Software Testing

Outline

- Regression testing
- Unit testing- unit test consideration, procedure, example using JUnit tool
- Integration testing – top down, bottom up, regression testing, smoke testing, strategic options, integration test work products
- Validation testing - Validation Criteria section, configuration review, alpha and beta testing
- System testing – Recovery, security, stress, performance, deployment testing
- Debugging – The debugging process, psychological considerations, debugging strategies, correcting the errors.

Regression testing

- Regression testing is the re-execution of some subset of tests that have already been tested, to ensure that a newly added module have not propagated unintended side effects.
- Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, or using automated capture/playback tools.
- The *regression test suite contains* three different classes of test cases:
  - A representative sample of tests that will exercise all software functions.
  - Additional tests that focus on software functions that are likely to be affected by the change.
  - Tests that focus on the software components that have been changed.
- Regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

Unit testing

- Unit testing focuses verification effort on the smallest unit of software design—the software component or module.
- It establishes to uncover errors within the boundary of the module.
- The unit test focuses on the internal processing logic and data structures within the boundaries of a component.
- This type of testing can be conducted in parallel for multiple components.
- Unit-test considerations. The following are exercised during this test.
  - Module interface
  - Local data structures
  - All independent paths
  - Boundary conditions
  - All error-handling paths
  - Data flow across
  - Local data structures
- Among the potential errors that should be tested when error handling is evaluated are:
  - (1) error description is unintelligible,
  - (2) error noted does not correspond to error encountered,
  - (3) error condition causes system intervention prior to error handling,
  - (4) exception-condition processing is incorrect,
  - (5) error description does not provide enough information to assist in the location of the cause of the error

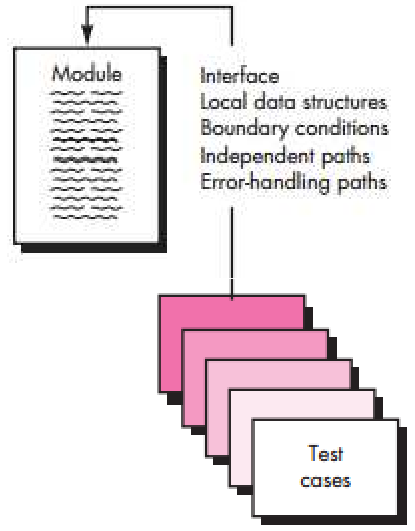


Fig. Unit test

Unit testing (Contd..)

- Unit-test procedures.
- Driver and/or stub software must often be developed for each unit test.
- A driver is nothing more than a “main program” that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- Stubs or “dummy subprogram” serve to replace modules that are subordinate (invoked by) the component to be tested. May do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs represent testing “overhead.” This software is not delivered with the final software product.
- Unit testing is simplified when a component with high cohesion is designed.

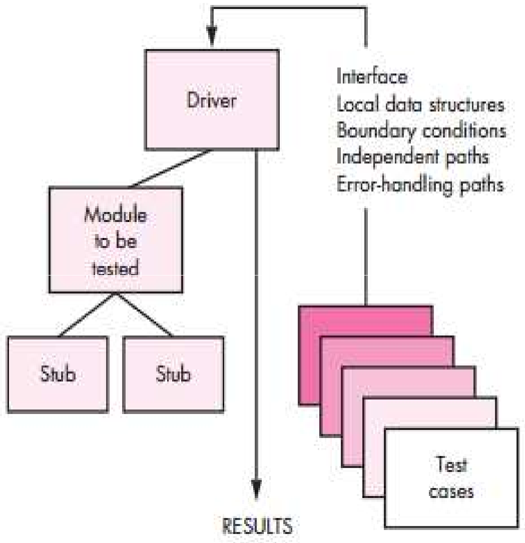
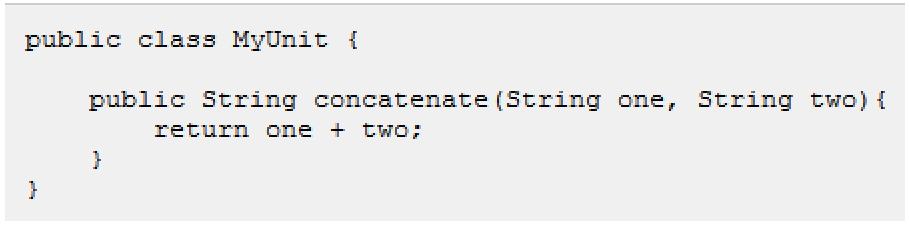


Fig. Unit test environment

Unit testing (Contd..)

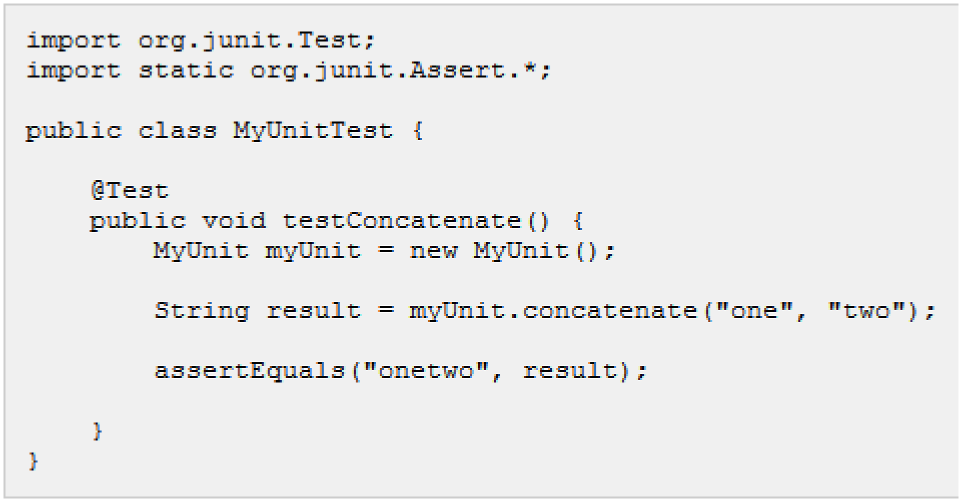
- Unit test example using JUnit tool : The class to test the concatenate() method:



```
public class MyUnit {

    public String concatenate(String one, String two){
        return one + two;
    }
}
```

- The assertEquals() method is called to do the actual testing. In this method we compare the output of the called method (concatenate()) with the expected output
  - If the two values are equal, it is normal.
  - If the two values are not equal, an exception is thrown, and the test method stops executing here.



```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyUnitTest {

    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();

        String result = myUnit.concatenate("one", "two");

        assertEquals("onetwo", result);

    }
}
```

Integration Testing

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
- The objective of integration testing is to take unit-tested components and build a program structure that has been dictated by design.
- Number of different incremental integration strategies:
  - Top down
  - Bottom up
  - Regression testing
  - Smoke testing
  - Strategic options
  - Integration test work products

Integration Testing(Contd..)

- Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated into the structure in either a depth-first or breadth-first manner.
- The integration process is performed in a series of five steps:
  - The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
  - Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
  - Tests are conducted as each component is integrated.
  - On completion of each set of tests, another stub is replaced with the real component.
  - Regression testing may be conducted to ensure that new errors have not been introduced.

- Bottom-up integration testing, begins construction and testing with atomic modules.
- A bottom-up integration strategy may be implemented with the following steps:
  - Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
  - A driver (a control program for testing) is written to coordinate test case input and output.
  - The cluster is tested.
  - Drivers are removed and clusters are combined moving upward in the program structure.

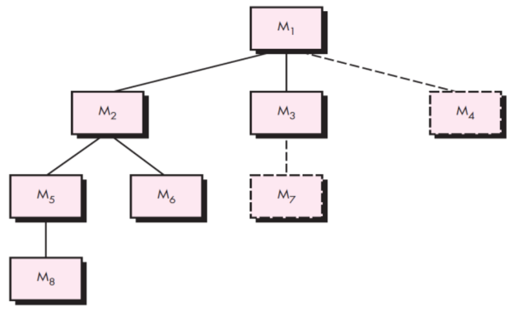


Fig. Top down integration testing

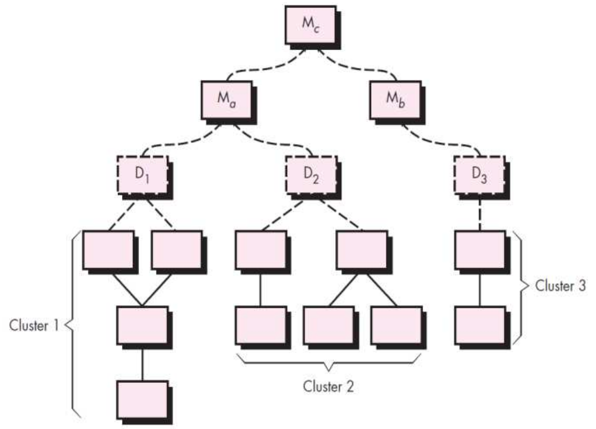


Fig. Bottom up integration testing

Integration Testing(Contd..)

- Smoke testing is an integration testing approach that is commonly used when product software is developed. The integration approach may be top down or bottom up
- It is designed for time-critical projects, allowing the software team to assess the project on a frequent basis.
- Smoke-testing approach encompasses the following activities:
  - Software components that have been translated into code are integrated into a build. A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
  - The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily.
- Smoke testing provides a number of benefits when it is applied on complex, time critical software projects:
  - Integration risk is minimized, the quality of the end product is improved, Error diagnosis and correction are simplified, Progress is easier to assess.

- Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called sandwich testing) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.
- As integration testing is conducted, the tester should identify critical modules.
- A critical module has one or more of the following characteristics:
  - (1) addresses several software requirements,
  - (2) has a high level of control (resides relatively high in the program structure),
  - (3) is complex or error prone
  - (4) has definite performance requirements.
- Critical modules should be tested as early as is possible.
- In addition, regression tests should focus on critical module function.

- **Integration test work products.** An overall plan for integration of the software and a description of specific tests is documented in a Test Specification. Test specification incorporates a test plan and a test procedure.

Validation testing

- Validation testing begins at the culmination of integration testing.
- Validation testing succeeds when software functions in a manner that can be reasonably expected by the customer.

- Validation Criteria section that forms the basis for a validation-testing approach.
- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- After each validation test case has been conducted, one of two possible conditions exists:
  - (1) The function or performance characteristic conforms to specification and is accepted
  - (2) a deviation from specification is uncovered and a deficiency list is created.
- Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery.

- Configuration review – element of the validation process.
- *The intent* of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail for maintenance activities.

- Alpha testing –
  - The *alpha test is conducted at the developer's site by a representative group of end* users.
  - It record errors and usage problems.
  - Alpha tests are conducted in a controlled environment.
- Beta testing (also called customer acceptance testing)
  - The *beta test is conducted at one or more end-user sites.*
  - The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
  - The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer

System testing

- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.
- The types of system tests for software-based systems are Recovery Testing, Security testing, Stress testing, Performance testing, and Deployment testing.
- Recovery testing : Test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.
- Security testing attempts to verify that protection mechanisms built into a system. The tester plays the role(s) of the individual who desires to penetrate the system The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.
- Stress testing : Executes a system resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created.
- Performance testing : Test the run-time performance of software. Measure resource utilization (e.g., processor cycles), monitor execution intervals, log events (e.g., interrupts), and sample machine states on a regular basis. The tester can uncover situations that lead to degradation and possible system failure.
- Deployment testing : Deployment or configuration testing, exercises the software in each environment in which it is to operate. Examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers.

Debugging

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

- Debugging process : Debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found.

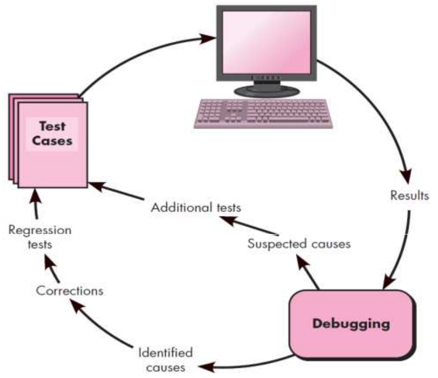


Fig. The Debugging process

- Psychological Considerations : debugging prowess is an innate human trait. Some people are good at it and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.
- Debugging Strategies : Debug is to find and correct the cause of a software error or defect. three debugging strategies have been proposed (1) brute force, (2) backtracking, and (3) cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.
- Correcting the Error : Correction of a bug can introduce other errors and therefore do more harm than good. Three simple questions to ask before making the "correction": *Is the cause of the bug reproduced in another part of the program? What "next bug" might be introduced by the fix I'm about to make? What could we have done to prevent this bug in the first place?*