



GÉNIE LOGICIEL

Projet - Génie Logiciel

Proposition
d'amélioration du projet
GSON
GROUPE 1 GL

PRÉSENTÉ PAR: MOUNGUENGUI YANNICK

Sommaire

I. Petites modifications

II. Moyennes
modifications

III. Grandes
modifications

IV. Conclusion

Lien gitlab du projet:

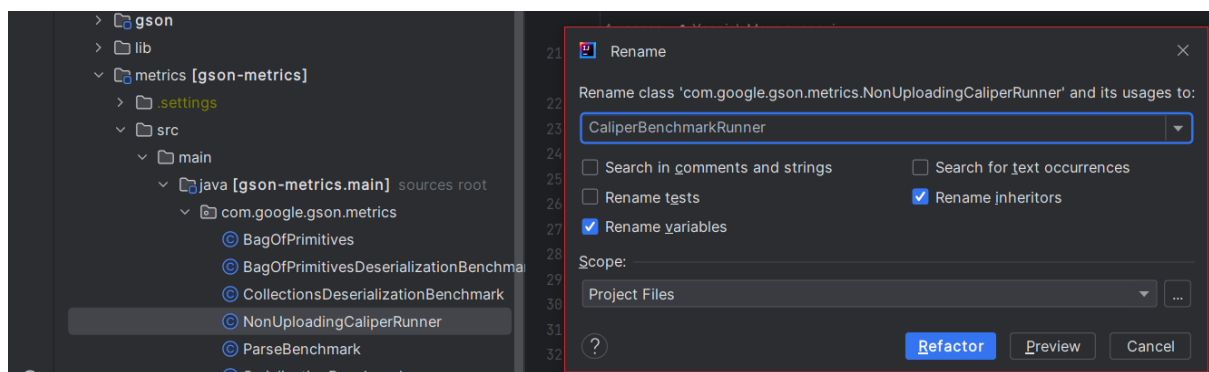
[projet_partie2_gl · main · Yannick Mounguengui / Mounguengui_Yannick_GL · GitLab \(univ-lille.fr\)](#)

I. Petites modifications

- renommer une classe, une méthode, une variable

1) Une classe

[renommage de la classe NonUploadingCaliperRunner du paquetage... \(77f2043d\) · Commits · Yannick Mounguengui / Mounguengui_Yannick_GL · GitLab \(univ-lille.fr\)](#)



Ici nous avons analysé la classe NonUploadingCaliperRunner du paquetage com.google.gson.metrics du sous dossier metrics. Nous avons constaté que cette classe utilitaire qui fournit une méthode pour exécuter des benchmarks de performances à l'aide de Caliper, un framework de benchmarking pour Java. La méthode run() prend une classe en paramètre qui définit les benchmarks à exécuter,

ainsi qu'un tableau d'arguments qui sont utilisés pour personnaliser l'exécution des benchmarks. Ainsi il a été plus raisonnable de renommer cette classe en CaliperBenchmarkRunner qui représente ce que fait la classe que NonUploadingCaliperRunner moins représentatif.

2) Une méthode

[renommage des methodes add de JSONArray en des noms differents \(55fd0b6f\)](#) ·

[Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

Dans la classe JSONArray du paquetage com.google.gson du sous dossier gson, nous remarquons qu'il y a énormément de méthodes add de paramètres différents. Le problème est le fait que le surnombre de méthodes ayant le même nom add peut porter à confusion les personnes lisant le code. Ainsi il est judicieux de renommer chacune des méthodes en un nom précis en fonction de leur utilité, nous avons alors:

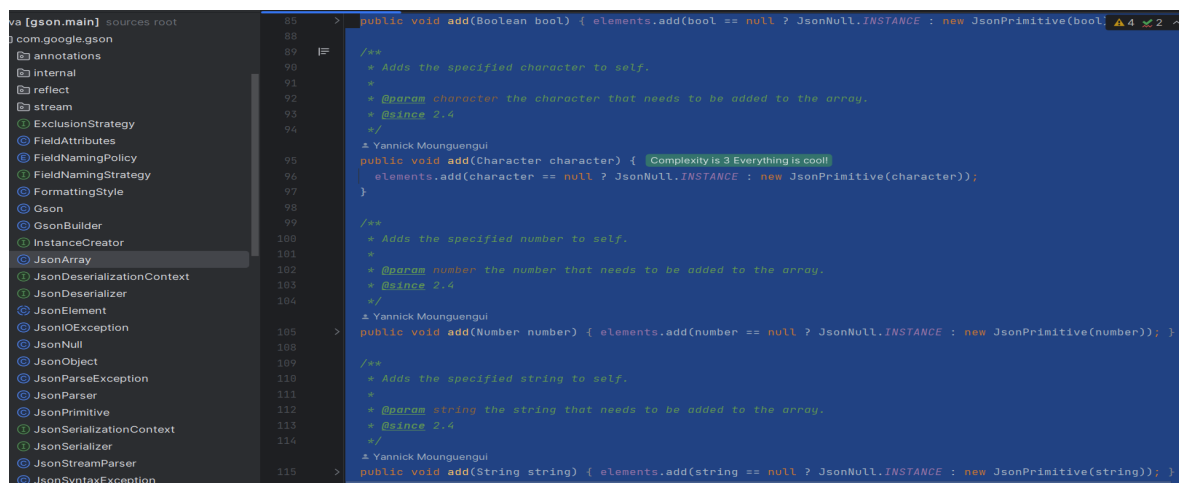
-addBoolean(Boolean bool) {...}

-addCharacter(Character character) {...}

-addNumber(Number number) {...}

- addString(String string) {...}

- addElement(JsonElement element) {...}



```
85 > public void add(Boolean bool) { elements.add(bool == null ? JsonNull.INSTANCE : new JsonPrimitive(bool)); }
86
87 /**
88  * Adds the specified character to self.
89  *
90  * @param character the character that needs to be added to the array.
91  * @since 2.4
92  */
93
94
95
96
97
98
99
100
101
102
103
104
105 > public void add(Character character) { elements.add(character == null ? JsonNull.INSTANCE : new JsonPrimitive(character)); }
106
107 /**
108  * Adds the specified number to self.
109  *
110  * @param number the number that needs to be added to the array.
111  * @since 2.4
112  */
113
114
115
116
117
118
119
120
121
122
123
124
125 > public void add(String string) { elements.add(string == null ? JsonNull.INSTANCE : new JsonPrimitive(string)); }
```

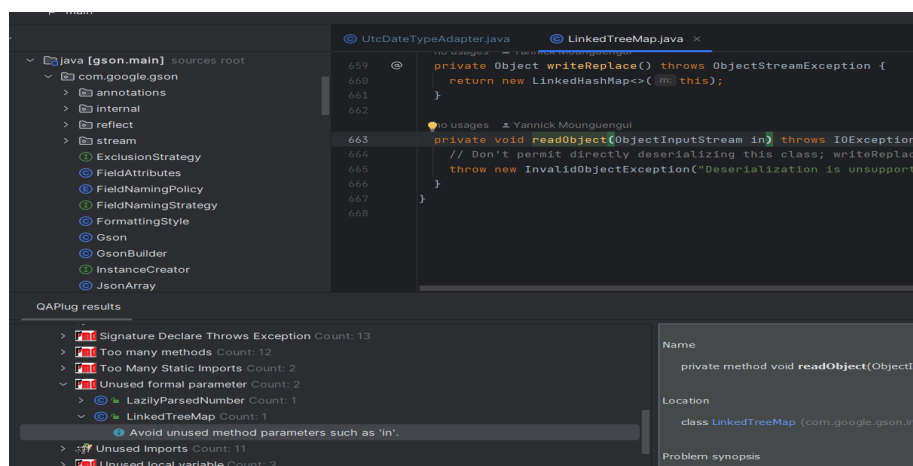
3) Une variable



Tous les noms des variables définies représentent parfaitement leur rôle dans le code de chaque classe du Projet. Par exemple ceux de la classe `JsonScope` du package `com.google.gson.stream` du dossier `gson` ont une documentation explicites en plus par rapport au nommage des variables.

- changer le type ou le nombre de paramètres d'une méthode

[modification du nombre de parametre de la methode readObject \(c06f6f5e\) · Commits · Yannick Mounguengui / Mounguengui_Yannick_GL · GitLab \(univ-lille.fr\)](#)



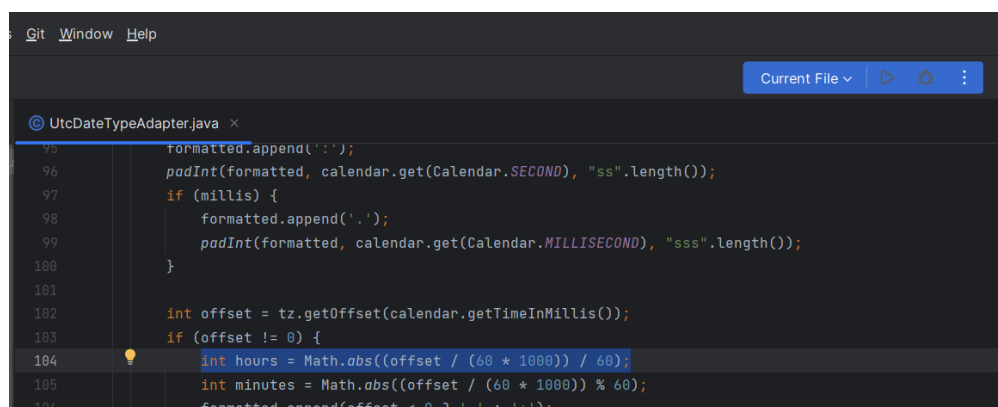
Dans la classe `LinkedTreeMap` du package `com.google.gson.internal` du dossier `gson` nous voyons que la fonction `readObject(objectinputstream in)` son paramètre `in`, n'est jamais utilisé dans cette méthode. Le problème d'avoir

une fonction avec un paramètre non utilisé est qu'il peut rendre le code source plus difficile à comprendre et à maintenir. Les développeurs qui lisent le code peuvent se demander pourquoi le paramètre a été défini si la fonction ne l'utilise pas, ce qui peut entraîner des erreurs de compréhension et de modification du code. Ainsi on peut l'enlever pour un code facile à comprendre et à maintenir et la méthode ne contient plus aucun paramètre.

- créer des variables pour supprimer des nombres magiques.

[modification du nombre de parametre de la methode readObject \(c06f6f5e\) · Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

(erreur séparation de commit)



```
95 formatted.append(':');
96 padInt(formatted, calendar.get(Calendar.SECOND), "ss".length());
97 if (millis) {
98     formatted.append('.');
99     padInt(formatted, calendar.get(Calendar.MILLISECOND), "sss".length());
100 }
101
102 int offset = tz.getOffset(calendar.getTimeInMillis());
103 if (offset != 0) {
104     int hours = Math.abs((offset / (60 * 1000)) / 60);
105     int minutes = Math.abs((offset / (60 * 1000)) % 60);
106     formatted.append(offset < 0 ? '-' : '+');
```

Dans la classe `UtcDateTypeAdapter` du paquetage `com.google.gson` du dossier `gson` nous observons un nombre magique qui se répète beaucoup "60". Dans le projet, l'utilisation de nombres magiques peut rendre le code difficile à comprendre et à maintenir. Ici, le nombre magique "60" à plusieurs endroits, il peut être difficile de savoir si chaque occurrence de "60" a la même signification ou si elles sont utilisées pour des raisons différentes. Alors on va créer une constante `private static final int MINUTES_IN_HOUR = 60`; Nous pouvons utiliser cette variable à la place du nombre magique 60.

Cela améliore la lisibilité du code et rend plus facile la maintenance du code si la valeur doit être modifiée à l'avenir.

```
        formatted.append('.');
        padInt(formatted, calendar.get(Calendar.MILLISECOND), "sss".length());
    }

    int offset = tz.getOffset(calendar.getTimeInMillis());
    if (offset != 0) {
        int hours = Math.abs((offset / (MINUTES_IN_HOUR * 1000)) / MINUTES_IN_HOUR);
        int minutes = Math.abs((offset / (MINUTES_IN_HOUR * 1000)) % MINUTES_IN_HOUR);
        formatted.append(offset < 0 ? '-' : '+');
        padInt(formatted, hours, "hh".length());
        formatted.append(":");
        padInt(formatted, minutes, "mm".length());
    }
}
```

- supprimer du code mort

[suppression code mort classe RuntimeAdapterFactory \(dc5416d5\) · Commits · Yannick Mounguengui / Mounguengui_Yannick_GL · GitLab \(univ-lille.fr\)](#)

```
152  /**
153   * Creates a new runtime type adapter using for {@code baseType} using {@code
154   * typeFieldName} as the type field name. Type field names are case sensitive.
155   *
156   * @param maintainType true if the type field should be included in deserialized objects
157   */
158  @Yannick Mounguengui
159  public static <T> RuntimeAdapterFactory<T> of(Class<T> baseType, String typeFieldName, boolean maintainType) {
160      return new RuntimeAdapterFactory<T>(baseType, typeFieldName, maintainType);
161  }
162
163  /**
164   * Creates a new runtime type adapter using for {@code baseType} using {@code
165   * typeFieldName} as the type field name. Type field names are case sensitive.
166   */
167  @Yannick Mounguengui
168  public static <T> RuntimeAdapterFactory<T> of(Class<T> baseType, String typeFieldName) {
169      return new RuntimeAdapterFactory<T>(baseType, typeFieldName, maintainType: false);
170  }
171
172  /**
173   * Creates a new runtime type adapter for {@code baseType} using {@code "type"} as
174   * the type field name.
175   */
176  @Yannick Mounguengui
177  public static <T> RuntimeAdapterFactory<T> of(Class<T> baseType) {
178      return new RuntimeAdapterFactory<T>(baseType, typeFieldName: "type", maintainType: false);
179  }
180
```

La classe RuntimeAdapterFactory du paquetage com.google.gson du

dossier extras comporte du code,ici `public static <T>`

```
RuntimeAdapterFactory<T> of(Class<T> baseType, String
typeFieldName, boolean maintainType) {
```

```
return new RuntimeTypeAdapterFactory<>(baseType,  
typeFieldName, maintainType);  
}
```

non utilisé. Le code mort peut poser plusieurs problèmes, tels que:

Inefficacité : le code mort peut augmenter la taille du projet et ralentir les performances globales du projet.

Confusion : le code mort peut rendre le code source plus difficile à comprendre, car il peut sembler que certaines parties du code sont nécessaires alors qu'elles ne le sont pas.

Risque de bugs : le code mort peut également créer des risques de bugs, car les développeurs peuvent être tentés de modifier le code mort, ou de le réutiliser dans d'autres parties du projet, ce qui peut causer des erreurs et des bogues dans le projet.

Ainsi pour éviter tout ceci on va supprimer ce code mort pour une meilleure efficacité et lisibilité du projet.

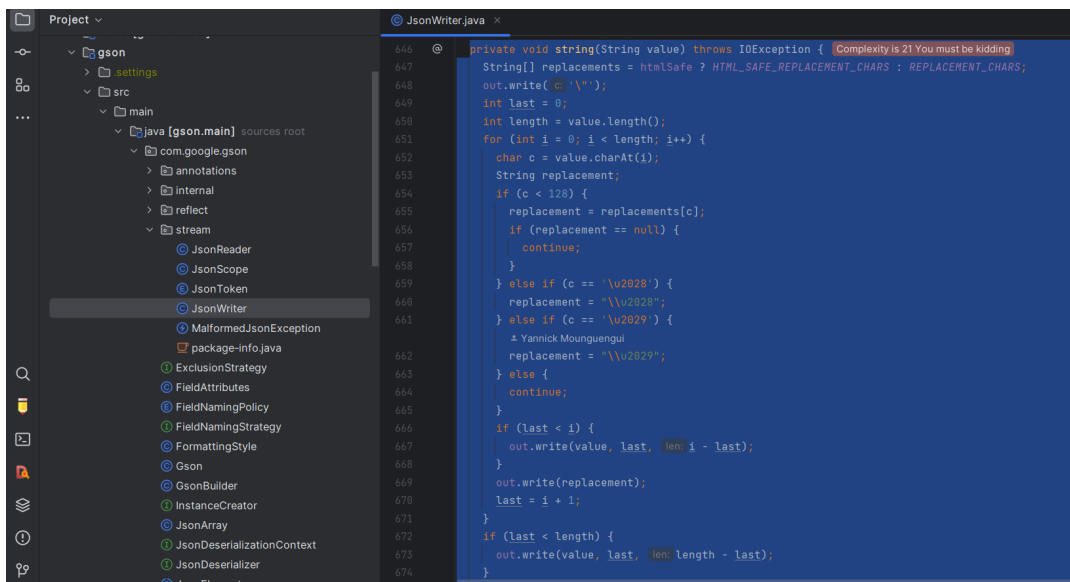
- réorganiser une classe pour le code soit bien structuré, les variables d'instance en début de classe, puis méthodes publiques et enfin méthodes privées

Dans ce projet, nous avons constaté que les classes sont toutes bien structurées avec des déclarations d'instance en début de classe et des méthodes en publiques et privées. De ce fait il est plus facile à lire et à comprendre, ce qui peut rendre les tâches de développement plus efficaces et plus rapides. Contrairement à du code mal structuré qui peut ralentir le processus de développement, car il peut être difficile de modifier et d'ajouter de nouvelles fonctionnalités au code existant.

II. Moyennes modifications

- réduire la complexité cyclomatique ou le nombre de lignes d'une méthode

[refactorisation de la methode String de la classe JsonWriter \(b6a8481b\) · Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)



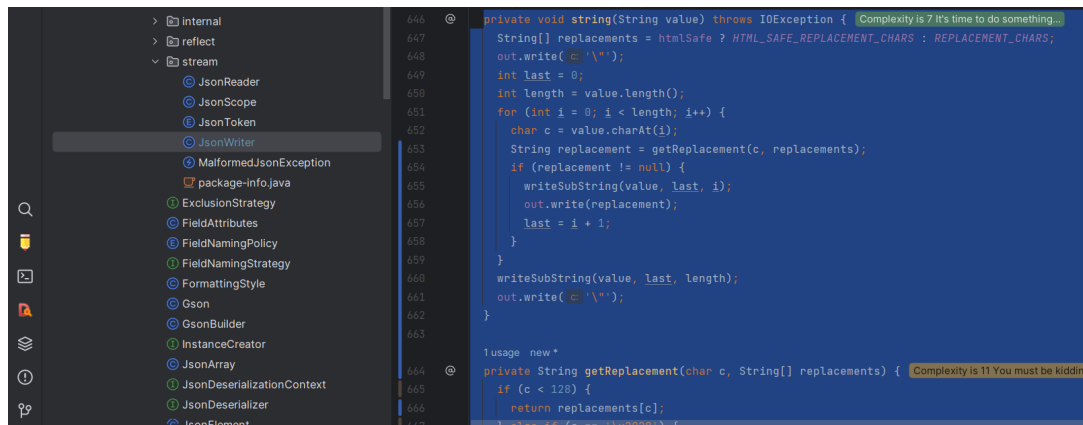
```
646 private void string(String value) throws IOException { Complexity is 21 You must be kidding
647     String[] replacements = htmlSafe ? HTML_SAFE_REPLACEMENT_CHARS : REPLACEMENT_CHARS;
648     out.write( @: "\");
649     int last = 0;
650     int length = value.length();
651     for (int i = 0; i < length; i++) {
652         char c = value.charAt(i);
653         String replacement;
654         if (c < 128) {
655             replacement = replacements[c];
656             if (replacement == null) {
657                 continue;
658             }
659         } else if (c == '\u2028') {
660             replacement = "\\u2028";
661         } else if (c == '\u2029') {
662             replacement = "\\u2029";
663             // Yannick Mounguengui
664             replacement = "\\u2029";
665         } else {
666             continue;
667         }
668         if (last < i) {
669             out.write(value, last, len: i - last);
670         }
671         out.write(replacement);
672         last = i + 1;
673     }
674     if (last < length) {
675         out.write(value, last, len: length - last);
676     }
```

Dans le paquetage com.google.gson.stream on a la classe JsonWriter qui a la méthode string(String value) ayant une complexité de 21. Cela montre que plus une méthode a une complexité élevée, plus elle est difficile à maintenir, à tester et à déboguer. Alors on va décomposer cela en différentes sous méthodes. Dans cette version refactorisée, nous avons extrait les fonctionnalités suivantes dans des méthodes distinctes :

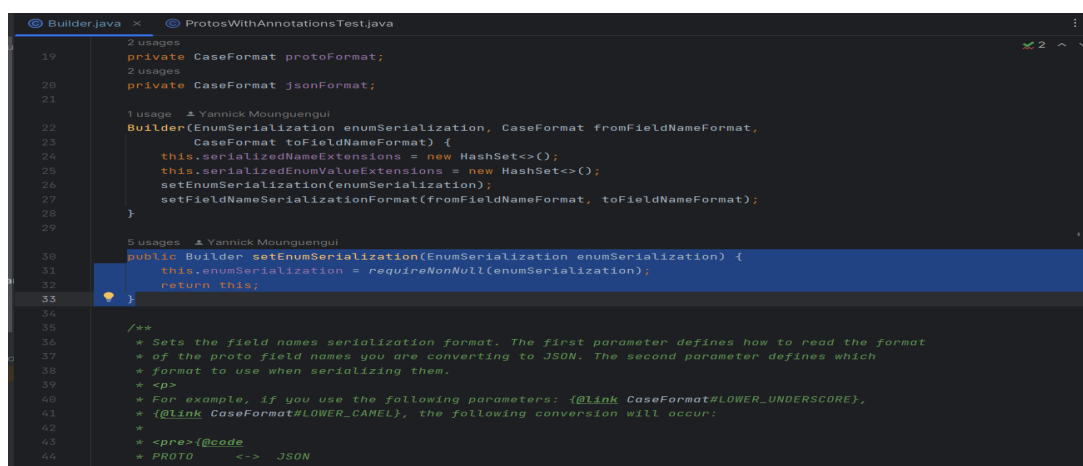
getReplacement : cette méthode prend un caractère et un tableau de chaînes de remplacement, et renvoie la chaîne de remplacement appropriée pour le caractère (ou null si aucun remplacement n'est nécessaire).

`writeSubString` : cette méthode écrit une sous-chaîne de la chaîne d'entrée dans le flux de sortie. Il prend la chaîne d'entrée, un index de début et un index de fin, et écrit les caractères de la sous-chaîne de l'index de début (inclusif) à l'index de fin (exclusif) dans le flux de sortie.

Ainsi la nouvelle complexité de la méthode est de 7. Une complexité plus faible et des sous méthodes permettent une meilleure compréhension de celle-ci.



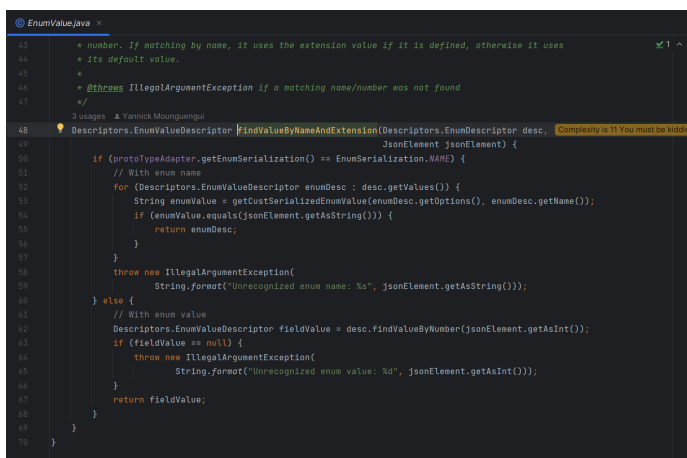
- décomposer une méthode qui à la fois retourne des informations et modifie l'état d'un objet



Dans la classe `Builder` la méthode `public Builder setEnumSerialization(EnumSerialization enumSerialization);`

du paquetage com.google.gson.protobuf du dossier proto fait à la fois getter et setter. Le problème d'une fonction faisant à la fois un setter et un getter est que cela peut causer de la confusion et de l'ambiguïté dans le code, en particulier pour les personnes qui ne sont pas familières avec le code. Cela peut également causer des erreurs car le comportement de la fonction peut ne pas être clair ou cohérent. Ainsi on va diviser la fonction en un getter et setter. Après modification, nous avons observé qu'il y a des erreurs de création d'objet Gson lors de plusieurs tests, la méthode modifiée jouait un rôle important dans le code de la classe Gson du paquetage com.google.gson du dossier gson et l'exécution de ses tests. Il est préférable de laisser celle-ci comme avant car il est un peu plus compliqué de modifier le fonctionnement de la classe Gson pour tout adapter ainsi que les différents tests.

- remplacer le fait qu'une méthode retourne un code d'erreur par le fait qu'elle lève une exception



```
43  * number. If matching by name, it uses the extension value if it is defined, otherwise it uses
44  * its default value.
45  *
46  * @throws IllegalArgumentException if a matching name/number was not found
47  */
48  Descriptors.EnumValueDescriptor findValueByNumberAndExtension(Descriptors.EnumValueDescriptor desc,
49  JsonElement jsonElement) {
50  if (protoTypeAdapter.getEnumSerialization() == EnumSerialization.NAME) {
51  // With enum name
52  for (Descriptors.EnumValueDescriptor enumDesc : desc.getValues()) {
53  String enumValue = getCustomSerializedEnumValue(enumDesc.getOptions(), enumDesc.getName());
54  if (enumValue.equals(jsonElement.getString())) {
55  return enumDesc;
56  }
57  }
58  throw new IllegalArgumentException(
59  String.format("Unrecognized enum name: %s", jsonElement.getString()));
60  } else {
61  // With enum value
62  Descriptors.EnumValueDescriptor fieldValue = desc.findValueByNumber(jsonElement.getAsInt());
63  if (fieldValue == null) {
64  throw new IllegalArgumentException(
65  String.format("Unrecognized enum value: %d", jsonElement.getAsInt()));
66  }
67  return fieldValue;
68  }
69  }
70 }
```

La méthode FindvalueByNameAndExtension de la classe EnumValue du paquetage com.google.gson du dossier proto peut être considérée comme une fonction qui retourne un code d'erreur. Mais elle retourne une valeur si elle est trouvée et elle peut être considérée comme fonction qui lève un code d'erreur sinon. De ce fait il n'est pas vraiment utile de la modifier un peu plus.

- supprimer de la duplication de codes entre méthodes

[ajout de la superclasse ReadTypeAdaptersNumber pour supprimer la duplication...](#)
(3a79b295) · Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab
(univ-lille.fr)

```
346 //
347 public static final TypeAdapter<Number> FLOAT = new TypeAdapter<Number>() {
348     @Override
349     public Number read(JsonReader in) throws IOException {
350         if (in.peek() == JsonToken.NULL) {
351             in.nextNull();
352             return null;
353         }
354         return (float) in.nextDouble();
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     @Override
378     public void write(JsonWriter out, Number value) throws IOException {
379         if (value == null) {
380             out.nullValue();
381         } else {
382             out.value(value.doubleValue());
383         }
384     }
385 };
```

La classe TypeAdapters du paquetage com.google.gson comporte 2 méthodes qui ont du code dupliquées qui sont read(JsonReader in). Cela augmente la complexité du code et peut rendre la maintenance plus difficile.

Ainsi pour résoudre ce problème on a supprimé ces méthodes dupliquées en les mettant dans une nouvelle classe ReadTypeAdaptersNumber pour réduire la complexité, éviter la duplication et rendre la maintenance plus facile.

```

package com.google.gson.internal.bind;

import com.google.gson.Gson;
import com.google.gson.JsonSyntaxException;
import com.google.gson.ToNumberStrategy;
import com.google.gson.ToNumberPolicy;
import com.google.gson.TypeAdapter;
import com.google.gson.TypeAdapterFactory;
import com.google.gson.reflect.TypeToken;
import com.google.gson.stream.JsonReader;
import com.google.gson.stream.JsonToken;
import com.google.gson.stream.JsonWriter;
import java.io.IOException;

public abstract class ReadTypeAdaptersNumber extends TypeAdapter<Number> {
    @Override
    public Number read(JsonReader in) throws IOException {
        if (in.peek() == JsonToken.NULL) {
            in.nextNull();
            return null;
        }
        return readNumber(in);
    }

    protected abstract Number readNumber(JsonReader in) throws IOException;
}

```

- ajouter un test pertinent

[ajout test pertinent à la classe SecurityTest vérifiant que la propriété... \(dbb11b03\) ·](#)

[Commits · Yannick Mounguengui / Mounguengui_Yannick_GL · GitLab \(univ-lille.fr\)](#)

The screenshot shows an IDE with the Gson class open. The left sidebar shows a package explorer with 'Gson' selected. The main editor shows the Gson class with a new test method being added. The test method is named 'testJsonValueDeserialization' and is annotated with '@Test'. It uses 'GsonBuilder' to create a Gson instance and 'fromJson' to deserialize a JSON string. The test asserts that the 'stringValue' field is null.

```

@Test
public void testJsonValueDeserialization() {
    String json = "{"stringValue":null}";
    Gson gson = gsonBuilder.create();
    BagOfPrimitives target = gson.fromJson(json, BagOfPrimitives.class);
    assertEquals(target.stringValue, null);
}

```

La classe Gson du paquetage com.google.gson a une classe test qui est SecurityTest. Dans cette classe tous les tests semblent être pertinents,nous allons alors ajouté un test qui vérifie que Gson est capable de désérialiser un flux JSON contenant un token non exécutable même s'il est créé sans GsonBuilder: `testJsonValueDeserialization`. Ce test permet de vérifier une autre fonctionnalité du code non testé auparavant pour une meilleure compréhension de celui-ci via les tests.

- corriger un test rouge ou orange

```

[INFO] -----
[INFO] Reactor Summary for Gson Parent 2.10.2-SNAPSHOT:
[INFO]
[INFO] Gson Parent ..... SUCCESS [ 0.893 s]
[INFO] Gson ..... SUCCESS [ 39.126 s]
[INFO] Gson Extras ..... SUCCESS [ 1.987 s]
[INFO] Gson Metrics ..... SUCCESS [ 0.268 s]
[INFO] Gson Protobuf Support ..... SUCCESS [ 9.542 s]
[INFO] -----
[INFO] BUILD SUCCESS

```

Dans le projet, il n'y a aucun test rouge ou orange à corriger car tous les tests passent correctement. Ainsi on ne peut rien corriger.

III. Grandes modifications

- décomposer une god classe

[creation classe EnumSerialization à partir de la god class ProtoTypeAdapter \(683b0459\) ·](#)

[Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

[extraction de la classe Builder de la God class ProtoTypeAdapter et... \(5b52510c\) ·](#)

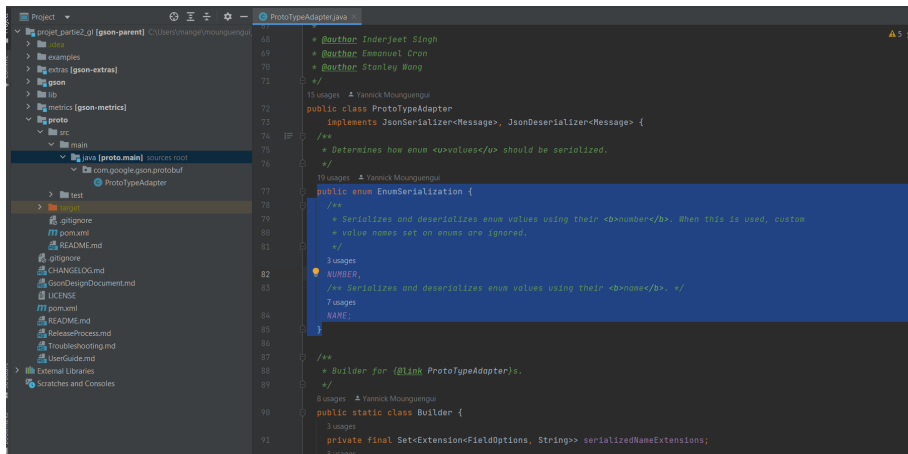
[Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

[refactoring de la methode deserialize de la God class ProtoTypeAdapter \(03e54d39\) ·](#)

[Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

[creation de la classe EnumValue à partir de la God class ProtoTypeAdapter \(d0a6908d\) ·](#)

[Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)



La classe ProtoTypeAdapter est une God classe comportant énormément de méthodes(à grande complexité ou pas),de classes et à l'intérieur d'elle.Cela entraîne trop de complexité dans le code et elle devient incompréhensible pour les développeurs ou visiteurs Pour la décomposer nous allons chercher à extraire les classes présentes,d'abord nous remarquer qu'elle comporte une classe Enum,on va l'extraire dans une sous classe à part EnumSerialization. Cette classe va définir les deux options de sérialisation/désérialisation possibles pour les valeurs Enum lors de la conversion d'objets entre le format Proto et JSON

```

363 /**
364  * Returns the enum value to use for serialization, depending on the value of
365  * {@link EnumSerialization} that was given to this adapter.
366  */
367 private Object getEnumValue(EnumValueDescriptor enumDesc) {
368     if (enumSerialization == EnumSerialization.NAME) {
369         return getCustomSerializedEnumValue(enumDesc.getOptions(), enumDesc.getName());
370     } else {
371         return enumDesc.getNumber();
372     }
373 }
374 /**
375  * Finds an enum value in the given {@link EnumDescriptor} that matches the given JSON element.
376  * either by name if the current adapter is using {@link EnumSerialization#NAME}, otherwise by
377  * number. If matching by name, it uses the extension value if it is defined, otherwise it uses
378  * its default value.
379  *
380  * @throws IllegalArgumentException if a matching name/number was not found
381  */
382 private EnumValueDescriptor findValueByNameAndExtension(EnumDescriptor desc,
383     JsonElement jsonElement) {
384     if (enumSerialization == EnumSerialization.NAME) {
385         // With enum name
386         for (EnumValueDescriptor enumDesc : desc.getValues()) {
387             String enumValue = getCustomSerializedEnumValue(enumDesc.getOptions(), enumDesc.getName());
388             if (enumValue.equals(jsonElement.getAsString())) {
389                 return enumDesc;
390             }
391         }
392         throw new IllegalArgumentException(
393             String.format("Unrecognized enum name: %s", jsonElement.getAsString()));
394     } else {
395         // With enum value
396         EnumValueDescriptor fieldValue = desc.findValueByNumber(jsonElement.getAsInt());
397         if (fieldValue == null) {
398             throw new IllegalArgumentException(

```

Ensuite on va extraire les différentes méthodes telle que EnumValueDescriptor qui utilise les valeurs de l'enum dans une classe appelée EnumValue. Cette classe fournit des méthodes pour travailler avec les énumérations, en particulier pour récupérer la valeur d'une énumération pour la sérialisation et la désérialisation, ainsi que pour trouver la valeur d'énumération correspondante en fonction d'un élément JSON.

```

/**
 * public static class Builder {
 *     private final Set<Extension<FieldOptions, String>> serializedNameExtensions;
 *     private final Set<Extension<EnumValueOptions, String>> serializedEnumValueExtensions;
 *     private EnumSerialization enumSerialization;
 *     private CaseFormat protoFormat;
 *     private CaseFormat jsonFormat;
 *
 *     private Builder(EnumSerialization enumSerialization, CaseFormat fromFieldNameFormat,
 *                     CaseFormat toFieldNameFormat) {
 *         this.serializedNameExtensions = new HashSet<>();
 *         this.serializedEnumValueExtensions = new HashSet<>();
 *         setEnumSerialization(enumSerialization);
 *         setFieldNameSerializationFormat(fromFieldNameFormat, toFieldNameFormat);
 *     }
 *
 *     public Builder setEnumSerialization(EnumSerialization enumSerialization) {
 *         this.enumSerialization = requireNonNull(enumSerialization);
 *         return this;
 *     }
 *
 *     /**
 *      * Sets the field names serialization format. The first parameter defines how to read the field
 *      * of the proto field names you are converting to JSON. The second parameter defines which
 *      * format to use when serializing them.
 *      * <p>
 *      * For example, if you use the following parameters: {@link CaseFormat#LOWER_UNDERSCORE},
 *      * {@link CaseFormat#LOWER_CAMEL}, the following conversion will occur:
 *      *
 *      * <pre>{@code
 *      * PROTO    <->   JSON
 *      * my_field  myField
 *      * foo       foo
 *      * n_id_ct   nIdCt
 *      * }</pre>

```

De plus nous avons une autre classe présente à l'intérieur qui est Builder, nous allons l'extraire pour en faire une classe à part. La classe Builder va contenir plusieurs méthodes pour configurer les options de sérialisation et de désérialisation pour la conversion des objets entre le format Proto et JSON. Enfin nous remarquons que la méthode deserialize a une complexité élevée, on va la refactoriser en différentes sous méthodes pour réduire la complexité générale de ProtoTypeAdapter. Ainsi après une nouvelle analyse des God class, Nous constatons qu'on a maintenant 11 God Class au lieu de 12 trouvé en 1ere partie d'analyse.

```

> Fmt Collapsible If Statements Count: 1
> Cyclomatic Complexity Count: 29
v Fmt God Class Count: 11
  > $Gson$Types Count: 1
  > Excluder Count: 1
  > Gson Count: 1
  > GsonBuilder.java Count: 1
  > ISO8601Utils Count: 1
  > JsonPrimitive Count: 1
  > JsonTreeWriter Count: 1
  > JsonWriter.java Count: 1
  > LinkedTreeMap Count: 1
  > TypeToken Count: 1
  > UtcDateTypeAdapter Count: 1

```

Ce qui fait que ProtoTypeAdapter a été divisé en 3 sous classes pour répartir les différentes responsabilités qu'il avait avant et réduire sa complexité.

Dans la classe `BagOfPrimitivesDeserializationBenchmark` du paquetage `com.google.gson.metrics` du dossier `metrics` on a 2 méthodes qui utilisent du code dupliquée. Cela peut entraîner divers problèmes:

Maintenance difficile : Si du code doit être modifié ou mis à jour, il devra être modifié dans deux endroits différents, ce qui augmente le risque d'erreur humaine et peut rendre la maintenance difficile

.Augmentation de la taille du code : Le code dupliqué augmente la taille du code et peut rendre la classe plus difficile à comprendre. Cela peut également rendre la compilation et l'exécution du code plus lentes.

Pour y remédier nous allons créer une classe abstraite `AbstractDeserializationBenchmark` permet de réduire la duplication de code en regroupant des fonctionnalités similaires dans une classe parente , afin que les sous-classes puissent hériter de ces fonctionnalités et les personnaliser selon leurs besoins spécifiques. Dans ce cas précis, les méthodes `timeBagOfPrimitivesStreaming` et `timeBagOfPrimitivesReflectionStreaming` sont toutes deux implémentées dans la classe parente `AbstractDeserializationBenchmark`, qui fournit une base pour les sous-classes qui souhaitent mesurer les performances de désérialisation de différentes structures de données. En héritant de cette classe, `BagOfPrimitivesDeserializationBenchmark` peut réutiliser ces méthodes et se concentrer sur la définition de la structure de données qu'elle souhaite désérialiser plutôt que de réécrire le code de base pour mesurer les performances de désérialisation.

```

 * @author Jesse Wilson
 * @author Joel Leitch
 */
public class BagOfPrimitivesDeserializationBenchmark extends AbstractDeserializationBenchmark {

    @Override
    protected BagOfPrimitives deserializeBagOfPrimitives(JsonReader jr) throws IOException, IllegalAccessException {
        jr.beginObject();
        BagOfPrimitives bag = new BagOfPrimitives();
        while (jr.hasNext()) {
            String name = jr.nextName();
            for (Field field : BagOfPrimitives.class.getDeclaredFields()) {
                if (field.getName().equals(name)) {
                    Class<?> fieldType = field.getType();
                    if (fieldType.equals(long.class)) {
                        field.setLong(bag, jr.nextLong());
                    } else if (fieldType.equals(int.class)) {
                        field.setInt(bag, jr.nextInt());
                    } else if (fieldType.equals(boolean.class)) {
                        field.setBoolean(bag, jr.nextBoolean());
                    } else if (fieldType.equals(String.class)) {
                        field.set(bag, jr.nextString());
                    } else {
                        throw new RuntimeException("unexpected: type: " + fieldType + ", name: " + name);
                    }
                }
            }
        }
        jr.endObject();
        return bag;
    }

    /**
     * Benchmark to measure deserializing objects using streaming.
     */
    @Benchmark
    public void timesBagOfPrimitivesStreaming(int reps) throws IOException, IllegalAccessException {
        for (int i = 0; i < reps; ++i) {
            StringReader reader = new StringReader(json);
            JsonReader jr = new JsonReader(reader);
            deserializeBagOfPrimitives(jr);
        }
    }

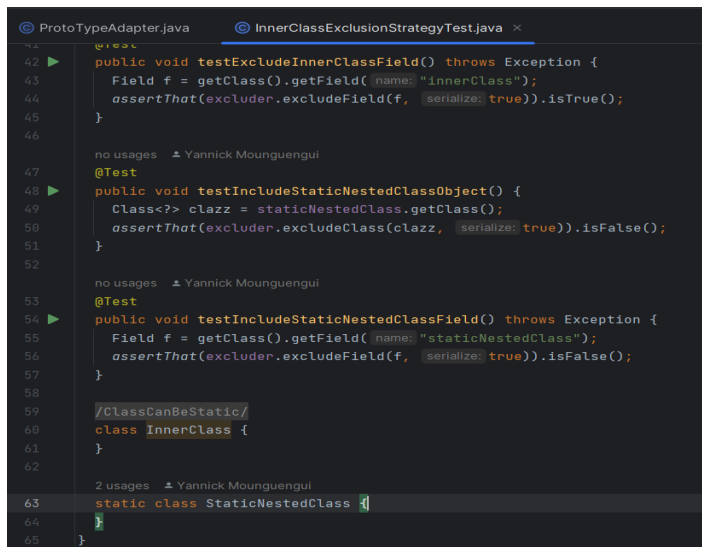
    /**
     * Benchmark to measure deserializing objects using reflection.
     */
    @Benchmark
    public void timesBagOfPrimitivesReflectionStreaming(int reps) throws Exception {
        for (int i = 0; i < reps; ++i) {
            StringReader reader = new StringReader(json);
            JsonReader jr = new JsonReader(reader);
            BagOfPrimitives bag = deserializeBagOfPrimitives(jr);
        }
    }
}

```

- supprimer des classes static

La suppression d'une classe statique peut être compliquée car elle peut être utilisée dans plusieurs parties du projet et il peut être difficile de déterminer toutes les références qui y sont faites. Si la classe a des membres statiques utilisés par d'autres parties du projet, cela peut entraîner des erreurs lors de l'exécution des tests si ces membres ne sont plus disponibles. De plus, la suppression d'une classe statique peut également affecter les performances de l'application, car cela peut nécessiter une réécriture du code qui utilise la classe ou une partie de celle-ci. Dans notre projet, il est recommandé de ne pas supprimer de classes statiques pour ne

pas affecter le bon fonctionnement du projet.

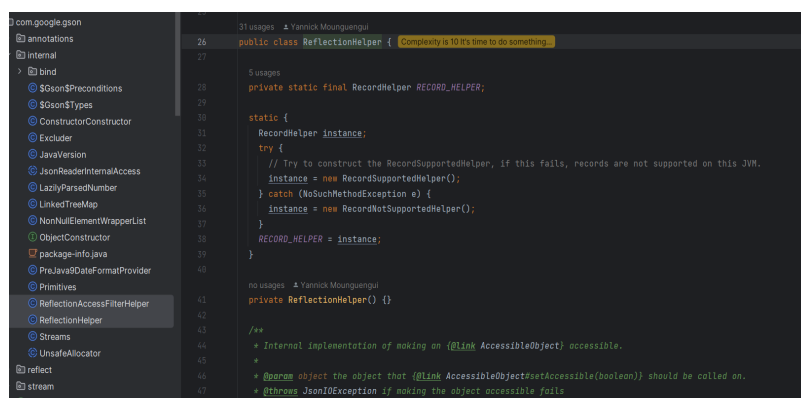


```
42 public void testExcludeInnerClassField() throws Exception {
43     Field f = getClass().getField( name: "innerClass");
44     assertThat(excluder.excludeField(f, serialize: true)).isTrue();
45 }
46
47 no usages 1 Yannick Mounguengui
48 @Test
49 public void testIncludeStaticNestedClassObject() {
50     Class<?> clazz = staticNestedClass.getClass();
51     assertThat(excluder.excludeClass(clazz, serialize: true)).isFalse();
52 }
53
54 no usages 1 Yannick Mounguengui
55 @Test
56 public void testIncludeStaticNestedClassField() throws Exception {
57     Field f = getClass().getField( name: "staticNestedClass");
58     assertThat(excluder.excludeField(f, serialize: true)).isFalse();
59 }
60
61 //ClassCanBeStatic/
62 class InnerClass {
63 }
64
65 2 usages 1 Yannick Mounguengui
66 static class StaticNestedClass {
67 }
```

Ici par exemple la classe statique StatiqueNestedClass de la classe InnerClassExclusionStrategyTest, est utilisée dans les tests pour vérifier que les classes statiques sont incluses dans la sérialisation par défaut. Si on supprime cette classe, les tests correspondants ne passeront plus car ils dépendent de la présence de cette classe. En effet, si vous retirez cette classe, la méthode testIncludeStaticNestedClassObject échouera car la classe InnerClassExclusionStrategyTest.StaticNestedClass ne sera plus trouvée.

• fusionner des classes

[fusion des classes ReflectionHelper et ReflectionAccessFilterHelper en une classe ReflectionHelper \(79c3409a\) · Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)



```
26 public class ReflectionHelper {
27     // Complexity is 10. It's time to do something.
28
29     // usages 1 Yannick Mounguengui
30     private static final RecordHelper RECORD_HELPER;
31
32     static {
33         RecordHelper instance;
34         try {
35             // Try to construct the RecordSupportedHelper, if this fails, records are not supported on this JVM.
36             instance = new RecordSupportedHelper();
37         } catch (NoSuchMethodException e) {
38             instance = new RecordNotSupportedHelper();
39         }
40         RECORD_HELPER = instance;
41     }
42
43     // usages 1 Yannick Mounguengui
44     private ReflectionHelper() {}
45
46     /**
47      * Internal implementation of making an (@Link AccessibleObject) accessible.
48      *
49      * @param object the object that (@Link AccessibleObject#setAccessible(boolean)) should be called on.
50      * @throws JsonIOException if making the object accessible fails
51      */
52 }
```

Il a été préférable de fusionner les classes `ReflectionAccessFilterHelper` et `ReflectionHelper` du paquetage `com.google.gson` du dossier `gson` en une seule classe `ReflectionHelper` car ces 2 classes fournissent des méthodes utilitaires pour travailler avec la réflexion dans Java telles que `isJavaType(Class<?> c)` de la classe `ReflectionAccessFilterHelper` : Cette méthode prend une instance de classe comme argument et renvoie un booléen indiquant si cette classe appartient au package java.

Et `makeAccessible(AccessibleObject object)` de la classe `ReflectionHelper`: rend accessible un `AccessibleObject`, c'est-à-dire un objet qui est déclaré accessible grâce au mot-clé "accessible". Cette fonction lève une exception `JsonIOException` si elle échoue. Ainsi ça évite de séparer les classes qui font les mêmes fonctionnalités même si c'est pour réduire la taille et la complexité d'une seule classe.

- supprimer des packages contenant peu de classes (en fonction du nombre de classes et de ce qui est fait, cette modification peut être considérée comme moyenne)

[suppression du cycle de dependance du paquetage google.gson.internal.sql avec... \(89ceb8d6\) · Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

mounguengui_yannick_gl / projet_partie2_gl / gson / src / main / java / com
/ google / gson / internal / sql /

Name	Last commit
..	
 <code>SqlDateTypeAdapter.java</code>	ajout dossier partie 2
 <code>SqlTimeTypeAdapter.java</code>	ajout dossier partie 2
 <code>SqlTimestampTypeAdapter.java</code>	ajout dossier partie 2
 <code>SqlTypesSupport.java</code>	ajout dossier partie 2

Nous avons remarqué qu'il y a un paquetage `com.google.gson.internal.sql` du dossier `gson` contenant peu de classes. Il n'y a pas nécessairement de problème à avoir des paquetage contenant peu de classes en soi. Cependant, cela peut rendre la structure du projet moins claire et plus difficile à comprendre pour les développeurs qui y travaillent. La suppression de paquetages contenant peu de classes peut améliorer la lisibilité et la maintenance d'un projet. En effet, si un paquetage contient très peu de classes, il peut être difficile de justifier son existence et cela peut ajouter de la complexité à la structure globale du projet. De plus, si un paquetage ne contient que quelques classes, il peut être plus facile de les déplacer vers un autre paquetage sans causer de perturbations majeures dans le code. Ainsi dans ce projet nous avons supprimé le paquetage `com.google.gson.internal.sql` et déplacé les classes de ce paquetage dans le paquetage `com.google.gson.internal`.

- utiliser un design pattern (MVC, Strategy, Composite, Decorator)

[ajout du pattern decorator de la classe UpperCaseFieldNamingStrategyDecorator... \(abe11744\) · Commits · Yannick Mounguengui / Mounguengui Yannick GL · GitLab \(univ-lille.fr\)](#)

Les design patterns sont des solutions éprouvées et documentées aux problèmes courants rencontrés dans le développement de logiciels. Ils fournissent un langage commun et une structure de conception éprouvée pour aider les développeurs à résoudre des problèmes similaires de manière cohérente et efficace. En utilisant des patterns, les développeurs peuvent améliorer la qualité de leur code, augmenter la réutilisabilité, faciliter la maintenance et réduire les coûts de développement. Notre projet contient divers patterns déjà existants tel que le pattern Strategy de la classe `FieldNamingStrategy` du paquetage `com.google.gson` du dossier `gson`. On va

ajouter un pattern en plus qui est le pattern Decorator pour apporter une fonctionnalité en plus au pattern Strategy précédemment.

```
1 package com.google.gson;
2
3
4 import java.lang.reflect.Field;
5
6 no usages
7 class UpperCaseFieldNamingStrategyDecorator implements FieldNamingStrategy { Complexity is 3 Everything is cool!
8     2 usages
9     private final FieldNamingStrategy delegate;
10
11 no usages
12 public UpperCaseFieldNamingStrategyDecorator(FieldNamingStrategy delegate) {
13     this.delegate = delegate;
14 }
15
16 4 usages
17 @Override
18 public String translateName(Field f) {
19     String fieldName = delegate.translateName(f);
20     return fieldName.toUpperCase();
21 }
22 }
```

La classe `UpperCaseFieldNamingStrategyDecorator` utilisant le pattern Decorator va ajouter une fonctionnalité de conversion des noms de champs en majuscules. Cela signifie que lorsque la méthode `translateName()` est appelée sur une instance de `UpperCaseFieldNamingStrategyDecorator`, elle appelle d'abord la méthode `translateName()` sur la instance `FieldNamingStrategy` qu'elle enveloppe, puis elle convertit le résultat en majuscules avant de le renvoyer. De ce fait nous allons ajouter un nouveau test dans `GsonTest` pour voir si cette fonctionnalité marche.

```
60 @Test
61 public void testUpperCaseFieldNamingStrategyDecorator() {
62     // Create a Gson instance with UpperCaseFieldNamingStrategyDecorator
63     Gson gson = new GsonBuilder()
64         .setFieldNamingStrategy(new UpperCaseFieldNamingStrategyDecorator(FieldNamingPolicy.IDENTITY))
65         .create();
66
67     // Create an object to serialize
68     MyClass obj = new MyClass("test", 123);
69
70     // Serialize the object using Gson
71     String json = gson.toJson(obj);
72
73     // Verify that the field names in the JSON are in uppercase
74     assertTrue(json.contains("\"MYSTRING\":\"test\""));
75     assertTrue(json.contains("\"MYINT\":123"));
76 }
77
78 2 usages new *
79 private static class MyClass { Complexity is 3 Everything is cool!
80     2 usages
81     private String myString;
82     2 usages
83     private int myInt;
84
85     1 usage new *
86     public MyClass(String myString, int myInt) {
87         this.myString = myString;
88         this.myInt = myInt;
89     }
89 }
```

Les codes ajoutés dans `GsonTest` testent l'utilisation du pattern Decorator avec la classe `UpperCaseFieldNamingStrategyDecorator` pour modifier la stratégie de

nommage des champs de la classe MyClass lors de la sérialisation avec Gson. La stratégie par défaut est FieldNamingPolicy.IDENTITY qui ne modifie pas les noms de champs, tandis que la nouvelle stratégie UpperCaseFieldNamingStrategyDecorator ajoute la fonctionnalité de changer les noms de champs en majuscules. Le test crée un objet de la classe MyClass et le sérialise en utilisant l'instance de Gson configurée avec la stratégie de nommage de champs personnalisée UpperCaseFieldNamingStrategyDecorator. Le test vérifie ensuite que les noms de champs dans la chaîne JSON résultante sont en majuscules en utilisant la méthode assertTrue() de JUnit. L'ensemble des tests exécutés nous montre qu'il n'y a aucune erreur ainsi notre nouveau test marche bien.

```
[INFO] -----
[INFO] Reactor Summary for Gson Parent 2.10.2-SNAPSHOT:
[INFO]
[INFO] Gson Parent ..... SUCCESS [ 0.971 s]
[INFO] Gson ..... SUCCESS [ 15.647 s]
[INFO] Gson Extras ..... SUCCESS [ 2.073 s]
[INFO] Gson Metrics ..... SUCCESS [ 0.316 s]
[INFO] Gson Protobuf Support ..... SUCCESS [ 10.169 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 29.749 s
[INFO] Finished at: 2023-04-10T01:15:56+02:00
[INFO] -----
```

Ainsi en combinant ces deux patterns, on peut obtenir une grande flexibilité dans la conception du projet.

- supprimer des cycles dans les dépendances entre packages.

En tenant cette partie j'ai rencontré 2 problèmes que j'ai répertorié:

Dépendances croisées : Parfois, deux paquetages peuvent dépendre l'un de l'autre de manière croisée, créant ainsi un cycle. Dans ce cas, la suppression du cycle peut nécessiter une refonte complète de la structure du code.

Dépendances cycliques complexes : Dans certains cas, il peut y avoir plusieurs cycles complexes dans les dépendances entre paquetages, ce qui rend la suppression des cycles encore plus difficile.

Cela n'a pas marché pour la dépendance entre le paquetage `com.google.gson.stream` et `com.google.gson.internal.bind`, plusieurs tests ne marchaient plus après que j'ai tenté de faire une interface pour que les méthodes du paquetage `com.google.gson.internal.bind` passent par une interface avant d'être utilisée dans le paquetage `com.google.gson.stream`.

IV. Conclusion

En somme, ce que nous avons appris grâce à ce projet d'un point de vue du génie logiciel est le fait que malgré un projet peut avoir une taille importante, on pourrait penser qu'il ne contient pas d'erreurs dans sa conception vu qu'il aurait eu énormément de temps à être conçu. Cela s'avère faux, après analyse du projet sous différents plans, nous constatons que plusieurs parties contiennent des erreurs de conception telles que du code mort, des méthodes non commentées, du code dupliqué, des méthodes avec une complexité trop grande et des classes faisant plusieurs fonctionnalités à la fois (God class). Tout ceci est mauvais car cela fait en sorte que le projet comporte énormément de bugs, est difficile à comprendre, à maintenir, à faire des extensions, à tester. Pour remédier à tout ceci, nous listons toutes ces erreurs et nous proposons différentes améliorations en changeant les parties du code concernées en ajoutant de nouveaux tests sur du code non testé, l'utilisation de design pattern pour de la fluidité, l'usage de superclasse pour la suppression de code dupliqué, la refactorisation des méthodes à complexité trop grande en différentes sous-méthodes et la décomposition de God class en plusieurs sous-classes par exemple. De ce fait, après ces diverses modifications le projet est mieux lisible, extensible et fluide, d'autres développeurs pourront y travailler sans avoir énormément de soucis.