# Software Dependability Project Report

Aouni Wrocklage
Janna Piontek
Luka Nola
https://github.com/orgs/Erasmus-In-Salerno/repositories

## 1 INTRODUCTION

Why did we choose this project? We chose Apache Commons dbUtils for our Software Dependability course because it offers several advantages for a thorough security analysis. First, its **manageable size** makes it an ideal candidate for an in-depth examination without the complexity of larger frameworks. This allows us to focus on key components and security-related aspects without getting overwhelmed by an extensive codebase.

Additionally, although dbUtils is not part of Spring itself, it is commonly used in **Spring applications** to simplify complex JDBC (Java Database Connectivity) operations. This makes it highly relevant in real-world use cases, particularly in terms of securing database interactions in Spring environments. By analyzing dbUtils, we can gain valuable insights into its role within Spring-based applications and identify potential security concerns tied to database access.

A major factor in our decision is the emphasis of dbUtils on **resource management and security**. By abstracting JDBC operations, developers can avoid common issues, such as resource leaks and unclosed connections, which are often sources of security vulnerabilities. This makes it an excellent project for evaluating how well it handles these risks.

Finally, as part of the **Apache Commons project**, dbUtils benefits from strong community support and comprehensive documentation. This facilitates our ability to conduct a thorough security audit while assessing how well it adheres to security best practices over time.

In summary, we selected dbUtils because it strikes a balance between relevance, simplicity, and real-world security implications, particularly in the context of database operations within Spring applications.

## 2 SONARCLOUD ANALYSIS

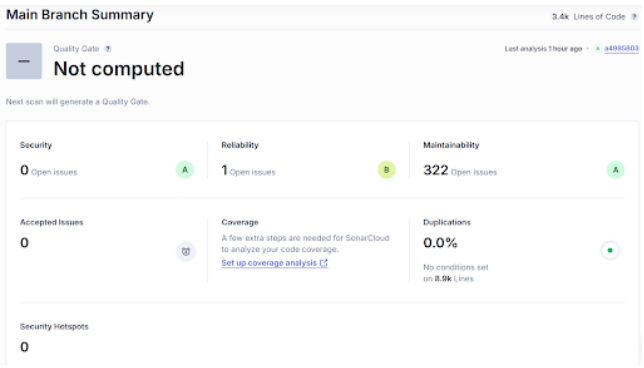


**Figure 1: SonarCloud analysis**

Analyzing the main branch of commons-dbutils with SonarCloud, one reliability issue and 322 maintainability code smells were found. The maintainability issues are separated into 239 low-severity issues, 56 medium-severity issues, and 27 high-severity issues, the latter representing the issues we will focus on. 16 of the high-severity issues relate to JUnit tests and are based on the rule 'tests should include assertions', therefore requesting to add at least one assertion to the test case. Most of these issues relate to the DBUtilsTest class (src/test/java/org/apache/commons/dbutils/DbUtilsTest.java). Resolving them is marked to contribute to the clean code principle of adaptability. Another six of the high-severity issues violate the clean code principle of intentionality by leaving insufficiently documented empty methods in the code. To resolve the issue it is necessary to add a nested comment, to complete the implementation or to throw an UnsupportedOperationException. Three more high-severity issues refer to the repeated duplication of string literals as error messages inside one class, instead of defining a constant for them. The last two high-severity issues relate to one method each, whose cognitive complexity is too high, meaning that the code is difficult to understand. While in general several different aspects contribute to the cognitive complexity of a method (https://docs.codeclimate.com/docs/cognitive-complexity), the criticized methods mainly use multiple conditionals and loops nested inside each other. Solving the issues therefore requires a restructuring of the methods.

**Not fixed high-severity issues:**

- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMl&id=Erasmus-In-Salerno_commons-dbutils
- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMm&id=Erasmus-In-Salerno_commons-dbutils
- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMn&id=Erasmus-In-Salerno_commons-dbutils

These three issues weren't fixed because the underlying rule "tests should include at least one assertion" did not really apply to the specific code sections. All three issues are related to JUnit tests calling an overloaded method, that closes a connection, result set or a statement respectively. They test null handling, but as the method is supposed to do nothing when given a null parameter, the only assertion that could be made, is that no exception is thrown. As JUnit tests fail by default when an exception is thrown, adding an assertion would be unnecessary.

- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMo&id=Erasmus-In-Salerno_commons-dbutils
- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMp&id=Erasmus-In-Salerno_commons-dbutils
- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMq&id=Erasmus-In-Salerno_commons-dbutils

- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMr&id=Erasmus-In-Salerno_commons-dbutils
- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMs&id=Erasmus-In-Salerno_commons-dbutils
- https://sonarcloud.io/project/issues?open=AZKLhIh2Sh2-aRAfcuMt&id=Erasmus-In-Salerno_commons-dbutils

The same applies to these six issues, all of which refer to tests without assertions of the same method. This method (closeQuietly()) calls the close method that was tested in the test cases mentioned above. The only difference between the two methods is that SQL exceptions are ignored. Therefore, calling the closeQuietly method with null has the same effect as calling the close method with null, meaning that the justification why the corresponding (3) issues were not fixed is the same as above. The other three issues test whether SQL exceptions are properly ignored. If everything works as expected, no exception reaches the test method, so the test succeeds. If something goes wrong and an exception is thrown, the test will fail anyway, so again, no assertions are needed.

## 3 DOCKER

Since the project is a database utility library there was no real way of building, dockerizing and presenting the image without some sort of application that uses it as a dependency. For this reason, a web app was created. The web app, built using Spring Boot, consists of a simple UI that allows the user to see all the animals in the database, to add a new one, or to delete one by id. The database connection and queries are managed with our fork of the DbUtils library. For the detailed explanations on how to run, build and compose the app, check out the project README.md. The image we are building contains this app while we compose it together with the postgres database it needs to be connected to.

### Building the Docker Image

In the case of Spring Boot, building a Docker image is trivial; by running the command:

```
$ mvn spring-boot:build-image -DskipTests
```

`-DskipTests` was used because the tests have not been set up on this project so there was no need to run them.

This created the image named `dbutils-webapp:latest`. That was tagged and pushed to DockerHub using the commands:

```
$ docker tag dbutils-webapp:latest
    lukanola/dbutils-webapp:latest
$ docker push lukanola/dbutils-webapp:latest
```

and is available on this link: https://hub.docker.com/repository/docker/lukanola/dbutils-webapp/. This image is now ready for orchestration.

An alternative approach would be to write the `Dockerfile` similar to Figure 2 which would start with a `maven` base to build the package and after that using the `openjdk` package as a base, would run the built `jar`.

### Composing the Containers

Since the app requires a database connection to function and demonstrate the DbUtils library we need to orchestrate two containers, one containing the app itself and another containing the postgres database. We can do this by creating a `docker-compose.yml` file and defining two services - db and app.

```
FROM maven:3.8.5-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

**Figure 2: Dockerfile example**

The **db container** runs from the public image `postgres:latest` while exposing the correct port and setting the correct environment variables.

The **app container** runs from the dbutils-webapp we built and while exposing the port 8080 and setting the environment variables. Since we don't want the app to run before the database is running in the container we set that the app container **depends** on the db container.

Now the project is runnable and deployable on any machine (that has Docker installed) and can be managed by a service for managing containerized apps such as Kubernetes.

## 4 CODE COVERAGE ANALYSIS WITH JACOCO

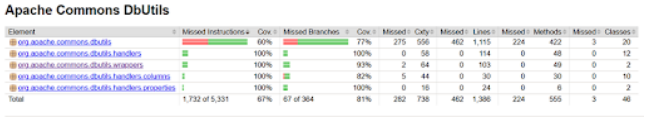


**Figure 3: Overview of JaCoCo Results**

When looking more closely at the result you see that the missed things from the tests are nearly all concentrated in the base package, where also the biggest part of the logic lies.

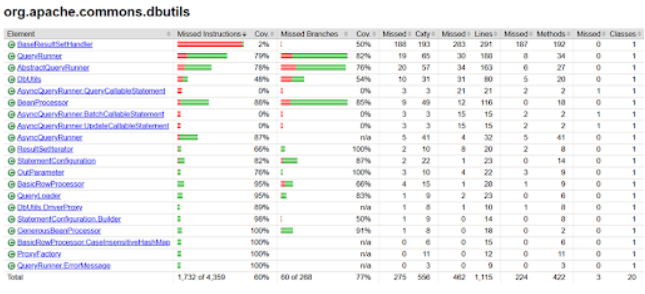


**Figure 4: JaCoCo overview for the package "org.apache.commons.dbUtils"**

A closer analysis of the results highlights significant disparities in test coverage among the classes. Most classes exhibit close to 80% coverage, yet the `BaseResultSetHandler` stands out negatively, with an abysmal 2% instruction coverage, 2.5% complexity coverage, and just 3% line coverage. This class, the largest in the package, serves as a utility for interacting with `ResultSet` objects by providing pre-defined methods to streamline database operations and minimize boilerplate code while maintaining flexibility for various conversion needs.

While the library as a whole demonstrates a generally acceptable test coverage level, the glaring coverage gap in the `BaseResultSetHandler` is a critical weakness, given the extensive logic and code it contains. Addressing this shortfall would significantly improve the library's reliability and robustness.

Beyond the `BaseResultSetHandler`, the `DbUtils` class also demands attention, with only 48% test coverage. Additionally, the `QueryRunner` and `AbstractQueryRunner` classes, both falling just below the 80% threshold, further underline the need for more comprehensive testing. These classes have been identified as priorities for additional test case generation to enhance overall code quality and coverage metrics.

## Brief Description of the Worst Tested Classes

In the following section, the classes that do not have good test coverage are described briefly.

***BaseResultSetHandler***. The `BaseResultSetHandler` is an abstract class designed to simplify the process of converting SQL `ResultSet` objects into various target types, such as lists, maps, or custom objects. It serves as a base class for implementing reusable result set handlers. The class provides a range of standard utility methods for handling `ResultSets` (e.g., `absolute`, `beforeFirst`, `deleteRow`, `getObject`, and many more). These methods are intended to make working with `ResultSet` easier and to eliminate repetitive code when interacting with SQL query results.

***DbUtils***. `DbUtils` is a utility class that provides a simple set of helper functions for working with JDBC (Java Database Connectivity). This class is designed to make handling database resources easier and to address common issues, such as properly closing connections, statements, and result sets.

***QueryRunner***. The `QueryRunner` class simplifies executing SQL queries and updates in Java applications. It provides a high-level API for managing database interactions, focusing on reducing repetitive boilerplate code required when working with JDBC directly. Supports SQL queries (`SELECT`), updates (`INSERT`, `UPDATE`, `DELETE`), and batch operations. Integrates with pluggable `ResultSetHandler` implementations to process and map `ResultSet` data into Java objects.

***AbstractQueryRunner***. The base class for the described `QueryRunner` above.

## 5 MUTATION TESTING WITH PITEST

We are using Pitest to evaluate the quality of the existing tests in the "**Apache Commons DBUtils**" repository. For executing the pitest-analysis, here and in future applications, the command "mvn org.pitest:pitest-maven:mutationCoverage" was used to evaluate the mutation coverage with pitest.

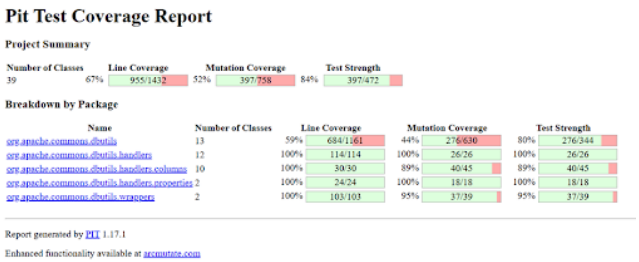


**Figure 5: Pitest Analysis Overview**

While the **line coverage** was previously analyzed in the JaCoCo report, the primary focus of the Pitest analysis lies in the **mutation coverage** and **test strength**. Although the overall Test Strength of 84% is relatively high, the Mutation Coverage of 52% is noticeably lower.

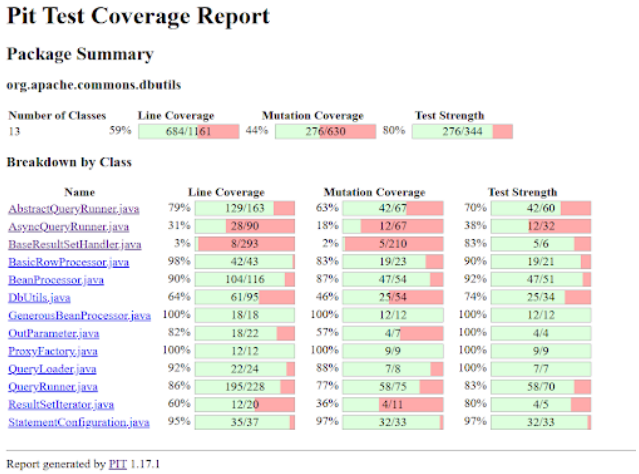


**Figure 6: "org.apache.commons.dbutils" Package Pit Test Overview**

A low mutation score is often related to low line coverage because, although mutations are applied to all lines of code, only those mutations in lines executed by tests can potentially be killed. For untested lines of code, any mutations will inevitably survive, as there are no tests to evaluate them.

As seen in Figure 6, this is particularly evident in the classes **`AsyncQueryRunner`** and **`BaseResultSetHandler`**, as both have very low line coverage and, consequently, fail to achieve meaningful mutation coverage. To improve these metrics, it is essential to add tests for these classes.

The **`AsyncQueryRunner`** also stands out negatively in the analysis of "Test Strength": while all other classes achieve a test strength

of over 70%, this class reaches only 36%. This indicates that the few existing tests neither sufficiently cover nor effectively evaluate this class.

These results clearly demonstrate that the critical and central class `AsyncQueryRunner` requires a more thorough examination, both in terms of test quantity, quality, and their ability to target and cover mutations effectively.

## 6 MICROBENCHMARK PERFORMANCE TESTS USING JMH

The **JMH** library was used to create microbenchmark performance tests for some of the most cumbersome classes and methods in the project. Those are:

- `BasicRowProcessorBenchmark.java`
  - `toArray`
  - `toBean`
  - `toBeanList`
- `DbUtilsBenchmark.java`
  - `close`
  - `closeQuietly`
  - `commitAndClose`
  - `rollbackAndClose`
- `QueryRunnerBenchmark.java`
  - `query`
  - `batch`
  - `update`

A new project, located in the root of the original library, was created to separate the dependencies and their effects on the result of the tests. The original library was added as a dependency of this new project.

### Running the benchmark

The benchmark can be run in the terminal with:

```
mvn clean install in the root
mvn clean install in ./benchmarks and
java -jar target/prova-1.0-SNAPSHOT.jar -rf json in
     ./benchmarks
```

`-XX:+EnableDynamicAgentLoading` can be added as an option to the last command to prevent the Mockito dynamic agent loading warnings.

### Test Results

The result of running these tests can be seen in the Figure 7 or in the `jmh-result.json` included in the project.

```
# Run complete. Total time: 01:14:17

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.

NOTE: Current JVM experimentally supports Compiler Blackholes, and they are in use. Please exercise
extra caution when trusting the results, look into the generated code to check the benchmark still
works, and factor in a small probability of new VM bugs. Additionally, while comparisons between
different JVMs are already problematic, the performance difference caused by different Blackhole
modes can be very significant. Please make sure you use the consistent Blackhole mode for comparisons

Benchmark                                          (rowCount)   Mode  Cnt    Score     Error   Units
BasicRowProcessorBenchmark.benchmarkToArray                 1  thrpt    5   75.652 ±   9.166  ops/ms
BasicRowProcessorBenchmark.benchmarkToArray                10  thrpt    5   77.311 ±   3.681  ops/ms
BasicRowProcessorBenchmark.benchmarkToArray               100  thrpt    5   76.071 ±   5.573  ops/ms
BasicRowProcessorBenchmark.benchmarkToBean                  1  thrpt    5   35.419 ±   6.404  ops/ms
BasicRowProcessorBenchmark.benchmarkToBean                 10  thrpt    5   35.457 ±   5.300  ops/ms
BasicRowProcessorBenchmark.benchmarkToBean                100  thrpt    5   35.355 ±   6.551  ops/ms
BasicRowProcessorBenchmark.benchmarkToBeanList              1  thrpt    5  358.894 ± 157.917  ops/ms
BasicRowProcessorBenchmark.benchmarkToBeanList             10  thrpt    5  366.503 ±  84.414  ops/ms
BasicRowProcessorBenchmark.benchmarkToBeanList            100  thrpt    5  358.533 ± 103.410  ops/ms
QueryRunnerBenchmark.benchmarkBatch                         1  thrpt   10   18.750 ±   6.729  ops/ms
QueryRunnerBenchmark.benchmarkBatch                        10  thrpt   10   18.963 ±   6.583  ops/ms
QueryRunnerBenchmark.benchmarkBatch                       100  thrpt   10   18.936 ±   6.901  ops/ms
QueryRunnerBenchmark.benchmarkComplexBatch                  1  thrpt   10   15.181 ±   2.287  ops/ms
QueryRunnerBenchmark.benchmarkComplexBatch                 10  thrpt   10    2.233 ±   1.476  ops/ms
QueryRunnerBenchmark.benchmarkComplexBatch                100  thrpt   10    0.220 ±   0.179  ops/ms
QueryRunnerBenchmark.benchmarkComplexQuery                  1  thrpt   10   46.460 ±   2.078  ops/ms
QueryRunnerBenchmark.benchmarkComplexQuery                 10  thrpt   10   46.643 ±   1.234  ops/ms
QueryRunnerBenchmark.benchmarkComplexQuery                100  thrpt   10   46.466 ±   2.179  ops/ms
QueryRunnerBenchmark.benchmarkQuery                         1  thrpt   10   46.005 ±   1.744  ops/ms
QueryRunnerBenchmark.benchmarkQuery                        10  thrpt   10   46.370 ±   1.537  ops/ms
QueryRunnerBenchmark.benchmarkQuery                       100  thrpt   10   45.947 ±   2.368  ops/ms
QueryRunnerBenchmark.benchmarkUpdate                        1  thrpt   10   49.432 ±   5.697  ops/ms
QueryRunnerBenchmark.benchmarkUpdate                       10  thrpt   10   49.153 ±   6.368  ops/ms
QueryRunnerBenchmark.benchmarkUpdate                      100  thrpt   10   48.963 ±   5.470  ops/ms
DbUtilsBenchmark.benchmarkClose                           N/A   avgt    5    2.657 ±   0.827   us/op
DbUtilsBenchmark.benchmarkCloseQuietly                    N/A   avgt    5    2.648 ±   1.035   us/op
DbUtilsBenchmark.benchmarkCommitAndClose                  N/A   avgt    5    5.259 ±   1.834   us/op
DbUtilsBenchmark.benchmarkRollbackAndClose                N/A   avgt    5    5.332 ±   1.168   us/op

Benchmark result is saved to jmh-result.json
```

**Figure 7: JMH results**

All of the results were expected. In this state, they represent the baseline for changes; meaning there should not be big differences when it comes to changing these methods in the future.

### The Way the Test Was Conducted

The tests are run in 5/10 iterations to reduce the effect of randomness and variance in the results. Also, 5/10 warmups are performed before the actual iterations to allow the system/JVM to stabilize, ensuring that any setup or transient state does not skew the final results.

The simplest are the tests for the **`DbUtils`** class. Each benchmark starts by opening a mock connection and closing it in the end. The chosen benchmark mode for these methods was **average time** since we want to benchmark the time it takes for the connection to close and commit/rollback to execute. Since `benchmarkCommitAndClose` and `benchmarkRollbackAndClose` both include the close call as well, it makes sense that the average time for them to execute is longer than just `benchmarkClose` and `benchmarkCloseQuietly`. We can notice that the difference in average execution time between the `close` and `closeQuietly` methods is negligible, being around 2.6 microseconds per operation. The same can be said about `commit` and `rollback` with those being around 5.25 microseconds per operation.

The chosen benchmark mode for **`BasicRowProcessor`** was **throughput** because checking the amount of operations the system can run in a time frame tells us how the system performs under load and what the expected output is under **high load**. The tests consist of a setup that mocks a database connection and the data in the database that needs to be managed. In testing, we change the number of rows from 1 to 10 to 100 to better gauge the system performance under higher load. What we can see from the results is that the change in the number of rows doesn't affect the results as much; perhaps a higher number would do so, but for the scope of the usage of this library, the performance differences

are negligible. In terms of performance, `toBeanList` is generally more efficient when dealing with multiple rows because it processes the entire `ResultSet` in a single operation, reducing the overhead associated with multiple method calls. This aligns with the benchmarking results observed, where `toBeanList` has a significantly higher throughput than `toBean` and `toArray`.

Since the **`QueryRunner`** class deals with data and includes lots of connections that need to be tended to at the same time, **throughput** was the best benchmark mode. These benchmarks showed a lot of variations in the results and during warmup, so the number of iterations for both was increased to 10, which seems to have had a positive impact on the variance of the results. In the same way we changed the number of rows from 1 to 10 to 100 for the `BasicRowProcessor`, the same was done here with the parameters. The benchmarks all include mocking of the database connection and the data; however, some methods are tested twice.
The reason for this is to demonstrate how **costly and complex operations** affect the **throughput**. Between `benchmarkQuery` and `benchmarkComplexQuery`, as well as `benchmarkBatch` and `benchmarkComplexBatch`, we notice that the complexity of the method lowers the throughput, essentially making the execution slower (as expected).

It shows that the results of these benchmarks are as expected and now serve a purpose of being the baseline for future changes; allowing for easier detection of performance regressions and also helping validate performance improvements. Other than serving as a reference for changes, benchmarks such as these can help point out bottlenecks in the system, find issues before they pop up in usage, and much more.

# 7 AUTOMATED TEST GENERATION

## Randoop

To identify poorly tested classes for generating additional test cases, the JaCoCo code coverage and PIT mutation testing results were analyzed. Based on this analysis, six classes were selected as priorities for further testing due to their low coverage or significant importance within the project. Even though having a high instruction coverage, the `BeanProcessor` was added because of its total number of missed instructions, which provided room for improvements. The chosen classes are:

- `org.apache.commons.dbutils.BaseResultSetHandler`
- `org.apache.commons.dbutils.DbUtils`
- `org.apache.commons.dbutils.QueryRunner`
- `org.apache.commons.dbutils.AsyncQueryRunner`
- `org.apache.commons.dbutils.AbstractQueryRunner`

To generate test cases with Randoop, the following command was issued on project level in Windows 10 PowerShell:

```
& 'C:\Program Files\Java\jdk-11.0.2\bin\java.exe'
-classpath
     "..\randoop-4.3.3\randoop-all-4.3.3.jar;target\classes"
randoop.main.Main gentests
     --classlist=..\PoorlyTestedClasses.txt
     --time-limit=600
```

As the project's `pom.xml` Maven file declares the language and compilation level as 11, Java 11 is explicitly called to run Randoop to ensure compatibility with the project. The time-limit was set to 600, following the suggestion in the tool's documentation (https://randoop.github.io/randoop/manual/, accessed 8.12.2024) to multiply the default 100 by the number of tested classes.

Randoop generated a JUnit test suite of 9092 regression test cases divided into 19 test classes. Neither error-revealing nor invalid tests were generated. After moving the test files to the project's test folder and adding the corresponding package declaration to the beginning of each file, the regression test suite ran successfully, as expected. Despite the high number of test cases and the length of each test file (approximately 35,000 lines of code), the test suite could be run reasonably fast.

## JaCoCo

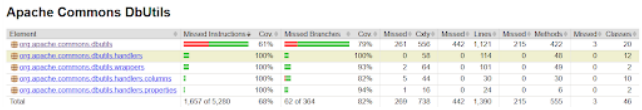


**Figure 8: Overview of JaCoCo results**

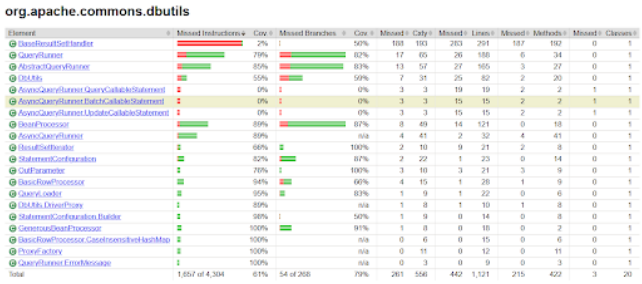


**Figure 9: JaCoCo overview for the package "org.apache.commons.dbUtils"**

## Randoop-Generated Test Suite Results

The with Randoop generated test suite significantly improved the test coverage of the `AbstractQueryRunner` and `DbUtils` classes and also provided better coverage results for the `AsyncQueryRunner` classes while it barely improved the `QueryRunner` class and did not improve the test coverage of the `BaseResultSetHandler` class at all.

There are multiple possible reasons for the results. As the randomly generated test cases do not take into account the already existing test cases that existed before using Randoop, it is more unlikely that test cases are generated, which cover prior untested code. This especially applies to larger classes which are already quite well tested. The `QueryRunner` class is such an example.

Differences in the results can also originate from the structure of the classes under test. As the `BaseResultSetHandler` is an abstract class, containing only protected and private methods, meant to be

extended by the user of the dbUtils project, Randoop was not able to generate any test cases for it.

Afterall, Randoop is a tool based on random generation. If no randomness controlling seed is provided which stays the same, test generations will differ from each other, even if the code base stays unchanged, making it impossible to fully explain the results in detail.

## Generation of Test Cases with GitHub Copilot

While it makes sense to use Randoop on more classes, because in theory the tool is meant to be used on whole projects, we wanted to target the test generation with GitHub Copilot even more on still poorly tested classes. Therefore, the BaseResultSetHandler, AsyncQueryRunner (especially the inner classes), and DbUtils classes were chosen as subjects for the test generation with GitHub Copilot, as they stick out with results below 75% for line, instruction, and branch coverage and mutation coverage below 50%.

***Test Generation for BaseResultSetHandler.*** GitHub Copilot Chat (hereafter just referred to as Copilot) was used as an integrated tool in IntelliJ Ultimate to generate more test cases for the prior very poorly tested (line coverage of 2%) BaseResultSetHandler class. Copilot mainly used the BaseResultSetHandler (class under test) and the BaseResultSetHandlerTest classes to generate the test cases, but sometimes also used the pom.xml and the BaseTestCase class, which the BaseResultSetHandlerTest class extends, as resources.

After the initial prompt "generate test cases for the BaseResultSetHandler class," some fine-tuning was needed to generate proper test cases. As the BaseResultSetHandler is an abstract generic class, its test class needs to provide its own class which extends the BaseResultSetHandler and overwrites the handle() method. All other methods of the BaseResultSetHandler can only be called from inside the overwritten handle() method, because otherwise, the ResultSet, on which all methods work, is null. At first, Copilot did not understand this and tried to call all methods separately, which always caused NullPointerExceptions. Copilot was taught that each test method needs its own implementation of the handle() method, by providing it with an example improvement of a faulty test method.

Another issue was that Copilot generated test cases which used raw forms of the BaseResultSetHandlerTest, so all "Object" return results needed to be cast back to what they should resemble. Even though it worked, it is not a good way to do it, so Copilot was asked to change the prior generated test methods to use a specific type instead of Object.

When the generated test cases looked good and did not cause any problems anymore, Copilot was asked to generate test cases for all still untested methods. As the answer was interrupted because it became too long, the prompt was changed to "generate test methods for the next 20 still untested methods of the BaseResultSetHandler." This worked well several times after each other, each time resulting in an output of about 300 lines. A downside of generating test cases iteratively with Copilot was the increasing number of test methods which were already defined in the test class. Finally, "make sure not to generate test methods which are already defined in BaseResultSetHandlerTest.java" was added to the prompt to contain the problem, but it did not really improve the result. If an output included already existing methods, these few methods were deleted, while the rest was included in the test class. Other small variations of the prompt were tried until the line coverage (analyzed by IntelliJ) of the BaseResultSetHandler class was above 80%.

In the process, almost 200 test methods were generated, spanning around 2650 lines of code. The quality of the tests was not analyzed. All tests passed, no errors were revealed.

***Test Generation for DbUtils.*** Compared to the BaseResultSetHandler, DbUtils is much smaller and already better tested (line coverage around 50%). Therefore, a more focused approach was needed. After highlighting some of the untested methods, the prompt "/tests write tests for the highlighted methods, that can be added to the tests in #DbUtilsTest.java" was used to generate test cases for the methods. The command was repeated with another highlighted section of untested methods. In total, around 15 test methods were generated, improving the test coverage of the DbUtils class significantly.

***Test Generation for AsyncQueryRunner.*** The AsyncQueryRunner itself already has a satisfactory amount of tests. However, its inner classes BatchCallableStatement, UpdateCallableStatement, and QueryCallableStatement do not have any tests. Copilot was used to address this gap. Tests for the first two inner classes were not generated because both classes are marked as "Deprecated." For QueryCallableStatement, Copilot did generate a test, but the test did not work as Copilot had issues instantiating a correct object of type AsyncQueryRunner.QueryCallableStatement. This issue had to be resolved manually.

This example highlights the limitations of Copilot for flawless test creation, particularly when dealing with more complex code structures, such as correctly instantiating objects with intricate dependencies or nested class designs.

## Changes in code coverage with JaCoCo

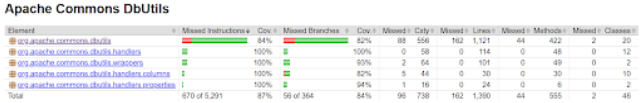


**Figure 10: Overview of JaCoCo results**

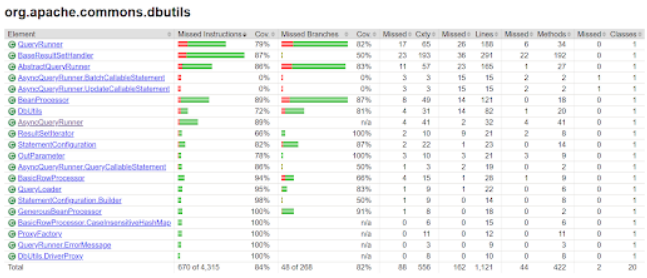


**Figure 11: JaCoCo Overview for the package org.apache.commons.dbUtils**

The generation of test cases with Copilot significantly improved the instruction coverage of the classes `BaseResultSetHandler` (2% → 87%) and `DbUtils` (55% → 72%). As the generated tests did not reveal errors, they can be classified as regression tests. As these, they lower the risk of introducing bugs and unintended changes in method behavior, improving maintainability especially for the `BaseResultSetHandler`.

## Comparison: Randoop - Github Copilot

In comparison to the test generation with Randoop, the results show a clear superiority of Copilots test generation in terms of bringing test coverage close to the 80% threshold. The generated tests could be seamlessly integrated into existing test files, opening the possibility to adapt them and change their order manually, which would not be possible in the code overhead Randoop produces for the coverage improvements. On the other hand the generation needed more manual interactions, making it a less automatic process and increasing the time needed for the improvements.

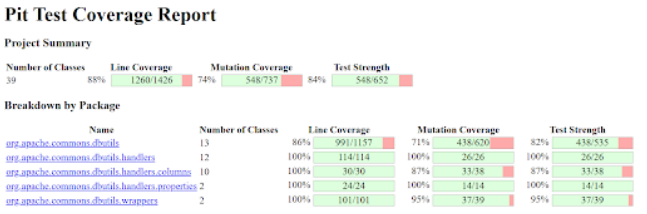


**Figure 12: Pitest Analysis Overview**

## Changes in Mutation Coverage with Pitest

As highlighted in the Pit Test analysis, the primary focus of this evaluation is on Mutation Coverage and Test Strength. Following the new test generation, the total number of tests included in the test strength calculation increased significantly, rising from 344 to 652—an improvement of nearly 90%. Alongside the increase in test quantity, the test strength itself improved by 4 percentage points, reaching 84%.

Similarly, mutation coverage, another critical metric in this analysis, experienced a notable increase from 44% to 74%. Overall, these results demonstrate that the generated tests have significantly enhanced the quality of testing within this library. However, a more detailed examination of the `dbUtils` base package is necessary to identify potential areas for further improvement and address any missed opportunities.

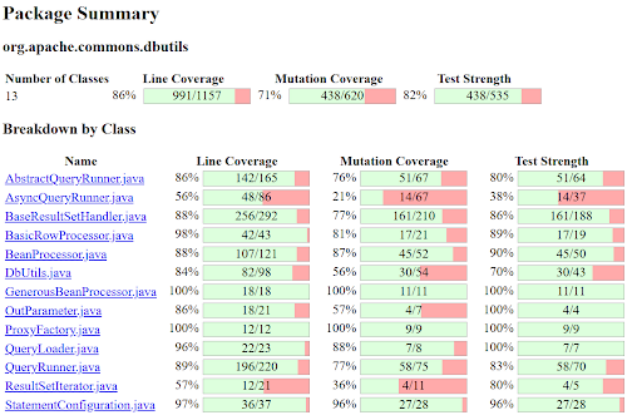


**Figure 13: "org.apache.commons.dbutils" Package Pit Test Overview**

Although the overall Mutation Coverage of this package improved significantly from 44% to 71%, and Test Strength saw a modest increase of 2 percentage points alongside a substantial rise in the number of tests, some classes still lack adequate mutation coverage and test strength. The test generation primarily targeted the classes `AsyncQueryRunner`, `DbUtils`, `QueryRunner`, `AbstractQueryRunner`, `BaseResultSetHandler`, and `BeanProcessor`. While all six classes experienced notable improvements in both Mutation Coverage and Test Strength, there is still room for enhancement in `DbUtils` and `AsyncQueryRunner`. In particular, `AsyncQueryRunner` remains deficient in handling a significant number of mutations.

In future efforts, special attention should be given to these two classes to further strengthen the quality of their tests and address the existing gaps.

# 8 SECURITY ANALYSIS USING OWASP FINDSECBUGS AND OWASP DC.

## FindSecBugs

All compiled classes of the project were analyzed with SpotBugs version 4.5.3 using the following command:

```
find-sec-bugs-version-1.13.0\cli\findsecbugs.sh
     -progress -html
-output findSecBugsReport.html
     target\classes\org\apache\commons\dbutils
```

## Metrics

2192 lines of code analyzed, in 51 classes, in 5 packages.

| Metric | Total | Density* |
|---|---|---|
| High Priority Warnings | | 0.00 |
| Medium Priority Warnings | 8 | 3.65 |
| **Total Warnings** | **8** | **3.65** |

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

**Figure 14: OWASP FindSecBugs results**

Eight medium-priority warnings were found, all of them with the warning type "Security Warning." One of them was about "possible information exposure through an error message" in the `DbUtils` class, while the other seven warnings were about possible weak points for SQL injections in both the `QueryRunner` and `AbstractQueryRunner` classes.

**Security Warnings**

| Code | Warning |
|---|---|
| ERRMSG | Possible information exposure through an error message |
| SECSQLIJDBC | This use of java/sql/Connection.prepareCall([Ljava/lang/String;)Ljava/sql/CallableStatement; can be vulnerable to SQL injection (with JDBC) |
| SECSQLIJDBC | This use of java/sql/Connection.prepareStatement([Ljava/lang/String;)Ljava/sql/PreparedStatement; can be vulnerable to SQL injection (with JDBC) |
| SECSQLIJDBC | This use of java/sql/Connection.prepareStatement([Ljava/lang/String;I)Ljava/sql/PreparedStatement; can be vulnerable to SQL injection (with JDBC) |
| SECSQLIJDBC | This use of java/sql/Connection.prepareStatement([Ljava/lang/String;[I)Ljava/sql/PreparedStatement; can be vulnerable to SQL injection (with JDBC) |
| SECSQLIJDBC | This use of java/sql/Statement.executeUpdate(Ljava/lang/String;)I can be vulnerable to SQL injection (with JDBC) |
| SECSQLIJDBC | This use of java/sql/Statement.executeQuery(Ljava/lang/String;)Ljava/sql/ResultSet; can be vulnerable to SQL injection (with JDBC) |
| SECSQLIJDBC | This use of java/sql/Statement.executeUpdate(Ljava/lang/String;I)I can be vulnerable to SQL injection (with JDBC) |

**Figure 15: OWASP warnings list**

***(Not) Addressing the Warnings.*** The information exposure warning was not addressed, as the line which caused the problem (`next.printStackTrace(pw);`) is part of the `DbUtils` method `printStackTrace`. As the purpose of this method is to print the stack trace, the warning can be seen as a false positive. Calling the method, on the other hand, could indeed present a risk of exposing too much information to the user, so it should be used with care.

In the `AbstractQueryRunner`, the factory methods `prepareCall` and two variations of `prepareStatement` use an SQL string as a parameter to create `CallableStatement` or `PreparedStatement`. As the SQL string is not checked, it was flagged as a potential threat. The methods mainly encapsulate the eponymous methods of the `java.sql.Connection` in a functional style and do not add security checks. The SQL string should not include the parameters yet but instead fill them later with the `fillStatement` method, also declared in the `AbstractQueryRunner` class. As the `apache.commons.dbutils` project is a library to be used in other projects, constructing the SQL statement securely lies with the developers of those projects. Therefore, nothing was changed in these methods.

The same applies to the `QueryRunner`'s `insert`, `query`, and `update` methods, which are the source of the other four warnings. They use the SQL string provided as a method parameter and insert the parameters into it if they were also provided as a method parameter. The method is only insecure if the caller chooses not to use the `params` parameter and constructs the SQL string beforehand in an insecure way.

## Dependency Check

With the given plugin for the dependency-check and the command:

```
mvn dependency-check:check
```

the dependencies of the project were analyzed.


**Figure 16: Dependency check**

The dependency check found 0 vulnerabilities because the project does not have any dependencies outside of the test scope.