CMPT 473 | SOFTWARE QUALITY ASSURANCE

Assignment I Project Report

SEPTEMBER 15, 2015

SCHOOL OF COMPUTING SCIENCE, SIMON FRASER UNIVERSITY
Daphne Ordas, Matthew Chow, Young Yoon Choi (Justin), Wei Wang
Team DJW

Project Report

Contact Information

Daphne Ordas	dordas@sfu.ca
Matthew Chow	mkc25 @sfu.ca
Young Yoon Choi (Justin)	yychoi@sfu.ca
Wei Wang	wwa53@sfu.ca

Project Summary

Target Project:	Hawkular Android Client
Author:	Artur Dryomov
Supporting Organization:	JBoss Community
Code Base Size:	36.3K lines of code (265.7 KB)
Evaluation Platform:	Android Studio under Windows with Java
Project Build Time:	13.600s (Depend on actual environment)
Test Execution Time:	2.216s

Project Details

Our group, DJW, will perform a mutation adequacy study on the Hawkular Android Client from the JBoss Community. The original project consists of 36.3K lines of code with more than 20 existing test cases. The test application shall be obtained from "Google Summer of Code 2015 Project" so further modification from author will not applied. As reference, current GitHub repository can be accessed for further information.

Google Summer of Code 2015 Project Link:

https://www.google-melange.com/gsoc/project/details/google/gsoc2015/artur_dryomov/5635617715126272

GitHub Repository Link:

https://github.com/hawkular/hawkular-android-client

Each group member will apply mutation test on a section of source code contains dozen of files. Each section has its own unite test cases and a integration test for the whole project. Tasks are distributed as follow:

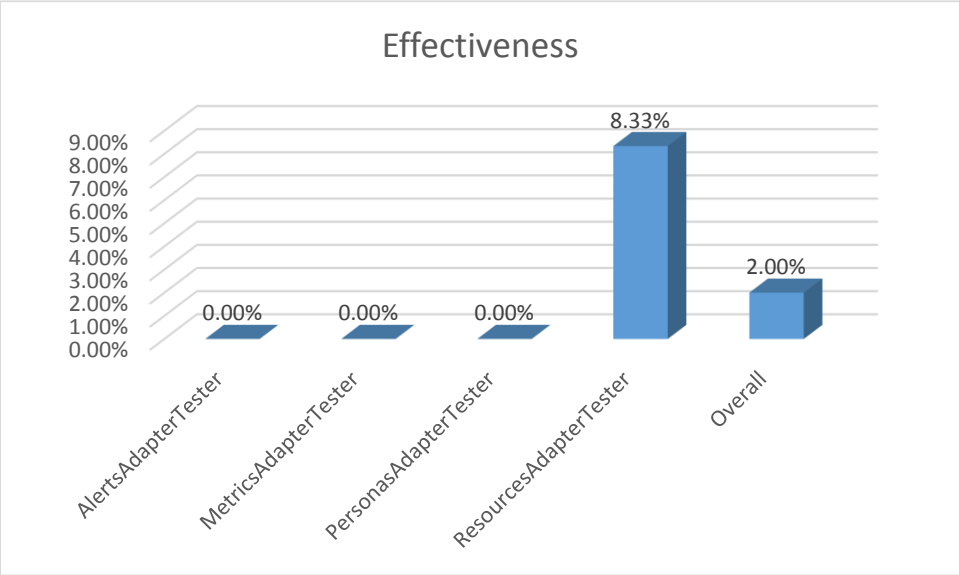
Daphne:	Activity Module
Matthew:	Fragment Module
Justin:	Backend Module
Wei:	Utility Module

Activity Module ----------

Mutation Operators

Arithmetic Operator Changes '++' to '--'
Relational Operator Changes '==' to '!='
Conditional Operator Changes '&&' to '||'
Unary Conditional Operator Change and Removal '!'
Add 1 Operator to a constant
Replace Operator '+' to '-'
Access Modifier Change Operator and Removal of 'private', 'protected', and 'public'

Effectiveness of Test Suite



2 mutations were caught by the tests, and 100 mutants were seeded, meaning that overall, the test suite was 98% not effective. The below graph show that the only test that was able to catch mutations was ResourcesAdapterTester. Inside the ResourcesAdapter.java, 24 mutations were applied, and the 2 mutations that were caught were due to a Unary Conditional Operator Removal '!' in line 91 of ResourceAdapter.java:

```
for (Resource resource : resources) {
-       if (!containsResourceType(resourceTypes,
+       if (containsResourceType(resourceTypes,
```

and the Add 1 Operator to a constant mutation in line 120 of ResourceAdapter .java:

```
private int getResourcePosition(int position) {
-       int resourceTypesCount = 0;
+       int resourceTypesCount = 1;
```

These were determined by reapplying the 24 mutations to pinpoint which lines would fail the test suite.

The live mutants aren't live because of equivalent mutants, and this was determined by looking at the test suite. The test suite doesn't cover some of the files mutated; in particular, the activity folder didn't seem to have any test case coverage, while the adapter folder only had 8 tests that hardly covered anything due to hardly any of the mutants being killed. Some possible equivalent mutants could arise from the Access Modifier Change Operator where most mutations were the changing of 'private' to 'protected'.

```
-private void setUpBindings() {
+protected void setUpBindings() {
      ButterKnife.bind(this);
}
```

The access modifiers are very similar to one another in what they do, but because these were not covered in the test suite, equivalence could not be determined.

Section Conclusion

Test cases that could be added include accessibility tests so that the Access Modifier Change mutants mentioned from before could be killed. Also, mutants like Relational Operator Changes, Conditional Operator Changes, and Unary Conditional Operator Changes were not detected, so additional unit tests to go through the different test paths would help branch coverage and make the coverage of the tests more complete. To catch a mutation such as the one in AuthorizationAcivity.java:

```
private void setUpBackendCommunication(Persona persona) {
-     if (!isPortAvailable()) {
+     if (isPortAvailable()) {
            BackendClient.of(this).configureCommunication(getHost()
            , persona);
      } else {
            BackendClient.of(this).configureCommunication(getHost()
            , getPortNumber(), persona);
      }
}
```

we could use assertions to check the value of isPortAvailable() to make sure that the code would be able to go through the if block and another assertion to check that the return value of BackendClient.of(this).configureCommunication(getHost(), persona) is correct. Within the same test, we would check the else part of the if block to make sure that what is happening is correct.

In addition, Replace Operators and Arithmetic Operators could use some test coverage by having tests that check the value of the variables changed. For example, if we want to check alertsValueCount++ we would have an assertion at the end of our test case to make sure that alertsValueCount is the correct value at the end.

Mutants were manually applied to the test suite, as the framework chosen for the automation of mutants (PIT mutation tool) was not compatible with Android, even though the tool was meant for Java applications. Automation of mutants would be achieved by taking the aforementioned mutation operators and implementing them

in the automation in single places of code. If a mutation caused the program to not compile, then that mutation would not be applied.

Interesting observations of the test suite were that the test suite only consisted of 43 tests. This suggests that the author of the code did not focus on Test Driven Development, and focused more on building the code's functionality.

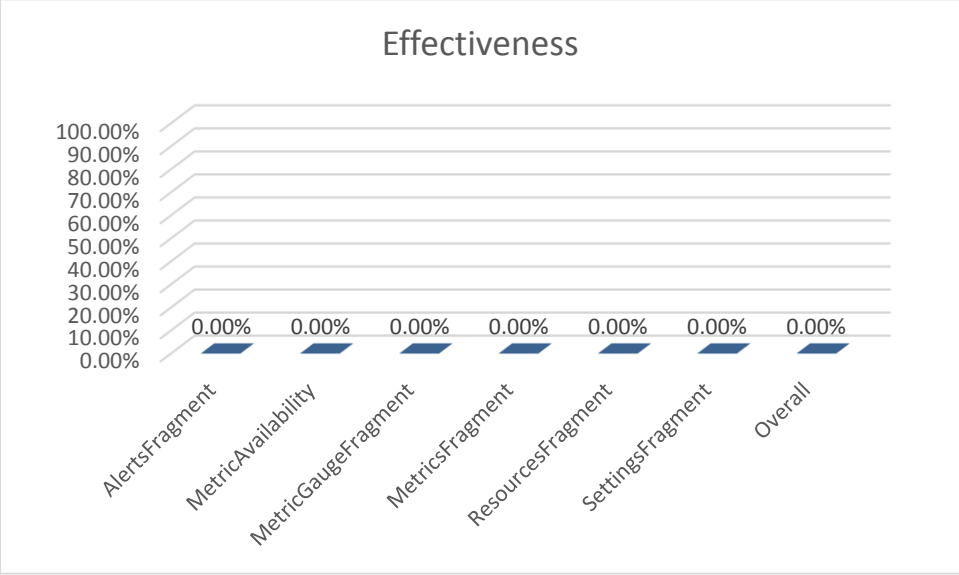
To summarize, by adding the suggested test cases that check for the activity files and live mutations, the test suite could become more robust. Because the tests weren't the focus of the author's implementation, mutations were able to easily become undetected. If there were more unit tests that focused on checking the value of variables and the scope of the methods, most of the manual mutations made would be caught.

Fragment Module ----------

Mutation Operators

Boolean: true -> false, false -> true
Integer/Float: 1 -> 0, –1, 10, 1.9, 0.9 etc.
String: addition of character
Relational modifier: != -> ==, == -> !=
Access modifier: Private -> Protected
Deletion of a single line of code

Effectiveness of Test Suite



Altogether, zero out of one hundred mutants seeded were killed in the test suite. This means that the test suite was 0% effective across all routines. Test cases had not yet been written specifically for the fragment folder, and the test suite only contained test cases for the util, backend, adapter, and event folders. The Event folder had insufficient code to modify (<15 lines). Equivalent mutants could not be determined as there were an insufficient of test cases that covered possible errors. Additionally, it is unfortunate that the test suite is comprised of only unit tests. As such, integration

and systems test cases were lacking. This affects the ability of the test suite to catch mutants as components are less likely to fail and affect the overall system.

Section Conclusion

There are several methods to help the project move forward effectively. They include adding proper test cases for a more comprehensive test suite and incorporating automated UI test tools. Test cases should account for:

- Extreme values. Tests could incorporate random integer or float values between specified ranges at boundary conditions.
- Since many of the operators were concerned with the visualization of data, tests should be written to ensure images, buttons, and other UI assets appear properly on the device.

Because this component of the application is so visual, an effective and simple test oracle would be a human one. Mutations might not crash the application, but may affect the graphics and images presented to the user. Additionally, one can take advantage of a number of automated UI testing tools that exist for the Android platform. For example, the Maveryx tool detects GUI objects at run time. Tests can be written to ensure UI elements appear and appear where they ought to when mutations might change the positioning and placement of the asset.

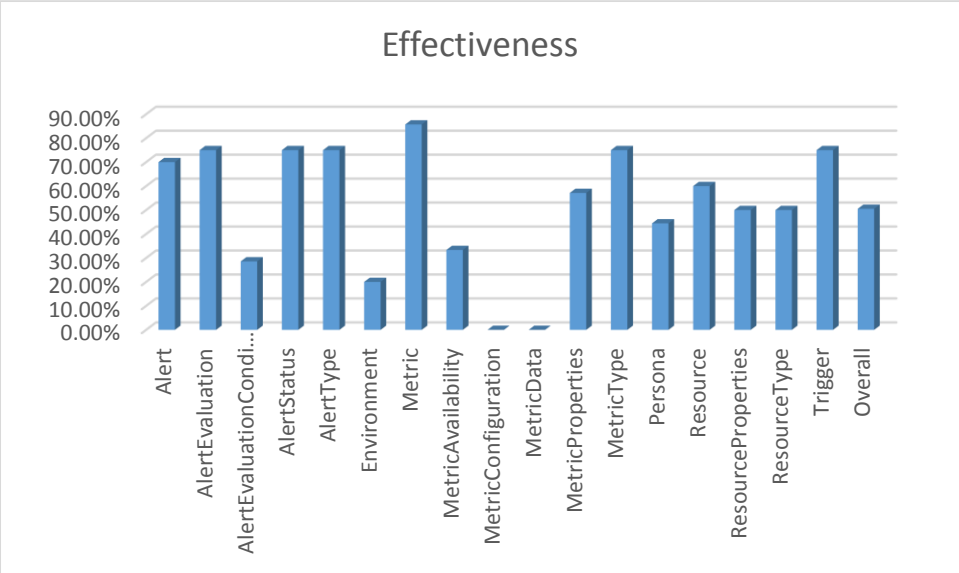
All in all, the test suite for the Hawkular project is insufficient in coverage and in scope, especially in the area of visual graphic representation. This is understandable as the Hawkular Project is still in its Alpha development phase (1.0.0.Alpha5) and many intended features are still in development. There are a number of solutions to aid in the development of the Hawkular project, including the addition of test cases concerning large datasets, large/small/positive/negative values, windows and other UI assets, and the usage of automated UI testing tools.

Backend Module ----------

Mutation Operators

Adding 1 to value: Alert[size+1]
This keyword insertion: this.type = this.type
This keyword deletion: type = type
Return null instead of object or string: return null
Delete line
Negate return value: return -timestamp
Change class to be non final
Access modifier change: private -> public
Change switch condition value: switch(value+1)
Change default return value of switch statement
Change variable to constant

Effectiveness of Test Suite



Total 101 mutants and 51 KILLED mutants -> 51 / 101 = 50.5% effectiveness

Almost half of mutants are live due to lack of test coverage.

List of Survived Mutants:

Changing value of size for array: Check size of array created and parameter

```
-       return new Alert[size];
+       return new Alert[size+1];
```

Negating return value: Check return value

```
-       return timestamp;
+       return -timestamp;
```

Deleting line: Check for values changed

```
-       parcel.readList(evaluations, List.class.getClassLoader());
+
```

Changing variable to constant: Check return value

```
-       return dataTimestamp;
+       return 0;
```

Deleting / Adding this keyword: Check if type in argument is same as type in object

```
-       this.type = type;
+       this.type = this.type;
```

Changing class to be non final: Check if inheritance is allowed

```
-       public final class AlertEvaluationCondition implements Parcelable {
+       public class AlertEvaluationCondition implements Parcelable {
```

Returning null instead of object: Check return object with desired object

```
-       return new AlertStatus[size];
+       return null;
```

Access modifier change: Checking if method can be accessed by outside class

```
-       private Environment(Parcel parcel) {
+       public Environment(Parcel parcel) {
```

Section Conclusion

The test suite was not really precise, because mostly mutants that were not caught were not functionally equivalent to the original source code. The test suite was only concerned about a few cases, not for general cases such as only checking for a few variables and not all variables. The test suite should contain more test cases to check every method and variable, such as checking the return object with the expected object for accessors. Also, an area for concern would be access modifiers, since the test suite does not kill the mutants created by changing access modifiers.

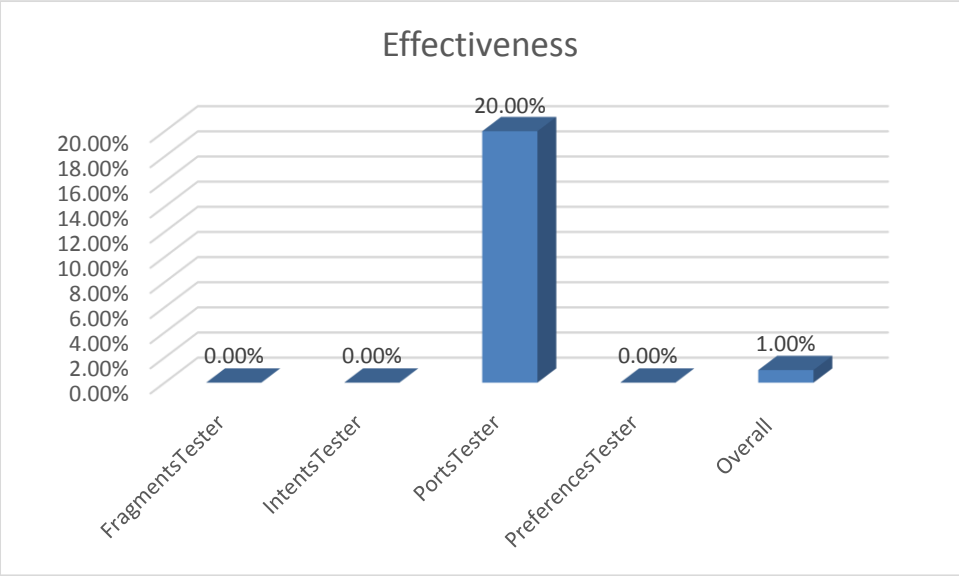
Due to a lack of mutant generating tools for android applications, I have created mutants manually. To automate mutation generation, we can use a script to apply a few mutation operator lists, and then check for syntax errors for each mutation generated. For the automation of applying the test suite, first store the mutants into one project which contains the test suite by using the mutation generating tool and then apply the test suite to each mutant in the project folder.

Utility Module ----------

Mutation Operators

==	!=
>=	>
<=	<
int	int + 1
private	public
static	static
boolean	!boolean

Effectiveness of Test Suite



First of all, equivalent mutants are factors that reduce the accuracy of the mutation test. The number of equivalent mutants shall be reduced as few as possible. In addition, all mutants in the application are inserted manually. Based on these, all equivalent mutants are screened out during insertion. No equivalent mutants are detected in survived mutants. However, there are a few examples that illustrate potential equivalent mutants:

```
if (x > 0) ... if (!(x <= 0))
static final MAX + 1;  ...  function.(MAX - 1);
public fun(){}  ...  private fun(){} //(Empty function)
```

The above example shows it is possible to have equivalent mutants with 1 or more mutation operators depending on the scope where the mutation is applied. To detect other survived mutants, the mutants shall be categorized into different types. For each type, there are approaches to handle these types accordingly in the following discussion:

- **Value Shift Mutation**

  This type of mutation is done by shifting a variable one bit. Some examples of this include changing the constant 100 to 101, i++ to i+=2, and TRUE to FALSE. The ultimate way to detect this type of mutant is a white-box approach: design a walkthrough to cover the program as much as possible and input designated parameters to compare the result with the theoretically correct answer. This approach can screen most value shift mutations if designed well. However, there are still exceptions, such as:

  > A hash function mapping a giant superset into a much smaller subset.

  In this case, since the superset is much larger than the subset, boundary values may not be able to be tested due to resource constraints. Thus, even well designed test cases can still miss some mutants. The example here is

trying to convince that detecting mutants is a task relatively based on the resources. The more sophisticated the test case is, the more likely mutants are killed. However, more resources are needed. Also, the mutants escaping from the sophisticated test cases tend to be less prone to cause problems. Thus, the testing method shall be balanced.

- **Scope Change Mutation**

  These are the most common mutants among this particular experiment. Every single "private" keyword has been mutated to "public" in this application. Generally speaking, reducing entity scope is a good practice in coding. In other words, the application is less likely to have problems if all entity scopes are limited exactly enough for doing their task. However, this doesn't mean if the scopes are not limited, there will be problems later. This is the reason why most mutants are passed by the test cases since they didn't cause real problems. But, if multiple override and variable shadowing are introduced in the code, unlimited scopes can cause serious problems. Since people tend to use different names for their entities, these types of mutants are less destructive than the previous one.

  A potential method to detect these mutants are from the program platform. Normally IDE or execution environment like JVM will maintain a list of the global variables, functions and some other local ones on a heap. It is possible to write a test program to monitor the environment and compare the stack/heap changes from mutants. When other types of mutations are minimized, this factor shall be easily compared.

- **Oracle Specific Mutation**

  This type of mutation has the least impact to the program generally. The main characteristic is that the test oracle must be a human to decide if the mutation changed the program or not. Some examples include if a mutant changed the warning window 1 pixel higher than normal, or if a color #AAAAA is changed to #ABABAB. This type of mutation can be dependent on other types. For example, a value shift mutation changed RGB(255, 255, 255) to RGB(254, 255, 255), which shows it is also an oracle specific mutation. The key here is the actual change done by the mutation is most related to visual appearance. In this case, only a human who is actually using the software is able to decide if the mutant changed the appearance or not.

  To detect this type of mutant, a human oracle is suggested. It is possible to use another program to detect the output pixel by pixel. However, humans may feel little difference, while this program may claim the output is completely different. More advanced computer vision algorithms may be introduced to achieve a better result.

Section Conclusion

Since the application here is relatively small scaled and development is still in progress, the major task is to make it working, rather than offer a stable tested and

structured program. Thus, the limited test cases are focused on critical errors rather than detecting all types of mutations. Most of the mutants (99%) are not killed in this experiment.

Although the test cases are quite ineffective, there are a few observations according to mutation generation. Since the purpose of mutation analysis is to estimate the test case efficiency in the future, the more similar mutations are to the real bugs, the better estimation will be achieved. In other words, it has no meaning to design a test case to screen a slight change out, such as an extra blank added. The test case shall be sensitive to critical error even if some mutants are survived by it. Thus, the mutation operation shall mimic real bugs, such as boundary value change in a for loop rather than shifting the background color. The test case shall be considered good if it is able to capture the boundary error, while other mutants are missed significantly. In contrast, a test case shall be considered bad when it detects only slight, non-vital changes to the source code. It is a good thing to kill mutants. However, more time and money will be spent on designing a test case to screen out the non-vital changes. The resources spent here may not able to compensate the performance gain.

To automate the process, a script can be introduced to change a single operation. There must be more elegant approaches for automation. However, the most trivial way is to format the source code to a standard looking, then let the script apply every possible mutation operator. Each time a mutation is applied, run the compiler to check if there is any syntax error. Mutation shall be reversed if any syntax error is detected. Some optimization may be introduced, such as applying a block of mutation then using binary search to detect the syntax error (if it occurs). Also, increase block size while applying less critical mutations since they are less likely to cause syntax errors. This is only some trivial guessing.

Final Conclusion

Because the project is still in Alpha, it is evident that the author of the code did not put as much focus into the tests, and instead of adapting facets of Test Driven Development focused on making the code functional. Therefore, the overall consensus is that the test cases were quite ineffective.

The addition of test cases described above should generally include more test cases for checking the boundary value of variables, and accessibility of the methods. There should also be more test cases in general, because the coverage of the tests aren't covering much of the source code. To explain, most mutants that weren't caught were mutants that were not equivalent to the original source code's functionality.

Across all files mutated, manual mutation was used, but the creation of mutants could have been automated by using a script to apply mutation operator lists. Mutants that create syntax errors will not be accepted. Mutation testing becomes more effective the more mutants emulate real bugs, such as boundary values in for loops, because applying a non-vital mutation wastes resources and may not compensate for performance gain.

For some other files of the application, creating test cases is difficult because there are some things that are difficult for automation to detect. For instance, some of the application deals with the display of graphics and images, and if the test oracle is the automation then it is

hard to determine the differences between a right or wrong image. In this case, the recommended test oracle would be a human to manually check over these graphics and images to determine if they are correct.