Most enterprise developers (i.e. developers that write software for business) have come up against a situation where some sort of dynamic structure is needed. Most often this comes in the sort of innocuous request from a user to be able to “define their own fields”. In this simple request lies much complexity in development because we know it means eschewing our favoured database designs and creating one that can be “altered” by the user. I’ve encountered this situation in laboratory systems and manufacturing where each machine must be user definable. The most obvious case though is the management of assets, so let’s start with a simple example so we’re on the same page.

So you’re in the initial stages of designing a system in which the user can capture their company’s assets with a view to tracking their value for accounting purposes, and after some discussion you come up with a core object model like this:



Diagram1-AssetClass1.vsd

This, if implemented, might map onto a database structure like this:



Diagram2-AssetTable1.vsd

Note: typically you wouldn’t worry about database implementation until you were happy with your design, but I’m going to show each design’s database structure to make things a little clearer (hopefully).

The user goes away, thinks about it and comes back with the idea that for vehicles they wish to capture the license details in the system. In other words, they wish to do a little more than track value in the system – they want to use it to track information related to each vehicle too. Now you might add new fields for the license plate number, chassis number and engine number, and then you might decide to create a new class in your application called VehicleAsset to reflect this:



Diagram3-AssetClass2.vsd

I’ve implemented this an inheritance here because a VehicleAsset is most definitely an Asset, but with a little more information. Again, this might map on to a database with the following structure:



Diagram4-AssetTable2.vsd

Here I’ve used the simplest of the inheritance mapping structures, single table inheritance, or table-per-heirarchy. (see PoEAA) This structure is still not very complex, and is suitable for everything the customer has needed this far.

Soon enough, the user comes with the request to be able to store model numbers, makes, warranty information and all kinds of other information. You could add each of these fields, creating classes for each new asset type, but it’s swiftly becoming clear that a static data structure might not be what is best for this system. Of course, you must be sure before you jump to a dynamic structure because the massive extensibility and flexibility means a non-trivial increase in development complexity levels.

But what do I mean by a dynamic structure? In a class diagram it might mean a move from the above structure to the following:



Diagram5-AssetClass3.vsd

Now a Vehicle asset type will be defined by a set of AttributeDef (short for Attribute Definition) objects. You can see that each AttributeDef has a name, a description, a flag indicating whether it is compulsory and a type such as string or integer. I would argue for keeping the number of types supported to a select few unless you find that is also required to be extensible, in which case you will have to go beyond the scope of this article. The Asset class still has all the static properties needed for every asset, but whenever a new Asset is created it will also get a collection of newly created Attributes that are defined for the Asset Type – one per AttributeDef.

Once again I will show the database diagram for the above class diagram:



Diagram6-AssetTable3.vsd

This structure solves the data storage issue, and it’s possible to use the above class structure to implement what is needed. There is, however, much complexity involved with creating user interfaces to allow users to capture the asset information, as well as some hand crafted mapping required to get the data from the user interface to your domain classes and then to the database. Also, note that this is the simplest of cases – a real-world scenario might require some AttributeDefs to have a lookup field, or require some rules to be defined (such as maximum or minimum values, or pattern matches) or might require some configuration of the user interfaces that are created such as a grouping of attributes under a heading.

Before we tackle how it might be possible to implement this in a way that almost makes Asset a generic type (at run-time), I want to discuss a Business Object Layer that we have used. After discussing this we will come back to the dynamic properties problem.

The Habanero Business Object Layer

The Business Object Layer (BOL) discussed here is that implemented in the Habanero Enterprise Application Framework. The Framework offers a lot more than simply a BOL, and will be used to reduce both object-relational mapping and user interface complexities.

At its heart the BOL of Habanero has a class structure as shown here:



Diagram7-HabaneroBO1.vsd

The model has a BusinessObject class which is intended to be the base class of all domain classes, so my Asset class, for instance, would inherit the collection of BOProp (short for Business Object Property) objects and a reference to the ClassDef (short for Class Definition) as indicated by the diagram. The ClassDef and its collection of PropDefs (short for Property Definition) define my class in a run-time structure that is usually built up from an XML definition file but can be modified after it has been loaded. There is much symmetry between the structure shown here and the Asset structure mentioned before – the classes seem to map directly: AssetType -> ClassDef, AttributeDef -> PropDef, Asset -> BusinessObject, Attribute -> BOProp. This indicates that it might be possible to utilise this structure effectively for our Asset management example.

At the Habanero BOL object model’s heart is the ClassDef class. Each and every business object class that you require will have one (or more, but we will get to that) ClassDef object. So if I have an Asset class inheriting from BusinessObject, I will also have a ClassDef object linked to it. All objects of a type share this same ClassDef object, and it is used when creating the objects to set up its properties. Here is simple XML representation of the ClassDef for our Asset (excluding any relationships to other classes for now):

<class name="Asset" assembly="MyAssets.BO" >

<property name="AssetID" type="Guid" compulsory="true" />

<property name="AssetTypeID" type="Guid" compulsory="true">

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

<property name="SerialNo" />

<property name="PurchasePrice" type="Decimal" />

<property name="PurchaseDate" />

<property name="CurrentValue" type="Decimal" />

<primaryKey>

<prop name="AssetID" />

</primaryKey>

</class>

Source1-AssetClassDef.txt

The class element maps to an object of type ClassDef, while for each property element a PropDef object is created. Note that the Asset class could be as simple as :