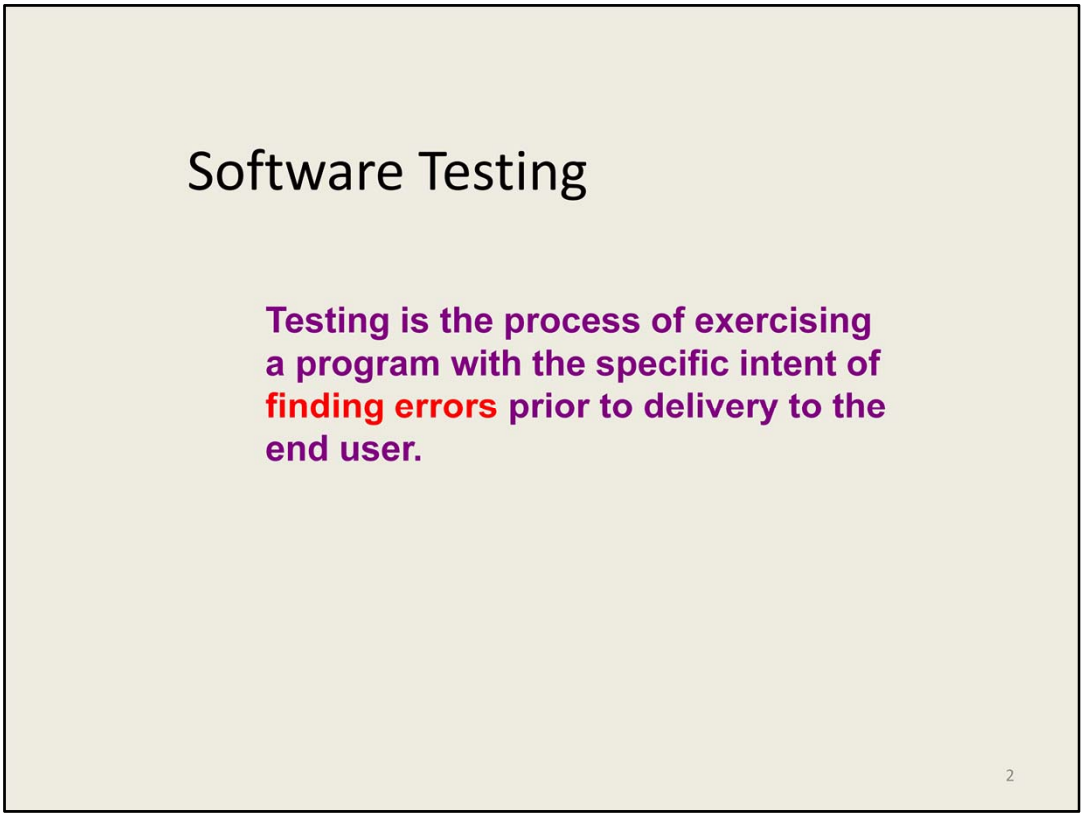
Software Quality Management

Software Testing Strategies

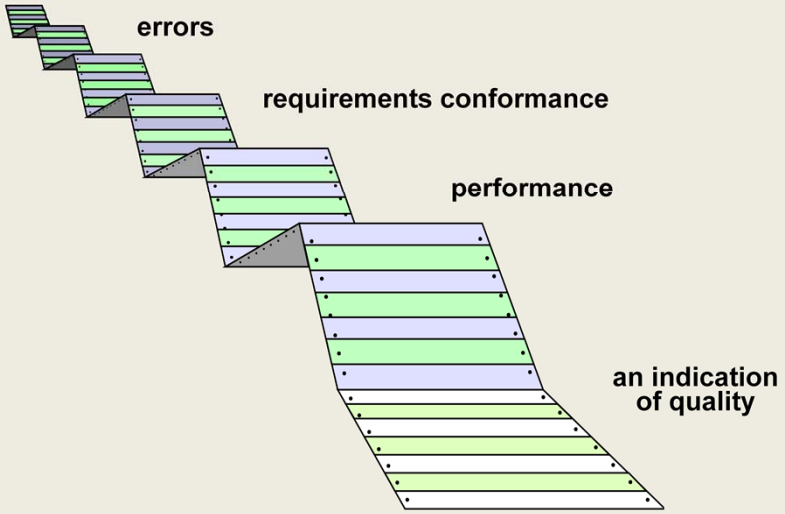
1



different test methods are beginning to cluster themselves into several distinct approaches and philosophies.

These “approaches and philosophies” are what I call *strategy*

What Testing Shows



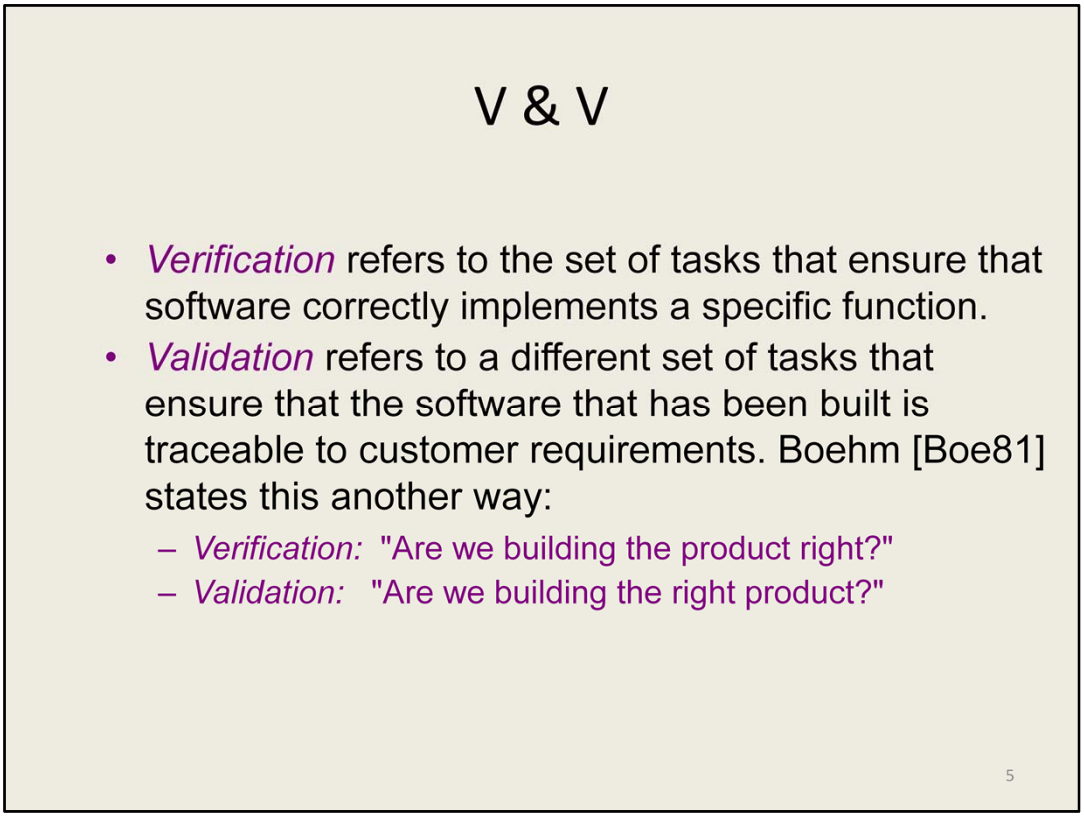
3

Strategic Approach

- 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.

4

A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:



Can you remind me of the software process model that is featured with V&V also, why (because V model also emphasizes quality assurance via 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 and validation includes a wide array of SQA activities: technical

reviews, quality and configuration audits, performance monitoring, simulation, feasibility

study, documentation review, database review, algorithm analysis, development

testing, usability testing, qualification testing, acceptance testing, and installation testing.

Although testing plays an extremely important role in V&V, many other

activities
are also necessary.

Who Tests the Software?

developer

Understands the system
but, will test "gently"
and, is driven by "delivery"

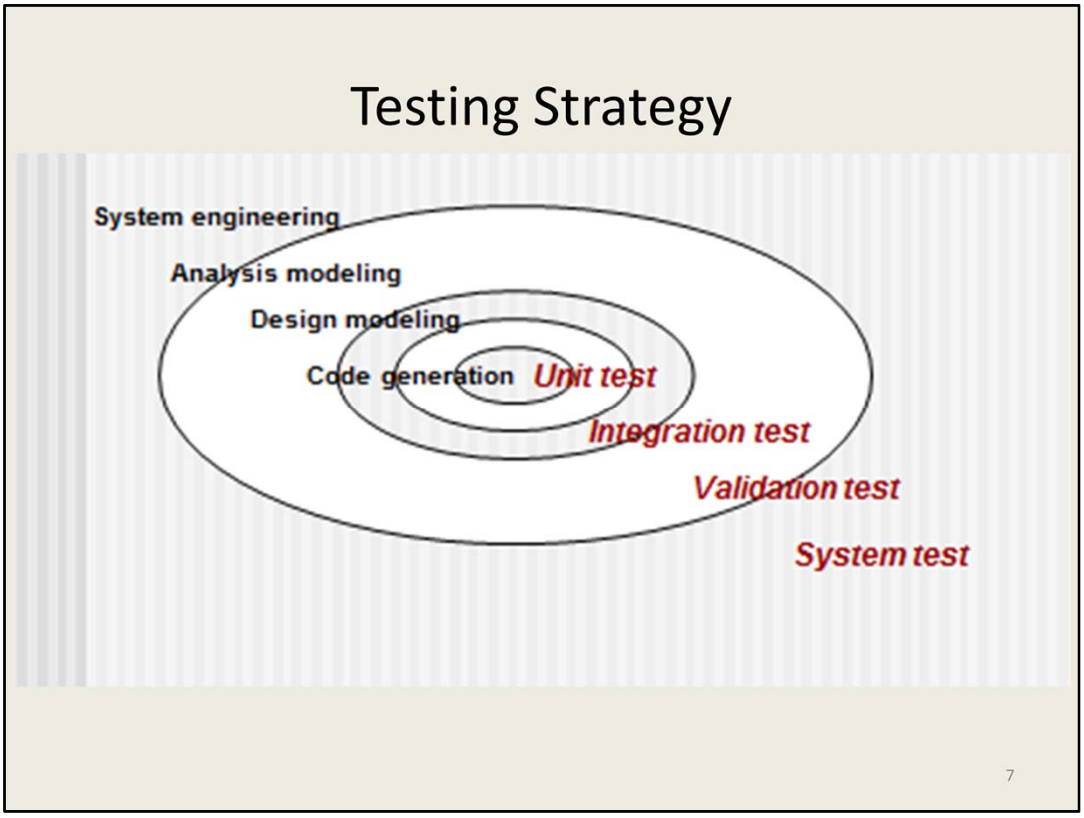
independent tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

6

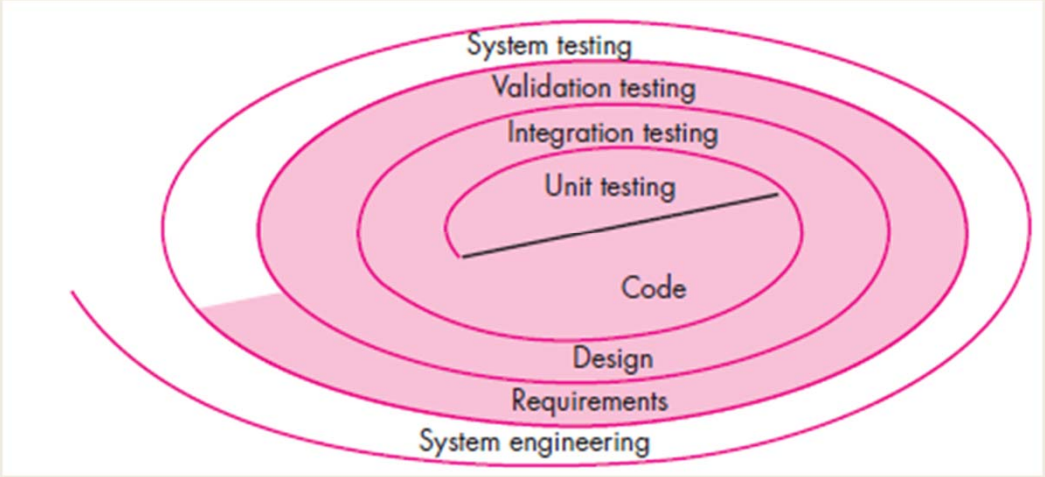
The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating

that the program is error-free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigate against thorough testing.



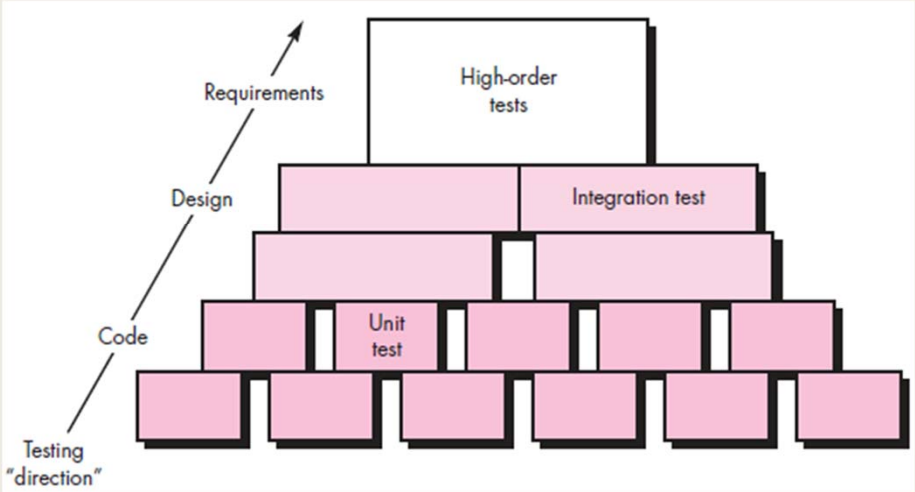
A strategy for software testing may also be viewed in the context of the spiral (Figure 17.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving 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 in a clockwise direction along streamlines that broaden the scope of testing with each turn.

Testing Strategy



8

Testing Strategy



9

Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing*

Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction.

After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware,

people,

databases). *System testing* verifies that all elements mesh properly and that overall

system function/performance is achieved.

Testing Strategy

- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective technical reviews as a filter prior to testing
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

11

a software testing strategy will succeed when software testers:

Specify product requirements in a quantifiable manner long before testing commences.

Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability,

and usability (Chapter 14). These should be specified in a way that is measurable so that testing results are unambiguous.

State testing objectives explicitly. The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, meantime-

to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours should be stated within the test plan.

Understand the users of the software and develop a profile for each user category.

Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

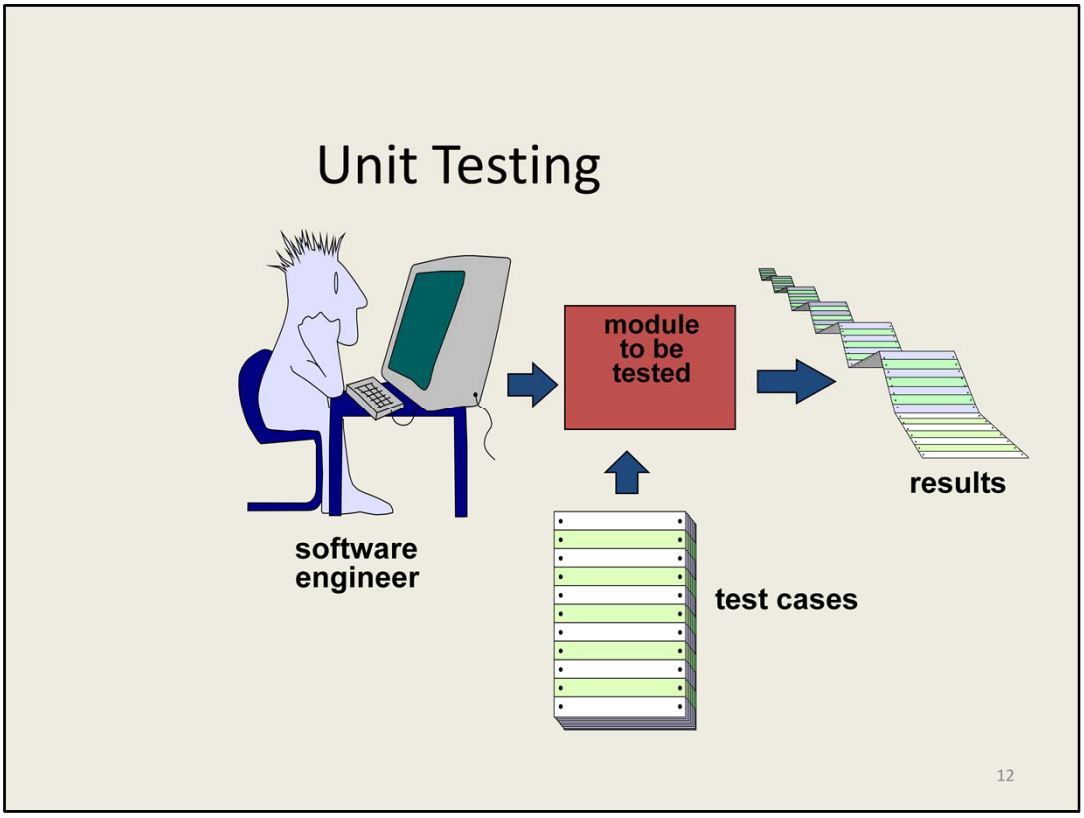
Develop a testing plan that emphasizes "rapid cycle testing." Gilb [Gil95] recommends that a software team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Build "robust" software that is designed to test itself. Software should be designed in a manner that uses antibugging (Section 17.3.1) techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.

Use effective technical reviews as a filter prior to testing. Technical reviews (Chapter 15) can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce highquality software.

Conduct technical reviews to assess the test strategy and test cases themselves. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.



Unit testing focuses verification effort on the smallest unit of software design—the

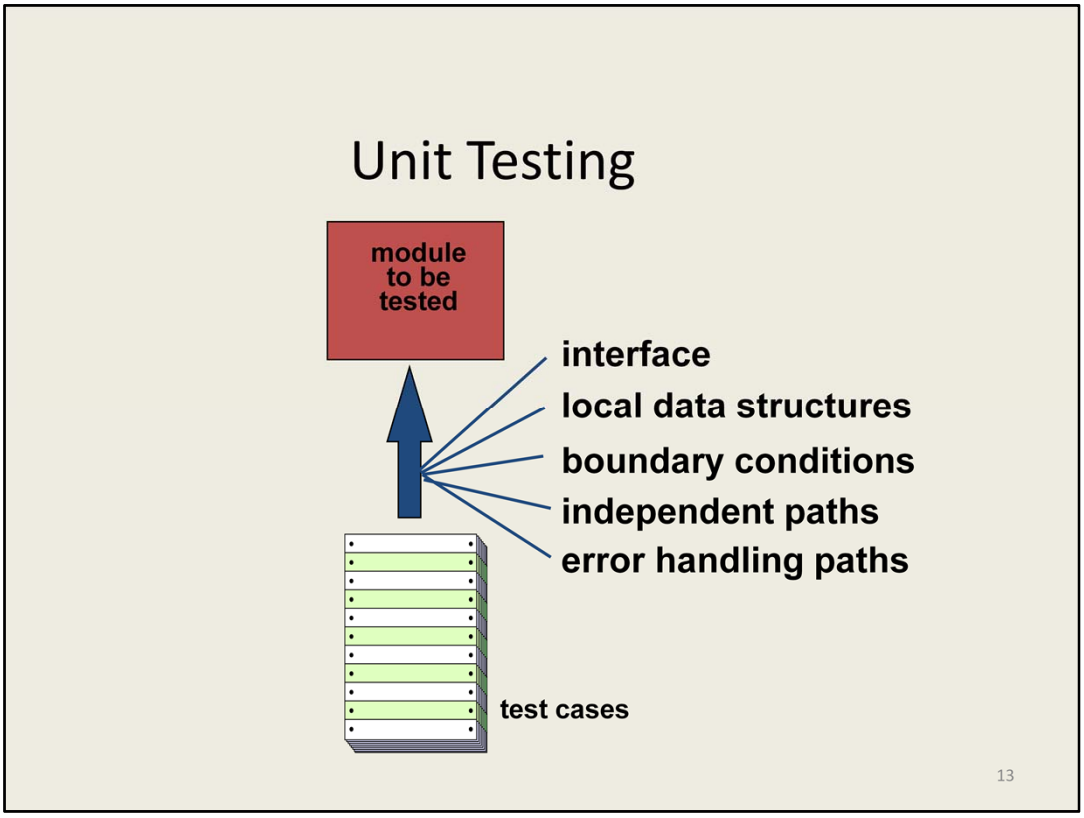
software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of

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.



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.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect

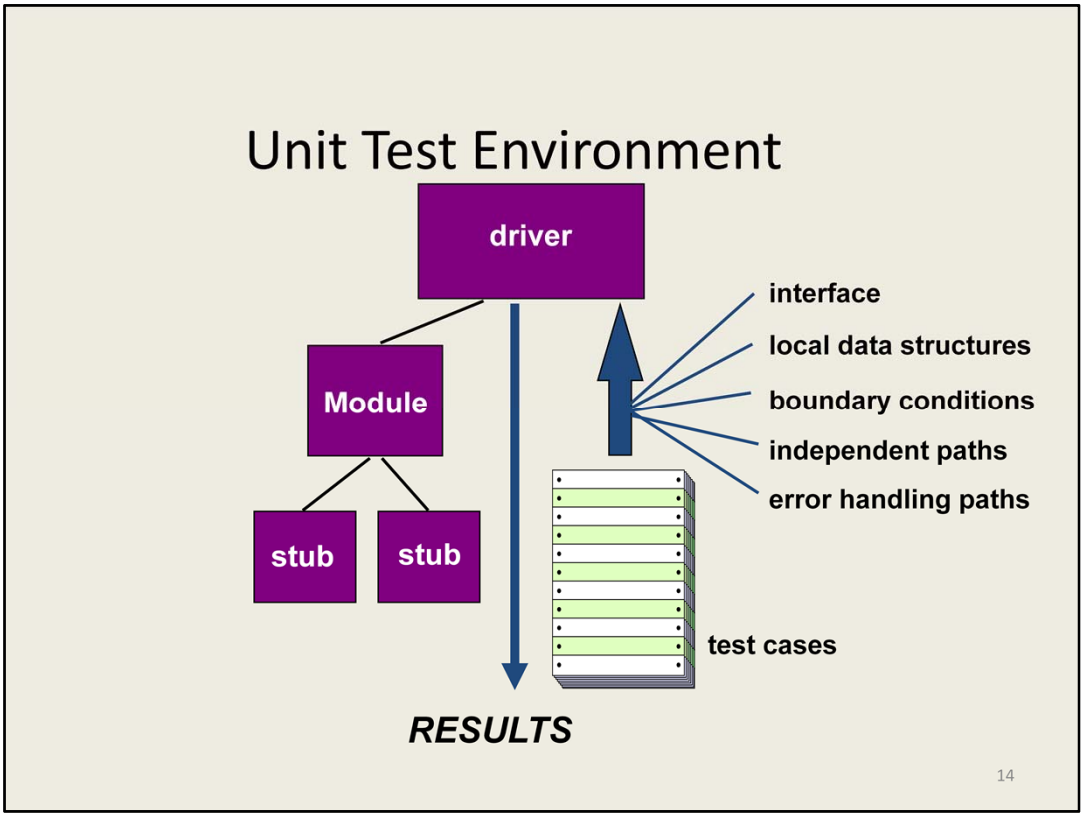
comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is

invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at,

and

just above maxima and minima are very likely to uncover errors.



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

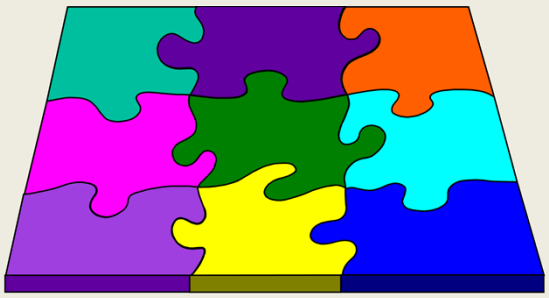
are tested to ensure that the module operates properly at boundaries established to

limit or restrict processing. And finally, all error-handling paths are tested.

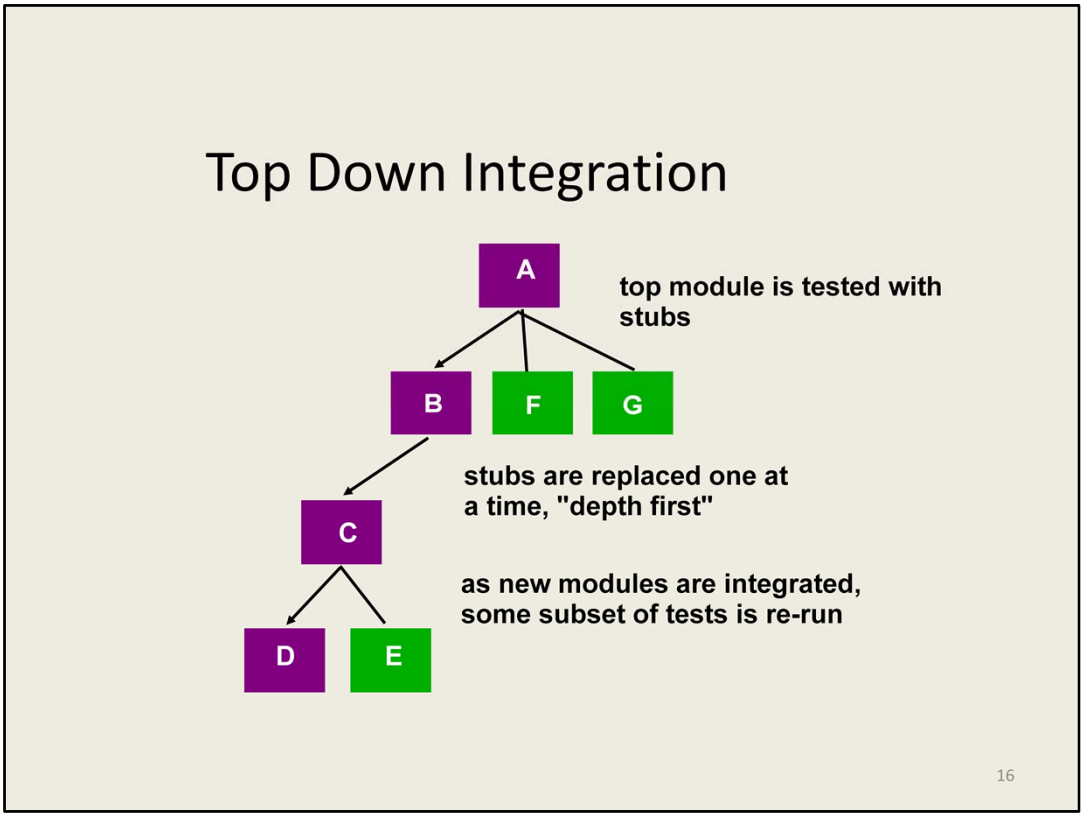
Integration Testing Strategies

Options:

- **the “big bang” approach**
- **an incremental construction strategy**



15



The integration process is performed in a series of five steps:

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.

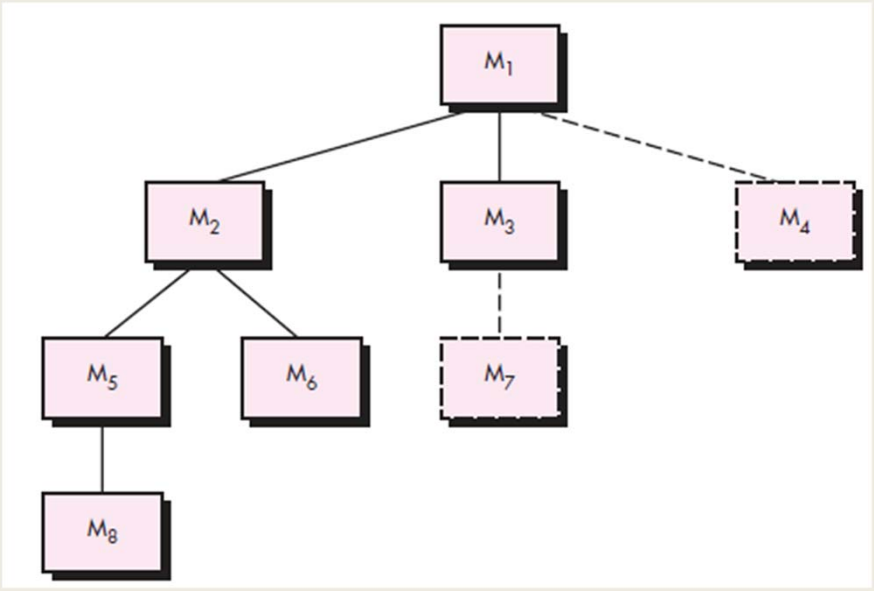
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

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.

Top Down Integration



17

depth-first integration integrates all components on a

major control path of the program structure. Selection of a major path is somewhat

arbitrary and depends on application-specific characteristics. For example, selecting

the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if

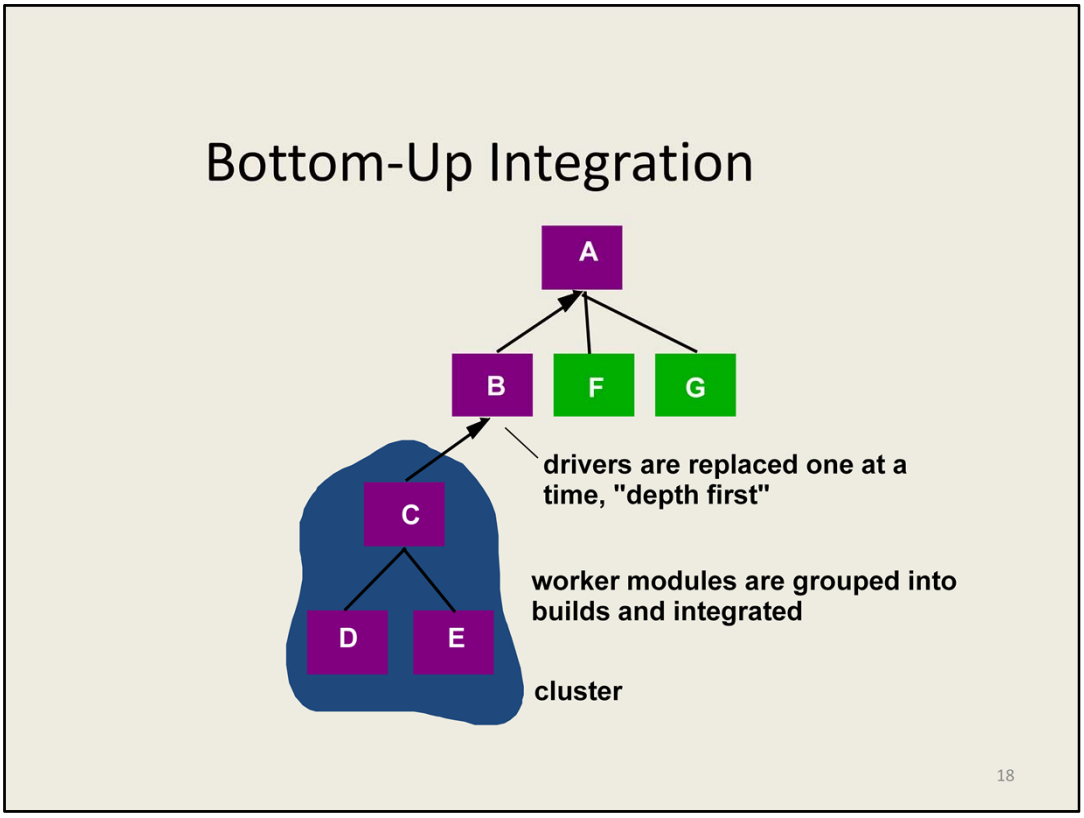
necessary for proper functioning of M2) M6 would be integrated. Then, the central

and right-hand control paths are built. *Breadth-first integration* incorporates all components

directly subordinate at each level, moving across the structure horizontally.

From the figure, components M2, M3, and M4 would be integrated first. The next control

level, M5, M6, and so on, follows.



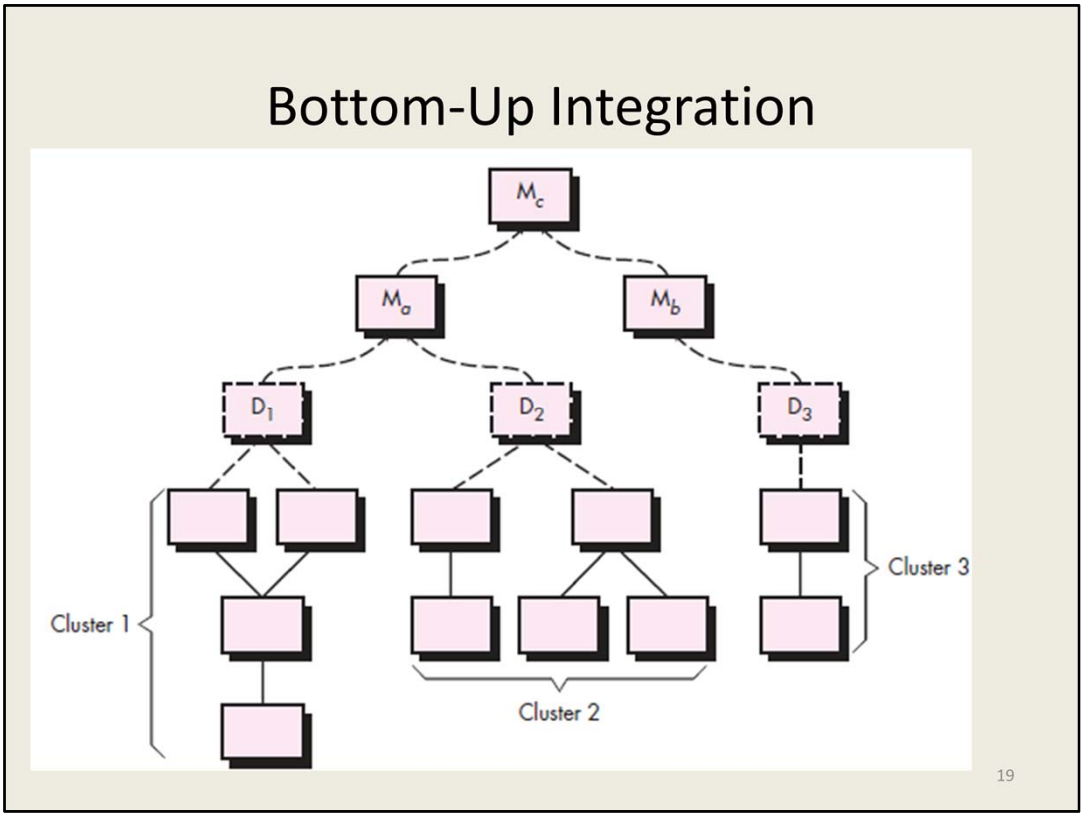
A bottom-up integration strategy may be implemented with the following steps:

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.



Integration follows the pattern illustrated in Figure 17.6. Components are combined

to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown

as a dashed block). Components in clusters 1 and 2 are subordinate to M*a*. Drivers D1

and D2 are removed and the clusters are interfaced directly to M*a*. Similarly, driver D3

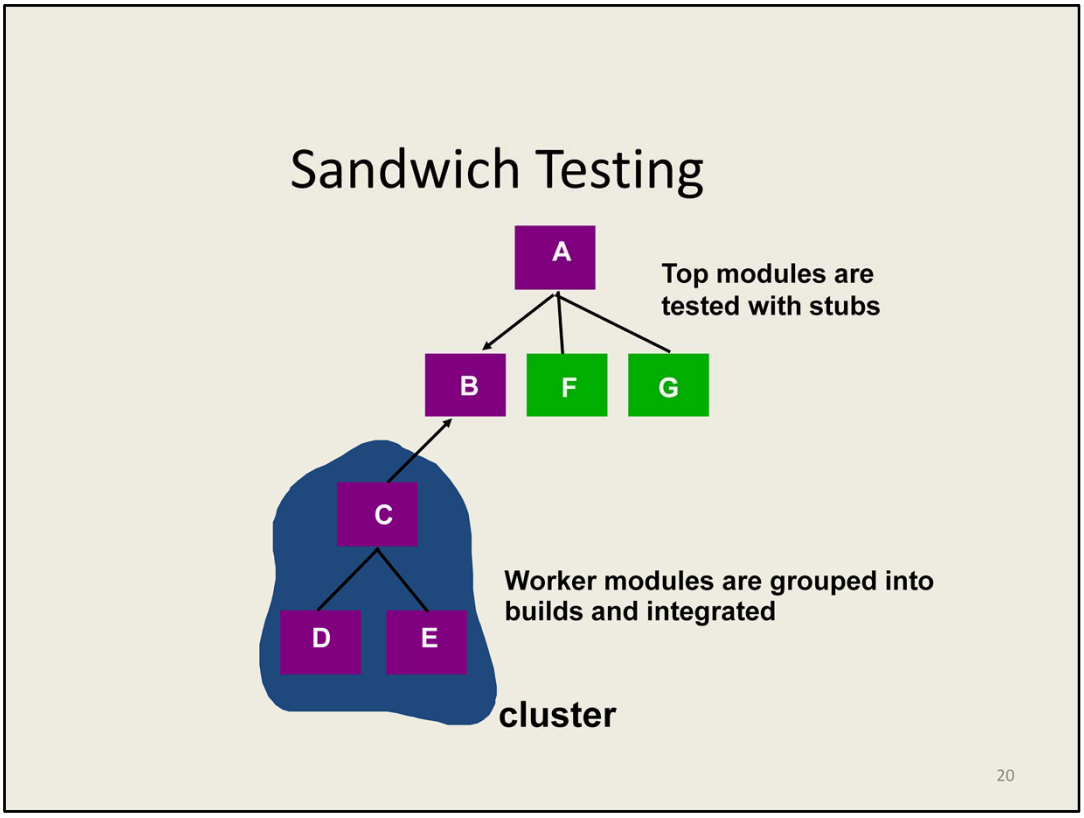
for cluster 3 is removed prior to integration with module M*b*. Both M*a* and M*b* will

ultimately be integrated with component M*c*, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact,

if the top two levels of program structure are integrated top down, the number of

drivers can be reduced substantially and integration of clusters is greatly simplified.



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, or (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.

Regression Testing

- *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

21

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.

……..

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.

Smoke Testing

- A common approach for creating “daily builds” for product software
- Smoke testing steps:
  - 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 intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

22

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:

Object-Oriented Testing

- begins by evaluating the correctness and consistency of the analysis and design models
- testing strategy changes
  - the concept of the 'unit' broadens due to encapsulation
  - integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
  - validation uses conventional black box methods
- test case design draws on conventional methods, but also encompasses special features

23

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span.

Although this fundamental objective remains unchanged for object-oriented software,

the nature of object-oriented software changes both testing strategy and testing tactics

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data.

An encapsulated class is usually the focus of unit testing. However,

operations (methods) within the class are the smallest testable units

Broadening the View of "Testing"

- It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

24

The construction of object-oriented software begins with the creation of analysis and design models. Because of the evolutionary nature of the OO software engineering paradigm, these models begin as relatively informal representations of system requirements and evolve into detailed models of classes, class connections and relationships, system design and allocation, and object design incorporating a model of object connectivity via messaging). At each stage, the models can be tested in an attempt to uncover errors prior to their propagation to the next iteration.

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code levels. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

For example, consider a class in which a number of attributes are defined during the first iteration of OOA. An extraneous attribute is appended to the class (due to a misunderstanding of the problem domain). Two operations are then specified to manipulate the attribute. A review is conducted and a domain expert points out the error. By eliminating the extraneous attribute at this stage, the following problems and unnecessary effort may be avoided during analysis:

1. Special subclasses may have been generated to accommodate the unnecessary attribute or exceptions to it. Work involved in the creation of unnecessary subclasses has been avoided.
2. A misinterpretation of the class definition may lead to incorrect or extraneous class relationships.

3. The behavior of the system or its classes may be improperly characterized to accommodate the extraneous attribute.

If the error is not uncovered during analysis and propagated further, the following problems could occur (and will have been avoided because of the earlier review) during design:
1. Improper allocation of the class to subsystem and/or tasks may occur during system design.
2. Unnecessary design work may be expended to create the procedural design for the operations that address the extraneous attribute.
3. The messaging model will be incorrect (because messages must be designed for the operations that are extraneous).

If the error remains undetected during design and passes into the coding activity, considerable effort will be expended to generate code that implements an unnecessary attribute, two unnecessary operations, messages that drive interobject communication, and many other related issues. In addition, testing of the class will absorb more time than necessary. Once the problem is finally uncovered, modification of the system must be carried out with the ever-present potential for side effects that are caused by change.

During later stages of their development, OOA and OOD models provide substantial information about the structure and behavior of the system. For this reason, these models should be subjected to rigorous review prior to the generation of code.

All object-oriented models should be tested (in this context, the term testing is used to incorporate formal technical reviews) for correctness, completeness, and consistency within the context of the model's syntax, semantics, and pragmatics.

Testing the CRC Model

- 1. Revisit the CRC model and the object-relationship model.
- 2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
- 3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
- 4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
- 5. Determine whether widely requested responsibilities might be combined into a single responsibility.
- 6. Steps 1 to 5 are applied iteratively to each class and through each evolution of the analysis model.

OO Testing Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

26

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies (Section 17.3.2) have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class.

There are two different strategies for integration testing of OO systems.

The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

The use of drivers and stubs also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

WebApp Testing - I

- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.

27

WebApp Testing - II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

28

High Order Testing

- Validation testing
  - Focus is on software requirements
- System testing
  - Focus is on system integration
- Alpha/Beta testing
  - Focus is on customer usage
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system

29

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.

As an example, consider the Internet-accessible version of *SafeHome* software that would allow a customer to monitor the security system from remote locations.

The *SafeHome* WebApp must be tested using all Web browsers that are likely to be
encountered. A more thorough deployment test might encompass combinations of Web browsers with various operating systems (e.g., Linux, Mac OS, Windows).

Because security is a major issue, a complete set of security tests would be integrated
with the deployment test.

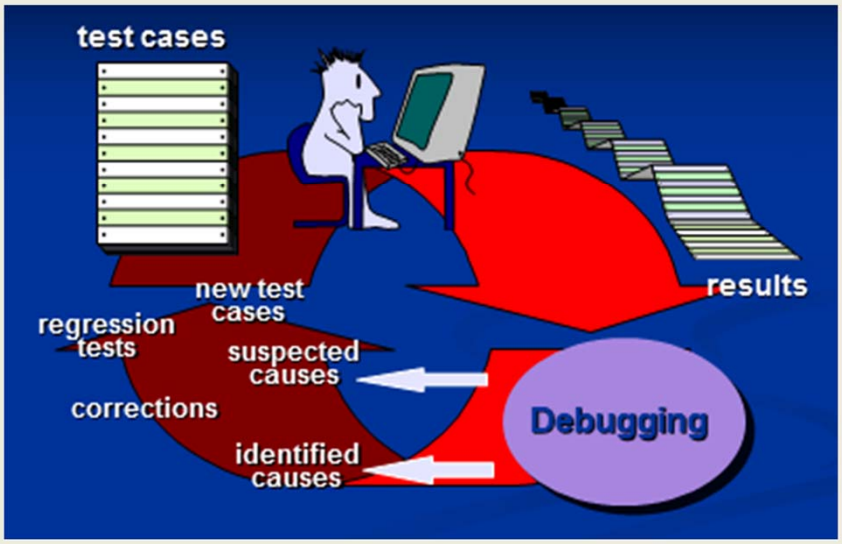
Debugging: A Diagnostic Process

30

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 is not testing but often occurs as a consequence of testing. 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.

The Debugging Process

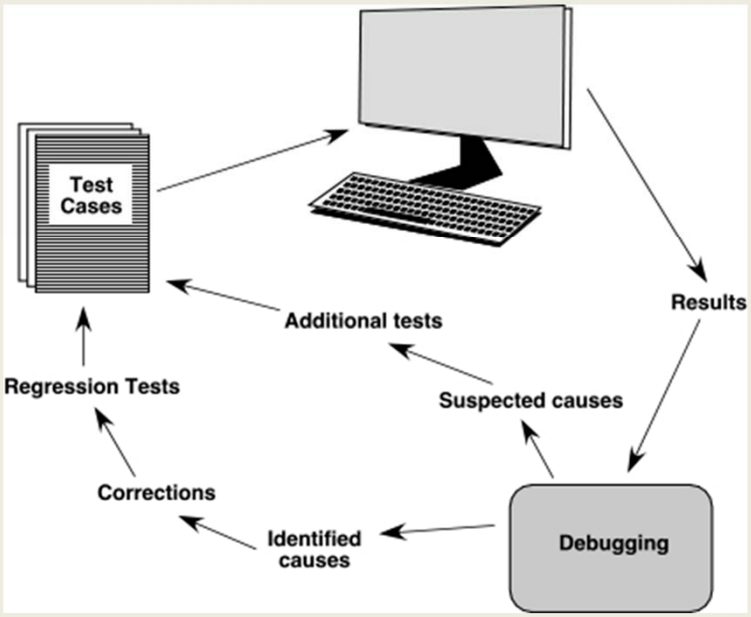


31

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.

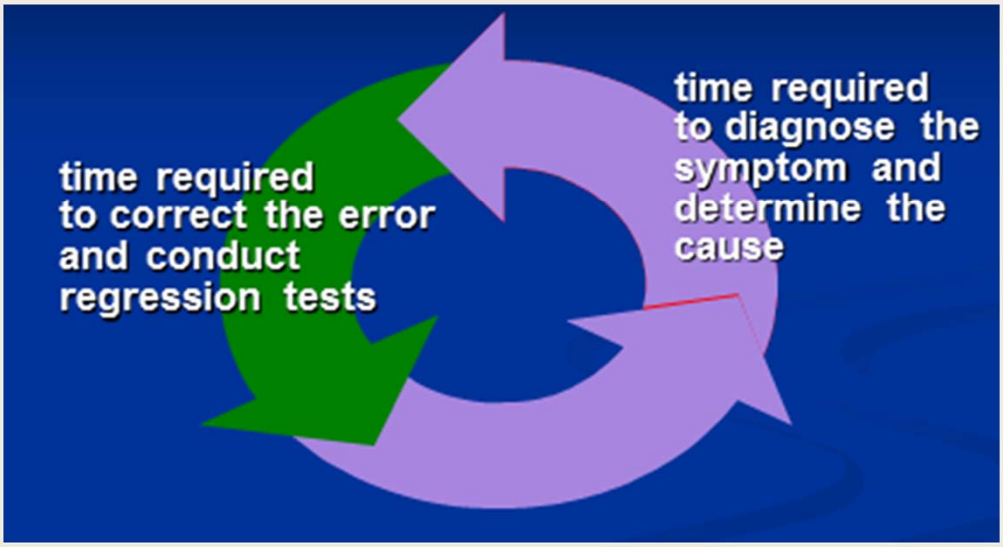
The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

The Debugging Process

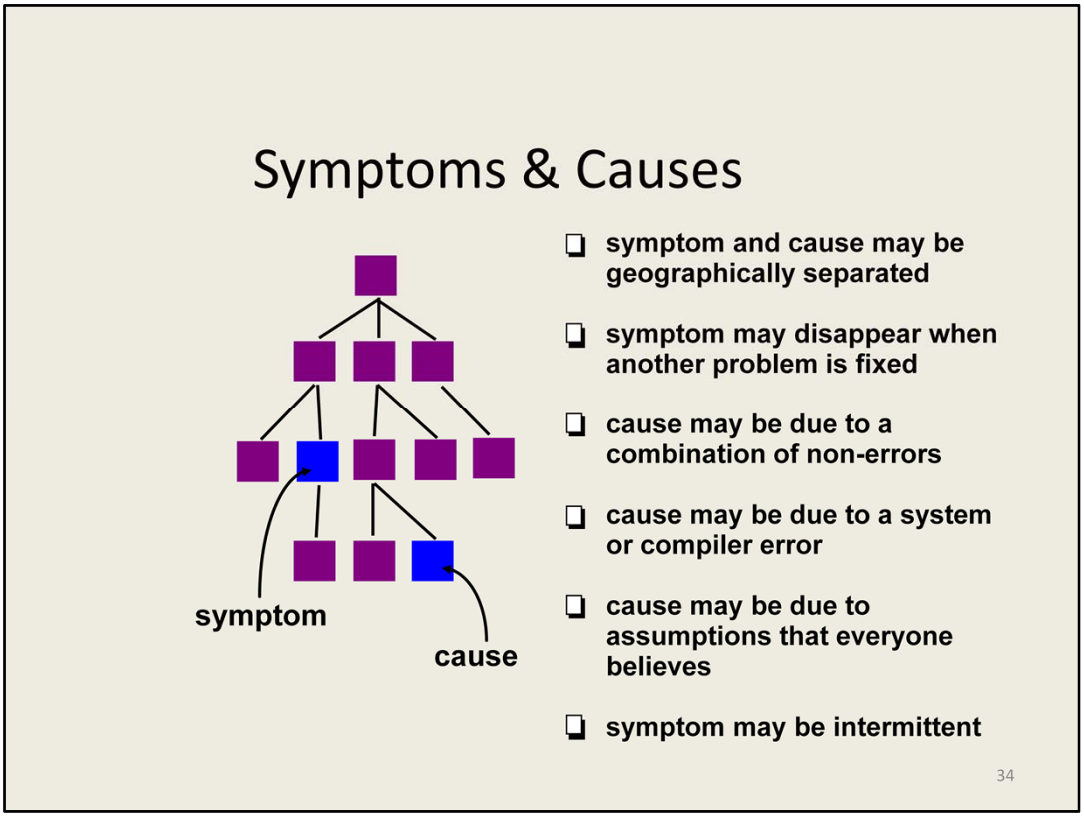


32

Debugging Effort



33



Why is debugging so difficult? In all likelihood, human psychology (see Section 17.8.2) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 8) exacerbate this situation.

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).

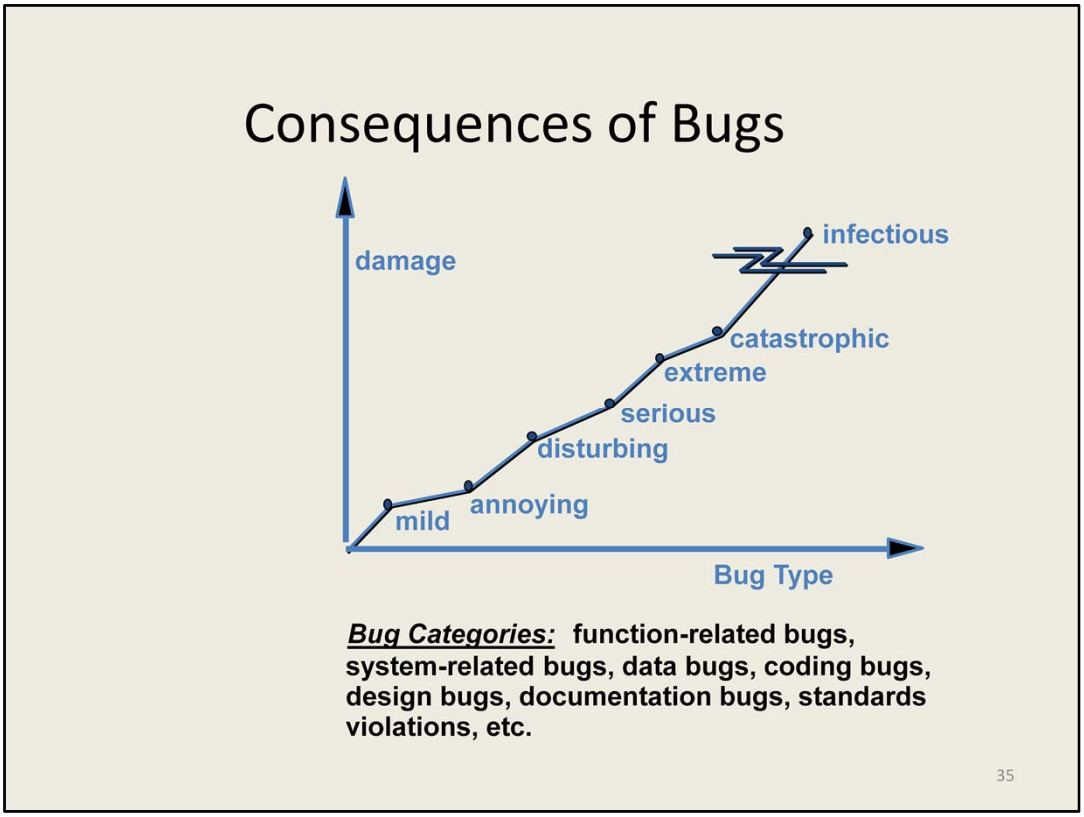
4. The symptom may be caused by human error that is not easily traced.

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. This is particularly common in embedded systems that couple hardware and software inextricably.

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



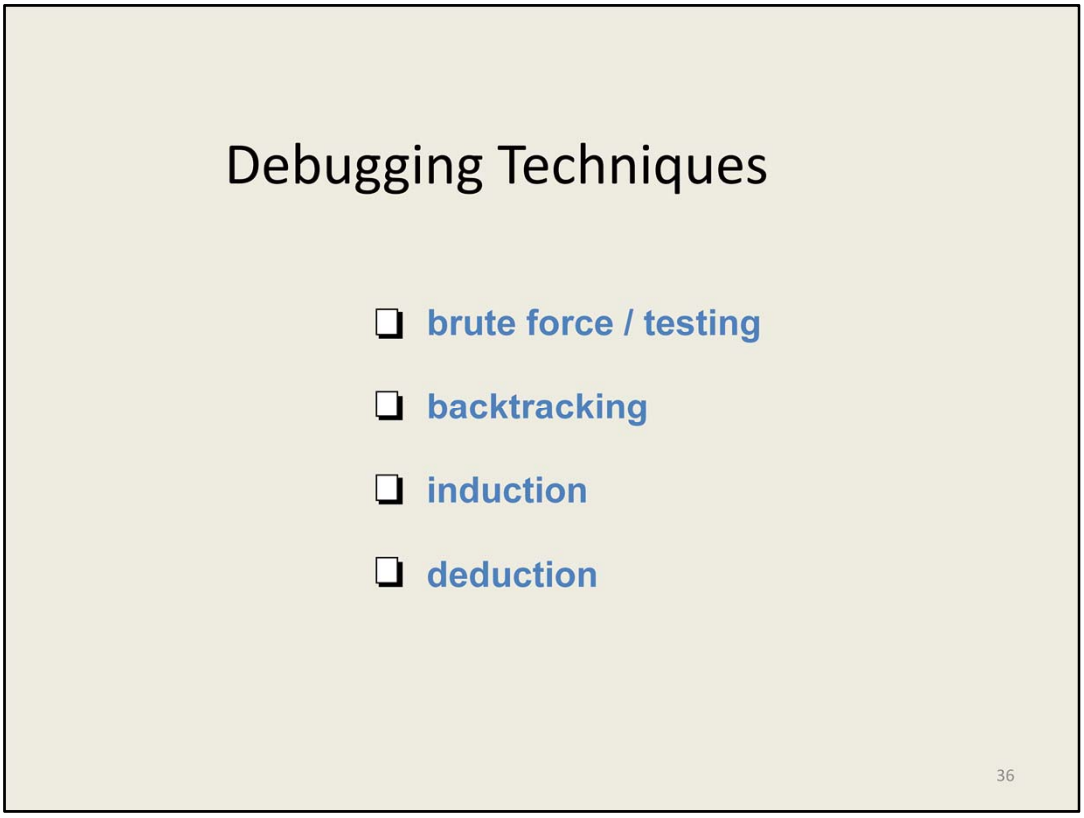
Debugging is one of the more frustrating parts of programming. It has elements of problem

solving or brain teasers, coupled with the annoying recognition that you have made

a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors

increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of

tension when the bug is ultimately . . . corrected.



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 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. Thought must be expended first!

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.

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.

Correcting the Error

- *Is the cause of the bug reproduced in another part of the program?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
- *What "next bug" might be introduced by the fix I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
- *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

37

Once a bug has been found, it must be corrected. But, as we have already noted, the

correction of a bug can introduce other errors and therefore do more harm than

good. Van Vleck [Van89] suggests three simple questions that you should ask before

making the “correction” that removes the cause of a bug:

Final Thoughts

- *Think* -- before you act to correct
- Use tools to gain additional insight
- If you're at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects

Summary

- Software testing usually accounts for a large share of technical efforts in software process.
- Software shall start with 'testing-in-the-small' and move toward 'testing-in-the-large'
  - Unit testing, integration testing, validation testing, and system testing
- Testing and quality assurance in general shall be planned ahead and start early in the process.
- In the Object-Oriented context, unit testing and integration testing take different meanings and techniques from their traditional approaches.
- Debugging must track down the course of an error.
  - Regression test suite shall be extended to prevent the same type of bugs from re-occurring.