Learning Objectives

Learners will be able to…

- **Define TDD methodology**
- **Differentiate TDD and other forms of testing**
- **Create a script using TDD**
- **Identify the advantages and disadvantages of TDD**

info

Make Sure You Know

Intermediate Javascript including arrow functions

Limitations

This is only a primer on test-driven development

TDD

Test-Driven Development (TDD) is also called test-driven design. TDD is a method of software development during which code is tested over and over with unit testing.

Test Driven Development (TDD) is a software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each feature of the application are created and tested first. If the test fails, then the new code is written in order to pass the test.

Test-Driven Development starts with designing and developing tests for every small feature of an application. The TDD framework instructs developers to write new code only if an automated test has failed. This avoids duplication of code.

This technique is a balanced approach to programming that blends three activities: coding, testing (unit tests), and designing (refactoring). You must first have a comprehensive specification before you write any tests. In other words, TDD may be a smart approach to identify and streamline the specification before writing functional code within the line of Agile principles.

You should remember, that:
- TDD approach is neither about “Testing” nor about “Design”.
- TDD does not mean “write some of the tests, then build a system that passes the tests.”
- TDD does not mean “do lots of Testing.”

TDD Advantages

You only write code that's needed. Following TDD principles, you stop writing code when all of your tests pass. If your project needs another feature, you would use another test to drive the implementation of the feature. The code you write is the simplest code possible. So, all the code ending up within the product is actually needed to implement the features.

Code has a more modular design. In TDD, you consider one micro-feature at a time. Since you write the test first, the code automatically becomes easy to check. Code that is easy to check features a clear interface. This leads to a modular design for your application.

Code is easier to maintain. Because the different parts of your application are decoupled from one another and have clear interfaces, the code becomes easier to take care of. The modular nature of code made with TDD means you can update one micro-feature without affecting another. You can even keep the tests and rewrite the entire application. When all the tests pass, you are done.

Code is easier to refactor. Every feature is thoroughly tested. You do not need to be afraid to make drastic changes because if all the tests still pass, everything is okay. As your skills as a developer grow, you can go back and write better code for existing features. You can do this even when you do not recall exactly how the code works. With an entire test suite, you can easily improve the code without fear of breaking your application.

Code has high test coverage. There is a test for each feature of the specification, and the only code written is that which satisfies a test. This leads to a high test coverage, and develops confidence in your code.

Tests document the code. The test code shows you ways your code is supposed to be used. This acts as informal documentation for your code. The test shows what the code does and the way the interface should be used.

You spend less time debugging. How often have you ever wasted each day seeking out a nasty bug? How often have you copied a mistake message from your IDE and looked for it on the web?

TDD Disadvantages

TDD is not a silver bullet. Using this technique does not solve all of your problems. Tests help to seek out bugs, but they can not find them for you. Bugs can sometimes get past the testing process. Ultimately, humans write code, and humans are imperfect. More tests are not a substitute for quality code.

TDD is a slow process. If you begin TDD, you get the feeling that you need more time for straightforward implementations. After all, you write tests that you know will fail. Even simple features need the additional time of a test or two. It feels like there is a certain amount of “busy work” before finally starting to write code.

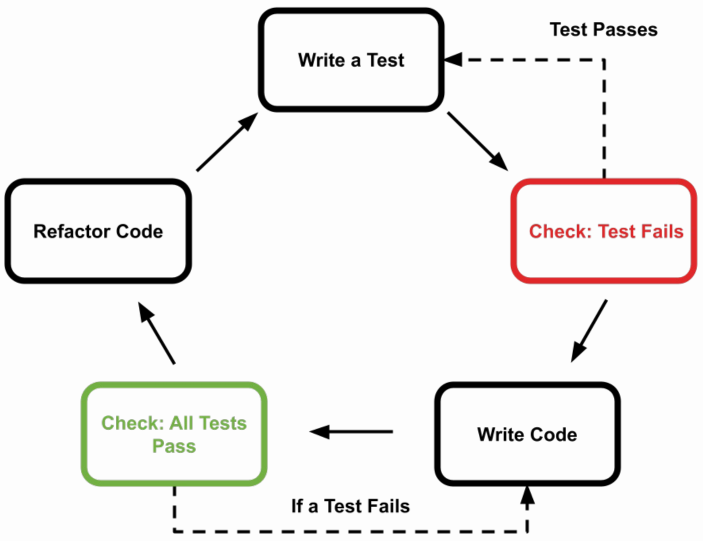
TDD works best when all the members of a team use it. As TDD influences the planning of code, it is recommended that either all the members of a team use TDD or nobody does. It can be difficult to justify TDD to management because they often believe that the implementation of the latest features takes longer if developers spend all that time writing tests when that code does not make it into the product. It helps if the entire team agrees on the importance of unit tests.

Tests have to be maintained when requirements change. Probably, the strongest argument against TDD is that it is a continual process. Just because the project enters production does not mean you can forget about it. When the specification changes, so must the code. That includes the tests as well. Because TDD is a slower process, this maintenance takes even more time.

TDD Life Cycle

TDD is cyclical in nature. Each test has a series of steps to follow. Once you get to the end, you repeat the process with the next test. Continue this until the specification has been fulfilled. The steps and image below depict the TDD life cycle.

1. Add a test.
2. Run the test. If it passes go back to step 1 and write a new test for the next feature.
3. Write some code with the goal of passing the failed test.
4. Run the test again. If it fails, go back to step 3.
5. Refactor the working code. Go back to step 1 and start the process for the next feature in the specification.



This image depicts the life cycle of TDD. There are five ovals in the shape of a pentagon. Arrows point from one oval to the next. Theprocess starts in the uppermost oval which says “Write a test”. The next oval says “Check test fails”. The third oval says “Write code”. The fourth oval says “Check all tests are OK”. The final oval says “Refactoring”.

TDD vs. Traditional Testing

The TDD approach is primarily a specification technique. At the outset, the specification is clearly stated. Every test written affirms a part of the specification. TDD ensures that your source code is thoroughly tested at a confirmatory level.

With traditional testing, a failing test finds one or more defects. This means going back and you have to go back and fix your code. This means reasoning about code that is already written. In contrast, a failing test in TDD informs you about what new feature to implement. You can think of failing tests as a sign of progress. The previous feature is complete, which means you are ready for the next one. Failing tests in the traditional paradigm indicate a lack of progress.

TDD ensures that your system actually meets requirements. Every test is linked to a part of the specification. Every piece of code is linked to passing a test. It helps to build your confidence in your system — the code you write is aligned with the specification.

In TDD more focus is on production code that verifies whether testing will work properly. That is, all code is written with the constraints of the pre-written test. In traditional testing, more focus is on test case design. That is, can you write a test that shows the current code is correct? Whether the test will show the proper/improper execution of the application in order to fulfill requirements.

In TDD, you achieve 100% coverage test. Every single line of code is tested, unlike traditional testing. You can achieve 100% coverage with traditional testing, but it is guaranteed like it is in TDD.

In Agile Modeling (AM), you should “test with a purpose”. You should know why you are testing something and what level it needs to be tested. This mentality aligns closely with TDD, as the tests are directly related to the specification. You can also do this with traditional testing, but it requires more forethought and intentionality.

There are some similarities between TDD and the traditional approach. The combination of both traditional testing and TDD leads to the importance of testing the system rather than perfection of the system. Achieving perfection is almost impossible. A robust set of tests, however, can ensure that the system can accomplish all of its goals, even if it is not perfect.

TDD Example

For our first TDD example, we are going to use the Node shell. We will copy and paste a few lines of code at a time. Pressing `ENTER` means Node will execute those lines and print any results in the terminal. Start by entering the command below in the terminal. When you see the `>` character, you are ready to begin.

```
node
```

This example is going to focus on TDD principles, so we will manually write simple tests without Jest. This example revolves around the function `gramsToKg` which takes a number representing grams and returns its equivalent in kilograms.

The first step of the TDD process is always writing a test. Let’s start by checking if passing `1000` to the `gramsToKg` returns `1`. Copy and paste the following code into the shell and press `ENTER`.

```
gramsToKg(1000) === 1;
```

We have yet to define the `gramsToKg` function, so we get a reference error since we did not define the function. Errors are a part of the TDD process, so the next step is to add code to address the error.

```
function gramsToKg(value) {}
gramsToKg(1000) === 1;
```

This new iteration of code is an improvement we no longer get an `undefined` error. However, we have another issue. Our boolean test returns `false`. Update our function so the boolean returns `true`.

```
function gramsToKg(value) {
  return 1;
}
gramsToKg(1000) === 1;
```

So far, so good; our test passes. Let’s add an additional test case that uses `100` instead of `1000`. This should fail since our function only returns `1`.

```
gramsToKg(100) === 0.1;
```

Modify `gramsToKg` so that it performs the proper conversion from grams to kilograms. Then add our two test cases from before. Both should pass.

```
function gramsToKg(value) {
  return value / 1000;
}
gramsToKg(1000) === 1;
gramsToKg(100) === 0.1;
```

To speed up our testing, create `testValues` which is an array of objects. Each object has the key `g` which represents an amount in grams, as well as the key `kg` which represents the same weight in kilograms.

```
const testValues = [
    { g: 1000, kg: 1 },
    { g: 100, kg: 0.1 },
    { g: 5000, kg: 5 },
    { g: 6, kg: 0.006 }
];
```

Iterate over the array with `map` and pass the value from `g` to the function. Verify that the function's output matches the value in `kg`. The return value should be an array with four elements. Each is the boolean value `true` since each test case passed.

```
testValues.map(item => gramsToKg(item.g) === item.kg);
```

Here we have implemented our first function using the TDD approach. We did not write a lot of code. Yet, this is a very simple example to understand the order in which you need to think when developing via testing.

What's great about this approach is that we know the result. We know exactly when that part of the program is ready and we won't write any more code. Because new code is written after a new test, we have confidence that the code is working correctly, and we have documented any special cases.

In the process, we run the code all the time and have a clear idea of how it works. If something doesn't work right, we fix it immediately. Otherwise, if an error is detected, we'll have to fix it in the finished code. This can take more time and effort.

Sum Function

Let’s create a better example of a function using TDD. This time, however, we are going to use an IDE and Jest. Normally you would not write the function and tests in the same file. We are breaking this rule for the sake of clarity in the TDD process.

As always, start first with a test for the function `sum`. Our first test expects that the function should be defined. The `toBeDefined` method looks for the definition of a specified function.

```
describe("Sum function", () => {
  test("function is defined", () => {
    expect(sum).toBeDefined();
  });
});
```

Run the test.

```
npm run test index.test.js
```

We should get the following error:

```
 FAIL  ./index.test.js
  Sum function
    ✕ function is defined (2 ms)

  ● Sum function › function is defined

    ReferenceError: sum is not defined
```

This should be expected as we have not yet defined the `sum` function. To fix this, define the function which takes two parameters and returns their sum.

```
const sum = (a, b) => {
  return a + b;
};

describe("Sum function", () => {
  test("function is defined", () => {
    expect(sum).toBeDefined();
  });
});
```

Run our test again, and this time it should pass.

```
npm run test index.test.js
```

Since all of the tests have passed, let's write another test. This time, check that the return value of the function is a number.

```
const sum = (a, b) => {
  return a + b;
};

describe("Sum function", () => {
  test("function is defined", () => {
    expect(sum).toBeDefined();
  });

  test("always return a number", () => {
    expect(typeof sum(1, 2)).toBe("number");
  });
});
```

Run the test once more. This should also pass.

```
npm run test index.test.js
```

Now we are going to add a test that verifies that the `sum` function is returning the proper value. Check that `sum(1, 2)` returns `3`.

```
const sum = (a, b) => {
  return a + b;
};

describe("Sum function", () => {
  test("function is defined", () => {
    expect(sum).toBeDefined();
  });

  test("always return a number", () => {
    expect(typeof sum(1, 2)).toBe("number");
  });

  test("sum 1 + 2 = 3", () => {
    expect(sum(1, 2)).toBe(3);
  });
});
```

All of the tests should pass when the code runs.

```
npm run test index.test.js
```

We need to break our `sum` function with some tests. For example, pass an object and an array to the function as arguments. The function should return `NaN` since we are trying to add two non-numerical values. Add these two tests to our code:

```
  test("[] + {}", () => {
    expect(sum([], {})).toBe(NaN);
  });

  test("{} + []", () => {
    expect(sum({}, [])).toBe(NaN);
  });
```

As expected, these tests should fail.

```
npm run test index.test.js
```

Update `sum` function to return `NaN` if either argument is not a number.

```
const sum = (a, b) => {
  if (typeof a !== "number" || typeof b !== "number") {
    return NaN;
  }
  return a + b;
};
```

Run the tests one final time. They should all pass, which means our function is complete.

```
npm run test index.test.js
```

While this example is rather simple, it does a good job of walking you through the TDD cycle. Start with a test, get the test to fail, add code so the test passes, write a new test, etc. This process continues until the code and tests meet the specifications.

▼ **Code**

Your code should look like this:

```javascript
const sum = (a, b) => {
  if (typeof a !== "number" || typeof b !== "number") {
    return NaN;
  }
  return a + b;
};

describe("Sum function", () => {
  test("function is defined", () => {
    expect(sum).toBeDefined();
  });

  test("always return a number", () => {
    expect(typeof sum(1, 2)).toBe("number");
  });

  test("sum 1 + 2 = 3", () => {
    expect(sum(1, 2)).toBe(3);
  });

  test("[] + {}", () => {
    expect(sum([], {})).toBe(NaN);
  });

  test("{} + []", () => {
    expect(sum({}, [])).toBe(NaN);
  });
});
```