**ADVANCED SOFTWARE ENGINEERING**

**PROBLEM SET-4**

NAME: **SWETCHA REDDY VANGALA**

CLASS ID**: 41**

1. ***DESIGN OF UNIT TESTING***

**A. Unit Testing Principles and Guidelines**

* **Don’t comment tests**

A unit test itself is a verification of behavior of small part of application, and so we need not comment those tests. Comments make tests complex and slow.

* **Avoid logic in your tests**

Using simple statements in tests is a better option. There shouldn’t be loops and conditions.

* **Don't write unnecessary expectations**

Unit testing is designing how something should work, but not everything the code has to do.

### Properly setup the actions that apply to all the tests involved

### Set all the actions that are applicable to all the tests, instead of setting them individually.

### Consider using factory functions in the tests

### Setup code can be reduced, especially when dependency injection is used. Test can be made more readable. Factory functions provide flexibility when creating new instances.

* **Know your testing framework API**

### Knowledge on API prior to testing helps reducing the complexity of the test code.

* **Don't test multiple concerns in the same test**

Every method should be tested separately, so that even if there is any error, it can be located easily.

* **Cover the general case and the edge cases**

General cases must be tested and the edges cases too, because there are chan ces of strange behavior at the edges.

* **When applying TDD, always start by writing the simplest failing test**

When the testing is started with failure case, defects can be found and functionalities can be improved step by step.

* **When applying TDD, always make small steps in each test-first cycle**

Build the tests from simple to complex. Testing and delivering software should be done incrementally in multiple iterations.

* **Test the behaviour, not the internal implementation**

We must test the behaviour of the test case, not how it is actually implemented. But, when we encounter an error, we must know which internal part of the code has an error, so that it can be corrected.

* **Don't mock everything**

Once we start mocking everything, complexity increases and one error needs a modification in some other part and it might be complex. Only when it leads to a easy, simple tests, when speed is within the limit, it is not lowered, when there comes no shared state between tests, no side effects.

* **Create new tests for every defect**

Any test created should test a bug found, not affecting any other.

* **Don't write unit tests for complex user interactions**

Complex user interactions involve many units of work. These can be functionally tested instead of being unit tested. These take more time to execute.

* **Test simple user actions**

Simple user actions should be tested. These involve small code and effort.

Eg:

Form validation to register to an application.

* **Review test code first**

It is important to review the code first before testing, because test cases are smaller parts of code. We can understand the developer’s view if we review the code first.

* **Practice code katas, learn with pair programming**

Practice makes a man perfect, whatever might be the case. By applying theory again and again and getting a review after every application, so that better programming can be done.

**B.**

**Test case**

it(‘should add a user in memory’,()=>{

userManager.addUser(‘Dr. Falker’, ‘Joshua’);

expect(userManager.\_users[0].name).toBe(‘Dr. Falker’);

expect(userManager.\_users[0].password).toBe(‘Joshua’);

});

1. ***Explanation***

A user’s credentials must be added to memory.

**EXPECTED RESULT**

Name: Dr. Falker

Password: Joshua

1. The issue with this test case is that just the behaviour must be tested, not the internal implementation. How the memory actually stores the username and password is not necessary, what is necessary is just that the credentials should result in a successful login.

**BETTER APPROACH**

it(‘should add a user in memory’, ( ) => {

userManager.addUser(‘Dr. Falker’, ‘Joshua’);

expect(userManager.loginUser(‘Dr. Falker’, ‘Joshua’)).toBe(true);

});

**C.**

**Feature or issue tested**

When the song has been paused, player should indicate the song that

was paused and should be able to resume.

**Output**

The song was paused and player indicated the song that was paused, but it was not able to resume. So, the output is FAIL.

**Expected output**

Song paused, indication of song paused, and player should be able to resume. Output is PASS.

**Unit tests**

describe(“player”, function( ) {

it(“pauses song” , function ( ) {

expect(pause song).toEqual(paused song)

}

it(“displays song” , function ( ) {

expect(display(paused song)).toEqual(paused song)

}

it(“resumes song” , function ( ) {

expect(play(paused song)).toEqual(play paused song)

}

**D.**

**Javascript testing frameworks**

***QUnit***

QUnit was developed to test jQuery and jQuery UI, no longer depend on jQuery. There was an error in earlier stages of development of QUnit, a function with an asynchronous call back called by a test case forgets to invoke the call back and QUnit is terminated with exit code 0. In most cases, it is interpreted as PASS, which interrupts the testing.

***Mocha JS***

Mocha JS could cover various testing needs in its earliest days. There were some issues later, but now all those are resolved and allows to choose test result reporter. It can handle accidental double callback.

***Tape***

To plan using t.plan() and end each test case using t.end() was not very complicated. Although if a single test case fails entire test suite is aborted, yet fixing that issue was easy using require(‘tape-catch’). Summary of Tape has the number of assert passes or fails which makes it technically correct.

***AVA***

Parallel testing can be done. Highlights the difference between actual and expected results.

***Jasmine***

One of the oldest frameworks is Jasmine. There are built in spies. PASS is reported after running all test cases.

***Karma***

Karma is actually a test runner which can take test suite from Mocha, QUnit, or Jasmine. The test suite is then run on multiple browsers which report back results that are presented as terminal output. Angular JS uses Karma. Whenever we want to run the test cases in browser, we must use Karma.

***Jest***

React project uses Jest framework. Initially, Jest was similar to Jasmine, but now there is almost nothing that is similar to Jasmine in Jest. Jest cannot run tests inside a browser. Regressions in output can be detected using Snapshot testing which is one of the striking features of Jest.

**E. Performance Testing**

* Performance Testing is a testing practice performed to determine how a system’s response and stability is for certain workload. This helps to improve performance standards. It is a practice to ensure applications work as expected under expected workload. Focus of Performance Testing is on program’s speed, scalability and stability.
* ***LOAD TESTING-*** Simplest form of Performance testing. Workload is the expected users performing specific transactions with a specific duration. Database and application server also monitored.

***STRESS TESTNING-*** This is the process of determining the ability of a computer or a network or a program or a device to maintain effectiveness under unfavorable conditions like in the favorable conditions. To conduct stress testing, adverse conditions are deliberately created.

* ***PERFORMANCE TESTING OF MOBILE APPLICATION***

If a mobile application is big and complex, set some thresholds to which define maximum operation completion time. Use the following tools

* Calabash
* Appium

To check the performance of the backend i.e, when many users are

using the application at the same moment. Collect the backend metrics and key performance indicators and analyze them. This can be done using **Apache JMeter.**

***Apache JMeter-*** This is an open source software, a pure Java application designed to test functional behavior and measure performance.

* ***RESPONSE TIME IN PERFORMANCE TESTING***

Response time is the time taken for a request to return back from the application. It represents how long a user must wait for a request to be processed. It should be as small as possible, because users like fast applications, rather than slow applications. It should be around 1s-3s.

* ***RESPONSE TIME-*** Response time is the time taken for a request to return back from the application. It represents how long a user must wait for a request to be processed.

***THROUGHPUT-*** Number of transactions that an application can handle per second is the throughput. Throughput is high means that an application is efficient and performance of it is as expected.

**References**

[**https://github.com/mawrkus/js-unit-testing-guide**](https://github.com/mawrkus/js-unit-testing-guide)

[**https://code.tutsplus.com/tutorials/testing-your-javascript-with-jasmine--net-21229**](https://code.tutsplus.com/tutorials/testing-your-javascript-with-jasmine--net-21229)

[**http://mo.github.io/2017/06/05/javascript-unit-testing.html**](http://mo.github.io/2017/06/05/javascript-unit-testing.html)

[**http://jmeter.apache.org/**](http://jmeter.apache.org/)

[**https://sqa.stackexchange.com**](https://sqa.stackexchange.com)

[**https://www.joecolantonio.com**](https://www.joecolantonio.com)

[**https://en.wikipedia.org**](https://en.wikipedia.org)