Contents

1 Definition **2**

2 Objectives **2**

3 Strategic approach to testing **2**
3.1 General characteristics 2
3.2 Verification and validation 2
3.3 Organising software testing 2
3.4 Levels of testing 2
3.5 Testing strategy appliet to Object-Oriented Software 3

4 Ensuring a successful software test strategy **3**

5 Defect testing **3**
5.1 The testing process 4
5.2 Testing priorities 4

6 Black box testing **4**

7 Equivalence partitioning **4**
7.1 Binary search equivalence partitions 4
7.2 Testing guidelines (sequences) 5
7.3 Equivalence class Test cases 5

8 Structural testing **5**

9 Path testing **5**
9.1 Program flow graphs 5
9.2 Cyclomatic complexity 6

10 Path vs Line coverage **6**

11 Integration testing **6**
11.1 Approaches to integration testing 7
11.2 Testing approaches 7

12 Interface testing **7**
12.1 Interface types 7
12.2 Interface errors 7
12.3 Interface testing guidelines 7

13 Stress testing **8**

14 Object-oriented testing **8**
14.1 Testing levels 8
14.2 Object class testing 8
14.3 Object integration 8

15 Approaches to cluster testing **8**

16 Scenario based testing **8**

17 Debugging **9**
17.1 Why is it difficult 9
17.2 Debugging strategies 9
17.2.1 Brute forcing 9
17.2.2 Backtracing 9
17.2.3 Cause elimination 9

18 Key points **10**

1 Definition

Software testing is a formal process carried out by a **specialized testing team** in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer.

2 Objectives

Direct objectives

- To identify and reveal as many errors as possible in the tested software
- To bring the tested software, after correction of the identified errors and retesting, to an acceptable level of quality
- To perform the required tests efficiently and effctively, within the limits budgetary and scheduling limitations

Indirect objectives

- To compile a record of software errors for use in error prevention (by corrective and preventive actions)

3 Strategic approach to testing

3.1 General characteristics

- Software team should conduct effective formal technical reviews
- Testing begins at the component level and work outward toward the integration of the whole system
- Testing is conducted by the **developer** of the software and by **independent test group**.
- Testing and debugging are different activities, but debugging must be accomodated in any testing strategy.

3.2 Verification and validation

- Verification: (Are algorithms coded correctly?). The set of activities that ensure that software corrctly implements specfic function or algorithm.
- Validation (Does it meet user requirements). The set of activities that ensure that the software that has been build is traceable to customer requirements.

3.3 Organising software testing

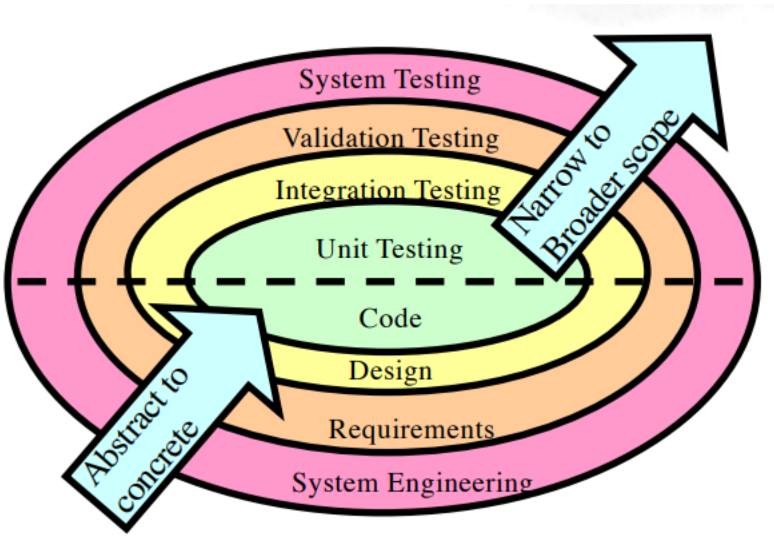
Testing should aim at breaking the software

- Independent test group
  - Removes the inherent problems asociation with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present
  - Works closely with the software developer during analysis and design to ensure that throughout testing occues.

3.4 Levels of testing

- Unit testing: Concentratest on each component/function of the software as implemented in source code
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection.
  - Components are then assembled and integrated
- Integration testing: Focuses on the design and construction of the software architecture
  - Focuses on inputs and outputs, and how well components fit and work together
- Validation testing: Requirements are validated against each constructed software
  - Provides final assurance that the software meets all functional, behaviour, and performance requirements

- System testing: The software and other system elements are tested as a whole
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved



3.5 Testing strategy appliet to Object-Oriented Software

- Include detections of errors in analysis and design models
- Unit testing loses some of its meaning and integration testing changes signifcantly
- Use the same philosophy but different approach as in conventional software testing
- Test "in the smal" and the work out to testing "in the large"
  - Testing in the small involves class attrivutes and operations, main focus in on communication and collaboration within the class
  - Testing in the large involves a series of regression tests to uncover errors due to communication and collabolration among classes
- Finally, the system as a whole is tested to detect errors in fulfilling requirements

4 Ensuring a successful software test strategy

- Specify product requirements in a **quantifiable** manner long before testing commences
- State testing objectives explicitly in measurable terms
- Understand the user of the software (through use cases) and develop a profile for each user category
- Develop a testing plan that emphasizes rapid cycle testing to get quick feedback to control quality levels and adjust the test strategy
- Build robust software that is designed to test itself and can diagnose certain kinds of errors
- Use effective formal technical reviews as a filter prior to testing to reduce the amount of testing required
- Conduct formal technical reviews to asses the test strategy and test casess themselves
- Develop a continous improvement approach for the testing process through the gathering of metrics

5 Defect testing

A successful defect test is a test which causes a program to behave in an anomalous way. Tests show the presence not the absence of defects

5.1 The testing process

Component testing

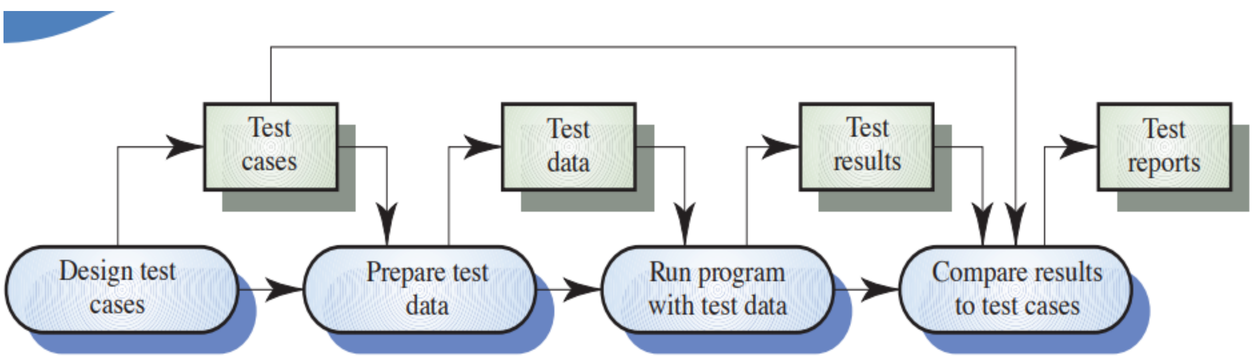
- Testing of individual program components
- Usually the responsiblity of the component developer (except for critical systems)
- Tests are derived for the developers experience

Integration testing

- Testing of groups of components integrated to create a system or a sub-system
- The responsibility of an independent testing team
- Tests are based on a system specification

5.2 Testing priorities

- Only exhaustive testing can show a program is free from defects
- Test should exercise a system's capabilities rather than its components
- Testing old capabilities is more important than testing new capabilities. In order for compatability purposes.
- Testing typical situations is more important than boundary value cases.



6 Black box testing

- An approach to testing where the program is considered as a black-box
- The program test cases are based on the system specification
- Test planning can begin **early** in the software process as we don't need to test the internals, just the visible output

7 Equivalence partitioning

- 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* where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

7.1 Binary search equivalence partitions

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array
- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values Input array has an odd number of values

7.2 Testing guidelines (sequences)

- Test software with sequences which have only a single value
- Use sequences of different sizes in different tests
- Derive tests so that the first, middle and last elements of the sequence are accessed
- Test with sequences of zero length

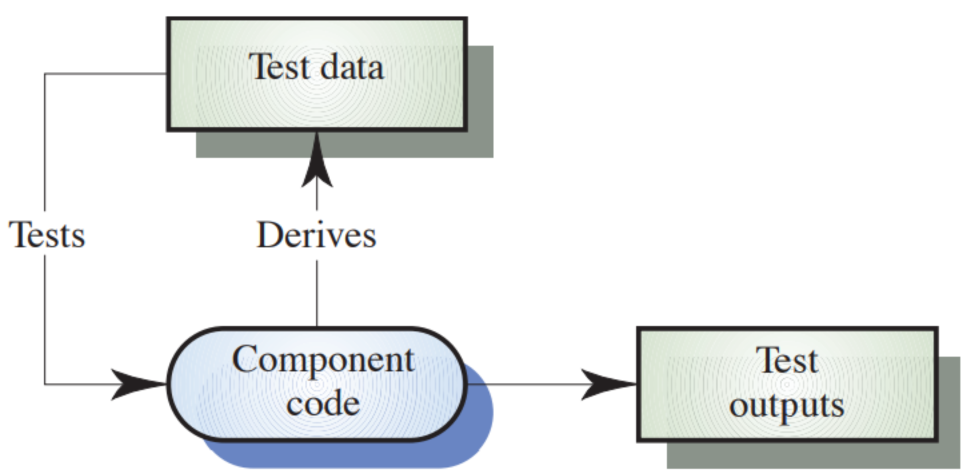
7.3 Equivalence class Test cases

According to the equivalence class (EC) partitioning method:

- Each **valid** EC and each **invalid** EC are included in at least one test case. With definitions made separately.
- In defining a test case for the valid ECs, we try to cover as many as possible new ECs in that same test case.
- In defining invalid ECs, we must assign one test case to **each new invalid EC**, as a test case that includes more than one invalid EC may not allow the tester to distinguish between the programs separate reactions to each of the invalid ECs.
- Test cases are added as long as there are uncovered ECs.

8 Structural testing

- Sometimes called **white-box** testing
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Objective is to exercise all program statements (not all path combinations)



9 Path testing

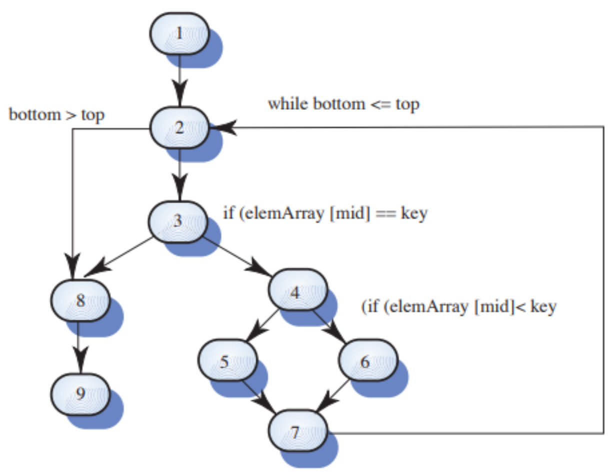
- Ensure that the set of test cases is such that each path through the program is executed at least once
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control
- Statements with conditions are therefore **nodes** in the flow graph

9.1 Program flow graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node
- This is also used as a basis for computing the cyclomatic complexity
- Cyclomatic complexity = Number of edges - Number of nodes +2

9.2 Cyclomatic complexity

- The number of tests to **test all control statements** equals the *cyclomatic complexity*
- Cyclomatic complexity equals number of conditions in a program
- Useful if used with care. Does not imply adequacy of testing.
- Although all paths are executed, **all combinations** of paths are not executed

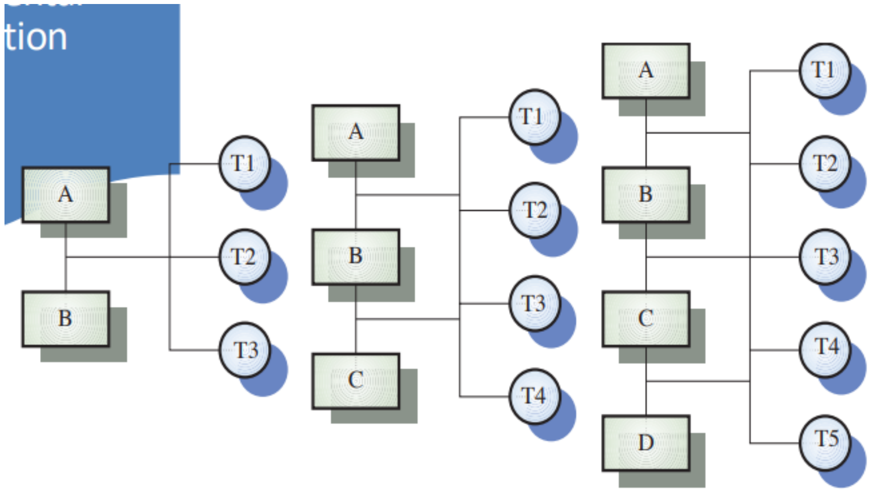


10 Path vs Line coverage

Path coverage of a test is measured by the percentage of all possible program paths included in planned testing. **Line coverage** of a test is measured by the percentage of program code lines included in planned testing.

11 Integration testing

- Tests **complete systems** or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- **Incremental integration** testing reduces this problem



11.1 Approaches to integration testing

- **Top-down testing** Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- **Bottom-up testing** Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

11.2 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

12 Interface testing

- Takes place when modules or sub-systems are integrated to create larger systems
- Objectives are 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

12.1 Interface types

- **Parameter interfaces** Data passed from one procedure to another
- **Shared memory interfaces** Block of memory is shared between procedures
- **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

12.2 Interface errors

- **Interface misuse** A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- **Interface misunderstanding** A 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

12.3 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

13 Stress testing

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

14 Object-oriented testing

- The components to be tested are object classes that are instantiated as objects
- Larger grain than individual functions so approaches to white-box testing have to be extended
- No obvious top to the system for top-down integration and testing

14.1 Testing levels

- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

14.2 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 states
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised

14.3 Object integration

- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these cluster

15 Approaches to cluster testing

- Use-case or scenario testing
  - Testing is based on a user interactions with the system
  - Has the advantage that it tests system features as experienced by users
- Thread testing Tests the systems response to events as processing threads through the system
- Object interaction testing: Tests sequences of object interactions that stop when an object operation does not call on services from another object

16 Scenario based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Consider the scenario in the weather station system where a report is generated

17 Debugging

- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

17.1 Why is it difficult

- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors(e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced
- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors(e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced

17.2 Debugging strategies

- Objective of debugging is to **find and correct the cause of a software error**
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code.

17.2.1 Brute forcing

- Most commonly used and least efficient method
- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

17.2.2 Backtracing

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

17.2.3 Cause elimination

- Involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- Data related to the error occurrence are organized to isolate potential causes
- A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

18 Key points

- Test parts of a system which are commonly used rather than those which are rarely executed
- Equivalence partitions are sets of test cases where the program should behave in an equivalent way
- Black-box testing is based on the system specification
- Structural testing identifies test cases which cause all paths through the program to be executed
- Test coverage measures ensure that all statements have been executed at least once.
- Interface defects arise because of specification misreading, misunderstanding, errors or invalid timing assumptions
- To test object classes, test all operations, attributes and states
- Integrate object-oriented systems around clusters of objects

Reference section

equivalence partitioning

Equivalence partitioning or equivalence class partitioning (ECP) is a software testing technique that divides the input data of a software unit into **partitions of equivalent data from which test cases can be derived**. In principle, test cases are designed to cover each partition at least once.