

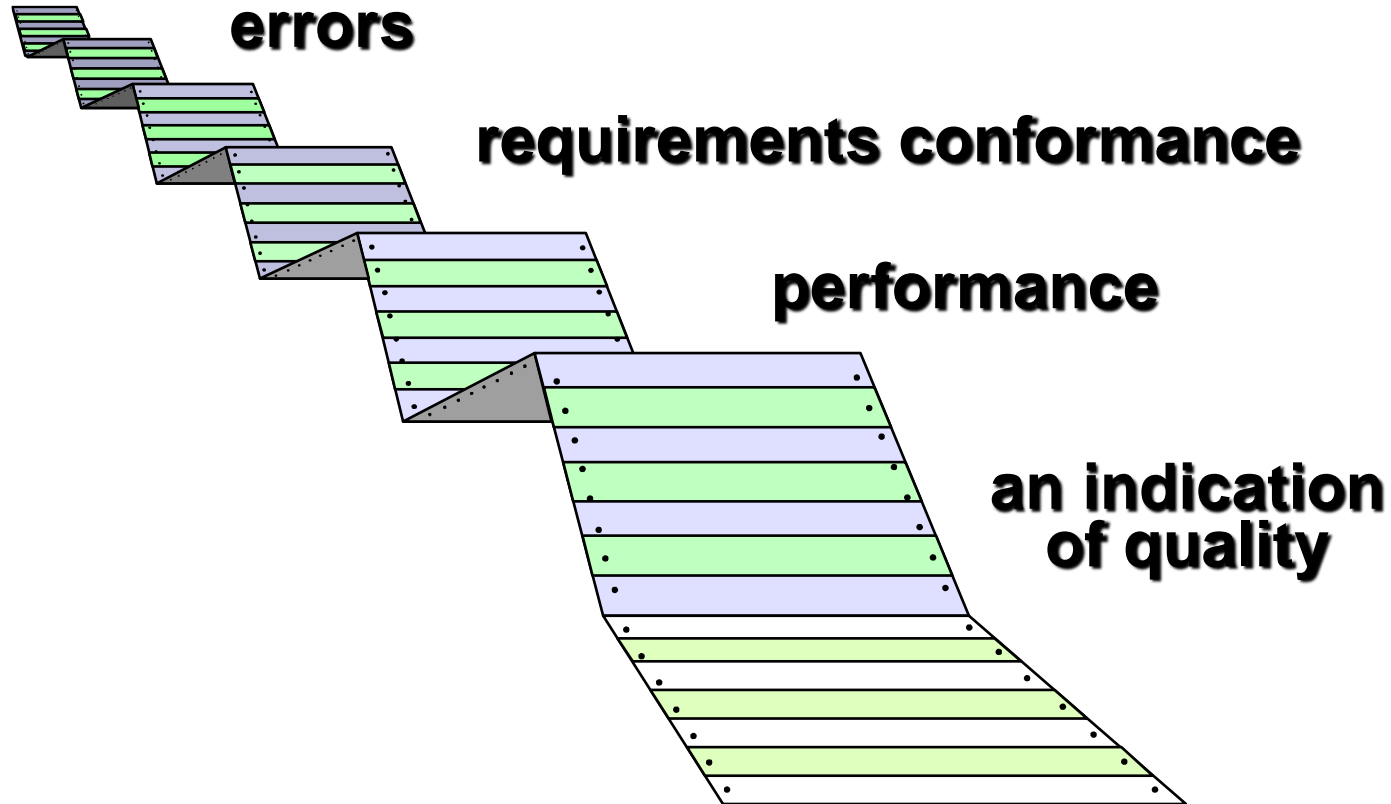
# Software Testing Strategies

# Software Testing

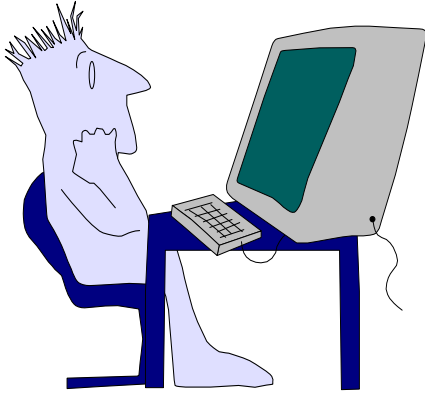
**Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.**



# What Testing Shows

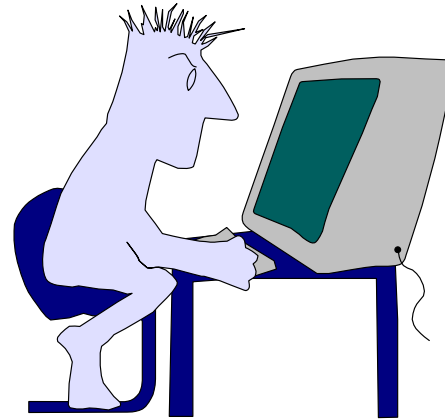


# Who Tests the Software?



***developer***

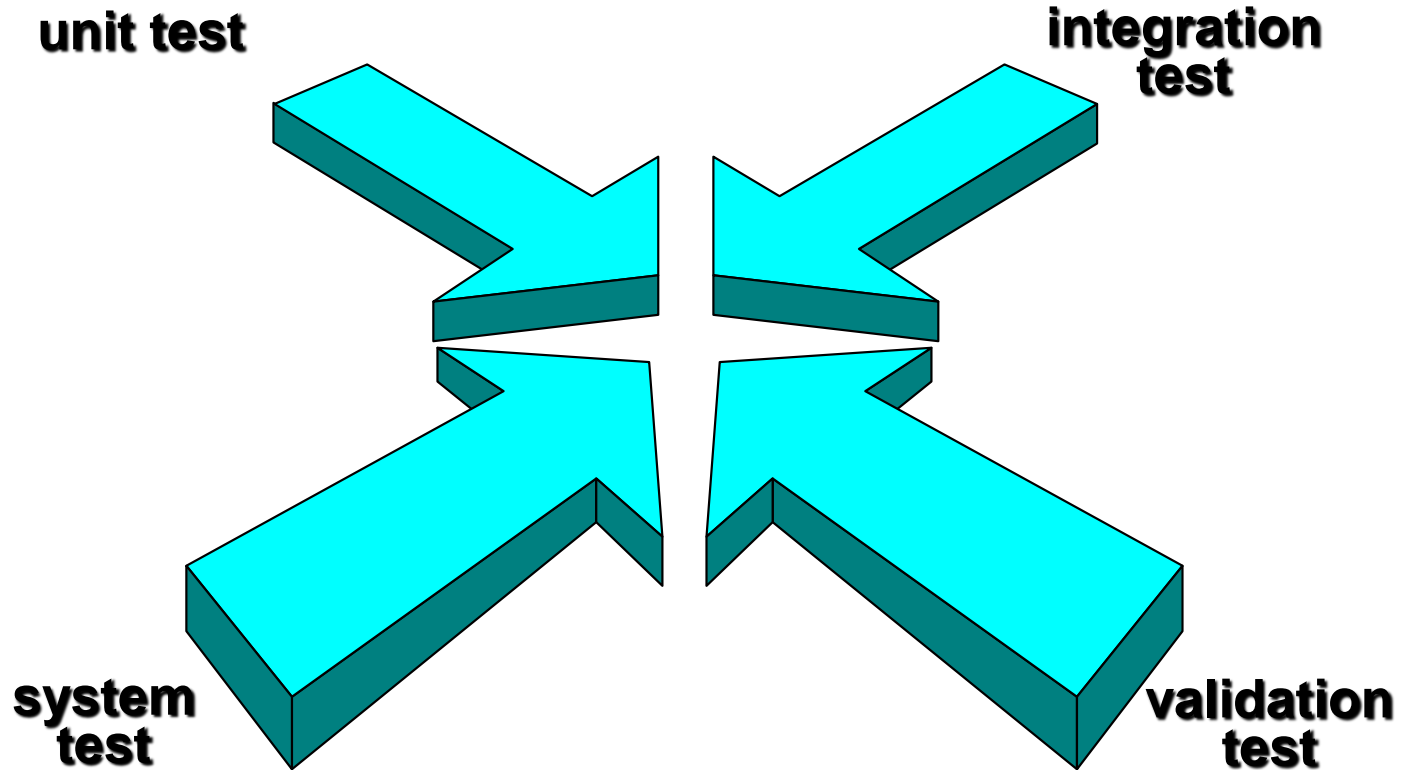
**Understands the system  
but, will test "gently"  
and, is driven by "delivery"**



***independent tester***

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

# Testing Strategy



# 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

- 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 formal technical reviews as a filter prior to testing
- Conduct formal technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

# Verification and validation

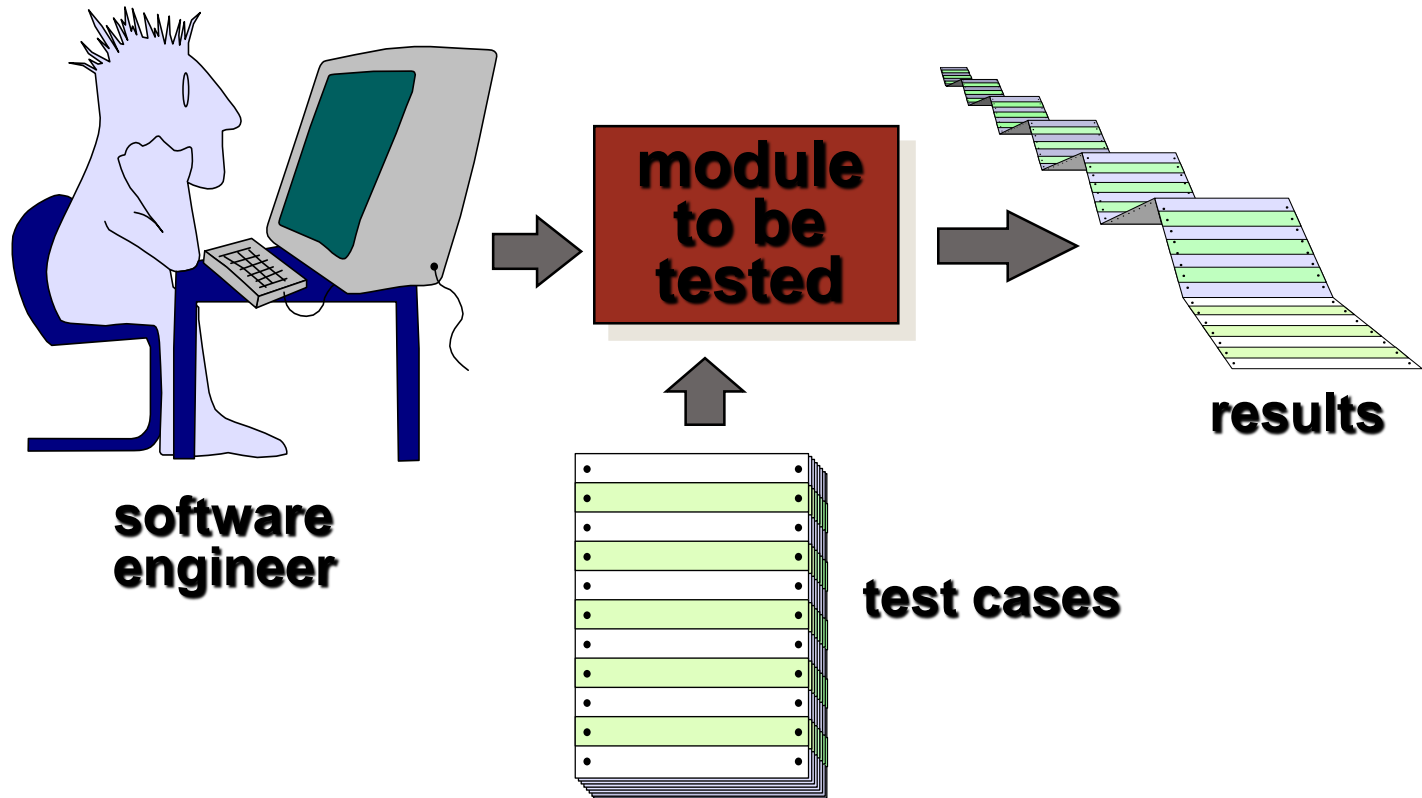
- **Verification : are we building the product right?**
- **Validation : are we building the right product?**



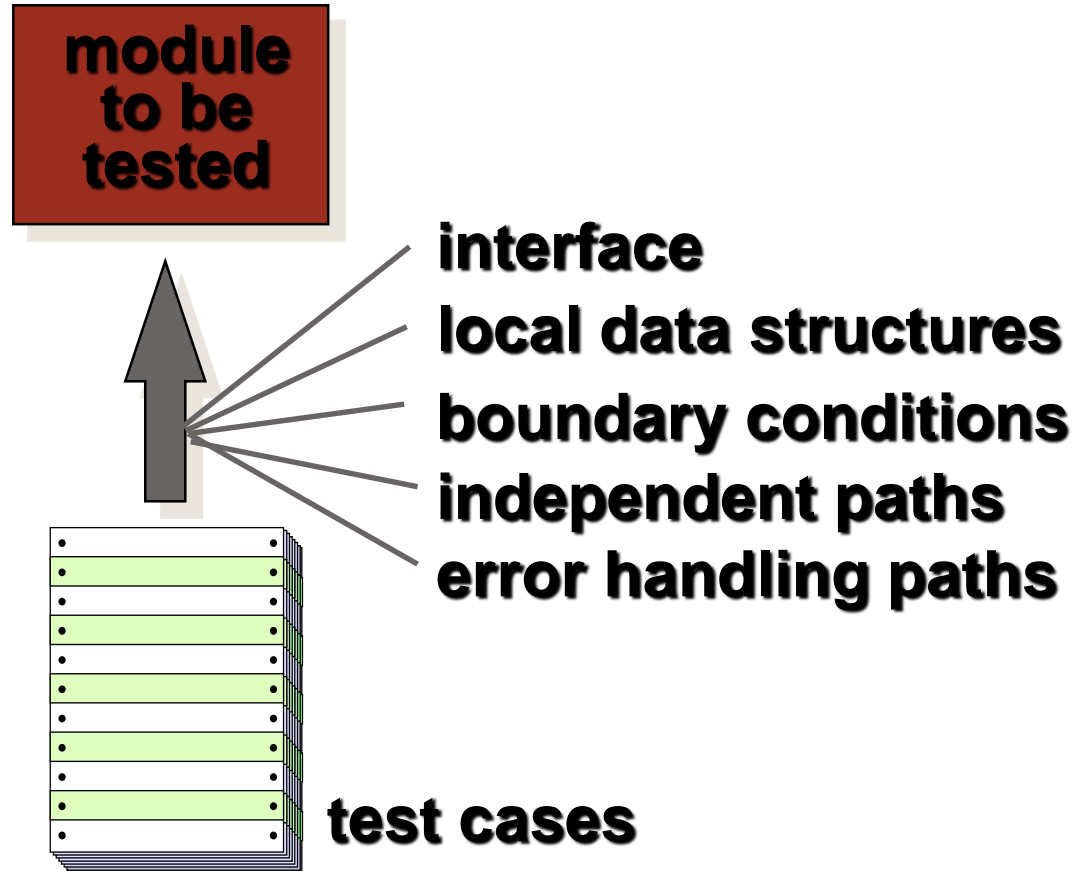
# Testing Strategies for Conventional Software

- Unit Testing
- Integration Testing

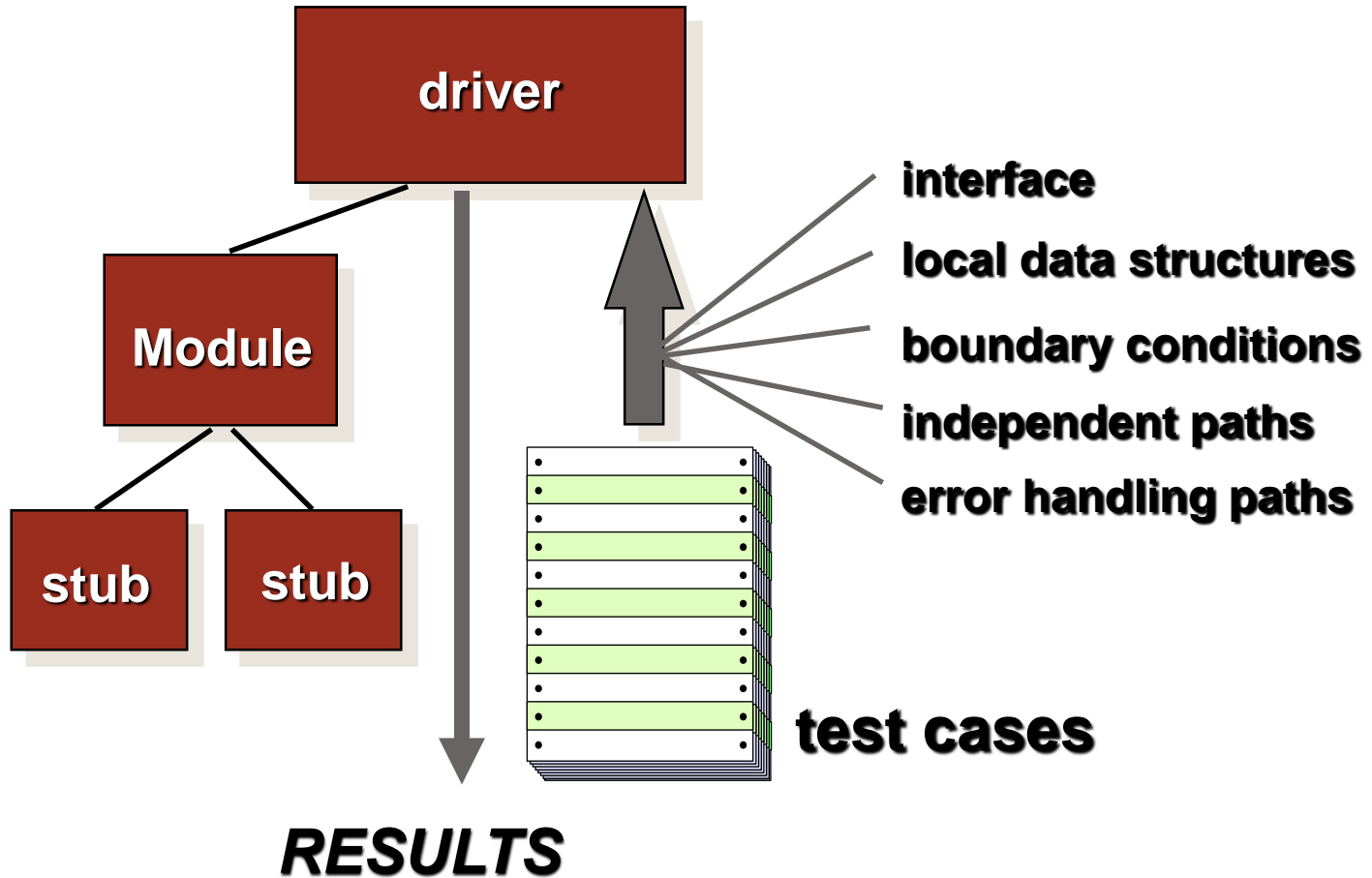
# Unit Testing



# Unit Testing



# Unit Test Environment



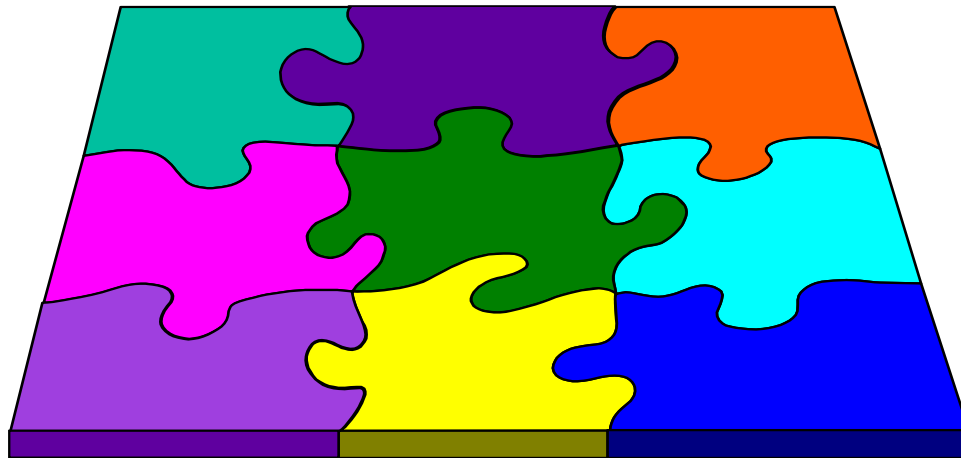
# Unit Testing

- Algorithms and logic
- Data structures (global and local)
- Interfaces
- Independent paths
- Boundary conditions
- Error handling

# Integration Testing Strategies

## **Options:**

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



# Why Integration Testing Is Necessary

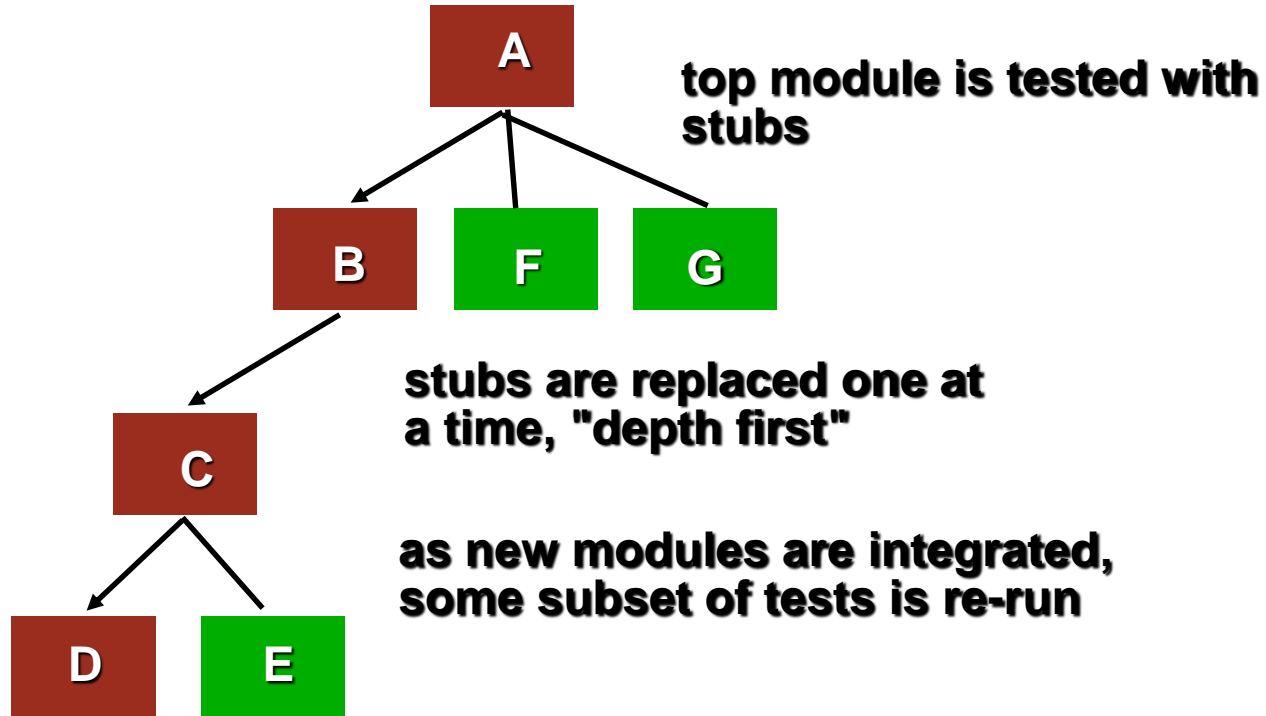
- One module can have an **adverse effect** on another
- Subfunctions, when combined, **may not produce** the desired major function

# Why Integration Testing Is Necessary (cont'd)

- **Interfacing errors** not detected in unit testing may appear
- **Timing problems** (in real-time systems) are not detectable by unit testing
- **Resource contention** problems are not detectable by unit testing



# Top Down Integration



## Top-Down Integration (cont'd)

3. Tests are run as **each individual module** is integrated.
4. On the successful completion of a set of tests, **another stub is replaced** with a real module
5. **Regression testing** is performed to ensure that errors have not developed as result of integrating new modules

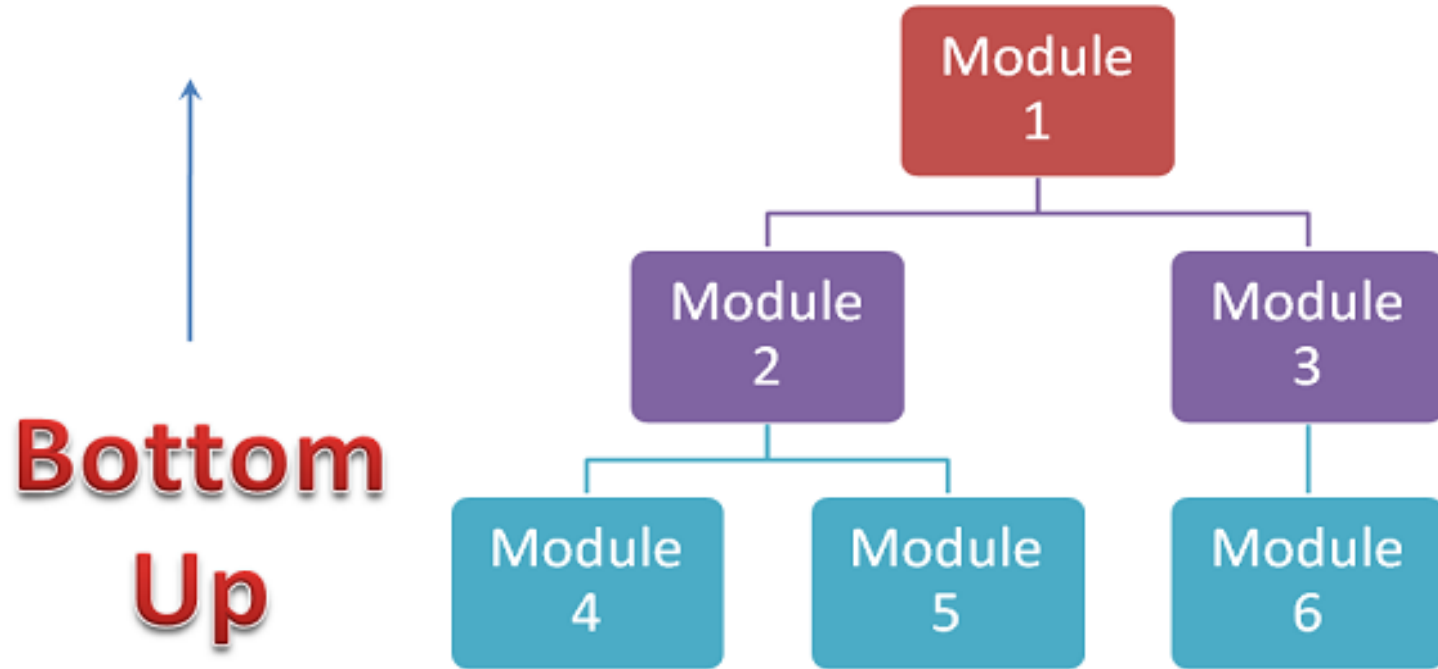
# Problems with Top-Down Integration

- Many times, **calculations are performed** in the modules at the **bottom of the hierarchy**
- Stubs typically **do not pass data** up to the higher modules
- Developing stubs that can pass data up is almost as much work as **developing the actual module**

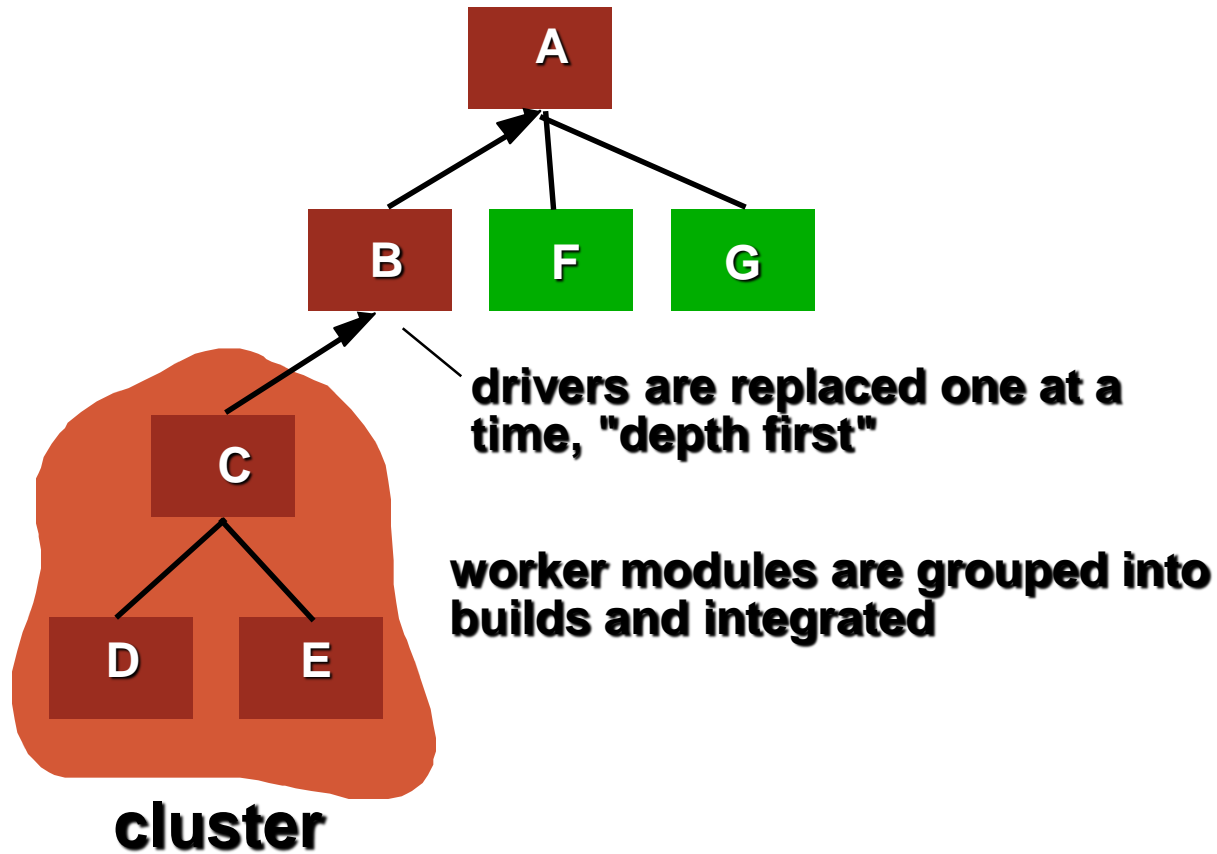
# Bottom-Up Integration

- Integration begins with the **lowest-level modules**, which are combined into **clusters**, or builds, that perform a specific software subfunction
- Drivers (control programs developed as stubs) are 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



# Bottom-Up Integration



# Problems with Bottom-Up Integration

- The **whole program does not exist** until the last module is integrated
- Timing and resource contention problems are **not found until late** in the process

# In short,

- 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



# Testing Web Applications

# Testing Quality Dimensions-I

- Content is evaluated at both a syntactic and semantic level.
  - syntactic level—spelling, punctuation and grammar are assessed for text-based documents.
  - semantic level—correctness (of information presented), consistency (across the entire content object and related objects) and lack of ambiguity are all assessed.
- Function is tested for correctness, instability, and general conformance to appropriate implementation standards (e.g., Java or XML language standards).
- Structure is assessed to ensure that it
  - properly delivers WebApp content and function
  - is extensible
  - can be supported as new content or functionality is added.

# Testing Quality Dimensions-II

- Usability is tested to ensure that each category of user
  - is supported by the interface
  - can learn and apply all required navigation syntax and semantics
- Navigability is tested to ensure that
  - all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).
- Performance is tested under a variety of operating conditions, configurations, and loading to ensure that
  - the system is responsive to user interaction
  - the system handles extreme loading without unacceptable operational degradation

# Testing Quality Dimensions-III

- Compatibility is tested by executing the WebApp in a variety of different host configurations on both the client and server sides.
  - The intent is to find errors that are specific to a unique host configuration.
- Interoperability is tested to ensure that the WebApp properly interfaces with other applications and/or databases.
- Security is tested by assessing potential vulnerabilities and attempting to exploit each.
  - Any successful penetration attempt is deemed a security failure.

# WebApp Testing Strategy-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.
- Selected functional components are unit tested.

# WebApp Testing Strategy-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.

# Content Testing

- Content testing has three important objectives:
  - to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media
  - to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs, and
  - to find errors in the organization or structure of content that is presented to the end-user.

# Assessing Content Semantics

- Is the information factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?
- Is the content offensive, misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?



# User Interface Testing

- Interface features are tested to ensure that design rules, aesthetics, and related visual content is available for the user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing.
- Each interface mechanism is tested within the context of a use-case for a specific user category.
- The complete interface is tested against selected use-cases to uncover errors in the semantics of the interface.
- The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible.

# Testing Interface Mechanisms

- Links—navigation mechanisms that link the user to some other content object or function.
- Forms—a structured document containing blank fields that are filled in by the user. The data contained in the fields are used as input to one or more WebApp functions.
- Client-side scripting—a list of programmed commands in a scripting language (e.g., Javascript) that handle information input via forms or other user interactions
- Dynamic HTML—leads to content objects that are manipulated on the client side using scripting or cascading style sheets (CSS).
- Client-side pop-up windows—small windows that pop-up without user interaction. These windows can be content-oriented and may require some form of user interaction.

# Navigation Testing

- The following navigation mechanisms should be tested:
  - Navigation links—these mechanisms include internal links within the WebApp, external links to other WebApps, and anchors within a specific Web page.
  - Redirects—these links come into play when a user requests a non-existent URL or selects a link whose destination has been removed or whose name has changed.
  - Bookmarks—although bookmarks are a browser function, the WebApp should be tested to ensure that a meaningful page title can be extracted as the bookmark is created.
  - Frames and framesets—tested for correct content, proper layout and sizing, download performance, and browser compatibility
  - Site maps—Each site map entry should be tested to ensure that the link takes the user to the proper content or functionality.
  - Internal search engines—Search engine testing validates the accuracy and completeness of the search, the error-handling properties of the search engine, and advanced search features

# Configuration Testing

- Server-side
  - Is the WebApp fully compatible with the server OS?
  - Are system files, directories, and related system data created correctly when the WebApp is operational?
  - Do system security measures (e.g., firewalls or encryption) allow the WebApp to execute and service users without interference or performance degradation?
  - Has the WebApp been tested with the distributed server configuration (if one exists) that has been chosen?
  - Is the WebApp properly integrated with database software? Is the WebApp sensitive to different versions of database software?
  - Do server-side WebApp scripts execute properly?
  - Have system administrator errors been examined for their affect on WebApp operations?
  - If proxy servers are used, have differences in their configuration been addressed with on-site testing?

# Configuration Testing

- Client-side
  - Hardware—CPU, memory, storage and printing devices
  - Operating systems—Linux, Macintosh OS, Microsoft Windows, a mobile-based OS
  - Browser software—Internet Explorer, Mozilla/Netscape, Opera, Safari, and others
  - User interface components—Active X, Java applets and others
  - Plug-ins—QuickTime, RealPlayer, and many others
  - Connectivity—cable, DSL, regular modem, T1
- The number of configuration variables must be reduced to a manageable number

# Security Testing

- Designed to probe vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and the server-side environment
- On the client-side, vulnerabilities can often be traced to pre-existing bugs in browsers, e-mail programs, or communication software.
- On the server-side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client-side or used to disable server operations

# Performance Testing

- Does the server response time degrade to a point where it is noticeable and unacceptable?
- At what point (in terms of users, transactions or data loading) does performance become unacceptable?
- What system components are responsible for performance degradation?
- What is the average response time for users under a variety of loading conditions?
- Does performance degradation have an impact on system security?
- Is WebApp reliability or accuracy affected as the load on the system grows?
- What happens when loads that are greater than maximum server capacity are applied?

# Load Testing

- The intent is to determine how the WebApp and its server-side environment will respond to various loading conditions
  - N, the number of concurrent users
  - T, the number of on-line transactions per unit of time
  - D, the data load processed by the server per transaction
- Overall throughput, P, is computed in the following manner:
  - $P = N \times T \times D$



# Stress Testing

- Does the system degrade ‘gently’ or does the server shut down as capacity is exceeded?
- Does server software generate “server not available” messages? More generally, are users aware that they cannot reach the server?
- Does the server queue requests for resources and empty the queue once capacity demands diminish?
- Are transactions lost as capacity is exceeded?
- Is data integrity affected as capacity is exceeded?
- What values of N, T, and D force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?
- If the system does fail, how long will it take to come back on-line?
- Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?

# Regression Testing

- Regression testing is the re execution of some subset of tests that have already been conducted
- Regression testing is an important strategy for reducing **side effects**.
- Run test every time a **major change** is made to the software.

# Regression Testing

- Test case contains
  - Sample of tests that will exercise all software functions
  - Additional test that contain software function that may change
  - Test focus on software component that have been change

# Smoke Testing

- Smoke testing is integration testing approach.
- It is designed as a pacing mechanism for time critical project
- Smoke testing following activates
  - Software component translate into code.
  - A series of test is designed to expose error.
  - Build is integrated with other builds and the entire product is smoke tested.

# Smoke Testing

Smoke testing provide number of benefit when applied on complex time critical software projects:

- ☐ Integration risk is minimized
- ☐ The quality of end product is improved
- ☐ Error diagnosis and correction are simplified
- ☐ Progress is easier to access

# Validation testing

- Validation testing- when a software functions in a manner that can be expected by customer
- Information contained in Validation test
  - Validation test criteria
    - Classes of test to conducted
    - Test procedures( test cases)

# Validation testing

- Configuration review
  - Ensure all elements of the software configuration have been properly developed
- Alpha and beta testing
  - Alpha test- conducted by developers site by end user
  - Beta test- conducted at end users site

# Validation Testing

- Determine if the software meets all of the **requirements defined in the SRS**
- Having written requirements is essential
- Regression testing is performed to determine if the software still meets all of its requirements in light of changes and modifications to the software
- Regression testing involves selectively repeating existing validation tests, not developing new tests



# Alpha and Beta Testing

- It's best to provide customers with an outline of the things that you would like them to focus on and specific test scenarios for them to execute.
- Provide with customers who are actively involved with a commitment to fix defects that they discover.

# Alpha Testing versus Beta testing:

- Alpha testing performed by Testers who are usually internal employees of the organization
- Beta testing is performed by Clients or End Users who are not employees of the organization
- Alpha Testing performed at developer's site
- Beta testing is performed at client location or end user of the product
- Reliability and security testing are not performed in-depth Alpha Testing
- Reliability, Security, Robustness are checked during Beta Testing
- Alpha testing involves both the white box and black box techniques
- Beta Testing typically uses black box testing
- Long execution cycle may be required for Alpha testing
- Only few weeks of execution are required for Beta testing

# Alpha Testing versus Beta testing:

- Alpha testing requires lab environment or testing environment
- Beta testing doesn't require any lab environment or testing environment. Software is made available to the public and is said to be real time environment
- Critical issues or fixes can be addressed by developers immediately in Alpha testing
- Most of the issues or feedback is collected from Beta testing will be implemented in future versions of the product
- Alpha testing is to ensure the quality of the product before moving to Beta testing
- Beta testing also concentrates on quality of the product, but gathers users input on the product and ensures that the product is ready for real time users.

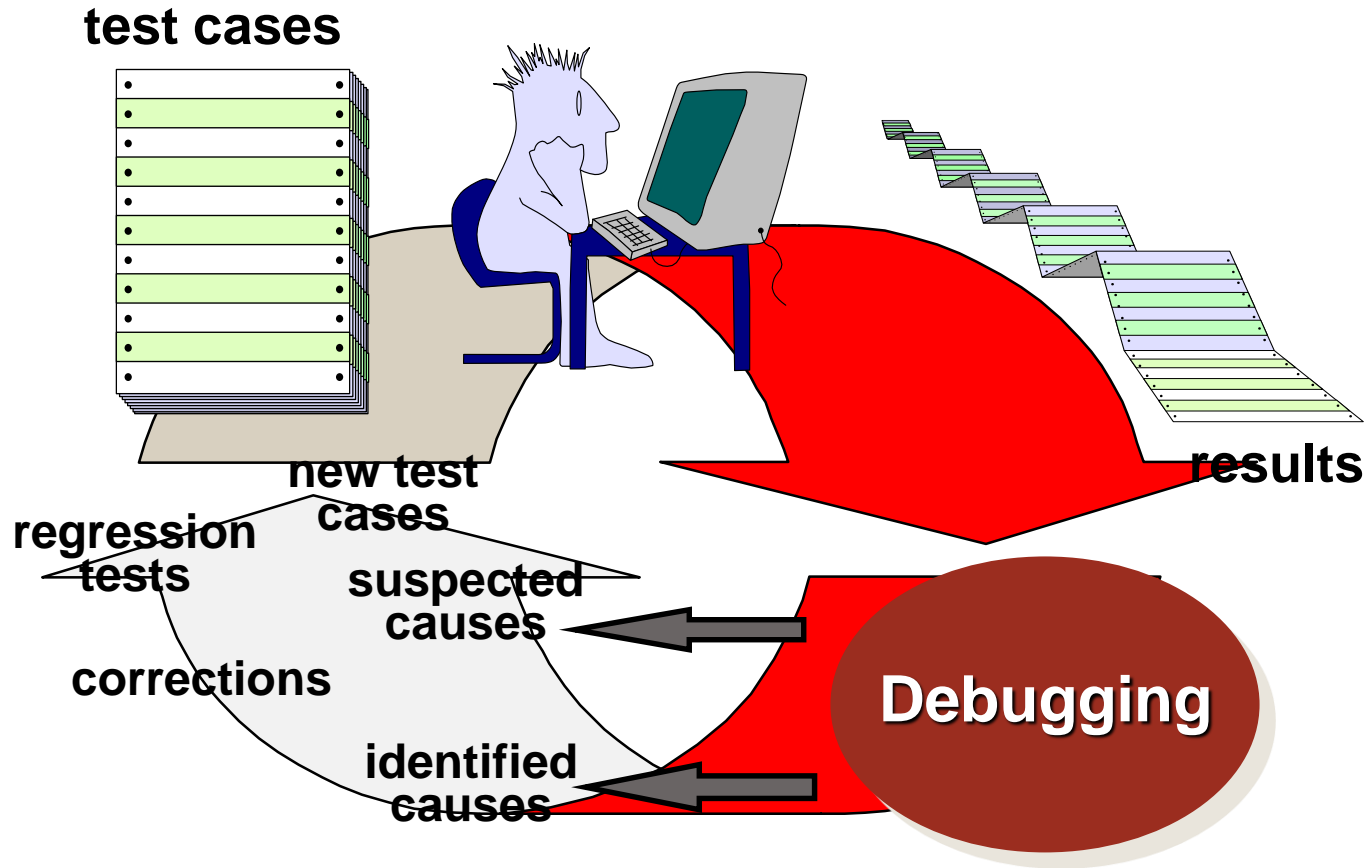
# System Testing

- To verify system elements are properly integrated and perform allocated functions
- Types of System Testing
  - Recovery Testing
  - Security Testing
  - Stress Testing
  - Performance Testing

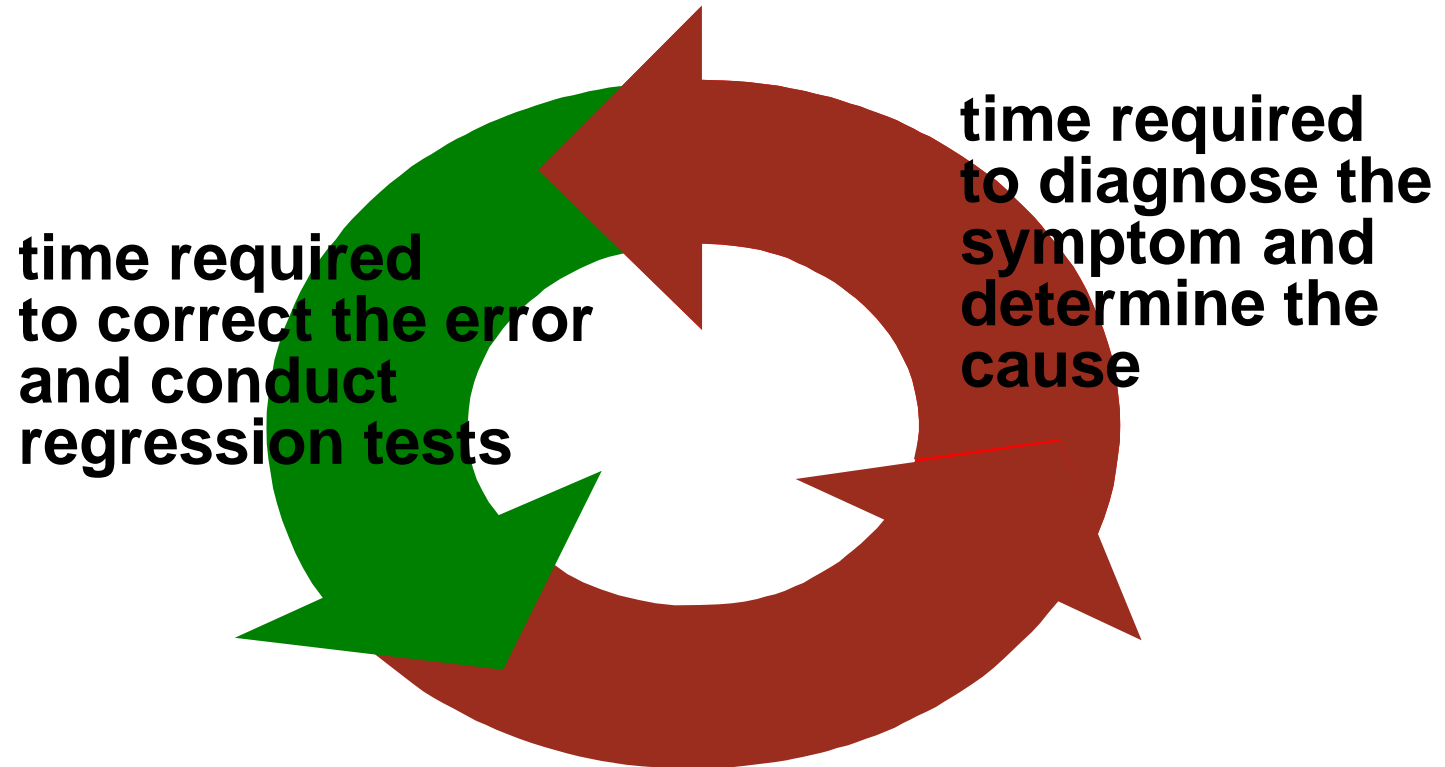
# Debugging: A Diagnostic Process



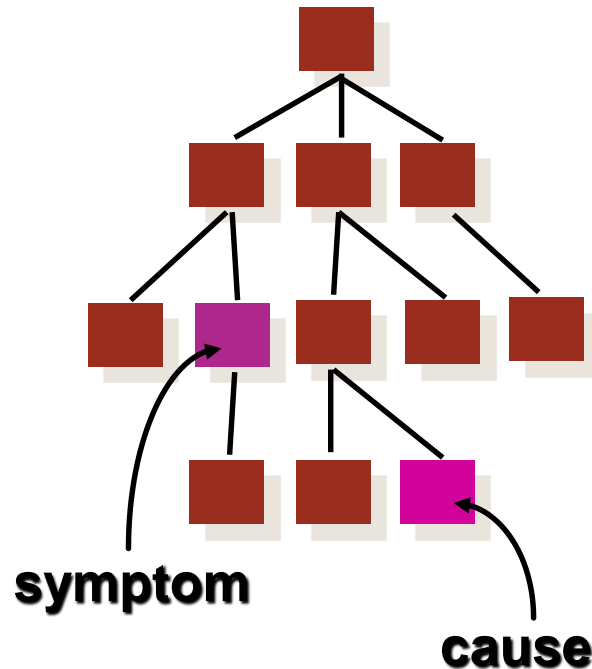
# The Debugging Process



# Debugging Effort



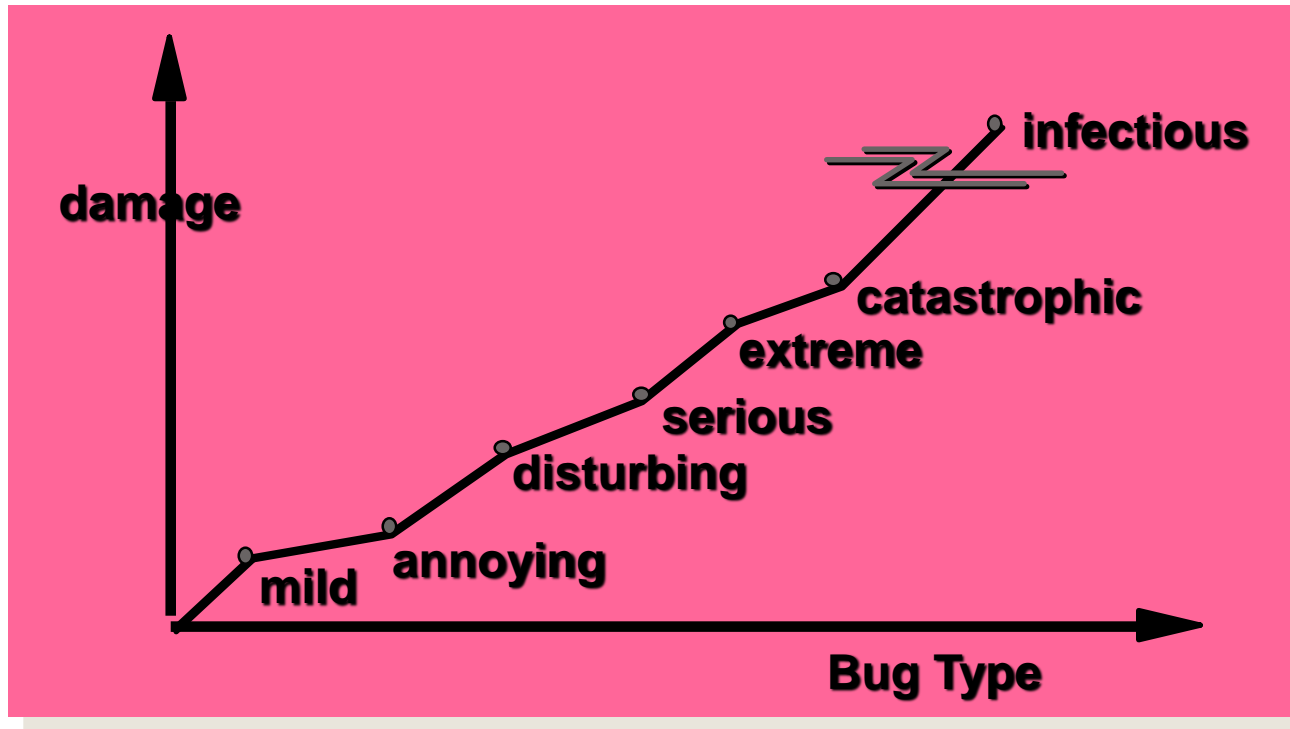
# Symptoms & Causes



- ❑ **symptom and cause may be geographically separated**
- ❑ **symptom may disappear when another problem is fixed**
- ❑ **cause may be due to a combination of non-errors**
- ❑ **cause may be due to a system or compiler error**
- ❑ **cause may be due to assumptions that everyone believes**
- ❑ **symptom may be intermittent**



# Consequences of Bugs



**Bug Categories:** function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.

# Debugging Techniques

- ❑ **brute force / testing**
- ❑ **backtracking**
- ❑ **Cause elimination**

# Debugging: Final Thoughts

1. Don't run off half-cocked, think about the symptom you're seeing.
2. Use tools (e.g., dynamic debugger) to gain more insight.
3. If at an impasse, get help from someone else.
4. Be absolutely sure to conduct regression tests when you do "fix" the bug.

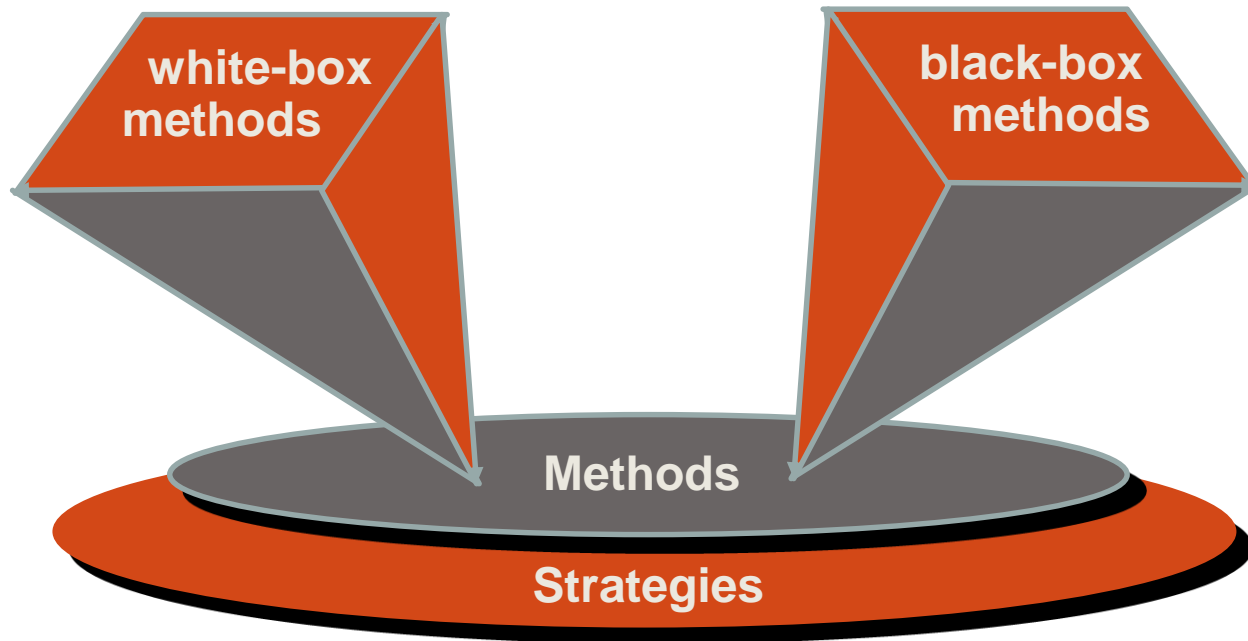
# Testability

- **Operability**—it operates cleanly
- **Observability**—the results of each test case are readily observed
- **Controllability**—the degree to which testing can be automated and optimized
- **Decomposability**—testing can be targeted
- **Simplicity**—reduce complex architecture and logic to simplify tests
- **Stability**—few changes are requested during testing
- **Understandability**—of the design

# Testing Vs Debugging

Testing	Debugging
1. Process to find Bugs	1. Process to Fix Bugs
2. Done By Testers	2. Done by Developers.
3. Aim is to find maximum bugs	3. Aim is to make bug free

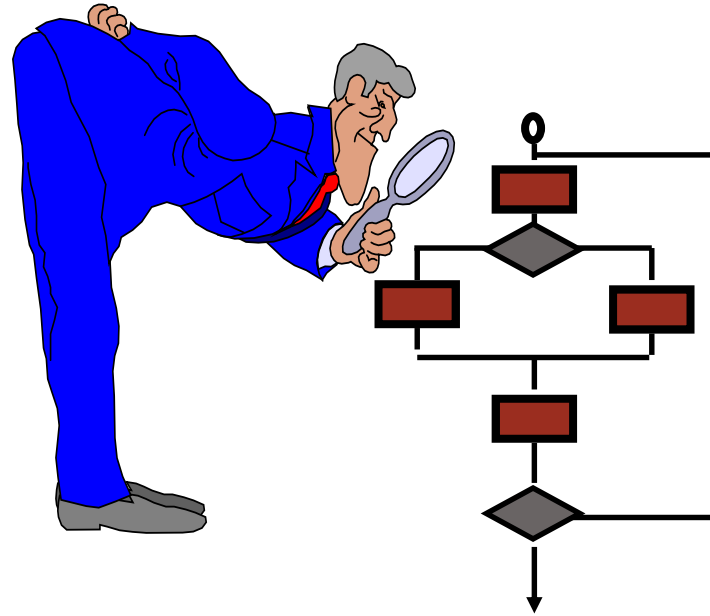
# External and Internal View of Testing



# White Box Testing

- All independent path within module is executed at least once.
- Check all logical decision
- Execute all loops at their boundaries
- check internal data structure

# White-Box Testing(Glass box testing)



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



# Basis Path Testing

The structured constructs in flow graph form:

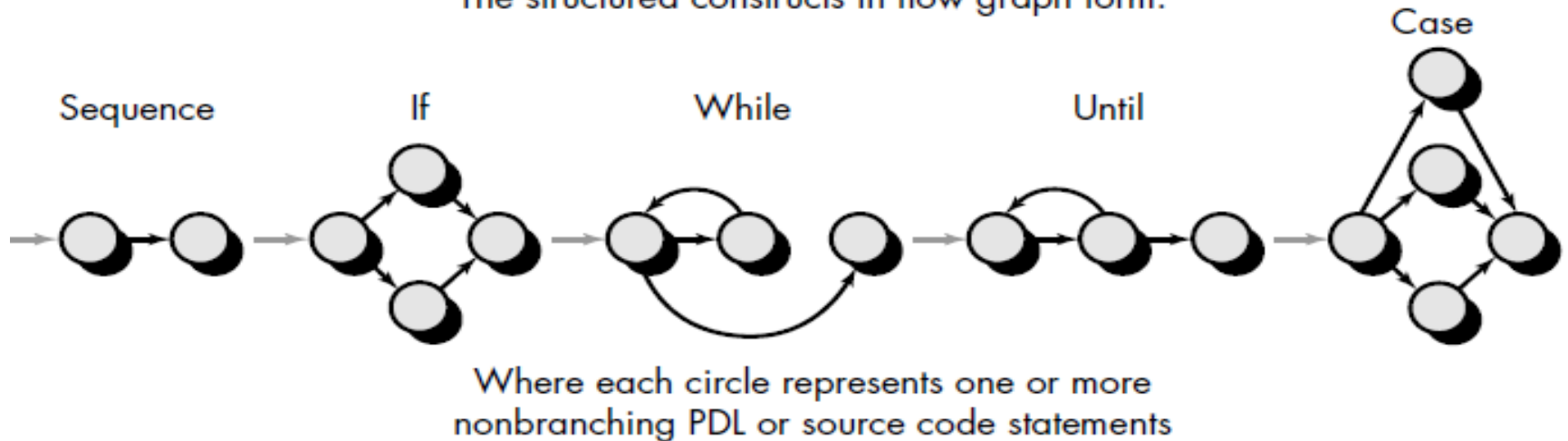
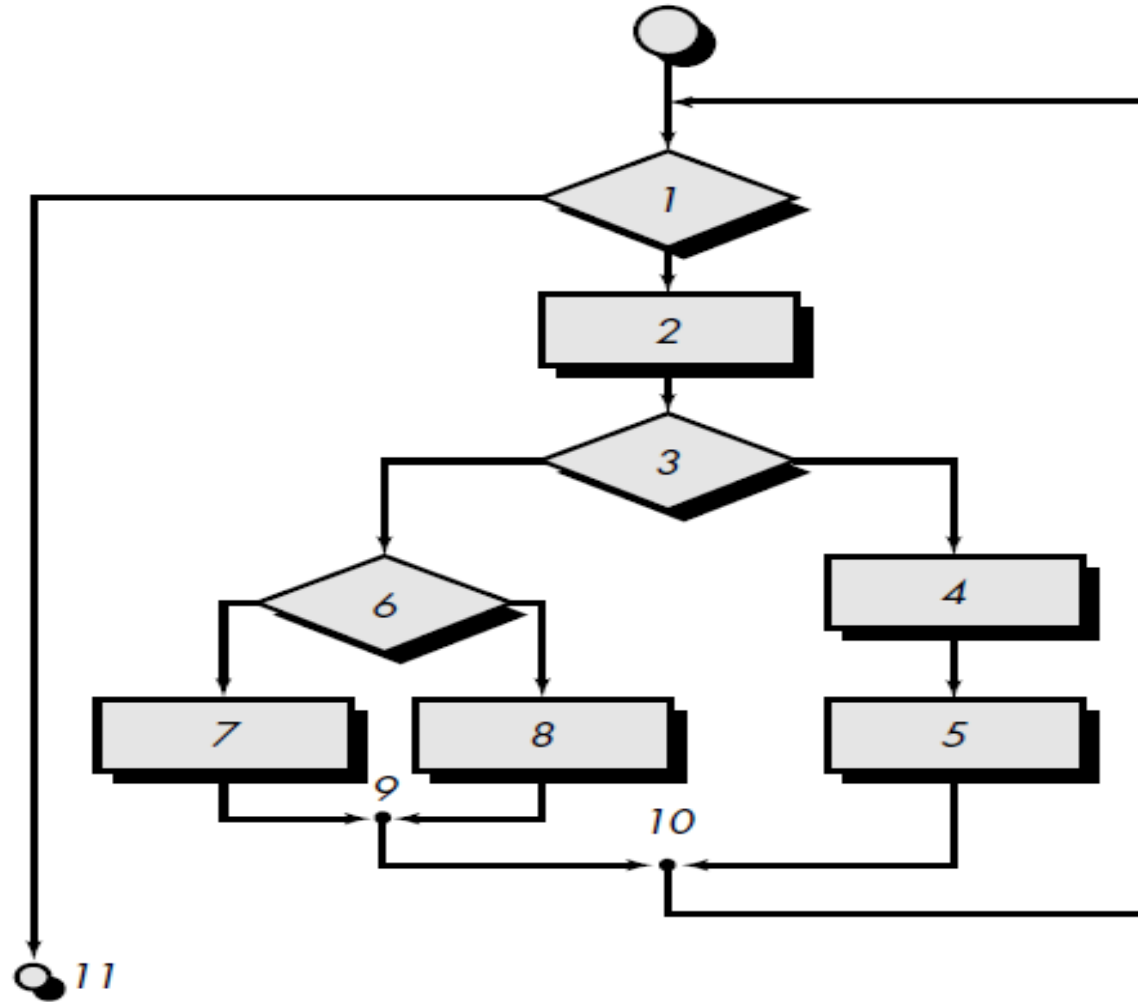
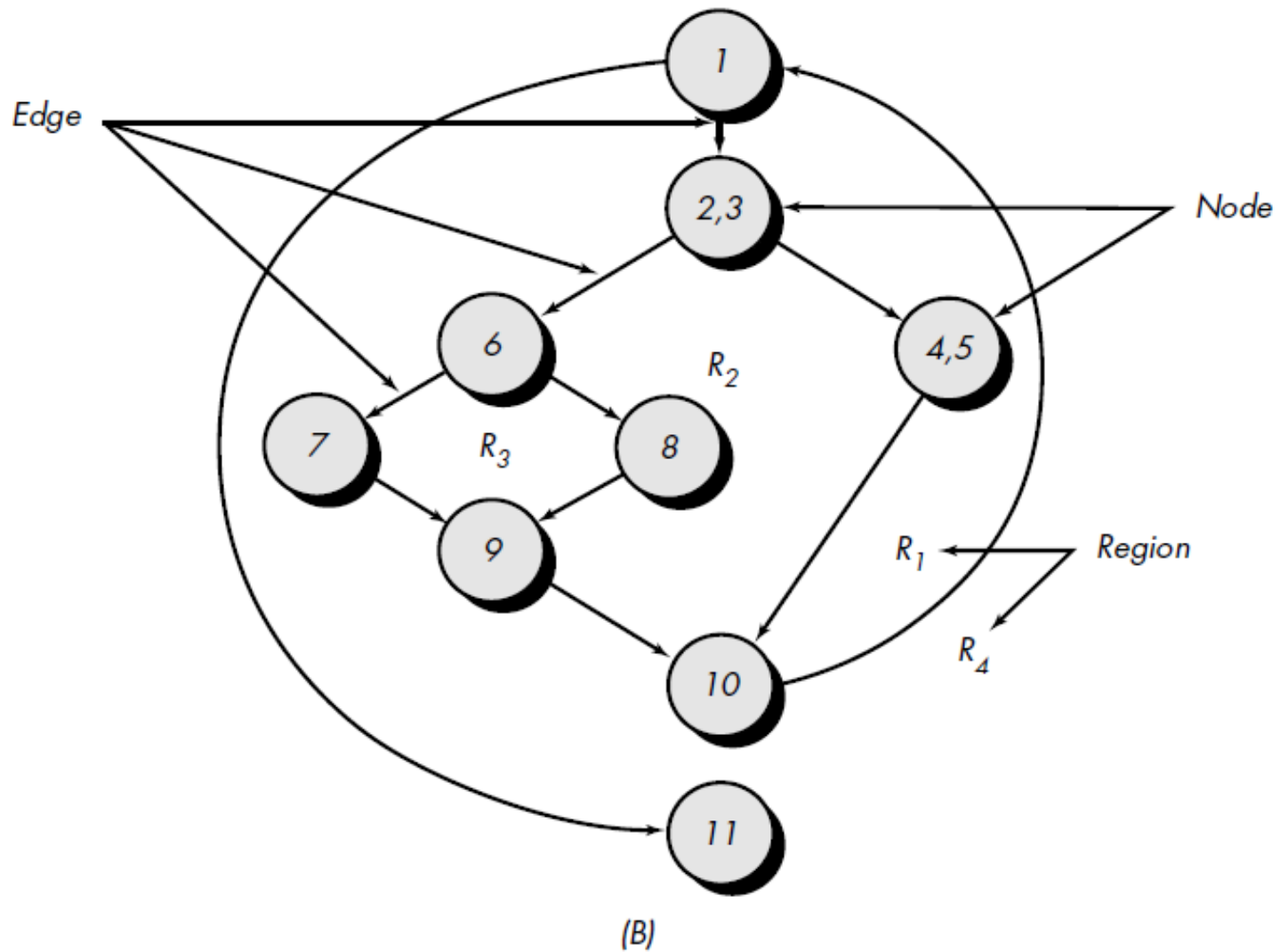


Fig:-Flow graph notation



(A)



# Independent Program Paths

- Independent path is any path through the program that introduces **at least one new set of processing statement**
- In terms of flow graph, **path must move at least one edge that has not been traversed before**
  - 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

# Cyclomatic Complexity

- How do we know how many paths to look for?
- Cyclomatic complexity is the answer
- **Cyclomatic Complexity**
  - The number of regions of the flow graph correspond to the Cyclomatic complexity.
  - Cyclomatic complexity,  $V(G)$ , for a flow graph,  $G$ , is defined as  $V(G) = E - N + 2$
  - Cyclomatic complexity,  $V(G)$ , for a flow graph,  $G$ , is also defined as  $V(G) = P + 1$

# Cyclomatic Complexity

- For above example:
- $V(G) = 11 - 9 + 2 = 4$
- $V(G) = 3 + 1 = 4$

i.e  $V(G) = \text{Cyclomatic Complexity} = 4$

# Basis Path Testing

**Next, we derive the independent paths:**

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

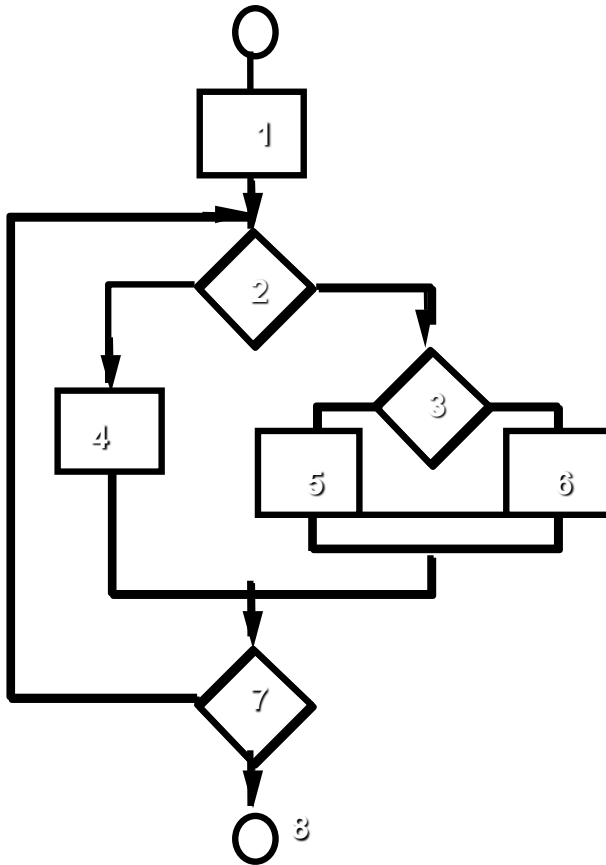
**Path 1: 1,2,3,6,7,8**

**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.**



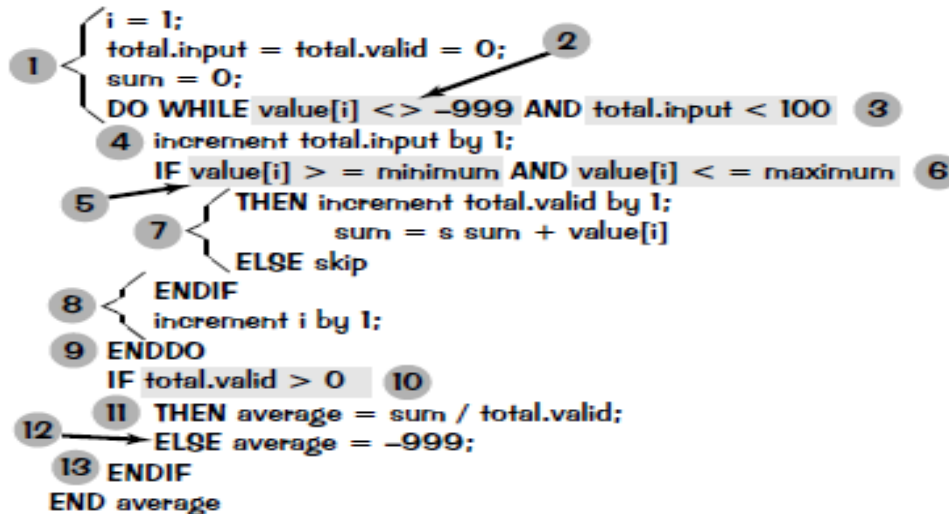
# Deriving a test case

**PROCEDURE average;**

- \* This procedure computes the average of 100 or fewer numbers that lie between bounding values; it also computes the sum and the total number valid.

**INTERFACE RETURNS** average, total.input, total.valid;  
**INTERFACE ACCEPTS** value, minimum, maximum;

**TYPE** value[1:100] **IS** **SCALAR ARRAY**;  
**TYPE** average, total.input, total.valid;  
    minimum, maximum, sum **IS** **SCALAR**;  
**TYPE** i **IS** **INTEGER**;



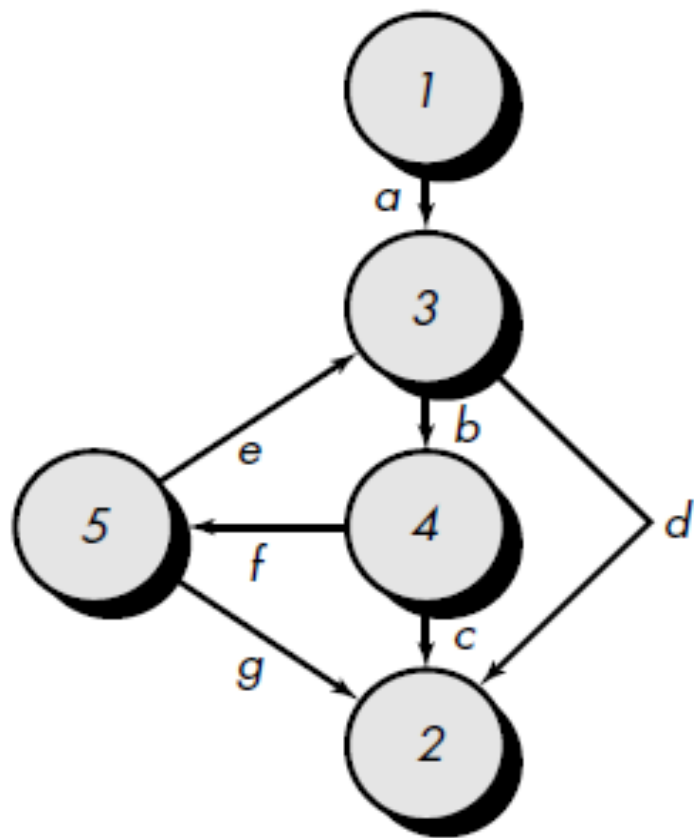


# Contd..

- Using design or code as a foundation, draw a corresponding flow graph
- Determine Cyclomatic complexity of a resultant flow graph
- Determine basic set of linear independent paths
- Prepare test case that will force execution of each path in the basic 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



*Flow graph*

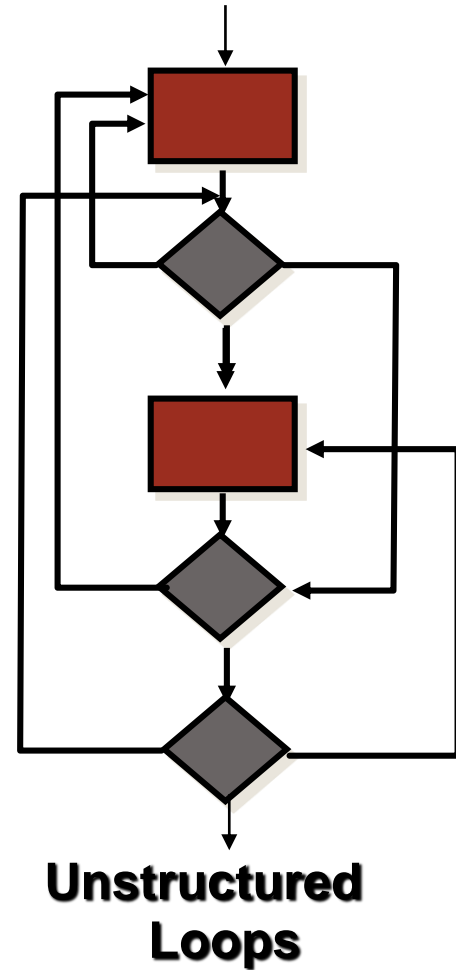
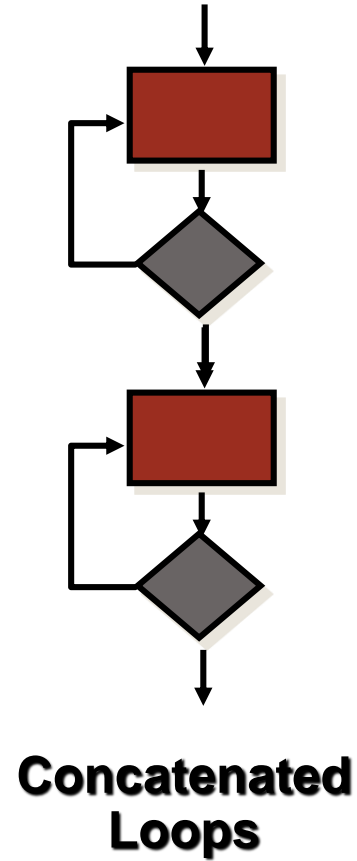
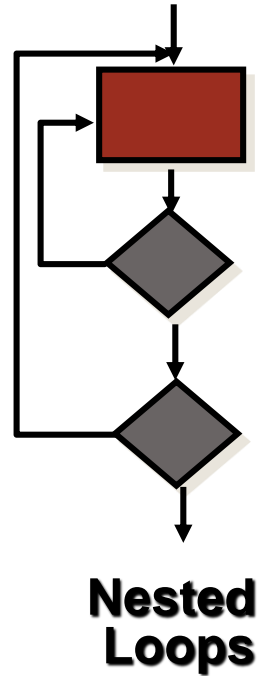
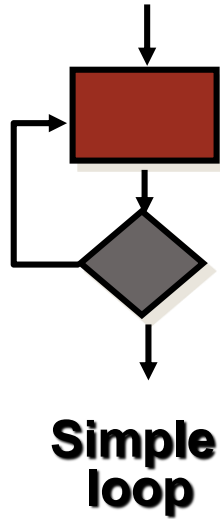
Connected to node		1	2	3	4	5
Node		1	2	3	4	5
1				a		
2						
3			d		b	
4			c			f
5			g	e		

*Graph 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

# 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.***

# Acceptance Testing

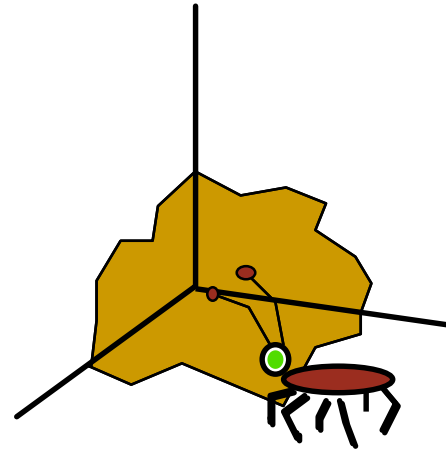
- Similar to validation testing except that customers are present or directly involved.
- Usually the tests are developed by the customer



# Test Case Design

"Bugs lurk in corners  
and congregate at  
boundaries ..."

*Boris Beizer*

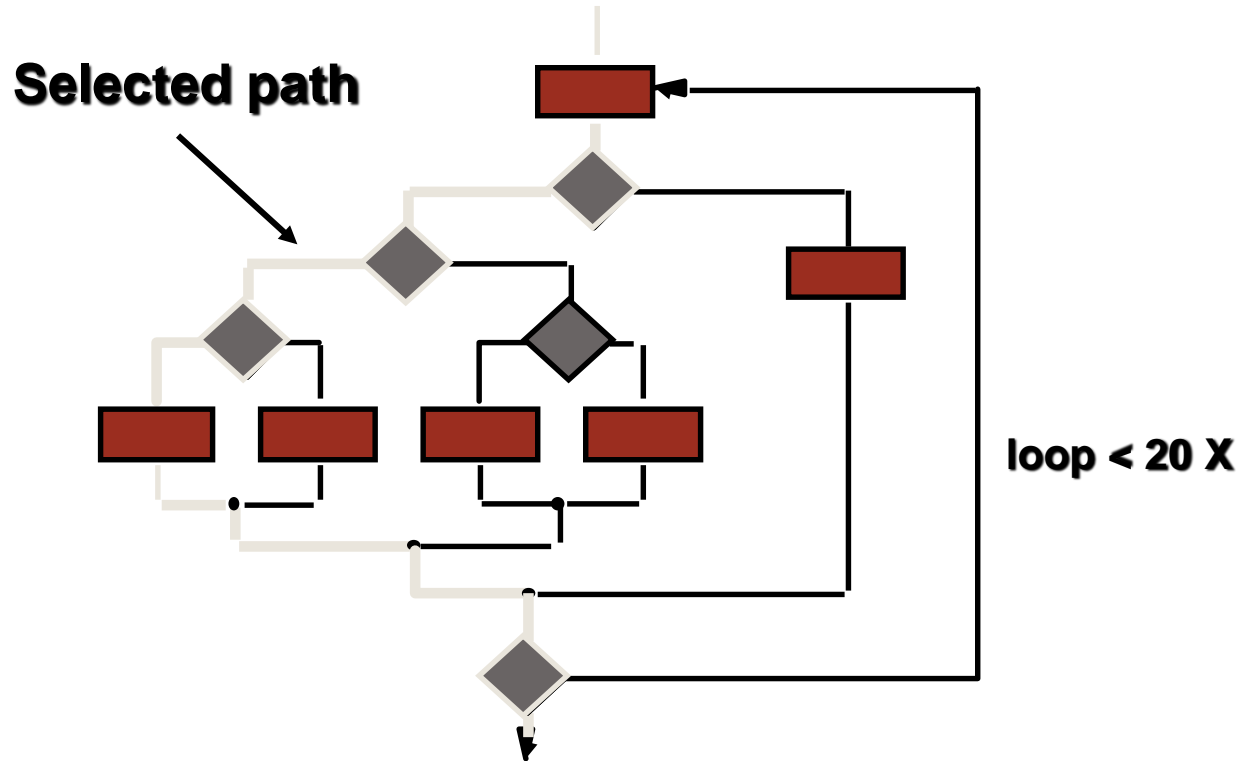


**OBJECTIVE**      to uncover errors

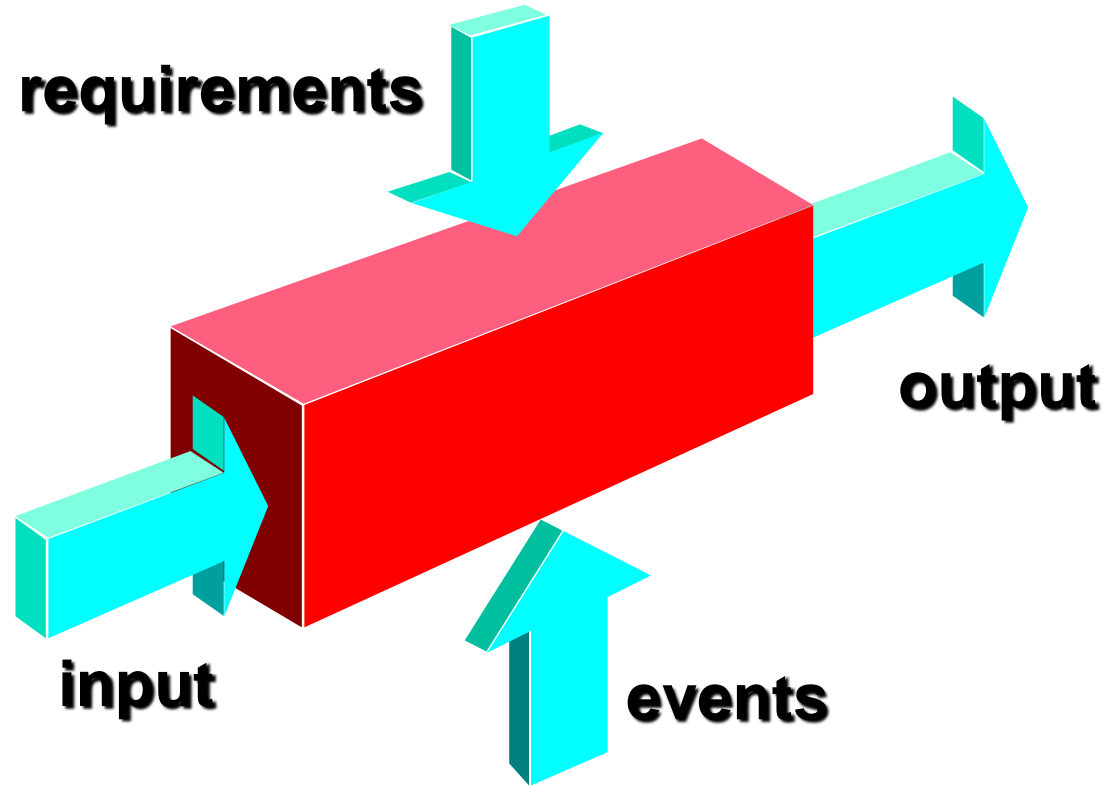
**CRITERIA**        in a complete manner

**CONSTRAINT**    with a minimum of effort and time

# Selective Testing



# Black-Box Testing



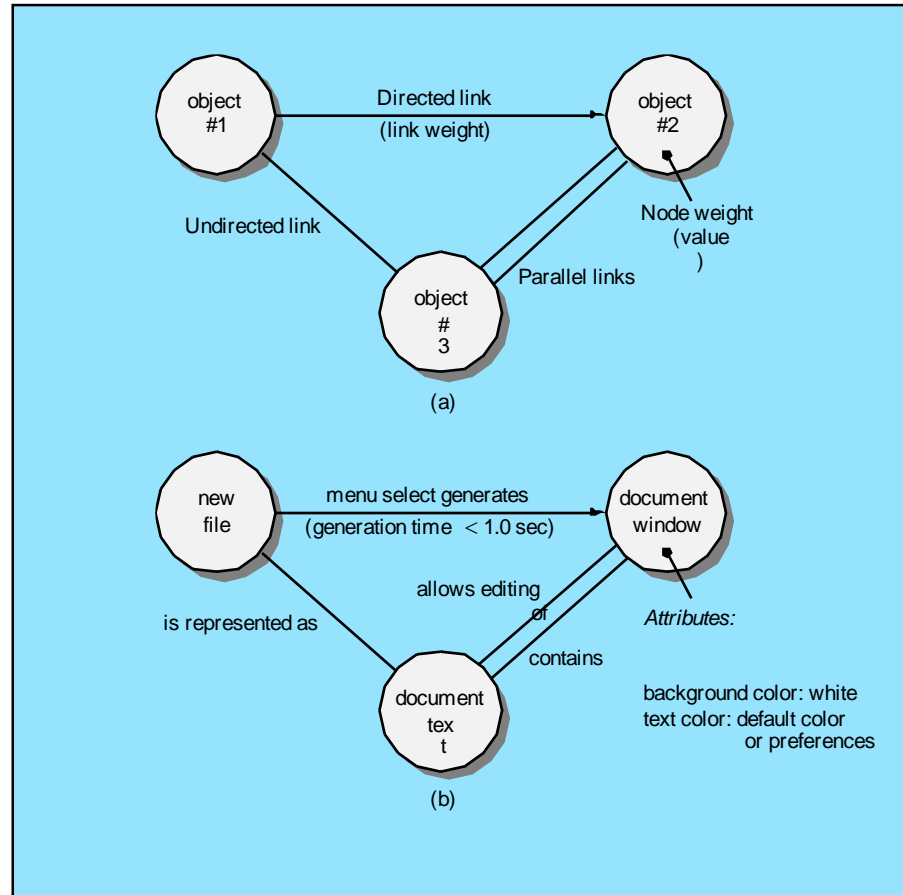
# Black-Box Testing

- 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?

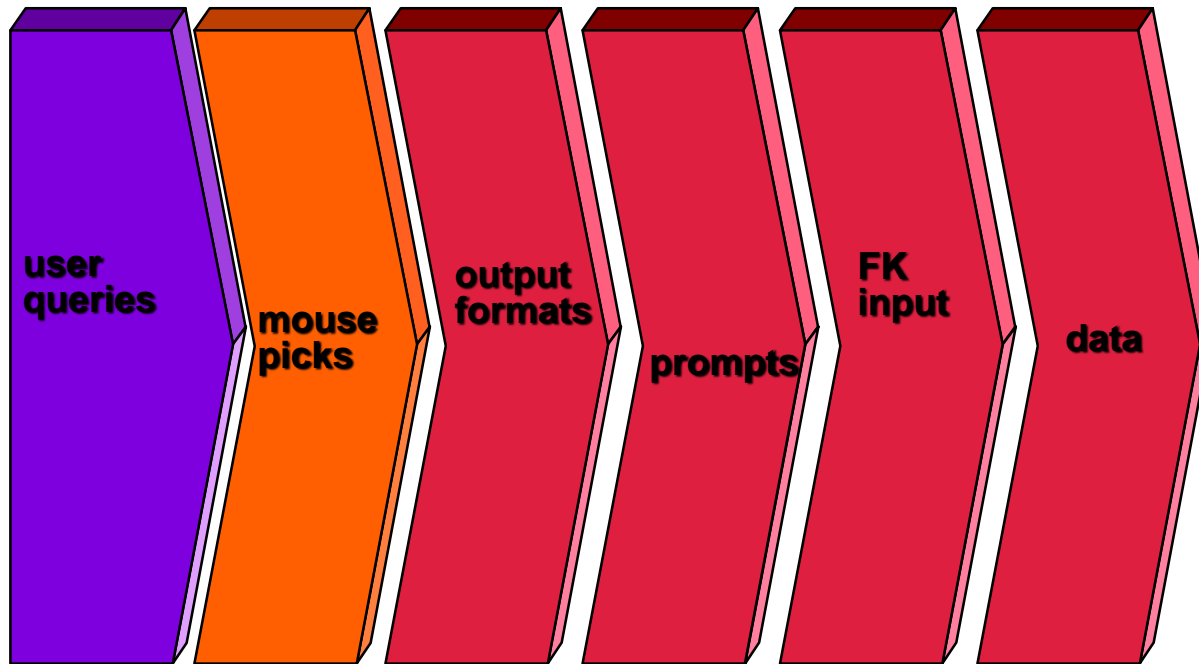
# Graph-Based Methods

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

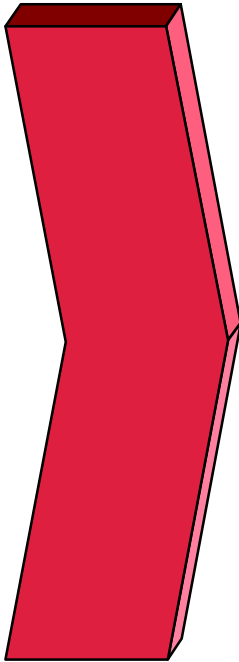
In this context, we consider the term “objects” in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



# Equivalence Partitioning



# Sample Equivalence Classes



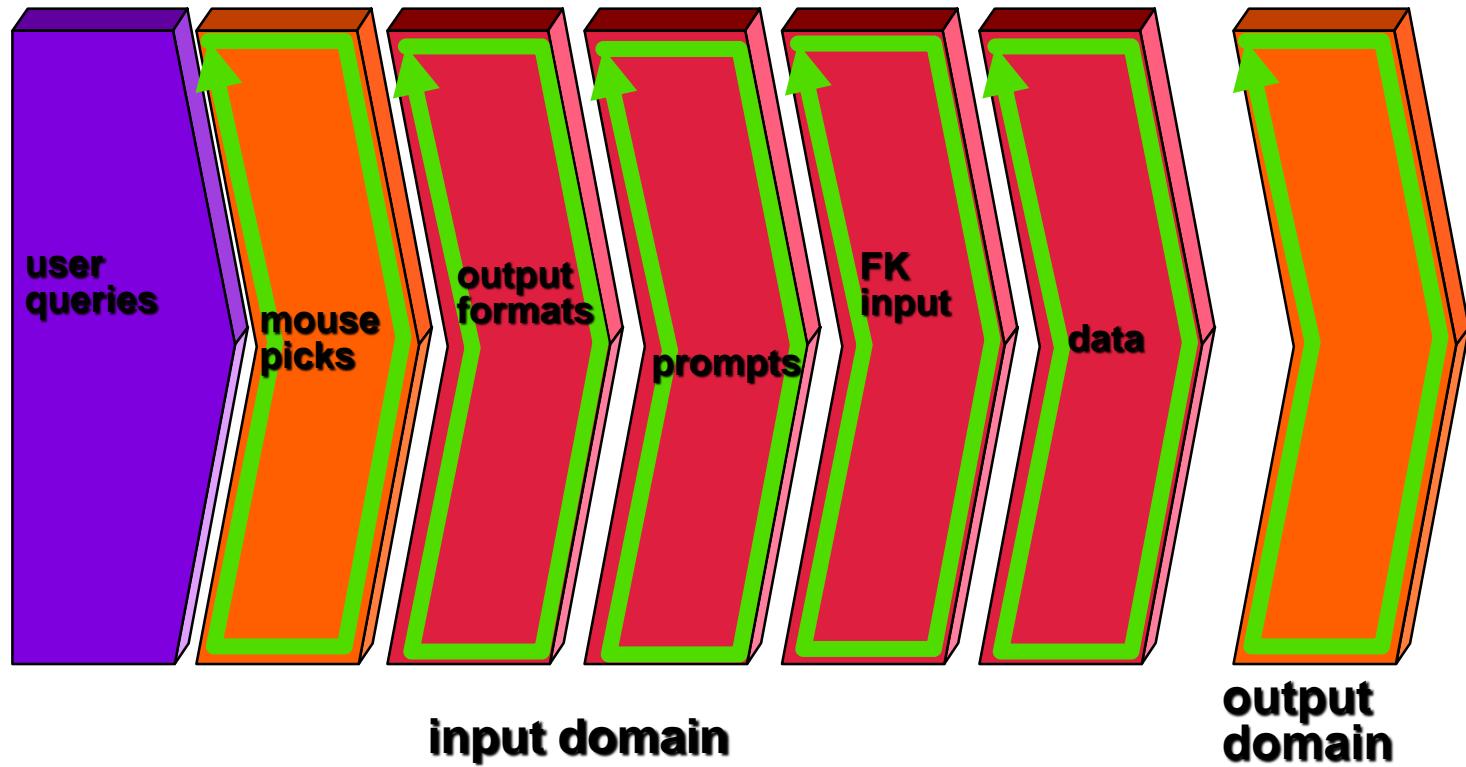
## **Valid data**

**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



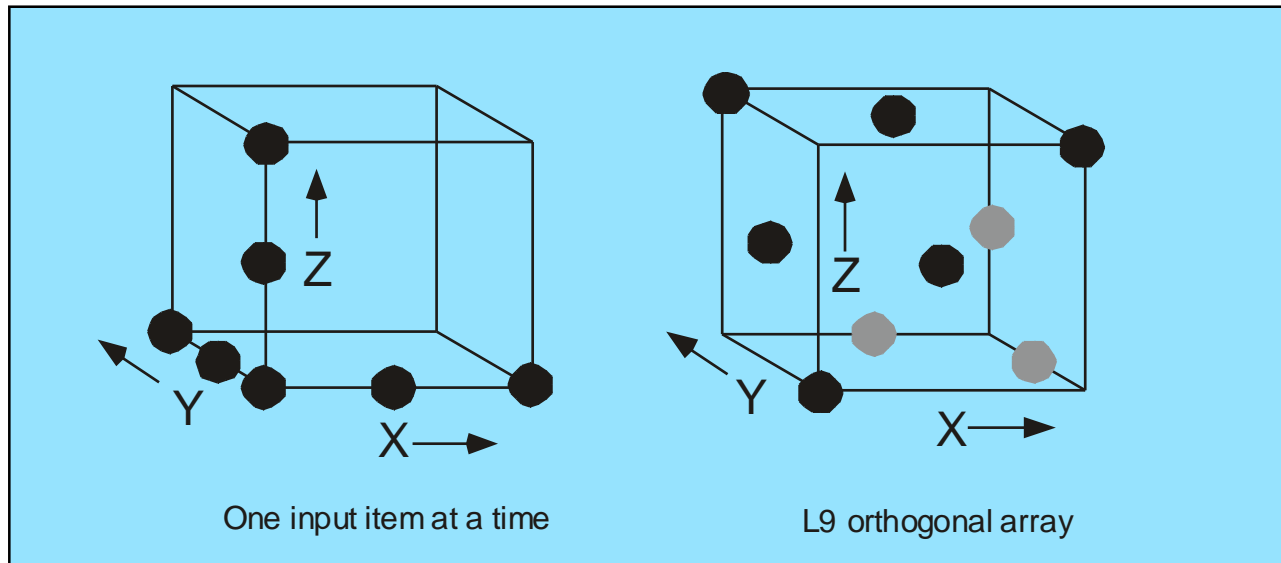


# Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

# Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



# 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.

# Testing for Specialized Environment, Architectures

- Testing GUIs
  - Reusable component testing
  - Finite state modeling graph techniques
- Testing of Client Server Architectures
  - Individual client testing (disconnected mode)
  - Individual client and server testing
  - Complete client server testing

# Contd..

- Test types during client server systems

Application function tests, Server tests, Database tests, Transaction tests, Network communication tests

- Testing Documentation and help facilities
  - Technical review
  - Live test (in conjunction with actual program)

# Contd

- Testing for the real time systems  
(Example Photo Copier Machine)
  - Task testing
  - Behavioral testing
  - Intertask testing
  - System testing

# Object-Oriented Testing

- begins by evaluating the correctness and consistency of the OOA and OOD 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

# 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).



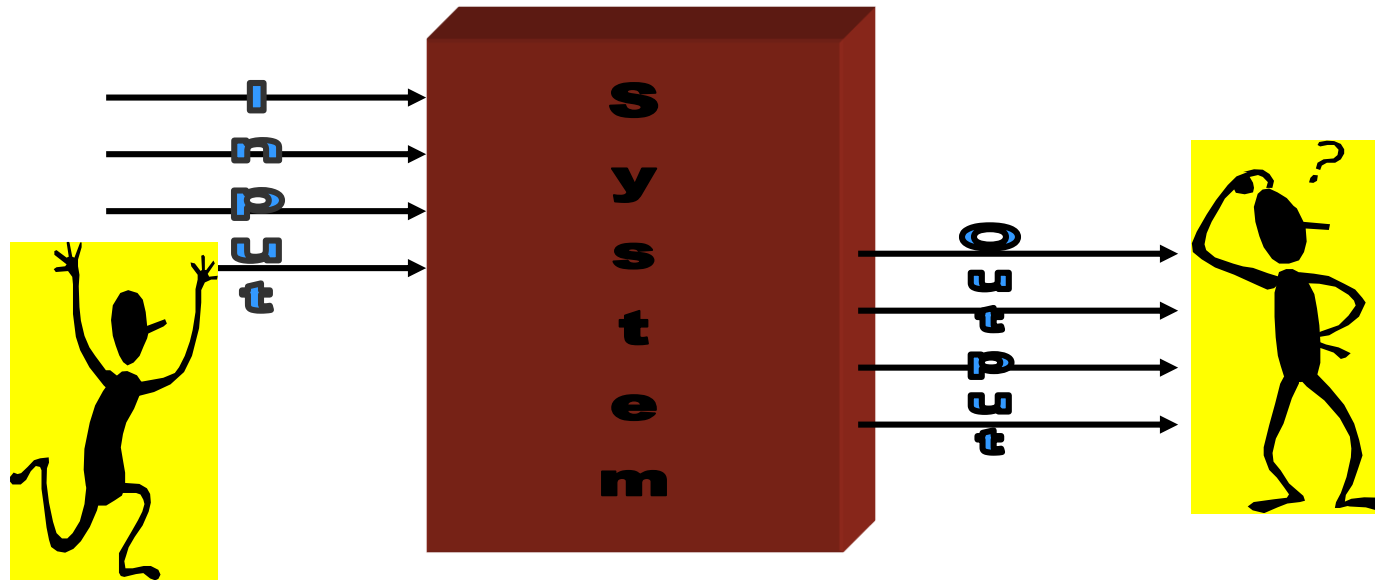
# OOT Strategy

- 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

# How do you test a system?

- Input test data to the system.
- Observe the output:
  - Check if the system behaved as expected.

# How do you test a system?



# How do you test a system?

- If the program does not behave as expected:
  - note the conditions under which it failed.
  - later debug and correct.

# Errors and Failures

- A failure is a manifestation of an error (aka defect or bug).

# Test cases and Test suite

- A **test case** is a triplet  $[I, S, O]$ :
  - I is the data to be input to the system,
  - S is the state of the system at which the data is input,
  - O is the expected output from the system.

# Verification versus Validation

- Verification is the process of determining:
  - whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining
  - whether a fully developed system conforms to its SRS document.

# Verification versus Validation

- Aim of Verification:
  - phase containment of errors
- Aim of validation:
  - final product is error free.



# Verification versus Validation

- Verification:
  - are we doing right?
- Validation:
  - have we done right?

# Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.
- Consider an example:
  - finding the maximum of two integers  $x$  and  $y$ .

# Design of Test Cases

- If  $(x > y)$   $\max = x$ ;  
    else  $\max = y$ ;
- The code has a simple error:
- test suite  $\{(x=3, y=2); (x=2, y=3)\}$  can detect the error,
- a larger test suite  $\{(x=3, y=2); (x=4, y=3); (x=5, y=1)\}$  does not detect the error.

# Design of Test Cases

- Two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

# Test Cases and the Class Hierarchy

- Subclasses may contain operations that are inherited from super classes
- Subclasses may contain operations that were redefined rather than inherited
- All classes derived from a previously tested base class need to be tested thoroughly

# Testing Concepts for WebApps

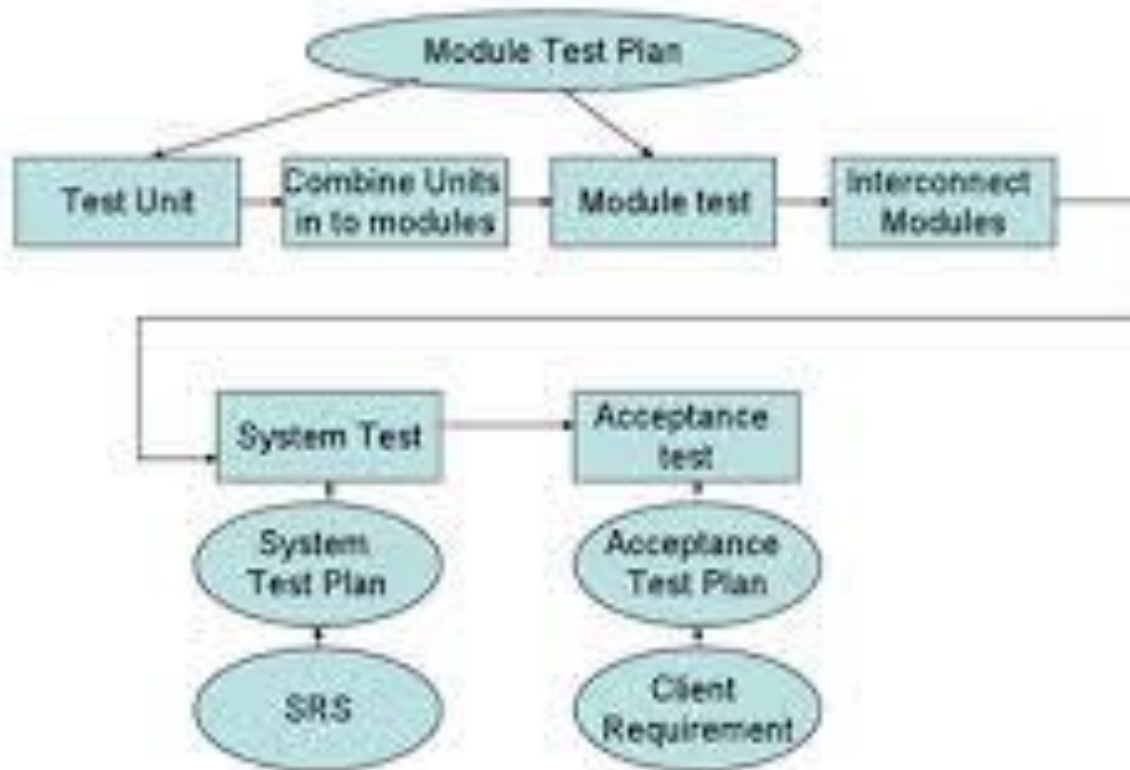
- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation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

# WebApp Testing

- 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.

# Testing Process—An Overview

## Testing Process





# GUI-based Testing: process

1. Identify the testing objective by defining a coverage criteria
2. Generate test cases from GUI structure, specification, model
  - Generate sequences of GUI events
  - Complete them with inputs and expected oracles
3. Define executable test cases
4. Run them and check the results

## GUI .... more formally

A GUI (**Graphical User Interface**) is a hierarchical, graphical front end to a software system

A GUI contains **graphical objects**  $w$ , called widgets, each with a set of properties  $p$ , which have discrete values  $v$  at run-time.

At any time during the execution, the values of the properties of each widget of a GUI define the GUI state:  $\{\dots (w, p, v), \dots\}$

A **graphical event**  $e$  is a state transducer, which yields the GUI from a state  $S$  to the next state  $S'$ .

# GUI-based Testing

- **Testing GUI software systems is different from testing non-GUI software**
- **Non-GUI testing:** suites are composed of test cases that invoke methods of the system and catch the return value/s;
- **GUI-based testing:** suites are composed of test cases that are:
  - able to recognize/identify the components of a GUI;
  - able to exercise GUI events (e.g., mouse clicks);
  - able to provide inputs to the GUI components (e.g., filling text fields);
  - able to test the functionality underlying a GUI set of components;
  - able to check the GUI representations to see if they are consistent with the expected ones;
  - often, strongly dependent on the used technology;

# Which type of GUI-based testing?

- **System testing**
  - Test the whole system
- **Acceptance testing**
  - Accept the system
- **Regression testing**
  - Test the system w.r.t. changes

# Approaches for GUI-based testing

- **Manual based**
  - Based on the domain and application knowledge of the tester
- **Capture and Replay**
  - Based on capture and replay of user sessions
- **Model-based testing**
  - Based on the execution of user sessions selected from a model of the GUI
    - Which type of model to use?
      - Event-based model
      - State-based model
      - Domain model
    - How do obtain the model to be used?
      - Specification-based model
      - Model recovered from existing software systems
      - Log-based model

# Coverage criteria for GUI-based testing

- Conventional code-based coverage cannot be adequate;
- GUIs are implemented in terms of event-based system, hence, the abstraction level is different w.r.t. the conventional system code. So mapping between GUI events and system code can not be so easy.
- Possible coverage criteria:
  - **Event-coverage:** all events of the GUI need to be executed at least once
  - **State-coverage:** “all states” of the GUI need to be exercised at least once
  - **Functionality-coverage:** .. using a functional point of view

# Event-based Model

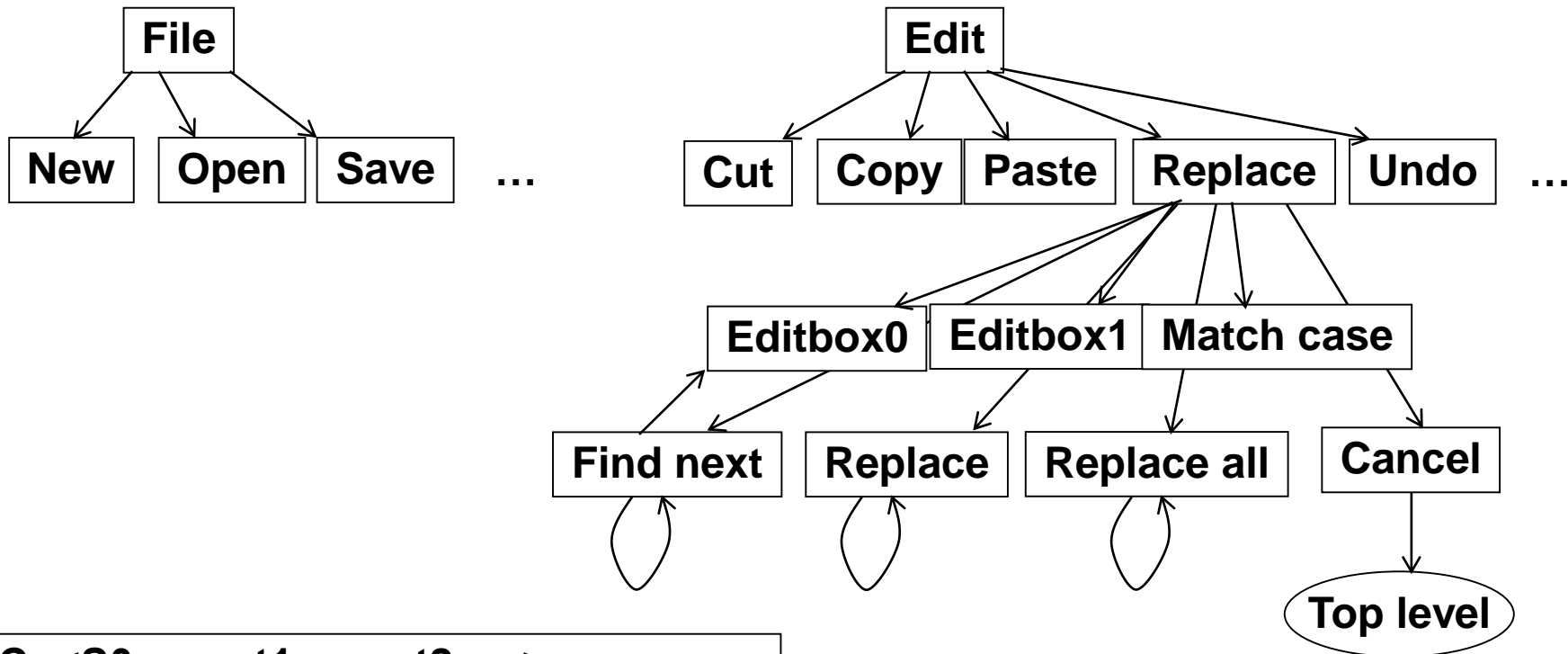
**Model the space of GUI event interactions as a graph**

**Given a GUI:**

- 1. create a graph model of all the possible sequences that a user can execute**
- 2. use the model to generate event sequences**

# Event-based Model

“Event-flow graph”



**TC:** <S0, event1, event2, ...>  
**Oracle:** <State1, State2, ...> & !CRASH



# Event-based Model

## Model Type:

- Complete event-model
- Partial event-model

## Event types:

- Structural events (*Edit, Replace*)
- Termination events (*Ok, cancel*)
- System interaction events (*Editbox0, Find next*)

## Coverage criteria

- Event coverage
- Event coverage according the exercised functionality
- Coverage of semantically interactive events
- 2-way, 3-way coverage
- ....

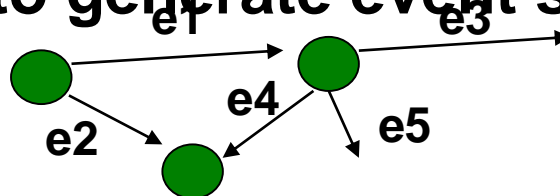
# State-based Model

**Model the space of GUI event interactions as a state model, e.g., by using a finite state machine (FSM):**

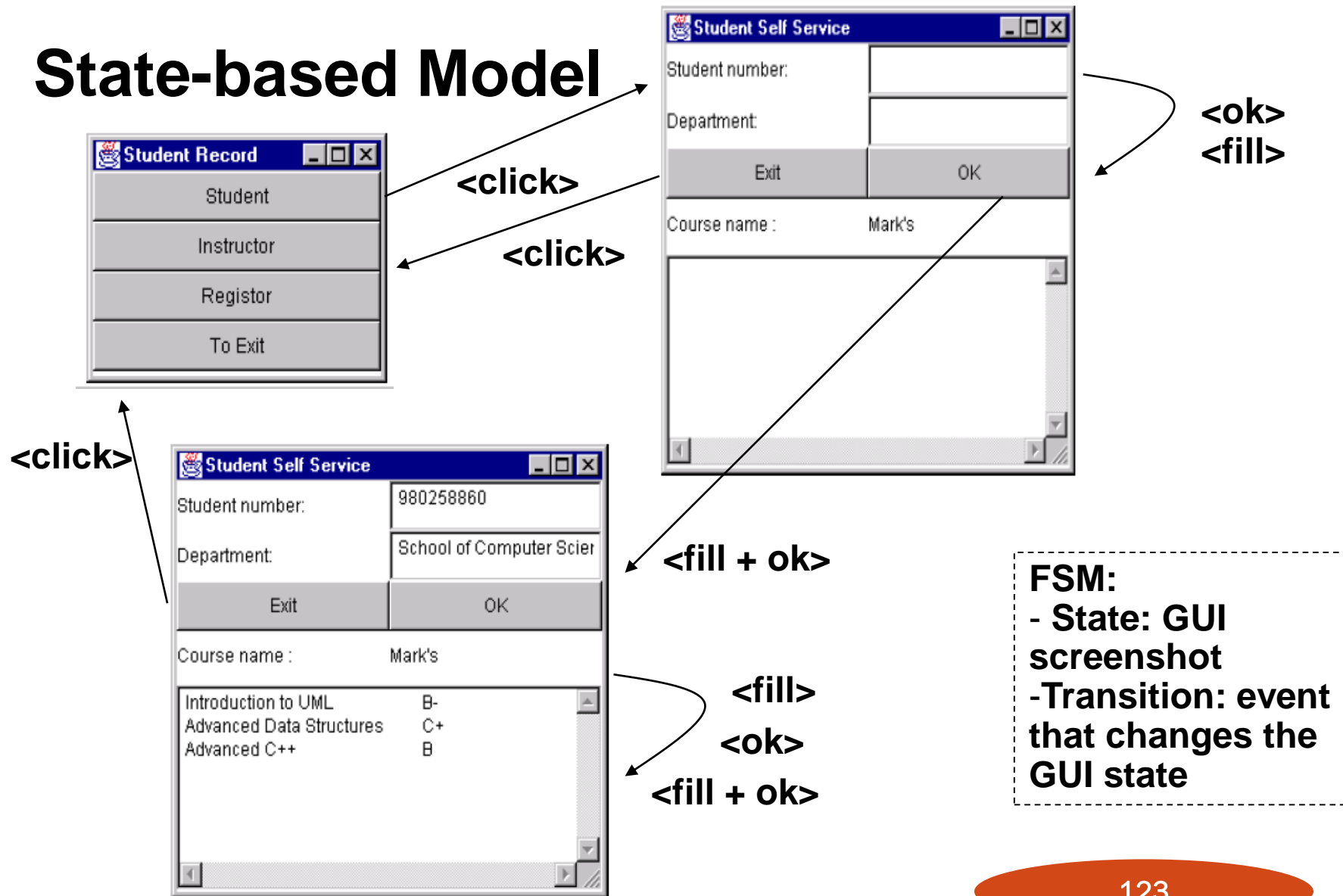
- **States are screenshot/representation of the GUI**
- **Transitions are GUI events that change the GUI state**

**Given a GUI:**

- 1. create a FSM of the possible sequences that a user can execute, considering the GUI state**
- 2. use the FSM to generate event sequences**



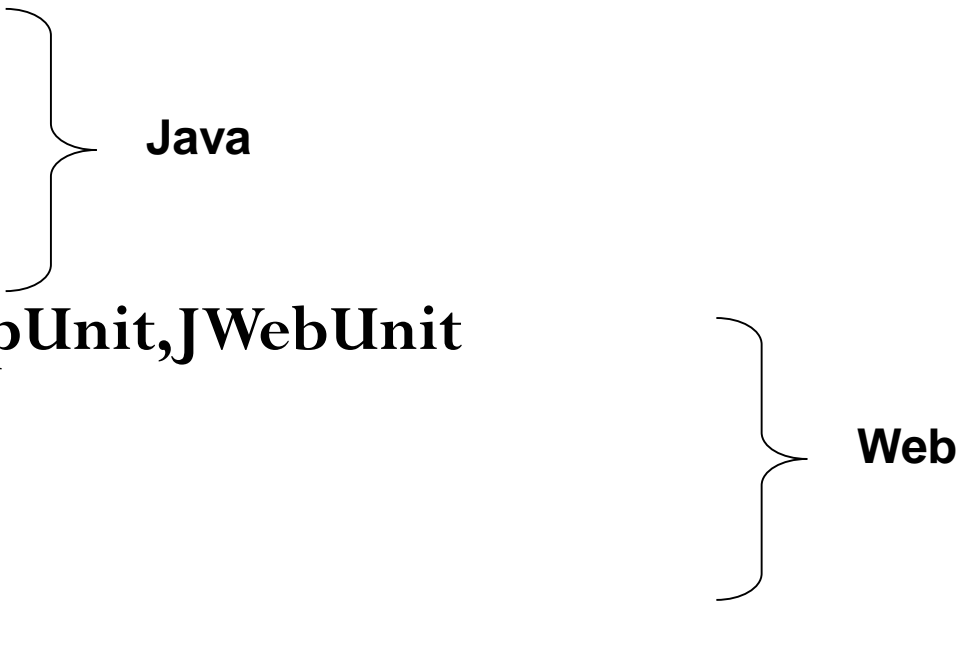
# State-based Model



# GUI errors: examples

- Incorrect functioning
- Missing commands (e.g., GUI events)
- Incorrect GUI screenshots/ states
- The absence of mandatory UI components (e.g., text fields and buttons)
- Incorrect default values for fields or UI objects
- Data validation errors
- Incorrect messages to the user, after errors
- Wrong UI construction
- ....

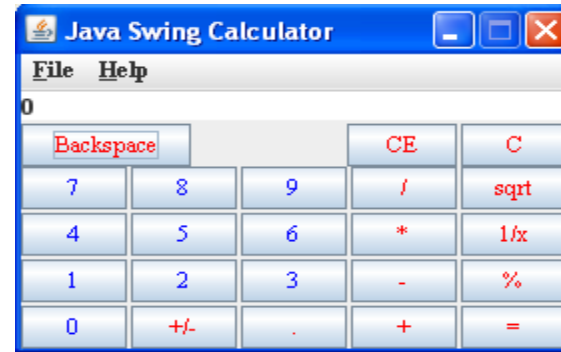
# Tools for GUI-based testing

- **Marathon**
  - **Abbot**
  - **Guitar**
  - **HtmlUnit, HttpUnit, JWebUnit**
  - **HtmlFixture**
  - **Selenium**
  - **....**
- 
- The diagram uses curly braces to group the tools into two categories. A brace on the right side of the first three items (Marathon, Abbot, Guitar) points to the label 'Java'. A larger brace on the right side of the last four items (HtmlUnit, HttpUnit, JWebUnit, HtmlFixture, Selenium, ....) points to the label 'Web'.
- Java**
- Web**

# Running Example: Calculator

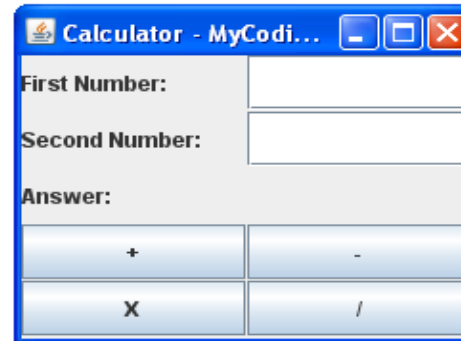
Calc\_1 :

- Logic mixed to GUI
- GUI realized by using Swing



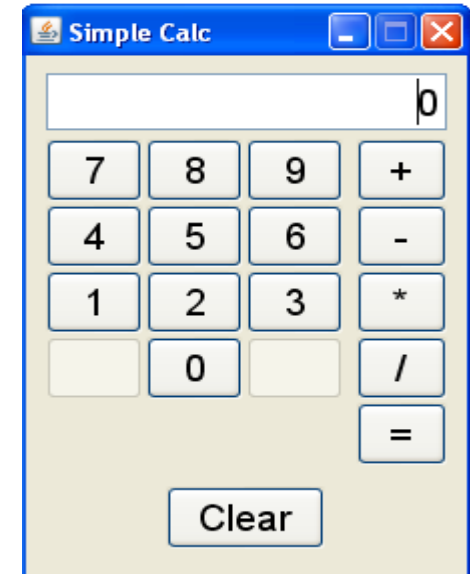
Calc\_2 :

- Logic mixed to GUI
- GUI realized by using AWT



Calc\_3 :

- Logic separated from the GUI
- GUI realized by using AWT + Swing



# Quality

■ Quality means “conformance to requirements”

- ▷ The best testers can only catch defects that are contrary to specification.
- ▷ Testing does not make the software perfect.
- ▷ If an organization does not have good requirements engineering practices then it will be very hard to deliver software that fills the users’ needs, because the product team does not really know what those needs are.

# Test Plans

- The goal of test planning is to establish the list of tasks which, if performed, will identify all of the requirements that have not been met in the software. The main work product is the *test plan*.
  - ▷ The test plan documents the overall approach to the test. In many ways, the test plan serves as a summary of the test activities that will be performed.
  - ▷ It shows how the tests will be organized, and outlines all of the testers' needs which must be met in order to properly carry out the test.
  - ▷ The test plan should be inspected by members of the engineering team and senior managers.



# Test Plan Outline

## *Purpose*

A description of the purpose of the application under test.

## *Features to be tested*

A list of the features in the software that will be tested. It is a catalog of all of the test cases (including a test case number and title) that will be conducted, as well as all of the base states.

## *Features not to be tested*

A list of any areas of the software that will be excluded from the test, as well as any test cases that were written but will not be run.

## *Approach*

A description of the strategies that will be used to perform the test.

## *Suspension criteria and resumption requirements*

Suspension criteria are the conditions that, if satisfied, require that the test be halted. Resumption requirements are the conditions that are required in order to restart a suspended test.

## *Environmental Needs*

A complete description of the test environment or environments. This should include a description of hardware, networking, databases, software, operating systems, and any other attribute of the environment that could affect the test.

## *Schedule*

An estimated schedule for performing the test. This should include milestones with specific dates.


## *Acceptance criteria*

Any objective quality standards that the software must meet, in order to be considered ready for release. This may include things like stakeholder sign-off and consensus, requirements that the software must have been tested under certain environments, minimum defect counts at various priority and severity levels, minimum test coverage numbers, etc.


## *Roles and responsibilities*

A list of the specific roles that will be required for people in the organization, in order to carry out the test. This list can indicate specific people who will be testing the software and what they are responsible for.

# Test Cases

 A *test case* is a description of a specific interaction that a tester will have in order to test a single behavior of the software. Test cases are very similar to use cases, in that they are step-by-step narratives which define a specific interaction between the user and the software.

- ▷ A typical test case is laid out in a table, and includes:
  - A unique *name* and *number*
  - A *requirement* which this test case is exercising
  - *Preconditions* which describe the state of the software before the test case (which is often a previous test case that must always be run before the current test case)
  - *Steps* that describe the specific steps which make up the interaction
  - *Expected Results* which describe the expected state of the software after the test case is executed

 Test cases must be repeatable.

- ▷ Good test cases are data-specific, and describe each interaction necessary to repeat the test exactly.

# Test Cases – Good Example

Name	TC-47: Verify that lowercase data entry results in lowercase insert
Requirement	FR-4 (Case sensitivity in search-and-replace), bullet 2
Preconditions	The test document TESTDOC.DOC is loaded (base state BS-12).
Steps	<ol style="list-style-type: none"><li>1. Click on the “Search and Replace” button.</li><li>2. Click in the “Search Term” field.</li><li>3. Enter <i>This is the Search Term</i>.</li><li>4. Click in the “Replacement Text” field.</li><li>5. Enter <i>This IS THE Replacement TeRM</i>.</li><li>6. Verify that the “Case Sensitivity” checkbox is unchecked.</li><li>7. Click the OK button.</li></ol>
Expected results	<ol style="list-style-type: none"><li>1. The search-and-replace window is dismissed.</li><li>2. Verify that in line 38 of the document, the text <i>this is the search term</i> has been replaced by <i>this is the replacement term</i>.</li><li>3. Return to base state BS-12.</li></ol>

# Test Cases – Bad Example

Steps	<ol style="list-style-type: none"><li>1. Bring up search-and-replace.</li><li>2. Enter a lowercase word from the document in the search term field.</li><li>3. Enter a mixed-case word in the replacement field.</li><li>4. Verify that case sensitivity is not turned on and execute the search.</li></ol>
Expected results	<ol style="list-style-type: none"><li>1. Verify that the lowercase word has been replaced with the mixed-case term in lowercase.</li></ol>

# Test Execution

- The software testers begin executing the test plan after the programmers deliver the *alpha build*, or a build that they feel is feature complete.
  - ▷ The alpha should be of high quality—the programmers should feel that it is ready for release, and as good as they can get it.
- There are typically several iterations of test execution.
  - ▷ The first iteration focuses on new functionality that has been added since the last round of testing.
  - ▷ A *regression test* is a test designed to make sure that a change to one area of the software has not caused any other part of the software which had previously passed its tests to stop working.
  - ▷ Regression testing usually involves executing all test cases which have previously been executed.
  - ▷ There are typically at least two regression tests for any software project.

# Test Execution



When is testing complete?

- No defects found
- Or defects meet acceptance criteria outlined in test plan

*TABLE 8-6 . Acceptance criteria from a test plan*

1. Successful completion of all tasks as documented in the test schedule.
2. Quantity of medium- and low-level defects must be at an acceptable level as determined by the software testing project team lead.
3. User interfaces for all features are functionally complete.
4. Installation documentation and scripts are complete and tested.
5. Development code reviews are complete and all issues addressed. All high-priority issues have been resolved.
6. All outstanding issues pertinent to this release are resolved and closed.
7. All current code must be under source control, must build cleanly, the build process must be automated, and the software components must be labeled with correct version numbers in the version control system.
8. All high-priority defects are corrected and fully tested prior to release.
9. All defects that have not been fixed before release have been reviewed by project stakeholders to confirm that they are acceptable.
10. The end user experience is at an agreed acceptable level.
11. Operational procedures have been written for installation, set up, error recovery, and escalation.
12. There must be no adverse effects on already deployed systems.

# Positive Testing

- **Positive testing** can be performed on the system by providing the **valid data as input**. It checks whether an application behaves as expected with the positive input. . This is to test to check the application that does what it is supposed to do so

- For example – **Enter Only Numbers**

A rectangular text box with a blue border. Inside the box, the number '99999' is displayed in a blue font.

## Positive Testing

- There is a text box in an application which can accept only numbers. Entering values up to 99999 will be acceptable by the system and any other values apart from this should not be acceptable. To do positive testing, set the valid input values from 0 to 99999 and check whether the system is accepting the values.

# Negative Testing

- **Negative Testing** can be performed on the system by providing **invalid data as input**. It checks whether an application behaves as expected with the negative input. This is to test the application that does not do anything that it is not supposed to do so. For example -

Enter Only Numbers

## Negative Testing

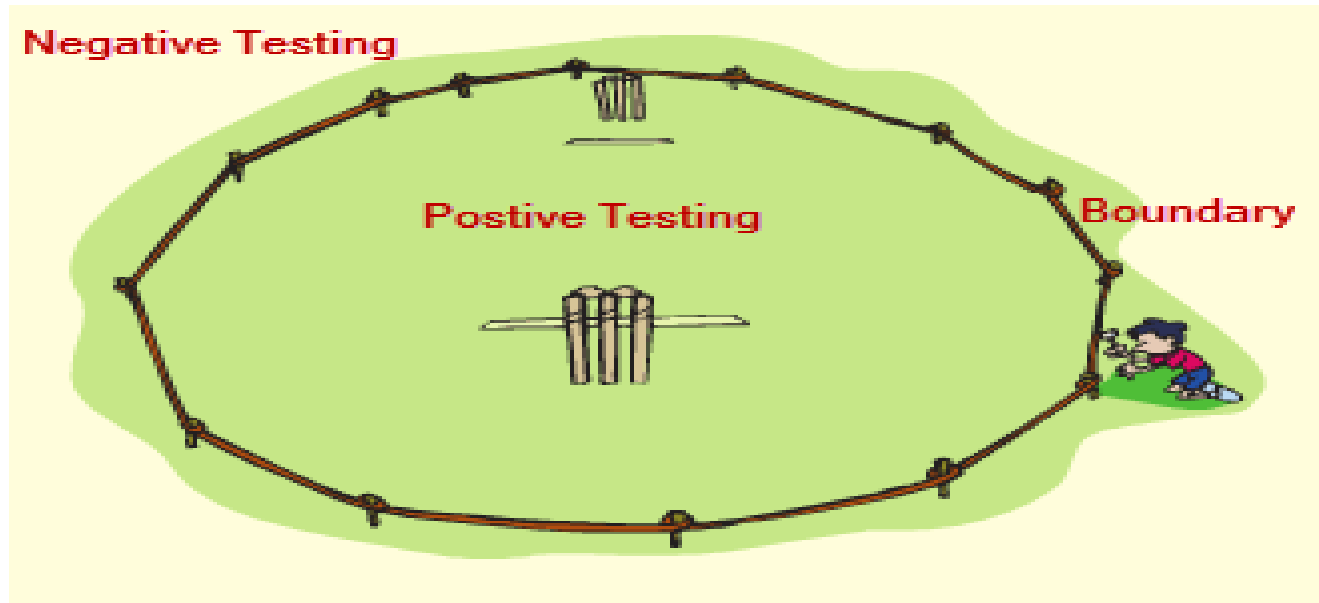
- For the example above Negative testing can be performed by testing by entering alphabets characters from A to Z or from a to z. Either system text box should not accept the values or else it should throw an error message for these invalid data inputs.



# Testing Technique used for Positive and Negative Testing:

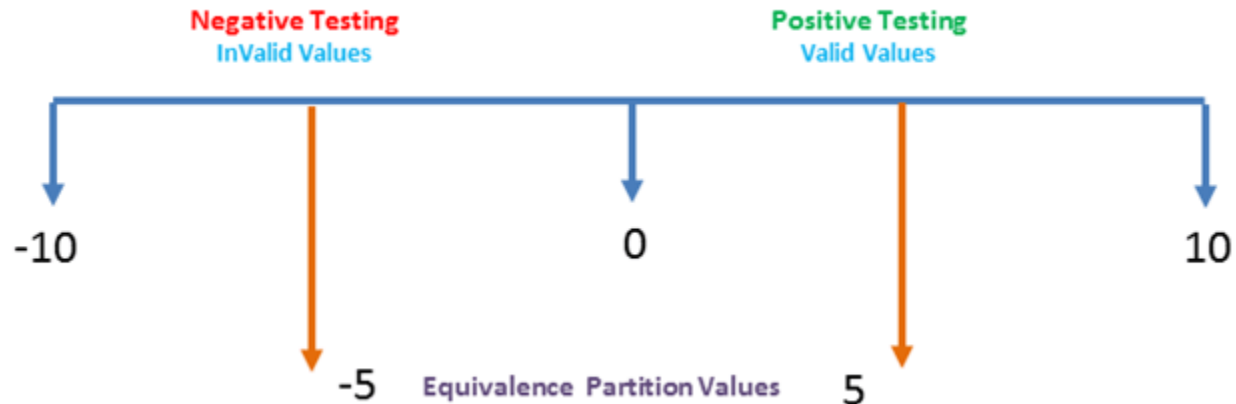
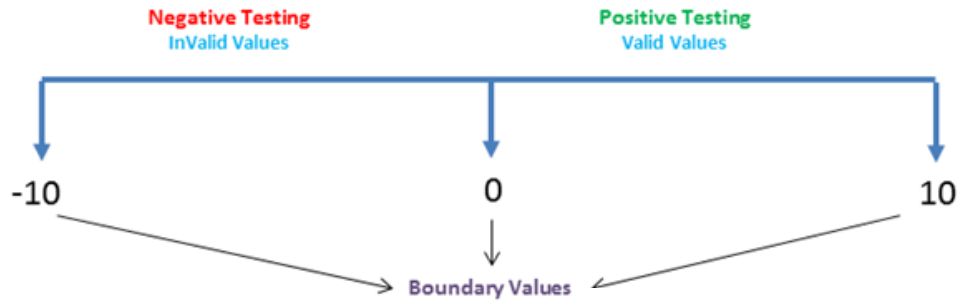
- Following techniques are used for Positive and negative validation of testing is:

Boundary Value Analysis



# Testing Technique used for Positive and Negative Testing:

- Equivalence Partitioning



# Conclusion

- Testing helps deliver quality software application and ensures the software is bug free before the software is launched. For effective testing, use both - Positive and Negative testing which give enough confidence in the quality of the software. Real time users are can input any values and those needs to be tested before release.