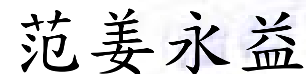
ESW聯盟「嵌入式系統與軟體工程」

Software Quality Assurance and Testing

課程：嵌入式系統與軟體工程

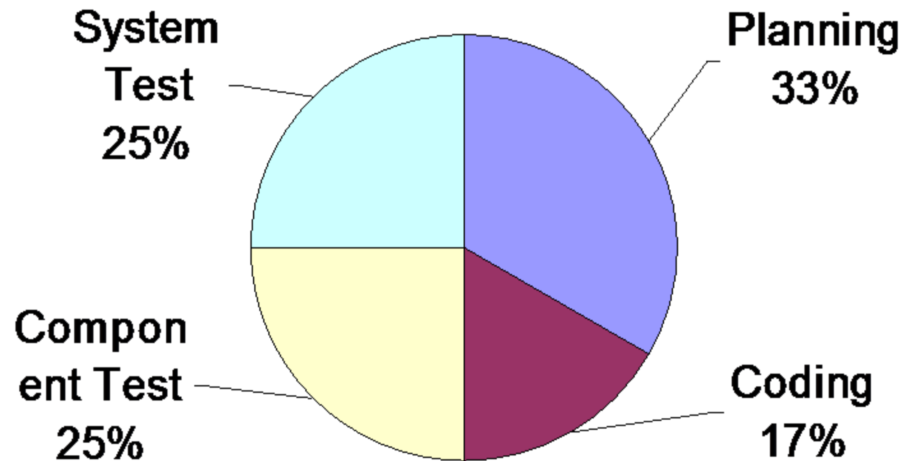
開發學校：輔仁大學資工系



范姜永益

Testing

- Brooks (MMM): Preferred time distribution – mostly planning and testing



- *The sooner you start coding, the longer it will take to finish the program*

What is Software Testing?

- The process of executing a program with the intent of finding errors. [Myers]
- The act of designing and executing tests. [Beizer]
- The process of executing a software system to determine whether it matches its specification and executes in its intended environment. [Whittaker]
- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of system or component. [IEEE]

What is Software Testing?

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to end users
- Principles
  - All tests should be traceable to customer requirements
  - Tests should be planned long ago before testing begins
  - Incremental testing
    - Testing should begin "**in the small**" and progress toward testing "**in the large**"
  - Exhaustive testing is not possible
  - To be more effective, testing should be done by an independent third party

What is Software Testing?

- Why do we test?
  - To check if there are any errors in a part or a product.
  - To gain the confidence in the correctness of the product.
  - To ensure the quality and satisfaction of the product

- Testing vs. debugging
  - Testing is to show that a program has bugs
  - Debugging is to locate and correct the error or misconception that cause the program failures
    - A failure is an unacceptable behaviour exhibited by a system

Who Tests the Software?

- Development engineers
  - Understand the system, but test "gently"
  - Driven by "delivery"
  - Only perform unit tests and integration tests
- Test engineers
  - Need to learn the system, but attempt to break it
  - Driven by "quality"
  - Define test cases, write test specifications, run tests, analyze results
- Customers
  - Driven by "requirements"
  - Determine if the system satisfies the acceptance criteria

Verification vs. Validation (V&V)

- **Verification**
  - "Are we building the product right"
  - The software should conform to its specification
- **Validation**
  - "Are we building the right product"
  - The software should do what the user really requires

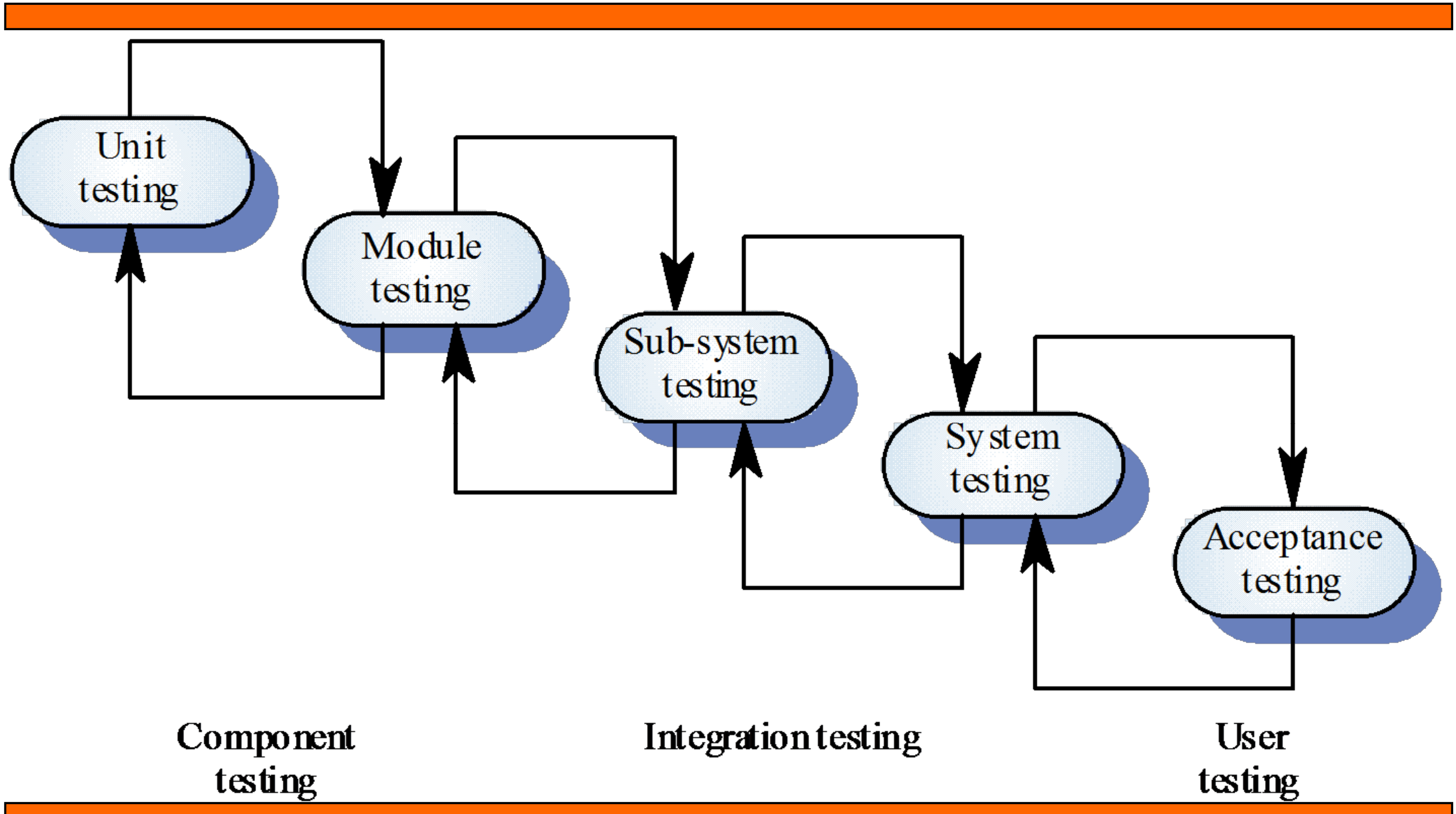
- Process
  - Is a **whole** life-cycle process - V & V must be applied at each stage in the software process
  - Has two principal objectives
    - The **discovery** of **defects** in a system
    - The **assessment** of whether or not the system is usable in an operational situation

Dynamic and Static V&V

- **Dynamic Testing**
  - Concerned with exercising and observing product behavior (testing)
  - Software testing

- **Static Testing**
  - Concerned with analysis of the static system representation to discover problems
  - Software inspection

Testing Process



Testing Stages

- **Unit testing**
  - testing of **individual** components
- **Module testing**
  - testing of collections of dependent components
- **Sub-system testing**
  - testing collections of modules integrated into sub-systems
- **System testing**
  - testing the complete system **prior to delivery**
- **Acceptance testing**
  - testing by users to check that the system satisfies requirements
  - sometimes called **alpha testing**

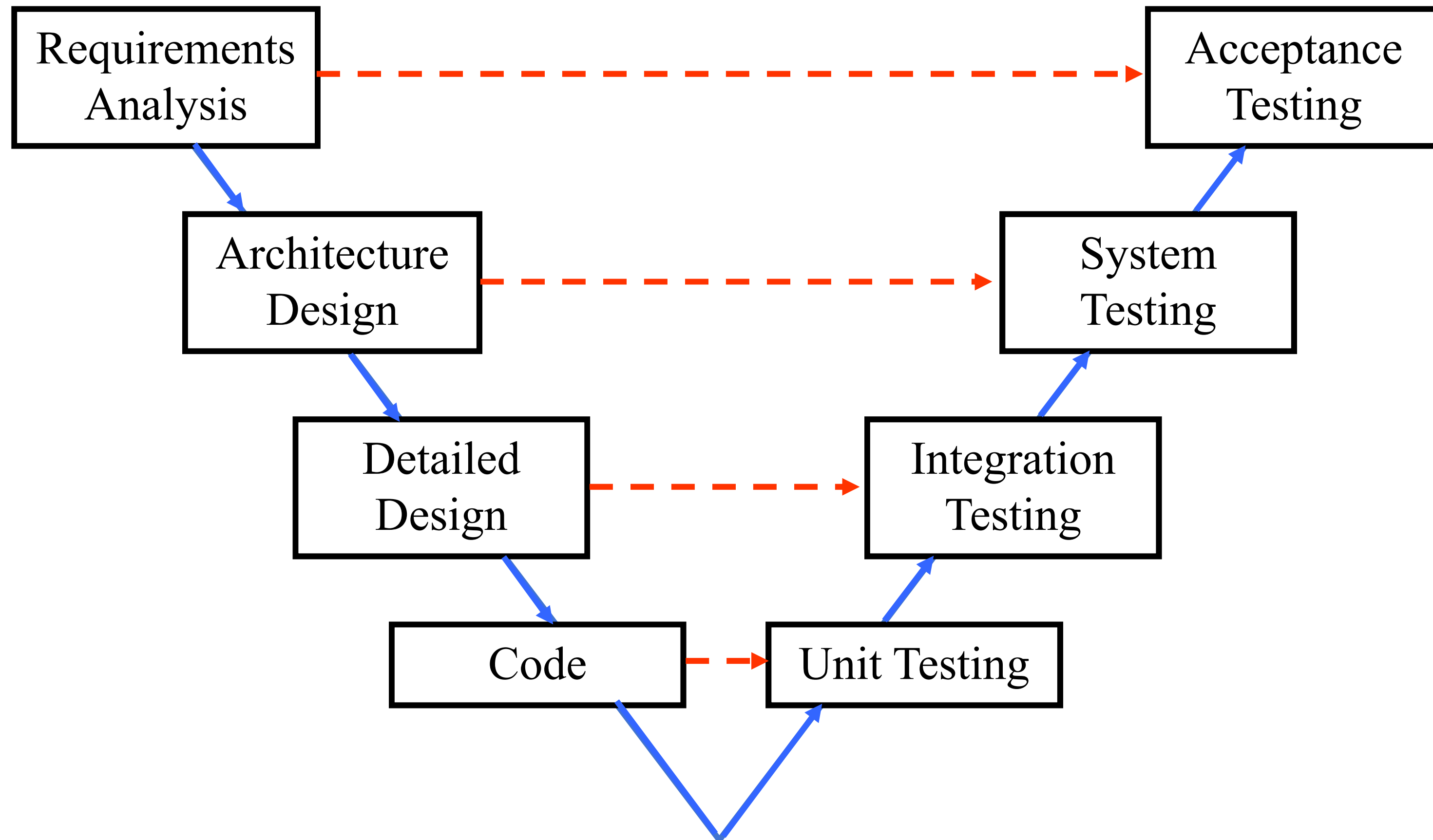
Test Planning and Scheduling

- Describe major phases of the testing process
- Describe **traceability** of tests to requirements
- Estimate overall schedule and resource allocation
- Describe relationship with other project plans
- Describe recording method for test results

The V Model

- The V model
  - Emerged in reaction to some waterfall models that showed testing as a single phase following the traditional development phases of requirements analysis, high-level design, detailed design and coding.
  - The V model portrays several distinct testing levels and illustrates how each level addresses a different stage of the software lifecycle.
  - The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side.

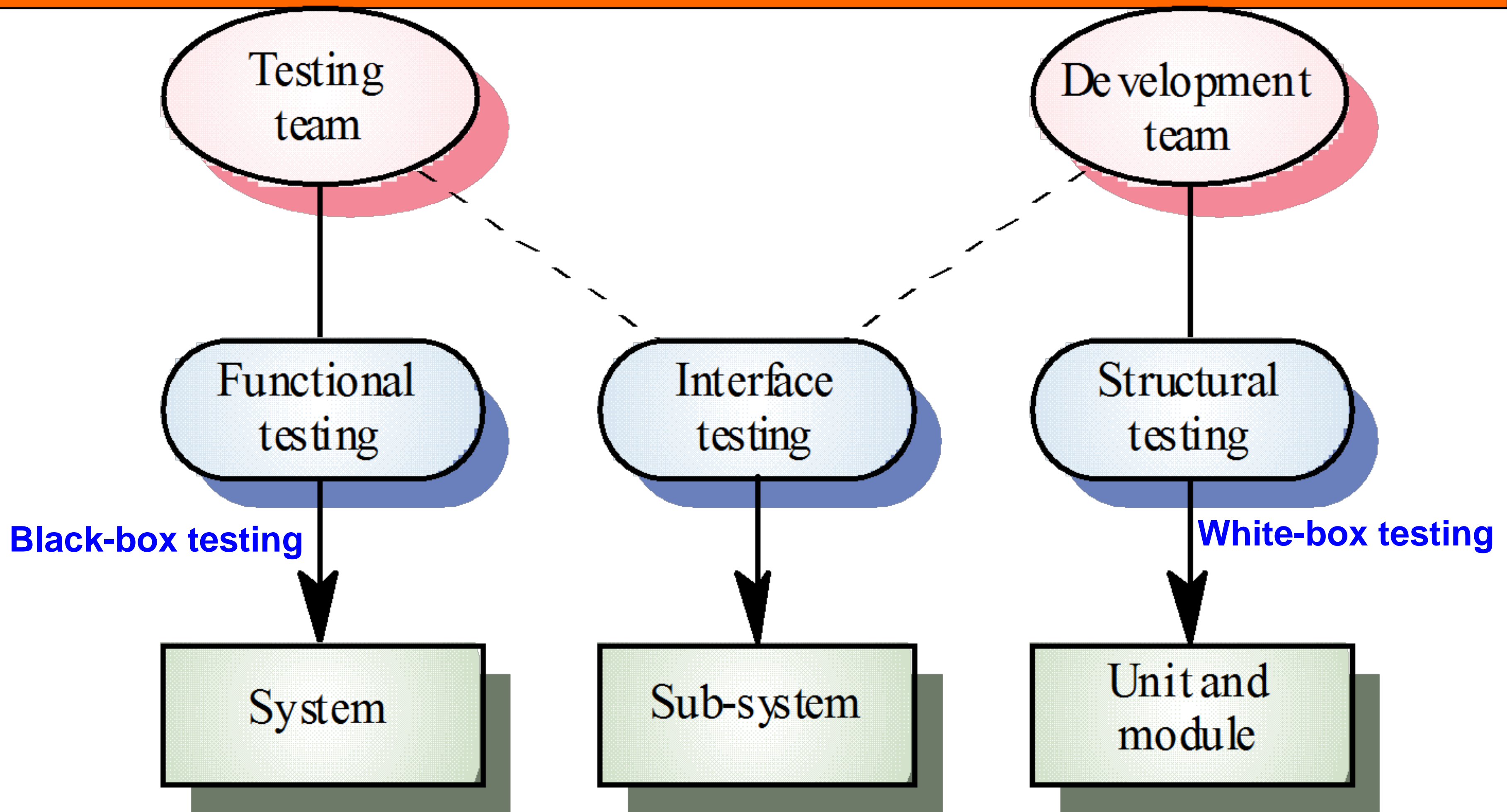
The V Model



Testing Methods

- Two general software testing methods:
  - White-box testing:
    - Design tests to exercise internal structures of the software to make sure they operates according to specifications and designs
  - Black-box testing:
    - Design tests to exercise each function of the software and check its errors.
  - White-box and black-box testing approaches can uncover different class of errors and are complement each other

Defect Testing Approaches



White-Box Testing

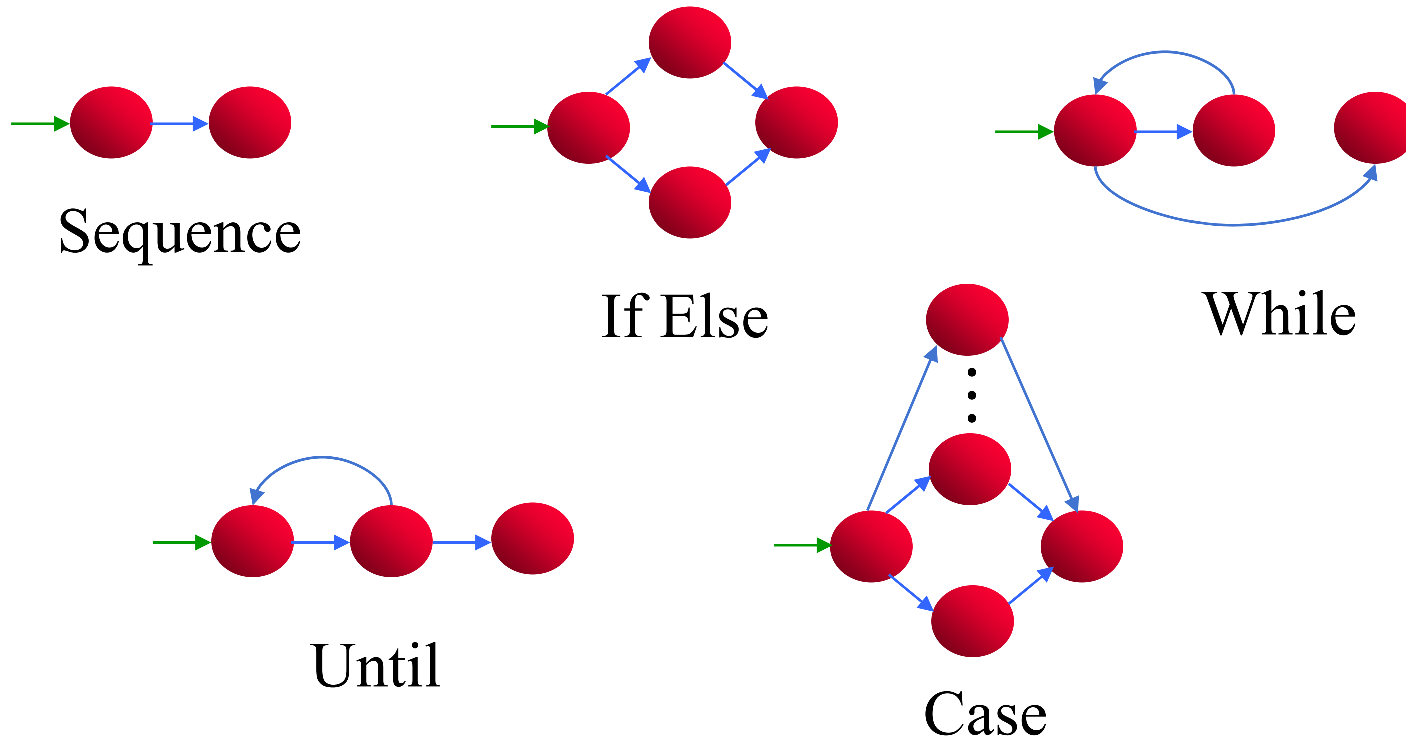
- Also known as glass-box testing or structural testing
- Objective is to exercise all program statements
- Knowledge of the program is used to identify additional test cases
- A test case design method that uses the control structure of the procedural design to derive test cases
- Focus on the control structures, logical paths, logical conditions, data flows, internal data structures, and loops.
- W. Hetzel describes white-box testing as “testing in the small”

Basis Path Testing

- Basic path testing (a white-box testing technique)
  - First proposed by Tom McCabe
  - Can be used to derive a logical complexity measure for a procedure design
  - Used as a guide for defining a basis set of execution path
  - Guarantee to execute every statement in the program at least one time

Basis Path Testing

- The basic structured-constructs in a flow graph

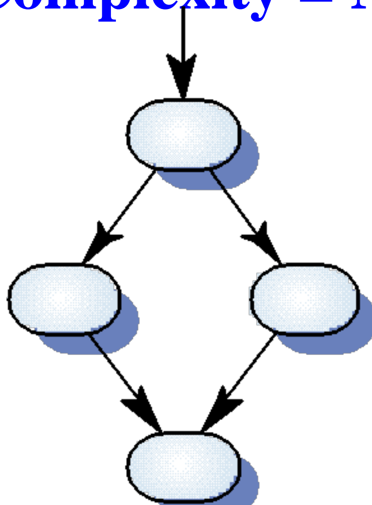


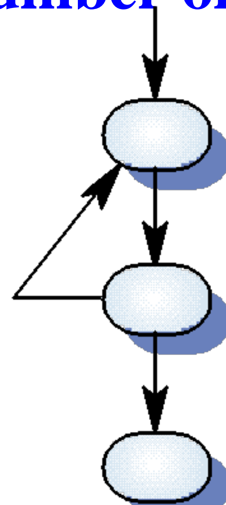
Path Coverage

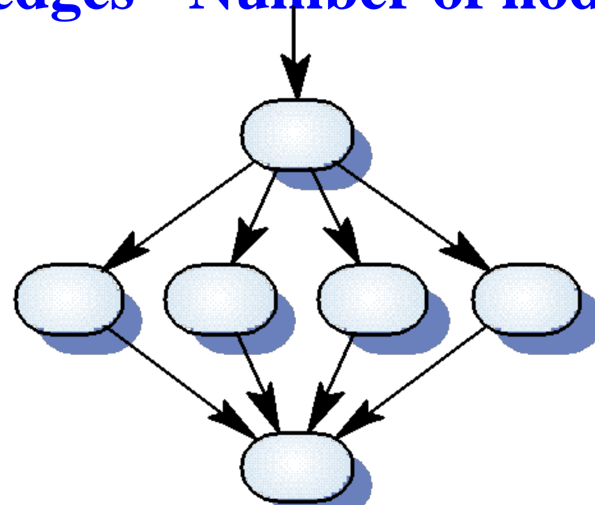
- **Path testing**
  - To exercise every independent execution path through the component
  - If every independent path is executed then all statements in the program must have been executed at least once
    - All conditional statements are tested for both true and false cases
- **Flow graph**
  - Describes the program control flow
  - Is constructed by replacing program control statements by equivalent diagrams
  - If there are no ‘goto’ statements in a program, it is a straightforward mapping from a program to a flow graph
  - Used as a basis for computing the cyclomatic complexity

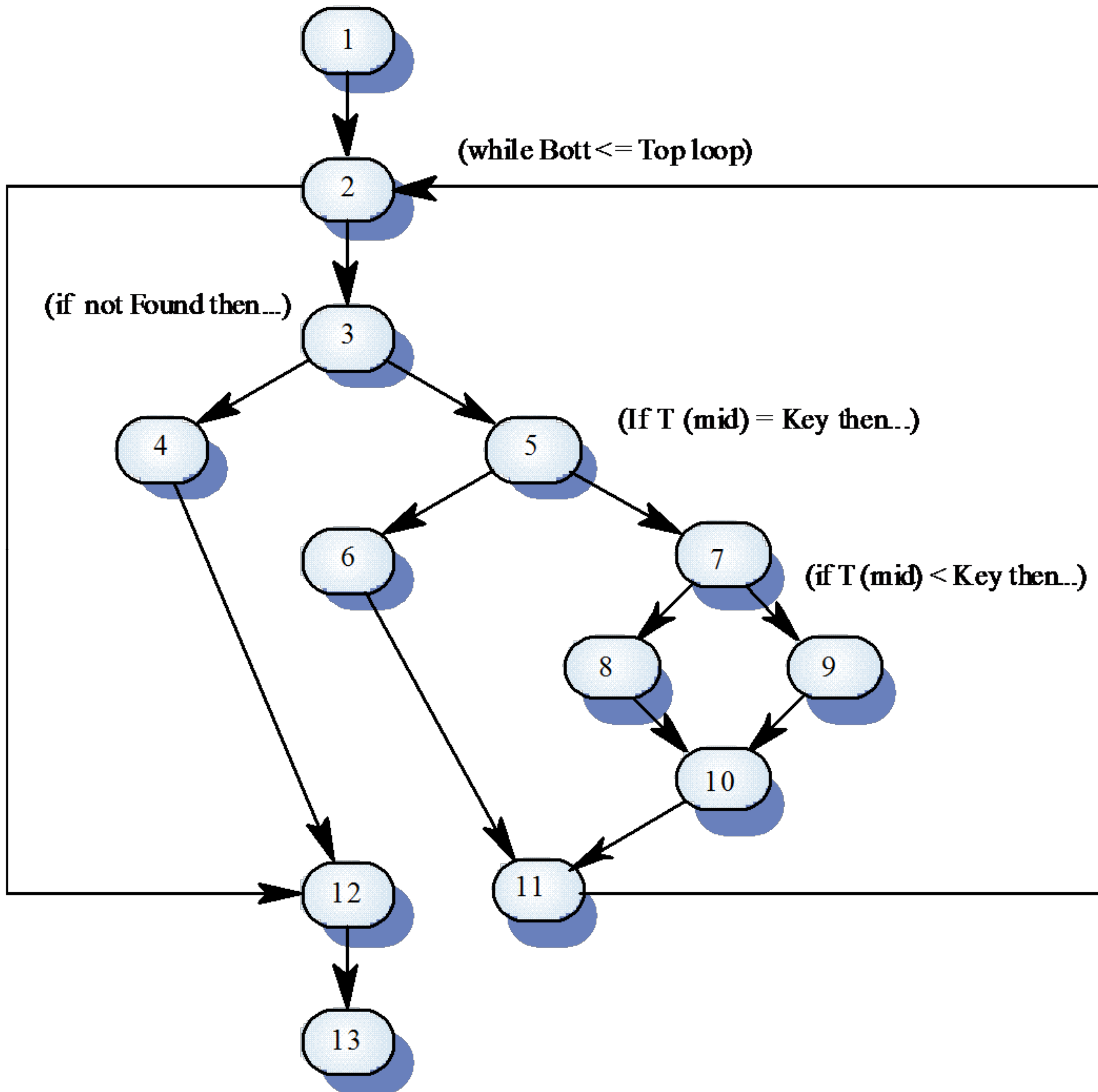
Program flow graphs

- The number of independent path in a program can be discovered by computing the cyclomatic complexity (McCabe 1976)
  - **Complexity = Number of edges - Number of nodes +1**








1
2
(while Bott <= Top loop)
(if not Found then...)
3
4
5
(If T (mid) = Key then...)
6
7
(if T (mid) < Key then...)
8
9
10
11
12
13

Cyclomatic Complexity

- The number of tests to test all control statements equals the cyclomatic complexity
- Cyclomatic complexity equals number of conditions in a program
- Independent paths
  - 1, 2, 12, 13
  - 1, 2, 3, 4, 12, 13
  - 1, 2, 3, 5, 6, 11, 2, 12, 13
  - 1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13
  - 1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13
  - A dynamic program analyzer may be used to check that paths have been executed

Black-Box Testing

- Also known as functional testing, behavioral testing, or specification-based testing
- Does not have the knowledge of the program's structures
- Discover program errors based on program requirements and product specifications
- Derive sets of inputs to fully exercise all functional requirements for a program
- Complementary to white-box testing

Black-Box Testing

- Focuses on the functional requirements of the software including functions, operations, external interfaces, external data and information
- Attempts to find errors in the following:
  - Incorrect of missing functions
  - Interface errors
  - Errors in data structures or external database access
  - Performance errors
  - Initialization and termination errors

Equivalence Partitioning

- Equivalence partitioning
  - The input domain of a program is partitioned into a finite number of equivalence classes from which test cases are derived
  - An equivalence class consists of a set of data that is treated the same by the program or that should generate the same result
  - Test case design is based on an evaluation of equivalence classes for an input condition
  - Can reduce the total number of test cases to be developed

Equivalence Partitioning

- The equivalence classes are identified based on the set of valid or invalid states for each input condition
- An input condition can be
  - A specific numeric value
  - A range of values
  - A set of related values
  - A Boolean condition

Equivalence Partitioning

- The following guidelines can help to define the equivalence classes [Pressman 2004]:
  - If an input condition specifies a range, one valid and two invalid equivalence class are defined
  - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined
  - If an input condition is Boolean, one valid and one invalid classes are defined

Equivalence Partitioning Example

- Consider we are writing a program for computing letter grades based on the numerical scores of students, where the input variable is *Score*. The rule of computing the grade is as follows:

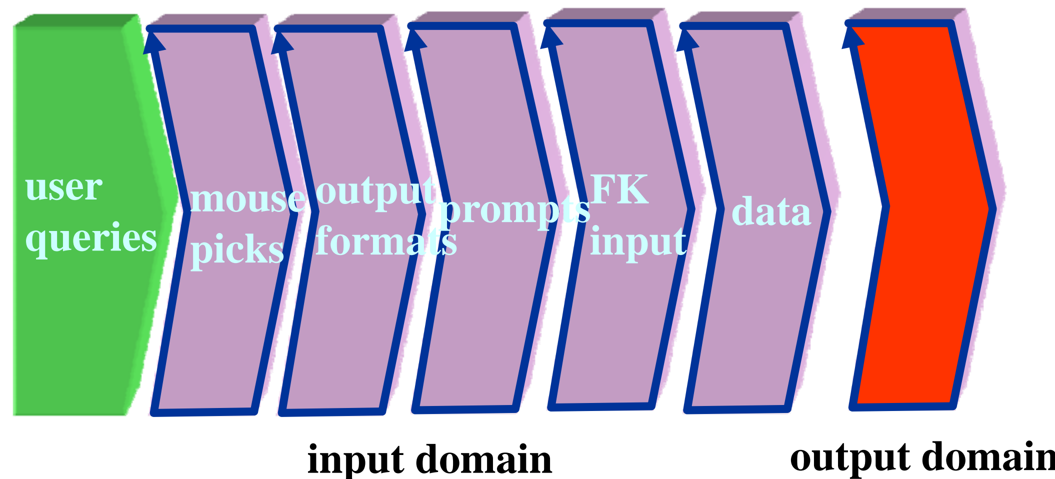
Score	Grade
90~100	A
80~89	B
70~79	C
60~69	D
0~59	F

Equivalence Partitioning Example

- The input domain of *Score* can be partitioned into 5 valid equivalence classes and 2 invalid equivalence classes
  - Valid classes: 0~59, 60~69, 70~79, 80~89, 90~100
  - Invalid classes: smaller than 0 and greater than 100
- Any data value within a class is considered equivalence in terms of testing
- Using the equivalence partitioning testing, we can reduce the test cases from 100 (assume 0 <= score <= 100) to 7

Boundary Value Testing

- Based on programming experience, more errors are found at the boundaries of an input/output domain than in the “center”.
- In addition to select test data “inside” an equivalence class, data at the “edges” of the equivalence class also need to be examined



Note: Redraw from [Pressman 2004]

Boundary Value Testing

- Boundary value analysis (BVA)
  - A test case design technique complements to equivalence partition
  - Rather selecting any element from an equivalence class, BVA leads to the selection of test cases that exercise bounding values (“edge” of the class)
  - Unlike equivalence partition that derives test cases only from input conditions, BVA derives test cases from both input conditions and output domain

Boundary Value Testing

- Guidelines [Pressman 2004]:
  1. If an input condition specifies a range [*a*, *b*], test cases should be designed with value *a* and *b*, just above and below *a* and *b*
     - Example: Integer D with input condition [-3, 5], BVA test values are -3, 5, -2, 6, -1, 4
  2. If an input condition specifies a number values, test cases should be developed to exercise the minimum, number, maximum number, and values just above and below minimum and maximum
     - Example: Enumerate data E with input condition: {3, 5, 100, 102}, BVA test values are 3, 102, 2, 4, 101, 103

Boundary Value Testing

3. Guidelines 1 and 2 are applied to output condition
4. If internal program data structures have prescribed boundaries, make sure to design a test case to exercise the data structure at its boundary
   - Array input condition:
     - Empty, single element, full element, out-of-boundary
   - Search output condition:
     - Element is inside array or the element is not inside array

Boundary Value Testing Example

- Consider the letter grade assignment program in previous example
- The input domain of *Score* can be partitioned into 5 valid equivalence classes and 2 invalid equivalence classes
  - Valid classes: 0~59, 60~69, 70~79, 80~89, 90~100
  - Invalid classes: smaller than 0 and greater than 100
- With BVA, we can obtain the following test values

Classes	Just below minimum	Minimum	Just above minimum	Just below maximum	Maximum	Just above maximum
>100	100	101	102	-	-	-
90~100	89	90	91	99	100	101
80~89	79	80	81	88	89	90
70~79	69	70	71	78	79	80
60~69	59	60	61	68	69	70
0~59	-1	0	1	58	59	60
<0	-	-		-2	-1	0

Error Guessing

- Identify potential errors and design test cases based on intuition and experiences
- Test cases can be derived by making a list of possible errors or error-prone situations
  - Empty or null lists
  - Zero instances or occurrences
  - Blanks or null strings
  - Negative numbers
  - Historical defects (need to maintain defect history)

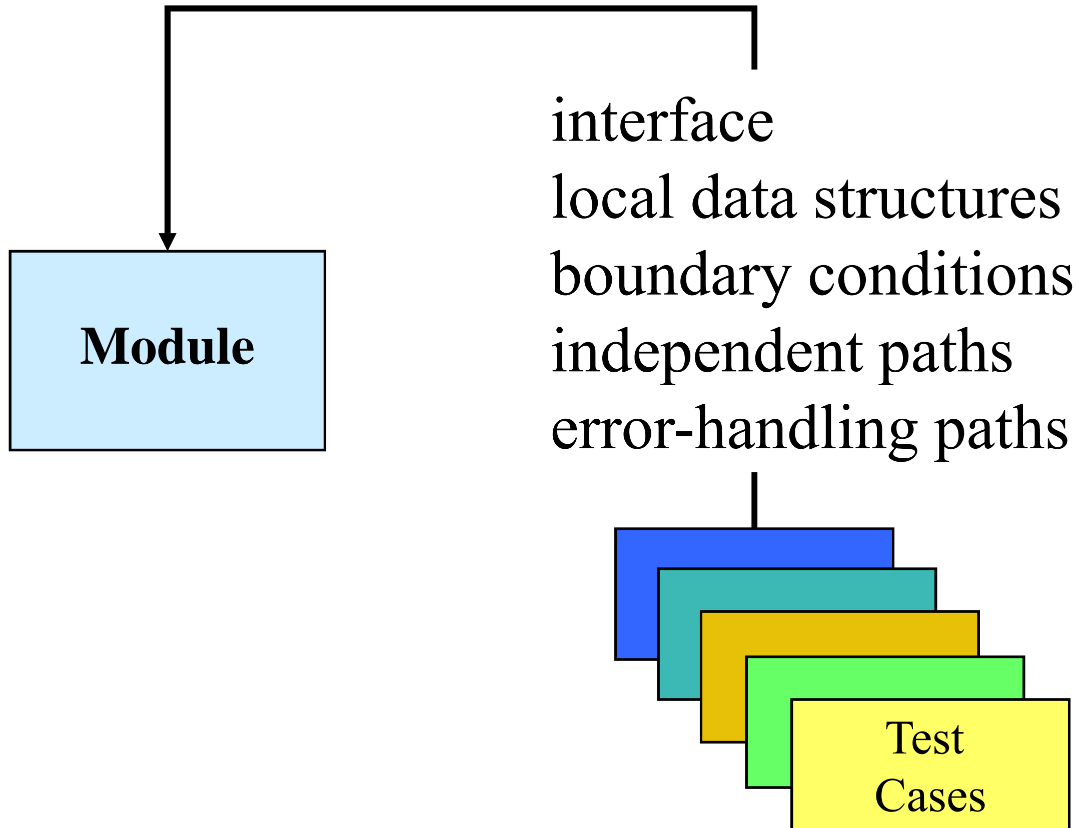
Testing Strategy

- Testing strategies are ways of approaching the testing process
- Different strategies may be applied at different stages of the testing process
- Strategies covered
  - **Top-down testing**: starting with the most abstract
  - **Bottom-up testing**: starting with the fundamental component
  - **Thread testing**
  - **Stress testing**: how well the system can cope with overload situations
- Whenever testing strategy is adopted, it is always sensible to adopt an incremental approach to subsystem and system testing

Unit Testing

- Is normally considered as an adjunct to the coding step
- Focuses verification effort on the smallest unit of software design – the software component or module
- Using the component-level design description as a guide
- Provide a release criterion for a programming task
- Encourage early defect discovery
- Discourage big-bang integration with its late defect discovery
- Is white-box oriented and the step can be conducted in parallel for multiple components

Unit Testing



Note: Redraw from [Pressman 2004]

Unit Test Cases

- Unit test cases should be designed to uncover errors due to erroneous computation, incorrect comparisons, or improper control flow
- Common errors in computation
  - Misunderstood or incorrect arithmetic precedence
  - Mixed mode operations
  - Incorrect initialization
  - Precision inaccuracy
  - Incorrect symbolic representation of an expression

Unit Test Cases

- Common errors in comparison and control flow
  - Comparison of different data types
  - Incorrect logical operators or precedence
  - Expectation of equality when precision error makes equality unlikely
  - Incorrect comparison of variables
  - Improper or nonexistent loop termination
  - Failure to exit when divergent iteration is encountered
  - Improperly modified loop variables

Unit Test Cases

- What should be tested when error handling is evaluated?
  - 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
- Boundary testing
  - An important task of the unit test step
  - 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

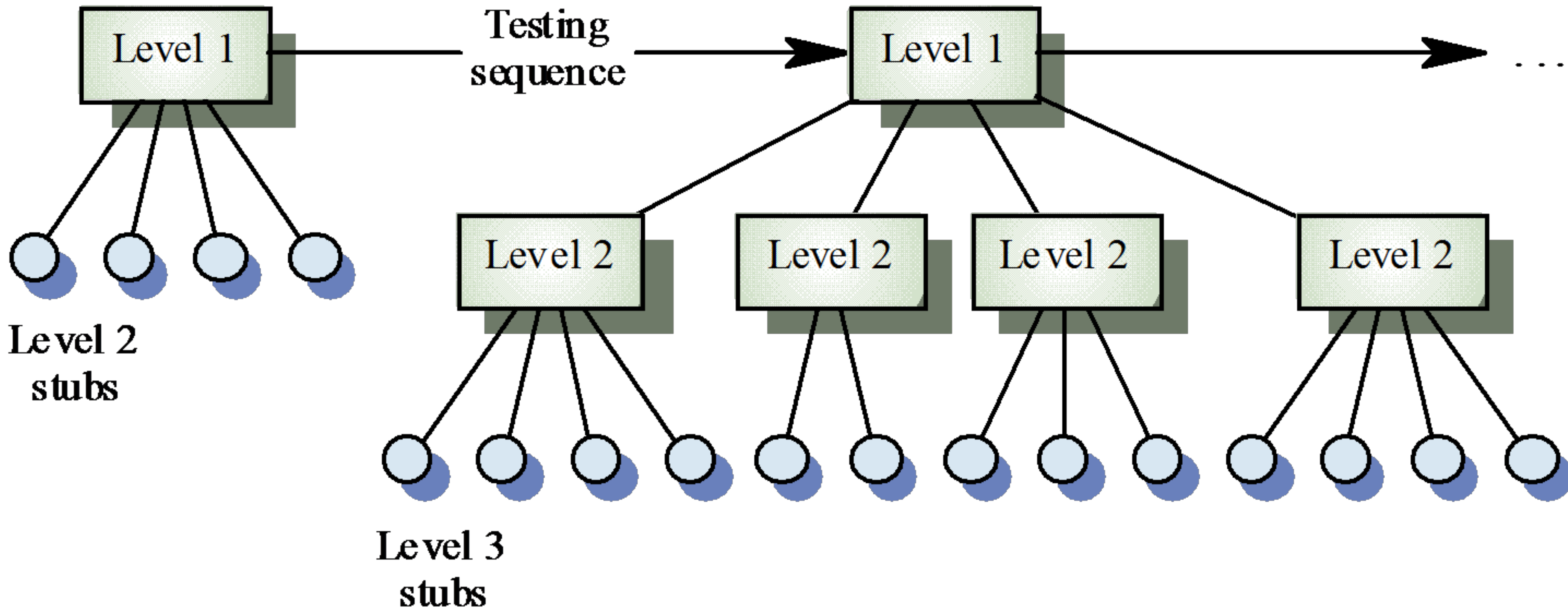
Integration Testing

- Big bang (non-incremental integration)
  - All components are combined in advance. The entire program is tested as a whole
  - When a set of errors is encountered, correction is difficult because isolation of causes is complicated by the vast expanse of entire program
  - Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop

Integration Testing

- Incremental integration
  - The program is constructed and tested in small increments, where errors are easier to isolate and correct
  - Interfaces are more likely to be tested completely
  - A systematic test approach may be applied
    - Top-down integration
    - Bottom-up integration
    - Sandwich testing (combination of above two approaches)

Top-Down Testing



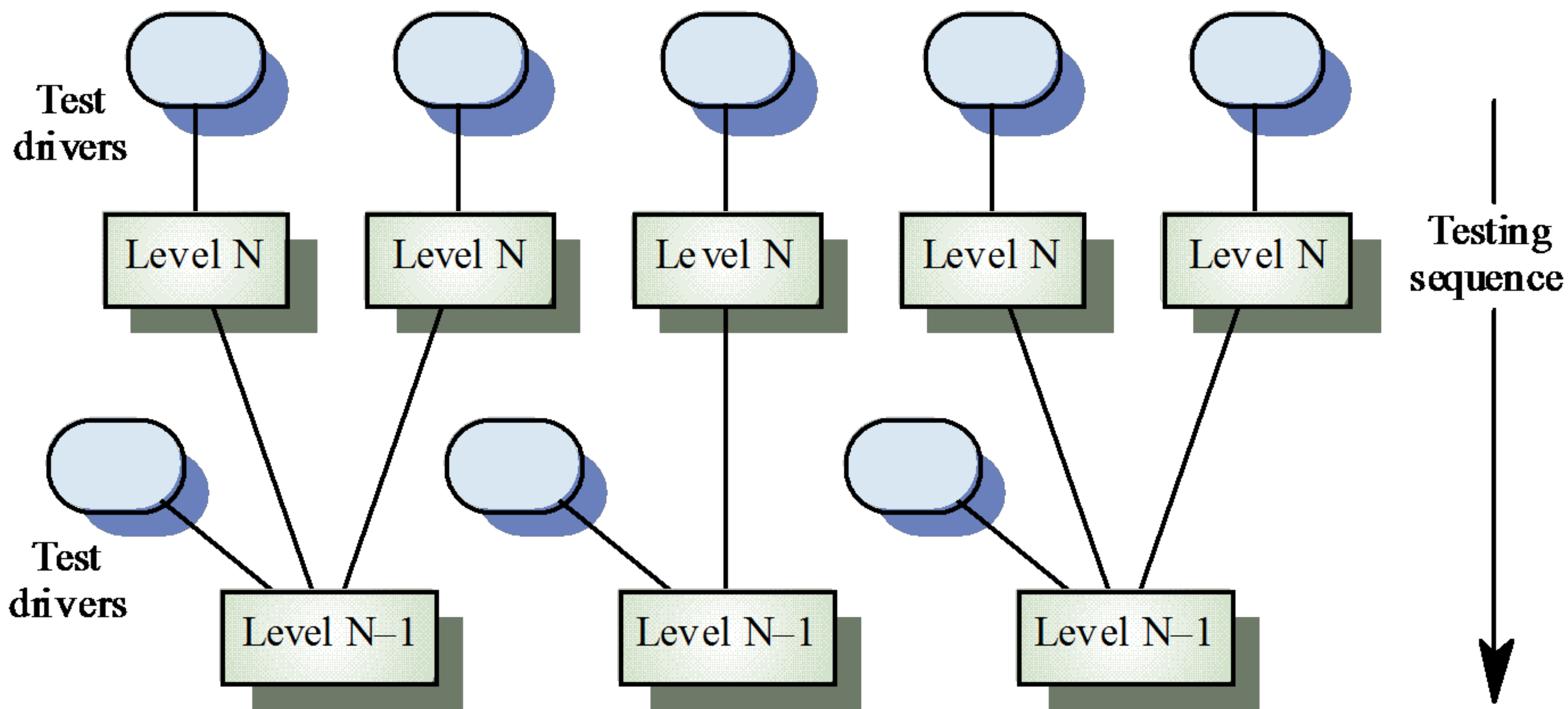
Top-Down Testing

- Start with the high-levels of a system and work your way downwards
- Testing strategy which is used in conjunction with top-down development finds architectural errors
  - It demonstrates the feasibility of the system to management
  - Validation can begin early in the testing process
- May be difficult to develop program stubs
  - Consider a function which relies on the conversion of an array of objects into a linked list, this stub is not easy to do

Top Down Testing

- Advantages
  - Is better at discovering errors in the system architecture (verifies major control or decision points early)
  - Can demonstrate a complete function of the system early (if depth-first integration is selected)
- Disadvantages
  - Logistical problems can raise
  - Need to write and test stubs (costly)

Bottom-up testing



Bottom-Up Testing

- Necessary for critical infrastructure components
- Start with the lower levels of the system and work upward
- Needs test drivers to be implemented
- Does not find major design problems until late in the process
- Appropriate for object-oriented systems

Bottom Up Testing

- Advantages
  - Easier test case design and no need for stubs
  - Can starts at an early stage in the development process (not require to wait until the architectural design of the system to be completed)
  - Potentially reusable modules are adequately tested
- Disadvantages
  - User interface components are tested last
  - The program as an entity does not exist until the last module is added
  - Major design faults show up late

Stress Testing

- Exercises the system beyond its maximum design **load** until the system fails
- Stressing the system often causes defects to come to light
- Systems should not fail catastrophically
- Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded

Regression Testing

- Each time a new module is added as part of integration testing, or a bug is fixed (as the results of uncovering errors through testing) in the software, the software changes.
- These changes may cause problems with functions that previously worked.
- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not generated unintended side effects.

Regression Testing

- Regression testing may be conducted
  - Manually by re-executing a subset of all test cases
  - Using automated capture/playback tools – enable software engineer to capture test cases and results for subsequent playback and comparison.
- Test suite
  - A set of individual tests that are executed as a package in a particular sequence
  - Typically related by a testing goal or implementation dependency
- The regression test suite contains three different classes of test cases:
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the software components that have been changed

Regression Testing

- As integration testing proceeds, the number of regression tests can grown quite large
  - The regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.
  - It is impractical and inefficient to re-execute every test for every program function once a change has occurred

Alpha and Beta Testing

- Alpha and beta testing
  - When software is developed as a product to be used by many customers, a process called alpha and beta testing is used to uncover errors that only end-user seems able to find.
- Alpha Testing
  - Conducted at developer's site
  - Conducted by customer
  - Developer looks over the shoulder
  - Developer records the errors and problems
  - Conducted in a controlled environment

Alpha and Beta Testing

- Beta Testing
  - Conducted at one or more customer sites
  - Conducted by end-user
  - Developer is not present
  - Uncontrolled environment
  - Errors may be real or imagined
  - Customer reports the errors
  - Accordingly modifications are made
  - Final product is released
- For commercial products α - β testing is done

Acceptance Testing

- Acceptance tests are performed after system testing if the software is being developed for a specific client
- Acceptance tests are usually carried out by the clients or end users
- Acceptance test cases are based on requirements
  - User manual can be an additional source for test cases
  - System test cases can be reused
- The software must run under real-world conditions on operational hardware/software
- The clients determine if the software meet their requirements

Software Test Plan - Template

- Scope
  - What are to be tested?
- Testing environment
  - Testing site
  - Required hw/sw configuration
  - Participating organization
  - Preparation and training requirement of the test team
- The testing details (for each test)
  - Test identification, test class, test level, test case, recording procedure
- Testing schedule
  - Link to project development schedule
  - Preparation → testing → error correction → regression testing

Philosophy of Testing

- Common misconceptions
  - "A program can be tested completely"
  - "With this complete testing, we can ensure the program is correct"
  - "Our mission as testers is to ensure the program is correct using complete testing"

- Questions to be answered
  - What is the point of testing?
  - What distinguishes good testing from bad testing?
  - How much testing is enough?
  - How can you tell when you have done enough?

Clearing up the Misconceptions

- Complete testing is impossible
  - There are too many possible inputs
    - Valid inputs
    - Invalid inputs
    - Different timing on inputs
  - There are too many possible control flow paths in the program
    - Conditionals, loops, switches, interrupts…
    - Combinatorial explosion
    - And you would need to retest after every bug fix
  - Some design errors can’t be found through testing
    - Specifications may be wrong
  - You can’t prove programs correct using logic
    - If the program completely matches the specification, the spec may still be wrong
  - User interface (and design) issues are too complex

Testing for Embedded Software

- Developed on custom hardware configurations
- Tools and techniques of one not applicable on another
- Ad Hoc approach to integration and system testing
- Environments: Host and the Target. Target has little support for software development tools

Current State of Embedded Testing

- Incorrect handling of interrupts
- Distributed communication problems
- Incorrect ordering of concurrent events
- Resource contention
- Incorrect use of device protocols and timing
- Incorrect response to failures or transients

Current Solutions

- Hardware Solutions
  - Attempt to gain execution visibility and program control
  - Bus monitors, ROM monitors, in-circuit emulators
  - Minimal effectiveness for software development, Information gathering on low level machine data

Current Solutions

- Software Solutions
  - Attempts to reduce costs of testing on target
  - Factors: Level of criticality, Test platform availability, test classification etc.
  - Lead to extensive modifications and therefore hence extensive retesting

Problems with Embedded Testing

- Expense of testing process
- Level of functionality on target
- Late discovery of errors
- Poor test selection criteria
- Potential use in advancing architectures