TESTES E PERFORMANCE WEB

TECNOLOGIAS E SISTEMAS DE INFORMAÇÃO PARA A WEB

POLITÉCNICO DO PORTO
ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

M09 – UNIT TESTS

TSIW 2023/2024

AGENDA

1. The goal of unit testing;
2. What is a unit test?
3. The anatomy of a unit test;
4. The four pillars of a good unit test;
5. Mocks;
6. Styles of unit testing;
7. JEST – JavaScript Testing Framework.

THE GOAL OF UNIT TESTING

1. THE GOAL OF UNIT TESTING

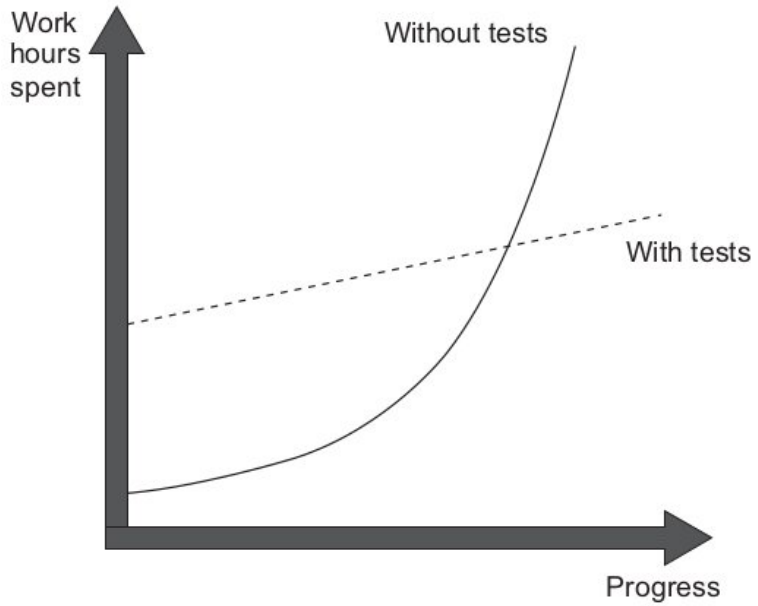
The current state of unit testing

- Unit test is now considered mandatory in most companies;
- There is no longer any dispute as to whether unit tests is important or not;
- When it comes to enterprise application development, almost every project includes at least some unit tests;
- A significant percentage of such projects go far beyond that: they achieve good code coverage with lots and lots of unit tests;
- The ration between the production code and the test code could be anywhere between 1:1 and 1:3 (for each line of production code, there are one to three lines of test code);
- Sometimes, this ration goes much higher than that, to a whopping 1:10;
- But with all new technologies, unit testing continues to evolve.

1. THE GOAL OF UNIT TESTING

The goal of unit testing

- The difference between good and bad tests is not merely a matter of taste or personal preference, it is a matter of succeeding or failing at critical project you are working on;
- The goal of unit testing is to enable sustainable growth of the software project;
- A project without tests has a head start but quickly slows down to the point that is hard to make any progress;
- This phenomenon of quickly decreasing development speed is also known as software entropy;
- Entropy is the among of disorder in a system.



1. THE GOAL OF UNIT TESTING

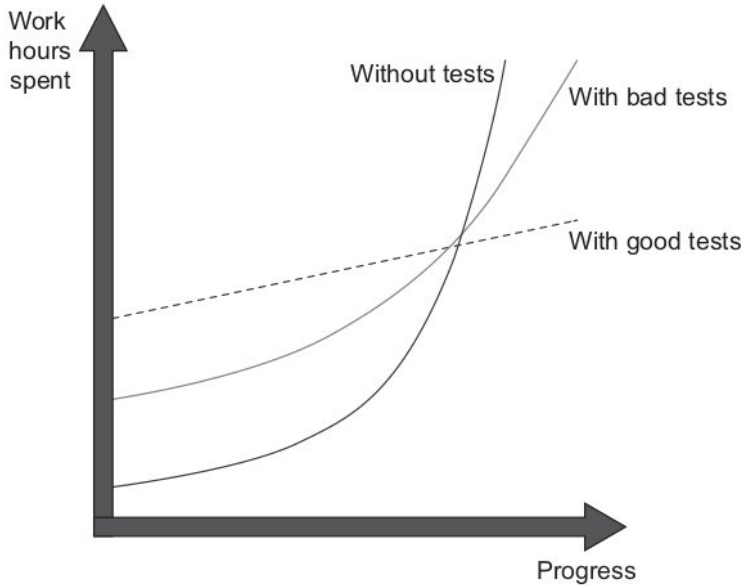
The goal of unit testing

- In software, entropy manifest in the form of code that trends to deteriorate;
- Each time you change something in a code base, the among of disorder in it increases;
- If left without proper care, such as constant cleaning and refactoring, the system becomes increasingly complex and disorganized;
- Tests help overturn this tendency. They act as a safety net – a tool that provides insurance against a vast majority of regressions;
- Tests help make sure the existing functionality works, even after you introduce new features or refactor the code to better fit new requirements;
- The downsize is that tests require initial – sometimes significant – effort.

1. THE GOAL OF UNIT TESTING

The goal of unit testing

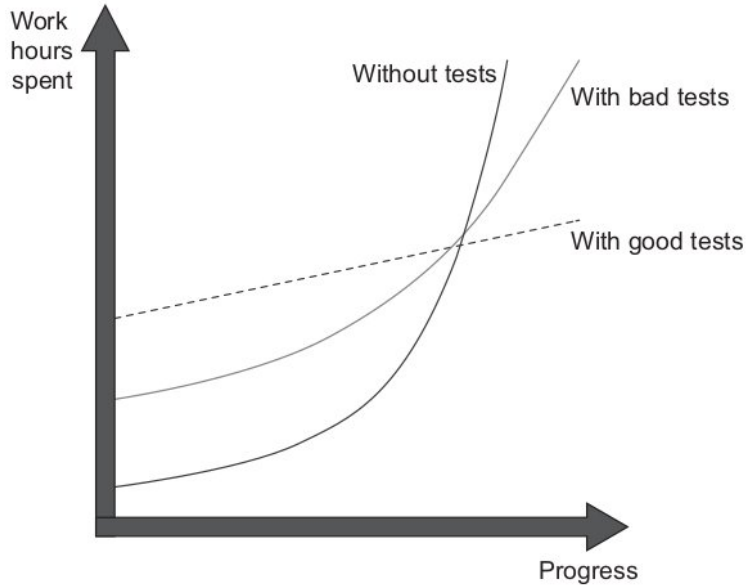
- Sustainability and scalability are the keys. They allow you to maintain development speed in the long run.
- Although unit testing helps maintain project growth, it is not enough to just write tests;
- Badly written tests still result in the same picture;
- Stagnation is still inevitable.



1. THE GOAL OF UNIT TESTING

The goal of unit testing

- Not all tests are created equal. Some of them are valuable and contribute a lot to overall software quality. Others don’t;
- They raise false alarms, don’t help to catch regression errors, and are slow and difficult to maintain;
- To enable sustainable growth, you have to exclusively focus on high-quality tests.



1. THE GOAL OF UNIT TESTING

Coverage metrics to measure test suite quality

- A coverage metric shows how much source code a test suite executes, from 0 to 100%;
- The common belief is that the higher the coverage number, the better;
- Unfortunately, coverage metrics, while providing valuable feedback, can't be used to effectively measure the quality of a test suite;
- If a metric shows that you covered 10% of your code base it means that you are not testing enough but if a measure shows that 100% of the code base is covered it doesn't mean that you have a good quality test suite;
- A test suit can provide high coverage but can be of poor quality.

1. THE GOAL OF UNIT TESTING

Coverage metrics to measure test suite quality

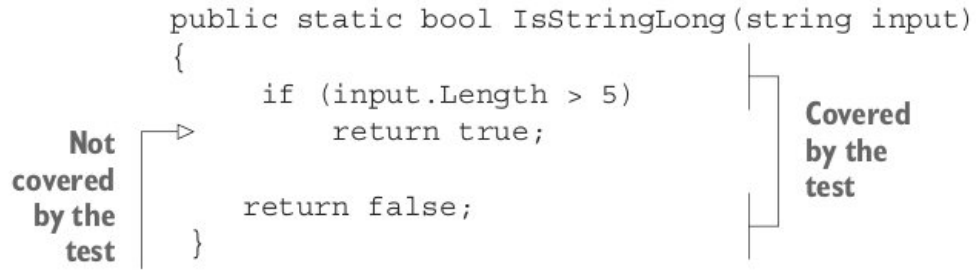
- The first and most used coverage metric is ***code coverage*** (or test coverage) and shows the ratio of the number of code lines executed by at least one test and the total number of lines in the production code base:

$$\text{Code coverage (test coverage)} = \frac{\text{Lines of code executed}}{\text{Total number of lines}}$$

1. THE GOAL OF UNIT TESTING

Coverage metrics to measure test suite quality

- In the following example we have a code coverage of 4/5, which means 80%:




```
public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

- With refactoring we can increase coverage. In the following example we have 3/3, which means 100%:

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}
```

```
public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

1. THE GOAL OF UNIT TESTING

Coverage metrics to measure test suite quality

- In the previous cases the increase in coverage doesn't mean that the test itself was improved;
- The previous example shows how easy is it to game the coverage numbers;
- The more compact your code is, the better the test coverage metric becomes;
- Another coverage metric is called **branch coverage.** This provides more precise results than code coverage because it helps cope with code coverage's shortcomings;
- Instead of using the raw number of code lines, this metric focuses on control structures, such as if and switch statements;
- It shows how many of control structures are traversed by at least on test in the suite:

$$\text{Branch coverage} = \frac{\text{Branches traversed}}{\text{Total number of branches}}$$

1. THE GOAL OF UNIT TESTING

Coverage metrics to measure test suite quality

- In this case we have a branch coverage of 1/2, which means 50% because we have two branches (argument greater than 5 and less than 5) and the test covers only one branch:

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

1. THE GOAL OF UNIT TESTING

Coverage metrics to measure test suite quality

- Although the branch coverage metric yields better results than code coverage, you still can rely on either of them to determine the quality of your test suite:
  - You can't guarantee that the test verifies all the possible outcomes of the system under test:

```
public static bool WasLastStringLong { get; private set; }

public static bool IsStringLong(string input)
{
    bool result = input.Length > 5;
    WasLastStringLong = result;
    return result;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

First outcome

Second outcome

The test verifies only the second outcome.

  - No coverage metric can take into account code paths in external libraries.

1. THE GOAL OF UNIT TESTING

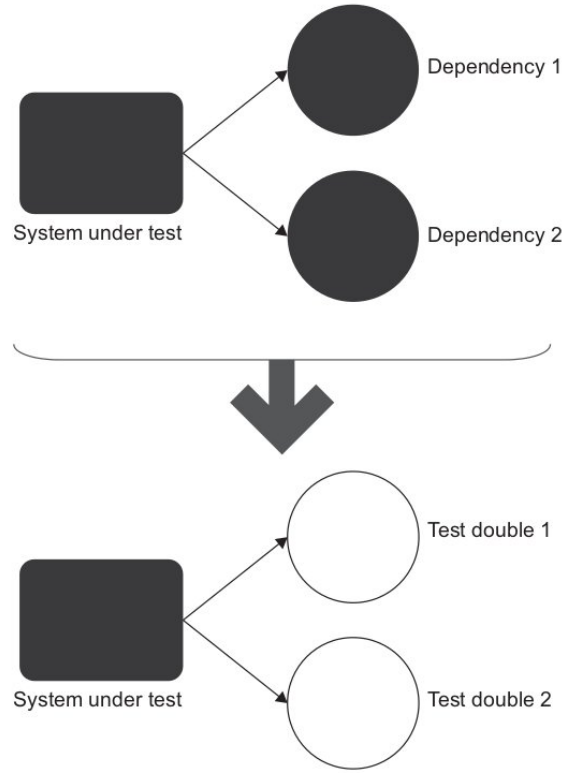
What makes a successful test suite?

- A successful test suite has the following properties:
  - It is integrated into the development cycle;
  - It targets only the most important parts of your code base;
  - It provides maximum value with minimum maintenance costs.

WHAT IS A UNIT TEST?

2. WHAT IS A UNIT TEST?

- A unit test is an automated test that:
  - Verifies a small piece of code/behaviour (also known as unit);
  - Does it quickly;
  - And does it in an isolated manner (London School):
    - Means that if a class has a dependency on another class, or several classes, you need to replace all with test doubles;
    - Test doubles are objects that looks and behaves like its release-intended counterpart but is actually a simplified version that reduces the complexity and facilitates testing;
    - If test fails you know for sure that the problem is in the system under test.



2. WHAT IS A UNIT TEST?

- In this example (Classical School) the test doesn't replace the dependency but rather uses a production-ready instance of it;

```
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(store, Product.Shampoo, 5);

    // Assert
    Assert.True(success);
    Assert.Equal(5, store.GetInventory(Product.Shampoo));
}
```

Reduces the product amount in the store by five

- Any bug inside store class can affect the customer. The two classes are not isolated.

2. WHAT IS A UNIT TEST?

- In this example (London School) we replace dependencies with test doubles — specifically, mocks;

```
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(true);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    // Assert
    Assert.True(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Once);
}
```

2. WHAT IS A UNIT TEST?

- A mock is a special kind of test double that allows you to examine interactions between the system under test and its collaborators (dependencies);
- Mocks is a subset of test doubles;
- Test doubles describes all kinds of non-productive ready, fake dependencies in a test.

	Isolation of	A *unit* is	Uses test doubles for
London school	Units	A class	All but immutable dependencies
Classical school	Unit tests	A class or a set of classes	Shared dependencies

2. WHAT IS A UNIT TEST?

- So, the London School states that the units under test should be isolated from each other. A unit under test is a unit of code, usually a class. All of its dependencies, except immutable dependencies, should be replaced with test doubles in tests;
- The Classical School (Detroit) states that the unit tests need to be isolated from each other, not units. A unit under test is a unit of behaviour, not a unit of code. Only shared dependencies should be replaced with test doubles. Shared dependencies are dependencies that provide means for tests to affect each other's execution flow.

THE ANATOMY OF A UNIT TEST

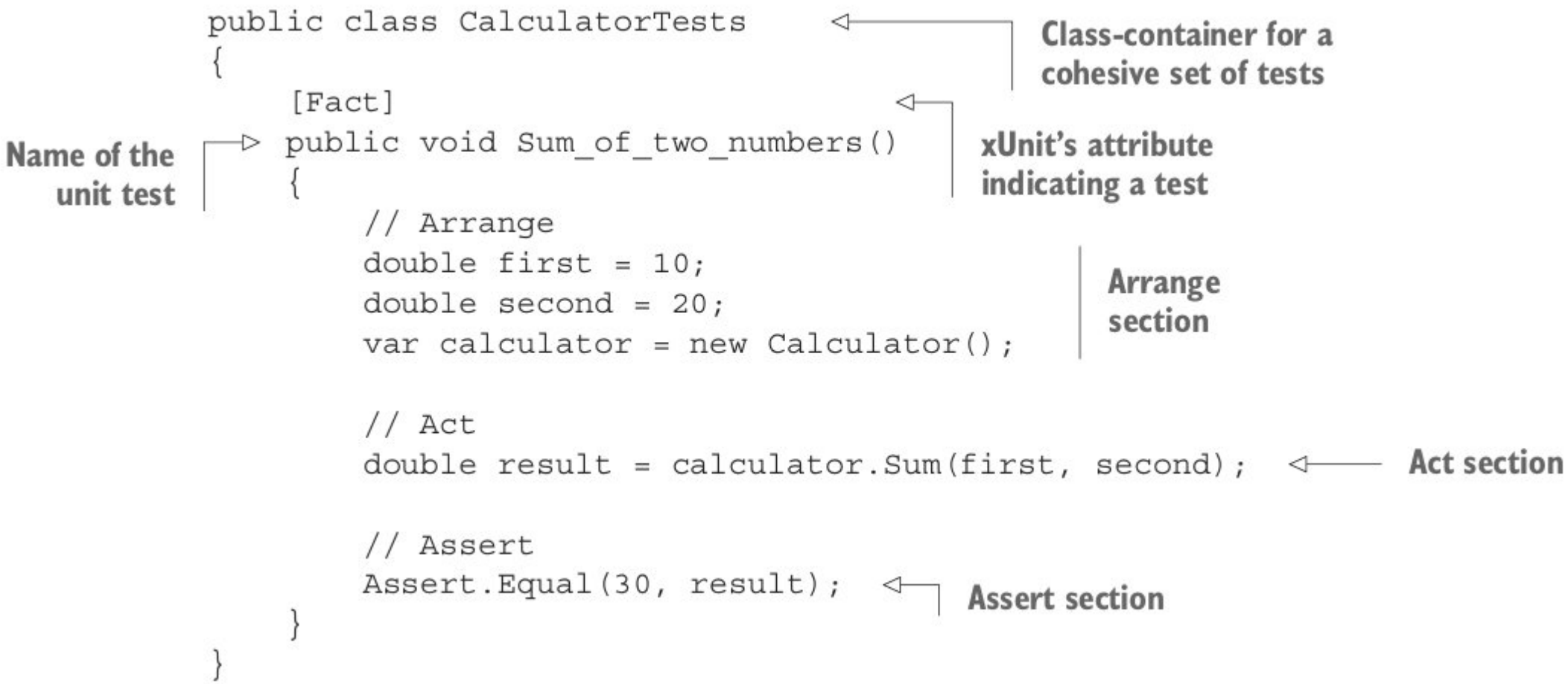
3. THE ANOTOMY OF A UNIT TEST

AAA Pattern

- The AAA pattern advocates for splitting each test using the following stages:
  - Arrange – bring the system under test (SUT) and its dependencies to a desire state;
  - Act – call methods on the SUT, pass the prepared dependencies, and capture the output value (if any);
  - Assert – verify the outcome. The outcome may be represented by the return value, the final state of the SUT and its collaborators, or the methods the SUT called on those collaborators.

3. THE ANOTOMY OF A UNIT TEST

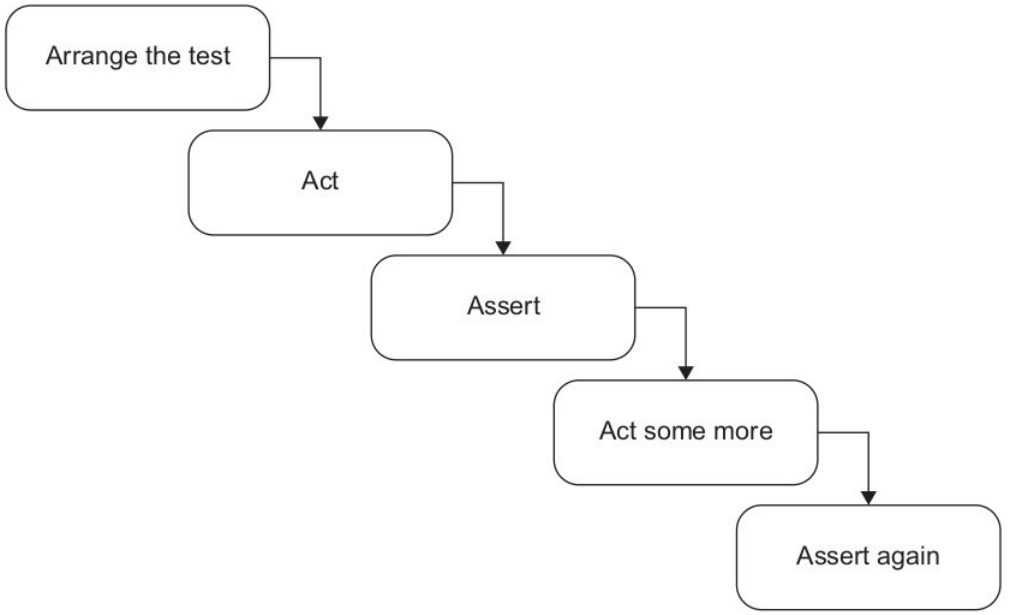
AAA Pattern



3. THE ANOTOMY OF A UNIT TEST

Avoid multiple AAA sections

- Occasionally, you may encounter a test with multiple AAA sections;
- Multiple act sections, separated by assert and, possibly, arrange sections means the test verifies multiple units of behaviour. And, such test is no longer a unit test but rather is an integration test;
- It is better to avoid such structure. A single action ensures that your tests remain within the realm of unit testing;



3. THE ANOTOMY OF A UNIT TEST

Avoid if statements in tests

- You may sometimes encounter a unit test with an if statement;
- This is also an anti-pattern;
- A test should be a simple sequence of steps with no branching;
- An if statement indicates that the test verifies too many things at once. Such a test, therefore, should be split into several tests;
- If statements make the tests harder to read and understand.

3. THE ANOTOMY OF A UNIT TEST

How large should each section be?

- The arrange section is usually the largest;
- The act section is normally just a single line of code. A two or more lines could indicate a problem with the system under test;
- Regarding assert section, a single unit of behaviour can exhibit multiple outcomes, and it is fine to evaluate them all in one test.

```
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(store, Product.Shampoo, 5);

    // Assert
    Assert.True(success);
    Assert.Equal(5, store.GetInventory(Product.Shampoo));
}
```

THE FOUR PILLARS OF A GOOD UNIT TEST

4. THE FOUR PILLARS OF A GOOD UNIT TEST

- A good unit test has the following four attributes:
  - Protection against regressions;
  - Resistance to refactoring;
  - Fast feedback;
  - Maintainability.

4. THE FOUR PILLARS OF A GOOD UNIT TEST

1. Protection against regressions

- A regression is a software bug;
- It is when a feature stops working as intended after some code modification, usually after you roll out new functionalities;
- The more features you develop, the more chances there are that you will break one of those features with a new release;
- The larger the codebase, the more exposure it has to potential bugs;
- That' is why it is crucial to develop a good protection against regressions. Without such protection, you won't be able to sustain the project growth in a long run.

4. THE FOUR PILLARS OF A GOOD UNIT TEST

1. Protection against regressions

- To evaluate how well a test scores on the metric of protecting against regressions, you need to take into account the following:
  - The amount of code that is executed during the test;
  - The complexity of that code;
  - The code's domain significance.

4. THE FOUR PILLARS OF A GOOD UNIT TEST

2. Resistance to refactoring

- Refactoring means changing existing code without modifying its observable behaviour. The intention is usually to improve the code's non-functional characteristics;
- The resistance to refactoring is the degree to which a test can sustain a refactoring of the underlying application code without turning red (failing);
- A **false positive** is a false alarm and it is a result indicating that the test fails (after a refactoring), although in reality, the functionality it covers works as intended;
- To evaluate how well a test scores on the metric of resisting to refactoring, you need to look at how many false positives the test generates. The fewer, the better.

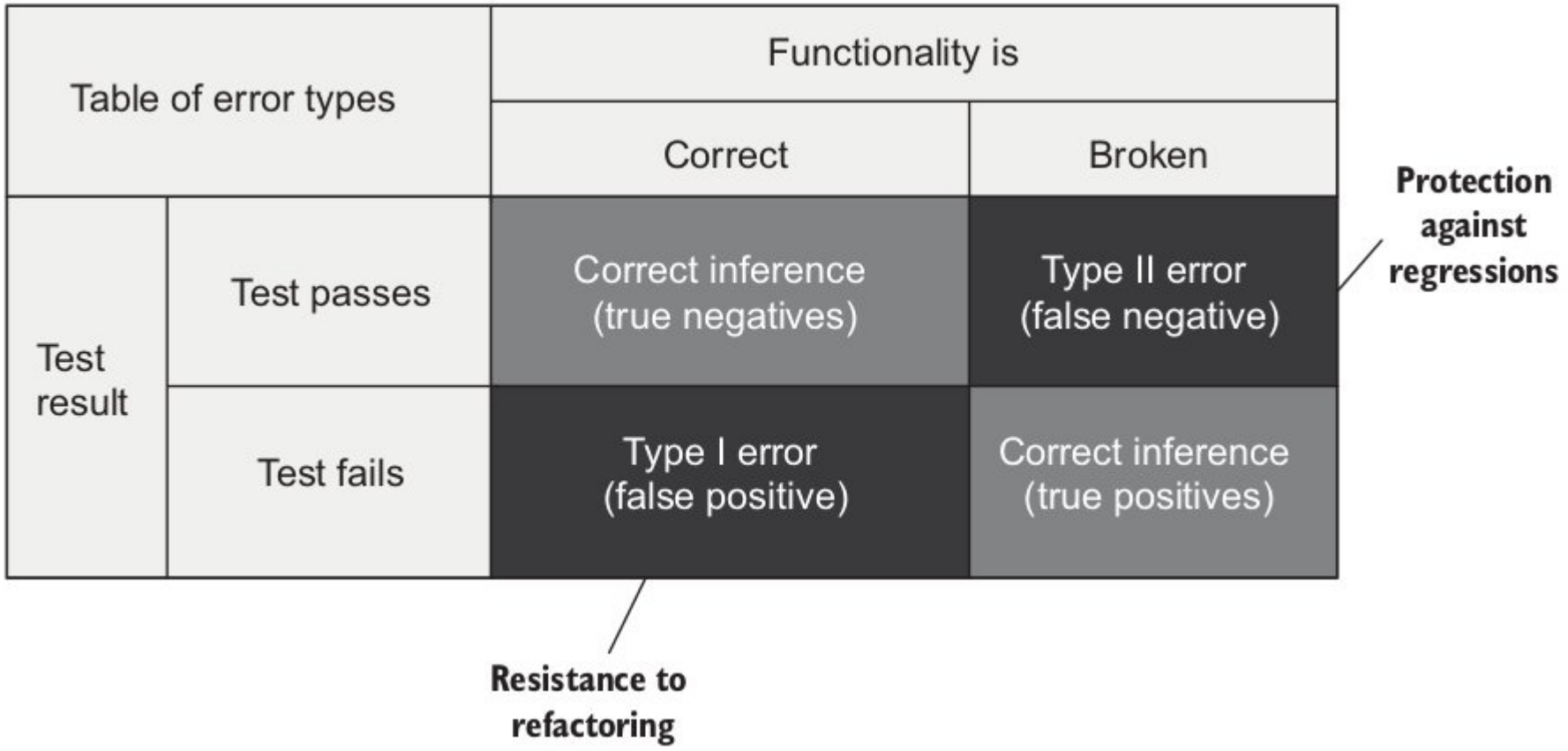
4. THE FOUR PILLARS OF A GOOD UNIT TEST

2. Resistance to refactoring

- The more the test is coupled to the implementation details of the SUT, the more false alarms it generates;
- The only way to reduce the change of getting a false positive is to decouple the test from those implementation details;
- You have to make sure that the test verifies the end result the SUT delivers, its observable behaviour, not the steps it takes to do that.

4. THE FOUR PILLARS OF A GOOD UNIT TEST

Protection against regressions and resistance to refactoring importance



4. THE FOUR PILLARS OF A GOOD UNIT TEST

3. Fast feedback

- Fast feedback is an essential property of a unit test;
- The faster the test, the more of them you can have in the suite and the more often can run them;
- The point here is that the tests begins to warm you about bugs as soon as you break the code, thus reducing the cost of fixing those bugs almost to zero.

4. THE FOUR PILLARS OF A GOOD UNIT TEST

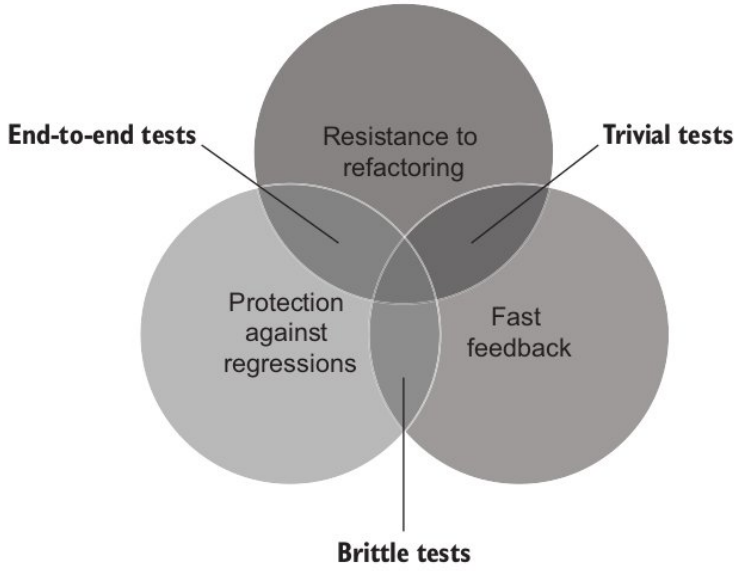
4. Maintainability

- How hard it is to understand the test – this component is related to the size of the test. The fewer lines of code in the test, the more readable the test is. The quality of the test code matters as much as the production code. Don’t cut corners when writing tests;
- How hard it is to run the test – if the test works with out-of-process dependencies, you have to spend time keeping those dependencies operational: reboot the database sever, resolve the network connectivity issues, and so on.

4. THE FOUR PILLARS OF A GOOD UNIT TEST

In search of ideal test

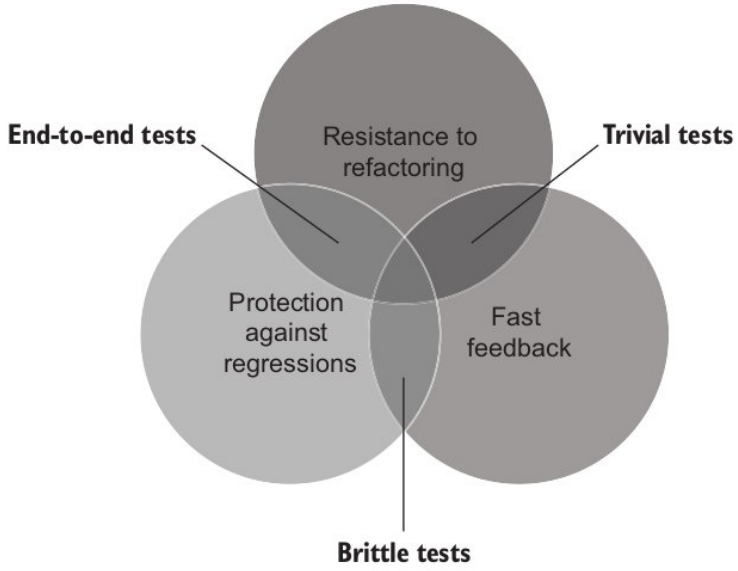
- **End-to-end tests**
  - Look at the system from the end user's perspective;
  - They normally go through all of the system's components, including the UI, database, and external applications;
  - Since they exercise a lot of code, they provide best protection against regressions;
  - Are also immune to false positives because they test the behaviour, so they have a good resistance to refactoring;
  - The major drawback is that they are slow.



4. THE FOUR PILLARS OF A GOOD UNIT TEST

In search of ideal test

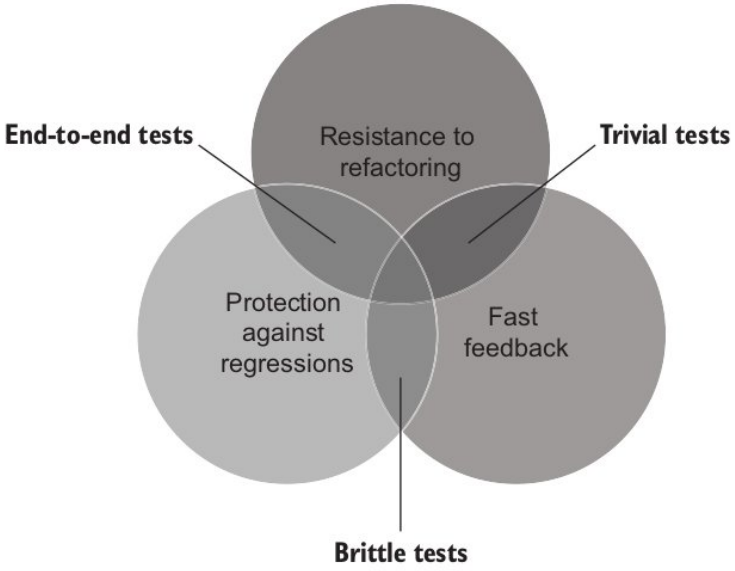
- **Trivial tests**
  - Such test covers a simple piece of code, something that is unlikely to break because is is too trivial;
  - They provide fast feedback;
  - They have a slow chance of producing false positives, so they are good resistance to refactoring;
  - But are unlikely to reveal any regressions, because there's not much room for a mistake in the underlying code.



4. THE FOUR PILLARS OF A GOOD UNIT TEST

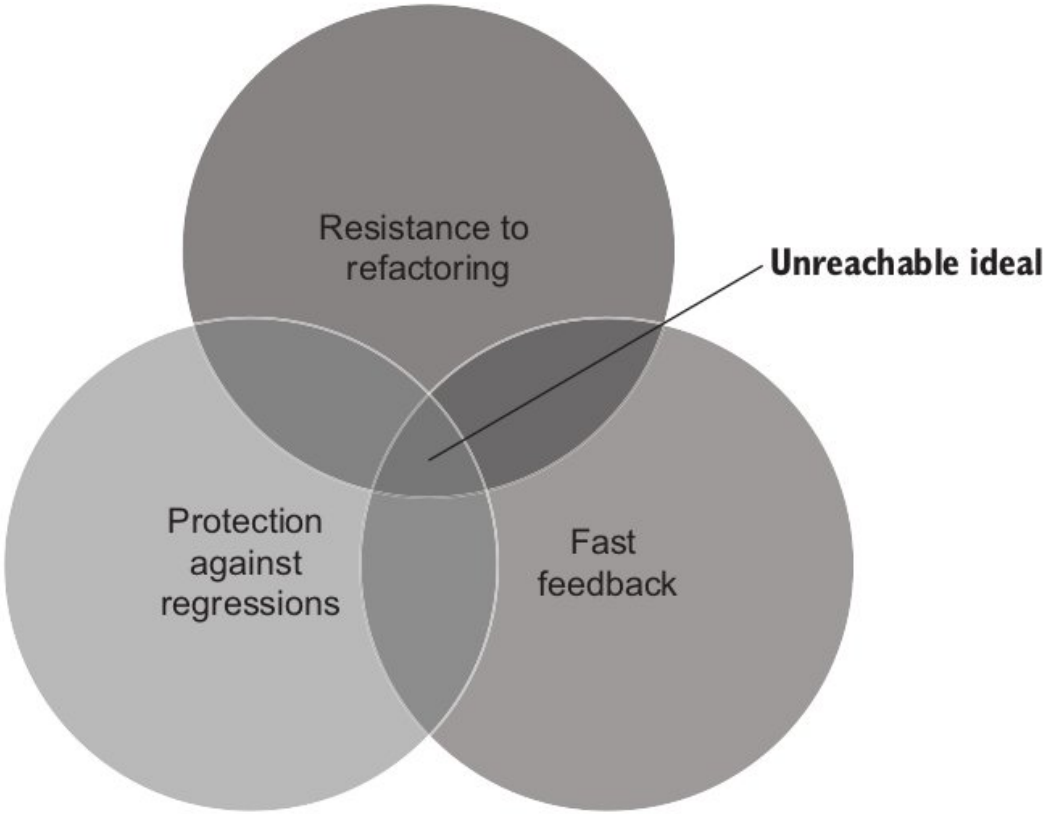
In search of ideal test

- **Brittle tests**
  - When a test is focused in the implementation and not in the behaviour we are in the presence of a Brittle test;
  - This can run fast and provides a good protection against regressions, but have little resistance to refactoring.



4. THE FOUR PILLARS OF A GOOD UNIT TEST

In search of ideal test



4. THE FOUR PILLARS OF A GOOD UNIT TEST

Choosing between black-box and white-box testing

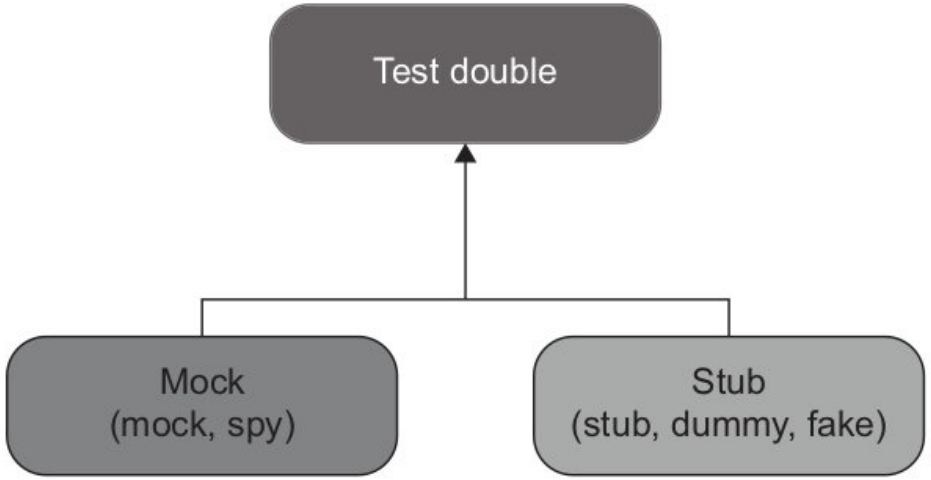
- Black-box testing is a method of software testing that examines the functionality of a system without knowing its internal structure. Such testing is normally built around specifications and requirements: what the application is supposed to do, rather then how it does it;
- White-box testing is the opposite of that. It is a method of testing that verifies the application's inner workings. The tests are derived from the source code, not requirements or specifications.

	Protection against regressions	Resistance to refactoring
White-box testing	Good	Bad
Black-box testing	Bad	Good

MOCKS

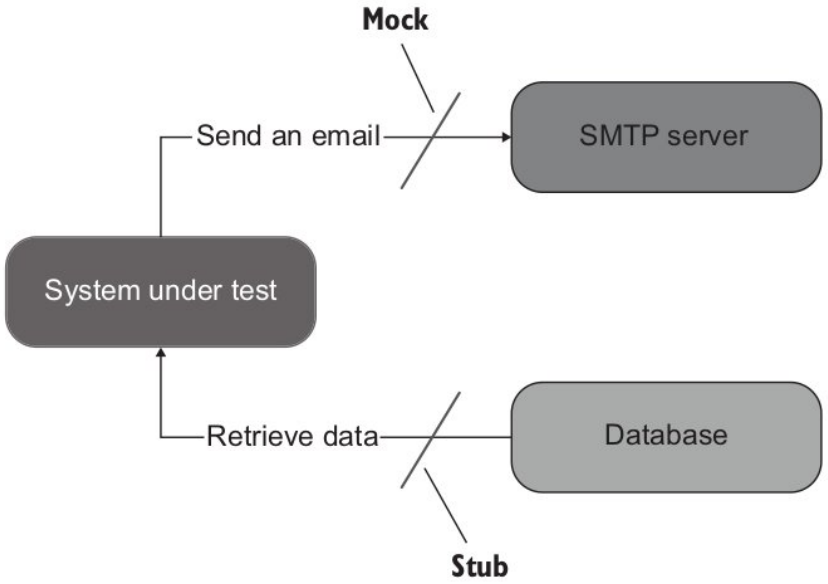
5. MOCKS

- A test double describes all kinds of non-productions-ready fake dependencies in tests;
- The major use of test doubles is to facilitate testing;
- They are passed to the system under test (SUT) instead of real dependencies, which could be hard to set up or maintain.



5. MOCKS

- **Mocks** – helps to emulate and examine outcoming interactions. These interactions are calls the SUT makes to its dependencies to change their state;
- **Stubs** – helps to emulate incoming interactions. These interactions are calls the SUT makes to its dependencies to get input data.



STYLES OF UNIT TESTING

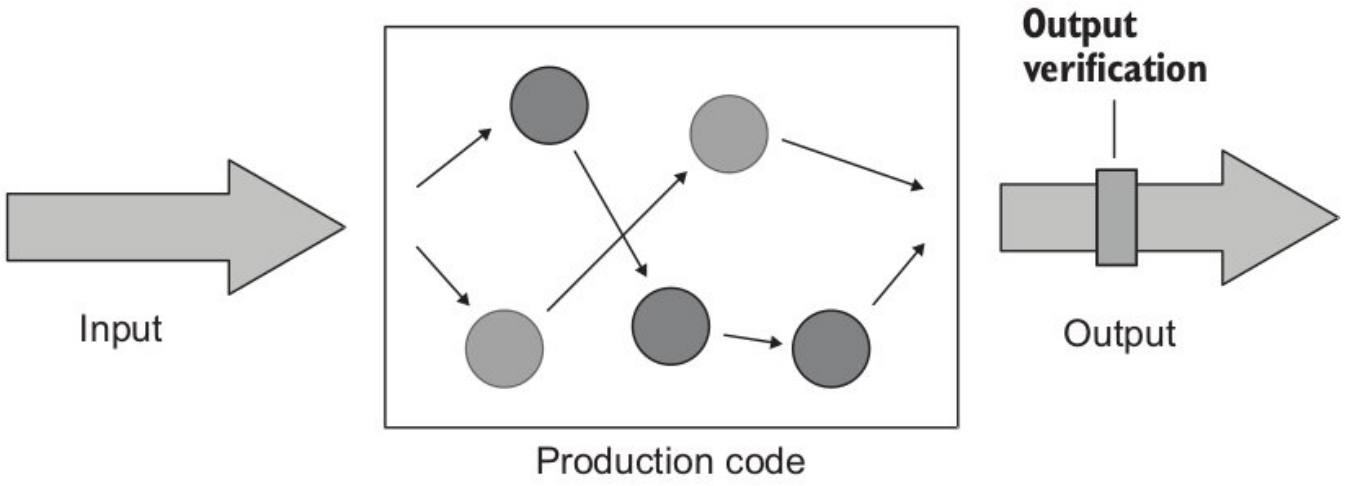
6. STYLES OF UNIT TESTING

- There are three styles of unit testing:
  - Output-based testing;
  - State-based testing;
  - Communication-based testing.
- You can employ one, two, or event all three styles together in a single test.

6. STYLES OF UNIT TESTING

Output-based testing

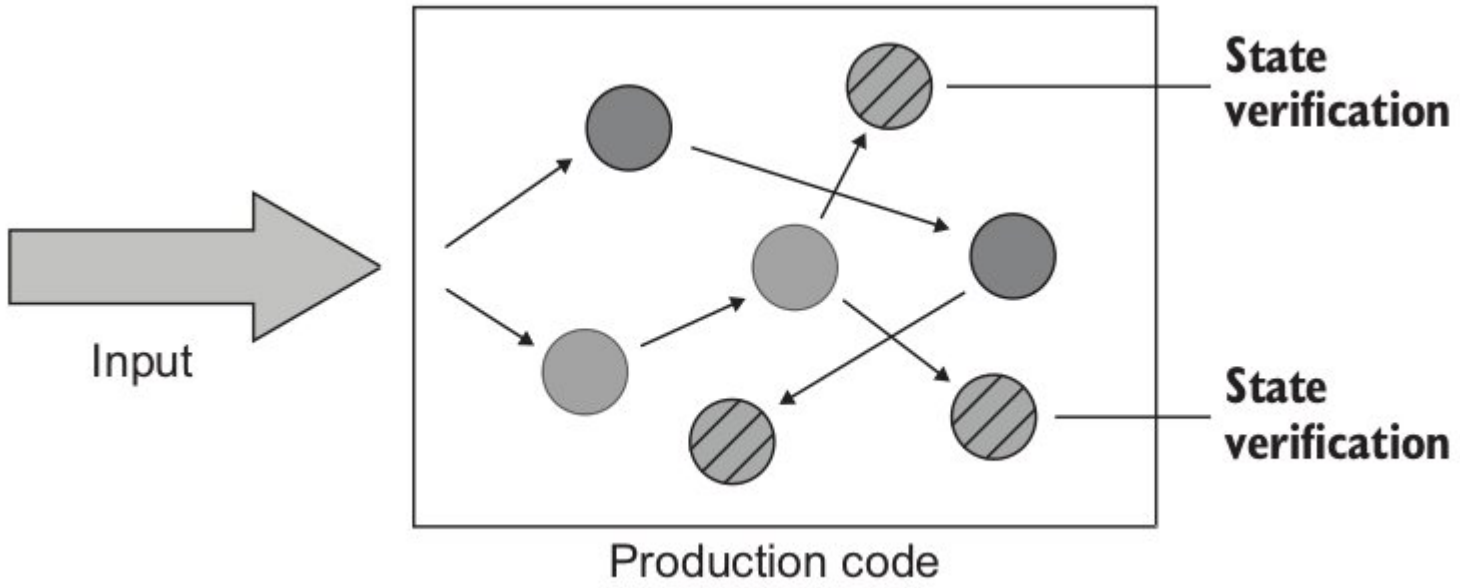
- In this style you feed an input to the system under test (SUT) and check the output it produces;
- This type of test is only applicable to code that doesn't change a global or internal state, so the only component to verify is its return value.



6. STYLES OF UNIT TESTING

State-based testing

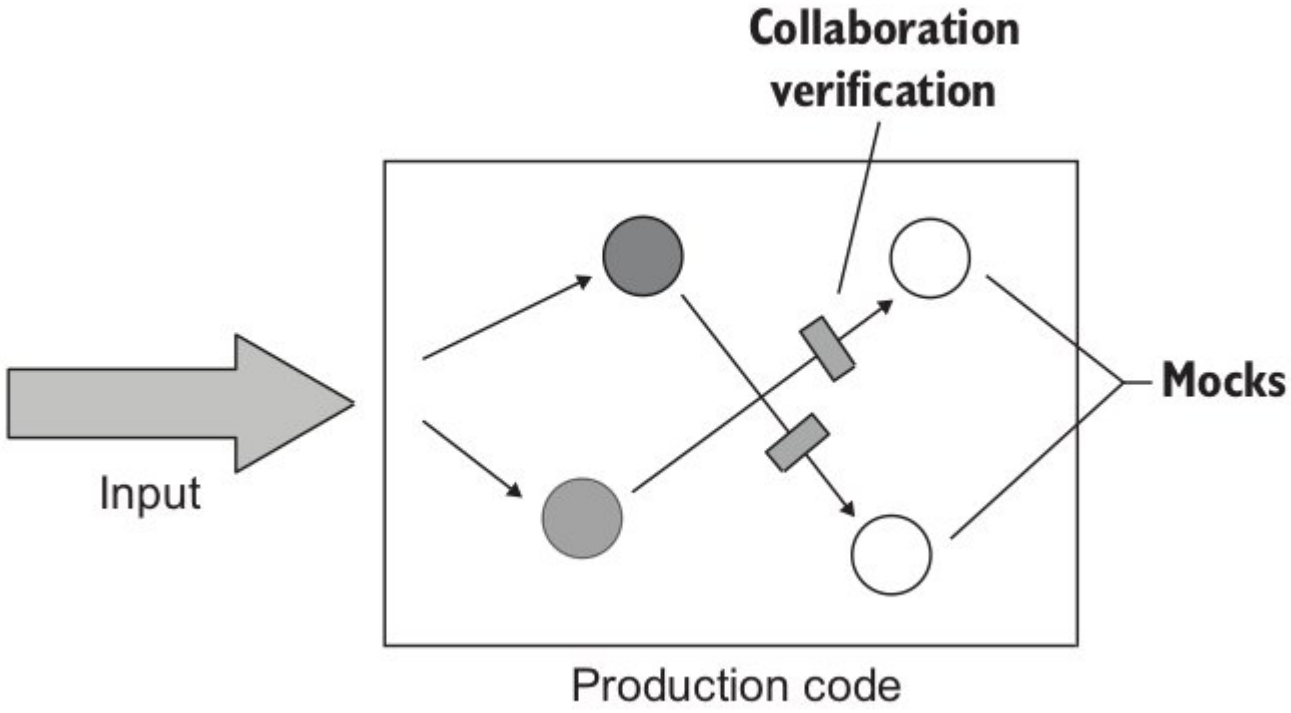
- This style is about verifying the state of the system after an operation is complete;
- The term state in this type of testing can refer to the state of the SUS itself, or one of its collaborators, or of an out-of-precess dependency, such as the database or the filesystem.



6. STYLES OF UNIT TESTING

Communication-based testing

- This style uses mocks to verify communication between the SUT and is collaborators.



JEST – JAVASCRIPT TESTING FRAMEWORK

7. JEST - JAVASCRIPT TESTING FRAMEWORK

- Jest is a JavaScript Testing Framework with focus on simplicity;
- It works with projects using: Babel, TypeScript, Node, React, Angular, Vue, Etc...;
- Jest aims to work out of the box, config free, on most JavaScript projects;
- Make tests which keep track of large objects with ease. Snapshots live either alongside your projects, or embedded inline;
- Tests are parallelized by running them in their own processes to maximize performance;
- Has the entire toolkit in one place, well documented, well maintained.

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Fast and safe

- By ensuring your tests have unique global state, Jest can reliably run tests in parallel;
- To make things quick, Jest runs previously failed tests first and re-organizes runs based on how long tests files take.

Code coverage

- Generate code coverage;
- No additional setup needed;
- Jest can collect code coverage information from entire projects, including untested files.

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Easy mocking

- Jest uses a custom resolver for imports in your tests, making it simple to mock any object outside of your test's scope;
- You can used mocked imports with the Mock Functions API to spy on functions call with readable test syntax.

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Getting Started - https://jestjs.io/docs/getting-started

npm

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

Let's get started by writing a test for a hypothetical function that adds two numbers. First, create a `sum.js` file:

```
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

Then, create a file named `sum.test.js`. This will contain our actual test:

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Getting Started - https://jestjs.io/docs/getting-started

Add the following section to your `package.json`:

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

Finally, run `yarn test` or `npm test` and Jest will print this message:

```
PASS  ./sum.test.js
✓ adds 1 + 2 to equal 3 (5ms)
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Matchers

- Jest uses “matchers” to let you test values in different ways;
- https://jestjs.io/docs/using-matchers
- Common matchers:
  - **toBe()** - test exact equality;
  - **toEqual()** – recursively checks every field of an object or array.

```
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});
```

```
test('object assignment', () => {
  const data = {one: 1};
  data['two'] = 2;
  expect(data).toEqual({one: 1, two: 2});
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Matchers

- Truthiness matchers:
  - **toBeNull()** – matches only null;
  - **toBeUndefined()** – matches only undefined;
  - **toBeDefined()** – is the opposite of toBeUndefined();
  - **toBeTruthy()** – matches anything that an if statement treats as true;
  - **toBeFalsy()** – matches anything that an if statement treats as false.

```
test('null', () => {
  const n = null;
  expect(n).toBeNull();
  expect(n).toBeDefined();
  expect(n).not.toBeUndefined();
  expect(n).not.toBeTruthy();
  expect(n).toBeFalsy();
});

test('zero', () => {
  const z = 0;
  expect(z).not.toBeNull();
  expect(z).toBeDefined();
  expect(z).not.toBeUndefined();
  expect(z).not.toBeTruthy();
  expect(z).toBeFalsy();
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Matchers

- Numbers matchers:
  - **toBeGreaterTnan();**
  - **toBeGreaterThanOrEqual();**
  - **toBeLessThan();**
  - **toBeLessThanOrEqual();**
  - **toBe() and toEqual()** – are equivalent for numbers;
  - **toBeclose()** – used for floating point equality.

```
test('two plus two', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);

  // toBe and toEqual are equivalent for numbers
  expect(value).toBe(4);
  expect(value).toEqual(4);
});
```

```
test('adding floating point numbers', () => {
  const value = 0.1 + 0.2;
  //expect(value).toBe(0.3);           This won't work because of rounding error
  expect(value).toBeCloseTo(0.3); // This works.
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Matchers

- String matchers:
  - **toMatch()** – check strings against regular expressions.

```
test('there is no I in team', () => {
  expect('team').not.toMatch(/I/);
});

test('but there is a "stop" in Christoph', () => {
  expect('Christoph').toMatch(/stop/);
});
```

- Arrays and iterables:
  - **toContain()** – check if an array or iterable contains a particular item.

```
const shoppingList = [
  'diapers',
  'kleenex',
  'trash bags',
  'paper towels',
  'milk',
];

test('the shopping list has milk on it', () => {
  expect(shoppingList).toContain('milk');
  expect(new Set(shoppingList)).toContain('milk');
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Matchers

- Exceptions matchers:
  - **toThrow()** – test whether a particular functions throws an error when it is called.

```
function compileAndroidCode() {
  throw new Error('you are using the wrong JDK');
}

test('compiling android goes as expected', () => {
  expect(() => compileAndroidCode()).toThrow();
  expect(() => compileAndroidCode()).toThrow(Error);

  // You can also use the exact error message or a regexp
  expect(() => compileAndroidCode()).toThrow('you are using the wrong JDK');
  expect(() => compileAndroidCode()).toThrow(/JDK/);
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- Often while writing tests you have some setup work that needs to happen before test runs;
- Often you have also some finishing work that needs to happen after test runs;
- **Setup can be repeated for many times:**
  - **beforeEach()** – repeats inner content before each test being executed;
  - **afterEach()** – repeats inner content after each test has been executed.

```
beforeEach(() => {
  initializeCityDatabase();
});

afterEach(() => {
  clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **One-time setup:**
  - In some cases, you only need to do setup once, at the beginning of a file;
  - **beforeAll()** – executes its inner content only once, in the beginning of the execution of a test suite before all tests;
  - **afterAll()** - executes its inner content only once, in the end of the execution of a test suite after all test being executed.

```
beforeAll(() => {
  return initializeCityDatabase();
});

afterAll(() => {
  return clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **Scoping:**
  - By default, the beforeAll() and afterAll() blocks apply to every test in a file;
  - You can also group tests together using a describe block;
  - When they are inside a describe block, the beforeAll() and afterAll() only apply to the tests within that block.

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **Scoping:**

```
// Applies to all tests in this file
beforeEach(() => {
  return initializeCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});

describe('matching cities to foods', () => {
  // Applies only to tests in this describe block
  beforeEach(() => {
    return initializeFoodDatabase();
  });

  test('Vienna <3 veal', () => {
    expect(isValidCityFoodPair('Vienna', 'Wiener Schnitzel')).toBe(true);
  });

  test('San Juan <3 plantains', () => {
    expect(isValidCityFoodPair('San Juan', 'Mofongo')).toBe(true);
  });
});
```

```
beforeAll(() => console.log('1 - beforeAll'));
afterAll(() => console.log('1 - afterAll'));
beforeEach(() => console.log('1 - beforeEach'));
afterEach(() => console.log('1 - afterEach'));
test('', () => console.log('1 - test'));
describe('Scoped / Nested block', () => {
  beforeAll(() => console.log('2 - beforeAll'));
  afterAll(() => console.log('2 - afterAll'));
  beforeEach(() => console.log('2 - beforeEach'));
  afterEach(() => console.log('2 - afterEach'));
  test('', () => console.log('2 - test'));
});

// 1 - beforeAll
// 1 - beforeEach
// 1 - test
// 1 - afterEach
// 2 - beforeAll
// 1 - beforeEach
// 2 - beforeEach
// 2 - test
// 2 - afterEach
// 1 - afterEach
// 2 - afterAll
// 1 - afterAll
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **Order of execution of describe and test blocks:**
  - Jest executes all describe handlers in a test before it executes any of the actual tests;
  - Once the describe blocks are complete, by default Jest runs all the tests serially in the order they where encountered in the collection phase, waiting for each to finisish and be tidied up before moving on.

```
describe('outer', () => {
  console.log('describe outer-a');

  describe('describe inner 1', () => {
    console.log('describe inner 1');
    test('test 1', () => {
      console.log('test for describe inner 1');
      expect(true).toEqual(true);
    });
  });

  console.log('describe outer-b');

  test('test 1', () => {
    console.log('test for describe outer');
    expect(true).toEqual(true);
  });

  describe('describe inner 2', () => {
    console.log('describe inner 2');
    test('test for describe inner 2', () => {
      console.log('test for describe inner 2');
      expect(false).toEqual(false);
    });
  });

  console.log('describe outer-c');
});

// describe outer-a
// describe inner 1
// describe outer-b
// describe inner 2
// describe outer-c
// test for describe inner 1
// test for describe outer
// test for describe inner 2
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- If a test is failing, one of the first things to check should be whether the test is failing when it is the only test that runs;
- To run only one test with jest, temporarily change that test command to a test.only.

```
test.only('this will be the only test that runs', () => {
  expect(true).toBe(false);
});

test('this test will not run', () => {
  expect('A').toBe('A');
});
```

7. JEST - JAVASCRIPT TESTING FRAMEWORK

Coverage

- To check the coverage metrics of your test suite you need to install jest globally;
- Example: npm install –g jest;
- To get report you need to run jest with --coverage flag.

```
PASS tests/sum.test.js
-----------------|---------|----------|---------|---------|-------------------
File             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------|---------|----------|---------|---------|-------------------
All files        |      75 |       50 |     100 |      75 |
 sum.js          |     100 |      100 |     100 |     100 |
 validate-nif.js |   66.66 |       50 |     100 |   66.66 | 3,8
-----------------|---------|----------|---------|---------|-------------------
```