Software Engineering: Final Report

Team: Lamp

Project: Glucosio

By: Jason Adler, Shefali Emmanuel, & Dwayne Ferguson

Date: 11/21/19

Table of Contents

Chapter 1: Project Selection and Development

1.1: Project Selection

At the start of this semester, we were advised to pick the Humanitarian Free Open Source Software Project that stemmed from a language we, as a team, we're most comfortable with (java). The team initially considered three projects: Nightscout, Spatiotemporal Epidemiological Modeler (STEM), and Glucosio. Nightscout is an application that allows a user to monitor their blood glucose levels, but the project had not been worked on in quite some time. STEM is a tool designed for helping scientists and public health officials create and use spatial and temporal models of emerging infectious diseases. While this project is active, it is built-in Eclipse, is extremely large, very complicated, and initial attempts to build it in Eclipse failed.

Out of all the incredible Java HFOSS projection options available, we decided on Glucosio for three primary reasons. Firstly, Glucosio is the only open-source diabetes app for both type 1 & type 2 diabetes. Secondly, it is designed to be a user-centered free and open source app with the goals of diabetes management and research. The users seem to find it quick & easy to track all aspects of their diabetes such as blood glucose, HB1AC, Cholesterol, Blood Pressure, Ketones, Body Weight and many more. Thirdly, there has not been any project development in the past 2 years. This is a vital characteristic as there

would be no sudden alterations to the project we were currently working on.

1.2: Requirement Specification

To bring this project to life, the tools utilized were Android Studio 3.5, GitLab, GitHub, VirtualBox 6.0.10, and Gradle 4.4.1. The primary languages we used were Java 1.8.0_222 and Bash.

1.3: Complications

Upon building the project in Android Studio we came across a few road bumps. Our initial attempts to build test the project so far have had mixed results. The project has been idle for roughly 2 years and therefore there may be problems with the more up to date versions of Gradle. Additionally, Android Studio is new to the team and as when learning any new IDE, learning a new interface takes time. We have had issues with using Virtual Box as well. The tools needed to build Glucosio have been larger than expected and caused us to have issues with the allotted hard disk space on the virtual device. Expanding the size of the hard disk space can be challenging. The team sought clarification from Dr. Bowring about the testing process and he suggested becoming more familiar with JUnit to create tests for our project. He also shed light on using the Android Studio built-in-terminal for running the program and scripts in the future.
We followed a strict timeline to achieve our outcomes. After finalizing our project topic, we evaluated and built the project as we found it. Next, we

developed a test plan along with our top twelve methods to test. During our next sprint, we designed and built the Driver, as well as, created formal documentation and architectural models. Next, we created twenty-five test cases. During our fifth sprint, we design and inject five faults into the code. Lastly, we finalize the driver, presentation, project, paper.

1.4: Build Glucosio

1.4.1 Required Software

To build the project the team used:

- Ubuntu 18.4
- VirtualBox 6++
- Android Studio 3.5
- Android Sdk 27
- Java Development Kit 11.0.5
- Gradle 4.4.1

1.4.2 Build Instructions

The first step is to install VirtualBox and Ubuntu 18.4 or higher. The team used default settings in both of these installs and should be self-explanatory. Next, install Android Studio 3.5 in the default location. This can be done in Linux via this link or the install menu in Linux. Once again, the team used the default install location and settings for Android Studio. After Android Studio is installed open

the configuration menu at the initial splash screen then AndroidSDK>SDK Platforms and check Android 8.1 (Oreo) and click apply. This will install Android SDK 27 and allow the user to accept the license agreement. While this process should be possible through the command line, the team faced consistent problems attempting to do so due to the inability to accept the license agreement. The Android SDK must be installed in the default location as there is a properties file that is generated by the script. This is not necessary if building the project via the Android Studio environment. Finally, installing Java Development Kit 11.0.5 and Gradle by calling Sudo apt install default-JDK and Sudo apt install Gradle respectively.

Chapter 2: Test Plan Production

2.1: Driver Goals

The goal is to eventually test 25 cases for the Glucosio Software. The team will identify five methods to test five times each. We will be writing a script to open our driver. The driver will send arguments through the main method, which will create an instance of the class to test. The arguments will be run through the method being tested and then output to a file on the computer. The driver will then locate the file, open it, and compare the outcome with the oracle. Finally, the console will display the expected and actual outcomes.

2.2: Requirements Traceability

We do not know the requirements the original developer was trying to meet but as Glucosio was created as a diabetes management tool we have developed a core requirement that will likely envelope all of our test cases.

- Function: Compute Glucose & A1C levels
- Description: Compute glucose and A1C levels at regular intervals throughout the day and when the sensor transmits data to the application or the user inputs data.
- Inputs: User weight, blood sugar reading

- Source: Reading from sensor or user input.
- Outputs: The calculated blood sugar and recommended levels according to the ADA, AACE, and UKNICE.
- Destination: User Screen (MainPresenter)
- Action: Glucose readings are received by a sensor or input from a user. The user's self-entered weight is used with the glucose reading to calculate A1C levels according to the NGSP and IFCC. These values are then compared to the recommended levels by the ADA, AACE, and UKNICE and displayed on the user's screen.
- Requires: User or sensor input of weight and glucose levels.
- Precondition: None
- Postcondition: Read value is stored in a user's profile.
- Side effects: None.

2.3: Tested Methods

glucosio-android -> app -> src -> main -> java -> org -> glucosio -> android -> tools -> ...

1. GlucosioConverter.java -> kgToLb(double kg)
2. GlucosioConverter.java -> lbToKg(double lb)
3. GlucosioConverter.java -> a1cToGlucose(double a1c)
4. GlucosioConverter.java -> glucoseToA1c(double mgDl)
5. GlucosioConverter.java -> glucoseToMmolL(double mgDl)

6. GlucosioConverter.java -> glucoseToMgDl(double mmolL)

2.3: Test Case Template

As outlined in "The testing process" above, the team will write a script that will execute the methods with test values. We will identify five different methods to test five different times each. The results of these executions will be outputted to a file on the user's computer. The file will contain the following test cases:

1. Test Number
2. Method being tested
3. Expected outcome(s)
4. Test input(s) including command-line argument(s)
5. Requirements

2.4: Constraints

We developed a list of primary constraints that would have helped us develop our project rapidly. Our main constraint was the lack of team members, but we were able to resolve this problem by voluntarily working on aspects of the project that aligned with our skill sets. We all had never used Android Studio before this project which provided a steep learning curve. Lastly, there was a lack of familiarity with the correct version of the project to utilize from HFOSS Developers.

Chapter 3: Drivers and Test Cases

3.1: Driver Development

3.1.1 Original Driver Idea

The team originally pictured the driver as a java file that would read through the test case text files and parse out all of the details and then put the information found in an array to output to another text file and then finally put into an HTML document. In fact, there was a working version of nearly all of those things but after we sought some clarity, it was apparent that the bulk of the work should be accomplished by the script itself. The team did not have a lot of experience working with bash and this was a hurdle we had to overcome.

3.1.2 Bash Scripting

There were growing pains when working with bash but after learning the basics it was apparent why it was the prefered method to use in our relatively simple testing framework. The speed and simplicity at which it handles text and can then be used to execute commands would have taken far more work, not to mention time, to do in Java.

The script had significant changes throughout the process, one example being initially it was iterating through a set number of test cases but was redesigned to take an unlimited number instead. As the team grew more familiar with bash,

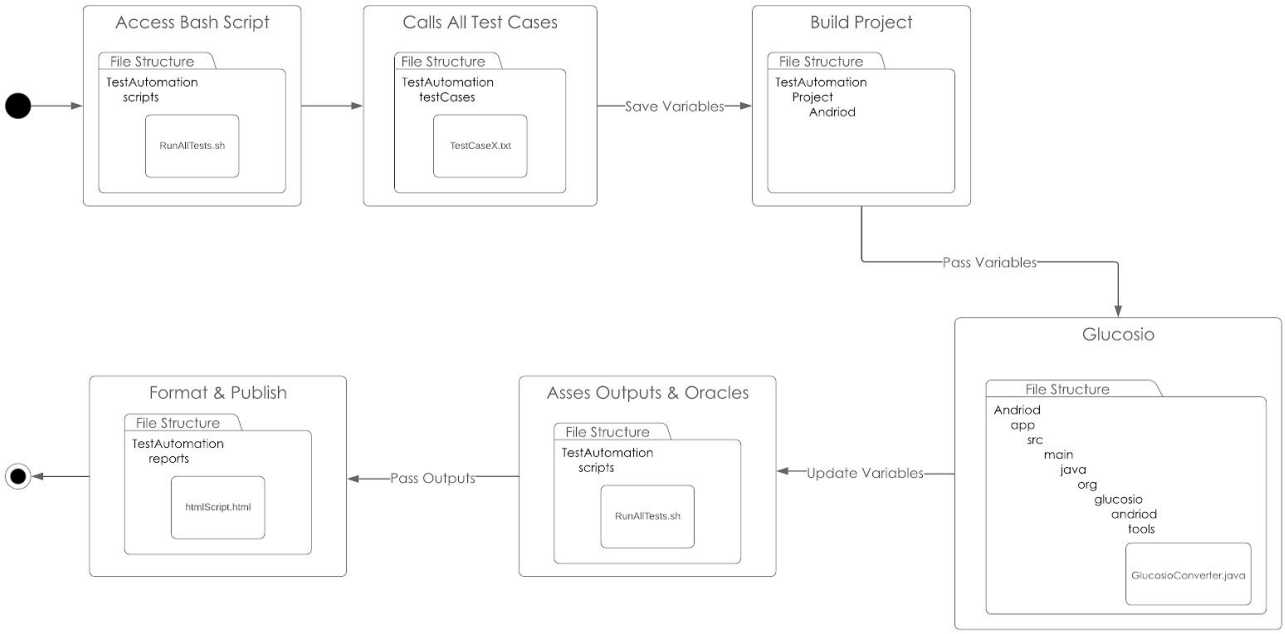
the improvements to the script became less daunting and in the end we were able to create a driver that performed all of the functions necessary to complete the project.

3.2: Initial Test Cases

The third deliverable included reworking the test plans if instructed to do so from the previous deliverable, as well as building the automated testing framework. The automated testing framework includes an architectural description of the framework, full how-to documentation, and a set of at least five of the eventual 25 test cases. For the initial test cases, the team developed six, three from the lbToKg() method and three from the kgToLb() method.

3.3: Architecture Diagram

The architecture description was depicted using an activity diagram. The activity starts with running the bash script. The script will begin a loop structure to call all test cases which are saved as .txt files. Next, the script builds the project and passes the previously saved variables into Glucosio. The result then updated the variable and compares them to their adjacent test cases oracle. Lastly, the script formats and publishes the results in a web page.



3.4: How-To Run All Test Cases

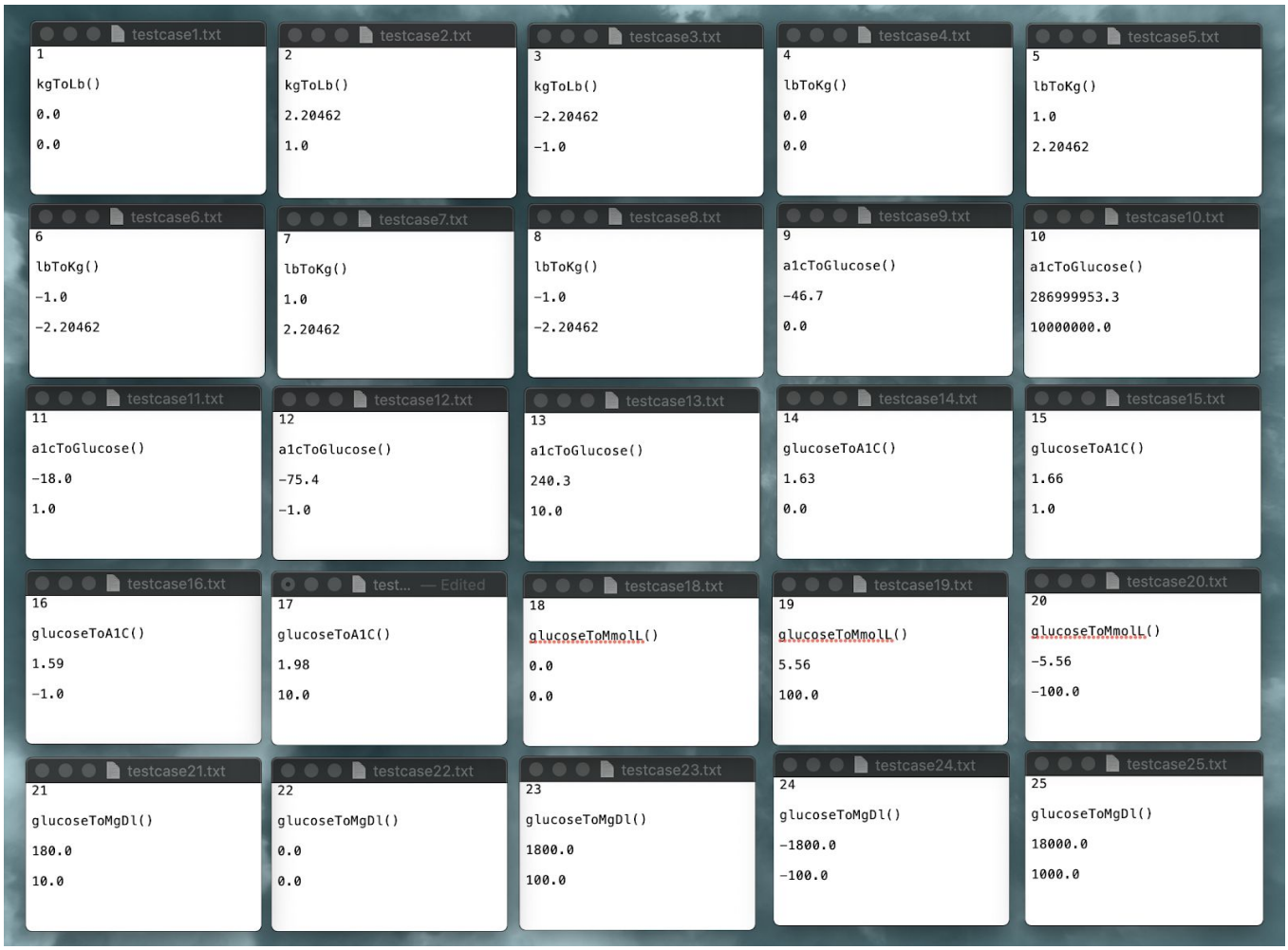
The how-to run all test cases:

1. Clone project to a computer

2. Open terminal
3. Navigate to Lamp-master/TestAutomation/scripts
4. Use command './runAllTests.sh
    a. To build the project first use ./runAllTests.sh -b
5. Testing should complete and output to HTML file in browser

Chapter 4: Complete Test Case Development

The team needed to develop more test cases to increase the total number to 25. We selected methods that seemed easy to test, with basic math operations and conversions. The twenty-five test cases came from the following methods: kgToLb(), lbToKg(), a1cToGlucose(), glucoseToA1c(), glucoseToMmolL(), and glucoseToMgDl(). Test inputs included negative and positive values, as well as zero and large numbers. All tests passed with their expected inputs and outputs. There does not seem to be any errors in the code that the team tested in Glucosio (NOTE: test cases do not include requirements which were added later).



Chapter 5: Fault Injection Development

Fault injection is a method by which the developer purposefully creates a fault in the program to see how it is handled. The idea for the project was to inject 5 faults into the code of the project and run the testing framework we had built and compared the results. The team chose to utilize mutation testing to see if there were any gaps in our current testing coverage. As stated above, there were 5 changes, some of which overlapped in their expected impact on the testing results. All changes took place in the GlucosioConverter.java.

Original method #1.	Mutation 1. Changed conversion value

public static double glucoseToA1C(double mgDl) {
return round((mgDl + 46.7) / 28.7, 2);
}
	public static double glucoseToA1C(double mgDl) {
	return round((mgDl + 44.7) / 28.7, 2);
	}

Original method #2	Mutation 2. Changed - to +

public static double a1cToGlucose(double a1c) {
return round((a1c * 28.7) - 46.7, 2);
}
	public static double a1cToGlucose(double a1c) {
	return round((a1c * 28.7) + 46.7, 2);
	}

Original method #3	Mutation 3. Changed rounding decimals.

public static double glucoseToMmolL(double mgDl) {
return round(mgDl / MG_DL_TO_MMOL_CONSTANT, 2);
}
	public static double glucoseToMmolL(double mgDl) {
	return round(mgDl / MG_DL_TO_MMOL_CONSTANT, 0);
	}

Original method #4	Mutation 4. No longer rounds the output.

public static double glucoseToA1C(double mgDl) {
return round((mgDl + 46.7) / 28.7, 2);
}
	public static double glucoseToA1C(double mgDl) {
	return ((mgDl + 44.7) / 28.7);
	}

Original method #5	Mutation 5. Changed constant value
private static final double KG_TO_LB_CONSTANT = 2.20462;	private static final double KG_TO_LB_CONSTANT = 2.2;

Prior to injecting the faults, all 25 of our tests passed. After injecting the five faults, 17 out of 25 tests failed.

Test Number	Method Name	Requirement	Input Value	Output Value	Oracle	Result
1	kgToLb()	Double input converts from kilograms to pounds. Output must also be double. Used in other conversion methods.	0.0	0.0	0.0	PASSED
2	kgToLb()	Double input converts from kilograms to pounds. Output must also be double. Used in other conversion methods.	1.0	2.2	2.20462	FAILED
3	kgToLb()	Double input converts from kilograms to pounds. Output must also be double. Used in other conversion methods.	-1.0	-2.2	-2.20462	FAILED
4	lbToKg()	Double input converts from pounds to kilograms. Output must also be double. Used in other conversion methods.	0.0	0.0	0.0	PASSED
5	lbToKg()	Double input converts from pounds to kilograms. Output must also be double. Used in other conversion methods.	2.20462	1.0020999999999998	1.0	FAILED
6	lbToKg()	Double input converts from pounds to kilograms. Output must also be double. Used in other conversion methods.	-2.20462	-1.0020999999999998	-1.0	FAILED
7	lbToKg()	Double input converts from pounds to kilograms. Output must also be double. Used in other conversion methods.	2.20462	1.0020999999999998	1.0	FAILED
8	lbToKg()	Double input converts from pounds to kilograms. Output must also be double. Used in other conversion methods.	-2.20462	-1.0020999999999998	-1.0	FAILED
9	a1cToGlucose()	Converts from A1C to Glucose. A1C Input, Glucose Output (Double): A1C X 28.7 - 46.7.	0.0	46.7	-46.7	FAILED
10	a1cToGlucose()	Converts from A1C to Glucose. A1C Input, Glucose Output (Double): A1C X 28.7 - 46.7.	10000000.0	287000046.7	286999953.3	FAILED
11	a1cToGlucose()	Converts from A1C to Glucose. A1C Input, Glucose Output (Double): A1C X 28.7 - 46.7.	1.0	75.4	-18.0	FAILED
12	a1cToGlucose()	Converts from A1C to Glucose. A1C Input, Glucose Output (Double): A1C X 28.7 - 46.7.	-1.0	18.0	-75.4	FAILED
13	a1cToGlucose()	Converts from A1C to Glucose. A1C Input, Glucose Output (Double): A1C X 28.7 - 46.7.	10.0	333.7	240.3	FAILED
14	glucoseToA1C()	Converts from Glucose to A1C. Glucose Input, A1C Output (Double): AVG glucose + 46.7 / 28.7.	0.0	1.5574912891986064	1.63	FAILED
15	glucoseToA1C()	Converts from Glucose to A1C. Glucose Input, A1C Output (Double): AVG glucose + 46.7 / 28.7.	1.0	1.5923344947735194	1.66	FAILED
16	glucoseToA1C()	Converts from Glucose to A1C. Glucose Input, A1C Output (Double): AVG glucose + 46.7 / 28.7.	-1.0	1.5226480836236935	1.59	FAILED
17	glucoseToA1C()	Converts from Glucose to A1C. Glucose Input, A1C Output (Double): AVG glucose + 46.7 / 28.7.	10.0	1.9059233449477353	1.98	FAILED
18	glucoseToMmolL()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Millimoles per liter.	0.0	0.0	0.0	PASSED
19	glucoseToMmolL()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Millimoles per liter.	100.0	6.0	5.56	FAILED
20	glucoseToMmolL()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Millimoles per liter.	-100.0	-6.0	-5.56	FAILED
21	glucoseToMgDl()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Milligrams per 100 millilitres.	10.0	180.0	180.0	PASSED
22	glucoseToMgDl()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Milligrams per 100 millilitres.	0.0	0.0	0.0	PASSED
23	glucoseToMgDl()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Milligrams per 100 millilitres.	100.0	1800.0	1800.0	PASSED
24	glucoseToMgDl()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Milligrams per 100 millilitres.	-100.0	-1800.0	-1800.0	PASSED
25	glucoseToMgDl()	Calculate concentration of glucose. Input(Double): glucose, Output(Double): Milligrams per 100 millilitres.	1000.0	18000.0	18000.0	PASSED

Due to the nature of the methods tested, even a slight change was expected to fail. If the methods initially are chosen were more complicated the results of this testing may be more insightful. Although, because nearly every test failed, it seems that our current testing framework can identify issues that may arise as the code is changed.

Chapter 6: Final Thoughts

6.1: Overall Experience

The experience was largely positive throughout the project, and after it was completed. The team felt satisfied with the final submission. There were definitely moments early on which were tough, for example, one of the biggest hurdles the team faced was trying to initially build the project. Once that was completed, things became easier and with each bit of progress that was made, the team felt more and more comfortable talking about, working with, and amending the project. Each of the group members communicated well and did their part when necessary. Finding meaningful roles so that progress can be made without getting in each other's way can be a tedious task, but the team seemed to put it together organically. Consistent communication through the Slack channel and using our resources, whether it was Google or Dr. Bowring, proved to be key to the success of the project.

6.2: What We Learned

The project was very productive. As students, this is one of the first times that many of us are exposed to automated testing frameworks, GitHub, command line, Linux OS, and our group specifically, Android Studio and Gradle. Although

there is still much to learn with each of these technologies, it was great to be introduced to them in a classroom setting, with plenty of time to accomplish the project, and nothing to be afraid of.

6.3: Assignment Evaluation and Future Suggestions

The project seemed to be timed well, with each deliverable being a necessary step in the process. It didn't seem overwhelming at any one point, there was a balance in the workload throughout the semester. Good time management was necessary, but with cooperative and involved teamwork, it is doable and appropriate for the semester-long project.

In the future, the team felt as if the deliverables could have been defined as better and more clear. For example, in deliverable five, the team originally thought that they were supposed to write test cases that would cause the project to fail rather than simply change the project code to cause the failure. More discussion about the deliverables and their specifics could have been more helpful. Also, in deliverable three, during the in-class presentations, almost every team had incorrect test case templates. There was extra information, such as descriptors, rather than only the specific information needed. Had this been better explained, the teams would have been better prepared for that deliverable. The team felt that more discussion about drivers could have been helpful, this was the first exposure to writing a driver for most of the students, and a little more direction could have been valuable. Finally, the team felt that the

final report could have been directed better. There has been confusion as to whether it should be an iterative document with each of the previously written chapters simply pasted together, or if it should be edited to reflect a final paper written at the end of the completed project.