Unit Testing

Contents

Overview

Below we'll be covering the basics of unit testing and explore how and why we use this testing approach.

Unit Testing

What is Unit Testing

Unit testing is a testing approach that targets the very fundamental building blocks of an application, the idea is to prove that each 'unit' of the application is functioning as expected.

Unit testing is:

- An `automated` process; it is designed to be reusable and provide reliable results.
- A `white-box` testing approach, meaning that the code is known to the tester.

In order to know what a unit might be in an application, we need to determine precisely what it would consist of. Using an object-oriented language such as `Java`, we know that the application is constructed in parts by using `class` objects to help define grouped functionality.

Below we have a simple class. When the single method within the `TargetClass` is called, it would only return "Hello World".

Example:

```
class TargetClass{

    public int id = 0;

    public TargetClass(){}

    protected String rtnMessage(){
        return "Hello World";
     }
}
```

The class itself is a unit of the application and can be "applied in conjunction"/"applied with" other `class` objects. When a unit communicates to another, the behaviour becomes dependent which makes it no longer within the scope of a unit test.

The idea of unit testing is to ensure that our `unit` functions in isolation, this means we want to check that the performed actions of that `unit`/`class` meet our expectations.

Whilst this process can appear redundant because a method should simply run and behave perfectly as defined, there are reasons why we might still perform this form of testing.

Why do we perform Unit Testing

The code we write may not always be code that we have designed. If you have worked with programming languages, you will find that you can import functionality designed by others to help facilitate your program.

In these cases, when we use a dependency within our code we are also introducing a risk.

If we were to update the dependencies, even the most basic elements of our program could suffer from unforeseen changes that can impact our release process. Using unit testing, we can isolate issues as low-level as a class-specific method by using supporting technologies such as `Junit` to help us write tests, which in turn will help automate and identify possible future errors.

Tutorial

How to perform Unit Testing

Automated testing is generally supported with technologies such as `Junit`. This supporting technology allows users to write code that sets an environment which tracks the code we run and whether or not we have effectively covered it from potential failures.

Using the example before, we can break down our class into `methods` and `attributes` and run these with a certain expectation.

In `Junit` we use assertions to prove that the code we run has behaved correctly, running through the example below we can begin to break down what we need to test to complete a `unit` test.

Example:

```
class TargetClass{

    public int id = 0;

    public TargetClass(){}

    protected String rtnMessage(){
        return "Hello World";
     }
}
```

The example above is a `unit` of our application, but what do we need to do to test it?

Ideally, we'd set up a test suite or application that calls this object and performs either the methods or returns the attributes to prove that the `unit` behaves in the way we expect.

The `unit` has the following methods and attributes:

- `int id` (attribute, line 3)
- `TargetClass` (Constructor, line 5)
- `String rtnMessage()` (method, line 7)

To perform a `unit` test on `TargetClass`, we need to create an instance of the object and then create a test for each method and attribute uniquely.

for example:

```
//Junit example
class TargetClass_TEST {
    @Test
    public void constructor_TEST(){
        TargetClass object = new TargetClass();

        // expectation, result
        AssertTrue(object instanceof TargetClass.class);
    }

    @Test
    public void getID_TEST(){
        TargetClass object = new TargetClass();

        // expectation, result
        AssertEquals(0, object.id);
    }

    @Test
    public void rtnMessage_TEST(){
        TargetClass object = new TargetClass();
        String result = object.rtnMessage();

        // expectation, result
        AssertEquals("Hello World", result);
    }
}
```

With this test, we have ensured that `TargetClass` has been checked against our expectations, and running it with a tool such as `Junit` has provided coverage to help reinforce our code maintainability.

Branching

A concept that often creeps up as part of testing, especially `unit` testing is the idea of branching code.

Whilst `unit` tests might cover one scenario, sometimes a method's behaviour may differ based on the data that has been provided to it. We might use an `if` or `switch` statement to make the behaviour dynamic.

Unfortunately just testing the 'successful'/'desired' outcome is not enough in these cases, `unit` testing is an approach that completely covers a method and as such every line of that class needs to have been executed and checked to ensure that bugs do not "slip through the net".

Example:

```
class TargetClass{

    public int id = 0;

    public TargetClass(){}

    protected String rtnMessage(boolean flag){
        if(flag){
            return "Hello World";
        }else{
            return "GoodBye!";
        }
     }
}
```

The above example shows that if we simply test for `rtnMessage(true)`, we will only cover lines 8 and 9. We would also be required to complete a test that also calls `rtnMessage(false)`, that way we have ensured testing will cover all of the `unit`/`class` in question.

Sociable or Solitary(Isolated)

In certain cases you might be testing a `class` that calls another `class` method as part of its functionality, this is part of an open-ended case of unit testing.

Sociable:

`Sociable` tests are unit tests that allow dependent code to execute for whatever reason is required. This is where code from another class **is allowed to execute** in favour of facilitating the unit test. This code may be from a dependency or driver that you did not write yourself.

> This inclusive nature of unit testing is often an accident because such execution of code is better regarded as `integration` or `system`.

You might use `sociable` unit tests. This is when you use `drivers` or other external technologies.

- Drivers

  External applications or dependencies that help facilitate testing code.

Solitary:

Solitary unit tests are tests that run without including dependent code, in this case, we may utilise `stubs` or `mocks`. These will allow you to test the `unit` in isolation, removing the potential to accidentally write `integration` or `system` tests.

- Stub/Mocks

  Stubs or Mocks are practices of replacing the expected data from a dependency with a fixed value, rather than allowing the application to execute the dependency.

However, you should focus on keeping tests isolated in order to only target the code of the intended `unit`. This is what is meant by `solitary` or `isolated unit` testing.

Exercises

If you have not already covered `Junit`, then please read the `Java Beginner/Junit` module on QA Community. The topic covers the technology and will help get you started on practising unit testing.