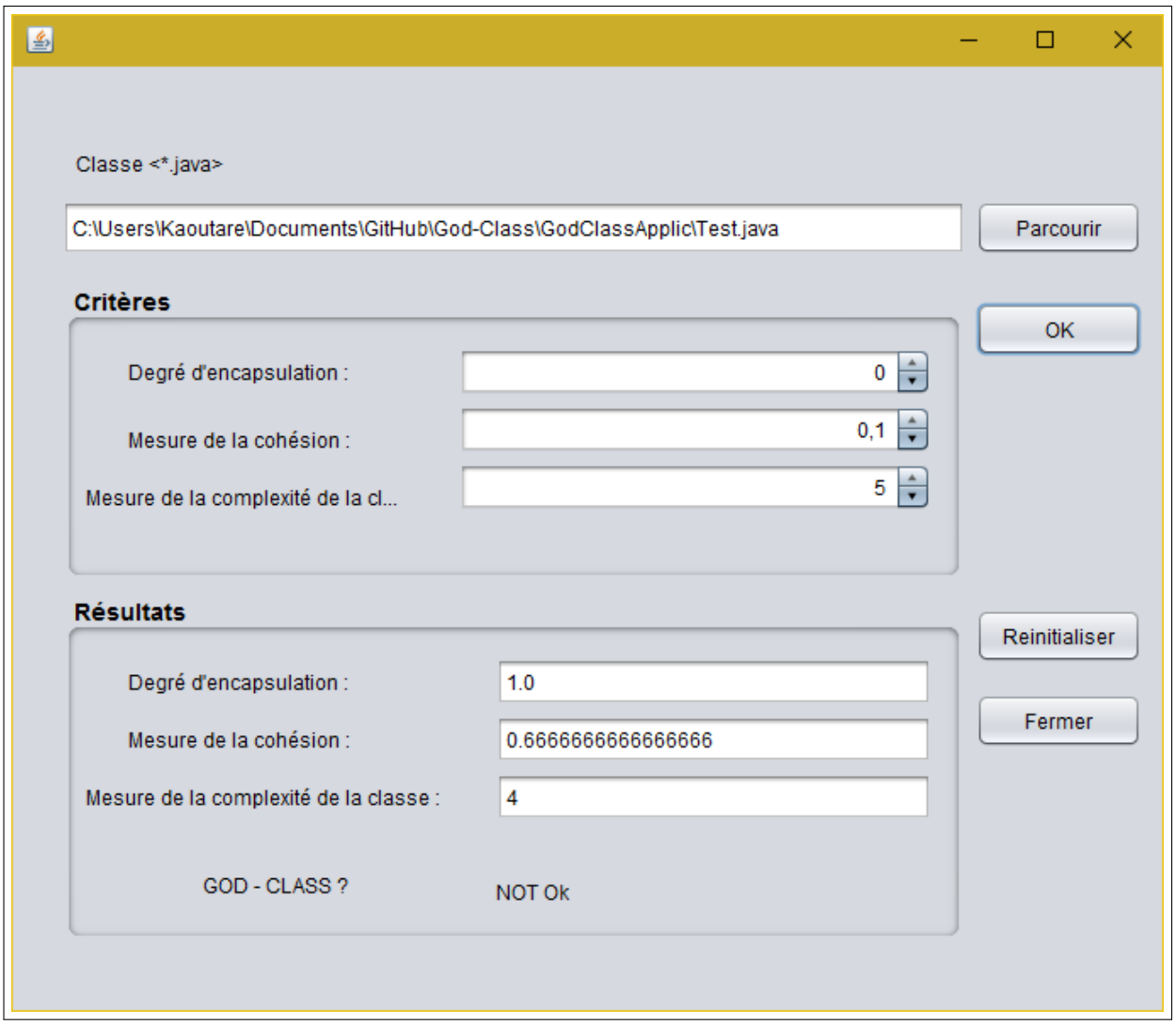

Fig. 4.5 – Exemple d'application

4.4 Diagrammes UML

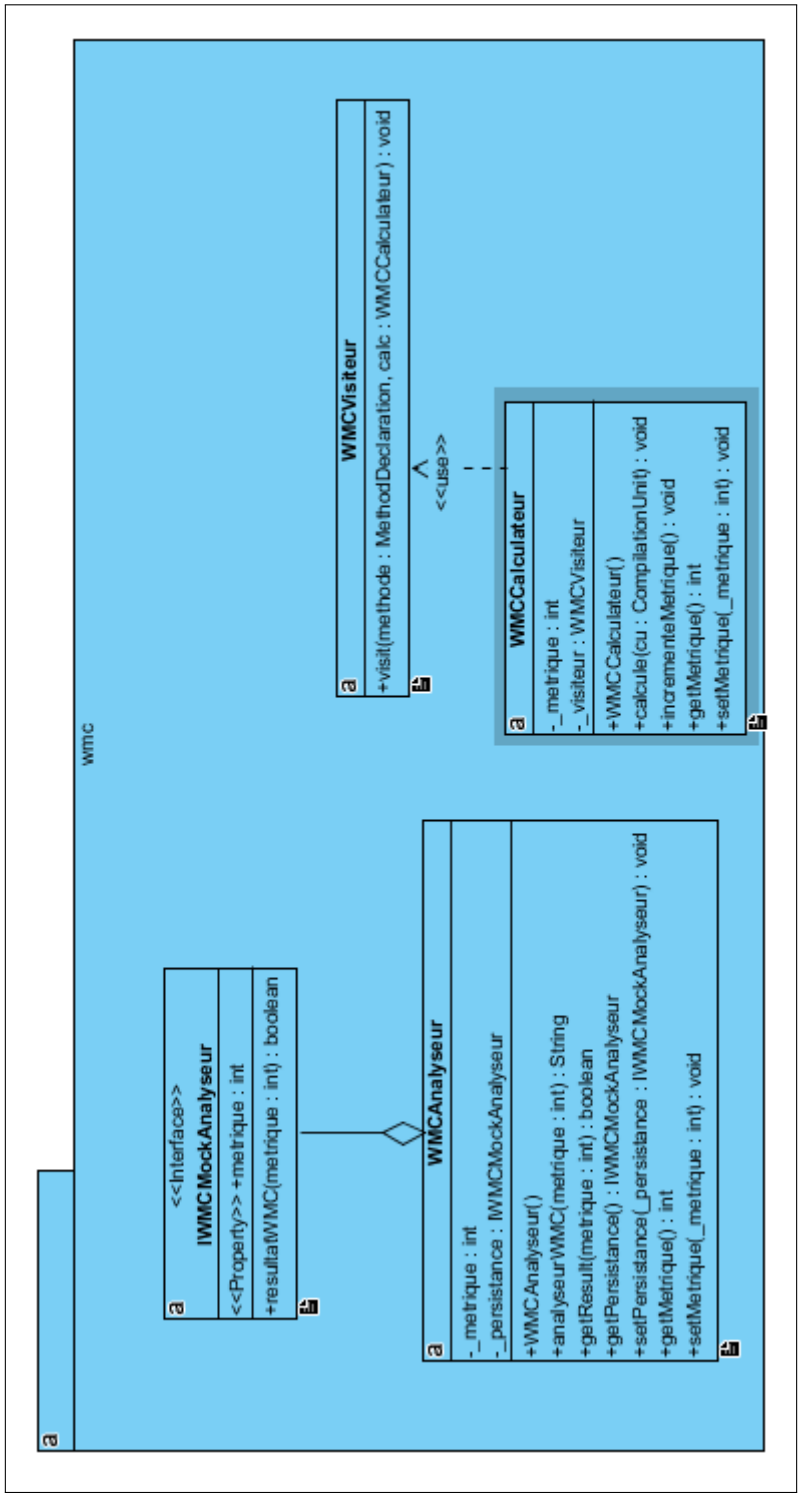

FIG. 4.6 – Diagramme UML du package wmc

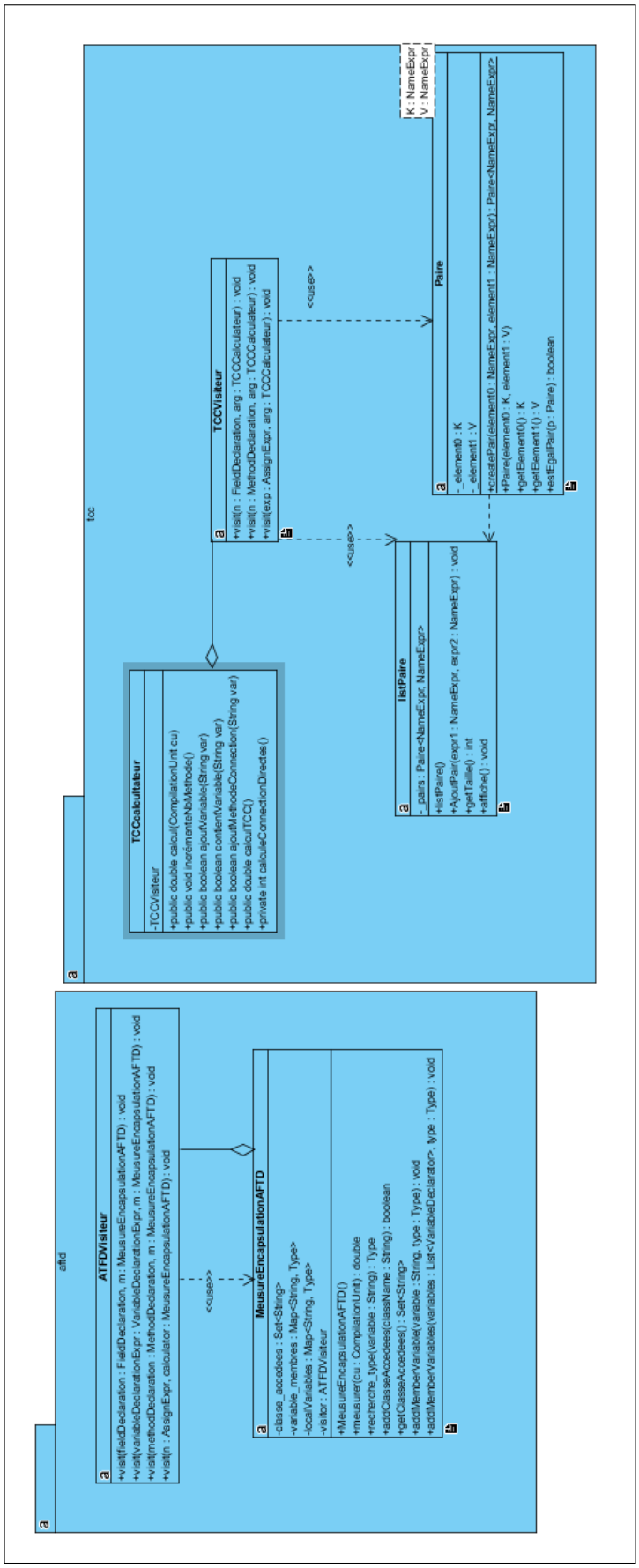

FIG. 4.7 – Diagramme UML des packages aftd et tcc

Chapitre 5

Conclusion

Avec l'expérience de programmation que nous avons acquis jusqu'à présent, nous avons toujours eu l'habitude d'écrire le code avant d'écrire des tests pour tester notre code. La plupart du temps, nous n'avions pas recours à des librairies comme JUnit qui génèrent la structure des tests à effectuer. Nous effections les tests au sein d'un main. Cependant, pour l'élaboration de ce travail, il nous a été demandé de fonctionner en TDD (*Test Driven Development*). Si, en soit, la théorie à sujet paraissait claire l'adaptée en pratique a été assez difficile dans un premier temps. En effet, cela allait à l'encontre de notre façon de programmer habituelle. Néanmoins, après moult recherches sur le sujet, nous avons enfin résussi à appréhender cette nouvelle façon de programmer. En fait, le TDD consistait à partir d'une méthode vide et à la compléter en fonction des résultats attendus et implémentés au niveaux des tests. Plus le nombre de tests augmentait plus la méthode était étoffée. Cette manière de procéder évite grandement les erreurs de programmation, puisqu'on focntionne étape par étape et qu'en cas d'erreur nous savons localiser aisément l'erreur. C'est là un des avantages du TDD.

Chapitre 6

Annexes

CAHIER DES CHARGES TP FINAL GÉNIE LOGICIEL

Contexte

Les classes que l'on cherche à détecter dans un projet pour lequel on se propose d'étudier le code sont donc celles qui sont trop complexes, avec peu de cohésion et qui accèdent directement aux variables membres des autres classes.

Contraintes fonctionnelles

Pour les classes importantes, il faut une classe de test unitaire générée avec jUnit. Si les méthodes prennent comme paramètres des classes qui ne sont pas encore développées, il faudra utiliser un mock ou un stub (documentation voir Kaoutare). Les classes de test devront être push sur le git, et prises en compte par le serveur d'intégration Jenkins (documentation voir Bob). L'application contiendra 4 packages de base : un pour le calcul de chaque critère/indicateur (documentation et un pour le gui. Pour chaque partie effectuée, un diagramme UML et une explication doit être remise dans le fichier « Rapport ».

Conventions de nommage.

Tous les noms devront être en français, les packages en minuscule et avec des noms explicites. La séparation entre deux mots devra se faire avec une majuscule. Les variables membres devront commencer par un underscore.

Répartition des tâches

1. Critère TTC
2. Critère ATFD + gui
3. Critère WMC + rapport

Délais

Mardi 10/05/2016 : Présentation de la logique et création de la structure et des tests
Mardi 17/05/2016 : Présentation du projet fini.
Jeudi 19/05/2016 : Présentation du rapport.

Documentation

Indicateurs de God Class

Métrique de cohésion

- TCC :
  Tight Class Cohesion, est le nombre de méthodes publiques directement connectées (qui s'appellent directement) au sein d'une classe, divisé par le nombre maximum de connexions possibles entre

- LCOM1, LCOM2 :
  Il s'implémente comme si dessous.
  Prenez chaque pair de méthode dans une classe. Si elles accèdent au variable membres disjointes, incrémenter P de 1. Si elles partagent au moins une variable, incrémenter Q de 1. Le critère s'exprime comme si dessous.
  LCOM1 = P - Q ,si P > Q sinon LCOM1 = 0
  LCOM1 = 0 indique une classe cohésive
  LCOM1 > indique que la classe n'est pas cohésive puisqu'il n'y a pas de méthode utilisant les variables membres.

- LCOM4 :
  Est le critère le plus recommandé. LCOM4 mesure le nombre d'ensemble connectés. Un ensemble connecté est un ensemble contenant des variables membre et des méthodes qui sont liées. Il devrait y avoir qu'un seul ensemble dans une classe s'il y en a plus la classe manque de cohésion.
  Quelles méthodes sont liées ? Soient les méthodes a et b, elles sont liées si

  1. Elles utilisent la même variable membre
  2. A appelle b ou b appelle a

  - LCOM4=1 indicates a cohesive class, which is the "good" class.
  - LCOM4>=2 indicates a problem. The class should be split into so many smaller classes.
  - LCOM4=0 happens when there are no methods in a class. This is also a "bad" class.

Détail d'implémentations :
https://github.com/cleuton/jqana/wiki/LCOM4-Calculation

Métrique de complexité

WMC :
Weighted Method Count, est la mesure de la complexité d'une classe. Cette complexité peut être calculée comme étant :

- le nombre de méthodes dans une classe (WMC) Il faut qu'elle soit minimale.
- Ou la complexité cyclomatique

Les limites : 20 ou 50

Détail d'implémentations :
Il faut analyser chaque fonction, operateur et accesseur. Le constructeur et les event handler compte aussi comme méthode. Les méthodes héritées ne comptent pas. Il suffit alors de sommer le nombre de méthodes ou de pondérer les termes de la somme par la complexité cyclomatique de chaque méthode.

Métrique pour l'encapsulation

Indicateur du respect de l'encapsulation des autres classes : Access to Foreign Data (ATFD) : représente le nombre de classes extérieures auxquelles la classe accède les attributs (de manière directe ou par les accesseurs). Pas plus que quelques-uns (à définir) comme limite.

Parser java

Javadoc : http://www.javadoc.io/doc/com.github.javaparser/javaparser-core/2.4.0

Pour chacune des expressions suivantes il existe une visitor. Pour l'implémenter il faut hériter de la classe VoidVisitorAdapter qui implémente l'interface VoidVistor. Cet interface va contenir la déclaration des visitors de tout les types de nœuds. La liste des nœuds possibles se trouve dans :
https://github.com/ftufek/javaparser/tree/master/src/japa/parser/ast/visitor

Voici une liste de nœud non exhaustive qui pourrait nous être utiles.

```
public final class AssignExpr extends Expression {

    public static enum Operator {
        assign, // =
        plus, // +=
        minus, // -=
        star, // *=
        slash, // /=
        and, // &=
        or, // |=
        xor, // ^=
        rem, // %=
        lShift, // <<=
        rSignedShift, // >>=
        rUnsignedShift, // >>>=
    }

    private Expression target;

    private Expression value;

    private Operator op;
```

FieldAccessExpr représente les getter et seter d'une class

```java
public final class FieldAccessExpr extends Expression {

    private Expression scope;

    private List<Type> typeArgs;

    private String field;

    public FieldAccessExpr() {
    }

    public FieldAccessExpr(Expression scope, String field) {
        this.scope = scope;
        this.field = field;
    }
```

Les Appels de méthodes

```java
public final class MethodCallExpr extends Expression {

    private Expression scope;

    private List<Type> typeArgs;

    private String name;

    private List<Expression> args;

    public MethodCallExpr() {
    }

    public MethodCallExpr(Expression scope, String name) {
        this.scope = scope;
        this.name = name;
    }

    public MethodCallExpr(Expression scope, String name, List<Expression> args) {
        this.scope = scope;
        this.name = name;
        this.args = args;
    }
```

Les conditions

```java
*/
public final class ConditionalExpr extends Expression {

    private Expression condition;

    private Expression thenExpr;

    private Expression elseExpr;

    public ConditionalExpr() {
    }

    public ConditionalExpr(Expression condition, Expression thenExpr, Expression elseExpr) {
        this.condition = condition;
        this.thenExpr = thenExpr;
        this.elseExpr = elseExpr;
    }

    public ConditionalExpr(int beginLine, int beginColumn, int endLine, int endColumn, Expression condition, Expression thenExpr, Expressio
        super(beginLine, beginColumn, endLine, endColumn);
        this.condition = condition;
        this.thenExpr = thenExpr;
        this.elseExpr = elseExpr;
    }
```

JUnit

C'est un module intégré dans netbeans. Il sert au TDD (test driven developpement) . Comment ça marche ? Lorsqu'on crée une classe java, on écrit que le squelette de la classe avec ses méthodes. Ensuite on génère, grâce à JUnit, une classe test décidé à notre classe. Dans cette classe de test nous retrouvons les méthodes test qui vont instancier notre classe et tester ses méthodes en fonction des paramètres qu'on aura choisis. Ainsi on peut tester les valeurs qui risquent de faire planter notre méthode.

Voici un tutoriel vidéo très bien fait pour JUNIT sous netbeans : https://www.youtube.com/watch?v=Q0ue-T0Z6Zs