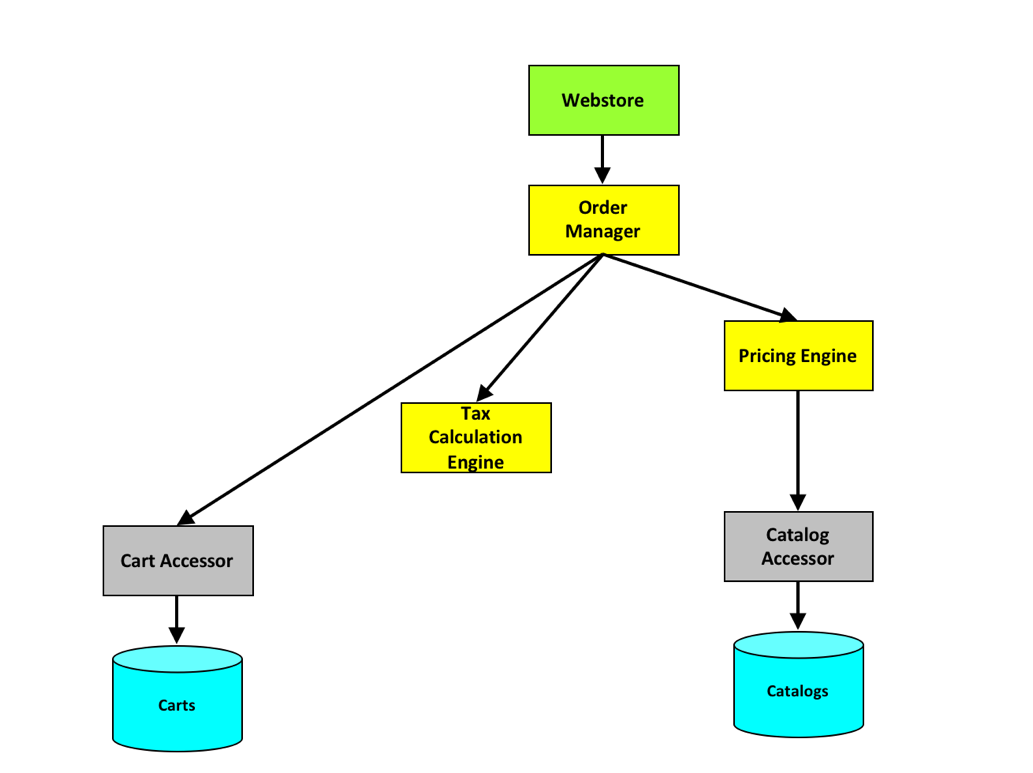
Sales Tax Activity

The reference application follows the patterns and practices we encourage in our software solutions we create. The software we write looks a lot like our reference application, but a big difference is our reference implementation uses the world’s most perfect user experience, a console application.

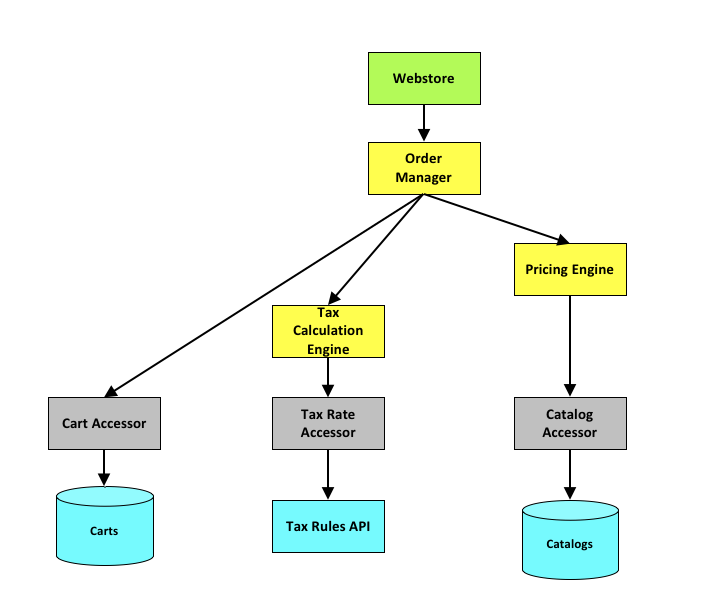
The ecommerce application isn’t a fully function ecommerce application, it has many short cuts, so we can just demo the application. One of those short cuts is that the application doesn’t do a real sales tax calculation, it just does a flat 7% for all items.

In this activity we will take you through modifying our reference application to support a 3rd party tax library. We are playing pretty loose with the term 3rd party. We wrote a really basic library that does sales tax for the state of Nebraska. Now we won’t stand behind our tax library as production ready, but it is ready enough for the purpose of our reference implementation.

Changing our reference application to support calling this 3rd party library will cause us to change our architecture. Before using this library our architecture for adding an item to the shopping cart looked like below.



Changed to call our 3rd party library will affect the architecture diagram.



To make this change we will add a new accessor into our system, a TaxRateAccessor. This accessor will call our 3rd party library to get the sales tax. In a real system this would probably be a web call to another system.

We will have to load the USATaxer project into VisualStudio and build the library. That will create a USATaxer.dll. How does the USATaxer library work, well let’s look at its unit test.

[TestMethod]

public void USATaxer\_Initialized()

{

var taxer = new USATaxerLib();

taxer.Init();

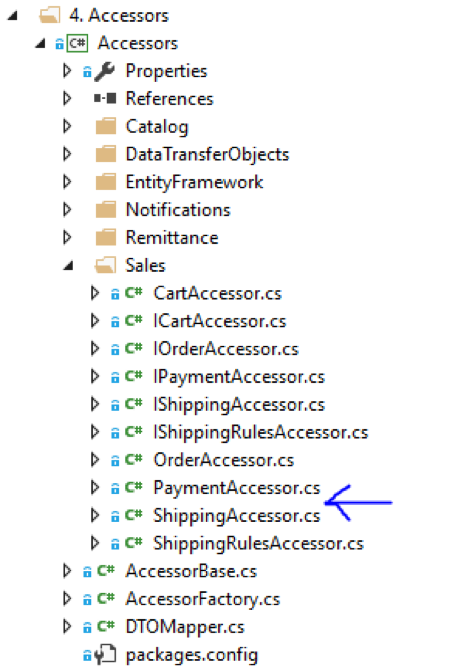
Assert.AreEqual(0.0725m, taxer.Rate("68512"));

Assert.AreEqual(0.065m, taxer.Rate("68031"));

}

Copy the USATaxer.dll to the root of our ecommerce solution and add a reference to it in our Accessors project.

The next step here is to add a TaxRateAccessor to our solution. This will require us to add a new file, a TaxRateAccessor.cs file to be exact. We will add that file to the Accessor project.



public interface ITaxRateAccessor : IServiceContractBase

{

decimal Rate(Address address);

}

class TaxRateAccessor : AccessorBase, ITaxRateAccessor

{

public decimal Rate(Address address)

{

USATaxer.USATaxerLib taxer = new USATaxer.USATaxerLib();

taxer.Init();

return taxer.Rate(address.Postal);

}

}

The TaxRateAccessor needs to be wired up into our dependency injection system. To do that we need to modify our AccessorFactory.cs file. In our reference implementation we are doing all the dependency injection ourselves. In many production systems we often use something like unity, but in our reference implementation we wanted to keep things simple.

public AccessorFactory(AmbientContext context, UtilityFactory utilityFactory) : base(context)

{

// NOTE: this is here to ensure the factories from the Manager are propogated down to the other factories

\_utilityFactory = utilityFactory ?? new UtilityFactory(Context);

AddType<ICartAccessor>(typeof(CartAccessor));

AddType<ICatalogAccessor>(typeof(CatalogAccessor));

AddType<IEmailAccessor>(typeof(EmailAccessor));

AddType<IOrderAccessor>(typeof(OrderAccessor));

AddType<IEmailAccessor>(typeof(EmailAccessor));

AddType<IPaymentAccessor>(typeof(PaymentAccessor));

AddType<IShippingAccessor>(typeof(ShippingAccessor));

AddType<ISellerAccessor>(typeof(SellerAccessor));

AddType<IRemittanceAccessor>(typeof(RemittanceAccessor));

AddType<IShippingRulesAccessor>(typeof(ShippingRulesAccessor));

AddType<ITaxRateAccessor>(typeof(TaxRateAccessor));

}

At this point we have a new accessor, and it should be supported by our DI framework. All that is left would be to use it. But wait, we should test it first. Add a very simple test for TaxRateAccessor.

[TestClass]

public class TaxAccessorTests

{

private ITaxRateAccessor CreateAccessor()

{

var context = new Common.Contracts.AmbientContext();

var factory = new AccessorFactory(

context,

new Utilities.UtilityFactory(context));

return factory.CreateAccessor<ITaxRateAccessor>();

}

[TestMethod]

public void TaxAccessor\_LincolnNe()

{

var accessor = CreateAccessor();

var lincoln = new Address()

{

Postal = "68512"

};

accessor.Rate(lincoln);

}

}

Now that we have a test (and hopefully it passes) we can finally use this accessor in our code. To do this we will have to change our TaxCalculationEngine. This can be found in the TaxCalculationEngine.cs file in the Engine project.

public WebStoreCart CalculateCartTax(WebStoreCart cart)

{

if (cart != null && cart.BillingAddress != null && !string.IsNullOrWhiteSpace(cart.BillingAddress.Postal))

{

var taxRate =

AccessorFactory.CreateAccessor<ITaxRateAccessor>()

.Rate(cart.BillingAddress);

foreach (var item in cart.CartItems)

{

cart.TaxAmount += Math.Round(item.ExtendedPrice \* taxRate, 2);

}

// update the cart total with the tax amount

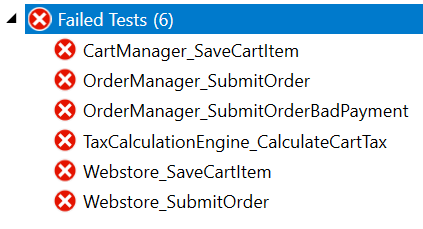
cart.Total += Math.Round(cart.TaxAmount, 2);

}

return cart;

}

Now after wiring in to use this new TaxRateAccessor something will happen. We will break some unit tests. Now this is okay because we have changed out taxes are calculated. Go through and update the unit tests to use what we are now calculating for tax.



This activity shows how to extend our existing reference implementation by calling a 3rd party library to do sales tax calculations. This example also shows how volatility-based decomposition helps to prevent a simple change from exploding through all of the source. The only existing code that changed was the TaxCalculationEngine.

# Bonus

Bonus 1: Update code to support only charging sales tax on shippable goods.

Bonus 2: Update to support tax exempt products.

* Add a script to the database project to add a tax-exempt column to the database.
* Update the DTOs for a product to include this column.
* Update TaxCalculationEngine to check if the product is tax exempt.