ICT 2402 Software Engineering

Verification and Validation

Topics covered

- Verification and validation planning
- Software inspections
- Automated static analysis
- System testing
- Component testing
- Test case design
- Test automation
- Software quality assurance

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

The V & V 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 useful and useable in an operational situation

V& V goals

- Verification and validation should establish confidence that the software is fit for purpose
- This does NOT mean completely free of defects
- Must be good enough for its intended use and the type of use will determine the degree of confidence that is needed

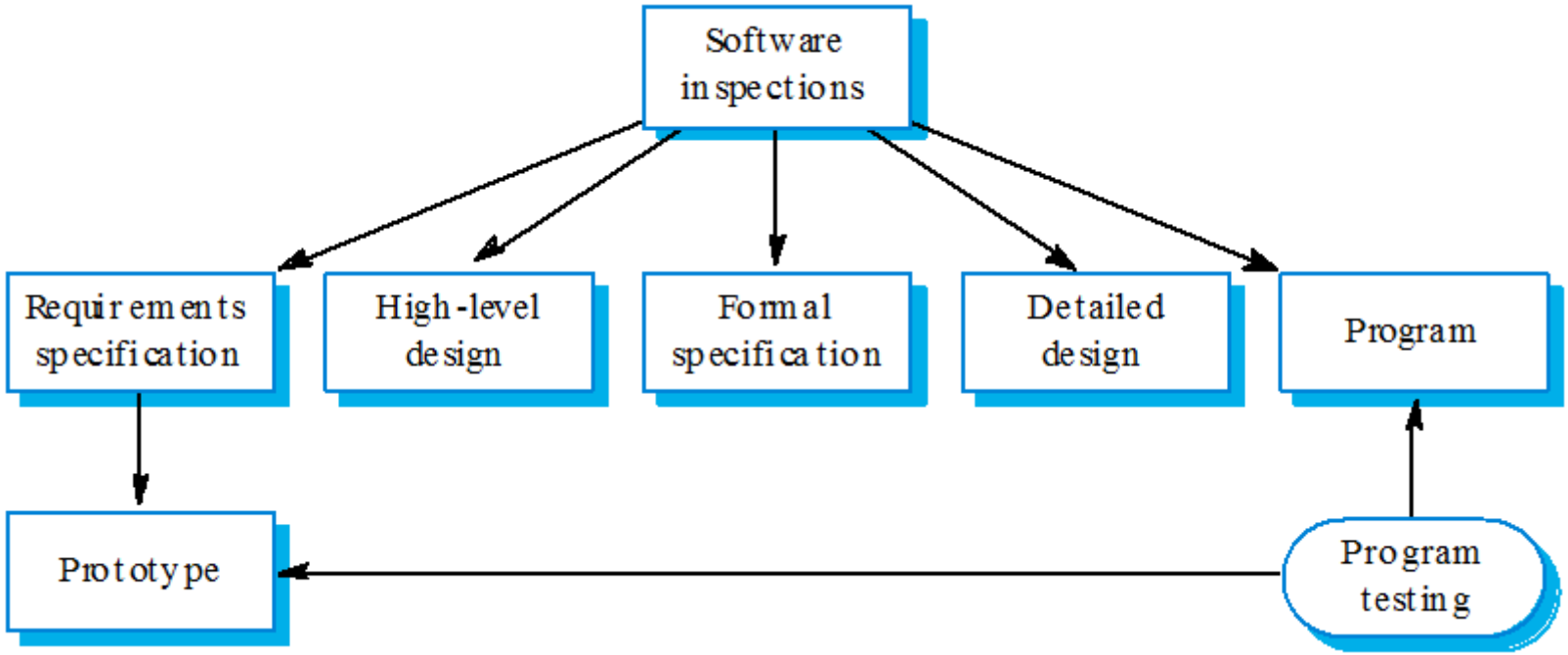
V & V confidence

- Depends on system's purpose, user expectations and marketing environment
  - Software function
    - 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

- Static verification
  - Software inspections. Concerned with analysis of the static system representation to discover problems
  - May be supplement by tool-based document and code analysis
- Dynamic verification
  - Software testing. Concerned with exercising and observing product behaviour
  - The system is executed with test data and its operational behaviour is observed

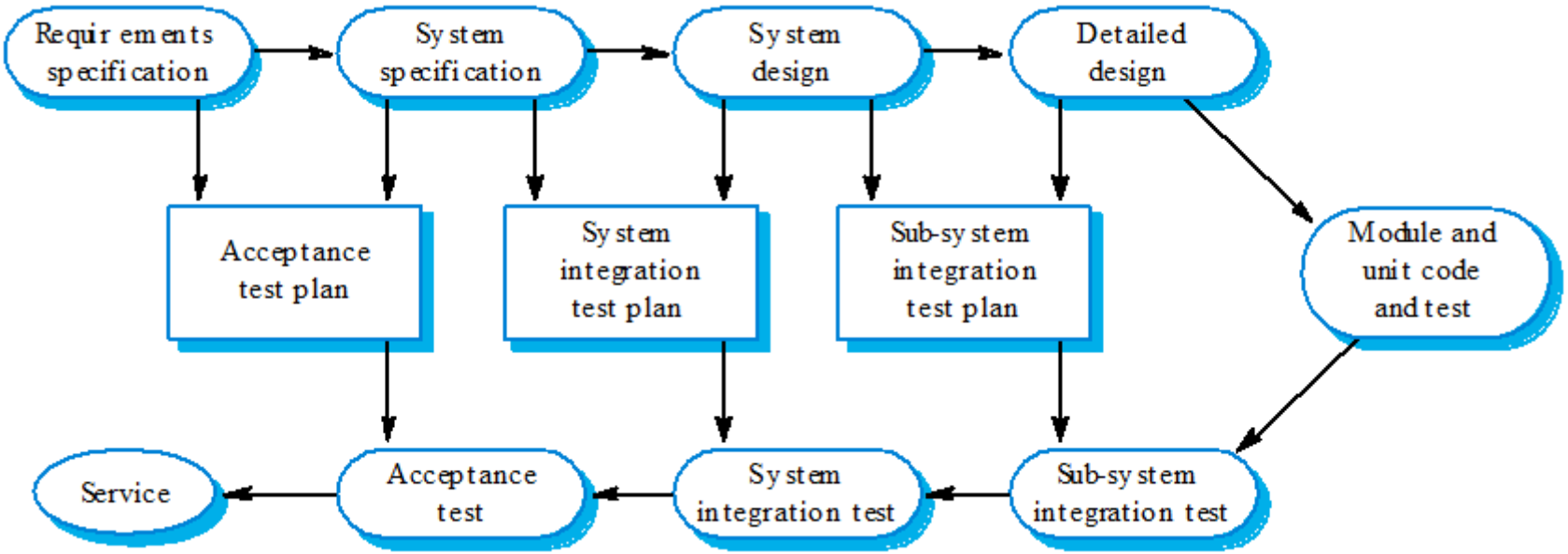
Static and Dynamic V&V



V & V planning

- Careful planning is required to get the most out of testing and inspection processes
- 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 the testing process rather than describing product tests

The V-model of development



Software inspections

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

Inspection success

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

Inspections and testing

- Inspections 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
- Inspections cannot check non-functional characteristics such as performance, usability, etc

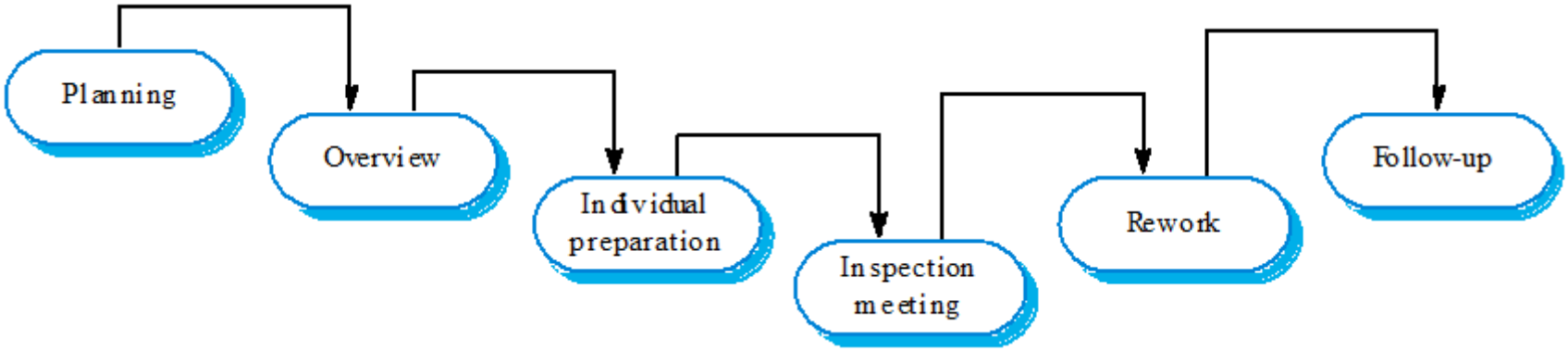
Program inspections

- Formalised approach to document reviews
- Intended explicitly for defect detection (not correction)
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialised variable) or non-compliance with standards

Inspection pre-conditions

- 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)

The inspection process



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 checklists

- 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: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection checks 1

- Data faults
  - Are all program variables initialized before their values are used?
  - Have all constants been named?
  - Should the upper bound of arrays be equal to the size of the array or Size -1?
  - If character strings are used, is a delimiter explicitly assigned?
  - Is there any possibility of buffer overflow?
- Control faults
  - For each conditional statement, is the condition correct?
  - Is each loop certain to terminate?
  - Are compound statements correctly bracketed?
  - In case statements, are all possible cases accounted for?
  - If a break is required after each case in case statements, has it been included?
- Input/output faults
  - Are all input variables used?
  - Are all output variables assigned a value before they are output?
  - Can unexpected inputs cause corruption?

Inspection checks 2

- Interface faults
  - Do all function and method calls have the correct number of parameters?
  - Do formal and actual parameter types match?
  - Are the parameters in the right order?
  - If components access shared memory, do they have the same model of the shared memory structure?
- Storage management faults
  - If a linked structure is modified, have all links been correctly reassigned?
  - If dynamic storage is used, has space been allocated correctly?
  - Is space explicitly de-allocated after it is no longer required?
- Exception management faults
  - Have all possible error conditions been taken into account?

Inspection rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 500 lines costs about 40 man/hours effort - about £2800 at UK rates

Automated static analysis

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

Static analysis checks

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, etc.
- 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

Stages of static analysis

- 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. Again, 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

Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced
- They 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 for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors
- They can detect implementation errors before testing when the program is analysed alongside the specification

Arguments against formal methods

- Require specialized notations that cannot be understood by domain experts
- Very expensive to develop a specification and even more expensive to show that a program meets that specification
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques

Program testing

- Can reveal the presence of errors NOT their absence
- The only validation technique for non-functional requirements as the software has to be executed to see how it behaves
- Should be used in conjunction with static verification to provide full V&V coverage

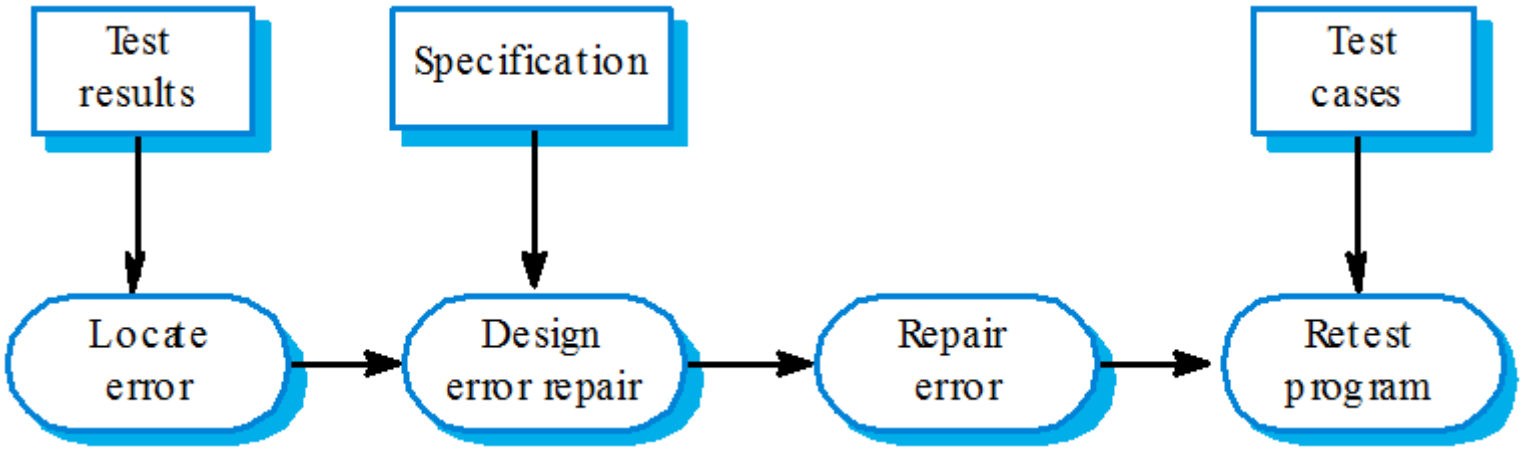
Types of testing

- Defect testing
  - Tests designed to discover system defects
  - A successful defect test is one which reveals the presence of defects in a system
- 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
- Verification and validation 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

The debugging process



The structure of a software test plan

- The testing process
- Requirements traceability
- Tested items
- Testing schedule
- Test recording procedures
- Hardware and software requirements
- Constraints

The 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.

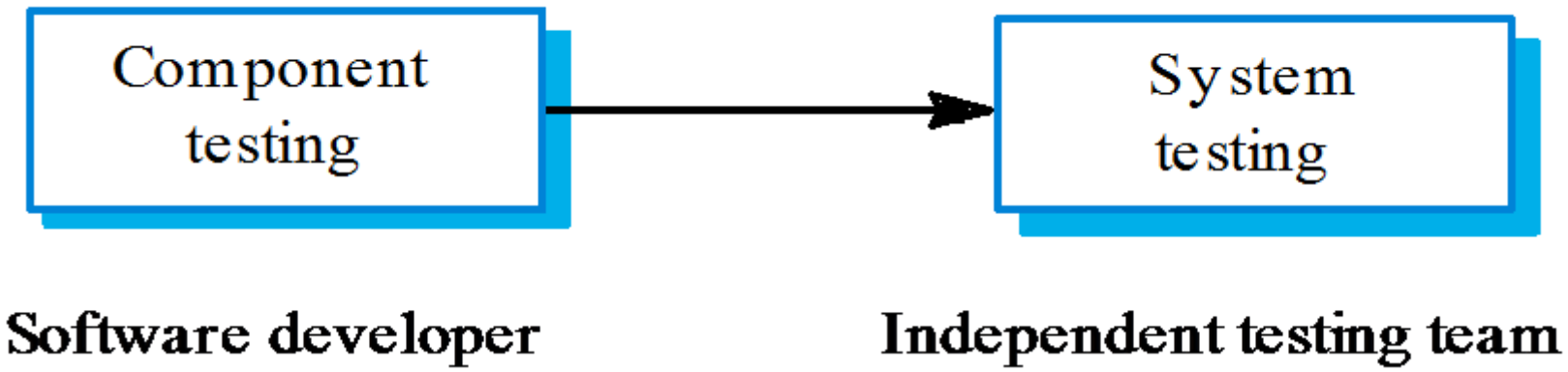
The software test plan(Cont…)

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

The Testing Process

- Component testing: Test individual program components
  - Usually the responsibility of the component developer
  - Tests are derived from the developer's experience
- System testing: Test groups of components integrated to create a system or sub-system
  - The responsibility of an independent testing team
  - Tests are based on a system specification

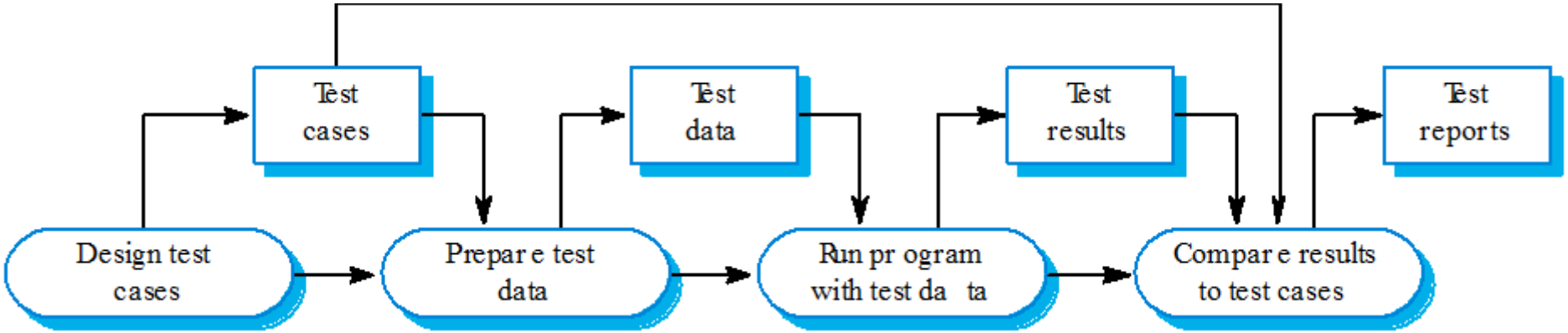
Testing Phases



Validation testing vs defect testing

- Validation testing
  - Demonstrates to the developer and the system customer that the software meets its requirements
  - There should be at least one test for every requirement in the user and systems requirements documents
- Defect testing
  - Discovers faults or defects in the software where its behavior is incorrect or not in conformance with its specification
  - A successful defect test shows the system performing incorrectly, exposing defects in the system

Software Testing Process



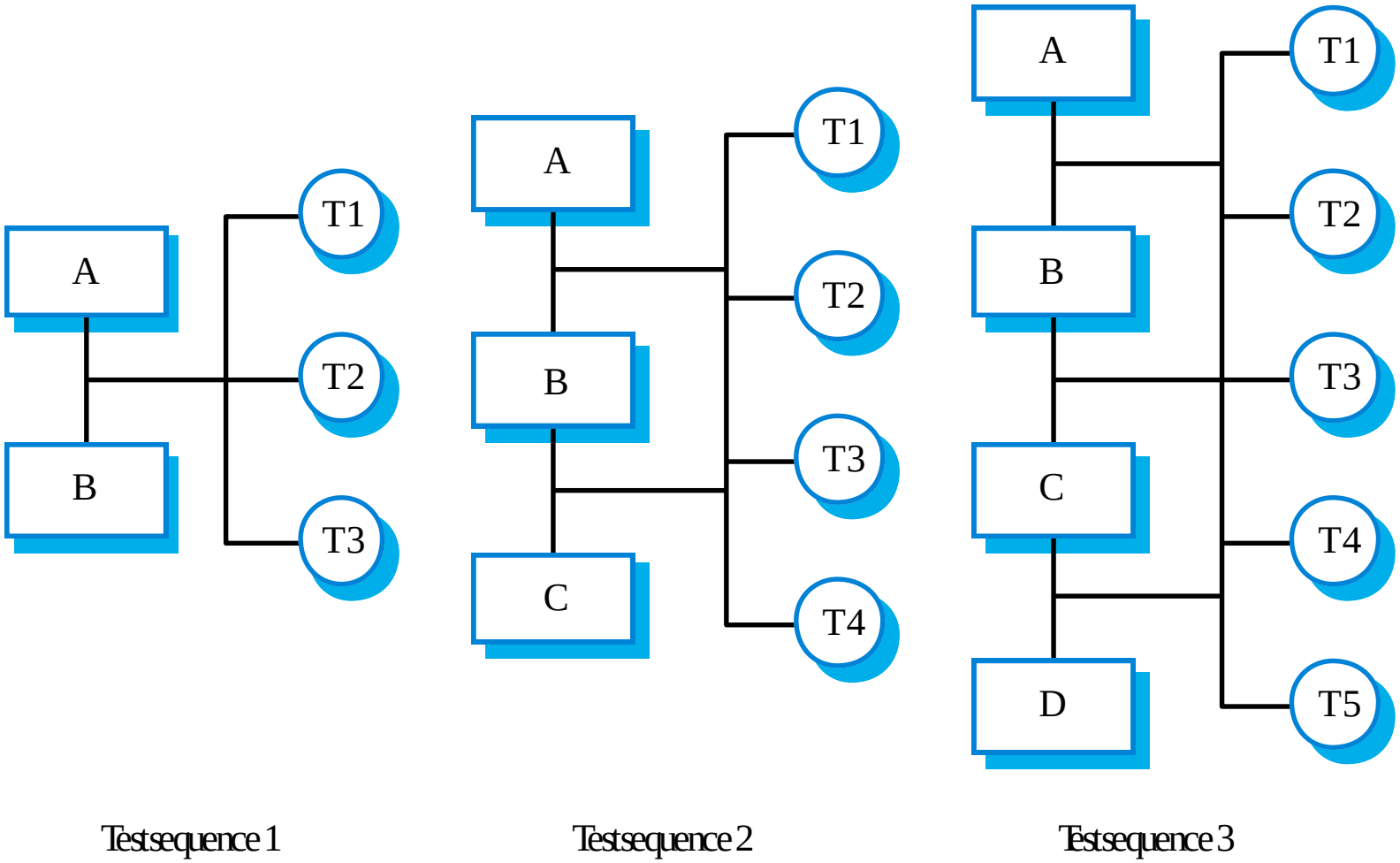
System Testing

- 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 - the test team have access to the system source code. The system is tested as components are integrated
  - Release testing - the test team test the complete system to be delivered as a black-box

Integration Testing

- Involves building a system from its components and testing the system for problems caused by component interactions
- Integration strategies:
  - Top-down integration
    - Develop the skeleton of the system and populate it with components
  - Bottom-up integration
    - Integrate infrastructure components then add functional components
- Errors are simpler to detect when systems are incrementally integrated

Incremental Integration Testing

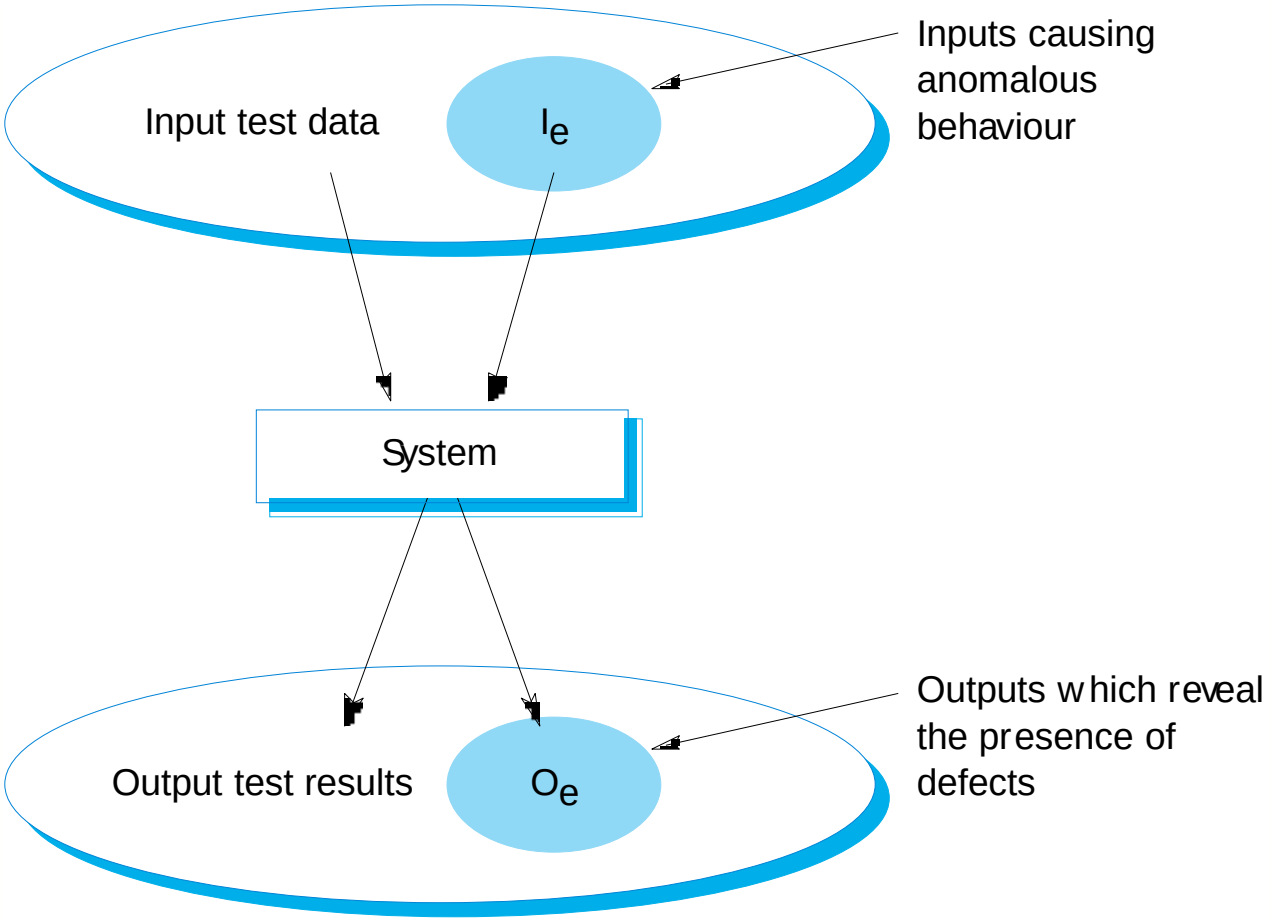


Testsequence 1 Testsequence 2 Testsequence 3

Release Testing

- The process of testing a release of a system that will be distributed to customers
- Primary goal is to increase the stakeholders confidence that the system meets its requirements
- Release testing is usually considered as a black-box or functional testing
  - Based on the system specification only
  - Testers do not have knowledge of, or access to the system implementation (source code or documents)

Black Box Testing



Performance Testing

- Part of release testing may involve testing the emergent properties of a 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 appear
- Tests how the system behaves when it fails. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded

Component Testing(unit testing)

- Component or unit testing is the process of testing individual components in isolation
- It is a defect testing process
- Components may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components made up of several different components

Object Class Testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible events
- Inheritance makes it more difficult to design object class tests. Must test:
  - all operations inherited from super-class(es)
  - all overriding operations performed by sub-classes

Test Case Design

- Involves designing the test cases (inputs and outputs) used to test the system
- The goal of test case design is to create a set of tests that are effective in validation and defect testing
- Design approaches:
  - Requirements-based testing
  - Partition testing
  - Structural testing

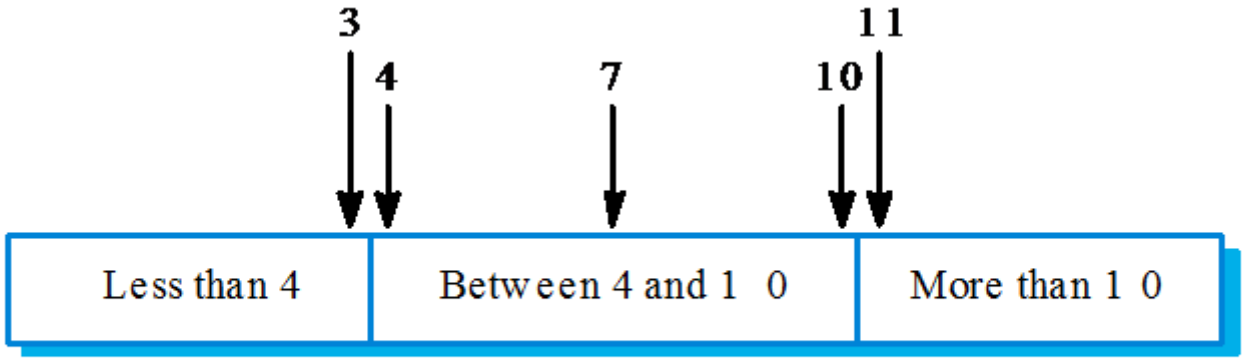
Requirements Based Testing

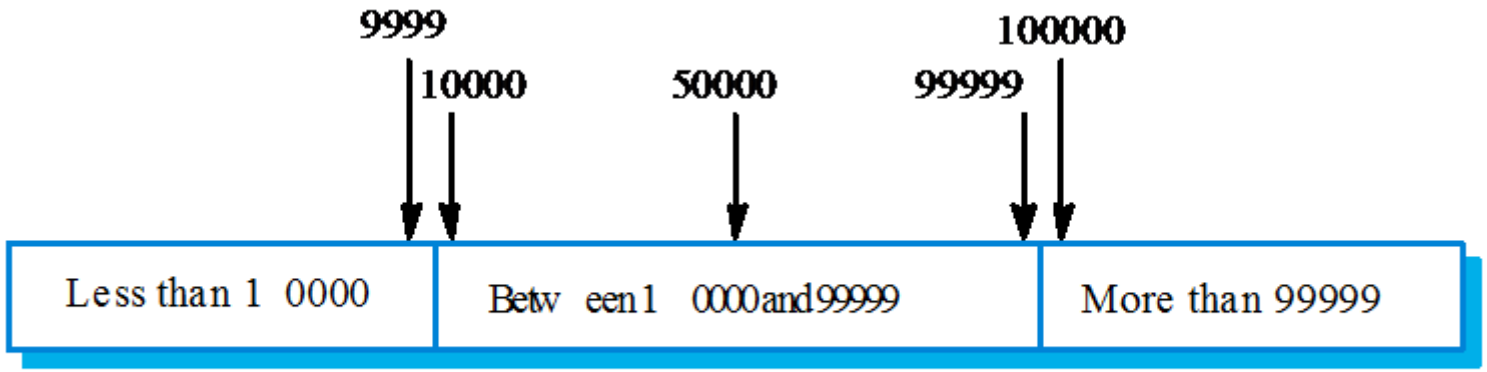
- A general principle of requirements engineering is that requirements should be testable
- Requirements-based testing is a validation testing technique where you derive a set of tests for each requirement

Partition Testing

- Input data and output results often fall into different classes where all members of a class have common characteristics (i.e. positive numbers, strings etc)
- Each of these classes is an equivalence partition or domain where the program normally behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

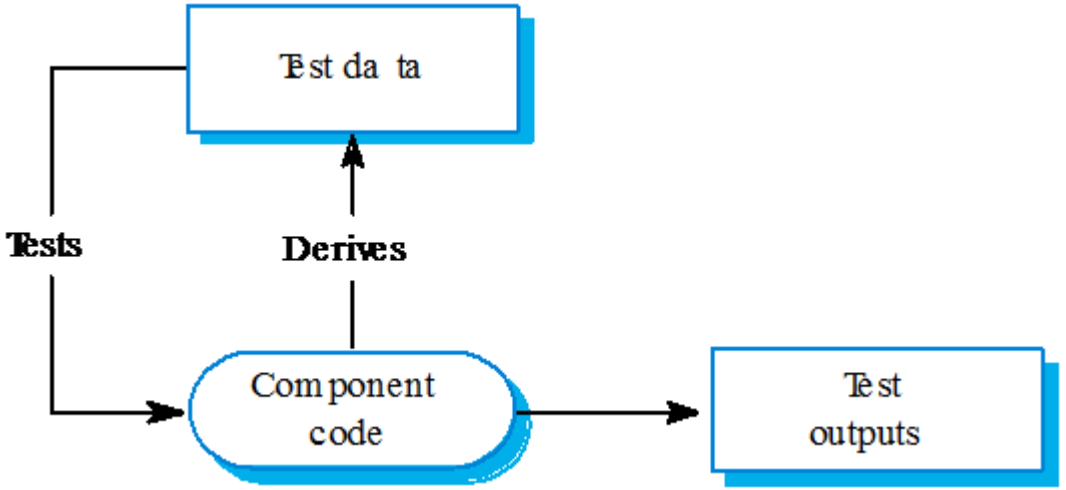
Equivalence Partitions





Structural (White-Box) Testing

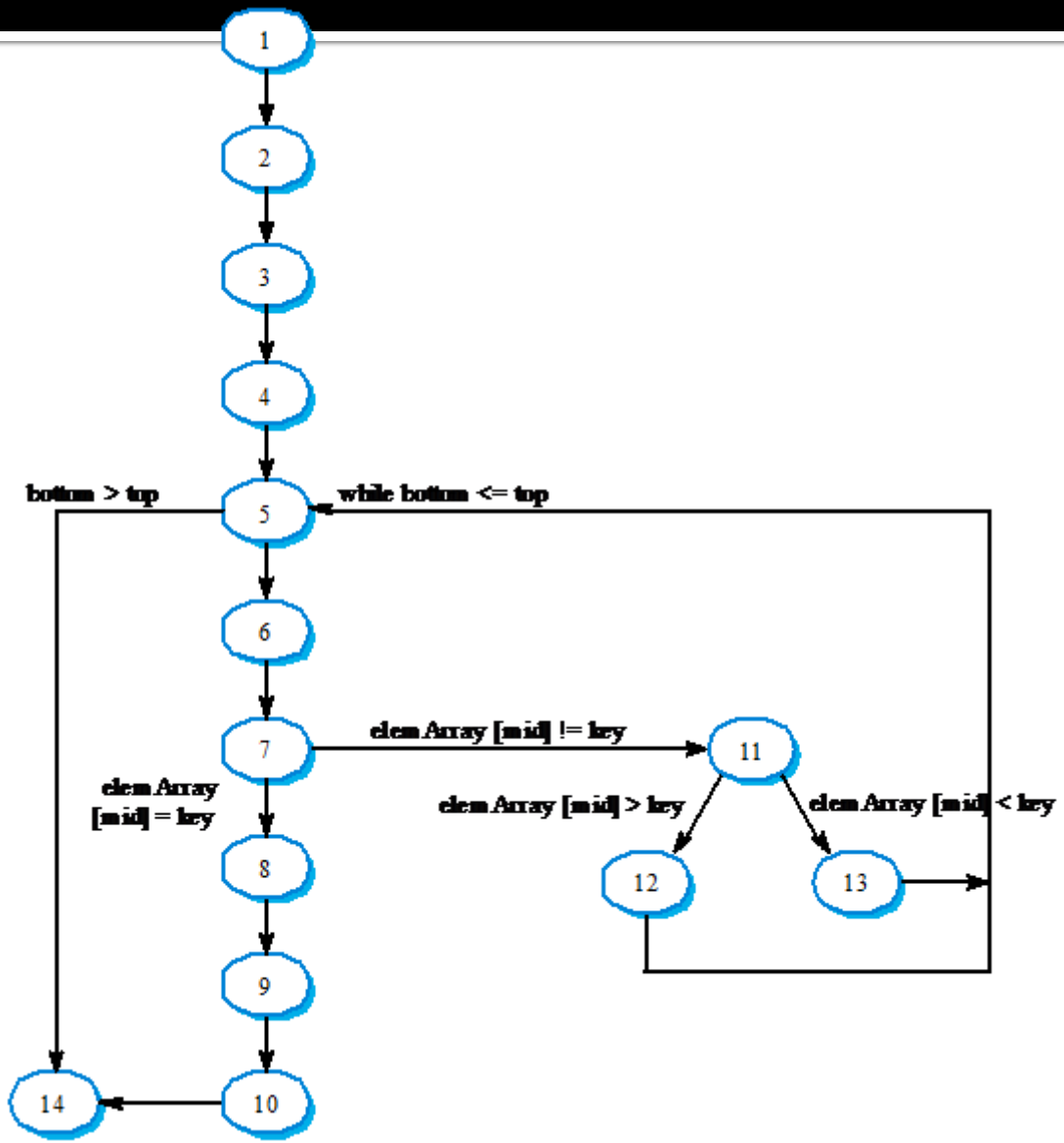
- Derives test cases according to program structure. Uses knowledge of the program to identify additional test cases
- Objective is to exercise all program statements



Path Testing

- The objective of path testing is to ensure that the set of test cases executes each path through the program at least once
- Path testing is based on the program flow graph, where the nodes represent program decisions and the arcs represent control flow
- Statements with conditions are nodes in the flow graph

Example Flow Graph



Independent 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, …

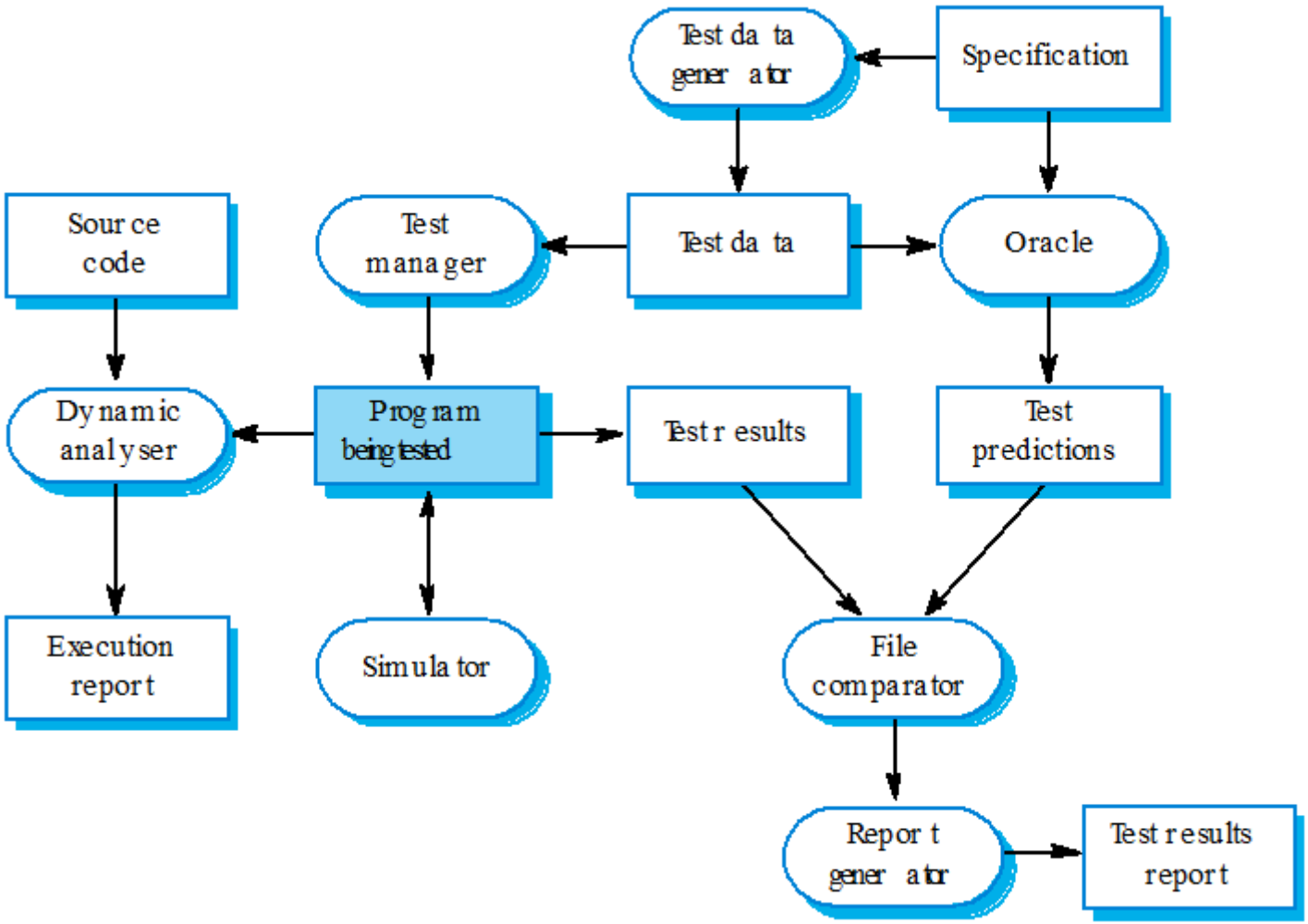
1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …

- Test cases should be derived so that all of these paths are executed
- A dynamic program analyzer 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
- Systems such as Junit support the automatic execution of tests
- Most testing workbenches are open systems because testing needs are organization-specific
- Open testing workbench systems are sometimes difficult to integrate with closed design and analysis workbenches

A Testing Workbench



Testing Workbench Components

- **Test manage**r-Manages the running of program tests
- **Test data generator**-Generates test data for the program to be tested
- **Oracle**-Generates predictions of expected test results
- **File comparator**-Compares the results of program tests with previous test results and reports differences between them
- **Report generator**- Provide report definition and generation facilities for test results.
- **Dynamic analyzer**- Adds code to a program to count the number of times each statement has been executed.
- **Simulator**-Provide different kinds of simulators

Software quality assurance

- Quality assurance covers a broad set of activities to make sure the project is completed based on the previously agreed specifications without any defects.
- SQA activities might include:
  - Training of testers
  - Documentation of test procedures
  - Documentation and analysis of test results
  - Review of test plans
  - Proper management of testing activities
  - Adherence to testing standards
  - Develop and update of inspection checklists

  ….

Responsibilities of the SQA group

- Audit and review of all product deliverables
- Execution of test plans according to the SQA plan, and reporting software problems
- Dealing with new or modified standards and practices while the project is in progress
- Ensuring the proper handling of media and code, and their libraries
- Ensuring quality control over supplied, subcontracted and outsourced products
- Collection of process and product quality related metrics
- Execution of the SQA plan, updating its schedule, and ensuring its progress
- Assessing and auditing the above main SQA responsibilities

Questions?