

Audit Spring-Boot — Projet de GL

Abdous-salam CISSE — Lucas SAUVAGE

07 Mars 2025



Contents

1	Présentation globale du projet	4
1.1	Fonctionnalités du projet	4
1.2	Comment lancer le projet ?	4
1.2.1	Pré-requis	4
1.2.2	Compilation et construction du projet	5
1.3	Description du projet	5
1.3.1	Analyse du readme	5
2	Historique du logiciel	6
2.1	Analyse du git	6
3	Architecture logicielle	8
3.1	Utilisation de bibliothèques extérieures	8
3.2	Organisation en paquetage	8
3.3	Répartition des classes dans les paquetages	9
3.4	Organisation des classes	11
4	Pour aller plus loin	13
4.1	Tests	13
4.2	Commentaires	15
4.2.1	class	16
4.2.2	interface	18
4.2.3	package	19
4.2.4	project	20
4.3	Les méthodes	20
4.3.1	Statistiques	20
4.3.2	God Classes	21
4.4	Code inutile	25
5	Nettoyage de Code et Code smells	27
5.1	Règles de nommage	27
5.2	Nombre magique	27
5.3	Structure du code	28
6	Annexe	30

Introduction

Ce document correspond à l'audit de l'outil de développement **Spring-Boot** dans le cadre du projet de Génie Logiciel en L3.

Spring-Boot est un composant qui simplifie l'utilisation de Spring Framework en automatisant la configuration et en fournissant des starters pour intégrer facilement d'autres modules.

Spring-Boot plus particulièrement permet de lier tous les composants du framework :

Spring-Core : Fournit l'IoC ¹ et l'injection de dépendances qui permettent de gérer les objets et leurs dépendances. Il fournit également Spring MVC ².

Spring-JPA : Facilite la gestion de bases de données.

Spring-Security : Assure la sécurité des applications en gérant l'authentification et l'autorisation.

Spring-Data : Simplifie l'accès aux bases de données (relationnelles ³ ou non).

Spring-Cloud : Gère les architectures micro-services.

Comme ce projet est de taille conséquente, nous avons choisi de nous concentrer sur le package boot⁴, car après lecture et analyse du code, c'est ce package qui réunit les fonctionnalités de base du projet.

Trivialement, le package boot implémente la logique du framework. Par exemple, la classe **ApplicationHome** est utilisée pour déterminer et fournir l'accès au répertoire principal (home directory) de l'application, qu'elle soit exécutée à partir d'un fichier JAR, d'un répertoire ou directement depuis le code source.

¹Inversion de Contrôle : Délègue la gestion des dépendances à un conteneur externe et facilite la maintenance, les tests et l'extensibilité du code.

²Utilisé pour créer des applications web.

³Utilisant des tableaux à 2 dimensions : des *tables* ou *relations*.

⁴spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot

1 Présentation globale du projet

1.1 Fonctionnalités du projet

Le logiciel permet de concevoir une application java complète(application classique, API,..) de l'étape de la conception au déploiement en générant un fichier jar exécutable, ce qui facilite les tests. On peut prendre comme exemple **Spring Petclinic** qui est une application qui permet aux vétérinaires d'organiser leurs rendez-vous, la gestion des fiches client ...

Ce projet est divisé en 8 modules :

- **spring-boot** : Spring Boot est la bibliothèque principale qui fournit des fonctionnalités pour créer des applications Spring avec une configuration par défaut.
- **spring-boot-autoconfigure** : Sert à configurer automatiquement de nombreuses parties de l'application en fonction des dépendances présentes dans le classpath.
- **spring-boot-starters** : Ensembles de dépendances permettant d'ajouter facilement des fonctionnalités Spring au projet sans avoir à chercher chaque dépendance individuellement
- **spring-boot-actuator** : Permet de surveiller et d'interagir avec l'application via des points de terminaison (endpoints), comme la vérification de l'état de santé ou la configuration.
- **spring-boot-actuator-autoconfigure** : Configure automatiquement les points de terminaison d'Actuator en fonction du classpath et des propriétés définies, facilitant ainsi leur utilisation.
- **spring-boot-test** : Contient des outils et annotations utiles pour écrire des tests pour l'application.
- **spring-boot-test-autoconfigure** : Configure automatiquement les tests en fonction des dépendances présentes dans le classpath.
- **spring-boot-loader** : Permet de créer un fichier JAR exécutable, simplifiant ainsi le déploiement et l'exécution de l'application
- **spring-boot-devtools** : Fournit des fonctionnalités pour améliorer l'expérience de développement, comme le redémarrage automatique de l'application lors des modifications du code.

1.2 Comment lancer le projet ?

1.2.1 Pré-requis

La liste des pré-requis pour utiliser ce projet est la suivants :

- JDK 17
- Maven ou Gradle
- Un IDE tel que IntelliJ

1.2.2 Compilation et construction du projet

Spring-Boot utilise soit Gradle soit Maven mais les développeurs de ce projet nous indiquent qu'il faut utiliser Gradle.

```
# Pour publier le projet dans le cache Maven local :  
$ ./gradlew publishToMavenLocal  
  
# Pour compiler et tester :  
$ ./gradlew build
```

1.3 Description du projet

1.3.1 Analyse du readme

Un readme est présent avec le projet. On peut s'apercevoir directement que la dernière modification de ce projet a eu lieu il y a un mois, ce qui prouve que le readme est tenu à jour.

En analysant plus précieusement les commits concernant ce fichier, on peut voir un nombre important de mises à jour au cours des années 2021 et 2024, ainsi que deux mises à jour pour 2022 et 2023.

Le readme donne également les commandes de bases pour compiler, exécuter le projet et les tests, il donne en plus toutes les informations sur les différents modules du projet. Au vu de tout ce qu'on a testé sur le projet, les informations de lancement et d'utilisation sont suffisantes et le projet contient une javadoc qui est plutôt bien détaillée.

2 Historique du logiciel

2.1 Analyse du git

En terme de composition d'équipe ce projet a fait appel à 1147 contributeurs au total, de façon équilibrée sur le projet et dans le temps il y a un nombre de contributeurs peu élevé (une vingtaine) par rapport au nombre de contributeurs total. Concernant l'activité, on remarque une bonne activité régulière sur le temps(le projet est toujours actif, plusieurs commits en février 2025).

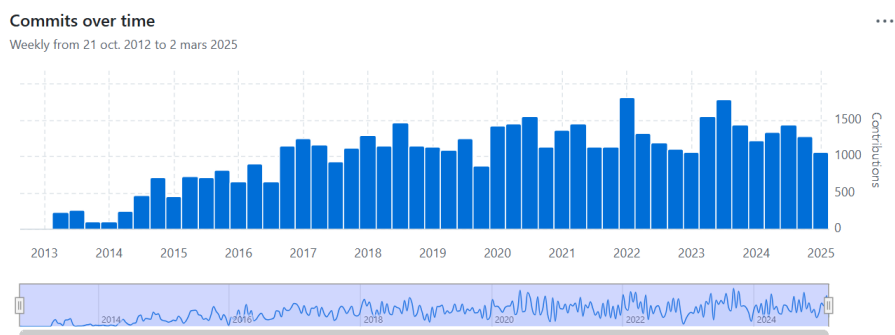


Figure 1: Nombre de commits en fonction de l'année

Au niveau de la création et l'utilisation des branches, nous sommes sur 22 branches créées avec seulement 11 (y compris la branche principale) utilisées au cours du développement de ce projet. Aujourd'hui seulement 4 (ou 3 si l'on exclut la branche principale) sont actives.

Branch	Updated	Check status	Behind / Ahead	Pull request
main	5 minutes ago	0 / 11	Default	
3.4.x	5 minutes ago	0 / 11	665 0	
4.0.x	16 minutes ago	0 / 12	2 12	
3.3.x	3 hours ago	12 / 12	2517 0	

Figure 2: Nombre de pull-request acceptées

Pour le nombre de pull-request, il y en a 22 dont 3 qui ont été fusionnées avec la branche principale(voir image ci-dessous), toutes les autres sont en attente.

On peut voir qu'il y a un total de 213 pull-request (209 fermées et 4 ouvertes).

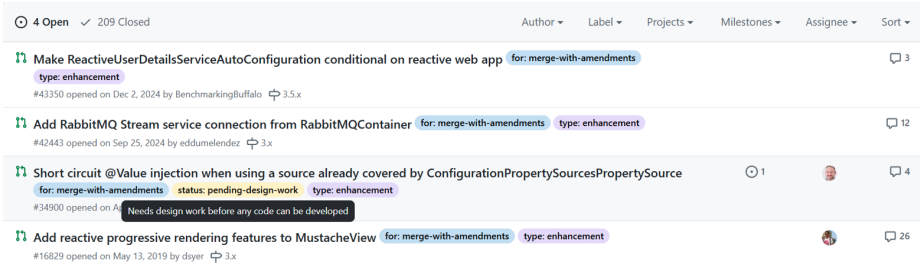


Figure 3: Nombre de pull-request acceptées

3 Architecture logicielle

Comme nous pouvons le voir dans ce tableau récapitulatif des métriques du projet en entier, ce projet est conséquent, c'est pourquoi nous avons choisi de nous focaliser sur un package du projet.

	Lignes Java	Classes	Fichiers	Packages
Projet	672,050	13,673	10,042	1,123

Table 1: Statistiques du projet

Nous avons donc choisi de nous intéresser à **org.springframework.boot** car ce package est quand même conséquent et implémente la "base" du projet. Nous avons donc logiquement choisi de travailler sur ce package.

3.1 Utilisation de bibliothèques extérieures

Ce projet contient un grand nombre de bibliothèques extérieures (97) dont **log4j-core**, **jackson-databind**, **hibernate-core**, **tomcat-embed-core**... et on a remarqué un point important avec cette commande

```
$ grep -rnw './' -e 'import org.NOM_DE_LA_BIBLIOTHEQUE'
```

C'est qu'il y a quelques bibliothèques extérieures importées tels que **spring-core-test**, **reactor-test** et **spring-context-support** dans ce fichier¹ ne sont pas utilisées donc dans un sens elles sont inutiles et cela pourrait avoir un impact sur la lisibilité du code. Sur les bibliothèques utilisées, il y a évidemment plusieurs qui font les mêmes choses mais cela permet d'avoir une meilleure compatibilité, flexibilité et conformité entre toutes les parties de l'application et aussi de gérer plusieurs phases d'une même étape (par exemple junit et mockito pour les tests) donc ce n'est pas forcément une mauvaise pratique.

3.2 Organisation en paquetage

Dans la section que nous avons décidé d'analyser, il est présent 84 paquetages. Une analyse approfondie conclut par l'image ci-dessous nous a permis de constater des dépendances entre les paquetages, par exemple **cloud**² et **config**³ qui sont mutuellement dépendants. On remarque une organisation en couche pour les paquetages pas très stricte dû au fait qu'il existe des cycles entre paquetages notamment **config** qui est impliqué dans 10 cycles et **cloud** dans 6 cycles, ce qui pourrait fortement impacter sur la maintenabilité du projet.

¹spring-boot/spring-boot-project/spring-boot/build.gradle

²spring-boot/spring-boot-project/spring-boot/src/main/java/org.springframework.boot.cloud

³spring-boot/spring-boot-project/spring-boot/src/main/java/org.springframework.boot.context.config

Le projet comptabilise 6 niveaux de paquetage, avec le niveau 6 qui contient le paquetage **buffering**¹. La hiérarchie des paquetages pour les tests suit exactement la même organisation que celle des sources et il y a des hiérarchies parallèles à certains niveaux, par exemple le niveau 3 qui contient les paquetages **context**, **cloud** et **web** où chaque branche semble gérer des aspects différents, mais elles suivent une structure parallèle avec une profondeur similaire. Il existe aussi des paquetages qui ne contiennent que d'autres paquetages sans classes directes, par exemple le paquetage **servlet**². Ces différents aspects nous indiquent une hiérarchie relativement profonde mais bien organisée.

Dans la construction du projet, les noms des paquetages ont été donnés à titre indicatif donc peuvent donner une orientation sur le ou les design patterns utilisés, l'existence d'un lien quelconque avec une base de donnée. Les noms des paquetages comme **web**, **http**, et **webservices** tous situés dans le paquetage **boot** suggèrent la présence d'une couche de présentation qui pourrait être liée à la partie View du pattern MVC(Model-View-Controller). Les paquetages **Orm**, **Jdbc**, **r2dbc** qui sont aussi inclus dans le paquetage **boot** suggèrent l'interaction avec une base de donnée. De façon plus générale, les noms des paquetages indiquent que le projet est basé sur une architecture bien définie et est équipé pour gérer tous les aspects majeurs(sécurité, environnement, persistance des données...) du début à la fin.

package	^	Cyclic	PDcy	PDpt	PDpt*
org.springframework.boot.web.servlet.filter		0	0	0	1
org.springframework.boot.web.servlet.server		0	4	4	6
org.springframework.boot.web.servlet.support		0	7	0	1
org.springframework.boot.web.servlet.view		0	0	0	1
org.springframework.boot.webservices.client		0	3	0	1
org.springframework.boot.cloud		1	6	1	2
org.springframework.boot.context.config		1	10	1	2
Total					
Average		0.02	2.19	2.19	9.25

Figure 4: dépendances entre les paquetages

3.3 Répartition des classes dans les paquetages

Pour chaque paquetage, on a généré le nombre de classes avec la commande :

```
$ find . -name "*.java" -exec grep -l "\bclass\b" {} \; | awk
-F'/' '{print $1"."$2}' | sort | uniq -c
```

¹spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/context/metrics/buffering/

²spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/web/servlet

Néanmoins il ne faut pas oublier le fait qu'il y a certains paquetages tels que **context**, **web**... qui contiennent eux aussi des paquetages et qu'il y a des classes qui ne sont dans aucun paquetage à part le paquetage principal **boot**¹.

En ajoutant une étude récursive de tous les dossiers qui contiennent des paquetages, on en a déduit que le nombre maximum de classes par paquetage est de 27 dans le paquetage **logback**² et le nombre minimum est 3 donc en moyenne on a 12.6 classes par paquetages. Au total, on a 1058 classes (sans les classes de tests).

package ^	C
org.springframework.boot.web.servlet.context	10
org.springframework.boot.web.servlet.error	3
org.springframework.boot.web.servlet.filter	6
org.springframework.boot.web.servlet.server	14
org.springframework.boot.web.servlet.support	8
org.springframework.boot.web.servlet.view	2
org.springframework.boot.webservices	
org.springframework.boot.webservices.client	8
Total	1,058
Average	12.60

Figure 5: Statistiques des classes dans les packages

Concernant la répartition des classes par paquetage, on remarque que toutes les classes sont dans le même paquetage (paquetage principal **boot**, équivalent d'un dossier src). Hormis ce fait il y a une grande répartition des 1058 classes sur 84 paquetages.

Les paquetages non-feuilles contiennent des classes et les paquetages qui ont le plus de classes dans une hiérarchie n'en ont pas nécessairement dans les autres. Il y a aussi une forte cohésion au sein des paquetages: par exemple dans le paquetage **admin**³ les deux classes servent à la gestion et au contrôle du projet via **JMX**⁴; dans le paquetage **validation**⁵ les classes gèrent l'interpolation des messages, c'est à dire rendre certains messages lisibles.

Avec cette organisation des classes dans les paquetages, on peut affirmer que les classes sont bien structurées avec de forte cohésion au sein des paquetages, c'est à dire que les fonctionnalités sont regroupées par paquetages, ce qui permet

¹spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot

²spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/logging/logback

³spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/admin

⁴permet de surveiller et gérer des applications

⁵spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/validation

une bonne visualisation et lecture du projet.

3.4 Organisation des classes

Ce projet est développé en Java, il est donc logique de s'intéresser au nombre de fichiers, classes et interfaces que contient ce projet.

```
# Pour compter le nombre de fichiers  
$ find -type f -name "*.java" | wc -l
```

Rapidement, voici une explication des commandes :

- `find` : Recherche des fichiers.
- `-type f` : Recherche de fichiers (exclusion des dossiers).
- `-name "*.java"` : Filtre pour sélectionner uniquement les fichiers Java.rs.
- `wc -l` : Compte le nombre de lignes contenant le mot recherché.

On trouve donc 793 fichiers java dans le package que nous avons sélectionné, ce qui fait une moyenne de 1.33 classe par fichier.

Etant donné l'envergure du projet et le fait que nous n'avons pas réussi à utiliser un outil d'analyse sur ce point, nous avons décidé pour notre analyse de réaliser manuellement un arbre d'héritage(DIT) uniquement que sur certains paquetages notamment **json**¹ et **orm**² et de calculer le nombre de sous-classes directes d'une classe (NOC).

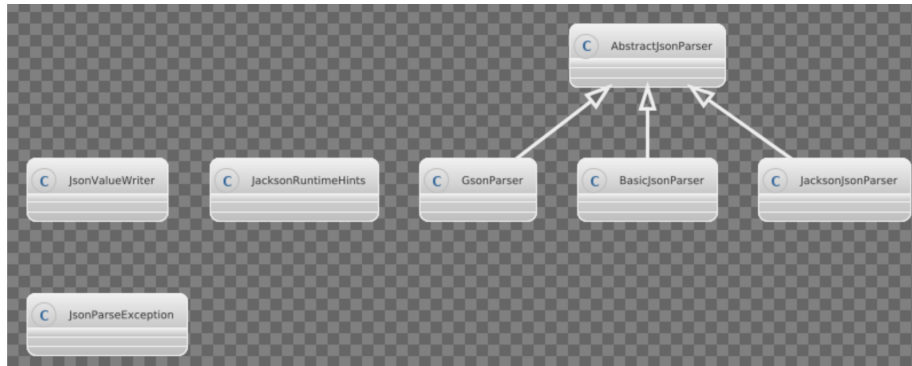


Figure 6: DIT pour le paquetage json

¹spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/json

²spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/orm

$$\begin{aligned}
\text{DIT max} &= 1 \\
\text{NOC min} &= 0 \\
\text{NOC max} &= 3 \\
\text{NOC moyenne} &= \frac{3}{1}
\end{aligned}$$



Figure 7: DIT pour le paquetage orm

$$\begin{aligned}
\text{DIT max} &= 1 \\
\text{NOC min} &= 0 \\
\text{NOC max} &= 2 \\
\text{NOC moyenne} &= \frac{4}{3}
\end{aligned}$$

A travers ces résultats et une étude visuelle des autres paquets, on remarque que l'organisation des classes est la même à quelques modifications près donc en général on a un DIT max à 1 et un NOC moyen à environ 3. On peut donc en conclure que nous sommes sur une hiérarchie plate. Ce type de hiérarchie facilite la compréhension du code, évite les dépendances complexes et les effets de bords dus à des héritages profonds. Malgré tout, cela apporte un inconvénient minime : la réutilisation du code pourrait être compliquée en cas de hiérarchie trop plate.

En se basant une fois de plus sur cette image ci-dessus et notre étude visuelle, il faut noter un faible couplage pour pratiquement toutes les classes, c'est à dire de faibles dépendances entre les classes donc la modification d'une classe aboutit à peu d'ajustement au niveau des autres classes, ce qui permet une meilleur stabilité, maintenance et flexibilité du projet au fil du temps.

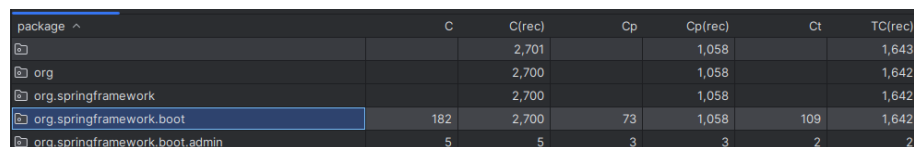
4 Pour aller plus loin

Dans cette section, nous allons nous appuyer sur le plugin d’IntelliJ nommé **CodeMetrics**. Ce plugin propose de sélectionner des paramètres dans un vaste panel, ce qui nous permet d’analyser méthodiquement les différentes classes. Toutes les captures d’écran viennent de cet outil et les graphiques sont issus de ce même outil, avec une surcouche pour le style. Les captures d’écran sources sont tout de même présentées à la fin de ce rendu.

Cet outil propose plusieurs niveaux de visibilité, en partant d’une analyse au niveau du fichier, puis en élargissant progressivement la portée jusqu’à afficher uniquement les éléments visibles à l’échelle du projet.

4.1 Tests

Sur cette capture d’écran, on peut voir qu’en surface, le code semble plutôt bien testé.



package ^	C	C(rec)	Cp	Cp(rec)	Ct	TC(rec)
org		2,701		1,058		1,643
org.springframework		2,700		1,058		1,642
org.springframework.boot	182	2,700	73	1,058	109	1,642
org.springframework.boot.admin	5	5	3	3	2	2

Figure 8: Tableau récapitulatif du nombre de lignes

Il y a un total de 2700 classes réparties en 1 058 classes¹ consacrées au code et 1 642 classes² de tests. Même s’il est normal d’avoir plus de classes de tests que de classes consacrées à la production, un ratio de 155 % laisse à penser que ce code est bien testé.

Ce projet utilise JUnit pour tester comme le montre les imports sur la capture d’écran ci-dessous :

¹Paramètre $CP(rec)$

²Paramètre $TC(rec)$

```

1 > /.../
16
17 package org.springframework.boot;
18
19 import org.junit.jupiter.api.AfterEach;
20 import org.junit.jupiter.api.Test;
21
22 import org.springframework.beans.factory.annotation.Autowired;
23 import org.springframework.context.ConfigurableApplicationContext;
24 import org.springframework.context.annotation.Bean;
25 import org.springframework.context.annotation.Configuration;
26 import org.springframework.context.annotation.Primary;
27
28 import static org.assertj.core.api.Assertions.assertThat;
29
30 /**
31  * Tests for {@link SpringApplication} {@link SpringApplication#setSources(java.util.Set)}
32  * source overrides}.
33  *
34  * @author Dave Syer
35  */
36 class OverrideSourcesTests { Complexity is 4 Everything is cool! no usages
37

```

Figure 9: Imports JUnit

Cependant, en regardant la moyenne de couverture, on remarque que la couverture des tests¹ ne concerne que 48 % des fonctionnalités des classes de production. Sachant que les méthodes privées ne sont pas incluses, cela indique que le projet est insuffisamment testé. Une couverture d'au moins 70 % est vivement recommandée, car avec seulement la moitié du code de production testé, de nombreux bugs peuvent passer inaperçus et compromettre la fiabilité et la maintenabilité du projet.

¹Paramètre *TEST.RAT*

package ^	Ct	LOCt	TEST_RAT
resources.org.springframework.boot.socket.netty		81	100.00%
resources.org.springframework.boot.ssl.pem		181	100.00%
resources.org.springframework.boot.ssl.pem.pkcs1		92	100.00%
resources.org.springframework.boot.ssl.pem.pkcs8		455	100.00%
resources.org.springframework.boot.ssl.pem.sec1		89	100.00%
resources.org.springframework.boot.web.embedded.netty		24	100.00%
resources.org.springframework.boot.web.embedded.tomcat		24	100.00%
resources.org.springframework.boot.web.reactive.server		81	100.00%
resources.org.springframework.boot.web.server		113	9.91%
resources.org.springframework.boot.web.server.pkcs1		92	100.00%
resources.org.springframework.boot.web.server.pkcs8		401	100.00%
resources.org.springframework.boot.web.server.sec1		88	100.00%
resources.org.springframework.boot.web.servlet.context		8	100.00%
resources.org.springframework.boot.web.servlet.server		81	100.00%
Total	1,643	76,044	
Average	14.41	535.52	48.52%

Figure 10: Imports JUnit

4.2 Commentaires

Une autre opération pour analyser la couverture est de compter le nombre de lignes de code bêtement, puis de compter le nombre de classes, méthodes et interfaces ... bien commentées, c'est-à-dire compter le nombre de JavaDoc présentes par rapport au nombre de classes ou bien la métrique indiquant le "True Comment Ratio" qui est proposée par l'IDE IntelliJ et permet d'enlever les espaces blancs, les sauts à la ligne, etc.

4.2.1 class

class ^	COM_RAT	Jf	Jm	TCOM_RAT
org.springframework.boot.web.servlet.server.StaticResourceJars	6.93%	100.00%	0.00%	7.45%
org.springframework.boot.web.servlet.ServletComponentHandler	13.64%	0.00%	0.00%	15.79%
org.springframework.boot.web.servlet.ServletComponentRegisteringPost	9.64%	0.00%	0.00%	10.67%
org.springframework.boot.web.servlet.ServletComponentScanRegistrar	15.91%	0.00%	0.00%	18.92%
org.springframework.boot.web.servlet.ServletComponentScanRegistrar.S	0.00%	0.00%	0.00%	0.00%
org.springframework.boot.web.servlet.ServletContextInitializerBeans	11.17%	16.67%	0.00%	26.06%
org.springframework.boot.web.servlet.ServletContextInitializerBeans.Filt	27.27%	100.00%	0.00%	37.50%
org.springframework.boot.web.servlet.ServletContextInitializerBeans.See	8.33%	0.00%	0.00%	9.09%
org.springframework.boot.web.servlet.ServletContextInitializerBeans.Ser	27.27%	100.00%	0.00%	37.50%
org.springframework.boot.web.servlet.ServletContextInitializerBeans.Ser	20.00%	0.00%	0.00%	17.65%
org.springframework.boot.web.servlet.ServletListenerRegistrationBean	45.54%	0.00%	75.00%	83.64%
org.springframework.boot.web.servlet.ServletRegistrationBean	50.62%	0.00%	81.25%	102.53%
org.springframework.boot.web.servlet.support.ErrorPageFilter	12.12%	9.09%	5.88%	15.50%
org.springframework.boot.web.servlet.support.ErrorPageFilter.ErrorWrap	5.56%	0.00%	0.00%	5.88%
org.springframework.boot.web.servlet.support.ErrorPageFilterConfigurat	26.32%	100.00%	0.00%	35.71%
org.springframework.boot.web.servlet.support.ServletContextApplicator	38.30%	0.00%	40.00%	62.07%
org.springframework.boot.web.servlet.support.SpringBootServletInitializ	47.37%	0.00%	66.67%	114.08%
org.springframework.boot.web.servlet.support.SpringBootServletInitializ	21.43%	0.00%	0.00%	27.27%
org.springframework.boot.web.servlet.support.SpringBootServletInitializ	18.18%	0.00%	0.00%	22.22%
org.springframework.boot.web.servlet.view.MustacheView	37.04%	0.00%	33.33%	58.82%
org.springframework.boot.web.servlet.view.MustacheViewResolver	38.78%	0.00%	50.00%	63.33%
org.springframework.boot.web.servlet.WebFilterHandler	13.16%	100.00%	0.00%	15.15%
org.springframework.boot.web.servlet.WebListenerHandler	27.78%	100.00%	0.00%	38.46%
org.springframework.boot.web.servlet.WebListenerHandler.ServletComp	0.00%	0.00%	0.00%	0.00%
org.springframework.boot.web.servlet.WebServletHandler	12.20%	100.00%	0.00%	13.89%
org.springframework.boot.WebApplicationType	46.34%	42.86%	0.00%	86.36%
org.springframework.boot.WebApplicationType.WebApplicationTypeRunti	0.00%	100.00%	0.00%	0.00%
org.springframework.boot.webservices.client.HttpWebServiceMessageSr	50.00%	0.00%	87.50%	100.00%
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	48.12%	0.00%	81.25%	109.77%
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	33.33%	0.00%	0.00%	50.00%
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	33.33%	0.00%	0.00%	50.00%
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	33.33%	0.00%	0.00%	50.00%
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	14.81%	0.00%	0.00%	17.39%
Total				
Average	28.95%	12.81%	25.89%	47.00%

Figure 11: Metriques des classes pour la javadoc

À ce niveau, la couverture des méthodes¹ est très faible, seulement $\sim 26\%$. Cela s'explique car la documentation est rédigée seulement si elle concerne un attribut public. De plus, les méthodes qui surchargent, ou bien celles qui implémentent, ne sont pas documentées, car documentées dans leur classe mère (ou interface).

¹Paramètre *Jm*


```

35  /**
36   * Provides access to the application home directory. Attempts to pick a sensible home for
37   * both Jar Files, Exploded Archives and directly running applications.
38   *
39   * @author Phillip Webb
40   * @author Raja Kolli
41   * @since 2.0.0
42   */
43  public class ApplicationHome { 16 usages  ⚡ Phillip Webb +6 Complexity is 17 You must be kidding
44
45      private final File source; 3 usages
46
47      private final File dir; 2 usages
48
49      /**
50       * Create a new {@link ApplicationHome} instance.
51       */
52  > public ApplicationHome() { this( sourceClass: null); }
53
54      /**
55       * Create a new {@link ApplicationHome} instance for the specified source class.
56       * @param sourceClass the source class or {@code null}
57       */
58  public ApplicationHome(Class<?> sourceClass) { 4 usages  ⚡ Phillip Webb Complexity is 3 Everything is cool!
59      this.source = findSource((sourceClass != null) ? sourceClass : getClass());
60      this.dir = findHomeDir(this.source);
61  }
62
63  @ private Class<?> getStartClass() { 1 usage  ⚡ Phillip Webb Complexity is 4 Everything is cool!
64      try {
65          ClassLoader classLoader = getClass().getClassLoader();
66          return getStartClass(classLoader.getResources( name: "META-INF/MANIFEST.MF"));
67      }
68      catch (Exception ex) {
69          return null;
70      }
71  }
72
73

```

Figure 12: Exemple : javadoc de la classe ApplicationHome

C'est, à mon avis, la bonne façon de documenter un framework, car seules les méthodes publiques seront utilisées par l'utilisateur. Seulement, dans un projet coopératif comme spring-boot avec 1146 contributeurs, les méthodes privées sont tout aussi utiles pour les développeurs. Même si la plupart des méthodes non documentées sont assez compréhensibles, il existe certaines méthodes un peu moins explicites, telle que :

```

/unchecked/ 1 usage  ⚡ Phillip Webb +1
private void postProcessBeforeInitialization(WebServerFactory webServerFactory) {
    LambdaSafe.callbacks(WebServerFactoryCustomizer.class, getCustomizers(), webServerFactory)
        .withLogger(WebServerFactoryCustomizerBeanPostProcessor.class)
        .invoke(( WebServerFactoryCustomizer customizer) -> customizer.customize(webServerFactory));
}

```

Figure 13: Oubli de javadoc

4.2.2 interface

interface ^	CLOC	COM_RAT	JLOC	Jm	LOC	NCLOC	TCOM_RAT
org.springframework.boot.web.embedded.undertow.ConfigurableUndertow	63	80.77%	63	100.00%	78	15	420.00%
org.springframework.boot.web.embedded.undertow.HttpHandlerFactory	17	80.95%	17	100.00%	21	4	425.00%
org.springframework.boot.web.embedded.undertow.UndertowBuilderCustomizer	11	73.33%	11	100.00%	15	4	275.00%
org.springframework.boot.web.embedded.undertow.UndertowDeploymentInfo	11	73.33%	11	100.00%	15	4	275.00%
org.springframework.boot.web.embedded.undertow.UndertowWebServer	3	60.00%	3	100.00%	5	2	150.00%
org.springframework.boot.web.reactive.context.ConfigurableReactiveWebServerFactory	6	66.67%	6	100.00%	9	3	200.00%
org.springframework.boot.web.reactive.context.ConfigurableReactiveWebServer	7	77.78%	7	100.00%	9	2	350.00%
org.springframework.boot.web.reactive.error.ErrorWebExceptionHandler	6	75.00%	6	100.00%	8	2	300.00%
org.springframework.boot.web.reactive.error.ErrorWebExceptionHandler	26	78.79%	26	100.00%	33	7	371.43%
org.springframework.boot.web.reactive.error.ErrorWebExceptionHandler	7	70.00%	7	100.00%	10	3	233.33%
org.springframework.boot.web.reactive.filter.OrderedWebFilter	9	75.00%	9	100.00%	12	3	300.00%
org.springframework.boot.web.reactive.function.client.WebClientCustomizer	14	77.78%	14	100.00%	18	4	350.00%
org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory	6	75.00%	6	100.00%	8	2	300.00%
org.springframework.boot.web.reactive.server.ReactiveWebServerFactory	16	80.00%	16	100.00%	20	4	400.00%
org.springframework.boot.web.server.ConfigurableWebServerFactory	50	80.65%	50	100.00%	62	12	416.67%
org.springframework.boot.web.server.ErrorPageRegistrar	10	71.43%	10	100.00%	14	4	250.00%
org.springframework.boot.web.server.ErrorPageRegistry	10	71.43%	10	100.00%	14	4	250.00%
org.springframework.boot.web.server.GracefulShutdownCallback	11	73.33%	11	100.00%	15	4	275.00%
org.springframework.boot.web.server.WebServer	37	77.08%	37	100.00%	48	11	336.36%
org.springframework.boot.web.server.WebServerFactory	9	81.82%	9	100.00%	11	2	450.00%
org.springframework.boot.web.server.WebServerFactoryCustomizer	22	84.62%	22	100.00%	26	5	550.00%
org.springframework.boot.web.servlet.error.ErrorAttributes	23	79.31%	23	100.00%	29	6	383.33%
org.springframework.boot.web.servlet.error.ErrorController	8	80.00%	8	100.00%	10	2	400.00%
org.springframework.boot.web.servlet.filter.OrderedFilter	9	75.00%	9	100.00%	12	3	300.00%
org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory	88	83.81%	88	100.00%	105	17	517.65%
org.springframework.boot.web.servlet.server.CookieSameSiteSupplier	73	66.36%	73	100.00%	110	37	197.30%
org.springframework.boot.web.servlet.server.ServletWebServerFactory	17	80.95%	17	100.00%	21	4	425.00%
org.springframework.boot.web.servlet.ServletContextInitializer	24	85.71%	24	100.00%	28	4	600.00%
org.springframework.boot.web.servlet.ServletContextInitializerBeans.Registration	6	60.00%	6	0.00%	10	5	150.00%
org.springframework.boot.web.servlet.WebListenerRegistrar	10	76.92%	10	100.00%	13	3	333.33%
org.springframework.boot.web.servlet.WebListenerRegistry	10	76.92%	10	100.00%	13	3	333.33%
org.springframework.boot.web.services.client.WebServiceMessageSender	29	65.91%	29	100.00%	44	15	193.33%
org.springframework.boot.web.services.client.WebServiceTemplateCustomizer	10	71.43%	10	100.00%	14	4	250.00%
Total	3,575		4,123		5,123	1,800	
Average	23.37	69.78%	26.95	93.66%	33.48	11.76	412.81%

Figure 14: Métriques des interfaces pour la javadoc

Au niveau des interfaces, il y a de grosses proportions de code commenté, que ce soit en prenant le ratio le plus basique¹ ou celui des méthodes². Ce n'est cependant pas très pertinent d'analyser le ratio brut car les interfaces sont souvent très courtes (on peut d'ailleurs le remarquer car le ratio "True Comment"³ plante et affiche 413 %). Aussi, on peut remarquer que le ratio de la documentation des méthodes compense les métriques des classes.

¹Paramètre *COM_RAT*

²Paramètre *Jm*

³Paramètre *TCOM_RAT*

4.2.3 package

package ^	Jc(rec)	Jf(rec)	Jm(rec)	TCOM_RAT(rec)
org	90.89%	13.86%	31.88%	18.27%
org.springframework	90.89%	13.86%	31.88%	18.27%
org.springframework.boot	90.89%	13.86%	31.88%	18.27%
org.springframework.boot.admin	66.67%	0.00%	21.05%	28.12%
org.springframework.boot.ansi	100.00%	5.00%	31.25%	26.19%
org.springframework.boot.availability	100.00%	66.67%	45.00%	40.70%
org.springframework.boot.builder	83.33%	0.00%	59.02%	10.13%
org.springframework.boot.cloud	100.00%	58.33%	38.89%	12.53%
org.springframework.boot.context	94.30%	12.04%	29.56%	16.87%
org.springframework.boot.context.annotation	100.00%	0.00%	48.00%	21.80%
org.springframework.boot.context.config	98.31%	16.86%	38.29%	15.87%
org.springframework.boot.context.event	90.00%	0.00%	54.55%	35.57%
org.springframework.boot.context.logging	50.00%	29.17%	15.62%	7.79%
org.springframework.boot.context.metrics	80.00%	0.00%	35.48%	19.23%
org.springframework.boot.context.metrics.buffering	80.00%	0.00%	35.48%	19.23%
org.springframework.boot.context.properties	93.38%	10.24%	25.49%	16.65%
org.springframework.boot.context.properties.bind	95.00%	5.77%	24.86%	15.50%
org.springframework.boot.context.properties.bind.handle	75.00%	0.00%	9.09%	30.50%
org.springframework.boot.context.properties.bind.validat	100.00%	0.00%	10.81%	20.38%
org.springframework.boot.context.properties.source	92.50%	16.49%	23.99%	15.20%
org.springframework.boot.convert	95.00%	35.56%	16.76%	25.50%
org.springframework.boot.diagnostics	76.67%	0.00%	7.69%	24.85%
org.springframework.boot.diagnostics.analyzer	70.83%	0.00%	1.80%	21.38%
org.springframework.boot.env	87.50%	15.62%	12.90%	15.18%
org.springframework.boot.flyway	100.00%		0.00%	142.86%
org.springframework.boot.http	76.47%	11.11%	27.18%	14.27%
org.springframework.boot.http.client	76.47%	11.11%	27.18%	14.27%
org.springframework.boot.info	86.96%	8.16%	21.57%	14.61%
org.springframework.boot.io	100.00%	0.00%	32.14%	18.58%
org.springframework.boot.jackson	71.43%	22.22%	35.56%	19.59%
org.springframework.boot.jdbc	80.49%	41.79%	28.66%	19.17%
org.springframework.boot.jdbc.init	100.00%	0.00%	50.00%	28.17%
org.springframework.boot.jdbc.metadata	100.00%	0.00%	23.81%	39.59%
org.springframework.boot.jms	100.00%		60.00%	56.79%

Figure 15: Metriques des packages pour la javadoc

Les packages n'ont plus de visibilité sur les attributs "private", mais à la différence du projet dans son ensemble, les packages ont accès aux attributs "protected" présents dans le même package.

La grosse différence a lieu quant à la documentation des classes : Comme toutes les classes privées ne sont plus comptabilisées, le projet atteint une couverture de $\sim 91\%$,

4.2.4 project

project ^	CLOC	COM_RAT	J	Je	Jf	JLOC	Jm	L(J)	NCLOC	TCOM_RAT
project	31,748	40.53%	793	100.00%	13.86%	19,643	31.88%	78,332	46,591	68.14%

Figure 16: Metriques du projet pour la javadoc

La documentation des classes est donc parfaite pour un utilisateur voulant utiliser ce framework car toutes les classes qui lui sont disponibles sont documentées, mais il manque tout de même de la documentation dans les attributs privés ou protégés pour un projet aussi collaboratif que spring-boot

Pour mieux se représenter, voici un graphique avec le pourcentage de javadoc des classes en fonction de la visibilité :

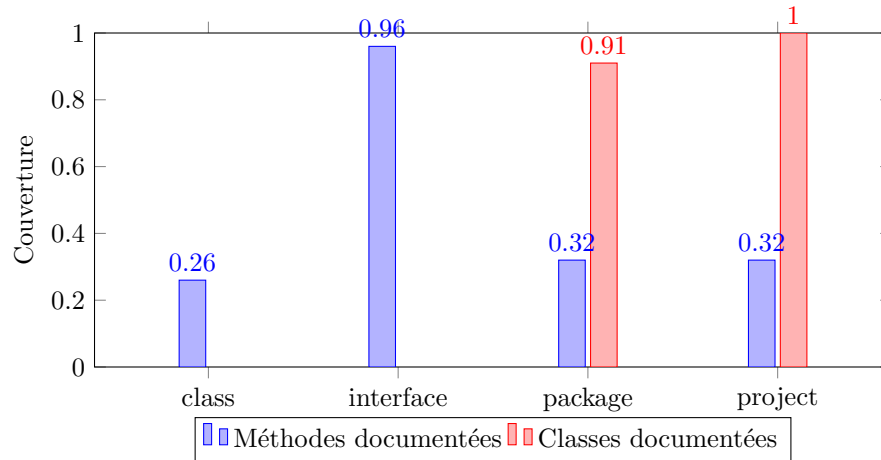


Figure 17: Couverture par catégorie et légende

On peut en conclure que ce projet est bien documenté. Il a pour vocation d'être un framework, et donc d'être utilisé par un utilisateur externe et probablement ignorant du code sous-jacent. Cependant, même si les méthodes sont souvent explicites, il peut être un peu contraignant de se lancer dans l'étude voire l'amélioration de ce code sans aide intérieure.

4.3 Les méthodes

4.3.1 Statistiques

Ce projet compte un total de **20 894** méthodes réparties dans 1058 classes, ce qui nous donne une moyenne de 24.35 méthodes par classe. La classe qui compte le plus de méthodes est **TomcatServletWebServerFactory**¹ et en compte 165

¹[spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/web/embedded/tomcat](https://github.com/spring-projects/spring-boot/blob/main/spring-boot/src/main/java/org/springframework/boot/web/embedded/tomcat)

alors que la classe **HttpComponentsClientHttpRequestFactoryBuilder**¹ en compte seulement 13, ce qui en fait la classe avec le moins de méthodes.

class ^	CSO	OSavg
org.springframework.boot.WebApplicationType.WebApplicationTypeRunTime	15	3.50
org.springframework.boot.webservices.client.HttpWebServiceMessageSender	21	2.88
org.springframework.boot.webservices.client.WebServiceTemplateBuilder	45	2.91
org.springframework.boot.webservices.client.WebServiceTemplateBuilder	16	1.00
org.springframework.boot.webservices.client.WebServiceTemplateBuilder	16	1.00
org.springframework.boot.webservices.client.WebServiceTemplateBuilder	16	1.00
org.springframework.boot.webservices.client.WebServiceTemplateBuilder	19	1.17
Total	20,894	
Average	24.35	3.07

Figure 18: Nombre de méthodes

Il est également utile de regarder la complexité cyclomatique des méthodes. On peut voir que la complexité totale² est de **9430** ce qui correspond à une moyenne de **1.72**³. Comme la complexité est très basse en moyenne, ce projet semble bien organisé avec des méthodes courtes et faciles à maintenir.

method ^	ev(G)	iv(G)	LOOP_NEST	v(G)
org.springframework.boot.webservice:	1	1	0	1
org.springframework.boot.webservice:	1	1	0	1
parse(String, ChronoUnit)	1	1	0	2
parseInt(Matcher, int)	1	2	0	2
print(Duration, ChronoUnit)	1	1	0	1
print(Period, ChronoUnit)	1	1	0	1
shouldNotFilterAsyncDispatch()	1	1	0	1
Total	6,749	8,239		9,430
Average	1.23	1.51	0.08	1.72

Figure 19: Complexité cyclomatique des méthodes

4.3.2 God Classes

Grâce au logiciel CodeMR, nous pouvons voir qu'il existe quelques classes ayant des valeurs élevées traduisant des God Classes. Toutefois, la liste montre assez rapidement des paramètres allant de *low* à (quelques) *medium-high*, preuve qu'il n'y a pas beaucoup de classes centralisant trop de logique ou de fonctionnalités.

¹spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/http/client

²Paramètre *Total* : $v(G)$

³Paramètre *Average* : $v(G)$

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHERION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHERION	SIZE
1	Binder	■	■	■	■	263	medium-high	high	medium-high	low-medium
2	SpringApplication	■	■	■	■	685	medium-high	medium-high	very-high	high
3	ConfigurationProp...	■	■	■	■	132	medium-high	medium-high	low-medium	low-medium
4	LoggingApplicatio...	■	■	■	■	209	low-medium	medium-high	high	low-medium
5	Log4J2LoggingSystem	■	■	■	■	277	medium-high	low-medium	medium-high	low-medium
6	LogbackLoggingSystem	■	■	■	■	288	low-medium	low-medium	medium-high	low-medium
7	SpringApplication...	■	■	■	■	180	low-medium	low-medium	high	medium-high
8	JsonValueWriter	■	■	■	■	158	low-medium	low-medium	high	low-medium
9	ServletWebServerA...	■	■	■	■	152	low-medium	low-medium	low	low-medium
10	MapBinder	■	■	■	■	124	low-medium	low-medium	medium-high	low-medium
11	NoUnboundElements...	■	■	■	■	79	low-medium	low-medium	medium-high	low-medium
12	JavaBeanBinder	■	■	■	■	228	low	low-medium	low	low-medium
13	ConfigDataEnviron...	■	■	■	■	219	low	low-medium	low	low-medium

Figure 20: Liste des God Classes CodeMR

L'outil MetricsReloaded nous fournit une liste de Brain Classes et 4 types de God Classes que voici :

Profile Name	Metric Profile	Class
Brain Class	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3 ∧ TCC < 0.5 ∧ WMC ≥ 34	UndertowServletWebServerFactory
Brain Method	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3	ConfigurationPropertyName
Complex Method	CC ≥ 8	SpringApplication
Data Class	NOAM ≥ 4 ∧ NOPA ≥ 3 ∧ WMC < 15 ∧ WOC < 0.34	ElementsParser
Deeply Nested Conditions	CND ≥ 3	UndertowWebServer
Dispersed Coupling	CDSP ≥ 0.66 ∧ CINT ≥ 8 ∧ MND ≥ 2	JettyWebServer
Feature Envy	ATFD ≥ 5 ∧ FDP ≥ 5 ∧ LAA < 0.33	ServletWebServerApplicationContext
God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0.33 ∧ WMC ≥ 47	Binder
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44	TomcatWebServer
God Class (type 3)	ATFD ≥ 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44	JsonValueWriter
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0.37 ∧ WMC ≥ 46	Mappings
High Coupling	CBO ≥ 20	OriginTrackedPropertiesLoader
Intensive Coupling	CDSP < 0.5 ∧ CINT ≥ 8 ∧ MND ≥ 2	StandardConfigDataLocationResolver
Long Method	LOC ≥ 16	ConfigDataEnvironmentContributor
Long Parameters List	NORM ≥ 4	JettyServletWebServerFactory
Too Many Fields	NOA ≥ 15	AbstractReactiveWebServerFactoryTests
Too Many Methods	NOM ≥ 10	JettyServletWebServerFactoryTests

Figure 21: Liste des Brain Classes

God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0.33 ∧ WMC ≥ 47
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44
God Class (type 3)	ATFD ≥ 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0.37 ∧ WMC ≥ 46

Figure 22: Les 4 types de God Classes

Tout d'abord, voici une explication des différents paramètres pris en compte par Metrics :

1. **CC**(Cyclomatic Complexity) mesure la complexité d'une méthode en fonction du nombre de chemins d'exécution possibles.

2. **MND**(Maximum Nested Depth) mesure la profondeur maximale des boucles et conditions
3. **WMC**(Weighted Methods per Class) mesure la complexité d'une classe en fonction du nombre de méthodes et de leur complexité. Si cette valeur est élevée, cela indique une classe complexe.
4. **CBO**(Coupling Between Objects) mesure le degré de couplage entre une classe et d'autres.
5. **AFTD**(Access to Foreign Data) mesure le nombre d'accès aux données d'autres classes.
6. **RFC**(Response for a Class) mesure le nombre de méthodes pouvant être exécutées en réponse à un message reçu par cette classe.
7. **TCC**(Tight Class Cohesion) mesure à quel point les méthodes sont liées. Inverse de LCOM
8. **NOA**(Number of Attributes) mesure le nombre d'attributs de la classe.
9. **LOC**(Lines of Code) mesure le nombre de lignes de code
10. **NOAV**(Number of Accessed Variables) mesure le nombre de variables accédées par une méthode

Les classes qui nous intéressent sont les Brain Classes et les 4 types de God Classes. Une Brain Class est une classe qui a une complexité élevée¹, une faible cohésion², une imbrication excessive³ et beaucoup d'accès aux variables⁴. Ce sont des classes qui centralisent trop la logique du framework et sont trop complexes. Nous allons maintenant voir les God Classes relevées par Metrics et leurs différents types

Le premier type regroupe les classes qui accèdent largement aux données d'autres classes⁵, tout en présentant une faible cohésion interne et une complexité élevée. Leur forte interaction avec d'autres classes, combinée à cette complexité et à ce manque de cohésion, complique leur maintenance. En conséquence, ces classes deviennent difficiles à réutiliser ou à tester, car elles sont trop dépendantes d'autres éléments du code et intègrent trop de fonctionnalités.

Une classe appartient au second type lorsqu'elle a un nombre élevé d'attributs⁶. Bien que l'accès aux données externes soit modéré, la classe reste très complexe. Ces classes gèrent trop de données et peuvent indiquer une mauvaise encapsulation ou une violation du principe de responsabilité unique. De plus, sa complexité rend la classe difficile à mettre à jour et à étendre.

¹Paramètres *WMC* & *CC*

²Paramètre *TCC*

³Paramètre *MND*

⁴Paramètre *NOAV*

⁵Paramètre *AFTD*

⁶Paramètre *NOA*

Le troisième type est marqué par un couplage élevé avec d'autres classes¹ ainsi qu'un accès fréquent aux données externes tout en ayant une complexité élevée. Comme le couplage est élevé, il peut être difficile de modifier cette classe sans changer le comportement des classes où elle est utilisée.

Le dernier type possède au moins un couplage avec d'autres classes, un nombre très élevé de méthodes pouvant être exécutées en réponse à un message² (Cette métrique mesure le nombre total de méthodes qui peuvent être invoquées lorsqu'une méthode de la classe est appelée. Cela inclut les méthodes de la classe elle-même et les méthodes d'autres classes qui sont appelées directement ou indirectement) et une faible cohésion interne. Ce type de classe, avec un couplage élevé, un grand nombre de méthodes invoquées et une faible cohésion interne, devient difficile à maintenir et à comprendre. Ces caractéristiques compliquent les modifications sans affecter d'autres parties du code, ce qui nuit à sa flexibilité et à son évolutivité.

En résumé, il est utile de mettre en lumière ces classes, car ce seront vraisemblablement elles qui causeront des problèmes lors de la maintenance. L'idée de les séparer selon des paramètres en différents types permet d'analyser plus efficacement quelle classe posera problème en fonction de l'action de mise à jour à effectuer. De plus, la séparation entre Brain Classes et God Classes fait un filtrage préliminaire et divise les classes en deux catégories : celles qui gèrent trop la logique (Brain Classes) et celles qui implémentent trop de fonctionnalités ou qui dépendent trop les unes des autres (God Classes).

Voici un graphique représentant le nombre de classes dans le projet :

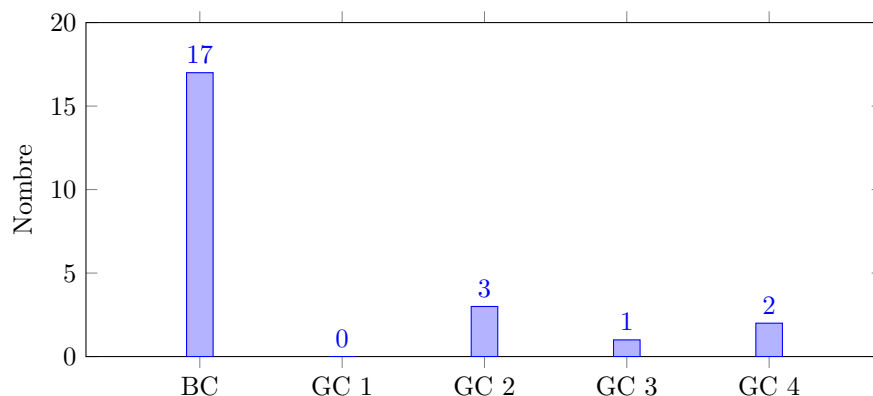


Figure 23: Nombre de classes en fonction du type

Dans notre projet, nous comptons 17 Brain Classes, 0 God Classes de type 1, 3 de type 2, 1 de type 3 et 2 de type 4. Cela représente, à l'échelle du projet, 2 % des classes (23 sur 1058). Bien que ce nombre puisse paraître faible, ces classes peuvent avoir un impact significatif sur la maintenabilité et

¹Paramètre *CBO*

²Paramètre *RFC*

l'évolutivité du projet en raison de leur complexité et de leur forte centralisation des responsabilités.

4.4 Code inutile

Il est ensuite utile d'analyser s'il reste du code inutile. C'est-à-dire du code qui a pu être utilisé autrefois mais qui est désormais obsolète ou bien des méthodes vides etc.

Comme le montre la capture d'écran ci-dessous, il y a bien du code inutile, par exemple des méthodes vides. On peut compter 1 méthode de la sorte pour la classe **JettyServletWebServerFactory**¹ et 3 pour **SpringApplication**².

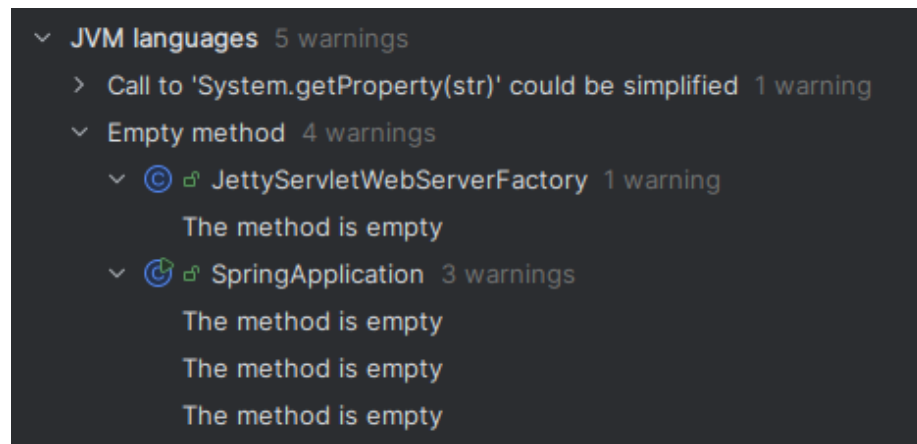


Figure 24: Capture d'écran des fonctions vides

De plus, il existe 6 classes qui sont dépréciées et qui pourraient être supprimées. On remarque que **ClientHttpRequestFactory** apparaît deux fois car il y a la présence d'un lien vers une autre classe dépréciée et visiblement retirée du projet **ClientHttpRequestFactorySettings**. On peut également remarquer 2 appels à `printStackTrace()`, ce qui est probablement lié au débogage et n'a rien à faire dans le projet final. Il y a un appel dans **ExitCodeGenerators** et un autre dans **SpringApplication** qui affichent des codes de sortie, ce qui est du débogage.

¹spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/web/embedded/jetty

²spring-boot/spring-boot-project/spring-boot/src/main/java/org/springframework/boot

class ^	CSO	OSavg
org.springframework.boot.WebApplicationType.WebApplicationTypeRunTi	15	3.50
org.springframework.boot.webservices.client.HttpWebServiceMessageS	21	2.88
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	45	2.91
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	16	1.00
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	16	1.00
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	16	1.00
org.springframework.boot.webservices.client.WebServiceTemplateBuilde	19	1.17
Total	20,894	
Average	24.35	3.07

Figure 25: Métriques des méthodes

On peut tout de même noter que ces classes sont annotées de ”@Deprecated” et ne sont plus utilisées ailleurs :

```

25  /**
26   * Logback {@link ClassicConverter} to convert the
27   * {@link LoggingSystemProperty#APPLICATION_NAME APPLICATION_NAME} into a value suitable
28   * for logging. Similar to Logback's {@link PropertyConverter} but a non-existent property
29   * is logged as an empty string rather than {@code null}.
30   *
31   * @author Andy Wilkinson
32   * @author Phillip Webb
33   * @since 3.2.4
34   * @deprecated since 3.4.0 for removal in 3.6.0 in favor of
35   *   {@link EnclosedInSquareBracketsConverter}
36   */
37  @Deprecated(since = "3.4.0", forRemoval = true) 1 Andy Wilkinson +1
38  public class ApplicationNameConverter extends ClassicConverter { Complexity is 7 It's time to do something...
39
40      private static final String ENVIRONMENT_VARIABLE_NAME = LoggingSystemProperty.APPLICATION_NAME 2 usages
41          .getEnvironmentVariableName();
42
43      @Override 1 Phillip Webb +1
44      public String convert(ILoggingEvent event) { Complexity is 6 It's time to do something...
45          String applicationName = event.getLoggerContextVO().getPropertyMap().get(ENVIRONMENT_VARIABLE_NAME);
46          applicationName = (applicationName != null) ? applicationName : System.getProperty(ENVIRONMENT_VARIABLE_NAME);
47          return (applicationName != null) ? applicationName : "";
48      }
49
50  }

```

Figure 26: Capture d'écran d'une classe dépréciée

5 Nettoyage de Code et Code smells

Dans cette section, nous procéderons à l'analyse du code dans les détails, plus précisément les règles de nommage des variables, méthodes, tests et les nombres qui apparaissent dans le code, sans oublier la structure du code et l'existence de code mort.

5.1 Règles de nommage

Le choix des noms dans tous le programme a été fait de manière descriptif, de manière à guider les programmeurs ou toute autre personne dans l'implémentation des différents éléments; par exemple dans la classe **SpringApplicationAdminMXBeanRegistrar.java**¹ il y a la méthode **shutdown()** qui indique par son nom l'arrêt ordonné d'une application et aussi la méthode **onApplicationReadyEvent()** qui indique un gestionnaire d'évènement.

Dans ce contexte, on interprète structure de donnée comme des classes ou objets utilisés dans la conception du projet. En se basant sur cette définition on remarque que les classes et objets sont plutôt bien nommées, on arrive à cerner l'idée générale des fonctionnalités dont la classe est en charge. En général, la majorité des noms sont tous prononçables sauf quelques uns comme **SpringApplicationAdminMXBeanRegistrar.java**, **LazyInitializationBeanFactoryPostProcessor**, **AotInitializerNotFoundException**.

5.2 Nombre magique

On constate l'existence de nombre magique dans certaines classes grâce à cette commande

```
$ grep -E "\b\d+(\.\d+)?\b" -r | grep -vE "(0|1|2)" | grep -v "const"
```

Par exemple la classe **CorrelationIdFormatter.java**² contient : **(padding ≤ 0)** à la ligne 178, **matcher.group(1)** à la ligne 193 et **matcher.group(2)** à la ligne 195. La classe **PeriodStyle.java**³ contient: 1,2,3,4,7 de la ligne 53 à 57. En regardant bien le code où se trouve tous ces nombres magiques, on a réussi à comprendre uniquement le **(padding ≤ 0)** à la ligne 178 qui semble être une condition pour savoir si l'on doit ajouter du remplissage (**padding**), on pourrait remplacer ce nombre magique par une constante nommée **MIN_PADDING**, ce qui nous en dirait plus sur la fonction de ce nombre. Cette écriture du code avec les nombres magiques a un impact prépondérant sur la lisibilité, le refactoring et la compréhension du code donc pour éviter tous ces problèmes, il serait plus intéressant de nommer tous les nombres magiques avec des noms clairs qui reflètent leur rôle.

¹spring-boot-project/spring-boot/src/main/java/org/springframework/boot/admin/SpringApplicationAdminMXBeanRegistrar.java

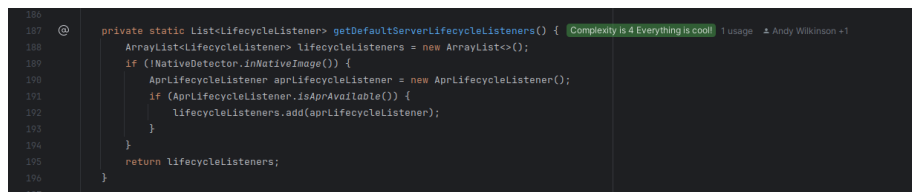
²spring-boot-project/spring-boot/src/main/java/org/springframework/boot/logging/CorrelationIdFormatter.java

³spring-boot-project/spring-boot/src/main/java/org/springframework/boot/convert/PeriodStyle.java

5.3 Structure du code

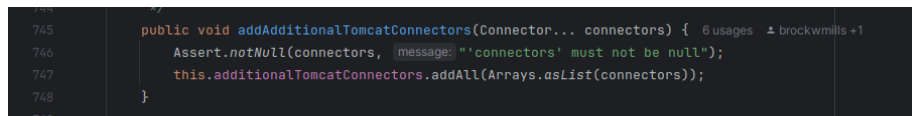
Dans la section 4.3, la classe *TomcatServletWebServerFactory* était la classe qui comptait le plus de méthodes (165) et la classe **HttpComponentsClientHttpRequestFactoryBuilder** était celle qui en comptait le moins (13). La classe **Binder** quant à elle était la classe qui ressortait en premier lors de l'analyse de la complexité des classes. Nous allons donc analyser ces 3 classes et la classe sélectionnées au hasard dans l'arborescence de fichiers afin d'avoir un panel assez étendu afin d'analyser la structure du code et d'en tirer des conclusions assez globales.

La classe **TomcatServletWebServerFactory** présente bien ses attributs d'instance en premier. Cependant, les méthodes publiques ne sont pas forcément les premières à apparaître et les classes ne sont pas regroupées par visibilité. Cependant, grâce à l'historique de modification du fichier, nous pouvons voir que les fonctions ne sont pas classées par ordre chronologique, ce qui pourrait indiquer que les fonctions sont, par contre, regroupées par utilité. Cependant, dans cette classe les fonctions ayant beaucoup d'usages ne sont pas forcément en haut. Comme nous travaillons sur uniquement sur le package spring-boot, cette donnée atteint rapidement sa limite.



```
186  
187  
188 private static List<LifecycleListener> getDefaultValueListeners() { Complexity is 4 Everything is cool! 1 usage Andy Wilkinson +1  
189     ArrayList<LifecycleListener> lifecycleListeners = new ArrayList<>();  
190     if (!NativeDetector.inNativeImage()) {  
191         AprLifecycleListener aprLifecycleListener = new AprLifecycleListener();  
192         if (AprLifecycleListener.isAprAvailable()) {  
193             lifecycleListeners.add(aprLifecycleListener);  
194         }  
195     }  
196     return lifecycleListeners;  
197 }
```

Figure 27: Fonction pas utilisée en haut du code



```
745  
746 public void addAdditionalTomcatConnectors(Connector... connectors) { 6 usages Brockwills +1  
747     Assert.notNull(connectors, "message: 'connectors' must not be null");  
748     this.additionalTomcatConnectors.addAll(Arrays.asList(connectors));  
749 }
```

Figure 28: Fonction utilisée loin dans le code

La classe **Binder** suit le même schéma et semble classer selon l'importance des méthodes.org.springframework.boot.context.properties.bind;

La classe **HttpComponentsClientHttpRequestFactoryBuilder** présente également ses attributs d'instance en haut. Cette classe est assez récente (la première modification a eu lieu le 24 octobre 2024), et semble regrouper les fonctions par utilité. Elles ne sont donc pas classées. Cette technique atteint sa limite lorsqu'il n'y a plus aucune ligne de code du premier commit inchangée.

```

745     public void addAdditionalTomcatConnectors(Connector... connectors) {
746         Assert.notNull(connectors, message: "'connectors' must not be null");
747         this.additionalTomcatConnectors.addAll(Arrays.asList(connectors));
748     }

```

Figure 29: Classe `HttpComponentsClientHttpRequestFactoryBuilder`

Finalement, **spring-boot** à l'air d'être organisé selon une logique interne au développeur, mais une logique qui est respectée. Toutes les variables d'instance sont déclarées dans les premières lignes de la classe et ensuite les fonctions sont regroupées par utilité car elles ne sont ni classées chronologiquement, ni par visibilité.

6 Annexe

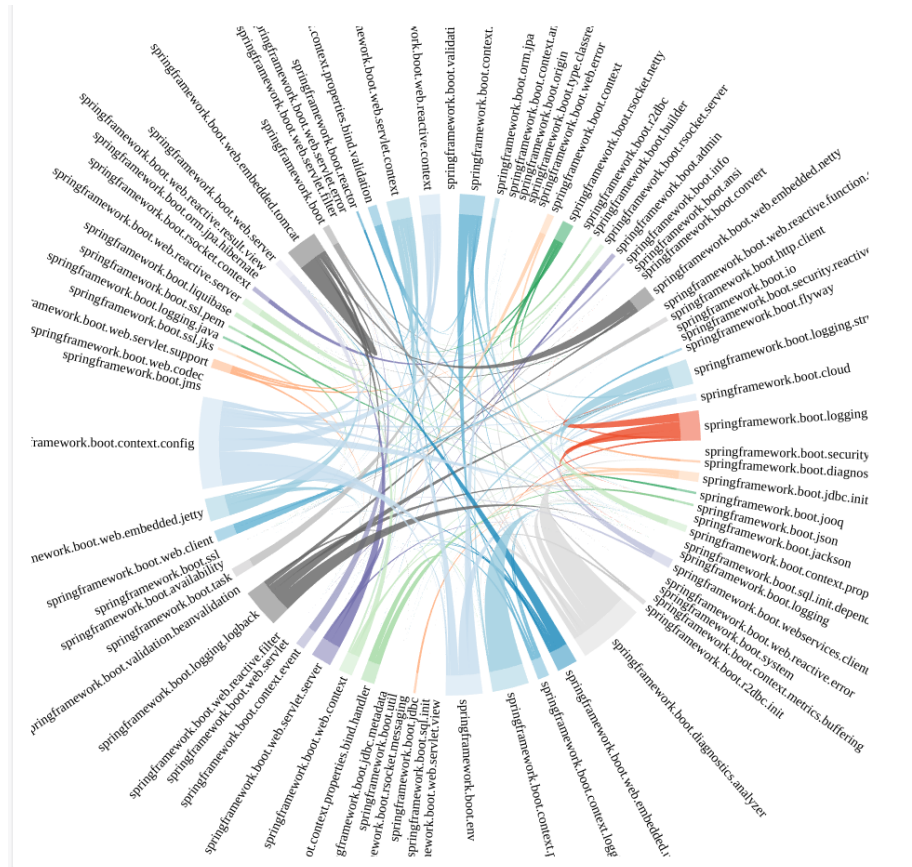


Figure 30: Dépendences cycliques des packages

Profile Name	Metric Profile	Class
Brain Class	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3 ∧ TCC < 0.5 ∧ WMC ≥ 34	
Brain Method	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3	
Complex Method	CC ≥ 8	
Data Class	NOAM ≥ 4 ∧ NOPA ≥ 3 ∧ WMC < 15 ∧ WOC < 0.34	
Deeply Nested Conditions	CND ≥ 3	
Dispersed Coupling	CDISP ≥ 0.66 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Feature Envy	ATFD ≥ 5 ∧ FDR ≥ 5 ∧ LAA < 0.33	
God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0.33 ∧ WMC ≥ 47	
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44	
God Class (type 3)	ATFD < 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44	
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0.37 ∧ WMC ≥ 46	
High Coupling	CBO ≥ 20	
Intensive Coupling	CDISP < 0.5 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Long Method	LOC ≥ 16	
Long Parameters List	NOPM ≥ 4	
Too Many Fields	NOA ≥ 15	
Too Many Methods	NOM ≥ 10	

Figure 31: Liste God Class Type 1

Brain Class	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3 ∧ TCC < 0.5 ∧ WMC ≥ 34	UndertowServletWebServerFactory
Brain Method	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3	JettyServletWebServerFactory
Complex Method	CC ≥ 8	TomcatServletWebServerFactory
Data Class	NOAM ≥ 4 ∧ NOPA ≥ 3 ∧ WMC < 15 ∧ WOC < 0.34	
Deeply Nested Conditions	CND ≥ 3	
Dispersed Coupling	CDISP ≥ 0.66 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Feature Envoy	ATFD ≥ 5 ∧ FDP ≥ 5 ∧ LAA < 0.33	
God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0.33 ∧ WMC ≥ 47	
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44	
God Class (type 3)	ATFD ≥ 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44	
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0.37 ∧ WMC ≥ 46	
High Coupling	CBO ≥ 20	
Intensive Coupling	CDISP < 0.5 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Long Method	LOC ≥ 16	
Long Parameters List	NOPM ≥ 4	
Too Many Fields	NOA ≥ 15	
Too Many Methods	NOM ≥ 10	

Figure 32: Liste God Class Type 2

Profile Name	Metric Profile	Class
Brain Class	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3 ∧ TCC < 0.5 ∧ WMC ≥ 34	Processor
Brain Method	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3	
Complex Method	CC ≥ 8	
Data Class	NOAM ≥ 4 ∧ NOPA ≥ 3 ∧ WMC < 15 ∧ WOC < 0.34	
Deeply Nested Conditions	CND ≥ 3	
Dispersed Coupling	CDISP ≥ 0.66 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Feature Envoy	ATFD ≥ 5 ∧ FDP ≥ 5 ∧ LAA < 0.33	
God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0.33 ∧ WMC ≥ 47	
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44	
God Class (type 3)	ATFD ≥ 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44	
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0.37 ∧ WMC ≥ 46	
High Coupling	CBO ≥ 20	
Intensive Coupling	CDISP < 0.5 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Long Method	LOC ≥ 16	
Long Parameters List	NOPM ≥ 4	
Too Many Fields	NOA ≥ 15	
Too Many Methods	NOM ≥ 10	

Figure 33: Liste God Class Type 3

Profile Name	Metric Profile	Class
Brain Class	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3 ∧ TCC < 0.5 ∧ WMC ≥ 34	AbstractServletWebServerFactoryTests
Brain Method	CC ≥ 3 ∧ LOC ≥ 30 ∧ MND ≥ 3 ∧ NOAV ≥ 3	SpringApplication
Complex Method	CC ≥ 8	
Data Class	NOAM ≥ 4 ∧ NOPA ≥ 3 ∧ WMC < 15 ∧ WOC < 0.34	
Deeply Nested Conditions	CND ≥ 3	
Dispersed Coupling	CDISP ≥ 0.66 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Feature Envoy	ATFD ≥ 5 ∧ FDP ≥ 5 ∧ LAA < 0.33	
God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0.33 ∧ WMC ≥ 47	
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44	
God Class (type 3)	ATFD ≥ 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44	
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0.37 ∧ WMC ≥ 46	
High Coupling	CBO ≥ 20	
Intensive Coupling	CDISP < 0.5 ∧ CINT ≥ 8 ∧ MND ≥ 2	
Long Method	LOC ≥ 16	
Long Parameters List	NOPM ≥ 4	
Too Many Fields	NOA ≥ 15	
Too Many Methods	NOM ≥ 10	

Figure 34: Liste God Class Type 4