**Week 5: Software Testing - Lecture**

|  |  |  |
| --- | --- | --- |
|  |  | [Print This Page](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&) |

**Testing**

[**Terminal Course Objectives**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#tco) **|** [**Introduction**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#intro) **|** [**Testing Objectives**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#bk1) **|** [**Types of Testing**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#bk2) **|** [**Equivalence Classes**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#bk3) **|** [**Unit Testing**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#bk4) **|** [**Integration Testing**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#bk5) **|** [**Test Plan**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#bk6) **|** [**Summary**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#summay) **|** [**References**](http://vizedhtmlcontent.next.ecollege.com/(NEXT(85a440c708))/Main/CourseMode/VizedHtmlView/RenderVizedHtmlView.ed?courseItemSubId=351574353&courseItemType=CourseContentItem&#references)

|  |  |
| --- | --- |
| **Terminal Course Objectives** |  |

|  |  |
| --- | --- |
| **9** | Given an outside client-provided CIS project or approved alternative CIS case project, the statement of work, analysis document, and the detailed design document create the detailed plan for testing the system components of the project and conduct the planned tests to ensure that the project requirements are met. |

|  |  |
| --- | --- |
| **Introduction** |  |

**Testing** is the controlled exercise of a program's code in order to expose errors. A successful test is NOT a test that runs perfectly. As developers, we can all write a test that demonstrates "correct" behavior; but a successful test is one that actually uncovers a defect. Also, in a complex program, it is physically impossible to perform exhaustive testing where every condition is examined; however, we can develop test cases that allow us to infer the general behavior of the complete system based on a subset of conditions. Uncovering these sets of test cases is one of the key objectives of any test plan, and in this lecture, we will look at the concepts that allow your team's project to generate the minimum number of test cases that will provide adequate test coverage. In particular, we will review testing objectives, types of software testing, normal range testing, robustness testing, unit testing, and integration testing.

|  |  |
| --- | --- |
| **Testing Objectives** |  |

We are not developing systems for the sake of technology, nor as a hobby. We are, however, developing systems for end users to accomplish a business need. As a result, the ultimate goal of testing is to ensure that the customer is satisfied Pressman2 suggests the following testing principles:

1. All tests should be traceable to customer requirements. The most severe defects are the ones that affect the end user, so it only makes sense to concentrate on the requirements.
2. Tests should be planned long before testing begins. Test planning should start as soon as the requirements are completed.
3. Pareto principle applies. This implies that 80% of all errors will be traceable to 20% of the modules. The trick is to identify the 20%.
4. Testing should begin in the small and progress towards testing in the large. Initially focus on the individual modules and then move your way up the architecture.
5. Exhaustive testing is not possible. Even in average-sized programs, the permutations are too large to test everything, so focus on finding error equivalence sets.
6. To be most effective, testing should not be done by the developer. Every developer knows how to write a test to show what works, but the goal is to find errors, not show the absence of errors.

As shown by the figure, testing begins at the low-level **unit tests** to ensure that the low-level modules are constructed correctly (verification) and that they satisfy the allocated requirements. Once the team is satisfied that the units are correct, incrementally combine the modules into their higher level components and continue the incremental "build" and "test." As you progress up the hierarchy, you are moving away from verification (building the product correctly) to validation testing (ensuring the requirements are satisfied).

The following sections will address the specific types of test that can be done at each of the levels and strategies in determining test cases.

|  |  |
| --- | --- |
| **Types of Testing** |  |

The goal of testing is to demonstrate that the software satisfies its requirements and to demonstrate with a high degree of confidence that errors, which could lead to unacceptable failure condition, have been removed. The following sections describe three basic areas of testing that apply to testing programs: (1) normal range testing, (2) robustness testing, and (3) stress testing.  These categories of test cases should be applied to unit, or class, testing and integration testing, which are described in the following sections.  The following figure illustrates four basic categories of testing2:

1. **White box,** or **structural testing,** is testing that takes into account the internal mechanism of a component or system.  This includes branch testing, path testing, and statement testing.  It is important to note that a white box test, may or may not be directly traceable to a user requirement.
2. **Black box,** or **requirements-based testing,** is testing that ignores the internal mechanism of a component, or system, and focuses solely on the outputs generated in response to a selected set of inputs and execution conditions.  Black box tests are directly traceable to user and derived requirements.

Normal range testing is where you will test the system under nominal operating conditions with nominal inputs to validate that the software performs its intended function. To meet this objective, a series of normal range test cases should be developed to demonstrate the ability of the software to respond to normal inputs and conditions.

* Real and integer input variables should be exercised using valid equivalence classes and boundary values.
* For time-related functions, such as filters, integrators, and delays, multiple iterations of the code should be performed to check the characteristics of the function.
* For state transitions, test cases should be developed to exercise the transitions possible during normal operation.
* For requirements expressed by logic equations, the normal range test cases should verify the variable usage and boolean operators.

Robustness test cases evaluate what happens when the system is outside the normal operational environment (i.e., what happens when something does go wrong).  These robustness test cases demonstrate the ability of the software to respond to abnormal inputs and conditions. They should include real and integer variables that are exercised using equivalence class selection of invalid values.

* System initialization should be exercised during abnormal conditions.
* The possible failure modes of the incoming data should be determined, especially complex digital data strings from external systems.
* For loops where the loop count is a computed value, test cases should be developed to attempt to compute out-of-range loop count values.
* A check should be made to ensure that the protection mechanism for exceeded frame times responds correctly.
* For time-related functions, such as filters, integrators, and delays, test cases should be developed for arithmetic overflow protection mechanisms.

For state transitions, test cases should be developed to provide transitions that are not allowed by the software requirements.

The following section will discuss the concept of test case equivalence classes, which will help your team determine an adequate set of test cases.

|  |  |
| --- | --- |
| **Equivalence Classes** |  |

If you review the following flow diagram, you will see that the algorithm selects one of the functions depending on where an input falls within a range:

* If the input is between 0-10, execute Function 1.
* If the input is between 11-20, execute Function 2.
* If the input is between 21-30, execute Function 3.

In this case, you can easily see that all the integers between 0-10 are part of one equivalence class, integers between 11-20 are in a second equivalence class, and integers between 21-30 are in a third equivalence class.

As a result, when developing test cases, you do not need to test cases for all items in the equivalent class; you only need to sample the values in the equivalence classes.  However, a judicious choice of values can increase your test coverage.  For example, you will want to select **boundary values**, which are the values on the "edge" of the equivalence classes.  For example, 0 and 10 are on the boundary of the first equivalence class.  The boundary value tests will ensure that the boundary values are in the correct class.  Then, to ensure the normal operation, you should select one or two internal values to gain full confidence that the algorithm is performing as expected.  However, using equivalence classes, there is no need to exhaustively test all of the inputs.

Normal range tests examine the typical operation of the program but normal range tests do not evaluate what the program will do when the input values are illegal (incorrect data type) or if the values are invalid (correct type, but not in the logical range of values).

While there are other types of testing (load/stress tests, time-related tests), your team should develop both types of tests. They are: (1) normal range and (2) robustness for your case project.  These two general categories of tests can be conducted during all levels of testing.  In the next two sections, we will discuss unit testing and integration testing.

|  |  |
| --- | --- |
| **Unit Testing** |  |

A **unit** is a logical grouping of modules, normally 1-10, which supports an identifiable, traceable software function or group of related functions. The purpose of unit testing is to verify that the unit fulfills its specification and performs all allocated requirements satisfactorily. This is divided into two aspects, verifying that the implementation of the unit is in accordance with the design and verifying that the unit satisfies its allocated function, specified interface, and performance requirements.

Unit tests verify computations, data input/output, logic, structure, and exception conditions. They also verify interfaces and sequencing logic and standards and margin budgets. Unit testing is guided by the general unit test plan strategies and verifies satisfaction of design requirements.

The general unit test plan should define, at a minimum, the following categories of information:

1. Functional relationships - identify the requirements, top-level design specifications, and detailed design relationships.
2. Functional index - identifies how the functional requirements are documented in the unit development folder (UDF) and how the test cases are to be related to the requirements.
3. Organizational roles and responsibilities - define how, in general, the process of unit testing should be organized and structured and how the support functions are organizationally interfaced to the unit test process.
4. Unit test procedures - define how the individual unit test cases, both integration and functional, are to be documented in the UDFs, how the unit test scenarios and data are to be defined, developed, and reviewed, and who is responsible for development and conduct of the test cases.
5. Unit test reporting - defines how and by whom the results of unit testing are to be documented, evaluated, and reviewed.
6. Unit configuration definition - defines how the individual unit configurations are integrated, documented, evaluated, and reviewed.
7. Project interfaces - define how the unit test activities integrate with the related levels of software testing, how the relationships between unit test and the configuration management segments of the project are defined and maintained, and how support resources are requested and provided.
8. Unit test tools, techniques, and methodologies - describe what generic tools, techniques, and methodologies are available throughout the software project for use during unit testing and what the rules and constraints are for tailoring these to specific unit test application.

The techniques and the criteria for successful completion are determined by the attributes of the individual software element, rather than the need to demonstrate interface or system performance.

1. Is the design clear? Does it do what is intended?
2. Is the coding clear? Did you have trouble understanding it?
3. Are the comments helpful in understanding the routine? Would you have trouble modifying it?
4. Would you be proud of this work if it were yours?
5. Does the code meet the program's coding standards?
6. Do the tests meet the program's standards for unit testing?
7. Does input data vary over allowable values including maximum, minimum, and nominal values? (All alike data, especially all zeros, are usually a poor choice).
8. Is erroneous input data used? (In fact, all error conditions should be checked).
9. Can you think of erroneous data conditions which were not used?
10. Do the tests show that the routine has the functional capabilities allocated to it?
11. Do the tests demonstrate that the code completely satisfies each requirement allocated to it?
12. Does the actual output match the expected output?

Although unit testing is an informal process, it is the basic testing task evaluating the fundamental integrity of all software components. As such, the process must be rigorously planned and controlled. It must also be structured in accordance with the needs, characteristics, and requirements of the project.

|  |  |
| --- | --- |
| **Integration Testing** |  |

Software integration is the combining of components (software modules, programs, etc.) into an executable, operational software entity. Successfully integrating software involves developing an incremental delivery and integration scheme that best meshes with the selected integration approach, combining the software units into a cohesive, nominally executing product, and demonstrating software operability over a range of expected operating conditions. One aim of this is to reduce to a minimum the amount of redundant testing.

Integration should be an orderly and progressive process for building the system. It must be carefully planned and implemented in a timely manner. Software integration is related to all testing activities. The development process usually treats testing and integration as a common process. Integration options include "big bang," incremental development, and phased delivery. Most programs use combinations of integration techniques. The selections impact integration and test planning.

It is important for each project to develop an integration plan. This plan includes:

1. Top-level concepts, goals, and strategy.
2. Test activity network showing integration strategy in terms of test event and schedule interdependencies.
3. Source for prerequisite test data for each software entity at each integration level.
4. Strategy for verifying essential system services prior to use.
5. Concepts and methods for stressing software system.
6. Activities required to generate and validate preset database.
7. Identification of data generators, test drivers, environmental simulations, and test analysis programs required.
8. Activities required to produce nominal or calibration test data and stress test data.

During integration testing, units are integrated into an operational configuration; data relationships and internal execution characteristics, including performance benchmarks, are verified against the software design; internal execution sequences and support characteristics are verified. Software integration testing reviews and evaluates the software in relation to system level activities. It also provides assurance that the system satisfies the design criteria in the system and development specifications. It is the key to validating the computer resources.

Three basic approaches to integration:

* Top-down, where you integrate the modules by moving downward through the architecture
* Bottom-up, where you integrate modules by moving upward through the architecture
* Big bang, where you integrate all modules at once.

Top-down and bottom-up are organized, incremental approaches to integration and the program is constructed and tested in small segments, or builds.  Using a planned, incremental approach allows you to isolate and find errors, and the interfaces are more completely tested.  However, big bang is a non-incremental approach to integration and, while done very often, makes complete testing very difficult.  However, in reality, a combination of the three strategies is typically done and your team needs to develop a workable, incremental strategy in order to ensure that your project is correctly integrated and adequately tested.

|  |  |
| --- | --- |
| **Test Plan** |  |

As you have guessed, testing is hard and can be be just as hard as the primary project development.  As a result, testing requires careful planning that should be fully documented and for your course project, you will document your test plan using the following outline:

1. Scope of Testing
2. Unit Testing
   1. Unit Test Policy
   2. Unit Testing Procedures and Responsibilities
   3. Unit Test Case Results (one for each unit test)
3. Integration Testing
   1. Integration Testing Policy
   2. Integration Strategy
   3. Integration and Test Procedures (one for each build)
   4. Integration Testing Results

|  |  |
| --- | --- |
| **Summary** |  |

Testing is hard and it impossible to cover testing details in a short lecture and difficult to implement a full testing project in the short amount of time that you have available in the project.  However, you can use the principles discussed in the lecture to plan and develop simplified test plans, procedures, and test cases in order to ensure that your project meets the goal of the development--satisfying the customer's requirements and expectations.

|  |  |
| --- | --- |
| **References** |  |

1. Edward Kit, *Software Testing in the Real World,* Addison-Wesley, 1998.

2. Howard Pressman, *Software Engineering, A Practical Approach, 4th ed.*, McGraw Hill, 1997