Apache Commons Validator

Dependability Analysis Report - AY 2022/23

Delogu Nicolò
n.delogu@studenti.unisa.it
Università degli Studi di Salerno
Fisciano (SA), Campania, Italy

Mazza Dario
d.mazza6@studenti.unisa.it
Università degli Studi di Salerno
Fisciano (SA), Campania, Italy

***Abstract*—This report thoroughly explores the dependability of Apache Commons Validator, covering key aspects such as CI/CD, Docker containerization, code coverage, SonarCloud analysis, mutation testing, performance assessment, test case generation, vulnerability detection, and web application deployment. We highlight the significance of categorizing code smells, conducting refactoring to improve code quality and promoting eco-design principles for sustainability, as these practices collectively enhance the library's reliability and trustworthiness.**

I. Introduction

Apache Commons Validator is a widely-used Java library that plays a pivotal role in ensuring data integrity and reliability in software applications. This versatile library offers a wide range of validation tools, empowering developers to validate data input, from basic fields like email addresses and phone numbers to more complex data structures.

To conduct the analysis, we forked the original apache commons validator repository; all the work we have done can be found at this github repository:
https://github.com/xDaryamo/commons-validator.
The relevant branches include:

- **webapp**: which contains a spring boot sub-module and a related dockerfile;
- **refactoring**: which includes a refactored version of the master branch (we removed a certain amount of code smells)
- **evosuite_randoop**: which comprises tests generated with two tools, EvoSuite and Randoop;
- **microbenchmark_pre_refactoring**: which includes the benchmark packaging used to conduct performance testing;
- **security_analysis**: which holds the results of dependency check and find security bugs;
- **master**: the unaltered branch;

The Docker image for the project can be found at this link:
https://hub.docker.com/r/xdaryamo/commons-validator/tags

II. CI/CD

To begin with, the project was already in a state where it could be built without encountering any specific problems, and various GitHub Actions workflows had already been set up in the respective YAML files as required. Additionally we have prepared another action named "SonarCloud" (it has been set up in a related YAML file). In particular, our GitHub Actions workflow automates the process of building and analyzing a Java project using SonarCloud, a cloud-based code analysis platform. Triggered by pushes to the master branch or specific pull request events, the workflow runs on an ubuntu-latest runner. It ensures a full clone of the repository for precise analysis, sets up Java 17 using the 'zulu' distribution and optimizes performance by caching SonarCloud and Maven packages. The core step of the workflow involves executing the Maven command to build the project and run a SonarCloud analysis, utilizing environment variables for authentication and project key specification. This comprehensive approach streamlines code quality assessment and static code analysis, with the results accessible on the SonarCloud dashboard for further examination and action.

III. Docker and Web-App Deployment

In the GitHub Actions workflow, the "docker-image" action is configured in its dedicated YAML file. It plays a crucial role in the CI/CD pipeline, allowing secure authentication with Docker Hub to facilitate the subsequent steps of building and pushing our Docker image. The associated Dockerfile showcases a multi-stage build process for a Java web application utilizing Maven. In the initial "base" stage, it leverages a Maven image based on Eclipse Temurin 17 to compile the application. It creates a non-root user for enhanced security, resolves dependencies, and builds the application, ensuring a clean and isolated development environment. The subsequent "production" stage utilizes a minimal JRE image, copying the compiled JAR file from the previous stage. This separation of build and runtime environments minimizes the image size and reduces potential security risks. The resulting Docker image is optimized for production deployment, exposing port 8080 and launching the application with a single command. The Spring Boot web application described here offers users the ability to verify a diverse range of inputs, including URLs, email addresses, credit card numbers, IBANs, and dates. Users are presented with an intuitive menu interface, enabling them to choose the specific data type they wish to validate. Upon input submission, the application evaluates the provided information and promptly returns a comprehensible verdict, indicating whether the data is valid or invalid.

IV. Coverage

The number of lines of code covered by test case have been computed by employing JaCoCo, a code coverage report generator for Java projects. The overall coverage scores are illustrated in Figure 1, which shows the results at the beginning of our campaign. In summary, at an initial examination, it can be inferred that this open-source project boasts strong code coverage.

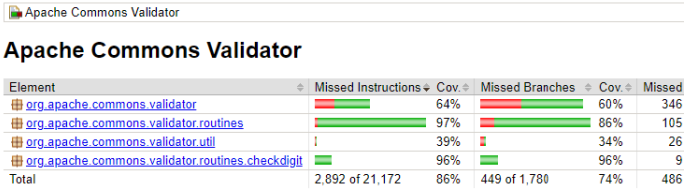


Figure 1: Overall coverage of Apache Commons Validator

The code coverage analysis of Apache Commons Validator reveals useful insights into its key packages: the routines" package as well as the "checkdigit" package, showcase thorough testing efforts dedicated to these aspect of the library (their scores stand respectively at 96% and 97%) whereas the "validator" package is the one which scored the lowest among the three at 64% (excluding the utility classes in the "utils" package).

V. Sonar-Cloud Analysis

A. Categorizing Code Smells

We employed Sonar Cloud as our analysis tool to scrutinize the Apache Commons Validator codebase, grouping code smells by their severity levels: **block, major, critical, minor, and informational**. Through this process, we discerned five major categories of code smells prevalent within its classes. These included the usage of multiple if statements instead of switch statements, instances of test cases devoid of assertions, return statements that could benefit from simplification, the recommendation to replace clone methods, and the need to steer clear of global variables. It is worth noting that during our analysis, we manually reviewed and confirmed that instances of multiple if statements and global variables were, in fact, false positives as SonarCloud is not always accurate.

B. Refactoring

To tackle the code smells that had been previously identified, we embarked on a comprehensive refactoring effort. Notably, we opted to replace the outdated clone method implementation with a more modern and maintainable copy constructor, effectively eliminating the necessity of relying on the cloneable interface. Additionally, we fortified our test suite by introducing assertions to test cases that had previously lacked them, thereby enhancing the robustness and reliability of the library. Furthermore, we streamlined the codebase by simplifying return statements, removing redundant and unnecessary boolean conditions, resulting in code that is not only more efficient but also easier to comprehend and maintain. Figure 2 shows the results at the end of our campaign.

C. Eco-Design

We incorporated the EcoCode plugin for Sonarqube into our development workflow. EcoCode is a collaborative project dedicated to scrutinizing code at a granular level to identify and rectify code structures that may have adverse ecological consequences, such as excessive energy and resource consumption. During our analysis, we pinpointed three distinct anomalies that had the potential to contribute to this negative impact: the use of arrays within for each loops, the preference for i++ over ++i in for loops, and the invocation of methods within the evaluation of for loop conditions. To mitigate these issues, we took proactive steps to enhance the code's eco-friendliness. This entailed substituting for each loops with more efficient while loops, making the switch from i++ to ++i for better performance, and moving method calls outside of the for loop conditions to minimize resource utilization.

VI. Mutation Testing

We conducted mutation testing on the Apache Commons Validator project using the Pitest plugin, a powerful tool for assessing the effectiveness of the test suite. This mutation campaign, namely its execution, required approximately six minutes to complete. During this process, Pitest introduced numerous artificial mutations into the codebase to evaluate the ability of the existing tests to detect these alterations. In order to provide a comprehensive assessment of the mutation testing results, we have presented the overall project-wide findings and insights. Figure 3 shows the results at the end of our campaign, offering a holistic view of how the entire Apache Commons Validator project performed.

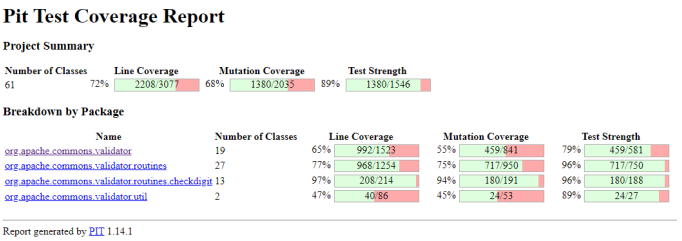


Figure 3: Overall Pitest Results

In conclusion, the current mutation coverage stands at 68%, a good percentage which makes us infer that the tests of Commons Validator are somewhat robust.

VII. Performance Testing

In order to benchmark Apache Commons Validator before and after refactoring, we employed JHM as our performance evaluation tool. Initially, we ran the benchmark on the unaltered codebase, allowing us to establish a baseline performance measurement. Subsequently, we embarked on a systematic refactoring process aimed at eliminating identified code smells. After successfully removing the targeted code smells, we reran

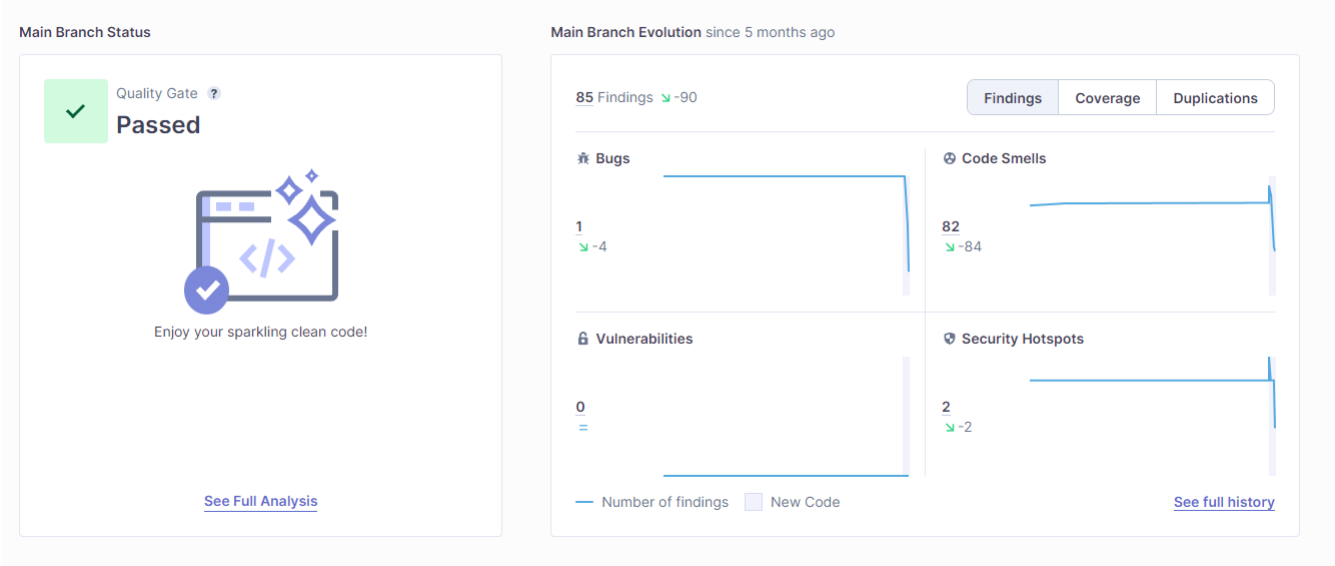


Figure 2: Code Smells Refactoring Campaign

the benchmark using JHM once more. By running the tool in throughput mode, we observed little to no improvement in performance (which can still be considered a positive outcome). Despite the absence of significant gains in raw throughput, the fact that the performance remained on par with the pre-refactoring benchmark indicates that the refactoring efforts were successful in enhancing code readability and maintainability without introducing any noticeable regressions. This underscores the value of refactoring in terms of code quality and long-term maintainability, even if it doesn't immediately translate into substantial performance gains. We selected the EmailValidator as our sample to illustrate our findings, as showed by Figure 4 and Figure 5.

Benchmark	Mode	Cnt	Score	Error	Units
EmailValidatorBenchmark.validateLongInvalidEmail	thrpt	5	224815,318 ±	3330,366	ops/s
EmailValidatorBenchmark.validateLongValidEmail	thrpt	5	215296,812 ±	5484,094	ops/s
EmailValidatorBenchmark.validateMultipleInvalidEmails	thrpt	5	1042,914 ±	9,296	ops/s
EmailValidatorBenchmark.validateMultipleValidEmails	thrpt	5	581,725 ±	6,656	ops/s
EmailValidatorBenchmark.validateShortInvalidEmail	thrpt	5	1855303,076 ±	40905,942	ops/s
EmailValidatorBenchmark.validateShortValidEmail	thrpt	5	1032211,155 ±	12495,359	ops/s
EmailValidatorBenchmark.validateSingleInvalidEmail	thrpt	5	1322249,406 ±	40493,070	ops/s
EmailValidatorBenchmark.validateSingleValidEmail	thrpt	5	733021,905 ±	15371,851	ops/s
EmailValidatorBenchmark.validateVeryLongInvalidEmail	thrpt	5	240206,400 ±	2666,202	ops/s

Figure 4: Pre Refactoring Benchmarking Results.
Benchmark executed on a system with an Intel Core i5-6600K (stock frequency) processor and 16GB of RAM

Benchmark	Mode	Cnt	Score	Error	Units
EmailValidatorBenchmark.validateLongInvalidEmail	thrpt	5	226023,649 ±	5185,551	ops/s
EmailValidatorBenchmark.validateLongValidEmail	thrpt	5	220097,621 ±	6768,077	ops/s
EmailValidatorBenchmark.validateMultipleInvalidEmails	thrpt	5	1018,256 ±	18,111	ops/s
EmailValidatorBenchmark.validateMultipleValidEmails	thrpt	5	594,920 ±	16,374	ops/s
EmailValidatorBenchmark.validateShortInvalidEmail	thrpt	5	1825389,316 ±	30233,167	ops/s
EmailValidatorBenchmark.validateShortValidEmail	thrpt	5	1025217,280 ±	22905,307	ops/s
EmailValidatorBenchmark.validateSingleInvalidEmail	thrpt	5	1359427,624 ±	23068,646	ops/s
EmailValidatorBenchmark.validateSingleValidEmail	thrpt	5	740675,074 ±	11576,712	ops/s
EmailValidatorBenchmark.validateVeryLongInvalidEmail	thrpt	5	239932,833 ±	4173,851	ops/s

Figure 5: Post Refactoring Benchmarking Results.
Benchmark executed on a system with an Intel Core i5-6600K (stock frequency) processor and 16GB of RAM

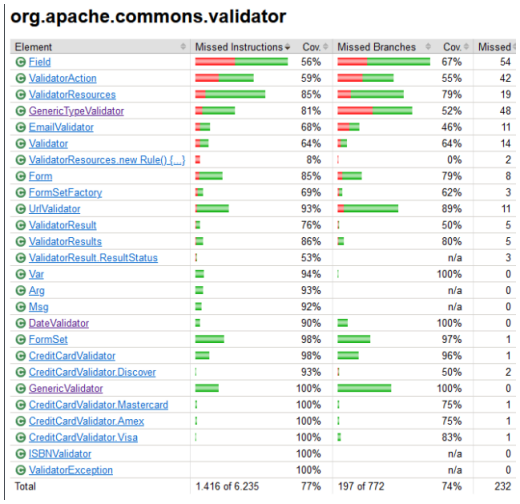
VIII. Generating Test Cases

In our quest to enhance the testing coverage of Apache Commons Validator, we harnessed the power of both EvoSuite and Randoop, two leading automated test case generation tools. Our journey began with a proper configuration, ensuring that both tools were finely tuned to the intricacies of our validation library. Once set up, we initiated the execution of their respective plugins, and the results were mostly positive. Our primary objective was to augment its code coverage, and the outcome met our expectations. As depicted in Figure 6 and Figure 7, our efforts yielded a moderate increase in the total test coverage of the "validator" package, around 13% more than the initial one. We did not include the "routines" package and the "checkdigit" package in our campaign due to their already excellent coverage scores (both are described in Section IV).

org.apache.commons.validator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed
Field		56%		67%	54
ValidatorAction		59%		55%	42
FormSet		46%		36%	22
GenericValidator		41%		35%	46
EmailValidator		13%		0%	21
ValidatorResources		85%		79%	19
GenericTypeValidator		80%		52%	48
Msg		0%		n/a	11
Form		77%		76%	10
Var		31%		0%	4
Arg		30%		n/a	5
DateValidator		0%		0%	12
Validator		58%		50%	16
ValidatorResources.new Rule() {...}		8%		0%	2
FormSetFactory		69%		62%	3
UrlValidator		93%		89%	11
ValidatorResult		76%		50%	5
ValidatorResults		86%		80%	5
ValidatorResult.ResultStatus		53%		n/a	3
ValidatorException		57%		n/a	1
CreditCardValidator		98%		96%	1
CreditCardValidator.Discover		93%		50%	2
CreditCardValidator.Mastercard		100%		75%	1
CreditCardValidator.Amex		100%		75%	1
CreditCardValidator.Visa		100%		83%	1
ISBNValidator		100%		n/a	0
Total	2.243 of 6.235	64%	306 of 772	60%	346

Figure 6: Coverage before test generation campaign



org.apache.commons.validator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed
Field		56%		67%	54
ValidatorAction		59%		55%	42
ValidatorResources		85%		79%	19
GenericTypeValidator		81%		52%	48
EmailValidator		68%		46%	11
Validator		64%		64%	14
ValidatorResources.new Rule() {...}		8%		0%	2
Form		85%		79%	8
FormSetFactory		69%		62%	3
UrlValidator		93%		89%	11
ValidatorResult		76%		50%	5
ValidatorResults		86%		80%	5
ValidatorResult.ResultStatus		53%		n/a	3
Var		94%		100%	0
Arg		93%		n/a	0
Msg		92%		n/a	0
DateValidator		90%		100%	0
FormSet		98%		97%	1
CreditCardValidator		98%		96%	1
CreditCardValidator.Discover		93%		50%	2
GenericValidator		100%		100%	0
CreditCardValidator.Mastercard		100%		75%	1
CreditCardValidator.Amex		100%		75%	1
CreditCardValidator.Visa		100%		83%	1
ISBNValidator		100%		n/a	0
ValidatorException		100%		n/a	0
Total	1.416 of 6.235	77%	197 of 772	74%	232

Figure 7: Coverage after test generation campaign

IX. Detecting Vulnerabilities

At this stage, two tools were employed to conduct a security inspection of the library: Find Security Bugs, an extension of the widely recognized SpotBugs, and OWASP Dependency-Check. For Find Security Bugs, we opted to utilize its command-line interface to carry out the analysis whilst for OWASP Dependency-Check we chose to employ its Maven plugin to a generate a related report.

Find Security Bugs identified potential vulnerabilities within the library, but it is important to note that all of these findings were of minimal relevance. All potential uncovered vulnerabilities were categorized as denial of service (DoS) issues. However, these findings were exclusively detected within regex rules, which are particularly relevant from a user-centric perspective and thus they were deemed false positives.

In contrast, OWASP Dependency-Check identified a total of 6 vulnerable dependencies within the 111 of the Commons Validator library, although some of them were marked as disputable. We have included Figure 8 for depicting this scenario as well as for showcasing the level of severity of each issue:

- **Bouncy Castle versions prior to 1.74**
  It contains a critical LDAP injection vulnerability. This vulnerability specifically impacts applications employing an LDAP CertStore from Bouncy Castle to validate X.509 certificates. In the course of certificate validation, Bouncy Castle inadvertently includes the certificate's Subject Name in an LDAP search filter without proper escaping, creating a vulnerability susceptible to LDAP injection attacks. It underscores the importance of promptly updating affected versions of Bouncy Castle to address this security issue and safeguard against potential exploits.
- **H2 Database Engine up to version 2.1.214**
  It centers around the ability to start the admin console via the command-line interface (CLI) with the "-webAdminPassword" argument, which allows the user to specify the password in plain-text for the web admin console. Nonetheless it is arguable that this behavior is not a vulnerability and that passing passwords on the command line is generally discouraged and considered insecure. It also suggests that database administrators (DBAs) and system administrators should be aware of these security practices. The dispute revolves around whether this behavior represents a vulnerability or if it is a result of improper usage by administrators who should be aware of security best practices. Ultimately, the assessment of this issue may vary depending on the perspective of security experts and administrators.
- **Jackson-databind library up to version 2.15.2**
  The report suggests that attackers can potentially exploit the library to cause a denial of service or another unspecified impact by using crafted objects that involve cyclic dependencies. However the perspective of DPC on this issue is that it does not qualify as a valid vulnerability. It argues that the steps required to construct a cyclic data structure and attempt to serialize it cannot be carried out by an external attacker, implying that it is a scenario that would typically involve internal or controlled processes. The dispute here hinges on whether the reported scenario is a realistic threat from an external attacker's perspective or if it requires conditions or actions that are generally considered within the control of trusted entities.
- **Maven core 3.8.0 and Maven settings 3.0**
  Apache Maven has historically followed repositories defined in a project's Project Object Model (POM). This behavior can be surprising to some users and could potentially introduce risks if a malicious actor takes control of a repository or impersonates it. In response to this concern, Maven is altering its default behavior in version 3.8.1 and later. Starting from this version, Maven will no longer automatically follow HTTP (non-SSL) repository references by default. This change is aimed at enhancing security by reducing the risk associated with non-SSL repositories.
- **Maven share utils 3.1.0**
  In versions of Apache Maven's maven-shared-utils earlier than 3.3.3, a vulnerability exists in the Commandline class where it can output double-quoted strings without adequate escaping. This issue can potentially be exploited for shell injection attacks.

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity
bcprov-jdk18on-1.71.jar	cpe:2.3:a:bouncycastle:bouncy-castle-crypto-package:1.71:*:*:*:*:*:*:* cpe:2.3:a:bouncycastle:bouncy_castle_crypto_package:1.71:*:*:*:*:*:*:* cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle:1.71:*:*:*:*:*:*:* cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle-java-cryptography-api:1.71:*:*:*:*:*:*:* cpe:2.3:a:bouncycastle:the_bouncy_castle_crypto_package_for_java:1.71:*:*:*:*:*:*:*	pkg:maven/org.bouncycastle/bcprov-jdk18on@1.71	MEDIUM
h2-2.1.214.jar	cpe:2.3:a:h2database:h2:2.1.214:*:*:*:*:*:*:*	pkg:maven/com.h2database/h2@2.1.214	HIGH
jackson-databind-2.15.2.jar	cpe:2.3:a:fasterxml:jackson-databind:2.15.2:*:*:*:*:*:*:* cpe:2.3:a:fasterxml:jackson-modules-java8:2.15.2:*:*:*:*:*:*:*	pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.15.2	MEDIUM
maven-core-3.0.jar	cpe:2.3:a:apache:maven:3.0:*:*:*:*:*:*:*	pkg:maven/org.apache.maven/maven-core@3.0	CRITICAL
maven-settings-3.0.jar		pkg:maven/org.apache.maven/maven-settings@3.0	CRITICAL
maven-shared-utils-3.1.0.jar	cpe:2.3:a:apache:maven_shared_utils:3.1.0:*:*:*:*:*:*:* cpe:2.3:a:utils_project:utils:3.1.0:*:*:*:*:*:*:*	pkg:maven/org.apache.maven.shared/maven-shared-utils@3.1.0	CRITICAL

Figure 8: Dependency Check Results