**Familiarity Review**

**Name:** Jeremy James Alkire

**Date:** March 9th, 2020

**Week:** 10

**Coding Topic:** JUnit

**Description of Understanding:** JUnit is a testing framework for the Java programming language. It promotes the idea of testing before coding: Ideally to encourage a pattern of testing, followed by coding, and so forth. Continual testing throughout the development process can help ensure stability of the program, and help prevent the programmer from creating a pile of errors that they only discover after a rigorous few hours of programming (without testing). It saves time and resources, both. JUnit is linked as a jar file, and resides under package org.junit for versions 4 and later.

**Teaching Video:**

**Starting at:**

**Also Integrated with:** JSON & QCJSON

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Sandbox or Your code?** |
| JJUTest.java |  | This class is contained in the src/test/java/cit360 folder and uses @Test annotations to indicate what parts of it are to be tested with JUnit. Running this class as a JUnit test will test it, although it currently isn’t running any assertions, so it’s merely testing if it runs successfully. | My code. |
| QCTest.java |  | This class is contained in the src/test/java/cit360 folder and uses @Test annotations to indicate which parts of it are to be tested with JUnit. On the contrary to the above example, this one runs a single assertion method: assertEquals(). This method compares two strings and determines whether they are equal to each other. If they are, the test succeeds, if not, it fails. It uses this method to compare the ideal JSON string (hypothetically) to the actual JSON string generated from a Java object and written to a file by Converter.java. | My code. |
| AssertionsExample.java |  | This class is also contained in the src/test/java/cit360 folder. This is where the bulk of the real demonstrative work is done.  Lines 35-41 are set aside for tests that will pass. These tests utilize the following assert methods:   * assertTrue() * assertFalse() * assertEquals() * assertArrayEquals() * assertNotSame() * assertNotNull() * assertSame()   Lines 43-45 are set aside for the tests that will fail. These tests utilize the following assert methods:   * assertArrayEquals() * assertNotSame() * assertNull()   I chose to deliberately write tests that I know will pass/fail to demonstrate my understanding of the assert methods and how they work, as notated in the program’s comments.  The program will stop execution upon reaching a failure, so to test the last two assert methods (that are intended to fail) you’ll have to comment out the failures previous to it.  Just for good measure, line 58 has an assertTrue() method, and lines 76-80 use either assertNotSame() or assertNotNull() methods. |  |

**Coding Topic:** System Level Tests

**Description of Understanding:** System level tests test each component of a program (ensuring its functionality as a whole) rather than testing a single part. Complex systems that consist of multiple components such as a database, web application, web server, and more are tested individually to verify their functionality. A system level test must conduct multiple tests in order to verify complete system functionality. A system level test should be able to be executed by someone who knows little to nothing about the inner workings of the system. It typically runs through a script, and the tester follows up upon running this script by documenting the results of the test in a spreadsheet.

**Teaching Video:**

**Starting at:**

**Also Integrated with:** Hibernate, Servlets, JSP, Collections

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Sandbox or Your code?** |
| SystemTest.jsp |  | This is the same JSP servlet file used in one of my earlier examples, but I integrated the capability for system testing into this by adding a series of commands to my application controller’s map. This allows the user to specify which tests they want to run. This is where a user should begin their system test.  (Unrelated: It uses the Bootstrap Framework for its frontend as well as some custom CSS and a small custom JavaScript file to implement audio.)  This file was previously called Index.jsp and has been renamed to be more axiomatic.   In order to run the test, you must run the JSP file on a server (such as Tomcat). From here, it will load the home page of the testing application. You’ll see directions on this page. It prompts you to type either “Input” or “Hibernate” into the navigation bar’s text field. Upon typing one of the two, hit the “Submit” button.   From here, the JSP servlet sends this request to the application controller, which checks a map to determine which course of action to take. The key “Hibernate” maps to the value “HibernateTest.jsp” which should redirect the user to HibernateTest.jsp in their browser.  The key “Input” maps to the value “Input.jsp” which should redirect the user to Input.jsp in their browser. | My code. |
| HibernateTest.jsp |  | Upon loading the HibernateTest.jsp page, the functionality of Hibernate, GetAllData.java, and the Application Controller are tested.  The user is met with the results of this test on the HibernateTest.jsp page. It should state that the Application Controller is functioning as intended and ask them to check below for the results of a database query made by Hibernate.  (The file itself makes a call to main() from GetAllData.java, which is a part of the hibernate.utils package. This uses Hibernate to query the SQL database for the contents of the tables contained in the hibernate5 database.)  If the query is displayed successfully to the HibernateTest.jsp webpage, the tester knows the following:   * The application is successfully utilizing Hibernate. * Hibernate has queried the database. * The Java application “GetAllData.java” is functioning as intended.   The webpage itself should state this, which should ensure that ***any*** user (even a user without technical knowledge) can determine whether the test has passed or failed. | My code. |
| Input.jsp |  | If the user types “Input” into the text field in the navigation bar they are taken to the Input.jsp page.  Upon loading this page, the user is met with two text fields. They’re prompted to enter a number into each. Upon doing this and hitting submit, they are redirected to “Logic.jsp” which displays the sum of the two values they entered the text fields. This tests the functionality of the “Addition.java” file located in the project’s “systemtest” package.  The user is met with a message that explains this to them, that way they’re able to determine if the application passed the test. |  |
| Error.jsp |  | If the user enters invalid input into the navigation bars text field at any point during the test, they’re redirected to Error.jsp, which states that they’ve provided invalid input, and then offers them examples of valid entries.  From here, they can continue the test after entering a valid command.  Alternatively, they can return to the Home page (SystemTest.jsp) by typing “Home” in the text field. |  |
| Addition.java |  | This is one of two classes being tested by the SystemTest. It simply adds two numbers provided to it by the Input.jsp page.  It does not send its results anywhere on its own. |  |
| Logic.jsp |  | This file creates an instance of the Addition class, passes the parameters entered into the Input.jsp page’s text fields to the setAddend1() and setAddend2() methods as arguments, and displays the result of the add() method to the tester.  From here, it explains that if the user sees the sum of the two values they entered on the Input.jsp page, the test has passed. |  |
| Systems Level Test (Documentation).xlsx |  | This is my systems level test excel spreadsheet. | My spreadsheet. |

**Diagram**: Use Case Diagram

**Description of Understanding:** Use case diagrams are used to display the requirements of a system. It is a very high-level representation of the system that doesn’t portray many specific details at all. The components of a use case diagram are actors, associations, system boundaries, and use cases. Actors are individuals who interact with a use case. They are named by nouns, and they trigger the use cases. They often provide input and expect output. Use cases are system functions, either automated or manual, and are named by verbs. Actors are linked to use cases, though not all use cases are linked to actors. Communication / associative links connect actors to use cases, to represent their relationship. System boundaries may outline the entire system, or simple modules. Actors lie outside of the system boundary, and use cases lie within it.

**Teaching Video:** None

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Example**  **Or Your code?** |
|  |  | In this use case diagram, there are three actors:   * The user. * The server. * The site owner.   There are 10 use cases displayed in the diagram.  There is an ***include*** relationship between Email Verification and Create Account because the application will verify a user’s email when they create an account, or at the very least request that they do.  There is an ***include*** relationship between Login and Login Authentication because when the user attempts to log in, the server tries to authenticate their login, which is an example of one actor’s use case using another’s.  There is an ***extend*** relationship between Edit Status and Set Status because upon a user setting their status, they *might* opt to edit an existing one, though not always.  There is an ***extend*** relationship between Browse Content and View Advertisement because a user *might* end up viewing an advertisement while browsing site content.  There is an ***include*** relationship between Add Advertisement and View Advertisement as this is a case in which one actor’s use case uses another. (On second thought, this might be a bad example, since viewing an advertisement doesn’t add one. But viewing one is dependent upon one being added.)  I considered adding ***extend*** relationships between Login, Delete Account, Set Status, Edit Status, and Browse Content, as those use cases are contingent upon a user having logged in, but unless I’m misunderstanding the meaning of extension, it isn’t indicative of chronological dependency, but rather an instance in which one case may use another, which is why I did not add ***extend*** relationships between these. I’m not certain that someone could rightfully say that setting a status or deleting account **uses** “logging in”, even if it depends on it having been completed. | This use case diagram is based upon a hypothetical social media website. |

**Coding Topic:** MVC

**Description of Understanding:** The MVC is a three-part architectural design pattern used for developing user interfaces. The three components are the model, view, and controller. The model manages the data and logic of the application. The view displays a visual representation of the data. It handles both input and output and is what the user interacts with. The model and view components do not communicate with each other directly, they are both managed by the controller, which acts as a medium between the two. The controller is essentially the engine of the program. It calls methods from the view and the model and passes data between them as necessary.

**Teaching Video:**  None.

**Starting at:** N/A

**Also Integrated with:** Use Case Diagrams, Use Case Documents, Sequence Diagrams

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Sandbox or Your code?** |
| RequestData.java |  | This class operates as the **view** within the MVC architecture. The method *requestAthleteData()* outputs a prompt to the user that requests that they input data into the console. That data is handled and stored in method variables and returned to the method caller. It functions as a text-based UI. By itself, it does nothing. The controller calls its functions. | My code. |
| Athlete.java |  | This class operates as the **model** within the MVC architecture. It specifies the attributes of the Athlete object. It has private non-static variables, public getters & setters, and a universal configuration method *configureAthlete()* that allows you to set the value of all variables with a single function call (and three arguments.) | My code. |
| Engine.java |  | This class operates as the **controller** within the MVC architecture. It creates an object instance of the Athlete (model) class, calls *requestAthleteData()* from the RequestData (view) class and stores its output in a string variable. It then splits the string into a list of strings by comma (,) characters. From there, it calls *configureAthlete()* from the model, and passes each element in the list to model’s method via its parameters. | My code. |

**Coding Topic:** Java Collections

**Description of Understanding:** A collection, sometimes called a container, is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. These typically represent items that form a natural group, such as telephone directories. A collections framework is a unified architecture for manipulating collections. Collection frameworks contain interfaces, implementations, and algorithms.  
  
Interfaces are abstract data types that represent collections. These allow collections to be manipulated independently of the details of their representation, and generally form a hierarchy. Nearly all interfaces inherit methods from the Collection superclass and pass these methods down to their implementations. Core interfaces in Java include: Collection, Set, List, Queue, Deque, Map, Sorted Set, and Sorted Map.

Implementations are concrete implementations of the collection interfaces. They are reusable data structures, and children of the interfaces, whom are their parents, and whom they inherit methods from. General-purpose implementations in Java include: HashSet, HashMap, ArrayList, ArrayDeque, TreeSet, TreeMap, LinkedList, LinkedHashSet, and LinkedHashMap.  
  
Algorithms are methods that perform useful computations, such as searching, sorting, inserting, and deleting elements inside of a collection. They’re described as polymorphic, meaning they’re able to be used in many different implementations. These are typically implemented via methods, such as add() and put().

**Teaching Video:**

**Starting at:**

**Also Integrated with:** ArrayList is used briefly in my controller class (Engine.java) in my MVC coding examples.

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Sandbox or Your code?** |
| ArrayListExample.java |  | This class utilizes the ArrayList implementation of the List interface. | My code. |
| CollectionExample.java |  | This class also utilizes the ArrayList implementation of the List interface. | My code. |
| HashMapExample.java |  | This class utilizes the HashMap implementation of the Map interface. | My code. |
| LinkedListExample.java |  | This class utilizes the LinkedList implementation of the List and Queue interfaces. | My code. |
| TreeMapExample.java |  | This class utilizes the TreeMap implementation of the Map interface. | My code. |
| HashSetExample.java |  | This class utilizes the HashSet implementation of the Set interface. | My code. |

**Diagram**: Sequence Diagram

**Description of Understanding:** A sequence diagram demonstrates the interaction between messages and objects within a system. It displays these relationships in a sequential manner, so that people can clearly see how messages flow throughout the system. Components of a sequence diagram include objects, messages, and sequence numbers, which indicate how methods are called one after another.

**Teaching Video:** None

|  |  |  |
| --- | --- | --- |
| **File** | **What should I be looking for?** | **Example**  **Or Your code?** |
|  | The link on the left links directly to an image of my sequence diagram.  This diagram sequentially shows how data flows through the program by outlining what methods are called and what data is passed to them as arguments. It also displays what data is returned by methods (such as getters) and attempts to showcase this flow of information chronologically. | My code. The same code used to demonstrate the MVC architectural pattern. |

**Diagram**: Use Case Document

**Description of Understanding:** A use case document offers a story of how an actor utilizes a system (via use cases) to achieve its goals. It provides a detailed step by step description of how the actor will use the system to achieve its desired outcome. Unlike a use case diagram, it provides a written representation of this information, rather than illustrated. It is typically in table format and is composed of a general description of the use case, its name, detail about the author and date in which this information was documented, a list of actors, preconditions, postconditions, an explanation of flow, alternative flows, exceptions, and requirements.

**Teaching Video:** None

|  |  |  |
| --- | --- | --- |
| **File** | **What should I be looking for?** | **Example**  **Or Your code?** |
|  | The link on the left links directly to a .doc file, which is my use case document for the program I used in my MVC example.  The document specifies the following:   * Actors. * Three preconditions. * Three postconditions. * Flow. * Alternative flows. * Exceptions. * Requirements. | My code. The same code used to demonstrate the MVC architectural pattern.  The main method of the program being documented is in the Engine.java class. |

**Diagram**: State Diagram

**Description of Understanding:** State diagrams represent the current condition of a system at a specific point in time. Much as its name implies, it is used to outline the dynamics of a system’s ability to change state. State diagrams are sometimes called state machine diagrams and are also known as state chart diagrams. State diagrams model the reaction of objects and classes to stimulation.

**Teaching Video:** None

|  |  |  |
| --- | --- | --- |
| **File** | **What should I be looking for?** | **Example**  **Or Your code?** |
|  | This state diagram is extremely simple, predominantly because it showcases the only semblance of state that the PlaySong.java class has.  The PlaySong.java class creates instances of the Audio.java class, which has a static AtomicBoolean variable called ***running*** that it uses to keep track of thread state.  This diagram outlines the changes in state that the Audio objects in the PlaySong class undergo in accordance with the status of the ***running*** variable, and the methods that control its state. By extension, this means the diagram displays the state changes of the Audio threads, and therefore the program (as that is the purpose of the class). | My code. This diagram showcases the PlaySong.java class which utilizes thread technology. |