Javascript Testing Frameworks and Tools (Jasmin, Jest, Mocha, Tape, Cypress, Playwright)

Unit & Regression Testing, Code Coverage

Java & Python Testing Frameworks and Tools (JUnit5, Hamcrest, TestNG, PyTest, ..)

Azat Satklyčov
azats@seznam.cz,
http://sahet.net,
https://github.com/azatsatklichov/java-and-ts-tests

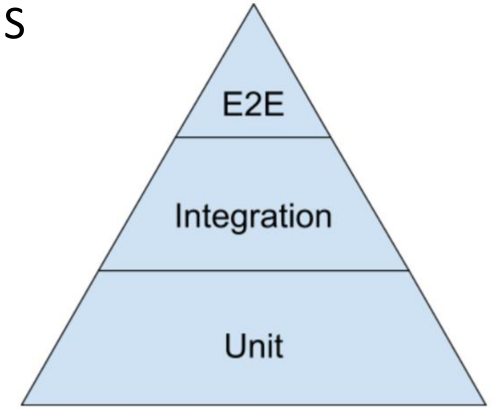
Agenda

- ❑ Types of Javascript Testing Frameworks & Tools
- ❑ Most Used Testing Tools
- ❑ Functional Testing Tools
- ❑ Jasmin
- ❑ Jest
- ❑ Mocha + Chai + Sinon
- ❑ Choose Your Unit and Integration Tests Framework
- ❑ Tape
- ❑ Choose Your Functional Tests (AAT) Framework
- ❑ Cypress
- ❑ Playwright
- ❑ Pyramid of Clean Code
- ❑ Testing Types
- ❑ Unit & Regression Testing, Code Coverage
- ❑ Java & Python Testing Frameworks & Tools

Javascript Testing via Java

- **Rhino** is a JavaScript engine written fully in Java and managed by the Mozilla Foundation, started at Netscape in 1997
- 2011 **Nashorn** is a JavaScript engine, 2014 part of Java 8 (Rhino in Java7 replaced)
- 2018, Java 11, Nashorn is deprecated, and has been removed from JDK 15 onwards, use Graal JS

JS Tests, Testing Frameworks and Testing Tools



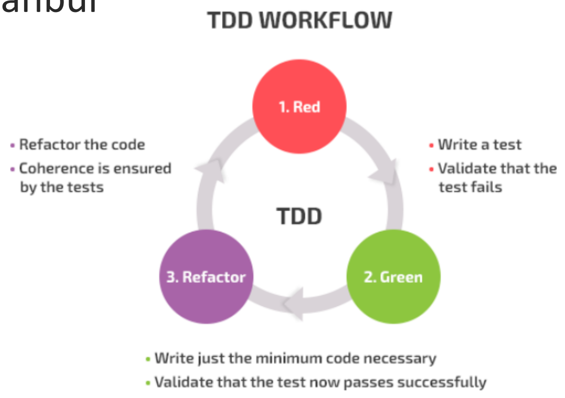
Types of Tests: Unit Tests, Integration Tests, E2E Tests **Running:** Browser, Headless, NodeJS

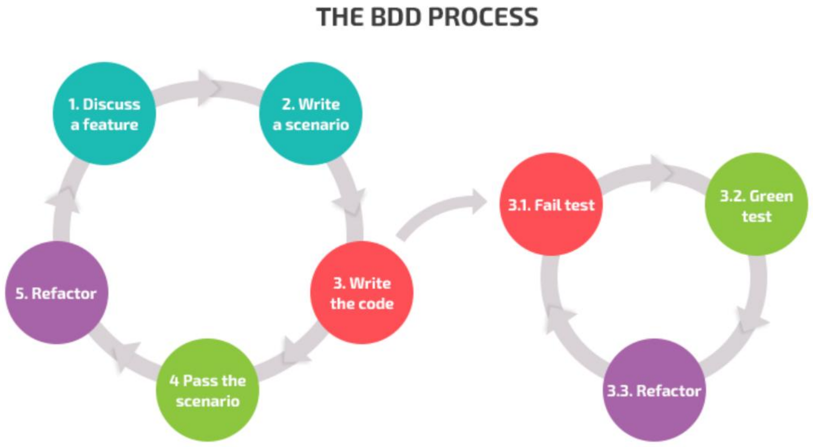
Test launchers(runners): Karma, Jasmine, Jest, TestCafe, Cypress, webdriverio
Testing structure providers: **Mocha**, **Jasmine**, **Jest**, Cucumber, TestCafe, Cypress,
Assertion functions: Chai, **Jasmine**, **Jest**, Unexpected, TestCafe, Cypress, Assert.js, Should.js
Mocks, spies, and stubs: Sinon, **Jasmine**, enzyme, **Jest**, Vitest, testdouble,
Generate and compare snapshots: **Jest**, Ava
Generate code coverage: Istanbul, **Jest**, Blanket
Browser Controllers (crawl, structure, screenshot, form-sub.): Nightwatch, Nightmare, Phantom, Puppeteer, TestCafe, Cypress
Visual Regression Tools: Applitools, Percy, Wraith, WebdriverCSS, Hapo, LooksSame, BackstopJS, AyeSpy, reg-suit, Differencify
Functional Testing Tools (Automated Acceptance Testing): Selenium WebDriver, Protractor, WebdriverIO, Nightwatch, Appium, TestCase, **Cypress**, **Puppeteer**, **Playwright**, PhantomJS, Nightmare, CodeceptJS
No Coding Functional Testing Tools: testim, Chromatic, Screener, Ghost Inspector

E.g. Some frameworks e.g. Jest, Vitest, Jasmine (still needs CC), TestCafe, and Cypress provide all of these out of the box.
Some provides only spec. functionality, then **combinations of tools** would be used: mocha + chai + sinon.

Most Used Testing Tools

- **jsdom** simulated browser env., tests run fast. But not everything can be simulated, e.g. can't take a screenshot..
- **Testing Library** testing utilities encourages good testing practices, helps to test UI components in a user-centric way
- **Istanbul / NY (->, es6)** tells how much of your code is covered with unit tests. **Jest/Tap** has by default Istanbul
- **Karma** hosts a test server with a special web page to run your tests in the page's environment.
- **Chai** is the most popular assertion library. It has many plugins and extensions.
- **Unexpected** is an assertion library with a slightly different syntax from Chai.
- **Enzyme** is used to render components and makes it easier to test your React Components
- **Sinon** has powerful standalone test spies, stubs and mocks for JS to work with any framework (Mocha, Tape,..).
- **testdouble** like Sinon but with better in design, philosophy, and features that could make it useful in many cases.
- **Wallaby** runs on your IDE (e.g. IntelliJ, MOCHA SIDEBAR) and runs tests, shows anything fails in real time alongside your code.
- **Cucumber** help with writing tests in BDD by dividing them between the acceptance criteria files using the **Gherkin** syntax (**Given/When/Then**/ format) and the tests that correspond to them.


TDD WORKFLOW
1. Red
• Write a test
• Validate that the test fails
TDD
2. Green
• Write just the minimum code necessary
• Validate that the test now passes successfully
3. Refactor
• Refactor the code
• Coherence is ensured by the tests



TDD is about *doing things right* and BDD is about *doing the right things*

Functional Testing Tools

Selenium WebDriver dominated the market of Functional Tests for years. ..

Protractor wraps Selenium and provides us with improved syntax and special built-in hooks for **Angular.**

WebdriverIO has its own implementation of the selenium WebDriver.

Nightwatch has its own implementation of the selenium WebDriver.

Apium provides an API similar to Selenium for testing websites on a mobile devices iOS, Android, Windows Phone

TestCafe is a great alternative to Selenium-Based tools. It was rewritten and **open-sourced** at the end of 2016.

Cypress is a direct competitor of TestCafe. Cypress.io runs itself in the browser and controls your tests from there where TestCafe runs in Node.js and controls the tests through a serialized communication with its injected script in the browser.

Puppeteer developed by **Google**. It provides a convenient Node.js API to control Chrome or **Headless Chrome.**

Playwright is a exactly like **Puppeteer**, but it is developed by **Microsoft**

PhantomJS implements the chromium engine to create a controllable Chrome-like headless browser.

Nightmare offers a very simple test syntax. Uses Electron which uses Chromium to control the browser's behavior.

Codecept like CucumberJS it provides another abstraction, different philosophy that focuses on user behavior.

Jasmine

Jasmine is a behavior-driven development (BDD) framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. Can be used to write tests for React apps. as well.

Why Use Jasmine?

- Jasmine does not depend on any other JavaScript framework.
- Jasmine does not require any DOM.
- All the syntax used in Jasmine framework is clean and obvious so that you can easily write tests.
- Jasmine is heavily influenced by Rspec (BDD testing for Ruby), JS Spec, and JSpec (Java test assertions).
- Jasmine is an open-source framework, versions available like stand-alone, ruby gem, Node.js, etc.

- **Ready-To-Go:** Comes with everything you need to start testing.
- **Globals:** Comes with all the important testing features in the global scope as well.
- **Community:** It has been on the market since 2009 and gathered a vast amount of articles, suggestions and tools that are based on it.
- **Angular:** Has widespread Angular support and it is recommended in the official Angular documentation.

Jasmine API

Jasmine Matchers: **Inbuilt** matchers (toEqual, toBe, not…, toBeTruthy, toThrow, …) and **Custom** matchers – addMatchers

Setup and Teardown: Jasmine provides the global beforeEach, afterEach, beforeAll, and afterAll functions.

Another way to share variables between a beforeEach, **it,** and afterEach is through the **this** keyword.

xdescribe, xit: Suites, blocks can be disabled(skipped) via xdescribe, xit functions respectively. Pending specs do not run, but shown

Spies: Jasmine has test double functions called spies. spyOn, createSpy, createSpyonObj,

Special matchers to interacting with spies: toHaveBeenCalled, toHaveBeenCalledTimes, toHaveBeenCalledWith

Matching with more finesse: Jasmine .any, .anything, .objectContaining, .arrayContaining, .stringMatching

Custom asymmetric equality tester: custom asymmetric equality providing an object that has asymmetricMatch function.

Jasmine Clock, mocking Date: jasmine.clock() .install(), .uninstall(), .tick(), .mockDate()

Jest (see also next generation Testing Framework – Vitest)

- **Jest** (based on Jasmine) is the **testing framework** created and maintained by **Facebook**. Self sufficient.
- **Performance -** faster for big projects with many test files by implementing a **parallel testing mechanism.**
- **Ready-To-Go** has assertions, spies, and mocks, no need combination-of-tools like mocha + chai + sinon
- **Globals** as in Jasmine, can be considered bad, it makes your tests less flexible but makes your life easier
- **Snapshot testing -** is to ensure that your app's UI doesn't unexpectedly change between releases.
- **Great modules mocking** - Easy way to mock heavy modules to improve testing speed.
- **Code coverage** - Includes a powerful and fast built-in code coverage tool that is based on Istanbul.
- **Reliability**- Has a huge community, used in many very complex projects
- **Support-** It is currently supported by all the major IDEs and tools.
- **Development-** jest only updates the files updated, so tests are running very fast in watch mode.

Mocha + Chai + Sinon

Mocha is the most used library.
Unlike Jasmine, it is used with third party assertions, mocking, and spying tools (usually Sinon and Chai).

- ❖ Community- Has many plugins and extension to test unique scenarios.
- ❖ Mocha includes the test structure as globals, saving time by not having to include or require it in every file.
- ❖ Mocha is **a little** harder to set up and divided into more libraries but it is more flexible and open to extensions.
- ❖ Flexibility in it’s assertions, spies and mocks is highly beneficial.

Choose Your Unit and Integration Tests Framework

The first choice you should probably make is which framework you want to use.

- *Angular apps first choice is **Jasmin***. Clean and obvious so that you can easily write tests.
- ***Jest** (Jasmin based) is very fast, clear, has many features in case you need to cover complex scenarios.*
- *If you want a very flexible and extendable configuration, go with **Mocha** **(Mocha+Chai+Sinon)**.*
- *If you are **minimalist** go with **Ava**. (no globals, install libs for more: mock, snapshot, parallelism)*
- ***QUnit** is mainly to test the jQuery (core, UI, and Mobile) JS libs but can be used to test other JS apps.*

What's Wrong with Mocha, Jasmine, etc…?

- *1. A lot config.(runner, assertion, report lib, ..). 2. Globals (`describe`, `it`, before ..) 3. Shared State(before..)*
  Moreover: Above tools leads to analysis paralysis (wide API tries e2e solution, code smell - non used mocks, …)

Tape

- tape *is a modular testing library* . Simplify your tests and app. by breaking into more modular chunks.
  1. Just loaded. 2. Simple module export. 3. Instead, call setup and teardown, & contain state to local test var
- If you prefer low-level, or dev. choice – writing maintainable tests) [*plan, deepEqual, looseEquals, equals, .. blue-tape]*
- Mock services - proxyquire module makes the process quite easy, require('proxyquire'), require('sinon');
- TAP-producing (tap-dot,nyc..) test harness for node & browsers. Its API is a small superset of the node core assert module.

Choose Your Functional Tests (AAT) Framework

Tools for the purpose of **functional testing** differ very much from each other in their implementation, philosophy, and API. So better understand the different solutions and testing them on your product.

- ❖ *If you want to "just get started" with a simple to set-up cross-browser all-in-one tool, go with **TestCafe**.*
- ❖ *For a convenient UI, clear doc., overall fun all-in-one tool Functional Testing experience go with **Cypress.io**.*
- ❖ *If you prefer older and more time-proven tools(*), you can "just get started" with **Nightwatch.js**.*
- ❖ *(*) with the maximum community support and flexibility, **Selenium** **Webdriver/IO** is the way to go.*
- ❖ *If you want the most reliable and Angular friendly solution, use **Protractor**.*
- ❖ *New, open-source, JavaScript-based, cross-browser automation library (aims fast&reliable) for E2E* **Playwright**

Automated Acceptance Testing Frameworks (other Langs): FitNesse (Java, ..), Robot[RIDE] (Python), etc.

A set of tools are built on top of Selenium to make this process even faster by directly transforming the BDD specifications into executable code. Some of these are ***JBehave, Capybara and Robot Framework***.

Cypress

Cypress is a free and open source automation tool, MIT-licensed and written in JS. Current version: 1.19

- Parallel testing was introduced in version 3.10.
- **Documentation**- Solid and clear.
- **Native access to all your application's variables** without serialization (TestCafe on the other hand turns objects to JSON, sends them to Node.js as text and then parses them back to objects).
- **Very convenient running and debugging tools**- Easy debugging and logging of the test process.
- **Cross-browser Support**- since version 4.0.
- **Some use-cases are missing** but in constant development such as lack of HTML5 drag-n-drop.
- **Using Mocha** as the test structure provider makes its use pretty standard

Limitations of Cypress

- One cannot use Cypress to drive two browsers at the same time.
- It doesn't provide support for multi-tabs.
- Cypress only supports JavaScript for creating test cases.
- Cypress doesn't provide support for browsers like Safari and IE at the moment.
- Limited support for iFrames.

Alternatives: Puppeteer by Google or Playwright by Microsoft

Read my blog about Playwright: [Test automation with Playwright for VS Code extensions]

Playwright

- Powerful automation capabilities
- Run tests across all browsers on all 3 platforms (Win, macOS, Linux) [cross browser, cross platform]
- **Cross-language.** Use the Playwright API in TypeScript, JavaScript, Python, .NET, Java. (async-TS/JS, sync-Java, Python-both)
- Execute tests in parallel.
- Enjoy context isolation out of the box.
- Capture videos, screenshots and other artifacts on failure.
- Integrate your POMs as extensible fixtures.
- **Powerful Tooling** – Codegen, Playwright Inspectors, Trace Viewer

Playwright is written by some of the same people who authored **Puppeteer** and it is maintained by Microsoft.

Limitations of Playwright

- No legacy MS Edge or IE11 support
- Desktop browsers are used to emulate mobile devices
- New so less community support and docs, ..

Installation

```
# Run from your project's root directory
> npm init playwright
# Or create a new project
➢ npm init playwright new-project
```

```
//Add dependency and install browsers.
> npm i -D @playwright/test
# install supported browsers
> npx playwright install
> npx playwright install webkit
```

Plaground: https://try.playwright.tech

Playwright

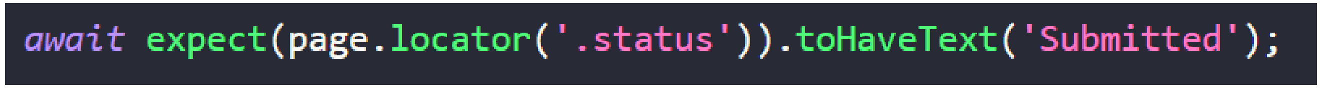
Configuration: to config default browser(or multi), context, page fixtures: actionTimeout, headless, viewport, storageState, …

- Global configuration: create playwright.config.ts | playwright.config.js or other file & specify opt. in testConfig.use section
- Global configuration: with test.use(options) you can override opt. in(or out of) test.describe(title, callback) block.

```
> npx playwright test --config=tests/my.config.js
```

Emulation: to emulate different env. Like mobile devices, locale, timezones, …
Network: to configure networking (downloads, errors, proxy, offline, network mocking .. -) `context.route`
Automatic screenshots: by default off, but can be tuned: ‘off’, ‘on’, ‘only-on-failure’
Record video and test trace: by default off, but can be tuned: ‘off’, ‘on’, ‘only-on-failure’
Testing options: to configure how tests run (forbidOnly[CI], retries, workers, testDir, …)

Playwright



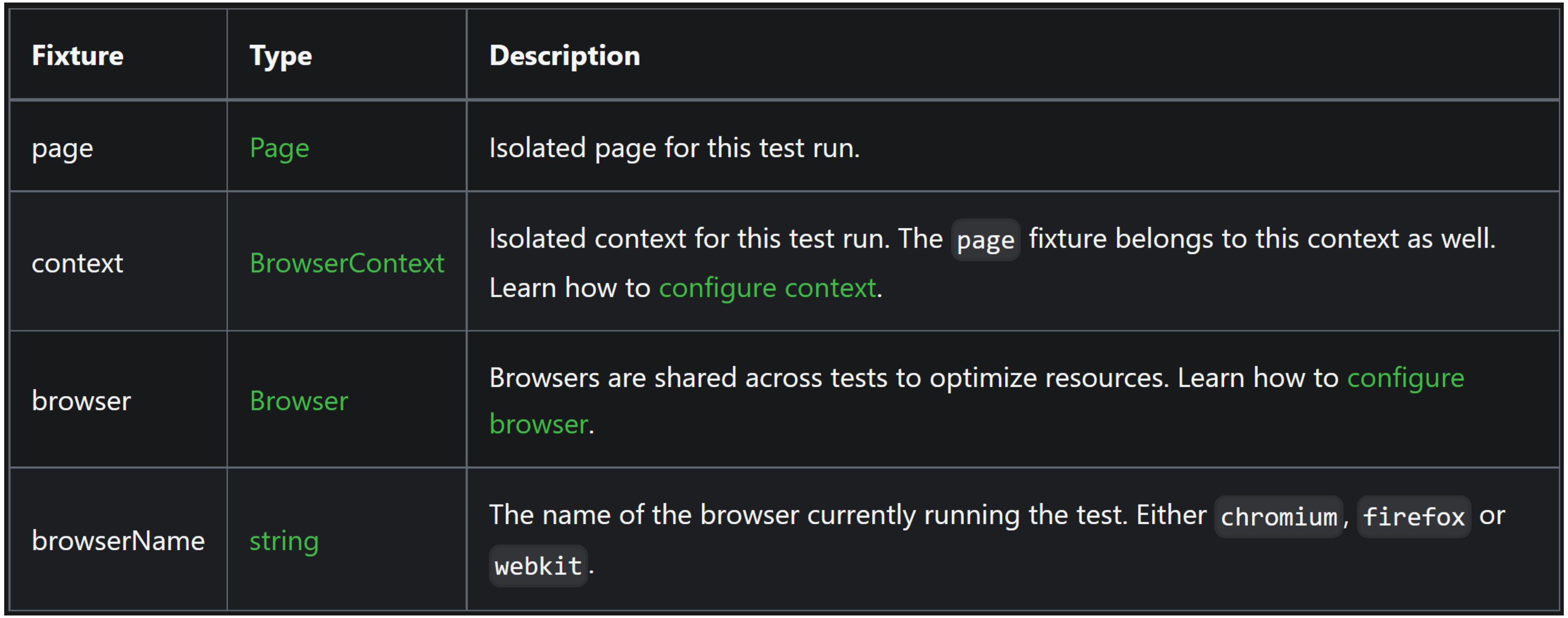
```
await expect(page.locator('.status')).toHaveText('Submitted');
```

Assertions:

Playwright uses expect library for test assertions.
Provides matchers like: toBeTruthy, toEqual, toContain, toMatch, .toHaveTitle, .toHaveAttribute, .toBeVisible(), toHaveCSS, and etc.
Negative matchers [.not], Soft matchers [.**soft** - not fail, just mark failed]

Testing Fixtures:

Fixtures are objects that are created for each test run. Below are pre-defined, and you can add yours as well.



Fixture	Type	Description
page	Page	Isolated page for this test run.
context	BrowserContext	Isolated context for this test run. The `page` fixture belongs to this context as well. Learn how to configure context.
browser	Browser	Browsers are shared across tests to optimize resources. Learn how to configure browser.
browserName	string	The name of the browser currently running the test. Either `chromium`, `firefox` or `webkit`.

Playwright

Test Hooks: (or setup/tearDown)

You can use test.beforeAll and test.afterAll hooks to set up and tear down resources shared between tests, and test.beforeEach and test.afterEach hooks for each test individually.

Command Line Patterns:

> npx playwright test #Run all tests on all browsers in a headless way
> npx playwright test --project=webkit

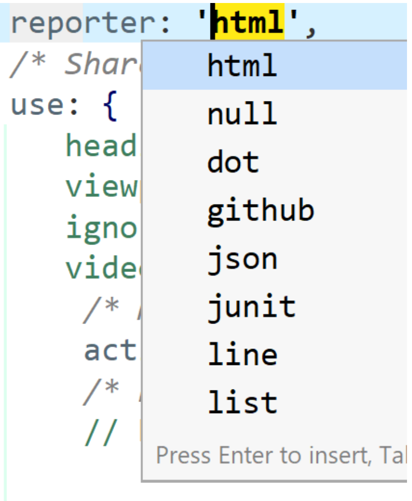
> npx playwright test –headed // in a headed browser

> npx playwright test tests/test1.spec.ts tests/test2.spec.ts #Run selected single, or more files

> npx playwright test --reporter=dot //**report options:** html, json, ..
> npx playwright show-report

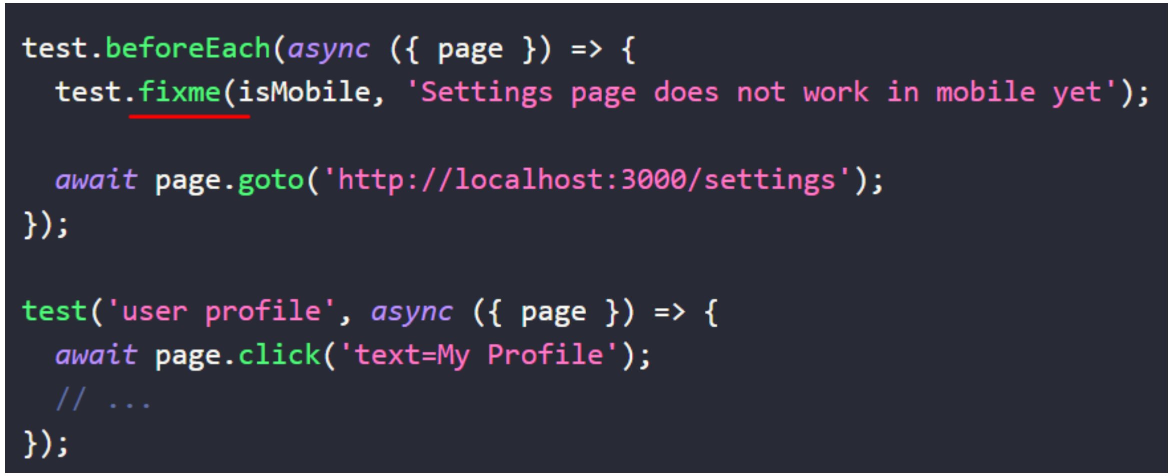
> npx playwright test --workers=1 //disable parallelization

> npx playwright test --debug //in debug mode with playwright inspector
> npx playwright test --help



```
reporter: 'html',
/* Shar
use: {
   head
   view
   igno
   vide
    /*
    act
    /*
    //
```

Playwright



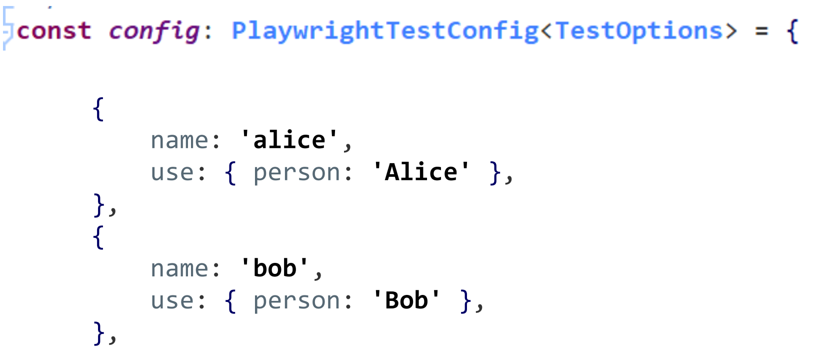
```
test.beforeEach(async ({ page }) => {
  test.fixme(isMobile, 'Settings page does not work in mobile yet');

  await page.goto('http://localhost:3000/settings');
});

test('user profile', async ({ page }) => {
  await page.click('text=My Profile');
  // ...
});
```

Annotations: to deal with failures, flakiness, skip, focus and tag tests.

test.**skip**(title, testFunction), test.**fail**([condition, description]), test.**fixme**(title, testFunction), test.**slow**([condition, description]) ,
test.only() – focused test, Tag tests @fast, @slow, ..

> npx playwright test tests/test9-iframe.spec.ts

Parametrize tests: can be parametrized on test or project level



```
const config: PlaywrightTestConfig<TestOptions> = {

    {
        name: 'alice',
        use: { person: 'Alice' },
    },
    {
        name: 'bob',
        use: { person: 'Bob' },
    },
```

API Testing:

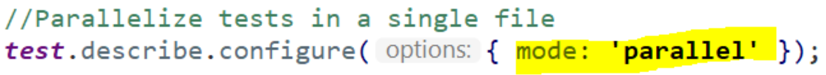
- ✓ Test your server API.
- ✓ Prepare server side state before visiting the web application in a test.
- ✓ Validate server side post-conditions after running some actions in the browser.

POM – Page Object Model: Abstraction over web-app pages to simplify interactions with them in multiple tests

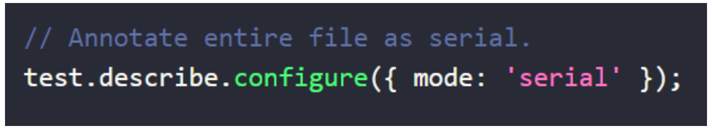
Parallelism and Sharding: Runs test in parallel (by default) by workers processes.

E.g. > npx playwright test **--workers 4** //to disable parallelism --workers=1

- Control Test Order, Serial mode, :
- SHARDs – Playwright can shard a test suite: >npx playwright test **–shard=x/y**
- Limit Failures & fail fast >npx playwright test --max-failures=10



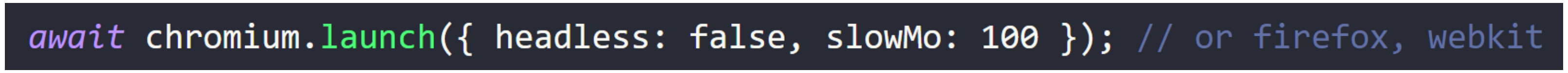
```
//Parallelize tests in a single file
test.describe.configure( options: { mode: 'parallel' });
```



```
// Annotate entire file as serial.
test.describe.configure({ mode: 'serial' });
```

Playwright

Tools



```
await chromium.launch({ headless: false, slowMo: 100 }); // or firefox, webkit
```

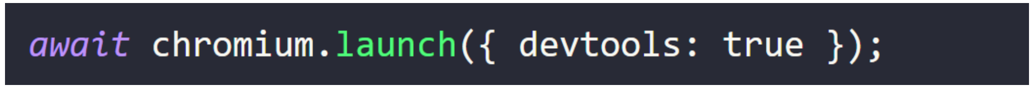
Playwright Inspector:

> npm run test PWDEBUG=1 //launch in headed mode, no timeout

And call [page.pause()](https://playwright.dev/docs/api/class-page#page-pause) in code

> playwright.inspect('text=Log in')

// https://playwright.dev/docs/debug



```
await chromium.launch({ devtools: true });
```

Code Generator: (code generate, emulate device)

> npx playwright codegen wikipedia.org OR > npx playwright codegen –save-storage=auth //save cookies, locatStorage timeout

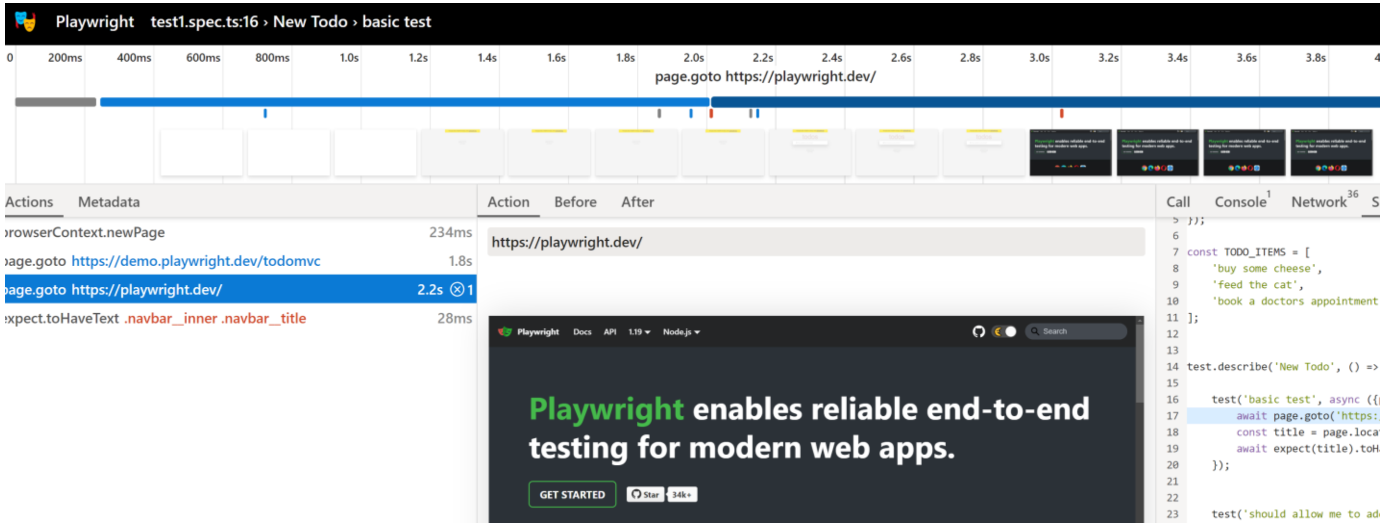
> npx playwright codegen --device="iPhone 11" wikipedia.org //emulate DEVICE, color scheme, size, geolocation, timezone/local, ..

Trace Viewer: Config: trace: **'on-first-retry'**, | 'ON'

- ➢ npx playwright show-trace trace.zip

or

- ➢ https://trace.playwright.dev/



Playwright test1.spec.ts:16 › New Todo › basic test

rowserContext.newPage	234ms
age.goto https://demo.playwright.dev/todomvc	1.8s
age.goto https://playwright.dev/	2.2s ⊗1
xpect.toHaveText .navbar__inner .navbar__title	28ms

https://playwright.dev/

Playwright enables reliable end-to-end testing for modern web apps.

```
});

const TODO_ITEMS = [
  'buy some cheese',
  'feed the cat',
  'book a doctors appointment
];


test.describe('New Todo', () =>

  test('basic test', async ({
    await page.goto('https:
    const title = page.loca
    await expect(title).toH
  });


  test('should allow me to ad
```

Pyramid of Clean Code

What is technical (design/code) debt?

Priority (speedy delivery) over Quality for long periods,

Poor skills, workarounds or easy fixes, etc.

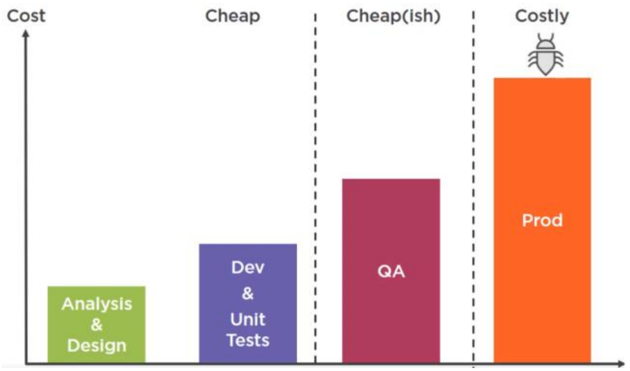
“Errors should be detected

as soon as possible […]

Ideally at compile time

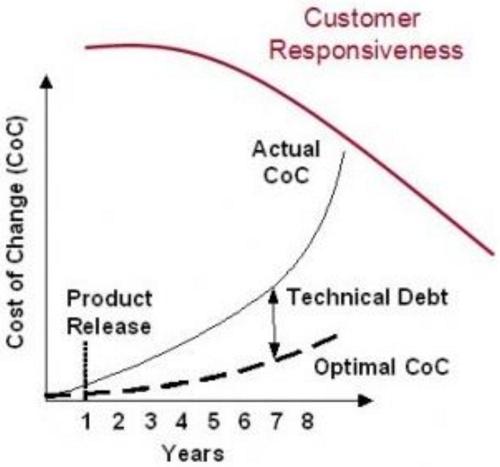
”

Joshua Bloch, Effective Java

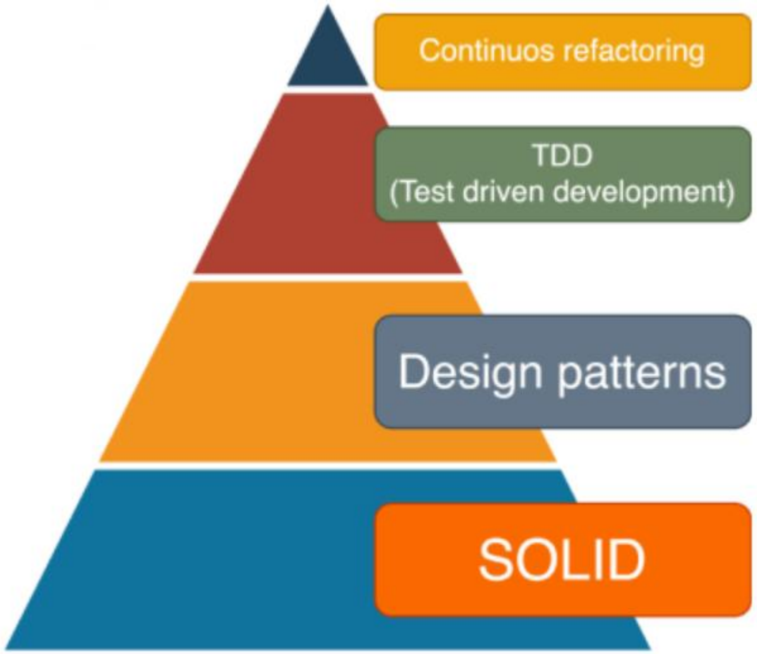


What is a Good Software?

Reliable, Efficient, Maintainable, Usable



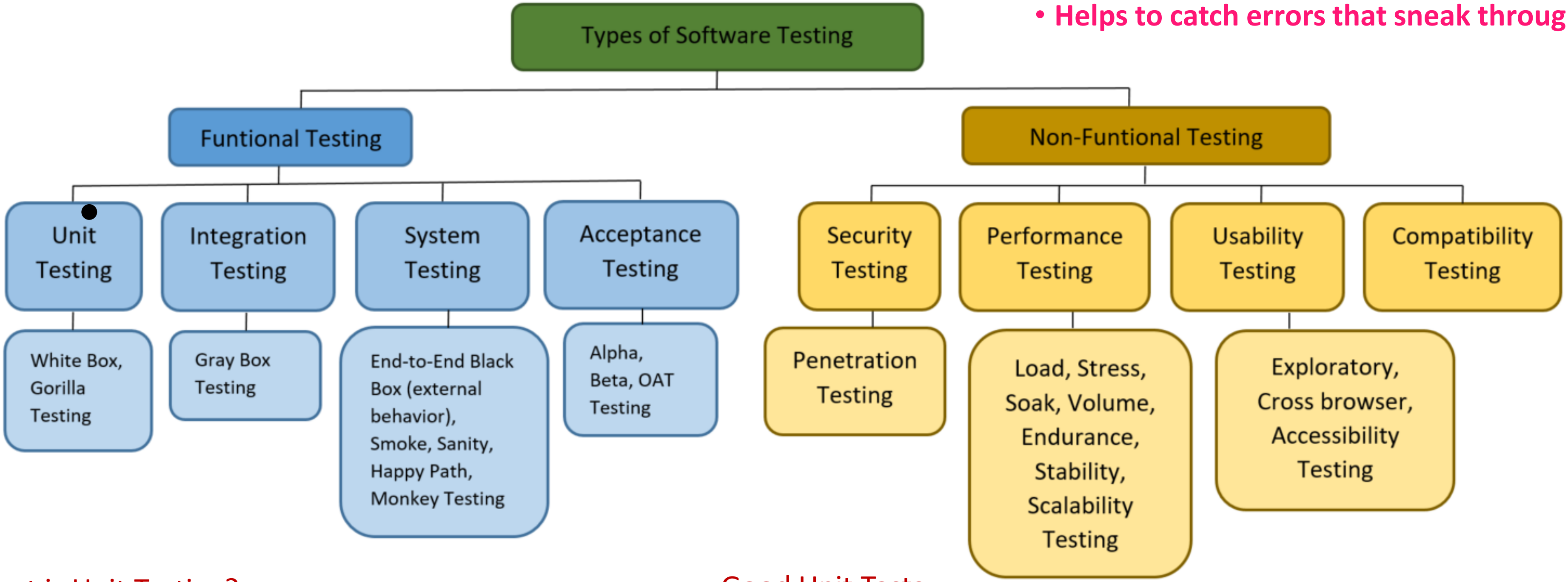
To keep compromise between **Time to shift** and **Technical debt**, simply keep clean code principles.



Testing Experience

Testing

- **A comprehensive testing strategy including unit tests and integration / end-to-end tests**
- **Helps to catch errors that sneak through the development stage**



Types of Software Testing

Funtional Testing

Unit Testing

White Box, Gorilla Testing

Integration Testing

Gray Box Testing

System Testing

End-to-End Black Box (external behavior), Smoke, Sanity, Happy Path, Monkey Testing

Acceptance Testing

Alpha, Beta, OAT Testing

Non-Funtional Testing

Security Testing

Penetration Testing

Performance Testing

Load, Stress, Soak, Volume, Endurance, Stability, Scalability Testing

Usability Testing

Exploratory, Cross browser, Accessibility Testing

Compatibility Testing

Other Testing Types:

- Negative
- Recovery
- **Regression**
- Vulnerability
- Ad-hoc
- more ..

What is Unit Testing?

Enables sustainable growth of the software

It is an automated test that:

- Verifies a small piece of code (unit)
- Does it quickly
- Does it in an isolated manner
- Integrated with development cycle

Unit Testing Benefits

reduce cost, ensure quality, mocking, doc, debugging easier, test-coverage-reports, TDD/BDD ...

Good Unit Tests

- Protects against regressions
- Resistance to refactoring
- Fast feedback [run all tests and see **;)**]

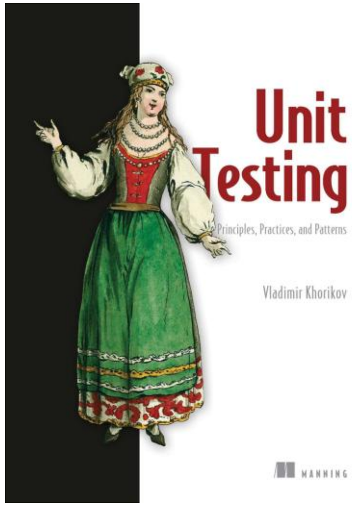
Follow

AAA|GivenWhenThen rule - TDD/BDD, ...

Try max test-cover, & run as a part of CI/CD

Challenges

test type and test double confusion, not whole-app test



Unit Testing

What is Unit Testing? A type of software testing where individual units or components of a software are tested. E.g. Unit Testing Frameworks (Java [Junit4|5, TestNG, Mockito,…], NodeJS [Jest, Jasmin, ..]

Why Unit Testing?

Unit testing takes into consideration the documentation (about functionality) of the entire system.

- ✓ Unit tests check the value and behavior of functions in various scenarios.
- ✓ Code covered with tests is more **reliable,** and **ensures** quality standards before its **deployment**.
- ✓ The debugging process can be simplified on certain test fail
- ✓ Design: Write unit tests *first*, then write the code, a.k.a **TDD**[developers+tester] or **BDD** [everyone involved, e.g. tools Cucumber, Gherkin]
- ✓ Reduce costs – Finding bugs in early stages reduces costs (UAT test, ..) significantly.
- ✓ Unit test fixtures can be used by performance tools to measure the success of an operation.
- ✓ **Enables CI/CD, publish test-reports, performance-reports, coverage on pipeline**

Unit Testing Challenges in Agile Team

- Writing wrong type of tests
- Misunderstanding doubles – is your stubs/mocks are more complicated than production code
- It is not testing the whole app, just a piece of the unit. E.g. None of the tests involve real DB transactions, etc.

Testing Best Practices

Following best practices when testing can give us:

- A test suite that is readable, maintainable, and easy to keep in sync with the app code
- A test suite that is efficient and runs as quickly as possible
- A test suite that is easy to extend and gives developers confidence when refactoring the application code

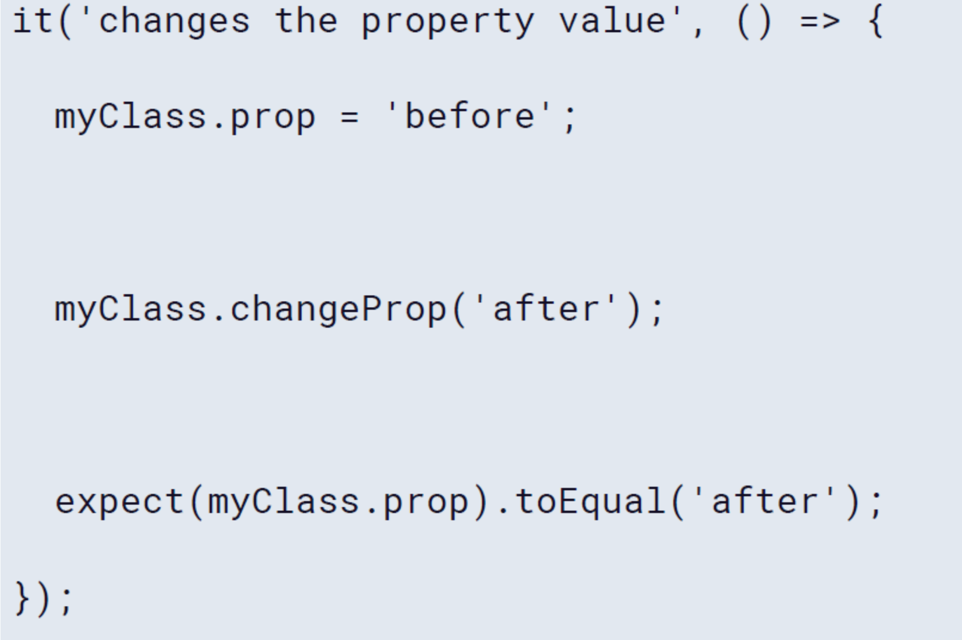
Unit Tests

- Typically written with and reside alongside the app code
- Most commonly written by developers
- Integration/e2e tests *usually* written by testers

Write descriptive and meaningful test names

- Get a good idea of what the application code does just by reading the test names
- Easily identify failing tests – especially helpful when there are many tests
- Clearly communicate to non-technical stakeholders the app's capabilities

Always use the AAA or GWT pattern: Arrange, Act, Assert or Given,When,Then



```
it('changes the property value', () => {

  myClass.prop = 'before';


  myClass.changeProp('after');


  expect(myClass.prop).toEqual('after');

});
```

◀ Arrange – create the necessary starting conditions for the test

◀ Act – perform the action which will lead to the required code being executed

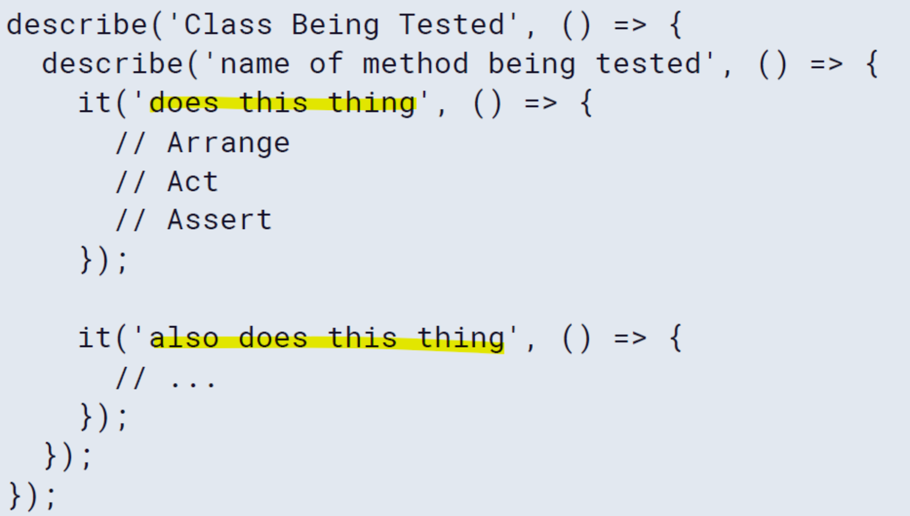
◀ Assert – check the result of performing the action to make sure the code did what it should

Test one thing, and test it well!

Keep tests small and focused for readability

Test Organization

- Test frameworks usually allow tests and test suites to be created
- Use `describe` to create suites
- Use `it` to create individual tests
- Group related tests together into suites



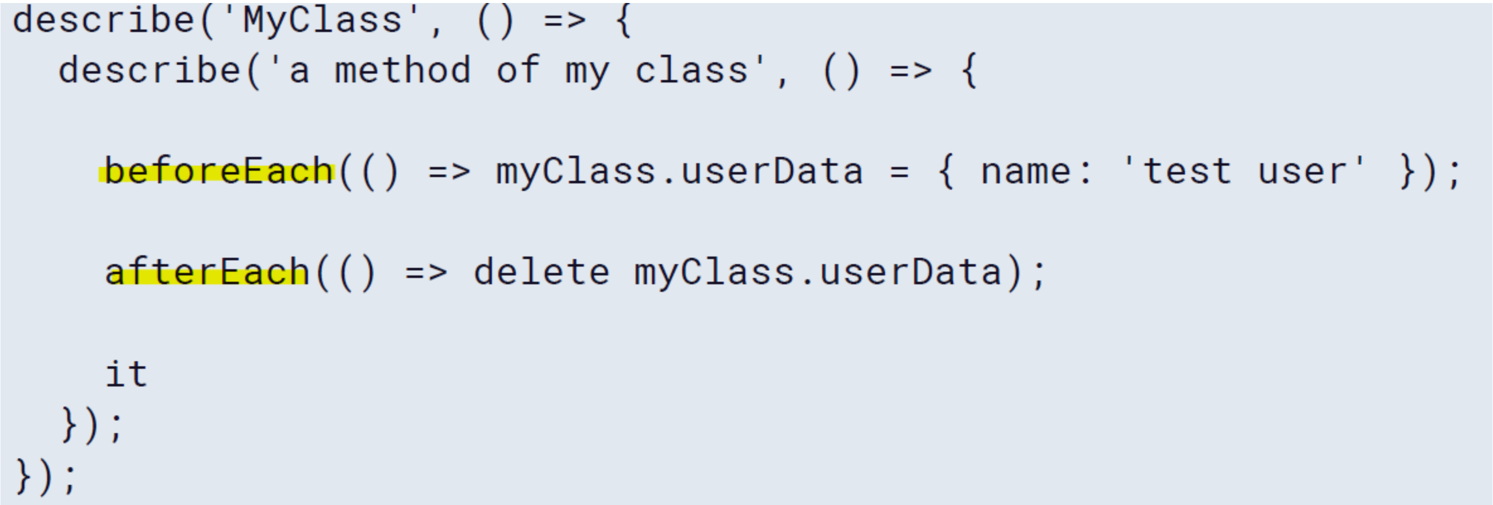
```
describe('Class Being Tested', () => {
  describe('name of method being tested', () => {
    it('does this thing', () => {
      // Arrange
      // Act
      // Assert
    });

    it('also does this thing', () => {
      // ...
    });
  });
});
```

Test Independence

- Tests should be able to run in any order
- Modern test frameworks usually run tests in a random order
- Tests should not be coupled to each other to avoid false-negatives
- Tests should only fail if the code does not work in the expected way
- Reliable tests are a good indicator of the health of the project
- All external dependencies **should be mocked**, e.g. services, repos
- Try to minimize mocking internal methods
- Improves coverage as private methods can also be exercised

Data Independence

- Ensure that tests don't need specific data to pass
- Avoid global or shared state
- Generate test data instead of using an external data-source
- Test frameworks provide set-up and tear-down functionality to setup the required data before each test, and clean it up after each test



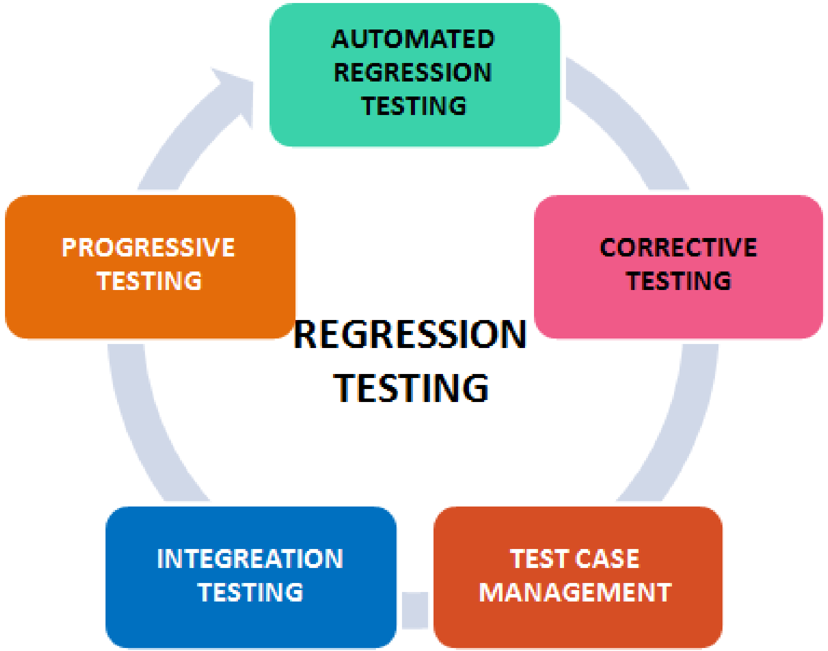
```
describe('MyClass', () => {
  describe('a method of my class', () => {

    beforeEach(() => myClass.userData = { name: 'test user' });

    afterEach(() => delete myClass.userData);

    it
  });
});
```

Regression Testing

What is Regression Testing? Used to find out whether the updates or changes (new patches, enhancements, performance upgrade, and new integration) had caused new defects in the existing functions. E.g. Frameworks (Java [Selenium, Cucumber, Robot Framework], NodeJS [Cypress, Playwright, Puppeteer ...]

Why Regression Testing? [acceptance, re-test, reduce cost]

- ✓ Ensures that new coding doesn’t interrupt existing code features, and no defects or bugs
- ✓ Evaluates the functionality of new code
- ✓ Allows for re-testing existing software after application code changes
- ✓ Under agile practice regression testing plays critical role as continuous testing is key to keep product stable
- ✓ Automating regression testing in agile processes helps to reduce re-work and frees up the testers to be used for other important testing activities
- ✓ What are the Various Types of Regression Testing?



Regression Testing Challenges in Agile Team

- Too many changes, e.g. in addition to planned features last time change-request or hot-fixes
- Time consuming – create, manage and maintain regression test suite requires a lot time
- Poor communication – developers, testing team, business analyst, and business stakeholders

Code Coverage

Code Coverage - Code Coverage Formats - Code Coverage Tools

In computer science, test coverage is a percentage measure of the degree (via instrumenting code) to which the source code of a program is executed when a particular test suite is run.

Testing criteria: is based on actual coded behavior, not requirements. So code behavior must reflect requirements.

Measure: Used to measure quality of test suite or compared test suite. Measuring is not so easy.

Basic coverage criteria (metrics):
Statement, Function, Branch, there are more .. …

Here is something you may find useful.

Green is for fully covered lines of code,
Yellow is for partly covered lines i.e. There may be some branches that missed to reach) and
Red is for lines that have not been executed at all.

Apart from these addition colored **diamond** are also shown and they have same meaning as above.

Test coverage vs Code Coverage
(*test coverage analysis or coverage monitor*)

Test Coverage *= (Executed Test cases/Total Test cases) * 100*
e.g *(300/500) * 100 = 60 %*
Code Coverage *= (number of lines of code executed / total number of lines of code)* 100*
e.g. *(400/500) * 100= 80 %*

Statement Coverage

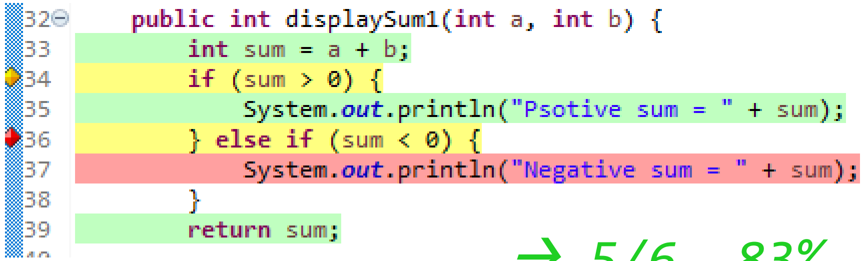
```
public int displaySum1(int a, int b) {
int sum = a + b;
if (sum > 0) {
    System.out.println("Psotive sum = " + sum);
} else if (sum < 0) {
    System.out.println("Negative sum = " +
sum);
} return sum;
}
```

Statement coverage – has each statement in the program been executed?

Statement coverage

$$\frac{\#\ \text{executed statements}}{\#\ statements} *100\ \%$$

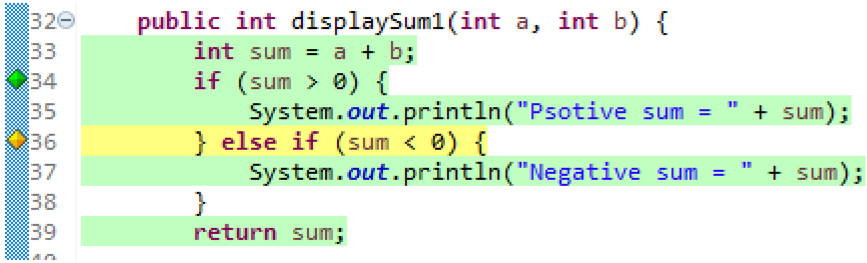
```
assertEquals(15, calculator.displaySum1(5, 10));
```



```
    public int displaySum1(int a, int b) {
        int sum = a + b;
        if (sum > 0) {
            System.out.println("Psotive sum = " + sum);
        } else if (sum < 0) {
            System.out.println("Negative sum = " + sum);
        }
        return sum;
```

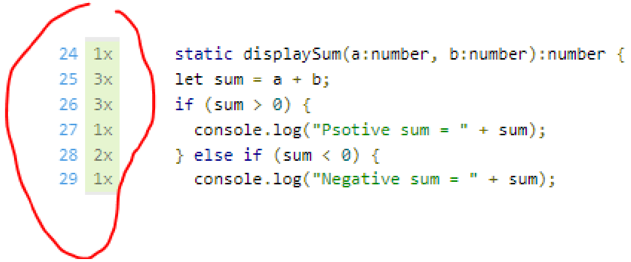
→ 5/6, 83%

```
assertEquals(-5, calculator.displaySum1(-10, 5));
```



```
    public int displaySum1(int a, int b) {
        int sum = a + b;
        if (sum > 0) {
            System.out.println("Psotive sum = " + sum);
        } else if (sum < 0) {
            System.out.println("Negative sum = " + sum);
        }
        return sum;
```

→ 6/6, 100%, but what is missing?



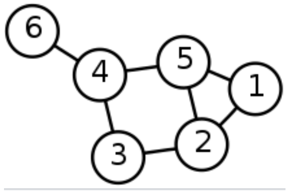
```
1x  static displaySum(a:number, b:number):number {
3x  let sum = a + b;
3x  if (sum > 0) {
1x    console.log("Psotive sum = " + sum);
2x  } else if (sum < 0) {
1x    console.log("Negative sum = " + sum);
```

Statement Coverage - Ensures that every statement in the application has been executed (**hits**) at least once.

Branch coverage

Edge (path) coverage – has every edge in the control-flow graph been executed?

- ❑ **Branch coverage** – has each branch (also called the DD-path) of each control structure (such as in *if* and *case* statements) been executed? For example, given an *if* statement, have both the *true* and *false* branches been executed? (This is a subset of edge coverage.)



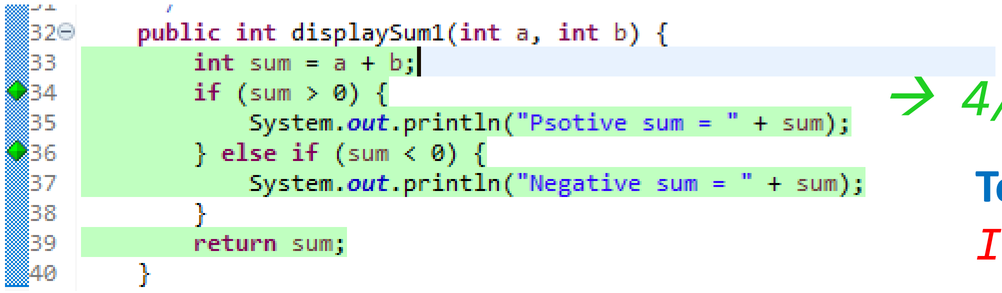
Branch coverage

$$\frac{\#\text{ executed branches}}{\#\text{branches}} *100\ \%$$

```
public int displaySum1(int a, int b) {
int sum = a + b;
if (sum > 0) {
    System.out.println("Psotive sum = " + sum);
} else if (sum < 0) {
    System.out.println("Negative sum = " + sum);
}
//else ???
return sum;
}
```

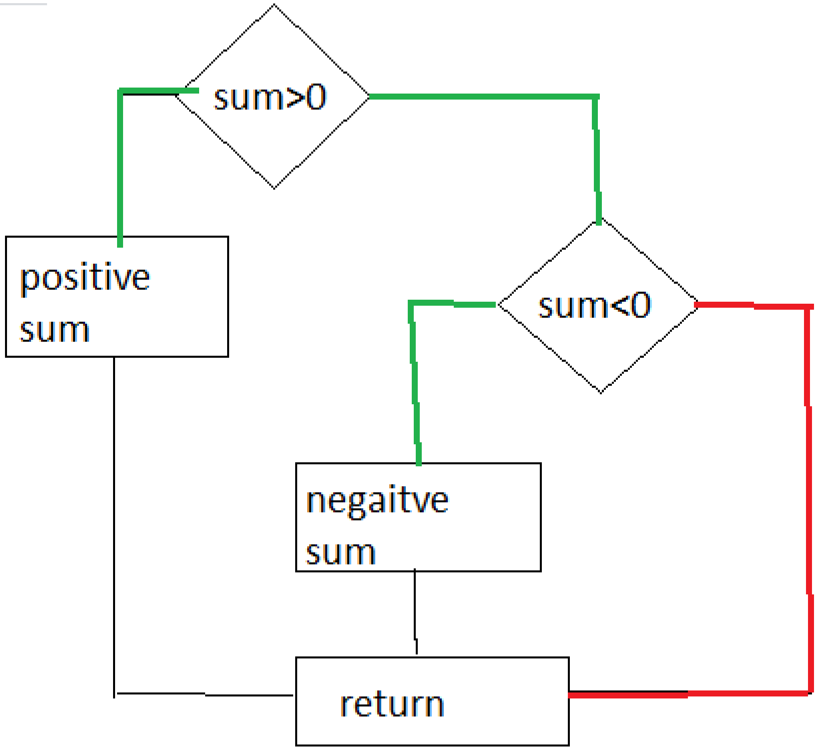
→ *3/4, 75%*

```
    assertEquals(0, calculator.displaySum1(-10, 10));
```



```
    public int displaySum1(int a, int b) {
        int sum = a + b;
        if (sum > 0) {
            System.out.println("Psotive sum = " + sum);
        } else if (sum < 0) {
            System.out.println("Negative sum = " + sum);
        }
        return sum;
    }
```

→ *4/4, 100%*



Test criteria – if you cover all **branches** it implies all **statements** cover

Is that guarantee if all the time 100% coverage?

Branch Coverage - Ensures that each branch of the program has been executed.

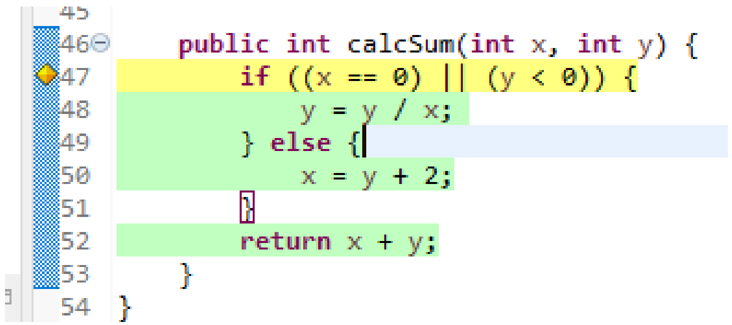
Condition coverage

Condition coverage – has each Boolean sub-expression evaluated both to true and false? (Also called predicate coverage.)

Branch coverage

$$\frac{\#\text{ condition both T and F!}}{\#\text{conditions}} *100\ \%$$

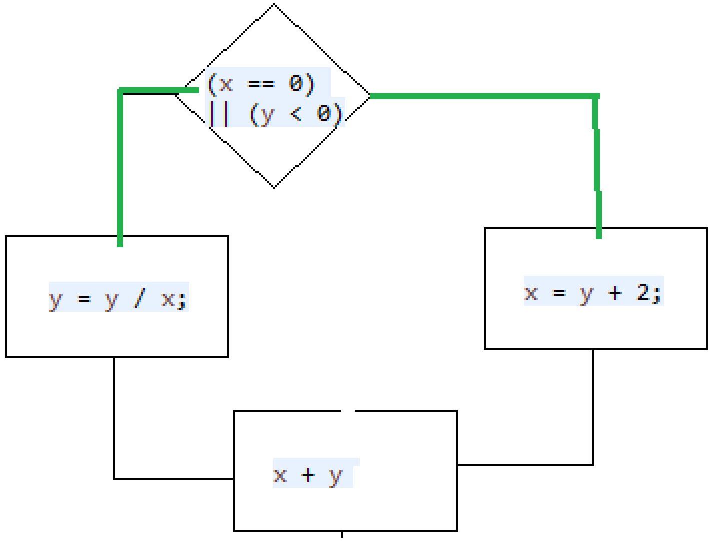
```
assertEquals(12, calculator.calcSum(5, 5));
assertEquals(-4, calculator.calcSum(-5, -5));
```



```
    public int calcSum(int x, int y) {
        if ((x == 0) || (y < 0)) {
            y = y / x;
        } else {
            x = y + 2;
        }
        return x + y;
    }
}
```

→ Branch coverage 100%

→ But condition X==0 (not covered)



Test criteria – does condition coverage implies branch coverage? **YES?/NO [*X=0,y<0*]**
does condition coverage implies statement coverage? **NO [*X=0,y<0*]**
How can we make 100% coverage then?

Condition Coverage - Ensures that each decision has evaluated to true and false.

Branch and Condition coverage

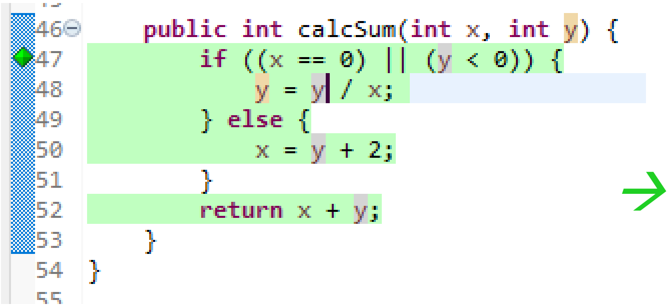
Branch and Condition coverage

$$\frac{\#\ \text{branches + condition both T and F}}{\#\text{branches+conditions}} *100\ \%$$

Branch and Condition coverage *On e.g. X==0 || y<0*

Test Case	X	y	Branch
1	0	5	T
2	5	5	F
3	5	5	T

```
//1
ArithmeticException thrown = assertThrows(ArithmeticException.class, () -> {
calculator.calcSum(0, 5);
}, "by zero");
//2
assertEquals(12, calculator.calcSum(5, 5));
//3
assertEquals(-4, calculator.calcSum(-5, -5));
```



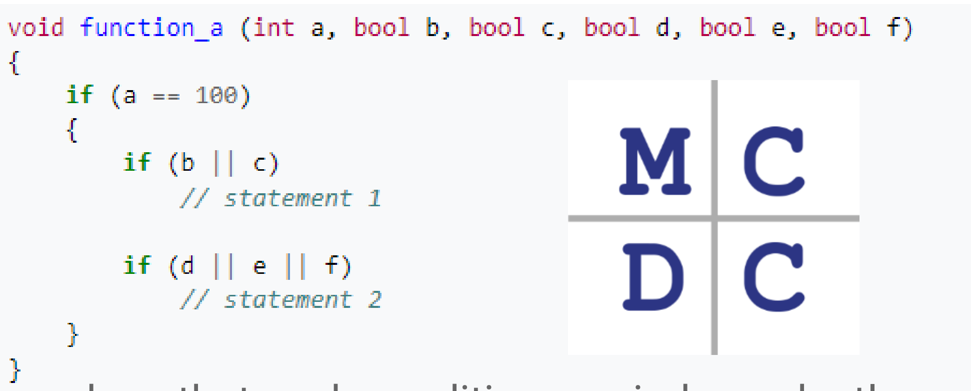
```
    public int calcSum(int x, int y) {
        if ((x == 0) || (y < 0)) {
            y = y / x;
        } else {
            x = y + 2;
        }
        return x + y;
    }
}
```

→ *100%*

Test criteria – branch and condition coverage implies statement coverage

MC|DC coverage

For more detailed MC|DC coverage (modified condition decision coverage) – see MC|DC, e.g. **Aviation**, **Cosmonautics** must have 100% MC|DC coverage



```
void function_a (int a, bool b, bool c, bool d, bool e, bool f)
{
    if (a == 100)
    {
        if (b || c)
            // statement 1

        if (d || e || f)
            // statement 2
    }
}
```

MC/DC - Ensures that each decision has evaluated to true and false, as well as show that each condition can independently affect the decision outcome.

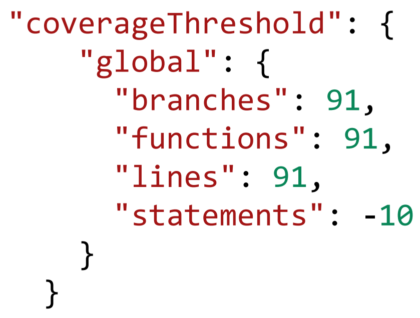
Function coverage

❑ **Function coverage** – has each function (or subroutine) in the program been called? All functions that are in the source code get tested during test execution.

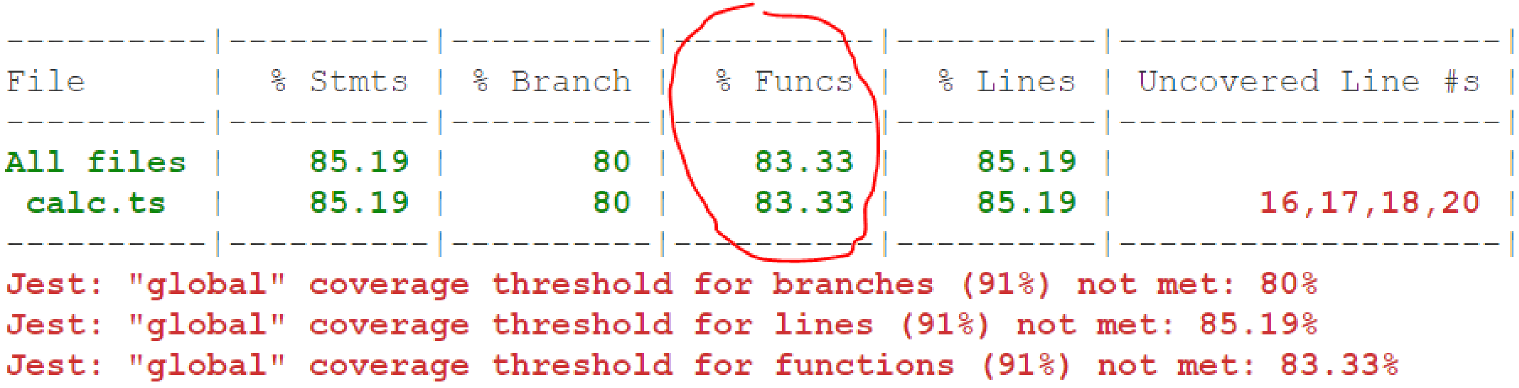
Branch coverage

$$\frac{\text{\# functions covered with test}}{\text{\#function}} * 100\ \%$$

Call Coverage - This metric reports whether you **executed each function call**. The hypothesis is that bugs commonly occur in interfaces between modules.



```
"coverageThreshold": {
    "global": {
      "branches": 91,
      "functions": 91,
      "lines": 91,
      "statements": -10
    }
  }
```



```
----------|----------|----------|----------|----------|-------------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------|----------|----------|----------|----------|-------------------|
All files |    85.19 |       80 |    83.33 |    85.19 |                   |
 calc.ts  |    85.19 |       80 |    83.33 |    85.19 |       16,17,18,20 |
----------|----------|----------|----------|----------|-------------------|
Jest: "global" coverage threshold for branches (91%) not met: 80%
Jest: "global" coverage threshold for lines (91%) not met: 85.19%
Jest: "global" coverage threshold for functions (91%) not met: 83.33%
```

Choosing good intermediate coverage goals

Choosing **good intermediate coverage goals** can greatly increase testing productivity.

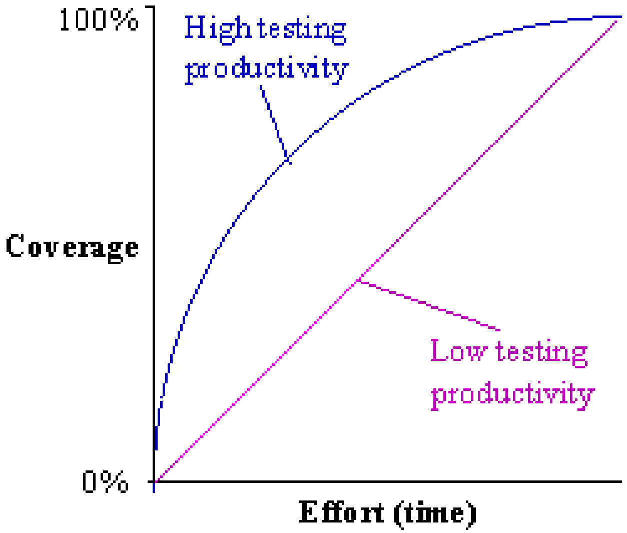


Figure 1: Coverage rate

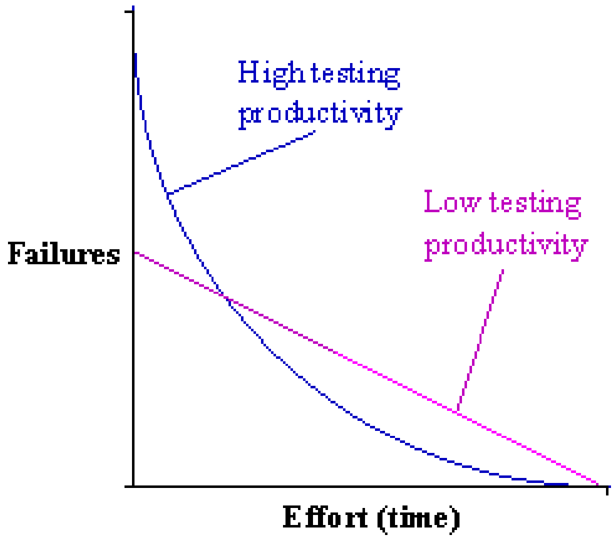


Figure 2: Failure discovery rate

Use coverage analysis strategy
that increases coverage

Figure 1 illustrates the coverage rates for high and low productivity. **Figure 2** shows the corresponding failure discovery rates.

The sequence of coverage goals listed below illustrates a possible implementation of this strategy.

1. Invoke at least one function in 90% of the source files (or classes).
2. Invoke 90% of the functions.
3. Attain 90% condition/decision coverage in each function.
4. Attain 100% condition/decision coverage.

Simply SET "coverageThreshold":

Format of coverage.json/coverage-final.json

coverage-final.json

coverage.json contains a report object, which is a hash where **keys** are file names (absolute paths), and values are coverage data for that file (result of JSON.stringify(coverageFinal,..)) Each entry consists of:

- **path** - The path to the file.
- **s** - Hash of statement counts, where keys as statement IDs. Used for “l”-lines as well
- **b** - Hash of branch counts, where keys are branch IDs and values are arrays of counts. For an if statement, the value would have two counts; one for the if, and one for the else. Switch statements would have an array of values for each case.
- f - Hash of function counts, where keys are function IDs.
- **fnMap** - ..
- **statementMap** - ..
- **branchMap - ..**

- **Location objects** are a {start: {line, column}, end: {line, column}} object that describes the start and end of a piece of code. Note that line is 1-based, but column is 0-based.

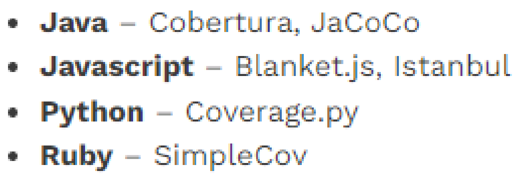
Format of lcov.info

lcov.info lcov.info

Following is tracefile format for **lcov.info**, which is as used by genhtml, geninfo and lcov. tools

- **TN** - <test name> or key to identify
- **SF** – absolute path to the source file
- statements - not mapped in LCOV.info, represented inside DA
- **BRDA** - Branch coverage (condition coverage) information, branchMap
- **BRF**(# branches found), **BRH** (# branches hit [covered info])
- **FNDA** - Function coverage information, fnMap
- **FNF** (# functions found)), **FNH** (# function hit [covered info])
- **FN** - function
- **LF** - number of instrumented lines
- **LH** - number of lines with a non-zero execution count/hit
- **DA**:<line number>,<execution count>[,<checksum>] - list of execution counts for each instrumented line. an optional checksum present for each instru- mented line.
- each sections ends with **end_of_record**

Code Coverage Tools



- **Java** – Cobertura, JaCoCo
- **Javascript** – Blanket.js, Istanbul
- **Python** – Coverage.py
- **Ruby** – SimpleCov

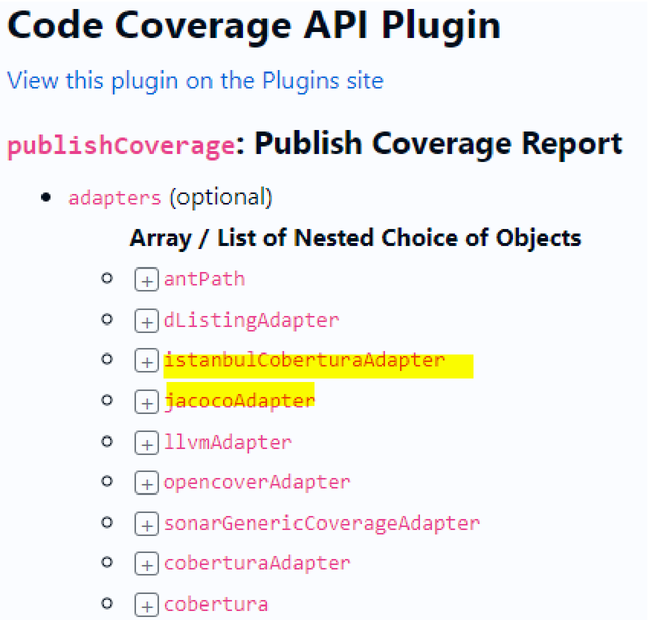
Enlisted below are some of the tools for your reference (free & paid, …)

- Parasoft JTest
- Testwell CTC++
- PITest
- BullseyeCoverage
- OpenCover
- NCover
- Squish COCO
- CoverageMeter
- GCT
- TCAT C/C++
- Gretel
- AtlassianClover
- **Jcov**
- CodeCover (Java&Cobol
- Cobertura
- EMMA (Eclemma)
- **JaCoCo**
- Istanbul/NYC
- my PoC LINK – Code Coverage

Cobertura is an open source code coverage tool for Java. This is a Jcoverage based tool. To use this tool one should declare Maven plug-in in POM.XML file.

JaCoCo is a free code coverage toolkit developed by EclEmma. It was developed for the replacement of Emma code coverage tool. It can be used only for measuring and reporting Java-based applications.

https://www.jenkins.io/doc/pipeline/steps/code-coverage-api/



Code Coverage API Plugin

View this plugin on the Plugins site

`publishCoverage`: **Publish Coverage Report**

- `adapters` (optional)

  Array / List of Nested Choice of Objects

  - ⊞ `antPath`
  - ⊞ `dListingAdapter`
  - ⊞ `istanbulCoberturaAdapter`
  - ⊞ `jacocoAdapter`
  - ⊞ `llvmAdapter`
  - ⊞ `opencoverAdapter`
  - ⊞ `sonarGenericCoverageAdapter`
  - ⊞ `coberturaAdapter`
  - ⊞ `cobertura`

Code Coverage Tools - Istanbul/NYC

Istanbul instruments your ES5 and ES2015+ JavaScript code with line counters, so that you can track how well your unit-tests exercise your codebase. NYC - the Istanbul command line interface works with mocha, tap, AVA etc.

○ JS code coverage tool that computes statement, line, function and branch coverage.
○ Multiple report formats: HTML, LCOV, Cobertura , clover(XML), txt.
○ Can be used on the command line as well as a library
○ Coverage **thresholds**, high and low **watermarks**

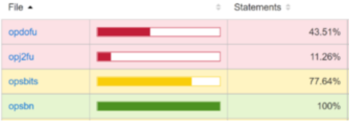
```
{
"branches": 80,
"lines": 80,
"functions": 80,
"statements": 80
}
```

```
{
 "watermarks": {
 "lines": [80, 95],
"functions": [80, 95],
"branches": [80, 95],
 "statements": [80, 95] }
}
```

```
"scripts": { //jest  uses nyc internally
  "test": "jest",
  "coverage": "jest --coverage“ //report generated
},
```

>npm i –D (or --save-dev) nyc // or yarn add -D nyc

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



File ▲		Statements
opdofu		43.51%
opj2fu		11.26%
opsbits		77.64%
opsbn		100%

lcov.info   clover.xml   cobertura-coverage.xml

//or in case you have “coverage-final.json” , runs in cmd.

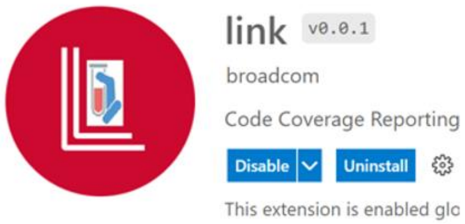
```
 "scripts": {
     "report": "nyc report",  //needed nycrc.json
     "coverage": "npm run report && start coverage/index.html",
  },
```

.nycrc.json

//or programmatically

```
const NYC = require("nyc");
    const nyc = new NYC({
     cwd: vscode.workspace.workspaceFolders[0].uri.fsPath,
     reporter: ["html", "lcov", "cobertura", "clover"],
     reportDir: "./coverage",
     hookRequire: true,
     extension: [".cob"],
     tempDir: "./coverage",    //default ./.nyc_output (looks coverage file here)
    });

    nyc.report((err: any) => { ..  });
```

Code Coverage Visualization Tool


link v0.0.1

broadcom

Code Coverage Reporting

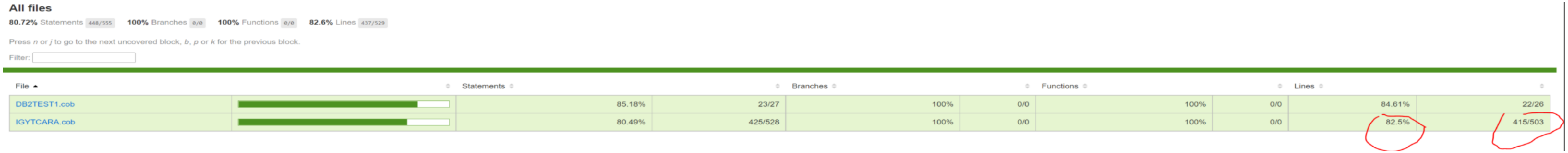
This extension is enabled glo

- **PROBLEM**

Michelle wants to see coverage info for her COBOL programs which has unit tests

- **SOLUTION**

Use VSCode extension called “LINK Code Coverage Visualization” which enables/shows code **coverage metrics in dashboard**, coverage **ratio info** on status bar, **gutter indicators**, **warning messages** for not-covered lines in console, and COBOL **control flow highlighted** with coverage colors (red/amber/green). Also provides **configuration**, and moreover dashboard can be opened in **vscode tab** [todo.]

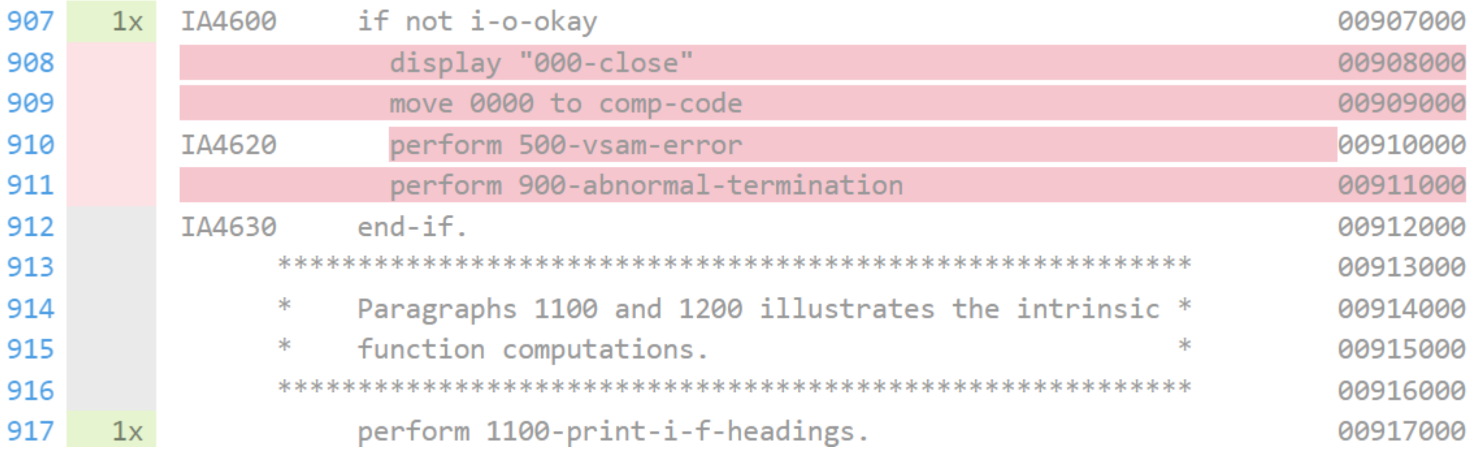

All files

80.72% Statements 448/555 **100%** Branches 0/0 **100%** Functions 0/0 **82.6%** Lines 437/529

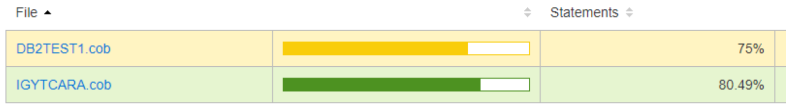
Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

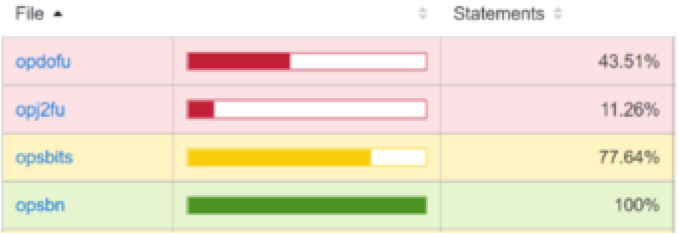
Filter:

File ▲		Statements		Branches		Functions		Lines	
DB2TEST1.cob		85.18%	23/27	100%	0/0	100%	0/0	84.61%	22/26
IGYTCARA.cob		80.49%	425/528	100%	0/0	100%	0/0	82.5%	415/503

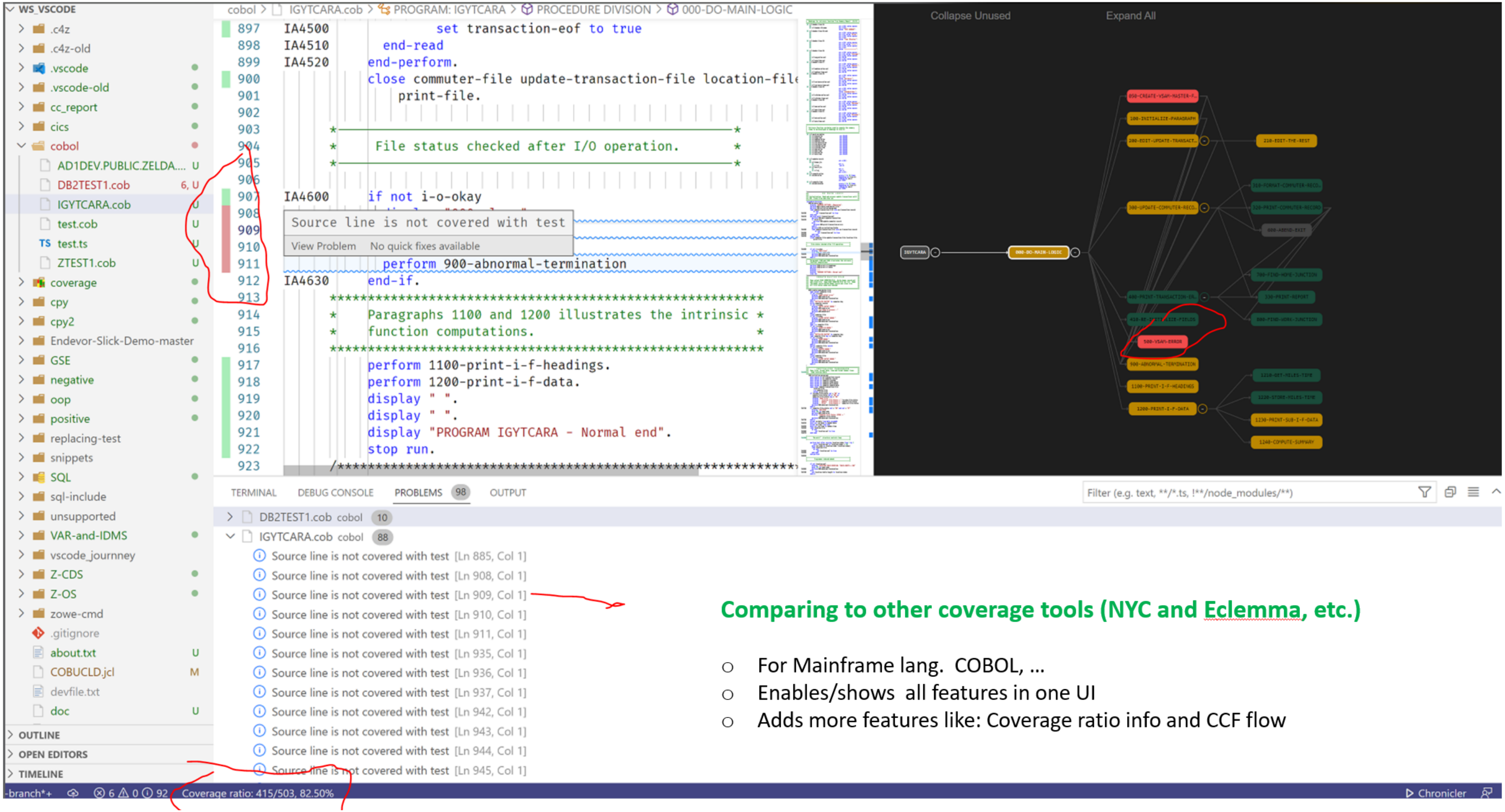

```
1x  IA4600     if not i-o-okay                                  00907000
                 display "000-close"                            00908000
                 move 0000 to comp-code                         00909000
    IA4620       perform 500-vsam-error                         00910000
                 perform 900-abnormal-termination               00911000
    IA4630     end-if.                                          00912000
           ****************************************************  00913000
           *    Paragraphs 1100 and 1200 illustrates the intrinsic *  00914000
           *    function computations.                          *  00915000
           ****************************************************  00916000
1x             perform 1100-print-i-f-headings.                 00917000
```

More examples


File ▲		Statements
DB2TEST1.cob		75%
IGYTCARA.cob		80.49%


File ▲		Statements
opdofu		43.51%
opj2fu		11.26%
opsbits		77.64%
opsbn		100%

Full/Partial/No Coverage



```
897  IA4500           set transaction-eof to true
898  IA4510        end-read
899  IA4520       end-perform.
900               close commuter-file update-transaction-file location-file
901                   print-file.
902
903           *------------------------------------------------------------*
904           *    File status checked after I/O operation.               *
905           *------------------------------------------------------------*
906
907  IA4600       if not i-o-okay
908
909
910
911                  perform 900-abnormal-termination
912  IA4630       end-if.
913           ***********************************************************
914           *   Paragraphs 1100 and 1200 illustrates the intrinsic *
915           *   function computations.                              *
916           ***********************************************************
917               perform 1100-print-i-f-headings.
918               perform 1200-print-i-f-data.
919               display " ".
920               display " ".
921               display "PROGRAM IGYTCARA - Normal end".
922               stop run.
923           /**************************************************************
```

```
Source line is not covered with test
```

```
> DB2TEST1.cob cobol 10
v IGYTCARA.cob cobol 88
  ⓘ Source line is not covered with test [Ln 885, Col 1]
  ⓘ Source line is not covered with test [Ln 908, Col 1]
  ⓘ Source line is not covered with test [Ln 909, Col 1]
  ⓘ Source line is not covered with test [Ln 910, Col 1]
  ⓘ Source line is not covered with test [Ln 911, Col 1]
  ⓘ Source line is not covered with test [Ln 935, Col 1]
  ⓘ Source line is not covered with test [Ln 936, Col 1]
  ⓘ Source line is not covered with test [Ln 937, Col 1]
  ⓘ Source line is not covered with test [Ln 942, Col 1]
  ⓘ Source line is not covered with test [Ln 943, Col 1]
  ⓘ Source line is not covered with test [Ln 944, Col 1]
  ⓘ Source line is not covered with test [Ln 945, Col 1]
```

Coverage ratio: 415/503, 82.50%

Comparing to other coverage tools (NYC and Eclemma, etc.)

- For Mainframe lang. COBOL, ...
- Enables/shows all features in one UI
- Adds more features like: Coverage ratio info and CCF flow

THANK YOU

References

https://github.com/azatsatklichov/java-and-ts-tests
https://medium.com/welldone-software/an-overview-of-javascript-testing-7ce7298b9870
https://catonmat.net/writing-javascript-tests-with-tape
https://ci.testling.com/
https://codecept.io/basics/