

# Persistence Brings Success - Pt 2

by Joanna Carter

In my last article, I started to look at the benefits of using an Object Store to look after the persistence of Business Objects, rather than using TDataset derivatives and data-aware controls on forms. After an initial discussion of some of the pros and cons of both approaches, I went on to describe how we can inherit from TPDOject and TPDCollection to create classes that describe a business domain and that know how to CRUD (create/retrieve/update/delete) instances of those classes.

In this article, I will look at the classes of an Object Store that are used in the writing of an object-oriented application. I cannot claim total responsibility for the design of the Object Store that I am discussing here. A lot of the class names and behaviours are the idea of Scott Ambler who has written an excellent paper called 'The Design of a Robust Persistence Layer for Relational Databases'; but I did find that some of the ideas were either awkward to realise in Delphi or sometimes just conceptually difficult to grasp. I would recommend that you download Scott's paper from [www.ambysoft.com/persistence/layer.pdf](http://www.ambysoft.com/persistence/layer.pdf) and compare it with what I am talking about in these articles; should you find anything that doesn't gel, please contact me so we can all benefit from another view on this subject.

## Unknown Helpers

There are only a few out of the 50+ classes that make up an Object Store that actually get used in the building of an application; all the rest are written once for use within the storage layer and, after testing, should never need looking at again. To start with I will deal with the classes that are likely to be used, either within the code on a Form, or preferably in a Manager class (a Manager being the equivalent of a Form for use when you are dealing with HTML or other non-form ways of interacting with a system).

### \* TPDTTransaction

This class allows us to bundle together several operations on related objects, within the context of a single transaction that can be tried and if successful, committed or rolled back. Because we are dealing with the idea of transactions within the Object Store and because an Object Store can broker more than one database, we can even try the Transaction over several databases and handle any exceptions that might occur in one database by rolling back all connected databases. Likewise, Referential Integrity can be enforced within one class that is related to other classes without ever needing to set up RI in the database.

```
procedure TCustomerManager.CommitChanges;
var
  Tx: TPDTTransaction;
begin
  Tx := TPDTTransaction.Create;
  with TPDTTransaction.Create do
  try
    if fComments.Count > 0 then
    begin
      fComments.First;
      while not fComments.IsDone do
      begin
```

```
        if osModified in
          fComments.CurrentItem.ObjectState then
          AddStoreObject(fComments.CurrentItem);
          fComments.Next;
        end;
      end;
    if osModified in fCustomer.ObjectState then
    AddStoreObject(fCustomer);
    if TaskCount > 0 then
    begin
      if ProcessTransaction then
        Commit;
      else
      begin
        Rollback;
        raise
          ECustomerException.Create('Customer', 'Error
          saving');
        end;
      end;
    finally
      Free;
    end;
  end;
end;
```

This is an example of how a TPDTTransaction can be used to group together several operations in one Transaction. After creating the Transaction, we then iterate through the Comments that are related to this Customer and add them to the Transaction only if they have been modified; then, if the Customer has been modified, we add that to the Transaction as well. If any of the Comments cannot be stored successfully, an exception will be raised which will cause ProcessTransaction to stop processing and return False. In this case Rollback will be called and all of the previously tried Comments will be rolled back; the remaining Comments, as well as the Customer, will not get written to the underlying storage.

Here is the public interface to TPDTTransaction :

```
TPDTTransaction = class(TPDTTask)
public
  procedure AddStoreObject(AObject: TPDOject);
  procedure AddDeleteObject(AObject:
    TPDOject);
  procedure AddRetrieveObject(AObject:
    TPDOject);
  procedure AddTransaction(ATransaction:
    TPDTTransaction);
  procedure Commit;
  function CurrentTask: TPDTTask;
  procedure First;
  function IsDone: Boolean;
  procedure Next;
  function ProcessTransaction: Boolean;
  function Retry: Boolean;
  procedure Rollback;
  property TaskCount: Integer
    read GetTaskCount;
end;
```

As you can see, it is possible to add objects that need storing, deleting or retrieving as well as nesting another transaction within the current one. I have followed Scott Ambler's suggestion that if a sub-transaction succeeds before something else in the main transaction fails, the sub-transaction will not get rolled back; but of course you could also arrange for everything to rollback.

This ability to do an all-or-nothing operation allows us to use this mechanism in place of triggers and stored procedures in a SQL database. So now we never have to write the same logic again, just because we have to change the database from Oracle to Interbase to SQL Server to...

Some RDBMSs allow us to enforce Referential Integrity, but only in a limited way; we are allowed to specify whether a Delete or Update should either: not affect foreign key records, cascade the delete or update, set the foreign keys to a default value or set them to null.

Using transactions combined with, say, overriding the Delete method of an object, we can not only allow the RI operations above, but we can also give the user the opportunity to choose what to do with, for example, Customers that were looked after by the Sales Rep that is about to be deleted. This then allows the possibility of assigning these Customers to another Sales Rep; if there were any restriction as to why a certain Customer could not be transferred, then alternative action could be taken, whilst all the rest of the Customers would be added to a single transaction for updating along with the Sales Rep for deletion. Only if all the Customers could be successfully transferred, would the transaction continue on to delete the Sales Rep. The rules for handling these Use Cases or Scenarios need only be written once in the Business Rules classes and would never need changing, even if the underlying database or the user interface changed.

## \* TSelectionCriteria

The only other classes that get used in actual application code are those in the TSelectionCriteria hierarchy and TOrderBy; both used in the construction of meaningful Collections of Objects. Here is the public interface to the TSelectionCriteria base class :

```
TSelectionCriteria = class
protected
  function Operator: string; virtual; abstract;
public
  constructor Create(AClass: TPDObjectClass;
AttrName: string; Value: array of const;
IsOrCriteria: Boolean = False); reintroduce;
virtual;
  destructor Destroy; override;
  procedure AddOrCriteria(ACriteria:
TSelectionCriteria); virtual;
  property AsSQLClause: string
    read GetAsSQLClause;
  property IsOr: Boolean
    read fIsOr;
end;
```

It is from this base class that we can derive a variety of different Criteria to cope with most queries; usually all that needs altering is the constructor, and the protected Operator method that is responsible for determining what goes between the property and the value in something like a SQL where clause. Here are most of the varieties of criteria that I use :

```
TEqualsCriteria = class(TSelectionCriteria)
TGreaterThanCriteria =
  class(TSelectionCriteria)
TGreaterThanOrEqualToCriteria =
  class(TSelectionCriteria)
TLessThanCriteria = class(TSelectionCriteria)
TLessThanOrEqualToCriteria =
  class(TSelectionCriteria)
```

```
TStringMatchCriteria =
  class(TSelectionCriteria) // abstract class
TStartingCriteria = class(TStringMatchCriteria)
TContainingCriteria =
  class(TStringMatchCriteria)
TBetweenCriteria = class(TStringMatchCriteria)
TIsNullCriteria = class(TSelectionCriteria)
TNotNullCriteria = class(TIsNullCriteria)
```

And here is a typical query designed to retrieve a Stock List that matches the following :

- Stock Code begins with the string in Value  
AND
- Must be located in Bins 1 – 99  
OR
- Qty must be more than the Minimum value

```
StockList.AddCriteria(TStartingCriteria,
'Code', [Value]);
with StockList.AddCriteria(TBetweenCriteria,
'Bin', [1, 99]) do
  AddOrCriteria(TGreaterThanCriteria.Create
    (StockList.ItemClass, 'Qty', [Minimum]));
end;
StockList.AddOrderBy('Code');
StockList.Connect;
```

## \* TOrderBy

The TOrderBy class is very simple and the interface to it is as follows :

```
TOrderBy = class
public
  constructor Create(AClass: TPDObjectClass;
APropertyName: string; ADirection:
TOrderByDirection = obdAscending);
  property Direction: TOrderByDirection
    read fDirection;
  property ColumnName: string
    read GetColumnName;
  property PropertyName: string
    read fPropertyName;
end;
```

Due to the lack of a 'friend' construct in Delphi, I have to declare properties like AsSQLClause and the ColumnName property in the public section of their classes, otherwise the class that builds the query SQL could not see it. The 'friend's' ability to see normally private and protected parts of a class can only be implemented in Delphi by placing the related classes in the same unit. This is neither practical or desirable bearing in mind the large number classes that would also end up in the same unit through inheritance. Otherwise this property would have been hidden from everyday use except by the SQL classes; however the ColumnName property should never need to be used in an application's code. It was minor annoyances like this that were the biggest obstacles to realising parts of Scott Ambler's design with any sense of integrity. It is very important that a balance is struck between getting OO analysis right and going to such extremes of purity that nothing ever gets written or needs more code than sense would dictate.

A typical use of TOrderBy is in the previous example for Selection Criteria; all that is required is the name of the property by which the collection is to be sorted. Multiple TOrderBy can be added and will simply become sub-sorts of the first property specified.

## I Did It My Way

The Object Store is a brokering mechanism that mediates between Business Objects in an application, and one or more mechanisms that are responsible for maintaining the state of those Objects between closing the application and re-opening it.

Although it would be possible to use flat files, RDBMSs, ODBMS, etc to achieve persistence, I have concentrated on using SQL RDBMSs as a primary mechanism, mainly due to the fact that the majority of people are already using them and a reasonable mapping can be achieved between Objects and Collections and records and tables. Having said that, very little use is made of anything other than the most elementary SQL statements. As has previously been demonstrated, complicated operations tend to be controlled within Business Objects; this lack of complexity in a database means that Business Rules are maintained, regardless of what is happening to the flavour of the underlying database.

### \* TInterbaseMechanism

So what is my way of storing objects ? Well, most of the time I use InterBase (especially since v6 is free!) and so I had to write a class that knows all about connecting to an InterBase database. There is one interface that has to be supported by any potential storage class that is going to be managed by a Persistence Broker. IPersistenceMechanism defines what is required for the Persistence Broker to talk to a mechanism for storing objects :

```
IPersistenceMechanism = interface
  procedure Close;
  function CollectionIsConnected(ACollection:
    TPDCollection): Boolean;
  procedure CommitTransaction(ATransaction:
    TPDTransaction);
  procedure ConnectCollection(ACollection:
    TPDCollection);
  procedure DisconnectCollection(ACollection:
    TPDCollection);
  function GetNewObjectID: LongWord;
  function IsOpen: Boolean;
  procedure Open;
  procedure ProcessTransactionTask(ATask:
    TPDTask);
  procedure RollbackTransaction(ATransaction:
    TPDTransaction);
  procedure StartTransaction(ATransaction:
    TPDTransaction);
  function Supports(AClass: TPDBObjectClass):
    Boolean;
end;
```

This should be all that the Broker needs to talk to a storage class; however, because we are going to be using a Relational Database, a storage mechanism also needs to comply with another interface for use by those classes that are concerned with handling the translation between objects and SQL :

```
IRelationalDatabase = interface
public
  function GetClauseStringSelect: string;
  function GetClauseStringListField: string;
  function GetClauseStringFrom: string;
  function GetClauseStringWhere: string;
  function GetClauseStringOrderBy: string;
  function GetClauseStringAscending: string;
  function GetClauseStringDescending: string;
  function GetClauseStringEndBracket: string;
```

```
function GetClauseStringDeleteFrom: string;
function GetClauseStringInsertInto: string;
function GetClauseStringValues(AType:
  TPropertyType): string;
function GetClauseStringListValue(AType:
  TPropertyType): string;
function GetClauseStringUpdate: string;
function GetClauseStringSet(AType:
  TPropertyType): string;
function GetClauseStringEquals(AType:
  TPropertyType): string;
function GetClauseStringListEquals(AType:
  TPropertyType): string;
end;
```

You may find that there are other SQL strings that become necessary, but this should be the only way that you get hold of SQL for use in the derived Persistence Mechanism.

Therefore, to write a Persistence Mechanism for use with InterBase could look something like :

```
type
  TInterbaseMechanism =
  class(TInterfacedObject, IPersistenceMechanism,
    IRelationalDatabase)
    ....
  end;
```

Or you may wish not to use interfaces; in which case you would substitute TPersistenceMechanism for IPersistenceMechanism and TRelationalDatabase for IRelationalDatabase, all methods would become abstract and you would inherit like this :

```
type
  TPersistenceMechanism = class
    ....
  end;

  TRelationalDatabase =
  class(TPersistenceMechanism)
    ....
  end;

  TInterbaseMechanism =
  class(TRelationalDatabase)
    ....
  end;
```

TInterbaseMechanism would then be one of only two classes that knows anything about ways of talking to InterBase, whether it be FreeIB, IBObjects or IBX. The other class that needs to know about InterBase is not part of Scott Ambler's design, but something I put together to handle the provision of data for a collection; it also allows for such nice features as on demand loading of objects and read ahead (by supplying data in chunks, as discussed in my previous article).

### \* TInterbaseCollectionBroker

This is the other class that directly connects to an InterBase database; it derives from TPDCollectionBroker which is almost a completely abstract class :

```
TPDCollectionBroker = class
private
  fCollection: TPDCollection;
protected
  procedure GetFirstHandler; virtual; abstract;
  function GetItemClassHandler(ID: TOID):
    TPDBObjectClass; virtual; abstract;
  procedure GetNextHandler(var ChunkSize:
    Integer); virtual; abstract;
```

```

    procedure IsDoneHandler(var IsDone: Boolean);
    virtual; abstract;
    public
        constructor Create(ACollection:
        TPDCollection); virtual;
        procedure Refresh; virtual; abstract;
        property Collection: TPDCollection
            read fCollection;
    end;

```

This is a declaration of all the behaviour that is required to handle the brokering of objects into a collection from any type of storage. It mainly consists of four event handlers that respond to activity in the collection. The collection has no idea how it is going to get its data, all it knows is that it has to call these events and hope that somebody is listening. Therefore, all that is known about the collection at this level of the broker hierarchy is that it has these four events; the constructor looks like this :

```

    constructor
    TPDCollectionBroker.Create(ACollection:
    TPDCollection);
    begin
        inherited Create;
        fCollection := ACollection;
        fCollection.OnFirst := GetFirstHandler;
        fCollection.OnGetItemClass :=
        GetItemClassHandler;
        fCollection.OnIsDone := IsDoneHandler;
        fCollection.OnNext := GetNextHandler;
    end;

```

It is the responsibility of the GetFirstHandler to add a first object to the empty collection, whilst the GetNextHandler must check that there is any more data to read and, if there is, to add objects, in chunks, to the end of the collection whenever it is called. The GetIsDoneHandler, logically, responds to requests from the collection by saying whether there is any more data left to read from the underlying storage.

Because the Object Store has the capability of retrieving collections of inherited objects, this means we can retrieve a list of, for example, TPerson regardless of whether they are TStudent, TCustomer, TStaff or TManager. The GetItemClassHandler is called by the collection when it wants to create new objects that may be of a class derived from the base class that the collection is retrieving. This then allows us to use the class of each object in the list to create an appropriate editing form from a Form Factory.

## Conclusion

This article looked at some of the classes involved in using an Object Store in applications. We started by looking at three 'helper' classes that are used when manipulating objects and collections in Forms or Manager classes :

- TPDTransaction, which is responsible for managing related operations on groups of objects.
- TSelectionCriteria, which is used for limiting the scope of objects retrieved in a collection.
- TOrderBy, which is used for specifying the way collections are sorted.

Apart from domain classes that inherit from TPDOject, collection classes that inherit from TPDCollection, Manager and Factory classes that encapsulate business logic, these will tend to be the only classes that you deal with in the everyday writing of an application.

The two other classes that we discussed are both involved with interacting directly with an InterBase database, used to provide a means of persisting our business objects between executions of our application.

- TInterbaseMechanism, which is designed to run InterBase SQL queries that allow us to Create / Retrieve / Update / Delete single business objects.
- TInterbaseCollectionBroker, which manages the provision of objects into a collection on an as required basis, by maintaining a SQL cursor on a multi-row result set.

Please let me have feedback on these articles, as I am finding that the more people I discuss this subject with, the more mature this design of Object Store becomes.



*Joanna Carter is a writer, developer and trainer specialising in requirements-driven training and consultancy. Joanna is involved with mentoring several companies in the setting up of mapping object-oriented systems into relational databases. You can e-mail her at [JoannaC@btinternet.com](mailto:JoannaC@btinternet.com) and her web site is at [joannac.btinternet.co.uk](http://joannac.btinternet.co.uk).*