



HAUTE ÉCOLE DE LA PROVINCE DE LIÈGE

GÉNIE LOGICIEL ET CONDUITE DES PROJETS
INFORMATIQUES

DOSSIER DE GÉNIE LOGICIEL

Rapport du dossier final

Étudiant :
MAWET Xavier

Professeur :
MADANI Mounawar

28 août 2015

Table des matières

Étude sur les « God Classes/Objects »	2
1.1 Définition	2
1.2 Détection et métriques (normalisés) d'un logiciel	2
Étude des différentes technologies à utiliser	4
2.1 Outil de versionning : Git et Gitlab	4
2.2 Outil de construction (build) de projet : Maven	5
2.2.1 Project Object Model : POM	5
2.2.2 La gestion des dépendances	6
2.2.3 Exemple de fichier POM	7
2.3 Outils de test	8
2.3.1 Outil de tests unitaires : JUnit	8
2.3.2 Outil de (rapport de) couverture de code : JaCoCo	9
2.3.3 Outils pour les tests fonctionnels	10
2.4 Outil d'intégration continue : Jenkins	11
2.4.1 Rapport de couverture de code sur Jenkins avec JaCoCo	13
TDD : Le développement piloté par les tests	15
Patrons de conception utilisés	17
4.1 Observer	17
4.1.1 Le DragDropPanel	17
4.1.2 Le NumberSpinner	21
4.2 Adapter	23
4.3 Visitor	23
4.4 Médiateur	25
Autres diagrammes	28
5.1 Diagramme des différents packages	28

Étude sur les « God Classes/Objects »

1.1 Définition

Une classe God est définie comme étant une classe qui a tendance à centraliser toutes les fonctionnalités du système/logiciel. Cette conception va à l'encontre du principe « Orienté Objet » qui, à l'inverse, préconise de répartir les fonctionnalités entre les classes.

Une telle conception peut poser des problèmes plus tard, lors de l'ajout de nouvelles fonctionnalités ou lors de la correction de bugs lors de la maintenance du logiciel.

L'idée serait donc d'avoir un outil pour évaluer la qualité des classes, afin de déceler les parties du logiciel qui ne respectent pas les « bonnes » règles de conceptions des langages orientés objets.

1.2 Détection et métriques (normalisés) d'un logiciel

La détection d'une classe God peut se faire en testant si :

- la classe utilise directement des attributs d'autres classes
- la complexité de la classe est très grande
- la cohésion de la classe est basse

Selon les travaux de Radu Marinescu, la détection d'une classe God (ou plus généralement d'une mauvaise conception orientée objets) consiste à utiliser un ensemble de métriques (logiciels) statiques et des conditions logiques. Les métriques concernés sont les suivants :

- **ATFD** : Access to Foreign Data, est le nombre d'accès que fait une classe aux attributs d'une autre classe (directement ou via l'utilisation d'accesseurs)
- **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) ou
 - la somme des complexités cyclomatiques de **McCabe** de toutes les méthodes d'une classe (MCC)
- **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 les méthodes de cette classe.

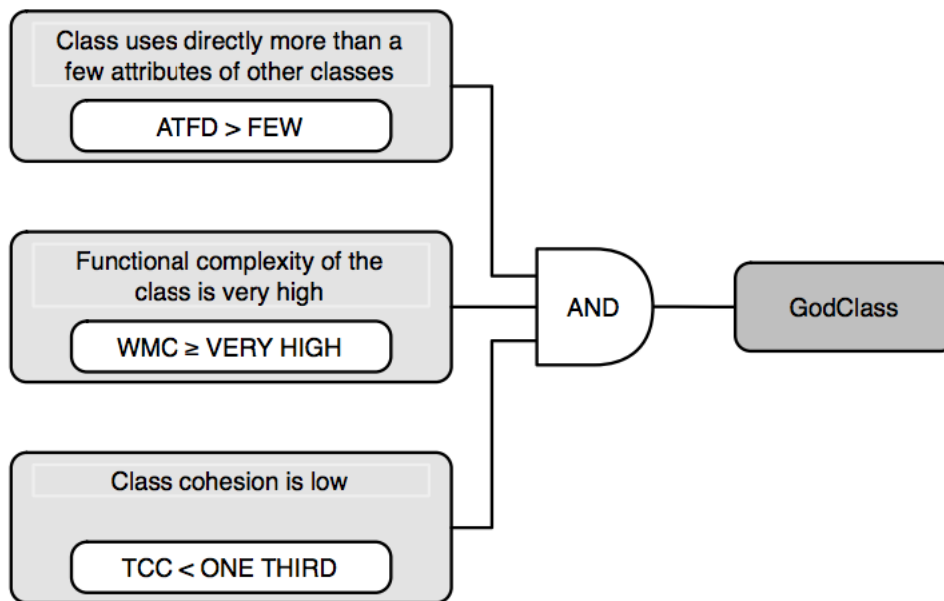


FIGURE 1.1 – Détection d’une classe God

Il ne reste plus qu’à déterminer les valeurs (seuils) de **FEW** et **VERY HIGH**.

Comme je trouvais cette illustration très parlante, j’ai décidé de l’intégrer dans l’application graphique que j’ai développé dans le cadre de ce laboratoire :

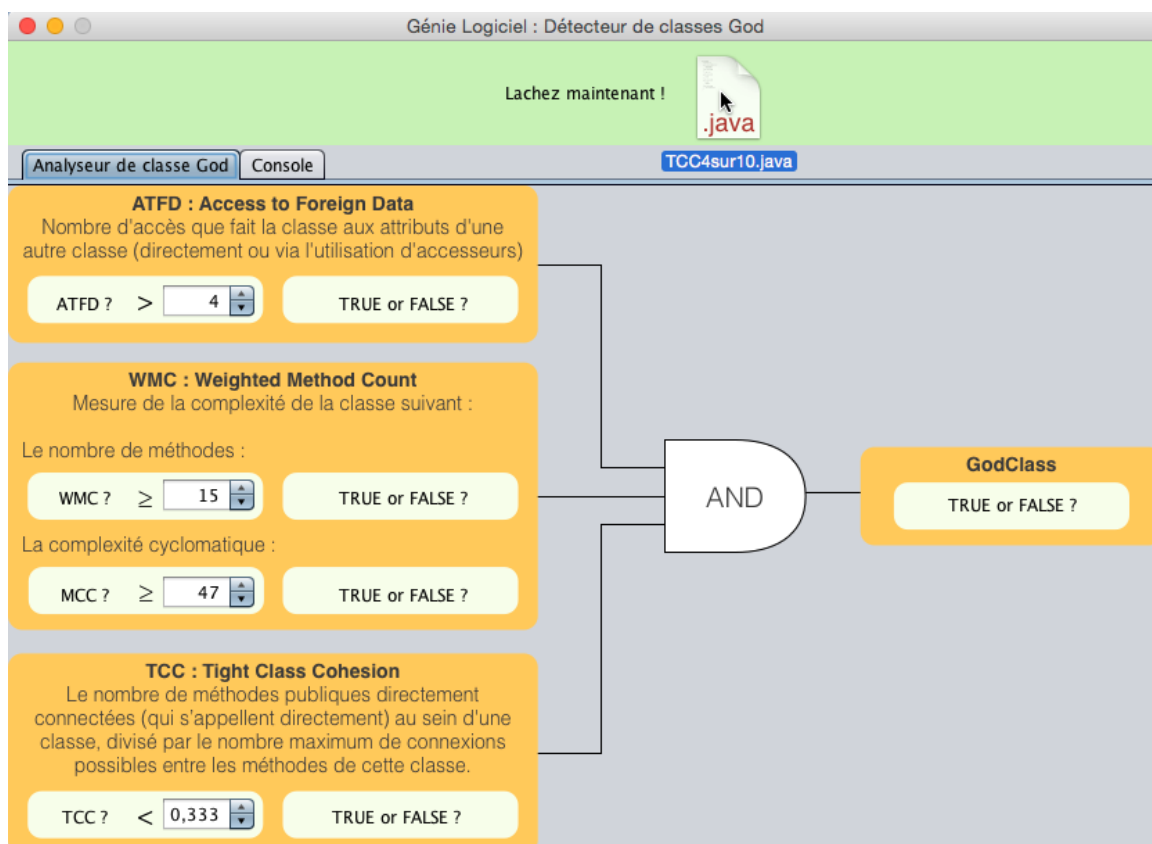


FIGURE 1.2 – Application développée pour le dossier final

Étude des différentes technologies à utiliser

Dans le cadre de mes anciens projets informatiques, je n'ai (presque) jamais eu recours à :

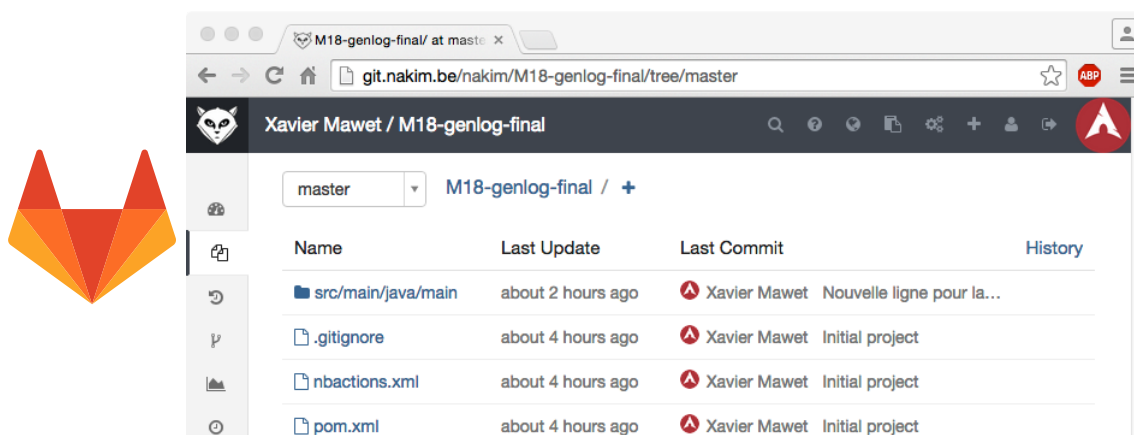
- des outils d'intégration continue
- des outils de couverture de code
- des outils de construction (build) de projets
- des outils de tests (unitaires, fonctionnels, ...)
- etc ...

C'est la raison pour laquelle, en ce qui me concerne, la réalisation de ce projet a débuté par une (longue) phase de recherche et d'étude sur ces différentes technologies. Je présenterais donc, au sein de ce chapitre, les différents outils que j'ai choisis pour mener à bien le travail demandé.

2.1 Outil de versionning : Git et Gitlab

Pour la quasi totalité de mes projets informatiques j'utilise le logiciel **git**. Il s'agit d'un logiciel de contrôle de version, comme Subversion, CVS, Arch ou encore Darcs, qui utilise des dépôts décentralisés. Il permet de développer/travailler tout en gardant une trace des modifications apportées et, en cas de problème, de retourner à un état antérieur des mes données. Il facilite également le travail à plusieurs sur un même projet.

De plus, voilà maintenant 3 ans que je fais de l'auto-hébergement pour mes codes sources sur un serveur au Canada. J'y ai installé GitLab, un serveur git moderne, complet et surtout **libre**. Il est accessible à l'adresse <http://git.nakim.be> et les dépôts publics via un lien sur mon site internet <http://www.nakim.be>.



GitLab et un service web d'hébergement et de gestion de versions de code source, facile à installer et à administrer, avec une très bonne communauté de développeurs et une documentation très riche. C'est à partir de là que l'outil d'intégration continue (Jenkins, présenté à la section 2.4) va récupérer l'ensemble du code publié pour l'exécuter sur sa plateforme grâce à un gestionnaire de builds (Maven, présenté à la section 2.2).

Si j'ai décidé de posséder mon propre serveur Git c'est principalement pour avoir le contrôle absolu sur mes sources. En effet, les solutions gratuites, telles que GitHub, n'offrent généralement pas la possibilité de créer des dépôts privés (service payant).

2.2 Outil de construction (build) de projet : Maven



Maven est un outil (open source) de construction (build) de projets. Il permet de faciliter et d'automatiser certaines tâches (répétitives) de la gestion d'un projet Java en exécutant un script qui contient une suite d'objectifs appelés « targets ». Chaque target joue un rôle précis. Elles permettent par exemple :

- d'automatiser certaines tâches : compilation, nettoyage, tests unitaires et déploiement des applications qui composent le projet
- de gérer des dépendances vis à vis des bibliothèques nécessaires au projet → utilise un ou plusieurs dépôts (repositories) qui peuvent être locaux (`.maven/repository`) ou distants (par défaut)
- de générer des documentations/sites web concernant le projet en se basant sur les commentaires dans le code

Maven est extensible grâce à un mécanisme de plugins qui permettent d'ajouter des fonctionnalités.

Contrairement à Ant, Maven impose par défaut une arborescence et un nommage des fichiers du projet car il suit le principe de « convention over configuration ».

2.2.1 Project Object Model : POM

Chaque projet Maven possède ce qu'on appelle un Project Object Model (POM) dans un fichier `pom.xml` qui permet à Maven de traiter le projet. Ce fichier contient la description du projet, configure les plugins, contient la déclaration des dépendances vers d'autres projets ou bibliothèques nécessaires à la compilation (comme `javaparser` dans notre cas), décrit les spécificités de construction (compilation et packaging), éventuellement le déploiement, dicte la génération de la documentation, l'exécution d'outils d'analyse statique du code, les tests, ...

Le listing 2.1 propose un aperçu sur le fichier `pom.xml` par défaut, obtenu après la création du projet

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
           ↪ maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <groupId>be.nakim</groupId>
7     <artifactId>M18-genlog-GodClass</artifactId>
8     <version>1.0-SNAPSHOT</version>
9     <packaging>jar</packaging>
10    <properties>
11        <project.build.sourceEncoding>UTF-8</project.build.
           ↪ sourceEncoding>
12        <maven.compiler.source>1.7</maven.compiler.source>
13        <maven.compiler.target>1.7</maven.compiler.target>
14    </properties>
15 </project>

```

Listing 2.1 – Fichier pom.xml par défaut

2.2.2 La gestion des dépendances

L'ajout d'une dépendance à un projet Maven est d'une simplicité déconcertante. Il suffit d'ajouter au POM la dépendance en renseignant :

- son groupId
- son artifactId
- sa version

```

1 <dependencies>
2
3     [...]
4
5     <!-- Ajout d une nouvelle dépendance -->
6     <dependency>
7         <groupId>GOUUP-ID</groupId>
8         <artifactId>ARTEFACT-ID</artifactId>
9         <version>VERSION</version>
10    </dependency>
11
12    [...]
13
14 </dependencies>

```

Listing 2.2 – Ajout des dépendances javaparser - junit - AbsoluteLayout

À la compilation du projet, Maven ira chercher la dépendance dans le dépôt local ou, s'il ne la trouve pas, il ira automatiquement la télécharger depuis un dépôt sur Internet et l'ajoutera à votre dépôt local.

De plus, Maven prend en charge la gestion des dépendances transitives, c'est-à-dire les dépendances requises par un artéfact¹, les dépendances de ces dépendances et ainsi de suite.

1. composant packagé possédant un identifiant unique composé d'un groupId, d'un artifactId et d'un numéro de version.

2.2.3 Exemple de fichier POM

À titre d'exemple, voici le fichier `pom.xml` complet de mon projet pour le dossier final :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www
   ↳ .w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
   ↳ apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>be.nakim</groupId>
5   <artifactId>M18-genlog-GodClass</artifactId>
6   <version>1.0-SNAPSHOT</version>
7   <packaging>jar</packaging>
8
9   <repositories>
10
11     <!--These are needed for Swing/Netbeans -->
12     <repository>
13       <id>maven2-repository.netbeans.maven2</id>
14       <name>Netbeans Maven Repository</name>
15       <url>http://bits.netbeans.org/maven2</url>
16       <layout>default</layout>
17     </repository>
18
19   </repositories>
20
21   <dependencies>
22
23     <!-- Java Swing AbsoluteLayout -->
24     <dependency>
25       <groupId>org.netbeans.external</groupId>
26       <artifactId>AbsoluteLayout</artifactId>
27       <version>RELEASE802</version>
28     </dependency>
29
30     <!-- Java Parser -->
31     <dependency>
32       <groupId>com.github.javaparser</groupId>
33       <artifactId>javaparser-core</artifactId>
34       <version>2.1.0</version>
35     </dependency>
36
37     <!-- JUnit 4 -->
38     <dependency>
39       <groupId>junit</groupId>
40       <artifactId>junit</artifactId>
41       <version>4.10</version>
42       <scope>test</scope>
43     </dependency>
44
45   </dependencies>
46
47   <properties>
48     <project.build.sourceEncoding>UTF-8</project.build.
   ↳ sourceEncoding>
49     <maven.compiler.source>1.7</maven.compiler.source>

```



```

50     <maven.compiler.target>1.7</maven.compiler.target>
51 </properties>
52
53 <build>
54     <plugins>
55
56         <!-- JaCoCo : Code Coverage -->
57         <plugin>
58             <groupId>org.jacoco</groupId>
59             <artifactId>jacoco-maven-plugin</artifactId>
60             <version>0.7.5.201505241946</version>
61             <executions>
62                 <execution>
63                     <goals>
64                         <goal>prepare-agent</goal>
65                     </goals>
66                 </execution>
67                 <execution>
68                     <id>report</id>
69                     <phase>prepare-package</phase>
70                     <goals>
71                         <goal>report</goal>
72                     </goals>
73                 </execution>
74             </executions>
75         </plugin>
76
77     </plugins>
78 </build>
79
80 </project>

```

Listing 2.3 – Fichier pom.xml de mon projet

2.3 Outils de test

La mise en œuvre des tests peut être facilitée par l'utilisation d'outils :

- d'automatisation des tests unitaires : xUnit (JUnit), TestNG, ...
- outils de couverture de code (code coverage) : emma, cobertura, ...
- outils pour automatiser les tests des GUI : Squish GUI Tester
- outils pour les tests fonctionnels : Fitnesse

2.3.1 Outil de tests unitaires : JUnit

Les tests unitaires sont destinés à tester une unité (portion d'un programme, par exemple une méthode) du programme et s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications.



Dans le cadre de ce projet, j'ai utilisé la version 4 du framework JUnit. Cette

version apporte un lot de nouveautés (par rapport aux versions 3.x) qui facilite la rédaction des cas de tests :

- la suppression des conventions de nommage²(pour les classes et les méthodes de test) et, par conséquent, de l'introspection
- l'utilisation des annotations introduites dans Java 5
- deux nouvelles surcharges de la méthode `assertEquals()` qui permettent de comparer deux tableaux d'objets
- l'attribut `expected=<ClassName>.class` de l'annotation `@Test` pour faciliter la vérification de la lever d'une exception.
- l'attribut `timeout=<time milliseconds>` de l'annotation `@Test` pour vérifier qu'un cas de tests s'exécute dans un temps maximum donné.
- etc ...

L'idée est de créer une classe de test par classe à tester. Chaque classe de test contiendra alors une méthode de test par cas de test (et non pas une méthode de test par méthode). En effet, Il est généralement préférable de n'avoir qu'un seul `assertXXX()` par test car un test ne devrait avoir qu'une seule raison d'échouer. Il est cependant possible d'utiliser plusieurs asserts dans un cas de tests si ceux-ci concernent un même cas fonctionnel.

2.3.2 Outil de (rapport de) couverture de code : JaCoCo

L'utilisation d'un outil d'analyse de la couverture de code permet de savoir les parties du code qui n'ont pas encore été testées (partie du code source non exécutée/-couverte par les tests) et pour lesquelles des tests additionnels devront être écrits. Il s'agit donc d'une mesure importante qui reflète la qualité des tests.



JaCoCo (Java Code Coverage) est une librairie open source d'analyse de couverture de code pour Java. Contrairement à **Cobertura** et **Emma**, il supporte pleinement Java 7 et Java 8. JaCoCo permet de mesurer :

- la couverture des instructions : statement coverage
- la couverture des conditions : branch (condition) coverage

Dans NetBeans 8.2, l'IDE que j'ai utilisé pour développer ce projet, **JaCoCo** est supporté nativement comme moteur de couverture de code (sous la forme d'un plugin). Pour activer ses fonctionnalités et l'intégrer au projet **Maven**, il suffit simplement de l'ajouter (sous forme de dépendance) dans la section « plugins » du fichier `pom.xml` du projet.

Le listing 2.4 reprend la configuration de JaCoCo que j'ai utilisée pour mon projet. Je me suis inspiré des exemples disponibles sur les sites suivants :

2. Il est tout de même conseillé de maintenir des conventions de nommage pour faciliter l'identification des classes de tests et des cas de tests. L'avantage de JUnit 4 est que les conventions ne sont plus imposées.

- http://wiki.netbeans.org/MavenCodeCoverage#Using_JaCoCo
- <http://www.eclemma.org/jacoco/trunk/doc/maven.html>
- https://blogs.oracle.com/geertjan/entry/code_coverage_for_maven_in

```

1 <plugins>
2
3     [...]
4
5     <!-- JaCoCo : Code Coverage -->
6     <plugin>
7         <groupId>org.jacoco</groupId>
8         <artifactId>jacoco-maven-plugin</artifactId>
9         <version>0.7.5.201505241946</version>
10        <executions>
11            <execution>
12                <goals>
13                    <goal>prepare-agent</goal>
14                </goals>
15            </execution>
16            <execution>
17                <id>report</id>
18                <phase>prepare-package</phase>
19                <goals>
20                    <goal>report</goal>
21                </goals>
22            </execution>
23        </executions>
24    </plugin>
25
26    [...]
27
28 </plugins>

```

Listing 2.4 – Ajout du plugin **JaCoCo** dans le fichier `pom.xml`

2.3.3 Outils pour les tests fonctionnels

Très souvent lorsqu'on parle de tests fonctionnels automatisés on pense immédiatement aux outils de tests d'interface graphique (GUI). Cette approche est en effet la plus naturelle car elle consiste en fait à simuler l'utilisateur final par un outil qui reproduit son comportement.

Défauts :

- Les tests sont décrits dans un formalisme technique peu compréhensible par des utilisateurs, leur rédaction requiert donc l'intervention systématique d'informaticiens.
- Pour palier à ce manque, certains outils proposent un « recorder » permettant de créer un scénario de test en enregistrant les manipulations d'un utilisateur (souris, clics, entrées clavier, etc). Mais on perd alors l'un des principes clé des démarches de développement piloté par les tests : la capacité à écrire ses tests en amont des développement.

L'avantage évident du test fonctionnel d'interface graphique est qu'il permet de reproduire en intégralité les cas d'utilisation d'une application (vu de l'utilisateur).

2.4 Outil d'intégration continue : Jenkins

On peut définir l'intégration continue comme étant « l'art de tester des modules développés par une équipe sur une plateforme automatisée afin d'assurer la cohésion du code créé. Elle permet d'accélérer la productivité et de mettre en avant plus rapidement les bugs pouvant être présents. Elle permet en outre de s'assurer de la non regression des logiciels et libère aussi quelques ressources humaines grâce à l'automatisation des tests »³.



Comme serveur d'intégration continue, j'ai décidé d'utiliser **Jenkins**. Il s'agit d'un serveur d'intégration continue gratuit très en vogue, notamment pour les projets Java développés avec Maven (ça tombe bien). Bien que je n'ai pas beaucoup d'expérience avec d'autres serveurs d'intégration, j'ai été agréablement surpris par sa simplicité d'utilisation (prise en main facile) grâce à son interface très intuitive. Il en va de même pour l'ajout des plugins (présents en grande quantité) et la création des jobs.

S	M	Nom du projet ↓	Dernier succès	Dernier échec	Dernière durée
		testmaven	2 mn 7 s - #7	s. o.	15 s

W	Description	%
	Coverage: All coverage targets have been met.	100
	Stabilité du build: Aucun build récent n'a échoué.	100

Pour réaliser le travail demandé, j'ai utilisé 2 serveurs Jenkins : le premier tournant sur **Mac OS X El Capitan** et le second sur **Windows 10**. Ils ont été configurés pour interagir avec mon serveur GitLab. Cette interaction est rendu possible par :

1. l'installation de plugins sur les serveurs Jenkins :

Activé	Nom ↓	Version	Version précédente	Épinglé	Désinstaller
<input checked="" type="checkbox"/>	GitLab Hook Plugin Enables GitLab web hooks to be used to trigger SMC polling on Gitlab projects	1.4.0			Désinstaller
<input checked="" type="checkbox"/>	GitLab Logo Plugin Display GitLab Repository loon on dashboard	1.0.0			Désinstaller
<input checked="" type="checkbox"/>	GitLab Merge Request Builder Integrates Jenkins with Gitlab to build Merge Requests	1.2.2			Désinstaller
<input checked="" type="checkbox"/>	GitLab Plugin This plugin integrates GitLab to Jenkins by faking a GitLab CI Server.	1.1.25			Désinstaller

3. <http://www-igm.univ-mlv.fr/~dr/XPOSE2012/Integration%20Continue/concept.html>

2. la configuration du job de la manière suivante :

Gestion de code source

☐ Aucune
☐ CVS
☐ CVS Projectset
☒ Git

Repositories

Repository URL

Credentials

Branches to build

Branch Specifier (blank for 'any')

Navigateur de la base de code

URL

Version

3. la configuration d'un service dans les paramètres du dépôt sur mon serveur GitLab :

git.nakim.be/nakim/M18-genlog-final/services/gitlab_ci/edit?notice=Successfully...

Xavier Mawet / M18-genlog-final

We sent a request to the provided URL

GitLab CI ●

Continuous integration server from GitLab

[← to services](#)

Active ☒

Trigger ☒ **Push events**
 This url will be triggered by a push to the repository

☒ **Tag push events**
 This url will be triggered when a new tag is pushed to the repository

Token

Project url

Cette interaction permet aux serveurs Jenkins de récupérer les sources de mon projet Maven sur mon serveur GitLab. À chaque événement « push » dans le dépôt `M18-genlog-final`, Jenkins va rechercher les dernières modifications et les derniers ajouts commités avant de « build » le projet. Maven est alors lancé pour construire le projet et lancer les tests.

Ce qui déclenche le build

☒ Lance un build à chaque fois qu'une dépendance SNAPSHOT est construite

☐ Construire après le build sur d'autres projets

☒ Construire périodiquement

Planning

HH**

Aurait été lancé à lundi 3 août 2015 22 h 08 min 36 s CEST;
prochaine exécution à mardi 4 août 2015 22 h 08 min 36 s CEST.

☒ Build when a change is pushed to GitLab. GitLab CI Service URL: <http://localhost:8080/project/M18-genlog-final>

Build on Merge Request Events

☒

Build on Push Events

☒

Rebuild open Merge Requests on Push Events

☒

Enable [ci-skip]

☒

Set build description to build cause (eg. Merge request or Git Push)

☒

Add note with build status on merge requests

☒

Vote added to note with build status on merge requests

☒

En cas de problème de détection des événements push sur mon serveur git, j'ai également configuré un build quotidien du projet `M18-genlog-final`. Ce build n'aura lieu que si des changements ont été opérés.

2.4.1 Rapport de couverture de code sur Jenkins avec JaCoCo

En ce qui concerne la couverture de code, j'ai installé le plugin **JaCoCo** et configuré le job de manière à obtenir un rapport de couverture de code après chaque build :

Actions à la suite du build

Record JaCoCo coverage report

Path to exec files (e.g.:

`**/target/**/*.exec, **/jacoco.exec)`

`**/*.exec`

Path to class directories (e.g.:

`**/target/classDir, **/classes)`

`**/classes`

Path to source directories (e.g.:

`**/mySourceFiles)`

`**/src/main/java`

Inclusions (e.g. `**/*.class`)

Exclusions (e.g. `**/*Test*.class`)



Instruction

% Branch

% Complexity

% Line

% Method

% Class

0

0

0

0

0



0

0

0

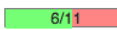
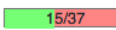
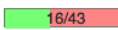
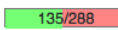
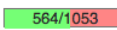

0

0

☐ Change build status according the thresholds

FIGURE 2.3 – Ajout d'une action à la suite du build : Rapport de couverture de code

Project Coverage summary

Name	Packages	Files	Classes	Methods	Lines	Conditionals
Cobertura Coverage Report	55% 	41% 	37% 	47% 	54% 	72% 

Coverage Breakdown by Package

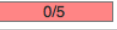
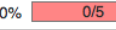
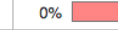

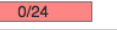
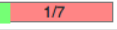
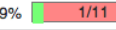
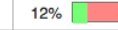

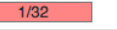
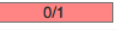

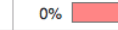

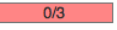



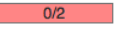

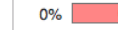


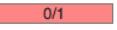

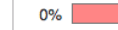




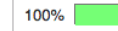

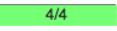



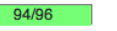



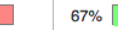
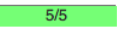


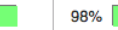
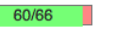
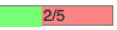
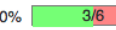

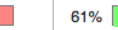
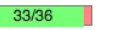
Name	Files	Classes	Methods	Lines	Conditionals
dragdrop	0% 	0% 	0% 	0% 	0% 
godclass	14% 	9% 	12% 	16% 	3% 
gui	0% 	0% 	0% 	0% 	N/A
gui.mediator	0% 	0% 	0% 	0% 	N/A
gui.spinner	0% 	0% 	0% 	0% 	0% 
main	0% 	0% 	0% 	0% 	0% 
metrics	100% 	100% 	100% 	100% 	N/A
metrics.calculators	100% 	100% 	100% 	100% 	98% 
metrics.exceptions	67% 	67% 	67% 	67% 	N/A
metrics.visitors	100% 	100% 	100% 	98% 	91% 
utils	40% 	50% 	53% 	61% 	92% 

FIGURE 2.4 – Exemple d'un rapport de couverture de code

TDD : Le développement piloté par les tests

Le test-driven development (TDD) est une méthode de développement logiciel qui préconise de rédiger les tests unitaires avant d'écrire le code source d'un logiciel. C'est l'écriture des tests automatisés qui dirige l'écriture du code source.

Concrètement, la marche à suivre est composée de 3 phases :

1. **R (Red)** : écrire un code de test, l'exécuter et vérifier qu'il échoue bien (d'où la couleur rouge) car le code qu'il teste n'a pas encore été rédigé.
2. **G (Green)** : écrire le code métier (l'implémentation) en rapport avec le test, exécuter le test et vérifier qu'il est valide (d'où la couleur verte). Au fur et à mesure du développement, de plus en plus de tests vont réussir et lorsqu'ils réussissent tous, le logiciel est être considéré comme terminé.
3. **R (Refactor)** : remaniement du code afin d'en améliorer la qualité mais en conservant les mêmes fonctionnalités. Relancer également les tests unitaires afin de s'assurer que le refactor n'a pas modifié les fonctionnalités implémentées.

La figure 3.5, extraire du rapport de couverture de code générée sur Jenkins, témoigne de l'utilisation de la méthode TDD lors du développement du programme :

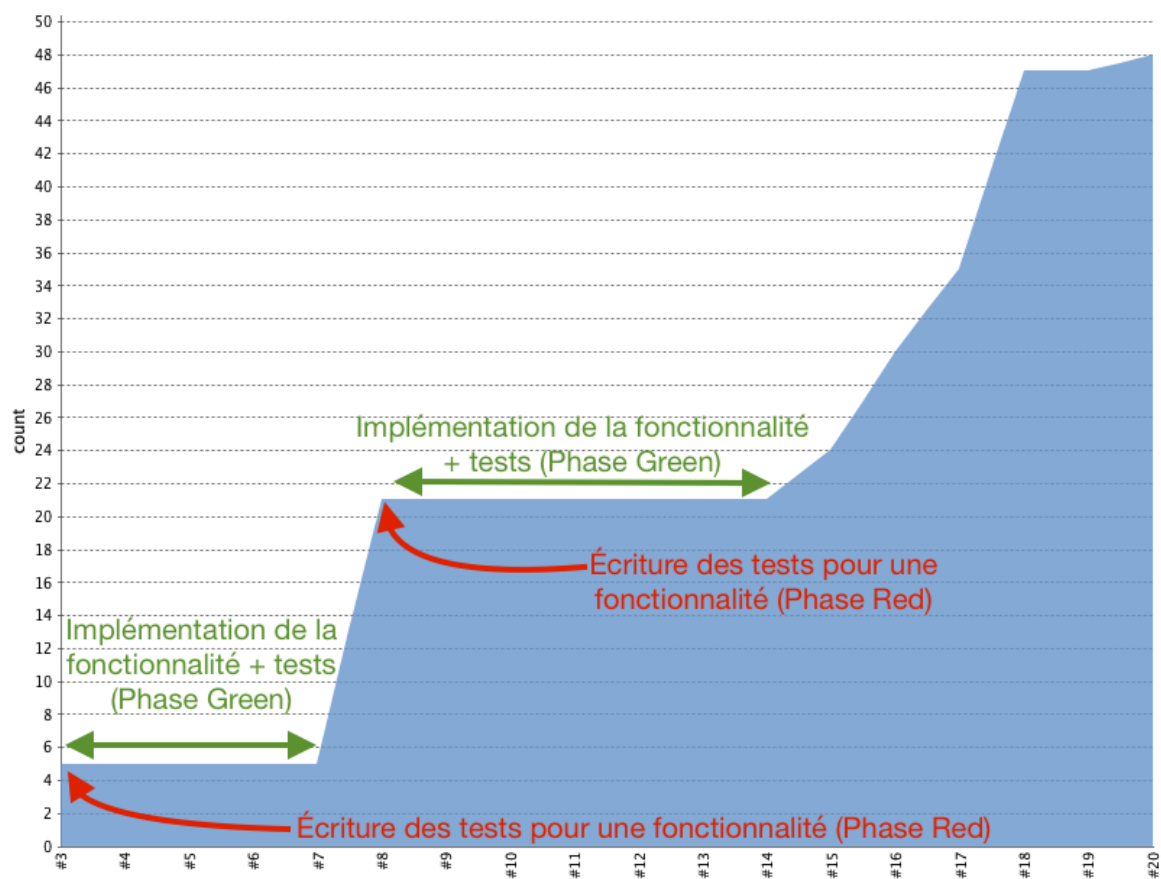


FIGURE 3.5 – Nombre de tests développés au fur et à mesure des différents builds

Patrons de conception utilisés

Au niveau de la conception, j'ai utilisé 4 design patterns différents que je détaillerai dans ce chapitre :

1. Observer
2. Adapter
3. Visitor
4. Mediator

4.1 Observer

L'objectif du pattern **Observer** est de définir une dépendance du type un-à-plusieurs (1,n) entre des objets de manière que, lorsqu'un objet change d'état, tous les objets dépendants en soient notifiés afin de pouvoir réagir conformément.

4.1.1 Le DragDropPanel

Afin d'avoir une application plus « user friendly », j'ai également développé un panel qui gère les fonctionnalités du « drag and drop » des fichiers (voir figure 1.2). Ce panel simplifie grandement le processus d'importation des fichiers à analyser. De plus, il présente l'avantage de pouvoir spécifier en un seul « glisser-déposer » un nombre quelconque de fichiers à analyser de manière séquentielle.

Le diagramme UML 4.6 donne un aperçu des différentes classes qui ont été nécessaires au développement de cette fonctionnalité. Le principe consiste à connecter un composant AWT (ou Swing) à un `DropTargetListener`, en utilisant un `DropTarget`.

La classe `DragDropPanel` implémente directement l'interface `DropTargetListener`. Ce panel va donc gérer en interne les événements liés au « drag and drop » (le changement de couleur lors du survole avec des fichiers, la récupération des éléments déposés, etc ...) et « émettre » les événements `DragDropEvent` et `MultipleDragDropEvent`. L'événement représenté par la classe `DragDropEvent` permet de renseigner le path d'un unique fichier déposé, tandis que l'événement représenté par la classe `MultipleDragDropEvent` permet de renseigner les paths d'une multitude de fichiers déposés en un seul « drag and drop ». L'implémentation de la classe `DropPanel` est donnée dans le listing 4.5.

Cette fonctionnalité du « drag and drop » utilise le pattern **Observer** dans le sens où n'importe quel objet implémentant l'interface `DragDropListener` peut s'enregistrer comme listener au près d'un `DropPanel`.

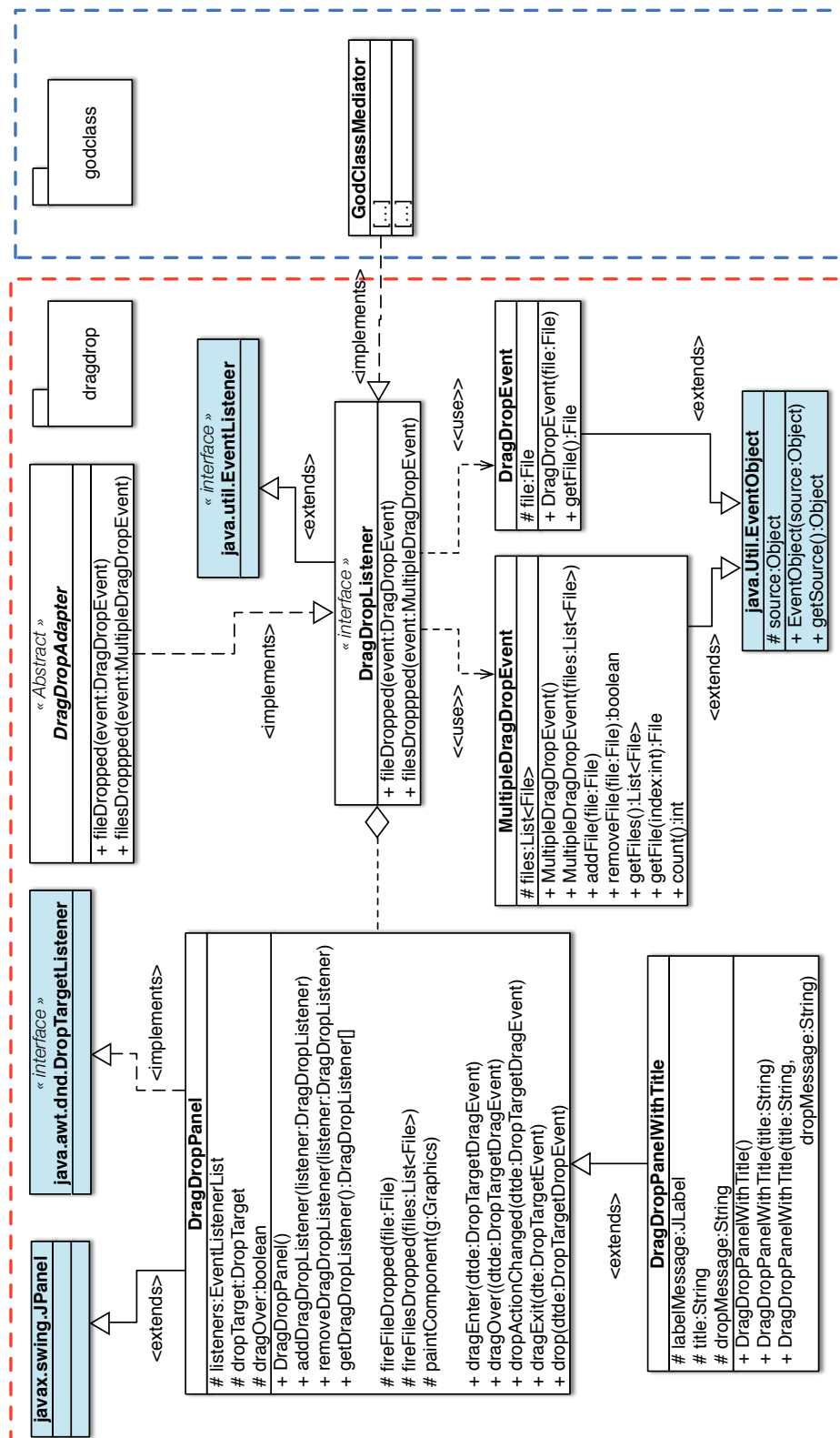


FIGURE 4.6 – Diagramme de classe pour le package `dragdrop`

A titre d'exemple, voici le code de la classe `DragDropPanel` :

```

1 public class DragDropPanel extends javax.swing.JPanel
2                               implements DropTargetListener
3 {
4     protected final EventListenerList listeners = new EventListenerList
5         ↪ ();
6
7     protected DropTarget dropTarget;
8     protected boolean dragOver = false;
9
10    public DragDropPanel()
11    {
12        this.dropTarget = new DropTarget(this, this);
13        this.dragOver = false;
14    }
15
16    public void addDragDropListener(DragDropListener listener)
17    {
18        this.listeners.add(DragDropListener.class, listener);
19    }
20
21    public void removeDragDropListener(DragDropListener listener)
22    {
23        this.listeners.remove(DragDropListener.class, listener);
24    }
25
26    public DragDropListener[] getDragDropListeners()
27    {
28        return this.listeners.getListeners(DragDropListener.class);
29    }
30
31    protected void fireFileDropped(File file)
32    {
33        DragDropEvent event = null;
34        for (DragDropListener listener : this.getDragDropListeners())
35        {
36            if (event == null)
37                event = new DragDropEvent(this, file);
38
39            listener.fileDropped(event);
40        }
41    }
42
43    protected void fireFilesDropped(List<File> files)
44    {
45        if (files.isEmpty())
46            return;
47
48        if (files.size() == 1)
49        {
50            this.fireFileDropped(files.get(0));
51        }
52        else
53        {
54            MultipleDragDropEvent event = null;
55            for (DragDropListener listener : this.getDragDropListeners())
56            {

```

```

56         if (event == null)
57             event = new MultipleDragDropEvent(this, files);
58
59         listener.filesDropped(event);
60     }
61 }
62
63
64 @Override
65 protected void paintComponent(Graphics g)
66 {
67     super.paintComponent(g);
68
69     if (this.dragOver)
70         this.setBackground(new Color(209, 245, 189));
71     else
72         this.setBackground(UIManager.getColor("Panel.background"));
73 }
74
75 @Override
76 public void dragEnter(DropTargetDragEvent dtde)
77 {
78     this.dragOver = true;
79     repaint();
80 }
81
82 @Override
83 public void dragOver(DropTargetDragEvent dtde)
84 {
85 }
86
87 @Override
88 public void dropActionChanged(DropTargetDragEvent dtde)
89 {
90 }
91
92 @Override
93 public void dragExit(DropTargetEvent dte)
94 {
95     this.dragOver = false;
96     repaint();
97 }
98
99 @Override
100 public void drop(DropTargetDropEvent dtde)
101 {
102     // Accept copy drops
103     dtde.acceptDrop(DnDConstants.ACTION_COPY);
104
105     // Update background color
106     this.dragOver = false;
107     repaint();
108
109     // Get the transfer which can provide the dropped item data
110     Transferable transferable = dtde.getTransferable();
111
112     // Get the data formats of the dropped item
113     DataFlavor[] flavors = transferable.getTransferDataFlavors();

```

```

114
115 // Loop through the flavors
116 for (DataFlavor flavor : flavors)
117 {
118     try
119     {
120         // If the drop items are files
121         if (flavor.isFlavorJavaFileListType())
122         {
123             // Get all of the dropped files
124             List transfertData = (List) transferable.
                ↪ getTransferData(flavor);
125
126             List<File> files = new ArrayList<>();
127             Iterator iter = transfertData.iterator();
128             while (iter.hasNext())
129                 files.add((File)iter.next());
130
131             this.fireFilesDropped(files);
132         }
133     }
134     catch (UnsupportedFlavorException | IOException e)
135     {
136         e.printStackTrace();
137     }
138 }
139
140 // Inform that the drop is complete
141 dtde.dropComplete(true);
142 }
143
144 private static final long serialVersionUID = 1L;
145 }
146

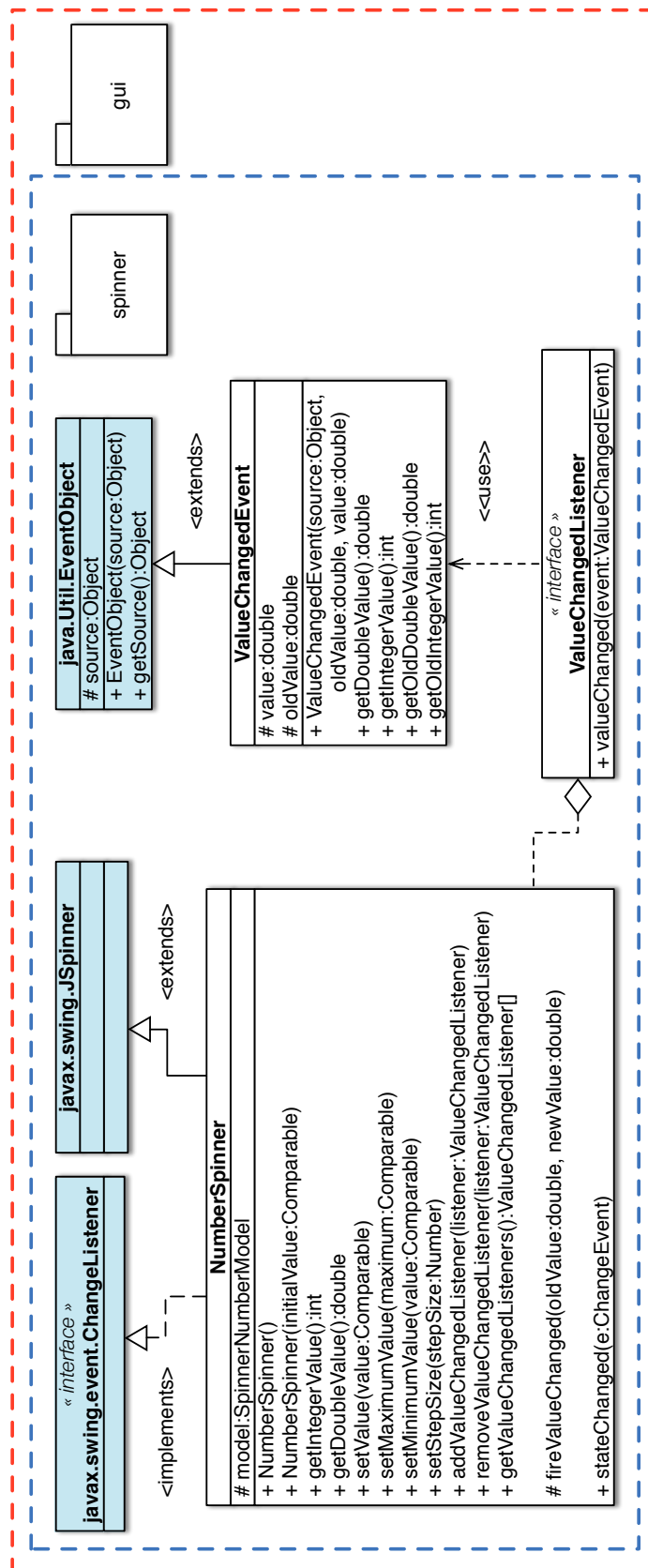
```

Listing 4.5 – Classe DropPanel

4.1.2 Le NumberSpinner

J'ai également utilisé le pattern Observer afin de développer un Spinner exclusivement numérique et plus simple d'utilisation que le spinner fourni par Swing. Ce Spinner émet un événement `ValueChangedEvent` à chaque modification du nombre affiché. Cet événement nous donne accès à la nouvelle valeur entrée dans le spinner mais également la précédente valeur qui vient d'être remplacée.

Le diagramme UML 4.7 donne un aperçu des différentes classes qui ont été nécessaires au développement de cette fonctionnalité.


 FIGURE 4.7 – Diagramme de classe pour le package `spinner`

4.2 Adapter

Comme mentionné dans le JDK, le pattern **Adapter** est très largement utilisé dans le domaine de l'« Event Handling » (c'est-à-dire de manière conjointe avec le pattern Observer). Il est principalement utilisé lorsque l'interface **Listener** possède beaucoup de méthodes.

Malgré le fait que l'interface **DragDropListener** ne possède que 2 méthodes, j'ai développé une classe abstraite « adapter », appelée **DragDropAdapter**, qui implémente cette interface et qui définit toutes ses méthodes avec un corps vide. Cela permettra au client de ne devoir surcharger que les méthodes nécessaires :

```

1 public abstract class DragDropAdapter implements DragDropListener
2 {
3     @Override
4     public void fileDropped(DragDropEvent event)
5     {
6         // Nothing to do here ...
7     }
8
9     @Override
10    public void filesDropped(MultipleDragDropEvent event)
11    {
12        // Nothing to do here ...
13    }
14 }
```

Listing 4.6 – Classe DropPanel

4.3 Visitor

Le pattern Visitor permet de définir une nouvelle opération (le calcul de métriques dans notre cas) qui agit sur une hiérarchie (les éléments d'une structure d'objets) sans changer les classes sur lesquels s'appliquent ces opérations. Le pattern VISITOR permet d'ouvrir la voie à une variété illimitée d'extensions pouvant être apportées par un développeur n'ayant pas accès au code source.

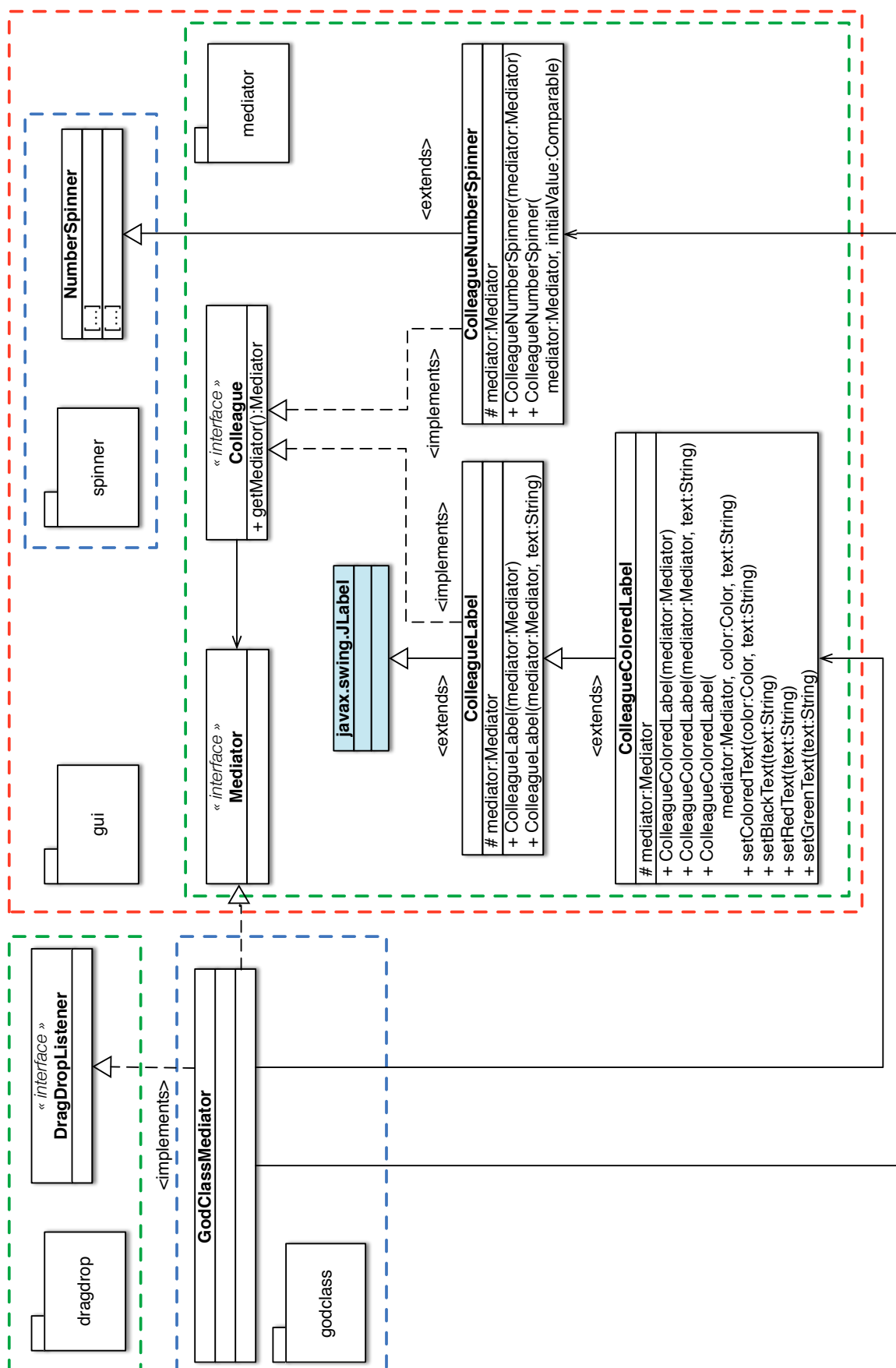
C'est notamment le cas avec la librairie **javaparser** qui est construite sur ce modèle de fonctionnement. La figure 4.8 montre la hiérarchie des classes qui ont été développées afin de permettre le calcul de 4 métriques (ATFD, WMC, MCC, TCC).

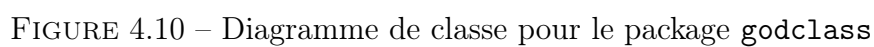


4.4 Médiateur

Le but ici est de définir un objet (`GodClassMediator`) dont le rôle est d'encapsuler les interactions (logique métier) qu'il existe entre un ensemble de widgets graphiques (instance des classes `ColleagueNumberSpinner` et `ColleagueColoredLabel`). Cela promeut un couplage lâche, évitant aux objets d'avoir à se référer explicitement les uns aux autres, et permet de varier leur interaction indépendamment. Casser un couplage fort entre un ensemble d'objets (d'un GUI comme c'est le cas ici). Lorsque les interactions entre les objets reposent sur une condition complexe impliquant que chaque objet d'un groupe connaisse tous les autres, il est utile d'établir une autorité centrale.

Chaque élément interagissant (widgets du GUI) est un « collègue », instances de l'interface `Colleague`. La classe dédiée à la communication est le médiateur, instance de l'interface `Mediator` (`GodClassMediator` en l'occurrence). Cet objet `Mediator` contient une référence vers tous les widgets du GUI et il est également listener de tous ces widgets. Il va donc contrôler et coordonner les interactions des composants graphiques.


 FIGURE 4.9 – Diagramme de classe pour le package `gui.mediator`



Autres diagrammes

5.1 Diagramme des différents packages

