Patterns And Persistence

Using design patterns to create a persistence framework

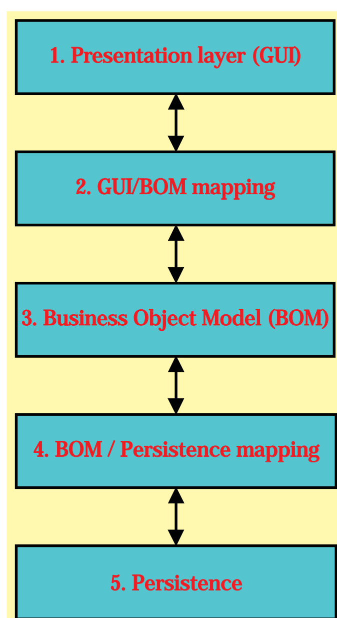
by Peter Hinrichsen

The tools that come with Delphi can be used to build a database application very quickly. The combination of the BDE alias, `TDatabase`, `TQuery`, `TDataSource` and `TDBEdit` has incredible power. The problem, though, is that with every `TDatabase` or `TQuery` you drop on a `TDataModule`, you have tied yourself more closely to the BDE. With every `TDBEdit` added to a form, you have coupled yourself to a specific field name in a specific database.

The alternative is to roll your own persistence layer. This is hard work and will take hundreds of hours before it comes close to matching the functionality of what comes out of the box with Delphi. However, if the business case justifies this work, then the results can be stable, optimised and versatile, plus extremely satisfying to build.

I will discuss some of the problems with data aware controls, then take a high level look at the visitor framework as an alternative, delving into the detail of the persistence layer which I have written, which is constructed around the iterator and visitor design patterns.

➤ *Figure 1*



I Hate Data Aware Controls...

A couple of months ago there was a discussion on the Australian Delphi User Group's mailing list (www.adug.org.au) on the topic of developing your own persistence framework versus using data aware controls. Many posts were made and everyone seemed to have a strong opinion one way or the other. Most agreed there was a place for data aware controls in single-tier file-based applications or client/server prototypes. Many experienced client/server developers agreed that there was no room for data aware controls in sophisticated client/server applications.

Here are four problems with data aware controls.

Tight coupling to the database design. The database layer is very tightly coupled to the presentation layer: any change to the database means that changes might be needed in many places through the application. It is often hard to find where these changes must be made, as the links from the data aware controls in the application to the database are made with published properties and the Object Inspector. This means that the places to make changes cannot be found with Delphi's search facility. Also, the amount of code checking done by the compiler is reduced. This means that some bugs may not be detected until runtime, or may not be detected at all until the user navigates down a path the developer did not foresee. Developing a persistence framework allows you to refer to a data object's values by property name rather than by using a `DataSet`'s `FieldByName` method. This gives greater compile-time checking and leads to simplified debugging.

Data aware controls create more network traffic. It is a simple exercise to drop a few data aware controls on a form, connect them to a `TDataSource`, `TQuery` and `TDatabase`, then load the DBMonitor program (sqlmon.exe) and watch the excess network traffic they create. There is a good article on how the network monitor hooks into the BDE's API in the August 1999 issue. A custom persistence framework can be optimised to reduce this superfluous network traffic and is a topic large enough to justify a dedicated article.

Tight coupling to vendor specific database features. At the simplest level, all SQL-aware databases accept the same SQL syntax. For example, a simple, `select * from customers` will work for all the systems I have come across. As you become more sophisticated with your SQL, you will probably want to start using special functions, a stored procedure, or perhaps an outer join, which will be implemented differently by each database vendor. Data aware controls make it difficult to build your application so it can swap between databases seamlessly.

Tight coupling to a data access API. The BDE allows us to swap aliases when we want to change databases, but what if you want to switch from BDE data access to ADO, IBObjects, Direct Oracle Access (DOA), `TClientDataSet` or a custom data format? This is not the fault of the data aware controls, but is still a problem with the component-on-form style of developing. A custom persistence framework can be designed to eliminate this tight coupling of an application to a data access API.

Roll Your Own Framework

These four problems are all addressed by building your own persistence framework, but it is incredibly hard work. Danny Thorpe was presenting at the 1999 Borland Conference in Australia and several of us had a chat about roll-your-own persistence frameworks. Danny was amazed that

anybody would go to all that trouble, considering Borland had done such a great job with Delphi. I believe he is right, except when you want true database vendor independence, or if your business rules are very complex. See Table1.

The framework we will develop consists of five layers, as shown in Figure 1. At the centre of the framework is layer 3, the Business Object Model (BOM). In the address book application we will build as an example, there is a list of `TPerson` objects. Each person has a list of addresses and phone numbers.

The persistence layer (layer 5) comprises a family of `TQuery` objects that are responsible for reading and saving data to and from the database. The BOM/persistence layer (layer 2) maps data from the business object layer (layer 3), to and from the `TQuery`(s) in layer 5. The presentation layer (layer 1) displays the data using a `TreeView`, `ListView`, `Edits` or a graph, and allows the user to interact with the data. The GUI/BOM mapping layer (layer 4) maps data from level 3 (the BOM) into the GUI and also manages the saving of changes back into the BOM.

To change database vendors or database types, we simply replace the persistence layer (layer 5). This layer can even be distributed as a runtime package so this switching can be done at runtime: a very powerful technique.

Design Targets

As an example, we will construct an address book application to store names and contact details. Our application must be flexible enough to cater for new address types without any re-engineering. We need to allow for two types of addresses: postal addresses and electronic addresses, such as phone, fax, email and website.

Our presentation layer has to have an Explorer/Outlook look and feel, making extensive use of Microsoft's TreeView and ListView common controls. The application must perform well and not have the look and feel of a conventional, form-based, client/server app.

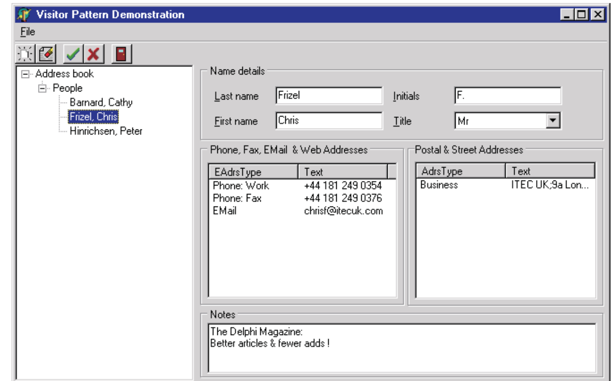
The data is to be stored using Interbase and accessed using the BDE, but the framework must be flexible enough to allow us to move to `IBObjects`, `ClientDataSet` or a custom file format with little work. Our application must be easy to move to multi-tier should this become a requirement.

To achieve our design goals, we use the iterator and visitor patterns to build a persistence framework. The main form of our application is shown in Figure 2.

A right mouse click on the tree will let you add or delete a person. A right click on either of the list views will open a modal dialog and let you edit, insert or delete an address or e-address.

Business Object Model

The core of the address book application is its business object model layer and the UML for this



➤ *Figure 2*

➤ *Table 1: Data aware controls or a persistence framework?*

	Advantages	Disadvantages	When Do I Use?
Data aware controls	Good for prototypes. Good for simple, single tier apps. Good for seldom used forms, like one-off setup screens that may be used to populate a new database with background data.	Higher maintenance and debugging costs. Higher network traffic. Limited number of data aware controls in Delphi (but plenty if you use third party libraries). Can't be used to edit custom file formats, registry entries or data not contained in a TDataSet. Hard to develop your own data aware controls. Difficult when the database does not map directly into the GUI, ie a well normalised database. Extensive code reuse is difficult.	Low end customers (small businesses with few users). Throw away prototypes. Data maintenance apps that my customers will not see. Systems where I have total control over the database design. When the user wants the app to look and perform as if it were written in VB.
Persistence framework	Good for complex applications. Lower network traffic. Lower total cost of ownership. When the database is storing non text data like multimedia, or data which must be manipulated with complex algorithms. Decouple GUI from database.	More skilled development team. Higher up front development cost. Many reporting tools take input from a TDataSet. Some extra code would be needed to connect the persistence framework to the reporting tool. Must re-build what comes out of the box with Delphi.	High end (corporate) customers with many users where performance is important. Systems with complex data models that I have little control over. Systems that require a TreeView, ListView look and feel. Systems that must be database vendor independent.

layer is shown in Figure 3 which shows the public properties and the relationships between the main classes.

The classes that are displayed in the TreeView or ListView all have a `Caption` property, used in conjunction with RTTI to map the BOM into the GUI, as discussed below.

At the top of the tree of containment is the `TAdrsBook` class that owns a single `TPersonList`. The `TAdrsBook` class was added so we could implement a list of companies as well as a list of people, in the future. The `TPersonList` contains one `TPerson` object for each person in the database. The `TPerson` class has some 'flat' information like `last_name` and `first_name`, and also a list of `TAddress`(es) and of `TEAddress`(es). Each instance of `TAddress` holds information like address type, street, town, etc, while each instance of `TEAddress` holds an electronic address.

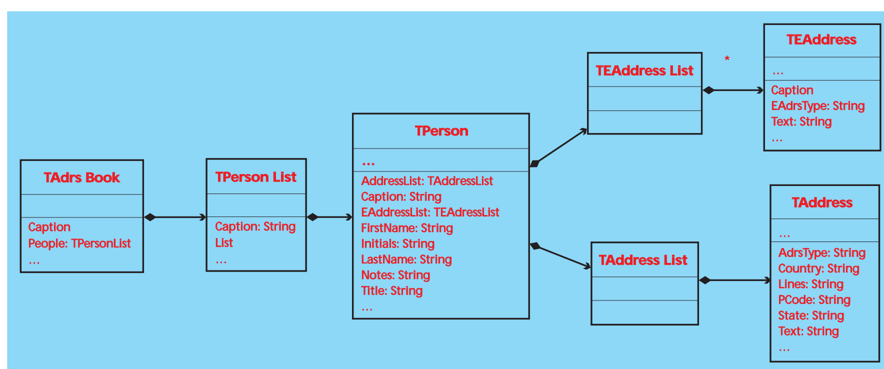
Figure 4 shows the inheritance of our business object model with Delphi's `TPersistent` class at the top of the tree. I chose `TPersistent` as the base class because it gives access to RTTI, which will enable us to construct generic `TTreeView` and `TListView` controls to browse the business object model. Next comes the `TVisitedAbs` class, which introduces abstract methods that allow us to process visitors (more on this later). At the next level are the `TPerObjAbs` and `TVisList` classes. The `TPerObjAbs` (short for Persistent Object Abstract) adds functionality required when saving data. The `TVisList` is a `TList` replacement that knows how to iterate over all its items. Lastly we have the container classes (`TPersonList`, `TAddressList` and `TEAddressList`) and data classes (`TAddressBook`, `TPerson`, `TAddress` and `TEAddress`).

Database Structure

For this example, we'll be saving our data to an Interbase database. The structure of the main tables maps directly to the class hierarchy. The SQL to create the database is shown in Listing 1 (which is on the disk with all the code).

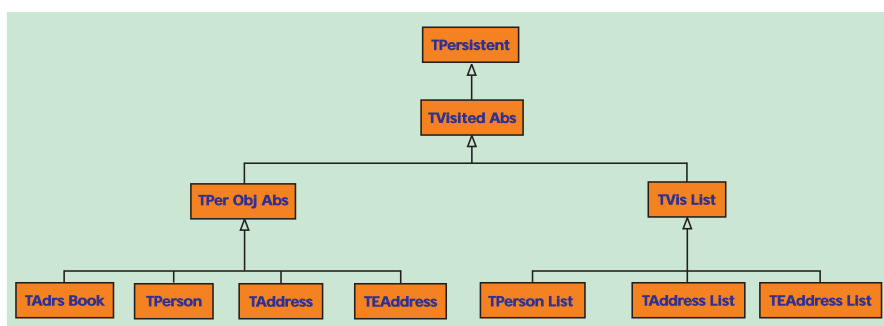
There are several important things to note about how we map our object oriented business model into the relational database.

First, classes map to tables. For each persistent class, there is a corresponding table. This is simple to achieve with our example, as there is no inheritance within the persistent classes. Modelling inheritance in a relational database is hard work.



➤ Above: Figure 3

➤ Below: Figure 4



Secondly, properties map to columns. For each 'flat' property, there is a corresponding column in that class's table.

Lastly, objects in memory are uniquely identified by their memory address, but this is of no use to us when we are trying to persist our objects. To overcome this problem, we add the Object ID (OID) property high up in the class hierarchy, then use the OID to locate the correct record in the database for updates and deletes. This is discussed in a paper referenced at the end of the article.

An OID must be generated by Delphi when a new instance of a class is created. One way to do this is to create a sequence generator in the database, and reference this as each object is created. This, however, will mean a round trip to the database every time a new object is created. Also, sequence generators differ between database vendors, so we have created the table `NEXT_OID`, which is dedicated to holding the next OID number. The details of this technique are shown later.

Patterns We Shall Use

The persistence framework is based on the three patterns shown in Table 2.

The Iterator. The BOM comprises a tree of people, addresses and e-addresses. To save the elements of this tree to a database we must visit all the nodes of the tree, identify those that have changed, then execute SQL to insert, update or delete the data from the database. To ensure this process touches all nodes of the tree, we start at the top of the tree and the child nodes are processed automatically. This functionality is managed by a class we call `TVisitedAbs`.

The Visitor. In this example, we are persisting to a relational database, so each class in our object model will have a family of SQL strings to manage the interaction with the database. For example, the `Person` class will have some SQL to create a new person, delete or update an existing person. There will also be SQL to read all people matching certain search

```
// Create an Object ID (OID) domain
create domain domain_oid as integer not null ;
// Create a domain to hold info about the address type
create domain domain_type as varchar( 20 ) not null ;
// Create a table to allow the generation of new OIDs
create table Next_OID ( next_oid domain_oid ) ;
insert into next_oid
  ( next_oid )
values
  ( 100 ) ;
// Create a table to hold information about people
create table person
  ( oid        domain_oid,
    last_name  varchar( 60 ),
    first_name varchar( 60 ),
    title      varchar( 10 ),
    initials   varchar( 10 ),
    notes      varchar( 250 ),
    primary key ( oid )) ;
// Create a table to hold street addresses
create table adrs
  ( oid         domain_oid,
    owner_oid   domain_oid,
    adrs_type   domain_type,
    lines       varchar( 180 ),
    state       varchar( 20 ),
    pcode       varchar( 10 ),
    country     varchar( 20 ),
    primary key( oid ),
    foreign key( owner_oid ) references person (oid )
      on delete cascade ) ;
// Create a table to hold phone, fax, EMail addresses
create table EAdrs
  ( oid         domain_oid,
    owner_oid   domain_oid,
    eadrs_type  domain_type,
    text        varchar( 60 ),
    primary key( oid ),
    foreign key( owner_oid ) references person (oid )
      on delete cascade ) ;
```

➤ *Listing 1*

```
// Example of passing a visitor to a class to be visited
Procedure Save ;
Var lVisSaveAddress : TVisSaveAddress ;
Begin
  lVisSaveAddress := TVisSaveAddress.Create ;
  try
    FPerson.Iterate( lVisSaveAddress ) ;
  finally
    LVisSaveAddress.Free ;
  end ;
End ;
// Example of the iterate method of the person class
Procedure TPerson.Iterage( pVisitor : TVisitorAbs ) ;
var i : integer ;
Begin
  // Execute the visitor against the person object
  pVisitor.Execute( self ) ;
  // Execute the visitor against all the address objects owned by the person
  for i := 0 to FPersonList.Count - 1 do
    pVisitor.Execute( TPerson( FPersonList.Items[I] )) ;
End ;
```

➤ *Listing 2*

conditions into a list. This SQL performs the basic create, read, update and delete functionality common to most databases. The address book has three persistent classes (person, address and e-address), so there will be 12 SQL statements, 12 `TQuery` objects and 12 classes to manage the mapping of the business objects to the `TQuery` objects. The visitors are responsible for mapping the business objects to the appropriate SQL statement and executing the SQL against the database. This functionality is encapsulated in the `TVisAbs` and `TVisDB` classes.

The Factory (`TVisitorMgr`*).* This is a fairly simple example with only three classes being persisted, but there are 12 visitor classes to manage this persistence. The main disadvantage of using the visitor pattern to implement a persistence framework is that it leads to a proliferation of classes. To help manage these classes, we need to create a visitor manager which will be responsible for creating, caching and freeing the visitor objects as required. A modified factory will be used to create the visitors on demand, store them in a cache once they have been created, cause them to be iterated over the BOM in the correct order, manage database transactions, and free them when the application closes.

The Iterator

Once we have created the family of visitors to manage persistence, we must devise a way of passing each of these visitors to every node of the BOM. There are a number of ways to achieve this functionality, but after some months of using visitors and iterators, I settled on the idea of defining a method called `Iterate` that accepts a single parameter of type `TVisitorAbs` in a base class.

For example, say we have an instance of the `TPerson` class called `FPerson`, and an instance of the `TVisSaveAddress` class called `lVisSaveAddress`. We want to pass the visitor to the class at the top of the tree, and have it passed over all the child nodes of that object. The code in Listing 2 shows this.

You can imagine that it is a time-consuming and error-prone process to write an iterate method on every class in the BOM. Each time a `TList` property is added or removed from a class, its `Iterate` method must be modified. After a week or two of debugging frustration, I decided to rewrite the `Iterate` method using RTTI so it would automatically process all list properties (as long as they are published.) Listing 3 shows the code for the modified `Iterate` method.

This method performs four tasks. After asserting that the passed visitor is not nil, it executes the visitor against the current object (`self`). Next, a `TStringList` is created, and Delphi 5's new 'Easy Access RTTI' is used to populate the list with the names of all the properties of type `TObject`. The `tiGetPropertyNames` procedure

➤ *Table 2: Patterns used.*

Pattern Name	Intent (from 'Gang of Four' book)
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitors let you define an operation without changing the classes of the elements on which it operates.
Factory (Visitor Manager)	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

```
// Pass a visitor object to all contained list or object
// properties.
procedure TVisitedAbs.Iterate(pVisitor: TVisitorAbs) ;
var
  lsl      : TStringList ;
  i,j      : integer ;
  lVisited : TObject ;
begin
  // Before processing, confirm that nil was not passed
  Assert( pVisitor <> nil, 'Visitor unassigned' ) ;
  // Execute the visitor against self
  pVisitor.Execute(self);
  // Use RTTI to scan through all properties of type TList
  // Create a string list to hold the property names
  lsl := TStringList.Create ;
  try
    // Get all property names of type tkClass
    tiGetPropertyNames( self, lsl, [tkClass] ) ;
    for i := 0 to lsl.Count - 1 do begin
      // Get a pointer to the property
      lVisited := GetObjectProp( self, lsl.Strings[i] ) ;
      // If the property is a TVisitedAbs, then visit it.
      if ( lVisited is TVisitedAbs ) then
        TVisitedAbs( lVisited ).Iterate( pVisitor ) ;
      // If the property is a TList, then visit it's items
      if (lVisited is TList ) then
        for j := 0 to TList( lVisited ).Count - 1 do
          if (TObject(TList(lVisited).Items[j]) is
            TVisitedAbs) then
            TVisitedAbs(
              TList(lVisited).Items[j]).Iterate(pVisitor);
    end ;
  finally
    lsl.Free ;
  end ;
end ;
```

➤ *Listing 3*

takes a `TPesistent`, a `TStringList` and a set containing the types of the properties to be returned. The code for `tiGetPropertyNames` on the disk is worth a look if you want to explore Delphi 5's simplified RTTI.

Now we have a list of properties that are objects, we can scan the list and get a pointer to the instance of the object pointed to by these properties. If the property is a `TVisitorAbs`, then the visitor is executed with the property as a parameter. If the property is a `TList` its list members are iterated.

Abstract Persistent Object

All the classes that must be saved in some way have several things in common, so they all descend from the same abstract. The interface of `TPerObjAbs` is shown in Listing 4. The most important properties are OID, a unique integer used to identify a particular instance of an object in the database. The `ObjectState` property is of type `TPersistentObjectState`, an ordinal type with the possible values shown in Listing 5.

The constructor `Create` adds one extra line that sets the `ObjectState` property to `posClean`. The additional constructor `CreateNew` calls the main constructor `Create` but adds the additional functionality to call the OID generator to return a new and unique OID.

The property `Owner` is an optional back-pointer to the owning object. This is useful when you need to know who owns a child object, and is used in much the same way as a `TComponent`'s `Parent` or `Owner` property. The `Owner` property is necessary when a child object is being

```
TPerObjAbs = class( TVisitedAbs )
private
  FIntOID : integer ;
  FObjectState : TPersistentObjectState ;
  FOwner: TPerObjAbs;
  function  GetDeleted: boolean;
  procedure SetDeleted(const Value: boolean);
  function  GetDirty: boolean;
  procedure SetDirty(const Value: boolean);
protected
  procedure SetOID(const Value: integer); virtual ;
public
  constructor Create ; override ;
  constructor CreateNew ; virtual ;
  property    OID : integer read FIntOID write SetOID ;
  property    ObjectState : TPersObjState read FObjectState write FObjectState ;
  property    Owner : TPerObjAbs read Fowner write FOwner ;
  property    Deleted : boolean read GetDeleted write SetDeleted ;
  property    Dirty   : boolean read GetDirty write SetDirty ;
end ;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
TPersistentObjectState =
( posCreate,  // The object is new and must be created in the DB
  posRead,    // The object has been created, but not filled with data
  posPK,      // The object has been created, but only it's primary key read
  posUpdate,  // The object has been changed, the DB must be updated
  posDelete,  // The object has been deleted, it must be deleted from the DB
  posDeleted, // The object was marked for deletion, and has been deleted
  posClean    // The object is 'Clean' no DB update necessary) ;
```

saved to a database where there is a primary key/foreign key relationship and is used to determine the foreign key value.

The `Deleted` property is used when reading data into the GUI. An object may have been deleted from the user's point of view, but it has only been marked for deletion and it has not yet been removed from the database. Objects with a `Deleted` property equal to true will be excluded from the GUI.

`Dirty` is used to remind the user if any changes have been made. For example, the main form's `CloseQuery` event checks the `AddressBook` object's `Dirty` property and if true, asks the user if they want to save their changes. A `TPerObjAbs` object will be considered dirty if any of its child objects have been changed, so we need a way of iterating over the object tree and testing all child objects. This is achieved with a `Visitor` called `TVisPerObjIsDirty` and the `AcceptVisitor` and `Execute` methods are shown in Listing 6.

The `Delete` property has a `SetDelete` method that sets the `ObjectState` property to `posDeleted`, and triggers a visitor to scan all child objects and set their `ObjectState` to `posDeleted`. This is necessary when a `Person` is deleted as all the addresses and e-addresses must also be deleted. This can be achieved by implementing a cascading delete as part of the referential integrity set up in the database, but this is starting to create database dependencies so we avoid it here.

The Visitor

Listing 7 shows the interface and implementation sections of the `TVisitorAbs` class. There are three methods of interest: `Create`, `Execute` and `AcceptVistior`, which can be overridden in the descendant classes.

```
function TVisPerObjIsDirty.AcceptVisitor: boolean;
begin
  result := ( Visited is TPerObjAbs ) and ( not Dirty ) ;
end;
procedure TVisPerObjIsDirty.Execute(pVisited: TVisitedAbs);
begin
  inherited Execute( pVisited ) ;
  if not AcceptVisitor then exit ; //==>
  // If the visited object is marked to be created, updated
  // or deleted, then set Dirty to true.
  Dirty := TPerObjAbs( pVisited ).ObjectState in
            [ posCreate,   // The object is new
              posUpdate,   // The object has been changed
              posDelete];  // The object has been deleted
end;
```

➤ *Listing 6*

Create is no different to TObject.Create, except that it is declared as Virtual, which makes it easier to override and is also necessary when the class is instantiated from a factory. Execute is called with an instance of the class to be visited passed as a parameter and will probably be customised in the descendant class. AcceptVisitor is the function where the decision to process the object being visited is made.

We require two special visitor classes to handle relational database access and these are called TDBVisSelect (to read the results of an SQL SELECT statement into a TList) and TDBVisUpdate (to execute some create, update or delete SQL for the visited object.) Both these DB visitors are descendant from the TVisDBAbs abstract class. The interface of TVisDBAbs is shown in Listing 8. Most of the implementation is introduced in the concrete descendant classes, except for a TQuery that is created and freed in the classes' constructor and destructor. The DBConnection property is a pointer to a DBConnection that is set and cleared by the visitor manager. This allows a number of visitors to be processed in the context of one database transaction so a single error will prevent any data from being saved.

The Virtual methods Init, SetupParams, MapRowsToObject and UpdateObject are introduced but not implemented. Init is where the Query.SQL.Text property can be assigned according to the functionality required by the visitor. SetupParams is used when a select visitor requires a parameter. MapRowsToObject is used when a select visitor is reading data from a SQL result set into a list of objects. UpdateObject is called when an update visitor has been processed and the state of the visited object has been changed by the execution of the visitor. This will typically be called when create, update or delete SQL has been called and the object is no longer Dirty.

TVisDBSelect and TVisDBUpdate override the Execute method and introduce custom processing. The Execute method of TVisDBSelect is shown in Listing 9. The first step is to test if the visitor is to be accepted. Next, if the visitor has been accepted, the Init method is called. This will probably set the Query.SQL.Text property if it has not already been set. Next, SetupParams is called to map any parameters to the SQL, then Query.Open is called to execute the SQL. The rows of the result set are now scanned and MapRowsToObject is called for each row. This is where an object can be added to the list for each result set row.

The Execute method of TVisDBUpdate is shown in Listing 10 and simply calls the methods AcceptVisitor, Init, SetupParams, MapRowsToObject, Query.ExecSQL and UpdateObject in turn.

It is intended that TVisDBSelect takes a single object, with a TList property and reads the rows of an SQL select statement into the TList property. The TVisDBUpdate works the other way round and takes a single TPerObjAbs object and updates the database accordingly.

Visitor Manager

The final piece of the framework to look at is the visitor manager. The interface of the TVisitorMgr and its associated class, the TVisMapping, are shown in Listing 11. The VisitorMgr is based on the Factory Pattern that was discussed in the September edition of *The Delphi Magazine*. The VisitorMgr contains two methods that are important: RegisterVisitor and Execute.

```
  TVisitorAbs = class( TObject )
  private
    FVisited    : TVisitedAbs ;
  protected
    function    AcceptVisitor : boolean ; virtual ;
  public
    constructor create ; virtual ;
    procedure   execute( pVisited : TVisitedAbs ) ; virtual ;
    property    Visited    : TVisitedAbs read FVisited write FVisited ;
  end ;
constructor TVisitorAbs.create;
begin
  inherited create ;
end;
function TVisitorAbs.AcceptVisitor : boolean;
begin
  result := true ;
end;
procedure TVisitorAbs.execute(pVisited: TVisitedAbs);
begin
  Visited := pVisited ;
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
TVisDBAbs = class( TVisitorAbs )
private
  FQuery         : TQuery ;
  FDBConnection : TtiDBConnection;
  procedure    SetDBConnection(const Value: TtiDBConnection);
protected
  function     AcceptVisitor : boolean ; override ;
  procedure    Init              ; virtual ;
  procedure    SetupParams       ; virtual ;
  procedure    MapRowsToObject ; virtual ;
  procedure    UpdateObject      ; virtual ;
  property     Query : TQuery read FQuery ;
public
  constructor Create ; override ;
  destructor  Destroy ; override ;
  property DBConnection: TtiDBConnection
    read FDBConnection write SetDBConnection;
end;
```

```
procedure TVisDBSelect.Execute(pVisited: TVisitedAbs);
begin
  // Call inherited to set Visited property making it accessable to other methods
  inherited Execute( pVisited ) ;
  try
    // Test if the visitor is to be executed against this visited object
    if not AcceptVisitor then exit; //==>
    if not InitCalled then Init;    // If necessary, then call Init
    SetupParams;    // Map any parameters into the Query
    Query.Open ;    // Execute the Query
    // Scan result set and call MapRowsToObject for each row
    while not Query.EOF do begin
      MapRowsToObject ;
      Query.Next ;
    end ;
    UpdateObject;   // Set object's ObjectState to reflect its changed state
  finally
    Visited := nil ;
  end ;
end;
```

➤ *Listing 9*

```
procedure TVisDBUpdate.Execute(
  pVisited: TVisitedAbs);
begin
  inherited Execute( pVisited ) ;
  try
    if not AcceptVisitor then exit;
    if not InitCalled then Init;
    Init;
    SetupParams;
    MapRowsToObject;
    Query.ExecSQL;
    UpdateObject;
  finally
    Visited := nil ;
  end ;
```

➤ *Listing 10*

RegisterVisitor takes two parameters: a string to identify the process and a class reference to identify the visitor being registered. The code in Listing 12 registers three visitors to manage the persistence of the TAddress class. When RegisterVisitor is called, an instance of TVisMapping is created and added to the visitor manager's list. When the visitor manager's Execute method is called, the list of TVisitorMapping(s) is scanned and each matching TVisitorMapping has its DBConnection property assigned, then its Execute method called. The TVisitorMgr.Execute method is shown in Listing 13.

Listing 14 shows the TVisMapping.Execute method, which performs four main functions. First, if the visitor property of the TVisMapping (remember, this is a cache of visitors) is nil, then it must be created by calling the virtual constructor on the class reference. Next, if the visitor is a TVisDBAbs descendant, and hence is managing database persistence, its DBConnection property will be set. Thirdly, the visitor is passed to the pVisited.Iterate method and finally, the DBConnection property is set to Nil.

This completes the core classes in the visitor framework, which manages database persistence. In summary, you can see the classes in Table 3.

The relationship between the TVisitedAbs and TvisitorAbs and their descendants is shown in the UML class diagram in Figure 5. The visitor manager framework is shown in Figure 6. We shall now look at how persistence is implemented using this framework.

Implementing Persistence

To illustrate this framework in action, we'll look at reading a list of TPerson(s) from the database, then saving any changes back to the database. To improve performance, the initial read pass only reads primary key (OID) information, and enough text to display the person's name in the TreeView. When the node on the tree view is clicked, the detail is

```
TVisMapping = class( TObject )
private
  FsGroupName : string ;
  FClassRef   : TVisClassRef ;
  FVisitor    : TVisitorAbs ;
  FDBConnection: TtiDBConnection;
public
  constructor CreateExt(const psGroupName: string;
    const pClassRef: TVisClassRef);
  property    GroupName : string read FsGroupName write FsGroupName ;
  property    ClassRef  : TVisClassRef read FClassRef write FClassRef ;
  property    Visitor   : TVisitorAbs read FVisitor write FVisitor ;
  property DBConnection: TtiDBConnection read FDBConnection write FDBConnection;
  procedure   Execute( pVisited : TVisitedAbs ) ;
end ;
TVisitorMgr = class( TObject )
private
  FVisMappings : TStringList ;
  FDBConnection : TtiDBConnection ;
public
  constructor create ;
  destructor  destroy ; override ;
  procedure RegisterVisitor(const psGroupName: string ;
    const pClassRef : TVisClassRef ) ;
  procedure Execute(const psGroupName: string; pVisited: TVisitedAbs);
end ;
```

➤ *Above: Listing 11* ➤ *Below: Listing 12*

```
gVisitorMgr.RegisterVisitor( cgsAdrs_Update, TVisAdrsCreate ) ;
gVisitorMgr.RegisterVisitor( cgsAdrs_Update, TVisAdrsUpdate ) ;
gVisitorMgr.RegisterVisitor( cgsAdrs_Update, TVisAdrsDelete ) ;
```

```
procedure TVisitorMgr.Execute(const psGroupName: string;pVisited : TVisitedAbs);
var
  i : integer ;
  lsGroupName : string ;
begin
  lsGroupName := upperCase( psGroupName ) ;
  for i := 0 to FVisMappings.Count - 1 do
    if FVisMappings.Strings[i] = lsGroupName then begin
      TVisMapping( FVisMappings.Objects[i] ).DBConnection := FDBConnection ;
      TVisMapping( FVisMappings.Objects[i] ).Execute( pVisited ) ;
    end ;
end;
```

➤ *Above: Listing 13* ➤ *Below: Listing 14*

```
procedure TVisMapping.Execute( pVisited : TVisitedAbs ) ;
begin
  Assert( pVisited <> nil, 'Visited not asigned' ) ;
  if Visitor = nil then
    Visitor := ClassRef.Create ;
  if Visitor is TVisDBAbs then
    TVisDBAbs( Visitor ).DBConnection := DBConnection ;
  pVisited.Iterate( Visitor ) ;
  if Visitor is TVisDBAbs then
    TVisDBAbs( Visitor ).DBConnection := nil ;
end;
```

read, along with the addresses and e-addresses. This is a great technique for improving performance. Another three visitors manage the saving of data. In all there will be five visitor classes, Listing 15 shows their interface sections.

The implementation of the visitor to read a person's details when a node on the TreeView is clicked is shown in Listing 16. The `AcceptVisitor` function checks that the visited class is a `TPerson`, and then it checks that its `ObjectState` is `posPK` (ie persistent object state, primary key). If `AcceptVisitor` returns true, the other methods are called. `Init` sets the `Query.SQL.Text` property to select a person's details using their OID. `SetupParams` sets the `Query.ParamByName('OID').AsInteger` property to the person's OID. `MapRowToObject` copies the results of the `Query` to the object and `UpdateObjectState` sets the object's `ObjectState` to `posClean`, meaning it has been read from the database and no changes have been made.

These families of visitors are created for each of the persisted classes (`TPerson`, `TAddress` and `TEAddress`) and are then registered with the visitor manager. As you can imagine, this involves a lot of work but can be justified if you are wanting true database independence and blindingly fast performance.

All that remains is to map the BOM into the GUI. This is done using the three custom components: `TtiTreeView`, `TtiTreeViewPlus` and finally `TtiListView`.

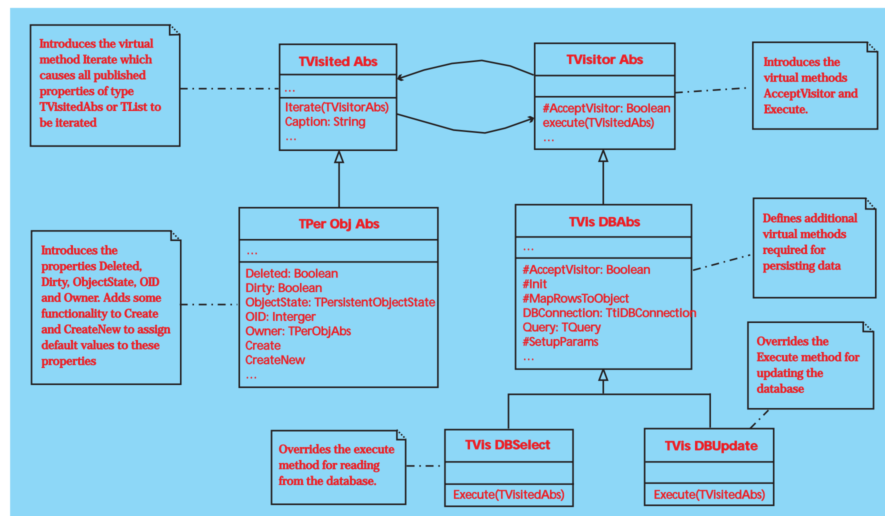
The `TtiTreeView` is a descendant of the `TCustomTreeView` and has an additional property, `Data` that is of type `TPersistent`. The `TtiTreeView` uses RTTI to read the `Data Caption` property to show in the tree, and `List` properties to show as child nodes.

The `TtiTreeViewPlus` adds a panel to the right of the TreeView for displaying the node's data. A family of forms is registered with the `TtiTreeViewPlus`, along with the class reference of the data they will be displaying. This automates the process of displaying a node's data in the right hand panel of the form.

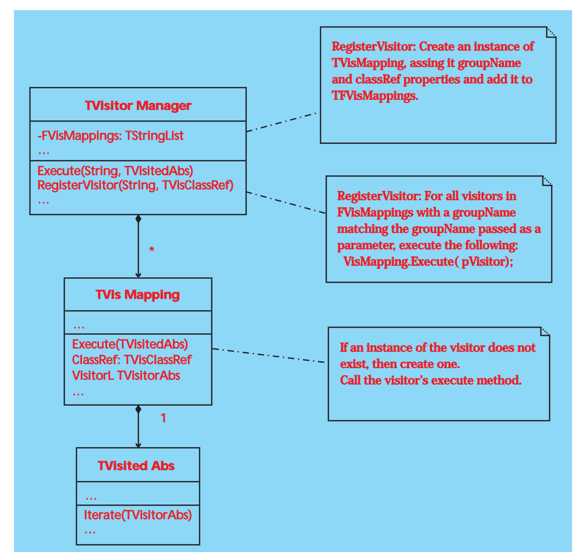
The `TtiListView` is much like the `TtiTreeView`, in that it descends from the `TCustomListView` and has an extra property called `Data` of type `TList`. This must contain a `TList` of `TPersistents` so the ListView can scan for published properties and set up its columns.

TVisitedAbs	Descends from TPersistent and introduces the virtual method Iterate, which causes all published properties of type TVisitedAbs or TList to be iterated.
TPerObjAbs	Descends from TVisitedAbs and adds the OID, Owner, ObjectState, Deleted and Dirty properties along with the constructor CreateNew which will call Create and assign a new, unique OID.
TVisitorAbs	The abstract visitor that contains the virtual methods AcceptVisitor and Execute.
TVisDBAbs	Descends from TVisitorAbs and adds a DBConnection property, along with an instance of TQuery. The Init, SetupParams, MapRowsToObject and UpdateObject virtual methods are also introduced.
TVisDBSelect	Descends from TVisDBAbs and overrides the Execute method to call the methods Init, SetupParams, MapRowsToObject and UpdateObject in turn.
TVisDBUpdate	Descends from TVisDBAbs and overrides the Execute method to call the methods Init, SetupParams, MapRowsToObject and UpdateObject in turn.
TVisitorMgr	Manages a list of TVisMapping(s), which are created when a visitor is registered. Calls a family of visitors and sets their DBConnection property when the Execute method is called.
TVisMapping	Stores a Visitor's class reference, then an instance of the visitor once it has been created. Executes the visitor after assigning its DBConnection property if necessary.

➤ *Table 3*



➤ *Figure 5*



➤ *Figure 6*

The source code for these three components is on the disk and has been compiled into a package called tiPatterns.dpk, which must be installed in the Delphi 5 IDE to view the source of the demo.

Database Independence

Now that we have seen the persistence framework in action, we can revisit the importance of achieving database independence. We only need to replace the visitors if we are switching between SQL databases, or both the visitors and the visitor manager if we are changing the data access API.

The possibilities of persisting the data are almost endless and include: relational database (as shown here), a file-based database (like Access), a custom file format, XML, direct access components (eg IBObjects or DOA), multi-tier access to a remote server, and data access APIs like ADO. I hope to cover these in a future article.

Summary

We have done a lot of work but only managed to replace a very small part of the persistence framework that comes out of the box with Delphi. However, what we have done is overcome the limitations inherent in Delphi's persistent architecture through the use of the visitor and iterator patterns. We have created an elegant way of displaying hierarchical data using a TreeView and ListView, and (most importantly) have created the potential for true database independence. The outcome is a highly optimised, flexible framework that will outperform most conventional client/server applications.

We shall take a look at the `TPersistent` aware TreeView and ListView, along with substituting an alternative persistence layer in a future article. I hope you find this information useful and you are welcome to contact me at the address below if you would like to discuss this article further.

Further Information

The object modelling tool I used to prepare the class diagrams was the highly regarded SimplyObjects, which allows you to draw class and interaction diagrams, forward and reverse engineer Delphi code and generate website documentation too. Visit www.adaptive-arts.com/prod_downloads.htm for an evaluation copy.

The *Melbourne Patterns Group* is a discussion group that meets twice a month to discuss the evolution of the pattern language of programming (PLOP) and the use of design patterns to solve real-life business problems. The address of the *Melbourne Patterns Group* website is www.win32dev.com/patterns. It contains some useful links to other sites that discuss the use of design patterns in software development.

The pattern language *Crossing Chasms* is well worth a read as it gives a clear insight into the problems of mapping an object oriented system into a relational database. Download it from

```
www.ksccary.com/Articles/
  ObjectRDBMDPatterns/
   ObjectRDBMSPattern.htm
```

A useful source of information on OO to relational database persistence is at www.ambysoft.com

Reference

Design Patterns: Elements of Reusable Object-Oriented Software. Gamma, Helm, Johnson, Vlissides. Addison Wesley, 1995.

Peter Hinrichsen is a Certified Inprise Consultant and Director of TechInsite P/L in Melbourne, Australia, specialising in client/server and multi-tier OO development in Delphi. Email Peter at peter_hinrichsen@ techinsite.com.au
© TechInsite Pty Ltd 2000

```
Interface
Type
  // Each of these classes implement the following methods,
  // but as they are the same across classes, they are not shown here.
  //protected
  //  function    AcceptVisitor : boolean ; override ;
  //  procedure   Init                     ; override ;
  //  procedure   SetupParams              ; override ;
  //  procedure   MapRowsToObject          ; override ;
  TVisPersonRead_PK = class( TVisDBSelect )
  TVisPersonRead_Detail = class( TVisDBSelect )
  TVisPersonUpdate = class( TVisDBUpdate )
  TVisPersonDelete = class( TVisDBUpdate )
  TVisPersonCreate = class( TVisDBUpdate )
// The classes are registered with the visitor manager
// in the initialazation section.
//.The registration order is important. For example,
// child classes must be deleted before a parent class,
// and parent classes must be created before a child
// class.
initialization
  gVisitorCache.RegisterVisitor( cgsAdrs_Read_PK, TVisPersonRead_PK ) ;
  gVisitorCache.RegisterVisitor( cgsAdrs_Read_Detail, TVisPersonRead_Detail ) ;
  gVisitorCache.RegisterVisitor( cgsAdrs_Update, TVisPersonCreate ) ;
  gVisitorCache.RegisterVisitor( cgsAdrs_Update, TVisPersonUpdate ) ;
  gVisitorCache.RegisterVisitor( cgsAdrs_Update, TVisPersonDelete ) ;
```

➤ *Above: Listing 15*

➤ *Below: Listing 16*

```
function TVisPersonRead_Detail.AcceptVisitor: boolean;
begin
  result := (Visited is TPerson) and (TPerObjAbs(Visited).ObjectState = posPK);
end;
procedure TVisPersonRead_Detail.Init;
begin
  Query.SQL.Text := 'select        ' + '  title        ' + ' ,initials    ' +
    ' ,notes        ' + 'from         ' + '  person       ' + 'where          ' +
    '  oid = :oid  ' ;
end;
procedure TVisPersonRead_Detail.SetupParams;
begin
  Query.ParamByName('OID').AsInteger := TPerson(Visited).OID ;
end ;
procedure TVisPersonRead_Detail.MapRowsToObject;
begin
  with Visited as TPerson do begin
    Title    := Query.FieldByName( 'Title' ).AsString ;
    Initials := Query.FieldByName( 'Initials' ).AsString ;
    Notes    := Query.FieldByName( 'Notes' ).AsString ;
  end ;
end;
procedure TVisPersonRead_Detail.MapRowsToObject;
begin
  ObjectState := posClean ;
End ;
```