

Domain Modelling with Habanero

Part 2: Growing the domain model with relationships

By Peter Wiles, Chillisoft Solutions

Summary

In the previous article we created a Pizza class for Bob's Pizza Place's new computerised order system. Just one domain class is not particularly useful, so this time we will create add Orders to our domain model so that Bob can start recording orders in the system instead of writing them out by hand.

Once again, we will do with without any tools – just the humble IDE and hand-coded classes. We will continue to eschew doing a graphical user interface because I don't want to add those complexities on top of understanding domain modelling using Habanero.

What you need

The sample code in the article and in the attached solution is written in C# using Visual Studio 2008 and .NET 3.5. You can write the sample code without Visual Studio, but this article assumes you are using it.

The Habanero libraries required for this tutorial are contained within the zip this document was found (in the lib folder). If you wish to download the latest full Habanero release, please go to <http://www.habanerolabs.com>. Note that the libraries used here are from release 2.3.2 and the code will not work against version 2.3.1.

Who this is for

This article is best read in sequence after Part 1 in the series and the sample code continues from where that article left off. Once again, some knowledge of Domain Driven Design [Evans] concepts would be helpful but isn't necessary, and an understanding of the Active Record pattern as described by Martin Fowler in Patterns of Enterprise Application Architecture [Fowler] would also add much value to what is done here. I would highly recommend reading those sources to understand the theory behind what Habanero is trying to do.

This article uses hands-on code and is primarily aimed at developers who are interested in using Habanero and understanding it from first-principles. If you find you have trouble following the steps in the code, you can load up the solution included and look at the final product (just set BobsPizzaPlace.Console as the startup project if you want to run it). Please supply any feedback, positive or negative, requests or comments, at the Habanero forum at <http://www.habanerolabs.com>

Because I did not want to increase the complexity of the example code there is no error-handling done in this article. We advocate intense use of Test-Driven-Development, but I have left testing out of this article – it will be covered in a future article. The sample code also isn't necessarily the best

possible code because it has been written specifically to demonstrate concepts and to be easy to write as you follow along.

The User Story: Enter an Order

A customer enters the Pizza place, decides on a number of pizzas to buy and places an order with the cashier. The cashier enters the order into the system and prints out two copies of a ticket – one for the customer and one for the kitchen staff.

This is the next user story we need to implement. It's clear from the story that we will need the concept of an Order in our domain model. It's also clear that an order can be made up of one or more pizzas. We will call these Order Items.

Creating the Domain Model

The new domain model classes needed to implement this user story could look something like this:

```
public class Order {
    public int OrderNumber { get; set; }
    public List<OrderItem> OrderItems { get; private set; }
}

public class OrderItem {
    public Pizza Pizza { get; set; }
    public int NumberOfItems { get; set; }
}
```

If you're following along, go ahead and create these classes in your BO project now. (Note: I'm assuming you've loaded if you haven't read

These classes are very simple and to the point, and seem intuitive to use. And if we were never saving the objects, this would be great. The problem comes in persisting and later loading – how do you store that list of Order Items when you save the Order? You could serialise the object tree using .NET's serialisation. But what if your object tree got really large?

These kinds of domain entities are often called Business Objects, named so because they represent business concepts or business data. There is a set of functionality and behaviour that is very often required of these kind of objects in business applications. For instance, a user might edit the number of items on an Order Item, but then might decide to cancel the change they've made – it would be great if the object remembered its previous state. Or, maybe only someone with enough authorisation should be able to change an order once it's been captured. We'd also like to always ensure that an OrderItem has both a Pizza selected and a value in the NumberOfItems field. The classes we've made here have none of that functionality available.

Habanero contains a Business Object layer that provides these features – features specifically designed for business-oriented software.

Configuring the Order and OrderItem Class definitions for Habanero

To change our Order and OrderItem classes to work in Habanero we once again create an xml description of the classes. This should be added below, or above, the xml description of the Pizza class, but inside the <classes> tags.

```
<class name="Order" assembly="BobsPizzaPlace.B0">
  <property name="OrderID" type="Guid" />
  <property name="OrderNumber" type="Int32" />
  <primaryKey>
    <prop name="OrderID" />
  </primaryKey>
</class>
<class name="OrderItem" assembly="BobsPizzaPlace.B0">
  <property name="OrderItemID" type="Guid" />
  <property name="OrderID" type="Guid" />
  <property name="PizzaID" type="Guid" />
  <property name="NumberOfItems" type="Int32" />
  <primaryKey>
    <prop name="OrderItemID" />
  </primaryKey>
</class>
```

There are a few things needing comment in this xml. Firstly, I've added ID fields that weren't in our original class. Although Habanero can write to a flat file, like we saw in the last article, the most common scenario for its use is to write your data to a database. For this reason, the primary key field, or ID, is necessary. It's also the field I use to map related objects. For example, in the Order Item xml description, you can see an OrderID field. This is the field we will use to link an OrderItem to an Order, and you can imagine it mapping to a database field that is a foreign key to the Order table.

Adding the Relationships between Orders, OrderItems and Pizzas

I've also got a PizzaID field on OrderItem, which links the OrderItem with the Pizza on that item. One thing that has been omitted from this xml is the description of these relationships. I'll add those now:

```
<class name="Order" assembly="BobsPizzaPlace.B0">
  <property name="OrderID" type="Guid" />
  <property name="OrderNumber" type="Int32" />
  <primaryKey>
    <prop name="OrderID" />
  </primaryKey>
  <relationship name="OrderItems" type="multiple" relatedClass="OrderItem"
relationshipType="Composition">
    <relatedProperty property="OrderID" relatedProperty="OrderID" />
  </relationship>
</class>
<class name="OrderItem" assembly="BobsPizzaPlace.B0">
  <property name="OrderItemID" type="Guid" />
  <property name="OrderID" type="Guid" />
  <property name="PizzaID" type="Guid" />
  <property name="NumberOfItems" type="Int32" />
  <primaryKey>
    <prop name="OrderItemID" />
  </primaryKey>
  <relationship name="Pizza" type="single" relatedClass="Pizza"
relationshipType="Association">
    <relatedProperty property="PizzaID" relatedProperty="PizzaID" />
  </relationship>
</class>
```

The first relationship added is the relationship between Order and its items. This is a multiple relationship because there are 1 or more items on every Order. Note the properties that the relationship is built upon are described, and in this case there is just one property – OrderID, and it

has the same name in both classes. If, for example, you would rather call your ID fields simply “ID”, then this line would read:

```
<relatedProperty property="ID" relatedProperty="OrderID" />
```

One final thing to note is the type of the relationship – this is a classic example of a composition relationship, but that will be discussed in detail a little later.

The other relationship described in this xml is the one between an OrderItem and a Pizza – every OrderItem has one Pizza related to it. This is why the relationship is described as “single”. This relationship is the standard relationship type – Association, but again that will be discussed a little later.

Changing the C# Classes

Now we need to make some adjustments to the classes themselves to use the Business Object functionality Habanero provides:

The updated Order class:

```
using Habanero.BO;

namespace BobsPizzaPlace.BO
{
    public class Order : BusinessObject
    {
        public virtual int OrderNumber
        {
            get { return ((int)(base.GetPropertyValue("OrderNumber"))); }
            set { base.SetPropertyValue("OrderNumber", value); }
        }

        public virtual BusinessObjectCollection<OrderItem> OrderItems
        {
            get
            {
                return Relationships.GetRelatedCollection<OrderItem>("OrderItems");
            }
        }
    }
}
```

And the updated OrderItem class:

```
using Habanero.BO;

namespace BobsPizzaPlace.BO
{
    public class OrderItem : BusinessObject
    {
        public virtual Pizza Pizza
        {
            get { return Relationships.GetRelatedObject<Pizza>("Pizza"); }
            set { Relationships.SetRelatedObject("Pizza", value); }
        }

        public virtual int NumberOfItems
        {
            get { return ((int)(base.GetPropertyValue("NumberOfItems"))); }
            set { base.SetPropertyValue("NumberOfItems", value); }
        }
    }
}
```

```
}
```

Note that although the ID fields are in the xml they are unnecessary in the classes – Habanero will populate those fields and use them for relating the objects without them needing to be on your domain model.

Note also that from the outside the classes look almost exactly the same as they used to. The only difference is the new return type of OrderItems – `BusinessObjectCollection<OrderItem>`. The `BusinessObjectCollection` class is a more fully featured List – it fires `BusinessObject` related events and tracks the state of the `BusinessObjects` within it. These features will be covered in a later article.

Capturing an Order

Now let's implement our original user story. For the purposes of this example I'm going to ignore printing to a printer and simply display the order on the screen (in the console). First we'll add a "Capture an Order" menu item to the menu:

```
private static void DisplayMenu(BusinessObjectCollection<Pizza> pizzas)
{
    int i = 0;
    pizzas.ForEach(pizza =>
        System.Console.Out.WriteLine("{0}. {1} {2:0.00}",
            i++, pizza.Name, pizza.Price));

    System.Console.Out.WriteLine("A. Add a new pizza");
    System.Console.Out.WriteLine("D. Delete a pizza");
    System.Console.Out.WriteLine("O. Capture an order");
    System.Console.Out.WriteLine("Q. Quit");
}
```

Then we'll add the code to handle the menu item:

```
while (input != "Q")
{
    if (input == "A")
    {
        AddNewPizza();
        pizzas.Refresh();
    }
    else if (input == "D")
    {
        System.Console.Out.WriteLine("Which pizza would you like to delete? ");
        int pizzaNumber = Convert.ToInt32(System.Console.ReadLine());
        DeletePizza(pizzas[pizzaNumber]);
    }
    else if (input == "O")
    {
        CaptureOrder(pizzas);
    }
    else
    {
        int pizzaNumber = Convert.ToInt32(input);
        EditPizza(pizzas[pizzaNumber]);
    }
    DisplayMenu(pizzas);
    input = System.Console.ReadLine().ToUpper();
}
```

Now I'll add the completely new methods that capture the Order:

```
private static void CaptureOrder(BusinessObjectCollection<Pizza> pizzas)
{
```

```

        Order order = new Order();
        CaptureOrderItemsForOrder(pizzas, order);
        order.Save();
    }

```

For now we won't display the order after capturing it. We still need to implement the `CaptureOrderItemsForOrder()` method, but so far we've created an `Order` object and after capturing the order items we will simply save the order. By doing this, the `OrderItems` linked to the `Order` will be automatically saved. This is because we set the `OrderItems` relationship up as a composition relationship, which implies that the `OrderItems` compose the `Order`, or are "part of" the order and must be treated as a unit with the order. (Note: although we won't be doing this in this article, if you then delete the `Order` all the `OrderItems` will be deleted along with it – because the relationship type is composition Habanero treats them as a unit.)

So let's create that `CaptureOrderItemsForOrder()` method:

```

private static void CaptureOrderItemsForOrder(
    BusinessObjectCollection<Pizza> pizzas, Order order)
{
    do
    {
        System.Console.Out.WriteLine("Select a pizza (or 'x' to complete order):");
        int i = 0;
        pizzas.ForEach(pizza =>
            System.Console.Out.WriteLine("{0}. {1} {2:0.00}",
                i++, pizza.Name, pizza.Price));

        string input = System.Console.ReadLine();
        if (input.ToUpper() == "X") return;

        int numberSelected = Convert.ToInt32(input);
        Pizza selectedPizza = pizzas[numberSelected];
        System.Console.Out.WriteLine(
            "{0} selected. How many?", selectedPizza.Name);
        int numberOfPizzas = Convert.ToInt32(System.Console.ReadLine());

        OrderItem orderItem = new OrderItem
            {Pizza = selectedPizza, NumberOfItems = numberOfPizzas};
        order.OrderItems.Add(orderItem);
    } while (true);
}

```

In this method we are looping until the user types 'X', and each time we loop we're creating a new `OrderItem` with the selected pizza and the entered number of items. Then we're adding the `OrderItem` to the `Order` in the line:

```
order.OrderItems.Add(orderItem);
```

That's it! When we save the `Order`, the `OrderItems` are saved too.

Checking on the Saved Data

Let's have a look at the xml data store after creating an order with two items. The following xml is taken from my data store. Yours won't be formatted like this, or even in this order – I've manually formatted it to represent the data structure:

```

<BusinessObjects>
  <bo __tn="Pizza" __an="BobsPizzaPlace.B0"
    PizzaID="cfae6ccb-7ee4-47e5-a28b-b333154ae06b"
    Name="Margherita" Price="25" />
  <bo __tn="Pizza" __an="BobsPizzaPlace.B0"

```

```

        PizzaID="abaf90ae-6237-4c39-8b17-955cd3749e9f"
        Name="Pomodoro" Price="30" />

<bo __tn="Order" __an="BobsPizzaPlace.BO"
    OrderID="b811b077-63a8-457a-bc4c-d75dfa6d2651"
    OrderNumber="" />
    <bo __tn="OrderItem" __an="BobsPizzaPlace.BO"
        OrderItemID="3d6bb8fc-e2ba-493c-a1d8-8e9494f3a8e6"
        OrderID="b811b077-63a8-457a-bc4c-d75dfa6d2651"
        PizzaID="cfae6ccb-7ee4-47e5-a28b-b333154ae06b"
        NumberOfItems="2" />
    <bo __tn="OrderItem" __an="BobsPizzaPlace.BO"
        OrderItemID="1f0fe831-7727-4005-b06f-e002e116dc5b"
        OrderID="b811b077-63a8-457a-bc4c-d75dfa6d2651"
        PizzaID="abaf90ae-6237-4c39-8b17-955cd3749e9f"
        NumberOfItems="1" />
</BusinessObjects>

```

The OrderItems in the xml are linked to the correct Order and Pizza using the fields we defined in the xml description because of the relationships we also defined.

Configuring the Order Number

You will notice the OrderNumber field of an Order is not being populated at the moment. We would like it to be automatically populated by the system. Typically this is done by the database using an autonumber field, which Habanero then reads to retrieve the number the database created. While using the in-memory data store we can still use an autonumber field in the same way by slightly altering the field's definition in the classdefs.xml:

```

<class name="Order" assembly="BobsPizzaPlace.BO">
    <property name="OrderID" type="Guid" />
    <property name="OrderNumber" type="Int32" autoIncrementing="true" />
    <primaryKey>
        <prop name="OrderID" />
    </primaryKey>
    <relationship name="OrderItems" type="multiple" relatedClass="OrderItem"
relatedAssembly="BobsPizzaPlace.BO" relationshipType="Composition">
        <relatedProperty property="OrderID" relatedProperty="OrderID" />
    </relationship>
</class>

```

Just doing this will make the OrderNumber field automatically populated. Note that this field cannot be set to be compulsory because the number is retrieved and populated after saving, not before. If you now create an Order you will see a BOSequenceNumber entry in your data store's xml file – this is where the current value of the sequence is persisted.

Displaying the Order

The final thing I would like to do is display the order on screen after it has been captured, including the total cost of the order. To do this, we can add a call to a DisplayOrder() method:

```

private static void CaptureOrder(BusinessObjectCollection<Pizza> pizzas)
{
    Order order = new Order();
    CaptureOrderItemsForOrder(pizzas, order);
    order.Save();
    DisplayOrder(order);
}

```

We then need to add the DisplayOrder() method to our Program class:

```
private static void DisplayOrder(Order order)
{
    System.Console.Out.WriteLine("Order Number: " + order.OrderNumber);
    order.OrderItems.ForEach(
        item =>
            System.Console.Out.WriteLine(
                "Pizza: {0}, Number: {1}, Subtotal: {2:0.00}",
                item.Pizza.Name, item.NumberOfItems, item.Subtotal));
    System.Console.Out.WriteLine("Total: {0:0.00}", order.Total);
}
```

If you add this code you will get some compile errors because we don't currently have a Subtotal property on our OrderItem class or a Total property on the Order class. Let's add these to their respective classes – to the OrderItem class add the following:

```
public decimal Subtotal
{
    get { return this.Pizza.Price*NumberOfItems; }
}
```

And to the Order class add the following:

```
public decimal Total
{
    get { return OrderItems.Sum(item => item.Subtotal); }
}
```

The code within these properties illustrates the expressive power of the kind of domain model Habanero provides. The call to this.Pizza in the Subtotal property will retrieve the Pizza that is linked to the OrderItem. This object is only loaded when it is needed, so until you make the call to the Pizza property the actual Pizza object is not loaded, but will be when the call is made.

The Sum method in the Total property is a Linq extension method to IEnumerable<T>, so it's available on the OrderItems collection as it implements IEnumerable<T>.

Now, when running the program and entering a sample order, it displays it back to me as follows:

```
Order Number: 1
Pizza: Pomodoro, Number: 1, Subtotal: 30.00
Pizza: Margherita, Number: 2, Subtotal: 50.00
Total: 80.00
```

You can see the Order Number has been automatically populated, and the totals and subtotals are calculated correctly. At the same time, most of the logic and business rules of the system are contained in the Business Object layer – things like how to calculate the total of an Order or how Orders, OrderItems and Pizzas are related.

More on the Relationship Types

In the example code in this article we used a composition relationship (Order.OrderItems) and an association relationship (OrderItem.Pizza). There is also a third relationship type supported by Habanero called aggregation, which is a sort of looser composition. The differences between these relationship types are subtle but quite extensive in the way the objects are treated. I'll end this article with a brief "by example" description of these three relationship types:

Associations are the most common relationship type, and represent a loose association between two things. For example, the relationship between a Person and a Car would normally be an association because a Person can exist independently of a Car and a Car can exist independently of a

Person. How this affects Habanero is in the way object trees are persisted. Let's say you've previously created and saved a Person object, and now add a Car object to their Cars collection. If you save the Person (say you changed their address details) then changes to the Car (perhaps a paint job) wouldn't be saved at the same time because the Car is independent of the Person.

An Order and its OrderItems is the most common example of **composition**. If you change an OrderItem (perhaps the number of items) then saving the Order means Habanero will save the Order and the changed OrderItems. In other words, changing the status of the OrderItem means changing the status of the Order because the Order is composed of the OrderItems.

An example of **aggregation** is the relationship between a Shipment and its Packages. This is similar to the Order/OrderItem relationship, except that Packages can be moved from Shipment to Shipment, unlike OrderItems intrinsic link to its Order. This is the only real difference – objects in compositions can't be moved between parents, where objects aggregations can.

For a more detailed discussion of relationships in Habanero please go to <http://www.habanerolabs.com/>

Conclusion

In this article we implemented an “Enter an Order” user story using a domain model.

We created Order and OrderItem classes to represent orders and the parts of an order, and then configured these to be used in Habanero.

We captured a set of OrderItems, adding them to an Order and finally displayed the Order on the screen, adding some logic to our Business Object classes, thus enriching the domain model.

We also configured the OrderNumber property of our Order class to be automatically populated by Habanero.

We finally discussed the types of relationships you can model using Habanero – associations, aggregations and compositions. Details of these will be fleshed out in later articles.

A domain model is an excellent way of storing business logic and rules in one place and of managing complexity. The rich model can be used by multiple applications to implement all kinds of user stories with little repetition. Habanero allows you to focus on the domain model itself rather than the wiring behind the scenes.

References

[Evans]

Eric Evans, Domain Driven Design: Tackling Complexity in the Heart of Software, 2004, Addison-Wesley

[Fowler]

Martin Fowler, Patterns of Enterprise Application Architecture, 2002, Addison-Wesley