



Projet GL _ GSON

Réalisé par :

- ☐ Yannick MOUNGUENGUI
- ☐ NOUHA BOUKILI

Licence 3 Info groupe 1 GL

1] Présentation globale du projet

1.1. Utilité du projet

. Quel projet on a choisis?

On a choisi d'étudier la qualité logicielle du projet Gson qui se trouve dans le lien qui suit : <https://github.com/google/gson>

. Que fait le projet?

Gson est une librairie qu'on peut ajouter aux projets java dans le but de convertir des objets Java vers leurs représentations JSON. Ce n'est pas le premier projet open-source qui s'intéresse à la conversion des annotations Java vers Json, pourtant ce projet met en valeur deux points très importants contrairement aux autres, l'utilisation des Java Generics (with the aim of reducing bugs and adding an extra layer of abstraction over types) et aussi la possibilité de convertir vers JSON sans ajouter les annotations Java dont on a pas accès à leurs code source tout le temps.

. Quels sont les buts du projet?

- fournir les méthodes toJson() et fromJson() pour convertir à JSON et vice-versa.
- permettre les objets Java qui définis précédemment comme étant non modifiables à se convertir vers JSON.
- utiliser les java Generics.
- permettre les représentations clients aux objets
- prise en charge d'objets arbitrairement complexes (avec des hiérarchies d'héritage profondes et une utilisation intensive de types génériques)

. Comment lancer le projet?

Comme le projet est sous forme d'une bibliothèque Java, on peut l'installer en l'ajoutant aux dépendances en deux manières :

façon 1 : par les dépendances du pom.xml pour un projet Maven comme suit :

Maven:

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
```

façon 2 : par les dépendances Gradle :

```
dependencies {
  implementation 'com.google.code.gson:gson:2.10.1'
}
```

. Quel est le résultat du projet : quelles sont les sorties du logiciel ?

Le but principal du projet c'est pouvoir utiliser les méthodes qui convertissent au Json.

La classe principale à utiliser est Gson qu'on peut simplement créer en appelant `new Gson()`. Il existe également une classe `GsonBuilder` disponible qui peut être utilisée pour créer une instance `Gson` avec divers paramètres tels que le contrôle de version, etc.

L'instance `Gson` ne conserve aucun état lors de l'appel d'opérations JSON. Ainsi, on est libre de réutiliser le même objet pour plusieurs opérations de sérialisation et de désérialisation JSON.

On peut donner comme exemple d'utilisation du projet ce qui suit:

// Serialization

```
Gson gson = new Gson();
gson.toJson(1); // ==> 1
gson.toJson("abcd"); // ==> "abcd"
gson.toJson(new Long(10)); // ==> 10
int[] values = { 1 };
gson.toJson(values); // ==> [1]
```

// Deserialization

```
int i = gson.fromJson("1", int.class);
Integer intObj = gson.fromJson("1", Integer.class);
Long longObj = gson.fromJson("1", Long.class);
Boolean boolObj = gson.fromJson("false", Boolean.class);
String str = gson.fromJson("\"abc\"", String.class);
String[] strArray = gson.fromJson("[\"abc\"]", String[].class);
```

1.2. Description du projet

. Le readme est-il pertinent?

Il y a bien un README dans ce projet, qui de plus est à jour (dernière modification effectuée le 6 janvier 2023 pour annoncer la sortie du gson-parent-2.10.1). Ce dernier présente globalement le projet sans entrer dans les détails. On peut ainsi retrouver une brève introduction pour se mettre dans le contexte, les objectifs du projet ou encore des liens utiles vers des documentations.

. La documentation du projet est-elle présente?

En plus de l'extrait de documentation retrouvée dans le README, il y a à disposition une documentation supplémentaire détaillée et complète du projet dans le fichier UserGuide.md. On y retrouve également des informations sur

- l'installation
- l'utilisation : nous constatons qu'il y a suffisamment d'exemples permettant d'illustrer clairement comment utiliser Gson sans avoir besoin de consulter le code source
- ou encore l'utilité du projet.

. Les informations en terme de lancement sont-elles suffisantes ?

Toutefois, les informations en termes d'installation et de lancement ne sont pas très explicites, on peut avoir des bugs lors de la première installation.

2] Historique du projet

2.1. Analyse du git

. Quel est le nombre de contributeurs?

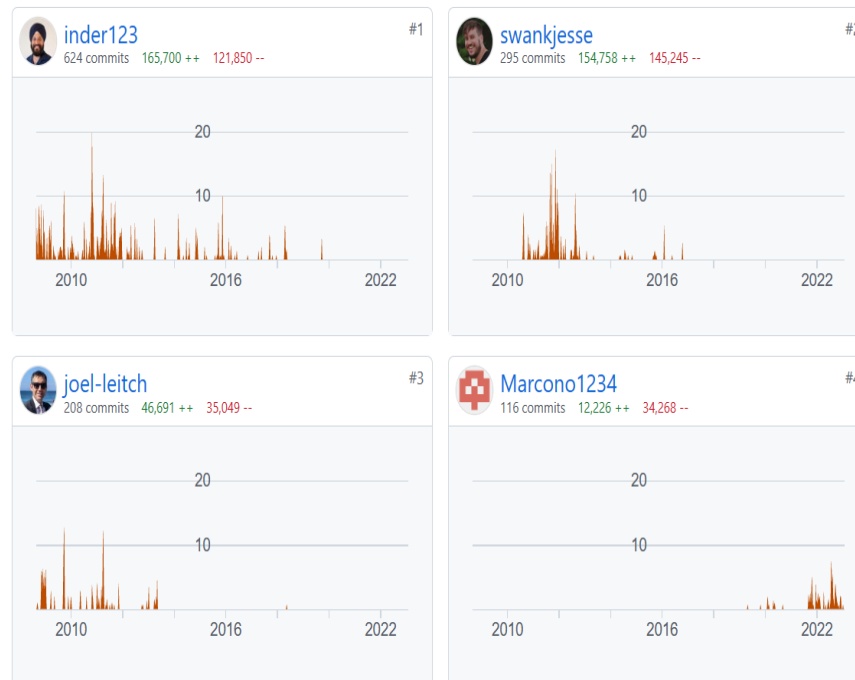
D'après le dépôt Github, 137 personnes contribuent pour l'instant au développement du projet, ce qui est bien important comme nombre de développeurs travaillant sur ce projet.

. Est ce qu'ils ont contribué de façon équilibré sur le projet?
et dans le temps?

Non, on peut constater clairement que les contributions ne sont pas équilibrées, on trouve certains qui ont fait 624 commits et d'autres qui ont fait qu'un seul.

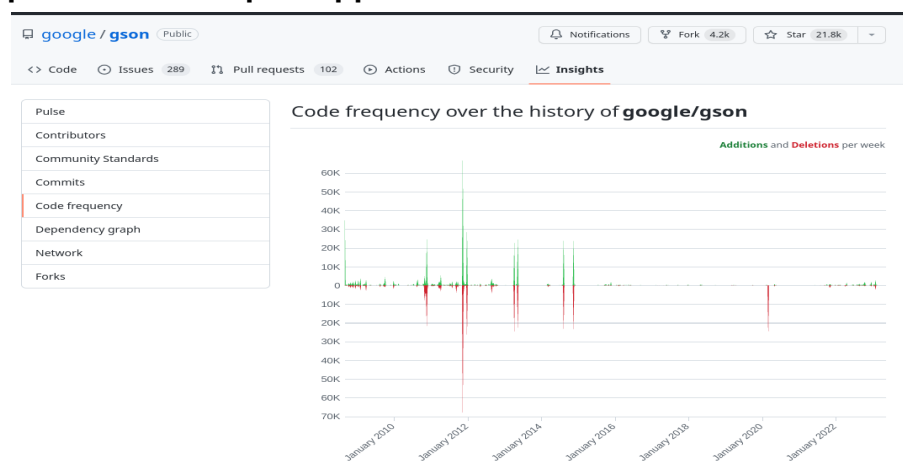
Aussi, on trouve une majorité de personnes à travailler en 2010, contrairement à d'autres qui ont commencé en 2022.

On donne un exemple des contributions d'utilisateurs dans l'image qui suit:



. Le projet est-il toujours actif? l'activité est-elle répartie régulièrement sur le temps?

Le projet est toujours actif, on trouve des modifications faites le jour même ou on vérifie, mais l'activité n'est pas régulière, on trouve que les ajouts et les suppressions en 2012 sont beaucoup plus nombreux par rapport à ces dernières années.



Combien de branches sont définies? et combien parmi elles sont utilisées?

Le projet est constitué de 15 branches dont 6 sont closed et le reste est open. On peut dire que 15 branches est un nombre moyen regardant le projet, mais les 6 branches non utilisées sont non négligeables.

. Le mécanisme de pull request est-il utilisé?

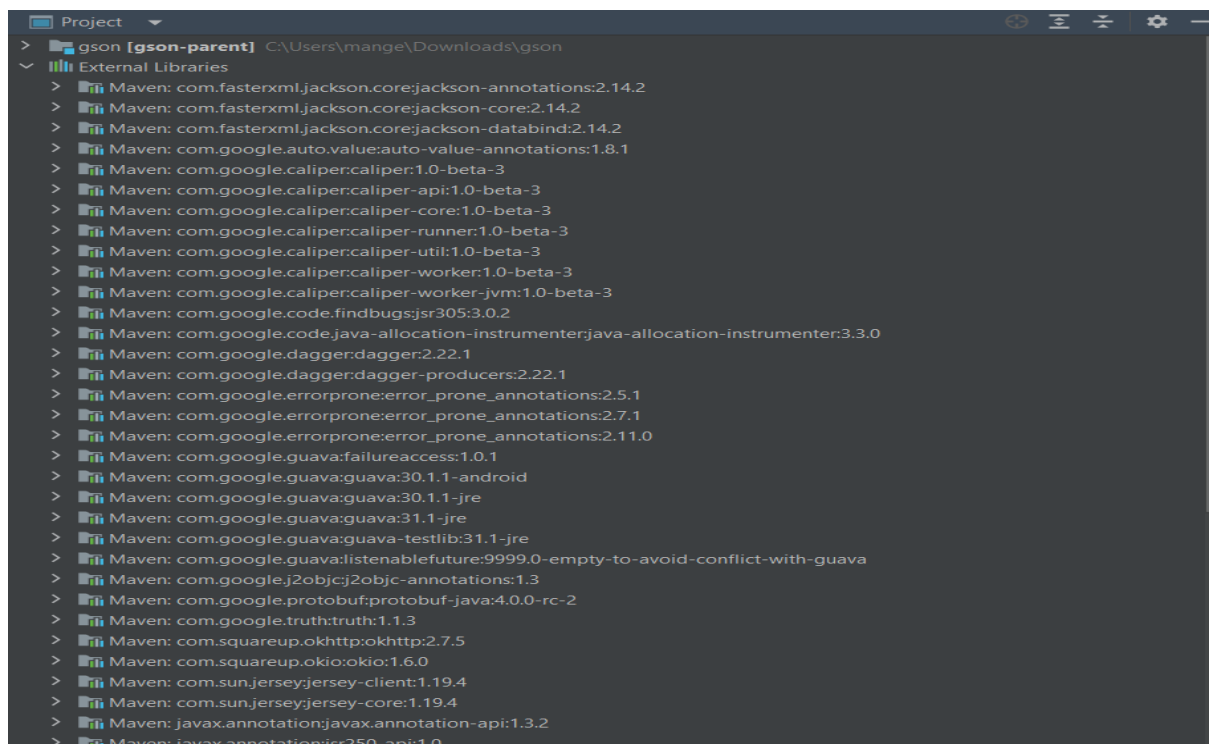
Il y a 102 demandes de pull qui sont en attente et 627 demandes qui sont closed, c'est un nombre assez grand de pull requests, ce qui montre que le projet est utilisé par une grande diversité de personnes.

3] Architecture Logicielle

3.1. Utilisations des bibliothèques extérieures

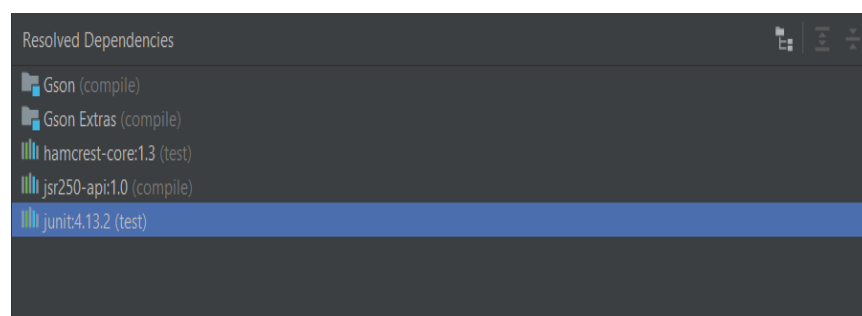
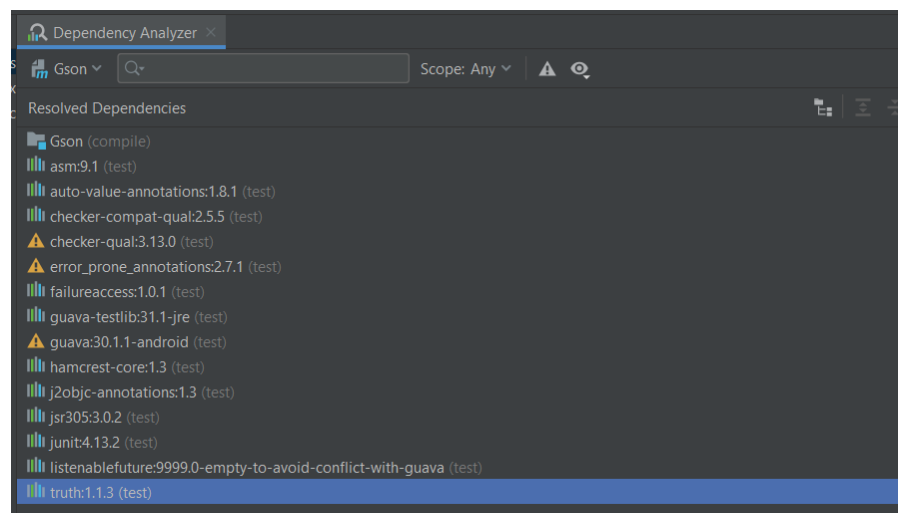
. Quel est le nombre des bibliothèques extérieures?

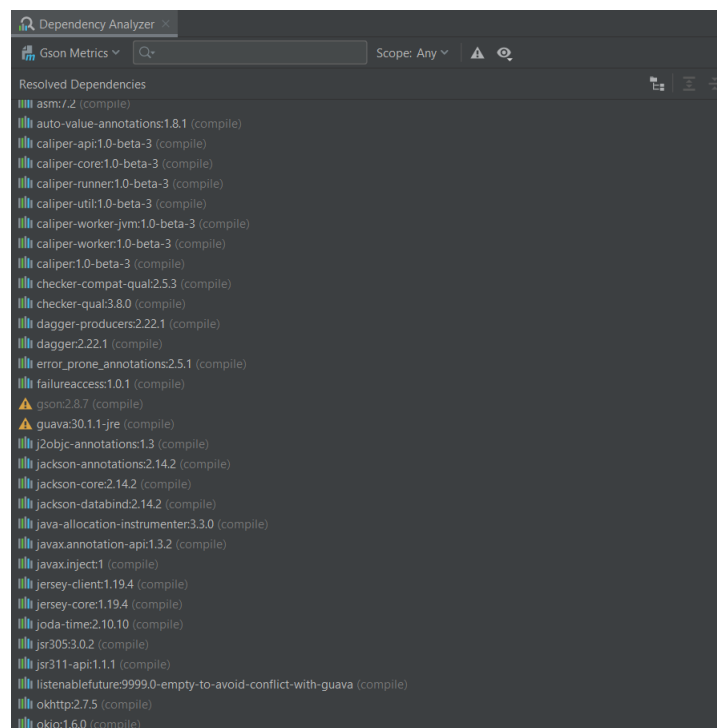
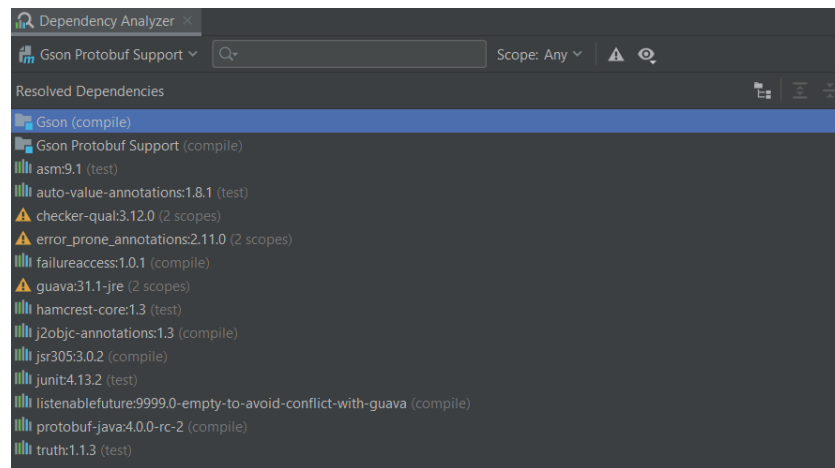
A l'aide de IntelliJ IDEA, nous avons trouvé qu'il y a 49 bibliothèques extérieures Maven (on les compte sur l'image qui suit). On trouve que 49 est un nombre assez grand de bibliothèques par rapport à ce projet, mais ce qui est bien étonnant c'est qu'ils sont tous des bibliothèques Maven. La gestion des dépendances est importante dans tout projet, car elle peut affecter la qualité, la maintenabilité et la sécurité du code.



. Analysons la différence entre les bibliothèques référencées et celles utilisées : certaines dépendances sont-elles donc inutiles?

Toutes les bibliothèques sont utilisées dans les différents dossiers du projet. Cela signifie que toutes les dépendances référencées sont nécessaires pour le bon fonctionnement du code. Cependant, il est possible que certaines dépendances ne soient pas utilisées directement dans les classes(gson:30.1.1-android sur la photo qui suit par exemple), mais qu'elles soient nécessaires pour les bibliothèques tierces utilisées dans le projet. Alors on peut dire qu'elles sont toutes utiles.





. Analysons les bibliothèques réellement utilisées : que nous apprend l'utilisation de ces bibliothèques ? A-t-on, par exemple, plusieurs bibliothèques qui servent à la même chose (par exemple plusieurs ORM, ou bibliothèques graphiques) ? Est-ce justifié ?

D'après l'image montrant globalement les bibliothèques extérieures, nous voyons que certaines d'entre elles servent à l'annotation comme

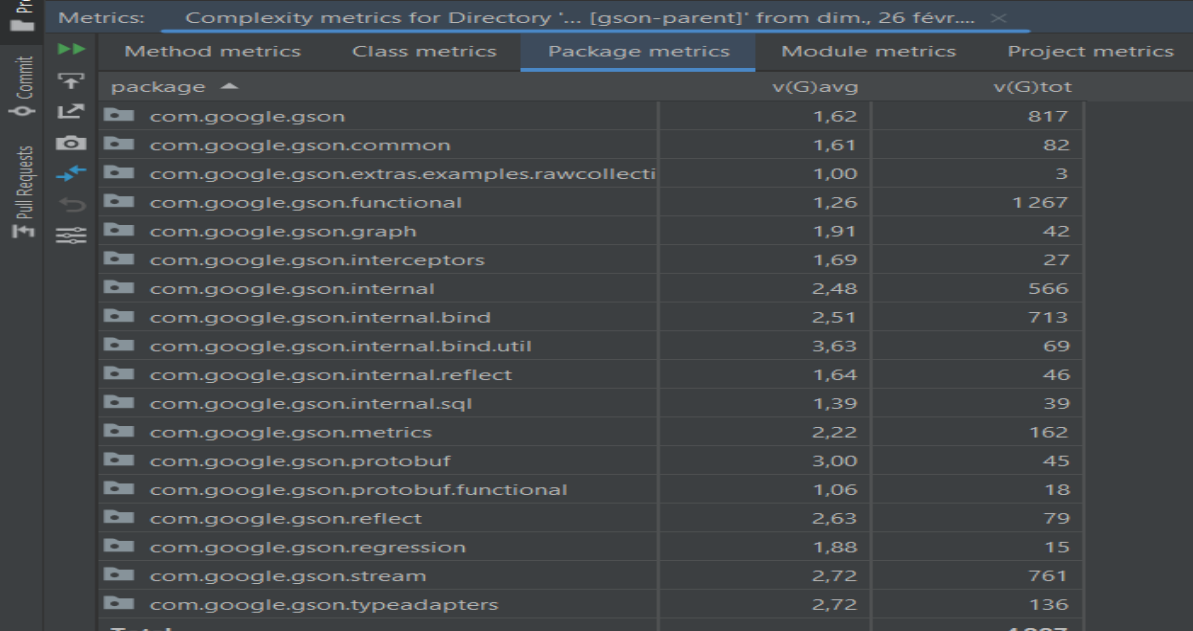
Maven:com.fasterxml.jackson.core:jackson-annotations et **Maven:com.google.auto.value:auto-value-annotations** servant aux développeurs de marquer le code avec des informations supplémentaires qui peuvent être utilisées par le compilateur, les outils de génération de code, les frameworks de développement

ou d'autres outils tiers. Pour améliorer l'utilisation des bibliothèques d'annotations dans un projet Java on peut éviter les dépendances inutiles et considérer l'utilisation de bibliothèques alternatives.

3.2. Organisation en paquetages

. Quel est le nombre de paquetages?

L'analyse a montré que le nombre de paquetages de ce projet vaut 18, ce n'est pas un grand nombre par rapport à un projet de cette taille, on pouvait avoir plus mais il faut savoir aussi qu'un nombre élevé de paquetages peut rendre le projet plus modulaire et plus facile à gérer, mais peut également introduire une complexité supplémentaire si la structure n'est pas bien conçue. Une amélioration possible est d'éviter les paquetages trop petits ou trop grands : évitez les paquetages trop petits, car ils peuvent introduire de la complexité supplémentaire. Évitez également les paquetages trop grands, car ils peuvent rendre la gestion plus difficile.



Method metrics	Class metrics	Package metrics	Module metrics	Project metrics
package			v(G)avg	v(G)tot
com.google.gson			1,62	817
com.google.gson.common			1,61	82
com.google.gson.extras.examples.rawcollecti			1,00	3
com.google.gson.functional			1,26	1 267
com.google.gson.graph			1,91	42
com.google.gson.interceptors			1,69	27
com.google.gson.internal			2,48	566
com.google.gson.internal.bind			2,51	713
com.google.gson.internal.bind.util			3,63	69
com.google.gson.internal.reflect			1,64	46
com.google.gson.internal.sql			1,39	39
com.google.gson.metrics			2,22	162
com.google.gson.protobuf			3,00	45
com.google.gson.protobuf.functional			1,06	18
com.google.gson.reflect			2,63	79
com.google.gson.regression			1,88	15
com.google.gson.stream			2,72	761
com.google.gson.typeadapters			2,72	136
Total				4 887

. Analysons les liens entre les paquetages : quels paquetages dépendent de quels autres ? Les paquetages sont-ils organisés en couche ? Existe-t-il des cycles entre paquetages ?

Il y a plusieurs paquetages qui dépendent d'autres comme:

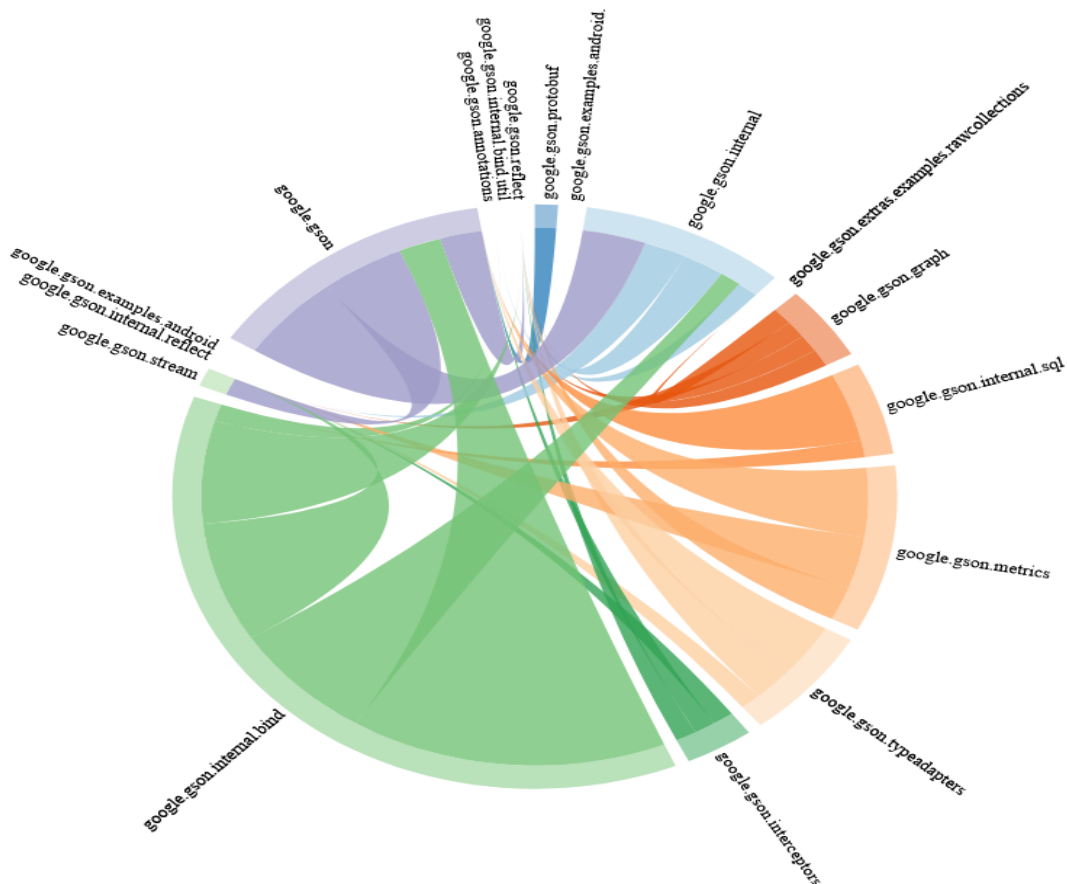
- le paquetage " `gson.internal.bind` " qui dépend beaucoup du paquetage `gson`
- le paquetage `gson` dépend beaucoup du paquetage `gson.internal`

. Les paquetages sont organisés en plusieurs couches. Le paquetage `gson.internal.sql` est en cycle avec le paquetage `google.gson` .

Une amélioration possible à faire serait de refactoriser le code : Il peut être utile de refactoriser le code pour séparer les fonctionnalités et réduire les dépendances croisées.

L'image qui suit montre les dépendances de chaque sous paquet:

Package Dependencies
Hover on the wheel to see the details



. Analysons la hiérarchie de paquetage : quel est le nombre de niveaux de paquetage ? La hiérarchie de paquetages pour les tests suit-elle la hiérarchie de paquetages des sources ? Ou ailleurs a-t-on des hiérarchies parallèles (ou presque) ? Existe-t-il des paquetages qui ne contiennent qu'un seul paquetage sans aucune classe ?

Le projet étudié comporte des niveaux de paquetage allant de 2 à 5. Bien que l'on aurait pu s'attendre à un niveau de paquetage plus élevé pour un projet de cette envergure, la hiérarchie de paquetages des classes sources et tests permet de facilement identifier les tests correspondant à une classe ou un paquetage spécifique. Chaque paquetage contient au moins une classe et il est important de noter que la hiérarchie de paquetages peut avoir un impact significatif sur la maintenabilité et la compréhension du code. La majorité des tests suit la même hiérarchie de paquetages que les sources, ce qui peut faciliter la maintenance et la compréhension du code de test. Un paquetage ne contenant qu'un seul paquetage sans classe peut indiquer une mauvaise utilisation des paquetages. Une amélioration possible serait d'éviter les paquetages avec une seule classe ou aucune classe.

Metrics: Complexity metrics for Directory '... [gson-parent]' from dim., 26 févr....				
	Method metrics	Class metrics	Package metrics	Module metrics
package			v(G)avg	v(G)tot
com.google.gson			1,62	817
com.google.gson.common			1,61	82
com.google.gson.extras.examples.rawcollecti			1,00	3
com.google.gson.functional			1,26	1 267
com.google.gson.graph			1,91	42
com.google.gson.interceptors			1,69	27
com.google.gson.internal			2,48	566
com.google.gson.internal.bind			2,51	713
com.google.gson.internal.bind.util			3,63	69
com.google.gson.internal.reflect			1,64	46
com.google.gson.internal.sql			1,39	39
com.google.gson.metrics			2,22	162
com.google.gson.protobuf			3,00	45
com.google.gson.protobuf.functional			1,06	18
com.google.gson.reflect			2,63	79
com.google.gson.regression			1,88	15
com.google.gson.stream			2,72	761
com.google.gson.typeadapters			2,72	136
Total				4 887

. Analysons les noms des paquetages : les noms des paquetages nous apprennent-ils des choses sur les designs patterns utilisés (MVC par exemple) ? Sur l'existence d'un lien avec une base de données ? Que nous apprennent les noms des paquetages de façon plus générale ?

Nous constatons que le paquetage commun à tout le projet est *com.google.gson* qui est le nom de domaine du projet. En revanche, les autres noms de paquetages sont fortement liés au domaine métier du projet. Ainsi, il est difficile d'avoir une idée précise de ce qu'ils peuvent contenir comme on le ferait par exemple pour *Model-View-Controller*. Les noms des paquetages peuvent nous fournir des informations utiles sur leur contenu, leur fonctionnalité, leurs dépendances et leur version. Une amélioration possible serait d'adopter une convention de nommage standardisée pour les packages liés au domaine métier du projet, afin de faciliter leur compréhension et leur utilisation.

3.3. Répartition des classes dans les paquetages

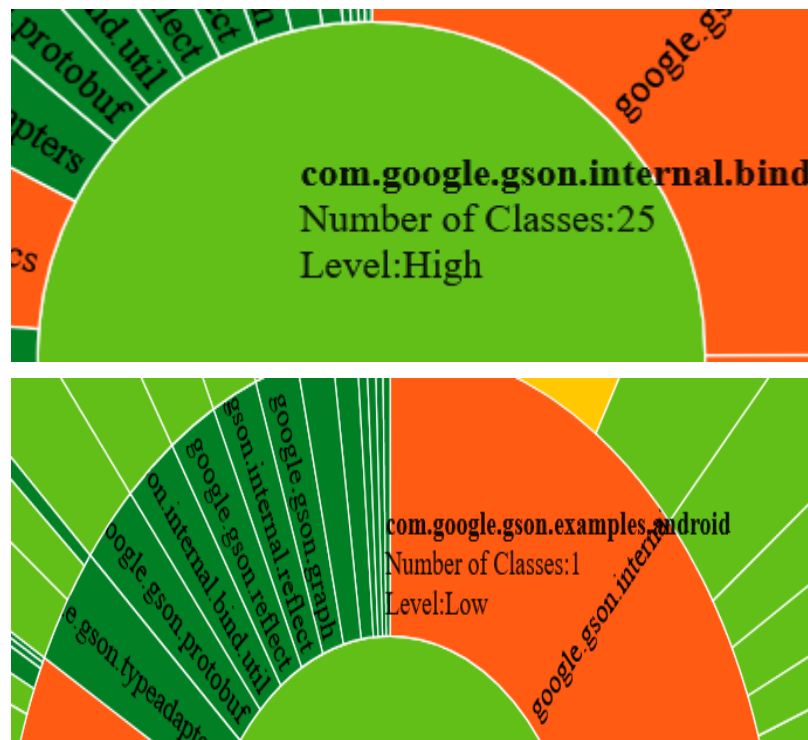
. Quel est le nombre minimum, maximum par paquetage et la moyenne? Également, on pourra fournir le nombre total de classes

Le projet a un total de 639 classes réparties en paquetages avec une moyenne de 7,3 classes par paquetage et un maximum de 25 classes par paquetage. Ces chiffres peuvent indiquer que le projet est de taille importante et bien modulaire, avec des classes cohérentes. Cependant, d'autres mesures telles que la complexité cyclomatique, le taux de couplage et le taux de cohésion doivent être analysées pour évaluer plus en détail la qualité et la maintenabilité du code. Les améliorations possibles pour cela sont: Réduire la complexité cyclomatique des classes qui ont une complexité élevée. Cela peut être fait en extrayant des méthodes plus petites et plus spécialisées à partir de méthodes trop longues ou trop complexes.

Réduire le taux de couplage en limitant les dépendances entre les classes et en utilisant des interfaces pour accéder aux implémentations de classes.

Améliorer la cohésion en regroupant les méthodes et les attributs dans des classes de manière à ce qu'ils aient une fonctionnalité et un but communs.

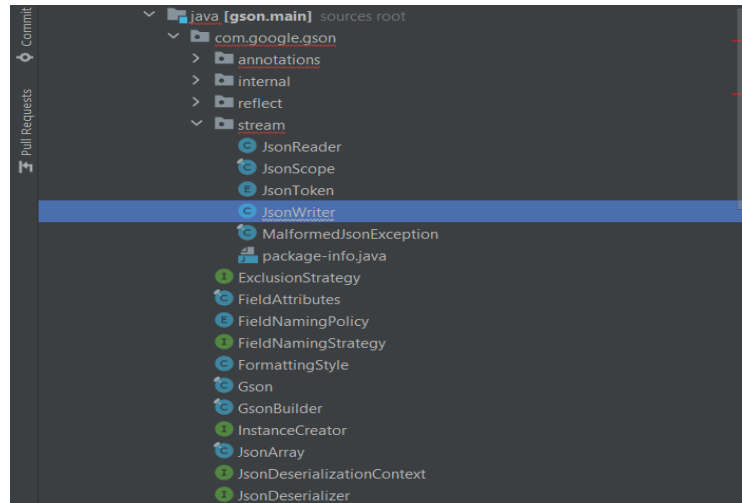
On a utilisé Code Monsieur afin d'avoir des informations plus précises concernant le nombre de classes et le niveau par rapport aux autres paquetages, on trouve les résultats qui suivent par exemple:



Analysis of gson	
General Information	
Total lines of code:	6409
Number of classes:	100
Number of packages:	9
Number of external packages:	17
Number of external classes:	539

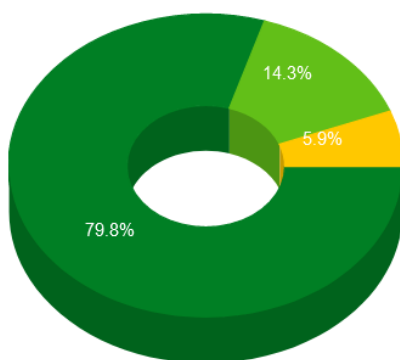
. Analysons la répartition des classes dans les différents paquetages : toutes les classes ou presque sont-elles dans le même paquetage ? Est-ce que des paquetages non feuilles contiennent des classes ? S'il y a plusieurs hiérarchies parallèles, les paquetages qui ont le plus de classes dans une hiérarchie ont-ils aussi le plus de classes dans les autres ?

Presque toutes les classes sont majoritairement dans un paquetage(`com.google.gson`) et nous avons constaté que les paquetages non feuilles contiennent également des classes. Cela peut indiquer une certaine indépendance et modularité dans l'organisation des différentes fonctionnalités du projet. Pour améliorer cela, on peut refactoriser les classes existantes pour les placer dans les paquetages nouvellement créés.

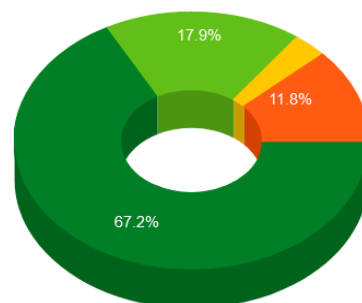


. Analysons le couplage et la cohésion au sein de quelques paquetages en particulier.

Nous constatons donc que ce projet a plus de classes avec un couplage faible qu'un couplage fort. Cela est le même cas concernant le manque de cohésion des classes. Il est possible que les classes aient été conçues sans une bonne organisation ou une planification adéquate, ce qui a entraîné une mauvaise structure. De plus, cela peut être dû à une mauvaise compréhension des responsabilités de chaque classe. Pour améliorer la situation, il est possible d'appliquer les principes SOLID qui visent à concevoir des classes faciles à comprendre, à maintenir et à étendre.



Coupling



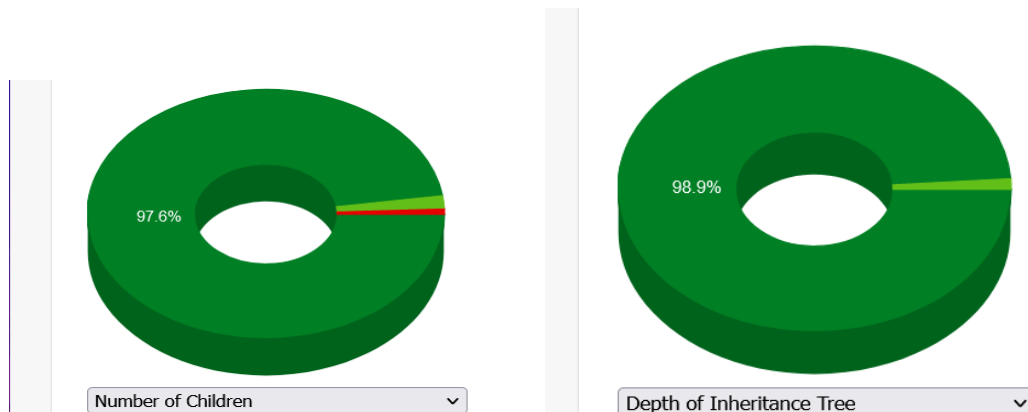
Lack of Cohesion

3.4 Organisations des classes

. Etudions la hiérarchie des classes. La hiérarchie est-elle plutôt plate (peu de niveaux de hiérarchie) ou à l'inverse profonde ? On pourra par exemple s'appuyer sur la profondeur de l'arbre d'héritage (DIT), le nombre d'enfants par classes (min, max ou moyenne) (NOC)

On constate qu'il y a plus de classes avec un DIT et un NOC faibles, cela peut indiquer une hiérarchie plus plate et plus simple, ce qui peut faciliter la compréhension et la maintenance du code. Cependant, il est important de noter que cela peut également être dû à une conception inappropriée des classes et de l'architecture logicielle.

Pour améliorer la situation, il est possible d'appliquer les principes SOLID mentionnés précédemment. En particulier, le principe de responsabilité unique (SRP) et le principe d'inversion de dépendance (DIP) peuvent aider à réduire la complexité et à simplifier la hiérarchie de classes.

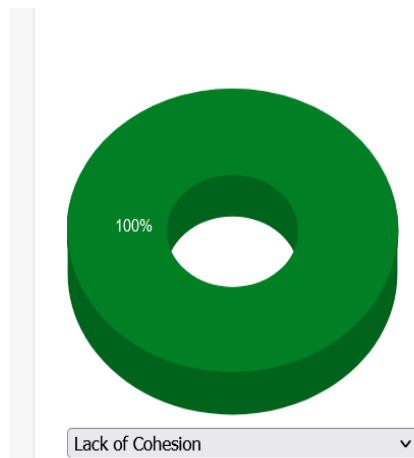


. étudier la stabilité des classes en général ou de quelques-unes en particulier. Pour cela, on pourra s'appuyer sur la notion de couplage.

Le fait que ce projet a plus de classes avec un couplage faible qu'un couplage fort peut avoir une incidence sur la stabilité des classes. En général, les classes avec un couplage faible sont plus stables que les classes avec un couplage fort, car elles sont moins dépendantes d'autres classes et donc moins susceptibles d'être affectées par des changements dans d'autres parties du code.

Pour améliorer la situation, il est recommandé de revoir la conception de chaque classe et de s'assurer que chaque classe a une responsabilité clairement définie. Ensuite, il est possible d'identifier les dépendances entre les classes et de trouver des moyens de les réduire ou de les supprimer si possible.

.étudions la cohésion des classes au sein d'un paquetage en particulier.



Toutes les classes du paquetage `com.google.gson.annotations` ont un taux de manque de cohésion faible. Le manque de cohésion faible indique que les méthodes et les champs d'une classe ne sont pas étroitement liés et qu'ils pourraient être séparés en classes distinctes avec des responsabilités plus claires et mieux définies. Cela peut entraîner une complexité accrue, une difficulté à comprendre le code et à effectuer des modifications, et une augmentation de la probabilité d'erreurs et de bogues. Dans le cas de `com.google.gson.annotations`, cela peut indiquer que le paquetage contient des classes qui ne sont pas suffisamment spécialisées et qui pourraient être séparées en classes plus petites et plus cohérentes. Une amélioration possible serait de refactoriser les classes existantes pour les rendre plus spécialisées et cohérentes. Cela peut inclure la création de sous-classes ou la réorganisation de la hiérarchie de classes existantes.

4] Analyse Approfondi

Coverage	84.8%
Lines to Cover	5,554
Uncovered Lines	846
Line Coverage	84.8%
Conditions to Cover	2,763
Uncovered Conditions	422
Condition Coverage	84.7%
Tests	
Unit Tests	1,287
Errors	0
Failures	0
Skipped	19
Success	100%
Duration	3s

4.1. Tests

. Combien de classes de tests existent? combien de méthodes de tests? Combien d'assertions? Que peut-on en déduire sur la structuration des tests?

Il existe 128 classes de tests et 1282 méthodes de tests. Le grand nombre de méthodes de test indique que le projet possède une bonne couverture de tests, ce qui est un indicateur positif de la qualité du code. Cela suggère également que les tests sont bien structurés et organisés en classes dédiées à des fonctionnalités spécifiques. Cela facilite la maintenance et la compréhension du code de test. Une amélioration possible pour cela serait d'utiliser des techniques de test automatisé pour exécuter les tests à chaque fois que le code est modifié. Cela garantit que les tests sont régulièrement exécutés et que les erreurs sont détectées rapidement.

▼ Scope	
📄 Main code	335
📄 Test code	128

. étudions la couverture de tests. Quel est le pourcentage de code couvert par les tests ?

Le pourcentage de couverture de code est de 84,8% mais il y a de nombreuses parties non couvertes. Cela peut être considéré comme un bon pourcentage, car cela signifie que la majorité du code a été couverte par des tests unitaires. Cependant, cela ne garantit pas que toutes les fonctionnalités ont été testées de manière exhaustive ou que tous les bugs ont été détectés. Une amélioration possible serait d'ajouter des tests pour les zones

non couvertes en utilisant des techniques telles que les tests de bordure, les tests d'intégration et les tests de performance. Cela peut aider à améliorer la qualité du code et à détecter les erreurs potentielles.

. étudions le type de tests. S'agit-il de tests unitaires, fonctionnels. . . ?

Ce projet comporte une variété de type de tests notamment `CommentsTest.java`(test unitaire) du package `gson.src.test.java.com.google.gson`, `DefaultInetAddressTypeAdapterTest.java`(test fonctionnel) du package `gson.src.test.java.com.google.gson` et `EnumWithObfuscatedTest.java`(test de non regression) du package `gson.src.test.java.com.google.gson.fonctionnal`. Les tests unitaires tels que "`CommentsTest`" sont importants pour s'assurer que des parties spécifiques du code fonctionnent correctement de manière isolée, tandis que les tests fonctionnels tels que "`DefaultInetAddressTypeAdapterTest`" testent le comportement du code dans son ensemble. Les tests de non-régression, comme "`EnumWithObfuscatedTest`", s'assurent que des changements ultérieurs dans le code n'ont pas affecté négativement les fonctionnalités existantes. Avoir une combinaison de ces types de tests permet de s'assurer que le projet fonctionne correctement dans son ensemble et qu'il est résistant aux changements futurs. Une amélioration possible serait d'augmenter la couverture de tests : pour que chaque fonctionnalité et chaque branche du code soient couvertes par des tests. Il est important de s'assurer que toutes les parties du code sont testées, pour minimiser les risques d'erreurs et de bugs. Comme amélioration on peut améliorer la couverture de tests dans ce projet. Identifier les parties du code qui ne sont pas encore couvertes par des tests : Pour augmenter la couverture de tests, il est important de savoir quelles parties du code ne sont pas encore testées. Cela peut être réalisé en utilisant des outils de couverture de code, qui vous aideront à identifier les zones qui ne sont pas couvertes par des tests.

. étudions si les tests passent tous.

L'image ci-dessus nous montre que malgré qu'il y a beaucoup de tests passés il y en a 19 qui ont été skipped". Cela peut indiquer que ces tests ne sont pas implémentés ou que leur implémentation est en cours de développement. Cela peut également indiquer que ces tests sont volontairement ignorés car

ils sont considérés comme moins importants ou qu'ils posent des problèmes de fiabilité. Pour améliorer la situation des tests "skipped", on peut: Examiner les raisons pour lesquelles les tests ont été ignorés : Avant de prendre toute mesure, il est important de comprendre pourquoi les tests ont été ignorés. Il est possible que certains tests n'aient pas encore été implémentés ou que leur implémentation soit en cours. D'autres tests peuvent être ignorés car ils sont considérés comme moins importants ou qu'ils posent des problèmes de fiabilité. Il est important de bien comprendre ces raisons pour pouvoir agir en conséquence.

4.2. Commentaires

. Comptons le nombre de lignes de commentaire

A l'aide de Sonarqube, on trouve qu'il y a 4372 commentaires, avec un pourcentage de 28.1% de code commenté. On ne peut pas nier que le nombre donne l'impression que le code est bien documenté, mais une analyse plus approfondie pourrait être nécessaire pour déterminer si les commentaires sont pertinents et utiles pour la compréhension du code. Pour améliorer la qualité des commentaires dans le code, voici quelques suggestions : Éviter les commentaires inutiles : il est important de ne pas commenter l'évident. Commenter les parties complexes : les parties complexes du code doivent être commentées pour faciliter leur compréhension.

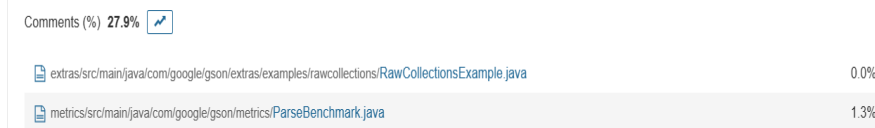
Size	
Lines of Code	11,171
Lines	20,521
Statements	4,803
Functions	832
Classes	139
Files	101
Comment Lines	4,372
Comments (%)	28.1%

. Identifions le type de commentaire: Javadoc, code commenté, licence, commentaire pertinent

On retrouve les différents types de commentaires dans ce projet dans diverses classes, par exemple la classe `ExclusionStrategy.java` du package `gson.src.main.java.com.google.gson` qui contient des commentaires de type licence, javadoc, code commenté et commentaire pertinent. La présence de différents types de

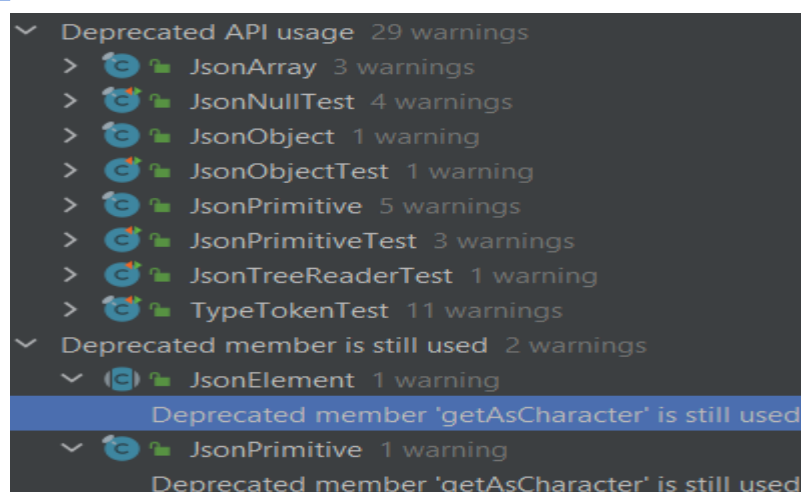
commentaires montre que les développeurs ont utilisé différentes stratégies pour documenter leur code. Cela peut aider à faciliter la compréhension du code pour d'autres développeurs et contribuer à une meilleure maintenabilité du projet à long terme. Comme amélioration on peut demander des revues de code à d'autres développeurs pour obtenir des commentaires et des suggestions sur la qualité des commentaires dans le projet.

. Existent-t-ils des parties sans commentaires?



Il y a de nombreuses parties du projet en outre les méthodes et les attributs qui ne sont pas commentés notamment une classe entièrement non commentée (comme le montre le pourcentage ci-dessus). Un projet ne nécessite pas une documentation totale, à l'aide des noms d'attributs et des méthodes explicites, ce n'est pas grave d'avoir des bouts de codes non commentés, au contraire, ce fera un gaspillage d'espace si on commente chaque méthode même si elle respecte les règles de nommage. Pourtant, ne pas commenter une classe entièrement peut rendre la compréhension du code plus difficile pour les développeurs qui n'ont pas participé à l'écriture du code. Cela peut entraîner une augmentation du temps nécessaire pour comprendre le code existant, ainsi que des erreurs de compréhension et des erreurs de maintenance.

4.3 Dépréciation



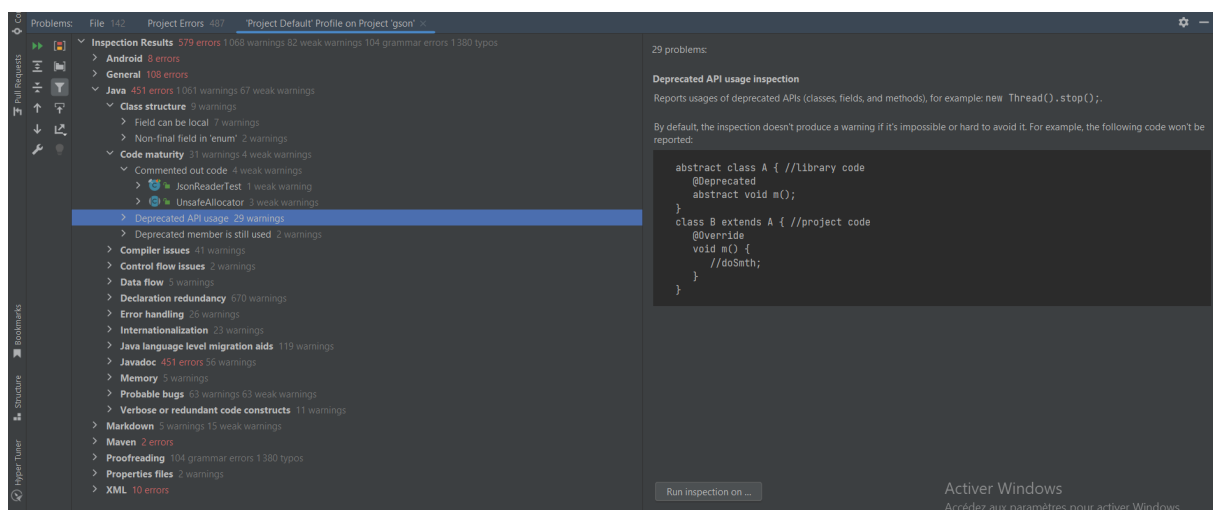
. Identifions les bouts de code dépréciés (classes, méthodes)

Il y a 8 classes utilisant des méthodes ou est elles mêmes sont dépréciées dans le projet, cela peut signifier que certaines classes y compris les méthodes ne sont plus recommandées pour une utilisation future et peuvent être retirées du code.

. Identifions les appels à du code déprécié. Du code non déprécié appelle-t-il du code déprécié? Si oui quelle en est la conséquence

getAsCharacter() est une méthode dépréciée du package gson.com.google.gson.JsonElement.: implémentée et utilisée uniquement dans JsonElement et ses sous classes JsonArray et JsonPrimitive. L'appel à cette méthode peut causer des problèmes de compatibilité avec les nouvelles versions de la bibliothèque et peut également entraîner des erreurs ou des comportements inattendus.

Bien que des méthodes, on trouve aussi des API dépréciés (comme sur la capture ci-dessous), on peut aussi faire des améliorations sur ces dépréciations pour la deuxième partie du projet.



4.4 Duplication de code

A l'aide de sonarQube, on peut constater qu'on a pas mal de blocs de code dupliqués, sur l'exemple qui suit, la classe TypeAdapters.java du package "gson.src.main.java.com.google.gson.internal.bind" a 24 lignes et deux blocs dupliqués, ce qui augmente la complexité du projet inutilement parfois, sur l'image par exemple la fonction

`read(JsonReader in)` est écrite deux fois avec une différence de `cast (float)`.

Il est possible de supprimer la deuxième fonction et faire un `cast` pendant l'utilisation de la première version de la fonction.

(la duplication du code fera partie du refactoring de la deuxième partie de notre projet, elle sera plus traitée par)

Category	Value
Overview	
Overall	
Density	2.3%
Duplicated Lines	24
Duplicated Blocks	2
Duplicated Files	1

```
348 public Number read(JsonReader in) throws IOException {
349     if (in.peek() == JsonToken.NULL) {
350         in.nextNull();
351         return null;
352     }
353     return (float) in.nextDouble();
354 }
355
356 @Override
357 public void write(JsonWriter out, Number value) throws IOException {
358     if (value == null) {
359         out.nullValue();
360     } else {
361         // For backward compatibility don't call `JsonWriter.value(float)` because that method has
362         // been newly added and not all custom JsonWriter implementations might override it yet
363         Number floatNumber = value instanceof Float ? value : value.floatValue();
364         out.value(floatNumber);
365     }
366 }
367
368 public static final TypeAdapter<Number> DOUBLE = new TypeAdapter<Number>() {
369     @Override
370     public Number read(JsonReader in) throws IOException {
371         if (in.peek() == JsonToken.NULL) {
372             in.nextNull();
373             return null;
374         }
375         return in.nextDouble();
376     }
377 }
```

4.5 God Classes

. compter le nombre de méthodes par classe (min, max, moyenne, médiane).

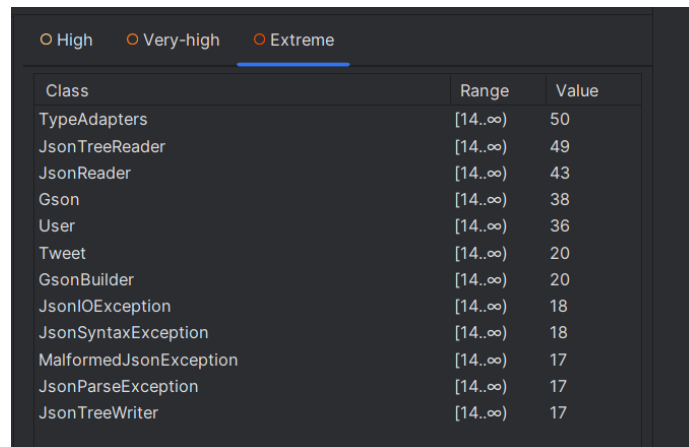
Class	Number of Methods
Gson	41 [+35]
JsonReader	40 [+34]
JsonWriter	37 [+31]
JsonArray	33 [+27]
GsonBuilder	32 [+26]
JsonTreeReader	28 [+22]

Avec un minimum de 2 méthodes comme la classe `rawcollectionsExample` du package `com.google.extras.examples.rawcollections` et un maximum de 41 méthodes de la classe `gson` du package `gson.src.main.java.com.google.gson`. Comme analyse on peut dire que la classe `rawcollectionsExample` du package `com.google.extras.examples.rawcollections` a un nombre minimal de méthodes (2), ce qui suggère qu'elle est

probablement utilisée pour des fonctions de base ou de démonstration. En revanche, la classe gson du package `gson.src.main.java.com.google.gson` a un nombre maximal de méthodes (41), ce qui suggère qu'elle est probablement utilisée pour des fonctions plus complexes et étendues. Cependant, sur la base de ces deux exemples extrêmes, il est probable que le nombre de méthodes varie considérablement en fonction de la complexité de la classe et de son utilisation. Ainsi comme amélioration possible on peut diviser les classes en sous-classes plus petites et plus spécialisées et utiliser l'héritage comme moyen de réutiliser le code commun et de partager des fonctionnalités entre les classes.

. compter le nombre de variables d'instances par classe (min, max, moyenne, médiane).

Comparaison avec les résultats trouvés à la question précédente ?



Class	Range	Value
TypeAdapters	[14..∞)	50
JsonTreeReader	[14..∞)	49
JsonReader	[14..∞)	43
Gson	[14..∞)	38
User	[14..∞)	36
Tweet	[14..∞)	20
GsonBuilder	[14..∞)	20
JsonIOException	[14..∞)	18
JsonSyntaxException	[14..∞)	18
MalformedJsonException	[14..∞)	17
JsonParseException	[14..∞)	17
JsonTreeWriter	[14..∞)	17

Avec un minimum de 2 variables d'instances comme la classe `rawcollectionsExample` du package `com.google.extras.examples.rawcollections` et un maximum de 50 variables d'instances de la classe `TypeAdapters` du package `gson.src.main.java.com.google.gson.internal.bind`. En comparant les résultats de la question précédente (nombre de méthodes par classe) avec les résultats de cette question (nombre de variables d'instances par classe), nous pourrions tirer des conclusions sur la complexité des classes dans le code analysé. Si une classe a un nombre élevé de méthodes et un nombre élevé de variables d'instances, cela peut indiquer une classe complexe qui a beaucoup de responsabilités et de fonctionnalités, ce qui peut rendre la maintenance et la compréhension du code plus difficiles. D'un autre côté, si une classe a un nombre faible de méthodes et un nombre faible de variables d'instances, cela peut indiquer une classe simple avec une responsabilité limitée. Comme

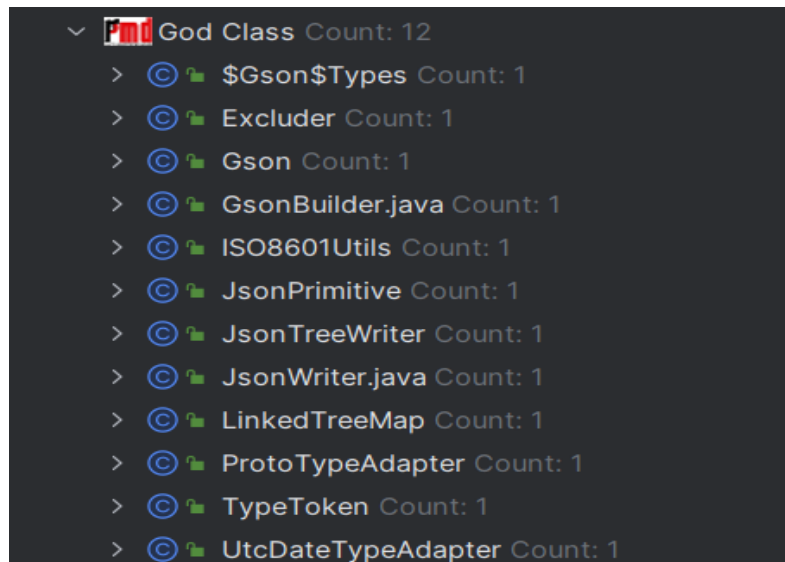
amélioration, dans le cas où une classe a un nombre élevé de variables d'instances, il peut être judicieux d'examiner si certaines de ces variables peuvent être regroupées en sous-classes ou en structures de données pour réduire la complexité de la classe principale.

.compter le nombre de lignes de code par classe (min, max, moyenne, médiane),

Comparaison avec les résultats trouvés à la question précédente ?

Method metrics				Class metrics				Interface metrics				Package metrics				Module metrics				File type metrics				Project metrics			
class																											

. identifier les gods classes. Pour les grosses classes, identifier si elles utilisent plus des choses en interne ou référencent de nombreuses classes



Nous avons remarqué qu'il y a 12 God classes au total dans le projet, ce qui est un grand nombre pour notre projet. Il y a donc trop de longues classes dans ce projet, chacune d'elles ayant trop de responsabilités et inévitablement trop de méthodes, la plupart étant incompréhensibles car trop longues. Aussi, bien que le nom de chacune des classes peut laisser deviner la fonctionnalité principale, en analysant plus en détails nous constatons quand même qu'il reste délicat de déterminer l'objectif principal.

Pour améliorer cela, il est important de répartir les responsabilités de la classe entre plusieurs classes plus petites et spécialisées, en utilisant le principe de responsabilité unique (SRP) pour que chaque classe ait une seule responsabilité. Cela aidera à réduire la complexité cyclomatique et à minimiser la dépendance sur d'autres classes.

4.6 Analyse des méthodes

. Calculer la complexité cyclomatique des méthodes en général ou de quelques unes en particulier (avec min, max, moyenne, médiane).












Method metrics				
Class metrics				
Package metrics				
Module metrics				
Project metrics				
method	CogC	ev(G)	iv(G)	v(G)
com.google.gson.stream.JsonReader.nextNonWhitespace(boolean)	30	12	8	16
com.google.gson.functional.ParameterizedTypesTest.MultiParameters.equals(Object)	21	17	6	17
com.google.gson.internal.bind.JsonElementReaderTest.testWrongType()	16	1	1	17
com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.createBoundField(Gson, F	31	8	13	17
com.google.gson.stream.JsonReader.skipValue()	17	3	3	19
com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.getBoundFields(Gson, Typ	51	11	12	20
com.google.gson.stream.JsonReader.readEscapeCharacter()	17	13	5	20
com.google.gson.typeadapters.UtcDateTypeAdapter.parse(String, ParsePosition)	27	5	6	20
com.google.gson.internal.LinkedTreeMap.rebalance(Node<K, V>, boolean)	47	9	7	22
com.google.gson.internal.\$Gson\$Types.resolve(Type, Class<?>, Type, Map<TypeVariab	62	13	19	23
com.google.gson.internal.bind.util.ISO8601Utils.parse(String, ParsePosition)	53	8	15	32
com.google.gson.stream.JsonReader.peekNumber()	57	21	4	37
com.google.gson.stream.JsonReader.doPeek()	47	26	20	40
Total	2793	3349	3665	4706
Average	1,06	1,29	1,42	1,82

Le minimum de la complexité cyclomatique des méthodes est de 0, son maximum quant à lui est de 40, cela nous donne une moyenne de 1,82. Ce qui suggère que le code est relativement simple et facile à comprendre. Cependant, il convient de noter que le maximum de la complexité cyclomatique est de 40, ce qui est assez élevé et pourrait indiquer la présence de méthodes complexes qui pourraient être simplifiées et divisées comme amélioration à faire.

. Analysons les commentaires. Les commentaires sont-ils bien placés (les méthodes avec une plus grande complexité cyclomatique ont elles également le plus grand nombre de lignes de commentaire).

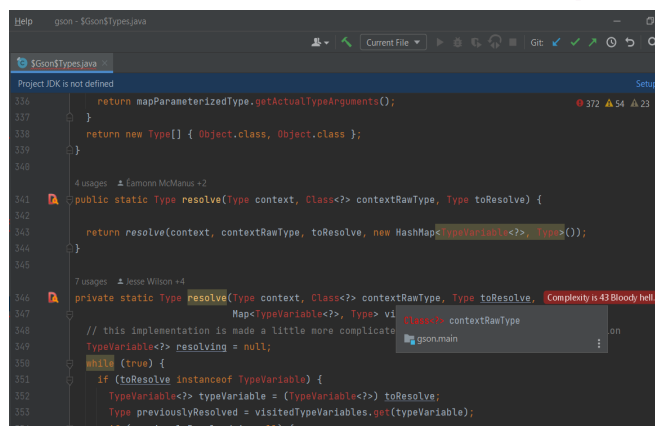
La méthode `doPeek` de la classe `com.google.gson.stream.JsonReader` étant la méthode ayant le plus grand nombre en complexité comporte 7 commentaires. Contrairement à la méthode `write` de la classe `com.google.gson.internal.bind.JsonTreeWriter` de complexité 1 qui ne possède qu'un commentaire. Il est difficile de dire si les commentaires sont bien placés ou pas seulement en se basant sur le nombre de commentaires d'une méthode et sa complexité cyclomatique. En effet, la complexité cyclomatique peut donner une indication sur la complexité d'une méthode, mais cela ne prend pas en compte d'autres facteurs tels que la longueur de la méthode, sa clarté et sa lisibilité. En amélioration on peut commenter le code de manière pertinente et les parties complexes.

. Calculons le nombre de lignes de codes des méthodes (avec min, max, moyenne,médiane)

Method metrics	Class metrics	Interface metrics	Package metrics	Module metrics	File type metrics	Project metrics	
method					CLOC	JLOC ▲	LOC
 com.google.gson.stream.JsonReader.skipValue()					22	15	85
 com.google.gson.internal.\$Gson\$Types.resolve(Type, Class<?>, Type, Map<TypeVariable<?>, Type>)					4	0	86
 com.google.gson.stream.JsonReader.nextNonWhitespace(boolean)					23	6	89
 com.google.gson.internal.bind.JsonElementReader.testWrongType()					0	0	90
 com.google.gson.Gson.Gson.Excluder, FieldNamingStrategy, Map<Type, InstanceCreator>					6	0	91
 com.google.gson.Gson.GsonTest.testGetAdapter_FutureAdapterConcurrency()					22	12	92
 com.google.gson.stream.JsonReader.JsonReader.peekNumber()					8	0	95
 com.google.gson.typeadapters.UtcDateTypeAdapter.parse(String, ParsePosition)					16	9	95
 com.google.gson.graph.GraphAdapterBuilderFactory.create(Gson, TypeToken<T>)					23	0	107
 com.google.gson.stream.JsonReader.doPeek()					9	0	131
 com.google.gson.internal.bind.util.ISO8601Utils.parse(String, ParsePosition)					34	9	147
Total					3 612	3 424	29 273
Average					1.37	1.32	11.31

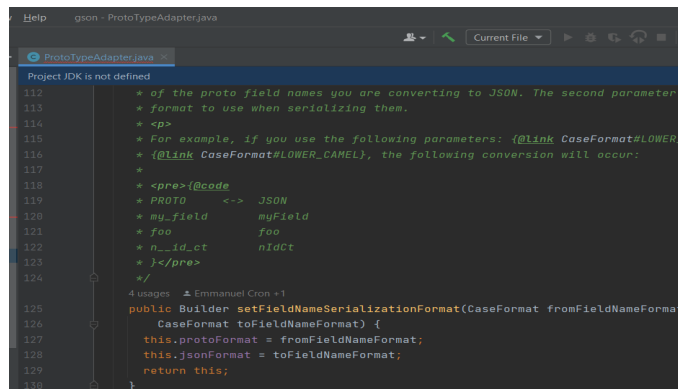
Nous observons un minimum de 0 lignes de codes des méthodes avec un maximum de 147 pour une moyenne de 11,31 lignes de codes par méthode. Cela montre que les méthodes ont une taille moyenne assez faible, ce qui est généralement considéré comme une bonne pratique de programmation pour faciliter la lisibilité et la maintenabilité du code. Cependant, le fait que le maximum soit de 147 lignes de code montre qu'il y a quand même des méthodes très longues qui peuvent être difficiles à comprendre et à maintenir.

. Identifions les méthodes avec beaucoup d'arguments



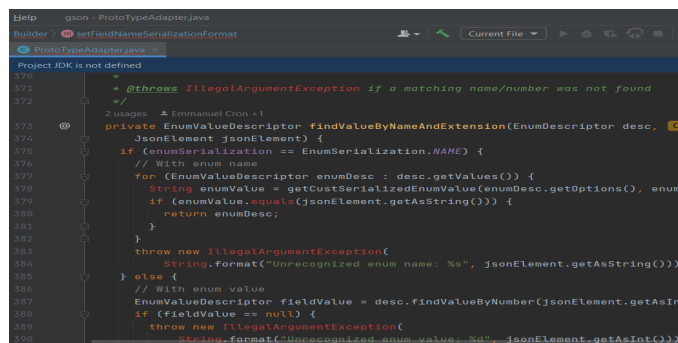
La méthode `resolve()` de la classe `$Gson$Types.java` compte 5 arguments. Cela peut indiquer une complexité plus importante, car plus il y a d'arguments, plus il peut être difficile de comprendre leur signification et leur utilisation. Cela peut également rendre la méthode plus difficile à tester et à maintenir. Si possible, réduire le nombre d'arguments peut aider à simplifier la méthode comme amélioration lorsqu'on se retrouve dans ce genre de cas.

. Identifions des méthodes qui à la fois modifie l'état d'un objet et retourne des informations concernant cet objet



La méthode `setFieldNameSerializationFormat` de la classe `ProtoTypeAdapter.java` est une méthode qui à la fois modifie l'état d'un objet et retourne des informations concernant cet objet. Le fait qu'une méthode modifie l'état d'un objet et retourne des informations concernant cet objet peut rendre le code plus complexe et difficile à comprendre, car il est souvent plus facile de comprendre une méthode qui ne fait qu'une seule chose. Cependant, dans certains cas, il peut être justifié d'avoir une méthode qui combine ces deux fonctionnalités. Séparer la méthode en 2 méthodes pour que chaque méthode a son utilité précise on peut faire comme amélioration.

. Identifions des méthodes qui retournent un code d'erreur



La méthode `findValueByNameAndExtension` de la classe `prot.src.main.java.com.google.gson.protobuf.prototypeadapater.java` est une des méthodes du projet qui retourne un code d'erreur. Cela peut aider à détecter rapidement les erreurs et à corriger le problème. Il serait également utile d'avoir des messages d'erreur clairs et explicites, pour aider à comprendre la source du problème et faciliter la correction en vue d'une amélioration possible.

5] Nettoyage de Code et Code smells

5.1. Règles de nommage

. Les noms utilisés dans le programme pour les différents éléments sont-ils descriptifs et sans ambiguïté?

Après la vérification de plusieurs classes, on peut dire que la plupart des noms des différents attributs ou méthodes correspondent bien à la description de ce qu'ils font, comme l'attribut `stack` suivant qui représente une liste de `int` ou une pile plutôt qui sert à stocker des données `int`. Le stockage ou la suppression des données sont traités exactement comme en pile, c'est pour ça qu'on l'a nommé `stack`.

```
175  /** The JSON output destination */
176  private final Writer out;
177
178  private int[] stack = new int[32];
179  private int stackSize = 0;
```

Aussi, quand le nom apparaît un peu ambigu, on ajoute la documentation dessus, ce qui explique bien son rôle. c'est le cas pour la fonction `replaceTop` qui s'occupe de remplacer la tête de la pile avec une valeur donnée.

```
105  /**
106   * Replace the value on the top of the stack with the given value.
107   */
108  private void replaceTop(int topOfStack) {
109      stack[stackSize - 1] = topOfStack;
```

Mais on peut pas dire que tous les noms sont bien descriptifs, on trouve souvent des noms qu'il faut chercher dans les méthodes, il manque un peu de documentation.

```
196  private boolean htmlSafe;
```

Pour cet attribut par exemple, on réalise pas directement la relation de `html` et `safe`, mais en regardant les fonctions on trouve que c'est fait pour vérifier si un changement vers `Json` n'est pas dangereux pour une inclusion directe en `html`.

```
280  /**
281   * Configure this writer to emit JSON that's safe for direct inclusion in HTML
282   * and XML documents. This escapes the HTML characters {@code <}, {@code >},
283   * {@code &} and {@code =} before writing them to the stream. Without this
284   * setting, your XML/HTML encoder should replace these characters with the
285   * corresponding escape sequences.
286   */
287  public final void setHtmlSafe(boolean htmlSafe) {
288      this.htmlSafe = htmlSafe;
289  }
```

. Les noms participent-ils à la désinformation en particulier pour les structures de données?

On trouve que l'ensemble de mots choisis n'affectent pas le but des méthodes ou les rôles des attributs, comme dans l'exemple qui suit, "deserialize" revient à sa description qui est écrite dessus, ça renvoie bien un boolean selon si le bout de code est sérialisé ou pas.

```
74  /**
75   * If {@code true}, the field marked with this annotation is deserialized from the JSON.
76   * If {@code false}, the field marked with this annotation is skipped during deserialization.
77   * Defaults to {@code true}.
78   * @since 1.4
79   */
80   public boolean deserialize() default true;
```

. Les noms sont prononçables et sont-ils des termes du domaine?

Oui, les noms utilisés ont tous une relation avec le sujet et le domaine en général, comme l'exemple de l'attribut boolean `htmlSafe` donné précédemment.

Grosso modo, on constate qu'un pourcentage important du code source respecte les critères de nommage mais il manque certainement quelques parties qui peuvent être améliorées.

5.2. Nombre magique

. Existent-ils des nombres magiques dans le code?

Oui, à l'aide de IntelliJ on trouve qu'on a 77 nombres magiques dans la totalité du projet, notamment sur la classe "\$Gson\$Types.java" du package "[gson.gson.src.main.java.com.google.gson.internal](https://github.com/google/gson/blob/master/gson/src/main/java/com/google/gson/internal/$Gson$Types.java)", on trouve le nombre magique 30 (sur l'exemple) et 2 autres nombres.

The screenshot shows the IntelliJ IDEA interface. The top editor displays the source code of `$Gson$Types.java`, with line 542 highlighted: `StringBuilder stringBuilder = new StringBuilder(capacity: 30 * (length + 1));`. Below the editor, the 'QAPlug results' window is open, showing a list of quality rules and their counts. The 'Magic Number' rule is expanded, showing a count of 77. Underneath, the '\$Gson\$Types' class is listed with a count of 3. A tooltip for the '30' magic number is visible, stating: "'30' devrait être défini comme une constante." (The number 30 should be defined as a constant).

Rule	Count
Close Resource	19
Compare Objects With Equals	13
Design For Extension	31
Finalize Overloaded	3
Magic Number	77
\$Gson\$Types	3
BagOfPrimitives	3
BagOfPrimitivesDeserializationBenchmark	1

. Est-il possible de comprendre pourquoi telle valeur ou pas telle autre, en essayant de comprendre l'intention?

Il est difficile de comprendre ces valeurs parce qu'elles ne sont pas documentées et ne sont pas intuitives, il nous faut une bonne connaissance du contexte pour savoir par exemple à quoi le nombre entier 30 (dans l'image ci-dessus). On ne peut que projeter notre intuition pour essayer de les comprendre.

. Peut on avoir une proposition pour remplacer l'utilisation de ce nombre magique (dans le code et éventuellement dans les tests) ?

Il est fortement conseillé de changer les nombres magiques par des constantes, ou bien donner des noms plus explicites aux variables, ce qui n'est pas toujours évident, les constantes rendent le code plus facile à maintenir et mettre à jour car on peut directement changer la valeur en déclaration sans avoir à rechercher toutes les occurrences dans le code.

(le remplacement des nombres magiques en des constantes sera parmi les changements qu'on fera en deuxième partie).

5.3. Structure de code

. Vérifions qu'au sein des classes, est-ce que toutes les variables d'instance sont ensemble?

On ne respecte pas souvent la règle de déclaration des variables d'instances en même endroit, on trouve pas mal d'attributs mis juste après la déclaration de la classe et avant la déclaration du constructeur, mais on trouve aussi des variables déclarées au milieu de la classe entre les méthodes, comme sur la deuxième image.

```
21  * gautnor inderjeet singh
22  */
23  public class BagOfPrimitives {
24      public static final long DEFAULT_VALUE = 0;
25      public long longValue;
26      public int intValue;
27      public boolean booleanValue;
28      public String stringValue;
29
30      public BagOfPrimitives() {
31          this(DEFAULT_VALUE, 0, false, "");
32      }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85  */
86  public static Builder newBuilder() {
87      return new Builder(EnumSerialization.NAME, CaseFormat.LOWER_UNDERSCORE, CaseFormat.LOWER_CAMEL);
88  }
89
90  private static final com.google.protobuf.Descriptors.FieldDescriptor.Type ENUM_TYPE =
91      com.google.protobuf.Descriptors.FieldDescriptor.Type.ENUM;
92
93  private static final ConcurrentMap<String, ConcurrentMap<Class<?>, Method>> mapOfMapOfMethods =
94      new MapMaker().makeMap();
95
```

. Est-ce que les méthodes publiques et les plus utilisées sont en haut dans la définition des classes et les méthodes privées plus bas?

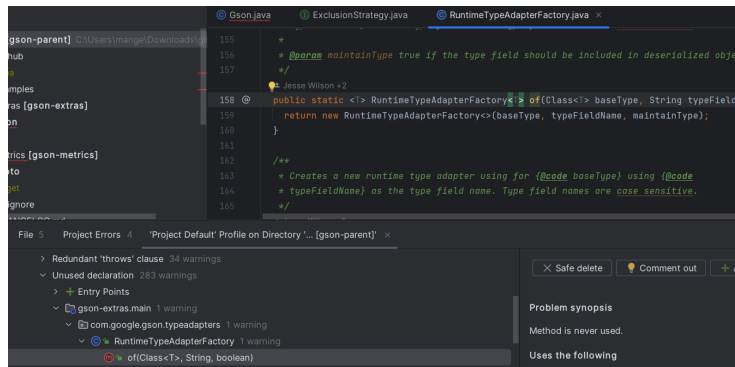
Non, ce critère n'est pas respecté, on trouve souvent des méthodes privées déclarées juste avant d'être utilisées, selon le besoin. Comme dans la capture qui suit, une méthode privée est faite entre deux autres méthodes Override.

```
44  @Override public int size() {
45      return delegate.size();
46  }
47
48  private E nonNull(E element) {
49      if (element == null) {
50          throw new NullPointerException("Element must be non-null");
51      }
52      return element;
53  }
54
55  @Override public E set(int index, E element) {
56      return delegate.set(index, nonNull(element));
57  }
58
```

5.4. Code mort

. Existe-t-il du code mort au sein du projet (rappel : code mort code pas appelé)?

Oui il existe du code mort, nous relevons 283 warnings à propos de déclaration non utilisées. Par exemple on a la méthode public static <T> RuntimeTypeAdapterFactory<T> of(Class<T> baseType, String typeFieldName, boolean maintainType) de la classe RuntimeTypeAdapterFactory du package extras.src.main.java.google.gson.typeadapters qui n'est jamais utilisée. Cela peut être un signe que cette méthode n'est plus nécessaire ou qu'elle a été remplacée par une autre méthode. Comme amélioration possible on peut écrire des tests automatisés ou utiliser des outils de détection de code mort pour les supprimer par exemple.



Ce code est-il testé ? maintenu ?

Pour le cas de la méthode public static <T>

RuntimeTypeAdapterFactory<T> of(Class<T> baseType, String typeFieldName, boolean maintainType) de la classe

RuntimeTypeAdapterFactory du package

extras.src.main.java.google.gson.typeadapters, grâce à sonarqube on a constaté que cette méthode n'est pas testée. Cela signifie qu'elle n'est pas couverte par des tests unitaires. Cela peut indiquer un risque pour la qualité du code, car si la méthode n'est pas testée, il est difficile de savoir si elle fonctionne correctement ou si elle peut causer des problèmes dans le code. Pour remédier à ce problème, on peut créer des tests unitaires pour cette méthode.

```

/**
 * Creates a new runtime type adapter using for {@code baseType} using {@code
 * typeFieldName} as the type field name. Type field names are case sensitive.
 *
 * @param maintainType true if the type field should be included in deserialized objects
 */
public static <T> RuntimeTypeAdapterFactory<T> of(Class<T> baseType, String typeFieldName, boolean maintainType) {
    return new RuntimeTypeAdapterFactory<>(baseType, typeFieldName, maintainType);
}

```

Ce code devrait-il être supprimé ?

On doit d'abord examiner attentivement la méthode pour déterminer si elle est effectivement inutile et si sa suppression peut améliorer la qualité du code. Si la méthode est effectivement inutile, elle peut être supprimée en s'assurant qu'elle n'a pas d'effets secondaires sur le reste du code.

Pour résumer, après l'analyse de cette partie de nettoyage de code, on peut dire qu'on a pas mal de parties de code qui nécessite du refactoring, du code qui peut être supprimé, étant mort ou redondant, des attributs qui sont mal placés ou mal

décrits... Aussi, en regardant toutes les autres parties, on cite les modifications majeures qu'on prévoit faire pour la deuxième partie du projet:

- **La duplication du code(sera fait essentiellement par Yannick)**
- **Les nombres magiques qui peuvent être remplacés par des constantes pour la clarté du code.(sera fait par Nouha)**
- **Le nommage des attributs/méthodes , l'utilisation des noms plus explicites(sera ajouté par Nouha)**
- **La documentation des méthodes qui ne semblent pas claires(sera vue par Yannick)**
- **la suppression des méthodes dépréciées et même des API dépréciées.(sera vue par Yannick)**
- **réduire la redondance du code et la suppression des méthodes dupliqués(sera fait principalement par Nouha)**