Unit Testing and the Jest Testing Framework

- Now that we have a general idea about what software testing is, let's start to explore more deeply by writing some tests. We'll start off with unit testing.

- Remember, a ***unit test*** is a test that's designed to validate a small, focused piece (or "unit") of code, like a single function or a single class.

- Unit tests are typically small and inexpensive both to write and to run, and for this reason, they often comprise the bulk of a test suite. Thus, implementing effective unit tests is an important skill for developers to learn.

- Let's start off by exploring what a unit test actually looks like. Then, we'll explore a widely-used framework for implementing software tests called Jest (which, as we'll eventually see, can be used to implement broader-scoped tests than just unit tests). As we go, we'll also look at some important characteristics of well-written unit tests.

What does a unit test look like?

- Recall that the essential job of every test is to compare the expected behavior of the code being tested against its actual behavior. If the actual behavior matches the expected behavior, the test passes. Otherwise, it fails.

- So what does this look like in terms of actual code? Well, let's look at a very simple example. Imagine we're implementing a library of mathematical functions, and we want to test whether our `sum()` function works. Here's a simple test:

```
var sum = require('./lib/sum')
const result = sum(2, 2)
const expected = 4
if (result !== expected) {
    throw new Error(`${result} is not equal to
${expected}`)
}
```

- Note the simplicity of this test. In particular, it focuses on just **a single behavior with known inputs**, specifically the correct computation of the sum of 2 and 2, and it throws an error if the result of that behavior is not the same as what we expect. This test, as written, starts to give us confidence that the `sum()` function works as expected.

- However, as you might expect, this is not the way most tests are actually implemented. There are a few notable shortcomings with the test implementation above.

- For example, we'll typically want to include many tests in our test suite, e.g. multiple tests for the `sum()` function, tests for the `subtract()` function, etc. If all of our tests are written like the one above, the tests will stop running after any one of them fails and throws an error. This is a problem, since our test suite will be most helpful if it informs us about *all* the tests that are broken.

- In addition, if a test fails, it will be hard to discover which test failed. Specifically, while the error messaging that results from directly throwing an `Error` will contain useful information, it will be very verbose, and it'll take some real effort to pull the most important information out of that messaging. The time we spend here wading through error messaging to figure out which test failed and why will be time we can't spend doing more meaningful things like actually fixing the problem or developing new features.

- Typically, real-world tests are typically implemented using a ***testing framework***, which is an application-independent collection of rules and tools/components that can be used to define and execute tests.

- A good testing framework should have the following properties:
  - It should provide clear, straightforward mechanisms that make it easy to specify tests.
  - It should make it simple to run all of the tests associated with a project (or to easily run a specific subset of the tests).
  - It should execute tests quickly and in isolation from each other.
  - It should provide clear, helpful messaging about the results of each test that makes it easy to identify which tests failed and why.

- In this class, we'll use a wonderful JavaScript testing framework called Jest.

    - Testing frameworks for other programming languages will have many elements similar to Jest, though obviously the syntax and details will differ.

Getting started with the Jest testing framework

- Let's start to explore how to start writing tests using the Jest testing framework. Jest is one of the most widely used testing frameworks for JavaScript code. It is typically used in Node.js projects managed by NPM (or Yarn), which is how we'll use it in this course.

- **To follow along here, you'll need to make sure you have a relatively recent version of Node.js installed on your development machine.** To do this, you can follow the installation instructions at https://nodejs.org.
    - As of this writing, at least version 14.15.0 of Node.js is required to run the latest version of Jest.
    - If you're on a machine where you don't have control enough to install a newer version of Node.js (e.g. on the ENGR servers), you can use NVM to install a custom personal version: https://github.com/nvm-sh/nvm.

- To start with Jest from an existing Node.js project, we'll need to begin by installing it from the command line using `npm`:

```
npm install --save-dev jest
```

- Once Jest is installed, we'll add an entry to the "scripts" field of our project's `package.json` file to make it easier to run the tests we'll write with Jest:

```
"scripts": {
  "test": "jest"
}
```

- This will specifically allow us to execute our tests using any of the following command-line commands (they are all equivalent):

```
npm run test
npm test
npm t
```

- If we do run any of those commands now, before we've written any tests, we'll see that Jest will run and print an error message that says "no tests found." Within the rest of the text of this error message is a small tidbit that helps us better understand one of the nice features of Jest:

  ```
  testMatch: **/__tests__/**/*.[jt]s?(x),
    **/?(*.)+(spec|test).[tj]s?(x) - 0 matches
  ```

- This line is a bit cumbersome to read, especially if you aren't familiar with regular expressions, but if you squint, it'll tell you how Jest knows where to find our tests.

- Specifically, Jest looks for files within our project whose names/paths match one of the following two patterns. Any file matching either one of these patterns is assumed to contain tests and will be automatically executed by Jest:
  - Any file with one of the following file extensions within a directory named `__tests__/`: `.js`, `.jsx`, `.ts`, or `.tsx` (the latter two file extensions are for [TypeScript]() projects).
    - The `__tests__/` directory can live anywhere within our project, and we can even have multiple directories named `__tests__/` at different locations in the project.
  - Any file, anywhere in the project, whose name has the suffix `.test` (or `.spec`) immediately before one of the following file extensions: `.js`, `.jsx`, `.ts`, or `.tsx` (e.g. `calculator.test.js`).

- Let's start to see how to write tests in Jest. We'll start by creating a file that will match one of the patterns described above. For now, we'll create an isolated test file called `hello.test.js`, where we'll implement a few very basic initial tests.

- Jest provides different ways to implement and organize our tests, which we'll explore here as we go. For now, we'll look at the simplest mechanism Jest provides for implementing a test, which is the `test()` function.

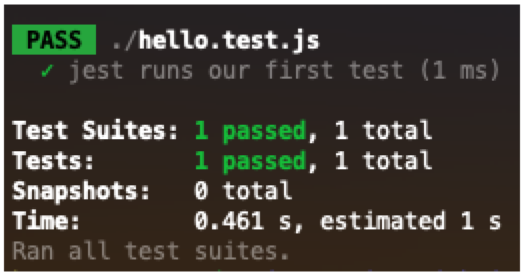
- Jest's `test()` function is used to specify a single test. It takes two arguments (plus an optional third argument, which we won't explore for now):
  - **The name/description of the test.** This name should be explicit and descriptive. It should typically describe both the action being taken and the expected outcome of that action, e.g. "multiplication of two positive numbers returns a positive number."

    - **A function encoding the test itself.** We'll talk more about how to design and structure this function in a minute.

- For now, let's start by writing a very, very basic test, just to make sure Jest successfully finds and runs our tests:

```
test("jest runs our first test", function () {})
```

- Here, you can see that the test function literally does nothing. However, if we run `npm test`, we'll see that Jest does indeed find and execute our test, reporting results that look something like this:



```
PASS  ./hello.test.js
  ✓ jest runs our first test (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.461 s, estimated 1 s
Ran all test suites.
```

Making assertions with matchers

- Every testing framework needs to support some mechanism for making ***assertions*** about the code being tested, that is, a mechanism that allows us to express the expected behavior of the code and to assert that its actual behavior matches the expected behavior.

- Jest comes with a built-in assertion framework that we'll explore here, but third-party assertion libraries also exist (such as Chai for JS).

- Jest's built-in assertion framework is built on top of a set of ***matchers*** that allow you to validate behaviors of the code being tested. These matchers are all used in conjunction with a function called `expect()`, which is used to register the *actual* behavior of the code being tested.

- Using `expect()` and a matcher to create an assertion looks something like this:

```
expect(2+2).toBe(4)
```

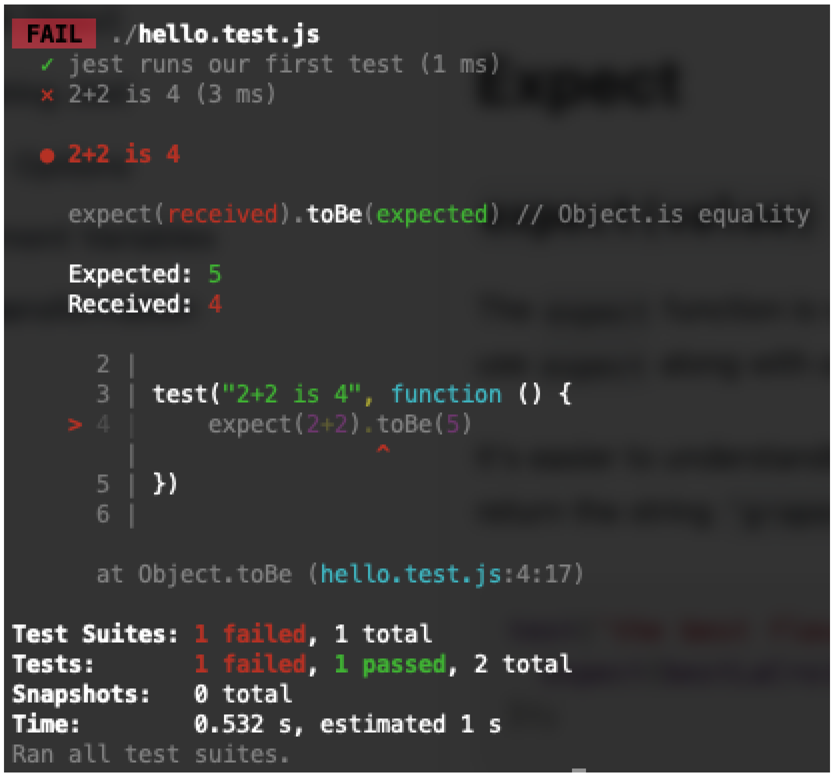
- Here, we're creating an assertion that validates the behavior of the `+` operator. In this case, `2+2`, the value passed to `expect()`, represents the actual/observed behavior of this operator. It specifically evaluates to the value that results from applying the `+` operator to two operands, 2 and 2. The value passed to the `toBe()` matcher represents the expected result of this operation.

- You may have noticed that the `expect()` function and Jest's matchers are set up so that an assertion can be read somewhat like an English sentence. The assertion above reads "Expect 2+2 to be 4."

- If we add the assertion above into a test, we should be able to run Jest again and see the new test pass:

  ```
  test("2+2 is 4", function () {
      expect(2+2).toBe(4)
  })
  ```

- If we change the value we pass into the `toBe()` matcher in the test above from 4 to 5 and rerun the tests, we'll see Jest report the test failure to us. It should look something like this:



```
 FAIL  ./hello.test.js
  ✓ jest runs our first test (1 ms)
  × 2+2 is 4 (3 ms)

  ● 2+2 is 4

    expect(received).toBe(expected) // Object.is equality

    Expected: 5
    Received: 4

      2 |
      3 | test("2+2 is 4", function () {
    > 4 |     expect(2+2).toBe(5)
        |                 ^
      5 | })
      6 |

      at Object.toBe (hello.test.js:4:17)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        0.532 s, estimated 1 s
Ran all test suites.
```

- Note that Jest provides us with a lot of important information here in an easy-to-understand format. We can specifically see which tests passed ("jest runs our first test") and which tests failed ("2+2 is 4"), and for the failed tests, we can see which specific assertions failed and how specifically the observed result (labeled "received" in the report above) differed from the expected result of each failed assertion.

- In the assertion above, we're using Jest's `toBe()` matcher, which performs an equality comparison between the expected and observed value. If those values are not equal (using JavaScript's `Object.is()` method for comparison), then the assertion fails. If they are equal, the assertion passes.

- The `toBe()` matcher is just one of many different matchers provided by Jest. We'll use a couple others below, and then we'll see yet more of them as we progress in this course.

Writing a real unit test

- The tests we implemented above were demonstrational, but they didn't actually validate anything. Let's actually implement a real unit test that validates a piece of code.

- Specifically, let's assume we're working on a project that contains a class called `Calculator`, which can be used to perform basic arithmetic operations. Assume the `Calculator` class's operations are executed following a pattern like this:

  ```
  const calc = new Calculator
  calc.calculate(2, 2, Calculator.ADD)
  ```

- Let's implement a test that validates the `Calculator` class's addition operation. Before covering *how* to implement our test, let's first talk about *where* to implement it. Let's assume that our `Calculator` class lives within our project at the file `lib/calculator.js`.

- It's typically considered good practice to store our tests close to the code they're testing. In this course, we'll typically follow the pattern of implementing our tests within a `__tests__/` directory directly next to the code being tested. Tests in the `__tests__/` directory will typically be factored into different files whose names match the names of the files being tested.

- In other words, we'll implement tests for the code in `lib/calculator.js` at `lib/__tests__/calculator.js`.
  - Note that other testing frameworks and other programming languages might dictate a different structuring of the tests.

- Now, in our test file, we'll import the code being tested and specify our first real unit test.

- Remember, **each unit test should be focused on a single behavior**, e.g. a single function call with a single set of known inputs. We should be careful not to try to do too much within a single test. **It's better to implement lots of small, focused, clear unit tests** than to try to implement fewer sprawling tests, each of which tries to validate multiple behaviors.

- Here, we'll start off by implementing a simple test that validates the result of passing a single known set of inputs (with a known expected output) into the calculator's addition operation. Remember, we need to provide a name for our test that's descriptive and explicit:

```
const Calculator = require("../calculator")
test("calculator adds 2 and 2 to get 4", function () {
    const calc = new Calculator()
    expect(calc.calculate(2, 2, Calculator.ADD)).toBe(4)
})
```

- And we have a real test! Now, we can run our tests (e.g. by running `npm test`), and if our `Calculator` class's addition operation is working correctly, we should see the new test pass.

- If you ran `npm test` to execute the tests, you may have noticed that Jest executed *all* of the tests in our project, including the ones we implemented above in `hello.test.js`. Maybe in this case, we don't want to run those tests, only the ones for the `Calculator` class. To accomplish this, we can simply pass one or more additional command line arguments to Jest to tell it which tests to execute.

- Specifically, additional arguments passed to Jest are treated as regular expressions, and only test files whose names match one of these regular expressions are executed. For example, if we only want to run the test suite in `lib/__tests__/calculator.js`, we can run the tests like this:

  ```
  npm test calculator
  ```

- Here, the `calculator` argument is passed to the underlying `jest` command that's executed by `npm test`. The command that's ultimately executed is `jest calculator`.

- Before we implement more tests, let's explore a mechanism for structuring tests to help make them easy to read and understand.

The arrange-act-assert pattern

- There are many different ways to structure tests. Whatever structuring mechanism we use, we want to make sure it gives us tests that are clear and easy to understand. In particular, **another developer should be able to look at**

our test code and quickly understand what each test is doing.

- One simple but widely used strategy for structuring tests is the ***arrange-act-assert* pattern** (or just the ***AAA*** pattern). Under the AAA pattern, each test is broken down into three separate sections:
    - **Arrange** – In the "arrange" section of the test, the preconditions for the test are set up to represent the state of the world before invoking the behavior being validated. For example, the "arrange" section is where objects are created and variables are set up to get ready to run the code being tested.
    - **Act** – The "act" section of the test, is where the behavior being validated is actually executed.
    - **Assert** – In the "assert" section of the test, we implement assertions to verify that the actual behavior of the code being tested matches its expected behavior.

- The AAA pattern is also sometimes referred to as the ***given-when-then* pattern**. Whatever name we use for this pattern, its main benefit is that it helps keep tests clear.

- For example, one of the most common ways the AAA pattern is violated is by mixing multiple actions and assertions together. While this can occasionally be acceptable for testing complex, multi-step behaviors, in general, it makes it hard to distinguish between the action being performed and the assertions on that action. In general, **we should strive to implement tests with a single "act" section and a single "assert" section**.

- The test we wrote above for the `Calculator` class follows the AAA pattern, though it compresses the "act" section of the test and the "assert" section of the test into a single line of code. If we wanted to explicitly follow the AAA pattern, we could have implemented the test like this:

```
test("calculator adds 2 and 2 to get 4", function () {
    // Arrange - set up a Calculator object to be tested
    const calc = new Calculator()

    // Act - use the calculator to compute 2+2
    const result = calc.calculate(2, 2, Calculator.ADD)
    // Assert - verify that the calculator computes 2+2=4
    expect(result).toBe(4)
})
```

Implementing behavior-driven unit tests

- Again, it is important for each unit test to validate a single behavior. Our first instinct may be to implement tests whose structure matches the structure of our code, e.g. by implementing ***method-driven tests***, that is, tests where each method in our code is validated completely by a single test.

- For example, we might be tempted to implement a single test for our `Calculator` class's `calculate` method, which validated *all* of that method's functionality, e.g.:

  ```
  // This is an anti-pattern.  Don't do this.
  test("Calculator.calculate() works correctly", function() {
      const calc = new Calculator()
      expect(calc.calculate(2, 2, Calculator.ADD).toBe(4)
      expect(calc.calculate(2, 2, Calculator.SUB).toBe(0)
      expect(calc.calculate(2, 2, Calculator.MUL).toBe(4)
      expect(calc.calculate(2, 2, Calculator.DIV).toBe(1)
  })
  ```

- **Writing method-driven tests like this is bad practice because it encourages the implementation of tests that *change* as our code changes.** We've discussed previously why we should strive to implement unchanging tests. With method-driven tests like this, though, we need to change the test every time we add new functionality to our method.

- Thus, a method-driven test like this can become a sprawling hodgepodge of different validations (and often different hacks) accumulated over time as the code being tested changes, becoming harder and harder to understand and maintain.

- Instead of structuring tests to match the structure of the code being tested (i.e. implementing method-driven tests), **we should instead strive to structure tests**

around *behaviors*, i.e. we should implement ***behavior-driven tests***.

- In the case of our `Calculator` class, focusing on testing behavior would mean implementing one or more tests to validate each individual part of the class's functionality. For example, we might implement a test (or more!) for each different arithmetic operation:

```
// 4 small tests for 4 behaviors is better than one big
test.
test("calculator adds 2 and 2 to get 4", function () {
    const calc = new Calculator()
    expect(calc.calculate(2, 2, Calculator.ADD)).toBe(4)
})

test("calculator subtracts 2 from 2 to get 0", function ()
{
    const calc = new Calculator()
    expect(calc.calculate(2, 2, Calculator.SUB)).toBe(0)
})

test("calculator multiplies 2 and 2 to get 4", function ()
{
    const calc = new Calculator()
    expect(calc.calculate(2, 2, Calculator.MUL)).toBe(4)
})

test("calculator divides 2 by 2 to get 1", function () {
    const calc = new Calculator()
    expect(calc.calculate(2, 2, Calculator.DIV)).toBe(1)
})
```

- It should be easy to see that none of these tests should need to change as our `Calculator` class's functionality expands. To test new functionality, we will simply add more tests, one for each behavior.

Testing boundary cases

- You may have heard the terms “***edge case***” and “***corner case***” and assumed they meant the same thing. Though these terms are often used interchangeably, these two terms technically have different definitions:
    - **Edge case** – An edge case is an input configuration (e.g. a set of arguments to a function) where a *single* parameter/setting lies at the extreme of its normal range (i.e. the configuration is at the “edge” of the possible configuration space).
    - **Corner case** – A corner case is an input configuration where *multiple* parameters/settings are at the extremes of their normal ranges (i.e. the configuration is at a “corner” of its possible configuration space).

- Here, we’ll use the umbrella term ***boundary case*** to refer to an input configuration that is either an edge case or a corner case, i.e. an input configuration that is *somewhere* at the boundary of the configuration space.

- Regardless of which of these terms we use (and it’s generally just fine to use them interchangeably—everyone will know what you mean), **it is important that we implement tests that explicitly cover boundary cases**.

- For example, a boundary case for our `Calculator` class occurs when we try to execute its division operation with a second argument of 0 (i.e. when we try to divide by 0). It is important to include a test that explicitly validates the behavior in this situation.

- For example, if we expect our `Calculator` class to throw a `RangeError` when we try to divide by 0, we should implement a test to verify that it does indeed behave that way, e.g.:

```
test("throws a RangeError when dividing by 0", function ()
{
    const calc = new Calculator()
    expect(function () {
        calc.calculate(2, 0, Calculator.DIV)
    }).toThrow(RangeError)
})
```

- Here, we're using Jest's `toThrow()` matcher, which is used to assert that an error/exception is thrown in a given situation. When using `toThrow()`, it's important to pass a *function* to `expect()` that executes the code we expect to throw an error, as we do in the example above. Otherwise, Jest will not be able to catch the error, and the assertion will always fail.

- Boundary cases can arise in many different situations:
    - One or more inputs is null/undefined.
    - One or more inputs is an empty array/string/object/etc.
    - One or more inputs is 0.
    - One or more inputs is a negative number.
    - One or more inputs is a very large number/string/array/etc.
    - One or more inputs is an object missing a required field.
    - Etc., etc., etc.

- However they may arise, it is very important when implementing tests that we think carefully about what the boundary cases are for the behaviors we're validating. When we determine what the boundary cases are, we must implement tests to cover them.

Organizing a test suite

- Often (usually, actually), we'll want to implement test suites that have several related tests, e.g. multiple tests on the same method. When this is the case, it can be useful to organize the test suite to group related tests together. This, in turn, makes the tests easier to understand and maintain, since it makes it easier to see at a glance all of the tests that are related to each other.

- Jest follows a pattern that is used in many testing frameworks by allowing us to group multiple related tests together within a `describe()` block.

- For example, if we had multiple tests for our `Calculator` class's multiplication functionality, we could group them together like this:

```
describe("multiplication operation", function () {
    test(
        "returns negative for opposite-signed inputs",
        function () {
            const calc = new Calculator()
            expect(calc.calculate(10, -10, Calculator.MUL))
                .toBeLessThan(0)
        }
    )

    test(
        "returns positive for two positive inputs",
        function () {
            const calc = new Calculator()
            expect(calc.calculate(10, 10, Calculator.MUL))
                .toBeGreaterThan(0)
        }
    )

    test(
        "returns positive for two negative inputs",
        function () {
            const calc = new Calculator()
            expect(calc.calculate(-10, -10,
Calculator.MUL))
                .toBeGreaterThan(0)
        }
    )

    ...
})
```

- When reporting failures for tests grouped this way, Jest will name the test by prepending the name passed to `describe()` to the name passed to `test()`, e.g.:

  ```
  multiplication operation › returns positive for two negative inputs
  ```

- Typically, we will want to arrange the tests for our project so that each source file being tested has its own separate file of tests nearby (e.g. in an adjacent `__tests__/` directory). For example, the most common pattern for structuring tests might produce a directory structure something like this:

```
project/
  ⊢-lib/
  |  ⊢-calculator.js
  |  ⊢-textUtils.js
  |  ⊢-__tests__/
  |      ⊢-calculator.js
  |      ⊢-textUtils.js
  ⊢-client/
     ⊢-formHandler.js
     ⊢-autocomplete.js
     ⊢-__tests__/
         ⊢-formHandler.js
         ⊢-autocomplete.js
```

Unit testing antipatterns: Logic and DRYness

- There are a few antipatterns that can make our tests harder to read, understand, and maintain, and we want to avoid these as much as possible.

Antipattern #1: Logic in tests

- The first antipattern we want to avoid is including logic in our tests, even simple logic.

- We may often be tempted to include logic in our tests, especially to avoid duplication of code in our tests or simply to streamline the testing code.

- It's important to remember, though, that **our goal should be to write tests that are clear, i.e. trivially correct at a quick inspection**. Including logic (e.g. loops, conditionals, and operations) in tests introduces complexity that requires mental computation to understand the test. This can often lead to the concealment of bugs within our tests.

- Here's an example of a test that includes logic that is very basic but that nonetheless helps to conceal a subtle bug in the test (see if you can figure out the bug before reading on):

```
// This test implements an antipattern by including logic.
test.skip(
    "Navigator generates correct URL when navigating to
/about",
    function () {
        const baseUrl = "http://www.example.com/"
        const path = "/about"
        const nav = new Navigator(baseUrl)
        nav.goTo(path)
        expect(nav.url()).toBe(baseUrl + path)
    }
)
```

- At a glance, it might be very hard to figure out the problem with this test (or even to notice the logic in the test).
- The problem arises through the introduction of the string concatenation logic: `baseUrl + path`. The result of this string concatenation here is actually a URL that (incorrectly) contains a double slash: `http://www.example.com//about`.
- If we assume the desired behavior of the `Navigator` class is to correctly handle and eliminate double slashes, then this test would fail to detect incorrect behavior by that class. In fact, the test would fail unless the `Navigator` class behaved *incorrectly* by generating a double slash in the URL.
- A better version of this test would eliminate the logic:

```
// This test does not include logic.
test.skip(
    "Navigator generates correct URL when navigating to
/about",
    function () {
        const nav = new
Navigator("http://www.example.com/")
```

```
        nav.goTo("/about")

expect(nav.url()).toBe("http://www.example.com/about")
    }
)
```

- Note how clear and easy to understand this test is. It requires no mental computation to figure out what the test is doing, and we can see at a glance that it is doing the correct thing.

- The test above includes just very simple logic but still results in a bug that is hard to detect. More complex logic like loops and conditionals can make bugs in our tests even harder to spot, so we want to avoid this.

Antipattern #2: Tests that are too DRY

- You may have heard someone use the acronym ***DRY*** (***don't repeat yourself***) to capture an important aspect of good software development.

- When someone encourages us to write DRY code, they mean we should generally strive to avoid implementing code that contains a lot of repetitive, redundant, or duplicated logic. Instead, when we recognize that similar or identical logic is needed in multiple places in the code, we should factor that logic into a reusable form, like a function or a class.

- In general, the DRY principle is a good one to follow when we're coding, since it typically makes our code far easier to maintain. For example, when we factor repeated logic into a function, we only have to modify that function if we want to change the behavior of the logic in question instead of modifying each duplicated block of unfactored logic.

- **The one exception where DRY code is not necessarily good is in our tests.** This is because when we're writing tests, we want each test to be simple, clear, and self-evidently correct. Factoring testing code into separate functions or classes or even maintaining constant values that are shared by multiple tests can have the same effect as including logic in tests, adding complexity to the tests and making them harder to follow.

- **When writing tests, it's OK to have repeated code** if that repeated code makes each test stand on its own.

- Here's an example of a set of tests that are too DRY—pretty much all of the setup and validation logic in them is factored into helper functions that may be defined very far from where the tests are defined:

```
// These tests implement an antipattern: they are too DRY.
test(
    "chat room allows multiple users to register",
    function (){
        const users = createUsers([ false, false ])
        const chatroom =
createChatroomAndRegisterUsers(users)
        validateChatroomAndUsers(chatroom, users)
    }
)

test(
    "chat room does not allow a banned user to register",
    function () {
        const users = createUsers([ true ])
        const chatroom =
createChatroomAndRegisterUsers(users)
        validateChatroomAndUsers(chatroom, users)
    }
)

/*
 * Imagine there are hundreds of lines of tests here
between the
 * tests above and the helper functions below...
 */
function createUsers(isBannedValues) {
    const users = []
    isBannedValues.forEach(function (isBanned) {
        users.push(new User(isBanned ? User.BANNED :
User.NORMAL))
    })
    return users
}

function createChatroomAndRegisterUsers(users) {
    const chatroom = new Chatroom()
    users.forEach(function (user) {
        try {
            chatroom.register(user)
        } catch (e) {
            /* ignore BannedUserError */
        }
    })
    return chatroom
}

function validateChatroomAndUsers(chatroom, users) {
    expect(chatroom.isReachable()).toBeTruthy()
    users.forEach(function (user) {
        expect(chatroom.hasRegisteredUser(user))
            .toBe(user.state !== User.BANNED)
    })
}
```

- Here, although the bodies of the tests themselves are very concise (and DRY), the important details of the tests are factored away into helper functions (which themselves contain logic, making everything that much harder to understand at a glance). These tests are much clearer and easier to verify when they are written to stand on their own:

```
// These tests stand on their own and are not too DRY.
test(
    "chat room allows multiple users to register",
    function () {
        const user1 = new User(User.NORMAL)
        const user2 = new User(User.NORMAL)
        const chatroom = new Chatroom()
        chatroom.register(user1)
        chatroom.register(user2)


expect(chatroom.hasRegisteredUser(user1)).toBe(true)

expect(chatroom.hasRegisteredUser(user2)).toBe(true)
    }
)

test(
    "chat room does not allow a banned user to register",
    function () {
        const user = new User(User.BANNED)
        const chatroom = new Chatroom()
        try {
            chatroom.register(user)
        } catch (e) {
            /* ignore BannedUserError */
        }


expect(chatroom.hasRegisteredUser(user)).toBe(false)
    }
)
```

- There are many different ways that DRYness can creep into tests and make them harder to understand. **We should try to avoid all of the following patterns**:
    - Using shared values (e.g. global values) across multiple tests in a way that hides the purpose of those values.

- Factoring test setup into a function in a way that makes it hard to understand how the setup is performed or hard to override setup values to make it more clear what values are being used in individual tests.
- Factoring complex validations (i.e. assertions) into a function.