Program Analysis Techniques

Program Analysis Techniques – Static and Dynamic

- **Static**
  - **Analysis without regard for run-time behavior**
- **Dynamic**
  - **Program executed under controlled circumstances and its behavior**

Software

"The software is done. We are just trying to get it to work."

– Taken from a Joint STARS E-8A FSD Executive Program Review

Testing

The systematic attempt to find faults in a planned way.

Testing – Dominating Problem

Test Data Generation

Testing Motivation

- Negative costs
  - Costs of failure
  - Costs of repair
- Positive costs
  - Cost for a software quality assurance program – personnel
  - Cost in adding computing resources for formal testing activity
- Benefits
  - Better user acceptance
  - Better opportunity for re-use due to higher quality
  - Improved "psychology" surrounding software development process

Software Testing

- Goal:
  - To affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances
  - Maximize the number of discovered faults, allowing developers to correct them and increase the reliability of the system

- Basic Principle:
  - Controlled execution of a program with know inputs and outputs (both predicted and observed) combined with internal measurement of the behavior of the program

Testing

- Process of executing a program with the intent of finding an error
- Cannot show the absence of defects – can only show that software defects are present
- Good test is one that has a high probability of finding an as-yet undiscovered error

Testing

- A successful test is one that uncovers an as-yet undiscovered error
- Testing begins at the module level and works outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing and debugging are different but debugging must accompany testing
  - Debugging: Locating an error and correcting it

Techniques to Increase Reliability

- Fault avoidance techniques
  - Techniques that try to detect faults statically
  - Tries to prevent insertion of faults into the system before it is released
- Fault detection techniques (debugging, testing)
  - Uncontrolled and controlled experiments used to identify erroneous states and find underlying faults before releasing system.
- Fault tolerance techniques
  - Assumes that a system can be released with faults and that system failures can be dealt with by recovering from them at runtime

Techniques to Increase Reliability

- Fault avoidance techniques
  - Techniques that try to detect faults statically
  - Tries to prevent insertion of faults into the system before it is released
- Fault detection techniques (debugging, testing)
  - Uncontrolled and controlled experiments used to identify erroneous states and find underlying faults before releasing system.
- Fault tolerance techniques
  - Assumes that a system can be released with faults and that system failures can be dealt with by recovering from them at runtime

Related Terms

- Reliability
  - A measure of success with which the observed behavior of a system conforms to the specification of its behavior.
- Software reliability
  - Probability that a software system will not cause system failure for a specified time under specified conditions (IEEE Std. 98)
- Failure
  - Any deviation of the observed behavior from the specified behavior
- Erroneous state (error)
  - System is in a state such that further processing by the system will lead to failure, which causes the system to deviate from its intended behavior.
- Fault (bug)
  - Mechanical or algorithmic cause of an erroneous state

Fault Detection Techniques

- Reviews
  - Walkthrough
    - Developer presents code to review team
    - Review team comments
    - Formalism
  - Inspection
    - Similar to walkthrough
    - Less formal
  - Reviews have been shown to be effective at detecting faults

- Testing
  - Tries to create failures in a planned way
- Debugging
  - Assumes that faults can be found by starting from an unplanned failure

Testing Concepts

- Component
  - Part of a system that can be isolated for testing
  - Can be object, group of objects or subsystems

- Test Oracle
  - Mechanism to determine the expected behavior

Testing Concepts

- Test Case
  - Set of inputs and expected results that exercises a component with purpose of causing failures and detecting faults
- Test Stub
  - Partial implementation of components on which the tested component depends
- Test driver
  - Partial implementation of a component that depends on the tested component
- Correction
  - Change to a component
  - Purpose is to repair a fault

Testing by Process Phase

• Requirements/Specification	• Include rough indications of appropriate test data
• Design	• Explicit requirements that design meets clearly stated – minimum standards of testability
• Implementation	• Static analysis
• Maintenance	• Extent of regression testing

Software Errors – Why?

- Specification may be wrong
- Specification may specify something that is physically impossible given hardware and software prescribed by customer
- System design may be at fault
- Program design may be at fault
- Program code may be wrong

Errors in Program Code

- Algorithmic
  - An error in which program module’s algorithm or logic does not produce proper output for a given input due to mistakes in processing steps
    - Branching too soon
    - Branching too late
    - Testing for wrong condition
    - Forgetting to initialize variables or set loop invariants
    - Forgetting to test for particular condition (x/0)
    - Comparing variables of inappropriate data types

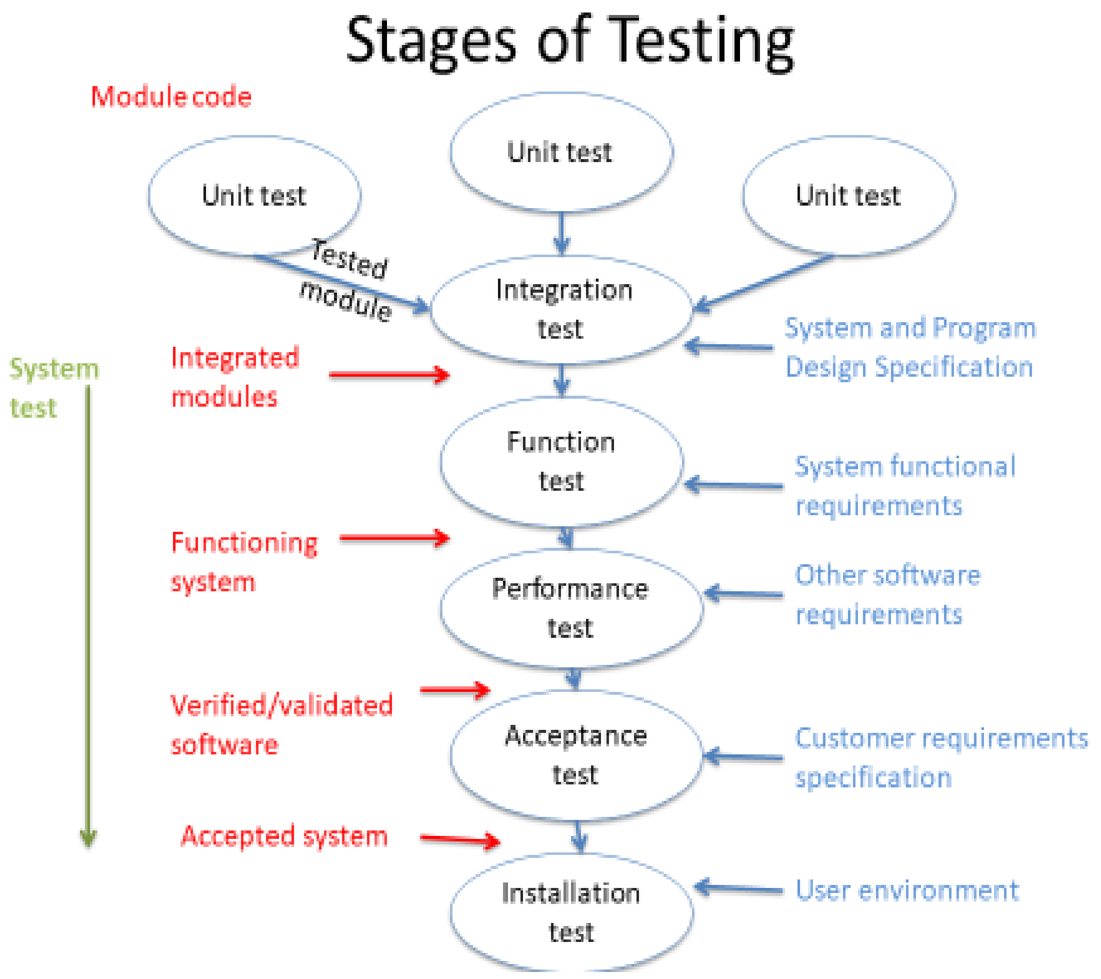
Errors in Program Code- Cont.

- Syntax
- Computation and precision
  - Formula wrong or does not compute to required degree of accuracy
- Documentation
  - Documentation does not match what the program actually does
- Stress or overload errors
  - Data structures filled past their specified category

Errors in Program Code- Cont.

- Capacity or boundary errors
  - When performance of system becomes unacceptable as the activity on the system reaches its specified limit
- Timing or coordination errors
  - Errors in coordinating processes in real-time processing
- Throughput or performance errors
  - When system does not perform at speed prescribed by requirements
- Recovery errors
  - Inability to recover from errors in an acceptable manner
- Hardware and system software errors
  - Documentation for supplied hardware and software does not match operation
- Standards and procedure errors
  - Error in local standards and practices

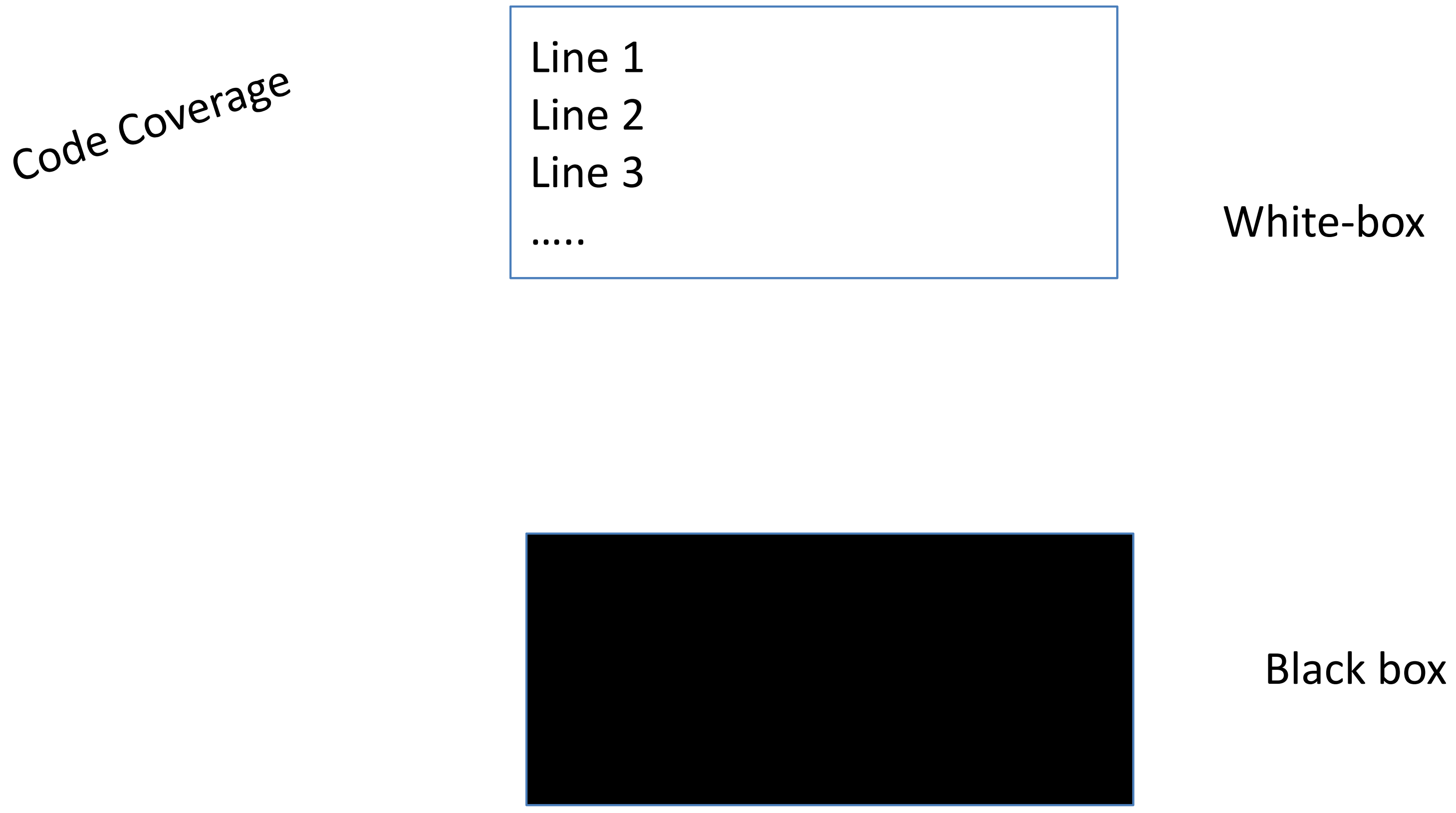
Stages of Testing



Unit Testing

- Unit testing – testing of individual modules
  - Each program module is tested as a single program
- Two main approaches
  - White-box testing
    - Testing based on internal structure of code
  - Black-box testing
    - Testing the functional requirements without regard to internal structure of code

Unit Testing



White-Box Testing

- Based on internal structure

- Derived from program logic

White-box Testing

- Statement coverage
  - each elementary statement of program executed at least once
- Branch coverage
  - Every possible outcome of every decision
- Path testing
  - May be infinite number of paths in program

White-box Testing

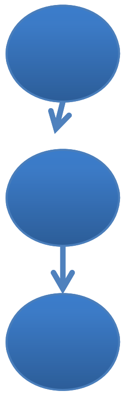
- Condition coverage
  - Every condition in a decision statement must take on all possible outcomes
    - If A = B or B = C
- Domain testing
  - Picks test points on and slightly off a given border

Path Testing

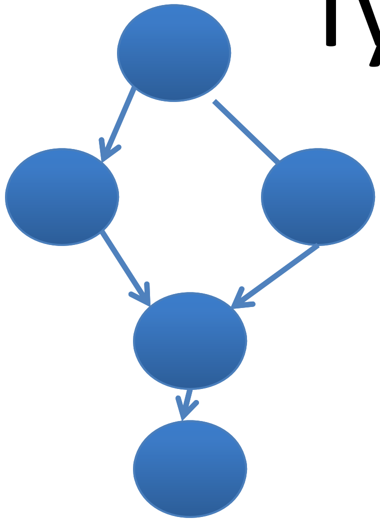
- McCabe’s Theory - Cyclomatic Complexity
  - Use cyclomatic complexity to determine # of independent paths in the basis set of a program
- Independent path
  - Path that introduces at least one new set of processing statements or a new condition
- Derive program flow graph

Types

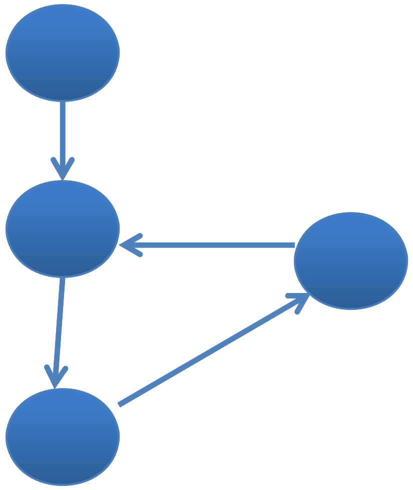
Sequence

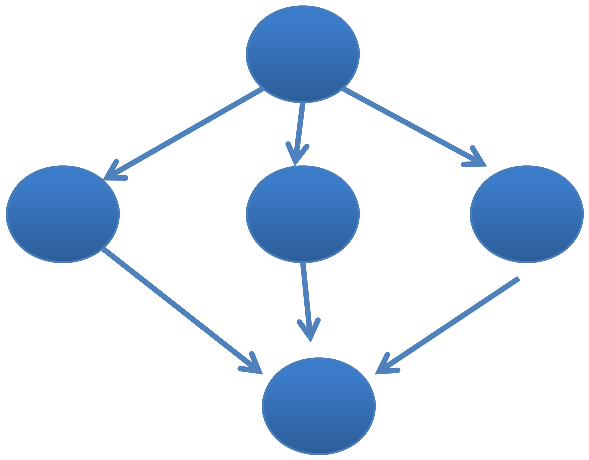


If Then Else



DoWhile





Case

McCabe's Cyclomatic Complexity

$V(G) = e - n + 2p$

Where

- e = number of edges
- n = number of nodes
- p = number of connected components of program graph G

McCabe’s Cyclomatic Complexity

- $V(G) \geq 1$
- V(G) is the maximum number of linearly independent circuits in G
- Adding or subtracting a functional statement to G does not affect V(G)
- In general, complexity of a collection c_j of h programs is sum of the h V(cj)s.
-

Cyclomatic Complexity

- Number of predicates plus one
- Count of # of regions in connected graph if graph is planar
- Maximum # of linear independent paths

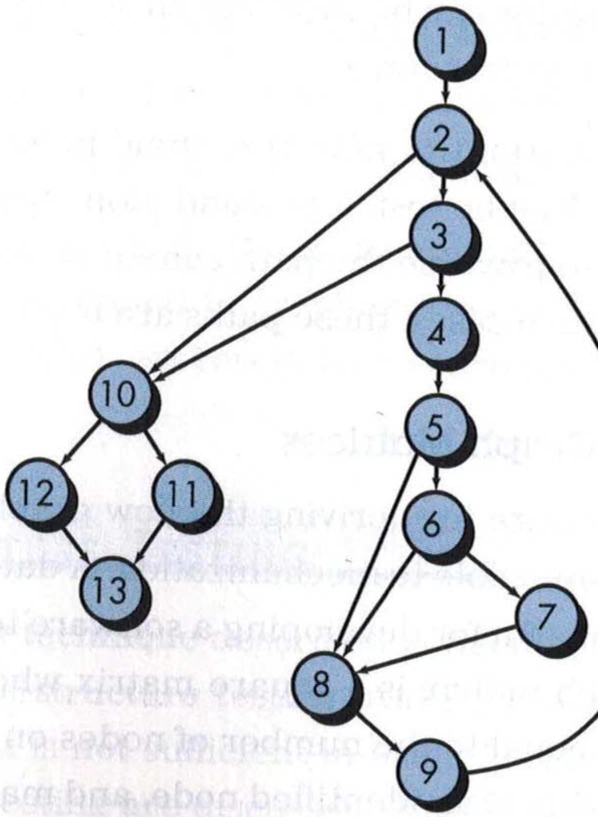
```
PROCEDURE average;

   *  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;
     ⎧ i = 1;
  1  ⎨ total.input = total.valid = 0;          2
     ⎪ sum = 0;
     ⎩ DO WHILE value[i] <> -999 AND total.input < 100     3
  4     increment total.input by 1;
        IF value[i] > = minimum AND value[i] < = maximum     6
  5
              ⎧ THEN increment total.valid by 1;
           7  ⎨       sum = s sum + value[i]
              ⎩ ELSE skip
      ⎧ ENDIF
   8  ⎩ increment i by 1;
   9 ENDDO
     IF total.valid > 0     10
  11   THEN average = sum / total.valid;
  12   ELSE average = -999;
  13 ENDIF
  END average
```

Cyclomatic Complexity



Unit Testing in the OO Context

- Class testing
  - Driven by the operations encapsulated by the class and the state behavior of the class
  - It not possible to test a singe operation in isolation but rather as part of a class
  - When subclass inherits an operation it must be tested in the context of the appropriate attributes

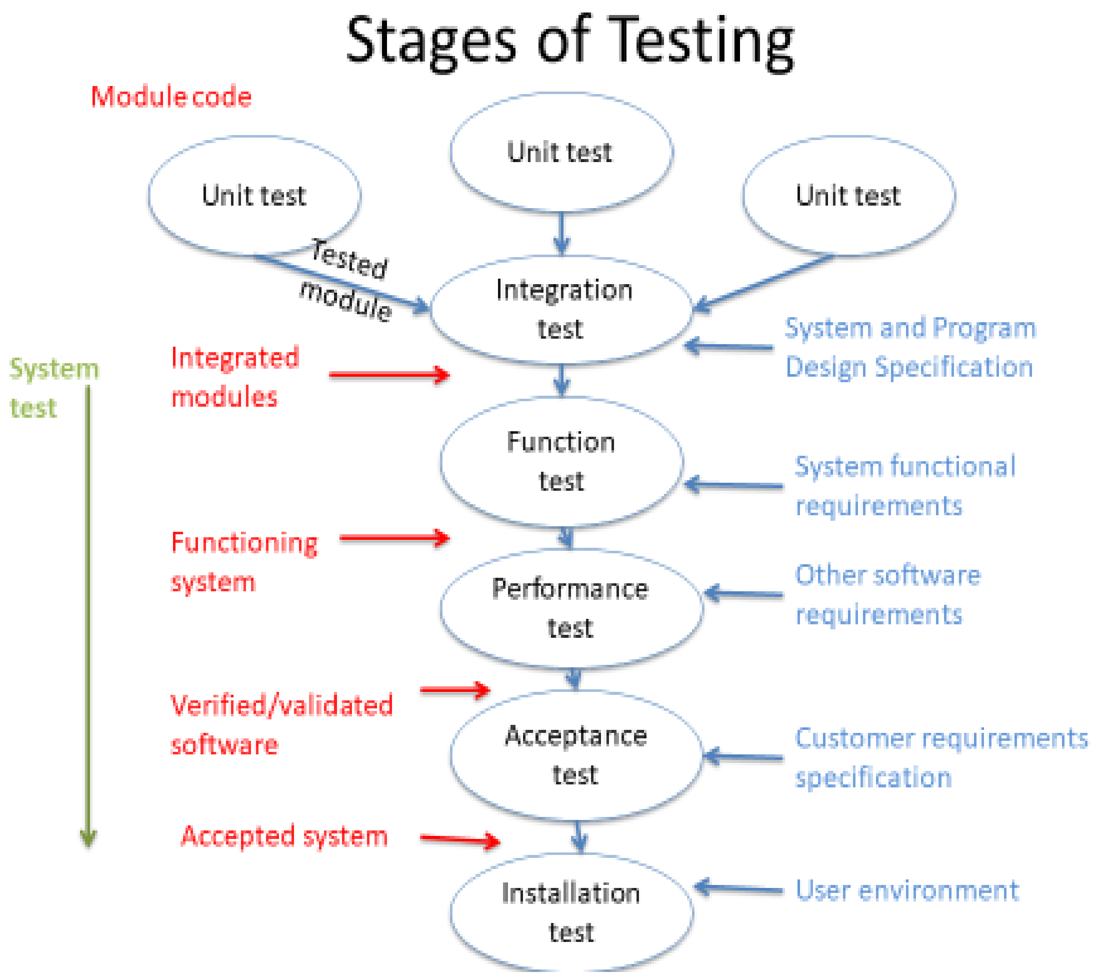
Unit Testing in the OO Context

- Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality

Object Class Testing

- Test coverage of a class includes
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object In all possible states
- Must take into account inheritance

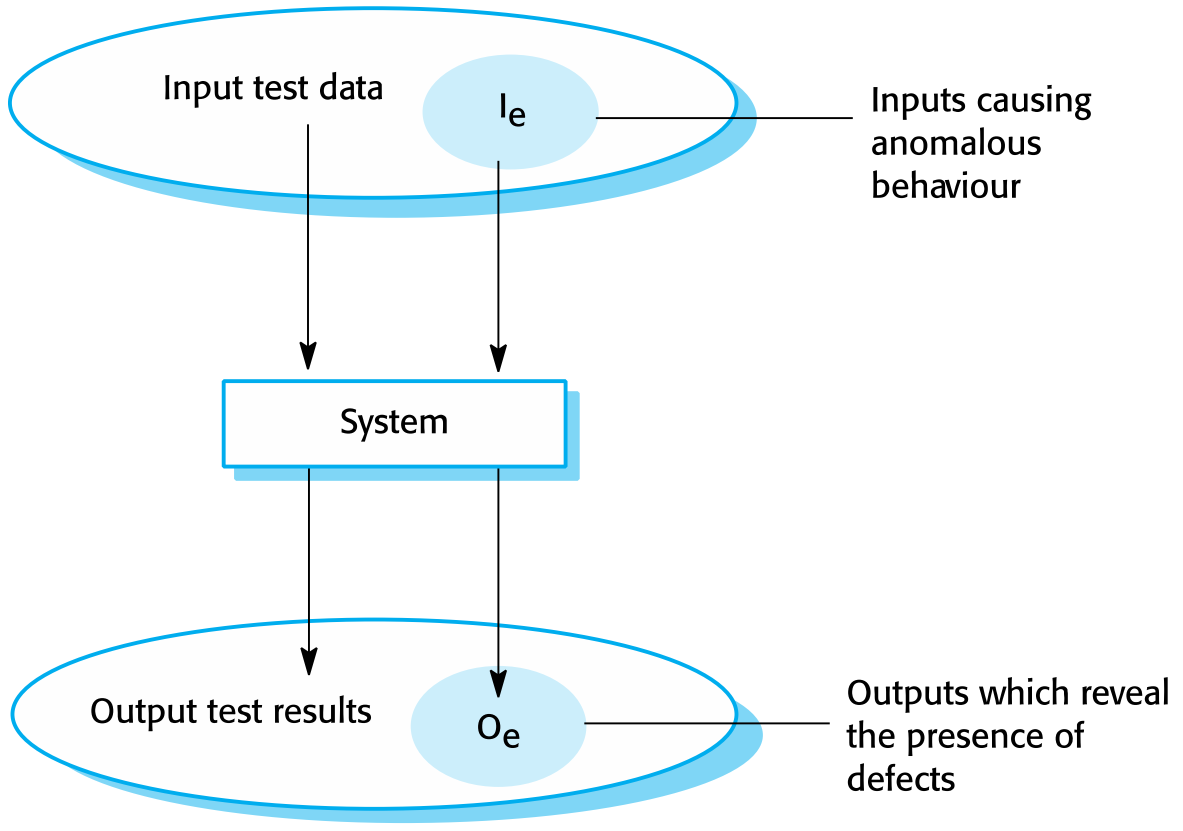
Stages of Testing



Black-box Testing

- Testing the functional requirements without regard to internal structure of the code
- Typically performed after white-box testing
- Equivalence testing
- Boundary value analysis

An input-output model of program testing


Input test data
I_e
Inputs causing anomalous behaviour
System
Output test results
O_e
Outputs which reveal the presence of defects

Black-box Testing – Equivalence Partitioning

- Technique for determining which classes of input data have common properties

- Equivalence relation

  - Let S be a set, let x and y be members of S
  - let R be a relation that indicates whether x is in relation R to y
  - R on set S is $f(S \times S) \longrightarrow \{true, false\}$
  - Special relation: reflexivity, symmetry, transitivity
  - Partitions the set S into mutually exclusive independent classes

- Equivalence class
  - Set of elements of S that are equivalent to each other
  - Operations can be defined on equivalence classes using representatives of the classes

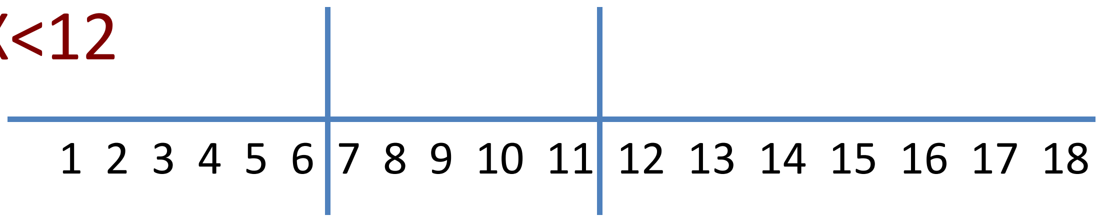
Equivalence Partitioning

- Input equivalence classes

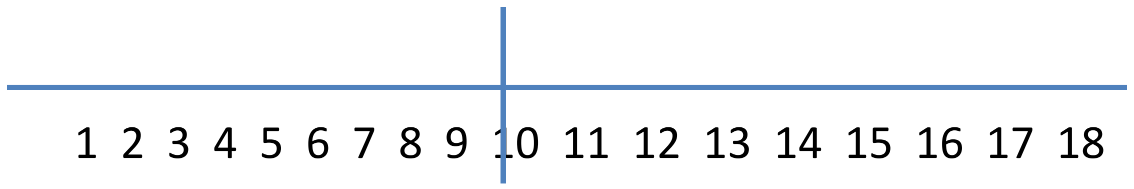
- Guidelines
  - If input condition specifies a **range**
    - one valid class, two invalid classes
  - If input condition requires a specific **value**
    - one valid class, two invalid classes
  - If input condition specifies **member of a set**
    - one valid class, one invalid class
  - If input condition is **Boolean**
    - one valid class, one invalid class

Equivalence Testing

- Range: 6<X<12



  - 3, 9, 14
- Specific value: x = 10
  - {10,3,13}



- Specific member of set: S={4,6,8,}, x = 6
  - {6,9}
- Boolean: x = {True}
  - {true, false}

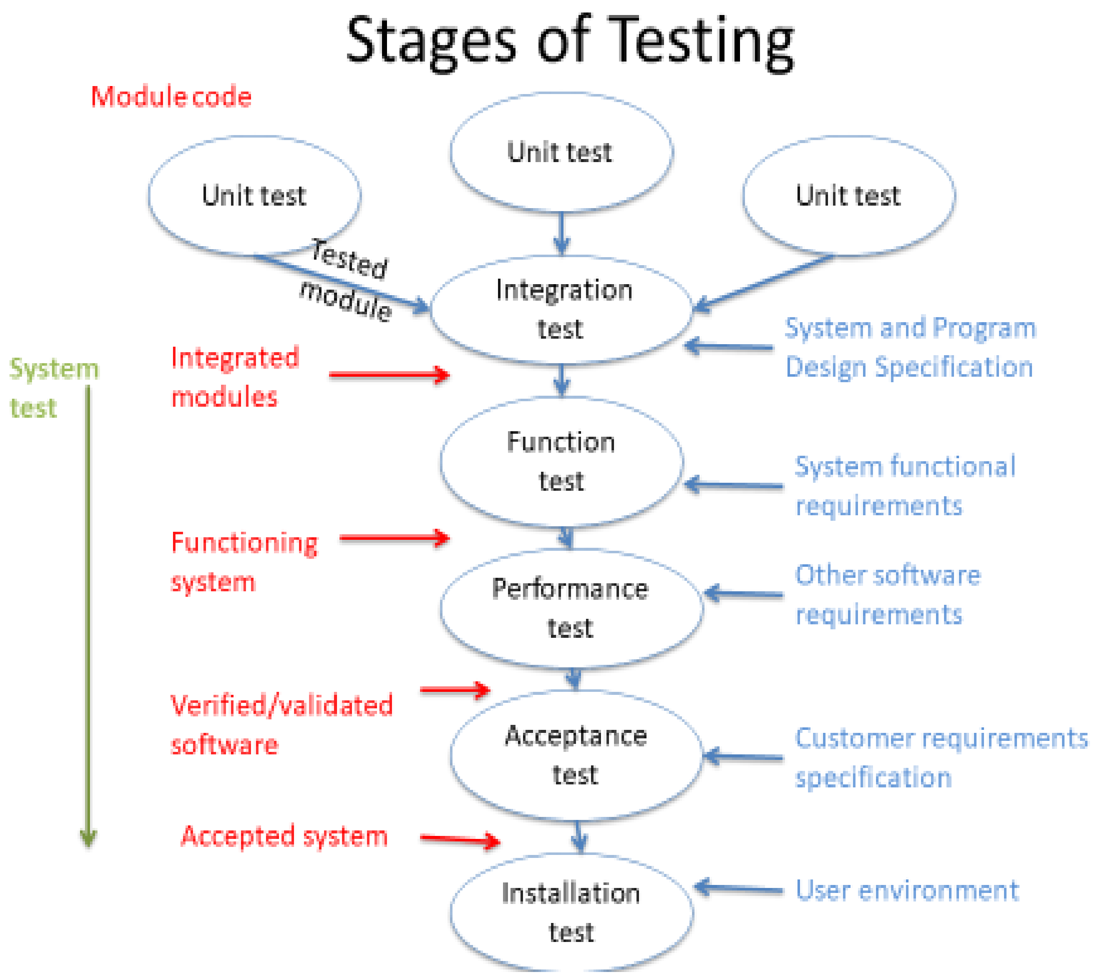
Boundary Value Analysis

- Complements equivalence partitioning but differs
  - Selection of test cases at the edges of the class
  - Derives test cases from both the output domain and the input domain
- Test on output classes and input classes
- If input/output condition specifies range, bounded by a and b
  - use values just above or below a and b
- If input/output condition specifies a number of values
  - test cases should address maximum and minimum numbers
  - also use values above and below maximum and minimum values

Boundary Value Analysis

- Output values
  - {5, 8, 25, 40}
  - Test cases example {4, 5, 40, 41}

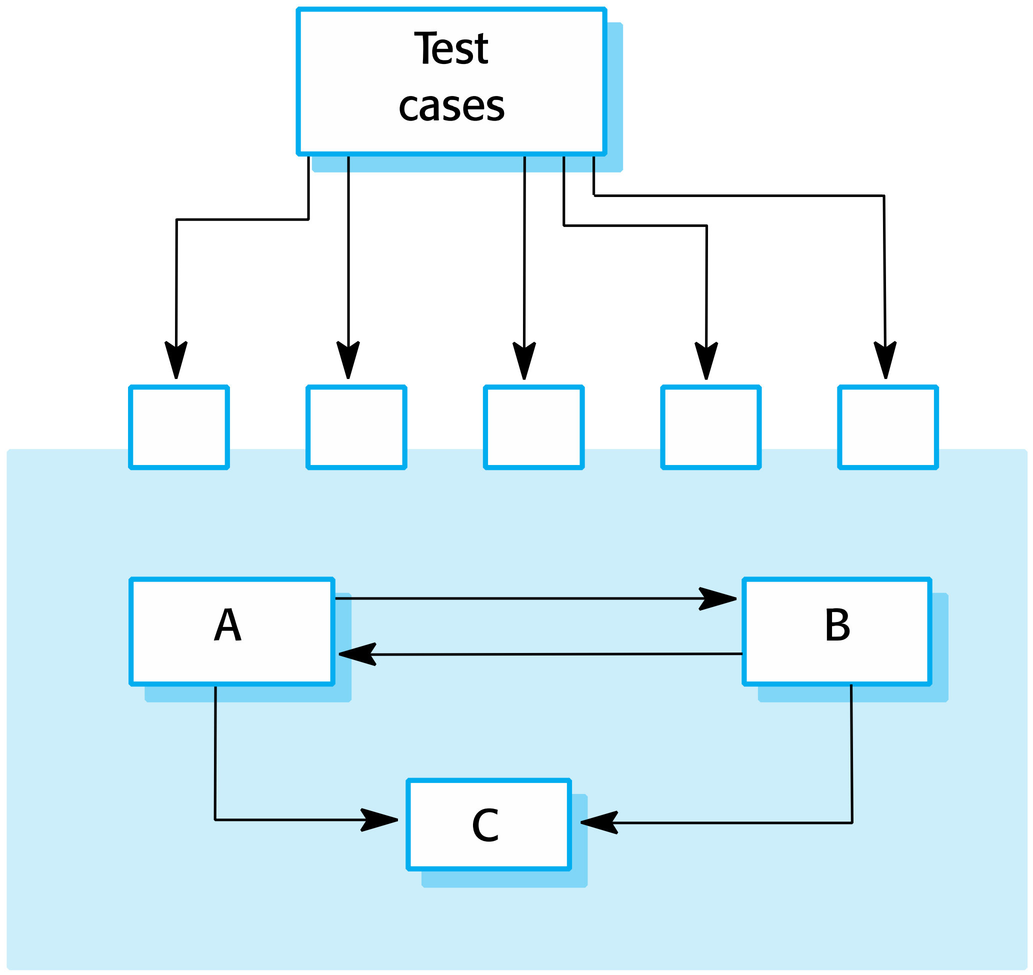
Stages of Testing



Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
  - Parameter interfaces
    - Data or function references passed from one method or procedure to another
  - Shared memory interfaces
    - Block of memory is shared between procedures or functions
    - Often used in embedded systems
  - Procedural interfaces
    - Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - Message passing interfaces
    - Sub-systems request services from other sub-systems by passing a message

Interface testing



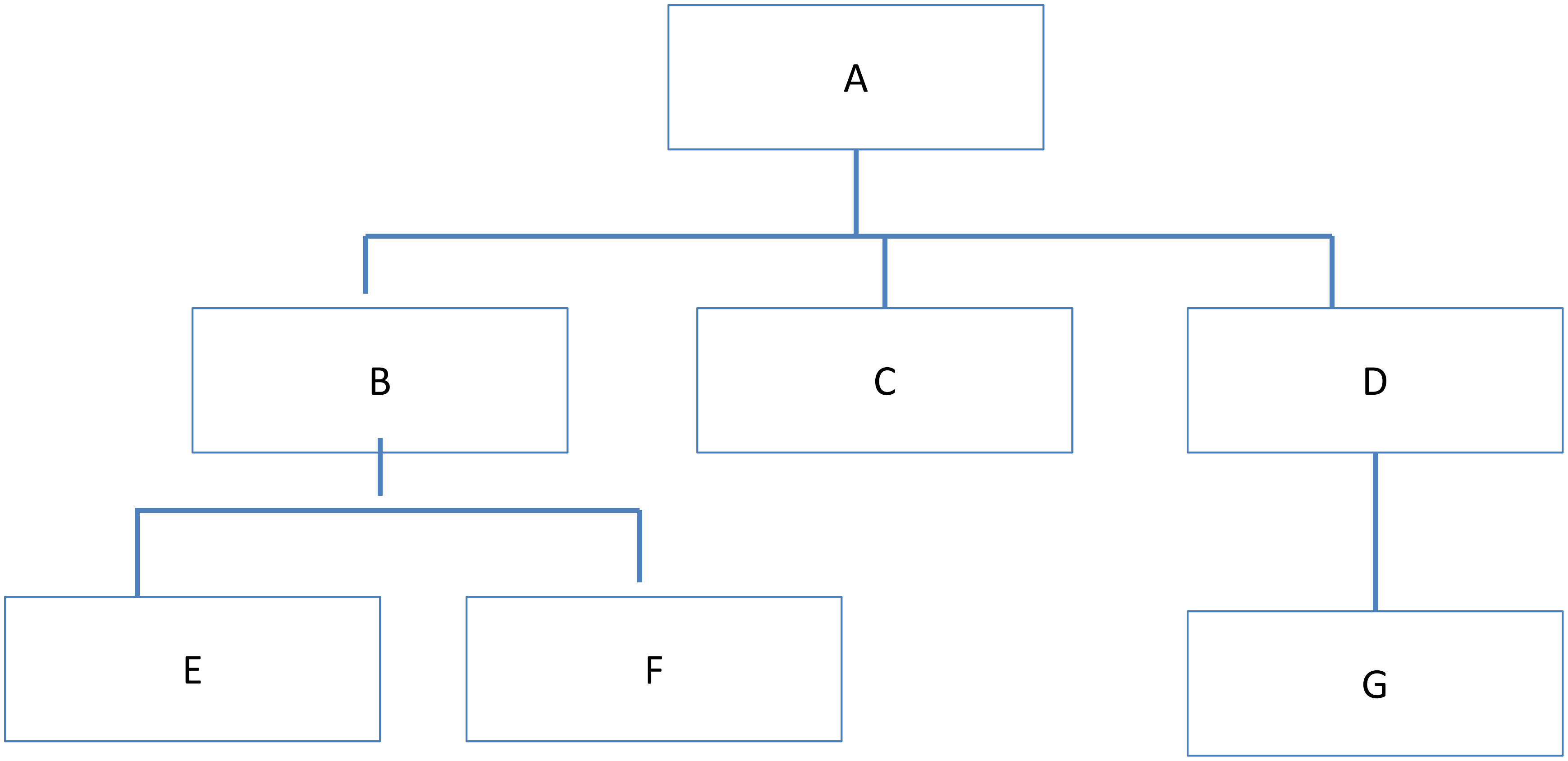
Interface Testing Guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

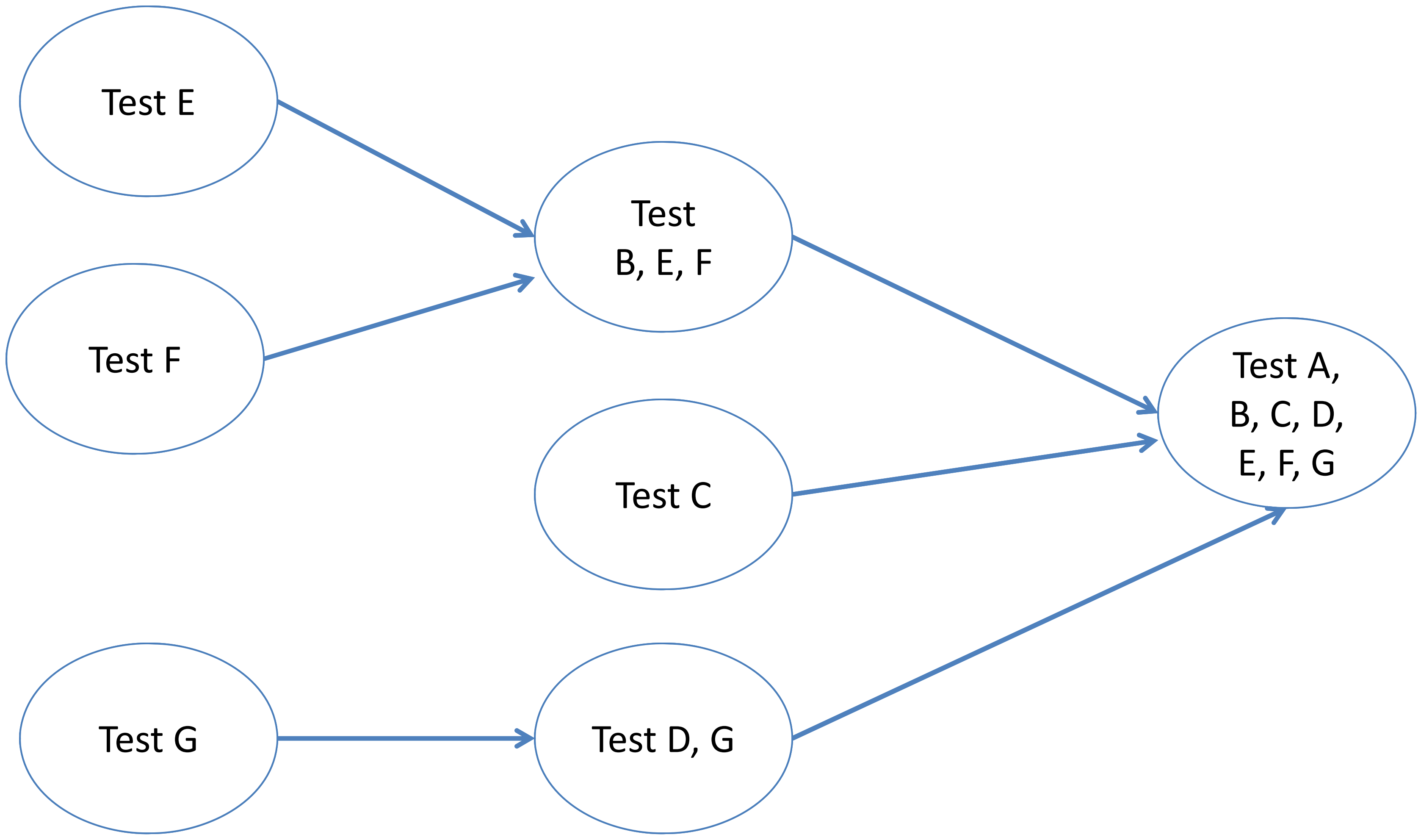
Integration Testing

- Bottom-up ------------------------------Top-Down
- Module driver
  - Routine that calls a module and passes a test case to it

Example



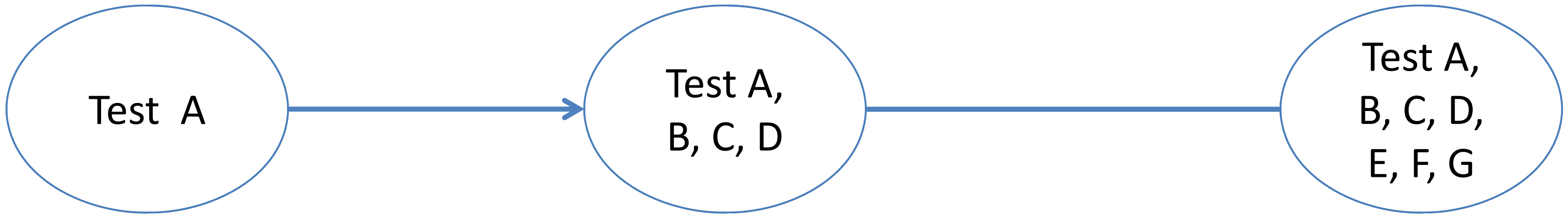
Bottom-up Testing Strategy



Top-Down Approach

- Top level tested by itself
- All modules called by tested modules are combined and tested as a larger unit
- Test Stub
  - Program that simulates the activity of a missing module by answering to the identical calling sequence of the module and passing back output data that allows the testing process to continue

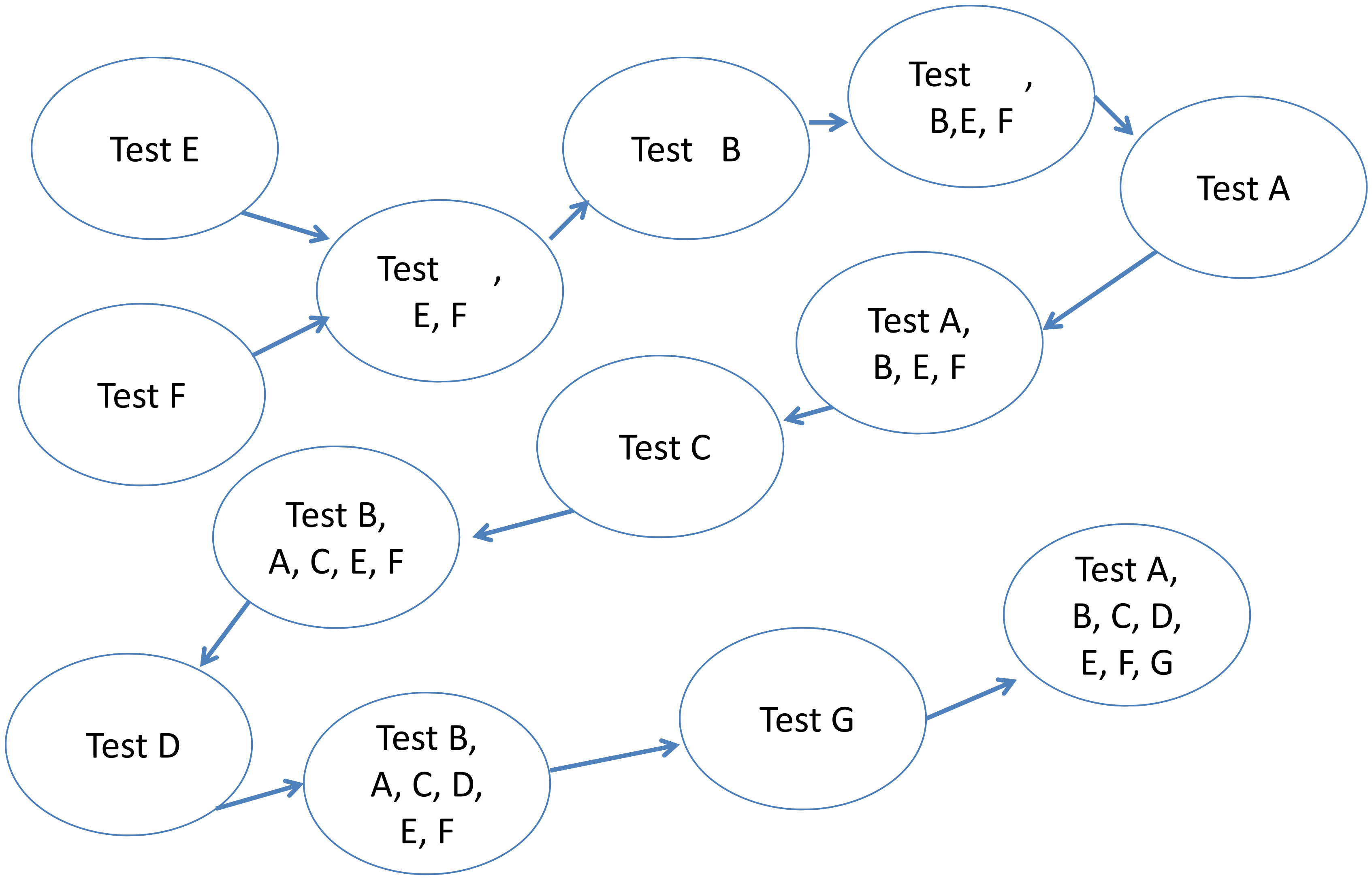
Top-Down Testing Strategy



Incremental Testing

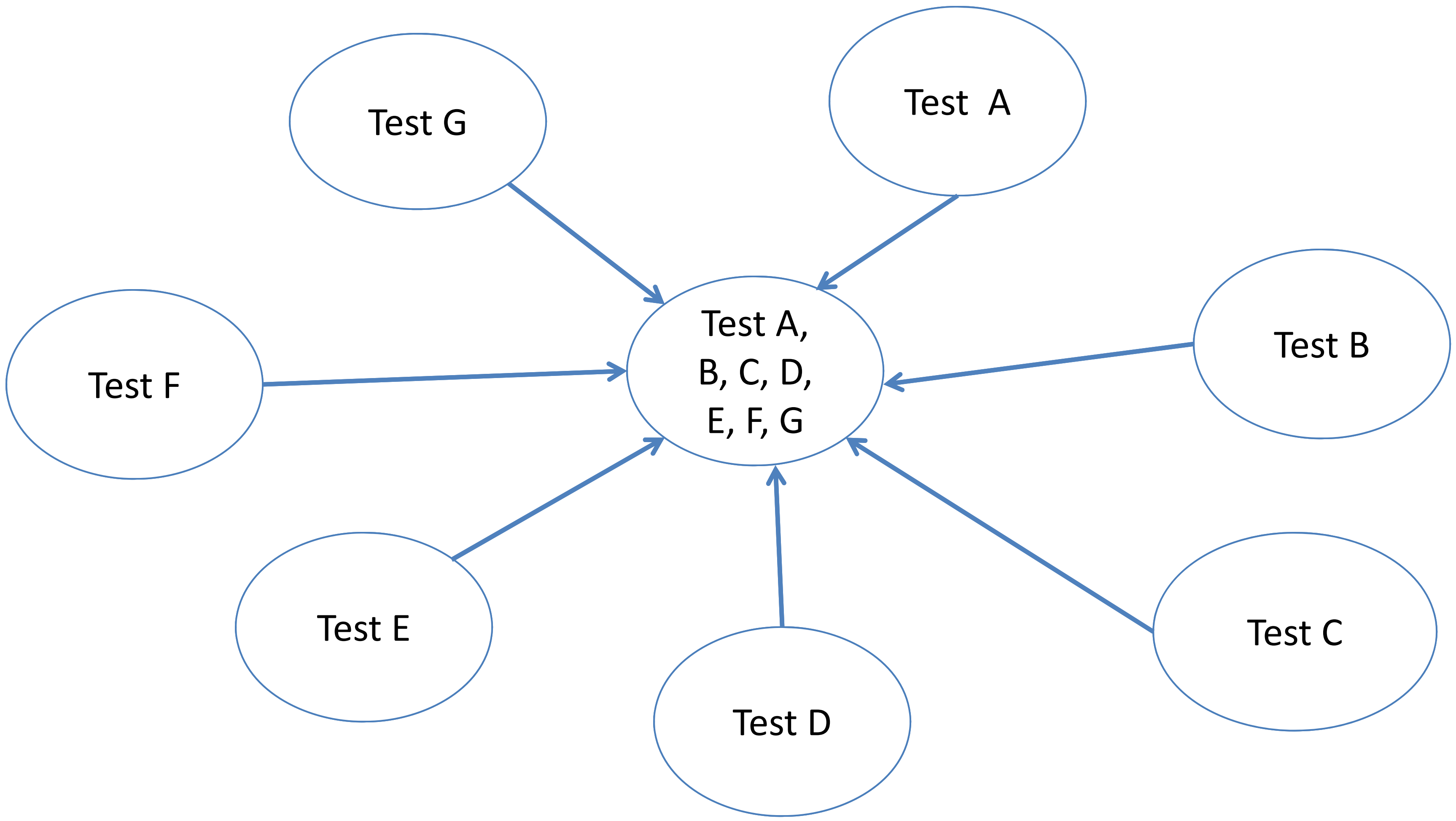
- Add one module at a time
- Driven by system characteristics
- Combination of top-down and bottom-up integration testing.

Incremental Testing Strategy



Big Bang Testing Strategy

Test each module individually



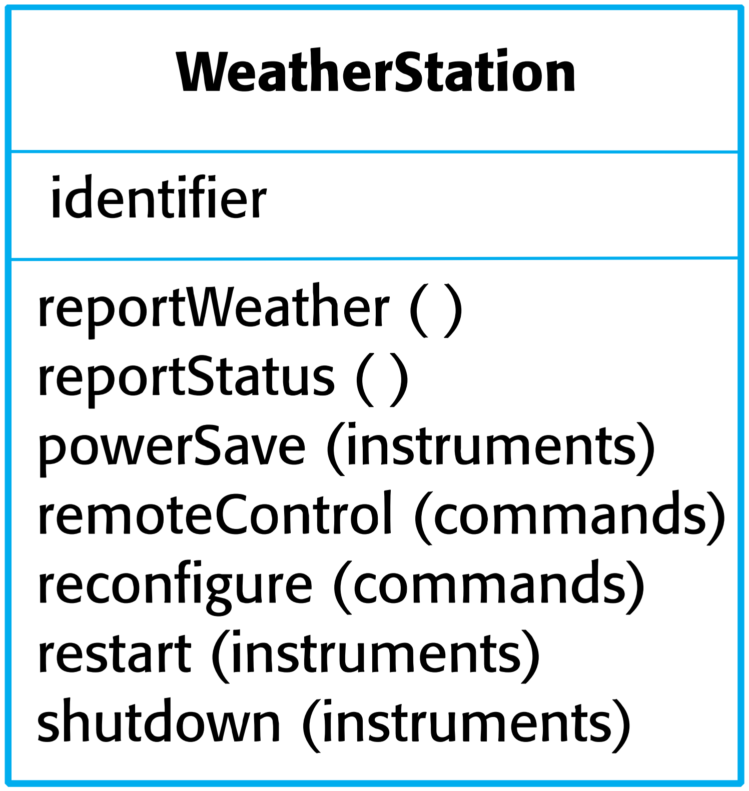
Smoke Testing

- Integration testing approach
- Pacing mechanism for time-critical components
  - Software components integrated into a build
  - A series of tests designed to expose errors in the build
  - The build, integrated with other builds is smoke tested daily
  - Can be a top-down or bottom-up approach

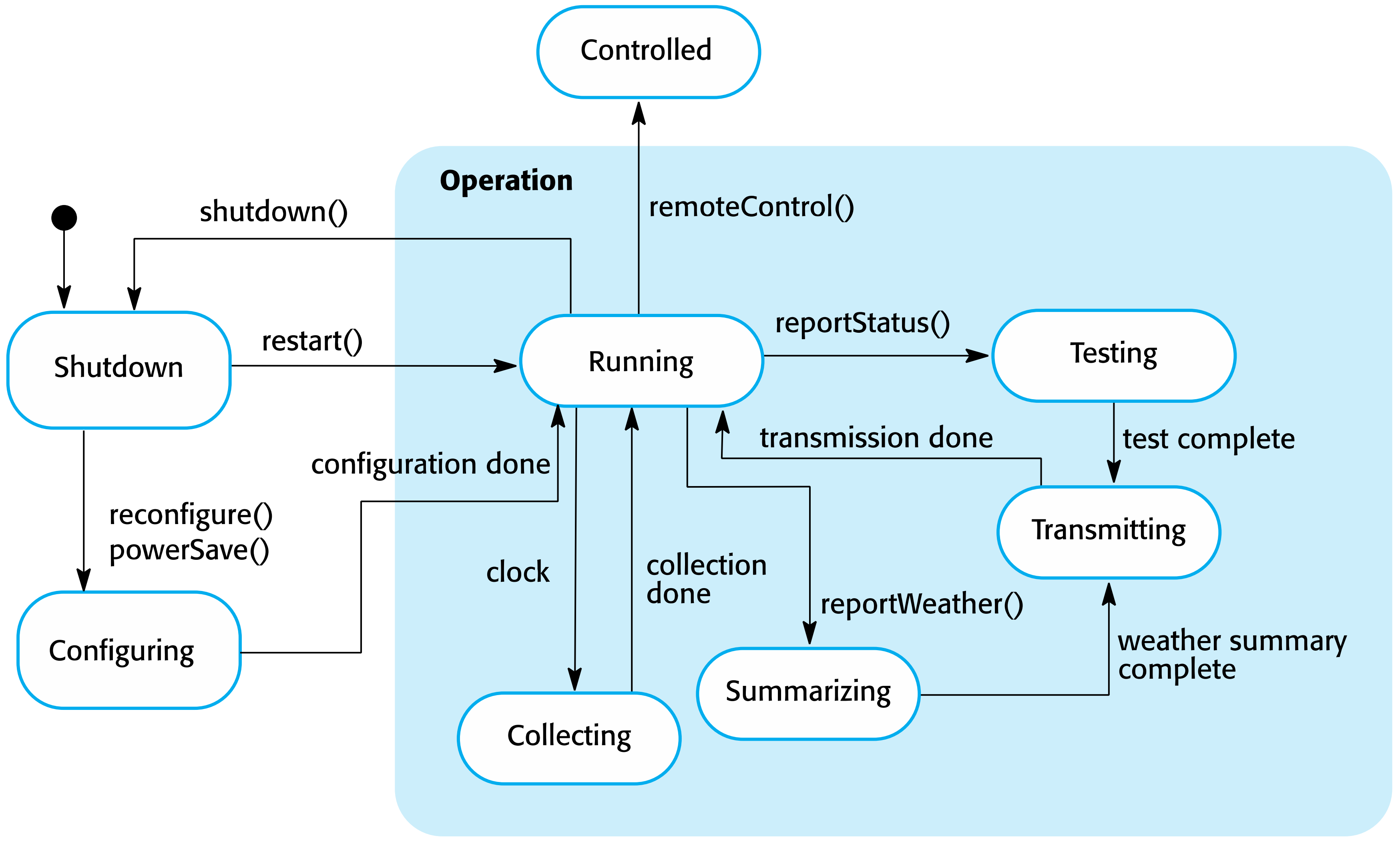
Weather Station Testing

- Need to define test cases for reportWeather, reportStatus, reconfigure, restart, shutdown….
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing ->Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

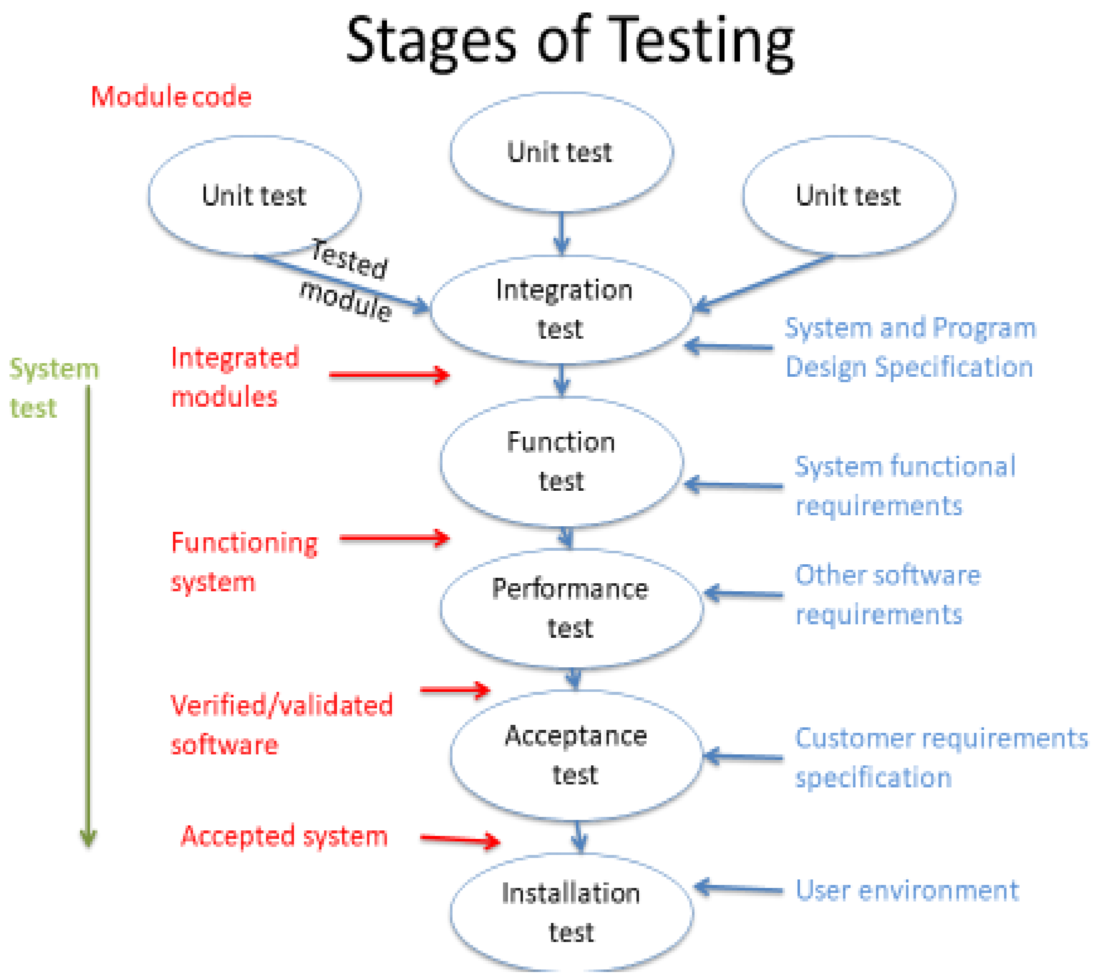
Weather Station Object Interface



Weather Station State Diagram



Stages of Testing



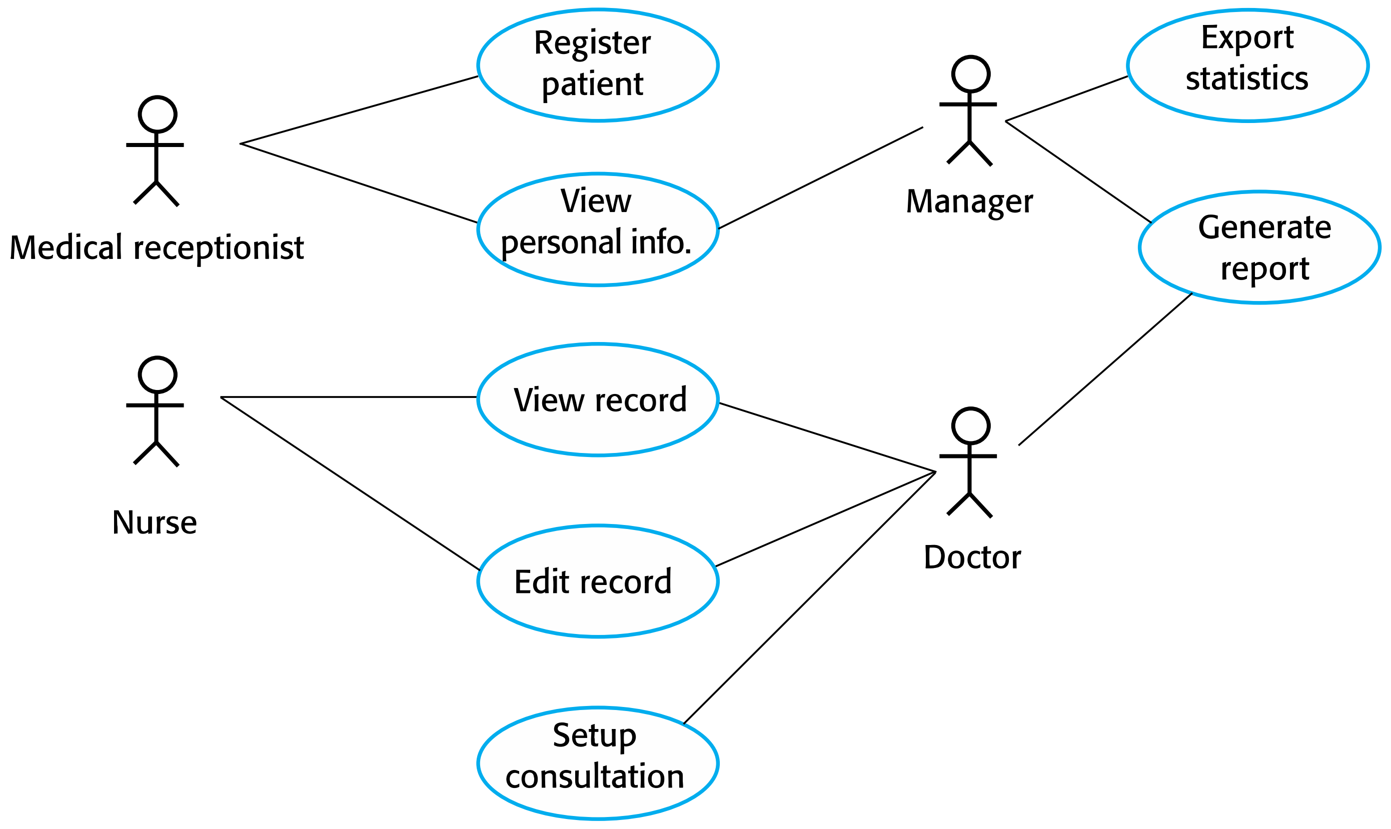
System Testing

- Unit and integration
  - Is the implementation working as designed?
- System
  - Is the problem solved?
  - Steps in system testing
    - Function test
    - Performance test
    - Acceptance test
    - Installation test

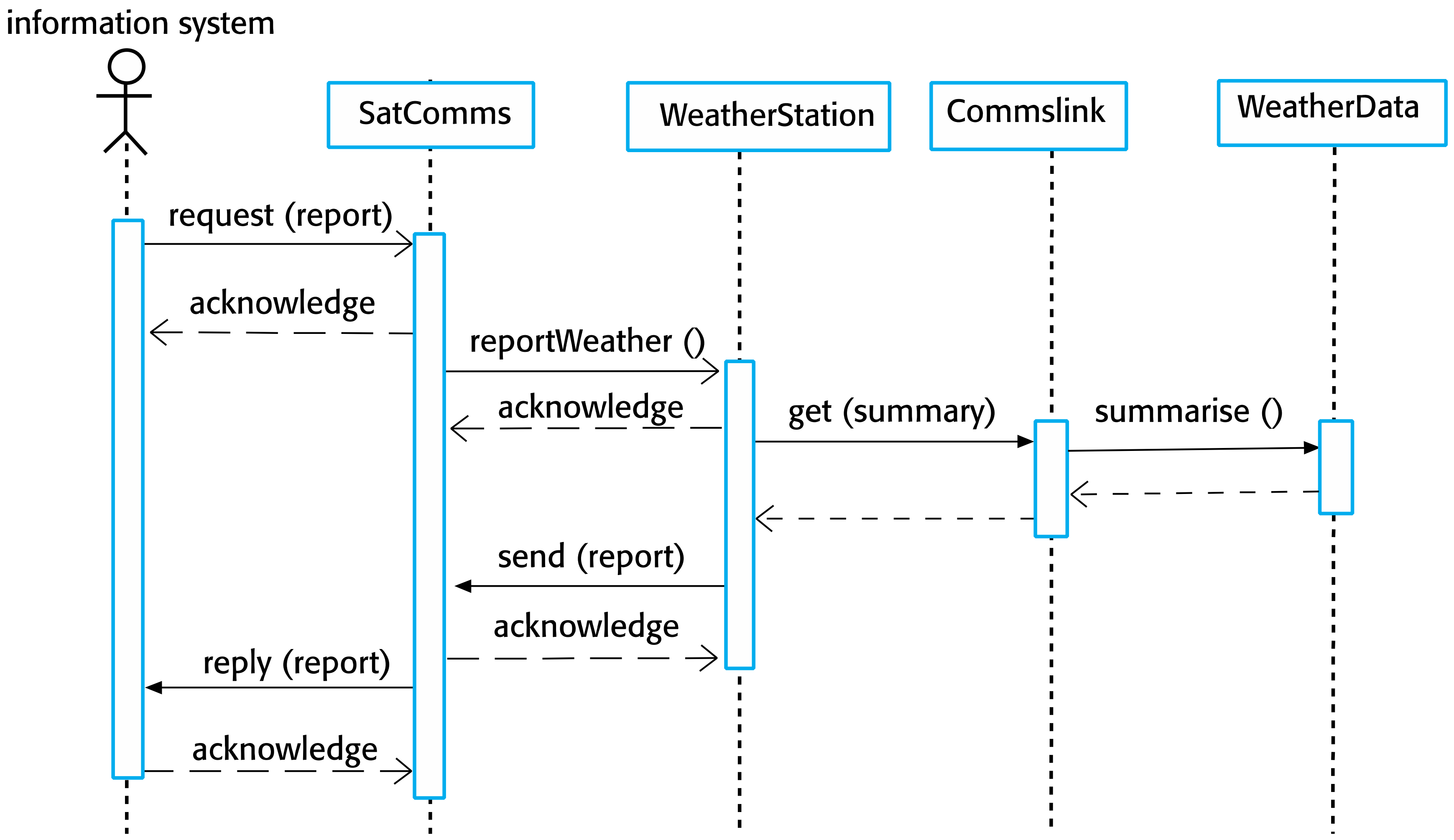
Use-case testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.

Use Cases for the Mentcare System



Collect Weather Data Sequence Chart



Test Cases Derived from Sequence Diagram

- Request has an associated acknowledgement.
  - Report should be returned from the request.
  - Summarized data used to check that report is correctly organized
- Input request for a report to WeatherStation results in a summarized report being generated.

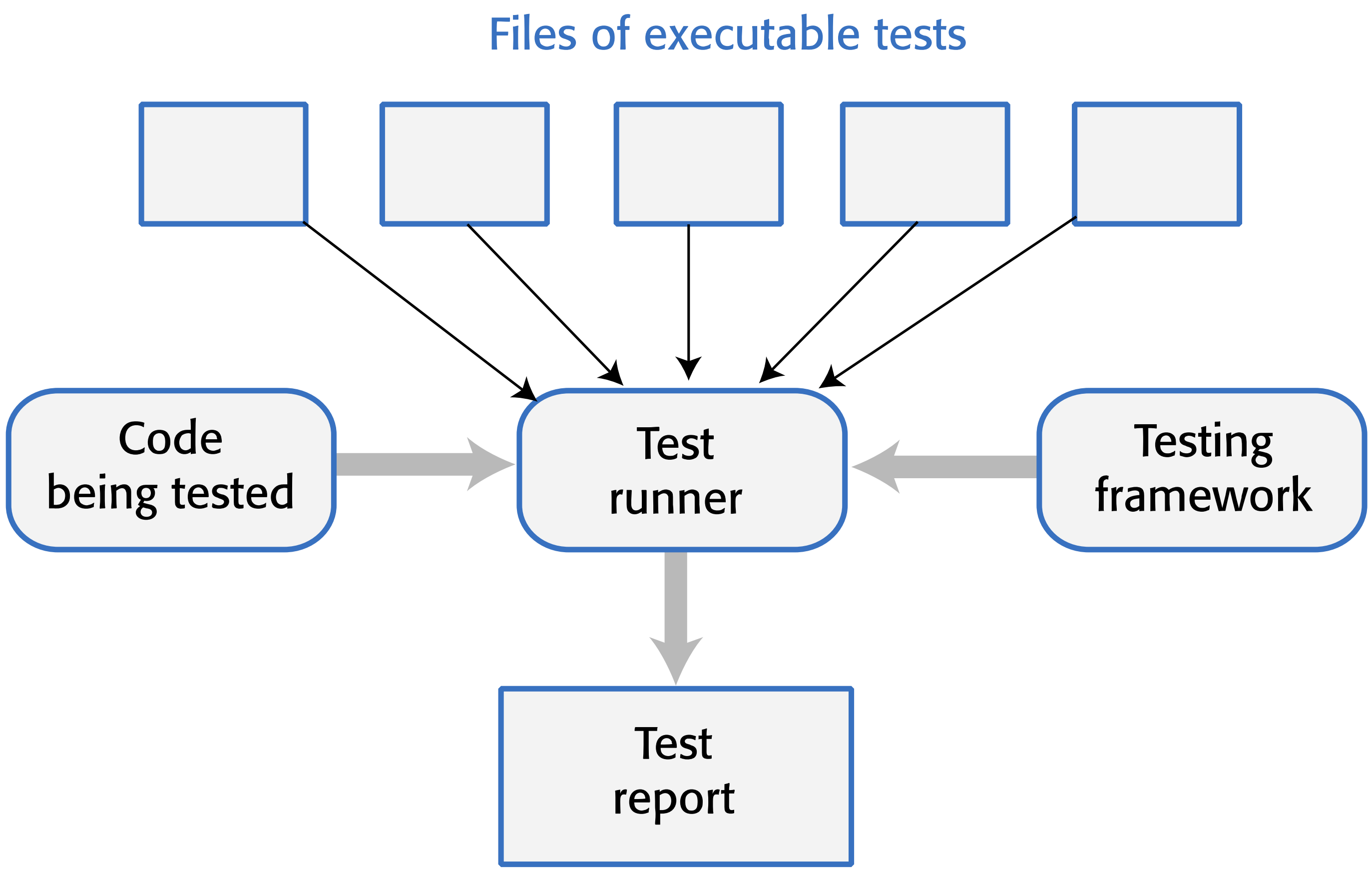
Automated Testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, use of a test automation framework (such as JUnit) to write and run program tests.
- Unit testing frameworks provide generic test classes that are extended to create specific test cases.
  - They can then run all of the tests all the implemented tests and report, often through some GUI, on the success or otherwise of the tests.

Automated Test Components

- A setup part
  - Initialize the system with the test case, namely the inputs and expected outputs.
- A call part
  - Call the object or method to be tested.
- An assertion part
  - Compare the result of the call with the expected result.
  - If the assertion evaluates to true, the test has been successful if false, then it has failed.

Figure 9.4 Automated testing



Automated feature testing

- Generally, users access features through the product's graphical user interface (GUI).
- However, GUI-based testing is expensive to automate so it is best to design your product so that its features can be directly accessed through an API and not just from the user interface.
- The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI.
- Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

Figure 9.6 Feature editing through an API

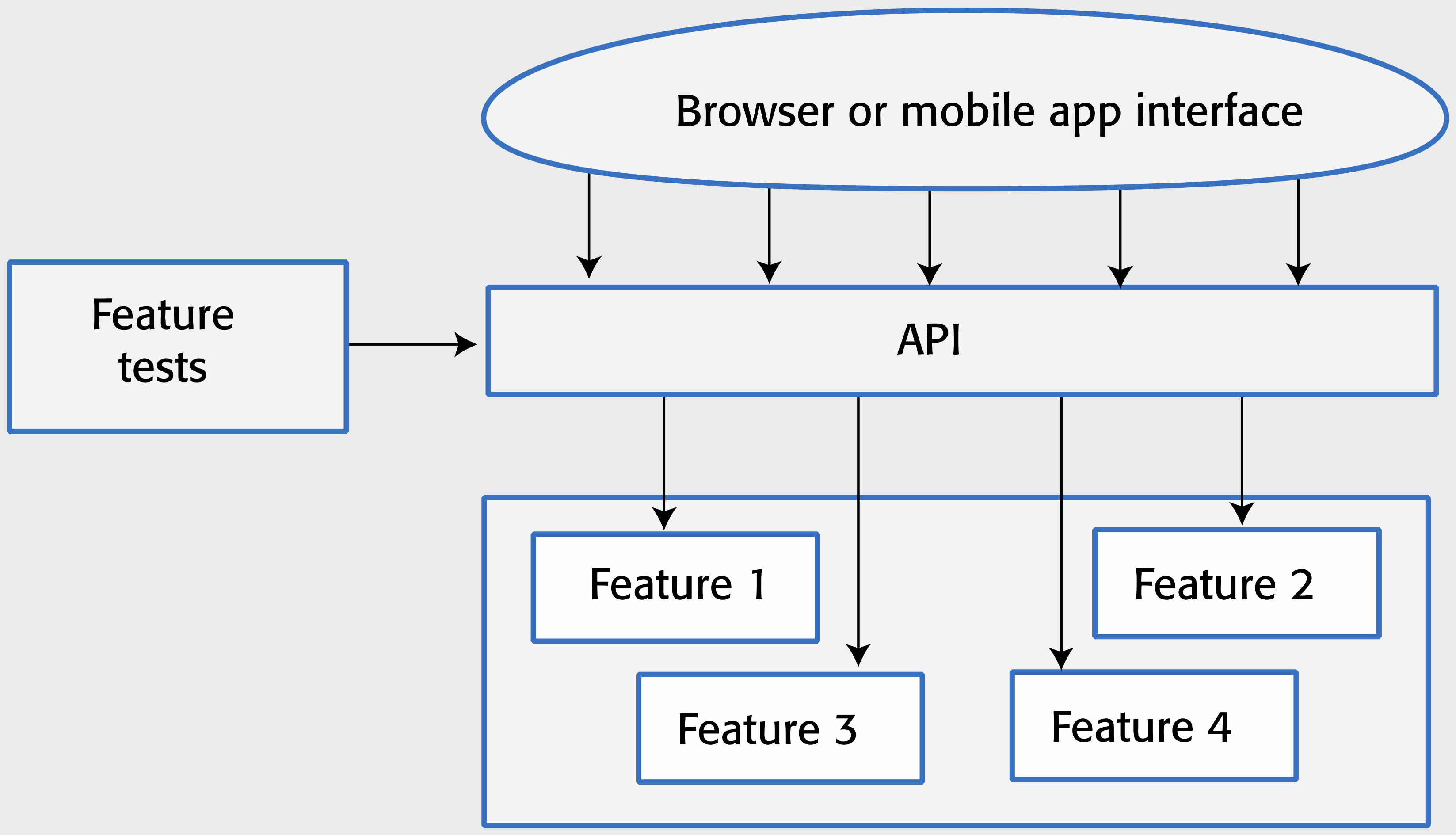
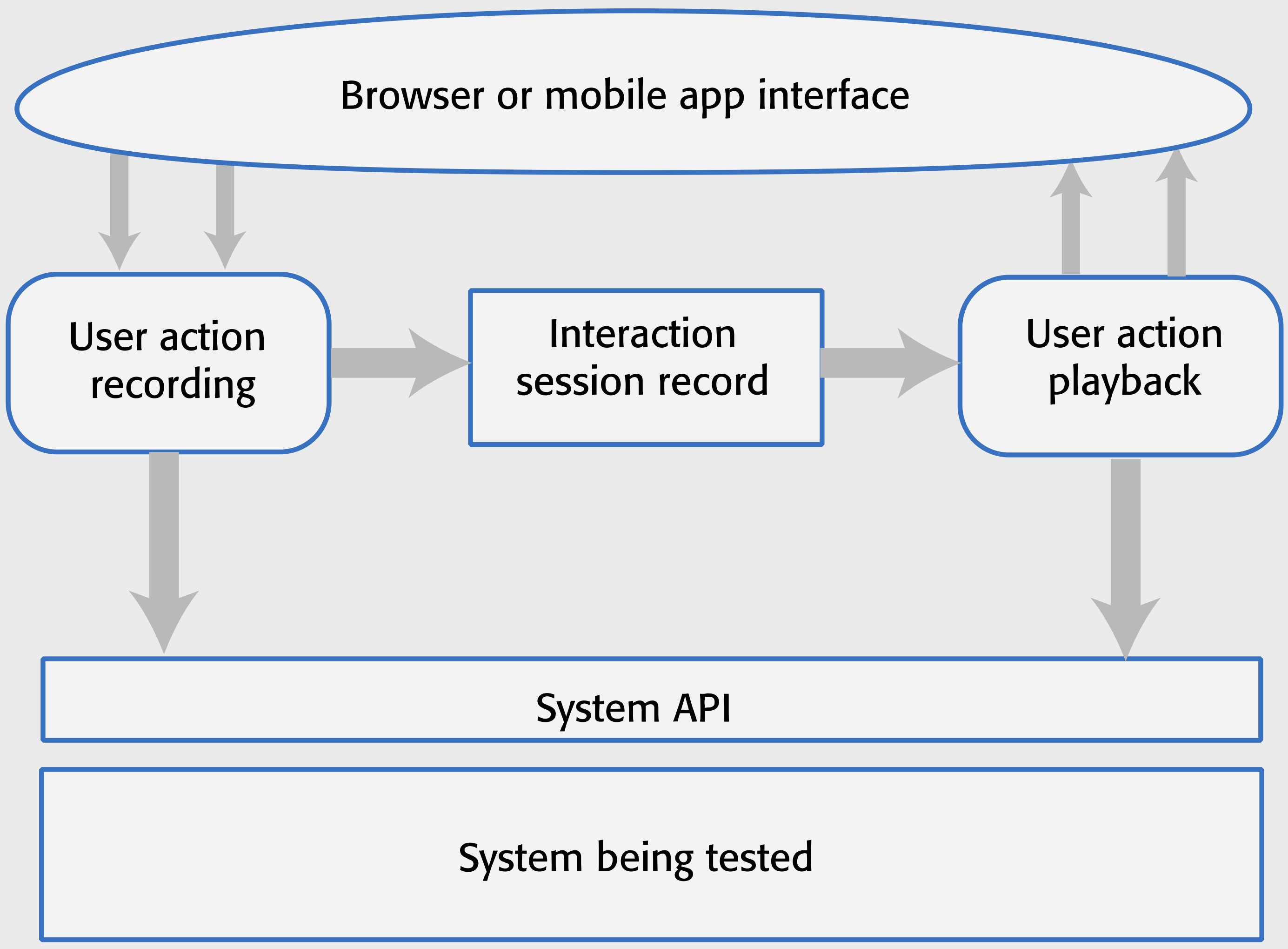


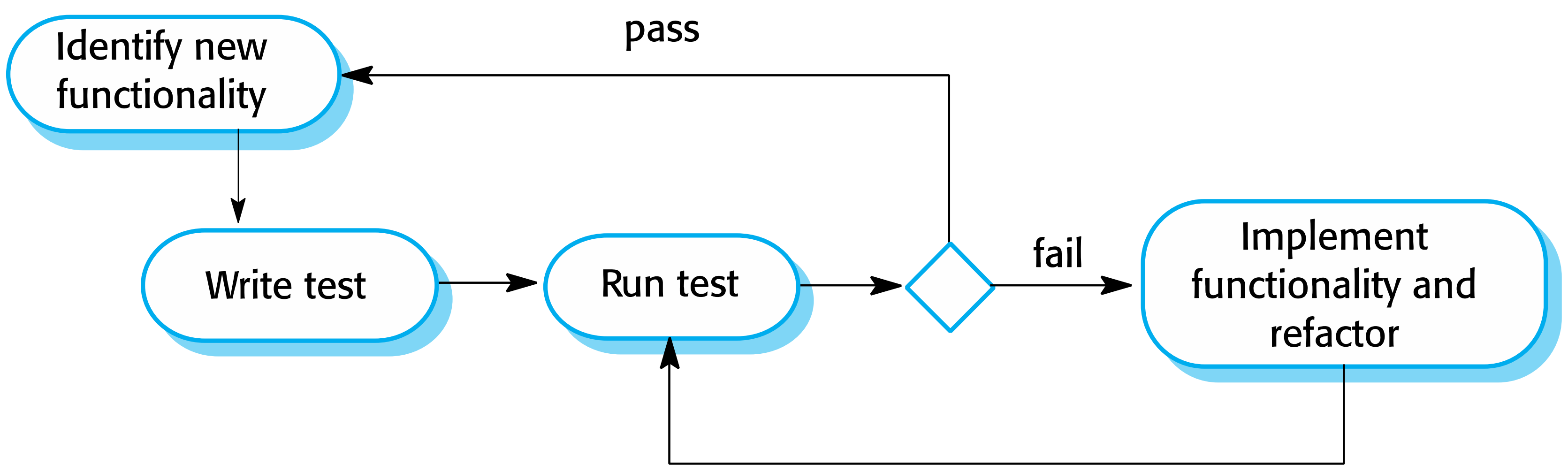
Figure 9.7 Interaction recording and playback



Test-driven Development

- Approach to program development where testing and code development are inter-leaved.
- Tests are written before code, and ‘passing’ the tests is the critical driver of development.
- Code is developed incrementally, along with a test for that increment.
  - Next increment is not started until code for current increment passes its test
- TDD was introduced as part of agile methods such as Extreme Programming.
  - It can also be used in plan-driven development processes.

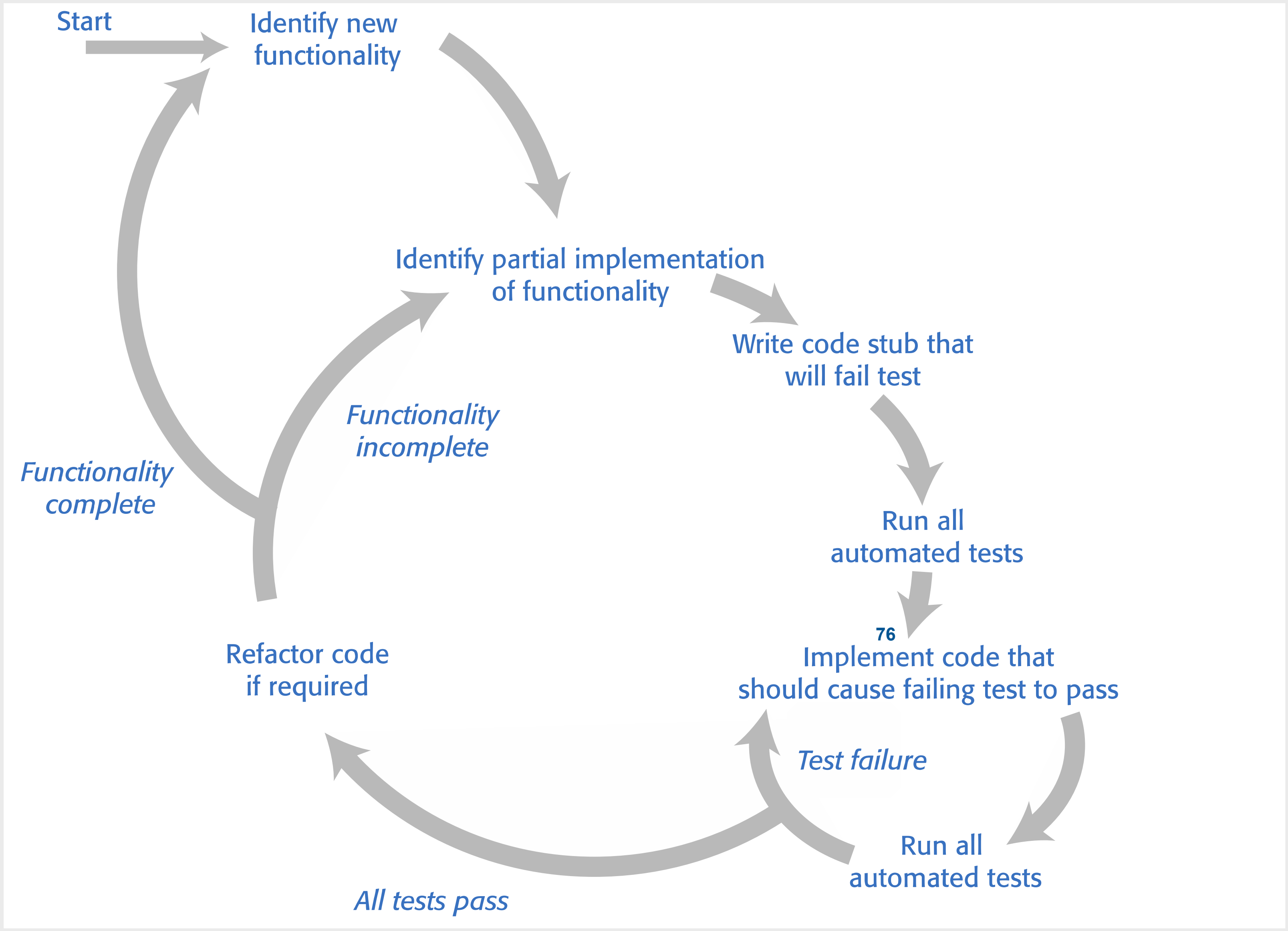
Test-driven Development


Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

TDD Process Activities

- Start by identifying the increment of functionality that is required.
  - Should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement it as an automated test.
- Run the test, along with all other tests that have been implemented. Implement the functionality and re-run the test.
- Once all tests run successfully, move on to implementing the next chunk of functionality.

Figure 9.8 Test-driven development

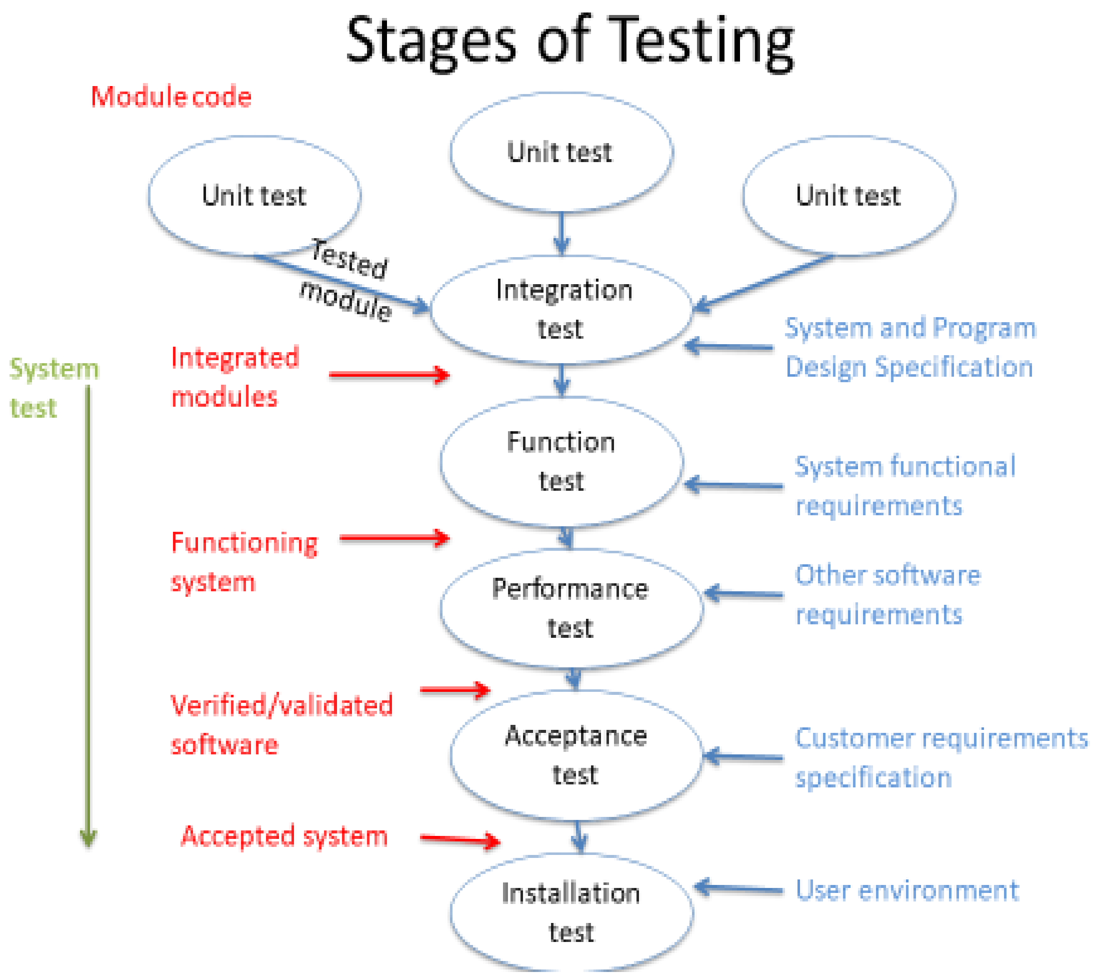


Start
Identify new functionality
Identify partial implementation of functionality
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing test to pass
Test failure
Run all automated tests
All tests pass
Refactor code if required
Functionality incomplete
Functionality complete

Benefits of Test-driven Development

- Code coverage
  - Every code segment will have at least one associated test so all code written has at least one test.
- Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

Stages of Testing



Function Test

- Test the individual functions as defined in the specifications
- Need to instrument the code so it can be tested
- Must have a test oracle
- May refer to Use Cases
- Type of black-box testing

Function Testing (~Thread)

- Thread
  - Set of module actions associated with a function
- Developed from Requirements Document
- Compares actual performance with requirements
- Can be useful for OO systems
- Effective function testing
  - High probability of detecting errors
  - Performed by test team independent of designers and programmers
  - Knowledge of expected actions
  - Test both valid and invalid inputs
  - No modifications of the system being tested just to make testing easier
  - Know when to stop testing

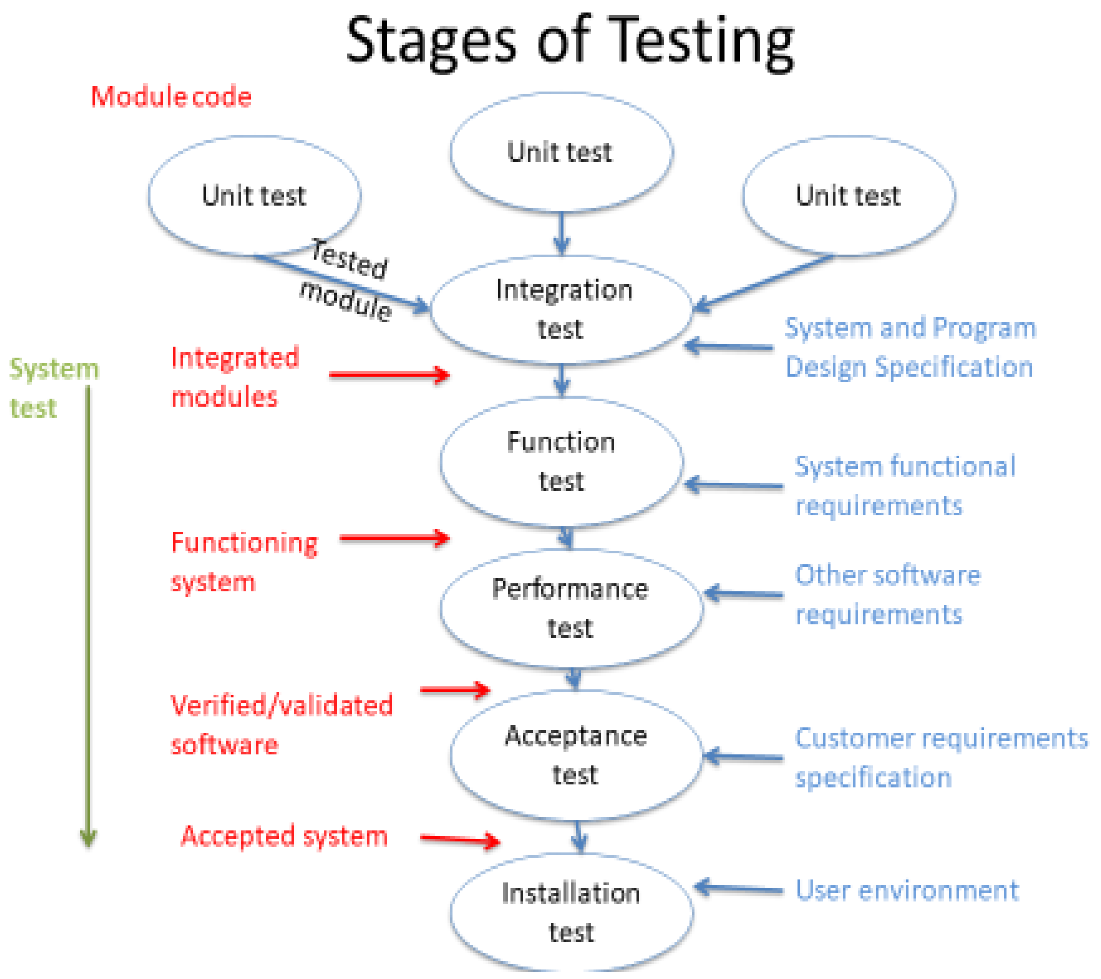
Requirements-based Testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- Mentcare system requirements:
  – If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
  – If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

Requirements Tests

- Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
- Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
- Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
- Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

Stages of Testing



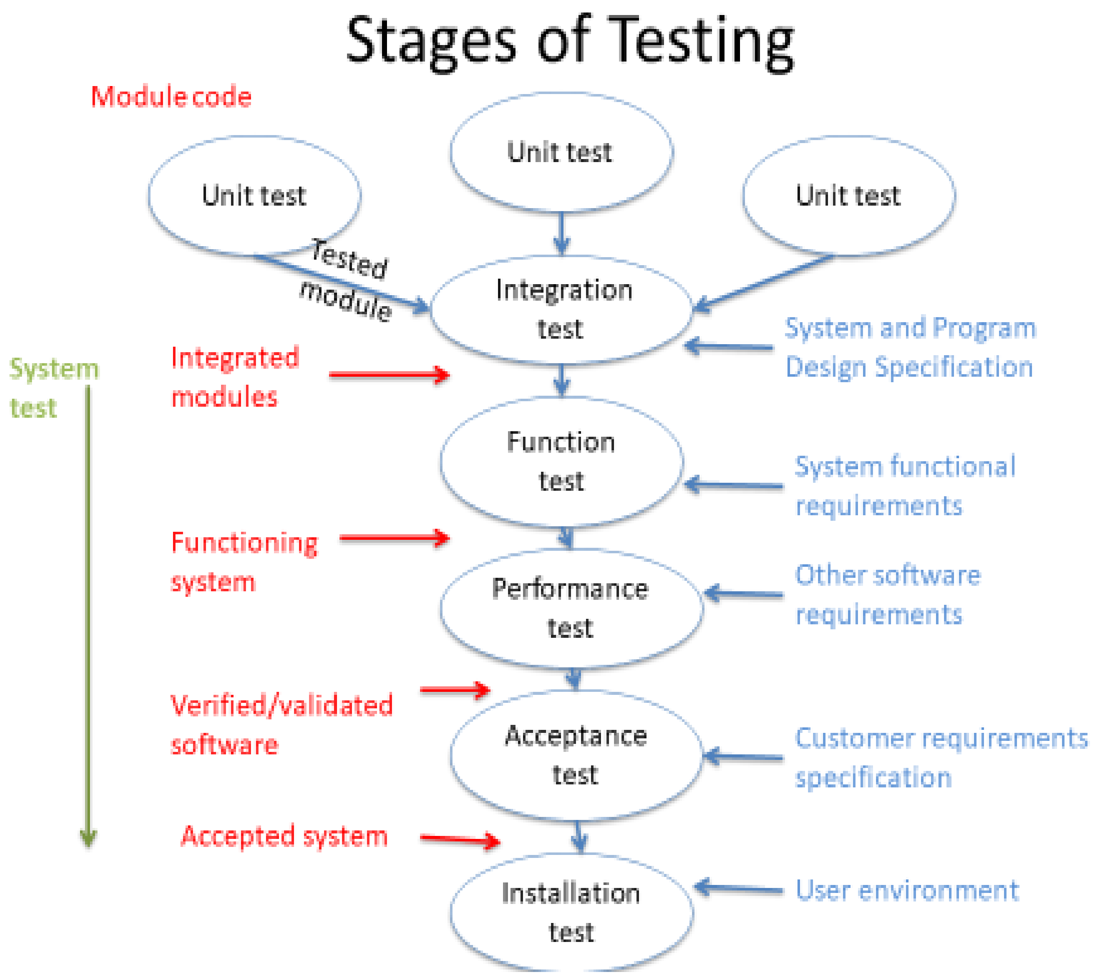
Performance Testing

- System performance measured against objectives set by customer
- Types of performance tests
  - Stress (Load) test
  - Volume test
  - Configuration test
  - Compatibility test – interface with other systems
  - Security test
  - Timing test

Performance Tests

- Environmental test
- Quality test
  - Reliability, maintainability,.....
- Recovery test
- Maintenance test
- Documentation test
- Human Factors test

Stages of Testing



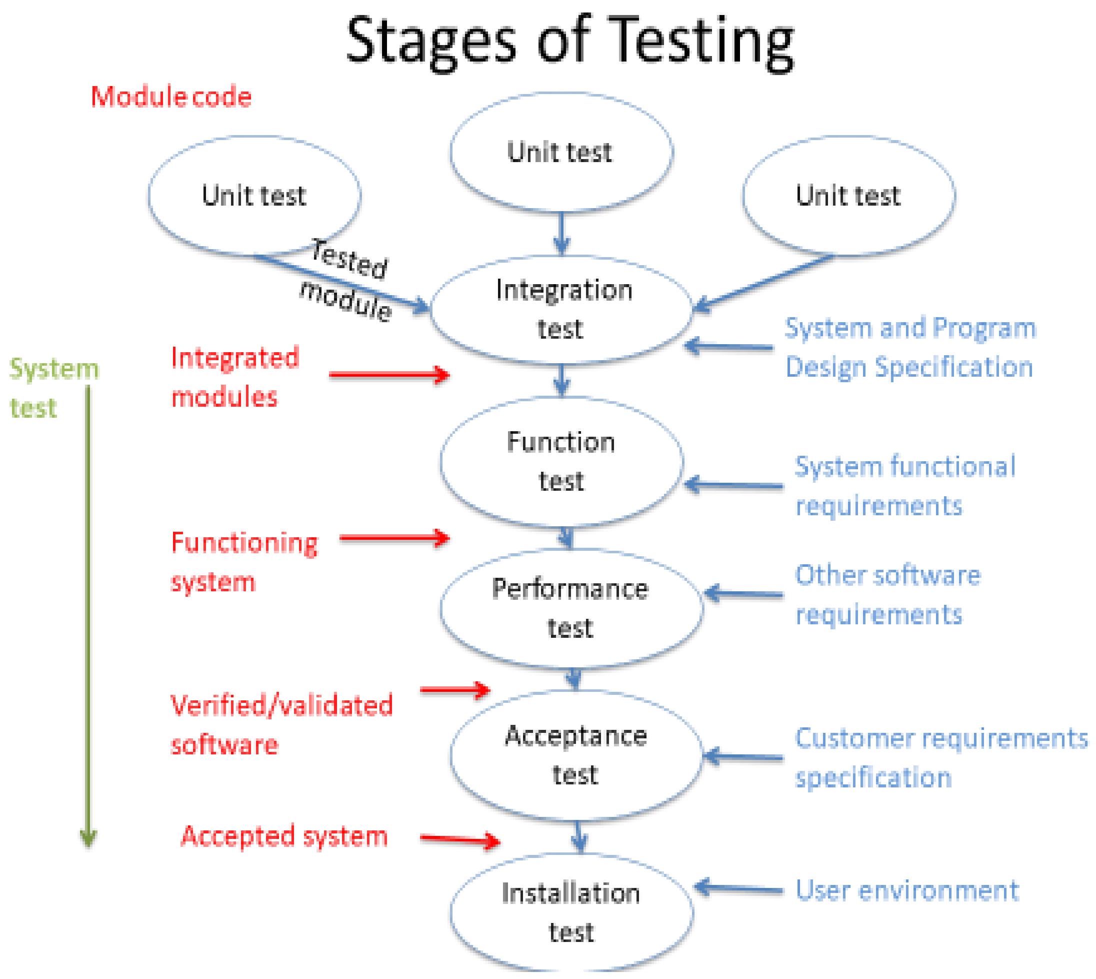
Acceptance Testing

- Customer leads testing and defines cases to be tested
- Three methods:
  - Benchmark test
    - Customer prepares set of test cases representing typical conditions
    - Performed with actual users
  - Pilot test (beta test)
    - System installed on an experimental basis
  - Parallel test
    - New system operates in parallel with previous version

Agile methods and Acceptance Testing

- User/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem is whether or not the embedded user is ‘typical’ and can represent the interests of all system stakeholders.

Stages of Testing

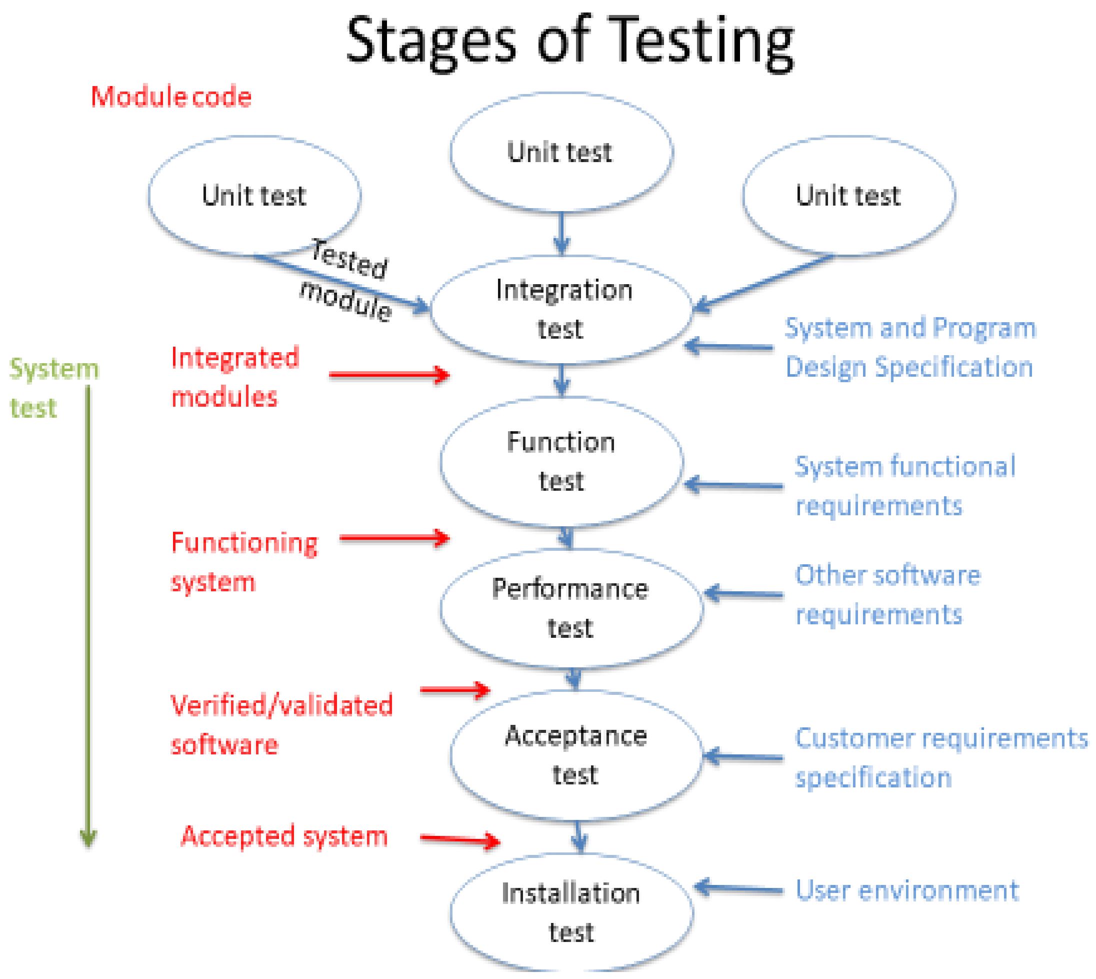


Installation Testing

- If acceptance testing done on site
  - acceptance = installation

  else
  - additional on site testing to
    - ensure completeness of system
    - verification of any functional or nonfunctional characteristics affected by
      - site conditions

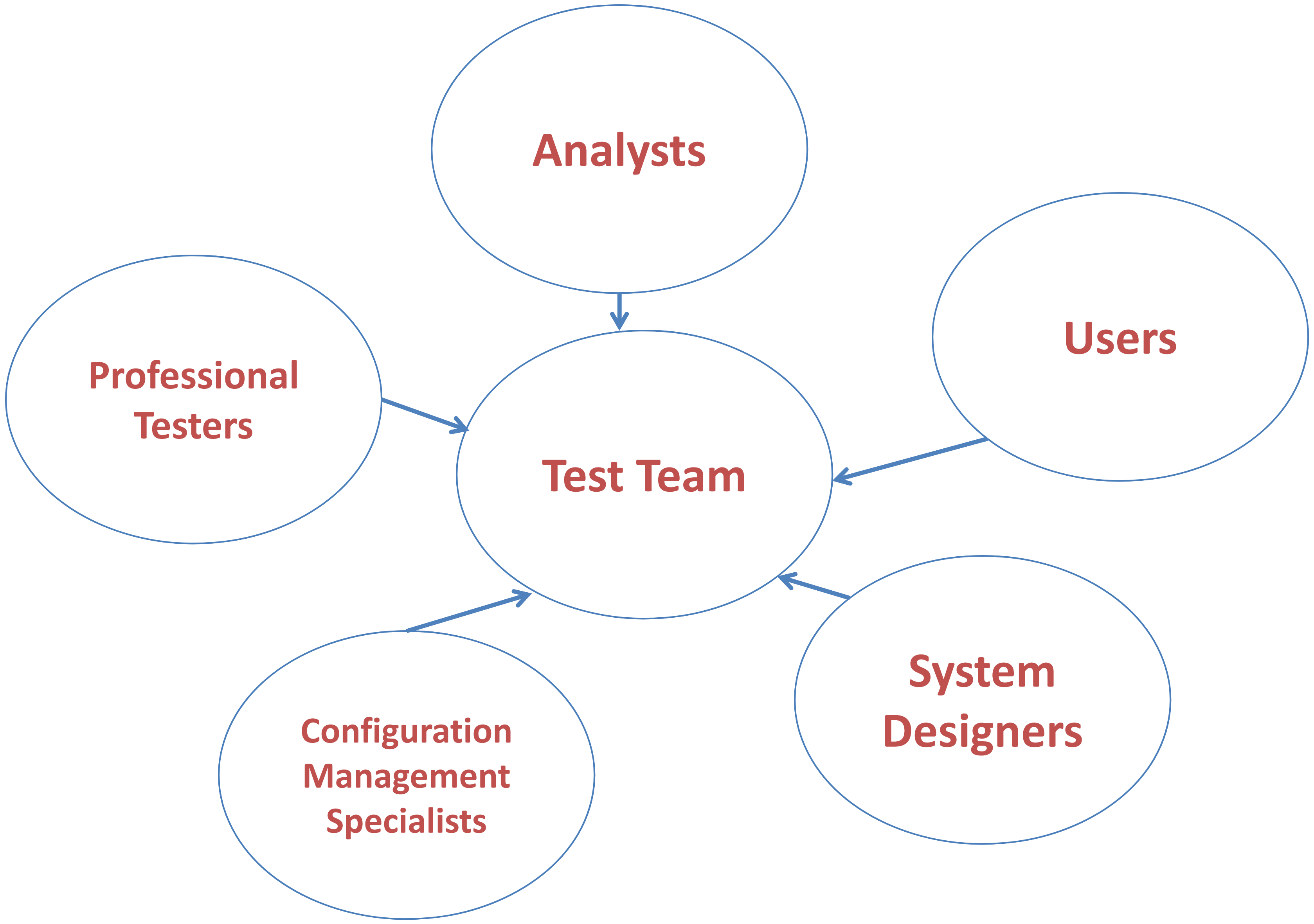
Stages of Testing



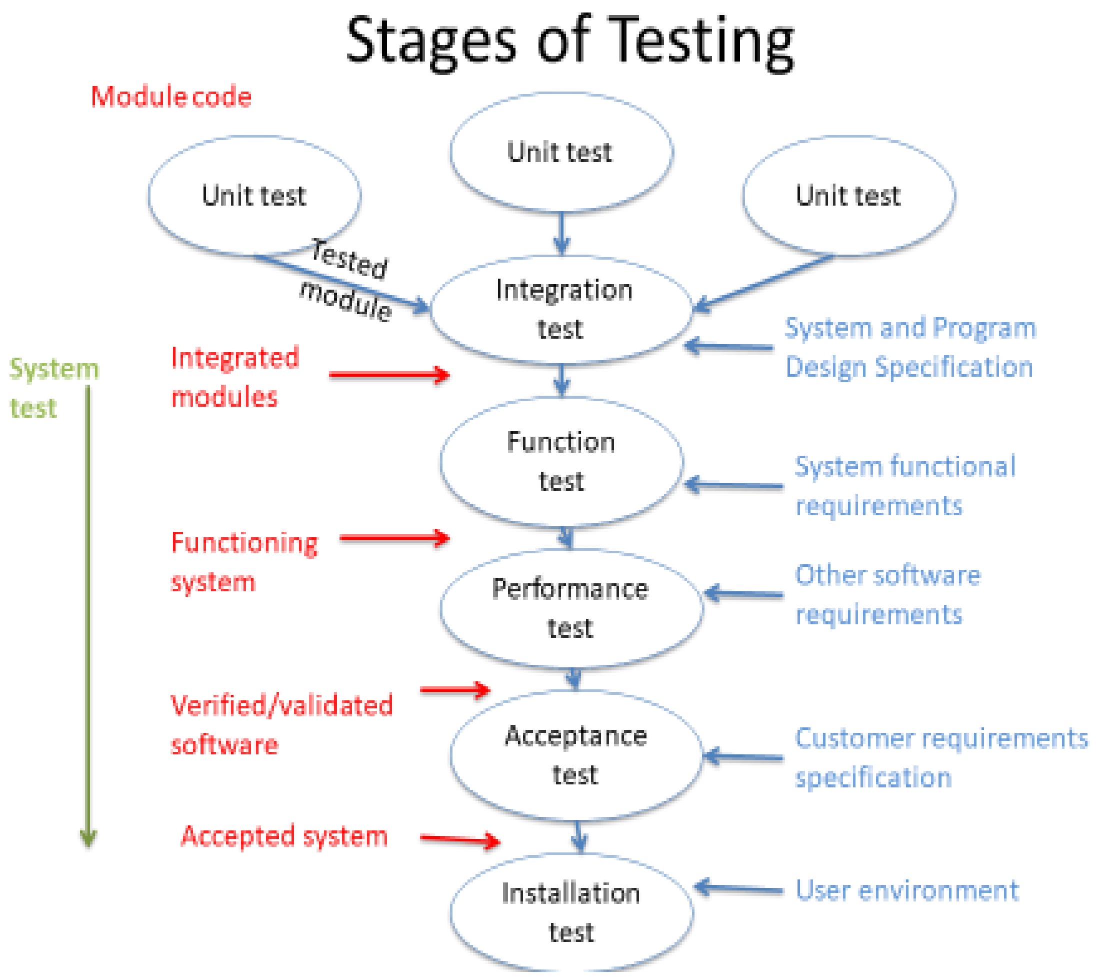
Regression Testing

- Ensure that changes have not introduced unexpected behavior
- Re-execution of a suite of tests that have already been conducted
  - Representation sample of tests that exercises all software functions
  - Additional tests that focus on areas that are likely to be affected
  - Tests that focus on the components that have been changed.

Test Team Composition



Stages of Testing



Verification vs Validation

- 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.