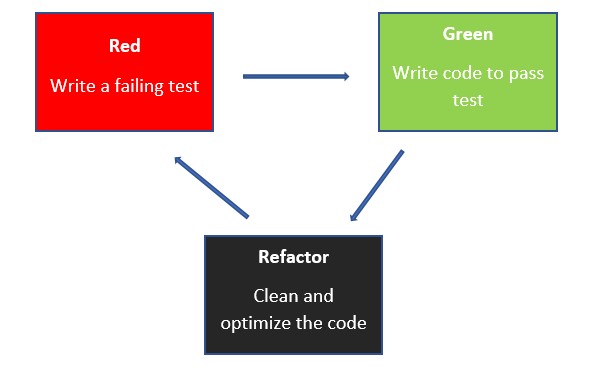
Test-Driven Development

Test-Driven Development, or TDD for short, is a method used to write tests before we start our implementation. Before you start, you might list the requirements that need to be fulfilled in your application.

We use RED – GREEN – REFRACTOR Cycle in TDD

Take the first requirement and write a failing test. The test fails and it is RED as you haven’t developed it yet.

But the test describes already what your code should do to fulfill the requirement. Now you need to make the test GREEN by writing the necessary code to make the test pass. After you write the code to make the test green you need to Refactor the code. Maybe you can simplify the code or extract few code lines into a method to make code more readable and maintainable. After you are done with the first requirement, you can continue with the next requirement. This means you iterate the entire cycle (Red -> Green -> Refactor) for another requirement and so on.



**Advantages**



1. TDD makes you think with the needed API from the beginning. You need to think about what classes, properties, API’s are needed. This will usually lead to great API design.
2. After you know the class and properties, another big advantage is that you need to think about what the code should do than how it should do. As you start with test you don’t need to have any idea about implementation. You just need to write a test for what the code should do. After writing the test you can think of requirements and their development.
3. While thinking of your requirements, you get quick feedback about your requirements by running the test. The fact that you get quick feedback means you even don’t need a fully working application at all. You just need a class library to build your business logic and don’t need the entire project.
4. This helps you create modular code. You can decouple the dependencies from the beginning and TDD makes you do that from the beginning. This decoupling of dependency makes you write a modular code by isolating the dependencies like a database that is not ready yet and the web API isn't ready when beginning the development.
5. This leads to a maintainable codebase, as you will have one test per requirement. You can write code to add new functionality and run all the unit tests to ensure that the existing code doesn’t break. You can be confident about your new code as well as the existing code.
6. These tests will serve as good documentation. For example, the test for the code written by others will help you understand why the code has been written.

**Disadvantage**

The only disadvantage is that TDD is not so easy to start by writing tests for beginners. In fact, TDD is an art that every developer should master.

**BDD – Business driven development**

BDD is a way for software teams to work that closes the gap between business people and technical people by:

* Encouraging collaboration across roles to build shared understanding of the problem to be solved
* Working in rapid, small iterations to increase feedback and the flow of value
* Producing system documentation that is automatically checked against the system’s behaviour

We do this by focusing collaborative work around concrete, real-world examples that illustrate how we want the system to behave. We use those examples to guide us from concept through to implementation, in a process of continuous collaboration.

**BDD and agile**

We assume that your team are using some kind of agile methodology already, planning work in small increments of value like [User Stories](https://cucumber.io/docs/terms/user-story/). BDD does not replace your existing agile process, it enhances it.

Think of BDD as a set of plugins for your existing process that will make your team more able to deliver on the promises of agile: timely, reliable releases of working software that meets your organisation’s evolving needs, requiring some maintenance effort and discipline.

Essentially, day-to-day BDD activity is a three-step, iterative process:

1. First, take a small upcoming change to the system – a [User Story](https://cucumber.io/docs/terms/user-story/) – and talk about concrete examples of the new functionality to explore, discover and agree on the details of what’s expected to be done.
2. Next, document those examples in a way that can be automated, and check for agreement.
3. Finally, implement the behaviour described by each documented example, starting with an automated test to guide the development of the code.

**Types of testings**

### Accessibility Testing

Accessibility testing is the practice of ensuring your mobile and web apps are working and usable for users without and with disabilities such as vision impairment, hearing disabilities, and other physical or cognitive conditions.

### Acceptance Testing

Acceptance testing ensures that the end-user (customers) can achieve the goals set in the business requirements, which determines whether the software is acceptable for delivery or not. It is also known as user acceptance testing (UAT).

### Black Box Testing

Black box testing involves testing against a system where the code and paths are invisible.

### End to End Testing

End to end testing is a technique that tests the application’s workflow from beginning to end to make sure everything functions as expected.

### Functional Testing

Functional testing checks an application, website, or system to ensure it’s doing exactly what it’s supposed to be doing.

### Interactive Testing

Also known as manual testing, interactive testing enables testers to create and facilitate manual tests for those who do not use automation and collect results from external tests.

### Integration Testing

Integration testing ensures that an entire, integrated system meets a set of requirements. It is performed in an integrated hardware and software environment to ensure that the entire system functions properly.

### Load Testing

This type of non-functional software testing process determines how the software application behaves while being accessed by multiple users simultaneously.

### Performance Testing

Performance testing examines the speed, stability, reliability, scalability, and resource usage of a software application under a specified workload.

### Regression Testing

Regression testing is performed to determine if code modifications break an application or consume resources.

### Sanity Testing

Performed after bug fixes, sanity testing determines that the bugs are fixed and that no further issues are introduced to these changes.

### Security Testing

Security testing unveils the vulnerabilities of the system to ensure that the software system and application are free from any threats or risks. These tests aim to find any potential flaws and weaknesses in the software system that could lead to a loss of data, revenue, or reputation per employees or outsides of a company.

### Unit Testing

Unit testing is the process of checking small pieces of code to ensure that the individual parts of a program work properly on their own, speeding up testing strategies and reducing wasted tests.

**Testing frameworks for Angular**

Jasmine, karma, mocha, chai , protractor, cypress and etc

* **Jasmine** - is the framework we are going to use to create our tests. It has a bunch of functionalities to allow us the write different kinds of tests. Provided with many built in libraries to implement unit test cases .
* **karma.**[Karma](https://karma-runner.github.io/1.0/index.html) is a task runner for our tests. It uses a configuration file in order to set the startup file, the reporters, the testing framework, the browser among other things.

 karma configuration file created by angular-cli.

Text

Description automatically generated

Most of the configuration properties are

* **frameworks**: this is where jasmine gets set as a testing framework. If you want to use another framework this is the place to do it.
* **reporters**: this is where you set the reporters. You can change them or add new ones.
* **autoWatch**: if this is set to true, the tests run in watch mode. If you change any test and save the file the tests are re-build and re-run.
* **browsers:**this is where you set the browser where the test should run. By default it is chrome but you can install and use other browser launchers.

## Test entry file

The angular-cli configuration of karma uses the file “test.ts” as the entry point of the tests for the application. Let’s take a look to that file;

Text

Description automatically generated

* An environment to run angular tests is being created using all the imports at the beginning of the file.
* TestBed is a powerful unit testing tool provided by angular, and it is initialized in this file.
* Finally, karma loads all the test files of the application matching their names against a regular expression. All files inside our app folder that has “spec.ts” on its name are considered a test.

Our First Test File

import { TestBed, async } from '@angular/core/testing';

import { RouterTestingModule } from '@angular/router/testing';

import { AppComponent } from './app.component';

describe('AppComponent', () => {

  beforeEach(async(() => {

    TestBed.configureTestingModule({

      imports: [

        RouterTestingModule

      ],

      declarations: [

        AppComponent

      ],

    }).compileComponents();

  }));

  it('should create the app', () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.debugElement.componentInstance;

    expect(app).toBeTruthy();

  });

  it(`should have as title 'My First'`, () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.debugElement.componentInstance;

    expect(app.title).toEqual('My First');

  });

  it('should render title in a h1 tag', () => {

    const fixture = TestBed.createComponent(AppComponent);

    fixture.detectChanges();

    const compiled = fixture.debugElement.nativeElement;

    expect(compiled.querySelector('h1').textContent).toContain('My First');

  });

});