

Javascript Unit Testing and TDD Techniques

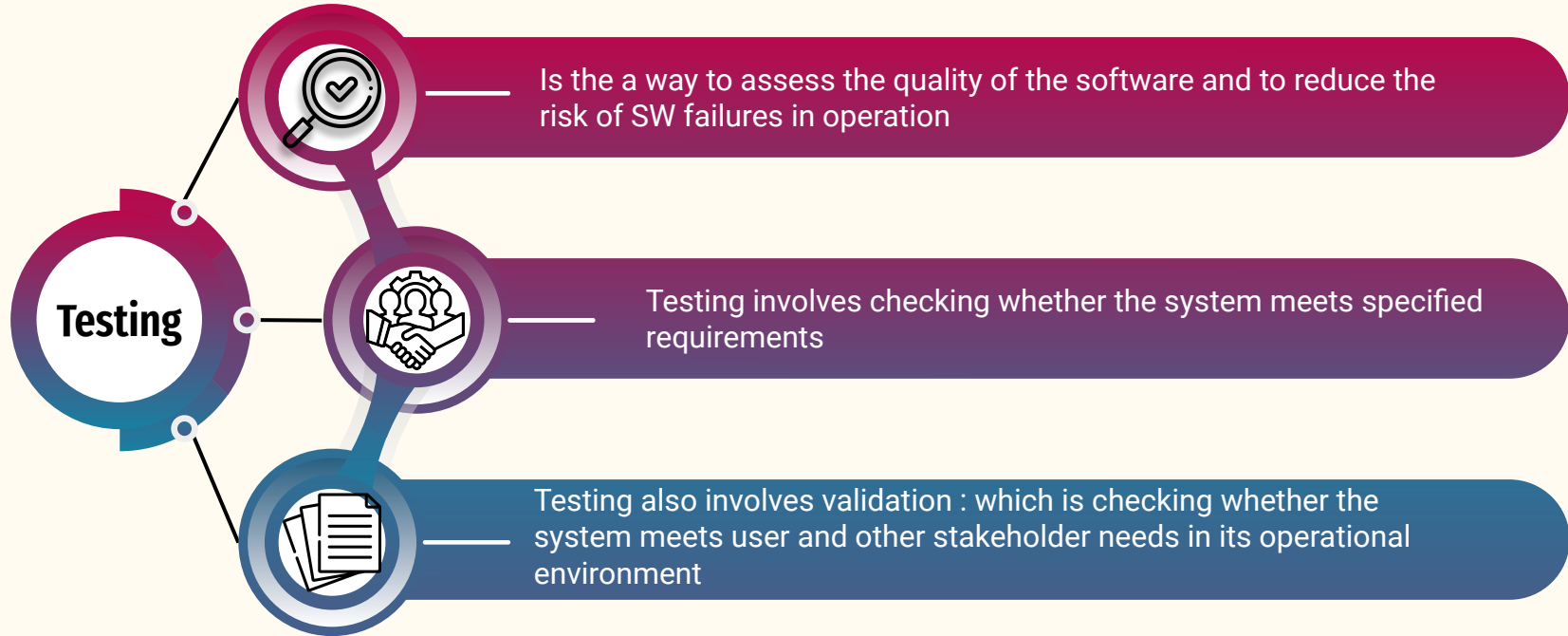
By Mona Mohsen

Course Content

- Testing Life Cycle
- Types of Test
- Unit Test Features
- Test Utility Code
- Structure Inspection and Test-Specific identity
- Writing unit test using Mocha
- Writing unit tests using Jasmine
- Introduction to TDD

What is the testing

What's the testing ?



Why do we test?

Why do we test?

- Most of people have had an experience with Sw that didn't work as expected
- SW that does not work correctly can lead to many problems including:
 - Loss of money, time, or business reputation, and even injury or death .
- To prevent defects by evaluate work products such as :
 - Requirements, User stories, Design, and Code
- To verify whether the test objective is complete, and works as the users and other stakeholder expect .

Objectives of Testing

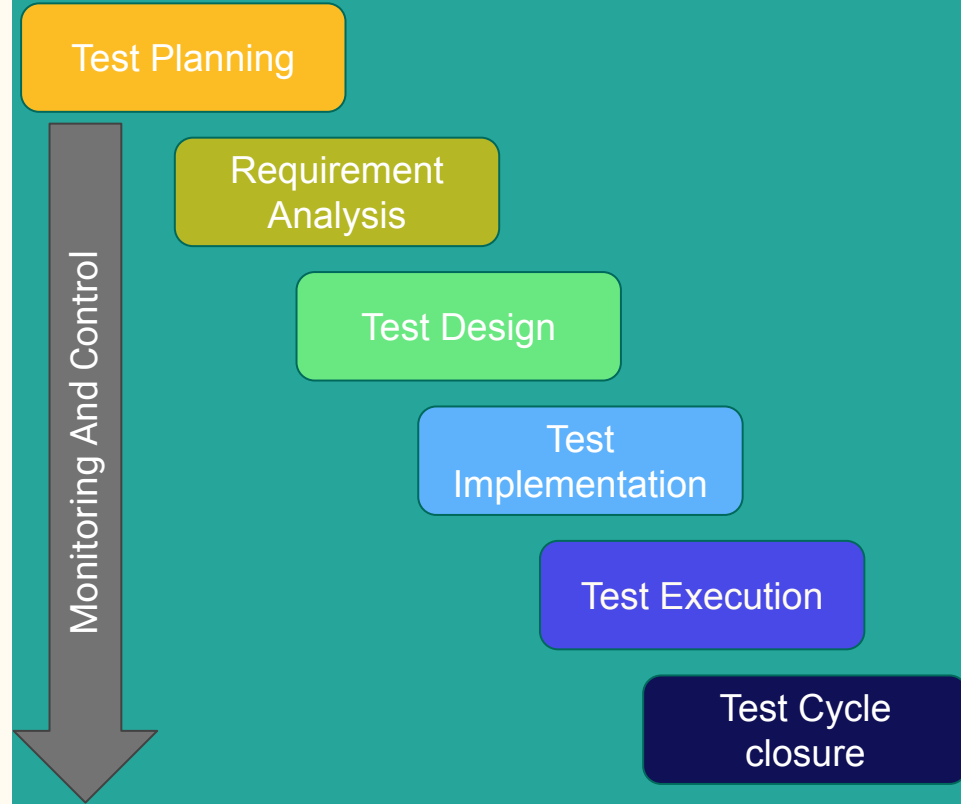
- Build confidence in the level of the quality of the SW.
- Find defects and failures thus reduce the level of risk in tested SW
- Provide sufficient information to stakeholders to allow them to make information decisions especially regarding the level of quality of the test objectives
- To comply with contractual, legal, or regulator requirements. Or standards, and /or to verify the test objectives compliance with such requirements or standards

Testing Life Cycle

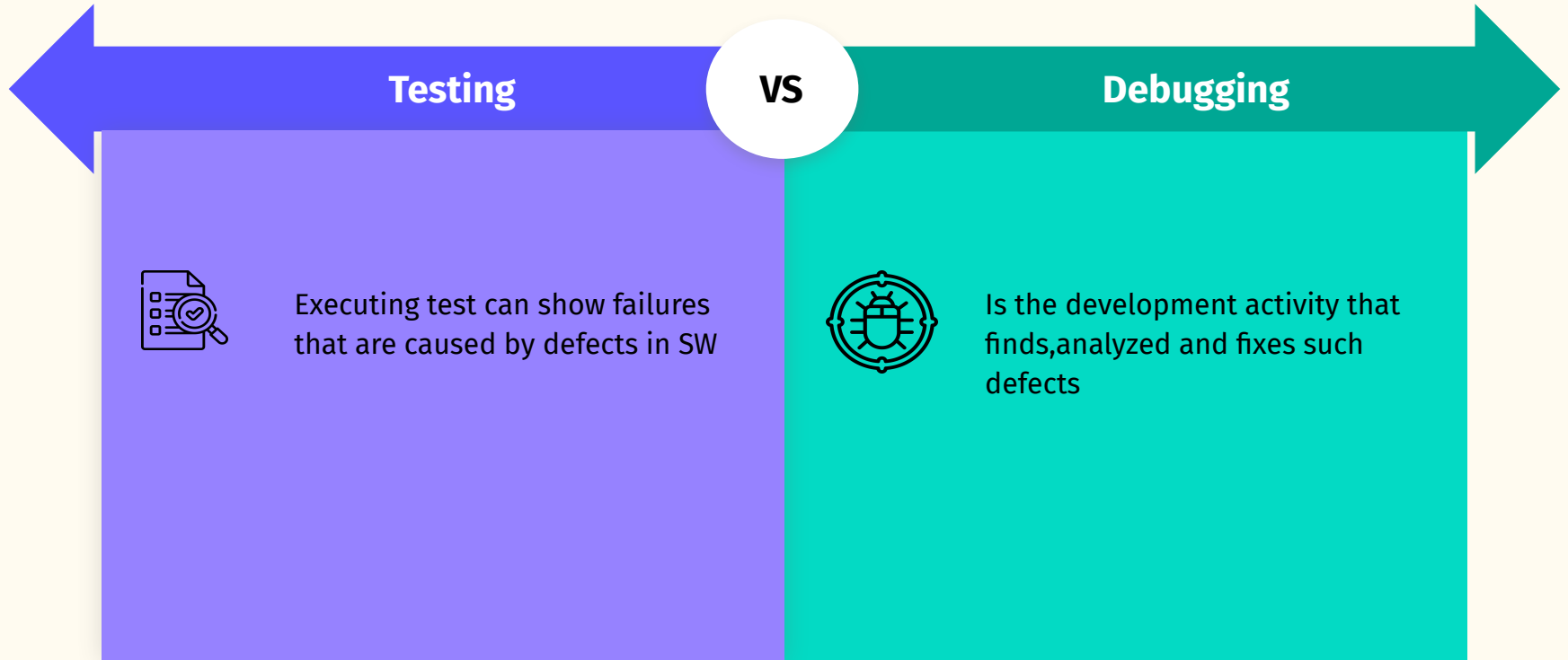
Software Testing Life Cycle (STLC):

Testing life Cycle

is a sequence of specific activities conducted during the testing process to ensure software quality goals are met.



Testing vs Debugging

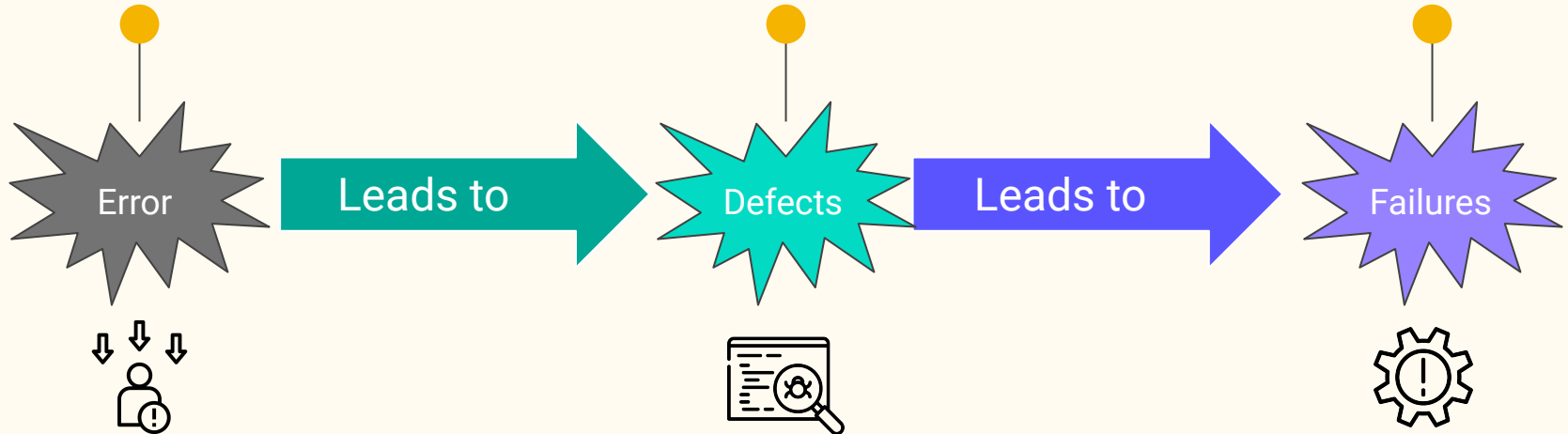


Failures life cycle

A person can make an mistake

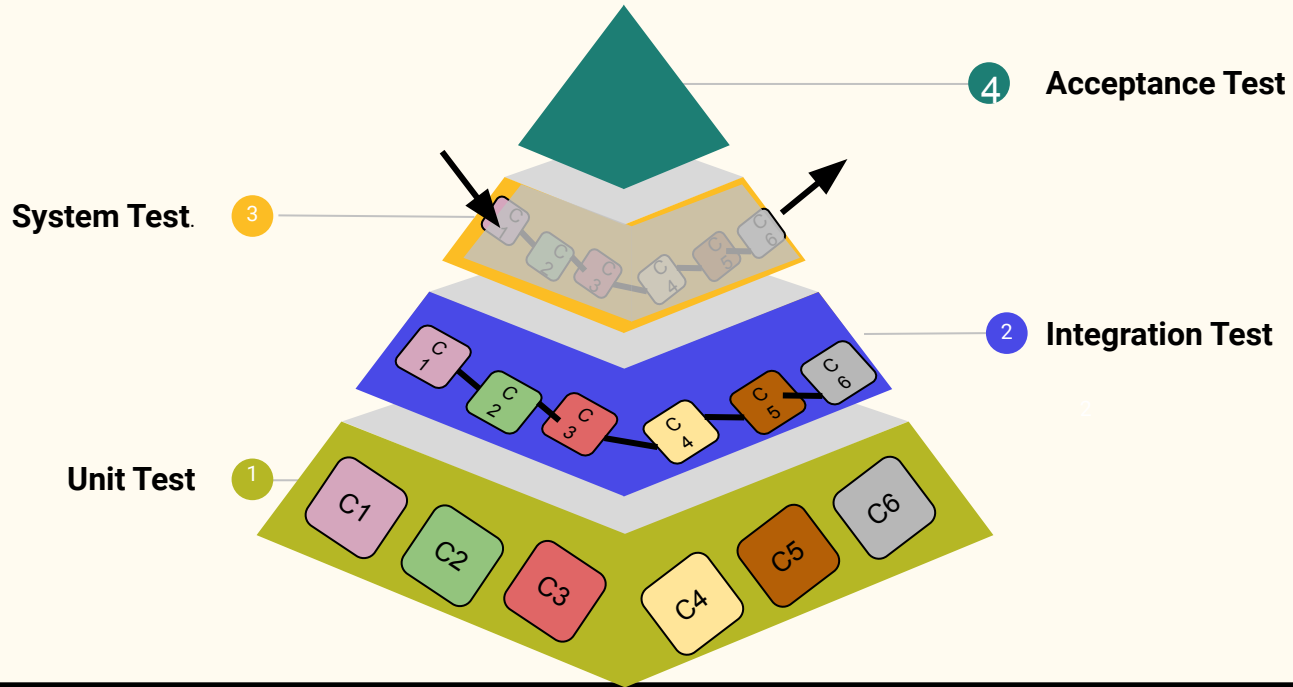
**Fault/bug in SW code or
in some other related
works products**

When code executing



Test Levels

Test Levels



Types of Test

Types of Test

Functional Test

Testing performed to evaluate if a component or system satisfies functional requirements.

White-box

Testing based on an analysis of the internal structure of the component or system.



Non Functional Test

Testing the attributes of a component or system that do not relate to functionality, e.g., reliability, efficiency, usability, maintainability and portability.

Change-Related

When the system undergoes changes due to defect fixes, two types of tests are needed; one to verify the change has not broken the existing, working code and the second to ensure the defect has been fixed. These tests are called Regression and Confirmation testing.

Unit Test Features

What is SW Unit ?

A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc.

What is the Unit Test ?

Is a level of software testing where individual units / components of a software are tested. The purpose is to validate that each unit of the software performs as designed.

What is the Unit Test Methods ?

Unit Testing is usually performed by using the White Box Testing method and is normally automated.

When is it performed?

Unit Testing is the first level of software testing and is performed prior to Integration Testing. Though unit testing is normally performed after coding, sometimes, specially in test-driven development (TDD), automated unit tests are written prior to coding.

Who performs it?

It is normally performed by software developers themselves or their peers. In rare cases, it may also be performed by independent software testers but they will need to have access to the code and have an understanding of the architecture and design.

Unit Testing - Advantages:

- Reduces Defects in the Newly developed features or reduces bugs when changing the existing functionality.
- Reduces Cost of Testing as defects are captured in very early phase.
- Improves design and allows better refactoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well.

Test Utility Code

Unit Testing Techniques:

- Software testing techniques are the ways employed to test the application under test against the functional or non-functional requirements gathered from business.
- Each testing technique helps to find a specific type of defect. For example, Techniques which may find structural defects might not be able to find the defects against the end-to-end business flow.

Unit Testing Techniques:

Structure Based Techniques

White box techniques :are focused on how the code structure works and test accordingly.

- Statement coverage
- Decision coverage
- Conditional/Multiple condition coverage:

Specification Based Techniques

It basically means creating and executing tests based on functional or non-functional specifications from the business.

- Equivalence partitioning
- Boundary Value Analysis (BVA)
- Use case-based Testing:

Structure Based Techniques

Statement Coverage

$$\frac{\text{Number of Statements of code exercised}}{\text{Total number of statements}}$$

If a code segment has 10 lines and the test designed by you covers only 5 of them then we can say that statement coverage given by the test is 50%.

Decision Coverage

$$\frac{\text{Number of decision outcomes exercised}}{\text{Total number of Decisions}}$$

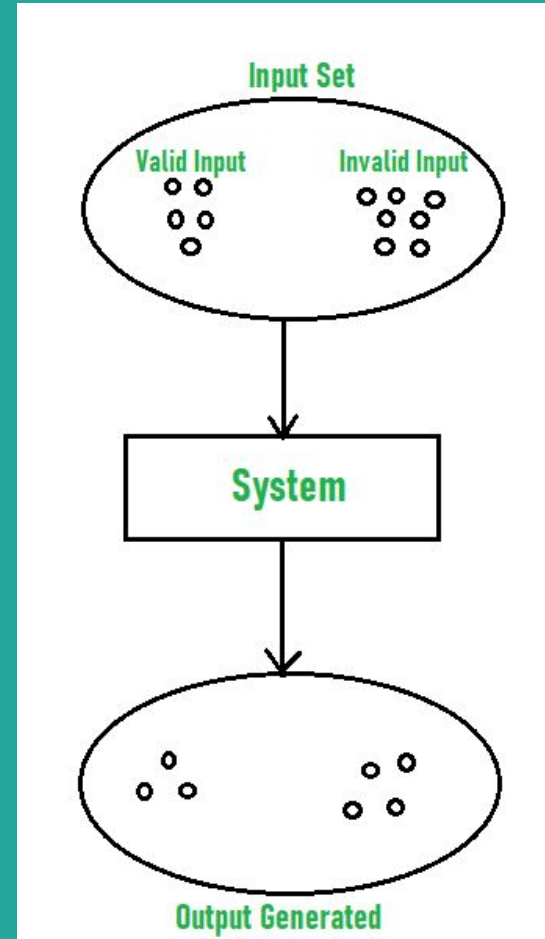
For Example, If a code segment has 4 decisions (If conditions) and your test executes just 1, then decision coverage is 25%

Conditional/Multiple condition coverage:

It has the aim to identify that each outcome of every logical condition in a program has been exercised.

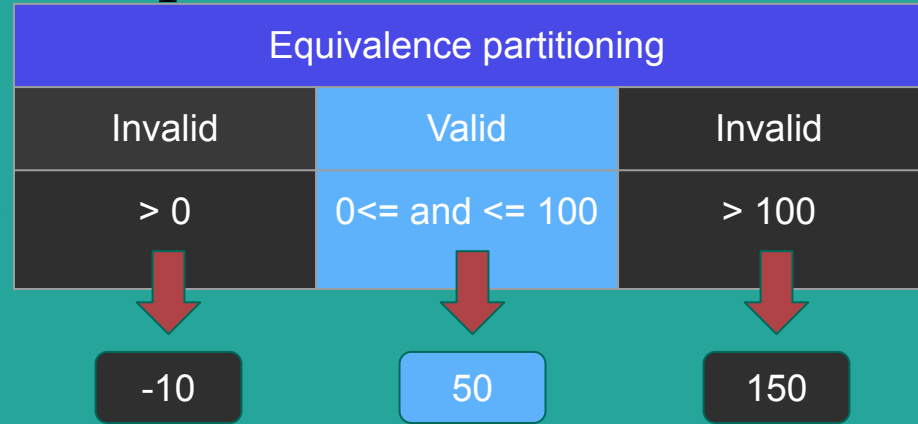
Equivalence partitioning

- Partition the input range of data into valid and non-valid sections (equivalent)
- Select to test with any value in a given partition assuming that all values in the partition will behave the same.



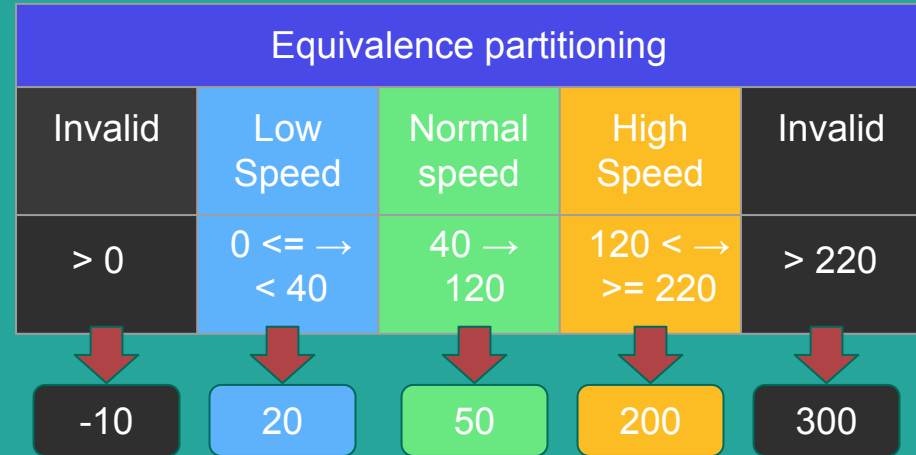
Equivalence partitioning examples

Students Scores in an exams let assume Valid Scores between 0 and 100 and other than that is invalid



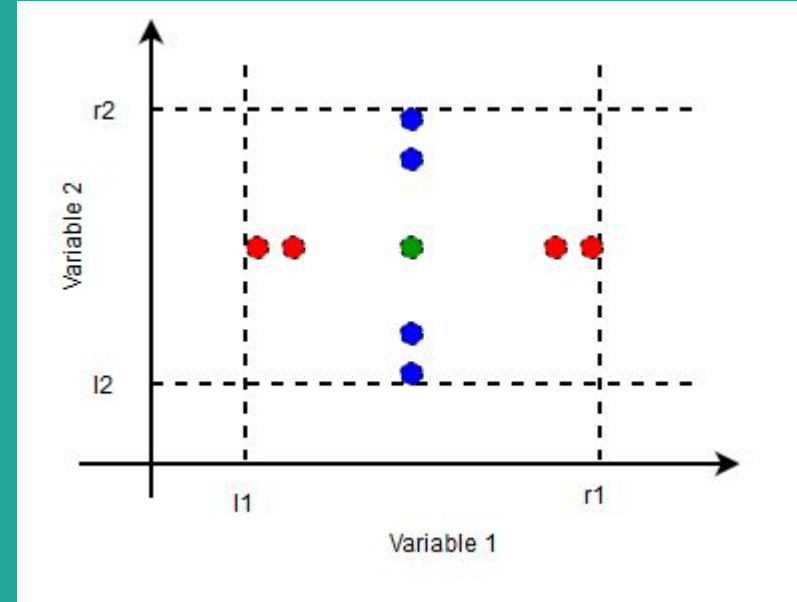
Car speed identification system :

- Valid range for valid car speed is from 0 to 220km/h .
- low speed if car speed is below 40km/h
- high speed is above 120
- otherwise will be normal speed



Boundary Value Analysis (BVA)

- Test cases are designed using boundary values.
- BVA is based on the single fault assumption, which states that the bugs are most commonly concentrate at the boundary.



Boundary Value Analysis examples

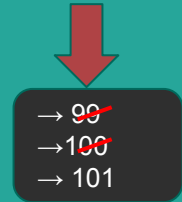
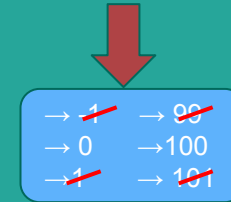
Students scores in an exams let assume Valid scores between 0 and 100 and other than that is invalid

Car speed identification system :

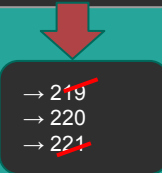
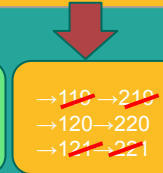
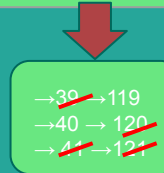
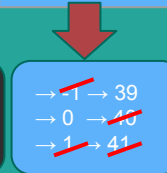
- Valid range for valid car speed is from 0 to 220km/h .
- low speed if car speed is below 40km/h
- high speed is above 120
- otherwise will be normal speed



BVA		
Invalid	Valid	Invalid
> 0	$0 \leq \text{ and } \leq 100$	> 100



BVA				
Invalid	Low Speed	Normal speed	High Speed	Invalid
> 0	$0 \leq \rightarrow < 40$	$40 \rightarrow 120$	$120 < \rightarrow \geq 220$	> 220



Use case-based Testing

- Identify test cases that execute the system as a whole- like an actual user (Actor), transaction by transaction

Use Case testing Examples

Use cases are a sequence of steps that describe the interaction between the Actor and the system. They are always defined in the language of the Actor, not the system. This testing is most effective in identifying the integration defects. Use case also defines any preconditions and postconditions of the process flow. ATM machine example can be tested via use case:



Happy Flow A: Actor S: System	1	A: Insert Card
	2	A: Enter PIN
	3	S: Validte PIN
	4	S: Allow to withdraw

Test Techniques

Exercises

Notes

In each exercise please identify the followings:

1. Used Techniques
2. Reasons
3. Test cases

Ex 1

Write test ideas for this Scenario: You are at the grocery store checkout counter. You have bought five items (x, y, z, a, and b). You make payment and move to the EXIT door.

Example Test ideas as a hint:

If the checkout counter is humanless, scan all the five items, scan your card and make payment.

The scanners should scan proper relevant information.

Ex2 :

The text box accepts numeric values in the range of 18 to 25 (18 and 25 are also part of the class). So this class becomes our valid class.

Ex 3:

One of the fields on a form contains a text box that accepts alphanumeric values.
Identify the Valid Equivalence class.

- a) BOOK
- b) Book
- c) Boo01k
- d) Book

Ex 4:

The Switch is switched off once the temperature falls below 18 and then it is turned on when the temperature is more than 21.

Ex5

A program validates numeric fields as follows: values less than 10 are rejected, values between 10 and 21 are accepted, values greater than or equal to 22 are rejected.

Structure Inspection and Test-Specific identity

Introduction To BDD

Behavior-Driven Development (BDD)

- A collaborative approach to development in which the team is focusing on delivering expected behavior of a component or system for the customer, which forms the basis for testing.

ISTQB :Advanced Agile Technical Tester - 2019

- BDD is designed to test an application's behavior from the end user's standpoint

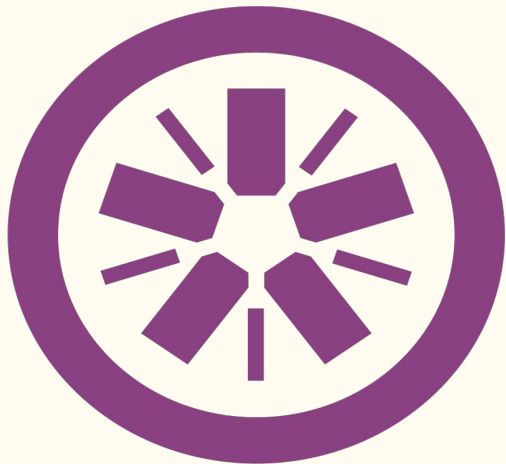
Cucumber

How is testing using BDD ?

Behavior-driven development involves a **developer**, **test engineer** and a **product manager** (and potentially other stakeholders). The group meets to come up with concrete examples of **acceptance criteria** in a user story. These examples are described using a domain-specific language,

Examples for BDD frameworks for testing JavaScript Code

Jasmine



Mocha



Writing Unit Tests using Mocha

Writing Unit Tests using Jasmine

- Jasmine is a behavior-driven JavaScript unit testing framework
- It aims to run on any JavaScript-enabled platform, to not intrude on the application nor the IDE, and to have easy-to-read syntax.



First Test

```
describe("Math module Suite", function(){  
    it("should return the exponent", function(){  
        expect(Math.pow(2,3)).toBe(8)  
    });  
});
```

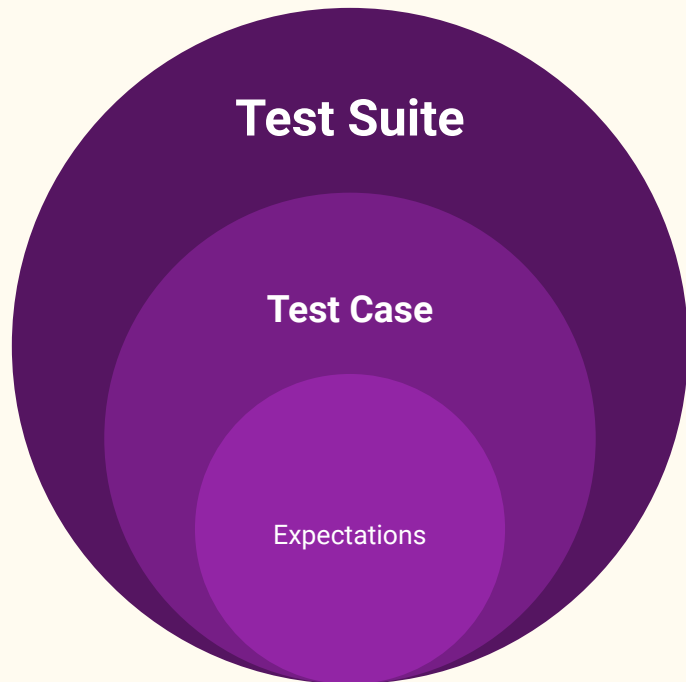
1 spec, 0 failures

finished in 0.006s

Math module suite

- should return the exponent

Test structure in Jasmine



Test Suite

A **test suite** is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

`describe(suite_title, fn)`

```
describe("Math module Suite", function(){
  it("should return the exponent", function(){
    expect(Math.pow(2,3)).toBe(8)
  });
  describe("Constants Suite", function(){
    // Test Cases
  });
});
```

Test Case

A **test case** is the individual unit of testing. It checks for a specific response to a particular set of inputs.

`it(spec_title, fn)`

```
describe("Math module Suite", function(){  
  it("should return the exponent", function(){  
    expect(Math.pow(2,3)).toBe(8)  
    expect(Math.pow(2,4)).toBe(16)  
  });  
});
```

Expectations

Expectations are built with the function `expect` which takes a value, called the actual. It is chained with a `Matcher` function, which takes the expected value.

```
expect( actual_value ).matcher_fn( expected_value )
```

```
describe("Math module Suite", function(){
  it("should return the exponent", function(){
    expect(Math.pow(2,3)).toBe(8)
  });
});
```

Matchers

Matchers: Matching

```
describe("Math module Suite", function(){
  it("should return the exponent", function(){
    Var output = Math.pow(2,3)
    expect(output).toBe(8)           // === Matching
    expect(Math.pow(2,4)).toEqual(16) // == Matching
  });
});
```

Matchers:Undefined & Null

```
var x, y=2, z=null
describe("Math module Suite", function(){
  it("should return the exponent", function(){
    expect(y).toBeDefined()      // === Matching against defined
    expect(x).not.toBeDefined()  // === Matching against undefined
    expect(x).toBeUndefined()    // === Matching against undefined
    expect(z).toBeNull()         // === Matching against null
  });
});
```


Matchers:Regular Expressions

```
describe("Regular Expressions Suite", function(){  
  it("Should return if string matched with string", function(){  
    expect("01007882343").toMatch(/[0-9]{11}/)    // Matching regex  
  });  
});
```

Matchers: True & False

```
describe("Checking Regular expressions Suite", function(){
  it("Should return True or false", function(){
    expect("Ahmed").toBeTruthy      // Matching Truthy Values
    expect(0).toBeFalsy            // Matching Falsy Values
    expect(2 === "2").toBeTruthy    // Matching Falsy Values
  });
});
```

Matchers:In collections

```
describe("Checking Regular expressions Suite", function(){
    it("Should return True or false", function(){
        expect("Ahmed").toContain ("h")
        expect([1,"ahmed",True].toContain ("True")
    });
});
```

Skip

Skip

We can skip any test case or test suite by prepend it definition by x

```
xdescribe(suite_title, fn)
```

```
xit(spec_title, fn)
```



Setup & Teardown

Setup & Teardown



To help a test suite DRY up any duplicated setup and teardown code

- `beforeEach(fn)` # It runs before every spec
- `afterEach(fn)` # It runs after every spec
- `beforeAll(fn)` # It runs at the beginning of test
- `afterAll(fn)` # It runs at the end of test

Jasmine vs Mocha

Features	Jasmine 	Mocha 
APIs: <ul style="list-style-type: none">• Test suite• Test case• Expectations	<ul style="list-style-type: none">→ <code>describe</code> blocks→ <code>Spec.it</code> function→ Build-in assertion library	<ul style="list-style-type: none">→ <code>describe</code> blocks→ <code>Spec.it</code> function→ No build-in assertion library, instead use external library like : Chai, should.js, expect.js, and better-assert
Doubles	Test doubles come in the form of spies	Mocha does not come with a test double library. Instead, you will need to load in Sinon into your test harness.

Jasmine vs Mocha

Features	Jasmine 	Mocha 
Asynchronous Testing	Asynchronous testing can be a bit of a headache	Asynchronous testing is very simple
Fake Server	Jasmine does not is a fake server	Sinon supports features
Running Tests	Jasmine does not have a command line utility to run tests. But Karma could use instead	Mocha comes with a command line utility that you can use to run tests <code>mocha tests --recursive --watch</code>

Jasmine vs Mocha



Pros

- Simple setup for node through jasmine-node
- Headless running out of the box
- Nice fluent syntax for assertions built-in
- Supported by many CI servers
- Descriptive syntax for BDD paradigm(which is a big plus for us)



Pros

- Clear, Simple API
- Headless running out of the box
- Allows use of any assertion library that will throw exceptions on failure, such as Chai
- Supported by some CI servers
- Has aliases for functions to be more BDD-oriented or TDD-oriented
- Highly extensible
- Asynchronous testing is very simple



Jasmine vs Mocha



Cons

- Asynchronous testing can be a bit of a headache
- Expects a specific suffix to all test files (*spec.js by default)



Cons

- Tests cannot run in random order.
- Allows use of any assertion library(Both pro and con)
- No auto mocking or snapshot testing

Jasmine Exercises

Create Tests using Jasmine framework

Using the test techniques \Rightarrow write test cases to test the following function with respect to the following rules :

- Mentioned used techniques
- Create 2 Test suites
- In each test suite create at least 2 test cases
- Use the following methods
 - before All
 - afterAll

Ex #1 :Test function to evaluate Car speed

Req # 1 : Input to the function is the car speed and output shall be speed level

Req# 2 : Function shall calculate output to be :

- If $\text{Speed} < 0$ Level shall be Invalid
- If $0 \leq \text{Speed} < 40$ Level shall be Low
- If $40 \leq \text{Speed} < 120$ Level shall be Normal
- If $120 \leq \text{Speed} < 200$ Level shall be High
- If $200 \leq \text{Speed} < 220$ Level shall be V.High
- If $220 < \text{Speed}$ Level shall be Invalid

Ex #2 :Test function to evaluate students Speeds

Req # 1 : Input to the function is student Speed and output student Level

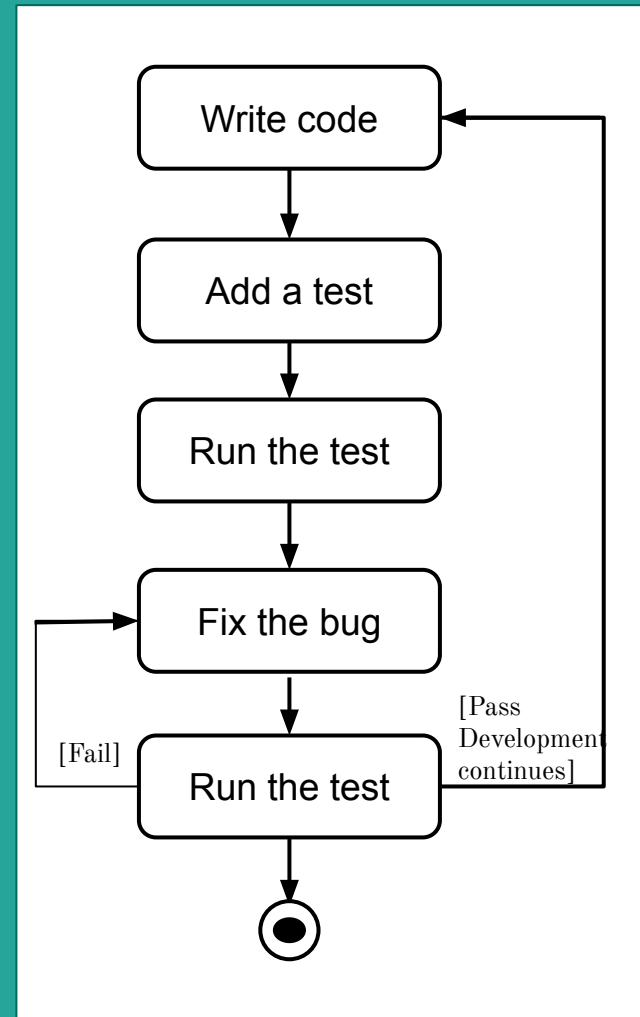
Req# 2 : Function shall calculate output to be :

- If $\text{Speed} < 0$ Level shall be Invalid
- If $0 \leq \text{Speed} < 50$ Level shall be Failed
- If $50 \leq \text{Speed} < 65$ Level shall be Passed
- If $65 \leq \text{Speed} < 75$ Level shall be Good
- If $75 \leq \text{Speed} < 85$ Level shall be V.Good
- If $85 \leq \text{Speed} < 100$ Level shall be Excellent
- If $100 \leq \text{Speed}$ Level shall be Invalid

Introduction To TDD

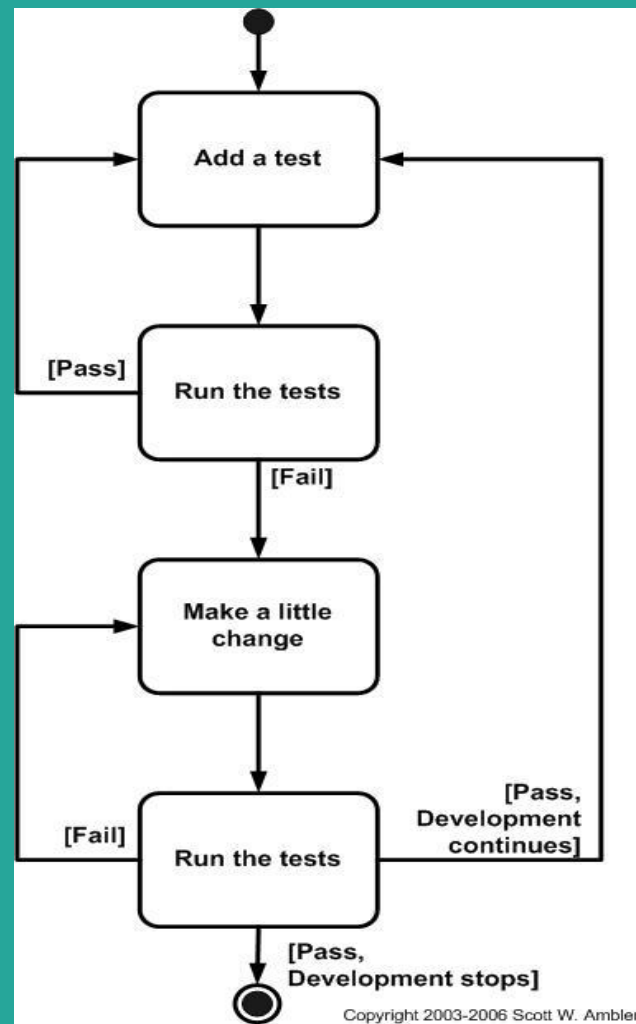
Introduction To TDD

Traditionally, the software development workflow is mostly a loop of the following steps:



Introduction To TDD

Test-driven development changes this workflow by writing automated tests, and by writing tests before we write the code



BDD vs TDD

Features	BDD	TDD
What You're Testing ?	<ul style="list-style-type: none">• BDD is designed to test an application's behavior from the end user's standpoint• The BDD test is only concerned about the result of the higher level scenario.	<ul style="list-style-type: none">• TDD is focused on testing smaller pieces of functionality in isolation• The TDD test asserts the result of a specific method
How You're Testing ?	BDD involves product managers, developers, and test engineers who collaborate to come up with concrete examples of desirable functionality	TDD can be done by a solo developer without any external input from product managers or stakeholders.

BDD vs TDD

BDD	TDD
<p>Behavior-driven development represents an evolution beyond TDD, where business goals can be better communicated to developers.</p>	<p>Test-driven development has become the default approach for Agile software development over the past several years.</p>
<p>By bridging the gap between business and technical teams, BDD helps reduce any confusion about acceptance criteria, identify potential problems with user stories early, and ensure that the application functions as-expected for end users.</p>	<p>The approach minimizes bugs reaching production and ensures that software can be continuously released without issue.</p>

TDD Exercises

TDD Exercises

Using TDD approach develop the following functions :

TDD EX