**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 *createControllers* method.

The command pattern was also used to encapsulate test commands. Each type of element requires a different testing mechanism. 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 the addition of one line of code.

**Factory Design Pattern**

We used the Factory Method Design pattern to create the UI controllers in our application. We did this to capture controller creation in one class. This was used with the command pattern as previously discussed. This has the additional benefit of ensuring the controllers are only created once, they are instantiated when they are added to a data structure as per the command design pattern and called from there.

**Architectural Use Cases:**

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.

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.

Running a test

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

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, and the test name. For tests containing multiple elements, a success or fail note is included for each element.

**Tactics to support architectural use cases (quality attributes)**

**Portability**

We supported portability through our choice of programming language java. Java is a highly portable language, as it can run on practically 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, 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.

**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.

**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.

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 necessitating recompilation.



