Eco 3 Services

Table of Contents

Index

a

1 An Introduction to ECO Services

ECO Services provide the developer with the ability to perform a standard set of operations against objects, collections, and the persistence storage.

The ECO framework has been designed in such a way that business logic and framework logic are kept as separate as possible. For example, examining the generated source code for an ECO class will not reveal methods such as “Delete” or “Refresh”, as you would expect to find on a traditional dataset component.

Keeping framework methods out of our business classes is an important step towards making our source code more readable, and manageable. Having a clear, and almost invisible, separation means that when we inspect the source code of our business classes, we only see methods relating to the logical functioning of the class in question. This clearly makes our source code easier to understand, refactor, and debug.

2 The ECO service provider

The key to ECO services is the EcoSpace or, more accurately, the EcoSpace's implementation of the IEcoServiceProvider interface. The IEcoServiceProvider has only one method, named "GetEcoService". GetEcoService accepts a single parameter identifying the type of the service we want to retrieve, and returns an object which we must then typecast to the correct type.

[Delphi]

```
var                                                                    1
DirtyListService: IDirtyListService;

begin                                                                  2
DirtyListService :=
EcoSpace.GetEcoService(typeof(IDirtyListService))
as IDirtyListService;
end;
```

[C#]

```
IDirtyListService dirtyListService;                                    1

dirtyListService = (IDirtyListService)                                 2
EcoSpace.GetEcoService(typeof(IDirtyListService))
```

1. First we declare a variable of the correct type that we will use to hold a reference to the service returned to us.
2. Using the GetEcoService method of the EcoSpace we ask for the IDirtyListService by passing typeof(IDirtyListService). GetEcoService returns an instance of type System.Object, so the returned value then needs to be typecast to the correct type.

An alternative method of retrieving an ECO service is to use one of the static methods of the new Borland.Eco.Services.EcoServiceHelper class. This class has one static method for each of the standard ECO services, returning a strongly typed result, saving us from having to typecast the result. For the remaining standard ECO services I will continue to use this new helper class, but also keep in mind the first technique demonstrated, as it will prove to be useful later on in this chapter.

[Delphi]

```
var
DirtyListService: IDirtyListService;                                    1

begin
DirtyListService :=
EcoServiceHelper.GetDirtyListService(EcoSpace);                         2
end;
```

[C#]

```
IDirtyListService dirtyListService;                                     1

dirtyListService =
EcoServiceHelper.GetDirtyListService(EcoSpace);                         2
```

1. First we declare a variable of the correct type that we will use to hold a reference to the service returned to us.
2. Using the EcoServiceHelper class we can then retrieve a strongly typed reference to the ECO service we require.

Each of the EcoServiceHelper's static methods requires a single parameter of type System.Object. This object may be the EcoSpace itself, or an instance of an ECO class belonging to the EcoSpace. Now that it is clear how to retrieve an instance of an ECO service, it is time to explain each of the standard services in turn.

3 The object factory service

Using the IObjectFactoryService interface, the object factory service provides the developer with an alternative way of creating an instance of a modelled class. Ordinarily a new instance of a modelled class is created like so

[Delphi]

```
var
    NewPerson: Person;
begin
    NewPerson := Person.Create(EcoSpace);
end;
```

[C#]

```
Person newPerson;
newPerson = new Person(EcoSpace);
```

Creating a new instance is so simple that it may at first seem unnecessary to have a service for the purpose of creating object instances, however, the object factory service makes life much easier when the type of the object is not known until runtime, or is determined by reading model information.

3.1 CreateNewObject(type)

I will first explain the method that is the most similar to the source code illustrated above. That is, I will demonstrate how to create an instance of a specific type "Person".

[Delphi]

```
//1
var
NewPerson: Person;
NewObjectInstance: IObjectInstance;
ObjectFactoryService: IObjectFactoryService;
//2
begin
    ObjectFactoryService :=
        EcoServiceHelper.GetObjectFactoryService(EcoSpace);
    //3
    NewObjectInstance :=
    ObjectFactoryService.CreateNewObject(typeof(Person));
    //4
    NewPerson := NewObjectInstance.AsObject as Person;
end;
```

[C#]

```
//1
Person newPerson;
IObjectInstance newObjectInstance;
IObjectFactoryService objectFactoryService;
//2
objectFactoryService =
    EcoServiceHelper.GetObjectFactoryService(EcoSpace);
//3
newObjectInstance =
    objectFactoryService.CreateNewObject(typeof(Person));
//4
```

```
newPerson = (Person)
    newObjectInstance.AsObject;
```

1. First we declare variables of the correct types.
2. Using the EcoSpaceHelper we obtain a reference to the object factory service.
3. Next we instruct the object factory service to create an instance of our object by passing a .net class type, in this case "Person". ECO will return an IObjectInstance reference.
4. Finally, we switch from the ECO world back over to our business classes by referencing the AsObject property and then casting it to the appropriate business class type.

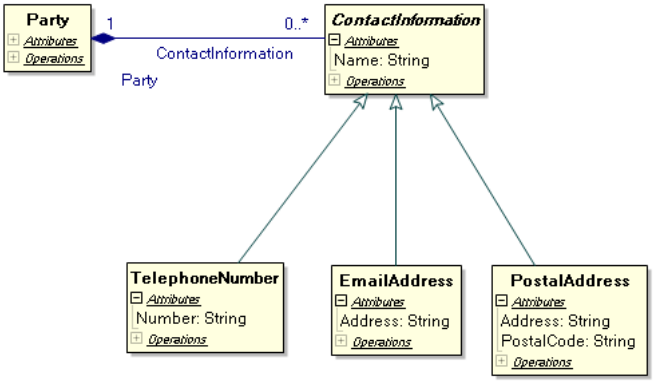
3.2 CreateNewObject(string)

The steps involved in creating a class instance by name are identical to creating an instance by class type. The only difference is that we are now expected to pass the name of the class as a string rather than a .net class type. Passing a type is much less susceptible to human error, as it is checked during compilation rather than at runtime.

3.3 CreateNewObject(IClass)

Again the steps here are almost identical to the previous two techniques. The only difference this time is that the method expects an IClass. The IClass interface will be described in more detail in "The type system service" section of this chapter.

This method is useful when you want to navigate through the model structure at runtime, and then create instances of objects found within it. For example, you may wish to find a complete list of classes in the model that descend from a particular class, and then create an instance of one of those subclasses.



In the model above we would never want to add an instance of "ContactInformation' directly, in fact we cannot create an instance of ContactInformation because it is abstract. Instead our GUI could find all concrete subclasses of ContactInformation and then present a list to the user, finally instructing the object factory to create an instance of the IClass the user selected from the model.

Passing an invalid type to any of the overloaded CreateNewObject methods of IObjectFactoryService will result in an exception being thrown.

4 The dirty list service

Whenever a persistent object is created, modified, or deleted, it is considered to be"Dirty", meaning that it has somehow been "modified" or "altered". This Dirty state is an indication to ECO that the persistence storage needs to be updated in order to reflect the changes made to the object instance in question.

Using the IDirtyListService interface, the dirty list service enables the developer to obtain a list of objects that have been modified in someway, and need to be updated to the persistence storage.

Note that the "Dirty" state is reserved for persistent objects only. Object instances of a class marked as Transient in the model are never saved to the persistence storage, and therefore cannot have such a state.

4.1 HasDirtyObjects()

This method returns a Boolean result. If there are one or more modified object instances in the EcoSpace, this method will return True, otherwise it will return False.

4.2 AllDirtyObjects()

This method returns an IObjectList instance containing an IObject for each dirty object held within the EcoSpace. This list is immutable, meaning that if you try to modify it using Remove() for example, a System.InvalidOperationException will be thrown.

4.3 Subscribe(Borland.Eco.Subscription.ISubscriber)

This method accepts an instance of the ISubscriber interface. Each time the AllDirtyObjects list is altered due to an object being made dirty, or being marked not-dirty due to either an UpdateDatabase call or RollbackTransaction, the Receive() method of the ISubscriber will be executed.

This subscription is useful when the application needs to perform an action on each object modified by the user, for example, updating a GUI element displaying a list of modified objects, or checking the constraints of all modified objects and either enabling or disabling GUI controls respectively (such as a "Save" button).

5 The undo service

Using the IUndoService, the programmer is able to perform in-memory transactions on objects within the EcoSpace. These transactions may be committed or reversed at any point, ensuring that if an operation fails the state of the objects is returned to the last known valid state.

By "valid state" I do not mean that the objects may be invalid in an ECO sense, but invalid from a business logic perspective. To reuse a classic example, if a funds transfer is initiated from bank account "A" to bank account "B", two operations must take place. The balance of account "A" must decrease by the transaction value, and the balance of account "B" must increase by the transaction value. This kind of atomic operation has been available in all good databases for quite some time now, but the ECO undo service allows the same kind of atomic operation to be performed in-memory as well.

The undo service provides two main pieces of functionality. It provides named undo-blocks, changes within the undo block may be reversed or reapplied. Secondly it provides in-memory transaction support, which internally uses the undo-block functionality.

5.1 StartTransaction, RollbackTransaction, and CommitTransaction

This first example will demonstrate how to perform an in-memory transaction on a number of objects within the EcoSpace. The example will transfer a given amount of money from one bank account to another, it will adjust the CurrentBalance of each account, and additionally create a transaction object to record the transfer. If an exception of some kind occurs within the transfer method then all changes will be rolled back, otherwise the in-memory transaction will be committed.

[Delphi]

```
//1
procedure BankAccount.TransferMoneyToAccount(
    DestinationAccount: BankAccount; Amount: Double);
var
NewTransactionRecord: TransactionRecord;
UndoService: IUndoService;
    EcoServiceProvider: IEcoServiceProvider;
//2
begin
    EcoServiceProvider := Self.AsIObject.ServiceProvider;
    UndoService :=
        EcoServiceHelper.GetUndoService(Self);
    UndoService.StartTransaction;
    try
        //3
        NewTransactionRecord:=
            TransactionRecord.Create(EcoServiceProvider);
        NewTransactionRecord.BankAccount := Self;
        NewTransactionRecord.Amount := -Amount;
        NewTransactionRecord.Description :=
            'Transfer to account ' +
            DestinationAccount.AccountNumber;
        //4
        NewTransactionRecord:=
            TransactionRecord.Create(EcoServiceProvider);
        NewTransactionRecord.BankAccount :=
            DestinationAccount;
        NewTransactionRecord.Amount := Amount;
```

```
        NewTransactionRecord.Description :=
            'Transfer from account ' +
            Self.AccountNumber;
        //5
        Self.CurrentBalance := Self.CurrentBalance – Amount;
        DestinationAccount.CurrentBalance :=
            DestinationAccount.CurrentBalance + Amount;
        //6
        UndoService.CommitTransaction;
    //7
    except
        UndoService.RollbackTransaction;
        raise;
    end;
end;
```

[C#]

```
//1
public void TransferMoneyToAccount
    (BankAccount destinationAccount, Double  amount)
{
    TransactionRecord newTransactionRecord;
    IUndoService undoService;
    IEcoServiceProvider ecoServiceProvider;
    //2
    ecoServiceProvider = this.AsIObject().ServiceProvider;
     undoService = EcoServiceHelper.GetUndoService(this);
    undoService.StartTransaction();
    try
    {
        //3
        newTransactionRecord =
            new TransactionRecord (ecoServiceProvider);
        newTransactionRecord.BankAccount = this;
        newTransactionRecord.Amount = -amount;
        newTransactionRecord.Description =
            "Transfer to account " +
            destinationAccount.AccountNumber;
        //4
        newTransactionRecord:=
            TransactionRecord.Create(ecoServiceProvider);
        newTransactionRecord.BankAccount :=
            destinationAccount;
        newTransactionRecord.Amount := amount;
        newTransactionRecord.Description :=
            "Transfer from account " +
            this.AccountNumber;
        //5
        this.CurrentBalance -= amount;
        destinationAccount.CurrentBalance += amount;
        //6
        undoService.CommitTransaction();
    //7
    }
    catch
    {
        undoService.RollbackTransaction();
        throw;
    }
}
```

1. First the method signature is declared, a destination account + amount. Variables are declared to hold references to the undo service, the new transaction objects, and the EcoServiceProvider (for creating new object instances).
2. The EcoServiceProvider and UndoService variables are initialized, and an in-memory transaction is started.
3. A new instance of a TransactionRecord class is created. This instance records the reason that "amount" was deducted from the current bank account.
4. A new instance of a TransactionRecord class is created. This instances records the reson that "amount" was credited to

the destination bank account.

5. The balances of the current bank account, and destination bank account are adjusted appropriately.
6. Presuming all went well, the in-memory transaction changes are kept.
7. If an exception occurs at any point during this process, the changes within the in-memory transaction are rolled back, leaving all touched objects in the state in which they originally started, and deleting (un-creating) the new TransactionRecord instances.

ECO in-memory transactions may also be nested. Inner transactions may be committed, but the changes within them will only be committed to the EcoSpace permanently if all owning transactions are also committed. The following steps should clarify this requirement.

1. StartTransaction()
   - StartTransaction()
   - CommitTransaction()
2. RollbackTransaction()

In this scenario a transaction is started, and then maybe some changes are made to objects held within the EcoSpace. Next another transaction is started, probably as a result of executing another method of a class that requires a transaction. The inner transaction is committed, but the outer transaction is finally rolled back, as a result none of the changes made during this process are applied to the EcoSpace permanently, all object states are reverted back to the point in time before the initial transaction was started.

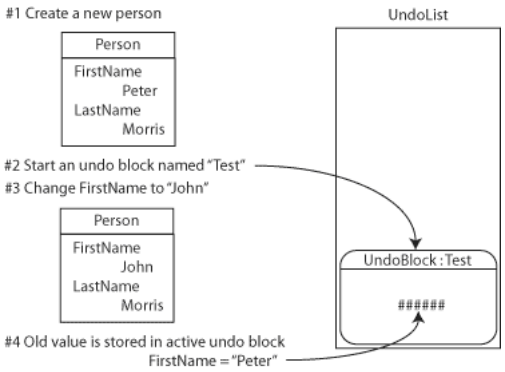
Changes are only applied to the EcoSpace permanently if ultimately all nested transactions are committed. To safeguard this rule is not possible to commit an outer transaction once an inner transaction has been rolled back. The following steps should clarify this requirement.

1. Method A - StartTransaction()
   - Method B - StartTransaction()
     - Method C - StartTransaction()
     - Method C - CommitTransaction()
   - Method B - RollbackTransaction()
2. Method A - CommitTransaction()

In this scenario various transactions are started as various methods of a class are executed, the innermost transaction (started by Method C) is committed. Next the transaction that was started second (by method B) is rolled back. Now that part of the entire transaction has been rolled back it is illogical for the entire transaction to continue, a transaction after all is an entire operation, parts of it cannot fail. As a result the only option left for the outermost transaction (started by Method A) is to roll back, in the example steps the CommitTransaction() call would result in a System.InvalidOperationException.

5.2 Undo blocks

Undo blocks provide a mechanism similar to transactions. Whenever a modification is made to an ECO element (object / attribute / association), ECO will check if there are any undo blocks present in the undo service. If an undo block is found, and the element in question is not already in the topmost undo block, ECO will record the elements original value (or state in the case of objects) in the undo block.



Holding a collection of elements plus their original values/states allows the changes recorded in an undo block to be reversed, restoring the EcoSpace to the exact state it was in at the point the undo block was created.

The undo service may hold multiple undo blocks, only the topmost undo block is considered to be active, therefore changes made within the EcoSpace will always be applied only to the topmost undo block, new undo blocks are always placed at the top of the undo list. This makes it possible to have multiple separate transactions being performed within the EcoSpace at the same time, each with the ability to be independently reversed. This feature is also useful for tracking changes made to objects from within a specific WinForm instance, simply by moving the relevant undo block to the top of the undo service's undo block list, making it the active undo block.

5.3 Creating undo blocks

Undo blocks in ECO are identified using a unique block name. Although it is possible to hold a reference to an undo block using an IUndoBlock, it is only advisable to hold such a reference for a short period of time, only as a local variable for example. The reasoning is quite simple, undo blocks may be removed from the undo service completely (effectively "committed"), accessing the undo block by name will correctly return nil/null, whereas holding onto an IUndoBlock reference would result in your application performing operations on an undo block that is no longer valid.

To ensure that block names are unique, the undo service provides the GetUniqueBlockName method. Executing this method will provide your application with a block name that is guaranteed to be unique.

[Delphi]

```
var
    LoopIndex: Integer;
    UniqueName: string;
    UndoService: IUndoService;
begin
    UndoService := EcoSpace.UndoService;
    for LoopIndex := 1 to 3 do
    begin
        UniqueName := UndoService.GetUniqueBlockName('Test');
        UndoService.StartUndoBlock(UniqueName);
        MessageBox.Show('Unique name is ' + UniqueName);
    end;
end;
```

[C#]

```
IUndoService undoService;
undoService = EcoSpace.UndoService;
for (int loopIndex = 1; loopIndex <= 3; loopIndex++)
{
    string uniqueName = undoService.GetUniqueBlockName("Test");
    undoService.StartUndoBlock(uniqueName);
    MessageBox.Show("Unique name is " + uniqueName);
}
```

In this example the application creates three undo blocks. Rather than hard-coding the block name as "Test", the application asks the undo service to return a unique name using "Test" only as a suggested name. The output of the program, as each iteration of the loop is executed is

1. Unique name is Test
2. Unique name is Test 1
3. Unique name is Test 2

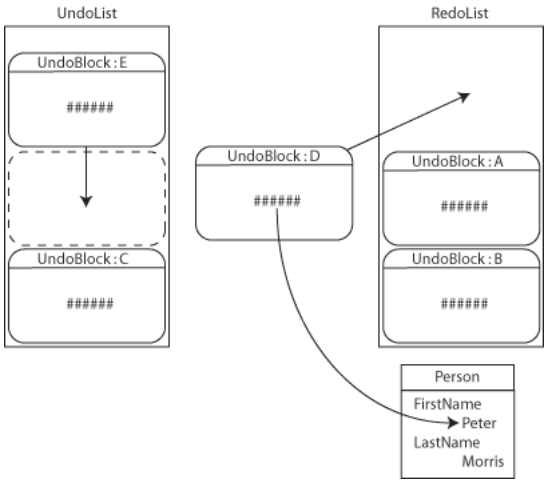
Trying to start an undo block with the same name as an existing undo block will result in a System.InvalidOperationException being thrown.

Note that if the application makes a modification to an object in the EcoSpace, and there are no undo blocks present, ECO will automatically create an undo block named "UnNamed".

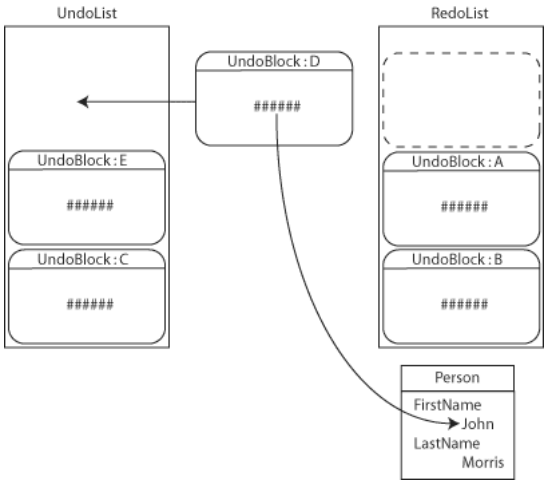
5.4 Working with undo blocks

Once one or more undo blocks have been created, the undo service allows us to interact with these undo blocks via a set of methods.

The UndoBlock() method will reverse all of the changes made whilst the undo block with the specified name was active, a System.ArgumentException will be thrown if an invalid block name is specified. Once the undo block's changes have been reversed, the undo block now becomes the topmost redo block. In actual fact the undo block remains almost exactly the same, except ECO will now also record the current modified value of each element so that we can "redo" the changes if we wish to. The undo block is now removed from UndoService.UndoList and added to the top of the UndoService.RedoList. The topmost block in the UndoList may be reversed using the UndoLatest() method of the undo service.



Once an undo block has been moved into the UndoService.RedoList it is now possible to reapply the changes it contains by calling the UndoService.RedoBlock() method, again a System.ArgumentException will be thrown if an invalid block name is specified. It is possible to undo/redo the changes within an undo block many times over. The topmost block in the RedoList may be reapplied using the RedoLatest() method of the undo service.



5.5 Working with undo/redo lists

The undo service has two list properties, RedoList and UndoList, both of which return an IUndoBlockList. These lists are a kind of stack, building from the bottom upwards, new undo blocks are always added to the top of the stack (the bottom of the stack being index zero). IUndoBlockList contains a number of properties and methods that should require no explanation, these are; RenameBlock(), IndexOf(), Count, Item[string|integer], and TopBlock.

The following methods require a little more explaining.

5.5.1 RemoveBlock(string)

This method will remove the undo block from the list completely. Once the block has been removed it cannot be reinserted, removing the block from both lists effectively identifies the undo block's current status as final (changes applied, or changes reversed).

5.5.2 MergeBlocks(string, string)

MergeBlocks() takes two parameters, the first one is the name of the destination block, the second is the source block. This method will take each of the changes recorded in the source block, and add them to the destination block. Where a conflict exists, because the same element was modified in both undo blocks, the oldest value will take priority. The source block will then be removed from the undo service.

5.5.3 CanMoveBlock(integer, integer)

Undo blocks within an IUndoBlockList may be rearranged. As stated earlier in this chapter, the topmost block of the UndoList is always considered to be the active block. This means that to "activate" an undo block in the undo list we would need to move the block so that it was the topmost block, either using MoveBlock() or MoveToTop().

However, there is a restriction that must be adhered to when reordering blocks in an IUndoBlockList. When the same element (object, association end, attribute) exists in more than one undo block, it is important that the position of these two

undo blocks is never reversed. The reasoning behind this is simply to ensure that changes to an individual element are always undone in a logical sequence, of course it would make no sense if multiple undo blocks were reversed only to result in an element not holding its very original value.

Whenever an undo block is undone / redone it is first moved to the top of its respective list (UndoList / RedoList). The purpose of this move is to ensure that performing the operation does not clash with any other undo blocks, enabling an older modification to an element to be undone before a newer one.

CanMoveBlock() accepts two parameters, the current block index and the proposed new index. If moving the specified block to the new index would violate the above rule then "False" is returned, otherwise CanMoveBlock() will return"True".

5.5.4 MoveBlock(integer, integer)

This method accepts two parameters, the current block index and the proposed new index. First a call is made to CanMoveBlock() to ensure that the move is valid, if the move is valid then the block will be moved to the new location, otherwise a System.InvalidOperationException is thrown.

5.6 The IUndoBlock

The IUndoBlock is a simple interface with only three members; Name, ContainsChanges, and GetChangedObjects(). The method GetChangedObjects() returns an IObjectList, making it possible to determine which objects were modified whilst the undo block was active.

GetChangedObjects() is also useful for updating the persistence storage with the objects affected by this undo block (see "The persistence service"). There are two things to be aware of when doing this

Not only the changes from the current undo block will be saved. All changes will be saved for every object that was modified while this undo block was active

Any other undo blocks that contain changes for an object that is updated via the persistence service will be removed from the undo service

6 The OCL service

Using the IOclService, the OCL service enables the developer to specify an expression in OCL (The object constraint language), which will be evaluated against the EcoSpace's model. The result of the evaluation is then returned to the developer. The OCL service has only three methods, but there are many overloaded alternatives available.

This chapter will not demonstrate every overloaded variation, instead I will demonstrate the more common overloads. This chapter will not cover the object constraint language itself, the content would warrant a book of its own. In fact, there are already a number of books available on this subject.

6.1 Evaluate()

The Evaluate() method allows the developer to ascertain the result of an OCL expression evaluated against the EcoSpace or an IElement. The resulting element is not a "live" element, meaning that its value is not updated when the contents of the EcoSpace are altered.

The following example assumes a model with a single class named "Person". This class has two attributes, "FirstName" and "LastName", both of which are strings.

[Delphi]

```
//1
var
OclQuery: string;
ObjectCount: Integer;
PeterMorris: Person;
JohnMorris: Person;
JohnSmith: Person;
OclResult: IElement;
begin
    //2
    PeterMorris := Person.Create(EcoSpace);
    PeterMorris.FirstName := 'Peter';
    PeterMorris.LastName := 'Morris';
    //3
    JohnMorris := Person.Create(EcoSpace);
    JohnMorris.FirstName := 'John';
    JohnMorris.LastName := 'Morris';
    //4
    JohnSmith := Person.Create(EcoSpace);
    JohnSmith.FirstName := 'John';
    JohnSmith.LastName := 'Smith';
    //5
    OclQuery := 'Person.allInstances' +
        '->select(lastName=''Morris'')->size';
    //6
    OclResult := EcoSpace.OclService.Evaluate(OclQuery);
    //7
    ObjectCount := OclResult.AsObject as Integer;
    MessageBox.Show(ObjectCount.ToString +
        ' people named Morris');
    //8
    PeterMorris.LastName := 'Johnston';
    ObjectCount := OclResult.AsObject as Integer;
    MessageBox.Show(ObjectCount.ToString +
        ' people named Morris');
end;
```

[C#]

```
//1
string oclQuery;
int objectCount;
Person peterMorris;
Person johnMorris;
Person johnSmith;
IElement oclResult;
//2
peterMorris = new Person(EcoSpace);
peterMorris.FirstName = "Peter";
peterMorris.LastName = "Morris";
//3
johnMorris = new Person(EcoSpace);
johnMorris.FirstName = "John";
johnMorris.LastName = "Morris";
//4
johnSmith = new Person(EcoSpace);
johnSmith.FirstName = "John";
johnSmith.LastName = "Smith";
//5
oclQuery = "Person.allInstances" +
    "->select(lastName='Morris')->size";
//6
oclResult = EcoSpace.OclService.Evaluate(oclQuery);
//7
objectCount = (int) oclResult.AsObject;
MessageBox.Show(objectCount.ToString() +
     " people named Morris");
//8
peterMorris.LastName = "Johnston";
objectCount = (int) oclResult.AsObject;
MessageBox.Show(objectCount.ToString() +
     " people named Morris");
```

1. First a number of variables are declared. The variable worth noting here is "OclResult" which is of type IElement.
2. A person is created named "Peter Morris".
3. A person is created named "John Morris".
4. A person is created named "John Smith".
5. An OCL expression is built up, consisting of
    1. Person.AllInstances - returns all Person instances as a collection.
    2. ->select(...) - filters the collection so that it only contains people who's last name is "Morris".
    3. ->size - calculates the size of the collection.
6. The OCL expression is evaluated against the EcoSpace that owns the IOclService instance.
7. The resulting IElement is first coerced to a .net object using its AsObject property, and then typecast to an integer before displaying "2 people named Morris".
8. The last name of Peter Morris is changed to "Johnston". The query is not evaluated again using the OCL service, therefore the final message box reveals that the OclResult element still believes there are "2 people named Morris".

The previous example uses the EcoSpace as the parent context. It is possible to evaluate OCL expressions against any IElement, for example, evaluating "firstName" or "self.FirstName" against a Person would return an IElement from which a string may be retrieved.

[Delphi]

```
OclResult :=
    EcoSpace.OclService.Evaluate(PeterMorris.AsIObject,
    'firstName');
MessageBox.Show(OclResult.AsObject as string);
```

[C#]

```
oclResult =
    EcoSpace.OclService.Evaluate(PeterMorris.AsIObject(),
    'firstName');
MessageBox.Show( (string) OclResult.AsObject );
```

For information on using the IExternalVariableList parameter please refer to the section in this chapter entitled "The variable factory service".

6.2 EvaluateAndSubscribe()

This method is identical to the Evaluate() method, except that it has two additional parameters; a re-evaluate subscriber, and a re-subscribe subscriber, both of type Borland.Eco.Subscription.Subscriber.

This method is intended more for use within component/class development, where the class owning the IElement would pass subscribers in order to ensure it was informed when the element's value needed to be re-evaluated. Working with subscriptions will not be covered in this chapter.

6.3 GetDerivedElement()

Previously in this chapter I demonstrated how the IElement returned by Evaluate() was not a "live" element, meaning that whenever the contents of the EcoSpace are altered the value of the IElement would not be updated.

The GetDerivedElement() method not only evaluates the OCL expression provided, but also internally places the subscriptions required to ensure that the value is current. The return value of the expression is calculated immediately, if any of the elements within the EcoSpace that were read during the evaluation are altered the value held by the IElement is marked internally as out-of-date. When the application attempts to read the value again by calling AsObject, ECO will automatically recalculate the value held by the element. This lazy-calculation approach ensures that minimal CPU time is spent, calculating values only if they are actually requested.

A single line modification to the previous example will result in the following output:

1. Before - Using Evaluate()
   - 2 people named Morris
   - 2 people named Morris
2. After - Using GetDerivedElement()
   - 2 people named Morris
   - 1 people named Morris

[Delphi]

```
//OclResult := EcoSpace.OclService.Evaluate(OclQuery);              //Before
OclResult := EcoSpace.OclService.GetDerivedElement(nil, OclQuery);  //After
```

[C#]

```
//oclResult = EcoSpace.OclService.Evaluate(oclQuery);               //Before
oclResult = EcoSpace.OclService.GetDerivedElement(null, oclQuery);  //After
```

7 The OCL PS service

Using the IOclPsService interface, the OCL PS service allows the developer to evaluate OCL expressions within the database server instead of on the client.

Evaluating OCL can prove to be quite expensive in terms of memory and network usage. A harmless looking expression such as "Person.allInstances->select(lastName="Morris")->size" may result in an integer, but behind the scenes the following steps take place

1. All instances of the Person class are fetched from the persistence storage.
2. The resulting collection is filtered down to include only instances where the last name equals "Morris".
3. The size of the resulting collection is returned.

The potential bottleneck here is step #1. If there are over 1 million people stored in your database, then all 1 million objects will be fetched into memory for the OCL select() to be evaluated. "allInstances" is an expression that should be used as sparingly as possible.

The OCL PS service was introduced in ECO III, this service allows OCL to be evaluated by the persistence storage or, in other words, converted to SQL and evaluated by the database server. The benefit of evaluating selections etc on the database server are obvious, returning only two rows from a 1 million row table is the most obvious, selecting attributes where the mapped DB column has a server index is another. Originally parts of this interface were declared as part of the IOclService interface, but these were moved in order to provide a clear separation between in-memory and in-ps evaluation.

However, there are a number of restrictions that must be taken into account when choosing to use the OCL PS service.

1. All expressions must return a collection of objects. This means that an expression cannot end with operators such as ->size or ->first. For a list of valid PS OCL operations see appendix Services.A
2. All attributes and associations within the expression must be persistent. Obviously this means that transient model elements cannot be used as part of the expression, but also means that any derived attributes / associations are also invalid as they do not exist within the database as a column.
3. Only a subset of OCL is supported, certain operations such as ->subSequence (which returns a subset of a collection) are not supported by all databases, and are therefore not part of this subset.
4. All OCL is evaluated by the database server, none of it is evaluated in memory afterwards. If any of the objects returned have been modified locally by the application, but not updated to the persistence storage, it is possible to give the impression that the result set is incorrect.

Although the subsequence collection operation is not permitted, there is an overloaded version of the IOclPsService.Execute method that accepts "MaxAnswers" and "Offset" parameters.

To elaborate further on point #4, take a look at the following abbreviated example.

[Delphi]

```
//1
PeterMorris := Person.Create(EcoSpace);
PeterMorris.FirstName := 'Peter';
PeterMorris.LastName := 'Morris';
EcoSpace.UpdateDatabase();
OclPsService := EcoServiceHelper.GetOclPsService(EcoSpace);
//2
OclQuery := 'Person.allInstances' +
    '->select(lastName=''Morris'')';
OclResult := OclPsService.Execute(OclQuery);
ObjectCount := (OclResult as IElementCollection).Count;
MessageBox.Show(ObjectCount.ToString() + ' person named Morris');
```

```
//3
PeterMorris.LastName := ‘Johnston’;
//4
OclResult :=EcoSpace.OclPsService.Execute(OclQuery);
ObjectCount := (OclResult as IElementCollection).Count;
MessageBox.Show(ObjectCount.ToString() + ‘ person named Morris’);
```

[C#]

```
//1
peterMorris = new Person(EcoSpace);
peterMorris.FirstName = “Peter”;
peterMorris.LastName = “Morris”;
EcoSpace.UpdateDatabase();
oclPsService = EcoServiceHelper.GetOclPsService(EcoSpace);
//2
oclQuery = “Person.allInstances” +
    “->select(lastName=’Morris’)”;
oclResult = oclPsService.Execute(oclQuery);
objectCount = (oclResult as IElementCollection).Count;
MessageBox.Show(objectCount.ToString() + “ person named Morris”);
//3
peterMorris.LastName = “Johnston”;
//4
oclResult =EcoSpace.OclPsService.Execute(oclQuery);
objectCount = (oclResult as IElementCollection).Count;
MessageBox.Show(objectCount.ToString() + “ person named Morris”);
```


1. An instance of the Person class is created, with the name Peter Morris, the new instance is stored in the persistence storage by calling UpdateDatabase().
2. An in-PS evaluation is executed, selecting all Person objects whose last name equals "Morris". The result is typecast to IElementCollection so that the Count can be displayed. Run against an empty database the output correctly states that there is "1 person named Morris".
3. The last name is changed to "Johnston", but this change is not saved, so the change exists only locally within the EcoSpace.
4. The same in-PS evaluation now incorrectly states that there is "1 person named Morris".

Although this may seem like a big problem, using a combination of in-PS and in-memory OCL evaluation it is possible to return a list that is completely accurate, and even includes new instances that have not yet been saved. This can be achieved using the following technique

1. Evaluate the in-PS expression first.
2. Evaluate the expression again, but using "allLoadedObjects" instead of "allInstances".

The first evaluation performed within the persistence storage will fetch a list of objects into the EcoSpace. Evaluating "Person.allLoadedObjects" will now only evaluate against objects already fetched from the persistence storage, this includes all objects fetched during the in-PS evaluation plus any additional objects that may have been created but not yet saved. The in-memory evaluation will therefore provide a list of Person objects that have the last name "Morris", without having to fetch every Person object from the persistence storage.

[Delphi]

```
//1
OclQuery := ‘Person.allInstances’ +
    ‘->select(lastName = ‘’Morris’’)’;
EcoSpace.OclPsService.Execute(OclQuery);
//2
OclQuery := ‘Person.allLoadedObjects’ +
    ‘->select(lastName = ‘’Morris’’)’;
OclResult := EcoSpace.OclService.Evaluate(OclQuery);
//3
ObjectCount := (OclResult as IElementCollection).Count;
MessageBox.Show(ObjectCount.ToString() + ‘ person named Morris’);
```

[C#]

```
//1
oclQuery = “Person.allInstances” +
    “->select(lastName = ‘Morris’)”;
EcoSpace.OclPsService.Execute(oclQuery);
//2
oclQuery = “Person.allLoadedObjects” +
    “->select(lastName = ‘Morris’)”;
oclResult = EcoSpace.OclService.Evaluate(oclQuery);
//3
objectCount = (oclResult as IElementCollection).Count;
MessageBox.Show(objectCount.ToString() + “ person named Morris”);
```

The "allInstances" query is formulated, and executed by the database server. We do not need a reference to the result, we just need ECO to load the objects in question.

The same query is formulated again but this time using "allLoadedObjects" instead of "allInstances". The query is evaluated in-memory using the IOclServiceProvider.

The correct object count is displayed.

8 The extent service

Whenever ECO is instructed to fetch a complete list of objects by class, for example using "Person.allInstances", the resulting object list will be retrieved and then cached. The next time a complete list of instances is requested for the same class, the cached list will be returned.

Using the IExtentService allows the developer to interact with this cache. The methods within this interface have various overloaded variations, but essentially this service provides four functions.

8.1 AllInstances(IClass | Type | string)

This method has three overloaded implementations. Given either an IClass (See The type system service for details on how to retrieve an IClass reference for a modelled class) from the model, a type of a .net class generated by ECO, or the name of a class in the model, this method will return an IObjectList containing all object instances of the specified class. This list will not only include objects that have already been fetched from the persistence storage, but also any new and so far unsaved objects within the local EcoSpace.

8.2 AllLoadedInstances(IClass | Type | string)

If all instances have already been requested for the class then this method will return the same list as AllInstances(). If the application has not yet requested all instances, because it has only added new objects or because all OCL has been evaluated by the database, then this method will instead return only a list of all object instances that have already been retrieved.

8.3 Unload(IClass)

The first time a request is made for AllInstances of a class, ECO will access the persistence service and instruct it to return a list of object locators for all objects of the specified class within the database. This list will then be cached by the EcoSpace, so any subsequent requests for AllInstances will return the existing list.

This method will relinquish the cached AllInstances object list for a specified class. The next time a request is made for all instances of the specified class, ECO will reload the object locator list from the persistence storage.

This method can be useful when developing a multi-user application that does not use a remote ECO application for persistence (applications with remote persistence automatically invalidate this list when necessary). The application might unload the AllInstances cache for a certain class before displaying a form showing all object instances, in order to ensure that the user sees new object instances added by other users.

Unlike the Unload() method of the persistence service, this method will not unload the cached member values of the objects in the list.

8.4 SubscribeToObjectAdded(ISubscriber, IClass | Type | string)

This method allows the developer to register a Borland.Eco.Subscription.ISubscriber instance, which will be called back whenever an object locator for the class is added to the AllInstances list of the extent service; or put another way, whenever an object becomes known to the EcoSpace.

This method is useful when the application needs to perform an operation on every object within the EcoSpace. For example, if a utility class needed to place subscriptions on every loaded instance it would do something like this

1. Implement Borland.Eco.Subscription.ISubscriber.
2. Use the AllLoadedInstances() method to place the required subscriptions on objects already loaded.
3. Use the SubscribeToObjectAdded() method to ensure that it is informed whenever new objects are created or retrieved in the EcoSpace.
4. Place the required subscriptions whenever an object is added.

A reference to the added object may be obtained by typecasting the System.EventArgs parameter to a Borland.Eco.ObjectRepresentation.ElementChangedEventArgs

[Delphi]

```
//1
function MyHelperClass(Sender: System.Object; E: System.EventArgs): Boolean;
var
    ElementChangedArgs: ElementChangedEventArgs;
    AddedPerson: Person;
begin
    //2
    ElementChangedArgs := (E as ElementChangedEventArgs);
    //3
    if (ElementChangedArgs.Element.AsObject is Person) then
    begin
        AddedPerson := ElementChangedArgs.Element.AsObject as Person;
        //4
        MessageBox.Show('Added person ' + AddedPerson.LastName);
    end;
    Result := True;
end;
```

[C#]

```
//1
bool ISubscriber(object sender, System.EventArgs e)
{
    ElementChangedEventArgs elementChangedArgs;
    Person addedPerson;
    //2
    elementChangedArgs = (ElementChangedEventArgs) e;
    //3
    if (elementChangedArgs.Element.AsObject is Person)
    {
        addedPerson = (Person) elementChangedArgs.Element.AsObject;
        //4
        MessageBox.Show("Added person " + addedPerson.LastName);
    }
    return true;
}
```

1. The method signature is specified in accordance with the ISubscriber.Receive method declaration, and variables are declared.
2. The System.EventArgs parameter is typecast to a Borland.Eco.ObjectRepresentation.ElementChangedEventArgs instance.
3. The .net class type of the added object is checked to see if it is a Person or not.
4. If the added object is a person, the person's name is displayed.

8.5 SubscribeToObjectRemoved(ISubscriber, IClass | Type | string)

This method is almost the mirror of the SubscribeToObjectAdded() method. Instead of triggering whenever an object is created or loaded, this method will cause the subscriber to trigger each time an object is deleted. The trigger is executed as soon as IObject.Delete() is executed, rather than when a deleted object is updated using the persistence service (which the object locator to be relinquished).

Note that the subscriber is not triggered when an object is unloaded. The SubscribeToObjectAdded() method only triggers for both Create and Load because both of these actions cause a new object locator to exist within the cached AllInstances list within the EcoSpace. Merely unloading an object does not cause the EcoSpace to relinquish an object locator, therefore simply unloading an object does not cause this event to trigger.

9 The persistence service

Using the IPersistenceService, the persistence service is responsible for mediating all persistence (typically database) related operations on behalf of the EcoSpace. It will fetch objects, update objects, and if in a multi-user application will retrieve a list of changes made by other users for conflict reconciliation purposes.

9.1 UpdateDatabase()

The most simple method that this service implements is the UpdateDatabase() method. This method retrieves a list of new or modified (dirty) objects from the dirty list service, and then saves them to the persistence storage. If the persistence storage is a transactional database, then all updates will be performed within a single database transaction.

9.2 UpdateDatabaseWithList(IObjectList)

This method is identical to the UpdateDatabase() method, except that it accepts a list of objects to update in the form of an IObjectList instance. This list may be generated manually, by using the GetChangedObjects() method of an IUndoBlock, or by passing any other existing IObjectList. In fact, UpdateDatabase() does exactly that, it executes this method passing the entire list of dirty objects from IDirtyListService.AllDirtyObjects().

When updating a list of objects provided by an IUndoBlock it is important to be aware that whenever an object is saved using the persistence service, all undo blocks containing that object are removed from the undo service.

9.3 EnsureEnclosure(IObjectList)

This method ensures that the list passed to it contains the minimum set of objects required to ensure a logical update. For example, if two new objects are created, and are associated in some way, it would be illogical to update the persistence storage with only one of these objects; EnsureEnclosure() would identify the associated object as an additional item that should be added to the list. It is then safe for this list to be passed to the UpdateDatabaseWithList() method.

9.4 Unload(IObject | IObjectList)

This method has two overloads, one that accepts an IObject, and one that accepts an IObjectList. This method discards the cached attribute values held within the EcoSpace's memory, the values of the object's attributes will be re-fetched from the persistence storage the next time an attempt is made to read one of their values.

It is only possible to unload an object if the object is persistent, and not dirty. Transient objects can only be deleted; they cannot be unloaded because they are never loaded. Whereas dirty objects should either have their changes saved, or undone using an undo block. Attempting to unload an ineligible object will result in a System.InvalidOperationException being thrown.

9.5 EnsureRange(IObjectList, integer, integer)

Before describing this method it is important to understand something about ECO. Like many programmers, ECO is very lazy. When you ask ECO for a list of objects it doesn't actually fetch the objects from the persistence storage, what it does instead is to fetch a list of unique identifiers (internally known as an ObjectLocator). Only when an attempt is made to read one of the objects attributes does ECO decide it is time to fetch the data for the object in question.

Using this "lazy fetch" approach it is possible to save a potentially enormous amount of memory by simply not fetching any data unless it is actually needed. You can probably imagine that this is less than ideal if, for example, you wanted to iterate through a list of objects and check an attribute on each one. Each time you attempted to read the attribute ECO would fetch the values for that single object from the persistence storage, resulting in lots of single result row queries to the database, which we all know is nowhere near as efficient as a multi result row query.

Internally ECO will actually fetch object values as multi result row queries when evaluating queries that access an attribute of a class. "Person.allInstances" will fetch only a list of object locators, whereas "Person.allInstances->select(lastName="Morris")" will first fetch the object locators, and then perform a multi result row query to fetch the attributes of the objects in the list.

When it comes to accessing the persistence service via application code we can speed up our application, and reduce database trips by explicitly instructing ECO to fetch all of the attributes for our object list in one go, and this is exactly what the EnsureRange() method does for us. Accepting an IObjectList as a parameter, and also a "FromIndex" and "ToIndex" parameter, this method will pre-fetch the attributes as few queries as possible.

This technique is most commonly used when iterating through a list of associated objects. If for example we have a Person class, and this class has a multi role association named "DiaryEntries" to a class named "DiaryEntry", we may wish to iterate through PeterMorris.DiaryEntries and perform some kind of operation on each entry. Instead of allowing ECO to lazy fetch the attributes for each DiaryEntry object upon request, we can instruct ECO to pre-fetch the values for the associated objects like so

[Delphi]

```
//1
DiaryEntries :=
    PeterMorris.AsIObject().Properties. GetByLoopbackIndex(
    Person.Eco_LoopbackIndices.DiaryEntries) as IObjectList;
//2
EcoSpace.PersistenceService.EnsureRange(DiaryEntries, 0, DiaryEntries.Count - 1);
```

[C#]

```
//1
diaryEntries = (IObjectList)
    peterMorris.AsIObject().Properties. GetByLoopbackIndex(
    Person.Eco_LoopbackIndices.DiaryEntries);
//2
EcoSpace.PersistenceService.EnsureRange(diaryEntries, 0, diaryEntries.Count - 1);
```

1. First we switch from the .net class context over into the "ECO world" using AsIObject(). We then obtain an IProperty reference for our DiaryEntries multi role and type cast it as an IObjectList into a local variable named "DiaryEntries".
2. Using this IObjectList it is now a simple task to instruct the persistence service to fetch all attributes of the objects in the list.

9.6 EnsureRelatedObjects()

EnsureRelatedObjects() is similar in operation to the EnsureRange() example just illustrated. Instead of preloading all objects in a list, this method will instead iterate through every object in an IObjectList, and then preload all objects in the specified association.

The following example will demonstrate how to ensure that all OrderLine objects are fetched for all loaded PurchaseOrder objects.

[Delphi]

```
//1
var
    PurchaseOrders: IObjectList;
    ExtentService: IExtentService;
begin
    //2
    ExtentService := EcoServiceHelper.GetExtentService(EcoSpace);
    PurchaseOrders :=
        ExtentService.AllLoadedInstances(typeof(PurchaseOrder));
    //3
    EcoSpace.PersistenceService.EnsureRelatedObjects(
        PurchaseOrders, “Lines”);
end;
```

[C#]

```
//1
IObjectList purchaseOrders;
IExtentService extentService;
//2
extentService = EcoServiceHelper.GetExtentService(EcoSpace);
purchaseOrders =
    extentService.AllLoadedInstances(typeof(PurchaseOrder));
//3
EcoSpace.PersistenceService.EnsureRelatedObjects(
    purchaseOrders, “Lines”);
```

1. First a variable is declared to hold a list of all loaded purchase order objects, and a variable to hold a reference to the extent service.
2. Using the extent service, a list of purchase orders is obtained. Note that the list contains IObject instances rather than PurchaseOrder instances.
3. Finally, EnsureRelatedObjects() is executed, ensuring that all purchase order line objects (associated via PurchaseOrder.Lines) are loaded for every purchase order in the list.

9.7 Multi-user persistence methods

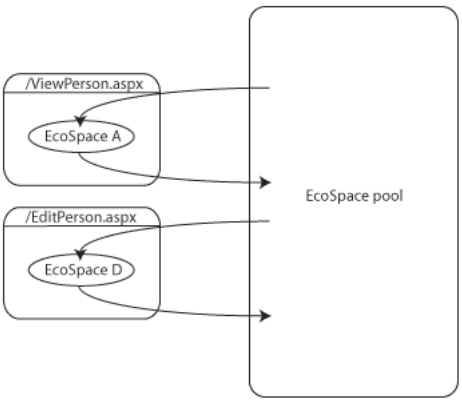
There are a number of methods within the persistence service that are implemented for the purpose of reconciling update conflicts between multiple running instances of applications. Although these are part of the persistence service, they will not be described in this chapter.

10 The external ID service

Every object instance within an EcoSpace is uniquely identifiable. Whether this is by an ECO generated object id, or a single/part primary key on a database, ECO requires a unique identifier so that it can perform persistence operations on the correct object when updating the database.

Using the IExternalIdService it is possible to either retrieve a string representation of an object instance's unique identifier, or to provide such a string representation and have ECO provide an object instance.

The ObjectForId() and IdForObject() methods of the external ID service should be used to hold weak references to objects when working with a pool of EcoSpaces. This is especially prevalent in ECO powered web service / web application projects, where you cannot guarantee that you will be working with the same EcoSpace instance across different page requests.



If there is more than one EcoSpace in your web application's/service's EcoSpace pool (which is recommended) then the flow in Figure 05 illustrates a likely scenario. A user views an object in ViewPerson.aspx and then decides to edit that object, at which point they are redirected to EditPerson.aspx. Storing the Person object in a session is a bad idea, because the Person instance belongs to EcoSpace "A", whereas EditPerson.aspx was allocated EcoSpace "D" from the pool.

Instead of passing ECO object instances between page requests, the web application/service should instead pass around the "id" of the object using IdForObject(). The receiving page should retrieve an object instance from its allocated EcoSpace using the mirror method ObjectForId().

Although this service is used primarily for web applications/web services, there are many more possible applications. Any time the identification of an ECO modelled object needs to be stored in some way, this service is the answer; for example, an ID may be useful if you create your own object -> XML -> object streaming service.

11 The variable factory service

Using the IVariableFactoryService interface, the variable factory service enables the developer to create a number of IElement based objects, which may then be used in various different parts of the ECO framework.

The IElement interface is prevalent throughout the ECO framework, many of the services expect IElement parameters, or return IElement results. The two OCL services for example may evaluate OCL expressions against the EcoSpace itself, or a specified IElement. IElement has many ancestor interfaces, including IObject, IProperty, IElementCollection and others.

In most circumstances it is anticipated that the developer will want to use a handle for declaring variables.

11.1 CreateConstant([IClassifier]

The CreateConstant() method accepts a single parameter of type System.Object, or an IClassifier identifying the type of value to expect plus a System.Object value. The object type must be a standard .net value type such as System.String or enum, or be an instance of a modelled class. See The type system service for information about the IClassifier interface.

Probably the most common use for CreateConstant() is when an application needs to add additional event-derived columns to one of the ECO handles such as ExpressionHandle or OclPsHandle. This chapter will not describe the steps required to implement an event-derived column on an ECO handle.

Given a reference to an IElement, it is possible to ascertain its value by reading its AsObject property. As this is a constant element, the AsObject property cannot be written to.

11.2 CreateVariable()

This method is almost identical to CreateConstant(), except that once the element has been created its AsObject property may be modified. Although possible, it is not advisable to change the type of the value that your variable holds.

This method has three overloaded variations. The first accepts a string identifying the name of the type the variable should hold, such as "System.String". The second accepts a .net Type, this is much less susceptible to error as a Type can be checked during compilation whereas a string can only be checked at runtime. The final overload accepts an IClass identifying the type of the variable this element will hold. For information on obtaining an IClass reference see "The type system service" elsewhere in this chapter.

11.3 CreateUntypedObjectList(Boolean)

This method accepts a single Boolean parameter, indicating whether or not it is permissible for the returned list to accept duplicate entries. The method then returns an IObjectList to which an IObject of any class type may be added.

This method is useful when the developer wishes to update the persistence storage with a custom list of objects.

11.4 CreateTypedObjectList([Type | IClass]

This method is similar to CreateUntypedObjectList(), the exception that it ensures that IObjects added to the list are of a certain type. The first parameter identifies a class by either its .net Type, or by an IClass retrieved from the type system service. If an attempt is made to add an object that is not of the specified type, and not a descendant of the specified type, a Borland.Eco.ObjectRepresentation.ArgumentNonConformException will be thrown.

11.5 CreateUntypedElementList(Boolean)

This method accepts a single Boolean parameter, indicated whether or not it is permissible to add duplicate elements to the list, it then returns an IElementCollection. It is then possible to add any IElement to this collection, including constants, variables, or IObjects.

11.6 CreateVariableList()

This method creates and returns an IModifiableVariableList instance. IModifiableVariableList descends from IExternalVariableList, which means that it may be used as a parameter for both of the OCL services.

Once an IModifiableVariableList reference has been obtained, it is possible to add any IElement descended interface instances to the list, and then access the element using a unique name.

[Delphi]

```
//1
var
    Vars: IModifiableVariableList;
    NameElement: IElement;
begin
    //2
    NameElement :=
        EcoSpace.VariableFactoryService.CreateVariable(typeof(string));
    NameElement.AsObject := “Peter Morris”;
    //3
    Vars := EcoSpace.VariableFactoryService.CreateVariableList();
    Vars.Add(‘Name’, NameElement);
    //4
    MessageBox.Show(Vars[‘Name’].Element.AsObject.ToString);
```

[C#]

```
//1
IModifiableVariableList vars;
IElement nameElement;
//2
nameElement = EcoSpace.VariableFactoryService.CreateVariable(typeof(string));
nameElement.AsObject = "Peter Morris";
//3
vars = EcoSpace.VariableFactoryService.CreateVariableList();
vars.Add("Name", nameElement);
//4
MessageBox.Show(vars["Name"].Element.AsObject.ToString());
```

1. Variables are declared to hold a name (IElement) and variable list (IModifiableVariableList).
2. The name element is created using CreateVariable(), and its value is then assigned using its AsObject property.
3. The variable list is created using CreateVariableList(), and then the name element is added to the list using "Name" as its unique name.
4. The name element is retrieved from the variable list using its unique name, its value is then obtained from Element.AsObject, before finally being displayed in a message box using ToString.

The IModifiableVariableList also supports some other standard list-type operations, such as Remove(string), RemoveAt(Integer), and Count. It also has a Subscribe method, so that a subscriber can receive notifications whenever an item is added/removed from the list.

12 The version service

Using the version service via the IVersionService interface, the developer is able to retrieve historical information about objects that have been identified as "Versioned" in the ECO model.

Object instances of classes that have been marked as Versioned are treated differently by the ECO persistence mechanism. By default each object within the database will have two additional columns, "TimeStampStart" and "TimeStampStop". These columns identify the life span of versioned objects.

Each time UpdateDatabase is executed a new integer timestamp is value allocated, and the current date/time recorded against it. These integers are used to identify at which date/time a versioned object instance is created, modified, or deleted.

When a new object instance is created the current timestamp is entered into its TimeStampStart column, and 2147483647 is entered into its TimeStampStop column, this records when the object came into existence, and the high TimeStampStop indicates that this row in the database is the current "live" data for the object.

TimeStampStart	TimeStampStop	ECO_ID	FullName
10	2147483647	5	Miss Jane Smith

When a versioned object is modified the TimeStampStop column of the live row is updated to the current timestamp value, and a new row is inserted into the table. This new row has the same ECO_ID (the unique identifier for an ECO object instance), the current timestamp for TimeStampStart, and the new modified attribute values.

TimeStampStart	TimeStampStop	ECO_ID	FullName
10	10	5	Miss Jane Smith
11	2147483647	5	Mrs Jane Jones

Finally, when a versioned object is deleted, the TimeStampStop column of the live row is updated with the current timestamp - 1.

TimeStampStart	TimeStampStop	ECO_ID	FullName
10	10	5	Miss Jane Smith
11	11	5	Mrs Jane Jones

With this overview of ECO object versioning out of the way, we can now cover the various methods of the IVersionService interface.

12.1 GetVersion(Integer, IElement)

When passed a timestamp number and an element (typically an IObject), this method will return a historic version of the element. Historical elements are immutable, so none of the objects attributes or associations may be modified.

12.2 ElementVersion(IElement)

This method will return the timestamp for the element passed, if the element is the live object then 2147483647 will be returned. It is possible to pass both live objects, and historical elements obtained via GetVersion() for example.

12.3 TimeForVersion(Integer)

This method accepts an integer representing a timestamp, and returns the date and time at which the timestamp was created. Combined with ElementVersion() it is possible to determine the date and time at which changes were made to an object.

12.4 VersionAtTime(DateTime)

This method will return an integer value, representing the timestamp at a give date and time. This integer value may then be used for various other version service methods, such as GetVersion(), or a number of the overloaded GetChangePointCondition() methods.

12.5 CurrentVersion

This property is for convenience only. It will always return the integer timestamp value of 2147483647, this can be compared with the result obtained from ElementVersion() in order to ascertain whether or not an element is historical.

12.6 MaxSavedVersion

Each time UpdateDatabase is executed, ECO will record the timestamp used by storing it in this property. Before any database updates are performed this property will return - 1.

12.7 GetChangePointCondition(IObject, Integer, Integer)

There are various overloads for this method, the one I shall concentrate on accepts an IObject, and start / stop timestamps. GetChangePointCondition() will return an AbstractCondition, which may then be used as a parameter for IPersistenceService.GetAllWithCondition() to return an IObjectList of all historical versions of an object. This feature is useful for displaying a version history of an object, which is a technique I will now demonstrate.

This next example will show how to display a list of historic versions of a given object in a DataGrid, and how to display the date and time of the version as an additional column. There are a number of preliminary steps required for this example that will not be covered; these include creating a model with a versioned class, configuring the persistence mapper, and also configuring the various ECO handles required.

[C#]

```
using Borland.Eco.Persistence
…
//1
IVersionService versionService;
IPersistenceService persistenceService;
IObjectList allHistoricalVersions;
AbstractCondition versionCondition;
//2
versionService = EcoServiceHelper.GetVersionService(EcoSpace);
persistenceService = EcoServiceHelper.GetPersistenceService(EcoSpace);
//3
versionCondition =
    versionService.GetChangePointCondition(rhRoot.Element as IObject,
        0, versionService.CurrentVersion);
//4
allHistoricalVersions = persistenceService.GetAllWithCondition(versionCondition);
//5
rhHistory.SetElement(allHistoricalVersions)
```

1. First we declare variables to hold references to
    1. The version service
    2. The persistence service
    3. An object list that will hold an IObject for each historical version of the object
    4. An AbstractCondition that will be used to instruct the persistence service to retrieve all historical versions from the database
2. The required services are requested from the EcoServiceHelper class.
3. IVersionService.GetChangePointCondition() is executed; passing an instance of a versioned class (stored in rhRoot), the first version required (zero), and the last version required. The variable "VersionCondition" then stores the result of the method call.
4. IPersistenceService.GetAllWithCondition() is executed using the condition created in step 3, this returns an IObjectList which is then stored in the "AllHistoricalVersions" variable.
5. Finally the IObjectList returned from the persistence service is stored in a ReferenceHandle named "rhHistory", the StaticValueTypeName of this handle is set to "Collection(Person)".

To hook up a DataGrid to this collection all we need to do is

1. Add a new expression handle to the form
2. Set the RootHandle property to rhHistory
3. Set the Expression property "self"
4. Set the expression handle as the DataSource for the DataGrid

Setting the element of rhRoot using rhRoot.SetElement(), and then executing the above code will result in something similar to the following screenshot.

In order to add a column showing the date and time of the version we need to first add a code-derived column to the expression handle, and then write the necessary code to retrieve the data from the version service.

1. Bring up the editor for the Columns property of the expression handle.
2. Click the drop-down arrow to the right of the "Add" button and select "EventDerivedColumn".
3. Name the column "TimeStamp".
4. Set the TypeName to "System.DateTime".
5. Now add the following code to the DeriveValue event.

```
//1
IVersionService versionService;
IVariableFactoryService variableFactoryService;
int versionNumber;
DateTime timeStamp;
//2
versionService = EcoServiceHelper.GetVersionService(EcoSpace);
variableFactoryService = EcoServiceHelper.GetVariableFactoryService(EcoSpace);
//3
switch (e.Name)
{
    case “TimeStamp”:
        //4
        versionNumber = versionService.ElementVersion(e.RootElement);
        //5
        timeStamp = versionService.TimeForVersion(versionNumber);'
        //6
        e.ResultElement = variableFactoryService.CreateConstant(timeStamp);
        break;
    default:
        //7
        throw new Exception(e.Name + “ not derived properly”);
}
```

1. First we declare variables to hold references to
    1. The version service
    2. The variable factory service
    3. The version number of the current historical object instance
    4. The date and time of the historical object instance
2. The required services are requested from the EcoServiceHelper class.
3. The name of the column being code-derived is checked to see if it is the "TimeStamp" column we added.
4. The version number for the current historical object instance is retrieved from the version service.
5. The version number is then used to retrieve a DateTime from the version service.
6. Finally the variable factory service is used to create a result element based on the date time.
7. Finally, for the sake of good programming, an exception is thrown if the column name was unrecognised.


WinForm
Name
John
Update DB
Name
TimeStamp
Pete
25 November 2005 16:44:56
Lyn
25 November 2005 16:45:00
John
25 November 2005 16:45:12

13 The state service

Using the IStateService interface, the state service allows the developer to determine if an object or property is dirty, or if an object is new.

13.1 IsNew(IObject)

The IsNew() method accepts a single parameter of type IObject, and returns True or False depending on whether or not the object passed is new or not. Once a new object has been saved to the persistence storage it is no longer considered to be new. An IObject reference may be obtained for any .net ECO class by executing its AsIObject() method.

13.2 IsDirty(IObject | IProperty)

The IsDirty() method returns "True" if the object or attribute/association end has been modified. An IProperty is used by ECO to represent both attributes of a class, and association ends (both single and multi roles).

Using the model in Figure 08 I will now demonstrate how to determine the Dirty state of various members of a class, and explain why IsDirty() for one of the members (perhaps unexpectedly) returns "False".

[Delphi]

```
//1
var
    StateService: IStateService;
    PeterMorris: Person;
    NewDiaryEntry: DiaryEntry;
    FirstNameProperty: IProperty;
    LastNameProperty: IProperty;
    DiaryEntriesProperty: IProperty;
    PersonProperty: IProperty;
//2
begin
    PeterMorris := Person.Create(EcoSpace);
    PeterMorris.FirstName := 'Peter';
    PeterMorris.LastName := 'Morris';
    EcoSpace.UpdateDatabase();
    //3
    NewDiaryEntry := DiaryEntry.Create(EcoSpace);
    PeterMorris.DiaryEntries.Add(NewDiaryEntry);
    PeterMorris.LastName := 'Johnston';
    //4
    FirstNameProperty :=
        PeterMorris.AsIObject().Properties.GetByLoopbackIndex(
        Person.Eco_LoopbackIndices.FirstName);
    LastNameProperty :=
        PeterMorris.AsIObject().Properties.GetByLoopbackIndex(
```

```
        Person.Eco_LoopbackIndices.LastName);
    DiaryEntriesProperty :=
        PeterMorris.AsIObject().Properties.GetByLoopbackIndex(
        Person.Eco_LoopbackIndices.DiaryEntries);
    PersonProperty :=
        NewDiaryEntry.AsIObject().Properties.GetByLoopbackIndex(
        DiaryEntry.Eco_LoopbackIndices.Person);
    //5
    StateService := EcoServiceHelper.GetStateService(EcoSpace);
    //6
    MessageBox.Show('Person.FirstName is dirty: ' +
        StateService.IsDirty(FirstNameProperty));
    MessageBox.Show('Person.LastName is dirty: ' +
        StateService.IsDirty(LastNameProperty));
    MessageBox.Show('Person.DiaryEntries is dirty: ' +
        StateService.IsDirty(DiaryEntriesProperty));
    MessageBox.Show('DiaryEntry.Person is dirty: ' +
        StateService.IsDirty(PersonProperty));
end;
```

[C#]

```
//1
IStateService stateService;
Person peterMorris;
DiaryEntry newDiaryEntry;
IProperty firstNameProperty;
IProperty lastNameProperty;
IProperty diaryEntriesProperty;
IProperty personProperty;
//2
{
    peterMorris = new Person(EcoSpace);
    peterMorris.FirstName = "Peter";
    peterMorris.LastName = "Morris";
    EcoSpace.UpdateDatabase();
    //3
    newDiaryEntry = new DiaryEntry(EcoSpace);
    peterMorris.DiaryEntries.Add(newDiaryEntry);
    peterMorris.LastName = "Johnston";
    //4
    firstNameProperty =
        peterMorris.AsIObject().Properties.GetByLoopbackIndex(
        Person.Eco_LoopbackIndices.FirstName);
    lastNameProperty =
        peterMorris.AsIObject().Properties.GetByLoopbackIndex(
        Person.Eco_LoopbackIndices.LastName);
    diaryEntriesProperty =
        peterMorris.AsIObject().Properties.GetByLoopbackIndex(
        Person.Eco_LoopbackIndices.DiaryEntries);
    personProperty =
        newDiaryEntry.AsIObject().Properties.GetByLoopbackIndex(
        DiaryEntry.Eco_LoopbackIndices.Person);
    //5
    stateService = EcoServiceHelper.GetStateService(EcoSpace);
    //6
    MessageBox.Show("Person.FirstName is dirty: " +
        stateService.IsDirty(firstNameProperty));
    MessageBox.Show("Person.LastName is dirty: " +
        stateService.IsDirty(lastNameProperty));
    MessageBox.Show("Person.DiaryEntries is dirty: " +
        stateService.IsDirty(diaryEntriesProperty));
    MessageBox.Show("DiaryEntry.Person is dirty: " +
        stateService.IsDirty(PersonProperty));
}
```

1. The variables used within the method are declared.
2. A person named "Peter Morris" is created, and a call is made to UpdateDatabase() to ensure that the object is not dirty.

3. A new DiaryEntry is created, and the following modifications are made to the Person object
    1. The new DiaryEntry object is added to Peter Morris's DiaryEntries.
    2. Peter Morris's last name is changed to Johnston.
4. An IProperty reference is obtained for each of the Person object's members, and for the DiaryEntry's "Person" member. This is achieved by performing the following
    1. Switching from .net class context over to the "ECO world" by calling AsIObject() on the class,
    2. Accessing the "Properties" property of IObject,
    3. Using the GetByLoopbackIndex() method to access a specific IProperty,
    4. Using the Person.Eco_LoopbackIndices.<MemberName> constant to identify the member index, rather than depending on a property name,
5. A reference is obtained for the state service, using the EcoServiceHelper,
6. Finally a message is displayed for each member, and an indication as to whether or not the member is dirty.

The output from the above method is as follows

1. Person.FirstName is dirty: False
2. Person.LastName is dirty: True
3. Person.DiaryEntries is dirty: False
4. DiaryEntry.Person is dirty: True

Earlier I sneaked the words "perhaps unexpectedly" into a sentence. Did you spot which one it was? The answer is, number 3! One of the lines of code in the example clearly added a new DiaryEntry object to Peter Morris's DiaryEntries, so why does IsDirty() return False?

13.3 ECO embedding

A full description of embedding would be out of place for a chapter on ECO services, however, I will briefly explain the previous scenario in order to clarify the IsDirty() result we received.

When generating the database schema for the Person / DiaryEntry model, ECO needs some way of identifying to which Person a DiaryEntry belongs. This is achieved by recording (embedding) the unique ID of the person object into the DiaryEntry table, so, the DiaryEntry table would have a column named "Person". When ECO needs to determine which DiaryEntry objects should appear in Person.DiaryEntries it instructs the persistence service to retrieve all DiaryEntry objects where the Person column has the same value as the current Person's unique ID.

Seeing as IsDirty() is related to whether or not the element needs to be updated to the persistence storage, it may now be quite obvious that only DiaryEntry.Person may be dirty, as it is the only member with a direct mapping into the database.

14 The type system service

The type system allows the developer to validate the model, or to inspect its structure, using the ITypeSystemService interface.

14.1 ValidateModel(StringCollection

This method accepts two parameters; the first is a StringCollection to which the type system services adds a string for each model error encountered, and the second is an SqlDatabaseConfig which is used to validate persistence related information such as PersistenceMapper settings for attributes and table/column names. This method returns a Boolean, "True" if the model is valid, "False" if any errors were added to the StringCollection.

14.2 TypeSystem

The TypeSystem property returns an instance of IEcoTypeSystem. The IEcoTypeSystem interface enables the developer to retrieve IClassifier / IClass interfaces from the model given a .net type or a modelled class name, these instances are often used as parameters in various other services' methods; IExtentService.Unload() for example.

The TypeSystem also allows the developer to examine the model, iterating over classes, attributes, association ends, etc. For each of these elements it is then also possible to examine the model element (IModelElement) further, for example, by checking for any tagged values added to the element within the modeller.

The IEcoTypeSystem will be explained in more detail later, once some of the lower level interfaces have first been described.

14.2.1 IModelElement

The first time an ECO powered application is started, ECO will use reflection to discover the structure of the model. Instead of using .net reflection each time the model information is used, ECO will improve performance by creating an in-memory representation of the model.

Every element within the model implements IModelElement. This interface provides us with the following useful information

1. Name: A name for the element. This could be the class name, attribute name, method name, etc.
2. Package: A reference to the package (IPackage) to which the element belongs.
3. TaggedValues: A collection of ITaggedValue instances. This allows the developer to attach meta-data to model elements, somewhat similar to how .net adds meta-data to classes / properties etc.
4. Constraints: A collection of IConstraint instances. This allows the developer to specify a list of OCL expressions against an element for validation purposes.

14.2.2 IClassifier

Although the name of this interface may imply that it represents a modelled class, it does not. An IClassifier is used to

describe any type used within the model, this does include modelled classes (via the IClass interface), but it also represents types such as System.Byte and System.String. A list of all classifiers within the model can be obtained via the TypeSystem.AllDataTypes property.

The IClassifier has the following interesting properties

1. ObjectType: This property returns a .net type for the classifier, for example, "System.Byte".
2. SubTypes: This property returns a collection of IClassifier instances, one for each immediate subclass of the current type.
3. SuperTypes: Because the UML specification allows for multiple-inheritance, this property returns a collection of IClassifier instances. However, due to the fact that .net does not support multiple-inheritance, this collection will contain exactly one or zero instances.
4. Features: This property returns a collection of IFeature instances. If the IClassifier is a modelled class then list list will contain one IFeature instance for each attribute, association end, operation, and trigger.
5. EcoClassifier: This property returns an instance of IEcoClassifier, an instance that contains additional information about the current classifier.

14.2.3 IEcoClassifier

This interface contains information about an IClassifier that is specific to ECO, and will not be found within the UML specification. This interface contains only a few properties/methods of interest, these are

1. LeastCommonType(IClassifier): This method will traverse up the inheritance tree of the model and return the first common ancestor of both classes. If the two classes do not share a common ancestor, then an IClassifier named "ECOModelRoot" is returned, this is the implicit root class of all models.
2. IsA(IClassifier): This method with check if the current classifier descends from the classifier passed as a parameter, it will return "True" if it is, otherwise it will return "False".
3. IsAbstract: This property indicates whether or not it is possible to create an instance of the class.

14.2.4 IClass

The IClass interface is used to represent a modelled class. The properties of interest here are

1. IsAssociationClass: This property returns "True" if this class is an association class (link table in RDBMS talk) between two classes.
2. EcoClass: This property returns an instance of IEcoClass, an instances that contains additional information about the current modelled class.

14.2.5 IEcoClass

This interface contains information about a modelled class (IClass) that is specific to ECO, and will not be found within the UML specification. This interface contains a number of properties, the main properties of interest are

1. AllStructuralFeatures: This property returns a collection of IStructuralFeature. The difference between this property and its inherited IClassifier.Features is that IClassifier.Features will return only attributes / association ends introduced in the current class, whereas IEcoClass.AllStructuralFeatures returns a list of all attributes/ association ends from the current

class plus all of its combined super classes.

2. AllMethods: This property returns a collection of IMethod. This list will contain all operations and triggers introduced in the current class plus the combined operations and triggers of all its super classes.
3. Persistent: This Boolean property returns "True" if the class has been modelled as persistent, and "False" if it has been modelled as transient.

14.2.6 IAttribute

This interface is used to represent a modelled attribute, owned by a class. This interface inherits the usual TaggedValues etc from IModelElement, in addition it introduces the two following properties

1. InitialValue: This property returns an IExpression indicating what the intial value of this attribute should be when a new object instance is created.
2. EcoAttribute: This property returns an instance of IEcoAttribute, an instances that contains additional information about the current attribute.

IAttribute instances are obtained by iterating through either IClassifier.Features or IEcoClass.AllStructuralFeatures and checking if the IStructuralFeature returned may be type-cast to an IAttribute.

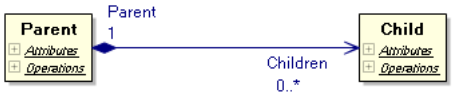
14.2.7 IEcoAttribute

This interface contains additional (non UML specification) information about an attribute.

1. AllowNull: This Boolean property indicates whether or not a "NOT NULL" constraint should be created in the database when creating the table for the owning class.
2. DefaultDbValue: This property returns a string that should be used as a "DEFAULT" on a table column when creating the table for the owning class.
3. Length: This integer property returns the maximum allowed length for the attribute, this is only relevant if the attribute is a string.
4. InitialValueAsObject: This property returns a System.Object, representing a .net value of the IAttribute.InitialValue.
5. IsStateAttribute: This Boolean property indicates whether or not the attribute is used to store the current state of the owning class's state machine.

14.2.8 IAssociationEnd

This interface is used to represent a single end of and association between two classes.



In the above figure there are two classes, Parent and Child. There is a single association between these two classes, the ends of the association are named Parent (single) and Children (multiple) respectively.

The association is navigable only in one direction, the Child class cannot see which object its parent is. There is an aggregate symbol appearing on the association line next to Parent, this means that the opposite end of the association

(Child) is an aggregate of the Parent class.

When an IAssociationEnd is encountered on a class, you are not actually looking at the end that is attached to the class. What we get instead is information about the opposite end. In other words, when we encounter an IAssociationEnd on Parent, what we are looking at is an association end named "Children", the property that the Parent class will own.

[Delphi]

```
//1
uses Borland.Eco.UmlRt;
…
var
    AssociationInfo: string;
    ParentClass: IClass;
    ChildrenEnd: IAssociationEnd;
//2
begin
    ParentClass := EcoSpace.TypeSystem.GetClassByType(typeof(Parent));
    ChildrenEnd :=
        ParentClass.Features.GetItemByName('Children') as IAssociationEnd;
    //3
    AssociationInfo :=
        'Name: ' + ChildrenEnd.Name + #13#10 +
        'Aggregation: ' + ChildrenEnd.Aggregation.ToString + #13#10 +
        'Navigable: ' + ChildrenEnd.IsNavigable.ToString;
    //4
    MessageBox.Show(AssociationInfo);
end;
```

[C#]

```
//1
using Borland.Eco.UmlRt;
…
string associationInfo;
IClass parentClass;
IAssociationEnd childrenEnd;
//2
parentClass = EcoSpace.TypeSystem.GetClassByType(typeof(Parent));
childrenEnd = (IAssociationEnd) parentClass.Features.GetItemByName("Children");
//3
associationInfo =
    "Name: " + childrenEnd.Name + "\n" +
    "Aggregation: " + childrenEnd.Aggregation.ToString() + "\n" +
    "Navigable: " + childrenEnd.IsNavigable.ToString();
//4
MessageBox.Show(associationInfo);
```

1. Variables are defined to hold the IClass for Parent, and the IAssociationEnd for the parent class's "Children" feature.
2. An IClass is obtained for the Person class. The IClass is then used to return an IStructuralFeature by the name of "Children", which is then type-cast to an IAssociationEnd.
3. The AssociationInfo string is built up of
   - Name
   - Aggregation
   - Navigable
4. Finally a message box is shown, displaying the association information.

The output of the message box reads

- Name: Children
- Aggregation: Composite
- Navigable: True

Clearly this is describing the association end that is visually linked to the Child class in the diagram. The opposite end of the association is easily obtainable via the OppositeEnd property of the IAssociationEnd.

14.2.9 IEcoAssociationEnd

The most interesting property of this interface is without doubt the Class_ property, which is of type IClass. Using the Class_ property it is possible to determine the base type of object that should be inserted into the class's association end.

14.2.10 IPackage

The TypeSystem.Packages property returns a collection of IPackage. This collection will contain a single IPackage instance for every package used by the EcoSpace that returned the service.

The IPackage interface has an "ownedElements" property, which returns a complete list of elements owned by the package. However, non-standard UML features of ECO packages were separated into a much more useful IEcoPackage interface, accessible via the package's EcoPackage property.

14.2.11 IEcoPackage

This interface represents a modelled package, and has the following interesting properties

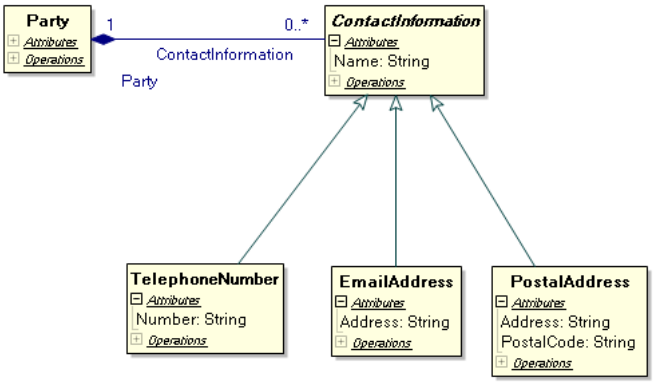
1. Id: This property contains the unique identifier (GUID) for the modelled package.
2. Classes: This property returns a collection of IClass instances, one for each modelled class within the package.
3. Associations: This property returns a collection of IAssociation instances.

14.3 Using the type system service

When programming a user interface for an ECO driven application, it is quite easy to make the mistake of hard-coding parts of the UI; for example, providing a list of available classes in a listbox. This is not only inflexible, but requires a lot of maintenance whenever the model is altered.

A more flexible way to approach this kind of scenario is to read the model information itself, and to then populate the UI based on the model that the application is currently executing. Two very common requirements are

1. Find a list of all subclasses of a certain class (recursively), and check if they are abstract or not.
2. Check one or more tagged values on a class.



Going back to the Party / ContactInformation example from earlier in this chapter, you will see that a Party has zero to many pieces of contact information. Each piece of contact information is structurally different, some contain email addresses, some telephone numbers, and some postal addresses; but fundamentally they are all a way of contacting the party in question.

In a WinForm application it might be tempting to show a list (a context menu for example) of classes that descend from ContactInformation, but what happens when the model is updated and a new type of contact information is added? What we really need is a way to discover all non-abstract descendents of ContactInformation at runtime, and then allow the user to select a class from a list using a user-friendly display name.

To achieve this, first I created a simple MenuItem descendant class called "ClassifierMenuItem". The only addition to this class was a property of type IClass named "Classifier", which is responsible for identifying which ECO object type should be created when the menu item is selected. I will not list the source for this class as it is so trivial.

Next a button is added to the form, "Add contact information", the Click event code for this button will do the following

1. Check if NewContactInformationContextMenu has any menu items, if not then find create the menu structure.
2. Display the context menu

[Delphi]

```
//1
var
    ContactInformationClass: IClass;
//2
begin
    if (NewContactInformationContextMenu.MenuItems.Count = 0) then
    begin
        ContactInformationClass:=
            EcoSpace.TypeSystem.GetClassByType(
            typeof(ContactInformation));
        AddMenuItemsForClass(ContactInformationClassifier);
    end;
    //3
    NewContactInformationContextMenu.Show(
        NewContactInformationButton, Point.Create(0, 0));
end;
```

[C#]

```
//1
IClass contactInformationClass;
//2
if (NewContactInformationContextMenu.MenuItems.Count == 0)
{
    contactInformationClassifier =
        EcoSpace.TypeSystem.GetClassByType(
        typeof(ContactInformation));
    AddMenuItemsForClass(contactInformationClass);
}
//3
NewContactInformationContextMenu.Show(
    NewContactInformationButton, new Point(0, 0));
```

AddMenuItemsForClass() will accept a single IClass parameter, and perform the following steps

1. If the classifier is not abstract then create a menu item for the class
2. Add all subclasses of the current class to the menu

[Delphi]

```
//1
procedure Winform1.AddMenuItemsForClass(CurrentClass: IClass);
var
    NewMenuItem: ClassifierMenuItem;
    SubClass: IClassifier;
begin
    if not CurrentClass.EcoClass.IsAbstract then
    begin
        NewMenuItem = ClassifierMenuItem.Create();
        NewContactInformationContextMenu.MenuItems.Add(
            NewMenuItem);
        NewMenuItem.Text = CurrentClass.Name;
        NewMenuItem.Classifier := CurrentClass;
        Include(NewMenuItem.Click, Self.CreateNewObject);
    end;
    //2
    for SubClass in CurrentClass.SubTypes do
        AddMenuItemsForClass(SubClass as IClass);
end;
```

[C#]

```
//1
private void AddMenuItemsForClass(IClass currentClass);
{
    ClassifierMenuItem newMenuItem;
    //2
    if (! currentClass.EcoClass.IsAbstract)
    {
        newMenuItem = new ClassifierMenuItem();
        NewContactInformationContextMenu.MenuItems.Add(
            newMenuItem);
        newMenuItem.Text = currentClass.Name;
        newmenuItem.Classifier = currentClass;
        newMenuItem.Click +=
            new System.EventHandler(this.CreateNewObject);
    }
    //3
    foreach(IClassifier subClass in currentClass.SubTypes)
        AddMenuitemsForClass( (IClass) subClass);
}
```

Now that we have a way to discover all non-abstract subclasses, and have added menu items for each, all that is needed is the event code that creates an instance of the object and assigns it to the current Party object. This is achieved using "The object factory service".

[Delphi]

```
//1
procedure Winform1.CreateNewObject(Sender: System.Object; E: System.EventArgs);
var
    CurrentParty: Party;
    NewIObject: IObject;
    NewContactInformation: ContactInformation;
    MenuItem: ClassifierMenuItem;
begin
    MenuItem := Sender as ClassifierMenuItem;
    //2
    NewIObject :=
        EcoSpace.ObjectFactoryService.CreateNewObject(
        MenuItem.Classifier);
    //3
    NewContactInformation := NewIObject.AsObject as ContactInformation;
```

```
    //4
    CurrentParty := rhRoot.Element.AsObject as Party;
    CurrentParty.ContactInformation.Add(NewContactInformation);
end;
```

[C#]

```
//1
private void CreateNewObject(object sender, System.EventArgs e)
{
    Party currentParty;
    IObject newIObject;
    ContactInformation newContactInformation;
    ClassifierMenuItem menuItem = (ClassifierMenuItem) sender;
    //2
    newIObject =
        EcoSpace.ObjectFactoryService.CreateNewObject(
            menuItem.Classifier);
    //3
    newContactInformation = (ContactInformation) newIObject.AsObject;
    //4
    currentParty = (Party) rhRoot.Element.AsObject;
    currentParty.ContactInformation.Add(newContactInformation);
}
```

1. The sender (a menu item) is type-cast to a ClassifierMenuItem.
2. A new instance of the menu item's class is created using the object factory service.
3. The IObject is converted to a .net object type, and then type-cast to a ContactInformation reference.
4. The current party object is obtained from the rhRoot handle on the form, and then the new contact information is added to its ContactInformation association.

The popup menu will display names such as "TelephoneNumber", "EmailAddress", and "PostalAddress". To improve the user experience it would be much better if the menu items would read "Telephone number", "Email address", and "Postal address". To achieve this is a very simple process. A tagged value holding the display text may be attached to each of the classes in the model, this text may be read from the model when constructing the model. Only a few additional lines of code are required.

First, attach a tagged value to each class like so:

[Delphi]

```
type
    [UmlTaggedValue('MyApp.DisplayName', 'Telephone number')]
    TelephoneNumber = class(ContactInformation)
```

[C#]

```
[UmlTaggedValue(“MyApp.DisplayName”, “Telephone number”)]
public class TelephoneNumber : ContactInformation
```

and then change the original source code so that the NewMenuItem.Text is determined like this

[Delphi]

```
//1
if CurrentClass.TaggedValues.GetItemByTag(‘MyApp.DisplayName’) <> nil then
   //2
    NewMenuItem.Text :=
        CurrentClass.TaggedValues.GetItemByTag(
        ‘MyApp.DisplayName’).Value
//3
else
    NewMenuItem.Text = CurrentClass.Name;
```

[C#]

```
//1
if (currentClass.TaggedValues.GetItemByTag(“MyApp.DisplayName”) != null)
    //2
    newMenuItem.Text =
        currentClass.TaggedValues.GetItemByTag(
        “MyApp.DisplayName”).Value;
//3
else
    newMenuItem.Text = currentClass.Name;
```

1. First a check is made to see if the current modelled class has a tagged value named "MyApp.DisplayName".
2. If it does then the new menu item's Text property is set to the value of the tagged value
3. Otherwise the new menu item's Text property is set to the name of the modelled class.

Of course the tagged value does not need to return the actual display name. Tagged values are meta-data for your model and may be used for all sorts of things. It would be just as easy to return the ID of a resource string, making it possible to have your menu items multilingual.

15 The action language service

16 The action language type service

17 Retrieving ECO services from the business layer

So far we have only covered examples of how we retrieve an ECO service from within the application's source code. Of course it is perfectly reasonable to require the services of the ECO framework from within the business classes themselves.

Considering the fact that it is possible (I would say "recommended") to create your business classes as part of a stand alone binary package so that they may be reused by different GUI applications, we are presented with a situation where our business classes belong to a binary in which there is no EcoSpace for us to retrieve ECO services from.

The typical example of transferring money from one bank account to another would require the use of a transaction, the business logic in question would require the StartTransaction, CommitTransaction, and RollbackTransaction methods of the IUndoService.

17.1 Stepping into the ECO world

Earlier in this chapter I explained how the generated source code for our business classes does not contain any ECO framework methods, despite this it is still possible to move from the class based structure of your business classes, over into what is commonly known as “The ECO world”.

To switch to this interface driven world is very simple. Each business class in our generated source code implements an interface called ILoopBack. I will not describe the features of this interface except to say that it inherits a method named AsIObject(). This means that at any point or classes can switch over from “Business context” to “Framework context” simply by retrieving an IObject reference from a method named “AsIObject”.

IObject provides us with a lot of ECO information about the class including model information, properties, and so on. The property we are interested in at this point is one by the name of ServiceProvider. ServiceProvider is an instance of IEcoServiceProvider, the same interface we used earlier against the EcoSpace. In fact, the ServiceProvider instance returned from this property is the EcoSpace of the application that created the current instance of our business class, but we should never rely on this fact because it is purely coincidental, and may not be the case in future versions of ECO.

So, retrieving a service from within a method of a business class itself is as simple as calling AsIObject().ServiceProvider.GetEcoService, as demonstrated below:

[Delphi]

```
//1
var
DirtyListService: IDirtyListService;
//2
begin
DirtyListService :=
    AsIObject.ServiceProvider.GetEcoService(typeof(IDirtyListService))
    as IDirtyListService;
end;
```

[C#]

```
//1
IDirtyListService dirtyListService;
//2
dirtyListService = (IDirtyListService)
AsIObject().ServiceProvider.GetEcoService(typeof(IDirtyListService));
```

1. First we declare a variable of the correct type that we will use to hold a reference to the service returned to us.
2. Next we switch from “business classes” context over into the “ECO world” by obtaining an IObject reference from AsIObject(). Using the ServiceProvider we then request an instance of the IDirtyListService by calling GetEcoService.

In this chapter I have switched between three different techniques for obtaining a service, each approach has its benefits and drawbacks:

1. EcoSpace.ServiceName
   - **PRO:** Returns a strongly typed reference (IOclService etc) so no additional typecast is required.
   - **PRO:** It is very short, so simple to write.
   - **CON:** These properties are not part of the EcoSpace’s ancestor (DefaultEcoSpace), so they are only accessible within the application that owns the EcoSpace.
   - **CON:** By default only standard ECO services are available, custom services would require manual addition of a property in the EcoSpace class.
2. EcoServiceHelper.Get{ServiceName}(Object)
   - **PRO:** Returns a strongly typed reference.
   - **PRO:** The service may be obtained by passing an EcoSpace, IObject, or .net object implementing ILoopback.
   - **PRO:** Available to classes within models compiled as separate assemblies, as well as the application.
   - **CON:** Only standard ECO services are available.
3. IEcoServiceProvider.GetEcoService(Type)
   - **PRO:** Able to retrieve both standard ECO services and custom services that have been registered with the EcoSpace.
   - **PRO:** Available to classes within models compiled as separate assemblies, as well as the application.
   - **CON:** A much more complicated statement means more time taken to type it out.
   - **CON:** The result must be typecast to the correct service type.

18 Registering custom services

ECO allows the application developer to register additional services. Once registered, references to these services may be obtained anywhere an IEcoServiceProvider instance is available; this includes the application itself, the business classes, or any 3rd party helper classes that accept an EcoSpace / IEcoServiceProvider / IElement descendant.

Custom services can be thought of as EcoSpace extensions. For example, a class could very easily query its ServiceProvider to see if a MyCompanyName.IAuditTrailService has been registered, if it receives a reference then it could execute methods on that service to record changes made to the object.

The first step when creating custom services is to design the interface for your service. Once the interface has been created it is important that the interface definition is compiled into its own assembly, by "own assembly" I mean that it should not be compiled into the same assembly as the class that implements the interface. Although this is not an ECO requirement it is good practise, ECO itself holds all of its interface definitions in a file named Borland.Eco.Interfaces.dll.

[Delphi]

```
uses Borland.Eco.ObjectRepresentation;
…
type
    IAuditTrailService = interface
        procedure ObjectDeleted(deletedObject: IObject;
            currentUserName: string);
    end;
```

[C#]

```
using Borland.Eco.ObjectRepresentation;
…
public interface IAuditTrailService
{
    void ObjectDeleted(IObject deletedObject,
        string currentUserName);
}
```

The next step is to create another assembly and then to add a class that will implement the interface. This very simple example will simply show a message box, obviously this is not a good idea for real applications, because the service may be consumed by an ASP .net based application, or an application with no UI at all (such as a Windows service).

[Delphi]

```
uses
    Borland.Eco.ObjectRepresentation, Borland.Eco.Services,
    System.Windows.Forms;
…
interface
type
    AuditTrailService = class(System.Object, IAuditTrailService)
        procedure ObjectDeleted(DeletedObject: IObject;
            CurrentUserName: string);
    end;
…
implementation

procedure AuditTrailService.ObjectDeleted(DeletedObject: IObject;
    CurrentUserName: string);
var
    ExternalIdService: IExternalIdService;
begin
    ExternalIdService := DeletedObject.ServiceProvider.GetEcoService(
        typeof(IExternalIdService)) as IExternalIdService;
```

```
        MessageBox.Show(CurrentUserName + ‘ just deleted ‘ +
            ExternalIdService.IdForObject(deletedObject));
end;
```

[C#]

```
using Borland.Eco.ObjectRepresentation;
using Borland.Eco.Services;
using System.Windows.Forms;
…
public class AuditTrailService : IAuditTrailService
{
    public AuditTrailService()
    {
    }

    public void ObjectDeleted(IObject deletedObject,
        string currentUserName)
    {
        IExternalIdService externalIdService;
        externalIdService = (IExternalIdService)
            deletedObject.ServiceProvider.GetEcoService(
            typeof(IExternalIdService));
        MessageBox.Show(currentUserName + " just deleted " +
            externalIdService.IdForObject(deletedObject));
    }
}
```

The implementation of the service uses the ServiceProvider of the deleted IObject to obtain a reference to the ExternalIdService. The current user's name is the displayed along with the external ID (the unique ID of the object) in a message box.

Before this service may be consumed it must be registered with the application's EcoSpace. It is not possible to replace a service that has already been registered. Registering a service for an existing type (IPersistenceService for example) will result in a System.InvalidOperationException being thrown.

[Delphi]

```
MyAuditTrailService := AuditTrailService.Create();
EcoSpace.RegisterService(typeof(IAuditTrail), MyAuditTrailService);
```

[C#]

```
myAuditTrailService = new AuditTrailService();
EcoSpace.RegisterService(typeof(IAuditTrail), myAuditTrailService);
```

Finally this new service may be used anywhere a reference to IEcoServiceProvider is available. For example, logging when an object instance is deleted.

[Delphi]

```
procedure Person.PreDelete();
var
    AuditTrailService: IAuditTrailService;
begin
    AuditTrailService := AsIObject.ServiceProvider.GetEcoService(
        typeof(IAuditTrailService)) as IAuditTrailService;
    if (AuditTrailService <> nil) then
        AuditTrailService.ObjectDeleted(AsIObject,
        “Peter Morris”);
end;
```

[C#]

```
public void PreDelete()
{
    IAuditTrailService auditTrailService;
    auditTrailService = (IAuditTrailService)
```

```
        AsIObject().ServiceProvider.GetEcoService(
        typeof(IAuditTrailService));
    if (auditTrailService != null)
        auditTrailService.ObjectDeleted(AsIObject(),
        "Peter Morris");
}
```

19 Appendix Services.A

Name	Source	Parameters	Result
<	<Any>	<Any>	System.Boolean
<=	<Any>	<Any>	System.Boolean
>	<Any>	<Any>	System.Boolean
>=	<Any>	<Any>	System.Boolean
<>	<Any>	<Any>	System.Boolean
=	<Any>	<Any>	System.Boolean
+	System.Double	System.Double	System.Double
-	System.Double	System.Double	System.Double
*	System.Double	System.Double	System.Double
/	System.Double	System.Double	System.Double
allInstances	<Type>		(Collection<Instances of source type>)
and	System.Boolean	System.Boolean	System.Boolean
average	Collection(System.Double)		System.Double
difference	Collection(<Any>)	Collection(<Any>)	<Same as source>
div	System.Int32	System.Int32	System.Int32
exists	Collection(<Any>)	System.Boolean	System.Boolean
forAll	Collection(<Any>)	System.Boolean	System.Boolean
implies	System.Boolean	System.Boolean	System.Boolean
includes	Collection(<Any>)	<Any>	System.Boolean
intersection	Collection(<Any>)	Collection(<Any>)	Collection(<Lowest common class>)
isEmpty	Collection(<Any>)		System.Boolean
isNull	<Any>		System.Boolean
length	System.String		System.Int32
maxValue	Collection(System.Double)		System.Double
minValue	Collection(System.Double)		System.Double
mod	System.Int32	System.Int32	System.Int32
not	System.Boolean		System.Boolean
otEmpty	Collection(<Any>)		System.Boolean
oclAsType	<Any>	<Type>	<typecast object>
or	System.Boolean	System.Boolean	System.Boolean
orderBy	Collection(<Any>)	<Any>	<Same as source>
orderDescending	Collection(<Any>)	<Any>	<Same as source>
reject	Collection(<Any>)	System.Boolean	<Same as source>
select	Collection(<Any>)	System.Boolean	<Same as source>
size	Collection(<Any>)		System.Int32
sqlLike	System.String	System.String	System.Boolean
sqlLikeCaseInsensitive	System.String		System.String

sum	Collection(System.Double)		System.Double
toLower	System.String		System.String
toUpper	System.String		System.String
union	Collection(<Any>)	Collection(<Any>)	Collection(<Lowest common class>)

Index

U

V

W