FULL STACK

ISTQB® Certified Tester Foundation Level Certification

CTFL 4.0

ISTQB® CTFL 4.0 Course Content

1. Fundamentals of Testing
2. Testing Throughout the Software Development Lifecycle
3. Static Testing
4. **Test Analysis and Design**
5. Managing the Test Activities
6. Test Tools

Static Testing

Md Rashed Karim
ISTQB Authorized Trainer
Member, ISTQB CTFL WG

After completion of this chapter the student will –

1. learn how to apply black-box, white-box, and experience-based test techniques to derive test cases from various software work products.
2. learn about the collaboration-based test approach.

4. Test Analysis and Design: Course Content

4.1.Test Techniques Overview

- Test techniques support the tester in test analysis (what to test) and in test design (how to test).
- Test techniques help to develop a relatively small, but sufficient, set of test cases in a systematic way.
- Test techniques also help the tester to define test conditions, identify coverage items, and identify test data during the test analysis and design.

4.1.Test Techniques Overview

Test techniques are classified as–

- Black-box test techniques
- White-box test techniques
- Experience-based test techniques

4.1.Test Techniques Overview

Black-box test techniques

- The Tester looks the test object as a Black Box, internal structure of the test object is irrelevant or unknown,
- Testing of input/ output behavior,
- Black box testing is also called functional testing or specification oriented testing

4.1.Test Techniques Overview

White-box test techniques

- Tester knows the internal structure and processing of the program and code. i.e. component hierarchy, control flow, data flow
- Test case are selected on the basis of internal program Code/ program structure
- White box testing is also called structure based testing Or control flow based testing

4.1.Test Techniques Overview

Experience-box test techniques

- Effectively use the knowledge and experience of testers for the design and implementation of test cases.
- The effectiveness of these techniques depends heavily on the tester’s skills.
- Experience-based test techniques can detect defects that may be missed using the blackbox and white-box test techniques. Hence, experience-based test techniques are complementary to the black-box and white-box test techniques

4.2.Black-Box Test Techniques

Commonly used black-box test techniques discussed in the following sections are:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State Transition Testing

4.2.1. Equivalence Partitioning (EP)

- Divides data into partitions (known as equivalence partitions) based on the expectation that all the elements of a given partition are to be processed in the same way by the test object.
- The theory behind this technique is that if a test case, that tests one value from an equivalence partition, detects a defect, this defect should also be detected by test cases that test any other value from the same partition. Therefore, one test for each partition is sufficient. Equivalence partitions can be identified for any data element related to the test object, including inputs, outputs, configuration items, internal values, time-related values, and interface parameters. The partitions may be continuous or discrete, ordered or unordered, finite or infinite. The partitions must not overlap and must be non-empty sets. For simple test objects EP can be easy, but in practice, understanding how the test object will treat different values is often complicated. Therefore, partitioning should be done with care.

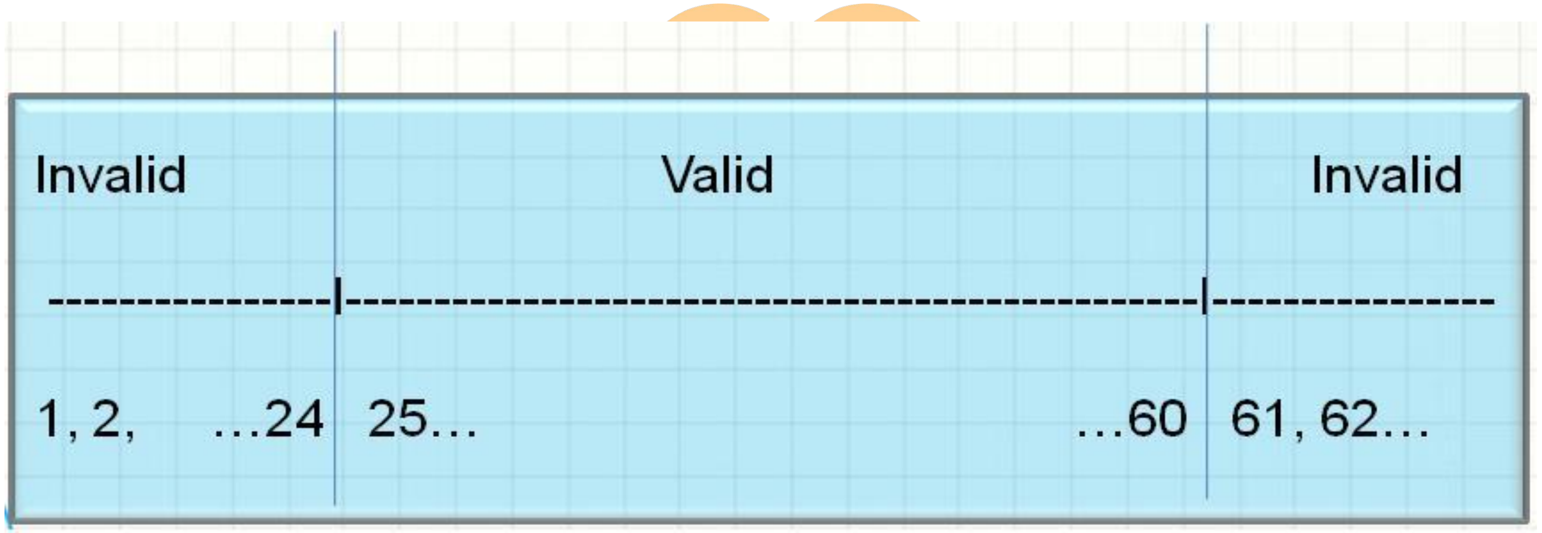
4.2.1. Equivalence Partitioning (EP)

- The range of defined values is grouped into equivalence classes, for which the following rules apply:
  - All values, for which a common behavior of the program is expected, are grouped together in one equivalence class
  - Equivalence class may not overlap and may not contain any gaps
  - Equivalence class may contain a range of values(e.g. 0<=x<=10) or a single value (e.g. x="Yes")
- invalid EP
- valid EP:

4.2.1. Equivalence Partitioning (EP)

age

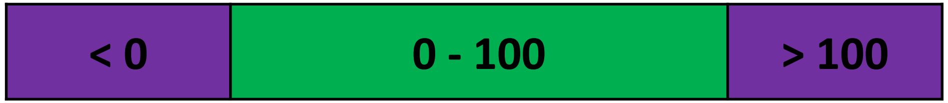
age



4.2.1. Equivalence Partitioning (EP)

Equivalence partitioning – example

- if a value x is defined as 0≤ x ≤ 100, then when we can initially identify 3 equivalence partitions:
  1. x<0 (invalid input values)
  2. 0 ≤ x≤100 (valid input)
  3. x> 100 (invalid input values)
- Further invalid EP can be defined, containing, but not limited to:
  - non-numerical inputs,
  - numbers to big or too small,
  - non-supported format for numbers



4.2.1. Equivalence Partitioning (EP)

Equivalence partitioning – example

The percentage value will now be displayed in a bar chart.

The following additional requirements apply (both values included):

- values between 0 and 15 :	Orange bar,
- values between 16 and 50:	Green bar,
- values between 51 and 85:	Yellow bar,
- values between 86 and 100:	Blue bar,

Solution for valid EC:

- Now there are four instead of one valid equivalence classes:

- 1st valid equivalence class:	**$0 \le x \le 15$**
- 2nd valid equivalence class:	**$16 \le x \le 50$**
- 3rd valid equivalence class:	**$51 \le x \le 85$**
- 4th valid equivalence class:	**$86 \le x \le 100$**

< 0	0 - 15	16 - 50	51 - 85	86 – 100	> 100

4.2.1. Equivalence Partitioning (EP)

Variable	EC	Status	Representative
Percentage Value	EC 1: 0<= X <=15	Valid	+10
	EC 2: 16<= X <=50	Valid	+20
	EC 3: 51<= X <=85	Valid	+80
	EC 4: 86<= X <=100	Valid	+90
	EC 5: X<0	Invalid	-10
	EC 6: X>100	Invalid	+200
	EC 7: X not integer	Invalid	1.5
	EC 8: X non number	Invalid	fred

4.2.1. Equivalence Partitioning (EP)

Equivalence partitioning – example 2 /1

Analyzing the specification

A piece of code that computes the **price** of a sales order, based on its **Product Price**, a **discount** in % and **shipping costs** (BDT 19, 29, 49 depending on shipping mode)

Variable	Equivalence class	Status	Representatives
Product Price	EC_{11}: x >= 0	Valid	1000
	EC_{12}: x < 0	invalid	-1000
	EC_{13}: x non-numerical value	Invalid	Rashed
Discount	EC_{21}: 0% ≤ x ≤ 100%	valid	10%
	EC_{22}: x < 0%	Invalid	-10%
	EC_{23}: x > 100	Invalid	200%
	EC_{24}: x non numeric value	Invalid	Karim
Shipping costs	EC_{31}: x = 19	valid	19
	EC_{32}: x = 29	valid	29
	EC_{33}: x = 49	valid	49
	EC_{34}: x ≠ {19, 29, 49}	Invalid	30
	EC_{35}: x non numeric value	invalid	Student

Variable	Equivalence class	Status	Representatives	T1	T2	T3
Product Price	EC_{11}: x >= 0	Valid	1000	*	*	*
	EC_{12}: x < 0	invalid	-1000			
	EC_{13}: x non-numerical value	Invalid	Rashed			
Discount	EC_{21}: $0\% \leq x \leq 100\%$	valid	10%	*	*	*
	EC_{22}: x < 0%	Invalid	-10%			
	EC_{23}: x > 100	Invalid	200%			
	EC_{24}: x non numeric value	Invalid	Karim			
Shipping costs	EC_{31}: x = 19	valid	19	*		
	EC_{32}: x = 29	valid	29		*	
	EC_{33}: x = 49	valid	49			*
	EC_{34}: $x \neq \{19, 29, 49\}$	Invalid	30			
	EC_{35}: x non numeric value	invalid	Student			

Variable	Equivalence class	Status	Representatives	T4	T5	T6	T7	T8	T9	T10
Values of goods	EC_{11}: x >= 0	Valid	1000			*	*	*	*	*
	EC_{12}: x < 0	invalid	-1000	*						
	EC_{13}: x non-numerical value	Invalid	Rashed		*					
Discount	EC_{21}: 0% ≤ x ≤ 100%	valid	10%	*	*				*	*
	EC_{22}: x < 0%	Invalid	-10%			*				
	EC_{23}: x > 100	Invalid	200%				*			
	EC_{24}: x non numeric value	Invalid	Karim					*		
Shipping costs	EC_{31}: x = 19	valid	19	*	*	*	*	*		
	EC_{32}: x = 29	valid	29							
	EC_{33}: x = 49	valid	49							
	EC_{34}: x ≠ {19, 29, 49}	Invalid	30						*	
	EC_{35}: x non numeric value	invalid	Student							*

FULL STACK

Variable	Equivalence class	Status	Representative	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Product Price	EC_{11}: $x >= 0$	Valid	1000	*	*	*			*	*	*	*	*
	EC_{12}: $x < 0$	invalid	-1000				*						
	EC_{13}: x non-numerical value	Invalid	Rashed					*					
Discount	EC_{21}: $0\% \leq x \leq 100\%$	valid	10%	*	*	*	*	*				*	*
	EC_{22}: $x < 0\%$	Invalid	-10%						*				
	EC_{23}: > 100	Invalid	200%							*			
	EC_{24}: x non numeric value	Invalid	Karim								*		
Shipping costs	EC_{31}: $x = 19$	valid	19	*			*	*	*	*	*		
	EC_{32}: $x = 29$	valid	29		*								
	EC_{33}: $x = 49$	valid	49			*							
	EC_{34}: $x \neq \{19, 29, 49\}$	Invalid	30									*	
	EC_{35}: x non numeric value	invalid	Student										*

Equivalence partitioning – coverage

Equivalence partitioning coverage can be used as exit criteria to end testing activities

$$\text{EP Coverage} = \frac{\text{Number of EP tested}}{\text{Number of EP defined}} * 100\%$$

Boundary Value Analysis

Boundary Value Analysis

- BVA is a technique based on exercising the boundaries of equivalence partitions.
- BVA can only be used for ordered partitions.
- The minimum and maximum values of a partition are its boundary values.
- if two elements belong to the same partition, all elements between them must also belong to that partition.
- BVA focuses on the boundary values of the partitions because developers are more likely to make errors with these boundary values.
- Typical defects found by BVA are located where implemented boundaries are misplaced to positions above or below their intended positions or are omitted altogether.
- This syllabus covers two versions of the BVA: **2-value** and **3-value** BVA. They differ in terms of coverage items per boundary that need to be exercised to achieve 100% coverage.

Boundary Value Analysis

2-value BVA:

- for each boundary value there are two coverage items: this boundary value and its closest neighbor belonging to the adjacent partition.
- To achieve 100% coverage with 2-value BVA, test cases must exercise all coverage items, i.e., all identified boundary values.
- Coverage is measured as the number of boundary values that were exercised, divided by the total number of identified boundary values, and is expressed as a percentage.

$$\text{BVA Coverage} = \frac{\text{Number of BV tested}}{\text{Total Number of BV defined}} * 100\%$$

Boundary Value Analysis

3-value BVA:

- for each boundary value there are three coverage items: this boundary value and both its neighbors.
- Therefore, in 3-value BVA some of the coverage items may not be boundary values. To achieve 100% coverage with 3-value BVA, test cases must exercise all coverage items, i.e., identified boundary values and their neighbors.
- Coverage is measured as the number of boundary values and their neighbors exercised, divided by the total number of identified boundary values and their neighbors, and is expressed as a percentage.
- 3-value BVA is more rigorous than 2-value BVA as it may detect defects overlooked by 2-value BVA. For example, if the decision “if (x ≤ 10) …” is incorrectly implemented as “if (x = 10) …”, no test data derived from the 2-value BVA (x = 10, x = 11) can detect the defect. However, x = 9, derived from the 3-value BVA, is likely to detect it

Boundary Value Analysis

Evaluation:

- 3-value BVA is more rigorous than 2-value BVA as it may detect defects overlooked by 2-value BVA.
- For example, if the decision “if (x ≤ 10) …” is incorrectly implemented as “if (x = 10) …”, no test data derived from the 2-value BVA (x = 10, x = 11) can detect the defect. However, x = 9, derived from the 3-value BVA, is likely to detect it

Boundary Value Analysis

- Boundary value analysis extends equivalence partitioning by introducing a **rule** for the **choice of representatives**
- The **edge values** of the **equivalence class** are to be tested **intensively**
- **Why put more attention to the edges?**
  - Often, the boundaries of values ranges are **not well defined** or lead to different interpretations
  - Checking if the boundaries were **programmed correctly**

Please note:

Experience shows that **error** occur **very frequently** on the **boundaries** of value ranges!

Boundary Value Analysis

Defining boundary values:

- If the EP is defined as single numerical value, for example x= 5, the **neighboring values** will be used as well
- the representatives (of the class and its **neighboring values**) are: 4,5 and 6

Boundary Value Analysis

Boundary analysis example:

Value range for a discount in % : $0 \leq x \leq 100$

Definition of EP

- EC1: $x < 0$
- EC2: $0 \leq x \leq 100$
- EC3: $x > 100$

Boundary analysis

extends the representatives with neighboring values:

EC2: -1; 0; 1; 99; 100; 101

Please note:

Instead of one representative for the valid EP, there are now six representatives (four valid and two invalid)

Boundary Value Analysis

Boundary values optimization:

- **Basic scheme**: choose three values to be tested the exact boundary and the two neighboring values (within and outside the EP) [3-Point Boundary Value]
- **Alternative point of view**: since the boundary value belongs to the EP, only two values are needed for testing: one within and one outside the EP [2-Point Boundary Value]

Example:

Value range for a discount in %: 0 ≤ x ≤ 100

Valid EC: 0 ≤ x ≤ 100

Boundary analysis

Additional representatives are: -1; 0;   100; 101

1 – same behavior as 0

99 - same behavior as 100

A programming error caused by a wrong comparison operator will be found with the two boundary values

Decision Table Testing

Decision Table Testing

- Equivalence class partitioning and boundary analysis deal with **isolated input** conditions.
- However, an input condition may have an effect only in **combination** with other input conditions.
- All previously described methods do not take into account the effects of **dependencies and combinations.**
- Using the **full set of combinations** of all input equivalence classes usually leads to a very **high number of test cases** (***test case explosion***).
- With the help of decision tables derived from them, **the amount of possible combinations** can be **reduced** systematically to a subset.

Decision Table Testing

Conditions (Causes)	Condition Entry
Action (Effects)	Action Entry

Decision Table Testing: Business Case

Online-Banking

The user has identified himself via account number a PIN. If having **sufficient balance**, s/he is able to set a transferal. To do this he must input the **correct details of the Recipient** and a **valid TAN** (OTP)

Decision Table Testing: Business Case

	T1	T2	T3	T4	T5	T6	T7	T8
Sufficient Salance	Y	Y	Y	Y	N	N	N	N
Correct Recipient	Y	Y	N	N	Y	Y	N	N
Valid TAN	Y	N	Y	N	Y	N	Y	N
Do transferal	X							
Mark TAN as used	X							
Deny transferal			X	X	X	X	X	X
Request again TAN		X						

FULL STAC

Decision Table Testing: Business Case

	T1	T2	T3	T4	T5	T6	T7	T8
Sufficient Salance	Y	Y	Y	Y	N	N	N	N
Correct Recipient	Y	Y	N	N	Y	Y	N	N
Valid TAN	Y	N	Y	N	Y	N	Y	N
Do transferal	X							
Mark TAN as used	X							
Deny transferal			X	X	X	X	X	X
Request again TAN		X						

FULL STACK

Decision Table Testing: Business Case

	T1	T2	T3	T4	T5
Sufficient Salance	Y	Y	Y	Y	N
Correct Recipient	Y	Y	N	N	Y
Valid TAN	Y	N	Y	N	Y
Do transferal	X				
Mark TAN as used	X				
Deny transferal			X	X	X
Request again TAN		X			

FULL STACK

Decision Table Testing: Business Case

	T1	T2	T3	T5
Sufficient Salance	Y	Y	Y	N
Correct Recipient	Y	Y	N	Y
Valid TAN	Y	N	Y	Y
Do transferal	X			
Mark TAN as used	X			
Deny transferal			X	X
Request again TAN		X		

Decision Table Testing: Business Case

Online-Banking

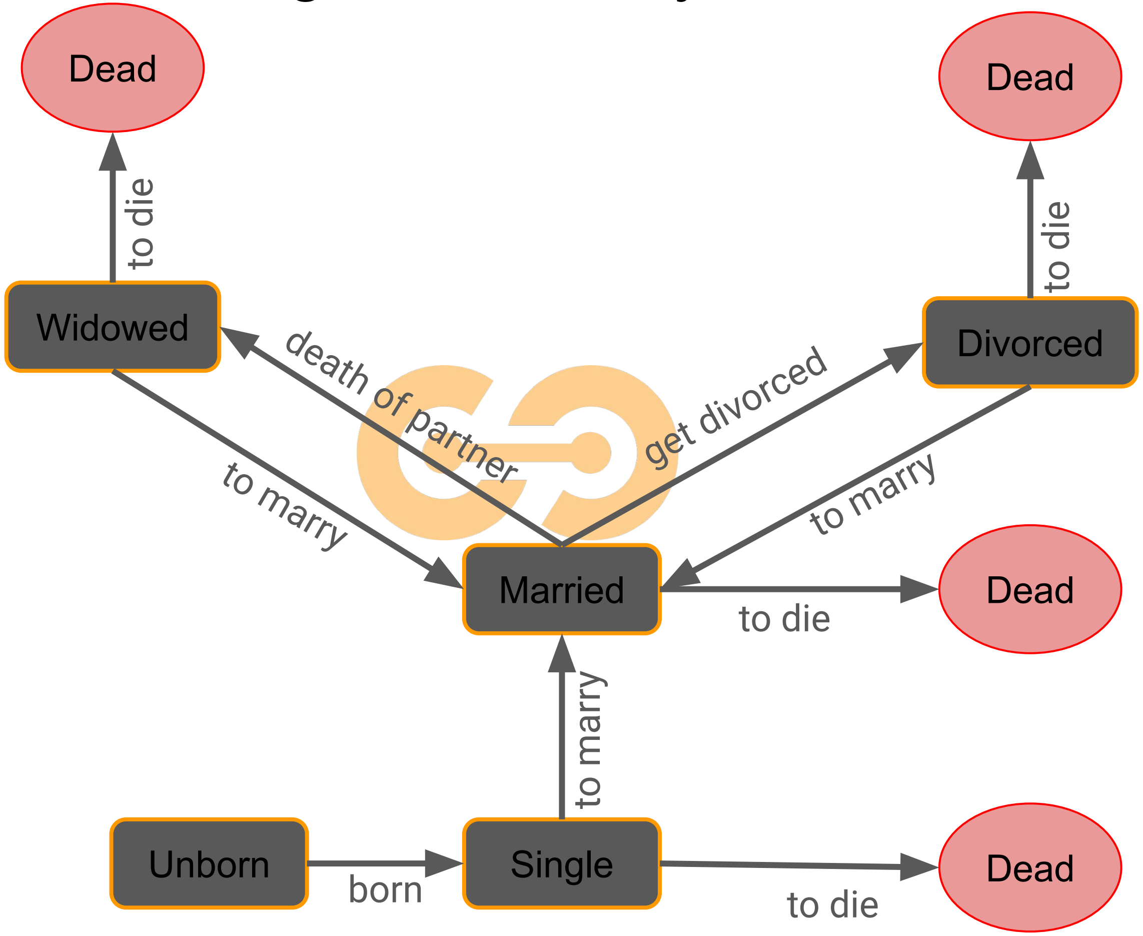
- Each table column represents a test case
- Choose an effect / action
- Trace back along the diagram to identify the cause
- Each combination of causes represents a column of the decision table (a test case)
- Identical combinations of causes, leading to different effects, may be merged, to form a single test case

State Transition Testing

State Transition Diagram

- A state transition diagram models the behavior of a system by showing its possible states and valid state transitions.
- A transition is initiated by an event, which may be additionally qualified by a guard condition.
- The transitions are assumed to be instantaneous and may sometimes result in the software taking action.
- The common transition labeling syntax is as follows: “event [guard condition] / action”.
- Guard conditions and actions can be omitted if they do not exist or are irrelevant for the tester.

State Transition Diagram on lifecycle of a person



State Transition Table

- A state table is a model equivalent to a state transition diagram. Its rows represent states, and its columns represent events.
- Table entries (cells) represent transitions, and contain the target state, as well as the resulting actions, if defined.
- In contrast to the state transition diagram, the state table explicitly shows invalid transitions, which are represented by empty cells.

FULL STACK

State Table on lifecycle of a person

SL	State1	State2	State3	State4	State5	State6	End State
1	Unborn	Single	Dead				Dead
2	Unborn	Single	Married	Dead			Dead
3	Unborn	Single	Married	Widowed	Dead		Dead
4	Unborn	Single	Married	widowed	Married	Dead	Dead
5	Unborn	Single	Married	divorced	Dead		Dead
6	Unborn	Single	Married	divorced	Married	Dead	Dead

State Transition Test Case

- A test case based on a state transition diagram or state table is usually represented as a sequence of events, which results in a sequence of state changes (and actions, if needed).
- One test case may, and usually will, cover several transitions between states.

There exist many coverage criteria for state transition testing

- **All states coverage**
- Valid transitions coverage
- All transitions coverage

Coverage criteria: All states coverage

All states coverage:

- the coverage items are the states.
- To achieve 100% all states coverage, test cases must ensure that all the states are visited.

$$\text{All states Coverage} = \frac{\text{Number of visited states}}{\sum \text{(states)}} * 100\%$$

Coverage criteria for state transition testing

Valid transitions coverage (also called 0-switch coverage):

- the coverage items are single valid transitions.
- To achieve 100% valid transitions coverage, test cases must exercise all the valid transitions.

$$\text{Valid transitions Coverage} = \frac{\text{Number of exercised valid transitions}}{\sum \text{(valid transitions)}} *100\%$$

Coverage criteria for state transition testing

All transitions coverage:

- the coverage items are all the transitions shown in a state table.
- To achieve 100% all transitions coverage, test cases must exercise all the valid transitions and attempt to execute invalid transitions.
- Testing only one invalid transition in a single test case helps to avoid fault masking, i.e., a situation in which one defect prevents the detection of another.

$$\text{All transitions Coverage} = \frac{\text{Number of valid and invalid transitions exercised}}{\sum \text{(valid and invalid transitions)}} * 100\%$$

Coverage criteria for state transition testing

Evaluation:

- All states coverage is weaker than valid transitions coverage, because it can typically be achieved without exercising all the transitions.
- Valid transitions coverage is the most widely used coverage criterion.
- Achieving full valid transitions coverage guarantees full all states coverage.
- Achieving full all transitions coverage guarantees both full all states coverage and full valid transitions coverage and should be a minimum requirement for mission and safety-critical software.

4.3. White-Box Test Techniques

4.3. White-Box Test Techniques

Because of their popularity and simplicity, this section focuses on two code-related white-box test techniques:

- Statement testing
- Branch testing

4.3.1. Statement Testing and Statement Coverage

FULL STACK

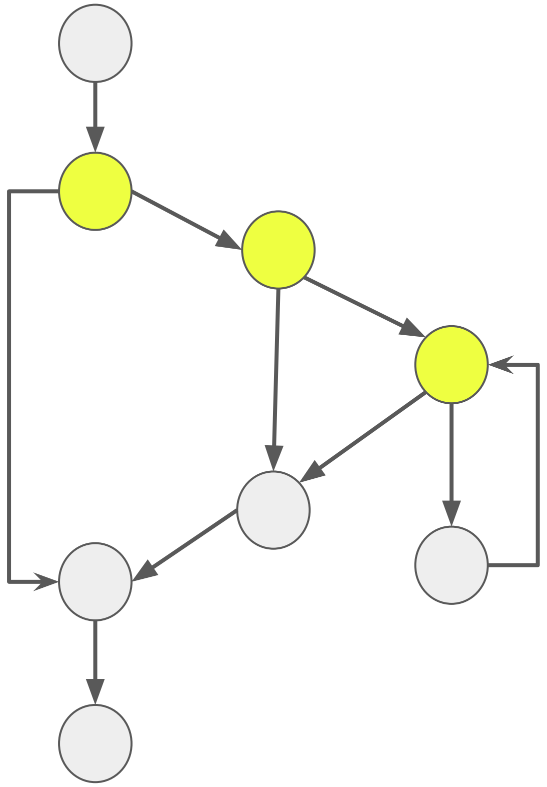
4.3.1. Statement Testing and Statement Coverage

- The coverage items are executable statements.
- The aim is to design test cases that exercise statements in the code until an acceptable level of coverage is achieved.

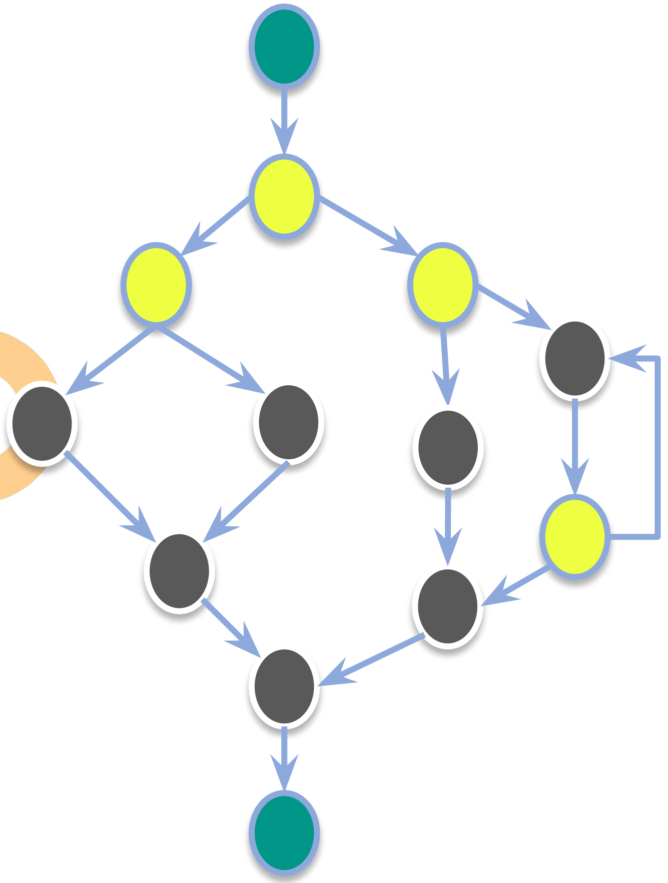
$$\text{Statement Coverage} = \frac{\text{Number of statements exercised}}{\sum \text{(executable statements)}} * 100\%$$

Statement Coverage /1

```
if(i>0) {
     if(j>10){
               while(k>10){
                    Do task 1
               }
     }
     Do task 2
}
Do End task
```



Statement Coverage /2



4.3.1. Statement Testing and Statement Coverage

- When 100% statement coverage is achieved, it ensures that all executable statements in the code have been exercised at least once.
- In particular, this means that each statement with a defect will be executed, which may cause a failure demonstrating the presence of the defect.
- However, exercising a statement with a test case will not detect defects in all cases. For example, it may not detect defects that are data dependent (e.g., a division by zero that only fails when a denominator is set to zero).
- 100% statement coverage does not ensure that all the decision logic has been tested.

4.3.2. Branch Testing and Branch Coverage

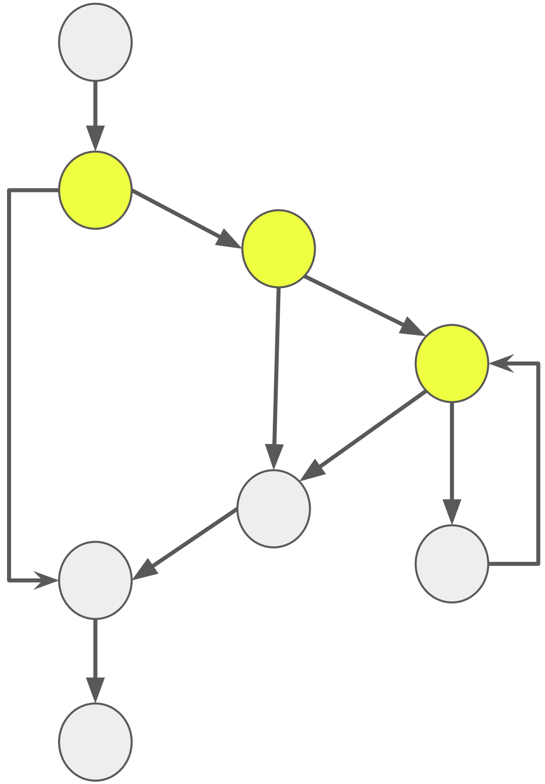
4.3.2. Branch Testing and Branch Coverage

- A branch is a transfer of control between two nodes in the control flow graph, which shows the possible sequences in which source code statements are executed in the test object.
- Each transfer of control can be either unconditional (i.e., straight-line code) or conditional (i.e., a decision outcome).
- In branch testing the coverage items are branches and the aim is to design test cases to exercise branches in the code until an acceptable level of coverage is achieved.

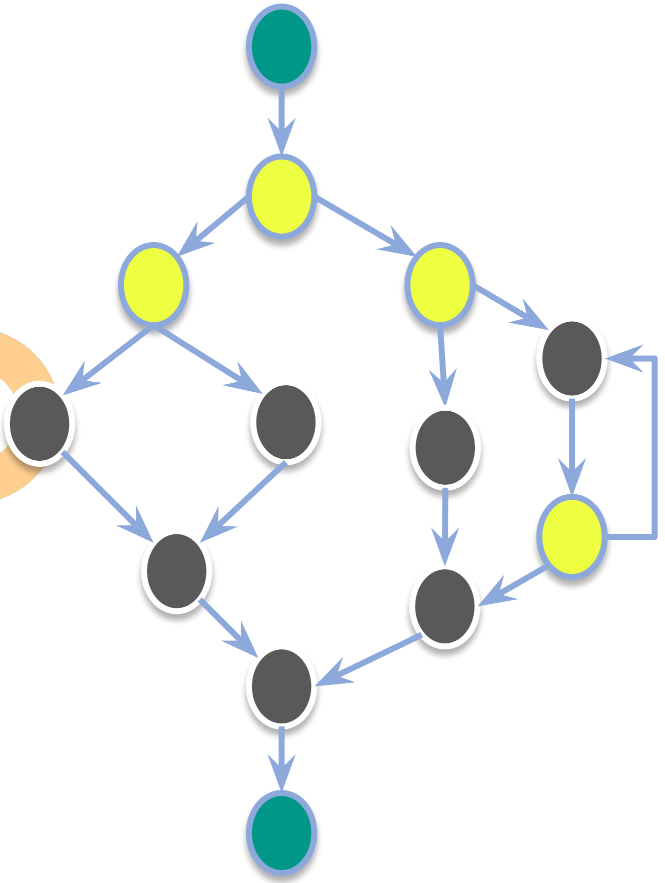
$$\text{Branch Coverage} = \frac{\text{Number of branches exercised}}{\sum \text{(executable branches)}} * 100\%$$

Branch Coverage /1

```
if(i>0) {
    if(j>10){
        while(k>10){
            Do task 1
        }
    }
    Do task 2
}
Do End task
```



Branch Coverage /2



4.3.2. Branch Testing and Branch Coverage

- When 100% branch coverage is achieved, all branches in the code, unconditional and conditional, are exercised by test cases.
- Conditional branches typically correspond to a true or false outcome from an “if...then” decision, an outcome from a switch/case statement, or a decision to exit or continue in a loop.
- However, exercising a branch with a test case will not detect defects in all cases. For example, it may not detect defects requiring the execution of a specific path in a code.
- 100% branch coverage also achieves 100% statement coverage (but not vice versa).

4.3.3. The Value of White-box Testing

Use Case:

- White-box techniques can be used in static testing (e.g., during dry runs of code).
- Well suited to reviewing code that is not yet ready for execution, as well as pseudocode and other high-level or top-down logic which can be modeled with a control flow graph.
- Performing only black-box testing does not provide a measure of actual code coverage.

Strength:

- White-box technique facilitates defect detection even when the software specification is vague, outdated or incomplete.

Weakness:

- if the software does not implement one or more requirements, white box testing may not detect the resulting defects of omission.

Outcome:

- White-box coverage measures provide an objective measurement of coverage and provide the necessary information to allow additional tests to be generated to increase this coverage, and subsequently increase confidence in the code.

4.4.Experience-based Test Techniques

4.4.Experience-based Test Techniques

Commonly used experience-based test techniques discussed in the following sections are:

- **Error guessing**
- **Exploratory testing**
- **Checklist-based testing**

4.4.1. Error Guessing

4.4.1. Error Guessing

Error guessing is a technique used to anticipate the occurrence of errors, defects, and failures, based on the tester’s knowledge, including:

- How the application has worked in the past
- The types of errors the developers tend to make and the types of defects that result from these errors
- The types of failures that have occurred in other, similar applications

4.4.1. Error Guessing

In general, errors, defects and failures may be related to:

- Input (e.g. correct input not accepted, parameters wrong or missing)
- Output (e.g. wrong format, wrong result)
- Logic (e.g. missing cases, wrong operator)
- Computation (e.g. incorrect operand, wrong computation)
- Interfaces (e.g. parameter mismatch, incompatible types)
- Data (e.g. incorrect initialization, wrong type)

4.4.1. Error Guessing

- Fault attacks are a methodical approach to the implementation of error guessing.
- This technique requires the tester to create or acquire a list of possible errors, defects and failures, and to design tests that will identify defects associated with the errors, expose the defects, or cause the failures.
- These lists can be built based on experience, defect and failure data, or from common knowledge about why software fails.

4.4.2. Exploratory Testing

4.4.2. Exploratory Testing

- Tests are simultaneously designed, executed, and evaluated while the tester learns about the test object.
- Testing is used to learn more about the test object, to explore it more deeply with focused tests, and to create tests for untested areas.
- Exploratory testing is sometimes conducted using session-based testing to structure the testing.
    - Session-based approach, exploratory testing is conducted within a defined time-box.
    - The tester uses a test charter containing test objectives to guide the testing.
    - Test session is usually followed by a debriefing that involves a discussion between the tester and stakeholders interested in the test results of the test session.
    - In this approach test objectives may be treated as high-level test conditions.
    - Coverage items are identified and exercised during the test session.
    - The tester may use test session sheets to document the steps followed and the discoveries made.

4.4.2. Exploratory Testing

Use Case:

- is useful when there are few or inadequate specifications or there is significant time pressure on the testing.
- is also useful to complement other more formal test techniques.
- will be more effective if the tester is experienced, has domain knowledge and has a high degree of essential skills, like analytical skills, curiosity and creativeness.
- can incorporate the use of other test techniques (e.g., equivalence partitioning)

4.4.3. Checklist-Based Testing

4.4.3. Checklist-Based Testing

- A tester designs, implements, and executes tests to cover test conditions from a checklist.
- Checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails.
- Checklists should not contain items –
  - that can be checked automatically,
  - better suited as entry/exit criteria, or
  - that are too general.
- Checklist items are often phrased in the form of a question.
- It should be possible to check each item separately and directly.
- These items may refer to requirements, graphical interface properties, quality characteristics or other forms of test conditions.
- Checklists can be created to support various test types, including functional and non-functional testing.

4.4.3. Checklist-Based Testing

- Some checklist entries may gradually become less effective over time because the developers will learn to avoid making the same errors.
- New entries may also need to be added to reflect newly found high severity defects. Therefore, checklists should be regularly updated based on defect analysis.
- However, care should be taken to avoid letting the checklist become too long.
- In the absence of detailed test cases, checklist-based testing can provide guidelines and some degree of consistency for the testing.
- If the checklists are high-level, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

4.5.Collaboration-based Test Approaches

4.5.Collaboration-based Test Approaches

- Each of the above-mentioned techniques has a particular objective with respect to defect detection.
- Collaboration-based approaches, on the other hand, focus also on defect avoidance by collaboration and communication.
- There are three Collaboration-based approaches–
    - Collaborative User Story Writing
    - Acceptance Criteria
    - Acceptance Test-driven Development (ATDD)

4.5.1. Collaborative User Story Writing

- A user story represents a feature that will be valuable to either a user or client. User stories have three critical aspects, called together the "**3C's**":
  - **Card** – the medium describing a user story
  - **Conversation** – explains how the software will be used
  - **Confirmation** – the acceptance criteria
- The most common format for a user story is

  "As a [*role*],

  I want [*goal to be accomplished*],

  so that I can [resulting business value for the role]"

  followed by the acceptance criteria.

4.5.1. Collaborative User Story Writing

- Collaborative authorship of the user story can use techniques such as brainstorming and mind mapping.
- The collaboration allows the team to obtain a shared vision of what should be delivered, by taking into account three perspectives:
  - Business,
  - Development and
  - Testing.
- Good user stories should be: Independent, Negotiable, Valuable, Estimable, Small and Testable (INVEST). If a stakeholder does not know how to test a user story, this may indicate that the user story is not clear enough, or that it does not reflect something valuable to them, or that the stakeholder just needs help in testing (Wake 2003).

4.5.2. Acceptance Criteria

Definition:

- Acceptance criteria for a user story are the conditions that an implementation of the user story must meet to be accepted by stakeholders.
- Acceptance criteria are the test conditions that should be exercised by the tests.
- Acceptance criteria are usually a result of the Conversation.
- Acceptance criteria are used to:
  - Define the scope of the user story
  - Reach consensus among the stakeholders
  - Describe both positive and negative scenarios
  - Serve as a basis for the user story acceptance testing
  - Allow accurate planning and estimation

4.5.2. Acceptance Criteria

There are several ways to write acceptance criteria for a user story. The two most common formats are:

- Scenario-oriented (e.g., Given/When/Then format used in BDD)
- Rule-oriented (e.g., bullet point verification list, or tabulated form of input-output mapping)

Most acceptance criteria can be documented in one of these two formats. However, the team may use another, custom format, as long as the acceptance criteria are well-defined and unambiguous.

Simple User Story Template

Title:	Priority:	Estimate:
User story As a [type of user], I want to [perform some task] so that I can [achieve some goal].		
Acceptance criteria **Given** that [some context], **when** [some action is carried out] **then** [a set of observable outcomes should occur].		

User Story for banking system

As a banking system user,
I want to be able to transfer funds between my accounts
so that I can manage my finances conveniently.

Acceptance criteria /1

- Given that I am logged into the banking system, when I navigate to the "Transfer Funds" section, I should see a form to initiate a fund transfer.
- Given that I am in the "Transfer Funds" section, when I enter the source account number, destination account number, and the transfer amount in the provided fields, I should be able to submit the transfer request.
- Given that I have submitted a transfer request, when the request is successfully processed, the amount should be deducted from the source account balance and added to the destination account balance.
- Given that I have submitted a transfer request, if the source account does not have sufficient funds to cover the transfer amount, I should receive an error message indicating insufficient balance and the transfer should not be processed.
- Given that I have submitted a transfer request, if the destination account number is invalid or does not exist, I should receive an error message indicating an invalid account and the transfer should not be processed.

Acceptance criteria /2

- Given that a transfer request is successfully processed, I should receive a confirmation message indicating the details of the transfer, including the source and destination account numbers and the transferred amount.
- Given that I am on the "Transfer Funds" section, I should have the option to view a transaction history or statement that includes all my past fund transfers.
- Given that I am on the "Transfer Funds" section, I should be able to view the current available balance for each of my accounts before initiating a transfer.
- Given that I am on the "Transfer Funds" section, the transfer form should include appropriate validation to ensure that valid account numbers and transfer amounts are entered.
- Given that I am on the "Transfer Funds" section, the system should enforce appropriate security measures, such as user authentication, to prevent unauthorized access to the transfer functionality.

Sample tabulated form of input-output mapping for a banking system transaction:

Input	Output
Account Number: 123456789 Amount: $100.00	Transaction Successful New Balance: $500.00
Account Number: 987654321 Amount: $50.00	Transaction Successful New Balance: $250.00
Account Number: 123456789 Amount: $1000.00	Insufficient Funds
Account Number: 555555555 Amount: $25.00	Invalid Account Number
Account Number: 123456789 Amount: $-50.00	Invalid Transaction Amount

Sample tabulated form of input-output mapping for a banking system transaction:

- In this example, the input represents different scenarios for a banking system transaction. Each row represents a specific input-output pair. The inputs include the account number and the transaction amount, while the outputs reflect the result of the transaction.
- For instance, when the input includes Account Number: 123456789 and Amount: $100.00, the output is "Transaction Successful" and the new account balance is $500.00. On the other hand, if the input is Account Number: 123456789 and Amount: $1000.00, the output is "Insufficient Funds" indicating that the transaction cannot be completed due to insufficient balance in the account.
- Please note that the actual inputs and outputs in a banking system would involve additional details such as authentication, transaction IDs, timestamps, and more. The provided example is simplified for illustrative purposes.

4.5.3. Acceptance Test-driven Development (ATDD)

- Test cases are created by team members with different perspectives, e.g., customers, developers, and testers prior to implementing the user story and executed manually or automated.
- The first step is a specification workshop where the user story and its acceptance criteria are analyzed, discussed, and written by the team members. Incompleteness, ambiguities, or defects in the user story are resolved during this process.
- The next step is to create the test cases. This can be done by the team as a whole or by the tester individually.
- The test cases are based on the acceptance criteria and can be seen as examples of how the software works.
- This will help the team implement the user story correctly.

4.5.3. Acceptance Test-driven Development (ATDD)

- Typically, the first test cases are positive, confirming the correct behavior without exceptions and comprising the sequence of activities executed if everything goes as expected.
- After the positive test cases are done, the team should perform negative testing.
- Finally, the team should cover non-functional quality characteristics as well.
- Test cases should be expressed in a way that is understandable for the stakeholders.
- Typically, test cases contain sentences in natural language involving the necessary preconditions (if any), the inputs, and the postconditions.

4.5.3. Acceptance Test-driven Development (ATDD)

- The test cases must cover all the characteristics of the user story and should not go beyond the story. However, the acceptance criteria may detail some of the issues described in the user story. In addition, no two test cases should describe the same characteristics of the user story.
- When captured in a format supported by a test automation framework, the developers can automate the test cases by writing the supporting code as they implement the feature described by a user story. The acceptance tests then become executable requirements.

Thank You