COS 301 Testing Policy Document

Team Recursive Recursion

April 12, 2018

1 Definitions

1.1 Testing as a whole

Testing refers to the act of making sure software is reliable, bug resistant and satisfies the defined criteria of the component under scrutiny.

2 System Structure and what to test for

2.1 Repository testing

A repository refers to a decision based system that decides how to interact with models by invoking either a data access object system or requesting another system to provide it with the requested data.

2.1.1 Argument testing

Components that take input needs to test the input provided, If the input does not fit in the argument domain an argument exception should be present. Invalid arguments should be tested for in a Theory approach providing a list of test criteria and iterating through them.

2.1.2 Result testing

Result testing implies a correct argument domain. The resulting artefact of system should be tested against an expected artefact. All the properties individually should be tested against the expected values if the object as a whole cannot be compared in an extensive manner.

2.2 Controller Testing

Controller testing refers to testing on a system that can do things with a data model. This include updating and inserting items to name a few functions.

2.2.1 Relational testing

Tests should always exist if an object is created that is referenced by another object within a model. This should happen in a bi-directional manner to ensure referential integrity.

2.2.2 Object integrity testing

Tests should be in place to verify the integrity of an object before and after a change have been made on a persistence storage system to verify that the object have indeed been correctly stored.

2.3 View Testing

Views should be able to handle any length and state of the system, thus size and quantity of objects that the view can accurately display without ambiguity should be tested.

2.3.1 Error handling

Any error or exception that the controller can return should be tested for. This is not limited to status errors and should be descriptive. View testing can only be complete if all possible end states of the system in handled.

3 Test implementation structure

Tests should follow a structured implementation.

3.1 Assign

In this section all possible variables that will be needed to test the component in question should be created. Test separation is of importance in the sense that the objects created in this phase should react as expected. If the object

that is created in this section can be tested, it should be in a distinct test case as only one component should be tested per test.

3.2 Action

In this fragment of a test, the object under scrutiny should be created. The creation process should then return an object that will be tested against expected values. This includes storing exceptions as all tests should pass to view code as working.

3.3 Assert

Assertion should be set up in a fashion to extensively evaluate the artefact created by an action. The artefact can be an error and in that case the specific error should be asserted with the expected parameter. This reiterates that each test only tests one aspect of a object created through an action.

4 Test Procedure

Following a Test Driven Development methodology improves the effectiveness of the coding process. It can eliminate bugs at an early stage of development and can guide the development process.

4.1 Testing

1. Produce a skeleton of the class you want to implement. Each function has an empty body or returns a default value such as an empty string or null. As development proceeds the skeleton is expanded into the end product.

2. Select a feature you want to implement. Usually features of high priority are implemented first. Which feature to implement next is also dependent on the flow of data and processing. If one function (say f1) uses results of another function (say f2) then f2 needs to be implemented before f1. The same logic is followed if f1 makes a call to f2.

3. Create tests for the feature. Remember to cover all branches. If a feature branches based on some criteria into an if-else or a number of if-else blocks then tests should cover all the branches.
   Implement and test the feature. Edit the code and the tests according to need until the feature passes all the tests while complying to all testing standards.

4. Repeat this process until all features are implemented.

5. Make sure that the tests cover the whole class.