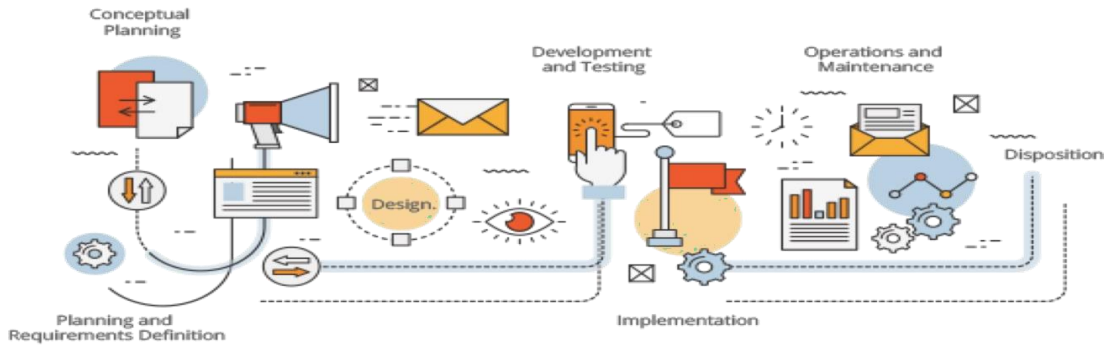
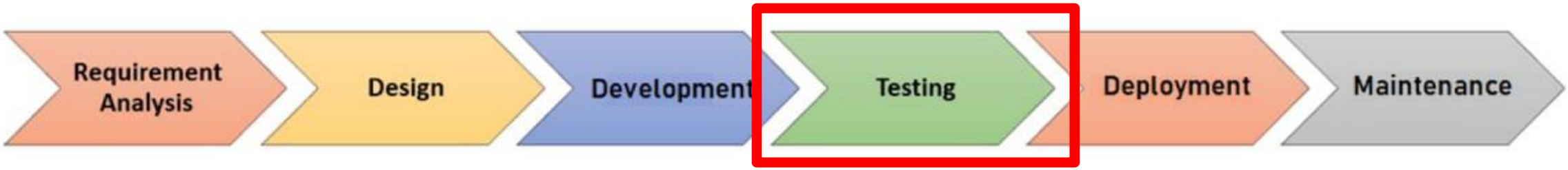
Software Engineering

Requirement Engineering



Software Development Life Cycle (SDLC)



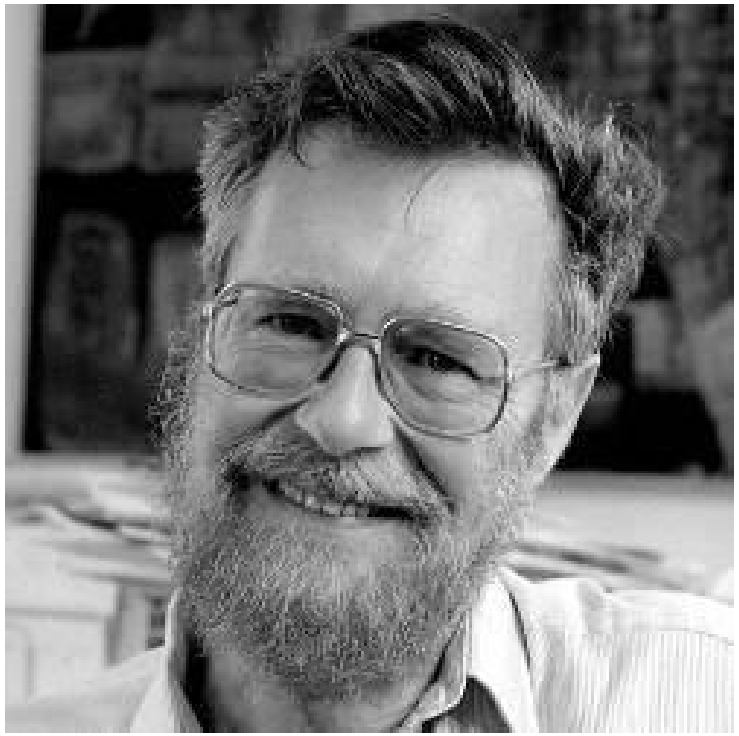
Testing Overview

“Microsoft, in terms of this quality stuff – we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing.”

“We're more of a testing, a quality software organization than we're a software organization.”

- Bill Gates, Information Week Interview, May 2002

Testing Overview



““… program testing can be used very effectively to show the presence of bugs but never to show their absence.”

- E. W. Dijkstra in EWD303

What is Software Testing?

- Testing is intended to show that a program does what it is intended to do
- Discover program defects before it is put into use.
- To achieve this, you execute the program with artificial data
- You check the results of the test run for errors, anomalies or information about program's non-functional attributes.

What is Testing and What is it Not?

- **Testing is a process for finding semantic or logical errors as a result of executing a program**
  - A run-time process, not a compile-time process
- **Testing is not aimed at finding syntactic errors**
  - The code is expected to be working code
  - Static checking, prior to run time, is separate
- **Testing improve software quality**
  - Testing helps identify defects or bugs in the software. By detecting and fixing these defects before the software is released to users, the overall quality of the software improves.
- **Testing can reveal the presence of errors, not their absence**
  - If your tests don't find errors, then get more effective tests

Software Testing in Practice

- **Testing amounts to 40% to 80% of total development costs***
  - 40% for information systems
  - 80% for real time embedded systems
- **Testing receives the least attention and often not enough time and resources**
  - Testers are often forced to abandon testing efforts because changes have been made
- **Testing is (usually) at the end of the development cycle**
  - Often rushed because other activities have been late

*Source: David Weiss, Iowa State

1. NASA Mars Climate Orbiter (1999)

- **Failure:** The spacecraft burned up in Mars' atmosphere.
- **Cause:** One team used imperial units (pounds) while another used metric units (newtons).
- **Lack of Testing:** No end-to-end system testing to catch unit mismatches.
- **Impact:** $125 million mission failure.

2. Ariane 5 Rocket Explosion (1996)

- **Failure:** Rocket self-destructed seconds after launch.
- **Cause:** Software reused from Ariane 4, but an untested integer overflow led to failure.
- **Lack of Testing:** No simulation of Ariane 5's faster acceleration.
- **Impact:** $370

4. Healthcare.gov Launch Failure (2013)

- **Failure:** The US government's health insurance website crashed under high traffic.
- **Cause:** Poor performance testing; couldn't handle thousands of users.
- **Lack of Testing:** No large-scale load testing before launch.
- **Impact:** Cost overrun and public embarrassment.

5. Therac-25 Radiation Overdose (1985-1987)

- **Failure:** A medical radiation therapy machine gave patients fatal overdoses.
- **Cause:** Software bug caused incorrect dosage calculations.
- **Lack of Testing:** No proper validation of safety-critical systems.
- **Impact: Six patient deaths**; medical industry reform.

6. Toyota Prius Brake Software Bug (2009-2010)

- **Failure:** Some Toyota vehicles had delayed braking response.
- **Cause:** Software glitch in the electronic braking system.
- **Lack of Testing:** Insufficient real-world testing of software.
- **Impact: 8.1 million car recalls**, lawsuits, and brand damage.

7. Samsung Galaxy Note 7 Battery Explosions (2016)

- **Failure:** Batteries in the Note 7 overheated and caught fire.
- **Cause:** Battery management software failed to detect faulty batteries.
- **Lack of Testing:** Inadequate stress testing under real-world conditions.
- **Impact: $17 billion loss**, recall of all Note 7 devices.

8. Boeing 737 MAX MCAS Software Failure (2018-2019)

- **Failure:** Faulty **MCAS (Maneuvering Characteristics Augmentation System)** caused two plane crashes.
- **Cause:** Software automatically pushed the nose down based on a faulty sensor reading.
- **Lack of Testing:** No rigorous real-world safety validation.
- **Impact: 346 deaths**, global grounding of 737 MAX, and **$20 billion in losses**.

Key Lessons from These Failures:

- **Rigorous Testing is Essential** – Bugs in production can lead to catastrophic failures.
- **System-Wide Validation Matters** – Small untested changes can have major ripple effects.
- **Performance & Load Testing is Crucial** – Websites, finance systems, and cars must handle real-world stress.
- **Human Lives Depend on Software** – Safety-critical systems (aviation, healthcare) need **extensive validation**.

Good tests have …

Desirable properties of tests

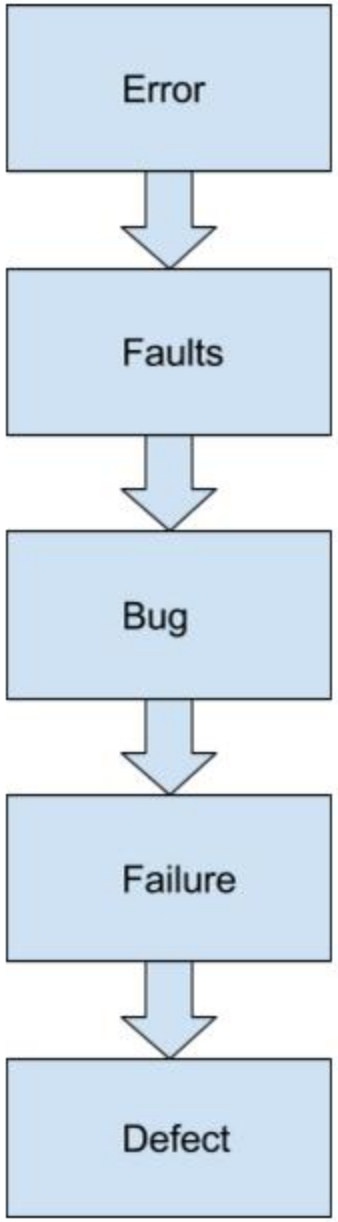
1.Power: This refers to the effectiveness of the tests in uncovering problems within the software. Effective tests should have the ability to detect a wide range of issues.
2.Validity: The problems identified by the tests should indeed be genuine issues that impact the functionality, performance, or other aspects of the software.
3.Value: The tests should reveal information that is relevant and important to the clients or end-users of the software. This ensures that the testing effort aligns with the needs and expectations of stakeholders.
4.Credibility: The tests should simulate likely operational scenarios, meaning they should closely resemble real-world usage conditions to provide meaningful results.
5.Non-redundancy: Each test should provide new and unique information about the software. Redundant tests that cover the same scenarios or functionalities do not add value and may waste resources.
6.Repeatability: Tests should be easy and inexpensive to re-run, allowing for consistent and reliable results. This is crucial for regression testing and ensuring that fixes or changes do not introduce new issues.
7.Maintainability: Tests should be easily revisable as the software evolves or is revised. This ensures that the testing effort remains relevant and effective

8. **Coverage**: Tests should exercise the product in ways that have not already been tested, ensuring comprehensive coverage of different functionalities, inputs, and scenarios.
9. **Ease of Evaluation**: Test results should be easy to interpret, allowing stakeholders to quickly understand the findings and take appropriate actions.
10. **Diagnostic Power**: Tests should help pinpoint the root cause of identified problems, facilitating effective debugging and resolution.
11. **Accountability**: It should be possible to explain, justify, and prove that the tests were conducted as planned, demonstrating accountability and adherence to testing processes.
12. **Low Cost**: The development and execution of tests should require reasonable time and effort, ensuring that the testing process remains cost-effective.
13. **Low Opportunity Cost**: The time and resources invested in testing should provide better value compared to other potential activities or tasks that could be undertaken.

These characteristics collectively contribute to effective and efficient software testing, ensuring that the testing effort adds value, identifies genuine issues, and supports the overall quality and reliability of the software.

Terminology

- **Error** is a deviation from the actual and the expected result. It represents the mistakes made by the people.
- **Faults** are the result of an error. It is the incorrect step or process due to which the program or the software behaves in an unintended manner
- **Bug** is an evidence of Fault in a program due to which program does not behave in an intended manner
- **Failure** is an inability of the system or components to perform its required function. Failure occurs when Faults executes
- **Defect** is said to be detected when Failure occurs.



An example of failure, fault and error.

```
pre: param is an integer.
post: returns the product of the param multiplied by 2.

1. int double (int param) {
2.   int result;
3.   result = param * param;
4.   return result;
5. }
```

- A call to double(3) returns 9, but the post condition says it should return 6.
- Result 9 represents a **failure**.
- The failure is due to the **fault** at line 3, ("* param" is used instead of "* 2")
- The **error** is a typo, (someone typed "* param" instead of "* 2" by mistake)

Terminology

- **Fault avoidance**

  Build systems with the objective of creating fault-- free (bug-- free) software.

- **Fault detection (testing and verification)**

  Detect faults (bugs) before the system is put into operation or when discovered after release.

- **Fault tolerance**

  Build systems that continue to operate when problems (bugs, overloads, bad data, etc.) occur.

Fault Avoidance Techniques

Fault avoidance is the practice of designing and developing software to minimize the introduction of faults (bugs) in the first place. Here are some examples:

1. **Formal Methods** – Using mathematically rigorous techniques such as model checking or theorem proving to ensure correctness before implementation.
2. **Code Reviews & Pair Programming** – Ensuring multiple developers review the code to identify potential issues early.
3. **Strict Coding Standards** – Enforcing coding guidelines and best practices to prevent common errors.
4. **Use of Proven Design Patterns** – Implementing well-established software design patterns to avoid known pitfalls.
5. **Automated Static Analysis** – Running tools that analyze source code for potential errors before compilation.
6. **Defensive Programming** – Writing code that anticipates potential issues, such as invalid inputs or unexpected states.
7. **Comprehensive Documentation** – Maintaining clear and detailed documentation to reduce misunderstandings in development.
8. **Modular Design & Encapsulation** – Designing software in independent, reusable components to reduce complexity and errors.
9. **Continuous Integration & Automated Builds** – Ensuring frequent, automated testing to detect integration issues early.

Fault detection techniques

1. **Unit Testing** – Testing individual components or functions to ensure they work as expected.
2. **Integration Testing** – Checking if multiple components work together correctly.
3. **System Testing** – Validating the entire system against the specified requirements.
4. **Regression Testing** – Running tests to ensure that new changes do not break existing functionality.
5. **Static Code Analysis** – Using automated tools to detect coding errors, security vulnerabilities, and performance issues.
6. **Formal Verification** – Using mathematical proofs and model checking to verify software correctness.
7. **Code Reviews & Walkthroughs** – Having peers review code to catch errors before execution.
8. **Fuzz Testing** – Inputting random or unexpected data to uncover crashes or unexpected behavior.
9. **Alpha & Beta Testing** – Allowing internal and external users to test the software before final release.

Fault tolerance

1. **Replication** – Using multiple hardware components (e.g., multiple processors, power supplies) so that if one fails, others take over.
2. **Checkpointing & Rollback Recovery** – Saving system states at intervals so it can restart from the last checkpoint in case of failure.
3. **N-Version Programming** – Running multiple, independently developed versions of a program and comparing results to detect errors.
4. **Try-Catch Blocks** – Handling unexpected runtime errors without crashing the entire system.
5. **Microservices Architecture** – Breaking applications into smaller, independent services to prevent system-wide failures.

Test Cases

- ***"A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement." (IEEE)***

  - A test case is a detailed set of conditions or steps that are designed to verify whether a specific aspect or functionality of a software application behaves as expected. Test cases are written based on requirements, user stories, or design specifications and are used to validate that the software meets its intended goals and functions correctly.

Test case No:	Scenario	Data Value 1	Data Value 2	Expected Result	Test
1	As a User, I want to create a category with a particular colour	Click on the plus button → Enter Category name → Choose Colour	Click on the Add button → Press back button	Data Value1: Display the Category in dropdown list Data value 2: Come back to home page	Pass
2	As a User, I want to select the category created and input an activity to it	Select Category from Drop Down List → Enter Activity name, description, date, time , duration → Save	Click on View Button	Data value1: Category selected and data values inserted Data value2:View entered details on recycler view	Pass
3	As a User, I want to a login page with pin option	Click on register button → Enter Credentials and register	Click on Back to Login Button	Data value1: Credentials captured to DB Data value2: Going back to login page	Pass
4	As a User, I want to check if I can login with the registered PIN only	Enter any PIN on the respective field add	Press Login button	Data value1: PIN displayed as **** Data value2: Login result	Pass
5	As a User, I want to export activities/categories and all details to a PDF file	Press download button on respective page	Press View button to open the PDF inside the app	Data value1: Message displaying PDF is generated Data value 2: Opening PDF on application itself	Pass
6	As a user, I need to email the PDF to anyone as we wish	Press Email PDF button and fill all required fields	Attach file and Press send button	Data value 1: Redirecting to Email creation page and entering all fields Data value 2: Attachment successful and going to email clients when pressing send Button	Pass
7	As a user, I want to filter activities created category wise	Go to overview page, select a category and type first letter to activity already created Press Search button	Press Search Button	Data value 1: Activity pops up automatically if match found with first letter Data value 2: Displays all activities as per the details entered	Pass
8	As a user, I want to see my activities as a pie chart	Go to Overview page and type in the required date	Press PIE CHART button	Data value 1: Date captured Data value 2: Pie chart displayed with percentage in terms of activity duration	Pass

How to write good test cases

When you begin writing test cases, there are a few steps that you need to follow to ensure that you are writing good test cases

1. **Identify the purpose of testing**

   You need to understand the requirements to be tested. Writing test cases for a module requires that you understand the features/user requirements for that module.

2. **Define how to perform the testing**

   This involves developing typical scenarios of use.

3. **Identify any non-functional requirements**

   Understand other aspects of the software related to non-functional requirements.

Structuring you Test Cases

1. **Test Case ID**: A unique identifier for the test case, often a combination of letters and numbers for easy reference.
2. **Test Case Description**: A brief description of the test case, outlining its purpose and objective.
3. **Preconditions**: Any necessary conditions that must be met before the test case can be executed, such as setting up specific configurations or data.
4. **Inputs**: The input data or parameters that will be used as part of the test case.
5. **Expected Results**: The expected outcome or behavior of the software when the test case is executed successfully.
6. **Steps**: The detailed steps or actions that need to be performed to execute the test case, including any specific interactions with the software interface or functionality.
7. **Actual Results**: The actual outcome or behavior observed when the test case is executed, which is compared against the expected results to determine if the test case passed or failed.
8. **Status**: The status of the test case (e.g., Pass, Fail, Blocked, In Progress) indicating whether the test case has been executed and its results.
9. **Notes**: Any additional comments, observations, or notes related to the test case execution, including any defects or issues encountered.

Use Case: User Login
Actor: User
Goal: The user wants to access the application by logging in with valid credentials.

Main Flow:

1. The user navigates to the login page of the application.
2. The system presents the user with input fields for username and password.
3. The user enters their valid username and password into the respective fields.
4. The user clicks on the "Login" button.
5. The system verifies the entered credentials.
6. If the credentials are valid, the system logs in the user and redirects them to the dashboard/homepage.
7. The system displays the user's name or profile information on the dashboard/homepage to indicate a successful login.
8. The user interacts with the application's features and functionalities on the dashboard/homepage.

A test case for a hypothetical login functionality in a web application

Test Case ID: TC001
Test Case Description: Verify that a user can successfully log in to the application using valid credentials.

Preconditions:
- The web application is accessible.
- The user has valid login credentials (username and password).

Inputs:
- Username: [valid username]
- Password: [valid password]

Expected Results:
- After entering valid credentials and clicking the "Login" button, the user should be redirected to the dashboard/homepage of the application.
- The user's name or profile information should be displayed to indicate that the login was successful.

Steps:

1. Open the web browser and navigate to the login page of the application.
2. Enter the valid username into the "Username" field.
3. Enter the valid password into the "Password" field.
4. Click on the "Login" button.
5. Wait for the application to process the login request.
6. Verify that the user is redirected to the dashboard/homepage.
7. Verify that the user's name or profile information is displayed on the dashboard/homepage.
8. Verify that no error messages or warnings are displayed indicating a failed login attempt.

Actual Results:

- The user is successfully redirected to the dashboard/homepage.
- The user's name "John Doe" is displayed on the dashboard/homepage.
- No error messages or warnings are displayed.

Status: Pass

Notes:

- Test case executed on [date] by [tester name].
- No defects or issues encountered during test execution.
- The login functionality appears to be working as expected with valid credentials.

R-001:

The users should be able to add two numbers and view their result on the display.

Use Case: *UC01*	Add Two Numbers
Actors:	**User**
Purpose:	**Add two numbers and view their result**
Overview:	**The user inputs two numbers and (then adds them and) checks the result, displayed on the screen.**
Type:	**Primary, Real**
Cross References:	**R-001**

Typical Course of Events

Actor Action	System Response
1. The actor opens the calculator. The keypad and display screen appears.	
2. The actor input the first number by clicking on the keypad or using keyboard.	3. The digit is displayed on the screen.
4. The actor clicks or presses the "+" key.	
5.The actor then adds the second number as (2).	6. The pressed digit is displayed on the screen.
7. The actor clicks the "=" key.	8. The sum of the two digits is displayed on the screen.

USE CASE FOR ADDING NUMBERS

Test Case ID: T-101 Test Item: Add Numbers

Wrote By: (tester name) Junaid Documented Date: 26th April 2005

Test Type: Manual Test Suite#: NA

Product Name: Windows Calculator Release and Version No.: V 1.0

Test case description:

Add any two Numbers

Operation procedure:

Open Calculator

Press "1"

Press "+"

Press "2"

Press "="

Pre-conditions: Post-conditions:

Calculator Opened Result Displayed

Inputs data and/or events: Expected output data and/or events:

1 + 2 = 3

Required test scripts (for auto): NA

Cross References: (Requirements or Use Cases) R-001, UC-01

Test Case for Adding Numbers