

VNU-HCM University Of Science  
Faculty of Information Technology



# UNIT TESTING

## CSC13010 – Software Design

### INSTRUCTORS

Mr. Trần Duy Thảo

Mr. Hồ Tuấn Thanh

Mr. Nguyễn Lê Hoàng Dũng

### STUDENT

Lâm Tiến Huy

22127151

Ho Chi Minh City, 2025

# Table of Contents

Table of Contents.....	2
1. Draw the maze and define the problem clearly.....	Error! Bookmark not defined.
2. Depth-first Search and Breadth-first Search.....	Error! Bookmark not defined.
2.1 Pseudo-code For Concept.....	Error! Bookmark not defined.
2.2 Time and Space Complexity .....	Error! Bookmark not defined.
2.3 Visualization .....	Error! Bookmark not defined.
3. Heuristic Function and A* Search.....	Error! Bookmark not defined.
3.1 Pseudo-code for Concept.....	Error! Bookmark not defined.
3.1 Time and Space Complexity .....	Error! Bookmark not defined.
3.2 Visualization .....	Error! Bookmark not defined.
4. Objective Function and Local Search .....	Error! Bookmark not defined.
4.1 Pseudo-code for Concept.....	Error! Bookmark not defined.
4.2 Time and Space Complexity .....	Error! Bookmark not defined.
4.2 Visualization .....	Error! Bookmark not defined.
5. A* Search for Both Goals.....	Error! Bookmark not defined.
5.1 Time and Space Complexity .....	Error! Bookmark not defined.
5.2 Visualization .....	Error! Bookmark not defined.

# Chapter 1

# Definition

## 1.1 Unit Testing

According to GeeksForGeeks (2024), **Unit Testing** is the process of testing the smallest parts of your code, like individual functions or methods, to make sure they work correctly. It's a key part of software development that improves code quality by testing each unit in isolation. These units are the smallest pieces of code, typically functions or methods, ensuring they perform as expected.

Unit testing helps in identifying bugs early in the development cycle, enhancing code quality, and reducing the cost of fixing issues later. It is an essential part of **Test-Driven Development (TDD)**, promoting reliable code. Unit tests are written for these code units and run them automatically every time you make changes. If a test fails, it helps you quickly find and fix the issue. Unit testing promotes modular code, ensures better test coverage, and saves time by allowing developers to focus more on coding than manual testing.

## 1.2 Unit Test

Stated by AWS, A **unit test** is a block of code that verifies the accuracy of a smaller, isolated block of application code, typically a function or method. The unit test is designed to check that the block of code runs as expected, according to the developer's theoretical logic behind it. The unit test is only capable of interacting with the block of code via inputs and captured asserted (true or false) output.

A single block of code may also have a set of unit tests, known as **test cases**. A complete set of test cases cover the full expected behavior of the code block, but it's not always necessary to define the full set of test cases.

When a block of code requires other parts of the system to run, you can't use a unit test with that external data. The unit test needs to run in isolation. Other system data, such as databases, objects, or network communication, might be required for the code's functionality. If that's the case, you should use data stubs instead. It's easiest to write unit tests for small and logically simple blocks of code.

## 1.3 Simple Examples

A unit can be almost anything you want it to be – a line of code, a method, or a class (SmartBear). Generally though, smaller is better. Smaller unit tests give you a much more granular view of how your code is performing. There is also the practical aspect that when you test very small units, your tests can be run fast; like a thousand tests in a second fast.

*Consider this sample code:*

```
def divider (a, b)
  return a/b
end
```

*Using Ruby, those small tests might look something like this:*

```
class smallTest < MiniTest::Unit::testCase
  def tiny_test
    @a=9
    @b=3
    assert_equal(3, divider(a, b))
  end
end
```

This example is too simple, but it gives you an idea of what we mean by small. Small unit tests also have the benefit of making it harder to cross systems, from code into a database, or third-party system. Strictly speaking, there isn't anything wrong with crossing systems, but there are consequences, such as gradually slowing your tests.

## 1.4 Advantages and Disadvantages

### 1.4.1 Advantages

1. **Early Detection of Issues:** Unit testing allows developers to detect and fix issues early in the development process before they become larger and more difficult to fix.
2. **Improved Code Quality:** Unit testing helps to ensure that each unit of code works as intended and meets the requirements, improving the overall quality of the software.
3. **Increased Confidence:** Unit testing provides developers with confidence in their code, as they can validate that each unit of the software is functioning as expected.

4. **Faster Development:** Unit testing enables developers to work faster and more efficiently, as they can validate changes to the code without having to wait for the full system to be tested.
5. **Better Documentation:** Unit testing provides clear and concise documentation of the code and its behavior, making it easier for other developers to understand and maintain the software.
6. **Facilitation of Refactoring:** Unit testing enables developers to safely make changes to the code, as they can validate that their changes do not break existing functionality.
7. **Reduced Time and Cost:** Unit testing can reduce the time and cost required for later testing, as it helps to identify and fix issues early in the development process.

#### ***1.4.2 Disadvantages***

1. **Time and Effort:** Unit testing requires a significant investment of time and effort to create and maintain the test cases, especially for complex systems.
2. **Dependence on Developers:** The success of unit testing depends on the developers, who must write clear, concise, and comprehensive test cases to validate the code.
3. **Difficulty in Testing Complex Units:** Unit testing can be challenging when dealing with complex units, as it can be difficult to isolate and test individual units in isolation from the rest of the system.
4. **Difficulty in Testing Interactions:** Unit testing may not be sufficient for testing interactions between units, as it only focuses on individual units.
5. **Difficulty in Testing User Interfaces:** Unit testing may not be suitable for testing user interfaces, as it typically focuses on the functionality of individual units.
6. **Over-reliance on Automation:** Over-reliance on automated unit tests can lead to a false sense of security, as automated tests may not uncover all possible issues or bugs.
7. **Maintenance Overhead:** Unit testing requires ongoing maintenance and updates, as the code and test cases must be kept up-to-date with changes to the software.

## Chapter 2

# Test Coverages & Best Practices

### 2.1 Code Coverage

Quoting Atlassian, **Code coverage** is a metric that can help you understand how much of your source is tested. It's a very useful metric that can help you assess the quality of your test suite, and we will see here how you can get started with your projects. The common metrics that you might see mentioned in your coverage reports include:

- **Function coverage:** how many of the functions defined have been called.
- **Statement coverage:** how many of the statements in the program have been executed.
- **Branches coverage:** how many of the branches of the control structures (if statements for instance) have been executed.
- **Condition coverage:** how many of the boolean sub-expressions have been tested for a true and a false value.
- **Line coverage:** how many of lines of source code have been tested.

These metrics are usually represented as the number of items actually tested, the items found in your code, and a coverage percentage (items tested or items found).

### 2.2 Test Coverage

#### *2.2.1 Definition*

According to Topdev, **Test coverage** is defined as a technique that determines whether test cases actually cover the application code and how much code is executed when running those test cases.

If there are 10 requirements and 100 tests created and if 90 tests are executed then the test coverage is 90%. Now based on this figure the tester can create additional test cases for the remaining tests. Here are some advantages of test coverage.

- You can identify gaps in requirements, test cases, and bugs at early and code levels.
- You can prevent unwanted defect leakage by using test coverage analysis.
- Test coverage also helps in regression testing, test case prioritization, test suite augmentation and test suite minimization.

### 2.2.2 Metrics

Test coverage metrics can be divided into three parts: code-level metrics, feature testing metrics, and application-level metrics.

- *Code level data*

It is also known as executed tests which is calculated as the percentage of executed/executed testcases out of total testcases.

$$\begin{array}{l} \text{Executed Tests} \\ \text{Or} \\ \text{Test Execution} \\ \text{Coverage Percentage} \end{array} = \frac{\text{Number of tests run}}{\text{Total number of tests to be run}} \times 100\%$$

- *Feature testing metrics*

**Required coverage:** Requirement coverage is used to determine how well the test cases cover the software requirements. For that, you simply divide the number of covered requirements by the total number of requirements in scope for a sprint, release, or project.

$$\begin{array}{l} \text{Requirements} \\ \text{Coverage} \end{array} = \frac{\text{Number of requirements covered}}{\text{Total number of requirements}} \times 100\%$$

**Test cases on demand:** This metric is used to see what features are being tested and how many tests are appropriate for the requirements. Most requirements contain multiple test cases. It is very important to know which test cases are failing for a particular requirement so that test cases can be rewritten for other specific requirements.

Request	Test Case	Test Result
Request 1	Test Case 1	Pass
Request 2	Test Case 2	Failed
Request 3	Test Case 3	Incomplete

This metric is important for stakeholders as it shows the progress of application or software development.

- *Application-level metrics*

**Defect density:** Defect density is a measure of the total number of known defects divided by the size of the software entity being measured.

$$\text{Defect Density} = \frac{\text{Number of known defects}}{\text{Size of software entity}}$$

It is used to identify areas that require automation. If defect density is high for specific functionality it requires retesting. To reduce retesting efforts, test cases for known defects can be automated. It is important to consider the priority of the defect (low or high) while evaluating the defects.

**External Requirements Test Coverage:** After calculating the scope of requirements, you will find some requirements that are not covered. Now, it is important to know about each requirement that is not covered and what stage the requirement is in.



Request ID	Request Name	Request Status
Request 001	Request A	To Do
Request 002	Request B	Done

This metric helps engineers and developers identify and eliminate unexplored requirements from the total requirements before they send them to production.

## 2.3 Coverage Aims

### 2.3.1 *Goals in coverage*

To Atlassian, there is no silver bullet in code coverage, and a high percentage of coverage could still be problematic if critical parts of the application are not being tested, or if the existing tests are not robust enough to properly capture failures upfront. With that being said it is generally accepted that 80% coverage is a good goal to aim for. Trying to reach a higher coverage might turn out to be costly, while not necessary producing enough benefit.

For instance, in the example above we reached 100% coverage by testing if 100 and 34 were multiples of 10. But what if we called our function with a letter instead of a number? Should we get a true/false result? Or should we get an exception? It is important that you give time to your team to think about testing from a user perspective and not just by looking at lines of code. Code coverage will not tell you if you're missing things in your source.

### 2.3.2 *Unit testing priority*

Unit tests consist in making sure that the individual methods of the classes and components used by your application are working. They're generally cheap to implement and fast to run and give you an overall assurance that the basis of the platform is solid. A simple way to increase quickly your code coverage is to start by adding unit tests as, by definition, they should help you make sure that your test suite is reaching all lines of code.

### 2.3.3 *Coverage reports*

Soon you'll have so many tests in your code that it will be impossible for you to know what part of the application is checked during the execution of your test suite. You'll know what breaks when you get a red build, but it'll be hard for you to understand what

components have passed the tests. This is where the coverage reports can provide actionable guidance for your team.

#### ***2.3.4 Good coverage and good tests***

Getting a great testing culture starts by getting your team to understand how the application is supposed to behave when someone uses it properly, but also when someone tries to break it. Code coverage tools can help you understand where you should focus your attention next, but they won't tell you if your existing tests are robust enough for unexpected behaviors.

### **2.4 Best Practices**

- Be **easy to maintain** and **self-contained** (using mocks or stubs instead of real databases or persistent storage).
- Cover both **happy cases** and **edge cases**.
- Focus on testing behavior rather than internal implementation details to avoid brittle tests.
- Be integrated into CI/CD pipelines so that coverage is continuously monitored and maintained.

## Chapter 3

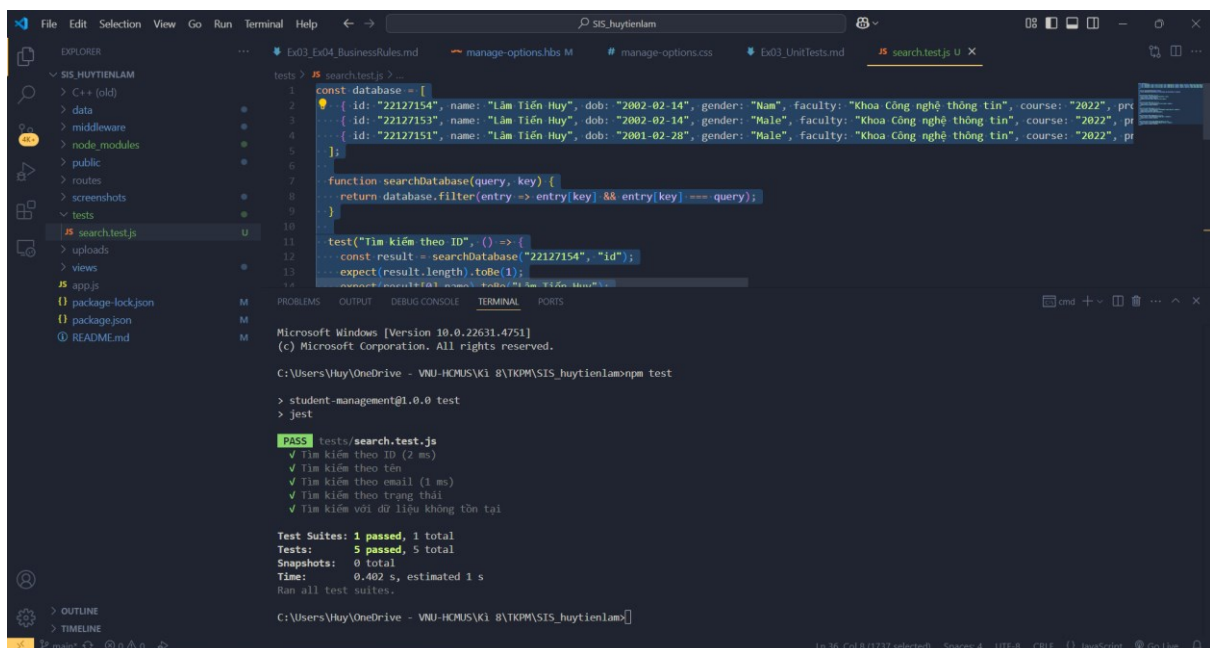
# Testing My Product

### 3.1 Environment

- Jest will be used for testing my project.
- Run `npm install --save-dev jest` to install the Jest tool.
- Create a folder `/tests` and store all the test files.
- The files have to be in the format `name.test.js` for Jest to recognize.
- Run `npm test` from the terminal to start unit testing phase.

### 3.2 Search Test

- File: `search.test.js`
- Test cases include:
  - Searching by ID (ensuring only one result is returned).
  - Searching by name (ensuring all matching results are returned).
  - Searching by email (ensuring the correct result is returned).
  - Searching by status (ensuring all matching results are returned).
  - Searching with non-existent data (ensuring an empty array is returned).
- Result: Pass



The screenshot shows a VS Code editor with a file named `search.test.js` open. The file contains a `const database` array with three objects and a `searchDatabase` function. A test case is written using Jest's `test` and `expect` functions. The terminal at the bottom shows the command `npm test` being executed, resulting in a 'PASS' status for the test suite.

```
const database = [
  { id: "22127154", name: "Lâm Tiến Huy", dob: "2002-02-14", gender: "Nam", faculty: "Khoa Công nghệ thông tin", course: "2022", pr
  { id: "22127153", name: "Lâm Tiến Huy", dob: "2002-02-14", gender: "Male", faculty: "Khoa Công nghệ thông tin", course: "2022", pr
  { id: "22127151", name: "Lâm Tiến Huy", dob: "2001-02-28", gender: "Male", faculty: "Khoa Công nghệ thông tin", course: "2022", pr
];

function searchDatabase(query, key) {
  return database.filter(entry => entry[key] && entry[key] === query);
}

test("Tìm kiếm theo ID", () => {
  const result = searchDatabase("22127154", "id");
  expect(result.length).toBe(1);
});
```

```
Microsoft Windows [Version 10.0.22631.4751]
(c) Microsoft Corporation. All rights reserved.

C:\Users\huy\OneDrive - VNU-HCM\VS\1_8\TKPM\SIS_huytienlam>npm test

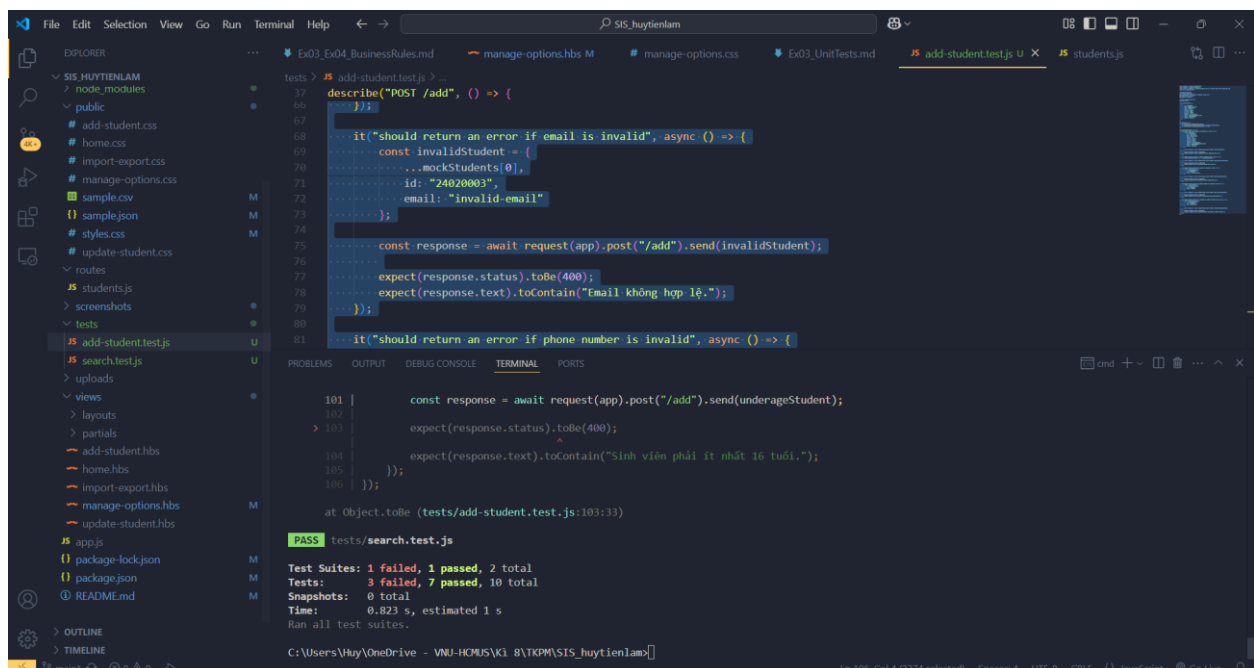
> student-management@1.0.0 test
> jest

PASS tests/search.test.js
  ✓ Tìm kiếm theo ID (2 ms)
  ✓ Tìm kiếm theo tên
  ✓ Tìm kiếm theo email (1 ms)
  ✓ Tìm kiếm theo trạng thái
  ✓ Tìm kiếm với dữ liệu không tồn tại

Test Suites: 1 passed, 1 total
Tests: 5 passed, 5 total
Snapshots: 0 total
Time: 0.402 s, estimated 1 s
Ran all test suites.
```

### 3.2 Add Students Test

- File: **add-student.test.js**
- Test cases include:
  - Student addition.
  - Student ID error.
  - Email error.
  - Phone number error.
  - Underaged students error.
- Result: Fail



```
describe("POST /add", () => {
  // ...
  it("should return an error if phone number is invalid", async () => {
    const invalidStudent = {
      id: "24020003",
      email: "invalid-email",
      phone: "1234567890",
    };
    const response = await request(app).post("/add").send(invalidStudent);
    expect(response.status).toBe(400);
    expect(response.text).toContain("Số điện thoại không hợp lệ.");
  });
});

it("should return an error if phone number is invalid", async () => {
  const response = await request(app).post("/add").send(underageStudent);
  expect(response.status).toBe(400);
  expect(response.text).toContain("Sinh viên phải ít nhất 16 tuổi.");
});
});
```

at Object.toBe (tests/add-student.test.js:103:33)

**PASS** tests/search.test.js

Test Suites: 1 failed, 1 passed, 2 total  
Tests: 3 failed, 7 passed, 10 total  
Snapshots: 0 total  
Time: 0.823 s, estimated 1 s  
Ran all test suites.

C:\Users\Huy\OneDrive - VNU-HCM\VS\8\TKPM\SIS\_huytienlam

### 3.3 More Tests Incoming

### 3.4 Verdict

Some of the features are working well. Some are not yet testable because of the logic of not including an error code but more of a notification validation. Test document will be updated in next versions.

# References

*What is unit testing?* (n.d.). smartbear.com. <https://smartbear.com/learn/automated-testing/what-is-unit-testing/>

*What is Unit Testing? - Unit Testing Explained - AWS.* (n.d.). Amazon Web Services, Inc. [https://aws.amazon.com/what-is/unit-testing/?nc1=h\\_ls](https://aws.amazon.com/what-is/unit-testing/?nc1=h_ls)

GeeksforGeeks. (2024, October 22). *Unit testing software testing.* GeeksforGeeks. <https://www.geeksforgeeks.org/unit-testing-software-testing/>

IEvangelist. (2024, January 30). *Use code coverage for unit testing - .NET.* Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-code-coverage?tabs=windows>

Atlassian. (n.d.). *What is Code Coverage? | Atlassian.* <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>

TopDev. (2022, January 25). *Tại sao Test Coverage là một phần quan trọng của Kiểm thử phần mềm?* TopDev. <https://topdev.vn/blog/tai-sao-test-coverage-la-mot-phan-quan-trong-cua-kiem-thu-phan-mem/>

Bose, S. (2023, May 4). *Code Coverage vs Test Coverage | BrowserStack.* BrowserStack. <https://www.browserstack.com/guide/code-coverage-vs-test-coverage>