**CS4227**

**Software Design and Architecture**

Automated Testing Framework

*James Reidy* 14147009

*Aoife Watson* 14159295

*Jiasen Tian* 16060245

*George Rickard* 14118637

**2. Table of Contents**

3. Requirements ................................................................................................................................

3.1 Scenario Outline .............................................................................................................

3.2 Discussion on architectural use cases ............................................................................

3.2 Discussion on tactics to support architectural use cases ................................................

4. Discussion on Interceptor Architectural Pattern and Design Patterns .........................................

5. System Architecture .....................................................................................................................

6. Diagrams ......................................................................................................................................

Architectural Diagram .........................................................................................................

Structural Diagram ................................................................................................................

Runtime Diagram ....................................................................................................................

7. Demonstration of quality attributes .................................................................................................

8. Added Value ....................................................................................................................................

9. Testing ..............................................................................................................................................

10. Problems encountered and their solutions .....................................................................................

11. Evaluation and critique ..................................................................................................................

12. References .....................................................................................................................................

13. Contribution ...................................................................................................................................

**3. Requirements**

**3.1 Scenario Outline**

This project is an automated test framework, to be used for testing front end web applications. It makes use of 3rd party APIs such as JSoup for scanning and extracting web elements from websites. These elements can then be stored in XML or JSON format. This gives the user the ability to construct tests from these collections of elements or edit previous tests. When running an automated test, elements are loaded from a file, read sequentially, and sent to the Selenium API that navigates a website from the input given which simulates a user and logs information of the validity of elements.

**3.2 Use Case Descriptions**

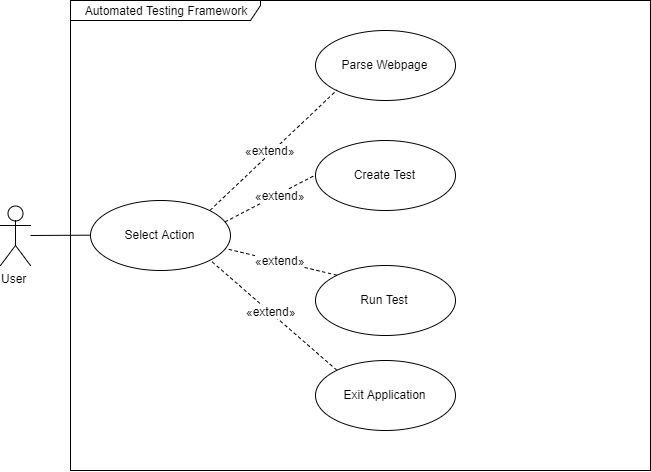
|  |  |
| --- | --- |
| **Use Case** | **Select Option** |
| *Success Scenario* | *System Response* |
| 1. User selects desired action | 1. System returns appropriate window |
| *Extension* |  |
| 1. Click “Exit” button | 1. System exits application |

|  |  |
| --- | --- |
| **Use Case** | **Parse Page** |
| *Success Scenario* | *System Response* |
| 1. Enter webpage URL 2. Enter a name for the parsed file 3. Select XML or JSON format 4. Click “Parse” button | 1. System checks validity of URL 2. System parses page into desired format 3. System returns message indicating successful parse |
| *Extension* |  |
|  | 1. System returns message indicating unsuccessful parse |
| 1. Click “Main Menu” button | 1. System presents main menu |

|  |  |
| --- | --- |
| **Use Case** | **Create Test** |
| *Success Scenario* | *System Response* |
| 1. Select page elements to test 2. Enter inputs/actions to take on each element 3. Select XML or JSON format 4. Click “Save Test” button | 1. System checks if path exists 2. System saves tests 3. Test name and path are written to text file |
| *Extension* |  |
|  | 1. System returns error message indicating that a test with that name already exists |
| 1. Click “Main Menu” button | 1. System presents main menu |

|  |  |
| --- | --- |
| **Use Case** | **Run Test** |
| *Success Scenario* | *System Response* |
| 1. Select desired test to run 2. Click “Run” button | 1. System runs tests and logs results |
| *Extension* |  |
| 1. Click “Main Menu” button | 1. System presents main menu |

**Use Case Diagram**

****

**3.3 Discussion on Architectural Use Cases**

1. Parsing a web page

Users must be able to parse a web page of their choosing. The parsed webpage elements can be saved in either XML or JSON format.

2. Creating a Test

All elements from the parsed web pages are displayed in a table. Users can pick elements from this table to add to a test, or insert their own. All tests must have a unique name and tests with duplicate names will not be saved.

3. Running a test

All saved tests appear in a list. The user picks one and clicks the run test button.

4. Log messages saving to the log file

Running a test returns a success or fail string. The logger takes this string and saved it to the log.txt, along with the date, the test name. For tests containing multiple elements, a success or fail note is included for each element.

**3.4 Discussion on Tactics to Support Architectural Use Cases**

**Performance**

We support performance through our use of a QueryBuilder. The QueryBuilder was used to extract information from a text file. Statements are easy to prepare using the QueryBuilder. Statements can be prepared in advance, with execution occurring at a later stage. This improved performance as we were not executing any unnecessary file manipulation.

**Usability**

Usability is supported via two methods in our application. One method is the inclusion of an undo method when creating a test. An user can add an element to test, click the undo button and the element will be removed from the test.

Usability is also supported through our use of JavaFX. JavaFX is a graphical user interface (GUI) builder. This made building the GUI extremely quick and painless.

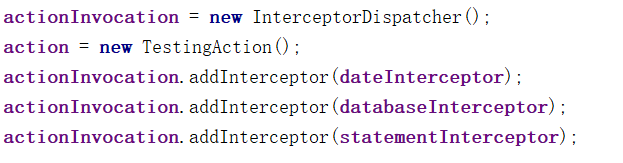
**4. Discussion of Interceptor Architectural Pattern and Design Patterns**

**Interceptor Architectural Pattern**

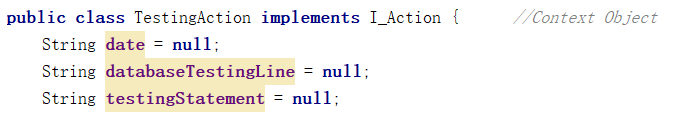
In the field of software development, an interceptor pattern is a software design pattern that is used when software systems or frameworks want to offer a way to change, or augment, their usual processing cycle.

Key aspects of the pattern are that the change is transparent and used automatically. In essence, the rest of the system does not have to know something has been added or changed and can keep working as before. To facilitate this, a predefined interface for extension has to be implemented, some kind of dispatching mechanism is required where interceptors are registered and context objects are provided, which allow access to the framework's internal state.

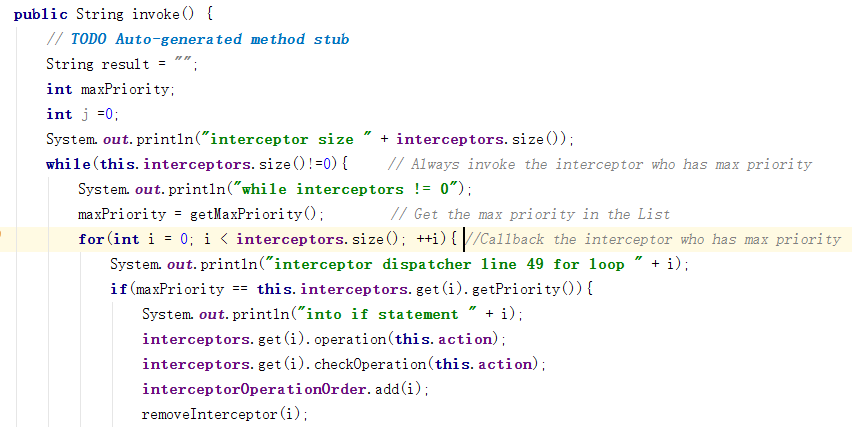
In our project, we used Interceptor Architectural Pattern to implement a logging operation. There are three different interceptors - *DateInterceptor* (logs current time and date), *DatabaseTestingLineInterceptor* (logs test name) and *TestingStatementInterceptor* (logs test result of each element). These interceptors are added to the dispatcher as specific out-of-band services and they will use the context object (called *action* in the code snippet below) to control the concrete framework.



I our application, the Context Object class is called *TestingAction*. *TestingAction* allows access to the framework’s internal state. For instance, a *TestingAction* object will get database testing information and testing result from testing framework which will be operated by different interceptors.

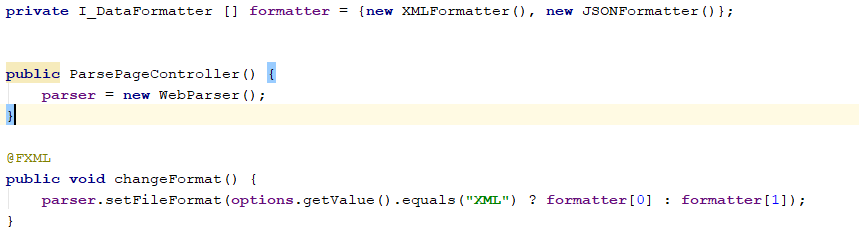


The *InterceptorDispatcher* allows applications to register and remove concrete interceptors. It uses a priority callback strategy - each interceptor has its own priority. The dispatcher always invokes the interceptor who has the highest priority first. Below is a method that illustrates this.



**Strategy Design Pattern**

Our use of the strategy pattern was to pick different file formats when saving web elements, with either the choice of XML or JSON. Our implementation was completed using a *I\_DataFormatter* interface for *XMLFormatter* and *JSONFormatter*. The *ParsePageController* holds a reference to an array of concrete types with the interface and the *WebParser*.



The *ParsePageController* has a *WebParser* and an *I\_DataFormatter*. When a user selects one of the options XML or JSON, the *changeFormat* method is called which changes the concrete formatter type using the *setFileFormat* method in *WebParser* which has a different *convertFile* implementation.



**Command Design Pattern**

The command design pattern was implemented in two classes in our project. In the first instance, the command design pattern was used in conjunction with the factory design pattern to create the UI controller classes. This supports extensibility as adding another controller class to the factory method requires only one line of code in the *setHandlers* method.

The command pattern was also used to encapsulate commands for testing elements. Each type of element requires a different testing procedure. We put in place the command pattern to create a handler for each type of test. This also extends extensibility as adding a new element handler to this scenario requires no change to the runTest method in the TestCase class, as seen below.

**Factory Design Pattern**

We used the Factory Method Design pattern to create the UI controllers in our application. We did this as to capture all controller creation in one class. This was used with the command pattern as previously discussed, and with the Singleton design pattern. The singleton design pattern was used to ensure only one instance of the *ControllerFactory* was instantiated. This had the additional benefit of ensuring the controllers are only created once, they are instantiated in the *setHandlers* method, which only called once in the *ControllerFactory* constructor.

**Memento design pattern**

Memento design pattern can capture and externalise an object’s internal state so that the object can be restored to this state later, in such a way that does not violate encapsulation. In our project, the memento design pattern was used in an undo operation in select XML/JSON elements which be chosen as a test case of running test. This operation allows the user to ‘back out’ and recover from erroneous operations. There are three important aspects to the memento design pattern - the *DataOriginator* (object whose state changes), the *Caretaker* (object who changes the originator) and the *Memento* (previous state of the originator).

In the *DataOriginator*, the restore method will get the previous state of Originator. In the *Memento*, it can store the state of selection operation. In Caretaker, there are two important methods: *undoOperation* and *setOriginatorValue.* *undoOperation* gets a previous state from stack and restore it which will make originator return to previous state. In *setOriginatorValue,* the caretaker will create new memento to store the current state and push it into stack.

**Builder**

The Builder was implemented in the QueryBuilder system in our application. QueryBuilder constructs simple “Queries” to be performed on CSV files in an SQL-like fashion.

*SelectOperation* in the example shown takes in the columns to be selected, *TableTestCases* acts as a wrapper for the path to the text file containing the data, and *WhereClause* narrows the data down as the user desires. The SQL equivalent to the example shown would be “SELECT name FROM TestCases WHERE name = n”.

**Visitor**

As well as incorporating the Builder design pattern, the concrete *Query* the *QueryBuilder* constructs utilizes the visitor design patter to perform an action. This depends onthe *DataOperation* being performed. Shown below is the visitor being passed into the *dataOperation*. Once the visitor is passed in it performs the necessary action based on its type; select, delete, insert or update.

**Singleton**

The singleton design pattern was implemented for both the *FactoryController* and the *TestController*. This was because both of these classes needed to be instantiated only once. A *getInstance* method was invoked when any object needed to access either of these controllers.

**5. Discussion of System Architecture**

Our system is structured using an Model-View-Controller architectural pattern. In our application, as we are using JavaFX for the user interface (UI), the FXML files take the place of any view java files that would typically exist.

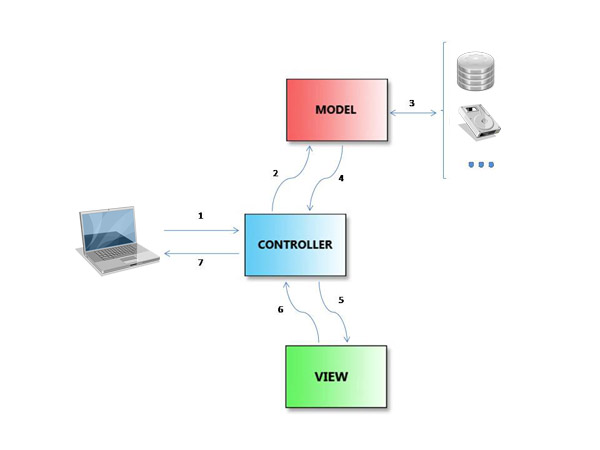


Figure 1. Adapted from [www.code.tutsplus.com](http://www.code.tutsplus.com)

We have two types of controllers in our application – UI controllers and model controllers. The UI controllers only interact with the FXML files, and delegate out any other responsibility to the model controllers. We have one model controller in our application as we have only one model class. The UI controllers and model controller are kept completely separate, with minimal dependencies. Only *TestSelectionController* and *CreateTestController* have a dependency on the model controller, as they need to delegate actions to it.

The model class in our application simulates an entity class from a database. It encapsulates the data in the TEST\_CASES.txt file. In this class, the QueryBuilder is used extensively to select, update and insert data to and from the text file. The model class is situated in the models package in our business logic layer (bll).

We also have a data access (dal) layer to our application. This packages contains the implementation of the QueryBuilder, and its related operations.

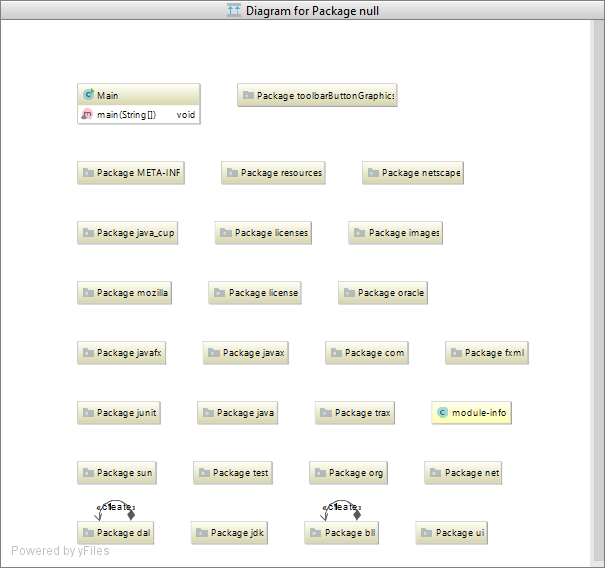
We have included a properties text file with our application. This file contains values for various external variables in our application. The advantage of using a properties file is that the values of these variables can be altered with needing recompilation.

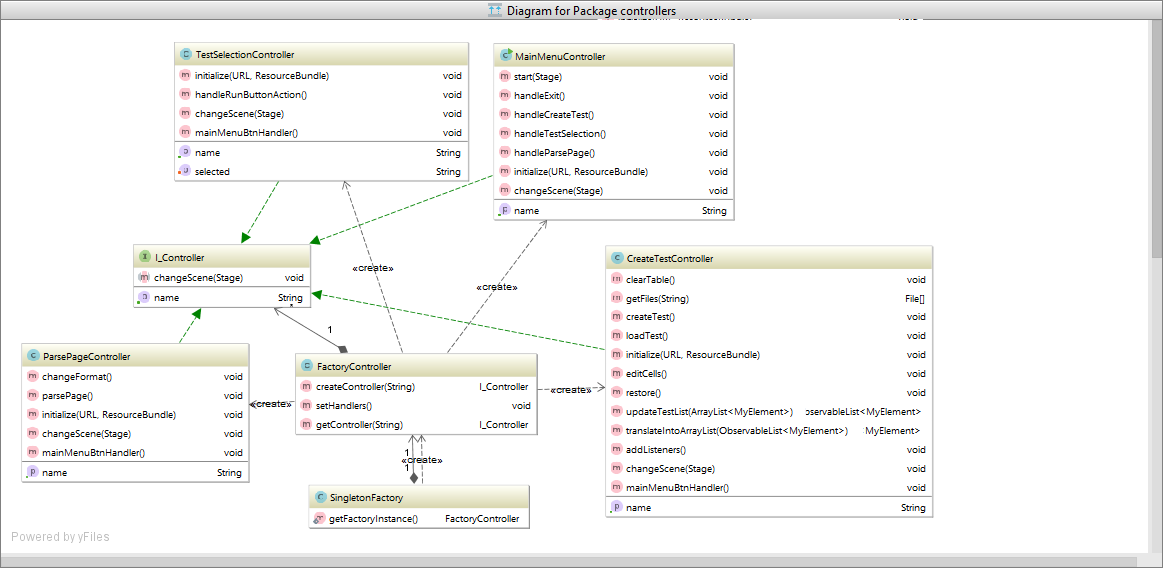
**6. Structural, Runtime and Architectural Diagrams**

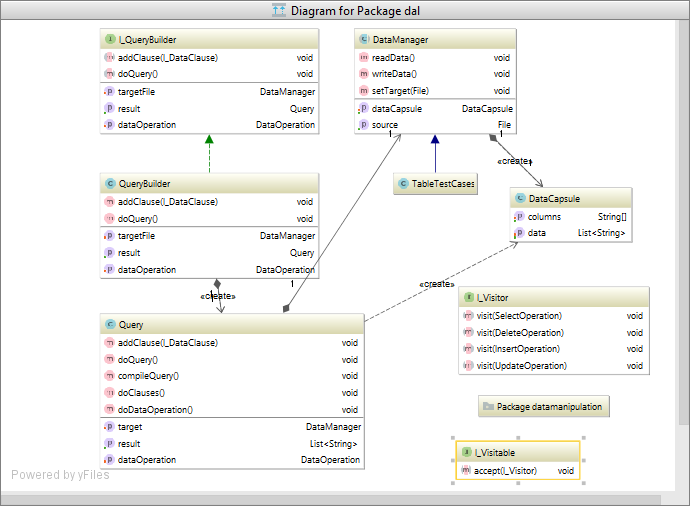
**Structural Diagram : Class Diagram**

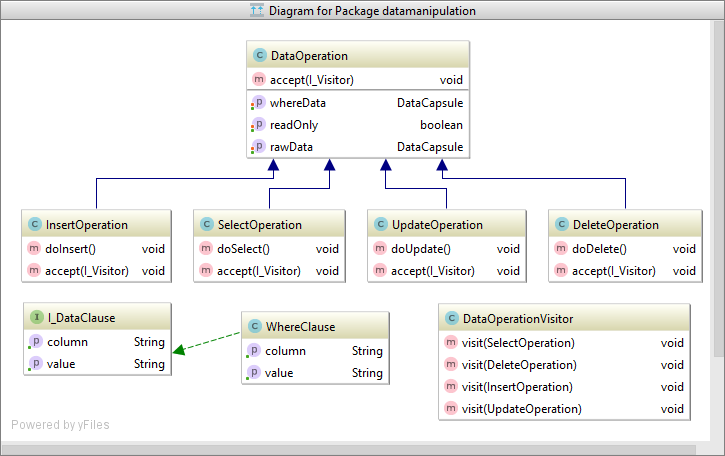
Package diagram for package src

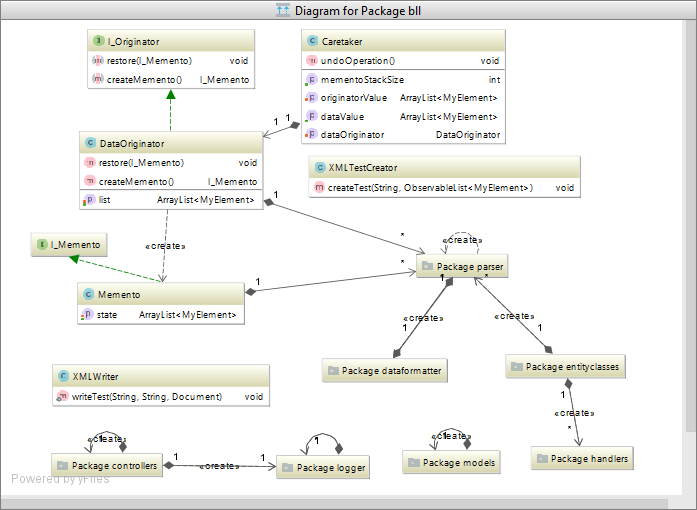
\*Hightlighted in the diagram below are the packages we implemented

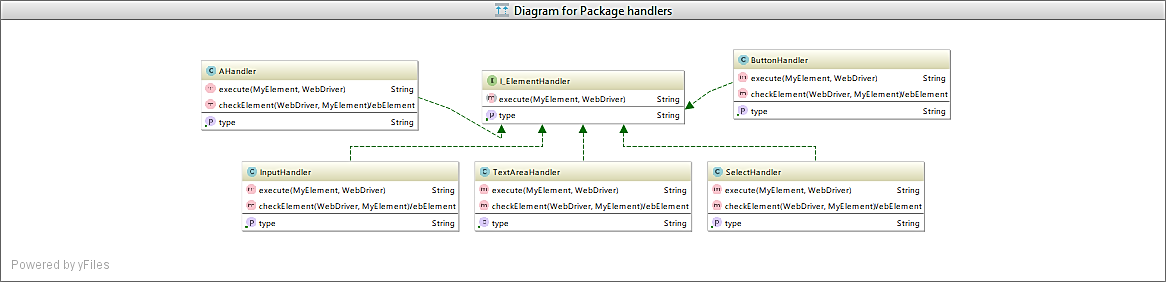


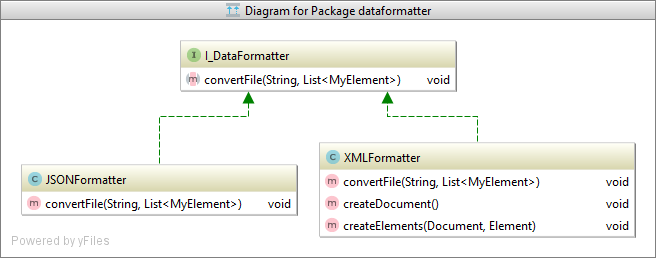


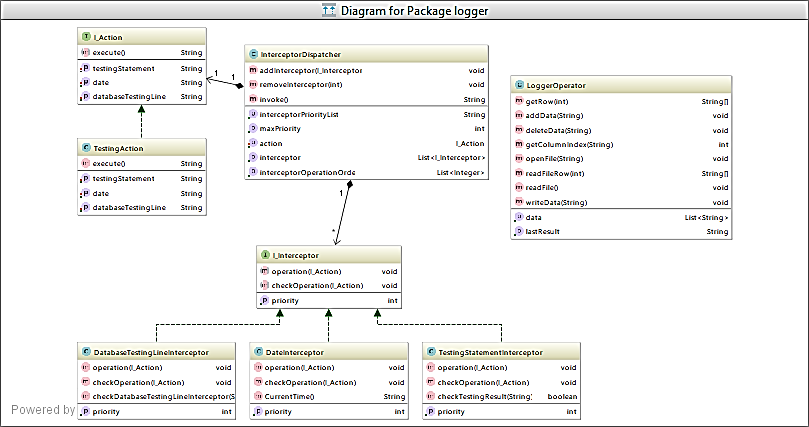


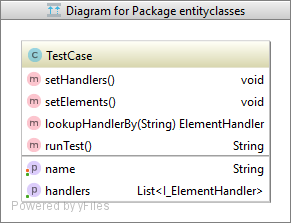
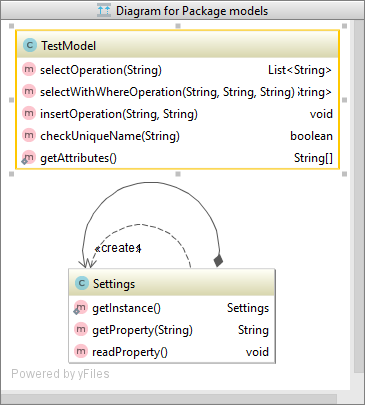


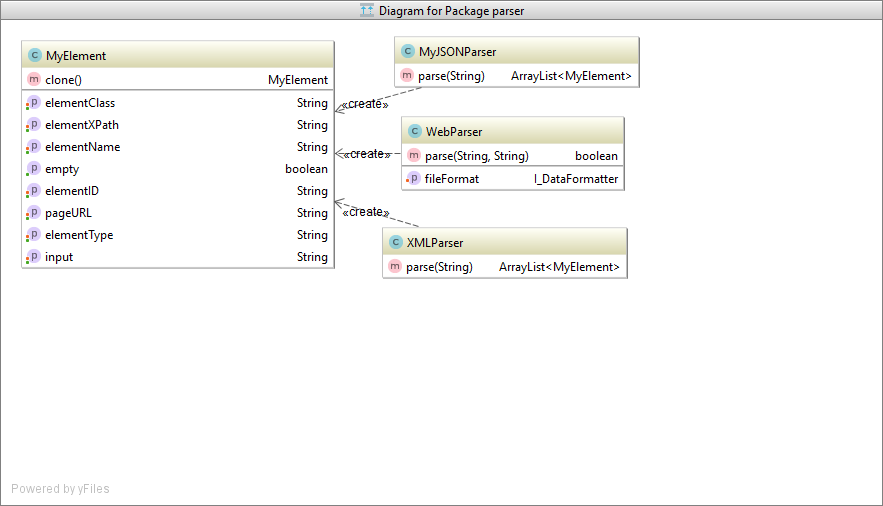






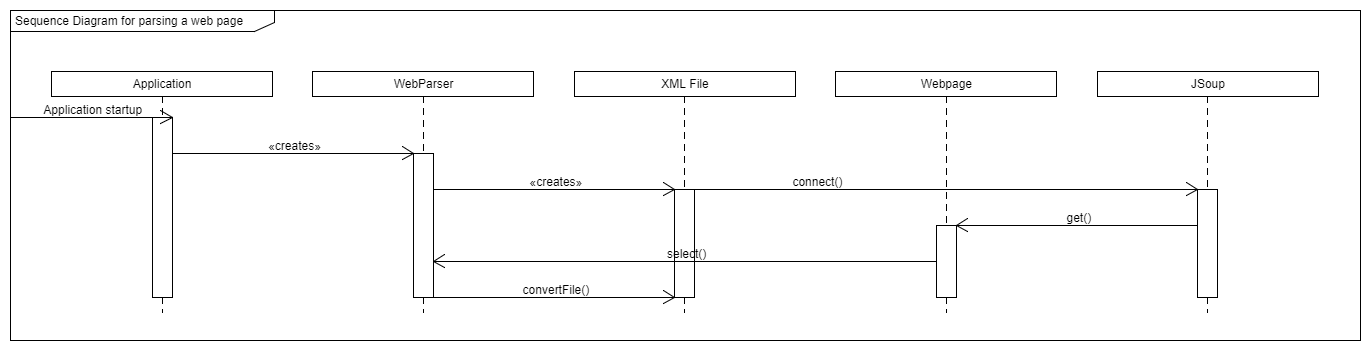




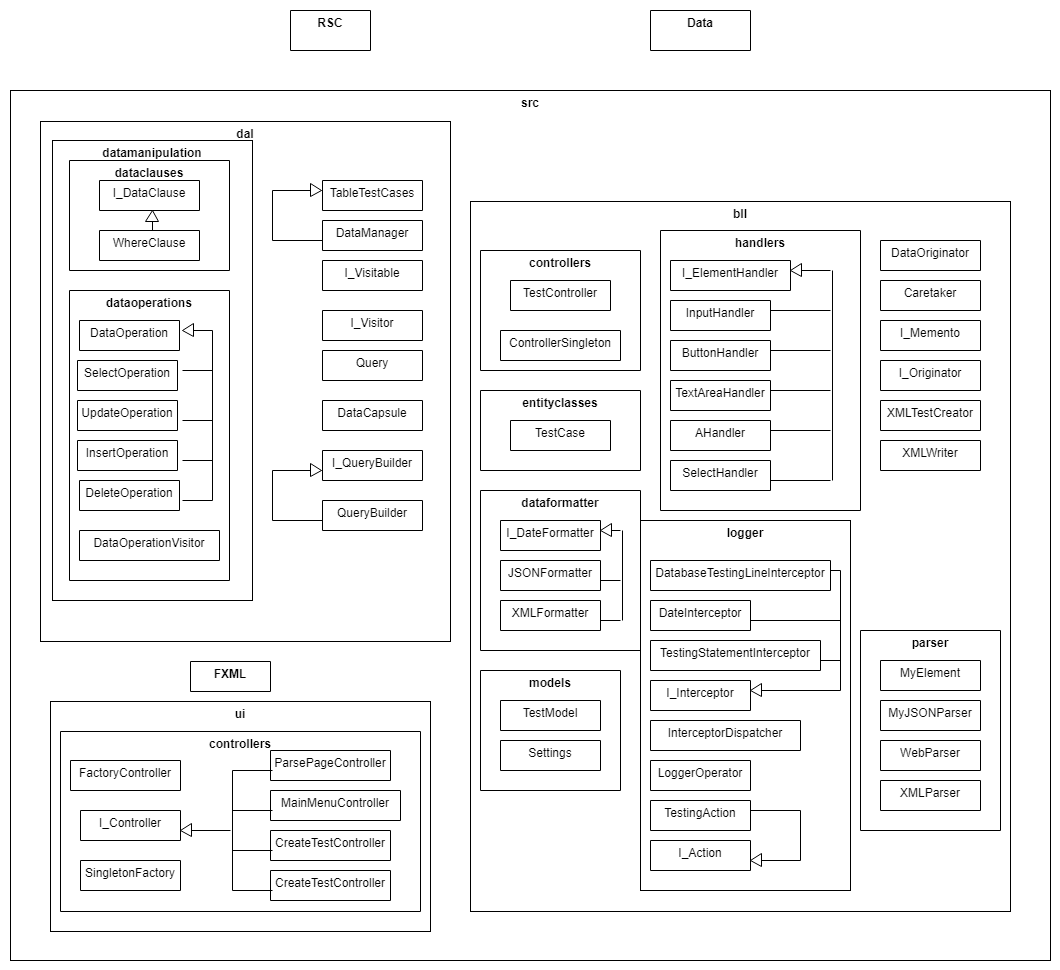


**Runtime Diagram : Sequence Diagram**

Scenario is parsing a webpage into XML



**Architecture Diagram**



**7. Demonstration of Quality Attributes**

**Portability**

We supported portability through our choice of programming language java. Java is a highly portable language, as it can run on any platform. Additionally, all of our team members are proficient in java so this also contributed to our decision to use java.

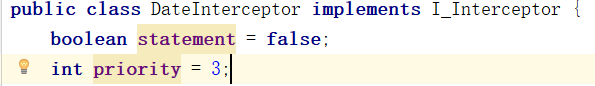
**Extensibility**

Our extensive use of design pattern creates lots of opportunities for extensions in our application. One such opportunity is creating JSON tests. When parsing a webpage, the element can be saved in XML or JSON. At the moment, tests are automatically created in XML format. However, to create tests in JSON, only an additional class – a JSON parser equivalent to our XML parser class – is necessary, along with an option create the test in JSON or XML on the create test menu. This option could easily be implemented using the strategy design pattern.

**8. Documentation of Added Value**

**Interceptor Priority**

In our project, we implemented the interceptor design pattern in a logger operation. We used priority strategy as a callback strategy. We designed our project so that each interceptor has its own priority, and the dispatcher will always invoke the interceptor who has the highest priority currently in the list. For example, the priority number of *DateInterceptor* is 3, if *DateInterceptor* has the max priority in the list, it will be called back firstly and then be removed from the list after the callback operation.

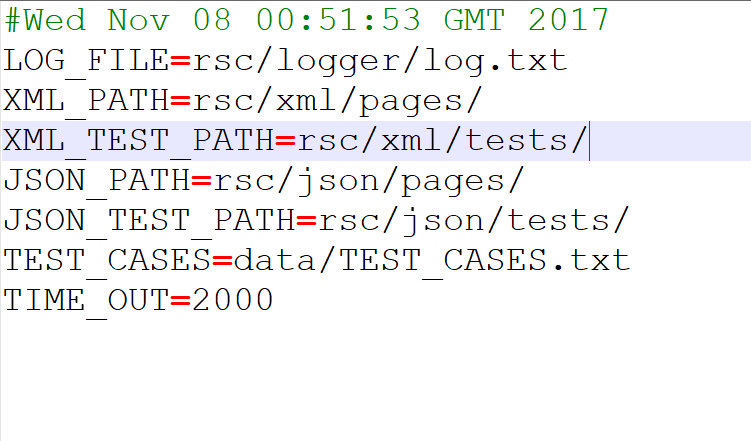


**Selenium**

A web automation tool with a rich API that is used to automate and control web browsers. It has many different bindings in other programming languages such as Java, C#, Perl, Python, Ruby, and PHP. Selenium works by mimicking a user that navigates a webpage by select different element types based on given attribute. It can simulate button clicks, enter input, and select dropdowns and much more. This is a very popular tool used for web crawling and widely used in the industry for testing web applications. To use selenium, it needs to make use of a web driver. That’s is passed to it, there are different types of drivers for different web browsers but the most supported is the geckodriver that used by FireFox which is an executable that can be used from any programming language.

**Properties**

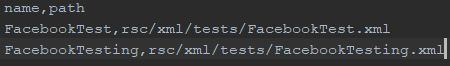
The properties file main purpose was used to read and write to different file paths read in from dataConfig.properties which gives the program more flexibility rather than hardcoding file paths and as well changing speed of selenium navigation time between elements. This is much better for configuration management and avoids recompilation of the whole system which saves time and money.

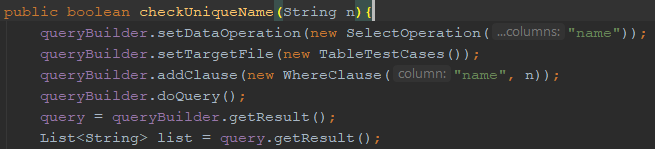


This is done using key value pairs and changing the values will not affect the system code. The above example shows the file paths that can be changed very easily.

**QueryBuilder**

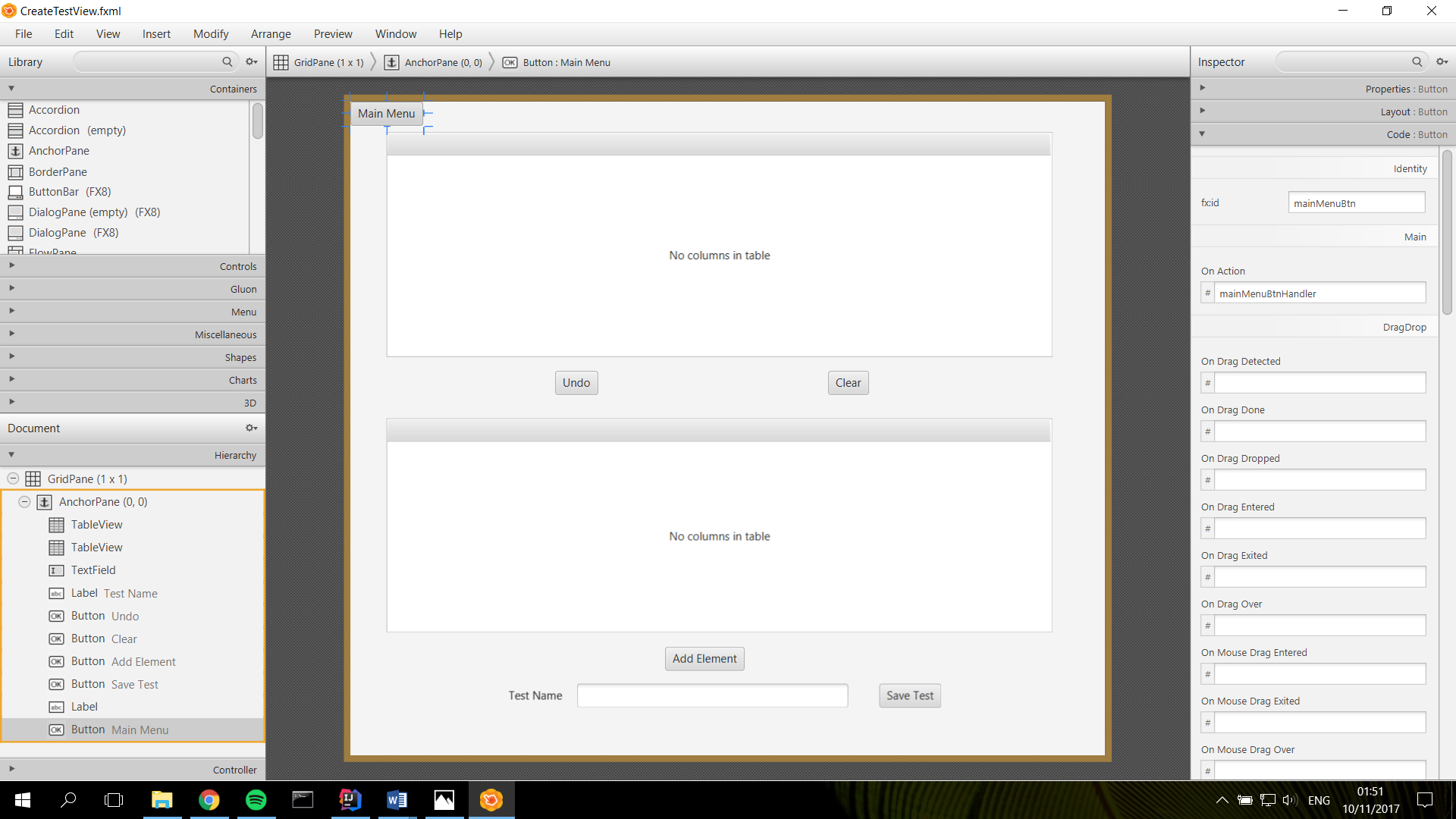
As discussed earlier in this report, QueryBuilder is an attempt to model CSV files as if they were tables in an SQL database. This allows the user to perform select, update, insert and delete operations on it and passing in where clauses when needed. Its creation was motivated from a project completed last semester that also made use of CSV files. Information was pulled from a single class and team members created members to pull very specific information, which became messy, convoluted and posed a security concern. Thus, attempting to model an industry standard in its most basic form felt like a fitting solution to this problem. Below is the format of the text file required and a *SelectOperation* being performed on this file.





**JavaFX and Scene Builder**

JavaFX is Oracle’s latest set of tools for developing GUIs. We used it in our project for two reasons; ease of use thanks to Scene Builder, and to gain experience with newer technologies instead of working with old, soon-to-be deprecated JPanels. It allows for drag and drop capabilities of different widgets and editing different parameters all from the same window. This information is then stored on an FXML file. Shown below is Scene Builder in action on CreateTestView.fxml. Along the left-hand side are available widgets, while the right-hand side contains information about the selected widget, such as ID and handler.



Controllers use the @FXML annotation for objects to create the link between the controller and the view. The first line of code shown below is the FXML for the “Main Menu” button on the upper left-hand corner of the window above. The next line is the corresponding controller calling that instance of that button.



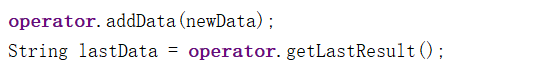
**Git**

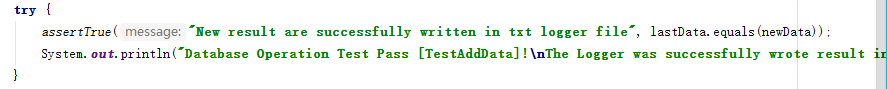
Git was our version control system of choice for this project. While there are many platforms that utilize git i.e. BitBucket, GitLab, we went with GitHub since it what we were most familiar with. Using version control allowed us to seamlessly make changes to code without having to worry too much about breaking or accidentally removing work by other team members. Git also served as backup of our work, so that if anything became corrupted locally, we could simply clone from the remote repository.

**9. Evidence of Testing**

We implemented a Test Primer which runs two test suites.The fiirst one is DatabaseOperationTest: which is JUnit testing for TestAddData, Second one is InterceptorDispatcherTest which is also a JUnit test for TestInvoke and TestLoggerResultForm. These tests told us whether our database access method (text file operation) was working as we had intended and whether dispatcher was working as we designed it to.

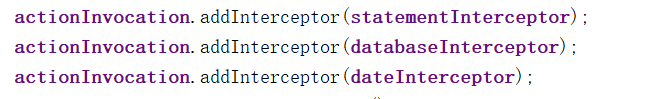
For DatabaseOperationTest: Testing of AddData, addData is a text file operation method, which will get a new String type of testing result and write it into text file. In this JUnit test, the test will pass once the new line is successfully added into the text file and is found to be the last line.



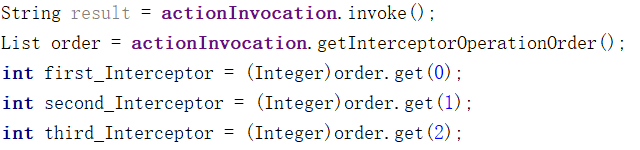


For *InterceptorDispatcherTest*: *TestInvoke* and *TestLoggerResultForm*, both of these methods are JUnit tests for the Dispatcher.

*TestInvoke* will check whether dispatcher calls back interceptors by priority strategy. As per our project design, each interceptor has its own priority, and the dispatcher will always invoke the interceptor who has the highest priority number. Inorder to test this, the interceptors will be registered in the dispatcher in a random ordering.



And it will check the callback ordering of three interceptors and theirs own priority.

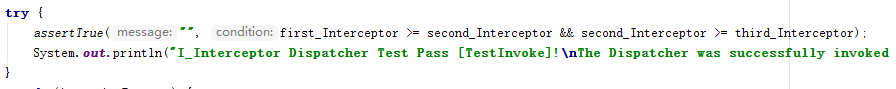


This test will pass if the callback ordering is correct. In this case, the correct result order is:

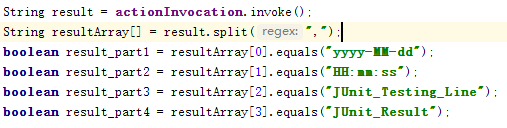
1. *DateInterceptor*

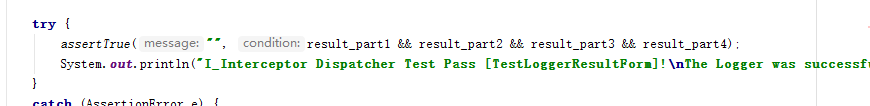
2. *DatabaseInterceptor*

3. *StatementInterceptor*.

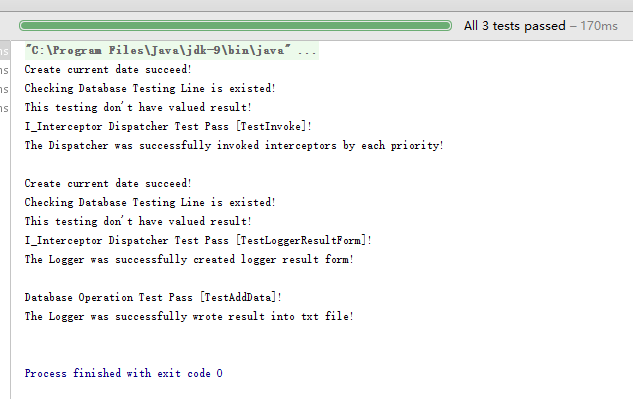


The *TestLoggerResultForm* test is designed to check whether dispatcher will return a right result to be written to log.txt. The JUnit testing will pass once the created form is “Date” + “Testing line” + “Testing Result”.





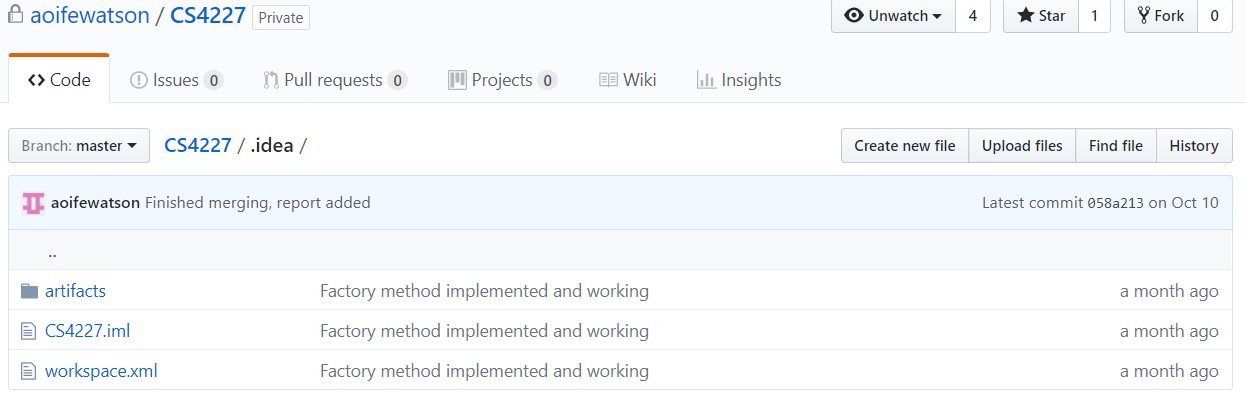
The test result as picture shows below. All 3 tests passed.



**10. Discussion of Problems Encountered**

**.gitignore**

In the early stages of the project we experienced issues with Git. Code was failing to compile due to configuration issues within the IDE we were using, IntelliJ. The root of these issues was the misconfiguration of gitignore, which contains a list of files and/or file types to be ignore by Git. These include configuration files specific to each machine. Displayed below is an early repository with workspace.xml included in Git.



The issue was resolved by creating a new repository for the project with a .gitignore file that had been created specifically for IntelliJ.

**11. Evaluation of Support for Non-Functional Requirements**

Parsing a Web Page and writing out to different file formats was important. This has been achieved using the strategy design pattern which gives the uses the ability to add in more file output without modifying the code extensively that supports extensibility.

Creating test cases, this supports lots of flexibility by using the Memento pattern where a simple undo can revert changes to a previous state give the user a much better user experience and reusability where can load other test cases.

Running tests and mentioned about under the command design pattern already. This is very easy to extend and add extensibility with little or no modification. Where uses can add in new element types and create a handler for that type of element and perform different commands. This also makes use of the QueryBuilder which is efficient as it performs commands in advance before they run.

Logging messages use the interceptor and can easily add new interceptors without much change to the code and extensibility if new interceptors needed to be added. This is important for readability for the log files.

**12. References**

Figure 1 - Adapted from <https://code.tutsplus.com/tutorials/mvc-for-noobs--net-10488>

Selenium - <http://www.seleniumhq.org/>

JavaFX - <https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>

**13. Contributions**

