**ThoughtWorks Code Assignment - PROBLEM TWO: SALES TAXES:**

Language Used: C#

Development Tool: Visual Studio 2012 (.Net Framework 4.5)

Description: Windows Console Application

* Application Architecture: It’s divided into 3 projects;
  + TaxCalculation.ConsoleApp – To display the calculated output to command line.
  + TaxCalculation.Domain – Holds all classes and logic that does all the tax calculations.
  + TaxCalculation.Tests – Unit test for the application.
* External Libraries:
  + I have used NUnit Framework and RhinoMocks.
* Application build:
  + I have used MSBuild (Microsoft Build Engine).

1. Even though I haven’t followed a particular architectural pattern for the overall application, I have followed an important principle of software engineering which is Separation of Concerns (SoC). So the application is easily maintainable because I have modularized it and it’s also extensible because I have used interfaces (meaning a new implementation for an existing interface is possible).
2. If I have to port this application into MVC architecture as an example, I could easily do that. The layer TaxCalculation.Domain can be reused for that.
3. I tried to follow the SOLID principles of class design: For example, the following interface ITax which is implemented by Class Tax has a single responsibility which is to calculate tax “decimal CalculateTax(decimal itemPrice);”, Rounding is handled by another class Rounding instead of it being the responsibility of Tax.

namespace TaxCalculation.Domain.Interfaces

{

public interface ITax

{

IRounding Rounding { get; }

decimal Rate { get; }

decimal CalculateTax(decimal itemPrice);

}

}

1. For testing framework I chose to use NUnit because it has a lot more functionality and it’s flexible, for example, instead of having a Try Catch block for Expected exception message assertion I can use attributes:

[TestCase("SalesData.txt", Description = "Invalid File Test",

ExpectedException = typeof(FileNotFoundException))]

[TestCase("SalesData.txt",Description = "Positive Test")]

public void GetOrdersVerifiedMethodCallTestCase(string filePath)

{

mock.Expect(s => s.ReadOrders(filePath)).Return(listOrders);

ordersRepositoryTestClass.ActOnIOrdersRepository(mock, filePath).ToList();

//Verifying that the method (ReadOrders) was was called.

mock.VerifyAllExpectations();

}

1. For mocking I have used Rhino Mocks which allows me to create a mock object without having add an actual repository from an application, I just mock the interface:

mock = MockRepository.GenerateMock<IOrdersRepository>();

1. Two design patterns I have incorporating into the application are:

Factory Pattern: Used it to create a LineItem –

LineItemFactory.GetLineItem("book", 12.49M,1,ItemType.Exempt)

Decorator Pattern: Used it to extend a LineItem functionality –

LineItemTaxDecorator: ILineItem