JavaScript
Test-Driven Development (TDD)

https://www.thruskills.com

+91 831 737 5392

What you will learn

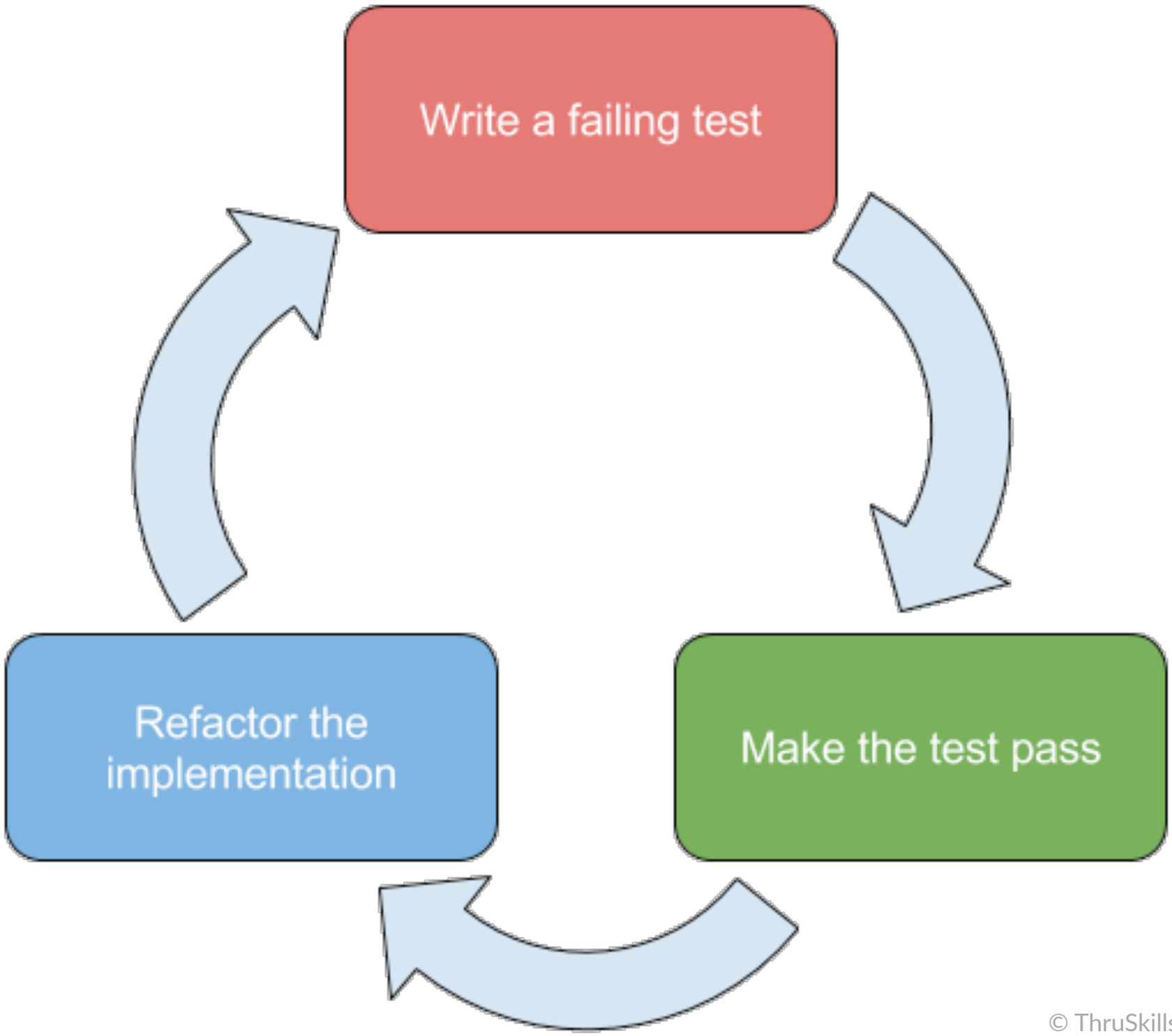
- [x] Introduction to TDD
- [x] Types Of Automated Test

What is TDD?

Test-driven development is a programming methodology with which one can tackle the design, implementation, and testing of units of code, and to some extent the expected functionality of a program.

Complementing the test-first approach of Extreme Programming, in which developers write tests before implementing a feature or a unit, TDD also facilitates the refactoring of code; this is commonly referred to as the Red-Green-Refactor Cycle.

What is TDD?



What is TDD?

1. Write a failing test

Write a test that invokes your logic and assert that the correct behavior is produced

- In a unit test, this would be asserting the return value of a function or verifying that a mocked dependency was called as expected

- In a functional test, this would be ensuring that a UI or an API behaves predictably across a number of actions

What is TDD?

2. Make the test pass

Implement the minimum amount of code that results in the test passing, and ensure that all other tests continue to pass

3. Refactor the implementation

Update or rewrite the implementation, without breaking any public contracts, to improve its quality without breaking the new and existing tests

Types of automated tests

Unit tests – ensure that individual units of the app, such as functions and classes, work as expected. Assertions test that said units return the expected output for any given inputs

Integration tests – ensure that unit collaborations work as expected. Assertions may test an API, UI, or interactions that may result in side-effects (such as database I/O, logging, etc…)

Types of automated tests

End-to-end tests – ensure that software works as expected from the user's perspective and that every unit behaves correctly in the overall scope of the system. Assertions primarily test the user interface

Benefits of Test-Driven Development

1. **Immediate test coverage**

By writing test cases for a feature before its implementation, code coverage is immediately guaranteed, plus behavioral bugs can be caught earlier in the development lifecycle of a project. This, of course, necessitates tests that cover all behaviors, including error handling, but one should always practice TDD with this mindset.

Benefits of Test-Driven Development

2. **Refactor with confidence**

Referring to the red-green-refactor cycle above, any changes to an implementation can be verified by ensuring that the existing tests continue to pass. Writing tests that run as quickly as possible will shorten this feedback loop; while it's important to cover all possible scenarios, and execution time can vary slightly between different computers, authoring lean and well-focused tests will save time in the long term.

Benefits of Test-Driven Development

3. **Design by contract**

Test-driven development allows developers to consider how an API will be consumed, and how easy it is to use, without having to worry about the implementation. Invoking a unit in a test case essentially mirrors a call site in production, so the external design can be modified before the implementation stage.

Benefits of Test-Driven Development

4. **Avoid superfluous code**

As long as one is frequently, or even automatically, running tests upon changing the associated implementation, satisfying existing tests reduces the likelihood of unnecessary additional code, arguably resulting in a codebase that's easier to maintain and understand. Consequently, TDD helps one to follow the KISS (Keep it simple, stupid!) principle.

Benefits of Test-Driven Development

5. **No dependence upon integration**

When writing unit tests, if one is conforming to the required inputs, then units will behave as expected once integrated into the codebase. However, integration tests should also be written to ensure that the new code's call site is being invoked correctly.

TDD Example

For example, let's consider the function below, which determines if a user is an admin:

```
'use strict'

function isUserAdmin(id, users) {
    const user = users.find(u => u.id === id);
    return user.isAdmin;
}
```

Rather than hard code the users data, we expect it as a parameter. This allows us to pass a pre populated array in our test:

TDD Example

```
const testUsers = [
    {
        id: 1,
        isAdmin: true
    },

    {
        id: 2,
        isAdmin: false
    }
];

const isAdmin = isUserAdmin(1, testUsers);
// TODO: assert isAdmin is true
```

This approach allows the unit to be implemented and tested in isolation from the rest of the system.

TDD With JavaScript

With the advent of full-stack software written in JavaScript, a plethora of testing libraries has emerged that allow for the testing of both client-side and server-side code; an example of such a library is Mocha.

A good use case for TDD, in my opinion, is form validation; it is a somewhat complex task that typically follows these steps:

TDD With JavaScript

- Read the value from an <input> that should be validated

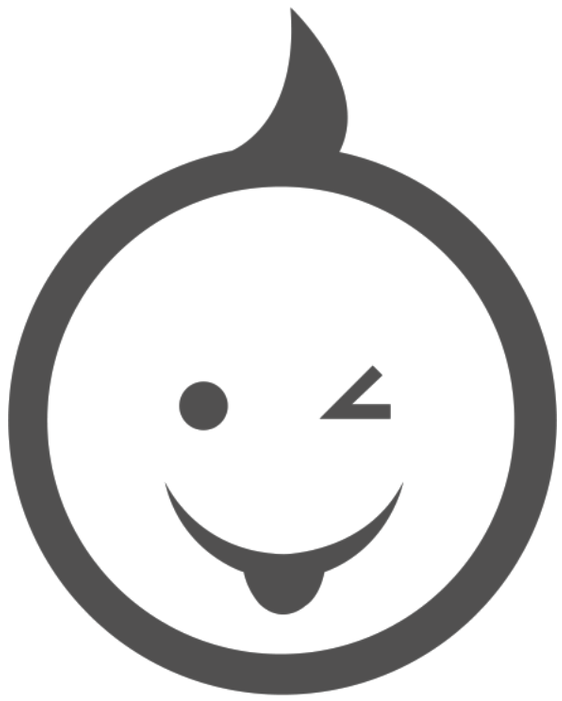
- Invoke a rule (e.g. alphabetical, numeric) against said value

- If it is invalid, provide a meaningful error to the user

- Repeat for the next validatable input

Lets get it done…

Any Questions



Thank You

https://www.thruskills.com

+91 831 737 5392

thruskills