Testing in JavaScript with Mocha and Chai

Miguel Ordoñez Morales alu0101281087@ull.edu.es
Basilio Gómez Navarro alu0101049151@ull.edu.es

Universidad de La Laguna

“There are two ways to write error-free programs; only the third one works.”

Alan J. Perlis

Index

Software Testing

- Definition
- Importance
- Types
- Testing Pyramid

Methodology

- Priori
- Posteriori

Index

Suppose that we want to check if this function works correctly...

```
function isPrime(n) {
 for (let i = 2; i < n; i++) {
   if (n % i === 0) {
     return false;
   }
 }
 return true;
}
```

- What guarantee do we have that it works for **any** number I give it as input?
- And if af I give it a negative number?
- Numbers are infinite.
- We **can't** check all the numbers, it's **impossible**.

What is Software Testing?

*Activity to check whether the actual results match the expected results and to ensure that the software system is defect **free**.*

Bug #415

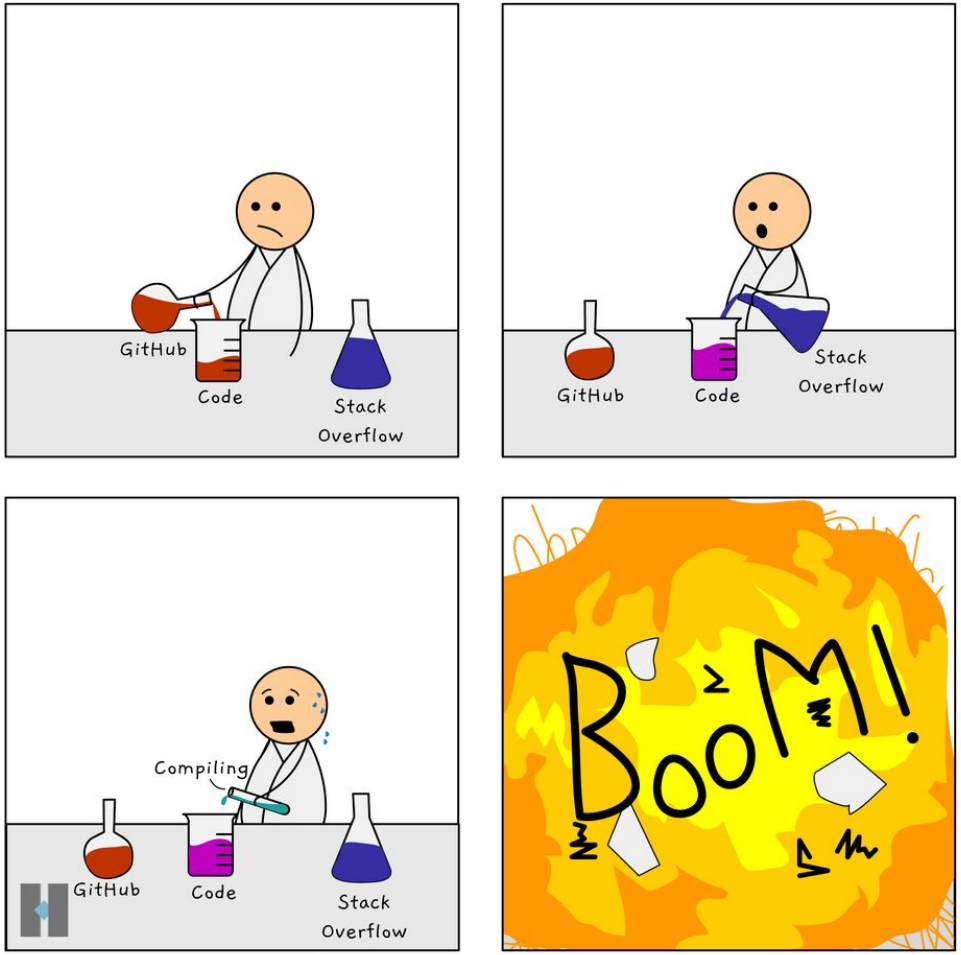
Bug #416

Bug #417

Bug #418

Bug #419

Bug #420



Why is Software Testing Important?

MONEY

DANGER

History Examples

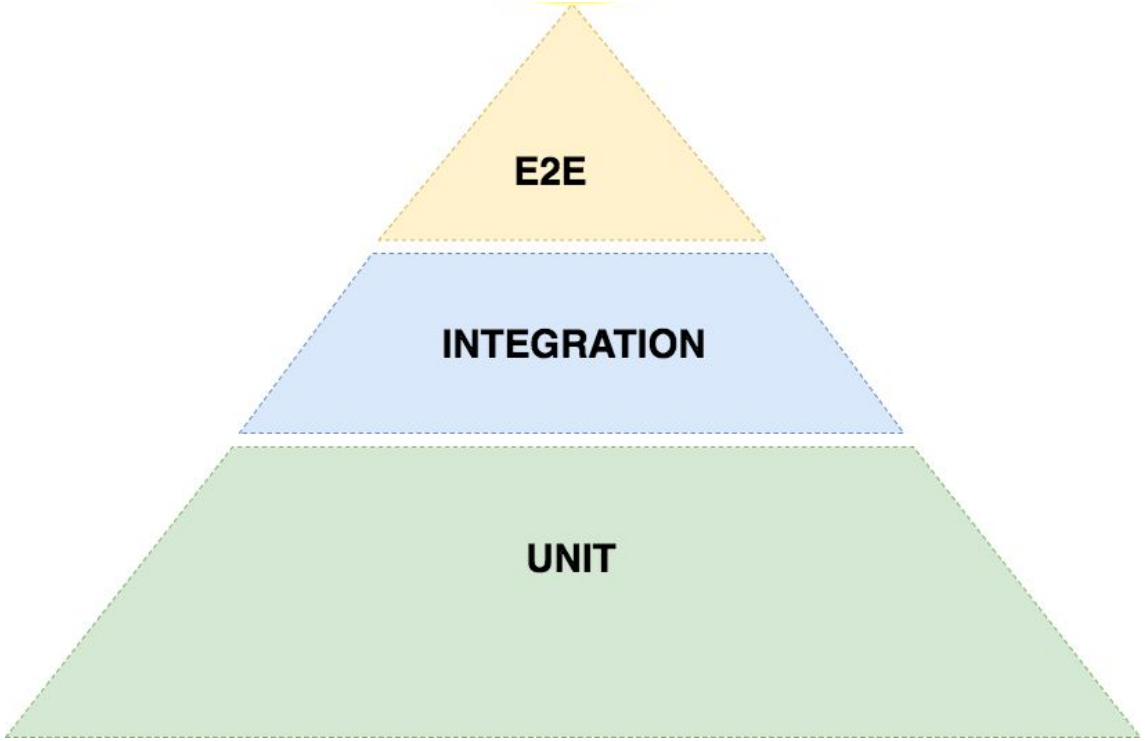
- In may of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with **920 million US dollars**.

- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving **3 people dead** and critically injuring 3 others.

Testing types

Functional Testing	Non-Functional Testing	Maintenance
Test each function providing appropriate input and verifying the output against the Functional requirements.	Checks non-functional aspects such as performance, usability, reliability, etc, of software application.	Maintenance of the software and correcting errors that may occur during its use.

Testing Pyramid



Testing Pyramid

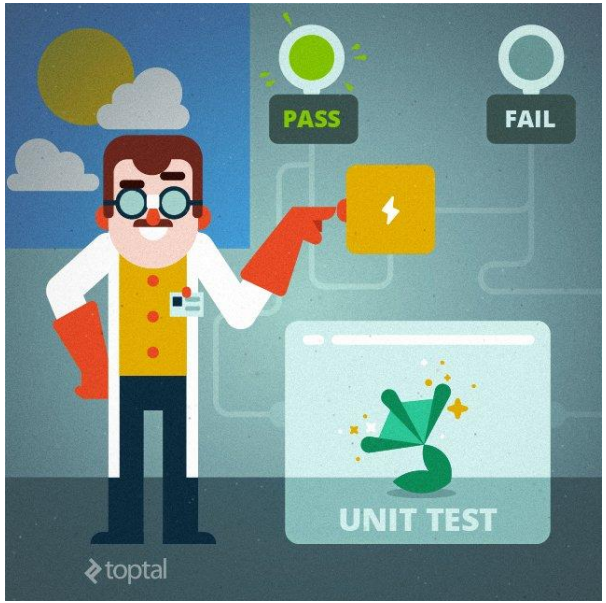
Unit tests: Test individual units of code in isolation.

Integration tests: Test the integrations between different units.

E2E: Test the system as a whole, from the UI down to the data store, and back.

Methodology

- Posteriori
- Priori



Unit Test a Posteriori

It consists of the following steps

1. Write a functionality code
2. Write and run test

Unit Test a Posteriori

It is a structured and automated way to prove this

```
const add = function(valueOne, valueTwo) {
    const suma = valueOne + valueTwo;
    return suma;
};
```

Unit Test a Posteriori

It is a structured and automated way to prove this

```
const add = function(valueOne, valueTwo) {
   const suma = valueOne + valueTwo;
   return suma;
};

if (add(1, 2) !== 3) {
   throw new Error('Failed');
}
```

TDD: Unit Test a Priori

Test-driven development is mostly a <u>loop</u> of the following steps:

1. Write test
2. Run test... **Fail**
3. Write a minimum amount of code for the test to work
4. Run test... **Pass**

TDD: Unit Test a Priori

```
if (add(1, 2) !== 3) {
   throw new Error('Failed');
}
```

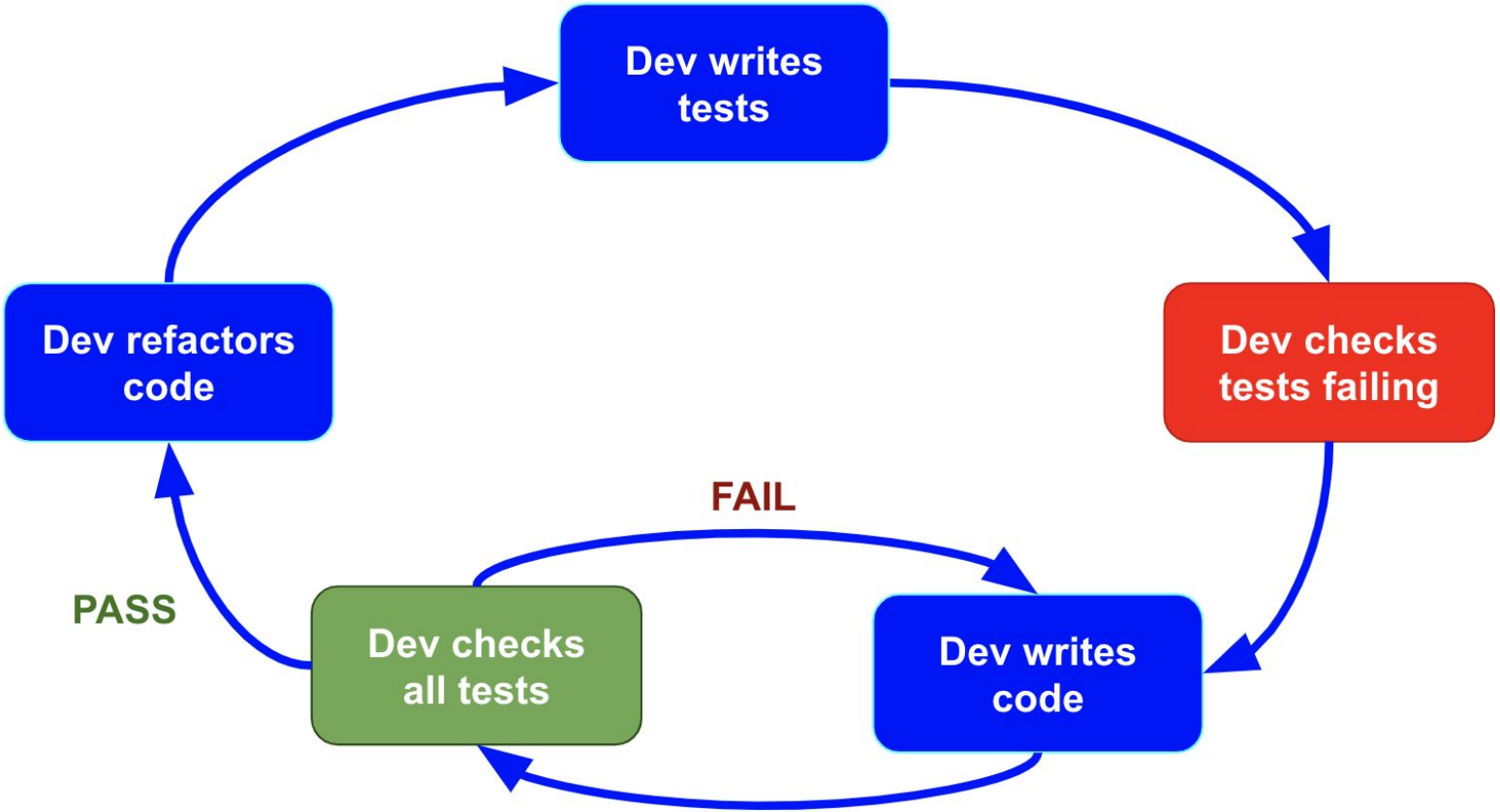
TDD: Unit Test a Priori

```
const add = function(valueOne, valueTwo) {
    return 3;
};


if (add(1, 2) !== 3) {
    throw new Error('Failed');
}
```

TDD: Unit Test a Priori

```
const add = function(valueOne, valueTwo) {
    const suma = valueOne + valueTwo;
    return suma;
};
```


Dev writes tests
Dev checks tests failing
Dev writes code
Dev checks all tests
Dev refactors code
FAIL
PASS

Popular Testing Frameworks for JavaScript

Jest

Mocha and Chai

Mocha:

- JS Framework.
- Runs on Node.js and browsers.
- **Allows**:
  - Asynchronous testing.
  - Test coverage reports.
  - Use of any assertion library.

Mocha and Chai

Chai:

- BDD / TDD assertion library for Node.js and Browsers
- It has several interfaces:
  - assert
  - expect
  - should

Mocha and Chai

Chai:

```
const EXAMPLE = 'matchByMatch';

assert.equal(EXAMPLE, 'matchByMatch');
expect(EXAMPLE).to.equal('matchByMatch');
EXAMPLE.should.equal('matchByMatch');
```

Although it has a different structure the same results are obtained.

Mocha and Chai installation

To test code in the browser run on terminal:

```
npm install mocha chai --save-dev
```

To test Node.js code, in addition to the above, run:

```
npm install -g mocha
```

How can I test?

The structure of the unit tests is formed by **6** different elements:

1. **describe**
2. **it**
3. **before**
4. **beforeEach**
5. **after**
6. **afterEach**

How can I test?

1. **describe:** It's used to group, which you can nest as deep
2. **it:** It's the test case

Receive two parameters: A message and a function

```
describe('<message>', function() {});

it('<message>', () => {});
```

How can I test?

Scenario: **One test case**

```
it('It works on my machine', () => {
  /** Test cases */
});
```

How can I test?

Scenario: **A nested test case**

```
describe('Historical phrases', () => {
  describe('Computer engineering', () => {
    it('It works on my machine', () => {
      /** Test cases */
    });
  });
});
```

How can I test?

Scenario: **Two test cases in one test**

```
describe('Historical phrases', () => {
  it('It works on my machine', () => {
    /** Test cases */
  });
  it('Match By Match', () => {
    /** Test cases */
  });
});
```

How can I test?

3. **before**: It's a code to run before the *first* it()
4. **beforeEach**: It’s a code to run *before each* it()
5. **after**: It’s a code to run *after* it()
6. **afterEach**: It’s a code to run *after each* it()

How can I test?

```
describe('Description', () => {
  beforeEach(() => {
    console.log('Random message');
  });

  it('Test # 1', () => {});
  it('Test # 2', () => {});
});
```

How can I test?

A)

Random message

✓ Test # 1

Random message

✓ Test # 2

B)

Random message

Random message

✓ Test # 1

✓ Test # 2

C)

Random message

✓ Test # 1

✓ Test # 2

How can I test?

A)

Random message

✓ **Test # 1**

Random message

✓ **Test # 2**

B)

Random message

Random message

✓ Test # 1

✓ Test # 2

C)

Random message

✓ Test # 1

✓ Test # 2

Define “it”

```
const assert = require('chai').assert;

const add = function(valueOne, valueTwo) {
  const suma = valueOne + valueTwo;
  return suma;
};

it('add test', function() {
  assert.equal(add(2, 3), 5);
});
```

Define “it”

```
const assert = require('chai').assert;

const add = function(valueOne, valueTwo) {
  const suma = valueOne + valueTwo;
  return suma;
};

it('add test', function() {
  assert.equal(add(2, 3), 5);
});
```

Define “it”

```
const assert = require('chai').assert;

const add = function(valueOne, valueTwo) {
  const suma = valueOne + valueTwo;
  return suma;
};

it('add test', function() {
  assert.equal(add(2, 3), 5);
});
```

Define “it”

```
const assert = require('chai').assert;

const add = function(valueOne, valueTwo) {
  const suma = valueOne + valueTwo;
  return suma;
};

it('add test', function() {
  assert.equal(add(2, 3), 5);
});
```

Directory Structure

Source code → **/src/** program.js

Evidence Code → **/test/** program_test.js

Directory Structure

../src/add.js

```
exports.add = function(valueOne, valueTwo) {
    const suma = valueOne + valueTwo;
    return suma;
};
```

Directory Structure

../test/add_test.js

```
const assert = require('chai').assert;
const {add} = require('../src/add');

it('add test', function() {
  assert.equal(add(2, 3), 5);
});
```

Test Runner

- With **Mocha** and **Chai** it is possible to run the tests both in **Node.js** and in **browsers**.

- But there are some differences in the way to do it...

How to run tests in browsers?

Very important:

It’s necessary a *.html* file to run the tests.

```html
<!DOCTYPE html>
<html>
 <head>
   <title>Mocha Tests</title>
   <link rel="stylesheet" href="../node_modules/mocha/mocha.css" />
 </head>
 <body>
   <div id="mocha"></div>
   <script src="../node_modules/mocha/mocha.js"></script>
   <script src="../node_modules/chai/chai.js"></script>
   <script>
     mocha.setup("bdd");
   </script>

   <!-- load code you want to test here -->
   <script src="src/add.js"></script>

   <!-- load your test files here -->
   <script src="test/add-browser-test.js"></script>

   <script>
     mocha.run();
   </script>
 </body>
</html>
```

How to run tests in node.js?

It is necessary to follow the following steps:

1. Create a working directory to develop the tests.
2. Change to that directory.
3. Execute: `npm init` to initialize a new package.
4. Next, we run: `npm install` to install local dependencies.

How to run tests in node.js?

5. In the **package.json** file, we must change the "test" line to:

```
"test": "mocha"
```

6. Run the command:

```
npm test
```

How to run tests in node.js?

7. For run a specific file:

```
npm test <file path>
```

8. For run once and get continuous monitoring:

```
npm run test:watch
```

How to run tests in node.js?

Very important syntax tip:

In the file with the code to be tested, it is necessary to use the "**exports**" directive as shown on the next slide.

```
module.exports = {
 addClass: function(el, newClass) {
   if (el.className.indexOf(newClass) !==
-1) {
     return;
   }
   if (el.className !== "") {
     // Ensure class names are separated by
a space
     newClass = " " + newClass;
   }

   el.className += newClass;
 }
};
```

```
function addClass(el, newClass) {
 if (el.className.indexOf(newClass) !== -1)
{
   return;
 }

 if (el.className !== "") {
   // Ensure class names are separated by a
space
   newClass = " " + newClass;
 }

 el.className += newClass;
}
```

Exclusive Tests: Only

Execute only the block or the test case to which we add it.

```
it.only('If you can dream it, you can do it', () => {});
```

Exclusive Tests: Only

```
describe('Description', () => {
  it('Test # 1', () => {});
  it.only('Test # 2', () => {});
  it('Test # 3', () => {});
                .
                .
                .
  it('Test # n', () => {});
});
```

Exclusive Tests: Only

```
describe('Description', () => {
  it('Test # 1', () => {});
  it.only('Test # 2', () => {});
  it('Test # 3', () => {});
                .
                .
                .
  it('Test # n', () => {});
});
```

Inclusive Tests: Skip

Don’t execute the block or test case to which we add it.

```
it.skip('Think, believe, dream and dare', () => {});
```

Exclusive Tests: Skip

```
describe('Description', () => {
  it('Test # 1', () => {});
  it.skip('Test # 2', () => {});
  it('Test # 3', () => {});
                .
                .
                .
  it('Test # n', () => {});
});
```

Exclusive Tests: Skip

```
describe('Description', () => {
  it('Test # 1', () => {});
  it.skip('Test # 2', () => {});
  it('Test # 3', () => {});
                .
                .
                .
  it('Test # n', () => {});
});
```

Bad Practice

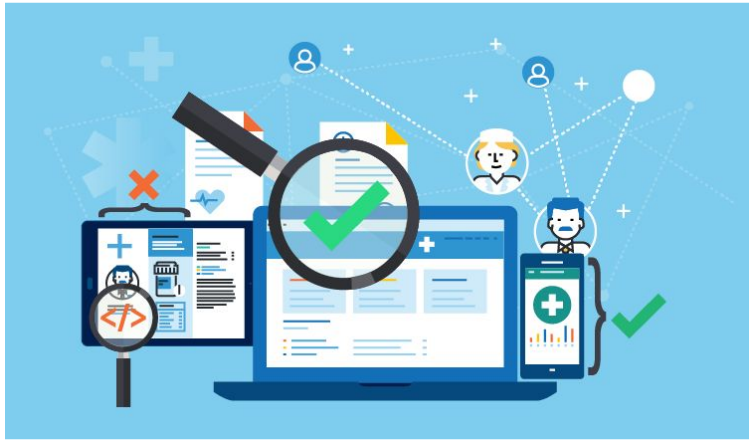
```
it('try absolutely everything', function() {
  assert.equal(add(2, 3), 5);
  assert.equal(sub(8, 2), 6);
  assert.equal(mult(4, 2), 8);
  assert.equal(div(10, 2), 5);
});
```

Good Practice

```
describe('Basic operations', () => {
  it.only('test add', function() {
    assert.equal(add(2, 3), 5);
  });
  it.only('test sub', function() {
    assert.equal(sub(8, 2), 6);
  });
});
```

Conclusion

It is **necessary** to use tests since it avoids problems and corrects functionality errors



We also make sure that each unit works properly as the product develops

Conclusion

Advantage:

- Identify errors in the development phase.
- Have a good quality software.
- We minimize maintenance and development costs
- We guarantee a software is reliable.

“There are two ways to write error-free programs; only the third one works.”

Alan J. Perlis

Bibliography

GitHub repository

1. Non Functional Testing
2. Functional Testing
3. Software Testing
4. Testing Frameworks
5. Testing In JS
6. Unit Test with Mocha and Chai
7. Test JS Mocha and Chai
8. Mocha and Chai

Thanks for your attention!

Any questions?

Miguel Ordoñez Morales alu0101281087@ull.edu.es
Basilio Gómez Navarro alu0101049151@ull.edu.es