DELIVERABLE ONE REPORT

Yuer Shen (SID: 11421959)

Ran Tao (SID: 11488080)

Courtney Snyder (SID: 11463078)

CptS 422 Software Engineering Principle II

Spring 2017

Instructor: Venera Arnaoudova

STRUCTURAL METRICS

Our team was chosen to implement structural metrics. The structural metrics all seemed fairly reliant on another function like length or number of operands, so as a team, we decided to metrics that were related to each other and minimize the number of outside metrics to implement. We found six metrics that fit this criteria; operator count, operand count, length, volume, difficulty, and vocabulary.

For declaration, we defined that operators are TokenTypes."ASSIGN, BAND, BAND_ASSIGN, BNOT, BOR, BOR_ASSIGN, BSR, BSR_ASSIGN, BXOR, BXOR_ASSIGN, COLON, COMMA, DEC, DIV, DIV_ASSIGN, DOT, EQUAL, GE, GT, INC, INDEX_OP, LAND, LITERAL_INSTANCEOF, LNOT, LOR, LT, MINUS, MINUS_ASSIGN, MOD, MOD_ASSIGN, NOT_EQUAL, PLUS, PLUS_ASSIGN, POST_DEC, POST_INC, QUESTION, SL, SL_ASSIGN, SR, SR_ASSIGN, STAR, STAR_ASSIGN, UNARY_MINUS, and UNARY_PLUS and operands are TokenTypes."NUM_INT and NUM_FLOAT".

Ran: implement the operator count and operand count metrics
Yuer: implement the length and volume metrics.
Courtney: implement the vocabulary and difficulty metrics.

All team members wrote their own implementations of their metrics. Ran used the Checkstyle definition of operators and operands to implement his metrics. He also did a lot of research with Checkstyle Plugins, so he taught Yuer and Courtney how to set up the project and how to write their own Checkstyle checks.

WORK PROCESS

All team members wrote their own implementations of their metrics. Ran used the Checkstyle definition of operators and operands to implement his metrics. He also did a lot of research with Checkstyle Plugins, so he taught Yuer and Courtney how to set up the project and how to write their own Checkstyle checks.

We quickly realized that since many metrics depend on each other, this would be a very involved project. Since vocabulary and difficulty depend on operator count and operand count metrics, Courtney waited for Ran to implement those metrics.

After looking at the implementation of the operator and operand count metrics, Courtney and Ran decided to adjust their original implementations by having them return dictionaries with the operator or operand as the key and the number of occurrences as the value. This greatly helped with Courtney's implementation of the vocabulary and difficulty metrics since both require number of unique operators and unique operands, or dictionary entries with a value of 1.

Similarly, volume depends on vocabulary, so Yuer had to wait for Courtney to finish her implementation. Again, Yuer and Courtney looked at the implementation and decided to adjust the original implementation by adding a function that separated the vocabulary calculation from the tokenization of the DetailAST. This helped with Yuer's implementation of the unit tests because it removed some cohesion from the functions and made them easier to individually test.

UNIT TEST

Yuer implemented our unit tests. Since all of our functions are very dependent on each other, she ran into difficulty separating the functions. For example, length depends on number of operators and number of operands, so it was hard to test just length without integration testing.

For the Unit test, Yuer use PowerMockito.whenNew to create fake value. She noticed that the older power Mockito version that professor Venera suggest to implement for our project doesn't support that method; when we run the PowerMockito with the 1.6.5 version, we got the "initializationError". In order to fix the Junit test error, we import 1.7.1 version of PowerMockito.

```
java.lang.IllegalStateException: Failed to transform class with name net.sf.eclipsecs.sample.checks.UnitTestY. Reason: java.io.IOException: invalid constant type: 15
at org.powermock.core.classloader.MockClassLoader.loadMockClass(MockClassLoader.java:267)
at org.powermock.core.classloader.MockClassLoader.loadModifiedClass(MockClassLoader.java:180)
at org.powermock.core.classloader.DeferSupportingClassLoader.loadClass(DeferSupportingClassLoader.java:70)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Unknown Source)
at org.powermock.modules.junit4.common.internal.impl.JUnit4TestSuiteChunkerImpl.createDelegatorFromClassloader(JUnit4TestSuiteChunkerImpl.java:145)
at org.powermock.modules.junit4.common.internal.impl.JUnit4TestSuiteChunkerImpl.createDelegatorFromClassloader(JUnit4TestSuiteChunkerImpl.java:40)
at org.powermock.tests.utils.impl.AbstractTestSuiteChunkerImpl.createTestDelegators(AbstractTestSuiteChunkerImpl.java:244)
at org.powermock.modules.junit4.common.internal.impl.JUnit4TestSuiteChunkerImpl.<init>(JUnit4TestSuiteChunkerImpl.java:61)
at org.powermock.modules.junit4.common.internal.impl.AbstractCommonPowerMockRunner.<init>(AbstractCommonPowerMockRunner.java:32)
at org.powermock.modules.junit4.PowerMockRunner.<init>(PowerMockRunner.java:34)
at java.lang.reflect.Constructor.newInstance(Unknown Source)
Caused by: java.lang.RuntimeException: java.io.IOException: invalid constant type: 15
at javassist.CtClassType.getClassFile2(CtClassType.java:203)
at javassist.CtClassType.makeFieldCache(CtClassType.java:837)
at javassist.CtClassType.getMembers(CtClassType.java:828)
at javassist.CtClassType.getMethod0(CtClassType.java:1126)
at javassist.CtClassType.getMethod(CtClassType.java:1115)
at javassist.expr.MethodCall.getMethod(MethodCall.java:114)
at org.powermock.core.transformers.impl.MainMockTransformer$PowerMockExpressionEditor.edit(MainMockTransformer.java:272)
at javassist.expr.ExprEditor.loopBody(ExprEditor.java:191)
at javassist.expr.ExprEditor.doit(ExprEditor.java:90)
at javassist.CtClassType.instrument(CtClassType.java:1374)
at org.powermock.core.transformers.impl.MainMockTransformer.transform(MainMockTransformer.java:74)
at org.powermock.core.classloader.MockClassLoader.loadMockClass(MockClassLoader.java:252)
... 28 more
Caused by: java.io.IOException: invalid constant type: 15
at javassist.bytecode.ConstPool.readOne(ConstPool.java:1090)
```

The unit test for length, vocabulary, difficulty, and volume are different with number of operator and number of operand. Because inside of length, vocabulary, difficulty and volume check function, they call number of operator and number of operand function. Therefore Yuer

need to create fake number of operators and number of operand to test those check functions. Yuer use PowerMockito.whenNew().withNoArguments().thenReturn() and doReturn().when().<> methods to fake the operator and operand numbers to unit test LengthOfOOCheck, VocabularyCheck, DifficultyCheck, and VolumeOfOOCheck.

INTEGRATION TEST

Ran implemented our integration tests using the bottom-up testing strategy discussed in class.

Ran began by integrating number of operator and number operand metrics, used in the length metric. Next, he integrated vocabulary, which uses number of unique operators and number of unique operands. Third, he integrated difficulty, which depends on number of unique operators, number of unique operands, and total number of operands. Finally, he integrated volume, which depends on the program vocabulary. Notice that vocabulary had to be successfully integrated before volume since volume depends on vocabulary, and therefore everything vocabulary depends on.

SYSTEM TEST

Courtney wrote the report and system test. She re-cloned the GitHub repository onto her computer after integration testing was complete.

First, she ran all six metrics on an empty project. Difficulty and Volume had divide by 0 errors, so Courtney updated her difficulty metric to return 0 if there are no unique operators, no unique operands, or no total operands. Yuer updated her volume metric to return 0 if vocabulary is 0. Courtney then typed a small test program with one print statement, first printing a string. She ran all six metrics on her computer, and all metrics were counted correctly. She changed the print statement to print an integer and all metrics executed correctly.

Next, she typed a test program with only operators and another with only operands. She tested an operator only file first and found that only real operators; that is, those not part of a string, are counted. She also noticed that strings are not counted as operands, so this was the easiest way to test only the operators. In order to test only operators, she wrote a bunch of strings and added them together; for example, "hello" + "world" + "how's" + "it" + "going?". Number of operands was 0, number of unique operands was 0, and difficulty was 0. She struggled with testing an operand only file since the only choices for operands are integers and floats, so those need to be assigned to variables or printed using System.out.print, which used "." operators. Eventually, she settled on multiple print statements since print and periods count as 2 operators because then at least number of operators would be consistent and countable.

USER TEST

We each did our own user test using the same test cases.

```
//Empty
package test;
public class test {}

//One print statement with one string
package test;
public class test {
	public test() {
		System.out.print(“hello world”);	}}

//One print statement with one integer
package test;
public class test {
	public test() {
		System.out.print(10);  }}

//One print statement with one float
package test;
public class test {
	public test() {
		System.out.print(1.234);	}}

//Operator Only
package test;
public class test {
	public test() {
		System.out.print(“hello” + “world” + “how’s” + “it” + “going?”);	}}


//Operand Only
package test;
public class test {
	public test() {
		System.out.print(1);
		System.out.print(2);
		System.out.print(3);
		System.out.print(4);
		System.out.print(5);
		System.out.print(6);    }}
```

```
//functionality test 1
package test;
public class test {
	public test() {
		double a = 0;
		a = 1 + 0;   }}

//functionality test 2 with multiple function
package test;
public class test {
	public test() {
		double a = 0;
		a = 1 + 0;  }
	public void A()
	{	int b = 0;
		b ++; }
	public void B()
	{	//empty test } }

//functionality test 3 with '<' sign
package test;
public class test {
	public test() {
		int a = 0;
		int b = 1;
		if(a<b) {  } }
```

GITHUB

We found GitHub to be a very handy tool in terms of version control and a convenient way to collaborate. We used GitKraken as a way to push to and pull from the repository and keep track of what branches were being worked on by what teammate.

Courtney had the most GitHub and GitKraken experience, so she showed Ran and Yuer the basics. We each had our own development branch, named after our GitHub usernames. We pushed our own metrics and other adjustments to these branches. We then cloned master to another branch called dev, and merged all of our development branches with dev to make sure that all of our implementations were compatible. We ran into a few merge conflicts in the form of code being in the same place and manually moving it, but no real incompatibility issues arose.

REFERENCE

Checkstyle.sourceforge.net. (2017). *checkstyle – Writing Checks*. [online] Available at: http://checkstyle.sourceforge.net/writingchecks.html [Accessed 13 Sept. 2017].

Checkstyle.sourceforge.net. (2017). *checkstyle 8.2 API*. [online] Available at: http://checkstyle.sourceforge.net/apidocs/index.html [Accessed 21 Sept. 2017].