Unleashing enterprise models with Borland® Delphi™ 8 Architect for the Microsoft® .NET Framework

An in-depth look at the features of

Borland® Enterprise Core Objects (ECO™)

using a real-life sample application

A Borland White Paper

By Christophe Floury,

Chief Architect, Neosight Technologies Limited

July 2004

Borland®

Contents

Introduction 4

What are Borland® Enterprise Core Objects (ECO™)? 5

UML® runtime 5
Microsoft® .NET Framework metadata vs. UML metadata 5
ECO objects extend .NET objects 6
ECO objects need .NET objects 6
The ECO namespaces 7
The Borland.Eco.ObjectRepresentation 8
The Borland.Eco.UmlRt namespace 8
The Borland.Eco.Services namespace 10
From ECO Space to .NET 10
From ECO types to .NET types 11
The role of ECO tagged values 11

Modeling our sample application: Introduction to Consultancy Manager 1.0 12

Core classes 13
Contact management 14
Product and services 15
Using the DefaultStringRepresentation ECO tagged value 17
Controlling aspects of the database schema 18
Project management 19
Requirements 19
Model overview 20
Using derived attributes with inheritance structures 20
Derived attributes and recursive OCL 21
Considerations regarding the performance of OCL 22
Further performance considerations regarding late fetching 22
Converting OCL into SQL 23
System parameters 23
Implementing a singleton for SystemParameters 24
Implementing CurrentUser() 25

Building the ECO space ... 27
The database is in the same directory ... 28
Implementing update of audit attributes ... 28
User-interface basics: binding objects instead of data ... 29
Logical three-tier ... 29
Initializing the ECO Space ... 30
Binding a grid ... 31
Binding combo boxes to objects ... 33
A simple example ... 33
An advanced example ... 34
Using ECOWinforms and the AutoContainer ... 35
Using the ubiquity of databinding in .NET ... 36
Building an optimized search ... 37
Improving the search screen ... 38
Saving data and multi-user considerations ... 40
Schemes for updating the database ... 40
Other ECO services ... 41
The Object Factory Service ... 41
The OCL Type Services ... 41
The State Service ... 41
The Undo Service ... 41
The Version Service ... 41
Conclusion ... 42

Introduction

With its Model Driven Architecture® (MDA®) initiative, the Object Management Group™ (OMG™) has recognized that in order for its Unified Modeling Language® (UML®) to succeed in further automating the software industry, it had to take a more active role in the development process.

To significantly reduce development time, the use of UML had to extend beyond mere sketching of ideas, reverse-engineering through code visualization, and one-way generation of code. It had to effectively separate the intentions of the system (business rules) from the underlying implementation technology and, above all, drive the maintenance process.

Borland® Delphi™ 8 for the Microsoft .NET Framework, Architect Edition, brings this vision to market with one of the tightest integrations of a GUI builder, a modeling tool, and a UML execution runtime platform for the Microsoft® .NET Framework. Borland succeeds in providing higher productivity through an approach that combines a two-way UML/Delphi language synchronization engine, which generates the minimum code necessary to unleash strongly typed OO programming.

By making such lightweight code generation target prepackaged framework services, Delphi 8 Architect augments the feature set of traditional .NET code and helps avoid the cumbersome traditional roundtrips and their copious amounts of code to manage. These prebuilt services, named Borland® Enterprise Core Objects (ECO™) ensure industrial strength handling of associations, derived attributes, persistence, subscription, region-based optimistic locking, object versioning, OCL-based constraint checking and query capabilities, and much more.

This white paper investigates the power of ECO through the development of a medium-sized system (30 classes) to manage time and billing for consulting organizations: “Consultancy Manager 1.0.” For maximum benefit, download the source code and study it in conjunction with this paper.

What are Borland® Enterprise Core Objects (ECO™)?

Enterprise Core Objects is a set of software components designed to use UML during the initial design, execution, and design evolution (refactoring and extension) phases. The largest ECO components act as a runtime framework for executing a UML model. Design-time ECO components include, for example, the Borland® Together® UML/Delphi language synchronizer (code generator), which handles components to visually connect ECO objects to UI elements and model validation.

UML® runtime

UML model execution keeps the model in efficient data structures[1] and uses its rich information to drive horizontal services such as instantiating UML classes, mapping objects to database structures, synchronizing both ends of an association, enforcing semantically rich constraints thanks to OCL, and forcing objects states to follow designed transitions in state machines.

Microsoft® .NET Framework metadata vs. UML metadata

Since the advent of the Java™ and .NET platforms, the use of metadata and reflection has gained much use and popularity. The .NET runtime is a fine example of an execution environment where metadata survives the compilation stage and takes a central, more dynamic role during execution.

[1]A UML model is a collection of interrelated modeling elements such as classes, associations, attributes, states, and transitions, and is not merely a collection of diagrams. Diagrams are visual representations of carefully selected portions of the model to convey a particular understanding of one aspect of the system. Indeed, not all modeled information is represented in diagrams. And while many diagrams are needed to visualize a system, it is incorrect to assume that several models are necessary in order to specify a system.

However, .NET metadata lacks the richness of the UML semantic. In fact, in many ways it can be considered a fairly rich subset. It has been a noticeable trend in new languages to progressively adopt many of the features introduced by declarative modeling languages such as the UML. For example, we can find namespaces, properties, tagged values (code attributes in CLR) that have long been part of the UML. The point is to increase the level of abstraction by reusing neat packages of horizontal features (appropriate for many domain applications) and thus increase productivity.

ECO objects extend .NET objects

.NET objects are not UML objects; the .NET common language runtime (CLR) is not as rich as that of UML models. For example, in UML, you simply flag a class as persistent and expect that to be the end of the work you have to do. ECO objects are a particular implementation of UML objects. As such, they provide the extra services beyond those of .NET objects.

The ECO runtime is a system that takes as a parameter a complete UML model and creates an instance of it called an ECO Space, in very much the same way that .NET uses metadata to create a process where objects will be created and reside. You can then ask ECO to create instances of UML Classes and Associations and (also unlike a standard .NET process); you can directly link the ECO Space to a file or a database so that ECO objects are persistent.

ECO objects need .NET objects

Before the advent of Action Semantics (now part of the UML 2.0 specification), UML did not provide full execution support. Algorithms and method body definition are left to other languages to define. ECO objects rely on intermediate language (IL) code for their method implementations. ECO locates the implementation for its object methods by letting the developer specify and link traditional .NET classes with UML classes. The Delphi 8 Architect two-way UML/Delphi code synchronization feature automatically does all of this for you. This solution is shown in **Figure 1** below which illustrates how it is possible to navigate both

ways between the .NET Objects and the ECO objects that are surfaced by the powerful `Borland.Eco.ObjectRepresentation` interfaces.

Figure 1: *The main ECO namespaces*

The ECO namespaces

The `Borland.Eco` namespace is divided into six main namespaces:

1. `UmlRt` gives you access to the UML model at runtime
2. `ObjectRepresention` gives you access to ECO objects and values instantiated from the UML model
3. `Subscription` allows you to be notified when an ECO object or value has changed

4. `Persistence` provides automatic saving of ECO objects to files and databases

5. `Handles` provide components available at design-time to reach into the ObjectRepresentation namespace and bind ECO objects to user interface components

6. `Services` expose functionality of the ECO engine such as OCL evaluation, versioning, and undo/redo

The Borland.Eco.ObjectRepresentation

The ECO namespace `Borland.Eco.ObjectRepresentation` shows that ECO objects consist of several elements (`IElement`). It allows you to interact with any elements through a dedicated interface.

These interfaces enable you to navigate the internal structures of an ECO object and to modify values such as clearing a "multilink" as `Person.addresses.Clear`, or adding ECO objects to the association with a simple: `Person1.addresses.Add(Address1);`

Because ECO objects are made up of discrete elements, these elements can have powerful built-in behavior. For example, if `Address1` has an `Owner` property representing the other end of the `Addresses` association, `Address1.Owner` is automatically set when the address is added to the address collection of the owner.

Notable in ECO elements is the built-in support for immutability (or invariant calculated values) and properties representing collections of elements such as what you find when navigating a UML link such as `Organization.employees : PersonCollection`.

The Borland.Eco.UmlRt namespace

The `Borland.Eco.UmlRt` namespace shows that UML model information is available at runtime and can be navigated to from any ECO Object (or part thereof). Notably, we can find out whether a UML class or UML structural feature (such as attributes in UML 1.4) is flagged as persistent. **Figure 2** is an extract of the UML metamodel (the model that describes the data structures to store a UML model and, as such, defines the OMG standard). Note by comparing

the UML 1.4 and the ECO.UmlRt interfaces that ECO contains a speed-optimized (Rt stands for runtime) subset of the UML metamodel.

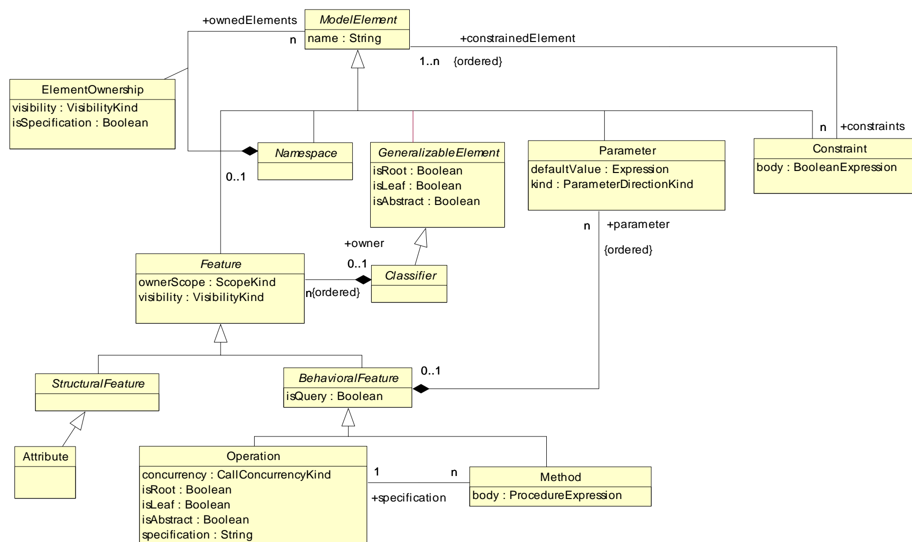

Figure 2: *The Foundation/Core/"Backbone" diagram of the UML 1.4 metamodel*

The top of **Figure 1** illustrates one role of Delphi 8 code generation: generated code can be considered .NET "wrappers" around ECO objects. By calling the AsIObject method on any of the instances of your .NET generated classes at runtime, you gain access to the richer world of ECO objects through the IObject interface pointing to the underlying "implementation" objects.[2]

---

[2]ECO is itself built using .NET technologies. This means that the implementation objects underlying your .NET objects are .NET objects. For example, you could navigate to an IProperty for the "surname" attribute by calling:

```
Var p: IProperty;
…
p := MyDotNetPersonObject.AsIObject.properties['surname'];
```

You could then investigate where this property is defined in the UML model by following:

The UML model is completely stored as .NET metadata via standard Delphi language constructs augmented with code attributes. At initialization, ECO uses reflection to convert this metadata into the `Eco.UmlRt` structures. This solution has two major consequences. The first is that the model is never out of sync with the code, refactoring at the model level (through the modeling surface), refactors the code. The second consequence is that because the design-time ECO components make use of the `ECO.UmlRt` structures to perform various tasks such as validating the model (including all the OCL expressions) and generating schema, the code must be compiled beforehand for the metadata to be available.

The Borland.Eco.Services namespace

Now let's talk about the last part of **Figure 1**: `Borland.Eco.Services`. Those of you familiar with Bold (the predecessor of ECO) might notice that the `IElement` interfaces in `Eco.ObjectRepresentation` are unaware of their underlying functionality, such as the ability to evaluate OCL expressions. In Bold, such services were available through inheritance. All your objects inherited from `TBoldObject`, which itself inherited from `TBoldElement` and therefore had a very large API. Access to such services is now done by requesting an Interface to a specialized subset of functionality. These services are known as ECOServices.

From ECO Space to .NET

You can navigate back to the .NET world by navigating the AsObject property of any IElement. This includes IPrimitive.AsObject, which takes a boxed .NET object version of the primitive value and makes it available in the ECO world (known as ECO Space). Why would you want to have ECO versions of .NET primitive types? The answer is incredible: all ECO elements are subscribable.

`p.StructuralFeature`.

From ECO types to .NET types

Another interesting route you can take is from the ECO types (UmlRt) to the .NET types. If you are navigating through the UML model and want to instantiate a particular type, access the ObjectType property and invoke the constructor. This, in effect, creates an underlying ECO object where the ObjectType is a generated class (wrapper for ECO Object), complete with a constructor that takes care of the details.

The role of ECO tagged values

Some of you familiar with the UML specification might wonder how EcoServices can be configured. For example, because we now have an IPersistenceService, surely ECO will give us the ability to fine-tune the SQL schema for storing objects. But UML, although richer than any other language, wants to remain technology agnostic? If this were true, it would not make sense for the UML standard to include an attribute on ModelElements for Table names and Column names. The solution is to keep your UML model clean of such platform-specific elements and use UML profiles (collection of tagged values) to annotate any model element with further information about how to execute the platform-independent model (PIM) onto a particular platform, thus how to transform it into a platform-specific model (PSM).

There again, note that `Borland.Eco.UmlRt` is an optimized UML structure. Accessing the generic tagged values associated with model elements repeatedly at runtime could be costly. So, although `UmlRt.IModelElement` has a TaggedValues property that tries to map as closely as possible to the UML standard, the standard UML metaclasses are augmented with ECO versions in which tagged values are fully blown attributes. An example is the `IEcoClass.Versioned` attribute.

Modeling our sample application: Introduction to Consultancy Manager 1.0

Now that you have read about basic theories and concepts introduced by the ECO platform, the rest of this white paper focuses on useful techniques through a practical example application named Consultancy Manager.

Consultancy Manager is designed to help any organization that provides products and services, especially consulting services. The basic features are that of contact and address management, time recording and billing, and basic statistical reporting for management. **Figure 3** shows a package interdependency diagram of the classes.

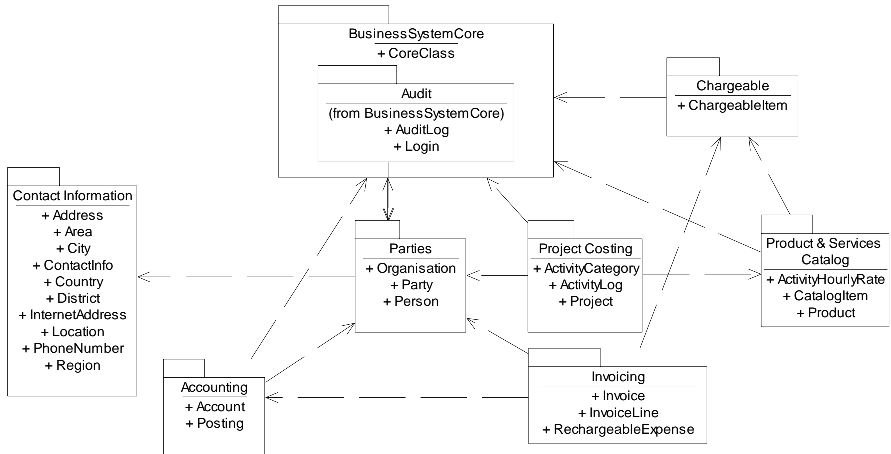

Figure 3: *Consultancy Manager, a high-level map*

The steps involved in building the application:

1. Use the Borland® Together® modeling surface to visually model classes and set up UML and ECO properties on modeling elements
2. Implement method implementations in Delphi
3. Build an ECO Space to host classes and set up the persistence mechanism

4. Build a user interface

Figure 4 is a screen shot of the finished Consultancy Manager application.

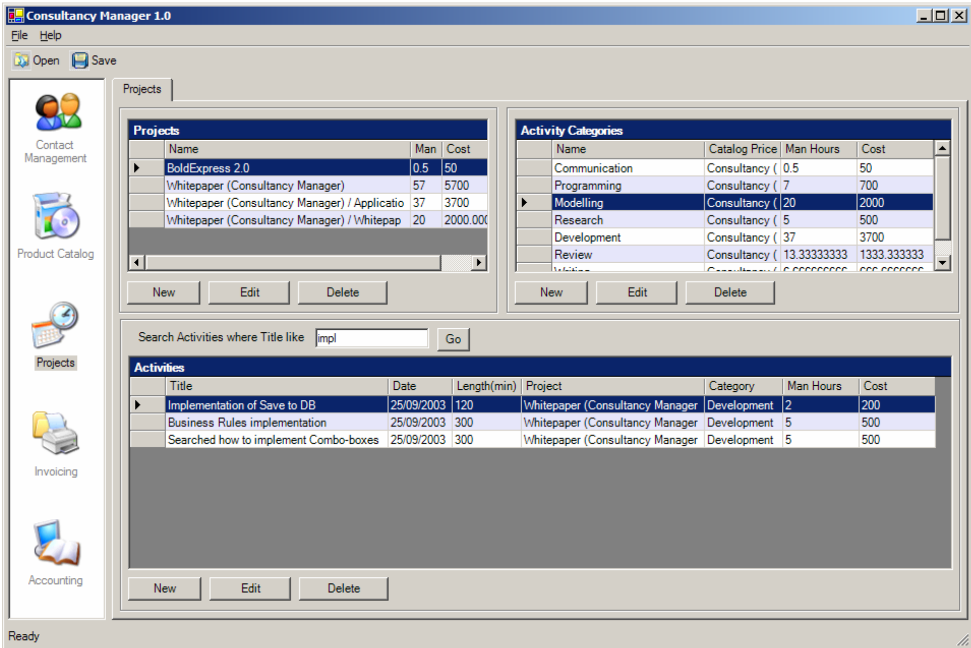

Figure 4: *Consultancy Manager's projects page*

Core classes

Every corporate software development team can benefit from building a reusable core set of classes to be used in all applications. As more horizontal features are required from applications to conform to corporate development guidelines, developers can retrofit these reusable classes easily. For example, it is not uncommon to want to log every change made in a system. If circumstances are not so stringent, being able to link the last version of an object to a user can prove useful to colleagues.

Figure 5 shows that all classes inherit from an ultimate ancestor class that we have named `CoreClass`.[3] Every object in our system is time stamped at its creation, and subsequent modifications are associated with a `Login` object representing one user of the system.

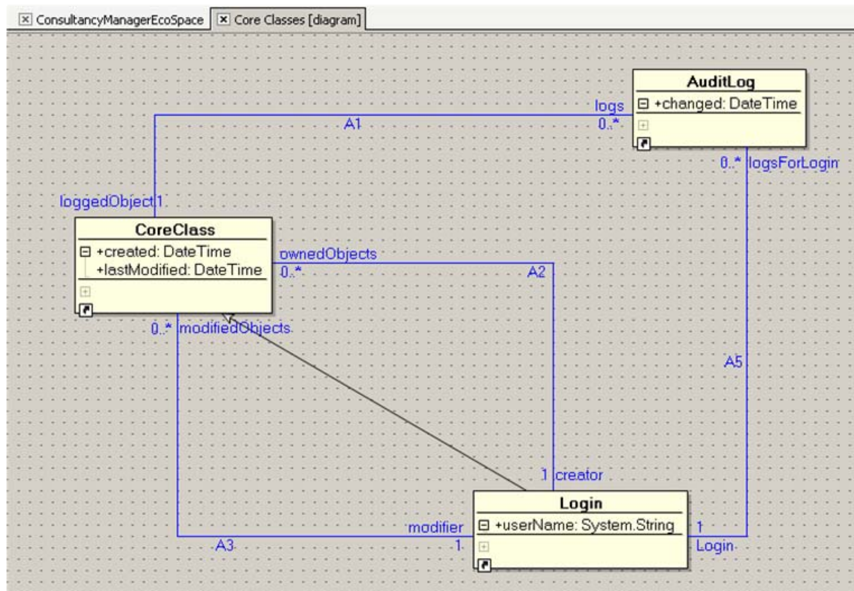

Figure 5: *The classes diagram introducing horizontal features of Audit*

Contact management

Nearly every business system needs contact management capabilities. **Figures 6** and **7** show that `Party` is the abstract ancestor of `Organization` and `Person`. They have in common their ability to be associated with various `ContactInfo` objects.

[3] Please note that the use of word *Core* here makes *no* reference to Core in Borland® Enterprise Core Objects (ECO).

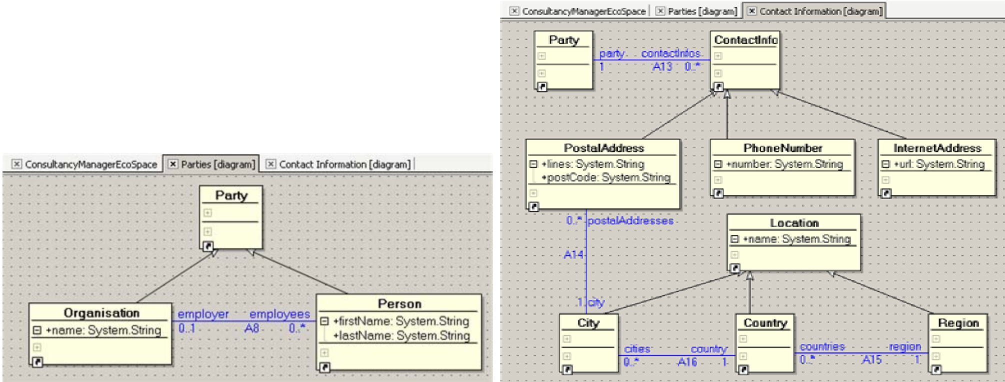

Figure 6: *Contact management diagrams with parties, addresses, and phone numbers*

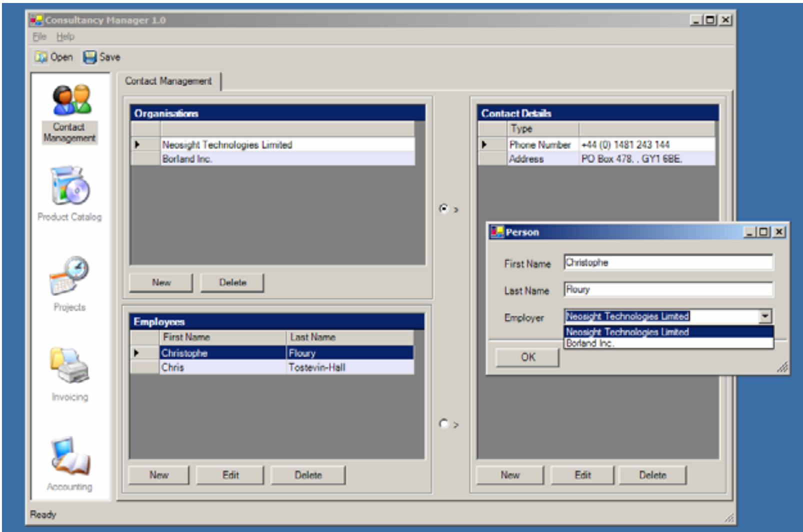

Figure 7: *The contact management page shows instances of UML classes*

Product and services

This subset of functionality allows us to capture the consulting organization's product and services catalog. **Figure 8** shows two products and one hourly rate for consultancy. Note that the audit functionality is available by inheriting from the `CoreClass` described earlier.

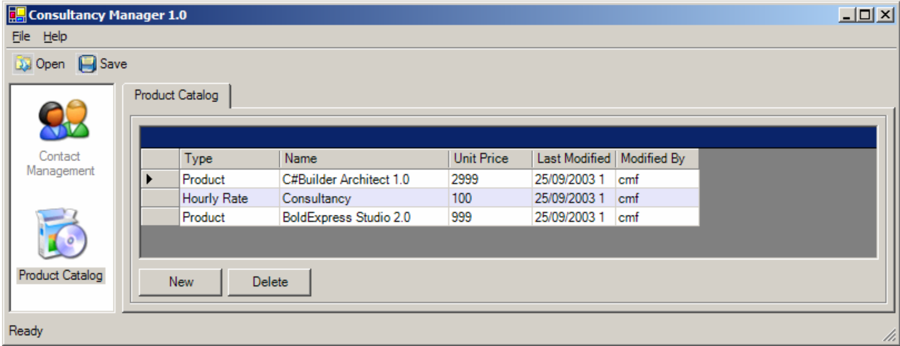

Figure 8: *The product catalog showing the core audit functionality*

The *Product and Services Catalog* diagram (**Figure 9)** introduces intermediate ancestors for `Product` and `ActivityHourlyRate`. `ChargeableItem` introduces the derived `cost` attribute to its descendent classes. Both `Product` and `ActivityHourlyRate` have their `DefaultStringRepresentation` ECO Tagged Value set. **Figure 9** shows some aspects of parameterizing the ECO database schema generation capabilities.

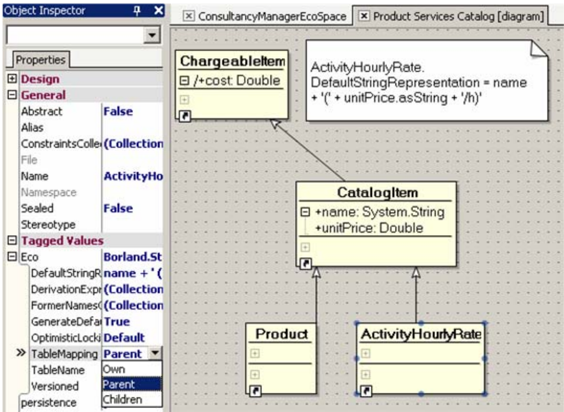

Figure 9: *Providing an OCL-based string representation and changing TableMapping*

Using the DefaultStringRepresentation ECO tagged value

In .NET, all objects can provide a string to display an object's value.[4] This is achieved by the presence of the `System.Object.ToString()` virtual method that you can override in your classes. ECO—or, more precisely, Bold for Delphi—had this ability long before with its *StringRepresentation*. In fact, Bold can have several *StringRepresentations*, with one being called *Default*. In ECO, you could add the ToString method to your model to provide the code to calculate it.

With ECO, OCL can do this automatically for you. Provide a valid OCL expression into the *DefaultStringRepresentation* ECO tagged value in your model (**Figure 8**). The two major advantages of using OCL are that your calculations (or each intermediate step of your calculation) are subscribed, so if one element changes, you will be notified. The second advantage is that you need not check for null references each time you follow a node, as you do in code. For example, the `City.ToString` would have to be written:

```
function City.ToString: string;
begin
  Result := name;
  if Assigned(country) then
    Result := Result + ‘ (’ + country.name + ‘)’;
end;
```

Whereas, in the following. OCL is safe : `name + ‘ ( ‘ + country.name + ‘)’`

For those unfamiliar with the power of OCL, note that constructs such as the following are possible:

```
Country.StringRepresentation
name + if country->notEmpty then ‘ ( ‘ + country.name + ‘)’ else ‘’
endif
```

[4] This is not to be confused with .NET serialization, despite the fact that the `ToString()` often is used for serialization of individual objects' members.

Controlling aspects of the database schema

Figure 10 shows that we can not only choose the table and column names in which our data is stored, but also we can specify how we want inheritance to be represented in relational databases. The default scheme of ECO ("*Own*") is to create a new table for each Class containing columns for the attributes it introduces. An Object's members will therefore be split across several tables. Each object in ECO has an ID, which is used as primary key for each table. ECO performs a join on these primary keys to bring all the columns (attributes) together into one view.

You can change this behavior by changing the `TableMapping` Tagged value (**Figure 10**). The "*Parent*" option instructs ECO to store the class attributes in the table used by the parent class. This means that `ActivityHourlyRate` does not have its own table, as shown in the schema extract on the right.

The "*Children*" option could have been used for our audit attributes on `CoreClass`. A `creator, modifier, created, and lastmodified` column would be present in all our tables!

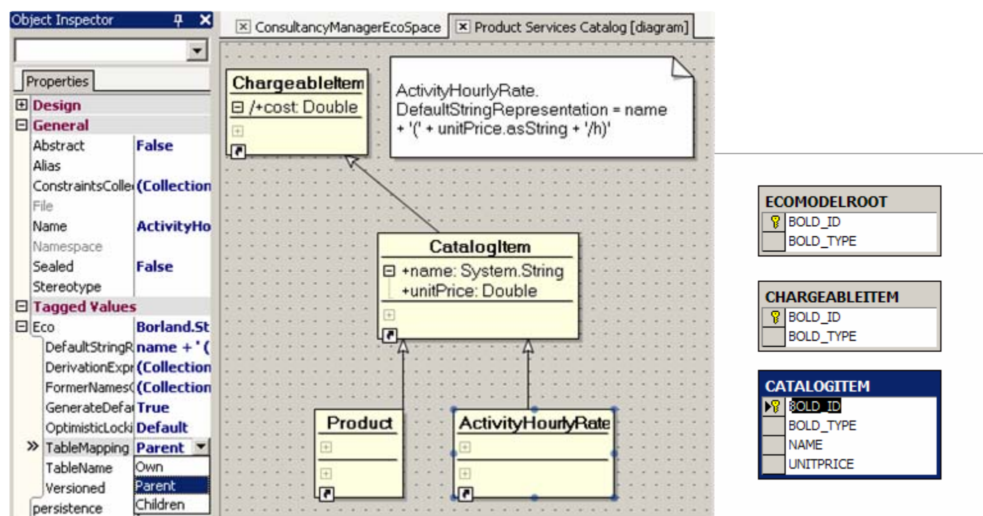

Figure 10: *Customizing the ECO scheme to represent inheritance within relational databases*

Project management

Requirements

In the Project Management module, we want to create hierarchies of projects and sub-projects, and we want to record the time that a certain activity took as well as the human resources that were involved. Finally, we want to categorize activities. Some activities will be assigned the “Development” category; others will be pure “Communication.” The ability to organize categories into hierarchies allows us to group activities at an arbitrary level of precision. For example, we can further divide development into programming and testing.

The ultimate use for this module, apart from feeding information into the not-yet-implemented invoicing module, is management reporting. We want to sum up all activities, times, and costs by projects and subprojects as well as by categories (**Figure 11**).

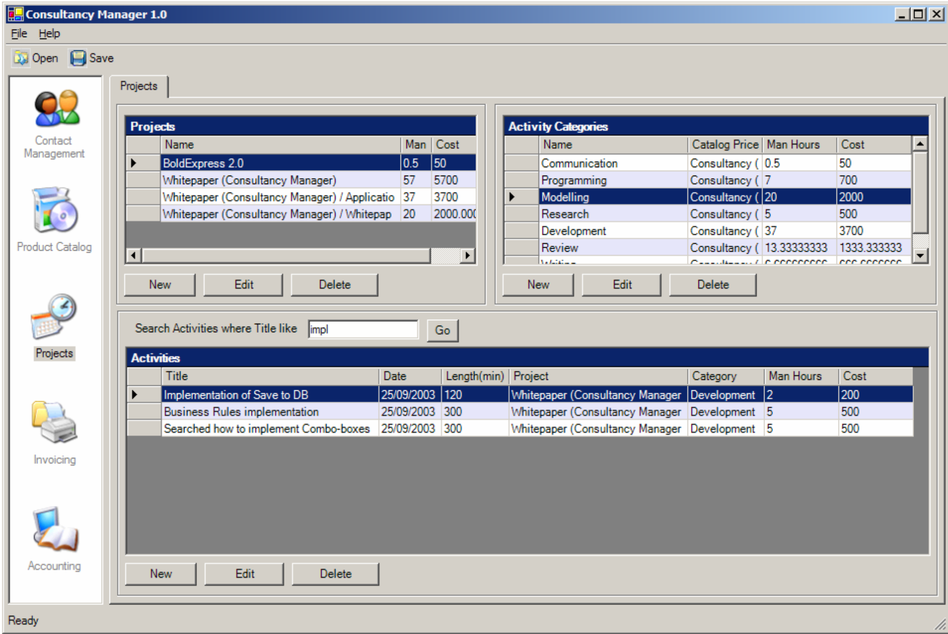

Figure 11: *The project management page of Consultancy Manager 1.0*

Model overview

Figure 12 shows the Project Management module functionality where the hierarchical structures are modeled with self-referencing associations and the calculations are modeled exclusively with OCL expressions at the attribute and class level.

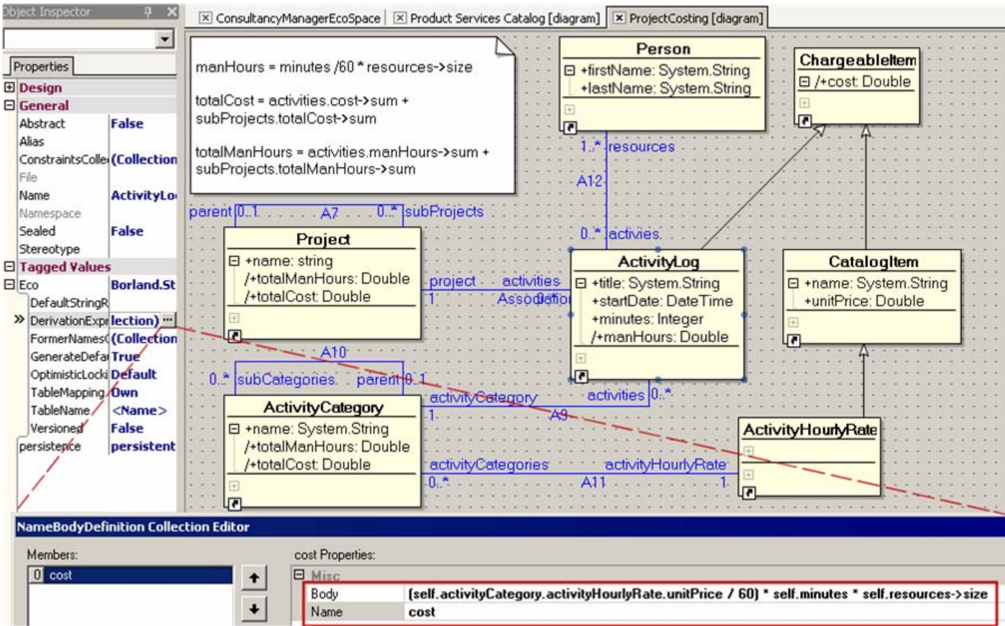

Figure 12: *Exclusive use of OCL derivation to perform complex calculations*

Using derived attributes with inheritance structures

When introducing a derived attribute to a class, you can provide its implementation with a Delphi code property getter or with an OCL expression in the attribute's `DerivationOCL` tagged value (**Figure 13**). Redefinitions in subclasses are done with the `DerivationExpressions` tagged value (**Figure 12**).

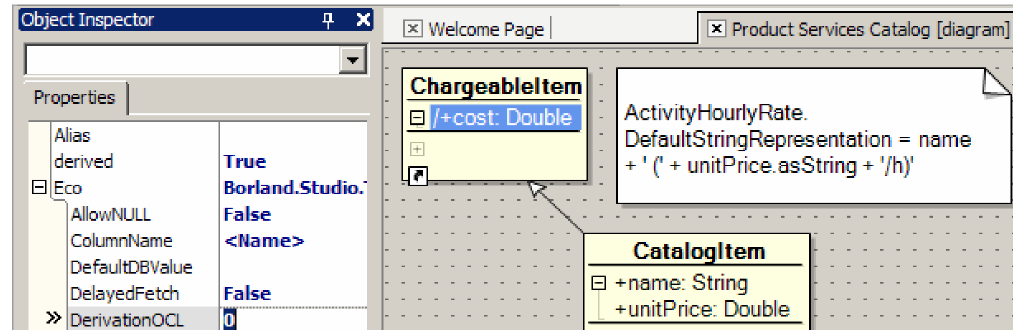

Figure 13: *Providing a derivation expression for an attribute in OCL (constant 0)*

Derived attributes and recursive OCL

Using OCL evaluation in a recursive manner is very powerful. For example, you can show the hierarchical nature of Project in their String representation by providing the following OCL expression, which gives us project names like “Consultancy Manager / Application / User Interface.”

```
Project.DefaultStringRepresentation
if self.parent->isEmpty then name else self.parent.AsString + ' / ' + name
endif
```

Similarly, using the following two simple OCL expressions, you can specify the nature of the calculated attributes for totals and let the power of ECO do the rest, including loading the appropriate objects from the database. We now leave the field of computation for the field of mathematical specification, where OCL has its origins as a formal language.

```
Project.totalCost.DerivationOCL
self.activities.cost->sum + self.subProjects.totalCost->sum
```

```
ActivityLog.DerivationExpressions
cost=(self.activityCategory.activityHourlyRate.unitPrice / 60) * self.minutes
* self.resources->size
```

Considerations regarding the performance of OCL

Of course, all of this power requires care and attention. The ability to merely specify what we want, without saying when things are calculated and in what order, saves time for the developer who knows he will get results; however, it might not always result in an efficient solution.

As the number of `ActivityLog` records grows in our system, you might not want to show the `ManHours` and `Cost` columns for `Project` and `Category` in the main screen. In order to calculate these values, we must load all `ActivityLog` records in memory. This is because *OCL evaluation happens in memory*. We could provide a button on our user interface to show these columns only when we are prepared to wait, as we would do when a complex report is being prepared. Only when we look at ECO elements do they shape themselves up.

Further performance considerations regarding late fetching

So we must load data in memory in order to calculate new information about it. This is nothing new; we are used to doing so—manually. To minimize memory consumption, ECO is heavily geared toward late fetching of objects. It fetches objects only when it needs them. This happens when you programmatically navigate a link with Delphi code like `Person.addresses[0]` or display data using the ECO handles (more on this later).[5] This also happens during the time ECO performs an OCL evaluation if an object is not in memory. The order in which things happen does matter sometimes, when performance must be taken into account. In such a case, ECO might fetch one object at a time, issuing an SQL command for each row instead of retrieving what it needs with a single call.

The solution is not necessarily to avoid OCL, but instead to ensure that objects have been loaded before the evaluation kicks in. ECO provides several methods in its API to help you achieve this. For example:

[5] ECO handles use OCL evaluation.

```
ObjectRepresentation.IElementCollection.EnsureRange(fromIndex: Int32;
toIndex: Int32)
```

loads several elements in a collection in one Database call. Similarly,

```
IObjectList.EnsureRelatedObjects(memberName: string)
```

helps you efficiently pre-fetch a list of objects at the other end of an association.

Converting OCL into SQL

One use of OCL is definitely not recommended: Using the `allInstances` operation to search for particular objects, as in:

```
ActivityLog.allInstances->select(title='Made a very slow application')
```

This effectively loads *every instance* of `ActivityLog` in memory to retain the matching ones in the collection. However, because of a technology dubbed *OCL2SQL*, ECO can translate this OCL expression, and quite a few others like it,[6] into an efficient SQL statement to retrieve only the matching objects.

System parameters

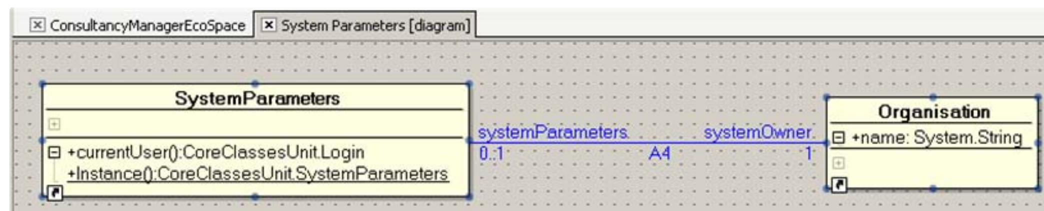

Figure 14: *The SystemParameters singleton*

[6]A limitation of OCL2SQL is the requirement that you mention only persistent elements.

A recurring pattern in all applications is necessary to provide a central location to store system-wide values or parameters. **Figure 14** shows the *System Parameters* diagram in which we have defined a class stereotyped as "singleton." Stereotypes in ECO are not used for execution. The `SystemParameter` class stores only as an application parameter, the `Organization` that owns the software. This is used to differentiate the internal staff (which can be used as resources for `ActivityLog`) from pure contacts.

Implementing a singleton for SystemParameters

The *Singleton* pattern can be implemented in several ways. We implement it with an `Instance` class function to illustrate how to use two important ECO services: the `TypeSystemService` and the `ExtentService`.

```
class function SystemParameters.Instance(serviceProvider:
Borland.Eco.ObjectRepresentation.IEcoServiceProvider): SystemParameters;
var
  TypeSystemService: ITypeSystemService;
  ecoTypeSystem: IEcoTypeSystem;
  c: IClass;
  ex:IExtentService;
begin
  TypeSystemService:=
ITypeSystemService(serviceProvider.GetEcoService(typeof(ITypeSystemService)));

  ecoTypeSystem := TypeSystemService.GetTypeSystem();
  c := IClass(ecoTypeSystem.GetClassifierByType(typeof(SystemParameters)));
  ex := IExtentService(serviceProvider.GetEcoService(typeof(IExtentService)));

  if ex.AllInstances(c).Count = 0 then
    Result := SystemParameters.Create(serviceProvider)
  else
    Result := SystemParameters((ex.AllInstances(c)[0].AsObject));
end;
```

Implementing CurrentUser()

The system parameter singleton pattern also can be used to implement your own services that operate on the ECO Space of the singleton object. For example, we have implemented a `CurrentUser` function to search for a `Login` object, of which the `username` attribute matches the name of the currently logged-in user. As such, we provide a kind of "integrated security mode," where we assume that the user interacting with the software locks his workstation when away from his desk. Logging in to the network is deemed sufficient for our audit purposes. We will show when to call this method later, when we populate the `created` and `modifier` links of our `CoreClass`.

The following implementation for `CurrentUser` illustrates the use of the ECO OclService. We enter the ECO world through `this.AsIObject`. We get hold of the OCL Service by calling its `ServiceProvider.GetEcoService(typeof(IOcl Service))`. We finally search for a `Login` object with the OCL `Login.allInstances->select(username = [Environment.UserName])` For performance considerations, we assume that the number of Login objects will remain almost static and low. If the currently logged-in user is using the application for the first time, we automatically create a `Login` object for him:

```
function SystemParameters.currentUser(): Login;
var
  oclExpression: String;
  oclService: IOclService;
  currentLogin: IElement;
  newLogin: Login;
begin
  oclExpression := 'Login.allInstances->select(username=''' +
     Environment.UserName + ''')->first';

  oclService :=
IOclService(Self.AsIObject.ServiceProvider.GetEcoService(typeof(IOclService)))
;

  currentLogin := oclService.EvaluateAndSubscribe(Self.AsIObject,
oclExpression,nil,nil);
```

```
  if not Assigned(currentLogin) then
  begin
    newLogin := Login.Create(Self.AsIObject.ServiceProvider);
    newLogin.userName := Environment.UserName;
    Result := newLogin;
  end
  else
    Result := Login(currentLogin.AsObject);
end;
```

Building the ECO space

At design time, an ECO Space represents a particular combination of model elements. You use an ECO Space to bring together several packages. At runtime, an ECO Space is a cache of ECO objects. The Delphi 8 wizards automatically create a descendent for your application (**Figure 15**). The ECO Space can be persisted.

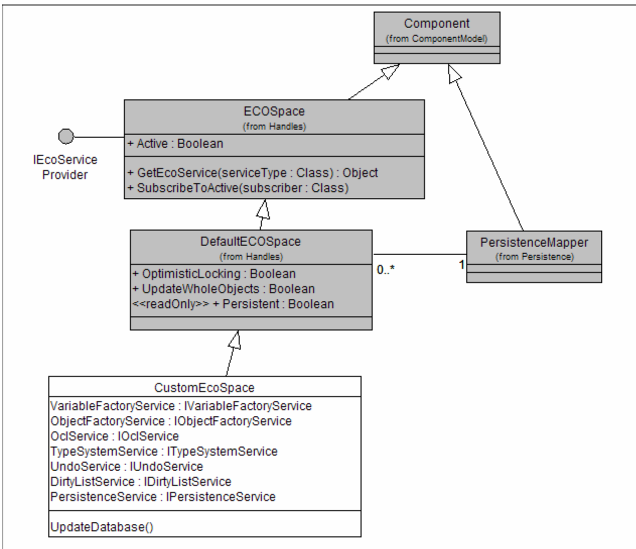

Figure 15: *Delphi 8 wizards generate a descendant of ECO Space for your application*

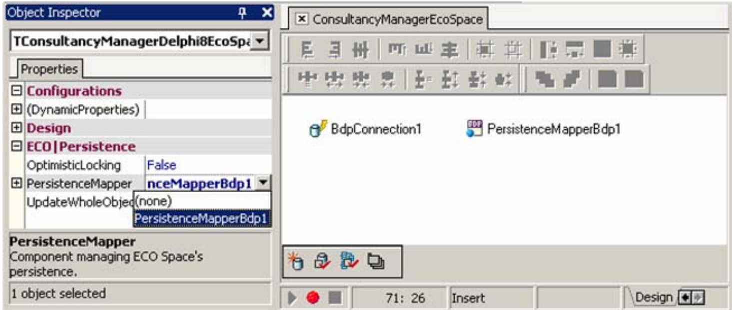

Figure 16: *Setting the persistenceMapper using the ECO Space designer*

The database is in the same directory

```
constructor TConsultancyManagerDelphi8EcoSpace.Create;
var
  dbPath: string;
begin
  inherited Create;
  InitializeComponent;
  dbPath := Environment.CurrentDirectory;
  self.BdpConnection1.ConnectionString :=
'assembly=Borland.Data.Interbase,Version=1.5.1.0,Culture=neutral;vendorclient=
gds32.dll;database=' + dbPath +
'\DATABASE.GDB;provider=Interbase;username=sysdba;password=masterkey';
end;
```

Implementing update of audit attributes

The generated ECO Space is a good placeholder for writing our code that will update audit attributes in a central place. In the `UpdateDatabase()` method, we use the ECO DirtyList Service. This service keeps an up-to-date list of objects that have been modified (or never were saved to the database) since the last update.

```
procedure TConsultancyManagerDelphi8EcoSpace.UpdateDatabase;
var
  dirtyObjects: IObjectList;
  myobject: CoreClass;
  i: integer;
begin
  if Assigned(PersistenceService) and Assigned(DirtyListService) then
  begin
    dirtyObjects := DirtyListService.AllDirtyObjects;
    for i := 0 to dirtyObjects.Count - 1 do
    try
      myObject := CoreClass(dirtyObjects[i].AsObject);
      myObject.lastModified := DateTime.Now;
      myObject.modifier := SystemParameters.Instance(Self).currentUser;
    except
      //
    end;

PersistenceService.UpdateDatabaseWithList(DirtyListService.AllDirtyObjects);
  end;
end;
```

User-interface basics: binding objects instead of data

In this section, we look at building a professional looking WinForm UI for our Consultancy Manager ECO back-end.

Logical three-tier

The greatest strength of ECO is the separation of the three tiers of an application in decoupled layers with clear interfaces. **Figure 17** shows that your model and its objects are hosted within a *middle tier* called the ECO Space. Without any knowledge of the underlying *Persistence tier*, the objects in your ECO Space can be persisted either in files or in relational databases. ECO provides a set of visual components called “handles” to connect the ECO world to the

.NET databinding world. Handles typically point to `IElements` in your ECO Space which can be a single or collection of objects, as well as calculated values. Handles have columns, the values of which are calculated with OCL expressions within the context of the `IElement`. Handles can be chained so that the value (element) of a "roothandle" can provide context for other handles.

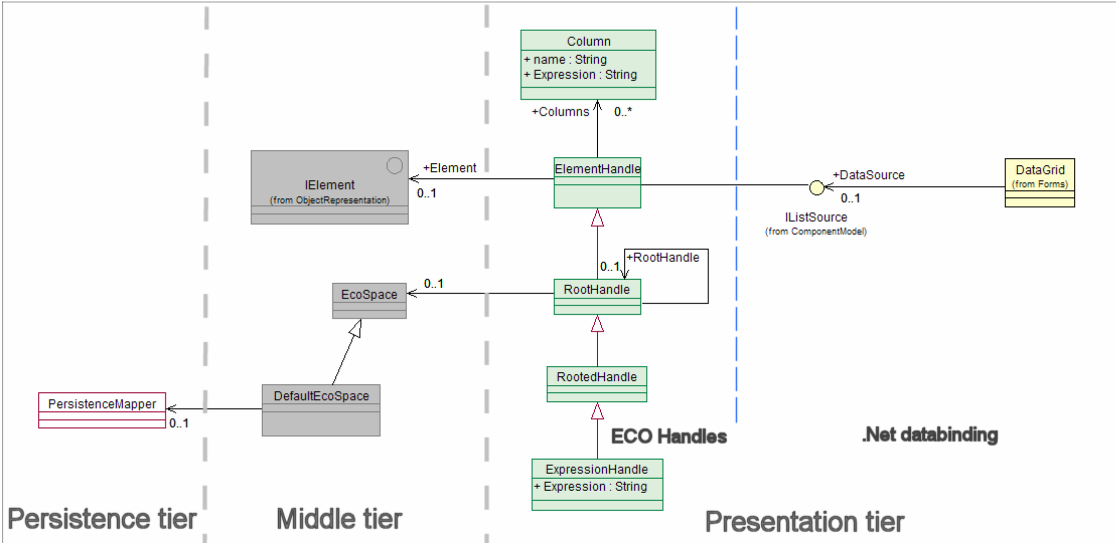

Figure 17: *The three-tier architecture of ECO*

Initializing the ECO Space

ECO wizards generate code for the main form of your application. This code provides an ECO Space singleton as a property of your form and links the ECO Space to a rootHandle component as an entry point for chaining further handles. Note that this happens because chained OCL execution eventually must link to the context of an ECO Space.

The ECO Space constructor that is also code generated by the ECO wizards initialize the ECO Space by possibly linking it to a persistenceMapper component. The main form constructor then sets the ECO Space active property to true.

```
function TMainForm.get_EcoSpace: TConsultancyManagerDelphi8EcoSpace;
begin
  if not Assigned(fEcoSpace) then
  begin
    fEcoSpace := TConsultancyManagerDelphi8EcoSpace.Create;
    rhRoot.EcoSpace := fEcoSpace;
  end;
  result := fEcoSpace;
end;

constructor TMainForm.Create;
begin
  inherited Create;
  // Required for Windows Form Designer support
  //
  InitializeComponent;
  EcoSpace.Active := True;
end;
```

Binding a grid

Binding a grid to data is a simple three-step process (**Figure 18**).

1. Link a rootHandle to the ECO Space
2. Use an rexpression handle rooted to the rootHandle, and set its OCL expression
3. Set the DataSource property of your grid to the ExpressionHandle

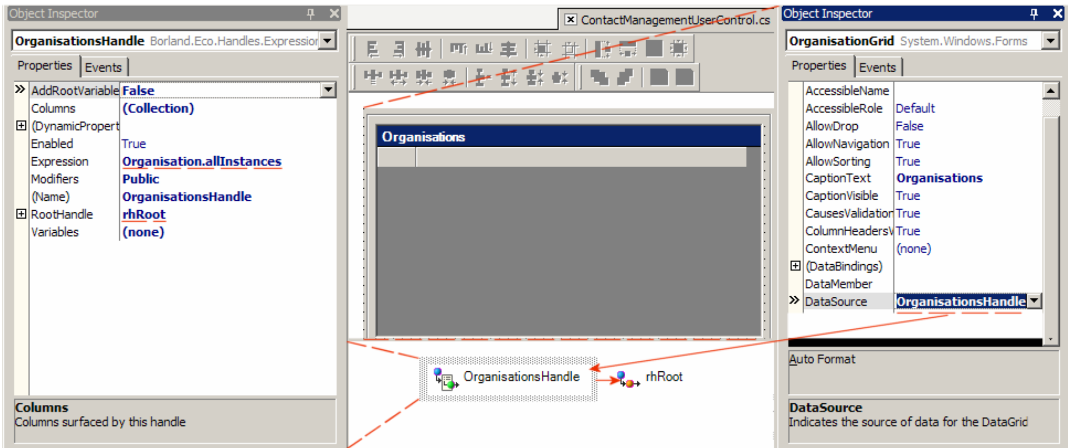

Figure 18: *Binding a WinForm grid*

Note: You do not need to specify a `Columns` collection for your handle. By default, if the `IElement` the handle points to is a collection of `IObjects`, a *MappingName* is defined for each member of the Object (**Figure 19**). Defining new columns for the handle allows you to specify new OCL expressions evaluated in the context of each row.

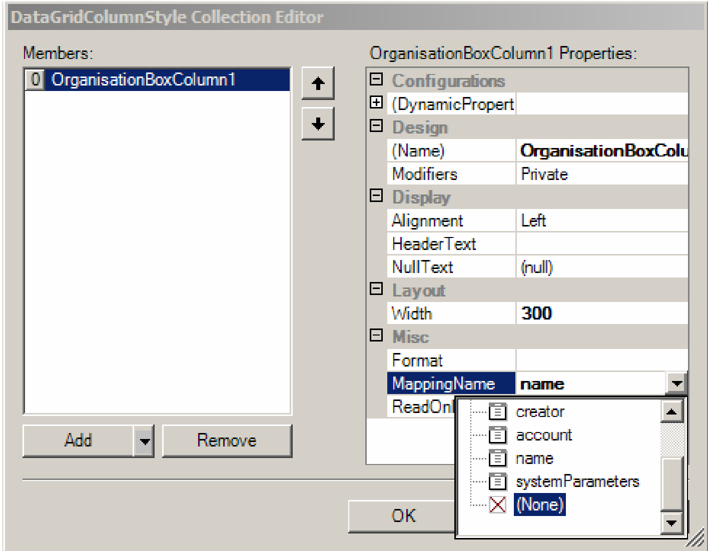

Figure 19: *The default MappingNames are shown in the grid column editor*

Binding combo boxes to objects

A simple example

Databinding with ECO behaves the same way as with database datasources. Combo-boxes are designed to display "friendly" versions of foreign key values in lookup tables. You would traditionally provide to the combo box the *ValueMember*, which would be an integer representing the foreign key column and the *DisplayMember* to display the name of the item chosen in the list.

With ECO, we do not have a foreign key/primary key setup. Instead, objects are linked to one another. To set the link between the objects on selection of an item, we must add a column to the citiesHandle named "self" and an "updateable foreign key" (**Figure 20**).

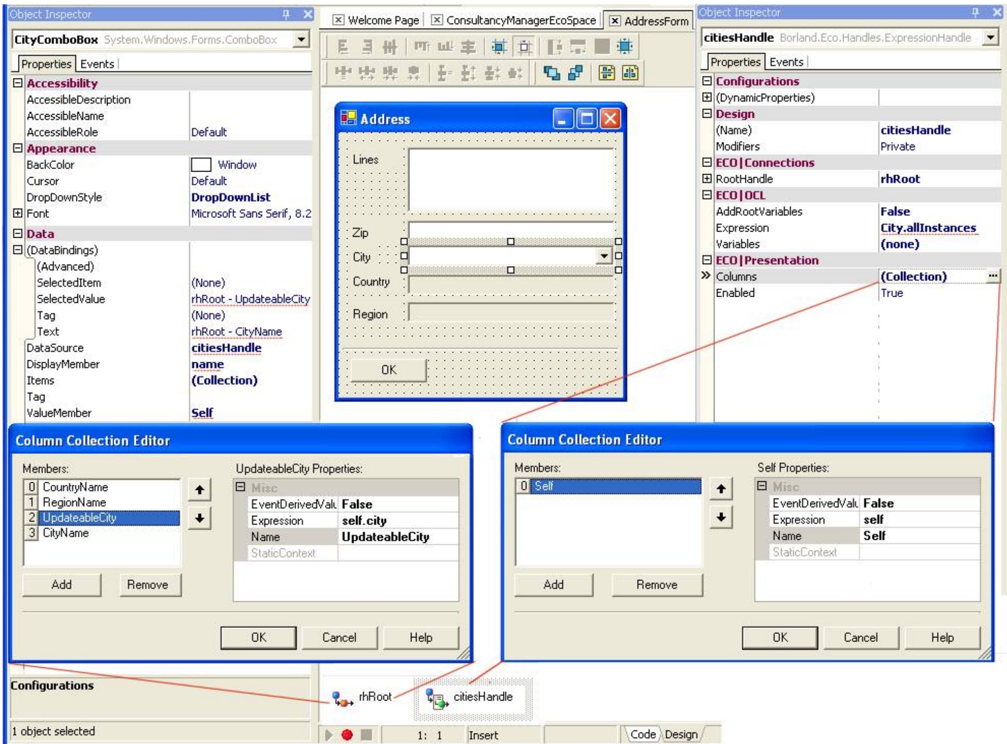

Figure 20: *A simple combo box databinding example*

An advanced example

We might want to unlink a previously selected item using the combo box **(Figure 21)**:

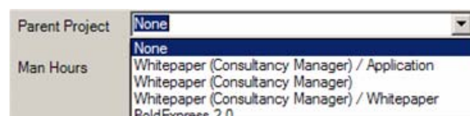

Figure 21: *A combo box allowing us to unlink an object with the "None" option*

The solution (**Figure 22**) uses the power of OCL to create a list of strings representing the list of projects to which the "None" string has been pre-appended. The event handler uses a second "normal" `ProjectsHandle` to locate the appropriate object to link. Use the `IObjectContainer.Clear()` method when the "None" value is selected.

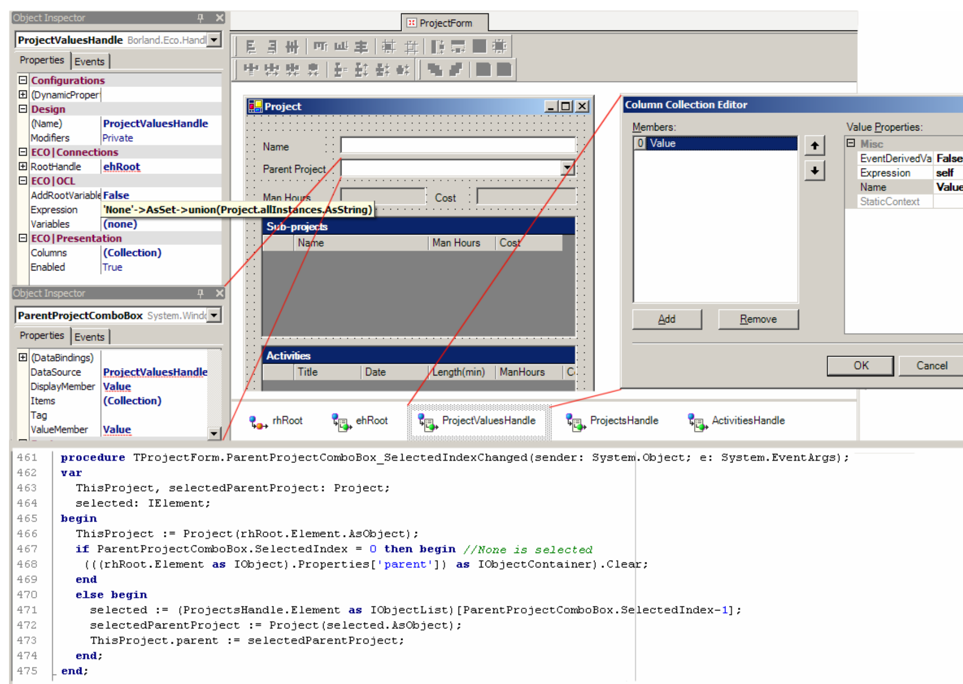

Figure 22: *An advanced combo box databinding example*

Using ECOWinforms and the AutoContainer

ECO wizards automatically generate an "ECOWinform" for your project. An ECOWinform is a WinForm with a constructor that accepts an ECO Space as a parameter. This way, the handles on the form can be connected to the ECO Space. In fact, the ECOWinform has a default rootHandle.

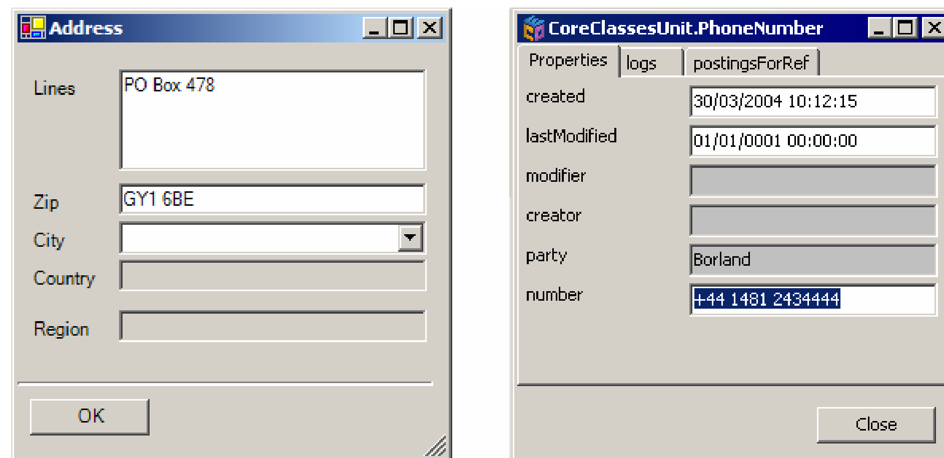

Figure 23: *An ECOWinform and the AutoContainer*

Figure 23 shows what happens when the code below is exercised. For Address objects, we create an instance AddressForm, the constructor of which has been enhanced to accept a PostalAddress as well as an ECO Space. For all other subclasses of ContactInfo, instead of designing a custom form, we have decided to showcase the ECO AutoContainer. The AutoContainer Service dynamically creates WinForms from the UML model.

```
procedure TUserControl.ShowContactInfo(c: ContactInfo);
var
  af: TAddressForm;
  autocontainer : IAutoContainer;
begin
  if c is PostalAddress then
  begin
    af := TAddressForm.create(c as PostalAddress,EcoSpace);
    af.showDialog;
  end
  else
  begin
    autoContainer :=
AutoContainerService.Instance.CreateContainer(EcoSpace,c.AsIObject);
    Form(autoContainer).ShowDialog;
  end;
end;
```

Using the ubiquity of databinding in .NET

The last example (**Figure 24**) illustrates how .NET databinding can be used for driving most properties, including the title of forms. In this example, we show how to make the form title `Activity` include the title attribute of the object it displays.

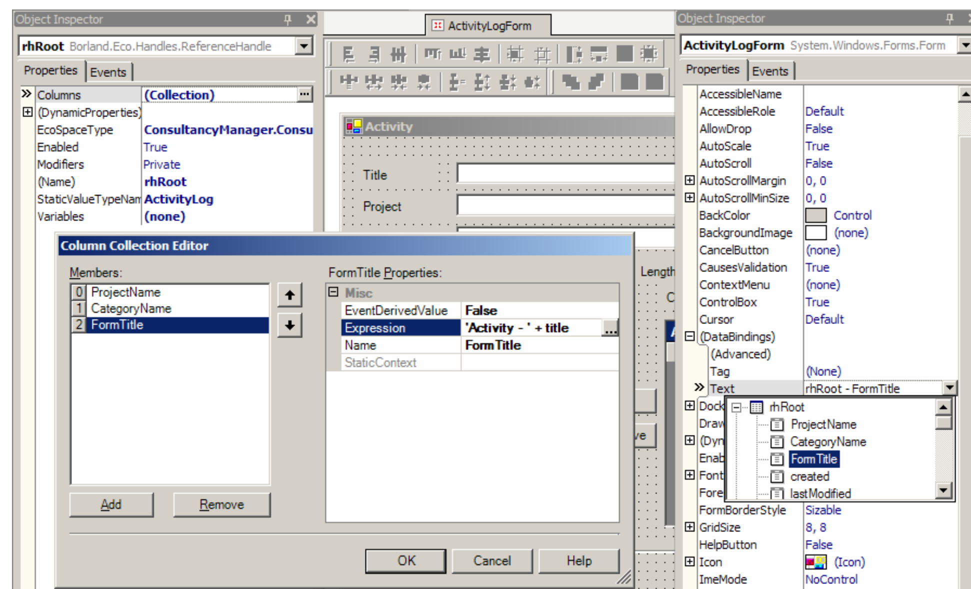

Figure 24: *Using advanced .NET bindings to calculate the title of a WinForm*

Building an optimized search

We mentioned that ECO can evaluate OCL expressions within the *Persistence System* (database) by converting OCL to an SQL expression using the technology known as OCL2SQL. In **Figure 25**, we show how to use the special *oclPSHandle,* which behaves similarly, except that its `Execute()` method must be used to trigger the "evaluation." Another notable difference is that the result is not subscribed.

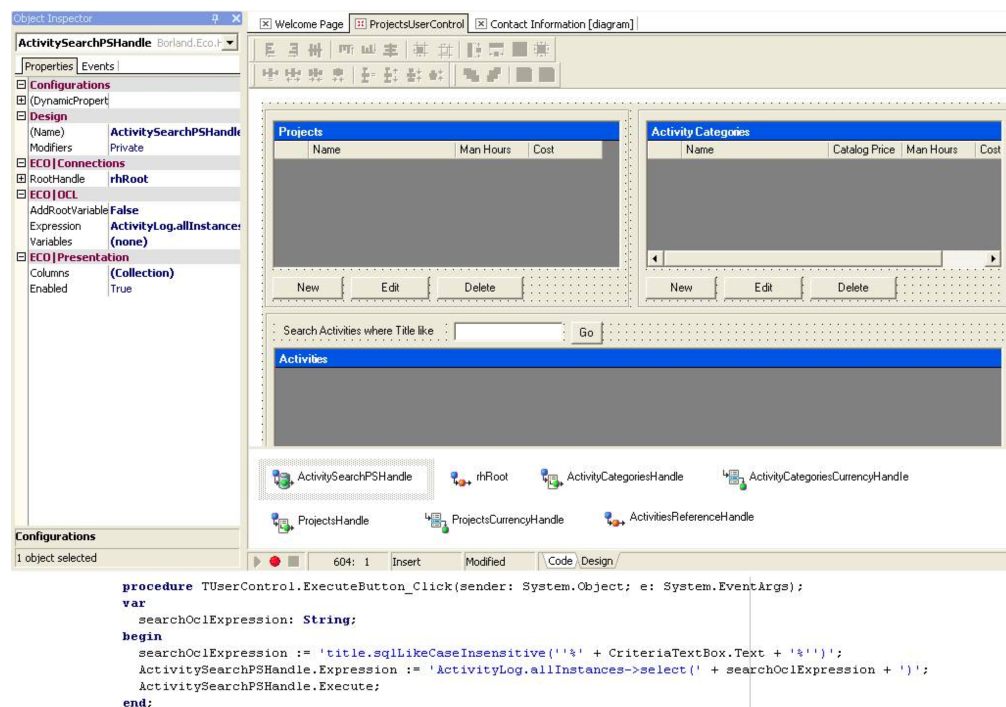

Figure 25: *Using the oclPSHandle to transform OCL into SQL for fast searches*

The code for the event handler above builds the OCL expression from the criteria the user typed and calls `Execute()`. Note that the *sqlLikeCaseInsenstive* OCL operation is not related to our executing SQL; it is an OCL operation introduced by Borland to the OCL specification for practical reasons. Indeed, it also can be used within the ECO space.

Improving the search screen

The main problem with our current search implementation is that it will find only objects that are in the database. Any new `ActivityLog` object that has yet to be persisted will not be listed in our grid. This gives us the opportunity to investigate the ECO `DirtyListService`, which maintains a list of objects that are different from what is in the database. This list includes new objects not yet persisted.

The code below starts exactly like our previous version by Executing the *oclPSHandle*. It then gets hold of the `IDirtyListService.AllDirtyObjects()` list. To show how to retrieve the result of OCL execution returning primitive types, we programmatically navigate through the list of objects and perform the "`title.sqlLikeCaseInsensitive()`" test in their individual contexts. We add matching objects to the oclPSHandle.Element (which is a list):

```
procedure TUserControl.ExecuteButton_Click(sender: System.Object; e:
System.EventArgs);

var

  searchOclExpression: String;

  DirtyListService: IDirtyListService;

  AllDirtyObjects, list: IObjectList;

  oclService: IOclService;

  i: integer;

  dirtyObject: IObject;

  Result: IElement;

begin

  ActivitiesGrid.DataSource := nil;
```

```
  searchOclExpression := 'title.sqlLikeCaseInsensitive(''%' +
CriteriaTextBox.Text + '%'')';

  ActivitySearchPSHandle.Expression := 'ActivityLog.allInstances->select(' +
searchOclExpression + ')';

  ActivitySearchPSHandle.Execute;



  //Let's also add objects not yet in the database (i.e Dirty objects)

  DirtyListService :=
IDirtyListService(EcoSpace.GetEcoService(typeof(IDirtyListService)));

  if DirtyListService.HasDirtyObjects then

  begin

    AllDirtyObjects := DirtyListService.AllDirtyObjects;

    oclService := IOclService(EcoSpace.GetEcoService(typeof(IOclService)));

    list := IObjectList(ActivitySearchPSHandle.Element);

    for i:=0 to AllDirtyObjects.Count -1 do

    begin

      dirtyObject := AllDirtyObjects[i];

      if (dirtyObject.AsObject is ActivityLog) then

      begin

        Result :=
oclService.EvaluateAndSubscribe(dirtyObject,searchOclExpression,nil,nil);

        if boolean(Result.AsObject) then

          list.Add(dirtyObject);
```

```
      end;
    end;
  end;
  ActivitiesGrid.DataSource := ActivitySearchPSHandle;
end;
```

Saving data and multi-user considerations

ECO development is similar to cached dataset regarding synchronizing the object cache with the database. When you call EcoSpace.UpdateDatabase(), ECO can perform a check to see whether the values in the database have changed since it read them, to ensure that other people's changes are not overridden. This feature, called optimistic locking, can be set to on or off.

Schemes for updating the database

Once you have committed your changes, you have several choices, depending on the kind of application you build. You can clear all the objects in your cache by setting the Active property of the ECO Space to false and true again and continue using the portion of the database loaded in memory without needing to reread (unload) objects if you know that a user will affect only his area of the system.

Otherwise, you can devise a mechanism that determines, on a regular basis, which objects have changed in the database (or on user pressing F5) and unload them. If ECO needs them for calculations, don't worry; it quickly reloads the new version!

Other ECO services

In our sample application, we have used many ECO services. We evaluated OCL expressions with the OCL service, found the number of instances of a particular class for our implementation of the singleton pattern using the Extent Service, used the Type Service, used the DirtyList Service to locate new objects in memory that matched a certain criteria, and used the Persistence Service to save our changes to a database backend.

Although we have explored several features, we haven't used all of them yet—ECO is truly feature rich. Let's see what we have missed:

The Object Factory Service

Create objects through the UML type system rather than by using the constructors on the generated code.

The OCL Type Services

Similar to the OCL Service, this interface service returns the type of the expression to "expect" instead of calculating the result.

The State Service

DirtyList Service helped us find the list of dirty objects. The State Service determines which individual properties have changed**.**

The Undo Service

This comprehensive service allows you to implement undo and redo functionality.

The Version Service

Objects that are versioned can have one or more historical versions that are read-only and one current version that is live and can be updated.

Conclusion

Our first look at the new Enterprise Core Objects technology from Borland— now shipping with Borland Delphi 8 for the Microsoft .NET Framework, Architect Edition—included the theory behind the Model-driven architecture (MDA) and how UML can drive the development cycle. We explained the respective roles of the Together modeling technology and UML runtime ECO technology. We modeled a real-life application and followed it through to deployment, while considering audit requirements, performance, reusability, and usability with a rich WinForm User Interface.

We have seen how transparently ECO objects augment the capabilities of the standard .NET objects while looking like .NET objects and unleashing many .NET features, such as databindings. We have seen the power of expression of OCL compared with OQL and SQL.

Overall, we have seen the incredible productivity enhancements that can be achieved with not a single line of SQL crafted to implement complex database systems. Borland Enterprise Core Objects (ECO) truly increases the level of abstraction and gives us the opportunity to deliver powerful systems without leaving the world of objects while interchangeably targeting any of today's RDMS.

Made in Borland ® Copyright © 2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Microsoft, Windows, and other Microsoft product names are trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • www.borland.com • Offices in: Australia, Brazil, Canada, China, Czech Republic, Finland, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, Mexico, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. • 22241