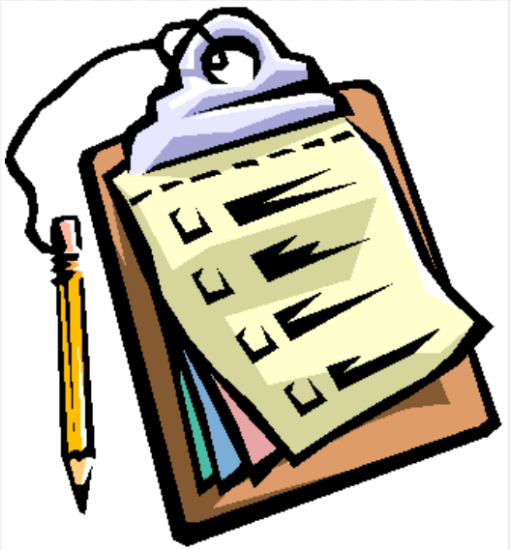
Software Testing

Quality Management: Chapters 22 & 23

Faezeh Gohari

Agenda

- Software Testing Concepts
- Unit Testing
- Integration Testing
- System Testing
- Debugging



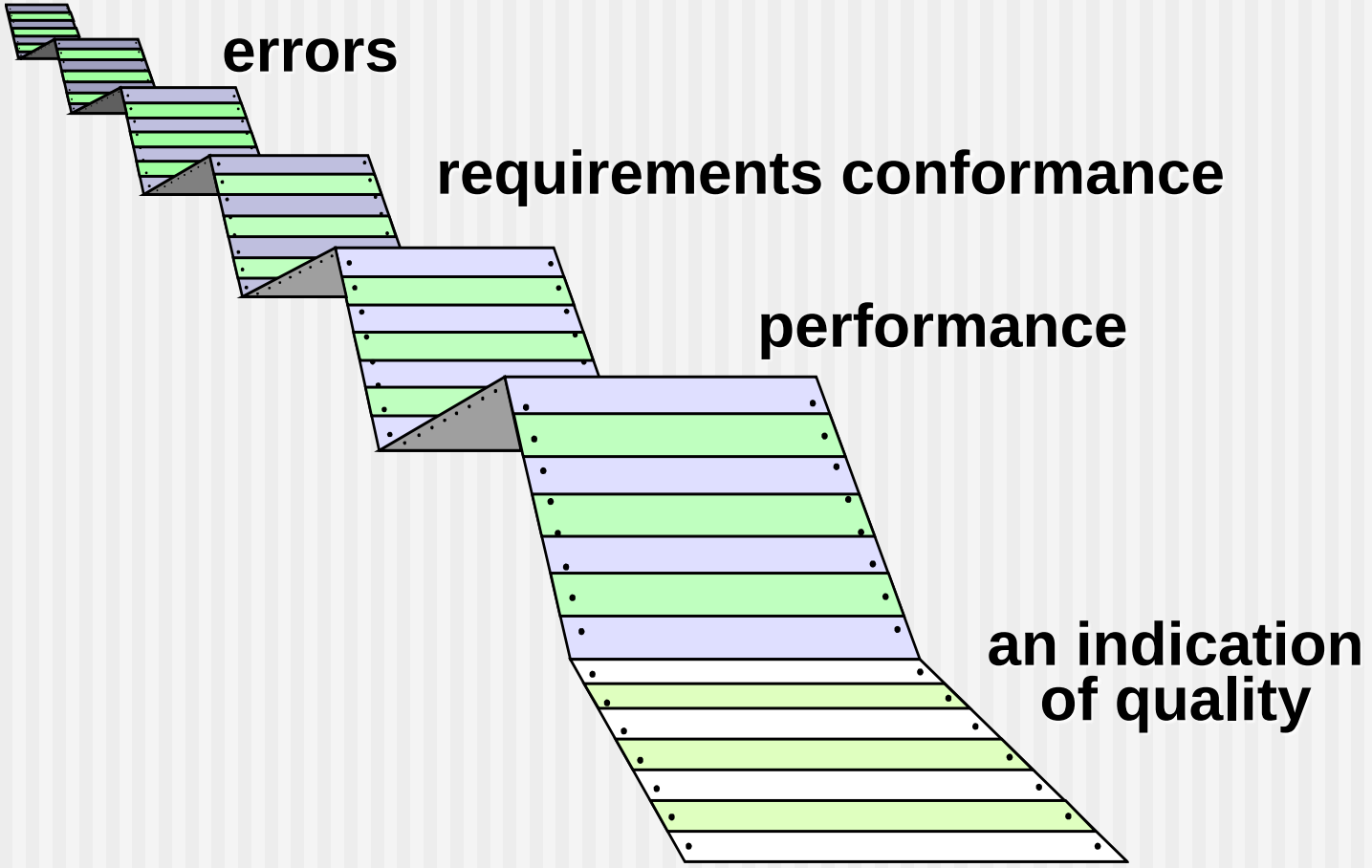
Software Testing Concepts

Software Testing

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

- Testing provides greater assurance that the software is of high **quality** and **reliability**
- Software is **not completed** until it is fully tested from different perspectives
  - Functional characteristics
  - Non-functional characteristics
    - performance, security, usability, …

What Testing Shows



Costly Software Failures

- Boeing A220 : Engines failed after software update allowed excessive vibrations
- Toyota brakes : Dozens dead, thousands of crashes

- Healthcare website : Crashed repeatedly on launch—never load tested
- Ariane 5 explosion: Millions of $$

We need our software to be dependable

Testing is *one* way to assess dependability

V & V

- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

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

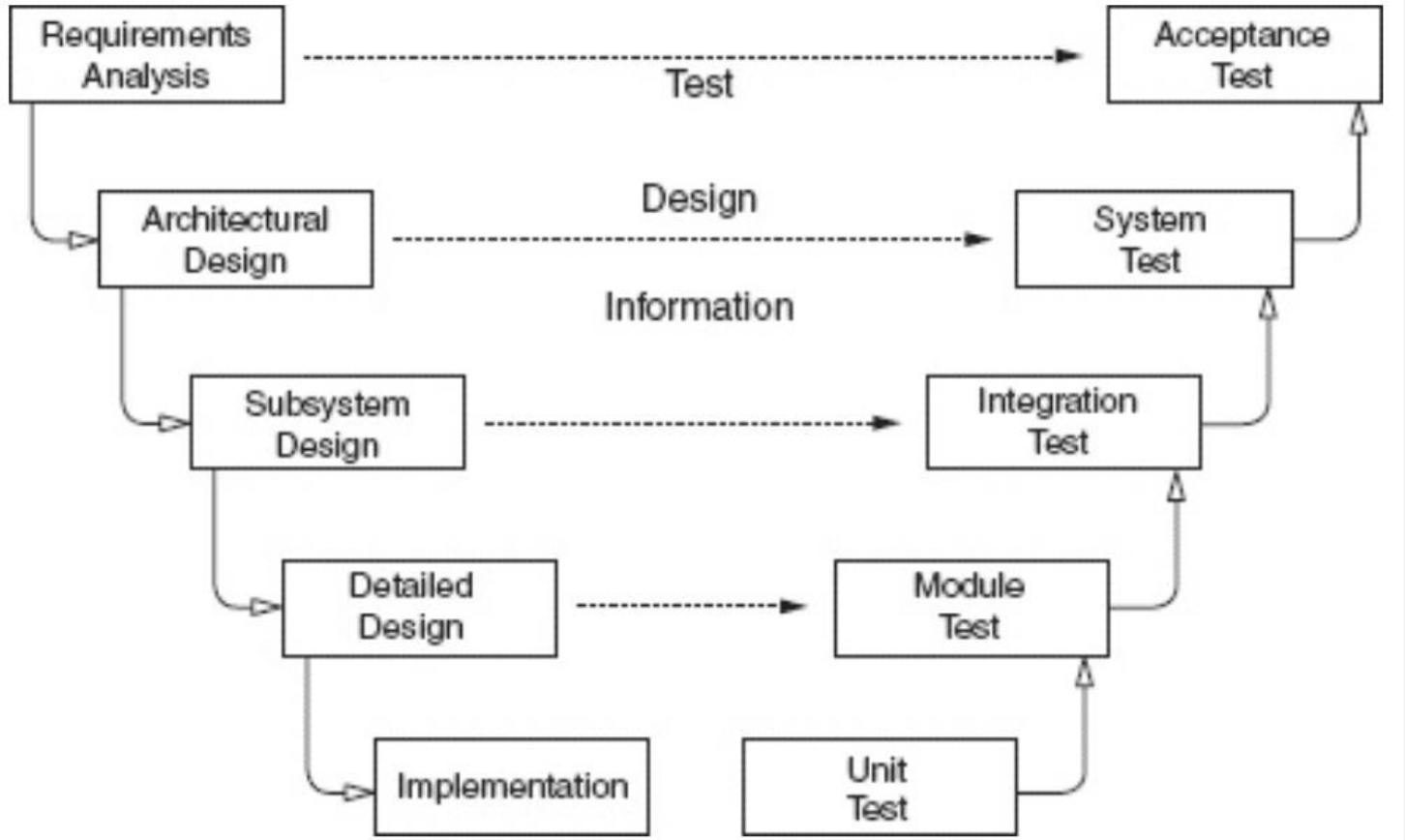
Independent Test Group (ITG)

- The software **developer** is **always responsible** for testing the individual **units**
- In many cases, the developer also conducts **integration testing**
- Only after the software architecture is complete does an **independent test group** become involved
  - To find hidden errors
- The developer and the ITG **work closely** throughout a software project to ensure that thorough tests will be conducted
- While testing is conducted, the developer must be available to correct errors that are uncovered

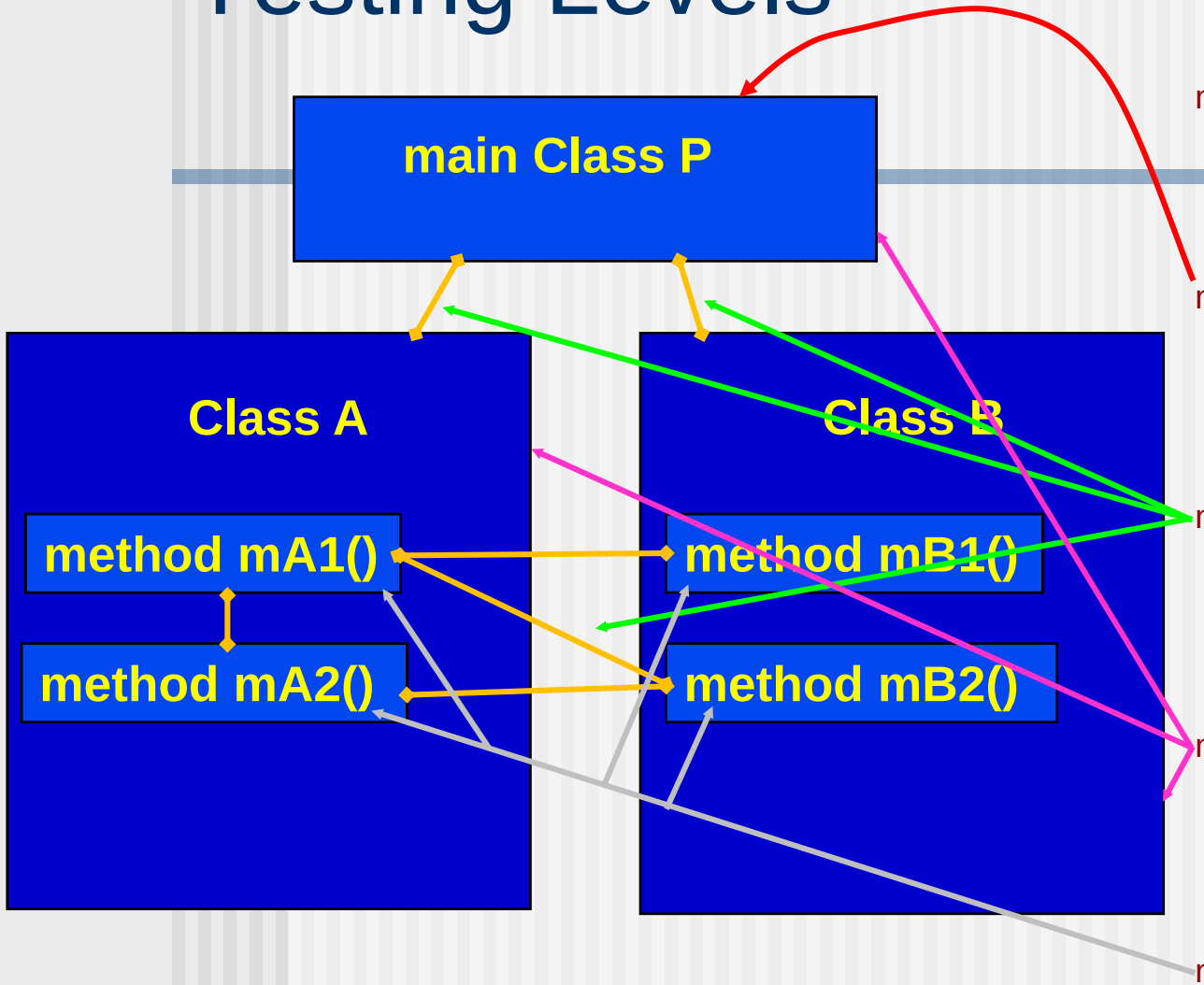
Software Testing Dimensions

- Level of testing
  - Unit test, Integration test, System test, Acceptance test
- Method of testing
  - White-box or Black-box
- Type of testing
  - Functional or Non-functional tests
- Execution manner
  - Automated or Manual
- Tester Role
  - Developer, QA, end-user, …

Testing Levels
Based on Software Activities



Testing Levels



- Acceptance testing : Is the software acceptable to the user?
- System testing : Test the overall functionality of the system
- Integration testing : Test how modules interact with each other
- Module testing (developer testing) : Test each class, file, module, component
- Unit testing (developer testing) : Test each unit (method) individually

Acceptance Test

- **Acceptance tests** are a series of specific tests **conducted by the customer** in an attempt to uncover product errors before accepting the software from the developer
- When a software product is built for one customer, it is reasonable for that person to conduct a series to validate all requirements
- If software is developed to be used by many customers, it is impractical to allow each user to perform formal acceptance tests
- Most software product builders use a process called **alpha and beta testing** to uncover **errors that only end users seem able to find**

Alpha and Beta Testing

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

Alpha and Beta Testing

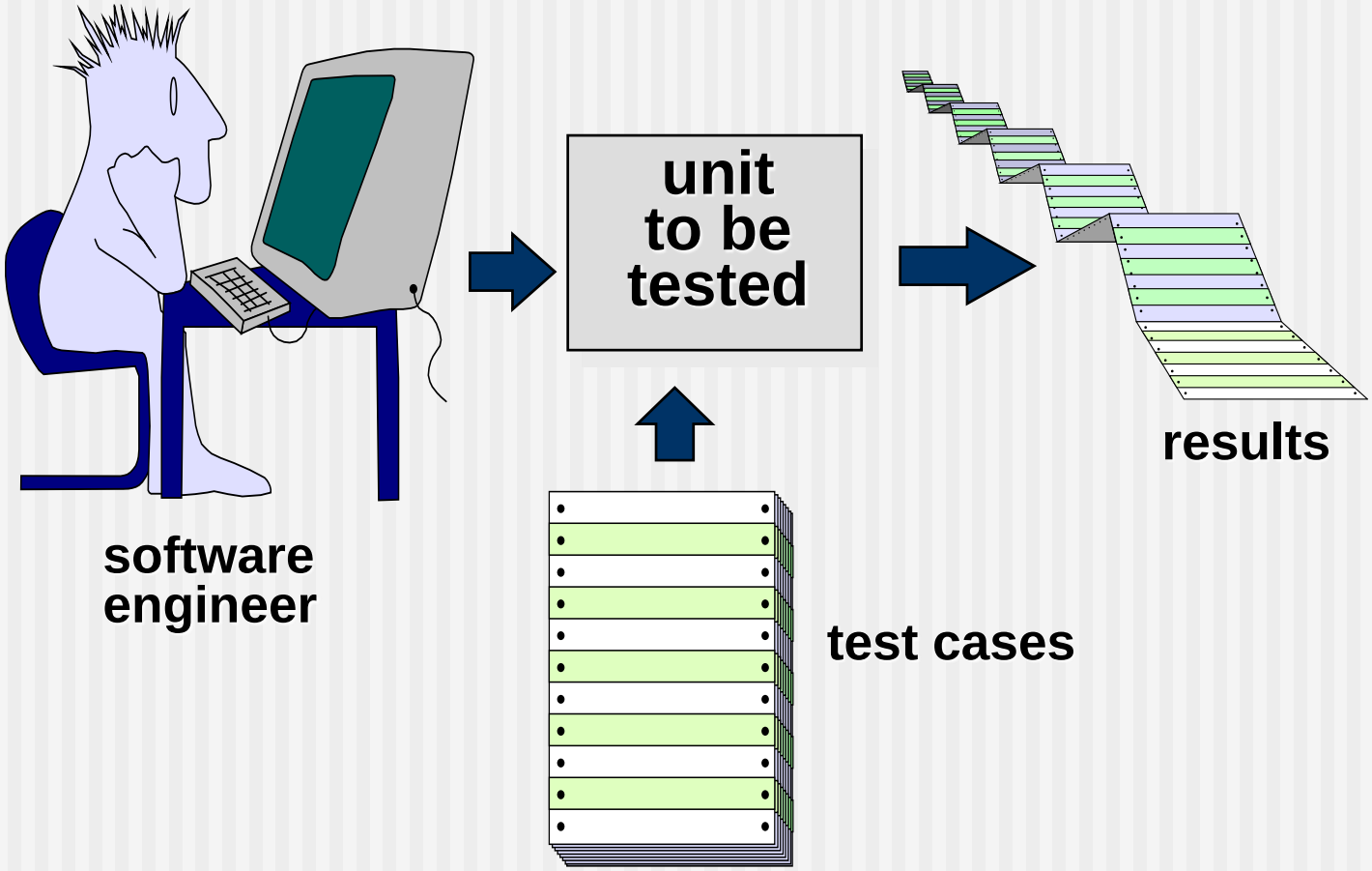
- **Beta testing:**
  - Conducted at one or more **end-user sites**
  - The developer generally is not present
  - A "**live**" application of the software in an environment **not controlled by the developer**
  - The **customer records all problems** (real or imagined) that are encountered during beta testing and reports these at regular intervals
  - The developer makes modifications and then **prepares for release** of the software product to the **entire customer base**

Colored Boxes

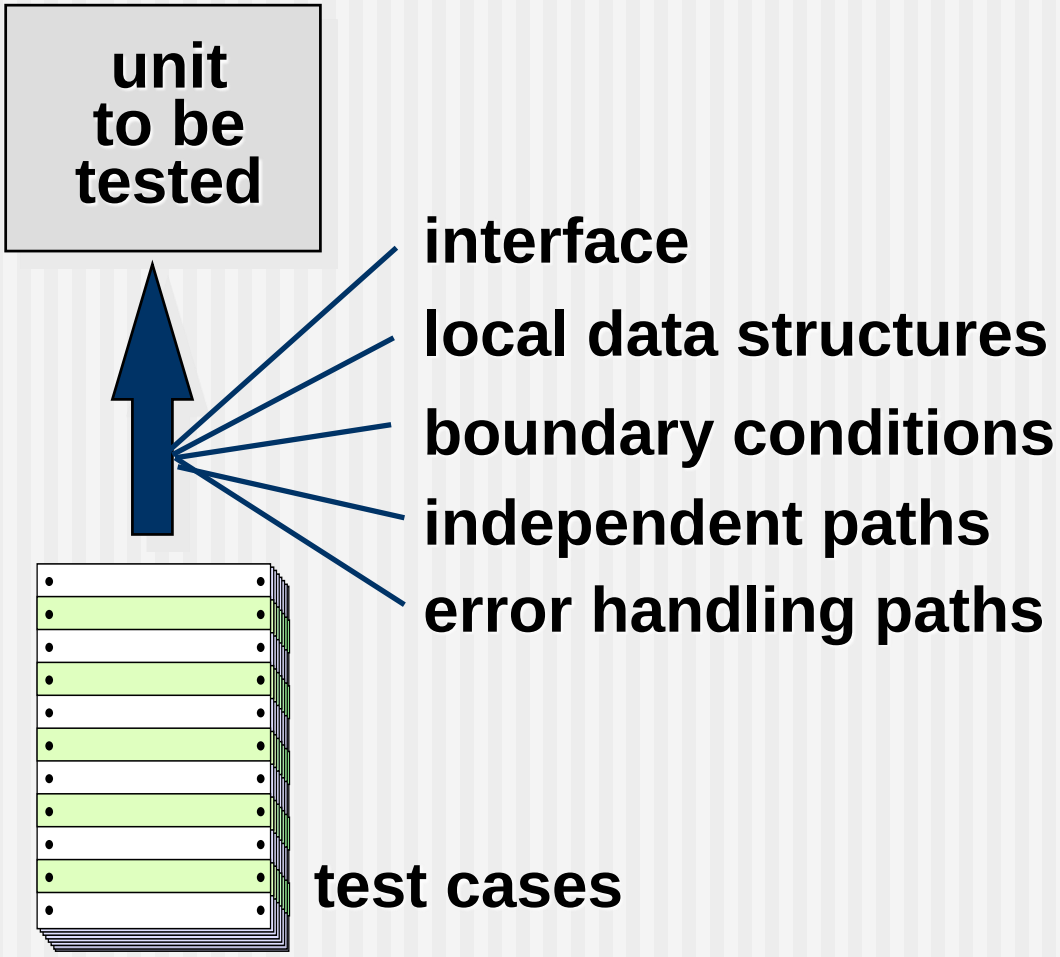
- **Black-box testing:**
  - Derive tests from **external descriptions** of the software, including specifications, requirements, and design
- **White-box testing:**
  - Derive tests from the **source code internals** of the software
    1. All independent paths within a module are exercised at least once
    2. All logical decisions on their true and false sides
    3. All loops at their boundaries and within their bounds
    4. All internal data structures to ensure their validity

Unit Testing

Unit Testing



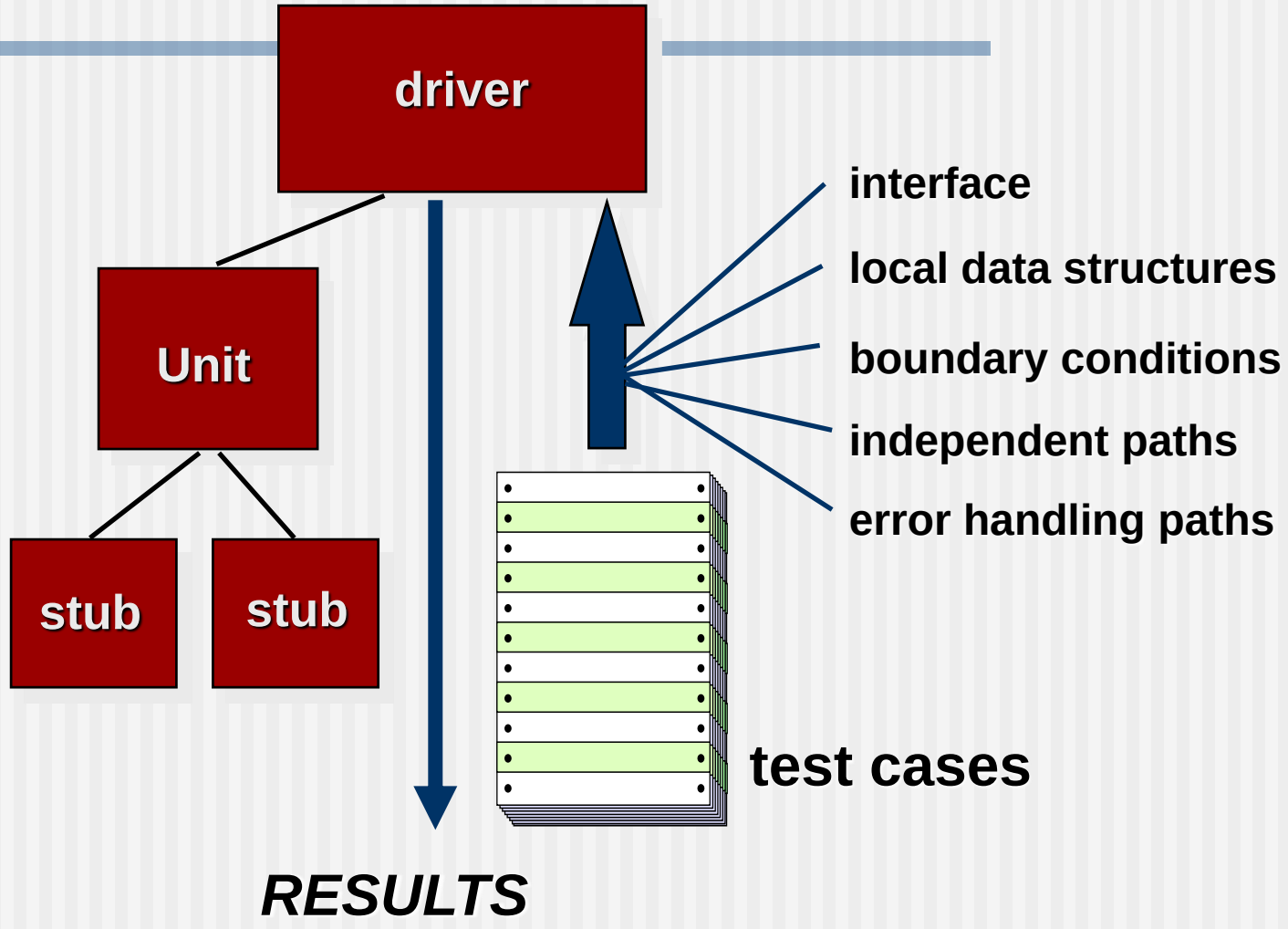
Unit Testing



Role of Scaffolding

- Because a unit is not a stand-alone program, some type of scaffolding is required to create a testing framework
  - **Driver** and/or **Stub** software must often be developed for each unit test
- **Driver**: In most applications a driver is nothing more than a "main program" that accepts test-case data, passes such data to the unit (to be tested), and prints relevant results
- **Stubs:** serve to replace units that are subordinate (invoked by) the unit to be tested
  - A stub or "dummy subprogram" uses the subordinate unit's interface, may do minimal data manipulation and returns control to the unit under test

Unit Test Environment



Scaffolding Overhead

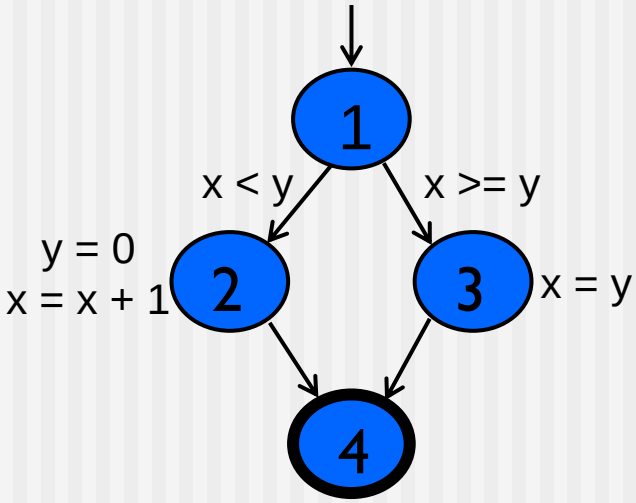
- Drivers and stubs represent testing “overhead”
  - Both are software that must be coded but that is not delivered with the final software product
- If drivers and stubs are kept simple, actual overhead is relatively low
- Unfortunately, many units cannot be adequately tested with “simple” scaffolding software
- In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also ready to use)

Control Flow Graph (CFG)

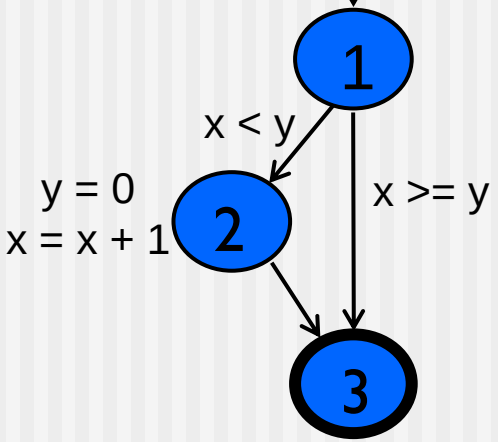
- A **CFG** models all executions of a method by describing control structures
- **Nodes**: Statements or sequences of statements (basic blocks)
- **Edges**: Transfers of control
- **Basic Block**: A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses

CFG: The if Statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```

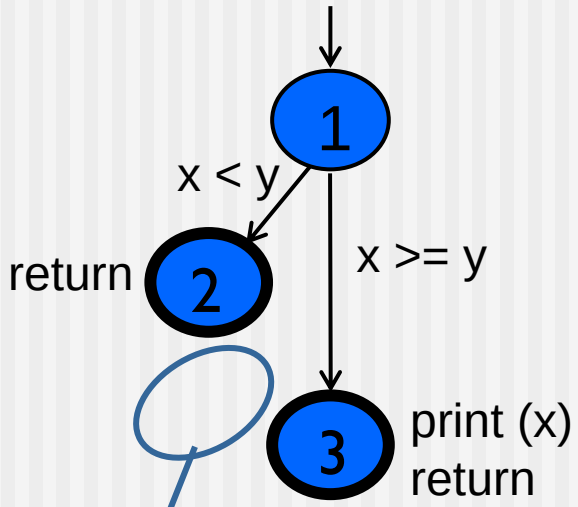



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG: The if-Return Statement

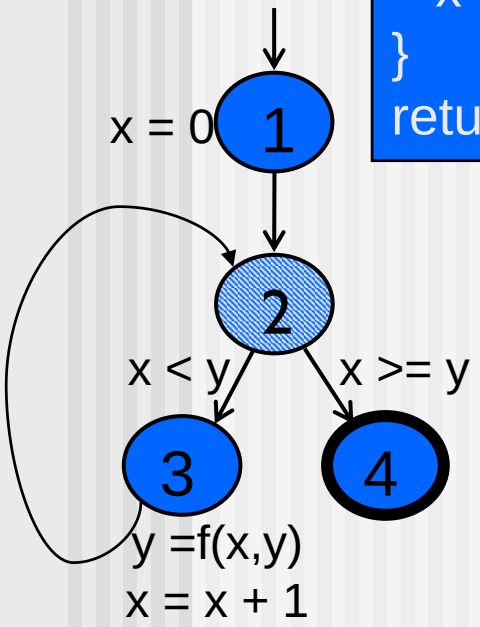
```
if (x < y)
{
   return;
}
print (x);
return;
```



No edge from node 2 to 3.
The return nodes must be distinct.

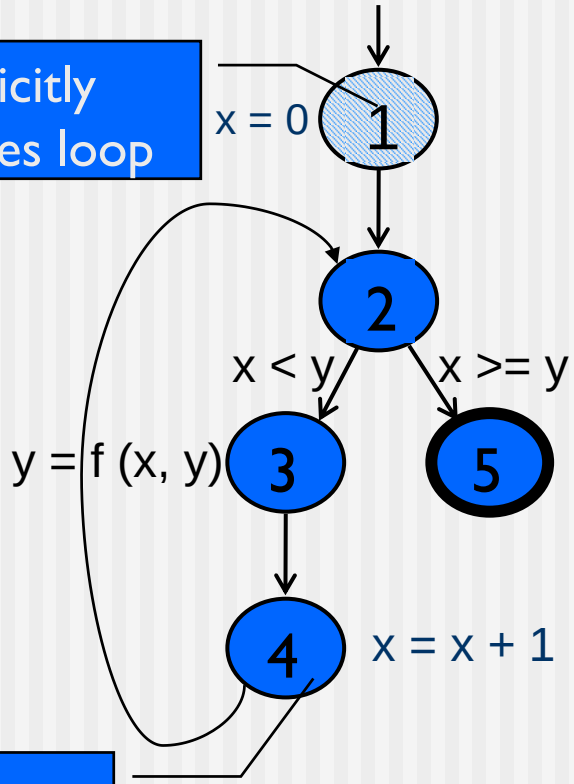
CFG: while and for

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
return (x);
```




```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
return (x);
```

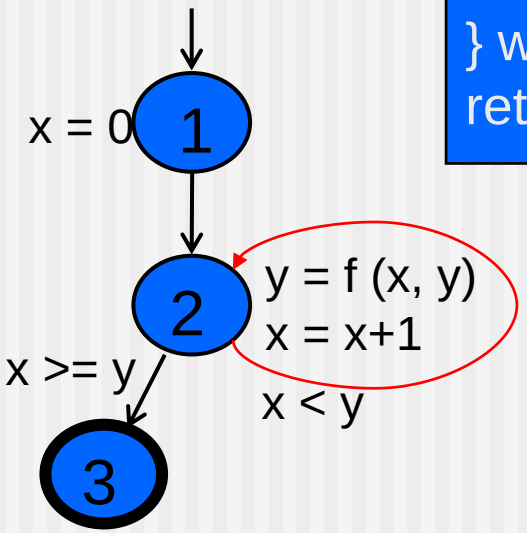
implicitly initializes loop



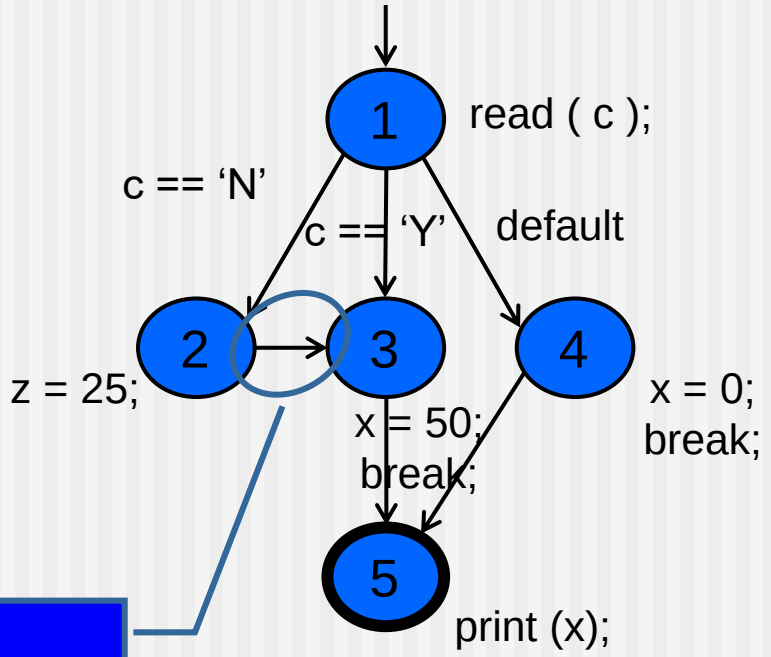
implicitly increments loop

CFG: do Loop

```
x = 0;
do
{
   y = f (x, y);
   x = x + 1;
} while (x < y);
return  (y);
```



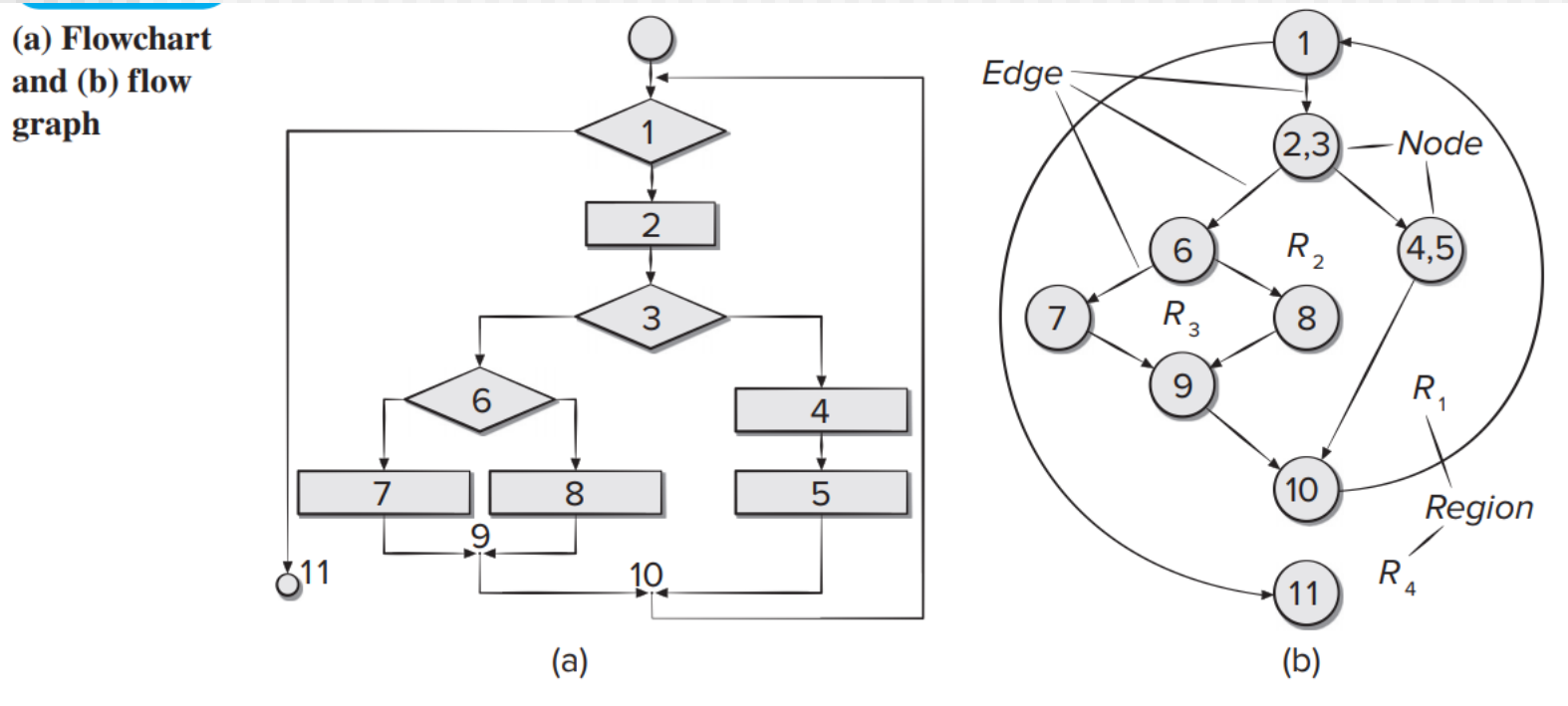
CFG: case (switch) Structure



Cases without breaks fall through to the next case

```
read ( c) ;
switch ( c )
{
   case 'N':
      z = 25;
   case 'Y':
      x = 50;
      break;
   default:
      x = 0;
      break;
}
print (x);
```

Mapping Flowchart to CFG



Areas bounded by edges and nodes are called ***regions***
When counting regions, we include the area outside the graph as a region

Tests on CFG

- **Structural Testing**

  (or Control Structure Testing): Defined on a graph just in terms of nodes and edges

  - Basis path testing
  - Loop testing

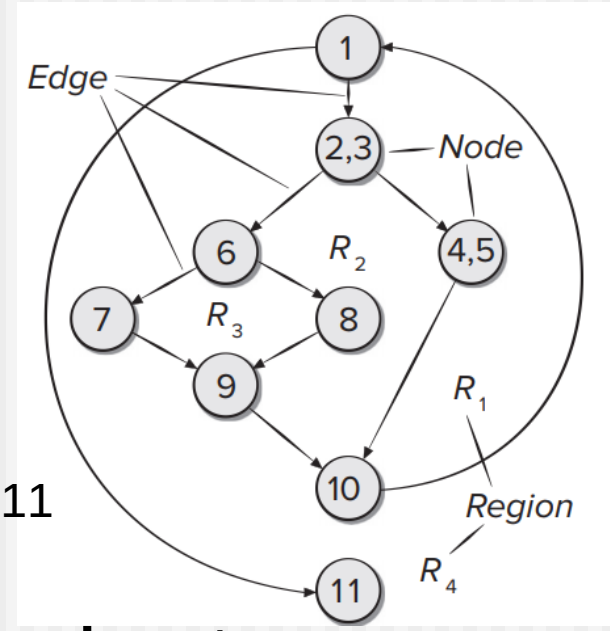
  Our focus in this course

  - …
- **Data Flow Testing:** Requires a graph to be annotated with references to variables
  - Selects test paths of a program according to the locations of definitions and uses of variables in the program

Independent Path

- Any path through the program that introduces **at least one new** set of statements or a new condition
- For example:
  - A set of independent 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
  - So <u>for the above set</u>, the path
    - 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

      is not an independent path
- Paths 1 through 4 constitute **a basis set**



Basis Path Testing

- If you can design tests to force execution of a basis set, then:
  - every statement in the program will have been guaranteed to be executed at least one time
  - and every condition will have been executed on its true and false sides (all branches are tested)
- We **derive test cases** to execute basis paths
- Note: The basis set is **not unique**
  - A number of different basis sets can be derived for a given CFG
- How do you know how many paths to look for?
- The computation of **cyclomatic complexity** provides the answer

Cyclomatic Complexity

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program
- In the context of basis path testing, the value of cyclomatic complexity indicates
  - **An upper bound for the number of tests** that must be conducted to ensure that all branches have been executed at least once
- Cyclomatic complexity has a foundation in **graph theory**

Computing Cyclomatic Complexity

- Cyclomatic complexity V(G) for a control flow graph G is computed in one of three ways:

1. The number of regions of G

2. $V(G) = E - N + 2$

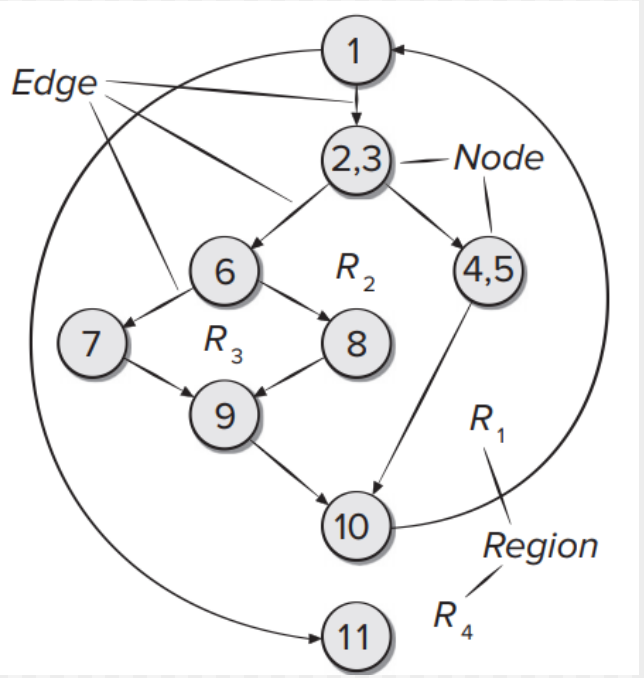
   where E is the number of CFG edges and N is the number of CFG nodes

3. $V(G) = P + 1$

   where P is the number of predicate nodes contained in the flow graph G

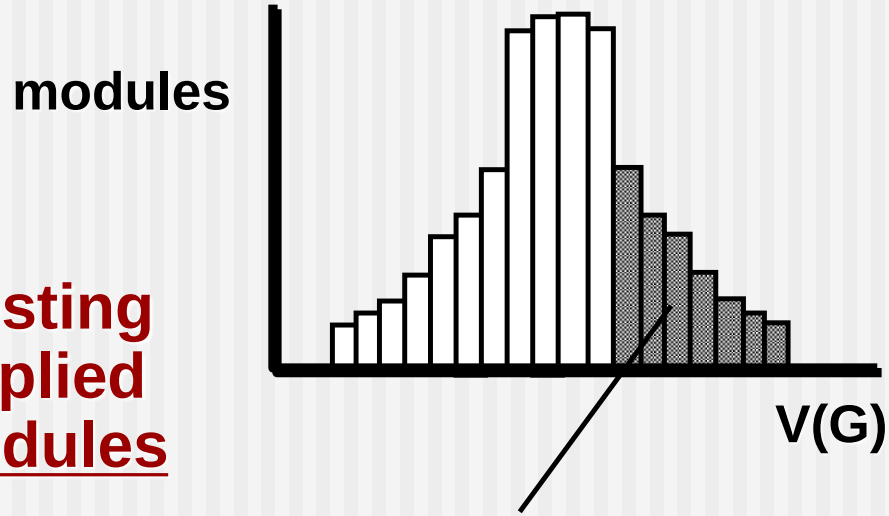
Cyclomatic Complexity: An Example

1. The flow graph has four regions
2. *V*(*G*) = 11 edges - 9 nodes + 2 = 4
3. *V*(*G*) = 3 predicate nodes + 1 = 4



Cyclomatic Complexity & Error

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

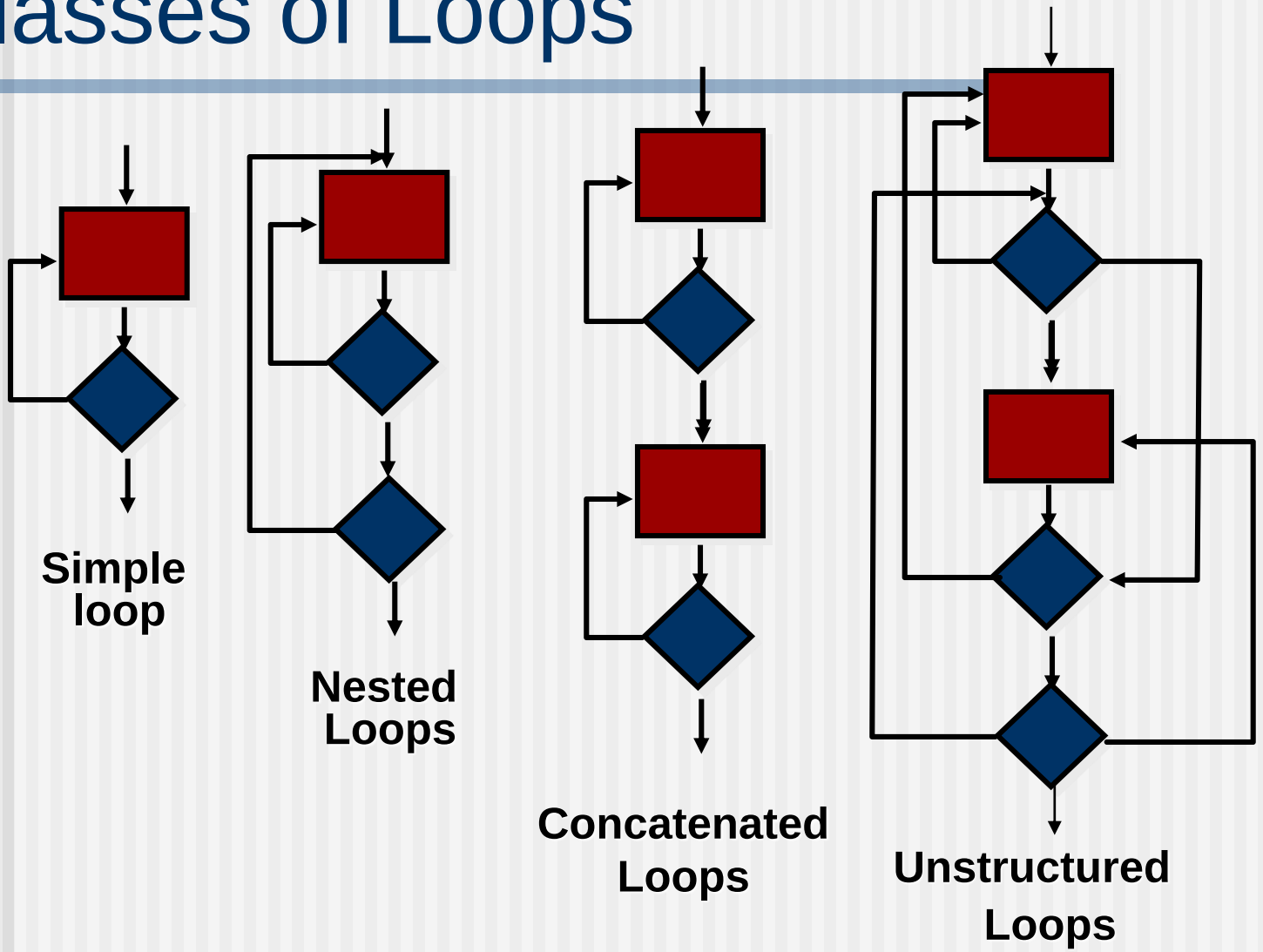


Basis path testing should be applied to critical modules

Loop Testing

- Focuses exclusively on the validity of loop constructs
- Four different classes of loops [Bei90]:
  1. simple loops
  2. concatenated loops
  3. nested loops
  4. unstructured loop

Classes of Loops



Loop Testing: Simple Loops

- The following set of tests can be applied to 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 through the loop.

Loop Testing: Nested Loops

- If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases
- This would result in an impractical number of tests
- Beizer [Bei90] suggests an approach that will help to reduce the number of tests:
  1. Start at the **innermost** loop. Set all **other loops to minimum** values.
  2. Conduct **simple loop tests for the innermost** loop while holding the outer loops at their minimum iteration values.
  3. **Work outward**, conducting tests for the next loop, but keeping all other **outer loops at minimum** values and other **nested loops to "typical"** values.
  4. Continue until all loops have been tested.

Loop Testing: Concatenated Loops

- If the loops are independent of one another
  - Treat each as a simple loop
- When the loops are not independent
  - The approach applied to nested loops is recommended
  - For example when the final loop counter of loop 1 is used as the initial value of loop 2

Unstructured Loops

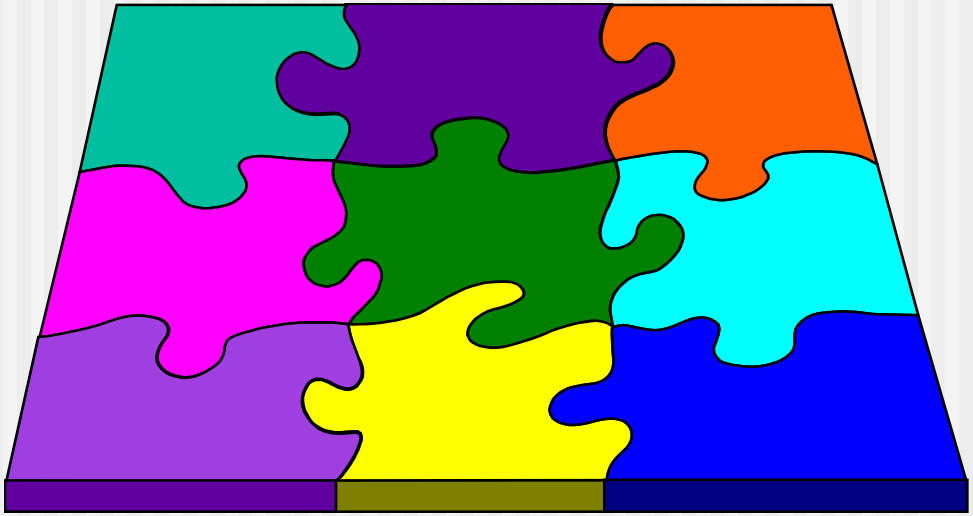
- Whenever possible, this class of loops should be redesigned to reflect the structured loops

Integration Testing

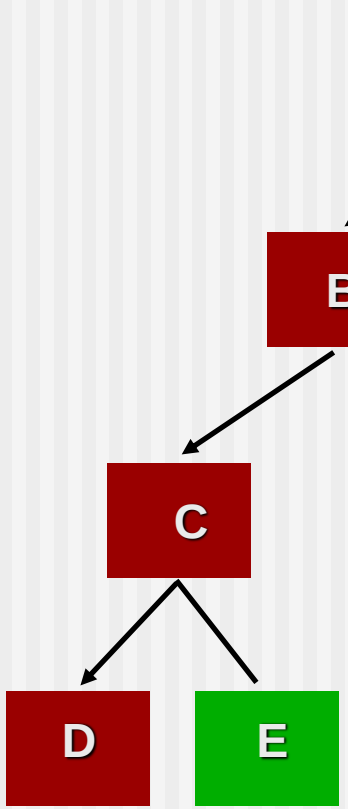
Integration Testing Strategies

Options:

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



Top Down Integration

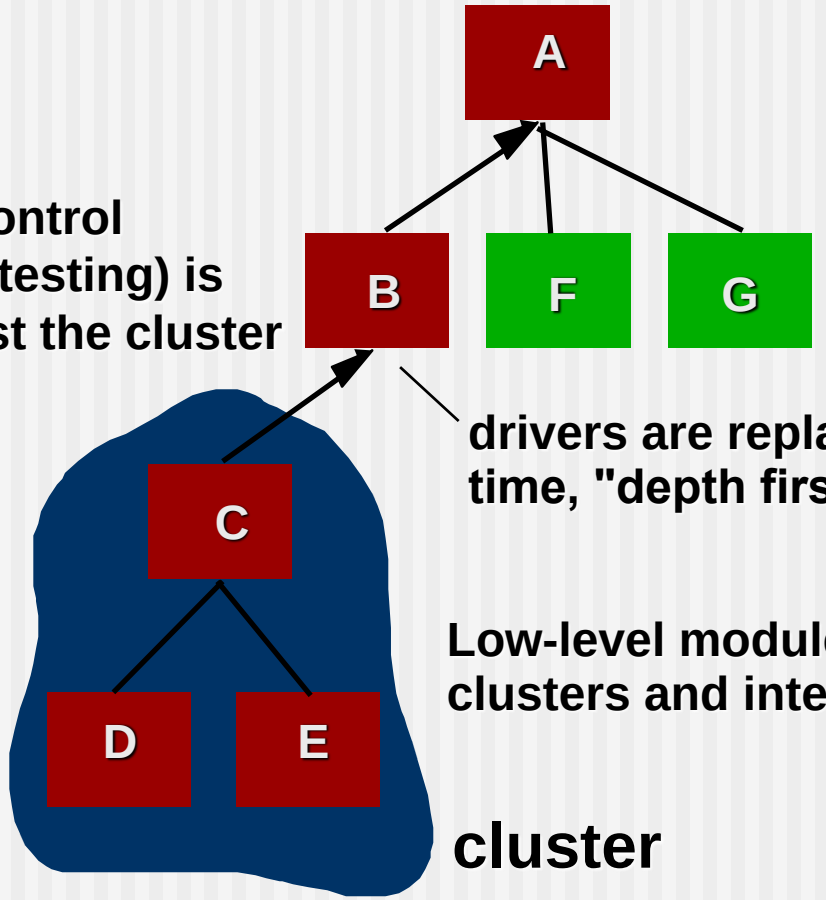


top module is tested with stubs

stubs are replaced one at a time with actual modules (in a "depth first" or "breadth first" manner)

as new modules are integrated, some subset of tests is re-run

Bottom-Up Integration



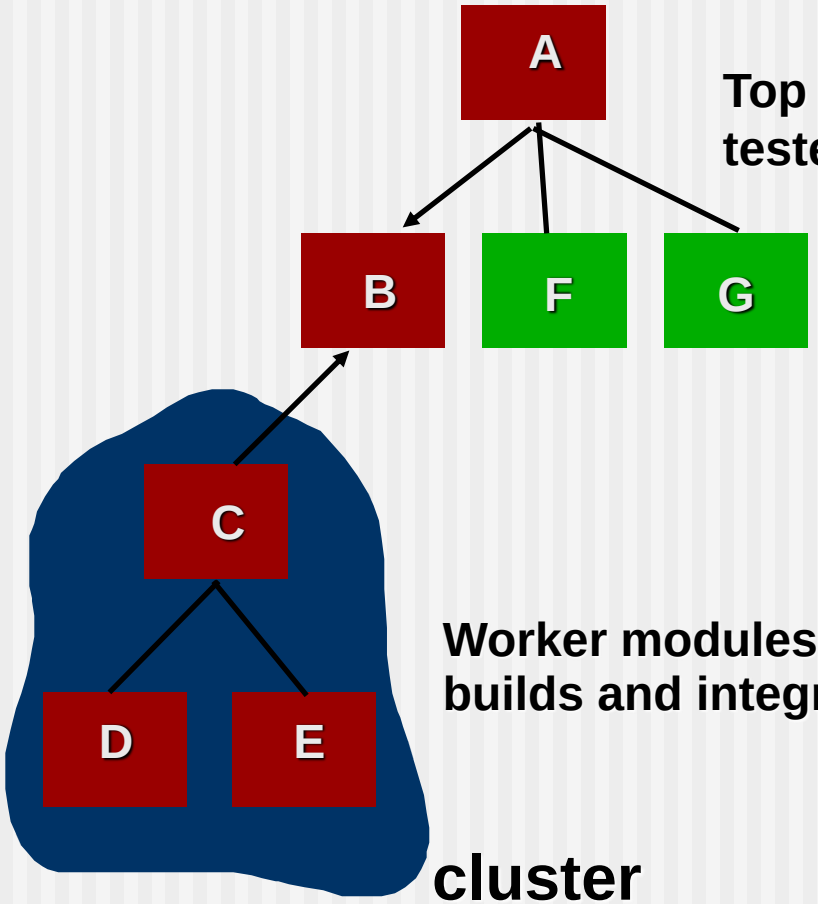
A driver (a control program for testing) is written to test the cluster

drivers are replaced one at a time, "depth first“ or “breadth first”

Low-level modules are grouped into clusters and integrated

cluster

Sandwich Testing



Top modules are tested with stubs

Worker modules are grouped into builds and integrated

cluster

Continuous Integration

- Merging components into the evolving software increment **once or more each day**
- A common **agile** development practice
- Quick and efficient integration testing to always have a working program as part of **continuous delivery**
- **Smoke testing** is an integration testing approach that can be used when software is developed by an agile team using short increment build times
  - A continuous integration strategy
  - The software is rebuilt (with new components added) and smoke tested every day
  - Assess the project on a frequent basis

Smoke Testing-I

- Smoke Test refers to an **initial testing** which is performed on **newly developed software build**
- Determines whether the deployed software **build is stable or not**.
- It acts a confirmation for the team to accept a build or reject and **proceed with further testing**.
- It consists of **a minimal set of tests** that run on each build to test software **core functionalities**.
  - Also called '**Surface Level** Testing '**—not deep** testing
  - **Low cost** testing
- It is used to ensure that all the critical functionalities are working properly or not.

Smoke Testing-II

- As it ensures the correctness of the software at the **initial stage**, it requires **less amount of effort and cost**.
- If we don’t perform smoke testing at an early stage, **defects may be encountered in later stages**
  - Costly defects
  - Defects found in the later stage can be **show stopper**
    - May **affect the release** of the deliverables
- Smoke testing minimizes the integration risks

Smoke Testing Steps

- Software components that have been translated into code are integrated into a “build.”
  - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function.
  - The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
  - The integration approach may be top down or bottom up.

Regression Testing

- In **depth** and **through** examination of software to ensure that recent **change** has **not adversely affected** the **existing features**
  - **High cost testing**
- The **re-execution** of some subset of tests that have already been conducted to ensure that changes have not propagated **unintended side effects**
- The verification of software **after any changes**
  - bug fixes, requirement changes, defect fix or any new module development
- In the case of newly developed builds
  - **Smoke** Test is always **followed by Regression** Test
  - Test Cases of Smoke Test is a part of Regression Testing and covers only the core functionalities.
- Regression testing can be done manually or **automated**.

System Testing

System Testing

- **Recovery testing**
- **Security testing**
- **Stress testing**
- **Performance testing**
- **Deployment testing**

Recovery Testing

- Many systems must recover from faults and resume processing with little or no downtime.
- A **fault tolerant system**
  - processing faults must not cause overall system function to cease
- Recovery testing is a system test that **forces the software to fail** in a variety of ways and **verifies that recovery is properly performed**.
- If recovery is automatic (performed by the system itself),
  - reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention,
  - the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits

Security Testing

- Security testing attempts to **verify** the **protection mechanisms** built into a system to protect it from improper **penetration**.
- 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.

Stress Testing

- Earlier software testing steps result in thorough evaluation of normal program functions and performance
- Stress tests are designed to confront programs with **abnormal situations**
- “How high can we crank this up before it fails?”
- Stress testing executes a system in a manner that demands resources in abnormal quantity or frequency
  - For example, special tests may be designed that generate 10 interrupts per second, when one or two is the average rate
  - test cases that require maximum memory
  - test cases that may cause thrashing in a virtual OS
  - …
- The tester attempts to **break the program**

Performance Testing

- For **real-time and embedded systems**, software that provides required function but does not conform to performance requirements is unacceptable
- Performance testing is designed to test the **run-time performance** of an integrated software system
- Performance testing occurs **throughout all steps** in the testing process
  - Even at the unit level, the performance of an individual module may be assessed
- However, the **true performance** of a system cannot be assessed until all system elements are **fully integrated**

Deployment Testing

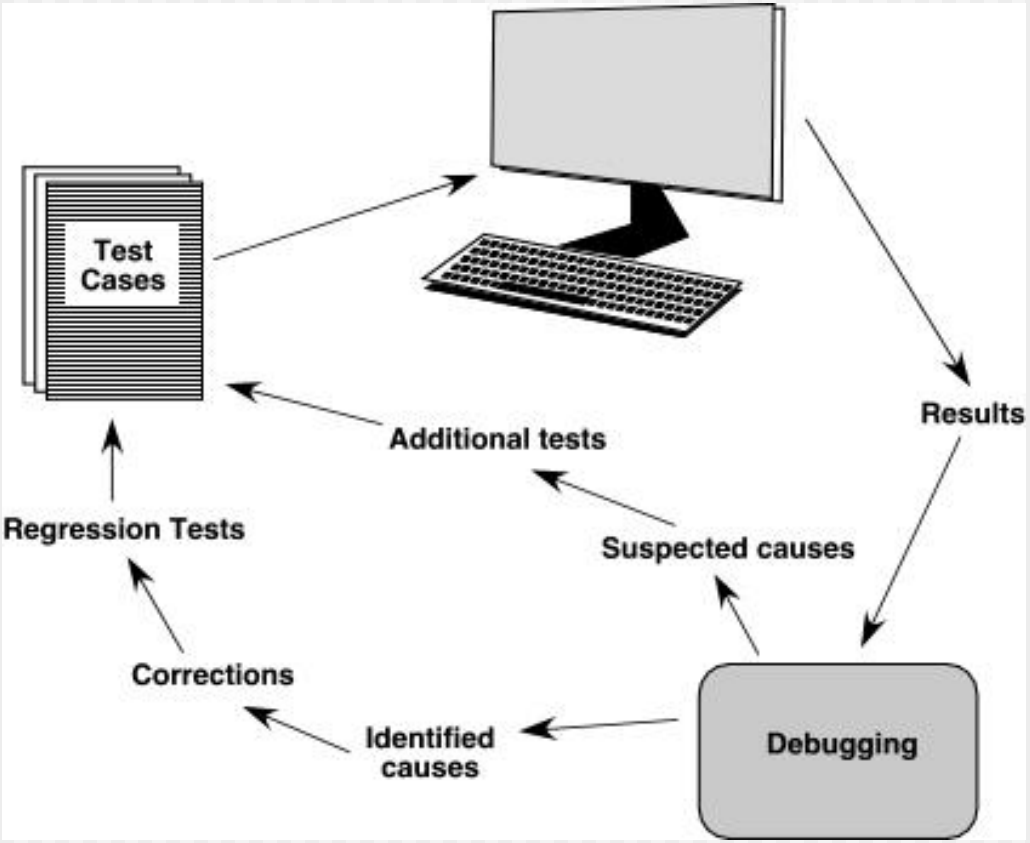
- In many cases, software must execute on a **variety of platforms** and under more than one operating system environment
- Also called **configuration testing**
- Deployment testing exercises the software in each operating environment
- In addition, deployment testing examines:
  - all installation procedures
  - all specialized installation software (e.g., “installers”) that will be used by customers, and
  - all documentation that will be used to introduce the software to end users

Debugging

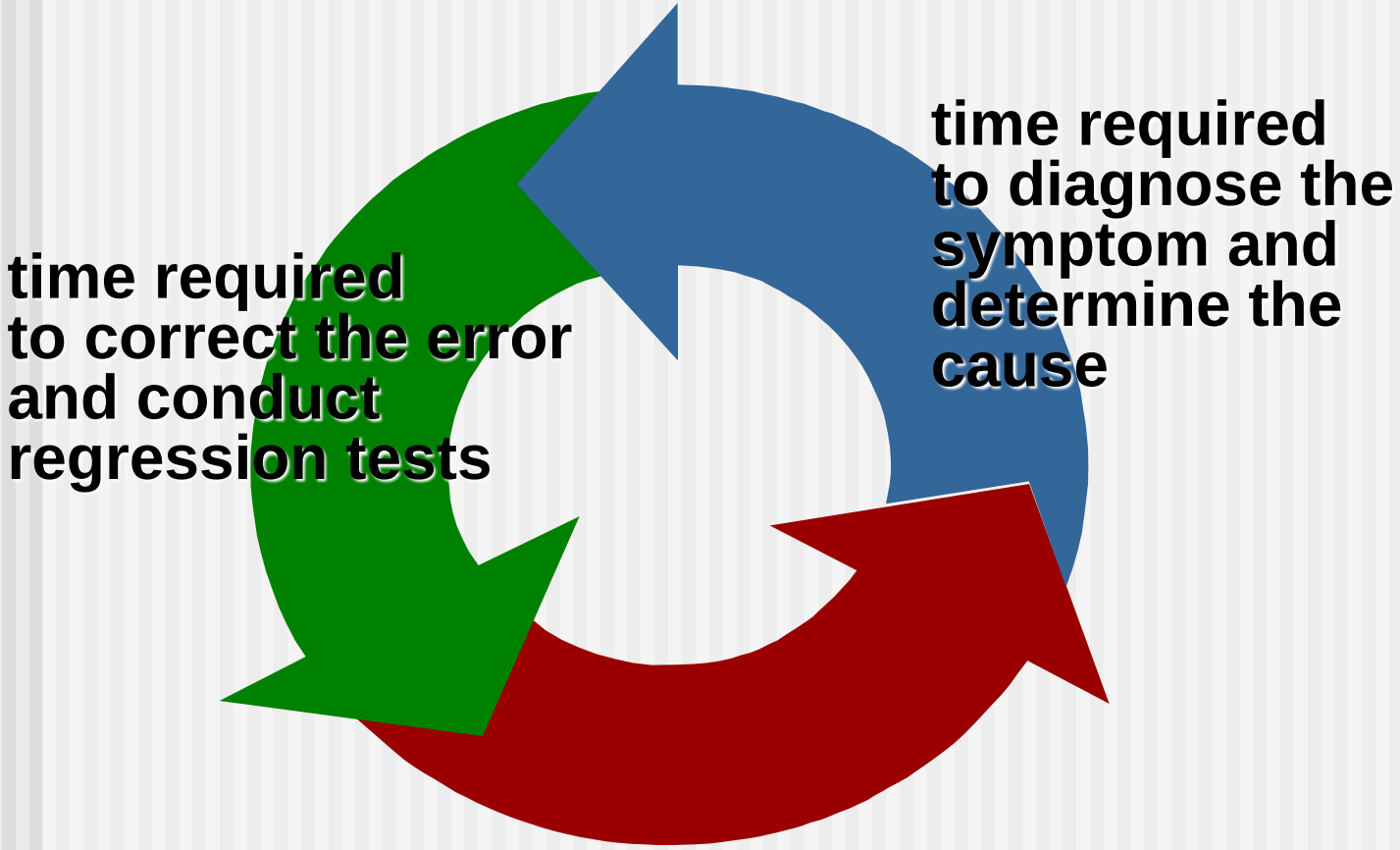
Debugging: A Diagnostic Process

- When a test case uncovers an error, debugging is the process that results in the **removal of the error**
- "**Symptomatic**" indication of a problem during test results evaluation
  - The **external manifestation** of the error and its **internal cause** may have **no obvious relationship** to one another
- The process that connects a symptom to a cause is debugging

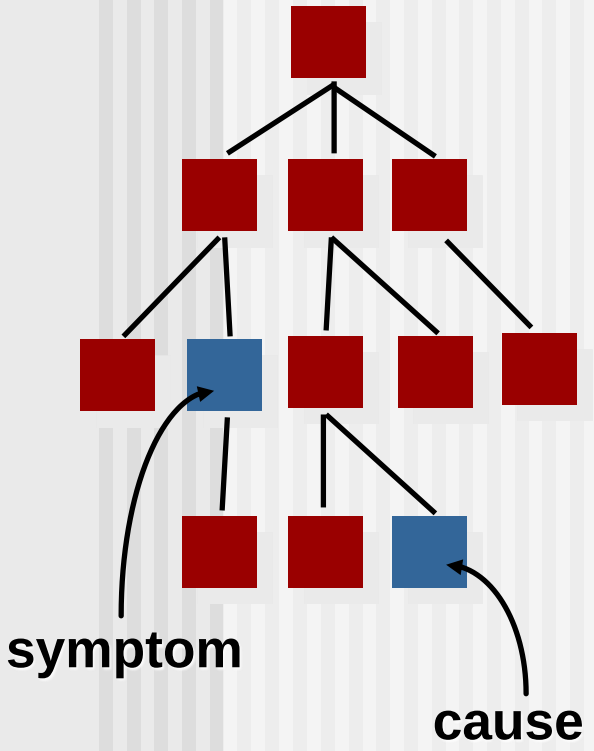
The Debugging Process



Debugging Effort



Symptoms & Causes



- symptom and cause may be geographically remote
- symptom may disappear (temporarily) when another error is fixed
- symptom may be caused by non-errors (e.g., round-off inaccuracies)
- symptom may be caused by human error (not easily traceable)
- may be difficult to accurately reproduce input conditions
- symptom may be due to causes distributed across multiple tasks running on different processors

Further Reading

The End