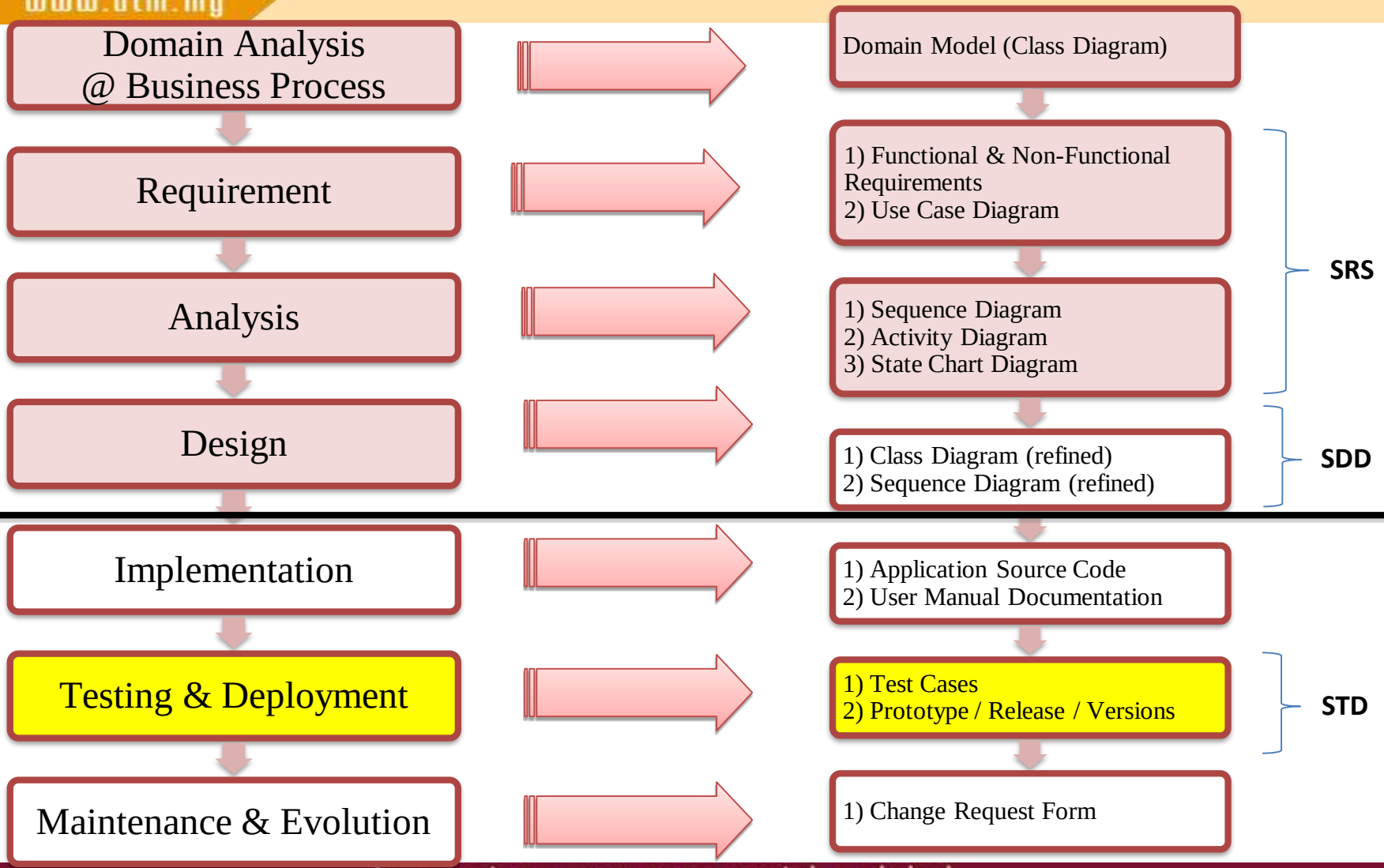
Module 8: Software Verification, Validation and Testing

Software Engineering

Faculty of Computing
Universiti Teknologi Malaysia

Recap on SDLC Phases & Artefacts



Outline

- Verification and Validation (V&V): Definition, goal, techniques and purposes
- Inspection vs. testing: Complement to each other
- Software testing:
  - Definition, goal, techniques and purposes
  - Stages: development, release, user/customer
  - Process: test cases, test data, test results, test reports
  - Focus in designing **test cases** to perform testing

Objectives

- To discuss V&V differences and techniques
- To describe different types of testing and its definition
- To learn three strategies for generating system test cases

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**

Verification vs. Validation

Verification	Validation
The verifying process includes checking documents, design, code, and program	It is a dynamic mechanism of testing and validating the actual product
It does ***not*** involve executing the code	It always involves executing the code
Verification uses methods like reviews, walkthroughs, inspections, prototyping etc.	It uses methods like Black Box Testing, White Box Testing, and non-functional testing, prototyping etc.
Whether the software conforms to specification is checked	It checks whether the software meets the requirements and expectations of a customer
Target is application and software architecture, specification, complete design, high level, and database design etc.	Target is an actual product
It comes before validation	It comes after verification

V&V Goal

- V&V should establish confidence that the software is **fit for purpose**
- This does **NOT** mean completely free of defects
- It must be **good enough** for its intended use and the type of use will determine the **degree of confidence** that is needed

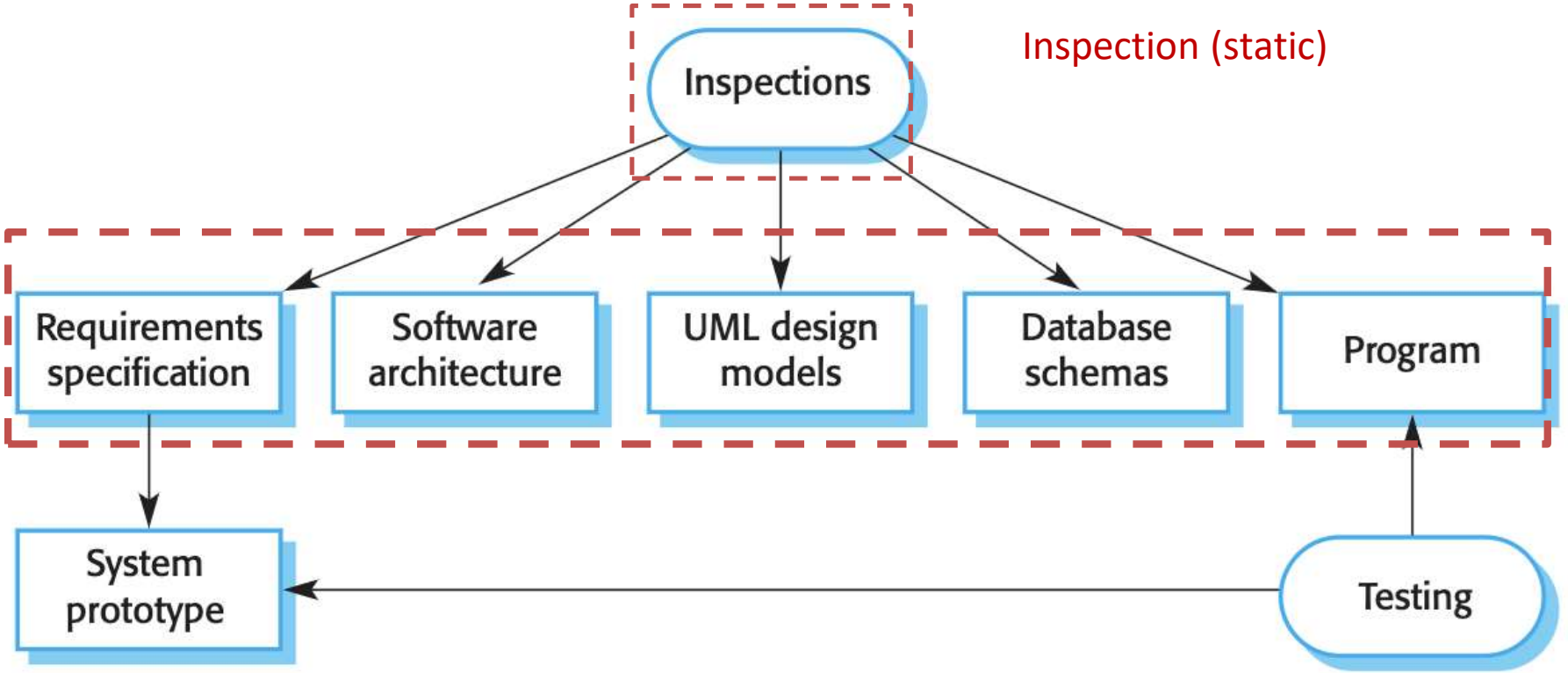
V&V: Degree-of-Confidence Category

- **Software function/purpose**
  - The level of confidence depends on **how critical the software** is to an organisation e.g. safety-critical system
- **User expectations**
  - Users may have **low expectations** of certain kinds of software (user previous experience, unreliable software especially newly installed software)
- **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)

Verification Technique: Software Inspection

- The technique involves **people examining the source representation** with the aim of discovering anomalies (deviation from standard/expectation) and defects (errors)
- It **does not require execution** of a system – static test, so it may be used before implementation
- They may be applied to any **representation of the system** (e.g. requirements, design, test data)
- It is an effective technique for discovering program errors

Software Inspection: Static Testing



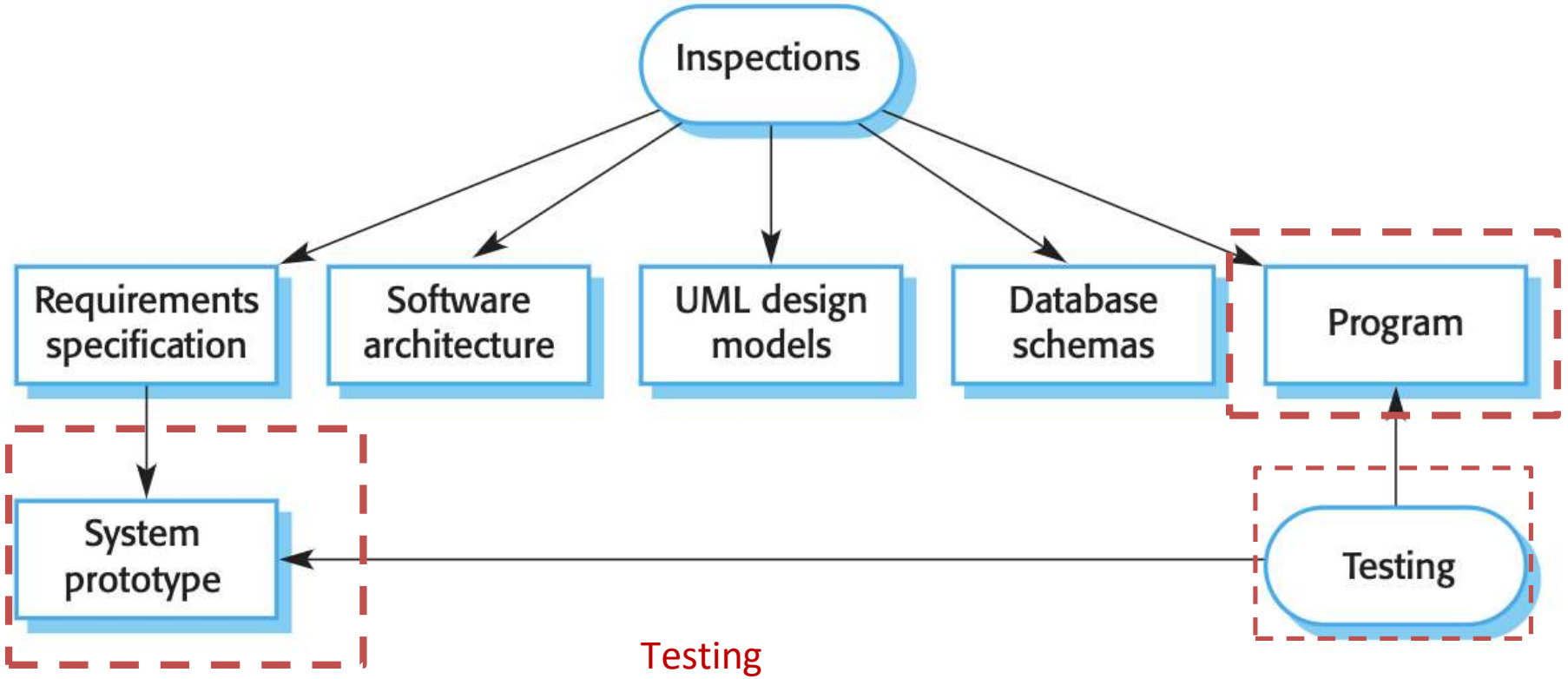
Advantages of Inspections

- During testing, errors can mask (hide) other errors. As inspection is a **static process**, it **need not to be concerned with interactions** between errors.
- Incomplete versions of a system can be **inspected without additional costs**. Compared to dynamic testing, if a program is incomplete, need to develop specialized test that is connected in order to test the parts that are available.
- Besides searching for program defects, an inspection can also **consider broader quality attributes of a program**, such as compliance with standards, portability and maintainability (inefficiencies, inappropriate algorithms, poor programming style which make a system to be difficult to maintain & update).

Validation Technique: Dynamic Testing

- Validation involves evaluating the software **system to ensure that it meets the intended requirements** and satisfies the user's needs and expectations.
- Validation **requires the execution of the software in real or simulated environments** to validate its functionality, usability, and performance.
- This includes **interacting with the software, providing inputs, and evaluating the outputs** to confirm that they align with the expected results and meet the user's requirements.
- Validation also includes **assessing non-functional aspects such as scalability, security, and reliability** to ensure that the software meets the desired performance and quality standards. **Compliance with industry regulations and standards** may also be evaluated during the validation process.

Testing (Dynamic)



Testing (dynamic)

Inspections vs. Dynamic Testing?

- Software **inspection** and review concerned with checking and analysis of the static system representation to discover problems ("static test": **no execution** needed)
  - May be supported by tool-based document and code analysis ("static analysis")
- Software **testing** concerned with exercising and observing product behaviour ("dynamic test": **needs execution**)
  - The system is executed with test data and its operational behaviour is observed

Inspections vs. Dynamic Testing?

- Inspections and dynamic testing are V&V 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.

Recap on Software (Dynamic) Testing

- Software (dynamic) testing is concerned with exercising and observing **product behaviour**
- Dynamic test: The system is executed with test data and its operational behaviour is observed
- "Testing can only show the presence of errors, not their absence" (Dijkstra et al. 1972)

V&V Technique: Prototyping

- "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]
- Prototyping can be **used as a form of verification** by verifying requirements, verifying design choices, validating functionality, and gathering user feedback. It allows stakeholders to visualize and interact with a working model of the software, enabling early identification of errors and issues before full-scale development.
- While prototyping serves as a valuable verification technique, it should be supplemented with other verification activities, such as inspections, reviews, and testing, to ensure that the software meets the specified requirements and quality standards.

V&V Technique: Prototyping

- Prototyping is **used for validation** in software development by providing stakeholders with a tangible representation of the software system. Through prototypes, requirements can be validated, design choices can be assessed, and user feedback can be gathered and incorporated.

- Prototypes allow for testing and validation of the system's functionality, user experience, and alignment with user needs and expectations. By using prototyping for validation, software engineers can identify and address issues early in the development process, ensuring that the final product meets the desired objectives and delivers value to the end users.

V&V Technique: Model Analysis

- Verification:
  - Is the model **well formed**?
  - Are the parts of the model **consistent** with one another?
- 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 Technique: Model Analysis Example in Basic Cross-Check for UML Model

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?

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?

StateChart Diagrams

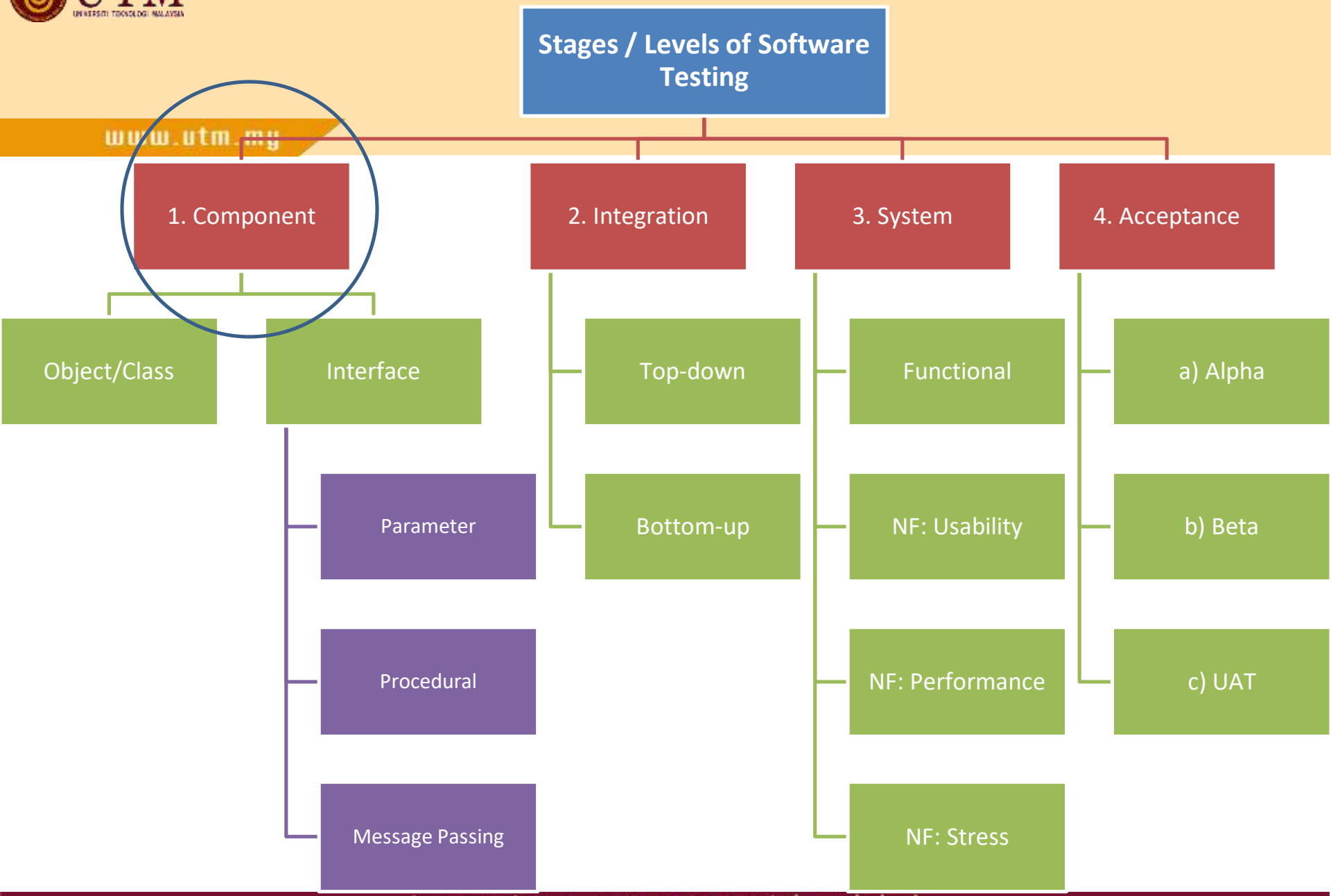
- Does each statechart diagram capture (the states of) a single class?
  - Is that class in the class diagram?
- Does each transition have a trigger event?
  - Is it clear which object initiates each event?
  - Is each event listed as an operation for that object's class in the class diagram?
- Does 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?

Stages of Testing

Commercial software requires **4 stages** of testing:

1. **Component** test: Individual software components are tested during development to discover bugs and defects
2. **Integration** test: Testing is performed between integrated components during development
3. **System** test: Separate testing team tests a complete version of the system before it is released to users
4. **Acceptance** test: User/customers/authorised entity determine whether to accept the system through formal testing, which is conducted to check whether the system satisfies the acceptance criteria

Rex Black, Erik Van Veenendaal, Dorothy Graham (2012), *Foundations of Software Testing - ISTQB® Certification*, 3rd ed., Cengage Learning


Stages / Levels of Software Testing
1. Component
2. Integration
3. System
4. Acceptance
Object/Class
Interface
Top-down
Bottom-up
Functional
NF: Usability
NF: Performance
NF: Stress
a) Alpha
b) Beta
c) UAT
Parameter
Procedural
Message Passing

Stage 1: Component Test

- 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
- Usually, it is the responsibility of the **component developer** (except sometimes for critical systems)
- Tests are derived from the developer's experience
- Types of testing: (i) Object Class Testing (ii) Interface Testing

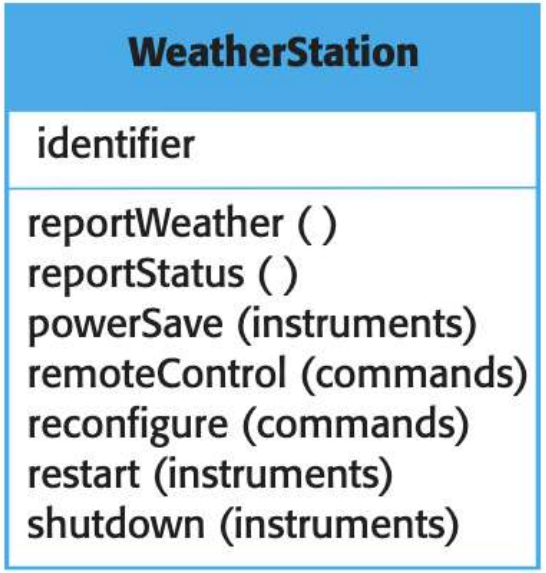
Component Test: Object Class Testing

- 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

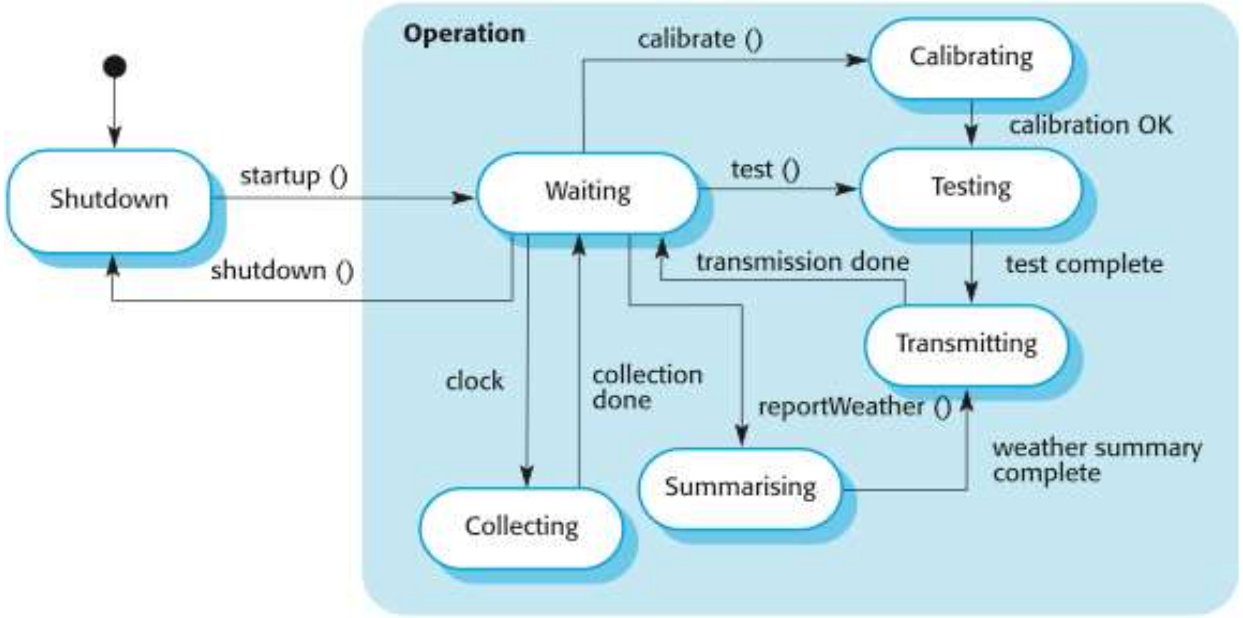
- Need to define test cases for reportWeather, calibrate, test, startup and shutdown
- Using a state model, identify **sequences of state transitions** to be tested and the event sequences to cause these transitions
- Example:

Waiting → Calibrating → Testing →
Transmitting → Waiting



Object/Class Testing Example: Weather Station Class (Cont.)

- From weather class, create the related state diagram:
  - Objects have state(s)
  - One state transits from another state(s) that is triggered by an event that occurs, certain specific condition and action taken by the object



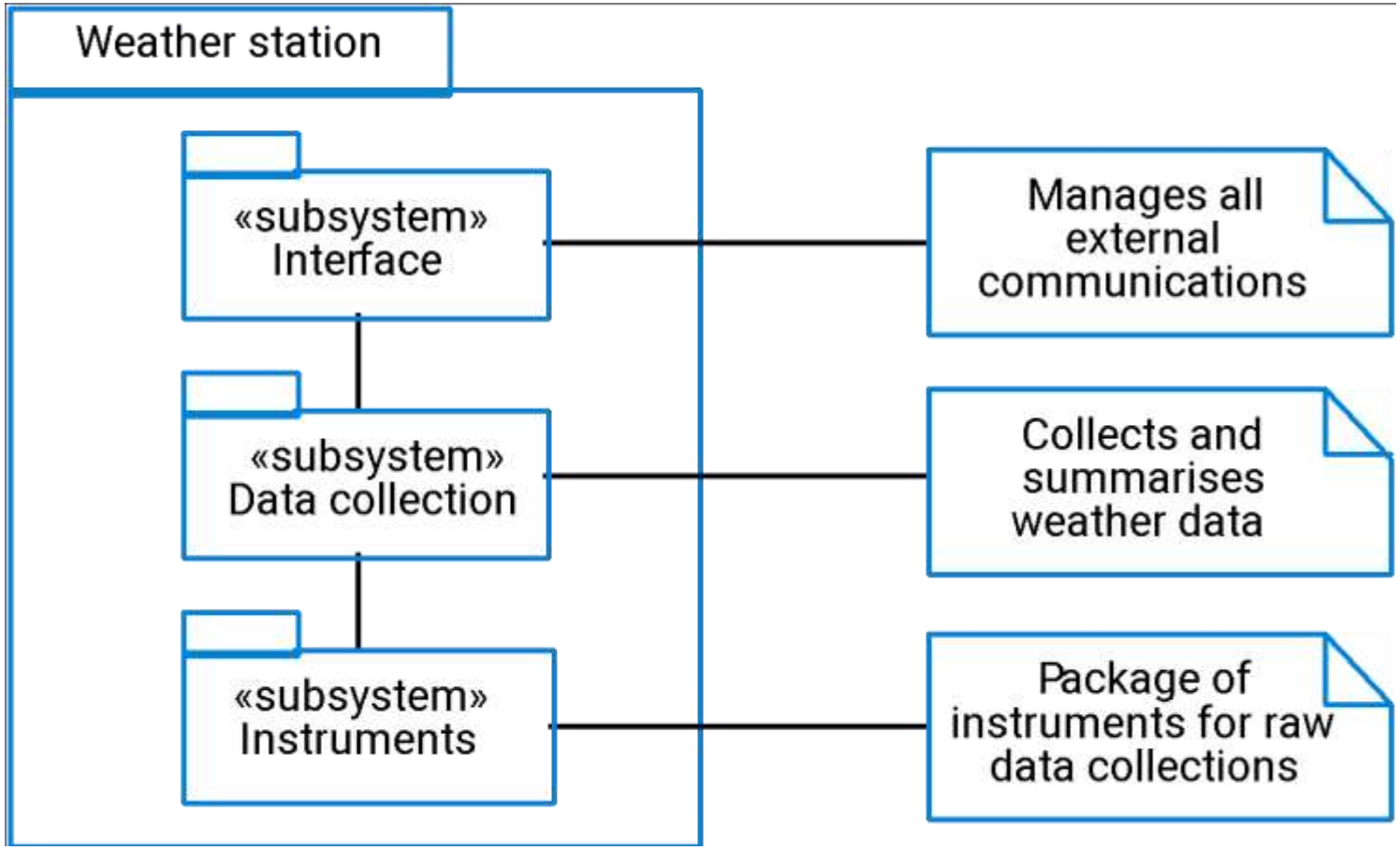
Component Test: Interface Testing

- 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

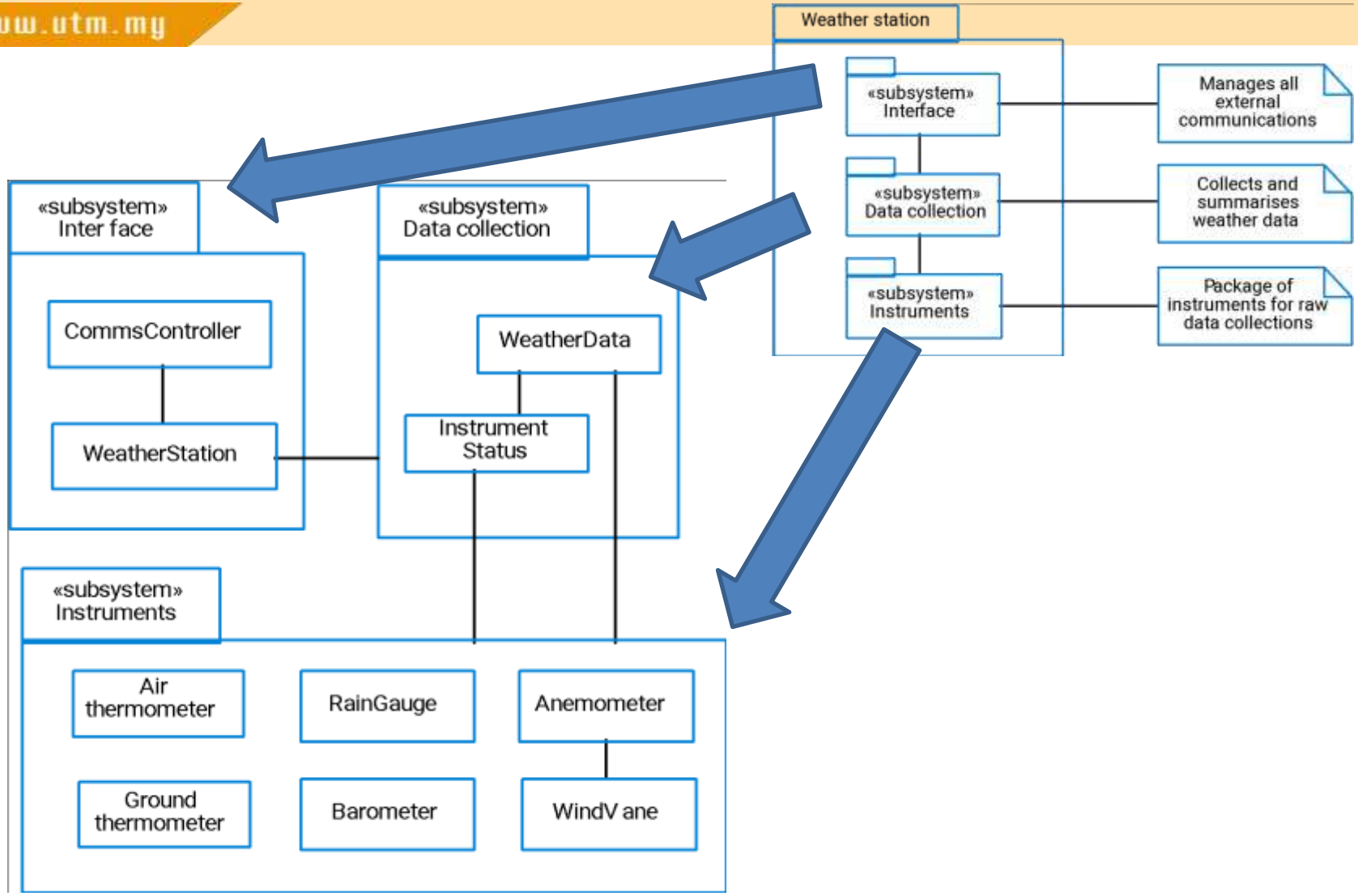
Interface Testing Types

- Parameter interface
  - Data passed from one procedure to another
- Procedural interface
  - Sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interface
  - Sub-systems request services from other sub-systems

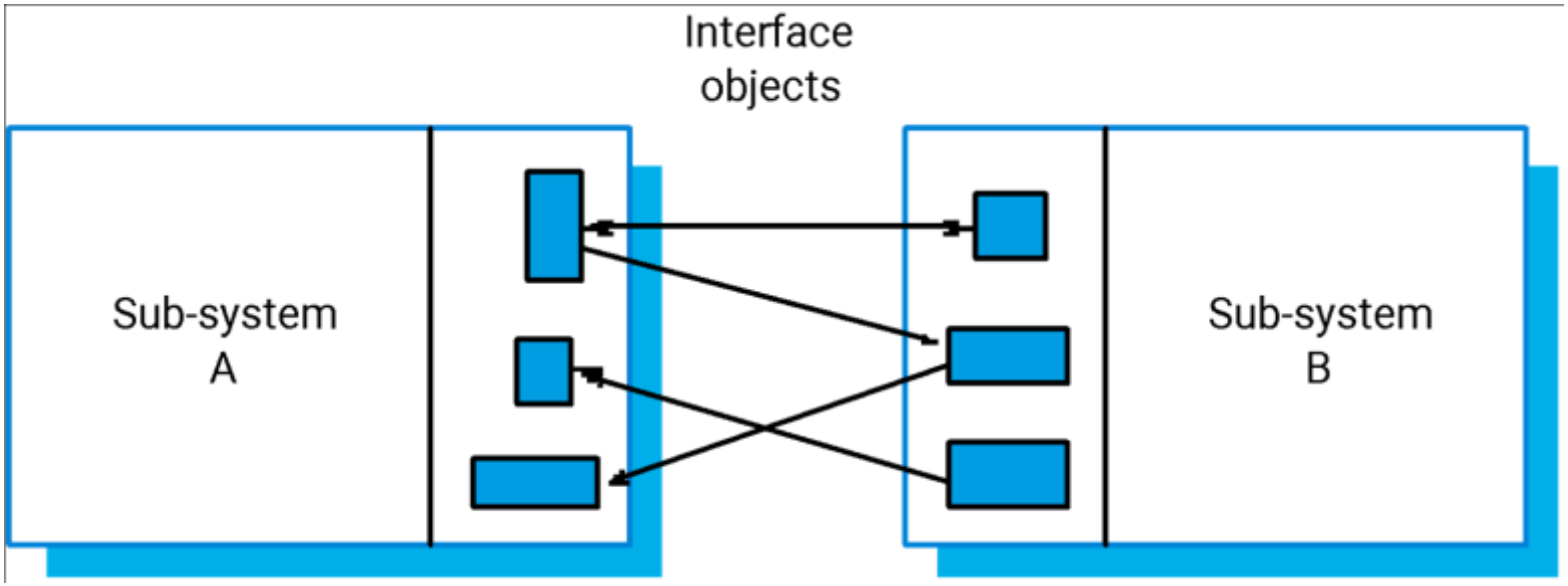
Layered Architecture - 3 Layers



Weather Station Subsystems

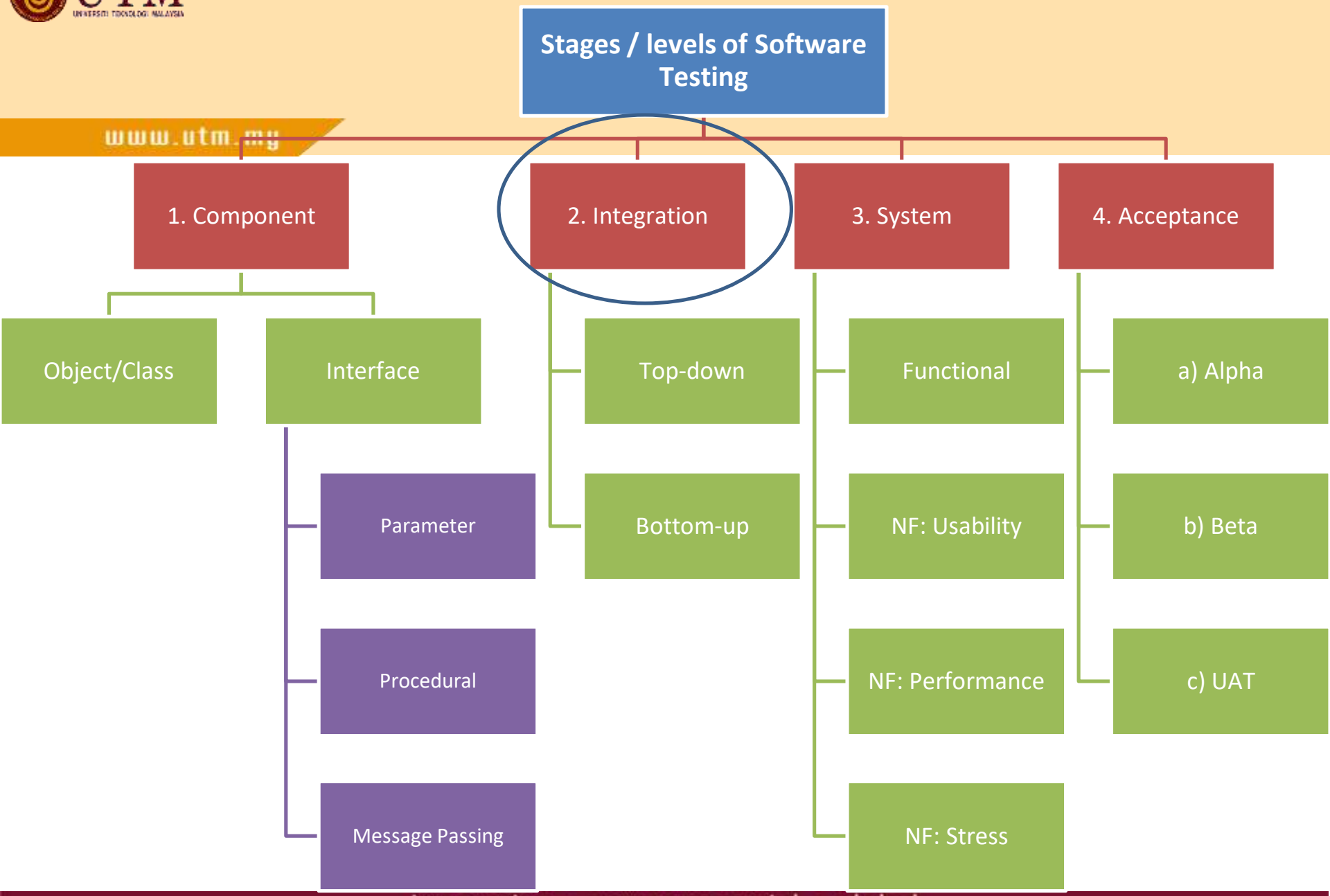


Sub-System Interfaces



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


Stages / levels of Software Testing
1. Component
2. Integration
3. System
4. Acceptance
Object/Class
Interface
Top-down
Functional
a) Alpha
Parameter
Bottom-up
NF: Usability
b) Beta
Procedural
NF: Performance
c) UAT
Message Passing
NF: Stress

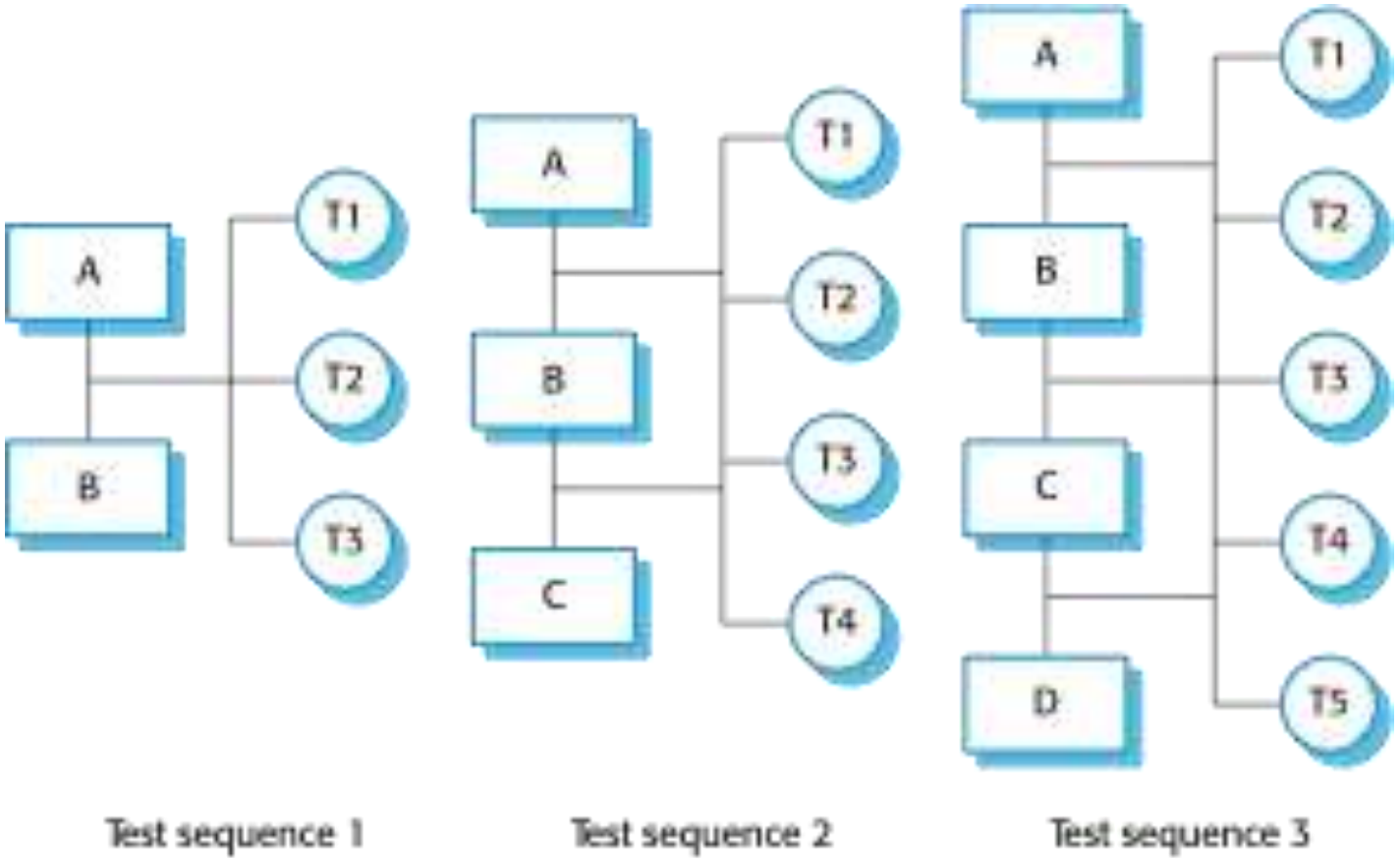
Stage 2: Integration Test

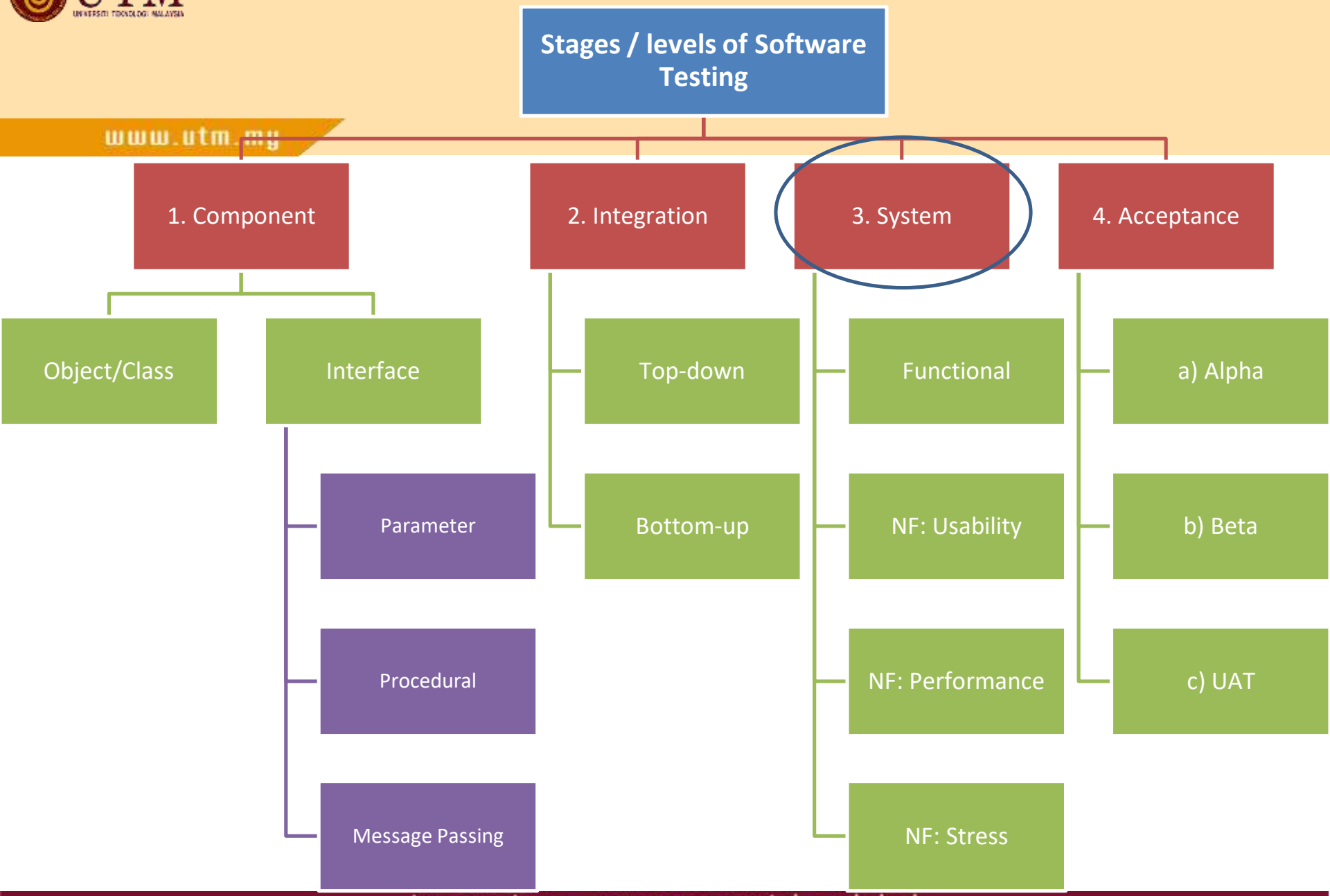
- Testing during development involves **integrating components** to create a version of the system and then testing the integrated system
- The test team has access to the system source code and the system is tested as integrated components
- The focus in integration test is **testing the interactions** between components
- Integration test checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces

Integration Test (cont.)

- 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**

Incremental Integration Testing




Stages / levels of Software Testing
1. Component
2. Integration
3. System
4. Acceptance
Object/Class
Interface
Top-down
Bottom-up
Functional
NF: Usability
NF: Performance
NF: Stress
a) Alpha
b) Beta
c) UAT
Parameter
Procedural
Message Passing

Stage 3: System Test

- Process of testing an integrated system to verify that it meets specified requirements
- The test team (preferably independent testing team) **test the complete system** to be delivered as a black-box
- Includes function test, function interaction (flow) and non-functional attributes
- Three examples of system testing to test **non-functional** attributes:
  1. Stress testing
  2. Performance testing
  3. Usability testing

System Test (cont.)

- The process of testing is a release of a system that will be distributed to customers
- Primary goal is to increase suppliers' confidence that the system meets its requirements
- System test is usually **black-box or functional testing**:
  - Based on the system specification only
  - Testers do not have knowledge of the system implementation
  - (see requirements test section later)

Stress Testing

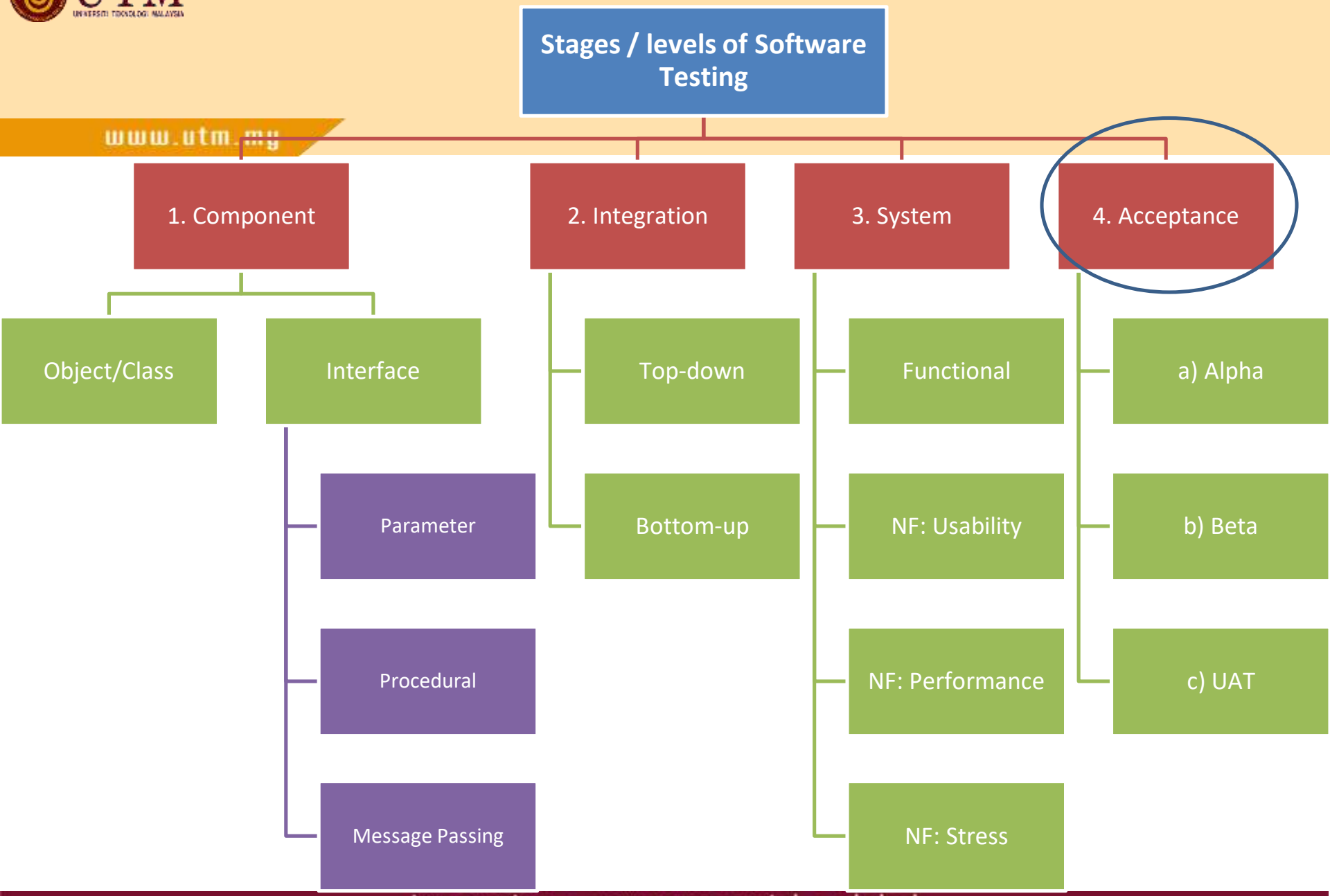
- The application is tested against heavy load e.g. complex numerical values, large number of inputs, large number of queries etc. which **checks for the stress/load** the applications can withstand
- Example: Developing software to run cash registers
  - Non-functional requirement:
    - "The server 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

Performance Testing

- Part of **release testing** may involve testing the emergent properties of a system e.g. 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)

Usability Testing

- Testing is 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 / levels of Software Testing
1. Component
2. Integration
3. System
4. Acceptance
Object/Class
Interface
Top-down
Functional
a) Alpha
Parameter
Bottom-up
NF: Usability
b) Beta
Procedural
NF: Performance
c) UAT
Message Passing
NF: Stress

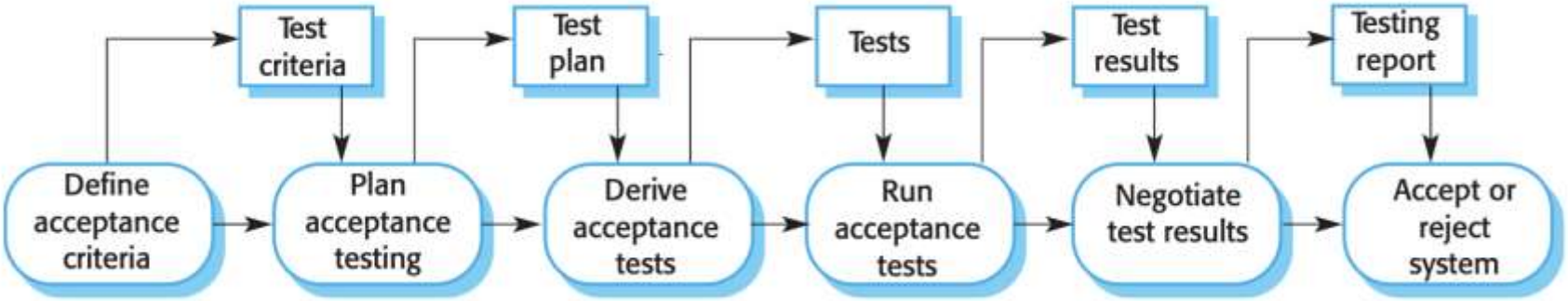
Stage 4: Acceptance Test

- Formal testing with respect to user needs, requirements, and business processes conducted to **determine whether or not a system satisfies the acceptance criteria** and to enable the user, customers or other authorized entity to determine whether or not to accept the system
- 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
- Goal is to **establish confidence in the system/part-system** or specific non-functional characteristics (e.g. performance), usually for ensuring the system is **ready for deployment** into production

Acceptance Test (cont.)

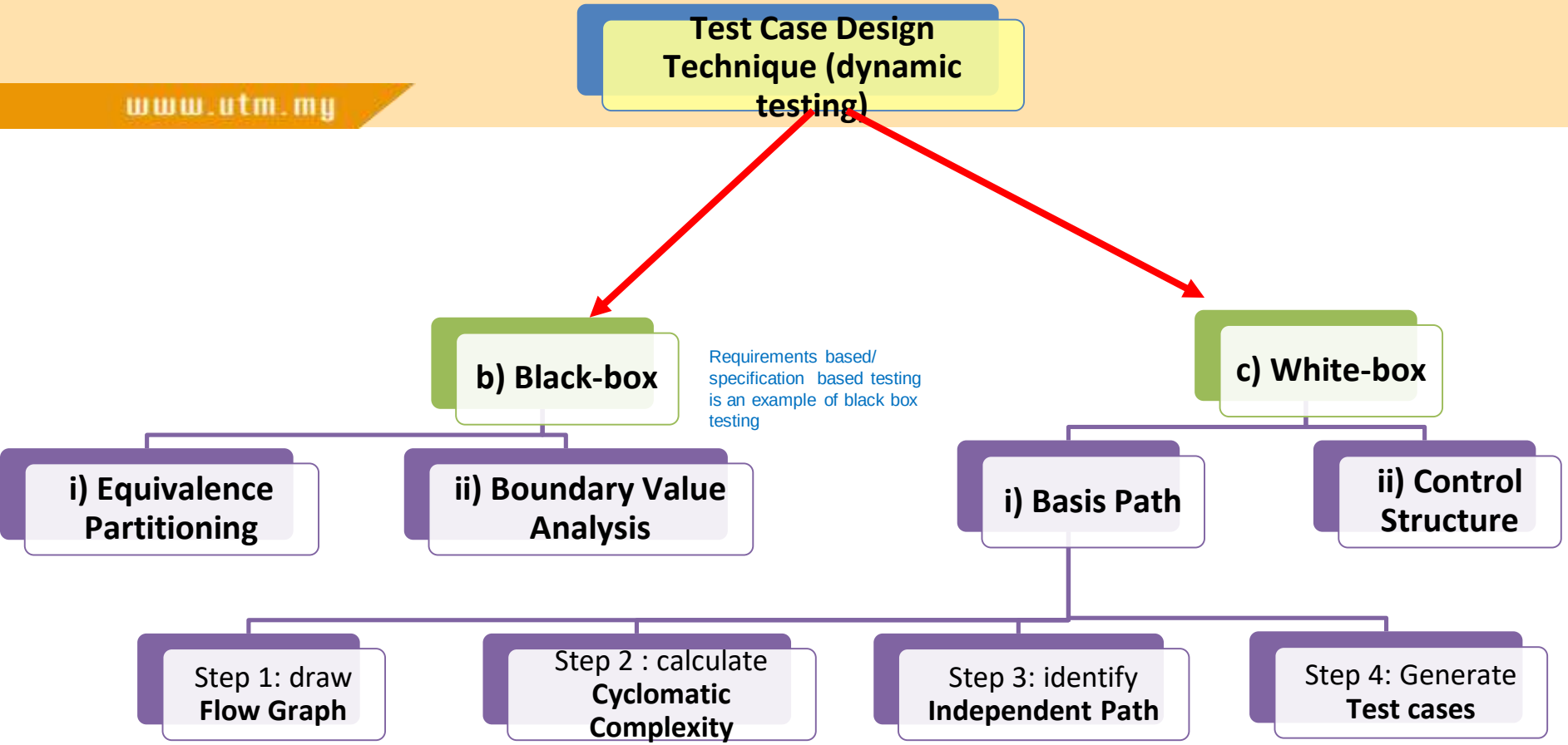
- Acceptance test is a stage in the testing process in which users or customers provide input and advice on system testing
- Acceptance test is essential, even when comprehensive system testing have been carried out
- The reason for this is that **influences from users' working environment have a major effect** on the reliability, performance, usability and robustness of a system
- These cannot be replicated in a testing environment

Acceptance Testing Process



Types of Acceptance Test

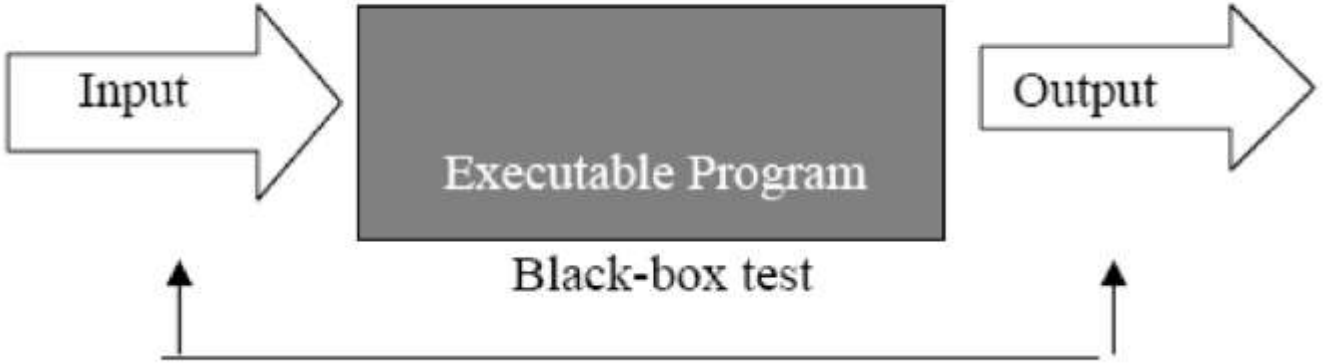
- Alpha testing
  - Users of the software work with the development team to test the software **at the developer's site**
  - In-house test
- Beta testing
  - Early testing of stable product by customers / users
  - A release of the software is **made available to users to allow them to experiment at their site** and to raise problems that they discover with the system developers
  - published reviews of beta release test results can make or break a product (e.g. PC games)
- User Acceptance Testing (UAT)
  - Conducted by or visible to the end user & customer at the **final stage of validation**
  - Testing based on the defined user requirements which often uses the 'thread testing' approach – the same ones for functional system test


Test Case Design Technique (dynamic testing)
b) Black-box
Requirements based/ specification based testing is an example of black box testing
c) White-box
i) Equivalence Partitioning
ii) Boundary Value Analysis
i) Basis 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 Technique: 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

Black-Box Testing (Cont.)



Takes into account only the input and output of the software **without any regard to the internal code** of the program

Test Case Design

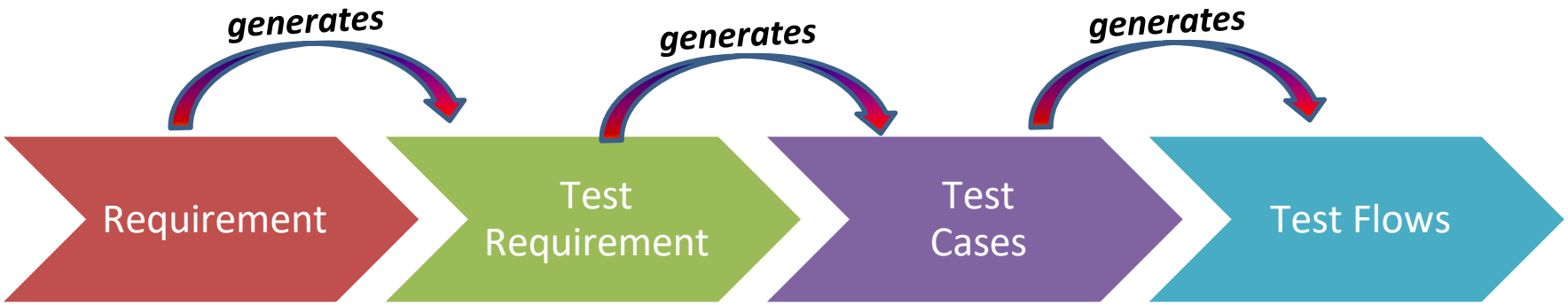
- Involves designing the test cases (**inputs and outputs**) used to test the system
- The goal of test case design is to create a set of tests that are effective in validation and defect testing
- Test case design **approaches for dynamic testing**:
  1. Black-Box testing (e.g requirements based testing)
  2. White-Box testing

What is Requirements-Based Testing?

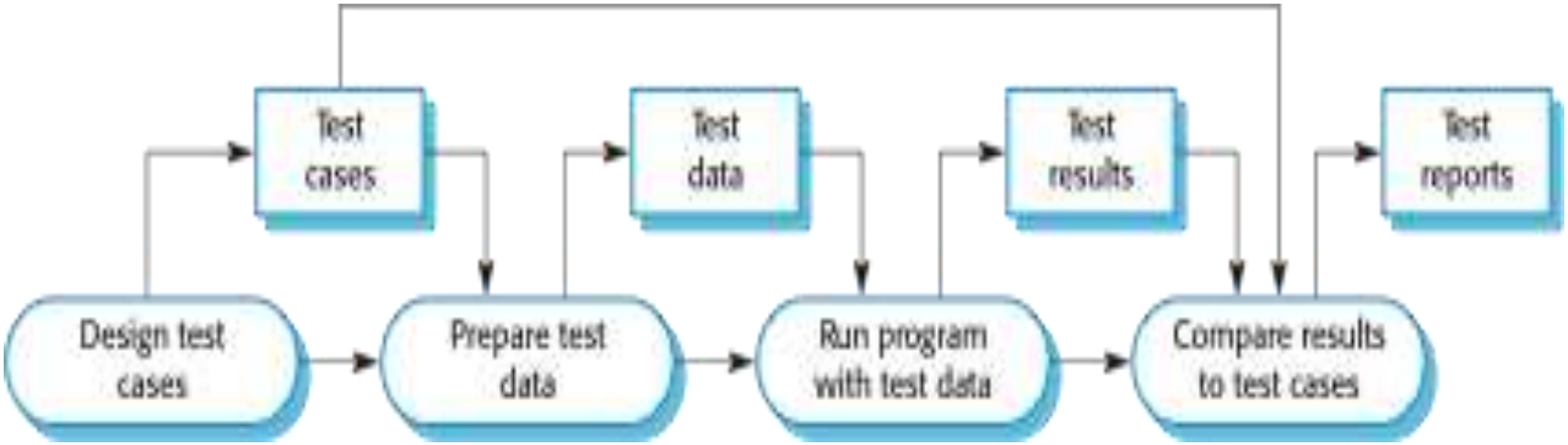
- **Requirements-Based Testing** is a type of **Black-Box Testing**.
- **Definition:** Requirements-based testing involves designing test cases directly from the documented requirements of a system **(SRS)**.
- **Focus:** It does not consider the internal structure or implementation of the software. Instead, it verifies that the output aligns with the expected results based on the inputs specified in the requirements.

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



Software Testing Process



Step 1: Collect Requirements

Example requirements from ATM system:

- The ATM system must allow the customer to do withdrawal transactions, which each withdrawal is allowed only between RM10-RM300 and in RM10 multiple
- The ATM system must provide the customer with the ability to check their account balance, displaying the current available balance on the screen.
- The ATM system should allow customers to change their Personal Identification Number (PIN) securely.
- The ATM system should provide customers with the ability to request a mini statement, which displays the most recent transactions on their account, including withdrawals, deposits, and transfers.
- The ATM system should have a mechanism to retain the customer's card temporarily in case of multiple invalid PIN attempts or suspicious activity, ensuring the security of the account.

Step 2: Generate 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 if the user has insufficient balance in his account

Step 3: Design Test Case based on Test Requirements

- Select requirement:
  "The ATM system must allow the customer to do withdrawal transaction, which each withdrawal is allowed only between RM10-RM300 and in RM10 multiple"

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

When designing the test cases, the Actual Results & Pass / Fail columns are left empty. These columns are filled in as tests are executed.
*Pass when expected = actual

Case #	(Data Value) entered	Expected Results	Actual Results	Pass/Fail

Step 4: List out all Test Cases

"Validate that a withdrawal of a multiple RM10, between RM10-RM300 can be done"

TC ID	RM entered	Expected Results	Actual Results	Pass/Fail
WD01	10	RM10 withdrawn		
WD02	20	RM20 withdrawn		
WD03	30	RM30 withdrawn		
:				
WD29	290	RM290 withdrawn		
WD30	300	RM300 withdrawn		
WD31	**301**	**Error Display**		

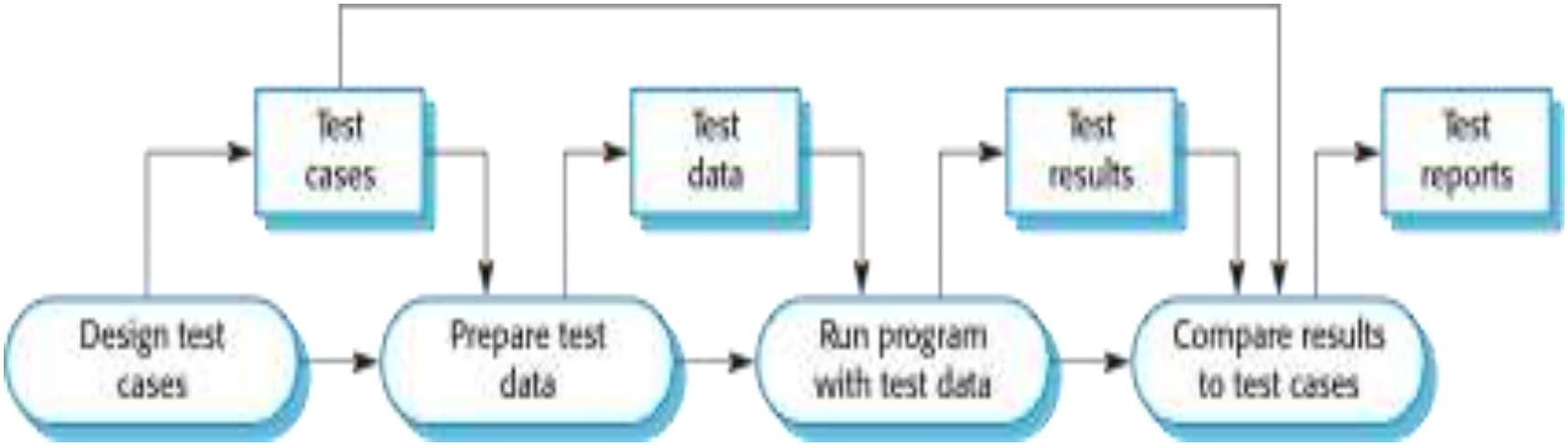
For WD31, if Error Display is expected (Expected Result), and what you get for Actual result is the same, i.e. Error Display, would this test be considered as Pass? Or Fail?

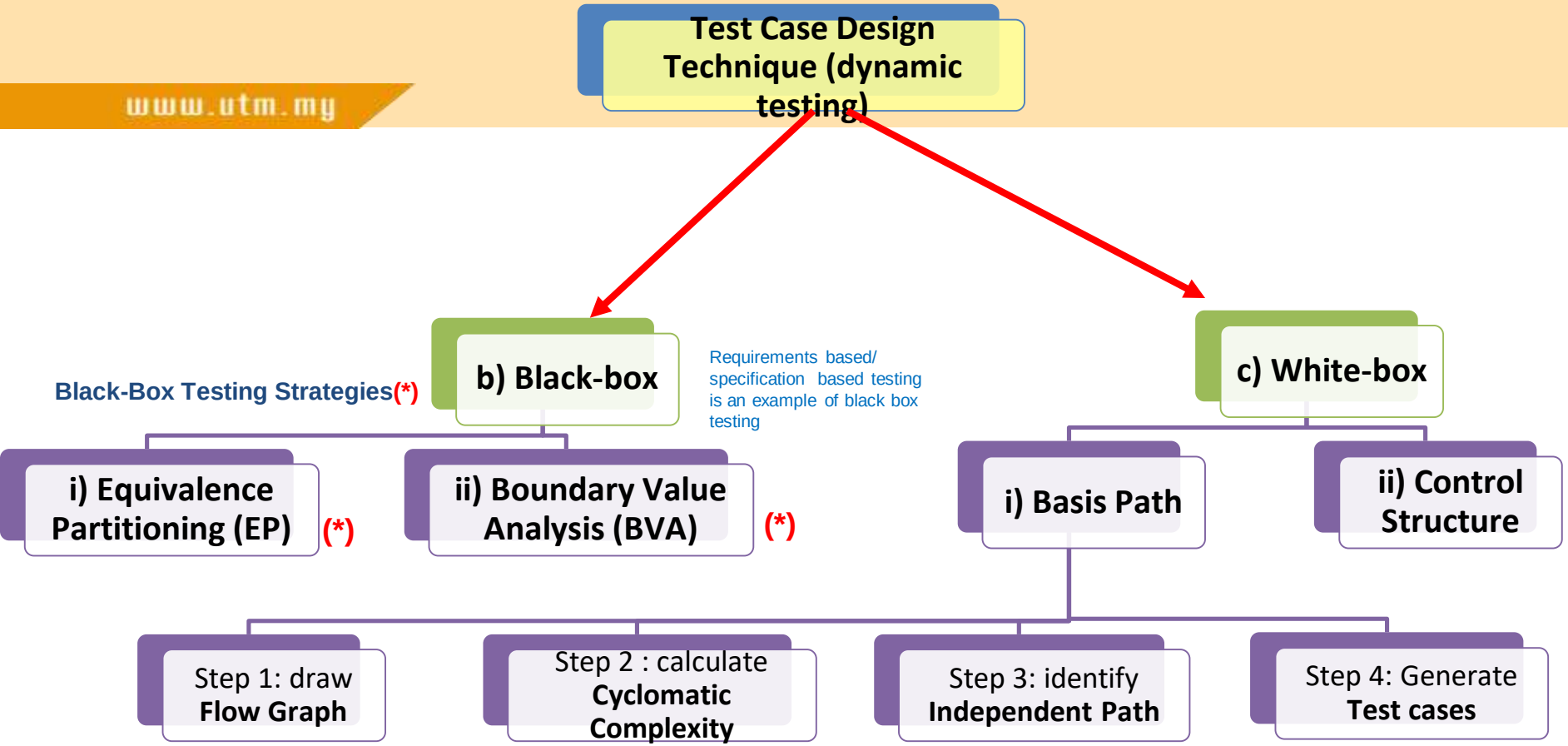
Difference Test Scenario/ Steps and Script

- **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"
    - :
    - :
    - :

Software Testing Process




Test Case Design Technique (dynamic testing)
b) Black-box
Black-Box Testing Strategies(*)
Requirements based/ specification based testing is an example of black box testing
c) White-box
i) Equivalence Partitioning (EP)
(*)
ii) Boundary Value Analysis (BVA)
(*)
i) Basis 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

- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)

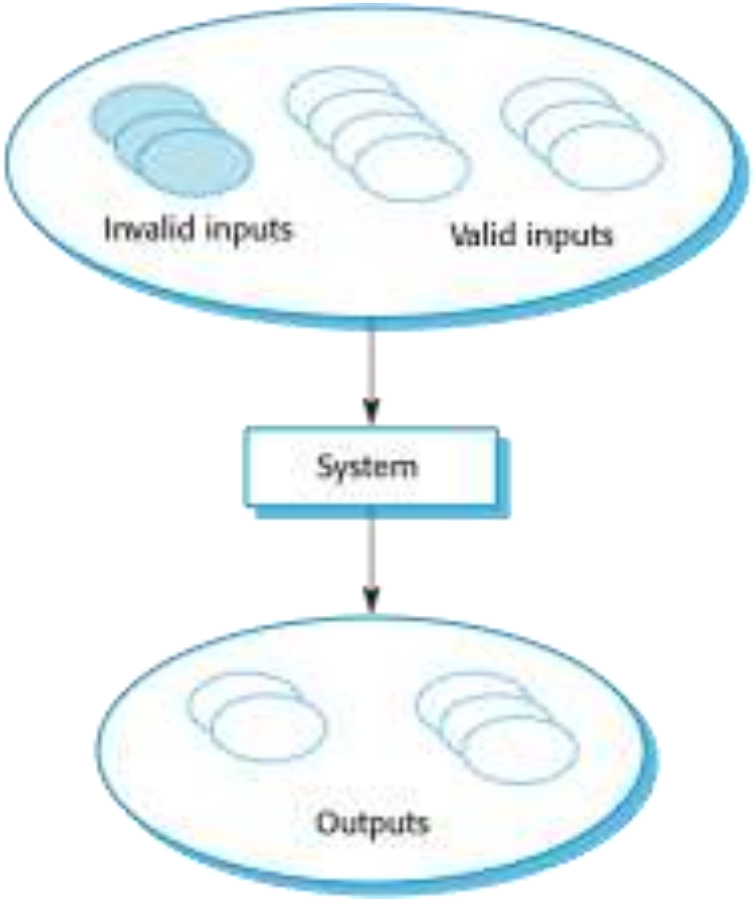
Black-Box Testing Strategies 1: Equivalence Partitioning

- 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:

- If an input condition specifies a **range** or a specific value, one valid and two invalid equivalence classes defined
- If an input condition specifies a **Boolean** or a member of a set, one valid and one invalid



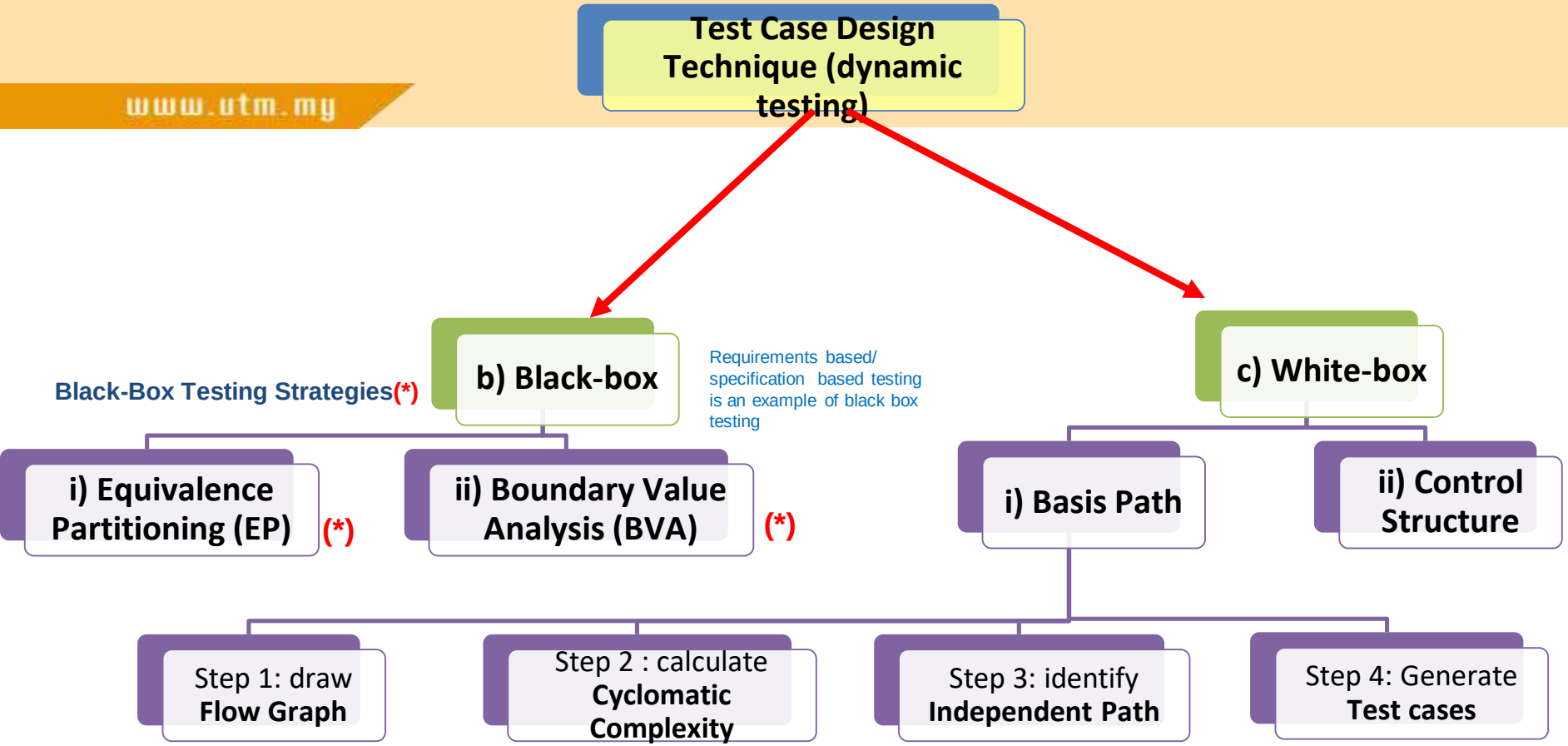
Black-Box Testing Strategies 1: Equivalence Partitioning (Cont.)

- 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 to 16383
- Invalid data: i) less than 1 ii) More than 16383
- For this product, there are 3 equivalence classes:
  1. **Equivalence class 1: less than one record (invalid)**
  2. **Equivalence class 2: from 1 to 16,383 records (valid)**
  3. **Equivalence class 3: more than 16,383 records (invalid)**
- Testing the database product then requires that one test class from each equivalence class be selected

Test Cases

Equivalence Class	Status	Representative	Expected Result
Record < 1	invalid	0	Invalid record
1 <= Record <= 16,383	valid	10,987	Product found
Record > 16,383	invalid	17,000	Invalid record

3 test cases


Test Case Design Technique (dynamic testing)
b) Black-box
Black-Box Testing Strategies(*)
Requirements based/ specification based testing is an example of black box testing
c) White-box
i) Equivalence Partitioning (EP)
(*)
ii) Boundary Value Analysis (BVA)
(*)
i) Basis 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 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 partitions:
    - **Valid range: 100 to 300, 100 and 300**
    - **Invalid range 1: <100**
    - **Invalid range 2: >300**
    - Then create test cases using the input values that sit at the border (apply inside/outside rule)

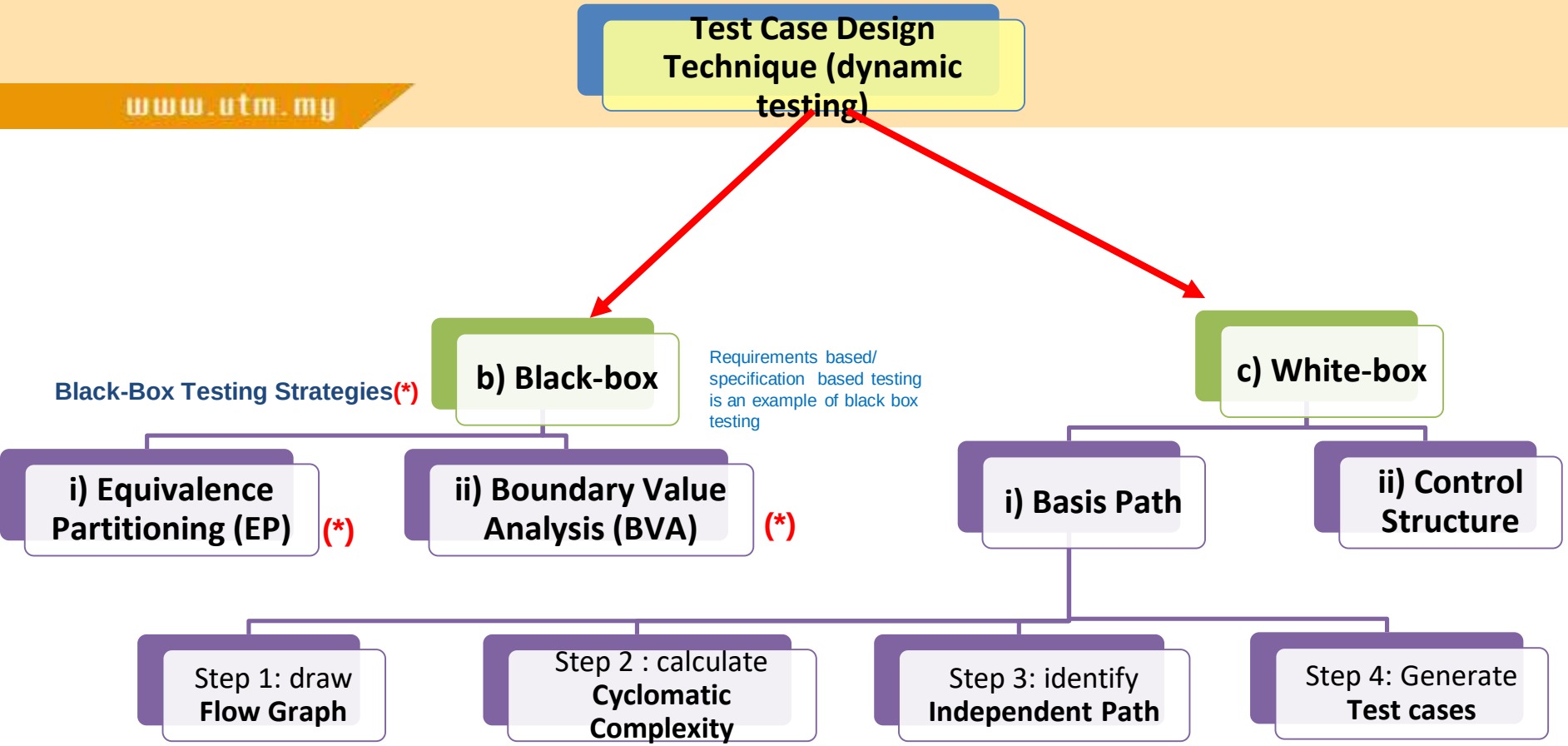
Invalid	Valid	Invalid
99	100 300	301

- **4 test cases, i.e. when value n = 99, 100, 300, and 301**

Test Cases

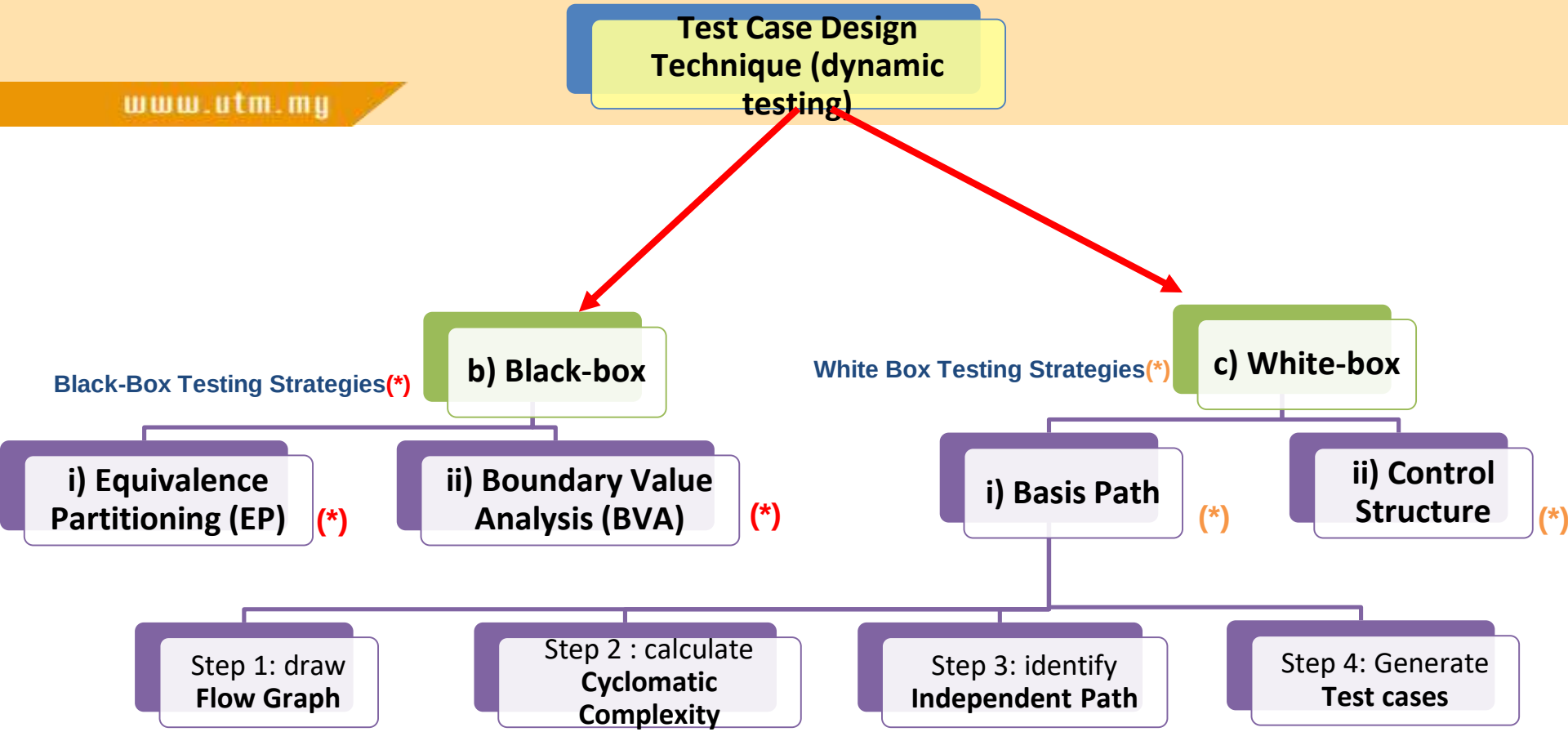
Equivalence Class	Status	Representative - BVA	Expected Result
Data < a (100)	invalid	99	Invalid data
a (100) <= Data <= b (300)	valid	100	Data valid
a (100) <= Data <= b (300)	valid	300	Data valid
Data > b (300)	invalid	301	Invalid record

4 test cases – due to 4 values of BVA


Test Case Design Technique (dynamic testing)
b) Black-box
Black-Box Testing Strategies(*)
Requirements based/ specification based testing is an example of black box testing
c) White-box
i) Equivalence Partitioning (EP)
(*)
ii) Boundary Value Analysis (BVA)
(*)
i) Basis Path
ii) Control Structure
Step 1: draw Flow Graph
Step 2 : calculate Cyclomatic Complexity
Step 3: identify Independent Path
Step 4: Generate Test cases

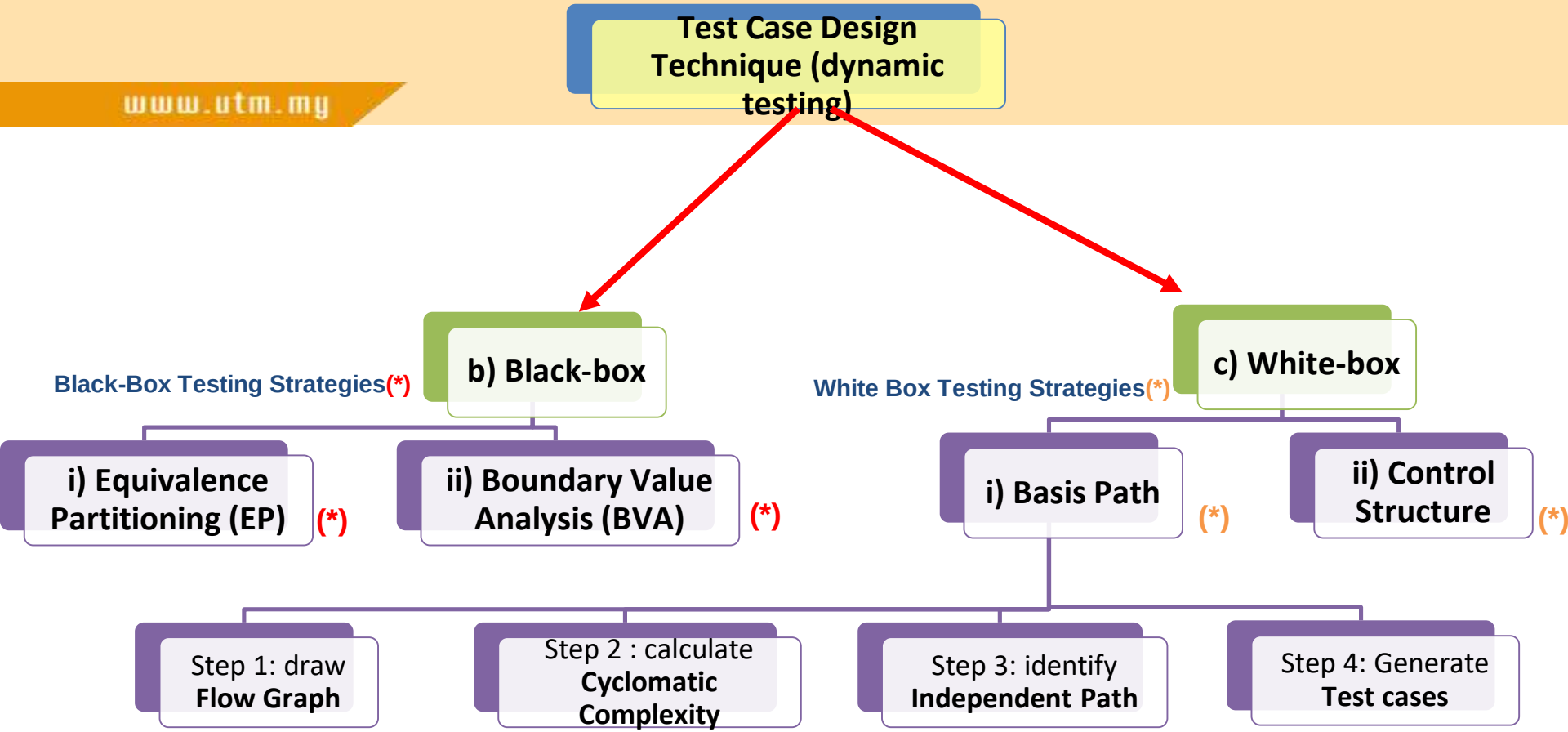
Test-Case Design Technique: White-Box Testing

- A verification technique that 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


Test Case Design Technique (dynamic testing)
b) Black-box
Black-Box Testing Strategies(*)
i) Equivalence Partitioning (EP)
(*)
ii) Boundary Value Analysis (BVA)
(*)
c) White-box
White Box Testing Strategies(*)
i) Basis 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 Technique: 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
  - exercise internal data structures to ensure their validity (Pressman, 2001)
- Strategies:
  1. **Basis Path Testing / Path Testing**
  2. **Control Structure Testing**

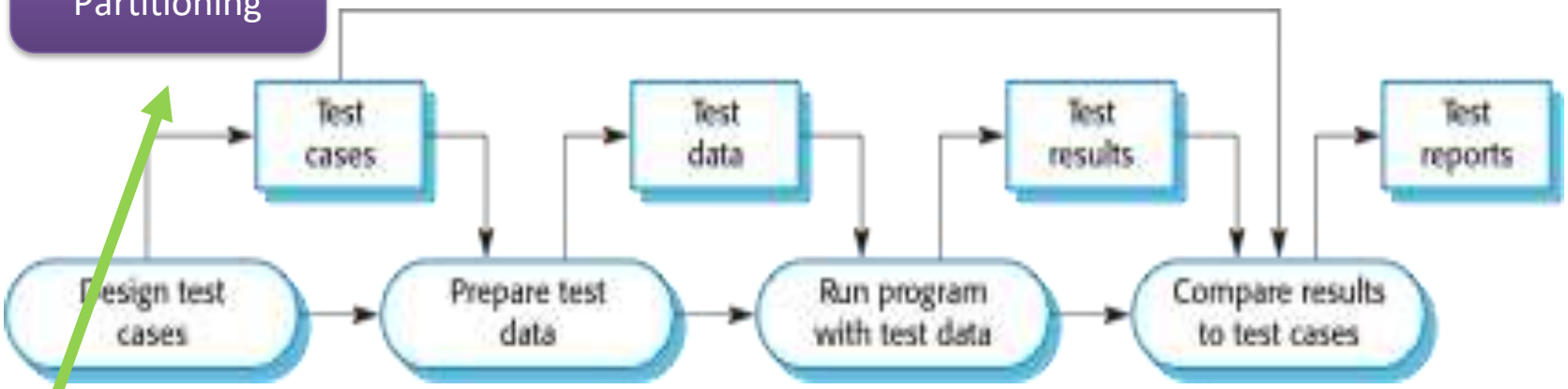

Test Case Design Technique (dynamic testing)
b) Black-box
Black-Box Testing Strategies(*)
i) Equivalence Partitioning (EP)
(*)
ii) Boundary Value Analysis (BVA)
(*)
c) White-box
White Box Testing Strategies(*)
i) Basis Path
(*)
ii) Control Structure
(*)
Step 1: draw Flow Graph
Step 2 : calculate Cyclomatic 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

Software Testing Process and Techniques (Summary)

Equivalence Partitioning



1. Black Box Testing
2. White Box Testing

Boundary Value Analysis