

# **How to test any application for maintain quality in Japan**

**Class 10**  
**29/3/2025**

# Acknowledgement

**The series of the IT & Japanese language course is  
Supported by AOTS and OEC.**



Ministry of Economy, Trade and Industry



Overseas Employment Corporation

# What you have Learnt Last Week

**We were focused on following points.**

- Usage of control and loop flow statement
- Performing Linear Algebra in Numpy
- Inspecting and Understanding Data
- Why Requirement Analysis is so important in the process?
- Review case studies that demonstrate successful requirement analysis practices
- Linear & Multi Linear Regression
- Support Vector Machine
- Cost Function

# What you will Learn Today

## **We will focus on following points.**

- Why Requirement Analysis is so important in the process?
- Review case studies that demonstrate successful requirement analysis practices
- Discuss the steps of testing process
- Types of testing such as unit testing, and integration testing etc.
- Explore the TDD process, including the steps of writing a failing test (Red), making it pass (Green), and then refactoring the code
- Quiz
- Q&A Session

# Software Development Life Cycle

The Software Development Life Cycle (SDLC) is a structured process used to design, develop, test, and deploy software applications

## [Phases of SDLC]

### 1. Requirement Analysis

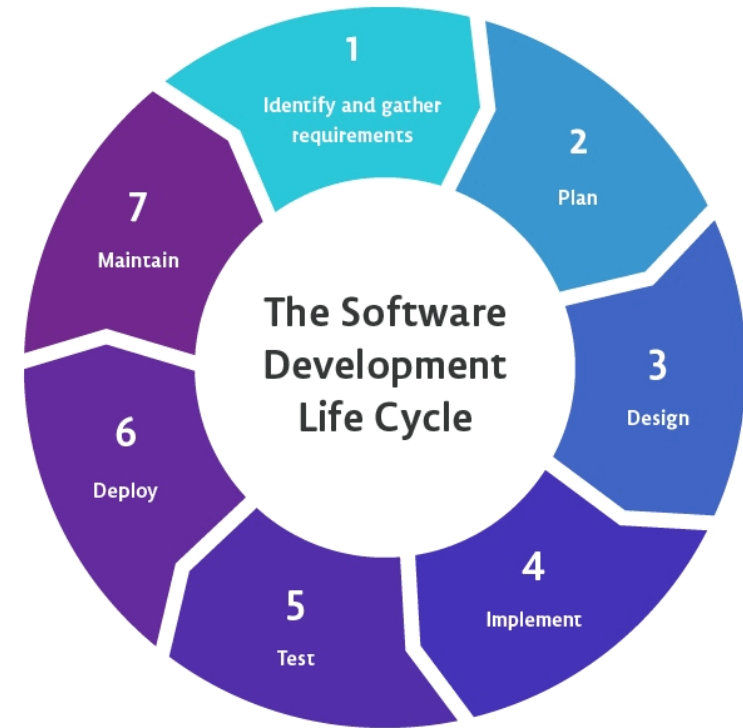
1. Gather business and technical requirements.
2. Identify stakeholders and project scope.

### 2. Planning

1. Define project timeline, cost estimation, and resource allocation.
2. Identify risks and mitigation strategies.

### 3. Design

1. Create architectural and UI/UX design.
2. Define database structures and system components.



# Software Development Life Cycle

The Software Development Life Cycle (SDLC) is a structured process used to design, develop, test, and deploy software applications

## 4. Development (Implementation)

1. Code the software based on the design specifications.
2. Follow best practices, use version control, and implement coding standards.

## 5. Testing

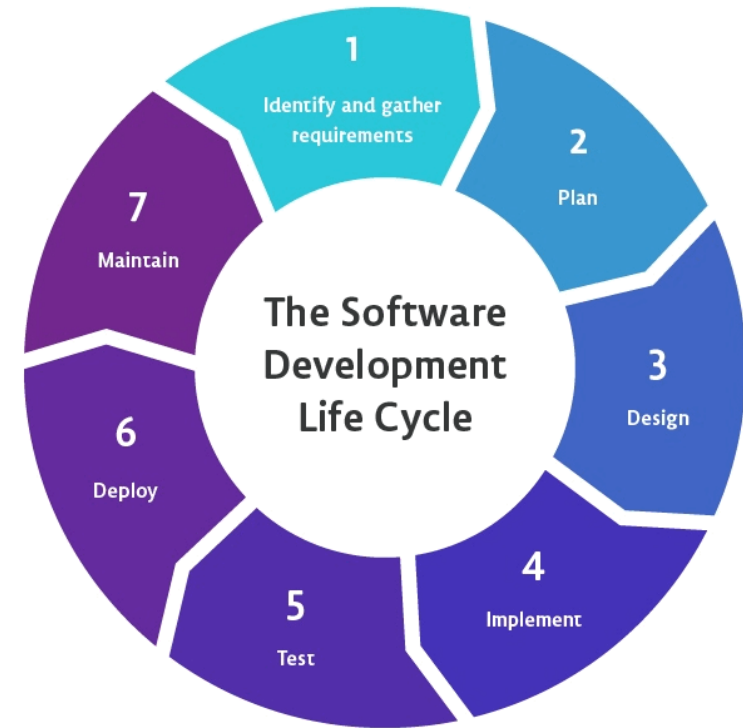
1. Perform unit testing, integration testing, and system testing.
2. Identify and fix bugs to ensure software reliability.

## 6. Deployment

1. Release the software to production.
2. Implement CI/CD pipelines for automated deployment.

## 7. Maintenance & Support

1. Monitor performance, fix bugs, and update features as needed.



Requirement Analysis is one of the most important part, Not only Coding/Programming

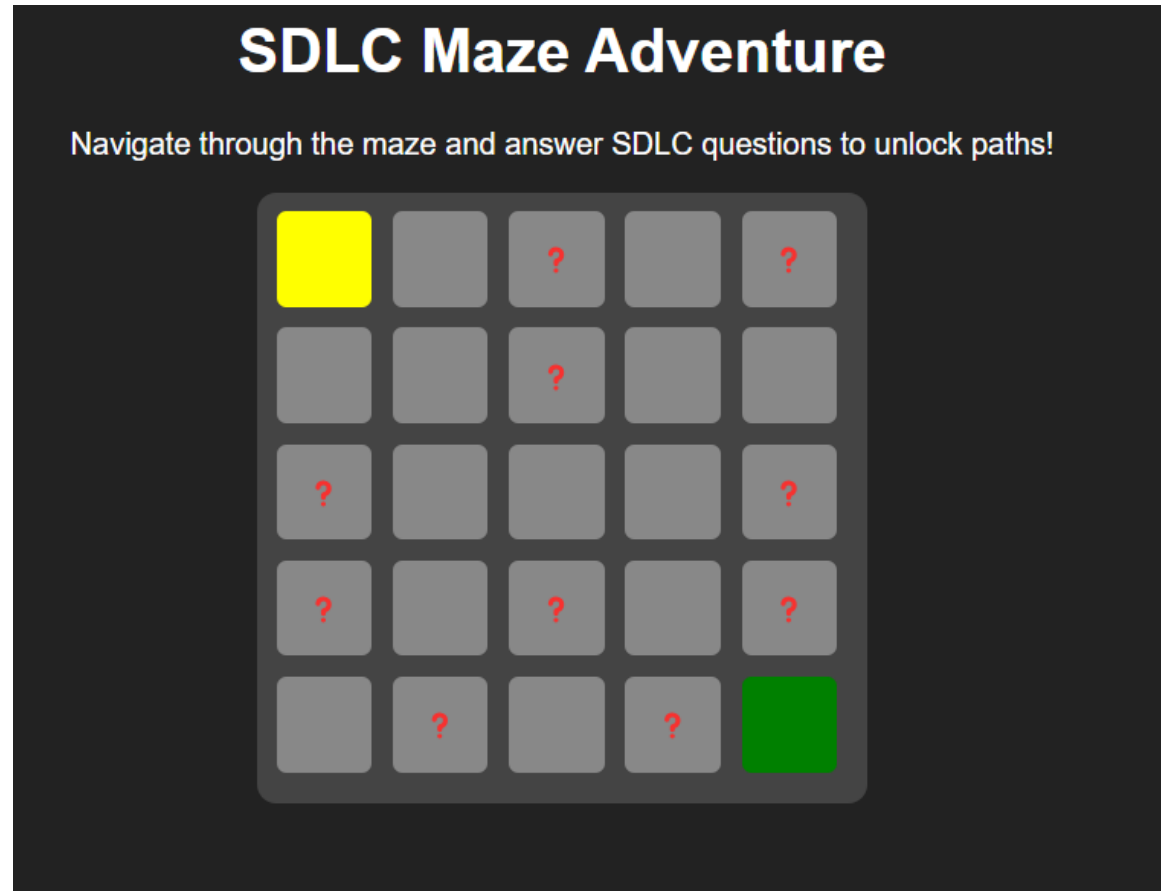


+W

# Task-1

# SDLC Maze Adventure

Let's learn the SDLC Maze Adventure by playing the game

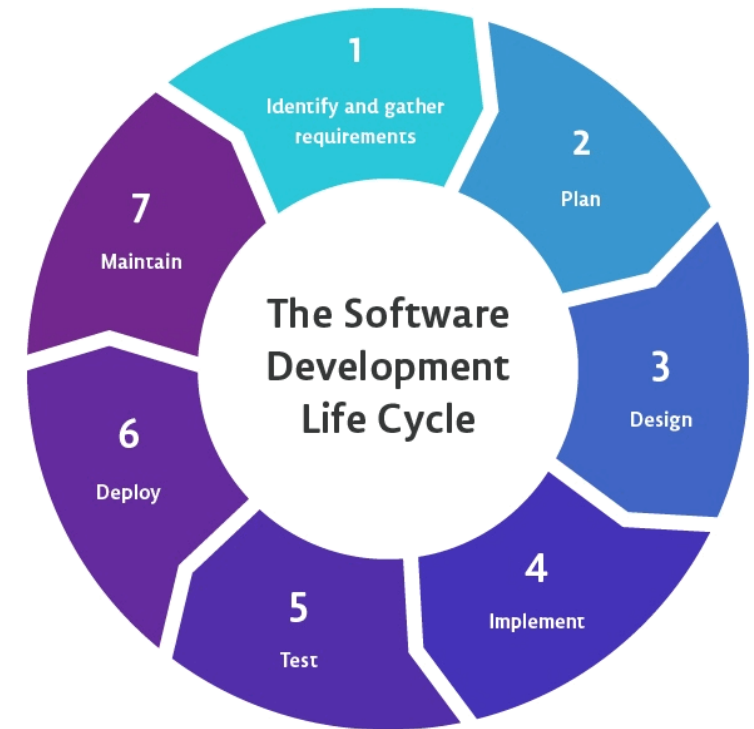




# What is Software Testing?

**The purpose of software testing is to ensure that a software application functions correctly**

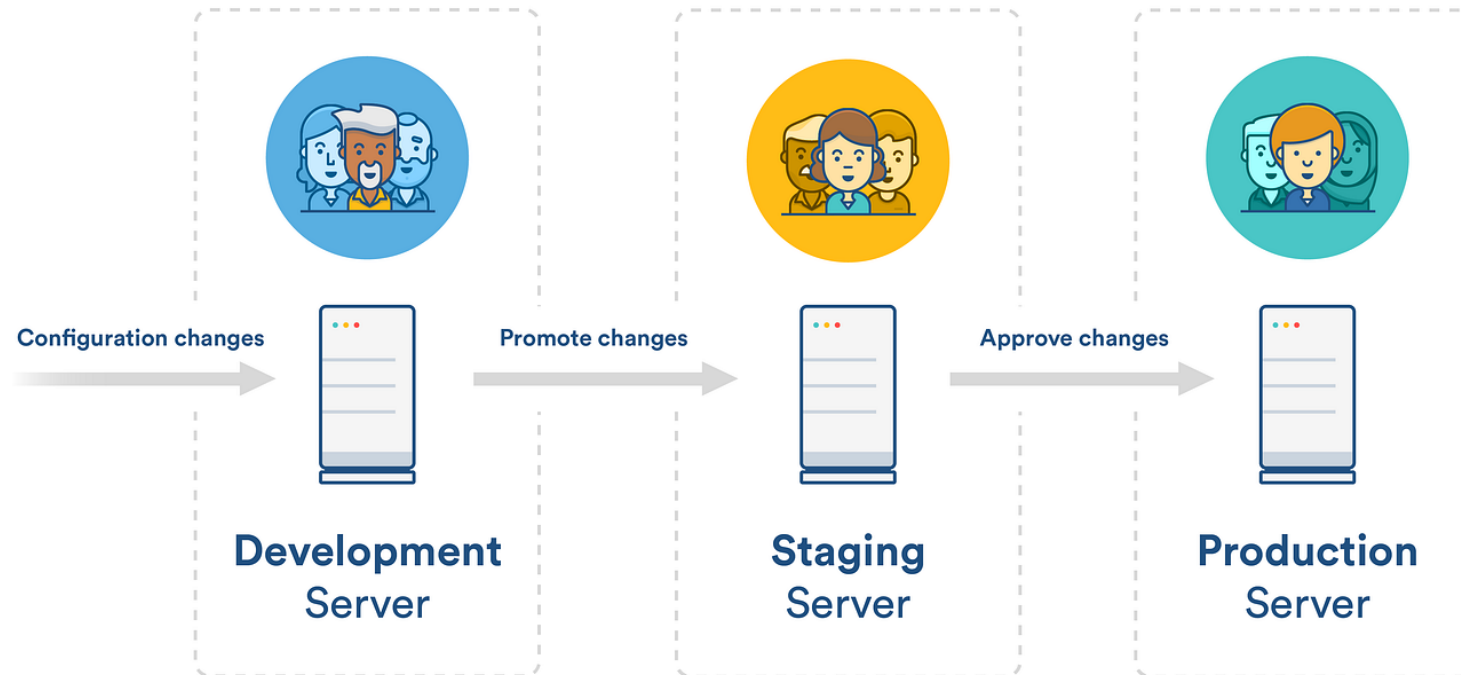
- Software testing is the process of evaluating a system or application to identify defects and ensure quality.
- It helps improve performance, security, and user experience.
- Testing can be manual or automated.
- Requirement Traceability Matrix (RTM) document helps in tracing requirements to test cases



Defect tracking is the process of **identifying, documenting, monitoring, and managing defects (bugs) found in software during testing and development**

# Dev Staging & Production workflow

A Dev → Staging → Production workflow ensures a smooth, structured, and secure software development lifecycle (SDLC) while minimizing risks during deployments



# Steps in Testing Process

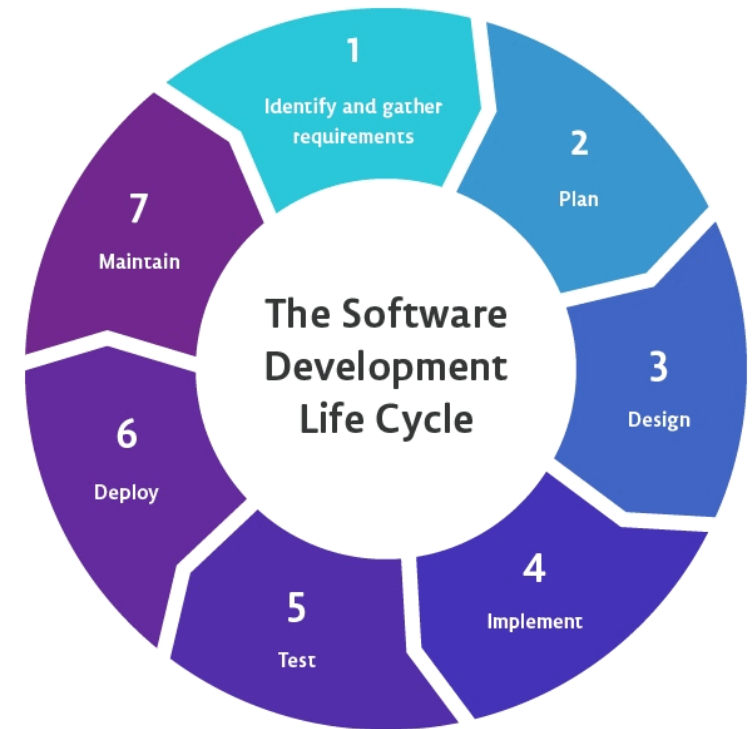
Following seven steps are involved in software testing

## 1. Requirement Analysis

- Understand project needs by collaborating with stakeholders, developers, and analysts.
- Example: A banking app requires secure login. Identify ambiguous details like password complexity rules.

## 2. Test Planning

- Define scope, resources, and testing types (unit, integration, system).
- Example: An e-commerce platform needs functional, performance, and security testing.



# Steps in Testing Process

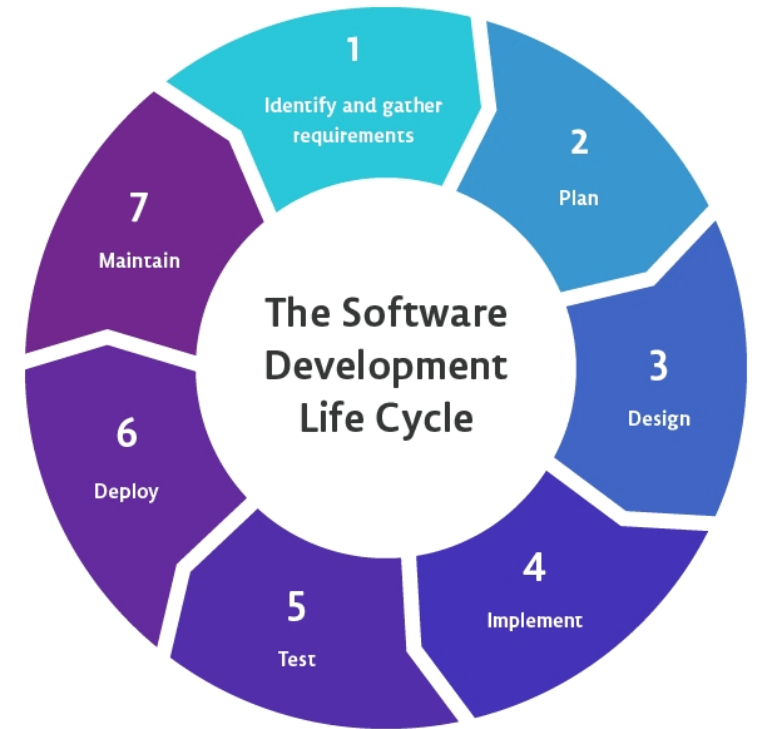
**Following seven steps are involved in software testing**

## **3. Test Case Development**

- Write test cases covering positive and negative scenarios.
- Example: For an "Add to Cart" button, test cases include:
  - Clicking the button adds an item.
  - Clicking multiple times increases quantity.
  - Adding an out-of-stock item gives an error.

## **4. Test Environment Setup**

- Configure the test setup, including hardware, software, and data.
- Example: A mobile banking app requires a test environment mimicking real-world network conditions.



# Steps in Testing Process

## Following seven steps are involved in software testing

### 5. Test Execution

- Run test cases and compare results with expectations.
- Example: A login test case verifies if a correct username/password logs in and an incorrect one shows an error.

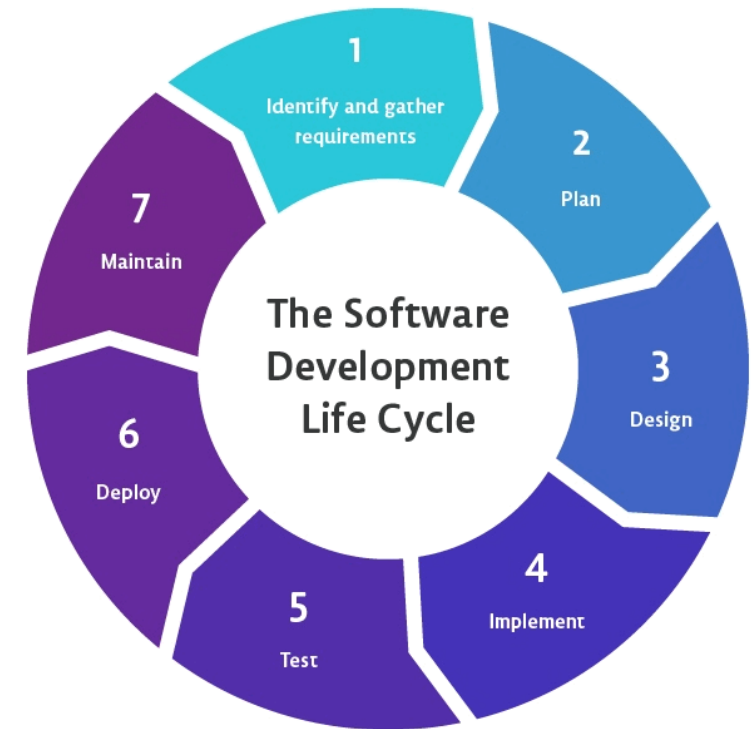
### 6. Defect Reporting & Tracking

- Log defects in a tool like JIRA, assign priorities, and track fixes.
- Example: A "Checkout" button not working is logged as a high-priority defect.

### 7. Test Closure

- Conduct review meetings, summarize key findings, and archive test reports.

Example: A test summary report for an HR system includes defect trends, pass/fail ratios, and improvement areas.

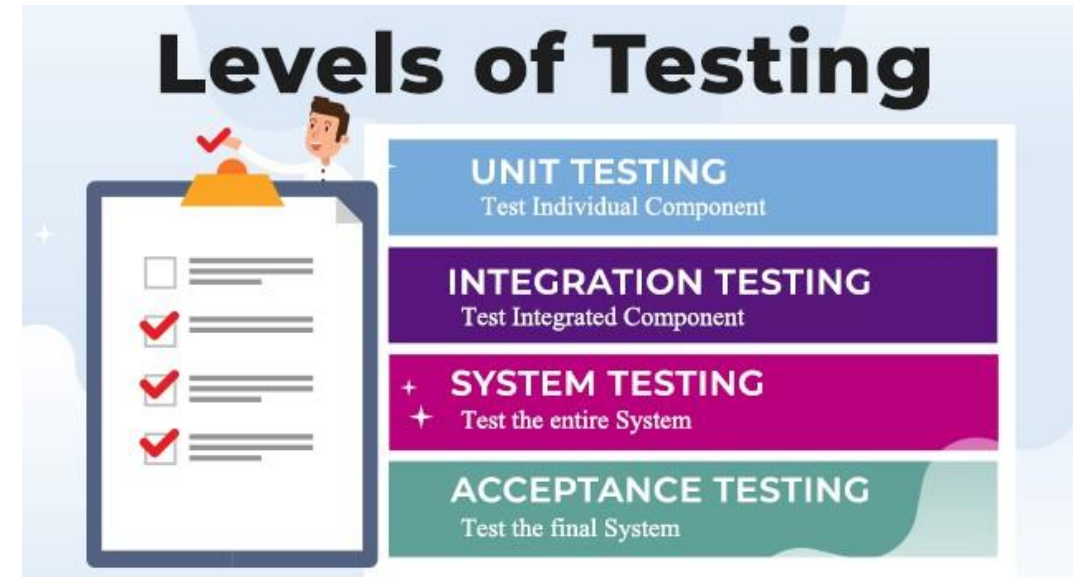




# Types of Software Testing

## Following four types are involved in software testing

1. **Unit Testing** - Tests individual components or modules.
  - **Smoke Test (Sanity-Test)**- basic functionalities of an application
  - **Regression Testing** - Ensures new changes do not affect existing functionality.
2. **Integration Testing** - Ensures different modules work together correctly.
  - **Performance Testing** - Assesses speed, scalability, and reliability.
3. **System Testing** - Evaluates the complete system's compliance with requirements.
  - **Security Testing** - Identifies vulnerabilities and ensures data protection.
4. **Acceptance Testing** - Determines if the software meets business needs.

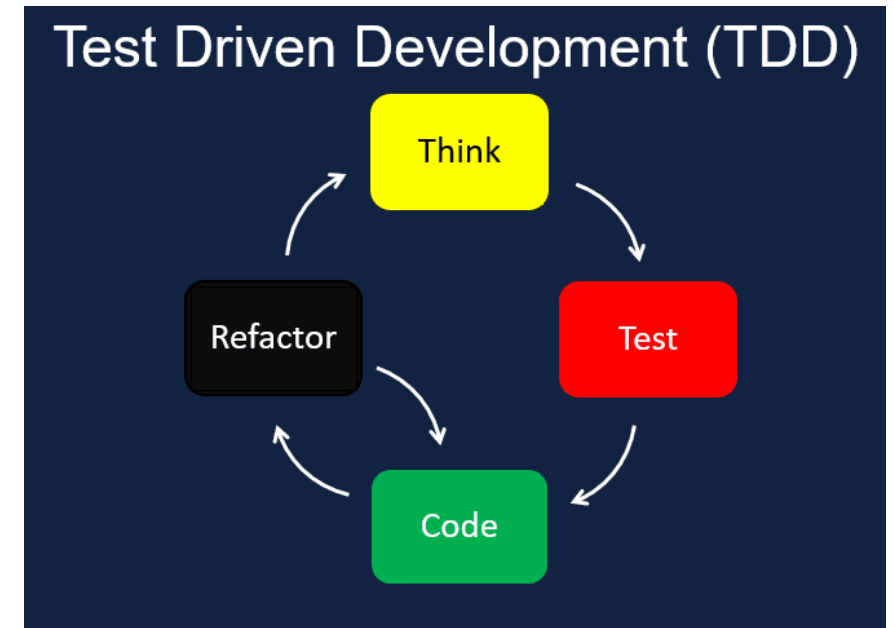


# Introduction to Test-Driven Development (TDD)

It is a software development approach that emphasizes writing tests before writing the actual code.




## TDD Cycle:

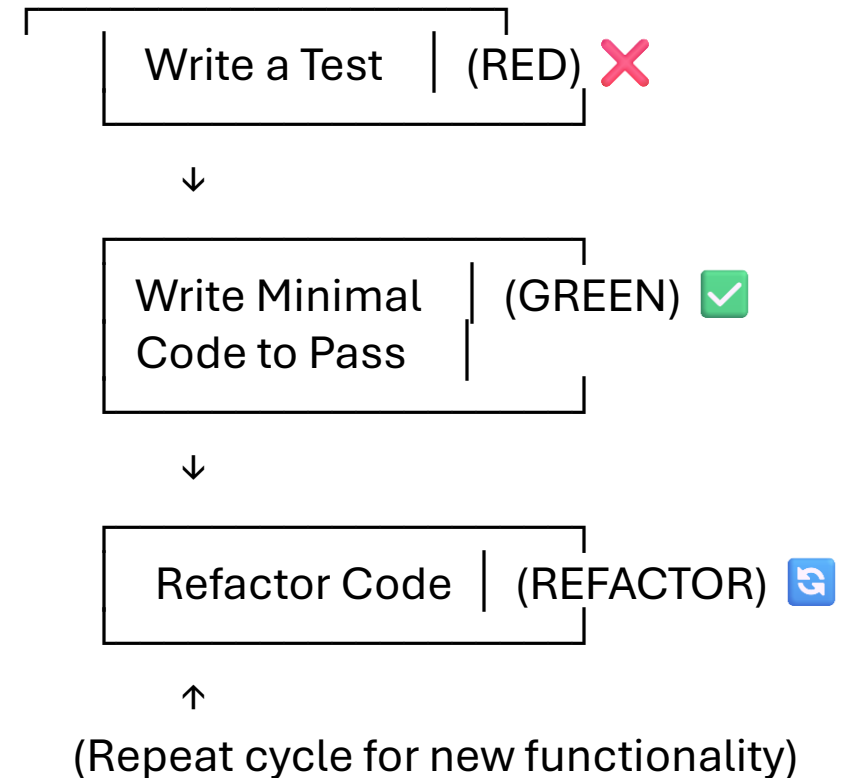
1. **Write a Failing Test** – First, a test is created based on the expected behavior of the function or feature. Since the implementation does not yet exist, the test will fail.
2. **Make the Test Pass** – The minimum amount of code is written to make the test pass.
3. **Refactor the Code** – The written code is improved for readability, efficiency, and maintainability while ensuring that the test still passes.



# TDD Process (Red-Green-Refactor)

It follows a simple Red-Green-Refactor cycle

- **Red Phase**  → Write a failing test based on requirements.
- **Green Phase**  → Write the minimum code to make the test pass.
- **Refactor Phase**  → Optimize and improve the code while keeping the test passing.



# Case:1 Implementing an add() function

**We write a test for a function `add(a, b)` that should return the sum of two numbers.**

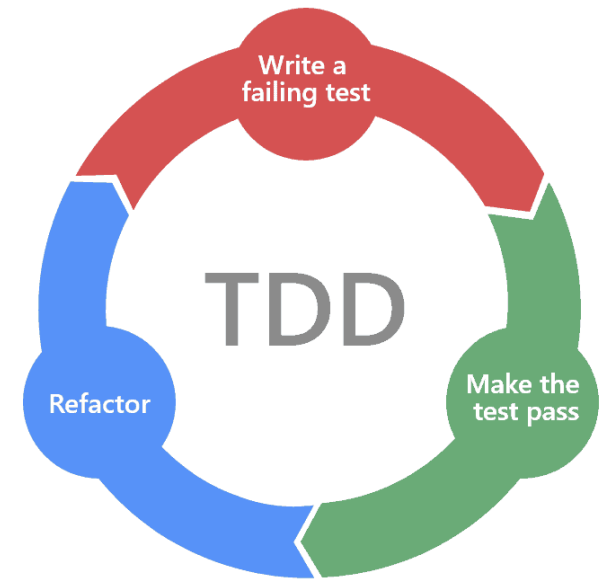
## **Step 1: Red Phase (Write a failing test)**

We write a test for a function `add(a, b)` that should return the sum of two numbers.

```
def test_add():  
    assert add(2, 3) == 5 # Expected output is 5
```

[Note]

● This test fails because `add()` does not exist yet.



# Case:1 Implementing an add() function

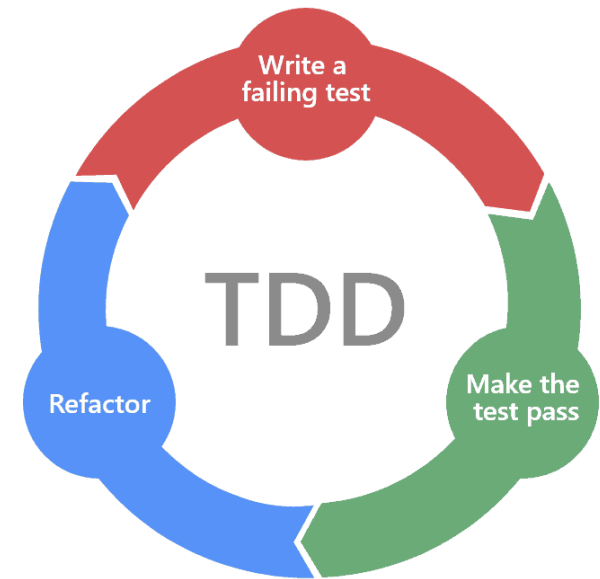
**We write a test for a function `add(a, b)` that should return the sum of two numbers.**

**Step 2: Green Phase (Write minimal code to pass the test)**

We now create a simple `add()` function that just returns the sum.

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

● The test passes now!





# Case:1 Implementing an add() function

**We write a test for a function add(a, b) that should return the sum of two numbers.**

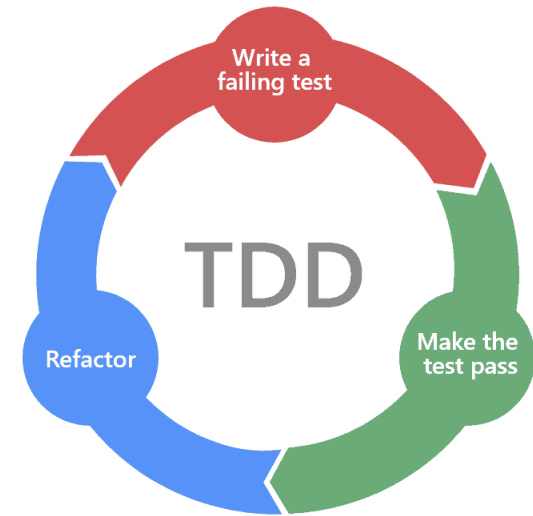
## Step 3: Refactor the Code (REFACTOR)

Now that the function works, we check if we can improve it (e.g., adding input validation):

```
def add(a, b):  
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):  
        raise ValueError("Inputs must be numbers")  
    return a + b
```

 **Run tests again to ensure nothing is broken.**

```
print(add(3, 5))      # Output: 8  
print(add(2.5, 1.5)) # Output: 4.0
```



# Case:2 Slack Notification Failure

**When a deployment is completed, message is not successfully sent to Slack.**

## Step 1: Red Phase (Write a failing test)

A test is written to verify that a message is successfully sent to Slack.

```
// slackNotifier.test.js

const slackNotifier = require('../slackNotifier');

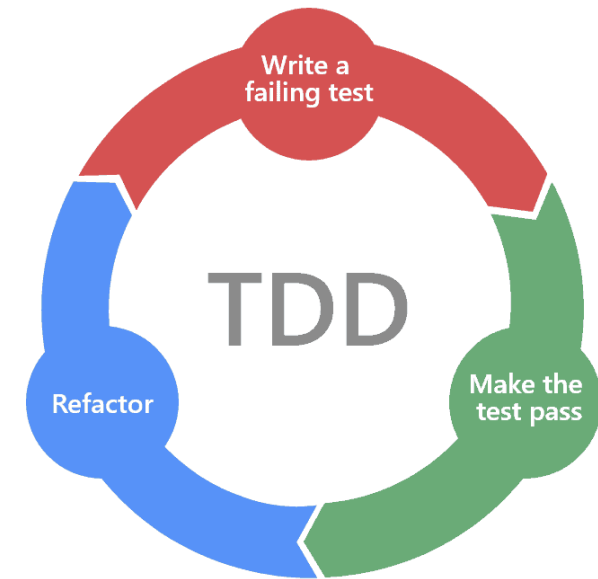
test("should send a Slack notification successfully", async ()
    const response = await slackNotifier.sendMessage("Deployme
successful!");

    expect(response.status).toBe(200);

});
```

✅ **Expectation:** The function should return **status 200** if the message is sent successfully.

❌ **Failure:** The test fails because `sendMessage` is not implemented yet.



# Case:2 Slack Notification Failure

When a deployment is completed, message is not successfully sent to Slack.

## Step 2: Green Phase (Write minimal code to pass the test)

```
// slackNotifier.js (Minimal Implementation)

const axios = require('axios');

const sendMessage = async (message) => {

  const response = await axios.post("https://slack.com/api/chat.postMessage", {

    channel: "#deployments",

    text: message}, {

    headers: { Authorization: `Bearer YOUR_SLACK_TOKEN` }

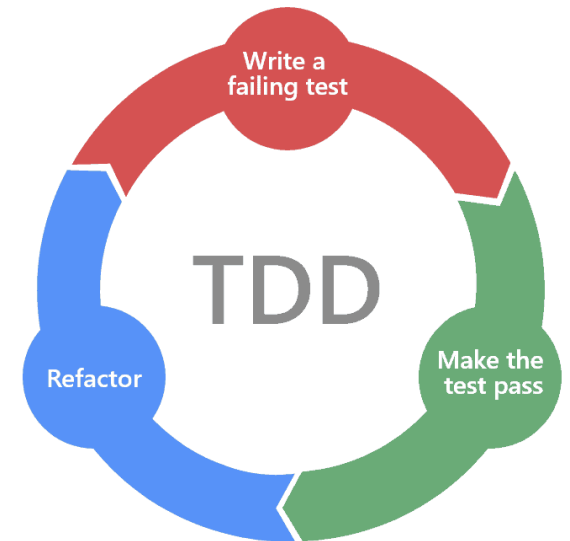
  });

  return response;

};

module.exports = { sendMessage };
```

🔄 Run the test again → ✅ Passes!



# Case:2 Slack Notification Failure

When a deployment is completed, message is not successfully sent to Slack.

## Step 3: Refactor the Code (REFACTOR)

Now, the code is improved by:

1. Handling **failures** when Slack is unreachable.
2. **Logging errors** for debugging.

 Run the test again →  Still passes!

// slackNotifier.js (Refactored Version with Error Handling)

```
const axios = require('axios');

const sendMessage = async (message) => {

  try {const response = await axios.post("https://slack.com/api/chat.postMessage", {

    channel: "#deployments",

    text: message

  }, {}

  return response;

} catch (error) {

  console.error("Failed to send Slack notification:", error.message);

  return { status: 500, error: error.message };

}

};

module.exports = { sendMessage };
```

# Case:3 Implementing a User Authentication System

A startup wants to build a user authentication system with a login feature. They follow Test-Driven Development (TDD) to ensure reliability.

## Step 1: Red Phase (Write a failing test)

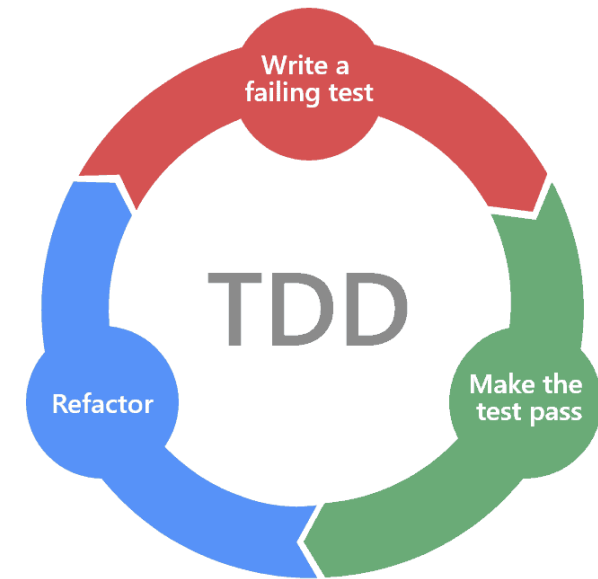
Before writing any implementation code, the team writes a test that defines the expected behavior:

```
// login.test.js (Jest Test)
const auth = require('../auth');

test("should return success for valid login credentials", () => {
  const result = auth.login("testuser", "correctpassword");
  expect(result).toBe("Login successful");
});
```

✅ **Expectation:** If valid credentials are provided, the function should return "Login successful".

❌ **Failure:** The test fails because the login function does not exist yet.





# Case:3 Implementing a User Authentication System

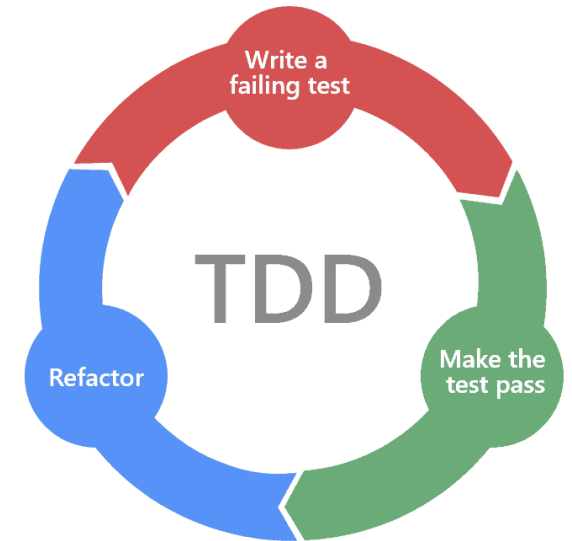
A startup wants to build a user authentication system with a login feature. They follow Test-Driven Development (TDD) to ensure reliability.

## Step 2: Green Phase (Write minimal code to pass the test)

Now, we implement just enough code to **make the test pass**.

```
// auth.js (Minimal Implementation)
const login = (username, password) => {
  if (username === "testuser" && password === "correctpassword") {
    return "Login successful";
  }
  return "Invalid credentials";
};
module.exports = { login };
```

🔄 Run the test again → ✅ Passes!




# Case:3 Implementing a User Authentication System

A startup wants to build a user authentication system with a login feature. They follow Test-Driven Development (TDD) to ensure reliability.

## Step 3: Refactor the Code (REFACTOR)

Now that the test is passing, we **refactor** the code to make it more robust and scalable.

 **Run the test again** →  **Still passes!**

 **Benefits:** Code is now more maintainable and can handle multiple users.

```
// auth.js (Improved Version with Secure Password Handling)

const users = [
  { username: "testuser", password: "correctpassword" } // In
  real apps, passwords should be hashed
];

const login = (username, password) => {
  const user = users.find(user => user.username === username);
  return user && user.password === password ? "Login
  successful" : "Invalid credentials";
};

module.exports = { login };
```

+W

# Task-2

# Case:4 Validating User Email Input Using TDD

You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.

## Step 1: Red Phase (Writing a Failing Test)

1. What test cases would you write to validate an email?
2. What inputs should pass, and what should fail?
3. Why does the test fail initially?

### Expected Cases:

- `"test@example.com"` → Should be **valid**
- `"invalid-email"` → Should be **invalid**
- `"user@domain"` → Should be **invalid**
- `"@missinguser.com"` → Should be **invalid**
- `"user.name@domain.com"` → Should be **valid**

# Case:4 Validating User Email Input Using TDD

You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.

- What test cases would you write to validate an email?

- The test cases should include valid and invalid email formats:

- "test@example.com" → **Valid**
- "invalid-email" → **Invalid**
- "user@domain" → **Invalid**
- "@missinguser.com" → **Invalid**
- "user.name@domain.com" → **Valid**

- Why does the test fail initially?

- The function `isValidEmail` does **not exist yet**, so when we run the test, it fails.
- Even after creating the function, if it does not correctly validate emails, the test will still fail.

- What inputs should pass, and what should fail?

- **Pass:**

- "test@example.com"
- "user.name@domain.com"
- "user123@sub.domain.co"

- **Fail:**

- "plainaddress"
- "@missinguser.com"
- "user@.com"
- "user@domain"



# Case:4 Validating User Email Input Using TDD

**You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.**

**Step 2: Green Phase  (Writing Minimum Code to Pass the Test)**

- 1. What is the simplest rule to validate an email?**
- 2. How can you ensure the test passes with minimal implementation?**
- 3. What is the risk of writing only minimal code?**

# Case:4 Validating User Email Input Using TDD

You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.

- **What is the simplest rule to validate an email?**
  - The simplest check is to see if the email contains "@".
  - Example: `return email.includes("@");`
- **How can you ensure the test passes with minimal implementation?**
  - Instead of writing a **complex regex**, start with a simple condition that makes the test pass.
- **What is the risk of writing only minimal code?**
  - A simple check like `email.includes("@")` will **incorrectly pass invalid emails**, such as `"user@domain"`.
  - It does not check for proper structure (e.g., missing a top-level domain like `.com`).

# Case:4 Validating User Email Input Using TDD

You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.

**Step 3: Refactor Phase**  (Optimizing Without Breaking Tests)

1. What improvements can you make without breaking the test?
2. Why is regular expressions (regex) a better way to validate emails?
3. What are the benefits of refactoring the code?

# Case:4 Validating User Email Input Using TDD

You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.

- What improvements can you make without breaking the test?
  - Use **regular expressions (regex)** to validate the email format properly.
  - A **regex-based check** ensures that the email follows the correct structure.
- Why is regular expressions (regex) a better way to validate emails?
  - A regex ensures:
    - ✓ At least one character before @
    - ✓ A valid domain name after @
    - ✓ A valid top-level domain like .com, .org, etc.
  - Example regex: `/^[^\s@]+@[^\s@]+\.[^\s@]+$/`
- What are the benefits of refactoring the code?
  - Improves **accuracy** of email validation.
  - Prevents **false positives** (invalid emails mistakenly considered valid).
  - Keeps the code **clean and maintainable** without affecting existing functionality.

# Outcomes

**You are designing a registration system that requires email validation. The system should check if an email is valid before accepting user signups.**

TDD helps in building robust, testable, and scalable validation systems. 🚀

By following the **Red-Green-Refactor** cycle, we ensure:

- ✓ Better software design
- ✓ Bug-free and maintainable code
- ✓ Efficient and testable systems
- ✓ Ensures the correctness of code before implementation
- ✓ Reduces bugs and debugging time
- ✓ Forces developers to think about requirements first
- ✓ Bugs are caught early in the development process

🚀 **TDD is a must-have practice for professional developers!**

# Benefits of TDD

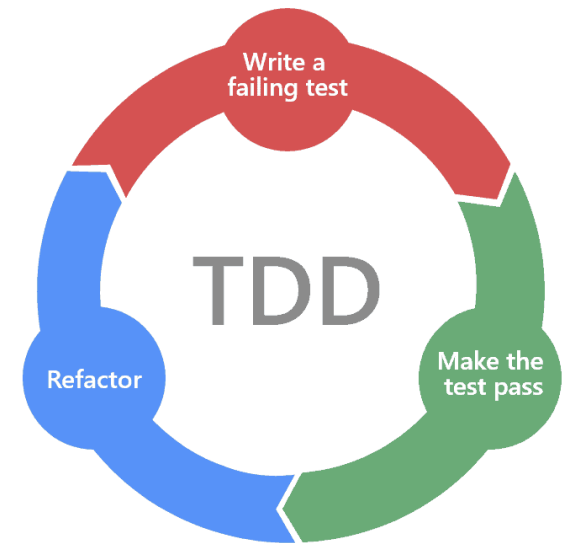
## Following benefits of TDD are

### 1. Ensures Code Correctness from the Start

- Writing tests first ensures that the code meets requirements before development.
- **Example:** A login function is developed only after writing test cases for valid and invalid credentials.
- **Outcome:** Fewer bugs in production.

### 2. Reduces Debugging Time and Maintenance Costs

- Since tests are written first, errors are caught early.
- **Example:** If a developer refactors code and a test fails, they immediately know what broke.
- **Outcome:** Less time spent on debugging and lower maintenance costs.



# Benefits of TDD

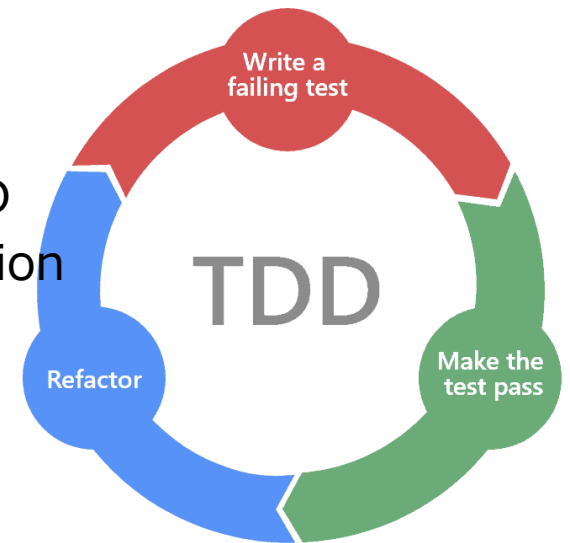
## Following benefits of TDD are

### 3. Encourages Better Design and Modular Code

- Forces developers to write smaller, testable functions, leading to better code structure.
- **Example:** Instead of a large function handling user authentication, TDD encourages separate modules for validation, database checks, and session handling.
- **Outcome:** Improved code reusability and maintainability.

### 4. Enhances Collaboration Between Developers and Testers

- Testers can define test cases before development, improving understanding.
- **Example:** Developers and QA teams collaborate on defining acceptance criteria for a shopping cart functionality before implementation.
- **Outcome:** Fewer misunderstandings, better team synergy.



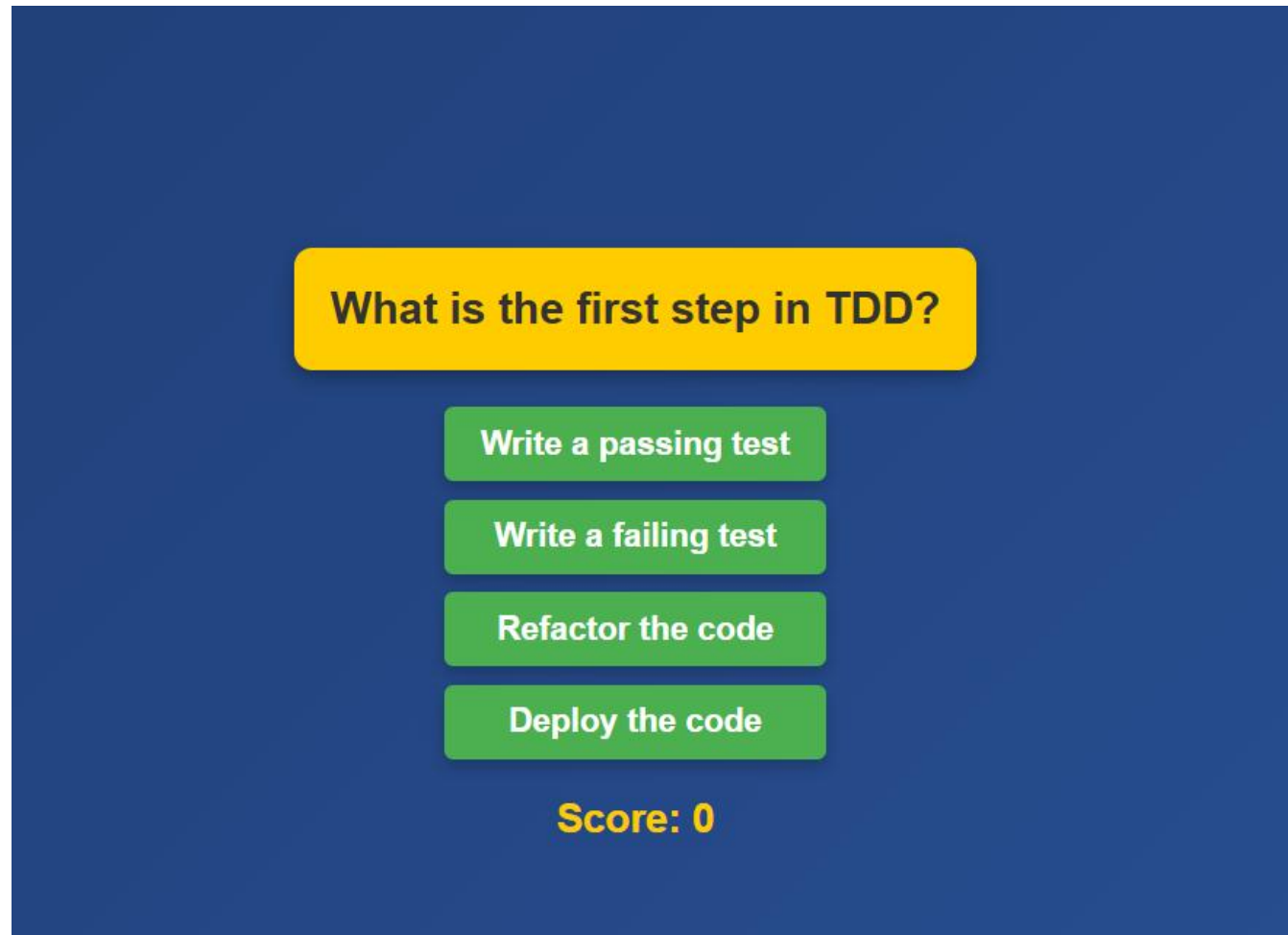


+W

# Task-3

# Software Testing and TDD process

Let's learn the Software testing and TDD process by playing the game



# Testing Phase in SDLC

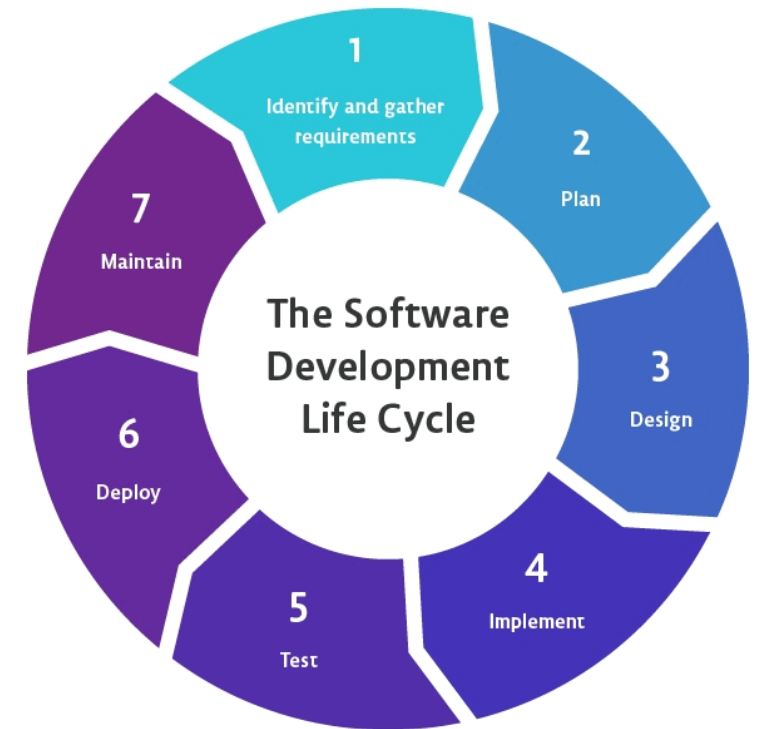
This phase ensures the software works as intended, is free of major bugs, and meets business requirements

## [Key Activities in the Testing Phase]

### 1. Perform Different Types of Testing

#### Unit Testing (Testing Individual Components)

- Tests **smallest units** (functions, methods, or classes).
- Ensures each part of the software **works in isolation**.
- Automated using tools like **Jest and Mocha (JavaScript)**, **JUnit (Java)**, **PyTest (Python)**.



It involves different levels of testing, such as **unit testing, integration testing, and system testing**, to verify that all components function correctly.



# Quiz Section

# Quiz

**Everyone student should click on submit button before time ends otherwise MCQs will not be submitted**

## **[Guidelines of MCQs]**

1. There are 20 MCQs
2. Time duration will be 10 minutes
3. This link will be share on 12:25pm (Pakistan time)
4. MCQs will start from 12:30pm (Pakistan time)
5. This is exact time and this will not change
6. Everyone student should click on submit button otherwise MCQs will not be submitted after time will finish
7. Every student should submit Github profile and LinkedIn post link for every class. It include in your performance

# Assignment

**Assignment should be submit before the next class**

## **[Assignments Requirements]**

1. Create a post of today's lecture and post on LinkedIn.
2. Make sure to tag @Plus W @Pak-Japan Centre and instructors LinkedIn profile
3. Upload your code of assignment and lecture on GitHub and share your GitHub profile in respective your region group WhatsApp group
4. If you have any query regarding assignment, please share on your region WhatsApp group.
5. Students who already done assignment, please support other students



# Q&A Session

ありがとうございます。

Thank you.

شكريا



For the World with Diverse Individualities