Unit Testing

Unit testing is a fundamental tool in every developer's toolbox. Unit tests not only help us test our code, they encourage good design practices, reduce the chances of bugs reaching production, and can even serve as examples or documentation on how code functions. Properly written unit tests can also improve developer efficiency.

Unit testing also is one of the most commonly misunderstood forms of testing. Unit testing refers to a very specific type of testing; a unit test should be:

- **Provably reliable** - should be 100% reliable so failures indicate a bug in the code
- **Fast** - should run in milliseconds, a whole unit testing suite shouldn't take longer than a couple seconds
- **Isolated** - removing all external dependencies ensures reliability and speed

Why Unit Testing

It is no secret that writing unit tests is hard, and even harder to write well. Writing unit tests also increases the development time for every feature. So why should we write them?

Unit tests

- reduce costs by catching bugs earlier and preventing regressions
- increase developer confidence in changes
- speed up the developer inner loop
- act as documentation as code

For more details, see all the detailed descriptions of the points above.

Unit Testing Design Blocks

Unit testing is the lowest level of testing and as such generally has few components and dependencies.

The **system under test** (abbreviated SUT) is the "unit" we are testing. Generally these are methods or functions, but depending on the language these could be different. In general, you want the unit to be as small as possible though.

Most languages also have a wide suite of **unit testing frameworks** and test runners. These test frameworks have a wide range of functionality, but the base functionality should be a way to organize your tests and run them quickly.

Finally, there is your **unit test code**; unit test code is generally short and simple, preferring repetition to adding layers and complexity to the code.

Applying the Unit Testing

Getting started with writing a unit test is much easier than some other test types since it should require next to no setup and is just code. Each test framework is different in how you organize and write your tests, but the general techniques and best practices of writing a unit test are universal.

Techniques

These are some commonly used techniques that will help when authoring unit tests. For some examples, see the pages on using abstraction and dependency injection to author a unit test, or how to do test-driven development.

Note that some of these techniques are more specific to strongly typed, object-oriented languages. Functional languages and scripting languages have similar techniques that may look different, but these terms are commonly used in all unit testing examples.

Abstraction

Abstraction is when we take an exact implementation detail, and we generalize it into a concept instead. This technique can be used in creating testable design and is used often especially in object-oriented languages. For unit tests, abstraction is commonly used to break a hard dependency and replace it with an abstraction. That abstraction then allows for greater flexibility in the code and allows for the a mock or simulator to be used in its place.

Limitation

One of the side effects of abstracting dependencies is that you may have an abstraction that has no test coverage. This is case where unit testing is not well-suited, you can not expect to unit test everything, things like dependencies will always be an uncovered case. This is why even if you have a robust unit testing suite, integration or functional testing should still be used - without that, a change in the way the dependency functions would never be caught.

When building wrappers around third-party dependencies, it is best to keep the implementations with as little logic as possible, using a very simple facade that calls the dependency.

An example of using abstraction can be found here.

Dependency Injection

Dependency injection is a technique which allows us to extract dependencies from our code. In a normal use-case of a dependant class, the dependency is constructed and used within the system under test. This creates a hard dependency between the two classes, which can make it particularly hard to test in isolation. Dependencies could be things like classes wrapping a REST API, or even something as simple as file access. By injecting the dependencies into our system rather than constructing them, we have "inverted control" of the dependency. You may see "Inversion of Control" and "Dependency Injection" used as separate terms, but it is very hard to have one and not the other, with some arguing that Dependency Injection is a more specific way of saying inversion of control. In certain languages such as C#, not using dependency injection can lead to code that is not unit testable since there is no way to inject mocked objects. Keeping testability in mind from the beginning and evaluating using dependency injection can save you from a time-intensive refactor later.

One of the downsides of dependency injection is that it can easily go overboard. While there are no longer hard dependencies, there is still coupling between the interfaces, and passing around every interface implementation into every class presents just as many downsides as not using Dependency Injection. Being intentional with what dependencies get injected to what classes, is key to developing a maintainable system.

Many languages include special Dependency Injection frameworks that take care of the boilerplate code and construction of the objects. Examples of this are Spring in Java or built into ASP.NET Core

An example of using dependency injection can be found here.

Test-Driven Development Test code → System code

Test-Driven Development (TDD) is less a technique in how your code is designed, but a technique for writing your code that will lead you to a testable design from the start. The basic premise of test-driven development is that you write your test code first and then write the system under test to match the test you just wrote. This way all the test design is done up front and by the time you finish writing your system code, you are already at 100% test pass rate and test coverage. It also guarantees testable design is built into the system since the test was written first!

For more information on TDD and an example, see the page on Test-Driven Development

Best Practices

Arrange/Act/Assert

One common form of organizing your unit test code is called Arrange/Act/Assert. This divides up your unit test into 3 different discrete sections:

1. Arrange - Set up all the variables, mocks, interfaces, and state you will need to run the test
2. Act - Run the system under test, passing in any of the above objects that were created
3. Assert - Check that with the given state that the system acted appropriately.

Using this pattern to write tests makes them very readable and also familiar to future developers who would need to read your unit tests.

EXAMPLE

Let's assume we have a class `MyObject` with a method `TrySomething` that interacts with an array of strings, but if the array has no elements, it will return false. We want to write a test that checks the case where array has no elements:

```
[Fact]
public void TrySomething_NoElements_ReturnsFalse()
{
    // Arrange
    var elements = Array.Empty<string>();
    var myObject = new MyObject();

    // Act
    var myReturn = myObject.TrySomething(elements);

    // Assert
    Assert.False(myReturn);
}
```

Keep Tests Small and Test Only One Thing

Unit tests should be short and test only one thing. This makes it easy to diagnose when there was a failure without needing something like which line number the test failed at. When using Arrange/Act/Assert, think of it like testing just one thing in the "Act" phase.

There is some disagreement on whether testing one thing means "assert one thing" or "test one state, with multiple asserts if needed". Both have their advantages and disadvantages, but as with most technical disagreements there is no "right" answer. Consistency when writing your tests one way or the other is more important!

Using a Standard Naming Convention for All Unit Tests

Without having a set standard convention for unit test names, unit test names end up being either not descriptive enough, or duplicated across multiple different test classes. Establishing a standard is not only important for keeping your code consistent, but a good standard also improves the readability and debug-ability of a test. In this article, the convention used for all unit tests has been `UnitName_StateUnderTest_ExpectedResult`, but there are lots of other possible conventions as well, the important thing is to be consistent and descriptive. Having descriptive names such as the one above makes it trivial to find the test when there is a failure, and also already explains what the expectation of the test was and what state caused it to fail. This can be especially helpful when looking at failures in a CI/CD system where all you know is the name of the test that failed - instead now you know the name of the test and exactly why it failed (especially coupled with a test framework that logs helpful output on failures).

Things to Avoid

Some common pitfalls when writing a unit test that are important to avoid:

- Sleeps - A sleep can be an indicator that perhaps something is making a request to a dependency that it should not be. In general, if your code is flaky without the sleep, consider why it is failing and if you can remove the flakiness by introducing a more reliable way to communicate potential state changes. Adding sleeps to your unit tests also breaks one of our original tenets of unit testing: tests should be fast, as in order of milliseconds. If tests are taking on the order of seconds, they become more cumbersome to run.
- Reading from disk - It can be really tempting to the expected value of a function return in a file and read that file to compare the results. This creates a dependency with the system drive, and it breaks our tenet of keeping our unit tests isolated and 100% reliable. Any outside dependency such as file system access could potentially cause intermittent failures. Additionally, this could be a sign that perhaps the test or unit under test is too complex and should be simplified.
- Calling third-party APIs - When you do not control a third-party library that you are calling into, it's impossible to know for sure what that is doing, and it is best to abstract it out. Otherwise, you may be making REST calls or other potential areas of failure without directly writing the code for it. This is also generally a sign that the design of the system is not entirely testable. It is best to wrap third party API calls in interfaces or other structures so that they do not get invoked in unit tests. For more information see the page on mocking.

Unit Testing Frameworks and Tools

Test Frameworks

Unit test frameworks are constantly changing. For a full list of every unit testing framework see the page on Wikipedia. Frameworks have many features and should be picked based on which feature-set fits best for the particular project.

Mock Frameworks

Many projects start with both a unit test framework, and also add a mock framework. While mocking frameworks have their uses and sometimes can be a requirement, it should not be something that is added without considering the broader implications and risks associated with heavy usage of mocks.

To see if mocking is right for your project, or if a mock-free approach is more appropriate, see the page on mocking.

Tools

These tools allow for constant running of your unit tests with in-line code coverage, making the dev inner loop extremely fast and allows for easy TDD:

- Visual Studio Live Unit Testing
- Wallaby.js
- Infinitest for Java
- PyCrunch for Python

Things to Consider

Transferring Responsibility to Integration Tests

In some situations it is worth considering to include the integration tests in the inner development loop to provide a sufficient code coverage to ensure the system is working properly. The prerequisite for this approach to be successful is to have integration tests being able to execute at a speed comparable to that of unit tests both locally and in a CI environment. Modern application frameworks like .NET or Spring Boot combined with the right mocking or stubbing approach for external dependencies offer excellent capabilities to enable such scenarios for testing.

Usually, integration tests only prove that independently developed modules connect together as designed. The test coverage of integration tests can be extended to verify the correct behavior of

the system as well. The responsibility of providing a sufficient branch and line code coverage can be transferred from unit tests to integration tests. Instead of several unit tests needed to test a specific case of functionality of the system, one integration scenario is created that covers the entire flow. For example in case of an API, the received HTTP responses and their content are verified for each request in test. This covers both the integration between components of the API and the correctness of its business logic.

With this approach efficient integration tests can be treated as an extension of unit testing, taking over the responsibility of validating happy/failure path scenarios. It has the advantage of testing the system as a black box without any knowledge of its internals. Code refactoring has no impact on tests. Common testing techniques as TDD can be applied at a higher level which results in a development process that is driven by acceptance tests. Depending on the project specifics unit tests still play an important role. They can be used to help dictate a testable design at a lower level or to test complex business logic and corner cases if necessary.

Conclusion

Unit testing is extremely important, but it is also not the silver bullet; having proper unit tests is just a part of a well-tested system. However, writing proper unit tests will help with the design of your system as well as help catch regressions, bugs, and increase developer velocity.

Resources

- Unit Testing Best Practices

Last update: August 26, 2024