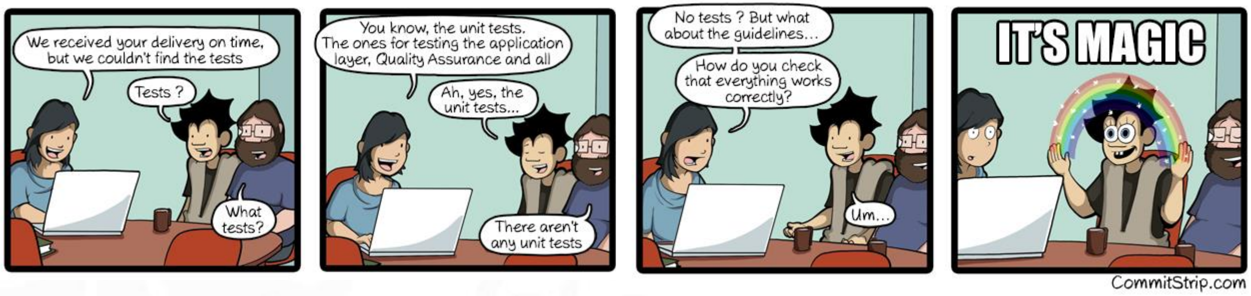
Testing your Code


We received your delivery on time, but we couldn't find the tests
Tests ?
What tests?
You know, the unit tests. The ones for testing the application layer, Quality Assurance and all
Ah, yes, the unit tests...
There aren't any unit tests
No tests ? But what about the guidelines...
How do you check that everything works correctly?
Um...
IT'S MAGIC
CommitStrip.com

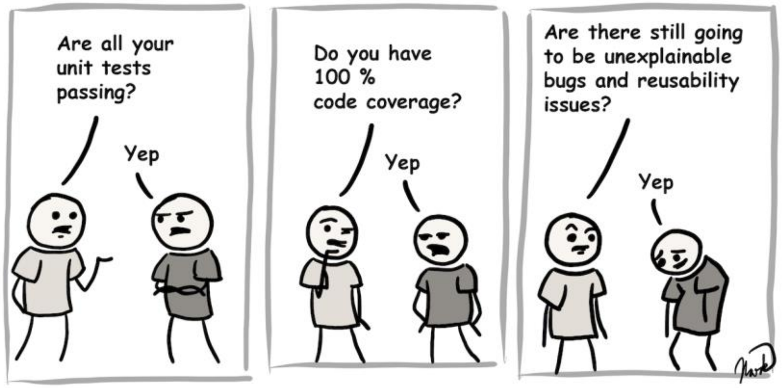
Contents

Introduction

Programming is writing code to solve problems. Good programming is the practice of using a **structured process to solve problems**.

As engineers, we want to have a codebase we can **change, extend, and refactor** as required.

Tests ensure our program works as intended and that changes to the codebase do not break existing functionality.


Are all your unit tests passing?
Yep
Do you have 100 % code coverage?
Yep
Are there still going to be unexplainable bugs and reusability issues?
Yep

Introduction

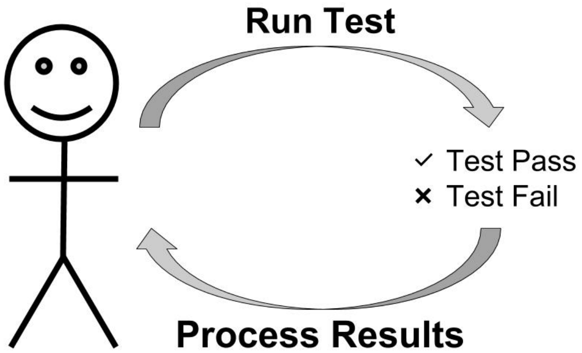
However, not all the code must be tested. It’s all about being pragmatic in **what** you test, **how** you test and **when** you test. You should leverage tools and techniques that allow you to test your code as efficiently as possible.

Testing needs to be **easy and free of barriers**; once testing feels like a chore, programmers won't do it... and this is how software quality slips.

Testing is folklore in the sense that best practices and techniques are passed down from programmer to programmer while working on projects as part of a team. If you are new to the industry and are trying to grok testing, it's hard to figure out **how to get started**. It feels like there is a lot of conflicting advice out there, and that's because there is.

What is testing?

When we write code, we need to run it to ensure that it is doing what we expect it to. **Tests are a contract with our code: given a value, we expect a certain result to be returned**.



While passing tests **cannot prove the absence bugs**, they do inform us that our code is working in the manner defined by the test. In contrast, a failing test indicates that something is not right. We need to understand why our test failed so we **can modify code and/or tests,** as required.

What is testing?

Properties of Tests

1. **Fast**

A slower feedback loop hampers development as it takes us longer to find out if our change was correct. If our workflow is plagued by slow tests, we won't be running them as often.

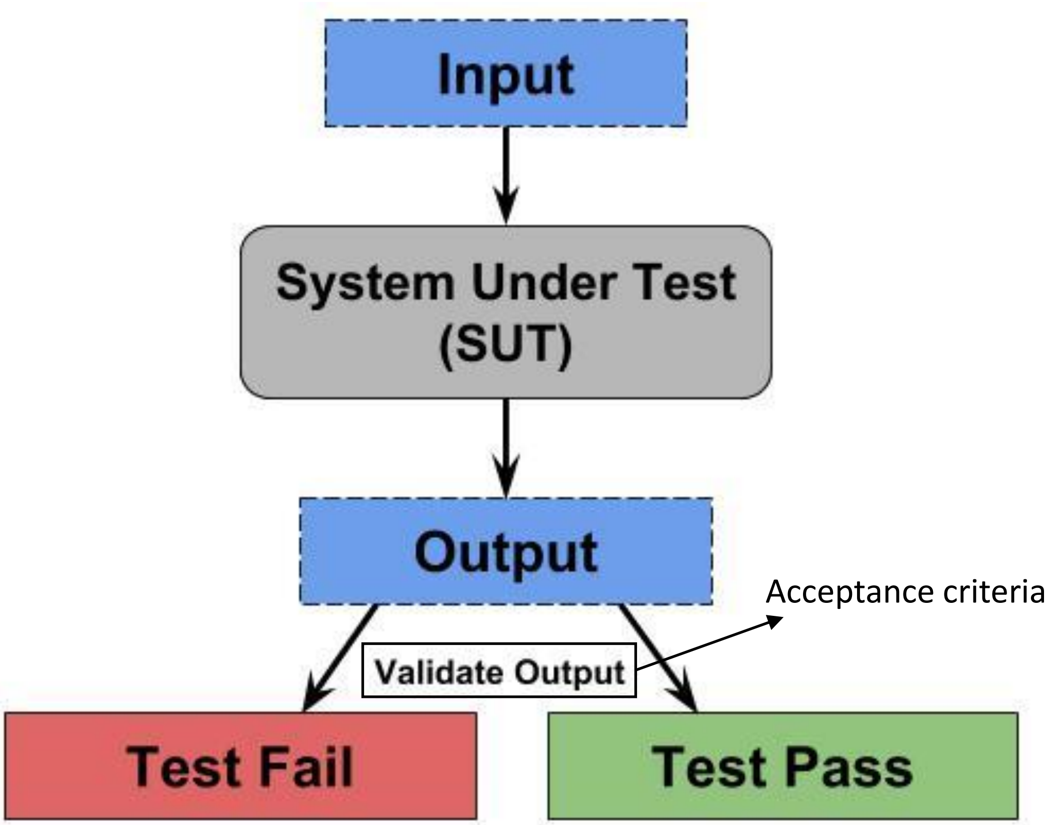
2. **Deterministic**

Tests should be deterministic, i.e. the same input will always result in the same output. If tests are non-deterministic, we have to find a way to account for random behavior inside of our tests.

3. **Automated**

We can confirm our program works by running it. While manual testing is fine for small projects, it becomes unmanageable as our project grows in complexity. By automating our test suite, we can quickly verify our program works on-demand.

What is testing?
Formal Definition



What is testing?
Benefits of Testing

1. Modify Code with Confidence

If a program does anything of interest, it has interactions between functions, classes, and modules. This means a single line change can break our program in unexpected ways. By running our tests after we modify our code, we can confirm our changes did not break existing functionality as defined by our tests

2. Identify Bugs Early

Fixing bugs gets more expensive the further you are in the project. True Cost of a Bug digs into this issue.

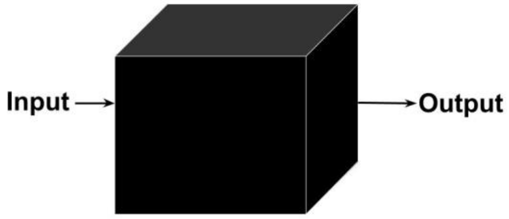
3. Improve System Design

A thorough test suite shows that the developer has actually thought about the problem in some depth. Writing tests forces you to use your **own API** by writing modular code. Complex interdependencies between modules lead to spaghetti code.

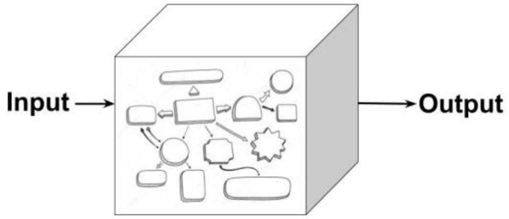
Types of Testing

Black Box vs White Box

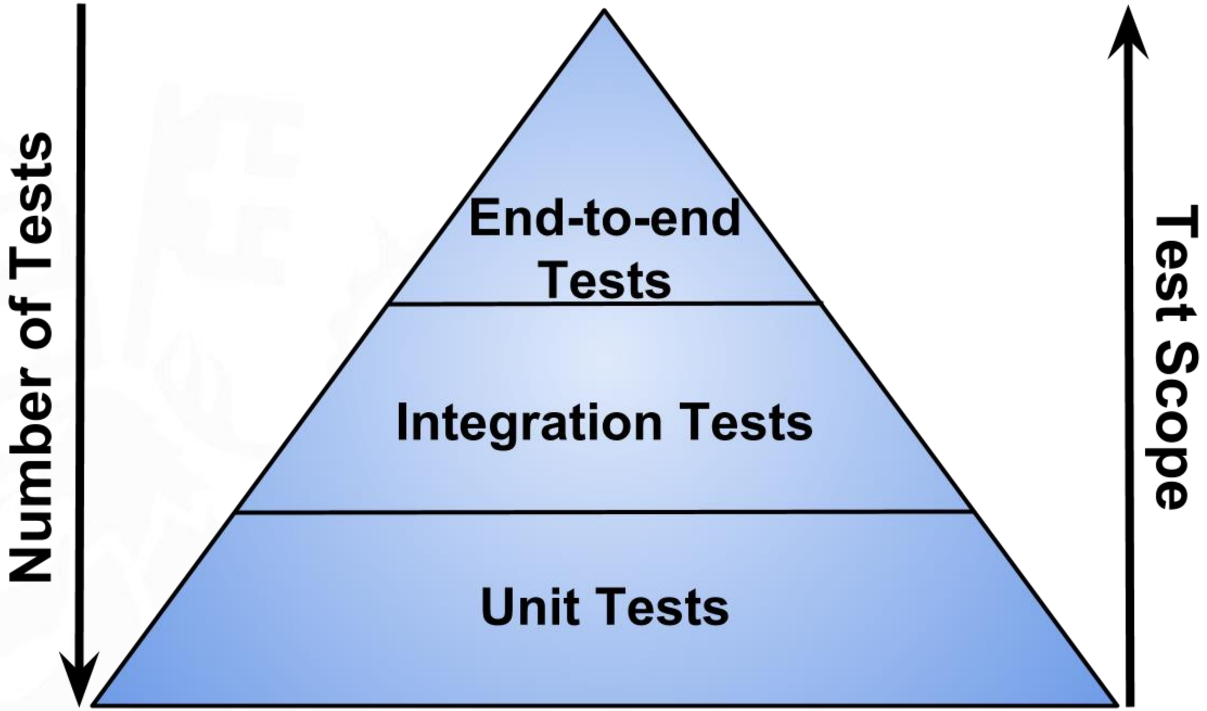
Black Box Testing refers to testing techniques in which the tester cannot see the inner workings of the item being tested.



White Box Testing is the technique in which the tester can see the inner workings of the item being tested.



Types of Testing
Test pyramid



Types of Testing
Unit tests

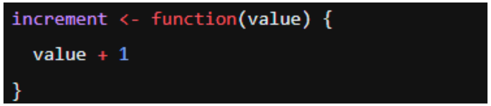
Unit tests are low-level tests that focus on testing a specific part of our system. They are cheap to write and fast to run. Test failures should provide enough contextual information to pinpoint the source of the error.

Unit tests should be **independent and isolated**; interacting with external components increases both the scope of our tests and the time it takes for tests to run.

These tests give us confidence that our program works as expected. Writing new code or modifying existing code might require us to rewrite some of our tests.

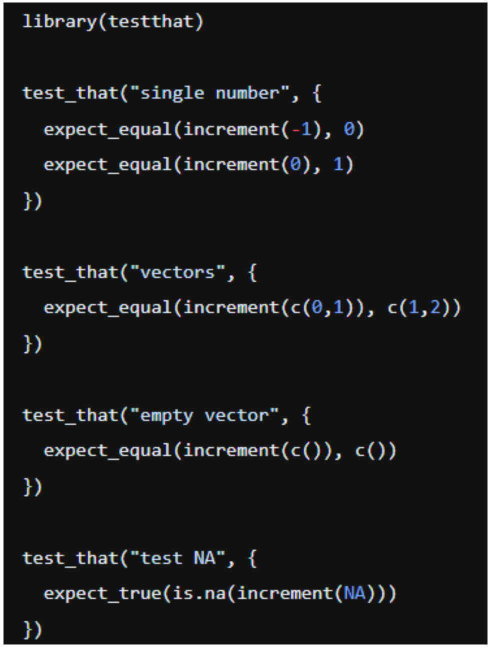
Types of Testing
Unit tests

Supposed we have the following function:



```r
increment <- function(value) {
  value + 1
}
```

This function can be tested using the following code:



```r
library(testthat)

test_that("single number", {
  expect_equal(increment(-1), 0)
  expect_equal(increment(0), 1)
})

test_that("vectors", {
  expect_equal(increment(c(0,1)), c(1,2))
})

test_that("empty vector", {
  expect_equal(increment(c()), c())
})

test_that("test NA", {
  expect_true(is.na(increment(NA)))
})
```

Types of Testing

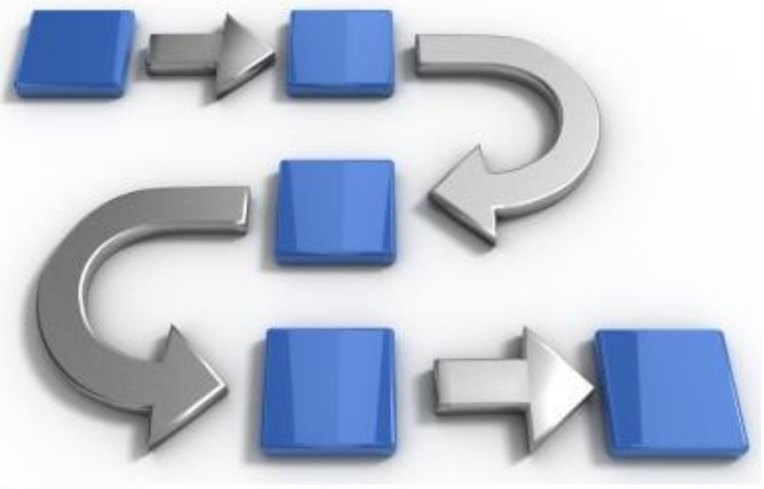
Integration tests

Every complex application has internal and external components working together to do something interesting. In contrast to units tests which focus on individual components, integration tests **combine various parts** of the system and test them together as a group.

By definition, integration tests are larger in scope and take longer to run than unit tests. This means that test failures require some investigation: **we know that one of the components in our test is not working, but the failure's exact location needs to be found**. This is in contrast to unit tests which are smaller in scope and indicate exactly where things have failed.

Types of Testing
End-to-End tests

End-to-end tests check to see if the system meets our defined business requirements. A common test is to trace a path through the system in the same manner a user would experience.



Types of Testing

End-to-End tests

We can conduct end-to-end tests **through our user interface (UI).** This creates a dependency between our UI and our tests, which makes our tests brittle: a change to the front-end requires us to change tests.

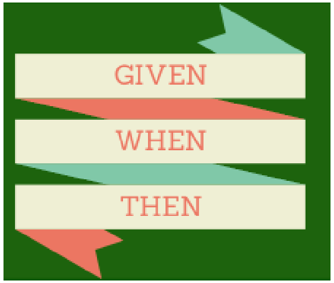
A better solution is to **test the subcutaneous layer**, i.e., the layer just below our user interface. End-to-end tests are considered **black box** as we do not need to know anything about the implementation in order to conduct testing. This also means that **test failures provide no indication of what went wrong**; we would need to use logs to help us trace the error and diagnose system failure.

Structuring Tests

Each test case can be separated into the following phases:

- setting up the **system under test** (SUT) to the environment required by the test case (pre-conditions)
- performing the **action** we want to test on SUT
- verifying if the **expected outcome** occurred (post-conditions)
- tearing down SUT and putting the environment back to the state we found it in

There are two widely used frameworks for structuring tests: Arrange-Act-Assert and Given-When-Then.



Tests Patterns
Arrange-Act-Assert (AAA)

The AAA pattern is abstraction for separating the different part of our tests:

- **Arrange** all necessary pre-conditions
- **Act** on the SUT
- **Assert** that our post-conditions are met

This is the most common test pattern. Example:

```python
def test_find_top_word():
    # Arrange
    words = ["foo", "bar", "bat", "baz", "foo", "baz", "foo"]

    # Act
    result = find_top_word(words)

    # Assert
    assert result[0] == "foo"
    assert result[1] == 3
```

What to test

In order to prove that our program is correct, we have to test it against **every conceivable combination of input values**. This type of exhaustive testing (also called **brute-force test**) is not practical so we need to **employ testing strategies** that allow us to select test cases where errors are most likely to error.

Seasoned developers can balance writing code to solve business problems with writing tests to ensure correctness and prevent **regression**. Finding this balance and knowing what to test can feel more like an art than a science. Fortunately, there are a few rules of thumb we can follow to make sure our testing is thorough.

What to test
Basics

- **Functional requirements**

Make sure that all relevant requirements have been implemented. There is no point building something if it doesn't meet the criteria you set forth.

- **Basis Path**

We have to test each statement at least once. If the statement has a conditional, we have to test all branches of the conditional.

- **Equivalence Partitioning**

Two test cases that result in the same output are said to be equivalent. We only require one of the test cases.

- **Boundary Analysis**

Not only tests all branches of the conditional, but also the boundary condition.

What to test
Basics

- **Classes of Bad Data**

This refers to: too little data (or no data), too much data, invalid data, wrong size of data, uninitialized data.

- **Data Flow Testing**

Focuses on tracing the control flow of the program with a focus on exploring the sequence of events related to the status of data objects. For example, we get an error if we try to access a variable that has been deleted.

- **Error Guessing**

Past experience provides insights into parts of our code base that can lead to errors. Keeping a record of previous errors can prevent committing them again.

Test results

Code Coverage

Code coverage is a white-box testing technique performed to verify the extent to which the code has been executed. Code coverage tools use static instrumentation in which statements monitoring code execution are inserted at critical junctures in the code.

Code coverage is primarily performed at the unit testing level. Unit tests are created by developers, thus giving them the best advantage from which to decide what tests to include in unit testing. At this point, code coverage answers several questions such as:

- Are there enough tests in the unit test suite?
- Do more tests need to be added?

Test results

Code coverage

There are a few levels of code coverage:

1. **Function Coverage** – The functions in the source code that are called and executed at least once.
2. **Statement Coverage** – The number of statements that have been successfully validated in the source code.
3. **Path Coverage** – The flows containing a sequence of controls and conditions that have worked well at least once.
4. **Branch or Decision Coverage** – The decision control structures (loops, for example) that have executed fine.
5. **Condition Coverage** – The Boolean expressions that are validated and that executes both TRUE and FALSE as per the test runs.
6. **Finite State Machine Coverage** - This works based on the frequency of visits from static states and other transactions.

Test results

Test coverage

Test coverage is a black-box testing technique that monitors the number of tests that have been executed. **Test cases** are written to ensure maximum coverage of requirements outlined contained in:

- **FRS** (Functional Requirements Specification)
- **SRS** (Software Requirements Specification)
- **URS** (User Requirement Specification)

The test coverage report provides information about parts of the software where test coverage is being implemented.

This metric validates if a product can be released to the public, even if it is not explicitly calculated.

Test results

Test coverage

There are a few levels of test coverage:

- **Features Coverage**: Test cases are developed to implement maximum coverage of product features.
- **Risk Coverage**: Every product requirement document mentions the risks associated with the project and how to mitigate them. They are addressed in this stage of test coverage.
- **Requirements Coverage**: Tests are defined to provide maximum coverage of the product requirements mentioned in the requirement documents.

Tests Patterns

When to test

While there is a lot of interesting discussion about when to write tests, it takes away from the point of testing. It doesn't matter when you write tests, it just matters that you write tests.

If you are interested in exploring this topic, check out:

- RubyOnRails creator David Heinemeier Hansson (DHH) criticizing TDD at RailsConf 2014
- Is TDD Dead? discussion with DHH, Martin Fowler, and Kent Beck
- TDD: A Academic Survey by Ted M. Young

Bibliography

- Sivji, A. (2018). *Testing 101: Introduction to Testing*. URL: https://alysivji.github.io/testing-101-introduction-to-testing.html
- Meszaros, G. *Four-Phase Test*. URL: http://xunitpatterns.com/Four%20Phase%20Test.html
- Fowler, M. (2013). *GivenWhenThen*. URL: https://martinfowler.com/bliki/GivenWhenThen.html
- Cooke, J. (2017). *Arrange Act Assert pattern for Python developers*. URL: https://jamescooke.info/arrange-act-assert-pattern-for-python-developers.html
- Knight, A. (2018). *Behavior-Driven Python at PyCon 2018*. URL: https://www.youtube.com/watch?v=EtIAbfCrsFI
- De Caro, J. (2018). *A Beginner's Guide to Testing: Error Handling Edge Cases*. URL: https://www.freecodecamp.org/news/a-beginners-guide-to-testing-implement-these-quick-checks-to-test-your-code-d50027ad5eed
- Kanat-Alexander, M. (2013). *The Philosophy of Testing*. URL: https://www.codesimplicity.com/post/the-philosophy-of-testing/

Bibliography

- Okken, B. *Test & Code in Python: Developing Software with Automated Tests*. Podcast. URL: https://testandcode.com/

Alberto Aguilera 23, E-28015 Madrid - Tel: +34 91 542 2800 - http://www.iit.comillas.edu