Universidad de La Laguna

Test Driven Development and Code Coverage

Eduardo Suárez Ojeda
alu0100896565@ull.edu.es

Florentín Pérez González
alu0101100654@ull.edu.es

Universidad de La Laguna

Test Driven Development

Universidad de La Laguna

Topics to talk about

Test Driven Development

- What is TDD
- Why you should use TDD
- RED-GREEN-REFACTOR
- Implementation
- TDD and BDD
- TDD limitations

What is it?

It’s a Software Development Process.

Iterative and incremental development.

Good practices to write code.

Before TDD

TDD vs No TDD

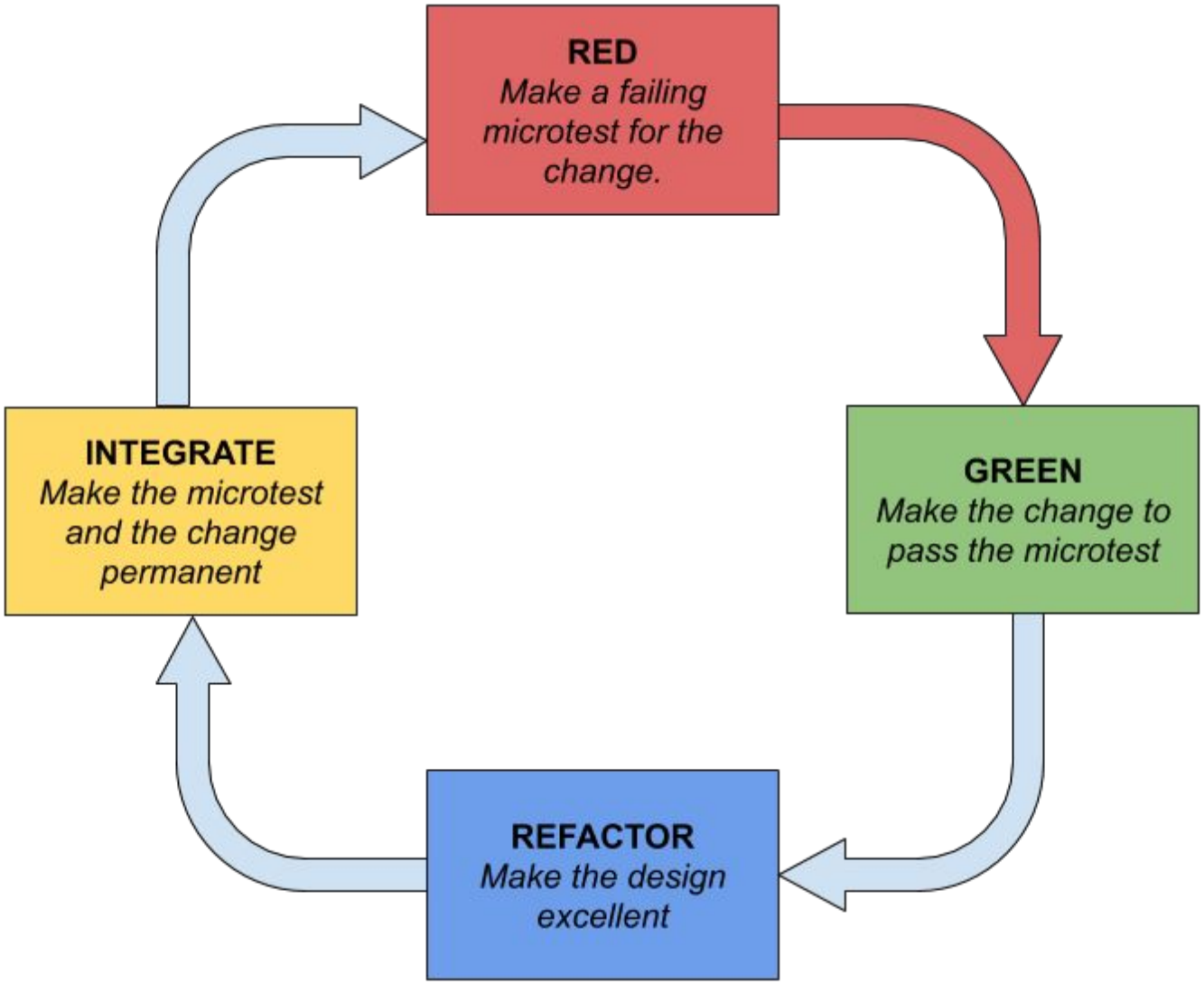
TDD pros

- Easy to change code.
- Robust and safe.
- Easy to maintain.
- Faster Development speed (seriously).

3 Rules of TDD

- You will not write code without first writing a failed test.
- You will not write more than one unit test enough to fail.
- You will not write more code than necessary to pass the test

Iterative and Incremental



RED

```
Failures:

  1) Player increments player goal tally by 1
     Failure/Error: expect(player.goals).to eq 1

       expected: 1
            got: 0

       (compared using ==)
     # ./spec/player_spec.rb:9:in `block (2 levels) in <top (required)>'

Finished in 0.02291 seconds (files took 0.09492 seconds to load)
1 example, 1 failure
```

GREEN

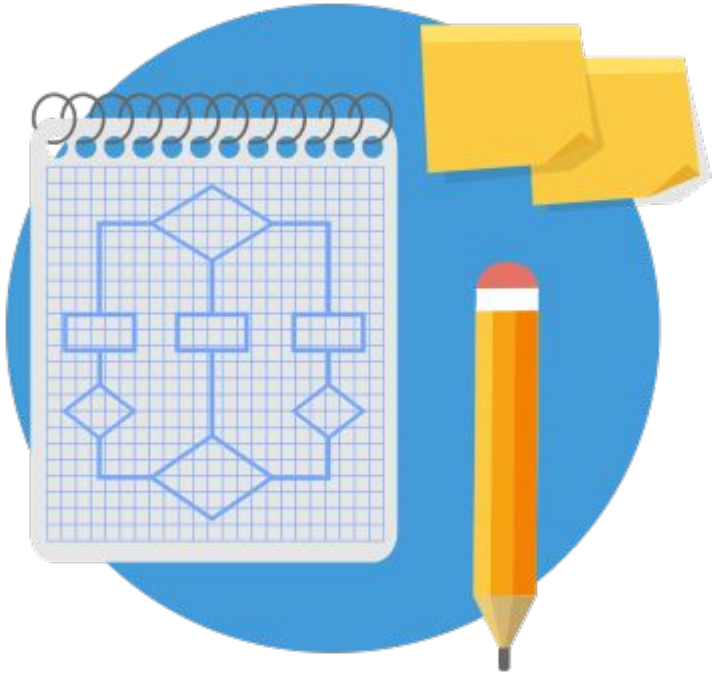
```
Player
  increments player goal tally by 1

Finished in 0.00089 seconds (files took 0.10744 seconds to load)
1 example, 0 failures
```

REFACTOR

INTEGRATE

TDD as design tool

- Choose a Requirement.
- Write a failing test.
- Write the minimum code to pass the test.
- Run tests.
- Refactor.
- Update the list of requirements.



TESTS

Containers: describe, context, it...

Allocation methods: beforeAll, beforeEach, afterAll, afterEach...

Test Functions: assert, expect...

Check types: Equal, not equal, to be true...

TDD IMPLEMENTATION

FALSE IMPLEMENTATION

Create the function and return the correct value.

Step by step progress.

Increases motivation to see that the test passes.

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144

Sucesión de Fibonacci

TRIANGULAR IMPLEMENTATION

Choose the simplest case.

Apply Red-Green-Refactor.

Repeat until every case is covered.

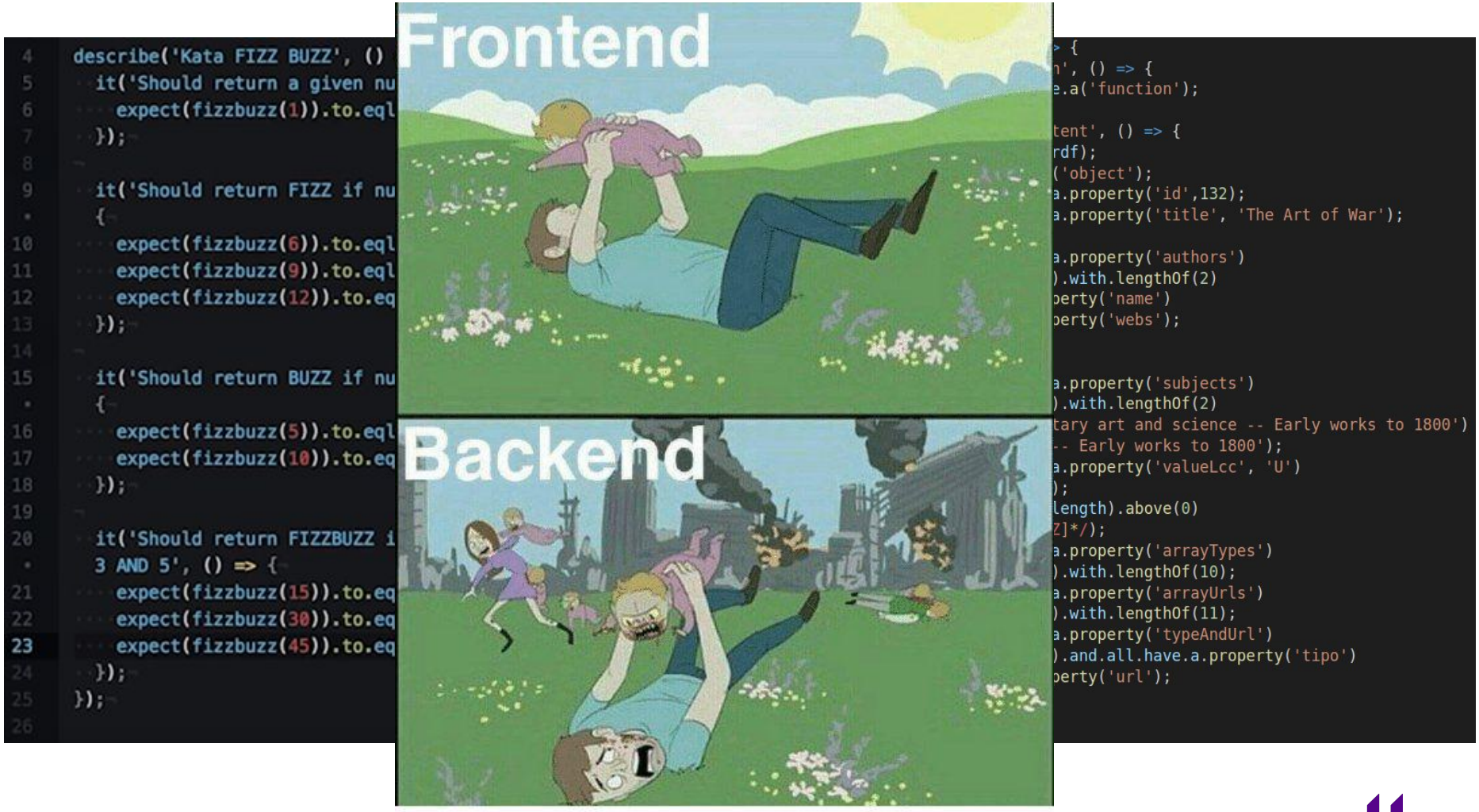
OBVIOUS IMPLEMENTATION

When the solution seems easy.

Write the obvious implementation in the first iteration.

Can cause problems when the solution was not as obvious.

TDD and BDD


describe('Kata FIZZ BUZZ', ()
it('Should return a given nu
expect(fizzbuzz(1)).to.eql
});
it('Should return FIZZ if nu
{
expect(fizzbuzz(6)).to.eql
expect(fizzbuzz(9)).to.eql
expect(fizzbuzz(12)).to.eq
});
it('Should return BUZZ if nu
{
expect(fizzbuzz(5)).to.eql
expect(fizzbuzz(10)).to.eq
});
it('Should return FIZZBUZZ i
3 AND 5', () => {
expect(fizzbuzz(15)).to.eq
expect(fizzbuzz(30)).to.eq
expect(fizzbuzz(45)).to.eq
});
});
Frontend
Backend
> {
', () => {
e.a('function');
tent', () => {
rdf);
('object');
a.property('id',132);
a.property('title', 'The Art of War');
a.property('authors')
).with.lengthOf(2)
perty('name')
perty('webs');
a.property('subjects')
).with.lengthOf(2)
tary art and science -- Early works to 1800')
-- Early works to 1800');
a.property('valueLcc', 'U')
);
length).above(0)
Z]*/);
a.property('arrayTypes')
).with.lengthOf(10);
a.property('arrayUrls')
).with.lengthOf(11);
a.property('typeAndUrl')
).and.all.have.a.property('tipo')
perty('url');

TDD LIMITATIONS

This is not an infallible technique.

It is not valid for all projects.

Not very useful for frontend programming.

You have to change tests when you change the implementation.

Universidad de La Laguna

CODE COVERAGE

Universidad de La Laguna

Topics to talk about

Code Coverage

- Basic Criterias
- Not Basic Criterias
- Code Coverage in JS

Codecov

- Requirements
- Live example

What is it?

It’s basically a measure used to check how much of our code is checked by our tests.

This measure is **ALWAYS** a percentage.

Ok, a percentage but… of what?

Well, it depends of different criterias.
The most usual are:

- Function coverage.
- Statement coverage.
- Branch coverage.
- Condition coverage.

Let's see all of them.

Function coverage

It focuses in the differents subroutines/functions called in our program.

```
1 function myFirstFunction () {
2   return 1;
3 }
4 function mySecondFunction () {
5   return 2;
6 }
7 const myVariable = myFirstFunction();
```

What percentage should be get?

Function coverage

It focuses in the differents subroutines/functions called in our program.

```
1 function myFirstFunction () {
2   return 1;
3 }
4 function mySecondFunction () {
5   return 2;
6 }
7 const myVariable = myFirstFunction();
```

What percentage should be get? **50 %**

Statement coverage

It focuses in the differents statements that are executed in our program.

```
1  const loopLimit = 5;
2  let myVariable = 0;
3  for (let i = 0; i < loopLimit; i++) {
4    myVariable += 2;
5  }
6  if (myVariable % 2 === 1) {
7    const otherVariable = myVariable / 2;
8    console.log(myVariable - otherVariable);
9    console.log(otherVariable - myVariable);
10 }
```

What percentage should be get?

Statement coverage

It focuses in the differents statements that are executed in our program.

```
1  const loopLimit = 5;
2  let myVariable = 0;
3  for (let i = 0; i < loopLimit; i++) {
4    myVariable += 2;
5  }
6  if (myVariable % 2 === 1) {
7    const otherVariable = myVariable / 2;
8    console.log(myVariable - otherVariable);
9    console.log(otherVariable - myVariable);
10 }
```

What percentage should be get? **50 %** (too)

Branch coverage

It focuses in the differents branch of conditional statements that are executed.

```
1  function myFunction(firstValue, secondValue) {
2    if (firstValue > 5) {
3      return 1;
4    } else if (secondValue > 5) {
5      return -1;
6    } else {
7      return 0;
8    }
9  }
10 myFunction(3, 2);
```

What percentage should be get?

Branch coverage

It focuses in the differents branch of conditional statements that are executed.

```
1  function myFunction(firstValue, secondValue) {
2    if (firstValue > 5) {
3      return 1;
4    } else if (secondValue > 5) {
5      return -1;
6    } else {
7      return 0;
8    }
9  }
10 myFunction(3, 2);
```

What percentage should be get? **33%**

Condition coverage

It focuses in the differents boolean sub-expressions of our program.

```
1 function myFunction(firstValue, secondValue) {
2   if ((firstValue > 5) && (secondValue > 5)) {
3     return 1;
4   } else {
5     return -1;
6   }
7 }
8 myFunction(5, 3);
```

What percentage should be get?

Condition coverage

It focuses in the differents boolean sub-expressions of our program.

```
1 function myFunction(firstValue, secondValue) {
2   if ((firstValue > 5) && (secondValue > 5)) {
3     return 1;
4   } else {
5     return -1;
6   }
7 }
8 myFunction(5, 3);
```

What percentage should be get? **50 %**

Let’s talk about the last one a little more

It’s similar to **branch coverage**, but they are not the same.

```
if ((firstValue > 5) && (secondValue > 5)) {
```

This is a condition

And this too

It checks that **every** condition has been evaluated to true and false.

Let’s talk about the last one a little more

So for this particular example we just need to evaluate the expression when both sub-expression are **true** and **false.**

```
if ((firstValue > 5) && (secondValue > 5)) {
```

In other words:

```
1 if ((false) && (true)) {}
2 if ((true) && (false)) {}
```

Let’s talk about the last one a little more

IMPORTANT

```
1 if ((false) && (false)) {}
2 if ((true) && (true)) {}
```

They are valid but **not necessary.**

The meaning of the percentage

- **A low percentage:** More chances of undetected software bugs.
- **A high percentage:** Lower chances of undetected software bugs.

IMPORTANT: Code coverage is not directly indicative of code quality.

What is a low/high percentage?

Well, it depends of the situation and the software that is been testing.

In general

X < 60	60<=X<85	X >= 85
This is terrible.	Not bad but neither good.	A good result.

How can I improve the result?

- Basically, you have to modify your current tests or, add new ones.

- In some cases, you will have to change your current code.

How can I improve the result?

- **Function coverage**: Use as much functions you can in your tests.
- **Branch coverage**: Make sure you test all the possible options.
- **Condition coverage**: Try to check all boolean sub-sequences possibilities.
- **Statement coverage**: Make sure you don’t have useless code.

Not basic coverage criteria

There are other measure criteria that, despite being not so common, they can be useful for your purpose.

Some of them:

- Multiple condition coverage.
- Parameter value coverage.

Multiple condition coverage

Not difficult but neither too useful.

```
if (var1 && var2 && var3){}
```

You have to check the condition for **ALL** the possibilities.

```
var1 = false, var2 = false, var3 = false;
var1 = false, var2 = false, var3 = true;
var1 = false, var2 = true, var3 = false;
var1 = false, var2 = true, var3 = true;
var1 = true, var2 = false, var3 = false;
var1 = true, var2 = false, var3 = true;
var1 = true, var2 = true, var3 = false;
var1 = true, var2 = true, var3 = true;
```

Parameter value coverage

Applied to function calls that need, **at least**, one parameter.

We have to test those functions with all the **common values** for their parameters.

```
function myFunction (thisIsAString) {
  return thisIsAString;
}
```

What are the common values for that parameter?

Parameter value coverage

```
function myFunction (thisIsAString) {
  return thisIsAString;
}
```

- Null/Undefined.
- Empty.
- Whitespace.
- Valid string.
- Invalid string (if there were).
- Single letter string.
- Long string.

Other coverage criteria

There are maaaaaaany more, but we are not going to talk about them today.

- Path Coverage.
- Entry/exit coverage.
- Loop coverage (This one is interesting).
- State coverage (Very uncommon).
- Data-flow coverage (very uncommon).

Code coverage in JS

Basically we need to install some valid **coverage tool.**
Some of them:

- Istanbul (Our recommendation)
- Blanket.js
- jscoverage

But that’s not all...

You will need, apart from the coverage tool, a **test framework**.
Some of them:

- Mocha
- Karma

It’s important to check that both tools are compatible.

Assertions library

You can download and use an assertion library, as Chai, if you want. They do not affect in any way to the coverage tool.

Code coverage with Istanbul

Let’s see an example of code coverage in Javascript using **Istanbul** as coverage tool and **Mocha** as the test framework.

Code coverage with Istanbul

1) First of all, we have to **download and install** our tools.

Mocha:

```
npm install --global mocha
npm install --save-dev mocha
```

Istanbul:

```
npm install --save-dev nyc
```

Code coverage with Istanbul

2) Configure mocha as you want.

3) Add support to Istambul using babel-plugin-istanbul.

```
npm install --save-dev babel-plugin-istanbul
```

Add this to your **.babelrc** file.

```
{
  "env": {
    "test": {
      "plugins": [ "istanbul" ]
    }
  }
}
```

Code coverage with Istanbul

4) Add the command **nyc** in front of your existing test command in your **package.json.**

```
{
  "scripts": {
    "test": "nyc mocha"
  }
}
```

Code coverage with Istanbul

5) Add a this line to your **package.json** scripts section:

```
{
  "scripts": {
    "coverage": "nyc npm run test"
  }
}
```

6) Write your code.

7) Write your test and place them in your **test** folder.

Code coverage with Istanbul

8) Execute your test using **npm test**.

You should get something similar to this.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	50	0	0	50	
addition.js	50	100	0	50	2
division.js	50	100	0	50	2
example.js	50	0	0	50	12-19,21
multiplication.js	50	100	0	50	2
subtraction.js	50	100	0	50	2

Code coverage with Istanbul

Those are the **code coverage** criterias we were talking before.

Istanbul will show information for every file involved in your test, except the test file itself.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	50	0	0	50	
addition.js	50	100	0	50	2
division.js	50	100	0	50	2
example.js	50	0	0	50	12-19,21
multiplication.js	50	100	0	50	2
subtraction.js	50	100	0	50	2

Code coverage with Istanbul

What means each column?

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	50	0	0	50	
addition.js	50	100	0	50	2
division.js	50	100	0	50	2
example.js	50	0	0	50	12-19,21
multiplication.js	50	100	0	50	2
subtraction.js	50	100	0	50	2

In order:

- Files involved.
- Statement coverage.
- Branch coverage.
- Function coverage
- Line coverage.
- Uncovered lines for each file.

Code coverage with Istanbul

But that is not all. Istanbul allow you to use a collection of reporters that change the display you get, in both, format and content.

How? Well, do you remember the **last line** we add to our **package.json**.?

Code coverage with Istanbul

That line allows you to specify the reporter you want to use.

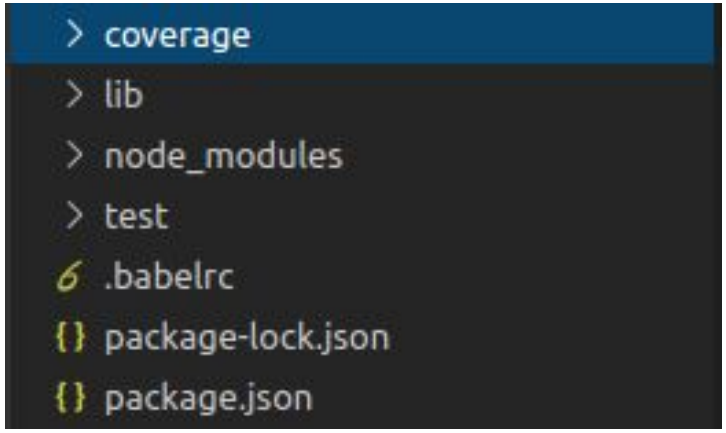
```
{
  "scripts": {
    "coverage": "nyc npm run test"
  }
}
```

Let's change it for this one now.

```
{
  "scripts": {
    "coverage": "nyc --reporter html npm test"
  }
}
```

Code coverage with Istanbul

If we use **npm coverage** instead of **npm test** we will see that a **new folder** (coverage)is created.



Code coverage with Istanbul

In that folder, you will see an **HTML report** instead of the previous one.

All files

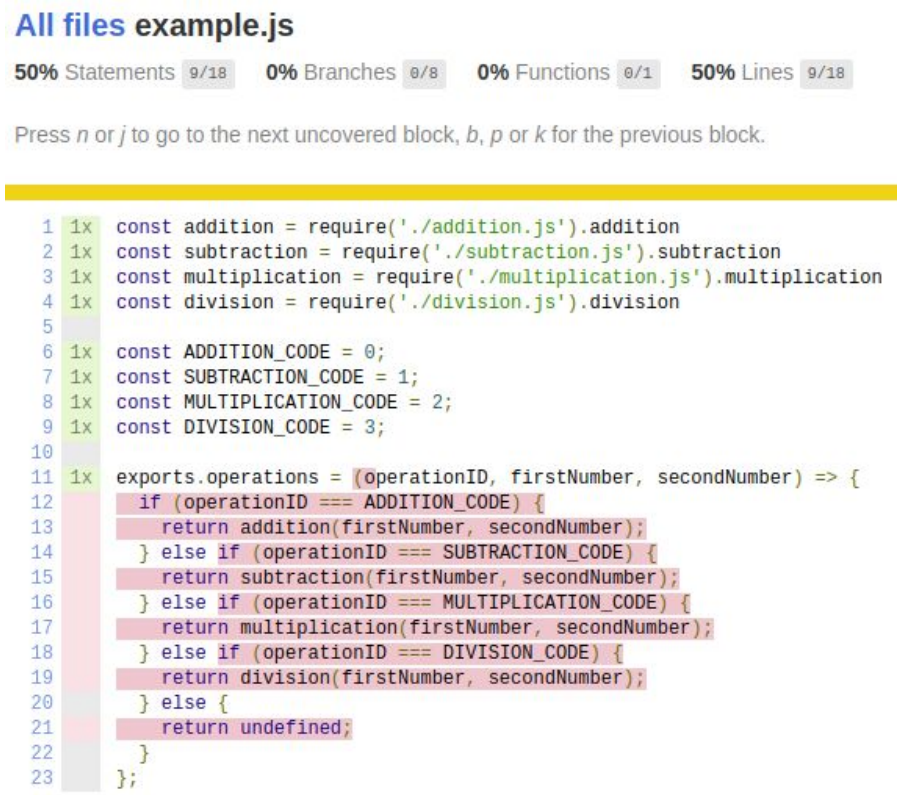
50% Statements `13/26` **0%** Branches `0/8` **0%** Functions `0/5` **50%** Lines `13/26`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
addition.js		50%	1/2	100%	0/0	0%	0/1	50%	1/2
division.js		50%	1/2	100%	0/0	0%	0/1	50%	1/2
example.js		50%	9/18	0%	0/8	0%	0/1	50%	9/18
multiplication.js		50%	1/2	100%	0/0	0%	0/1	50%	1/2
subtraction.js		50%	1/2	100%	0/0	0%	0/1	50%	1/2

Code coverage with Istanbul

You can even see in your browser what lines were the uncovered ones!!



All files example.js

50% Statements 9/18 0% Branches 0/8 0% Functions 0/1 50% Lines 9/18

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1  1x  const addition = require('./addition.js').addition
2  1x  const subtraction = require('./subtraction.js').subtraction
3  1x  const multiplication = require('./multiplication.js').multiplication
4  1x  const division = require('./division.js').division
5
6  1x  const ADDITION_CODE = 0;
7  1x  const SUBTRACTION_CODE = 1;
8  1x  const MULTIPLICATION_CODE = 2;
9  1x  const DIVISION_CODE = 3;
10
11 1x  exports.operations = (operationID, firstNumber, secondNumber) => {
12       if (operationID === ADDITION_CODE) {
13         return addition(firstNumber, secondNumber);
14       } else if (operationID === SUBTRACTION_CODE) {
15         return subtraction(firstNumber, secondNumber);
16       } else if (operationID === MULTIPLICATION_CODE) {
17         return multiplication(firstNumber, secondNumber);
18       } else if (operationID === DIVISION_CODE) {
19         return division(firstNumber, secondNumber);
20       } else {
21         return undefined;
22       }
23     };
```

Code coverage with Istanbul

Other reporters available:

- JSON format (json-summary)
- XML (clover and covertura too)
- text-summary (text-sumary)

Consult the documentation to learn more.

Code coverage with Istanbul

Something more about the reporters.

```
{
    "scripts": {
    "test": "mocha",
    "coverage": "nyc --reporter html --reporter text npm test"
  },
}
```

You can use more than one at the same time, and you can remove the **nyc** in you test command if you want.

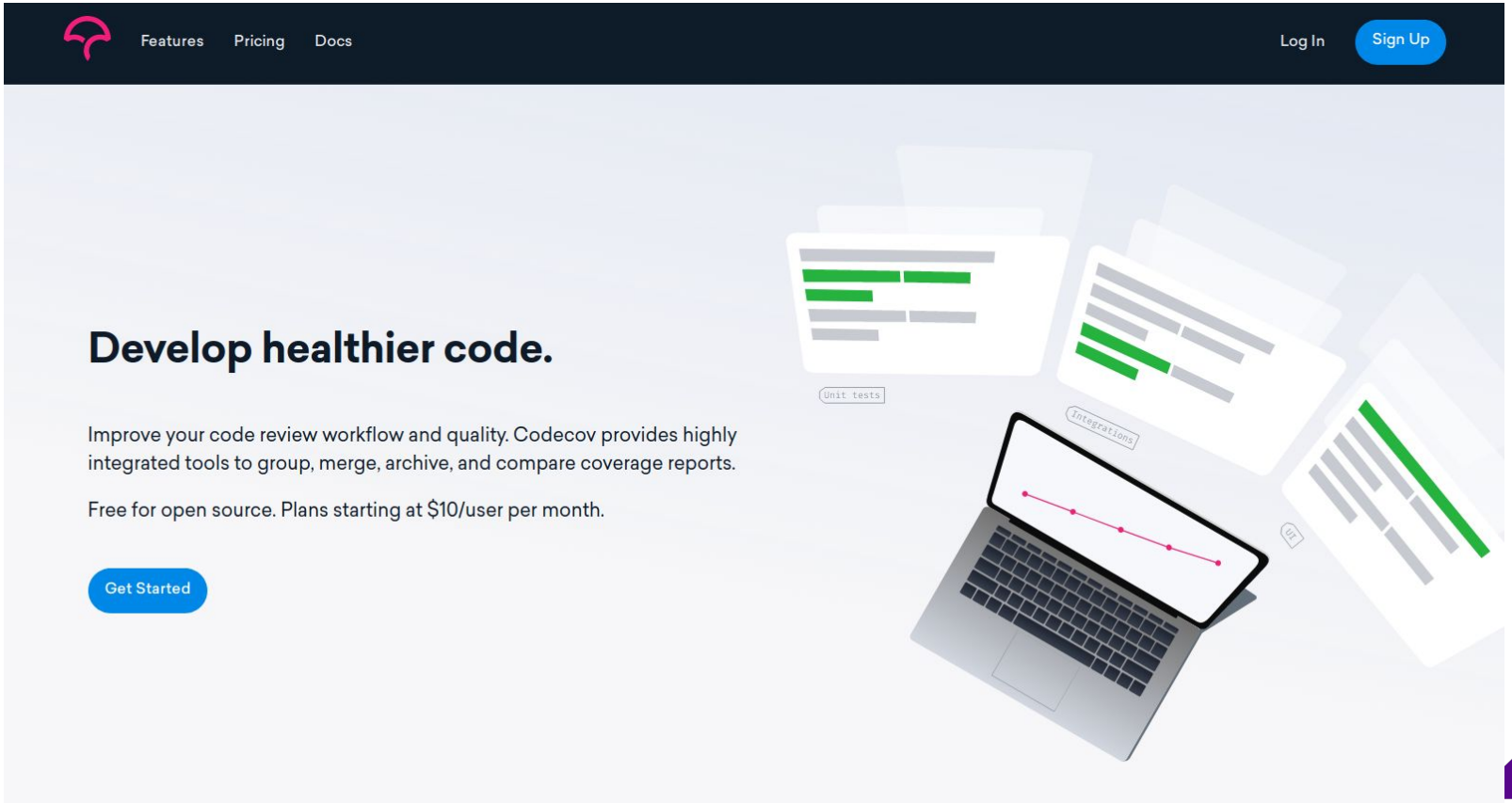
Codecov

We can compare our coverage reports thanks to **codecov**.

We need the next to use codecov properly.

- A GitHub account.
- Coverage reports.
- A continuous integration provider (not needed but recommended).

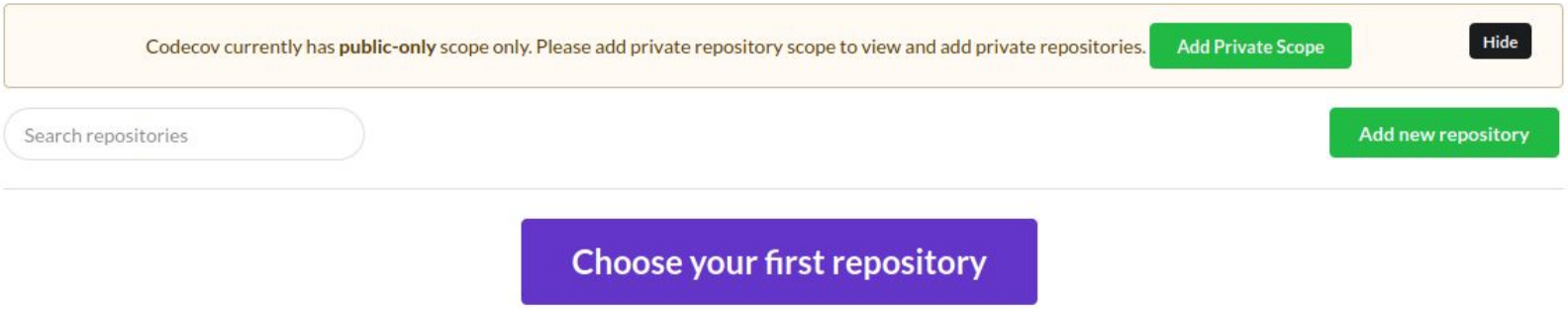
This is the main page of **codecov**.



Codecov

Once you logged in with GitHub, you will see a list of your **activated** repositories.

Of course, the first time it will be empty, so you will have to add one.



Codecov

Codecov will give you a token for your repository. This token will allow Codecov to collect the coverage reports.

With the token, you have to:

- Set it in you CI environment variables.
- Create a **codecov.yml** file and add this lines.

```
codecov:
  token: <YourToken>
```

Codecov

Don’t forget to add this ones too.

```
script:
  bash <(curl -s https://codecov.io/bash)
```

Codecov

There are certains CI services that does not need the token to work with Codecov.

They are displayed in a message under your token.

The next slides will show how to use codecov without a CI environment.

Codecov

We need to install the **codecov package** for NodeJS.

```
npm install -g codecov
```

Then we have to add a new line to our **package.json.**

Codecov

You have to use a **text based reporter**.

```
{
  "scripts": {
    "report-coverage": "nyc report --reporter lcovonly > coverage.lcov &&
      codecov"
  },
}
```

Codecov

Now whenever you want to upload your coverage reports, you want first to generate then and after that, use:

npm run report-coverage

After some time, you will see in the codecov webpage for your repository, the report you sent.

Codecov

You should **always upload first** your commit to GitHub and then, send the coverage reports.

Codecov will allow you to:

- See graphics.
- Compare coverage reports.
- See changes realized.
- Compare coverage by branch.

BIBLIOGRAPHY

TDD (Test Driven Development) en JavaScript

TDD Vs BDD - Analyze The Differences With Examples

Test-driven development

dwyl/learn-tdd: A brief introduction to Test Driven Development (TDD) in JavaScript (Complete Beginner's Step-by-Step Tutorial)

BIBLIOGRAPHY

About the Codecov yaml

Code coverage

babel-plugin-module-resolver

codecov/example-node: Example repo for uploading reports to Codecov

Codecov Bash uploader

dwyl/learn-istanbul: Learn how to use the Istanbul JavaScript Code Coverage Tool

Mocha and Istanbul in 5 minutes

BIBLIOGRAPHY

istanbuljs/nyc: the Istanbul command line interface

https://github.com/gotwarlost/istanbul

Istanbul, a JavaScript test coverage tool.


Universidad
de La Laguna