# Chapter 3: The Domain Model – Business Objects

Making a start

“There is no abstract art. You must always start with something. Afterward you can remove all traces of reality.”…. Pablo Picasso (Spanish Cubist painter (1881 - 1973)

The heart of a well designed enterprise application is a good Business Domain model. This is why we start here.

The mapping of the rich domain model to the data store, the implementation of complex use cases using the process logic layer and the development of the user interface layer will be discussed later in relation to the domain model. Habanero provides the framework for developing, maintaining and refactoring a rich domain model. Firestarter provides a tool for rapidly implementing and managing the domain model in code.

A good understanding of business objects is a pre-requisite to understanding domain models and domain driven design (which essentially is the modeling of business or domain objects and their relationships).

### The Business Objects

In a well designed object oriented system, there are many objects that collaborate to implement any functionality. So what differentiates a business object from other objects in the system?

A business object:

* Is an abstract representation of an entity or concept that exists in the business problem domain, e.g. An invoicing system will contain concepts such as Invoice, Invoice Line
* Has a state that exists beyond the life cycle of a single instance of the application i.e. a business object can be persisted or retrieved from a data store.
* Implements business rules related to the validation, editing and persisting of the business objects data.
* Doesn’t know how it will be used i.e. it can be used by a Windows Form, ASP or a web service.
* Encapsulates its data and implements business rules so that it protects the validity of its state independently of the client using it, i.e. if the business rule is that the Invoicing Date must be less than today, the Business Object will enforce this rule.
* Encapsulates any security rules relating to it, e.g. only a user with profile ‘Invoicing User’ can create a new invoice.
* Has behavior i.e. the business object implements functionality expected of the entity in the domain e.g. an invoice might have a method to Calculate Days Overdue for payment, Amount due for payment.
* Has an identity. The identity of a business object is an important issue because it provides the ability to persist and retrieve the business object from a data store. Ideally the Identity should be globally unique and immutable.

#### Example of a Business Object

Let’s take a look at how this is implemented in Habanero. We will start with a business object called Customer, and for now the only property that customer will have is customer name. Customer name is compulsory and must have a length greater than 5 characters and less than 100. For the purposes of explaining the use and functionality of the Customer Class we will be using Unit tests developed using the NUnit Framework. The Full Code for the Customer class and the Customer Test class used are available from the book downloads Chapter 3\Example1\ Customer.sln.

The FireStarter Project set up in Chapter 3\Example1\Customer.fsproj.

All the Customer object code is generated using Firestarter and is thus generated into the Customer.Def Partial Class.

In the Customer.Def.cs you will see the following implementation. The customer inherits from the Business Object Class (following the Layer Supertype Pattern – Fowler - xxx).

public partial class Customer : BusinessObject

{

#region Properties

public virtual String CustomerName

{

get

{

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

}

set

{

base.SetPropertyValue("CustomerName", value);

}

}

public virtual Guid? CustomerID

{

get

{

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

}

set

{

base.SetPropertyValue("CustomerID", value);

}

}

#endregion

}

As you can see this class contains two properties: the CustomerName and the object’s ID property CustomerID.

In addition to the Customer Class we have a ClassDefs.xml file. Although the application developer does not need to understand the details of this file, it is important to understand that it exists and what it does. The ClassDefs.xml assists Habanero in implementing the Meta-Data Mapping Pattern (See Fowler p. 306) and Meta Programming paradigm.

In the ClassDefs.xml file you can see the class Customer is defined with its two properties CustomerName and CustomerID. CustomerID is defined as the PrimaryKey (Object ID).

<classes>

<class name="Customer" assembly="Customer.BO">

<property name="CustomerName" />

<property name="CustomerID" type="Guid" readWriteRule="WriteNew" displayName="CustomerID" />

<primaryKey>

<prop name="CustomerID" />

</primaryKey>

</class>

</classes>

#### Using the Customer Business Object

From here we are going to show the basics of how to use a Customer Business Object, and the functionality and capability that it provides to the application developer. Most of this functionality will be demonstrated using the appropriate unit tests in TestCreateCustomer.

##### Creating a new Customer

[Test]

public void TestCreateNewCustomer\_NoPropertyRules()

{

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

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

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

Customer customer = new Customer();

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

Assert.IsTrue(customer.Status.IsNew);

Assert.IsNotNull(customer.CustomerID);

Assert.IsFalse(customer.Status.IsDeleted);

Assert.IsFalse(customer.Status.IsDirty);

Assert.IsFalse(customer.Status.IsEditing);

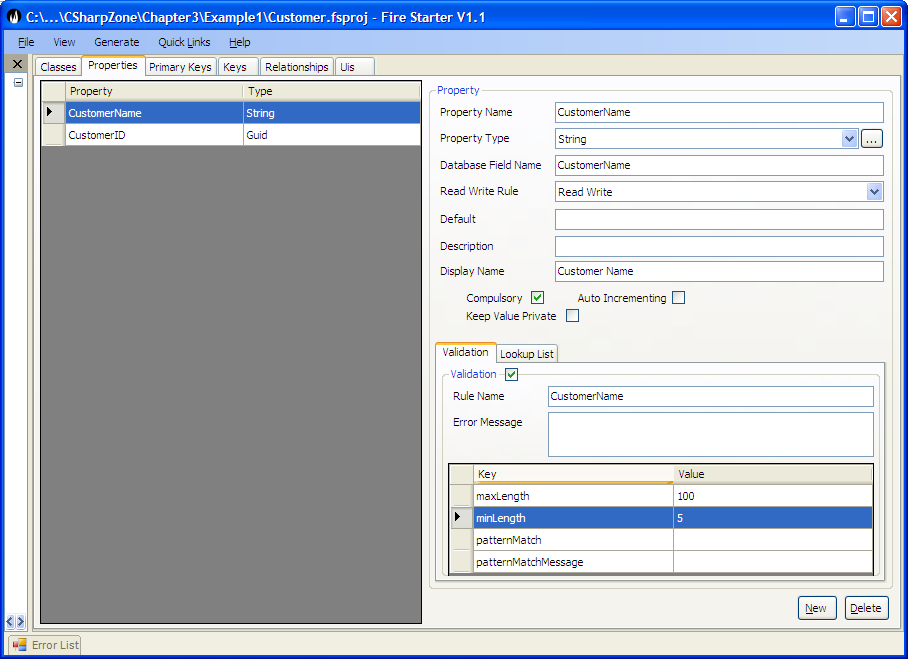
Assert.IsTrue(customer.Status.IsValid());

}

From the above you can see that a new Business Object Customer is created with a status IsNew. Habanero has also already created an object ID for the Customer. The business object has not yet been edited and it’s IsDirty and IsEditing Status is therefore false. No rules have yet been set up for the Customer Business Object – its IsValid property is therefore true.

##### Creating a new Customer with CustomerName Compulsory

The Business rule that the customer name is compulsory is set in FireStarter along with the minimum (5 characters) and maximum length (100 characters) of the CustomerName.



This does not in any way change the Customer.Def.cs but instead changes the metadata rules set up in the ClassDef.xml.

<property name="CustomerName" compulsory="true">

<rule name="CustomerName">

<add key="maxLength" value="100" />

<add key="minLength" value="5" />

</rule>

</property>

The Customer name property definition is modified to be:

The Habanero Framework implements the Customer Compulsory rule as demonstrated in the following tests.

[Test]

public void TestCreateNewCustomer\_CustomerNameCompusory()

{

//--------------------------------------------------

//When a new customer is created it is created with

// broken rules for any

// compulsory properties that do not have a default value set

// (In this case Customer name).

//--------------------------------------------------

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

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

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

Customer customer = new Customer();

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

Assert.IsTrue(customer.Status.IsNew);

Assert.IsNotNull(customer.CustomerID);

Assert.IsFalse(customer.Status.IsDeleted);

Assert.IsFalse(customer.Status.IsDirty);

Assert.IsFalse(customer.Status.IsEditing);

Assert.IsFalse(customer.Status.IsValid());

StringAssert.Contains("'Customer Name' is a compulsory field and has no value",

customer.Status.IsValidMessage);

}

##### Set Property Value

The Habanero Framework implements the CustomerNameRule as demonstrated in the following tests.

[Test]

public void TestNewCustomer\_SetCustomerName\_ToValidValue()

{

//When a property is set to a valid value for a compulsory field

// that has a broken rule the rule is set to no longer broken.

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

Customer customer = new Customer();

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

Assert.IsTrue(customer.Status.IsNew);

Assert.IsFalse(customer.Status.IsDirty);

Assert.IsFalse(customer.Status.IsEditing);

StringAssert.Contains("'Customer Name'

is a compulsory field and has no value",

customer.Status.IsValidMessage);

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

customer.CustomerName = "Valid Name";

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

Assert.IsTrue(customer.Status.IsDirty);

Assert.IsTrue(customer.Status.IsEditing);

Assert.AreEqual("", customer.Status.IsValidMessage);

}

The Test class contains some additional tests that demonstrate other variants of setting and getting property values, and examples of how setting property values changes the Customer Status based on the property rules.

##### Saving Business Objects

The implementation details of saving and loading Business objects from a DataStore are discussed in detail in section 4. For the purposes of this chapter we are just going to look at some methods to save a single object to a DataStore. The tests used in these examples are all from the TestSaveCustomer. It is clearly evident from this test that a saved business object is no longer dirty and is no longer new. The test Test\_Save\_Invalid\_ValidCustomer illustrates that a business object in an invalid state cannot be persisted and the error BusObjectInAnInvalidStateException will be thrown if the application tries.

[Test]

public void Test\_SaveValidCustomer()

{

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

Customer customer = new Customer();

customer.CustomerName = "Valid Name";

customer.CustomerCode = "Code";

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

Assert.IsTrue(customer.Status.IsNew);

Assert.IsTrue(customer.Status.IsDirty);

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

customer.Save();

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

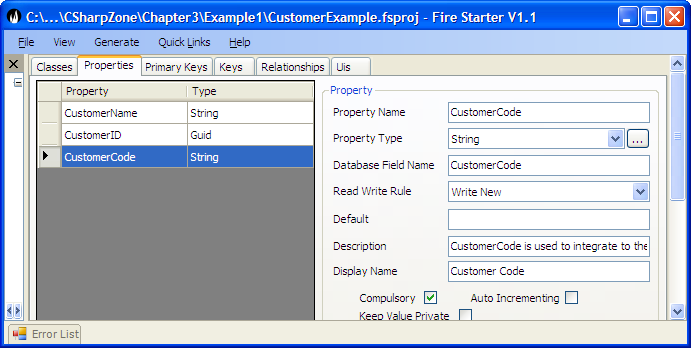
Assert.IsFalse(customer.Status.IsNew);

Assert.IsFalse(customer.Status.IsDirty);

}

Test\_Save\_Invalid\_ValidCustomer – do test in code

##### Using Property Read Write Rules

One of the other rules that is often implemented for a Business Object Property relates to the reading and writing of a property depending on the status of the business object. For instance, a customer has a property CustomerCode whose value is used to integrate to the External systems. This property is compulsory and can never be changed once it has been set. We add this property to FireStarter with the ReadWrite Rule – WriteNew.

The other available read-write options are:

* Read Only – used when a properties value is loaded from a data store but is never updated to the business object e.g. if a stored procedure or external application updates this property’s value and it is only ever read from the datastore by the business object and can never be set, updated or persisted back.
* Read Write – The property’s value can be read and updated many times in the life of the Business Object. This is the default setting.
* Write New – The property’s value can only be set when the object is new (i.e. the business object has never previously been updated to a data store).
* Write Not New – A value can never be set if the business object is new. The value can be set at any time that the business object is not new.
* Write Once – A value for a property can only be set to a business object once. This may be set at any stage of the business object’s life but once the value for the property is set it can never be changed.

There are a few tests in the TestSetCustomerBOProp class relating ReadWriteRules but the most pertinent test is illustrated below. The test shows that the BusinessObjectReadWriteRuleException error is raised if the application attempts to update the Business Object property in contravention of its ReadWrite Rules. The details of how this is implemented are shown in Implementing Business Objects below

[Test]

public void Test\_SetCustomerCode\_ForPersistedCustomer()

{

//A WriteNew property can not be written to when the business object already persisted.

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

Customer customer = CreateSavedCustomer();

IBOProp customerCodeBoProp = customer.Props["CustomerCode"];

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

Assert.AreEqual(PropReadWriteRule.WriteNew, customerCodeBoProp.PropDef.ReadWriteRule);

Assert.IsFalse(customer.Status.IsNew);

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

try

{

customer.CustomerCode = "Code New";

Assert.Fail("expected Err");

}

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

catch (BusinessObjectReadWriteRuleException ex)

{

StringAssert.Contains("Error writing to property 'Customer Code' because it is configured as a 'WriteNew' property", ex.Message);

}

}

Customer Lookup Rule

TODO: Needs section of using a lookup list for a property.

##### Deleting a Business Object

Deleting a Business Object is extremely simple. First the object is marked for deletion using customer.Delete(); and then the deletion is persisted to the DataStore using customer.Save(); The only thing to remember is that when a customer is set as Deleted this has not yet been persisted to the DataStore – only when the customer.Save() is called is the object actually deleted. This is implemented so as to allow objects marked for deletion to be added to a transaction along with other objects. (See Section 4: Data Access Layer -Transaction Committer).

[Test]

public void Test\_DeleteCustomer()

{

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

Customer customer = CreateSavedCustomer();

customer.Delete();

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

Assert.IsFalse(customer.Status.IsNew);

Assert.IsTrue(customer.Status.IsDeleted);

Assert.IsTrue(customer.Status.IsDirty);

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

customer.Save();

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

Assert.IsTrue(customer.Status.IsNew);

Assert.IsTrue(customer.Status.IsDeleted);

Assert.IsFalse(customer.Status.IsDirty);

}

##### Business Object Identity

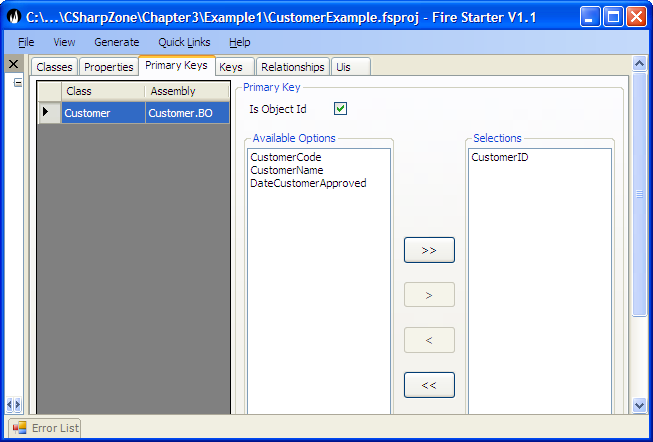
Business object identity primarily impacts the Data Access layer. However it should be noted that it has important consequences in the Business Object Layer, hence the reason why it is briefly discussed here (i.e. a more detailed discussion can be found in Section 4 – Data Access Layer). The managing and tracking of the Identity of an object is essential for any persistable object (i.e. any object that persists its state beyond the life of a single instance of an application), because the object identity is used to retrieve the object from the external datastore. It is thus important to implement a strategy for managing object identity since all business objects are persistable.

Principles of Object Identity:

1. Ideally an object identity should be immutable. An object identity that can be modified creates a set of problems for managing an application, particularly concurrency issues. For example, User1 and User2 load the same object from the dataStore. User1 updates the object’s identity and persists it to the database. Then User2 edits some other properties on the object and updates it. The original object cannot be found in the data store because its identity has been changed.
2. The object identity should be a globally unique identifier. This ensures that business objects can be found and distributed while only being identified by their identities, independently of the application, class or type of object.
3. Object identity should have no meaning. The object should be identified and stored, based on a value that has no meaning in the business domain. Properties such as ProductCode, should not serve as the identifier of an object even if they are immutable. The reason is simply that at some point the ‘immutable’ value is going to be changed. Typically, a user is going to type in the incorrect code and this is going to have to be edited or the organisation is going to modify the algorithm for generating ProductCode and when this happens the ProductCode will be changed.
4. From the above we can conclude that the Business Object’s identity should not be a composite of more than one value e.g. Surname, Firstname should not be used to provide a composite identity.

From the above, we can conclude that in the ideal case where we have control or influence over the database design we can implement these principles. Unfortunately we have found that there are many cases when these principles cannot be adhered to. A typical example is when rewriting an existing system, where the system already has a database that cannot be changed. Consequently, the Habanero Framework provides strategies whereby you can circumvent any of these object identity principles when developing an application. It is however important to point out that if you choose to use a mutable property for the object identity you should implement an appropriate concurrency control strategy for dealing with this. See Section 4 – Data Access Layer – Concurrency Control.

The Object Identity is modelled in FireStarter. The Object identity is always composed of one or more properties of the Business Object. Ideally you will be able to create a property (and field in the database) specifically for this purpose – in this example CustomerID property.



##### Loading a Business Object from the DataStore

In cases where the application knows the ID, the business object is easily retrieved from the data store. All object retrieval is carried out using the BusinessObjectLoader (BORegistry.DataAccessor.BusinessObjectLoader. The details that are involved in Loading Business Objects from a database are discussed in Detail in Section 4 – Data Access Layer.

[Test]

public void Test\_LoadCustomerUsingID()

{

///This test shows that if a persisted object is loaded from the

/// dataStore using the BusinessObjectLoader.GetBusinessObject.

/// Then an object with the exact same status and data as

/// the persisted object is loaded.

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

Customer customer = CreateSavedCustomer();

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

Assert.IsFalse(customer.Status.IsDirty);

Assert.IsFalse(customer.Status.IsNew);

Assert.IsTrue(customer.Status.IsValid());

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

Customer customer2 = GetBusinessObjectLoader().GetBusinessObject<Customer>(customer.ID);

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

Assert.IsFalse(customer2.Status.IsDirty);

Assert.IsFalse(customer2.Status.IsNew);

Assert.IsTrue(customer2.Status.IsValid());

Assert.AreEqual(customer2.CustomerCode, customer.CustomerCode);

Assert.AreEqual(customer2.CustomerName, customer.CustomerName);

Assert.AreEqual(customer2.CustomerID, customer.CustomerID);

Assert.AreSame(customer, customer2);

}

It is important to note that in cases such as this where a reference to the original customer is still being held by the application the second customer returned by the BusinessObjectLoader (i.e. customer2) is in fact the exact same object as the original customer i.e. Assert.AreSame(customer, customer2). The implementation of this will be discussed in Implementing Business Objects Loading below.

You can also retrieve a business object from the BusinessObjectLoader using a criteria (e.g. CustomerCode = ‘HO13787’). This is frequently used when loading a collection of business objects but can also be used when loading a particular business object. Once again, the test shows that e*xactly* the same instance of the customer is returned by the BusinessObjectLoader.

[Test]

public void Test\_LoadCustomerUsingStringCriteria()

{

///This test shows that if a persisted object is loaded from the

/// dataStore using the BusinessObjectLoader.GetBusinessObject.

/// Then an object with the exact same status and data as

/// the persisted object is loaded.

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

Customer customer = CreateSavedCustomer();

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

Assert.IsFalse(customer.Status.IsNew);

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

string loadCriteria = "CustomerCode = " + customer.CustomerCode;

Customer customer2 =   
GetBusinessObjectLoader().GetBusinessObject<Customer>(loadCriteria);

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

Assert.IsFalse(customer2.Status.IsNew);

Assert.AreSame(customer, customer2);

}

The Criteria objects and criteria strings are significantly more powerful than shown so far and includes all the normal operators that are required i.e. AND, OR, opening and closing brackets, <>, =, <>, >=, <=, >, <, IS NOT, IS, NOT LIKE, LIKE.

Note:

* If you try loading an individual object (GetBusinessObject) with a criteria and more than one matching object is found in the database an error will be raised.
* The CustomerCode is the Business Object’s property name and not the database field name. This is very important because by working only with the domain model the application developer is totally insulated from the database. The Habanero Framework interprets the property name into the appropriate field name when querying the datastore.

Once the object is loaded it can be refreshed multiple times from the database. An object that is currently being edited (customer2.Status.IsDirty == true) may not be refreshed but a business object that is not dirty can be refreshed. This is very useful when you are about to carry out a calculation and want to ensure that you have the latest version of the object (i.e. Another user may have edited the object since you loaded it). For more details on concurrency control see Section 4 – Data Access Layer – Concurrency Control.

The object is refreshed using the GetBusinessObjectLoader().Refresh(customer) method.

##### Cancelling Edits on an object

If the application or user has made an error then edits (changes to the object not yet persisted to the datastore) can be cancelled using the Restore Method on the business object.

For restoring a new business object see Test\_RestoreNewCustomer.

The more interesting test however is retrieving an object from the DataStore, editing it and then restoring it as shown in Test\_RestoreLoadedCustomer

[Test]

public void Test\_RestoreLoadedCustomer()

{

//Tests Restoring a loaded business object.

// This test shows that when the customer's Restore

// method is called. The customers Status and Data is

// restored so as to be the same as an object just loaded

// from the DataStore.

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

Customer customer = CreateSavedCustomer();

string origCustomerName = customer.CustomerName;

customer.CustomerName = "New customer Name";

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

Assert.IsTrue(customer.Status.IsDirty);

Assert.AreNotEqual(origCustomerName, customer.CustomerName);

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

customer.Restore();

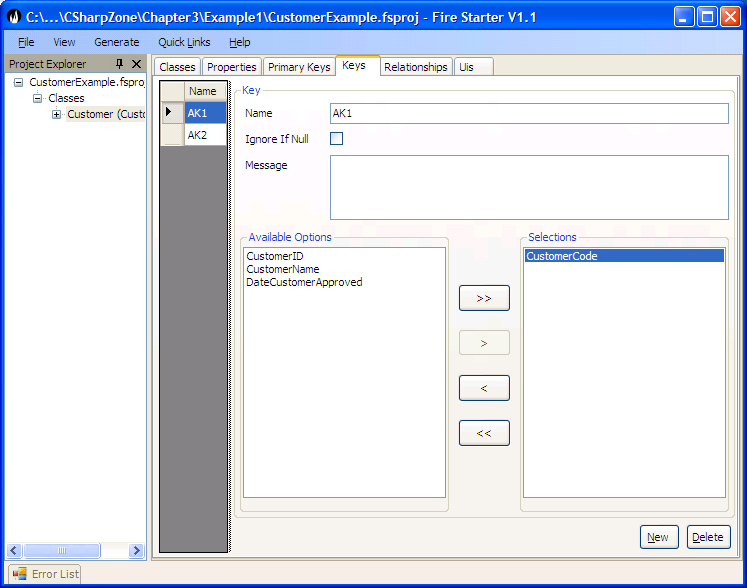
//---------------Test Result -----------------------

Assert.IsFalse(customer.Status.IsDirty);

Assert.AreEqual(origCustomerName, customer.CustomerName);

}

##### Alternate Key for a Business Object

It is possible for a business object to have one or more alternate keys. An alternate key could be composite or simple. The alternate key identifies one or more properties which together acts as a rule preventing a duplicate business object from being persisted. A classical example of an alternate key for the customer is where the customer code must be unique (i.e. two customers with the same code cannot exist in the data store).

TODO: Reference test example code

#### Implementing Business Objects

Do something to explain this is advanced knowledge not required by most application developers.

When an application using Habanero is started, it loads the appropriate ClassDefs.xml (we will go into detail of where and when this happens in Section 5 – UI Layer). The ClassDefs are loaded into the class ClassDef which contains a static collection of the class definitions for each Business Object defined in the ClassDefs.xml.

##### Creating a new Business Object

When a business object of type Customer is created the business object is constructed with its BOStatus appropriately set as new, not deleted and not dirty.

The business object properties and any related rules are retrieved for the Customer business object from the ClassDef class. From these class definitions a collection (BOPropCol) of Business Object properties (BOProp) is created for the customer. Each business object property tracks the current value for the Business object property, whether the property is dirty, valid etc. The BOProp also contains a reference to the definition for the Property i.e. The PropDef.

Figure 3.1: The relationship between a Business object instance and its definitions.

The above class diagram represents a pattern that is used throughout the framework and therefore warrants some discussion and explanation. The ClassDef and its related PropDefs are loaded from the ClassDefs.xml. The ClassDef and PropDef classes contain the definitions of a class and its properties for a particular BusinessObject type e.g. Customer. The definition classes ClassDef and PropDef contain the definitions that are common to all instances of the Customer. The BOProp contains only the data that is unique for a particular instance of the Business Object i.e. CustomerX (example with CustomerName, the PropDef defines that its type is a string and defines the PropertyRules associated with this property, while the BOProp for CustomerName will store the actual value for that instance of Customer).

##### Set Property Value

Let’s demonstrate this with some tests by referring to the TestSetCustomerBOProp class. Note that the Application Developer would never user BOProp directly (The previous examples of setting and getting properties should always be used).

These tests are shown here purely to demonstrate how the framework implements Business Objects and their properties.

[Test]

public void Test\_SetCustomerName\_ToInvalidValidValue\_FromValidValue()

{

//When a property is set to an In Valid Value for a compulsory field

// that has no value the broken rule is changed from compulsory to

// Invalid Value.

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

Customer customer = new Customer();

IBOProp customerNameBoProp = customer.Props["CustomerName"];

customer.CustomerName = "Valid Name";

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

Assert.AreEqual("Valid Name", customerNameBoProp.Value);

Assert.IsTrue(customerNameBoProp.IsValid);

Assert.AreEqual("", customerNameBoProp.InvalidReason);

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

customer.CustomerName = "Inv";

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

Assert.AreEqual("Inv", customerNameBoProp.Value);

Assert.IsFalse(customerNameBoProp.IsValid);

StringAssert.Contains("'Inv' for property 'Customer Name' is not valid for the rule 'CustomerName'. The length cannot be less than 5 character", customerNameBoProp.InvalidReason);

}

From this test you can see that the BOProp tracks its current value (Value) as well as its Status (IsDirty, IsValid) and any invalid reasons. The Business Objects Status is then derived from the composite status of all its BOProps.

##### Implementing Property Rules

Property Rules are implemented using a Strategy Pattern (See GOF). This pattern allows the application developer to easily extend the framework by adding new rule types and creating custom Property Rules for the application. (See Extending the Habanero Framework).

The property rules are implemented in such a way that they are available to the user interface allowing for the development of responsive applications with no duplication of rules.

Figure 3.3: The relationship between the Business Object properties and its property rules.



##### Implementing Saving Business Object to DataStore

The details of how an object is saved to the data store are discussed in detail in Section 4. For the purposes of this chapter we are only going to look at how the Business Objects Properties are updated when the object is saved. This test demonstrates how the business object property is updated.

The BOProp has two values, namely the Value and the PersistedPropertyValue. The Value always holds the current value for that BOProp whereas the PersistedPropertyValue holds the value of the property that is currently held in the datastore. As illustrated in this test, a new Customer’s BOProps have no PersistedPropertyValue, when the Business Object has been saved to the Datastore the PersistedPropertValue is updated with the Value.

[Test]

public void Test\_SaveBOProp()

{

//Testing that when a customer is saved the Persisted Property

// Value is updated to the PropertyValue

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

Customer customer = new Customer();

const string newCustomerName = "Valid Name";

customer.CustomerName = newCustomerName;

customer.CustomerCode = "Code";

IBOProp customerNameProp = customer.Props["CustomerName"];

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

Assert.IsNull(customerNameProp.PersistedPropertyValue, "A new object should not have a persisted value");

Assert.AreEqual(newCustomerName, customerNameProp.Value);

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

customer.Save();

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

Assert.AreEqual(newCustomerName, customerNameProp.Value);

Assert.AreEqual(newCustomerName, customerNameProp.PersistedPropertyValue,

"After saving the PersistedPropertyValue should be backed up to the property value");

}

##### Implementing Business Object Identity

The Business Object identity is implemented in a highly flexible yet simple manner in Habanero. The implementation follows the identity field pattern (Fowler - 216). The object identity (PrimaryKey) is defined as consisting of one or more properties (BOProp) of the business object. The ClassDef for the customer will be loaded with a PrimaryKeyDef. The PrimarKeyDef will be loaded with the PropDef(s) that make up its primary key (in this case the CustomerID). When the Customer Object is created, the Customer Object is created with a BOPrimaryKey that references the CustomerID BOProp (i.e. the BOPrimaryKey is thus merely a collection of BOProps). If at any point the user requests the ID, the BOPrimaryKey Object will be returned.



Figure 3.2: The relationship between a Business object instance and its primary key definition.

From this test it is evident that the PrimaryKeyDef is created with a single PropDef and that the BOPrimaryKey is created with a reference to the BOProp for CustomerID property. See the TestBusinessObjectIdentity class.

[Test]

public void Test\_CreateCustomerWithIdentity()

{

//This test shows the that the PrimaryKeyDef and BOPrimaryKey

// are set up according to the class definition defined for

// customer. It also shows that the CustomerID is automatically

// set to a new GUID Value.

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

Customer customer = new Customer();

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

ClassDef customerClassDef = customer.ClassDef;

PrimaryKeyDef primaryKeyDef = customerClassDef.PrimaryKeyDef;

Assert.AreEqual(1, primaryKeyDef.Count);

IPrimaryKey customerPrimaryKey = customer.ID;

Assert.AreEqual(1, customerPrimaryKey.Count);

Assert.IsTrue(customerPrimaryKey.Contains("CustomerID"));

IBOProp customerIDBOProp = customerPrimaryKey["CustomerID"];

Assert.IsNotNull(customerIDBOProp, "Since the CustomerID is an object id it should be set to a value");

}

In the TestBusinessObjectIdentity class we also have the example Test\_CreateBO\_WithNonGuidID of creating an object that does not use a Guid Object ID. This Test is interesting because it also shows how a BusinessObject can be defined directly in code, without the use of ClassDefs.xml. We do not go into the details of defining BusinessObjects programmatically in the book – if you are interested look at the test and it’s BusinessObject.

If you are using a non Guid Object identifier the following should be remembered:

1. All properties that are part of the PrimaryKey should be Compulsory and WriteNew (Properties cannot be null and cannot be changed once the business object has been persisted).
2. If you are using your own Number Generator to generate a unique identifier for the Business Object then you should refer to Extending the Habanero Framework.
3. If the properties must be mutable (i.e. changeable) then you should ensure that the modification of the ID can only be done via a separate business process which is well isolated and/or that only users who have a high level of authority can edit these properties. You should use concurrency control with pessimistic locking (see Section 4) when editing a primary key ensure that no concurrency control conflicts occur.

##### Implementing Alternate Keys

Alternate keys are implemented in an almost identical manner to primary keys. In fact the BOPrimaryKey inherits from BOKey and PrimaryKeyDef inherits from KeyDef. The difference is in the relationship to the business object whereas a business object can only have one primary key it can have many alternate keys. As with the primary key the BOKey can be composite or single. The uniqueness of the alternate key is only verified when trying to persist the object to the database. The alternate key can be mutable or immutable as required by the business rules.



Figure 3.4: The relationship between a Business object instance and its alternate key definitions.

##### The Business Object Manager

It is essential when developing enterprise applications that a single business object is never represented by two instances of the same business object in the same application instance because of concurrency control issues. Imagine that a user loads one instance of Customer X via one user interface, then the same user loads a second instance of Customer X and edits the second instance of Customer X. If the user now returns to the original user interface, the edited data will not be shown i.e. the user will see the old data even though they just edited the object. If you want more details on this issue then refer to the Identity Map Pattern - Fowler – 216.

In order to prevent the issue discussed above, the Habanero Framework implements the Identity Map Pattern using the BusinessObjectManager Class. The BusinessObjectManager is in essence a dictionary of Weak References to Business Objects. The weak references are keyed on the ObjectID. A weak reference is in essence a safe pointer to an object i.e. it is a reference to an object that does not prevent the object from being garbage collected.

There are no tests in CustomerExample for the BusinessObjectManager because most of its methods are internal but the Habanero.Test.BO project (which ships with the framework) contains the TestBusinessObjectManager Class that shows the behaviour of the object manager in detail.

In summary:

* When a new Business Object is first persisted it is added to the BusinessObjectManager.
* When a Business Object is retrieved (loaded from the datastore) for the first time, it is added to the BusinessObjectManager.
* When a Business Object that already exists in the BusinessObjectManager is retrieved, the already existing Business Object from the BusinessObjectManager is returned. (Note: there are various options for refreshing or not refreshing the already loaded business object that are covered in Section 4 – Data Access Layer).
* When a Business Object is collected by the garbage collector it is removed from the BusinessObjectManager.

##### Loading Business Object from DataStore

The details of mapping and loading a business object from a data store will be discussed in Section 4.

For this section we will look at how the BOProp is loaded with the appropriate data. When a Business Object is loaded from the datastore the BOProp’s PersistedPropertyValue and Value are loaded with the value currently held in the DataStore.

TODO: reference test

##### Summarising the Business Object Definition

In summary the business object consists of a BOPrimaryKey and a collection of BOProperties.

The BOProperties are responsible for maintaining their own state and for validating any business rules such as ReadWriteRules, Compulsory rules or IPropRules.

The Business Object is constructed with the appropriate BOPrimaryKey and BOProps based on the ClassDefinition contained in the ClassDefs Collection.

The BOProp is validated using the Property Rules associated with its PropDef.

The advantages of having a Business Object made up of intelligent BOProps representing the Business Object’s data instead of simple fields should already be evident. The advantages will become even clearer when studying the UI Layer and the DataAccess layer. In preparation however we will briefly mention the following:

1. The BOProp has the knowledge of whether the user is allowed to edit it. The appropriate user control can therefore be enabled or disabled appropriately.
2. The BOProp has direct access to its property rules. The user interface can utilise these rules directly to do data input validation. This results in highly responsive user interfaces while still ensuring that the Business Objects enforce all their rules with no duplication of rules.
3. The PropDef contains the required information on how it is mapped to any field in the DataStore allowing the application developer to load Business Objects without having to know the database table or field names.

#### Applying Security to Business Objects

Every enterprise application requires reliable application security. Security is a broad topic and encompasses many areas that beyond the scope of this book. However, having said that, it is a requirement that security is applied appropriately to the Business Object Layer. Security based rules for accessing, deleting and editing data is just as appropriate in the Business Object Layer as ReadWrite rules, Compulsory rules and Validation rules. In other words authorisation rules regarding which functions and data a user can access logically belong in the Business Object layer and the Business Objects should implement them where appropriate.

Security is one of the areas of application development that involves many decisions and is highly dependent on the environment in which the application is being deployed. It is for this reason, that we do not include a security policy in the business object layer but instead provide the strategy (and thus the hooks) that enables an application developer to implement his/her own security policy.

To understand security we need to clearly differentiate between authentication and authorisation. Authentication involves the verification that the user is who he/she says he/she is. Authorisation involves determining whether the user is authorised to use a particular function (whether it is an application, a business object, a business object property, a user interface or a report).

In implementing authentication, there are primarily two security options:

1. Integrated Authentication: Use Windows Integrated security i.e. use the Windows user who is currently logged onto the Windows operating system as the authenticated user.
2. Custom Authentication: A custom application is used for logging users into the application or into a set of applications. The logged on user as per the application could be different from the logged on user as per the Windows operating system.

The Habanero Framework supports the use of both Integrated authentication and custom authentication. Authentication will be discussed in more detail under Section 5 – The UI Layer.

In implementing authorisation, there are once again two main options:

1. Integrated Authorisation: Use Windows Integrated Authorisation. Windows AD, for instance, allows the user to be set up as a member of certain roles or user profiles. These profiles can then be used by the application to grant or deny the user permissions for specific functions. If integrated authorisation is used then Integrated authentication must be used.
2. Custom Authorisation: A custom application is used to associate the user as a member of a set of Roles/Profiles. These profiles are then used by the application to grant or deny the user with permission for certain functions. Note that a user could be authenticated using Windows authentication but the profiles could be managed using custom authentication.

The Habanero Framework supports the use of both Integrate Authorisation and Custom Authorisation.

##### Implementing Authentication

Authentication will be discussed in more detail under Section 5 – The UI Layer.

##### Implementing Authorisation

The Business Object layer has three areas in which security rules are typically applied in. These are:

1. Implementing CRUD (Create, Read, Update, and Delete) rules for a Business Object.

Typically a certain Group (Role/Profile) of users may have different permissions to Creating, Reading, Updating and Deleting Business Objects (Typically called CRUD permissions).

1. Implementing Read Write Rules for a particular property.

The common use is that some users may be able to view/edit certain Properties of a Business Object and others users may be able to view/edit other properties.

1. Implementing Permission on specific methods.

Once again specific methods may only be executed by certain groups of users.

The security rules are implemented by the business object layer. With this approach the user interface can respond appropriately to the rules by enabling or disabling controls without requiring duplication of the rules in the user interface. This allows the business objects to be safely used by different clients (e.g. Windows Forms, ASP, WebServices or other Services).

On a practical basis the Business Objects are by default distributed in a component that is separate to the User Interface. If security is only implemented in the User Interface a malicious user could create an application that uses the Business Logic Layer DLL’s and access functionality that is not normally allowed.

Before we go any further we will look briefly at the DotNet Security Model. This is a role-based Security Model available in the System.Security.Permissions namespace.

Security can easily be implemented on Methods using the following Code. This enforces that only a user who is a member of the accounts role can access this method.

[PrincipalPermission(SecurityAction.Demand,Role="Accounts")]

public long ApproveOrder() {//DoSomething}

By Default the Role=”Accounts” is set up in a User Group in Windows, AD etc. This is seldom useful for serious Application Development because the UserGroup Roles are seldom finely grained enough for an Enterprise Application.

For this book we are going to focus on how to use the appropriate Security Components to implement Security in the Business Object Layer.

This discussion therefore assumes that you have the role based security components set up appropriately. It turns out that this is the simple part – the next thing to achieve is to assign permissions to any relevant entity based on the role. For instance, a member of the Accounts Group can create an Invoice and edit an Invoice but cannot delete an invoice or edit an Invoice’s Text. A member of the Account Managers Group can edit an Invoice and can edit an Invoice’s Text but cannot delete an Invoice.

Role based security

The Dot Net security model provides the user with a powerful role based model for implementing security. The user can be assigned roles by windows e.g. by AD or the user can be assigned roles by custom code. We have seldom found that the Roles and groups assigned to users for network permissions are fine grained enough for application security so we have tended to use windows authentication with custom roles. (see the Section 5 – UI Layer).

##### Applying Authorisation to a Business Object

Once roles are assigned to a user, the framework still needs to assign the user with permissions to implement specific functionality e.g. a member of the ‘Account Manager’ role may edit the description on a non revenue recognised invoice.

In our experience the authorisation requirements are so varied that the only practical method to implement security is using the strategy pattern. Each business object has a reference to an IBusinessObjectAuthorisation object. This allows us to deal with complex security policies e.g. scenarios where a Business Object such as an invoice could have the description edited by members of the Customer Services and by member of Account Manager. Once the Invoice had been printed and sent to the tax authorities for a clearance certificate then only the Account Manager could edit the description. The design illustrated below allows this flexibility in implementing business objects security. In most cases the same authorisation object will be applied to all business objects of a certain type. In this case the business objects will all reference the same instance using the singleton pattern.

As can be seen in the diagram below a business object references an instance of the IBusinessObjectAuthorisation class.



Chapter 3-5: The relationship between Business Object and the security interface.

The tests and setting up of tests for security is somewhat complicated so we will not go into all the details in this book. The test class TestSecurityCustomer has all the details required.

In summary:

if the security policy does not allow the business object to be read then the customer.IsReadable(out message); will return false and the message will give the reason that it is not readable.

The IsEditable, IsDeletable and IsCreatable work the same.

IsCreatable is an instance method and not a static method because static methods cannot be overridden. These methods can be used by the user interface enable or disable controls. In addition, these methods are used by the business object itself to prevent the Creating, Reading, Editing and Deleting of the Business Object. The tests in TestSecurityCustomer illustrate this.

Below we have shown only the tests for Reading but there are similar tests for Editing, Deleting and Creating.

[Test]

public void Test\_BusinessObjectAuthorisation\_AllowRead\_False()

{

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

IBusinessObjectAuthorisation authorisationStub = new AuthorisationStub();

Customer customer = new Customer();

customer.SetAuthorisation(authorisationStub);

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

Assert.IsFalse(authorisationStub.IsAuthorised( BusinessObjectActions.CanRead));

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

string message;

bool isReadable = customer.IsReadable(out message);

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

Assert.IsFalse(isReadable);

StringAssert.Contains("The logged on user", message);

StringAssert.Contains("is not authorised to read ", message);

}

[Test]

public void Test\_LoadExistingBO\_Fail\_AllowRead\_False()

{

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

IBusinessObjectAuthorisation authorisationStub = GetAuthorisationStub\_CanCreate\_True();

Customer customer = CreateNewCustomerValid();

customer.SetAuthorisation(authorisationStub);

customer.Save();

authorisationStub = GetAuthorisationStub\_CanRead\_False();

customer.SetAuthorisation(authorisationStub);

IPrimaryKey id = customer.ID;

BusinessObjectManager.Instance.ClearLoadedObjects();

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

Assert.IsFalse(authorisationStub.IsAuthorised( BusinessObjectActions.CanRead));

Assert.IsFalse(customer.Status.IsNew);

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

customer = BORegistry.DataAccessor.BusinessObjectLoader.GetBusinessObject <Customer>(id);

try

{

customer.GetPropertyValue("Prop1");

Assert.Fail("expected Err");

}

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

catch (BusObjReadException ex)

{

StringAssert.Contains("The logged on user", ex.Message);

StringAssert.Contains("is not authorised to read ", ex.Message);

}

}

From these tests it is evident that all you have to do is set the correct authorisationpolicy when the Business Object is being constructed.

protected internal Customer(ClassDef def) : base(def)

{

SetAuthorisationRules(new CustomerAuthorisationRules());

}

Once this is done, the framework will do everything else that is required. The example project ??? shows this in great detail. If no authorisation rules are set for a business object then it is assumed that everyone has create, read, update and delete permissions.

Note the IsEditable, IsReadable etc methods are overridable in the Business Object. It is important that these methods should be marked as Sealed methods in the Sub Classes since this will prevent malicious use of the Component by a developer who might try to inherit from the base class.

##### Applying Authorisation to a Business Object Properties

Business object properties follow a similar pattern to the business object with the difference being that a BOProp can only have Read and Write (Edit) permissions set. Setting permissions on a particular Business Object Property (BOProp) are much less frequent than setting permissions on a Business Object.

The BOProp Rules are again setup for a particular Business Object in its constructor redo should be set from a lazy initialisation method e.g.

protected internal Customer(ClassDef def) : base(def)

{

IBOPropAuthorisation propAuthorisationStub = new CustomerNameAuthorisationPolicy();

BOProp propCustomerName = (BOProp)this.Props["CustomerName"];

propCustomerName.SetAuthorisationRules(propAuthorisationStub);

}

The BOProp has methods IsEditable and IsReadable. These methods return true or false based not only on the security policy for the BOProp but also on the ReadWrite Rules set for the BOProp. These two methods can be used by the user interface to enable/disable a control or in the case of IsReadable being false the user interface can blank out the control to show that it is not readable. Regardless of the user interface, the Business Object and its business object properties will prevent the reading and writing of business objects properties where the rules prevent it. For detailed tests showing this functionality refer to the TestBOPropAuthorisation class in the Habanero.Test.BO.Security namespace of the Habanero.Test.BO Project that is downloaded with the application framework.

##### Applying Authorisation to a Business Object Method

The business object methods can also have security applied to them. In these cases the mechanism is similar but different because there is only one action that the user can do on a method and that is to execute the method.

The Interface used is IBOMethodAuthorisation. An authorisation rules object of this type is created. The roles that can execute the method are added using AddAuthorisedRole probably during object construction and the method then calls IsAuthorised on the authorisation rules object prior to executing. If the use is not authorised, then the appropriate error must be raised by the method.

#### Extending the Business Object

There are a number of points provided in the Business Object framework that are specifically intended for extending the framework. We will briefly mention each method, how it is intended to be used and where you can find tests that illustrate their use.

#### AfterLoad

This method is virtual on the Business Object and by default, implements no code. The application developer can override this method to carry out any custom code that is required after the object is loaded from the DataStore. This method will be called if the object is loaded for the first time or if the object is refreshed.

#### AfterSave

This method is virtual on the Business Object and by default implements no code. The application developer can override this method to carry out any custom code that is required after the object is saved to the DataStore. This method will be called every time the object is persisted.

##### UpdateObjectBeforePersisting

This method is virtual on the Business Object and by default implements code that adds a transaction log object to the transaction committer if a transaction log is defined for the Business object. The application developer can use this method to carry out any custom code that is required before the object is persisted to the DataStore. This method is called just before the object is persisted and allows the application developer to implement any custom code. A typical example where we use this is where there is a requirement to generate a custom number or sequential code for a business object, e.g. each invoice must have a number which is unique. For these cases, a particular implementation of the INumberGenerator class is usually used (note the framework contains a few implementation of INumberGenerator). The INumberGenerator object is then added to the transaction Committer to ensure that the number generator and business object are both either updated or rolled back. For more details on the TransactionCommitter see Section 4 – Data Access Layer.

##### Adding rules for a BOProp

The framework allows the Application developer to add custom BOProp Rules in addition to the standard rules defined in the ClassDefs.xml.

The PropRules are added to the BusinessObject in its constructor e.g.

protected internal Customer(ClassDef def)

: base(def)

{

BOProp propCustomerName = (BOProp)this.Props["CustomerName"];

PropDef propDef = (PropDef) propCustomerName.PropDef;

propDef.AddPropRule(new MyPropRule(this, propCustomerName));

}

##### Are Custom Rules Valid?

This method is virtual on the Business Object and by default returns true. This method is called prior to saving any Business Object. The application developer can override this method so as to implement any complex custom rules that cannot be handled by the IPropRule strategy. The disadvantage of placing custom rules in this method is that they are unavailable as broken rules to the user interface, thus making it difficult to show them using the ErrorProvider. Most rules could be implemented using either the IPropRule Strategy or the AreCustomRules Valid strategy the reason we allow both is that rules where the value of 1 property relies on the value of another property may not be best validated using PropRules. The best way to demonstrate this is via an example

Lets assume you have a Business Object that has a property for Date Of Birth and also for Identity Number. The identity number has 4 numerics that indicate the month and year of birth. There is a requirement that the date of birth is validated against the Identity number to ensure the month and year of birth are correct. A prop rule could be set on the DateOfBirth Validating against the Identity Number but this would imply that the identity number has already been captured causing a coupling between the Business Object being updated with an identity number and the Date of birth.IsEditable

By default the IsEditable Method checks the authorisation Rules set up for the Business Object. The Application developer can however override this and implement any other rules that may change whether the object is edited or not.

public virtual bool IsEditable(out string message)

{

message = "";

if (\_authorisationRules == null) return true;

if (\_authorisationRules.IsAuthorised(BusinessObjectActions.CanUpdate)) return true;

message = string.Format("The logged on user {0} is not authorised to update {1} Identified By {2}",

Thread.CurrentPrincipal.Identity.Name, this.ClassName, this.ID.GetObjectId());

return false;

}

##### IsDeletable

By default the IsDeletable Method checks the authorisation Rules set up for the Business Object. The Application developer can however override this and implement any other rules that may change whether the object is deletable or not.

##### Logging Edits

The Framework once again provides the application developer with the hooks to implement a logging strategy if he/she requires. The required logging strategy is set up for the Business Object in its constructor using the protected void SetTransactionLog(ITransactionLog transactionLog) method. The framework also ships with objects that implement a particular logging strategy. For more details see the TestTransactionLogger class which shows the use of the TransactionLogTable.

##### Concurrency Control

The framework once again provides the application developer with the hooks to implement a any concurrency control strategy if he/she requires. Concurrency Control is a complicated topic and requires a fair amount of interaction with the database to implement. The subject is therefore fully discussed in the Section 4 – The Data Access layer. The Habanero Framework ships with a number of different implementations of various concurrency control strategies. The test classes TestConcurrencyControl\_OptimisticLockingVersionNumberDB and TestConcurrencyControl\_PessimisticLockingDB provide examples of setting up and using concurrency control in a business object.