SOFTWARE TESTING STRATEGIES

A STRATEGIC APPROACH TO SOFTWARE TESTING

SOFTWARE TESTING STRATEGIES

- A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required.
- Therefore, any testing strategy must incorporate test planning, test-case design, test execution, and resultant data collection and evaluation. A software testing strategy should be flexible enough to promote a customized testing approach.
- At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses

A STRATEGIC APPROACH TO SOFTWARE TESTING

Verification and Validation

Organizing for Software Testing

Software Testing Strategy—The Big Picture

Criteria for Completion of Testing

A STRATEGIC APPROACH TO SOFTWARE TESTING

- Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specifi c test-case design techniques and testing methods—should be defi ned for the software process

A STRATEGIC APPROACH TO SOFTWARE TESTING

Generic characteristics of software testing strategies:

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

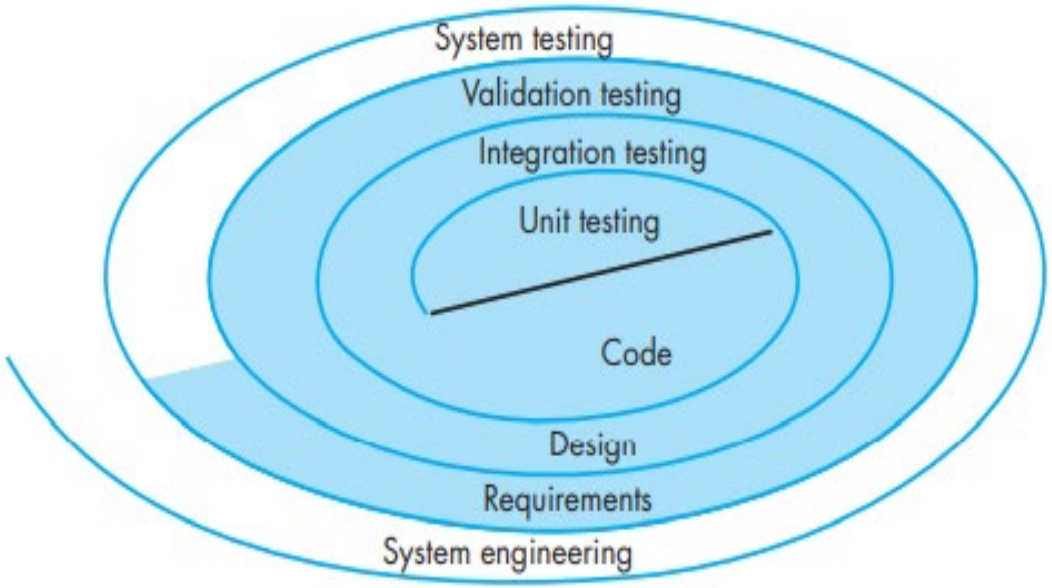
Verification and Validation

- Software testing is one element of a broader topic that is often referred to as verification and validation (V&V).
- Verification refers to the set of tasks that ensure that software correctly implements a specific function.
- Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Verification: “Are we building the product right?”
- Validation: “Are we building the right product?”

Organizing for Software Testing

- The **software developer** is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed.
- In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture.
- Only after the software architecture is complete does an independent test group become involved.
- The role of an **independent test group (ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built.
- Independent testing removes the conflict of interest that may otherwise be present.
- The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted.
- While testing is conducted, the developer must be available to correct errors that are uncovered.

Software Testing Strategy—The Big Picture

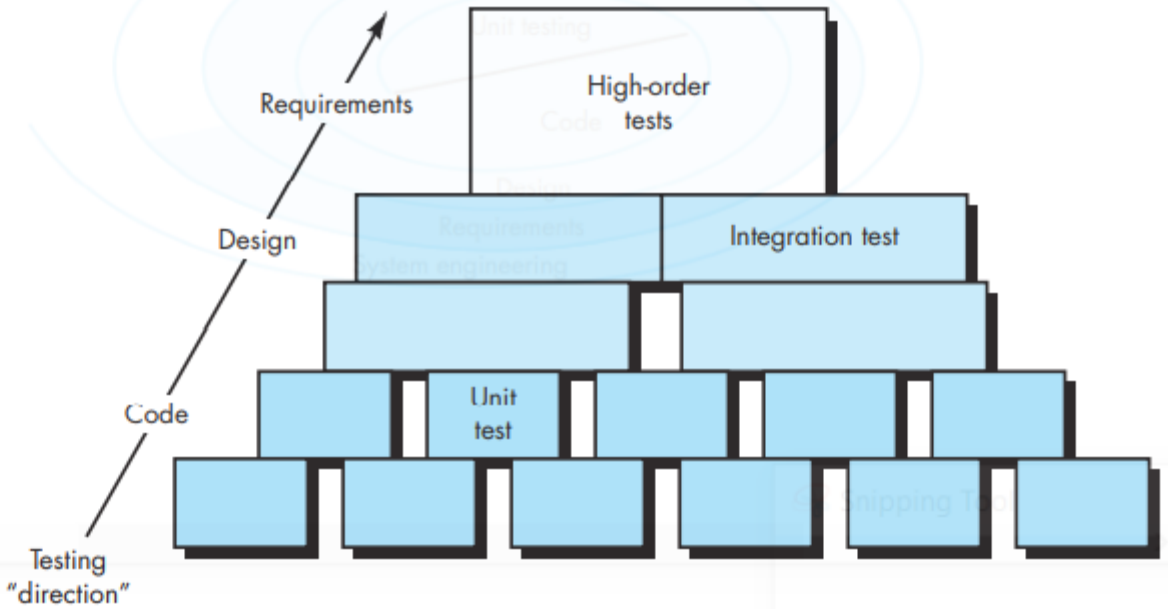


Software Testing Strategy—The Big Picture

- Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.
- Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture.
- Taking another turn outward on the spiral, you encounter validation testing, where requirements established as part of requirements modeling are validated against the software that has been constructed.
- Finally, you arrive at system testing, where the software and other system elements are tested as a whole.
- To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.

FIGURE 22.2

Software testing steps



- “You're never done testing; the burden simply shifts from you (the software engineer) to the end user.”
- Every time the user executes a computer program, the program is being tested.
- “You’re done testing when you run out of time or you run out of money.”

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

Unit Testing

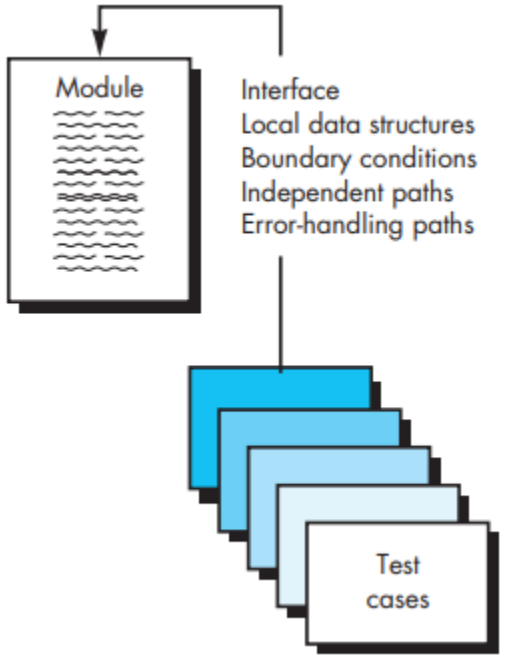
Unit Testing

- Unit testing focuses verification effort on the smallest unit of software design— the software component or module.
- Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.
- 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.

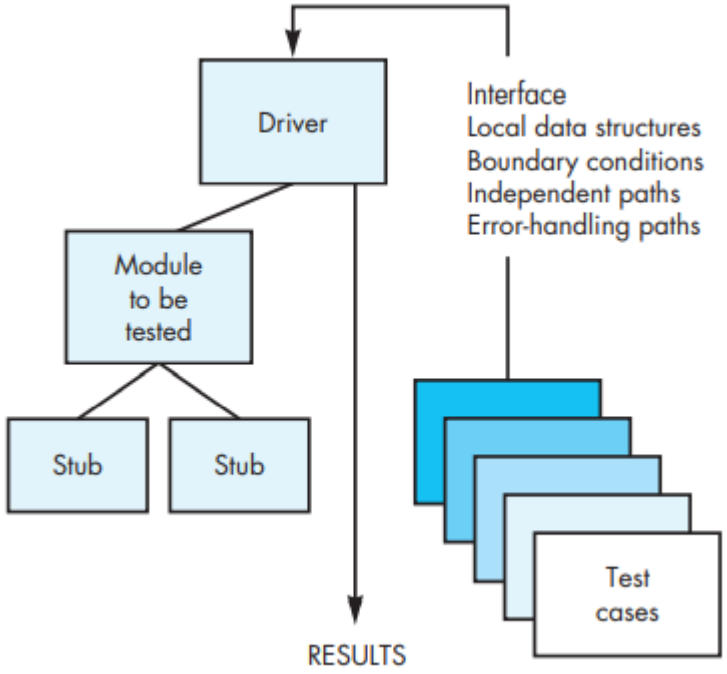
- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- And finally, all error-handling paths are tested.
- Data flow across a component interface is tested before any other testing is initiated.
- If data do not enter and exit properly, all other tests are moot.
- In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Unit Test Considerations.



Unit-Test Procedures.

- The design of unit tests can occur before coding begins or after source code has been generated.
- Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.
- The unit test environment is illustrated in Figure
- In most applications 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 serve to replace modules that are subordinate (invoked by) the component to be tested.
- A stub or “dummy subprogram” uses the subordinate module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs represent testing “overhead.” That is, both are software that must be coded (formal design is not commonly applied) but that is not delivered with the final software product.
- If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with “simple” overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).



Integration Testing

Top-Down Integration

Bottom-Up Integration

Regression Testing

Smoke Testing

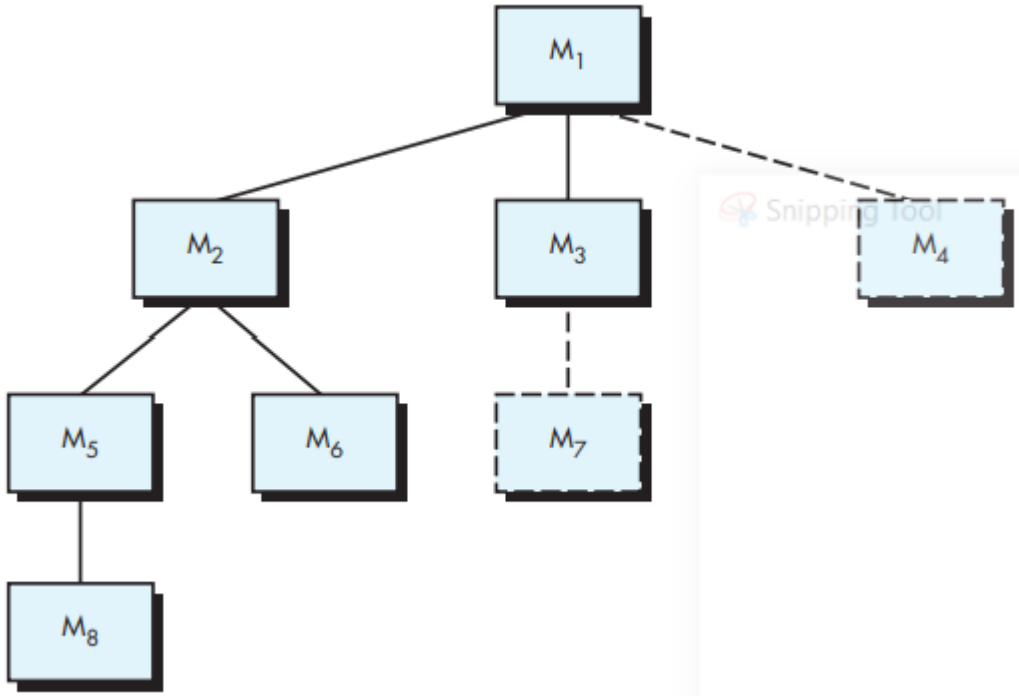
Integration Testing

- 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 is to take unit-tested components and build a program structure that has been dictated by design.
- Non incremental integration -to construct the program using a “big bang” approach. All components are combined in advance and the entire program is tested as a whole. Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-Down Integration

- .Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depthfirst or breadth-first manner.

Top-Down Integration



Top-Down Integration

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

- The top-down integration strategy verifies major control or decision points early in the test process.

Bottom-Up Integration

- Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).
- Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

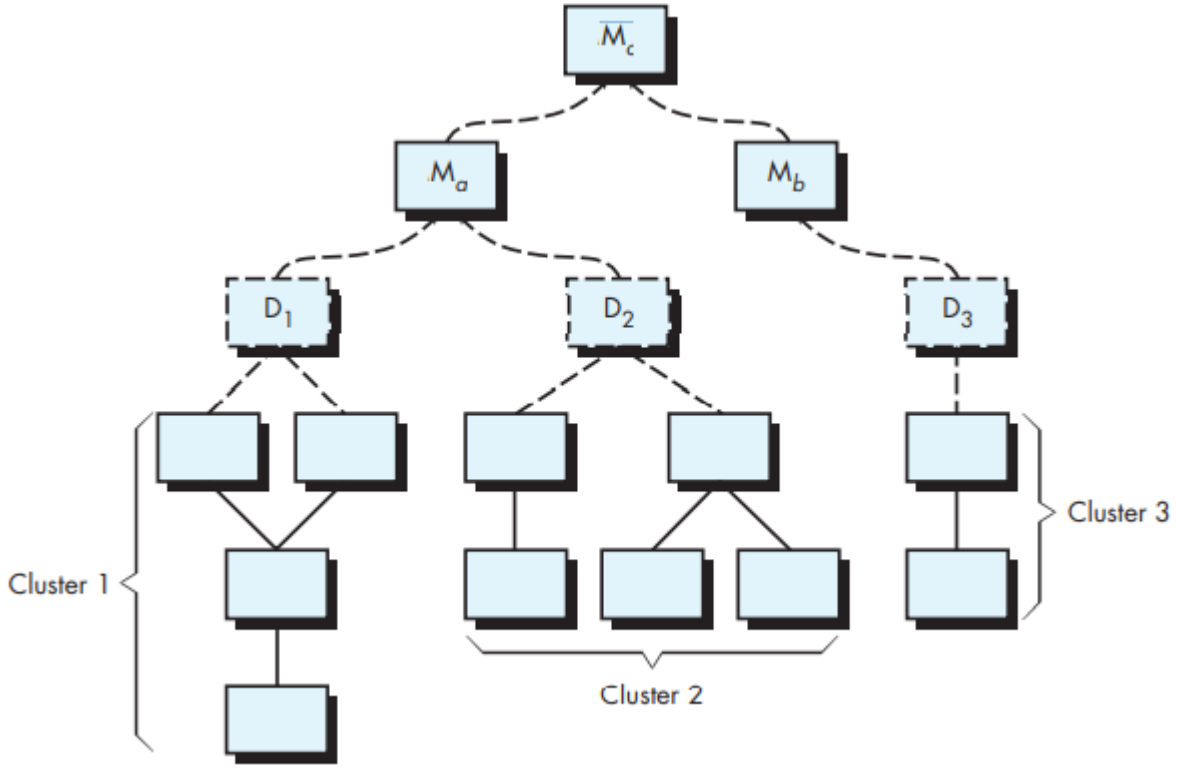
1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

2. A driver (a control program for testing) is written to coordinate test-case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Bottom-Up Integration.



Regression Testing

- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked. Side effects associated with these changes may cause problems with functions that previously worked flawlessly.
- In the context of an integration test strategy, **regression testing** is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression Testing

- Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools.
- Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) 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.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the 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.

Smoke Testing

- Smoke testing is an integration testing approach that is commonly used when product software is developed.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

In essence, the smoke-testing approach encompasses the following activities:

1. 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.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function.

3. 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 benefi ts 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.

Validation testing

- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.
- At the validation or system level, the distinction between different software categories disappears.
- Testing focuses on user-visible actions and user-recognizable output from the system.
- Validation succeeds when software functions in a manner that can be reasonably expected by the customer

Validation testing

Validation-Test Criteria

- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
- If a deviation from specification is uncovered, a deficiency list is created.
- A method for resolving deficiencies (acceptable to stakeholders) must be established.

Configuration Review

- An important element of the validation process is a configuration review.
- 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 to bolster the support activities.
- The confi guration reviewis sometimes called an audit.

Validation testing

- Alpha and Beta Testing
- Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

Alpha Tests

- The alpha test is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
- Alpha tests are conducted in a controlled environment.

Beta Tests

- The beta test is conducted at one or more end-user sites
- Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
- As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

Customer Acceptance Testing

- A variation on beta testing, called customer acceptance testing, is sometimes performed when custom software is delivered to a customer under contract.
- The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SYSTEM TESTING

- A classic system-testing problem is “finger pointing.” This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem.
- Rather than indulging in such nonsense, you should anticipate potential interfacing problems and

(1) design error-handling paths that test all information coming from other elements of the system,

(2) conduct a series of tests that simulate bad data or other potential errors at the software interface,

(3) record the results of tests to use as “evidence” if finger pointing does occur, and

(4) participate in planning and design of system tests to ensure that software is adequately tested.

SYSTEM TESTING

Recovery testing

- Recovery testing is a system 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

- Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."
- Given enough time and resources, good security testing will ultimately penetrate a system.
- The role of the system designer is to make penetration cost more than the value of the information that will be obtained

SYSTEM TESTING

Stress Testing

- Stress tests are designed to confront programs with abnormal situations.
- In essence, the tester who performs stress testing asks: “How high can we crank this up before it fails?”
- Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- For example, (1) special tests may be designed that generate 10 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.
- Essentially, the tester attempts to break the program.
- A variation of stress testing is a technique called **sensitivity testing.** In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing

SYSTEM TESTING

Performance Testing

- Performance testing is designed to test the run-time performance of software within the context of an integrated system.
- Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted.
- However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.
- That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.
- External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis.
- By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

SYSTEM TESTING

Deployment Testing

- Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

Debugging

The Debugging Process
Psychological Considerations
Debugging Strategies
Correcting the Error

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.

The Debugging Process

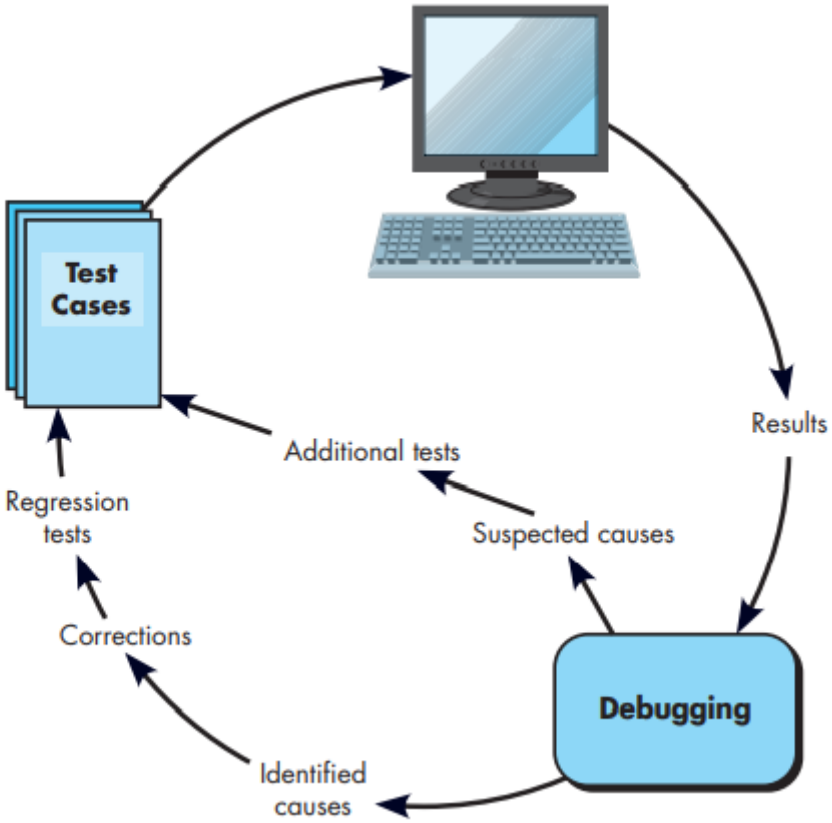
- The debugging process begins with the execution of a test case.
- Results are assessed and a lack of correspondence between expected and actual performance is encountered.
- In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden.
- The debugging process attempts to match symptom with cause, thereby leading to error correction.
- 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.

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

The Debugging Process



The Debugging Process

Why is debugging so difficult?

1. The symptom and the cause may be geographically remote.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. 8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to fi nd the cause also increases.

Psychological Considerations

- Debugging process is an innate human trait. Some people are good at it and others aren’t.
- Large variances in debugging ability have been reported for programmers with the same education and experience.

SafeHome

Debugging

The scene: Ed's cubical as code and unit testing is conducted.

The players: Ed and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Shakira (looking in through the entrance to the cubical): Hey . . . where were you at lunchtime?

Ed: Right here . . . working.

Shakira: You look miserable . . . what's the matter?

Ed (sighing audibly): I've been working on this bug since I discovered it at 9:30 this morning and it's what, 2:45 . . . I'm clueless.

Shakira: I thought we all agreed to spend no more than one hour debugging stuff on our own; then we get help, right?

Ed: Yeah, but . . .

Shakira (walking into the cubical): So what's the problem?

Ed: It's complicated, and besides, I've been looking at this for, what, 5 hours. You're not going to see it in 5 minutes.

Shakira: Indulge me . . . what's the problem?

[Ed explains the problem to Shakira, who looks at it for about 30 seconds without speaking, then . . .]

Shakira (a smile is gathering on her face): Uh, right there, the variable named *setAlarmCondition*. Shouldn't it be set to "false" before the loop gets started?

[Ed stares at the screen in disbelief, bends forward, and begins to bang his head gently against the monitor. Shakira, smiling broadly now, stands and walks out.]

Debugging Strategies

Three debugging strategies have been proposed :

- brute force
- backtracking
- cause elimination.

Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging Strategies

Brute force

- The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error.
- You apply brute force debugging methods when all else fails.
- Using a “let the computer find the error” philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you’ll find a clue that can lead us to the cause of an error.
- Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time.

Backtracking

- Backtracking is a fairly common debugging approach that can be used successfully in small programs.
- Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found.
- Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

Cause elimination

- The third approach to debugging— cause elimination—is manifested by induction or deduction and introduces the concept of binary partitioning.
- Data related to the error occurrence are organized to isolate potential causes.
- A “cause hypothesis” is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each.
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Debugging Strategies

Automated Debugging

- Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted.
- Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation."
- A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available.
- However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

Debugging Strategies

- **The People Factor.**

Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people!

Correcting the Error

- Once a bug has been found, it must be corrected.
- Three simple questions that you should ask before making the “correction” that removes the cause of a bug.
  - Is the cause of the bug reproduced in another part of the program?
  - What “next bug” might be introduced by the fi x I'm about to make?
  - What could we have done to prevent this bug in the first place?

Testing

- Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer

SOFTWARE TESTING FUNDAMENTALS

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer-based system or a product with “testability” in mind.
- The **tests** themselves **must exhibit a set of characteristics** that achieve the goal of finding the most errors with a minimum of effort.
  - Testability.
  - Operability.
  - Observability
  - Controllability.
  - Decomposability.
  - Simplicity.
  - Stability.
  - Understandability.

Test Characteristics

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex.

INTERNAL AND EXTERNAL VIEWS OF TESTING

- Any engineered product can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

- The first test approach takes an external view and is called **black-box testing**. The second requires an internal view and is termed **white-box testing.**

- **Black-box testing** alludes to tests that are conducted at the **software interface**. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
- White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.
- At first glance it would seem that very thorough **white-box testing** would lead to “100 percent correct programs.” All we need do is **define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively.**
- Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large.
- White-box testing should not, however, be dismissed as impractical. **A limited number of important logical paths can be selected and exercised**. Important data structures can be probed for validity.