

Software Engineering

Software Testing

Learning Outcomes

Be able to differentiate between different approaches to Validation and Verification

Understand the testing process

Be able to produce test cases using black-box and white box testing

Be able to apply static verification

Validations and Verification

- Validation and verification (V & V) is the name given to the checking and analysis processes that ensure that software conforms to its specification and meets the needs of the customers who are paying for the software.
- V & V is a whole life-cycle process. It starts with requirements reviews and continues through design reviews and code inspections to product testing. There should be V& V activities at each stage of software process.
- Validation : Are we building the right product?
Verification : Are we building the product right?

Validation and Verification

Within the V & V process, two techniques of system checking and analysis may be used:

Software Inspections- Analyse and check system representations such as the requirements documents, design diagrams and program source code. They may be applied at all stages of the development process. Inspections may be supplemented by some automated analysis of the source text of a system or associated documents. Software inspections and automated analysis are static V & V techniques as they do not require the system to be executed.

Software Testing - involves executing an implementation of the software with test data and examining the outputs of the software and its operational behaviour to check that it is performing as required. Testing is a dynamic technique of V & V because it works with an executable representation of the system.

Software Testing Procedure

- ❑ Testing procedures should be established at the start of any software project. All testing carried out should be based on a test plan, this should detail which tests are to be carried out. For each test, the following information should be included in the test plan:
 - * the pre-requisites for the tests.
 - * the steps required to carry out the tests
 - * The expected results of the test.

The outcome of any tests should be recorded in a test results document that include whether the test succeeded or failed and a description of the failure. Test results for all passes through the test plan must be recorded to allow accurate records to be kept of where problems occur and when they were identified and corrected.

Testing Process

1. Run the tests as defined by the test plan.

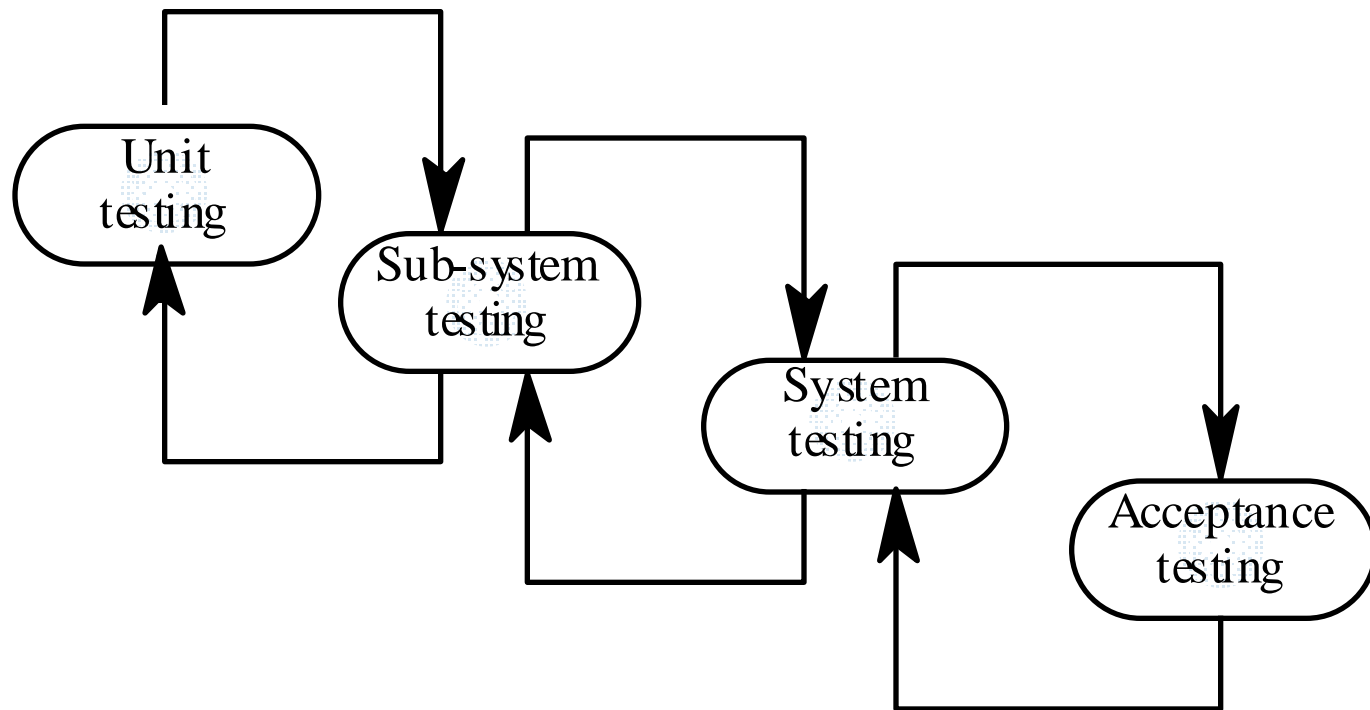
Note : Testing should not stop when the first problem is encountered, unless it is so severe that the rest of the tests would be meaningless. Rather all testing in the test plan should be carried out and then the errors addressed.

2. Record the outcome of each test in the test report, both success and failure should be reported. For failed tests the nature of the problem should be described in sufficient detail to allow it to be corrected and to allow analysis of the types of errors being found.
3. Correct the errors that were documented from the test run.
4. Repeat the process until no errors are identified or error rate is sufficiently low. If the error rate is low then it may be sufficient to simply re-test the failed errors. If the error rate is high then all tests should be re-run.

Dynamic and static verification

- ❑ *Dynamic verification* Concerned with exercising and observing product behaviour (testing)
 - includes executing the code
- ❑ *Static verification* Concerned with analysis of the static system representation to discover problems
 - does not include execution

The dynamic testing process



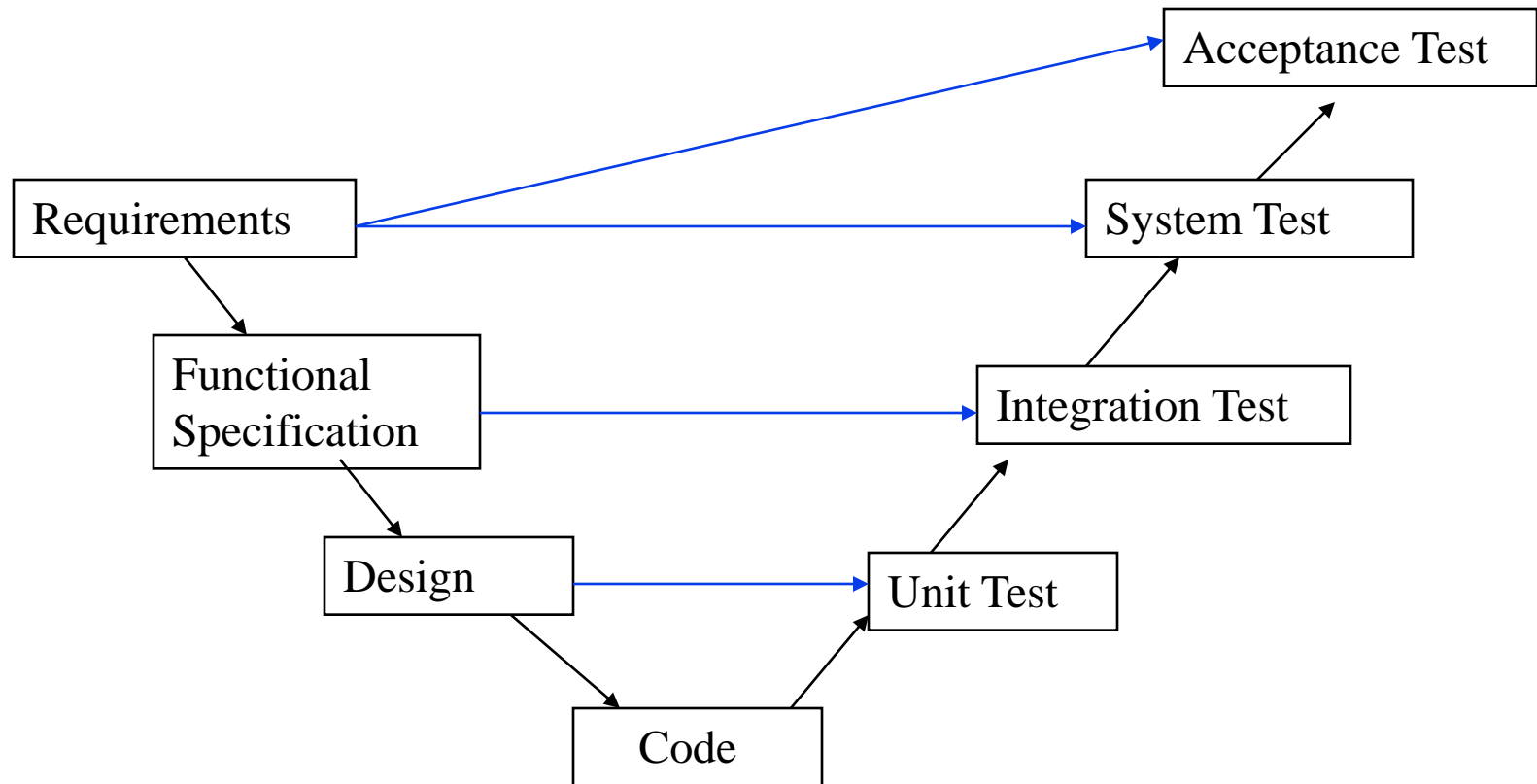
individual
components

collections of
components (sub-
systems

The whole
finished system
- developers

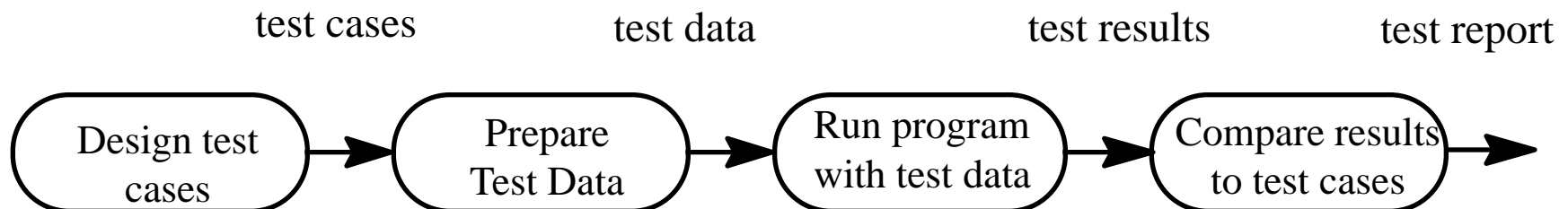
The finished
system - users

Testing in the project lifecycle



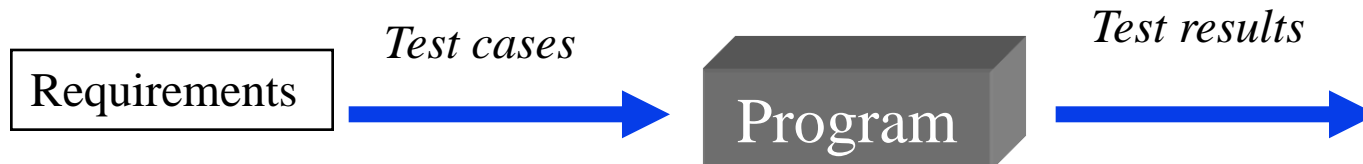
Defect testing

- ❑ Defect Testing objective - to discover defects in programs
- ❑ Successful defect test - a test which causes a program to behave in an anomalous way
- ❑ Tests show the *presence* not the *absence* of defects
- ❑ Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible
- ❑ *Test data* = Inputs which have been devised to test system
- ❑ *Test cases* = Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification.



Black-box testing

- ❑ Approach to testing where the program is considered as a ‘black-box’
- ❑ The program test cases are based on the system specification
- ❑ Inputs from test data may reveal anomalous outputs i.e. defects
- ❑ Test planning can begin early in the software process
- ❑ Main problem - selection of inputs
 - equivalence partitioning

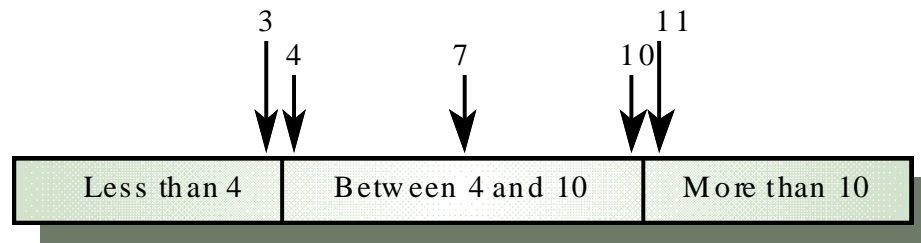


Equivalence partitioning

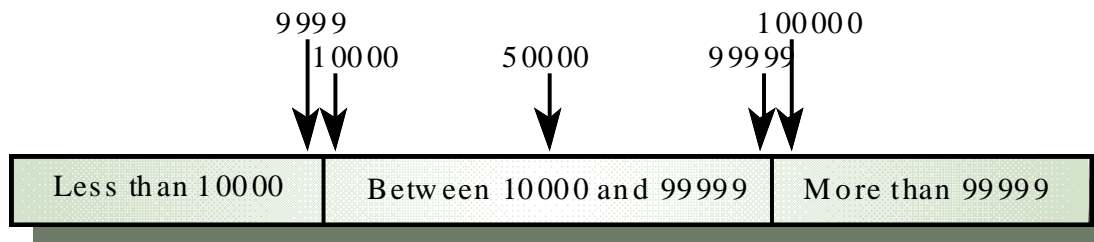
- ❑ Partition system inputs and outputs into ‘equivalence sets’
 - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are
 - » $<10,000$, $10,000-99,999$ and $> 99,999$
 - Input equivalence partitions are sets of data where all of the set members should be processed in equivalent way.
 - Once you have identified a set of partitions, you then choose test cases from each of these partitions. A good guideline to follow for test case selection is to choose test cases on the boundaries of the partitions plus cases close to the mid point of the partition.

Equivalence partitioning

- Choose test cases at the boundary of the equivalence sets



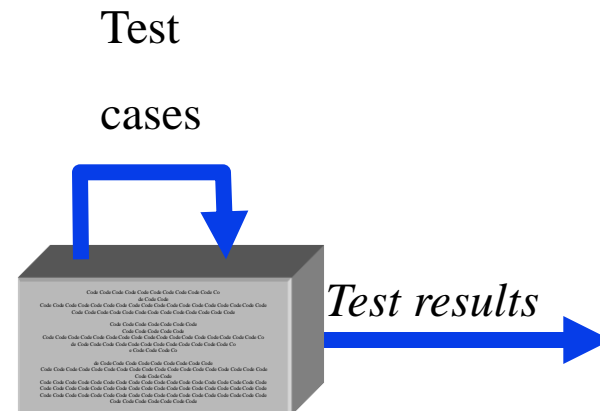
Number of input values



Input values

Structural testing

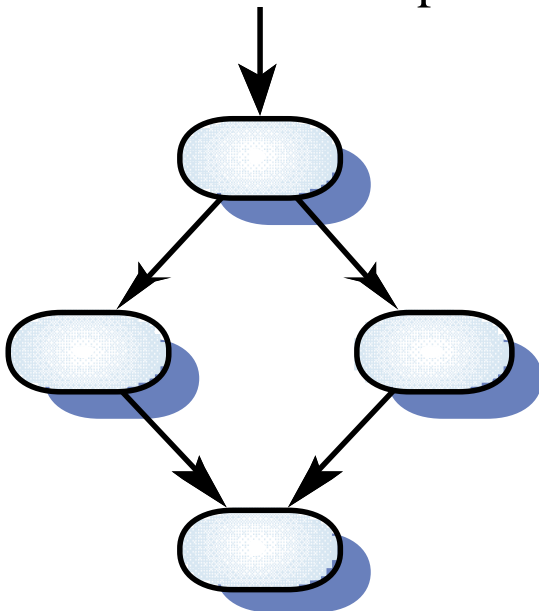
- ❑ Sometimes called **white-box** testing or glass box testing
- ❑ Derivation of test cases according to program structure.
Knowledge of the program used to identify additional test cases
- ❑ Objective is to exercise all program statements (not all path combinations)



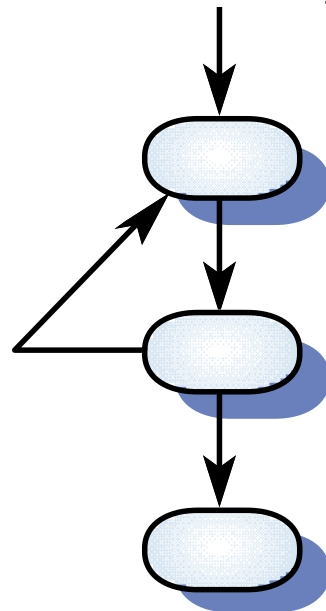
Path Testing

□ Program flow graphs

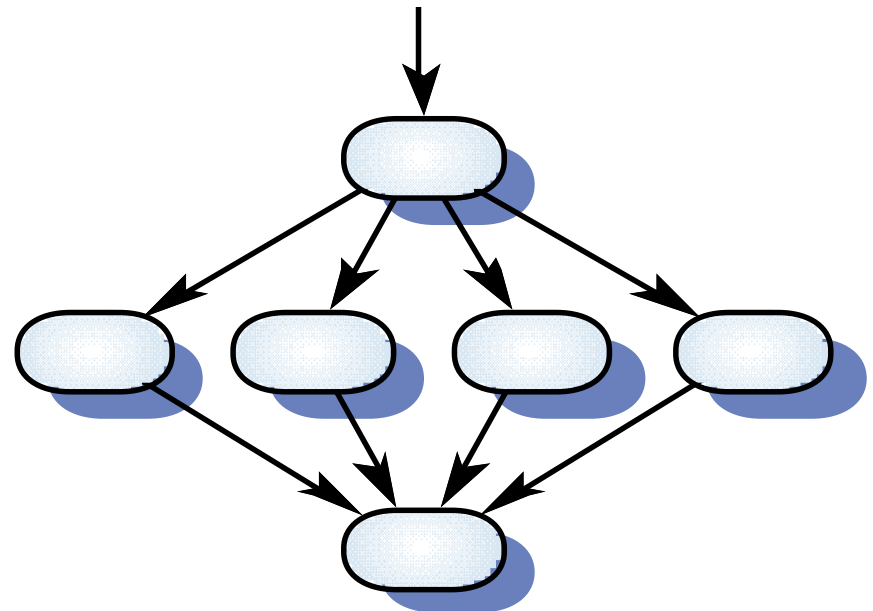
- Describes the program control flow
- Used as a basis for computing the cyclomatic complexity
- $\text{Complexity} = \text{Number of edges} - \text{Number of nodes} + 2$



if-then-else



loop-while



case-of

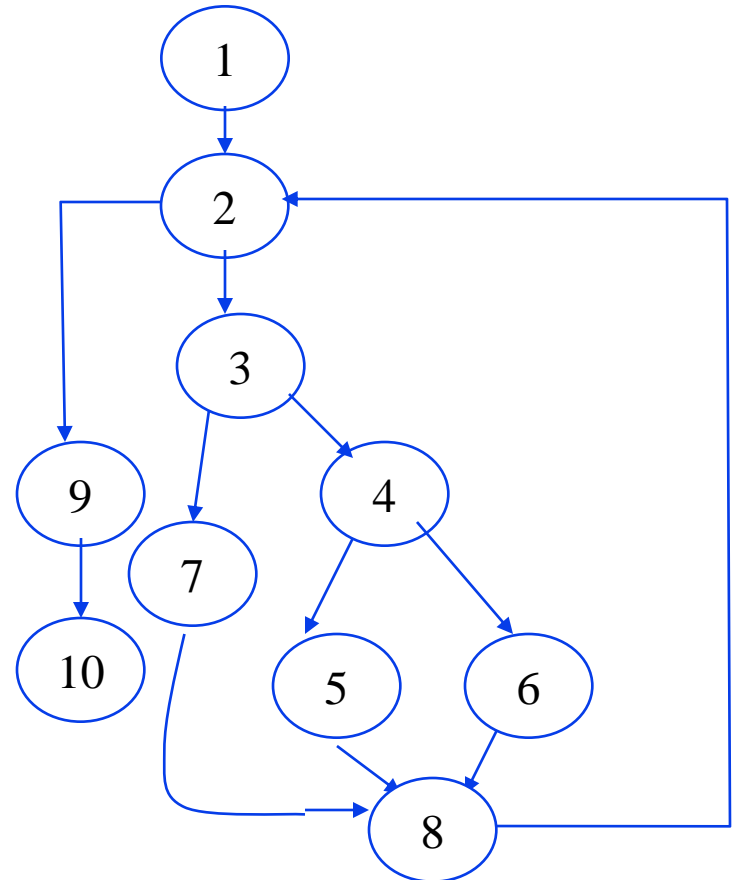
Binary search flow graph

```

class BinSearch {
public static void search( int key, int[] elemArray, Result r)
{
    int bottom = 0;
    int top = elemArray.length - 1;
    int mid;
    r.found = false; r.index = -1;

    while (bottom <= top)
    {
        mid = (top + bottom) / 2;
        if (elemArray [mid] == key)
        {
            r.index = mid;
            r.found = true;
            return;

        }
        else
        {
            if(elemArray[mid] < key)
                bottom = mid + 1;
            else
                top = mid -1;
        }
    }
} //while
} //search
} //BinSearch
    
```



Cyclomatic Complexity (CC)= Number of edges – Number of nodes+2

$$CC = 12 - 10 + 2 = 4$$

Independent paths

- ❑ CC = The number of tests to test all control statements equals = number of conditions in a program
- ❑ Independent Paths
 - 1, 2, 9, 10
 - 1, 2, 3, 7, 8, 9, 10
 - 1, 2, 3, 4, 5, 8, 9, 10
 - 1, 2, 3, 4, 6, 8, 9, 10
- ❑ 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

Static verification

- ❑ Verifying the conformance of a software system and its specification *without* executing the code
- ❑ Involves analysis of source text by humans or software
- ❑ on ANY documents produced as part of the software process
- ❑ Discovers errors early in the software process
- ❑ Usually more cost-effective than testing for defect detection at the unit and module level
- ❑ > 60% of program errors can be detected program inspections
- ❑ 2 main techniques
 - **Code walkthroughs** – The author explain the code to the other team members. They examine the code and suggest improvements.
 - **Code reviews (program inspections)** – The author distribute the code lists to the team members. At a review meeting improvements are suggested.

Test Phases

Test Phase	Test Plan	Author	Technique	Run by
Unit Test	Code design	Designer	White Box, Black box, static	Programmer
Integration Test	Functional specification	Author of specification	Black box, white box, Top-down, bottom-up	Programming team
System Test	Requirements	Analyst	Black box, stress testing, performance testing	System test team
Acceptance Test	Requirements	Analyst/customer	Black box	Analyst/customer
Alpha Test	No test plan		Black box	Selected set of users
Beta Test	No test plan		Black box	Any user
Regression Test	Functional specification/Requirements	Analyst	Black box	Development team, system test team

Unit Testing

Unit Testing is carried out as a part of the coding task. This phase is based on the design of the software for a piece of code. Unit testing should prove the following about the code

- **Robust** – the code should not fail under any circumstances.
- **Functionally correct** – the code should carry out the task defined by the code design
- **Correct interface** – the inputs and outputs from the code should be as defined in the design

Unit Test Plan –

The unit test plan must be based on the design of the code and not the code itself. Therefore, the test plan will be written after the completion of design but before the start of the coding phase.

The test plan is the responsibility of the designer of the code. The testing is usually carried out by the author of the code.

Integration/Sub-systems Testing

Integration Testing is carried out after the separate software modules have been unit testing. Integration testing is based on the functional specification of the software. Integration testing should prove the following about the software:

Integration - the modules of the system should interact as designed.

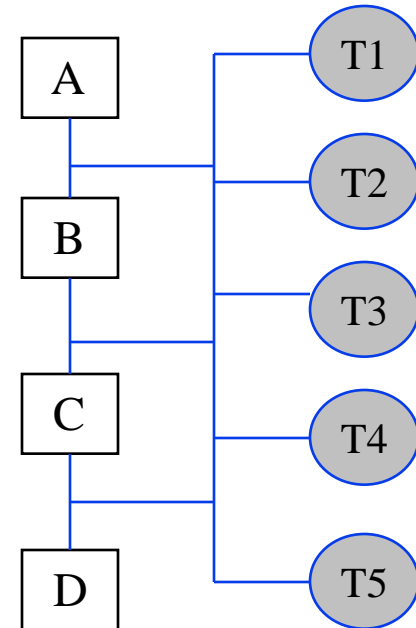
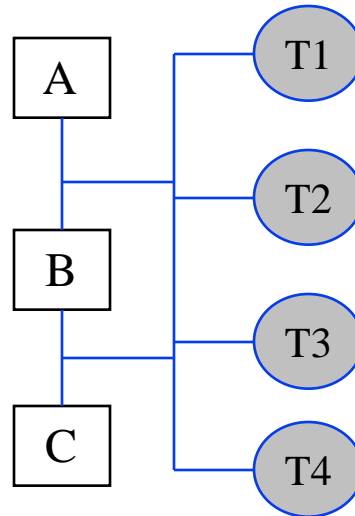
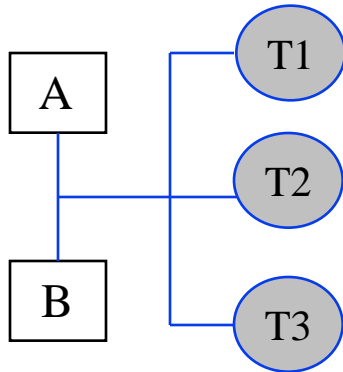
Functionally correct - the system should behave as defined in the functional specification

Integration test plan –

This is based on the specification of the system. The test plan is usually written by the authors of the system specification to avoid any assumptions made during design from being incorporated into the test plan.

Incremental Integration Testing

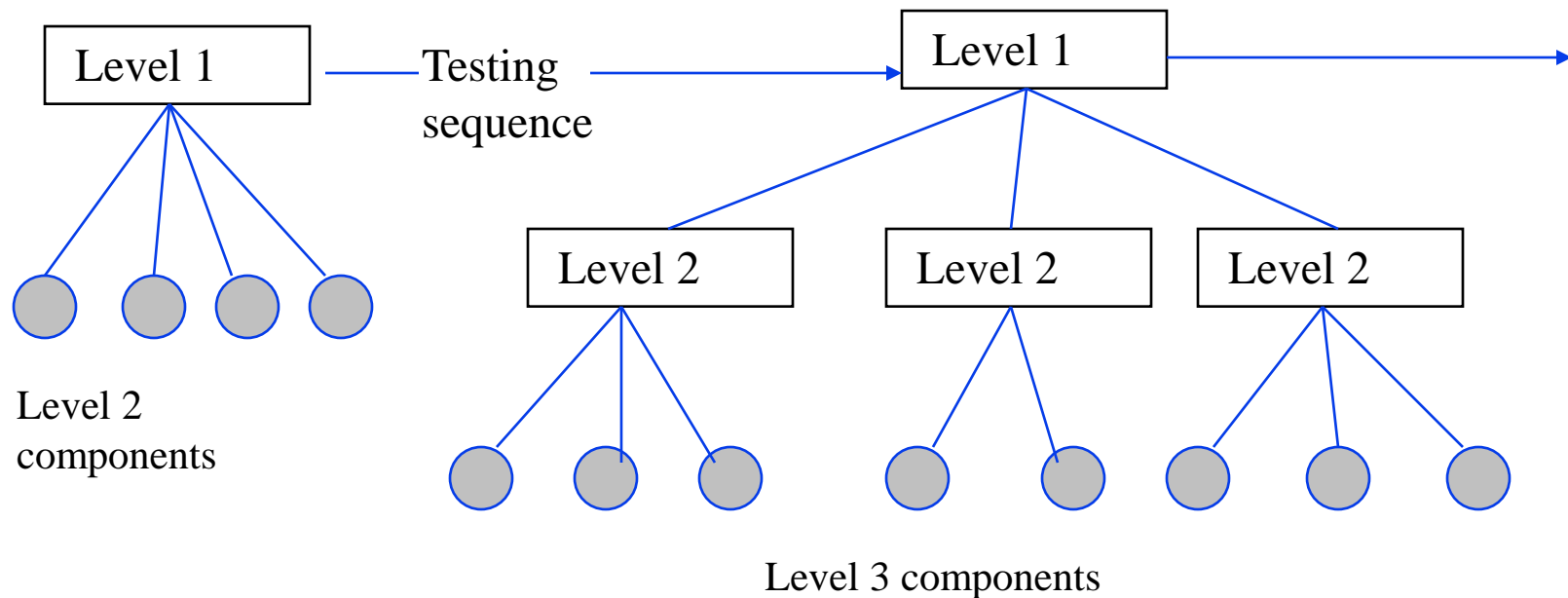
Initially, integrate a minimal system configuration and test the system. Then add components to this minimal configuration and test after each added increment.



T1, T2 and T3 tests are carried out after integration of components A and B. If the tests are successful, add C and carry out tests T1, T2, T3 and T4. If new errors introduced, that is due to the integration of C.

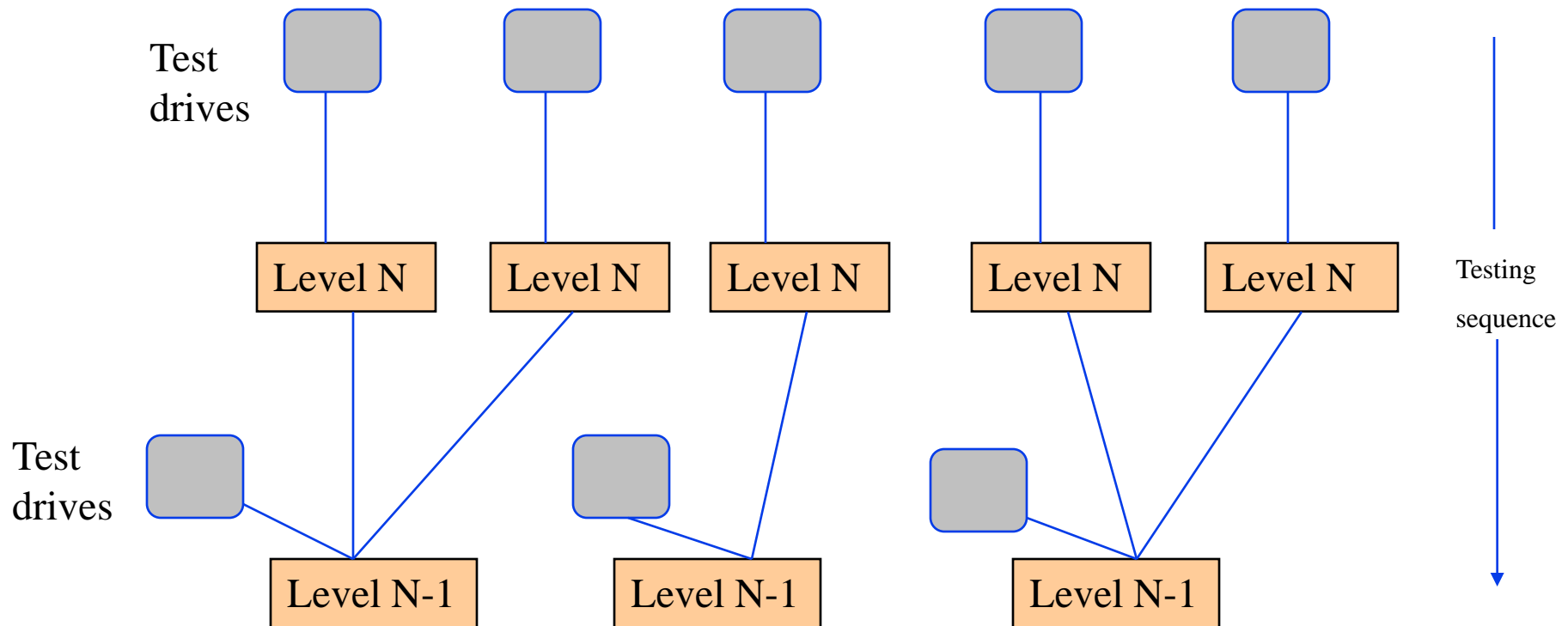
Top-Down Integration Testing

In top-down integration, the high-level components of a system are integrated and tested before design and implementation of some of the high-level components has been completed.



Bottom-up Integration Testing

Low-level components are integrated and tested before the higher-level components have been developed.



System Testing

System testing is carried out at the completion of the integration testing. The purpose of system testing is to prove that the software meets the agreed user requirements and works in the target environment. System testing covers both functional and non-functional requirements.

System Test Plan-

The system test plan is based around the agreed requirements. Test plan covers functional requirements and non-functional requirements such as performance. The system test plan is written by the author of the requirements document to avoid assumption introduced during specification. The system test plan will also include tests to cover

- **Recovery** – Force the system to crash and then try to recover to a sensible state.
- **Security** – Attempt to access the system without the correct authority, or attempt to carry out restricted functions.
- **Stress** - Attempt to break the system by overloading it.
- **Performance**- Ensure the system meets the performance requirements.

Acceptance Testing

Acceptance testing is carried out at the customers site with the customer in attendance. The purpose of the acceptance test is to show to the customer that the software does indeed work. These tests are usually a sub set of the system test.

Acceptance test plan –

This should be agreed with the customer after the requirements for the software have been agreed. Sometimes the customer will write the test plan in which case it must be agreed with the software developers.

Alpha, Beta and Regression Testing

Alpha Testing –

Alpha testing is the first real use of the software product. Having completed system testing the product will be given to a small number of selected customers or possibly just distributed within the company itself. The software will be used ‘in anger’ and errors reported to the development and maintenance team.

Beta Testing –

After alpha testing has been completed and any changes made a new version of the product will be released to much wider audience. The objective of Beta testing is to iron out the remaining problems with the product before it is put on the general release.

Regression Testing –

Regression testing carried out after any changes are made to a software application. The purpose of regression test is to prove that the change has been made correctly and that the change has not introduced any new errors.

Regression test plans are usually a sub-set of the integration or systems test plans. It is designed to cover enough functionality to give confidence that the change has not effected any other part of the system.

Back-to-back testing and Thread testing

Back-to-back testing

Back-to-back testing means that multiple versions of the software are executed on the same test data. If they all agree on the answer, it is assumed that they are all correct.

Thread testing

Based on testing an operation which involves a sequence of processing steps which thread their way through the system

Start with single event threads then go on to multiple event threads

Advantages

Suitable for real-time and object-oriented systems

Disadvantage

Difficult to assess test adequacy, because of large number of event combinations

Object Oriented Testing

In an object oriented system, four levels of testing can be identified:

1. ***Testing the individual operations associated with objects*** – These are functions or procedures and the black-box and white-box approaches may be used.
2. **Testing individual object classes** – The principle of black-box testing is unchanged but the notion of an equivalence class must be extended to cover related operation sequences.
3. ***Testing clusters of objects*** - Strict top-down or bottom-up integration may be inappropriate to create groups of related objects. Other approaches such as scenario based testing should be used.
4. ***Testing the object oriented system*** - V & V against the systems requirements specification is carried out in exactly the same way as for any other type of system.

Object Class Testing

When testing objects, complete test coverage should include:

1. the testing in isolation of all operations associated with the object;
2. the setting and interrogation of all attributes associated with the object;
3. the exercise of the object in all possible states. This means that all events that cause a state change in the object should be simulated.

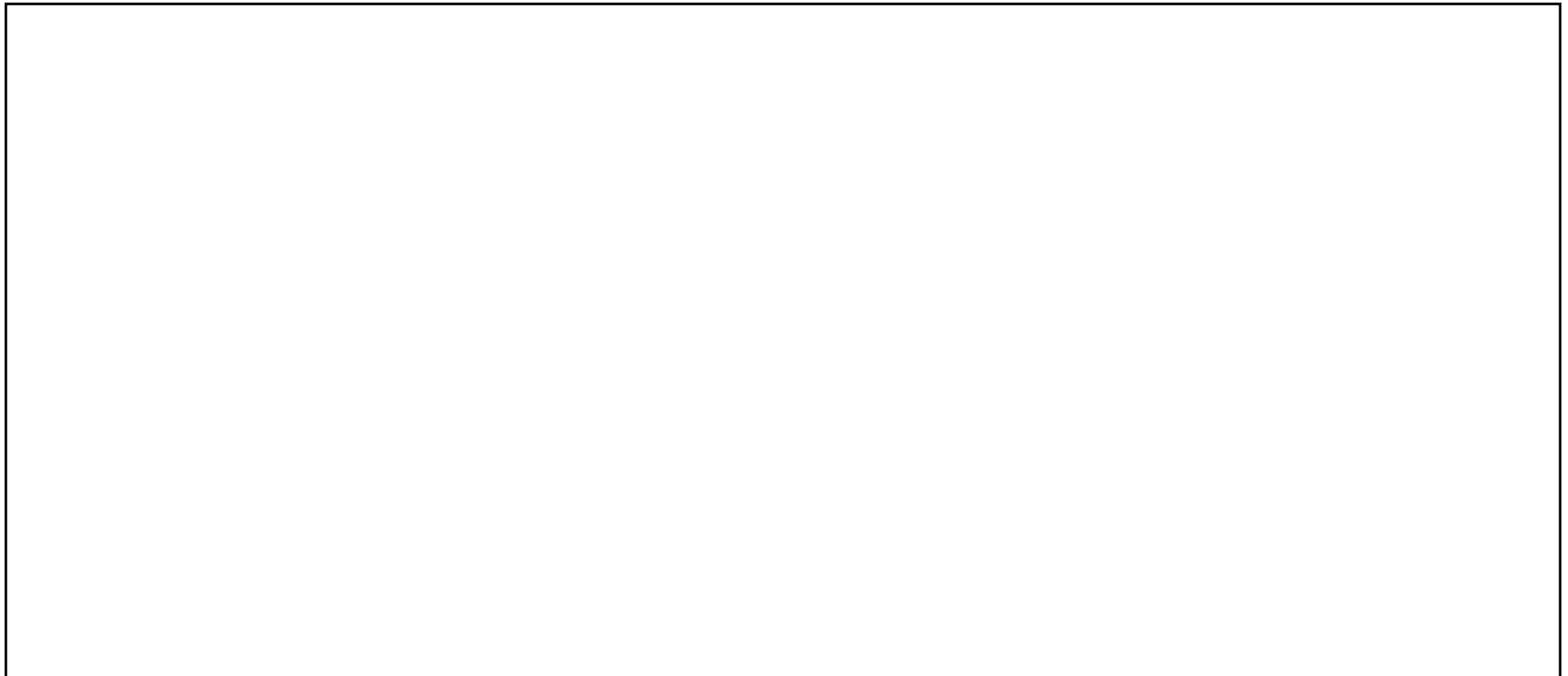
Object Integration

In object oriented systems, there is no obvious ‘top’ that provides for the integration nor is there a clear hierarchy of objects that can be created. Clusters therefore have to be created using knowledge of their operation and the features of the system that are implemented by these clusters. There are three possible approaches to integration testing that may be used:

1. ***Use-case or scenario- based testing*** – Testing can be based on the scenario descriptions and object clusters created to support the use-cases that relate to that mode of use.
2. ***Thread testing*** – Thread testing is based on testing the system’s response to a particular input or set of input events.
3. ***Object interaction testing*** – An intermediate level of integration testing can be based on identifying ‘method – message ‘ paths. These are traces through a sequence of object interactions which stop when an object operation does not call on the services of any other object.

Review Questions

1. Explain the difference between validation and verification in software engineering. Give examples of two validation and two verification activities in a typical software development process.



Review questions (cont..)

2. Use a 'black box' approach to generate test cases for the Java method that calculates the perimeter of a rectangle whose sides are a and b. The sides a and b are represented by integer parameters of this method.

Equivalence classes :

Review questions (cont...)

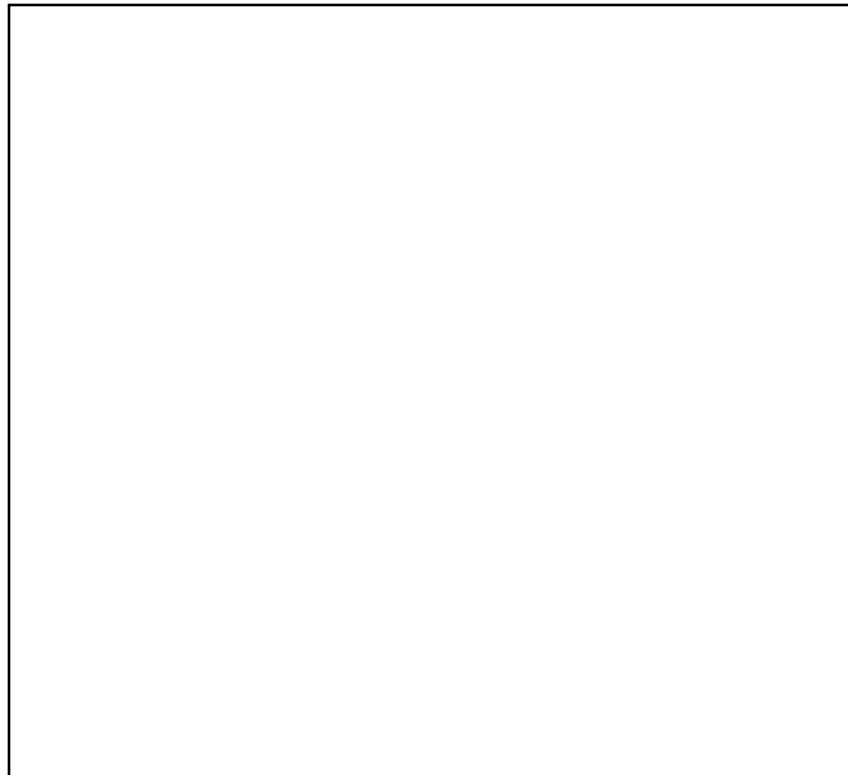
Values (boundary in particular)

Test cases

Review questions Cont..

Draw a flow graph and calculate cyclomatic complexity of the following program segment.

```
IF A Then  
  B  
  If C Then  
    D  
    E  
  Else  
    If F Then  
      G  
    Else  
      H  
  Endif  
Endif  
Endif
```



CC=