**Project Description**

To make our experiment more convincing, we're choosing 5 java open source projects, in which 3 of them are greater than 100K LOC. Moreover, for each project, we choosing 3-4 different versions to collect the data. So we are able to collect the difference during the version evolution period. In addition, all of the projects that we choose has an issue-tracking system, which we used for collecting the data for maintenance relevant metrics.

*Project 1: Apache commons Lang*

Lang provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally, it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode. toString and equals[1].

It is a large open source project which has 79.8K LOC and a continuous issue records in its issue tracking system. We are using versions from 3.0 to 3.8 to collecting the data for our experiment.

*Project 2: Apache commons configuration*

The Commons Configuration software library provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources[2].

The configuration is a large apache project, which contains serval active versions as well as a continuous bug-tracking system, which list out all the issues and its detail description, solving status and timestamp. It makes our data collection work for metrics 5 easier. We’re collecting data using versions from 2.1 to 2.4.

*Project 3: Apache commons Codec*

Apache Commons Codec (TM) software provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs[3].

Commons Codec is a perfect project for us to collect the data from. The whole project is built by Maven, and it contains a lot of developer test cases. It is very convenience for us to collect the Jacoco and Pitest report since the only thing we need to do is the configuration. Meanwhile, it also contains an issue tracking system and a lot of subversions. We are using versions from 1.10 to 1.12 for the experiment.

*Project 4: Apache commons collections*

The Java Collections Framework was a major addition in JDK 1.2. It added many powerful data structures that accelerate the development of the most significant Java applications. Since that time it has become the recognized standard for collection handling in Java[4].

We are using from version 4.0 to 4.4 for our experiments in this project. The size of collections is a 132K LOC which is the ideal size of our experiments. Just like what other project does, it contains a continues issue-tracking system and build in Maven, which makes our data collecting work very convenient.

*Project 5: JFreeChart*

JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications[5].

JFreeChart is a maven project and in the size of 167K LOC, and a continuous issue-tracking system. However it doesn’t have too many versions we can collect the data from, we only analysis this project’s data from version 1.0.19 – 1.5.0.

[1] <https://commons.apache.org/proper/commons-lang/>

[2] <https://commons.apache.org/proper/commons-configuration/>

[3] <https://commons.apache.org/proper/commons-codec/>

[4] <https://commons.apache.org/proper/commons-collections/>

[5] <http://www.jfree.org/jfreechart/>

**Steps to collecting the data**

Our data collecting work can be totally divided into 6 steps:

S1.selecting projects.

S2.building projects.

S3.configuring Jacoco plugin.

S4.adding pit test plugin.

S5. selecting the active period for issues tracking and collecting related data from the issue tracking system.

S6.write shell script for change-report and collecting changes-data from different subversions.

*Step1: Selecting projects*

In order to boost our later process, we’re carefully choosing the projects which meet the following standards:

* It is an open source project which is also programmed in Java Language.
* It is ether build by Maven or by Ant.
* It should be a single module project and the size of it shouldn’t be too small.
* There is an issue-tracking system which contains continuous issue-solving records.
* There are several subversions for us to collect data.

After filtering many unqualified projects, we finally narrow down our searching scope to Apache project, since most of them are meet our standards in terms of size, programming language, issue tracking system as well as serval subversions.

Step 2: Building the projects

After selecting projects, we tried to build all of them, in order to see if there are some crucial problems or doesn’t contain any unit test cases. For those contains some small problem, such as JDK version difference, we will fix it. However, for those projects which have crucial problems or doesn’t exist any unit test cases, we will drop this project and then go back to step 1.

In conclusion, in step two, we’re validating the selecting to see whether it is suitable for our experiment.

*Step 3: adding Jacoco plugin*

In order to collect the data for statements coverage, branch coverage as well as complexity, we’re configuring for each project including its subversions that we choose to generate the Jacoco reports. As you can see in Figure 1, We adding Jacoco plugin into build file for each project(pom for maven projects) and adding Jacoco reports task into the test phase.

During the process, some of the projects show some problems such as some test cases cannot be passed, so it will prevent Jacoco to generate the report. We are using two solutions to solving this problem. First, changing the expected value for that test case so it can be passed. Second, delete this test case. Since all of the projects that we chose contains thousands of test cases, so one or two test cases won’t affect the final result. The example Jacoco report is shown in Figure 2.

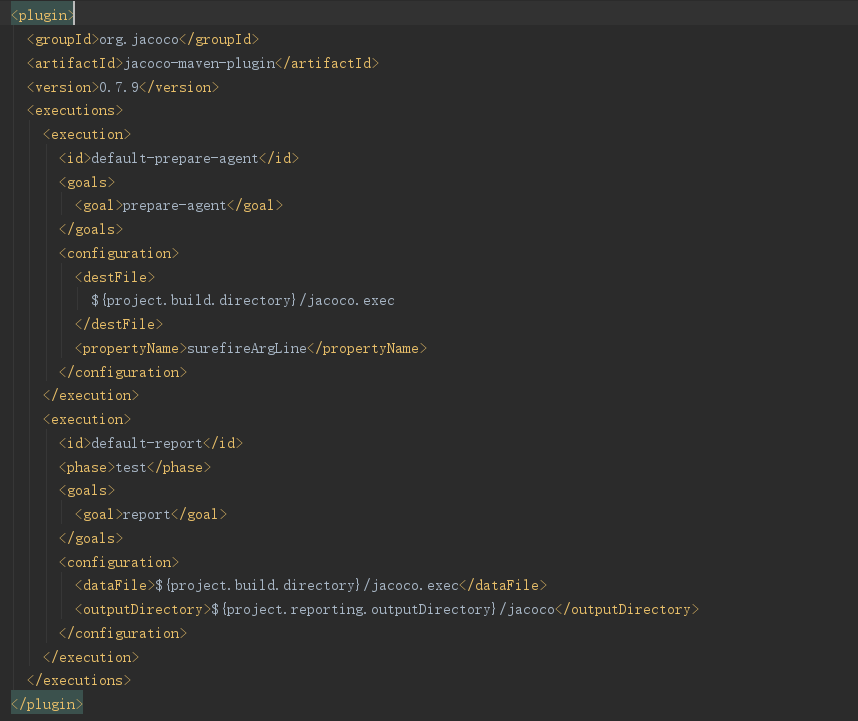


Figure 1. Jacoco configuration (Commons Lang)

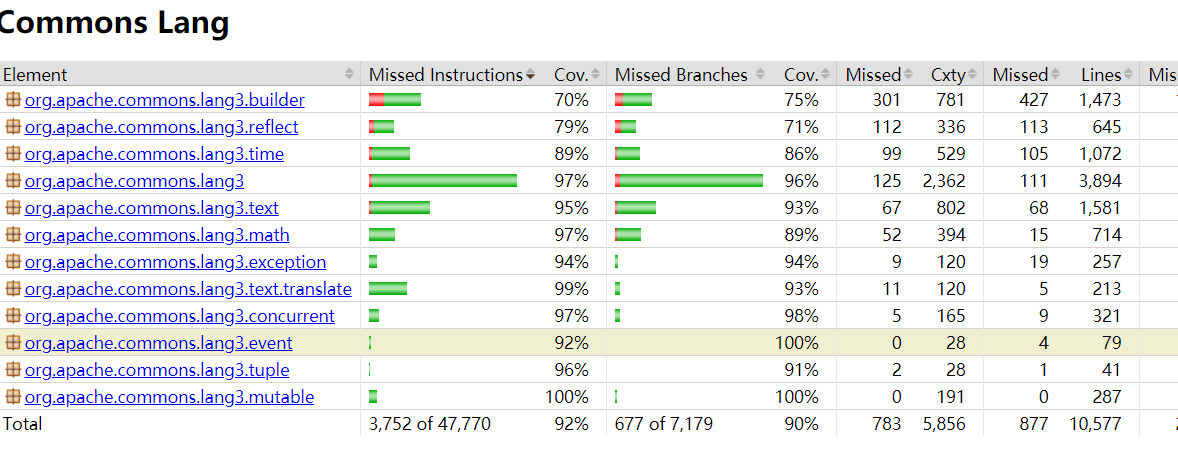


Figure 2. Jacoco report (Commons Lang)

*Step 4: adding pit plugin*

For the mutation score, we are using pit plugin to generate the report. In the configuration, it allowed us to choose the mutator, target java classes, target test cases (configuration shown in Figure 3).

We used 7 default mutators(operators) to generate the mutation which contains: Conditionals Boundary, Increments, Invert Negatives, Math, Negate Conditionals, Return Values and Void Method Calls.

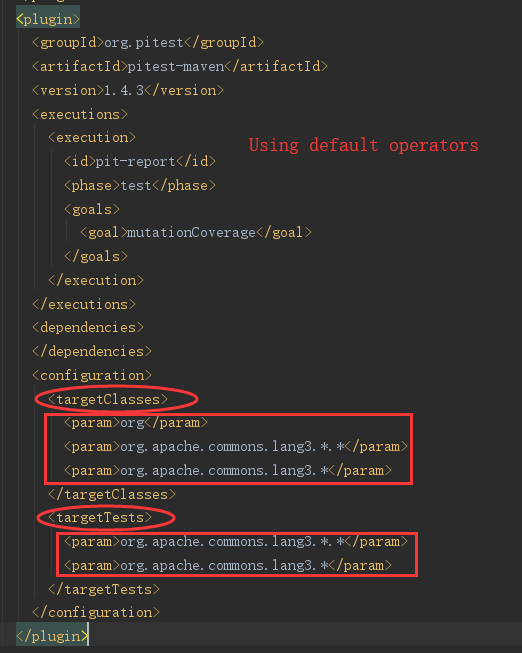


Figure 3. Pitest configuration(Commons Lang)

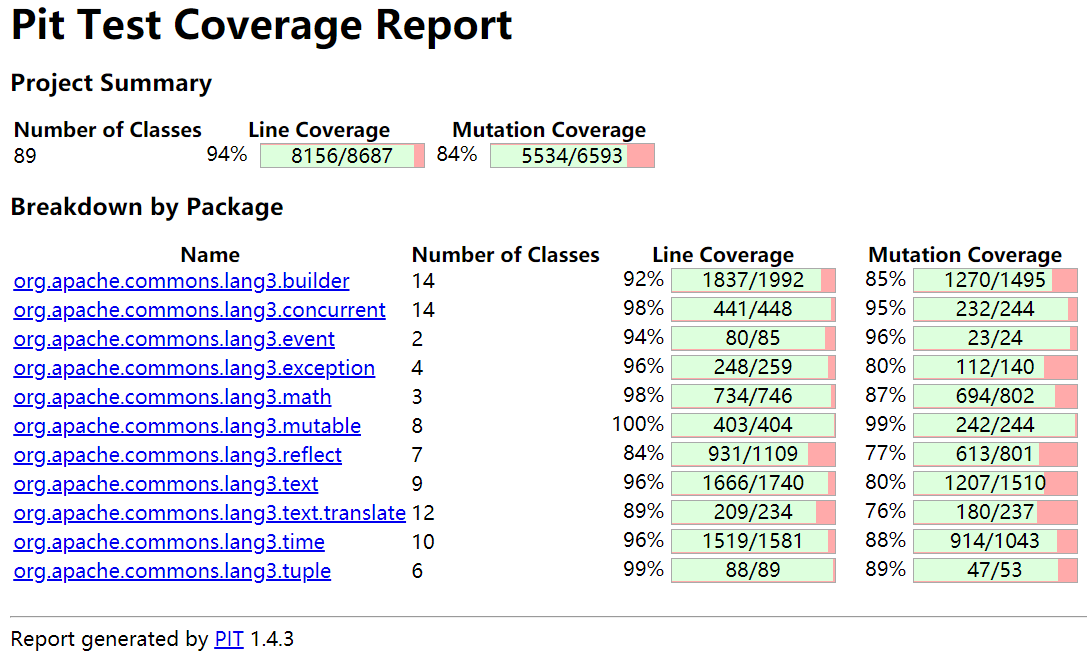


Figure 4. Pitest Report (Commons Lang)

*Step 5: selecting active month and collecting data from the issue tracking system.*

We decided to collect three active month data in each subversion and then get the average value among these three months for Fix Backlog and Backlog Management Index. Since for software, a new version comes to release doesn’t mean the previous version is out of the stage. So you cant using the total life of the projects to calculate BMI value. We carefully located the 3 active continuous months which has the largest number of issue arrivals for each subversion and then calculate the BMI based on the average value of these three months.

The issue-tracking system has basic statistics information of the issues and what we need to do is manually search the active month by SQL that it offered like Figure 5. Finally using excel to calculate all BMI values.

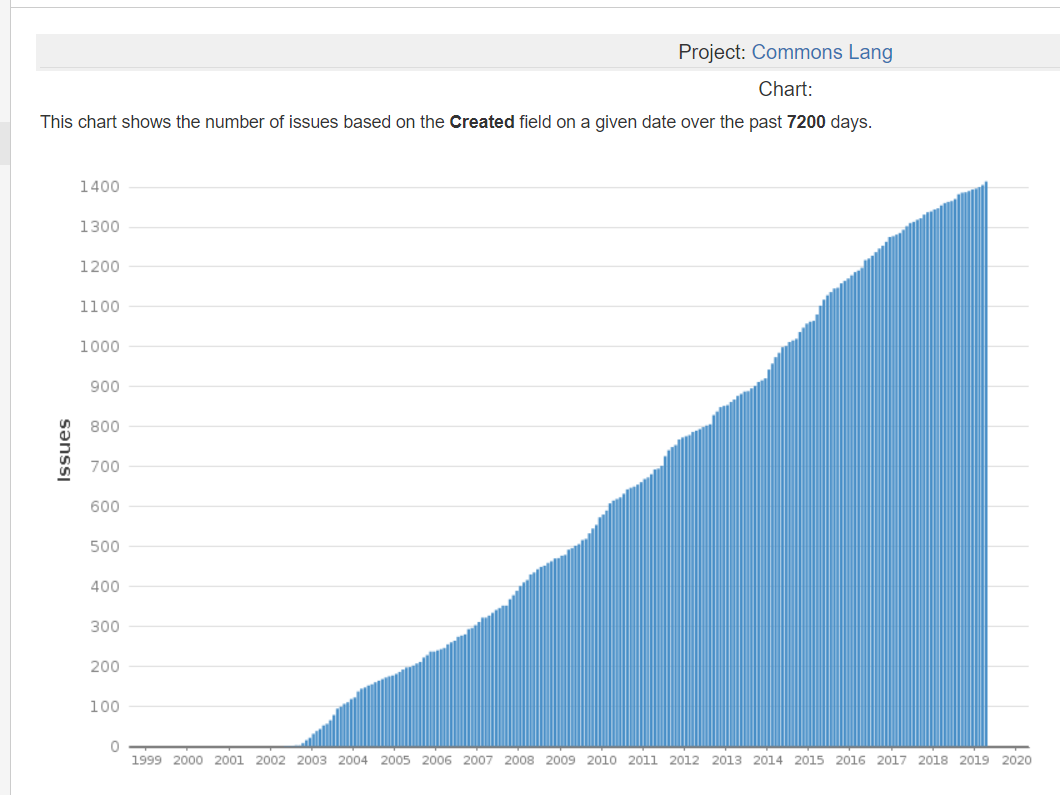


Figure 5. Active month located work (Lang)

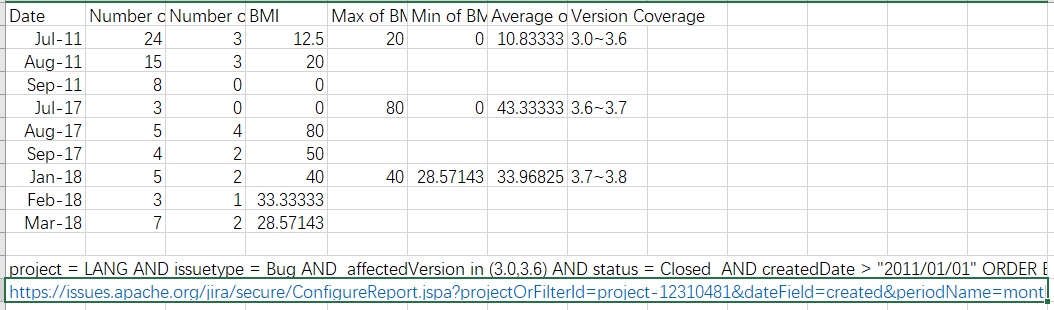
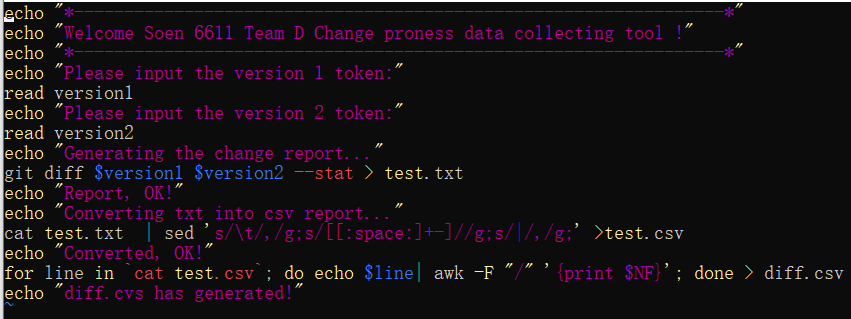


Figure 6. BMI Report (Commons Lang)

*Step 6: write script for collecting change-report and collecting changes-data from different subversions.*

We’re collecting data mainly for change proneness in this part. All of our collection is based on git log, which will compare two different submits or two different timestamps, and then give us the number of change lines for each file. To make the whole process more smooth, we write a script in shell, which will get the log file base on the input of 2 subversion’s token, then do the calculation and convert it into the form what we want, and finally, convert it to a CSV file, so it will be easy for our later analysis work. The script shows in Figure 7.

After we got the report CSV file, we remove those file which is not Java, since we want only compare the changed line in each class and the total number of changes. What’s more, we also delete all the test case file, because we also need to analyze the relation with code coverage, the Jacoco report won’t generate the report for the test case. Our final report for Change proneness shown in Figure 8.



*Figure 7. Change report script*