ERIC WEBB

Nova Southeastern University

College of Engineering and Computing

Fall 2019 - Master Level Course

CISC 680 - Software Engineering - CRN – 21741

Term Code: Fall 2019 (202020) Course

Assignment 4: (Question Set 2) - See Syllabus for assignment % and Due

1. What are the attributes of a good software test?

Software tests should be able to communicate clearly their objectives and expected results. There should not be any argument or doubt on whether it is an effective test or what the results should be. The intent of a software test should be direct and easy to understand in such a way that a non-technical person can comprehend what the objective and tasks at hand are. A test should be significant that one can understand and grasp the importance of the test and its results. A test could be clear on the results, but one may not understand why these results are important. A good test should be isolated in the fact that it should not be dependent on other tests. If it depends on other tests then results could be skewed depending on the results of each individual test. Test should be automated so that excessive time is not spent creating and running a test. These automated test should also be quick to write and quick to run. These test should be unique in such a way that the results provide confidence that is not offered from other tests. A good software test typically only invokes a portion of code and does not use the code in its entirety. The test will usually deal with a certain portion of code like a single method or a particular piece of business logic. A good test should be order independent so it can be ran regardless of order of other tests, giving the tester a choice on which test to choose. It is no secret that a good test should be re-creatable and be able to be mimicked. It is said that a good test should be thought of in a manner similar to an end user. Thinking outside the box and asking questions are good attributes of a software test. Asking questions like why and why not something occurred or how something occurred can lead to better understanding of the applications functionality. Test results should be easily displayed and readable to the human eye.

2. Describe three control structure testing strategies.

Three examples of control structure testing are Branch testing, Condition Testing, and Data Flow testing. To begin, branch testing is sometimes synonymous with the term “decision testing.” Branch testing is based off of obvious decision statements such as for, while, if, etc. These branch tests can also include more subtle decision statements such as a Boolean, try-catch, and ternary expressions. Essentially they use decisions statement in the program. For these branch tests each condition needs to be executed at least once. An example of this could be explained in pseudo code as follows, if dog is an instance of animal then test bark, else leave dog house. This is an example of a branch test. Next we have loop testing, loops are an integral part of most programs and need to be tested accordingly. There are four types of loops that need to be tested; simple, nested, concatenated, and unstructured. A loop can cause many instances to occur so one needs to make sure that each instance created is properly tested. One popular testing method I use when testing a program is an on change condition. For example, if a loop creates fifteen instances of a dog and the last instance changes to be a cat it alerts me as a false. It is good to test simple loops by testing a few scenarios such as skipping the loop, the n-1 equivalent, and the 1+n equivalent. You should test nested loops by starting in the most inner loop and setting all other passes to their minimum values and then working from the inner loop outwards increasing passes till all loops are tested. If you are testing concatenated loops test them as simple loops if they are independent of each other, if they are dependent of each other test them as you would a nested loop. Lastly when it comes to unstructured loops it is best to redesign the looping structure or the algorithm. When it comes to basis path testing you will create a test case scenario for every method , statement , or function in your design at least once. This can be done in a flow graph notation such as basically a top down approach to testing, Next you can use a flow graph notion that breaks the testing down into regions where nodes and edges meet. An edge being the control flow itself, a node being the procedural function. A node and edge have to bind and meet, this becomes a region. This way you can have different parts of you code tested that are similar to other parts of your code deriving away from the top down fashion of a flowchart. When testing independent program paths it is important to calculate the cyclomatic complexity of how many paths in your program to test for. One more useful tool for basis path testing is a graph matricie. These are a good rendition of nodes line weight along with the likely hood of how often they might occur.

3. Why is regression testing an important part of any integration testing procedure?

When testing code it is imperative to implement a form of regression testing. This is a part of any integration testing for good reason. Integration testing aims to test individual parts of code but as a whole. While testing components of code individually you might not get the same results as if you tested them all together. This can test from the main program down toward the more atomic pieces of code called top down integration, also it can be tested from the smallest units up towards the main program called bottoms-up integration. When it comes to Objected Oriented Programing this becomes more difficult since code is not ran in hierarchal fashion but can be resolved using methodologies such as thread and use-,based testing. This helps verify the program works as a whole instead of verifying small individual parts to make up as the whole. Every time a new piece of code is added, new input output paths are created establishing new data flows. You must go back in your code and verify that your new changes not only work, but they do not break any existing logic or functionality. This is why regression testing is so crucial to integration testing because it verifies that new code has not broken the old code. This can be tested in a variety of ways within the regression test suite. For example, a test suite can use a representative test to verify all the programs functionality. They can perform additional tests on code that is more likely to be changed in a more specific format. From that they can also perform tests on components that have indeed been changed. If it were not for regression testing new functional code might be added but the now broken older functionality would not be discovered till it was too late. Imagine adding a nice new user interface to a banking system, but breaking the older functionality of allowing users to withdraw funds. This would cost the banking company a lot of grief with complaints and maybe even law suits. From this example you could see how regression testing is not something to be overlooked and sometimes you must go back and test very basic functionality of code no matter how simple or small. This is why regression testing is crucial to any integration testing.

4. What are the key differences between validation testing goals and acceptance testing goals?

When it comes to validation testing and acceptance testing sometimes they get referenced in similar fashions, when in fact they have very different goals. When it comes to validation testing the goal is to make sure that the user can not break the program from their input. If a user enters an input that does not make sense it can cause faulty functionality. For instance, if you are putting in input for a numerical value you want to make sure that there are no other characters besides numbers such as an improper input of a letter or symbol. Also you would not want a number to be below 0 or above a variable finite amount. This could be an example of validation testing. If validation testing was not present users could stumble upon inputs to code that could cause potential damage. That is why it is imperative to test as many inputs as possible to guard against this type of manipulation. A popular way mitigate this is by using test driven development. You can write many correct and incorrect inputs to be tested in your validation at once instead of testing each one individually at a time. If all your tests handle the correct inputs and block your incorrect inputs then your validation testing is complete. Although validation testing can become pretty extensive because users simply have so many options to enter information, software engineers must think ahead to protect their code from all these options. That is the importance of validation testing. When it comes to acceptance testing, you are not testing user input but rather if the code is right for the user at all! You may have designed a perfect form with validation for all inputs, but if the form doesn’t send an email to its intended recipient but just goes to the next page, then it has failed its user acceptance test. Acceptance testing aims to provide whether the functionality is working as intended, or is it doing something completely at random? Another feature of acceptance testing is how difficult it is for the user to use it. Does it work as smoothly as intended? Acceptance testing is not user to protect code from the user, but to make sure the user is protected from having improper, defective, unplanned code.

5. Describe how test cases are derived from behavior models to facilitate interclass testing?

Since object oriented programs are not meant to be tested in a traditional hierarchal fashion, analyzing behavioral models and testing their transitions is recommended for interclass testing. This is done by looking at the transitions between classes. It is recommended to do this in a breadth-first fashion. This allows you to test transitions consisting of only transitions that were previously used. For example, if you were to create a banking application with the following three behaviors: login, account, and logout. In a breadth-first fashion you would not be able to test account or logout without first using the login. This follows the logical pattern that you must first login to be able to test your account and you obviously must first be logged in to be able to test logging out. This is an example of how the breadth-first test transitions that only contain transitions previously used. Without this previously used behavior might cause improper testing results if tested together. Although a test might work correctly in one scenario with some behaviors it might not work the same with other behaviors. This is especially true with behavior that it is not related to it as well. It also might not work properly without any previous related behavior first initiated. Starting from the breadth and working you way outwards in the code using only the same transitions allows for a more accurate representation of your application rather than testing it with unrelated behavior.

6. List the components of a formal specification language and describe their roles.

When it comes to developing software it can be very helpful to define and model logic in a formal specification language. This could include formal specification languages such as LARCH, Object Constraint Language, and the Vienna Development method. The point of these formal specification languages is to help with system analysis, system design, and requirements analysis to better describe the system as a whole. The beauty of formal specification languages are they allow you to design the system at a much higher level that is then replicable via a traditional programming language. Formal specification languages are typically comprised of three main components. The first being the syntax of the specification language itself. If the formal specification language does not have proper syntax in place then there is no set definition on how the read, write, or interpret the language. This could cause confusion and arguments of interpretation among developers. The second component of a formal specification language is the semantic domain. The semantic domain are the objects within the system and allows for the formal specification language to define a universe of objects that can be used to describe the system. Without the semantic domain we would not be able to concretely define what the formal specification language it truly defining. Again, this leaves room for error and argument when designing if not applied. The last component in the formal specification language architecture is the relations component. This component is what modifies and manipulates are semantic component with the proper syntax. This is what defines the relations for the data and what we can do with it. Without the relations component we would only have semantic objects but would not provide any concrete functionality to manipulate them. This would intern leave the functionality and business logic of the formal specification language up for grabs. So to recap the three component are essentially they syntax, the objects, and their relations in the form of the syntax, semantic domain, and relations components.

7. Describe the process of writing a formal specification for some system function.

It is important to exercise your ability to write formal specifications for system functions. For this example I will be demonstrating the process of writing a formal specification for a payroll website with a single Manager and multiple Employees as Users. We will add a User to this website. This will be conducted using Z formal specification. To begin let USER be the set of all people: [User]. Next, a USER has a set of Managers and Employees where the Manager is also an Employee. The first process for this Z notation example would go as follows, the Schema name Website will have two declarations, Multiple Employees and one Manger. The invariant states that a Manager is an employee. This is donated in Z notation below. Taking note of the U as a variable indicating multiplicity and the bottom representing that a Manager is a User with the symbol.

|  |
| --- |
| Website |
| Employees : U User Manager : User |  |
| Manager  User |  |

The Next process is adding an Employee to the website. To do this we need to add an Employee to a set of Users. When doing this we need to verify the Manager does not change when a new Employee is added. This is denoted in z notation below. Take note of the  notation to include are original website schema. Also taking not of the  symbol, unioning are Employees with the new Employee that was added.

|  |
| --- |
| addUser |
| Website  New? : User |  |
| Employees’ = Employees  new?  Manager’ = Manager |  |

From the above process we can see how that to make a formal specification. Essentially a schema needs to be defined with a schema name, declarations, and invariants. Following that the schema needs to be manipulated via the declarations and predicates.

8. Technical testing metrics fall into two major categories. What are they?

There are two categories that fall within the technical testing metrics. To begin, one category is estimating the number of techniques needed to be tested. This predicts the likely number of tests that will be required at various testing levels. Sticking to a good estimation of testing requirements and techniques builds good rapport with clients and helps sustain a good reputation in the field. It is difficult to blindly place a finite estimation of how many tests are needed that is why experience is very beneficial because an experienced engineer will have worked on multiple projects and can more accurately estimate the number of the test needed. Another good way to get an estimation of tests would be to locate documents or artifacts online pertaining to your test environment. This could be a good lead to understanding what a good quantity of tests would be. Sticking to an excessive number of tests allows for almost of a forced testing situation where you might be able to test ten things in one scenario but it is required you write 1 test for each one instead. The next technical test metric focuses on the test coverage for a given component. When testing components you must think in terms of OOP. It is important to remember and consider encapsulation and inheritance. When testing individual components you might unlock data flows and I/O paths further downstream in the code. When testing individual components its allows you to think more directly on ways to test, in contrast of a broad pattern of the code as a whole, this allows for more specific test cases in the component. So, essentially the main differences between the two testing metric categories is quality vs quantity, with quantity being the standard estimated number of test cases that should be performed vs the quality of testing that could be done in an individual component.

9. Describe the five activities associated with the software measurement

process.

10.Describe the role of class-oriented metrics in assessing the quality of an

OO system.

11.Why is it important for software developers to make use of measurement

to guide their work?

12.Why is the "make-buy" decision and deciding whether to outsource

software development an important part of the software planning

process?

13.Describe the process of building a risk table.

14.What is forward engineering?

15.What characteristics need to be exhibited by organization to improve its

software process?