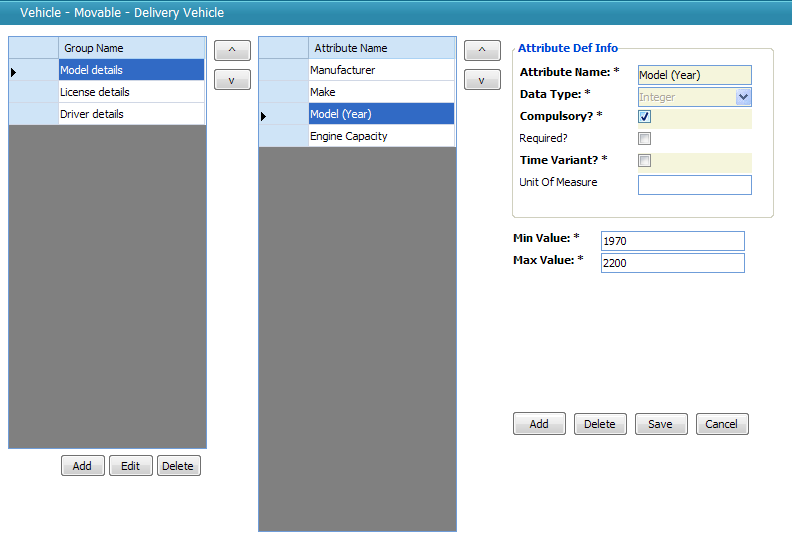
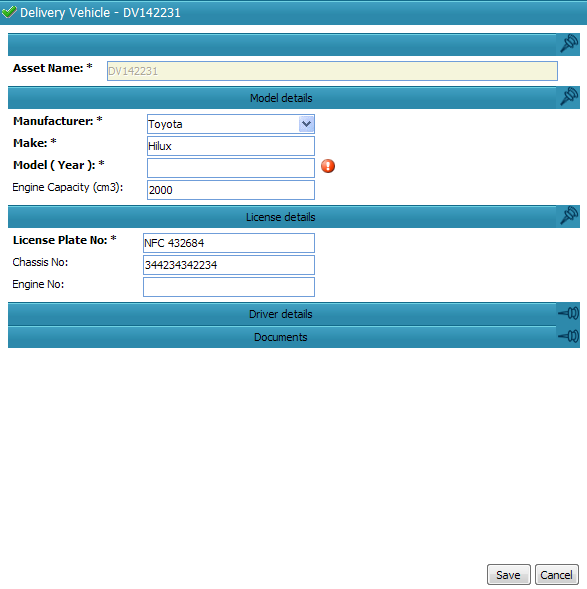
# Dynamically Generated, User Defined Interfaces

Recently our company encountered a project which required the ability for a superuser to configure the data stored for an asset as well as the way in which the asset is displayed in grids and report as well as for data capture forms. In other words the Business Object properties, database data and user interface froms all had to be definable and customisable by a super user at the customer’s site. The problem domain was asset management and each organisation using the system had to be able to define their asset types via the system. The user interface was required to be created on the fly to cater for the asset attributes and groups set up by each organisation. Here is an example of the screen in which an asset type is configured:

Below is an example of a Delivery Vehicle asset type being defined in the system:



The user is able to create groups of attributes (a group of attributes was merely a set of attributes that would be displayed together on the form), assign attributes to those groups, alter the order of each group and attribute. In addition for each attribute (Property of the Asset) the user would be able to define the data type of the attributes as well as any rules that govern the values of the attributes (In this case Max, Min Values and Compulsory or not) but also an lookup lists values. For instance, the selected asset type *Delivery Vehicle* has three groups of information, and within the *Model details* group we find four attributes. The one selected is *Model (Year)*, and it is a compulsory integer field with a minimum and maximum value set. With this definition of an asset, the system creates a screen that can be used to capture a delivery vehicle’s details:



This form is created on the fly at run time complete with indications of which fields are compulsory, with error providers that indicate any contravention of the rules set up for the attribute, with correct control types based on the types of the attributes (such as the combo box provided for the Manufacturer field) and with panels that group together the attributes.

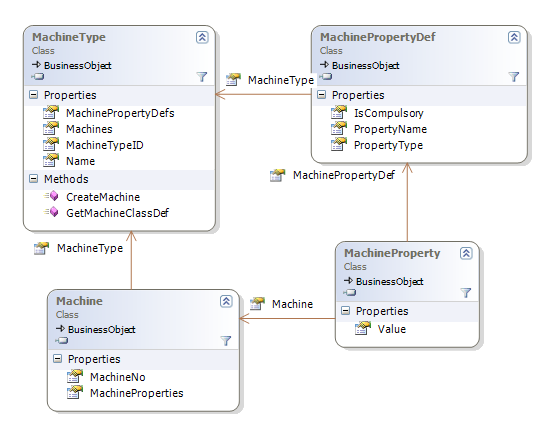
Developing an application where the user defines the user interface for each business object as well as the data stored and the data capture rules for this data would generally be considered highly complex. The most impressive part from our point of view was that these forms are not generated by hand-crafted code for this specific project – instead they leveraged the form building that we had been using for years in the Habanero Enterprise Application Framework (or Habanero for short).

### Defining User Interfaces at Run Time: Code Sample

Let me move on to the example contained in the code sample. We have in the past encountered a very similar idea in the manufacturing sector, where a plant’s machine types needed to be modelled and where new, upgraded machines and new processes are added on a fairly regular basis making it impossible to statically create user interfaces for each of them.

Please note that the code sample is not from the production application – it is simply an example put together in a few hours. It is not thoroughly tested and even has some known issues. It was put together to show the core concepts under discussion. It also is using an in-memory data store to simplify distribution, but this can be swapped out for a real database if required. It uses the Habanero Enterprise Application Framework, but all the required DLL’s are contained in the zip.

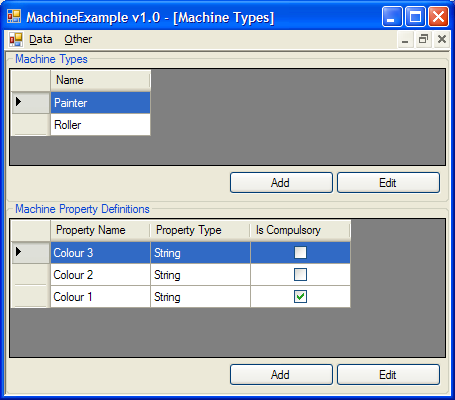
If you open the solution in the example and look at the class diagram in the main project, you should see something like this:



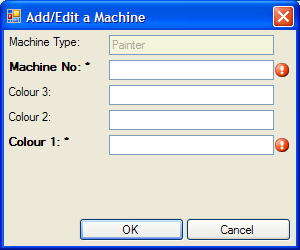
In this example a Machine Type is defined by a collection of Machine Property Definitions. If I use the example of aluminium rolling machines, these might be properties such as maximum width, max gauge in, max gauge out and so on. Each Machine that is captured will have a Machine Type, and will receive all the properties defined in the definitions for that Machine Type. If you map this to traditional object oriented principles you can think of the Machine Type and Machine Property Definitions together as the “Class” and the Machine and Machine Properties as the instantiated “Object”.

In the sample application I have written some code to set up some test data when the programme starts up – two machine types along with a few properties each machine type.

The screen below shows the definition of the Machine Type. In the screen below (which appears under Data | Machine Types) you can see the Painter machine type has three configured property definitions, one of which is a compulsory property.



If you then go to Data | Machines and choose to add a Painter machine, a form will pop up that looks like this one below. The form contains the properties as defined above and has full validation. You therefore get the appropriate errors showing if you try to press OK without entering data:



You can see that the form has the configured properties on it, including the indications of compulsory fields. If you had added a few more property definitions to the Painter machine type before creating this new Painter machine the new fields would have shown up on the form too.

### Getting a little more technical

Now to the technical details: at its heart, Habanero’s UI layer has an object model describing a form structure:



This simplified class diagram indicates how Habanero models a form as a set of fields. Each field has the necessary information to be able to instantiate the correct control as well as the correct mapper to maps the assigned business object’s property to the control and vice versa. Habanero’s UI layer is able to take a structure like this and produce a form like we have seen above. The UI layer also has the ability to map that form to a particular business object and each property to a particular control.

Because of this inherent capability the task of dynamically creating user interfaces becomes the task of modelling the correct data structure to be used in the form creation process. This is easily done by looping through a Machine Type’s Property Definitions and creating a UIFormField for each:

foreach (MachinePropertyDef machinePropertyDef in MachinePropertyDefs) {

UIFormField uiProperty = new UIFormField(null, machinePropertyDef.PropertyName, "TextBox", "System.Windows.Forms", "",

"", true, "", new Hashtable(), null);

form.Add(uiProperty);

}

The various parameters passed into the UIFormField’s constructor simply populate the fields shown in the class diagram above. In the example code I have used TextBox controls for all fields since I am only supporting Strings and Integers for now, but in the real world asset management example above we provided for all common controls. The system determined the default control based on the data type. It is simple enough to select the correct control based on the data type of the field as set up by the user.

Once this structure is set up and registered against the corresponding Machine Type, Habanero’s controls (such as the grid, the form builder etc) simply use it as it is.

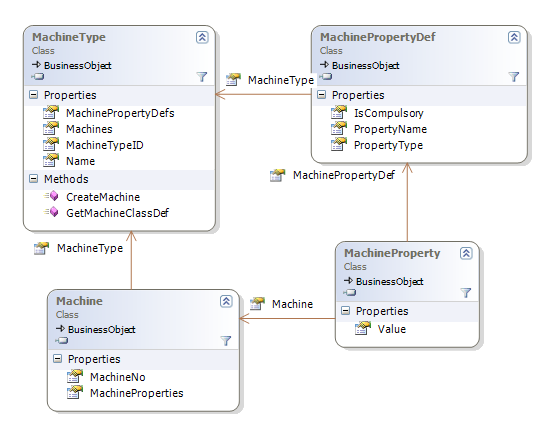
This project showed the power and extensibility of using a framework that maintains the form definitions and business object definition as an object as their structure and behaviour can be modified at run-time based on input data.

Of course, there is still the question of how the controls are mapped onto business object properties, how the business object properties are created based on the definitions tables and how the rules are loaded for each business object. Stay tuned for part 2 tomorrow where I’ll discuss the sample project code in much more detail and explain the mapping from the UI layer to the BO layer and from there to the database.

## Part 2

In the previous article I described the problem I was trying to solve by dynamically generating user interfaces, that of how to allow the user to create their own “types” and how to represent these on the screen. This article focuses more on the concepts and code involved with mapping a dynamic user interface layer onto a domain model, or more specifically, creating a dynamic domain model to go with the dynamic user interface so that the user interface can map easily on to it.

Recall the class diagram contained in the sample project:



At compile time we have no idea what Machine Types are going to be created by the user, and what fields these machine types have. This is why we’ve created this structure, after all. We have the option of modelling our Machine class’ behaviour just as shown in the diagram, a structure that easily maps to the database but makes mapping to a user interface and treating a Machine as a particular Machine Type doable but a little tricky and quite a bit of work. Conceptually what I want is for Machine to be a generic type and to have types such as Machine of Roller and Machine of Painter or, more generically, Machine of T that I can work with, where Roller, Painter or T are machine types that are defined by the user at run time.

In order to get this effect I need to somehow define a Machine domain type in an object form. One way to model this is as follows:



In this example, for each type in my domain model I would have one ClassDef object which consists of one or more PropDef objects that define the properties of the class. The idea is that when I create a domain model object such as MachineType, the object gets a collection of properties based on the PropDefs of the ClassDef configured for MachineType.



In our example MachineType is a class that inherits from BusinessObject. Thus, when a MachineType object is instantiated it receives a collection of BOProp objects as defined by MachineType’s ClassDef. These then become the true properties of the domain model object. We can write C# properties for those we know of at compile time as follows:

public class MachineType : BusinessObject {

public virtual String Name

{

get

{

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

}

set

{

base.SetPropertyValue("Name", value);

}

}

//...

}

The C#, strongly typed, property uses the BusinessObject’s protected GetPropertyValue() and SetPropertyValue() methods to retrieve and set the property values. Having these makes the objects easier to work with (Intellisense!) and behave exactly like normal domain model objects (ie those without dynamically stored properties) would.

This overall structure gives us the benefit of a dynamically changeable set of properties along with the benefit of type-safety for those that are known at compile time.

Now, in order to create a user interface that maps onto this object we must create a set of UIFormFields that define our form (see the brief discussion of this in part 1) and link them to the ClassDef object:



The ClassDef has a collection of UIForms to allow for different views on the same type – one might be a view that only shows a few properties, another might show all of them. Each UIForm contains a collection of UIFormFields and each of these fields corresponds to a PropDef (the link being the PropertyName field). These UIFormFields describe how the instantiated BOProp will be displayed. Through this structure it is possible to dynamically create a form for any subtype of BusinessObject and link the controls to the object underlying them.

However, this does not solve the problem of wanting to have multiple “parametrized” Machine types as there is only one ClassDef per type. So what we did was to change the structure from a one to one mapping between ClassDef and Type, and replace it with a many to one mapping as follows:



The inclusion of the TypeParameter property on ClassDef allows us to differentiate between the many ClassDefs for one Type. Now, in our Machine example we can effectively get run-time “types” by instantiating Machines with the appropriate ClassDef object. In the example programme you can select a MachineType and choose to create new Machine of that type. At this point the system asks the MachineType object for a new Machine as follows:

public partial class MachineType

{

public Machine CreateMachine()

{

ClassDef machineClassDef = GetMachineClassDef();

var machine = new Machine(machineClassDef) {MachineType = this};

//…

return machine;

}

The first line of CreateMachine() gets the Machine ClassDef object corresponding to the MachineType of the Machine being created (we’ll look at GetMachineClassDef() next). The next line creates the Machine object, giving it the appropriate ClassDef object so that it gets the correct properties. After this there is some code used in supporting persistence and loading to a database structure, but that will be discussed in part 3 so I have left it out here. Finally we return the new Machine object which has the correct properties (BOProps) for this MachineType. In other words it is a Machine of T!

Continuing the MachineType class, we move on to the GetMachineClassDef() method:

public ClassDef GetMachineClassDef()

{

string machineTypeClassDefName = "Machine\_" + Name;

ClassDef machineClassDef;

string machineAssemblyName = "MachineExample.BO";

if (ClassDef.ClassDefs.Contains(machineAssemblyName, machineTypeClassDefName))

{

machineClassDef = ClassDef.ClassDefs[machineAssemblyName, machineTypeClassDefName];

ClassDef.ClassDefs.Remove(machineClassDef);

}

machineClassDef = CreateNewMachineClassDef();

ClassDef.ClassDefs.Add(machineClassDef);

return machineClassDef;

}

This method retrieves the ClassDef object corresponding to a Machine of T. When searching the ClassDefs dictionary we give it a parametrized name such as “Machine\_Roller”, or “Machine\_Painter”, and the ClassDefs collection retrieves the ClassDef object that matches this.

An important consideration to note is that even if it finds the matching ClassDef it removes it and creates a new one. This is because the very definition of the ClassDef might have been altered by an action of the user since the ClassDef was constructed, for example by adding or changing a MachinePropertyDef for that MachineType. To ensure that the Machine being created has got the latest definition of its type we create the ClassDef every time. Of course this is the naïve solution – a better one would be to replace or update the ClassDef when, and only when, a MachineType is changed and that is how we have implemented it in production projects.

The final piece to look at is how the ClassDef is created because it is in this process that the UI form definition is built up too:

private ClassDef CreateNewMachineClassDef()

{

ClassDef baseMachineClassDef = ClassDef.ClassDefs[typeof (Machine)];

ClassDef machineClassDef = baseMachineClassDef.Clone();

machineClassDef.TypeParameter = Name;

Here the basic Machine ClassDef is taken, cloned and a type parameter is applied. If we’re getting the ClassDef for a MachineType called “Roller”, the TypeParameter property will be set to “Roller” and conceptually we now have a ClassDef for a Machine of Roller.

UIDef uiDef = machineClassDef.UIDefCol["default"];

UIGrid uiGrid = uiDef.UIGrid;

UIForm form = uiDef.UIForm;

form.Title = "Add/Edit a Machine";

UIFormTab tab = form[0];

UIFormColumn column = tab[0];

This code is simply getting the UIFormColumn object out that contains all the UIFormFields. I left the UIFormTab and UIFormColumn out of the diagram earlier as they are simply organisational structures used for layout that don’t affect the way we conceptually think of a form as simply containing a set of fields. You’ll see that a UIGrid is also retrieved – this is so that we can set up the columns for grids showing this Machine Type (grids are modelled in the same way as forms but with a slightly different structure).

foreach (MachinePropertyDef machinePropertyDef in MachinePropertyDefs)

{

var machinePropertyPropDef = new PropDef(machinePropertyDef.PropertyName, "System", machinePropertyDef.PropertyType, PropReadWriteRule.ReadWrite, "", null, machinePropertyDef.IsCompulsory.Value, false);

machinePropertyPropDef.Persistable = false;

machineClassDef.PropDefcol.Add(machinePropertyPropDef);

Here we set up the Machine of X ClassDef with all of X’s properties as they are defined by the MachineType’s MachinePropertyDefs. They’re marked non-persistable as their persistence is dealt with separately via the MachineProperty objects. For now I am ignoring persistence – it will be covered in part 3. Note that when the PropDef object is created it is given the Machine Property Definition’s PropertyType and IsCompulsory values so that they can be validated before saving.

var uiProperty =

new UIFormField(null, machinePropertyDef.PropertyName, "TextBox", "System.Windows.Forms", "", "",

true, "", new Hashtable(), null);

column.Add(uiProperty);

This adds a form field for the MachinePropertyDef we’re working with, a field that will show up on the form for a Machine of T.

uiGrid.Add(new UIGridColumn(machinePropertyDef.PropertyName, machinePropertyDef.PropertyName, "", "",

false, 100, UIGridColumn.PropAlignment.left, new Hashtable()));

}

In the same way we add a grid column. In the sample application you can see this in action: when you select the different Machine Types in the Data | Machines screen the grid columns change depending on which one is selected. The grid is configured at run time with the columns added in this line of code.

return machineClassDef;

}

}

Finally we return the newly created and configured ClassDef for a Machine of T.

These three methods on the MachineType class have fully configured the dynamic creation, viewing and editing of whatever Machine Types you can imagine. No further customisation is required – when a Machine is edited the UI generation classes simply use the ClassDef associated with that Machine of T to create the form or grid.

The last item to look at is how the dynamic properties of the Machine are persisted to and loaded from the database. Look out for part 3 in the next few days!