

Domain Modelling with Habanero

Part 1: The first domain class

By Peter Wiles, Chillisoft Solutions

Summary

Habanero's main strength is in modelling your domain in a way that is both intuitive to use and powerful. In this article I will describe how to go about modelling a very simple domain (only one class), how to load and save domain objects from and to an in-memory data store and then how to save that to a file.

This article is done without the use of FireStarter - a modelling tool provided with Habanero for generating your domain classes. It is also done without the use of any UI. This is because it should be used to understand just what Habanero does and how you can use it as your domain model framework in any application. Other examples will show how you can use Habanero business objects with ASP.NET, Windows Forms or WPF. Further articles in this series will discuss using a larger domain model, using a database and adding more intelligence to your domain model classes.

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 assumes some knowledge of Domain Driven Design concepts which are most completely explained in Eric Evan's Domain Driven Design: Tackling Complexity in the Heart of Software [Evans]. Habanero is at its most fundamental level an implementation of the Active Record pattern as described by Martin Fowler in Patterns of Enterprise Application Architecture [Fowler]. If you haven't read either of these, or don't know anything about domain concepts you will still be able to follow along and pick them up.

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. 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. It 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: Capture a Pizza

Building a Habanero domain model class

Our first step is to create a domain model class that represents a real world concept, and then make some adjustments to it to enable it as a Habanero domain model class.

Bob has a pizza place called Bob's Pizza Place. In his pizza place he sells pizzas with various names, and each pizza has a price. The first user story to implement is to capture new pizzas and their prices. A simple domain model Pizza class for this user story would look like:

```
public class Pizza {
    public virtual String Name { get; set; }
    public virtual Decimal Price { get; set; }
}
```

Create a blank solution (Go to File | New | Project..., then to Other Project Types | Visual Studio Solutions, and select Blank Solution), then create a Class Library called BobsPizzaPlace.BO and create the above Pizza class within this class library.

The Pizza class is a persistence-ignorant domain model class. For performance and simplicity-of-use reasons, Habanero requires you make a few changes to a domain model class like this to get it persistence-enabled (that is, able to save and load from a data store such as a database). For starters you need to map the class to a persistence model. This is done in xml:

```
<classes>
  <class name="Pizza" assembly="BobsPizzaPlace.BO">
    <property name="PizzaID" type="Guid" />
    <property name="Name" />
    <property name="Price" type="Decimal" />
    <primaryKey>
      <prop name="PizzaID" />
    </primaryKey>
  </class>
</classes>
```

Create a file called ClassDefs.xml in your domain model project (BobsPizzaPlace.BO) and place this xml in it.

This xml describes the Pizza class, its properties and their types to the persistence layer of Habanero. Because Habanero is based on the Active Record pattern, we have to make a few modifications to our Pizza class to get it persisting using the business object layer of Habanero. First we add references to Habanero.BO.dll and Habanero.Base.dll (found in the lib folder of the archive this article is in), and then change the Pizza class to look as follows:

```
using Habanero.BO;
public class Pizza : BusinessObject {

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

```

    }

    public virtual decimal? Price
    {
        get { return ((decimal?)(base.GetPropertyValue("Price"))); }
        set { base.SetPropertyValue("Price", value); }
    }
}

```

The first thing to note is that Pizza now inherits from BusinessObject, a class in Habanero.BO. This enables Habanero to load and persist Pizza objects and to perform other domain layer operations with those objects such as property validation and relationship loading. When inheriting from BusinessObject the class receives a collection of property objects. Getting and setting the values of these objects is done via the GetPropertyValue() and SetPropertyValue() methods, so we change the auto-properties of the first example to instead store and retrieve the values of the Pizza's properties from this inherited collection of properties.

At this stage Pizzas are ready to be used as domain objects. Let's play with them a bit.

Creating a Habanero application

Now that we have a domain model class called Pizza, the next step is to use them in a simple console application. We're using a console application here so that we don't have to deal with the complexities of graphical user interfaces just yet.

Create a new console application project in your solution. I called mine BobsPizzaPlace.Console. In it we want to start up a Habanero application, which means configuring the data store and where the class definitions xml is found. For a console Habanero application, we need to add a reference to Habanero.Console, Habanero.Base, Habanero.BO and log4net, all found in the lib folder. We also need to reference our domain model project, BobsPizzaPlace.BO. Once all these references are added we can create a HabaneroApp object. Declare a HabaneroAppConsoleInMemory field on your Program class:

```
private static HabaneroAppConsoleInMemory _habaneroApp;
```

Then we kickstart the app with the following code in the Main(string[] args) method:

```

_habaneroApp = new HabaneroAppConsoleInMemory("Bob's Pizza Place", "0.1");
_habaneroApp.Startup();

System.Console.In.ReadLine();

```

You'll also need to add the following line to your usings:

```
using Habanero.Console;
```

There is one more necessary change. Right-click on your ClassDefs.xml file in the domain model project and go to its properties page. Change the "Copy to Output Directory" property to "Copy always". This puts this file in the program's output folder so that our application can read it. Usually we embed it as a resource but this is simpler for now.

Set BobsPizzaPlace.Console as the startup project, set BobsPizzaPlace.Console.Program as the startup object and run it. You will get an error on the console output screen about log4net because we haven't configured it, but otherwise the program shouldn't give you any other errors. Let's quickly configure log4net to help with logging any errors that might occur in Habanero. To do this

add an Application Configuration File to your console application and put this xml within the configuration tags:

```
<configSections>
  <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,
log4net"/>
</configSections>

<log4net>
  <appender name="FileAppender" type="log4net.Appender.FileAppender">
    <param name="File" value="BobsPizzaPlace-log-file.txt"/>
    <param name="AppendToFile" value="true"/>
    <layout type="log4net.Layout.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] - %m%n"/>
    </layout>
  </appender>

  <root>
    <level value="DEBUG"/>
    <appender-ref ref="FileAppender"/>
  </root>
</log4net>
```

Now running the program should show a blank console screen with no errors. This means the application is successfully configured.

Using the Pizza domain model class

The next step is to create some Pizzas and save them. To do this you simply create a Pizza object and call its Save() method:

```
Pizza margarita = new Pizza {Name = "Margarita", Price = 30};
margarita.Save();
```

You'll need to add the following using to get this to compile:

```
using BobsPizzaPlace.BO;
```

Running your application now will do nothing, however. This is because the pizza you created is being saved to an in memory data store, not to disk. The in memory data store is particularly useful for testing because it behaves almost exactly the same as a database, the only difference being that foreign key violations are not picked up.

Saving the in-memory data store to a file

You can save the data store to an xml file quite easily; add the following code after creating and saving your pizza:

```
WriteDataStoreToFile();
```

And create a method called WriteDataStoreToFile():

```
private static void WriteDataStoreToFile()
{
    using (Stream pizzasStream = File.Create("pizzas.xml"))
    {
        DataStoreInMemoryXmlWriter writer =
            new DataStoreInMemoryXmlWriter(pizzasStream);
        writer.Write(_habaneroApp.DataStoreInMemory);
    }
}
```

This saves the contents of the in memory data store to a file called pizzas.xml in xml format. After running the program the contents of my pizzas.xml file was:

```
<?xml version="1.0" encoding="utf-8"?>
<BusinessObjects>
  <bo __tn="Pizza" __an="BobsPizzaPlace.B0"
    PizzaID="44bf178d-06cf-464a-be04-727fea48aa55"
    Name="Margarita"
    Price="30" />
</BusinessObjects>
```

Yours will be almost identical, only the PizzaID field will contain a different Guid. Note that you can save to a binary file instead using a `DataStoreInMemoryBinaryWriter`, but this format is fixed to the Habanero version you are using.

Now we'd like to load the saved pizzas and perhaps edit them or add to them.

Loading from file into a data store

The first step after program start-up is to load the pizzas we have previously created into the `DataStoreInMemory`, so add the following line after the Habanero app startup:

```
LoadDataStoreFromFile();
```

Now create a method called `LoadDataStoreFromFile()`:

```
private static void LoadDataStoreFromFile()
{
    using (Stream pizzasStream = File.OpenRead("pizzas.xml"))
    {
        DataStoreInMemoryXmlReader reader =
            new DataStoreInMemoryXmlReader(pizzasStream);
        _habaneroApp.DataStoreInMemory.AllObjects = reader.Read();
    }
}
```

The loading is done with a `DataStoreInMemoryXmlReader`, and its `Read()` method returns a dictionary that can be used to give to the `DataStoreInMemory`.

Now the in memory data store has all our domain model objects. This strategy is only useful for applications where the amount of working data is relatively small, but it's also a way to put off moving to a database until later in the development of your application, as most agile practitioners recommend.

To view the Pizzas in the data store we still need to do a load out of the data store. This is done exactly how one would load from a database in Habanero:

```
BusinessObjectCollection<Pizza> pizzas =
    Broker.GetBusinessObjectCollection<Pizza>("", "Name");
```

In this example I'm loading all Pizzas (empty criteria string) and ordering by their name field. Now I will list them on screen:

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

Now delete the line that creates a Margarita pizza and then saves it. You can run the program now and check that the changes you made work. It should display the following on screen:

0. Margarita 30.00

Adding a new pizza

Having just the Margarita available is not the best option for pizza place that isn't in Italy, so next I'd like to add the ability to create a new pizza from the console. First we add a couple more menu options:

```
System.Console.Out.WriteLine("A. Add a new pizza");
System.Console.Out.WriteLine("Q. Quit");
```

Then we extract the whole menu to another method called `DisplayMenu()`:

```
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("Q. Quit");
}
```

Now we add code to perform the “Add a new pizza” menu item:

```
string input = System.Console.ReadLine().ToUpper();
while (input != "Q")
{
    if (input == "A")
    {
        AddNewPizza();
        pizzas.Refresh();
    }
    DisplayMenu(pizzas);
    input = System.Console.ReadLine().ToUpper();
}
```

And the method `AddNewPizza()`:

```
private static void AddNewPizza() {
    Pizza newPizza = new Pizza();
    System.Console.Out.WriteLine("New Pizza");
    System.Console.Out.Write("Name: ");
    newPizza.Name = System.Console.ReadLine();
    System.Console.Out.Write("Price: ");
    newPizza.Price = Convert.ToDecimal(System.Console.ReadLine());
    newPizza.Save();
}
```

The last line of the `AddNewPizza()` method saves the new `Pizza` object into the data store. Note that after the `AddNewPizza()` method call there is a call to `pizzas.Refresh()`. This is so that the collection of pizzas that we originally loaded is refreshed from the data store. An alternate would be to add the `Pizza` to the collection with its `Add()` method, but the `Refresh()` method is simpler because it will take care of placing the new object into the correct sorted position.

Remove the last `Console.ReadLine()` because the menu takes care of that for us now.

My `Main()` method now looks like this:

```
static void Main(string[] args)
{
    _habaneroApp = new HabaneroAppConsoleInMemory("Bob's Pizza Place", "0.1");
    _habaneroApp.Startup();
}
```

```

LoadDataStoreFromFile();

BusinessObjectCollection<Pizza> pizzas =
    Broker.GetBusinessObjectCollection<Pizza>>("", "Name");

DisplayMenu(pizzas);

string input = System.Console.ReadLine().ToUpper();
while (input != "Q")
{
    if (input == "A")
    {
        AddNewPizza();
        pizzas.Refresh();
    }
    DisplayMenu(pizzas);
    input = System.Console.ReadLine().ToUpper();
}

WriteDataStoreToFile();
}

```

Now you can add new Pizzas to your menu. After adding a few pizzas my menu looks like this:

```

0. Habanero 50.00
1. Margarita 30.00
2. Regina 35.00
3. Tikka Chicken 40.00
A. Add a new pizza
Q. Quit

```

And my pizzas.xml (after a bit of manual formatting):

```

<BusinessObjects>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="674cffa0-181f-402a-bfda-ce2e80a10dbe"
    Name="Margarita" Price="30"/>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="90c2b09e-52b5-4612-a09e-4ad28cd09d0e"
    Name="Regina" Price="35"/>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="da48d31b-7a3b-4fa6-beae-ae0daacef164"
    Name="Habanero" Price="50"/>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="b4c555eb-3edd-4a5a-9810-34de2be9d9fa"
    Name="Tikka Chicken" Price="40"/>
</BusinessObjects>

```

Editing a pizza's price

Editing a pizza is even simpler than creating a new one. If the user types a number of a pizza we offer them the ability to enter a new price. The menu code becomes (the new code is highlighted):

```

while (input != "Q")
{
    if (input == "A")
    {
        AddNewPizza();
        pizzas.Refresh();
    } else
    {
        int pizzaNumber = Convert.ToInt32(input);
        EditPizza(pizzas[pizzaNumber]);
    }
}

```

```

        DisplayMenu(pizzas);
        input = System.Console.ReadLine().ToUpper();
    }

```

The new EditPizza() method is:

```

private static void EditPizza(Pizza pizza) {
    System.Console.Out.WriteLine("Updating price of {0} pizza", pizza.Name);
    System.Console.Out.WriteLine("Old price: {0:0.00}", pizza.Price);
    System.Console.Out.Write("New price: ");
    pizza.Price = Convert.ToDecimal(System.Console.ReadLine());
    pizza.Save();
    System.Console.Out.WriteLine("Price updated");
}

```

That's it. Simply set the Price property and call the Save() method on the Pizza.

Deleting a pizza

The final operation I want to add is the option to delete a pizza off the menu. For this I'll add a menu option to the DisplayMenu() method (just before the Quit option):

```

    System.Console.Out.WriteLine("D. Delete a pizza");

```

We also need to add the logic to the menu:

```

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
{
    int pizzaNumber = Convert.ToInt32(input);
    EditPizza(pizzas[pizzaNumber]);
}

```

And the new DeletePizza() method:

```

private static void DeletePizza(Pizza pizza) {
    pizza.MarkForDelete();
    pizza.Save();
}

```

Calling MarkForDelete() marks the pizza for deletion, but doesn't delete it from the data store just yet. Only once Save() is called is the object removed from the data store. After deleting my Habanero pizza and changing the price of a Tikka Chicken pizza to R45.00, my pizzas.xml looks like this:

```

<BusinessObjects>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="674cffa0-181f-402a-bfda-ce2e80a10dbe"
    Name="Margarita" Price="30"/>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="90c2b09e-52b5-4612-a09e-4ad28cd09d0e"
    Name="Regina" Price="35"/>
  <bo __tn="Pizza" __an="BobsPizzaPlace.BO"
    PizzaID="b4c555eb-3edd-4a5a-9810-34de2be9d9fa"
    Name="Tikka Chicken" Price="45"/>
</BusinessObjects>

```


Preventing duplicate pizzas from being saved

I'd like to do one more thing with the Pizza domain model class: I'd like to prevent two pizzas being created with the same name. Using Habanero this is very straightforward to do. In fact, all we need to do is declare the Name field as part of a "Key" of the Pizza class in the xml class definition (ClassDefs.xml):

```
<?xml version="1.0" encoding="utf-8" ?>
<classes>
  <class name="Pizza" assembly="BobsPizzaPlace.B0">
    <property name="PizzaID" type="Guid" />
    <property name="Name" />
    <property name="Price" type="Decimal" />
    <key>
      <prop name="Name" />
    </key>
    <primaryKey>
      <prop name="PizzaID" />
    </primaryKey>
  </class>
</classes>
```

We also need to catch any error that might be thrown when trying to persist now that we have a rule preventing it in the case of a Pizza already existing with the same name, so the saving code in AddNewPizza() becomes:

```
try
{
    newPizza.Save();
}
catch (BusinessObjectException ex)
{
    System.Console.Out.WriteLine(ex.Message);
}
```

Be sure to "Rebuild All" before running again so that your ClassDefs.xml file gets updated in the output folder. Now if you try to add a new Margarita pizza you will get the following message and the new pizza won't be saved:

A 'Pizza' already exists with the same identifier: Name = Margarita.

Conclusion

In this article we created a Pizza domain class and enabled it to be persisted using Habanero.

We then created a Habanero console application from scratch that uses an in-memory data store and can save this data store to a file in xml format.

We then added the ability to create a new pizza, edit a pizza and delete a pizza from the data store.

Finally we added a key to the Pizza domain class so that Habanero checks for duplicate pizzas before saving a new one.

The in-memory data store is a quick way to get a domain model up and running without worrying about database structures. Changing to a database is a simple process when it is required and shouldn't change the behaviour of your application at all. This will be the subject of a future article in this series.

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