Sri Lanka Institute of Information Technology

SOFTWARE ENGINEERING PROCESS AND QUALITY MANAGEMENT

Lecture 7 – Software Metrics (Weighted Composite Complexity Metric)

Weighted Composite Complexity (WCC)

- Measure the complexity of the program.
- Object Oriented Metric
- Based on 4 key factors
  - Size
  - Type of control structures
  - Nesting level of control structures
  - Inheritance level of statements

Computing the WCC Value

- WCCM value of a program = $\sum_{j=1}^{n} S_j * (W_t)_j$
  - Sj = Size of j^{th} executable statement in terms of token count
  - n = Total number of executable statements in a program
  - (Wt)j = Total weight of the j^{th} executable statement in the program

- Wt = Wc + Wn + Wi
  - Wc = Weight due to type of control structures
  - Wn = Weight due to nesting level of control structures
  - Wi = Weight due to inheritance level of statements

Identify the Size of a Statement

The **Size (S)** of a statement is the **total number of tokens** it contains.

In WCC, a **token** is a fundamental program element used to measure the **size** of a statement.

However, **not everything in the code is a token**.

Refer to the guidelines document to identify the tokens in a program statement. (uploaded in Courseweb)

Identify the tokens and the size value of each statement of the following program

Line No	Program Statements
1	public class Result{
2	public void outresult(int marks) {
3	if (marks > -1 && marks < 50)
4	System.out.println("Fail");
5	else
6	System.out.println("Pass");
	}
7	public static void main(String args[ ]){
8	Result r = new Result( );
9	r.outresult(50);
	}
	}

Line No	Program Statements	Tokens	Size (S)
1	public class Result{		
2	public void outresult (int marks) {	void, outresult( )	2
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8
4	System.out.println("Fail");	System, ., out, ., println( ), "Fail"	6
5	else		
6	System.out.println("Pass");	System, ., out, ., println( ), "Pass"	6
	}		
7	public static void main(String args[ ]){	void, main( )	2
8	Result r = new Result( );	Result, r, =, new, Result( )	5
9	r.outresult(50);	r, ., outresult( )	3
	}		
	}		

Weight due to Type of Control Structure (Wc)

Type of control structure	Weight
Sequential	0
Branch	1
Iterative	2
Switch statement with **n** cases	n

Line No	Program Statements	Tokens	S	Wc
1	public class Result{			
2	public void outresult (int marks) {	void, outresult( )	2	0
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0
5	else			
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0
	}			
7	public static void main(String args[ ]){	void, main( )	2	0
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0
9	r. outresult(50);	r, ·, outresult( )	3	0
	}			
	}			

Weight due to Nesting level of Control Structure (Wn)

Nesting Level of Statements	Weight
Sequential statements	0
Statements inside the outer most level/first level of control structures	1
Statements inside the second level control structures	2
Statements inside the third level control structures	3
Statements inside the n^{th} level control structures	n

Line No	Program Statements	Tokens	S	Wc	Wn
1	public class Result{				
2	public void outresult (int marks) {	void, outresult( )	2	0	0
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1	1
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0	1
5	else				
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0	1
	}				
7	public static void main(String args[ ]){	void, main( )	2	0	0
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0	0
9	r. outresult(50);	r, ·, outresult( )	3	0	0
	}				
	}				

Weight due to Inheritance level of Statements (Wi)

Inheritance Level of Statements	Weight
Statements inside the base class/root class	0
Statements inside the first derived class	1
Statements inside the second derived class	2
Statements inside the n^{th} derived class	n

Line No	Program Statements	Tokens	S	Wc	Wn	Wi
1	public class Result{					
2	public void outresult (int marks) {	void, outresult( )	2	0	0	1
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1	1	1
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0	1	1
5	else					
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0	1	1
	}					
7	public static void main(String args[ ]){	void, main( )	2	0	0	1
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0	0	1
9	r. outresult(50);	r, ·, outresult( )	3	0	0	1
	}					
	}					

Total Weight (Wt)

Total Weight of a statement. It combines three dimensions of complexity into one value:

$$Wt = Wc + Wn + Wi$$

Wc = Weight due to Type of Control Structure
Wn = Weight due to Nesting Level
Wi = Weight due to Inheritance Level

Line No	Program Statements	Tokens	S	Wc	Wn	Wi	Wt
1	public class Result{						
2	public void outresult (int marks) {	void, outresult( )	2	0	0	1	1
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1	1	1	3
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0	1	1	2
5	else						
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0	1	1	2
	}						
7	public static void main(String args[ ]){	void, main( )	2	0	0	1	1
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0	0	1	1
9	r. outresult(50);	r, ·, outresult( )	3	0	0	1	1
	}						
	}						

Weighted Complexity for a Single Statement (WC)

It tells us how complex that one line is by factoring in:

- How many tokens it contains (Size S)
- How "heavy" or complex its context is (Total Weight Wt)

$$WC = S \times Wt$$

Line No	Program Statements	Tokens	S	Wc	Wn	Wi	Wt	WC
1	public class Result{							
2	public void outresult (int marks) {	void, outresult( )	2	0	0	1	1	2
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1	1	1	3	24
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0	1	1	2	12
5	else							
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0	1	1	2	12
	}							
7	public static void main(String args[ ]){	void, main( )	2	0	0	1	1	2
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0	0	1	1	5
9	r. outresult(50);	r, ·, outresult( )	3	0	0	1	1	3
	}							
	}							

Weighted Composite Complexity for the Program (WCC)

The final WCC is the sum of all WC values for each line.

Line No	Program Statements	Tokens	S	Wc	Wn	Wi	Wt	WC
1	public class Result{							
2	public void outresult (int marks) {	void, outresult( )	2	0	0	1	1	2
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1	1	1	3	24
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0	1	1	2	12
5	else							
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0	1	1	2	12
	}							
7	public static void main(String args[ ]){	void, main( )	2	0	0	1	1	2
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0	0	1	1	5
9	r. outresult(50);	r, ·, outresult( )	3	0	0	1	1	3
	}							
	}							
		WCC Value						**60**

THANK YOU!

Software Testing Life Cycle

(STLC)

10th April 2025

Mr. Suresh Fernando

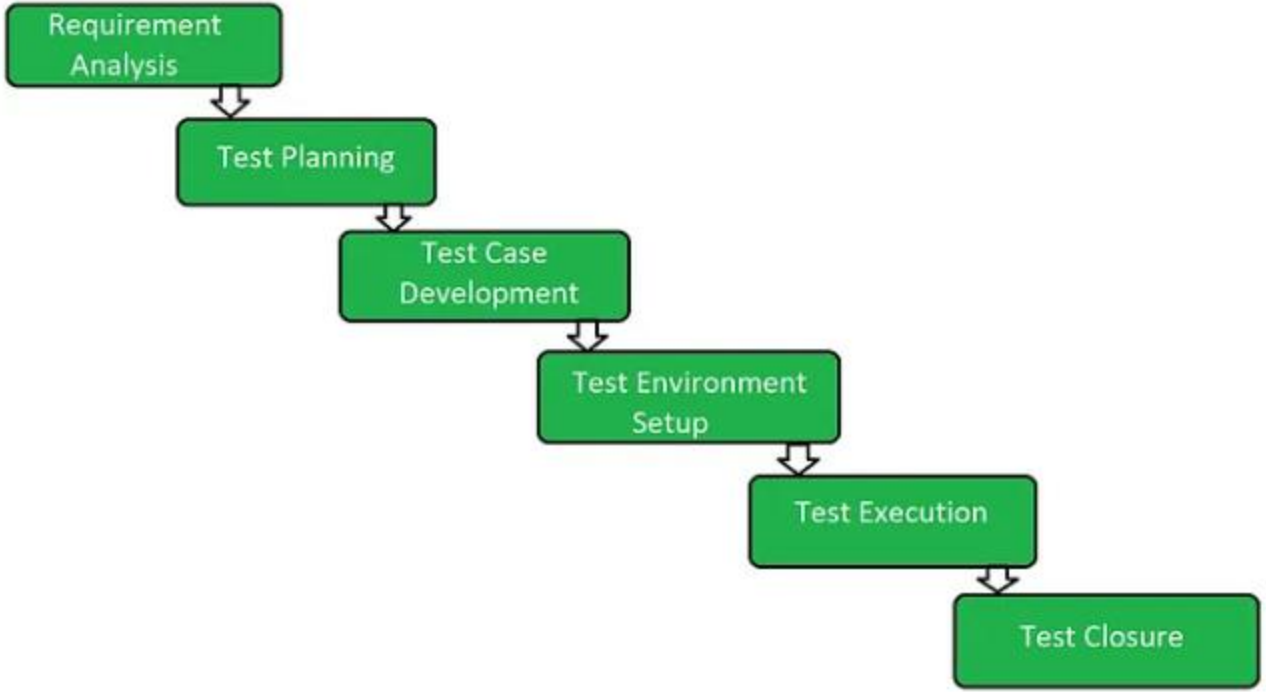
Software Testing Lifecycle- Overview

- Introduction to STLC
- SDLC vs STLC
- Requirement Analysis
- Test Planning
- Test Case Development
- Test Environment Setup
- Test Execution
- Test Cycle Closure
- Challenges in STLC
- Quiz Time

Introduction to STLC

- STLC is a **systematic approach** to testing a software application to ensure that it **meets the requirements** and is **free of defects.**
- Follows a **series of steps** or phases with specific objectives
- Fundamental part of **SDLC**
- Why STLC is Important?
  - Helps in finding defects early
  - Ensures a systematic approach to testing.
  - Makes testing measurable and repeatable.
  - Supports better planning, coverage, and quality control.

Stages of STLC



Why is it called a Cycle?

because testing is rarely a one-way street

feedback loops, rework, iterations, and continuous improvement make it cyclic

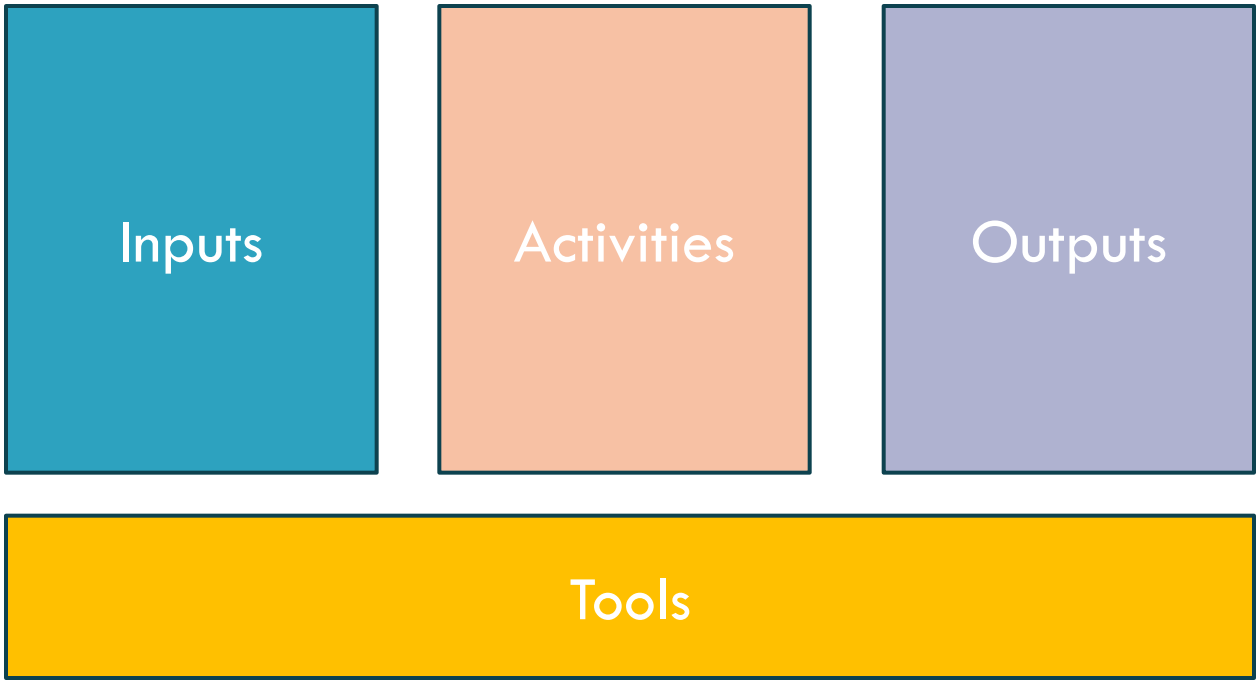
Key Characteristics of SDLC

- Phased Approach
- Goal-Oriented
- Process-Driven
- Early Defect Detection
- Improves Quality
- Traceability
- Reusability

STLC vs SDLC

Aspect	SDLC > Develop Software	STLC > Test Software
Goal	To build a functioning, high-quality application	To ensure the software works as expected and is bug-free
Focus Area	Covers the entire development (requirements to deployment)	Covers only testing-related activities
Who Performs It	Developers, Architects, Business Analysts, DevOps, etc.	Testers / QA team
Starts When	At the beginning of the software project	When requirements are ready
Output/Deliverables	Working software, system architecture, code, documentation	Test cases, bug reports, test summary reports
End Result	A product ready for release	Verified and validated software

STLC Phase



Phase 1 – Requirement Analysis

Phase 1 – Requirement Analysis

- **Inputs**
  - Business Requirements Document (BRD)
  - Functional Requirements (FRS)
  - Meetings with stakeholders
- **Activities**
  - Analyze requirements for testability
  - Identify types of testing needed
  - Review risks and priorities
  - Identify gaps or missing areas

Phase 1 – Requirement Analysis

- **Outputs**
  - Requirement Traceability Matrix (RTM)
  - Test automation feasibility
  - Clarification questions or assumptions
- **Tools**
  - JIRA, Confluence (for documentation & tracking)
  - Excel, RTM tools (e.g. Jama Connect)
  - Requirement analysis templates

Requirement Traceability Matrix (RTM)

A document that maps each requirement to its corresponding design, development, and testing elements.

REQUIREMENT TRACEABILITY MATRIX							
Project Name:	E-commerce Application						
Project ID:	112						
Business Requirmeent Document (BRD)		Functional Requirements Document (FSD)		Test Case Document			
BR_ ID	BR_User Case	FR_ID	FR_User Case	Priority	Test Case ID	Status	Comments
BR_1	Product Listing	FR_1	Sort by	High	TC_001 TC_002 TC_004	Finished	Dec 1: Testing started Dec 6: Defect reported Dec 12: Defect Fixed Dec 15: FS_ Passed
		FR_2	Filters	High	TC_001 TC_002 TC_003	Finished	Dec 1: Testing started Dec 6: defect reported Dec 12: Defect Fixed Dec 15: FS_ Passed
BR_2	Payment Module	FR_3	By Credit Card	High	TC_005	In Progress	Dec 1: Testing started
		FR_4	By Debit Card	High	TC_006	In Progress	Dec 1: Testing started
		FR_5	By Reward/Referral P	Medium	TC_007 TC_008	Not Started	

Phase 2 – Test Planning

- **Inputs**
  - Requirement documents
  - RTM
  - High-level project plan
  - Risk assessment
- **Activities**
  - Define test scope
  - Identify test strategy and approach
  - Estimate time, effort, and resources
  - Assign roles and responsibilities

Phase 2 – Test Planning

- **Outputs**
  - Test Plan document
  - Effort estimation sheet
  - Resource plan
- **Tools**
  - TestRail, Xray
  - Microsoft Project, Excel
  - Risk analysis tools

Test Plan Template

TEST PLAN

TEMPLATE

FEATURES TO BE TESTED (IN-SCOPE)

- Sign-up/Sign-in
- Forget Password
- Delete Account

FEATURES NOT TO BE TESTED

- Edit Account Information
- Create Multiple Accounts

TEST LEVELS AND TEST TYPES

Test Levels
- System Testing
- Acceptance Testing

Test Types
- Functional Testing
- Usability Testing
- Regression Testing

ESTIMATION

- Sign-up/Sign-in: **6 hours**
- Forget Password: **1 hour**
- Delete Account: **1 hour**

STAFFING AND TRAINGING

Staffing
- Manual Tester: Anees, Tester 2, ...
- Automation Tester: Tester 3, ...

Training
- TestZephyr: **16 hours**

ASSUMPTIONS

- Developers (Backend + Frontend) deliver the code related to the features
- License for TestZephyr

EXIT CRITERIA

- Testing is finished and there are no functional bugs
- All remaining bugs have low severity
- No more than 10% of medium-severity bugs are open

SUSPENSION CRITERIA

- Critical Bugs are open and they are blocking testing
- All remaining test cases are blocked by an open bug

TEST DELIVERABLES

- Test Cases
- Bugs Report
- Test Summary Report

TEST ENVIRONMENT

- Operating System: Windows 10
- Server: QA - Staging
- Browser: Google Chrome - The last available version
- Network: Wi-Fi

RISKS

List of Risks
- Risk 1 = Late Delivery for Features
- Risk 2 = QA Environment is down
- Risk 3 = Un-planned vacations
- Risk 4 = Critical Bugs keep showing which affect the time frame

Risk Mitigation
- Risk 1 = Risk Acceptance
- Risk 2 = Risk Transfer
- Risk 3 = Risk Monitoring
- Risk 4 = Risk Acceptance

TEST REFERENCES

- User Stories
- Figma Design
- System Design

Phase 3 – Test Case Development

- **Inputs**
  - Requirement documents
  - Test Plan
  - RTM
- **Activities**
  - Write test cases & test scripts
  - Review and baseline test cases
  - Create test data

Phase 3 – Test Case Development

- **Outputs**
  - Test cases
  - Test scripts
  - Test data
  - Reviewed and approved test cases
- **Tools**
  - TestLink, Zephyr, TestRail
  - Excel, Word
  - SQL (for test data), Selenium (for test scripts)

Test Scenario vs Case vs Script vs Data

Test Scenario	Test Case	Test Script	Test Data
A high-level idea of what needs to be tested. **Example:** *Verify that the user can successfully log in with valid credentials.*	A detailed set of steps and expected results to validate a specific part of the scenario. **Example:** **Title:** Login with valid username and password **Steps:** 1. Navigate to the login page. 2. Enter a valid username. 3. Enter a valid password. 4. Click the "Login" button. **Expected Result:** User is redirected to the dashboard.	An automated or manual script that performs the steps in the test case. **Example (Manual):** A document listing each step to follow. **Example (Automated - Selenium in Python):** driver.get("https://example.com/login") driver.find_element(By.ID, "username").send_keys("testuser") driver.find_element(By.ID, "password").send_keys("password123") driver.find_element(By.ID, "login").click() assert "Dashboard" in driver.title	The input values used during testing. **Example:** **Username:** testuser **Password:** password123

Phase 4 – Test Environment Setup

- **Inputs**
  - Environment requirements document
  - Software and hardware specs
- **Activities**
  - Set up testing hardware/software
  - Configure test servers, networks
  - Validate setup

Phase 4 – Test Environment Setup

- **Outputs**
  - Environment ready for testing
  - Test Environment checklist
  - Access and credentials
- **Tools**
  - Cloud platforms (AWS, Azure)
  - Configuration management tools

Different Test Environments

Development Environment	Integration Test Environment	System Test Environment
UAT (User Acceptance Testing) Environment	Staging / Pre-Production Environment	Performance / Load Testing Environment
Security Testing Environment	Mobile/Device Test Environment	Sandbox Environment

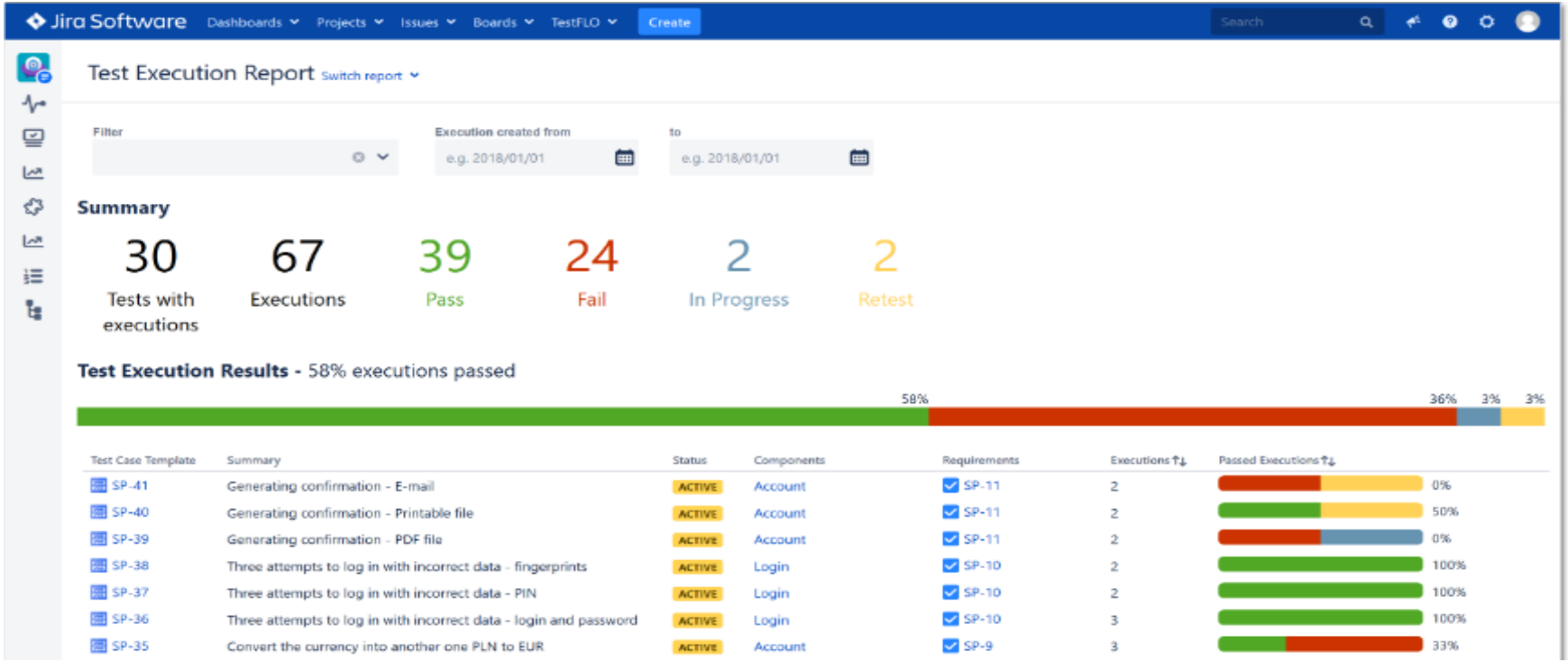
Phase 5 – Test Execution

- **Inputs**
  - Approved test cases
  - Test data
  - Test environment
- **Activities**
  - Execute test cases
  - Log defects/bugs
  - Retest after fixes
  - Update test results

Phase 5 – Test Execution

- **Outputs**
  - Test execution report
  - Defect Report
  - Updated RTM
- **Tools**
  - Selenium, Appium (automation)
  - JIRA, Bugzilla, Mantis (bug tracking)
  - TestRail, Zephyr

Test Execution Report



Phase 6 – Test Cycle Closure

- **Inputs**
  - Test execution results
  - Defect reports
- **Activities**
  - Evaluate test completion criteria
  - Analyze test coverage, defect density
  - Document lessons learned
  - Archive test artifacts

Phase 6 – Test Cycle Closure

- **Outputs**
  - Test Summary Report
  - Test metrics & closure checklist
  - Lessons learned document
- **Tools**
  - Excel/Google Sheets (for reporting)
  - Reporting tools (QMetry, TestRail)
  - Confluence, SharePoint (for documentation)

Test Metrics

Process

- Test Case Effectiveness
- Cycle Time
- Defect Fixing Time

Product

- Number of Defects
- Defect Severity
- Passed/Failed Test Cases

Project

- Test Coverage
- Cost of Testing
- Budget/Schedule Variance

SDLC - In Summary

Requirement Analysis	→	Understand what to test
Test Planning	→	Plan how to test
Test Case Development	→	Write test steps
Test Environment Setup	→	Get systems ready
Test Execution	→	Run tests, find bugs
Test Closure	→	Wrap up and report

Challenges in STLC

- Unclear or Changing Requirements
- Time Constraints
- Lack of Collaboration
- Environment Issues
- Tooling and Automation Challenges
- Frequent Scope Creep or Last-Minute Changes
- Knowledge Gaps / Inexperienced Testers

Quiz Time

Join at menti.com | use code 7234 9121

Mentimeter

Instructions

Go to

www.menti.com

Enter the code

7234 9121

Or use QR code

Thank You

30

Q & A

suresh.n@sliit.lk | 755841849

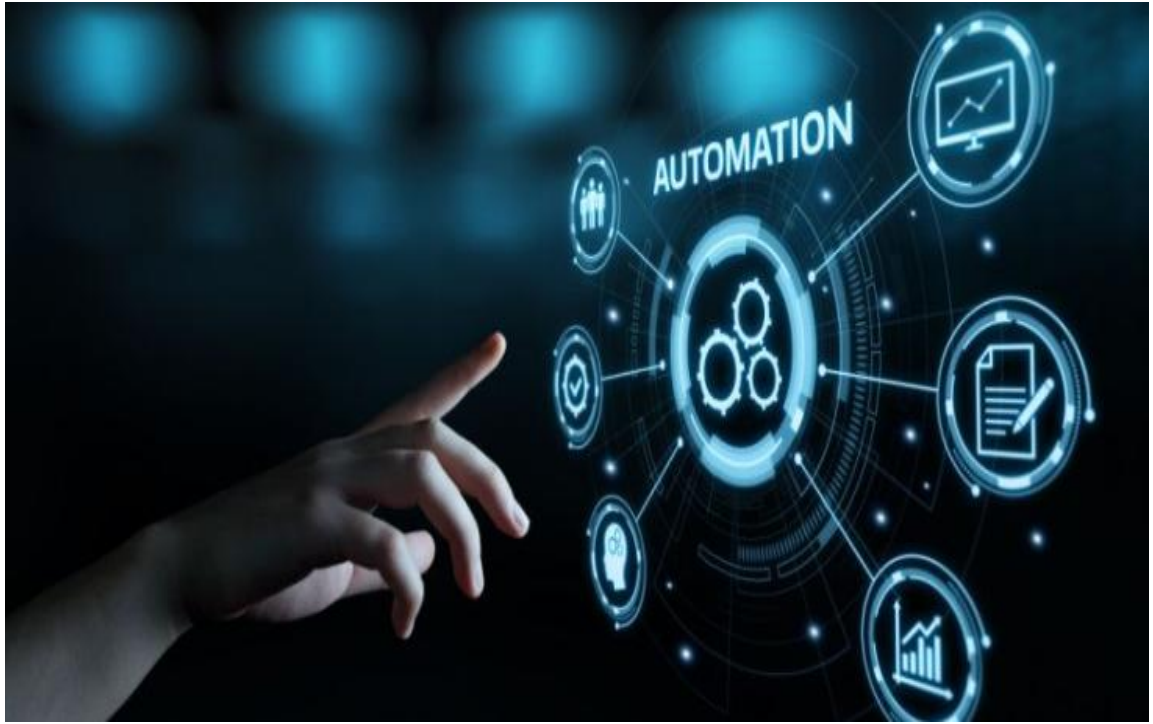
19th April 2025

Mr. Suresh Fernando

Test Automation - Overview

- Introduction to Test Automation
- Manual Vs Automated Testing
- Principles of Test Automation
- Test Automation Lifecycle
- Test Automation Frameworks
- Challenges & Myths in Test Automation
- Assignment 2

What is Automation?



Making an apparatus, a process, or a system operate automatically.

What is Test Automaton?

Test automation is the use of software tools to automatically run and validate test cases, enhancing testing efficiency...

What/When to Automate?

Regression Testing
Run **frequently** to ensure that new code hasn't broken existing functionality (e-com app).

Smoke Testing / Sanity Checks
Quick checks to see if the basic functionalities work before deeper testing

High Volume / Repetitive Tests
Saves time and reduces human error (load testing)

Data-Driven Testing
Same test logic with multiple input data sets (500 combinations of inputs like names, emails, and contact number.)

Cross-Browser / Cross-Device Testing
Tools can quickly test web apps on different browsers/device

Stable Features
Automate tests for features that don't change often (Login/Logout)

What/When **Not** to Automate?

Exploratory Testing
Requires human intuition, creativity, and observation (to find visual bugs)

Short-Lived Features
Not worth it for features that will be removed soon (a holiday sale)

Unstable or Frequently Changing UI
Tests will break often, causing more maintenance than value (Promotional page)

Usability Testing
Needs human feedback on look, feel, and ease of use (New color scheme)

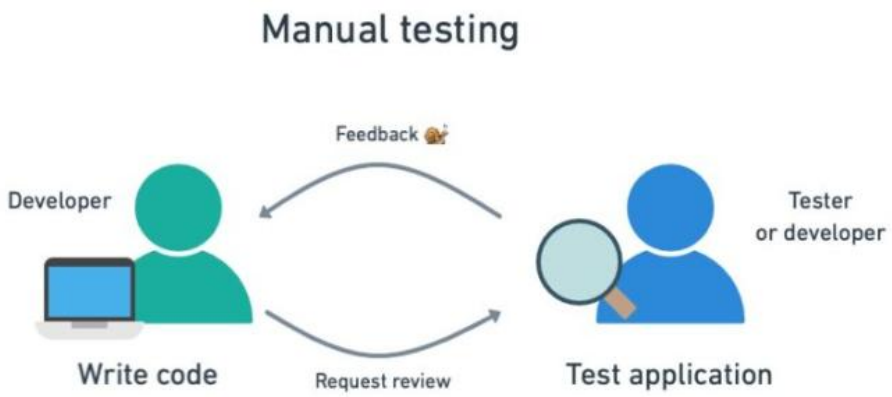
Tests That Run Only Once
The effort to automate it outweighs the benefit (One-time data migration)

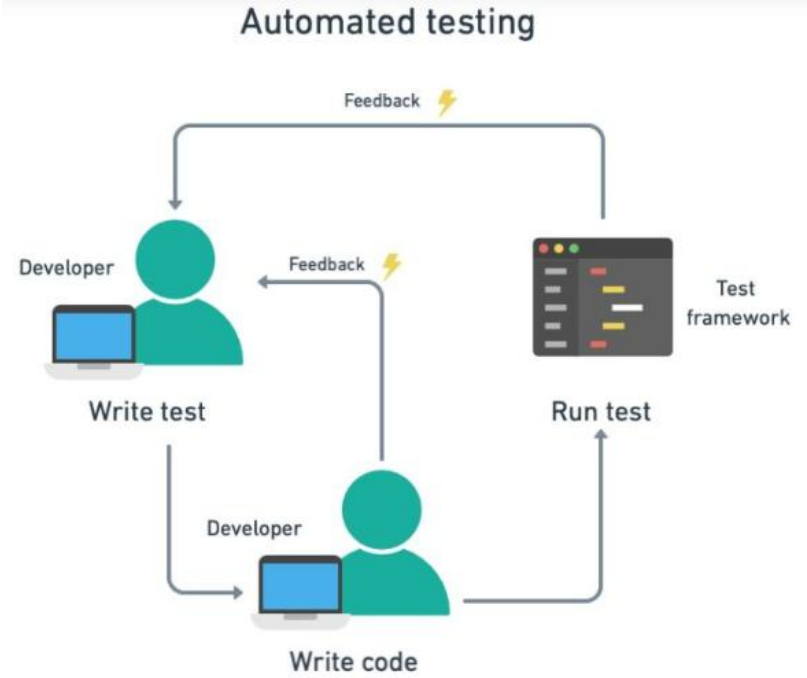
Complex Logics Involved
Some things are just too delicate to automate easily (Image comparison, CAPTCHA)

Benefits of Test Automation

- Application-wise
  - Improved Quality
  - Improved Accuracy
  - Enhanced Test Coverage
  - Early Bug Detection
  - Reusable Test Scripts
- Cost-wise

Manual vs Automated Testing





Manual vs Automated Testing

Key Differences	Manual Testing	Automated Testing
Execution Time	Requires more time as tests are performed sequentially, increasing resource usage.	Faster execution due to automated tools, leading to quicker product delivery with less resource consumption.
Inital setup	Less effort is needed in the initial setup as tests are executed manually.	Initially, more effort is required to create and maintain scripts and tools.
Reliability	Less reliable due to potential human error, impacting precision.	Higher reliability as tests are consistently executed the same way each time.
Programming	Non-programmable, limiting the creation of complex tests for defect detection.	Allows for programming complex tests to uncover hidden defects.

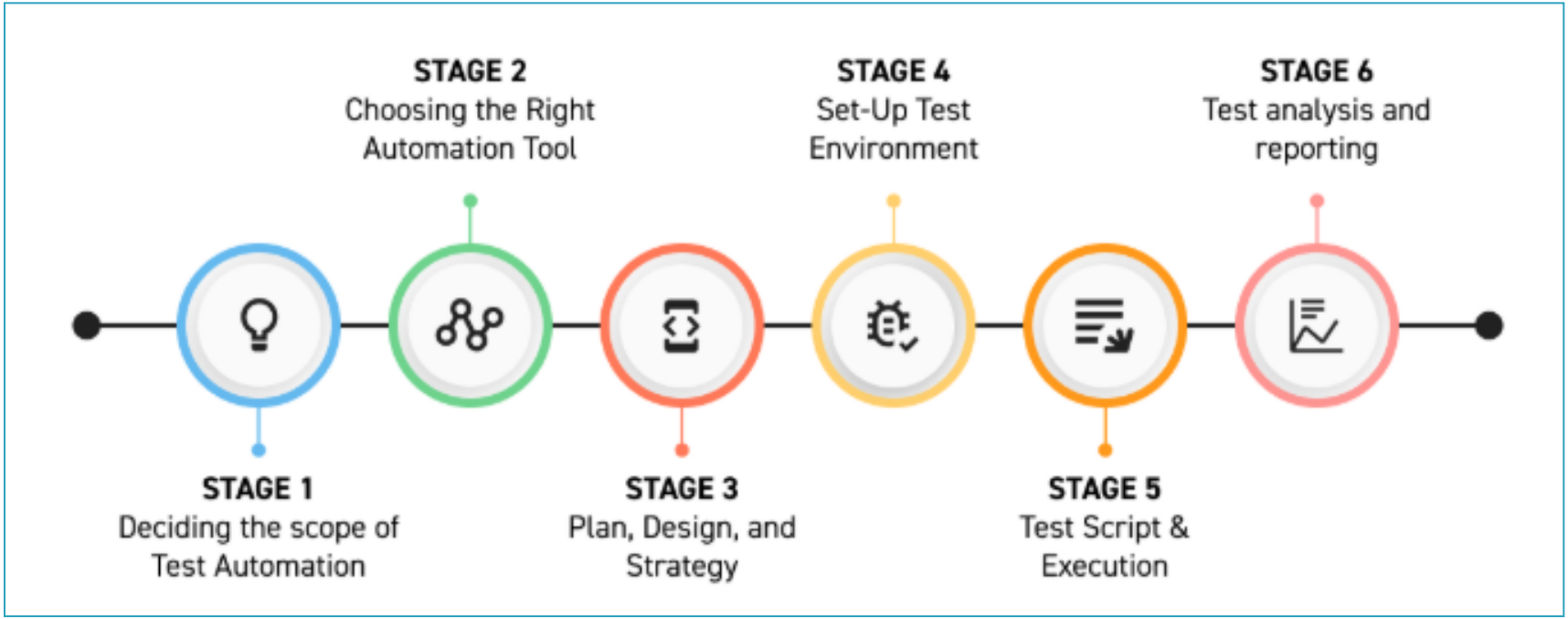
Manual vs Automated Testing

Key Differences	Manual Testing	Automated Testing
Reusability	Often requires new test cases for each function.	Test scripts can be reused across different software cycles, enhancing efficiency.
Reporting	Reporting can vary as it is manually implemented.	Ensures standardized, consistent tracking and reporting.
Flexibility	More adaptable to changing requirements and testing scenarios.	Less flexible to unexpected changes due to its pre-programmed nature.

Principles of Test Automation

- Tests should improve quality.
- Tests should reduce the risk of introducing failures.
- Testing helps to understand the code.
- Tests must be easy to write.
- A test suite must be easy to run.
- A test suite should need minimal maintenance.

Test Automation Lifecycle



Scope of Test Automation

- The first step in automation is figuring out what to automate
- Decide *what* and *why* to automate
- Find which can be suitable for automation based on:
  - how complex they may be
  - how often they get run
  - and how vital they may be.
- Focus on automating tests for important functions or matters that humans are not good at.
- Work with designers, developers, QA engineers, and others to decide what to automate.

Choosing Right Automation Tools

- Choose the right automation tool(s) for your tech stack and team skills
- Must do a comparative study of automation tools before making a decision.
- Factors to consider
  - Technology Support: Web, Desktop, Mobile
  - Scripting or Programming Requirements: Code, No-Code, Low-Code
  - Open-source vs. Licensed
  - Ease of Use
  - Adoption Time
  - Customer Support
  - End-to-End Testing

Automation Tools

Automation Tools

Tool	Scripting Languages Supported	Platforms Supported	Pricing
Selenium	Java, C#, Python, Ruby, JavaScript, Kotlin	Web (Desktop/Mobile via Appium)	Free (Open Source)
Appium	Java, Python, Ruby, JavaScript, Kotlin	Mobile (iOS, Android), Web (Mobile)	Free (Open Source)
Playwright	JavaScript, TypeScript, Python, C#, Java	Web (Desktop, Mobile)	Free (Open Source)
Cypress	JavaScript, TypeScript	Web (mostly Chrome-based browsers)	Free, Paid for dashboard
TestComplete	JavaScript, Python, VBScript, DelphiScript	Web, Desktop, Mobile	Paid
Ranorex	C#, VB.NET	Web, Desktop, Mobile	Paid
Katalon Studio	Groovy (based on Java), supports low-code	Web, Mobile, API, Desktop	Free basic, Paid plans
Robot Framework	Python, also supports Java and others via libs	Web, Mobile (via Appium), APIs	Free (Open Source)
TestCafe	JavaScript, TypeScript	Web (Desktop)	Free (Open Source)
QTP/UFT One	VBScript	Web, Desktop, Mobile	Paid

Automation Tools

Tool	Web (Desktop)	Web (Mobile)	Mobile Apps	Desktop Apps
Selenium	✔	⚠ (via Appium)	⚠ (via Appium)	✘
Appium	✘	✔	✔	✘
Playwright	✔	✔ (emulated)	✘	✘
Cypress	✔	✔ (emulated)	✘	✘
TestComplete	✔	✔	✔	✔
Ranorex	✔	✔	✔	✔
Katalon Studio	✔	✔	✔	✔
Robot Framework	✔	✔ (via libs)	✔ (via Appium)	✘
TestCafe	✔	✔ (limited)	✘	✘
UFT One	✔	✔	✔	✔

Automation Tools

- **Best for Web Automation**: Playwright, Selenium, Cypress
- **Best for Mobile Testing**: Appium, Katalon Studio
- **Beginner Friendly**: Katalon, TestCafe, Cypress
- **Enterprise-Grade Tools**: TestComplete, UFT One, Ranorex
- **Open-Source Leaders**: Selenium, Appium, Robot Framework, Playwright

Automation Tools

Front-end

***TestSigma**: Easy-to-use tool for each tech and non-tech users.*

***Selenium**: Popular open-sourse tool for web browsers.*

***TestComplete**: A commercial tool supporting multiple platforms.*

Performance

***JMeter**: Open-source tool for trying out web apps with big person hundreds.*

***LoadRunner**: Commercial tool for web / mobile apps.*

***Gatling**: Open-source tool for web apps with real-time reporting.*

Database

***dbForge Studio**: Commercial tool for SQL databases.*

***SQLTest**: Commercial tool for SQL databases*

Database Benchmark: An open-source .NET tool designed to stress test databases with large data flows.

Plan, Design and Strategy

Activity	Description	Example
Define Automation Objectives	Clarify what you want to achieve with automation	Speed up regression testing; improve test coverage
Finalize Scope	Decide **what to automate** and what to leave out	Automate login, checkout flow; skip UI animations
Select Tools & Frameworks	Choose suitable tools for tech stack & platforms	Selenium + TestNG for web app; Appium for mobile
Assess Skills & Resources	Understand team capability, skill gaps, and training needs	QA team needs Python training; hire automation engineer
Develop Automation Strategy	High-level strategy: test levels, CI/CD, data management, reporting, test maintenance plan	Run tests nightly in CI, Use page object model for maintainability, Report via Allure in GitHub Actions
Estimate Timeline & Cost	Time and cost estimates for implementation	Initial setup = 2 weeks; monthly maintenance = 8 hrs
Define KPIs & Metrics	How success will be measured	Execution time, pass rate, failure rate, test coverage

Setup Test Environment

- **Application Under Test (AUT):** Deployed version of the app being tested
- **Test Tools/Frameworks:** Tools for writing/executing tests
- **Test Data:** Structured data needed for test scenarios
- **Browsers/Devices:** Configured devices, browsers, or emulators
- **Environment Configs:** Variables, credentials, API keys, database info
- **CI/CD Integration:** Connection with CI to run tests automatically
- **Monitoring & Logs:** Logs and dashboards for debugging test results

Test Script & Execution

- Write test scripts based on test cases using the chosen framework and tools.
- **Example:**

Function Under Testing

```python
# math_utils.py

def add(a, b):
    return a + b
```



```python
# test_math_utils.py

from math_utils import add

def test_add_positive_numbers():
    assert add(2, 3) == 5

def test_add_negative_numbers():
    assert add(-2, -3) == -5

def test_add_mixed_numbers():
    assert add(-2, 3) == 1

def test_add_zero():
    assert add(0, 0) == 0
```

Test Script

Test Script & Execution

- **Example:** A sample test script using Selenium + Python:

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

def test_login_success():
    driver = webdriver.Chrome()
    driver.get("https://example.com/login")

    driver.find_element(By.ID, "username").send_keys("valid_user")
    driver.find_element(By.ID, "password").send_keys("secure_password")
    driver.find_element(By.ID, "loginBtn").click()

    # Assertion
    assert "Dashboard" in driver.page_source

    driver.quit()
```

Test Analysis & Reporting

- Final and insight-driven phase of the Test Automation Lifecycle, where all the hard work of automation pays off.
- To evaluate **test outcomes**, identify issues or trends, and communicate the **health of the application** and automation efforts.
- **Key Activities:**

Activity	Description
Collect Test Results	Gather execution data: passed, failed, skipped tests
Analyze Failures	Investigate root causes: app bug, flaky test, environment issue
Generate Reports	Use tools to generate human-readable reports (HTML, PDFs, Dashboards)
Visualize Metrics	Show trends in execution time, pass rate, test coverage
Share Insights	Communicate findings with QA, Dev, PMs via dashboards, emails, standups
Improve Test Quality	Refactor or stabilize flaky tests; add missing coverage

Test Analysis & Reporting

Useful Metrics to Track

Metric	Why It Matters
Pass/Fail Rate	Overall test stability
Test Flakiness Rate	Helps flag unstable or unreliable tests
Time to Execute	Indicates efficiency of the suite
Test Coverage	Ensures all critical paths are tested
Defects Detected Early	Measures value of automation in early QA

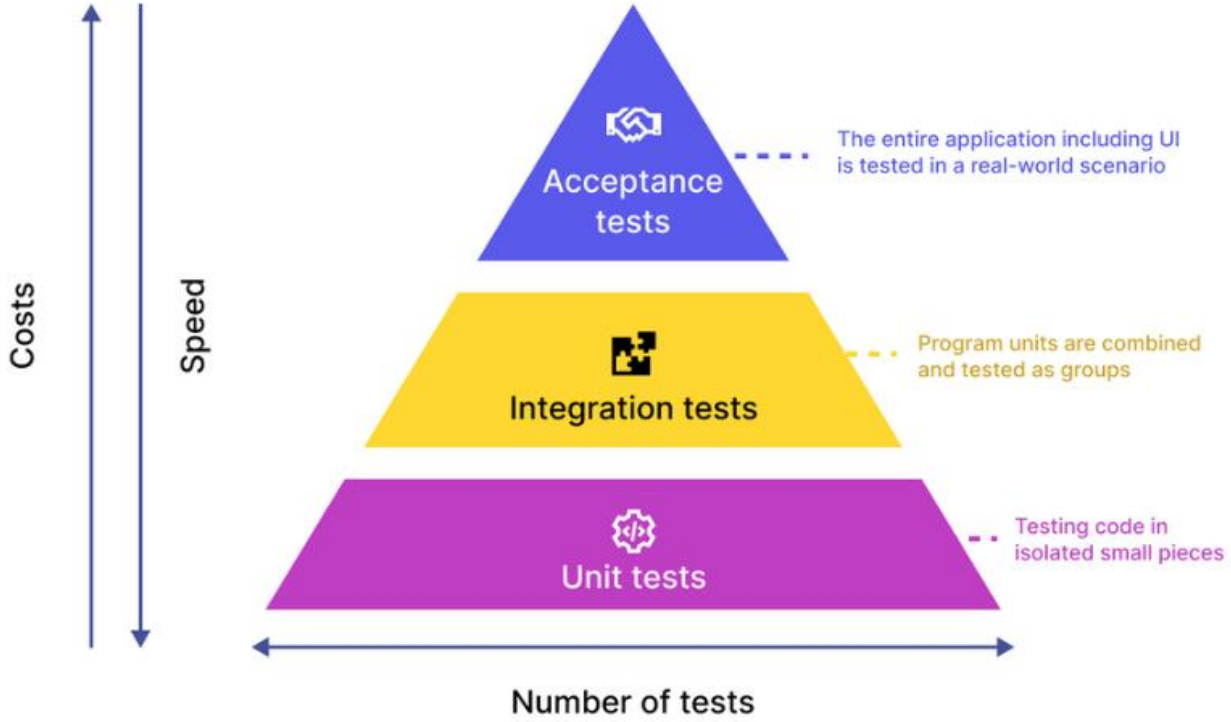
Example

Project: Online Banking App

Daily Report Shows:

- 98 tests run → 90 passed, 8 failed
- 5 flaky tests identified (same failures across runs)
- Avg execution time: 6.5 mins
- Coverage: 80% of core flows tested
- Shared with QA & Dev on Slack + CI dashboard

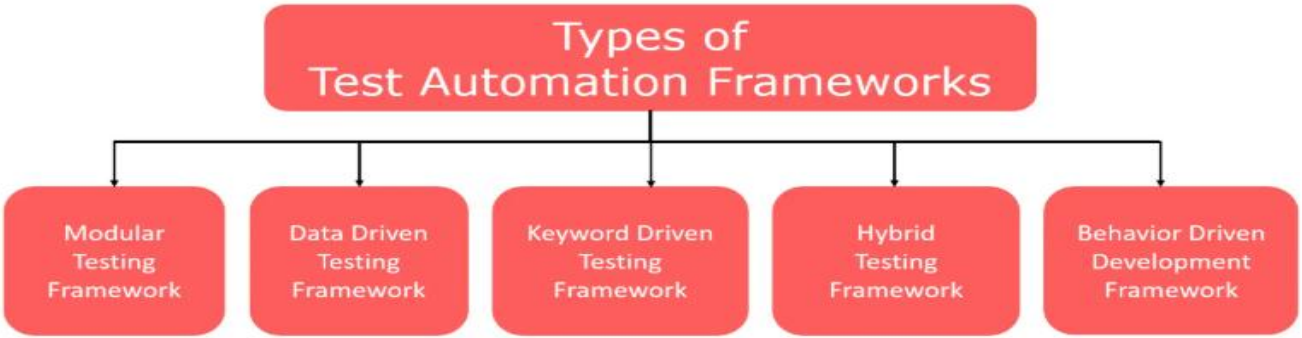
Test Automation Pyramid



Classification of Test Automation

Test Automation Frameworks

- A test automation framework is a set of **guidelines, tools,** and **practices** designed to help automate software testing in a structured and efficient way.
- It's the **foundation and toolbox** you use to write, organize, and run automated tests consistently across your project.



Key Features of Automation Frameworks

- **Reusability** – Common functions and utilities used across tests.
- **Maintainability** – Easier to update tests as the app changes.
- **Scalability** – Supports adding more tests without breaking.
- **Consistency** – Standardizes test writing, naming, and structure.
- **Reporting** – Provides logs, pass/fail summaries, screenshots, etc.
- **Integration** – Can connect with CI/CD tools, bug trackers, etc

Linear Scripting Framework

- Also known as the '**Record and Playback**' framework
- **Simple** and best for small projects or beginners
- Each test is written individually without much reuse
- **Advantages:**
  - Coding knowledge is not required
  - A quick way to generate test scripts
- **Disadvantages**
  - Hard to maintain as project grows
  - Lack of reusability
- **Example:** Selenium IDE – record browser actions and play them back.

Modular Testing Framework

- Breaks tests into independent, reusable modules.
- Each test case calls these modules as needed.
- Use page object model (POM) for maintainability
- **Advantages:**
  - Better scalability and easier to maintain
  - Can write test scripts independently
- **Disadvantages**
  - Requires more initial effort to develop scripts
  - Requires coding skills to set up the framework

Example:

```
# login_module.py
def login(username, password):
    # login steps here


# test_cart.py
from login_module import login
login('user1', 'pass123')
# continue with cart tests
```

Data-driven Framework

- Focused on separating the test scripts logic and the test data from each other.
- Test data set is kept in the external files or resources such as MS Excel Sheets, MS Access Tables, SQL Database, XML files, etc.,
- **Advantages:**
  - It supports multiple data sets
  - Modifying the test scripts won't affect the test data
- **Disadvantages**
  - Require coding skills
  - Setting up the framework and test data takes more time
- **Example:** Use an Excel or CSV file to test a form with 100 inputs.

Keyword-driven Framework

- **Advantages:**
  - No need to be an expert to write test scripts
  - It is possible to reuse the code. We can point the different scripts to the same keyword
- **Disadvantages:**
  - Take more time to design
  - The initial cost is high

Keyword-driven Framework

Example: Login to a website.

Step	Keyword	Object	Test Data
1	OpenBrowser	chrome	URL
2	EnterText	username_field	testuser
3	EnterText	password_field	password123
4	Click	login_button	
5	VerifyText	welcome_label	Welcome, User!
6	CloseBrowser		

Hybrid Framework

- Combines two or more approaches above (like data + keyword driven).
- Most real-world projects use hybrid frameworks.
- **Example:** Selenium with Python where test logic is modular, data is externalized in Excel, and actions are keyword-based.

Behavior-Driven Development (BDD) Framework

- BDD is a development approach that encourages collaboration between developers, testers, and business stakeholders.
- It focuses on writing tests in **natural language** that non-technical stakeholders can understand.
- Tests are written in **Gherkin syntax** (Given–When–Then format), which describe expected behavior of the system from a user perspective.

Behavior-Driven Development (BDD) Framework

Gherkin Syntax

Feature: *Login functionality*

Scenario: *Successful login with valid credentials*

Given *the user is on the login page*

When *the user enters valid username and password*

And *clicks the login button*

Then *the user should be redirected to the dashboard*

Automation Frameworks & Tools

Framework Type	Tools/Libraries
Linear	Selenium IDE, Katalon Recorder
Modular	Selenium + Python/Java
Data-Driven	TestNG + Excel, JUnit + CSV
Keyword-Driven	Robot Framework, Katalon Studio
Hybrid	Selenium + TestNG + Apache POI
BDD	Cucumber (Java), Behave (Python)

Integrating Test Automation with CI/CD

- What is CI/CD?
  - **Continuous Integration**: Automating the integration of code changes into the main codebase.
  - **Continuous Delivery**: Automating the deployment process to ensure new changes are automatically released.
- Automation in CI/CD:
  - Integrating automated tests into the CI/CD pipeline to run tests automatically on every code commit.
  - Benefits: Faster feedback loops, early bug detection, and seamless delivery process.

Challenges in Test Automation

High Initial Investment
Significant time and cost to choose tools, develop frameworks, and train staff

Test Maintenance Overhead
Changes in UI or application logic, requiring frequent updates.

Unstable Tests (Flaky Tests)
Sometime passes, sometime fails. Causes confusion and reduce trust

Partial Test Coverage
Not all test cases can be automated

Choosing the Right Tool
Matching with tech stack, team skill level, and project needs is difficult

Lack of Skilled Resources
Requires both testing knowledge and programming skills

What are Flaky Tests

- A flaky test is a test that sometimes passes and sometimes fails without any change to the code!
- The test result is not reliable even though the code and environment haven't changed.
- **Common Causes of Flaky Tests:**

Cause	Example
Timing issues / async waits	Test clicks a button before it's actually clickable
Element not yet loaded	Trying to read a field before it appears
Network/API delays	Test fails if an API call is slow but eventually works
Dependency on external systems	Test hits a 3rd-party API that is temporarily down

Myths in Test Automation

Myth #1: Test automation is expensive and requires a lot of resources.
Myth #2: Test automation totally eliminates the need for manual testing and replaces jobs.
Myth #3: Test automation is only suitable for large projects with ample resources.
Myth #4: It's better to use Selenium or another open-source tool to test.
Myth #5: You need to be a technical expert to utilize automation.
Myth #6: Test automation can be done absolutely by anyone.
Myth #7: Test automation is just a fad.
Myth #8: Test automation is a one-time activity.
Myth #9: Test automation is only for regression testing.
Myth #10: Test automation is inflexible and cannot adapt to changes in software requirements. It is hard to maintain.

Assignment 2

- Part A (4 marks) – Report on Software Test Automation
- Part B (4 marks) – Demonstration of Test Automation
- Part C (12 marks) – Individual Viva Presentation

Tutorial / Quiz

Thank You

Q & A

suresh.n@sliit.lk | 755841849

Test Driven Development

27th April 2025

Mr. Suresh Fernando

Test Driven Development - Overview

- Introduction to Test Driven Development
- TDD Vs Traditional Testing
- Three Rules in TDD
- TDD Lifecycle
- SOLID Principles for TDD
- TDD Impact on Developers
- Benefits and Challenges
- TDD Myths

What is Test-***Driven*** Development?

- A software development technique which reiterates the importance of testing
- Promotes writing software requirements as tests as the initial step in developing a code
- Initially writes tests and then moves forward with the least amount of code needed to get through the tests.

Brief History of TDD

- Late 1990s: TDD as a formalized practice was popularized by **Kent Beck** during the development of Extreme Programming (XP).
- Kent Beck later published the book **"Test-Driven Development: By Example" (2002)** which became a foundational text.
- The origins of TDD are closely tied to the **Agile Movement**, where iterative, feedback-driven development became the norm.

TDD in Agile Context

- TDD fits Agile because it supports:
  - ***Short feedback loops*** (you know if your code works instantly).
  - ***Continuous integration*** (tests support rapid changes).
  - ***Customer collaboration*** (cleaner, testable code is easier to evolve).
- Agile methodologies like **Scrum** and **Extreme Programming** (XP) use TDD as a key technical practice to ensure quality and adaptability.

Test-Driven vs Traditional Testing

Aspect	TDD	Traditional Testing
Approach	Tests are written before code implementation	Testing is performed after code implementation
Test Focus	Emphasizes small, incremental tests for specific features	Focuses on comprehensive testing of the entire application
Test Creation	Tests are written by developers based on requirements	Tests are typically created by dedicated QA testers
Code Coverage	Encourages high code coverage with a focus on critical paths	May have varying levels of code coverage depending on test cases
Feedback Loop	Provides immediate feedback on code changes	Feedback may be delayed until the testing phase
Bug Detection	Helps identify bugs early in the development process	Bugs may be detected later in the development cycle

Steps in Test-Driven Development

1. Read and understand the feature request from clients.

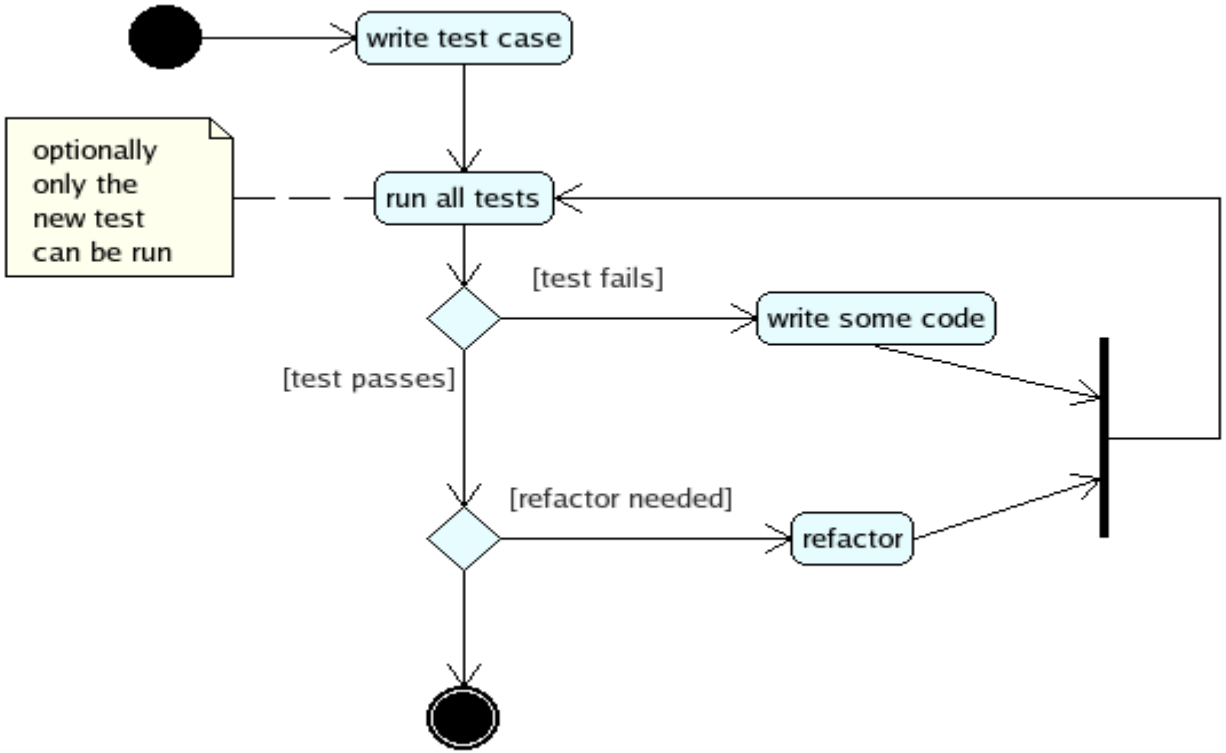
2. Translate them by creating a test. This test will fail when you run it because no code has been implemented so far.

3. Now, write and implement the code to pass the test. Then, run the test and now it should pass. If it doesn't pass yet, repeat the steps until it passes.

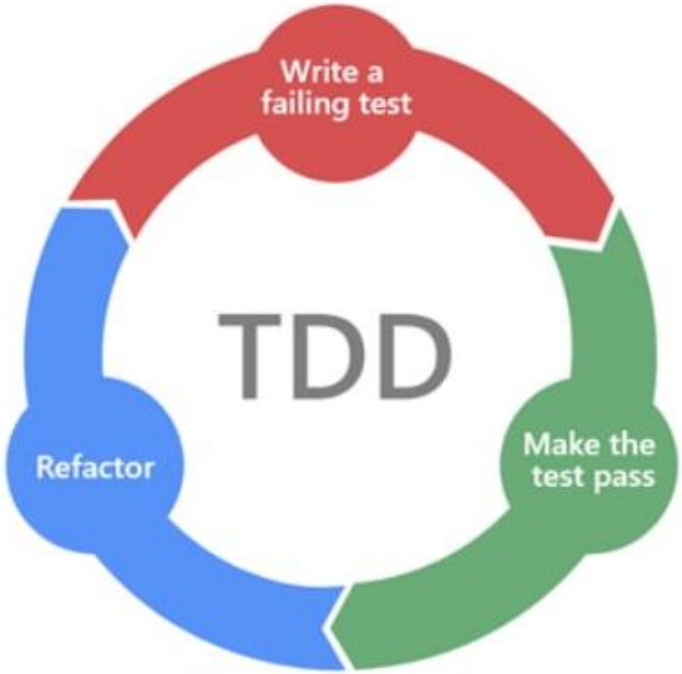
4. Once it passes, clean your code up. You can do it by refactoring.

5. Repeat above steps again for another requirement

Steps in Test-Driven Development



TDD Cycle (Red-Green-Refactor)



- Here, the **red** phase denotes that the code is not working.
- The **green** phase is an indication that everything is working. However, they do not work optimally.
- The **blue** phase denotes that the code is being refactored.

TDD Cycle – Red

- Goal:

Write a **test that fails** because the functionality you want **doesn't exist yet.**

- Why?

- Forces you to **think about the problem** and define **expected behavior** *before* implementation.
- Ensures your test is **valid** – if it doesn't fail when the feature is missing, something's wrong with the test.
- Prevents writing unnecessary code – you only write code to **make the test pass.**

TDD Cycle – Green

- Goal:

Write just enough code to make the failing test from the Red Phase pass – nothing more.

"Get it working first. Make it beautiful later."

- Why?

- Ensures your test was valid.
- Gives you confidence that the system behaves as expected.
- Helps you build functionality incrementally.
- Keeps you focused – no extra code means less chance of bugs.

TDD Cycle – Refactor

- Goal:

Improve the code's structure, readability, and maintainability - without changing its behavior.

"Now that it works, let's make it nice."

- Why?

- Keeps code clean and maintainable.
- Prevents ***technical debt*** from accumulating.

- What?

- Remove duplication.
- Rename variables/methods for clarity.
- Simplify logic or extract methods.

Activity

Discuss in Pairs – “What are the possible benefits and challenges you could expect from TDD?”

Three Rules of TDD

- **Rule #1 – No code without a test**
  - No new features or logic unless there's already a test failing because that logic is missing.
- **Rule #2 – No more test than needed**
  - Don't write multiple tests or long tests upfront. Just write *enough* to see the test fail.
- **Rule #3 – No more code than needed**
  - Only write code that's needed to make the current test pass. No extra logic, no predictions of future features.

#1- No code without a test

Code

```
def add(a, b):
    return a + b
```

You can't write this ***add*** function until you've written a test like mentioned below

Test

```
def test_addition():
    assert add(2, 3) == 5  # ✗ This test will fail initially.
```

Once the test fails, you write the minimal code to make it pass.

Test

```
def test_password_too_short():
    validator = PasswordValidator()
    assert not validator.is_valid("abc")  # Password must be at least 6 characters
```

Code

```
class PasswordValidator:
    def is_valid(self, password):
        return len(password) >= 6
```

#2- No more test than needed

Suppose you want to build a function that sums numbers from a comma-separate string input.

First Test : To check if an empty string will return zero

```
def test_add_empty_string_returns_zero():
    assert add("") == 0  # Fails because 'add' isn't implemented yet.
```

Code: Then write ***minimal*** production code:

```
def add(numbers):
    return 0  # ✅ Now the test passes.
```

Here we are not jumping ahead and writing tests for comma-separated numbers yet but just focused on the first test

#3- No more code than needed

Next Test : To check if a string with single number will return the same number

```
def test_add_single_number():
    assert add("5") == 5
```

Code: Update code ***just enough*** to pass

```
def add(numbers):
    if numbers == "":
        return 0
    return int(numbers)
```

Here we don't yet add codes to support for two numbers or handle delimiters. That will come with the future tests accordingly.

Benefits of TDD

Improve code quality
Code is written to pass specific tests

Immediate Feedback
Finds out immediately if something is broken.

Better Design & Maintainability
Encourages writing loosely coupled, highly cohesive code

Documentation Through Tests
Tests act as **live documentation** of how the system should behave

Confidence to Refactor
You can improve or change code structure without fear, because tests will catch regressions.

Facilitates Debugging
When something breaks, you already have unit tests that isolate logic

Challenges of TDD

Initial Learning Curve
Beginners may struggle with writing tests first or designing testable code

Slowdown Development at First
Writing tests before writing any code will consume time

Maintenance Overhead
A large suite of outdated or redundant tests becomes a burden.

UI, Legacy, or Non-Deterministic Code
TDD works best with deterministic, modular code

Over-Testing / Rigid Code
Writing too many detailed tests can make refactoring painful

Mindset Change
TDD requires a **fundamental shift** in how developers think about writing code.

SOLID Principles for TDD

- SOLID principles are key to object-oriented design and closely related to Test-Driven Development (TDD).
- SOLID is not exclusive to TDD, but practicing TDD tends to enforce or encourage adherence to these principles.

Letter	Principle	Purpose
S	Single Responsibility Principle (SRP)	One reason to change
O	Open/Closed Principle (OCP)	Open to extend, closed to modify
L	Liskov Substitution Principle (LSP)	Subclasses must be replaceable
I	Interface Segregation Principle (ISP)	Prefer many small interfaces
D	Dependency Inversion Principle (DIP)	Depend on abstractions, not concrete classes

S - Single Responsibility Principle (SRP)

Principle: A class should have **one and only one reason to change.**

TDD Implication: As you write unit tests first, you tend to **split responsibilities** to make the code more testable.

- Each module, class, or function should **do one thing and do it well**.
- If a class is responsible for more than one thing, those responsibilities become **coupled**.
- A change to one responsibility may impact the others.

S - Single Responsibility Principle (SRP)

Example: Suppose you're writing a *ReportService*.

Before (Violates SRP):

```
class ReportService:
    def generate_report(self):
        # gathers data, formats it, sends email
```

Too many roles in one place

TDD forces to separate these roles

After TDD forces separation:

```
class DataFetcher:
    def get_data(self): pass

class ReportFormatter:
    def format(self, data): pass

class EmailSender:
    def send(self, report): pass
```

O – Open/Closed Principle

Principle: Software entities should be **open for extension but closed for modification.**

TDD Implication: TDD encourages you to write code that can **evolve** via extension - because changing tested code is risky.

- You should be able to **add new functionality** to a class or module **without changing its existing code.**
- This protects existing behavior from bugs and makes the system easier to maintain and extend.

O – Open/Closed Principle

Example: You write tests for a tax calculator. Later you want to support a new tax strategy.

```
class TaxStrategy:
    def calculate(self, amount): pass

class FixedTax(TaxStrategy):
    def calculate(self, amount):
        return amount * 0.1

class ProgressiveTax(TaxStrategy):
    def calculate(self, amount):
        if amount > 1000
            return amount * 0.2
        else
            return amount * 0.1
```

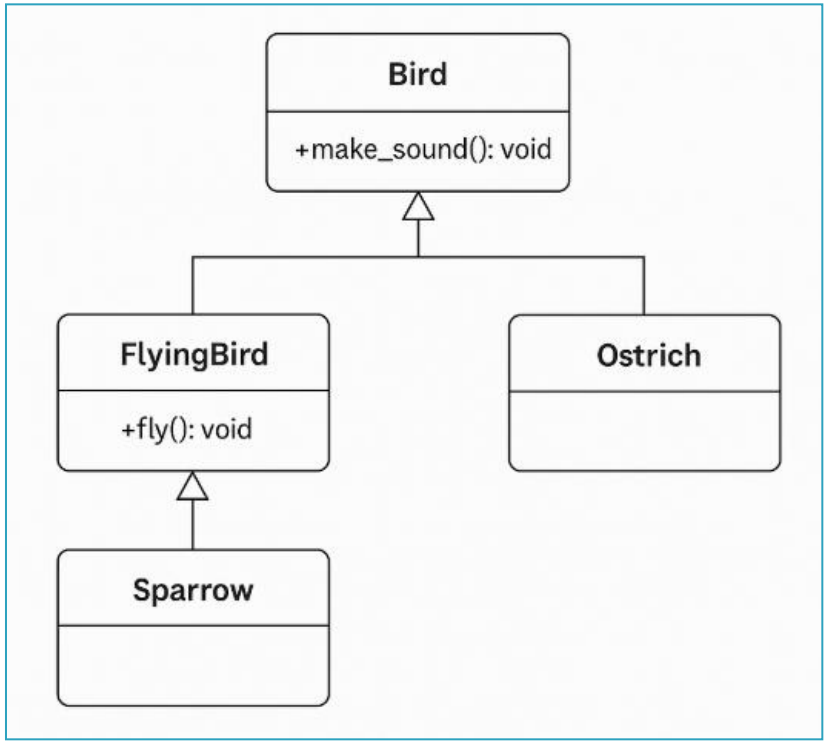
You don't modify existing logic — you **extend** with a new class so that you can run new tests for it.

L – Liskov Substitution Principle

Principle: Subclasses should be **substitutable** for their base classes without altering behavior.

TDD Implication: TDD helps spot LSP violations early - when a test for a base class fails with a subclass.

L – Liskov Substitution Principle



L – Liskov Substitution Principle

Example: Let's say you're testing *Bird.fly()* but your Ostrich subclass throws an error.

```
class Bird:
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostriches can't fly")
```

```
class Bird:
    def make_sound(self):
        print("Chirp")

class FlyingBird(Bird):
    def fly(self):
        print("Flying")

class Sparrow(FlyingBird):
    pass

class Ostrich(Bird):
    pass
```

I – Interface Segregation Principle

Principle: Clients shouldn't be forced to depend on interfaces they don't use.

TDD Implication: TDD naturally leads to **smaller, focused interfaces,** because large ones are hard to test.

I – Interface Segregation Principle

Example: You create a test for a SimplePrinter, but it doesn't scan. You're forced to ignore scan_document()

Before ISP Principle:

```
class Printer:
    def print_document(self): pass
    def scan_document(self): pass
```

Better design with ISP

```
class Printable:
    def print_document(self): pass

class Scannable:
    def scan_document(self): pass

class SimplePrinter(Printable): pass
class MultiFunctionPrinter(Printable, Scannable): pass
```

D – Dependency Inversion Principle

Principle: Depend on **abstractions**, not on concrete implementations.

TDD Implication: You often start writing tests by **mocking dependencies**, which encourages depending on interfaces, not classes..

D – Dependency Inversion Principle

Example: Imagine you're writing a **UserSignupService**. After signing up, the user should receive a welcome message.

In a tightly coupled system (no DIP)

** UserSignupService directly uses a concrete EmailService.*
** Your tests now:*
- *Send real emails (bad).*
- *Rely on infrastructure (fragile, slow).*
- *Are hard to isolate and mock.*

Better design with DIP

UserSignupService depends on a MessageSender interface.
Now:
- *Tests are isolated, fast, and reliable.*
- *You can verify that "a welcome message was sent" without sending real emails.*
- *Your tests stay green even if you later switch to SMS or in-app notifications.*

Using Mockups in TDD

- **Goal**: Focus your test on only the **behavior of the system under test (SUT)**, without involving the real implementations of its dependencies.
- **Why Use Mocks in TDD?**
  - To test only one unit at a time.
  - To avoid slow or unreliable dependencies (e.g., network, databases).
  - To simulate specific behaviors like errors, timeouts, or success cases.
  - To verify interaction between the SUT and its dependencies.

Using Mockups in TDD

- **System Under Test:: *OrderService*** – Responsible for placing an order.
- **Dependencies: *PaymentGateway*** and ***EmailNotifier***

Without Mocks:

- When you test ***OrderService***, it actually tries to charge a credit card and send an email.
- This means:
  - You need real accounts.
  - Tests are slow and can fail due to network/email issues.

With Mocks:

- You create mock objects for ***PaymentGateway*** and ***EmailNotifier***.
- You inject these mocks into ***OrderService***.
- In the test:
  - You simulate a successful payment.
  - You verify that the email notifier was called correctly.
  - Now you're truly testing just the behavior of ***OrderService***.

TDD Impact on Developer's Life

Reversing the Usual Workflow

- Traditional approach: Write code → Then test it.
- TDD approach: Write test → Fail it → Write code → Pass the test → Refactor.
- This feels backward at first and can be uncomfortable for developers used to jumping straight into coding logic.

Thinking in Small, Testable Units

TDD forces developers to:

- Break problems into very small steps.
- Focus on testability, which often leads to better design – but takes more thought up front.

TDD Impact on Developer's Life

Letting Tests Drive Design

- Instead of designing your code then writing tests around it you let the tests guide your architecture.
- This feels strange, especially for developers used to object-oriented or top-down design habits.

Immediate vs. Long-Term Payoff

- TDD can feel slower in the short term.
- But over time, it creates fewer bugs, cleaner code, and faster refactoring.
- Convincing teams of this long-term ROI can be a tough cultural sell.

TDD Impact on Developer's Life

- Team and Organizational Culture
  - If only one developer adopts TDD, it may not mesh well with the rest of the team's workflow.
  - Organizational pressure to "deliver fast" often discourages writing tests first.

TDD Myths and Reality

Myth #1: TDD slows down development	**Reality**: While TDD may initially seem slower, it actually speeds up development in the long run.
Myth #2: TDD is just about writing tests and has no impact on the actual development process.	**Reality**: TDD is more than just testing. It is a design methodology that drives the development process.
Myth#3: TDD is only suitable for certain types of projects or specific programming languages.	**Reality:** TDD can be applied to almost any project, regardless of size, complexity, or technology stack.

Tutorial / Quiz

Join at menti.com | use code 2916 0928

Mentimeter

Instructions

Go to

www.menti.com

Enter the code

2916 0928

Or use QR code

Thank You

Q & A

suresh.n@sliit.lk | 755841849

Trends in Software Testing

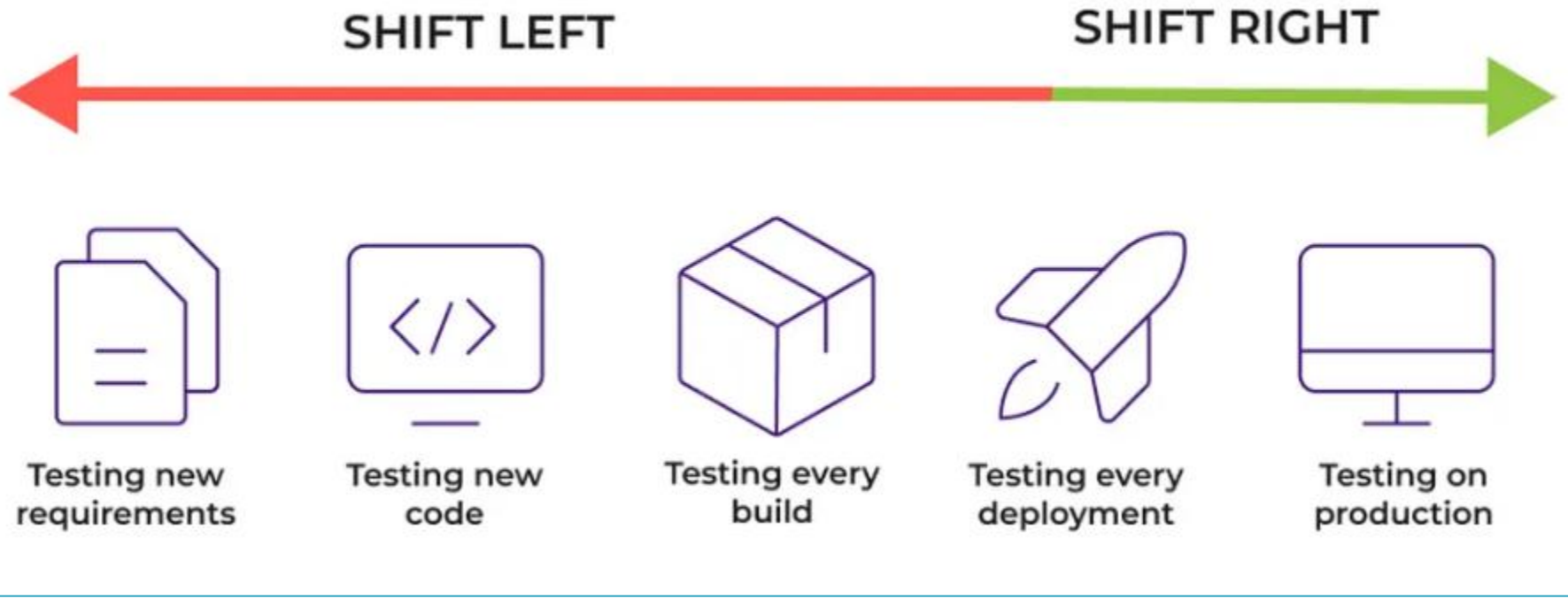
4th May 2025

Mr. Suresh Fernando

Trends in Software Testing

Shift-Left Testing	AI and ML in Testing	Growing Use of QAOps
Crowdtesting	Enhanced Focus on Security Testing	IoT Testing
Mobile Test Automation	Enhanced API Testing and Automation	Focus on accessibility testing

1. Shift-Left Testing



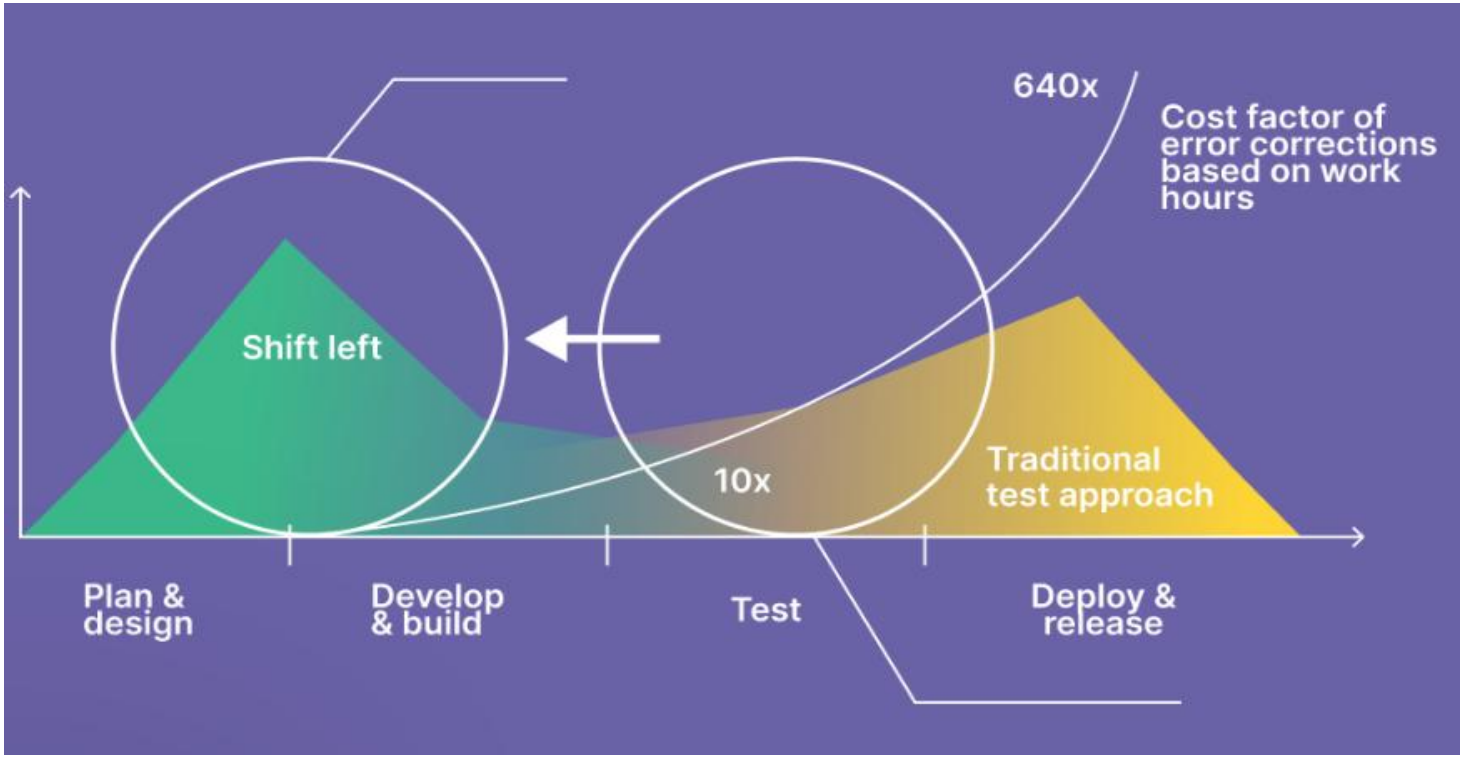
Shift-Left Testing

- Emphasizing early and frequent integration of testing in the software development cycle
- Why important?
  - Identifying and addressing issues sooner
  - Accelerating market time
  - Enhancing software release quality
  - Reducing the time spent on debugging
  - Enabling teams to devote more effort to feature and functionality enhancement

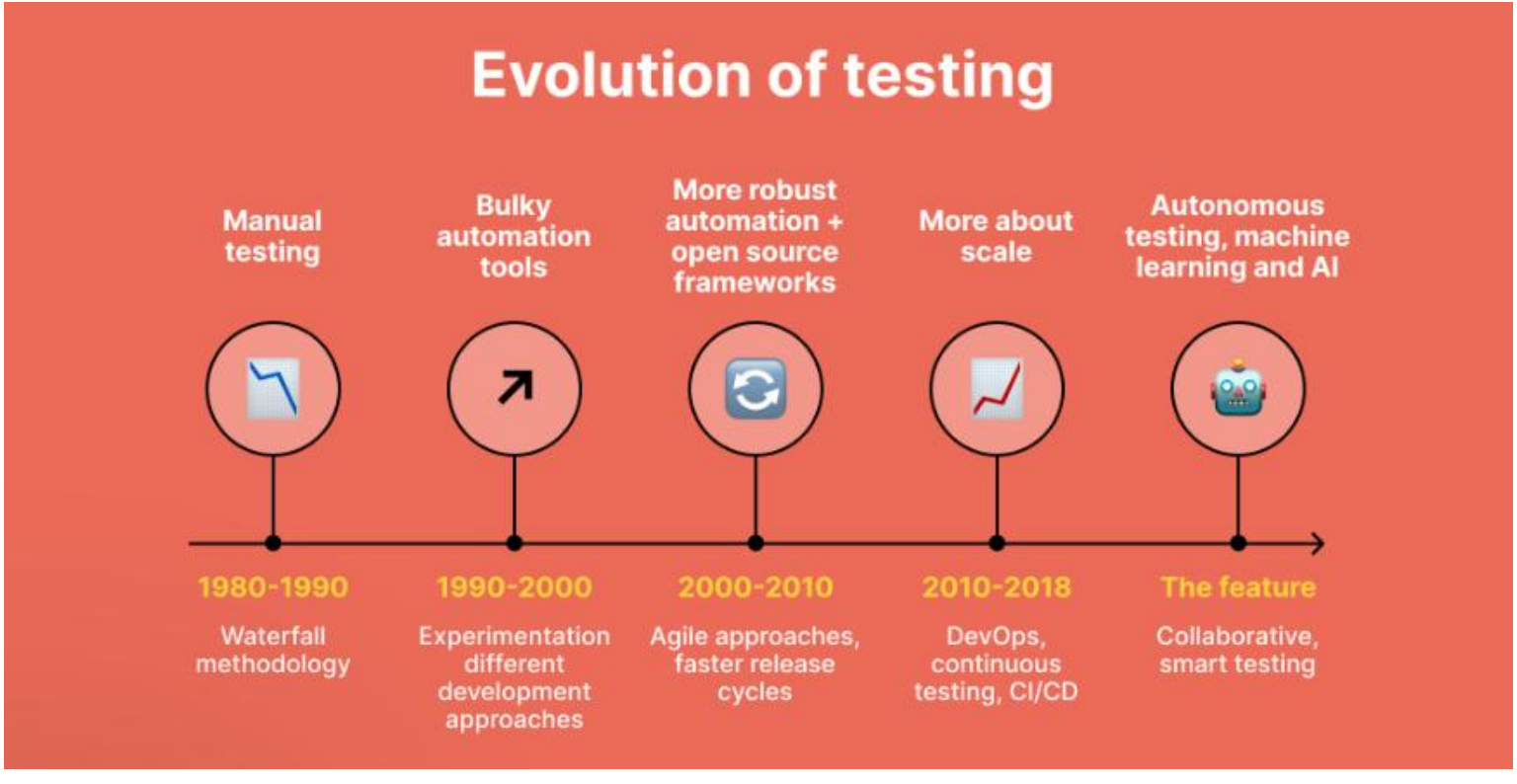
Shift-Left Testing

- Delayed testing will result in:
  - Insufficient testing resources
  - Missed design
  - Architectural or requirements flaws
  - Complexities in debugging and issue resolution, and
  - Project delays.

Shift-Left Testing



2. AI and ML in Software Testing



AI and ML in Software Testing

Test Case Generation: AI and ML algorithms can automatically generate test cases based on software requirements and historical data

Test Prioritization: AI can analyze code complexity, recent changes, and historical defect data to identify and prioritize the most critical test cases for execution.

Defect Prediction: By analyzing patterns in historical data, AI can predict potential areas of failure, allowing testers to focus their efforts on those areas.

Test Execution and Result Analysis: AI can automate the execution of test cases and analyze the results, identifying potential errors and defects.

Intelligent Feedback: AI can provide more accurate and context-aware feedback to testers, helping them understand the root cause of defects.

Benefits of Using AI and ML in Testing

Increased Efficiency: Automating tasks and prioritizing test cases saves time and resources.

Improved Accuracy: AI can help identify and fix defects more effectively, leading to higher quality software.

Cost Reduction: By automating tasks and reducing manual effort, AI and ML can lower the overall cost of software testing.

Enhanced Software Quality: By identifying and fixing defects early, AI and ML can help ensure that software meets quality standards.

Improved Developer Productivity: AI can help developers by providing them with feedback on their code and identifying potential issues before they become bugs.

Challenges of Using AI and ML in Testing

Data Requirements: AI and ML algorithms require large amounts of data to train and learn, which can be a challenge for some projects.

Algorithm Complexity: Developing and maintaining AI and ML algorithms can be complex and require specialized expertise.

Integration with Existing Tools: Integrating AI and ML tools into existing software testing workflows can be challenging.

Ethical Considerations: As AI and ML become more prevalent in software testing, it's important to consider the ethical implications of using these technologies, such as bias and fairness.

AI/ML Techniques in Software Testing

Natural Language Processing (NLP)
Understanding user stories, requirements, or test cases written in natural language
(Automatic test case generation | Requirement traceability)

Predictive Analytics
Using historical test data to forecast future defects or risky components
(Defect prediction | Test prioritization)

Clustering & Classification
Grouping test cases or defects by similarity or categorizing them
(Test suite optimization | bug triage)

Reinforcement Learning
Systems learn the best sequence of actions (e.g., in test generation) through trial and error
(Intelligent exploratory testing)

Anomaly Detection
Identifying patterns that deviate from the norm
(Monitoring in production – AIOps | detecting flaky tests)

NLP in Testing

- NLP in software testing is used to help automate, understand, or improve testing tasks that involve human language.
- **How NLP is used:**
  - **Analyzing requirements:** NLP can read written requirements (in English or other languages) and automatically suggest or generate test cases.
  - **Generating test cases from user stories:** By parsing user stories or acceptance criteria written in natural language, NLP tools can help draft test cases, reducing manual effort.
  - **Analyzing bug reports:** NLP can cluster, categorize, or prioritize bug reports by reading the language used - e.g. identifying duplicate issues or estimating severity.
  - **Automated testing of conversational systems:** For chatbots or voice assistants, NLP helps test whether the system understands and responds correctly to varied natural language inputs.

NLP in Testing - *An Example*

- Imagine a software team working on an e-commerce platform. The product team writes a user story:

  "As a customer, I want to add products to my shopping cart so I can purchase them later."

- An NLP-based test tool can read this sentence and suggest:
  - Test case 1: Verify adding a single product to the cart.
  - Test case 2: Verify adding multiple products.
  - Test case 3: Verify the cart persists across sessions.
  - Test case 4: Verify error handling when adding an out-of-stock item.
- Without NLP, a human tester would manually write these cases.
- With NLP, the system **automates the interpretation of the text and speeds up test preparation.**

Predictive Analytics in Testing

- **Predictive analytics** uses historical data, statistical models, and machine learning to forecast future outcomes in the testing process.
- It helps testers and managers make informed decisions about **where, what, and how** to test more effectively.
- **How Predictive analytics is used:**
  - **Defect Prediction**: Predicts which modules or features are likely to have defects based on past bugs, code changes, and complexity.
  - **Test Effort Estimation**: Estimates the amount of time and resources needed for testing based on previous project data.
  - **Test Case Prioritization**: Ranks test cases by likelihood of finding defects, helping focus on high-value tests first.
  - **Release Readiness Prediction**: Forecasts whether the software is stable enough for release based on defect trends and testing progress.

Predictive Analytics in Testing – *An Example*

- **Example Scenario**: Defect Prediction
- **Context**: Your team maintains a large web application. Historically, bugs are more frequent in certain modules after big changes.
- **How Predictive Analytics Helps**:
  - You collect historical data: Code churn (how often code changes) | Module complexity | Past defect density | Developer activity
  - You train a predictive model that identifies modules likely to have defects in the next release.
  - The model predicts that the "payment" and "checkout" modules have a high risk of defects.
- **Outcome**:
  - You prioritize test cases for those modules.
  - Allocate more testers and automation to those areas.
  - Prevent costly defects before release.

Clustering & Classification in Testing

- **Clustering** is an unsupervised learning technique used to group similar data points without predefined labels.
- **Classification** is a supervised learning technique that assigns labels to data based on learned patterns from labeled examples.
- Both are widely used in software testing to organize test data, streamline processes, and improve defect prediction.
- **How Clustering is used**:
  - Groups similar test cases or bug reports to identify duplicates, overlaps, or patterns.
  - Helps optimize test suites by removing redundant cases.
- **How Classification in Testing**:
  - Predicts outcomes like test case failure, defect severity, or module risk.
  - Automatically filters and routes bug reports (e.g., valid vs. invalid, critical vs. minor).

Clustering & Classification in Testing - Example

- **Example Scenario**: Bug Triage System
- **Clustering**:
  - You have thousands of bug reports in free-text format.
  - A clustering algorithm (like K-means or DBSCAN) groups similar bug descriptions.
  - You notice multiple clusters of bugs related to “login errors” and “payment failures.”
  - This helps in identifying duplicate reports and prioritizing high-impact issues.
- **Classification**:
  - You train a classifier (like Decision Tree or SVM) on historical bug data with features such as: Text of the report, Component affected, Steps to reproduce,
  - Labels include "High," "Medium," or "Low" severity.
  - New bug reports are automatically classified into severity levels to assist developers in prioritizing them.

Reinforcement Learning in Testing

- **Reinforcement Learning** is a type of machine learning where an agent learns to make decisions by interacting with an environment, receiving rewards or penalties based on the outcomes of its actions.
- In software testing, RL is used to **learn optimal testing strategies over time.**
- **How Reinforcement Learning is Used:**
  - Test case prioritization
  - Test path exploration (especially for UI or API testing)
  - Automated bug discovery
  - Dynamic test suite optimization
  - Self-healing test automation

Reinforcement Learning in Testing - Example

- **Example Scenario:** UI Testing with Reinforcement Learning
- **Context**: You are testing a web application's user interface. Manually exploring all possible user interactions is time-consuming.
- **How RL Works Here**:
  - **Agent**: The RL agent acts as an automated tester.
  - **Environment**: The web application's UI.
  - **States**: Current screens/pages and element states.
  - ***Actions***: Clicking buttons, filling forms, navigating pages.
  - ***Reward***: Positive reward for finding bugs or increasing coverage; penalty for dead ends or redundant paths.
- **Workflow**:
  - The agent starts on the homepage.
  - It chooses to click a button (e.g., "Login").
  - If the action leads to a new page or exposes a bug (e.g., error message), the agent gets a reward.
  - Over many test sessions, the agent learns which actions yield the most information or errors.
  - Eventually, it builds a test strategy that explores the application efficiently and effectively.

Anomaly Detection in Testing

- **Anomaly detection** involves identifying data or behavior that significantly deviates from what is expected or typical during testing.
- It helps uncover issues not directly covered by test cases - such as performance drops, unexpected errors, or subtle bugs.
- **How Anomaly Detection is Used:**
  - Detecting performance regressions
  - Identifying unstable modules
  - Spotting new, rare, or unexpected bugs
  - Monitoring system logs for unusual events

Anomaly Detection in Testing - Example

- **Context:** You are testing a web application, and you routinely run load tests to measure API response time under varying user loads.
- **Expected Behavior:** Under normal conditions, the API should respond in 100–200 milliseconds.
- **Observed Behavior During a Test Run:**
  - 90% of requests: ~150 ms (normal)
  - 10% of requests: 700 ms (unexpected spike)
- **How Anomaly Detection Helps:**
  - A statistical or machine learning-based anomaly detection model analyzes the response time data and flags the 700 ms results as anomalies, even though the test case itself did not "fail."

3. Rising significance of QAOps

- QA is no longer just a separate testing phase
- It is **embedded continuously** throughout development, deployment, and delivery, using **automation, tools,** and **close collaboration** between QA, development, and operations teams.
- Key Ideas:
  - **Continuous** testing across the CI/CD pipeline.
  - **Automated test execution** after each code change or deployment.
  - **Shift-left testing –** QA works early in the development process.
  - **Monitoring in production** for ongoing quality (not just pre-release).

QAOps Cycle

QAOps **integrates** QA into the DevOps cycle

DevOps vs QAOps

DevOps	QAOps
Operations and Developers have prime roles, and QA will function as a subset of development.	QA specialists work collaboratively with Operations and Developers in primary roles.
Give importance more to deploying software rapidly.	Give more importance to guaranteeing the quality of software
The software app's quality will be good in this case.	The software app's quality will be excellent in this case.

4. Crowdtesting

- A method that involves a large group of testers **who are not part of the company's internal QA team**
- Engage with these testers **through crowdsourcing platforms** (Amazon Mechanical Turk, Upwork, 99designs, etc.) where they can submit their software or applications for testing.
- As more organizations adopt this strategy, the global market for crowdtesting is **expected to expand**.

Crowdsourcing vs Outsourcing

Crowdsourcing	Outsourcing
Global Not limited by an office location - workers can be anywhere in the world	**Single location center** Based around center locations, typically offshore and limited to the local talent pool
24/7 Crowd workers can work from anywhere. The are no tied to office hours and can create their own work schedule	**Set work hours** Workers execute from the facility in shifts to meet customer requirements
Flexible workforce On-demand access to specialize resources, in any geography an multiple languages	**Rigid workforce** Clients commit to fixed staffing models that require lead time for ramp up and down activities

Crowdsourcing vs Outsourcing

Crowdsourcing	Outsourcing
Output based pricing Clients pay or returned work meeting quality standards allowing transparency, predictability and accountability for business results	**Headcount pricing** Usually based on headcount and hourly rates making it difficult for clients to predict throughput
No overhead costs No facility or fixed costs associated with the model	**Fixed costs** Facility, bench and other fixed costs add to the price of the outsourcing model

Benefits of Crowdtesting

- Scalable testing resources that are available as needed.
- More comprehensive test coverage.
- Quicker feedback loop with end users.
- The influx of specialized expertise and experience.

5. Evolution of Test Automation

- Revise ‘Test Automation’ lecture…
  - Introduction to Test Automation
  - Manual Vs Automated Testing
  - Principles of Test Automation
  - Test Automation Lifecycle
  - Test Automation Frameworks
  - Challenges & Myths in Test Automation

6. Enhanced focus on Security Testing

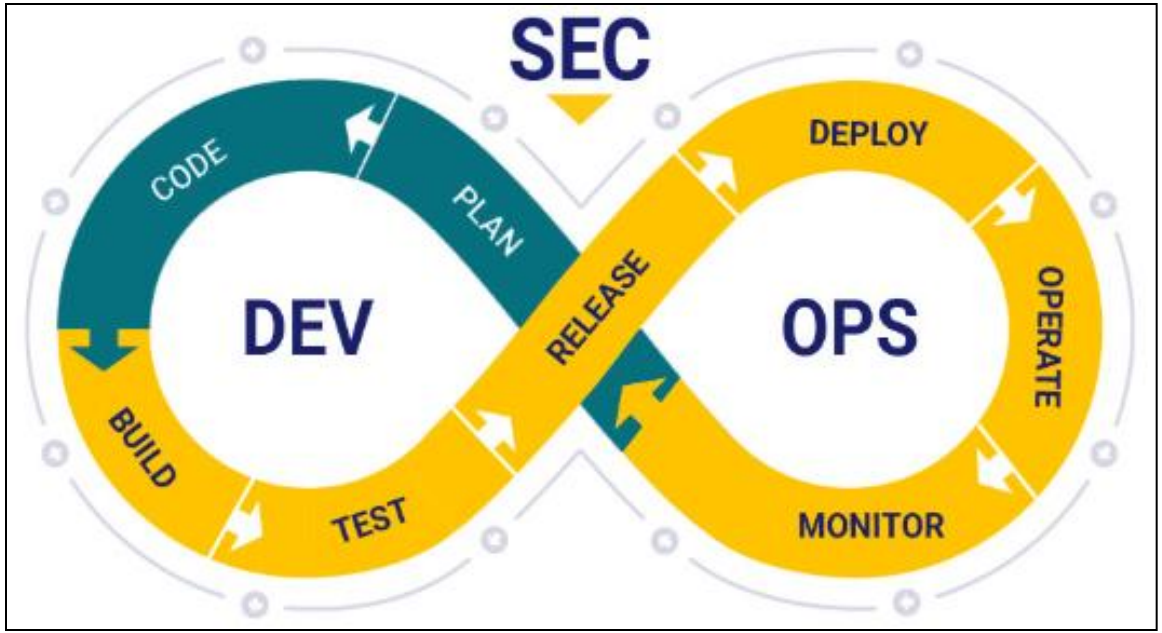
- **Cybersecurity threats and data breach** incidents surged recently
- Integrating security from the **initial stages of product design and development** is now essential.
- Emerging field of **DevSecOps**
- Emphasizes the importance of various security testing methodologies:

  Vulnerability scanning | Penetration testing | API testing | Web application security testing.

Cost of Cybercrime

The cost of cybercrime is predicted to grow exponentially, with a **70% increase by 2028**. This means the annual cost will jump from $8.15 trillion in 2024 to **$13.82 trillion by 2028**

DevSecOps

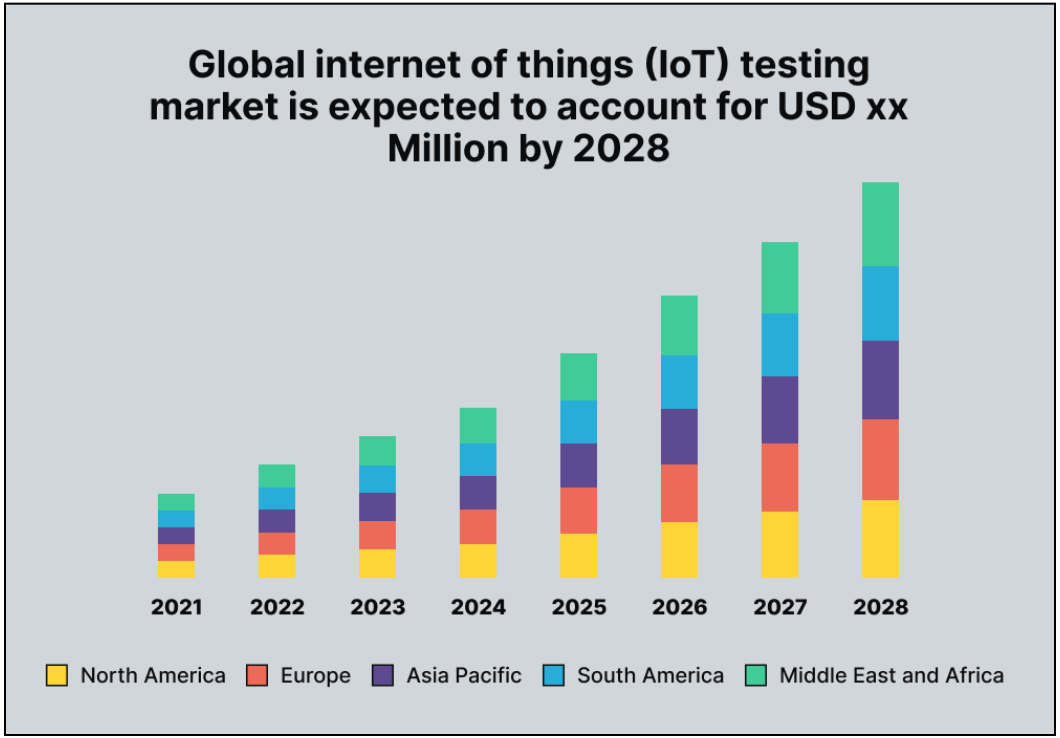


- DevSecOps is a methodology to provide security to application and infrastructure based on the principles of DevOps.
- This approach makes sure that the application is less vulnerable and ready for user's use.
- An automated process, and security checks started from the beginning of the application's pipelines.

7. IoT (Internet of Things) Testing

- Significant software testing trend focusing on IoT Instruments':
  - Security
  - Data integrity
  - Performance
  - Scalability
  - Compatibility.
- Is crucial for enhancing system productivity by preventing unexpected glitches.
- provides greater control over devices and helps improve network and device efficiency, accessibility, and usage for the users

IoT Testing Market



Significant growth in IoT Testing market reflects:

- The **increasing reliance** on IoT devices
- the need for effective testing to ensure their **functionality and security.**

8. Mobile Test Automation

- Development of mobile applications is **rapidly growing,** hence mobile test automation will greatly help
- Mobile test automation involves **using software tools and scripts** to **automatically test** mobile applications across **various devices and platforms.**
- Integrating **cloud-based mobile labs** and test automation tools represents a promising development, potentially elevating mobile test automation to new heights.

Cloud-based Mobile Labs

- A remote testing environment where mobile applications can be tested on various **real devices or emulators** hosted in the cloud.
- This allows testers to access and test apps on different operating systems, device models, and screen sizes **without the need for a physical on-site lab**
- **Key Features:** Remote Access | Real Devices | Virtual Devices | Scalability | Cost-Effectiveness | Flexibility | Integration
- **Examples of Cloud-Based Mobile Labs:** Sauce Labs | BrowserStack | LambdaTest | AWS Device Farm | Perfecto.io | Kobiton

9. Enhanced API testing and Automation

- Rise of microservices architectures has significantly increased the number of **Application Programming Interfaces (APIs)**
- Hence API Test Automation is way to go (***"Make frequent cases faster"***)
- Increasingly becoming the **standard practice** as API-driven development more relevant than ever
- API test automation is the solution for **achieving higher efficiency,** enabling more tests to be performed in a shorter time

API Testing Vs GUI Testing

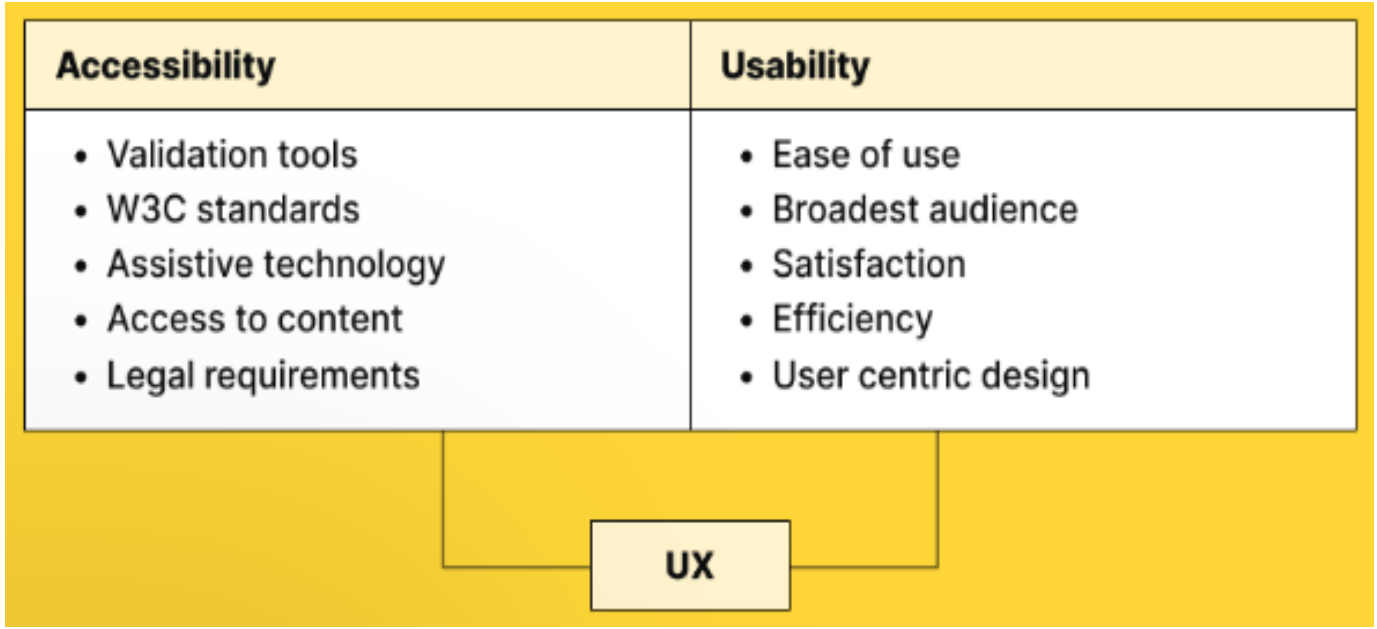
API	GUI
An API is a collection of communication protocols & subroutines	GUI is a software platform with visual & audio indicators
It enables the interaction between two programs or other technology products	It enables the interaction between a human & a computer program
API requires back-end storage supported by a logical architecture & a library of scripts	GUI doesn't require as many resources as an API
High technical skills are required to use it	It is easier to use and doesn't require technical skills
They are easier to automate and so can be tested quickly	It is complicated to automate and so takes a long time for testing
It allows the exchange of data through XML or JSON	GUI doesn't allow the exchange of data through XML or JSON

API testing is a more efficient alternative to extensive GUI testing in Agile or DevOps environments, where speed is crucial.

10. Focus on Accessibility Testing

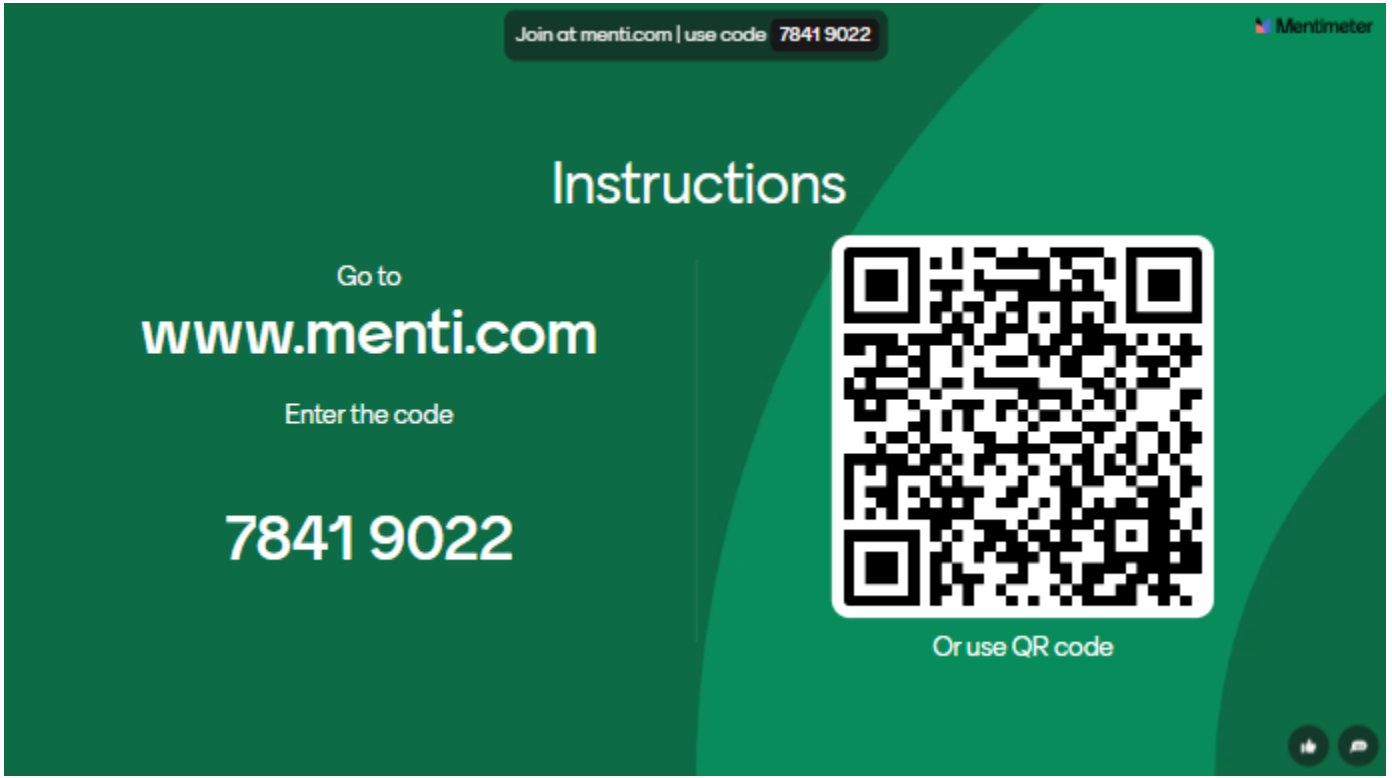
- This is an era of stringent **web and mobile accessibility regulations**
- Compliance through accessibility testing has become a critical component of software development.
- Accessibility standards: ***Americans with Disabilities Act (ADA), Section 508, and the Web Content Accessibility Guidelines (WCAG) 2.1.***
- Crowdtesting offers valuable perspectives from users with disabilities.
- For global applications, refining accessibility testing practices is essential to **bridge compliance gaps** and cater to a **diverse user base**

Focus Factors on Accessibility / Usability



Incorporating automation and AI in testing can efficiently handle the repetitive aspects (**screen readers, magnifiers, and captions**).

Quiz Time


Join at menti.com | use code 7841 9022
Mentimeter
Instructions
Go to
www.menti.com
Enter the code
7841 9022
Or use QR code

Thank You

Q & A

suresh.n@sliit.lk | 755841849