# What Is Unit Test?

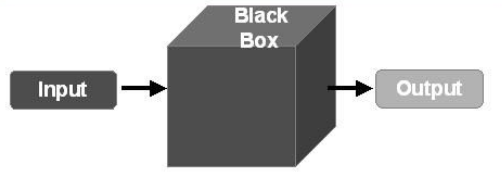
Unit testing is a set of activities that **take the smallest piece of testable software** in the application (unit of work), isolate it from the remainder of the code, **and determine whether it behaves exactly as you expect.**

A unit of work is a task that is not directly dependent on the completion of any other task. It is a single method of the source code.

# Types of Unit Tests

## Black Box Unit Test

A black box test (also known as "functional test") is one in which you **feed it inputs and verify the** outputs without being able to inspect the internal workings.



In details, a black box doesn't usually have information regarding:

* How the box handles errors
* Whether your inputs are executing all code pathways
* How to modify your inputs so that all code pathways are executed
* Dependencies on other resources

**Pros**:

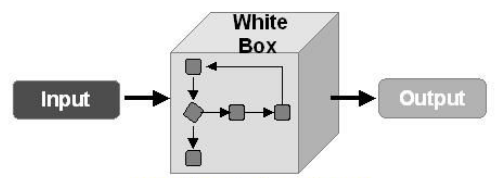
* Simple because of focusing on verifying that good inputs result in good outputs.

**Cons**:

* Limits your ability to thoroughly test the code, primarily because you don't know if you're testing all the code pathways.  Typically, a black box test

## White Box Unit Test

A white box test **provides the information necessary to test all the possible pathways**. This includes not only correct inputs, but incorrect inputs, so that error handlers can be verified as well.



**Pros**:

* You know how the box handles errors.
* You can usually write tests that verify all code pathways.
* The unit test, being more complete, is a kind of documentation guideline that the implementer can use when actually writing the code in the box.
* Resource dependencies are known.
* Internal workings can be inspected.

**Cons**:

* Take more time to write unit test code than black box testing.

# Basic Concepts

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Test | Same as **test case**. |
| Assertions | Statements that **check whether a condition is true**. |
| Test suite | Contains one or **many tests**. |
| Test fixture | When multiple tests in a test suite need to **share common objects and subroutines**, you can put them into a test fixture class. |
| Test program | Contain one or **many test suites**. |
| Test harness | A set of software and test data configured to test a program unit by running it under varying conditions and monitor its behavior and outputs. It has two main parts: the test execution engine and the test script repository. |
| SUT | *Software Under Test*.  This refers to **production code** that we’re writing unit test for. |
| DOC | *Dependent-On Component*.  This refers to a **dependency used by the SUT**. This might include system libraries, their-party libraries, databases, or external APIs. To effectively test a unit, you often use test doubles to simulate these dependencies in order to verify its behavior without relying on the actual external components. |
| Test doubles | This refers to simulated object that mimics the behavior of real object in controlled ways. There are various sub-types of test doubles:   * **Dummy object**: This refers to objects which are passed as parameters to method calls of the SUT. E.g. *nullptr*, *"ignored string"*, *new Object()* … * **Test stub**: Replace a real component on which the SUT depends so that the test has a control point for the indirect inputs of the SUT. * **Test spy**: Like a test stub, a test spy may need to provide values to the SUT in response to method calls. The test spy, however, also captures the indirect outputs of the SUT as it is exercised and saves them for later verification by the test * **Mock object**: This includes the functionality of a test stub in that it must return values to the SUT if it hasn’t already failed the tests but the emphasis is on the verification of the indirect outputs (verify correctness agains expectations). * **Fake object**: Implements the same functionality as the real DOC but in a much simpler way (is not used to verify indirect input and output). E.g. in-memory databas emulator. |

# Philosophy of Unit Test

## General Rules

What make a good unit test?

* Tests should be independentandrepeatable. It’s a pain to debug a test that succeeds or fails as a result of other tests. We must **isolate test by running each of them on a different object**.
* Tests should be well-organized and reflect the structure of the tested code. We must **group related tests into test suites** that can share data and subroutines.
* Tests should be portable and reusable. Your code can work on different OSs, with different compilers, with or without exceptions, so tests **should work with a variety of configurations**.
* When tests fail, they should provide as much information about the problem as possible. So **testing shouldn’t stop at the first test failure, but should continue with the next test**. Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.
* Tests should be fast so that we can reuse shared resources across tests and pay for the set-up/tear-down only once, without making tests depend on each other.
* Writing tests should be quick and easy. The amount of effort it takes to write or modify tests should not exceed the effort it takes to implement the corresponding functionality.
* Production code should be designed and implemented in a way that makes unit test easy. This can be achieved by applying design-for-testability patterns (e.g., [coding to interface – not concrete](#_Separation_of_Interface), etc.).
* Test logic should be out of production code
* A good ***unit test framework*** should help us do all of above points. Also, it provides:
  + Base classes for creating test case, test suite.
  + Support for creating mock objects.
  + Various assertion statements.
  + An easy way for executing unit test as well as outputting the test result.

## Test First or Last?

Even if we design for testability, the likelihood that we can write the tests easily and naturally without modifying the production code is low. When tests are written first and we write only enough code to make the tests pass, the production code tends to be more minimalist.

🡺 Should be Test First (but in reality, this is very hard to achieve)

## Outside-In or Inside-Out?

With Outside-In, development starts at the outside using Test Doubles in place of the DOCs and proceeds inward as requirements for each DOC are identified. With Inside-Out, development starts with the innermost components and proceeds toward the user interface, building on the previously constructed components.

|  |  |
| --- | --- |
| Outside-In | Inside-Out |
|  |  |

🡺 Should be Outside-In

# Design-for-Testability Patterns

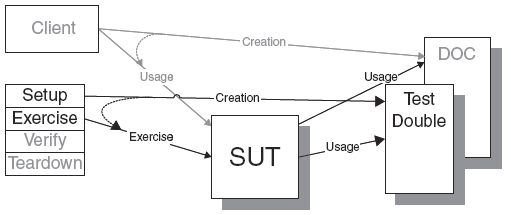
## Dependency Injection

*How do we design the SUT so that we can replace its dependencies at runtime?*

* ***Separation of interface from implementation***

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. For example, a class that depends on a database – to test it, we often writes code that interacts with the database. This is a mistake, because a unit test should never go outside of its own class boundary.

As a result, we should abstract an interface around the database connection, and then implements that interface with our own test double object. Then we can set any dummy test data without having to depend on the database.



## Dependency Lookup

*How do we design the SUT so that we can replace its dependencies at runtime?*

* ***The SUT asks another object to return the DOC’s object before using it****.*

To do that:

* Implementing either a singleton, a registry, or some kind of thread-specific storage.
* An interface that fully encapsulates which implementation we are using.
* A built-in substitution mechanism for replacing the returned object with a test double.
* Access via well-known global name.
* Factory objects.

|  |  |
| --- | --- |
|  |  |

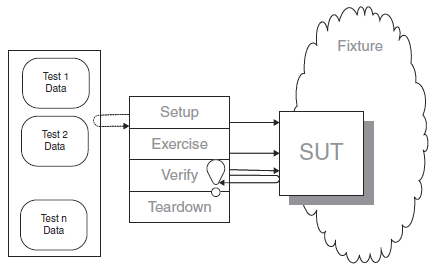
## Humble Object

## Test Hook

## Test Data

*How do we prepare automated tests for our software? And how do we reduce test code duplication?*

* ***We store all the information needed for each test in a data file and write an interpreter that reads the file and executes the tests.***



# Code Coverage

## What Is Code Coverage?

Code coverage is a measure used in software testing. It **describes the degree to which the source code of a program has been tested**. It is expressed as *percentage*.

It is a form of testing that looks at the code directly (white box testing).

There are a number of different ways of measuring code coverage:

* **Statement/Line Coverage**: Has each line of the source code been executed and tested?
* **Function Coverage**: Has each function been executed and tested?
* **Condition Coverage**: Has each evaluation point (such as a true/false decision) been executed and tested?
* **Path Coverage**: Has every possible route through a given part of the code been executed and tested?
* **Entry/Exit Coverage**: Has every possible call and return of the function been executed and tested?

Notice that Path Coverage and Entry/Exit Coverage are like subsets of Condition Coverage. So, we can consider them as Condition Coverage for short.