Testing Private

A friend and former colleague of mine, Chris 'Frankie' Salt, recently popped up on Facebook messenger and asked me a question:

“I wonder if you'd mind answering a Java question for me? It's more of a best practices thing. So, encapsulation vs availability of methods for testing. Splitting your code into functions makes it a lot more readable and it makes sense to make these private as they will only ever be used once. However unit testing demands access to these private methods, I know there are ways around this but I was interested in your opinion.”

Me? Have an opinion about unit testing? Many stranger things have happened!

Encapsulation is all about hiding code away so that you can change it with minimal or no impact on other parts of the code base which use it indirectly. You shouldn't (ever) compromise encapsulation for the sake of testing. Every private method you write must be callable from at least one public method or via a chain of other private methods which is ultimately called from a public method. Otherwise, reflection shenanigans aside, it would never get called at all.

Does that mean reflection is the solution to testing private methods? No. It's a great tool for poking values into objects which are initialised using reflection at runtime, so that you don't have to add a public non-default constructor or public setters, but beyond that it should be avoided for testing.

Two solutions sprang to mind, *triangulation* and *sprout classes*.

Triangulation

I first encountered the term triangulation in Kent Beck's TDD [TDD] book, although it was a practice I had been using for some time. Triangulation is the practice of passing different values into the public methods of a class in order to test the non-public parts of the class. This is probably best demonstrated with a (contrived) example. Take a look at the `Invoice` class below:

```
public class Invoice
{
	private final List<LineItem> lineItems = new ArrayList<LineItem>();
	private final double vatPc;

	public Invoice(double vatPc)
	{
		this.vatPc = vatPc;
	}

	public void add(LineItem lineItem)
	{
		lineItems.add(lineItem);
	}

	public double grossTotal()
	{
		double total = 0;
		for(LineItem lineItem : lineItems)
			total += lineItem.getValue();
		return total;
	}

	public double netTotal()
	{
		return addVat( grossTotal() );
	}

	private double addVat(double value)
	{
		return vatMultiplier() * value;
```

```
        }

        private double vatMultiplier()
        {
                return (100 + vatPc) / 100;
        }
}
```

It has two private methods, `addVat` and `vatMultiplier`. `addVat` is used to add VAT to the total value of all of the line items. It uses `vatMulitplyer` to convert the VAT rate, which is passed into the invoice as a percentage, into value which can be multiplied by the gross total to get the net total.

As a conscientious developer you would want to write tests which test how `addVat` and `vatMultiplier` work in general cases, such as 20% and corner cases such as 0%. This can be achieved using triangulation and passing different VAT percentages to an invoice, adding some line items and asserting the result of calling `netTotal`, for example:

```
@Test
public void totalNetWithLineItems()
{
        Invoice invoice = new Invoice(20);
        invoice.add(new LineItem(50));
        invoice.add(new LineItem(20));
        invoice.add(new LineItem(20));
        invoice.add(new LineItem(10));
        assertEquals(120, invoice.netTotal(), 0.01);
}

@Test
public void totalNetWithLineItemsAndZeroVat()
{
        Invoice invoice = new Invoice(0);
        invoice.add(new LineItem(50));
        invoice.add(new LineItem(20));
        invoice.add(new LineItem(20));
        invoice.add(new LineItem(10));
        assertEquals(100, invoice.netTotal(), 0.01);
}
```

In a simple example like the `Invoice` class, testing using triangulation is probably sufficient. However, it leaves me with an uncomfortable feeling that the calculation of VAT probably isn't really the responsibility of the `Invoice` class and any more than two test methods for VAT in the `Invoice` test class feels wrong. Of course you can write another test class which just tests the VAT parts of the `Invoice` class, but that doesn't feel right either.

Sprout Classes

I first encountered sprout classes in Michael Feathers' Working Effectively with Legacy Code [Legacy] book, although it was a practice I had been using for some time. A similar technique, called *Extract Class*, is described by Martin Fowler in his Refactoring [Refactoring] book. The basic idea is that you extract parts of one class and put them into a new class which is instantiated and used from the first class. Let's take a look at this in the context of the `Invoice` class:

```
public class Invoice
{
        private final List<LineItem> lineItems = new ArrayList<LineItem>();
        private final double vatPc;

        public Invoice(double vatPc)
        {
                this.vatPc = vatPc;
        }

        public void add(LineItem lineItem)
        {
                lineItems.add(lineItem);
        }
```

```
        public double grossTotal()
        {
                double total = 0;
                for(LineItem lineItem : lineItems)
                        total += lineItem.getValue();
                return total;
        }

        public double netTotal()
        {
                return new Vat(vatPc).add( grossTotal() );
        }
}
```

`addVat` and `vatMultiplier` have been removed and replaced with the instantiation of the `Vat` class and a call to it's `add` method. The `Vat` class looks like this:

```
public class Vat
{
        private final double rate;

        public Vat(double rate)
        {
                this.rate = rate;
        }

        public double add(double value)
        {
                return multiplier() * value;
        }

        private double multiplier()
        {
                return (100 + rate) / 100;
        }
}
```

The responsibility for calculating VAT has been moved away from the `Invoice` class and into the `Vat` class. Having a separate `Vat` sprout class with a public `add` method also means that it can be tested in isolation away from the `Invoice` class. What was previously the private `addVat` method on the `Invoice` class is now a public method on the `Vat` class which can be tested directly. The `Vat` class still has a private method, `multiplier`, but it can be easily tested using triangulation. Writing more tests also feels more comfortable:

```
@Test
public void vatAt20pc()
{
        assertEquals(120, new Vat(20).add(100), 1);
}

@Test
public void vatAt15pc()
{
        assertEquals(115, new Vat(15).add(100), 1);
}

@Test
public void vatAt17_5pc()
{
        assertEquals(117.5, new Vat(17.5).add(100), 1);
}

@Test
public void vatAt0pc()
{
        assertEquals(100, new Vat(0).add(100), 1);
}

@Test
public void vatAt100pc()
{
        assertEquals(200, new Vat(100).add(100), 1);
```

```
}
```

Finally

By using triangulation or sprout classes or a combination of the two you can fully and easily test private methods without the need to compromise encapsulation or the design of public interfaces. Which to use depends on the complexity of the private methods being tested and/or the number of different tests that need to be written. When you have simple private methods which require few test cases, triangulation can be sufficient. As the complexity and number of test cases increases, sprout classes become a better solution.

Regardless of how you decide to test private methods, keep your interfaces clean, don't change them for the sake of testing and measure your test coverage.

The example code is available here: https://bitbucket.org/pjgrenyer/testing-private

References

[TDD] Test Driven Development by Kent Beck. ISBN-13: 978-0321146533

[Legacy] Working Effectively with Legacy Code by Michael Feathers. ISBN-13: 978-0131177055

[Refactoring] Refactoring: Improving the Design of Existing Code by Martin Fowler. ISBN-13: 978-0201485677

Paul Grenyer

Technology Solutions for Innovators

Paul has a strong track record of solving complex problems with robust software solutions for individuals and in organisations ranging from startups to investment banks and large insurers.

He builds cohesive and efficient solutions for disparate and labour intensive business processes which improve accuracy and demonstrate real cost savings.

Connect with Paul on linkedin: https://uk.linkedin.com/in/pgrenyer