# Section 2 – Business Logic Layer:



Business logic layer can usually be divided into two types.

The ***Business Domain layer/***model has to do primarily with the static behaviour and their relationships to each other e.g. Contacts, orders, Order Approval strategies. This layer is the heart of business software. This layer maintains the State of the business e.g. the State of an Order and the rules involved in changing the state of the order. This layer interacts with the Data Access Layer to persist and retrieve these states over application instances.

***Business Process logic*** layer also called the process logic, application logic, Application layer, service layer or workflow logic. This layer is kept very thin its primary responsibility is to delegate responsibility to a collaborating group of Business Objects to achieve a Use Case. The application logic covers things like notifying a senior manager if the order exceeds certain criteria, revenue recognizing all orders delivered for month. The Business Process Logic Layer does not contain business rules but instead is responsible for co-ordinating the Business objects as such this layer does not typically store or retrieve any long lived application state (state that persists between an application instance). The exception to this is where long lived workflow logic is stored i.e. where the state of the progress of a task is stored.

Perhaps this can be clarified by looking at a practical example from our order entry application.

Some typical static rules belonging in the domain model would be: customer can have many orders, an order can have only one customer, delivery date of a new order must be beyond the current date, or stock of an item can never fall below zero.

Examples of rules that fall in the process logic layer could be: to capture a new order, the customer must have sufficient credit and the order item must be linked to a current product. Clearly the latter is more approprate within the process flow of entering a new order and does not apply to the static state of a business object.

*Why separate these layers*: The Domain objects are less reusable across applications and modules if they contain substantial application logic. It is harder to implement application logic using a workflow application if the logic is mixed up with the domain model. By seperating the concerns it is easier to maintain and modify the application.

*When to use a Business Process Logic Layer*: It is sometimes difficult to decide when program logic belongs in the business domain layer and when in the business process layer. These guidelines are general rules of thumb but as you use these layers you will develop a feel for what belongs where. Also if you are using a good automated testing philosophy and refactoring it is relatively easy to move logic between these layers if it is in the incorrect place to start. As we develop examples through this book we will show examples of when we have separated the logic into these layers. Here we present a few brief guidelines on when to use business process objects:

* In our experience 60% - 80% of use cases in enterprise applications are simple CRUD use cases (create, read, update, delete data). In these cases there is no need for a Business Process logic layer. The User interface can simply create read, update and delete business objects.
* More complex use cases that have significant process flow, decision paths etc map naturally to business process logic operations. These interact with the various domain objects to implement the use case.
* In cases where transactions cross multiple objects e.g. capturing a new order results in updating the customer account, the planning system buckets, the available product and all these must be persisted in a single transaction, then it is essential to use a Business Process object.
* Complex logic such as messaging, integration to other applications and batch processes usually require a business process object.
* In cases where there are expected to be multiple interfaces to your application e.g. Web services, Rich client interface, Web interfaces, console interface.
* Functionality that has a strong relationship to a single business object and its composite children belongs in the business objects (in the domain layer).
* Functionality that does not have a strong relationship to a single business object belongs more naturally in the business process logic layer.

Discusss:- Rich domain model advantages and why even use on small projects. How benefit as project complexity grows and as project enhanced and modified. How works with agile methods of development. How domain model evolves as understanding between domain experts, analysts and developers evolves.

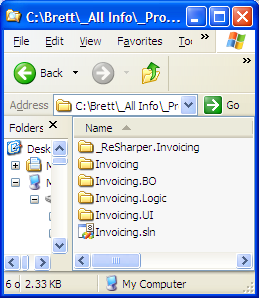
To follow the examples in this chapter you should have downloaded the Habanero Framework, Firestarter and the Examples (See Appendix A Setting up the Framework)

## Chapter 6: Implementing this domain model using the Habanero Framework.

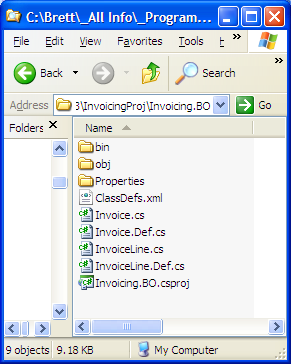
### Generating the Business Objects

Once the Model is captured into fire starter we can generate Code (Ctrl G). This will generate a number of folders for the various application layers, a solution file, project files and some C Sharp files (.cs)

For the Business Object layer this will generate an XML file (ClassDef.xml) representing the Class Definitions captured in Firestarter and the Code to use interpret and use this definition file. The Application developer does not need to concern himself



At the moment we are interested in the BO layer which a project file a ClassDef.xml file and 4 .cs files.



The two .Def.cs files contain the automatically generated code using partial classes and these should not be edited since they will be regenerated any time that the project is regenerated.

A brief look at these two files will show the code generated based on the Model captured in FireStarter.

Invoice.Def.cs has the relationship InvoiceLines which returns a strongly typed collection of invoice lines.

public virtual BusinessObjectCollection<InvoiceLine> InvoiceLines

{

get

{

return Relationships.GetRelatedCollection<InvoiceLine>("InvoiceLines");

}

}

And the InvoiceLines.Def.cs has the properties defined in FireStarter

public virtual Guid? InvoiceLineID

{

get

{

return ((Guid?)(base.GetPropertyValue("InvoiceLineID")));

}

}

public virtual String InvoiceLineNumber

{

get

{

return ((String)(base.GetPropertyValue("InvoiceLineNumber")));

}

set

{

base.SetPropertyValue("InvoiceLineNumber", value);

}

}

public virtual Decimal? InvoiceLineValue

{

get

{

return ((Decimal?)(base.GetPropertyValue("InvoiceLineValue")));

}

set

{

base.SetPropertyValue("InvoiceLineValue", value);

}

}

As well as the single relationship Invoice which returns the Invoice related to the Invoice Line.

public virtual Invoice Invoice

{

get

{

return Relationships.GetRelatedObject<Invoice>("Invoice");

}

set

{

Relationships.SetRelatedObject("Invoice", value);

}

}

From the simplicity of this code it is obvious that that the Validity and Read Write rules captured in Firestarter are not represented here. This is because these are all represented using the ClassDef.xml File. This file along with the relevant classes implements the MetaData mapping pattern (See Fowler – 306).

The Application Developer should not be required to view or edit the Definition file but it is important to understand conceptually how the file works to implement the Business rules of the domain model.

<classes>

<class name="Invoice" assembly="Invoicing.BO">

<property name="InvoiceID" type="Guid" readWriteRule="WriteNew" compulsory="true" />

<primaryKey>

<prop name="InvoiceID" />

</primaryKey>

<relationship name="InvoiceLines" type="multiple" relatedClass="InvoiceLine" relatedAssembly="Invoicing.BO" deleteAction="DeleteRelated" orderBy="InvoiceLineNumber">

<relatedProperty property="InvoiceID" relatedProperty="InvoiceID" />

</relationship>

</class>

<class name="InvoiceLine" assembly="Invoicing.BO">

<property name="InvoiceLineValue" type="Decimal" readWriteRule="WriteNew" compulsory="true" description="The monetary value of this invoice line (determined by the product price multiplied the mass of all the cases invoiced)">

<rule name="MinInvoiceLineValue">

<add key="min" value="0" />

</rule>

</property>

<property name="InvoiceLineID" type="Guid" readWriteRule="WriteNew" compulsory="true" />

<property name="InvoiceID" type="Guid" readWriteRule="WriteNew" compulsory="true" />

<property name="InvoiceLineNumber" compulsory="true" />

<key name="AK1" message="An invoice cannot have two invoice lines with the same invoice line number.">

<prop name="InvoiceID" />

<prop name="InvoiceLineNumber" />

</key>

<primaryKey>

<prop name="InvoiceLineID" />

</primaryKey>

<relationship name="Invoice" type="single" relatedClass="Invoice" relatedAssembly="Invoicing.BO" deleteAction="DoNothing">

<relatedProperty property="InvoiceID" relatedProperty="InvoiceID" />

</relationship>

</class>

</classes>

### Using the Business Objects

If you look at the Invoice and the InvoiceLine you will notice that both Classes inherit from Business Object. From this you can conclude that much of the interesting behaviour associated with the Invoice and Invoice lines is inherited from BusinessObject.

From the ClassDef.xml you can see that all the Properties, Relationships and Rules defined in FireStarter are generated to this ClassDef.xml File.

To implement this Habanero makes extensive use of the Layer SuperType Pattern (Fowler -xxx )

#### Setting a Business Object Property

When a business object property is set by the user of the business object e.g. using code such as invLine.InvoiceLineValue = 12.00m; We will demonstrate that the functionality and use of the business object via tests that show the changes that occur to a business object when the objects state has changed.

[Test]

public void TestUpdateInvoiceLineProperty()

{

//---------------Set up test pack-------------------

InvoiceLine invLine = new InvoiceLine();

//---------------Assert Precondition----------------

Assert.IsTrue(invLine.State.IsNew);

Assert.IsFalse(invLine.State.IsValid());

Assert.IsFalse(invLine.State.IsDeleted);

Assert.IsFalse(invLine.State.IsDirty);

Assert.IsFalse(invLine.State.IsEditing);

Assert.IsTrue(invLine.State.IsValidMessage.Contains("'Invoice Line Value' is a compulsory field and has no value"));

string inValidReason;

Assert.IsFalse(invLine.IsValid(out inValidReason));

Assert.IsTrue(inValidReason.Contains("'Invoice Line Value' is a compulsory field and has no value"));

Assert.IsNull(invLine.InvoiceLineValue);

IBOProp prop = invLine.Props["InvoiceLineValue"];

Assert.IsFalse(prop.IsDirty);

Assert.IsTrue(prop.IsObjectNew);

Assert.IsFalse(prop.IsValid);

//---------------Execute Test ----------------------

invLine.InvoiceLineValue = 12.00m;

//---------------Test Result -----------------------

Assert.IsTrue(invLine.State.IsNew);

Assert.IsFalse(invLine.State.IsValid());

Assert.IsFalse(invLine.State.IsDeleted);

Assert.IsTrue(invLine.State.IsDirty);

Assert.IsTrue(invLine.State.IsEditing);

Assert.IsFalse(invLine.State.IsValidMessage.Contains("'Invoice Line Value' is a compulsory field and has no value"));

Assert.AreEqual(12m, invLine.InvoiceLineValue);

Assert.IsTrue(prop.IsDirty);

Assert.IsTrue(prop.IsObjectNew);

Assert.IsTrue(prop.IsValid);

}

If all you intend to do is utilise the Habanero Framework and Firestarter then this is all you need to know about the Framework and you can stop here. You can return to the remainder of this chapter at a later stage.

List the main namespaces and classes of interest.

Generic types and the fact not polymorphic how overcome this with interface i.e. IBusinessObjectCollection.

Tracking objects status valid dirty, new, markedForDeletion, etc.

IsNew : When object is creates marked as new.

Load from Database: IsNew = False.

It is not assumed that all data in the database is valid the reason for this is that business rules can be modified over time and the recently loaded business object may now no longer pass the validation rules.

Property level dirty tracking such that only the properties that are dirty are updated to the datastore. This reduces the probability of concurrency control failures.

Deleting objects from database deferred or immediate. The deferred approach is used here i.e. the object is marked as deleted when the UI developer calls save the object is physically deleted from the database.

Custom rules historical data

Validation – Any custom or standard rules are verified and marked as broken when the property value is changed however an exception is only thrown if the UI developer tries to save an object that is in an invalid state.

The framework creates the BusinssObjectInvalidStateException which is thrown. The UI developer can therefore easily catch this exception when required.

Authentication and Authorisation: Authentication is the process of logging a user on. Authorisation is the process of verifying that a user may use certain features e.g. Edit An object View a particular property or object..

Non Duplication of code so security is implemented for business object and its properties.

This can then be used by the UI Developer to disable certain buttons on the UI as required (typical in windows development) or to raise appropriate error if the user is not authorized (typical in ASP web development). Of course the UI developer could implement more sophisticated security if required. This feature is implemented using the strategy pattern so that the Application developer can implement any security strategy required. The application developer could therefore implement a strategy using any custom security or using the security provided in Chillisoft security. The implementation as a strategy means that if no security strategy is implemented then there is no performance cost in having security as part of the framework.

GetHashCode, ToString, Equals should be implemented by every business object.

Object ID maybe should be virtual check

Lhotka: businessObjectBase is generic page 141 why?

System.Security.Principal.IPricipal – this is the standard security interface in .Net.

This exposes the IsInRole method.

Tracking state IsDirty if the object or any of its children are dirty.

IsValid if the object and all of its children are valid. The object is valid if all of its properties are valid.

Details of lookup lists. Simple, Business Object??

public class **BindingList<T>** : **System.Collections.ObjectModel.Collection<T>**

Member of **System.ComponentModel**

**Summary:**

Provides a generic collection that supports data binding.

Very practical view of the main issues facing developers developing systems in business environment. We do not delve into the theory unless it is necessary to understand the issues. There are many books covering the theoretical aspects of most of these issues we have however found a dirth of books that help a real life developer implement solutions for these issues in real life.

# 

# 

## Security

To use Windows Authentication call the following Code in the Application Startup.

AppDomain currentDomain = AppDomain.CurrentDomain;

currentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);

Using Windows Authentication is ideal where this is possible this prevents the user from having to log onto the application each time he/she uses it. In our experience it is however often not applicable for any number of reasons. Our guideline is where you can use Windows Authentication use it where you cannot use it the Downloadable Habanero.Security Application can be used or can give you an idea of how to develop an application that uses custom Authentication.

## Automating the process. Code Templates, Code Generators and MetaProgramming

Anyone who has tried to create a project using business objects will have experienced the extent of mind numbingly boring repetitive code that is created to get data from the database update the database, check validation rules etc. Half way through even the smallest project one will realize that there has to be a better way Here we will discuss 3 techniques all these techniques are important techniques for creating robust business applications quickly and all have advantages and disadvantages. These will be discussed below.

Code Templates are the simplest and are typically used to give the basics outline of a project, class or method. These are very useful and should be used when applicable. The problem with templates is that they lack the ability to implement real complexity e.g. a template for a business object does not have the sophistication to allow the user to generate fields, properties or relationships. Templates on the positive side are very easy to set up share and reuse.

Code generators are used to rapidly produce the basics of usable code that can then be edited and modified by the developer. Generators offer many of the advantages of templates but are more powerful you could model the properties and relationships of a business object in the generator and have the generator generate the code for you. This is obviously a very powerful technique for rapidly producing a large amount of repetitive potentially boring code. The generated code can be edited and modified to meet your special need providing high levels of extendibility. Generators are significantly more complex to create since all the individual templates that are put together to for the class(es) have to be created, the generator has to be able to capture the properties and relationships of classes. The templates in the generator are difficult to test in any automated manner since the templates are effectively stored as code snippets. The only way to effectively test is to have a standard project. Generate from this standard project and test the results. Generators by their very nature result in massive amounts of duplicate code since the template methods are in effect duplicated. Generated code can be time consuming to maintain since it is difficult to remove an item that is already generated to code. It is difficult to fix a bug that is found in the templates since the generated code is spread into many classes and many projects. It is difficult to upgrade an old project to a new version of a framework since the generated code has been generated from templates that assume a certain framework. Changes to rules such as string length rules for a property are made directly in the code. The numbers of templates required for generation quickly becomes large and the maintenance of these becomes a problem in its own right. In short excessive use of generated code breaks the DRY (Don’t repeat Yourself) principle.

Meta data programming. Meta data programming is the practice of storing the metadata of a program in a computer readable form e.g. XML. The class(es) are modeled in a ‘generator’ in much the same way as with a generator but the ‘generator’ is primarily responsible for maintaining the MetaData (e.g. the XML). The metadata is then used at run time to implement the program rules as modeled using the metadata. There is preferably little or no generation and generation should be limited to wrapping calls to the framework in helper methods. Meta data programming overcomes many of the problems with templates and generators as follows. The framework can be fully tested automatically. Changes and enhancements to the framework are easily managed and existing applications can easily be moved to the new framework. There is no duplication of code. Changing rules adding new properties removing properties and changing relationships between classes all becomes simple. There are some disadvantages to Meta data programming namely the framework is more complex and the design of the framework has to be designed to be highly flexible and extendible to allow for the development of custom behaviors. Over the years of programming frameworks we have steadily moved from code generators to meta data programming. This evolved over time as we experienced maintenance and testing problems with generated code. Our experience with Meta Data programming over the last 6 years has lead us to believe that is a powerful practice that results in a more maintainable testable applications. As our practices have steadily moved to incorporate more agile methodologies we have found that the meta data programming framework model provides significantly better support for incremental design and development whereas code generation is perfectly acceptable in a design up front environment. The recent emergence of WPF (XAML) etc are showing that we are not the only people who have come to this conclusion over the years.

For the Habanero framework we follow a Meta data programming model but use FireStarter (a custom code generator) to enable the management of the metadata as well as to generate the helper methods (e.g. the properties and relationship calls) when required. Firestarter has been designed to facilitate Agile Model Driven Development. We recongnise that most programming is a mental activity. We also recognize that developers have periods of intense creativity where they are in the ‘Zone’. The objective of Firestarter and Habanero is to provide a framework and Modeling tool that keeps the programmer in the zone by automating boring repetitive time-consuming error prone tasks. Habanero and Firestarter have evolved through years of use in actual software development project from small projects 1 programmer – 2 month to large 20 programmers for 4 year projects. Firestarter allows you slowly grow the application add rules classes and relationships as you move on. By having a tool that generates these easily you reduce the effort in creating a model. This frees the developer up to truly allow the solution to evolve. On the other hand when the appropriate methodology is up front design and generating the entire solution FireStarter handles this as well.

## Advanced Topics

Directg access to configuration-time metadatamodel. Via a set of classes. This allows you to define new properties and rules at run time. (Ala asset management). This has proven critical for user defined classes. E.g. Asset management, Recipe management etc.