

**Disclaimer:** This write-up and its videos may refer to versions of the tools that differ from the ones you will be using. Similarly, pay attention to the versions used in tutorials found in youtube. Despite these differences, this material is useful to study and to try to actually carry out yourself.

Please first focus on JUnit, Cucumber and Selenium as we may not use the other material.

### 3 Table of Contents

	Pages.
6.1 Motivation . . . . .	07
6.2 Methodology . . . . .	07
6.3 Setting up the Environment and TDD with Junit . . . . .	08
6.3.1 Test Driven Development (TDD) . . . . .	08
6.3.2 Git and GitHub . . . . .	09
6.3.3 Eclipse and Junit. . . . .	09
6.3.4 Creating Exercise One. . . . .	10
6.4.1 Running Tests and Creating a Test Suite in JUnit. . . . .	10
6.4.2 Running Tests. . . . .	11
6.4.3 Creating Test Suite. . . . .	12
6.4.4 Creating Exercise Two. . . . .	12
6.5.1 Creating logs with Log4j. . . . .	13
6.5.2 Log4j. . . . .	13
6.5.3 Creating Exercise Three. . . . .	14
6.6.1 TestNG Integration. . . . .	14
6.6.2 TestNG. . . . .	14
6.6.3 Creating exercise Four. . . . .	15
6.7.1 Web App Automation with Selenium. . . . .	15
6.7.2 Selenium. . . . .	16
6.7.3 Potential Web App Problems. . . . .	17
6.7.4 Creating Exercise Five . . . . .	17
6.8.1 BDD with Cucumber and Client/Server system. . . . .	18
6.8.2 Behaviour Driven Development (BDD) . . . . .	18
6.8.3 Cucumber. . . . .	19
6.8.4 Client and Server Testing. . . . .	20
6.8.5 Creating Exercise Six. . . . .	20
6.9.1 Threads, Race Condition, and Deadlocks . . . . .	22
6.9.2 Threads . . . . .	22
6.9.3 Race Conditions . . . . .	23

	Pages.
6.9.4 Deadlocks . . . . .	23
6.9.5 Creating Video Seven . . . . .	24
6.10.1 TOTEM Methodology's Issues with Concurrent Systems . . . . .	25
6.10.2 Testing Object-oriented Systems with the Unified Modeling Language (TOTEM) . . . . .	25
6.10.3 TOTEM Process. . . . .	26
6.10.4 TOTEM Difficulties with Concurrent Systems. . . . .	26
6.10.5 Difficulty in Recreating Faults in Concurrent Systems. . . . .	27
6.10.6 Creating Video Eight . . . . .	28
7 Results. . . . .	28
8 References . . . . .	30

## 4 List of Figures

	Pages.
Figure 1 – Green JUnit test . . . . .	11
Figure 2 - Red JUnit Test . . . . .	11
Figure 3 - JUnit Expected vs Actual. . . . .	11
Figure 4 - List of test cases ran. . . . .	12
Figure 5 - Inspect Element . . . . .	17
Figure 6 - Cucumber Feature. . . . .	19
Figure 7 - Step Definitions. . . . .	20
Figure 8 - Thread Randomness. . . . .	22
Figure 9 – Deadlock Visualization . . . . .	25

## 5 List of Tables

	Pages.
Table 1 - Sequences of Lines Run by Threads. ....	27

## **6.1 Motivation**

In Comp 4004, students are taught software quality assurance and testing. In this course students complete a variety of assignments with different software tools to ensure that the applications they produce through their work is thoroughly tested and working as intended. As students are expected to learn how to use the software tools on their own time, the content taught in lectures is not aimed at the usage of the tools but rather the ideas and concepts of software testing and quality assurance. As most of the material taught during lectures are higher concepts the material is quite stale as students do not have the time to practice the techniques and concepts taught until they are working on their projects during their own time.

The objective of this honours project is to create a set of exercises to enhance the learning experience during lectures. With short exercises and examples of the tools being used, ideally this would increase attentiveness in lectures and may act as a supplementary resource for the students outside of lectures. The exercises cover the following tools used in the course: JUnit, Log4j, TestNG, Selenium, Cucumber. It also covers some concepts taught in lectures such as Test-Driven Development (TDD), Client / Server Testing, Java Threads, Concurrent Threads Testing and Testing Object-oriented Systems with the Unified Modeling Language (TOTEM).

## **6.2 Methodology**

Initially, the plan was to create short written exercises / walkthroughs that students would download and then follow the steps to complete the exercise. The problem with written exercises is that it is very difficult to communicate specific aspects of how certain tools need to be set-up. Another problem with written exercises being done during lectures is the speed at which they are completed. There may be a large variance on the speed at which the exercises are completed as different students may complete the task faster or slower than other students. This may lead to students finishing quickly and idling during the lecture or other students feeling pressured to complete the exercise faster instead of ensuring that they learn the objective of the exercise. This

would defeat the purpose of the exercises as they are meant to make lectures more engaging and to ensure that the students have the tools necessary for their assignments.

The solution to this problem is to create video exercises. Each exercise has a main objective to be completed and follows a demonstrator performing the actions in ‘real time’. In addition to showing what the demonstrator is doing, it also verbally guides them through the exercise, explaining certain aspects that may be more complex to the student. This allows students to get a visual of how certain tools are set up, as well as having a more controlled time for completion (the length of the video). This solution also has the benefit of being accessible at whatever time that is convenient to the student if they were not in attendance on the day of the exercise. This approach has been taken with all the required tools and some concepts that will be utilized in Comp 4004.

### **6.3 Setting up the Environment and TDD with Junit**

This section will focus on the initial technologies used for the environment setup. Git and GitHub are used as a tool to monitor the Test-Driven Development methodology (TDD). Eclipse Integrated Development Environment (IDE) will be the workspace used and all additional tools are installed with respect to the Eclipse IDE. After the workspace is properly setup, the first tool used to demonstrate TDD is JUnit. JUnit is a testing framework which allows unit tests which are a core part of software testing. The section’s sub-sections below will explain in-depth the tools and concepts being used for this exercise.

#### **6.3.1 Test Driven Development (TDD)**

This exercise covers what TDD is and how to replicate the TDD ideology and incorporate it into your own work. TDD requires the tester to (1) write the tests first, (2) run the test, (3) write new code for the test to pass as the test initially fails because it has not been implemented, and (4) then finally refactor the code. This process is then repeated until testing is complete. It

would be difficult to monitor TDD without certain tools as there is no effective way to ensure that the student has followed the TDD methodology when writing their assignments. To remedy this problem, we will have the students begin the semester by setting up their work environment so that it is possible to check if they have been following the proper conventions.

### **6.3.2 Git and GitHub**

GitHub is a software development platform which will allow students to upload their work to a directory that is located on the platform. With Git, they can easily clone a directory from their own computer to the platform and can update their code on the platform with Git. Given permission by the user, GitHub displays the commits that are done on a directory and what is included in each commit; with this, it is possible to check the consistency of the tests that the students write and it is easy to identify that they are following a TDD methodology.

### **6.3.3 Eclipse and JUnit**

The final component of the setup is installing the Eclipse Integrated Development Environment (IDE). With this IDE, students can easily install additional tools that are used for software testing and quality assurance. Along with being able to install certain tools, Eclipse has its own marketplace that allows for easy installation of certain course tools such as TestNG and Cucumber. Eclipse IDE is commonly used for Java development and there are many helpful tools that come with it. After the Eclipse installation, the students will need to create a Maven project in Eclipse and add JUnit to the dependencies file. Once that has been completed, JUnit is now usable in Eclipse and they can begin writing tests to mimic TDD.

### **6.3.4 Creating Exercise One**

Once all the components were gathered, the video could then be recorded. The video begins with creating a GitHub account followed by installing Git and Eclipse. A new directory is then created on GitHub which is then cloned onto the desktop's eclipse-workspace directory with Git. Following this, a Maven Project is created on Eclipse using the directory that was previously cloned by Git; this allows the user to clone / push the new code made on Eclipse to GitHub. When a Maven Project is created, a pom.xml file is also added to the directory; with this file, JUnit can be added to the Maven dependencies. Once JUnit is added to the Maven dependencies, it can be noted that new files appear in the Maven Dependencies directory: those being JUnit and hamcrest-core. With new files added to the directory, a demonstration with Git is performed to show how to add the changes and then push them onto GitHub. Following the installation, a few simple test cases are created and ran. After the test cases failed, the newly created files are pushed onto GitHub with Git to demonstrate the TDD methodology. Following the push to GitHub, the video covers quick run-through of JUnit and how to write test cases with it. The video exercise provides a walkthrough of how to get the Eclipse, Maven, and JUnit working together as well as explaining the process of how files are cloned to GitHub with Git.

### **6.4.1 Running Tests and Creating a Test Suite in Junit**

This section will explain the importance of running tests, as well as how to parse the test results. Understanding the test results is an important part of the troubleshooting process as it allows the tester to determine what has caused the problem. In addition to running tests, this section will cover creating test suites, which allow for customization of which set of tests would be run.



## 6.4.2 Running Tests

Running tests and being able to identify the results of the test is important for diagnosing the potential errors throughout the system's code. After running a JUnit test, the tester can see that their test cases return either a green result (Fig. 1) or a red result (Fig. 2). The green result indicates that the test has passed, and the red result indicates that the test has failed. It is important to note that failed test cases show which test has failed and displays to the tester why it failed (Fig. 3). With this information, it should give a rough idea on where the problem is occurring so it may be fixed.

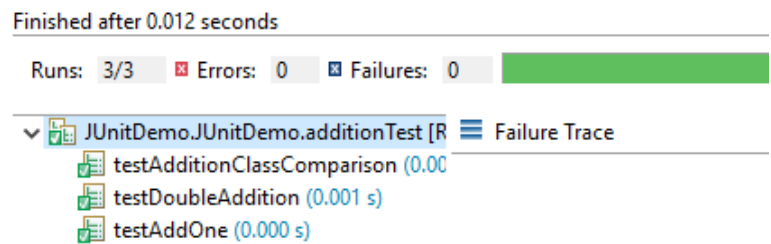


FIGURE 1  
Green Test Case

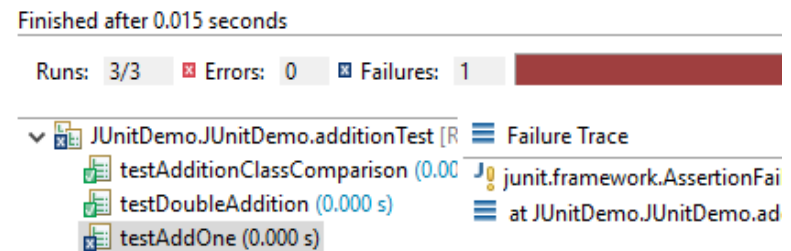


FIGURE 2  
Red Test Case

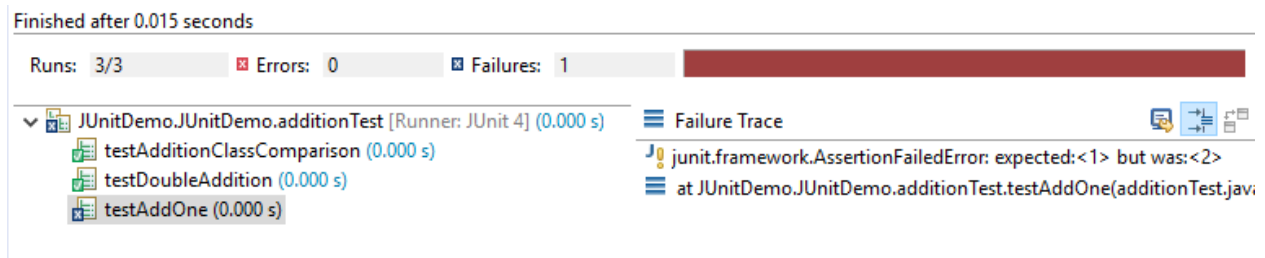


FIGURE 3  
Failed Test with Failure Trace

### 6.4.3 Creating Test Suite

It is beneficial to separate test cases into separate files as it is easier to manage the test cases. One drawback of creating multiple test cases is the requirement of running multiple tests at the same time. It is time consuming to run each test case individually to demonstrate that all the tests have passed. With JUnit, it is possible to create a test suite that allows the tester to run all of the test cases included in the test suite. This allows the tester to stay organized when creating test cases as well as being able to efficiently demonstrate that all the test cases have passed. In addition, test suites allow a tester to easily determine if any refactored code has produced an error in any other part of the system. When running a test suite, it displays all of the test cases as well as all of the test methods inside of those cases. This allows for easy identification of the test cases seen in (Fig. 4).

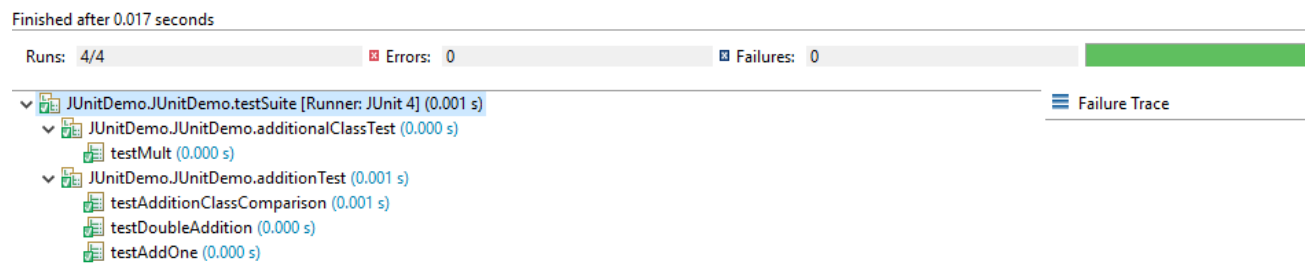


FIGURE 4

List of Tests ran in Test Suite

### 6.4.4 Creating Exercise Two

With this exercise, the objective is to ensure that the students understand how to run tests with JUnit, be able to diagnose whatever problems that may occur, as well as create a test suite so that they may implement it in their own work. The nature of this exercise is to get accustomed to the tools provided (JUnit and Eclipse); therefore, there is no need for the student to spend time creating the Java classes that will be tested. Skeleton test cases are provided for the student to download, but they are missing the crucial components that JUnit requires to run the tests. The

students begin the exercise by downloading the files and then following the video and the comments provided to implement the methods required for JUnit to work. Once they have written the test cases, they run the file as a JUnit test and can see that their tests either pass or fail. If there is a failure, JUnit shows what problem caused the test case to fail and allows the students to diagnose the problem in their own work. The video covers the basic JUnit methods, such as assertEquals() and assertEquals(), which allow the tester to compare if two objects are either equal or the same. This is usually used to compare the expected result to the one that the program has returned. Finally, the video demonstrates how to create a test suite and how to add test cases to said test suite. By completing this exercise, the student will have had experience with running tests / test suites and analyzing the results of the test cases.

### **6.5.1 Creating logs with Log4j**

This section will cover the logging tool Log4j. Log4j is another important tool to be accustomed with when working in environments that do not have a dedicated debugger. It does this by taking in line log methods and writing their input into a log file. The format of how the logs are written to the log file can be configured in a log.properties file. The exercise for this tool includes the initial installation and setup as well as creating logs in a simple Java program.

### **6.5.2 Log4j**

Log4j is a logging tool for Java. Before using Log4j it must first be installed and log.properties file must be created, this file is essentially the rules that Log4j will follow when it is used to create logs. The properties file allows the tester to filter out different log levels as well as add helpful information such as which Java class has thrown the log. This tool is extremely helpful in situations where the system that is being used does not necessarily have a debugger. As the log statements are written in line with the rest of the code, it is easy to identify any faults that may occur during the testing period as it is easy to read the log file and trace the path of the log messages.

### **6.5.3 Creating Exercise Three**

The objective of this exercise is to allow the students to understand what the tool did and how to set up the tool in the Eclipse environment. The video begins by explaining how to navigate the Apache Log4j website in order to find the correct file to add to Eclipse's Java build path. Once everything is installed, the video demonstrates how to create a log4j.properties file along with a few lines of code to populate the properties file with. Following the creation of the properties file, a Java class is created to demonstrate how to create logs and how to insert them into the code. Throughout the demonstration, the following are covered: creating different logs with different log levels, how to use the properties file to filter out unnecessary log messages, and how to use the conversion pattern to display the requested information along with the log. By completing this exercise, students learn how to write log files with Log4j and will have another tool when trying to test their work for faults.

### **6.6.1 TestNG Integration**

This section covers the testing framework TestNG. It is similar to JUnit in basic functionality; the tests ran are similar in structure. However, TestNG has many additional methods that are potentially helpful to the tester. As TestNG is similar to JUnit, the transition from JUnit to TestNG should be beneficial as TestNG's additional tools allow for more specific test scenarios. This increase in control can allow a tester to run tests where certain pre-conditions must be met before other tests are ran.

### **6.6.2 TestNG**

TestNG is a testing framework which is inspired by JUnit; it holds some similarities as it uses very similar ways of creating tests. In addition to being able to test code with results similar

to JUnit, TestNG comes with a variety of additional features that allows the tester to run their test code in a more specific manner. TestNG uses annotations which can be used for a variety of different things. Annotations are mainly used to run tests; some annotations allow the tester to control the order in which test cases are run to ensure that the sequence of tests matches what the tester was planning. Other annotations allow for helpful functionality. For example, the `@DataProvider` annotation allows the tester to supply the connected test method with a list of multiple variables that needed to be tested. In addition, TestNG comes with its own test suite which has more control than the test suite with JUnit. TestNG's test suite allows the tester more control of which tests need to be included in the test suite; it does this by allowing the tester to group the test methods into specific categories. The tester may then select the specific tests that they would like to run by calling their project folder, class, or even group name.

### **6.6.3 Creating exercise Four**

At the beginning of this exercise's creation, the method of installation was the first problem. There are numerous of methods to install TestNG but installing TestNG via the Eclipse Marketplace is the fastest and simplest option. After installation, the video begins by demonstrating how to use annotations. The `@Test` annotation was first as it is a very important annotation for testing purposes. Other annotations were covered later in the video: `@BeforeSuite`, `@AfterSuite`, and `@DataProvider` were covered in the exercise. TestNG's test suite is built differently from JUnit's test suite and has different features as well. After finishing the annotations examples, the video demonstrates how to create a test suite with TestNG as well as how to modify the test suite to include different categories of tests. By completing this exercise, students will have learnt how to use TestNG's annotations to create more effective test cases.

### **6.7.1 Web App Automation with Selenium**

This section will cover the Selenium tool and the web driver associated with it. With the two systems, a tester can perform tests on web applications. Selenium allows for the automation

of web browsers; with this tool, it is possible for a tester to imitate human interaction with a web application. It may do this by performing actions similar to a human such as inputting text and clicking on various web elements. This process can be useful in finding potential faults in a web application. The following chapters explain the tools used and how the exercise was created. The exercise covers the basics of Selenium and gives a notion of the things to check for in a web application.

### **6.7.2 Selenium**

Selenium is an automated tool which allows for a tester to test web applications through web drivers. Instead of needing to manually open a web browser and entering the information required to perform the test, the tester can give instructions to Selenium so it may perform the task automatically. For Selenium to run properly, it must also have a web driver installed so Selenium can launch the correct web browsers to perform the tests. With the Selenium client and both server files and the web driver installed, the tester may then insert Selenium code into their own work to begin using it.

Selenium functions by performing tasks in the sequence that the tester has set. Most web applications have elements that a user would interact with which would lead to certain functionalities being performed. For Selenium to detect the correct elements to modify, the tester must first discover information regarding the element from the webpage. The tester may do this by viewing the web page and selecting the ‘inspect element’ in certain web browsers to gain insight on the web page’s elements (Fig. 5). Once the tester has determined which elements need to be modified by the user, they may use Selenium’s methods to either send information to the elements or perform actions such as a click on the element.

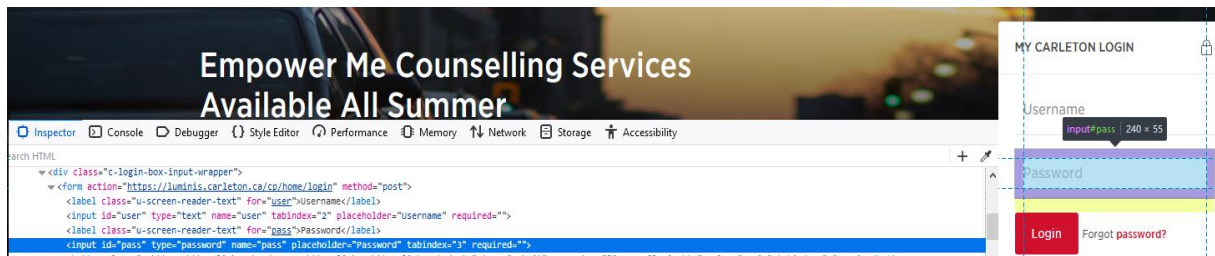


FIGURE 5

Web Element Information with Inspect Element

### 6.7.3 Potential Web App Problems

When testing web applications, there can be a few things that can go wrong. Some fields may have incorrect information, or the information can be in an entirely different format. It is important to test a variety of situations that may occur in case one of the situations leads to a fatal fault that would cause problems to the web application such as a crash or corruption of data. When testing applications, it is important to cover all the possible scenarios as they might be a potential threat to the system if they were not designed / tested properly. The more complex the system becomes, the more potential scenarios that need to be tested. Web applications are no exception as there are many possible outcomes that a human can perform when interacting with the application. In this exercise, the importance of covering multiple scenarios is addressed.

### 6.7.4 Creating Exercise Five

The objective of this exercise is to demonstrate how students can use Selenium to test web applications. The video begins by walking through how to install all the proper components to setup Selenium in the Eclipse environment. Following the installation, the students may download a Java class file which they can begin using Selenium with. This exercise begins with the students using a web driver to go to Carleton's student webpage. They then run the code to understand how the driver opens the webpage. While on the webpage, they learn how to use the

inspect element option to discover the relevant information on certain webpage elements. Once they have noted the reference to the element, they can return to the Java class file and begin giving instructions to the web driver. With the `findElement()` method, the student may instruct the web driver to locate the element on the webpage and then perform an action to that element. The student is taught how to fill out the webpage with a username, a password, and how to click the login button with the web driver to complete the login to Carleton. They repeat this cycle for an invalid login (with improper information), invalid credentials login (name and password do not exist), and finally a correct login (correct name and password). There are a few more conditions to be tested such as correct name and incorrect password, but to keep the video short, it only covers three scenarios. By completing this exercise, students learn how to use Selenium and the web driver as well as having an idea of the different things that one must account for when testing a web application.

### **6.8.1 BDD with Cucumber and Client / Server system**

This section covers Cucumber, a tool that allows for Behaviour Driven Development (BDD). The section covers what BDD is, as well as how Cucumber enables BDD. In addition, the section covers the difficulties that may occur during client / server testing. The exercise created in this section demonstrates how to use Cucumber in conjunction with a client / server environment.

### **6.8.2 Behaviour Driven Development (BDD)**

Behaviour Driven Development is a process which encourages communication between different groups of a project. With the increased communication between different groups, it may lead to less vague requirements from the client so the developers can deliver a more accurate product. The increased collaboration between different groups of a project may also reduce the ambiguity of the project; if all parties involved with the project understand the goal and create examples together, they will all be working towards completing the same tasks. Examples are



stories with rules: the rules are followed, and a result is obtained at the end. The examples created by collaboration can also allow different groups to discover the faults that can occur in the system. With the examples produced they can then be created into automated tests with Cucumber.

### 6.8.3 Cucumber

Cucumber is a tool which allows the tester to follow BDD. Cucumber files have the '.feature' file extension. This file extension tells Eclipse that the file is a Cucumber feature file and will run in a specific way. Feature files are written in plain language, which is understood by everyone on a team, as there is no technical jargon which may confuse non-technical members of the team (Fig. 6). A Cucumber feature file uses Gherkin's keywords to give the feature file proper functionality. A scenario (example) follows this pattern: Given, When, Then. This pattern allows the team to easily define scenarios and what happens at which step. When the scenarios are completed, there is no actual functionality behind the feature file, and the tester will then implement the step definitions (Fig. 7) to give the feature file its functionality. When the feature file is shown to other members of the team, they understand the plain language of the feature file and can visually see the test being performed.

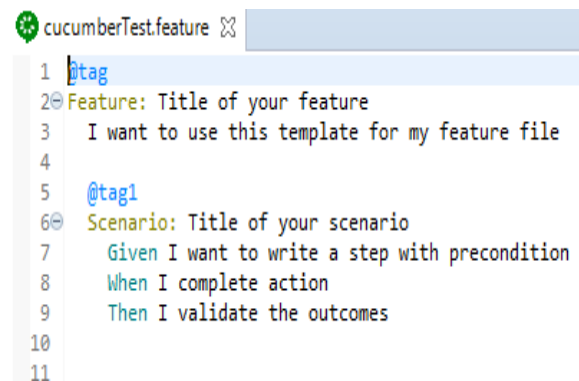
A screenshot of an Eclipse IDE editor showing a Cucumber feature file named 'cucumberTest.feature'. The file contains a Gherkin scenario with the following text: Line 1: '@tag', Line 2: 'Feature: Title of your feature', Line 3: 'I want to use this template for my feature file', Line 4: (empty), Line 5: '@tag1', Line 6: 'Scenario: Title of your scenario', Line 7: 'Given I want to write a step with precondition', Line 8: 'When I complete action', Line 9: 'Then I validate the outcomes', Line 10: (empty), Line 11: (empty). The text is color-coded: '@tag' is blue, 'Feature:' is green, 'I want to use this template for my feature file' is black, '@tag1' is blue, 'Scenario:' is green, 'Given I want to write a step with precondition' is blue, 'When I complete action' is blue, and 'Then I validate the outcomes' is blue. The editor has a light blue background and a white border.

FIGURE 6

Cucumber Feature File

```

1 package CucumberInstallation._CucumberInstallationCompleted;
2
3 import cucumber.api.java.en.Given;
4
5
6
7 public class glueCodeImplementation {
8     int number=0;
9     @Given("I want to write a step with precondition")
10    public void i_want_to_write_a_step_with_precondition() {
11        number=number+1;
12    }
13
14    @When("I complete action")
15    public void i_complete_action() {
16        number=number+2;
17    }
18
19    @Then("I validate the outcomes")
20    public void i_validate_the_outcomes() {
21        if(number==3) {
22            System.out.println("correct answer");
23        }
24    }
25 }

```

FIGURE 7

Cucumber Feature File Step Definitions

#### 6.8.4 Client and Server Testing

When creating a client and server environment, there are a multitude of potential errors to test for. Aside from the usual unit tests that need to be performed, there are also new errors that exist in client / server environments. Some problems include client disconnections, server disconnections, port problems, timeouts, and packet loss. When testing in a client / server environment, it is important to have proper tests for these systems in addition to testing the system itself.

#### 6.8.5 Creating Exercise Six

The objective of this exercise is to demonstrate BDD with Cucumber as well as some of the problems to account for during client / server development. The exercise begins with the

installation steps for Cucumber which include a Maven dependency as well as installing the Cucumber Eclipse plugin from the Eclipse marketplace. Once installed, the first task is to create a feature file. The video demonstrates how to create a simple feature file and how to create the associated step definitions. Following the initial introduction video, the second exercise goes into more depth regarding Gherkin's keywords such as Background and Example Tables. The Background keyword allows for all features to have a preset 'background' (initial rule); these initial rules reduce the redundancy of steps in the feature file. When the Example Table has elements, the keyword allows the tester to duplicate scenarios without manually creating different scenarios for each element in the table. Once the Background and Example Table examples are finished, the next video exercise includes Cucumber and its usage with a client / server application.

In the final exercise, once the files that include the client and server are downloaded, feature files need to be created. The client / server system used for demonstration is simple in its functionality. The server is active on a port number specified by the user (port 123 is used in the video) and the client can connect to it when it is started. The client then sends a specified number to the server (the number 12 is used in the video). The number is then squared by the server and then returned to the client. Cucumber tests are then written to ensure the functionality of the system is correct.

To test the functionality of the code along with the logic, JUnit is used to ensure that the results to be received match the expected results. Cucumber features were written with the standard Given, When, Then format. An example of the client feature file is as follows: Given (the server is on), When (the client sends the number five), Then (the client receives the squared number twenty-five from the server). This feature demonstrates the expected results that the client would receive under the rules that have been provided. A different scenario regarding invalid numbers is done for the client as well as two different scenarios for the server. One scenario depicts a server timeout, and another ensures that the server recorded the squared number properly. One problem that was encountered during the server scenario testing was that the JUnit test to check for the squared number would occur before the server performed the arithmetic. To remedy the problem, the client thread was called via the `.run()` method instead of

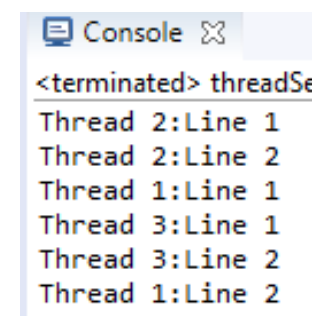
the `.start ()` method. This halts the program until the client has terminated; this ensures that the server has finished its arithmetic as the client would not terminate before receiving a result. The video exercises demonstrate how to install Cucumber, how to create feature files, how to use the Background and Example Table keywords, as well as demonstrate how to use Cucumber in a client / server environment

### 6.9.1 Threads, Race Condition, and Deadlocks

In this section, the basics of Java threads are explained and how they allow for concurrent programming. The section also covers two important faults that can occur during concurrent programming: race conditions and deadlocks. The solution to the two faults is also mentioned in their respective sections. The final video created in this section is not an exercise but rather a video explaining how the threads function and the solution to the problems that occur in concurrent development.

### 6.9.2 Threads

In concurrent programming, many actions are defined and performed simultaneously. To perform this in Java, we need to create Java threads. Threads are used as an independent execution of defined code. Meaning that if multiple threads of the same code are run, that instance of code will be run for each initialized thread. The time which threads run is random (Fig. 8) and is out of the control of the tester. This is because an internal Java scheduler handles the Java threads. Additional attention needs to be added when working with threads as concurrent programming opens a lot of more potential problems that may not be initially apparent.




```
Console 
<terminated> threadSe
Thread 2:Line 1
Thread 2:Line 2
Thread 1:Line 1
Thread 3:Line 1
Thread 3:Line 2
Thread 1:Line 2
```

FIGURE 8  
Console Output of  
Numerically  
Initialized Threads

### 6.9.3 Race Conditions

Race conditions are one of the problems that occur when working in a concurrent environment. Race conditions occur when two or more different threads try to manipulate the same object at the same time. This occurs when there are no prevention steps taken as threads do not run in a specific order. This may lead to a crash or an improper change of data. We can use two threads to illustrate this point. In this example, we will have two threads and a box with a single object. If the thread sees an object, it will then grab the object from the box. Thread one checks the box and determines if there is an object, and it then begins to grab the item. Thread two also checks the box after thread one checks it, and it proceeds to grab the object as the object has not yet been removed by thread one. Thread one grabs the object and then thread two crashes the program as it is grabbing a null value as nothing exists in the box anymore. To solve this problem, the keyword Synchronized can be used to protect the box object so that only one thread can access it at a time. This will allow thread one to grab the item, and then thread two will see that there is no item to grab and stop. Although synchronization prevents race conditions from happening, another problem can occur in concurrent programming, namely deadlocks.

### 6.9.4 Deadlocks

Deadlocks occur when there is a fault in the system that prevents the system from moving forward. This can occur when certain threads are waiting for resources to move forward in their execution. If the resources are never returned to the waiting threads, then they cannot continue their execution and the entire system may halt. To remedy this problem, it is essential to be aware of where the problem may occur. When a thread enters a synchronized method / object, it is essential that the thread must release its hold on the method if it is no longer using the resource. The release of a synchronized lock is done by calling the thread's wait method. The thread calling the method will release the lock on the object and allow another thread to use it. It is important to remember that waiting threads are suspended and need to be awoken with a notify method to begin running again.

### 6.9.5 Creating Video Seven

The objective of this exercise is to explain the basics of concurrent programming. The video begins by introducing what threads are and is then followed by an example of a race condition, and finally a deadlock situation. During the race condition proportion of the video, the race condition is demonstrated with four threads trying to remove three items from a box object. The program crashes as the final thread attempts to remove an item that does not exist. In addition to the race condition, this example demonstrates the randomness of Java thread (with numbered threads) and that it is observable that they all run in a random order despite being initialized numerically. The solution to the race condition is then demonstrated with the use of synchronized objects and with a combination of `wait()` and `notifyAll()` methods. The `wait()` method is called upon the threads that obtain the lock but cannot move on as there are no resources to obtain from the box. Thus, it relinquishes the lock on the box to another thread. The threads that obtain an item from the box hold onto the item for a few seconds before returning it to the box and then call `notifyAll()`. The `notifyAll()` method then awakens all waiting threads and the first thread scheduled by the Java scheduler will obtain the lock and the item, while the following threads will enter the waiting state once again. The program is run with ten threads and does not create a race condition / crash, nor does it run into a deadlock.

The next example was to display a deadlock situation; with the previous example no deadlock occurs as all the threads call the `notifyAll()` method on completion. Once `notifyAll()` is called, the waiting threads are notified so that they may begin running once more. To demonstrate a deadlock, when the system begins and the first few threads obtain the lock, the threads are suspended indefinitely. This ensures that the threads do not terminate their code and call the `notifyAll()` method. Once suspended, it is observable that the program has not terminated, but as all the resources are held by the suspended threads, the program cannot move forward. One additional deadlock example was shown afterwards; in this program, four different threads desire two objects each. Two different threads grab object one first while the others grab object two first. The deadlock occurs when a thread grabs object one and then waits for object two, while the reverse happens with the other thread. This creates a deadlock as a thread requires

object two to continue, while the other one requires object one to continue. Neither one can move on as the other resource is in use and the program halts. A visualization of the deadlock problem can be seen in the figure below (Fig. 9). The figure demonstrates ‘**Thread 1**’ halting, but it should be noted that ‘**Thread 2**’ is also halted as it cannot obtain ‘**Item 1**’ due to the same reason. This video demonstrates Java thread usage as well as common problems and solutions that can occur in a concurrent system.

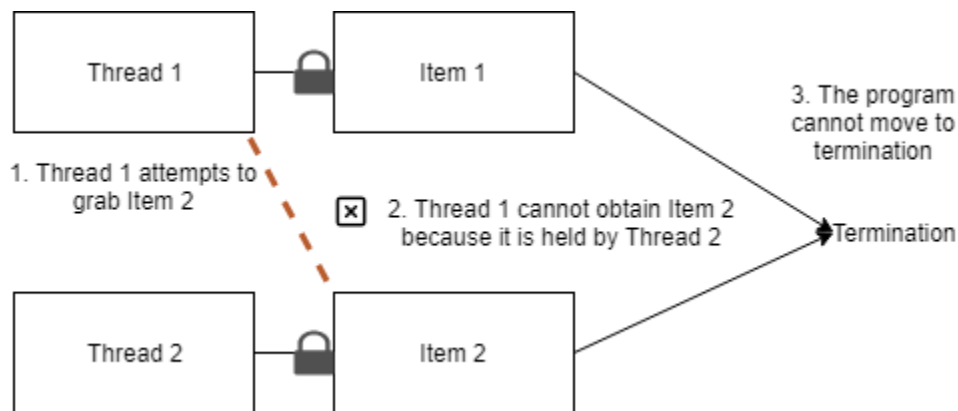


FIGURE 9

Deadlock Example with Two Items

### 6.10.1 TOTEM Methodology’s Issues with Concurrent Systems

This section covers the difficulties in modeling concurrent systems with the Testing Object-oriented Systems with the Unified Modeling Language (TOTEM). The section provides an overview of the TOTEM methodology and how it is difficult to capture concurrent systems with it. There is no exercise created from this section as it is mostly explanatory. The video explains the difficulties that are mentioned in depth in this section.

### 6.10.2 Testing Object-oriented Systems with the Unified Modeling Language (TOTEM).

The TOTEM methodology is a testing method proposed by Lionel Briand and Yvan Labiche [Lionel Briand and Yvan Labiche, 2002]. This test methodology derives the test cases from the artifacts that are produced at the end of the analysis stage in development. This includes

documents such as use case diagrams, unified modeling language diagrams, and class diagrams. With all these documents, it would be possible to derive the test requirements.

### **6.10.3 TOTEM Process**

In the TOTEM paper [**Lionel Briand and Yvan Labiche, 2002**], the writers explain the methodology with a library system as an example. At the beginning of the process, an activity diagram for the library is presented, and many sequences of events can be seen from the diagram. A sequence is a set of events that will occur in a potential use of the system. For example, a User can be added to the system, the user then borrows a book, then returns the book, and then the user is removed from the system. That covers one sequence, but there are many more as Users can renew loans, books can be removed from the library, etc. The different paths are then placed into a tree for easy identification of paths. From these paths, they have determined many potential sequences. The sequences can then be modeled for testing. The problem with concurrent programming begins when all the potential sequences are analyzed.

### **6.10.4 TOTEM Difficulties with Concurrent Systems**

Without adding concurrency to the problem, the TOTEM paper [**Lionel Briand and Yvan Labiche, 2002**] mentions the interleavings of sequences. When the sequences were listed, it can be noted that some are repeated instances in other sequences: the use cases sequences are combined, creating a large number of interleavings but creating complete sequences to be tested. This method of analysis was done under the assumption that there would be a correct sequence of events, as every line of code would run in order.

If someone were to analyze this system under a concurrent system, the number of interleavings would balloon out of proportion. As threads are random, the lines of code in each sequence may run randomly. For example, thread one might run line one, then thread two runs line one and two, then thread one completes with line two. That sequence may occur differently



depending on how the Java thread schedule schedules the threads; Table 1 shows potential execution paths in a theoretical program which has two lines of code. In the rather simple program, there are four sequences of code execution with the thread scheduler. If the TOTEM methodology allowed for concurrency, the number of interleavings would be too difficult to model as the number of potential sequences becomes too large.

“Sequence of activated lines with threads (T1 and T2)”

<b>Sequences</b>	<b>First Action</b>	<b>Second Action</b>	<b>Third Action</b>	<b>Fourth Action</b>
1	Line 1 (T1)	Line 1 (T2)	Line 2 (T1)	Line 2 (T2)
2	Line 1 (T1)	Line 2 (T1)	Line 1 (T2)	Line 2 (T2)
3	Line 1 (T2)	Line 1 (T1)	Line 2 (T1)	Line 2 (T1)
4	Line 1 (T2)	Line 2 (T2)	Line 1 (T1)	Line 2 (T1)

TABLE 1

#### 6.10.5 Difficulty in Recreating Faults in Concurrent Systems

If a fault is detected during the analysis of sequences, the fault could be found by tracing the sequences that occurred. As there is a specific sequence of events that occur to create the fault, it should be easy to reproduce it for testing purposes by simply repeating the sequences that lead to the fault. In a concurrent system, this process is more difficult to perform, as the sequence of events is determined by the Java scheduler. If the tester noticed a fault that occurs during a specific sequence, it would not be very plausible to ensure that the test can be automated as the Java scheduler may run a different sequence. This randomness prevents accurate testing for faults in concurrent systems which occur in specific sequences.

### **6.10.6 Creating Video Eight**

The objective of this exercise was to demonstrate the difficulties in testing concurrency with the TOTEM methodology. The video briefly covers the overview of the TOTEM system. Following the overview, the video covers the combinatory nature of interleaving sequences and how modeling a concurrent system would have the combinatorial sequences balloon out of proportion. Finally, the video mentions the problem in trying to fix faults that occur during a specific sequence of concurrent events. The automated tests would not always be able to spring the exact sequences as the Java scheduler runs the threads randomly. By completing this video, the main takeaway is that it is very difficult to model concurrent system with the TOTEM method.

## **7 Results**

The objective of this project was to enhance the learning experience during Comp 4004 lectures. Throughout the course of the project, video exercises and videos which explain certain concepts have been created. Instead of simply learning about the tools used, students may become accustomed to the tools during class with the video exercise. This will reduce the amount of trouble they encounter when they first begin to use the tools in their own project. As the video removes the troubleshooting required to install the tools as well as covers the basics of the tools, the student should not run into difficulties when attempting to setup and utilize the tool in their own works. Alongside the technical aspect, the videos cover certain conceptual aspects of different testing methodologies. Alongside the explanation of concepts such as TDD and BDD the videos demonstrate the methodologies being performed with the tools used with the course.

Alongside the videos, all written code is available if the student would like to attempt the exercises on their own time or would simply like to view the finished product. All the video recordings and example code will be available to the students and may be used as supplementary material with the Comp 4004 lectures. As these exercises have not yet been implemented, it is difficult to say that this approach will be successful; but even if the videos do not necessarily

enhance the learning experience, the videos and code can act as supplementary information to help the student in their studies with Comp 4004.

## **8 References**

Lionel Briand and Yvan Labiche (2002) A UML-Based Approach to System Testing, Software and Computer Engineering Department, Carleton University.

Jean-Pierre Corriveau (2019) Private Communication.

Connor Poland (2019) Private Communication.