239 Project A

1. INTRODUCTION

The aim of this project is to automate analysis of software profile and to perform exploratory analysis on it. We have taken 2 software projects for this analysis.

1. Apache Commons Lang

Standard Java libraries fail to provide enough methods for manipulation of core classes. Apace Commons Lang provides these extra methods. It provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals.

2. JLine

JLine is a Java library for handling console input. It is similar in functionality to BSD editline and GNU readline. People familiar with the readline/editline capabilities for modern shells (such as bash and tcsh) will find most of the command editing features of JLine to be familiar.

We chose Apache Commons Lang as it is quite a stable and mature project with over 70 releases. We considered this as the base project to develop and test our analysis automation.

After development and test, we chose JLine data set to run and validate the results on.

Each of these projects were compatible with Maven and came with an extensive set of test suites. The tests were in the form of JUnits.

2. PROJECT STATISTICS

STATISTICS	APACHE COMMONS LANG	JLINE
NUMBER OF TESTS RUN	3733	141
PASSED TESTS	3733	141
FAILED TESTS	0	0
NUMBER OF SOURCE FILE CLASSES	136	53
VERSION NUMBER	3.4	2.x
RELEASES	70	15
NUMBER OF TEST FILES	157	22
TIME TAKEN	32.261 sec	22.744 sec

3. PROCESS

We initially compiled the .java files in the project to generate the .class files which need to be given as input to the ASM byte code manipulation tool-kit. We then added instrumentation messages before and after method invocation calls, by extending ASM's in-built ClassVisitor and MethodVisitor classes to create custom visitors.

We then created the new .class files containing this instrumented byte code. On executing these newly instrumented files, we wrote the output to a text file to create the calling context tree representation for the tests.

We automated the above process by using shell scripts.

```
Call begin] java/lang/System::getProperty
Call end] java/lang/System::getProperty
Call end] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] java/lang/System::getProperty
Call end] java/lang/System::getProperty
Call end] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] java/lang/System::getProperty
Call end] java/lang/System::getProperty
Call end] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] java/lang/System::getProperty
Call end] java/lang/System::getProperty
Call end] org/apache/commons/lang3/SystemUtils::getSystemProperty
Call begin] org/apache/commons/lang3/JavaVersion::get
Call begin] org/apache/commons/lang3/JavaVersion::<init>
Call begin] java/lang/Enum::<init>
```

Calling context tree

Once the above file is generated, we parse it to generate a new file with annotated spectra information.

Our processing involved using a stack and a hashmap to represent and annotate the nested method calls.

We represent the method calls as "Nodes" where a Node consists of

a) Class Name b) Function Name c) Count

```
<0>:java/lang/Byte::valueOf (1)
<0>:java/lang/Byte::valueOf (1)
<0>:org/apache/commons/lang3/Range::between (1)
        <1>:org/apache/commons/lang3/Range::between (1)
                <2>:org/apache/commons/lang3/Range::<init> (1)
                        <3>:java/lang/Object::<init> (1)
                        <3>:java/util/Comparator::compare (1)
                                <4>:java/lang/Comparable::compareTo (1)
<0>:java/lang/Integer::valueOf (1)
<0>:java/lang/Integer::valueOf (1)
<0>:org/apache/commons/lang3/Range::between (1)
        <1>:org/apache/commons/lang3/Range::between (1)
                <2>:org/apache/commons/lang3/Range::<init> (1)
                        <3>:java/lang/Object::<init> (1)
                        <3>:java/util/Comparator::compare (1)
                                <4>:java/lang/Comparable::compareTo (1)
```

Annotated Context Calling tree

Once this file is created, we started collecting telemetry information from it.

We used Java in-built data structures such as HashMaps to record the pertinent information. The following information is captured in HashMaps and subsequently written to files:

a) Individual method call count

b) k-length call stack patterns, where k is user defined input

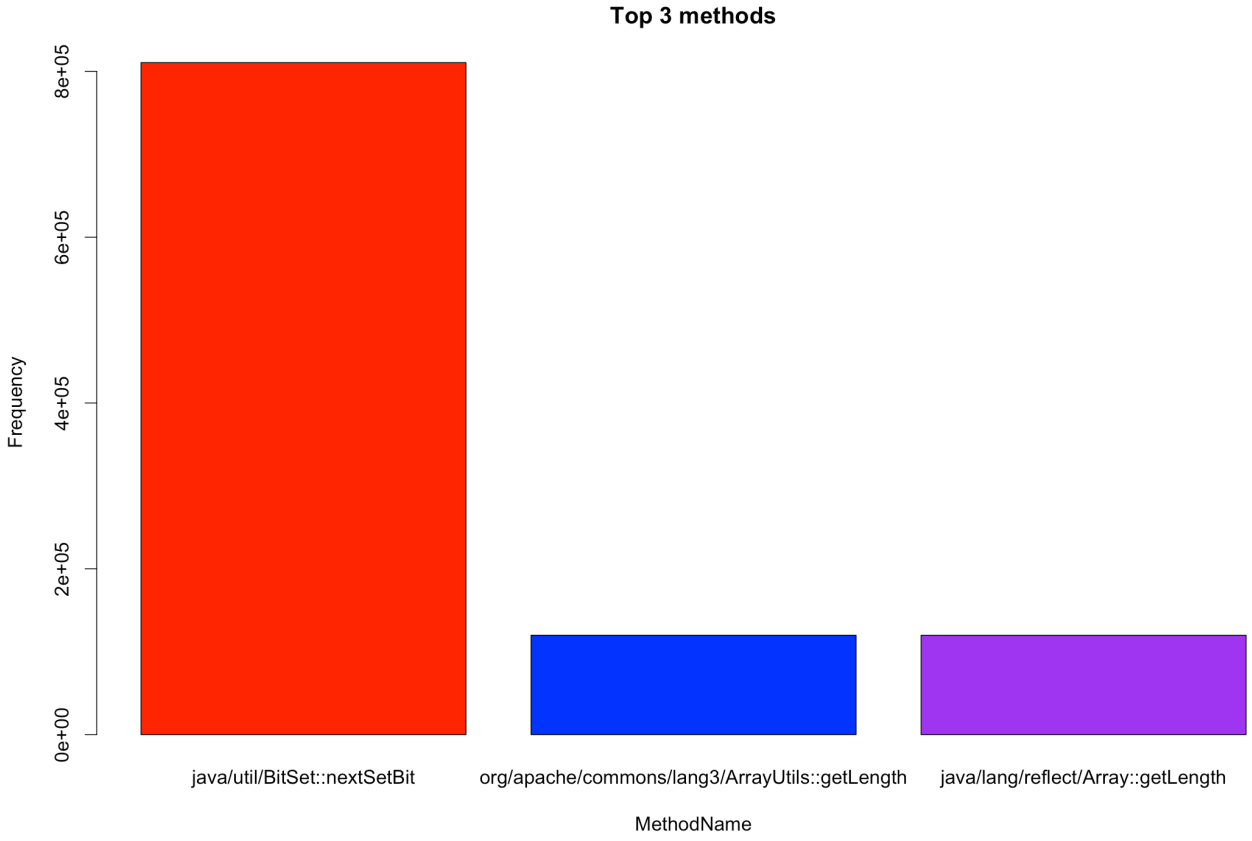
In our chosen projects, there were no failed test cases, but our code is able to identify the number of failed test cases and to report them.

4. EXPLORATORY QUESTIONS

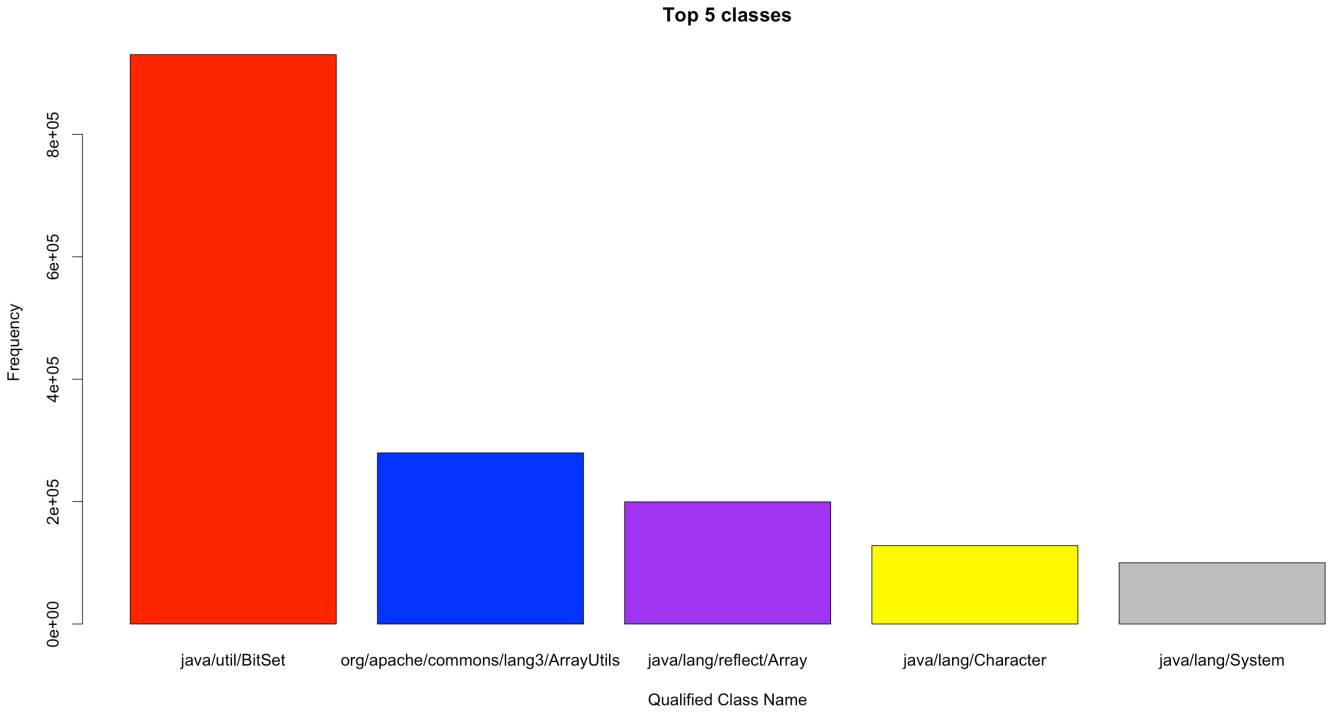
We then process and analyze the extracted metrics to answer a few basic questions related to the software profile of the project.

Q1. What are the top 3 most frequently called methods?

To answer this, we read the csv file in R, sort the methods in order of decreasing frequency and extract and plot the top 3 in the form of a bar plot as shown below:

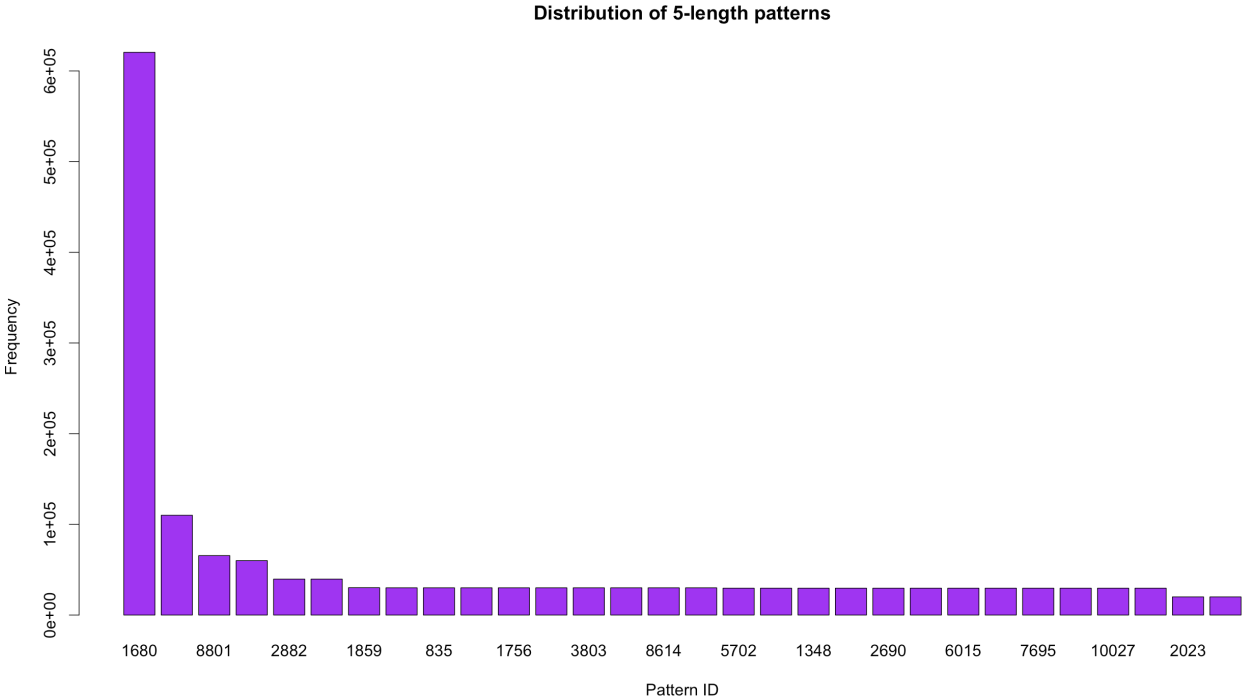


Q2. What are the top 5 most frequently used Classes?



To answer this, we extracted the fully qualified class name and analyzed it to extract the top 5.

Q3.What are the top 30 most frequent 5-length(i.e. k=5) call patterns?



For verbosity purposes, we enlist only the top 5-length sequence below:

```
> s[1]
[1] "org/apache/commons/lang3/Validate"
> s[2]
[1] "inclusiveBetweenorg/apache/commons/lang3/HashSetvBitSetTest"
> s[3]
[1] "printTimesorg/apache/commons/lang3/HashSetvBitSetTest"
> s[4]
[1] "timeExtractRemoveAllorg/apache/commons/lang3/HashSetvBitSetTest"
> s[5]
[1] "extractIndicesjava/util/BitSet"
```

5.CONCLUSION

Hence we have automated analysis of the software profiles and created generic scripts which can be applied to any software project. The analysis in R is also done via scripts which can automate analysis for any k-length call stack pattern.