UNIT - IV

Software Construction

Topics

- Software Construction
- Coding Standards
- Coding Framework
- Reviews – Desk Checks (Peer Reviews)
- Walkthroughs
- Code Reviews, Inspections
- Coding Methods
- Structured Programming
- Object-Oriented Programming
- Automatic Code Generation
- Software Code Reuse
- Pair Programming
- Test-Driven Development
- Configuration Management
- Software Construction Artefacts

Software Construction

Introduction

- A layman believes that software construction is the entire software development process.
- But, in fact, it is just one of the crucial tasks in software development; software requirement management, software design, software testing and software deployment are all equally crucial tasks.
- Furthermore, the process of software construction itself consists of many tasks; it not only includes software coding but also unit testing, integration testing, reviews and analysis.
- Construction is one of the most labor intensive phases in the software development life cycle.
- It comprises 30% or more of the total effort in software development.
- What a user sees as the product at the end of the software development life cycle is merely the result of the software code that was written during software construction.
- Due to the labor intensive nature of the software construction phase, the work is divided not only among developers, but also small teams are formed to work on parts of the software build.

Software Construction

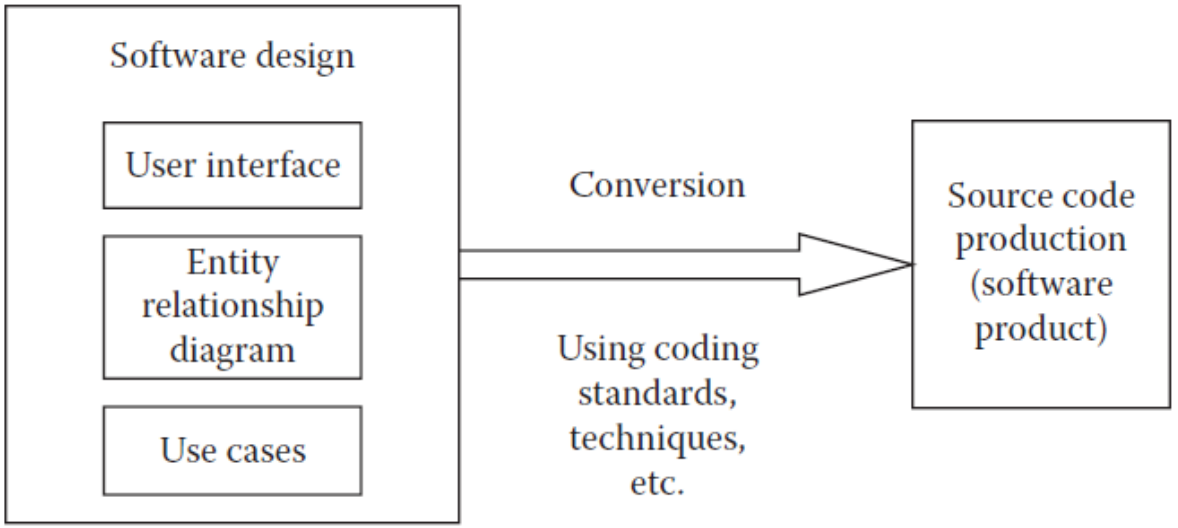
Introduction

- In fact, to **shrink the construction time**, many distributed teams, either internal or through contractors are deployed.
- The advantage to this is that these project teams do the software coding and other construction **work in parallel with each** other and thus the construction phase can be collapsed.
- This parallel development is known as **concurrent engineering**.
- Constructing an industry strength software product of a large size requires stringent coding standards.
- The whole process of construction should follow a proven process so that the produced code is maintainable, testable and reliable.
- The process itself should be efficient so that resource utilization can be optimized and thus cost of construction can be kept at a minimum.

Software Construction

Coding Standards

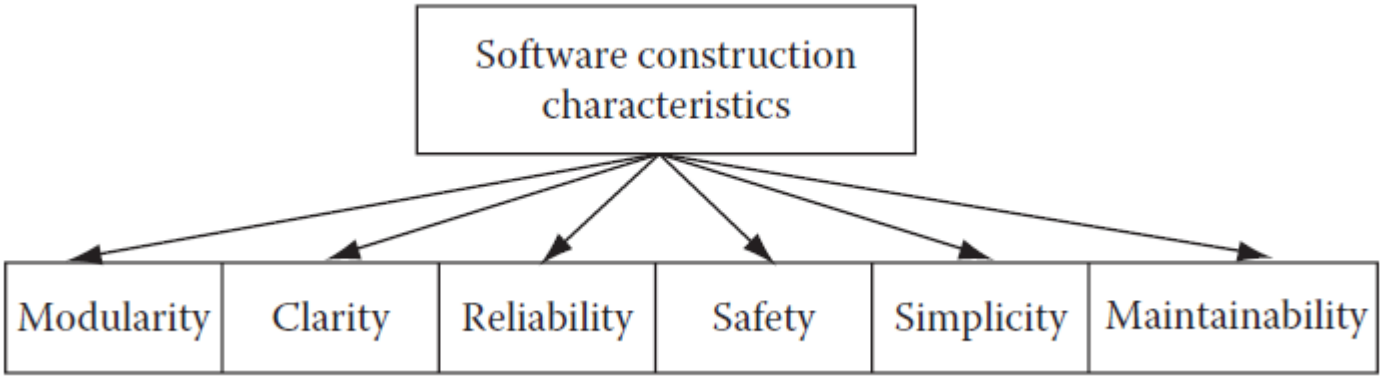
- Developers are given software design specifications in the form of use cases, flow diagrams, UI mock ups, etc., and they are supposed to write a code so that the built software matches these specifications.
- Converting the specifications into software code is totally dependent on the construction team.
- How well they do it depends on their experience, skills and the process they follow to do their job.
- Apart from these facilities, they also need some standards in their coding so that the work is fast as well as has other benefits like maintainability, readability and reusability (Figure-Source Code Production (Conversion) from Software Design).



Software Construction

Coding Standards

- At any time, a code written by a developer will always be different from that written by any other developer.
- This poses a challenge in terms of comprehending the code while reusing the code, maintaining it, or simply reviewing it.
- A uniform coding standard across all construction teams working on the same project will make sure that these issues can be minimized if not eliminated (Figure below - Software Construction Characteristics).
- Some of the coding standards include standards for code modularity, clarity, simplicity, reliability, safety and maintainability.



Software Construction

Coding Standards – Modularity

- The produced software code should be modular in nature.
- Each major function should be contained inside a software code module.
- The module should contain not only structure, but it should also process data.
- Each time a particular functionality is needed in the software construction, it can be implemented using that particular module of software code.
- This increases software code reuse and thus enhances productivity of developers and code readability.

Coding Standards – Clarity

- The produced code **should be clear for any person who would read the source code.**
- Standard **naming conventions** should be used so that the code has ample clarity.
- There should be **sufficient documentation** inside the code block, so that anybody reading the code could understand what a piece of code is supposed to do.
- There should also be **ample white spaces** in the code blocks, so that no piece of code should look crammed. White spaces enhance readability of written code.

Software Construction

Coding Standards – Simplicity

- The source code should have simplicity and **no unnecessary complex logic**; improvisation should be involved, if the same functionality can be achieved by a simpler piece of source code.
- Simplicity makes the code readable and will help in removing any defects found in the source code.
- Simplicity of written code can be enhanced by adopting best practices for many programming paradigms.
- For instance, in the case of object-oriented programming, abstraction and information hiding add a great degree of simplicity.
- Similarly, breaking the product to be developed into meaningful pieces that mimic real life parts makes the software product simple.

Coding Standards – Reliability

- Reliability is one of the most important aspects of industry strength software products.
- If the software product is not reliable and contains critical defects, then it will not be of much use for end users.

Software Construction

Coding Standards – Reliability

- Reliability of source code can be **increased by sticking to the standard processes** for software construction.
- During reviews, if any defects are found, they can be fixed easily if the source code is neat, simple, and clear.
- Reliable source code can be achieved by **first designing the software product with future enhancement in consideration** as well as by having a solid structure on which the software product is to be built.
- When writing pieces of source code based on this structure, there will be little chance of defects entering into the source code.
- Generally during enhancements, the existing structure is not able to take load of additional source code and thus the structure becomes shaky.
- If the development team feels that this is the case, then it is far better to restructure the software design and then write a code based on the new structure than to add a spaghetti code on top of a crumbling structure.

Software Construction

Coding Standards – Safety

- Safety is important, considering that software products are used by many industries where human lives are concerned and that human lives could be in danger because of faulty machine operation or exposure to a harmful environment.
- In these industries, the software product must be ensured to operate correctly and chances of error are less than 0.00001%.
- Industries like **medicine and healthcare, road safety**, hazardous material handling need foolproof software products to ensure that either human lives are saved (in case of medicine and healthcare) or human lives are not in danger.
- Here the software code must have inbuilt safety harnesses.

Coding Standards – Maintainability

- As it has been pointed out after several studies, maintenance costs are more than 70% of all costs including software development, implementation, and maintenance.
- To make sure that maintenance costs are under limit during software construction, it should be made sure that the source code is maintainable.
- It will be easy to change the source code for fixing defects during maintenance.

Software Construction

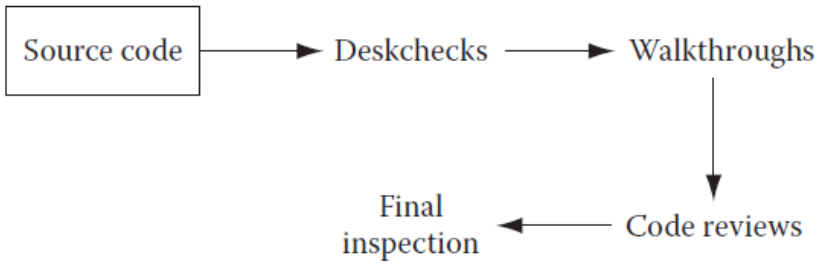
Coding Framework

- Like most construction work, you need to set up an infrastructure based on which construction can take place.
- For software construction, you need to have a coding framework that will ensure a consistent coding production with standard code that will be easy to debug and test.
- In object oriented programming, what base classes are to be made, which will be used throughout construction, is a subject that is part of the coding framework.
- In general, coding frameworks allow construction of the common infrastructure of basic functionality which can be extended later by the developers.
- This way of working increases productivity and allows for a robust and well structured software product.
- It is similar in approach to house building where a structure is built based on a solid foundation.

Software Construction

Reviews (Quality Control)

- It is estimated that almost 70% of software defects arise from faulty software code.
- To compound this problem, software construction is the most labor intensive phase in software development.
- Any construction rework means wasting a lot of effort already put in.
- Moreover, it is also a fact that it is cheaper to fix any defects found during construction at the phase level itself.
- If those defects are allowed to go in software testing (which is the next phase), then fixing those defects will become costlier.
- That is why review of the software code and fixing defects is very important.
- There are some techniques available like deskchecks, walkthroughs, code reviews, inspections, etc. that ensure quality of the written code (Figure below- Source code review methods and their operation sequence).



Software Construction

Reviews (Quality Control)

- These different kinds of reviews are done at different stages in software code writing.
- They also serve different purposes.
- While inspections provide the final go/no go decision for approval of a piece of code, other methods are less formal and are meant for removing defects instead of deciding whether a piece of code is good enough or not.

Reviews – Deskchecks (Peer Reviews)

- Deskchecks are employed when a complete review of the source code is not important.
- Here, the developer sends his piece of **code to the designated team members**.
- These team members review the code and send feedback and comments to the developer as suggestions for improvement in the code.
- The developer reads those feedbacks and may decide to incorporate or to discard those suggestions.
- So this form of review is totally **voluntary**.
- Still, it is a powerful tool to eliminate defects or improve software code.

Software Construction

Reviews – Walkthroughs

- Walkthroughs are formal code reviews initiated by the developer. The developer sends an invitation for **walkthrough to team members**.
- At the meeting, the **developer presents his method of coding** and walks through his piece of code.
- The team members then make **suggestions for improvement**, if any.
- The **developer then can decide to incorporate** those suggestions or discard them.

Reviews – Code Reviews

- Code reviews are one of the **most formal methods of reviews**. The project manager calls for a meeting for code review of a developer.
- At the meeting, team members review the code and point out any code errors, defects, or improper code logic for likely defects. An **error log is also generated and is reviewed by the entire team.**

Reviews – Inspections

- Code inspections are **final reviews of software code** in which it is decided whether to pass a piece of code for inclusion into the main software build.

Software Construction

Coding Methods

- Converting design into optimal software construction is a very serious topic that has generated tremendous interest over the years.
- Many programming and coding methods were devised and evolved as a result.
- As it is well known in the industry, the early software products were of small size due to limited hardware capacity.
- With increasing hardware capacity, the size of software products has been increasing.
- Software product size affects the methods that can be used to construct specific sized software products.
- Advancement in the field of computer science also allows discovery of better construction methods.
- To address needs of different sized software products in tandem with advancement in computer science, different programming techniques evolved.
- These include **structured programming, object-oriented programming, automatic code generation, test-driven development, pair programming**, etc.

Software Construction

Coding Methods – Structured Programming

- Structured programming evolved after mainframe computers became popular.
- Mainframe computers offered vast availability of computing power compared to primitive computers that existed before.
- Using structured programming, large programs could be constructed that could be used for making large commercial and business applications.
- Structured programming enabled programmers to store large pieces of code inside procedures and functions.
- These pieces of code could be called by any other procedures or functions.
- This enabled programmers to structure their code in an efficient way.
- Code stored inside procedures could be reused anywhere in the application by calling it.

Software Construction

Coding Methods – Object-Oriented Programming

- In structured programming, data and structured code are separate and accordingly they are modeled separately.
- This is an unnatural way of converting real life objects into software code because objects contain both data and structure.
- Widely used as an example in object-oriented programming books, a car consists of a chassis, an engine, four wheels, body, and transmission.
- Each of these objects has some specific properties and specific functions.
- When a software system is modeled to represent real-world objects, both data and structure are taken care of in object-oriented programming.
- From outside of a class that is made to represent an object, only the behavior of the object is visible or perceived.
- Unnecessary details about the object are hidden and in fact are not available from outside.
- This kind of representation of objects makes them robust and a system built on using them has relatively few problems.

Software Construction

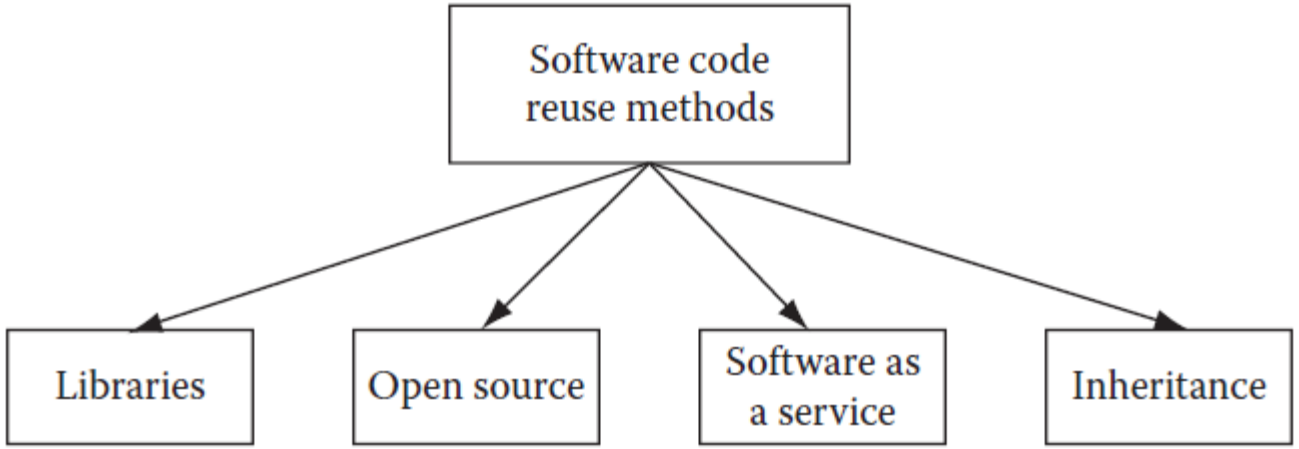
Coding Methods – Automatic Code Generation

- Constructing and generating software code is very labor intensive work. So there has always been fascination about automatic generation of software code.
- Unfortunately, this is still a dream. Some CASE and modeling tools are available that generate software code. But they are not sophisticated. They are also not complete.
- Then there are business analyst platforms developed by many ERP software vendors that generate code automatically when analysts configure the product.
- These analyst platforms are first built using any of the software product development methodologies.
- The generated code is specific to the platform and runs on the device (hardware and software environment) for which the code is generated.
- Generally, any code consists of many construction unit types.
- Some of these code types include control statements such as loop statements, if statements, etc., and database access, etc.
- Generating all of the software code required to build a software application is still difficult.
- But some companies like Sun Microsystems are working to develop such a system.

Software Construction

Coding Methods – Software Code Reuse

- Many techniques have evolved to reduce the labor intensive nature of writing source code.
- Software code reuse is one such technique.
- Making a block of source code to create a functionality or general utility library and using it at all places in the source code wherever this kind of functionality or utility is required is an example of code reuse.
- Code reuse in procedural programming techniques is achieved by creating special functions and utility libraries then using them in the source code.
- In object-oriented programming, code reuse is done at a more advanced level.



Software Construction

Coding Methods – Software Code Reuse

- The classes containing functions and data themselves can not only be reused in the same way as functions and libraries but the classes can also be modified by way of creating child classes and using them in the source code (Figure above – Code reuse methods).
- Apart from creating and using libraries and general purpose classes for code reuse, a more potent code reuse source has evolved recently.
- It is known as “service oriented architecture” (SOA).

Software Construction

Coding Methods – Pair Programming

- Pair programming is a quality driven development technique employed in the eXtreme Programming development model.
- Here, each development task is assigned to two developers.
- While one developer writes the code, the other developer sits behind him and guides him through the requirements (functional, nonfunctional).
- When it is the turn of the other developer to write the code, the first developer sits behind him and guides him on the requirements.
- So developers take turns for the coding and coaching work.
- This makes sure that each developer understands the big picture and helps them to write better code with lesser defects.

Software Construction

Coding Methods – Test-Driven Development

- This concept is used with iteration-based projects especially with eXtreme Programming technique.
- Before developers start writing source code, they create test cases and run the tests to see if they run properly and their logic is working.
- Once it is proved that their logic is perfect, only then they write the source code.
- So here, tests drive software development, and hence it is appropriately named test-driven development.

Testing Strategies

Strategic approach to software testing

- Generic characteristics of strategic software testing:
  - To perform effective testing, a software team should conduct effective *formal technical reviews.* By doing this, many errors will be eliminated before testing start.
  - Testing begins at the *component level* and works "*outward"* toward the integration of the entire computer-based system.
  - Different *testing techniques* are appropriate 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

- 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 activities that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
- State another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?“
- The definition of V&V encompasses many of the activities that are similar to *software quality assurance* (SQA).

- V&V encompasses a wide array of **SQA(sw quality assurance) activities** that include
  - Formal technical reviews,
  - quality and configuration audits,
  - performance monitoring,
  - simulation,
  - feasibility study,
  - documentation review,
  - database review,
  - algorithm analysis,
  - development testing,
  - qualification testing, and installation testing
- Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered.
- Quality is not measure only by no. of error but it is also measure on application methods, process model, tool, formal technical review, etc will lead to quality, that is confirmed during testing.

Verification and Validation

- Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase .
  - Does the product meet its specifications?
  - Are we building the **product right**?
- Methods of Verification : Static Testing
  - Walkthrough
  - Inspection
  - Review

Verification and Validation

- Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements
- Does the product perform as desired?
- Are we building the **right product**
- Methods of Validation : Dynamic Testing
  - Testing

Example of verification and validation

- In Software Engineering, consider the following specification

A clickable button with name Submet

- Verification would check the design doc and correcting the spelling mistake.
- Otherwise, the development team will create a button like

Example for Verification and Validation

A clickable button with name Submit

- Once the code is ready, Validation is done. A Validation test found –

Button NOT Clickable

- Owing to Validation testing, the development team will make the submit button clickable

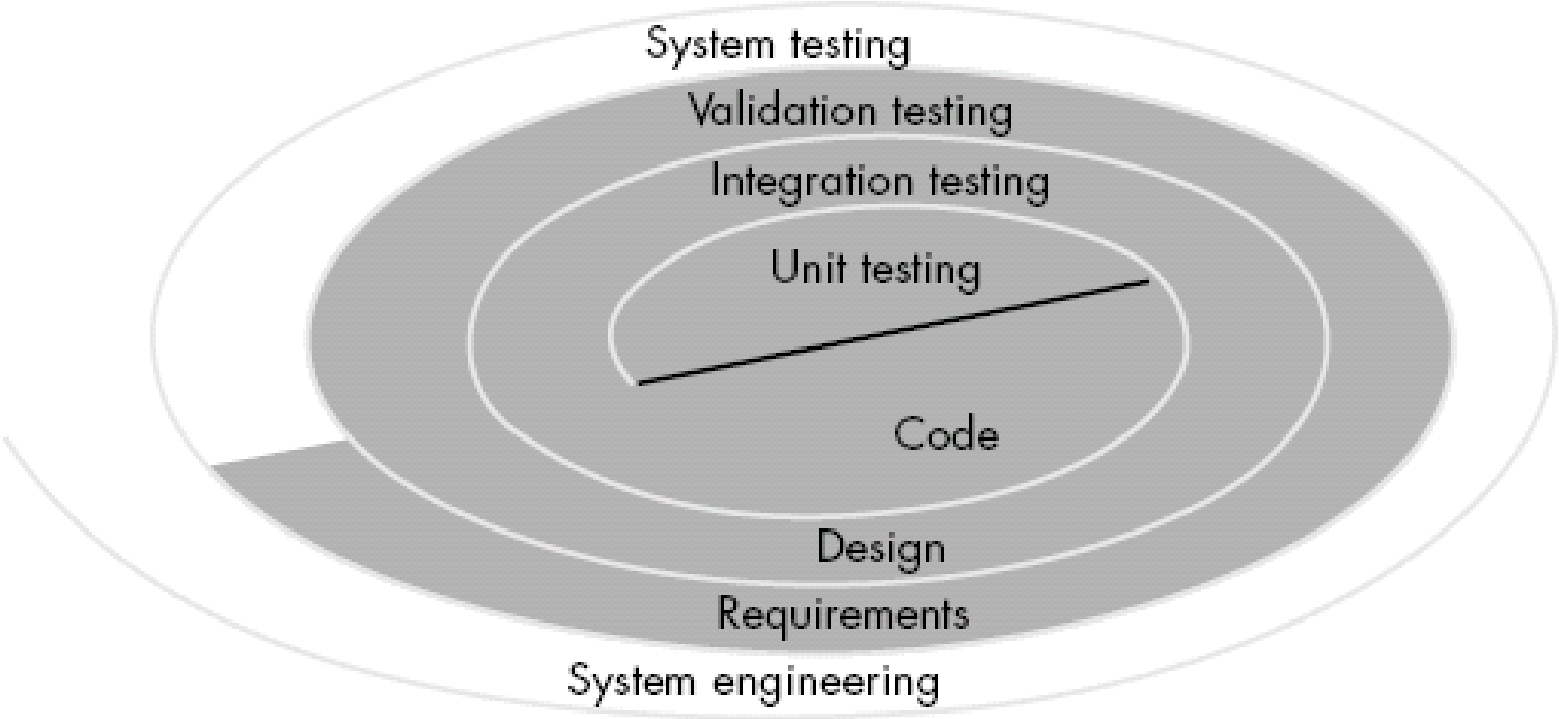
Difference between Verification and Validation

1. **Verification** is a static practice of verifying documents, design, code and program.	1. **Validation** is a dynamic mechanism of validating and testing the actual product.
2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	*3.* It is computer based execution of program.
4. **Verification** uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	*4.* **Validation** uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.

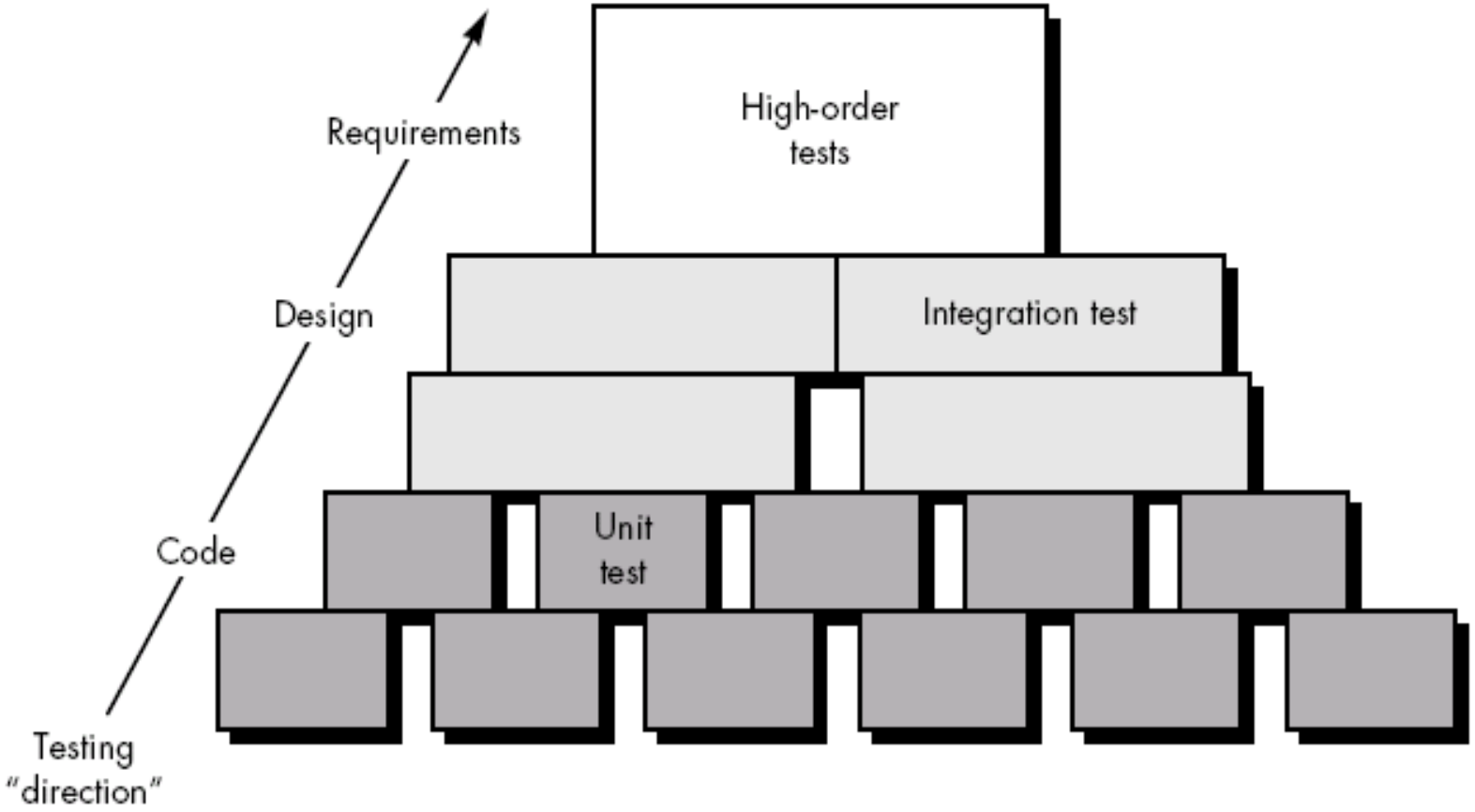
Difference between Verification and Validation

5. **Verification** is to check whether the software conforms to specifications.	5. **Validation** is to check whether software meets the customer expectations and requirements.
6. It can catch errors that validation cannot catch. It is low level exercise.	*6. It can catch errors that verification cannot catch. It is High Level Exercise.*
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	*7.* Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. **Verification** is done by QA team to ensure that the software is as per the specifications in the SRS document.	*8.* ***Validation*** *is carried out with the involvement of testing team.*
9. It generally comes first-done before validation.	*9.* It generally follows after **verification**.

Software Testing Strategy for conventional software architecture



- A *Software process* & *strategy for software testing* may also be viewed in the context of the *spiral.*
- *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software.
- Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction.
- Another turn outward on the spiral, we encounter *validation testing,* where requirements established as part of software requirements analysis are validated against the software.
- Finally, we arrive at *system testing,* where the software and other system elements are tested as a whole.



- Software process from a procedural point of view; a series of four steps that are implemented sequentially.

- Initially, tests focus on each component individually, ensuring that it functions properly as a unit.
- Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure.
- Integration testing addresses the issues associated with the dual problems of verification and program construction.
- Black-box test case design techniques are the most prevalent during integration.
- Now, validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
- Black-box testing techniques are used exclusively during validation.
- 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.

Criteria for Completion of Testing

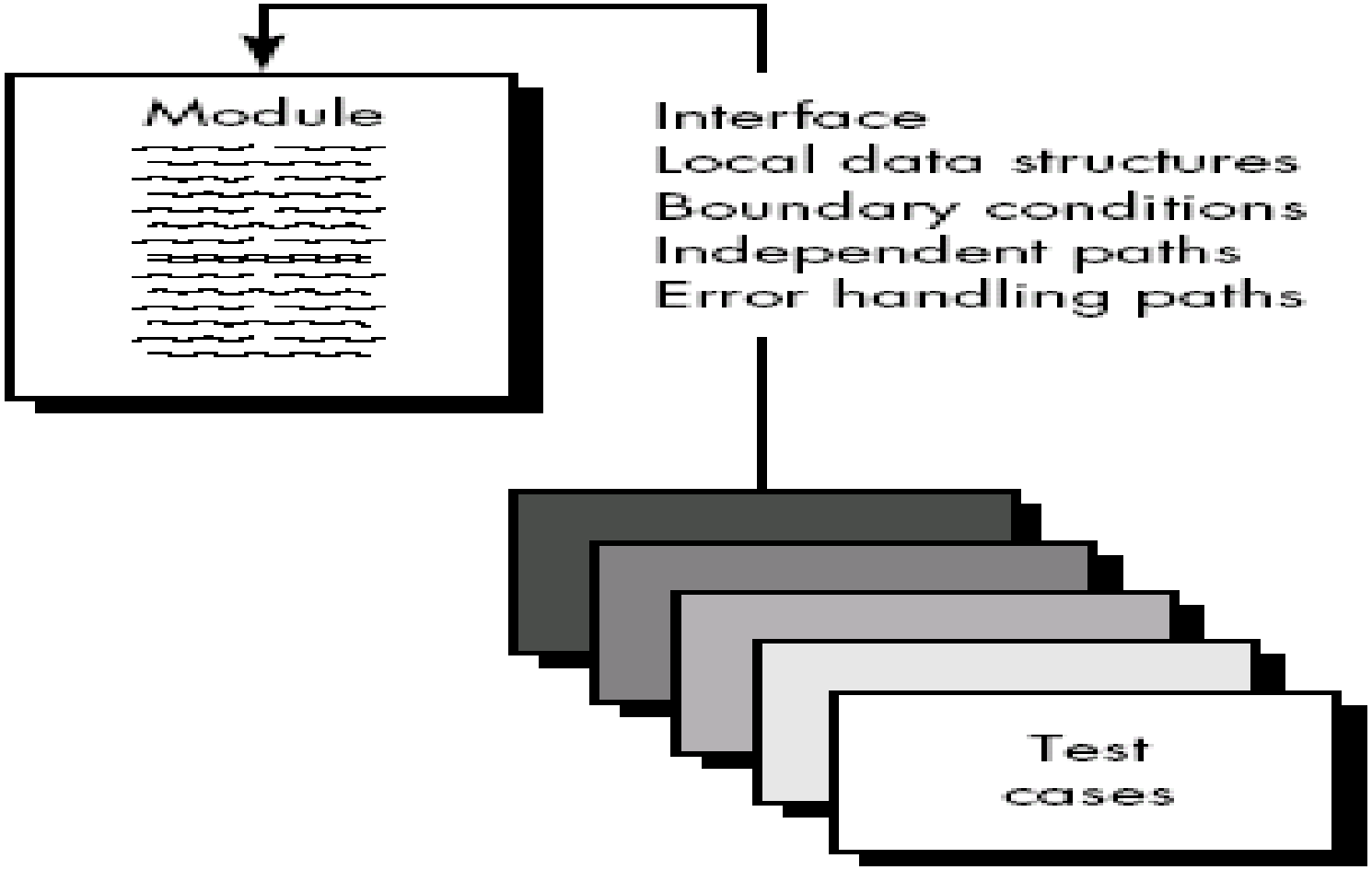
- There is no definitive answer to state that “we have done with testing”.
- One response to the question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/ user executes a computer program, the program is being tested.
- Another response is: "You're done testing when you run out of *time* (deadline to deliver product to customer) or you run out of *money (spend so much money on testing)*.

- But few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted.
- Response that is based on statistical criteria: "No, we cannot be absolutely predict that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that program will not fail.

Unit testing strategies for conventional software

- Focuses verification effort on the smallest unit of software design – 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.
- Unit test is white-box oriented, and the step can be conducted in parallel for multiple components.
- Unit test consists of
  - Unit Test Considerations
  - Unit Test Procedures

Unit Test Considerations



Contd.

- *Module interface* - information properly flows into and out of the program unit under test.
- *local data structure* - data stored temporarily maintains its integrity.
- *Boundary conditions* -module operates properly at boundaries established to limit or restrict processing
- Independent paths - all statements in a module have been executed at least once.
- And finally, all *error handling paths* are tested.

- *Module interface* are required before any other test is initiated because If data do not enter and exit properly, all other tests are debatable.
- In addition, *local data structures* should be exercised and the local impact on global data should be discover 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
  - Computations,
  - Incorrect comparisons, or
  - Improper control flow
- *Basis path* and loop testing are effective techniques for uncovering a broad array of *path errors.*

Errors are commonly found during unit testing

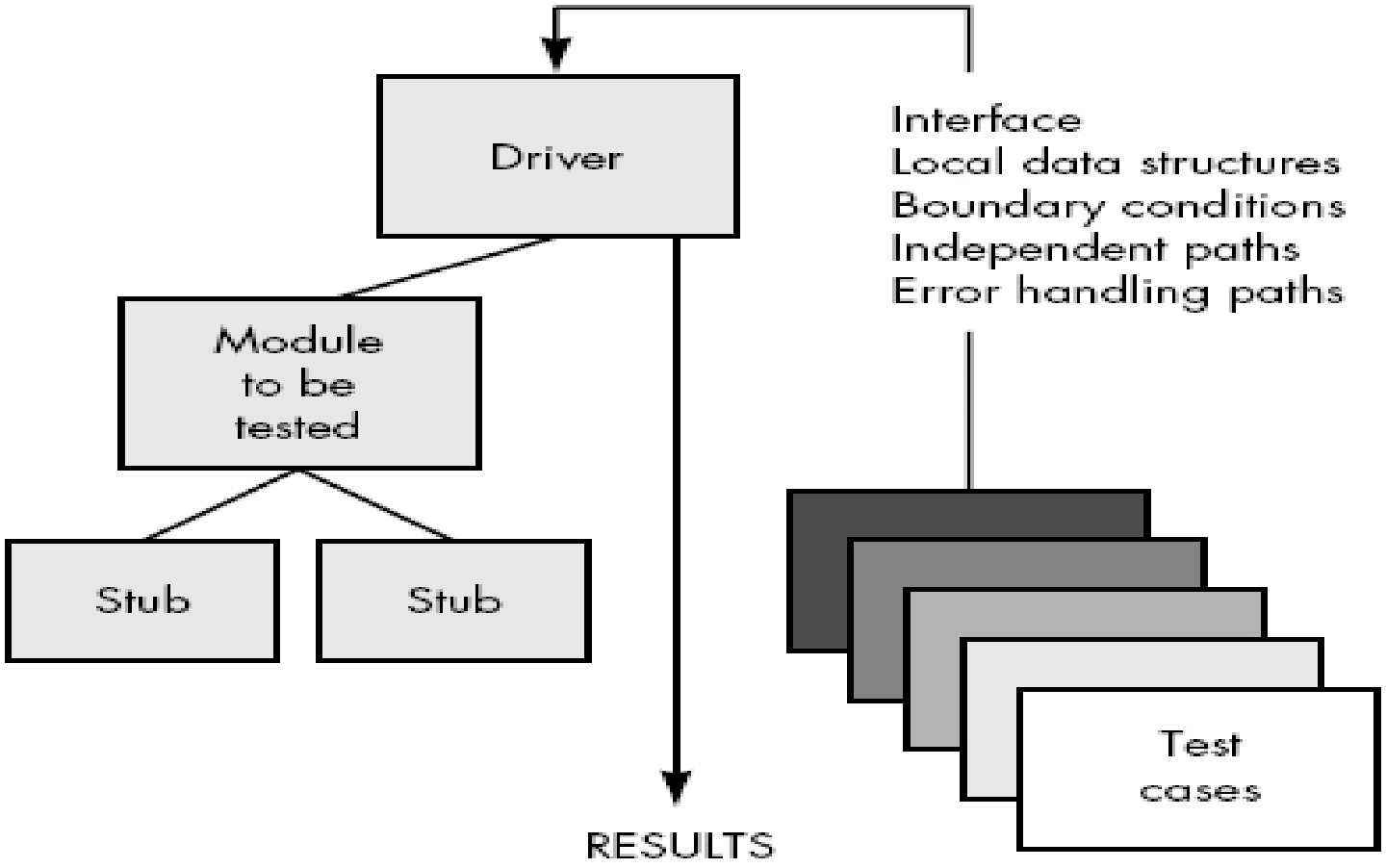
- More common errors in computation are
  - misunderstood or incorrect arithmetic precedence
  - mixed mode operations,
  - incorrect initialization,
  - precision inaccuracy,
  - incorrect symbolic representation of an expression.
- Comparison and control flow are closely coupled to one another
  - Comparison of different data types,
  - Incorrect logical operators or precedence,
  - Incorrect comparison of variables
  - Improper or nonexistent loop termination,
  - Failure to exit when divergent iteration is encountered
  - improperly modified loop variables.

- Potential errors that should be tested when error handling is evaluated are
  - Error description is unintelligible.
  - Error noted does not correspond to error encountered.
  - Error condition causes system intervention prior to error handling.
  - Exception-condition processing is incorrect.
  - Error description does not provide enough information to assist in the location of the cause of the error.
- Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed or when the maximum or minimum allowable value is encountered.
- So BVA test is always be a last task for unit test.
- 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.

Unit Test Procedures

- Perform before coding or after source code has been generated.
- A review of design information provides guidance for establishing test cases. 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 be developed for each unit test.
- 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.
- 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.
- *Stubs* serve to replace modules that are subordinate the component to be tested.

Unit Test Procedures



Unit Test Environment

- Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product.
- In such cases, complete testing can be postponed until the integration test step
- Unit testing is simplified when a component with high cohesion is designed.
- When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

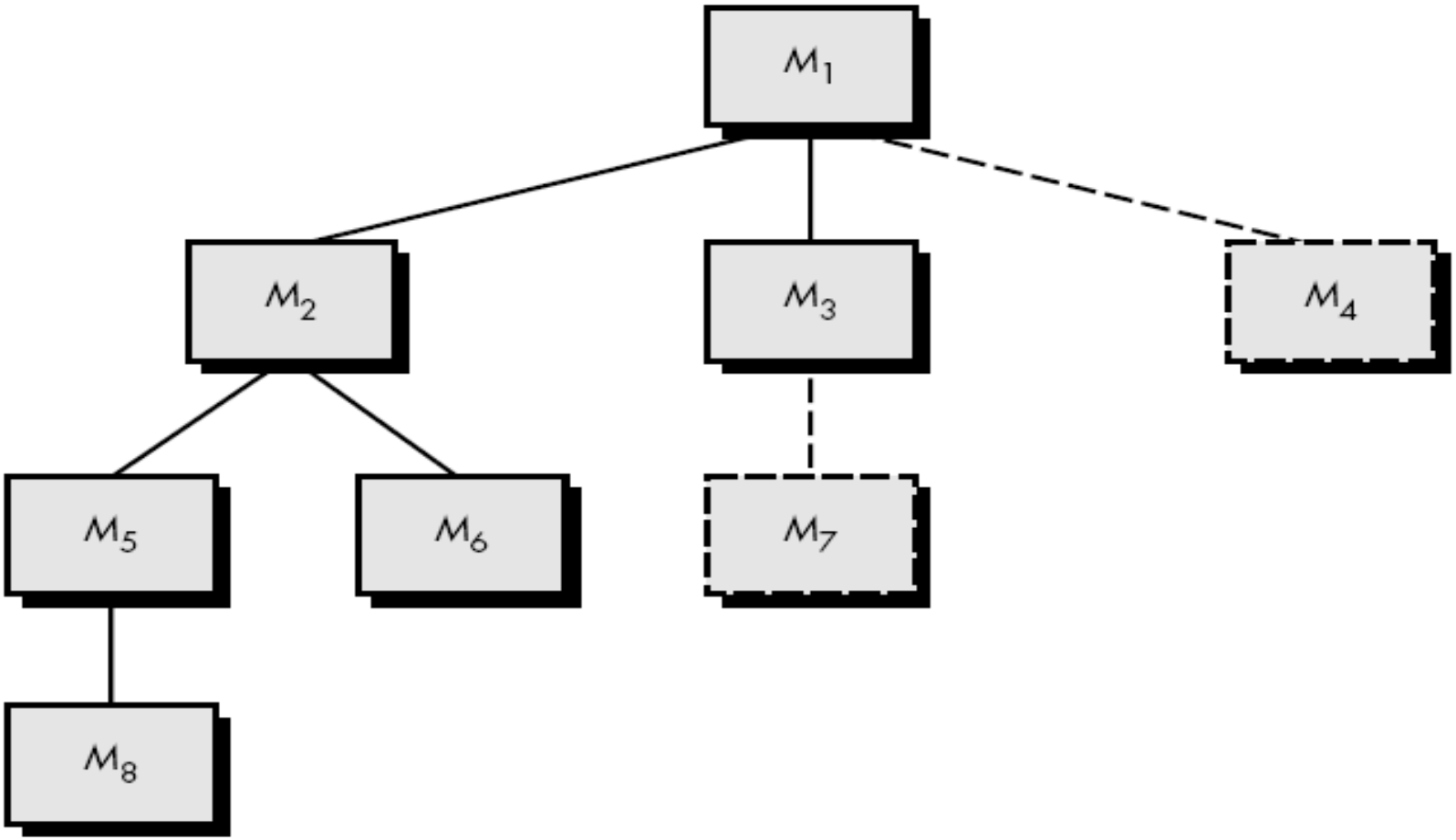
Integration testing

- Integration testing is a systematic technique for constructing the program structure 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.
- There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach.
- A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.
- Incremental integration is the exact opposite of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct;

Top-down Integration

- *Top-down integration testing is an incremental approach to construction of program* structure.
- Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.
- *Depth-first integration would integrate all components on* a major control path of the 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 M6 would be integrated
  - The central and right hand control paths are built.

Top down integration



- *Breadth-first integration incorporates all components* directly subordinate at each level, moving across the structure horizontally.
- Step would be:
  - components M2, M3, and M4 would be integrated first
  - next control level, M5, M6, and so on follows.

Top-down Integration process 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 may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

Problem occur in top-down integration

- Logistic problems can arise
- most common problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels.
- No significant data can flow upward in the program structure due to stubs replace low level modules at the beginning of top-down testing. In this case, Tester will have 3 choice
  - Delay many tests until stubs are replaced with actual modules
  - develop stubs that perform limited functions that simulate the actual module
  - integrate the software from the bottom of the hierarchy upward

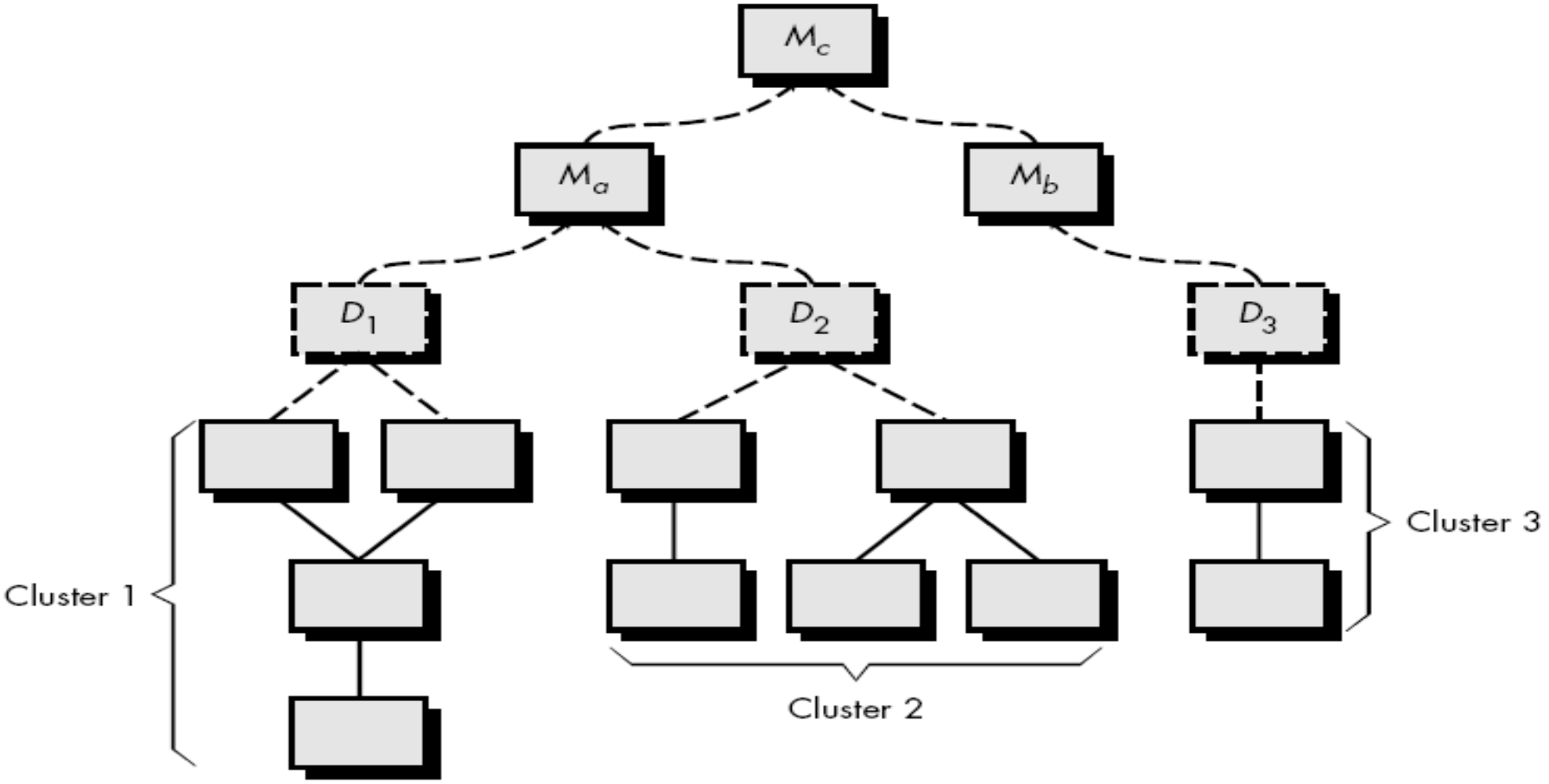
Bottom-up Integration

- *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, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

Bottom up integration process steps

- Low-level components are combined into clusters (sometimes called *builds)* that perform a specific software sub function.
- 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.

Bottom up integration



Example

- Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver.
- Components in clusters 1 and 2 are subordinate to Ma.
- Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb.
- Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

Regression Testing

- Each time a new module is added as part of integration testing
  - New data flow paths are established
  - New I/O may occur
  - New control logic is invoked
- Due to these changes, may cause problems with functions that previously worked flawlessly.
- ***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.

Contd.

- Regression testing is the activity that helps to ensure that changes 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.*
- Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.
- Regression testing contains 3 diff. 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.

Contd.

- As integration testing proceeds, the number of regression tests can grow quite large.
- 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.
- It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Smoke Testing

- *Smoke testing* is an integration testing approach that is commonly used when “shrink wrapped” software products are being developed.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis.

Smoke testing approach 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 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 is smoke tested daily.
  - The integration approach may be top down or bottom up.

Smoke Testing benefits

- *Integration risk is minimized.*
  - Smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early
- *The quality of the end-product is improved.*
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design defects. At the end, better product quality will result.
- *Error diagnosis and correction are simplified.*
  - Software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.*
  - Frequent tests give both managers and practitioners a realistic assessment of integration testing progress.

What is a critical module and why should we identify it?

- As integration testing is conducted, the tester should identify *critical modules.*
- A critical module has one or more of the following characteristics:
  - Addresses several software requirements,
  - Has a high level of control (Program structure)
  - Is complex or error prone
  - 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 Documentation

- An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification*
- It contains a test plan, and a test procedure, is a work product of the software process, and becomes part of the software configuration.
- The test plan describes the overall strategy for integration.
- Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software.
- Integration testing might be divided into the following test phases:
  - User interaction
  - Data manipulation and analysis
  - Display processing and generation
  - Database management

Contd.

- Therefore, groups of modules are created to correspond to each phase.
- The following criteria and corresponding tests are applied for all test phases:
- **Interface integrity-** Internal and external interfaces are tested as each module.
- **Functional validity -** Tests designed to uncover functional errors are conducted.
- **Information content -** associated with local or global data structures are conducted.
- **Performance -** to verify performance

Contd.

- A schedule for integration and related topics is also discussed as part of the test plan.
- Start and end dates for each phase are established
- A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort.
- Finally, test environment and resources are described.
- The order of integration and corresponding tests at each integration step are described.
- A listing of all test cases and expected results is also included.
- A history of actual test results, problems is recorded in the *Test Specification.*

Validation Testing

- *Validation testing* succeeds when software functions in a manner that can be *reasonably expected by the customer.*
- Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level— on things that will be immediately apparent to the end-user.
- Reasonable expectations are defined in the *Software Requirements Specification*— a document that describes all user-visible attributes of the software.
- Validation testing comprises of
  - Validation Test criteria
  - Configuration review
  - Alpha & Beta Testing

Validation Test criteria

- It is achieved through a series of tests that demonstrate agreement with requirements.
- A *test plan* outlines the classes of *tests to be conducted* and a *test procedure* defines specific test cases that will be used to demonstrate *agreement with requirements*.
- Both the plan and procedure are designed to ensure that
  - all functional requirements are satisfied,
  - all behavioral characteristics are achieved,
  - all performance requirements are attained,
  - documentation is correct,
  - other requirements are met
- After each validation test case has been conducted, one of two possible conditions exist:

1. The function or performance characteristics conform to specification and are accepted
2. A deviation from specification is uncovered and a deficiency list is created

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 the support phase of the software life cycle.
- The configuration review, sometimes called an *audit.*

Alpha and Beta Testing

- When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements.
- Conducted by the end-user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests.
- 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 testing

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

Beta testing

- The *beta test* is conducted at one or more customer sites by the end-user of the software.
- 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, software engineers make modifications and then prepare for release of the software product to the entire customer base.

System Testing

- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.
- Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.
- Types of system tests are:
  - Recovery Testing
  - Security Testing
  - Stress Testing
  - Performance 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, that is 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 break through .
- During security testing, the tester plays the role(s) of the individual who desires to break through the system.
- 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.
- The tester may attempt to acquire passwords through externally, may attack the system with custom software designed to breakdown any defenses that have been constructed; may browse through insecure data; may purposely cause system errors.

Stress Testing

- *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 ten interrupts per second
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 excessive hunting for disk-resident data are created

- A variation of stress testing is a technique called *sensitivity testing*

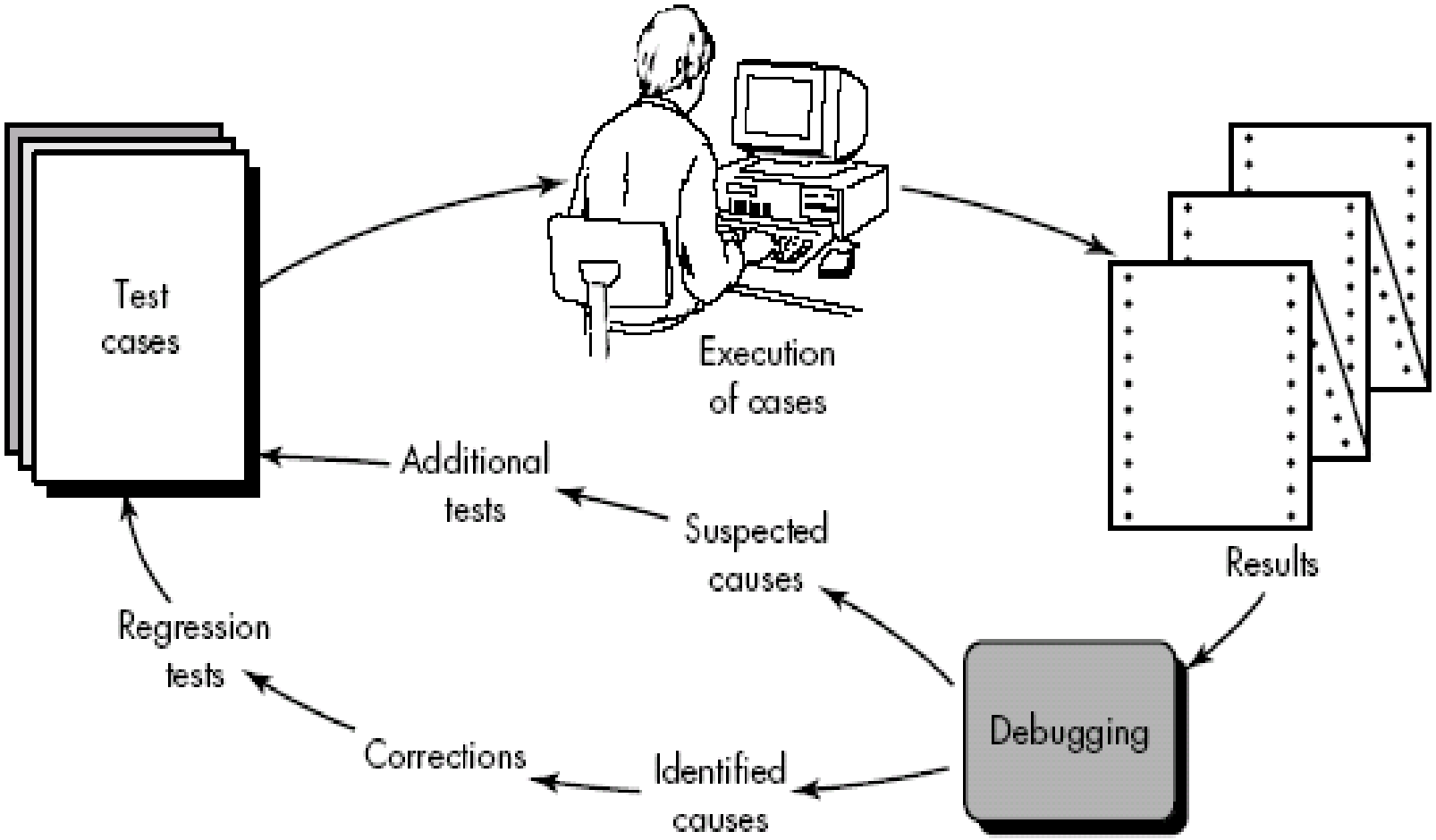
Performance Testing

- 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 white-box tests are conducted.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation
- It is often necessary to measure resource utilization (e.g., processor cycles).

THE ART OF DEBUGGING

- Debugging is the process that results in the removal of the error.
- Although debugging can and should be an orderly process, it is still very much an art.
- Debugging is not testing but always occurs as a consequence of testing.

Debugging Process



Debugging Process

- Results are examined and a lack of correspondence between expected and actual performance is encountered (due to cause of error).
- Debugging process attempts to match symptom with cause, thereby leading to error correction.
- One of two outcomes always comes from debugging process:
  - The cause will be found and corrected,
  - The cause will not be found.
- The person performing debugging may *suspect a cause*, design a test case to help validate that doubt, and work toward error correction in an iterative fashion.

Why is debugging so difficult?

1. The symptom may disappear (temporarily) when another error is corrected.
2. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
3. The symptom may be caused by human error that is not easily traced (e.g. wrong input, wrongly configure the system)
4. The symptom may be a result of timing problems, rather than processing problems.(e.g. taking so much time to display result).
5. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

6. The symptom may be intermittent (connection irregular or broken). This is particularly common in embedded systems that couple hardware and software
7. The symptom may be due to causes that are distributed across a number of tasks running on different processors

As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches or strategies

- Debugging has one overriding objective: to find and correct the cause of a software error.
- Three categories for debugging approaches
  - Brute force
  - Backtracking
  - Cause elimination

Brute Force:

- probably the most common and least efficient method for isolating the cause of a software error.
- 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 WRITE or PRINT statements
- It more frequently leads to wasted effort and time.

Backtracking:

- common debugging approach that can be used successfully in small programs.
- Beginning at the site where a symptom has been open, the source code is traced backward (manually) until the site of the cause is found.

Cause elimination

- Is cleared by induction or deduction and introduces the concept of binary partitioning (i.e. valid and invalid).
- A list of all possible causes is developed and tests are conducted to eliminate each.

Correcting the error

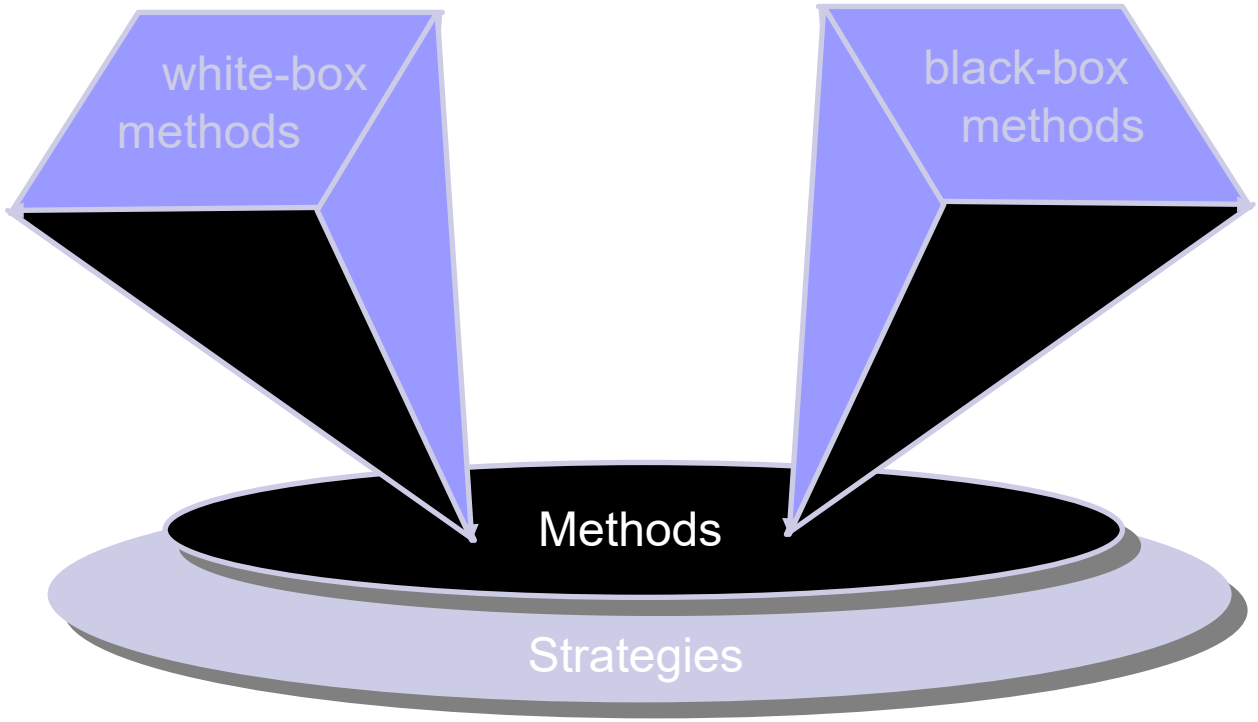
- The correction of a bug can introduce other errors and therefore do more harm than good.

Questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

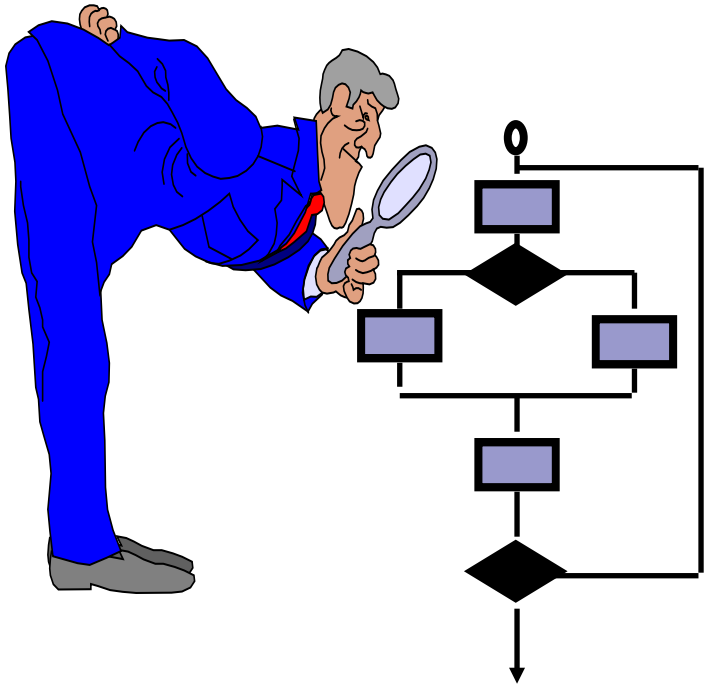
- **Is the cause of the bug reproduced in another part of the program?** (i.e. cause of bug is logical pattern)
- **What "next bug" might be introduced by the fix I'm about to make? (**i.e. cause of bug can be in logic or structure or design**).**
- **What could we have done to prevent this kind of bug previously? (** i.e. same kind of bug might generated early so developer can go through the steps**)**

Black box & White box Testing

Software Testing



White-Box Testing



... our goal is to ensure that all statements and conditions have been executed at least once ...

Why Cover?

- **logic errors and incorrect assumptions are inversely proportional to a path's execution probability**
- **we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive**
- **typographical errors are random; it's likely that untested paths will contain some**

White Box Testing

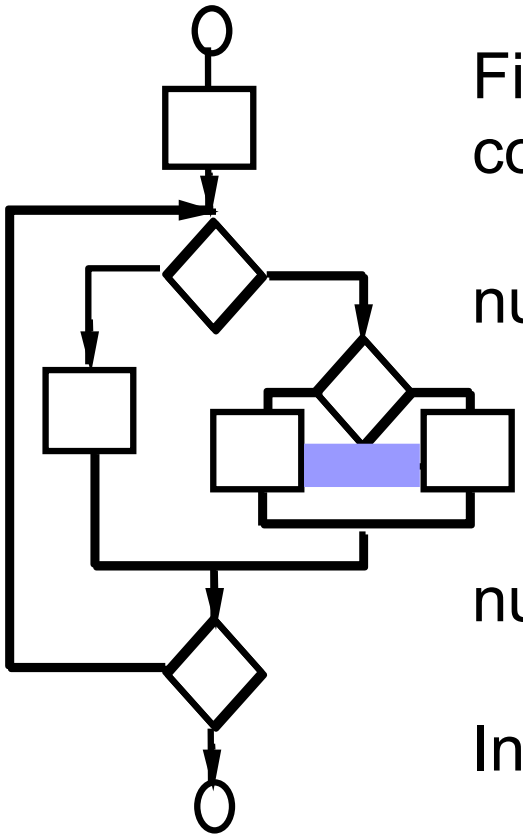
- It is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security.
- In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.
- It is one of two parts of the Box Testing approach to software testing.
- Its counterpart, Blackbox testing, involves testing from an external or end-user type perspective.
- On the other hand, Whitebox testing is based on the inner workings of an application and revolves around internal testing.

- The term "WhiteBox" was used because of the see-through box concept.
- The clear box or WhiteBox name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings.
- Likewise, the "black box" in "Black Box Testing" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

White Box Testing Techniques

- **Statement Coverage** - This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage** - This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage** - This technique corresponds to testing all possible paths which means that each statement and branch is covered

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

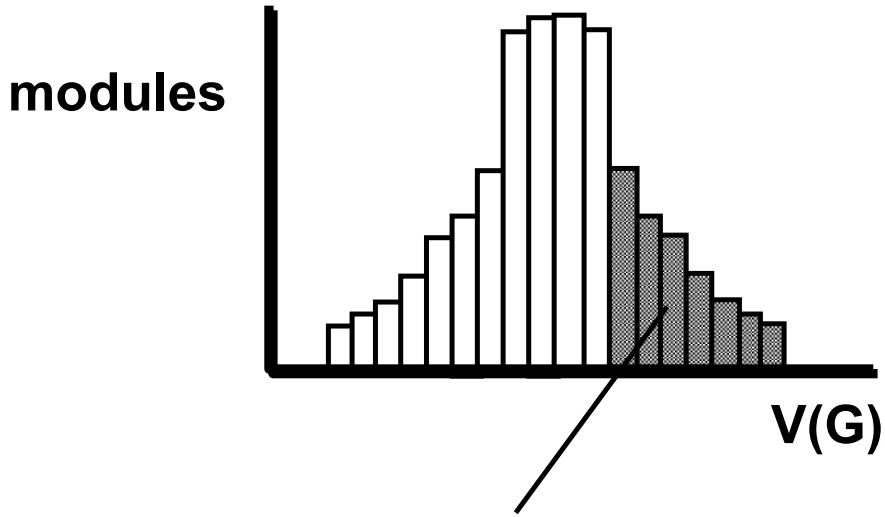
or

number of enclosed areas + 1

In this case, V(G) = 4

Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



Cyclomatic Complexity

- It is a software metric that measures the logical complexity of the program code.
- It counts the number of decisions in the given program code.
- It measures the number of linearly independent paths through the program code.
- Cyclomatic complexity indicates several information about the program code-

Cyclomatic Complexity	Meaning
1 – 10	•Structured and Well Written Code •High Testability •Less Cost and Effort
10 – 20	•Complex Code •Medium Testability •Medium Cost and Effort
20 – 40	•Very Complex Code •Low Testability •High Cost and Effort
> 40	•Highly Complex Code •Not at all Testable •Very High Cost and Effort

Importance of Cyclomatic Complexity

- It helps in determining the software quality.
- It is an important indicator of program code's readability, maintainability and portability.
- It helps the developers and testers to determine independent path executions.
- It helps to focus more on the uncovered paths.
- It evaluates the risk associated with the application or program.
- It provides assurance to the developers that all the paths have been tested at least once.

Properties of Cyclomatic Complexity

- It is the maximum number of independent paths through the program code.
- It depends only on the number of decisions in the program code.
- Insertion or deletion of functional statements from the code does not affect its cyclomatic complexity.
- It is always greater than or equal to 1.

Calculating Cyclomatic Complexity

- Cyclomatic complexity is calculated using the control flow representation of the program code
- In control flow representation of the program code,
  - Nodes represent parts of the code having no branches.
  - Edges represent possible control flow transfers during program execution
- There are 3 commonly used methods for calculating the cyclomatic complexity-

Calculating Cyclomatic Complexity

Method-01:

Cyclomatic Complexity = Total number of closed regions in the control flow graph + 1

Method-02:

Cyclomatic Complexity = E – N + 2

Here-
•E = Total number of edges in the control flow graph
•N = Total number of nodes in the control flow graph

Calculating Cyclomatic Complexity

Method-03:

Cyclomatic Complexity = P + 1

Here,
P = Total number of predicate nodes contained in the control flow graph

Note-

•Predicate nodes are the conditional nodes.
•They give rise to two branches in the control flow graph.

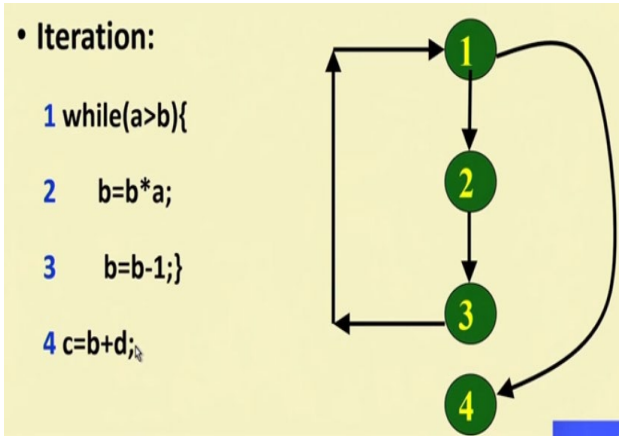
PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Problem-01

Calculate cyclomatic complexity for the given code-

Method-01:

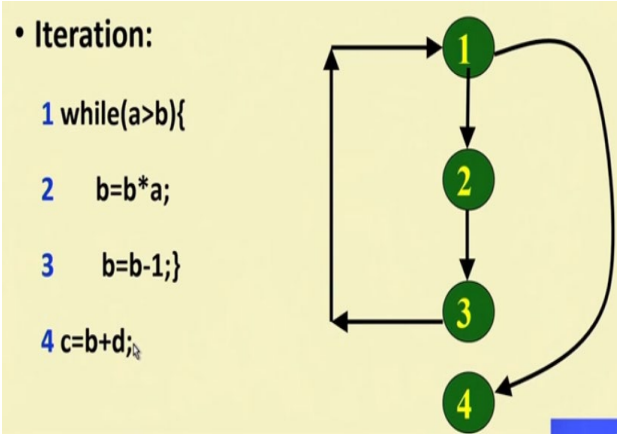
Cyclomatic Complexity
= Total number of closed regions in the control flow graph + 1
= 1+ 1
= 2



PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-02:

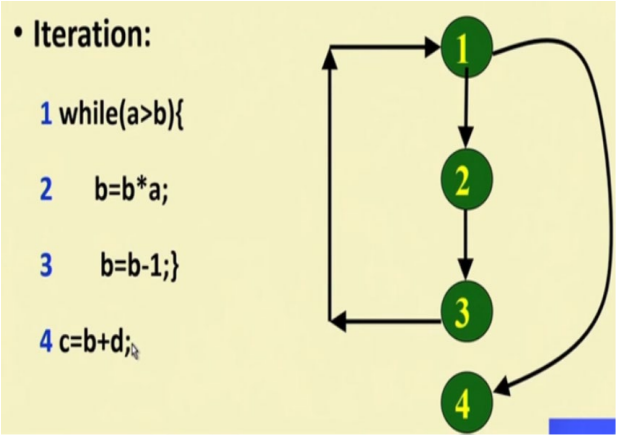
Cyclomatic Complexity
= 4 – 4 + 2
= 4 – 4 + 2
= 2



PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-03:

Cyclomatic Complexity
= P + 1
= 1 + 1
= 2

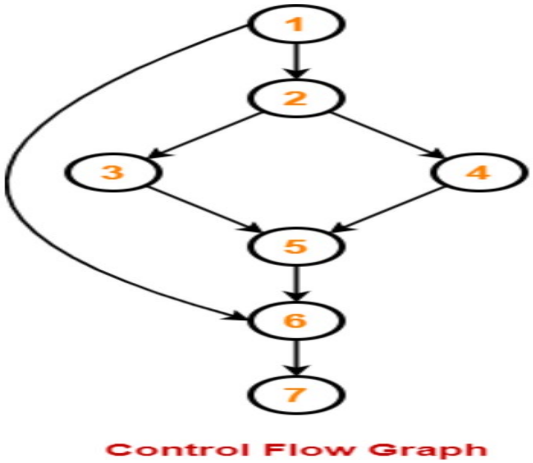


PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Problem-02

Calculate cyclomatic complexit for the given code-

```
1. IF A = 354
2. THEN IF B > C
3. THEN A = B
4. ELSE A = C
5. END IF
6. END IF
7. PRINT A
```

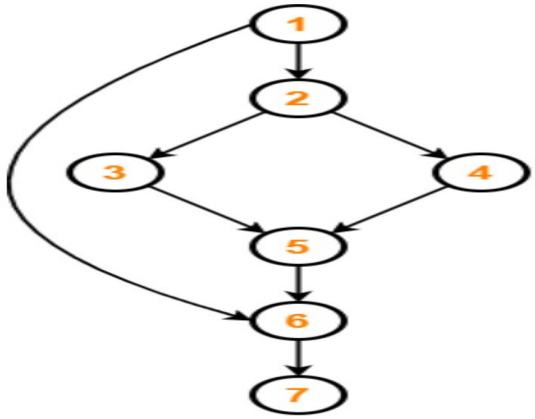


Control Flow Graph

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-01:

Cyclomatic Complexity
= Total number of closed regions in the control flow graph + 1
= 2 + 1
= 3

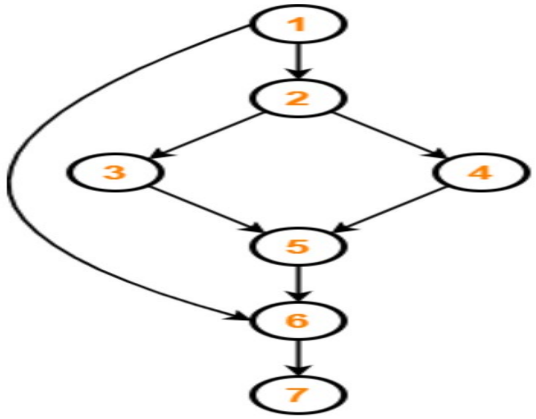


Control Flow Graph

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-02:

Cyclomatic Complexity
= E – N + 2
= 8 – 7 + 2
= 3

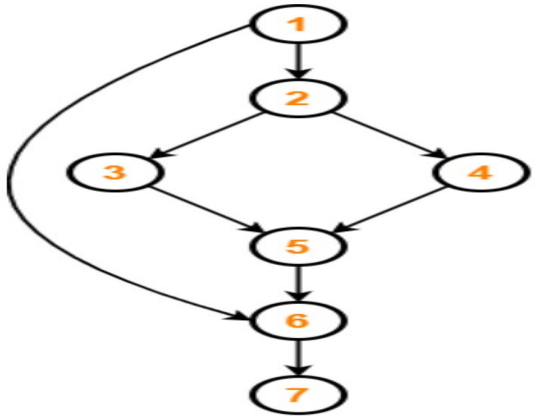


Control Flow Graph

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-03:

Cyclomatic Complexity
= P + 1
= 2 + 1
= 3

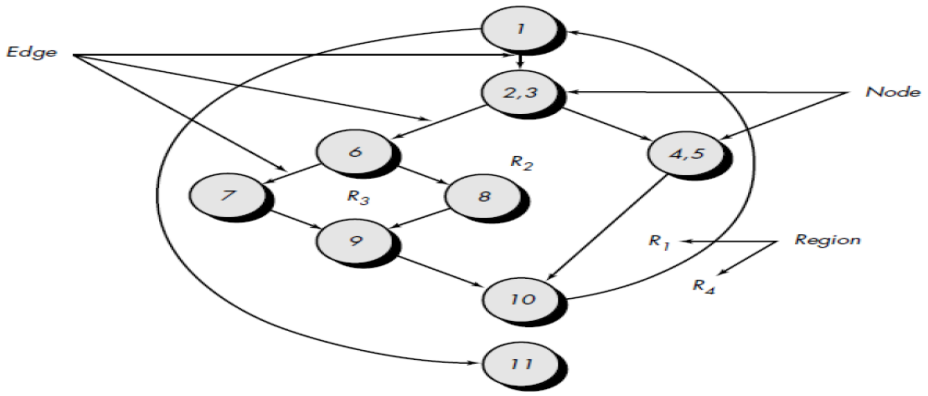


Control Flow Graph

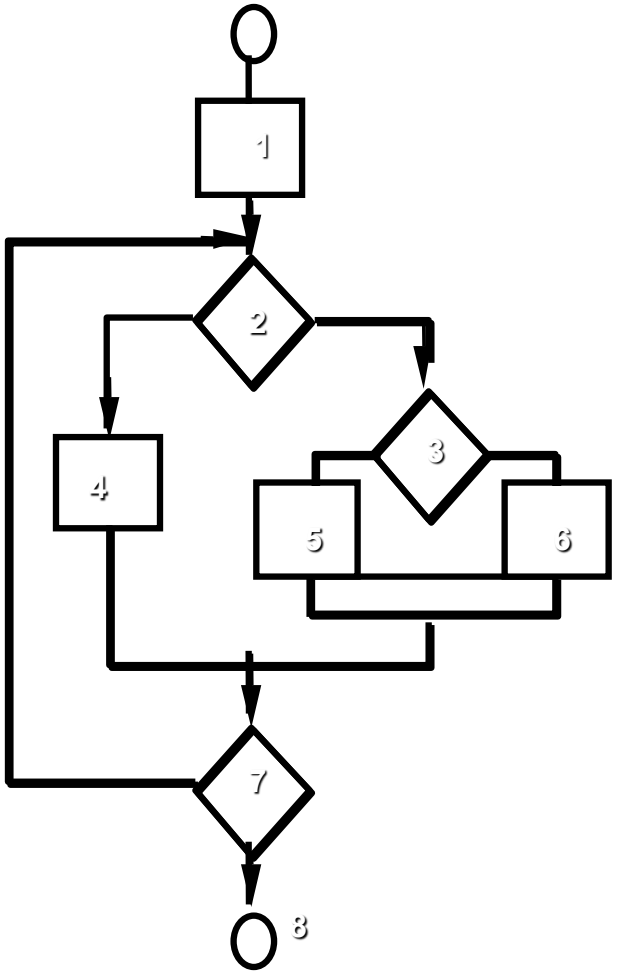
Problem 4

Find the Cyclomatic Complexity for the Control flow graph

Paths
- path 1: 1-11
- path 2: 1-2-3-4-5-10-1-11
- path 3: 1-2-3-6-8-9-10-1-11
- path 4: 1-2-3-6-7-9-10-1-11



Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

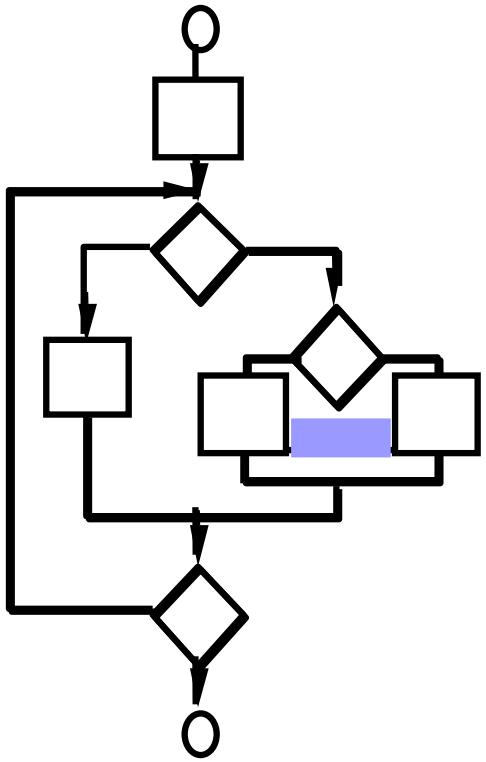
Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

Basis Path Testing Notes



- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

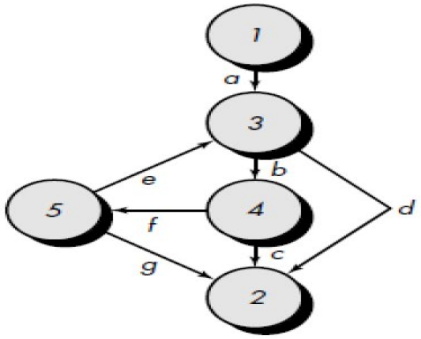
Deriving Test Cases

- *Summarizing:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

Graph Matrix

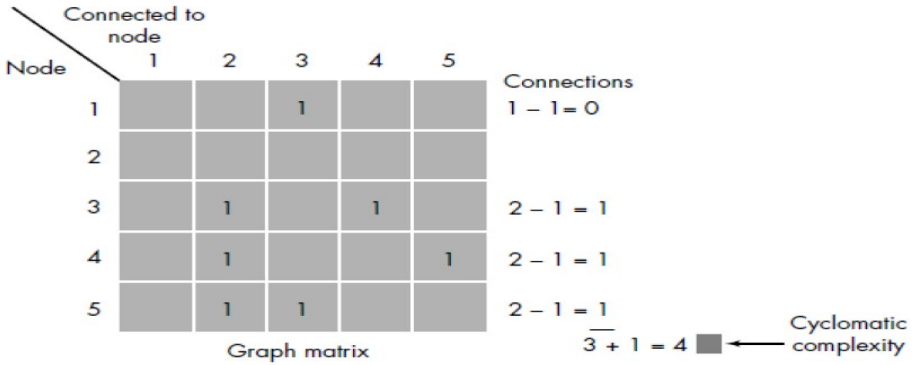


Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Connection Matrix



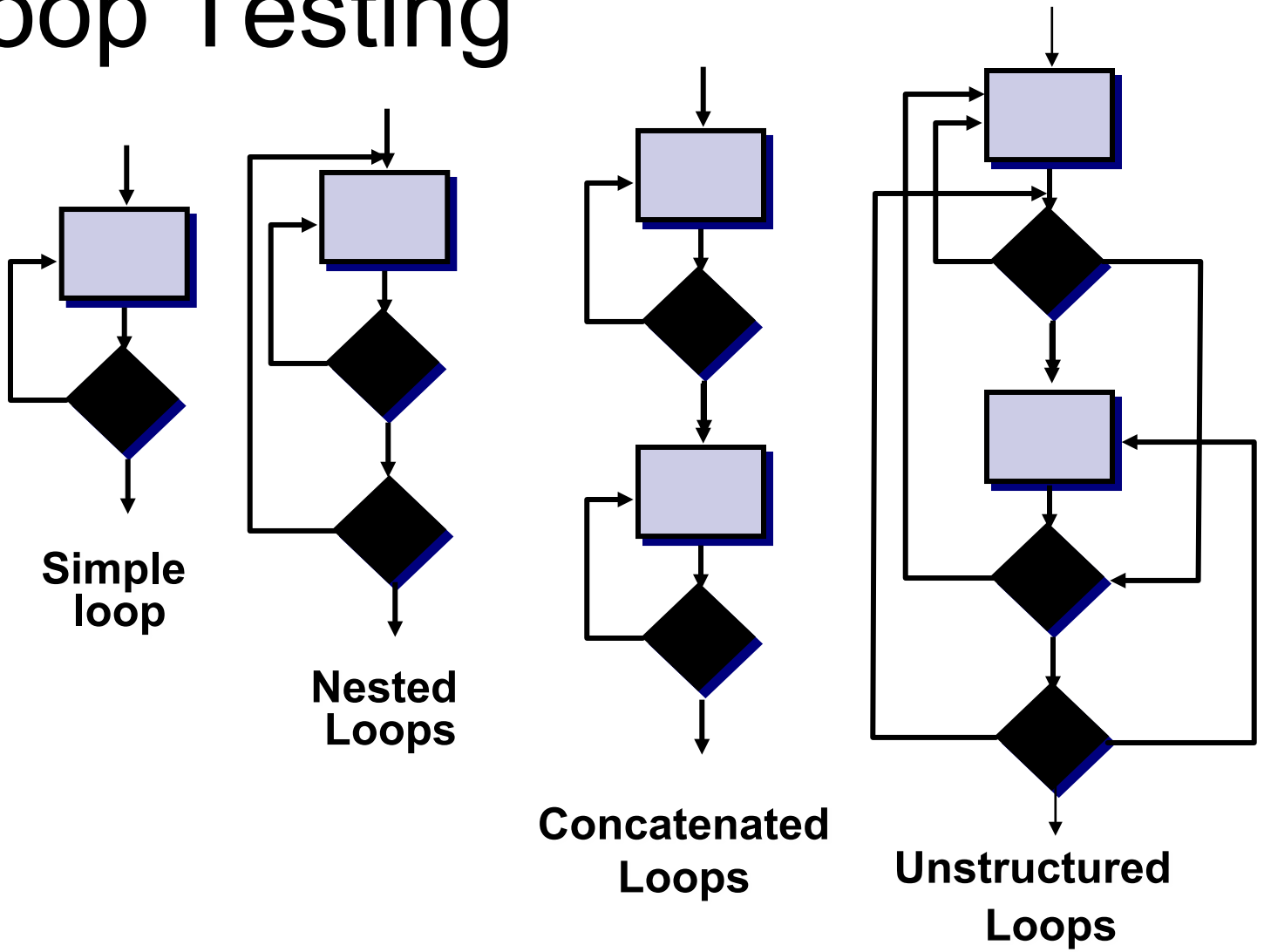
Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

Data Flow Testing

- The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number
    - DEF(*S*) = {*X* | statement *S* contains a definition of *X*}
    - USE(*S*) = {*X* | statement *S* contains a use of *X*}
  - A *definition-use (DU) chain* of variable X is of the form [*X, S, S'*], where *S* and *S'* are statement numbers, *X* is in DEF(*S*) and USE(*S'*), and the definition of *X* in statement *S* is live at statement *S'*

Loop Testing



Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. **skip the loop entirely**
2. **only one pass through the loop**
3. **two passes through the loop**
4. **m passes through the loop m < n**
5. **(n-1), n, and (n+1) passes through the loop**

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

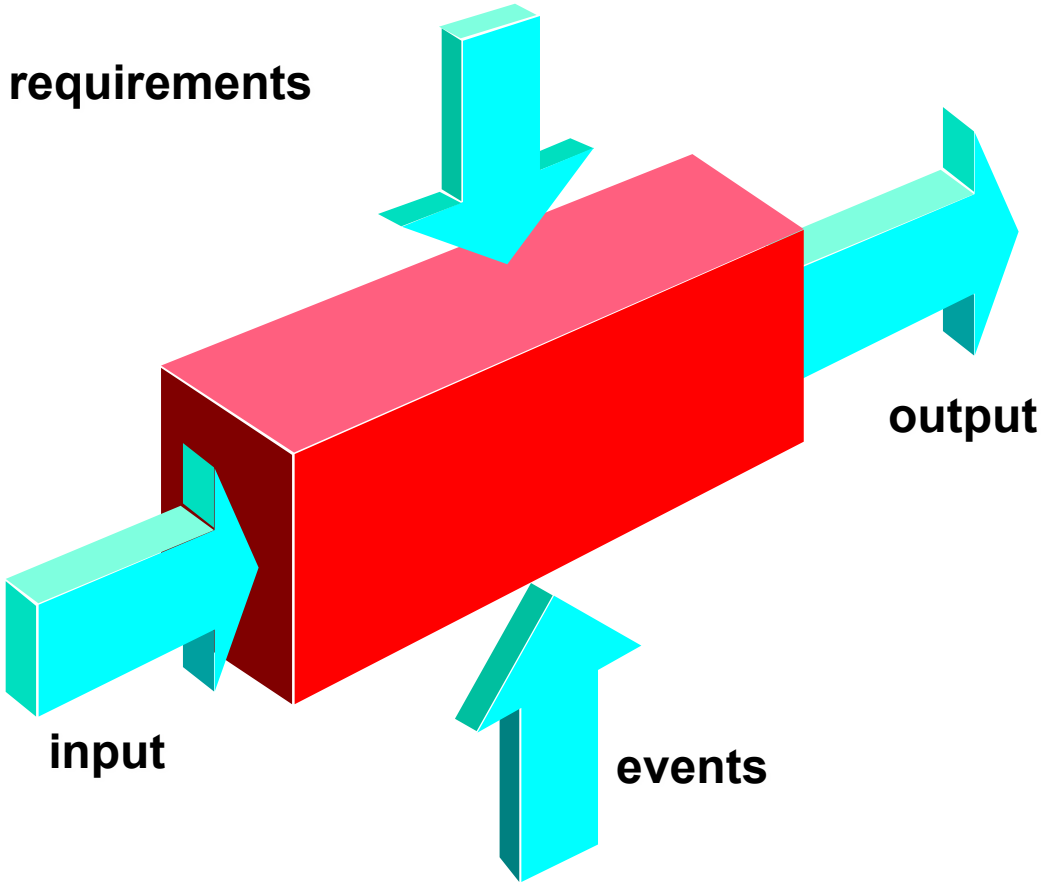
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-Box Testing



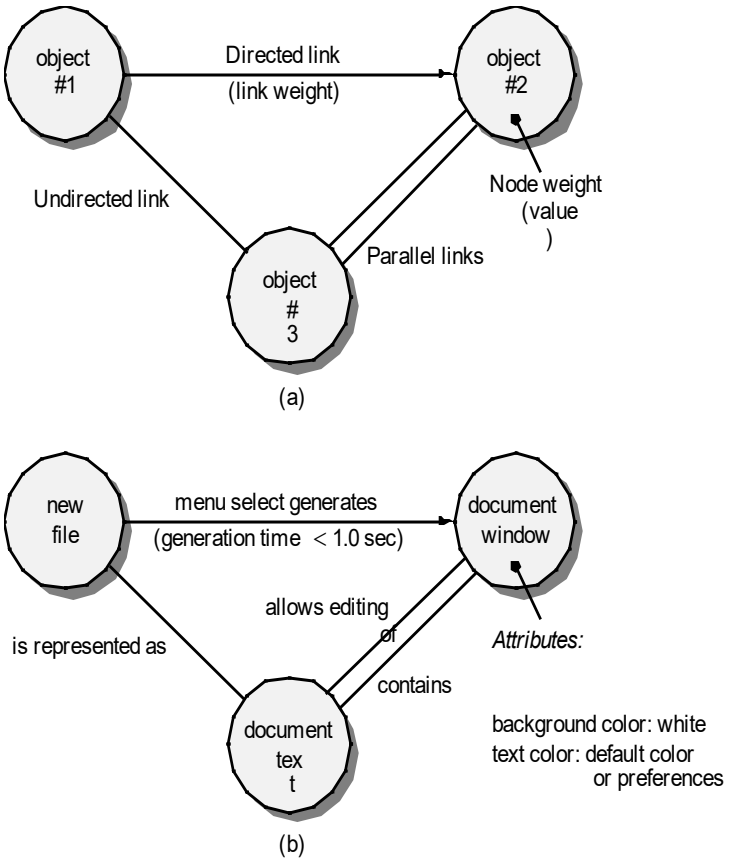
Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

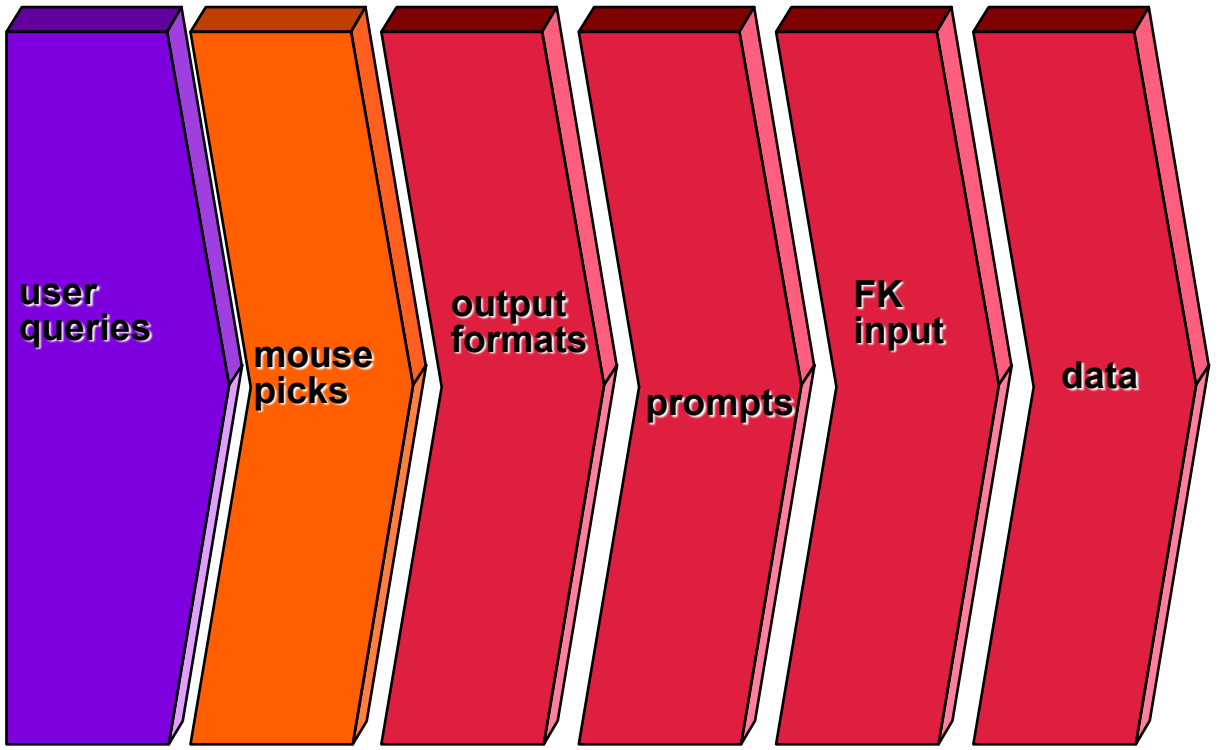
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

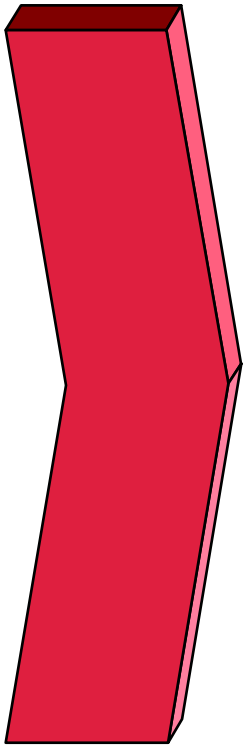
In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



Equivalence Partitioning



Sample Equivalence Classes



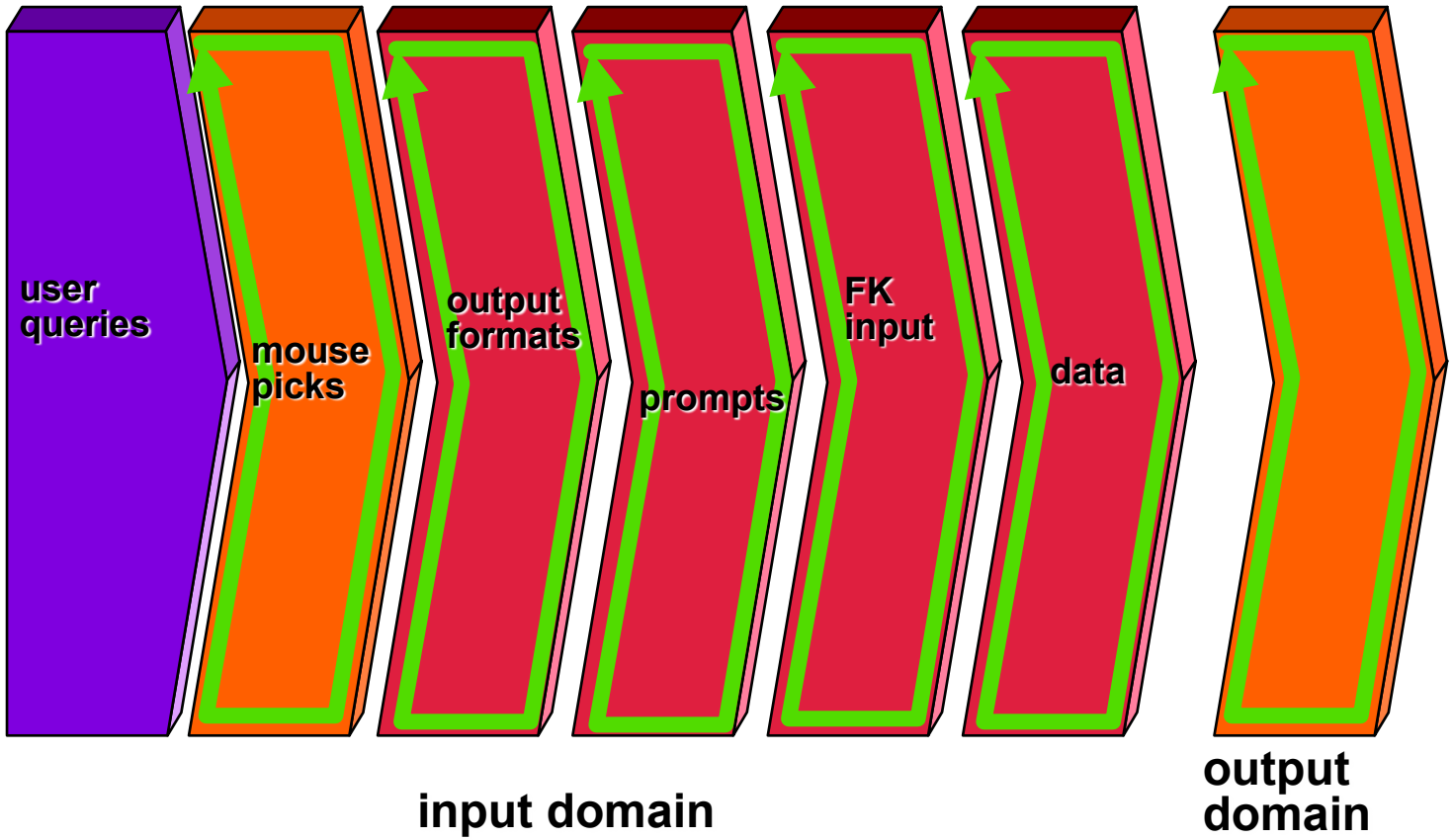
Valid data

- **user supplied commands**
- **responses to system prompts**
- **file names**
- **computational data**
  - **physical parameters**
  - **bounding values**
  - **initiation values**
- **output data formatting**
- **responses to error messages**
- **graphical data (e.g., mouse picks)**

Invalid data

- **data outside bounds of the program**
- **physically impossible data**
- **proper value supplied in wrong place**

Boundary Value Analysis



Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

THANK YOU