Gson: Java Objects to JSON

Ercolino Matteo matr. 0522501462, Silvestri Simone matr. 0522501419

ACM Reference Format:
Ercolino Matteo matr. 0522501462, Silvestri Simone matr. 0522501419. 2023. **Gson: Java Objects to JSON**. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Gson is a Java library developed by Google that provides an easy-to-use framework for converting Java objects to JSON (JavaScript Object Notation) representation and vice versa. JSON is a lightweight data interchange format that is widely used for data serialization and communication between web services. It is available on GitHub to the following link: https://github.com/google/gson.

Gson aims to simplify the process of working with JSON in Java applications by providing a simple and intuitive API. It allows developers to seamlessly convert Java objects into their JSON representation and vice versa without the need for complex manual parsing or formatting.

Gson's key features include object serialization and deserialization, customization and flexibility through adapters and type hierarchies, annotation support for fine-tuning data conversion, integration with existing Java libraries, and robust error handling.

The Gson size is 11860 LOC, with 148 classes. The Gson repository is organized into several modules, each serving a specific purpose. The main module is **gson**, which contains the core source code of Gson. It includes JSON parsers, serialization and deserialization classes, utilities, and supporting classes.

Additionally, there is the **gson-protobuf** module that provides integration with Google Protocol Buffers. This module allows for serializing and deserializing Protocol Buffers objects using Gson.

The **gson-metrics** module offers metrics monitoring capabilities for Gson. With this module, performance data and statistics can be collected during object serialization and deserialization.

Lastly, there is the **gson-parent** module that acts as the parent module for the entire Gson project. It contains the Maven configuration file for the project and defines common dependencies and other basic configurations for the sub-modules.

This modular structure helps keep the Gson code organized and facilitates extensibility and reusability of functionalities. Each module plays a specific role and can be utilized based on the application's needs.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 SOFTWARE QUALITY ANALYSIS

We performed a software quality analysis by scanning the Github repository of the project using SonarCloud tool. 16 bugs and 457 code smells were identified. The analysis of the Gson project using SonarCloud revealed a total of 16 bugs, categorized as 3 critical, 6 major, and 7 minor issues.

- Among the minor bugs identified, a noteworthy problem is the usage of Double Brace Initialization (DBI). This technique creates an anonymous class with a reference to the instance of the owning object, which can potentially lead to memory leaks if the anonymous inner class is returned and held by other objects. Moreover, DBI is considered obscure and can confuse most maintainers. To address this, it is recommended to use alternatives such as Arrays.asList for collections or explicitly adding each item directly to the collection.
- Regarding the major bugs, two issues involve the need to either override the Object.equals(Object obj) method or rename it entirely to prevent any confusion. Additionally, two other bugs relate to the naming conflict between a method in the current class and a private method in the parent class, requiring a renaming to ensure clarity. Another major bug suggests using an "instanceof" comparison instead of comparing classes by name. But this is documented in the comment: "*Class was added in Java 9, therefore cannot use instanceof*". Therefore we didn't fix this bug. Lastly, one major bug indicates the necessity to either rename a method or correct the type of the argument(s) to properly override a method in the parent class. We renamed the method from *locationString* to *pathOfString*.
- The critical bugs encompass a single issue that urges the renaming of a method to avoid any potential confusion with the Object.finalize() method. Object.finalize() is called by the Garbage Collector at some point after the object becomes unreferenced. Overloading Object.finalize() is a bad idea because the overload may not be called by the Garbage Collector and users are not expected to call Object.finalize() and will get confused. We fixed this bugs.

2.1 GitHub Actions

Gson's team from Google have already implemented Actions for GitHub, we have just edited them to remove not relevant actions and command for our context.

To facilitate early bug detection, faster development cycles, and smoother deployments also to automate build, test, and deployment steps we setup a CI/CD environment using GitHub Actions. We created a .github/workflows directory in our repository and defined a workflow YAML file "build.yml" to specify the desired workflow steps. This file represents a GitHub Actions workflow that is triggered when a push or pull request event occurs in the repository. Its main purpose is to perform a series of steps to build and verify a Java project using Maven as the dependency management and code compilation tool. The workflow is executed on a virtual machine

running Ubuntu, and it includes stages such as code checkout, Java environment setup, and Maven-based project compilation. The final result is a streamlined and automated build process for our Java project.

3 DOCKER

We configured Docker to build the Gson library by adding to the root of the project a dockerfile.

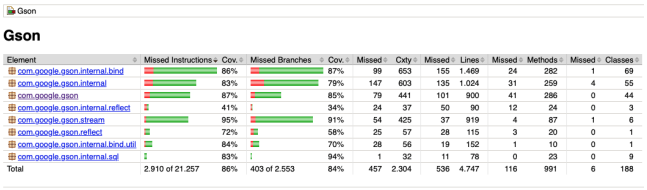
```
# Dockerfile
# Sets the base image as Maven version 3.8.5 with JDK 11.
FROM maven:3.8.5-jdk-11

# Sets the working directory inside the container to /app.
WORKDIR /app

# Copies the files and directories from the
# current directory (context) into the
# container's /app directory.
COPY . .

# Updates the package list inside the container
# and installs Nginx.
RUN apt-get update && apt-get install -y nginx

# Executes Maven commands to clean,
# verify the project,
# and generate Jacoco code coverage report.
RUN mvn clean verify jacoco:report

# Copies the Jacoco code coverage report
# generated in the previous step to
# the Nginx document root directory.
COPY gson/target/site/jacoco/ /var/www/html/

# Exposes port 80 of the container to allow
# incoming HTTP connections.
EXPOSE 80

# Specifies the command to run when the container starts.
# Starts Nginx in the foreground with
# the "daemon off" option.
CMD ["nginx", "-g", "daemon off;"]
```

In this dockerfile we configured the environment to use Maven image with Java JDK 11 from Eclipse Temurin. In addition we added also nginx a tool to expose some web-pages through the docker container. Since Gson library use Maven we added also the respective command adding also the JaCoCo code coverage. Finally, we exposed the results of JaCoCo through Ngix.

To pull the image from DockerHube, need to lunch this code:

```
docker pull silvered23/gson
```

4 CODE COVERAGE ANALYSIS



Gson

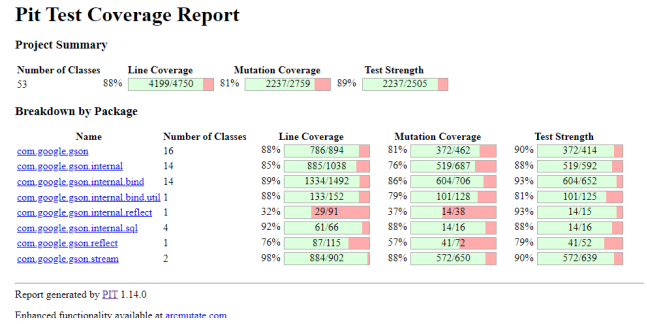
Gson

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.google.gson.internal.bind		86%		87%	99	653	155	1.469	24	282	1	69
com.google.gson.internal		83%		79%	147	603	135	1.024	31	259	4	55
com.google.gson		87%		85%	79	441	101	900	41	286	0	44
com.google.gson.internal.reflect		41%		34%	24	37	50	90	12	24	0	3
com.google.gson.stream		95%		91%	54	425	37	919	4	87	1	6
com.google.gson.reflect		72%		58%	25	57	28	115	3	20	0	1
com.google.gson.internal.bind.util		84%		70%	28	58	19	152	1	10	0	1
com.google.gson.internal.sql		83%		94%	1	32	11	78	0	23	0	9
Total	2.910 of 21.257	86%	403 of 2.553	84%	457	2.304	536	4.747	116	991	6	188

Figure 1: JaCoCo report

This chapter focuses on analyzing the code coverage of the Gson project using JaCoCo. Code coverage provides valuable insights into the effectiveness of testing by measuring the extent to which the source code is exercised. JaCoCo is a widely-used Java code coverage tool that generates detailed reports, helping developers identify areas that require more testing. Overall, the Gson project has a code coverage of 86%, indicating that 86% of the code has been tested. The highest code coverage percentage is achieved by the package "com.google.gson.stream" with 95% coverage, while the lowest coverage is found in the package "com.google.gson.internal.reflect" with 41% coverage. The package "com.google.gson.internal.bind" has the highest number of classes (69) and 282 methods. It also has a relatively high code coverage of 86%. Similarly, the package "com.google.gson.internal" has a significant number of classes (55) and methods (259), with a code coverage of 83%. The "com.google-gson" package has the highest number of methods (286) and a a relatively high code coverage of 87%. On the other hand, the package "com.google.gson.reflect" has the lowest number of classes (1) and 20 methods and achieves a high code coverage of 83%.

4.1 Mutation Testing



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
53	88% 4199/4750	81% 2237/2759	89% 2237/2505

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
com.google.gson	16	88% 786/894	81% 372/462	90% 372/414
com.google.gson.internal	14	85% 885/1038	76% 519/687	88% 519/592
com.google.gson.internal.bind	14	89% 1334/1492	86% 604/706	93% 604/652
com.google.gson.internal.bind.util	1	88% 133/152	79% 101/128	81% 101/125
com.google.gson.internal.reflect	1	32% 29/91	37% 14/38	93% 14/15
com.google.gson.internal.sql	4	92% 61/66	88% 14/16	88% 14/16
com.google.gson.reflect	1	76% 87/115	57% 41/72	79% 41/52
com.google.gson.stream	2	98% 884/902	88% 572/650	90% 572/639

Report generated by PIT 1.14.0

Enhanced functionality available at arcmutate.com

Figure 2: PiTest report

The mutation test analysis was done with PiTest. The line coverage percentage is 88%, that indicates that 88% of the lines in the class have been executed during the test suite's execution. This is a positive sign, suggesting that a significant portion of the code has been exercised by the tests. The mutation coverage of 81% means that out of a total of 2,759 mutations introduced by PiTest, 2,237 were successfully detected by the tests. A higher mutation coverage

implies that the tests are effective in detecting changes in the code and thus have a higher chance of identifying actual faults. The test strength of 89% indicates that out of the total 2,505 mutants that were generated, 2,237 were killed by the test suite. A higher test strength percentage suggests that the tests have a strong ability to identify and eliminate faults, indicating a robust test suite.

5 ENERGY TESTING

The SonarQube analysis was performed on the specified codebase, focusing on code quality and energy-related issues. The following results were obtained:

- **Bugs: 0.** The analysis did not detect any bugs in the codebase related to energy testing. This indicates that the code appears to be free from critical issues that could potentially impact the energy efficiency or performance.
- **Code Smells: 783.** SonarQube flagged a total of 783 code smells with the tag "ecocode." Code smells are programming practices or patterns that could indicate potential issues or areas for improvement. In the context of energy testing, these code smells might suggest opportunities to optimize the code for better energy efficiency.

While no bugs were found during the analysis, the presence of code smells indicates areas where the codebase could be further improved for energy efficiency. We fixed some code smells to remove unused variables but others cannot be fixed because of false positives or a lack of budget.

Table 1: EcoCode Code Smells

Rule	Number of Code Smells
(Java) Avoid multiple if-else statement	430
(Java) Avoid using global variables	159
(Java) Use ++i instead of i++	92
(Java) Do not unnecessarily assign values to variables	40
(Java) Don't concatenate Strings in loop, use StringBuilder instead.	28
(Java) Do not call a function when declaring a for-type loop	14
(Java) Avoid the use of Foreach with Arrays	12
(Java) Initialize builder/buffer with the appropriate size	4
(Java) Avoid getting the size of the collection in the loop	2
(Java) Avoid usage of static collections.	1
(Java) Free resources	1

6 PERFORMANCE TESTING

The project already has a module (gson-metrics) that implements internal benchmark tests against Gson using Caliper, a tool for measuring Java code performance. Nevertheless, we implemented our benchmarks with JMH because, as written in the GitHub repository of Caliper (https://github.com/google/caliper), it generally provides more accurate results than Caliper.

The provided results in the table 2 are from a performance testing conducted using JMH (Java Microbenchmark Harness) for a benchmark called *GsonBenchmark*. The benchmark consists of various operations involving JSON serialization and deserialization using the Gson library. We chose to test the performance of the main features of Gson.

Deserialization (fromJson): The `fromJsonEasyCar` operation demonstrates the highest throughput with a score of 1,908 ops/ns. It performs exceptionally well in deserializing a car object with simple attributes. The `fromJsonMediumGarage` operation has a lower throughput of 0.029 ops/ns, indicating slower performance in deserializing a more complex garage object. The `fromJsonHardLibrary` operation shows the lowest throughput of 0.009 ops/ns, indicating significantly slower performance in deserializing a library object with intricate dependencies.

Serialization (toJson): The `toJsonEasyPerson` operation exhibits the highest throughput with a score of 1,764 ops/ns. It performs well in serializing a person object with basic attributes. The `toJsonMediumStudent` and `toJsonHardPersonBook` operations show lower throughputs of 0.010 ops/ns and 0.016 ops/ns, respectively. These operations involve more complex objects and demonstrate comparatively slower performance.

Table 2 summarizes the throughput results for each operation.

Table 2: Performance Testing Results for GsonBenchmark

Operation	Deserialization Throughput (ops/ns)	Serialization Throughput (ops/ns)
fromJsonEasyCar	1.908	-
fromJsonMediumGarage	0.029	-
fromJsonHardLibrary	0.009	-
toJsonEasyPerson	-	1.764
toJsonMediumStudent	-	0.010
toJsonHardPersonBook	-	0.016

7 AUTOMATIC TEST CASE GENERATION

To improve coverage and test inadequately tested classes in the project, we used the Evosuite tool to automatically generate test cases. We also decided to generate tests for some classes that were already adequately tested but were important classes for the project.

For generating test cases, we utilized the *java -jar evosuite.jar* command. Here, *evosuite.jar* is the executable JAR file for the Evosuite tool. The *-class* flag is followed by the fully qualified name of the class for which we want to generate test cases. In this case, we used *com.google.gson.JsonParser* as the target class. Additionally, the *-projectCP* flag is used to specify the classpath for the project.

Once the test cases are generated, we compile them using the *javac* command. The *$(find ./evosuite-tests -name "*.java")* part is a command substitution that finds all the Java files in the *evosuite-tests* directory. The *-cp* flag is used to provide the classpath for compilation. Here, */gson/gson/target/classes*, */gson/evosuite.jar*, */gson/gson/target/ dependency/junit-4.13.2.jar*, and */gson/gson/target/*

dependency/hamcrest-core-1.3.jar are the paths to the required dependencies.

Finally, we execute the generated tests using the *java -cp* command. The *-cp* flag is followed by the classpath required for test execution. Here, *evosuite-tests*, */gson/gson/target/classes*, */gson/evosuite.jar*, and **/gson/junit-platform-console-standalone-1.9.2.jar* represent the necessary classpaths. The *org.junit.platform. console. ConsoleLauncher –scan-class-path* command launches the JUnit platform's console launcher and scans the classpath for test classes.

By utilizing these commands, we were able to automate the generation and execution of test cases, thus improving the testing process for our project.

To run the generated files, open the terminal in the root of the project and type:
java -cp evosuite-tests:/gson/gson/target/classes: */gson/evosuite.jar: */gson/junit-platform-console-standalone-1.9.2.jar org.junit.platform. console.ConsoleLauncher –scan-class-path*

8 SOFTWARE VULNERABILITIES

Metrics

6746 lines of code analyzed, in 219 classes, in 9 packages.

Metric	Total	Density*
High Priority Warnings		0.00
Medium Priority Warnings	1	0.15
Total Warnings	**1**	**0.15**

(Defects per Thousand lines of non-commenting source statements)*

Figure 3: Findsecbugs report

The metrics provided in the figure 3 are the result of utilizing Findsecbugs, a software tool used to analyze the Gson project. Findsecbugs conducted an extensive examination of the codebase, analyzing 6746 lines of code distributed among 219 classes and organized into 9 packages. We analyzed the jar file of the Gson module only since it is the only file that will be deployed for use. The purpose of this analysis was to identify potential software vulnerabilities and highlight areas that may require attention.

The analysis of the project using Findsecbugs revealed only one Medium priority warning and no High priority warnings. The detected warning relates to a security vulnerability in the regular expression pattern "-?(?:0|[1-9][0-9]*)(?:.[0-9]+)?(?:[eE][-+]?[0-9]+)?" which is susceptible to a denial of service attack (ReDOS). This type of vulnerability poses a potential risk of a denial of service scenario, wherein an attacker could exploit the weakness in the regular expression to craft malicious inputs that lead to excessive computation time, causing the application to become unresponsive or unavailable. While the absence of High priority warnings is reassuring, it is essential to address this Medium priority warning promptly by revisiting the regular expression implementation and applying necessary fixes to mitigate any potential ReDOS attacks.