

Unit testing

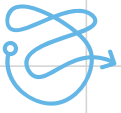
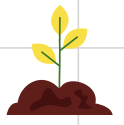
Principles, Practices, and Patterns



by Vladimir Khorikov

Goal of Unit testing

Enable **sustainable growth** of the software project



Project without tests

- Quickly slows down
- Hard to make any progress

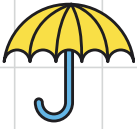


Fight entropy

- Constant cleaning and refactoring
- Tests act as a safety net

What makes a successful test suite?

Not all tests are created equal



- Integrated into the development cycle
- Targets most important parts of the code base
- Provides maximum value
 - With minimum maintenance costs



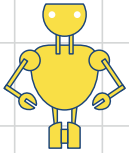
Bad tests : raise false alarms



- Unit tests are vulnerable to bugs
- Require maintenance

A tool that provides insurance against a vast majority of regressions

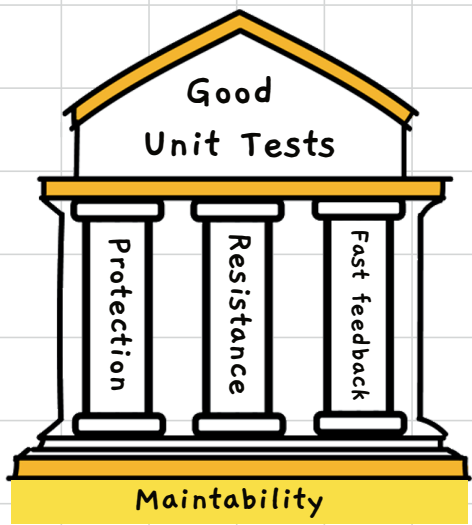
Tests are code too
View them as part of your code base that aims at solving a particular problem: ensuring the application's correctness



Automated test that :

- Verifies a **small piece of code** (also known as a **unit**)
- Does it **quickly**
- And does it in an **isolated** manner.

What is a unit test ?



Protection against regressions

- A regression = a software bug
- The larger the code base -> the more exposure to potential bugs
- Tests should reveal those regressions

Resistance to refactoring

The degree to which a test can sustain a refactoring of the underlying application code without turning red (failing)

Fast feedback

The more of them you can :

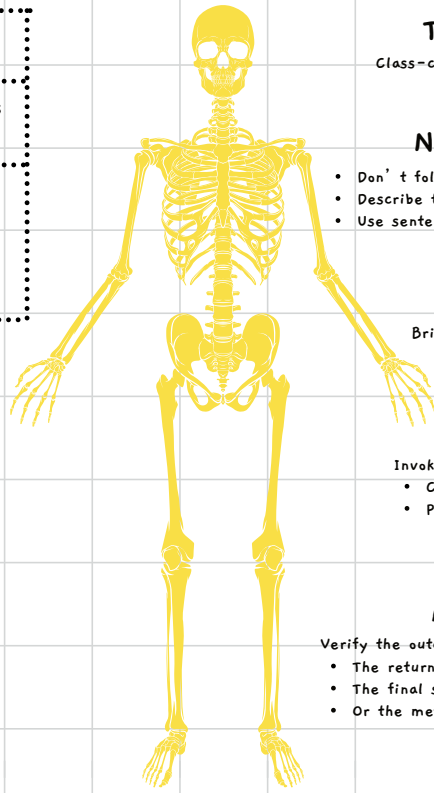
- Have in the suite
- Run them -> shorten the feedback loop

Maintainability (Maintenance costs)

- How hard it is to understand the test
- How hard it is to run the test

	London School	Classical School
A unit is	A class	<ul style="list-style-type: none"> • A class or set of classes • Behavior
Test doubles FOR	All but immutable dependencies	Shared dependencies

Anatomy



Test class name

Class-container for a cohesive set of tests

Name of the test

- Don't follow a rigid naming policy
- Describe the scenario to a non-programmer
- Use sentences

Arrange / Given

Bring the system under test (SUT) + dependencies to a desired state

Act / When

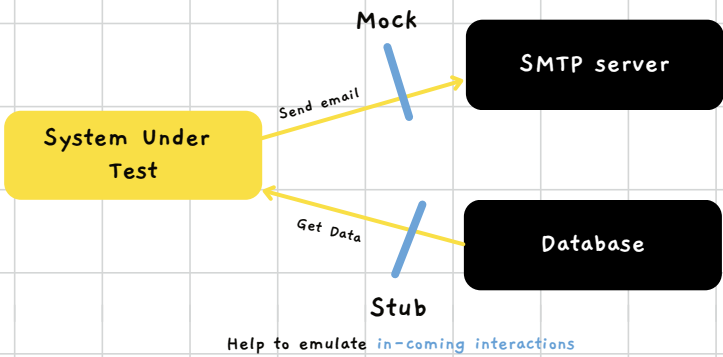
- Invoke the behavior :
- Call method / function on the SUT
- Pass the prepared dependencies

Assert / Then

- Verify the outcome :
- The return value
- The final state of the SUT and its collaborators
- Or the methods the SUT called on collaborators

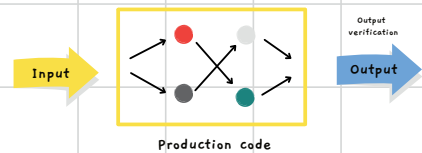
Test doubles

Help to emulate and examine **out-coming interactions**



Output-based

- Feed an input to the system under test (SUT)
- Check the output it produces

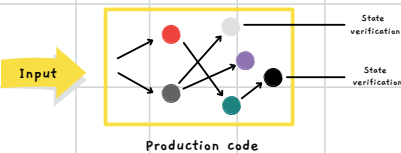


Assumes there are no side effects and the only result of the SUT is the value it returns to the caller -> functional

Both school use it

State-based

Verify the final state of the system after an operation is complete



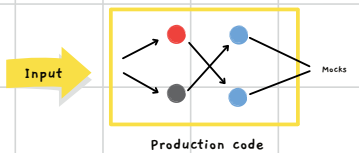
"State" can refer to the state of :

- The SUT itself
- One of its collaborators
- Or an out-of-process dependency (db / fs)

★ Classical preference

Communication-based

Verify that the SUT calls its collaborators correctly



Tests substitute collaborators with mocks

★ London preference

3 Styles of tests

Resistance to refactoring



Maintainability costs

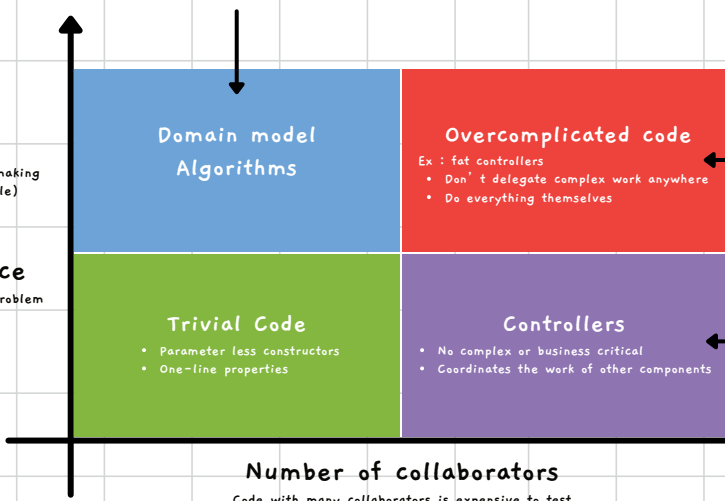


Unit test this gives the best return on investment

4 Types of Code

Complexity
Defined by the number of decision-making (cyclomatic complexity for example)

Domain significance
How significant the code is for the problem domain of your project



Refactor it by splitting into :

- Algorithms
- Controllers

Integration Tests

Number of collaborators
Code with many collaborators is expensive to test

