Software Engineering

A Faculty of Informatics and Computer Science Course: CSEN 303

Software Testing

11

Dr. Iman Awaad

iman.awaad@giu-uni.de

GIU GERMAN INTERNATIONAL UNIVERSITY
الجامعة الألمانية الدولية

Acknowledgments

The slides are **heavily** based on the **slides** by **Prof. Dr. John Zaki**.

They are also **heavily** based on the slides and textbook by **Ian Somerville**.

Their contribution is gratefully acknowledged.

Any additional sources are referenced.

Software testing

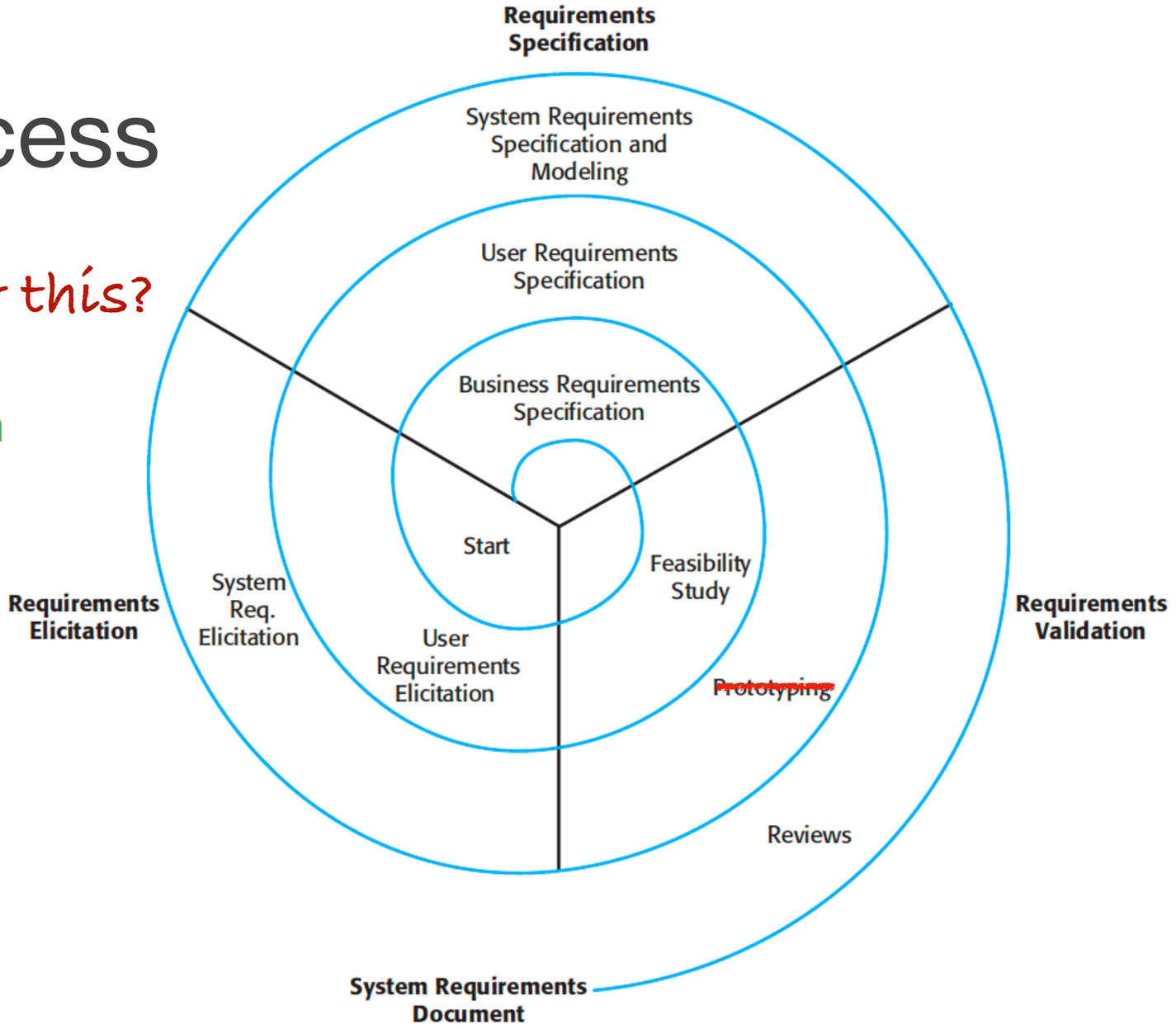
- Software testing:
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
- Ways of testing software:
  - White-box
  - Black-box
  - Grey-box
- Testing with Jest/Mocha

Why do we test software?

How do we test it?

Requirements engineering process

Remember this?



Phase 6: **Verification & Validation**

- Technical review, missing information, checking errors, built according to standards: Simple sanity check, test-case generation, inspection, requirement review, coded prototypes, design prototypes
- **Verification**: built the **product correctly**….
- **Validation**: built the **correct product**…

Software testing

…is the **process** in which you **execute your program using data that simulates user inputs**, then observe its behaviour to see whether your program is doing what it is supposed to do (or not)…

Pass
if it behaves as expected

Fail
if behaviour differs from the expected

assumption?

Software testing

…is the **process** in which you **execute your program using data that simulates user inputs**, then observe its behaviour to see whether your program is doing what it is supposed to do (or not)…

Pass
if it behaves as expected

Fail
if behaviour differs from the expected

assumption?

These inputs are **representative** of a larger set of inputs → the program will behave correctly for all members of the larger input set

Manual or automated testing…

Manual testing

Someone tests the software using some test data and compares the result to their expectations

Automated testing

Tests are encoded into a program that is run each time the system under development is to be tested.

Useful in **regression testing** (testing older, already tested, parts of the software after having made changes so as to ensure that updates haven't introduce new bugs)

Continuous integration

Remember this?

…integrated version of the system is **created** and **tested** every time a change is **pushed** to the system's shared repository: the repository sends a message to an integration server to build a new version of the product

Advantage: faster to find and fix bugs

If you make a small change and some system tests fail, the problem almost certainly lies in the new code that you have pushed to the **project repo**.

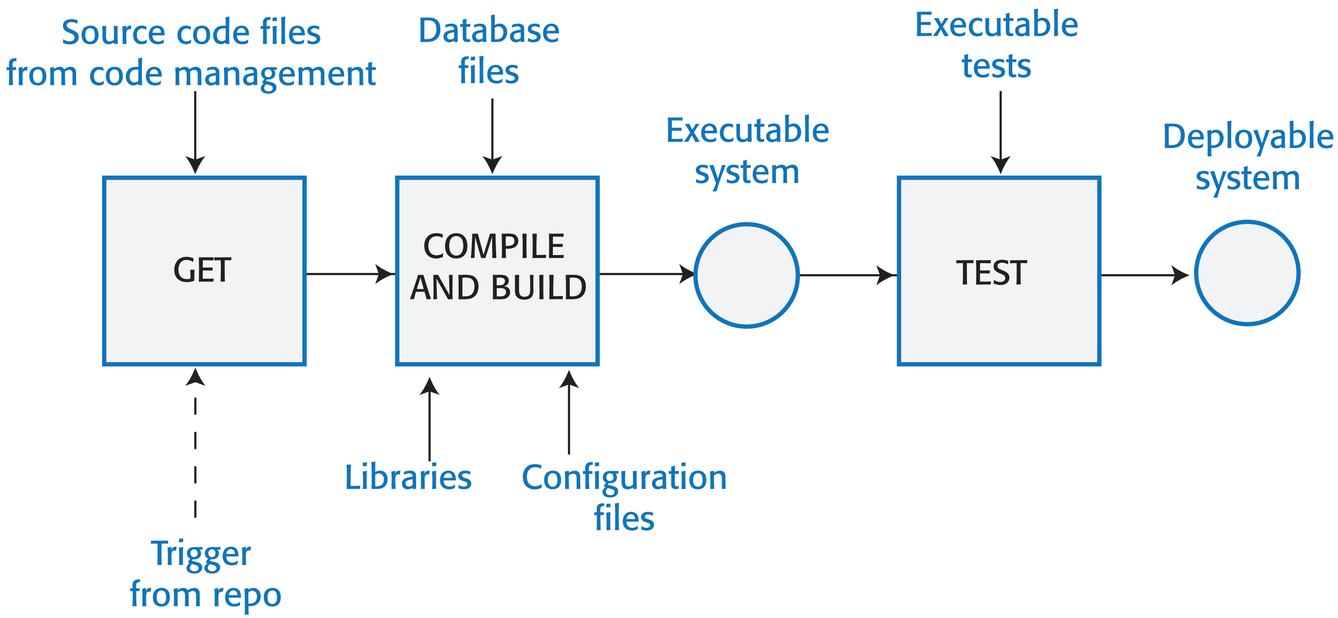


Figure 10.9 Continuous integration

"Don't break the build!"

Continuous delivery and deployment

Remember this?

Real environment in which software runs will be different from our development system… bugs may be revealed that weren't in test environment

Continuous delivery: after making changes to a system, we ensure that it is ready for delivery to customers!

Must test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance!

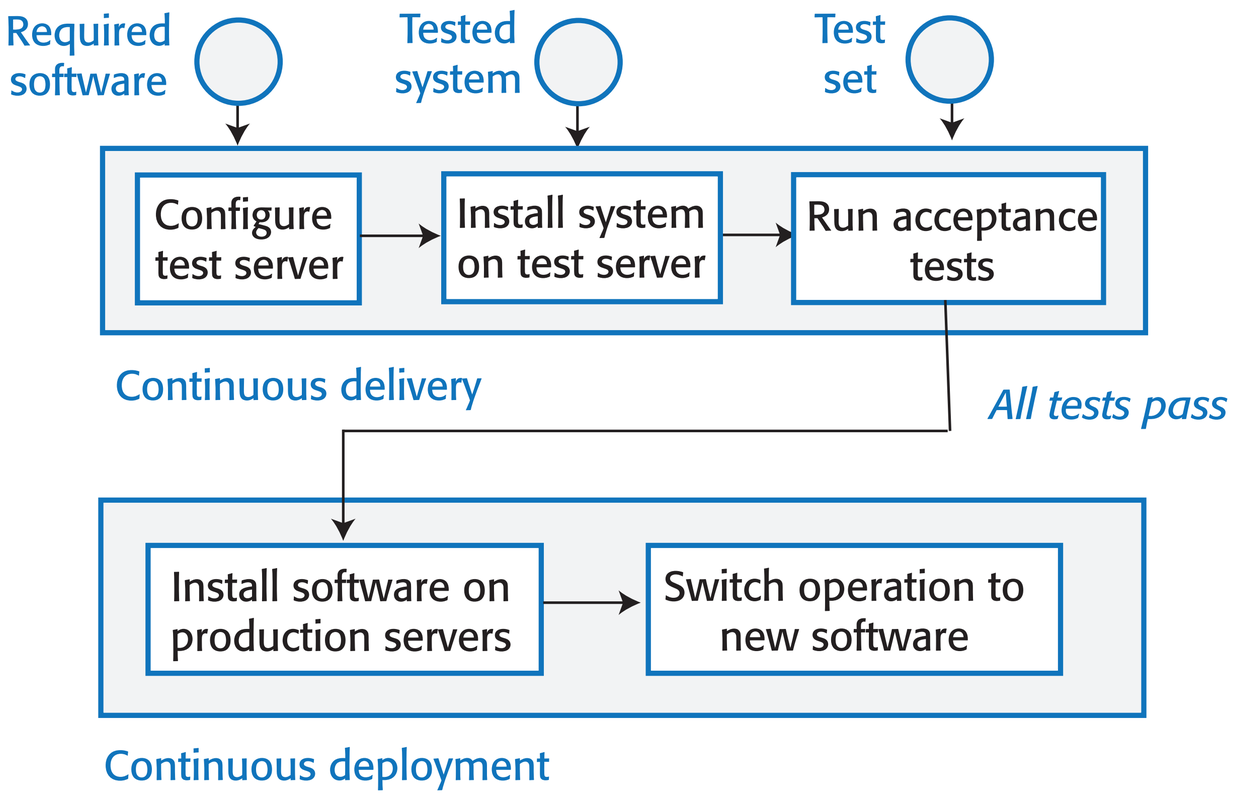


Figure 10.13 Continuous delivery and deployment

Deployment pipeline

Remember this?

After initial integration testing, a **staged test environment** is created: a replica of the actual production environment in which the system will run.

System acceptance tests (*e.g.* functionality, load and performance tests) are run to check the software works as expected.

If all of these tests pass, the changed software is installed on the production servers.

To **deploy** the system, you momentarily stop all new requests for service and leave the older version to process the outstanding transactions.

Once these have been completed, you **switch to the new version** of the system and **restart processing**.

Types of testing

Functional

Test the **functionality** of the **overall system**. The goals: discover as many bugs as possible and provide convincing evidence that the system is fit for its purpose

User

Test that the product is **useful to** and **usable by end-users**. Must show that the features **help users do what they want to do**. Also, that **users understand how to access the features** and can use them effectively

Performance & load

Test that the software **works quickly** and **can handle the expected load**. Must show that the **response** and **processing time** is acceptable to end-users. Also, show it can **handle different loads** and **scales gracefully** as the load increases

Security

Test that the software **maintains its integrity** and **can protect user information** from theft and damage

Types of testing

Functional

Test the **functionality** of the **overall system**. The goals: discover as many bugs as possible and provide convincing evidence that the system is fit for its purpose

User

Test that the product is **useful to** and **usable by end-users**. Must show that the features **help users do what they want to do**. Also, that **users understand how to access the features** and can use them effectively

Performance & load

Test that the software **works quickly** and **can handle the expected load**. Must show that the **response** and **processing time** is acceptable to end-users. Also, show it can **handle different loads** and **scales gracefully** as the load increases

Security

Test that the software **maintains its integrity** and **can protect user information** from theft and damage

Functional testing

Test the **functionality** of the **overall system**. The goals: discover as many bugs as possible and provide convincing evidence that the system is fit for its purpose…

It involves **developing** a large set of **program tests** so that all of a program's code is executed at least once

Number of tests? Depends on the size and the functionality of the application

Functional testing is a **staged activity** in which you **first** **test individual units of code**. You integrate **code units** with other units to create larger units then do more testing…

The process continues until you have created a complete system ready for release…

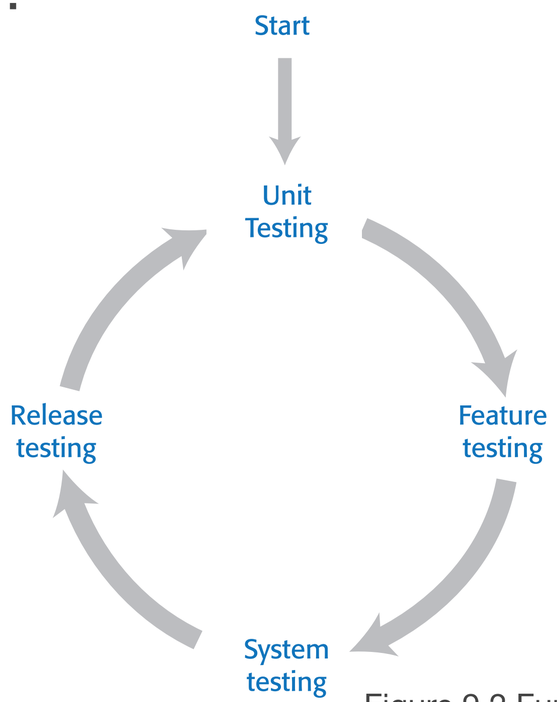


Figure 9.2 Functional testing process

Functional testing process

Unit testing: test program **units** **in isolation**. tests designed to execute all of the code in a unit at least once. Tested by the programmer as they are developing the code!

Feature testing: **Code units are integrated to create features.** Feature tests should test **all aspects of a feature**. All of the programmers who contribute code units to a feature should be involved in its testing!

System testing: **units are integrated to create a working** (even if incomplete) **version of a system**. Goal: check that there are **no unexpected interactions between the features** (may include checking the **responsiveness**, **reliability** and **security** of the system.

Release testing: system is packaged for release… tested to **check that it operates as expected**.

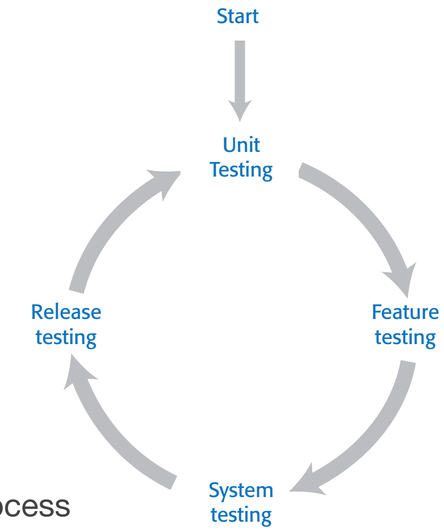


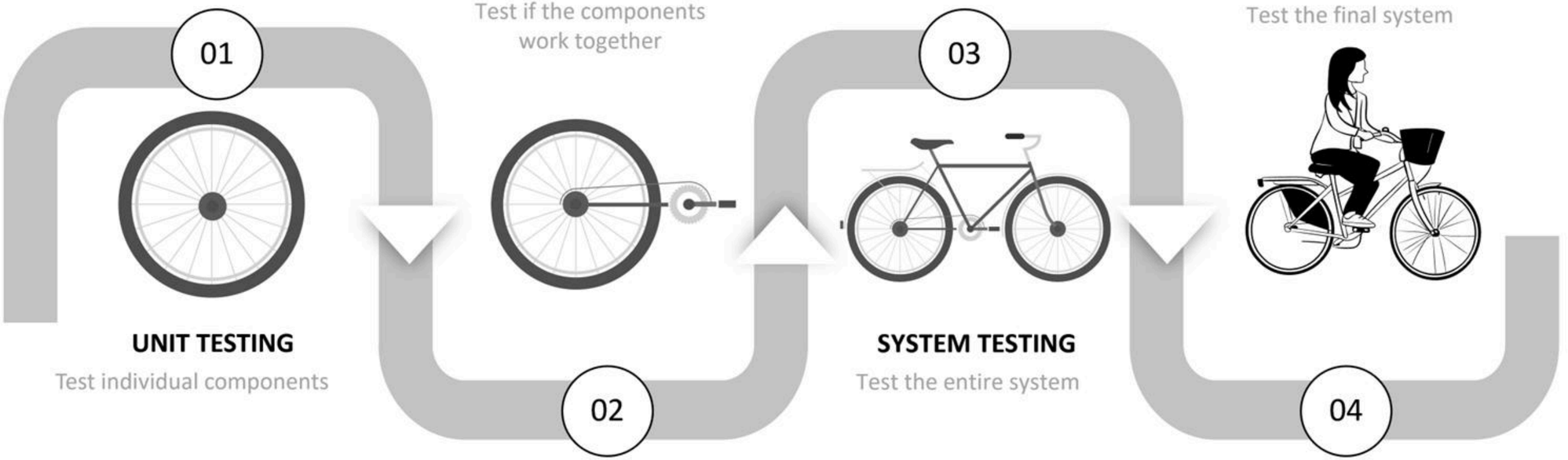
Figure 9.2 Functional testing process

Functional testing process

Sometimes the processes are named differently…

Verifying how different components work together and interact

Ensuring the system meets the user requirements provided at the start



Testing individual modules or components in isolation to verify their behaviour

Focus on the system as a whole and how the components are integrated and work together

Unit testing

Test program **units** **in isolation**. tests designed to execute all of the code in a unit at least once. Tested by the programmer as they are developing the code!

As you develop a **code unit**, you should also **develop tests** for that code!

Code unit: whatever has a clearly defined responsibility (usually a function or class method but could be a module that includes a small number of other functions)

To test a program efficiently: **identify sets of inputs** (equivalence partitions) that will be treated in the same way in your code.

The **equivalence partitions** that you identify should not just include those containing inputs that produce the correct values. **You should also identify incorrectness partitions where the inputs are deliberately incorrect!**

If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.

Unit testing with **Jest**

A simple function that adds two numbers…

Use **Jest** to unit test the function using different values

`expect` takes the input to test with and `.toBe` tells us what the output should be

JS utils.js > ...

```
1   // utils.js
2   function add(a, b) {
3     return a + b;
4   }
5
```

JS utils.test.js > ...

```
1   // utils.test.js (using Jest)
2   const { add } = require('./utils');
3
4   test('adds two numbers correctly', () => {
5     expect(add(2, 3)).toBe(5);
6     expect(add(-1, 1)).toBe(0);
7   })
8
```

Jest?

…is a **JavaScript testing framework** designed for simplicity and efficiency

Commonly used to test JavaScript and Node.js applications, focusing on unit testing, but it can also handle integration and snapshot testing…

JS utils.js > ...

```
1   // utils.js
2   function add(a, b) {
3     return a + b;
4   }
5
```

JS utils.test.js > ...

```
1   // utils.test.js (using Jest)
2   const { add } = require('./utils');
3
4   test('adds two numbers correctly', () => {
5     expect(add(2, 3)).toBe(5);
6     expect(add(-1, 1)).toBe(0);
7   })
8
```

Ways of testing

Black-box

Focus is on the **application's functionality** without considering internal implementation details

Tests the external behaviour of the software without any knowledge of the code (treats code like a black-box)

White-box

Examines the internal structure, logic, and design of the application to ensure the system internal operations are expected

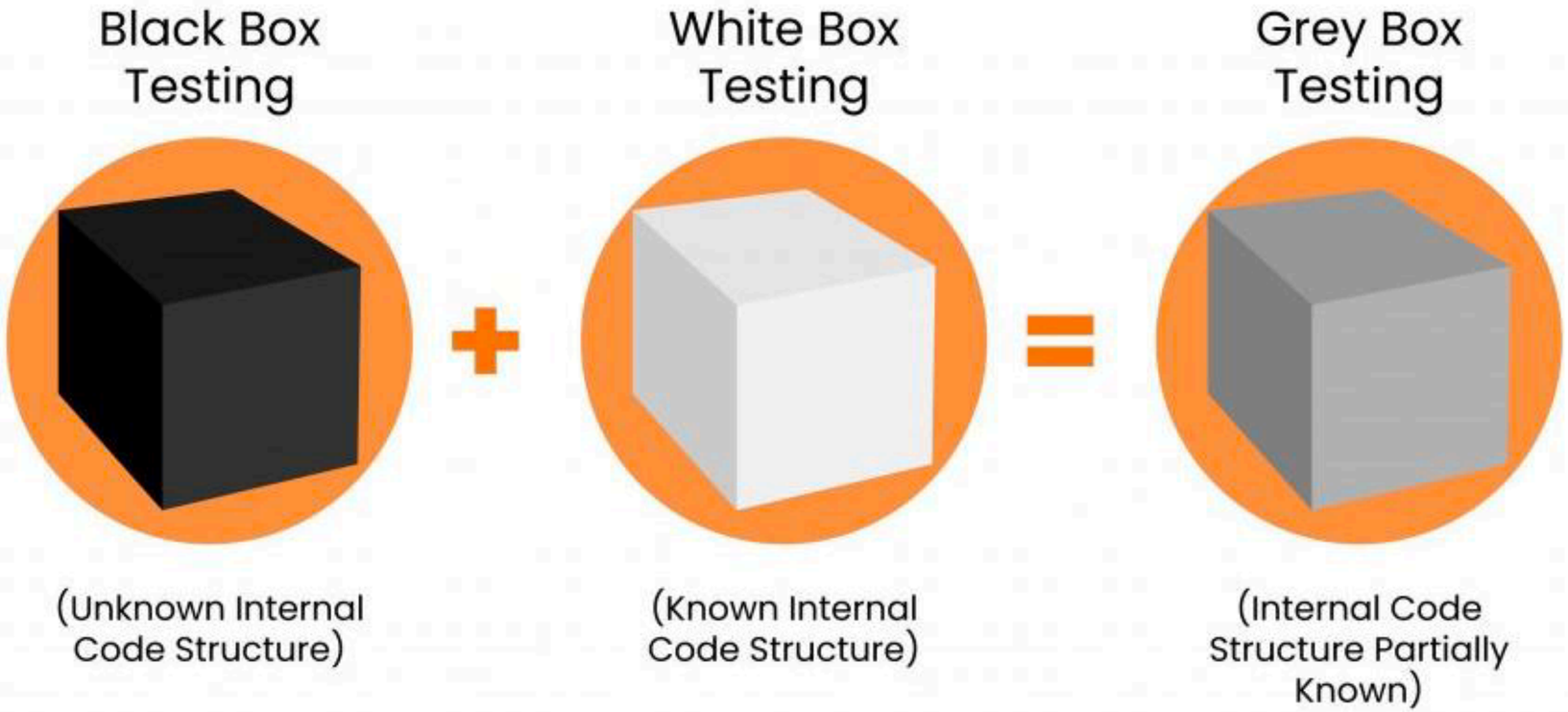
The tester knows the internals of the code

Grey-box

Tests individual modules or components in isolation to verify their behaviour.

Some knowledge is known about the system internal operation and codebase

Ways of testing



Ways of testing: **Black-box** testing

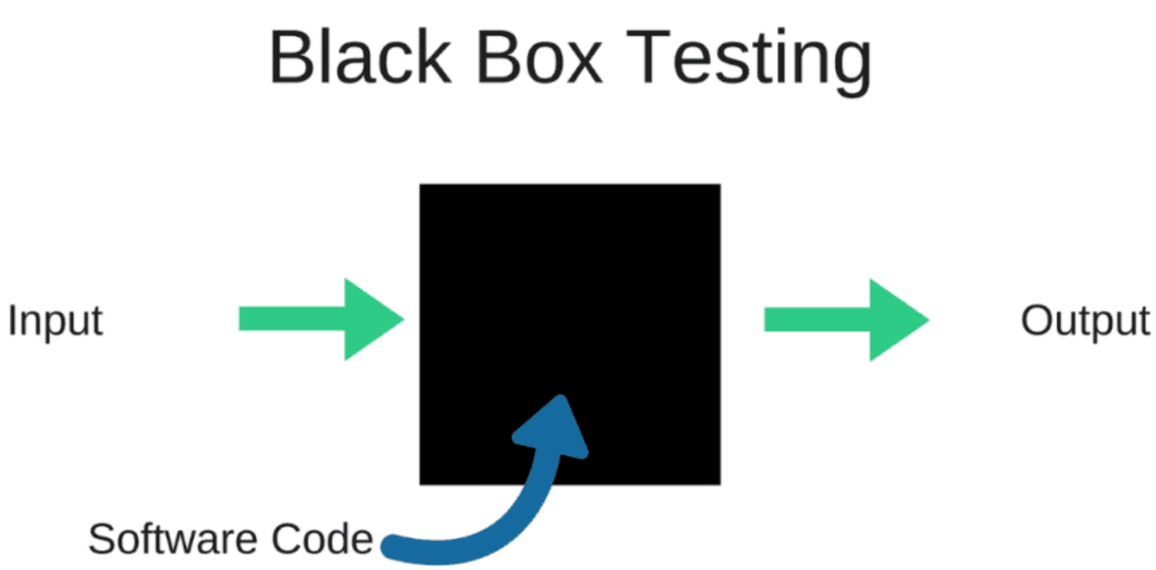
Focus is on the **application's functionality** without considering internal implementation details

Tests the external behaviour of the software without any knowledge of the code (treats code like a black-box)

The tester uses the system like an end user.

We provide input and observe output without knowing how the inputs are handled internally

User-centric



e.g. user checking a website's login page for functionality

Black-Box

```
// Example API
app.post('/api/login', (req, res) => {
  const { username, password } = req.body;
  if (username === 'admin' && password === 'password123') {
    res.status(200).json({ message: 'Login successful' });
  } else {
    res.status(401).json({ message: 'Unauthorized' });
  }
});
```

Test (using Postman or a similar tool)

```
Input: { "username": "admin", "password": "password123" }
Expected Output: { "message": "Login successful" }
```

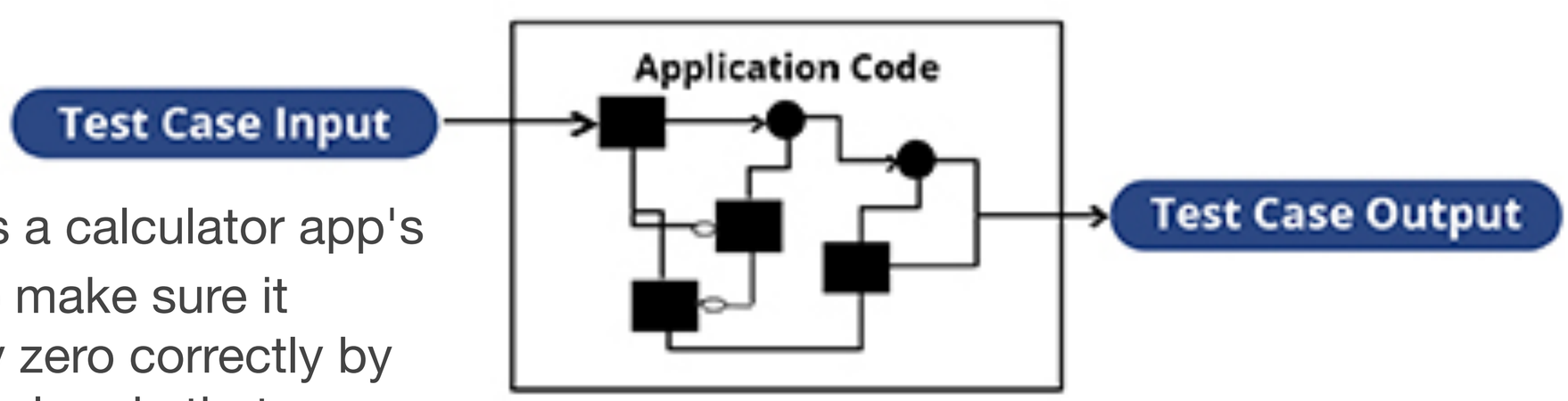
Ways of testing: **White-box** testing

Examines the internal structure, logic, and design of the application to ensure the system internal operations are expected

The tester knows the internals of the code

aka **glass-box**, **clear-box**, or **transparent-box**

Code-centric



e.g. developer tests a calculator app's division function to make sure it handles division by zero correctly by checking the internal code that implements the check.

White-Box

```
function calculateDiscount(price, userType) {
  if (userType === 'premium') {
    return price * 0.8;
  } else if (userType === 'regular') {
    return price * 0.9;
  } else {
    return price;
  }
}
```

White-box Test Cases

```
console.log(calculateDiscount(100, 'premium')); // Expected: 80
console.log(calculateDiscount(100, 'regular')); // Expected: 90
console.log(calculateDiscount(100, 'guest'));   // Expected: 100
```

Ways of testing: **Grey-box** testing

Tests individual modules or components in isolation to verify their behaviour.

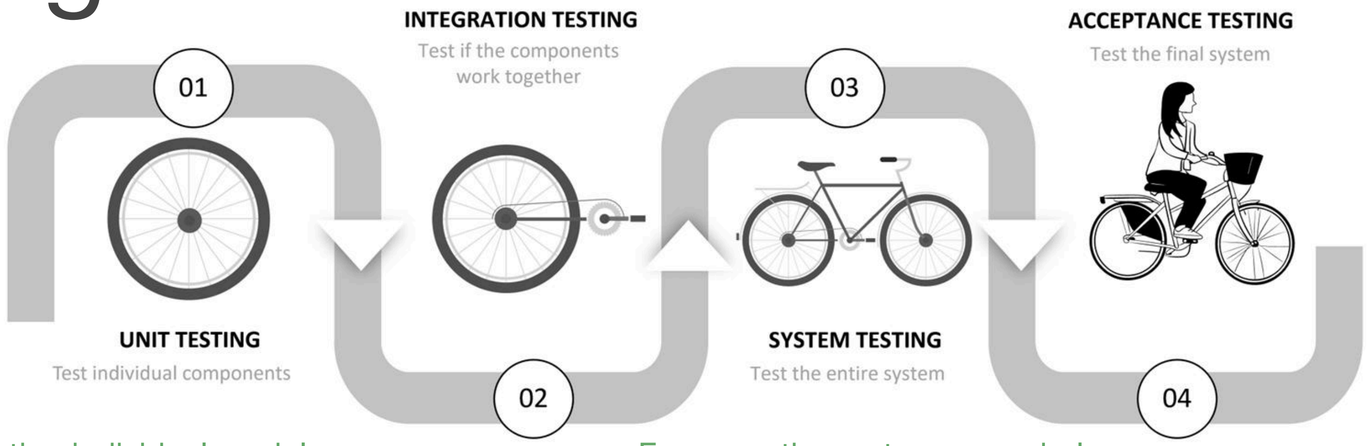
Some knowledge is known about the system internal operation and codebase

e.g. tester **with access to a database's schema** validates that the correct data is being stored after a user makes an entry via the web interface, **even without seeing** the application's source code.

Integration testing

Verifying how different components work together and interact

Ensuring the system meets the user requirements provided at the start



Testing individual modules or components in isolation to verify their behaviour

Focus on the system as a whole and how the components are integrated and work together

Top-down

Start from the top of the hierarchy

Bottom-up

Start from the lowest-level modules

Big-bang

All modules tested together at once (risky!!)

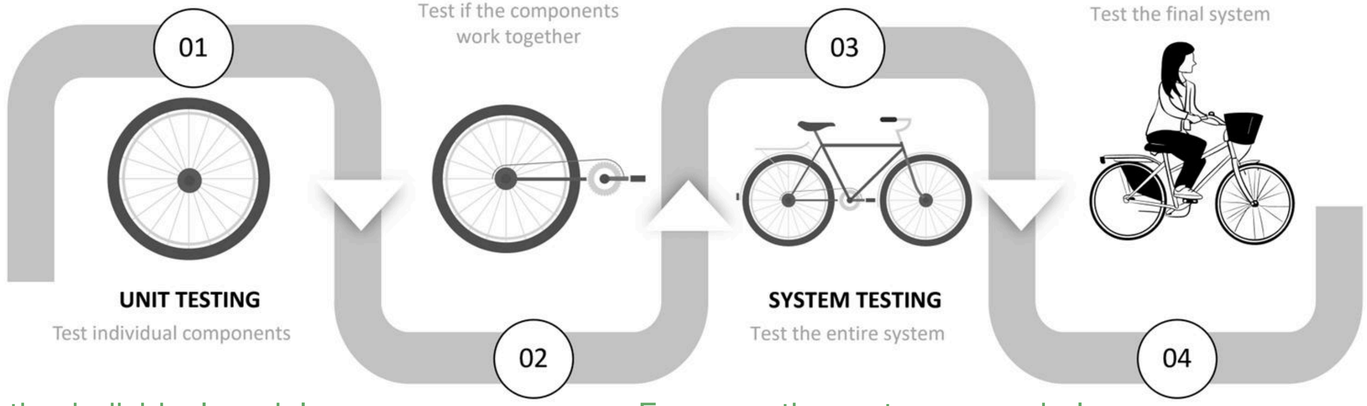
Sandwich (hybrid)

Combines top-down and bottom-up

End-to-end (E2E) testing

Verifying how different components work together and interact

Ensuring the system meets the user requirements provided at the start



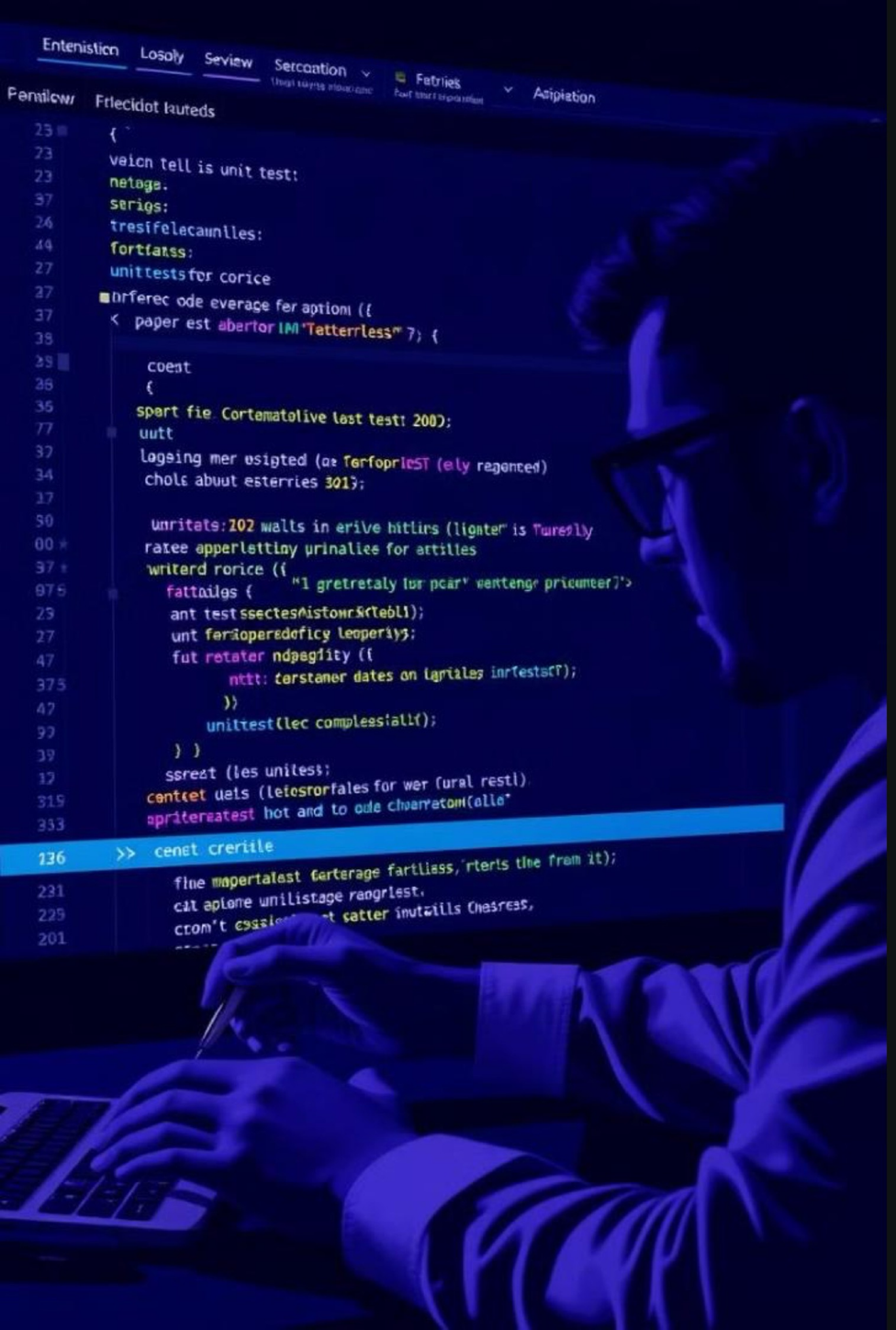
Testing individual modules or components in isolation to verify their behaviour

Focus on the system as a whole and how the components are integrated and work together

Simulates real user scenarios across the entire system (from UI to database)
Used to verify the full workflow: *e.g.* login → order → payment → confirmation

E2E tests are slower, require full system setup, are fragile (break easily), and need more resources (*e.g.* test environments, databases)…

Therefore: more expensive!



Unit Testing with Jest or Mocha

1 Write Testable Code

Ensure your code is modular and loosely coupled to enable effective unit testing.

2 Choose a Testing Framework

Popular options include Jest and Mocha, which provide assertion libraries and mocking capabilities.

3 Test Driven Development (TDD)

Write your tests first, then implement the functionality to pass the tests.

Real-World Example: Testing a Node.js Express API

Unit Tests

Verify the behavior of individual Express route handlers and middleware functions using Jest.

Integration Tests

Test the API endpoints and data flow between the Express application and a mocked database using Supertest.

End-to-End Tests

Simulate user interactions with the API and the frontend application using Cypress.io.

Developing Test Cases and Scenarios

Identify Critical Functionality

Focus on the core features and user journeys that are essential to your application's success.

Explore Edge Cases

Consider unexpected inputs, error conditions, and boundary cases that could impact your application's stability.

Collaborate with Stakeholders

Work closely with product owners, designers, and developers to ensure your test cases cover the right scenarios.

Test it yourself

- Write a unit test for a function that calculates the area of a rectangle.
- Create a black-box test for an API that retrieves a list of books.

Install Jest

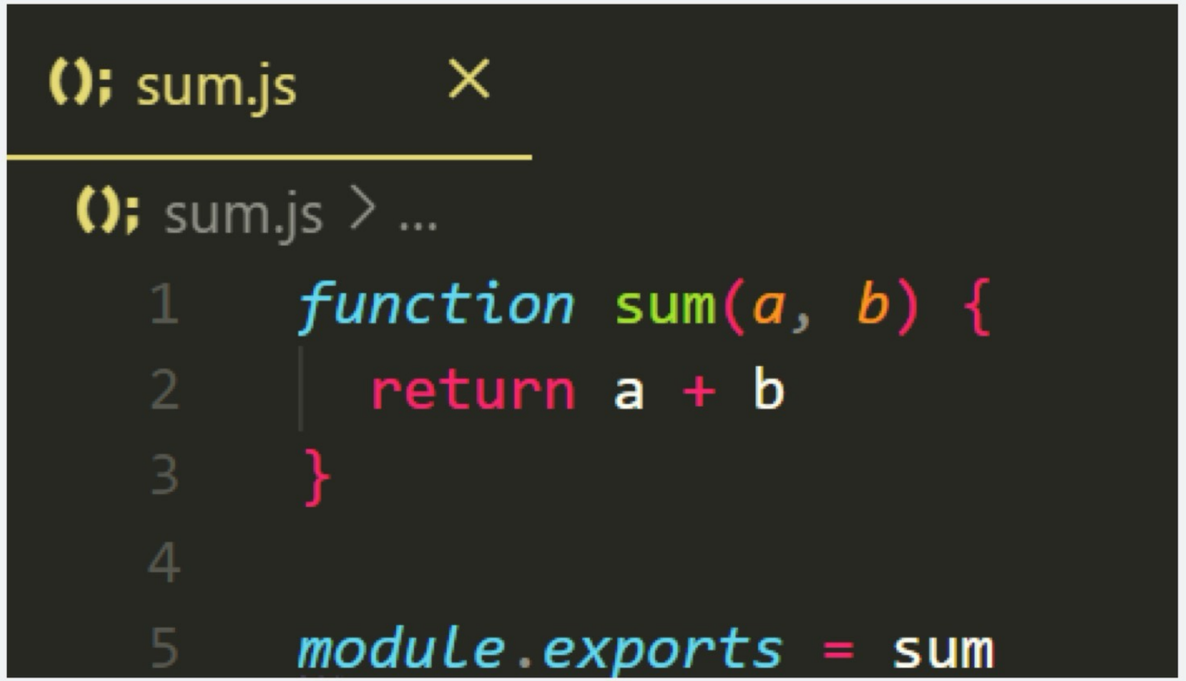
- Create a directory in VSCode.
- Open a terminal in your project directory.
- Initiate a project using “npm init –y” command
- Install Jest using npm: `npm install jest`
- Add a test script in your “package.json” file:

```
10    "scripts": {
11      "test": "jest",
12      "start": "node server.js"
13    },
```

Write your first test

1. Create a file to test

Example: sum.js

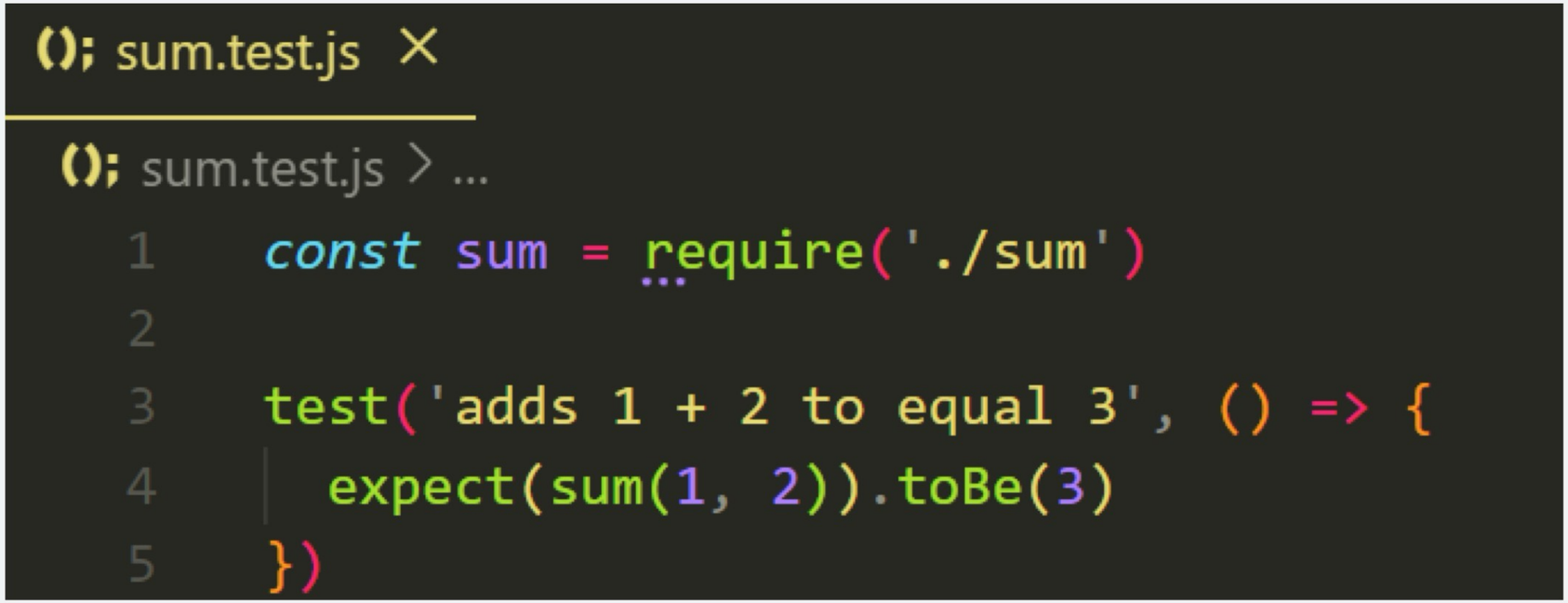


Write your first test

2. Create a test file

Jest *conventionally* looks for files ending in ***.test.js***

Example: sum.test.js

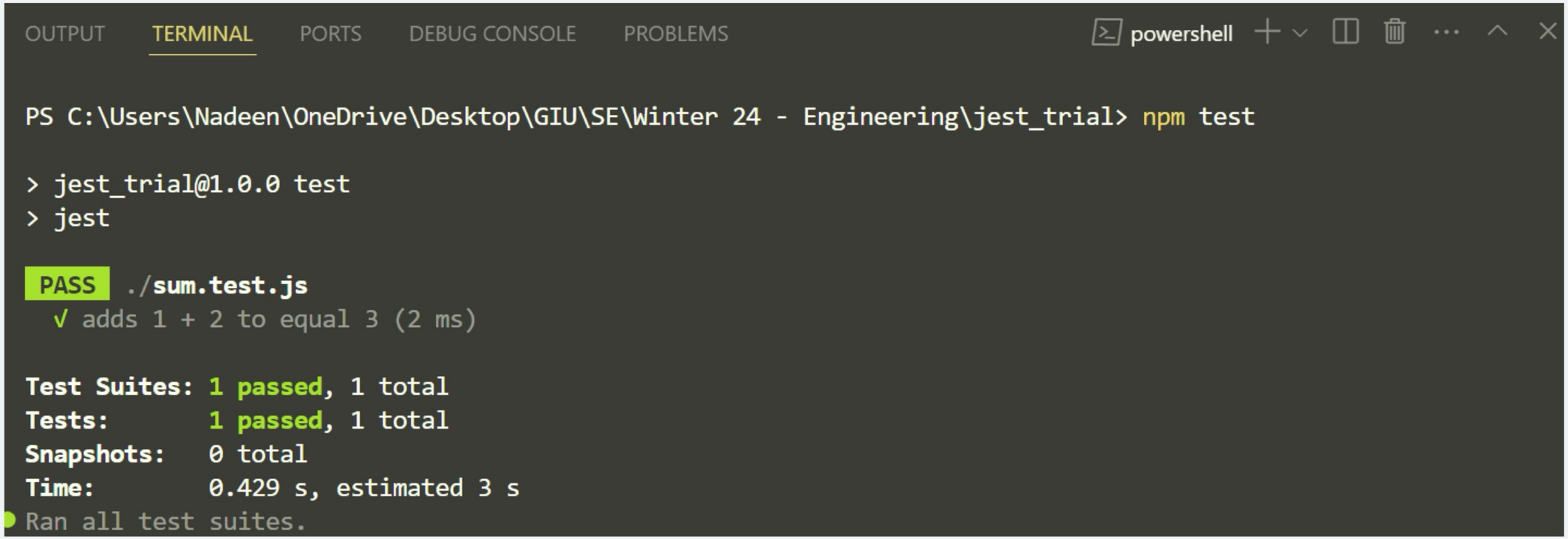


Write your first test

3. Run the test

Use the command: npm test

The result:



Understand basic concepts

- ***test()*** function: defines a test case

```
test('description of the test case', () => {
  //test logic
})
```

- ***Matchers***: verify the output, examples:

  toBe(): Checks exact equality

  toEqual(): Checks deep equality for objects/arrays

  toBeTruthy(): Checks if a value is truthy

  toBeFalsy(): Checks if a value is falsy

Understand basic concepts

- Mocking & Asynchronous testing:

1. **Mocking functions**: Replace real implementations during tests

```
const fetchData = jest.fn(() => 'data');
expect(fetchData()).toBe('data');
```

2. **Testing Async code**:

```
const fetchData = async () => 'data';

test('fetches data successfully', async () => {
  const data = await fetchData();
  expect(data).toBe('data');
});
```

Testing a simple server project

1. Setup the project:

mkdir jest-trial

cd jest-trial

npm init –y

npm install express jest supertest

Jest: for testing

Supertest: for testing HTTP endpoints

Testing a simple server project

2. Create the server

Download the attached project & check ***server.js, routes/index.js*** files

3. Write tests

Check ***server.test.js*** file

4. Run the tests: ***npm test***

Testing a simple server project

Output: