Demos

Simple Demo

```
// Hard to test
using System;
using System.Collections.Generic;
using System.Linq;

namespace TestableCodeDemos.Module1.Hard
{
    class Program // Computes the total price for the invoice based on the price of the parts, the prices of the service, and an optional discount.
    {
        static void Main(string[] args) // Entry point of the application.
        {
            var parts = decimal.Parse(args[0]); // get the price of the parts from the first argument passed into the console application and convert it into a decimal value.

            var service = decimal.Parse(args[1]); // get and convert the price of the service from the second argument.

            var discount = decimal.Parse(args[2]); // get the amount of the discount to be applied to the invoice from the third argument.

            var total = parts + service - discount; // Compute the total price by adding the price of the parts to the price of the service and subract the discount.

            Console.WriteLine("Total Price: $" + total); // print the total price out to the console window.
        }
    }
}

// The part of the application to be focused is the calculation to compute the total price of the invoices.
// Extracting the calculator logic into its own class, would make it easy to create automated unit tests for it.
```

```
// Extracted calculator Logic
// This class contains a single method called GetTotal
using System;
using System.Collections.Generic;
```

```
using System.Linq;

namespace TestableCodeDemos.Module1.Easy
{
    public class Calculator
    {
        public decimal GetTotal(decimal parts, decimal service, decimal discount)
// The method takes as input the price of the parts, price of the service, and the
amount of the discount.
        {
            return parts + service - discount; // calculates the total price in
the body of the method and returns the totoal price's output.
        }
    }
}
```

```
// Use the new Calculator class to replace the embedded logic in the hard
application.
using System;
using System.Collections.Generic;
using System.Linq;

namespace TestableCodeDemos.Module1.Easy
{
    class Program
    {
        static void Main(string[] args)
        {
            var parts = decimal.Parse(args[0]);

            var service = decimal.Parse(args[1]);

            var discount = decimal.Parse(args[2]);

            var calculator = new Calculator(); // construct a new calculator class

            var total = calculator.GetTotal(parts, service, discount); // call the
GetTotal method on this class, passing in the user's input and receiving the total
price as our output.

            Console.WriteLine("Total Price: $" + total);
        }
    }
}
```

```
// In order to test the functionality of the Calculator class, we create a test
class, also known as a test fixture called CalculatorTests.
using System;
using System.Collections.Generic;
using System.Linq;
using NUnit.Framework;

namespace TestableCodeDemos.Module1.Easy
{
    [TestFixture] // This attribute ells MUnit that the class contains unit tests
that should be run by test runner
    public class CalculatorTests
    {
        private Calculator _calculator; // contains subject under test, that is
the invoice class that we're testing.

        [SetUp] //This attribute tells NUnit that the method should be called
before each and every test is run to ensure that the test is up properly.
        public void SetUp() // marked with a [SetUp] attribute.
        {
            _calculator = new Calculator(); //construct the Calculator class so
that it is initialized before each and every test.
        }

        [Test] // This attribute tell NUnit that the method contains a test that
should be run each and every time the unit tests are run.
        public void TestGetTotalShouldReturnTotalPrice() // The name of the method
should make it clear to other developers what is being tested by the test method
        {
            var result = _calculator.GetTotal(1.00m, 2.00m, 0.50m); // execute the
action being tested, that is call the GetTotal method, passing in three numbers,
that is $1 for parts, $2 for service, and 50 cents for a discount. Assign the
return value to the variable result.

            Assert.That(result, Is.EqualTo(2.50m)); // Assert that the result is
equal to $2.50.
        }
    }
}
```

Seams

In this demo we create seams in our application to make our code testable.
Imagine that we're creating a feature in our automotive invoicing application to print an invoice

- When the user clicks the `Print` button in our application, it will trigger a command to print the selected invoice.
- The command will retrieve the specified invoice from the database and will write each line of the invoice to the printer.

For this demo, we're only going to worry about printing the:

- invoice number
- total price for the invoice
- date the invoice was printed

We'll worry about the rest of the invoice fields later

```
// Difficult to test
using System;
using System.Collections.Generic;
using System.Linq;
using TestableCodeDemos.Module2.Shared;

namespace TestableCodeDemos.Module2.Hard
{
    public class PrintInvoiceCommand // This class contains all the logic to print
an invoice
    {
        public void Execute(int invoiceId) // takes an invoiceID from the user
interface of the application.
        {
            var database = new Database(); // Creates an instance of a class
called Database, which provides access to the invoice data in the database.

            var invoice = database.GetInvoice(invoiceId); // gets an invoice from
the database using the specified invoiceID

            Printer.WriteLine("Invoice ID: " + invoice.Id); // uses a static
WriteLine method on the Printer class to write lines of text to the printer. It
prints invoiceID.

            Printer.WriteLine("Total: $" + invoice.Total); //It prints Total price
for the invoice.

            var dateTime = DateTime.Now; // gets the current date from the static
Now property of the DateTime Structure.

            Printer.WriteLine("Printed: " + dateTime.ToShortDateString()); //
Print the date that the inovice was printed to the printer.
        }
    }
}
```

This code is hard to test in isolation because of the dependencies on the:

- Database,
- Printer classes, and
- the DateTime structure

In order to test this code, we would need an actual database and printer connected plus we'd also need access to the current OS date and time.
In addition, we'd also need an actual invoice in the database with a known invoiceID and a known total price in order to verify the results.

This code is hard to test in insolation for three different reasons

- **There's a dependency on the database:** when we use the `new` keyword to construct a dependency inside of a class, we're tightly coupled to that class, so we can't test in isolation from the dependency or any of its dependencies either.
- **Using a static method on the Printer class rather than an instant method of a printer object:** This creates tight coupling between the PrintInvoice CommandClass and the Printer class.
- **Calling the static Now property on the DateTime structure:** This problem looks similar to the previous problem with the static method call in the Printer Class; however because the DateTime structure is part of the .NET Framework and thus not part of our code, we cannot change this code ourselves to make it easily testable, so we're going to have to take a slightly different approach to solve this problem using a `wrapper class`.

How to refactor this code to make it easily testable

- ***Use Interfaces***: An interface in programming is code that describes the actions that an object can perform. It describes the necessary inputs for each action and the expected outputs. In essence, an interface acts as a contract between a class using the specified interface and every class that implements the specific interface. Because of this, we can use an interface as a placeholder for an actual class and then substitute in any class that implements that interface later.

```
using System;
using System.Collections.Generic;
using System.Linq;
using TestableCodeDemos.Module2.Shared;

namespace TestableCodeDemos.Module2.Easy
{
    public interface IDatabase // defines the contract between a caller which
calls methods of the database and the class(es) that will implement this interface
    {
        Invoice GetInvoice(int invoiceId);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using TestableCodeDemos.Module2.Easy;
```

```
namespace TestableCodeDemos.Module2.Shared
{
    public class Database : IDatabase // The Database clas implements the
IDatabase Interface, this means that in order for the class to be valid at compile
time, It must implement all of the methods exactly as specified in the IDatabase
Interface, that is the method must have the same name, take the same input
arguments, and return the same type of output.
    {
        public Invoice GetInvoice(int invoiceId)
        {
            throw new System.NotImplementedException();
        }
    }
}
```

Note

We also do the same for exisity Printer class.

- We create an interface called Iprinter that defines the methods of the printer
- Then, in the Printer class, we refactor our existing WriteLine method, which was a static method of the class, into an instance method of the object that implements the WriteLine method we defined in our interface.
  To solve the problem with the dependency on the .NET Framework's DateTime structure
- Since this code is part of a third party API, that is Microsoft's .NET Framework, we can't just go out and refactor their code to make it easier for us to test
- rather, we're going to have to create a wrapper class which will wrap the DateTime structure and make it easier for us to use in a testable way.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace TestableCodeDemos.Module2.Easy
{
    public interface IDateTimeWrapper
    {
        DateTime GetNow(); // returns a DateTime structure just like the Now
property of the existing DateTime structure
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestableCodeDemos.Module2.Easy
{
    public class DateTimeWrapper : IDateTimeWrapper
    {
        public DateTime GetNow()
        {
            return DateTime.Now;
        }
    }
}
```

- The point of a wrapper class is to decouple our code from third-party code that makes our code difficult to test in isolation.
- We want to make our wrapper class as simple as possible to avoid any logic that might require unit testing.
- The reason for this is that if we keep our wrapper class free of logic, then we don't need to write unit tests for the wrapper class or the class that it's wrapping.
- This is because we make the assumption that the third party that created this code has fully tested it and that it works as specified.
- We can, however, test this functionality as part of our full system tests, which should be part of our comprehensive suite of tests for the application.

The PrintInvoiceCommand now been refactored for testability

- Rather working with the:
  - Database class,
  - Printer class and
  - DateTime structure directly.
- Now we can define three interfaces, that is:
  - IDatabase
  - IPrinter, and
  - IDateTimeWrapper to act as placeholders for our actual Database class, Printer class and the DataTime Structure.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace TestableCodeDemos.Module2.Easy
```

```
{
    public class PrintInvoiceCommand
    {
        private readonly IDatabase _database; // Placeholder for our actual
Database Class
        private readonly IPrinter _printer; // Placeholder for our actual Printer
Class
        private readonly IDateTimeWrapper _dateTime; // Placeholder for our actual
DateTime Strurture

        // The constructor takes three objects that must conform to the IDatabase,
IPrinter and IDateTimeWrapper interfaces
        public PrintInvoiceCommand(
            IDatabase database,
            IPrinter printer,
            IDateTimeWrapper dateTime)
        { // Assign the three objects to three fields in our class so that we can
access them from any of the methods inside of this class. This is referred to as
Dependency Injection via constructor Injection.
            _database = database;
            _printer = printer;
            _dateTime = dateTime;
        }

        public void Execute(int invoiceId)
        {
            var invoice = _database.GetInvoice(invoiceId); // get the invoice form
the database object that implements the IDatabase Interface.

            _printer.WriteLine("Invoice ID: " + invoice.Id);
            /*Print the first two lines of text to the printer using the printer
object that implements the IPrinter Interface*/
            _printer.WriteLine("Total: $" + invoice.Total);

            var dateTime = _dateTime.GetNow(); // get the current system date and
time from the DateTimeWrapper object that implements the IDateTimeWrapper
Interface

            _printer.WriteLine("Printed: " + dateTime.ToShortDateString()); //
write the date printed to the printer.
        }
    }
}
```

Testing the PrintInvoiceCommand

```
using System;
using Moq;
using NUnit.Framework;
```

```
using TestableCodeDemos.Module2.Shared;

namespace TestableCodeDemos.Module2.Easy
{
    [TestFixture]
    public class PrintInvoiceCommandTests
    {
        /*There are five private fields in of our TestFixture,a field called
command of the type PrintInvoiceComman which is our subject under test, three
fields for our mock objects, that is our test doubles, a mockDatabase which is a
mock class of type IDatabase, a mockPrinter which is a mock of thpe IPrinter, and
a mockDateTime wrapper which is of type IDateTimeWrapper, these three mock
classes, which are part of our mocking framework, will serve as test doubles for
our Database, Printer and DateTimeWrapper classes, respectively. the last field is
for our invoice class which we'll use as a stan-in for a real invoice during our
tests.*/
        private PrintInvoiceCommand _command;
        private Mock<IDatabase> _mockDatabase;
        private Mock<IPrinter> _mockPrinter;
        private Mock<IDateTimeWrapper> _mockDateTime;
        private Invoice _invoice;

        /*There are two constants called InvoiceID which is set to the integer 1,
and Total which is set to 1.23. We'll use these two fields to hold test values for
our invoiceID and total price, respectively, also there is a static readonly
DateTime structure called Date which will hold a test value for our system date
that is Feb 3,2001, NOte that in the .NET Framework the DateTime structure doesn't
support literals, so we can't create it as a constant like we did with our
InvoiceID and Total Price variables, However we can create a static readonly field
to store our DateTime test value instead.*/
        private const int InvoiceId = 1;
        private const decimal Total = 1.23m;
        private static readonly DateTime Date = new DateTime(2001, 2, 3);

        [SetUp]
        public void SetUp()
        {
            /*Create and invoice class and assign it the test value we created for
invoiceID and Total*/
            _invoice = new Invoice()
            {
                Id = InvoiceId,
                Total = Total
            };

            /*Construct our three mock classes to act as test doubles that is our
mockDatabase, mockPrinter, and mockDateTime wrapper. A mock object provides us
with complete control over its behaviour so that we can tell its methods to return
certain values and verify the methods have called with certain values. For example
our mockDatabase class will act as a test double for our real Database class. We
can make it behave like the real Database class by telling it to return specific
values when certain methods are called with specified parameters.*/
            _mockDatabase = new Mock<IDatabase>();
            _mockPrinter = new Mock<IPrinter>();
```

```
            _mockDateTime = new Mock<IDateTimeWrapper>();

            /*we'll setup the GetInvoice method so that when a user calls this
method with a specified InvoiceID, which in this case is 1, the mockDatabase class
will return the test invoice that we set up a few lines earlier.*/
            _mockDatabase
                .Setup(p => p.GetInvoice(InvoiceId))
                .Returns(_invoice);

            /*we'll setup the GetNow method so that it returns the test date that
we created earlier in the TestFixture*/
            _mockDateTime
                .Setup(p => p.GetNow())
                .Returns(Date);

            /*we create a new instance of the PrintInvoiceCommand class, passing
in our mockDatabase object, our mockPrinter object, and our mockDateTime wrapper
object. Note that the object propert of the mock class returns an instance of the
mock object itself. This is what we pass into the constructor of the subject under
test.*/
            _command = new PrintInvoiceCommand(
                _mockDatabase.Object,
                _mockPrinter.Object,
                _mockDateTime.Object);
        }

        [Test]
        public void TestExecuteShouldPrintInvoiceNumber() // This test will verify
that the correct invoice number is printed when the executed method is called.
        {
            // Call the execute method on the command passing in our specified
invoiceID using the test value we defined at the top of the tests, which was the
integer 1
            _command.Execute(InvoiceId);

            // we use our mockPrinter object to verify that the WriteLine method
has been called with the text invoiceID: 1 passed in as an argument, we specify
that this method should have been called exactly one time during the test. So if
the WriteLine method on the mockPrinter has been called with this text string
exactly once, then the test passes. However is the writeLine method has not been
called with this text string exacly one time, then the test fails.
            _mockPrinter
                .Verify(p => p.WriteLine("Invoice ID: 1"),
                    Times.Once);
        }

        [Test]
        public void TestExecuteShouldPrintTotalPrice()
        {
            /*We do the same for our Total price in this second unit test*/
            _command.Execute(InvoiceId);

            _mockPrinter
                .Verify(p => p.WriteLine("Total: $1.23"),
```

```
                    Times.Once);
        }

        [Test]
        public void TestExecuteShouldPrintTodaysDate()
        {
            /*We do the same for our Total price in this third unit test*/
            _command.Execute(InvoiceId);

            _mockPrinter
                .Verify(p => p.WriteLine("Printed: 2/3/2001"),
                    Times.Once);
        }
    }
}
```

***There are few things that we can do to reduce duplication and clean up our test class. The above can be refactored usig an automocking Framework called `AutoMoqer`

- `AutoMoqer:` eliminates much of the setup of our mock objects by automatically creating them as needed.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Moq;
using Moq.AutoMock;
using NUnit.Framework;
using TestableCodeDemos.Module2.Shared;

namespace TestableCodeDemos.Module2.Easy
{
    [TestFixture]
    public class PrintInvoiceCommandTestsWithAutoMocker
    {
        private PrintInvoiceCommand _command;
        private AutoMocker _mocker; // instead of having three mock classes, we
have a single field called mocker of type AutoMoqer, Automoqer is our automocking
framework
        private Invoice _invoice;

        private const int InvoiceId = 1;
        private const decimal Total = 4.50m;
        private static readonly DateTime Date = new DateTime(2001, 2, 3);

        [SetUp]
        public void SetUp()
        {
            _invoice = new Invoice()
```

```
            {
                Id = InvoiceId,
                Total = Total
            };

            _mocker = new AutoMocker(); // instead of creating our three mock
classes, we'll create an AutoMoqer instead.

           /*Then instead of using each mock class directly, we'll call the
GetMock method of our AutoMoqer class, passing in an IDatabase interface as a
generic method type parameter. This will return an automatically generated mock
that implements the IDatabase interface */
            _mocker.GetMock<IDatabase>()
                .Setup(p => p.GetInvoice(InvoiceId))
                .Returns(_invoice);

            _mocker.GetMock<IDateTimeWrapper>()
                .Setup(p => p.GetNow())
                .Returns(Date);

            _command = _mocker.CreateInstance<PrintInvoiceCommand>(); // we'll
create our subject under test using the AutoMoqer's Create Method,passing in a
PrintInvoiceCommand class as a generic method type parameter. This will tell
AutoMoqer to create a new PrintInvoiceCommand and automatically create and inject
the three mock dependencies that need to be injected into the constructor of this
class
        }

        [Test]
        /*we'll replace the three independent test method with a single test
method using test cases instead. The TestCase attribute tells NUnit that this test
method should be executed multiple times, once for each specified test case. The
character string inside of the TestCases will get passed in as an argument into
the line parameter of the Test method */
        [TestCase("Invoice ID: 1")]
        [TestCase("Total: $4.50")]
        [TestCase("Printed: 2/3/2001")]
        public void TestExecuteShouldPrintLine(string line)
        {
            _command.Execute(InvoiceId);

            _mocker.GetMock<IPrinter>()
                .Verify(p => p.WriteLine(line),
                    Times.Once);
        }
    }
}
```