Software Validation, Verification and Testing
Recap on SDLC Phases & Artefacts

Domain Analysis @ Business Process

Requirement

Analysis

Design

Implementation

Testing & Deployment

Maintenance & Evolution

Domain Model (Class Diagram)

1) Functional & Non-Functional requirement
2) Use Case diagram

1) System Sequence Diagram
2) Activity Diagram

1) Class Diagram (refined)
2) Detail Sequence Diagram
3) State Diagram

1) Application Source Code
2) User Manual Documentation

1) Test Cases
2) Prototype / Release/Versions

1) Change Request Form
Sub-Topics Outline

• Verification, validation
  – Definition, Goal, techniques & purposes

• Inspection vs. testing
  – Complementary to each other

• Software testing
  – Definition, goal, techniques & purposes
  – Stages: development, release, user/customer
  – Process: test cases, test data, test results, test reports
    • Focus in designing test cases to perform testing based on 3 strategies:
      i. requirement-based
      ii. black-box
      iii. white box
Objectives

1. To discuss about V & V differences, techniques
2. To know different types of testing and its definition
3. To describe strategies for generating system test cases
VERIFICATION & VALIDATION (V & V)
Verification vs validation (Boehm, 1979)

• 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 : Goal

• Verification and validation should establish confidence that the software is **fit 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 : Degree of Confidence

• 3 categories of degree-of-confidence:

1. **Software function/purpose**
   • The level of confidence depends on **how critical the software is to an organisation**.
     (i.e. safety-critical system)

2. **User expectations**
   • Users may have **low expectations of certain kinds of software**.
     (user previous experiences – i.e. buggy & unreliable software especially newly installed software)

3. **Marketing environment**
   • Getting a product to market early may be more important than finding defects in the program.
     (competitive environment – release program first without fully tested to get the contract from customer)
V&V: The Techniques

• Validation Technique
  1. Prototyping
  2. Model Analysis (e.g. model checking)
  3. Inspection and reviews (Static Analysis)

• Verification Technique
  4. Software Testing (Dynamic verification)
  5. Code Inspection (Static verification)

• Independent V&V
Technique: Prototyping (Validation)

• “A software prototype is a partial implementation constructed primarily to enable customers, users, or developers to learn more about a problem or its solution.” [Davis 1990]

• “Prototyping is the process of building a working model of the system” [Agresti 1986]
Technique: Model Analysis (V & V)

• Validation
  – Animation of the model on small examples
  – Formal challenges:
    • “if the model is correct then the following property should hold...”
  – ‘What if’ questions:
    • reasoning about the consequences of particular requirements;
    • reasoning about the effect of possible changes
    • “will the system ever do the following...”

• Verification
  – Is the model well formed?
  – Are the parts of the model consistent with one another?
Technique: Model Analysis Example
Basic Cross-Checks for UML (Verification)

Use Case Diagrams
- Does each use case have a user?
  - Does each user have at least one use case?
- Is each use case documented?
  - Using sequence diagrams or equivalent

Class Diagrams
- Does the class diagram capture all the classes mentioned in other diagrams?
- Does every class have methods to get/set its attributes?

Statechart Diagrams
- Does each statechart diagram capture (the states of) a single class?
  - Is that class in the class diagram?
- Does each transition have a trigger event?
  - Is it clear which object initiates each event?
  - Is each event listed as an operation for that object's class in the class diagram?
- Does each state represent a distinct combination of attribute values?
  - Is it clear which combination of attribute values?
  - Are all those attributes shown on the class diagram?
- Are there method calls in the class diagram for each transition?
  - ...a method call that will update attribute values for the new state?
  - ...method calls that will test any conditions on the transition?
  - ...method calls that will carry out any actions on the transition?

Sequence Diagrams
- Is each class in the class diagram?
- Can each message be sent?
  - Is there an association connecting sender and receiver classes on the class diagram?
  - Is there a method call in the sending class for each sent message?
  - Is there a method call in the receiving class for each received message?
Technique: Software inspections (Validation)

- These involve people examining the source representation with the aim of discovering anomalies (deviation from standard/expectation) and defects. (errors)
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.
Inspections (static) and testing (dynamic)
Inspections (static) and testing (dynamic)
Inspections (static) and testing (dynamic)
Advantages of inspections

1. During testing, errors can mask (hide) other errors. Because inspection is a static process, you **don’t have to be concerned with interactions between errors.**

2. **Incomplete versions** of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.

3. As well as searching for program defects, an inspection can also **consider broader quality attributes of a program**, such as compliance with standards, portability and maintainability. (i.e. inefficiencies, inappropriate algorithms, poor programming style which make system difficult to maintain & update)
Inspections vs. testing?

• **Software inspections** and reviews concerned with check and analysis of the static system representation to discover problems ("static" verification: *no execution needed*)
  – May be supplement by tool-based document and code analysis.
  – Discussed in Chapter 15 (Sommerville’s).

• **Software testing** concerned with exercising and observing product behaviour ("dynamic" verification: *needs execution*)
  – The system is executed with test data and its operational behaviour is observed.
  – “*Testing can only show the presence of errors, no their absence*” (Dijkstra et.al. 1972)
Inspections vs. 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 (system) but not conformance with the customer’s real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.
SOFTWARE TESTING : STAGES
Recap on software testing

- **Software testing** concerned with **exercising** and observing product behaviour
- **Dynamic verification** - The system is executed with test data and its operational behaviour is observed.
- “Testing can only show the presence of errors, no their absence” (Dijkstra et.al. 1972)
Stages in Software Testing

1. Development
   a) Component
      i. Object/Class
      ii. Interface
         - Parameter
         - Procedural
         - Message Passing
   b) System
      - Phase
         - Integration
            - Top-down
            - Bottom-up
         - Release

2. Release

3. User/Customer
   a) Alpha
   b) Beta
   c) Acceptance

Types
- Stress
- Performance
- Usability
Stages of testing

Commercial software system has to go through 3 stages of testing:

1. **Development testing**
   - where the system is tested during development to discover bugs and defects.

2. **Release testing**
   - where a separate testing team test a complete version of the system before it is released to users.

3. **User testing**
   - where users or potential users of a system test the system in their own environment.
Stages in Software Testing

1. Development
   a) Component
      i. Object/Class
      ii. Interface
         Parameter
         Procedural
         Message Passing
   b) System

2. Release
   Phase
   Integration
      Top-down
      Bottom-up
   Release

3. User/Customer
   a) Alpha
   b) Beta
   c) Acceptance

Types
   Stress
   Performance
   Usability
Stage 1: Development Testing

1. Component testing
   – Testing of individual program components;
   – Usually the responsibility of the component developer (except sometimes for critical systems);
   – Tests are derived from the developer’s experience.
   – Type of testing:
     1. Object Class Testing
     2. Interface Testing

2. System testing
   – Testing of 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.
Stage 1.1 : Component / 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 with defined interfaces used to access their functionality.
Stage 1.1.1: 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.
Object/Class Testing Example: Weather station class *(previous discussed case study)*

- Need to define test cases for `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
Object/Class Testing Example: Weather station class (cont.)

- From weather class, create the related state diagram
  - Object have state(s)
  - One state(s) transit from another state(s) triggered by an event happened, certain specific condition and action taken by the object
Stage 1.1.2: Interface testing

- 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.
Stage 1.1.2: Interface testing (cont.)

Types of interface testing:

1. **Parameter** interfaces
   - Data passed from one procedure to another.

2. **Procedural** interfaces
   - Sub-system encapsulates a set of procedures to be called by other sub-systems.

3. **Message passing** interfaces
   - Sub-systems request services from other sub-systems.
Layered architecture - 3 layers

Weather station

- "subsys stem" Interface
- "subsys stem" Data collection
- "subsys stem" Instruments

Manages all external communications
Collects and summarises weather data
Package of instruments for raw data collections
Weather station subsystems

- CommsController
- WeatherStation
- WeatherData
- Instrument Status
- Instruments
- Air thermometer
- RainGauge
- Anemometer
- Ground thermometer
- Barometer
- WindVane
Sub-system interfaces

Sub-system A

Interface objects

Sub-system B
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.
Stage 1.2: System testing

• System testing during development involves integrating components to create a version of the system and then testing the integrated system.
• The focus in system testing is testing the interactions between components.
• System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
• System testing tests the emergent behaviour of a system.
Stage 1.2: System testing *(cont.)*

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
  1. **Integration testing** - the test team have access to the system source code. The system is tested as components are integrated.
  2. **Release testing** - the test team test the complete system to be delivered as a black-box.
- Three types of system testing:
  1. Stress testing
  2. Performance testing
  3. Usability testing
System testing phase 1: Integration testing

- Involves building a system from its components and testing it for problems that arise from component interactions.

1. Top-down integration
   - Develop the skeleton of the system and populate it with components.

2. Bottom-up integration
   - Integrate infrastructure components then add functional components.

- To simplify error localisation, systems should be incrementally integrated.
Stage 1.2.1: Stress testing

• The application is tested against heavy load such as complex numerical values, large number of inputs, large number of queries etc. which checks for the stress/load the applications can withstand.

• Example:
  – Developing software to run cash registers.
  – Non-functional requirement:
    • “The server can handle up to 30 cash registers looking up prices simultaneously.”
  – Stress testing:
    • Occur in a room of 30 actual cash registers running automated test transactions repeatedly for 12 hours.
Stage 1.2.2: Performance testing

• Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.

• Example:
  – Performance Requirement
    • “The price lookup must complete in less than 1 second”
  – Performance testing
    • Evaluates whether the system can look up prices in less than 1 second (even if there are 30 cash registers running simultaneously).
Stage 1.2.3: Usability Testing

• Testing conducted to evaluate the extent to which a user can learn to operate, prepare inputs for and interpret outputs of a system or component.

• Usually done by human-computer interaction specialist that observe humans interacting with the system.
Stages in Software Testing

1. Development
   a) Component
      i. Object/Class
      ii. Interface
         Parameter
         Procedural
         Message Passing
   b) System

2. Release
   Phase
   Integration
      Top-down
      Bottom-up
   Release

3. User/Customer
   a) Alpha
   b) Beta
   c) Acceptance

Types
   Stress
   Performance
   Usability
Stage 2: Release testing

• The process of testing a release of a system that will be distributed to customers.
• Primary goal is to increase the supplier’s confidence that the system meets 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.
Stage 3: User/Customer testing

• User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.

• User testing is essential, even when comprehensive system and release testing have been carried out.
  – The reason for this is that influences from the user’s working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.
Stages in Software Testing

1. Development
   a) Component
      i. Object/Class
      ii. Interface
         Parameter
         Procedural
         Message Passing
   b) System

2. Release
   Phase
   Integration
      Top-down
      Bottom-up
   Release

3. User/Customer
   a) Alpha
   b) Beta
   c) Acceptance

Types
   Stress
   Performance
   Usability
Types of user testing

1. **Alpha** testing
   – Users of the software work with the development team to test the software at the developer’s site.

2. **Beta** testing
   – A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

3. **Acceptance** testing
   – Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.
Stage 3.3: The acceptance testing process
SOFTWARE TESTING : PROCESS
The software testing process

1. Design test cases
2. Prepare test data
3. Run program with test data
4. Compare results to test cases
5. Test cases
6. Test data
7. Test results
8. Test reports
Software Testing Process

1. Test Cases
   a) Requirement-based
      i) Equivalence Partitioning
   b) Black-box
      ii) Boundary Value Analysis
   c) White-box
      i) Basic Path
      ii) Control Structure

2. Test Data

3. Test Results

4. Test Reports

Step 1: draw Flow Graph
Step 2: calculate Cyclomatic Complexity
Step 3: identify Independent Path
Step 4: Generate Test cases
Testing process 1: 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:
  1. **Requirements-based** testing;
  2. **Black-Box** testing;
  3. **White-Box** testing.
Software Testing Process

1. Test Cases
   a) Requirement-based
      i) Equivalence Partitioning
      ii) Boundary Value Analysis
   b) Black-box
      i) Equivalence Partitioning
      ii) Boundary Value Analysis
   c) White-box
      i) Basic Path
      ii) Control Structure

2. Test Data

3. Test Results

4. Test Reports
   Step 1: draw Flow Graph
   Step 2: calculate Cyclomatic Complexity
   Step 3: identify Independent Path
   Step 4: Generate Test cases
Test-case design approach 1: 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.
Requirement \(\Rightarrow\) Test Requirement \(\Rightarrow\) Test Cases \(\Rightarrow\) Test Flows
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 account's permanent storage area.
• 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.
Exercise

• Requirement:
  “The ATM system must allow the customer to do withdrawal transaction, which each withdrawals are allowed only between RM10-RM300 and in RM10 multiple”

1. Derive the Test Requirement(s) - TR
2. Choose a TR, derive a set of Test Cases

<table>
<thead>
<tr>
<th>Case #</th>
<th>Pass/Fail</th>
<th>(Data Value) entered</th>
<th>Expected Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. Test Requirements

• Validate that the withdrawal >300 and <10 is not allowed.
• Validate that the withdrawal of multiple RM10, between RM10-RM300 can be done.
• Validate that the withdrawal option is offered by the ATM.
• Withdrawal of non-multiple RM10 is not allowed.
• Validate that withdrawal is not allowed if the ATM has insufficient money.
• Validate that withdrawal is not allowed is the user has insufficient balance in his account.
Test Cases

• “Validate that a withdrawal of a multiple RM10, between RM10-RM300 can be done”

<table>
<thead>
<tr>
<th>Case #</th>
<th>Pass/Fail</th>
<th>RM entered</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD01</td>
<td>Pass</td>
<td>10</td>
<td>RM10 withdrawn</td>
<td></td>
</tr>
<tr>
<td>WD02</td>
<td>Pass</td>
<td>20</td>
<td>RM20 withdrawn</td>
<td></td>
</tr>
<tr>
<td>WD03</td>
<td>Pass</td>
<td>30</td>
<td>RM30 withdrawn</td>
<td></td>
</tr>
<tr>
<td>WD29</td>
<td>Pass</td>
<td>290</td>
<td>RM290 withdrawn</td>
<td></td>
</tr>
<tr>
<td>WD30</td>
<td>Pass</td>
<td>300</td>
<td>RM300 withdrawn</td>
<td></td>
</tr>
<tr>
<td>WD31</td>
<td>Fail</td>
<td>301</td>
<td>Error Display</td>
<td></td>
</tr>
</tbody>
</table>
Test Flow/Procedure & Script

- **Flow/Procedure:**
  - Step 1: Insert Card
  - Step 2: Enter PIN
  - Step 3: Select Withdraw option
  - Step 4: Enter amount
  - Step 5: Validate amount received

- **Script: (in pseudo-code)**
  - Do until EOF
    - Input data record
    - Send data CARDINFOR to “Card_field”
    - Send data “Enter”

Think Manual!

Think Automated!
Software Testing Process

1. Test Cases
   - a) Requirement-based
     i) Equivalence Partitioning
   - b) Black-box
     ii) Boundary Value Analysis
   - c) White-box
     i) Basic Path
     ii) Control Structure

Step 1: draw Flow Graph
Step 2: calculate Cyclomatic Complexity
Step 3: identify Independent Path
Step 4: Generate Test cases
Test-case design approach 2: Black-Box Testing

• Also called **functional testing** and **behavioral testing**

• Focuses on determining whether or not the program does what it is supposed to do based on its functional requirements.

• Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.
Test-case design approach 2: Black-Box Testing (cont.)

- Takes into account only the input and output of the software without regard to the internal code of the program.
Test-case design approach 2: Black-Box Testing (cont.)

• Strategies:
  1. Equivalence Partitioning
  2. Boundary Value Analysis
Software Testing Process

1. Test Cases
   a) Requirement-based
   i) Equivalence Partitioning
   ii) Boundary Value Analysis
   b) Black-box
   i) Basic Path
   ii) Control Structure

2. Test Data

3. Test Results

4. Test Reports
   c) White-box
   i) Basic Path
   ii) Control Structure

Step 1: draw Flow Graph
Step 2: calculate Cyclomatic Complexity
Step 3: identify Independent Path
Step 4: Generate Test cases
Black-box testing strategies 1: Equivalence Partitioning

• A strategy that can be used to reduce the number of test cases that need to be developed.
• Divides the input domain of a program into classes.
• For each of these equivalence classes, the set of data should be treated the same by the module under test and should produce the same answer.
Black-box testing strategies 1: Equivalence Partitioning (cont.)

• Equivalence classes can be defined by:
  – If an input condition specifies a range or a specific value, one invalid and two valid equivalence classes defined.
  – If an input condition specifies a Boolean or a member of a set, one valid and one invalid
Suppose the specifications for a database product state that the product must be able to handle any number of records from 1 through 16,383.

- **Valid** data: *Range of 1-16383*
- **Invalid** data: *i) less than 1 ii) More than 16383*

Therefore, for this product, there are three equivalence classes:

1. Equivalence class 1: less than one record.
2. Equivalence class 2: from 1 to 16,383 records.
3. Equivalence class 3: more than 16,383 records.

Testing the database product then requires that one test class from each equivalence class be selected.
Software Testing Process

1. Test Cases
   a) Requirement-based
   i) Equivalence Partitioning
   ii) Boundary Value Analysis
   b) Black-box
   i) Basic Path
   ii) Control Structure
   c) White-box

   Step 1: draw Flow Graph
   Step 2: calculate Cyclomatic Complexity
   Step 3: identify Independent Path
   Step 4: Generate Test cases

2. Test Data

3. Test Results

4. Test Reports
Black-box testing strategies 2: Boundary Value Analysis (BVA)

• Large number of errors tend to occur at boundaries of the input domain.

• BVA leads to selection of test cases that exercise boundary values.

• BVA complements equivalence partitioning. Rather than select any element in an equivalence class, select those at the 'edge' of the class.
Black-box testing strategies 2: BVA (cont.)

• When creating BVA test cases, consider the following:
  – If input conditions have a range from a to b (e.g. a=100 and b=300), create test cases:
    • Immediately below a (a-1) → 99
    • At a → 100
    • Immediately above a (a+1) → 101
    • Immediately below b (b-1) → 299
    • At b → 300
    • Immediately above b (b+1) → 301
  – If input conditions specify a number of values n, test with input values:
    • (n-1)
    • n
    • (n+1)
Test-case design approach 3: White-Box Testing

• A verification technique software engineers can use to examine if their code works as expected.
• testing that takes into account the internal mechanism of a system or component (IEEE, 1990).
• Also known as structural testing, glass box testing, clear box testing
Software Testing Process

1. Test Cases
   a) Requirement-based
   b) Black-box
      i) Equivalence Partitioning
      ii) Boundary Value Analysis
   c) White-box
      i) Basic Path
      ii) Control Structure

2. Test Data

3. Test Results

4. Test Reports

- Step 1: draw Flow Graph
- Step 2: calculate Cyclomatic Complexity
- Step 3: identify Independent Path
- Step 4: Generate Test cases
Test-case design approach 3: White-Box Testing (cont.)

- A software engineer can design test cases that:
  - exercise independent paths within a module or unit;
  - Exercise logical decisions on both their true and false side;
  - execute loops at their boundaries and within their operational bounds; and
  - exercise internal data structures to ensure their validity (Pressman, 2001).

- Strategies:
  1. Basic Path Testing / Path Testing
  2. Control Structure Testing
Software Testing Process

1. Test Cases
   a) Requirement-based
   i) Equivalence Partitioning
   ii) Boundary Value Analysis
   b) Black-box
   i) Basic Path
   ii) Control Structure
   c) White-box
      i) Basic Path
      ii) Control Structure

2. Test Data

3. Test Results

4. Test Reports

Step 1: draw Flow Graph
Step 2: calculate Cyclomatic Complexity
Step 3: identify Independent Path
Step 4: Generate Test cases
White-box testing strategies 1: Basis Path Testing

• The basis path method allows for the construction of test cases that are guaranteed to execute every statement in the program at least once.

• This method can be applied to detailed procedural design or source code.

• Steps:
  1. Draw the flow graph corresponding to the procedural design or code.
  2. Determine the cyclomatic complexity of the flow graph.
  3. Determine the basis set of independent paths. (The cyclomatic complexity indicates the number of paths required.)
  4. Determine a test case that will force the execution of each path.
Basic Path Testing Steps 1: Draw Flow Graph

• On a flow graph:
  – Arrows called edges represent flow of control
  – Circles called nodes represent one or more actions.
  – Areas bounded by edges and nodes called regions.
  – A predicate node is a node containing a condition
How to Derive Flow Graph - if

S1
If (e)
S2
Endif
S3
How to Derive Flow Graph – if-else

S1
If (e)
S2
Else
S3
Endif
S4

Diagram:
- Node S1
- Node S2
- Node S3
- Node S4
- Edge from S1 to S2: True
- Edge from S1 to S3: False
- Edge from S2 to S4
- Edge from S3 to S4
How to Derive Flow Graph – boolean-AND

If (a AND b)

C

End If

D
How to Derive Flow Graph – boolean-OR

If (a OR b)
   C
End If

D

Diagram:

- Node 'a OR b' with edges to 'c' and 'd' labeled 'True' and 'False', respectively.
- Node 'c' with an edge to 'd' labeled 'False'.
- Node 'd'.
How to Derive Flow Graph – while

S1
While (e)
S2
End while
S3

Diagram:

- Node S1
  - Edge to S2
  - Edge to while
- Node while
  - True edge to S2
  - False edge to S3
- Node S2
- Node S3
How to Derive Flow Graph – for

S1
for(int i=0;i>5;i++)
S2
End for
S3
Exercise

• Given a program source code below, identify the suitable test cases to be enforced in white-box testing for the program based on EP and BVA black-box testing:

```c
int main()
{
    int i, n, t;
    printf("n= ");
    scanf("%d ", &n);
    
    if(n<0)
    {
        printf("invalid: %d\n",n);
        n=-1;
    }
    else
    {
        t=1;
        for(i=1;i<=n;i++)
        {
            t*=i;
        }
        printf("%d! = %d\n",n, t);
    }
    return 0;
}
```
int main()
{
    int i, n, t;
    printf("n= ");
    scanf("%d ", &n);
    if(n<0)
    {
        printf("invalid: %d\n", n);
        n=-1;
    }
    else
    {
        t=1;
        for(i=1;i<=n;i++)
        {
            t*=i;
        }
        printf("%d! = %d\n", n, t);
    }
    return 0;
}
simplifies
Basic Path Testing Steps 2: Determine Cyclomatic Complexity

• Gives a quantitative measure of the logical complexity.
• This value gives the number of independent paths in the basis set, and an upper bound for the number of tests to ensure that each statement is executed at least once.
• An independent path is any path through a program that introduces at least one new set of processing statements or a new condition (i.e., a new edge)

• 3 Formula:
  1. \( V(G) = \#\text{Edges} - \#\text{Nodes} + 2 \)
  2. \( V(G) = \#\text{Predicate Nodes} + 1 \)
  3. \( V(G) = \#\text{Region} \)
Basic Path Testing Steps 2: Determine Cyclomatic Complexity

• Using 3 formulas (*either 1*):
  1. \( V(G) = \text{#Edges} - \text{#Nodes} + 2 \)
    \[ = 10 - 9 + 2 \]
    \[ = 3 \]
  2. \( V(G) = \text{#Predicate Nodes} + 1 \)
    \[ = 2 + 1 \]
    \[ = 3 \]
  3. \( V(G) = \text{#Region} \)
    \[ = 3 \]
Basic Path Testing Steps 3: Determine Independent Path

- Independent Path:
  i. 1-2-3-9
  ii. 1-2-4-5-6-7-5-8-9
  iii. 1-2-4-5-8-9
Basic Path Testing Steps 4: Determine the Test Cases

- Equivalence partitioning (EP) and boundary value analysis (BVA) provide a strategy for writing white-box test cases.

- Input values based on EP: there are three equivalence classes:
  1. Equivalence class 1: \( n \) less than zero. \( n < 0 \)
  2. Equivalence class 2: \( n \) equals to zero. \( n == 0 \)
  3. Equivalence class 3: \( n \) is more than zero. \( n > 0 \)

- Input values (Identified by BVA): -1, 0, 1
Basis Path Testing Step 4: Prepare the Test Cases

To complete the white-box testing process, you would test your program with all three input sets, and verify that the output matches the expected result. If all outputs match, then the code passes the test.

<table>
<thead>
<tr>
<th>Independent Path</th>
<th>Test Cases</th>
<th>Expected Results/Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2-3-9</td>
<td>n = -1</td>
<td>“Invalid, 1”</td>
</tr>
<tr>
<td>1-2-4-5-6-7-5-8-9</td>
<td>n = 0</td>
<td>“Invalid, 1”</td>
</tr>
<tr>
<td>1-2-4-5-8-9</td>
<td>n = 1</td>
<td>1, 1</td>
</tr>
</tbody>
</table>
Software testing process and techniques (summary)

1. Requirement-based Testing
2. Black Box Testing
3. White Box Testing

- Equivalence Partitioning
- Test cases
- Test data
- Run program with test data
- Compare results to test cases
- Test reports

Boundary Value Analysis