Testing Techniques

Exploring Different Approaches to Testing

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University
http://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#QA-Fund

Testing Techniques

Definition, Purpose, Categories

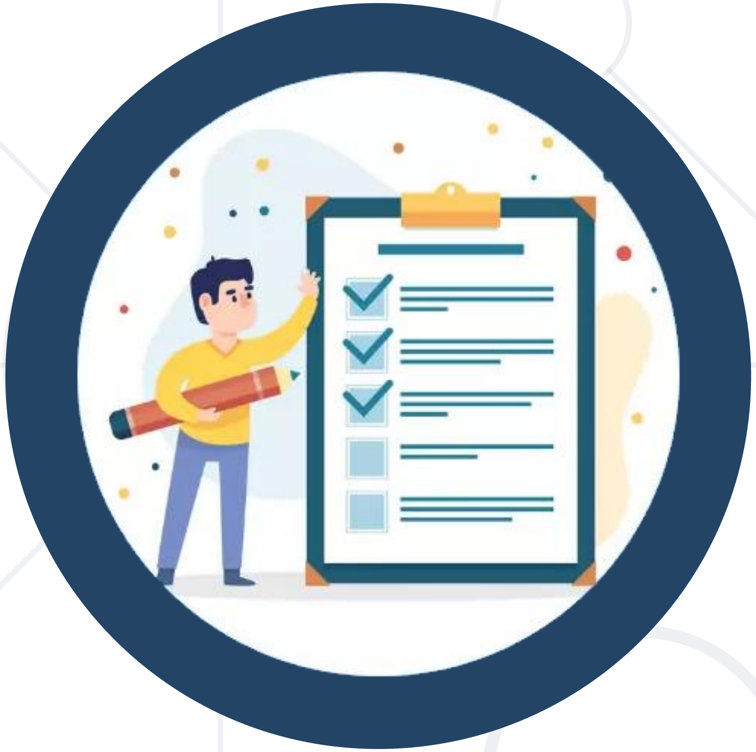
Testing Techniques Overview

Software University

- **Systematic approaches** for software testing
- **Why We Need Different Test Techniques:**
  - Address **diverse** and **complex** software **systems**
  - Detect **varying types** of defects
  - **Optimize resources** and testing efforts
- **Categories:**
  - **Static** - Analyzing code, requirements, and design without execution
  - **Dynamic** - Executing software and observing outcomes

Static vs. Dynamic Techniques

- **Static**
  - Emphasizes **early defect detection** and prevention
  - **Analyzing software** (code, design documents, requirements) **without execution**
  - Identifies **syntax** errors, **logical** errors, coding **standards violations**, and **document errors**
- **Dynamic**
  - Validates software **functionality / performance**
  - Testing the software by **executing** it
  - Identifies **functional errors**, **performance issues**, **runtime errors**, and **missing functionality**



Static Testing Techniques

Analyzing Code Without Execution

Static Testing Techniques

- **Static techniques** improve quality and productivity

- STATIC
  - Review
    - Informal
    - Walkthrough
    - Peer Review
    - Inspections
  - Static Analysis
    - Data Flow
    - Control Flow
    - Cyclomatic Complexity

Static Testing Techniques (2)

- **Reviews**
  - A human investigator is the primary defect finder
  - **Informal Reviews**
    - Casual and often unstructured
  - **Walkthroughs**
    - Systematic and detailed examination of the product's logic, structure, and functionality
  - **Peer Reviews**
    - Focused feedback and improvement process conducted by colleagues working at a similar level
  - **Inspections**
    - Most formal reviews

Static Testing Techniques (3)

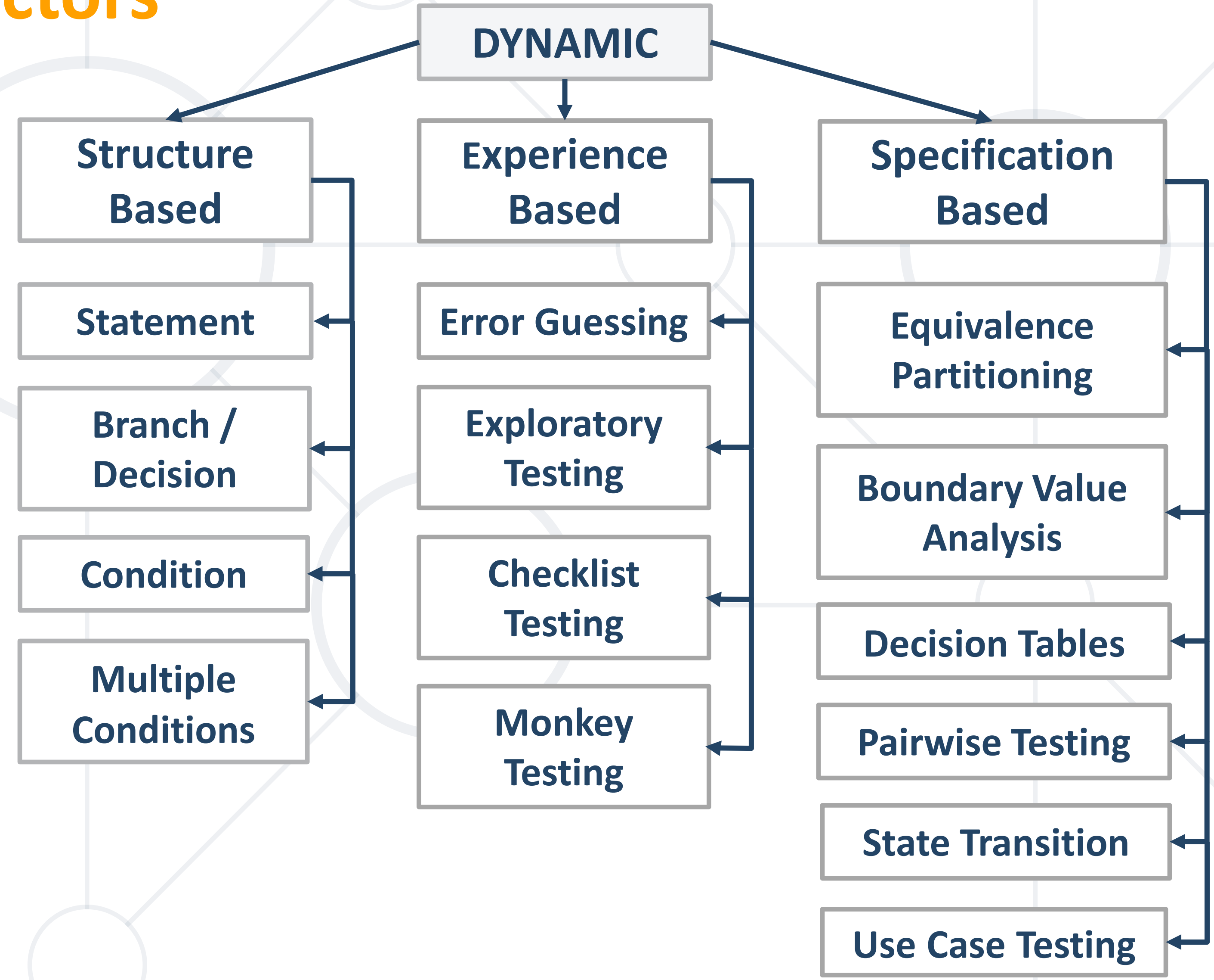
- **Static Analysis**
  - Software is examined without execution, focusing on detecting potential issues in the code, design, or documentation
  - **Data Flow**
    - Focuses on the paths and states data can take through a program, aiming to identify potential data-related errors
  - **Control Flow**
    - Examines the order in which individual statements, instructions, or function calls of a program are executed or evaluated
  - **Cyclomatic Complexity**
    - Counting the different routes a program's code can follow

Dynamic Testing Techniques

Testing in Action: Testing Through Execution

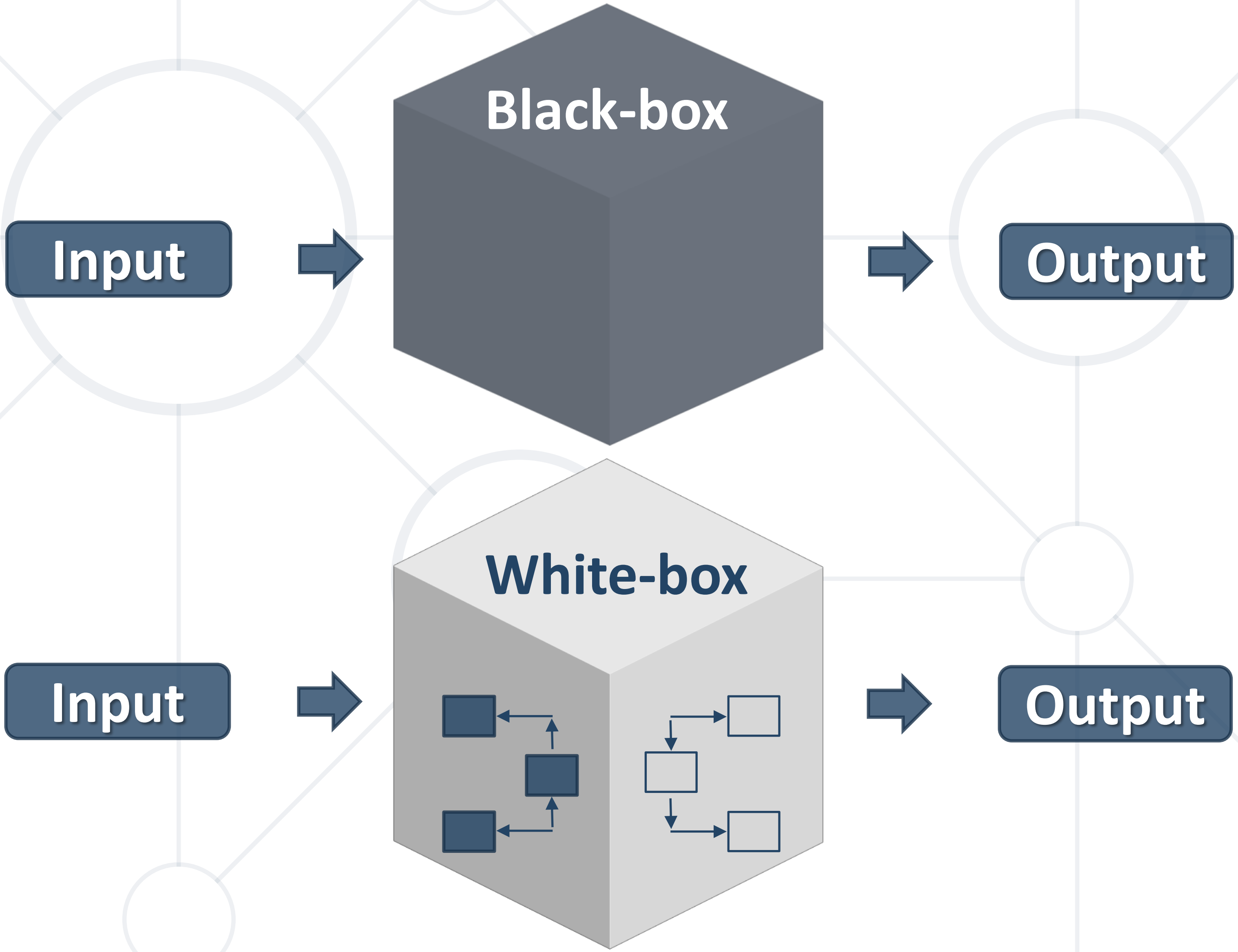
Dynamic Testing Techniques

- Based on **three factors**
  - Structure
  - Experience
  - Specification



Specification-Based vs Structure-Based

- Specification-Based techniques are **black-box**
- Structure-Based techniques are **white-box**



Black-Box, Grey-Box, White-Box

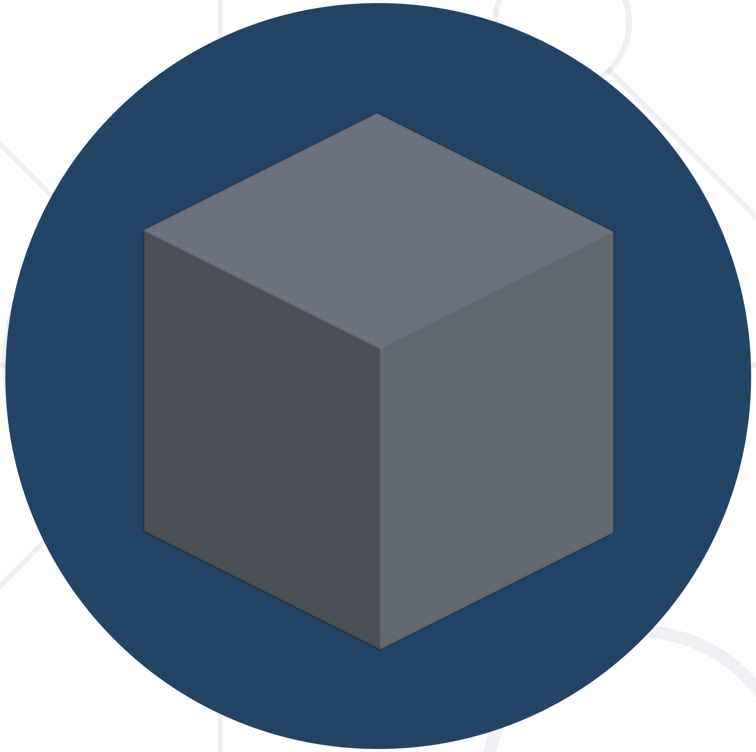
Software University

- Specification Based
- Equivalence Partitioning
- Boundary Value Analysis
- Decision Tables
- Pairwise Testing
- State Transition
- Use Case Testing

- Experience Based
- Error Guessing
- Exploratory Testing
- Monkey Testing

* Experience-based techniques can be considered as grey box testing techniques, but not all grey box testing techniques are experience-based

- Structure Based
- Statement
- Branch / Decision
- Condition
- Multiple Conditions



Black-Box Testing Techniques

Specification-Based Techniques

Equivalence Partitioning (EP)

- Divides the input data of a software unit into valid and invalid **partitions**
- Selects **representative values** from each partition
- Test cases are designed to cover **each** partition at least **once**
- Helps to **cut down** on the number of test cases
- Can be applied at any level of testing

Splitting Domains Into Partitions

- The operation of **equivalence partitioning** is performed by **splitting** a set (domain) into **two** or **more subsets**
  - All the members of each subset share some **trait in common**
  - This trait is **not shared** with the members of the other subsets

```
System should accept only numbers from 0 to 100
We have 3 subsets
Valid: 0 to 100 | Invalid: > 100 | Invalid: < 0
```

  - There must be at least one value selected from each subset

Boundary Value Analysis (BVA)

- **Boundary value analysis** is about testing the **edges** of equivalence classes
- It can be seen as an extension of equivalence partitioning
- A **boundary value** is
  - On the **edge** of an equivalence partition
  - The point where the **expected behavior** of the system **changes**

Boundary Values Explained

- The **primary focus** is on the **exact boundaries** of the **valid range**
- Given a **valid input** range of **100 000 to 999 999**, the boundary values would be:
  - **Lower boundary value**: 100 000
  - **Upper boundary value**: 999 999
- The **values** that are just outside this range are also of interest in BVA, as they test the system's response to input that is just **outside the valid range**:
  - **Just below lower boundary**: 99 999
  - **Just above upper boundary**: 1 000 000
- **Some methodologies** also consider the **values just inside the valid range** as part of the boundary testing, but these are not traditionally considered "boundary values":
  - **Just above lower boundary**: 100 001
  - **Just below upper boundary**: 999 998

Why Should This Work?

- A **common mistake** is using an incorrect operator or mismanaging indexes
- For example, using "**<**" instead of "**<=**" might seem a small error, but it can cause the system to behave incorrectly when processing the boundary value
- Because the edges of input ranges are the **points where** the **software changes its behavior**, they are places where bugs are often found
- If a software application is able to correctly **handle input at the edges** of its input ranges (i.e., its "edge cases"), it is likely to handle inputs within its range correctly as well

Decision Table Testing

- Decision tables testing connects **combinations** of **conditions with** the **actions** that should occur
- These actions are also called outputs or effects
  - Their combinations and permutations form the **decision table**
- This technique is also referred to as **"cause-effect"** table
- Often used in conjunction with equivalence partitioning

Problem: Credit Card

- You are a customer and you want to open a **credit card account**
- There are **three conditions**
  - You will get a **15%** discount on all your purchases today, if you are new customer
  - If you are an **existing customer** and you hold a **loyalty card** you get a **10%** discount
  - If you have a **coupon**, you can get **20%** off today
    - Coupons **can** be used together with a loyalty card
    - New customers **can use coupons**, but **not together** with a "new customer" discount
- Discount amounts are added **if applicable**

Steps to Solve Credit Card Problem

- **Go over the requirements**
- Pull out the conditions and start creating your first column
- Write out the **conditions and actions** in a list to get a **True** or **False** outcome
  - **Conditions:**
    - New customer (15%)
    - Loyalty card (10%)
    - Coupon (20%)
  - **Action:**
    - Discount Percentage

Steps to Solve Credit Card Problem (2)

- **Add all necessary columns**
- Figure out **how many columns** you'll need
- **Varies** depending on the **number of conditions** in your requirements
- For example, if you have two conditions, and each can have a true or false outcome, then you'll need four columns total
  - **1 condition = 2 columns**
  - **2 conditions = 4 columns**
  - **3 conditions = 8 columns**
- **Double the number of columns** you need for **each additional condition**
- It is **better** to have a lot of **small decision tables** instead of a couple of big ones
- That way, you avoid having your decision table **too large** to manage

Steps to Solve Credit Card Problem (3)

- **Shrink your table**
  - Find ways to **remove columns** that do not affect the outcome. That helps you eliminate redundancies
  - Next, **get rid** of any combinations **that appear invalid** or those that **can't happen** because of an internal conflict
  - Use an x or – symbol to indicate the removal of the column
  - Finally, **get rid** of any **duplicate columns**
- **Figure out your actions**
  - Once you've got your decision table into the correct format, start thinking of the actions that would result from each column
  - Give each column a name or identifier

Credit Card Solution

- **Decision table**

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	invalid	invalid	20	15	30	10	20	0

Pairwise Testing

Software University

- Also known as **All-Pairs testing**
- Handling the complexity of **testing multiple parameters** together
- Based on the observation that **most defects** in software are **caused by** either a **single factor** or an **interaction between pairs** of factors
- By testing **combinations of pairs** of parameters, we can still find most of the bugs
- Drastically **cuts down the number of test cases** while still maintaining reasonable test coverage

Pairwise Testing Explained

- We have **3 parameters**: **A**, **B** and **C**
- Each one can take the **values 1, 2 or 3**
- 3^3 = **27 combinations**
- Instead of testing all 27, pairwise testing **selects a subset of 9** test cases that **covers all pairs** of values
- **Each case covers a different combination** of pairwise values for parameters A, B, and C
- All combinations of pairs of values are **covered** in **at least one test case**

Pairwise Testing Example

- The table represents **nine test cases**
- Each case covers a **different combination** of pairwise values for parameters A, B, C
- All combinations of pairs of values are covered **in at least one test case**
- For example, the pair (A=1, B=2) is covered in test case 2, and the pair (A=2, C=3) is covered in test case 4, and so on
- Pairwise Testing Tool

Test Case	A	B	C
Test Case 1	1	1	1
Test Case 2	1	2	2
Test Case 3	1	3	3
Test Case 4	2	1	3
Test Case 5	2	2	1
Test Case 6	2	3	2
Test Case 7	3	1	2
Test Case 8	3	2	3
Test Case 9	3	3	1

State Transition Testing

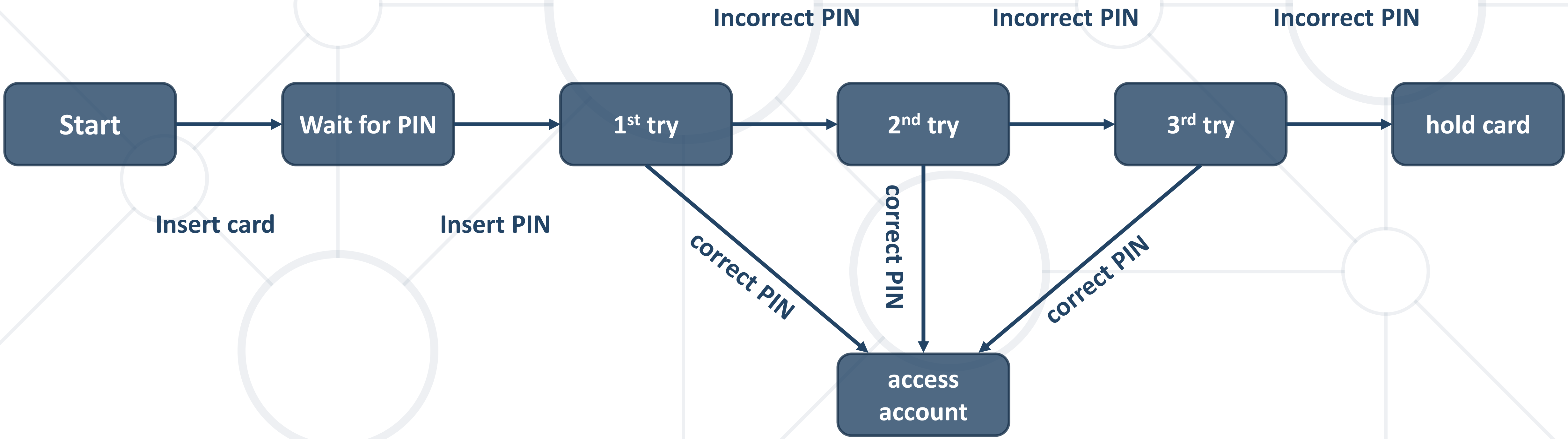
- A technique which is used when the system can be in a **finite number different states** and the transitions from one state to another needs to be tested
- Tests are designed to execute **valid** and **invalid state transitions**
- States of the system can be shown in a **state diagram** or **state table**

State Transition Model

- A **state transition model** has **four** basic **parts**
  - The **states** that the software may occupy
  - The **events** that cause a transition
  - The **transitions** from one state to another
  - The **actions** that result from a transition
- Simple light switch with two states: **ON and OFF**

Current State (State)	Input (Event)	Next State (Transition)	Resulting Action (Action)
Off	Flip Switch On	On	Light bulb turns on
On	Flip Switch Off	Off	Light bulb turns off

State Transition Diagram Example



State Transition Table Example

- Logging into an account

Current State (State)	Input (Event)	Next State (Transition)	Resulting Action (Action)
Logged Out	Correct Login Details	Logged In	User is Logged In
Logged Out	Incorrect Login Details, 1st Try	1st Attempt Failed	Warning message displayed
1st Attempt Failed	Incorrect Login Details, 2nd Try	2nd Attempt Failed	Warning message displayed
1st Attempt Failed	Correct Login Details	Logged In	User is Logged In
2nd Attempt Failed	Incorrect Login Details, 3rd Try	Account Locked	Account Locked message
2nd Attempt Failed	Correct Login Details	Logged In	User is Logged In
Account Locked	Any Login Details	Account Locked	Account still locked message
Logged In	Logout	Logged Out	User is Logged Out

What do We Expect to Find?

- **Spotting Unexpected Behavior**
  - State transition testing helps to identify when the system takes a wrong action or moves to an incorrect state in response to a particular event
- **Considering All Combinations**
  - Consideration of all possible combinations of states and their corresponding events or conditions
  - Ensuring comprehensive coverage and minimizing the risk of missed testing scenarios

What is a Use Case?

- A use case is a **description** of a **particular use** of the system by **an actor** (either a user or another system)
- Each use case **outlines** a **sequence of actions**, typically including variants, to achieve a specific goal or task
- Use cases capture **who** (**actor**) does **what** (**interaction**) with the system, for what **purpose** (**goal**), without dealing with how the system internally processes and responds to these interactions

Use-Case Testing

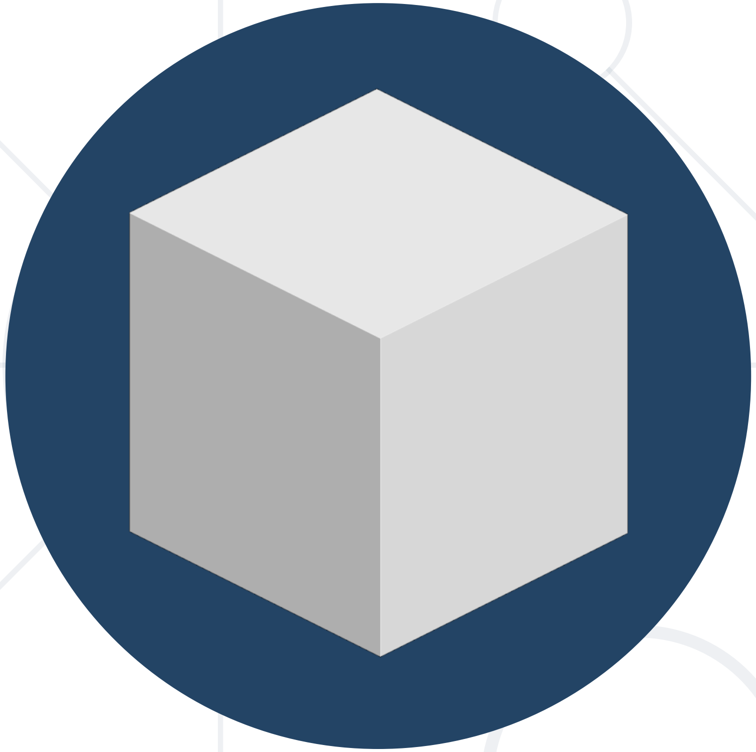
- Identifies and prepares tests to ensure that the system can handle a transaction from **start to finish**
- Beneficial in identifying **integration defects** and issues that could arise in **real-world scenarios**
- **Pre-conditions** in a use case are the conditions or requirements that must be met for the use case to start
- **Post-conditions** in a use case are the final conditions or state of the system once the use case has been completed
- Use-case testing is effective in ensuring that **all interactions** between the actors and the system **have been tested**
- Understanding the **system behavior** from the **user's point** of view and is especially beneficial in user acceptance testing

Use-Case Example: Bookstore

- **Use Case:** Purchase a book
- **Actor:** Customer
- **Precondition:** The customer has a registered account and is logged in; The book is in stock
- **Steps:** The customer searches for a book; Selects the desired book from the search results; Adds the book to the shopping cart; Proceeds to checkout; Enters shipping information; Selects a payment method and provides payment information; Confirms the order; The system processes the order and sends an order confirmation to the customer
- **Postcondition:** The book is marked as sold and its stock is reduced. The customer receives an order confirmation email. The order appears in the customer's order history

Use-Case Testing Example: Bookstore

- **Sample test cases derived from Purchase a book use case:**
  - Test the process with a customer who is not logged in
  - Test the process with a book that is not in stock
  - Test the search functionality with various inputs (book title, author name, etc.)
  - Test the process of adding a book to the shopping cart
  - Test the checkout process (entering shipping information, payment information, etc.)
  - Test the order confirmation process
  - Test the functionality of updating the book's stock after a purchase
  - Test the delivery of order confirmation email
  - Test the update of the customer's order history



White-Box Testing Techniques

Structure-Based Techniques

Structure-Based Techniques

Software University

- **White-box techniques**
- Test cases are chosen based on an analysis of the internal structure of a component or a system
- Aims to assess the amount of testing performed by specific tests, often in terms of code coverage
- After the initial set of tests are run and their coverage is analyzed, additional tests are designed to cover parts of the code that have not been tested yet
- The aim is to **increase** the test **coverage**

Coverage

- Test coverage is defined by the number of **items covered** in testing divided by the **total** number of **items**

```
                 Number of coverage items exercised
Coverage = -------------------------------- x 100%
                  Total number of coverage items
```

- The objective of testing is to achieve **maximum** code coverage
- **100% coverage** does **not** mean **100% tested**
- Measuring coverage requires tool support

Coverage Types

- Statement Coverage

```
                Number of statements exercised
Statement = ------------------------------ x 100%
Coverage          Total number of statements
```

- Branch/Decision Coverage

```
             Number of desicion outcomes exercised
Desicion = ------------------------------------ x 100%
Coverage       Total number of desicion outcomes
```

Example: Statement Coverage

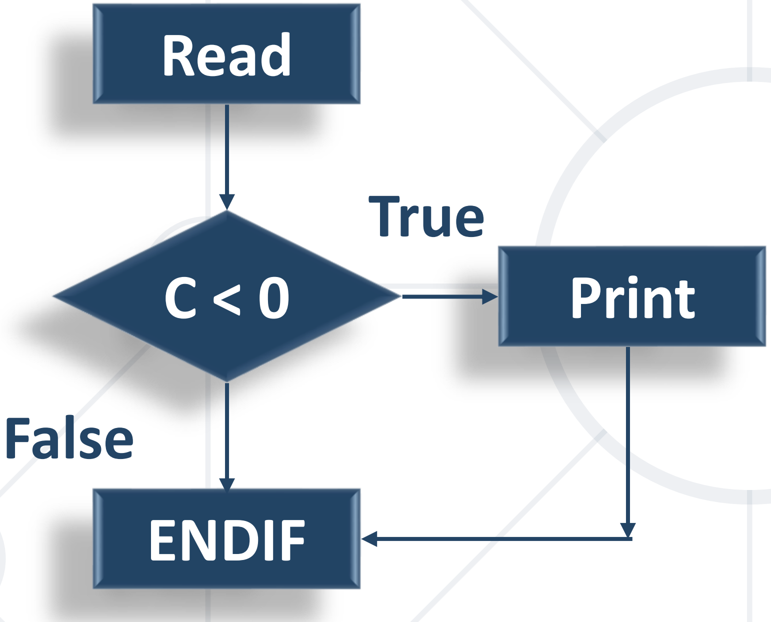
```
Code sample:

READ A
READ B
IF A > B THEN C = 0
ENDIF
```

- 100% statement coverage can be achieved with one test case
- It must ensure that A is greater than B, for example:

  A = 20, **B = 10**

Example: Branch/Decision Coverage




```
READ A
READ B
C = A-2*B
IF C<0 THEN
PRINT "C negative"
ENDIF
```

- We have a test that gives us **100% statement coverage** and covers the **"True"** outcome: **A = 20**, **B = 15**
- In order to cover the **"False"** outcome and achieve **100% branch/decision coverage**, we can use this test: **A = 10**, **B = 2**

Experience-Based Testing Techniques

Learning from Experience

Experience-Based Techniques

- Rely on the knowledge and expertise of the testers. The more experienced the tester, the more effective these techniques tend to be
- Offer **flexibility** as they are not constrained by a rigid testing plan
- Effectively target **known problem** areas and potential weaknesses in the system
- Can be used on their own, but **often complement** specification-based and structure-based testing techniques
- Useful in situations where the **system's documentation may be incomplete** or where the system is too complex

Experience-Based Techniques

- **Exploratory testing**
  - Minimum planning, maximum test execution
  - Most useful when there are no specifications or time is severely limited
- **Error guessing**
  - It should be used in addition to other more formal techniques
  - The tester thinks of situations, which could be problematic for the software

Experience-Based Techniques

- **Checklist-based Testing**
  - A checklist of common issues and areas to test, based on previous experience, is used to guide the testing process
  - The checklist can be built based on common issues found in similar systems, known problem areas, or general good testing practices
- **Random Testing or Monkey Testing**
  - Inputs to the system are generated randomly with the goal of finding hard-to-discover bugs
  - The "smartness" of the monkey can vary
    - "Dumb" monkeys inputting totally random data
    - "Smart" monkeys using some knowledge of the system to guide their random input

Choosing a Test Technique

Choosing a Test Technique

- Each technique is **good** for a certain situation and **not good** for other
- Structure-based are good at finding **errors** in the **code**
- Specification-based are good for finding **missing** parts of the **specification from the code**
- Experience-based are proper when there is both **missing parts of code** and **missing specification**
- **Each** individual **technique** is aimed at **particular** types of **defect**

Factors for Choosing

- **Choosing** the appropriate testing techniques is based on some **factors**
  - Development life cycle
  - Use case models
  - Type of system
  - Level and type of risk
  - Test objective
  - Time and budget
  - Tester's experience

Summary

- There are **static** and **dynamic** testing techniques
  - Static techniques include reviews which increase quality and productivity
  - Dynamic techniques are based on three factors – **structure**, **specification**, **experience**
- Behavior-based techniques are called **black box techniques**
- Structure-based techniques are called **white box techniques**
- **Experience-based** techniques
- Choosing a technique is done according to the parts of the system that need to be tested

Questions?

Software University

SoftUni

- Software University
- SoftUni Creative
- SoftUni Digital
- SoftUni Foundation
- SoftUni Kids
- Finance Academy

SoftUni Diamond Partners

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg