2008 Fall

Software Modeling & Analysis

Part 5. Verification and Validation

- Verification and Validation
- Software Testing

Lecturer: JUNBEOM YOO
jbyoo@konkuk.ac.kr

※ This lecture note is based on materials from Ian Sommerville 2006.

Chapter 22.

Verification and Validation

Objectives

- To introduce software verification and validation
- To discuss the distinction between them
- To describe program inspection process and its role in V & V
- To explain static analysis as a verification technique
- To describe the Cleanroom software development process

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.

V & V Process

- V&V is a whole life-cycle process : It must be applied at each stage in the software process.
- Two principal objectives
  - Discovery of defects in a system
  - Assessment of whether or not the system is useful and useable in an operational situation

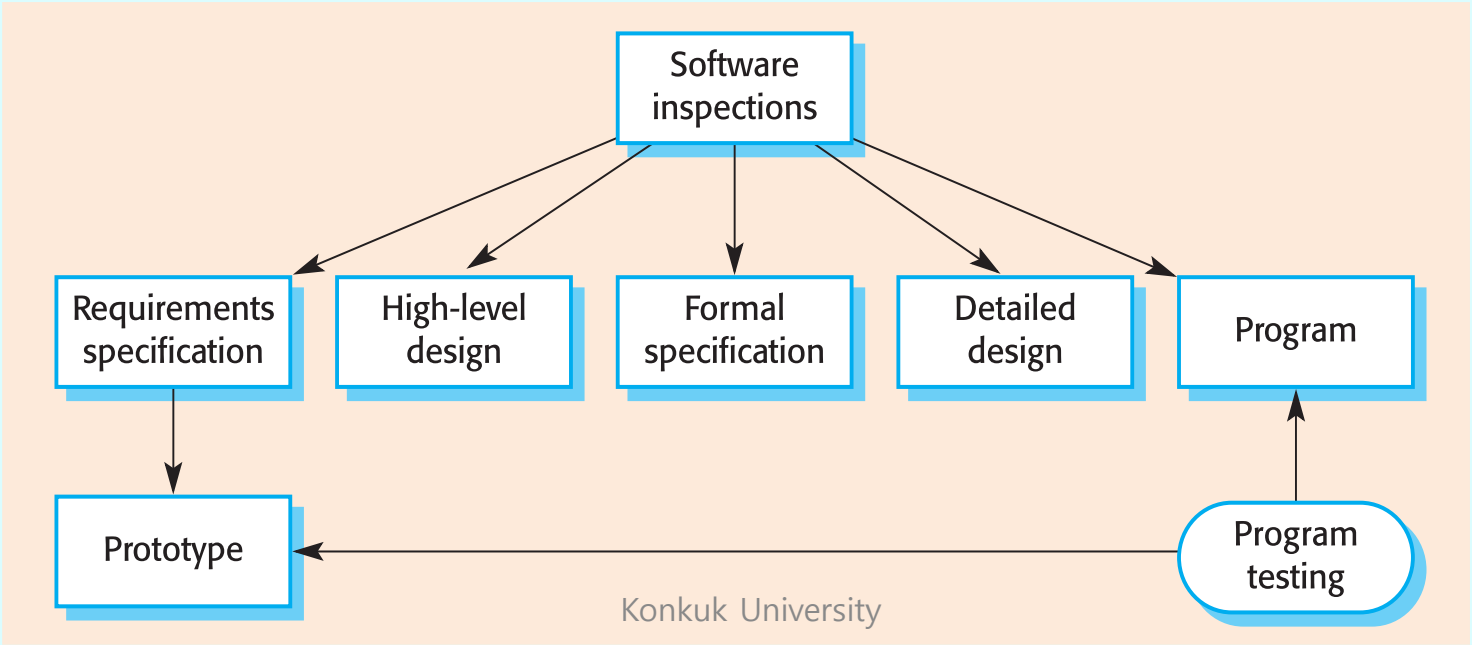
- Goals of V&V
  - V&V should establish confidence that the software is suitable for purpose.
  - This does NOT mean completely free of defects.
  - Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

V & V Confidence

- Depends on system's purpose, user expectations, and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organisation.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

Static and Dynamic Verification

- Software Inspections :
  - Analyze static system representation to discover problems (Static Verification)
  - May be supplemented by tool-based document and code analysis
- Software Testing :
  - Exercising and observing product behaviour (Dynamic Verification)
  - System is executed with test data and its operational behaviour is observed.



Program Testing

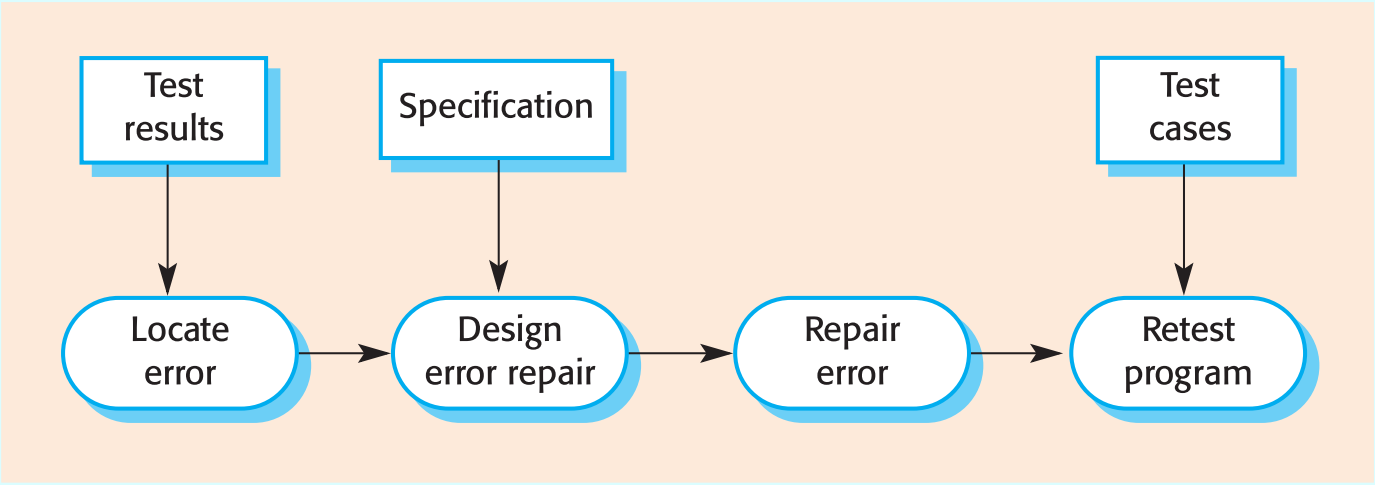
- Can reveal the presence of errors, NOT their absence.
- Can validate non-functional requirements because software is executed to see how it behaves.
- Should be used in conjunction with static verification to provide full V&V coverage.

- Types of testing
  - Defect testing
    - Tests designed to discover system defects
    - A successful defect test is one which reveals the presence of defects in a system.
    - Covered in Chapter 23 (Soon)
  - Validation testing
    - Intended to show that the software meets its requirements.
    - A successful test is one that shows that a requirements has been properly implemented.

Testing and Debugging

- Defect testing and debugging are distinct processes.
  - Testing is concerned with establishing the existence of defects in a program.
  - Debugging is concerned with locating and repairing these errors.

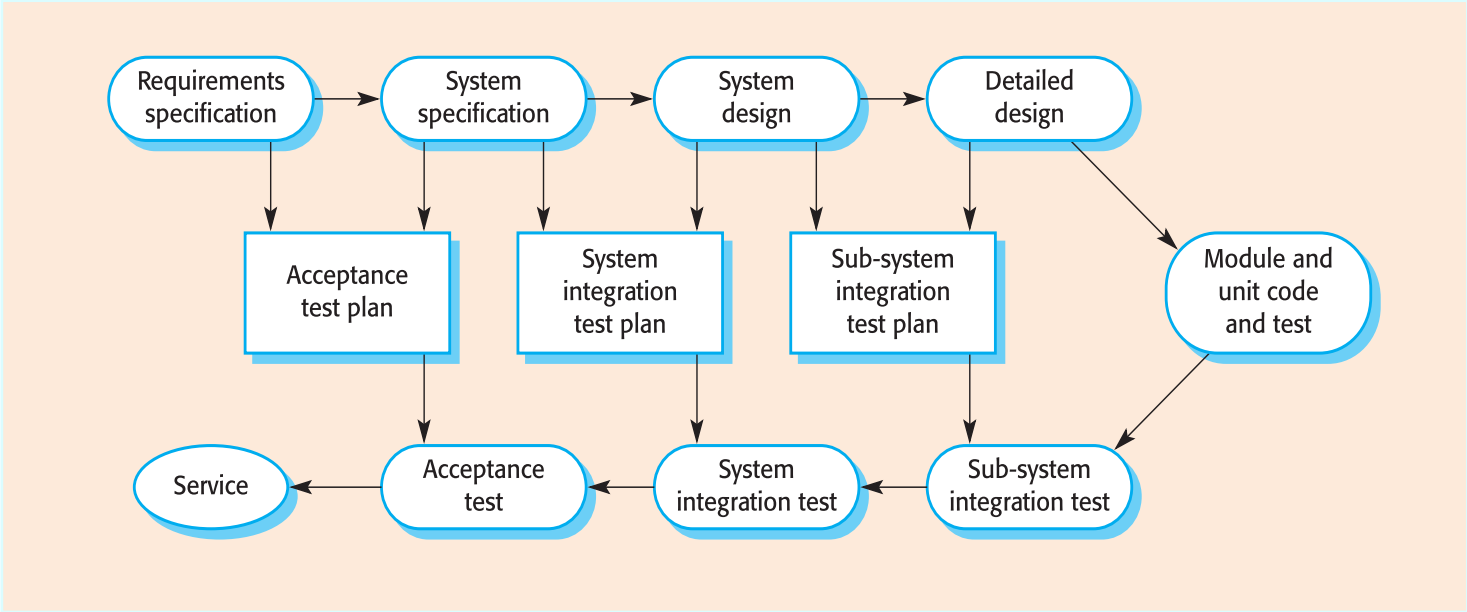
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.
- Debugging process:



V & V Planning

- V&V Planning should start early in the development process.
  - The plan should identify the balance between static verification and testing.
  - Test planning is about defining standards for testing process rather than describing product tests.

- V-Model for Software Testing



Software Test Plan

The testing process
A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability
Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items
The products of the software process that are to be tested should be specified.

Testing schedule
An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

Test recording procedures
It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it been carried out correctly.

Hardware and software requirements
This section should set out software tools required and estimated hardware utilisation.

Constraints
Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Software Inspection

- Involves people examining the source representation with aim of discovering anomalies and defects.
- Does not require execution of system, so may be used before implementation.
- May be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- Effective technique for discovering program errors

- Advantages:
  - Many different defects may be discovered in a single inspection.
  - In testing, one defect may mask another so several executions are required.
  - Using domain and programming knowledge, reviewers are likely to have seen the types of error that commonly arise.

Inspection and Testing

- Inspection and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.

- Inspections
  - Can check conformance with a specification but not conformance with the customer's real requirements.
  - Cannot check non-functional characteristics such as performance, usability, etc.

Program Inspection

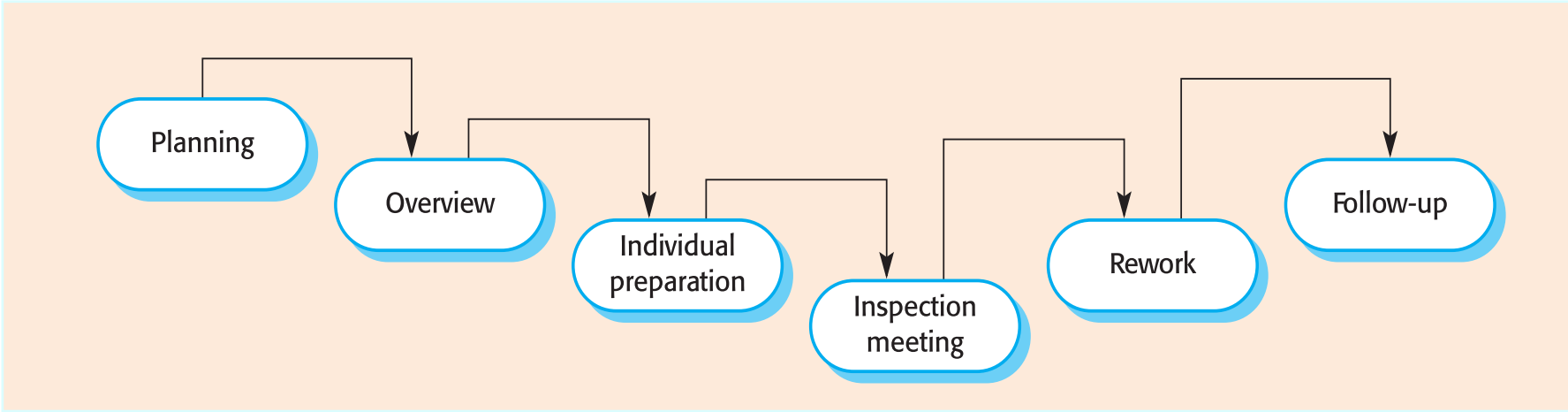
- Formalized approach to document reviews
- Intended explicitly for detecting defects (not correction).
- Defects may be
  - Logical errors
  - Anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable)
  - Non-compliance with standards

Pre-Conditions for Inspection

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal i.e. finding out who makes mistakes.

Inspection Procedure

- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair discovered errors.
- Re-inspection may or may not be required.



Inspection Roles

Author or owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting.
Scribe	Records the results of the inspection meeting.
Chairman or moderator	Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
Chief moderator	Responsible for inspection process improvements, checklist updating, standards development etc.

Inspection Checklist

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.

- Examples of common errors in checklists :
  - Data faults
  - Control faults
  - Input/Output faults
  - Interface faults
  - Storage management faults
  - Exception management faults

Automated Static Analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Stages of Static Analysis

- Control flow analysis
  - Checks for loops with multiple exit or entry points, finds unreachable code…
- Data use analysis
  - Detects uninitialized variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- Interface analysis
  - Checks the consistency of routine and procedure declarations and their use
- Information flow analysis
  - Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- Path analysis
  - Identifies paths through the program and sets out the statements executed in that path. It is potentially useful in the review process.
- Both these stages generate vast amounts of information. They must be used with care.

Use of Static Analysis

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler.
- Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation.

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
 int Anarray;
{   printf("%d?Anarray);  }

main ()
{
 int Anarray[5]; int i; char c;
 printarray (Anarray, i, c);
 printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Example of static analysis

Verification through Formal Methods

- Formal methods can be used when mathematical specification of system is prepared.
- Formal methods are the ultimate static verification technique.
- They involve detailed mathematical analysis of the specification and may develop formal arguments that a program conforms to its mathematical specification.

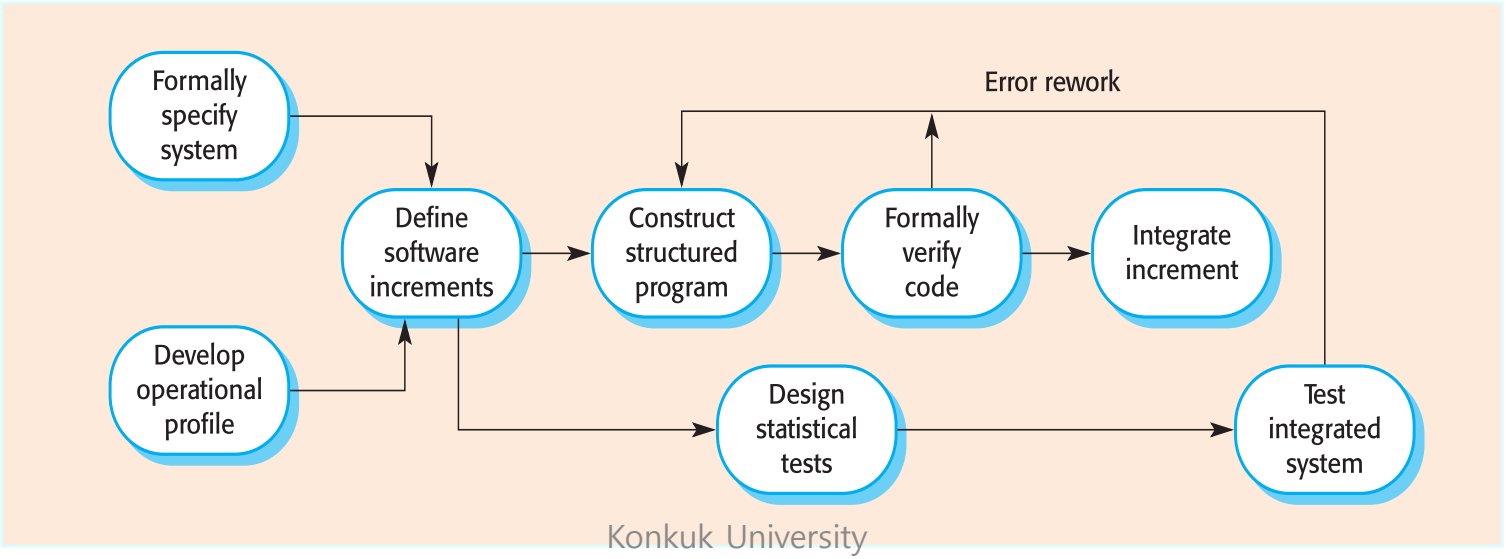
Arguments about Formal Methods

- Advantages:
  - Produce mathematical specifications which require detailed analysis of the requirements and this is likely to uncover errors.
  - Detect implementation errors before testing, when the program is analyzed alongside the specification.

- Disadvantages:
  - Require specialized notations that cannot be understood by domain experts.
  - Very expensive to develop specification sand even more expensive to show that the program meets that specification.
  - May be possible to reach the same level of confidence more cheaply using other V & V techniques.

Cleanroom Software Development

- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- Based on
  - Incremental development - Formal specification
  - Static verification using correctness arguments
  - Statistical testing to determine program reliability



Characteristics of Cleanroom Process

- Formal specification using a state transition model
- Incremental development where the customer prioritises increments
- Structured programming - limited control and abstraction constructs are used in the program.
- Static verification using rigorous inspections
- Statistical testing of the system

- Team organization:
  - Specification team: Responsible for developing and maintaining the system specification.
  - Development team: Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
  - Certification team: Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

Evaluation of Cleamroon Process

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- There were fewer errors than in a 'traditional' development process.

- However, the process is not widely used. It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers.

Summary

- Verification and validation are not the same thing. Verification shows conformance with specification; validation shows that the program meets the customer's needs.

- Test plans should be drawn up to guide the testing process.
- Static verification techniques involve examination and analysis of the program for error detection.
- Program inspections are very effective in discovering errors.
- Program code in inspections is systematically checked by a small team to locate software faults.
- Static analysis tools can discover program anomalies which may be an indication of faults in the code.
- Cleanroom development process depends on incremental development, static verification and statistical testing.

Chapter 23.

Software Testing

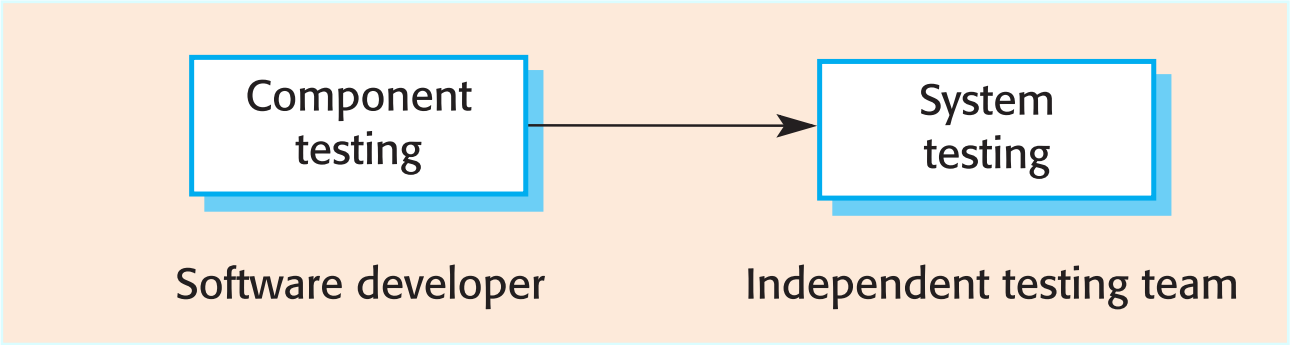
Objectives

- To discuss distinctions between validation testing and defect testing
- To describe principles of system and component testing
- To describe strategies for generating system test cases
- To understand essential characteristics of tool used for test automation

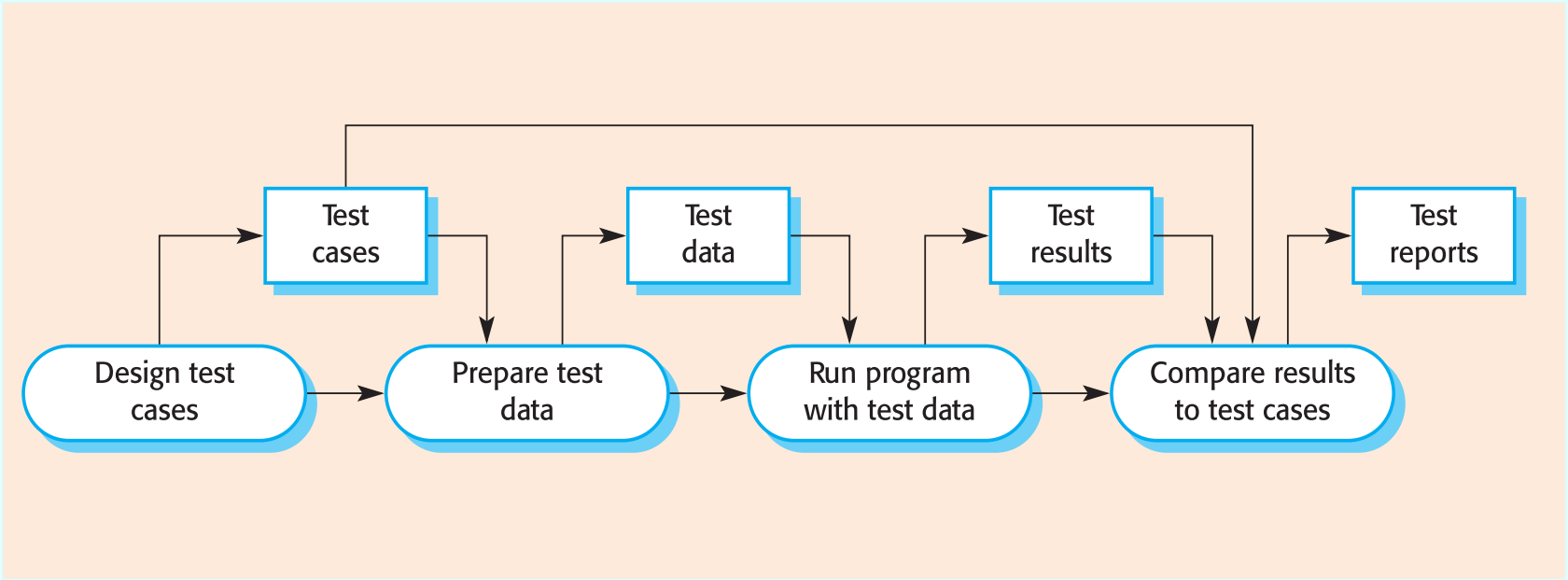
Software Testing

- Component testing
  - Testing of individual program components
  - Usually responsibility of component
  - Tests are derived from the developer's experience.

- System testing
  - Testing of groups of components integrated to create a system or sub-system
  - Responsibility of independent testing team
  - Tests are based on system specification.



Software Testing Process



Goals of Software Testing

- Validation testing
  - To demonstrate to developer and system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.

- Defect testing
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

System Testing

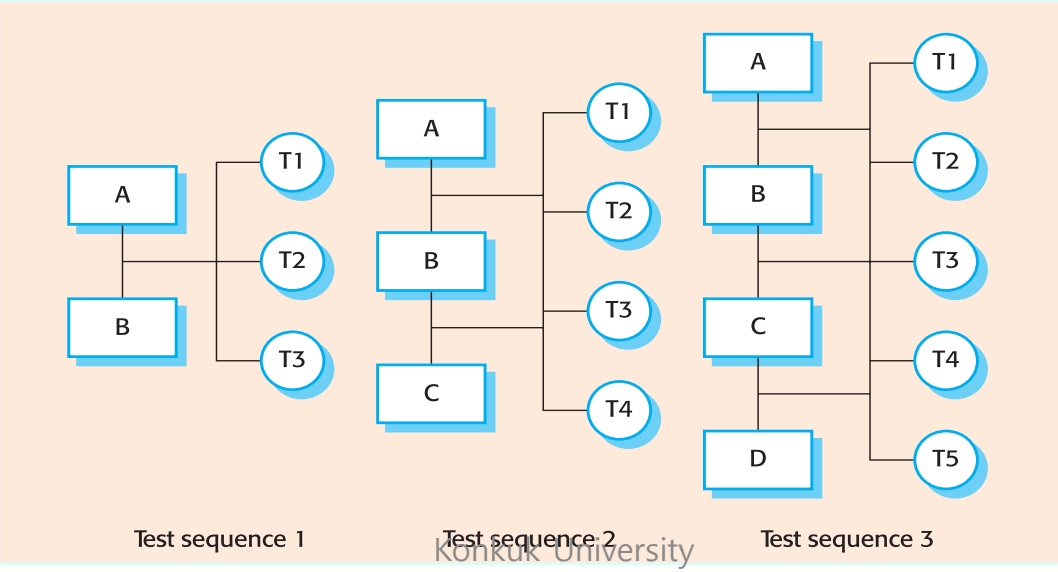
- Involves integrating components to create a system or sub-system
- May involve testing an increment to be delivered to the customer

- Two phases:
  - Integration testing
    - Test team has access to system source code.
    - System is tested as components are integrated.
  - Release testing
    - Test team tests a complete system to be delivered as a black-box.

Integration Testing

- Involves building a system from its components and testing it for problems that arise from component interactions.
  - Top-down integration
    - Develop the skeleton of the system and populate it with components
  - Bottom-up integration
    - Integrate infrastructure components then add functional components

- Incremental integration testing



Integration Testing Approaches

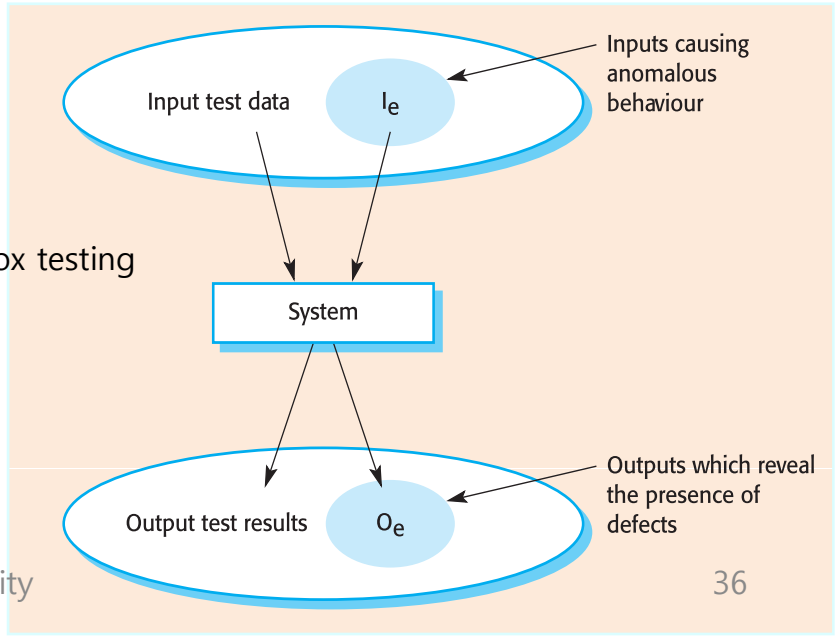
- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture.
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development.
- Test implementation
  - Often easier with bottom-up integration testing.
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests.

Release Testing

- Process of testing a system release that will be distributed to customers.
- Primary goal is to increase the supplier's confidence that the system mee ts its requirements.

- Release testing is usually black-box or functional testing
  - Based on the system specification only
  - Testers do not have knowledge of the system implementation.

- Release testing may include
  - Performance testing
  - Stress testing



Performance Testing

- Release testing may involve testing emergent properties of system, such as performance and reliability.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress Testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system to test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data too.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as network becomes overloaded.

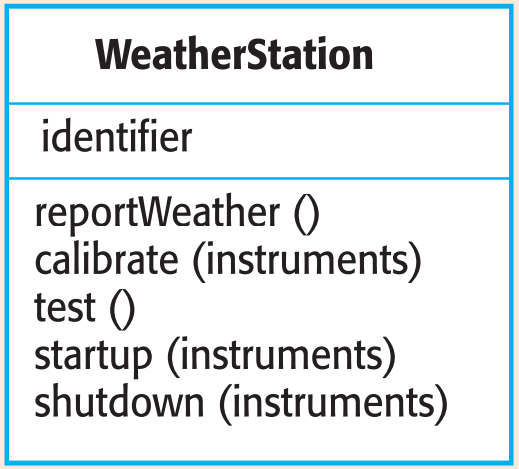
Component Testing

- Component testing is the process of testing individual components in isolation.
- Defect testing process

- Components 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

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising object in all possible states
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.



Need to define test cases for all methods
- reportWeather, calibrate,
- test, startup and shutdown

Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions

For example:
Waiting -> Calibrating -> Testing -> Transmitting -> Waiting

Interface Testing

- To detect faults due to interface errors or invalid assumptions about interfaces
- Particularly important for object-oriented development as objects are defined by their interfaces.

- Guidelines for interface testing
  - Design tests so that parameters to 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

Interface Types and Errors

- Interface types
  - Parameter interfaces
    - Data passes from one procedure to another.
  - Shared memory interfaces
    - Block of memory is shared between procedures or functions.
  - 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.

- Interface misuse : Calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding : Calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors : The called and the calling component operate at different speeds and out-of-date information is accessed.

Test Case Design

- Involves designing the test cases (inputs and outputs) used to test the system.
- Goal of test case design is to create a set of tests that are effective in validation and defect testing.

- Test case design approaches:
  - Requirements-based testing
  - Partition testing
  - Structural testing

Requirements based Testing

- A general principle of requirements engineering is that requirements should be testable.
- Requirements-based testing is a validation testing technique where you consider each requirement and derive a set of tests for that requirement.

Example : LIBSYS Requirements based Testing

LIBSYS requirements

The user shall be able to search either all of the initial set of databases or select a subset from it.

The system shall provide appropriate viewers for the user to read documents in the document store.

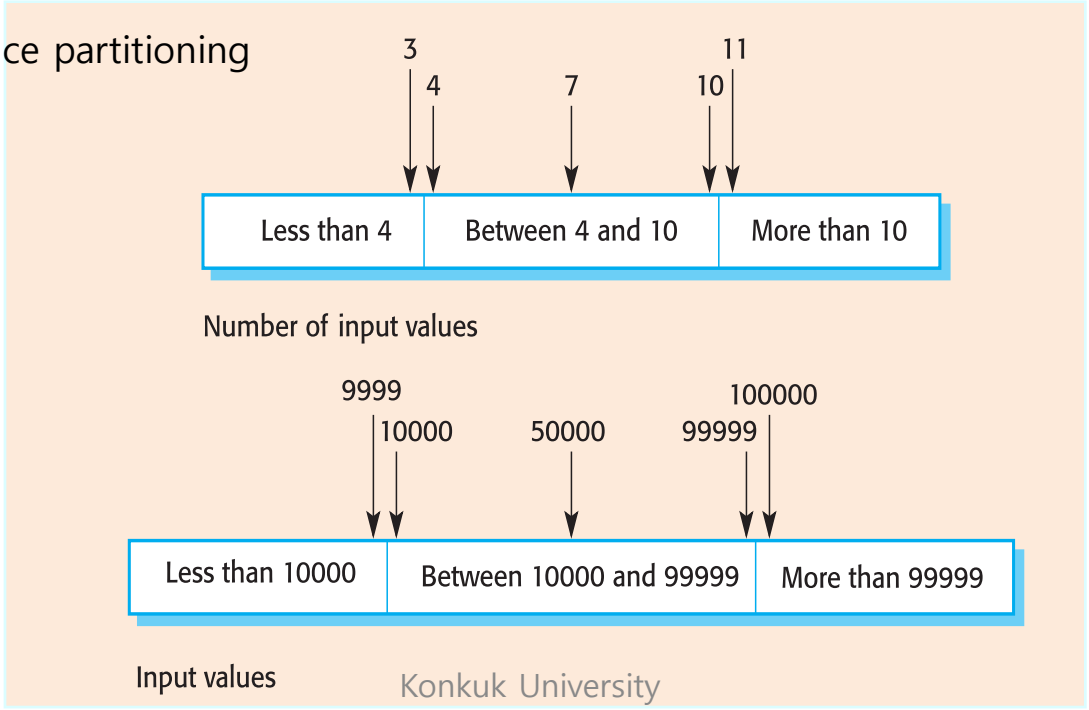
Every order shall be allocated a unique identifier (ORDER_ID) that the user shall be able to copy to the accountõ permanent storage area.

LIBSYS test cases

- Initiate user search for searches for items that are known to be present and known not to be present, where the set of databases includes 1 database.
- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes 2 databases
- Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than 2 databases.
- Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
- Select more than one database from the set of databases and initiate searches for items that are known to be present and known not to be present.

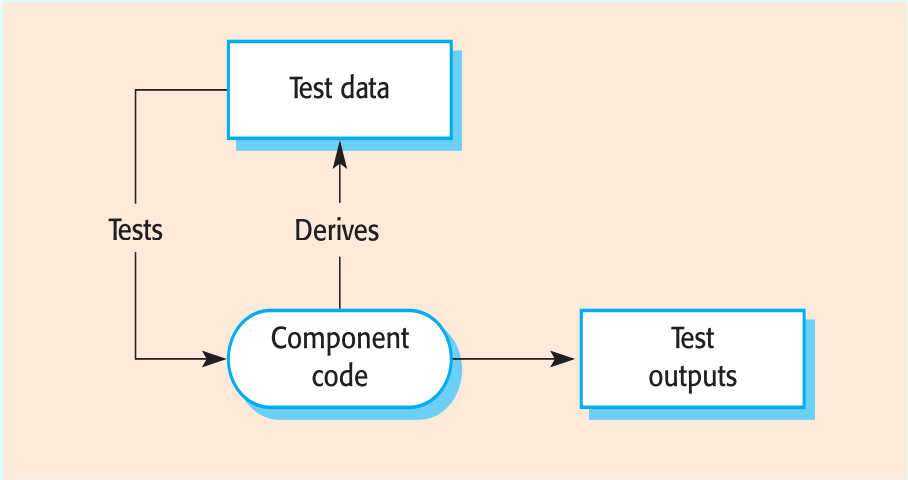
Partition Testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.



Structural Testing

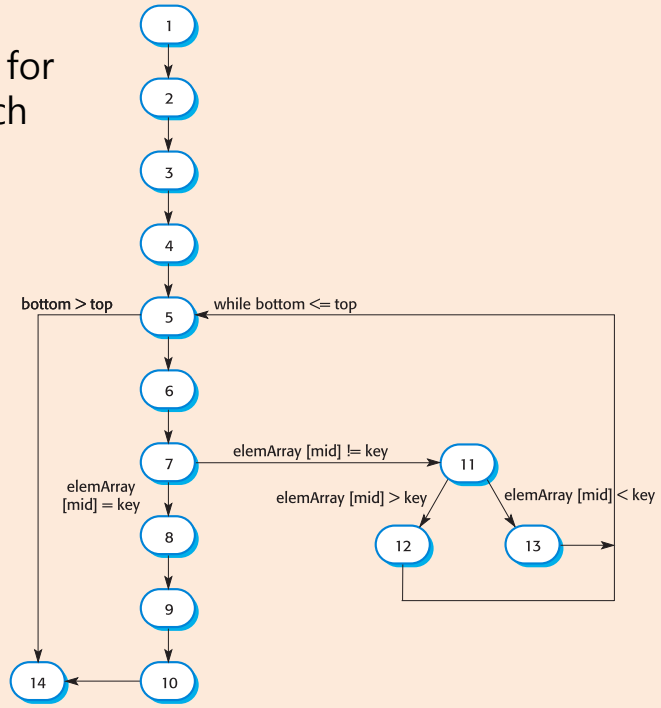
- Sometime called white-box testing.
- Derives test cases according to program structure.
- Knowledge of the program is used to identify additional test cases.
- Objective is to exercise all program statements.
  - A number of structural testing techniques exist, i.e. path testing
  - A number of testing coverage exist.



Path Testing

- To ensure that all the paths in the programs are executed.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control. Statements with conditions become nodes in the flow graph.

Flow-graph for binary search



Independent test paths:

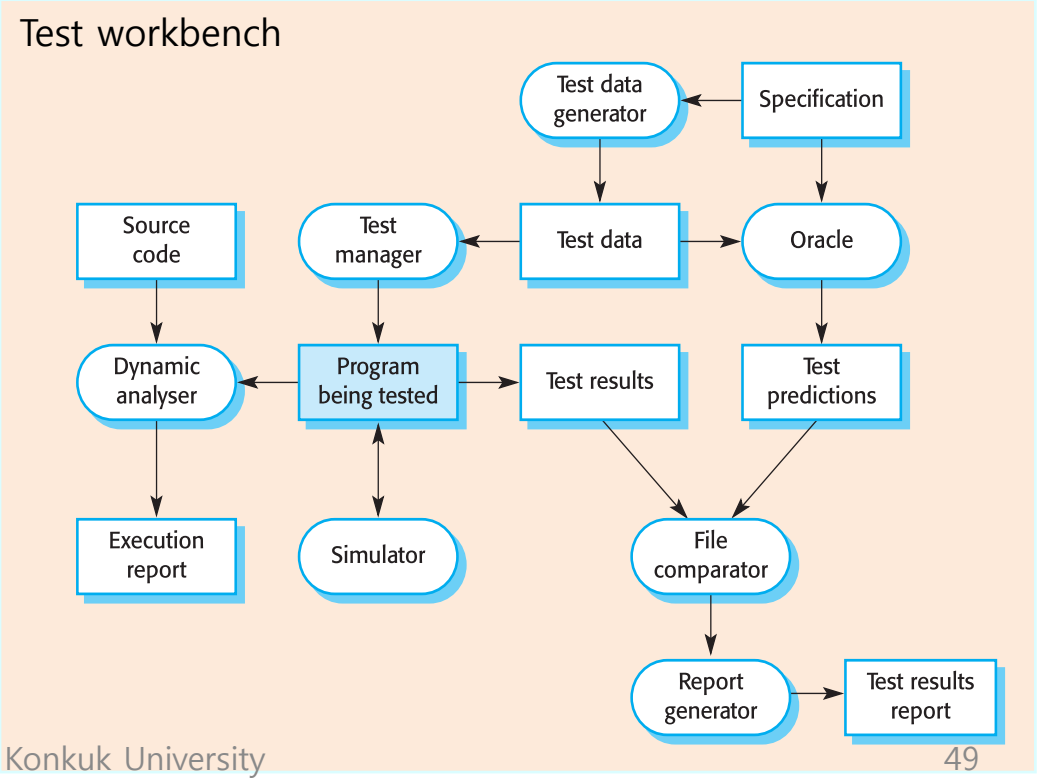
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
1, 2, 3, 4, 5, 14
1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …

Test cases should be derived so that all of these paths are executed

A dynamic program analyser may be used to check that paths have been executed

Test Automation

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Most testing workbenches are open systems, because testing needs are organisation-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.
- Scripts may be developed for user interface simulators and patterns for test data generators.
- Test workbench adaptation



Summary

- Testing can show the presence of faults in a system, but it cannot prove there are no remaining faults.
- Component developers are responsible for component testing. System testing is the responsibility of a separate team.
- Integration testing is testing increments of the system. Release testing involves testing a system to be released to a customer.
- Interface testing is designed to discover defects in the interfaces of composite components.
- Equivalence partitioning is a way of discovering test cases - all cases in a partition should behave in the same way.
- Structural analysis relies on analysing a program and deriving tests from this analysis.
- Test automation reduces testing costs by supporting the test process with a range of software tools.