

Software Dependability Project Report

ANTONIO GAROFALO, Università degli Studi di Salerno, Italy

1 INTRODUCTION

This is the report for the Software Dependability class project. It consists in forking an open-source project from a remote repository, and applying software dependability practises to it. The chosen project is called **Apache commons CLI**, an API used to handle command line operations, and the forked repository link is <https://github.com/Huntonion/commons-cli>. The list of operations performed is the following:

- Building the project
- Software Quality Analysis
- Containerization
- Code coverage Analysis
- Mutation testing campaign
- Energy greediness analysis
- Test case Generation
- Security Analysis

2 BUILDING THE PROJECT

2.1 Local build

First of all, in order start working on the project, a fork was necessary. This can be done by simply pressing a button on github. Then, the remote repository was cloned:

```
1 git clone https://github.com/Huntonion/commons-cli.git
```

This created a local repository on which it would be possible to work. The project makes use of Maven, a software project management and comprehension tool, used to manage build, report, documentation and more importantly dependencies of a project[4]. Finally, building the project is as simple as opening a terminal window into the project folder and running the command `mvn package`.

A **BUILD SUCCESSFUL** message is returned, which means that a new folder called *target* was created, in which the build artifact is located. Information about the artifact produced can be specified in the *pom.xml* file, as well as information about dependencies, plugins and the process of building itself.

2.2 Building in CI/CD

Continuous delivery is a software development methodology where some aspects of the release process are automated: such as building, testing, and deployment to production. Continuous integration consists into pushing development branches changes into production as often as possible. There are various tools that can assist with this process, such as **Jenkins**, an open source CI/CD server.

In order to run Jenkins, a docker container was used. The details about containers won't be specified here, but to put it in simple words it consists in a unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another[3]. Now, as Jenkins will need to run docker container within its container, it is necessary to run a `docker:dind` container. **Dind** stands for Docker in Docker, and it's often used to support CI/CD pipelines. In order to run the dind container:

```

1  docker run \
2  --name jenkins-docker \
3  --rm \
4  --detach \
5  --privileged \
6  --network jenkins \
7  --network-alias docker \
8  --env DOCKER_TLS_CERTDIR=/certs \
9  --volume jenkins-docker-certs:/certs/client \
10 --volume jenkins-data:/var/jenkins_home \
11 --publish 2376:2376 \
12 --publish 3000:3000 --publish 5000:5000 \
13 docker:dind \
14 --storage-driver overlay2

```

At this point the actual Jenkins container needs to run. In order to specify configuration preferences, a Dockerfile was created. A Dockerfile contains instructions to build Docker images, which can then be run and will effectively become running containers. The Dockerfile used was copied from the Jenkins documentation.

```

1 FROM jenkins/jenkins:2.387.3
2 USER root
3 RUN apt-get update && apt-get install -y lsb-release
4 RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \
5     https://download.docker.com/linux/debian/gpg
6 RUN echo "deb [arch=$(dpkg --print-architecture) \
7     signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \
8     https://download.docker.com/linux/debian \
9     $(lsb_release -cs) stable" > /etc/apt/sources.list.d/docker.list
10 RUN apt-get update && apt-get install -y docker-ce-cli
11 USER jenkins
12 RUN jenkins-plugin-cli --plugins "blueocean docker-workflow"

```

Now, in order to run the image we just created:

```

1  docker run \
2  --name jenkins-blueocean \
3  --detach \
4  --network jenkins \
5  --env DOCKER_HOST=tcp://docker:2376 \
6  --env DOCKER_CERT_PATH=/certs/client \
7  --env DOCKER_TLS_VERIFY=1 \
8  --publish 8080:8080 \
9  --publish 50000:50000 \
10 --volume jenkins-data:/var/jenkins_home \
11 --volume jenkins-docker-certs:/certs/client:ro \
12 --volume "$HOME":/home \
13 --restart=on-failure \
14 --env JAVA_OPTS="-Dhudson.plugins.git.\
15 GitSCM.ALLOW_LOCAL_CHECKOUT=true" \
16 myjenkins-blueocean:2.387.3-1

```

Jenkins is now running on localhost:8080.

The idea here is to apply a working CI/CD pipeline: the correct pipeline changes from project to project, but for now a simple one to just build the project will do. Also, the pipeline can be triggered by different events,

but the most common way is to assign a webhook to a remote repository which will trigger the pipeline when needed (mainly on pull requests). This project was worked on by a single person, so in this case it works like this: changes are made on a development environment, which then go through the CI/CD pipeline (building, testing, deployment), so that eventually changes may be merged to production. To do so a new branch was first created, using `git checkout -b dev`. This creates a branch called dev and switches to it.

2.2.1 Jenkins pipeline. A Jenkins pipeline is none other than a file describing the steps of the pipeline itself. For the purpose building the project, a very simple pipeline was created:

```

1  pipeline {
2    agent {
3      docker {
4        image 'maven:3.9.0-eclipse-temurin-11'
5        args '-v /root/.m2:/root/.m2'
6      }
7    }
8    stages {
9      stage('Build') {
10       steps {
11         sh 'mvn -B -DskipTests
12           -Drat.skip=true clean package'
13       }
14     }
15   }
16 }
```

Here two important things are defined: **Agents** and **Stages**.

Agents are executables, residing on a node, that are tasked by the controller (the Jenkins container itself) to run a job, which is described by the stages. The stages are already mentioned, they consists of steps describing the CI/CD pipeline. After pushing the Jenkinsfile into the remote repository, it's time to build!

As shown in Figure 1, the build was successful. The first build presented some maven errors, but then they were fixed.

Changes and improvements to the pipeline will be made during the development of this project, such as adding testing and deployment functionalities.

3 SOFTWARE QUALITY ANALYSIS WITH SONARCLOUD

SonarQube is a Code Quality Assurance tool that collects and analyzes source code, and provides reports for the code quality of your project. It combines static and dynamic analysis tools and enables quality to be measured continually over time [2]. SonarCloud provides most SonarQube functionalities through a SaaS web application. The initial setup for Sonarcloud is very straight-forward, it consists of just selecting the remote repository and it will start performing the first analysis. This is called **automatic analysis**, and it's good for general use. However, since a CI/CD pipeline is used for the project, we might just aswell setup a **build based analysis**, which will trigger the analysis everytime the project is built. To do so the following changes need to be applied in the *pom.xml* file:

```

1 <properties>
2   <sonar.organization>huntonion</sonar.organization>
3   <sonar.host.url>https://sonarcloud.io</sonar.host.url>
4 </properties>
```

Now, to trigger the analysis:

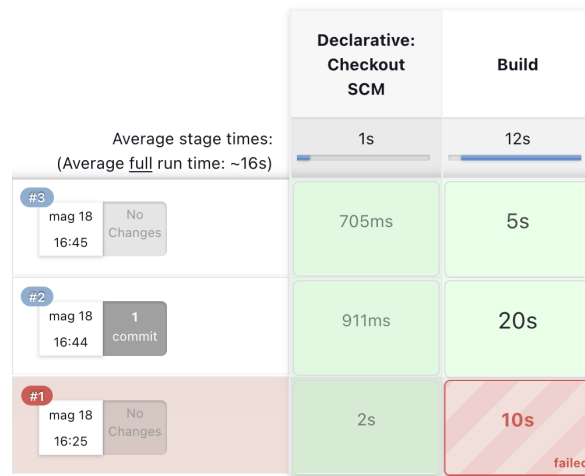


Fig. 1. Jenkins builds

```

1 mvn verify org.sonarsource.scanner.maven:\
2 sonar-maven-plugin:sonar \
3 -Dsonar.projectKey=Huntonion_commons-cli \
4 -Dsonar.branch.name=dev \
5 -Dsonar.branch.target=dev \

```

Beware that this only works by setting up an environment variable called **SONAR TOKEN**, its value will be omitted in this report for security reasons. By applying the same change in the *Jenkinsfile*, the sonarcloud analysis can also be triggered from the Jenkins pipeline. Some considerations about how Sonarcloud handles branches need to be made:

- **long lived branches** are branches that play a continuous role in the development of a project. For instance, the master branch and the develop branch are considered long lived branches.
- **short lived branches** are intended to exist temporarily.

Since the project was developed mainly working on the develop branch, branch analysis has been setup, otherwise Sonarcloud would analyze the master branch by default. To do so it is necessary to specify a regex, which in this case is: `(?:^|\W)master(?:$|\W)|(?:^|\W)dev(?:$|\W)`.

This will allow sonar to operate on both branches master and dev, by selecting it in the maven option mentioned above `-Dsonar.branch.target=dev`.

3.1 Analysis results

As shown in picture 2, the results of the quality analysis done using Sonarcloud are:

- 0 Bugs;
- 181 Code smells;
- 0 Vulnerabilities;
- 0 Security Hotspots;
- 0.0% Coverage;
- 0.7% Duplications;

The reason as to why the Coverage is 0.0% is that JaCoCo, or Java Code Coverage, was not set up at the moment of the analysis. Code coverage will be explained in better detail in the next sections. The other concerning aspect of this analysis are the **code smells**, they are none other than bad programming practices.

3.2 Code smells analysis and refactoring

The code smells can be splitted into different categories: Blocker, Critical, Major, Minor, and Info. Each category will be reported more in detail and, if possible, refactored in order to remove the code smell.

3.2.1 Blockers (60). Blockers are smells that need fix as soon as possible since they might impact the behaviour of the application in production. Here we have 60 Blockers, but upon further inspection they appear to be **false positives**, because all of them refer to the missing unit tests for two classes *BasicParserTest* and *GnuParserTest*: in both cases they are deprecated, and every need for a parser was substituted by a new class called *DefaultParser*. In fact, every test of these classes has a `@ignore` annotation, which means that no test is executed at all.

3.2.2 Critical (9). Very similar to blockers, just less severe but still need to be fixed as soon as possible. The reason for 7 out of these 9 is **Cognitive Complexity**, which is a measure of how difficult a unit of code is to intuitively understand [1]. I must admit that it is in fact truly difficult to understand, and as a simple contributor to the project I decided to leave the fix to whoever worked on these functions to begin with. The remaning two consist in adding a default statement to a switch case and making a list Serializable. The former has been fixed by

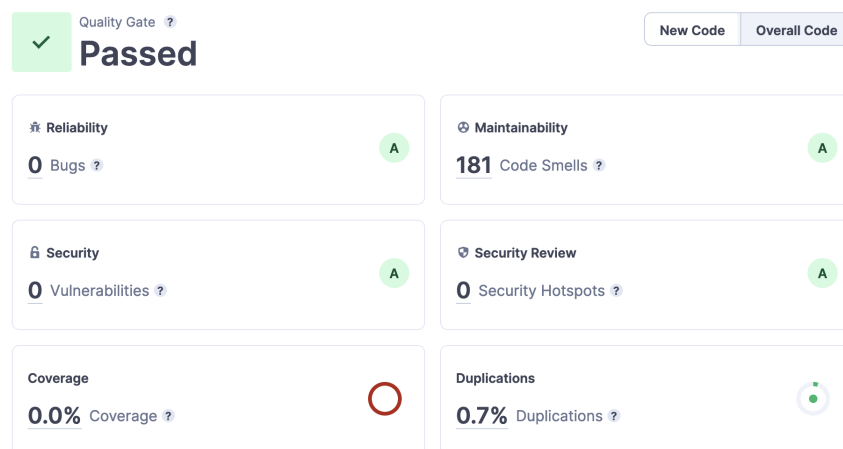


Fig. 2. Quality analysis results

simply adding a `return null` to the default statement of the switch case, the latter is a false positive since when trying to implement the `Serializable` interface it appears to be redundant as it is already defined by `Throwable`.

3.2.3 Major (68). Major smells are flaws which can highly impact the developer productivity. The great majority of these code smells are because of `assertEquals` methods. Apparently the values needed to be swapped, but upon further investigation it was noticed that they are in fact in the correct order of expected value and actual value, therefore these have been marked as false positives. Some other were related to multiple test parameters, but they also have been marked as false positives, because in reality the tests operate on the `args[]` variable, which is passed from command line, and then combined with another object, therefore making it effectively a single parameter.

3.2.4 Minor and Info (44). Minor smells can impact the productivity slightly, but they still need to be fixed, while info are just essentially reminders. Both of these categories were put together because all of the Minor smells found by the Sonarcloud analysis and the related info are about the usage of deprecated components. Unfortunately these are needed for the correct execution of the application, so they were not fixed.

4 CONTAINERIZATION

Containers were already briefly introduced in section 2. With the word "containerization" essentially we refer to the process of packaging a software into an image, in order to be run as a container. However, the project this work is based on consists of a Java API which is usually implemented through maven by adding the commons-cli dependency in the `pom.xml`, so in order to interact with it a little, a main class called `ParseTest.java` was created. On this class there are specified two command line options, `-h` to print "Hello World!", and `-t` to print the current time. Also, a change to the `pom.xml` file was necessary in order to set the main class for the application:

```

1 <plugin>
2   <!-- Build an executable JAR -->
3   <groupId>org.apache.maven.plugins</groupId>
4   <artifactId>maven-jar-plugin</artifactId>
5   <version>3.1.0</version>
6   <configuration>
7     <archive>
8       <manifest>
9         <addClasspath>true</addClasspath>
10        <classpathPrefix>lib</classpathPrefix>
11        <mainClass>org.apache.commons
12        .cli.ParseTest</mainClass>
13      </manifest>
14    </archive>
15  </configuration>
16 </plugin>

```

At this point a Dockerfile was created:

```

1 FROM openjdk:8
2 ADD target/commons-cli-1.6-SNAPSHOT.jar \
3 commons-cli.jar
4 ENTRYPOINT [ "java", "-jar","commons-cli.jar"]
5 CMD ["-h"]
6 EXPOSE 8080

```

This is a fairly simple Dockerfile, explaining it line by line:

- **FROM:** Docker images are built through "layers", starting from heavier layers that can be for instance linux virtual machines, so FROM pulls an image from Dockerhub and uses it as a layer on top of which the application is built. In this case openjdk 8 is used.
- **ADD:** Adds the jar executable in the image.
- **ENTRYPOINT & CMD:** Specifies the command line instruction to run when initially running the container through *Docker run*. The difference between Entrypoint and CMD is that CMD can be overwritten by user input, while Entrypoint cannot. In this case the -h will print "hello world", like specified in the *ParseTest.java* class.
- **EXPOSE:** Exposes a port in the container to communicate with external applications. Conventionally the same port is mapped, for instance -p 8080:8080 means that the port 8080 of the host machine is being mapped to the port 8080 of the running container.

docker run --it --rm commonscli.jar was run to get the image running, and a "Hello World!!" message was printed. As shown in picture 3 a use of the CMD option overwriting is reported, printing the current time.

```

Apple ~ /Doc/U/sd/commons-cli git dev > docker run -it --rm commonscli.jar
Hello World!!

Apple ~ /Doc/U/sd/commons-cli git dev > docker run -it --rm commonscli.jar -t
06/06/2023 18:19:27

Apple ~ /Doc/U/sd/commons-cli git dev >
  
```

Fig. 3. Docker run command. -it is used to attach an interactive terminal, -rm to delete the container after its execution

4.1 Adding the push to Dockerhub functionality to the CI/CD pipeline in Jenkins

The next step was to automate this process, in order to push an up-to-date docker image of the application on dockerhub whenever the build is successfully completed. First of all a repository on Dockerhub called *Huntonion/commons-cli* was created, then some important changes have been applied to the Jenkinsfile pipeline.

```

1  pipeline {
2  agent none
3  options {
4      buildDiscarder(logRotator(numToKeepStr: '5'))
5  }
6  environment{
7      DOCKERHUB_CREDENTIALS = credentials('docker');
8  }
9  stages {
10     stage('Build') {
11         agent {
12             docker {
13                 image 'maven:3.9.0-eclipse-temurin-11'
14                 args '-v /root/.m2:/root/.m2'
15             }
16         }
17         steps {
18             sh 'mvn -B -Drat.skip=true -DskipTests verify'
19         }
20     }
21 }
  
```

```

1      stage('Test') {
2          agent{
3      docker {
4          image 'maven:3.9.0-eclipse-temurin-11'
5          args '-v /root/.m2:/root/.m2'
6      }
7      }
8      steps {
9          sh 'mvn -Drat.skip=true test org.sonarsource.scanner.
10             maven:sonar-maven-plugin:sonar -Dsonar.projectKey=Huntonion_commons-cli
11             -Dsonar.branch.name=dev -Dsonar.branch.target=dev'
12      }
13      post {
14          always {
15              junit 'target/surefire-reports/*.xml'
16          }
17      }
18  }
19  stage('Build docker image') {
20      agent any
21      steps{
22          sh 'docker build -t huntonion/commons-cli .'
23      }
24  }
25  stage('Login Dockerhub'){
26      agent any
27      steps{
28          sh 'echo $DOCKERHUB_CREDENTIALS_PSW |
29             docker login -u $DOCKERHUB_CREDENTIALS_USR --password-stdin'
30      }
31  }
32  stage('Push'){
33      agent any
34      steps{
35          sh 'docker push huntonion/commons-cli'
36          sh 'docker logout'
37      }
38  }
39  }
40  }

```

The first thing that catches the eye is the "agent none" instruction. It had to be changed to this because the previous agent was a docker container running maven. If I tried to execute docker instructions in the pipeline such as `docker build -t huntonion/commons-cli .`, it would return "docker not found" error because it essentially was trying to run the instruction in the maven docker container. Therefore, different agents have been used. The new ones referred to as "agent any" mean that Jenkins itself will perform that operation, and not an external agent like with the maven docker container. The different steps speak for themselves, one is to build the docker image, one to log in to Dockerhub and the last one to push the image. I'd like to add that, beforehand, the Dockerhub credentials were setup in the Jenkins configuration. There's also present a Test stage but I will cover that in the next section. In order to automatically trigger the build, a github hooks needs to be setup. Unfortunately Jenkins is running locally on my machine, therefore accessing it from github it's impossible. As a workaround,

SCM polling was set up, it consists in having Jenkins scan the remote repository for changes in a specified time interval, which for the project was set to 1 minute.

5 CODE COVERAGE ANALYSIS

To calculate code coverage JaCoCo was used. JaCoCo (Java Code Coverage) is a library for java used to calculate, well, code coverage. Using it is as simple as adding some lines in the *pom.xml*:

```

1      <plugin>
2      <groupId>org.jacoco</groupId>
3      <artifactId>jacoco-maven-plugin</artifactId>
4      <executions>
5          <execution>
6              <id>report</id>
7              <goals>
8                  <goal>report-aggregate</goal>
9              </goals>
10             <phase>verify</phase>
11         </execution>
12     </executions>
13 </plugin>

```

But also a new line on the Jenkinsfile, in the "Test" step.

```

1 sh 'mvn -Drat.skip=true test org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
2 -Dsonar.projectKey=Huntonion_commons-cli -Dsonar.branch.name=dev -Dsonar.branch.target=dev'

```

JaCoCo will write informations related to the code coverage on some files, of these one is named *jacoco.xml*, and this file will then be used by sonarcloud to report the percentage, just like shown in picture 4. A code coverage of 94.4% was found.

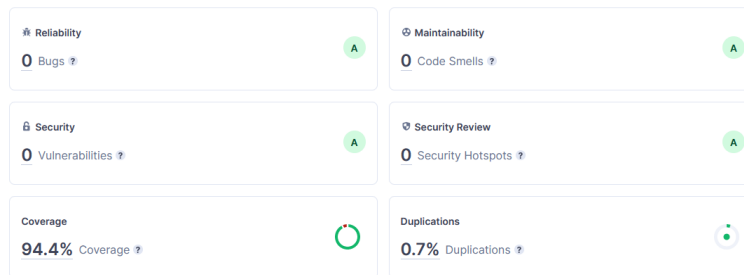


Fig. 4. Sonarcloud scan results

6 MUTATION TESTING CAMPAIGN

Now that code coverage has been calculated we have an idea on the overall quality level of the test suite for this project. But how are we so sure that the unit tests implemented are exhaustively testing the classes? Another practice that comes in handy for this task is called *mutation testing*. It works by applying changes, called *mutations* to the code in order to make the tests fail. Basically we say that the mutant was **killed** if it affected the behaviour of the test, or **survived** if it didn't, just like shown in picture 5.

A tool called PiTest to implement mutation testing to the project was used. First the maven plugin and dependencies were added:

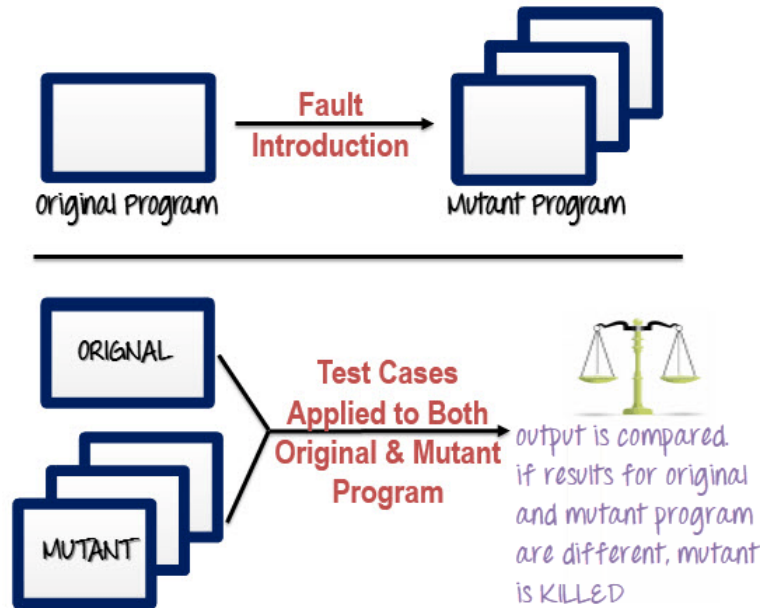


Fig. 5. Mutation Testing

```

1  <plugin>
2    <groupId>org.pitest</groupId>
3    <artifactId>pitest-maven</artifactId>
4    <version>1.10.3</version>
5    <dependencies>
6      <dependency>
7        <groupId>org.pitest</groupId>
8        <artifactId>pitest-junit5-plugin</artifactId>
9        <version>1.1.1</version>
10     </dependency>
11   </dependencies>
12 </plugin>

```

And then, in order to run PiTest,

```

1  mvn org.pitest:pitest-maven:mutationCoverage

```

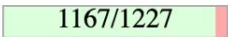

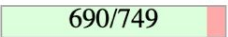
This maven goal created a folder in the target directory called *pit-reports*, and in it some interesting files are generated, like the *index.html*, which upon opening will show the overall results of the mutation testing campaign. These scores can be seen in figure 6 and in table 1. I would say the most important value to look at is the **mutation coverage**, which stands for the number of mutations killed over the total number of mutations.

Line coverage	Mutation coverage	Test strength
95%	90%	92%

Table 1. Mutation testing results.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
21	95% 	90% 	92% 

Breakdown by Package

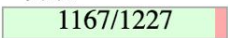
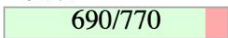
Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.apache.commons.cli	21	95% 	90% 	92% 

Fig. 6. Mutation testing metrics

By looking further into the *pit-reports* folder, it's possible to singularly analyze the classes in order to understand which test case needs to be fixed, for instance in figure 7, the class *DefaultParser* is analyzed and the PiTest report clearly states which kind of **syntactic change** or **mutant operator** is applied. Unfortunately, mutation testing is a very resource-intensive task to be performed so in a real-case scenario testing the whole project is pretty unrealistic, therefore some paths or classes to be mutated can be specified in the *pom.xml* as needed.

```

70  1. replaced return value with null for org/apache/commons/cli/DefaultParser$Builder::build → KILLED
97  1. replaced return value with null for org/apache/commons/cli/DefaultParser$Builder::setAllowPartialMatching → KILLED
117 1. replaced return value with null for org/apache/commons/cli/DefaultParser$Builder::setStripLeadingAndTrailingQuotes → KILLED
129 1. replaced return value with null for org/apache/commons/cli/DefaultParser::builder → KILLED
230 1. negated conditional → KILLED
    2. negated conditional → KILLED
242 1. negated conditional → KILLED
    1. changed conditional boundary → SURVIVED
257 2. Replaced integer subtraction with addition → KILLED
    3. negated conditional → KILLED
259 1. negated conditional → KILLED
265 1. replaced return value with "" for org/apache/commons/cli/DefaultParser::getLongPrefix → KILLED
275 1. negated conditional → KILLED
276 1. replaced return value with Collections.emptyList for org/apache/commons/cli/DefaultParser::getMatchingLongOptions → KILLED
279 1. negated conditional → KILLED
284 1. replaced return value with Collections.emptyList for org/apache/commons/cli/DefaultParser::getMatchingLongOptions → KILLED

```

Fig. 7. DefaultParser

7 ENERGY GREEDINESS ANALYSIS

This project report will also cover some energy greediness analysis. The only reason this could be useful for projects like this is to reduce the environmental footprint the application causes, but in different environments like for instance an IoT network, one of the main concerns is the battery consumption of a device, so this tool called **eco-code** might be very useful in such scenarios. Unfortunately Sonarcloud does not support plugins so we have to recur to Sonarqube to install the eco-code plugin. Sonarqube was installed through a docker container:

```

1  docker run -d --name sonarqube \
2  -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true \
3  -p 9000:9000 sonarqube:latest

```

! Blocker	0	Minor	182
↑ Critical	0	Info	0
↗ Major	0		

Fig. 8. Results of the eco-code analysis

Then, by accessing `localhost:9000` the sonarqube interface allows us to select what kind of project needs to be analyzed (in this case a maven project). But first the eco-code plugin needs to be installed. This plugin essentially defines a set of rules to be applied during code analysis, which are specified in what's called a **Quality profile**. A new Quality profile was created for this project, adding as the only rules the ones provided by the eco-code plugin. This was done because the overall analysis of the project was already performed on Sonarcloud, so for this part the only interest was on the energy greediness. Then, to trigger the analysis:

```

1 mvn clean verify sonar:sonar \
2 -Dsonar.projectKey=commons-cli \
3 -Dsonar.projectName='commons-cli' \
4 -Dsonar.host.url=http://localhost:9000 \
5 -Dsonar.token=sqp_56946ef307f43e5e48de0420d2c98e0904402740

```

The analysis resulted in **182 code smells**, luckily all of them of **minor severity**, as can be seen on picture 7. The great majority of them could be fixed by applying one of the following suggestions, depending on the code smell:

- Using a switch statement instead of multiple if-else if possible
- Use `++i` instead of `i++`
- Avoid using global variables

Upon further analysis of the codebase, it was decided to leave it as it is for the following reasons:

- Incrementing a variable before (`++i`) or after (`i++`) its assignment is not necessarily the same. It can sometimes affect the application in unpredictable ways.
- the If statements the analysis refers to are actually either single Ifs, or multiple Ifs but with different parameters, therefore changing them with a switch statement seems unpractical, if not impossible.
- The global variables are none other than instance variables for the classes, so this has been categorized as a false positive.
- They were all of minor severity.

8 TEST CASE GENERATION

Unfortunately more than often there might be classes not properly tested in an application. **Randoop** is a tool used to generate unit tests pseudo-randomly, by generating sequences of methods/constructor invocations for the classes under test. Randoop requires Junit and Hamcrest, Junit was already present in the *pom.xml*, so let's just add Hamcrest:

```

1  <dependency>
2    <groupId>org.hamcrest</groupId>
3    <artifactId>hamcrest</artifactId>
4    <version>2.2</version>
5    <scope>test</scope>
6  </dependency>
7  <dependency>
8    <groupId>org.hamcrest</groupId>
9    <artifactId>hamcrest-core</artifactId>
10   <version>2.2</version>
11   <scope>test</scope>
12 </dependency>

```

Then, let's copy the dependencies in the *target/dependency* folder:

```

1  mvn dependency:copy-dependencies -DincludeArtifactIds=junit,junit-jupiter, \
2  hamcrest,hamcrest-core

```

At this point everything is ready to run Randoop, so the following instructions were ran:

```

1  jdeps -apionly -v -R -cp commons-cli/src/main/java \
2  commons-cli/src/main/java/org/apache/commons/cli/ | grep -v '^[A-Za-z]' | \
3  sed -E 's/^.* -> ([^ ]+) .*$/\1/' | sort | uniq > myclasses.txt
4
5  java -cp randoop-all-4.3.2.jar:commons-cli/target/classes/ \
6  randoop.main.Main gentests --classlist=myclasses.txt --time-limit=20 \
7  --junit-output-dir=randoop-tests
8
9  javac $(find randoop-tests -name "*.java") -cp \
10 commons-cli/src/main/java/./randoop-all-4.3.2.jar:commons-cli/ \
11 target/dependency/junit-jupiter-5.9.1.jar:commons-cli/target/ \
12 dependency/hamcrest-2.2.jar
13
14 java -cp randoop-tests:commons-cli/src/main/java:./randoop-all-4.3.2.jar:./ \
15 junit-platform-console-standalone-1.9.2.jar \
16 org.junit.platform.console.ConsoleLauncher --scan-class-path \

```

The explanation is:

- **Row 1 to 3:** Randoop tests every Class in the specified Classpath by default, so this is to define a text file on which only the classess interested into unit testing are listed, and then "fed" to randoop through the `-classlist` option.
- **Row 5 to 7:** Runs Randoop and generates tests. The outputs can be of two kinds, *RegressionTest.java* which are tests with regression assertions (4 files of these were generated), and *ErrorTest.java*, which are tests with failure-revealing assertions (none were generated)
- **Row 9 to 12:** Compiles the java files
- **Row 14 to 16:** Run the compiled tests on Junit

1053 tests were performed, and all of them succeeded, as shown in Figure 9.

```

Test run finished after 114 ms
[       7 containers found      ]
[       0 containers skipped    ]
[       7 containers started    ]
[       0 containers aborted    ]
[       7 containers successful ]
[       0 containers failed     ]
[    1053 tests found          ]
[         0 tests skipped      ]
[    1053 tests started        ]
[         0 tests aborted      ]
[    1053 tests successful     ]
[         0 tests failed       ]

```

Apple logo ~/documents/uni/sd >

Fig. 9. Randoop results

9 SECURITY ANALYSIS

In order to find potential vulnerabilities, FindSecBugs was used. It's a tool written in Java. It's meant for web applications, but can also be used for simple java applications. Using it was very simple, after downloading it, it was run by passing the commons-cli directory as an argument, and some other options. But before doing so, a "myexclude.xml" file needs to be created, to specify a filter for FindSecBugs operates with:

```

1 <FindBugsFilter>
2   <Match>
3     <Class name="~.*Tests?$/>
4     <Bug category="SECURITY" />
5   </Match>
6 </FindBugsFilter>

```

```

1 ./findsebugs.sh -progress -html -output report.html -exclude myexclude.xml commons-cli

```

After the analysis, findsebugs will output a *report.html* file (because we specified it as an option, but it can also output files in other formats such as XML, that may eventually be used in a CI pipeline, for instance on Jenkins). The results are reported in figure 10.

Apparently, the analysis found 1 critical security concern related to object deserialization. Of the remaining 20 medium priority warnings, 14 are about reading a file whose location might be specified by user input (this can be risky as it might lead some important dangers like code injection) and the rest report possible information exposure through an error message.

Metrics

22479 lines of code analyzed, in 693 classes, in 89 packages.

Metric	Total	Density*
High Priority Warnings	1	0.04
Medium Priority Warnings	20	0.89
Total Warnings	21	0.93

Fig. 10. Metrics for the FindSecBugs analysis

LIST OF FIGURES

1	Jenkins builds	4
2	Quality analysis results	5
3	Docker run command. <code>-it</code> is used to attach an interactive terminal, <code>-rm</code> to delete the container after its execution	7
4	Sonarcloud scan results	9
5	Mutation Testing	10
6	Mutation testing metrics	11
7	DefaultParser	11
8	Results of the eco-code analysis	12
9	Randoop results	14
10	Metrics for the FindSecBugs analysis	15

REFERENCES

- [1] CODECLIMATE 2023. *cognitive complexity*. <https://docs.codeclimate.com/docs/cognitive-complexity>
- [2] DEVOPSSCHOOL 2023. *Dev ops school*. <https://www.devopsschool.com/blog/what-is-sonarqube-and-how-it-works-an-overview-and-its-use-cases/>
- [3] DOCKER 2023. *Docker official website*. <https://www.docker.com/resources/what-container/>
- [4] MAVEN 2023. *Maven official website*. <https://maven.apache.org/>