BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

COMPARATOR OF TEST RESULTS
KOMPARÁTOR VÝSLEDKŮ TESTŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PATRIK ČERBÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156966

Institut:	Department of Intelligent Systems (DITS)
Student:	**Čerbák Patrik**
Programme:	Information Technology
Title:	**Comparator of Test Results**
Category:	Software analysis and testing
Academic year:	2023/24

Assignment:

1. Learn about unit testing, integration testing, and Jenkins CI. Get familiar with Jenkins plugins for processing test results and their issues over huge datasets. Get familiar with the testing of OpenJDK in Red Hat Czech.
2. Design a tool which would be able to compare results among various configurations and different runtime environments. The tool should effectively identify different test results.
3. Implement such a tool as a CLI or web-based application with a focus on direct usage from the Jenkins environment.
4. Evaluate the tool on the results of testing OpenJDK in Red Hat Czech and Eclipse Adoptium organization.

Literature:

- V. Armenise. Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, Florence, Italy, 2015, pp. 24-27, doi: 10.1109/RELENG.2015.19.
- Parsons, D. Unit Testing with JUnit. In: Foundational Java. Texts in Computer Science. Springer, Cham. 2020. https://doi.org/10.1007/978-3-030-54518-5_10

Requirements for the semestral defence:
The first two steps of the assignment

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Smrčka Aleš, Ing., Ph.D.**
Consultant:	Jiří Vaněk
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	29.11.2023

Abstract

This bachelor's thesis deals with the topic of OpenJDK testing in Red Hat and how to make this testing more efficient. One of the main problems when testing is comparing whether a particular test failed on only one variant of a test suite or on multiple variants (one test suite with the same tests has many variants – they differ, for example, in operating systems, runtime architecture, etc.). Because of this, in this thesis, a tool is designed and implemented to compare test results on different variants of a test suite. This tool is implemented in the Java programming language and can run standalone as a CLI program or in a dedicated web environment. This comparator is also part of a so-called Jtreg plugin, which is an open source plugin to Jenkins that is used by the OpenJDK QA team at Red Hat. The tool itself allows you to do a lot of things, such as filtering test suites for comparison, creating a failed tests comparison table, a failed tests stack trace similarity table, or a table comparing the build result of individual test suites.

Abstrakt

Tato bakalářská práce se zaobírá tématem testování OpenJDK ve firmě Red Hat a jak toto testování zefektivnit. Jeden z hlavních problémů při testování je porovnání, zda určitý test spadl pouze na jedné variantě testovací sady nebo na více variantách (jedna testovací sada se stejnými testy má mnoho variant – liší se například v operačních systémech, běhové architektuře, atd.). Kvůli tomuto je v této práci navrhnut a implementován nástroj pro porovnávání výsledků testů právě na odlišných varintách jedné testovací sady. Tento nástroj je implementován v programovacím jazyce Java a může běžet samostatně jako CLI program nebo ve speciálním webovém prostředí. Tento komparátor je zároveň součástí takzvaného Jtreg pluginu, což je open source rozšíření do nástroje Jenkins, které se používá v OpenJDK QA týmu firmy Red Hat. Samotný nástroj umožňuje spoustu věcí, například filtrování testovacích sad pro porovnání, vytvoření porovnávací tabulky spadlých testů, tabulky s podobností stack trace spadlých testů nebo třeba tabulky porovnávající výsledek sestavení jednotlivých testovacích sad.

Keywords

comparator, OpenJDK, testing, Red Hat, quality assurance, Java, Jenkins, unit tests, JUnit, Jtreg

Klíčová slova

komparátor, OpenJDK, testování, Red Hat, quality assurance, Java, Jenkins, jednotkové testy, JUnit, Jtreg

Reference

ČERBÁK, Patrik. *Comparator of Test Results.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Během psaní této bakalářské práce jsem byl zaměstnán jako stážista ve firmě Red Hat Czech, konkrétně v OpenJDK QA týmu. Jelikož OpenJDK je velký projekt, jeho testování je složitá záležitost. Existuje spoustu testovacích sad (například sady s jednotkovými testy OpenJDK, sady s integračními testy, sady testující správnost implentace Javy, benchmarky, atd.) a každá z nich má více různých variant – jako třeba varianty jedné testovací sady na více operačních systémech, více architekturách procesoru, více nastavení OpenJDK a jiné. Každá jednotlivá testovací sada obsahuje až statisíce různých jednotlivých testů, které jsou během testování spuštěny.

Jedním z hlavních problémů během testovaní je tedy právě zjistění, zda určitý test, který spadnul, spadnul pouze na jedné variantě testovací sady nebo na více variantách, případně kterých (může jít třeba o test, který padá pouze na určité verzi operačního systému, ne jinde) a jestli na všech těchto variantách spadnul ze stejného důvodu. Tento problém byl dříve řešen zejména manuálně – bylo třeba ručně najít a zkontrolovat výsledky právě na rozdílných variantách.

Právě na vyřešení tohoto problému byla vytvořena tato práce – *Komparátor výsledků testů*. Tento komparátor je naprogramovaný v jazyce Java a je dostupný jako samostatný program spustitelný přes terminál, ale také k němu existuje speciální webové rozhraní přes které jde spoustět. Celý tento projekt je open source a je součástí speciálního *Jtreg pluginu* do nástroje Jenkins (nástroj na automatizaci sestavování a testování softwaru), jelikož tento plugin/rozšíření je již používán (v infrastruktuře Red Hatu) pro zpracovávání výsledků testů OpenJDK.

Samotný nástroj umožňuje uživateli filtrovat testovací sady (a také jednotlivé sestavení těchto sad a jednotlivých testů) na základě specifikovaných parametrů, aby uživatel mohl pracovat pouze s určitou podmnožinou testovacích sad, které ho zajímají (například varianty jedné testovací sady na různých operačních systémech). Nástroj podporuje dva základní druhy tohoto filtrování – první je pomocí názvu testovacích sad (například pomocí regulárního výrazu) a druhé je pomocí hodnot, které jsou uloženy v různých konfiguračních souborech dané sady (filtrovat se můžou hodnoty z XML, JSON nebo properties souborů).

Poté, co jsou sady vyfiltrovány, komparátor provede jednu z operací, kterou uživatel specifikoval. Kromě pomocných operací (operace bez nějakého většího informačního charakteru pro uživatele) má komparátor tři základní operace – vytvoření tabulky spadlých testů na vybraných testovacích sadách (řádky v tabulce jsou pro jednotlivé spadlé testu a sloupce pro jednotlivé sestavení testovacích sad, uvnitř tabulky jsou znaky X, kde test z řádku spadl na sestavení ze sloupce), vytvoření tabulky výsledků jednotlivých sestavení a vytvoření tabulky podobnosti stack trace jednotlivých spadlých testů (místo znaku X jsou v tabulce procenta reprezentující podobnost stack trace k referenčnímu sestavení, spočítané pomocí Levenštejnovy vzdálenosti).

Nástroj byl otestován na OpenJDK QA infrastruktuře v Red Hatu, pro kterou byl zároveň primárně vyvíjen (a během celého procesu vývoje byl nasazený a průběžně kolegy využívaný), ale také na lokální testovací infrastruktuře, která byla vytvořena, aby simulovala testovací infrastruktru organizace Eclipse Adoptium (jejich Jenkins instance je veřejně dostupná, včetně možnosti stáhnout si výsledky jejich testů), která poskytuje velmi populární předsestavené OpenJDK. Výsledek tohoto testování byl pozitivní, jelikož komparátor funguje na obou infrastrukturách "out of the box" a je možné jej bez většího/težšího nastavení využít.

Comparator of Test Results

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Aleš Smrčka, Ph.D. The supplementary information was provided by Mgr. Jiří Vaněk. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Patrik Čerbák
May 5, 2024

Acknowledgements

I would like to thank my supervisor Ing. Aleš Smrčka, Ph.D., and my consultant Mgr. Jiří Vaněk from Red Hat Czech, for leading me through the process of creating this thesis and giving me helpful advice along the way. I am thankful for the support I got from both of them, I appreciate it.

Contents

List of Figures

Chapter 1

Introduction

At the time of writing this thesis, I am employed as an intern in an OpenJDK Quality Assurance team at Red Hat. My main job there is to help with the processing of test results and to make this process easier for the whole team.

Since OpenJDK is an extensive project with a lot of customers depending on it, there are thousands of tests run every single build, so the task of processing the results can get really difficult very easily. This is where the aim of this bachelor's thesis comes in. I was asked to design and implement a tool for comparing results from different test suite variants (one test suite with the same tests can have many variants – they differ, for example, in operating systems, runtime architecture, etc.).

This tool will be part of an open source plugin for Jenkins, and it should be usable as a stand-alone tool through the terminal or a web interface. It should also be intuitive and easy to use but also as generic as possible so that it can be used by the Red Hat OpenJDK team and other teams and institutions that use similar infrastructures as we do.

In this thesis, all necessary technologies and tools are described to understand our infrastructure and the process of testing OpenJDK in Red Hat. Then, the design and implementation processes of the comparator tool are described. Lastly, there is a chapter that evaluates this tool on our infrastructure and the infrastructure of the Eclipse Adoptium organization.

Chapter 2

Technologies and Process of Testing the OpenJDK

In this chapter, the technologies and tools necessary for understanding our team's processes and this thesis as a whole are described. It contains a basic description of unit and integration testing and other technologies (for example, the Jenkins tool) that are important to understand.

There is also the analysis and description of our team's testing of the OpenJDK. From an outside perspective, this process can seem a bit difficult and confusing, but it is described in detail to shine a light on it.

2.1 Important technologies and terminologies

2.1.1 Java, JVM, JDK, and JRE

Java is a high-level object-oriented language that is also platform agnostic. This is achieved because rather than compiling into machine code, the Java compiler compiles a given code into a so-called *byte code*. This byte code is then interpreted by *Java Virtual Machine* (or JVM for short).

This JVM can be installed on a wide range of different platforms. So, for example, a program compiled on Linux can then be run on Windows, MacOS, and many more platforms without changing the original code. The visualization of this process can be seen in image 2.1. As you can tell, the files with Java code have a `.java` extension, and the compiled byte code has a `.class` extension.

To install Java, you can choose between two things. If you want to develop Java software, you need a *Java Development Kit* (or JDK for short), which contains all the necessary tools for development, mainly the Java compiler (called *javac*), tools for packaging compiled classes into JAR archives, debugger, and the JVM for running the Java programs. However, when you just need to run the already compiled programs, only *Java Runtime Environment* (or JRE), containing JVM, is needed.[10]

2.1.2 OpenJDK

You can get JDK and JRE binaries directly from Oracle (the company behind Java), but there is also an *OpenJDK*, which is a free and open source implementation of Oracle's Java Platform, Standard Edition.[7] There are a lot of companies and foundations that are

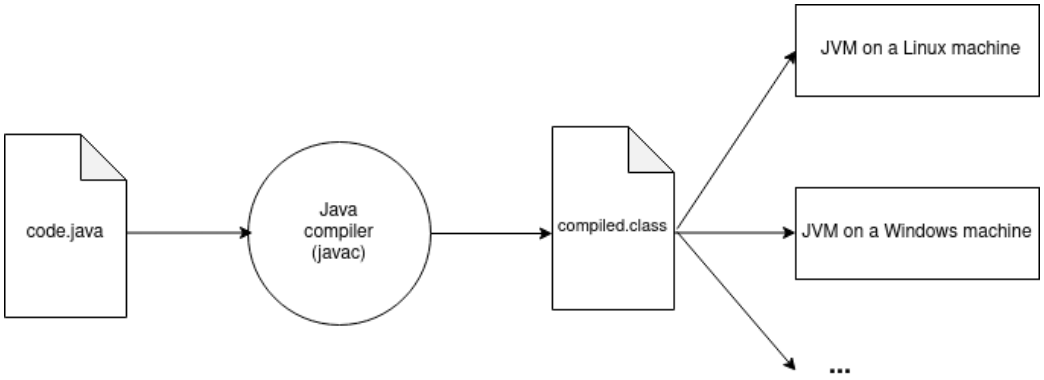

Figure 2.1: A diagram showing the Java compilation process.

building and distributing their build/version of OpenJDK, for example, Red Hat, Microsoft, Eclipse Adoptium, etc.

So, this is precisely where my team comes in. We are testing Red Hat's build of OpenJDK, which is then used in Red Hat Enterprise Linux or Fedora package repositories (but Red Hat also distributes containers with their OpenJDK build or even a Windows build).

2.1.3 Unit testing

Generally, testing is a process for evaluating whether or not a system meets its specified requirements. This process results in finding the difference between the expected and actual results. Software testing consists of finding bugs, errors, or even missing requirements in software.[3] Testing is a crucial (some may even say the most important) part of developing any software. Especially when dealing with large and complicated software that is used in production by millions of companies and people.

Probably the most common type of software testing is so-called *unit testing*. Unit testing focuses on testing a single unit of code, like a method or a class.[11] With unit testing, each individual piece (or component) of software is tested separately, without the interference of other pieces. Ideally, every part of any software should have some unit tests written with explicitly given criteria for the tests to pass (the criteria should be based on the expected behavior of the piece of code that they test), and the tests should be ideally run automatically with every change to the code (for example during the compilation process).

2.1.4 Integration testing

Another greatly used type of software testing is integration testing. This testing is done when all the individual units of a program are formed together to create a functioning program (or a bigger part of a program). It emphasizes the interaction between multiple software units.[6]

So unlike unit testing, where each test is separate and does not interact with the interface of other pieces of the program, integration testing does that and tries to cover the inter-unit interactions.

2.1.5 JUnit and its test output

JUnit is an open source and probably the most popular unit testing framework for Java, written by Kent Beck and Erich Gamma. Together with many other frameworks for different programming languages, it is collectively in an xUnit framework family.[11]

The output of every JUnit test is saved into an XML file when run. These XML files have a single standardized structure (equal to all the test frameworks from the xUnit family). The main element is the `<testsuite>` element with some properties describing it and `<testcase>` child elements for every test. When a failure occurs, a `<failure>` element is generated with the test's stack trace. You can take a look at an example of this XML file from a test called `RectangleTest` in figure 2.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="org.example.RectangleTest" time="0.04" tests="2" errors="0" skipped="0" failures="1">
  <testcase name="testArea" classname="org.example.RectangleTest" time="0.016"/>
  <testcase name="testPerimeter" classname="org.example.RectangleTest" time="0.005">
    <failure message="expected: &lt;16.0&gt; but was: &lt;30.0&gt;" type="org.opentest4j.AssertionFailedError">
      <![CDATA[org.opentest4j.AssertionFailedError: expected: <16.0> but was: <30.0>
      at org.junit.jupiter.api.AssertionFailureBuilder.build(AssertionFailureBuilder.java:151)
      at org.junit.jupiter.api.AssertionFailureBuilder.buildAndThrow(AssertionFailureBuilder.java:132)
      at org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:197)
      at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:70)
      at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:65)
      at org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:889)
      at org.example.RectangleTest.testPerimeter(RectangleTest.java:17)
      at java.base/java.lang.reflect.Method.invoke(Method.java:568)
      at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
      at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
      ]]>
    </failure>
  </testcase>
</testsuite>
```

Figure 2.2: A screenshot of the XML output of a JUnit test.

2.1.6 Jtreg

Jtreg is a test harness tool (it started as a regression test harness, but it also supports unit and other types of tests) for testing the JDK itself. It was created in 1997, and around that time, JUnit did not exist yet. For a long time, this was a proprietary tool and closed source. However, nowadays, it is open source and still the main method for testing the Java Development Kit.[5]

Jtreg currently supports multiple different types of tests, including *API tests*, which test the running of some program and whether it returns normally or not, *compiler tests*, which whether some source files compile or not, *applet tests*, which test Java applets in some HTML and *shell tests* which run a shell file and other files for testing some other parts of the JDK.[8]

2.1.7 Jenkins

According to its website, "Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software."[4] Generally speaking, Jenkins is a tool for continuous integration of software.

The Jenkins build system is written in Java and is very easily extensible thanks to its architecture. Because of that, Jenkins has a very vast plugin ecosystem and is very flexible in the tasks it can do.[1]

Jenkins jobs

In Jenkins, the users can define jobs (or projects), which are a set of defined tasks that happen when a predetermined condition is met. These jobs are the core part of Jenkins because all of the automation happens within them.

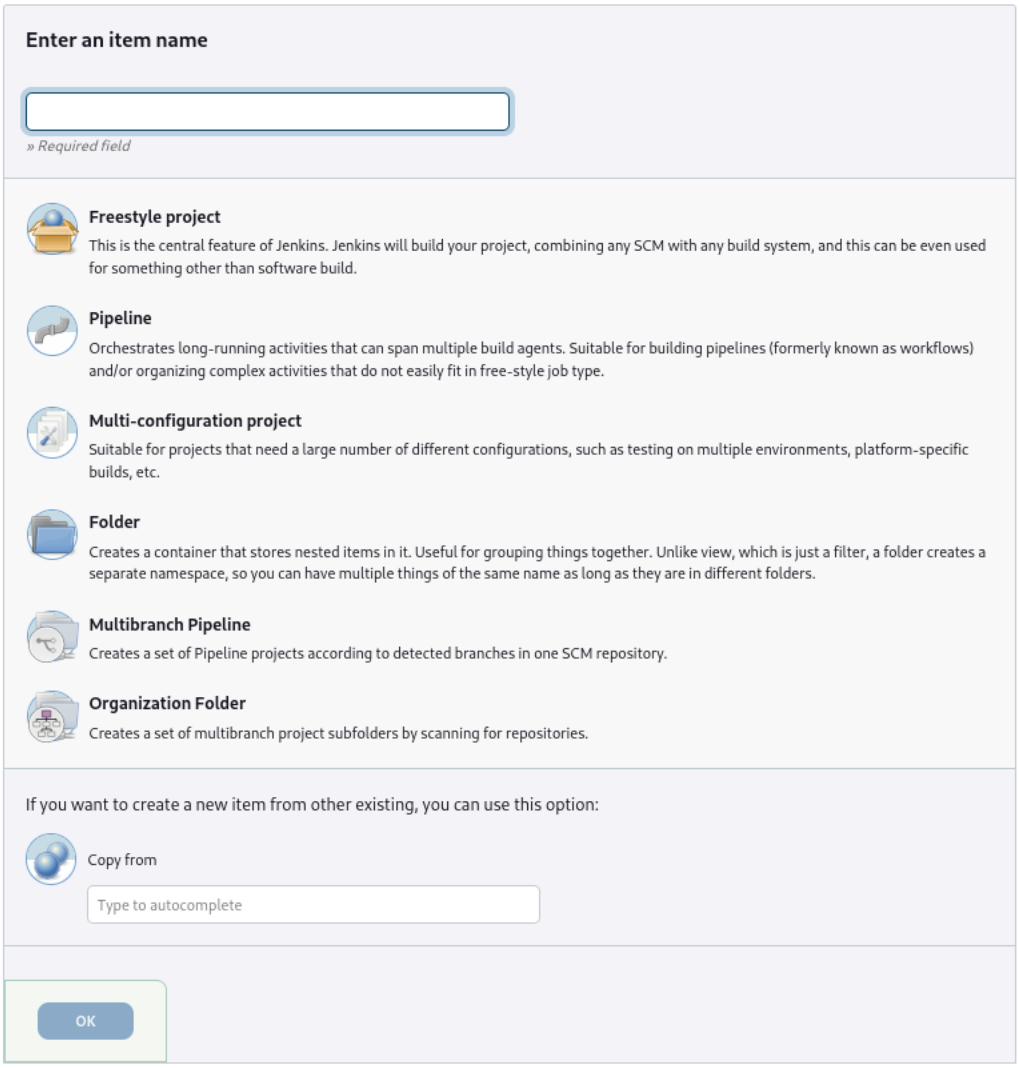

Figure 2.3: A screenshot of the Jenkins *create new job* screen.

When creating a job, the user can select when the job triggers. This can, for example, be when a new push is made into a certain Git branch, after other projects are built, or it can trigger periodically or even manually. After a job is triggered, a set of build steps is run. These build steps can range anywhere from executing a shell command to invoking compilated build and test scripts. Since there are a lot of plugins for Jenkins, the possibilities are almost endless.

Then, finally, after all the build steps are done, post-build actions are executed. These actions are, for example, used for packaging and publishing test results (or making a report of the test results), but they can also be used for sending notifications that the job has finished.

Results of builds

Every build of a Jenkins job can end up with a few different results depending on how successful it was. These results are as follows:

- **success** – no test in this build failed
- **unstable** – some tests in this build failed, but the build itself did not crash
- **failure** – the build crashed while building or testing
- **aborted** – the build was aborted
- **not built** – the build was not built (can be for many reasons)

Jtreg plugin

Jtreg plugin[1] (or officially `jenkins-report-jtreg`) is, in our infrastructure, a really important Jenkins plugin. This plugin's main purpose is to process results of JUnit, Jtreg, and TCK/JCK (Technology/Java Compatibility Kit – a suite of tests for checking whether a given implementation of a software, Java language in this case, is correct and according to the specification. It also contains tools for running the tests and reporting the results.[12]). The plugin reads archived XML files produced by these frameworks and creates summaries, which then show the results of tests in detail.

These archives contain thousands of XML files with the failed tests' stack traces and other data concerning the tests' running. To optimize this process, the plugin creates two JSON files from them, one with only failed tests and their stack traces and one with complete listings of all of the tests. Now, only these two files need to be accessed by the plugin, speeding up the process immensely because opening thousands of small files instead is inefficient and would take a lot of time.

Together with another plugin, `chartjs-api`[2] it can also generate graphs on the project's page showing the number of failures (and how they change in time) or the total number of tests run.

Even though this plugin was primarily created for our infrastructure and needs, it is open source and licensed under the MIT license. It is also a part of the official Jenkins plugin repository[3].

[1] https://github.com/jenkinsci/report-jtreg-plugin
[2] https://github.com/jenkinsci/chartjs-api-plugin
[3] https://plugins.jenkins.io/report-jtreg/

Figure 2.4: A screenshot of a report page of the Jtreg plugin.

2.2 The process of testing OpenJDK in Red Hat

This section focuses on the analysis and description of the process of testing the OpenJDK in our team at Red Hat and all of its pitfalls.

2.2.1 Critical Patch Update

First, explaining what a *Critical Patch Update* (or CPU for short) is is important. Critical Patch Updates are updates to Oracle's products (including the JDK) that mainly provide

security patches. They are released on the third Tuesday of January, April, July, and October.[9] The OpenJDK also follows this release structure.

2.2.2 Building and testing the OpenJDK

Each time a CPU to JDK comes, the QA team's job is crucial. The team needs to build all the supported versions of OpenJDK to all platforms (x86, ARM, etc.) and operation systems (Red Hat Enterprise Linux, Fedora Linux, etc.) Red Hat supports. Then, after an OpenJDK for any of these is built, a huge set of test suites is run on it, testing many aspects of the OpenJDK as a whole. There are suites for unit tests, regression tests, Technology Compatibility Kit tests, GUI tests, and cryptography tests, as well as benchmark and stress tests to check if there is no loss in performance with the new versions.

Each of these test suites is run in many different runtimes with many different JDK configurations. These configurations include whether or not to have debug symbols, whether or not to run with a default or alternative garbage collector, whether or not to run on x11 or Wayland display servers, whether or not to have the JDK Flight Recorder (an event recorder built in OpenJDK) on or off, and so on.

Each of these individual test suites has around a hundred thousand to half a million different tests run.

So, to sum up, each version of OpenJDK is built on many platforms and OSes. Each of these has a set of test suites with different configurations, and each suite runs hundreds of thousands of individual tests that all need to be recorded.

This whole process is automated using Jenkins. In our infrastructure, each test suite has a unique name depending on the OpenJDK version, operating system, platform, and configuration, and each suite is treated as an independent Jenkins job. An example of this name can be:

```
tck-jp8-ojdk8~rpms-el8.aarch64-fastdebug.sdk-el8.aarch64.beaker-x11.
    defaultgc.fips.lnxagent.jfroff
```

When is this name split by a dot or a hyphen character (`.` or `-`), each part of the name represents a different configuration.

When the tests are finished, Jenkins publishes the results for plugins to process them. The Jtreg plugin then reads those results and generates a report page for every build.

2.2.3 Checking the results

Members of our team then check the reports for any critical test fails or new fails on suites that were stable before. However, when encountering a failure, it is also important for my colleagues to know why the test they observe failed. One of the most important checks is to look at whether the test failed only on one platform (or configuration) or more. Another thing they might want to know is whether the test failed on the same problem as the build before (or the build with a different configuration).

2.2.4 A place for comparator

This is where this thesis' *comparator tool* comes in. It should speed up this process by comparing the results of the tests across the desired platforms and configurations and showing where the tests differ. As my team's whole testing pipeline is done using Jenkins,

the comparator tool will be implemented as a part of the above-mentioned Jtreg plugin. However, the main core should be usable as a standalone command line tool. But more on that in the next chapter.

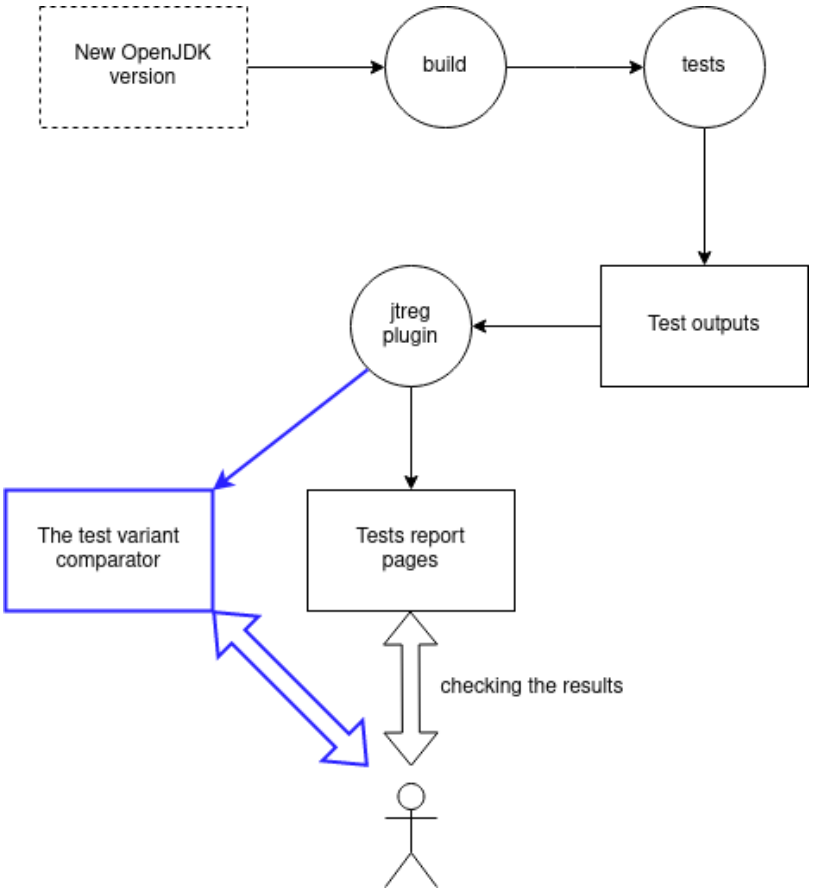

Figure 2.5: A chart showing a simplified workflow of our team while checking results (with the comparator tool).

Chapter 3

Designing the Comparator Tool

This chapter describes the comparator tool's design and its place in our team's infrastructure. First, an analysis of the tool's requirements and capabilities is conducted, and then the design is described, including how it should function and operate.

3.1 Comparator requirements

This section contains a review of every main requirement set for the comparator. Each requirement has a heading and a short description explaining its meaning.

3.1.1 Filtering the suites by name and other properties

It is crucial for the tool's usability to provide a simple solution for filtering which suites to work with since working with all of the suites would make the tool unusable. For example, the user may only want to compare suites that differ in the operating system but not anything else. This is where a good and easy system for filtering the suites comes in.

There are two major things the user should be able to filter the suites by. The first thing is the name of the suite (the name of the job in Jenkins), and the second is the job properties saved in some configuration files in the job's directory.

3.1.2 Showing on what variants did the tests pass and on what failed

This is probably the comparator's most important feature since this is where the results are compared. After thinking about this problem and discussing it with my team, the most effective and comprehensive solution that came to mind was showing the user a results table.

In this results table, each column should represent a different variant of a test suite (for example, a test suite ran on different operating systems), and each row should represent a specific test from this suite. This table's contents (in the middle) should be a character `X` when the test failed on the given variant from the column.

3.1.3 Filtering the individual tests to compare

Every test that failed at least in one variant should be shown in the table by default. However, if the user wants to just see a few predetermined tests, they should have the ability to do so. The comparator should have the option to filter tests by regular expressions.

3.1.4 Showing only volatile tests

There are tests that always fail in all variants. To avoid cluttering the table with them, users may wish to filter them out and not show them. The comparator should be able to do that and not show the tests (so only tests that fail in at least one variant, but not all, should be shown).

3.1.5 Showing how the test results differ in time

Another important thing for the comparator to be able to show is the evolution of the test result in time. This can be achieved by adding a possibility for the user to specify how many builds of every matched suite (a build of a Jenkins job) into the history they want to analyze. The tool should then take the last N builds of a job instead of just one and extend the results with them.

3.1.6 Listing matched test suites

This feature should be available mainly for users who are getting accustomed to the tool. It should match jobs (and their builds) based on all the criteria set by the user. Instead of creating a table of failed tests and cross-comparing them (or performing any other comparator operation), it should just print a list of those matched jobs.

3.1.7 Showing a table with the results of a build

Sometimes, it is convenient to show a table with the general results of the whole build of a test suite. This table should take all matched builds, get their result (according to 2.1.7), and just print a table of them.

3.1.8 Comparing test stack traces

Apart from just comparing where tests failed, the tool should also be able to compare whether the tests failed on the same problem – one way of doing this is by comparing the stack traces of the tests.

This functionality should give a similar table as the test failure compare functionality, but instead of putting `X`s into the table, there should be percentages of how similar the stack trace of a given test is to the stack trace of a referential build.

The referential build should be automatically chosen by some criteria (probably using the first build the tool finds is enough), but the user should also have the ability to choose a different referential build.

3.1.9 Integration into Jenkins

To integrate this tool more into the QA infrastructure, it should be easily accessible from Jenkins. On every report page of the Jtreg plugin, there should be an adjustable set of links that take the user to the comparator tool with preset arguments so the user can easily look at the results without writing the arguments themselves.

3.2 Designing the Tool

Based on these criteria for the tool, the next section focuses on designing the comparator. It starts by describing how this tool should be used. Then, it continues with an explanation of the basic workflow of the program, and lastly, it describes the individual parts of the tool (each individual part solves some requirement).

Even though this design is general and not specific, the tool is programmed in Java, so it references some Java-related terminology or technologies.

3.2.1 The way to use this tool

There are two basic ways the user can use this tool – running it in the command line by running the binary itself and running it through a web interface/wrapper, which runs the tool and returns the output to the user.

Running the tool in the command line is pretty straightforward. The user can run Java with the classpath (an option to locate external classes the JVM can use during runtime[2]) option to specify the location of the compiled comparator jar archive together with all of its dependencies. The tool takes arguments directly from the user, specifying them in the CLI. It can then show the output in multiple different formats, including text or HTML.

Another option is running the tool from a web interface – this interface is a simple wrapper around the tool, which takes arguments through the interface, runs the tool in the background, and then prints the output to the user (this is where the HTML formatting makes the most sense). This webpage contains the basic help prompt of the tool as well as an input field to specify the tool arguments and two output frames for the tool's standard output and the tool's error output.

This webpage can also take the comparator arguments from the HTTP GET method, enabling them to be encoded directly in the URL. This makes links from Jenkins to the comparator possible.

The tool, with all of its tooling, is a part of the Jtreg plugin since there are already classes and methods for getting test results and other things that are helpful for implementing the comparator.

3.2.2 Basic workflow

Now, in this section, a description of the basic workflow is provided in bullet points. This is important for understanding the whole comparator work process and its parts. Each of these is described in detail later in this chapter.

1. getting arguments from the user, parsing them, and saving the options the user chose
2. getting all available test suites
3. filtering the suites
4. getting builds of these filtered suites
5. filtering the builds
6. *running a comparator operation on the filtered builds*
7. format and print the output to the user

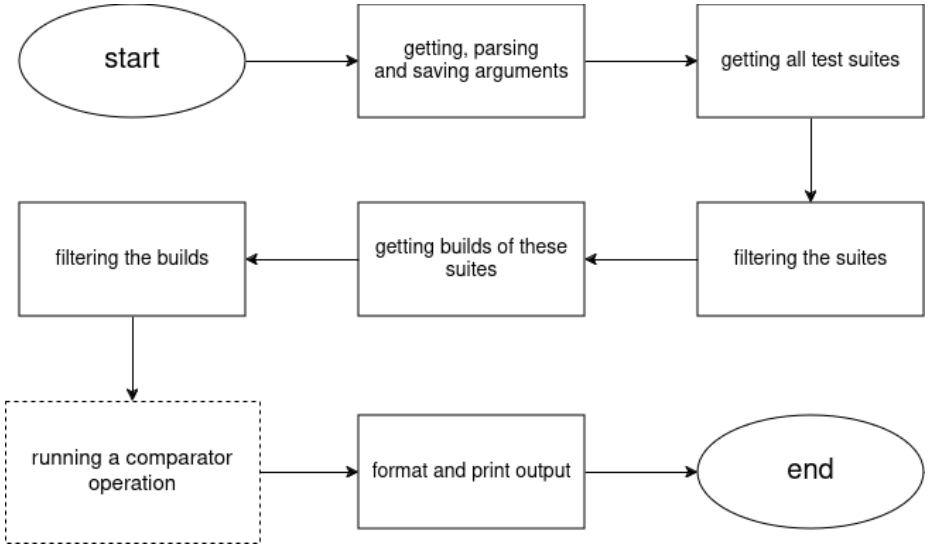

Figure 3.1: A simplified flowchart depicting the basic workflow of the comparator.

3.2.3 Getting and parsing arguments

The users will normally specify the arguments through the command line. These arguments are then parsed using an argument-parsing method, and the user-set options are saved to a specialized class (with some default values pre-set). This class is then available for other methods that can get the information through it.

If the parsing method encounters an unknown argument or a bad argument value, it throws an exception informing the user that the input was wrong. The method also checks for any missing mandatory arguments (also throwing an exception when missing).

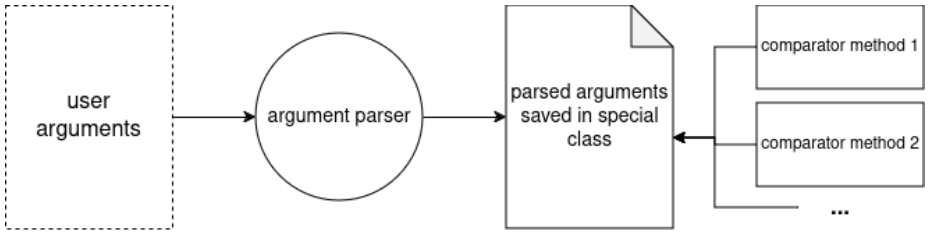

Figure 3.2: A diagram showing the process of argument parsing.

3.2.4 Getting all available test suites

Since a test suite is a Jenkins job and every Jenkins job has its own directory (which implies that every Jenkins job/test suite has a unique name), the system for getting the test suites works by accessing the filesystem, getting all directories from a given parent directory, and loading them into a list, which will be used later.

The user must specify the parent directory from which the tool will take the suites with an argument. This argument is mandatory because the tool cannot work without loading the test suites.

After loading all of the suites, the next step is to filter them to show only those that the user desires. The next two sections discuss this.

3.2.5 Filtering by suite names

When thinking about filtering the suites by their name, the first thought was to create a special query string system. This system can be very effective, but it only works in an infrastructure that follows our naming convention.

It works by splitting every job name by `.` or `-` characters and comparing each split part with a user-defined query string. The query string consists of N parts separated by spaces (or other whitespace characters), and each of these parts corresponds with the 1st up to the Nth part of the split job name.

You can see a LL(1) grammar of this syntax here:

$$
\begin{aligned}
\langle start \rangle &\rightarrow \langle expr \rangle \langle end \rangle \\
\langle expr \rangle &\rightarrow \langle expr \rangle\ \langle expr \rangle \\
\langle value \rangle &\rightarrow \text{value} \\
\langle expr \rangle &\rightarrow \langle value \rangle \\
\langle expr \rangle &\rightarrow !\langle value \rangle \\
\langle expr \rangle &\rightarrow * \\
\langle values \rangle &\rightarrow \langle value \rangle , \langle values \rangle \\
\langle values \rangle &\rightarrow \langle value \rangle \\
\langle expr \rangle &\rightarrow \{ \langle values \rangle \} \\
\langle expr \rangle &\rightarrow !\{ \langle values \rangle \} \\
\langle expr \rangle &\rightarrow \varepsilon \\
\langle end \rangle &\rightarrow \$
\end{aligned}
$$

In this syntax, there are some rules with terminal symbols that require further explanation:

- **value** – a part of a job's name. Only jobs that have exactly this value on the corresponding position of their split name will be matched.
- **{ and }** – an array with multiple values separated by commas (without spaces), jobs with one of the values in the array on the corresponding position of their name will be matched.
- **!** – reverses the matches, everything but this will be matched. It also works with arrays – everything but its contents will match.
- ***** – works as a wildcard that will match anything in this position.
- **$** – the end of the query string, it is not written. It is just a helping symbol in the syntax.

An example of this syntax:
`"jtreg~full jp17 * {f37,el8} !aarch64 !{fastdebug,slowdebug} * * *"`

- `jtreg~full` – specifies that the job's first part should be exactly `jtreg~full`.
- `jp17` – specifies that the job's second part should be exactly `jp17`.

- `*` - asterisk is a wildcard that matches everything, so in this example, the job's parts on the 3rd, 7th, 8th, and 9th positions do not matter – the tool takes everything on these positions.
- `{f37,el8}` – this is an array of possible matches, so the jobs' part in the 4th position can be either `f37` or `el8`. There can be as many elements as a user wants, but commas must split them with no spaces between them.
- `!aarch64` – `!` before a name is another wildcard that reverses the match, so this matches everything, but `aarch64`.
- `!{fastdebug,slowdebug}` – the exclamation mark also works before arrays. It matches everything but the values in the array.

This system works on our infrastructure because every test suite (Jenkins job) follows this naming convention. However, for this tool to be as generic as possible, there is a second way to filter the suites by their name, and that is by filtering with regular expressions. Matching with regex is very broad and works on every kind of infrastructure that uses unique names for every suite variant, but it can be harder to write than the simple query string. These two systems cannot be combined, so the user must choose if they want to filter with the query system or a regex.

3.2.6 Filtering by suite attributes

The comparator is able to filter the suites not only by name but also by their attributes. That is done because not every infrastructure has enough information in the job name to distinguish between the variants just by the name, but rather distinguishes between them by some file with their properties. After discussing the file formats that are widely used, the comparator supports XML, JSON, and basic properties (`key : value` pairs) files.

Now, continuing with the description of how this works. There is a file reader that finds a value for each of these file types. The values from XML files can be found using XPath, which is a special syntax for addressing parts of XML documents. The path primarily consists of one or more individual parts separated by `/` character, which together form a path to a node/nodes in the XML.[15] An example of this path: `/variants/properties[3]/names/gc`. Java has a parser for this path in its standard library.

Values from JSON files can be found using a simple subset of JSONpath[1]. This subset supports objects separated by dots, where the main object can (but it is not necessary) be represented by the dollar symbol, and getting a value from an array can be done using brackets. (This syntax was chosen because it is widely used in many programming languages for getting data from a JSON object, and many people are familiar with it.) An example of this query: `$.variants.properties[3].names.gc`.

Lastly, values from properties files can be found using a key from the pair.

While using this filter, the user declares it in a special argument. In this argument, they specify the file they want to look in for the specific value, specify how to look for the value (an XPath, JSONpath, or a key), and what the value is named (they can choose anything). After that, a new argument will be dynamically generated based on the user-specified value name. This argument takes a regular expression as a value and filters the suites by the specified value in files by the regex. The user can define as many filters as they want.

[1] https://www.rfc-editor.org/rfc/rfc9535.html

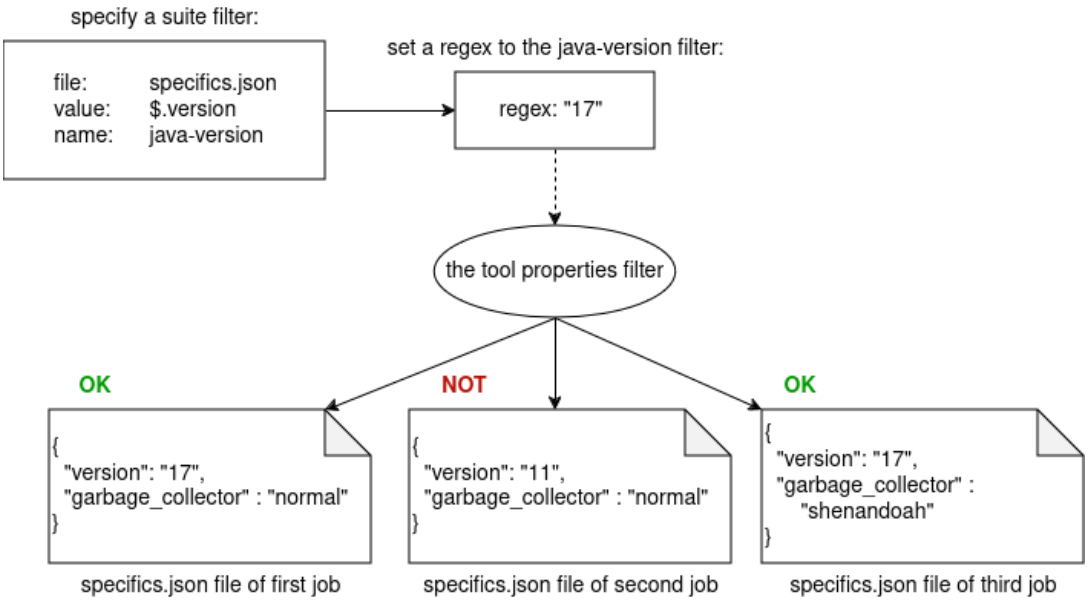

Figure 3.3: A diagram depicting the process of setting a filter and filtering.

This filtering by properties from files is more general than filtering just by suite name and can be used in more infrastructures. As figure 3.4 depicts, the user can combine the filtering by name and properties together.

3.2.7 Getting suite builds

Now, when the test suites are filtered comes the phase where the tool gets builds of each suite. Similar to the suites itself, the builds are stored in the filesystem. Each Jenkins job has a directory inside called `builds` in which there are directories corresponding for every build. These directories are called by their build number (so simply `1`, `2`, `3`, etc.).

So, to get the list of all builds, the tool goes through the filtered list of jobs, finds the `builds` directory, and gets all directories inside of it. When it has the list of all builds, it sorts and reverses the list (so the largest number is first – the latest build of the job). Now, the tool takes N jobs from the front of the list (the newest builds). By default, $N = 1$. However, the user has the option to change this number and take more builds. The N builds are now filtered, and all those that match the filter are added to a list of all matched builds from all matched jobs.

When a test suite is new, and the user wants to compare more builds into history than there are, the tool simply compares all the available builds it can get. Also, by default, the tool only compares builds of a suite that were successful or unstable, but it skips failed builds. So when the user wants to see N builds into history and M of them failed and others were successful/unstable, the tool only shows $N - M$ builds.

3.2.8 Filtering the builds

As with suites, builds can also be filtered. Since filtering the builds by their name makes no sense (the name is always just a number), only filtering by build attributes (like their build ID or other build artifacts) is applied. It functions almost the same as the filtering

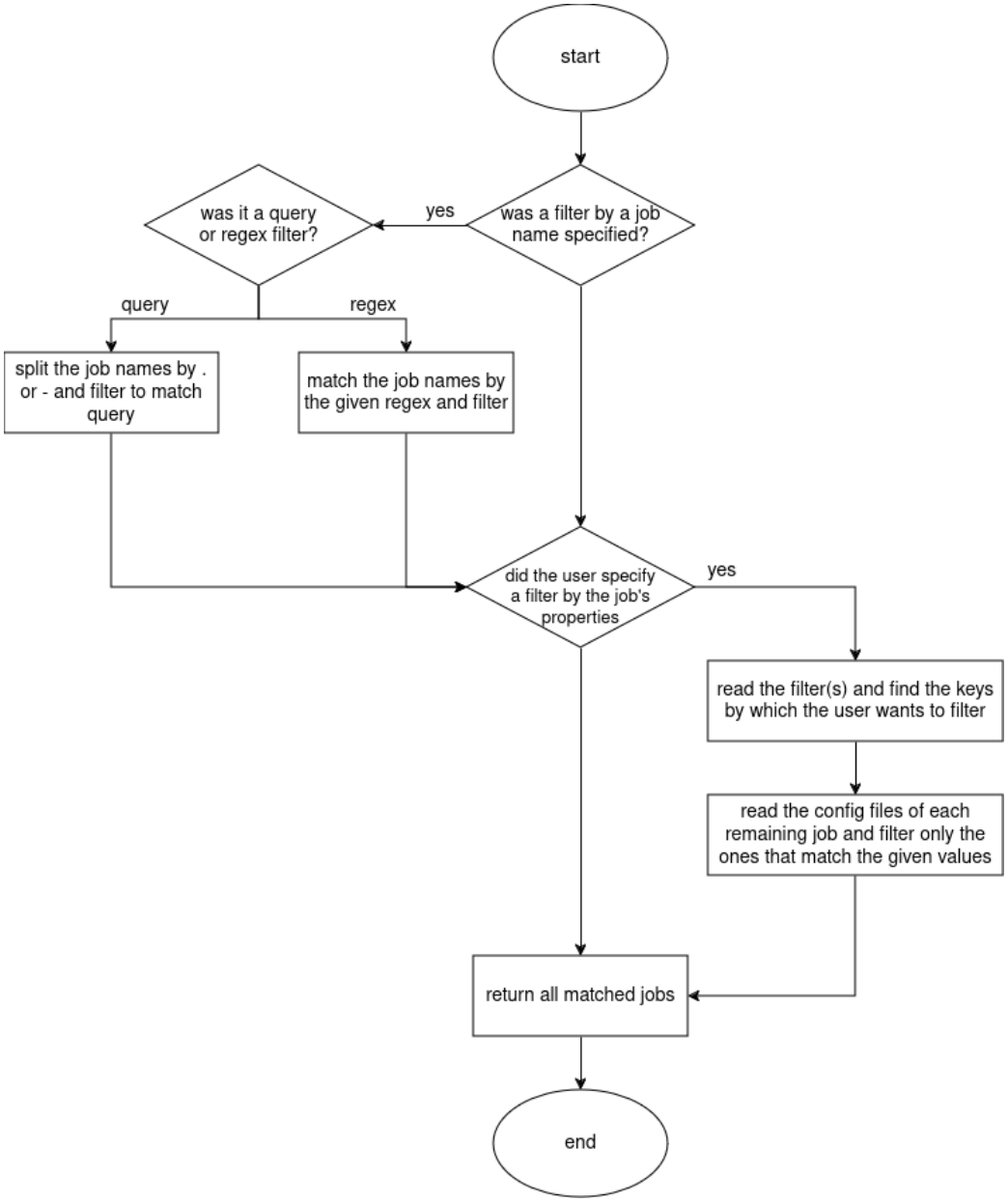

Figure 3.4: A flowchart depicting the whole process of filtering test suites.

of jobs by their properties (section 3.2.6), but it has its own argument for setting a filter (setting what file, name of the property, and where to find it in the file). The user can then use a dynamic argument for the filtering itself, which is the same as with the suite filtering.

One filter is there by default, which is the filter to only take successful or unstable builds (since there is a guarantee that all tests have a result). But if the user wishes to take all builds, even failed, there is a switch for that, which disables this filter.

The whole process of getting and filtering the builds can be seen in the figure 3.5.

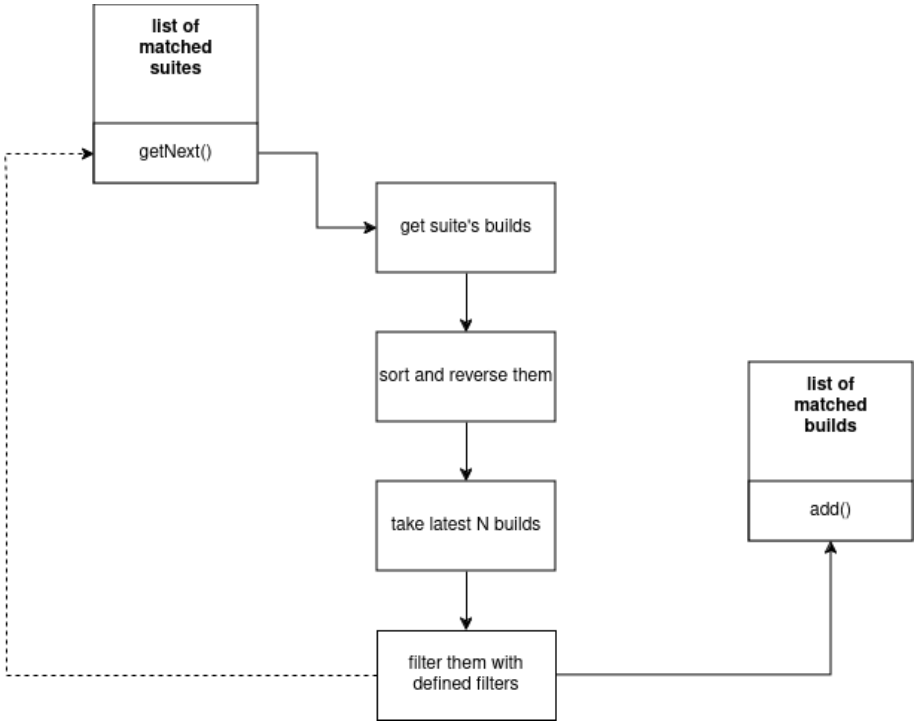

Figure 3.5: A graph showing the process of getting and filtering the suite's builds.

3.2.9 Comparing test failures on multiple variants in a table

Moving on to the actual comparison operation that this tool is able to perform. This is probably the most important feature and will be used the most. It is also the reason why this tool was created in the first place.

The tool first takes a list of all individual tests that failed for each applicable build (there are already methods for getting the failed tests in the Jtreg plugin, so those are used) and then composes those failed tests into a table, where each row of the table represents a test that failed in at least one of the builds and each column is assigned to one of the builds. If a certain test fails in a certain build of a suite, a character `X` is put in the cell that matches the test's row and the build's column.

To make the table more easily readable and tight, the tool gives the user a choice not to show builds without any failed tests (since there can be a lot of builds, and the table would be quite wide). However, this option is not enabled by default. The user has to specify that they want that with a special argument.

Another option for the user to edit this table is the ability to, first, show only volatile tests – meaning that tests that failed in all of the builds are not shown. And second, filtering the tests to be shown – meaning that the user has the ability to specify a regular expression and only tests that match this expression are put into the table. Both of these options have a separate argument that the user can use (and also combine).

An example of how this table can look (with example data) is shown in table 3.1.

From the table, users can easily see how the tests they are investigating differ from variant to variant. It can also give interesting patterns the user is looking for (a very simple example can be that a few tests concerning handling objects failed on variants with

- **1**: `suite_A` – `build:` 45
- **2**: `suite_A` – `build:` 44
- **3**: `suite_A` – `build:` 43
- **4**: `suite_A` – `build:` 42
- **5**: `suite_B` – `build:` 69
- **6**: `suite_C` – `build:` 37

	1	**2**	**3**	**4**	**5**	**6**
`failed_test_1`	X	X	X	X		
`failed_test_2`		X	X	X		
`failed_test_3`	X					X

Table 3.1: A failed test comparison table example with example data.

a default garbage collector but were successful on variants with a Shenandoah garbage collector), which would otherwise take a long time to cross-check manually.

There are two arguments for choosing this operation. The first one generates the table normally as described here, and the second one switches the rows and columns (but otherwise works the same).

3.2.10 Listing matched test suites and builds

As already mentioned in the requirements section, this operation will be for the users who want to try the tool and check how the filtering system works but do not want to wait for the tool to generate the comparison table. The tool normally matches all suites, gets all the builds that match the criteria, and prints the resulting matched data.

Another similar operation the tool will be able to do is to filter suites with the query string and print the variant groups. It matches the suites, but instead of printing them in a list, it splits the suite names by the `.` or `-` characters, and from each of the matched jobs, it puts the split parts together into groups (first part to the first group, second to second, etc.), enumerates these groups and prints them.

Both of these operations are not important for the comparator to function. However, they can help with getting accustomed to the tool and mainly the system of filtering the suites and builds. Both of these operations have a separate argument.

3.2.11 Showing a table with the results of builds

As was already described, sometimes it is convenient to also show a table of the result of a whole build (how the build ended up, not the individual tests). That is why this operation will be available.

The tool first filters all the jobs and then the builds, and after that, it looks into a file called `build.xml`, where the result of the build is stored (however, the user can change this default by setting a filter for the *result* value) and finds it.

When the file is missing, the comparator tracks it as if the build is currently running. But since the file can also be missing in the case of a, for example, very nasty crash, the

RUNNING keyword in the table has a question mark next to it to remind the user that the tool cannot be sure about that.

After that, the tool creates a table where rows represent all of the available results of the build (already described in section 2.1.7), and columns represent each matched build. Then, the character X is put in cells where the result in rows matches the build's result. An example of this is shown in table 3.2.

- **1**: suite_A – build: 23
- **2**: suite_A – build: 22
- **3**: suite_A – build: 21
- **4**: suite_A – build: 20
- **5**: suite_B – build: 42
- **6**: suite_C – build: 78

	1	**2**	**3**	**4**	**5**	**6**
SUCCESS		X				
UNSTABLE	X		X	X		
FAILURE					X	
ABORTED						X
NOT_BUILT						
RUNNING?						

Table 3.2: A build result table example with example data.

This operation has its own argument, and it is going to be the only operation that the user can combine with other operations since it can sometimes be helpful to show this table together with, for example, the table of individual test results.

3.2.12 Comparing test stack traces

With the operations that were already established in this thesis, users of this tool will be able to determine which variants of a test suite the tests failed and which variants they were successful in. However, it is also helpful to know if the test failed due to the same reason on all variants or not.

This is where this operation comes in. The tool first matches all jobs and their builds and gets all of the failed tests of the builds (till now, this has been the same process as getting the results for just the test failure comparison table – section 3.2.9). After that, instead of just creating a table of where the tests failed, the tool takes the outputs of all of the failed tests (each test can have multiple outputs, for example, standard output, standard error output, etc.), concatenate these outputs together and compare the similarity of them (in percentages). After the tool finishes running, it shows the output to the user in a table.

Levenshtein Distance

This comparison is done using an algorithm called the Levenshtein Distance. Levenshtein Distance is an algorithm created by mathematician Vladimir Levenshtein. The distance itself represents the number of operations (insert character, delete character, replace character) that need to be done to turn a string into another.[16]

An algorithm for calculating this distance, taken from Wikipedia (and adjusted for easier code integration based on a GitHub repository of a project called *similars*[13]), can be seen here:

s is the first string, and t is the second string to compare.[14]

$d \leftarrow [s.\text{length}() + 1, t.\text{length}() + 1]$
$i \leftarrow 0$
$j \leftarrow 0$
for $i \leq s.\text{length}()$ **do**
  for $j \leq t.\text{length}()$ **do**
    if $i = 0$ **then**
      $d[i][j] \leftarrow j$
    else if $j = 0$ **then**
      $d[i][j] \leftarrow i$
    else
      if $s.\text{charAt}(i-1) = t.\text{charAt}(j-1)$ **then**
        $sub \leftarrow d[i-1][j-1]$ // substitution
      else
        $sub \leftarrow d[i-1][j-1] + 1$
      end if
      $ins \leftarrow d[i][j-1] + 1$ // insertion
      $del \leftarrow d[i-1][j] + 1$ // deletion
      $d[i][j] \leftarrow \text{minimum}(sub, ins, del)$
    end if
    $j \leftarrow j + 1$
  end for
  $i \leftarrow i + 1$
end for
return $d[s.\text{length}(), t.\text{length}()]$

Then, to calculate the similarity percentage from this distance, this formula will be used:

$$100 - \frac{100 \cdot distance}{\text{maximum}(s.\text{length}(), t.\text{length}())}$$

Creating the table

These similarities are calculated for all stack traces from different builds that failed on the same test (or a single line in the table). By default, they take the first matched build as a referential and compare the other builds on a line to it. However, the user has the ability to change the referential build with an argument.

After that, the table with the percentages is printed. An example of how this can look is shown in the table 3.3.

- **1**: `suite_A` – `build:` 45
- **2**: `suite_A` – `build:` 44
- **3**: `suite_A` – `build:` 43
- **4**: `suite_A` – `build:` 42
- **5**: `suite_B` – `build:` 69
- **6**: `suite_C` – `build:` 37

	1	**2**	**3**	**4**	**5**	**6**
failed_test_1	100	91	91	91		
failed_test_2		100	34	89		
failed_test_3	100					100

Table 3.3: A stack trace comparison table example with example data.

3.2.13 Formatting the tool output

The tool will be able to print the output in different formats. The user can choose a formatter with an argument, but the default is always the plain formatter – meant for printing into the command line without any color or styling, just the text.

Another option for the user is printing with color formatting, which utilizes ANSI escape codes to add color or other formatting to the command line output. However, the contents of the output itself remain the same.

The last option for the user is HTML formatting, which utilizes HTML tags and CSS styling. It is the main output format for using the tool through the web interface.

These formatters are already present in the Jtreg plugin. However, they need to be extended. Mainly, a feature for printing tables through them needs to be added. Printing tables in HTML is easy since there are already tags for tables, but the job will get harder while implementing table printing to the command line because the table needs to be aligned with the right amount of spaces to be readable.

An example of this ASCII output of a table with example data can be seen below.

```
1) job_A - build:45
2) job_A - build:44
3) job_A - build:43
4) job_B - build:78

              | 1 | 2 | 3 | 4 |
failed_test_1 | X | X |   | X |
failed_test_2 |   | X | X |   |
failed_test_3 |   |   |   | X |
```

3.2.14 Integration into Jenkins

One of the goals of this tool was to be as easily usable as possible, ideally even without knowing all its different arguments and settings. To achieve that, the tool will be integrated into the Jenkins' Jtreg plugin.

Specifically, the report pages of some builds (report pages from the Jtreg plugin) will have dynamically generated links (or buttons) with already pre-filled comparator arguments that take users to the comparator web wrapper. So the Jenkins administrator can set the links in the Jenkins settings.

In the settings, the administrator can create multiple sets of links. Each set contains a regular expression and is only shown on report pages of suites whose names match the regular expression (because it can be unnecessary to have all links on all types of suites). For each set of links, the administrator can then define multiple links – each link contains a name shown on the report page, and each link has a field to write the comparator arguments (one on each line) that will be encoded in the URL.

Apart from this, in the Jenkins settings, there will also be forms in which the user can define a suite and build filters (there will be fields for four things in those forms: config file name, whether it is a job or build filter, name of the config item and how to find the config item) for looking into files. The data from these fields will then be compiled and added automatically as filter definition arguments to the links. The administrator can then only need to enter the dynamic argument with the value by which to filter (to individual links declaration), nothing else (it is primarily meant to make the declaration of arguments more straightforward).

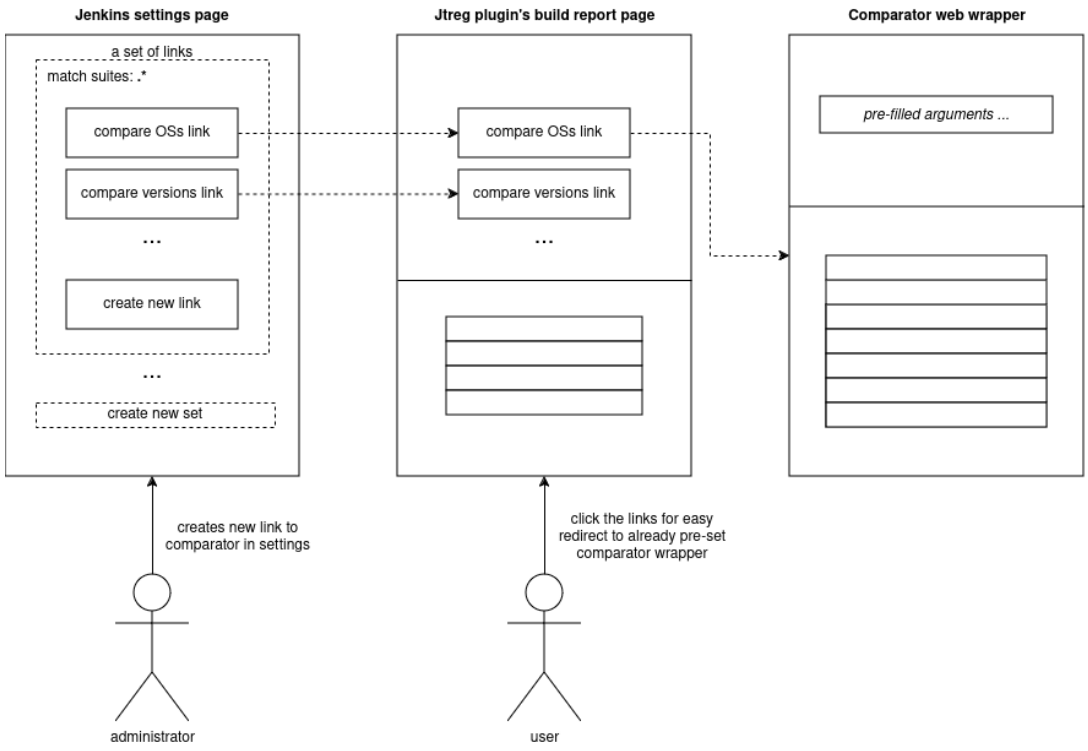

Figure 3.6: A diagram showing the creation and usage of links to the comparator.

Macro system

For these links to the comparator to function properly, they need to be generated dynamically (meaning that they are different on every report page). For this reason, the administrator can define some macros in the arguments field, and they will be replaced

with a value from a specific suite's or build's name or value from configuration files when a user loads the Jtreg plugin's report page with those links.

Since these macros work with a suite's name, there is one more field when declaring a link to the comparator, a *spliterator* – this field takes a regular expression by which the suite's name is split into parts. There will then be macros that refer to a specific part of the split name.

Those macros look like this, `%{macro}`, and there are multiple types of macros the administrator can use:

- `%{x}`, where the administrator will replace `x` with a number corresponding with the "`x`th" split part of the suite's name (indexing starting with 1), t will be replaced with that part.
- `%{N-y}`, where `y` will be replaced with a number corresponding to the "`y`th" split part of the suite's name from the back, and it will be replaced with that part. If just `%{N}` was entered, it will be replaced with the last part of the name.
- `%{SPLIT}` or `%{S}` will be replaced verbatim with the regex from the spliterator field.
- `%{dynamic}`, where `dynamic` is replaced with the name of a config item specified in the filter declaration section. It will be replaced by a value from the corresponding config file from the current job and build.

For an example of this system, let's say that there is a job with the job name `jtreg~full-jp11-ojdk11~rpms-f36.x86_64-fastdebug.sdk-f36.x86_64.testfarm-x11.defaultgc.legacy.lnxagent.jfroff`, spliterator is set to `[.-]` and a build filter with the name of the config item set to `nvr` (its evaluation in the config file is `java-17-openjdk-17.0.6.0.9-0.4.ea.el8`).

Now, if the administrator were to set these two arguments

```
--regex "%{2}%{S}%{N-1}_rhel9_%{N}"
--nvr "%{nvr}"
```

it would be evaluated and replaced with

```
--regex "jp11[.-]lnxagent_rhel9_jfroff"
--nvr "java-17-openjdk-17.0.6.0.9-0.4.ea.el8"
```

Chapter 4

Implementing the Comparator

This chapter describes the process of implementing the comparator tool based on the design from the previous chapter. It digs into the codebase of the Jtreg plugin and describes the modules and where the comparator is placed. Then, it describes where each feature of the comparator is implemented.

4.1 Jtreg plugin codebase and the place for comparator

The comparator tool itself is part of an already mentioned Jenkins plugin, the *Jtreg plugin*. This plugin already contains classes and tools for analyzing test results, which were used when implementing the comparator. Since plugins for Jenkins are programmed in Java (including this plugin), the comparator tool is also written in Java.

One of the tasks while implementing the comparator was to split the codebase of the Jtreg plugin from one big module (as it was before) to multiple modules, each for a logical part of the plugin. This was done so the comparator (and a *diff tool* that is described below) can be used independently as a command line tool without having dependencies on any Jenkins classes. You can read a brief description of each module in the list below.

- `report-jtreg-lib` – This module is a library module, it contains classes, that are used by the other modules. It, for example, contains formatting classes that are used for formatting the tools' output (plain text, formatting with color or HTML formatting), model classes that describe the model structure of the tests, their results, and whole suites (an abstraction used by the plugin), a class for constants, and many more useful tools.

- `report-jtreg` – This module is the module for the Jenkins plugin itself, it has classes and other files that describe the graphical interface of the plugin as well as settings and all the other classes, that in some way extend the classes of Jenkins itself.

- `report-jtreg-diff` – This module contains a standalone tool that shows differences between two builds of a test suite. It can show many differences, such as the number of tests run, the number of tests that were fixed, or the number of new fails. It can be viewed as a "spiritual predecessor" to the comparator.

- `report-jtreg-service` – This contains a service that is used for running the diff tool (and also the comparator tool) and showing the output. It acts as a web application

that takes the tool arguments from the user and runs the corresponding tool with them. After the tool finishes running, it shows the output to the user.

- `report-jtreg-comparator` – The comparator is placed into this module.

The Jtreg plugin uses *Apache Maven* for its build system. The top-level directory of the plugin contains a `pom.xml` file in which there is a description of all the modules. Each module also contains a `pom.xml` file that describes the building process of the module. Most of the modules simply build into a single *JAR* file (an archive of compiled Java classes), but the `report-jtreg` module builds into an *HPI* file – Jenkins uses this file format for plugins. Thanks to this partitioning, the comparator tool can be built separately and used as a stand-alone command line tool.

4.2 Implementing the features

If not specified otherwise, all of the Java classes described in this section are located in the `report-jtreg-comparator` module, specifically inside of the `report-jtreg-comparator/src/main/java/io/jenkins/plugins/report/jtreg/main/comparator` directory. That means that every class in this directory has `io. Jenkins.plugins.report.jtreg.main.comparator` package.

The main class is called `VariantComparator.java`, and it calls all other comparator classes and operations.

4.2.1 Parsing arguments

There are four classes used for parsing and saving arguments from the user:

- `arguments/Arguments.java` – This is a template class for all available arguments/-command line switches of this tool. It stores their name, help text for it, and the usage of it (the information about whether an argument is expected after the switch and what kind).
- `arguments/ArgumentDeclaration.java` – The specific arguments are declared in this class.
- `Options.java` – This class saves all of the options the user specifies through the arguments. It is created while parsing arguments and then passed to other classes that use the values saved into it. It also has dynamic lists for the filters from properties the user can specify.
- `arguments/ArgumentsParsing.java` – This class takes the arguments from the main class and parses them one by one. The result of this parsing is saved into the `Options` class (even the dynamic argument filters).

4.2.2 Getting all suites

These classes are used for getting (or "listing") all of the suites:

- `listing/DirListing.java` – This is an interface for other classes that do the listing of the suites (Jenkins jobs). It declares only one method, `getJobsInDir()`, which returns a list of all jobs it finds.

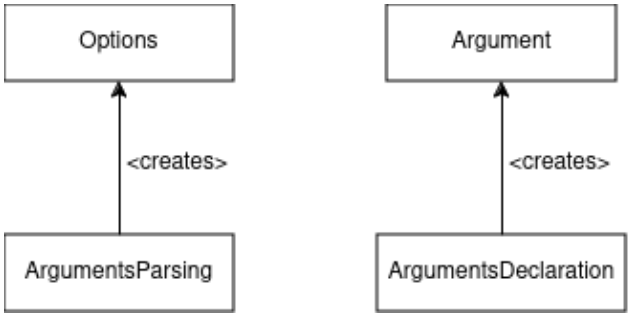

Figure 4.1: A simplified class diagram of classes associated with argument declaration and parsing.

- `listing/FsDirListing.java` – An implementation of this interface that finds the job in the file system. It looks into a user-set directory (the directory with all the Jenkins jobs) and returns all jobs it finds.
- `listing/ListDirListing.java` – This is another implementation of the interface, and it is only used in unit tests, as it replaces looking into the filesystem by just returning a list the user gives it in the constructor.

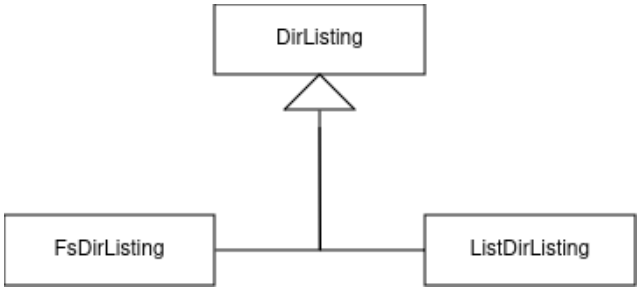

Figure 4.2: A simplified class diagram of classes associated with getting all suites.

4.2.3 Filtering the suites

There are four classes that are used for filtering suites by their name, specifically:

- `jobs/JobsProvider.java` – This is an interface for other classes that filter by name. It defines methods for adding jobs to it, filtering them, and returning them, as well as methods for parsing (and listing possible) additional arguments that these filters can get to specify the filtering behavior further.
- `jobs/JobsByRegex.java` – This is an implementation of this interface for filtering the jobs by their name using regular expressions.
- `jobs/JobsByQuery.java` – This is an implementation of this interface for filtering the jobs by their name using a query string described earlier.
- `jobs/DefaultProvider.java` – This is also an implementation of this interface, and it is used as the default if no filter by name is specified. It just returns all of the jobs.

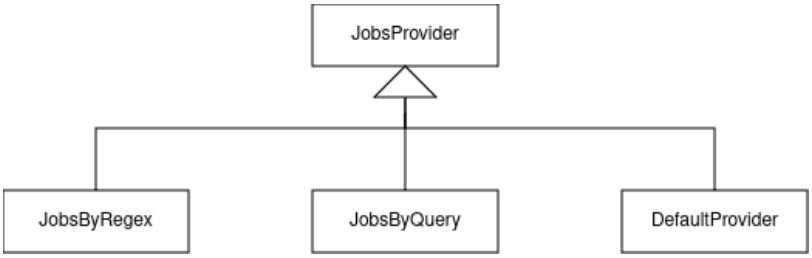

Figure 4.3: A simplified class diagram of classes associated with filtering the suites by their names.

In the `report-jtreg-comparator` module, there is also a class named `JobConfigFilter.java` that is only used for filtering the jobs by properties saved in their config files. For looking into config files and finding values in them, it uses a class from the `report-jtreg-lib` module called `ConfigFinder.java`.

This class takes the file to look into, the name of the value to find (so it can cache the value and not look for it in the filesystem multiple times during one run of the tool), and a query by which to find the value in the file (e.g. XPath) as arguments in its constructor. It also has a method called `findInConfig()` for finding the value and returning it. It currently supports finding in XML files by XPath, JSON files by a subset of JSONpath, and properties files by a key.

4.2.4 Getting and filtering the suite builds

The class for getting and filtering the builds is called `Builds.java`. There is a method called `getBuilds()` inside that gets the builds of a given job, filters them according to the filters, and returns them in a list. Apart from that, there are also methods for getting information about a single build, such as the build number or the name of a job to which this build belongs.

4.2.5 Comparing test failures

The main logic of this functionality is located in a `FailedTests.java` class. There are two public methods in this class. The first one is `createFailedMap()`, and it creates a hash map with builds and their respective tests that failed in the builds. The second one is named `printFailedTable()`, and it takes the hash map as an argument and creates a table of builds and their failed tests (with the `X` characters in it) and prints it using the chosen formatter.

This class gets the failed tests of a build using other classes from the `report-jtreg-lib` module, mainly `BuildSummaryParser.java` and `BuildReportExtended.java`. The functionality for filtering which tests to show in the table and the functionality to show only volatile tests is also implemented here.

4.2.6 Listing matched suites and builds

For this, there is a `JobsPrinting.java` class. It has two methods: `printJobs()` for printing all of the matched jobs and builds and `printVariants()` for printing and enumerating the matched jobs' variants. Both of these are meant to be used with some filters set.

4.2.7 Showing build result table

For printing a table of the builds' results (or a "virtual table" as it is called in the tool), there is a `VirtualJobsResults.java` class. The first method is private, and it is called `getBuildResult()`. It finds the result of the build in the builds' `build.xml` file (or another file if the user sets it differently than the default) using the `ConfigFinder.java` class from the `lib` module. If no result was found (null was returned), it marks the build with `RUNNING?` result, as it is most likely currently building.

The second method, `printVirtualTable()`, is public, and it creates the table from the builds' results it gets and prints it using a formatter.

4.2.8 Comparing test stack traces

The last main operation of this tool is located in a class called `StackTraceCompare.java`, and it has multiple important methods:

- `getTestTrace()` – A private method that gets a stack trace of a failed test of a build. As was already mentioned, every test can have multiple different outputs (like standard output, error output, etc.). In this method, a status line of a test together with every one of these outputs is taken and concatenated together. Since these outputs can be very long and the Levenshtein distance algorithm takes a lot of memory, there are four ways the user can choose to cut this output and make it shorter:
  - `Head` – Concatenate the outputs first and then take N characters from the beginning of the whole string.
  - `Tail` – Concatenate first and take N characters from the end.
  - `HeadEach` – Take the first N characters from each of the outputs and then concatenate them together into a string.
  - `TailEach` – Take the last N characters from the outputs and then concatenate them together.
- `getTraceSimilarity()` – This private method does the comparison of the similarity of two stack traces. It uses Levenshtein distance for it and converts it into percentages as was described in section 3.2.12. The Levenshtein distance algorithm is implemented according to the *similars* project.[1]
- `compareTraces()` A public method that takes the failed test hash map from the `FailedTests.java` class as an argument, creates a table with percentages representing the similarity of stack traces from it (using the other methods) and prints it to the user using a formatter.

4.2.9 Formatters

The formatters for printing output of this tool are located in the `jtreg-report-lib` module because they are, apart from the comparator, also used by the diff tool, meaning that they existed before work on the comparator was started. They were just extended. Specifically, they are all located in the `report-jtreg-lib/src/main/java/io/jenkins/plugins/report/jtreg/formatters` directory.

[1] https://github.com/judovana/similars

There are multiple classes in this directory, but the ones that are important for the comparator are as follows:

- `Formatter.java` – The interface class for the formatters. It defines methods for printing lines, adding styles, printing tables, generating headers for the tables, etc.
- `BasicFormatter.java` – This is an abstract class that implements the interface and defines functionality for the main methods like printing into the command line.
- `PlainFormatter.java` – A class with the plain formatting implementation. The methods for adding styles do nothing in this class. It extends the `BasicFormatter` class.
- `ColorFormatter.java` – A class with the color formatting – methods for adding styles print ANSI escape sequences for colors or other formatting like making the text bold. It extends the `BasicFormatter` class.
- `HtmlFormatter.java` – A class with the HTML formatting – methods for adding styles print HTML tags with styles. The method for printing a table does so using HTML tags. It also extends the `BasicFormatter` class.

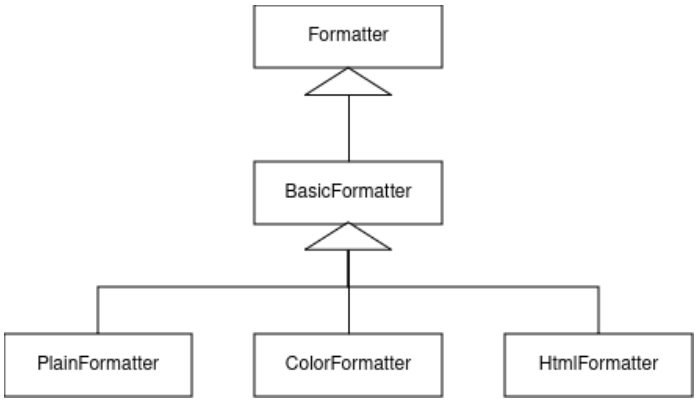

Figure 4.4: A simplified class diagram of classes associated with formatters.

4.2.10 Web interface and integration into Jenkins

The whole `report-jtreg-service` module works as a web interface for the comparator. The web interface was initially created for the diff tool, not the comparator, but it was reworked, and it now supports both tools.

As for the integration into Jenkins – the creation of links to a preset comparator, multiple classes deal with that in the `report-jtreg` module:

- `ConfigItem.java` – This class is meant for saving the information about job/build filters from config files the user can set in the Jenkins settings.
- `LinkToComparator.java` – This class is for saving information about a single link to the comparator. It saved data about the label, a spliterator for the job name, and the list of arguments (with unparsed dynamic macros).

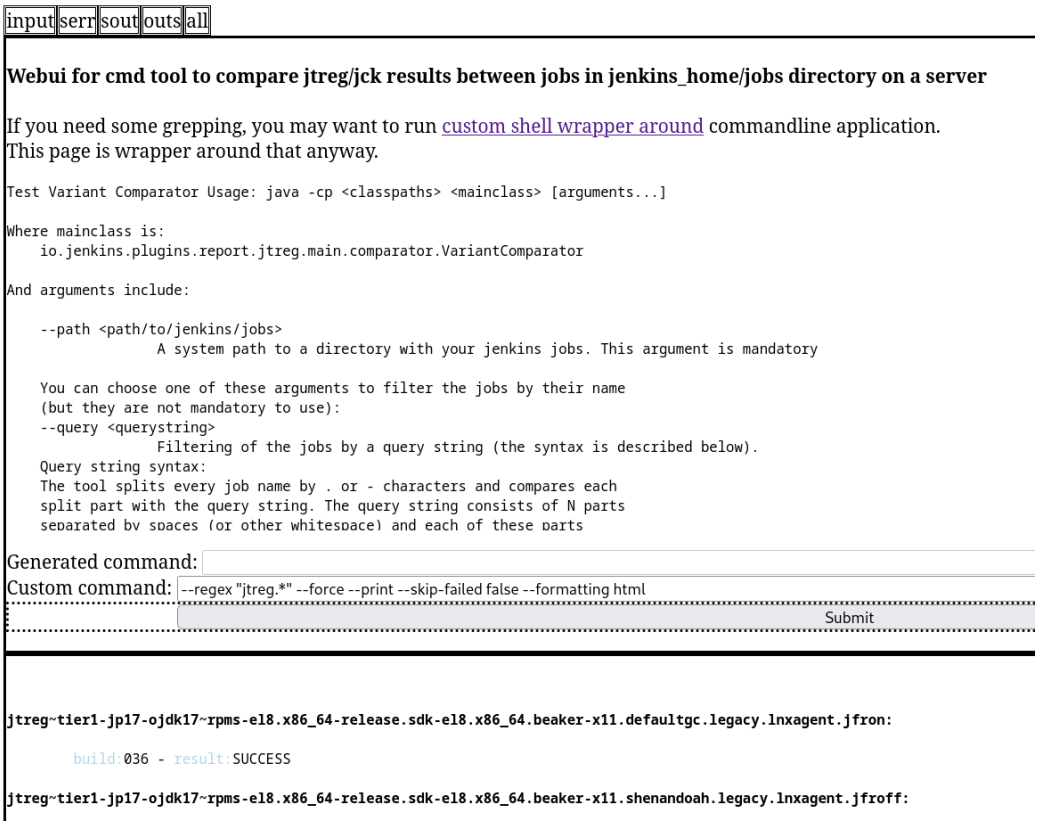

Figure 4.5: A screenshot of the comparator web wrapper.

- `ComparatorLinksGroup.java` – This class holds a list of individual links to the comparator as well as a regular expression, so the group of links is only shown on the jobs that match it.
- `JenkinsReportJckGlobalConfig.java` – This class is for getting user-given data from the forms in settings and creating individual classes from them.
- `BuildReportExtendedPlugin.java` – This is a class for the build report page that is generated for every build by the Jtreg plugin. It gets all of the groups of links that the user specified and checks if some of the groups' regex matches the name of the job this report belongs to. If so, it takes the links from this group, converts all of the dynamic macros to data for the specific build, and puts those links at the top of the report page.

4.3 Command line switches

In the table below, you can see every command line switch that the comparator supports, a type of value that is expected after the switch, and a text describing the argument (most of the descriptions are taken directly from the comparator's help message, which is located in the `HelpMessage.java` class).

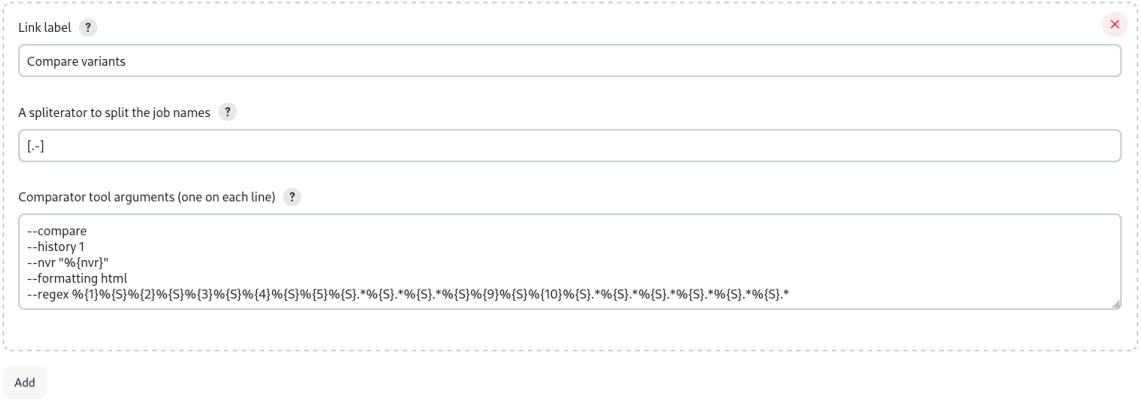

Figure 4.6: A screenshot of the form for adding links to the comparator.

Switch name	Expected value format	Description
`--path`	a system path to a directory	A system path to a directory with your Jenkins jobs. This argument is mandatory.
`--query`	a query string	Filtering of the jobs by a query string. The tool splits every job name by `.` or `-` characters and compares each split part with the query string. The query string consists of N parts separated by spaces (or other whitespace), and each of these parts corresponds with the 1st to the Nth part of the split job name.
`--exact-length`	a number	Meant to be used in combination with `--query`. It filters only the jobs that have this specified length (number of elements in its name).
`--regex`	a regular expression	The jobs will be filtered by the specified regex. Either `--query` or `--regex` can be used, not both together.
`--force`	–	Used for forcing vague filters (used with `--query` or `--regex`), that could potentially take a long time.
`--print`	–	Print all jobs and their builds that match the rest of the arguments without actually doing any operation on the builds or tests.
`--enumerate`	–	Print lists of all variants of jobs (that match the rest of the arguments).
`--compare`	–	Print a table of all failed tests (of matched job builds) and the builds where they failed.
`--list`	–	Print the same table as with `--compare`, but switch the rows and columns.
`--compare-traces`	–	Print a table of all failed tests (of matched job builds) with the similarities (in percentages, calculated with Levenshtein distance) of the stack traces to the referential build's (by default the left-most build in each table row) stack trace.

Switch name	Expected value format	Description
`--virtual`	–	Print a table of all matched jobs' builds and their result (e.g. `SUCCESS`, `UNSTABLE`, etc.). Can be used as a standalone operation or combined with any other operation. Probably should be run with `--skip-failed` set to `false`.
`--help`	–	Prints a help message.
`--skip-failed`	a boolean	Specify whether the comparator should skip failed tests (only take successful and unstable) or take all. The default value is true.
`--history`	a number	Specify the maximum number of builds to look into from every job.
`--only-volatile`	a boolean	Specify true to show only non-stable tests in the failed test table (shows only tests that are NOT failed everywhere). The default value is false.
`--formatting`	plain, color or html	Specify the formatting the output of the tool will have. The default is plain.
`--exact-tests`	a regular expression	Specify (with regex) the exact tests to show only. The rest of the tests will be ignored.
`--use-default-build`	a boolean	If set to true and no matching build with the given criteria was found, the tool will use the latest (default) build instead. The default value is false.
`--hide-passes`	a boolean	If set to true, when printing the compare table, only builds with at least one failed test will be shown. The rest will be hidden to make the table smaller. Set to false by default.
`--set-referential`	job name:build number	Use with `--compare-traces` operation if you want to set a different referential build than the left-most one in a row. Rows where the referential build passed (the test is successful) will be skipped.
`--cut-trace`	head, headEach, tail or tailEach:number of characters	Since Levenshtein distance consumes a lot of memory, long stack traces need to be cut. There are 4 options to cut them with this argument: `head` – concat all outputs and cut N characters from the start, `headEach` – if any test output is longer than N cut from the start from each, `tail`/`tailEach` – similar as head, but cut from the end. Default is `tailEach` and 5000 characters.

Switch name	Expected value format	Description
`--job-config-find`	config file name:name of the config item:query	This argument is used for declaring job/-suite filters. First, the name of a config file needs to be specified, then the name of the config item this filter is for (anything can be chosen), and then a query to define how to find the item in the config file. This query can be XPath for XML files, a subset of JSONpath for JSON files, or a key for properties files. After declaring this argument, you can use a new dynamic argument with the name you have chosen. It takes regex as a value, and it will filter the jobs by it. This argument can be used multiple times.
`--build-config-find`	config file name:name of the config item:query	Same as `--job-config-find`, but declares a filter for builds, not jobs.
`--dynamic`	a regular expression	Replace `dynamic` with the name of a config item declared through one of the two arguments above. An example of the dynamic arguments: `--build-config-find "changelox.xml:nvr/build/nvr"` was declared. Now, `--nvr "java-17-.*"` can be used to match only builds, that have `changelog.xml` file in their build directory and the value on XPath `/build/nvr` in that file matches `"java-17-.*`.

Chapter 5

Evaluating the Tool in Existing Environments

This chapter focuses on evaluating the usage and capabilities of this tool. It starts with an analysis of the tool on our infrastructure in the Red Hat OpenJDK QA team and then continues with an analysis on the infrastructure of the Eclipse Adoptium organization.

The main things it tries to dig into are the usability of the tool and its ability to save time by giving the users the features they are looking after.

5.1 Evaluating the tool in Red Hat

Since this tool was tailored to our infrastructure and needs, it was deployed on our server from the start of development. That means that the tool should work as intended in our infrastructure. This section focuses on how it is used and how my teammates can achieve tasks using it.

Since the Jtreg plugin was already in our infrastructure, adding the new features to it required only making a release and updating the plugin through Jenkins. On our internal server, there is also a systemd service that runs the web service with the comparator, and it has its own port from which it is accessible.

A simple methodology for testing was chosen – comparing how many web pages need to be visited in order to achieve a task (there are three tasks in this section) without and with the comparator tool. Comparing time may seem like a better methodology for this, however, it can be really misleading since every person has a different pace when achieving these tasks, and the speed of loading Jenkins' web pages and generating result tables with the comparator can also play a role in making it biased. That is why counting the number of visited web pages was chosen instead. The time difference between the comparator and non-comparator approaches can be easily deduced from these numbers.

Three different situations (based on real problems that can occur) in Red Hat's OpenJDK QA infrastructure were created to test whether this tool is useful and can help with the testing. The first situation covers a simple check of whether or not a test also failed on a different variant of a job or not. The second situation covers checking whether a test failed because of the same reason on other variants of a job or not (by calculating the similarity of their stack traces). Lastly, the third situation involves going into a history of a build and checking when a test started to fail.

These three situations cover all of the comparator's main functionalities that are actively used in Red Hat while testing the OpenJDK, so the testing should paint a convincing picture of whether or not the comparator works as expected in the Red Hat's infrastructure – which is the goal of this testing.

5.1.1 First situation – checking other OSs

In the first situation, there is a test suite called

`jtreg~tier1-jp17-ojdk17~portable-el7portable.x86_64-release.sdk-el7z.x86_64.vagrant-x11.defaultgc.legacy.lnxagent.jfroff`

(as the name of the suite suggests, it runs on Red Hat Enterprise Linux 7) with a test called

`runtime/os/TestHugePageDecisionsAtVMStartup.java#THP_enabled#TestHugePageDecisionsAtVMStartup_THP_enabled`

which failed in the 52nd build of this suite. The goal is to check whether or not the test also failed on other operating systems (at least RHEL 8 and Fedora 39) or not.

Without comparator

First, let's see how would this problem be tackled without using the comparator tool. Let's say that the starting point is on the report page (generated by the Jtreg plugin) of the 52nd build of this suite. On this report page, it is clearly shown, that the test failed.

To check the suite on another operating system, one of the easiest ways to do it manually is to take the suite name, change the value corresponding to OS, and use the search field in Jenkins to find the suite.

So, to get RHEL 8 and Fedora 39 versions of this suite, the `el7z` in the name has to be replaced either with `el8` or `f39` and searched through the search bar. The search result is a page of a suite, not of a build. So, on the page, the user needs to click the most recent build from the column on the left, and after the page of the build loads, they need to get to the Jtreg report by clicking a button.

Now, they are on the desired report page, where they can see that the test has not failed. It took loading and going through three web pages to get the information about one different OS, so it took six to get the information about both RHEL 8 and Fedora 39.

With comparator

With the comparator tool available, this task is much easier. The starting point is again the report page of the 52nd build, however, instead of manually rewriting the suite name to other OSs and checking the results manually, there is a link generated on this page called *Compare OSs*, which takes the user to a webpage with the result we are looking for.

So, with the comparator, this task took just one click and only a few seconds.

Results

As the comparator was designed mainly for tasks similar to this one, it excelled, and it was much quicker to get the results than by checking manually. Without the comparator, 7 pages needed to be loaded. With the comparator, only 2 – the initial report page and the page with the comparator results.

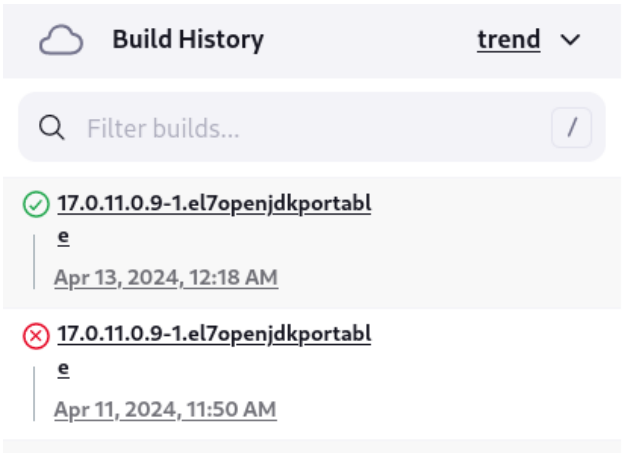

Figure 5.1: A screenshot of the left column on the job page with the two last builds shown.

Figure 5.2: A screenshot of the result table created by the comparator.

5.1.2 Second situation – checking if a test failed because of the same reason

This situation revolves around a test called

`compiler/uncommontrap/TestDeoptOOM.java#id0#TestDeoptOOM_id0`

which fails in

`jtreg~tier1-jp21-ojdk21~portable-el7portable.x86_64-fastdebug.sdk-el7z.x86_64.vagrant-x11.shenandoah.ignorecp.lnxagent.jfroff`

suite (in its build number 7) and its variant on different operating systems. The goal of this task is to check if the test failed on these variants (at least on RHEL 8 and 9) because of the same reason or not (involves checking the stack traces).

The starting point of this task is the report page of the mentioned suite.

Without comparator

To tackle this problem without a comparator, the process is similar to the process described in section 5.1.1.

Since the task starts on the report page of the first suite, the user can already see the first stack trace of the test. To see the stack traces on different operating systems, the user needs to change the suite name (change `el7z` to `el8z` and `el9z`) and search for this suite through the search bar. Then, they need to choose the latest build, open the build's page, and then go to the build's report page, where they can see the stack trace printed and check whether it was the same as on other variants or not.

This again takes 3 different pages to load for each variant, so 6 in total.

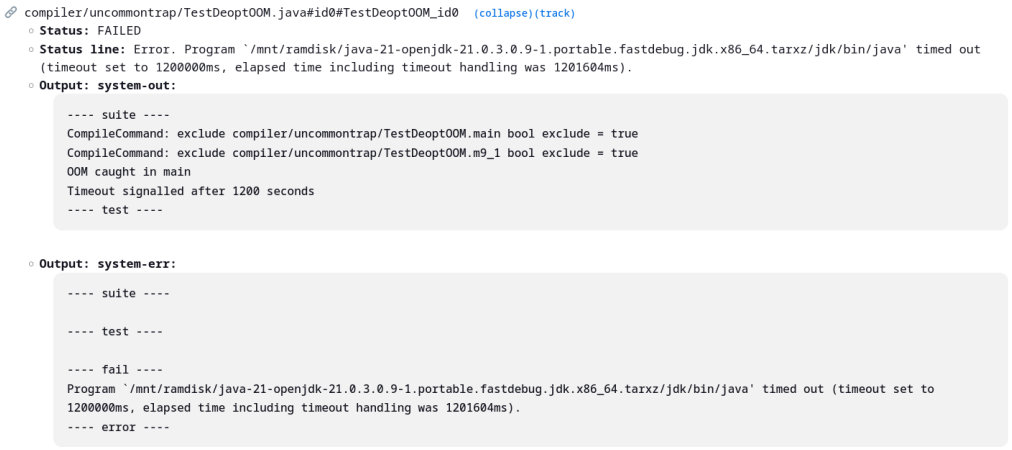

Figure 5.3: A screenshot of a stack trace of the test shown on the build's report page.

With comparator

To achieve this with comparator, the user can again just click the *Compare OSs* button on the first report page, and it takes them to the page with comparator. However, there is only a table of *where* the test failed (there are only `X`s in the table, not percentages), not *why*.

For this reason, the switch for operation needs to be changed in the input field there. After it is changed from `--compare` to `--compare-traces`, a table with the percentages representing similarities between the stack traces shows up. So, the user needed to visit 3 pages in total.

However, it is important to note that this step can be skipped by defining a new button in the Jenkins settings. This button can already exist directly for comparing stack traces. But since it is not defined in our infrastructure (at least not yet), the process was shown as it is now.

Results

The comparator is again much quicker and more efficient for this task. Without it, the user needs to visit 7 web pages in total, whereas with the comparator, only 3 (with editing a command line switch).

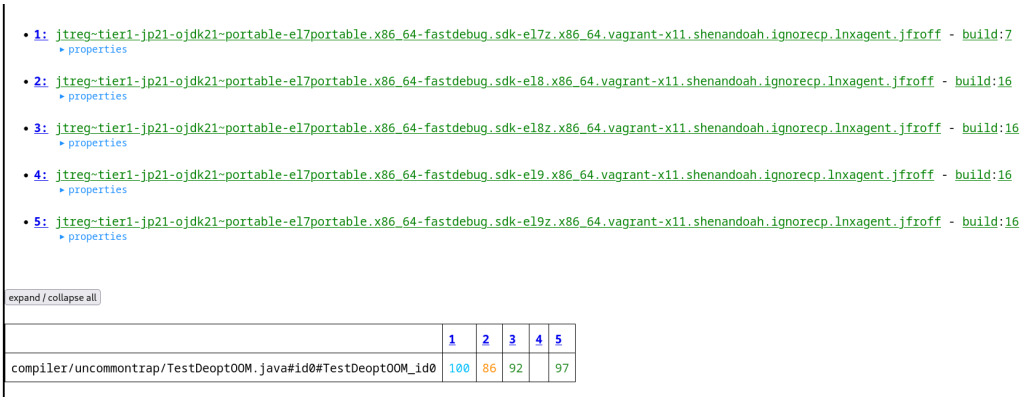

Figure 5.4: A screenshot of the result table with percentages created by the comparator.

It needs to be said that while checking the stack traces manually, the user can also see the specific stack trace, but when using the comparator, they can just see how similar the stack traces are. This is one of the things that this tool lacks (as of the time of writing this thesis), but it is one of the first features that are planned to be implemented in the future – comparing two stack traces.

5.1.3 Third situation – investigating when test began to fail

The last testing situation described in this section involves getting information about when a test started to fail. There is a test suite called

`jtreg~full-jp17-ojdk17~upstream~cpu-el7.x86_64-hotspot.release.sdk-el7z.x86_64.vagrant-x11.shenandoah.ignorecp.lnxagent.jfroff`

and in its 23rd build, a test called

`java/nio/channels/FileChannel/directio/DirectIOTest.java#DirectIOTest`

failed. The goal of this task is to determine in what build of the suite the test first started to fail. The task again starts on the build report page.

Without comparator

Without the comparator, the best way to tackle this problem would be to get to the page of the job from the report page. This can be done by searching for the job in the Jenkins search field or just editing the URL in a browser to cut the last part of it.

On this page, the user can choose any builds in the left column to open and inspect whether the test they are investigating failed or not.

To check the last 20 builds for this failing test, the user would need to visit 20+ pages and manually check for the test.

With comparator

To achieve the completion of this task with the comparator, the user can again just click a button (it can again be the *Compare OSs* button, but also any other since we only care about the current suite now, which is always in the comparison) and it takes them to a page with a generated table.

To show only relevant results, some pre-filled arguments need to be edited or added:

- `--regex ...` to `--regex jtreg~full-jp17-ojdk17~upstream~cpu-el7.x86_64-hotspot.release.sdk-el7z.x86_64.vagrant-x11.shenandoah.ignorecp.lnxagent.jfroff` – this is done to only show results of the suite, not other suites (as the link to the comparator intents)
- `--history 5` to `--history 20` – change the number of builds the comparator should look into
- add `--exact-tests java/nio/channels/FileChannel/directio/DirectIOTest.java#DirectIOTest` – this limits the table to just the one test that is needed to be investigated
- add `--skip-failed false` – do not skip failed builds, the user might be interested in them while solving this task

The correct results are shown to the user when submitting the form with the edited arguments. It took loading 3 pages and editing data in a form.

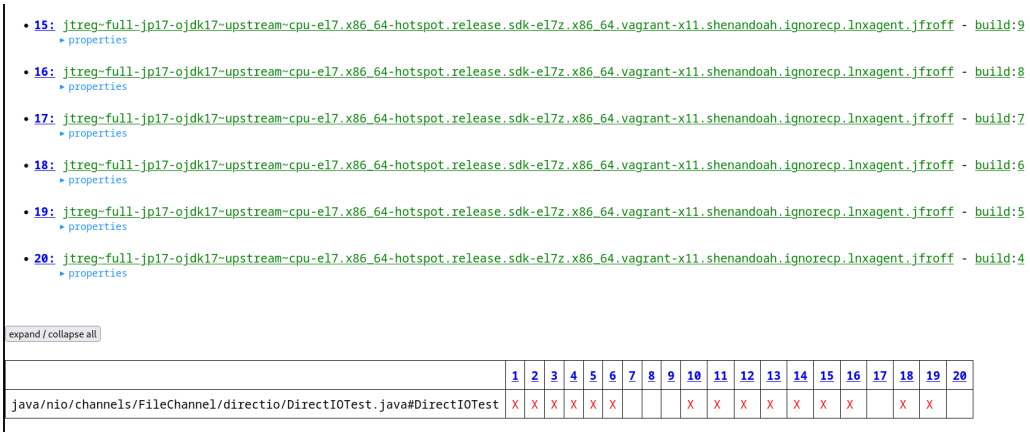

Figure 5.5: A (cut) screenshot of the result table created by the comparator.

Results

Once again, the comparator showed that it was suited for this task, and the user needed to visit much fewer web pages (3 with the comparator, 20+ manually) to get all the information they needed. It is also important to mention that if the buttons to comparator were set differently, they could achieve this task even without editing the command line arguments in the form (it is not currently necessary for our team because this task is not as common as other) and make the comparator approach even more efficient.

5.1.4 Concluding testing in Red Hat

In conclusion, the testing of this tool on OpenJDK QA infrastructure in Red Hat was a success. The tool was created for our infrastructure, so it suits it well and works as intended. The situations that were described in this section were all taken directly from real testing, and the comparator helped speed up all of them.

There are, of course, many more features of the comparator that were not looked at in this section. However, the main features (that were the most important for the comparator to be useful) were looked upon, and the result was great – the tool is capable of saving important time.

5.2 Evaluating the tool on Eclipse Adoptium results

Eclipse Adoptium is one of the organizations (their working group is composed of many companies, including Microsoft, Google, Red Hat, Canonical, or IBM) that provide prebuilt binaries of the OpenJDK. Their build of the OpenJDK is called Temurin. It is openly available, and it is currently one of the most popular builds.

This section describes their infrastructure for testing the Temurins, creating a local testing environment based on it (with test results downloaded directly from them), and testing whether the comparator works on it.

The evaluation of this tool on Adoptium's infrastructure/data is important for this thesis because one of the goals of this tool was to be as general as possible so it can be used in other infrastructures than Red Hat's. There is also an effort for Adoptium to adopt the Jtreg plugin, so the compatibility of the comparator should be tested. Adoptium also publishes test result archives of some builds of their jobs publicly on the internet, so it is easy to get example data from them.

The result of this testing should be proof that the comparator will work flawlessly on their infrastructure.

5.2.1 Adoptium testing infrastructure

Adoptium also uses Jenkins to run its builds and tests. Their Jenkins instance is available to be looked at by the public[1]. As opposed to Red Hat's OpenJDK infrastructure, their jobs (test suites) do not have names with every information about that job's configuration. Most of this is saved in a properties file every job and build contains.

In addition to normal jobs that run a set of tests (for example, Jtreg tests, JCK tests, etc.) and return the results normally, there is also a special job called *Grinder*. This job comes into play when other jobs fail, and a user wants to rerun the job. The job will be rerun with the same parameters (but it is also possible for the user to just rerun the failed tests, not the whole build), but it will not be under the original job. It will be under the Grinder job (the build of the Grinder will have a properties file and tags specifying the original job).

Apart from this, the infrastructure is very similar to Red Hat's. The jobs running Jtreg tests (and other compatible types of tests of the Jtreg plugin) normally generate archives with XML files with the test results, so the Jtreg plugin/comparator tool should work with it.

5.2.2 Creating local environment

To create the local environment, a fresh instance of Jenkins with the Jtreg plugin was installed. Then, 7 test jobs (based on real jobs that run Jtreg tests in Adoptium) were created:

[1] https://ci.adoptium.net/

- `Test_openjdk11_hs_sanity.openjdk_s390x_linux`
- `Test_openjdk11_hs_sanity.openjdk_x86-64_windows`
- `Test_openjdk17_hs_dev.openjdk_x86-64_linux`
- `Test_openjdk17_hs_dev.openjdk_x86-64_linux-alpine`
- `Test_openjdk17_hs_sanity.openjdk_aarch64_linux`
- `Test_openjdk22_hs_extended.openjdk_x86-64_linux`
- `Test_openjdk23_hs_sanity.openjdk_s390x_linux`

The pipeline of each of these jobs consists of downloading one of two (only one in some jobs to have them be stable) possible archives with test results (these results are downloaded directly from Adoptium in each run), copying a `testenv.properties` file with information about the build (the example data in these files correspond to the build's parameters like test type, Java version, architecture, specific operating system, etc.) and processing this with the Jtreg plugin.

An example of the shell script these jobs run for copying the environment properties file and downloading the archives can be seen here:

```
#!/bin/bash
mkdir jtreg

cp $HOME/testenvs/testenv-0.properties testenv.properties

if (( $RANDOM % 2 )) then
    VERSION=12
else
    VERSION=13
fi

wget https://ci.adoptium.net/view/Test_openjdk/job/
    Test_openjdk23_hs_sanity.openjdk_s390x_linux/$VERSION/artifact/
    openjdk_test_output.tar.gz -P jtreg/
```

For the simulation of the Grinder job, a job that runs a script that downloads a random result archive from a build of one of these 7 other jobs and copies the corresponding environment variable to it was created. This simulates a rerun of one of the other jobs on Grinder. The bash script (shortened to not take much space here) used for that can be seen here:

```
#!/bin/bash
mkdir jtreg

NUM=$((RANDOM % 7))

case $NUM in
0)
    TESTENV=0
```

```
    BUILD=Test_openjdk23_hs_sanity.openjdk_s390x_linux/13
    ;;
...
6)
    TESTENV=6
    BUILD=Test_openjdk23_hs_sanity.openjdk_s390x_linux/14
    ;;
esac

cp $HOME/testenvs/testenv-$TESTENV.properties testenv.properties
wget https://ci.adoptium.net/view/Test_openjdk/job/$BUILD/artifact/
    openjdk_test_output.tar.gz -P jtreg/
```

After setting the environment, each job was run a few times to download the results to the local system, and then the web service with comparator was set up according to the guide in section A.6. Now, the local environment is ready for the testing itself.

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Grinder	3 min 17 sec #13	26 min #1	2 min 13 sec
		Test_openjdk11_hs_sanity.openjdk_s390x_linux	3 min 6 sec #7	N/A	22 sec
		Test_openjdk11_hs_sanity.openjdk_x86-64_windows	2 min 44 sec #9	N/A	20 sec
		Test_openjdk17_hs_dev.openjdk_x86-64_linux	2 min 23 sec #7	N/A	2 min 15 sec
		Test_openjdk17_hs_dev.openjdk_x86-64_linux-alpine	11 min #7	N/A	1 min 10 sec
		Test_openjdk17_hs_sanity.openjdk_aarch64_linux	9 min 50 sec #7	N/A	12 sec
		Test_openjdk22_hs_extended.openjdk_x86-64_linux	9 min 15 sec #7	N/A	50 sec
		Test_openjdk23_hs_sanity.openjdk_s390x_linux	8 min 51 sec #6	N/A	14 sec

Figure 5.6: A screenshot of Jenkins with the local "Adoptium" jobs.

5.2.3 Testing the comparator

Two different approaches were used to test the comparator. One approach was to only compare (create a table of failed tests on multiple variants) normal jobs, and the second was to compare results from one job with its builds in the Grinder job.

Comparing test results from jobs

To check whether the comparator works on normal Adoptium jobs, it was tested by solving a task using the combination of filtering by the job names and the properties file that is in every build.

The task was to compare compiler failed tests of the last 3 builds of all jobs running on any version of Red Hat Enterprise Linux. To achieve this, these arguments of the comparator were used (the `--path` argument to the Jenkins jobs directory is already preset):

- `--compare` – the operation for creating the table of failed tests

- `--regex ".*linux$"` – for only getting the jobs that match this regular expression (jobs running on Linux)
- `--build-config-find "archive/testenv.properties:os:OS_SPECIFIC"` – for setting a build filter that looks into `archive/testenv.properties` file and finds the value with the `OS_SPECIFIC` key
- `--os "rhel"` – for only getting the builds that have `rhel` value with the `OS_SPECIFIC` key
- `--history 3` – for getting the 3 last builds
- `--exact-tests ".*compiler.*"` – for only matching and showing compiler tests
- `--formatting html` – for HTML formatting

As can be seen in pictures 5.7 and 5.8 and after cross-checking manually with the `testenv.properties` files and test results on Jtreg report pages, the matched jobs and their builds are correct, as well as the matched and shown tests. All of the results shown correspond to the manually checked data.

Figure 5.7: A screenshot of the matched jobs' builds.

	1	2	3	4	5	6	7
compiler/loopopts/TestRemixAddressExpressionsWithIrreducibleLoop.java#TestRemixAddressExpressionsWithIrreducibleLoop		X	X	X			
compiler/types/TestSubTypeCheckWithBottomArray.java#Xbatch#TestSubTypeCheckWithBottomArray_Xbatch		X	X	X			
compiler/types/TestSubTypeCheckWithBottomArray.java#Xcomp#TestSubTypeCheckWithBottomArray_Xcomp		X	X	X			
compiler/types/TestSubTypeCheckWithBottomArray.java#stress#TestSubTypeCheckWithBottomArray_stress		X	X	X			

Figure 5.8: A screenshot of the table of failed compiler tests.

To expand on this task, the comparator was also tried with `--compare-traces` argument instead of `--compare` to check the similarity of the test stack traces. The result of this was the same table as in 5.8, but instead of `X`s, there were only percentages. Specifically, there were only `100`s in the table. And indeed, after checking the test stack traces through the Jtreg plugin, they were all the same.

The last thing that was checked was creating a table with the results of the whole builds with the `--virtual` argument. This generated a table that can be seen in image 5.9, and after checking the build results through Jenkins, a conclusion can be made that also this table was generated correctly.

	1	2	3	4	5	6	7
SUCCESS					X		
UNSTABLE	X	X	X	X		X	X
FAILURE							
ABORTED							
NOT_BUILT							
RUNNING?							

Figure 5.9: A screenshot of the table of build results.

Comparing results from jobs with Grinder

Now, in this section, there is a simulation of the comparison of results from jobs with their equivalent builds in Grinder. To do that, a job called `Test_openjdk11_hs_sanity.openjdk_s390x_linux` needs to be taken, and its builds' `testenv.properties` file needs to be looked at. Based only on the properties in the file (using no filters by job name), the last 3 builds of the job should then be matched. This should only get the original job's builds and some Grinder's builds.

These comparator arguments were used:

- `--print` – setting the comparator operation to only print the matched jobs with their builds (this testing situation is more about matching the correct builds rather than comparing the results)
- **build config file filters** – I used 3 build config file filters for the testing because 3 were enough to match only the original job's builds. However, with more jobs, more

would probably need to be used (or the matching could be done by some tag in some config file the build's directories contain).

  - `--build-config-find "archive/testenv.properties:suite:SUITE_TYPE"` – for filtering by the test suite type
  - `--build-config-find "archive/testenv.properties:version:JAVA_VERSION"` – for filtering by the Java version
  - `--build-config-find "archive/testenv.properties:os:OS"` – for filtering by the operating system

- **values to those filters** – I used these values to be matched using the filters:
  - `--suite "sanity"`
  - `--version "11"`
  - `--os "linux"`

- `--history 10` – Even though only the 3 last builds are needed, the number of builds needs to be set to a larger number because there are also other job's builds in the Grinder, and to find the 3 from the original job, the tool needs to search more in the job's build history.

- `--formatting html` – for HTML formatting

As can be seen in picture 5.10, the builds that these commands matched are the ones that were expected. It matched builds from the original job as well as builds of this job in Grinder (these builds were also manually checked in Jenkins to be sure they were really of the original job).

The only setback to this is the fact that the `--history` argument needs to be used with a larger value than 3, even though getting only the last 3 builds is enough. That is because Grinder has builds of multiple jobs, and there is no clear way to tell what builds are of which jobs before filtering them, and the comparator only looks at the last N builds (based on the `--history` argument) regardless of whether it matches the filters or not – the matching is done after getting the builds.

5.2.4 Concluding testing on Adoptium's data

To sum up the testing on Adoptium's data, it would be fair to say it was a success. The tool (with the Jtreg plugin) works pretty much "out of the box" on the data directly taken from Adoptium, and the different structure of jobs does not seem to be an issue.

The tool was capable of doing the main operations on the normal Adoptium jobs as well as in combination with the Grinder job, and in my opinion, it can also be useful in their infrastructure and save them time.

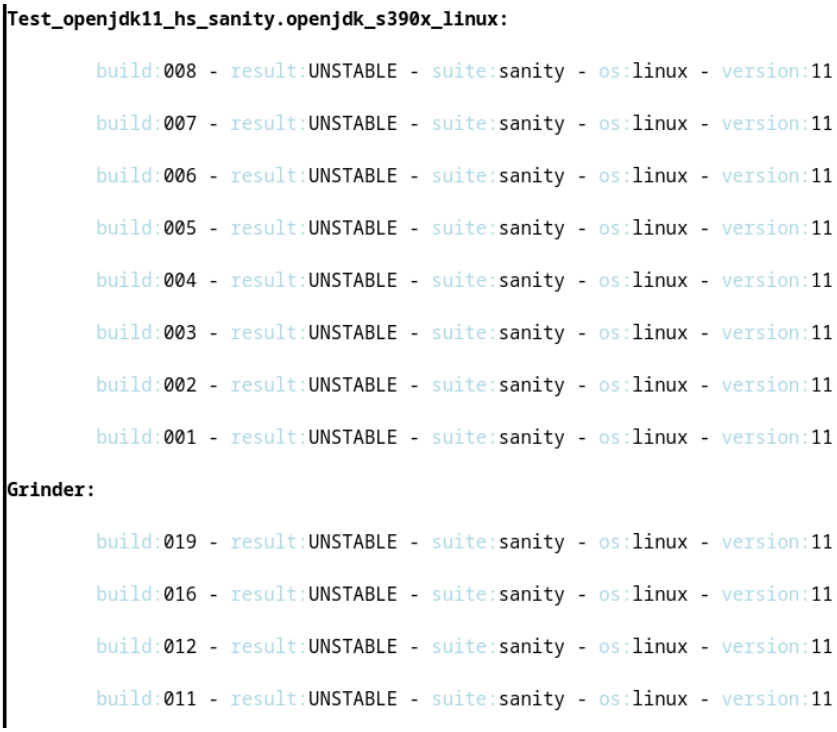

Figure 5.10: A screenshot of the list of matched builds with the Grinder job.

Chapter 6

Conclusion

In this thesis, I designed and created a tool for comparing the results of tests on multiple variants of a single suite, such as different operating systems, architectures, configurations, etc.

There were many challenges along the way, mainly in the tool's design stage, since I needed to develop an ideal way of showing the tool's results to the user from the ground up. Fortunately, this tool was continuously deployed in our infrastructure with every major change, so most design flaws were captured right in the beginning without being dealt with in later stages.

The tool's implementation was done according to the plan, which is also described in this thesis. The tool is programmed in Java, and it can be used independently through a terminal or a special web interface, which works as a wrapper around the tool.

This tool was mainly meant for Red Hat's OpenJDK QA team, which I am a part of, but it was created with the usage elsewhere also in mind. Because of that, the tool is very generic (there is a minimum of infrastructure-specific functions there), and it is ready to be used in other teams that test the OpenJDK or even other projects with a similar test structure (the Jtreg plugin of which the comparator is a part of, supports archives of XML test results from JUnit/xUnit, Jtreg, and JCK).

The tool was tested on Red Hat's OpenJDK QA test results (it was directly tested in the infrastructure) and on the test results from Eclipse Adoptium. This testing was a success – the tool worked as intended on both.

To sum up, I would say that the tool's design and implementation were successful since it is already running in our infrastructure and is actively used by my colleagues to check the results of the OpenJDK tests. The plugin with this tool has also been published as an open source project on GitHub[1] and is in the Jenkins' plugins repository[2].

[1] https://github.com/jenkinsci/report-jtreg-plugin
[2] https://plugins.jenkins.io/report-jtreg/

Bibliography

[1] Armenise, V. Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. Florence: IEEE, May 2015, p. 24–27. ISBN 978-1-4673-7070-7. Available at: https://doi.org/10.1109/RELENG.2015.19.

[2] IBM Corporation. *IBM i 7.5 Documentation: Java classpath* online. IBM Corporation, october 2023. Available at: https://www.ibm.com/docs/en/i/7.5?topic=usage-java-classpath. [cit. 2024-04-07].

[3] Jamil, M. A.; Arif, M.; Abubakar, N. S. A. and Ahmad, A. Software Testing Techniques: A Literature Review. In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. Jakarta, Indonesia: IEEE, November 2016, p. 177–182. ISBN 978-1-5090-4521-1. Available at: https://doi.org/10.1109/ICT4M.2016.045.

[4] Jenkins documentation contributors. *Jenkins User Documentation: What is Jenkins?* online. Jenkins. Available at: https://www.jenkins.io/doc/. [cit. 2024-03-25].

[5] Kasko, A.; Kobylyanskiy, S. and Mironchenko, A. Testing OpenJDK. In: *OpenJDK Cookbook*. 1st ed. Birmingham, UK: Packt Publishing, January 2015, p. 165–202. ISBN 978-1-84969-840-5.

[6] Leung, H. K. N. and White, L. A study of integration testing and software regression at the integration level. In: *Proceedings. Conference on Software Maintenance 1990*. San Diego, USA: IEEE, November 1990, p. 290–301. ISBN 0-8186-2091-9. Available at: https://doi.org/10.1109/ICSM.1990.131377.

[7] OpenJDK website contributors. *OpenJDK* online. Available at: https://openjdk.org/. [cit. 2024-03-26]. Path: Home.

[8] OpenJDK website contributors. An Introduction to jtreg. *OpenJDK* online. Available at: https://openjdk.org/projects/code-tools/jtreg/intro.html. [cit. 2024-04-06]. Path: Home; jtreg harness; An Introduction to jtreg.

[9] Oracle Corporation. Critical Patch Updates, Security Alerts and Bulletins. *Oracle* online. Available at: https://www.oracle.com/security-alerts/. [cit. 2024-04-03]. Path: Home; Resources; Critical Patch Updates;.

[10] Parsons, D. The Java Story. In: *Foundational Java: Key Elements and Practical Programming*. 2nd ed. Cham, Switzerland: Springer International Publishing,

September 2020, p. 1–10. Texts in Computer Science. ISBN 978-3-030-54518-5. Available at: https://doi.org/10.1007/978-3-030-54518-5_1.

[11] Parsons, D. Unit Testing with JUnit. In: *Foundational Java: Key Elements and Practical Programming*. 2nd ed. Cham, Switzerland: Springer International Publishing, September 2020, p. 251–278. Texts in Computer Science. ISBN 978-3-030-54518-5. Available at: https://doi.org/10.1007/978-3-030-54518-5_10.

[12] Søndergaard, H.; Korsholm, S. E. and Ravn, A. P. A Safety-Critical Java Technology Compatibility Kit. In: *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, USA: Association for Computing Machinery, October 2014, p. 1–9. JTRES '14. ISBN 978-1-4503-2813-5. Available at: https://doi.org/10.1145/2661020.2661021.

[13] Vaněk, J. FindDupes.java. *Similars GitHub repository* online. 14. november 2023. Available at: https://github.com/judovana/similars/blob/main/FindDupes.java. [cit. 2024-04-07]. Path: Repository home; FindDupes.java.

[14] Wikipedia contributors. Levenshtein distance. *Wikipedia, The Free Encyclopedia* online. 23. february 2024. Available at: https://en.wikipedia.org/wiki/Levenshtein_distance. [cit. 2024-04-07].

[15] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0* online. Edited by Clark, James and DeRose, Steve. World Wide Web Consortium, november 1999. Available at: https://www.w3.org/TR/xpath-10/. [cit. 2024-04-17].

[16] Zhang, S.; Hu, Y. and Bian, G. Research on string similarity algorithm based on Levenshtein Distance. In: *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. Chongqing, China: IEEE, March 2017, p. 2247–2251. ISBN 978-1-4673-8979-2. Available at: https://doi.org/10.1109/IAEAC.2017.8054419.

Appendix A

Installing the Tool

The guide in this section assumes you are using Debian 12. However, if you edit some of the commands, it should also work correctly on other Linux distributions. Even though there are pre-compiled binaries in the `binaries` directory, this guide assumes that you will compile the code yourself. If you want to use the pre-compiled binaries, you can do so, but you will have to edit the paths in some commands.

A.1 Install OpenJDK 17 and Maven

Other OpenJDK versions (the more recent ones, to be precise) might also work, but I recommend using 17. If you are using Debian, you can install both packages by running:

```
sudo apt install openjdk-17-jdk maven
```

A.2 Download and install Jenkins

There are multiple ways to download and use Jenkins. However, I will focus on running the Jenkins from the `.war` file you can download from https://www.jenkins.io/download/ (I recommend using the current LTS version that is listed there).

After you download the file, you can run it with this command (assuming you are in the same directory as the downloaded file and the file is called `jenkins.war`):

```
java -jar jenkins.war
```

Then, you can open the webpage `localhost:8080` in your browser and enter the password that is shown in the command line output.

After that, you can follow the instructions on the web page:

1. **Install suggested plugins** – wait for them to install
2. **Create first admin user** – you can either enter the data or click the "Skip and continue as admin" button (and use the initial password from the terminal earlier)
3. **Instance configuration** – you can keep the default or choose a different address
4. **Jenkins is ready!** – click the *Start using Jenkins* button

A.3 Compile and install the tool and plugin

First, extract the source code (it is located in the `src` directory) from this thesis into a directory and navigate to it through the terminal. (Note that you can use the upstream code from https://github.com/jenkinsci/report-jtreg-plugin. However, you must edit and set some things yourself – especially in the `report-jtreg-service` module. I will assume you are using the code that was submitted with this thesis.)

Now, you can compile the plugin and comparator by running:

```
mvn clean install
```

When running this for the first time, all of the dependencies are downloaded, so it might take some time.

A.3.1 Install the plugin

On your local Jenkins instance, navigate into `Manage Jenkins`, `Plugins` and `Advanced Settings`.

Here, under the *Deploy Plugin* header, you can click the *Browse* button, navigate into the source code directory, and then navigate into `report-jtreg/target`. Choose the `report-jtreg.hpi` file and click the *Deploy* button.

On the next page, you can check the box to restart Jenkins when the installation is done. After Jenkins restarts, the plugin should be installed.

A.4 Installing test jobs

In the archive that was submitted with this thesis, there is also a directory called `test-jobs`, which contains some empty (without any builds) example jobs you can use for trying the tool without compiling and testing the OpenJDK itself.

If you have not changed the default Jenkins directory, you can install those test jobs by running this command after navigating into the directory with the test jobs:

```
cp -r * $HOME/.jenkins/jobs/
```

Now, when you open the Jenkins homepage (it might need to be restarted), you should see the test jobs added.

However, these jobs do not contain any results. You have to supply them yourself (the archives with results are very large, so none were attached with this thesis to save space). You can get some archives with compatible results, for example, from https://ci.adoptium.net/view/Test_openjdk/ – unstable builds should have a *tar.gz* file with the test results.

The example jobs then expect exactly three archives located in the user's home directory: `results-0.tar.gz`, `results-1.tar.gz` and `results-2.tar.gz` – you can download some test archives from Adoptium, put them into your home directory and rename them to this (or change the jobs' configurations to take other archives, if you have different ones).

Now, you can run each job multiple times and then proceed to test the comparator tool itself.

A.5 Running the comparator from CLI

You are now ready to run the comparator tool from the CLI. To do so, you first need to navigate to the source directory with the compiled Jtreg plugin, and then you can run the comparator by entering:

```
java -cp report-jtreg-comparator/target/report-jtreg-comparator.jar:\
report-jtreg-lib/target/report-jtreg-lib.jar:\
$HOME/.m2/repository/com/google/code/gson/gson/2.10.1/gson-2.10.1.jar \
io.jenkins.plugins.report.jtreg.main.comparator.VariantComparator \
--help
```

If you followed this guide, this should work out of the box and print a help prompt. However, if it does not, chances are that the path to the `gson` jar file (a dependency of this tool) is different, and you need to change it.

Now, you can change the `--help` argument to other arguments and use the tool as was described in this thesis.

A.6 Setting the web service

To start the web service for running the comparator, you first need to change the permissions to some files by running this (assuming you are located in the parent directory of the source code in a terminal):

```
chmod +x run_comparator_tool.sh && chmod +x run_diff_tool.sh
```

And then, you can start the web service itself by running this from the compiled source code directory:

```
java -cp report-jtreg-service/target/report-jtreg-service.jar:\
report-jtreg-comparator/target/report-jtreg-comparator.jar:\
report-jtreg-diff/target/report-jtreg-diff.jar \
io.jenkins.plugins.report.jtreg.main.Service \
run_diff_tool.sh run_comparator_tool.sh 9090
```

This runs on port 9090, so you can visit it by entering `localhost:9090/comp.html` into your browser and try it for yourself. The path to the Jenkins jobs directory is already set, so you do not need to set the argument yourself. If this does not work or throws an exception in the bottom field on the website, the `run_comparator_tool.sh` file located in the top directory of the source code may be an issue, and you need to edit the file (to change the paths to dependencies or the `--path` argument to the directory with Jenkins jobs).

However, if you have followed this guide, this should work. Now, you can set links to the comparator in Jenkins and simulate the workflow in a real infrastructure.

Appendix B

Extending the Tool

B.1 A place in the code to extend

If you find yourself in a situation where you use this tool but lack some features, you can easily extend it yourself. Since the tool is programmed in Java, the extension should be pretty straightforward.

If you are reading this, you will most likely want to extend the features of the comparator. The comparator is located in the `report-jtreg-comparator` module, and its relevant classes are all described in the chapter 4, so you can take a look at that chapter and see where your new feature may fit.

Other modules that are part of this project as a whole you may be interested in extending are `report-jtreg-lib`, where some common classes other modules use are, `report-jtreg`, where the classes that extend the functionality of Jenkins are, or `report-jtreg-service`, where the classes used for running the comparator as a web service are.

Every modern Java IDE should work correctly with a multi-module project and guide you on where to look for the classes, what packages they belong to, and what dependencies they have. It is also important to mention that the tool (and the whole plugin) uses Apache Maven as its build system, so some basic knowledge of how to use it is needed.

B.2 Contributing to the tool

Since this tool is open source and licensed under the MIT license, contributions are welcome, whether you extend it with new functionality or fix bugs.

The repository with the code is located here `https://github.com/jenkinsci/report-jtreg-plugin`, and you can either open an issue or a pull request with your code. Both would be appreciated.